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`

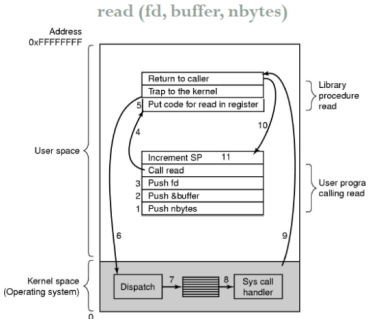


Figura 2.1: Passaggio dei parametri tramite `stack`

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.

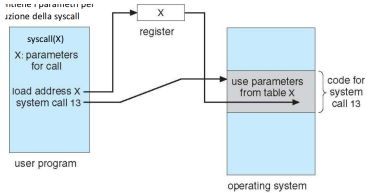


Figura 2.2: Passaggio dei parametri tramite tabella

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.

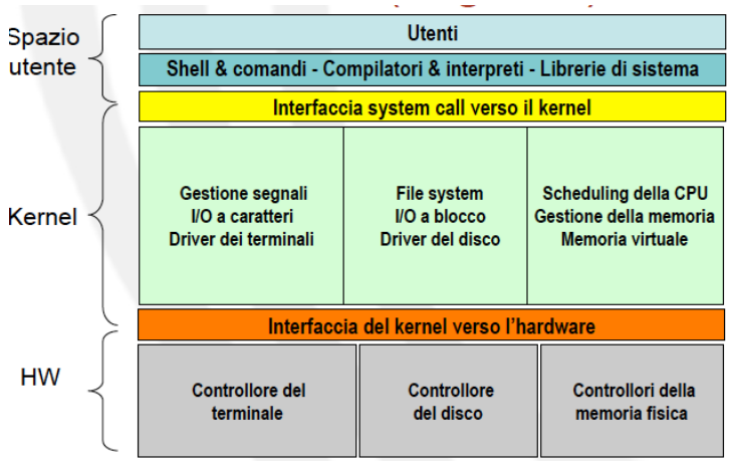


Figura 3.1: Struttura di `UNIX` originale

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.

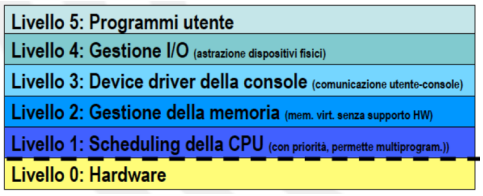


Figura 3.2: Struttura di `THE`

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.

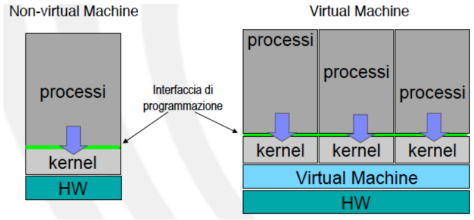


Figura 3.3: Differenze tra una macchina senza e con VM

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`.

Capitolo 5

Comunicazione tra processi

Normalmente i processi si dividono in processi indipendenti, ovvero quei processi la cui esecuzione è indipendente da quella degli altri processi ed non condivide i dati, e processi cooperanti, ovvero quei processi che condividono i dati e devono comunicare tra loro, la loro esecuzione non è deterministica e non è riproducibile.

In generale Esistono diversi motivi per cui i processi devono comunicare tra loro, tra cui:

- **Scambio di informazioni**: i processi devono scambiarsi informazioni per cooperare tra loro.
- **Accelerazione del calcolo**: i processi possono cooperare per eseguire un calcolo più velocemente.
- **Modularità**: i processi possono essere scritti in modo indipendente e comunicare tra loro per cooperare.
- **Convenienza**: è più semplice scrivere processi separati che cooperano tra loro piuttosto che scrivere un unico processo.

per ottenere una comunicazione tra processi è necessario che i processi condividano un canale di comunicazione, esistono due tipi di canali di comunicazione:

scambio di messaggi i processi comunicano scambiandosi messaggi che vengono inviati attraverso un canale di comunicazione tra il *kernel* e i processi, i messaggi possono essere inviati in modo sincrono o asincrono.

memoria condivisa i processi comunicano condividendo una regione di memoria, i processi possono leggere e scrivere nella memoria condivisa, la memoria condivisa è un canale di comunicazione molto più veloce rispetto allo scambio di messaggi, ma è più difficile da gestire.

Il primo risulta più sicuro in quanto i processi non possono accedere direttamente alla memoria degli altri processi ed il messaggio viene verificato dal *kernel* prima di essere inviato, mentre il secondo è più veloce in quanto non richiede l'intervento del *kernel* per la comunicazione.
Tutti i meccanismi di comunicazione tra processi sono implementati dal *kernel* del sistema operativo racchiusi nei protocolli di comunicazione tra processi (`IPC` - *Inter-Process Communication*).

5.1 `IPC` - *Message Passing*

Il protocollo `ICP` racchiude un insieme di meccanismi che permettono la comunicazione tra processi, tra i quali vi è il *message passing*, ovvero un meccanismo che permette ai processi di comunicare scambiandosi messaggi e senza condividere delle variabili e/o memoria. Le operazioni di base che ogni `SO` deve fornire per il *message passing* sono:

- `send` - invia un messaggio ad un processo. (con lunghezza fissa o variabile)
- `receive` - riceve un messaggio da un processo.

Prima ancora che i processi possano comunicare tra loro è necessario che essi siano in grado di identificarsi e stabilire un canale di comunicazione, per fare ciò è necessario che i processi abbiano un identificativo univoco, ovvero un *PID* (*Process IDentifier*).
L'implementazione di questo canale di comunicazione può essere realizzata in due modi:

livello fisico i messaggi vengono inviati attraverso un canale di comunicazione fisico, come ad esempio una rete o un bus.

livello logico i messaggi vengono inviati attraverso un canale di comunicazione logico, come ad esempio una coda di messaggi.

Le scelte di uno o dell'altro canale di comunicazione dipendono dalle esigenze del sistema e dalle prestazioni richieste. Fattori che influenzano la scelta sono:

- Come vengono stabiliti i canali
- Se un canale può essere utilizzato da più processi contemporaneamente
- Quanti canali possono essere aperti contemporaneamente tra una stessa coppia di processi
- La lunghezza massima del canale
- La lunghezza (fissa/variabile) massima dei messaggi
- Se il canale è *simplex*, *half-duplex* o *full-duplex*

5.1.1 Nominazione

A livello di nominazione, ovvero come i processi si identificano, esiste la comunicazione diretta e la comunicazione indiretta:

Comunicazione Diretta

Nella comunicazione diretta i processi si identificano direttamente, ovvero il mittente conosce l'identificativo del destinatario e viceversa, in questo modo il mittente può inviare il messaggio direttamente al destinatario. Questo metodo è molto veloce, ma presenta dei problemi:

- Il mittente deve conoscere l'identificativo del destinatario
- Il destinatario deve essere in esecuzione
- Nel caso in cui il destinatario o il ricevente cambi identificativo, il mittente deve essere aggiornato

La comunicazione diretta può a sua volta essere simmetrica o asimmetrica:

Simmetrica Il mittente e il destinatario si conoscono a priori e possono comunicare tra loro. Sia per l'invio che per la ricezione dei messaggi è necessario conoscere l'identificativo del processo con cui si vuole comunicare.

Asimmetrica Il mittente e il destinatario non si conoscono a priori. Solo il mittente conosce l'identificativo del destinatario, il destinatario non conosce l'identificativo del mittente ed ascolta qualsiasi messaggio che arriva.

Comunicazione Indiretta

Nella comunicazione indiretta i messaggi vengono inviati ad un canale di comunicazione comune detto *mailbox* (o porte), ognuna di queste *mailbox* ha associato un numero identificativo univoco, i processi possono inviare e ricevere messaggi da queste *mailbox* senza dover conoscere l'identificativo del destinatario, ma devono condividere una *mailbox* comune.

Flusso di una *mailbox* Prima di poter inviare un messaggio ad una *mailbox* è necessario che questa venga creata, una volta creata la *mailbox* il processo può inviare un messaggio ad essa, il messaggio viene inserito in una coda di messaggi associata alla *mailbox*, il destinatario può ricevere il messaggio dalla *mailbox* e leggerlo, una volta letto il messaggio viene rimosso dalla coda. Quando la *mailbox* non è più necessaria può essere eliminata.

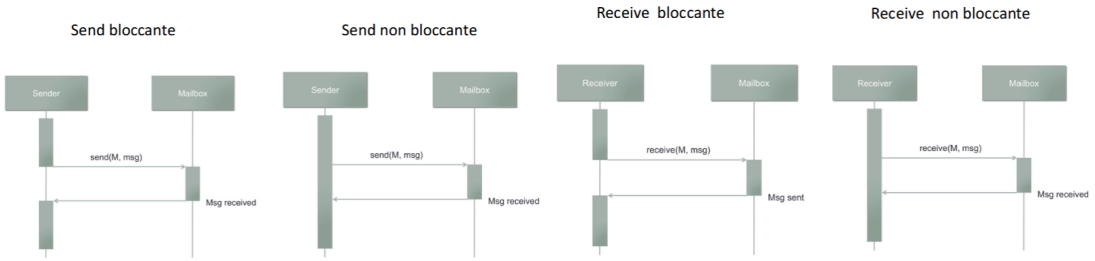
Invio e ricezione Per inviare e ricevere messaggi da una *mailbox* è necessario conoscere l'identificativo della *mailbox*, una volta conosciuto l'identificativo il processo può inviare e ricevere messaggi dalla *mailbox*.

Proprietà del canale Come già detto, il canale di comunicazione viene stabilito solo se i processi condividono una *mailbox*, ma una *mailbox* può essere associata a molti processi ed una stessa coppia di processi può avere più *mailbox* associate, inoltre una *mailbox* può essere o meno bi-direzionale.

Problema riceventi multipli Un problema che si può presentare è quello dei riceventi multipli, ovvero quando un mittente invia un messaggio ad una *mailbox* e ci sono più processi che ricevono i messaggi da quella *mailbox*, in questo caso il *SO* deve permettere solo ad uno dei processi di ricevere il messaggio, e questo viene fatto in maniera arbitraria.

5.1.2 Sincronizzazione

Uno scambio di messaggi può essere "bloccante" (sincrono) o "non bloccante" (asincrono), ovvero il mittente può continuare ad eseguire il proprio codice dopo aver inviato il messaggio o deve attendere che il destinatario riceva il messaggio.
Se il canale di comunicazione è bloccante, il mittente deve attendere che il destinatario riceva il messaggio ed assicurarsi che il messaggio sia stato ricevuto, se il canale di comunicazione è non bloccante il mittente può continuare ad eseguire il proprio codice dopo aver inviato il messaggio, senza dover attendere che il destinatario riceva il messaggio, in questo caso il mittente non può sapere se il messaggio è stato ricevuto o meno.



(a) Invio bloccante/non bloccante (b) Ricezione bloccante/non bloccante

Figura 5.1: Invio e ricezione bloccante/non bloccante

5.2 IPC - Memoria Condivisa

Un altro meccanismo di comunicazione tra processi è la memoria condivisa, ovvero un'area di memoria condivisa tra più processi, i processi possono leggere e scrivere nella memoria condivisa, la memoria condivisa è un canale di comunicazione molto più veloce rispetto allo scambio di messaggi, ma è più difficile da gestire, inoltre il *kernel* non può controllare l'accesso alla memoria condivisa, quindi è necessario che i processi si sincronizzino tra loro per evitare problemi di accesso concorrente.

Flusso di POSIX

Prendiamo come esempio la memoria condivisa in POSIX, per poter utilizzare la memoria condivisa è necessario che uno dei processi crei la memoria condivisa, una volta creata la memoria condivisa l'altro processo deve "attaccarsi" al segmento di memoria condivisa, una volta "attaccato" il processo può avere il permesso di leggere e scrivere oppure solo di leggere, una volta terminato il processo deve "staccarsi" dalla memoria condivisa. Il processo che ha creato la memoria condivisa deve rimuoverla una volta terminato.

Le *pipe*

Un altro meccanismo di comunicazione tra processi è la *pipe*, ovvero un canale di comunicazione tra processi.

La *pipe* è un canale di comunicazione che permette di inviare e ricevere messaggi tra processi, la *pipe* è un canale di comunicazione unidirezionale, ovvero i messaggi possono essere inviati in una sola direzione, ma è possibile creare due *pipe* per permettere la comunicazione in entrambe le direzioni. Distinguiamo tra *pipe* ordinarie e *pipe* con nome:

***Pipe* ordinarie** Le *pipe* ordinarie permettono la comunicazione in uno stile "Produttore-Consumatore" dove il produttore scrive nella *pipe* e il consumatore legge dalla *pipe*, questa tipologia richiede una relazione tra processi, queste infatti possono essere aperte solo da processi padri verso i processi figli.

***Pipe* con nome** Le *pipe* con nome sono simili alle *pipe* ordinarie, ma permettono la comunicazione bidirezionale, non è richiesta la relazione tra processi e più processi possono usare la stessa *pipe* per comunicare tra loro, ma è necessario che i processi si sincronizzino tra loro per evitare problemi di accesso concorrente. Le *pipe* con nome sono disponibili sia in sistemi `UNIX` che in sistemi `Windows`, ma in quest'ultimo caso non sono implementate come file di tipo *FIFO* ma come file temporanei.

Capitolo 6

Scheduling della CPU

Andremo an analizzare in questo capitolo lo *Scheduling* della CPU, ovvero il modo in cui il sistema operativo decide quale processo eseguire in un dato momento. Lo *Scheduling* è una parte fondamentale del sistema operativo, poiché influisce direttamente sulle prestazioni e sull'efficienza del sistema. Distingueremo inoltre i vari tipi di *Scheduling* (breve, medio e lungo termine) e i vari algoritmi di *Scheduling* (FIFO, SJF, Round Robin, ecc.).

