Appunti di Sistemi Operativi

di:

Facchini Luca

Corso tenuto dal prof. Sistemi Operativi

Università degli Studi di Trento

A.A. 2024/2025

Autore:
Facchini Luca
Mat. 245965
Email:luca.facchini-1@studenti.unitn.it
luca@fc-software.it

Corso:
Sistemi Operativi [146065]
CdL: Laurea Triennale in Informatica
Prof. Crispo Bruno
Email: bruno.crispo@unitn.it

Sommario

Appunti del corso di Sistemi Operativi, tenuto dal prof. Crispo Bruno presso l'Università degli Studi di Trento. Corso seguito nell'anno accademico 2024/2025.
Dove non specificato diversamente, le immagini e i contenuti sono tratti dalle slide del corso del prof. Crispo Bruno (bruno.crispo@unitn.it)

Indice

Capitolo 1

Definizioni e Storia

Capitolo 2

Componenti di un sistema operativo

Dopo aver definito cosa sia un sistema operativo, vediamo ora quali siano le sue componenti principali, a partire dalla gestione dei processi e della memoria (primaria e secondaria), per poi passare alla gestione dei dell `I/O` e dei file fino ad arrivare alla protezione, la gestione della rete e l'interprete dei comandi.

2.1 Le Componenti in generale

Gestione dei Processi

Definizione 2.1 (Processo)**.** *Un **processo** è un programma in esecuzione che necessita di **risorse** per poter funzionare. Questo inoltre è eseguito in modo **sequenziale** ed **una istruzione alla volta**, infine è possibile che un processo sia del `SO` o dell'utente.*

In materia di gestione dei processi il sistema operativo è responsabile nella loro creazione e distruzione, nella loro sospensione e ripresa e deve fornire dei meccanismi per la sincronizzazione e la comunicazione tra i processi stessi.

Gestione della memoria primaria

Definizione 2.2 (Memoria primaria)**.** *La **memoria primaria** è la memoria principale del computer che conserva dati condivisi dalla `CPU` e dai dispositivi `I/O` questa è direttamente accessibile dalla `CPU`, per essere eseguito un programma deve essere caricato in memoria.*

La gestione della memoria primaria richiede la gestione dello spazi di memoria oltre alla decisione su quale processo debba essere caricato in memoria e quale debba essere rimosso. Inoltre il sistema operativo deve fornire dei meccanismi allocare e de-allocare la memoria.

Gestione della memoria secondaria

Definizione 2.3 (Memoria secondaria)**.** *La **memoria secondaria** è una memoria **non volatile** ed **grande** rispetto alla memoria primaria, questa è utilizzata per memorizzare i dati e i programmi in modo **permanente**.*

Questa memoria consiste di uno o più dischi (magnetici) ed il sistema operativo deve fornire dei meccanismi per la gestione dello spazio libero, l'allocazione dello spazio ed lo *scheduling* degli accessi ai dischi.

Gestione dell'`I/O`

Il `SO` nasconde la complessità dell'`I/O` ai programmi utente, fornendo un'astrazione dell'`I/O` e fornendo dei meccanismi per: accumulare gli accessi ai dispositivi (*buffering*), fornire una interfaccia generica per i dispositivi e fornire dei *driver* specifici (scritti in `C`, `C++` o *assembly*).

Gestione dei file

Definizione 2.4 (File)**.** *Un **file** è una sequenza di byte memorizzata in un qualsiasi supporto fisico controllato da driver del sistema operativo.*

Un file è dunque un'astrazione logica per rendere più semplice la memorizzazione e l'uso della memoria **non volatile**. Il sistema operativo deve fornire dei meccanismi per la creazione, la cancellazione, la lettura e la scrittura di file e *directory* oltre a fornire delle primitive (copia, sposta, rinomina) per la gestione dei file.

Protezione

Il sistema operativo deve fornire dei meccanismi per controllare l'accesso a tutte le risorse da parte di processi e utenti, inoltre l'`SO` è responsabile della definizione di accessi autorizzati e non autorizzati, oltre a definire i controlli necessari ed a fornire dei meccanismi per verificare le politiche di accesso definite.

2.2 Come usare i servizi dei sistemi operativi

Il sistema operativo metta a disposizione le sue interface tramite delle *system call* che sono delle chiamate a funzione che permettono di accedere ai servizi del sistema operativo precedentemente descritti. Queste chiamate a funzione sono utilizzate per eseguire operazioni che richiedono privilegi di sistema, come ad esempio la gestione dei processi, della memoria, dell'`I/O` e dei file.

2.2.1 Interprete dei comandi

Un esempio di utilizzo delle *system call* è l'interazione con l'interprete dei comandi, che permette di eseguire comandi e programmi tramite una interfaccia testuale. Questo interprete tramuta i comandi in *system call* che vengono poi eseguite dal sistema operativo. Questo permette di creare e gestire processi, gestire `I/O`, disco, memoria e file oltre alla gestione delle protezione e della rete.
Nel `SO` esistono dei comandi predefiniti che possono essere chiamati direttamente per il loro nome, questi sono implementati con una semantica specifica e possono essere utilizzati per eseguire operazioni di base, nel caso di comandi non predefiniti è possibile scrivere dei programmi che vengono eseguiti dall'interprete dei comandi.

