

Sistemi Operativi

UniVR - Dipartimento di Informatica

Fabio Irimie

1° Semestre 2024/2025

Indice

1	Laboratorio	3
1.1	Shell	3
1.1.1	Comandi	3
1.1.2	File system	3
1.1.3	Comandi base	4
1.1.4	Processi	6
1.1.5	Redirezione dell'I/O	7
1.1.6	Variabili d'ambiente	8
1.1.7	Script	8
1.1.8	Programmi filtri	10
1.2	Processi e programmi	11
1.3	System call	13
1.3.1	Gestione degli errori	13
1.3.2	File	14
2	Sistema Operativo	16
2.1	Compiti del sistema operativo	16
2.1.1	Gestione delle risorse	16
2.1.2	Programma di controllo	16
2.2	Interrupt	17
2.2.1	Operazioni input/output	17
2.3	Multiprogrammazione	17
2.4	Protezione	18
2.4.1	Protezione I/O	18
2.4.2	Protezione della memoria	18
2.4.3	Protezione della CPU	18
2.5	Tipi di sistema operativo	18
3	Componenti di un sistema operativo	19
3.1	Gestione dei processi	19
3.2	Gestione della memoria primaria	20
3.3	Gestione della memoria secondaria	20
3.4	Gestione dell'I/O	20
3.5	Gestione dei file	20
3.6	Protezione	21
3.7	Rete	21
3.8	Interprete dei comandi	21
3.9	System call	22
3.10	Programmi di sistema	23
4	Architettura di un sistema operativo	23
4.1	Tipi di architetture	23
4.1.1	Sistemi monolitici	23
4.1.2	Sistemi a struttura semplice	23
4.1.3	Sistemi a livelli	24
4.1.4	Sistemi client-server	25

5	Programma e processo	25
5.1	Gestione dei processi	25
5.1.1	Stati di un processo	26
5.1.2	Scheduling	27
5.1.3	Creazione di un processo	28
5.2	Terminazione di un processo	29
5.3	Relazione tra processi	29
5.4	Thread	29
5.4.1	Vantaggi	30
5.4.2	Stati di un thread	30
5.4.3	Implementazione	30
5.5	Gestione dei processi del sistema operativo	31
5.5.1	Problematiche	32
6	Scheduling	32
6.1	Tipi di scheduling	33

1 Laboratorio

1.1 Shell

La shell è un interprete dei comandi che permette di comunicare con il sistema operativo. La shell principale di Linux è la **bash** (Bourne Again SHell) che è una shell che permette di eseguire comandi, script e programmi. Il funzionamento principale è il seguente:

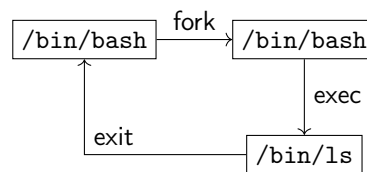


Figura 1: Funzionamento della bash

Le system call coinvolte sono:

- **fork**: Processo padre Bash crea un nuovo processo figlio
- **exec**: Processo figlio carica il codice del comando richiesto
- **exit**: Processo figlio termina
- **wait**: Processo padre Bash aspetta la terminazione del figlio

1.1.1 Comandi

La sintassi generale di un comando linux è:

```
1 comando [-opzioni] [argomenti]
```

Più comandi possono essere concatenati con il carattere `;`. Ad esempio:

```
1 comando1; comando2; ...; comandoN
```

Ogni comando linux ha un manuale che può essere consultato con il comando:

```
1 man <comando>
```

1.1.2 File system

Il file system è la struttura che permette di organizzare i file e le directory in un sistema operativo. Il file system di Linux è basato su una struttura ad albero con una radice `/` e permette di accedere ai file e alle directory tramite un **percorso** (path). Il path può essere di 2 tipi:

- **Path assoluto**: Parte dalla radice `/` del file system, ad esempio `/dir1/dir2`
- **Path relativo**: Parte dalla directory corrente, ad esempio `dir1/dir2`

Un path può anche contenere dei caratteri speciali, chiamati **wildcard**:

- * Qualsiasi stringa di caratteri
- ? Qualsiasi singolo carattere
- [] Qualsiasi carattere tra le parentesi

In linux ogni cosa è un file e i file sono di diversi tipi:

- **Directory**: Contiene altri file e sotto-directory
- **Special file**: È un entry point per un dispositivo di I/O, ad esempio /dev/sda
- **Link**: Puntatore ad un altro file. Si divide in:
 - **Hard link**: Puntatore diretto al file
 - **Symbolic link** (o soft link): Puntatore indiretto al file, quindi solo al percorso del file. Se la destinazione del link viene spostata, il link non funziona più
- **File ordinario**: Contiene dati

Ogni file ha associati dei **permessi** che definiscono chi può leggere (r, read), scrivere (w, write) e/o eseguire (x, execute) il file. I permessi sono divisi in 3 categorie:

- **User** (u): Il proprietario del file
- **Group** (g): Il gruppo del file
- **Others** (o): Tutti gli altri

1.1.3 Comandi base

```
• ls [-options] <path>
```

Mostra i file e le directory presenti nel **path** specificato.

```
• cat <file>
```

Stampa il contenuto del file specificato su standard output.

```
• head [-n] <file>
```

Visualizza le prime **n** righe del file.

```
• tail [-n] <file>
```

Visualizza le ultime **n** righe del file.

```
• cp <file> <dest>
```

Copia il file nella destinazione **dest**.

```
• mv <file> <dest>
```

Sposta il file nella destinazione **dest**. Se la destinazione è la stessa del file allora il file viene rinominato.

● `rm [-rf] <path>`

Rimuove il file o la directory specificata. L'opzione `-r` permette di rimuovere una directory in modo ricorsivo, cioè anche il contenuto della directory. L'opzione `-f` permette di rimuovere senza chiedere conferma.

● `cd <path>`

Cambia la directory corrente con quella specificata.

● `pwd`

Stampa il percorso della directory corrente.

● `mkdir <dir>`

Crea una directory con il nome specificato.

● `rmdir <dir>`

Rimuove la directory specificata se è vuota.

● `chgrp [-R] <group> <file>`

Cambia il gruppo del file specificato. L'opzione `-R` permette di cambiare il gruppo ricorsivamente.

● `chown [-R] <user:group> <file>`

Cambia il proprietario (e il gruppo) del file specificato.

● `chmod [-R] <permessi> <path>`

Cambia i permessi del file o della directory specificata. I permessi possono essere specificati in 3 modi:

- **Notazione numerica** (ottale): Ad esempio 777 che corrisponde ai permessi `rw-rw-rw-`. La prima cifra corrisponde ai permessi dell'utente, la seconda ai permessi del gruppo e la terza agli altri. I permessi sono:

- * `r` (read): 4
- * `w` (write): 2
- * `x` (execute): 1

e la somma di ogni singolo permesso corrisponde ai permessi totali, ad esempio:

$$7 = 4 + 2 + 1 = \text{rwx}$$

- **Notazione simbolica**: Ad esempio `ugo+r` che aggiunge il permesso di lettura a user, group e others
- **Notazione mista**: Ad esempio `u=rwx,g=rx,o=r` che assegna i permessi `rw-` all'utente, `rx` al gruppo e `r` agli altri

● `find <path> [-name <pattern>]`

Cerca i file e le directory nel path specificato e ritorna i file che rendono vera l'espressione specificata in pattern. Ad esempio:

```
1 find / -name "*.txt"
```

cerca tutti i file con estensione .txt partendo dalla radice.

```
• diff [-options] <file1> <file2>
```

Mostra le differenze tra i due file.