6.1 Concetto di *Scheduling*

Lo *Scheduling* è il processo di assegnazione di attività nel tempo, l'uso della multiprogrammazione permette di eseguire più processi in parallelo, ma il sistema operativo deve decidere quale processo eseguire in un dato momento, visto che la CPU può eseguire solo un processo alla volta. Bisogna quindi decretare se un programma può essere ammesso nella memoria e quale processo deve essere eseguito in un dato momento.
Come visto nella sezione 4.1.1, un processo può trovarsi in uno dei seguenti stati:

- **New**: il processo è stato creato, ma non è ancora pronto per essere eseguito.
- **Ready**: il processo è pronto per essere eseguito, ma non ha ancora ottenuto l'accesso alla CPU.
- **Running**: il processo sta attualmente eseguendo sulla CPU.
- **Waiting**: il processo è in attesa di un evento esterno (ad esempio, l'input dell'utente o la disponibilità di una risorsa).
- **Terminated**: il processo ha completato la sua esecuzione e sta per essere rimosso dalla memoria.

Inoltre esistono diverse code di Ready e di Waiting, a seconda del tipo di processo. Ad esempio, i processi in attesa di I/O potrebbero essere in una coda separata rispetto ai processi in attesa di un semaforo.

Implementazione delle Code

A livello pratico le code di Ready e di Waiting sono implementate come liste collegate (*linked list*) o come array.
Ogni coda ha un *queue header* che contiene informazioni sulla coda stessa, come il puntatore al primo elemento della coda ed il puntatore all'ultimo elemento della coda. Ogni processo ha un *process control block* (PCB) che contiene informazioni sul processo stesso, come il suo stato, il contenuto dei registri quando il processo era in esecuzione, il puntatore al prossimo processo da eseguire, ecc...
Una coda può essere o la coda di *ready*, dove i processi pronti per essere eseguiti sono in attesa di essere assegnati alla CPU, oppure una delle code di *waiting*, dove i processi sono in attesa di un evento esterno ed ogni coda rappresenta un evento diverso.

6.2 Tipi di *Scheduling*

Per la gestione dei processi si possono distinguere tre tipi di *scheduling*, di cui due principali ed uno secondario:

- ***Long-term scheduling*** - Pianificazione a lungo termine
- ***Short-term scheduling*** - Pianificazione a breve termine
- ***Medium-term scheduling*** - Pianificazione a medio termine

Pianificazione a lungo termine - (*job-scheduler*)

La pianificazione a lungo termine è il processo di selezione dei processi da ammettere nella memoria principale. Questo tipo di pianificazione è responsabile della creazione di nuovi processi e della loro ammissione nella memoria, e dunque nella coda *ready*, per essere eseguiti. La pianificazione a lungo termine determina il grado di multiprogrammazione del sistema, ovvero il numero di processi che possono essere eseguiti contemporaneamente. Se il grado di multiprogrammazione è troppo alto, il sistema potrebbe diventare instabile e i processi potrebbero non ricevere le risorse necessarie per essere eseguiti. Se il grado di multiprogrammazione è troppo basso, la CPU potrebbe rimanere inattiva per lunghi periodi di tempo, riducendo l'efficienza del sistema. Inoltre il *long-term scheduler* è responsabile della determinazione del tipo di processo da eseguire, ovvero se questo è CPU *bound* oppure I/O *bound*.

Frequenza La pianificazione a lungo termine viene eseguita con frequenza dell'ordine del secondo o di pochi secondi, poiché la creazione di nuovi processi e la loro ammissione nella memoria sono operazioni relativamente costose in termini di tempo e risorse.
Questo sistema è opzionale e può essere assente.

Pianificazione a breve termine - *CPU-scheduler*

La pianificazione a breve termine è il processo di selezione del processo da eseguire sulla CPU in un dato momento. Questo tipo di pianificazione è responsabile della gestione dei processi in esecuzione e della loro assegnazione alla CPU. La pianificazione a breve termine determina quale processo deve essere eseguito in un dato momento, in base a diversi criteri, come la priorità del processo, il tempo di attesa e il tempo di completamento. La pianificazione a breve termine è responsabile della gestione della CPU e della sua assegnazione ai processi in esecuzione.

Frequenza La pianificazione a breve termine viene eseguita con frequenza dell'ordine dei millisecondi, dunque deve essere una operazione molto veloce, infatti se il tempo di processo è $100ms$ ed il tempo di *scheduling* è $10ms$, il tempo di *scheduling* incide per il 9% sul tempo totale di esecuzione del processo. Se il tempo di *scheduling* è troppo alto, il sistema potrebbe diventare instabile e i processi potrebbero non ricevere le risorse necessarie per essere eseguiti. Se il tempo di *scheduling* è troppo basso, la CPU potrebbe rimanere inattiva per lunghi periodi di tempo, riducendo l'efficienza del sistema.
Questo sistema è sempre presente e non può essere assente, poiché è necessario per la gestione della CPU e dei processi in esecuzione.

Pianificazione a medio termine - *medium-term scheduler*

La pianificazione a medio termine è un processo intermedio tra la pianificazione a lungo termine e la pianificazione a breve termine. Questo è presente se e solo se il sistema operativo supporta la *swapping*, ovvero il trasferimento di processi dalla memoria principale alla memoria secondaria (disco) e viceversa. La porzione di disco usata per questo scopo è chiamata *swap space* ed sostanzialmente è della RAM virtuale che viene usata per memorizzare i processi che non sono attualmente in esecuzione. La pianificazione a medio termine è responsabile della gestione della memoria e della sua assegnazione ai processi in esecuzione. Questo tipo di pianificazione è responsabile dell'immagazzinamento dei processi nella memoria secondaria e del loro trasferimento nella memoria principale quando necessario. Questo processo viene eseguito con tutti i processi che escono dalla CPU per rientrare nella *ready queue* ma non può avvenire se il processo esce dalla CPU per inserirsi nella *waiting queue*.

6.3 Scheduling della CPU

Scheduler Lo *scheduler* della CPU è, a livello logico, il modulo del SO che decide quale processo eseguire in un dato momento, vista la frequenza di chiamate a funzioni di *Scheduling* e la velocità con cui i processi passano da uno stato all'altro, lo *scheduler* deve essere molto veloce.

Dispatcher Il *dispatcher* è il modulo del SO che effettivamente esegue il passaggio di controllo tra i processi, ovvero il passaggio da un processo all'altro. Il *dispatcher* è responsabile di:

- *Switch* del contesto: salva il contesto del processo corrente e carica il contesto del processo successivo.
- Passaggio alla modalità utente: il SO deve passare dalla modalità kernel alla modalità utente per eseguire il processo.
- Salto alla opportuna locazione nel codice: il `dispatcher` deve saltare alla locazione corretta nel codice del processo.

La latenza di un *dispatcher* consiste nel tempo necessario per eseguire queste operazioni, ovvero fermare il processo corrente e passare al successivo. La latenza del *dispatcher* è molto importante, poiché influisce sulle prestazioni del sistema. Un *dispatcher* veloce può migliorare le prestazioni del sistema, mentre un *dispatcher* lento può causare un degrado delle prestazioni.

Modello astratto del sistama

Quando parliamo di un processo a livello astratto consideriamo che questo possa essere o in CPU *burst* oppure in I/O *burst*

Distribuzione dei CPU *burst* Solitamente i processi hanno una distribuzione dei CPU *burst* che segue una distribuzione esponenziale, ovvero la maggior parte dei processi ha un CPU *burst* breve, mentre pochi processi hanno un CPU *burst* lungo. Questo è dovuto al fatto che i processi brevi sono più comuni rispetto ai processi lunghi. Per questo motivo è stato implementato il processo di prelazione (*preemption*)

Prelazione Come detto in precedenza, i processi brevi sono più comuni rispetto ai processi lunghi, per questo motivo è stato implementato il processo di prelazione (*preemption*), ovvero la possibilità di interrompere un processo in esecuzione per dare la precedenza ad un altro processo. Esistono dunque in circolazione due tipi di *scheduler*:

- **Non preemptive**: il processo in esecuzione non può essere interrotto, ma deve terminare la sua esecuzione prima di passare al successivo.
- **Preemptive**: il processo in esecuzione può essere interrotto in qualsiasi momento per dare la precedenza ad un altro processo.

La prelazione è utile per garantire che i processi brevi vengano eseguiti il prima possibile, evitando che i processi lunghi occupino la CPU per troppo tempo. Tuttavia, la prelazione può anche causare un aumento della latenza del *dispatcher*, poiché il *dispatcher* deve eseguire il passaggio di controllo tra i processi più frequentemente.

Metriche di *scheduling*

Esistono diverse metriche sulle quali scegliere un algoritmo di *scheduling* piuttosto che un altro, le più comuni sono:

- **Utilizzo della CPU** (*CPU Utilization*): percentuale di tempo in cui la CPU è occupata ad eseguire processi. Un utilizzo della CPU del 100% è l'ideale, ma è difficile da raggiungere.
- ***Throughput***: numero di processi completati in un dato intervallo di tempo. Un *throughput* elevato è desiderabile, poiché indica che il sistema sta eseguendo molti processi in un breve periodo di tempo.
- **Tempo di attesa** (*Waiting Time*): tempo medio che un processo trascorre in attesa di essere eseguito. Un tempo di attesa basso è desiderabile, poiché indica che i processi vengono eseguiti rapidamente.
- **Tempo di completamento** (*Turnaround Time*): tempo medio che un processo trascorre nel sistema, dalla sua creazione alla sua terminazione. Un tempo di completamento basso è desiderabile, poiché indica che i processi vengono eseguiti rapidamente.

- **Tempo di risposta** (*Response Time*): tempo medio che intercorre tra l'invio di una richiesta e la ricezione della risposta. Un tempo di risposta basso è desiderabile, poiché indica che il sistema risponde rapidamente alle richieste degli utenti.

Il compito di un algoritmo di *scheduling* è quello di massimizzare l'utilizzo della CPU e il *throughput*, minimizzando il tempo di attesa, il tempo di completamento e il tempo di risposta. Tuttavia, non è sempre possibile ottimizzare tutte queste metriche contemporaneamente, poiché spesso ci sono compromessi tra di esse. Ad esempio, un algoritmo che massimizza l'utilizzo della CPU potrebbe aumentare il tempo di attesa dei processi, mentre un algoritmo che minimizza il tempo di attesa potrebbe ridurre l'utilizzo della CPU.

6.3.1 Algoritmi di *scheduling*

Andiamo ora ad analizzare i vari algoritmi di *scheduling* della CPU, partendo da quelli più semplici e passando a quelli più complessi.

First-Come, First-Served (FCFS)

L'algoritmo FCFS è il più semplice degli algoritmi di *scheduling*, i processi vengono eseguiti nell'ordine in cui arrivano nella coda di Ready. Questo algoritmo è semplice da implementare e non richiede alcun calcolo complesso. Tuttavia, ha alcuni svantaggi:

- Non tiene conto della lunghezza dei processi, quindi i processi lunghi possono bloccare l'esecuzione dei processi brevi.
- Può causare un aumento del tempo di attesa e del tempo di completamento per i processi brevi.

L'algoritmo FCFS è un algoritmo non preemptive, poiché un processo in esecuzione non può essere interrotto fino al suo completamento. Questo può portare a una bassa efficienza del sistema, poiché i processi brevi possono rimanere in attesa per lungo tempo.

Esempio consideriamo questi processi:

Processo	**Tempo di arrivo**	***CPU burst***
P1	0	24
P2	2	3
P3	4	3

Allora i tempi di attesa, completamento e di risposta sono:

Processo	T_r	T_w	T_t
P1	0	0	24
P2	24	22	25
P3	27	23	30

Dunque il tempo medio di attesa è:

$$T_{w,med} = \frac{T_{w,P1} + T_{w,P2} + T_{w,P3}}{3} = \frac{0 + 22 + 23}{3} = 15$$

Il tempo medio di completamento è:

$$T_{t,med} = \frac{T_{t,P1} + T_{t,P2} + T_{t,P3}}{3} = \frac{24 + 25 + 30}{3} = 26$$

Se però cambiano i tempi di arrivo dei processi, ad esempio:

Processo	**Tempo di arrivo**	***CPU burst***
P1	4	24
P2	0	3
P3	2	3

Allora i tempi di attesa, completamento e di risposta sono:

Processo	T_r	T_w	T_t
P1	2	2	26
P2	0	0	3
P3	1	1	4

Dunque il tempo medio di attesa è:

$$T_{w,med} = \frac{T_{w,P1} + T_{w,P2} + T_{w,P3}}{3} = \frac{2+0+1}{3} = 1$$

Il che è molto più veloce rispetto al caso precedente, nonostante il processo P1 sia più lungo. Questo è dovuto al fatto che i processi brevi sono stati eseguiti prima di P1, riducendo il tempo di attesa per P1.

Shortest Job First (SJF)

L'algoritmo SJF è un algoritmo di *scheduling* che assegna la CPU al processo con il *CPU burst* più breve. Questo algoritmo è in grado di ridurre il tempo di attesa e il tempo di completamento dei processi, poiché i processi brevi vengono eseguiti per primi. Questo algoritmo può essere implementato sia in modo *preemptive* che in modo non *preemptive*. Se è implementato in modo *preemptive*, il processo in esecuzione può essere interrotto se arriva un processo con un *CPU burst* più breve rispetto al CPU *burst* rimanente del processo in esecuzione (*Shortest-Remaining-Time-First* - SRTF). Se è implementato in modo non *preemptive*, il processo in esecuzione non può essere interrotto fino al suo completamento.