2.2.2 L'interfaccia grafica

Un'altra interfaccia che permette di interagire con il sistema operativo è l'interfaccia grafica, che permette di interagire con il sistema operativo tramite il *mouse* e la tastiera. Questa interfaccia più intuitiva e facile da usare rispetto all'interprete dei comandi, permette di interagire con il `SO` tramite icone e finestre. Questa interfaccia, anche se più semplice, non è per forza più veloce dell'interprete dei comandi, in quanto l'interfaccia grafica è più lenta e richiede più risorse rispetto all'interprete dei comandi.

2.2.3 *System calls*

I processi non usano le *shell* per eseguire le *system call*, ma usano delle `API` (*Application Programming Interface*) che permettono di accedere ai servizi del sistema operativo. Queste `API` sono delle librerie di funzioni ad alto livello che permettono di accedere ai servizi del sistema operativo. Queste librerie sono scritte in `C` o `C++` e permettono di accedere ai servizi del sistema operativo in modo più semplice e più sicuro rispetto all'uso diretto delle *system call*.

Esempio di `API` Un esempio di `API` è la `Win32`, prendiamo in esame la funzione `ReadFile` che permette di leggere un file:

```
BOOL ReadFile (
    HANDLE file,
    LPVOID buffer,
    DWORD bytes to read,
    LPDWORD bytes read,
    LPOVERLAPPED ovl
);
```

Questa funzione ritorna un valore booleano che indica se la funzione è andata a buon fine o meno, inoltre questa funzione prende in input il file da leggere, il buffer in cui scrivere i dati letti, il numero di byte da leggere, il numero di byte letti e un puntatore a una struttura `OVERLAPPED` che permette di specificare un offset per la lettura.

Le `API` nei diversi `SO`

Le 2 `API` più comuni per Windows sono: `Win32` e `Win64` mentre per Linux sono: `POSIX` (*Portable Operating-System Interface*) che includono le *system call* per tutte le versioni di `UNIX`, *Linux* e *Mac OS X*, o tutte le distribuzioni `POSIX`-*compliant*.

Windows* su *Linux Per eseguire programmi *Windows* su *Linux* è possibile usare `Wine` che è un *emulatore* il quale traduce le chiamate `API` di *Windows* in chiamate `API` di *Linux on-the-fly*, ovvero durante l'esecuzione del programma. Questo permette di eseguire programmi *Windows* su *Linux* senza dover riscrivere il codice del programma.

Implementazione delle *System Call*

Ad ogni *system call* è associato un numero univoco, che permette al sistema operativo di identificare la *system call* richiesta. È compito dell'interfaccia tenere traccia dei numeri associati alle *system call* e di passare i parametri alla *system call* richiesta. Questa interfaccia invoca la *system call* nel *kernel* del sistema operativo, che esegue la *system call* e ritorna il risultato al chiamante. Questo meccanismo permette al chiamante di non dover conoscere i dettagli di implementazione della *system call* ma solo la sua interfaccia.

Esecuzione delle *system calls* Per eseguire una *system call* dopo che il processo ha eseguito la chiamata all'interfaccia del `SO` il quale conoscendo il numero della *system call* controlla dove questa è implementata tramite la *system call table* (una tabella che contiene i puntatori alla implementazione delle *system call*). Una volta trovata la *system call* il `SO` esegue la *system call* e ritorna il risultato al chiamante.

Opzioni per il passaggio dei parametri I parametri di una *system call* possono essere passati in diversi modi. I più comuni sono: passaggio tramite registri, passaggio tramite lo `stack` e passaggio tramite puntatori. Il passaggio tramite registri è il più veloce ma permette di passare pochi parametri e di piccola dimensione, il passaggio tramite lo `stack` permette di passare più parametri e di dimensioni maggiori, infine il passaggio tramite puntatori permette di passare parametri di dimensioni maggiori e di passare parametri complessi, ma và passata una tabella di parametri che deve essete passata tramite `stack` o registri.

Parametri tramite `stack`Il passaggio dei parametri tramite `stack` avviene in questo modo:

1-3 Salvataggio parametri sullo `stack`

4 Chiamata della funzione di libreria

5 Caricamento del numero della *system call* su un registro `Rx`

6 Esecuzione `TRAP` (Passaggio in *kernel mode*)

7-8 Esecuzione della *system call*

9 Ritorno al chiamante

10-11 Ritorno al codice utente ed incremento dello `stack pointer`

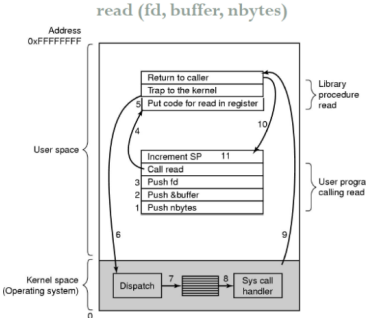

Figura 2.1: Passaggio dei parametri tramite `stack`

Passaggio di parametri tramite tabella Come anticipato il passaggio di parametri tramite tabella viene utilizzato per passare parametri complessi o di dimensioni maggiori andando a passare un puntatore alla tabella che contiene i parametri. Questo metodo permette di passare un numero maggiore di parametri e di dimensioni maggiori in quanto i parametri sono passati per riferimento alla memoria primaria.

