



# UNIVERSITÀ DI PISA

Dipartimento di Ingegneria  
Corso di Laurea Triennale in Ingegneria Informatica

Appunti

## Sistemi Operativi

**Professori:**

Prof. Avvenuti  
Prof. Palmieri

**Autore:**

Enea Passardi

---

Anno Accademico 2025/2026

## Indice

<b>Unimap</b> . . . . .	4
<b>Introduzione</b> . . . . .	7
<b>I Concetti introduttivi</b>	8
<b>1 Definizione di sistema operativo</b>	9
1 Gestione delle risorse e sicurezza . . . . .	10
2 Application Programming Interface . . . . .	11
<b>2 Cenni storici sui sistemi operativi</b>	12
1 I primi sistemi batch . . . . .	13
2 Sistemi di <i>Spooling</i> . . . . .	13
3 Sistemi multiprogrammati . . . . .	14
4 Sistemi <i>time-sharing</i> . . . . .	15
5 Sistemi in tempo reale . . . . .	15
<b>3 Architettura di Von-Neuman</b>	17
1 Processore . . . . .	17
2 Memoria . . . . .	18
3 Periferiche . . . . .	19
4 Interruzioni . . . . .	19
<b>II Struttura dei sistemi operativi</b>	21
<b>4 Gestione dei processi</b>	22
1 Processi . . . . .	23
2 Stato di un processo . . . . .	24
3 Struttura del descrittore di processo e code di processi . . . . .	25
4 Cambio di contesto . . . . .	26
5 Creazione e terminazione di processi . . . . .	27
<b>5 Schedulazione dei processi</b>	28
1 Algoritmi di scheduling a breve termine . . . . .	29
1.1 First-Come First-Served . . . . .	30
1.2 Shortest-Job First . . . . .	31
1.3 Short-Remaining-Time First . . . . .	32
1.4 Round-Robin . . . . .	33
2 Schedulazione a code multiple . . . . .	34

3	Schedulazione di sistemi in <i>tempo reale</i> . . . . .	34
4	Thread . . . . .	37
<b>6</b>	<b>Sincronizzazione tra processi</b>	<b>39</b>
1	Problema della mutua esclusione . . . . .	39
2	Problemi di comunicazione . . . . .	41
3	Semafori . . . . .	42
4	Elementi di sincronizzazione nel modello a scambio di messaggio . . . . .	44
5	Problema dei lettori e scrittori . . . . .	45
6	Problema dei filosofi a cena . . . . .	46
7	Monitor . . . . .	47
<b>7</b>	<b>Deadlock</b>	<b>51</b>
1	Grafo di assegnazione delle risorse . . . . .	51
2	Metodi per la gestione delle situazioni di stallo . . . . .	53
3	Meccanismi di <i>avoidance</i> dello stallo . . . . .	53
3.1	Algoritmo del banchiere . . . . .	54
4	Rilevamento delle situazioni di stallo . . . . .	55
5	Prevenzioni delle situazioni di stallo . . . . .	58
<b>III</b>	<b>Aspetti architetturali sui sistemi operativi</b>	<b>59</b>
<b>8</b>	<b>Gestione della memoria</b>	<b>60</b>
1	Implementazione della memoria . . . . .	61
2	Rilocazione della memoria . . . . .	61
3	Segmentazione nella memoria virtuale . . . . .	62
4	Tecniche di gestione della memoria . . . . .	63
4.1	Memoria partizionata . . . . .	63
4.2	Memoria segmentata . . . . .	66
4.3	Memoria paginata . . . . .	69
5	Segmentazione e Paginazione . . . . .	71
6	Gestione degli stati di un processo . . . . .	73
<b>9</b>	<b>Gestione delle periferiche</b>	<b>75</b>
1	Gestore di un dispositivo . . . . .	75
2	Driver di un dispositivo . . . . .	77
3	Gestione della temporizzazione . . . . .	79
4	Il processo esterno . . . . .	80
5	Gestione e organizzazione interna dei dischi . . . . .	81
<b>10</b>	<b>filesystem</b>	<b>84</b>
1	Organizzazione logica del filesystem . . . . .	85
2	Accesso al filesystem . . . . .	85
3	Organizzazione fisica . . . . .	86
3.1	Allocazione contigua . . . . .	87
3.2	Allocazione a lista concatenata . . . . .	88
3.3	Allocazione a indice . . . . .	89
4	filesystem di unix . . . . .	89
4.1	Strutture dati per l'accesso ai file . . . . .	91
<b>11</b>	<b>Meccanismi di protezione</b>	<b>92</b>
1	Politiche e meccanismi . . . . .	93
2	Domini di protezione . . . . .	93
3	Matrice degli accessi . . . . .	94
3.1	Realizzazione della matrice degli accessi . . . . .	94
4	Sicurezza multilivello . . . . .	95

<b>IV Laboratorio</b>	<b>98</b>
<b>12 Struttura e gestioni di un sistema unix</b>	<b>99</b>
1    Gestione degli utenti e dei permessi . . . . .	101
2    Gesitone degli archivi . . . . .	103
<b>13 Gestione dei processi con la libreria sys.h</b>	<b>104</b>
1    System call per la gestione dei processi . . . . .	104
2    Interazione e sincronizzazione tra processi . . . . .	105
<b>14 Gestione dei processi da terminale</b>	<b>108</b>
1    Organizzazione dei processi in Unix . . . . .	109
2    Gestione dei processi da terminale . . . . .	109
<b>15 Gestione dei Thread con la libreria pthread.h</b>	<b>111</b>
1    Struttura di un thread . . . . .	111
2    Mutua esclusione e Sincronizzazione . . . . .	112
<b>16 Il filesystem linux</b>	<b>116</b>
1    Primitive per l'accesso ai file . . . . .	118
2    Comunicazione mediante pipe . . . . .	118
<b>17 Pilotare applicazioni</b>	<b>119</b>
<b>V Appendici</b>	<b>121</b>
<b>A Classificazione delle architetture</b>	<b>122</b>
1    Metriche di prestazione . . . . .	124

# Unimap

1. **Mar 23/09/2025 08:30-11:30 (3:0 h)** lezione: Introduzione al corso: obiettivi formativi, contenuti, prerequisiti, modalità d'esame. Principali funzioni di un sistema operativo: facilità di programmazione e portabilità dei programmi; gestione delle risorse; protezione. (MARCO AVVENUTI)
2. **Mer 24/09/2025 13:30-15:30 (2:0 h)** lezione: Cenni storici sull'evoluzione dei sistemi operativi. I primi sistemi di elaborazione. I primi sistemi batch. Sistemi batch multiprogrammati. Sistemi time-sharing. Sistemi in tempo reale. Personal computer. (MARCO AVVENUTI)
3. **Gio 25/09/2025 10:30-13:30 (3:0 h)** lezione: Richiami di architetture dei sistemi di elaborazione: Modello di Von Neumann. Un calcolatore basato sul microprocessore mED. Blocchi funzionali e loro collegamento. Organizzazione dello spazio di memoria e dello spazio di I/O. Tipo di transazioni sul bus. La CPU: registri generali, registri di stato (IP, PSW, SP), il bus. Il ciclo della CPU. Instruction set, esempi di istruzioni in linguaggio assembler. (MARCO AVVENUTI)
4. **Mar 30/09/2025 08:30-11:30 (3:0 h)** lezione: La gerarchia di memoria, memorie cache. Chiamata di sottoprogramma. Controllo di programma e interruzione di programma. Gestione delle interruzioni e loro importanza nella costruzione di un sistema operativo. Direct Memory Access. Meccanismi di protezione. (MARCO AVVENUTI)
5. **Mer 01/10/2025 13:30-15:30 (2:0 h)** laboratorio: Breve introduzione storica ai sistemi UNIX, introduzione al filesystem, shell testuale. Comandi di base per la navigazione nel sistema e la manipolazione di file e cartelle. Esercitazioni in aula (MAURIZIO PALMIERI)
6. **Gio 02/10/2025 10:30-13:30 (3:0 h)** lezione: Gestione dei processi. Definizione di processo. Stati di un processo in un sistema multiprogrammato. Descrittore di un processo (PCB). Code di processi. Cambio di contesto. Creazione e terminazione dei processi. Processi correnti e interazione tra processi. Funzioni del nucleo di un sistema operativo. (MARCO AVVENUTI)
7. **Mar 07/10/2025 08:30-11:30 (3:0 h)** lezione: Scheduling: classificazione. Scheduling: metriche per la valutazione degli algoritmi. Algoritmo First-Come-First-Served (FCFS). Shortest-Job-First (SJF), stima del CPU-burst. Simulazione e discussione comparativa. (MARCO AVVENUTI)
8. **Mer 08/10/2025 13:30-15:30 (2:0 h)** lezione: Algoritmi di scheduling: Shortest-Remaining-Time-First (SRTF). Round-Robin (RR). Simulazione e discussione comparativa di algoritmi di scheduling. Scheduling a code multiple. (MARCO AVVENUTI)
9. **Gio 09/10/2025 10:30-12:30 (2:0 h)** laboratorio: Gestione utenti e gruppi con sistema operativo UNIX-like, modifica della password, permessi di accesso ai file, comando chmod, permessi SUID e GUID. Editor di testo da linea di comando. Esercizi. (MAURIZIO PALMIERI)
10. **Mar 14/10/2025 08:30-11:30 (3:0 h)** lezione: Scheduling a code multiple: multilevel feedback queue. Schedulazione di sistemi in tempo reale: modello. Definizione di un sistema hard real-time, problema della schedulabilità, soluzioni nel caso di processi periodici. Algoritmo Rate Monotonic (RM). (MARCO AVVENUTI)
11. **Mer 15/10/2025 13:30-15:30 (2:0 h)** lezione: Fattore di utilizzazione. Condizioni necessarie e sufficienti per la schedulazione di sistemi periodici. Algoritmo Earliest Deadline First (EDF). Esempi di simulazione. Processi leggeri (thread). (MARCO AVVENUTI)
12. **Gio 16/10/2025 10:30-12:30 (2:0 h)** laboratorio: File di configurazione con informazioni pubbliche, passwd, e sensibili, shadow, degli utenti. Password salvate cifrate con salt e con informazioni temporali sulla durata della password. Gestione dei gruppi di utenti con i comandi adduser deluser, gpasswd e newgroup. Esercizio. (MAURIZIO PALMIERI)

13. **Mar 21/10/2025 08:30-11:30 (3:0 h)** lezione: Tassonomia di Flynn: architetture, uso e metriche. Topologie per le reti di interconnessione. Sincronizzazione dei processi: Tipi di interazione tra i processi. Modelli ad ambiente globale e locale. (MARCO AVVENUTI)
14. **Mer 22/10/2025 13:30-15:30 (2:0 h)** lezione: Problema della mutua esclusione. Soluzioni per il problema della mutua esclusione. Semafori. Implementazione dei semafori. Atomicità delle primitive wait(), signal(). Soluzione al problema della mutua esclusione con semaforo mutex. (MARCO AVVENUTI)
15. **Gio 23/10/2025 10:30-13:30 (3:0 h)** lezione: Semafori in ambiente multiprocessore. Problema della comunicazione. Soluzione al problema della comunicazione con buffer condiviso, produttori-consumatori. Ambiente locale: primitive send, receive. Modelli di programmazione a scambio di messaggi e aspetti implementativi. (MARCO AVVENUTI)
16. **Mar 28/10/2025 08:30-11:30 (3:0 h)** lezione: Il problema lettori-Scrittori. Il problema dei 5 Filosofi. Soluzione ai 5 filosofi basata su semafori. Esempi di deadlock. Grafo allocazione delle risorse (possesso-attesa). Monitor: definizione, variabili condition, soluzione del problema dei 5 Filosofi. (MARCO AVVENUTI)
17. **Mer 29/10/2025 13:30-15:30 (2:0 h)** lezione: Realizzazione di monitor per mezzo di semafori: signal&wait, signal&continue. Monitor come allocatore di risorse. Il problema del deadlock: definizione, condizioni necessarie, resource-Allocation Graph. Metodi per il trattamento del blocco critico: prevenzione statica. (MARCO AVVENUTI)
18. **Gio 30/10/2025 10:30-13:30 (3:0 h)** laboratorio: Gestione dei processi in un sistema operativo Unix-like, Process structure e User structure. System calls per la gestione dei processi: fork, wait e exit. Famiglia di funzioni exec per modificare il codice e i dati di un processo. Esempi di utilizzo della fork ed esercizi. (MAURIZIO PALMIERI)
19. **Mar 04/11/2025 08:30-11:30 (3:0 h)** laboratorio: Metodi per la sincronizzazione tra processi, invio e gestione dei segnali. System calls signal, kill, pause alarm e sleep. Comandi kill e ps per gestire i processi da terminale. Esercizi (MAURIZIO PALMIERI)
20. **Mer 05/11/2025 13:30-15:30 (2:0 h)** laboratorio: Comandi per la gestione dei file in ambienti Unix: il comando find per cercare file e le differenze tra il comando find e il comando locate. Comando grep per cercare stringhe nei file e comando tar per la creazione di archivi. Esercizi. (MAURIZIO PALMIERI)
21. **Gio 06/11/2025 10:30-13:30 (3:0 h)** lezione: Metodi per il trattamento del blocco critico: prevenzione dinamica (deadlock avoidance). Resource-Allocation Graph. Wait-For Graph. Algoritmo del banchiere. (MARCO AVVENUTI)
22. **Mar 11/11/2025 08:30-11:30 (3:0 h)** lezione: Algoritmi di deadlock detection. Tecniche di recovery. Gestione della memoria: rilocazione statica e dinamica, spazio virtuale unico o segmentato, allocazione fisica contigua e non-contigua, caricamento tutto insieme o a domanda. Classificazione delle tecniche di allocazione della memoria. (MARCO AVVENUTI)
23. **Mer 12/11/2025 13:30-15:30 (2:0 h)** laboratorio: Gestione dei processi da terminale e dei job, definizione di process group e niceness dei processi. Intervento sulla niceness con i comandi nice, renice e top. Gestione della chiusura del terminale con i comandi nohup e disown. Esercizi (MAURIZIO PALMIERI)
24. **Gio 13/11/2025 10:30-13:30 (3:0 h)** lezione: Tecniche di gestione della memoria: partizioni fisse, variabili, multiple. Allocazione best-fit e first-fit. Problema della frammentazione. Memoria segmentata: traduzione degli indirizzi virtuali, protezione. Aspetti implementativi, descrittore di segmento, condivisione. Segmentazione a domanda. Segment-fault. Paginazione. (MARCO AVVENUTI)

25. **Mar 18/11/2025 08:30-11:30 (3:0 h)** lezione: Paginazione a domanda. Gestione del page-fault. Algoritmi di rimpiazzamento. Segmentazione con paginazione. Gestione della memoria virtuale in UNIX: pagedaemon, swapper. Gestione delle periferiche (I/O): concetti generali, organizzazione logica del sottosistema di I/O, funzioni del livello device-independent e funzioni del livello device-dependent. (MARCO AVVENUTI)
26. **Mer 19/11/2025 13:30-15:30 (2:0 h)** laboratorio: Libreria Pthread per gestire i threads. Tipo opaco per la gestione dei thread. Sintassi e comportamento di pthread\_create, pthread\_join e pthread\_exit. Gestione della mutua esclusione con i pthread\_mutex, primitive lock e unlock. Esercizi (MAURIZIO PALMIERI)
27. **Gio 20/11/2025 10:30-13:30 (3:0 h)** laboratorio: Condition variables per la sincronizzazione diretta tra threads, controllo ciclico delle condizioni associate, gestione della mutua esclusione. Sintassi e funzionamento delle primitive wait, signal e broadcast. Esempio produttore e consumatore. Esercizi (MAURIZIO PALMIERI)
28. **Mar 25/11/2025 08:30-11:30 (3:0 h)** laboratorio: Gestione dei file nel filesystem, I-node e hard e soft link. Primitive open, read, write e close: sintassi e comportamento. Esempio di utilizzo. Comunicazione tra processi tramite pipe: comportamento di read e write sul pipe. Esercizi (MAURIZIO PALMIERI)
29. **Mer 26/11/2025 13:30-15:30 (2:0 h)** laboratorio: Seminario dal titolo: "Contribuire allo sviluppo open source del Kernel Linux: l'esempio del driver virtio-vsock" tenuto dal dottor Luigi Leonardi, Software engineer di Red Hat. (MAURIZIO PALMIERI)
30. **Gio 27/11/2025 10:30-12:30 (2:0 h)** laboratorio: Redirezione degli standard input e output mediante l'uso della primitiva dup2. Uso combinato di dup2 e due pipe per pilotare un'applicazione con controllo a riga di comando. Esercizio. (MAURIZIO PALMIERI)
31. **Mar 02/12/2025 08:30-11:30 (3:0 h)** lezione: Gestione di un dispositivo: processi esterni, descrittore di un dispositivo, controllo di programma, interruzione. Struttura di un device driver. Flusso di controllo durante un trasferimento. Gestione del temporizzatore. Organizzazione fisica e logica dei dischi. Criteri di scheduling delle richieste di accesso al disco: Algoritmi FCFS, SSTF, SCAN. (MARCO AVVENUTI)
32. **Mer 03/12/2025 13:30-15:30 (2:0 h)** lezione: Organizzazione del file system. La struttura logica del file system. Accesso al file system. Strutture dati e operazioni per l'accesso ai file. (MARCO AVVENUTI)
33. **Gio 04/12/2025 10:30-13:30 (3:0 h)** lezione: Tecniche di allocazione dei file: contigua, a lista concatenata, a indice. File Allocation Table. Valutazioni sul costo di accesso al blocco e sulla scalabilità. UNIX: PCB, filesystem, tabelle per la gestione dei file dei file. Protezione e sicurezza. Modelli, politiche e meccanismi di protezione. (MARCO AVVENUTI)
34. **Gio 11/12/2025 11:30-13:30 (2:0 h)** laboratorio: Seminario dal titolo "Sviluppo di sistemi robotici con ROS", tenuto dagli ingegneri Marco Neri e Angelo Mineo dell'azienda SmartEngineering, sul tema del robotic operating system. (MAURIZIO PALMIERI)
35. **Lun 15/12/2025 14:30-17:30 (3:0 h)** lezione: Protezione e sicurezza. Il modello matrice degli accessi. Graham & Denning: Modifica dello stato di protezione. Realizzazione della matrice degli accessi. Access Control List. Riferimenti a Unix. Realizzazione della matrice degli accessi: Capability List. Sistemi di sicurezza multilivello (Bell-La Padula). Esempi di controllo flusso informativo, Trojan Horse. (MARCO AVVENUTI)

# Introduzione

Dopo i corsi di *Reti Logiche* e di *Calcolatori Elettronici*, il corso di **Sistemi Operativi** rappresenta l'ultimo tassello della formazione sull'aspetto architettonale del calcolatore. Molti argomenti presentano una sostanziale sovrapposizione con Calcolatori Elettronici, fungendo da ponte definitivo tra l'hardware fisico e il software applicativo.

Il corso è attualmente tenuto dai professori **Marco Avvenuti** e **Palmieri**. L'obiettivo centrale non è solo la trasmissione di nozioni tecniche, ma lo sviluppo di una *visione olistica* e critica del calcolatore. La "parola chiave" dell'intero insegnamento è il **sistema operativo multiprogrammato**: lo studente deve acquisire una capacità di astrazione che permetta di comprendere come il sistema gestisca l'esecuzione concorrente e l'accesso alle risorse. Il programma della parte teorica si articola nelle seguenti aree tematiche:

- **Concetti introduttivi:** Evoluzione dai sistemi batch ai sistemi time-sharing, interfacce hardware/software e meccanismi di interruzione.
- **Gestione delle risorse e dei processi:** Definizione di processo e thread, stati di transizione e algoritmi di scheduling della CPU.
- **Sincronizzazione:** Gestione della cooperazione e competizione tra processi, mutua esclusione (semafori e monitor) e risoluzione del *deadlock*.
- **Gestione della memoria:** Memoria virtuale, paginazione, segmentazione e politiche di rimpiazzamento delle pagine.
- **Gestione delle periferiche (I/O):** Architettura del sottosistema di I/O e ottimizzazione dell'accesso ai dischi.
- **Il Filesystem:** Organizzazione logica e fisica, con particolare attenzione alla struttura degli i-node in UNIX.
- **Protezione e sicurezza:** Matrici di controllo degli accessi e modelli di sicurezza multilivello (Bell-La Padula e Biba).

L'esame consiste in una **prova orale**. Il colloquio inizia solitamente con alcuni quesiti scritti o domande relative alle esercitazioni svolte in **laboratorio**. Nella valutazione finale, oltre alla padronanza dei contenuti, viene data enorme importanza alla capacità di utilizzare una **terminologia appropriata** e rigorosa. Il corpo docente valuta con particolare favore la capacità di **ragionamento logico**: è fondamentale dimostrare di saper analizzare un problema in modo critico, poiché la comprensione dei meccanismi profondi del sistema è ritenuta più importante della semplice memorizzazione delle nozioni.

## Struttura del Corso

L'attività didattica è strutturata in due moduli complementari:

- **Teoria dei Sistemi Operativi [60 ore]:** focalizzata sui modelli concettuali, gli algoritmi di gestione e le architetture logiche.
- **Laboratorio [30 ore]:** una parte pratica in ambiente *Unix-like* (Linux) volta a concretizzare la teoria attraverso lo studio della shell, della programmazione multi-thread e delle chiamate di sistema.

## Licenza

Quest'opera è distribuita con licenza Creative Commons "Attribuzione – Non commerciale – Condividi allo stesso modo 3.0 Italia".



# **Parte I**

## **Concetti introduttivi**

## Definizione di sistema operativo

La definizione più generale di **sistema operativo** è un *componente software di un sistema di elaborazione* che ha come compito principale quello di controllare l'esecuzione dei programmi applicativi e di agire come intermediario tra questi programmi e la **macchina fisica (hardware del computer)**. Andando quindi nel dettaglio, possiamo vedere un sistema operativo come quella componente software del nostro sistema che permette di eseguire i programmi e che semplifica la risoluzione dei problemi degli utenti, rendendo quindi il sistema più - globalmente parlando - facile da usare. Potremmo quindi rappresentare, in modo del tutto schematico, un sistema di elaborazione come una struttura a tre livelli, organizzati gerarchicamente.

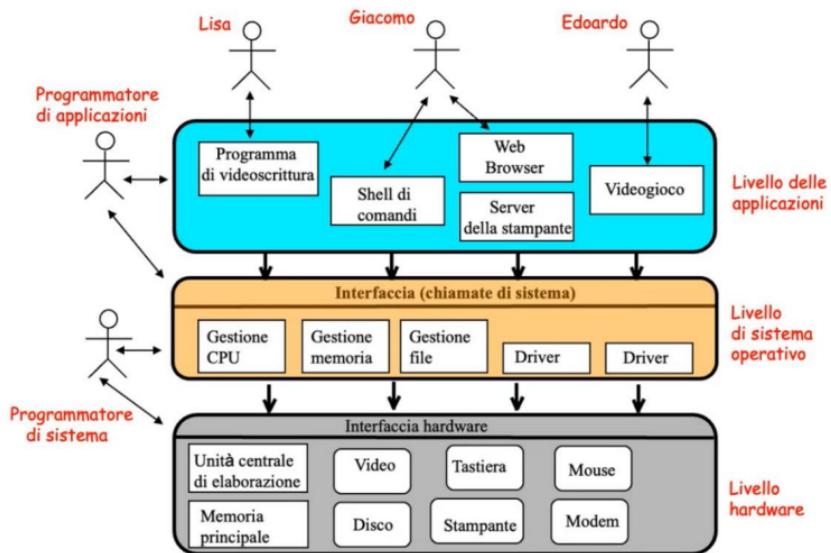


Figura 1.1: Organizzazione gerarchica di un sistema di elaborazione

Il livello hardware, sul quale gira anche il sistema operativo, è quello che corrisponde ai componenti fisici del sistema e comprende il **processore**, la **memoria principale**, e le **unità periferiche** (come **tastiera**, **video**, etc). Lo scopo di questo livello è quello di mettere a disposizione dei programmi le risorse necessarie per la loro esecuzione, attraverso delle interfacce (gerarchicamente visibili solo al sistema operativo). Immediatamente sopra a questo livello troviamo il livello del **sistema operativo**, il quale corrisponde a un insieme di componenti software che hanno il compito di gestire le **risorse fisiche del sistema** offrendo ai programmi applicativi un'interfaccia **standard** più semplice da utilizzare rispetto a quella messa a disposizione dal **livello hardware**. Tale interfaccia è composta da tutta una serie di funzioni che i vari programmi utente possono utilizzare per interfacciarsi con le risorse fisiche in modo agile e semplice, garantendo inoltre tutta una serie di

altre caratteristiche. L'ultimo livello è quello dei programmi applicativi, che corrisponde al livello delle applicazioni utilizzare direttamente dagli utenti del sistema di elaborazione. Un altro modo di vedere il sistema operativo è quello di immaginarlo come un insieme di programmi che astraggono i dettagli dell'hardware, offrendo una macchina virtuale più semplice e uniforme per i programmi applicativi. Questa astrazione semplifica lo sviluppo dei programmi e favorisce la portabilità tra sistemi con hardware diverso.

Inoltre, il fatto che l'accesso alle risorse fisiche avvenga sempre attraverso il sistema operativo permette di avere una maggiore efficienza e ottimizzazione dell'utilizzo di tali risorse. A tal proposito distinguiamo due elementi diversi di un sistema di elaborazione:

- I **meccanismi** sono delle porzioni di software **stabili** e con un funzionamento **invariato** che permettono di compiere determinate funzioni del nostro sistema.
- Le **politiche** sono delle *scelte* del sistema operativo che riguardano le modalità di utilizzo dei meccanismi.

Il sistema operativo, quindi, attraverso delle scelte (**politiche**) è in grado di gestire l'allocazione e l'utilizzo delle risorse fisiche in modo ottimizzato e, soprattutto, garantendo la sicurezza e la correttezza dei dati del sistema. Un esempio di tale differenza riguarda lo **scheduling dei processi**: lo **scheduling** dei processi sfrutta il **meccanismo del cambio di contesto** (introdotto a *calcolatori elettronici*) per cambiare il processo, ma è attraverso le **politiche** del sistema operativo che viene deciso quali processi dovranno essere scambiati. Ovviamente, le risorse che servono ad un processo per poter essere eseguito sono la **CPU**, che tuttavia può solo vedere una istruzione per volta e quindi richiede che le istruzioni siano caricate insieme ai dati in **memoria principale**. Ci possono essere anche ulteriori risorse fisiche, come ad esempio le periferiche per l'**I/O**, che però non sono fondamentali e/o obbligatorie. Un altro tipo di risorse sono le **risorse logiche**, alle quali fanno capo tutte quelle variabili o strutture dati condivise necessarie per il funzionamento del sistema (nel corso di calcolatori avevamo strutture come la **idt** o la variabile **esecuzione**) e, soprattutto, i **file**, i quali permettono di memorizzare dati in modo permanente all'interno della **memoria di massa**.

## 1 Gestione delle risorse e sicurezza

Una delle funzioni principali del sistema operativo è quella di **gestione delle risorse**. I moderni sistemi di elaborazione sono in grado di eseguire più programmi contemporaneamente, anche di più utenti diversi; questa caratteristica prende il nome di **multiprogrammazione**. Tuttavia la presenza di questo meccanismo causa non pochi problemi all'interno di un sistema. Prendiamo come esempio la figura 1.1

- Lisa cerca di salvare un file di testo in memoria permanente.
- Edoardo cerca di salvare i progressi di gioco in memoria permanente.

Entrambi gli utenti, contemporaneamente, cercano di accedere ad una medesima risorsa: la **memoria**. Una situazione come questa, se non gestita correttamente, potrebbe portare ad avere dati in memoria inconsistenti (con risultati potenzialmente **catastrofici**). L'idea è che quindi il sistema operativo si occupi di gestire questi conflitti coordinando l'accesso alla risorsa da parte dei vari programmi in modo che le due operazioni siano eseguite in modo sequenziale (attraverso delle **politiche**). Lo stesso ragionamento può essere fatto anche per tutte le risorse; ad esempio, se più programmi devono eseguire delle operazioni e hanno bisogno della **CPU**, è compito del sistema operativo decidere chi la utilizza per primo e per quanto, andando a fare quello che in gergo viene detto **scheduling dei processi**.

Un sistema operativo deve anche garantire che vengano rispettate **sicurezza** e **protezione** delle informazioni da possibili agenti malevoli esterni al sistema di elaborazione. Inoltre, oltre alla protezione da agenti esterni il sistema operativo si occupa anche della gestione di eventuali guasti interni al sistema.

## 2 Application Programming Interface

Riassumendo quanto detto fino ad ora, le principali funzioni di un sistema operativo sono:

- **Facilitare lo sviluppo e la portabilità dei programmi applicativi.** Il programmatore non dovrà quindi conoscere, nel dettaglio, le componenti di un determinato sistema, ma è sufficiente che conosca le sue interfacce.
- **Realizzare politiche di gestione delle risorse del sistema di elaborazione.** Ad esempio, nello scheduling vengono implementate **politiche** che decidono come assegnare la CPU la cui conoscenza da parte del programmatore non è necessaria.
- Fornire meccanismi di protezione e garantire la sicurezza del sistema ai guasti. I modelli di protezione riguardano il controllo degli accessi alle risorse del sistema: quando un programma richiede di accedere a una risorsa, la richiesta viene intercettata dal sistema operativo che verifica se il programma ha o meno il diritto di eseguire quell'azione.

All'inizio avevamo accennato alla presenza di alcune interfacce utilizzate dai programmi utente e disponibili al programmatore che mettono a disposizione funzionalità di interfacciamento con le risorse fisiche del sistema; queste interfacce prendono il nome di **API (Application Programming Interface)**. Queste **API** mettono quindi a disposizione le primitive necessarie al programmatore per interfacciarsi con la macchina anche senza possedere una conoscenza sull'architettura sottostante; queste primitive sono dette anche *chiamate di sistema*.

# 2

## Cenni storici sui sistemi operativi

Prima del 1950 circa le i calcolatori erano macchine **enormi** e **enormemente costose**, utilizzate soprattutto in ambito industriale e scientifico. Rispetto a quello che vediamo oggi, la struttura era ben diversa, le uniche periferiche di ingresso erano costruite dai lettori di schede perforate e i perforatori di schede. Non era inoltre previsto alcun sistema operativo e gli utenti interagivano direttamente con le **risorse fisiche del sistema**. L'utente, in particolare, si interfacciava direttamente con una console sulla quale era possibile impostare dei valori in **codice binario** all'interno della memoria e dei registri della **CPU**. In questo modo era possibile, modificando opportunamente il reigstro **IP**, decidere quale programma mettere in esecuzione impostando direttamente il valore binario della locazione della prima istruzione del programma. Un tipico programma passava quindi da diverse fasi, una di queste era la compilazione:

- Nella prima fase il programma veniva scritto su delle schede perforate, dove ogni scheda conteneva un'istruzione.
- Veniva caricato nella memoria del calcolatore un programma **compilatore**.
- Una volta attivato, il *compilatore*, produceva il programma tradotto perforando il risultato della traduzione su altre schede.

Una volta ottenuto il programma compilato si poneva su un lettore di schede, si caricava in memoria il programma **caricatore** per trasferire in memoria il programma tradotto e, infine, ne veniva attivata l'esecuzione. Pisa per un certo periodo di tempo rappresentò uno dei centri italiani dell'informatica grazie alla sua macchina **CEP**



Figura 2.1: Calcolatrice Elettronica Pisana

Un procedimento come questo risulta estremamente tedioso, con una componente di automazione praticamente pari a 0 (l'utente deve interagire con la macchina costantemente). Gli unici programmi presenti su questi sistemi di elaborazione erano costituiti dai componenti del **sistema di programmazione** e - in un periodo successivo - anche da librerie, etc.

## 1 I primi sistemi batch

Per far fronte ai problemi di efficienza tipici dei vecchi e monolitici calcolatori nacquero, tra gli anni 50 e 60, i primi **sistemi operativi**. Il primo passo verso l'introduzione di questi componenti

1. Venne introdotta una memoria di massa, costituita da **nastri magnetici**, che permetteva di rendere disponibili i programmi componenti.
2. L'utente programmatore forniva il suo codice ad un operatore, che era anche l'unico abilitato a poter operare sul calcolatore. Per abilitare l'utente a specificare quali programmi di sistemi avesse bisogno fu definito un linguaggio di controllo, detto anche *job control language*. Il programmatore presentava all'operatore il proprio pacco di schede contenente alcune schede relative al programma da eseguire e alcune schede, dette **schede di controllo**, caratterizzate dall'avere un primo carattere particolare.
3. Venne definito un nuovo programma di sistema, detto *monitor*, il quale era residente in una porzione della memoria a esso riservata. Lo scopo del **monitor** era quello di, ciclicamente, leggere le schede e, per ogni scheda di controllo, ne interpretava il contenuto. Una volta interpretato il contenuto il monitor si occupava di prelevare il programma sistema dalla memoria permanente per poi caricarlo in memoria e trasferire il controllo della cpu al programma sistema con un istruzione di salto. Al termine del programma sistema il monitor riotteneva il controllo della CPU e proseguiva nella sua esecuzione.

Un'ulteriore ottimizzazione adottata fu quella di permettere l'esecuzione di *job* di utenti diversi senza necessità di intervento dell'operatore per cambiare il pacco di schede. Quindi, l'unico intervento dell'operatore si aveva al termine dell'esecuzione del *batch* di *job*. Tuttavia, a fronte dell'incremento dell'efficienza di uso delle risorse della macchina si creavano due problematiche

- L'utente veniva "allontanato" dalla macchina. Il programmatore che portava il proprio pacco di schede doveva aspettare ore e/o giorni per avere il proprio risultato, con una diminuzione dell'efficienza del *debugging* del codice in caso di errore.
- I programmi venivano eseguiti sequenzialmente. Poteva quindi accadere che un utente che aveva presentato un programma molto semplice si trovasse ad aspettare l'esecuzione di un programma molto complesso solo perché il suo pacco di schede era stato messo sotto a quello del programma più complesso.

Nonostante l'efficienza nell'utilizzo delle risorse fosse aumentata, era comunque molto bassa. Questa problematica derivava dalla notevole differenza di velocità tra la CPU e i dispositivi periferici. Fu solo con l'avvento dei dischi, del meccanismo delle interruzioni e del meccanismo del DMA che fu possibile apportare dei miglioramenti significativi.

## 2 Sistemi di *Spooling*

La tecnica dello **spooling** prevede che un tutti i **job** di un **batch** venissero caricati su un disco. Facendo in questo modo la CPU, per caricare e leggere i dati, doveva solo interagire con tale disco. Mentre il calcolatore eseguiva i programmi di un **batch**, in parallelo avvenivano, grazie al DMA, due cose

- Venivano caricati sul disco i **job** del **batch** successivo.
- Venivano stampati i risultati dei **job** dei **batch** precedenti.

Grazie alla tecnica dello **spooling** si ottengono due vantaggi principali. Si diminuiscono i tempi di attesa della CPU, in quanto si trovava a operare soltanto con periferiche *veloci*. Avendo sul disco l'insieme dei programmi da eseguire era possibile implementare delle politiche di scheduling scegliendo, di volta in volta, quale programma mettere in esecuzione.

### 3 Sistemi multiprogrammati

Un'ulteriore miglioramento nelle performance del sistema fu dato dall'introduzione della multiprogrammazione, basata sul meccanismo delle interruzioni. L'idea alla base di questo meccanismo è quella di ottimizzare l'utilizzo della CPU in ogni momento. Supponiamo infatti di avere tre programmi caricati, chiamati  $P_1$ ,  $P_2$  e  $P_3$ :

- Mentre  $P_1$  è in esecuzione, ad un certo istante  $t_i$ , esegue un'istruzione di **I/O**.
- Il sistema operativo, a questo punto, altera il registro **IP** e pone l'indirizzo della prima istruzione del secondo programma.

Quindi, nel momento in cui  $P_1$  non ha più bisogno della CPU poiché è in attesa dell'operazione di **I/O**, il sistema operativo fa subentrare il programma  $P_2$  invece di lasciare la CPU in attesa. L'idea è che quindi tutti i programmi vengano fatti evolvere *"contemporaneamente"*, anche se in ogni istante solo uno di essi è in reale esecuzione. Possiamo osservare anche graficamente il vantaggio della multiprogrammazione

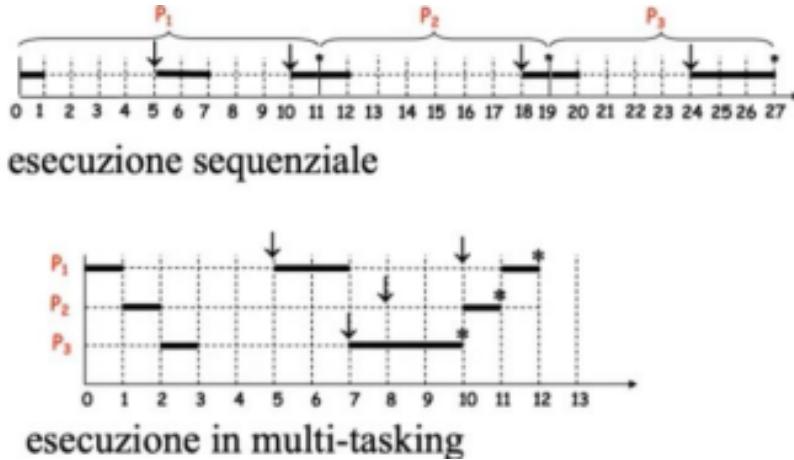


Figura 2.2: Differenza tra sistemi a batch e sistemi multiprogrammati

Notiamo immediatamente come sia i *tempi di risposta* del programma, che il *turn-around* vengono migliorati.

**Programmi I/O bound e CPU bound** Una delle problematiche della multiprogrammazione è legata al fatto che non è sempre possibile caricare tutti i **job** di un **batch** in memoria e quindi il sistema deve sceglierne solo un sottoinsieme da caricare in memoria. A tal proposito può essere utile creare una distinzione tra due tipologie di programmi

- Programmi **I/O bound**: programmi che richiedono molti trasferimenti dati e con poche istruzioni da eseguire.
- Programmi **CPU bound**: programmi che richiedono pochi trasferimenti, ma con molte istruzioni da eseguire.

Affinché sia possibile ottimizzare l'uso di tutte le risorse della macchina è ideale scegliere un mix di programmi delle due tipologie.

**Problema dell'overhead** La multiprogrammazione non è esente da problematiche. La problematica principale prende il nome di *overhead*: *agffinché il sistema sia in grado di eseguire più programmi secondo il modello della multiprogrammazione, deve essere in grado di eseguire del codice aggiuntivo*. Ad esempio, il codice relativo ai cambi di contesto o i codici relativi allo scheduling.

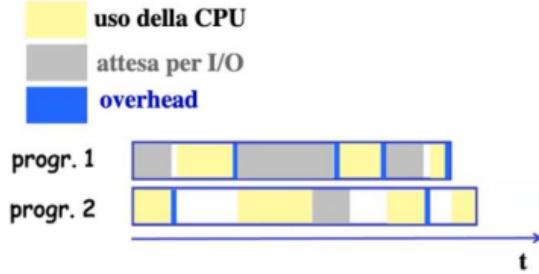


Figura 2.3: Rappresentazione grafica dell'overhead

La virtualizzazione introduce quindi la necessità di eseguire delle operazioni aggiuntive, che tuttavia toglie tempo all'esecuzione dei programmi e quindi occorre cercare di contenere questo tempo in una percentuale minima. Se non è possibile garantire questo limite sull'overhead non avrebbe proprio senso implementare il meccanismo della multiprogrammazione.

## 4 Sistemi *time-sharing*

Una ulteriore ottimizzazione dei sistemi multiprogrammati è quella dei **sistemi *time-sharing***. Il criterio che fu adottato fu quello di eseguire vari programmi assegnando loro una porzione di tempo di **CPU** (*time-slice*) per poter eseguire le proprie operazioni. Se però allo scadere del tempo il programma era ancora in esecuzione, la CPU veniva commutata verso un altro programma mediante un cambio di contesto. Tali sistemi sono in grado di revocare la cpu ad un processo (*preemption*) anche quando il processo non ha terminato il suo utilizzo della CPU (*cpu-burst*).

## 5 Sistemi in tempo reale

I sistemi in **tempo reale** sono dei sistemi particolari. L'idea è quindi che il calcolatore controlli l'evoluzione di un sistema esterno, che può essere di diversa natura, attraverso **sensori** e **attuatori**.



Figura 2.4: Struttura di un sistema in tempo reale

Il sistema di calcolo, mediante opportuni sensori, legge periodicamente il valore delle grandezze fisiche da controllare, esegue un programma (*task*) che stabilisce se tale valore rientra nell'ambito dei valori corretti oppure se si verifica uno scostamento rispetto ai valori corretti. Nel secondo caso il sistema di calcolo retroagisce sull'uscita cercando di riportare la grandezza nei valori corretti. Ciascuna di queste task costituisce un programma ciclico che viene eseguito periodicamente. I

sistemi in tempo reale sono caratterizzati dal fatto che ogni task deve completare ciascuna delle proprie esecuzioni entro un intervallo temporale imposto dall'applicazione. Inoltre, possiamo suddividere i sistemi in tempo reale in due categorie:

- Sistemi *hard-real-time*: dedicati ad applicazioni di controllo critiche, dove la violazione di anche solo una delle **deadline** può compromettere il corretto funzionamento del sistema.
- Sistemi *soft-real-time*.

## Architettura di Von-Neuman

L'architettura di riferimento che utilizzeremo durante il corso è quella dell'architettura di **Von Neuman**. L'architettura è composta da un unico bus, quindi un'insieme di fili, su cui viaggiano e si propagano le informazioni e - come già visto in altri corsi - si distingue in una parte relativa agli **indirizzi** e una parte relativa ai **dati**.

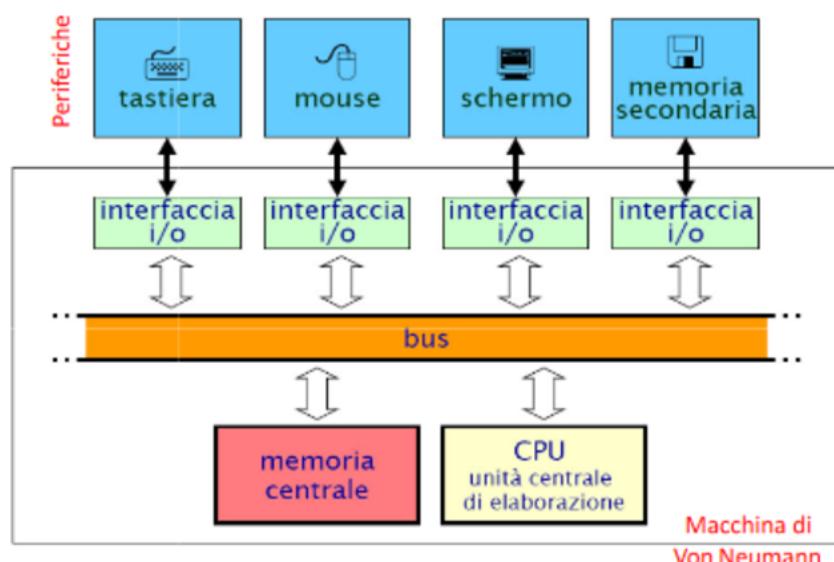


Figura 3.1: Rappresentazione schematica dell'architettura di Von Neuman

Oltre alla divisione del bus nelle sue diverse sezioni, è anche importante ricordare che il bus è **bidirezionale** e vari dispositivi **master** possono iniziare delle transazioni verso altri dispositivi **slave**. Dobbiamo quindi definire un **protocollo** per l'utilizzo del **bus**, così da evitare il formarsi di **corse concorrenti** e conseguenti problemi elettrici e di **inconsistenza dei dati**. Inoltre, il processore è connesso al bus attraverso due registri, un registro **MAR** in cui viene inserito l'indirizzo di memoria da cui si intende leggere e un registro **MDR** dove invece viene scritto il risultato dell'operazione di lettura.

## 1 Processore

Il componente centrale di tutta l'architettura è il processore (CPU o **Central Processing Unit**), il cui compito è eseguire, sequenzialmente, le istruzioni di un programma che si trova contenuto nella memoria centrale. L'elemento centrale del processore è costituito dall'insieme dei **registri**, all'interno del quale possiamo distinguere: **registri programmabili**, **registri di stato** e **registri**

**generali.** Insieme ai **registri** è di fondamentale importanza anche l'insieme delle **istruzioni** che permettono di dire alla CPU cosa deve fare e, a tal proposito possiamo distinguere due tipologie di processori

- **valuta se cambiare con tabella.**
- **Processori RISC:** Presentano un set di istruzioni ridotto. Il vantaggio di questa tipologia di processore è la riduzione del tempo di esecuzione delle istruzioni, grazie al formato delle istruzioni, che sono tutte della stessa lunghezza e dello stesso formato.
- **Processori CISC:** Presentano un set di istruzioni complesso. Il vantaggio di questa tipologia di processore è la maggior semplicità di programmazione e la presenza di un set molto ampio e specializzato, con moltissime istruzioni diverse.

L'insieme di tre registri particolari (**Program Counter**, **Instruction Register** e **Program Status Word**) costituisce quella che in gergo viene chiamata **unità di controllo**. Il primo registro si occupa di memorizzare l'indirizzo della prossima istruzione da eseguire, il secondo registro contiene l'istruzione che deve essere eseguita e il terzo è un registro di stato che contiene informazioni. L'unità di controllo gestisce le fasi che caratterizzano il **ciclo macchina**:

- **Prelievo** dell'istruzione.
- **Decodifica** dell'istruzione.
- **Esecuzione** dell'istruzione.

L'ultima fase del **ciclo macchina** viene gestita da un'ulteriore componente della CPU chiamata **Arithmetic Logic Unit**. Dal corso di reti logiche abbiamo visto che la **ALU** non è altro che un **sommatore** che, sfruttando delle reti combinatorie riesce a elaborare i dati trasformando ogni operazione in una operazione di somma.

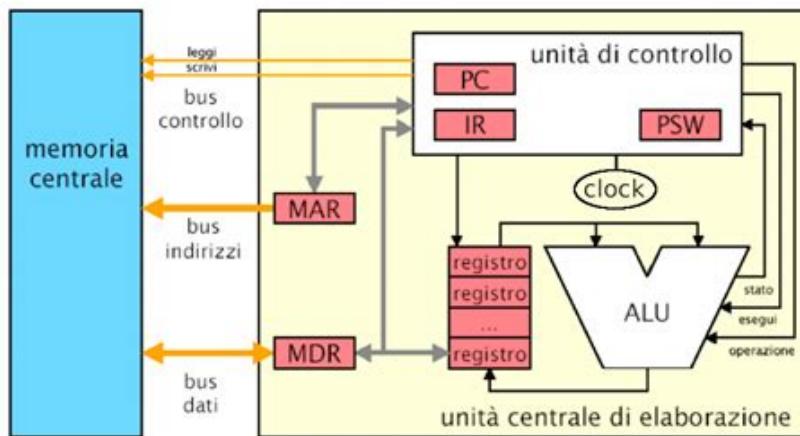


Figura 3.2: Architettura interna del processore

Un altro pezzettino importante della nostra architettura è lo **stack**, una serie di registro operativi in cui vengono alloggiati gli operatori utili ai fini delle istruzioni che sono attualmente in esecuzione. Poiché il contenuto dei registri, talvolta, andrà propagato in memoria sarà necessario predisporre un collegamento tra lo **stack** e il registro **MDR**.