Esempio consideriamo questi processi:

Processo	**Tempo di arrivo**	***CPU burst***
P1	0	7
P2	2	4
P3	4	1
P4	5	4

Allora i processi verranno eseguiti in questo ordine:

- P1 (0-7)
- P3 (7-8)
- P2 (8-12)
- P4 (12-16)

Creando quindi i seguenti tempi di attesa, completamento e di risposta:

Processo	T_r	T_w	T_t
P1	0	0	7
P2	6	6	10
P3	3	3	4
P4	7	7	11

Dunque il tempo medio di attesa è:

$$T_{w,med} = \frac{T_{w,P1} + T_{w,P2} + T_{w,P3} + T_{w,P4}}{4} = \frac{0+6+3+7}{4} = 4$$

Se gli stessi processi arrivano nello stesso ordine ma in un sistema *preemptive*, i processi verranno eseguiti in questo ordine:

- P1 (0-2) - 5 rimasti
- P2 (2-4) - 2 rimasti

- P3 (4-5)
- P2 (5-7)
- P4 (7-11)
- P1 (11-16)

Creando quindi i seguenti tempi di attesa, completamento e di risposta:

Processo	T_r	T_w	T_t
P1	0	0	16
P2	5	5	10
P3	4	4	5
P4	7	7	11

Dunque il tempo medio di attesa è:

$$T_{w,med} = \frac{T_{w,P1} + T_{w,P2} + T_{w,P3} + T_{w,P4}}{4} = \frac{0 + 5 + 4 + 7}{4} = 4$$

Il che è lo stesso del caso non *preemptive*, ma in questo caso il tempo di attesa è più basso per i processi brevi, mentre il tempo di attesa per i processi lunghi è più alto.
Il principale problema di questo algoritmo è quello che è impossibile determinare con precisione il *CPU burst* di un processo, poiché questo dipende da molti fattori esterni. Viene dunque usata una media esponenziale (*exponential average*) per stimare il *CPU burst* di un processo. La media esponenziale è una media che dà più peso ai valori recenti rispetto ai valori più vecchi. La formula per calcolare la media esponenziale è:

$$T_{n+1} = \alpha T_n + (1 - \alpha) T_{n-1}$$

dove:

- T_{n+1} è il nuovo valore della media esponenziale
- T_n è il valore corrente del *CPU burst*
- T_{n-1} è il valore precedente del *CPU burst*
- α è un valore compreso tra 0 e 1 che determina il peso dei valori recenti rispetto ai valori più vecchi. Un valore di α vicino a 1 dà più peso ai valori recenti, mentre un valore di α vicino a 0 dà più peso ai valori più vecchi.

Scheduling a priorità

Nello *scheduling* a priorità ad ogni processo viene associata una priorità, i processi con priorità più alta vengono eseguiti per primi. Questo algoritmo può essere implementato sia in modo *preemptive* che in modo non *preemptive*. Un esempio di *scheduling* con priorità è il comando `nice` di `Linux`, che permette di modificare la priorità di un processo.

Politiche di assegnamento priorità L'assegnamento di un livello di priorità rispetto ad un altro può essere influenzato da fattori interni o esterni al `SO`. Come fattori interni troviamo ad esempio: limiti di tempo, requisiti di memoria, numero di file richiesti, ...
Possono anche essere influenzati da fattori esterni, come ad esempio l'importanza (umana) del processo, se per quel determinato processo si ha un guadagno economico o anche motivi politici, ...

Problemi Il principale problema dei `SO` con *scheduling* a priorità è la *starvation*, ovvero certi processi a bassa priorità potrebbero non essere mai eseguiti in quanto ci sono sempre processi con priorità più alta in attesa di essere eseguiti. Questo problema può essere risolto utilizzando una tecnica chiamata *aging*, che consiste nell'aumentare gradualmente la priorità dei processi a bassa priorità man mano che trascorrono del tempo in attesa. In questo modo, anche i processi a bassa priorità avranno la possibilità di essere eseguiti, evitando la *starvation*.

***Higher Response Ratio Next* - HRRN** L'HRRN è un algoritmo di *scheduling* a priorità, sempre non-*preentive*. In questo algoritmo la priorità di un processo viene calcolata in base al suo tempo di attesa e al suo *CPU burst*. La formula per calcolare la priorità di un processo è:

$$R = \frac{T_w + T_{CPU}}{T_{CPU}}$$

il maggiore valore di R avrà la priorità più alta. Questo algoritmo è in grado di ridurre il tempo di attesa e il tempo di completamento dei processi, poiché i processi brevi vengono eseguiti per primi. Inoltre se un processo è in attesa per lungo tempo, la sua priorità aumenta, evitando la *starvation* e la priorità è dunque dinamica. Se alla fine di un processo è arrivato un altro processo allora la priorità dei processi in attesa viene ricalcolata, andando a modificare l'ordine di esecuzione dei processi se necessario, oppure può anche essere ricalcolata alla fine di ogni CPU *burst* indipendentemente dal fatto che sia arrivato un nuovo processo o meno (dipende dalle implementazioni).

Esempio consideriamo questi processi:

Processo	**Tempo di arrivo**	***CPU burst***
P1	1	10
P2	0	2
P3	2	2
P4	2	1
P5	1	5

Allora il calcolo della priorità dei processi è il seguente:

Processo	$t = 0$	$t = 2$	$t = 7$	$t = 8$
P1	-	1+1/10	1+6/10	1+7/10
P2	1	-	-	-
P3	-	1 + 0/2	1 + 5/2	1 + 6/2
P4	-	1 + 0/1	1 + 5/1	-
P5	-	1 + 1/5	-	-

[1]

***Round Robin* (RR)**

L'algoritmo RR è un algoritmo *scheduling preemptive* che assegna un piccolo tempo di CPU chiamato *quantum* ($10 - 100$ ms) ad ogni processo in coda. Quando il tempo di CPU di un processo scade, il processo viene interrotto e messo in coda, e il *dispatcher* passa al processo successivo. Questo algoritmo è in grado di garantire una buona risposta per i processi interattivi, poiché i processi brevi vengono eseguiti rapidamente. Tuttavia bisogna fare una scelta sul valore del *quantum*, se questo è molto grande allora è esattamente come l'algoritmo *First Come First Serve*, se invece è molto piccolo allora il tempo di *scheduling* aumenta, poiché ad ogni passaggio di processo il *dispatcher* deve eseguire il *context switch* e passare alla modalità utente, il valore ottimale per il tempo q deve essere scelto in modo che il tempo q sia minore dell'80% dei *CPU burst*

Quanto-*context-switch* Esiste una relazione direttamente inversa tra il numero di *context switch* e il tempo di CPU *burst*, ovvero più è lungo il *CPU burst* e meno *context switch* ci sono. Questo è dovuto al fatto che ogni volta che si esegue un *context switch* il sistema deve salvare lo stato del processo corrente e caricare lo stato del processo successivo, il che richiede tempo e risorse. Se i processi hanno un *CPU burst* lungo, ci saranno meno *context switch* e quindi meno tempo sprecato.

Quanto-Tempo di attesa A differenza del numero di *context-switch*, il tempo di attesa non è legato direttamente al tempo di *quantum*, ma è legato al numero di processi in coda e ai tempi di esecuzione dei processi stessi. Se ci sono molti processi in coda, il tempo di attesa per ciascun processo aumenta, poiché

[1] Nota come nel seguente esempio il calcolo della priorità è stato fatto anche per i tempi $t = 7$ e $t = 8$, anche se non sono arrivati nuovi processi.

ogni processo deve attendere il completamento del *quantum* degli altri processi prima di essere eseguito nuovamente. Tuttavia, un *quantum* troppo piccolo può aumentare il tempo di attesa complessivo a causa dell'overhead introdotto dai frequenti *context-switch*.

Code multi-livello

Le code multi-livello sono una suddivisione della *ready queue* in più code, ognuna con un ruolo specifico ed un algoritmo di *scheduling* specifico. Ad esempio si potrebbe avere una coda per i processi interattivi, una coda per i processi batch e una coda per i processi in background, queste potrebbero essere gestite con RR, FCFS e SJF rispettivamente. In questo modo si può garantire una buona risposta per i processi interattivi e una buona efficienza per i processi batch. Questo però al costo di dover gestire lo *scheduling* tra le varie code. Quest'ultimo solitamente è gestito o come *time slice*, ovvero ogni coda ha una percentuale di tempo di CPU da utilizzare, oppure tramite priorità fissa, ovvero ogni coda ha una priorità fissa e la coda con priorità più alta viene eseguita, e liberata, prima delle altre.

Code multi-livello a *feedback* Mentre in una coda multi-livello tradizionale quando un processo viene assegnato ad una coda specifica non può cambiare coda, in una coda multi-livello a *feedback* i processi possono cambiare coda sulla base delle proprie caratteristiche di esecuzione, spesso ciò viene fatto per implementare l'*aging* e per evitare la *starvation*. I parametri sui quali si può agire per la configurazione dello *scheduler* sono: il numero delle code, l'algoritmo usato, i criteri di promozione e/o retrocessione dei processi ed i criteri per definire la coda iniziale di un processo.

Scheduling fair share

Dato che un programmatore sapendo che in alcuni casi se pianifica molti *threads* allora il suo processo potrebbe ottenere più tempo di CPU rispetto ad un altro processo, è stato implementato lo *scheduling fair share*, che lavora per applicazione e non per processo. Ad ogni applicazione viene assegnato una percentuale del tempo di CPU e i processi di quell'applicazione possono usare solo quella percentuale del tempo di CPU. Questo algoritmo è in grado di garantire che ogni applicazione riceva una quantità equa di tempo di CPU, evitando che un'applicazione monopolizzi le risorse del sistema. Tuttavia, questo algoritmo può causare un aumento del tempo di attesa per i processi ed un aumento del tempo di completamento, poiché i processi devono attendere che il loro turno arrivi. Inoltre, questo algoritmo può essere più complesso da implementare rispetto ad altri algoritmi di *scheduling*, poiché richiede la gestione delle percentuali di tempo di CPU per ogni applicazione e la loro assegnazione ai processi.

Contesto reale

Solitamente nei sistemi operativi moderni viene usato un mix di algoritmi di *scheduling*, ad esempio il Linux usa un algoritmo *completely fair scheduler* (CFS) che è una combinazione di RR e SJF. Questo algoritmo è in grado di garantire una buona risposta per i processi interattivi e una buona efficienza per i processi batch. Inoltre, questo algoritmo è in grado di adattarsi alle diverse condizioni del sistema, modificando dinamicamente le priorità dei processi in base al loro comportamento.

6.3.2 Valutazione degli algoritmi

Esistono diversi metodi per valutare le prestazioni degli algoritmi di *scheduling*, noi affrontiamo il modello deterministico ed il modello a reti di code.

Modello deterministico (Analitico)

Il modello deterministico è un metodo di valutazione degli algoritmi di *scheduling* che si basa su un insieme di processi con tempi di arrivo e tempi di esecuzione noti.[2] Questo metodo consente di calcolare le prestazioni degli algoritmi di *scheduling* in modo preciso e dettagliato, poiché i tempi di arrivo e i tempi di esecuzione sono noti in anticipo. Tuttavia, questo metodo non tiene conto delle variazioni nei tempi di arrivo e nei tempi di esecuzione dei processi, il che può portare a risultati poco realistici. Inoltre, questo metodo richiede una conoscenza dettagliata dei processi e delle loro caratteristiche, il che può essere difficile da ottenere in un sistema reale. Per questo motivo, il modello deterministico è più adatto per la valutazione di algoritmi di *scheduling* in ambienti controllati e non in ambienti reali.

[2] Come è stato fatto in precedenza con i vari esempi.

Modello a reti di code

Il modello a reti di code è un metodo di valutazione degli algoritmi di *scheduling* che si basa su un preciso numero di processi sempre uguali ed con tempi di `CPU` *burst*, `I/O` *burst* ed arrivo basati su una distribuzione di Poisson della quale si varia il parametro λ. Da questo modello è possibile calcolare il tempo medio di attesa, il tempo medio di completamento e il tempo medio di risposta per ogni algoritmo di *scheduling*. Questo metodo consente di valutare le prestazioni degli algoritmi di *scheduling* in modo più realistico rispetto al modello deterministico, poiché tiene conto delle variazioni nei tempi di arrivo e nei tempi di esecuzione dei processi. Tuttavia, questo metodo richiede una conoscenza dettagliata delle distribuzioni dei processi e delle loro caratteristiche, il che può essere difficile da ottenere in un sistema reale. Per questo motivo, il modello a reti di code è più adatto per la valutazione di algoritmi di *scheduling* in ambienti reali e non in ambienti controllati.

Simulazione

La simulazione è un metodo di valutazione degli algoritmi di *scheduling* che si basa sull'esecuzione di un insieme di processi in un ambiente simulato. Questo metodo consente di valutare le prestazioni degli algoritmi di *scheduling* in modo realistico, poiché tiene conto delle variazioni nei tempi di arrivo e nei tempi di esecuzione dei processi. Inoltre, questo metodo consente di testare gli algoritmi di *scheduling* in condizioni controllate ma pseudo-reali, il che può essere utile per la valutazione di algoritmi di *scheduling* in ambienti reali. Tuttavia, questo metodo richiede che il modello del sistema sia già disponibile, inoltre il suo uso seppur garantendo una buona valutazione delle prestazioni degli algoritmi di *scheduling* può essere costoso in termini di tempo e risorse.