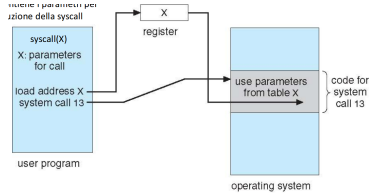

Figura 2.2: Passaggio dei parametri tramite tabella

Capitolo 3

Architettura di un Sistema Operativo

In un sistema operativo è molto importante separare le *policy* dai *meccanismi*. I meccanismi sono le funzionalità che il sistema operativo mette a disposizione, mentre le *policy* sono le regole che il sistema operativo segue per decidere come utilizzare i meccanismi.

Principi di progettazione Il principio di progettazione di un sistema operativo è quello di `KISS` (*Keep It Small and Simple*) usato per ottimizzare al meglio le *performance* implementando solo lo stretto necessario. Altro principio è il `POLP` (*Principle of Least Privilege*), ovvero dare il minimo dei privilegi necessari ad ogni componente per svolgere il proprio compito. Quest'ultimo principio è molto importante per garantire affidabilità e sicurezza.

3.1 Tipi di architetture

Sistemi monoblocco

Nei sistemi monoblocco non è presente una gerarchia tra i vari livelli del sistema operativo. Questo tipo di architettura è molto semplice e consiste in un unico strato *software* tra l'utente ed l'*hardware* del sistema. Le componenti sono dunque tutte allo stesso livello permettendo una comunicazione diretta tra l'utente e l'*hardware*. Questo tipo di architettura è molto semplice e veloce, ma il codice risulta interamente dipendente dall'architettura ed è distribuito su tutto il sistema operativo. Inoltre per testare ed eseguire il *debugging* di un singolo componente è necessario analizzare l'intero sistema operativo.

Sistemi a struttura semplice

Nei sistemi a struttura semplice è presente una piccola gerarchia, molto flessibile, tra i vari livelli del sistema operativo. Questo tipo di architettura mira ad una riduzione dei costi di sviluppo ed di manutenzione del sistema operativo. Non avendo una struttura ben definita, i componenti possono comunicare tra loro in modo diretto. Questo tipo di architettura è molto flessibile e permette di avere un sistema operativo molto piccolo e veloce come `MS-DOS` o `UNIX` originale.

`MS-DOS` Il sistema operativo `MS-DOS` è un sistema operativo a struttura semplice, molto piccolo e veloce. Questo sistema operativo è pensato per fornire il maggior numero di funzionalità in uno spazio ridotto. Infatti non sussistono suddivisioni in moduli, ed le interfacce e livelli non sono ben definiti. È infatti possibile accedere direttamente alle *routine* del sistema operativo ed non è prevista la *dual mode*.

`UNIX` **(Originale)** Struttura semplice limitata dalle poche funzionalità disponibili all'epoca in materia di *hardware*, con un *kernel* molto piccolo e veloce il quale scopo è risiedere tra l'interfaccia delle *system call* e l'*hardware*. Questo sistema operativo è stato progettato per essere molto flessibile e fornisce: *File System*, *Scheduling* della `CPU`, gestione della memoria e molto altro.

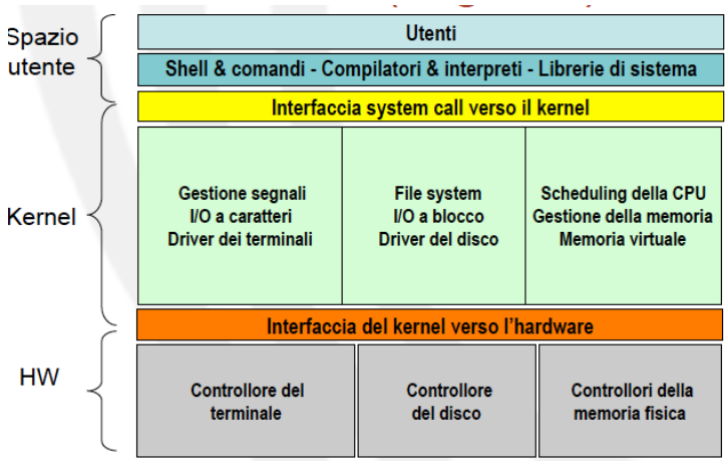

Figura 3.1: Struttura di `UNIX` originale

Sistema a livelli

Nei sistemi operativi organizzati a livelli gerarchici l'interfaccia utente risiede al livello più altro mentre l'*hardware* dal lato opposto. Ogni livello intermedio può solo usare funzioni fornite dal livello inferiore ed offrire funzionalità al livello superiore. Principale vantaggio di questa architettura è la modularità, infatti ogni livello può essere sviluppato e testato indipendentemente dagli altri. Questo tipo di architettura, d'altronde non è priva di svantaggi, infatti diventa difficile definire in modo approssimato gli strati, l'efficienza decresce in quanto ogni singolo strato aggiunge un costo di *overhead* ed le funzionalità dipendenti dal'*hardware* sono sparse su più livelli.

`THE` Il sistema operativo `THE` è un sistema d'uso accademico ed è il primo sistema operativo a struttura a livelli. Questo `SO` consiste in un insieme di processo che cooperano tra di loro usando la tecnica dei "semafori" per la sincronizzazione.

