

Sistemi Operativi

Corso di Laurea in Ingegneria Informatica, delle Comunicazioni
ed Elettronica

Mauro Brunato

Anno Accademico 2021-2022, secondo semestre
Ultima revisione: 30 maggio 2022

Caveat lector

Lo scopo principale di questi appunti è quello di ricostruire quanto detto a lezione. Queste note sono scombinate e incomplete, e la loro lettura non permette, da sola, di superare l'esame. Le fonti utili a ricostruire un discorso coerente e completo sono riportate alla pagina DidatticaOnLine del corso, dov'è disponibile anche la versione più recente di questo documento:

<https://didatticaonline.unitn.it/dol/course/view.php?id=34109>

Si suggerisce di confrontare la data riportata sul sito web con quella che appare nel frontespizio per verificare la presenza di aggiornamenti.

Le esercitazioni di laboratorio non sono contenute in questa dispensa perché riportate in dettaglio in documenti a sé stanti.

Indice

I Teoria	8
1 Definizione e storia	9
1.1 Che cos'è un sistema operativo?	9
1.2 Storia	9
1.2.1 Prima generazione (1940–1955): le valvole termoioniche	10
1.2.2 Seconda generazione (1955–1965): i transistor	11
1.2.3 Terza generazione (1965–1980): i circuiti integrati	12
1.2.4 Quarta generazione (dal 1980): i microprocessori	14
1.2.5 Quinta generazione (dal 1990?): i dispositivi mobili	15
2 Struttura hardware di riferimento	16
2.1 La CPU	16
2.1.1 Registri	16
2.1.2 Accesso alla memoria e all'I/O	16
2.1.3 L'unità di gestione della memoria	18
2.1.4 Modalità di esecuzione	18
2.1.5 Altre linee di controllo	19
2.2 La memoria principale	19
2.3 La memoria di massa e altri dispositivi di I/O	19
2.4 Il DMA (Direct Memory Access)	19
3 Funzioni di un sistema operativo moderno	21
3.1 Gestione dei processi	21
3.2 Gestione della memoria principale	21
3.3 Gestione della Memoria Secondaria	22
3.4 Gestione dell'I/O	22
3.5 Gestione dei File	22
3.6 Protezione	22
3.7 Sistemi Distribuiti	23
3.8 Tipi di sistema operativo	23
4 Architettura	25
4.1 Principi di progettazione	25
4.1.1 Requisiti	25
4.1.2 Policy e meccanismi	25
4.1.3 Realizzazione pratica	25
4.2 Struttura dei sistemi operativi	26
4.2.1 Strutture monolitiche	26
4.2.2 Strutture a strati	26
4.2.3 Sistemi a microkernel	28

4.2.4	Sistemi a moduli	29
4.3	Esempi	31
4.3.1	Mac OS X	31
4.3.2	Android	31
4.4	Macchine virtuali	32
4.4.1	Simulazione dell'hardware	32
4.4.2	Tipi di macchine virtuali	32
5	Processi e thread	34
5.1	Definizione	34
5.2	Le informazioni di stato di un processo	35
5.2.1	Il process control block	35
5.3	Lo scheduling	38
5.3.1	Le code di processi	38
5.3.2	Lo scheduler	40
5.4	Operazioni sui processi	40
5.4.1	Creazione	40
5.4.2	Terminazione di un processo	42
5.5	Il concetto di thread	42
5.5.1	Vantaggi	43
5.5.2	Tipi di thread	44
5.6	Comunicazioni fra processi	44
5.6.1	Memoria condivisa	45
5.6.2	Passaggio di messaggi	47
5.6.3	Esempi di sistemi a passaggio di messaggi	48
6	Scheduling	50
6.1	Lo scheduling a prelazione (preemptive)	50
6.1.1	Criteri per lo scheduling	51
6.2	Algoritmi di scheduling	51
6.2.1	First Come, First Served (FCFS)	51
6.2.2	Shortest Job First (SJF)	52
6.2.3	Esempio	52
6.2.4	Scheduling a priorità	54
6.2.5	Round-Robin (RR)	55
6.2.6	Code multilivello	56
6.2.7	Processi e applicazioni	58
7	Sincronizzazione fra processi	59
7.1	La regione critica	61
7.1.1	Soluzioni software	62
7.1.2	Soluzioni hardware	64
7.2	Semafori	66
7.2.1	Implementazione di un semaforo intero senza busy waiting	67
7.2.2	Altri usi dei semafori	70
7.2.3	Rischi nell'uso dei semafori	72
7.3	Altre primitive di sincronizzazione	74
7.3.1	I monitor	74
7.3.2	La parola chiave <code>synchronized</code> in Java	76
7.4	Analisi e prevenzione dei deadlock	76
7.4.1	Prevenzione	77
7.4.2	Prevenzione dinamica	78