1.1.4 Processi

Un processo è un'istanza di un programma in esecuzione (un file binario). Per vedere la lista dei processi in esecuzione si può usare il comando:

```
1 ps
2
3      PID TTY          TIME CMD
4 11877 pts/1    00:00:00 bash
5 12110 pts/1    00:00:00 ps
```

Dove:

- PID: Process ID, l'identificativo del processo
- TTY: Il terminale associato al processo
- TIME: Il tempo di CPU usato dal processo
- CMD: Il comando che ha generato il processo

I processi possono essere eseguiti in due modi:

- **Foreground:** Il processo viene eseguito in primo piano e il terminale rimane bloccato finché il processo non termina. In questa modalità il processo ha accesso a tre canali standard connessi al terminale:

- stdin: Standard input
- stdout: Standard output
- stderr: Standard error

Un processo in foreground può essere sospeso con la combinazione di tasti `Ctrl + Z (^Z)` e può essere ripreso con il comando `fg`.

Un processo può essere interrotto con la combinazione di tasti `Ctrl + C (^C)`.

- **Background:** Il processo viene eseguito in secondo piano e il terminale rimane libero. In questa modalità il processo non ha accesso ai canali standard. Per eseguire un processo in background si può usare il carattere `&` alla fine del comando, ad esempio:

```
1 ls &
```

I comandi principali per gestire i processi sono:

```
• jobs [-l]
```

Elenca i processi in background o sospesi.

- `bg [%<job>]`

Riattiva un processo sospeso in background.

- `fg [%<job>]`

Riattiva un processo sospeso in foreground.

- `kill [-signal] <PID>`

Manda un segnale al processo indicato. I più comuni sono:

- SIGKILL (9): Termina il processo
- SIGTERM (15): Termina il processo in modo pulito

1.1.5 Redirezione dell'I/O

La redirezione dell'I/O permette di ridirezionare i canali standard di un processo su un file o su un altro canale.

Per redirezionare l'I/O si possono usare i seguenti operatori:

- `<`: Redireziona lo standard input, ad esempio:

```
1 cat < file.txt
```

- `>`: Redireziona lo standard output, ad esempio:

```
1 ls > file.txt
```

- `>>`: Redireziona lo standard output in append mode (aggiunge il contenuto al file senza sovrascriverlo), ad esempio:

```
1 ls >> file.txt
```

- `[n]>`: Redireziona il canale `n`, dove `n` è un numero che rappresenta:

- 0: Standard input
- 1: Standard output
- 2: Standard error

Ad esempio:

```
1 ls 1> file.out 2> file.err
```

- `2>&1`: Redireziona lo standard error su standard output

- `|` (Pipe): Permette di concatenare più comandi in modo che l'output di un comando venga usato come input per il comando successivo, ad esempio:

```
1 ls | cat
```


1.1.6 Variabili d'ambiente

La shell ha un insieme di variabili d'ambiente, cioè delle variabili globali che contengono informazioni sul sistema. Ogni variabile rispetta la sintassi:

```
1 nome=valore
```

Le variabili d'ambiente principali sono:

- PWD: Path corrente del file system
- SHELL: Il percorso della shell corrente (ad esempio /bin/bash)
- USER: Il nome dell'utente
- HOME: La directory home dell'utente
- PATH: La lista delle directory in cui cercare i comandi (binari)

Per leggere le variabili d'ambiente si possono usare i seguenti comandi:

```
• printenv <var>
```

Stampa il valore della variabile d'ambiente specificata.

```
• env
```

Stampa tutte le variabili d'ambiente.

```
• echo ${var}
```

Stampa il valore della variabile d'ambiente specificata.

1.1.7 Script

Uno script è una lista di comandi di sistema eseguiti sequenzialmente dalla shell. Gli script sono salvati in file di testo con estensione .sh e devono avere come prima riga:

```
1 #!/bin/bash
```

che indica il percorso dell'interprete da usare per eseguire lo script, in questo caso la bash.

Per eseguire uno script:

- Si può usare il comando:

```
1 bash <script> [args]
```

che esegue lo script con la bash

- Si può rendere lo script eseguibile con il comando:

```
1 chmod +x <script>
```

e poi eseguirlo direttamente con:

```
1 ./<script> [args]
```

La bash memorizza gli argomenti della linea di comando dentro una serie di variabili speciali:

- \$0: Il nome dello script
- \$1, \$2, ...: Gli argomenti passati allo script
- \$*: Tutti gli argomenti passati allo script
- \$@: Tutti gli argomenti passati allo script
- \$? : Indica se il comando precedente è stato eseguito correttamente:
 - 0: Comando eseguito correttamente
 - 1: Comando non eseguito correttamente
- \$#: Il numero di argomenti passati allo script

Variabili:

Le variabili in uno script si dichiarano come: `nome=valore`, ma per accedere al valore della variabile si deve usare il simbolo di dollaro prima del nome. Ad esempio:

```
1 #!/bin/bash
2 nome="Fabio"
3 echo $nome
4
5 # Output:
6 # Fabio
```

Per acquisire input dallo standard input si può usare il comando `read`:

```
1 read <var>
```

ad esempio:

```
1 read x
2 < Fabio
3 echo $x
4 > Fabio
```

Tutte le variabili in bash sono stringhe, quindi per fare operazioni aritmetiche si deve specificare di volerlo fare con l'operatore `$(())` che permette di valutare un'espressione, ad esempio:

```
1 x = 0
2 echo $x+1
3 > 0+1
4 echo $((x+1))
5 > 1
```

È possibile utilizzare l'output di un comando come valore di inizializzazione di una variabile, per fare ciò si usa l'operatore `$()`:

```
1 lista_file=$(ls)
2 # Ora la variabile lista_file contiene l'output del comando ls
```

Un altro modo per fare la stessa cosa è usare le backticks:

```

1 lista_file='ls'
2 # Ora la variabile lista_file contiene l'output del comando ls

```

Condizionali:

Per fare un'istruzione condizionale si usa la seguente sintassi:

```

1 if [ <condizione> ]; then
2     <comandi>
3 fi

```

Oppure:

```

1 if [ <condizione> ]; then
2     <comandi>
3 elif [ <condizione> ]; then
4     <comandi>
5 else
6     <comandi>
7 fi

```

Cicli:

Per fare un ciclo for si usa la seguente sintassi:

```

1 for var in <lista>; do
2     <comandi>
3 done

```

Per fare un while si usa la seguente sintassi:

```

1 while [ <condizione> ]; do
2     <comandi>
3 done