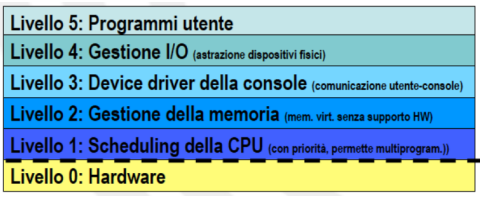

Figura 3.2: Struttura di `THE`

Sistemi basati su *Kernel*

I sistemi di questo genere hanno due soli livelli: i servizi *kernel* e quelli non-*kernel* (o *utente*). Il *file system* è un esempio di servizio non-*kernel*. Questo tipo di architettura è molto diffuso in quanto il ridotto e ben definito numero di livelli ne permette una facile implementazione e manutenzione, spesso però questo sistema può risultare troppo rigido e non adatto a tutti i tipi di applicazioni, oltre alla totale assenza di regole organizzative per le parti del `SO` al di fuori del *kernel*.

Micro-kenrel

Questo tipo di *kernel* è molto piccolo e fornisce solo i servizi essenziali per il funzionamento del sistema operativo. Tutte le altre funzionalità sono implementate come processi utente. Un esempio di ciò è `seL4` un *kernel open source* che implementa un *micro-kernel* e fornisce un'interfaccia per la gestione della memoria, dei processi e della comunicazione tra processi. `SeL4` è matematicamente verificato e privo di bug rispetto alle sue specifiche di forte sicurezza

Virtual Machine

L'architettura a VM è una estremizzazione dell'approccio a più livelli di IBM (1972), questo è pensato per offrire un sistema di *timesharing* "multiplo" dove il sistema operativo viene eseguito su una VM ed questa dà illusione di processi multipli, ma nella realtà ognuno di questi è in esecuzione sul proprio HW. In questo paradigma sono possibili più SO in una unica macchina.

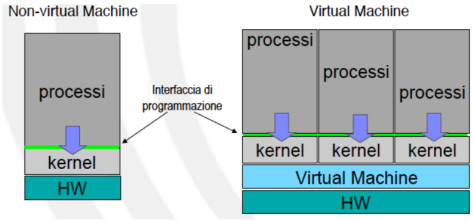

Figura 3.3: Differenze tra una macchina senza e con VM

Come è possibile notare ogni singolo processo è separato ed possiede il proprio *kernel*. Vengono quindi separata la multiprogrammazione ed la presentazione.

Tipo di *Hypervisor*

- **Tipo 1** (*Bare Metal*): Questo tipo di *Hypervisor* è installato direttamente sul'*hardware* e non necessita di un sistema operativo ospite. Questo tipo di *Hypervisor* è molto veloce e sicuro, ma è molto complesso da installare e configurare.
- **Tipo 2** (*Hosted*): Questo tipo di *Hypervisor* è installato sopra un sistema operativo ospite. Questo tipo di *Hypervisor* è molto più semplice da installare e configurare rispetto al tipo 1, ma è più lento e meno sicuro, inoltre è possibile avere problemi di compatibilità tra il sistema operativo ospite e il *Hypervisor*.

***Monolithic* vs *Micro-kernel* VM** Prima di fare una distinzione tra i due tipi di VM è necessario dire che entrambi rientrano nel tipo 1 di *Hypervisor* e dunque tutti i SO sono eseguiti direttamente sul'*hardware* virtualizzato.

- ***Monolithic***: Questo tipo di VM è molto simile ad un sistema operativo tradizionale, infatti ogni *VM* è un processo separato che esegue il proprio *kernel*. Questo tipo di VM è molto veloce, ma è molto complesso da implementare e mantenere.
- ***Micro-kernel***: Questo tipo di VM è molto simile ad un sistema operativo a *micro-kernel*, infatti il *kernel* della VM fornisce solo i servizi essenziali per il funzionamento del sistema operativo. Tutte le altre funzionalità sono implementate come processi utente. Questo tipo di VM è molto più semplice da implementare e mantenere rispetto al tipo 1, ma è più lento e meno sicuro.

Vantaggi - Svantaggi Principale vantaggio di questo tipo di architettura è la completa protezione del sistema, infatti ogni SO è separato e non può accedere alle risorse degli altri SO. Inoltre è possibile avere più SO in una sola macchina andando ad ottimizzare le risorse e ridurre i costi di sviluppo di un sistema operativo, oltre ad aumentare la portabilità delle applicazioni. Principale svantaggio riguardano le prestazioni del sistema, infatti ogni SO è eseguito su una VM e questo può portare ad un aumento dei tempi di esecuzione delle applicazioni. Inoltre è necessario avere gestire una *dual-mode* virtuale e non è possibile avere un sistema operativo in tempo reale, inoltre il fatto che una VM non possa accedere alle altre VM può portare ad un aumento dei costi di sviluppo e manutenzione del sistema.

Sistemi *client-server*

Poco diffusi ai giorni nostri, i sistemi *client-server* sono basati su un'architettura a due livelli: il *client* e il *server*. Questo sistema di basa sull'idea che il codice del sistema operativo vada portato sul livello superiore (il *client*) e il *server* rimanga molto piccolo e veloce andando solo a fornire i servizi essenziali per il funzionamento del sistema operativo ed la comunicazione tra il *client* e l'*hardware*. Questo tipo di architettura si presta bene per sistemi distribuiti.