Implementazione

L'implementazione di un algoritmo di *scheduling* è l'unico metodo certo per valutare le prestazioni di un algoritmo di *scheduling* in un sistema reale. Questo metodo consente di testare gli algoritmi di *scheduling* in condizioni reali e concrete e di valutare le loro prestazioni in un ambiente reale. Tuttavia, questo metodo richiede che questo algoritmo sia già codificato, inserito nel `SO` e solo dopo è possibile verificarne l'effettiva efficienza. Inoltre, questo metodo può essere costoso in termini di tempo e risorse, poiché richiede la modifica del `SO` e la sua re-installazione. Tuttavia, l'implementazione di un algoritmo di *scheduling* è il metodo più preciso e affidabile per valutare le prestazioni degli algoritmi di *scheduling* in un sistema reale.

Capitolo 7

Sincronizzazione dei processi

In questo capitolo andremo ad affrontare come gestire la sincronizzazione dei processi, ovvero come evitare che più processi accedano contemporaneamente a risorse condivise, causando inconsistenze nei dati o comportamenti imprevisti. La sincronizzazione è fondamentale in un sistema operativo per garantire che le operazioni sui dati condivisi siano eseguite in modo sicuro e prevedibile. Si parlerà di varie primitive di sincronizzazione ancora più complesse del meccanismi di `join` e `fork` già visti in precedenza. In particolare, ci concentreremo su mutex, semafori e variabili di condizione. Questi strumenti sono essenziali per la programmazione concorrente e ci permettono di gestire l'accesso alle risorse condivise in modo sicuro e controllato. Inoltre, esploreremo le problematiche legate alla sincronizzazione, come il deadlock e la starvation, e come evitarle attraverso tecniche di progettazione adeguate.

Modello astratto Il modello astratto di un processo è quello di produttore-consumatore dove un processo produce dati e un altro li consuma. Deve essere quindi garantita l'esecuzione concorrente di più processi, in modo che il produttore possa aggiungere ad un *buffer* condiviso e il consumatore possa prelevare dati da esso contemporaneamente. Questo *buffer* ha comunque dei vincoli, non deve essere permessa la scrittura se questo è pieno e se è vuoto non deve essere permesso il prelievo.

Buffer* `P/C`: Modello *software

Il *buffer* viene visto in maniera circolare con due puntatori, `in` ed `out` dove, `in` punta alla prossima posizione libera e `out` punta alla prossima posizione da prelevare. Il *buffer* vuoto ha `in` == `out` e il *buffer* pieno ha `out` == $(\texttt{in} + 1)\%n$. Nel corso per semplicità usiamo un contatore `counter` che indica il numero di elementi presenti nel *buffer* e quindi il *buffer* è vuoto se `counter == 0` e pieno se `counter == n`. Dunque con l'uso del contatore il processo produttore aumenta il contatore di uno e il processo consumatore lo diminuisce di uno, il problema di ciò è che l'istruzione `counter++` e `counter--` vengono divise in tre istruzioni assembly differenti:

```
mov eax, [counter] ; carica il contatore in eax
add eax, 1 ; incrementa il contatore
mov [counter], eax ; salva il contatore
```

Se due processi eseguono in parallelo il contatore potrebbe essere incrementato due volte o decrementato due volte, portando a risultati errati. Per evitare questo problema è necessario utilizzare un meccanismo di sincronizzazione che garantisca l'accesso esclusivo alla variabile `counter` durante l'operazione di incremento o decremento. Abbiamo appena visto un esempio di sezione critica costituita dalla lettura e scrittura della variabile `counter`.

7.1 Problema della sezione critica

La sezione critica è una porzione di codice che accede a una risorsa condivisa e deve essere eseguita in modo esclusivo da un solo processo alla volta. Per garantire che solo un processo alla volta possa eseguire la sezione critica. La soluzione deve soddisfare le seguenti proprietà:

- **Mutua esclusione**: Solo un processo alla volta può essere nella sezione critica.

- **Progresso**: Se nessun processo è nella sezione critica e ci sono processi in attesa, uno di essi deve essere in grado di entrare nella sezione critica. La decisione non può essere rimandata indefinitamente.
- **Attesa limitata**: Deve esistere un numero massimo di volte per cui un processo può essere bloccato in attesa di entrare nella sezione critica. Non deve essere possibile che un processo rimanga in attesa indefinitamente.

Struttura generica di un processo

La struttura generica di un processo che accede a una sezione critica è la seguente:

```
while (true) {
    // Sezione non critica
    // ... codice non critico ...

    // Sezione di entrata
    // ... codice per entrare nella sezione critica ...
    // Sezione critica
    // Sezione di uscita
    // ... codice per uscire dalla sezione critica ...
    // Sezione non critica
}
```

La sezione di entrata è il codice che consente al processo di entrare nella sezione critica, mentre la sezione di uscita è il codice che consente al processo di uscire dalla sezione critica. La sezione non critica è il codice che può essere eseguito in parallelo con altri processi senza problemi di sincronizzazione.

7.1.1 Soluzioni al problema della sezione critica

Quando si prova a risolvere il problema della sezione critica, è importante considerare le varie soluzioni e i loro vantaggi e svantaggi. Assumiamo di prima istanza che la sincronizzazione sia in ambiente globale, ovvero che esistono celle di memoria condivise tra i processi. In questo caso, possiamo sfruttare delle soluzioni *software* le quali richiedono solo un aggiunta di codice alle applicazioni esistenti, ma ciò non sfrutta nessun supporto da parte dell'*hardware* e/o dal sistema operativo. Le soluzioni *hardware* invece richiedono un supporto da parte dell'*hardware* e/o del sistema operativo, ma richiedono molte meno modifiche al codice delle applicazioni. Le soluzioni *hardware* sono più veloci e più efficienti rispetto a quelle *software*, ma richiedono un maggiore sforzo di implementazione e possono essere più complesse da gestire.

Algoritmo 1

```
PROCESS i;
int turn; /* Se turn == i processo i entra nella sezione critica */
while(1){
    while(turn != i); /* Attesa attiva */
    // Sezione critica
    turn = j; /* Passa il turno al processo j */
    // Sezione non critica
}
```

In questo modo si garantisce che solo un processo alla volta possa entrare nella sezione critica. Tuttavia, questo algoritmo presenta alcuni problemi, infatti se uno dei due processi termina, l'altro processo rimarrà bloccato in attesa dopo che questo ha passato il suo turno (non viene rispettato il progresso). Inoltre, questo algoritmo richiede una stretta alternanza tra i processi, infatti finché entrambi i processi non vogliono entrare nella sezione critica, non è possibile che uno dei due possa entrare. Infine, questo algoritmo non è adatto per più di due processi, poiché richiede una variabile `turn` per ogni coppia di processi.

Algoritmo 2

```
PROCESS i;
bool flag[2]; /* flag[i] == true se il processo i vuole entrare*/
while(true){
    flag[i] = true; /* Indica che il processo i vuole entrare */
    while(flag[j]); /* Attesa attiva */
    // Sezione critica
    flag[i] = false; /* Indica che il processo i e' uscito */
    // Sezione non critica
}
```

In questo algoritmo nel momento nel quale un processo vuole entra nella sezione critica, imposta il proprio flag a `true` e attende che l'altro processo imposti il proprio flag a `false`. In questo modo si garantisce che solo un processo alla volta possa entrare nella sezione critica. Qui il problema del progresso è risolto, ma questa soluzione presenta un problema di "stallo" (starvation), infatti se un processo imposta il proprio flag a `true` e poi avviene un timeout ed il processo viene messo in attesa, l'altro processo non potrà mai entrare nella sezione critica pur impostando il proprio flag a `true`.
Se l'impostazione del flag avvenisse dopo l'attesa attiva, allora non si presenterebbe il problema dello stallo, ma si presenterebbe un problema di mutua esclusione.

Algoritmo 3

```
PROCESS i;
int turn; /* Se turn == i processo i entra nella sezione critica */
bool flag[2]; /* flag[i] == true se il processo i vuole entrare*/
while(true){
    flag[i] = true; /* Indica che il processo i vuole entrare */
    turn = j; /* Passa il turno al processo j */
    while(flag[j] && turn == j); /* Attesa attiva */
    // Sezione critica
    flag[i] = false; /* Indica che il processo i e' uscito */
    // Sezione non critica
}
```

Questo algoritmo risolve il problema dello stallo, e garantisce la mutua esclusione, questo grazie alla doppia condizione di attesa. Infatti, se il processo i vuole entrare nella sezione critica, imposta il proprio flag a `true` e poi passa il turno al processo j. Se il processo j è in attesa e ha il turno, il processo i non può entrare nella sezione critica, se invece il processo j non vuole entrare nella sezione critica il suo flag sarà `false` e il processo i potrà entrare nella sezione critica.

Algoritmo del fornaio

L'idea di questo algoritmo è quella che quando un processo vorrebbe entrare nella sezione critica, questo deve prima scegliere un numero, il processo con il numero più basso entra nella sezione critica. Se due processi scelgono lo stesso numero, il processo con l'identificativo più basso entra per primo. Questo algoritmo se implementato correttamente garantisce tre proprietà fondamentali:

```
PROCESS i;
int number[N]; /* number[i] == numero scelto dal processo i */
while(true){
    number[i] = Max(number[0], number[1], ..., number[N-1]) + 1; /* Sceglie un numero */
    for (j = 0; j < N; j++){
        while(number[j] != 0 && number[j] < number[i]); /* Attesa attiva */
    }
    // Sezione critica
    number[i] = 0; /* Indica che il processo i e' uscito */
    // Sezione non critica
}
```

Se si fà particolare attenzione alla condizione di attesa, si può notare che il processo i entra nella sezione critica solo se il numero scelto è il più basso tra tutti i processi. Inoltre, se due processi scelgono lo stesso numero, il processo con l'identificativo più basso entra per primo. Questo algoritmo è molto semplice e intuitivo, ma presenta alcuni problemi di correttezza, infatti con questa implementazione non è garantita la mutua esclusione, in quanto due processi potrebbero scegliere lo stesso numero e entrare nella sezione critica contemporaneamente.

Algoritmo del fornaio v0.2

```
PROCESS i;
int number[N]; /* number[i] == numero scelto dal processo i */
int turn; /* Se turn == i processo i entra nella sezione critica */
bool choosing[N]; /* choosing[i] == true se il processo i sta scegliendo un numero */
while(true){
    choosing[i] = true; /* Indica che il processo i sta scegliendo un numero */
    number[i] = Max(number[0], number[1], ..., number[N-1]) + 1; /* Sceglie un numero */
    choosing[i] = false; /* Indica che il processo i ha scelto un numero */
    for (j = 0; j < N; j++){
        while(choosing[j]); /* Attesa attiva */
        while(number[j] != 0 && number[j] < number[i]); /* Attesa attiva */
    }
    // Sezione critica
    number[i] = 0; /* Indica che il processo i e' uscito */
    // Sezione non critica
}
```

In questo algoritmo, il processo i prima di scegliere un numero imposta il proprio flag `choosing` a `true`, in questo modo gli altri processi sanno che il processo i sta scegliendo un numero e non devono entrare nella sezione critica. Dopo aver scelto un numero, il processo i imposta il proprio flag `choosing` a `false`, in questo modo gli altri processi sanno che il processo i ha scelto un numero e possono entrare nella sezione critica. Questo algoritmo garantisce la mutua esclusione, il progresso e l'attesa limitata.

Algoritmo del fornaio v1.0

Per ovviare ai problemi di correttezza dell'algoritmo del fornaio, è possibile utilizzare un algoritmo che utilizza una variabile `int number[N]` che contiene l'ultimo processo scelto, in questo modo si garantisce che l'ultimo processo scelto non vada a ri-occupare subito la sezione critica.

```
PROCESS i;
bool choosing[N]; /* choosing[i] == true se il processo i sta scegliendo un numero */
int number[N]; /* ultimo numero scelto dal processo i */
while(1){
    choosing[i] = true; /* Indica che il processo i sta scegliendo un numero */
    number[i] = Max(number[0], number[1], ..., number[N-1]) + 1; /* Sceglie un numero */
    choosing[i] = false; /* Indica che il processo i ha scelto un numero */
    for (j = 0; j < N; j++){
        while(choosing[j]); /* Attesa che il processo j ha scelto un numero */
        while(number[j] != 0 && (
            number[j] < number[i]
                ||
            number[j] == number[i]
            && i < j)); /* Attesa attiva */
    }
    // Sezione critica
    number[i] = 0; /* Indica che il processo i e' uscito */
    // Sezione non critica
}
```

In questo algoritmo, viene risolto il problema della mutua esclusione, del progresso e dell'attesa limitata.

7.1.2 Soluzioni *hardware*

Le soluzioni *hardware* sono più veloci e più efficienti rispetto a quelle *software*, ma richiedono un maggiore sforzo di implementazione e possono essere più complesse da gestire. Un metodo "semplice" come gestione *hardware* è quello di disabilitare gli *interrupt* durante l'esecuzione della sezione critica. Questo metodo è semplice da implementare, ma presenta alcuni problemi, infatti se il test per l'accesso alla sezione critica viene eseguito in molto tempo gli *interrupt* dovrebbero essere disabilitati per molto tempo, causando un degrado delle prestazioni del sistema. Inoltre, questo metodo non è adatto per sistemi multiprocessore, poiché gli *interrupt* possono essere disabilitati solo per il processore corrente e non per gli altri processori. Una alternativa è quello di rendere l'operazione di accesso e scrittura alla variabile atomica, ovvero che impieghi un unico ciclo di *clock*. Esempio di questo è l'istruzione `test&set` oppure lo `compare&swap`.