```

Funzioni:

Per definire una funzione si usa la seguente sintassi:

```

1 function nome_funzione {
2     <comandi>
3 }

```

Oppure:

```

1 nome_funzione() {
2     <comandi>
3 }

```

Per chiamare una funzione si usa il nome della funzione:

```

1 nome_funzione

```

1.1.8 Programmi filtri

Sono programmi che ricevono dati di ingresso da stdin e generano risultati su stdout. Alcuni comandi più usati sono:

- grep, fgrep, egrep: Filtra le righe di testo che contengono una stringa specificata
- sort: Ordina le righe di testo
- uniq: Rimuove le righe duplicate

- **wc**: Conta le righe, le parole e i caratteri
- **sed**: Sostituisce stringhe di testo
- **awk**: Filtra e trasforma il testo
- **cut**: Estrae colonne di testo
- **more**, **less**: Visualizza il testo a schermo

1.2 Processi e programmi

Un **processo** è un'istanza di un programma in esecuzione, mentre un **programma** è un file binario che contiene un set di informazioni che descrive come costruire un processo in runtime (cioè quando il programma viene eseguito).

Dal punto di vista del kernel un processo è composto da:

- Memoria user-space che contiene il codice del programma
- Le variabili utilizzate dal codice
- Un insieme di strutture dati del kernel che mantengono informazioni riguardo allo stato del processo, ad esempio:
 - Tabella di file aperti
 - Tabelle di pagine
 - Segnali da inviare
 - Limiti e utilizzo delle risorse del processo

Alla creazione di un processo, esso viene caricato in memoria contenendo i seguenti campi:

- **Codice del programma**: Una sezione di sola lettura che contiene sottoforma di testo le istruzioni del programma (text)
- **Dati inizializzati**: Una sezione che contiene le variabili globali e statiche inizializzate (data)
- **Dati non inizializzati**: Una sezione che contiene le variabili globali e statiche non inizializzate (bss)
- **Heap**: Una sezione che contiene le variabili allocate dinamicamente. Nell'architettura x86 il heap cresce verso indirizzi di memoria più alti
- **Stack**: Una sezione che contiene le variabili locali e i parametri delle funzioni. Nell'architettura x86 lo stack cresce verso indirizzi di memoria più bassi

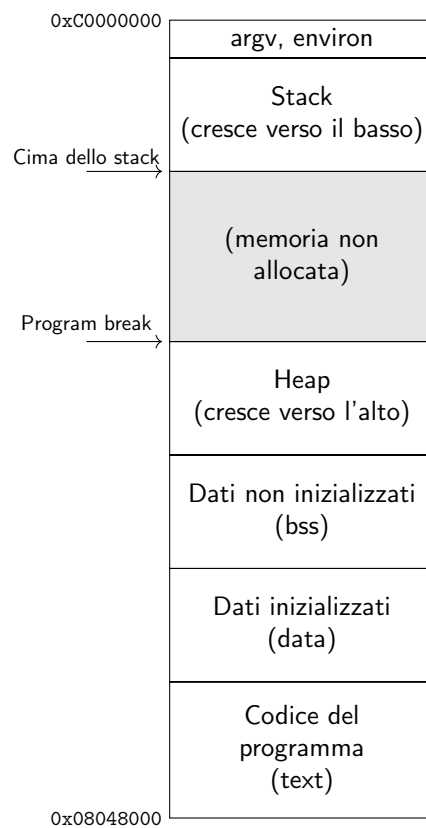


Figura 2: Struttura di un processo

Esempio 1.1. Il seguente programma mostra dove vengono allocate le varie sezioni di un processo:

```

1 #include <stdlib.h>
2 // Variabili dichiarate globalmente
3 char buffer[10]; // (bss)
4 int primes[] = {2, 3, 5, 7}; // (data)
5
6 // Implementazioni delle funzioni
7 void method(int *a) { // (stack)
8     int i; // (stack)
9     for (i = 0; i < 10; i++) {
10         a[i] = i;
11     }
12 }
13
14 // Entry point del programma
15 int main (int argc, char *argv[]) { // (stack)
16     static int key = 123; // (data)
17     int *p; // (stack)
18     p = malloc(10 * sizeof(int)); // (heap)
19

```

```

20     method(p);
21     free(p);
22
23     return 0;
24 }

```

Per vedere la grandezza delle varie sezioni si può usare il comando `size`:

```

1 gcc -o program program.c
2 size program
3      text    data     bss       dec       hex    filename
4      1017     260      12      1289       509     program

```

Per ogni processo il kernel salva una **file descriptor table**, che è una tabella che contiene i file aperti dal processo. Ogni riga della tabella contiene un **file descriptor**, cioè un intero positivo che rappresenta una risorsa di input/output aperta dal processo, ad esempio files, pipes, sockets, ecc.

Per convenzione ci sono sempre 3 file descriptor in un nuovo processo:

File descriptor	Descrizione	Nome POSIX
0	Standard input (stdin)	STDIN_FILENO
1	Standard output (stdout)	STDOUT_FILENO
2	Standard error (stderr)	STDERR_FILENO

Tabella 1: File descriptor standard

1.3 System call

Le system call sono delle funzioni che permettono ai processi di interagire con il kernel permettendo di accedere ai servizi offerti dal kernel, come ad esempio:

- Creare un nuovo processo
- Leggere e scrivere file
- Creare un socket
- ecc...

Richiamare una system call è simile ad una chiamata di funzione, però ciò che viene richiamato è solo un wrapper che passa i parametri alla system call effettiva che viene eseguita in kernel mode dopo aver ricevuto un interrupt software. Una volta che la system call è terminata, il controllo ritorna al processo chiamante in modalità utente.

1.3.1 Gestione degli errori

Le system call ritornano un valore intero che rappresenta il risultato dell'operazione. Quando una system call fallisce ritorna -1, oppure un NULL pointer. Successivamente il valore di errore viene salvato all'interno di una variabile globale chiamata `errno` che contiene un intero positivo che rappresenta l'errore che si è verificato. (Bisogna includere la libreria `<errno.h>` che fornisce una dichiarazione della variabile `errno` e delle funzioni per gestire gli errori).

Esempio 1.2. Un esempio di come gestire gli errori in una system call è il seguente:

```
1 // System call per aprire un file
2 fd = open(pathname, flags, mode);
3
4 // Gestione degli errori
5 if (fd == -1) { // La system call ha fallito
6     if (errno == EACCES) { // Errore di accesso al file
7         // Gestione dell'errore di accesso
8     } else {
9         // Altro tipo di errore
10    }
11 }
```

Per vedere il significato dell'errore si può usare la funzione `perror` che stampa su `stderr` un messaggio passato come parametro, seguito da un messaggio che descrive l'ultimo errore riscontrato da una system call

Per vedere il significato dell'errore si può usare la funzione `strerror` che ritorna una stringa che descrive l'errore passato come parametro. Esiste una tabella degli errori che associa un codice di errore ad una stringa che lo descrive.