3.2 Implementazione di un SO

I sistemi operativi sono tradizionalmente scritti in linguaggio *assembler* anche se è possibile scriverli in linguaggi di alto livello, come `C` o `C++`. La scrittura di un sistema operativo in linguaggio di alto livello permette di avere una implementazione molto rapida oltre ad aumentarne la compattezza e la mantienebilità. Inoltre è possibile avere una maggiore portabilità del sistema operativo, in quanto è possibile compilare il codice sorgente su più architetture. Tuttavia la scrittura di un sistema operativo in linguaggio di alto livello può portare ad un aumento dei tempi di esecuzione delle applicazioni e ad un aumento dei costi di sviluppo e manutenzione del sistema operativo.

Capitolo 4

Processi e *Thread*

In questo capitolo vedremo cosa sono i processi e i *thread* capendone le differenze e le somiglianze, vedremo come vengono gestiti e come vengono eseguiti. Infine vedremo come vengono gestiti i processi dal sistema operativo e come vengono eseguiti i processi dal sistema operativo.

4.1 Processi

Un processo è l'istanza di un programma in esecuzione, quando il programma viene eseguito e quindi caricato nella memoria primaria (`RAM`) diventa un processo. Mentre un programma è la parte statica di un software, il processo è la parte dinamica. Un processo viene eseguito in maniera sequenziale, ovvero un'istruzione alla volta, ma nei sistemi operativi moderni un processo può essere eseguito in maniera concorrente, ovvero più processi possono essere eseguiti in parallelo.

Immagine in memoria

Un processo quando viene caricato in memoria viene caricato in una zona di memoria chiamata *spazio degli indirizzi* (*address space*). Questo spazio è diviso in varie sezioni (da indirizzi alti ad indirizzi bassi):

- **Dati**: contiene le variabili globali e statiche del programma.
- ***Stack***: contiene le variabili locali e i parametri delle funzioni.
- eventuale memoria dinamica allocata durante l'esecuzione.
- ***Heap***: contiene la memoria dinamica allocata durante
- **Codice**: contiene il codice del programma.
- **Attributi del processo**: contiene informazioni sul processo.

4.1.1 Stato di un processo

Un processo durante la sua creazione ed esecuzione può trovarsi in diversi stati:

- **Nuovo**: il processo è stato creato ma non è ancora in esecuzione.
- **Pronto**: il processo è pronto per essere eseguito, ma non è ancora in esecuzione. (oppure è stato messo in attesa dalla `CPU`).
- **In esecuzione**: il processo è in esecuzione sulla `CPU`.
- **In attesa**: il processo è in attesa di un evento (es. `I/O`).
- **Terminato**: il processo è terminato.

Per la gestione di questi stati il sistema operativo usa un *dispatcher* il quale compito è quello di passare tra i processi e cambiare il loro stato. Per questo motivo il *dispatcher* è chiamato anche *scheduler*.

Scheduling

Lo *scheduling* è il processo di selezione del processo da eseguire sulla `CPU`. Esistono vari tipi di *scheduler*:

- ***Long time scheduler***: decide quali processi devono essere caricati in memoria. (Nella coda dei processi pronti).
- ***Short time scheduler***: decide quale processo deve essere eseguito sulla `CPU`. (Seleziona i processi dalla coda dei processi pronti).

Mentre lo *short-term* scheduler è un processo molto veloce in quanto viene chiamato molto spesso (ogni $10 - 100ms$), il *long-term* scheduler è un processo più lento in quanto viene chiamato molto raramente (ogni $1 - 10s$ o anche di più), questo però è responsabile del grado di multiprogrammazione del sistema.

Accantonamento L'accantonamento è il processo per il quale i processi pronti ad essere eseguiti vengono messi in una coda di attesa. Quando la `CPU` è pronta per eseguire un processo, il processo viene preso dalla coda e viene eseguito, nel caso nel quale il processo richieda un'operazione di `I/O` il processo viene messo in richiesta ed quando l'operazione di `I/O` (caratterizzata a sua volta da una coda per ogni dispositivo connesso) è completata il processo viene rimesso nella coda dei processi pronti.
Può anche succedere che il tempo per l'esecuzione di un processo sia scaduto, in questo caso il processo viene rimesso nella coda dei processi pronti. Se poi il processo generi dei processi figli, questi dopo la loro inizializzazione vengono messi nella coda dei processi pronti e vengono eseguiti, se il padre necessita che il processo figlio termini prima di lui, il padre viene messo in attesa che il figlio termini, altrimenti anche il padre viene messo nella coda dei processi pronti. Infine se un processo necessita di un segnale da parte di un altro processo, il processo viene messo in attesa finché non riceve il segnale (dal sistema o da un altro processo).

`I/O` vs `CPU` *bound* Un processo può essere `I/O` bound o `CPU` *bound*. Un processo `I/O` *bound* è un processo che richiede molte operazioni di `I/O` e poche operazioni sulla `CPU`, mentre un processo `CPU` *bound* è un processo che richiede molte operazioni sulla `CPU` e poche operazioni di `I/O`. Non è possibile stabilire a priori se un processo è `I/O` *bound* o `CPU` *bound*, ma è possibile stabilirlo solo durante l'esecuzione del processo analizzando quanta `CPU` usa e se richiede molte operazioni di `I/O`, sulla base di questo il processo viene classificato come `I/O` *bound* o `CPU` *bound*.