## 2 Memoria

Uno dei problemi principali legato all'architettura di Von Neumann è l'interfacciamento CPU-Memoria. Infatti la CPU è una componente velocissima, in grado di compiere milioni di operazioni ogni secondo, che però si trova ad interfacciarsi con la memoria che è invece molto più lenta,

portando a perdere preziosi cicli macchina. L'architettura della memoria è suddivisibile su tre tipologie

- Memoria centrale (RAM o **Random Access Memory**): La memoria centrale è la memoria che contiene tutti i programmi attualmente in esecuzione (contiene quindi le istruzioni) e i dati di tali programmi. Dal nostro punto di vista possiamo vedere questa memoria come un **array** dove, specificando un indice, è possibile accedere **direttamente** a una qualsiasi delle locazioni disponibili.
- Memoria a sola lettura (ROM o **Read Only Memory**): Una parte della memoria centrale è costituita dalla memoria ROM, una memoria persistente il cui scopo è quello di contenere il programma che deve essere eseguito all'avvio della macchina, detto anche programma di **bootstrap**. Per semplicità supporremo che la ROM non sia in alcun modo scrivibile, anche se nella realtà esistono tipologie di ROM (EPROM e EEPROM) che sono invece scrivibili.
- Memoria persistente (**Disco**): La memoria secondaria su cui sono memorizzati, in modo persistente (ma scrivibile) i programmi e i dati che non sono attualmente in uso. Il vantaggio della persistenza crea tuttavia un problema relativo ad un maggior consumo di energia e ad una minore velocità di accesso; questi due fatti rendono quindi questa memoria inadatta a adempiere al ruolo di memoria centrale.

Nel modello visto a calcolatori abbiamo la presenza anche di una memoria ad accesso veloce, chiamata **cache**, che permette di ovviare (in parte) al problema della differenza di velocità tra memoria e CPU. Questa memoria sfrutta dei principi di località (**località spaziale** e **località temporale**) per memorizzare alcune porzioni di memoria che suppone possano essere utili alla CPU nell'esecuzione e che quindi la CPU potrebbe voler accedere spesso. Nonostante la cache introduca tutta una serie di vantaggi, dall'altra parte porta anche alcune problematiche nei sistemi multiprogrammati: la presenza di memoria cache tende ad aumentare percentualmente l'**overhead** nel **cambio di contesto**. Infatti quando la CPU viene commutata da un programma a un altro, il sistema operativo deve anche occuparsi di **invalidare** la cache. Nelle architetture che lo prevedono esisteranno due memorie cache:

- **Instruction cache (I-Cache)**.
- **Data cache (D-Cache)**.

Il vantaggio di questa distinzione deriva dal fatto che nel momento in cui il contenuto della I-Cache viene variata non è necessario che la modifica debba essere propagata in memoria. Esistono anche cache di diverso livello; in particolare distinguiamo tre livelli: nel primo livello la cache è implementata all'interno del circuito della CPU, gli altri due non si trovano nel circuito della CPU e, quindi, risultano leggermente più lente.

### 3 Periferiche

I dispositivi di ingresso/uscita costituiscono i componenti del sistema che permettono il trasferimento dati da e verso l'esterno del nostro sistema. Per questo motivo tali dispositivi sono talvolta indicati anche con **dispositivi periferici**. Ognuno di questi dispositivi, come anche rappresentato in figura 3.1, è collegato al bus attraverso delle interfacce di ingresso/uscita, ognuna contenente dei registri indirizzati nello spazio di indirizzamento del nostro calcolatore. Questo fatto ci dà l'indicazione che l'interazione con una periferica non avvenga quindi direttamente con la CPU, ma attraverso un supporto intermedio che semplifica enormemente la gestione. Eseguendo apposite istruzioni il processore è in grado di operare sui registri del controllore di un dispositivo e attivare quindi il dispositivo per effettuare un trasferimento dati.

### 4 Interruzioni

Il processore che abbiamo visto fino ad adesso esegue un flusso di istruzioni dettato da un unico programma: ogni istruzione del programma ordina al processore le istruzioni da eseguire e l'ope-

razione successiva. Quindi, ogni **routine** del programma va in esecuzione solo perché l'istruzione precedente nel flusso sequenziale ha trasferito il flusso di controllo dalla **routine** in questione.

Quello che vogliamo fare noi è introdurre un meccanismo che, dato un insieme finito di **eventi**  $E = \{e_1, e_2, \dots, e_n\}$ , associa ad ognuno di questi elementi una routine da un'insieme finito di routine  $R = \{r_1, r_2, \dots, r_n\}$ . Nello specifico, vogliamo creare un'associazione del tipo

$$\text{Interruzione} : E \rightarrow R$$

che associa ogni evento  $e_i$ , una routine  $r_i$ ,  $\forall i = 1, \dots, n$ . Implementando questo meccanismo il programmatore è in grado di scrivere un programma, ma può anche scrivere delle routine  $r_i$  da associare a degli eventi  $e_i$ . In questo modo, se durante l'esecuzione di p uno di questi eventi avviene, il processore interromperà p ed eseguirà il codice della routine associata a quell'evento, per poi restituire il controllo a p. Il preconcetto su cui costruiamo questo meccanismo è che l'interruzione sia momentanea e che alla sua fine il flusso di controllo torni a programma p precedentemente in esecuzione.

Supponiamo di avere una stampante all'interno del quale troviamo un registro **TBR** e un registro **STS**. All'interno del registro **STS** troviamo un **FLAG**, chiamato **READY**: questo **FLAG** segnala alla **CPU** se la stampante ha finito di stampare il carattere presente all'interno del registro **TBR**. Il nostro programma dovrebbe quindi essere organizzato così: quando la CPU ha finito di calcolare un valore di  $f(x)$  prepara la riga da stampare all'interno del buffer di memoria e aspetta che la stampante sia pronta e poi scrive il primo carattere. A questo punto, invece di aspettare che la stampante sia pronta a ricevere il prossimo carattere, prosegue con il calcolo del secondo valore. Possiamo notare fin da subito una problematica non indifferente: la routine che esegue i calcoli deve controllare, di tanto in tanto, il registro **STR**, in modo da poter scrivere il prossimo carattere dentro **TBR** quando il **FLAG** di **READY** va ad 1, quello che però dobbiamo valutare è *quanto spesso fare il controllo*. Se il controllo viene fatto troppo spesso si rallenta l'esecuzione del programma, se il controllo viene fatto troppo raramente si rallenta la stampa e, tuttavia, anche se troviamo il corretto posizionamento, non possiamo essere ancora sicuri che sia quello ottimale. Ad esempio: supponiamo di avere il seguente ciclo e di dover controllare il flag **READY** di **STS** (supponiamo che la funzione per farlo sia `check_ready()`).

```
// check_ready() -- Posizionarlo qua fa un controllo tropo raramente.
for(int i = 0; i < 10000000; ++i){
    // check_ready() -- Posizionarlo qua fa un controllo tropo frequentemente.
    v[i]++;
}
// check_ready() -- Posizionarlo qua fa un controllo tropo raramente.
```

Inserendo il controllo di **STS** fuori dal ciclo si rischia di attendere troppo per inserire il carattere dentro **TBR**; inserendo il controllo all'interno del ciclo, invece, si rischia di perdere del tempo a controllare troppo di frequente il **flag**. Inoltre, se la funzione che esegue i calcoli non è stata scritta da noi siamo anche costretti a modificarla per poter applicare questo meccanismo di interruzioni e, come ben sappiamo, modificare una funzione da una libreria rischia di causare problemi di leggibilità e compatibilità del codice.

Per evitare le problematiche dovute al dover necessariamente fare un controllo attivo sulla stampante e, soprattutto, per evitare di perdere dei preziosi cicli di clock, è stato introdotto il meccanismo delle **interruzioni**. L'idea è che, mentre la stampante sta lavorando la CPU possa eseguire altre istruzioni senza preoccuparsi di fare alcun controllo e, nel momento in cui la stampante ha terminato ed è pronta a stampare il valore successivo segnali alla CPU di scrivere nel buffer di trasmissione. Questa segnalazione viene chiamata interruzione, poiché la CPU interrompe il suo flusso di esecuzione e gestisce la richiesta della stampante, eseguendo una routine che scrive in **TBR**. Una volta gestita la richiesta riprende il flusso di esecuzione che aveva in precedenza. Ovviamente non è obbligatorio che la CPU gestisca queste richieste, talvolta infatti potrebbe essere necessario vietare che, per qualche motivo, il flusso di esecuzione della CPU venga interrotto: si pensi solo al caso in cui si interrompa una scrittura in memoria per effettuarne un'altra, se le due scritture scrivessero nella stessa area di memoria si verificherebbero problematiche legate all'inconsistenza dei dati. Per questo motivo il programmatore ha a disposizione due comandi che permettono di abilitare o disabilitare il flag delle istruzioni chiamati, rispettivamente, **sti** e **cli**.

## **Parte II**

# **Struttura dei sistemi operativi**

# 4

## Gestione dei processi

Prima di entrare nel vivo della trattazione sulla gestione dei processi è opportuno fare una breve digressione sulla struttura dei sistemi operativi. L'organizzazione dei sistemi operativi che seguiranno durante questo corso segue una struttura **modulare**.

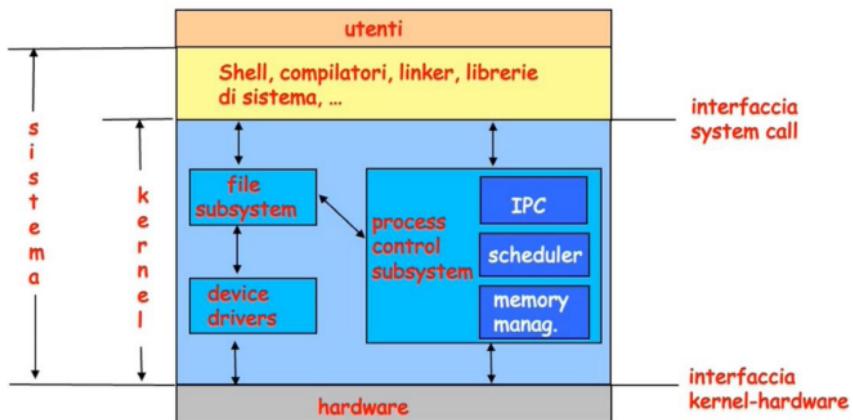


Figura 4.1: Struttura modulare del sistema operativo unix

In questa struttura immaginiamo di suddividere il sistema operativo in livelli distinti, ognuno con una struttura interna mascherata agli altri livelli e con delle interfacce. Queste interfacce hanno il solo scopo di esporre una serie di servizi ai livelli immediatamente successivi. I vantaggi di un'organizzazione di questo tipo sono molteplici e sono state già discusse negli scorsi capitoli, possiamo però elencarle nuovamente

- Semplicità nella programmazione del sistema.
- Maggiore sicurezza e protezione del sistema.

Nel caso dei sistemi **unix** fanno parte del sistema sia le tipiche componenti di un sistema operativo, invocate globalmente tramite le chiamate di sistema, eseguite in stato privilegiato e identificate globalmente dal termine **kernel**. In particolare, all'interno del kernel troviamo

- **File subsystem**.
- **Device Drivers**.
- **Process Control Subsystem**.

Le prime due strutture si occupano rispettivamente della gestione dei file e dell'interfacciamento con i dispositivi di I/O. Di tutte queste strutture la più importante è sicuramente il **PCS**, il quale, a sua volta, è suddiviso in moduli diversi:

- **Inter-Process Communication.** Permette la comunicazione tra processi.
- **Scheduler.** Permette di gestire la **schedulazione**, quindi il cambio di processo.
- **Memory manager.**

Unendo il livello kernel insieme a tutti quei servizi relativi a shell, compilatori, linker, etc. si ottiene il **livello di sistema**. Questo livello è quello che, di fatto, si interpone tra utenti e hardware e che permette l'interfacciamento con il sistema.

## 1 Processi

Quindi, all'interno di un sistema multiprogrammato più **processi** possono essere caricati nella memoria principale ed eseguiti in modo concorrente. Un processo può essere definito come **un programma in esecuzione**; in particolare:

- Rappresenta la sequenza di eventi generati dall'elaboratore durante l'esecuzione di un programma.
- Identifica l'unità di esecuzione all'interno di un sistema operativo multiprogrammato.

La CPU, in un sistema multiprogrammato, passa da un processo a un altro a una velocità tale da dare all'utente l'impressione che i vari programmi vengano eseguiti parallelamente. Di fatto però, il termine "parallelamente" è solo relativo al punto di vista dell'utente, ma la CPU non esegue nulla in parallelo; chiamiamo questo fatto *parallelismo virtuale*. Il parallelismo effettivo, detto *parallelismo fisico*, è possibile solo in presenza di un sistema dotato di molte CPU.

Occorre tuttavia prestare attenzione al fatto che un programma può dare luogo a istanze diverse di un processo, quindi **esecuzioni del medesimo processo con dati in ingresso diversi**. Affinché tutta questa struttura sia possibile è necessario creare una struttura dati in grado di memorizzare - o, comunque - tenere conto della posizione di tutti i dati che riguardano il processo. Dotiamo quindi ogni processo di una struttura dati chiamata **descrittore di processo**. Lo scopo di questa struttura è quindi quella di portare il processo, che è sostanzialmente qualcosa di astratto, a un piano fisico, relativo all'elaboratore. L'idea è che nel descrittore debbano essere contenute alcune informazioni tali da permettere lo *swap-in* di un processo interrotto senza che ci sia perdita o inconsistenza dei dati che stava utilizzando; in particolare troviamo:

- **Codice:** un puntatore a un file che contiene il codice del processo.
- **Dati:** i dati dipendono dal programma, è quindi necessario che il codice eseguibile sia definito in modo tale da dare un'idea chiara delle variabili e delle strutture dati di cui il programma ha bisogno. Verosimilmente anche qui ci sarà un puntatore alla memoria principale, memorizzare tutti i dati in una struttura sarebbe estremamente difficile
- **Program Counter, Registri e Stack:** per poter descrivere un processo che non è eseguito è necessario memorizzare, da qualche parte, tutti i registri di tale processo.

Un programma necessita anche di altre risorse. Lo spazio di memoria, in parte descritto dai dati, è una di queste; il programma potrebbe infatti chiedere di creare delle nuove strutture dati (e.t. *new*) e in questo caso è necessario avere spazio della memoria che sia disponibile e pronto all'uso. A tale scopo dobbiamo quindi inserire nel descrittore di processo un puntatore a questo spazio di memoria. Dovrà esserci anche un elenco di file aperti, quindi un file per il quale il programma ha richiesto l'accesso per fare operazioni di scrittura/lettura. Infine, è opportuno anche sapere quali dispositivi di I/O il programma sta utilizzando.

Tutte queste considerazioni ci portano a una conclusione abbastanza "filosofica". Idealmente, ogni processo è attivo fintato che sta utilizzando la CPU o sta facendo operazioni di I/O e, sempre idealmente, un processo non ha idea del fatto che un altro processo sta utilizzando la sua stessa CPU per fare delle operazioni. Potremmo quindi immaginare che ogni processo sia dotato di una propria **CPU virtuale**.

**Spazio virtuale di memoria** Lo spazio virtuale di ogni processo ha un’organizzazione ben definita, che tiene conto di tutti gli aspetti che abbiamo discusso nell’ambito del **descrittore di processo**. In primo luogo troviamo sicuramente una sezione `.text`, all’interno della quale risiede il codice della funzione; insieme ad essa, immediatamente sopra, troviamo le strutture dati del codice, contenute in un area di memoria detta `.data`. Immediatamente sopra a queste due sezioni abbiamo due aree di memoria fondamentali per il funzionamento del processo

- **Heap.** Permette l’allocazione dinamica di strutture dati durante il flusso di esecuzione del codice.
- **Stack.** Permette l’allocazione delle strutture dati che vengono utilizzate dalle varie funzioni del codice.

Entrambe queste sezioni non hanno uno spazio predefinito, possono crescere liberamente all’interno dell’area di memoria di ogni processo, che a tal proposito avrà un quantitativo di spazio vuoto.

## 2 Stato di un processo

Un processo, durante la sua evoluzione, è sottoposto a continue **transizioni di stato** definite da una parte dell’attività del processo stesso e dall’altra da eventi asincroni che possono avvenire durante la sua esecuzione. Con **eventi asincroni** intendiamo eventi relativi ad altri processi presenti nel sistema come, ad esempio, la fine dell’operazione di **I/O**.

In un sistema **monoprogrammato**, in cui possiamo avere un solo processo nel sistema, abbiamo l’alternanza di *cpu-burst* e di *io-burst*. Un processo è quindi da ritenersi, in ogni istante, in uno tra due possibili stati

- **Esecuzione:** corrisponde alla fase di *cpu-burst*.
- **Bloccato.**

Il passaggio dallo stato di bloccato allo stato di esecuzione è detto **riattivazione**, mentre il passaggio inverso, dallo stato di esecuzione a quello di bloccato, è detto **sospensione**. Lo stato di bloccato può avere diverse origini, il processo potrebbe essere in attesa del verificarsi di un qualche evento, come un’interruzione da parte di un dispositivo (la riattivazione del processo sarà quindi possibile solo a seguito di un’interruzione dello stesso dispositivo); un processo potrebbe essere bloccato anche per una decisione **autonoma** del **sistema operativo**, ad esempio quando una sua istruzione genera un’eccezione. Ovviamente, quando il processo viene sospeso la CPU inizierà a bruciare cicli di clock senza fare nulla (*idle*). In un sistema monoprogrammato un processo non può eseguire solo il suo codice, ma deve includere anche il codice del sistema operativo. Quindi nel suo spazio di memoria virtuale si dovrà trovare anche tutte le strutture dati necessarie al funzionamento del sistema operativo.

Nei sistemi multiprogrammati abbiamo un numero di processi attivi che è **sempre maggiore del numero delle CPU disponibili**. Nel nostro caso particolare supponiamo di avere un sistema **multiprogrammato** ma con un singolo processore (**monoprocesso**). In questa situazione, di tutti i processi attivi, uno solo può utilizzare l’unità di elaborazione ad ogni istante. Pertanto avere un solo stato **attivo** non risulta più sufficientemente esaustivo, poiché possiamo avere processi che vorrebbero essere eseguiti, ma non possono poiché un altro processo sta utilizzando la CPU; dobbiamo quindi suddividere lo stato di attivo in due ulteriori stati:

- **Esecuzione.** Il processo è pronto ad essere eseguito, diciamo che è dentro il *cpu burst*, e ha a disposizione la CPU.
- **Pronto.** Il processo è pronto quando è dentro il *cpu-burst*.

Il passaggio di stato tra **pronto** e **esecuzione** viene semplicemente detto **assegnazione della CPU**. La transizione tra **esecuzione** e **pronto** è detta *preemption*, quest’ultima non è sempre disponibile e viene eseguita in caso si verifichino delle situazioni particolari. Il passaggio tra **esecuzione** e **bloccato** è detto **sospensione** ed è del tutto analogo a quello del sistema multiprogrammato. Infine, il passaggio tra **bloccato** e **pronto** è detto **riattivazione**, anch’esso è analogo

a quello del sistema multiprogrammato. Andando a unire tutto ciò che abbiamo visto fino ad ora possiamo concludere che in un sistema multiprogrammato il grafo degli stati di un processo ha una struttura basata su 5 nodi

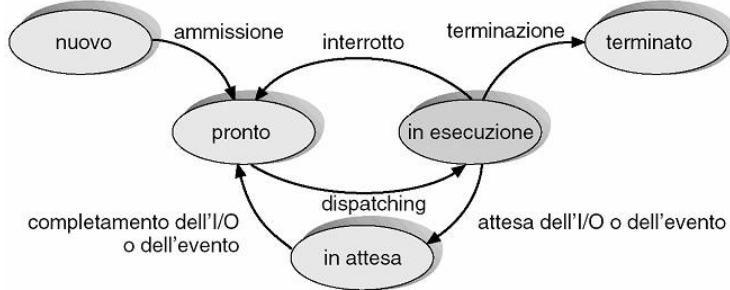


Figura 4.2: Grafo degli stati di un sistema operativo. Oltre alla suddivisione dello stato di attivo troviamo anche due nuovi stati: **terminato** e **nuovo**. Il primo indica un processo appena creato, l'altro indica che un processo ha terminato la propria esecuzione

La particolarità ed il grande vantaggio dei sistemi multi-programmati è che quando uno degli  $n$  processi è bloccato la CPU sarà comunque occupata (*busy*) ad eseguire uno degli  $n - 1$  processi che non sono bloccati o comunque una delle routine del sistema operativo.

**Scheduler** L'assegnazione della CPU è una delle fasi principali dell'esecuzione di un processo e viene gestita da una parte del sistema operativo detta **scheduler**. Questa componente sfrutta delle **politiche** per decidere, tra un certo numero di processi **pronti**, l'assegnazione ottimale del processore in base a delle **metriche** di efficienza che riguardano i processi. Lo *scheduler* entra in esecuzione quando il processo in esecuzione **cambia stato** o quando il processo **termina**. In alcuni sistemi è anche possibile prevedere che un processo vada in esecuzione prima di un altro, in questo caso lo *scheduler* entra in esecuzione anche quando un nuovo processo entra in coda pronti.

### 3 Struttura del descrittore di processo e code di processi

Come abbiamo detto, ad ogni processo viene associata ad una struttura dati, chiamata **descrittore di processo** (o *control process block*). Un descrittore di processo è organizzato come una struttura a campi, al cui interno troviamo

- **Nome del processo.** La scelta più comune è quella di identificare ogni processo con un indice numerico. Quando i processi sono organizzati in gruppi è utile specificare anche il blocco di appartenenza.
- **Stato del processo.** Uno degli stati in cui il processo può trovarsi.
- **Modalità di servizio dei processi.** La scelta a quale processo pronto assegnare la CPU può essere effettuata secondo diversi criteri. Associamo quindi ad ogni processo una priorità, fissa o varabile, che indica la sua importanza, relativamente al criterio scelto, rispetto agli altri processi.
- **Informazioni sulla gestione della memoria.** Contiene le informazioni necessarie a individuare l'area di memoria principale nella quale sono caricati il codice e i dati del processo.
- **Contesto del processo.** Tutte le informazioni contenute nei registri di macchina all'atto della sospensione dell'esecuzione di un processo vengono salvate nel suo descrittore. Tali informazioni riguardano il contatore di programma e un insieme dei registri (variabili a seconda dell'architettura).

- **Utilizzo delle risorse.** Lista dei dispositivi di I/O assegnati al processo, i file aperti, tempo di uso della CPU, etc.
- **Identificazione del processo successivo.** In base al loro stato i processi vengono inseriti in delle code. Questo identificatore indica il processo successivo nella particolare coda.

In un sistema multiprogrammato, vista la presenza di un numero di processi maggiore del numero delle CPU, ogni processo deve essere dotato di un proprio descrittore; dobbiamo quindi introdurre una tabella, detta **tabella dei processi**, che contiene al suo interno tutti i **PCB**.

I processi sono organizzate in code in funzione del loro stato. I processi che sono pronti e nell'attesa di essere eseguiti sono organizzati in una o più code, dette *code dei processi pronti*. Insieme alle *code dei processi pronti* abbiamo delle code che riguardano anche i processi bloccati, chiamate in generale *code dei processi bloccati*. Tra di queste ne possono figurare alcune relative all'accesso alle periferiche del sistema, come le *code per l'accesso al disco*. Ognuna di queste code ha un descrittore di coda, che contiene l'indice del **primo processo** in coda e l'indice dell'**ultimo processo** in coda. In alcuni sistemi sono disponibili anche delle *code dei descrittori disponibili*; la creazione di un nuovo processo corrisponde all'estrazione di un elemento da questa coda e il trasferimento in una delle *code dei processi pronti*.

## 4 Cambio di contesto

Il cambio del processo in esecuzione con un altro scelto alla coda pronti comporta l'esecuzione di un insieme di operazioni che vanno sotto al nome di **cambio di contesto**. Tali operazioni sono:

- Salvataggio del contesto del processo in esecuzione nel suo descrittore.
- Inserimento del descrittore nella coda dei processi bloccati o dei processi pronti.
- Selezione di un altro processo tra quelli contenuti nella coda dei processi pronti e caricamento del nome di tale processo nel registro processo in esecuzione.
- Caricamento del contesto del nuovo processo nei registri del processore.

La prima e l'ultima operazioni vanno sotto al nome di **salvataggio dello stato e caricamento dello stato**. Il cambio di contesto comporta l'aggiornamento di tutte le strutture dati che rappresentano le risorse utilizzate dai processi: memoria, dispositivi di I/O, file aperti, etc. Schematicamente possiamo rappresentare il cambio di contesto nel seguente modo

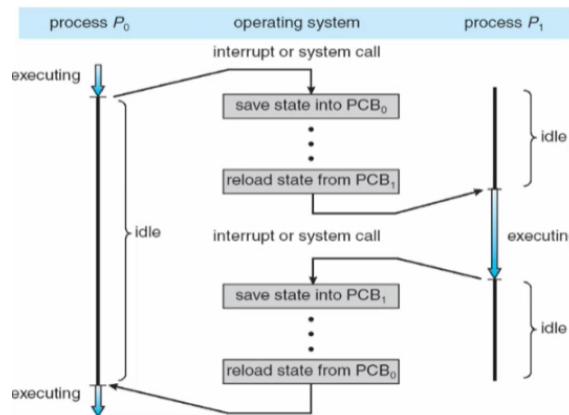


Figura 4.3: Rappresentazione dell'operazione di cambio di contesto

Nel caso di un sistema con cache, insieme a risorse e dati, occorre aggiornare anche la cache. Per farlo si procede facendo prima, se necessario, un *write-back* in memoria centrale e, successivamente, invalidandola. Tutte queste operazioni possono causare un **overhead** considerevole.

## 5 Creazione e terminazione di processi

In alcuni sistemi il numero di processi rimane costante e definito a priori. In generale tuttavia un processo è in grado di dare origine ad altri processi attraverso delle primitive di sistema. In una situazione di questo tipo il processo creante è detto **processo padre**, mentre, il processo creato è detto **processo figlio**. Un **processo figlio** può, a sua volta, diventare un processo padre dando origine ad altri processi.

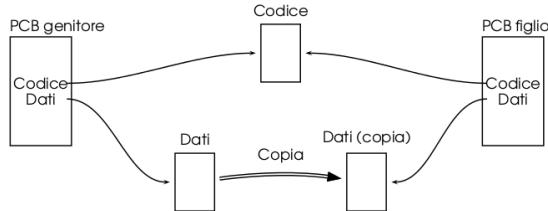


Figura 4.4: Creazione di un processo **figlio**

In un sistema in cui sono presenti molti processi, indipendentemente dal fatto che siano legati da una qualche forma di parentela, deve gestire tutta una serie di problematiche legate all'interazione di questi processi. In particolare, possiamo fare una distinzione tra due situazioni tipiche dei sistemi multiprogrammati

- Se nel sistema vi è una sola unità di elaborazione diciamo che i processi sono in uno stato di **interleaving**.
- Se nel sistema vi è un numero di unità di elaborazione maggiore o uguale al numero di processi diciamo che questi ultimi sono in uno stato di **overlapping**.

Ovviamente nel caso di overlapping si avranno prestazioni migliori, a fronte però di un costo di implementazione e della tecnologia maggiore. Una definizione fondamentale, che trascende le due situazioni appena elencate, è quella di processi concorrenti; in particolare

Due processi si dicono concorrenti quando la prima operazione di uno inizia prima dell'ultima operazione dell'altro.

Processi concorrenti possono essere **indipendenti** o **interagenti**. Il primo caso si verifica quando due o più processi non si influenzano reciprocamente, quindi quando non condividono dati e non si scambiano informazioni. Il secondo caso si verifica, invece, quando uno o più processi si influenzano reciprocamente. Questo può avvenire sia mediante scambio di informazioni e/o di messaggio temporali, sia tramite competizione per una risorsa.

# 5

## Schedulazione dei processi

La schedulazione dei processi viene svolto da una delle componenti poste all'interno del kernel, detta **scheduler**. Questa componente ha il compito di allocare, secondo delle opportune **politiche**, la CPU e la memoria ai vari processi che si trovano nello stato di **pronto**. In generale, in un sistema operativo troviamo tre livelli di scheduling:

- **short-term scheduling**: Lo scheduling a breve termine è quello che si lega al meccanismo del cambio di contesto ed è quello a cui faremo riferimento durante il corso. L'idea di questo scheduling è che il sistema operativo allochi o deallochi la CPU ai vari processi secondo delle politiche (*algoritmo di scheduling*) definite a priori. Una prima classificazione tra questi algoritmi di scheduling è relativa alla scelta di quali siano gli eventi in seguito ai quali lo scheduler può intervenire.
  - **Scheduling preemptive**. In questa tipologia di algoritmi lo scheduler può intervenire in qualsiasi momento, anche se un processo non vuole lasciare la CPU.
  - **Scheduling non preemptive**. In questa tipologia di algoritmi lo scheduler può intervenire soltanto quando il processo lascia spontaneamente la CPU.

La seconda tipologia di algoritmi è sicuramente più veloce, poiché viene ridotto il numero di cambi di contesto e quindi l'overhead sul sistema. D'altra parte però vi è un grosso problema in termini di flessibilità e anche sicurezza.

- **medium-term scheduling**: Lo scheduling a medio termine rappresenta invece la funzione del sistema operativo che si occupa di trasferire temporaneamente processi dalla memoria centrale a quella di massa e viceversa. Tale funzione permette quindi di liberare spazio a processi che si trovano già in coda pronti o che devono essere caricati.
- **long-term scheduling**: lo scheduling a lungo termine è tipico dei sistemi operativi multiprogrammati. Questa tipologia di scheduling ha due compiti. Il primo scopo è quello di scegliere nella memoria secondaria quali programmi caricare in memoria centrale, cercando un giusto equilibrio tra programmi **CPU-bound** e **I/O-bound** in modo che non crei uno svantaggioso sbilanciamento tra i due. Il secondo compito è quello di gestire il grado di multiprogrammazione. Se non vi fosse una gestione attiva del grado di multiprogrammazione si correrebbe il rischio di avere troppi processi attivi e continui cambi di contesto, con conseguente incremento dell'overhead.

Durante il corso studieremo principalmente lo scheduling a **lungo** e **medio** termine. Lo scheduling a breve termine agisce quindi direttamente sulla coda pronti e sulla CPU, prelevando processi dalla coda e mettendoli in esecuzione. Se un processo si sospende, rilasciando la CPU, finisce in una coda dei processi sospesi; nella realtà questa coda è scomponibile in una moltitudine di code (una per ogni evento). Alcune tecniche di virtualizzazione della memoria permettono anche di revocare la CPU ad un processo in esecuzione o in coda pronti per farne lo **swap-out**. Se un processo si

trovava in esecuzione durante lo **swap-out** verrà messo in una **coda dei processi parzialmente eseguiti** e dovrà attendere lo **swap-in** dei suoi dati nella memoria centrale prima di tornare in coda pronti. Ad oggi questa pratica è poco utilizzata visto l'enorme aumento di overhead che comporta; si preferisce fare **swap-out** della memoria dei processi sospesi.

## 1 Algoritmi di scheduling a breve termine

In generale però possiamo classificare gli algoritmi in base a due criteri: **implementazione della preemption** e **implementazione di un sistema prioritario**. Possiamo quindi classificare i sistemi operativi in quattro categorie:

- Algoritmi **senza preemption e senza priorità**: FCFS.
- Algoritmi **con preemption e senza priorità**: RR.
- Algoritmi **senza preemption e con priorità**: SJF.
- Algoritmi **con preemption e con priorità**: SRTF

Altri algoritmi che studieremo sono estremamente particolari e introducono concetti avanzati; in particolare: **schedulazione su base prioritaria**, **schedulazione a code multiple** e **schedulazione di sistemi in tempo reale**.

Per poter classificare gli algoritmi di schedulazione dobbiamo definire delle metriche che permettano di dare una caratterizzazione ai diversi algoritmi. In generale le metriche sono:

- **Utilizzazione della CPU**. Indica la frazione di tempo in cui la CPU è occupata nell'eseguire processi, ed è definita come il rapporto tra la somma degli intervalli di **CPU-burst** e il tempo totale di osservazione:

$$\text{Utilizzazione CPU} = \frac{\sum_{i=1}^n \Delta B_i}{T}$$

dove  $\Delta B_i$  è la durata del singolo burst di CPU e  $T$  è il tempo totale considerato.

- **Tempo medio di completamento** (o *turnaround medio*). Si definisce come tempo medio di completamento il tempo che un generico utente deve mediamente attendere per avere i risultati finali una volta che il suo programma è stato sottomesso al sistema per l'esecuzione. Matematicamente lo possiamo esprimere come la media degli intervalli temporali che intercorrono tra quando i singoli processi entrano per la prima volta in coda pronti e quando gli stessi terminano le loro esecuzioni. Detto  $T_i$  il tempo di completamento del processo *i-esimo* e detto  $n$  il numero di processi:

$$\text{Tempo medio di completamento} = \frac{\sum_{i=1}^n T_i}{n}$$

L'obiettivo, in questo caso, è quello di minimizzare il tempo medio di completamento.

- **Produttività**. Espressa come numero medio di processi completati nell'unità di tempo. Matematicamente corrisponde all'inverso del *tempo medio di completamento*.
- **Tempo di risposta**. Definito come il tempo intercorso tra l'istante in cui il processo entra in coda pronti e l'istante in cui fornisce la prima risposta.
- **Tempo di attesa**. Definito come il tempo che un processo aspetta nella coda pronti in attesa di avere a disposizione la CPU. Quindi, supponendo di avere  $n$  intervalli con ampiezza  $A_i$ , esprimiamo il tempo medio di attesa come

$$\text{Tempo medio di attesa} = \frac{\sum_{i=1}^n A_i}{n}$$

Anche in questo caso l'obiettivo è quello di minimizzare il tempo medio di attesa.

## 1.1 First-Come First-Served

Questo algoritmo è sicuramente l'algoritmo di scheduling più semplice. L'algoritmo non implementa il meccanismo di *preemption* e quindi il processo rilascia la CPU solo spontaneamente. L'idea dell'algoritmo è immediata e deducibile dal nome: **il processo che per primo è arrivato in coda pronti è il primo che viene eseguito**. In pratica, la coda pronti, è organizzata con una politica **FIFO**, dove i processi entrano in fondo alla coda ed escono dalla testa di essa. Supponiamo di avere la seguente situazione:

Processo	Istante di arrivo	Durata del CPU burst
$P_0$	0	10
$P_1$	2	100
$P_2$	4	24
$P_3$	6	16

Tabella 5.1: Tabella dei processi e delle loro durate CPU

Il primo processo a entrare in coda pronti sarà sicuramente  $P_0$ , che sarà quindi anche il primo ad essere eseguito. Siccome **FCFS** è un algoritmo che non implementa la preemption, allora il processo  $P_0$  starà in esecuzione per 10 secondi di tempo. Successivamente sarà il processo  $P_1$ , entrato per secondo nella coda pronti, a essere eseguito dall'istante 10 fino all'istante 110. Possiamo applicare lo stesso ragionamento anche per gli altri due processi. Rappresentando su un diagramma quanto detto si ottiene

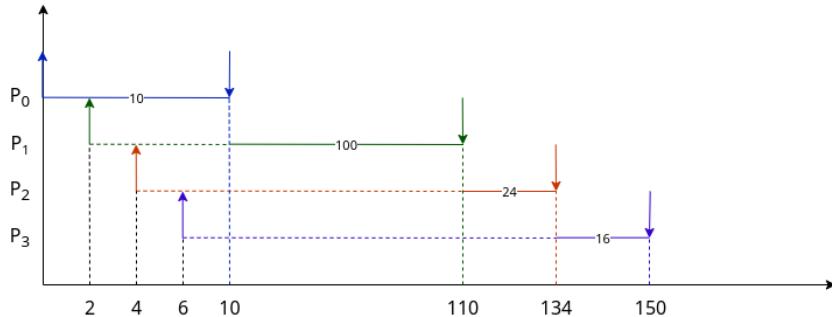


Figura 5.1: Esempio schedulazione FCFS

Il **turnaround time** (o **tempo di completamento**) del processo  $P_0$  sarà pari a 10 poiché entra in coda pronti e viene immediatamente eseguito. Mentre per il processo  $P_1$  questo turnaround time sarà pari a 108, quindi più di dieci volte più grande del processo  $P_0$ . Per il processo  $P_2$  e  $P_3$  questo tempo medio di attesa sarà pari, rispettivamente, a 130 e 144. Analizzando l'immagine ci si rende conto che in nessun istante di tempo la CPU è stata in *idle* e quindi l'efficienza dell'utilizzo (**utilizzazione della CPU**) della CPU è pari ad 1. Un'altra metrica che andiamo a calcolare è quella del **turnaround medio**, definito come la media tra i **turnaround time** di ogni processo

$$\frac{10 + 108 + 130 + 144}{4} = 98$$

osservando il turnaround medio ci si rende conto che i processi penalizzati sono quelli di più breve durata. Possiamo calcolare anche l'attesa media, calcolata come la media dei tempi di attesa di ogni processo

$$\frac{0 + 8 + 106 + 128}{4} = 60.5$$

Anche in questo caso si dimostra come la penalizzazione sia rivolta ai processi più piccoli. Quindi, FCFS, è un algoritmo che una forte componente **casuale**: la sua efficienza dipende dall'ordine con cui i processi entrano all'interno del sistema.

## 1.2 Shortest-Job First

Anche questo algoritmo, analogamente al primo, non implementa la **preemption** e, inoltre, funziona nell'ipotesi in cui sia noto il **CPU burst** richiesto da un processo quando lo stesso entra in coda pronti. Disponendo di tale informazione, l'algoritmo assegna la CPU al processo che richiede il minor tempo di esecuzione fra tutti quelli presenti nella coda dei processi pronti e, nel caso in cui due processi abbiano lo stesso **CPU burst**, applica la regola dell'**FCFS**. Supponiamo di avere la seguente situazione:

Processo	Istante di arrivo	Durata del CPU burst	Priorità
$P_0$	0	10	0
$P_1$	2	100	3
$P_2$	4	24	2
$P_3$	6	16	1

Tabella 5.2: Tabella dei processi e delle loro durate CPU. Rispetto al precedente algoritmo FCFS, in questo caso, introduciamo anche una priorità **statica** dove 0 indica la massima priorità

All'istante 0 l'unico processo nella coda pronta è  $P_0$  e quindi sarà anche il primo ad essere eseguito (indipendentemente dalla priorità). All'istante  $t = 10$ , quando termina, tutti gli altri processi sono entrati in coda pronta e, di conseguenza, è effettivamente possibile fare una schedulazione **SJF**. Successivamente a  $P_0$  andrà in esecuzione  $P_3$  essendo il processo a priorità più alta nel sistema (valore più basso implica priorità maggiore). Una volta terminato  $P_3$  all'istante  $t = 26$  andrà in esecuzione  $P_2$  e, infine,  $P_1$ .

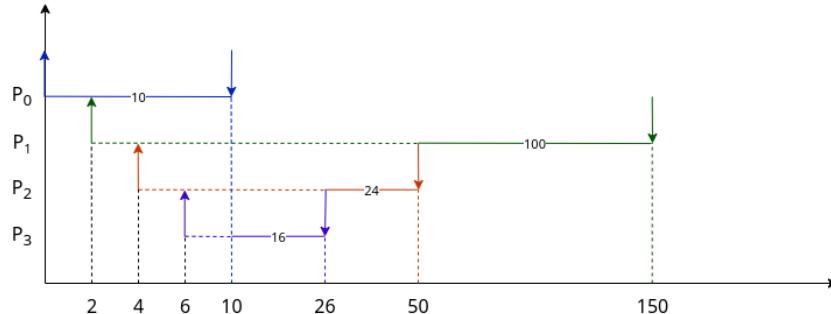


Figura 5.2: Schedulazione **SJF**

Nella realtà però ci rendiamo conto che la priorità non è propriamente statica. Infatti all'istante  $t = 2$  nel sistema ci sarà  $P_0$  in esecuzione e  $P_1$  in coda pronta, quindi  $P_1$  avrà priorità pari ad 1. Quando però, all'istante  $t = 4$ , arriverà nel sistema anche  $P_2$  la priorità di  $P_1$  dovrà essere cambiata vista la maggiore durata del suo **CPU burst**.

Anche in questo caso possiamo misurare turnaround medio e tempo di attesa medio

$$\begin{aligned} \text{Turnaround medio} &= \frac{10 + 24 + 44 + 144}{4} = 56 \\ \text{Tempo medio di attesa} &= \frac{0 + 8 + 22 + 44}{4} = 18.5 \end{aligned}$$

Si dimostra quindi facilmente che questo algoritmo minimizza il tempo medio di attesa dei processi. D'altra parte però non è detto che si conoscano tutti i **CPU-burst** dei vari processi e, soprattutto, la mancanza della preemption non permette di bloccare un processo nel caso se, in coda pronta, arriva un processo con maggiore priorità

Il problema della non conoscenza dei tempi di **CPU-burst** può essere risolto mediante delle stime statiche del **CPU-burst**. Questo valore può essere stimato come media esponenziale dei

valori misurati nei precedenti intervalli di esecuzione, facendo in modo che una misurazione pesi tanto meno tanto quanto più è vecchia

$$s_{n+1} = at_n + (1 - a)s_n$$

dove  $t_n$  indica la durata del CPU burst precedente,  $a$  un fattore compreso tra 0 e 1,  $s_n$  indica la stima al passo precedente. Andando a tarare  $a$  si decide l'importanza di  $t_n$  rispetto a  $s_n$  e viceversa.

### 1.3 Short-Remaining-Time First

Questo algoritmo è del tutto analogo all'algoritmo **SJF**, con l'unica aggiunta del meccanismo della **preemption**. L'idea è che venga eseguito il processo con il **CPU-burst** più basso fintanto che non entra in coda pronto un processo con un **CPU-burst** minore. Supponiamo di avere la seguente situazione:

Processo	Istante di arrivo	Durata del CPU burst
$P_0$	6	10
$P_1$	0	100
$P_2$	4	24
$P_3$	2	16

Tabella 5.3: Tabella dei processi e delle loro durate CPU

All'istante  $t = 0$  l'unico processo presente nel sistema è  $P_1$  e, di conseguenza, lo scheduler gli assegnerà l'utilizzo della CPU. Una volta giunti all'istante  $t = 2$  entrerà nel sistema anche il processo  $P_3$ . A questo punto lo scheduler farà un confronto tra i due **CPU-burst** e, siccome  $98 > 16$ , lo scheduler toglierà dall'esecuzione il processo  $P_1$  e metterà in esecuzione il processo  $P_3$ . All'istante  $t = 4$  entra in coda pronto anche il processo  $P_2$ , che però ha un **CPU-burst** maggiore di  $P_3$  e, quindi, rimane in coda pronta in attesa. Infine, all'istante  $t = 6$  entra in esecuzione anche il processo  $P_0$ , il quale ha invece un **CPU-burst** minore di quello rimanente di  $P_3$ . Il sistema farà quindi **preemption** su  $P_3$ , mettendo in esecuzione  $P_0$ .

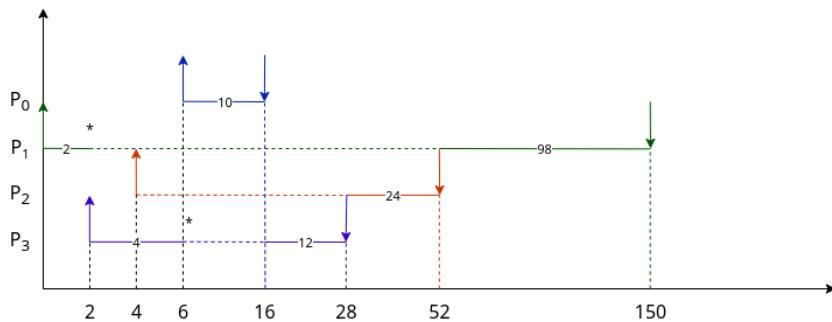


Figura 5.3: Esempio di schedulazione SRTF

Andando a provare a fare lo stesso esempio con l'algoritmo di **SJF** ci si rende immediatamente conto che sia in termini di **tempo medio di attesa** che di **tempo medio di completamento** l'algoritmo **SRTF** è sicuramente più efficiente. Tuttavia, l'introduzione della **preemption** comporta sicuramente un aumento dell'overhead.

**Problemi nella gestione mediante priorità** Una delle problematiche legate alla presenza del criterio di priorità negli algoritmi di scheduling è il problema dello **starvation**. Supponiamo che un processo abbia **CPU-burst**  $x$  molto grande; in un algoritmo con priorità il processo  $P_x$  sarà

sicuramente eseguito dopo tutti i processi  $P_y$  tali per cui  $x > y$ . Il problema è che se iniziassero ad arrivare infiniti processi con priorità  $y < x$  il processo  $P_x$  non verrebbe mai eseguito entrando, quindi, in starvation.

## 1.4 Round-Robin

L'algoritmo **RR** è stato concepito appositamente per sistemi a **partizione di tempo**. In questo tipo di sistema ogni processo è fondamentalmente costituito da un susseguirsi di intervalli di **CPU-burst** e intervalli di attesa. Durante l'intervallo di attesa il processo è sospeso in attesa che l'utente digiti un comando al sistema; questo comando specifica il tipo di programma che il processo eseguirà nel successivo intervallo di **CPU-burst**. L'idea dell'algoritmo è proprio quella di minimizzare il tempo di risposta ad ogni singolo comando e, quindi, il tempo di attesa per ogni processo. Per farlo sfrutta un'idea molto semplice: ad ogni processo viene assegnato lo stesso tempo di **CPU-burst**. Inoltre, non implementando un meccanismo di priorità la coda dei processi pronti è gestita mediante una coda di tipo **FIFO**. Supponiamo di avere la seguente situazione:

Processo	Istante di arrivo	Durata del CPU burst
$P_0$	0	100
$P_1$	0	50
$P_2$	0	10
$P_3$	0	10

Tabella 5.4: Tabella dei processi e delle loro durate CPU

All'istante  $t = 0$  tutti i processi si trovano in coda pronti, ma siccome la gestione di quest'ultima è **FIFO**, allora, il primo processo ad essere eseguito sarà sicuramente  $P_0$ . Supponendo di avere un **quanto** di tempo pari a  $\Delta = 20$  succede che, all'istante  $t = 20$ , il timer manderà un'interruzione esterna. Il processo  $P_0$  non ha sicuramente terminato il suo **CPU-burst** e viene quindi rimesso in coda pronti. Successivamente viene eseguito  $P_1$  e, analogamente a  $P_0$ , dopo  $\Delta$  viene rimesso in coda pronti. Successivamente entra in esecuzione  $P_2$  che però termina prima di  $\Delta$  e, quindi, non viene rimesso in coda pronti. Stessa cosa avviene per  $P_3$ . Successivamente lo schedulatore riparte mettendo in esecuzione  $P_0$ .