Per vedere quali system call sono state chiamate da un processo si può usare il comando `strace`:

```
1 strace <program> [args]
```

1.3.2 File

Le principali system call per gestire i file sono:

- Apre un file esistente o ne crea uno nuovo se viene specificato il flag adeguato. Se va a buon fine ritorna il file descriptor del file aperto, altrimenti ritorna -1

```
1 #include <sys/stat.h>
2 #include <fcntl.h>
3
4 int open(const char *pathname, int flags, .../*mode_t mode*/);
5
```

- `pathname`: Il percorso del file da aprire
- `flags`: I flag per aprire il file, ad esempio:
 - * `O_RDONLY`: Apre il file in sola lettura
 - * `O_WRONLY`: Apre il file in sola scrittura
 - * `O_RDWR`: Apre il file in lettura/scrittura
 - * `O_CREAT`: Crea il file se non esiste
 - * `O_APPEND`: Scrive alla fine del file
 - * `O_TRUNC`: Se il file esiste, lo svuota
- `mode`: I permessi del file se viene creato, ad esempio:
 - * `S_IRUSR`: Permessi di lettura per l'utente

- * S_IWGRP: Permesso di scrittura per il gruppo
- * S_IXOTH: Permesso di esecuzione per gli altri

- Legge da un file aperto e scrive i dati letti in un buffer. Ritorna il numero di byte letti, 0 se è arrivato alla fine del file, -1 se c'è stato un errore

```
1 #include <unistd.h>
2
3 ssize_t read(int fd, void *buf, size_t count);
4
```

- fd: Il file descriptor del file da leggere
- buf: L'indirizzo di memoria in cui scrivere i dati letti
- count: Il numero di byte da leggere

- Scrive su un file aperto i dati contenuti in un buffer. Ritorna il numero di byte scritti, -1 se c'è stato un errore

```
1 #include <unistd.h>
2
3 ssize_t write(int fd, const void *buf, size_t count);
4
```

- fd: Il file descriptor del file su cui scrivere
- buf: L'indirizzo di memoria da cui leggere i dati da scrivere
- count: Il numero di byte da scrivere

- Sposta il cursore di lettura/scrittura del file al byte specificato. Ritorna la nuova posizione del cursore, -1 se c'è stato un errore

```
1 #include <unistd.h>
2
3 off_t lseek(int fd, off_t offset, int whence);
4
```

- fd: Il file descriptor del file su cui spostare il cursore
- offset: Il numero di byte da spostare il cursore
- whence: Da dove iniziare a spostare il cursore, ad esempio:
 - * SEEK_SET: Inizia dall'inizio del file
 - * SEEK_CUR: Inizia dalla posizione corrente
 - * SEEK_END: Inizia dalla fine del file

- Chiude il file descriptor specificato. Ritorna 0 se va a buon fine, -1 se c'è stato un errore

```
1 #include <unistd.h>
2
3 int close(int fd);
4
```

- fd: Il file descriptor del file da chiudere

- Rimuove un "link" dal file specificato, se è l'ultimo link al file, il file viene rimosso. Ritorna 0 se va a buon fine, -1 se c'è stato un errore

```
1 #include <unistd.h>
2
3 int unlink(const char *pathname);
4
```

– pathname: Il percorso del file da rimuovere

(non può rimuovere le directory)

2 Sistema Operativo

Il **sistema operativo** è il livello del software che si pone tra l'hardware e gli utenti. E quindi il sistema operativo incapsula la macchina fisica. Per mettere in comunicazione l'utente e l'hardware solitamente si usano le **applicazioni**, ma quando si vuole accedere direttamente all'hardware si usano le interfacce utente, ad esempio:

1. **Interfaccia grafica** (GUI)
2. **Command line** (Terminale o Shell)
3. **Touch screen**

Gli obiettivi principali sono:

- Facilitare l'uso del computer
- Rendere efficiente l'utilizzo dell'hardware
- Evitare conflitti nell'allocazione delle risorse hardware e software

Questo rimuove la necessità di conoscere la struttura dell'hardware attraverso l'**astrazione** facilitando la programmazione.

2.1 Compiti del sistema operativo

2.1.1 Gestione delle risorse

Il sistema operativo deve gestire le risorse hardware, come ad esempio i dischi, la memoria, gli input/output e la CPU. Deve anche gestire le risorse software, come ad esempio i file, i programmi e la memoria virtuale.

2.1.2 Programma di controllo

Un altro compito del sistema operativo è quello di controllare l'esecuzione dei programmi e del corretto utilizzo del sistema.

2.2 Interrupt

Un **interrupt** è un segnale hardware che interrompe il normale flusso di esecuzione di un programma. Gli interrupt possono essere generati da:

- **Hardware:** Per esempio quando un dispositivo ha finito un'operazione
- **Software:** Per esempio quando un programma chiama una system call

Questo serve per permettere alla CPU di lavorare per più tempo.

2.2.1 Operazioni input/output

Per gestire gli input/output ad esempio si usano i **device driver** che sono programmi che permettono di programmare la periferica per comunicare con il sistema operativo. Finchè il dispositivo non ha finito l'operazione, la CPU esegue altri processi e quando riceve l'interrupt dal dispositivo che segnala la fine dell'operazione, la CPU interrompe il processo corrente e inizia a gestire l'interrupt.

- **Buffering:** Sovrapposizione di CPU e I/O dello **stesso** processo
- **Spooling:** Sovrapposizione di CPU e I/O di **diversi** processi

Lo spooling serve quando l'elaborazione dei dati letti da un processo è più veloce della lettura stessa, quindi si avrebbero dei tempi morti rimossi con lo spooling che permette di elaborare i dati, già pronti, letti da un altro processo.

2.3 Multiprogrammazione

È la possibilità di tenere caricati in memoria più programmi e di eseguirli in modo alternato. Questo permette di sfruttare al meglio la CPU e di ridurre i tempi morti dovuti all'I/O.

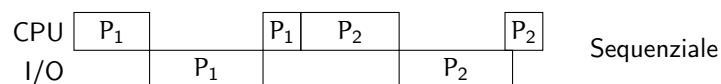


Figura 3: Senza multiprogrammazione

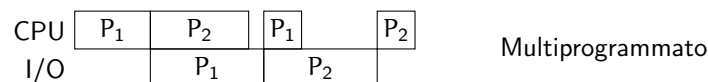


Figura 4: Con multiprogrammazione

Per eseguire più programmi alla volta, non soltanto quando un programma è in attesa di I/O, si utilizza il concetto di **time sharing** che permette di eseguire più programmi in modo alternato ad intervalli di tempo molto brevi chiamati **quanto di tempo** creando l'impressione che i programmi vengano eseguiti in parallelo.

Quando si hanno tanti processi che concorrono per l'utilizzo della CPU bisogna implementare delle regole per decidere quale processo eseguire. Queste regole sono definite dal sistema operativo, nello specifico dallo **scheduler**.