Operazione di *dispatch*

Quando si deve passare da un processo ad un altro si deve fare un'operazione di *dispatch*. Questa operazione consiste nel:

1. Cambiare il contesto (salvare lo stato del processo corrente (`PCB`) e caricare lo stato del processo successivo (`PCB`)).
2. Passare alla modalità utente (quando viene eseguito il *context switch* il sistema operativo è in modalità *kernel*, mentre il processo deve essere eseguito in modalità utente).
3. Salto alla prossima istruzione da eseguire del processo successivo.

Questa operazione è molto costosa in termini di tempo, in particolare l'operazione di *context switch* richiede risorse che rallentano il sistema senza eseguire nessuna operazione utile, la durata di ciò è strettamente dipendente dall'architettura del processore e dal sistema operativo.

4.1.2 Operazioni sui processi

Nella quasi totalità dei sistemi operativi moderni è possibile eseguire più processi in parallelo, per fare ciò il sistema operativo deve fornire delle operazioni per la gestione dei processi oltre ad un modo per l'identificazione dei processi. Di seguito vediamo quali sono le operazioni possibili sui processi.

Creazione di un processo

Un processo, come già detto, può creare altri processi, questi processi creati sono detti processi figli. Un processo padre può creare più processi figli, questi processi figli possono creare a loro volta altri processi figli e così via. Ai processi normalmente viene associato un *PID* (*Process IDentifier*) che è un numero

univoco che identifica il processo all'interno del sistema operativo.
Il processo figlio può ottenere le risorse necessarie per la sua esecuzione in due modi:

- Ereditando le risorse del processo padre (*sharing*)
- Ottenendo nuove risorse dal sistema operativo (*partitioning*)

Inoltre il processo figlio può essere eseguito in parallelo in maniera sincrona rispetto al processo padre (il processo padre aspetta che il processo figlio termini) o asincrona (il processo figlio viene eseguito in parallelo al processo padre).

Nei sistemi `UNIX` Nei sistemi `UNIX` esistono diverse *system call* per la creazione di processi, la principale è `fork()` che crea un processo figlio identico al processo padre, la differenza tra i due processi è il *PID* e il *PPID* (*Parent Process IDentifier*). Il processo figlio eredita tutte le risorse del processo padre, inoltre il processo figlio può modificare le risorse ereditate dal processo padre. Altra chiamata di sistema è `exec()` che permette di caricare un nuovo programma in un processo figlio, in questo caso il programma tra il processo padre e il processo figlio è differente. Infine la chiamata di sistema `wait()` permette l'esecuzione sincrona di un processo figlio rispetto al processo padre.

Terminazione di un processo

Un processo può terminare in tre modi:

- **Normalmente**: il processo termina la sua esecuzione invocando la *system call* `exit()` (con eventualmente un codice di uscita).
- **Forzatamente dal processo padre**: il processo padre può terminare il processo figlio invocando la *system call* `kill()`, oppure nel caso di un eccessivo uso di risorse, oppure a sua volta il processo padre termina anormalmente.
- **Forzatamente dal sistema operativo**: il sistema operativo può terminare un processo nel caso di un errore di esecuzione, oppure nel caso nel quale l'utente chiuda l'applicazione.

Nota come nel primo caso non sia esclusa la possibilità che il processo termini in maniera anomala, ad esempio per un errore di esecuzione gestito dal processo stesso, infatti quando il codice di uscita è diverso da `0` si intende che il processo è terminato in maniera anomala, ogni codice diverso da `0` ha un significato diverso.
Quando un processo termina il sistema operativo si occupa di liberare le risorse utilizzate dal processo come la memoria allocata, i file aperti, le connessioni di rete, o altre risorse.

4.1.3 Gestione dei processi del `SO`

Di fatto il sistema operativo non è altro che un programma a tutti gli effetti, e dunque la sua esecuzione è un processo come un altro. Questo non significa però che il sistema operativo non essere gestito separatamente dagli altri processi, infetti esistono diverse opzioni l'esecuzione del *kernel*:

- Il *kernel* viene eseguito completamente in maniera separata dagli altri processi.
- Il *kernel* viene eseguito all'interno di un processo utente.
- Il *kernel* viene eseguito come un processo separato.

***Kernel* separato** In questo caso il *kernel* è eseguito al di fuori degli altri processi, questo gli permette di avere uno spazio in memoria ben definito e riservato oltre ad avere il totale controllo del sistema ed a essere eseguito in modalità *kernel* (ovvero con privilegi elevati). I processi sono dunque solo propri all'utente ed un processo non potrà mai essere eseguito in modalità *kernel*.