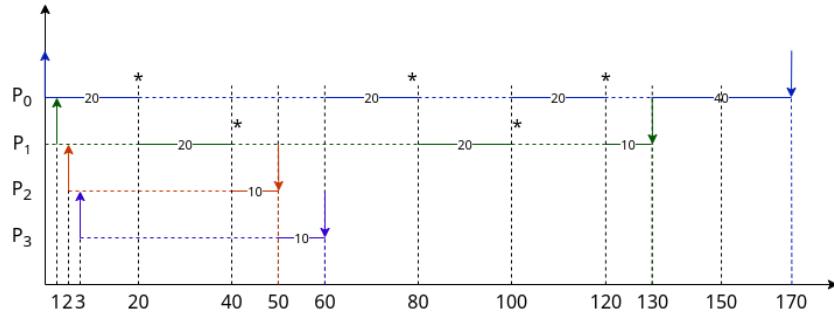


Figura 5.4: Esempio di schedulazione mediante **Round-Robin**

Visto il grande numero di scambi, per non aumentare troppo l'overhead di sistema, è opportuno che il tempo C per eseguire il cambio di contesto sia molto minore rispetto al quanto di tempo  $Q$ . In questo modo, se  $n$  è il numero di processi, ogni processo riceve  $Q$  unità di tempo ogni  $n(Q + C)$  unità di tempo reale.

## 2 Schedulazione a code multiple

In sistemi operativi particolarmente complessi possono coesistere nel sistemi processi con caratteristiche molto diverse tra di loro. Si potrebbe avere sistemi in cui coesistono processi **I/O-bound** e **CPU-bound**, così come anche processi che lavorano in *background* e processi cosiddetti *foreground*. In questi sistemi l'idea più efficiente è quella di organizzare i processi in **code di processi** in funzione delle loro caratteristiche. Ad ogni **coda di processi** associamo una priorità e lo scheduler sceglierà un processo da mandare in esecuzione da una coda solo nel momento in cui le code di livello superiore sono vuote.

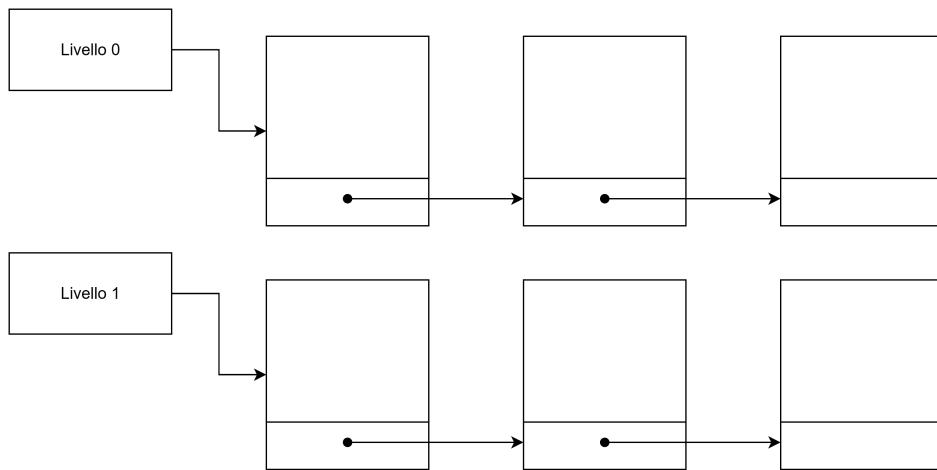


Figura 5.5: Schedulazione a code multiple

La gestione di una moltitudine di code è estremamente complessa; per questo motivo si utilizza generalmente il meccanismo del **multilevel feedback queue**. Le code di livello 0 e 1 sono gestite con **Round-Robin** con un quanto di tempo rispettivamente di  $\Delta_0$  e  $\Delta_1$ , dove vale  $\Delta_1 > \Delta_0$ . Quando un processo entra in coda pronti viene immediatamente messo nella coda a priorità più alta (**livello 0**). Se quando viene messo in esecuzione non termina il suo **CPU-burst** entro il quanto di tempo  $\Delta_0$ , allora, una volta tolto dall'esecuzione, viene messo nella coda di livello immediatamente inferiore (**livello 1**). In questo livello avviene una cosa del tutto analoga: se il processo non termina la sua esecuzione entro  $\Delta_1$ , una volta tolto dall'esecuzione, viene posto nella coda di livello ancora più bassa (**livello 2**). La coda di livello 2 viene gestita con una politica **FCFS**.

Uno dei problemi che si può creare si verifica quando, una volta svuotate le due code a priorità più alta entra in esecuzione un processo non interattivo della coda di livello 2. Il problema, nello specifico, è che se va in esecuzione un processo con un **CPU-burst** molto lungo, siccome la coda di livello due è gestita con **FCFS**, esso rimarrà in esecuzione fintanto che non termina o si sospende; in altre parole, il sistema è **non-responsive** fintanto che il processo in esecuzione non abbandona spontaneamente la CPU. La soluzione per evitare questo problema è quella di implementare una sorta di **preemption** nel sistema: si utilizza l'ingresso di un nuovo processo nella coda di livello zero come evento per far entrare in esecuzione lo scheduler.

## 3 Schedulazione di sistemi in *tempo reale*

Un sistema in **tempo reale** è un sistema all'interno delle quali sono imposti dei vincoli (**deadlines**) entro i quali i processi devono essere eseguiti. Questi sistemi sono utilizzati per la misurazione ed il controllo di grandezze fisiche di un ambiente esterno al sistema stesso attraverso l'utilizzo di:

- Sensori.
- Attuatori.

Abbiamo anche visto che nel caso dei sistemi **hard-real-time** la violazione dei vincoli costituisce un fatto inaccettabile che comporta anche conseguenza molto gravi, invece, nei sistemi **soft-real-time** la violazione di questi vincoli comporta soltanto una degradazione del servizio.

Una delle caratteristiche fondamentali di questi sistemi è la **prevedibilità**: è necessario essere sicuri, **a priori**, che durante l'esecuzione nessun processo possa violare nessuna delle **deadlines**, indipendentemente dalla condizione di carico del sistema. Diciamo che un'insieme di processi  $P = \{p_1, p_2, \dots, p_n\}$  è **schedulabile** mediante un dato algoritmo se e solo se, utilizzando quell'algoritmo di scheduling, e indipendentemente dalla condizione di carico del sistema, nessun processo potrà mai violare una **deadline**, generando quindi un **overflow temporale**. Questi sistemi operano, generalmente, con algoritmi di scheduling su **base priorità**. Il problema di questa tipologia di algoritmi deriva però dalla necessità di dover definire dei criteri con cui assegnare le priorità in modo che nessun processo possa violare una qualsiasi **deadline**. A tal proposito distinguiamo due tipologie di criteri

- **Criteri statici.** Si determinano le priorità dei processi in modo definitivo e invariato a seconda di alcune caratteristiche che sono note a priori.
- **Criteri dinamici.** Si determinano le priorità dei processi dinamicamente durante la loro esecuzione.

Un processo in tempo reale è caratterizzato da tre parametri: l'**istante di richiesta**  $r$ , che identifica l'istante in cui il processo viene attivato ed entra in coda pronti, **deadline**  $d$ , che identifica l'istante entro il quale l'esecuzione del processo deve essere terminata e, infine, il **tempo di esecuzione**  $C$ , che identifica il tempo della CPU complessivamente necessario al processo per terminare la propria esecuzione.

I **processi** di un sistema **real-time** sono periodici con periodo  $t_i$  (quindi le operazioni vengono ripetute ogni  $t_i$  millisecondi) e vengono quindi attivati con cadenza fissa in funzione delle operazioni che devono essere compiute sul sistema controllato. Ovviamente, affinché le deadlines siano rispettate, è necessario che, supponendo  $d_i$  come la deadline del processo  $i$ -esimo, valga  $d_i \leq t_i$ . Inoltre, in questi sistemi si suppone che l'ingresso in coda pronti dell'esecuzione  $i+1$ -esima del processo sia attuato all'istante:

$$r_{i+1} = r_i + t_i$$

Ciò implica che l'esecuzione  $i$ -esima del processo debba essere terminata prima dell'istante  $r_{i+1}$ . Inoltre, siccome il sistema è costituito da **soli processi periodici**, allora, sarà anch'esso periodico e, quindi, possiede un proprio **periodo** globale  $T$  definito come

$$T = \text{mcm}(t_1, \dots, t_n)$$

L'ultimo passo per ottenere un modello di facile comprensione è quello di supporre che la deadline relativa al processo  $i$ -esimo coincida con l'istante in cui il processo  $(i+1)$ -esimo entra in coda pronti, quindi;

$$r_{i+1} = d_i = r_i + t_i$$

Facendo in questo modo è possibile schedulare il sistema periodico in modo estremamente facile, poiché è possibile schedulare ogni singolo processo con processo  $T$  invece che  $t_i$  ottenendo, quindi,  $r_{i+1} = r_i + T$ .

Anche nella scheduling dei sistemi in tempo reale utilizziamo una scheduling su base prioritaria, in cui utilizziamo un criterio di proporzionalità inversa: **più alto è il periodo, minore sarà la priorità**. Questo criterio è noto come **rate-monotonic**:

$$P \propto \frac{1}{t_i}$$

Supponiamo di avere la seguente situazione: nel sistema si trovano due processi  $A$  e  $B$  tali per cui  $t_a = 2$  e  $t_b = 5$  e  $C_a = 1$  e  $C_b = 2$ . Ipotizziamo ora di avere entrambi i processi in coda pronti all'istante  $t = 0$  e di volerli scheduling con il **rate monotonic**. Siccome  $t_b > t_a$ , allora il processo  $A$  andrà sicuramente in esecuzione prima del processo  $B$ , poiché il **rate monotonic** privilegia tutti quei processi che hanno un periodo basso. Inoltre, il periodo totale del sistema corrisponde

a  $T = \text{mcm}(2, 5) = 10$ ; possiamo dunque limitare a studiare il sistema solo per questo periodo, poiché idealmente ogni  $T = k \cdot 10$  secondi si ripeterà quello che avviene nell'intervallo  $[0, 10]$ .

Una volta entrato in esecuzione, A, consumerà il suo **CPU burst** in una sola unità di tempo per poi sospendersi. A questo punto il sistema metterà in esecuzione il secondo processo; quest'ultimo però non riuscirà a terminare il suo **CPU burst** in quanto, a  $t = 2$ , il processo iniziale si riattiva (è passato un periodo) e quindi, avendo priorità maggiore, verrà messo in esecuzione. Una volta esaurito nuovamente il **CPU burst** del primo processo, il secondo entrerà nuovamente in esecuzione e terminerà di consumare il suo **CPU burst**.

Nuovamente, dopo la sospensione del processo B, tornerà in esecuzione il processo A che ripeterà le sue operazioni, concludendo il suo CPU burst all'istante 5. A questo istante si riattiverà anche il processo B, poiché è passato un intero periodo  $t_b$  dall'inizio della schedulazione. A questo punto si ripeterà quanto detto fino ad ora e, all'istante  $t = 9$ , entrambi i processi avranno esaurito i propri **CPU burst** per poi riattivarsi nuovamente all'istante  $t = 10$ .

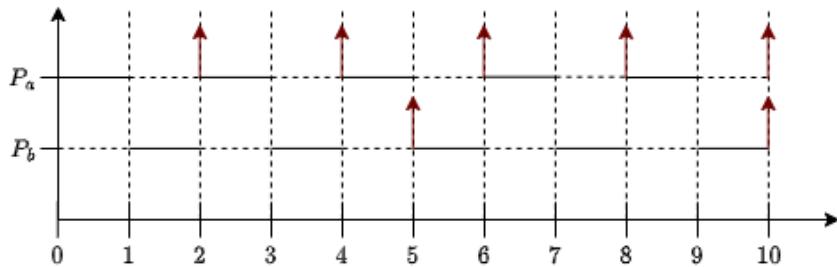


Figura 5.6: Esempio di schedulazione con *rate monotonic*

Possiamo anche dimostrare che l'algoritmo **rate monotonic** è ottimo nella classe degli algoritmi a priorità statica. Con ottimo intendiamo che se non è possibile schedulare un'insieme di processi  $P$  con **rate monotonic** allora non esiste alcun algoritmo a priorità statica che è in grado di schedulare  $P$ . Ricapitolando, supponendo di avere  $n$  processi da schedulare, ognuno con un **CPU-burst** di  $c_i$  e supponendo  $n_i$  come il numero di volte in cui esegue il processo  $p_i$ , dobbiamo garantire:

$$\sum_{i=1}^n C_i n_i \leq T \quad \sum_{i=1}^n \frac{T}{t_i} C_i \leq T \quad T \left( \sum_{i=1}^n \frac{C_i}{t_i} \right) \leq T \quad \boxed{\sum_{i=1}^n \frac{C_i}{t_i} \leq 1} \quad (1)$$

Definiamo la grandezza a sinistra come **fattore di utilizzazione** e la indichiamo con la lettera  $U$ . La condizione  $U \leq 1$  è una condizione **necessaria**, ma **non sufficiente**, affinché il sistema sia **schedulabile**. Supponiamo ora di prendere in esame il seguente **batch** di programmi:  $t_a = 4$ ,  $t_b = 10$  e  $C_a = 2$  e  $C_b = 5$

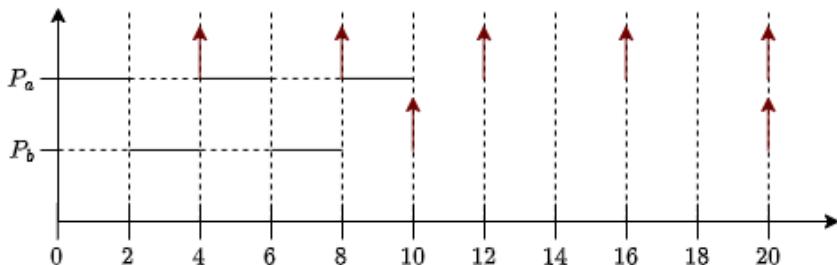


Figura 5.7: Esempio di schedulazione con *rate monotonic*

Il periodo in questo caso corrisponde a  $T = 20$ , quindi è sufficiente considerare 20 unità di tempo. Notiamo subito che con *rate-monotonic* si verifica un problema: all'istante  $t = 10$  il processo B non è ancora finito, ma comunque il suo periodo termina e quindi deve ripetere le operazioni.

Questa situazione corrisponde al verificarsi di un overflow temporale. Calcolando anche il fattore di utilizzazione notiamo che:

$$U = 1 \leq 1 \quad (2)$$

Quindi la condizione sull'utilizzazione viene rispettata, ma, nonostante ciò, l'insieme di processi non è schedulabile. Grazie a questo esempio abbiamo dimostrato che la condizione è necessaria, ma non sufficiente.

Per ottenere delle prestazioni migliori può essere comodo ricorrere a algoritmi di schedulazione a priorità dinamica, che assegnano la priorità non in funzione di criteri noti a priori, ma che l'assegnino in funzione di criteri che possono essere anche variaibili. Uno di questi algoritmi è noto come **Earliest Deadline First**; l'idea è quella di dare priorità maggiore ai processi con la deadline più vicina. Ipotizziamo di andare a schedulare il sistema precedente con questo algoritmo.

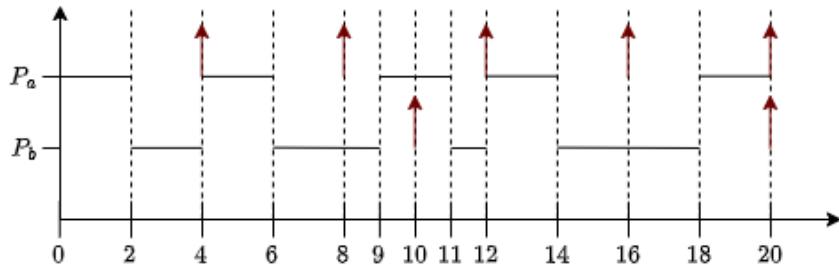


Figura 5.8: Esempio di schedulazione con *Earliest Deadline First*

L'algoritmo è abbastanza facile dal punto di vista esecutivo. All'istante  $t = 0$  il processo con la deadline più vicina sarà  $P_0$ , poiché la sua deadline è all'istante  $t = 4$ , mentre la deadline di  $P_1$  è all'istante  $t = 10$ . Una volta esaurito il CPU burst di  $P_0$ ,  $P_1$  potrà entrare in esecuzione per  $t = 2$  intervalli di tempo. All'istante  $t = 4$  il processo  $P_0$  e sarà nuovamente messo in esecuzione, in quanto la sua deadline, posta all'istante  $t = 8$ , è comunque più vicina della deadline del processo  $P_1$ . A questo punto  $P_0$  esaurirà nuovamente il suo CPU burst e lascierà la CPU, permettendo a  $P_1$  di tornare ad eseguire. A questo punto all'istante  $t = 8$  il processo  $P_0$  sarà nuovamente pronto, tuttavia però, essendo la deadline di  $P_0$  maggiore di quella di  $P_1$ ,  $P_1$  rimarrà in esecuzione, esaurendo definitivamente il suo CPU burst. Contrariamente a prima si ha quindi la schedulabilità. Analogamente al **rate monotonic**, anche l'algoritmo **EDF** è **ottimo** rispetto agli algoritmi con priorità dinamica.

## 4 Thread

Un processo è costituito da un'insieme di locazioni per i dati, le variabili locali e globali, dallo stack e dal suo descrittore. L'insieme di tutte queste informazioni prende il nome di **immagine** del processo. Come abbiamo anche visto, due processi coesistenti in un sistema possono comunicare tra di loro scambiandosi dei messaggi; quest'operazione ha però un costo che può essere anche oneroso in termini temporali. Una soluzione efficiente a questo problema è quello dei **thread**, detti anche **processi leggeri**. Un **thread** è definito come un flusso di esecuzione alternativo all'interno di un **processo** (o **processo pesante**). Suddividendo le varie funzionalità di un processo in flussi diversi di esecuzione in una moltitudine di thread (multithreading) si ottiene un netto miglioramento in termini di prestazioni. Ogni **thread** ha una sua area di memoria all'interno della memoria del processo e, ovviamente, anche un suo descrittore. Essendo però i thread dei processi leggeri tutte le funzionalità viste per i processi, come il **cambio di contesto**, sono estremamente semplificate. La gestione dei thread può avvenire in due modi

- **Gestione a livello utente:** in questo caso il sistema mette a disposizione delle primitive che permettono la creazione di Thread. Un thread creato a livello utente viene ignorato dal sistema operativo e viene invece gestito dal processo che lo ha creato quando viene messo in esecuzione.

- **Gestione a livello del nucleo:** In questo caso si assume che sia il sistema operativo a farsi carico di tutta la gestione della vita di un thread, mettendo a disposizione delle chiamate di sistema per terminazione, blocco, etc. Inoltre, il nucleo dovrà contenere anche i descrittori di tutti i thread oltre che i descrittori dei singoli processi.

La particolarità dei thread è anche data dalla possibilità di condividere alcuni file del processo che li ha creati, permettendo una comunicazione agile e semplificata.

# 6

## Sincronizzazione tra processi

In un sistema multiprogrammato i processi possono essere tra di loro concorrenti, quindi avere delle esecuzioni che possono essere sovrapposte nel tempo. Questa concorrenza può a sua volta portare a delle situazioni in cui i processi concorrenti interagiscono tra di loro; questa interazione può essere positiva o negativa, a seconda del tipo. Distinguiamo tre tipologie di concorrenza:

- **Cooperazione.** Situazione desiderata, dove i processi si scambiano informazioni e si sincronizzano per raggiungere uno scopo comune. Un caso molto famoso di **cooperazione** è quello del modello **produttore-consumatore**, dove i due attori si scambiano costantemente informazioni affinché l'accesso alla risorsa da parte di entrambi sia fatto correttamente e, soprattutto, rispettando dei vincoli di natura **temporale**. In questo caso parliamo di **sincronizzazione diretta o esplicita**.
- **Competizione.** Situazione **indesiderata** e **invisibile**, dove i processi competono per l'accesso a una risorsa condivisa che, tuttavia, non può essere usata in contemporanea. Si parla di situazione invisibile in quanto, a differenza della cooperazione, non è scritto nel programma che i processi dovranno competere per l'accesso ad una medesima risorsa. In questo caso parliamo di **sincronizzazione indiretta o implicita**.
- **Interferenza.** Situazione **pericolosa** e **indesiderata**, dove l'esecuzione di uno o più processi può interferire con l'esecuzione di altri processi portando a errori nel risultato finale.

La natura dei problemi di sincronizzazione è quindi, almeno sul piano concettuale, diversa. Si parla di **sincronizzazione diretta** (o **esplicita**) nel caso dei vincoli di cooperazione, oppure, di **sincronizzazione indiretta** (o **implicita**) per indicare i vincoli imposti dalla competizione. L'interferenza è, invece, la risposta a una gestione errata dei vincoli imposti dai problemi di competizione e di cooperazione.

La soluzione ai problemi di interazione è diversa e dipendente dagli strumenti di sincronizzazione messi a disposizione dalla tipologia di sistema scelta. Esistono due modelli principali di sistemi: il primo, detto anche **modello a memoria comune**, prevede che più processi pur avendo delle risorse private possano comunque condividere delle strutture dati comuni. La natura delle strutture condivise non è fondamentale, potremmo avere dei contatori condivisi così come dei buffer condivisi, l'idea generale è però che queste strutture dati abbiano come scopo quello di permettere la sincronizzazione, sia diretta che indiretta, dei processi. Il secondo, detto modello a **scambio di messaggio**, prevede che ogni processo abbia una sua memoria privata e che la sincronizzazione tra processi avvenga mediante lo scambio di messaggi con un formato specifico, definito dal protocollo scelto. Internet, nella sua visione organica, è un sistema basato sul modello a scambio di messaggio.

### 1 Problema della mutua esclusione

La competizione tra processi per una risorsa condivisa, la quale non è accessibile contemporaneamente, è un problema che richiede l'imposizione di vincoli speciali, che permettano l'accesso a tale

risorsa ad un processo per volta, detti **vincoli di mutua esclusione**. In altre parole, il vincolo di mutua esclusione richiede che le operazioni con le quali i processi accedono alle variabili comuni non si sovrappongano nel tempo. Tuttavia i vincoli di **mutua esclusione** non pongono alcuna condizione sull'ordine con cui i processi accedono alle risorse. L'insieme delle operazioni che vengono svolte da un processo su una risorsa condivisa sono dette **sezioni critiche**. Supponiamo di avere la seguenti situazione: due processi  $P_1$  e  $P_2$  hanno accesso a una struttura organizzata a pila per prelevare e inserire dati. I processi hanno a disposizione due primitive, **Inserimento** e **Prelievo**, per l'accesso alla risorsa condivisa; in particolare: la primitiva **Inserimento** inserisce un elemento in testa allo stack e incrementa il contatore di testa, la primitiva **Prelievo** legge il dato che si trova in testa e decrementa il rispettivo contatore. Tali operazioni possono essere quindi così descritte

```
void Prelievo(){
    void Inserimento(T y){
        top++;
        stack[top] = y;
    }
    T temp;
    temp = stack[top];
    top--;
    return temp;
}
```

Supponiamo anche che non sia imposto alcun vincolo di mutua esclusione e che i due processi eseguano, rispettivamente, una **Inserimento** e una **Prelievo**; una delle possibili sequenze di esecuzioni potrebbe essere

```
top++;           // P1
temp = stack[top]; // P2
top--;           // P1
stack[top] = y   // P2
```

Il risultato è che viene prelevato dalla pila un risultato non definito e, inoltre, l'ultimo valore contenuto nella pila viene cancellato dal nuovo valore. Una prima soluzione potrebbe essere quella di temporizzare i processi in modo che le **sezioni critiche** di entrambi vengano sempre eseguite in intervalli di tempo differenti. Una soluzione del genere però richiederebbe al programmatore la conoscenza della velocità relativa di entrambe le **sezioni critiche** e, ciò, non è in generale possibile. Un'ulteriore soluzione, anche se parziale, è quella di interrompere le interruzioni durante l'esecuzione delle **sezioni critiche**, ma nell'idea che si lavori con un sistema con più CPU quest'idea diventa, di fatto, inutile. L'idea vincente è quella di porre due condizioni:

- Le **sezioni critiche** devono essere **atomiche**.
- Deve essere definito un ordinamento temporale tra i processi.

Combinando queste due condizioni è possibile giungere a una situazione in cui le due **sezioni critiche** sono mutuamente esclusive e, soprattutto, non conflittuali. In generale è necessario definire un protocollo che i processi devono adottare per interagire correttamente con una risorsa comune. In altri termini, ogni processo prima di entrare in una sezione critica dovrà chiedere l'autorizzazione eseguendo una serie di operazioni che gli garantiscano l'accesso esclusivo alla risorsa, se è libera. Inoltre, una volta terminato di eseguire la sezione critica dovrà rilasciarla eseguendo un'ulteriore serie di operazioni; questi due insiemi di istruzioni prendono il nome di **prologo** e di **epilogo**. Il modo più semplice per definire il prologo e l'epilogo è mediante l'utilizzo di una variabile condivisa **occupato** che può assumere due valori, 1 quando la risorsa è occupata e 0 quando la risorsa è libera. Una possibile implementazione del controllo di mutua esclusione potrebbe essere:

```
//prologo
while(occupato == 1);
occupato = 1;

//epilogo:
occupato = 0;
```

Questa soluzione presenta però problemi nel caso in cui due processi siano in attesa nel ciclo while, infatti se all'istante  $t = t_x$  la variabile **occupato** viene messa a 0, entrambi i processi in attesa nel

prologo accederanno alla sezione critica, violando il vincolo di mutua esclusione. L'errore chiave di questo sistema deriva proprio dal fatto che `occupato` è a sua volta una variabile condivisa con delle **sezioni critiche** e, quindi, dovrebbe essere gestita come tale. Per gestire anche `occupato` come variabile condivisa si dovrebbe creare una variabile, anch'essa condivisa, che regoli l'accesso a `occupato`; si nota facilmente che questa soluzione è inattuabile, poiché introduce delle catene di variabili potenzialmente infinite. Un altro errore di questa soluzione è che il processo in attesa di eseguire la sezione critica si mette in una condizione di **busy-wait**, restando in attesa passiva senza poter fare nulla.

Una diversa soluzione che elimina il primo problema si basa su due primitive implementate in linguaggio macchina, chiamate `lock(x)` e `unlock(x)`, che hanno un meccanismo simile a quello di un lucchetto. Le due primitive sono così strutturate:

<code>lock(x):</code> TSL registro, x CMP registro, 0 JNE lock RET	<code>unlock(x):</code> MOV x, 0 RET
--	--

L'istruzione TSL, detta anche **Test and Set Lock**, funziona copiando il valore contenuto nella locazione  $x$  all'interno del registro  $R$  e viene scritto in  $x$  un valore diverso da 0. Affinché sia però garantita l'atomicità dell'istruzione TSL, cioè che le istruzioni di lettura e di scrittura vengano eseguite in modo indivisibile, la **CPU** deve bloccare il bus di memoria per impedire ad altre CPU di accedere alla locazione all'indirizzo  $x$ . Introducendo queste due primitive risulta semplice ricavare anche la corrispettiva soluzione del problema di mutua esclusione;

<code>lock(x)</code> <sezione critica A> <code>unlock(x)</code>	<code>lock(x)</code> <sezione critica di B> <code>unlock(x)</code>
---	--

Anche questa soluzione presenta però un problema, la `lock` causa una *busy-wait* del processore sulle istruzioni per l'accesso alla sezione critica. Più avanti verranno introdotti delle strutture particolari, dette **semafori**, in grado di sopperire a questo problema.

## 2 Problemi di comunicazione

Uno dei principali problemi nel campo della comunicazione tra processi è quello che va sotto al nome di **produttore-consumatore**. In questo problema due processi condividono un'area di memoria comune, generalmente un buffer. Il **produttore**, ciclicamente, inserisce informazioni all'interno del buffer, mentre il **consumatore** li preleva. Potremmo anche generalizzare il problema supponendo l'esistenza di **n** produttori e **m** consumatori. In questo caso i vincoli imposti nell'esecuzione delle rispettive operazioni sono due:

- Il produttore non può inserire nel buffer un nuovo messaggio prima che il consumatore abbia prelevato il precedente.
- Il consumatore non può prelevare dal buffer un nuovo messaggio prima che il produttore l'abbia depositato.

Anche in questo caso si presenta un problema di mutua esclusione nell'utilizzo del buffer, tuttavia, a differenza del problema visto in precedenza, la soluzione a questo problema impone un ordinamento delle operazioni da parte di **produttore** e **consumatore**. Se nel caso del problema di mutua esclusione l'ordinamento era indifferente, nel caso dei problemi di comunicazione l'ordinamento delle operazioni è una componente fondamentale e, quindi, è richiesto che i processi si scambino messaggi per indicare l'avvenuto deposito e l'avvenuto ritiro; abbiamo quindi un problema di **sincronizzazione diretta esplicita**. Una variante di questo problema prevedere di avere un buffer costituito da un array, permettendo quindi di depositare più messaggi insieme. Le regole in questo caso risultano meno vincolanti, infatti: il produttore non può inserire un messaggio nel

buffer se questo è pieno, mentre il consumatore non può prelevare un messaggio dal buffer se questo è vuoto.

In generale, anche in questo caso è possibile costruire diverse soluzioni, anche se in generale si preferisce optare per l'utilizzo di semafori.

### 3 Semafori

Uno strumento generale per la risoluzione dei problemi di sincronizzazione è costituito dai **semafori**. L'idea alla base di questo meccanismo è quella di utilizzare una coda per sospendere i processi in attesa di una particolare risorsa. Un semaforo è una struttura dati costituita da due campi:

- Una variabile intera non negativa, indicata con `s.value`, con valore iniziale  $s_0 \geq 0$ .
- Una lista di descrittori di processo, indicata con `s.queue`.

La variabile `s.value` serve a gestire l'accesso alla risorsa, funzionando come un vero e proprio gettone che viene prelevato dalla scatola quando un processo vuole accedere ad una risorsa e che viene depositato quando un processo vuole rilasciarla. La coda dei descrittori `s.queue` ha, invece, come unico scopo quello di tenere conto dei processi *in attesa* per la risorsa.

Ad ogni semaforo sono associate due primitive: una primitiva `wait(s)` e una primitiva `signal(s)`. La primitiva `wait` ha come scopo quello di permettere ad un processo di sospendersi nel caso in cui una risorsa sia occupata o di occuparla nel caso la risorsa sia libera, mentre la primitiva `signal(s)` ha come scopo quello di rilasciare una risorsa condivisa e, se ci sono processi sospesi in attesa della risorsa, risvegliare il primo nella coda. Le primitive sono così strutturate:

```

void signal(s){
    if (s.queue != nullptr){
        <estrai processo dalla coda
        s.queue>
        <cambia lo stato del processo
        estratto in pronto>
    } else {
        s.value += 1;
    }
}

```

Figura 6.1: Struttura delle primitive `signal` e `wait`. Un'implementazione più esaustiva è quella del nucleo di calcolatori elettronici, trovabile al seguente <https://calcolatori.iet.unipi.it/resources/nucleo-8.3.tar.gz>

I semafori risolvono quindi il problema della **busy-wait**, in quanto evitano il formarsi di condizioni di attesa attiva. Questa condizione viene garantita proprio dal fatto che nell'impossibilità, da parte di un processo, di accedere alla sezione critica, si sospenderà permettendo alla CPU di proseguire con un'altra elaborazione.

La risoluzione del problema di mutua esclusione diventa estremamente facile attraverso i semafori. Il primo passo è quello di creare un semaforo `mutex` per l'accesso a una particolare risorsa condivisa e, successivamente, strutturare la sezione critica nel seguente modo

```

wait(mutex);           // prologo
<sezione critica>
signal(mutex);       // epilogo

```

Il semaforo `mutex` dovrà essere, ovviamente, inizializzato ad uno, poiché solo un processo per volta può accedere alla risorsa. Questo tipo di semafori, infatti, prende il nome di **semafori binari**. Fintanto che i processi lavorano tutti sullo stesso processore, allora l'atomicità delle operazioni si `wait` e `signal` è garantito dal blocco delle interruzioni durante l'esecuzione. Qualora però i processi siano eseguiti su processori diversi, questa condizione non è più sufficiente. Infatti, se due processi in esecuzione su due processori diversi eseguissero una `wait` allo stesso istante, potrebbero accedere

alla struttura dati in modo non mutuamente esclusivo causando problemi di inconsistenza. Per garantire che `wait` e `signal` siano mutuamente esclusive sul semaforo `mutex`, bisogna ricorrere alle funzioni `lock` e `unlock`; uno dei possibili schemi è

```
lock(x);
    wait(mutex);
unlock(x);
<sezione critica>
lock(x);
    signal(mutex);
unlock(x);
```

Dove  $x$  deve corrispondere all'area di memoria dove è contenuto il semaforo. La soluzione al problema del produttore consumatore attraverso i semafori è relativamente facile. Il primo passo è quello di inizializzare due semafori, uno `messaggio_disponibile` e uno `spazio_disponibile`, inizializzati, rispettivamente, con un valore pari allo spazio disponibile nel buffer e con il numero di messaggi disponibili da prelevare nel buffer. Infine, produttore e consumatore, devono essere strutturati nel seguente modo. Questo caso però è relativo al caso in cui abbiamo un solo produttore

```
// produttore
do{
    <prod messaggio>;
    wait(spazio_disp);
    <inserisci mess>
    signal(messaggio_disp);
} while(!fine)

// consumatore
do{
    wait(messaggio_disp);
    <preleva mess>
    signal(spazio_disp);
} while(!fine)
```

e un solo consumatore. Nel caso in cui il problema preveda più produttori e più consumatori si rischia che più consumatori o più produttori possano accedere al buffer condiviso allo stesso tempo, creando un problema di **mutua esclusione**. Per risolvere questo problema è necessario ricorrere all'utilizzo del semaforo `mutex`

```
...
wait(mutex);
<preleva/inserisci messaggio>
signal(mutex);
...
```

Il problema di questa soluzione deriva dal fatto che se un produttore sta inserendo un dato alla posizione  $x$  del buffer e un consumatore, contemporaneamente, vuole prelevare un dato da una posizione  $y < x$  del buffer, non potrà farlo in quanto il semaforo `mutex` permette l'accesso a solo un processo per volta. Siccome la sincronizzazione tra produttori e consumatori è già garantita dai due semafori `spazio_disponibile` e `messaggio_disponibile` è sufficiente suddividere `mutex` in due semafori, uno `mutexP` per sincronizzare i produttori e uno `mutexC` per sincronizzare i consumatori

```
// produttore
do{
    <prod messaggio>;
    wait(spazio_disp);
    wait(mutexP);
    <inserisci mess>
    signal(mutexP);
    signal(messaggio_disp);
} while(!fine)

// consumatore
do{
    wait(messaggio_disp);
    wait(mutexC);
    <preleva mess>
    signal(mutexC);
    signal(spazio_disp);
} while(!fine)
```

## 4 Elementi di sincronizzazione nel modello a scambio di messaggio

In un qualsiasi ambiente a memoria comune ogni processo possiede una propria area di memoria privata e nessuna area di memoria condivisa. La sincronizzazione tra processi può quindi avvenire solo mediante uno scambio di messaggi. Questa comunicazione avviene grazie a un meccanismo chiamato **IPC** (o **Inter-Process-Communication**), offerto dal nucleo del sistema operativo mediante il quale avviene la comunicazione tra processi. Lo strumento **IPC** più generale sfrutta due primitive `send()` e `receive()`. In particolare:

- La primitiva `send(destinazione, messaggio)` spedisce un messaggio a una determinata destinazione.
- La primitiva `receive(origine, messaggio)` riceve un messaggio da una determinata origine.

Al centro di questa comunicazione troviamo il **canale di comunicazione**, che nel caso dei sistemi a memoria privata è, generalmente, realizzato attraverso un collegamento diretto tra i processi su cui operano le CPU. Esiste anche un formato di messaggio che i vari processi si scambiano tra di loro:

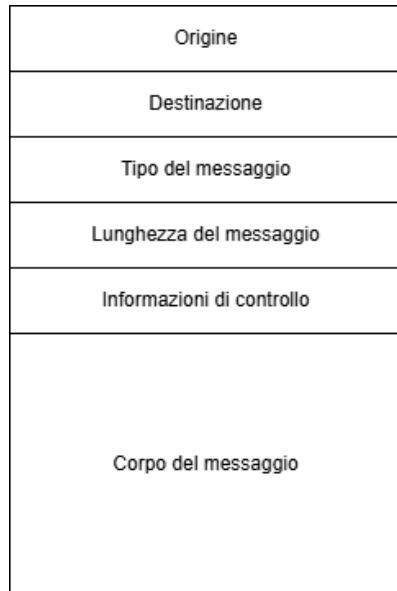


Figura 6.2: Formato dei messaggi nel modello a scambio di messaggio

In un sistema a memoria privata i processi possono comunicare tra di loro in due modi: comunicazione **asincrona** e comunicazione **sincrona**. Nella prima forma di comunicazione un processo  $P_1$  che fa una `send()` non si blocca in attesa della risposta, ma continua nel suo normale flusso di istruzioni. Nella seconda forma di comunicazione un processo  $P_1$  che fa una `send()` si blocca in attesa della risposta del destinatario. Esiste anche una terza forma di `send`, detta **send tipo chiamata a procedura remota**, il cui funzionamento è analogo a quello della send sincrona, ma il cui obiettivo è quello di far eseguire una particolare procedura al processo destinatario. Un'altra classificazione è quella che divide la comunicazione in **diretta** o **indiretta**:

- Nella comunicazione **diretta** il mittente specifica direttamente il processo destinatario all'interno della `send` e il destinatario specifica direttamente il mittente nella `receive`.
- Nella comunicazione **indiretta** il mittente specifica un buffer in cui vengono inseriti i messaggi, detto **mailbox**, gestito dal sistema operativo, e il destinatario preleva direttamente da quella casella.

Un caso particolare della comunicazione **diretta** è quello della **comunicazione diretta asimmetrica**, dove il mittente specifica direttamente il destinatario della comunicazione, ma il destinatario non specifica direttamente il mittente, in quanto può ricevere messaggi da diversi processi.

## 5 Problema dei lettori e scrittori

Un problema analogo a quello dei produttori consumatori è il problema dei **lettori-scrittori**. Supponiamo di avere un file condiviso e di avere  $n$  processi, chiamati **lettori**, che possono leggere il buffer senza però modificarlo e  $m$  processi, chiamati **scrittori**, che possono scrivere sul file. La regola di sincronizzazione, in questo problema, dovrebbe permettere l'accesso mutuamente esclusivo agli scrittori, cioè quando lo scrittore inizia la sua sezione critica non ci devono essere dei lettori che leggono nel file e ne tanto meno altri scrittori. In realtà esiste una variante del problema nella quale si ammette che un lettore possa continuare a leggere anche nel caso in cui uno scrittore inizi la sua sezione critica. Utilizzando i semafori dobbiamo definire due semafori

- Un semaforo di mutua esclusione, chiamato **wrt**, tra scrittori e lettori.
- Un intero, chiamato **read\_count**, utilizzato per contare quanti sono i lettori attivi.

Il primo semaforo, essendo di mutua esclusione, deve essere inizializzato ad uno, mentre **read\_count**, essendo un semaforo di conteggio, può anche assumere valori maggiori, ma deve essere inizializzato a 0. Utilizzando questi due semafori i codici per scrittori e lettori sono

```
// lettore
do{
    // scrittore
    do{
        wait(wrt);
        <sezione critica>
        signal(wrt);
    }
    // lettore
    do{
        read_count++;
        if(read_count == 1)
            wait(wrt);
        <sezione critica reader>
        read_count--;
        if (read_count == 0)
            signal(wrt);
    } while(!fine)
```

Il primo lettore dovrà garantire la mutua esclusione tra lettori e scrittore, mentre l'ultimo dei lettori sarà quello che eseguirà che eseguirà la **signal** per rilasciare la risorsa. Il problema di questa soluzione è immediato: **l'accesso a read\_count deve essere atomico, poiché è una variabile condivisa**. Inoltre, se l'accesso non fosse atomico si correrebbe il rischio che tutti i lettori dopo il primo accedano alla sezione critica anche prima che lo scrittore abbia finito di scrivere (la condizione per la **wait()** viene infatti applicata solo per il primo lettore). Per risolvere questo problema è quindi necessario introdurre un ulteriore semaforo, chiamato **mutex**, che permetta l'accesso mutuamente esclusivo a **read\_count**.

```
// lettore
do{
    wait(mutex);
    read_count++;
    if(read_count == 1)
        wait(wrt);
    signal(mutex);
    <sezione critica reader>

    wait(mutex);
    read_count--;
    if (read_count == 0)
        signal(wrt);
    signal(mutex);
} while(!fine)
```

In questa variante l'accesso a `read_count` viene gestito in modo atomico. Inoltre, la possibilità di entrare nella sezione critica per un lettore diventa disponibile solo quando il primo lettore rilascia `mutex`. Questo fatto implica che nella coda del semaforo `wrt` si può accedere solo il primo lettore, mentre tutti gli altri si accorderanno su `mutex`. Infatti, dalla struttura del codice, se il primo lettore si blocca sulla `wait(wrt)` non potrà eseguire `signal(mutex)` e, quindi, tutti gli altri lettori si bloccheranno sulla coda di `mutex`.

## 6 Problema dei filosofi a cena

Uno dei problemi più famosi della programmazione concorrente è il **problema dei filosofi a cena**, detto anche **problema dei cinque filosofi**. Si supponga di avere 5 processi, chiamati **filosofi**, che trascorrono la loro esistenza facendo una tra due operazioni: **mangiare** e **pensare**. I filosofi sono seduti ad un tavolo circolare, con una ciotola di riso al centro e 5 bacchette disposte tra i vari filosofi. Graficamente possiamo rappresentar la situazione come:

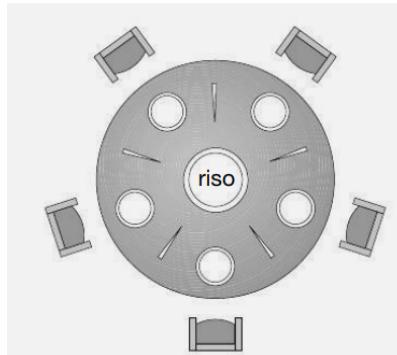


Figura 6.3: Problema dei cinque filosofi

Il funzionamento del filosofo è relativamente semplice: quando un filosofo **pensa** non interagisce con gli altri filosofi, mentre quando gli viene fame tenta di prendere le bacchette più vicine, quindi quelle che si trovano tra lui e i commensali alla sua destra e alla sua sinistra. Un primo vincolo che possiamo impostare è che un filosofo può prendere una sola bacchetta per volta e non può prendere una bacchetta se si trova già nelle mani di un suo collega. Quando un filosofo ha due bacchette può cominciare a mangiare e, una volta terminato, rilascia le bacchette. Da queste due regole possiamo facilmente dedurre due considerazioni:

- Se un filosofo sta mangiando nessuno dei filosofi adiacenti può mangiare.
- Solo due filosofi in contemporanea possono mangiare.

Il problema di sincronizzazione, in questo esempio, deriva dalla necessità di assegnare le bacchette ai filosofi in modo che si eviti la situazione di stallo in cui ogni filosofo ha una sola bacchetta ed è bloccato in attesa della seconda.

La prima soluzione, che è anche la più semplice, è quella di sfruttare i semafori. L'idea è quella di creare un semaforo di mutua esclusione per ogni singola bacchetta, quindi `semaphore chopstick[5]`, e che ogni filosofo esegua due `wait`: la prima `wait` per la bacchetta a destra, la seconda `wait` per la bacchetta a sinistra. Implementando questa soluzione il codice per un filosofo diventa:

```
do{
    wait(chopstick[i]);
    wait(chopstick[i+1] % 5);
    <mangiare>
    signal(chopstick[i]);
    signal(chopstick[i+1] % 5);
```

```

<pensare>
} while(true);

```

Tuttavia questa prima soluzione garantisce solo che due vicini non possano mangiare contemporaneamente, garantendo il primo vincolo di correttezza. Tuttavia non vengono esclusi i problemi di stallo dovuti alla corsa concorrente per le bacchette. Esistono diversi casi in cui si può verificare uno stallo applicando la seguente soluzione. Supponiamo, ad esempio, che un processo  $p_0$  inizi la sua esecuzione e prenda la prima bacchetta e ipotizziamo che il processo venga interrotto dal processo  $p_1$  prima di poter prendere la seconda. Il secondo processo,  $p_1$ , entrerà ora in esecuzione e anch'esso prenderà una bacchetta, arrivato però alla seconda `wait` troverà che la bacchetta non è disponibile, poiché già presa da  $p_0$  e si bloccherà. A questo punto, se entra in esecuzione  $p_2$ , per il stesso motivo di  $p_1$ , finirà per bloccarsi. A questo punto, se entrano in esecuzione  $p_3, p_4$  e  $p_5$ , in questo ordine, ogni processo finirà per bloccarsi per lo stesso motivo. Si crea quindi una situazione in tutti i processi filosofo possiedono una bacchetta e sono bloccati nella `wait` per la seconda bacchetta. Questa situazione è un chiaro esempio di **deadlock** del sistema: ogni processo è bloccato in attesa di una risorsa che possiede l'altro processo in modo circolare.

L'uso dei semafori è quindi sconveniente, oltre per quanto appena enunciato, anche per il fatto che non è immune a errori di programmazione. Potrebbe infatti capitare che un programmatore, involontariamente, inverta `signal()` e `wait()`, causando non pochi problemi.

## 7 Monitor

Per rimediare a questi errori sono stati sviluppati dei costrutti ad alto livello per la sincronizzazione, uno di questi è chiamato **monitor**. L'idea del monitor è quella di encapsulare i dati all'interno di un **contenitore protetto**, gestendo l'accesso alle risorse attraverso delle funzioni messe a disposizione dal monitor. Formalmente il monitor è un tipo di dato astratto che contiene un insieme di operazioni definite dal programmatore che, all'interno del monitor, sono contraddistinte dalla mutua esclusione. In generale, la struttura di un monitor è la seguente:

```

monitor name {
    function P1(...){
        ...
    }
    function P2(...){
        ...
    }
    function P3(...){
        ...
    }
}

```

Tuttavia questa struttura è solo parziale, se da una parte risolve i problemi di mutua esclusione, dall'altra non risolve i problemi di sincronizzazione. Sono quindi necessari ulteriori meccanismi forniti dal costrutto **condition**. Per implementare un particolare schema di sincronizzazione è opportuno definire una o più variabili di tipo **condition**. Le uniche due operazioni eseguibili su una variabile condition sono

- `wait()` è una funzione membro che blocca il processo che l'ha invocato.
- `signal()` è una funzione membro che risveglia un processo bloccato dalla `wait()`.