2.4 Protezione

La possibilità di eseguire più processi contemporaneamente crea dei problemi, come ad esempio la possibilità che un processo possa accedere ad un'altra area di memoria di un altro processo. Per evitare questo si usano delle tecniche di **protezione**:

- **Protezione I/O**: Programmi diversi non devono usare I/O contemporaneamente
- **Protezione della memoria**: Un processo non può leggere o scrivere in un'area di memoria che non gli appartiene
- **Protezione della CPU**: Un processo non può usare la CPU per più tempo di quello che gli è stato assegnato

In generale la protezione è realizzata tramite il meccanismo della **modalità duale** di esecuzione:

- **Modalità utente**: Il programma viene eseguito in modo normale, senza accesso alle risorse hardware
- **Modalità kernel**: Il programma viene eseguito con privilegi speciali che permettono di accedere all'hardware

2.4.1 Protezione I/O

Quando un processo vuole usare un dispositivo I/O, deve passare per il **device driver** che esegue le operazioni di I/O in modalità **supervisor** (o kernel mode). Questo viene fatto attraverso una **system call**, cioè un interrupt software che permette di passare dalla modalità utente alla modalità kernel per eseguire operazioni privilegiate. Al termine della system call si ritorna in modalità utente.

2.4.2 Protezione della memoria

Per proteggere la memoria si devono imporre dei limiti di accesso alla memoria ai processi. Il sistema operativo tiene traccia di questi limiti e li controlla ad ogni accesso alla memoria.

2.4.3 Protezione della CPU

Per proteggere la CPU bisogna garantire che il sistema operativo abbia sempre il controllo su di essa per poter interrompere qualsiasi processo che non restituisca il controllo della CPU. Per fare ciò si usano i **timer** che interrompono il processo se viene eseguito per troppo tempo.

2.5 Tipi di sistema operativo

Esistono diversi tipi di sistemi operativi che si differenziano e si classificano sulla base di determinate architetture di calcolo e determinati scopi. I principali tipi sono:

- **S.O. per PC e workstation**: Uso personale dell'elaboratore
- **S.O. di rete**: Separazione logica delle risorse remote e locali

- **S.O. distribuiti:** Non c'è la separazione logica tra risorse remote e locali, quindi l'accesso alle risorse remote viene effettuato nello stesso modo di quello alle risorse locali
- **S.O. real-time:** Vincoli sui tempi di risposta del sistema
- **S.O. embedded:** Per dispositivi con risorse limitate

3 Componenti di un sistema operativo

Le componenti principali di un sistema operativo sono:

- **Gestione dei processi**
- **Gestione della memoria primaria (RAM):** Spazio dei processi
- **Gestione della memoria secondaria (Memoria di massa):** Spazio dei programmi e dei dati
- **Gestione dell'I/O**
- **Gestione dei file**
- **Protezione**
- **Rete**
- **Interprete dei comandi**

3.1 Gestione dei processi

Un processo è l'istanza di un programma in esecuzione. Ogni processo ha bisogno di risorse per poter essere eseguito, come ad esempio la CPU, la memoria, i file e i dispositivi I/O. Il sistema operativo deve gestire i processi e assegnare le risorse per garantire che i processi vengano eseguiti in modo corretto e senza conflitti e deve gestire anche se stesso perché anche il sistema operativo è un'insieme di processi in modalità kernel.

Il sistema operativo è responsabile della:

- **Creazione e distruzione di processi:** Creare un processo significa allocare le risorse necessarie e inizializzare le strutture dati del processo; distruggere un processo significa rilasciare le risorse e liberare la memoria
- **Sospensione e riesumazione di processi:** Un processo viene sospeso quando deve aspettare un evento, come ad esempio un I/O
- **Sincronizzazione e comunicazione tra processi:** I processi devono poter comunicare tra di loro e sincronizzarsi per evitare conflitti

3.2 Gestione della memoria primaria

Un programma deve essere caricato in memoria per essere eseguito e nella RAM sono anche presenti dati condivisi tra CPU e dispositivi I/O.

Il sistema operativo è responsabile di:

- **Gestire lo spazio di memoria:** Decide quali parti della memoria allocare e a quali processi assegnarla
- **Decidere quale processo caricare in memoria quando esiste spazio disponibile**
- **Gestire l'allocazione e il rilascio dello spazio di memoria**

3.3 Gestione della memoria secondaria

La memoria secondaria è usata per memorizzare programmi e dati che non possono stare nella memoria primaria e per mantenere grandi quantità di dati in modo permanente. Il sistema operativo deve gestire l'ottimizzazione dell'uso della memoria secondaria.

Il sistema operativo è responsabile di:

- **Gestire lo spazio sulla memoria di massa**
- **Allocare spazio su memoria di massa**
- **Ordinare gli accessi ai dispositivi**

3.4 Gestione dell'I/O

Il sistema operativo deve creare un livello software che permetta all'utente di usare le periferiche senza doverne capire le caratteristiche fisiche.

Il sistema di I/O consiste di:

- **Sistema per accumulare gli accessi ai dispositivi:** Buffering, cioè l'accumulo di dati in memoria prima di essere scritti o letti dal dispositivo
- **Generica interfaccia verso i device driver**
- **Device driver specifici per alcuni dispositivi**

3.5 Gestione dei file

I file servono per memorizzare informazioni su supporti fisici diversi controllati da driver con caratteristiche diverse. Un file è l'**astrazione logica** per rendere conveniente l'uso della memoria non volatile e permette di raccogliere informazioni correlate, come ad esempio dati o programmi.

Il sistema operativo è responsabile di:

- **Creare e cancellare file e directory**

- **Fornire primitive per la gestione di file e directory:** Ad esempio aprire, leggere, scrivere, chiudere, rinominare, cancellare ecc...
- **Gestire la corrispondenza tra file e spazio fisico su memoria di massa:** Bisogna tenere traccia di dove sono memorizzati i file
- **Salvare informazioni a scopo di backup**

3.6 Protezione

È il meccanismo per controllare l'accesso alle risorse da parte di utenti e processi.

Il sistema operativo è responsabile di:

- **Definire accessi autorizzati e non**
- **Definire controlli per gli accessi**
- **Fornire strumenti per verificare le politiche di accesso**

3.7 Rete

Il sistema operativo deve gestire tutte le risorse di calcolo connesse tramite una rete.

Il sistema operativo è responsabile della gestione di:

- **Processi distribuiti**
- **Memoria distribuita**
- **File system distribuito**

3.8 Interprete dei comandi

L'interprete dei comandi è un programma che permette all'utente di comunicare con il sistema operativo. L'interprete dei comandi permette di:

- **Creare e gestire processi**
- **Gestire l'I/O**
- **Gestire il disco, la memoria, il file system**
- **Gestire le protezioni**
- **Gestire la rete**

Un interprete dei comandi è la **shell**, che è un programma che legge e interpreta i comandi.

