



RSO: Sistemi Operativi

spitfire

A.A. 2023-2024

Contents

1	Struttura e servizi	4
1.1	Componenti di un sistema di elaborazione	4
1.2	Requisiti per i sistemi operativi	5
1.3	La maledizione della generalità	5
1.4	Struttura dei sistemi operativi	6
1.5	Servizi offerti da un sistema operativo	7
1.6	Chiamate di sistema e Application Programming Interfaces	7
1.7	Programmi di sistema	8
1.7.1	Intefaccia utente: l'interprete dei comandi	9
1.7.2	Interfaccia utente: le interfacce grafiche	10
1.7.3	Intefaccia utente: Le interfacce touch-screen	11
1.8	L'implementazione dei programmi di sistema	11
2	Processi e thread: i servizi	12
2.1	Programmi e processi	12
2.2	Struttura di un processo	12
2.2.1	L'immagine di un processo	12
2.3	Operazioni sui processi	13
2.3.1	Creazione di processi	13
2.3.2	Terminazione di processi	14
2.4	API POSIX per le operazioni sui processi	14
2.5	Processi zombie e orfani	16
2.6	Comunicazione interprocesso	16
2.6.1	Modelli di IPC	16
2.6.2	IPC tramite memoria condivisa	16
2.6.3	IPC tramite message passing	17
2.6.4	Pipe	17
2.6.5	Notifiche con callback	18
2.7	API POSIX per l'IPC	18
2.7.1	Memoria condivisa in POSIX	18
2.7.2	Pipe anonime in POSIX	20
2.7.3	Named Pipes in POSIX	21
2.7.4	Segnali in POSIX	23
2.8	Multithreading	24
2.9	POSIX pthreads	25
2.9.1	Comportamento rispetto alle chiamate di sistema fork() ed exec()	26
2.9.2	Gestione dei segnali	26
2.9.3	Cancellazione dei thread	26
2.9.4	Dati locali dei thread	27
3	Gestione della memoria: i servizi	27
3.1	Lo spazio di indirizzamento	28
3.2	Associazione degli indirizzi	28
3.2.1	Loader e Linker	28
3.2.2	Librerie dinamiche	29

3.2.3	Varianti nell'associazione degli indirizzi	29
3.3	Spazio di indirizzamento virtuale	30
3.3.1	Librerie dinamiche	31
3.3.2	Memory mapping	31
3.4	Le API POSIX per la gestione della memoria	32
4	File system: i servizi	33
4.1	Operazioni dei processi sui file	33
4.2	Lock dei file	34
4.3	Tipi di file	34
4.4	Struttura dei file	35
4.5	Metodi di accesso a file	35
4.5.1	Accesso sequenziale	35
4.5.2	Accesso diretto	35
4.5.3	Accesso indicizzato	36
4.6	Directories	36
4.6.1	Struttura delle directory ad un livello	36
4.6.2	Struttura delle directory a due livelli	37
4.6.3	Struttura delle directory ad albero	37
4.6.4	Struttura delle directory a grafo aciclico	38
4.6.5	Struttura delle directory a grafo generico	38
4.7	Protezione	39
4.7.1	Liste di controllo degli accessi	39
4.8	Volumi e montaggio	40
4.9	API POSIX per operazioni sui file	41
4.9.1	API POSIX per i lock	42
4.10	API POSIX per operazioni su directory	42
4.11	API POSIX per la protezione	44
5	Interfaccia e struttura del kernel	44
5.1	Uso delle librerie dinamiche per le API	46
5.2	La scarsa portabilità degli eseguibili binari	46
5.3	Struttura del Kernel	47
5.3.1	Struttura monolitica	48
5.3.2	Struttura a strati	48
5.3.3	Struttura a microkernel	49
5.3.4	Struttura a moduli	50
5.3.5	Sistemi ibridi	50

1 Struttura e servizi

Cosa sappiamo sui sistemi operativi? Sappiamo che, per esempio, i principali sono **linux, Windows e MacOS**; che il sistema operativo è il **primo programma che viene eseguito dopo il boot**. Di solito un sistema operativo fornisce un **ambiente desktop a finestre e ci permette di installare nuove applicazioni**. Ci permette inoltre di eseguire tante applicazioni **contemporaneamente**, anche più dei **core dei processori**. Inoltre, esso **mantiene e organizza i nostri dati sotto forma di file e cartelle**. Quindi, cos'è un **sistema operativo**? Esso è:

- Un insieme di **programmi** (Software)
- Che gestiscono **gli elementi fisici di un computer** (Hardware)

E a cosa serve un sistema operativo?

- Fornire una **piattaforma di sviluppo per le applicazioni**, che permette loro di **condividere e astrarre** le risorse HW.
- Agisce da **intermediario** tra utenti e computer, permettendo agli utenti di **controllare l'esecuzione dei programmi applicativi** e l'assegnazione delle risorse HW ad essi
- **Protegge le risorse degli utenti** (e dei loro programmi) dagli altri utenti (e dai loro programmi) e da eventuali **attori esterni**

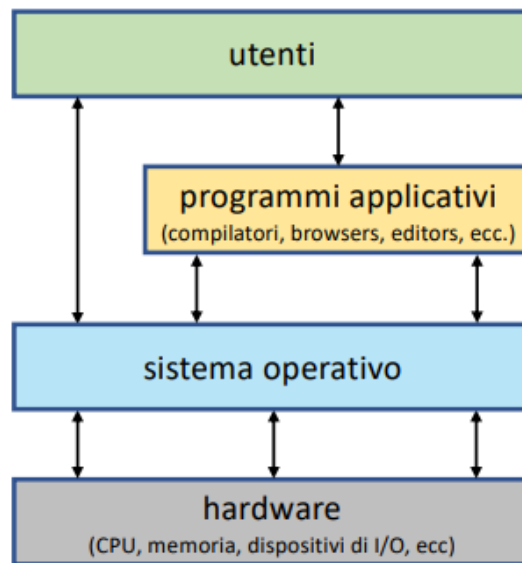
Un sistema operativo è quindi in primo luogo una **piattaforma di sviluppo**, ossia un insieme di funzionalità software che i programmi applicativi possono usare. Tali funzionalità permettono ai programmi di poter usare in maniera conveniente le risorse hardware di condividerle:

- Da un lato il sistema operativo **astrae** le risorse hardware, presentando agli sviluppatori di 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.

1.1 Componenti di un sistema di elaborazione

Le componenti di un sistema di elaborazione sono:

- **Utenti**: Persone, macchine, altri computer, ecc...
- **Programmi applicativi**: Risolvono i problemi di calcolo degli utenti
- **Sistema operativo**: Coordina e controlla l'uso delle risorse hardware
- **Hardware**: Risorse di calcolo (CPU, periferiche, memoria di massa, ...)



1.2 Requisiti per i sistemi operativi

Oggigiorno i computer sono ovunque: vi sono molteplici tipologie di computer utilizzati in scenari applicativi molto diversi. In quasi tutti i tipi di computer si tende ad installare un sistema operativo allo scopo di gestire l'hardware e semplificare la programmazione. Ma ogni scenario applicativo in cui viene usato un computer richiede che il sistema operativo che vi viene installato abbia caratteristiche ben determinate. Che cosa si richiede quindi ad un sistema operativo per supportare uno determinato scenario applicativo? Vediamo qualche scenario:

- **Server e Mainframe:** massimizzare le performance, rendere equa la condivisione delle risorse tra molti utenti
- **Laptop, PC e tablet:** massimizzare la facilità d'uso e la produttività della singola persona 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)

1.3 La maledizione della generalità

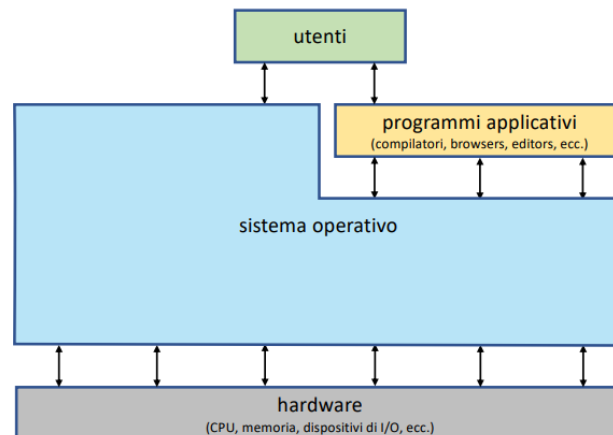
Nella storia (ed anche oggi) alcuni sistemi operativi sono stati utilizzati per scenari applicativi diversi. Ad esempio, Linux è usato oggi nei server, nei computer desktop e nei dispositivi mobili (come parte di Android). La **maledizione della generalità** afferma che, se un sistema operativo deve supportare un insieme di scenari applicativi troppo ampio, non sarà in grado di supportarne nessuno particolarmente bene. Esempio di questo si è visto con **OS/360**, il primo sistema operativo che doveva supportare una famiglia di computer diversi (la linea 360 IBM).

1.4 Struttura dei sistemi operativi

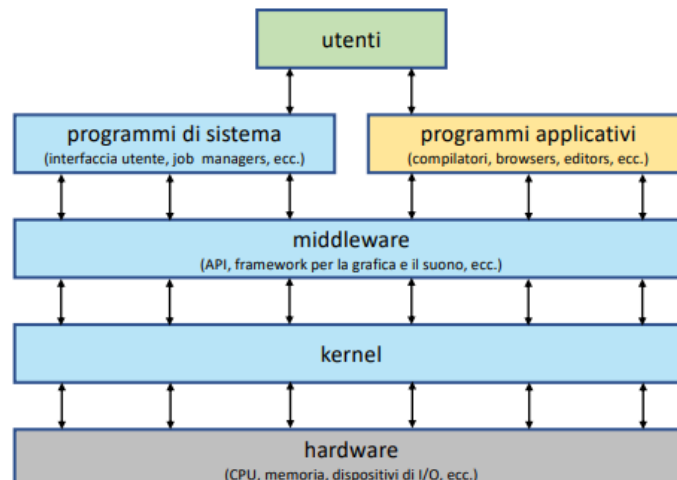
Non c'è una definizione universalmente accettata di quali programmi fanno parte di un sistema operativo. In generale però un sistema operativo almeno comprende:

- **Kernel:** Il "programma sempre presente" che si "impadronisce" dell'HW, 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 processi e jobs, UI, ...)

Alcuni sistemi operativi forniscono "out-of-the-box" anche dei **programmi applicativi** (editor, fogli di calcolo,...) ma non li considereremo come parti del sistema operativo. Data questa lista di componenti, possiamo rivisitare le **componenti di un sistema di elaborazione**:



Che visti in dettaglio diventano:

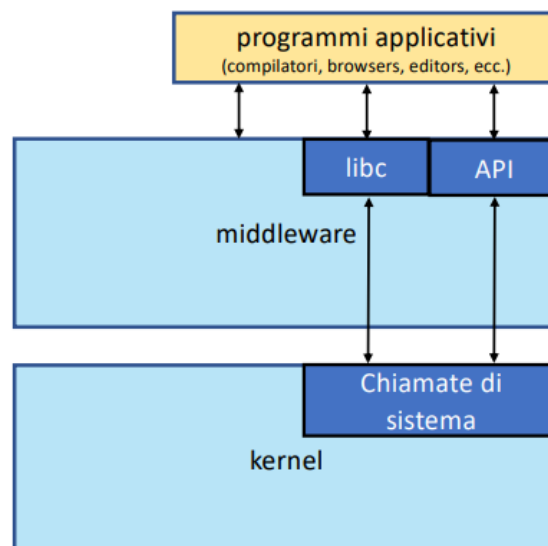


1.5 Servizi offerti da un sistema operativo

I principali servizi che un sistema operativo offre 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 errorea)
- **Gestione dei file:** questi servizi permettono di leggere, scrivere e manipolare files e directories
- **Gestione dispositivi:** questi servizi permettono ai programmi di effettuare operazioni di input/output, ad esempio leggere da/scrivere su un terminale
- **Comunicazione interprocesso:** 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 0); 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

1.6 Chiamate di sistema e Application Programming Interfaces



Il kernel offre i propri servizi ai programmi come **chiamate di sistema** (syscalls), ossia funzioni invocabili in un determinato linguaggio di programmazione (C, C++, ...). I programmi però non utilizzano direttamente le chiamate di sistema, ma delle librerie di middleware dette **Application Programming Interface** (API) implementate invocando le chiamate di sistema. Spesso le API sono fortemente legate con le librerie standard del linguaggio di implementazione (es. libc se le API sono implementate in C) al punto che anche queste diventano parte implicita dell'API. Bisogna ricordare che:

- Le API sono **esposte dal middleware**, mentre le chiamate di sistema **dal kernel**
- Le API usano le chiamate di sistema nella loro implementazione
- Le API sono standardizzate (es. POSIX, Win32), le chiamate di sistema no, quindi ogni kernel ha chiamate di sistema differenti
- Le API sono stabili, le chiamate di sistema possono variare al variare della versione del sistema operativo
- Le API offrono funzionalità più ad alto livello e più semplici da usare, le chiamate di sistema offrono funzionalità più elementari e più complesse da usare