Se non ci fossero processi sospesi l'operazione di `signal()` non avrebbe alcun effetto, vale a dire che lo stato della variabile **condition** rimarrebbe immutato. Una funzione definita all'interno di un monitor ha accesso unicamente alle variabili dichiarate localmente, cioè all'interno del monitor stesso. Analogamente, le variabili definite all'interno di un monitor possono accedere solo a procedure definite localmente. Per realizzare i monitor è possibile utilizzare i semafori; si associa, ad ogni monitor di sistema, un semaforo **mutex** che permette l'accesso ad un solo processo per volta all'interno del monitor. In altri termini, l'accesso di un processo ad un monitor può essere fatto solo

nel momento in cui il processo precedente ha eseguito la `signal()` sul `mutex` del monitor. Questa soluzione, nonostante risolva il problema della mutua esclusione del monitor, tralascia un dettaglio che non è trascurabile: all'interno del monitor è possibile definire delle variabili `condition` che permettono a un processo all'interno del monitor di sospendersi (`x.wait()`) e di svegliarne un altro (`x.signal()`). Le variabili `condition` creano quindi un problema: se il processo  $P_i$  che si trova all'interno del monitor si sospendesse con una `x.wait()` senza prima rilasciare il `mutex` si formerebbe una situazione di stallo totale. Infatti nessun processo sarebbe in grado di entrare nel monitor e, quindi, di risvegliare i processi bloccati. Un problema di questo tipo può essere risolto permettendo di rilasciare il `lock` su `mutex` prima che un processo si blocchi. Tuttavia, anche risolvendo questo problema, c'è un'altra questione che rimane irrisolta: se il processo  $P_i$  che si trova all'interno del monitor utilizzasse una `x.signal()`, risvegliando un altro processo, si formerebbe un problema di accesso ad una risorsa non condivisibili (due processi sarebbero attivi allo stesso tempo nel monitor). Per risolvere questo problema si adotta una politica **signal-and-wait**: se un processo che ne risveglia un altro attraverso una `signal()` si sospende fintanto che il processo risvegliato non ha terminato l'esecuzione.

Per ogni variabile `condition` abbiamo detto che sono definite due funzioni, la prima, detta `wait()` permette di sospendere il processo all'interno di una coda di processi, mentre la seconda, detta `signal()`, servirà a risvegliare un processo che si trova sospeso per effetto di una `wait()`. Siccome un processo si sospende effettivamente sulla variabile `condition`, è necessario che per ognuna di essa sia definito un semaforo; insieme al semaforo può essere anche utile tenere conto di quanti processi sono attualmente bloccati sulla `wait` della variabile. L'implementazione della `wait()` diventa quindi

```
void wait() {
    x_count++;
    wait(x_sem);
}
```

Tuttavia, una struttura di questo tipo non risolve il problema che avevamo descritto in precedenza, se infatti il processo che esegue la `wait()` non rilasciasse il semaforo `mutex`, finirebbe per bloccare interamente il monitor. Di conseguenza è necessario che al momento della `wait()` venga opportunamente gestita la situazione affinché un altro processo diventi attivo all'interno del monitor. Un'idea molto elegante potrebbe essere quella di utilizzare un particolare `token` per l'esecuzione delle operazioni all'interno del semaforo; l'idea del `token` è abbastanza semplice: *fintanto che ho il token posso eseguire qualsiasi operazione all'interno del monitor, nel momento in cui mi sospendo passo il token a qualcun altro*. Un processo che vuole eseguire un'operazione e che si trova all'interno del monitor (magari un processo risvegliato con la `signal()`) dovrà quindi mettersi in attesa di ricevere il `token`; definiamo quindi anche un semaforo `next` e un contatore `next_count`. A questo punto possiamo definire una funzione `wait` e una funzione `signal` più complete e sofisticate. In altri termini, nella `wait()`, prima di sospendersi si valuta se ci sono, prima di tutto, dei

```
// funzione wait()
x_count++;
if(next_count > 0)
    signal(next);
else
    signal(mutex);
wait(x_sem);
x_count--;
```

```
// funzione signal()
if(x_count > 0){
    next_count++;
    signal(x_sem);
    wait(next);
    next_count--;
```

processi in attesa già all'interno del monitor, in qual caso si chiama la `signal` per svegliare uno di quelli, altrimenti si esegue una `signal` su `mutex`. Quindi, per definire una generica procedura all'interno di un monitor è, innanzitutto, necessario eseguire una `wait()` sul semaforo `mutex`, così da poter accedere effettivamente al **monitor**. Una volta fatto l'accesso al monitor è quindi possibile eseguire il corpo della procedura e, infine, rilasciare il lock. Nel caso in cui ci siano altri processi già all'interno del monitor che intendono eseguire si farà una `signal` su `next`, altrimenti su `mutex`:

```
wait(mutex);
<body P>
```

```

if(next_count > 0)
    signal(next);
else
    signal(mutex);

```

Una politica differente rispetto a quella **signal-and-wait** è la politica **signal-and-continue**, in cui invece il processo risvegliato attende che il processo che era prima in esecuzione termini. Inoltre, se diversi processi sono in attesa su una **condition**, è necessario che sia definita una politica per scegliere quale dei processi risvegliare; generalmente si può pensare di utilizzare una politica **FCFS**, che però risulta inadeguata, in quanto è una politica che non tiene conto di tutti i processi che non sono nel monitor. Supponiamo di avere un costrutto di wait nella forma **x.wait(c)**, che assegna una priorità ad un processo e permetterà, al momento del risveglio di un processo, di decidere quale processo risvegliare.

### Soluzione del problema dei filosofi a cena con l'utilizzo dei monitor

Per risolvere il problema dei filosofi a cena attraverso i monitor è opportuno definire un **monitor** apposito. Tale monitor dovrà contenere una struttura dati che codifica lo stato in cui un filosofo può trovarsi, in particolare gli stati possibili sono: **THINKING**, **HUNGRY**, **EATING**. Un filosofo può impostare il suo stato a **EATING** solamente nel caso in cui i suoi due vicini non si trovano già in tale stato. Definiamo, quindi, anche una struttura dati di tipo **condition**, chiamata **self**, che permette al filosofo *i*-esimo di bloccarsi quando ha fame ma non riesce ad ottenere le bacchette di cui ha bisogno. All'interno del monitor dobbiamo anche definire le seguenti funzioni

- **test(int i)**: questa funzione verifica se i colleghi adiacenti al filosofo *i*-esimo non stanno mangiando e se il filosofo *i*-esimo è affamato. In caso queste condizioni siano verificate lo stato del filosofo *i* viene messo a **EATING** e, nel caso fosse stato bloccato da una **wait**, viene anche sbloccato.
- **pickup(int i)**: Imposta lo stato del filosofo *i* come **HUNGRY**, controlla se è possibile mangiare attraverso la funzione di **test(i)** e, se non è stato possibile, blocca il processo *i* con una **wait()**.
- **putdown(int i)**: questa funzione ripristina lo stato del filosofo *i* a **THINKING**, inoltre verifica anche se i filosofi adiacenti vogliono mangiare o meno.
- **initialization\_code()**: inizializza tutti i filosofi nello stato di **THINKING**.

Tuttavia, nonostante questa soluzione sia esente da **deadlock**, non lo è da un possibile **starvation** di uno dei filosofi. Se infatti  $p_0$  e  $p_3$  mangiassero ciclicamente, il processo  $p_2$  potrebbe entrare in **starvation** poiché impossibilitato a mangiare. La codifica del monitor corrisponde a:

```

monitor DiningPhilosopher{
    enum {THINKING, HUNGRY, EATING} state[5];
    condition self[5];

    void pickup(int i){
        state[i] = HUNGRY;
        test(i);
        if (state[i] != EATING)
            self[i].wait();
    }

    void putdown(int i){
        state[i] = THINKING;
        test((i+4) % 5);
        test((i+1) % 5);
    }
}

```

```
void test(int i){  
    if(state[(i+4) % 5] != EATING &&  
        state[i] == HUNGRY &&  
        state[(i+1) % 5] != EATING){  
        state[i] = EATING;  
        self[i].signal();  
    }  
}  
  
Initialization_code(){  
    for(int i = 0; i < 5; i ++);  
        state[i] = THINKING;  
}  
}
```

# 7

## Deadlock

Fino ad ora abbiamo parlato di **deadlock** come di una situazione indesiderata, in cui ogni processo è bloccato in attesa di una risorsa posseduta da un altro e che comporta uno stallo sistematico del sistema. Formalmente il deadlock è una situazione del sistema in cui ogni processo si trova nello stato di bloccato dalla quale non può uscire. Abbiamo visto che questa situazione si può verificare anche nel problema dei filosofi a cena, nel caso in cui ogni filosofo riesca ad acquisire una sola bacchetta.

Formalmente, affinché si verifichi un deadlock, è necessario che si verifichino contemporaneamente le seguenti quattro condizioni:

- **Mutua esclusione.** Almeno una risorsa deve essere non condivisibile, vale a dire che è utilizzabile da un solo processo alla volta.
- **Possesso e Attesa.** Un processo deve essere in possesso di almeno una risorsa e attendere di acquisire risorse già in possesso di altri processi.
- **Assenza di Preemption.** Le risorse non possono essere rimosse forzatamente dal sistema, vale a dire che una risorsa può essere rilasciata solo dal processo che la possiede volontariamente, dopo aver terminato la sua esecuzione.
- **Attesa circolare.** Deve esistere un insieme di  $\{P_0, P_1, \dots, P_n\}$  di processi tale che  $P_0$  attende una risorsa posseduta da  $P_1$ ,  $P_1$  attende una risorsa posseduta da  $P_2$  e così via fino a  $P_n$  che attende una risorsa posseduta da  $P_0$ .

Queste quattro condizioni devono essere vere allo stesso tempo affinché si possa verificare uno stallo; se anche solo una di esse fosse farla il deadlock sarebbe una situazione impossibile.

## 1 Grafo di assegnazione delle risorse

Per descrivere una situazione di stallo è possibile avvalersi di uno strumento di rappresentazione detto **grafo di assegnazione delle risorse**. Si tratta, formalmente, di un insieme di vertici  $V$  e di archi  $E$ , con l'insieme di vertici  $V$  composto da due sottoinsiemi: un primo insieme  $P = \{P_0, P_1, \dots, P_n\}$  che identifica i processi del sistema e un secondo insieme  $R = \{R_1, \dots, R_n\}$  che identifica le risorse del sistema. Inoltre, vale che

- Un arco diretto dal processo  $P_i$  alla risorsa  $R_j$  indica che il processo  $P_i$  richiede la risorsa  $R_j$ .
- Un arco diretto dalla risorsa  $R_j$  al processo  $P_i$  indica che il processo  $P_i$  possiede la risorsa  $R_j$ .

La prima tipologia di archi prende il nome di **arco di richiesta**, mentre la seconda tipologia prende il nome di **arco di assegnazione**. Graficamente ogni processo si rappresenta come un cerchio e ogni risorsa si rappresenta come un rettangolo. Giacché il tipo di risorsa  $R_j$  può avere più di una

singola istanza, ciascuna di loro si rappresenta con un puntino all'interno del rettangolo. Questo fatto si traduce che un arco di richiesta deve andare dal processo verso la risorsa, mentre un arco di assegnazione deve andare da una delle istanze della risorsa (quindi uno dei puntini) verso il processo. Quando un processo  $P_j$  richiede un'istanza del tipo di risorsa  $R_i$  è necessario disegnare un arco di richiesta nel grafo, il quale dovrà essere immediatamente trasformato nel momento in cui la risorsa viene assegnata. Un esempio di grafo delle risorse è il seguente

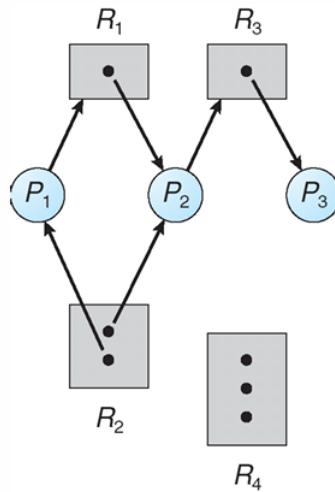


Figura 7.1: Grafo di assegnazione delle risorse. Si nota facilmente che, in questo sistema, non essendo presente alcun ciclo, non è possibile il verificarsi di stalli

Se ogni tipo di risorsa ha una sola istanza, allora l'esistenza di un ciclo è condizione **necessaria** e **sufficiente** all'esistenza di uno stallo. Se però la condizione di singola istanza per ogni risorsa decade, allora l'esistenza del ciclo diventa condizione **necessaria**, ma **non sufficiente** al formarsi di uno stallo. Uno stallo è quindi una condizione che porta a un blocco del sistema che, però non è detto sia totale. Uno stallo, infatti, non riguarda necessariamente l'intero sistema, ma potrebbe riguardare un sottoinsieme dei processi e delle risorse del sistema. Prendiamo come sempio il seguente grafo:

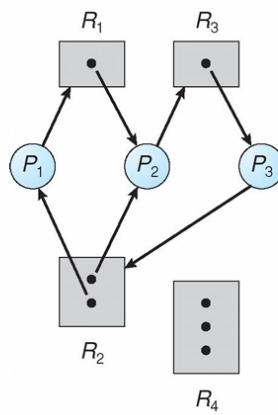


Figura 7.2: Grafo di assegnazione delle risorse

In questo caso è presente un ciclo, pertanto è possibile che si verifichi una situazione di stallo dei processi e delle risorse coinvolte nel ciclo. Però la risorsa  $R_4$  non è bloccata poiché non rientra all'interno del ciclo, di conseguenza, se esistesse un altro processo  $P_4$  nel sistema non coinvolto nel

ciclo sarebbe comunque possibile, per esso, interfacciarsi con  $R_4$  senza tenere conto dello stallo del sistema.

## 2 Metodi per la gestione delle situazioni di stallo

Il problema delle situazioni di stallo richiede che siano definiti degli approcci che permettano di gestirlo o, comunque, di evitarlo del tutto. Possiamo distinguere tre tipologie di approcci

- Si può usare un approccio che prevenga le situazioni di stallo, assicurandosi a priori che il sistema non possa entrarci.
- Si può permettere al sistema di entrare in stallo, individuarlo, e quindi di eseguire il ripristino.
- Si può ignorare il problema, fingendo che le situazioni di stallo non possano proprio verificarsi.

Il primo approccio ha due ulteriori diramazioni. La prima diramazione possibile è quella di **prevenire a priori le situazioni di stallo**, quindi di utilizzare metodi atti ad assicurare che non si verifichi almeno una delle condizioni di stallo. La seconda diramazione possibile è quella di **evitare le situazioni di stallo**, detta anche **deadlock avoidance**, quindi fare in modo che i sistemi operativi abbiano in anticipo delle informazioni sulle risorse che un processo richiederà e userà durante la sua attività. Utilizzando queste informazioni il sistema operativo è in grado di decidere se una richiesta di risorse da parte di un processo può essere soddisfatta o se il processo debba invece attendere. In altre parole, nella **deadlock avoidance**, il sistema operativo tiene conto delle risorse concorrentemente disponibili, di quelle concorrentemente assegnate e delle future richieste e futuri rilasci di ciascun processo.

## 3 Meccanismi di *avoidance* dello stallo

Allo scopo di controllare dinamicamente che non si verifichino mai delle situazioni di stallo è necessario, come anche detto in precedenza, che il sistema operativo conosca a priori delle informazioni. Gli algoritmi differiscono tra loro per quantità e tipo di informazioni richieste; negli algoritmi più elementari l'unica informazione necessaria è il numero di risorse di ciascun tipo di cui il processo ha bisogno.

Affinché sia possibile comprendere questi algoritmi è necessario introdurre il concetto di **stato sicuro**. Uno stato si dice *sicuro* se il sistema è in grado di assegnare risorse a ciascun processo in un certo ordine e impedire il verificarsi di uno stallo. Più formalmente, un sistema è in uno **stato sicuro** se e solo se esiste una **sequenza sicura**, cioè se esiste una sequenza di processi  $P_0, P_1, \dots, P_n$ , tale per cui ogni richiesta di allocazione di un processo  $P_i$  può essere soddisfatta con risorse attualmente disponibili più le risorse possedute da tutti i  $P_j$  con  $i > j$ . In questa situazione, se le risorse necessarie al processo  $P_i$  non sono disponibili immediatamente, allora  $P_i$  può attendere che tutti i  $P_j$  abbiano finito e, a quel punto, ottenere tutte le risorse di cui ha bisogno per completare il compito assegnato e, infine, terminare. Nel momento in cui  $P_i$  termina,  $P_{i+1}$  può ottenere le risorse richieste, e così via. Un sistema per cui non esiste una tale sequenza viene detto, semplicemente, **non sicuro**. Uno **stato sicuro** non è di **stallo**. Viceversa, in un sistema **non sicuro**, si può verificare uno **stallo**. Finché quindi lo stato del sistema rimane sicuro, il sistema operativo può evitare il verificarsi di stati non sicuri e/o di **stallo**. Supponiamo di avere la seguente situazione: un sistema con 12 unità di memoria e 3 processi  $P_1$ ,  $P_2$  e  $P_3$ :

	Richieste massime	Unità possedute
$P_0$	10	5
$P_1$	4	2
$P_2$	9	2

Tabella 7.1: Esempio. Il processo  $P_0$  può richiedere al massimo 10 unità, il processo  $P_2$  può richiedere 4 unità e il processo  $P_3$  può richiedere fino a 9 unità

Supponiamo che, all'istante  $t_0$ , il processo  $P_0$  possiede 5 e che i processi  $P_1$  e  $P_2$  ne possiedono 2 ciascuno. In questo stato è possibile definire una sequenza di esecuzione sicura, infatti è possibile assegnare al processo  $P_1$  tutte le unità di memoria richieste, che saranno poi restituite, quindi il processo  $P_0$  può ottenere tutte le unità di memoria richieste e restituirle e, infine,  $P_2$  potrebbe ottenere tutte le unità di memoria richieste e restituirle. La sequenza sicura è quindi  $P_1, P_0, P_2$ . Si supponga ora che all'istante  $t_1$  il processo  $P_2$  richieda un'ulteriore unità di memoria e che questa gli sia assegnata: il sistema non si trova più nello stato sicuro. A questo punto si possono assegnare tutte le unità a nastri richieste soltanto al processo  $P_1$ . Al momento della restituzione, il sistema avrà solo 4 unità a nastri disponibili. Poiché al processo  $P_0$  sono assegnate 5 unità a nastri, ma il numero massimo è 10, il processo può richiederne altre 5 unità di memoria; se lo fa, poiché queste non sono disponibili, il processo  $P_0$  si deve mettere in attesa. Analogamente, anche il processo  $P_2$  può richiedere altre 6 unità a nastri ed essere costretto ad attendere: il risultato è una situazione di stallo. L'errore è quindi stato l'assegnazione dell'ulteriore nastro al processo  $P_2$ , se  $P_2$  fosse stato costretto ad attendere la fine di uno dei due processi precedenti non si sarebbe verificato.

Dato il concetto di **stato sicuro**, si possono definire algoritmi che permettano di **evitare le situazioni di stallo**. L'idea è semplice: assicurare che il sistema rimanga sempre in uno **stato sicuro**. In particolare, ad ogni richiesta di una risorsa, **anche disponibile**, da parte di un processo, il sistema deve stabilire se la **risorsa** può essere **allocata** o se il processo debba **attendere**.

### 3.1 Algoritmo del banchiere

L'algoritmo con grafo di assegnazione delle risorse non si può applicare ai sistemi di assegnazione con istanze multiple di ciascun tipo di risorsa. L'algoritmo per evitare le situazioni di stallo in questa situazione prende il nome di **algoritmo del banchiere**. All'ingresso del sistema, un nuovo processo, deve dichiarare il numero massimo di istanze di ciascun tipo di risorsa di cui potrebbe aver bisogno. Ad ogni richiesta di assegnazione il sistema deve verificare se, assegnando tali risorse, il sistema rimane in uno stato sicuro:

- Se l'assegnazione mantiene lo stato del sistema sicuro si procede senza problemi.
- Se l'assegnazione porta il sistema in uno stato non sicuro il processo richiedente attende.

La realizzazione dell'algoritmo del banchiere richiede la gestione di alcune strutture dati che codificano lo stato di assegnazione delle risorse del sistema. Supponendo di avere  $n$  processi e  $m$  tipi di risorse, ognuna con un certo numero di istanze, sono necessarie le seguenti strutture dati

Struttura	Descrizione
Disponibili	Un vettore di lunghezza $m$ che indica il numero di istanze disponibili per ciascun tipo di risorsa.
Massimo	Una matrice $n \times m$ che definisce la richiesta massima di ciascun processo.
Assegnate	Un matrice $n \times m$ che definisce il numero delle istanze di ciascun tipo di risorse assegnate a ogni processo.
Necessità	Una matrice $n \times m$ che indica la necessità residua di risorse relativa ad ogni processo.

Tabella 7.2: Tabella delle opzioni per il comando `cd`

Ovviamente la matrice di necessità può essere popolata, per ogni coppia  $(i, j)$ , facendo la differenza tra il massimo di risorse, per quel tipo, di cui il processo ha bisogno e quante risorse, di quel tipo, gli sono state assegnate fino ad ora. Il primo algoritmo che analizziamo è quello che permette di controllare se il sistema è o meno in uno stato sicuro. L'algoritmo segue i seguenti passi:

1. Siano **Lavoro** e **Fine** vettori di lunghezza rispettivamente  $m$  e  $n$ , si inizializza **Lavoro = Disponibili** e **Fine[i] = false**, per ogni  $i = 1, 2, \dots, n - 1$ .
2. Si cerca un indice  $i$  tale per cui valgano contemporaneamente due condizioni:

- $\text{Fine}[i] == \text{falso}$ .
- $\text{Necessità}_i \leq \text{Lavoro}$ .

Se tale  $i$  non esiste, allora si esegue direttamente il passo 4.

3.  $\text{Lavoro} = \text{Lavoro} + \text{Assegnate}_i$ ,  $\text{Fine}[i] = \text{true}$  e si torna al passo 2.

4. Se  $\text{Fine}[i] == \text{vero}$  per ogni possibile  $i$ , allora il sistema è in uno stato sicuro.

Il primo passo dell'algoritmo è quello di definire le risorse che sono attualmente disponibili attraverso il vettore **Lavoro**; non utilizziamo il vettore reale poiché stiamo facendo una simulazione per capire se lo stato è sicuro. Il secondo passo è quello di definire un vettore per identificare se un processo è terminato o meno; questo vettore è chiamato **Fine**. Successivamente è possibile cercare un processo candidato che possa essere completato. Per farlo dobbiamo verificare che non sia già terminato ( $\text{Fine}[i] == \text{falso}$ ) e per cui la richiesta di una particolare risorsa sia soddisfacibile con le risorse attualmente disponibili ( $\text{Necessità}_i \leq \text{Lavoro}$ ). Se questo candidato non viene trovato si passa alla fine. Se troviamo invece un processo candidato, "facciamo finta" di assegnargli le risorse e il processo termina ( $\text{Fine}[i] = \text{true}$ ), rendendo nuovamente disponibili le risorse che aveva accumulato ( $\text{Lavoro} = \text{Lavoro} + \text{Assegnate}_i$ ). Se alla fine del ciclo tutti i processi sono terminati, allora lo stato è **sicuro**.

Questo algoritmo non è propriamente l'algoritmo del banchiere, ma permette solo di capire se uno stato è sicuro o meno. L'algoritmo del banchiere è leggermente più articolato. Quando si riceve una richiesta da un processo si deve controllare due cose

- Se la richiesta è coerente con quanto dichiarato dal processo all'inizio, cioè se il processo non richiede più di quanto aveva dichiarato all'inizio.

$$\text{Richieste}[i] \leq \text{Necessità}[i]$$

- Se la richiesta può essere soddisfatta, cioè se è possibile assegnare quanto richiesto rispetto a quanto è ancora disponibile.

$$\text{Richieste}[i] \leq \text{Disponibili}[i]$$

Nel caso in cui la richiesta non possa essere soddisfatta il processo richiedente si sospende in attesa. Nel caso in cui la richiesta possa essere soddisfatta si simula l'assegnazione al processo  $P_i$  delle risorse richieste modificando lo stato di assegnazione delle risorse:

$$\begin{aligned} \text{Disponibili} &= \text{Disponibili} - \text{Richieste}[i] \\ \text{Assegnate}[i] &= \text{Assegnate}[i] + \text{Richieste}[i] \\ \text{Necessità}[i] &= \text{Necessità}[i] - \text{Richieste}[i] \end{aligned}$$

Una volta aggiornato lo stato di assegnazione delle risorse è necessario verificare se tale stato è sicuro, per farlo si utilizza l'algoritmo che abbiamo visto poche righe sopra. Nel caso in cui lo stato sia sicuro allora la transazione è completata e al processo  $P_i$  si assegnano le risorse richieste. Tuttavia, se il nuovo stato non è sicuro  $P_i$  deve attendere e si ripristina il vecchio stato di assegnazione delle risorse.

## 4 Rilevamento delle situazioni di stallo

Se un sistema non si avvale di un algoritmo che prevenga a priori il verificarsi delle situazioni di stallo è possibile sfruttare un approccio alternativo. Invece che evitare le situazioni di stallo, si risolvono quando si verificano. In un ambiente di questo genere, il sistema può fornire i seguenti algoritmi:

- Un algoritmo che esamini lo stato del sistema per capire se si è verificata una situazione di stallo.
- Un algoritmo che ripristini il sistema dalla condizione di stallo.

Questo approccio, cioè con uno schema di rilevamento e ripristino, richiede un overhead che include sia i costi per la memorizzazione delle informazioni necessarie e per l'esecuzione dell'algoritmo di rilevamento, ma anche i potenziali costi dovuti alle perdite di informazioni connesse al ripristino delle situazioni di stallo.

Se tutte le risorse hanno una sola istanza si può definire un algoritmo di rilevamento che fa uso di una variante del grafo di assegnazione delle risorse, detto **grafo di attesa**. Questo grafo si ottiene partendo dal grafo delle risorse togliendo i nodi dei tipi di risorse e componendo gli archi tra processi. Un arco da  $P_i$  a  $P_j$  del grafo di attesa implica che il processo  $P_i$  attende che il processo  $P_j$  rilasci una risorsa di cui  $P_i$  ha bisogno. Un arco  $P_i \rightarrow P_j$  esiste nel grafo di attesa se e solo se il corrispondente grafo di assegnazione delle risorse contiene due archi  $P_i \rightarrow R_q$  e  $R_q \rightarrow P_j$ . Prendiamo come esempio i seguenti grafi

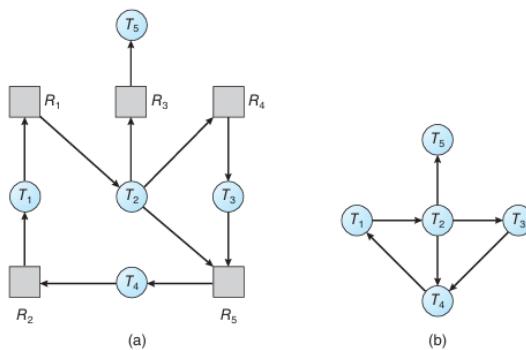


Figura 7.3: Esempio di corrispondenza tra **grafo di assegnamento delle risorse** e **grafo di attesa**

Questo algoritmo non sarà eseguito costantemente, ma andrà temporizzato in modo che l'overhead introdotto non sia eccessivo. Un'idea potrebbe essere quella di eseguire l'algoritmo, in modo **preventivo**, ogni volta che un processo fa una **wait** per richiedere l'allocazione di una risorsa. Questo approccio introduce però due problemi:

- **Overhead.** I processi sovente richiedono delle risorse e, quindi, eseguire l'algoritmo in ognuna di queste circostante consumerebbe molto tempo.
- **Corrispondenza con l'avoidance.** Il meccanismo di funzionamento è del tutto analogo a quello della deadlock avoidance vista nella sezione precedente.

Una soluzione opposta potrebbe essere quella di osservare alcuni parametri del sistema, come il tempo di esecuzione della CPU. Nel momento in cui l'occupazione della CPU scende sotto un certo livello potrei supporre che il sistema sia in deadlock e quindi eseguire l'algoritmo di deadlock detection. La via intermedia tra le due opzioni potrebbe essere quella di eseguire l'algoritmo periodicamente con degli intervalli di tempo decisi in funzione di un analisi statistica del verificarsi dei deadlock. Se ad esempio dall'analisi venisse fuori che il deadlock si verifica una volta ogni ora, potrei eseguire l'algoritmo ogni ora.

Supponiamo ora di avere delle istanze multiple per ogni tipo di risorse. In questo caso in grafo di attesa non può essere sfruttato e occorre quindi definire un nuovo algoritmo per identificare le situazioni di stallo. Analogamente al caso della deadlock avoidance, occorre utilizzare delle strutture dati variabili nel tempo

- **Disponibili.** Vettore di lunghezza  $m$  che indica il numero delle istanze disponibili per ciascun tipo di risorsa.
- **Assegnate.** Matrice  $n \times m$  che definisce il numero delle istanze di ciascun tipo di risorse assegnate a ciascun processo.
- **Richieste.** Matrice  $n \times m$  che indica la richiesta attuale di ciascun processo

In questo caso il numero di richieste non è definito come la differenza tra il massimo che un processo può richiederne e quante gliene sono allocate. Il motivo è presto detto, questo algoritmo non conosce alcuna informazione a priori e, quindi, non conosce quante risorse potrebbe richiedere un processo. La matrice delle richieste ha quindi la funzione di indicare quali processi sono bloccati e su quale risorsa lo sono. Un possibile algoritmo di detection potrebbe essere

1. Creiamo due strutture dati di lavoro, cioè due vettori `lavoro`, che rappresenta quante risorse sono attualmente disponibili, mentre `fine` è un vettore che codifica quanti processi sono terminati. In particolare, vale la seguente condizione

$$\text{Fine}[i] = \text{False} \iff \text{Assegnate}[i] \neq 0$$

L'algoritmo imposta quindi `Fine[i] = Vero` per tutti i processi che hanno zero risorse assegnate, infatti se un processo non possiede alcuna risorsa allora non può sicuramente fare parte di alcun ciclo di stallo.

2. Si cerca un indice per cui valgano due condizioni: la prima condizione `Fine[i] == Falso` indica che il processo non deve essere terminato, mentre la seconda condizione `Richieste[i] ≤ Lavoro`, indica che la richiesta del processo può essere soddisfatta. Se tale indice non esiste allora si esegue il passo 4.
3. Supponiamo che  $P_i$  non sia coinvolto in uno stallo. Di conseguenza, ottimisticamente parlando, è possibile ipotizzare che il processo  $P_i$  non intenda chiedere ulteriori risorse per completare il proprio compito, e che restituisca presto tutte le risorse. Possiamo quindi immaginare di liberare le risorse assegnate e segnare quel processo come terminato:

$$\begin{aligned} \text{Lavoro} &= \text{Lavoro} + \text{Assegnate}[i] \\ \text{Fine}[i] &= \text{vero} \end{aligned}$$

si torna al passo 2 e si ripete per ogni singolo processo. Se per qualche motivo l'ipotesi di terminazione del processo fosse sbagliata, non sarebbe un problema, lo stallo verrebbe rilevato alla successiva esecuzione dell'algoritmo.

4. Se alla fine delle iterazioni esiste ancora un processo  $P_i$  tale per cui `Fine[i] == Falso` allora il sistema è in stallo e, in particolare,  $P_i$  è in stallo. Infatti vuol dire che anche terminando tutti i processi presenti all'interno del sistema, non è possibile soddisfare la richiesta del processo  $i$ -esimo.

Una volta identificato lo stallo è necessario anche risolverlo. Esistono, a tal proposito, diverse possibilità implementative con cui si esce da una situazione di stallo:

- Terminazione dei processi.
- Prelazione delle risorse.

### Terminazione dei processi

Affinché sia possibile gestire le situazioni di stallo è possibile usare un approccio diretto, terminando uno o più processi in stallo; in particolare

- **Terminazione di tutti i processi in stallo.** Si interrompe il ciclo di stallo, con un costo computazionale molto oneroso; infatti questi processi possono aver già fatto moltissimi calcoli i cui risultati verranno scartati costringendo ad un ricalcolo totale.
- **Terminazione di un processo alla volta fino all'eliminazione dello stallo.** Questo metodo, nonostante diminuisca il numero di processi terminati, costringe ad un iterazione che aggiunge un overhead considerevole nel sistema, poiché costringe a invocare iterativamente l'algoritmo di rilevamento dello stallo.

Se poi il processo da terminare si trovasse nel mezzo dell'aggiornamento di un file, la terminazione lascerebbe il processo in uno stato inconsistente. Inoltre, utilizzando l'eliminazione parziale dei processi è necessario anche definire un criterio di scelta sulla base del quale viene terminato il processo. In generale si dovrebbero arrestare i processi la cui terminazione causa il **minimo costo**. Tuttavia, il termine **minimo costo** non ha una caratterizzazione esatta, ma è influenzato da diverse possibili scelte di metriche: **la priorità del processo, il tempo trascorso in computazione, la quantità di risorse impiegate dal processo, etc.**

### Prelazione delle risorse

Per eliminare uno stallo si può esercitare anche la **prelazione** sulle risorse del sistema: si sottraggono le risorse in successione ad alcuni processi e si assegnano ad altri fintanto che non si elimina lo stallo. Questa soluzione introduce però tre problemi:

- **Selezione di una vittima.**
- **Ristabilimento di un precedente stato sicuro.**
- **Attesa indefinita.**

## 5 Prevenzioni delle situazioni di stallo

Per prevenire la situazione di stallo è necessario che almeno una delle quattro condizioni affinché si verifichi lo stallo siano rimosse dal sistema. La prima condizione, cioè quella della **mutua esclusione** è forse quella più complessa da rimuovere, infatti per alcune risorse è necessario che l'accesso sia garantito in mutua esclusione e, quindi, non è possibile rimuovere del tutto questa condizione. Tuttavia è possibile limitarla andando a restringere il campo di utilizzo dell'accesso in mutua esclusione alle sole risorse che hanno strettamente bisogno. La seconda condizione, cioè quella del possesso e dell'attesa è risolvibile in due modi diversi:

- Si obbliga il processo a richiedere tutte le risorse prima di iniziare l'esecuzione.
- Si vincola il processo a dover rilasciare tutte le risorse che gli sono state allocate prima di richiederne di nuove.

La terza condizione può essere risolta facendo in modo che se una richiesta di un processo non può essere soddisfatta, allora tutte le risorse allocate vengono revocate e il processo viene messo in attesa. L'ultima condizione può essere rimossa definendo un ordinamento delle risorse tale per cui ogni processo può richidere solo risorse maggiori, nell'ordine di enumerazione, di quella più piccola, sempre nell'ordine, che possiede.

## **Parte III**

# **Aspetti architetturali sui sistemi operativi**

# 8

## Gestione della memoria

Uno dei componenti fondamentali all'interno di un calcolatore è la **memoria**. La **memoria centrale** è quel componente che permette ai diversi processi di un sistema di contenere i dati, tra cui il descrittore di processo e il codice stesso, per un tempo ampio e in uno stato sicuro. Quindi, affinché un programma possa essere in esecuzione, è necessario che siano disponibili due risorse fondamentali:

- Una unità di elaborazione che esegue effettivamente il programma.
- Un'area di memoria principale dove risiedono i programmi.

Partendo da questa semplice osservazione è possibile identificare tutta una serie di analogie tra la gestione delle risorse e quelle della memoria. Al contempo però, vista la sostanziale differenza nella natura delle due componenti è anche ovvio che ci siano diverse differenze.

La prima analogia deriva dal fatto che entrambe le risorse hanno un funzionamento diametralmente opposto a quello di una tradizionale risorsa per cui si adotta un approccio di **allocazione non revocabile**. Infatti un processo non potrebbe richiedere la CPU, poiché per richiedere una risorsa il processo deve trovarsi in esecuzione e, quindi, dovrebbe già avere la CPU; stessa cosa la possiamo dire anche della memoria, infatti affinché un processo possa richiedere della memoria è necessario che sia in esecuzione e quindi che le sue strutture dati siano caricate in memoria. Queste considerazioni portano a ritenere che a ogni processo debbano necessariamente essere biunivocamente associate le due risorse che sono necessarie per la sua stessa esistenza: **processore e memoria**. Ciò crea un'appartenente contraddizione, se il numero di processi è maggiore del numero dei processori, allora, non è possibile permettere l'associazione biunivoca tra processore e processo. Per risolvere tale problema è necessario applicare la tecnica della virtualizzazione della CPU, cioè ipotizzare di dare ad ogni processo un processore virtuale; tale tecnica è applicabile anche alla memoria. Anche in questo caso ad ogni processo viene associata una memoria virtuale nella quale viene allocato il programma che il processo deve eseguire.

La prima differenza nella gestione tra le due componenti deriva proprio dalla loro diversa organizzazione nella **struttura fisica**, che implica anche una diversa organizzazione nella **struttura virtuale**. Una **risorsa virtuale** non è altro che una **struttura dati** che deve rappresentare lo **stato della risorsa fisica** quando questa non è allocata al corrispondente processo. In questo senso la **CPU virtuale** è costituita da una struttura dati allocata nella **memoria del sistema**. Ovviamente ciò non potrà essere vero per la memoria virtuale, che invece risiede in un area del disco fisico, detta **swap-area**, dedicata a contenere le memorie virtuali di tutti i processi che non sono allocati in memoria principale. La seconda differenza riguarda il fatto che, mentre la CPU non è condivisibile tra più processi in contemporanea, la memoria principale è in grado di ospitare più processi allocati contemporaneamente. La condivisione della memoria crea un problema di protezione, infatti è necessario evitare che la memoria virtuale di un processo possa sovrapporsi alla memoria virtuale di un altro processo quando non si vuole che ciò avvenga. La protezione non implica quindi che due aree di memorie non possano mai sovrapporsi, ma che non lo facciano compromettendo la correttezza dei dati, poiché potrebbe capitare che due processi debbano eseguire

la stessa porzione di codice, condividendo l'area di memoria ad esso associata. Questo fenomeno di condivisione prende il nome di **sharing** codice.

## 1 Implementazione della memoria

La memoria virtuale di un processo è l'insieme di tutte le locazioni di memoria e delle informazioni in esse contenute i cui indirizzi possono essere generati dalla CPU durante l'esecuzione del processo. Quindi, un processo in esecuzione può generare solo indirizzi della propria memoria virtuale e, di fatto, la memoria virtuale rappresenta la **base di conoscenza** del nostro processo, cioè tutto quello che il processo conosce; più nello specifico, all'interno di questo spazio possiamo trovare:

- **Codice.**
- **Dati.**
- **Stack.**
- **Heap.**

Possiamo anche definire la memoria virtuale di un processo come la porzione di memoria fisica da esso utilizzata a partire dalla locazione fisica di indirizzo zero e quindi nell'ipotesi che nessun altro processo sia contemporaneamente allocato in memoria, ne consegue che la struttura della memoria virtuale coincide con il contenuto del **file eseguibile**, cioè del file in formato binario che contiene tutte le informazioni relative al programma da eseguire.

Uno degli indirizzi all'interno di questo spazio è detto **entry point** ed è l'indirizzo della prima istruzione da eseguire quando il processo viene caricato per la prima volta in memoria. Questi dati, che rappresentano l'**immagine del processo** e risiedono in memoria permanente, devono essere caricati all'interno della memoria centrale una volta che il processo deve essere messo in esecuzione. Il caricamento dei processi dalla memoria permanente alla memoria centrale crea due problemi:

- Non è definito uno standard sulla dimensione di questa area di memoria e, di conseguenza, è necessario calcolarne la grandezza per capire se c'è sufficiente spazio in memoria centrale.
- Deve esistere un meccanismo che esegua fisicamente lo scambio ogni volta che un processo deve essere caricato o rimosso dalla memoria.

## 2 Rilocazione della memoria

Uno dei problemi principali della memoria riguarda la sua rilocazione, cioè l'assegnazione della memoria ad un processo che si trova nella **swap-area**, la quale può coincidere anche con la revoca della memoria ad un processo che già risiedeva nella memoria principale; queste due fasi prendono rispettivamente il nome di **swap-in** e di **swap-out**. L'idea è che, in generale, l'area di memoria assegnata ad un processo dovrà essere sufficientemente grande da contenere l'intera memoria virtuale di un processo a partire da un qualunque indirizzo  $I$  diverso da zero. La memoria virtuale, per definizione, ha indirizzi uguali per ogni processo, cioè dall'indirizzo zero fino all'indirizzo  $n$ . Nella memoria fisica è però necessario che l'area di memoria di ogni processo si trovi ad un indirizzo diverso; a tal proposito possiamo quindi distinguere

- L'**indirizzo virtuale** di un processo, che corrisponde all'indirizzo di una locazione all'interno dello spazio di memoria virtuale del processo.
- L'**indirizzo fisico** di un processo, che corrisponde all'indirizzo di una locazione di memoria all'interno della memoria centrale.

I due spazi di indirizzamento non sono quindi indipendenti, ma sono legati da una particolare funzione, detta **funzione di rilocazione**, che ad ogni indirizzo fisico ne associa uno virtuale:

$$f : \text{Indirizzo Virtuale} \rightarrow \text{Indirizzo Fisico}$$

La funzione è, matematicamente, una somma che associa ad un indirizzo virtuale uno **spiazzamento**, che identifica la posizione del primo indirizzo dell'area di memoria virtuale di quel processo all'interno della memoria fisica. La rilocazione dell'area di memoria di un processo può essere fatta in due modi diversi, nel primo caso, cioè quello della rilocazione statica, si rilocalano tutti gli indirizzi da virtuali a fisici prima che il processo inizi la sua esecuzione (durante la fase di caricamento) utilizzando un dispositivo detto **caricatore rilocante**, mentre nella **rilocazione dinamica** si deve utilizzare un supporto hardware, detto **MMU**, o **Memory Management Unit**, che traduce gli indirizzi virtuali in fisici man mano che vengono richiesti dal processo in esecuzione. Quindi, in sostanza:

- Nella **rilocazione statica** gli indirizzi virtuali vengono tradotti a monte in indirizzi fisici.
- Nella **rilocazione dinamica** gli indirizzi virtuali vengono tradotti dinamicamente e quando richiesto in indirizzi fisici,

La **rilocazione statica** per quanto semplice è anche molto limitativa in quanto, una volta caricato in memoria un programma, questo non può più essere rilocato in una diversa porzione di memoria. Supponiamo, ad esempio, che la memoria assegnata a un processo debba essere revocata per fare spazio ad altri processi. In questo caso l'immagine del processo viene trasferita nella **swap-area** su memoria di massa da cui, successivamente, verrà recuperata e ricaricata in memoria. Ebbene, la scelta di aver rilocato staticamente in memoria il processo, e cioè di aver definito il suo spazio degli indirizzi fisici prima dell'inizio della sua esecuzione, non consente all'atto dell'operazione di **swap-in** di scegliere una diversa porzione di memoria in cui riallocare il processo. Come possiamo facilmente dimostrare, questo vincolo limita fortemente la flessibilità con cui la memoria può essere allocata ai processi.

La **rilocazione dinamica** sfrutta un dispositivo a livello hardware, chiamato **Memory Management Unit**, che si occupa di prendere in ingresso gli indirizzi virtuali generati dalla CPU e di restituire in uscita gli indirizzi fisici per l'accesso alla memoria. L'MMU, che si pone tra la memoria e la CPU, è quindi in grado di implementare la funzione di **rilocazione** in modo dinamico man mano che gli indirizzi virtuali vengono generati. Anche questo meccanismo però non è esente da varianti; la versione più semplice è quella che sfrutta due registri:

- Registro base.
- Registro limite.

Questi due registri, facenti parte dell'MMU, sono predisposti a contenere, rispettivamente, l'indirizzo fisico iniziale a la dimensione della partizione di memoria nella quale è caricata l'immagine del processo. Ogni volta che la CPU genera un indirizzo virtuale  $x$ , questo viene sommato al contenuto del registro base al fine di ottenere il corrispondente indirizzo fisico  $y$ . L'indirizzo virtuale  $x$  viene anche confrontato con il valore del registro limite  $e$ , se e solo se,  $x$  è inferiore o uguale al valore del registro limite l'indirizzo fisico  $y$  viene inviato alla memoria fisica per accedere all'informazione desiderata. Se invece  $x$  supera il valore del registro limite, viene generata un'interruzione per segnalare che si è usciti dallo spazio di memoria assegnato.

### 3 Segmentazione nella memoria virtuale

Nella precedente sezione abbiamo ipotizzato che la memoria venga allocato in modo contiguo, cioè che lo spazio di indirizzamento virtuale di ogni processo venga fatto corrispondere a indirizzi fisici contigui nello spazio di indirizzamento fisico. Affinché però sia possibile applicare questa idea è necessario che la memoria di un processo venga divisa in delle sezioni più piccole, dette segmenti, che possono essere allocate in parti di memoria fisica anche non contigui tra di loro. Infatti, se lo spazio virtuale è segmentato, ogni segmento può essere rilocato in memoria fisica indipendentemente dagli altri. Questa tecnica porta con sé due vantaggi principali:

- **Maggiore facilità nel caricamento.** Non essendo più richiesto di avere delle locazioni contigue di memoria fisica diventa anche quindi più semplice l'allocazione della memoria virtuale.

- **Protezione e condivisione.** È possibile associare selettivamente ad ogni segmento dei diritti di accesso che caratterizzano le modalità con cui è possibile accedere correttamente alle informazioni del segmento.

Uno dei vantaggi principali della possibilità di allocare i segmenti in porzioni di memoria fisica non contigua deriva da un fatto non trascurabile, ipotizziamo per ora che l'allocazione dei segmenti debba comunque avvenire in modo contiguo. Se, per qualche motivo, la memoria fisica avesse sufficiente spazio libero, ma non fosse possibile allocare contiguamente i segmenti dello spazio di memoria di un processo, allora si dovrebbe procedere con il **compattamento** della memoria, andando a compattare le varie aree di memoria occupate fintanto che l'area contigua complessivamente libera non è sufficiente. La compattazione introduce però un costo molto elevato in termini di complessità computazionale per ottenere un'area di memoria sufficientemente libera. L'unica soluzione per evitare l'inconveniente del compattamento è quella di permettere l'allocazione di uno spazio virtuale contiguo in locazioni di memoria fisica non necessariamente contigue.

## 4 Tecniche di gestione della memoria

Abbiamo visto, nelle scorse sezioni, che esistono quattro caratteristiche fondamentali che permettono di caratterizzare un meccanismo di gestione della memoria. Queste quattro caratteristiche dipendono sia dall'architettura del processore stesso, sia dalle scelte implementative che vengono fatte a livello di sistema operativo; in particolare, possiamo distinguere: A ciascuna combinazione

Parametro	Valori
Rilocazione	Statica o Dinamica.
Spazio Virtuale	Unico o Segmentato.
Allocazione memoria fisica	Contigua o Non contigua.
Caricamento Spazio virtuale	Tutto insieme o Su domanda

Tabella 8.1: Parametri per la definizione di meccanismi per la gestione della memoria

dei differenti parametri corrisponde un diverso meccanismo e una diversa tecnica di gestione della memoria. Ovviamente, alcune di queste combinazioni di parametri non sono significative, prendiamo solo il caso della rilocazione statica, l'allocazione della memoria fisica è sempre effettuata in modo contiguo e le dimensioni dello spazio virtuale non possono superare quelle della memoria fisica. Oppure, nel caso della **rilocazione dinamica** e **allocazione contigua di memoria** lo spazio virtuale è sempre segmentato. Infatti, la scelta di realizzare uno spazio virtuale unico non permetterebbe nessuna forma di condivisione di informazioni tra processi diversi.

