Sistemi Operativi

UniVR - Dipartimento di Informatica

Fabio Irimie

1° Semestre 2024/2025

Indice

1 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.

1.1 Compiti del sistema operativo

1.1.1 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.

1.1.2 Programma di controllo

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

1.2 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.

1.2.1 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.

1.3 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.

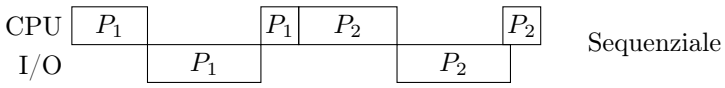

Figura 1: Senza multiprogrammazione

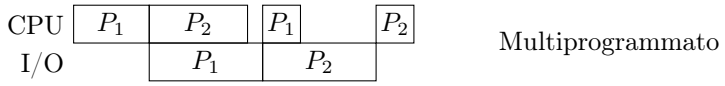

Figura 2: Con multiprogrammazione

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**.

1.4 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

1.4.1 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 esequire operazioni privilegiate. Al termine della system call si ritorna in modalità utente.

1.4.2 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.

1.4.3 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.

1.5 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

2 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**

2.1 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 perchè 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

2.2 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**

2.3 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**

2.4 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**

2.5 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 corrispondnza tra file e spazio fisico su memoria di massa**: Bisogna tenere traccia di dove sono memorizzati i file
- **Salvare informazioni a scopo di backup**

2.6 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**

2.7 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**

2.8 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.

2.9 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

```
// Programma utente
void main() {
  ...
  A(x);
  ...
}

A(int x) {
  ...
  push x;
  _A(); // Vera e propria system call
  ...
}

_A() {
  scrivi 13
  TRAP // Interruzione software
  ...
}
```

```
// Sistema operativo
Leggi 13
Salta al gestore 13

handler_13() {
  ...
}
```

3. Memorizzare i parametri in una tabella in memoria
   - L'indirizzo della tabella è passato in un registro o nello stack

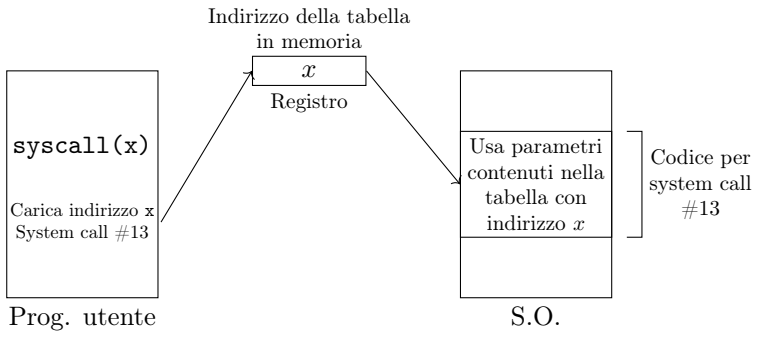

Figura 3: Passaggio dei parametri tramite tabella in memoria

2.10 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, assemblatori, editor...
- **Programmi di gestione della rete**: Login remoto, trasferimento file...
- **Formattazione documenti**
- **Mail**
- **Interprete dei comandi**
- **Utility varie**

3 Architettura di un sistema operativo

3.1 Tipi di architetture

3.1.1 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

3.1.2 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

3.1.3 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 hadrware all'utente.

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

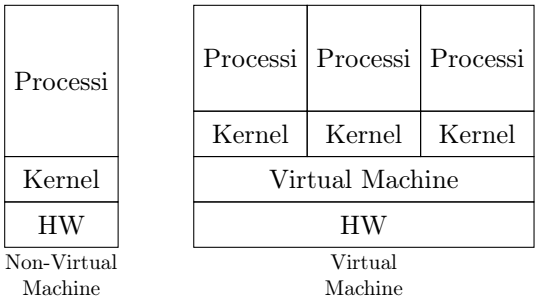

Figura 4: Differeza tra macchina virtuale e non virtuale

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.

3.1.4 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.

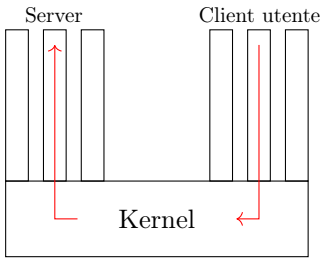

Figura 5: Sistema client-server

4 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.