Test&Set

L'istruzione `test&set` è un'istruzione atomica che consente di testare e impostare una variabile in un'unica operazione. Questa istruzione è utilizzata per implementare la mutua esclusione nei sistemi operativi. La sintassi dell'istruzione `test&set` è la seguente:

```
bool test_and_set(bool &var){
    bool temp;
    temp = var; /* Salva il valore corrente di var in temp */
    var = true; /* Imposta var a true */
    return temp; /* Restituisce il valore precedente di var */
}
```

Tutte e tre le operazioni sono eseguite in un'unica istruzione atomica, quindi non possono essere interrotte da altri processi. Quando la funzione viene chiamata, il valore corrente di `var` viene salvato in `temp`, quindi `var` viene impostato a `true` e infine il valore precedente di `var` viene restituito. Se il valore precedente di `var` era `false`, significa che la sezione critica è libera e il processo può entrarvi. Se il valore precedente di `var` era `true`, significa che la sezione critica è occupata e il processo deve attendere.

```
PROCESS i;
bool lock; /* lock == true se la sezione critica e' occupata */
while(true){
    while(test_and_set(lock)); /* Attesa attiva */
    // Sezione critica
    lock = false; /* Indica che il processo i e' uscito */
    // Sezione non critica
}
```

In questo esempio, il processo i utilizza l'istruzione `test&set` per testare e impostare la variabile `lock`. Se la sezione critica è occupata, il processo i rimane in attesa finché non diventa libera. Dato che l'istruzione `test&set` è atomica, non ci sono problemi di mutua esclusione e il processo i può entrare nella sezione critica in modo sicuro. Inoltre solo il primo processo che vede `lock == false` può entrare nella sezione critica, gli altri processi rimarranno in attesa finché non diventa libera. Una volta che il processo i esce dalla sezione critica, imposta `lock` a `false` per indicare che la sezione critica è ora libera. Il problema di questo algoritmo è che l'attesa finita non è garantita, infatti non sono presenti meccanismi per evitare che un processo rimanga in attesa indefinitamente in quanto un processo potrebbe entrare ed uscire dalla sezione critica più volte prima che il processo i possa entrarvi.

Swap

Uso dell'istruzione `test&set` L'istruzione `swap` è un'altra istruzione atomica che consente di scambiare il valore di due variabili in un'unica operazione. Questa istruzione è utilizzata per implementare la mutua esclusione nei sistemi operativi. La sintassi dell'istruzione `swap` è la seguente:

```
void swap(bool &a, bool &b){
    bool temp;
    temp = a; /* Salva il valore corrente di a in temp */
    a = b; /* Imposta a al valore di b */
    b = temp; /* Imposta b al valore di temp */
}
```

Tutte e tre le operazioni sono eseguite in un'unica istruzione atomica, quindi non possono essere interrotte da altri processi. Quando la funzione viene chiamata, il valore corrente di `a` viene salvato in `temp`, quindi `a` viene impostato al valore di `b` e infine `b` viene impostato al valore di `temp`. In questo modo, i valori di `a` e `b` vengono scambiati in un'unica operazione atomica.

```
PROCESS i;
bool lock; /* lock == true se la sezione critica e' occupata */
while(true){
    bool dummy = true; /* Dummy per evitare di passare un riferimento */
    do
        swap(lock, dummy); /* Attesa attiva */
    while(dummy); /* Dummy == false se la sezione critica e' occupata */
    // Sezione critica
    lock = false; /* Indica che il processo i e' uscito */
    // Sezione non critica
}
```

In questo esempio, il processo i utilizza l'istruzione `swap` per scambiare il valore di `lock` con un valore `dummy`. Se la sezione critica è occupata, il processo i rimane in attesa finché non diventa libera. Dato

che l'istruzione `swap` è atomica, non ci sono problemi di mutua esclusione e il processo i può entrare nella sezione critica in modo sicuro. Inoltre solo il primo processo che vede `lock == false` può entrare nella sezione critica, gli altri processi rimarranno in attesa finché non diventa libera. Una volta che il processo i esce dalla sezione critica, imposta `lock` a `false` per indicare che la sezione critica è ora libera. Il problema di questo algoritmo è che l'attesa finita non è garantita, infatti non sono presenti meccanismi per evitare che un processo rimanga in attesa indefinitamente in quanto un processo potrebbe entrare ed uscire dalla sezione critica più volte prima che il processo i possa entrarvi.

Test&Set con attesa limitata

```
PROCESS i;
bool waiting[N]; /* waiting[i] == true se il processo i sta aspettando */
bool lock; /* lock == true se la sezione critica e' occupata */
while(1){
    waiting[i] = true; /* Indica che il processo i sta aspettando */
    bool key = true;
    while(waiting[i] && key){ /* Attesa attiva */
        key = test_and_set(lock); /* Prova ad entrare */
    }
    waiting[i] = false; /* Indica che il processo sta per entrare */
    // Sezione critica
    int j = (i+1) % N; /* Prendo in considerazione il prossimo processo */
    while(j != i && !waiting[j]){ // Controllo se il processo j sta aspettando
        j = (j+1) % N; /* Prendo in considerazione il prossimo processo */
    }
    if(j == i){ /* Se non ci sono altri processi in attesa */
        lock = false; /* Indica che la sezione critica e' libera */
    }else{
        waiting[j] = false; /* Indica che il processo j puo' entrare */
    }
    // Sezione non critica
}
```

In questo esempio, il processo i utilizza l'istruzione `test&set` per testare e impostare la variabile `lock`. Se la sezione critica è occupata, il processo i rimane in attesa finché non diventa libera. Dato che l'istruzione `test&set` è atomica, non ci sono problemi di mutua esclusione e il processo i può entrare nella sezione critica in modo sicuro. Inoltre solo il primo processo che vede `lock == false` può entrare nella sezione critica, gli altri processi rimarranno in attesa finché non diventa libera, oppure il processo che è appena uscito dalla sezione critica gli passi il turno. Se nessun processo è in attesa allora l'ultimo processo che è uscito dalla sezione critica libera la sezione critica impostando `lock` a `false`. In questo modo si garantisce che solo un processo alla volta possa entrare nella sezione critica e che l'attesa sia limitata.

7.2 Semafori

Uso dell'istruzione `swap` I semafori sono una primitiva di sincronizzazione la quale è un numero intero non negativo, che può essere incrementato e decrementato. Per eseguire queste operazioni, i semafori utilizzano due operazioni fondamentali: `wait` - `P` e `signal` - `V`. La prima operazione decrementa il valore del semaforo, se questo è maggiore di zero, altrimenti il processo viene messo in attesa. La seconda operazione incrementa il valore del semaforo e, se ci sono processi in attesa, uno di essi viene risvegliato. Esistono due tipi principali di semafori, i semafori binari e i semafori generici. I semafori binari possono assumere solo i valori 0 e 1, mentre i semafori generici possono assumere qualsiasi valore intero non negativo. I semafori binari possono essere implementati tramite i semafori generici, ma non viceversa.

Semafori binari

L'implementazione concettuale delle funzioni `wait` e `signal` è la seguente:

```
P(semaphore s){
    while(s == false){ /* Attesa attiva */
        // Non fa nulla
    }
    s = false; /* Imposta il semaforo a false */
}
V(semaphore s){
    s = true; /* Imposta il semaforo a true */
}
```

In questo esempio, la funzione `P` attende che il semaforo sia libero (ovvero che il suo valore sia `true`) e poi lo imposta a `false`. La funzione `V` imposta il semaforo a `true`, indicando che la sezione critica è ora libera.

Semafori generici

L'implementazione concettuale delle funzioni `wait` e `signal` è la seguente:

```
P(semaphore s){
    while(s <= 0){ /* Attesa attiva */
        // Non fa nulla
    }
    s--; /* Decrementa il semaforo */
}
V(semaphore s){
    s++; /* Incrementa il semaforo */
}
```

In questo esempio, la funzione `P` attende che il semaforo sia maggiore di zero e poi lo decrementa. La funzione `V` incrementa il semaforo, indicando che la sezione critica è ora libera.

Note

Per essere efficienti le funzioni `P` e `V` devono essere implementate in modo atomico, in modo che non possano essere interrotte da altri processi. Come abbiamo visto però l'implementazione atomica delle funzioni `P` e `V` non è garantita. Andiamo dunque a vedere come garantire l'atomicità delle funzioni `P` e `V` utilizzando i semafori.

```
/* Inizializzato s a true */
P(bool &s){
    bool key = false;
    do{
        swap(s, key); /* Attesa attiva */
    }while(key == false);
    s = false; /* Imposta il semaforo a false */
}
V(bool &s){
    s = true; /* Imposta il semaforo a true */
}
```

In questo esempio, la funzione `P` utilizza l'istruzione `swap` per testare e impostare il semaforo. Se il semaforo è occupato, il processo rimane in attesa finché non diventa libero. La funzione `V` imposta il semaforo a `true`, indicando che la sezione critica è ora libera. In questo modo si garantisce che le funzioni `P` e `V` siano atomiche e che non possano essere interrotte da altri processi.

Implementazione semafori interi

Coi semafori interi l'implementazione si complica, dobbiamo comunque garantire l'atomicità delle funzioni `P` e `V`, ma dobbiamo anche garantire che il semaforo non possa assumere valori negativi ed che sia possibile incrementarlo di uno senza che durante questo tempo un altro processo possa decrementarlo. Per fare ciò possiamo usiamo due semafori binari, `mutex` e `delay` questi vengono usati rispettivamente per garantire la mutua esclusione e per garantire che le operazioni siano eseguite in modo atomico. La funzione `P` viene implementata come segue:

```
P(semaphore s){
    P(mutex); /* Acquisisce il semaforo mutex */
    s = s - 1; /* Decrementa il semaforo */
    if(s < 0){ /* Se il semaforo e' negativo */
        V(mutex); /* Rilascia il semaforo mutex */
        P(delay); /* Attende il semaforo delay */
    }else{
        V(mutex); /* Rilascia il semaforo mutex */
    }
}
```

In questo esempio, la funzione `P` acquisisce il semaforo `mutex` per garantire la mutua esclusione. Poi decrementa il semaforo e controlla se il suo valore è negativo. Se il valore è negativo, rilascia il semaforo `mutex` e attende il semaforo `delay` (che è Inizializzato a `false`). Se il valore è maggiore o uguale a zero, rilascia il semaforo `mutex` e termina. La funzione `V` viene implementata come segue:

```
V(semaphore s){
    P(mutex); /* Acquisisce il semaforo mutex */
    s = s + 1; /* Incrementa il semaforo */
    if(s <= 0){ /* Se il semaforo e' negativo */
        V(delay); /* Rilascia il semaforo delay */
    }
    V(mutex); /* Rilascia il semaforo mutex */
}
```

In questo esempio, la funzione `V` acquisisce il semaforo `mutex` per garantire la mutua esclusione. Poi incrementa il semaforo e controlla se il suo valore è negativo. Se il valore è negativo, rilascia il semaforo `delay` (che è Inizializzato a `false`). Se il valore è maggiore o uguale a zero, rilascia il semaforo `mutex` e termina. Nota come acquisire il semaforo `mutex` prima di modificare il semaforo `s` e rilasciarlo dopo averlo modificato, dato che questo è inizializzato a `true`, serva per garantire che ogni operazione sul semaforo `s` sia eseguita in modo atomico.
Notiamo come nell'implementazione, anche se risolviamo il problema della sezione critica, non risolviamo il problema del *busy-waiting* ovvero l'attesa attiva. Infatti, se un processo entra nella sezione critica quando il suo tempo di `CPU`-*burst* termina e il processo viene messo in attesa, il semaforo `mutex` rimarrà occupato fino a quando il processo non verrà risvegliato andando a far sprecare al processo corrente tempo di `CPU`.