### 4.1 Memoria partizionata

La prima classe di tecniche per la gestione della memoria è quella delle **partizioni**. Questa tecnica sfrutta il meccanismo della **rilocazione statica**, non prevede alcuna **segmentazione della memoria fisica** del processo, utilizza uno spazio virtuale unico, cioè ad ogni processo verrà assegnato un particolare indirizzo di base all'interno della memoria virtuale (suddividendo lo spazio di indirizzamento virtuale), e prevede che lo spazio virtuale di un particolare venga caricato tutto insieme. Possiamo distinguere due tecniche di gestione della memoria a partizioni:

- Partizioni fisse.
- Partizioni variabili.

In questa tecnica, nel momento in cui è necessario allocare memoria ad un processo, dovendo necessariamente caricare tutto lo spazio virtuale del processo in un'area di memoria che sia sufficientemente grande, è necessario cercare una **partizione** di memoria che rispetti il requisito sulla dimensione. Ipotizzato che venga trova questa partizione, il caricatore farà una rilocazione dello

spazio virtuale del processo all'interno della partizione, imponendo anche due condizioni: la **partizione di memoria** dovrà rimanere assegnata al processo **fintanto che non termina** e, nel caso in cui un processo subisse una **swap-out**, il **rilocatore** dovrà essere in grado di **rilocarlo**, in un momento futuro, all'interno della stessa partizione di memoria.

**Gestione delle partizioni libere** Una delle problematiche relative ad uno schema a partizionamento della memoria è mantenere aggiornate le informazioni su quali sono le partizioni libere, la loro dimensione e l'indirizzo base a cui si trovano. Per farlo, il **gestore della memoria**, mantiene aggiornata una struttura dati con la quale registrare, istante per istante, quante e quali sono le partizioni disponibili. Una possibile soluzione a questo problema, che viene spesso utilizzata, è quella di mantenere aggiornata una lista i cui elementi rappresentano le caratteristiche di ogni partizione libera. Definiamo quindi come **memoria.libera** una locazione contenente l'indirizzo di una partizione libera. Ogni partizione liberà conterrà poi, nelle prime due locazioni, una variabile **dimensione**, che specifica la dimensione della partizione, e una variabile **indirizzo**, che specifica l'indirizzo della successiva partizione libera.

**Scelta della partizione** L'ordinamento delle partizioni può essere fatto mediante due politiche: **first-fit** e **best-fit**. Nella politica **first-fit** si mantiene una lista ordinata per indirizzi crescenti delle partizioni. In fase di richiesta, quindi, la prima partizione che viene trovata in grado di ospitare l'immagine del processo verrà rilocata. Mentre nella politica **best-fit** la lista delle partizioni viene mantenuta ordinata per dimensioni crescenti delle partizioni e, in fase di richiesta di una partizione di almeno  $N$  byte, la lista viene scandita e la prima partizione che viene trovata in grado di soddisfare la richiesta è sicuramente quella più piccola tra tutte quelle di dimensioni superiori a  $N$ .

### Tecnica delle partizioni fisse

Nello schema più semplice, cioè quello a **partizioni fisse**, è stato scelto che la memoria venga suddivisa in  $(n + 1)$  partizioni, ognuna con un particolare **indirizzo iniziale** e **dimensioni fisse**, definite in fase di installazioni del **sistema operativo**; si presti però attenzione al significato di **partizione fissa**, infatti non è detto che le partizioni siano tutte della stessa dimensione, è possibile infatti che partizioni diverse abbiano dimensioni diversi, tuttavia queste dimensioni saranno invariabili nel tempo. Le partizioni saranno a loro volta divise in

- Una partizione destinata a contenere la parte residente del sistema operativo.
- $n$  partizioni destinate a contenere le immagini di altrettanti processi una volta **rilocati** i rispettivi spazi virtuali.

Tuttavia è necessario prestare attenzione ad un fatto: è raro che l'immagine di un processo sia esattamente della stessa dimensione di una partizione di memoria. Questa caratteristica crea un problema non indifferente; se definiamo infatti come  $D_i$  come la **dimensione della partizione  $i$ -esima** e come  $N_i$  la dimensione dell'immagine del processo  $i$ -esimo, possiamo definire come **spazio libero lasciato da ogni processo** la differenza tra  $D_i - N_i$ . Ipotizzando di avere  $n$  processi all'interno del nostro sistema, possiamo definire lo spazio libero totale come

$$\text{Spazio Libero} = (D_0 - N_0) + (D_1 - N_1) + \cdots + (D_n - N_n) = \boxed{\sum_{i=1}^n (D_i - N_i)}$$

La presenza di questo spazio **inutilizzato** crea un grandissimo problema di inefficienza e mancanza di flessibilità all'interno di un sistema; problema chiamato **frammentazione interna**. Per quanto possibile, un sistema operativo cerca di minimizzare questo problema, cercando di allocare i processi all'interno della partizione più piccola in grado di contenerli. Quindi, concludendo, uno schema a partizione fissa, nonostante quindi introduca un overhead bassissimo sul sistema, è comunque molto inefficiente per quanto riguarda l'uso della memoria. Infatti, la scelta di definire il numero e le dimensioni delle partizioni una volta per tutte in fase di installazione del sistema implica una totale mancanza di flessibilità.

## Tecnica delle partizioni variabili

Per risolvere il problema dell'inefficienza delle partizioni variabili, è possibile implementare uno schema di partizionamento più flessibile, in cui le partizioni non abbiano dimensioni definite a priori, ma siano definite al momento dello **swap.in** del processo. In altri termini, le caratteristiche delle singole partizioni in modo tale che queste corrispondano esattamente alle esigenze di memoria dei processi. Prendiamo come esempio la seguente situazione:

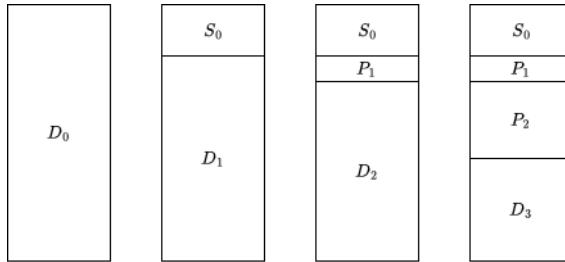


Figura 8.1: Organizzazione della memoria paginata con partizioni variabili

Inizialmente tutta la memoria costituisce un'unica partizione grande  $D_0$  Mbyte. La prima partizione che viene creata sarà, ovviamente, quella che ospita il sistema operativo, grande  $S_0$  byte. A questo punto, come si osserva in figura 8.1, si hanno quindi due partizioni, una grande  $S_0$  Mbyte e una grande  $D_1$  Mbyte tale per cui:

$$D_1 = D - S_0$$

Ipotizziamo che all'istante  $t = 1$  venga creato il primo processo  $P_1$  con un area di memoria pari a  $P_1$  Mbyte. A questo punto viene creata una partizione pari a  $P_1$  Mbyte dove il processo viene allocato, ottenendo quindi un sistema con tre diverse partizioni: la partizione  $S_0$ , la partizione  $P_1$  e una partizione grande  $D_2$  Mbyte tale per cui

$$D_2 = D_1 - P_1 = D - S_0 - P_1$$

All'istante  $t = n$  ci saranno  $n$  processi all'interno del sistema, ognuno che occupa un'area di memoria pari a  $P_i$  Mbyte. A questo punto si ottiene quindi una partizione di memoria libera  $D_{n+1}$  di dimensioni pari a

$$D_{n+1} = D_1 - \sum_{i=1}^n N_i$$

Nel momento in cui un processo termina, lascerà ovviamente una partizione libera di dimensione pari alla dimensione dell'immagine del processo all'interno della memoria. Nel momento in cui ciò accade però, se la partizione liberata è adiacente ad altre partizioni libere è necessario **fonderle** in una sola nuova partizione di dimensione pari alla somma di quelle compattate. Utilizzando il meccanismo a **partizioni variabili**, quindi, si risolve il problema della frammentazione interna, con un utilizzo molto più flessibile della memoria del sistema, in quanto ad ogni processo viene allocata solo la memoria necessaria per contenere la sua immagine. Tuttavia, anche l'approccio a partizioni variabili non è esente da problematiche, infatti che alcuni dei processi allocati potrebbero terminare e altri potrebbero essere creati, generando quindi uno scenario in cui la memoria viene dinamicamente suddivisa in partizioni libere intervallate da aree occupate dai processi allocati in memoria; questa frammentazione della memoria prende il nome di **frammentazione esterna**. Questa situazione spiacevole porta ad avere una memoria a *buchi*, con delle partizioni libere sparse e intervallate ad altre partizioni occupate dai processi. L'unica soluzione possibile per risolvere questa situazione è quella di **compattare** la memoria utilizzando una serie di **swap-out** e di **swap-in**. Il compattamento non è però un'operazione che è possibile fare con così tanta semplicità, poiché ci troviamo in un contesto di **rilocazione statica**.

## Tecnica delle partizioni multiple

Uno dei problemi delle tecniche a partizionamento che abbiamo visto fino ad ora è la poca flessibilità in termini di condivisione delle risorse; potrebbe infatti capitare una situazione in cui due processi

condividono il codice, eseguendolo su dati diversi. Utilizzando una tecnica a partizioni fisse o variabili sarebbe necessario replicare il codice per entrambi i processi, in quanto le memorie virtuali di ogni processo sono indipendenti tra di loro. Una tecnica che permette di risolvere questo problema è la tecnica a **partizioni multiple**: questa tecnica permette di segmentare lo spazio virtuale, abilitando il linker a suddividerlo, ad esempio, nei tre segmenti **codice**, **stack** e **dati**. In questo modo, ogni segmento, pur essendo ancora singolarmente allocato in locazioni contigue, e cioè in una partizione di memoria, può essere rilocato in maniera indipendente dalle partizioni usate per allocare gli altri segmenti. Quindi, utilizzando uno schema di questo tipo non è più necessario cercare un'unica grande partizione di memoria, è sufficiente cercarne tre più piccole in grado di ospitare le tre partizioni. Questa tecnica utilizza una rilocazione statica degli indirizzi, con uno spazio virtuale segmentato e un'allocazione della memoria fisica contigua.

## 4.2 Memoria segmentata

Alla base del funzionamento della memoria segmentata troviamo l'idea che lo spazio di indirizzamento virtuale di ogni processo sia diviso in **segmenti**. Questi segmenti possono essere di dimensioni variabili, ognuno di essi identificato da un particolare **indirizzo base del segmento**. Quindi, una particolare locazione all'interno di un segmento può essere quindi identificata attraverso una coppia di valori

$$x = (\text{Indirizzo base del segmento}, \text{Offset della locazione})$$

All'interno della memoria segmentata la rilocazione è dinamica, di conseguenza è necessario che venga implementato un meccanismo di traduzione tra indirizzi fisici e indirizzi virtuali. Inoltre, è anche necessario che venga introdotto un dispositivo hardware in grado di operare tali traduzioni; questo dispositivo è detto **Memory Management Unit**. Inoltre, siccome lo spazio virtuale è segmentato e potrebbe essere richiesto che i processi possano condividere tra di loro il **segmento** contenente il codice e, quindi, ogni segmento deve essere dinamicamente rilocato in modo indipendente dagli altri. La MMU non deve quindi contenere una sola coppia, ma tre coppie di **registri base** e **registri limite** (supponendo di segmentare la memoria virtuale in tre segmenti), una per ogni segmento che compone lo spazio virtuale del processo attualmente in esecuzione; l'MMU sarà quindi utilizzata per tradurre ogni **indirizzo virtuale** appartenente a quel **segmento** nell'indirizzo fisico corrispondente. Ad esempio, l'MMU nel caso della suddivisione in tre segmenti sarà quindi così strutturata:

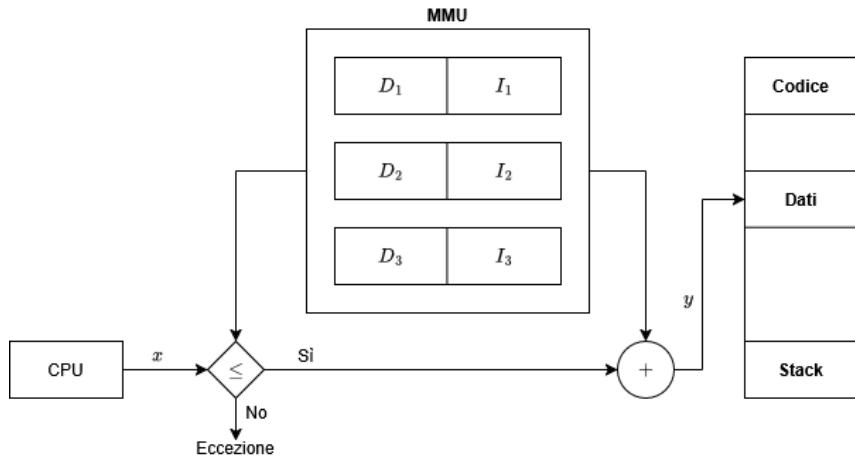


Figura 8.2: Struttura dell'MMU nel caso dei tre segmenti

Uno dei problemi fondamentali è riuscire ad identificare, dato un particolare indirizzo virtuale  $x$ , a quale dei tre segmenti appartiene. Nel caso della segmentazione che abbiamo visto fino ad ora è abbastanza facile determinarlo, infatti possiamo tranquillamente dire che:

- Gli indirizzi generati in fase di prelievo di una istruzione appartengono al segmento codice.

- Gli indirizzi generati durante le fasi di esecuzione o di salto appartengono anch'essi al segmento del codice.
- Gli indirizzi generati durante le fasi di esecuzione delle istruzioni `push` e `pop` appartengono allo stack.

La segmentazione è anche detta **tecnica delle partizioni rilocabili** e, come abbiamo visto, prevede che la rilocazione sia dinamica. Inoltre, l'allocazione della memoria fisica è contigua (per segmento), lo spazio virtuale è segmentato e il caricamento di quest'ultimo avviene tutto insieme. L'idea della segmentazione della memoria virtuale in tre diversi segmenti è però un'idea oramai considerata antiquata e inefficiente, nonostante però permettesse di mantenere i registri base e limite all'interno dell'MMU. Tuttavia, in un approccio che prevede una segmentazione in un numero maggiore di segmenti, diventa difficile riuscire a mantenere tutto all'interno della memoria dell'MMU (basi pensare che nel processore Intel 386 lo spazio virtuale poteva contenere fino a  $2^{14}$  segmenti). Diventa quindi necessario riuscire a mantenere le informazioni su registri base e limite all'interno della memoria centrale; definiamo quindi una struttura dati, chiamata **tabella dei segmenti**. All'interno della tabella, per ogni segmento, si trova un campo contenente l'indirizzo fisico a partire dal quale il segmento è allocato in memoria e un campo contenente le sue dimensioni.

Le tabelle dei segmenti di ogni processo sono, ovviamente, caricate all'interno della memoria centrale del sistema. Il descrittore di un processo, così come contiene lo stato del processore virtuale, contiene anche delle informazioni relative alla memoria virtuale; in particolare, contiene informazioni sul numero di segmenti che compongono la memoria virtuale e l'indirizzo in memoria della tabella dei segmenti. Nel momento in cui un processo entra in esecuzione i valori contenuti nel descrittore del processo vengono inseriti all'interno di due registri del processore

- **STBR**, o **Segment Table Base Register**, che contiene l'indirizzo della tabella dei segmenti del processo in esecuzione.
- **STLR**, o **Segment Table Length Register**, che contiene il numero di segmenti  $n$  del processo.

Per tradurre un indirizzo virtuale  $x = (\text{sg}, \text{of})$  in un indirizzo fisico il primo passo è quello di confrontare il valore  $\text{sg}$  con il valore del contenuto del registro **STLR**. Se  $\text{sg}$  è maggiore a  $n$  allora verrebbe generata un'eccezione di tipo **trap** per indicare il tentativo di accedere un segmento inesistente. In caso contrario  $\text{sg}$  viene utilizzato per accedere alla tabella dei segmenti e trovare l'elemento contenente i valori relativi all'indirizzo iniziale e alla dimensione  $D$  del segmento della partizione di memoria nella quale il segmento  $\text{sg}$  è stato caricato. Se lo scostamento  $\text{of}$  è inferiore o uguale alla dimensione  $D$  del segmento, il suo indirizzo base  $I$  viene sommato allo scostamento  $\text{of}$  al fine di ricaricare l'indirizzo fisico  $y$  che viene quindi inviato alla memoria per accedere all'informazione desiderata. Anche in questo caso, se  $\text{of}$  fosse superiore alla dimensione  $D$  del segmento, si verrebbe comunque a generare un'eccezione **trap**. Introdurre una segmentazione così strutturata porta però due svantaggi principali:

- **Maggior consumo della memoria.** Non è più possibile mantenere la tabella dei segmenti dentro l'MMU e, per questo motivo, la tabella viene mantenuta, come detto in precedenza, nella memoria centrale.
- **Perdita di efficienza da parte della CPU.** Rispetto al caso in cui la tabella si trova all'interno della MMU, si devono duplicare gli accessi in memoria: il primo accesso serve per accedere alla tabella dei segmenti per tradurre l'indirizzo da virtuale a fisico, un secondo per accedere all'informazione desiderata

Per comprendere meglio il problema della perdita di efficienza della CPU è possibile analizzare il funzionamento esatto di questa forma di segmentazione. Nel momento in cui la CPU genera un indirizzo avviene un primo controllo sulla bontà dell'indirizzo, controllando cioè che l'indirizzo sia corretto. Se il controllo restituisce esito positivo, allora il segmento viene utilizzato come indice della tabella dei segmenti. Una volta identificato il segmento si procede con il controllo del registro limite, così da identificare se l'indirizzo esce dal segmento. Se anche questo controllo restituisce

esito positivo allora è possibile procedere con un accesso in memoria. Insomma, si hanno tantissimi accessi in memoria per poter ottenere l'indirizzo virtuale fisico  $y$ .

Per cercare di ridurre questa perdita di efficienza si è scelto di mantenere comunque, all'interno della **MMU**, alcuni **registri associativi**. In ognuno di tali registri viene memorizzato un numero di segmento e i corrispondenti valori dei registri base e limite. In altri termini, se nella MMU sono presenti 8 registri associativi, in essi sono contenuti i valori base e limite degli ultimi 8 segmenti a cui è stato fatto riferimento. L'MMU diventa quindi un buffer vero e proprio che ha come scopo quello di implementare, attraverso il **Translation Lookaside Buffer**, un meccanismo di **caching** degli indirizzi. Il funzionamento può essere rappresentato in questo modo come:

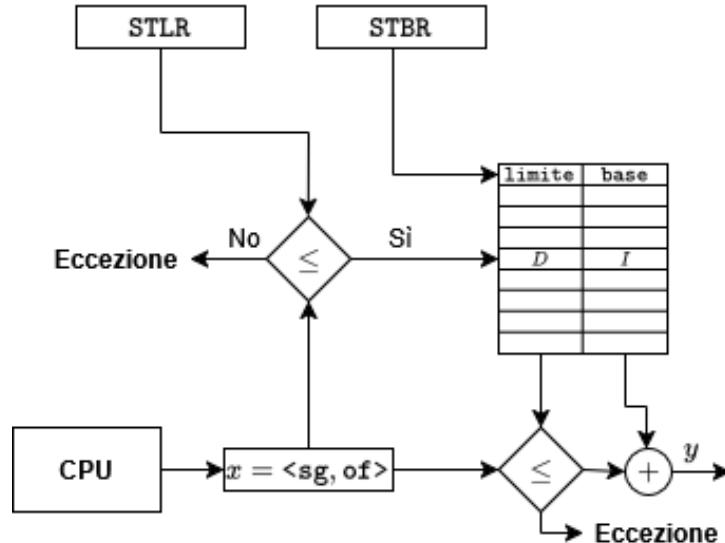


Figura 8.3: Funzionamento effettivo della memoria segmentata

In questo modo, nel momento in cui la CPU genera un indirizzo  $x$  la sua traduzione avviene per prima cosa accedendo al TLB e verificando se quel particolare segmento  $sg$  è già presente. Nel caso in cui fosse presente la traduzione potrebbe procedere prelevando i valori base e limite del registro associativo. Altrimenti la traduzione dovrebbe procedere come in figura 8.3. Ad ogni segmento è anche possibile associare degli specifici diritti d'accesso in modo tale che ogni riferimento a quel segmento sia consistente con tali diritti. Per consentire tali ulteriori controlli, il generico elemento della tabella dei segmenti di un processo contiene, oltre all'indirizzo base e alla dimensione del segmento, anche un terzo campo di controllo nel quale sono presenti alcuni bit che rappresentano i diritti di accesso del segmento. Ad esempio, i bit di accesso in lettura o scrittura.

### Segmentazione a domanda

L'allocazione in memoria dell'intero spazio virtuale di un processo, e cioè di tutti i suoi segmenti, limita la dimensione complessiva dello spazio virtuale del processo. Infatti, la somma delle dimensioni di tutti i segmenti deve essere inferiore alla dimensione della memoria fisica

$$\text{Dimensione memoria virtuale} = \sum_{i=1}^n \text{Dimensione segmento}_i \leq \text{Memoria fisica}$$

Tuttavia è possibile complicare la funzione di rilocazione in modo tale che, durante l'esecuzione di un processo, in ogni istante, sia possibile mantenere in memoria soltanto una parte dello spazio virtuale, cioè un sottoinsieme dei segmenti che compongono la memoria virtuale del processo e che siano strettamente necessari alla sua esecuzione. Per questo motivo è necessario abilitare la funzione di rilocazione, dato un indirizzo  $x = \langle sg, of \rangle$ , a restituire il corrispondente indirizzo fisico  $y$  se il segmento è presente in memoria, o altrimenti a restituire un'interruzione nel caso in cui il segmento non sia presente. Questa interruzione deve essere in grado di accedere alla

memoria, cercare una partizione libera dove caricare il segmento richiesto e, solo successivamente, riabilitare l'esecuzione dall'istruzione che ha generato l'interruzione. La tecnica in questione prende il nome di **segmentazione su domanda**, poiché sfrutta i meccanismi della segmentazione, ma permette di caricare i segmenti in memoria solo nel momento in cui sono necessari. Se all'atto del caricamento di un segmento non si trova una partizione sufficientemente grande per ospitare quel particolare segmento è possibile fare una `swap_out` di una o più partizioni del processo affinché si liberi una partizione tale per ospitare il segmento richiesto; questa tecnica prende il nome di **rimpiazzamento**. Affinché sia però possibile abilitare la funzione di traduzione degli indirizzi a generare un **segmentation fault** in mancanza del segmento richiesto è necessario complicare leggermente la struttura di ogni singolo descrittore di segmento; infatti, oltre ai classici campi **R** e **W** dobbiamo aggiungere un ulteriore bit **P** che indica se il segmento è presente o meno in memoria. Se **P** = 1 allora il segmento è presente in memoria, altrimenti, se **P** = 0 no, e viene quindi generato un **errore di segmentazione**. Un'interruzione di segmentation fault viene gestita in modo tale da caricare in memoria il segmento richiesto; la ricerca di una partizione libera può quindi produrre due risultati:

- Viene trovata la partizione libera e il segmento viene caricato in memoria.
- Non viene trovata una partizione sufficientemente grande (neanche compattando le partizioni libere) allora è necessario fare `swap_out` di segmenti del processo in esecuzione o di altri processi sospesi.

In generale la scelta dei segmenti da rimpiazzare rappresenta uno dei parametri fondamentali che determina l'efficienza di un sistema. La scelta del segmento può essere fatta mediante l'utilizzo di due bit di **uso** e **modifica** posti all'interno del descrittore di segmento; in particolare, il bit di modifica viene messo a uno quando il contenuto del segmento è stato modificato, mentre il bit viene posto a uno quando viene fatto riferimento al particolare segmento. Se un segmento ha il bit di modifica a zero è allora possibile liberare l'area senza la necessità di riscrivere sulla swap-area il contenuto del segmento stesso, ottenendo così un enorme risparmio di tempo.

### 4.3 Memoria paginata

L'allocazione della memoria fisica mediante partizioni, che è alla base di tutte le tecniche viste fino ad ora, risente del problema della frammentazione. Tale inconveniente può essere limitato all'interno dei sistemi che fanno uso di segmentazione attraverso le operazioni di compattazione della memoria. Il compattamento è però un'operazione che richiede comunque un tempo di CPU non indifferente. Per eliminare alla radice il problema sarebbe necessario poter allocare in memoria, in **locazioni fisiche non contigue**, informazioni i cui indirizzi virtuali sono contigui. Per fare ciò è sufficiente utilizzare una funzione di rilocazione dinamica che faccia corrispondere a una locazione fisica qualsiasi un indirizzo virtuale, senza però vincoli di contiguità. Per riuscire a implementare una soluzione di questo tipo è necessario che vengano allocati dei blocchi di locazioni virtuali di dimensioni  $d$  in dei blocchi di locazioni fisiche, non necessariamente contigui, anch'essi di dimensione  $d$ . Definiamo il blocco di locazioni virtuali come **pagina**, mentre il blocco di locazioni fisiche come **frame**. La realizzazione della funzione degli indirizzi diventa quindi abbastanza facile: è sufficiente memorizzare una tabella che registri le corrispondenze tra le **pagine virtuali** e le **pagine fisiche** che le contengono. Se per ogni processo viene mantenuta aggiornata una tabella di corrispondenza tra le pagine virtuali del suo spazio e le pagine fisiche in cui queste sono caricate, la traduzione di un **indirizzo virtuale**  $x$  nel corrispondente indirizzo fisico  $y$  si effettua molto semplicemente ricavando la pagina virtuale  $pv$  e l'offset  $of$  di  $x$ , quindi mediante l'indice  $pv$  di accede alla tabella delle pagine per verificare in quale pagina fisica  $pf$  la pagina virtuale  $pv$  è stata caricata. Noto  $pf$ , l'indirizzo  $y$  si ottiene semplicemente componendolo in modo tale che i suoi 10 bit meno significativi coincidano con  $of$ , mentre i bit più significativi coincidono con la pagina fisica  $pf$  individuata.

Ogni processo allocato in memoria possiede una propria tabella delle pagine, necessaria per realizzare la sua funzione di traduzione degli indirizzi. La dimensione della tabella dipendendo dal numero di pagine che compongono lo spazio virtuale del processo e questo numero, a sua volta, dipende dalla dimensione dell'intero spazio e dalla dimensione delle pagine. La paginazione

presenta, inoltre, un problema simile a quello della segmentazione, non è quindi possibile allocare interamente la tabella delle pagine all'interno dell'MMU. Esiste, a tale scopo, un particolare registro macchina che in ogni istante contiene l'indirizzo fisico della tabella delle pagine del processo in esecuzione; questo registro prende il nome di **RPTP** (Root Page Table Pointer). Per ridurre il numero di accessi alla tabella delle pagine, anche nei sistemi basati su paginazione, vengono mantenute le corrispondenze tra **pagine virtuali** e **pagine fisiche** usate più di recente (TLB).

Il gestore della memoria mantiene anche un elenco delle pagine fisiche disponibili in una vera e propria struttura dati, normalmente una tabella con tanti elementi quante sono le pagine fisiche della memoria. Ogni elemento della tabella contiene l'indicazione se la corrispondente pagina fisica è libera o occupata e, in quest'ultimo caso, contiene l'indice del processo a cui è allocata e l'indice della pagina virtuale in essa caricata. Quando un processo deve essere caricato in memoria, viene richiesto al gestore della memoria un numero di pagine fisiche disponibili uguale al numero di pagine virtuali del suo spazio. Le pagine ottenute, anche se non contigue, vengono utilizzate per caricarci le pagine virtuali e generare quindi la tabella delle pagine del processo. Se le pagine fisiche non sono sufficienti non è necessario eseguire alcuna compattazione, sarà sufficiente scaricare un certo numero di pagine mediante la primitiva di **swap\_out**. Inoltre, all'interno di ogni descrittore di processo troviamo due informazioni

- L'indirizzo in memoria della tabella delle pagine di processo.
- Un intero corrispondente al numero delle pagine virtuali del suo spazio, e cioè al numero di elementi della tabella delle pagine.

Un parametro importante nei sistemi paginati è costituito dalla dimensione delle pagine. Occorre infatti prestare attenzione a non creare pagine troppo piccole e/o troppo grandi, in quanto si potrebbero creare delle problematiche relative alla dimensione della tabella delle pagine. Infine, per quanto riguarda la condivisione, è opportuno ricordare che se le pagine sono condivise contengono degli indirizzi, essendo questi indirizzi virtuali, ciò comporta il vincolo che le informazioni condivise devono essere allocate nelle stesse locazioni dei rispettivi spazi virtuali.

### Paginazione a domanda

Con la tecnica della paginazione un processo deve avere l'intero spazio virtuale in memoria per poter eseguire. Tuttavia, anche analizzando quello che abbiamo visto nel caso della segmentazione su domanda, sono innumerevoli i vantaggi che possiamo ottenere abilitando il sistema a caricare solo alcune parti del proprio spazio virtuale solo nel momento in cui risultano essere necessari. Analogamente alla tecnica della **segmentazione su domanda**, possiamo anche definire una tecnica nota come **paginazione su domanda**. Anche il funzionamento è simile a quello della segmentazione su domanda, è infatti sufficiente aggiungere ad ogni descrittore di pagina un bit **P** che indica se la pagina è stata caricata o meno in memoria. Analogamente sono utilizzati anche i bit di **uso** e **modifica** per decidere quali pagine rimpiazzare.

Ad ogni riferimento relativo a una pagina non in memoria viene generata un'interruzione di **page-fault** (letteralmente **mancanza della pagina**) che viene gestita dal sistema operativo per ricercare nella tabella delle pagine fisica una pagina fisica in cui caricare la pagina virtuale richiesta. All'avvio del sistema, normalmente, tutte le pagine risiedono su altrettanti settori fisici della **swap\_area**. Infatti, il processo viene creato senza caricare nessuna pagina, ma può subito essere schedulato anche se non ha niente in memoria utilizzando questa tecnica di caricamento su domanda. Se però al momento del caricamento di una pagina, tutti i frame fossero occupati, sarebbe necessario procedere con un **rimpiazzamento** rispetto a una pagina. Tuttavia, questo rimpiazzamento, è molto più semplice rispetto a quello visto nella segmentazione, poiché abbiamo pagine di uguale dimensioni.

La scelta dell'algoritmo per il rimpiazzamento della pagine è una decisione fondamentale per la corretta efficienza del nostro sistema. L'idea più elementare potrebbe essere quella di realizzare un sistema che implementa una politica del tipo **FIFO**, dove la scelta della pagina da rimpiazzare viene fatta prendendo quella che è da più tempo in memoria, sperando che non sia più necessaria. L'idea che però viene effettivamente utilizzata al giorno d'oggi è quella della politica **LRU**, o **Last Recently Used**, dove la scelta viene fatta prendendo la pagina meno recentemente utilizzata, indipendentemente dal momento in cui è stata caricata. Si deve quindi aggiungere un **timestamp**

ai descrittori di pagina. In Unix l'algoritmo utilizzato per il rimpiazzamento delle pagine prende il nome di **second-chance**. In questo algoritmo la tabella delle pagine fisiche viene gestita come un array circolare; viene mantenuta una variabile **vittima**, che consiste in un puntatore contenente l'indice della pagina fisica successiva a quella rimpiazzata per ultima. Al **page-fault** si inizia a verificare la pagina il cui indice è contenuto nella variabile **vittima**, esaminando il bit **U** associato:

- Se il bit U è a zero, allora la pagina viene scelta per il rimpiazzamento.
- Se il bit U è a uno, allora si azzerà il bit e si incrementerà la variabile vittima, andando a esaminare la pagina successiva.

Il bit U è un bit che permette di capire se una particolare pagina è stata utilizzata di recente. Poiché l'incremento della variabile vittima viene fatto operando con il modulo  $M$ , allora è certo che una pagina con il bit di uso a zero venga trovata.

## 5 Segmentazione e Paginazione

Le due tecniche, cioè quella di segmentazione e quella di paginazione, nonostante possano sembrare concettualmente simili, presentano caratteristiche che le rendono profondamente diverse nella loro natura. La prima corrisponde infatti a un criterio di organizzazione della memoria virtuale, non più vista come una semplice sequenza di byte, ma come un insieme di moduli separati e semanticamente indipendenti. La seconda corrisponde a una tecnica per gestire la **memoria fisica**, permettendo la sua allocazione in maniera **non contigua** e eliminando il fenomeno della **frammentazione esterna**. Unendo queste due tecniche è possibile pensare di creare un'unica tecnica di allocazione che preveda sia la strutturazione dello **spazio virtuale** in **segmenti**, sia la loro allocazione con la **tecnica della paginazione**; questa tecnica prende il nome di **segmentazione paginata**. In questa tecnica la CPU genera un indirizzo virtuale  $x$  composta da due componenti, la prima è il segmento **sg** e la seconda è lo scostamento **sc** nel segmento. Ogni **segmento** è quindi dotato di una **tabella delle pagine** che specifica il modo in cui il segmento stesso è allocato all'interno della memoria fisica sui diversi frame. Lo scostamento, che rappresenta un indirizzo lineare nel segmento, viene paginato scomponendolo nei bit meno significativi (**offset** della pagina) e nei bit più significativi (**pagina virtuale**). L'indice della pagina virtuale viene utilizzato per accedere alla tabella delle pagine e ricavare l'indice della pagina fisica **pf** nella quale **pg** è stata caricata.

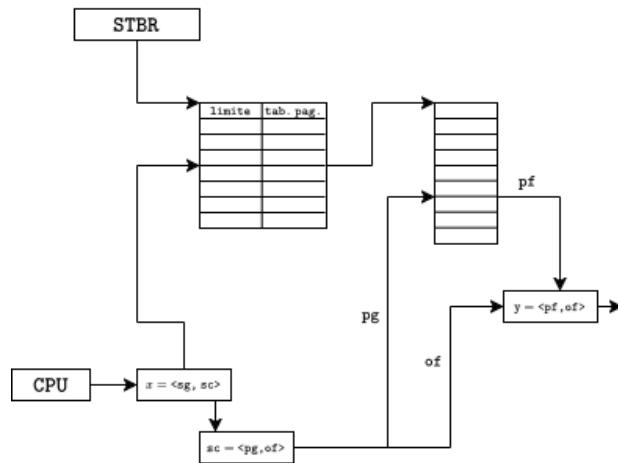


Figura 8.4: Struttura della segmentazione paginata

La tabella delle pagine per un particolare segmento non ne necessariamente in memoria, quindi l'interruzione di tipo **segmentation-fault** viene generata nel momento in cui è necessario accedere a tale tabella e deve essere recuperata dalla memoria. Un esempio di sistema operativo che sfrutta un approccio a segmentazione paginata è **unix**. In unix lo spazio di indirizzamento di ogni processo è segmentato in tre segmenti distinti: **codice** (*code segment*), **dati** (*data segment*) e **stack** (*stack*)

*segment*). L'allocazione dei segmenti viene gestita con la tecnica della paginazione su domanda effettuata dal kernel, con l'aiuto del processo di sistema **pagedeamon**. Questo processo esegue l'algoritmo di sostituzione delle pagine soltanto se il numero delle pagine fisiche libere è inferiore a un valore di soglia prefissato, rappresentato dalla costante di sistema **lotsfree**. Può darsi, tuttavia, che nonostante l'intervento del **pagedeamon**, la frequenza di paginazione sia troppo elevata, e il numero di segmenti liberi rimanga comunque inferiore alla soglia **lotsfree**: in questo caso interviene lo **swapper** che, in modo più drastico, provvede al trasferimento di uno o più processi dalla memoria centrale a quella secondaria. Per rilevare questa situazione il sistema definisce il numero minimo di pagine fisiche libere necessarie per evitare lo swpa-out dei processi (**minfree**) e il numero medio desiderabile di pagine fisiche libere (**desfree**). La relazioni tra le tre costanti è quindi:

$$\text{lotsfree} > \text{numero\_pagina\_libere} \approx \text{desfree} > \text{minfree}$$

La situazione ideale per il sistema è quella in cui si riesce a mantenere il **numero\_frame\_liberi** > **lotsfree**. In questo stato, la memoria è considerata sufficiente e non sono necessari interventi correttivi.

Tuttavia, quando il numero di pagine libere scende al di sotto di **lotsfree**, il sistema entra in uno stato di pressione e interviene il **pagedaemon**. Questo processo cerca di ripristinare la disponibilità di memoria al di sopra della soglia di sicurezza tramite il *paging-out* (liberando singoli frame non utilizzati). Se la pressione persiste, il pagedaemon aumenta l'intensità della sua scansione nel tentativo di contrastare la discesa verso la soglia successiva.

Se, nonostante l'azione del pagedaemon, la memoria continua a calare scendendo sotto la soglia **desfree**, il solo meccanismo di paginazione non è più ritenuto sufficiente. In questo scenario critico entra in gioco lo **swapper**. A differenza del daemon, lo swapper agisce in modo più drastico sospendendo interi processi e spostandoli in massa sul disco (swap-out). L'obiettivo finale di questi interventi non è mantenere il sistema in una zona di scarsità, bensì liberare un quantitativo di frame tale da riportare il numero di pagine libere nuovamente al di sopra di **lotsfree**.

Infine, la soglia **minfree** rappresenta il limite minimo assoluto: qualora la memoria dovesse scendere sotto questo livello, il sistema entrerebbe in uno stato di emergenza totale, bloccando ogni ulteriore allocazione per prevenire il collasso del kernel.

### Gestione di un page fault e di un segment fault

**segment fault** e **page fault** sono due errori che si possono verificare quando il sistema richiede una particolare area di memoria, segmento o pagina, il cui bit di presenza sia pari a 0. Nel caso della memoria a segmentazione paginata, quando si cerca di accedere ad un segmento non caricato in memoria si verifica un **segment fault** (da non confondere con il **segmentation fault**, che è invece l'errore che si verifica quando si cerca di accedere ad un'area di memoria "vietata"), questa interruzione manda in esecuzione un **interrupt handler** che esegue le seguenti operazioni:

- Verifica la presenza di un numero sufficiente di frame liberi per ospitare la **tabella delle pagine** per quel segmento.
- Se non trova un numero sufficiente di pagine allora procede con il rimpiazzamento di un numero sufficiente di pagine fisiche, secondo la politica adottata dal sistema.
- Se invece trova un numero sufficiente di pagine fisiche, oppure ha terminato il rimpiazzamento, procede con il caricamento della tabella delle pagine all'interno della memoria fisica del sistema.
- Aggiorna il bit di presenza all'interno della tabella dei segmenti.

A questo punto, una volta caricata la tabella delle pagine, il sistema prova ad accedere alla tabella delle pagine, all'indice specifico per quella pagina. Non essendo ancora alcuna pagina caricata in memoria, verrà sollevata un interruzione del tipo **page fault**. Questo **page fault** segue una procedura analoga a quella del segment fault:

- Verifica la presenza di una pagina fisica per ospitare la pagina richiesta.

- Se non trova alcuna pagina fisica libera si procede con un rimpiazzamento, facendo in modo di liberare sufficiente spazio e successivamente si carica la pagina.
- Se invece la pagina fisica è immediatamente disponibile si può caricare immediatamente la pagina.
- Si aggiorna la corrispondente entrata nella tabella delle pagine del segmento aggiornando il bit di presenza e l'indirizzo base della pagina caricata.

## 6 Gestione degli stati di un processo

Abbiamo visto inizialmente, nel capitolo dedicato alle informazioni generali sui processi, il **modello a cinque stati**, cioè il modello che rappresenta i possibili stati di un processo su cinque possibili stati: **creato**, **pronto**, **bloccato**, **esecuzione** e **terminato**. In questo capitolo abbiamo visto come, sfruttando la tecnica della **memoria segmentata**, un processo può essere in due diverse condizioni:

- Allocato in memoria.
- Non allocato in memoria.

Nella prima condizione i segmenti del processo sono tutti caricati all'interno di uno stesso numero di partizioni della memoria centrale, mentre nella seconda condizione i suoi segmenti sono allocati su un certo numero di **blocchi** all'interno della **memoria secondaria** del sistema (**swap-area**). Quando un processo si trova nella seconda condizione non può essere immediatamente schedulato, poiché il processo non è di fatto pronto, poiché parte delle sue strutture non sono allocate nella memoria. Oltre ai cinque stati è necessario quindi introdurre due stati di **pronto** e **bloccato**, ma relativi alla memoria secondaria. Il grafo di stato del processo diventa quindi

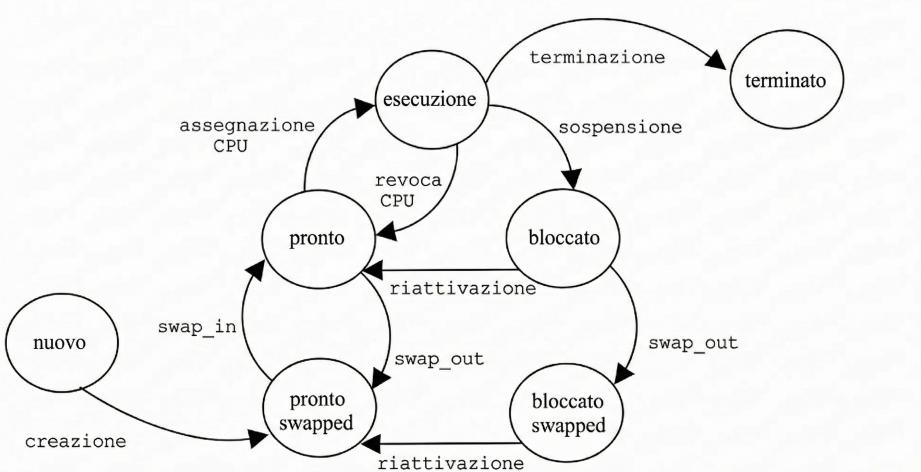


Figura 8.5: Diagramma degli stati di un processo che implementa **swap\_in** e **swap\_out**

La prima differenza sostanziale è che un processo appena creato non verrà subito messo nello stato di pronto, poiché buona parte delle sue strutture dati saranno contenute nella memoria secondaria; si porrà quindi il processo in uno stato di **pronto-swapped**. La scelta dei processi nello stato di **pronto-swapped** da caricare in memoria viene presa da quello che nel capitolo sugli algoritmi di scheduling abbiamo definito come **medium-term scheduling**. Anche un processo bloccato in attesa di un'operazione di **I/O** potrebbe essere **swappato** fuori dalla memoria, dobbiamo quindi introdurre uno stato di **bloccato-swapped**. Un processo in tale stato potrà essere riattivato, ma il suo stato non verrà posto a pronto, ma bensì a **pronto-swapped**, in quanto molte delle sue strutture dati sono ancora in memoria secondaria.

## Struttura dei processi in unix

Nella sezione sulla gestione dei processi abbiamo visto come un processo possa **avere parte della sua memoria virtuale** allocata all'interno della memoria secondaria, ma non della memoria principale. Il motivo per cui diciamo che solo parte della memoria virtuale di un processo viene spostata dalla memoria principale è dato dal fatto che alcune strutture contengono informazioni indispensabili sul processo, che non possono essere rimosse dalla memoria principale per alcun motivo. Unix utilizza questa distinzione suddividendo l'area di memoria virtuale di un processo in due parti

- **User Structure.** Contiene delle informazioni necessarie per la gestione del processo solo quando esso si trova in memoria principale.
- **Process Structure.** Contiene delle informazioni indispensabili sul processo, che non possono essere rimosse dalla memoria per alcun motivo.

La **process structure** contiene delle informazioni come il **PID** del processo, il suo **stato**, il **riferimento** alle aree relative a **dati** e **stack**. Contiene inoltre delle informazioni relative al **PID** del processo padre, alla priorità del processo. La **user structure**, da canto suo, contiene una copia dei registri della CPU (**CPU virtuale**), informazioni sulle risorse allocate (TFAP), informazione sulla gestione degli eventi asincroni (segnali), il **direttorio corrente**, **gruppo**, etc.

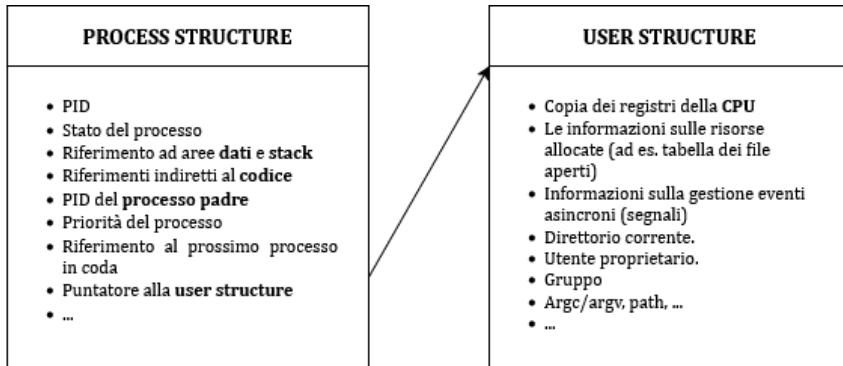


Figura 8.6: Struttura della User Structure e della Process Structure

Il codice dei processi unix è detto **rientrante**, cioè può essere condiviso tra più processi. Il kernel mantiene, a tal proposito, una struttura dati globale chiamata **text table** nella quale ogni elemento rappresenta il codice di un programma correttamente eseguito da uno o più processi. Ogni elemento di questa tabella viene detto **text structure** e contiene un puntatore all'area di memoria in cui è allocato il codice. Se il processo fosse **swapped** allora sarebbe un riferimento alla memoria secondaria.

## Gestione delle periferiche

Analizzando tutte le attività che vengono svolte da un calcolatore, è possibile distinguere due categorie di operazioni e di attività: **attività di I/O** e **attività di elaborazione svolte dalla CPU**. Le prime sono attività principalmente relative al trasferimento nel sistema dei dati da elaborare e alla restituzione dei risultati, mentre le seconde sono le attività relative alla generazione di questi risultati. Tutte le attività di **I/O** vengono svolta attraverso l’interazione del sistema con dei dispositivi, detti **periferiche**, che forniscono un interfacciamento tra la CPU e l’ambiente che lo circonda. Questo circuito di interfaccia, detto **controllore della periferica**, permette la **sincronizzazione** tra la CPU e la periferica ed è collegato al **bus** del sistema.

### 1 Gestore di un dispositivo

Il controllore è quindi quel componente che attraverso un ben definito protocollo è in grado di controllare il funzionamento di una periferica, inviando e ricevendo dalla stessa dei segnali. Allo stesso tempo è anche ovviamente collegato al bus, altrimenti non avrebbe modo di comunicare con la CPU. Tuttavia la CPU non ha effettivamente idea di cosa sia il controllore, infatti il controllore espone una piccola memoria e dei registri alla CPU, i quali devono essere però **indirizzati** all’interno dello spazio di **I/O** del sistema. Il vantaggio di un approccio basato sull’esporre dei registri è proprio quello di permettere al programmatore di ignorare la struttura interna del controllore, astraendo il dispositivo come, semplicemente, un insieme di registri.

Un qualsiasi dispositivo, in combinazione al suo controllore, è associabile a un processo; in particolare, a un **processo esterno**. Ogni controllore si trova in uno **stato di attesa**, dove attende l’arrivo di un **comando** attraverso la scrittura da parte del **processo interno** all’interno di uno dei due registri che il controllore espone nello spazio di I/O; in particolare, possiamo distinguere tre registri principali:

- **Registro di stato.** Il registro di stato è viceversa un registro in sola lettura per la CPU ed è utilizzato dal dispositivo per mantenere aggiornato lo stato in cui si trova il dispositivo.
- **Registro di controllo.** Il registro di controllo permette alla CPU di controllare il funzionamento del dispositivo. In pratica, rappresenta un registro **in sola scrittura** nel quale la CPU può trasferire valori al fine di attivare il dispositivo affinché lo stesso svolga un certo compito.
- **Registri dati.** Il registro di dati è un registro sia in lettura che in scrittura, dove la CPU può inserire i dati in uscita o prelevare quelli in ingresso.

