



RSO: Sistemi Operativi

spitfire

A.A. 2023-2024

Contents

1	Struttura e servizi	3
1.1	Componenti di un sistema di elaborazione	3
1.2	Requisiti per i sistemi operativi	4
1.3	La maledizione della generalità	4
1.4	Struttura dei sistemi operativi	5
1.5	Servizi offerti da un sistema operativo	6
1.6	Chiamate di sistema e Application Programming Interfaces	6
1.7	Programmi di sistema	7
1.7.1	Interfaccia utente: l'interprete dei comandi	8
1.7.2	Interfaccia utente: le interfacce grafiche	9
1.7.3	Interfaccia utente: Le interfacce touch-screen	10
1.8	L'implementazione dei programmi di sistema	10
2	Processi e thread: i servizi	11
2.1	Programmi e processi	11
2.2	Struttura di un processo	11
2.2.1	L'immagine di un processo	11
2.3	Operazioni sui processi	12
2.3.1	Creazione di processi	12
2.3.2	Terminazione di processi	13
2.4	API POSIX per le operazioni sui processi	13
2.5	Processi zombie e orfani	15
2.6	Comunicazione interprocesso	15
2.6.1	Modelli di IPC	15
2.6.2	IPC tramite memoria condivisa	15
2.6.3	IPC tramite message passing	16
2.6.4	Pipe	16
2.6.5	Notifiche con callback	17
2.7	API POSIX per l'IPC	17
2.7.1	Memoria condivisa in POSIX	17
2.7.2	Pipe anonime in POSIX	19
2.7.3	Named Pipes in POSIX	20
2.7.4	Segnali in POSIX	22

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**

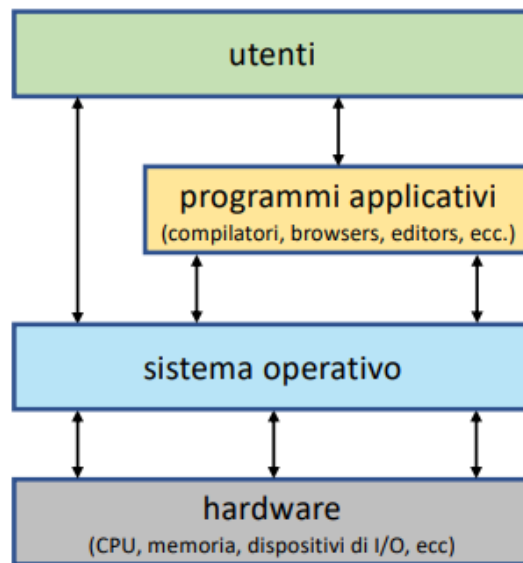
Un sistema operativo è quindi in primo luogo una **piattaforma di sviluppo**, ossia un insieme di funzionalità software che i programmi applicativi possono usare. Tali funzionalità permettono ai programmi di poter usare in maniera conveniente le risorse hardware di condividerle:

- 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

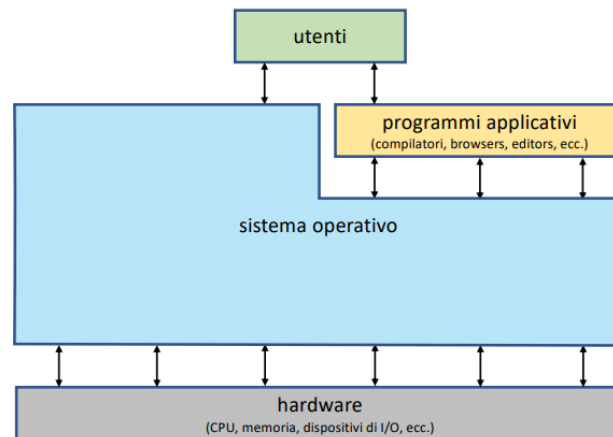
Nella storia (ed anche oggi) alcuni sistemi operativi sono stati utilizzati per scenari applicativi diversi. Ad esempio, Linux è usato oggi nei server, nei computer desktop e nei dispositivi mobili (come parte di Android). La **maledizione della generalità** afferma che, se un sistema operativo deve 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