EXAMPLE OF STANDARD API

As an example of a standard API, consider the `read()` function that is available in UNIX and Linux systems. The API for this function is obtained from the man page by invoking the command

```
man read
```

on the command line. A description of this API appears below:

```
#include <unistd.h>

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

ssize_t	read	(int fd, void *buf, size_t count)
return value	function name	parameters

A program that uses the `read()` function must include the `unistd.h` header file, as this file defines the `ssize_t` and `size_t` data types (among other things). The parameters passed to `read()` are as follows:

- `int fd`—the file descriptor to be read
- `void *buf`—a buffer into which the data will be read
- `size_t count`—the maximum number of bytes to be read into the buffer

On a successful read, the number of bytes read is returned. A return value of 0 indicates end of file. If an error occurs, `read()` returns `-1`.

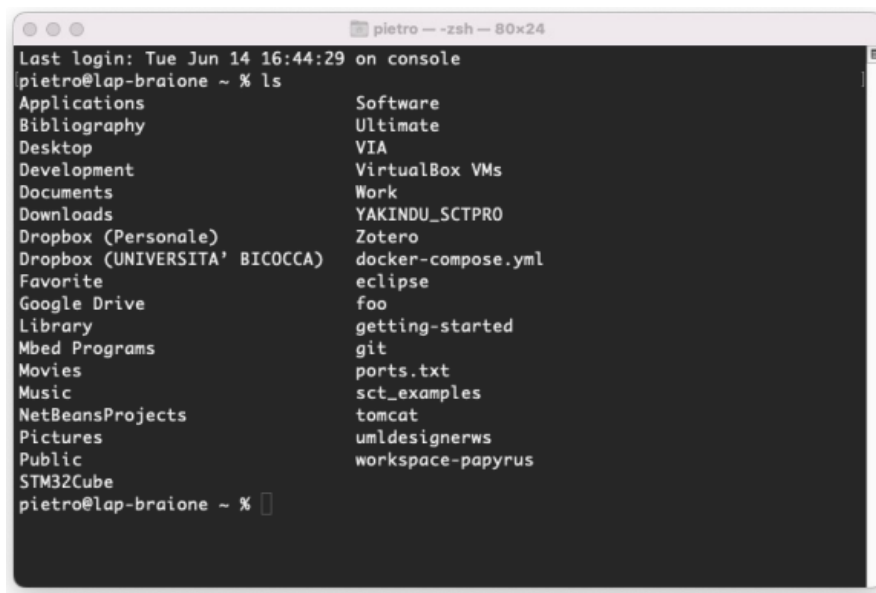
1.7 Programmi di sistema

La maggior parte degli utenti utilizza servizi del sistema operativo attraverso i programmi di sistema. 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. Vi sono diversi tipi di programmi di sistema:

- **Interfacce 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 di file e directories
- **Modifica dei file:** editor di testo, programmi per la manipolazione del contenuto dei file (Emacs)
- **Visualizzazione e modifica informazioni di stato:** data, ora, memoria disponibile, processi, utenti, ... fino a informazioni complesse su prestazione, 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 e debugger
- **Ambienti di supporto alla programmazione:** compilatori, assembleri, debugger, interpreti per diversi linguaggi di programmazione
- **Comunicazione:** forniscono i meccanismi per creare connessione 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 i file, ecc...
- **Servizi di 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, ...

1.7.1 Intefaccia utente: l'interpete dei comandi



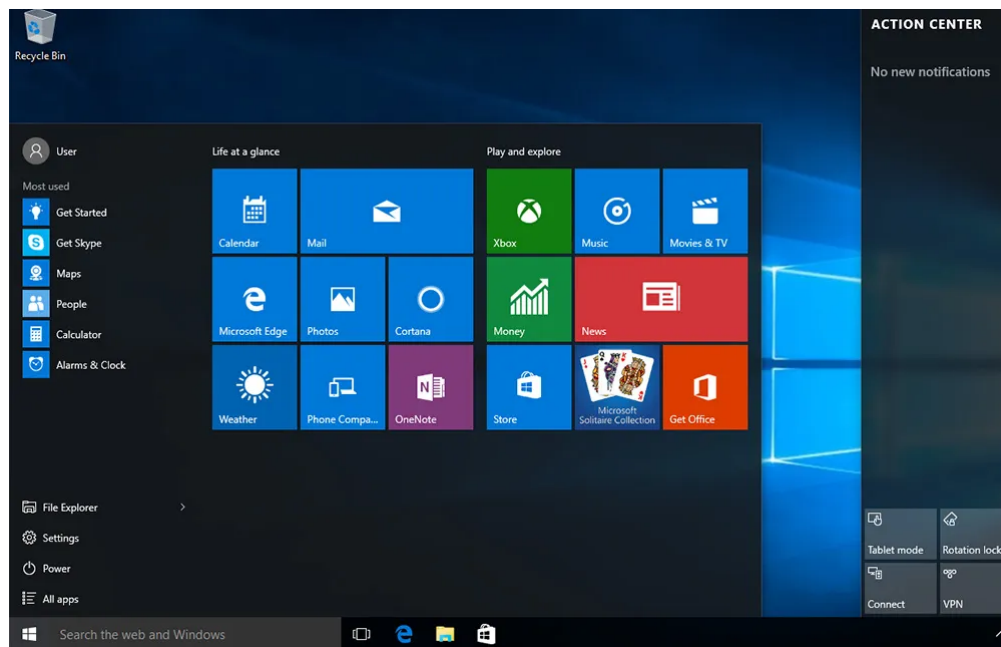
```
pietro -- -zsh -- 80x24
Last login: Tue Jun 14 16:44:29 on console
pietro@lap-braione ~ % ls
Applications          Software
Bibliography          Ultimate
Desktop              VIA
Development           VirtualBox VMs
Documents             Work
Downloads             YAKINDU_SCTPRO
Dropbox (Personale)   Zotero
Dropbox (UNIVERSITA' BICOCCA) docker-compose.yml
Favorite              eclipse
Google Drive          foo
Library               getting-started
Mbed Programs         git
Movies                ports.txt
Music                 sct_examples
NetBeansProjects      tomcat
Pictures              umldesignerws
Public                workspace-papyrus
STM32Cube
pietro@lap-braione ~ %
```

L'interprete dei comandi permette agli utenti di impartire in maniera testuale delle istruzioni al sistema operativo. In molti sistemi operativi è possibile configurare quale interprete dei comandi usare, nel qual caso è detto **shell**. Ci sono due modi per implementare un comando:

- **Built-in:** l'interprete esegue direttamente il comando (tipico dell'interprete dei comandi di Windows)
- **Come programma di sistema:** l'interprete manda in esecuzione un programma (tipico delle shell Unix e Unix-Like)

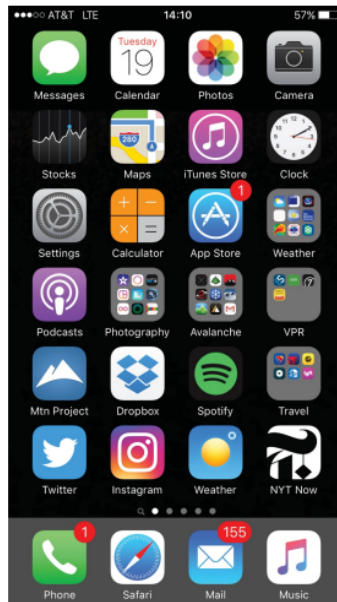
Spesso l'interprete riconosce **un vero e proprio linguaggio di programmazione** (es. Bash).

1.7.2 Interfaccia utente: le interfacce grafiche



Le interfacce grafiche(GUI) sono di solito basate sulla metafora della scrivania, delle icone e delle cartelle (corrispondenti alle directory). Nate dalla ricerca presso lo Xerox PARC lab negli anni 70, vennero popolarizzate dai computer Apple Macintosh negli anni 80. Su Linux le più popolari sono KDE e Gnome.

1.7.3 Intefaccia utente: Le interfacce touch-screen



I dispositivi mobili richiedono interfacce di nuovo tipo. Esse non prevedono nessun dispositivo di puntamento (mouse); sostituendolo con l'uso dei gesti (gestures). Inoltre esse possono offrire servizi come tastiere virtuali e comandi vocali.

1.8 L'implementazione dei programmi di sistema

- **Apri** *in.txt* in lettura
- Se non esiste
 - **Scrivi** un messaggio di errore su terminale
 - **Termina** il programma con codice errore
- **Apri** *out.txt* in scrittura
- Se non esiste, **crea** *out.txt*
- Loop
 - **Leggi** da *in.txt*
 - **Scrivi** su *out.txt*
- End loop
- **Chiudi** *in.txt*
- **Chiudi** *out.txt*
- **Termina** normalmente

I programmi di sistema sono implementati utilizzando le API, esattamente come i programmi applicativi. Consideriamo ad esempio il comando *cp* delle shell dei sistemi operativi Unix-like; la sua sintassi è:

cp in.txt out.txt

Esso copia il contenuto del file *in.txt* in un file *out.txt*. Se il file *out.txt* esiste, il contenuto precedente viene cancellato, altrimenti *out.txt* viene creato. L'immagine sopra rappresenta una possibile struttura del codice; le invocazioni delle API sono riportate in grassetto. *cp* è implementato come programma di sistema.

2 Processi e thread: i servizi

Un sistema operativo esegue un certo numero di programmi sullo stesso sistema di elaborazione. Il numero di programmi da eseguire può essere arbitrariamente elevato, di solito è infatti molto maggiore del numero di CPU del sistema. A tale scopo, il sistema operativo realizza e mette a disposizione un'astrazione detta **processo**. Un processo è quindi un'entità attiva astratta definita dal sistema operativo allo scopo di eseguire un programma. Per il momento, assumiamo che l'esecuzione di un processo sia sequenziale, tuttavia **rilasseremo presto questa assunzione**.

2.1 Programmi e processi

È fondamentale notare la differenza tra programma e processo!

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

Uno stesso programma può dare origine a **diversi processi**:

- Diversi utenti eseguono lo stesso programma
- Uno stesso programma viene eseguito più volte, anche contemporaneamente, dallo stesso utente

2.2 Struttura di un processo

Un processo è composto da diverse parti:

- Lo stato dei registri del processore che esegue il programma, incluso il PC
- 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, locks, ...)
- Più diverse informazioni sullo stato del processo per il sistema operativo

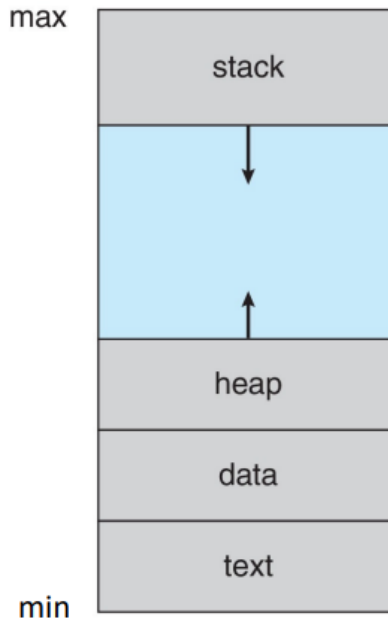
Notare che processi distinti hanno immagini distinte! Due processi operano su zone di memoria centrale separate! Le risorse del sistema operativo invece possono essere condivise tra processi (a seconda del tipo di risorsa)

2.2.1 L'immagine di un processo

L'intervallo di indirizzi di memoria in cui è contenuta l'immagine di un processo è anche detto **spazio di indirizzamento (address space)** del processo. L'immagine di un processo di norma contiene:

- **Text section:** contiene il codice macchina del programma

- **Data section:** contiene le variabili globali
- **Heap:** contiene la memoria allocata dinamicamente durante l'esecuzione
- **Stack delle chiamate:** contiene parametri, variabili locali e indirizzo di ritorno delle varie procedure che vengono invocate durante l'esecuzione del programma



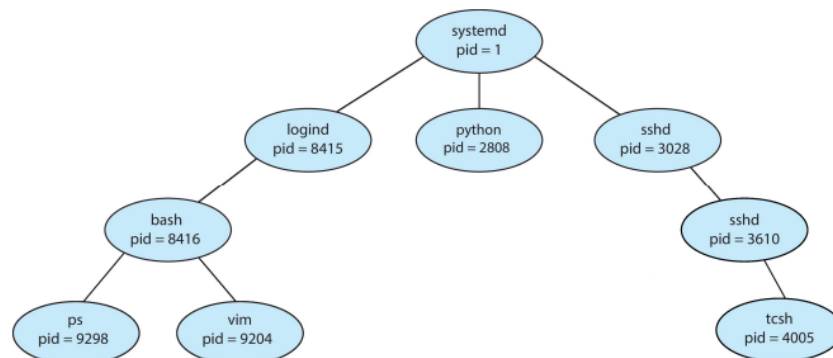
Text e data section hanno dimensioni costanti per tutta la vita del processo; mentre stack e heap invece crescono e decrescono durante la vita del processo.

2.3 Operazioni sui processi

I sistemi operativi di solito forniscono delle chiamate di sistema con le quali un processo può creare, terminare e manipolare altri processi. Dal momento che solo un processo può creare un altro processo, all'avvio il sistema operativo crea dei processi **primordiali** dai quali tutti i processi utente e di sistema vengono progressivamente creati.

2.3.1 Creazione di processi

Di solito nei sistemi operativi i processi sono organizzati in maniera **gerarchica**: Un processo **padre** può creare altri processi **figli** che a loro volta potranno essere i padri di nuovi processi. Ciò va a creare un **albero di processi**



La relazione padre/figlio è di norma importante per le politiche di condivisione delle risorse e di coordinazione tra processi. Vi sono diverse **politiche di condivisione di risorse**:

- Padre e figlio condividono **tutte le risorse**...
- ...o un **opportuno sottoinsieme**...
- ...o **nessuna risorsa**

Quando un processo padre crea un processo figlio, esso può adottare diverse **politiche di creazione di spazio di indirizzi**:

- Il figlio è un **duplicato** del padre (stessa memoria e programma1)...
- oppure no, e bisogna **specificare quale programma deve eseguire il figlio**

I processi padri e i loro figli possono inoltre **coordinarsi fra loro** seguendo delle **politiche di coordinazione padre/figli**:

- Il padre è **sospeso** finché i figli non terminano...
- oppure eseguono in maniera **concorrente**

2.3.2 Terminazione di processi

I processi di regola richiedono esplicitamente la propria terminazione al sistema operativo. Un processo padre può attendere o meno la terminazione di un figlio oppure la **può forzare** per una serie di ragioni:

- Il figlio sta usando **risorse in eccesso** (tempo, memoria, ...)
- Le funzionalità del figlio **non sono più richieste** (in questo caso, tuttavia, è meglio terminarlo in maniera ordinata tramite IPC)
- Il padre **termina prima del figlio** (in alcuni S.O.)

Riguardo all'ultimo punto, alcuni sistemi operativi **non permettono che i processi figli esistano se il loro processo padre è terminato**:

- **Terminazione a cascata**: anche i nipoti, pronipoti, ecc... devono essere terminati
- La terminazione viene iniziata dal **sistema operativo**

2.4 API POSIX per le operazioni sui processi

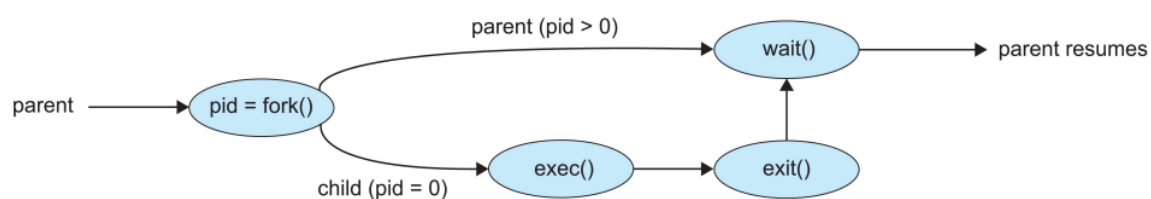
POSIX (Portable Operating System Interface for Unix) è una famiglia di standard specificata dalla **IEEE** per mantenere la compatibilità del software tra diversi sistemi operativi, in particolare tra le varianti di Unix (Linux e MacOS per esempio). In particolare definisce l'API disponibile (come la libreria **C POSIX** per il linguaggio C) e l'interfaccia a linea di comando utilizzabile per shell (come **bash** e **dash**) e altri comandi fondamentali. Un sistema operativo che segue gli standard POSIX si dice **POSIX-Compliant**. Per operare sui processi, POSIX definisce le seguenti API:

- **fork()**: Crea un nuovo processo figlio; il figlio è un duplicato del padre ed esegue concorrentemente ad esso; 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 usato dopo una **fork()** dal figlio per iniziare ad eseguire un programma diverso da quello del padre. **exec()** tuttavia definisce un'intera **famiglia di funzioni** in POSIX, ognuna distinta da dei **suffixi** di cui ogni lettera ha un significato particolare:
 - **e**: Un **array di puntatori a variabili d'ambiente** è passato esplicitamente alla nuova immagine del processo
 - **l**: Gli argomenti passati **da linea di comando** sono passati individualmente (come lista) alla funzione
 - **p**: Utilizza la variabile d'ambiente **PATH** per trovare il file nominato nell'argomento "file" per eseguirlo
 - **v**: Gli argomenti passati **da linea di comando** sono passati come un array di puntatori

Tendenzialmente, gli argomenti di una **exec()** sono:

- Il **path** del programma da eseguire oppure il **file descriptor** (fd) del file da eseguire
- Gli **argomenti** da passare all'entry point del programma da eseguire
- **wait()**: viene chiamata dal padre per attendere la fine dell'esecuzione di un figlio; ritorna:
 - il **PID** del figlio che è terminato
 - Il codice di ritorno del figlio (passato come parametro dal figlio in **exit()**)
- **exit()**: fa terminare il processo che la invoca:
 - Accetta come parametro un codice di ritorno numerico
 - Il sistema operativo elimina il processo e recupera le sue risorse
 - Quindi restituisce al processo padre il codice di ritorno (se ha invocato **wait()**, altrimenti lo memorizza per quando lo invocherà)
 - Viene implicitamente invocata se il processo esce dalla funzione **main**
- **abort()**: fa terminare forzatamente un processo figlio

Tendenzialmente la tipica sequenza di **fork-exec** è rappresentabile come segue:



2.5 Processi zombie e orfani

Se un processo termina ma il suo padre non lo sta aspettando (cioè non ha invocato `wait()`), il processo è detto essere **zombie**: le sue risorse non possono essere completamente deallocate (il padre potrebbe prima o poi invocare `wait()`). Se un processo padre termina prima di un suo figlio e **non vi è terminazione a cascata** allora i suoi processi figli si dicono **orfani**.

2.6 Comunicazione interprocesso

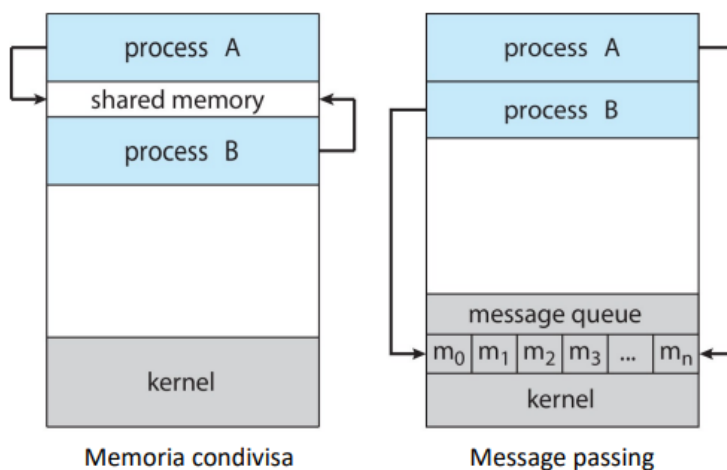
Più processi possono essere indipendenti o cooperare. Un processo coopera con uno o più altri processi se il suo comportamento "influenza" o "è influenzato da" il comportamento di questi ultimi. Vi sono più motivi per il quale si vogliono avere processi cooperanti:

- **Condivisione** di informazioni
- **Accelerazione** di computazioni
- **Modularità e isolamento** (come in Chrome)

Per permettere ai processi di cooperare il sistema operativo deve mettere a disposizione primitive di **comunicazione interprocesso** (IPC). Vi sono due tipi di primitive:

- **Memoria condivisa**
- **Message passing**

2.6.1 Modelli di IPC



2.6.2 IPC tramite memoria condivisa

Nella IPC tramite memoria condivisa viene stabilita una zona di memoria condivisa tra i processi che intendono comunicare. La comunicazione è **controllata dai processi che comunicano** e non dal sistema operativo. Un problema importante è permettere ai processi che comunicano tramite memoria condivisa di **sincronizzarsi** (un processo non deve leggere la memoria condivisa mentre l'altro sta scrivendo). Allo scopo, i sistemi operativi mettono a disposizione ulteriori primitive per la **sincronizzazione**.

2.6.3 IPC tramite message passing

Nell'IPC tramite message passing, si permette ai processi **sia di comunicare che di sincronizzarsi**. I processi comunicano tra di loro **senza condividere memoria** attraverso la mediazione del sistema operativo. Questo mette a disposizione:

- Un'operazione **send(message)** con la quale un processo può inviare un messaggio ad un altro processo
- Un'operazione **receive(message)** con la quale un processo può ricevere un messaggio o mettersi in attesa fino a quando non ne riceve uno.

Per comunicare fra loro, due processi devono:

- Stabilire un **link di comunicazione** tra di loro
- Scambiarsi **messaggi** usando *send* e *receive*

2.6.4 Pipe

Le **pipe** sono canali di comunicazione tra i processi e sono una forma di IPC tramite **message passing**. Ve ne sono di vario tipo:

- **Unidirezionale**
- **Bidirezionale**
 - **Half-Duplex**
 - **Full-Duplex**
- **Relazione** tra i processi comunicanti (sono padre-figlio oppure no)
- Usabili o meno in **rete**

Convenzionalmente, le pipe sono:

- **Unidirezionali**
- Non accessibili al di fuori del processo creatore; sono quindi di solito **condivise** con un processo figlio attraverso una **fork()**
- In Windows sono chiamate **pipe anonime**

Vi sono anche le **named pipes**:

- **Bidirezionali**
- Esistono anche **dopo la terminazione** del processo che le ha create
- Non richiedono una relazione padre-figlio tra i processi che la usano

In **Unix**, le named pipes sono:

- Half-duplex

- Sono accessibili solo sulla stessa macchina
- Trasportano solo dati byte-oriented

In **Windows** invece, le named pipes sono:

- Full-duplex
- Sono accessibili anche da macchine diverse
- Trasportano anche dati message-oriented

2.6.5 Notifiche con callback

In alcuni sistemi operativi (es. API POSIX e Win32) un processo può **notificare** un altro processo in maniera da causare **l'esecuzione di un blocco di codice** ("callback"), similmente a ciò che avviene durante un **interrupt**. Nei sistemi Unix-like (POSIX, Linux) tale notifiche vengono dette **segnali** ed interrompono in maniera **asincrona** la computazione del processo corrente, causando un **salto brusco alla callback di gestione**, al termine della quale la computazione **ritorna al punto di interruzione**. 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.

2.7 API POSIX per l'IPC

2.7.1 Memoria condivisa in POSIX

Un processo crea o apre un segmento di memoria condivisa con la funzione *shm_open*:

```
int shm_fd = shm_open(const char *name, int oflag, mode_t mode);
```

dove:

- Il parametro **oflag** può avere i seguenti valori:
 - **O_RDONLY**: Apertura in sola lettura
 - **O_RDWR**: Apertura per lettura o scrittura
 - **O_CREAT**: Crea lo spazio di memoria condivisa
 - **O_EXCL**: Se **O_EXCL** e **O_CREAT** sono settate; allora *shm_open* fallisce se l'oggetto di memoria condivisa con quel nome esiste già. Se invece Se **O_EXCL** è settata ma non Se **O_CREAT** allora il risultato è **indefinito**
 - **O_TRUNC**: Se l'oggetto esiste ed è stato aperto con successo tramite **O_RDWR**, allora l'oggetto sarà troncato a lunghezza 0.
 - Si possono utilizzare **O_RDONLY** e **O_RDWR** in combinazione con le altre flag, ma non insieme.

- Il parametro *mode* indica invece la **modalità di accesso** con cui si sta accedendo all'oggetto di memoria condivisa. La modalità è quindi con quali **permessi** si accede all'oggetto ed essi sono rappresentati da una **maschera di bit** in base 8 (da 0 a 7). In UNIX, i permessi sono rappresentati tramite **3 triadi**, che rappresentano rispettivamente quali permessi sul file hanno **l'utente, i membri del gruppo e gli utenti "esterni al gruppo"**. Diamo una tabella di quelli più comuni:

Symbolic notation	Numeric notation	English
-----	0000	no permissions
-rwx-----	0700	read, write, & execute only for owner
-rwxrwx---	0770	read, write, & execute for owner and group
-rwxrwxrwx	0777	read, write, & execute for owner, group and others
---x--x--x	0111	execute
--w--w--w-	0222	write
--wx-wx-wx	0333	write & execute
-r--r--r--	0444	read
-r-xr-xr-x	0555	read & execute
-rw-rw-rw-	0666	read & write
-rwxr-----	0740	owner can read, write, & execute; group can only read; others have no permissions

Un esempio di chiamata è:

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

Quando si crea un nuovo segmento, è necessario **impostarne la dimensione**; ciò viene fatto tramite la funzione **ftruncate** che ha firma:

```
int ftruncate(int fildes, off_t length)
```

Dove **fildes**, nel nostro caso, è il **file descriptor** del segmento di memoria e **length** sarà la dimensione che gli vogliamo dare (in byte). Infine, la funzione **mmap** mappa la memoria condivisa nello spazio:

```
void* mmap(void* addr, size_t len, int prot, int flags, int fildes, off_t off);
```

dove:

- Il parametro **addr** è l'indirizzo di partenza dello spazio di memoria condivisa. Questo parametro viene in verità usato come **SUGGERIMENTO** da parte del kernel del sistema operativo; ciò accade perché lo spazio di indirizzi indicato da **addr** e **len** **potrebbe essere già allocato**.
- Il parametro **length** specifica la lunghezza (in byte) dello spazio di memoria allocato
- Il parametro **prot** indica quali operazioni sono permesse sulla regione di memoria; esso può assumere:

- **PROT_READ**: I dati possono essere letti
 - **PROT_WRITE**: I dati possono essere scritti
 - **PROT_EXEC**: I dati possono essere eseguiti
 - **PROT_NONE**: I dati non possono essere acceduti
- Il parametro **flag** fornisce ulteriori informazioni riguardo la gestione dei dati mappati sullo spazio di memoria. Esso può assumere:
 - **MAP_SHARED**: I cambiamenti sono condivisi
 - **MAP_PRIVATE**: I cambiamenti sono privati
 - **MAP_FIXED**: Quando questa flag è impostata, la regione di memoria allocata **parte esattamente da addr** al posto di usarlo come un suggerimento.
 - Il parametro **fdes** indica il file (o l'oggetto) che rappresenta lo spazio di memoria condivisa
 - Il parametro **off** indica l'offset nel file da dove inizia lo spazio di memoria condivisa

Un'esempio di chiamata può essere:

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

Da questo momento si può quindi usare il puntatore `shm_ptr` ritornato da `mmap` per leggere/scrivere la memoria condivisa.

2.7.2 Pipe anonime in POSIX

Le pipe anonime in POSIX vengono create con la funzione *pipe*, che ritorna due descrittori, uno per punto di lettura e uno per il punto di scrittura. Ipotizziamo quindi di avere un array *int pd_fd[2]*; la funzione *pipe* ha la seguente firma:

$$pipe(int fildes[2])$$

Con *fildes[2]* l'array in cui verranno scritti i descrittori di lettura e scrittura (lettura nella cella 0 e scrittura nella cella 1). La funzione ritorna 0 se ha avuto successo, -1 altrimenti; in questo caso verrà impostata la flag *ERRNO* per indicare l'errore e i descrittori NON verranno allocati all'interno di *fildes[2]*. Un esempio di chiamata di *pipe* è:

$$int res = pipe(p_fd)$$

Le funzioni *read* e *write* permettono rispettivamente di leggere e scrivere:
La funzione *write* ha la seguente firma:

$$ssize_t write (int fildes, const void* buf, size_t nbyte)$$

Essa cercherà di scrivere *nbyte* dal buffer puntato da *buf* al file associato al file descriptor *fildes*. La funzione ritorna il numero di byte effettivamente scritto, altrimenti -1. Una scrittura sulla pipe anonima d'esempio può essere quindi:

```
ssize_t n_wr = write(p_fd[1], "Hello World!", 14);
```

La funzione *read* ha la seguente firma:

```
ssize_t read(int fildes, void* buf, size_t nbyte);
```

La funzione cercherà di leggere *nbyte* dal file associato al file descriptor *fildes* e di porre ciò che ha letto nel buffer puntato da *buf*. La funzione restituirà il numero di byte letti se ha avuto successo, -1 altrimenti. Un esempio di lettura dalla pipe anonima d'esempio può essere quindi:

```
char buffer[256];
ssize_t n_rd = read(p_fd[0], buffer, sizeof(buffer) - 1);
```

2.7.3 Named Pipes in POSIX

Le named pipes vengono anche chiamate **FIFO** nei sistemi POSIX. Per creare una FIFO, si utilizza l'API *mkfifo*, che ha la seguente firma:

```
int mkfifo(const char *path, mode_t mode);
```

La funzione andrà a creare un nuovo file speciale FIFO con nome dato dal percorso puntato da *path*. I bit dei permessi del file saranno impostati come specificato in *mode*. La funzione ritorna 0 se la creazione del file avviene con successo, -1 altrimenti. Un esempio di chiamata può quindi essere:

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

La FIFO quindi potrà utilizzare come un normale file; in particolare possiamo aprirla tramite l'API *open*:

```
int open(const char *path, int oflag, ...);
```

La funzione quindi aprirà il file specificato in *path*. Il parametro *flag* deve obbligatoriamente assumere uno dei seguenti valori:

- **O_RDONLY**: il file viene aperto in modalità di sola lettura
- **O_WRONLY**: il file viene aperto in modalità di sola scrittura
- **O_RDWR**: Il file viene aperto ed è possibile effettuare su di esso sia le operazioni di lettura che di scrittura

Inoltre è possibile specificare, in OR bitwise (|) con uno dei valori di *flag* sopra (cioè viene fatto l'or bit a bit dei valori), le seguenti flag:

- **O_APPEND**: Il file viene aperto in "append mode", cioè ogni scrittura viene fatta alla fine del file
- **O_CLOEXEC**: Viene impostata, per questo file, la flag **FD_CLOEXEC**, cioè il file verrà chiuso se un processo che lo gestisce usa una delle funzioni della famiglia *exec* viene eseguita sul file

- **O_CLOFORK**: Viene impostata, per questo file, la flag **FD_CLOFORK**, cioè il file verrà chiuso se un processo che lo gestisce effettua una *fork()*
- **O_CREAT**: Se il **path** non esiste, lo crea come un file normale. (n.d.a. Si faccia riferimento a *man* o alla documentazione POSIX per una spiegazione più dettagliata)
- **O_DIRECTORY**: Se *path* risolve a un file che NON è una directory, questa flag porta la funzione a fallire
- **O_EXCL**: Se **O_CREAT** e **O_EXCL** sono entrambe impostate, la funzione fallirà se il file già esiste. Se invece solo questa flag è impostata, il risultato è **indefinito**
- **O_TRUNC**: Se il file esiste ed è un file normale e il file è stato aperto con successo in modalità "sola scrittura" (flag ha valore *O_WRONLY*) oppure in è aperto in "modalità lettura e scrittura" (flag ha valore *O_RDWR*), allora tutti i contenuti del file verranno troncati (cioè cancellati) e la sua lunghezza verrà impostata a 0.

Quindi, un esempio di apertura di una FIFO può essere il seguente:

```
int fd = open("/home/pietro/myfifo", O_RDONLY);
```

Poiché una FIFO può essere utilizzata come un file normale, possiamo anche usare le API *read* e *write* viste prima per leggere e scrivere su di essa:

```
char buffer[256];
ssize_t n_rd = read(fd, buffer, sizeof(buffer) - 1);
```

Al termine dell'utilizzo, dobbiamo ricordarci di **chiudere la FIFO**; per farlo usiamo l'API **close**:

```
int close(int fildes)
```

La funzione prende in input il file descriptor e lo chiude (cioè chiude il file). La funzione ritorna 0 se ha successo, -1 se invece fallisce. Per chiudere la FIFO che abbiamo presentato fino a qui come esempio, possiamo quindi procedere in questo modo:

```
close(fd);
```

Per invece **eliminare** una FIFO, possiamo usare l'API **unlink**:

```
int unlink(const char* path)
```

L'API, in generale, si comporta in maniera differente in base a **cosa viene specificato come valore di path**:

- Se quel nome era **l'ultimo collegamento ad un file e nessun processo ha quel file aperto**, il file viene **eliminato** e lo spazio che stava usando viene reso disponibile per il riuso. Se invece il file è ancora in uso da parte di un processo, esso rimarrà in memoria fino a quando **l'ultimo file descriptor che lo riferenzia non viene chiuso**.

- Se il nome specificato è un **link simbolico** (un collegamento), esso viene rimosso
- Se il nome si riferisce ad una **Socket**, una **FIFO** o ad un **dispositivo**, allora il nome viene rimosso, ma i processi che lo stavano usando **possono continuare a farlo**

La funzione ritorna 0 se ha successo, altrimenti -1. Per eliminare quindi una FIFO usando *unlink*, possiamo procedere nel seguente modo:

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

2.7.4 Segnali in POSIX

Possiamo inviare un segnale ad un processo utilizzando l'API *kill*:

```
int kill(pid_t pid, int sig)
```

Dove *pid* è l'ID del processo e *sig* è il segnale che si vuole inviare. I segnali che si possono inviare sono definiti nella libreria **signal.h**. Un esempio di utilizzo può essere il seguente:

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

Per registrare una callback per un determinato segnale, possiamo usare l'API *sigaction*:

```
int sigaction(int sig, const struct sigaction *restrict act, struct sigaction *restrict oact);
```

In particolare, la funzione permette al processo chiamato di esaminare e/o specificare l'azione che deve essere associata ad un certo segnale. Il parametro *sig* specifica il segnale, i cui valori accettabili sono definiti in **signal.h**. La **struttura** *sigaction*, usata per descrivere l'azione da intraprendere, è definita anch'essa in **signal.h** ed include almeno i seguenti membri:

Member Type	Member Name	Description
void(*) (int)	<i>sa_handler</i>	Pointer to a signal-catching function or one of the macros SIG_IGN or SIG_DFL.
sigset_t	<i>sa_mask</i>	Additional set of signals to be blocked during execution of signal-catching function.
int	<i>sa_flags</i>	Special flags to affect behavior of signal.
void(*) (int, siginfo_t *, void *)	<i>sa_sigaction</i>	Pointer to a signal-catching function.

Abbiamo poi comportamenti diversi a seconda del valore degli argomenti *act* e *oact*. Per quanto riguarda *act* abbiamo:

- Se il suo valore NON è un **puntatore a null**, esso punta ad una struttura che specifica l'azione da essere associata con il segnale specificato
- Se il suo valore È un **puntatore a null**, la gestione dei segnali **rimane invariata** (rispetto alla gestione normale del S.O.)

Per quanta riguarda *oact* invece abbiamo:

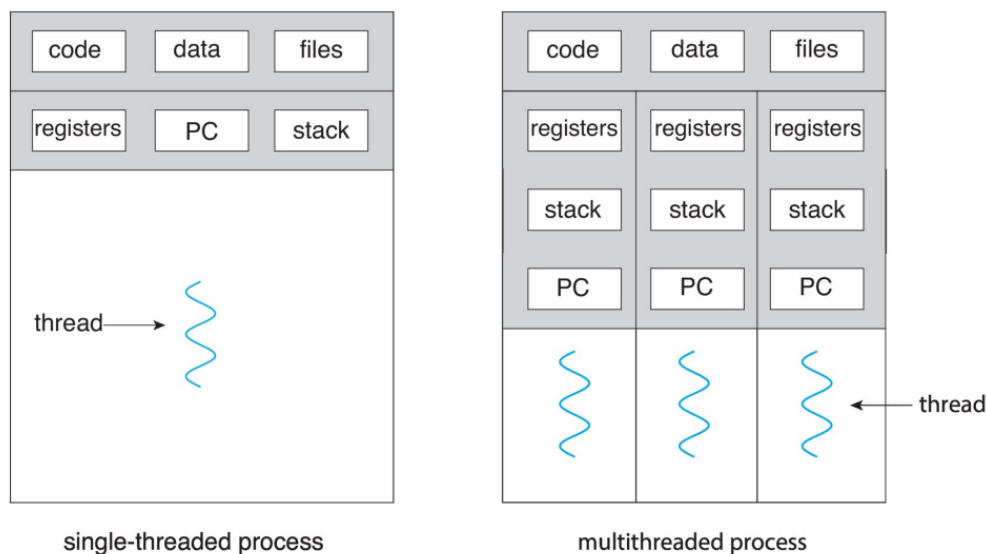
- Se il suo valore NON è un **puntatore a null**, l'azione precedentemente associata al segnale è immagazzinata nella locazione puntata da *oact*

L'API restituirà 0 se ha avuto successo, -1 altrimenti. Un esempio quindi di registrazione di una callback è il seguente:

```
struct sigaction act;
sigemptyset(&act.sa_mask); /* non bloccare gli altri segnali */
act.sa_flags = SA_SIGINFO; /* callback in act.sa_sigaction */
act.sa_sigaction = sigterm_handler; /* la callback */
int ok = sigaction(SIGTERM, &act, NULL);
```

2.8 Multithreading

Fino ad ora abbiamo assunto che un processo abbia un singolo flusso di esecuzione sequenziale (ossia, un singolo processore virtuale). Se supponiamo che un processo possa avere **molti** processori virtuali, più istruzioni possono eseguire concorrentemente, e quindi il processo può avere più percorsi (**thread**) di esecuzione concorrenti. 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 **proprio stack**, altrimenti le chiamate a subroutine di un thread interferirebbero con quello di un altro thread concorrente.



Le **librerie di un thread** sono le API fornite al programmatore per creare e gestire un thread. Le librerie più in uso sono **POSIX pthreads** e **Windows Threads**.

2.9 POSIX pthreads

La libreria **POSIX pthreads non sono un'implementazione**, ma una specifica (POSIX standard IEEE 1003.1c). Esse sono comuni nei sistemi Unix e Unix-like (BSD, Linux, MacOS). All'inizio di un processo, viene creato un singolo thread. Per creare un nuovo thread si utilizza l'API *pthread_create*, la quale ha la seguente firma:

```
pthread_create(pthread_t *restrict thread, const pthread_attr_t *restrict attr, void
               *(*start_routine)(void*), void* restrict arg);
```

L'API creerà un nuovo thread, con gli attributi specificati da *attr*, all'interno del processo. Se *attr* è NULL, gli attributi di default saranno usati. Se gli attributi specificati da *attr* sono vengono modificati successivamente, gli attributi del thread creato non verranno modificati. Se l'esecuzione va a buon fine, l'API immagazzinerà l'ID del thread creato nella locazione specificata da *thread*. Il parametro *start_routine* indica **la funzione che il thread andrà ad eseguire**, passandogli ***arg* come suo solo parametro**. Il thread continuerà ad eseguire fino a quando la funzione che sta eseguendo non ritorna. La funzione *pthread_create* ritornerà 0 se ha successo, altrimenti verrà ritornato un numero diverso da 0 per indicare l'avvenimento di un errore. Per invece attendere la fine dell'esecuzione di un thread si utilizza l'API *pthread_join*, la quale ha la seguente firma:

```
int pthread_join(pthread_t thread, void ** value_ptr);
```

L'API **sospenderà** l'esecuzione del thread chiamante fino a quando il thread obbiettivo (specificato nel parametro *thread*) termina, a meno che il thread obbiettivo non sia già terminato. Quando l'API ritorna con successo con un valore di *value_ptr* non NULL, il valore passato a *pthread_exit()* (API chiamata per cancellare un thread) dal thread terminante sarà reso disponibile nella locazione referenziata da *value_ptr*. L'API ritornerà 0 se la funzione ha successo, altrimenti sarà ritornato un numero diverso da 0 che indica l'errore. Un esempio di utilizzo di queste due API può essere il seguente:

```
void* thread_code(void* name) ...
    ...
    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);
```

2.9.1 Comportamento rispetto alle chiamate di sistema `fork()` ed `exec()`

Una `fork()` dovrebbe duplicare solo il thread chiamante o tutti i thread? Alcuni sistemi operativi Unix-like hanno **diverse** `fork()`.

`exec()` invocata da un thread che effetto ha sugli altri thread? Di solito **termina tutti i thread del processo** precedentemente all'esecuzione.

2.9.2 Gestione dei segnali

Quando un processo è single-threaded, un segnale interrompe l'unico thread del processo. Quando vi sono più thread, quale thread riceve il segnale? Vi sono diverse soluzioni:

- Il thread a cui si applica il segnale (ad es. il segnale `SIGSEGV` viene inviato al thread che ha generato il segmentation fault)
- Ogni thread del processo
- Alcuni thread del processo
- Un thread speciale del processo deputato esclusivamente alla ricezione dei segnali

2.9.3 Cancellazione dei thread

L'operazione di cancellazione di un thread determina la terminazione prematura del thread. Può essere invocata anche da un altro thread. Vi sono due approcci per la cancellazione:

- **Cancellazione asincrona:** il thread che riceve la cancellazione viene terminato immediatamente
- **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

Questi due approcci hanno i seguenti vantaggi:

- **Cancellazione differita:** dal momento che un thread controlla il momento della propria cancellazione, può effettuare una terminazione ordinata
- **Cancellazione asincrona:** nessuna necessità di controllare periodicamente se ci sono richieste di cancellazione pendenti

Nei **POSIX pthreads** si può attivare/disattivare la cancellazione, ed avere sia cancellazione differita (default) che asincrona. Se la cancellazione è inattiva, le richieste di cancellazione rimangono in attesa fino a quando (se) è attivata. In caso di **cancellazione differita**, questa avviene solo quando l'esecuzione del thread raggiunge un punto di cancellazione (di solito una chiamata di sistema bloccante). Il thread può aggiungere un punto di cancellazione controllando l'esistenza di richieste di cancellazione con la funzione `pthread_testcancel()`. Presentiamo un esempio:

```

void *thread_code(void *name) {
    int oldtype, oldstate;
    pthread_setcanceltype(PTHREAD_CANCEL_DEFERRED, &oldtype);
    pthread_setcancelstate(PTHREAD_CANCEL_ENABLE, &oldstate);
    while (true) {
        pthread_testcancel();
        ... /* fa qualcosa di non interrompibile ma di durata finita */
    }
}
...
pthread_id tid;
int ok = pthread_create(&tid, NULL, thread_code, "thread 1");
... /* dopo un po' */
ok = pthread_cancel(tid);
void *ret;
ok = pthread_join(tid, &ret);

```

2.9.4 Dati locali dei thread

In alcuni casi è utile assegnare ad un thread dei dati locali (**thread local storage, TLS**) non condivisi con gli altri thread dello stesso processo. La TLS è diversa dalle variabili locali (ad es. è visibile a tutte le funzioni). Essa è simile ai dati *static* del linguaggio C, ma unica per ciascun thread. È utile quando il programma non ha un controllo diretto sul momento di creazione dei thread (es. quando si usano i thread pools). In POSIX, i dati locali dei thread si possono creare utilizzando le funzioni:

- ***pthread_key_create(...)***: crea un oggetto opaco di tipo *pthread_key_t*, che può essere usato da tutti i thread per identificare un dato locale
- ***pthread_set_specific(...)***: permette di associare ad una *pthread_key_t* un valore di tipo *void**
- ***pthread_getspecific(...)***: permette di richiamare un oggetto *pthread_key_t* data la chiave

Ogni thread può associare ad una stessa chiave il proprio distinto valore locale (e successivamente richiamarlo).

3 Gestione della memoria: i servizi

Perché un programma possa andare in esecuzione esso deve avere a disposizione:

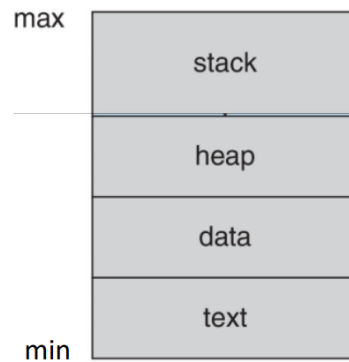
- Il processore, per eseguire il codice
- La memoria centrale, per memorizzare il codice e i dati sul quale il codice opera

Solo nei sistemi operativi più semplici un solo programma alla volta è in memoria: nei moderni sistemi operativi molti programmi sono contemporaneamente in memoria in uno stesso istante. Secondo la terminologia precedentemente introdotta: più immagini di più processi sono presenti contemporaneamente nella memoria centrale. Il sistema

operativo deve, pertanto, allocare porzioni di memoria centrale ai diversi processi in funzione delle necessità di tali processi.

3.1 Lo spazio di indirizzamento

Ogni processo ha disposizione uno spazio di indirizzamento che può usare per le proprie operazioni.



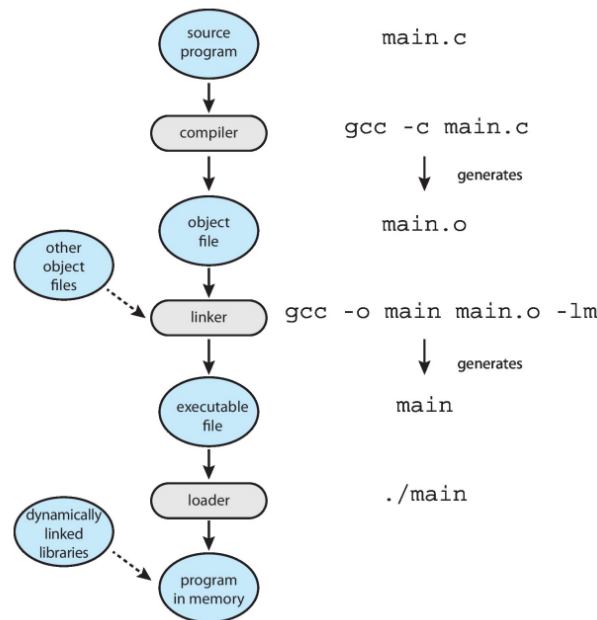
Nei primi sistemi operativi, tale spazio di indirizzamento era il range di indirizzi di memoria centrale che veniva assegnato al processo: ad esempio, se l'immagine di un certo processo avesse avuto dimensione 1 MB e fosse stata caricata in memoria centrale dall'indirizzo 001B:000... il suo spazio di indirizzamento sarebbe stato 001B:0000 - 002B:0000. Questo però non permette di caricare lo stesso programma in zone diverse della memoria!

3.2 Associazione degli indirizzi

In presenza di molti programmi in memoria, il sistema operativo di regola carica uno stesso programma, in momenti diversi, in diverse aree di memoria (dove trova spazio). 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? Una prima possibilità è che il compilatore produca codice indipendente dalla posizione, ossia **position-independent code (PIC)**, ossia codice macchina che usi solo indirizzi di memoria **relativi**, e che quindi funzioni correttamente in qualsiasi locazione di memoria venga caricato. Una seconda possibilità è produrre codice dipendente dalla posizione e tradurre gli indirizzi dipendenti dalla posizione negli indirizzi corretti. Questa operazione di "traduzione" è detta di **associazione (binding) degli indirizzi**.

3.2.1 Loader e Linker

Un programma sorgente è compilato in un file oggetto che deve poter essere caricato a partire da qualsiasi locazione di memoria fisica (**file oggetto rilocabile**). I **linker**, o linkage editor, combinano più file oggetto (diversi file sorgente + librerie) per formare un file eseguibile. I **loader** si occupano di caricare in memoria i file eseguibili nel momento in cui devono essere eseguiti. Inoltre, i loader (o ulteriori linker dinamici) effettuano il linking delle **librerie dinamiche**.



3.2.2 Librerie dinamiche

Nei sistemi operativi odierni, non tutto il linking viene fatto a compile time: le librerie dinamiche vengono collegate a quando il programma **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.

3.2.3 Varianti nell'associazione degli indirizzi

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

Questi approcci hanno sia **vantaggi e svantaggi**:

- **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

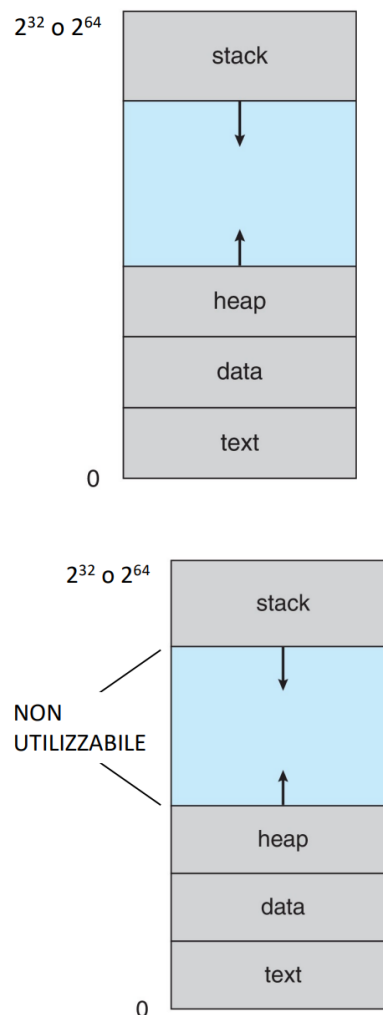
deve contenere delle opportune tabelle che indichino le istruzioni macchina da modificare

- **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

Il binding in esecuzione è quello di fatto usato in tutti i sistemi operativi moderni; il binding in compilazione è usato per **alcuni eseguibili speciali, come il kernel**, di cui si sa a priori l'indirizzo di caricamento.

3.3 Spazio di indirizzamento virtuale

Nei sistemi operativi moderni, ogni processo ha uno **spazio di indirizzamento virtuale**, o **virtual address space (VAS)** indipendente dagli indirizzi fisici della memoria centrale nella quale l'immagine è caricata.

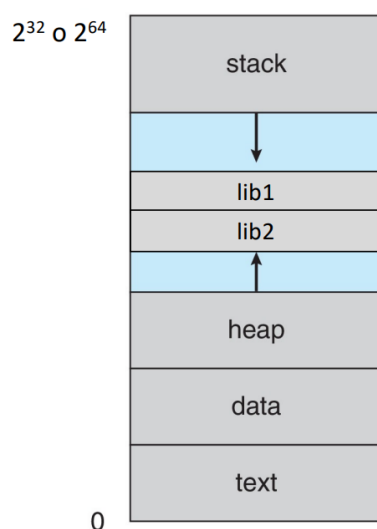


Tale spazio di indirizzamento si estende dall'indirizzo 0 al massimo indirizzo consentito dall'architettura del processo. Delle **tecniche di associazione degli indirizzi in esecuzione** fanno corrispondere lo spazio di indirizzamento virtuale del processo con

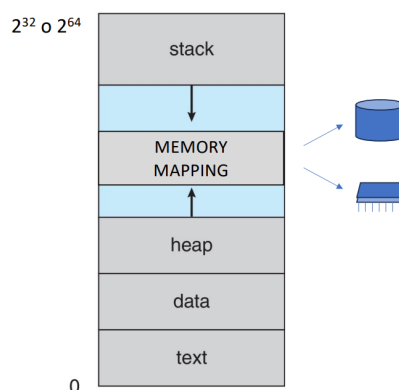
la regione (o le regioni) di memoria centrale che la sua immagine occupa. Lo spazio di indirizzamento virtuale di un processo è di regola **molto più ampio della memoria centrale**. Questo implica che buona parte dello spazio di indirizzamento virtuale **non è utilizzabile dal processo** perché non è associato a nessuna regione di memoria centrale. Tale parte inutilizzabile è di solito compresa tra stack e heap. Stack e heap possono essere dinamicamente estesi e ridotti (automaticamente lo stack, attraverso le API lo heap).

3.3.1 Librerie dinamiche

Le librerie dinamiche vengono caricate nella zona tra stack e heap. Dal momento che possono essere caricate in qualsiasi posizione nel VAS, devono essere **compilate come PIC**.



3.3.2 Memory mapping



In generale, i sistemi operativi mettono a disposizione API per mappare una regione inutilizzabile del VAS su memoria centrale, così che diventi utilizzabile. Esistono anche API che permettono di mappare una regione del VAS sul contenuto di un file (file mappati in memoria). In tal modo l'accesso al file può avvenire utilizzando le istruzioni macchina per accedere alla memoria, anziché le API del filesystem.

3.4 Le API POSIX per la gestione della memoria

Di norma non dobbiamo usare le API per gestire stack e heap:

- Lo stack è gestito automaticamente dal sistema operativo, non tramite API
- Lo heap è gestito di norma dal supporto runtime del linguaggio (`new` in C++) o dalla sua libreria (`malloc` in C), che **invocano API** per ridurre/espandere lo heap in funzione delle necessità del processo

Perché allora ci interessa sapere quali sono le API per la gestione della memoria?

- Ci permettono di avere regioni di memoria con **permessi particolari** (sola lettura, eseguibili...)
- Ci permettono di implementare componenti quali allocatori di memoria, compilatori just-in-time,... qualora volessimo implementare il nostro nuovo linguaggio di programmazione
- Ci permettono di utilizzare i file mappati in memoria e la memoria condivisa

Le API Unix legacy per cambiare la dimensione del segmento dati (che nello standard POSIX comprende le regioni data e heap) sono *brk* e *sbrk*. Tali API sono deprecate in favore dell'API *mmap* (già presentata in questi appunti), e incompatibili con questa (ma esistono ancora in diversi OS, ad esempio Linux). L'API *mmap* permette di mappare una regione ancora non utilizzata del VAS su:

- **Memoria centrale**
- **Un file** (che viene mappato in memoria)
- **Memoria condivisa**

Vediamo un'ulteriore API per la gestione della memoria: *msync*:

```
int msync(void* addr, size_t len, int flags);
```

L'API andrà a scrivere sul filesystem tutti i dati modificati in un file mappato in memoria da *mmap*. Senza l'uso di questa funzione, non c'è garanzia che i cambiamenti scritti sul file in memoria vengano mantenuti. Per essere più precisi, la parte del file che corrisponde all'area di memoria che inizia a *addr* e avente lunghezza *len* viene aggiornata. L'attributo flag può avere i seguenti valori:

- **MS_ASYNC**: La scrittura delle modifiche viene effettuata in maniera asincrona
- **MS_SYNC**: La scrittura delle modifiche viene effettuata in maniera sincrona
- **MS_INVALIDATE**: Viene richiesto di invalidare altra mappature dello stesso file, così che possano essere aggiornate con i valori appena scritti

Queste flag sono definite in `<sys/mman.h>` e possono essere combinate con un **bitwise-inclusive OR** (`|`). La funzione ritorna 0 se ha successo, -1 altrimenti. Vediamo un esempio di utilizzo di queste API:


```

// Mappiamo 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 scriviamo:
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, scriviamo:
int ok = msync(0xb0000000, 8192, MS_SYNC|MS_INVALIDATE);

```

4 File system: i servizi

Il **file system** è il modo attraverso il quale il sistema operativo memorizza in linea i dati e i programmi. Esso è costituito da:

- Un insieme di file
- Una struttura delle directory, che organizza i file

Un **file** è una unità di memorizzazione logica, un insieme di informazioni correlate, registrare in memoria secondaria, alle quali è **stato dato un nome**. Un file a sua volta è costituito da una **sequenza di record, righe, bit o byte**, il cui significato è definito dal creatore del file. Un file possiede i seguenti 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 S.O. 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 ecc...

Le informazioni sul file sono memorizzate nelle **directory**.

4.1 Operazioni dei processi sui file

I processi possono effettuare le seguenti operazioni sui file:

- **Creazione**: viene riservato spazio nel file system per i dati, e viene aggiunto un elemento nella directory

- **Apertura:** effettuata prima dell'utilizzo del 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 del file
- **Cancellazione e troncamento:** il troncamento cancella i dati **ma non il file** con i suoi attributi

4.2 Lock dei file

Uno stesso file può essere aperto **contemporaneamente da più processi che operano in concorrenza**. Alcuni sistemi operativi permettono di associare ai file (o a porzioni di esso) dei **lock** per coordinare i processi che operano sullo stesso file. Vi sono due tipi di lock:

- **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

Altre possibilità sono:

- **Lock obbligatori (mandatory):** il sistema operativo proibisce l'accesso al file ai processi che **non detengono il lock**
- **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

I sistemi **Windows** adottano i lock obbligatori, i sistemi **Unix-like** i lock consultivi.

4.3 Tipi di file

Vi sono principalmente due tipi di file:

- Dati (numerici, testo, binari)
- Programmi

Il sistema operativo può essere più o meno consapevole del tipo di file, **ma deve almeno riconoscere il tipo di file eseguibile**. Possibili tecniche per riconoscere il tipo di file sono:

- Schema del nome (nome.estensione)

- Attributi nel file (ad esempio in macOS, viene registrato il programma che ha creato il file)
- "Magic Number" all'inizio del file (ad esempio la "shebang" magic cookie ("#!") all'inizio degli script Unix)

4.4 Struttura dei file

I file possono avere o meno una struttura ben definita; alcune possibilità sono:

- Nessuna struttura (ad esempio, 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**, soprattutto per i file eseguibili (formato PE in Windows, a.out ed ELF nei sistemi Unix-like, Mach-O in macOS)

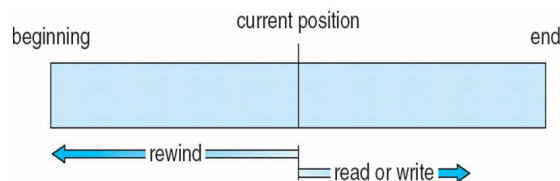
Più il sistema operativo supporta direttamente diverse strutture di file, **più diventa complesso**. Inoltre se il sistema operativo è troppo "rigido" sulle possibili strutture, potrebbe non supportare nuovi tipi, o tipi **ibridi**.

4.5 Metodi di accesso a file

Vediamo ora quali possono essere i principali metodi di accesso ad un file

4.5.1 Accesso sequenziale

In questo tipo di accesso, il file è **una sequenza di record a lunghezza fissa**



Le operazioni disponibili in questo caso sono:

- *read_next()* e *write_next()* leggono/scrivono il successivo record dalla posizione corrente
- Operazione di **riavvolgimento**

4.5.2 Accesso diretto

In questo tipo di accesso, abbiamo a disposizione le seguenti operazioni:

- Operazioni *read(n)* e *write(n)* per accedere direttamente all'n-esimo record
- Alternativamente, possiamo usare *read_next()*, *write_next()* e *position(n)*

4.5.3 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**. In tali sistemi, l'accesso può essere basato **sulla chiave**, e il sistema operativo mantiene un indice per velocizzare l'accesso. Esempio è **l'accesso ISAM** (Indexed Sequential Access Method) nei sistemi IBM, file systems **Files-11** prodotto da Digital per il sistema operativo OpenVMS (offre tutti e tre i tipi di accesso su record a lunghezza fissa o variabile).

4.6 Directories

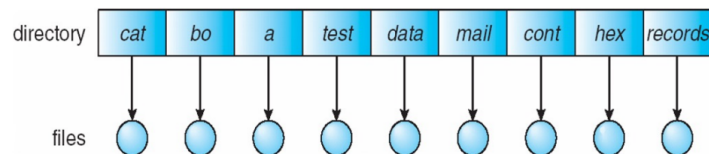
Una **directory** è, in pratica, un elenco di file presenti nel file system. Più formalmente, è **una tabella** che permette di associare il nome di un file ai dati (e metadati) contenuti nel file stesso. Sia i file che le directory risiedono sul disco: deve esservi **almeno una directory nel file system** (altrimenti non sarebbe possibile ritrovare i file!) I processi possono effettuare le seguenti operazioni sulle directory:

- **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 effettuare un backup

Le directories possono avere diversi tipi di struttura:

- Ad un livello
- A due livelli
- Ad albero
- A grafo aciclico
- A grafo generale

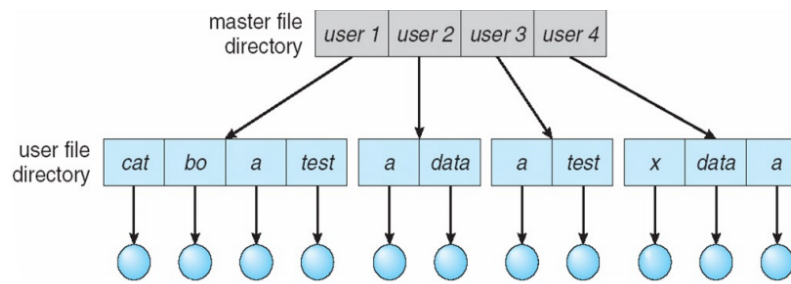
4.6.1 Struttura delle directory ad un livello



Nella struttura ad un livello, **esiste una sola directory per tutti i file**. Il vantaggio di questa struttura è la **semplicità**, tuttavia ha diversi svantaggi:

- Difficoltà nel nominare i file quando sono molti
- Difficoltà nel raggruppare i file per utenti diversi

4.6.2 Struttura delle directory a due livelli

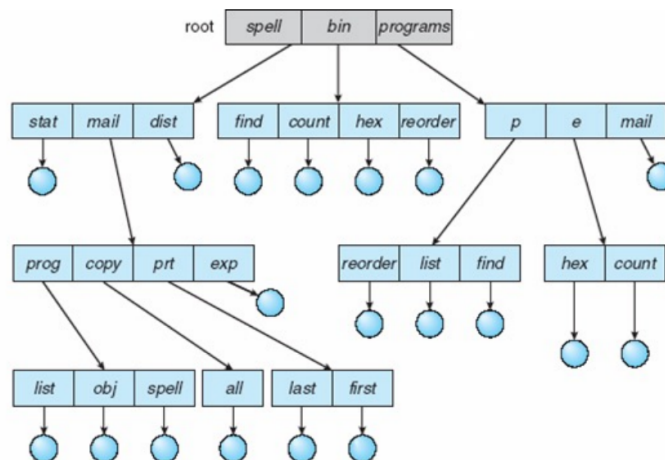


Nella struttura a due livelli, la directory principale **contiene delle sottodirectory, una per ogni utente**. La sottodirectory utente contiene i **file dell'utente**; utenti diversi possono quindi dare **lo stesso nome a file diversi**. Occorre quindi usare dei **nomi di percorso (path name)** per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

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

I file di sistema sono di solito posti **in una o più directory speciali**, e occorre che il sistema conosca un **percorso di ricerca** per trovarli.

4.6.3 Struttura delle directory ad albero



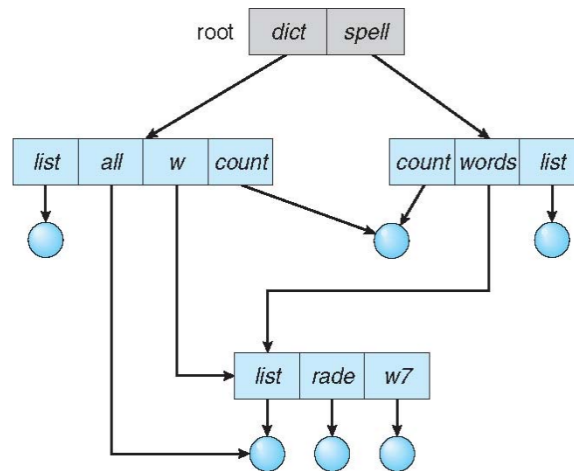
Nella struttura ad albero, ogni directory **contiene ricorsivamente files e altre directory**. Questa struttura quindi permette agli utenti di **raggruppare i propri file**. Per semplificare l'accesso, ad ogni programma è assegnata una **directory corrente**, dalla quale si possono specificare **path relativi**

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

In questo caso, come avviene la cancellazione della directory?

- Con la directory **cancello tutto il suo contenuto**
- Oppure permetto di cancellare una directory **solo se è vuota**. Questa soluzione è la più sicura

4.6.4 Struttura delle directory a grafo aciclico

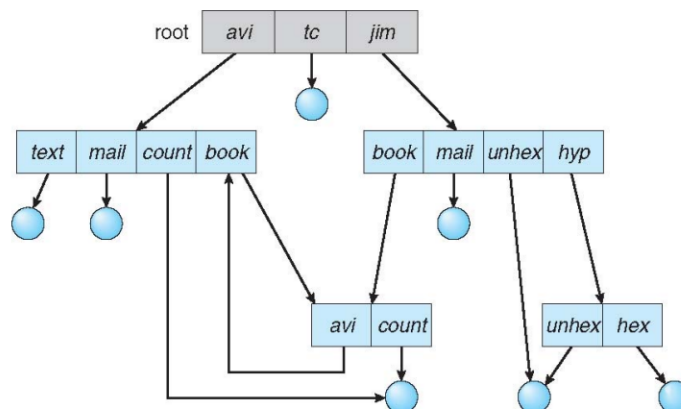


La struttura a grafo aciclico **permette l'aliasing** (più di un nome per lo stesso file). Cosa succede però se **cancello un file/directory con un alias**?

- **Hard links**: duplicazione voci di directory; viene introdotto un **contatore** ai riferimenti, quando è a zero viene **cancellato il file**
- **Link simbolici**: riferimenti simbolici ad un **path assoluto**, quando questo è cancellato restano **dangling**; non sono aggiunti al contatore dei riferimenti

Questo tipo di struttura delle directory introduce anche un **problema di attraversamento del file system**.

4.6.5 Struttura delle directory a grafo generico



In questa struttura, vi è la possibilità di **hard links** anche a **directory** a **livelli superiori**, persino che contengono **ricorsivamente il link stesso**. Per determinare se un file non è più referenziato, **un contatore al numero di riferimenti non basta più** ma occorre un autentico algoritmo di **garbage collection**. Attraversare il file system in questo tipo di struttura diventa quindi ancora più complesso.

4.7 Protezione

Le informazioni devono essere preservate dai danni fisici (affidabilità) e dagli accessi impropri (protezione). Un sistema multiutente permette un **accesso controllato ai file** di un certo utente da parte degli altri utenti. Le operazioni controllabili sono:

- Lettura
- Scrittura
- Esecuzione
- Aggiunta (scrittura in coda ad un file)
- Cancellazione
- Elencazione (elenco del contenuto di una directory)

4.7.1 Liste di controllo degli accessi

L'idea questo tipo di protezione è la seguente:

- Ad ogni file/directory è associata una **lista di controllo degli accessi** (access control list, ACL)
- l'ACL specifica gli utenti **che possono accedere al file/directory**, con i relativi permessi di accesso
- Il file system controlla l'ACL prima di ogni accesso al file

Il **principale svantaggio** di questo approccio è che le ACL **possono diventare molto lunghe**. Nei sistemi Unix-like, l'approccio usato è **raggruppare gli utenti in classi distinte**:

- **Proprietario**: l'utente che possiede il file/directory
- **Gruppo**: il gruppo di utenti che condivide il file/directory
- **Pubblico**: tutti gli altri utenti

Facciamo un esempio:

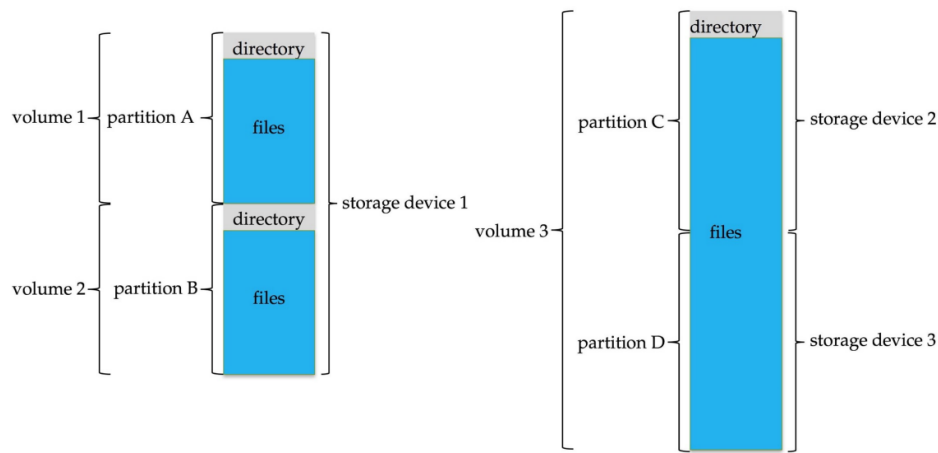
Gruppo	Bit di permesso (base 10)	r	w	x
Proprietario	7	1	1	1
Gruppo	6	1	1	0
Pubblico	4	1	0	0

Supponiamo di avere un file *game*, di volergli assegnare un gruppo G e di volergli attribuire i permessi di cui sopra. Allora procediamo come segue:

- *chgrp G game # cambia gruppo al file game*
- *chmod 764 game # cambia i permessi al file game*

4.8 Volumi e montaggio

Un sistema operativo deve permettere di aggiungere e rimuovere dinamicamente unità di memorizzazione dati. Un dispositivo di archiviazione può essere suddiviso in **partizioni**:

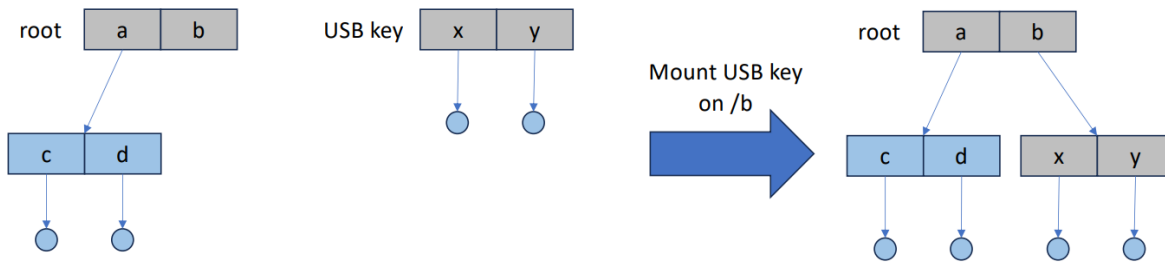


Un **volume** è una **zona di un dispositivo di archiviazione contenete un file system**. Un volume, di solito, è contenuto in una partizione, su un solo dispositivo, ma per alcuni filesystem particolari, un volume si può **estendere su più partizioni/dispositivi**. Il sistema operativo deve permettere di **montare e smontare** un volume all'interno dello spazio dei nomi del file system. Di solito 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 file system contenuto nel volume** (tipicamente una directory vuota)

Esistono diverse varianti di questo meccanismo:

- **Punto di montaggio:** directory vuota oppure no
- **Montaggio:** automatico o manuale
- **Utilizzo di identificatori (lettere) di unità** anziché punti di montaggio
- **Montaggio ripetuto di uno stesso volume permesso** oppure no



4.9 API POSIX per operazioni sui file

Per creare un file, possiamo utilizzare l'API *creat*:

```
int creat(const char* path, mode_t mode);
```

Essa si comporta tuttavia come se fosse implementata nel seguente modo:

```
int creat(const char *path, mode_t mode)
{
    return open(path, O_WRONLY|O_CREAT|O_TRUNC, mode);
}
```

Quindi il parametro *mode* dovrà assumere i valori delle flag presentate per *open*. I valori di ritorno della funzione saranno quindi quelli della funzione *open*. Per creare un file *foo.txt* con diritti di lettura e scrittura per il proprietario, e di sola lettura per i membri del gruppo e gli altri possiamo fare nel seguente modo:

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

oppure possiamo chiamare direttamente *open*. Per aprire un file, possiamo usare l'API *open* (già presentata), mentre per leggere da un file possiamo usare la funzione *read* (già presentata). Per scrivere su un file possiamo usare la funzione *write* (già presentata), mentre per **riposizionarci all'interno di un file** possiamo usare la funzione *lseek*

```
off_t lseek(int fildes, off_t offset, int whence);
```

La funzione riposizionerà il puntatore per il file aperto associato al file descriptor *fildes* nel seguente modo:

- Se *whence* ha valore *SEEK_SET*, il puntatore sarà impostato al byte *offset*
- Se *whence* ha valore *SEEK_CUR*, il puntatore sarà impostato al byte corrispondente alla somma della posizione corrente e di *offset*
- Se *whence* ha valore *SEEK_END*, il puntatore sarà impostato alla fine del file più *offset*
- Se *whence* ha valore *SEEK_HOLE*, il puntatore sarà impostato alla prima posizione con un "buco" (regione del file senza dati) con grandezza maggiore o uguale a *offset*, a meno che *offset* non cada oltre l'ultimo byte non all'interno di un buco; in questo caso, il puntatore viene impostato all'ultimo byte del file (cioè, c'è sempre un buco implicito alla fine di ogni file)

- Se *whence* ha valore *SEEK_DATA*, il puntatore sarà impostato alla più piccola posizione di un byte che NON è in un buco a distanza minima *offset* dalla posizione corrente. Verrà dato errore se questo byte non esiste

Le flag appena presentate sono presenti nella libreria *unistd.h*. Un esempio del suo utilizzo è il seguente:

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

La funzione ritornerà l'offset risultante, misurato in byte dall'inizio del file, se ha successo, altrimenti -1. Per chiudere un file utilizziamo l'API *close* (già presentata), mentre per cancellarlo dal file system usiamo l'API *unlink* (già presentata). Per troncare un file invece fino ad una lunghezza *n*, possiamo usare la funzione *truncate*:

```
int truncate(const char* path, off_t length);
```

La funzione quindi troncherà il file identificato dal path *path* fino alla posizione *length* (cioè imposta la sua lunghezza a *length*). I dati presenti dopo *length* sono **scartati**. La funzione ritornerà 0 se ha successo, -1 altrimenti. Un esempio del suo uso può essere il seguente:

```
int ok = truncate("foo.txt", n); //tronca il file alla lunghezza n
```

4.9.1 API POSIX per i lock

In POSIX, i lock sui file sono **consultivi**. Il lock su un file si può effettuare con l'operazione *fcntl*, che è un API che permette di effettuare **diversi tipi di operazioni sui file**. La sua firma è:

```
int fcntl(int fildes, int cmd, ...);
```

Essa ha parecchie funzionalità, quindi qui ci concentreremo solo su quelle necessarie per effettuare un lock. Occorre specificare il tipo di lock utilizzando una **struct flock**. Se si vuole effettuare un lock su un file puntato da *fildes*, l'argomento di *cmd* dovrà essere *F_SETLK*. Vediamo un esempio:

```
struct flock fl;
fl.l_type = F_RDLCK; // read lock
fl.l_whence = SEEK_CUR; // dalla posizione corrente...
fl.l_start = 10; //...più 10 byte
fl.l_len = 0; //fino alla fine del file
int ok = fcntl(fd, F_SETLK, &fl); // se non riesce ad impostare un lock, fallisce
```

4.10 API POSIX per operazioni su directory

Per creare una nuova directory, possiamo usare la funzione *mkdir*

```
int mkdir(const char* path, mode_t mode);
```

Essa creerà una nuova directory con nome *path*. Il parametro *mode* indica i bit di permesso per la nuova directory. Per esempio, creiamo una nuova directory, con diritti di lettura, scrittura e ricerca per il proprietario e di sola lettura per i membri del gruppo e gli altri nel seguente modo:

```
mkdir("/home/pietro/newdir", S_IRWXU|S_IRGRP|S_IROTH);
```

La funzione restituisce 0 se ha successo, -1 altrimenti. Come si può notare, possiamo anche usare le flag viste in precedenza, concatenare con un bitwise inclusive OR (`|`), per specificare i permessi. Per eliminare una directory (solo se vuota) possiamo usare la funzione *rmdir*:

```
int rmdir(const char* path);
```

La funzione rimuoverà la directory specificata da *path*. Ritorna 0 se ha successo, -1 altrimenti. Un esempio di un suo utilizzo può essere il seguente:

```
rmdir("/home/pietro/newdir");
```

Possiamo aprire una directory tramite l'API *opendir*:

```
DIR *opendir(const char *dirname);
```

La funzione aprirà la directory specificata da *dirname* (in particolare, apre un **directory stream**, che è una sequenza ordinata di tutte le voci in una directory). La funzione ritornerà un **puntatore ad una struttura di tipo DIR** (cioè un puntatore al directory stream) se ha successo, altrimenti ritornerà un puntatore a null. Per leggere da una directory possiamo usare l'API *readdir*:

```
struct dirent* readdir(DIR *dirp);
```

La funzione ritornerà un puntatore ad una struttura *dirent*, definita in *dirent.h*, che rappresenta la voce nella directory alla posizione corrente nello stream specificato nell'argomento *dirp*; dopodiché posizionerà lo stream **alla prossima voce della directory**. Se la funzione incontra un errore oppure arriva alla fine della directory, ritornerà un puntatore a null. Per cambiare il nome, o spostare, un file o una directory si può usare l'API *link* in combinazione con l'API *unlink*. Presentiamo prima la firma di *link*:

```
int link(const char* path1, const char* path2);
```

La funzione creerà un nuovo **hard link** nel percorso *path2* per il file identificato dal *path1*. La funzione ritorna 0 se ha successo, -1 altrimenti. Per ottenere l'effetto descritto sopra, possiamo fare come segue: supponiamo di voler spostare il file */foo/bar* nella directory */home/pietro* e, allo stesso tempo, cambiare il suo nome in *baz*:

```
int ok1 = link("/foo/bar", "/home/pietro/baz");  
int ok2 = unlink("/foo/bar");
```

Quando il link count di un file **arriva a zero**, il file **viene cancellato**. Per creare un link simbolico, possiamo usare l'API *symlink*:

```
int symlink(const char* path1, const char* path2);
```

La funzione creerà un link simbolico chiamato *path2* per il file con path *path1*. Facciamo un esempio:

```
int ok = symlink("/home/pietro/baz", "/home/pietro/alias");
```

I link simbolici non vengono aggiunti al link count.

4.11 API POSIX per la protezione

In POSIX, la protezione è basata su bit di lettura/scrittura/esecuzione per i gruppi proprietario/gruppo/pubblico. Per cambiare il proprietario e il gruppo di un file possiamo usare la funzione *chown*:

```
int chown(const char *path, uid_t owner, gid_t group);
```

La funzione imposterà per il file in *path* *owner* come nuovo proprietario del file e *group* come nuovo gruppo di appartenenza. La funzione ritorna 0 se ha successo, -1 altrimenti. Facciamo un esempio:

```
uid_t owner = ...; // Un nuovo proprietario
gid_t group = ...; // Un nuovo gruppo di appartenenza
int ok = chown("/home/pietro/baz", owner, group);
```

Per cambiare i diritti di lettura/scrittura/esecuzione, possiamo usare la funzione *chmod*:

```
int chmod(const char* path, mode_t mode);
```

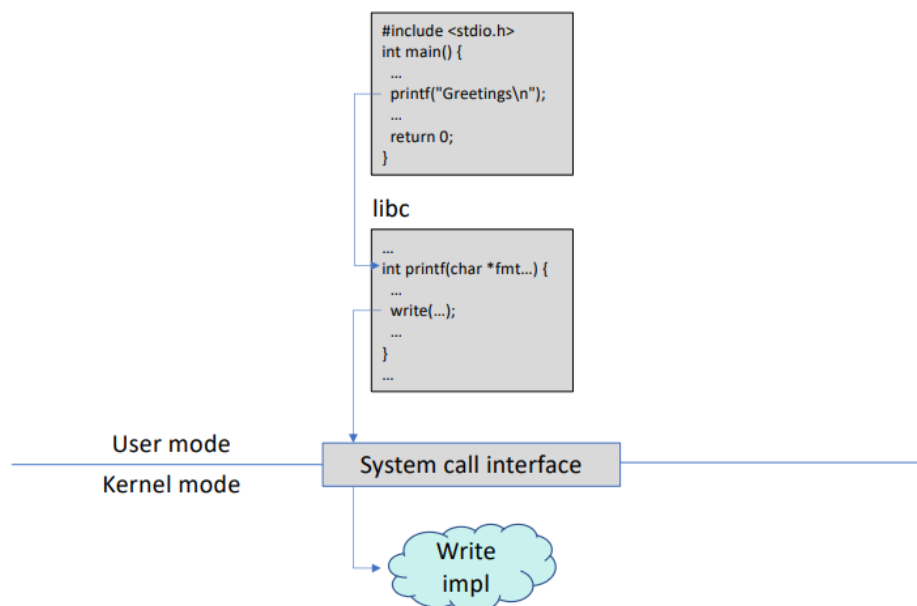
La funzione imposterà i nuovi bit di permesso per il file in *path* a *mode*, la quale può essere descritta con le flag viste fino ad ora per *mode*, concatenate con un bitwise inclusive OR. Facciamo un esempio:

```
int ok = chmod("/home/pietro/baz", S_IRUSR | S_IRGRP);
```

la funzione ritornerà 0 se ha successo, -1 altrimenti.

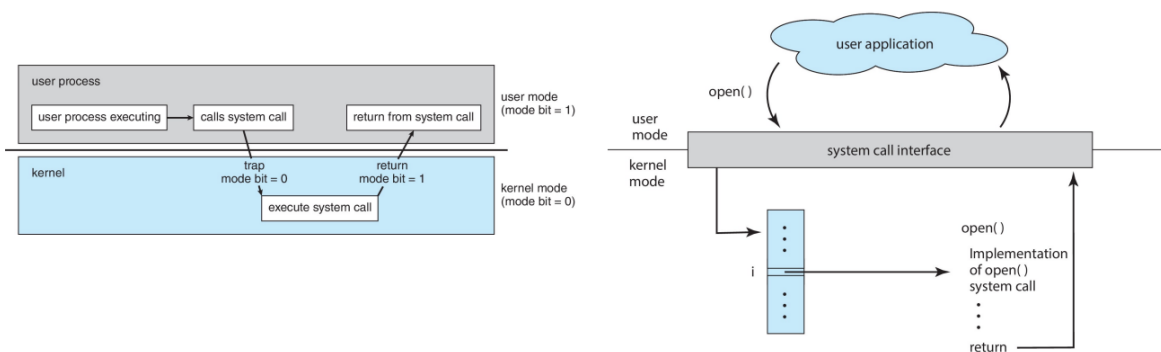
5 Interfaccia e struttura del kernel

Un'implementazione delle API attraverso chiamate di sistema segue il seguente schema:



ciò permette al sistema operativo di proteggere se stesso dai programmi in esecuzione. La CPU può funzionare in **modalità utente** (user mode) o in **modalità di sistema** (kernel mode) impostando un opportuno **bit di modalità**. Alcune istruzioni del processore sono quindi **privilegiate**, ossia sono eseguibili solamente in kernel mode: in particolare, in user mode la CPU **non può accedere alla memoria del kernel**. Una chiamata di sistema non è quindi semplice da implementare come una normale chiamata a funzione, poiché occorre **effettuare una transizione da modalità utente a modalità di sistema** (context switching):

1. Vengono prima **preparati i parametri necessari**
 - Un numero che identifica quale chiamata di sistema va effettuata...
 - ...più tutti i parametri necessari alla specifica chiamata di sistema
2. Quindi, viene invocata un'opportuna **istruzione macchina** che genera un'eccezione software; essa fa passare la CPU in kernel mode e trasferisce il controllo ad una **subroutine** ad un determinato indirizzo di memoria
3. La subroutine, chiamata **system call interface**, legge il numero identificativo della chiamata di sistema, effettua un **lookup da una tabella interna dell'indirizzo della routine che effettivamente implementa la chiamata di sistema**, e salta a tale indirizzo
4. La routine invocata legge i parametri ed esegue la funzionalità richiesta
5. Al ritorno, il processore passa di nuovo in user mode



Come avviene quindi il passaggio dei parametri alle chiamate di sistema? Dal momento che l'invocazione delle chiamate di sistema passa per un'eccezione software, il passaggio di parametri è più complesso rispetto a quello di una normale chiamata di procedura. Vi sono diversi modi:

- **Metodo più semplice:** passare i parametri nei registri del processore
 - **Vantaggio:** rapido
 - **Svantaggio:** utile solo per pochi parametri i cui tipi di dati hanno dimensione limitata
- **Altro metodo:** passo in uno dei registri un indirizzo di memoria ad un blocco nel quale sono memorizzati i parametri

- Utilizzato da Linux, insieme al primo metodo
- **Altro metodo:** faccio push dei parametri sullo stack
 - **Vantaggi:** flessibile, simile ad una normale chiamata di procedura
 - **Svantaggi:** lento e macchinoso

È necessario notare che **ogni thread ha di solito due stack**:

- Quello che viene utilizzato dal programma in user mode
- Quello che viene utilizzato quando il thread **passa in kernel mode**

Una chiamata di sistema, come prima cosa, **imposta lo stack del thread corrente allo stack di sistema** e, al termine della chiamata di sistema, ripristina lo stack a quello utente. Per quale motivo? **Per sicurezza:** dal momento che il processo potrebbe modificare a suo piacimento il registro stack pointer (che non è privilegiato), allora **non è possibile fidarsi che questo punti ad uno stack "sano"**: pertanto, in kernel mode, occorre usare uno stack sicuramente corretto.

5.1 Uso delle librerie dinamiche per le API

Si vuole far sì che, anche se il sistema operativo viene aggiornato, non vi sia bisogno di ricompilare/linkare le applicazioni qualora siano cambiate le chiamate di sistema, o l'implementazione delle API, **purché l'interfaccia delle API resti la stessa**. Un vantaggio si ha realizzando le API e le librerie standard del linguaggio come **librerie dinamiche**. Infatti, se queste sono modificate (nell'implementazione, non nell'interfaccia), non occorre ricompilare tutti gli eseguibili per aggiornarli alla nuova versione delle librerie. Occorre però un'ulteriore accortezza: oltre all'API **non deve cambiare l'Application Binary Interface (ABI)**. L'ABI è l'**insieme delle convenzioni attraverso le quali il codice binario dell'applicazione si interfaccia con il codice binario della libreria dinamica delle API**:

- Come si chiama internamente alla libreria la funzione da invocare? (**name mangling**)
- In che ordine i parametri vengono messi sullo stack delle chiamate?
- Come sono strutturati i tipi di dati? C'è padding? Qual è l'endianess? (Cioè, qual'è l'ordine da seguire per interpretare i byte)

5.2 La scarsa portabilità degli eseguibili binari

Come facciamo ad avere applicazioni **portabili su diversi sistemi di elaborazione**? Ci sono tre possibili approcci:

- Scrivere l'applicazione in un linguaggio con un **interprete portatile** (es. Python, Ruby): in tal caso, **l'eseguibile è il sorgente**
- Scrivere l'applicazione in un linguaggio con un **ambiente runtime portatile** (es. Java, .NET): in tal caso, **l'eseguibile è il bytecode**

- Scrivere l'applicazione utilizzando un **linguaggio con un compilatore portatile ed API standardizzate**: l'eseguibile è **file binario compilato e linkato**

Nei primi due casi, l'eseguibile è normalmente **uno solo per tutte le architetture**. Nel terzo caso invece, occorre, di norma, generare un eseguibile **distinto** a variazioni anche minime del sistema di elaborazione (spesso anche solo al variare della versione del sistema operativo). Come mai?

- Una prima banale ragione può essere la **differenza nell'architettura hardware**: ad esempio, un file binario prodotto per CPU ARM non può essere interpretato da un sistema di elaborazione con CPU x86-64, dal momento che le istruzioni macchina delle due CPU differiscono
- A parità di architettura hardware, **sistemi operativi diversi possono supportare API diverse**: ad esempio, Windows supporta le API Win32 e Win64 e non le API POSIX, supportate da Linux e MacOS
- A parità di architettura e API, **sistemi diversi possono supportare diversi formati per i file binari**: ad esempio, Linux riconosce il formato ELF, mentre macOS riconosce il formato MachO
- A parità di architettura, formato ed API, può esservi una **differenza nelle chiamate di sistema che le implementano** (se la libreria API è **collegata staticamente**)
- A parità di architettura, formato ed API, anche se la libreria API è collegata dinamicamente (o le chiamate di sistema sono le stesse), **può esservi una differenza nell'ABI**
- Solo quanto tutti questi fattori sono identici, un file binario è portabile da un sistema di elaborazione ad un'altro

5.3 Struttura del Kernel

Il kernel è strutturato in **sottosistemi**, basati sulle categorie dei servizi offerti dal kernel stesso (e quindi sulle categorie delle chiamate di sistema). I principali sono:

- Gestione dei processi e dei thread
- Comunicazione tra processi e sincronizzazione
- Gestione della memoria
- Gestione dell'I/O
- File System

Il kernel di un sistema operativo **general-purpose** è un programma

- Di **dimensioni elevate e complesso**

- Che deve operare **molto rapidamente** per non sottrarre tempo di elaborazione ai programmi applicativi
- Un cui malfunzionamento può provocare il crash dell'intero sistema di elaborazione

Si pone quindi il problema di come progettarlo in maniera da garantire **rapidità e correttezza** nonostante dimensioni e complessità. Alcune possibilità di struttura sono:

- Struttura monolitica
- Struttura a strati
- Struttura a microkernel
- Struttura a moduli
- Struttura ibrida

5.3.1 Struttura monolitica

Il sistema operativo UNIX originale aveva una struttura monolitica, dove il kernel è un **singolo file binario statico**. Il kernel forniva un elevato numero di funzionalità:

- Scheduling della CPU
- File system
- Gestione della memoria, swapping, memoria virtuale
- Device drivers
- ...

I **vantaggi** di questa struttura sono **le sue elevate prestazioni**. Tuttavia, essa soffre di numerosi **svantaggi**:

- Complessità
- Fragilità ai bug
- Necessità di **ricompilare il kernel** (e riavviare il sistema) se bisogna aggiungere una funzionalità, ad esempio il driver di una nuova periferica

5.3.2 Struttura a strati

Negli approcci stratificati, il sistema operativo è diviso in un **insieme di livelli, o strati**. Lo strato più basso **interagisce con l'hardware**, lo strato n -esimo interagisce solo con lo strato $n - 1$ esimo. L'approccio offre **tre vantaggi**:

- Ogni strato può essere progettato e implementato **indipendentemente dagli altri strati**

- È possibile verificare la **correttezza degli strati indipendentemente da quella degli altri strati**
- Ogni strato **nasconde le funzionalità degli strati sottostanti** e presenta allo strato soprastante **una macchina dalle caratteristiche più astratte**

In realtà, pochi sistemi operativi usano questo approccio in maniera pura:

- È difficile **definire esattamente quali funzionalità devono avere uno strato**
- Ogni strato introduce **un overhead** che peggiora le prestazioni

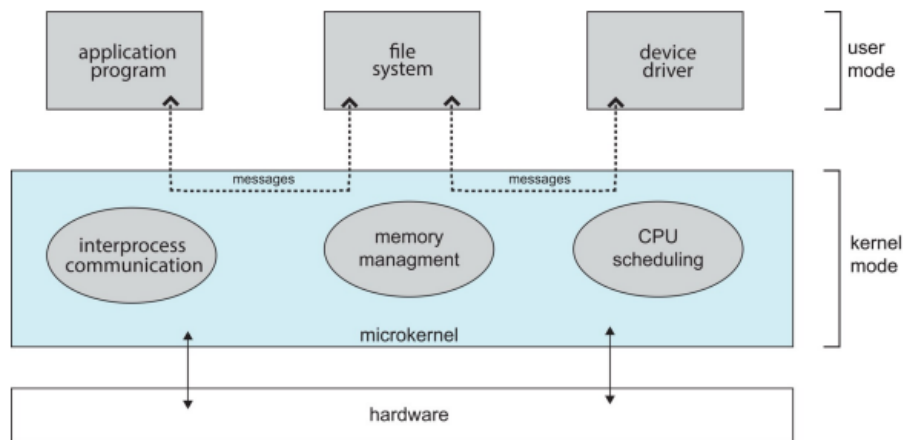
È comunque conveniente **strutturare alcune parti del sistema operativo a strati** (es. File System o stack di rete).

5.3.3 Struttura a microkernel

Il principale problema dei kernel monolitici è la loro complessità e, di conseguenza, la loro fragilità e inaffidabilità. La struttura a microkernel sposta **quanti più servizi possibile fuori dal kernel in programmi di sistema**, mantenendo nel kernel l'insieme **minimo** di servizi indispensabili per implementare gli altri. Il kernel è definito microkernel **dal momento che ha dimensioni molto ridotte**. L'approccio è stato proposto negli anni '80 con il sistema operativo Mach. Un microkernel offre **pochi servizi**, di solito

- Lo **scheduling dei processi**
- (Una parte della) **gestione della memoria**
- La **comunicazione tra processi**

Gli altri servizi (es. filesystem e device drivers) vengono implementati **a livello utente**. Per chiedere un servizio, un programma **comunica con il programma di sistema che lo implementa attraverso le primitive di comunicazione offerte dal microkernel**.



Vediamo i **vantaggi** di questa architettura:

- **Facilità di estensione del sistema operativo:** posso aggiungere un nuovo servizio senza dover modificare il kernel
- **Maggiore affidabilità:** se un servizio va in crash, non manda in crash il kernel; un kernel piccolo può essere quindi reso più affidabile con meno sforzo

vediamome quindi gli svantaggi:

- **Overhead:** una tipica richiesta di servizio deve **transitare dal processo richiedente al microkernel, al processo di sistema destinatario e viceversa**, con molti passaggi tra user e kernel mode, comunicazioni, context switching ecc...

I sistemi a microkernel puri vengono usati nelle applicazioni che **richiedono elevata affidabilità** (QNX neutrino, L4se). Altri sistemi, inizialmente a microkernel, si sono evoluti in **sistemi ibridi** (es. Windows NT, Darwin - kernel di macOS e iOS).

5.3.4 Struttura a moduli

In questa struttura, il kernel è **strutturato in componenti dinamicamente caricabili** (moduli), che parlano tra di loro **attraverso interfacce**. Quando il kernel ha bisogno di offrire un certo servizio, carica **dinamicamente** il modulo che lo implementa; quando il servizio non è più necessario, il kernel può scaricare il modulo. Questo approccio ha alcune **caratteristiche di quelli a strati e a microkernel**, ma i moduli **eseguono direttamente in kernel mode** e quindi con minore overhead (ma **anche con minore isolamento tra di loro**).

5.3.5 Sistemi ibridi