I registri di stato e di controllo possiedono, al loro interno, dei bit specifici che permettono di controllare o di attuare delle operazioni sul dispositivo. All’interno del registro di controllo troviamo infatti un **bit di start**, che permette, una volta posto a uno, al controllore di attivare il dispositivo inviando gli opportuni segnali, inoltre, qualora il dispositivo permetta di svolgere più operazioni, ci saranno dei bit aggiuntivi per poter determinare quale operazione svolgere e, infine, è presente

anche un **bit di abilitazione alle interruzioni** che, se posto a uno in fase di attivazione, abilita il controllore a inviare un segnale di interruzione alla CPU a fine delle operazioni del dispositivo. Nel registro di stato troviamo invece un bit di **flag** che viene posto a uno alla fine di un'operazione da parte del dispositivo.

### Gestione di un dispositivo a controllo di programma

Quindi, il comando viene scritto all'interno del registro di controllo dal processo in esecuzione, che successivamente si mette in attesa dell'esecuzione del comando. Il processo esterno è di default in attesa della ricezione di un comando, e al momento della sua ricezione, si attiva, ed esegue il comando. Una volta terminato segnala al processo in attesa attraverso la scrittura nel registro di stato di un particolare flag. A questo punto il processo che era in attesa del termine del processo esterno può controllare l'esito dell'operazione; nello specifico: se l'operazione è terminata con successo può procedere con il suo normale flusso d'esecuzione, altrimenti può decidere di ripetere nuovamente l'operazione.

```
for(int i = 0; i < n; i ++){  
    <preparazione comando>  
    <invia comando>  
    do(); while (flag == 0);  
    <verifica esito>  
}
```

Il processo in esecuzione rimane quindi in attesa passiva, controllando iterativamente il flag, fintanto che l'operazione non è terminata, prima di poter proseguire di nuovo. Il problema di questa tecnica, detta a **controllo di programma**, è che per tutto il tempo in cui il processo esterno esegue il comando, il processo in esecuzione occupa la CPU senza eseguire alcuna operazione. Data la sua natura ci si rende facilmente conto che uno schema a controllo di programma è poco conveniente in un sistema multiprogrammato.

### Gestione di un dispositivo a interruzione

Una soluzione più funzionale potrebbe essere quella di sfruttare il meccanismo delle **interruzioni**; nello specifico: il processo in esecuzione, una volta inviato il comando, si sospende su un semaforo **dato\_disponibile**, lasciando quindi la CPU libera per un altro processo che potrebbe voler eseguire. Nel momento in cui il processo esterno termina l'esecuzione del comando invia un segnale di interruzione, che risveglia un particolare **gestore dell'interruzione**, il cui compito sarà quello di risvegliare il processo attraverso una **signal**. L'esecuzione della **signal** risveglia il processo e lo mette in coda pronti, così che lo scheduler possa eseguirlo; lo schema temporale della seguente esecuzione è:

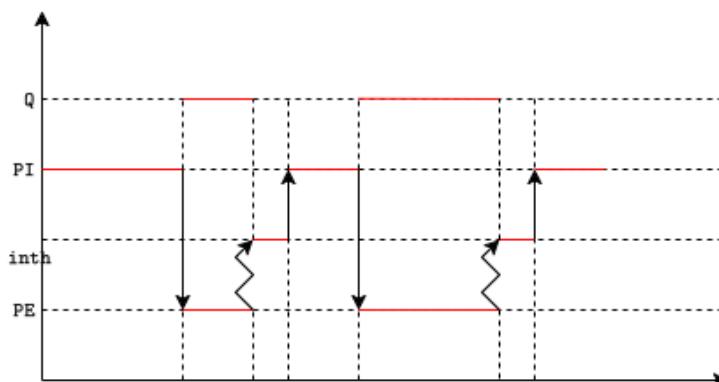


Figura 9.1: Temporizzazione della gestione a interruzione

In questo esempio, all'istante iniziale, il processo applicativo PI, mentre esegue la prima iterazione

del ciclo for, si blocca sul semaforo `dato_disponibile`. Successivamente alla `wait()`, il processo subisce un cambio di contesto e viene quindi schedulato un altro processo (Q) e, in parallelo all'attività della CPU inizia l'attività del dispositivo PE. Quando termina questa attività e viene posto a uno il flag, la CPU viene interrotta e va in esecuzione la routine di interruzione `inth` che, eseguendo una `signal` sul semaforo `dato_disponibile`, riattiva il processo PI il quale, nell'ipotesi che venga schedulato, torna in esecuzione. Successivamente si ripete questa sequenza di eventi per ognuno degli  $n$  dati da leggere (cioè per ognuna delle  $n$  iterazioni del ciclo for).

Tuttavia, anche questa soluzione presenta un problema: se il processo applicativo deve trasferire  $n$  dati, ed eseguire quindi  $n$  iterazioni del ciclo `for`, per  $n$  volte viene sospeso e di seguito riattivato inutilmente poiché deve subito sospendersi di nuovo. Tutta questa sequenza di inutili commutazioni di contesto genera esclusivamente una perdita di tempo. Si potrebbe quindi pensare di delegare il compito del trasferimento dal buffer del dispositivo al buffer interno e viceversa al gestore dell'interruzione. In questo modo è possibile ridurre il numero di cambi di contesto senza dover necessariamente attivare e disattivare il processo esterno. In particolare:

- Il processo interno attiva il processo esterno.
- Il processo esterno fa le sue operazioni e manda il segnale di interruzione.
- Il gestore delle interruzioni a questo punto dovrà fare tutti i suoi controlli e, nel caso, attivare nuovamente il processo esterno.
- Il gestore delle interruzioni, quando completa il trasferimento di tutti i byte, riattiva il processo interno.

Una struttura di questo tipo richiede che il controllore dell'interruzione sia in grado di capire quanti trasferimenti deve ancora fare prima di dover risvegliare il processo principale. Se infatti il gestore delle interruzioni non avesse questa informazione si troverebbe in una situazione in cui non saprebbe quando svegliare il processo principale. Un modo semplice per risolvere il problema è quello di riservare in memoria una struttura dati destinata a rappresentare il dispositivo e sulla quale possa operare sia il processo applicativo e sia la routine di gestione delle interruzioni lanciate dal dispositivo. Questa struttura dati prende il nome di **descrittore del dispositivo**. Un descrittore è costituito da una prima parte contenente i tre indirizzi che contengono i registri che permettono la visione funzionale del dispositivo. Questi valori vengono scritti nel descrittore durante la fase di bootstrap del sistema, il quale verifica i dispositivi effettivamente connessi al bus. Troviamo un anche un semaforo per l'accesso in mutua esclusione ai buffer, un contatore di quanti dati sono da inserire e anche un puntatore al buffer in memoria. L'ultimo campo invece è relativo al solo all'esito delle operazioni.

## 2 Driver di un dispositivo

Per semplificare quanto è stato detto, possiamo adesso illustrare come è strutturato il driver di un dispositivo, mostrando le funzioni di lettura o scrittura e la funzione di risposta alle interruzioni; dove il device driver è definito come la combinazione tra il **descrittore** del dispositivo e le **funzioni di accesso**. Prendiamo come esempio la funzione `read`:

```
int read(int disp, char* pbuf, int cont);
```

dove `disp` identifica il dispositivo su cui operare, `pbuf` il puntatore al buffer di sistema in cui trasferire i dati letti, e `cont` il numero di dati da leggere. La funzione trasferisce per prima cosa nei corrispondenti campi del descrittore i valori relativi al numero di dati da leggere e l'indirizzo del buffer dove trasferirli

```
descrittore[disp].contatore = cont;
descrittore[disp].puntatore = pbuf;
```

Quindi, attiva il dispositivo scrivendo nel registro di controllo del controllore e blocca il processo applicativo sul semaforo `dato_disponibile`

```

<attivazione del dispositivo>
descrittore[disp].dato_disponibile.wait();

```

Al fine del trasferimento il processo viene riattivato e la funzione termina leggendo il campo **esito** del descrittore nel quale è stato codificato se il trasferimento è terminato erroneamente o in modo scorretto. Nel primo caso viene restituito il valore  $-1$ , nell'altro il numero di dati effettivamente letti:

```

if(descrittore[disp].esito = <codice_errore>){
    return -1
}
return (cont-descrittore[disp].contatore);

```

All'arrivo di un'interruzione il processo in esecuzione viene interrotto e, via hardware, viene messo in esecuzione il gestore delle interruzioni **inth**. Una volta messo in esecuzione, il gestore delle interruzioni controlla il registro di stato per identificare la causa dell'interruzione, in particolare: se l'interruzione non è dovuta ad errori ma alla fine della lettura di un dato da parte del dispositivo, il dato stesso viene letto nella variabile locale **b**, prelevandolo dal registro dati, e successivamente trasferito nel buffer di sistema all'indirizzo contenuto nel campo puntatore **puntatore**. Quindi il campo **puntatore** viene incrementato per renderlo consistente col trasferimento del prossimo dato e il campo **contatore** viene decrementato per registrare che rimane da leggere un dato in meno. Se dopo il decremento il **contatore** è ancora diverso da zero ciò significa che il trasferimento non è ancora completato e quindi il dispositivo viene riattivato per il prossimo trasferimento. Altrimenti l'interruzione ricevuta è l'ultima dell'itero trasferimento e, quindi, il processo applicativo può essere riattivato. Viene infine scritto nel campo **esito** che il trasferimento si è concluso correttamente e il dispositivo viene disattivato. A questo punto la routine termina con un ritorno da interruzione. Il codice del gestore dell'interruzione è quindi

```

void int() {
    char b;
    <lettura registro di stato>
    if(bit_errore == 0){
        <lettura registro dati>
        <assegnamento variabile locale b>
        *descrittore[disp].puntatore = b;
        descrittore[disp].puntatore++;
        descrittore[disp].contatore--;
        if(descrittore[disp].contatore != 0){
            <riattivazione dispositivo>;
        } else {
            descrittore[disp].esito = <terminazione corretta>;
            <disattivazione dispositivo>
            // riattivazione processo
            descrittore[disp].dato_disponibile.signal();
        }
    } else {
        <routine gestione errori>
        if (<errore non recuperabile>)
            descrittore[disp].esito = <codice errore>;
        // riattivazione processo
        descrittore[disp].dato_disponibile.signal();
    }
    return;
}

```

Se il gestore delle interruzioni identifica che l'interruzione è dovuta ad un errore, prima di tutto esegue una routine di gestione degli errori. Se l'errore non è recuperabile allora restituisce come esito il codice di errore, altrimenti risveglia il processo senza modificare il campo **esito**. Dal punto

di vista del sistema il **device driver** è una serie di funzioni che il produttore dell'hardware mette a disposizione per interagire con la struttura che descrive il nostro dispositivo. Di fatto possiamo definire il device driver come l'unione delle funzioni di interfaccia e del descrittore di dispositivo:

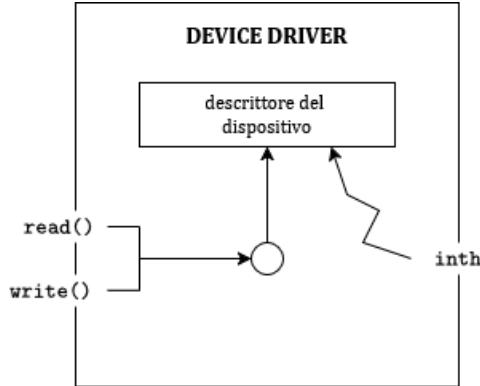


Figura 9.2: Struttura di un device driver. Attraverso le funzioni `read()` e `write()` il processo applicativo è in grado di accedere al descrittore dispositivo

L'utente utilizzando le funzioni di utilità è in grado di interagire indirettamente con il descrittore di dispositivo, andando a segnalare di svolgere una particolare operazione. L'interazione effettiva con la struttura dati del descrittore di dispositivo avviene però mediante il **gestore delle interruzioni** `inth`.

### 3 Gestione della temporizzazione

Un particolare dispositivo per la gestione delle interruzione che troviamo all'interno dei nostri calcolatori è un **timer**. Questo dispositivo non è altro che un vero e proprio **generatore di interruzioni** temporizzate, cioè interruzioni intervallate in modo omogeneo nel tempo. L'introduzione di questo dispositivo ha due motivazioni principali:

- Nella schedulazione con algoritmo **Round-Robin** ad ogni processo viene assegnata una **time slice**, al termine della quale al processo deve essere revocata una CPU.
- In molti sistemi in **tempo reale** è necessario fornire ai processi applicativi un insieme di **servizi** connessi alla **gestione del tempo**.
- In molti sistemi è necessario mantenere aggiornata la gestione della data e del tempo, così che la data corrente e l'orologio siano sempre coerenti.

Da un punto di vista hardware, il controllore di un timer contiene, oltre ai registri di controllo e di stato, un registro contatore nel quale la CPU può trasferire un valore che, successivamente, il timer decrementa con frequenza fissa, o in certo casi programmabile. Quando il contatore arriva a zero, il controllore lancia un segnale d'interruzione. Come ogni altro **dispositivo**, anche il timer avrà un sua sua entrata all'interno della tabella dei **descrittori di dispositivo** e una funzione di risposta alle interruzioni `inth`. Un **timer** è dunque un dispositivo il cui unico scopo è quello di *contare* e, di conseguenza, mette a disposizione una sola primitiva, chiamata `delay`. La funzione `delay` è quindi una funzione che un processo può invocare, passando come parametro un intero `n` che identifica il tempo da attendere. La struttura della primitiva è

```

void delay (int n){
    int proc;
    proc = <indice processo esecuzione>;
    descrittore.ritardo[proc] = n;
    descrittore.fine_attesa[proc].wait();
}

```

L'idea della funzione è quindi abbastanza semplice, all'interno del descrittore si mantengono due **array**, il primo array, detto **ritardo**, contiene il ritardo che ogni processo deve attendere, mentre il secondo array, detto **fine\_attesa** è un array di semafori su cui il processo che attende lo scadere del timer si sospende. Quando un processo chiama la primitiva **delay**, avverrà l'inizializzazione del **ritardo** per quel processo, che successivamente si sospende sul semaforo associato. Il gestore l'interruzione **inth** per il timer sarà quindi fatto come:

```
void intth(){
    for(int i = 0; i < N; i++){
        if (descrittore.ritardo[i] != 0){
            descrittore.ritardo[i]--;
            if (descrittore.ritardo[i] == 0){
                descrittore.fine_attesa[proc].signal();
            }
        }
    }
}
```

Anche in questo caso il codice è abbastanza semplice, per ogni processo all'interno del sistema si verifica che il timer sia inizializzato, successivamente si decrementa il contatore e si verifica che non sia arrivato a 0; nel caso in cui il timer sia arrivato a 0 si risveglia il processo.

## 4 Il processo esterno

Come abbiamo accennato all'inizio, un qualsiasi dispositivo in combinazione al suo controllore è associabile a un processo esterno. Il controllore è infatti un dispositivo programmabile, o comunque un dispositivo che esegue un particolare **firmware** interno, che gli permette di eseguire delle azioni che possiamo a tutti gli effetti classificare come dei processi esterni. Di fatti, quando parliamo di **gestione dei dispositivi di I/O** ci riferiamo ad un problema di sincronizzazione tra **processi applicativi** e **processi esterni**. Oltre quindi all'esecuzione delle operazioni sui vari dispositivi, il sottosistema di **I/O** deve anche

- Definire lo spazio dei nomi con cui identificare i dispositivi.
- Gestire eventuali malfunzionamenti.
- Garantire sincronizzazione tra l'attività di un dispositivo e quella del processo che lo ha attivato.
- Gestire la **bufferizzazione**.

La funzionalità corrispondente al primo punto prende il nome di **naming**. Questa funzionalità permette l'identificazione di ogni dispositivo mediante un nome simbolico, che agirà da puntatore al **descrittore di dispositivo** (in Unix il *naming* traduce il nome simbolico in un *i-node*). Un'altra delle funzionalità principali offerte dal sottosistema di I/O è quella di fornire una o più aree di memoria tampone, dette **buffer**, destinate a contenere i dati durante il trasferimento tra il dispositivo e l'area di memoria virtuale del processo applicativo da cui i dati devono essere prelevati e/o inseriti. La necessità di avere una bufferizzazione di questo tipo nasce dalla volontà di fare fronte alla notevole differenza di velocità con cui il processore può leggere e consumare dati e quella del dispositivo di consumarle e produrle. Ipotizziamo di avere una funzione

```
n = read(fd, ubuf, nbytes);
```

Normalmente quindi un processo applicativo deve sospendere la propria esecuzione fino a quando i dati richiesti sono stati scritti in **ubuf**. Un'organizzazione di questo tipo si può notare nel caso in cui il processo deve elaborare *sequenze multiple di blocchi*. In questi casi è anche possibile operare un'ottimizzazione, permettendo l'esecuzione in parallelo dell'esecuzione di un blocco dati e della lettura del successivo blocco dati (**politica read-ahead**).

## 5 Gestione e organizzazione interna dei dischi

Le unità di memoria di massa, soprattutto i dischi, rappresentano dei dispositivi di particolare importanza agli effetti dell'efficienza e dell'operatività dell'intero sistema. I dischi sono infatti quelle componenti che implementano le **swap\_area**, oppure dove sono memorizzati i file del sistema, ed è quindi necessarie che siano quanto più efficienti possibili. Un **disco** è fisicamente costituito da un piatto di plastica su cui viene depositato uno **strato di materiale magnetico** sul quale i dati possono essere scritti, e successivamente letti, tramite una **testina** di lettura/scrittura che modifica lo stato di magnetizzazione del materiale e ne rileva lo stato in fase di lettura. Esistono due diverse tipologie di dischi magneticici:

- **Dischi a testina fissa.**
- **Dischi a testina mobile.**

Nella prima tipologia di disco abbiamo tante testine di lettura/scrittura quante sono le tracce del disco. Quindi non è necessario fare delle operazioni particolari in fase di lettura o scrittura. Nella seconda tipologia abbiamo invece un'unica testina per tutti i settori, di conseguenza è necessario che prima di una operazione di lettura o di scrittura la testina si muova sulla traccia corrispondente. Un disco rigido è così organizzato:

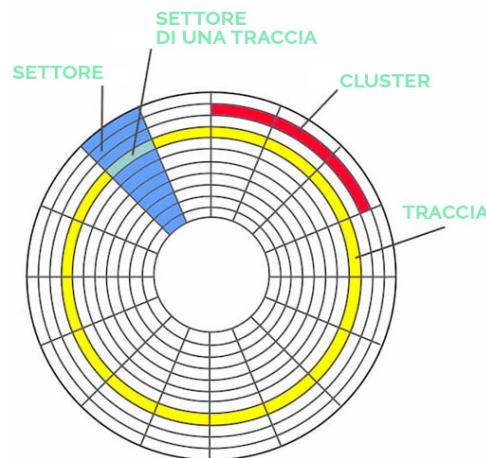


Figura 9.3: Organizzazione fisica di un **disco rigido**. Le tracce sono le sezioni *circolari* in cui è suddivisa la struttura del disco, i settori sono le sezioni *a spicchi* in cui è suddiviso il disco. Due settori adiacenti sono separati da uno spazio vuoto detto **intersector gap**.

Il numero di bit memorizzabili all'interno di ogni traccia è il medesimo per tutte le tracce. In questo modo è possibile semplificare la logica di controllo del dispositivo anche se, questo fatto, implica che i bit siano memorizzati con diversa densità man mano che ci si sposta verso le tracce con raggio minore. Il trasferimento tra i dati e la memoria principale e disco avviene in termini di unità di trasferimento di dimensione fissa corrispondenti ai settori nei quali è divisa ogni traccia. Per identificare i settori all'interno di una traccia, in fase di formattazione del disco, vengono memorizzati alcuni dati di controllo che consentono al controllore del disco di identificare l'inizio di ogni traccia e l'inizio e la fine di ogni settore. Inoltre, in molti dischi è possibile anche utilizzare entrambe le facce del piatto, raddoppiando così la sua capacità. Quindi, un singolo settore costituisce l'unità minima di trasferimento, ed è identificato

`(numero faccia, numero traccia, numero settore)`

Per semplificare il trasferimento tra memoria e disco normalmente tutti i settori vengono considerati come costituenti un array dove, quindi, ogni settore viene identificato tramite il proprio indice nell'array. Calcolato come  $i = \text{Numero Faccia} * N_t * N_s + \text{Numero Traccia} * N_s + \text{Numero Settore}$

## Efficienza nell'utilizzo del disco

Per valutare l'efficienza di uso di un disco viene spesso preso in considerazione un indicatore, detto **tempo medio di trasferimento** (TF), che corrisponde al **tempo mediamente necessario per leggere o scrivere una certa quantità di byte**, ad esempio i byte relativi ad un settore. Il valore di TF può essere scomposto in due quantità: **tempo necessario a portare la testina in corrispondenza dell'inizio del settore desiderato** e **tempo necessario a effettuare il vero e proprio trasferimento dei dati relativi al settore**:

$$TF = TA + TT = \text{Tempo Accesso} + \text{Tempo trasferimento}$$

Anche il **Tempo Accesso** è ulteriormente scomponibile in due componenti, a prima è il tempo necessario per posizionare la testina mobile sopra la traccia contenente il settore desiderato, detto **Seek Time** (o **ST**), e il tempo necessario per aspettare che il settore desiderato passi sotto alla testina, detto **Rotational Latency** (o **RL**). In termini matematici

$$TF = TA + TT = [ST + RL + TT]$$

Il **Tempo trasferimento** corrisponde al tempo necessario per far transitare sotto la testina l'intero settore e cioè, se indichiamo con  $t$  il tempo impiegato per far un giro completo e con  $s$  il numero di settori per traccia, possiamo approssimare TT come il rapporto tra  $t$  e  $s$ . Risulta però evidente che questa approssimazione è quantomeno scorretta, poiché non tiene conto del **gap** che esiste tra i vari settori, tuttavia però può essere una discreta misura per fare una stima. Quindi, il **tempo di trasferimento** dipende sostanzialmente dal **Seek Time** e dalla **Rotational Latency** e, di conseguenza, risulterà opportuno intervenire per ridurre TF.

## Algoritmi di schedulazione delle richieste

Tuttavia, fino ad adesso abbiamo considerato come unica metrica il **tempo medio di trasferimento**. In realtà, per valutare il tempo di attesa di un processo che ha richiesto di effettuare un trasferimento, dovremmo considerare anche il tempo che il processo resta sospeso nella coda di accesso al dispositivo. Risulta quindi ovvio che questa metrica dipenda strettamente dal criterio con cui viene gestita la coda di attesa. Una strategia molto semplice da implementare è quella di utilizzare una strategia **First-Come First-Served** che, fra l'altro, ha il privilegio di eliminare ogni condizione di starvation. Purtroppo, si nota che questa strategia non risponde a nessun criterio di ottimalità. Infatti, il parametro da ottimizzare in questo caso dovrebbe essere il numero degli spostamenti della testina, che deve essere portato al minimo, riducendo così il **Seek Time**. Analizzando graficamente il funzionamento di FCFS ci si rende conto che questo spostamento non viene minimizzato

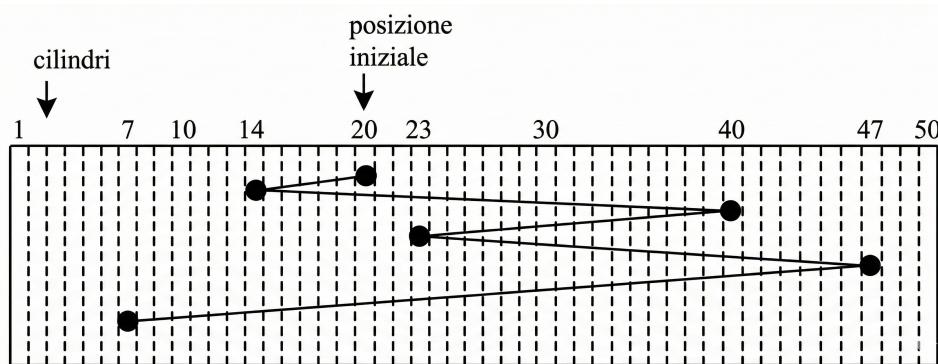


Figura 9.4: Algoritmo di scheduling FCFS

Un criterio molto più efficiente di FCFS potrebbe essere quello che, alla fine di un'operazione, tra le varie richieste pendenti, sceglie quello che richiede di operare sul cilindro più vicino alla posizione attuale della testina, in modo da minimizzare il **Seek Time** per ognuna delle operazioni pendenti.

L'algoritmo che sceglie le richieste pendenti secondo questo criterio è l'algoritmo **SSTF**, o **Shortest Seek Time First**. Questo algoritmo fornisce, almeno dal punto di vista computazionale, prestazioni migliori di quelle fornite da FCFS anche se non è detto che minimizzando il **Seek Time** per ognuna delle richieste pendenti si minimizzi il **Seek Time** medio del sistema. Inoltre, nel caso in cui si abbiano tante richieste in prossimità l'una delle altre e una singola richiesta che si trova in un settore *lontano*, si incorre nel rischio di **starvation**. Si osserva questo incremento di prestazione

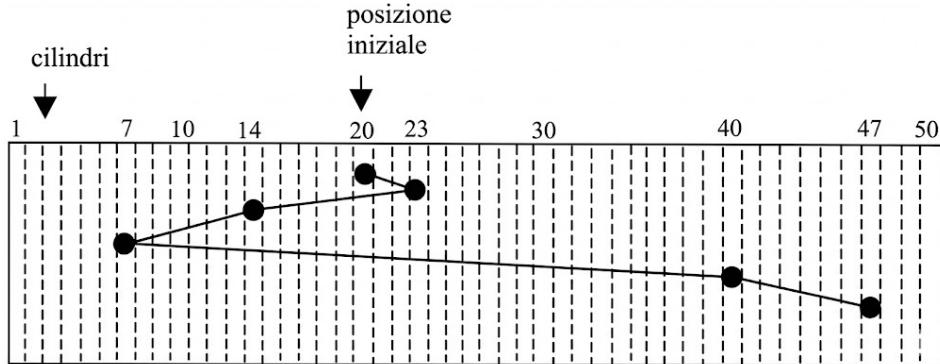


Figura 9.5: Algoritmo di scheduling SSTF

Un terzo algoritmo che risolve il problema della starvation è l'algoritmo **SCAN**. L'idea di questo algoritmo è che la testina proceda in un verso, gestendo le richieste man mano che le trova, e una volta arrivato all'altro bordo del disco, proceda nel senso opposto gestendo le richieste che incontra in ordine opposto. Ad esempio:

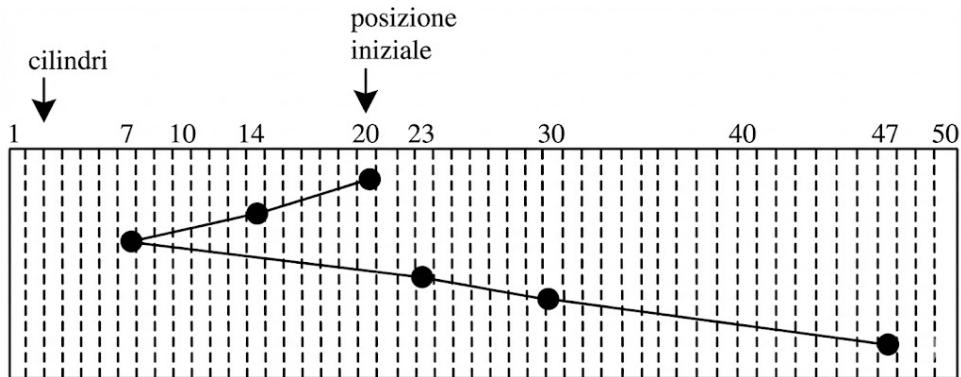


Figura 9.6: Algoritmo di scheduling SCAN

Come si nota abbastanza facilmente, l'algoritmo parte dal cilindro 50 e procede verso il bordo sinistro, gestendo le richieste in ordine decrescente (prima la richiesta 14 e successivamente la richiesta per il cilindro 7). Una volta arrivato al bordo sinistro, riparte verso il bordo destro, gestendo le richieste in ordine crescente (prima la 23, poi la 30, e così via).

**Rotational Latency** Il tempo medio di trasferimento è definito come la somma di tre diversi elementi, il primo è il **Seek Time**, il secondo è il **Tempo Trasferimento** e il terzo è la **Rotational Latency**. Abbiamo detto che il **Seek Time** è minimizzabile scegliendo l'opportuna politica di schedulazione delle richieste, il tempo di trasferimento è stimabile invece come il rapporto tra il tempo necessario a fare un giro diviso il numero di settori per traccia. La terza grandezza è invece la **Rotational Latency**, questa grandezza ha la particolarità di non essere stimabile, poiché dipende dalla posizione del blocco interessato appena la testina viene messa sopra la traccia corretta. Il caso peggiore è quello in cui l'inizio del blocco interessato ha già passato la testina nel momento in cui arriva in posizione.

# 10

## filesystem

Per garantire ad ogni processo l'accesso a grandi insiemi di dati e a informazioni che sopravvivano alla terminazione, o anche all'interruzione di un eventuale alimentazione elettrica, è necessario sfruttare dei dispositivi con caratteristiche di **persistenza** e di **grande capacità**. Questa memoria, come abbiamo ampiamente visto, viene detta **memoria secondaria**.

Lo scambio di dati tra **memoria primaria** e **memoria secondaria** richiede che vengano forniti dei meccanismi per l'accesso e l'archiviazione dei dati all'interno della suddetta memoria. Tutti questi meccanismi sono contenuti all'interno del **filesystem**; possiamo definire tale oggetto come: la parte dei sistema operativo che fornisce i meccanismi necessari per l'accesso e l'archiviazione delle informazioni in memoria secondaria. La struttura del filesystem può essere rappresentata da un insieme di componenti organizzati per livello:

- Il primo livello, detto **struttura logica**, presenta una visione astratta delle caratteristiche del dispositivo.
- Il secondo livello viene detto **accesso**.
- Il terzo livello rappresenta l'organizzazione **fisica**.
- Il quarto livello è il dispositivo virtuale, sotto al quale troviamo la memoria secondaria effettiva.

Il livello più alto del filesystem è la **struttura logica**, cioè la visione che viene data all'utente delle informazioni presenti sul disco, che prescinde dalle caratteristiche fisiche del dispositivo. Queste informazioni sono, in genere, memorizzate (e quindi organizzate) in **file**, o insiemi di file, detti *directory*. I processi, che quindi vedono la memoria persistente secondo l'organizzazione logica definita dal filesystem, possono accedere a questa struttura mediante delle operazioni chiamate **system call**. Il livello immediatamente **sottostante**, cioè il **livello di accesso**, **definisce** e **realizza** i meccanismi mediante i quali è possibile eseguire operazioni sul contenuto dei **file**. L'accesso ai **file** ha in genere due possibili finalità: **lettura** e/o **scrittura**. Il filesystem permette quindi di avere una visione astratta dell'organizzazione dei file all'interno della memoria persistente. A questo livello ogni file è visto come un insieme di **record logici**. Un **record logico** è caratterizzato da alcune proprietà come **tipo** e **dimensione** ed è, inoltre, definito come l'unità di trasferimento tra processo e file, cioè l'unità di informazione che ogni processo può estrarre o depositare ad o in ogni file a cui si accede. Il **livello di accesso** deve anche implementare i meccanismi di **protezione** dei vari file del sistema. Lo strato sottostante al livello di **accesso** è il livello di **organizzazione fisica**. Il compito di questo livello è quello di **allocare i record logici** di ogni **file** all'interno della memoria persistente; a questo livello lo spazio disponibile per l'allocazione viene visto come un insieme di **blocchi fisici**. Il **blocco fisico** è quindi l'unità di allocazione e di trasferimento delle informazioni sul dispositivo e ha, quindi, una propria posizione sulla superficie del disco. Il livello di **organizzazione fisica** implementa anche gli opportuni **metodi di allocazione** che stabiliscono il collegamento tra ogni file e l'insieme dei blocchi fisici sui quali esso è allocato.

# 1 Organizzazione logica del filesystem

Il file costituisce l'unità logica di memorizzazione all'interno del **filesystem**. Nella pratica è come se fosse un vero e proprio **contenitore di informazioni**. Ogni file è caratterizzato da un insieme di attributi:

- **Tip**. Stabilisce l'appartenenza a una particolare classe.
- **Indirizzo**. Puntatore alla memoria secondaria della posizione dove si trova il file.
- **Dimensione**. Numero di **record** contenuti nel file.
- **Data e Ora**.
- **Utente proprietario**.
- **Protezione**. Specifica tutte le politiche di accesso al file, relativamente agli utenti appartenenti allo stesso gruppo, al creatore del file e tutti gli altri utenti del sistema.

La directory è invece un'astrazione che consente di raggruppare più file. In pratica, è come se fosse un vero e proprio contenitore di file. Ogni directory può a sua volta contenere altre directory al suo interno, formando un vero e proprio albero di cartelle contenente i file della memoria secondaria. La struttura logica del filesystem è pertanto un'aggregazione di file e directory le cui caratteristiche dipendono dalle caratteristiche delle directory stesse. Nei sistemi operativi moderni la soluzione più comune è quella dei filesystem strutturati ad albero: in questo caso le directory possono contenere sia file, sia altre directory. Nei sistemi operativi basati su unix, più directory possono contenere il medesimo file; ciò è possibile attraverso un particolare meccanismo di **linking**.

Il sistema operativo realizza anche i meccanismi per la gestione del filesystem, che rende disponibili alle applicazioni mediante delle specifiche **system call**. Le operazioni fondamentali per la gestione del filesystem riguardano sia la **gestione dei file** e la **gestione delle directory**; possiamo distinguere:

- **creazione e cancellazione** delle directory.
- **aggiunta e cancellazione** di file.
- **listing**. Questa operazione consente di ispezionare il contenuto di uno o più directory, tramite la visualizzazione dell'elenco di tutti i file contenuti in essa.
- **attraversamento**. Questa operazione consente di *navigare* attraverso la struttura logica del **filesystem**, spostando la posizione corrente da un direttorio di origine a uno di destinazione.

# 2 Accesso al filesystem

L'insieme degli attributi di ogni file viene mantenuto all'interno di una particolare struttura dati, detta **descrittore di file**. Questo descrittore, così come anche il file, necessita di avere delle caratteristiche di **persistenza**, e pertanto viene mantenuto all'interno della memoria secondaria. Inoltre, siccome ogni **file** appartiene a una particolare **directory** è necessario che ognuna di esse abbia i collegamenti con i descrittori dei file contenuti al suo interno. Per risolvere questa problematica sistemi operativi diversi utilizzano delle strategie diverse

- Nei sistemi operativi **Windows** si utilizza una struttura di tipo tabellare che contiene un elemento per ogni file; ogni elemento include due campi: **nome** del file e **descrittore** del file.
- Nei sistemi operativi **Unix** l'insieme globale di tutti i descrittori è centralizzato all'interno di una particolare struttura dati, nota come **tabella dei descrittori di file**. La directory, in questo caso, è rappresentata da una struttura dati che ad ogni nome di file associa il riferimento ai suoi attributi nella tabella dei descrittori.

Nonostante il secondo meccanismo possa sembrare meno efficiente, permette però di implementare il meccanismo del linking. Infatti, se un file dovesse essere condiviso all'interno di due cartelle allora si sarebbe sufficiente che nella directory ci fosse il riferimento per il medesimo indice nella tabella globale. Il **filesystem** deve quindi permettere l'accesso *on-line* ai file, implementando determinate operazioni che devono essere consentite su di esso: **scrittura**, **lettura**, **scrittura in aggiunta**. Indipendentemente dal tipo di operazione è quindi necessario che, se un processo  $P$  vuole scrivere su un file  $f$ , abbia la possibilità di accedere al suo descrittore di file, che come abbiamo detto è localizzato all'interno della memoria secondaria. Per tale motivo, il sistema mantiene in memoria centrale una struttura dati, detta **tabella dei file aperti**, che contiene alcune informazioni associate ai file attualmente in uso, tipicamente: **copia del descrittore**, **puntatore al prossimo record logico da leggere/scrivere** (se l'accesso è di tipo **sequenziale**) e le informazioni relative al processo che accede al file. L'inserimento di un nuovo elemento all'interno di questa tabella viene detto **apertura** del file, che deve essere sempre effettuata prima di ogni sessione di accesso a quel particolare file. Al termine di ogni sessione di operazioni su un file si procede con la fase di **chiusura**, quindi la rimozione del file dalla **tabella dei file aperti**.

### Metodi di accesso ai file

Il **metodo di accesso** stabilisce le **modalità** con le quali i processi possono *leggere* o *scrivere* i file. Ogni **metodo di accesso** presuppone implicitamente una particolare organizzazione interna del file, che a questo livello viene visto come un insieme di **record logici** numerati:  $R_1, \dots, R_n$ . I **metodi di accesso** più diffusi sono:

- Accesso sequenziale.
- Accesso diretto.
- Accesso a indice.

Nella prima modalità di accesso, cioè quella **sequenziale**, si assume che ogni file sia organizzato come una **sequenza** di record logici. Pertanto, a questa sequenza di record logici contenuta all'interno del file deve essere applicata una **relazione d'ordine** che stabilisce univocamente il primo record, il secondo, e così via. La modalità di accesso sequenziale prevede quindi che tutti i record possano essere letti e/o scritti solo nell'ordine prefissato dalla relazione d'ordine. Ciò, ovviamente, implica che per accedere ad un record  $R_i$  sarà necessario accedere a tutti i record  $R_1, R_2, \dots, R_{i-1}$  che lo precedono nella sequenza.

La seconda modalità di accesso, cioè quella **diretta**, permette la scrittura e/o lettura di un qualsiasi record logico del file. A differenza del metodo di accesso **sequenziale** non è quindi necessario accedere a tutti i record precedenti al record interessato per poterlo modificare. Questo fatto, specie se il record da accedere si trova alla fine di una lunga sequenza, implica evidenti vantaggi dal punto di vista dell'efficienza. Possiamo quindi concludere che questo metodo risulta particolarmente vantaggioso nel momento in cui è necessario accedere a file di grandi dimensioni, di cui vogliamo modificare delle piccole porzioni.

L'ultima modalità di accesso, cioè quella a **indice**, prevede che ogni record logico sia strutturato in almeno due campi, uno dei quali contiene un'informazione (**chiave**) che serve ad identificare univocamente il record all'interno del file. Ad ogni file viene inoltre associata una struttura tabellare, detta **indice**, nella quale vi è un elemento per ogni chiave che contiene il riferimento al record all'interno del file. Quindi, per accedere ad un particolare file è sufficiente conoscere la sua **chiave** all'interno dell'**indice**.

## 3 Organizzazione fisica

Come abbiamo visto nella precedente sezione, a livello di struttura logica ogni file viene suddiviso in blocchi, detti **record logici**. Un qualsiasi programma caricato in memoria accederà a questi record, senza preoccuparsi di come sono organizzati all'interno della memoria secondaria. Tuttavia, è comunque importante studiare il meccanismo con cui i file sono memorizzati all'interno di questa memoria. All'interno della **memoria secondaria** definiamo un file come un'insieme di **blocchi**,

detti anche **record fisici**. Questi blocchi sono di dimensione costante e rappresentano l'unità minima di trasferimento nelle operazioni di **I/O** da e verso il disco. Analizzando questa struttura possiamo quindi notare la differenza che avevamo accennato all'inizio

- Il **filesystem** espone i file come insieme di record logici. I programmi accedono a dei record logici senza conoscere la struttura sottostante e l'organizzazione dei **record fisici**.
- All'interno della **memoria secondaria** i file sono organizzati in **record fisici**, quindi blocchi di dimensione costante allocati secondo delle politiche specifiche.

Possiamo fare un'analogia con il meccanismo della paginazione visto nello scorso capitolo, tuttavia con una sostanziale differenza. Mentre pagine e frame hanno la medesima dimensione, nel caso dei dischi non possiamo dire la stessa cosa, infatti i **record fisici** hanno dimensione **sempre maggiore** dei **record logici**. Possiamo quindi supporre che all'interno di un singolo blocco possano essere memorizzati più record logici; infatti, detta  $D_b$  la dimensione di un blocco e detta  $D_r$  la dimensione di un record possiamo definire il numero di record all'interno di un blocco  $N_b$  come

$$\text{Numero record} = N_b = \frac{\text{Dimensione blocco}}{\text{Dimensione record}} = \frac{D_b}{D_r}$$

Il motivo di questa scelta è infatti dato alla poca efficienza del disco, facendo dei blocchi molto piccoli, magari della stessa dimensione dei record, il disco potrebbe impiegare tantissimo tempo (dovrebbe infatti trascorrere **Seek Time** per poter spostare la testina tra i vari settori) per caricare tutti i record logici relativi ad un file; si preferisce quindi fare in modo che con un singolo spostamento della testina si riesca a prendere più record insieme. Lo spazio per l'allocazione dei file è visto come un vettore lineare di blocchi, ognuno dei quali può essere utilizzato per l'allocazione di un sottoinsieme di record logici appartenenti allo stesso file. Esistono tantissimi approcci possibili riguardo a come scegliere i blocchi su cui allocare i file; in generale distinguiamo tre possibili approcci: allocazione **contigua**, allocazione a **lista concatenata** e allocazione **a indice**

### 3.1 Allocazione contigua

Il metodo di allocazione contigua prevede che ogni file occupi un insieme di blocchi fisicamente contigui. L'idea è quindi relativamente semplice e permette una gestione semplice ed efficiente dell'accesso sequenziale. Seguendo questo approccio è anche possibile realizzare in modo efficiente anche il metodo di accesso diretto. In questo caso, infatti, il descrittore di ogni file mantiene l'indirizzo **B** del primo blocco utilizzato per l'allocazione; l'indirizzo del blocco su cui è allocato un particolare record logico viene quindi ottenuto direttamente mediante la formula

$$R_i = B + \frac{i}{N_b}$$

Dividendo infatti  $i$ , cioè il record che cerchiamo, per il numero di blocchi  $N_b$  otteniamo la **posizione relativa** del record all'interno della sequenza dei blocchi; sommando successivamente per  $B$ , quindi la posizione del primo blocco, troviamo la posizione esatta del record  $i$ . Tuttavia, questo metodo di allocazione presenta alcuni svantaggi, legati soprattutto alla gestione dello spazio disponibile:

- Il primo problema deriva proprio dal fatto che il costo per cercare l'insieme dei blocchi nel quale allocare un nuovo file cresce all'aumentare della dimensione del file in modo lineare. Per ogni nuova allocazione è infatti necessario cercare un insieme di blocchi liberi sufficientemente grande.
- Il secondo problema deriva dal fatto che raramente sarà possibile trovare una zona di dimensioni esatte di quelle del file, tendenzialmente si troveranno zone di maggiore dimensione, creando quindi il problema della frammentazione esterna.

Ricordiamo che sul primo punto è anche importante quindi operare una corretta scelta della politica di allocazione; abbiamo infatti visto che esistono diverse possibili: **best-fit**, **first-fit** e anche **worste-fit**. Per quanto riguarda il secondo problema, è possibile implementare un meccanismo di compattamento per cercare di compattare le locazioni libere intervallate da locazioni occupate, così da ottenere un'unica sequenza di locazioni libere; il compattamento introduce però un problema relativo al costo computazionale dell'operazione.

### 3.2 Allocazione a lista concatenata

La tecnica di allocazione a **lista** memorizza ogni **file** in un insieme di **blocchi non contigui** organizzati in una **lista concatenata**. L'organizzazione dei **blocchi** utilizzati per uno stesso file è quindi ancora **sequenziale**, ma blocchi successivi non devono essere necessariamente adiacenti. Per permettere l'accesso a file allocati con questo modo è quindi necessario mantenere nel **descrittore di file** il **puntatore al primo blocco utilizzato**, mentre il collegamento tra blocchi successivi della lista viene realizzato riservando all'interno di ogni blocco utilizzato lo spazio per il puntatore al blocco successivo nella lista.

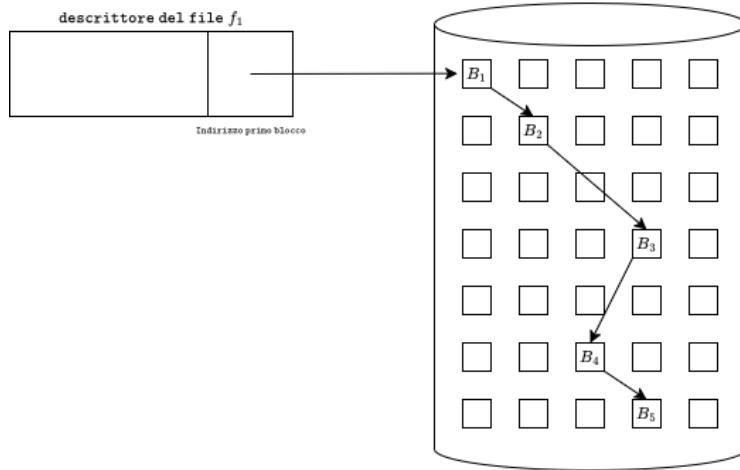


Figura 10.1: Allocazione a lista concatenata

Utilizzando l'allocazione a lista concatenata l'accesso sequenziale risulta poco costoso in termini realizzativi, infatti, nel momento in cui accedo al blocco  $i$ -esimo, avrò già fatto l'accesso a tutti i blocchi precedenti  $j$ -esimi, con  $j < i$ . La realizzazione dell'accesso diretto è invece molto costosa in termini di numero di accessi in memoria, poiché per accedere al blocco  $i$ -esimo è comunque necessario fare un accesso sequenziale a tutti i blocchi precedenti, con un numero di accessi in memoria pari a:

$$\text{Accessi in memoria} = \frac{\text{indice blocco da visitare}}{\text{numero di blocchi}} = \frac{i}{N_b}$$

Tuttavia, un'allocazione a lista concatenata elimina del tutto il problema della frammentazione esterna, non essendo infatti necessario che le locazioni siano contigue, è possibile utilizzare i gap che si formano tra le varie locazioni occupate, senza doverle compattare.

#### Problema della robustezza

Se però da una parte risolviamo il problema della frammentazione esterna, dall'altra introduciamo un altro problema, relativo alla perdita di informazioni. Ipotizziamo infatti per colpa di un guasto o di un errore software, il valore di un puntatore all'interno del blocco  $i$  si altera. La conseguenza è che tutti i blocchi  $b_j$  con  $j > i$  sono *persi*, nel senso che non sono più accedibili a partire da  $b_i$ , ma non sono comunque allocabili, poiché contengono delle informazioni relative ad un file. Per arginare questo problema è possibile realizzare una soluzione con doppio puntatore: ogni blocco contiene i puntatori al **blocco successivo** e al **blocco precedente**; ovviamente una soluzione di questo tipo introduce dei costi maggiori per l'allocazione. Un'altra possibile soluzione per risolvere questo problema, che è la soluzione adottata da windows, è quella di predisporre una particolare tabella di allocazione, detta **FAT**, memorizzata in una posizione predefinita sul dispositivo virtuale, che contiene un elemento per ogni blocco del dispositivo, specificando se il blocco è libero o meno e la posizione del blocco successivo nella lista. Grazie alla FAT è quindi possibile, in caso di perdita di un puntatore, ripristinarlo correttamente. Inoltre, questa tabella può essere copiata dal sistema in memoria centrale o in una cache: in questo modo è possibile

velocizzare notevolmente l'accesso diretto grazie al fatto che l'indirizzo del blocco da accedere può essere ottenuto senza accessi preliminari al disco.