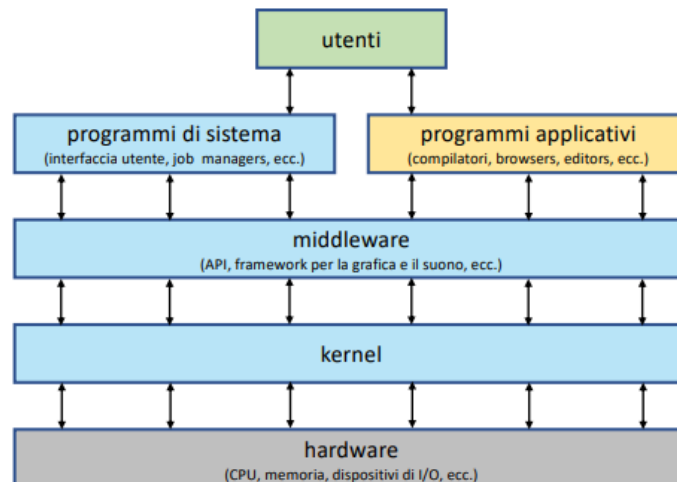
Non c'è una definizione universalmente accettata di quali programmi fanno parte di un sistema operativo. In generale però un sistema operativo almeno comprende:

- **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:

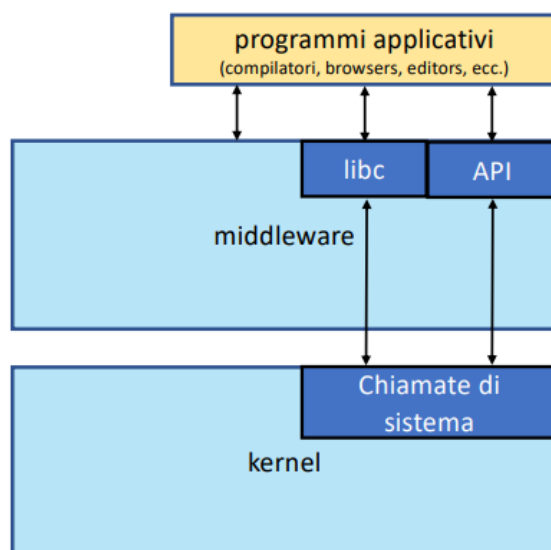


1.5 Servizi offerti da un sistema operativo

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`.