Implementazione senza *busy-waiting*

Per evitare il problema del *busy-waiting* possiamo utilizzare una lista `s.list` che contiene la lista dei processi in attesa. La funzione `P` viene implementata come segue:

```
P(semaphore s){
    P(mutex); /* Acquisisce il semaforo mutex */
    s.value = s.value - 1; /* Decrementa il semaforo */
    if(s.value < 0){ /* Se il semaforo e' negativo */
        append(process I, s.List); /* Aggiunge il processo alla lista */
        sleep() & V(mutex); /* Rilascia il semaforo mutex */
    }else{
        V(mutex); /* Rilascia il semaforo mutex */
    }
}
```

In questo esempio, la funzione `P` acquisisce il semaforo `mutex` per garantire la mutua esclusione. Poi decrementa il semaforo e controlla se il suo valore è negativo. Se il valore è negativo, aggiunge il processo alla lista dei processi in attesa e lo mette in attesa. Se il valore è maggiore o uguale a zero, rilascia il semaforo `mutex` e termina. La funzione `V` viene implementata come segue:

```
V(semaphore s){
    P(mutex); /* Acquisisce il semaforo mutex */
    s.value = s.value + 1; /* Incrementa il semaforo */
    if(s.value <= 0){ /* Se il semaforo e' negativo */
        PCB *p = remove(s.List); /* Rimuove il primo processo dalla lista */
        wakeup(p) & V(mutex); /* Rilascia il semaforo mutex */
    }else{
        V(mutex); /* Rilascia il semaforo mutex */
    }
}
```

In questo esempio, la funzione `V` acquisisce il semaforo `mutex` per garantire la mutua esclusione. Poi incrementa il semaforo e controlla se il suo valore è negativo. Se il valore è negativo, rimuove il primo processo dalla lista dei processi in attesa e lo risveglia. Se il valore è maggiore o uguale a zero, rilascia il semaforo `mutex` e termina. Notare come non abbiamo discusso l'assenza di *busy-waiting* sui semafori binari `mutex` e `delay`, ma generalmente il processo non rimane in attesa attiva, ma gli viene inviato un segnale di *sleep* e viene messo in attesa. Questo algoritmo è più efficiente rispetto all'implementazione precedente, poiché non richiede l'attesa attiva e consente di risparmiare tempo di `CPU`. Tuttavia, questo algoritmo richiede un maggiore sforzo di implementazione e può essere più complesso da gestire.
Per gestire l'implementazione senza *busy-waiting* per i semafori booleani `mutex` e `delay` dobbiamo far conto sul `SO`, infatti questo può gestire questi o disabilitando gli `interrupt`, oppure ignorando il *busy-waiting* su `mutex` in quanto questo è un semaforo binario ed il suo cambio è molto veloce.
Per garantire l'assenza di *starvation* la lista deve essere implementata tramite `FIFO`

Applicazioni dei semafori

Nel caso di voler eseguire in sequenza i processi A e poi B allora scriveremo il seguente codice:

```
/* S=0 */
PROCESS A;
    // Esecuzione del processo A
    V(S); /* Indica che il processo A e' uscito */
PROCESS B;
    // Attesa del processo A
    P(S); /* Attende il processo A */
    // Esecuzione del processo B
```

In ogni caso col semaforo `S` inizializzato a 0, il processo A eseguirà per primo e poi il processo B, sia che il tempo in `CPU` venga assegnato prima ad A e poi a B, sia che il tempo in `CPU` venga assegnato prima a B e poi a A. Infatti, il processo B non potrà entrare nella sezione critica finché il processo A non avrà rilasciato il semaforo `S`.
Se invece vogliamo eseguire in sequenza i processi $A, B, A, B, \ldots$ allora scriveremo il seguente codice:

```
/* S=0 S1=1 */
PROCESS A;
while(1){
    P(S1);
    // Esecuzione del processo A
    V(S); /* Indica che il processo A e' uscito */
}
PROCESS B;
while(1){
    P(S);
    // Esecuzione del processo B
    V(S1); /* Indica che il processo B e' uscito */
}
```

In questo caso, il processo A e il processo B si alternano nell'esecuzione. Infatti, il processo A entra nella sezione critica e poi rilascia il semaforo `S` per consentire al processo B di entrare nella sezione critica. Una volta che il processo B esce dalla sezione critica, rilascia il semaforo `S1` per consentire al processo A di entrare nella sezione critica. Questo ciclo continua fino a quando i processi non terminano.[1]

[1] Ulteriori esempi ed applicazioni trattate a lezione sono state omesse, in quanto anche se trattate approfonditamente a lezione, queste sono la merita applicazione del concetto di semaforo.

Capitolo 8

Deadlock

Quando l'uso contemporaneo delle risorse da parte di più processi porta a una situazione in cui nessun processo può proseguire senza aver accesso a una risorsa, si verifica un *deadlock*.
Un classico esempio di *deadlock* è quello di due processi devono acquisire due risorse per completare il loro lavoro. Se il processo A acquisisce la risorsa 1 e il processo B acquisisce la risorsa 2, entrambi i processi non possono proseguire perché ciascuno aspetta che l'altro rilasci la risorsa di cui ha bisogno. Generalizzando sono necessarie quattro condizioni affinché si verifichi un *deadlock*:

1. **Mutua esclusione**: almeno una risorsa deve essere assegnata in modo esclusivo a un processo. Se una risorsa è assegnata a un processo, nessun altro processo può accedervi.
2. ***Hold & Wait***: un processo che detiene almeno una risorsa sta aspettando di acquisire altre risorse.
3. **Nessuna *preemption***: una risorsa non può essere forzatamente rimossa da un processo che la detiene. La risorsa può essere rilasciata solo volontariamente dal processo che la detiene.
4. **Attesa circolare**: esiste un insieme di processi P1, P2, ..., Pn tali che P1 sta aspettando una risorsa detenuta da P2, P2 sta aspettando una risorsa detenuta da P3, ..., e Pn sta aspettando una risorsa detenuta da P1.

Per analizzare come una risorsa viene allocata ai processi, è utile costruire un `RAG` (*Resource Allocation Graph*), un grafo diretto in cui i nodi rappresentano processi e risorse. Le risorse sono rappresentate da cerchi, mentre i processi sono rappresentati da quadrati. Un arco diretto da un processo a una risorsa indica che il processo sta aspettando la risorsa, mentre un arco diretto da una risorsa a un processo indica che la risorsa è attualmente assegnata al processo. Se esiste un ciclo nel grafo, si verifica un *deadlock*, se ci sono più istanze di una risorsa ed esiste un ciclo allora non è detto che ci sia un *deadlock*.

8.1 Prevenzione del deadlock

Abbiamo diverse strategie per prevenire il *deadlock*:

- Prevenzione Statica
- Prevenzione Dinamica
- Rilevamento (*Detection*) e Recupero (*Recovery*)
- Struzzo - Non preoccuparsi del *deadlock*

8.1.1 Prevenzione Statica

La prevenzione statica del *deadlock* implica l'uso della scrittura di codice che eviti che si verifichi una delle quattro condizioni necessarie per il *deadlock*.

Mutua esclusione

Non è possibile evitare la mutua esclusione, poiché è una condizione necessaria per l'uso delle risorse.

Hold & Wait

Soluzioni Per evitare questa condizione, un processo deve acquisire tutte le risorse di cui ha bisogno prima di iniziare a eseguire. Inoltre un processo può acquisire una risorsa solo se non detiene già altre risorse.

Problemi Questo approccio può portare a un utilizzo inefficiente delle risorse e a un aumento del tempo di attesa.

No preemption

Soluzioni Per evitare questa condizione, un processo che richiedere una risorsa non disponibile deve rilasciare tutte le risorse che detiene, oppure può cedere la risorsa che detiene su richiesta di un altro processo.

Problemi Questo approccio è fattibile solo per risorse digitali come CPU e memoria, ma non per risorse fisiche come stampanti o dischi.

Attesa circolare

Soluzioni Per evitare questa condizione, è possibile assegnare una priorità ad ogni risorsa, in modo che un processo possa acquisire solo risorse con priorità superiore a quelle già detenute. In questo modo si evita la formazione di cicli di attesa in quanto un processo non può acquisire una risorsa con priorità inferiore a quelle già detenute.

Problemi Solitamente è difficile definire una priorità per le risorse, e questo approccio può portare a un utilizzo inefficiente delle risorse.

8.1.2 Prevenzione Dinamica

Dato che la prevenzione statica del *deadlock* può portare a un utilizzo inefficiente delle risorse, in quanto queste tecniche impostano dei vincoli rigidi sul modo in cui i processi possono acquisire le risorse, la prevenzione dinamica punta a basare la prevenzione del *deadlock* sulla base delle richieste delle risorse da parte dei processi. In questo modo, i processi possono acquisire le risorse in modo più flessibile, evitando il *deadlock* senza compromettere l'efficienza delle risorse.

Prerequisito Per utilizzare la prevenzione dinamica del *deadlock*, è necessario che il sistema operativo conosca a priori il caso peggiore, ovvero il numero massimo di risorse che ogni processo richiederà.

Il mentenimento del *safe state*

Definizione Lo stato di assegnazione delle risorse, calcolato come il numero di istanze di risorse disponibili e il numero di istanze di risorse assegnate a ciascun processo su le richieste massime, è definito *safe state* se esiste una *safe sequence* di processi.

Safe sequence Una sequenza di processi $(P_1, \ldots, P_n)$ è definita *safe sequence* se ogni processo P_i, le risorse che richiede P_i possono essere soddisfatte usando le risorse disponibili e le risorse rilasciate dai processi $P_1, \ldots, P_{i-1}$.
Se non esiste una *safe sequence* di processi, il sistema è in uno stato non sicuro (*unsafe state*) e si potrebbe verificare un *deadlock*, ma non è garantito che si verifichi.

La prevenzione

Per mantenere il sistema in uno stato sicuro, il sistema operativo attua degli algoritmi per decidere se una richiesta di risorse da parte di un processo può essere soddisfatta senza portare il sistema in uno stato non sicuro. Dunque ad ogni richiesta di risorse da parte di un processo, il sistema operativo attua questa verifica.

Svantaggi La prevenzione dinamica del *deadlock* porta comunque ad una riduzione dell'efficienza del sistema, poiché il sistema operativo deve eseguire calcoli complessi per determinare se una richiesta di risorse può essere soddisfatta senza portare il sistema in uno stato non sicuro. Inoltre, la prevenzione dinamica riduce l'uso delle risorse rispetto ad un non-uso della prevenzione del *deadlock*, poiché i processi devono attendere che il sistema operativo calcoli se la richiesta di risorse può essere soddisfatta.

Implenentazione

L'implementazione viene effettuata solitamente tramite o l'uso del `RAG`, oppure tramite l'uso dell' "algoritmo del banchiere".

Uso di `RAG` L'algoritmo che prevede l'uso di un `RAG` funziona solo se si ha una istanza per ogni risorsa. Il `RAG` viene esteso con archi di "rivendicazione", ovvero archi che rappresentano una "possibilità di richiesta" di una risorsa da parte di un processo, $P_i \to R_j$ se P_i in futuro potrebbe richiedere R_j. Questo comporta che ogni processo quando viene creato deve dichiarare le risorse che potrebbe richiedere in futuro.
In un secondo momento, il sistema operativo allocherà la risorsa R_j al processo P_i se l'arco $R_j \to P_i$ non crei un ciclo nel grafo. Se si verifica un ciclo, il sistema operativo non assegnerà la risorsa al processo P_i in quanto porterebbe il sistema in uno stato non sicuro. Se il grafo non ha cicli, il sistema operativo assegnerà la risorsa al processo P_i e rimuoverà l'arco di rivendicazione $P_i \to R_j$.

Algoritmo del banchiere L'algoritmo del banchiere è un algoritmo meno efficiente rispetto all'uso del `RAG`, ma funziona con qualunque numero di istanze di risorse.
Ogni processo è un cliente che possono richiedere del credito dalla banca (ovvero il sistema operativo) ogni risorsa allocabile è vista come del denaro. Ovviamente il `SO` non permetterà a tutti i clienti di raggiungere il massimo limite personale contemporaneamente. Ciò infatti porterebbe ad una situazione di *deadlock*.
Anche in questo algoritmo ogni processo dichiarare il numero massimo di risorse che andrà ad richiedere, ad ogni richiesta di allocazione si verifica se questa mantiene lo stato *safe*. Per fare ciò si sfruttano due algoritmi differenti, uno per l'allocazione ed uno per la verifica dello stato.

Listing 8.1: Algoritmo del banchiere

```
int available[m]// numero di istanze di R_i disponibili
int max[n][m]   // matrice delle richieste massime
int alloc[n][m] // matrice delle risorse allocate
int need[n][m]  // matrice delle richieste residue
                // (need[i][j] = max[i][j] - alloc[i][j])

void request(int req_vec[]){
    if(req_vec[i] > need[i][j])
        error("Richiesta maggiore del massimo richiesto");
    if(req_vec[i] > available[j])
        wait();
    available[] = available[] - req_vec[];
    alloc[i][] = alloc[i][] + req_vec[];
    need[i][] = need[i][] - req_vec[];
    if(!state_safe()){
        available[] = available[] + req_vec[];
        alloc[i][] = alloc[i][] - req_vec[];
        need[i][] = need[i][] + req_vec[];
        wait();
    }
}

bool state_safe(){
    int work[m] = available[];
    bool finish[n] = {false, false, ..., false};
    int i;
    while(finish != {true, true, ..., true}){
        // cerca P_i che non e' terminato
        // e che puo' terminare
```

```
        for(i=0; i<n && (finish[i] || need[i][] > work[]); i++);
        if(i == n) // non esiste P_i che puo' terminare
            return false; // non e' in uno stato sicuro
        work[] = work[] + alloc[i][]; // rilascia le risorse di P_i
        finish[i] = true; // P_i e' terminato
    }
    return true; // e' in uno stato sicuro
}
```

Notare come questo algoritmo sia molto dispendioso e che deve essere eseguito ogni volta che un processo richiede una risorsa.[1]

8.1.3 Rilevamento del *deadlock* & ripristino

Gli algoritmi di rilevamento del *deadlock* sono progettati per rilevare la presenza di un *deadlock* nel sistema e per ripristinare il sistema a uno stato sicuro. Questi algoritmi sono meno restrittivi rispetto alla prevenzione (statico o dinamico) e consentono ai processi di acquisire le risorse in modo più flessibile. Tuttavia, richiedono un monitoraggio costante del sistema per rilevare la presenza di un *deadlock* ed eventualmente correggere la situazione, questi possono comportare un sovraccarico significativo.
Esistono due approcci principali per il rilevamento del *deadlock*: il rilevamento basato su `RAG` di attesa e l'"algoritmo di rilevazione".