### 3.3 Allocazione a indice

L'allocazione a indice si basa ancora sull'utilizzo di blocchi non contigui per l'allocazione di file. In particolare, questo metodo prevede che per ogni file sia definito un blocco **indice** che contiene gli indirizzi dei blocchi effettivamente utilizzati per l'allocazione del contenuto del file. Con questa tecnica, come nel caso dell'allocazione a lista, si elimina il problema della frammentazione interna. In questo caso il descrittore del file contiene l'indirizzo del blocco **indice**, accedendo al quale è possibile ottenere l'indirizzo di ogni altro blocco utilizzato per l'allocazione del file. Un approccio di questo tipo ha però un problema in termini di scalabilità: essendo infatti la dimensione di un blocco fissa, è possibile soltanto definire un massimo numero di blocchi per ogni file, limitando quindi la possibilità di creare file di una certa dimensione.

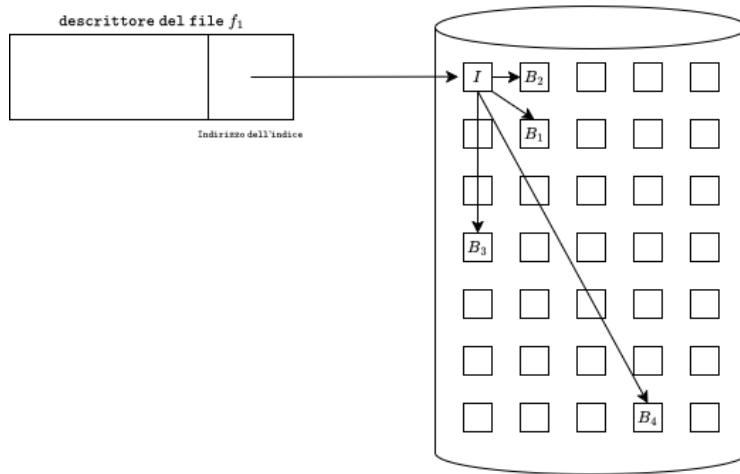


Figura 10.2: Allocazione a indice

Possiamo quindi provare a dare una stima della dimensione massima di un file, ipotizzando ovviamente di allocare i nostri blocchi con un'allocazione a indice. Dato un disco di capacità  $C$  e un blocco di dimensione  $D_B$  possiamo stimare quanti bit sono necessari per indicizzare tutti i blocchi di dimensione  $D_B$  in modo molto semplice

$$\text{Dimensione puntatore [bit]} = D_P = \log\left(\frac{C}{D_b}\right)$$

Una volta noto il numero di bit necessari a indicizzare i nostri blocchi possiamo calcolare quanti puntatori possiamo mettere all'interno di un blocco. Questo numero è, semplicemente, definito come il rapporto tra la dimensione di un blocco  $D_B$  e la dimensione di un singolo puntatore

$$\text{Numero puntatori} = \frac{\text{Dimensione blocco}}{\text{Dimensione puntatore}} = \frac{D_b}{D_p}$$

Se abbiamo, ad esempio,  $2^k$  blocchi, ci serviranno indirizzi di  $k$  bit di dimensione. Conoscendo quindi il numero dei puntatori è sufficiente moltiplicare per la dimensione del blocco nuovamente, ottenendo quindi la dimensione massima di un file all'interno del nostro sistema.

$$D_M = \text{Numero puntatori} \cdot D_B = \frac{D_b^2}{D_p}$$

## 4 filesystem di unix

Analizziamo ora un esempio di struttura di un filesystem reale, cioè quello di unix. Una delle caratteristiche fondamentali dei sistemi operativi **unix-based** è l'**omogeneità**; in particolare, in

unix ogni risorsa è rappresentata all'interno del **filesystem** sotto forma di **file**. Vista questa peculiare proprietà è necessario che i file siano categorizzati in funzione del loro scopo, in altri termini, è necessario che ad ogni file sia associato un tipo tra:

- **File ordinario.** Rappresentano un insieme di informazioni che sono effettivamente allocate in **memoria di massa**.
- **File speciale.** Rappresentano un dispositivo fisico come, ad esempio, un mouse o una tastiera.
- **File directory.** Rappresentano il concetto astratto di **directory**. Questo concetto è realizzato memorizzando all'interno del file la descrizione dell'insieme dei file e delle directory in esso contenute.

Il filesystem di unix ha una struttura gerarchica, rappresentabile sotto forma di un **grafo aciclico diretto**. All'interno di questo grafo le **directory** rappresentano i nodi dell'albero, mentre i **file** rappresentano le foglie di quest'ultimo. L'intero **filesystem** di unix è allocato all'interno di un unico dispositivo fisico, formattato in blocchi fisici di dimensione costante e prefissata e suddiviso in 4 regioni

Regione	Descrizione
<b>BootBlock</b>	Allocato a un indirizzo prefissato, viene utilizzato dal programma di inizializzazione del sistema
<b>SuperBlock</b>	descrive l'allocazione del filesystem; in particolare, contiene i limiti delle 4 regioni, il puntatore alla lista dei blocchi liberi e il puntatore alla lista degli <b>i-node</b> liberi.
<b>DataBlocks</b>	rappresenta la zona del disco effettivamente disponibile per la memorizzazione dei file.
<b>i-list</b>	contiene il vettore di tutti i descrittori ( <b>i-node</b> ) dei file, directory e dispositivi presenti nel filesystem. ogni <b>i-node</b> viene riferito mediante l'indice <b>i-number</b> che lo identifica univocamente

Tabella 10.1: Organizzazione fisica del filesystem di unix

Un **i-node** è quindi un vero e proprio descrittore di file, per cui contiene tutti gli attributi associati al file, tra cui: **tipo del file**, **proprietario** del file e **gruppo** del file, **dimensione** in termini di **numero di blocchi occupati**, **link**, quindi il **numero di nomi** che riferiscono il file, i **12 bit di protezione** del file e il **vettore di indirizzamento**, cioè l'insieme degli indirizzi che consente l'indirizzamento dei blocchi di dati sui quali è allocato il file.

### Allocazione dei file in unix

Unix utilizza un meccanismo di allocazione di blocchi fisici a **indice**, tuttavia con una piccola variazione rispetto a quanto detto nella scorsa sezione; l'allocazione a indice di linux si basa su una struttura a **più livelli di indirizzamento**. I vettori di indirizzamento contenuti all'interno dell'**i-node** sono infatti dei veri e propri puntatori a elementi contenuti dell'area **DataBlocks** del disco; in particolare, ipotizzando che questo vettore abbia 13 indirizzi, possiamo utilizzare la seguente suddivisione: i primi 10 elementi contengono al loro interno riferimenti per dei blocchi dati, che possono essere **direttamente** impiegati per l'allocazione dei file, l'undicesimo indirizzo punta a un blocco che contiene, a sua volta, degli **indirizzi di blocchi per l'allocazione di dati** (primo livello di **indirettezza**), il dodicesimo indirizzo contiene l'indirizzo di un blocco che contiene dei blocchi dati (primo livello di **indirettezza**) e, infine, il tredicesimo indirizzo consente l'accesso ai blocchi dati mediante tre livelli di **indirettezza**.

## 4.1 Strutture dati per l'accesso ai file

All'interno di unix ogni file è organizzato come una sequenza di byte: il **record logico** è quindi il **byte**. Il file può essere aperto in diverse modalità: scrittura, lettura, aggiunta. Inoltre, ogni sessione di accesso deve essere preceduta dall'operazione di **apertura** del file, mediante la quale vengono aggiornate le strutture dati del kernel. Il metodo di accesso adottato da **unix** è di tipo **sequenziale**: ad ogni file **aperto** è associato un particolare **I/O-Pointer**, che indica implicitamente il prossimo elemento a cui accedere. Ogni **lettura/scrittura** di un record logico del file provoca l'avanzamento dell'**I/o-Pointer** all'elemento successivo nella sequenza. A livello globale il kernel mantiene una tabella dei file aperti di sistema, detta **TFAS**: questa struttura dati contiene un elemento per ogni file aperto nel sistema. Più precisamente, viene allocato un elemento nella TFAS per ogni file che viene aperto; ciò implica che se un file viene aperto da due processi, vi saranno due entrate all'interno della tabella (mentre un processo generato con la `fork()` condividerà l'entrata con il padre). Ogni elemento della TFAS contiene un riferimento all'**i-node** (descrittore di file) aperto, che il sistema copia e mantiene in memoria centrale per tutta la durata della sessione di accesso al file, insieme anche all'**I/O-pointer** per quella particolare sessione. Gli **i-node** dei file aperti sono inseriti all'interno di un'altra struttura dati globale: la **Tabella Dei file Attivi**. Oltre a queste due tabelle, a ogni processo è associata una **tabella dei file aperti del processo**, organizzata a vettore: ogni elemento è identificato da un indice intero, che prende il nome del **file descriptor**. All'avvio di ogni processo il sistema apre automaticamente i file per lo `stdin`, `stdout` e `stderr`. Ogni elemento della TFAP contiene un riferimento all'elemento corrispondente nella TFAS. Inoltre, la TFAP è una struttura accessibile soltanto al kernel e pertanto è collocata nella user-structure del processo. Possiamo anche definire uno schema che collega le tre tabelle

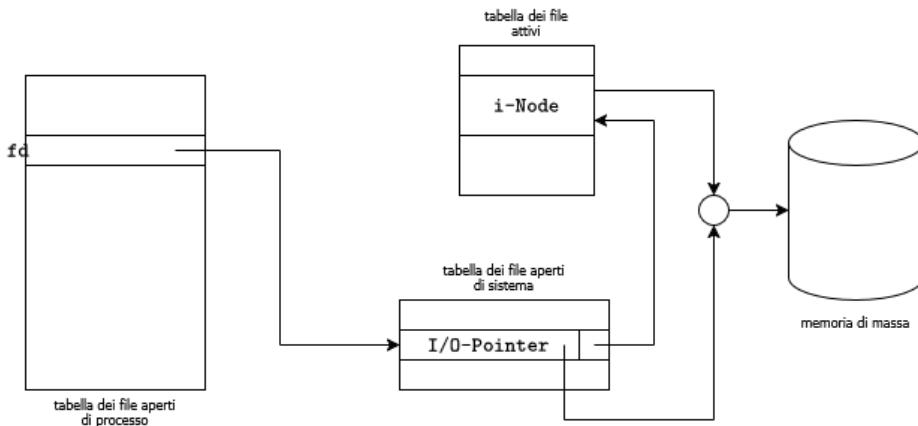


Figura 10.3: Gestione dell'accesso ai file in unix

Al momento di una lettura, ad esempio `read(fd, buffer, 100)`, succede quindi: il kernel prende il valore `fd`, accede alla TFAP e trova il puntatore alla TFAS. Nella TFAS legge il valore di **I/O-Pointer** (byte x-esimo), segue il puntatore all'**i-node** all'interno della TFA. Usa **i-node** per tradurre **byte** x-esimo nel blocco fisico (esempio `y`) in cui si trova. Comanda al disco di leggere il blocco `y` e, infine, aggiorna l'**I/O-pointer**.

# 11

## Meccanismi di protezione

Uno dei concetti fondamentali all'interno dei **sistemi operativi** è quello di **protezione**, cioè l'insieme delle attività che si preoccupano di garantire, a **runtime**, il **controllo dell'accesso** alle risorse **logiche** e **fisiche** del sistema. In altri termini, intendiamo l'insieme dei meccanismi che il sistema operativo mette a disposizione per assicurare che gli accessi alle risorse avvengano nel rispetto di determinate politiche. Insieme alla **protezione** troviamo anche il concetto di **sicurezza**, cioè la necessità di autenticare gli utenti impedendo gli accessi non autorizzati al sistema o tentativi di alterazione dolosi.

Durante tutto faremo principalmente riferimento ai meccanismi di protezione, mentre la sicurezza è lasciata a studi successivi. Il **controllo degli accessi**, cioè il concetto alla base dei meccanismi di **protezione**, è suddivisibile in tre **livelli concettuali**:

- **Modelli.** Il modello deve definire le relazioni che esistono tra le entità del sistema.
- **Politiche.** Le politiche determinano i diritti di accesso che sono assegnati alle entità definite dal modello.
- **Meccanismi.**

In particolare, un **modello di protezione** suddivide le entità del sistema in **soggetti** e **oggetti**, dove i soggetti hanno accesso agli oggetti con dei **diritti di accesso**, cioè le operazioni con le quali si può accedere agli oggetti. I **soggetti** sono la parte attiva del sistema, cioè i **processi** che agiscono per conto degli utenti per accedere a determinati oggetti. Un **soggetto** non è quindi il solo **processo**, cioè il PID, ma considera anche **chi** ha creato il processo e il suo **gruppo di appartenenza**; in altri termini, un soggetto in un sistema operativo **unix-based** potrebbe essere definito come:

$$S_i = \langle \text{UID}, \text{GID}, \text{PID} \rangle$$

Dove UID è l'**user ID**, il GID è **group ID** e il PID corrisponde al **process ID**. Gli **oggetti** sono la parte **passiva** del sistema, cioè le **risorse fisiche** e le **risorse logiche**. Ogni oggetto ha un **nome unico** e un insieme di operazioni, che dipendono dal **tipo** di oggetto, con le quali i soggetti possono accedervi.

Un **soggetto** generico  $S_i$  può avere **diritti d'accesso** sia per quanto riguarda le risorse  $R_i$ , sia per quanto riguarda altri **soggetti**  $S_j$ . Un soggetto può essere considerato come una coppia (**processo, dominio**), dove dominio è l'ambiente di protezione nel quale il soggetto sta eseguendo, in altri termini, il dominio specifica il perimetro entro il quale il processo può muoversi. Un dominio di protezione è **unico** per un **soggetto**, ma non lo è per un **processo**; infatti il soggetto  $S_i$  può rappresentare il processo  $P_i$  in un particolare **dominio di protezione**, ed il soggetto  $S_j$  può rappresentare il processo in un altro dominio di protezione.

# 1 Politiche e meccanismi

Le politiche di protezioni definiscono le regole con le quali i soggetti possono accedere agli oggetti e ai soggetti, cioè quali diritti di accesso ogni soggetto può esercitare nei confronti di ogni altro soggetto o oggetto. In altri termini, sono insiemi di regole che definiscono il perimetro nel quale il soggetto può muoversi. Distinguiamo tre politiche per l'accesso:

- **Discretionary Access Control** (o DAC). Il creatore di un oggetto controlla i diritti di accesso per quell'oggetto.
- **Mandatory Access Control** (o MAC). I diritti di accesso vengono gestiti centralmente.
- **Role Based Access Control** (o RBAC). Ad un ruolo sono assegnati specifici diritti di accesso sulle risorse.

Mentre la DAC è tipica dei sistemi operativi, così come anche **unix**, MAC e RBAC sono politiche di protezione tipiche delle organizzazioni, dove i soggetti non sono proprietari dei file che richiedono, ma è l'organizzazione stessa ad essere proprietaria dei file. Tuttavia, comunemente a tutte queste politiche, troviamo un principio comune, detto **principio del privilegio minimo**: a un **soggetto** sono garantiti i **diritti di accesso** solo agli **oggetti** strettamente necessari per la sua esecuzione. Per imporre una determinata politica è necessario che il sistema operativo metta a disposizione degli strumenti; questi strumenti prendono il nome di **meccanismo**. Nel capitolo sullo scheduling era già stata definita la differenza tra **politiche** e **meccanismo**; la politica definisce **cosa** va fatto, mentre il **meccanismo** come va fatto.

## 2 Domini di protezione

Come detto anche precedentemente un soggetto è un processo che opera in un dominio di protezione che specifica gli oggetti accessibili dal processo. Ogni dominio definisce, cioè, un insieme di oggetti e i tipi di operazioni che si possono eseguire su ciascun oggetto; in altri termini, un **dominio** è una doppia che associa ad un **oggetto** un **insieme di regole**

(Nome oggetto, Diritti di accesso)

Un soggetto può accedere solamente agli oggetti definiti nel dominio. Si possono avere anche domini **disgiunti** o domini con **diritti di accesso in comune**. L'associazione tra dominio e processo può essere di due tipologie: **statica** se l'insieme delle risorse disponibili a un processo rimane fisso durante il suo tempo di vita, **dinamica** se l'associazione tra **processo** e **dominio** varia durante l'esecuzione del processo. L'associazione statica è però in contraddizione con il principio del minimo privilegio in quanto, in questo caso, il dominio deve contenere tutti gli oggetti ai quali il processo accede durante tutta la sua esecuzione. Più comunemente l'associazione tra processo e dominio è dinamica; un processo può cioè cambiare dominio durante la sua esecuzione. Le regole per il cambio di dominio dipendono dal particolare **sistema di protezione**.

### Domini di protezione in unix

Il sistema di protezione tipico dei sistemi **unix** fornisce un'idea concreta di un dominio di protezione. A ogni processo sono associati un **Group ID (GID)** e **User ID (UID)**, corrispondenti a quelli dell'utente che ha chiesto la creazione del processo. Tale coppia definisce il dominio di protezione del processo, dove, in corrispondenza di questi due elementi risultano quindi i file a cui il processo è in grado di accedere e i relativi diritti di accesso. Due processi con la medesima coppia (UID, GID) avranno quindi il medesimo dominio di protezione e potranno, pertanto, accedere alle stesse identiche risorse e con i medesimi diritti di accesso. Tuttavia, come abbiamo detto, è possibile che durante l'esecuzione di un processo il dominio di protezione cambi; unix infatti associa ad ogni eseguibile un bit **SUID** che, nel caso in cui abbia valore pari ad uno, determinare che l'esecuzione del file avvenga nel dominio associato all'utente proprietario dell'eseguibile stesso. Quindi, se un processo che esegue in un dominio  $D_1$  passa a eseguire il programma contenuto in un file  $F$ , di proprietà dell'utente  $D_2$  e con il bit **SUID** di valore pari ad uno, l'effetto ottenuto sarà il passaggio, temporaneo, del processo dal dominio di protezione  $D_1$  al dominio di protezione  $D_2$ .

### 3 Matrice degli accessi

Come abbiamo visto, un modello di protezione si compone di un insieme di **soggetti** e **oggetti**, dove i **soggetti** possono accedere agli **oggetti** rispettando una particolare politica di protezione. Un sistema di protezione può essere rappresentato utilizzando il modello **matrice degli accessi**. Il modello mantiene tutta l'informazione che specifica il tipo di accessi che i soggetti hanno verso gli oggetti, che prende il nome di **stato di protezione**, e consente di:

- Rappresentare lo stato di protezione.
- Garantire il rispetto dei vincoli di accesso per ogni tentativo di accesso di un soggetto ad un oggetto.
- Permette la modifica controllata dello stato di protezione determinando una transizione di stato.

Nel modello lo stato di protezione è rappresentato come una matrice degli accessi  $A$ , dove l'elemento  $A[S, X]$  identifica i diritti che il soggetto  $S$  ha sull'oggetto  $X$ . L'accesso a un qualunque oggetto implica quindi una serie di passi: il soggetto  $S_i$  vuole compiere un'operazione  $M$  sull'oggetto  $O_j$ , viene quindi generata una tripla  $(S_i, M, O_j)$  che viene passata al meccanismo di protezione. Il meccanismo di protezione interroga la matrice degli accessi, in particolare valuta se l'elemento  $A[S_i, O_j]$  contiene tra le operazioni possibili  $M$ , allora il soggetto può eseguire l'operazione.

#### Modifica dello stato di protezione

La modifica dello stato di protezione può essere ottenuta mediante un opportuno insieme di comandi che possono essere utilizzati come base per modellare diversi sistemi di protezione. La loro esecuzione deve essere compatibile con la particolare politica di protezione che si vuole realizzare. In altre parole, anche la matrice degli accessi è un oggetto protetto, controllato a sua volta da un meccanismo di protezione. Il principale insieme dei comandi per la modifica dei diritti di accesso è quello detto **Graham-Denning**. Possiamo distinguere due classi di comandi:

- **Comandi di propagazione dei diritti d'accesso.** La possibilità di copiare un diritto di accesso per un oggetto  $O_j$  da un dominio ad un altro della matrice di accesso. In altri termini, un soggetto  $S_i$  può trasferire un diritto di accesso  $\alpha$  per un oggetto  $X$  ad un altro soggetto  $S_j$  solo se  $S_i$  ha accesso a  $X$  e il diritto  $\alpha$  ha il copy flag (quindi può essere trasferito). Per verificare se un particolare diritto di accesso ha il copy flag è sufficiente verificare se il diritto è scritto nella forma  $\alpha^*$ . Ovviamente la propagazione può avvenire in due modi, è possibile copiare il diritto oppure è possibile copiare il diritto più anche il copy flag.
- **Comandi per l'assegnazione dei diritti d'accesso.** La possibilità di assegnare un diritto  $\alpha$  per un oggetto  $X$  da un soggetto  $S_j$  a un soggetto  $S_i$ . L'autorizzazione all'esecuzione del comando stabilisce che  $S_i$  può eliminare un diritto di accesso da  $S_j$  se e solo se  $S_i$  controlla  $S_j$  oppure se  $S_i$  è proprietario di  $X$ .

#### 3.1 Realizzazione della matrice degli accessi

Il modello generale della matrice degli accessi presenta diverse problematiche, alcune delle quali non indifferenti, sia dal punto di vista realizzativo che dal punto di vista funzionale. Il primo problema deriva dalla dimensione, infatti in sistemi con tante risorse è facile che le dimensioni di questa matrice possano diventare anche abbastanza grandi e pesanti da memorizzare. Il secondo problema deriva dal fatto che la matrice potrebbe essere **sparsa**: generalmente un processo non richiede molte risorse e, inoltre, le risorse sono generalmente accedibili da pochi processi. Su adottano quindi due diversi modi di rappresentazione delle informazioni contenute nella matrice. La prima di queste modalità è quella della **Access Control List**; questo tipo di rappresentazione suddivide la matrice per colonne: per ogni oggetto è associata una particolare **Access Control List** che contiene tutti i soggetti dai quali è possibile accedere all'oggetto e per ogni soggetto i

diritti di accesso all'oggetto. La **lista di controllo** degli accessi per ogni oggetto è rappresentata dall'insieme delle coppie ordinate

(soggetto, insieme dei diritti)

Quando un soggetto  $S_i$  tenta di esercitare un diritto di accesso  $M$  sull'oggetto  $O_j$ , il meccanismo di protezione verifica se nella ACL associata a  $O_j$  esiste una coppia il cui primo elemento sia  $S_i$  e se tra i diritti di accesso contenuti nel secondo elemento della coppia ci sia  $M$ . In unix la protezione viene gestita semplificando i soggetti in solamente tre: **proprietario**, **membri del gruppo proprietario** e, infine, **altri soggetti**. Inoltre, per ogni soggetto è possibile definire solo tre diverse operazioni: **read**, **write** e **execute**. In alcuni casi potrebbe essere comodo comodo assegnare a un soggetto differenti diritti di accesso a seconda del gruppo cui appartiene, cioè del tipo di funzione che gli è richiesto. In questo caso, indicando con **UID** e **GID** gli identificatori di soggetto, l'entry dell'ACL ha la forma

$$\begin{aligned} \text{UID1, GID1 : } & \langle \text{insieme dei diritti} \rangle \\ \text{UID2, GID2 : } & \langle \text{insieme dei diritti} \rangle \end{aligned} \quad (1)$$

Uno degli svantaggi del metodo delle ACL deriva dall'inefficienza con cui il sistema operativo determina i diritti di accesso di un soggetto per un particolare oggetto. Si tratta infatti di esaminare il contenuto della ACL associata all'oggetto per verificare se nella lista è presente il soggetto in questione. Tuttavia, in compenso risulta semplice la revoca dei diritti di accesso per un oggetto. Si fa riferimento alla ACL associata all'oggetto e si cancellano i diritti di accesso che si vogliono revocare. Una diversa rappresentazione della matrice degli accessi è quella che la vede, invece che suddivisa per colonne, suddivisa per righe. Questa tecnica di rappresentazione della ACL è detta **Capability List**. In questo tipo di rappresentazione a ogni soggetto è associata una lista che contiene gli oggetti accessibili dal soggetto e i relativi diritti di accesso. Ogni elemento di questa lista prende il nome di **capability** e associa ad ogni soggetto, la lista degli oggetti a cui può accedere e i rispettivi diritti di accesso. Quando un soggetto  $S$  intraprende un'azione  $M$  su un oggetto  $O_j$  è necessario che il meccanismo di protezione controlli se nella lista delle **capability** per il soggetto  $S$  è presente l'oggetto  $O_j$  con dei diritti di accesso sufficienti per poter eseguire l'azione  $M$ . Di solito una capability si compone di un identificatore associato all'oggetto e una mappa di bit per i vari diritti; l'indice viene anche detto, su unix, **i-number**. Occorre prestare attenzione al fatto che le capability non sono direttamente accessibili dal soggetto interessato, ma sono gestite e mantenute dal sistema operativo stesso. A differenza delle ACL è quindi più facile risolvere il problema del conoscere i diritti di accesso ad un oggetto per un particolare soggetto, mentre è di più difficile risoluzione il problema della revoca dei diritti. Possiamo quindi dire che ACL e CL abbiano delle proprietà complementari e, infatti, in alcuni sistemi operativi possono essere usate in contemporanea. In particolare, quando un soggetto tenta per la prima volta di accedere a un oggetto, viene fatta una ricerca nella ACL associata all'oggetto: se l'accesso è consentito viene creata una capability che si associa al soggetto e gli accessi successivi vengono fatti utilizzando la capability per verificare se l'accesso è consentito.

### Gestione delle ACL e CL in unix

In unix, all'interno dei ogni **i-node** sono specificati una serie di bit che determinano, per ogni soggetto, i diritti di accesso a quel particolare oggetto; questa è di fatto l'**ACL**. Ogni volta che un file viene aperto da un processo, verrà copiato (se non presente) il descrittore di file all'interno della **tabella dei file attivi** in memoria centrale e verrà creata un'entrata all'interno della TFAS e della TFAP. Il sistema accede quindi alla tabella dei file attivi, preleva i diritti di accesso all'oggetto per il soggetto che ha richieste l'esecuzione del processo e li inserirà all'interno dell'entrata della TFAS; questa è la **CL**.

## 4 Sicurezza multilivello

Con la matrice di accesso è possibile descrivere in modo dinamico la sicurezza di un sistema con un approccio del tipo "chi può fare cosa". Questo tipo di approccio, quindi quello **DAC**, è

tipico dei sistemi operativi, dove i singoli utenti possono decidere delle proprie risorse e dei propri file. Esistono però delle organizzazioni dove i soggetti non sono diretti proprietari dei file, ma è l'organizzazione stessa ad esserlo, per cui è necessario definire dei meccanismi di protezione più sofisticati. Una politica di sicurezza di tipo **MAC** è quella che va sotto al nome di **politica di sicurezza multi-livello** dove il vincolo della segretezza delle informazioni è particolarmente stringente ed è pertanto richiesto di regolarne in modo opportuno il flusso tra i soggetti interessati. Alla base di questo meccanismo vi è una suddivisione dei soggetti in funzione del proprio livello di sicurezza delle informazioni in esse contenute:

- Non classificato.
- Confidenziale.
- Segreto.
- Top Secret.

I vari soggetti vengono assegnati alle stesse classi a seconda del tipo di documenti che possono accedere. Ogni processo eseguito per conto di un soggetto assume il suo livello di sicurezza. Un modello molto conosciuto per il controllo del flusso di informazioni è quello di **Bell-La Padula**, progettato per gestire la sicurezza in ambiente militare. Per ottenere l'obiettivo desiderato, cioè quello di evitare che un'informazione a livello  $k$  sia inaccessibile a livello  $i < k$ , è possibile implementare due semplici regole:

- **Proprietà semplice di sicurezza.** La lettura dell'informazione contenuta in un oggetto  $X$  da parte di un soggetto  $S$  richiede che  $\text{SC}(S) \geq \text{SC}(x)$ , cioè un soggetto può accedere solo a informazioni contenute in oggetti che appartengono a classi con livello di sicurezza minore o uguale al suo.
- **Star property.** La scrittura di informazione in un oggetto  $X$  da parte di un soggetto  $S$  richiede che  $\text{SC}(S) \leq \text{SC}(X)$ , cioè un soggetto può scrivere informazioni in oggetti che appartengono a livelli superiori al suo.

Analizzando le due regole ci si rende facilmente conto che un soggetto che sta a un determinato livello non può mai fornire informazioni a soggetti che si trovano a livelli più in basso del suo, garantendo così la sicurezza del sistema. Il modello Bell-La Padula permette anche di risolvere il problema di un eventuale cavallo di troia. Ipotizziamo infatti di avere la seguente situazione

	$O_1$	$O_2$	$O_3$
$S_1$	read, write	execute	write
$S_2$		execute, owner	read, write, owner

Tabella 11.1: Esempio 1. Applicazione delle regole Bell-La Padula per la risoluzione del cavallo di troia

Il soggetto  $S_1$  ha quindi il diritto di accesso esclusivo a  $O_1$ , sia in lettura che in scrittura. Ipotizziamo ora che un utente ostile  $S_2$  che ha avuto accesso al sistema, installa un programma (**cavallo di troia**) e un file privato, con diritti di lettura o scrittura che indicheremo rispettivamente come  $O_2$  e  $O_3$ . Avendo  $S_2$  il diritto di **owner** su  $O_2$  e  $O_3$  può trasferire i diritti di esecuzione e di scrittura, rispettivamente, a  $S_1$ . Se  $S_1$  esegue il programma contenuto in  $O_2$  (cavallo di troia), durante l'esecuzione potrebbe essere chiesto a  $S_1$  l'accesso a  $O_1$  e la scrittura dell'informazione in esso riservata su  $O_3$ . Il meccanismo di controllo degli accessi convalida entrambe le operazioni e quindi  $S_2$  è in grado di leggere il contenuto privato di  $O_1$ . Adottando una politica multilivello si possono introdurre due livelli: **pubblico** e **riservato**. Questi due livelli sono ordinati in modo che il livello pubblico sia inferiore di quello riservato. I livelli di sicurezza sono assegnati ai soggetti al momento del collegamento sulla base di criteri come il terminale da cui si accede al computer. In questo modo il soggetto  $S_1$  e l'oggetto  $O_1$  hanno il livello riservato, mentre  $S_2$  e i suoi oggetti quello pubblico. Quindi, quando  $S_1$  esegue il programma  $O_2$  potrà leggere  $O_1$ , ma essendo quest'ultimo

in un livello superiore, non potrà essere scritto all'interno di  $O_3$ , rimuovendo quindi il problema del cavallo di troia. Il modello Bell-La Padula è stato concepito per mantenere i segreti e non per garantire l'integrità dei dati. Esiste un altro modello detto BiBa che permette la garanzia dell'integrità dei dati, che di fatto inverte le regole del modello Bell-La Padula, leggendo solo i file di livello equo o superiore e scrivendo solo file di livello equo o inferiore. BiBa e Bell-La Padula sono fondamentalmente due specchi, mentre il primo garantisce solo la sicurezza, il secondo garantisce solo l'integrità.

# **Parte IV**

# **Laboratorio**

# 12

## Struttura e gestioni di un sistema unix

Tutti i file contenuti all'interno del nostro sistema sono resi disponibili e accessibili tramite un unico **filesystem** virtuale. In generale, il **filesystem**, è strutturato nel seguente modo Ogni documento,

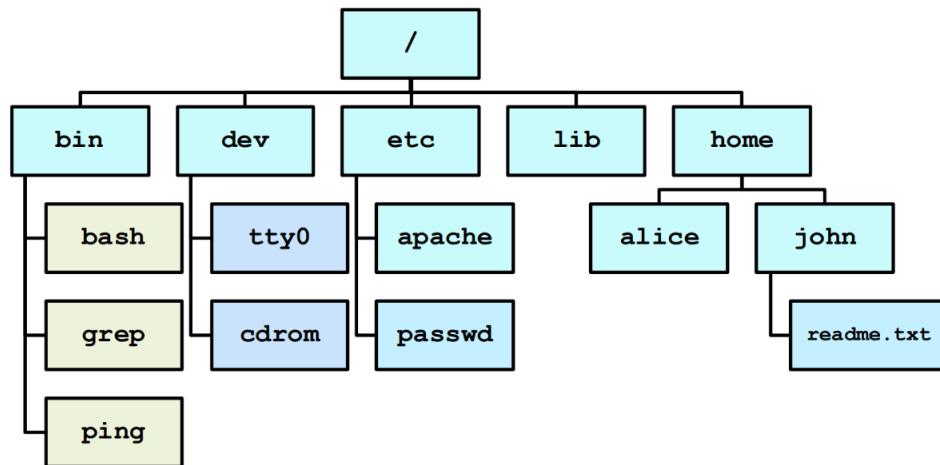


Figura 12.1: Struttura del filesystem unix

cartella o dispositivo I/O è accessibile dallo stesso filesystem e, di fatto, possiamo astrarre come un file. Ogni file è identificato dal proprio percorso; esistono due tipologie di percorsi

- **Percorso assoluto:** si esprime l'intero percorso partendo dalla radice.
- **Percorso relativo:** si esprime il percorso a partire dalla directory in cui mi trovo.

In unix troviamo anche alcuni caratteri speciali, il carattere **tilde** indica la **home directory** di un utente (quindi la cartella con lo stesso nome dell'utente messa dentro la cartella `all`), i simboli `.` e `..` indicano, rispettivamente, la directory corrente e la directory padre.

**Shell** Definiamo una shell come un interprete dei comandi che permette all'utente di richiedere informazioni e servizi al sistema operativo. Anche in questo caso dobbiamo fare una distinzione

- **Shell grafica** (o **GUI**).
- **Shell testuale** (o **CLI**).

La prima shell è sicuramente più facile da utilizzare, mentre la seconda permette di svolgere delle operazioni in un tempo molto minore rispetto a quello che ci vorrebbe con la **GUI**. La shell testuale legge un comando dall'utente, solitamente terminato dall'invio, e esegue il comando segnalando un errore se è impossibile portarlo a termine. Il formato generale di un comando unix è nella forma

**[comando] [opzione] [argomento dell'opzione]**

Prendiamo come esempio, anche se desueto, il comando **shutdown**. Questo comando ammette due possibili opzioni: **-r** per riavviare il sistema, **-h** per spegnerlo.

```
shutdown -h now
shutdown -r now
```

senza specificare l'argomento **now** il riavvio o lo spegnimento sarebbero stati programmati per un minuto dopo l'elaborazione del programma. Oltre a questi tre comandi troviamo anche i comandi per operare sulle directory: **ls**, **pwd**, **cd**. Il primo, detto anche *list*, serve per elencare il contenuto della directory specificata (senza argomenti specifica il contenuto della directory corrente); utilizzando l'opzione **-a** si visualizzano i file nascosti. Il secondo comando, *print working directory*, stampa il percorso assoluto della directory corrente. Infine, il terzo argomento, *change directory*, permette di andare a modificare la directory corrente; tale comando ammette i seguenti argomenti. Quando ci si trova nella situazione in cui non è chiaro come utilizzare un comando è comodo fare

Argomento	Significato
Path assoluto	Requisito fondamentale per il sistema
Path relativo	Imposta la sottodirectory con il nome
..	Imposta la directory padre come directory corrente
	Imposta la home come directory corrente

Tabella 12.1: Tabella delle opzioni per il comando **cd**

riferimento alla documentazione. A tal proposito, nei sistemi **unix**, esiste un comando che permette di fare ciò: il comando **man**. Questo comando permette, specificando il nome comando con la sintassi **man nome\_comando**, di avere una descrizione esaustiva di un qualsiasi comando, o anche di una qualsiasi funzione kernel, direttamente dalla documentazione. Esistono altri due comandi che permettono di ricavare delle informazioni utili sui comandi

- **whatis**. Serve per visualizzare la descrizione breve di una pagina del manuale.
- **apropos**. Serve per ricercare una parola in nomi e descrizioni.

Altri comandi per lavorare sulle directory sono

```
# crea una directory.
mkdir nome_dir

# rimuove una directory
rmdir nome_dir

# copia un file in un nuovo file o all'interno di una directory.
cp src dst

# nuove un file sorgente in un file destinazione o in una directory
mv src dst

# Modifica il file e lo crea se non c'è
touch nome_file

# concatena dei file
```

```

cat src1 src2 ... srcn

# rimuove dei file
rm file1 file2

# visualizza i file un po alla volta e interattivamente
less

#visualizzare la prima/ultima parte di uno o più file
head/tail

```

I processi hanno tre canali di input/output standard: `stdin`, `stdout`, `stderr`. Grazie a questi canali è possibile deviare l'output di un comando verso un file oppure acquisire l'input da un file. L'opzione `>` permette di inviare lo standard output o lo standard error a un file; ad esempio

```

# inviare lo standard output
ls -l > filelist.txt

# inviare lo standard error
ls -l 2> errorlist.txt

```

Se si utilizzassero due simboli `>` la scrittura del file sarebbe in **append**, quindi concatenata sul fondo. Per recuperare l'input da un file si utilizza il simbolo opposto `<`.

Il simbolo `|` specifica il comando di pipe, che permette di collegare l'output di un comando all'input del comando successivo al pipe; ad esempio:

```
ls -l mydir | less
```

Il comando `less` verrà quindi lanciato sul risultato dell'elaborazione del precedente comando. Gli ultimi due comandi sono `su` e `sudo nome_comando`:

- Il primo comando permette di accedere al terminare di un altro utente.
- Il secondo comando permette di lanciare il comando come un altro utente.

In entrambi i comandi, se non viene specificato l'utente, è implicito che venga utilizzato l'utente **root**. Suspendisse vitae elit. Aliquam arcu neque, ornare in, ullamcorper quis, commodo eu, libero. Fusce sagittis erat at erat tristique mollis. Maecenas sapien libero, molestie et, lobortis in, sodales eget, dui. Morbi ultrices rutrum lorem. Nam elementum ullamcorper leo. Morbi dui. Aliquam sagittis. Nunc placerat. Pellentesque tristique sodales est. Maecenas imperdiet lacinia velit. Cras non urna. Morbi eros pede, suscipit ac, varius vel, egestas non, eros. Praesent malesuada, diam id pretium elementum, eros sem dictum tortor, vel consectetur odio sem sed wisi.

## 1 Gestione degli utenti e dei permessi

Il meccanismo dei permessi gestisce l'accesso al filesystem da parte dei vari utenti del sistema. Per ogni file sono definiti un utente proprietario (**owner**) e un gruppo proprietario (**group owner**); di conseguenza, per ogni file distinguiamo tre tipologie di permessi:

- Permessi relativi al proprietario.
- Permessi relativi agli utenti del gruppo del proprietario.
- Permessi relativi agli altri utenti.

A ciascuna classe di utenti vengono applicati permessi specifici, i permessi possono essere di accesso in: **scrittura (r)**, **lettura (w)** e **esecuzione (x)**. Quando un utente prova ad utilizzare un file, vengono applicati i permessi relativi all'utente che decide di utilizzare il file. Per visualizzare i permessi relativi ai file presenti nella directory è possibile utilizzare il comando `ls -l`. Questi permessi possono essere rappresentati in formato ottale, utilizzando tre cifre in base otto, una cifra

per ogni quartetto di permessi. Analizzando singolarmente ogni gruppo di permessi notiamo che sono tre cifre binarie, quindi una singola cifra in base 8; per questo motivo è sufficiente utilizzare tre cifre per rappresentare i permessi di accesso al file. Infatti, ad esempio, se scriviamo la sequenza di permessi 111011001, possiamo scriverla in ottale come 731, il che vuol dire che l'owner ha tutti i permessi, il group owner ha i permessi di `write` ed `execute`, mentre gli utenti del sistema hanno solo permesso di `execute`. Altri esempi di permessi sono: Per modificare i permessi di accesso

Esempio	Descrizione
777	Sono garantiti tutti i permessi a ciascuno degli utenti.
750	L'owner ha tutti i permessi, il group owner ha il permesso in lettura ed esecuzione, gli altri utenti non hanno permessi.
000	Nessuno ha permessi.
510	L'owner può leggere ed eseguire il file, il group owner può eseguire il file e gli altri utenti non hanno nessun permesso

Tabella 12.2: Esempi di permessi di accesso con notazione ottale

relativi ad un file è possibile utilizzare il comando `chmod`. Questo comando permette di utilizzare la rappresentazione simbolica o quella ottale per la modifica dei permessi. Inoltre permette anche la modifica ricorsiva dei permessi di una directory o degli elementi in essa contenuti. L'esecuzione del comando è limitata solo all'utente proprietario del file. La sintassi è

```
-- notazione ottale
chmod xxx filename

-- notazione simbolica
chmod [who] [how] [which] filename
```

Nella notazione simbolica `who` rappresenta a chi viene applicata l'azione, quindi `u` per l'owner, `g` per il group owner e `o` per gli altri utenti. L'opzione `how` permette di specificare se i permessi devono essere aggiunti (+), rimossi (-) e/o assegnati (=). Infine, l'opzione `which` permette di specificare quale permesso. Ad esempio

```
-- Rimuovere il permesso di scrittura agli altri utenti
chmod o-w file.txt

-- Rimuovere il permesso di execute agli appartenenti al group owner
chmod g-x file
```

Altri permessi aggiuntivi possono essere quelli di **SUID** e **SGID**. Entrambi i bit permettono di specificare, rispettivamente, che durante l'esecuzione del file il processo acquisce i privilegi del proprietario del file o del group owner del file. Per implementarli è sufficiente utilizzare i bit di execute dell'owner del file (**SUID**) o del group owner del file (**SGID**).

### Cambiare il proprietario e il gruppo proprietario del file

Una delle possibilità offerte dai sistemi operativi unix è quella di cambiare dinamicamente l'`owner` o il `group owner` di un particolare `file` attraverso due primitive di sistema: `chown` e `chgrp`. Il primo comando ha una struttura del tipo

```
chown username filename
```

e permette di cambiare il proprietario del file `filename` a `username`. Affinché sia però possibile eseguire questo comando è necessario possedere i privilegi di root. Il secondo comando permette invece di cambiare il group owner del file e presenta una struttura del tutto analoga a quella di `chown`

```
chgrp groupname filename
```

Dove `groupname` è il nome del gruppo che si vuole rendere proprietario del `filename`.

## 2 Gesitone degli archivi

Una delle funzionalità offerte dai sistemi operativi unix è quella dell'archiviazione dei file all'interno di archivi compressi, cioè di archivi che, grazie ad un apposito **algoritmo di compressione**, riescano a memorizzare un grande quantitativo di file, riducendone la dimensione complessiva. Il comando che permette di archiviare o estrarre una raccolta di file e cartelle è

```
tar modalita[opzioni] [file]
```

La modalità specifica il **modo** con cui il comando deve operare (es. creare un archivio o estrarre da un archivio), le opzioni permettono di fornire ulteriori dettagli sul comportamento della funzione (es. specificare la tecnica di compressione). L'ultimo parametro è invece la lista di file/cartelle che si intende far archiviare/estrarre. Il formato del file dipende dalla compressione utilizzata

```
.tar      // non è stata usata compressione  
.tar.gz   // archivio compresso con gz  
.tar.bz2   // archivio compresso con bzip2
```

Le opzioni sono invece quelle che specificano in che modo il comando deve operare. Quindi, il primo passo è quello di specificare la modalità: ad esempio, specificando l'opzione **c** si crea un nuovo archivio, mentre specificando **r** si aggiunge file all'archivio. Altre opzioni permettono una gestione più articolata, ad esempio, l'opzione **u** aggiunge file all'archivio ma solo se differiscono dalla copia eventualmente già presente, oppure il comando **x** specifica di estrarre dei file dall'archivio. Successivamente al comando, troviamo le opzioni; in particolare: l'opzione per specificare invece il tipo di compressione da utilizzare è **z** se la compressione avviene con **gzip**, mentre **j** se la compressione avviene con **bzip2** e per specificare anche il nome dell'archivio su cui operare è necessario usare l'opzione **f**. Possiamo prendere come esempio alcune combinazioni possibili

```
-- crea un archivio contenente i file in percorso  
tar cvf archivio.tar percorso  
  
-- crea un archivio compresso con gz con il contenuto di percorso  
tar czf archivio.tar.gz percorso  
  
-- mostra il contenuto di un archivio.tar  
tar tf archivio.tar
```

Tuttavia, se si vuole comprimere un archivio creato precedentemente con una delle due modalità appena viste, è necessario utilizzare i comandi **gzip file1, file2, ...** e **gunzip file1, file2, ...** nel caso in cui si scelta di adottare **gzip**, altrimenti **bzip2** e **bunzip2**.

# 13

## Gestione dei processi con la libreria sys.h

Unix è una famiglia di sistemi operativi multiprogrammati basati sui processi. Un processo in unix mantiene spazi di indirizzamento separati per **dati** e **codice**; in particolare abbiamo uno **spazio di indirizzamento dei dati privato** e uno **spazio di indirizzamento del codice condivisibile**. Il descrittore di un processo è quindi una struttura dati suddivisa in due parti

- **Process Structure.** Questa struttura contiene al suo interno informazioni indispensabili, che devono essere sempre in memoria.
- **User Structure.** Informazioni utili solo quando il processo è residente in memoria.

## 1 System call per la gestione dei processi

Ogni processo è in grado di creare dinamicamente dei processi figli; ogni processo creato ha uno spazio dati separato, ma condivide con il processo padre il codice. Ogni processo figlio può a sua volta generare altri processi figli, che condivideranno il codice del padre. La **syscall** principale in unix per la creazione di processi figli è la chiamata **fork()**. La struttura della **fork** è così fatta:

```
pid_t fork(void)
```

Questa funzione non richiede alcun parametro e restituisce un risultato intero diverso tra padre e figlio. Infatti, nel caso in cui il processo padre legga il risultato della funzione, otterrà il PID del processo figlio appena creato, mentre nel caso in cui sia il processo figlio a leggere il risultato otterrà come risultato 0. Il processo figlio appena creato condivide il codice con il padre, ereditando una **copia** delle aree di dati **globali**, **stack**, **heap** e **User Structure**. Condividendo anche il **Program Counter** padre e figlio, dopo la fork, ripartiranno dalla stessa istruzione; per differenziare il comportamento dei due processi è quindi necessario utilizzare il valore di ritorno del **fork**

```
pid_t ret = fork();
if (ret == 0){
    // codice del figlio
} else {
    // codice del padre
}
```

Altre due funzioni di utilità sono la funzione **getpid()** e la funzione **getppid()**. La prima restituisce il PID del processo, mentre la seconda restituisce il PID del processo padre. Una funzione che invece permette di terminare volontariamente un processo è la **syscall exit()**. La struttura della funzione è:

```
void exit(int status);
```

I processi che terminano volontariamente possono utilizzare questa syscall, specificando come parametro lo stato di terminazione del processo. La chiamata non ha alcun tipo di ritorno e permette di comunicare al padre lo stato di terminazione del processo. Il problema diventa però uno: come è possibile per un processo conoscere lo stato di terminazione di un suo processo figlio senza aver la possibilità di comunicare direttamente con esso. Per farlo è necessario che utilizzi una particolare funzione, chiamata `wait()`. La struttura della funzione è:

```
pid_t wait(int* status);
```

Il padre può ottenere, grazie a questa funzione, lo stato di terminazione del figlio. La funzione ha come valore di ritorno il PID del figlio che è terminato, mentre `status` è l'indirizzo della variabile dove verrà salvato lo stato di terminazione del figlio. La `wait()` è una funzione che causa la sospensione del padre se tutti i figli sono ancora in esecuzione, l'effetto sul padre è quello del ritorno immediato con informazioni di terminazione se almeno un figlio è terminato, mentre ritorna un valore negativo se non ci sono processi figli.