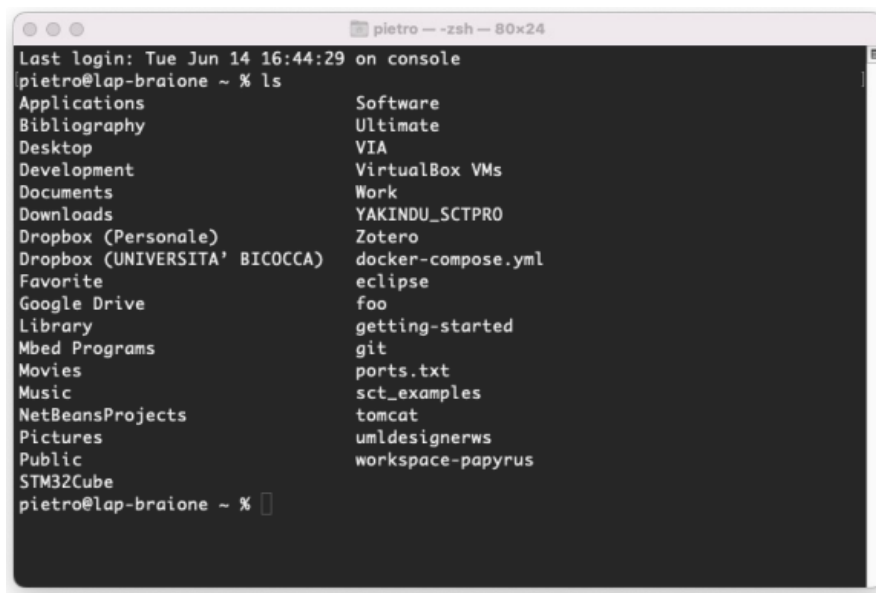
1.7 Programmi di sistema

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



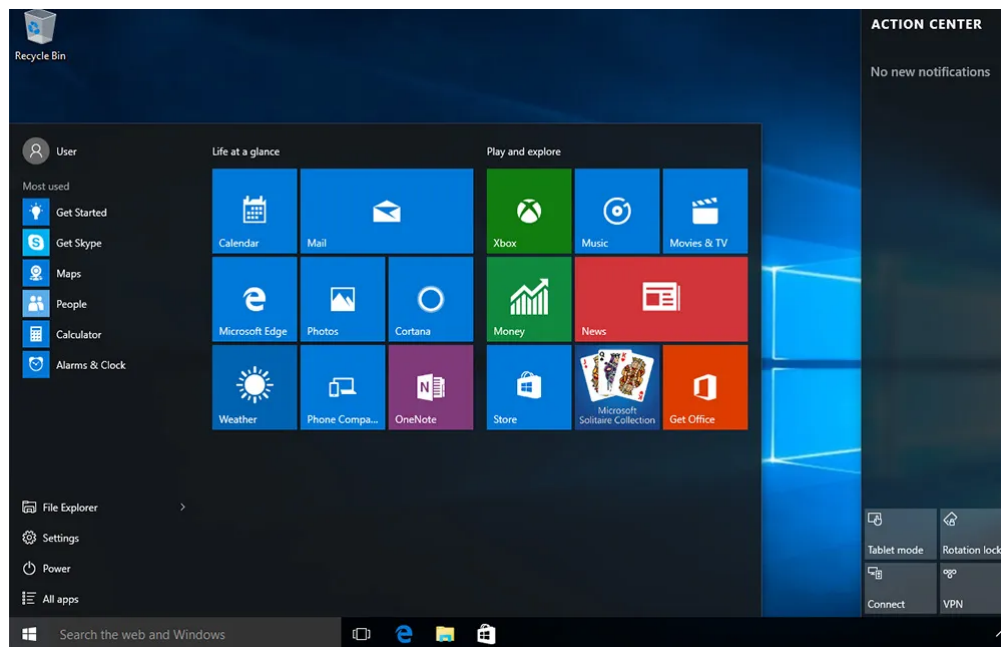
```
pietro -- -zsh -- 80x24
Last login: Tue Jun 14 16:44:29 on console
pietro@lap-braione ~ % ls
Applications          Software
Bibliography          Ultimate
Desktop              VIA
Development           VirtualBox VMs
Documents             Work
Downloads             YAKINDU_SCTPRO
Dropbox (Personale)   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)

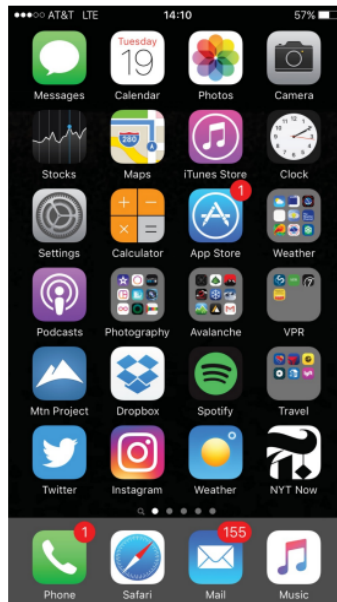
Spesso l'interprete riconosce **un vero e proprio linguaggio di programmazione** (es. Bash).

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

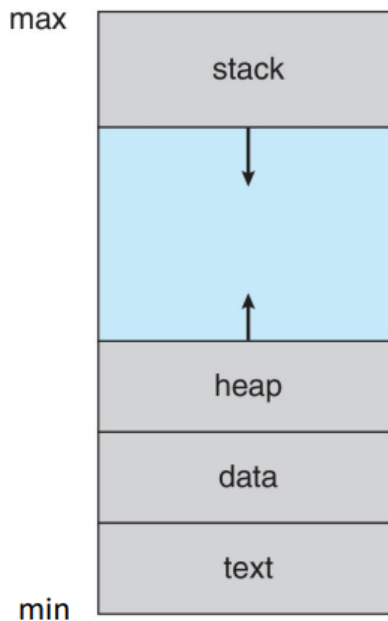
Notare che processi distinti hanno immagini distinte! Due processi operano su zone di memoria centrale separate! Le risorse del sistema operativo invece possono essere condivise tra processi (a seconda del tipo di risorsa)

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



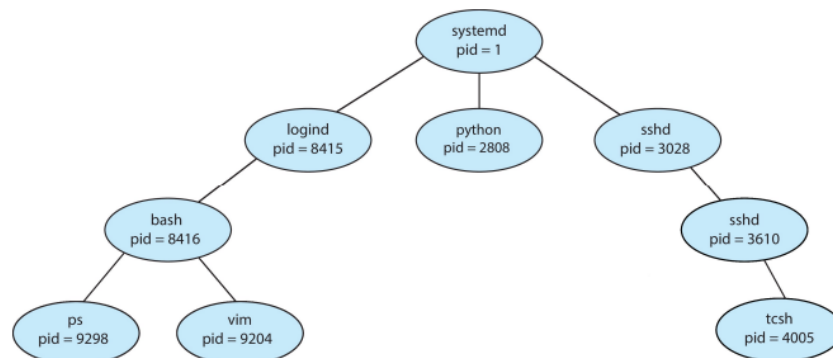