3.9 System call

Sono un'interfaccia tra processi e sistema operativo e permettono ai processi di comunicare con il sistema operativo. Le system call sono chiamate tramite un'istruzione software e permettono di eseguire operazioni privilegiate. Le possibili opzioni per comunicare tra sistema operativo e processo sono:

1. Passare i parametri della system call tramite registri
2. Passare i parametri tramite lo stack del programma

```
1 // Programma utente
2 void main() {
3     ...
4     A(x);
5     ...
6 }
7
8 A(int x) {
9     ...
10    push x;
11    _A(); // Vera e propria system call
12    ...
13 }
14
15 _A() {
16     scrivi 13
17     TRAP // Interruzione software
18     ...
19 }
20
```

```
1 // Sistema operativo
2 Leggi 13
3 Salta al gestore 13
4
5 handler_13() {
6     ...
7 }
8
```

3. Memorizzare i parametri in una tabella in memoria

- L'indirizzo della tabella è passato in un registro o nello stack

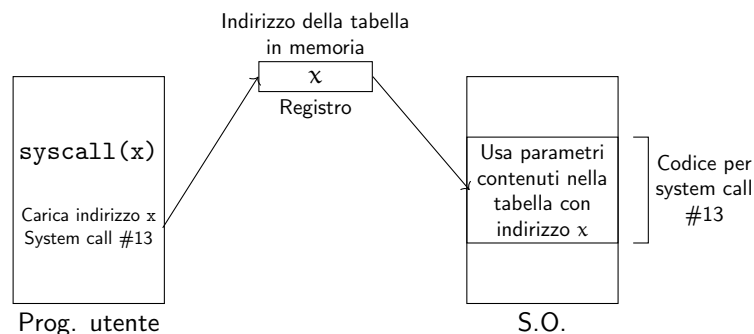


Figura 5: Passaggio dei parametri tramite tabella in memoria

3.10 Programmi di sistema

I programmi di sistema sono un'interfaccia comoda che ha l'utente delle operazioni che il sistema operativo può fare tramite le sue system call, ad esempio:

- **Gestione/manipolazione dei file:** Crea, copia, cancella...
- **Informazioni sullo stato del sistema:** Uso della CPU, data, spazio su disco...
- **Strumenti di supporto alla programmazione:** Compilatori, assembleri, editor...
- **Programmi di gestione della rete:** Login remoto, trasferimento file...
- **Formattazione documenti**
- **Mail**
- **Interprete dei comandi**
- **Utility varie**

4 Architettura di un sistema operativo

4.1 Tipi di architetture

4.1.1 Sistemi monolitici

I primi tipi di sistema operativo erano **monolitici**, cioè un unico programma che gestiva tutte le funzioni del sistema operativo e aveva tutti i componenti allo stesso livello, di conseguenza non c'era alcuna gerarchia e le funzioni potevano richiamarsi a vicenda.

Svantaggi:

- Codice dipendente dall'architettura hardware era distribuito su tutto il sistema operativo
- Difficile da testare, debuggare ed espandere

Svantaggi:

- Facile da implementare

4.1.2 Sistemi a struttura semplice

Si è cominciato a dividere il sistema operativo in livelli, implementando così una minima struttura gerarchica. Genericamente i livelli erano pochi e in alcuni casi anche bypassabili. Alcuni esempi di questi sistemi sono:

- UNIX
- MS-DOS

4.1.3 Sistemi a livelli

È un sistema operativo con più livelli che ha al livello più alto l'interfaccia utente e al livello più basso l'hardware. Tra l'utente e l'hardware ci sono diversi livelli **strettamente gerarchici** dove ogni livello svolge una funzionalità fornendo servizi al livello superiore e acquisendo servizi dal livello inferiore.

Vantaggi:

- Modularità, cioè facilità di sviluppo, test e debug senza impattare gli altri livelli

Svantaggi:

- Difficile definire appropriatamente i livelli
- Minor efficienza perchè ogni strato aggiunge **overhead** alle system call, cioè tempo di esecuzione aggiuntivo
- Minore portabilità, perchè funzionalità dipendenti dall'architettura sono sparse su vari livelli

Questa architettura a livelli crea l'idea di **macchina virtuale**, cioè un'astrazione dell'hardware tale da rendere trasparente la presenza di un hardware all'utente.

La macchina virtuale permette di effettuare un **multiplexing** (moltiplicazione) dell'hardware per permettere a più processi di accedere all'hardware contemporaneamente.

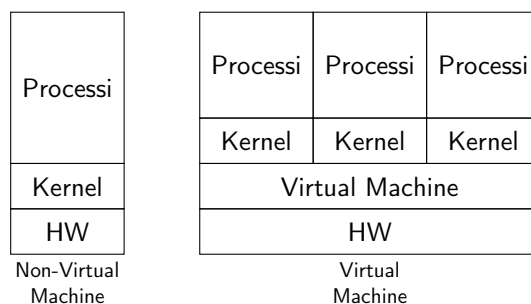


Figura 6: Differenza tra macchina virtuale e non virtuale

Vantaggi:

- Protezione completa del sistema: ogni VM è isolata dalle altre
- Più di un sistema operativo può essere eseguito sulla stessa macchina
- Ottimizzazione delle risorse: la stessa macchina può ospitare quello che senza VM doveva essere eseguito su macchine separate
- Ottime per lo sviluppo di sistemi operativi
- Buona portabilità

Svantaggi:

- Problemi di prestazioni
- Necessità di gestire dual mode virtuale: il sistema di gestione delle VM esegue in kernel mode, ma la VM esegue in user mode
- Ogni VM è isolata alle altre, quindi non c'è condivisione di risorse. Una possibile soluzione sarebbe condividere un volume del file system o definire una rete virtuale tra VM via software.

4.1.4 Sistemi client-server

I livelli non sono più gerarchici, ma sono paralleli e comunicano tra di loro tramite il kernel. Questo permette di avere un sistema operativo più flessibile e modulare.

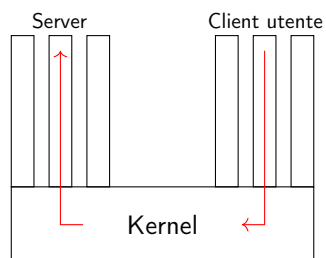


Figura 7: Sistema client-server

5 Programma e processo

- **Programma:** Un programma è un insieme di istruzioni e di dati contenuti in un file. È un'entità statica.
- **Processo:** Un processo è un'istanza di un programma in esecuzione. È un'entità dinamica.

5.1 Gestione dei processi

Un processo è un'**istanza di un programma in esecuzione** perché da un programma si possono creare più processi. Il processo è dinamico perché è in esecuzione, mentre invece il programma è statico perché è solo un insieme di istruzioni.

I processi evolvono **un'istruzione alla volta**, ma in un sistema multiprogrammato essi evolvono in modo **concorrente** perché hanno risorse fisiche e logiche limitate.