La variabile `status` contiene informazioni su come il figlio è terminato, oltre allo stato di terminazione eventualmente fornito dal figlio stesso attraverso la `exit()`. Se il byte meno significativo di `status` è pari a zero, allora la terminazione è stata volontaria. Per gestire lo status di terminazione di un processo è possibile utilizzare le macro definite nella libreria `sys/wait.h`

```
WIFEXITED(status);
WEXITSTATUS(status);
```

La prima funzione ritorna vero se il processo figlio è terminato volontariamente, mentre la seconda funzione ritorna lo stato di terminazione.

### Esecuzione di un comando tramite processo figlio

Un processo può sostituire il programma che sta eseguendo utilizzando una funzione della famiglia `exec()`. L'idea è che quindi si possa far eseguire un comando, come potrebbe essere `ls`, o `cd`, ad un processo figlio utilizzando una semplice primitiva, con la seguente struttura:

```
int execl(char* path, char* arg0, ..., char* argN, NULL);
```

Il primo campo corrisponde al percorso assoluto del comando, `arg0` rappresenta il nome del programma da eseguire, mentre i parametri `arg1, ..., argN` sono gli eventuali argomenti del programma.

## 2 Interazione e sincronizzazione tra processi

I modelli in unix aderiscono al modello ad ambiente locale; ogni processo ha un proprio spazio di indirizzamento privato e non c'è alcuna condivisione di variabili. L'unica forma di interazione tra processi è la cooperazione attraverso la **sincronizzazione** e la **comunicazione** (attraverso lo scambio di messaggi). Queste interazioni si basano su delle astrazioni realizzate dal **kernel**. L'astrazione in questione è quella dei **segnali**: i segnali sono il meccanismo messo a disposizione dai sistemi Unix/Linux per la sincronizzazione di processi:

- Permettono la notifica di eventi asincroni da parte di un processo a uno o più processi.
- Possono essere utilizzati dal sistema operativo per notificare il verificarsi di qualche eccezione ai processi utente.

I segnali sono delle **interruzioni software**. Ad ogni segnale è associato un particolare **handler** di default che specifica l'azione da compiere nel caso in cui a quel processo arrivi un particolare segnale, tuttavia è anche possibile per il programmatore, ridefinire un proprio **handler** personalizzato oppure ignorare del tutto il segnale. Nei primi due casi, cioè quello in cui l'arrivo del segnale viene gestito attraverso **handler**, i processi hanno un comportamento asincrono rispetto ai segnali: l'esecuzione viene interrotta per eseguire l'**handler** e successivamente, se il processo non è terminato, si riprende dall'istruzione successiva all'ultima eseguita prima dell'interruzione. La lista

dei segnali è inclusa all'interno del file `signal.h`, dove ciascun segnale è identificato da un intero e da un nome simbolico. Alcuni segnali di uso comune sono: `SIGHUP` quando il terminale viene chiuso, `SIGINT` che interrompe il processo da terminale quando si utilizza la combinazione `CTRL+C`, `SIGKILL` che è un segnale di interruzione immediata e anche `SIGUSR1/2` che sono due segnali definiti dall'utente, ma che di default terminano il processo. Le principali funzioni per interagire con i segnali in unix sono

Funzione	Descrizione
<code>signal</code>	Permette di definire la funzione che dovrà gestire il segnale.
<code>kill</code>	Invio di segnali
<code>pause</code>	Il processo va in stato di <code>sleep</code> in attesa di un segnale
<code>alarm</code>	Invio implicito di segnali
<code>sleep</code>	Uguale ad <code>alarm</code> .

Tabella 13.1: Funzioni per l'interazione tra processi mediante segnali

### Funzione `signal`

La funzione `signal` è una funzione che permette di associare ad un particolare segnale la funzione `handler` corrispondente. La struttura della funzione è

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int sig, sighandler_t handler);
```

Il primo parametro è il codice del segnale, mentre il secondo parametro è l'`handler` che si occupa di gestirlo. Inoltre, la funzione `handler` deve prevedere un parametro intero, che al momento della ricezione del segnale conterrà il codice del segnale. Un `handler` può valere anche `SIG_IGN` (ignora il segnale), oppure `SIG_DFL` (ripristina l'azione di default). Restituisce un puntatore al precedente `handler` del segnale o `SIG_ERR` in caso di errore. Il figlio eredita dal padre le informazioni relative alla gestione dei segnali, inoltre eventuali signal eseguite dal figlio non hanno effetto sul padre.

### Funzione `kill`

La funzione `kill` permette di inviare il segnale `sig` al processo `pid`. Se `pid > 0` il segnale viene inviato al processo `pid`, `pid == 0` specifica invece che il segnale dovrà essere inviato a tutti i processi nello stesso `process-group` del chiamante, se `pid == -1` il segnale viene inviato globalmente a tutti i processi al cui il chiamante è in grado di inviare segnali, infine, se `pid < -1` il segnale viene inviato ai processi il cui `process-group` è `-pid`. La struttura della funzione

```
int kill(pid_t pid, int sig);
```

Ritorna zero in caso di successo.

### Funzione `pause`

Funzione che permette al processo chiamante di sospendersi nello stato di *sleeping* fino alla ricezione di un segnale. In particolare, il processo chiamante va nello stato di `sleep` fino a che è passato un certo intervallo di tempo, oppure fintanto che non arriva un segnale che non viene ignorato. La struttura della funzione è:

```
unsigned int sleep (unsigned int seconds)
```

Il parametro `seconds` corrisponde all'intervallo di tempo che occorre attendere prima che il processo si risvegli. Quando è passato il tempo indicato, il processo viene risvegliato. Ritorna zero se è passato il tempo previsto, altrimenti il tempo rimasto dopo l'arrivo di un segnale.

### **Funzione alarm**

La funzione provoca la ricezione di un segnale **SIGALARM** dopo *seconds* secondi. Il comportamento di default di **SIGALARM** termina il processo, mentre un eventuale allarme invocato precedentemente viene cancellato. Se **seconds** è zero, viene eliminato un eventuale allarme invocato precedentemente. La struttura della funzione è:

```
unsigned int alarm (unsigned int seconds);
```

Ritorna zero se non c'era un allarme programmato, altrimenti ritorna il numero di secondi mancanti all'ultimo allarme programmato.

# 14

## Gestione dei processi da terminale

Oltre alla gestione mediante codice, è possibile anche gestire i processi attraverso il terminale unix. I sistemi unix prevedono infatti un **init** system, cioè un **processo speciale**, mandato in esecuzione dal kernel durante il boot di sistema. Questo processo è, quindi, il primo processo ad andare in esecuzione, con PID = 1, ed è, di fatto, il processo a partire dal quale tutti gli altri processi discendono (è possibile visualizzare l'albero dei processi eseguendo `ps tree` sul terminale). Nei sistemi basati su **Debian** o **Ubuntu**, i processi vengono gestiti da un componente chiamato **systemd**. Un processo in unix è identificato da cinque attributi I primi 3 attributi sono immediati

Nome attributo	Descrizione
PID	ID univoco del processo.
PPID	ID univoco del padre del processo.
PGID	ID del <i>process group</i> a cui appartiene il processo.
RUID, RGID	”Real” User/Group ID.
EUID, EGID	”Effective” User/Group ID

Tabella 14.1: Identificatori di un processo in unix

da un punto di vista funzionale. Tuttavia occorre prestare molta attenzione per quanto riguarda gli ultimi quattro attributi della tabella, infatti essi hanno un significato molto particolare e non sempre immediato. Il **RUID** è l'identificativo dell'**utente che ha mandato in esecuzione il processo**, mentre il **GUID** è l'identificativo del gruppo di appartenenza del **RUID**. D'altro canto l'**EUID** non è altro che l'identificativo dell'utente i cui privilegi vengono usati dal processo durante l'esecuzione e, quindi, il **EGID** sarà il gruppo di appartenenza del **EUID**. Facciamo un piccolo esempio a riguardo:

Se un utente *Mario* intende cambiare la sua password, per farlo deve riuscire a fare in modo che il sistema scriva la nuova password all'interno del file `/etc/shadow`. Tuttavia, per motivi di sicurezza, questo file è accessibile solo all'utente **root**, di conseguenza è necessario che Mario cambi la sua password utilizzando i privilegi di sicurezza di questo utente. Per farlo utilizza il comando `/usr/bin/passwd`, che prende temporaneamente in prestito i privilegi di root all'utente permettendogli di cambiare la password.

Osservando i parametri del comando `passwd` si potrà osservare quindi come il **RUID** corrisponde all'identificativo dell'utente **Mario**, mentre il **EUID** corrisponde a quello dell'utente **root**. Ovviamente non è sempre possibile avere **RUID** e **EUID** che differiscono, questa possibilità è infatti vincolata ai soli comandi che hanno il bit **SUID** o **SGID** attivo. Inoltre, un processo non **root** può mandare segnali solo ad un altro processo solo se il suo **EUID** e **RUID** coincide con il **RUID**.

# 1 Organizzazione dei processi in Unix

I processi, all'interno di un sistema UNIX, sono organizzati in gruppi. Un qualsiasi processo che viene mandato in esecuzione deve necessariamente fare parte di un gruppo (**process group**). Se questo processo, durante la sua esecuzione, genera dei figli, allora i figli avranno medesimo GUID del processo padre che li ha generati; il gruppo viene preservato anche dalla syscall `exec`. Grazie alla presenza dei gruppi è possibile mandare segnali ad una gerarchia di processi, implementando anche il meccanismo alla base del `job-control` offerto dalla shell.

Lo scheduler assegna la CPU dei processi tenendo conto di un livello di priorità assegnato a ciascun processo; questa priorità dipende generalmente dalla classe di scheduling del processo. La priorità dei processi normale può essere in parte controllata mediante il concetto di **niceness** e la relativa syscall `nice`. La **niceness** è un concetto molto particolare, che può essere tradotto, in estrema sintesi, come la permissività di un processo di attendere che altri processi siano eseguiti prima di lui. Ad ogni processo è associato un valore di **niceness** in un intervallo compreso tra  $[-20, 19]$ , dove un valore alto di niceness indica che il processo è favorevole a far eseguire altri processi prima di lui (quindi implica che abbia anche una minore priorità di esecuzione). In altri termini, la **niceness** è inversamente proporzionale alla priorità

$$\text{Niceness} \propto \frac{1}{\text{Priority}}$$

Da un punto di vista di sistema, solo l'utente `root` è in grado di diminuire la niceness, mentre un generico utente potrà solo aumentarla; questo fatto si traduce nella conseguenza che un utente è solo in grado di diminuire la priorità di un processo, ma mai di aumentarla. Le syscall che permettono la gestione della niceness in unix sono

```
-- viene mandato in esecuzione il processo nome_processo_più_risorse
-- con niceness pari a valore_nice
nice -n valore_nice nome_processo_più_risorse &

-- il valore di niceness del processo PID viene messo a valore_nice
renice valore_nice PID
```

# 2 Gestione dei processi da terminale

Con il termine `job-control` si intende la possibilità, di sospendere e riattivare gruppi di processi, offerta dalla shell mediante l'utilizzo di opportuni comandi. La shell associa un `job` ID distinto ad ogni comando eseguito o a ogni pipe di comandi che viene lanciata da terminale. I vari `job` sono salvati all'interno di una tabella specifica, visualizzabile attraverso il comando `jobs`. Possiamo distinguere due tipologie di `job`:

- **Foreground.** Un job in esecuzione in foreground ha il controllo di **standard input**, **standard output** e **standard error**. Di fatto un processo prende il controllo del terminale e lo restituisce alla shell quando termina.
- **Background.** La shell permette anche di eseguire job in **background**. Un processo in background non ha accesso agli standard input e lo si mette in esecuzione ponendo una `&` al termine del comando.

Un processo in foreground può essere fermato inviando il segnale `SIGTSTP`. Una volta lanciato questo segnale è possibile anche intervenire sui job che sono stati fermati, facendoli ripartire in **background** o in **foreground**. Questi comandi sono:

- `fg JOB_ID`. Fa ripartire il processo in foreground.
- `bg JOB_ID`. Fa ripartire il processo in background.

Un altro comando molto utile dal punto di vista del terminale è quello che permette di terminare i job; in particolare, specificando il comando `kill`, con il parametro `&JOB_ID` è possibile inviare il

segna **SIGTERM** al job specificato. Inoltre, nel momento in cui un terminale viene chiuso, tutti i job in esecuzione ricevono il segnale di **SIGHUP** e, di default, vengono terminati. Tuttavia è possibile fare in modo che questo segnale non termini i job alla chiusura del terminale; per farlo si possono utilizzare due comandi:

```
nohup nome_comando  
disown %JOB_ID
```

Grazie al primo comando si rende il job eseguito immune al segnale di **SIGHUP**, tuttavia il job non avrà accesso allo standard input e lo standard output verrà rediretto su un file chiamato **nohup.out**. Il secondo comando permette, invece, di rendere immune un job già in esecuzione, l'effetto di questa chiamata è che il job venga rimosso dalla tabella dei job, quindi la shell non invierà più il segnale **SIGHUP** quando viene chiusa. In questo caso è opportuno fare in modo che il job non legga dallo **stdin** e che l'eventuale output venga rediretto su file per evitare errori durante l'esecuzione.

# 15

## Gestione dei Thread con la libreria pthread.h

Un Thread (o **processo leggero**), ampiamente già discusso durante questo corso, non è altro che un **flusso di esecuzione indipendente** all'interno di un **processo**. Un singolo **processo pesante** è quindi in grado di dare vita a più thread che coesistono all'interno dello spazio virtuale del processo stesso, condividendone alcune risorse, ma avendo comunque a disposizione una propria area di memoria privata.

Un **thread** è talvolta anche chiamato come **processo leggero** in quanto:

- La creazione e la distruzione di un thread sono più leggeri rispetto alla creazione e distruzione di un processo.
- Il cambio di contesto fra thread dello stesso processo è meno oneroso rispetto al cambio di contesto fra processi.

L'utilizzo dei thread, come è evidente, permette di semplificare le interazioni attraverso l'utilizzo di variabili comuni, riducendo anche il costo computazionale dato dalla creazione e dalla distruzione di un processo pesante. Tuttavia, la presenza dei thread richiede però che venga opportunamente gestita la possibile concorrenza sulle risorse comune che si può creare; in particolare, è necessario che il codice utilizzato sia *thread safe*:

- Il codice è scritto in modo da evitare interazioni non volute (interferenza) tra thread.
- Le risorse condivise devono essere acceduta in mutua esclusione.

I thread possono essere definiti sia a livello di kernel, che a livello utente. Nel caso di Linux i thread sono supportati nativamente a livello kernel, ottenendo quindi i seguenti vantaggi:

- Il thread è l'unità di scheduling e può essere eseguito in parallelo con altri thread.
- Ogni processo unix può essere visto come un thread che non condivide risorse con altri thread.

Lo standard POSIX definisce la libreria **pthread**s per la programmazione di applicazioni multithreaded portabili. Per utilizzare questa libreria è necessario rispettare alcune notazioni nella compilazione, in particolare:

```
gcc <opzioni> file.c -lpthread
```

### 1 Struttura di un thread

Un thread è identificato da un **ID**, di tipo **pthread\_t**, ottenibile utilizzando la funzione **pthread\_self(void)**. Il tipo **pthread\_t** è detto **tipo opaco**, poiché può essere utilizzato solo mediante apposite funzioni. Il tipo opaco nasconde il modo in cui è effettivamente realizzato e ci permette di utilizzarlo e/o modificarlo solo mediante funzioni specifiche. L'esecuzione di un programma linux determina anche le'secuzione automatica di un thread che esegue il codice del main; a sua volta, il thread che esegue il main, può generare una gerarchia di thread utilizzando la funzione

```
int pthread_create(pthread_t* thread, const pthread_attr_t* attr,
                  void* (*start_routine)(void*), void* arg);
```

Il primo argomento, cioè l'id del thread, è un puntatore a un identificatore di thread dove verrà scritto l'id del thread creato. Il secondo argomento, cioè `const pthread_attr_t*`, è un puntatore a una struttura che contiene gli attributi del thread che è possibile impostare (specificando `null` se si utilizzano dei valori di default). Il terzo argomento, cioè `start_routine`, è un puntatore alla funzione che contiene il codice del nuovo thread, cioè identifica il comportamento del nuovo thread. L'ultimo argomento, cioè `void* arg`, è un puntatore a una cella di memoria che viene passato come argomento alla `start_routine`. Nel caso in cui il thread sia creato con successo, la funzione ritornerà zero.

La funzione duale a quella di creazione del thread è quella di terminazione, la quale può essere invocata dal thread stesso per terminare la sua esecuzione:

```
void pthread_exit(void* retval);
```

Questa funzione termina l'esecuzione del thread e libera le risorse allocate al thread. Se un thread padre termina prima dei thread figli

- Se non chiama la `pthread_exit` anche i figli vengono terminati.
- Se chiama la `pthread_exit` i figli continuano la loro esecuzione.

L'argomento `retval` corrisponde all'area di memoria dove, per ovvi motivi, il thread salva il proprio valore di ritorno, consultabile da altri thread usando la funzione

```
int pthread_join(pthread_t thread, void** retval);
```

Questa funzione permette a un thread di bloccarsi in attesa della terminazione di un thread specifico. Accetta come parametri un identificatore `pthread_t` del thread di cui si attende la terminazione e un puntatore a puntatore dove verrà salvato l'indirizzo restituito dal thread con la `pthread_exit`. Analogamente alla `pthread_create`, anche la `pthread_join` restituisce zero in caso di successo.

## 2 Mutua esclusione e Sincronizzazione

Per risolvere i problemi di mutua esclusione, la libreria `pthread` mette a disposizione l'astrazione della variabile binaria `mutex`, analoga all'astrazione di un **semaforo binario**. In particolare, la libreria `pthread` implementa il tipo `pthread_mutex_t` che permette di definire facilmente un semaforo binario

```
pthread_mutex_t M;
```

Per inizializzare la variabile `mutex`, si utilizza la funzione `pthread_mutex_init`, la quale accetta due argomenti: un riferimento alla variabile che vogliamo inizializzare, il secondo è un riferimento a una struttura `pthread_mutexattr_t` che contiene gli attributi di inizializzazione:

```
int pthread_mutex_init(pthread_mutex_t* M, const pthread_mutexattr_t* mattr)
```

Anche in questo caso, il parametro `mattr`, può essere posto a `NULL`; così facendo è possibile utilizzare le configurazioni degli attributi di default. Le due funzioni fondamentali per la gestione dei semafori, cioè `wait` e `signal`, sono implementate nel seguente modo

- `int pthread_mutex_lock`.
- `int pthread_mutex_unlock`.

Entrambe le funzioni ritornano zero in caso di successo. Se più thread provano ad accedere alla risorsa, solo uno di essi potrà accedere, mentre gli altri verranno bloccati. Il `mutex` è, tuttavia, lo strumento che permette solo la **sincronizzazione indiretta** tra più thread, ma non garantisce alcuna forma di **sincronizzazione diretta**. Per la sincronizzazione **diretta** dei `thread` la libreria definisce le variabili condizione:

- Un thread può sospendersi in attesa del verificarsi di una determinata condizione.
- Permette di realizzare politiche avanzate di accesso alle risorse condivise e di sincronizzare i thread.

Una variabile condizione è - di fatto - una coda nella quale i thread possono sospendersi volontariamente in attesa di una condizione. Il tipo che la definisce è

```
pthread_cond_t C
```

Per inizializzare una variabile **conditions** è possibile utilizzare una **funzione di inizializzazione** che ammette come parametri: un **puntatore** alla variabile condizione da inizializzare e un **puntatore** a una struttura che contiene gli attributi specificati per la condizione, inizializzata a **default** impostandola a **null**.

```
int pthread_cond_t(pthread_cond_t* C, pthread_cond_attr_t* attr);
```

Un generico thread può quindi compiere due operazioni su una variabile condizione: sospendersi sulla variabile e risvegliare uno o più thread sospesi sulla variabile. Nel primo caso, il thread, dopo aver verificato una determinata condizione logica, si sospende sulla variabile condition in attesa di essere risvegliato ad un altro thread. Queste due possibili opzioni sono codificate attraverso due funzioni appropriate

**Funzione wait()** La sospensione viene utilizzata al verificarsi di una particolare condizione logica. Ad esempio, un thread produttore ha verificato che sul buffer condiviso è pieno, quindi si blocca sulla condition "pieno" in attesa che un thread consumatore lo risvegli. Quindi, la wait su una variabile condition è sempre **bloccante** e segue uno schema generico del tipo

```
...
pthread_cond_t C;
...
while(condizione) wait(C);
```

La scelta di utilizzare un while e non un IF deriva dal fatto che quando un thread viene risvegliato non va subito in esecuzione: altri thread potrebbero inserirsi e alterare la condizione, ed è quindi necessario ricontrillare la condizione dopo essere stati svegliati. La condizione logica che viene utilizzata all'interno del while dovrà essere ovviamente basata su una risorsa condivisa, che dovrà quindi essere acceduta in mutua esclusione e, per tale motivo, la primitiva di **wait** della libreria **pthread** permette di associare una variabile mutex a una variabile condition

```
int pthread_cond_wait(pthread_cond_t* C, pthread_mutex_t* M);
```

Il primo parametro corrisponde alla variabile condizione su cui si vuole sospendersi, mentre il secondo argomento corrisponde al semaforo associato alla condizione: viene liberato automaticamente quando il thread si sospende e viene eseguito un nuovo lock quando il processo viene risvegliato. Quindi, la chiamata alla funzione avrà due effetti:

- Sospendere il processo sulla coda associata a C.
- Il semaforo M viene liberato.

**Primitiva signal()** Il risveglio di un thread sospeso su una variabile condition C avviene mediante la primitiva **signal()**:

```
int pthread_cond_signal(pthread_cond_t* C);
```

Se esistono thread in coda sulla signal, un thread bloccato **scelto casualmente** viene risvegliato, mentre se non ci sono thread sospesi non ha alcun effetto. La politica della signal della libreria **pthread** è di tipo **signal & continue**: il thread che esegue la signal continua la sua esecuzione e mantiene il controllo del mutex fino al suo esplicito, mentre il thread che viene risvegliato dovrà verificare nuovamente la condizione per il **wait()**.

La funzione vista fino ad ora però risveglia solo un thread scelto casualmente, se invece l'obiettivo fosse quello di risvegliare tutti i processi sarebbe possibile utilizzare la funzione

```
int pthread_cond_broadcast(pthread_cond_t* C);
```

Per maggiore stabilità, la **signal/broadcast** va invocata **dentro la sezione critica**, cioè prima della funzione `unlock`. Facendo in questo modo siamo sicuri che nel momento in cui la funzione `signal()` o `broadcast()` viene invocata la condizione è rispettata.

**Esempio 1. Produttore-Consumatore** Dei thread accedono a una risorsa condivisa, ad esempio un buffer di interi gestito in modo circolare, dove abbiamo due tipologie di thread: i thread consumatori che inseriscono valori nel buffer e i thread produttori che prelevano i valori del buffer. Inoltre, la gestione del buffer ha due vincoli di gestione, il primo è che non è possibile prelevare dal buffer vuoto, la seconda, duale alla prima, è che non è possibile inserire nel buffer pieno. Possiamo quindi impostare il modello del buffer

```
typedef struct {
    int buffer[BUFFER_SIZE];
    int readlnd, writelnd;
    int cont;

    pthread_mutex_t M;

    pthread_cond_t FULL;
    pthread_cond_t EMPTY;
} risorsa;
```

Una volta definita la struttura della risorsa è necessario che venga opportunamente inizializzata all'interno del main

```
risorsa r;
...
int main(){
    pthread_mutex_init(&r.M, NULL);
    pthread_cond_init(&r.FULL, NULL);
    pthread_cond_init(&r.EMPTY, NULL);

    r.readlnd = r.writelnd = r.cont = 0;
}
```

Il produttore deve quindi assicurarsi che il buffer non sia pieno, nel caso attendere, inserire un buffer all'interno del dato e, solo successivamente, risvegliare un eventuale consumatore in attesa.

```
int val;
...
pthread_mutex_lock(&r.M);
while(cont == BUFFER_SIZE) pthread_cond_wait(&r.FULL, &r.M);

r.buffer[r.writelnd] = val;
r.cont++;
r.writelnd = (r.writelnd +1) % BUFFER_SIZE;

pthread_cond_signal(&r.EMPTY);
pthread_mutex_unlock(&r.M);
```

Il consumatore deve invece assicurarsi che il buffer non sia vuoto, nel caso bloccarsi, e dopo aver prelevato il dato risvegliare un consumatore eventualmente sospeso

```

...
pthread_mutex_lock(&r.M);
while(r.cont == 0) pthread_cond_wait(&r.EMPTY, &r.M);

val = r.buffer[r.readlnd];
r.cont--;
r.readlnd = (r.readlnd + 1) % BUFFER_SIZE

pthread_cond_signal(&r.FULL);
pthread_mutex_unlock(%&r.M);

```

Immaginiamo ora il caso in cui tanti produttori e tanti consumatori privino ad accedere ad una risorsa condivisa, che è sempre rappresentata dal buffer, contemporaneamente. L'utilizzo delle variabili conditions permette di realizzare politiche di accesso a una risorsa più avanzate. Ipotizziamo quindi di avere NTHREADS che utilizzano periodicamente risorsa, che può essere utilizzata periodicamente solo da MAX\_T di thread. In questo caso si può usare una variabile conditions PIENO associato alla condizione

#### Processi che usano la risorsa == MAX\_T

Un thread dovrà quindi, prima di usare e di lasciare la risorsa, una fase, rispettivamente, di ingresso e di uscita. Impostiamo quindi altre variabili globali

```

#define MAX_T 10
...
int n_users = 0;
pthread_cond_t FULL;
pthread_mutex_t M;

```

Nella fase di ingresso un thread dovrà verificare la condizione che il numero di thread massimi che operano sulla risorsa non abbia già raggiunto il massimo e, nel caso, attendere. Successivamente potrà proseguire nelle operazioni già viste in precedenza. Infine, prima di rilasciare la risorsa, dovrà eseguire una fase di uscita nella quale dovrà decrementare il numero di utenti che utilizzano la risorsa, eseguire una signal sulla coda dei processi in attesa su FULL e, infine, eseguire la unlock su M.

```

pthread_mutex_lock(&M);
while(n_users == MAX_T) pthread_cond_wait(&FULL, &M);
n_users++;
pthread_mutex_unlock(&M);
...
// uso della risorsa
...
pthread_mutex_lock(&M);
n_users--;
pthread_mutex_signal(&FULL);
pthread_mutex_unlock(&M);

```

# 16

## Il filesystem linux

All'interno del kernel linux l'organizzazione logica del filesystem è strutturata a partire dalla radice / del sistema, dal quale possiamo trovare alcune altre cartelle come /sbin, che contiene i programmi di sistema, /etc, che contiene i file di configurazione o /dev, che rende accessibili le periferiche; graficamente, questa organizzazione, è così strutturata:

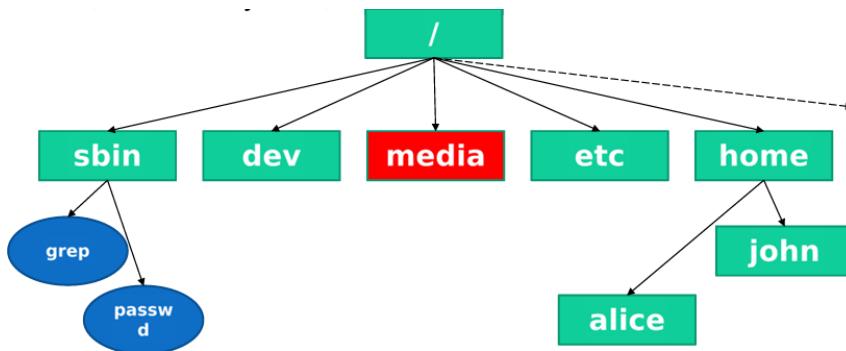


Figura 16.1: Organizzazione logica del filesystem di linux

Il comando per rendere accessibile un filesystem in una determinata posizione del **filesystem virtuale** è il comando **mount**. All'interno di linux ogni singola risorsa del sistema operativa è rappresentata come file, di cui possiamo distinguere tre tipologie:

- **File ordinario.** Insieme di informazioni allocate in memoria di massa.
- **File speciale.** Dispositivo fisico.
- **Directory.** Insieme di file.

Il contenuto di una **directory** è costituito da una serie di record che descrivono il contenuto della directory: <nome\_file1, i-number1>. Una parte del descrittore del disco è dedicata alla **i-list**, la lista di tutti i descrittori di file (**i-node**), dove ogni **i-node** è identificato da un **i-number**. In particolare, l'**i-node** descrive le caratteristiche del file

- Tipo.
- Informazioni sui permessi,
- Dimensione.
- Link. Numero di nomi che riferiscono al file
- Vettori di indirizzamento del file.

Alla creazione di un nuovo file viene creato un nuovo **i-node** identificato da un **i-number** e al contenuto della directory che contiene il file viene aggiunto un nuovo record contenente il nome del file e l'**i-number**. Dopo la creazione, il file, avrà un solo nome, detto **hard link** associato al descrittore del file. Tuttavia il filesystem permette di definire più **hard link** associati ad un singolo **i-node**, che rappresentano dei nomi alternativi per lo stesso file. Oltre agli **hard link**, è possibile definire dei **soft link**, cioè dei nomi simbolici alternativi per fare riferimento ad un hard link. Per spiegare le differenze è possibile fare riferimento alla seguente immagine

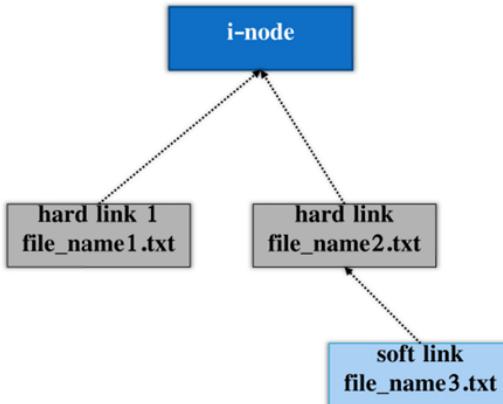


Figura 16.2: Organizzazione dei soft-link e hard-link

Gli hard link puntano allo stesso **i-node**; inoltre, se sposto o elimino un hard link, non c'è alcun effetto sugli altri hard link. Ciò vuol dire che affinché sia possibile eliminare un file è necessario che il suo numero di hard-link sia zero. Il **soft link** è invece un riferimento ad un hard link, quindi se elimino o sposto l'**hard link**, il **soft link** non è più in grado di accedere al file. Per creare hard e soft link è possibile utilizzare il comando **ln**

```
-- creazione di un hard link
ln target link_name

-- creazione di un soft link
ln -s target link_name
```

L'accesso ai file sfrutta meccanismo sequenziale; ad ogni file aperto è associato un I/O pointer riferimento per la lettura e la scrittura sequenziale sul file e le operazioni di lettura/scrittura provocano l'avanzamento del riferimento. La strutture dati per l'accesso ai file sono gestite dal kernel:

- **Tabella dei file aperti di processo.** Definita nella user structure del processo, ogni elemento al suo interno è un riferimento all'elemento corrispondente nella tabella di file aperti di sistema.
- **Tabella dei file aperti di sistema.** Contiene un elemento per ciascun file aperto dal sistema: se due processi aprono lo stesso file allora ci saranno due entry separate. Ogni elemento della tabella contiene un riferimento all'**i-node** del file.

I/O pointer e **i-node** permettono di trovare l'indirizzo fisico a cui effettuare la prossima lettura/scrittura sequenziale. Esistono anche dei descrittori di default all'interno di linux, cioè: STDIN, STDOUT e STDERR. Un eventuale processo figlio eredita dal padre una copia della User Structure del padre, quindi anche una copia del file descriptor. In questo caso i due processi hanno descrittori che puntano allo stesso elemento della **Tabella dei file di sistema**, e quindi condividono l'I/O pointer nell'accesso sequenziale ai file.

## 1 Primitive per l'accesso ai file

La funzione per aprire un file descriptor è la funzione `open`, questa funzione prende come argomento una stringa relativa al path del file, mentre il secondo è un interno che specifica la modalità di accesso al file

```
int open(const char* path, int flags)
```

Le modalità di accesso sono varie e sono definite all'interno di `fctl.h`. La funzione ritorna un **file descriptor**, così da poter identificare il file **aperto**. Dopo l'apertura, l'**I/O** pointer viene posizionato all'inizio del file se non è utilizzata la modalità `O_APPEND`. La funzione duale alla `open`, cioè quella che permette di chiudere il **file descriptor**, è la funzione `close`

```
int close(int fd)
```

Il primo argomento è il file descriptor che si vuole chiudere. La funzione ritorna 0 se la chiusura è riuscita, mentre -1 se ci sono degli errori. La primitiva che permette la lettura di un file è, molto banalmente, la funzione `read`

```
ssize_t read(int fd, void* buf, size_t count);
```

Il primo argomento è il descrittore di file dal quale vogliamo leggere, il secondo argomento è il puntatore al buffer in cui si vuole scrivere i file e il terzo corrisponde al numero massimo di byte da leggere. Il ritorno della funzione corrisponde al numero di byte letti, o eventualmente, anche numeri negativi in caso di errori. La scrittura su un file utilizza la funzione `write`:

```
ssize_t write(int fd, const void* buf, size_t count)
```

Il primo parametro è il descrittore del file dove vogliamo scrivere, il secondo puntatore al buffer da cui leggere i dati da scrivere e il terzo argomento sono i parametri da scrivere. La funzione ritorna il numero di byte effettivamente scritti oppure un codice di errore (se il ritorno è negativo). Queste funzioni sono chiamate di sistema non bufferizzate, mentre la libreria standard del C mettono a disposizione delle funzioni bufferizzate, permettendo anche scrittura e lettura formattate.

## 2 Comunicazione mediante pipe

L'utilizzo di queste primitive di questi meccanismi trova ampio uso nella comunicazione tra processi, in particolare, quando si implementa il meccanismo delle **pipe**. Questa comunicazione permette

- Una comunicazione indiretta.
- Realizzare il concetto di mailbox nella quale si possono accodare dei messaggi in modalità FIFO.
- Una comunicazione monodirezionali, con un estremo per la lettura e uno per la scrittura, dove a ogni estremo è associato un file descriptor.

La creazione dei descrittori della pipe è affidata alla funzione `pipe`, la quale prende come ingresso un vettore di due elementi, che conterrà i descrittori della pipe; in particolare: salva in `fd[0]` l'estremo della pipe per la lettura, in `fd[1]` l'estremo da usare per la scrittura.

```
int pipe(int fd[2]);
```

Ritorna zero se ha successo, -1 altrimenti. I figli ereditano gli stessi file descriptor e possono utilizzarli per comunicare con il padre e gli altri figli. Tuttavia è necessario prestare attenzione: se il buffer è vuoto e c'è almeno uno scrittore attivo, allora la `read` è bloccante. Con scrittore attivo intendiamo un processo che ha aperto il file descriptor in scrittura. Il comportamento di write sull'estremità della pipe è bloccante se il buffer è pieno e restituisce errore se non ci sono lettori attivi. Diventa quindi fondamentale chiudere gli estremi non utilizzati, cioè, chiudere `pipedf[1]` nei lettori e chiudere `pipefd[0]` negli scrittori.

# 17

## Pilotare applicazioni

Abbiamo visto che STDIN, STDOUT e STDERR sono dei **descrittori di defaults**, cioè vengono generati **automaticamente** al momento della generazione del programma; inoltre, la loro apertura e la loro chiusura sono gestite direttamente dal sistema operativo. Esistono delle macro, che permettono l'accesso a queste strutture, all'interno della libreria `unistd.h`; in particolare, troviamo le **macro**:

- `STDIN_FILENO`. Macro che rappresenta il **file descriptor** dello standard input.
- `STDOUT_FILENO`. Macro che rappresenta il **file descriptor** dello standard output.
- `STDERR_FILENO`. Macro che rappresenta il **file descriptor** dello standard error.

Una funzione che permette di duplicare un file descriptor è la funzione `dup2`. Questa funzione prende in ingresso il file descriptor da duplicare (`target`) e il descrittore di file (`newfd`) dove verrà messa la copia di `target`

```
int dup2(int target, int newfd);
```

Il tipo di ritorno è negativo solo se si è verificato un errore. Inoltre, la funzione chiude il **file descriptor** di `newfd` prima di duplicare `target`. Attraverso l'utilizzo di pipe è quindi possibile redirigere il flusso dei dati dagli standard verso altri **file descriptor**; ad esempio

```
int main(){
    int pipe_fd[2];
    pid_t pid;

    pipe(pipe_fd);
    pid = fork();
    if(pid == 0){
        close(pipe_fd[0]);

        dup2(pipe_fd[1], STDOUT_FILENO);

        execl("/bin/ls", "ls", "-l", NULL);
    }

    if(pid > 0){
        char buffer[1024];
        int nread = -1;
        int index = 0;

        close(pipe_fd[1]);
```

```

while(nread != 0){
    nread = read(pipe_fd[0], &buffer[index],
                 sizeof(buffer)-1);

    buffer[index+nread] = '\0';
    index += nread;
}
printf("il padre ha letto %s\n", buffer);
}
return 0;
}

```

Il primo passo è quello di inizializzare le variabili relative al descrittore di pipe, cioè `pipe_fd`, e relative al processi figlio, quindi `pid`. Successivamente si applica la funzione `pipe` al descrittore `pipe_fd` in modo da inizializzarlo. A questo punto è possibile eseguire la funzione `fork`, creando il processo figlio. A questo punto:

- Il processo figlio chiude l'estremità non utilizzata, cioè quella di lettura. Successivamente si redirige lo standard output verso la pipe e, infine, eseguirà il comando `ls` con l'opzione `-l`. In questo modo il risultato del comando `ls` finirà all'interno della pipe, nell'estremità di lettura.
- Il processo padre chiuderà l'estremità in scrittura del pipe e utilizzerà un approccio generale per la lettura.

Quindi, supponiamo ora di avere un'applicazione che non è possibile modificare, poiché magari non abbiamo il sorgente e non si conosce il linguaggio utilizzato. Tuttavia l'applicazione permette di inserire degli input da terminale e restituire un output sul terminale. Attraverso la combinazione di `pipe` e `dup2` è possibile pilotare l'applicazione da terminale in modo abbastanza semplice:

- Si crea due pipe.
- Si crea un processo figlio.
- Si redirige STDIN e STDOUT del figlio.
- Si trasforma il figlio nell'applicazione.
- Il padre pilota il figlio seguendo quelle che sono le esigenze del programmatore e/o i vincoli dell'applicazione

# **Parte V**

# **Appendici**

# A

## Classificazione delle architetture

La tassonomia di Flynn classifica i sistemi operativi in funzione di due parametri variabili: **flussi di dati** e **flussi di esecuzione**. Con **flussi** intendiamo quanti flussi possiamo avere in parallelo all'interno della nostra architettura; in particolare

- Flusso multiplo di dati indica che all'interno della nostra architettura più flussi di dati possono essere elaborati allo stesso tempo.
- Flusso multiplo di istruzione indica che all'interno della nostra architettura possono coesistere più flussi di istruzioni diversi tra di loro in parallelo.

Questa architettura, in cui è prevista una distinzione dei circuiti e la distinzione della memoria, è detta **architettura di Harvard**. Combinando queste due classificazioni si ottiene una matrice che suddivide i sistemi in 4 classi:

- Macchine **SISD** (**Single Instruction Single Data**).
- Macchine **MISD** (**Multiple Instruction Single Data**).
- Macchine **SIMD** (**Single Instruction Multiple Data**).
- Macchine **SISD** (**Multiple Instruction Multiple Data**).

Nel modello **SISD** è presente un singolo flusso di istruzioni e un singolo flusso di dati. Questo è il modello classico della macchina di **Von Neumann**, utilizzato nei calcolatori convenzionali. In tale modello, in ogni istante temporale, viene eseguita una sola istruzione.

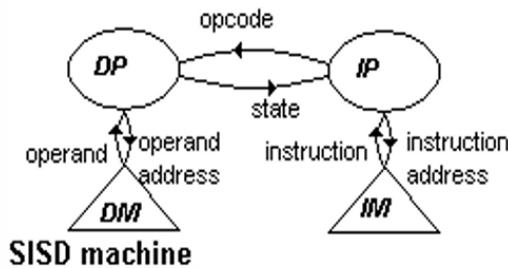


Figura A.1: Architettura di una macchina SISD. Il cerchio rappresenta un circuito in grado di eseguire istruzioni e/o di gestire dati, il triangolo rappresenta una memoria

Sulla destra troviamo il circuito denominato **Instruction Processor** (IP), destinato alla gestione del flusso di esecuzione. Esso è collegato in modo bidirezionale alla **Instruction Memory** (IM): tale collegamento permette all'IP di prelevare, dato un indirizzo, l'istruzione corrispondente.

Quando l'istruzione viene prelevata, l'**IP** ne interpreta il formato e trasmette al **Data Processor** (**DP**) l'**OPCODE** dell'istruzione, ricevendo in risposta un registro di stato. Il **DP** comunica analogamente con la propria memoria, detta **Data Memory** (**DM**), dalla quale legge e scrive i dati necessari all'esecuzione.

Le macchine **SIMD** (Single Instruction Multiple Data) dispongono di più unità di elaborazione (**Data Processor**), che eseguono contemporaneamente la stessa istruzione su flussi di dati differenti.

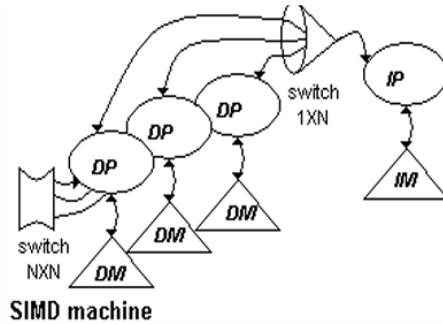


Figura A.2: Architettura di una macchina SIMD. Nell'immagine non è definita la rete di interconnessione tra i vari DP; per ora supponiamo che sia strutturata in modo da garantire comunicazioni regolari e senza conflitti

Ogni **Data Processor** ha la propria **Data Memory**, identificando così i diversi flussi di dati. Uno switch  $1 - n$  consente la trasmissione bidirezionale di **OPCODE** e **state** tra l'**Instruction Processor** e i vari **DP**. Inoltre, uno switch  $N \times N$  collega i diversi **DP** tra loro, permettendo lo scambio di dati.

Per mantenere elevato il grado di parallelismo, è necessario progettare programmi che minimizzino i trasferimenti tra memoria e processore. In questo modello si distinguono due tipi di parallelismo:

- **Parallelismo temporale:** diverse fasi di una singola istruzione vengono eseguite in pipeline.
- **Parallelismo spaziale:** la stessa operazione viene eseguita simultaneamente su un array di processori identici, sincronizzati dall'**IP**.

Le architetture **SIMD** trovano largo impiego nei supercomputer vettoriali e nei processori vettoriali (*vector processors*), specialmente per applicazioni che operano su grandi matrici.

Nel modello **MISD** (Multiple Instruction Single Data), più flussi di istruzioni vengono eseguiti simultaneamente sullo stesso flusso di dati. Ogni unità di elaborazione è associata a un proprio **Instruction Processor** e a una propria **Instruction Memory**, ma condivide un unico flusso di dati in ingresso. Questa categoria è principalmente teorica e trova applicazione limitata; un esempio approssimativo può essere una pipeline dell'elaborazione dati.

Nel modello **MIMD** (Multiple Instruction Multiple Data), diversi flussi di istruzioni operano su diversi flussi di dati, eseguiti in parallelo. Ogni **IP** e **DP** possiede la propria memoria e lavora in modo indipendente, garantendo massima flessibilità e parallelismo. Esistono due principali topologie di macchine **MIMD**:

- **MIMD a memoria distribuita:** ogni coppia **IP- DP** dispone di una propria memoria privata. I nodi comunicano attraverso una rete di interconnessione dedicata (switch  $N \times N$ ).
- **MIMD a memoria condivisa:** la memoria dati è centralizzata e condivisa tra i vari **DP**, interconnessi mediante uno switch efficiente.

Le macchine **MIMD** costituiscono la base dei moderni sistemi multiprocessore e multicomputer, incluse le architetture *cluster* e *massively parallel processing (MPP)*. Se si analizzasse la struttura di internet ci si potrebbe rendere facilmente conto che, nel suo complesso, è una macchina **MIMD**. Possiamo anche confrontare macchine **SIMD** e **MIMD**:

- Le **SIMD** richiedono meno hardware delle **MIMD**.
- Le **MIMD** usano spesso dei processori che sono *general-purpose*, riducendo i costi rispetto alle **SIMD**.
- Le **SIMD** usano meno memoria delle **MIMD**.
- Le **MIMD** godono di una grande flessibilità in termini di modelli computazionali supportati.

## 1 Metriche di prestazione

Per valutare l'efficienza di un sistema può essere utile definire delle metriche che stabiliscano dei criteri matematici per la valutazione delle prestazioni. Il primo di questi criteri è detto **speed-up** ed è definito come il guadagno di velocità rispetto ad una esecuzione su un processore:

$$S = \frac{T_1}{T_n}$$

Nel caso ideale lo speed-up dovrebbe essere lineare e crescere rispetto al numero di processori usati nella macchina parallela. Tuttavia ottenere un parallelismo ideale è quasi del tutto impossibile e, quindi, realisticamente lo speed-up non sarà mai lineare al numero di processori, ma  $S < N$ . Il valore dello speed-up dipende, oltre che dalle applicazioni, anche dalle architetture:

- Nel caso delle macchine SIMD è relativamente facile ottenere  $S = N$ .
- Nelle macchine **MIMD** è estremamente difficile fare crescere  $S$ .

Un'ulteriore metrica è quella dell'efficienza, definita come il **grado di parallelismo** che si riesce ad ottenere; matematicamente è definita come il rapporto tra lo **speed-up** e il **numero di processori**

$$E = \frac{S}{N} \tag{1}$$

Esiste anche una legge, chiamata **legge di Amhdal**, che teorizza che un parallelismo perfetto non è mai raggiungibile, poiché ci saranno sempre presenti sequenze di software intrinsecamente seriali. Inoltre, la legge di **Ahmdal** ridefinisce lo speed-up come

$$S = \frac{T_1}{T_{\text{seq}} + \frac{T_1 + T_{\text{seq}}}{N}}$$

Dove  $T_{\text{seq}}$  è definito come il tempo impiegato per eseguire istruzioni non parallelizzabili, come operazioni di **I/O**, costrutti condizionali e algoritmi intrinsecamente sequenziali.