8 Gestione della memoria	82
8.1 Indirizzamento e binding	82
8.2 Allocazione contigua	84
8.2.1 Partizioni fisse	85
8.2.2 Partizioni variabili	85
8.2.3 Compattamento	86
8.2.4 Il Buddy system	87
8.3 Paginazione	87
8.3.1 Protezione e condivisione	90
8.3.2 Dimensione dello spazio di indirizzamento: page table multilivello	90
8.3.3 Page table invertita	90
8.4 Segmentazione	91
8.5 Memoria virtuale	92
8.5.1 Paginazione su richiesta (demand paging)	92
8.5.2 Sostituzione delle pagine	94
8.5.3 Allocazione dei frame ai processi	95
9 Filesystem e memoria di massa	98
9.1 L'interfaccia del file system	98
9.1.1 I file	98
9.1.2 Le directory	99
9.2 Realizzazione	101
9.2.1 Strutture dati	102
9.2.2 Allocazione dello spazio su disco	102
9.2.3 Realizzazione delle directory	106
9.2.4 La gestione dello spazio libero	106
9.3 Efficienza e prestazioni	106
9.3.1 La cache del disco	107
9.3.2 Scheduling degli accessi al disco	109

II Esercizi 112

Changelog

- 2022-05-30 Indice delle prove scritte.
- 2022-05-21 Ultima parte di teoria, esercizi, alcune soluzioni.
- 2022-05-16 (secondo upload) Esercizi di paginazione su richiesta.
- 2022-05-16 Alcune soluzioni agli esercizi.
- 2022-05-09 Esercizi sulla paginazione della memoria.
- 2022-04-27 Esercizi sulla sincronizzazione; capitolo sulla gestione della memoria.
- 2022-04-03 Capitolo sulla sincronizzazione fra processi.
- 2022-03-28 Aggiunta di un esercizio sullo scheduling.
- 2022-03-24 Correzioni di errori e ritocchi alla parte sullo scheduling.
- 2022-03-21 Scheduling dei processi, primi esercizi sullo scheduling.
- 2022-03-09 Capitolo sui processi.
- 2022-03-03 Aggiunta di sezioni sui tipi di sistema operativo e loro architettura.
- 2022-02-27 Note per la prima settimana di lezione (lunedì 28 febbraio): presentazione del corso, cenni storici, architettura di riferimento, tipi di sistema operativo.

Informazioni sul corso

Organizzazione

Il corso si tiene in lingua italiana (sarà probabilmente in inglese dal prossimo anno accademico).

Docenti

- Teoria: Mauro Brunato mauro.brunato@unitn.it
- Laboratorio: Michele Grisafi michele.grisafi@unitn.it

Orario

Da lunedì 28 febbraio a lunedì 30 maggio.

- Lunedì dalle 13:30 alle 15:30 (2 ore), aula A106, teoria;
- Giovedì dalle 13:30 alle 16:30 (3 ore), aula A106, teoria;
- Venerdì dalle 08:30 alle 10:30 (2 ore), aula PC B106, laboratorio.

Sospensioni:

- Giovedì 14 aprile e venerdì 15 aprile: sospensione delle lezioni per prove intermedie;
- Lunedì 18 aprile: festa (Lunedì dell'Angelo);
- Lunedì 25 aprile: festa (Liberazione).

Materiale

Il materiale del corso è disponibile alla pagina

<https://didatticaonline.unitn.it/dol/course/view.php?id=34109>

Bibliografia

I testi non sono obbligatori, e la maggior parte degli argomenti è coperta da qualunque libro di base. È importante **non fidarsi delle sole dispense**, ma avere a disposizione qualche riferimento stampato o online. Ad esempio:

- ABRAHAM SILBERSCHATZ, PETER GALVIN, GREG GAGNE.
Operating System Concepts, 9th Edition.
John Wiley & Sons, Inc., 2005.
ISBN: 978-1-118-09375-7.