Text e data section hanno dimensioni costanti per tutta la vita del processo; mentre stack e heap invece crescono e decrescono durante la vita del processo.

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

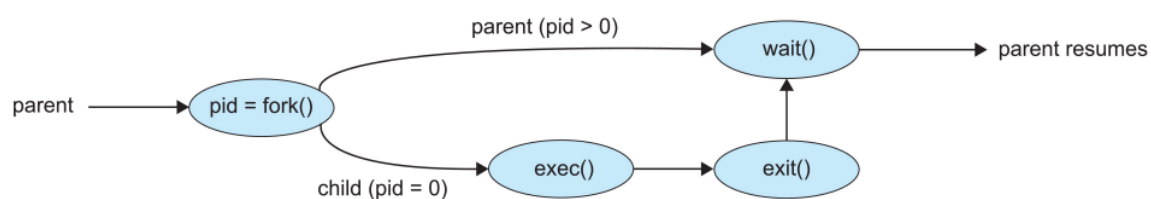
POSIX (Portable Operating System Interface for Unix) è una famiglia di standard specificata dalla **IEEE** per mantenere la compatibilità del software tra diversi sistemi operativi, in particolare tra le varianti di Unix (Linux e MacOS per esempio). In particolare definisce l'API disponibile (come la 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

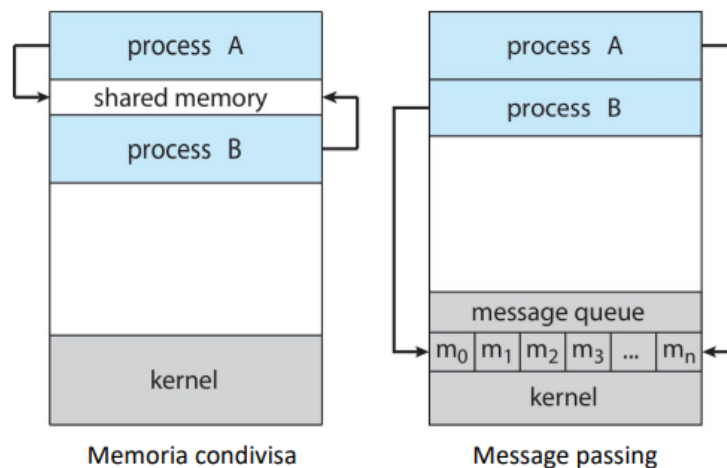
Più processi possono essere indipendenti o cooperare. Un processo coopera con uno o più altri processi se il suo comportamento "influenza" o "è influenzato da" il comportamento di questi ultimi. Vi sono più motivi per il quale si vogliono avere processi cooperanti:

- **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 **filedes** 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 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
```