Un processo è un'**immagine in memoria** che consiste di:

- **Istruzioni** (Sezione di codice o testo)
 - Parte statica del codice
- **Dati** (sezione dati)
 - Variabili globali
- **Stack**

- Variabili locali
- Chiamate a procedure e parametri

- **Heap**

- Memoria allocata dinamicamente

- **Attributi** rappresentati dal PCB (Process Control Block):

- PID
- Stato del processo
- Program counter
- Valori dei registri
- Informazioni sulla memoria (registri limite, tabella pagine)
- Informazioni sullo stato dell'I/O (richieste pendenti, file)
- Informazioni sull'utilizzo del sistema (CPU)
- Informazioni di scheduling (priorità)

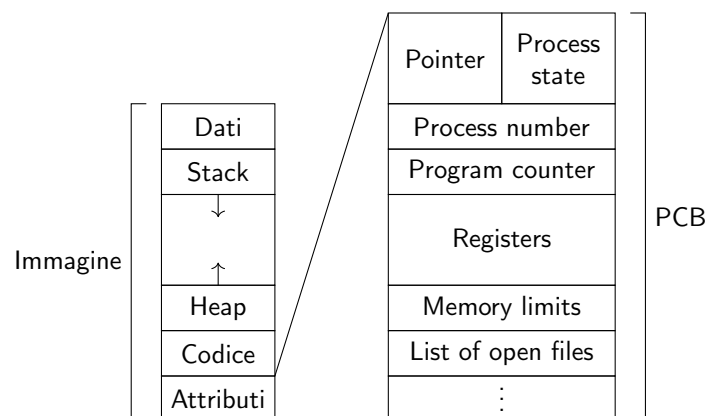


Figura 8: Immagine di un processo in memoria

5.1.1 Stati di un processo

Un processo durante la sua esecuzione può evolvere attraverso diversi stati:

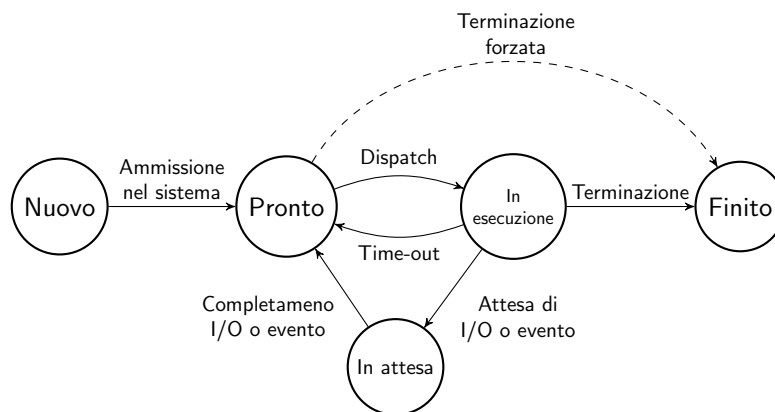


Figura 9: Stati di un processo

Il sistema operativo gestisce il passaggio tra gli stati del processo tramite:

- **Scheduler:** Decide quale processo eseguire
- **Dispatcher:** Cambia lo stato del processo che si chiama **context switch**, cioè il salvataggio dello stato del processo corrente in memoria (PCB) e il caricamento dello stato del nuovo processo. Il cambio di contesto è puro sovraccarico perchè la CPU non fa nulla mentre sta succedendo.

5.1.2 Scheduling

Lo **Scheduler** sceglie il processo da eseguire nella CPU al fine di garantire:

- **Multiprogrammazione:** con l'obiettivo di massimizzare l'uso della CPU caricando più di un processo in memoria
- **Time-sharing:** con l'obiettivo di commutare frequentemente la CPU tra processi in modo che ognuno creda di avere la CPU tutta per sé

Ogni processo è inserito in una serie di **code di scheduling**:

- **Ready queue:** Coda dei processi pronti ad essere eseguiti
- **Coda di un dispositivo:** Coda dei processi in attesa che il dispositivo si liberi

Ogni coda ha la propria **politica di scheduling**.

Gli stati di un processo possono anche essere rappresentati come un diagramma di accodamento:

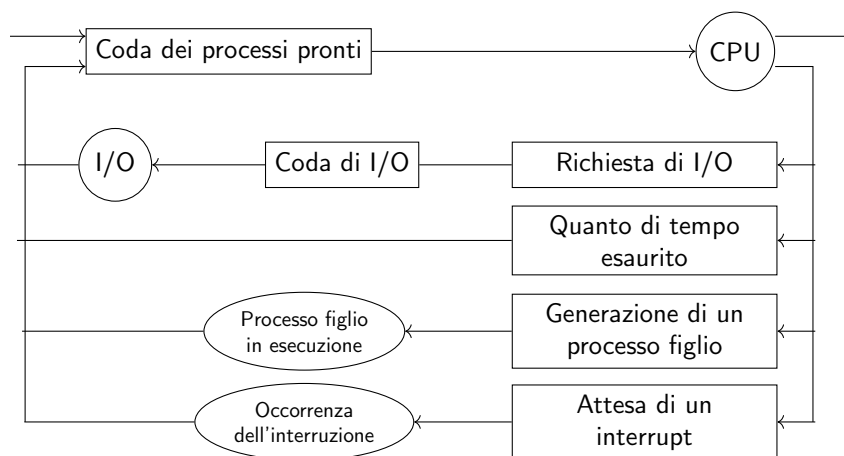


Figura 10: Diagramma di accodamento

5.1.3 Creazione di un processo

La creazione di un processo avviene per creazione da parte di un altro processo. Quando un computer viene acceso il primo processo creato è il **processo init** che è il padre di tutti i processi. La creazione di un processo avviene tramite la system call `fork()` che crea un processo figlio ed esso ottiene risorse dal sistema operativo o direttamente dal padre per:

- **Spartizione:** Il padre divide le risorse con il figlio
- **Condivisione:** Il padre e il figlio condividono le risorse

Un processo può essere eseguito in 2 modalità:

- **Sincrona:** Il padre aspetta che i figli terminino
- **Asincrona:** Avviene un'evoluzione **parallela** di padre e figli

Le system call principali per la creazione di un processo sono:

- `fork()`: Crea un processo figlio che è un duplicato esatto del padre
- `exec()`: Carica sul figlio un programma diverso da quello del padre
- `wait()`: Il padre aspetta che il figlio termini

Esempio 5.1. Un esempio in C della creazione di un processo figlio è il seguente:

```

1 #include <stdio.h>
2 void main(int argc, char *argv[]) {
3     int pid;
4     pid = fork(); // Genera un nuovo processo
5
6     if (pid < 0) {
7         printf("Errore di creazione\n");
8         exit(-1);
9     }
10 }
```

```

9  } else if (pid == 0) {
10 // Codice del processo figlio
11     execlp("/bin/ls", "ls", NULL); // Esegue il comando ls
12 } else {
13 // Codice del processo padre
14     wait(NULL); // Aspetta che il figlio termini
15     printf("Il figlio ha terminato\n");
16     exit(0);
17 }
18 }
19

```

Il padre riceve come pid il process id del figlio e il figlio riceve 0.