Rilevamento basato su `RAG`

Anche in questo caso il `RAG` funziona solo se si ha una istanza per ogni risorsa. Il grafo di attesa è un grafo composto da soli processi, ogni arco nel grafo ($P_i \to P_j$) indica che il processo P_i sta aspettando una risorsa detenuta dal processo P_j. Se esiste un ciclo nel grafo, si verifica un *deadlock*. Il sistema operativo può utilizzare questo grafo per rilevare la presenza di un *deadlock* e per determinare quali processi sono coinvolti nel *deadlock*. Tuttavia, questo approccio richiede che il sistema operativo monitori costantemente il grafo

Algoritmo di rilevamento

L'algoritmo di rilevamento esplora ogni possibile sequenza di allocazione dei processi che ancora non hanno terminato, se la sequenza và a buon fine, allora non c'è *deadlock* altrimenti potrebbe esserci.
L'algoritmo è il seguente:

Listing 8.2: Algoritmo di rilevamento

```
int available[m]// numero di istanze di R_i disponibili
int alloc[n][m] // matrice delle risorse allocate
int req_vec[n][m] // matrice delle richieste

void check(){
    int work[m] = available[]; // risorse disponibili
    bool finish = {false, false, ..., false}; // processi terminati
    bool found = true; // trovato un processo che puo' terminare
    while(found){
        found = false; // resetta la variabile
        for(int i=0; i<n && !found; i++){
            // cerca P_i che non e' terminato
            // e che puo' terminare
            if(!finish[i] && req_vec[i][] <= work[]){
                // Assumo che P_i possa terminare senza deadlock
                // dunque rilascia le risorse senza deadlock
                work[] = work[] + alloc[i][]; // rilascia le risorse di P_i
                finish[i] = true; // P_i terminera' senza problemi
                found = true; // trovato un processo che puo' terminare
            }
        }
    }
```

[1] Nota dell'autore: È utile imparare questo algoritmo a memoria, in quanto verrà richiesto in sede di esame.

```
    // Se finish[i] == false, per uno o piu' P_i
    // allora i processi P_i sono in deadlock
}
```

Ripristino

Prima di procedere al ripristino del *deadlock*, bisogna capire ogni quando eseguire l'algoritmo di rilevamento del *deadlock*, solitamente questo viene fatto o dopo ogni richiesta, o dopo N secondi, oppure quando l'uso della `CPU` scende sotto una soglia T preimpostata.
Una volta che il *deadlock* è stato rilevato, il sistema operativo deve decidere come ripristinare il sistema a uno stato sicuro, ci sono diversi approcci per il ripristino del *deadlock*.

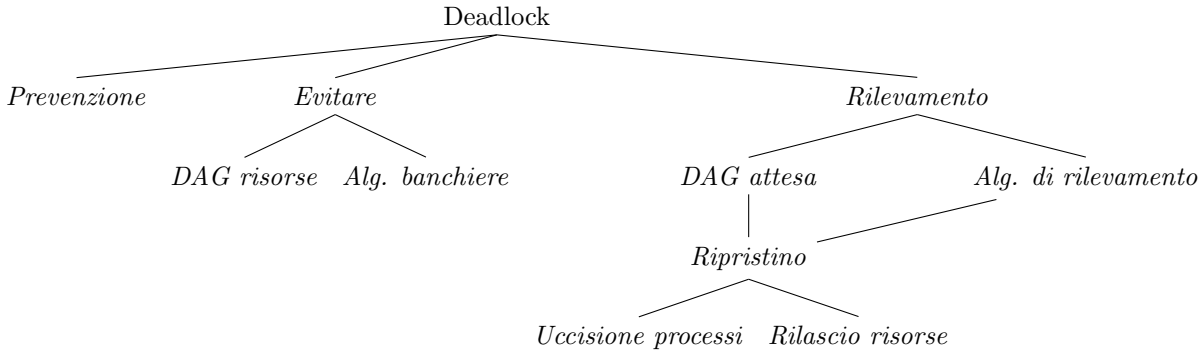
Uccisione di processi La soluzione più comune per il ripristino del *deadlock* è quella di uccidere uno o più processi coinvolti nel *deadlock*. Ci sono due approcci principali per uccidere i processi:

- Uccidere tutti i processi coinvolti nel ciclo - Soluzione molto costosa e drastica, ma è la più semplice da implementare.
- Uccidere selettivamente i processi coinvolti nel ciclo - Questa soluzione è più complessa da implementare, ma può essere più efficiente in termini di utilizzo delle risorse, ma non dal punto di uso in quanto deve essere ri-eseguita la rilevazione dopo ogni uccisione.

Rilascio di risorse Un'altra soluzione per il ripristino del *deadlock* è quella di rilasciare alcune risorse detenute dai processi coinvolti nel *deadlock*. Questa soluzione presente un problema: i processi che vengono forzati a rilasciare le risorse potrebbero non essere in grado di completare il loro lavoro normalmente, portando a un aumento del tempo di attesa e a una riduzione dell'efficienza del sistema. Inoltre se in situazione di *deadlock* rilascio sempre le risorse coinvolte in uno stesso processo, si potrebbe verificare una situazione di *starvation*, per ovviare questo si deve tener conto dei *rollback* effettuati in precedenza.

8.1.4 Conclusioni

Concludendo possiamo riassumere le varie soluzioni al *deadlock* del seguente diagramma:



Ognuno dei metodi ha i suoi vantaggi e svantaggi, e la scelta del metodo dipende dalle esigenze specifiche del sistema e dalle risorse disponibili. Principalmente però nessuno dei metodi di gestione è ottimale e delle soluzioni combinate di più metodi possono essere più efficaci se dividiamo le risorse in classi quali: risorse interne, memoria, risorse di processo (eg. file) e *swap* applicando ad ognuna di queste la soluzione più efficiente: ordinamento di risorse per la prima, prelazione per la memoria, prevenzione dinamica per i file, e prevenzione con pre-allocazione (*hold&wait*) per la *swap*.
In linea principale però la soluzione più semplice ed adottata dalla maggior parte dei `SO` è quella dello "struzzo" ovvero ignoriamo il problema del *deadlock* in quanto si verificano raramente e tutte le soluzioni sono costose o con algoritmi sbagliati.

Capitolo 9

Gestione della memoria

In questo capitolo si parlerà della gestione della memoria nei sistemi operativi, in particolare si analizzeranno gli spazi di indirizzamento, l'allocazione contigua, la paginazione, la segmentazione e la segmentazione con paginazione. Si analizzeranno anche le problematiche legate alla gestione della memoria, e le tecniche di allocazione della memoria. Si parlerà anche della memoria virtuale e delle tecniche di swapping, e si analizzeranno i problemi legati alla frammentazione della memoria. Infine si parlerà della gestione della memoria nei sistemi operativi moderni, e delle tecniche di allocazione della memoria nei sistemi operativi a *micro-kernel*.

9.1 Introduzione

Per rendere un sistema operativo efficiente la condivisione della memoria deve essere gestita in modo da evitare conflitti tra i processi ed efficienti.

Problematiche Le problematiche che un buon sistema operativo deve affrontare in merito alla gestione della memoria sono:

- Allocazione della memoria ai singoli *job*
- Protezione dello spazio di indirizzamento
- Condivisione dello spazio di indirizzamento
- Nei sistemi operativi moderni: Gestione della memoria virtuale (*swap*)

Tutti questi problemi devono obbligatoriamente essere affrontati in modo, oltre che efficiente, anche sicuro per evitare che un processo possa accedere alla memoria di un altro processo o di un dispositivo e compromettere la stabilità del sistema e/o la sicurezza dei dati.

9.1.1 Dal programma al processo

Ogni programma prima di essere eseguito deve essere caricato in memoria e trasformato in un processo. Il sistema operativo deve successivamente prelevare le istruzioni in memoria sulla base del `PC` (*Program Counter*) e caricare i dati necessari per l'esecuzione del programma. Quando questo è terminato, il sistema operativo deve liberare la memoria occupata dal processo e restituirla al sistema. La creazione di un processo è quindi un'operazione complessa che richiede l'allocazione di risorse e la gestione della memoria.
Prima di trasformare un programma in un processo, ci sono diverse operazioni da eseguire, in ognuna di queste operazioni possono esserci diverse semantiche degli indirizzi (spazio logico / spazio fisico), solitamente quando si scrive il codice sorgente si fa riferimento a indirizzi logici, mentre quando il codice viene eseguito si deve fare riferimento a indirizzi fisici.
Il compilatore traduce gli indirizzi logici in indirizzi simbolici ri-locabili, il linker o il loader si occupano di tradurre gli indirizzi ri-locabili in indirizzi assoluti, infine il sistema operativo si occupa di tradurre gli indirizzi assoluti in indirizzi fisici.
L'insieme delle operazioni che vengono eseguite per trasformare gli indirizzi simbolici in indirizzi assoluti è detto ***binding***.

Binding & Indirizzi

Il *binding* può avvenire in diversi momenti, e a seconda di quando avviene il *binding* si può dividere in *binding* statico e *binding* dinamico:

- ***Binding* a *compile time***: il *binding* avviene durante la compilazione del programma, gli indirizzi simbolici vengono tradotti in indirizzi assoluti e il programma viene caricato in memoria con gli indirizzi assoluti. Questo tipo di *binding* è molto veloce, ma non permette di modificare il programma una volta compilato. Questo tipo di *binding* è della categoria del *binding* statico.
- ***Binding* a *load time***: il *binding* avviene durante il caricamento del programma in memoria, gli indirizzi simbolici vengono tradotti in indirizzi assoluti e il programma viene caricato in memoria con gli indirizzi assoluti. Questo tipo di binding richiede che gli indirizzi siano ri-locabili, e in questo caso vengono usati indirizzi relativi alla posizione di memoria in cui il programma viene caricato. Questo tipo di *binding* è della categoria del *binding* statico in quanto se il programma viene caricato in una posizione di memoria diversa da quella prevista, il sistema operativo deve modificare gli indirizzi assoluti in indirizzi relativi alla nuova posizione di memoria.
- ***Binding* a *run time***: il *binding* avviene durante l'esecuzione del programma, gli indirizzi simbolici vengono tradotti in indirizzi assoluti ed il programma può essere caricato in memoria in qualsiasi posizione e spostato in memoria durante l'esecuzione. Questo tipo di *binding* è della categoria del *binding* dinamico ma richiede del supporto *hardware* aggiuntivo.

Collegamento (*linking*)

Un altro passaggio prerequisito alla creazione di un processo è il *linking*, ovvero l'associazione di un programma a tutte le librerie e i moduli necessari. Questo passaggio può avvenire in due modi:

- ***Linking* statico**: il *linking* avviene durante la compilazione del programma, le librerie e i moduli necessari vengono inclusi per intero nel programma e il programma viene caricato in memoria con tutte le librerie e i moduli necessari. Questo tipo di *linking* è molto veloce, ma aumenta la dimensione del programma e richiede più memoria. Inoltre se una libreria o un modulo viene aggiornato, il programma deve essere ricompilato per utilizzare la nuova versione della libreria o del modulo, oltre al fatto che se non si usano tutte le funzioni di una libreria, il programma occupa più memoria del necessario.
- ***Linking* dinamico**: il *linking* avviene durante l'esecuzione del programma, le librerie e i moduli necessari vengono caricati in memoria solo quando sono necessari e il programma viene caricato in memoria con i riferimenti alle librerie e ai moduli necessari. Questo tipo di *linking* è più lento, ma riduce la dimensione del programma e richiede meno memoria. Inoltre se una libreria o un modulo viene aggiornato, il programma non deve essere ricompilato per utilizzare la nuova versione della libreria o del modulo.

Il *linking* statico è più veloce, ma richiede più memoria e non permette di aggiornare le librerie e i moduli senza ricompilare il programma. Il *linking* dinamico è più lento, ma richiede meno memoria e permette di aggiornare le librerie e i moduli senza ricompilare il programma.

Caricamento (*loading*)

Il caricamento è l'operazione che permette di caricare un programma in memoria e prepararlo per l'esecuzione. Il caricamento può avvenire in due modi:

- ***Loading* statico**: l'intero programma viene caricato in memoria in un'unica operazione, il programma viene caricato in memoria con gli indirizzi assoluti e il programma viene eseguito. Questo tipo di *loading* è molto veloce, ma richiede che il programma sia di dimensioni fisse e non permette di modificare il programma una volta caricato in memoria.
- ***Loading* dinamico**: il caricamento di alcuni moduli avviene in modo dinamico, ovvero il programma viene caricato in memoria in più operazioni quando sono necessari. Questo tipo di *loading* è più lento, ma permette di caricare solo i moduli necessari e di modificare il programma una volta caricato in memoria. Inoltre se una porzione di codice non viene mai eseguita, non viene caricata in memoria e quindi non occupa spazio in memoria.

Il *loading* statico è più veloce, ma richiede che il programma sia di dimensioni fisse e non permette di modificare il programma una volta caricato in memoria. Il *loading* dinamico è più lento, ma permette di caricare solo i moduli necessari e di modificare il programma una volta caricato in memoria.

9.1.2 Spazi di indirizzamento

Come accennato in precedenza esistono due tipi di indirizzamento: l'indirizzamento logico, gestito dalla `CPU` e l'indirizzamento fisico, gestito dalla memoria. L'associazione tra i due indirizzamenti nel caso del *binding* statico è semplice, in quanto gli indirizzi logici e fisici coincidono. Nel caso del *binding* dinamico invece, l'associazione tra i due indirizzamenti è più complessa, in quanto gli indirizzi logici e fisici non coincidono e devono essere tradotti. Per gestire questa traduzione viene usato un processore detto `MMU` (*Memory Management Unit*), che si occupa di tradurre gli indirizzi logici in indirizzi fisici. Questo componente agisce a *run time* e a partite da un indirizzo base, pre-impostato per il processo in esecuzione, lo somma all'indirizzo logico per ottenere l'indirizzo fisico. Questo processo è detto **relocazione dinamica** e permette di eseguire più processi in memoria senza conflitti.