4.1 Gestione dei processi

Un processo è un'**istanza di un programma in esecuzione** perchè da un programma si possono creare più processi. Il processo è dinamico perchè è in esecuzione, mentre invece il programma è statico perchè è solo un insieme di istruzioni.
I processi evolvono **un'istruzione alla volta**, ma in un sistema multiprogrammato essi evolvono in modo **concorrente** perchè 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** rappresentat 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)

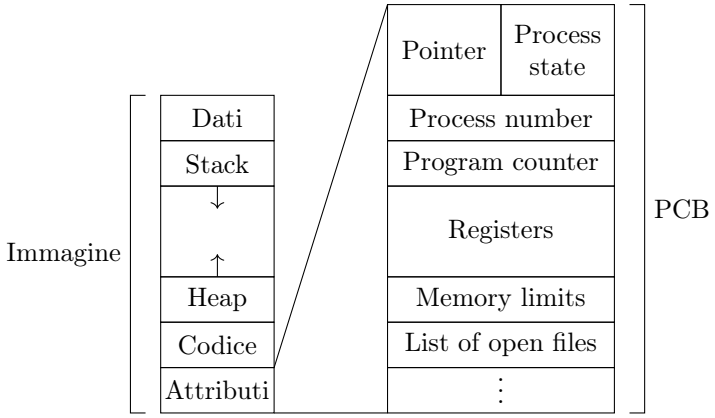

Figura 6: Immagine di un processo in memoria

4.1.1 Stati di un processo

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

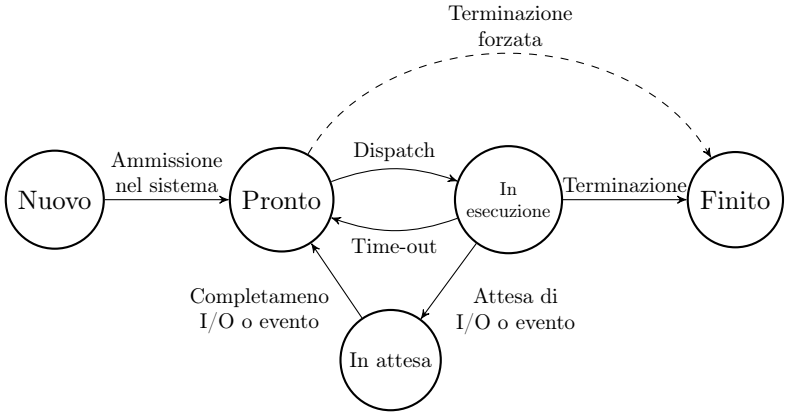

Figura 7: Stati di un processo

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.

4.1.2 Scheduling

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

- Multiprogrammazione: con l'obiettovo 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 sè

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:

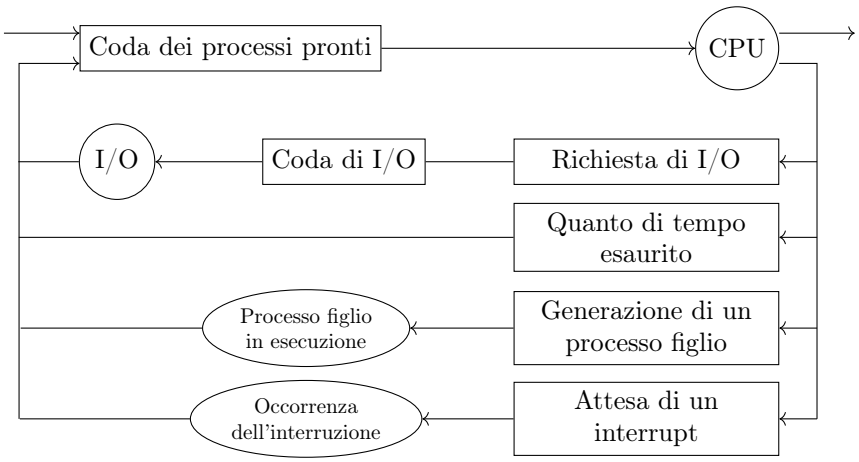

Figura 8: Diagramma di accodamento

4.1.3 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 figlii 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 4.1. Un esempio in C della creazione di un processo figlio è il seguente:

```c
#include <stdio.h>
void main(int argc, char *argv[]) {
  int pid;
  pid = fork(); // Genera un nuovo processo

  if (pid < 0) {
    printf("Errore di creazione\n");
    exit(-1);
  } else if (pid == 0) {
    // Codice del processo figlio
    execlp("/bin/ls", "ls", NULL); // Esegue il comando ls
  } else {
    // Codice del processo padre
    wait(NULL); // Aspetta che il figlio termini
    printf("Il figlio ha terminato\n");
    exit(0);
  }
}

```

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

4.2 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 operaivo 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...)

4.3 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.

4.4 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.

4.4.1 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

4.4.2 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.

4.4.3 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

4.5 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:

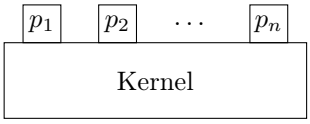

Figura 9: Kernel separato

- **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

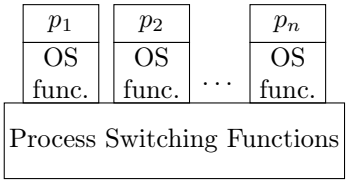

Figura 10: Kernel in processi utente

  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.

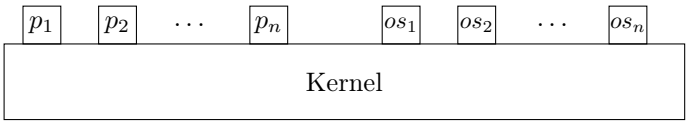

Figura 11: Kernel separato

4.5.1 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