5.2 Terminazione di un processo

Un processo può terminare in diversi modi:

- **Terminazione volontaria:** Il processo termina volontariamente perchè ha finito la sua esecuzione
- **Terminazione forzata dal padre:** Il padre decide di terminare il processo figlio per più motivi:
 - Eccesso nell'uso delle risorse
 - Il compito richiesto al figlio non è più necessario
 - Il padre termina e il sistema operativo termina tutti i processi figli
- **Terminazione forzata dal S.O.:** Il sistema operativo termina il processo quando:
 - L'utente chiude l'applicazione
 - Ci sono errori durante l'esecuzione (aritmetici, di protezione, di memoria...)

5.3 Relazione tra processi

I processi possono essere di due tipi:

- **Processi indipendenti:** Sono processi che dipendono soltanto dal proprio input e quindi non influenzano altri processi. L'esecuzione di un processo indipendente è deterministica e riproducibile.
- **Processi cooperanti:** Sono processi che influenzano e sono influenzati da altri processi. Possono condividere dati e informazioni e possono comunicare tra di loro. L'esecuzione di un processo cooperante è non deterministica e non riproducibile.

5.4 Thread

Un **thread** è un flusso di esecuzione di un processo. Un processo può avere più thread e ogni thread lavora in modo indipendente sulle risorse del processo e ad essi è associato un Thread Control Block (TCB) che serve a memorizzare lo stato del thread.

- Ad un processo sono associati:
 - Spazio di indirizzamento
 - Risorse del sistema
- Ad ogni singolo thread sono associati:
 - Stato di esecuzione
 - Program counter
 - Insieme di registri della CPU
 - Stack

I thread condividono le risorse e lo stato del processo e lo spazio di indirizzamento. Il **Multithreading** è l'abilità di un sistema operativo di supportare più thread all'interno di un processo.

5.4.1 Vantaggi

I vantaggi del multithreading sono:

- **Riduzione del tempo di risposta:** Un thread può continuare a lavorare mentre un altro thread è bloccato
- **Condivisione delle risorse:** I thread condividono le risorse del processo senza dover introdurre tecniche esplicite di condivisione come avviene per i processi.
- **Economia:** I thread sono più economici dei processi perchè la loro creazione e distruzione e il loro context switch sono più veloci
- **Scalabilità:** Aumenta il parallelismo se l'esecuzione avviene su un multiprocessore

5.4.2 Stati di un thread

Anche i thread hanno uno stato di esecuzione, come i processi:

- **Pronto:** Il thread è pronto ad essere eseguito
- **In esecuzione:** Il thread è in esecuzione sulla CPU
- **Bloccato:** Il thread è bloccato in attesa di un evento

Lo stato di un thread può non coincidere con lo stato del processo.

5.4.3 Implementazione

I thread possono essere implementati come:

- **User-level thread:** Sono implementati dal programmatore e la gestione è affidata all'applicazione, di conseguenza il kernel ignora l'esistenza dei thread. I vantaggi sono:
 - Non serve passare alla modalità kernel per la gestione dei thread
 - Lo scheduling può variare da applicazione ad applicazione

- Sono portabili, cioè possono essere implementati su qualsiasi sistema operativo senza modificare il kernel

Gli svantaggi sono:

- Il blocco di un thread blocca l'intero processo
- Non è possibile sfruttare i processori multi-core

- **Kernel-level thread:** La gestione è affidata al kernel e le applicazioni usano i thread tramite system call. I vantaggi sono:

- Scheduling a livello di thread, cioè il blocco di un thread non blocca l'intero processo
- Più thread possono essere eseguiti in parallelo su processori diversi
- Le funzioni del sistema operativo possono essere multithreaded

Gli svantaggi sono:

- Scarsa efficienza perchè il passaggio da un thread all'altro richiede il passaggio attraverso il kernel

5.5 Gestione dei processi del sistema operativo

Il sistema operativo è un programma come tutti gli altri e quindi può generare più processi. Il sistema operativo può essere eseguito in diversi modi:

- **Kernel separato:** Il kernel viene eseguito al di fuori da ogni processo:
 - Il sistema operativo possiede uno spazio di memoria riservato
 - Il sistema operativo prende il controllo del sistema
 - Il sistema operativo è sempre in esecuzione in modo privilegiato

È un processo che viene applicato soltanto ai processi utente:

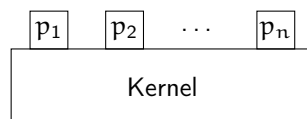


Figura 11: Kernel separato

- **Kernel in processi utente:** Il kernel è eseguito come un processo utente fornendo delle procedure richiamabili da programmi utente in modalità kernel. L'immagine dei processi (PCB) deve contenere anche:
 - Kernel stack per gestire il funzionamento del processo in modalità protetta (chiamate a funzione)
 - Codice/dati del S.O. condiviso tra processi utente

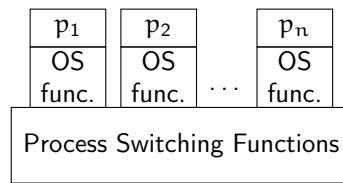


Figura 12: Kernel in processi utente

Il vantaggio è che basta cambiare la modalità di esecuzione per passare da utente a kernel che è più leggero di un context switch e dopo che il sistema operativo ha completato il suo lavoro può decidere se riattivare lo stesso processo utente (mode switch) o un altro (context switch)

- **Kernel come processo:** I servizi del sistema operativo sono processi individuali che si comportano come qualsiasi altro processo, solo che sono eseguiti in modalità kernel e non in modalità utente. Una minima parte del sistema operativo deve comunque eseguire al di fuori di tutti i processi (scheduler). È vantaggioso per i sistemi multiprocessore perchè i processi possono essere eseguiti in parallelo.

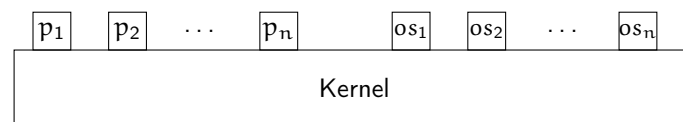


Figura 13: Kernel separato

5.5.1 Problematiche

I problemi principali sono:

- Allocazione delle risorse (CPU, memoria, spazio su disco) ai processi e ai thread.
- Coordinamento tra processi e thread (concorrenti):
 - Sincronizzazione
 - Comunicazione

6 Scheduling

Lo scheduling dei processi è il processo di selezione del processo da eseguire sulla CPU nel tempo. Serve perchè con l'utilizzo della multiprogrammazione bisogna implementare una strategia per regolamentare:

- Ammissione dei processi nel sistema (memoria)
- Ammissione dei processi all'esecuzione (CPU)

6.1 Tipi di scheduling

Lo scheduling gestisce in che modo viene ordinato l'accesso ad ogni coda di processi. Gli scheduler sono di due tipi:

- **Scheduler a lungo termine (job scheduler):** Seleziona quali processi devono essere portati dalla memoria alla redy queue. Questo scheduler è invocato molto spesso.
- **Scheduler a medio termine (swapper):** Seleziona quali processi devono essere spostati dalla memoria al disco e viceversa. Questo scheduler è invocato meno frequentemente.
- **Scheduler a breve termine (CPU scheduler):** Seleziona quale processo deve essere eseguito sulla CPU. Questo scheduler è invocato più raramente.