Considerazioni Bisogna considerare che in un sistema multi-programmato non è possibile conoscere in anticipo dove un processo può essere posizionato in memoria, e quindi è necessario che il sistema operativo gestisca la memoria in modo da evitare conflitti tra i processi. Inoltre l'esigenza di avere lo *swap* impedisce di poter usare indirizzi ri-locati in modo statico. Ne consegue che la ri-locazione dinamica viene usato per sistemi più "complessi" e la gestione è eseguita dal `SO`, mentre la ri-locazione statica viene usata solo per applicazioni specifiche ed il `SO` non può fare granché in materia di gestione della memoria.

9.2 Schemi di gestione della memoria

La gestione della memoria è un'altra funzione fondamentale del sistema operativo, che deve garantire l'allocazione della memoria ai processi in modo da migliorare l'uso della memoria e garantire la protezione dei processi.
Esistono diversi schemi di gestione della memoria, ognuno con i propri vantaggi e svantaggi. I principali schemi di gestione della memoria sono:

- Allocazione contigua
- Paginazione
- Segmentazione
- Segmentazione con paginazione

Anche se nelle soluzioni reali viene usata della memoria virtuale

9.2.1 Allocazione contigua

L'allocazione contigua è lo schema di gestione della memoria più semplice il quale prevede che la memoria sia suddivisa in partizioni, le quali possono essere o fisse, o variabili. Se la dimensione dell'immagine di un processo occupa $10Kb$ allora questo occuperà $10Kb$ consecutivi.

Allocazione con partizioni fisse Quando si usa l'allocazione con partizioni fisse, la memoria viene suddivisa in partizioni di dimensioni fisse (solitamente usando partizioni con dimensioni di potenze di 2) e ogni processo viene caricato in una partizione di dimensioni fisse. Questo tipo di allocazione è semplice e veloce, ma può portare a problemi di assegnazione di memoria a diversi *job*, se ad esempio un processo richiede $10Kb$ e la partizione più grande disponibile è di $8Kb$, il processo non può essere caricato in memoria finché non viene liberata una partizione di dimensioni sufficienti. Inoltre se un processo richiede meno memoria di quella disponibile in una partizione, la memoria rimanente non può essere utilizzata da altri processi, portando a problemi di frammentazione interna.

***Scheduling* a lungo termine**In questo caso lo *scheduling* viene eseguito o con più code (una per ogni partizione) oppure con una coda semplice. Nel primo caso ogni processo viene associato alla partizione più piccola che lo può contenere e viene eseguito in modo da non superare la dimensione della partizione. Nel secondo caso il processo che viene allocato potrebbe o essere il primo della coda (`FCFS`) che và ad occupare la partizione più piccola disponibile, oppure viene eseguita una scansione della coda e una determinata partizione viene assegnato o al processo che richiede la memoria più simile alla dimensione della partizione (*best-fit-only*) oppure viene assegnato il primo *job* che può stare nella partizione (*first-available-fit*).
In tutti i casi abbiamo diverse problematiche, nelle code per ogni partizione dopo che un *job* è entrato in coda non può essere spostato in un'altra coda, e quindi non può essere spostato in una partizione più piccola o più grande se questa è libera e non ci sono altri processi in attesa. Problema simile riguarda il `FCFS` in quanto se non è disponibile una partizione abbastanza grande per il primo processo, allora tutti i processi attendono anche se ci sono delle partizioni libere. Mentre nel caso del *best-fit-only* e del *first-available-fit* bisogna considerare che ogni volta bisogna analizzare l'intera coda.
In ogni caso nel caso di allocazione con partizioni fisse il grado di multiprogrammazione è limitato dal numero di partizioni disponibili, e quindi il numero di processi che possono essere eseguiti contemporaneamente è limitato dal numero di partizioni disponibili. Inoltre esiste un grande problema di frammentazione sia interna che esterna, in quanto se un processo richiede meno memoria di quella disponibile in una partizione, la memoria rimanente non può essere utilizzata da altri processi, portando a problemi di frammentazione interna. Inoltre se un processo richiede più memoria di quella disponibile in una partizione, il processo non può essere caricato in memoria finché non viene liberata una partizione di dimensioni sufficienti, portando a problemi di frammentazione esterna.

Allocazione con partizioni variabili Quando si usa l'allocazione con partizioni variabili, la memoria viene suddivisa in partizioni di dimensioni variabili e ogni processo viene caricato in una partizione di dimensioni variabili. Questo tipo di allocazione è più flessibile rispetto all'allocazione con partizioni fisse, ma può portare a problemi di assegnazione di memoria a diversi *job*, e può portare a problemi di frammentazione esterna.
In questo caso il `SO` deve tener conto oltre alle partizioni allocate anche delle *buche* ovvero delle aree di memoria libere. Quando arriva un processo viene allocato a questo la prima buca che lo può contenere, e se la buca è più grande del necessario viene creata una nuova buca. Se invece la buca è più piccola del necessario, bisogna provvedere a deframmentare la memoria, ovvero spostare i processi in modo da creare una buca più grande.

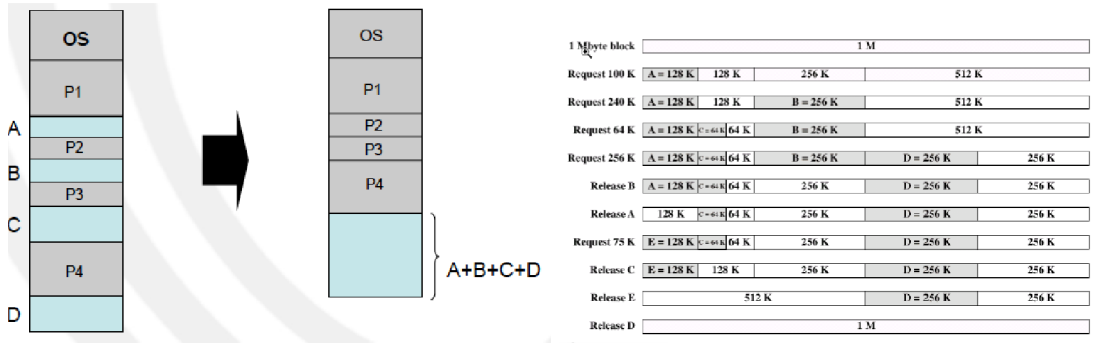
Stategie di *scheduling*Per l'allocazione con partizioni variabili esistono diverse strategie di *scheduling*:

- *First-fit*: viene allocata la prima buca che può contenere il processo, e se la buca è più grande del necessario viene creata una nuova buca.
- *Best-fit*: viene allocata la buca più piccola che può contenere il processo, e se la buca è più grande del necessario viene creata una nuova buca.
- *Worst-fit*: viene allocata la buca più grande che può contenere il processo, e se la buca è più grande del necessario viene creata una nuova buca.

Tipicamente *first-fit* è la migliore.

Tecniche di deframmentazione La deframmentazione è l'operazione che permette di spostare i processi in memoria in modo da creare buche più grandi che possono contenere processi più grandi. Esistono diverse tecniche di deframmentazione, le principali sono:

- **Compattazione**: ogni processo viene spostato in memoria in modo da creare buche più grandi, e le buche vengono unite in un'unica buca. Questo tipo di deframmentazione è molto veloce, ma richiede che i processi siano spostati in memoria e comunque richiede tempo.
- ***Buddy-system***: la memoria viene suddivisa in blocchi di dimensioni potenze di 2. Ogni volta che arriva un processo si procede a dividere la memoria in due finché una ulteriore divisione non permetterebbe di allocare il processo. Quando viene liberato un blocco di memoria, il `SO` verifica se il blocco può essere unito con un altro blocco adiacente, e se è possibile i due blocchi vengono uniti in un unico blocco. Questo tipo di deframmentazione è molto veloce, ma persiste il problema della frammentazione interna



(a) Compattazione (b) *Buddy-system*

Figura 9.1: Tecniche di deframmentazione

9.2.2 Paginazione

La paginazione è una tecnica per eliminare la frammentazione, sia esterna che interna, si basa sull'idea che lo spazio di indirizzamento di un processo può essere non-contiguo e che la memoria fisica può essere suddivisa in blocchi di dimensioni fisse detti *frame*, mentre la memoria logica viene divisa in blocchi di dimensioni fisse detti *page*.
Se vogliamo eseguire un programma il quale necessita n pagine allora bisogna trovare n *frame* liberi, indipendentemente che questi siano contigui o meno. Per trovare le pagine libere il `SO` mantiene una tabella delle pagine, la quale mappa ogni processo alle sue pagine associate agli *frame* fisici.

Tabulaizone degli indirizzi

Ogni indirizzo logico (generato dalla `CPU`) viene suddiviso in due parti:

- **Numero di pagina (p)**: il numero della pagina a cui appartiene l'indirizzo logico (se la pagina ha dimensione 2^n e la memoria ha dimensione 2^m allora il numero di pagina è di $m - n$ bit)
- **Offset (d)**: la posizione all'interno della pagina (se la pagina ha dimensione 2^n allora l'offset è di n bit)

L'indirizzo logico viene quindi rappresentato come una coppia di numeri (p, d), dove p è il numero della pagina e d è l'offset. Per ottenere l'indirizzo fisico, il `SO` deve tradurre il numero di pagina nell'indirizzo base della pagina fisica e sommarlo all'offset. L'indirizzo fisico viene quindi rappresentato come la coppia (f, d) dove f è il numero dell'*frame* e d è l'offset.

Implementazioni

Dato che l'efficienza è fondamentale in quanto la traduzione degli indirizzi deve essere eseguita in tempo reale, esistono diverse soluzioni per l'implementazione della traduzione degli indirizzi.

Implementazione tramite registri Tutte le *entry* della tabella sono archiviate in un registro, e ogni volta che viene generato un indirizzo logico, il `SO` deve accedere al registro per ottenere l'indirizzo fisico. Questo tipo di implementazione è molto veloce, ma questo al costo di un numero ridotto di *entry* ed un allungamento dei tempi di *context-switch*.

Implementazione in memoria Al posto di memorizzare l'intera tabella delle pagine in un registro, il `SO` memorizza solo un puntatore alla base della tabella delle pagine in un registro, ed un opzionale registro per la lunghezza della tabella delle pagine. Risulta quindi più veloce il *context-switch* in quanto la variazione riguarda solo il registro `PTBR` (*Page Table Base Register*) e il registro `PTLR` (*Page Table Length Register*). Questo però richiede due accessi alla memoria, uno per leggere la tabella delle pagine e uno per leggere l'*frame* fisico. Per ovviare a questo problema si usa una *cache* della tabella delle pagine, chiamata `TLB` (*Translation Look-aside Buffer*) che memorizza le traduzioni più recenti. La `TLB` è una memoria veloce che memorizza le traduzioni più recenti e permette di ridurre il numero di accessi alla memoria. Quando viene generato un indirizzo logico, il `SO` verifica se la traduzione è presente nella `TLB`, se è presente la traduzione viene eseguita in modo veloce ($< 10\%$ rispetto al *lookup* in memoria),

altrimenti viene eseguita la traduzione in memoria.
Il tempo di accesso effettivo medio alla memoria è dato dalla seguente formula:

$$T_{eff} = T_{TLB} + (1 - p) \cdot T_{mem} + p \cdot (T_{TLB} + T_{mem})$$

Dove T_{TLB} è il tempo di accesso alla `TLB`, T_{mem} è il tempo di accesso alla memoria e p è la probabilità che la traduzione sia presente nella `TLB`. Se la `TLB` è molto veloce e la probabilità che la traduzione sia presente è alta, il tempo di accesso effettivo alla memoria sarà molto basso.

Protezione

La protezione della memoria è implementata associando ad ogni *frame* un bit di protezione, che indica se il *frame* è accessibile o meno. Se un processo tenta di accedere a un *frame* non accessibile, il `SO` genera un'eccezione e il processo viene terminato. Esistono poi i bit di accesso che determinano se una pagina è modificabile o meno oppure se è eseguibile o meno.

Pagine condivise

La paginazione permette di avere più cope virtuali di una pagina in memoria ma una stessa copia fisica, ovvero più processi possono condividere la stessa pagina fisica. Questo permette condividere pagine (*read-only*) tra più processi mantenendo però la protezione della memoria.

Spazio di indirizzamento

Modernamente gli indirizzi sono o a `32` o a `64` bit, e quindi lo spazio di indirizzamento è di 2^{32} o 2^{64} byte, il quale è molto maggiore dello spazio di indirizzamento fisico. Per questo motivo vengono usati dei meccanismi per gestire il problema della dimensione della tabella delle pagine:

- Paginazione della tabella delle pagine: la tabella delle pagine viene suddivisa in pagine di dimensioni fisse, e ogni pagina della tabella delle pagine viene memorizzata in memoria. Questo permette di ridurre la dimensione della tabella delle pagine e di gestire lo spazio di indirizzamento in modo più efficiente.
- Tabella delle pagine invertita: la tabella delle pagine viene memorizzata in memoria e ogni *entry* della tabella delle pagine contiene l'indirizzo fisico della pagina. Questo permette di ridurre la dimensione della tabella delle pagine e di gestire lo spazio di indirizzamento in modo più efficiente.

La paginazione della tabella delle pagine è più complessa da implementare, ma permette di gestire lo spazio di indirizzamento in modo più efficiente. La tabella delle pagine invertita è più semplice da implementare, ma richiede più memoria e può portare a problemi di frammentazione.
La paginazione delle tabella delle pagine è una tecnica simile alla paginazione multi-livello, in quanto la tabella delle pagine viene suddivisa in più livelli di pagine. In questo caso la tabella delle pagine è suddivisa in più livelli di pagine, e ogni pagina della tabella delle pagine contiene un puntatore alla pagina successiva.