***Kernel* nel processo utente** In questo caso il *kernel* è eseguito all'interno di un processo utente, questo permette ai programmi utente di chiamare qualunque servizio del sistema operativo, ma tramite una modalità protetta (*kernel mode*) che permette al sistema operativo di controllare le chiamate e di evitare che un processo utente possa fare danni al sistema. Dato che il *kernel* è un processo a tutti gli effetti la sua immagine in memoria sarà composta dal "*kernel stack*" per la gestione delle chiamate di

sistema e dal "*kernel code*" che consiste nei dati e codice del SO condiviso tra tutti i processi.
Questo approccio porta ad una riduzione del tempo di *context switch* in quanto è necessario solo la *mode switch* e non l'intero *context switch* lasciando però intatte le possibilità di riattivazione del processo utente o di eseguire un altro processo eseguendo un *context switch* completo.

***Kernel* come processo separato** In questo caso ogni servizio del sistema operativo è eseguito come un processo separato in modalità protetta. L'unica parte del *kernel* che deve essere eseguita separatamente è lo *scheduler* in quanto deve essere eseguito in modalità *kernel*. Questo approccio è molto vantaggioso per sistemi multiprocessore in quanto permette di eseguire i servizi del sistema operativo in parallelo ed in un processore designato.

4.2 *Thread*

Un *thread* è l'unità di base d'uso della CPU, un processo può contenere uno o più *thread* che condividono lo stesso codice, dati e file aperti, ma ognuno ha un suo *stack*, lo stato del *program counter* e dei registri ed un numero identificativo.
Dunque le risorse e lo spazio di indirizzamento sono propri del processo, mentre lo stato della CPU è proprio del *thread* assieme al *program counter* e ai registri.
Classicamente un processo è composto da un solo *thread*, la capacità di avere più *thread* in un processo è chiamata *multithreading*. Questo permette di avere un processo con più *thread* separando il flusso di esecuzione e lo spazio di indirizzamento, ma condividendo le risorse del processo.

Vantaggi I vantaggi del *multithreading* sono:

- **Risposta più veloce**: Se sono necessari molti calcoli o operazioni di I/O è possibile eseguire queste operazioni in parallelo.
- **Condivisione delle risorse**: I *thread* possono condividere le risorse del processo, mentre processi separati devono usare meccanismi di comunicazione.
- **Economia**: Creare un *thread* è più veloce e meno costoso di creare un processo.
- **Scalabilità**: I *thread* possono essere eseguiti in parallelo su più processori o su più core.

4.2.1 Implementazione dei *thread*

Vediamo ora come sono implementati i *thread* nei sistemi operativi.

Stato dei *thread*

Un *thread*, come un processo, può trovarsi in diversi stati:

- **Pronto**: il *thread* è pronto per essere eseguito.
- **In esecuzione**: il *thread* è in esecuzione sulla CPU.
- **In attesa**: il *thread* è in attesa di un evento.

Un *thread* può essere in uno di questi stati, ma il processo può non essere nello stesso stato di un *thread* in quanto un processo può contenere più *thread* e quindi un processo può essere in uno stato diverso da quello dei suoi *thread*.
Un classico problema degli stati dei *thread* è la questione di cosa fare quando un *thread* è in attesa di un evento, questa "attesa" deve bloccare l'intero processo o solo il *thread* in attesa? Ciò dipende dall'implementazione dei *thread* nel sistema operativo.

Implementazione dei *thread*

Esistono due principali implementazioni dei *thread*:

- ***User-level threads***: I *thread* sono implementati a livello utente, il sistema operativo non è a conoscenza dei *thread* e non li gestisce. Le funzionalità sono implementate in una libreria che gestisce i *thread* e le chiamate di sistema.

- ***Kernel-level threads***: I *thread* sono implementati a livello del *kernel*, il sistema operativo è a conoscenza dei *thread* e li gestisce.

- ***Hybrid threads***: I *thread* sono implementati a livello del *kernel*, ma il sistema operativo permette di creare *thread* a livello utente. (es. `SOLARIS`)

User-level threads Se si opta per l'implementazione dei *thread* a livello utente, il sistema operativo non è a conoscenza dei *thread* e non li gestisce e dunque non è necessario passare in modalità *kernel* per la gestione dei *thread* risparmiando due *context switch*. Ogni applicazione deve però implementate lo *scheduler* dei *thread* e la gestione degli stati dei *thread*. Quanto detto garantisce una maggiore portabilità delle applicazioni senza dover riscrivere il codice per ogni sistema operativo, ma allo stesso tempo non permette di sfruttare appieno le potenzialità del sistema operativo. Se però un *thread* necessita di un'operazione di `I/O` o di un'operazione che richiede l'intervento del sistema operativo, tutti i *thread* del processo vengono bloccati in quanto il sistema operativo non è a conoscenza dei *thread* e non può gestire i *thread* in maniera indipendente. (es. *Green threads (`JDK1.1`)*, `GNU` *Portable Threads*, `POSIX` *Pthreads*)

Kernel-level threads Se si opta per l'implementazione dei *thread* a livello del *kernel*, il sistema operativo è a conoscenza dei *thread* e li gestisce, dunque il sistema operativo può gestire i *thread* in maniera indipendente e può sfruttare appieno le potenzialità del sistema operativo. Ogni *thread* è un processo a tutti gli effetti, dunque ogni *thread* ha il proprio *PCB* e il proprio spazio di indirizzamento. Questo permette di sfruttare appieno le potenzialità del sistema operativo, ma allo stesso tempo richiede due *context switch* per passare da un *thread* all'altro. (es. `Windows`, `Linux`, *Native Threads (`JDK1.2`)*)

Hybrid threads Se si opta per l'implementazione dei *thread* ibridi, il sistema operativo permette di creare *thread* a livello utente, ma i *thread* sono implementati a livello del *kernel*. Questo permette di sfruttare appieno le potenzialità del sistema operativo, ma allo stesso tempo permette di creare *thread* a livello utente. (es. `SOLARIS`)

4.2.2 Esempio di libreria - `pthreads`

Nel caso di implementazione dei *thread* a livello utente, il sistema operativo non è a conoscenza dei *thread* e dunque non li gestisce, ma è necessario utilizzare una libreria che gestisca i *thread*. Un esempio di libreria per la gestione dei *thread* è `pthreads` (`POSIX` *Threads*).
`pthreads` è una libreria standard per la gestione dei *thread* in sistemi `UNIX` e sistemi `UNIX-like`. La libreria fornisce un'interfaccia standard per la creazione, la sincronizzazione e la terminazione dei *thread* nel linguaggio `C`. La libreria fornisce la possibilità di caratterizzare i *thread* sulla base della priorità (influenza lo *scheduling*) e della dimensione dello *stack* (stabilisce quante risorse può utilizzare il *thread*). Gli attributi di un *thread* sono contenuti nell'oggetto di tipo `pthread_attr_t` e tramite la funzione `pthread_attr_init()` si inizializzano gli attributi del *thread*. Una volta inizializzati gli attributi tramite la funzione `pthread_create()` si crea il *thread* passando come argomenti:

1. Una variabile del tipo `pthread_t` che conterrà l'identificativo del *thread*.

2. Un oggetto del tipo `pthread_attr_t` che conterrà gli attributi del *thread*.

3. Un puntatore alla funzione che il *thread* dovrà eseguire.

4. Un puntatore agli argomenti della funzione.

Una volta creato il *thread* questo terminerà quando il codice della funzione terminerà, oppure quando nel codice della funzione verrà invocata la funzione `pthread_exit()` con parametro `value_ptr` che conterrà il valore di uscita del *thread*. Se invece il *thread* deve essere sospeso in attesa di un altro *thread* si può utilizzare la funzione `pthread_join()` con parametri:

1. Un oggetto del tipo `pthread_t` che identifica il *thread* da attendere.

2. Un puntatore alla variabile che conterrà il valore di uscita del *thread* atteso.

```
#include <pthread.h>
#include <stdio.h>
void *tbody(void *arg)
    {
    int j;
    printf("ciao sono un thread, mi hanno appena creato\n");
    *(int *)arg = 10;
    sleep(2) /* faccio aspettare un po il mio creatore poi termino */
    pthread_exit((int *)50); /* oppure return ((int *)50); */
}
main(int argc, char **argv)
{
    int i;
    pthread_t mythread;
    void *result;
    printf("sono il primo thread, ora ne creo un altro \n");
    pthread_create(&mythread, NULL, tbody, (void *) &i);
    printf("ora aspetto la terminazione del thread che ho creato \n");
    pthread_join(mythread, &result);
    printf("Il thread creato ha assegnato %d ad i\n",i);
    printf("Il thread ha restituito %d \n",result);
}
```

In questo esempio la variabile "mythread" assume dei valori corrispondenti all'identificativo del *thread* creato, mentre la variabile "result" assume il valore di uscita del *thread* creato. La funzione "tbody" è la funzione che il *thread* dovrà eseguire, mentre la variabile "i" è un argomento passato alla funzione. La funzione "pthread_exit()" termina il *thread* e restituisce il valore passato come argomento, mentre la funzione "pthread_join()" sospende il *thread* corrente in attesa del *thread* passato come argomento e restituisce il valore di uscita del *thread* atteso.

Condivisione dello spazio logico

Come già anticipato i *thread* condividono lo stesso spazio logico, questo significa che i *thread* possono accedere alle stesse variabili globali e statiche e se un *thread* modifica una variabile globale, la modifica sarà visibile a tutti gli altri *thread*. Questo può portare a problemi di sincronizzazione tra i *thread* e dunque è necessario utilizzare meccanismi di sincronizzazione per evitare problemi di accesso concorrente alle variabili globali. Possono esistere variabili locali ai *thread* che sono visibili solo al *thread* che le ha dichiarate, ma non sono visibili agli altri *thread*, ciò usando la classe `thread_specific_data`.

Per la sincronizzazione Per la sincronizzazione tra i *thread* si possono utilizzare o gli strumenti direttamente forniti dalla libreria `pthreads` (come i semafori) oppure si possono utilizzare le primitive di sincronizzazione fornite dal sistema operativo (come `sleep(n)` che sospende il *thread* corrente per n secondi). Per tenere traccia del tempo trascorso nella funzione possono essere usati due metodi:

- Un *interrupt Request* (`IRQ`) che viene generato ad intervalli regolari e che incrementa un contatore. Il `SO` controlla se ci sono delle `sleep` scadute e se ci sono le risveglia.
- Riconfigurazione delle `IRQ` in modo che avvenga una `IRQ` quando la prima `sleep` scade, e una seconda `IRQ` quando la seconda `sleep` scade e così via. Ciò comporta a migliore precisione ma alto *overhead* per la riconfigurazione delle `IRQ` ad ogni `sleep`.