- ANDREW TANENBAUM, HERBERT BOS.
Modern Operating Systems, 4th Edition.
 Pearson Higher Education, 2014.
 ISBN: 978-1-292-06142-9.

Un altro testo utile, per seguire alcuni casi d'uso:

- DANIEL BOVET, MARCO CESATI.
Understanding the Linux Kernel, 3rd Edition.
 O'Reilly.
 ISBN: 978-0-596-00565-8.

Nota bene — Vanno benissimo anche le edizioni tradotte!

Prerequisiti

- Bagaglio matematico di base (analisi, algebra lineare)
- Programmazione (linguaggio C, shell Unix)

programma del corso

Teoria

Molti dettagli potranno variare in corso d'opera.

- Definizione e storia; panoramica sull'architettura — Ruolo del sistema operativo e sua evoluzione. Elementi architettonici. Struttura e funzioni di un sistema operativo.
- Processi: definizione, gestione, scheduling, sincronizzazione — Processi. Stati dei processi. Cambiamento di contesto. Creazione e terminazione di processi. Thread: thread a livello utente e a livello kernel. Cooperazione e comunicazione fra processi: memoria condivisa, messaggi. Comunicazione diretta ed indiretta. Scheduling: Modello a ciclo di burst di CPU-I/O. Scheduling a lungo, medio, breve termine. Scheduling con prelazione e cooperativo. Criteri di scheduling. Algoritmi di scheduling: FCFS, SJF, a priorità, HRRN, RR, a code multiple con e senza feedback. Valutazione degli algoritmi: modelli deterministici e probabilistici, simulazione.
- Gestione dei deadlock — Condizioni per l'innesto di un deadlock. Rappresentazione dello stato di un sistema con grafi di allocazione. Tecniche di prevenzione, rilevazione e ripristino. Algoritmo del banchiere.
- Gestione della memoria primaria — Indirizzamento logico e fisico. Rilocazione, binding degli indirizzi. Swapping. Allocazione contigua della memoria. Frammentazione interna ed esterna. Paginazione. Supporti hardware alla paginazione: TLB. Tabella delle pagine. Paginazione a più livelli. Segmentazione. Tabella dei segmenti. Segmentazione con paginazione.
- Gestione della memoria virtuale — Paginazione su richiesta. Gestione di page fault. Algoritmi di sostituzione delle pagine: FIFO, ottimale, LRU, approssimazioni LRU. Buffering di pagine. Allocazione di frame in memoria fisica, allocazione locale o globale. Thrashing. Località dei riferimenti. Modello del working set. Controllo della frequenza di page fault. Blocco di pagine in memoria.

- Memoria secondaria: struttura del disco, gestione del filesystem — Struttura logica e fisica dei dischi. Tempo di latenza. Scheduling del disco: algoritmi FCFS, SSTF, SCAN, C-SCAN, LOOK, C-LOOK. Gestione della memoria di paginazione. Strutture RAID. File System: Concetto di file, attributi e operazioni relative. Tipi di file. Accesso sequenziale e diretto. Concetto di directory. Struttura di directory. Protezioni nell'accesso a file. Attributi e modalità di accesso. Semantica della consistenza. Struttura di un file system. Montaggio di un file system. Metodi di allocazione dello spazio su disco: contiguo, concatenato, indicizzato. Gestione dello spazio libero su disco: tramite vettore di bit, tramite liste. Realizzazione delle directory: liste lineari, tabelle hash.
- Sottosistema di input/output — Sistemi di Input/Output, Hardware per I/O. Tecniche di I/O: programmato, con interrupt, con DMA. Device driver e interfaccia verso le applicazioni. Servizi del kernel per I/O: scheduling, buffering, caching, spooling.
- Sicurezza e Protezione — ACL/Capability, login, protezione disco, secure boot. Malware, vulnerabilità e difese tipiche dei sistemi operativi.

Laboratorio

- Richiami e approfondimenti sul linguaggio C — Uso di gcc e make
- Il sistema operativo Linux — Installazione, chiamate di sistema, primitive di sincronizzazione
- Progetti e assignment

Ulteriori dettagli saranno forniti dal docente di laboratorio.

Valutazione

Il voto finale del corso sarà determinato come media aritmetica di due prove:

- una prova scritta di teoria con esercizi e domande aperte (5 appelli);
- una prova pratica di laboratorio (3 appelli).

Entrambe le prove dovranno essere superate con un punteggio di almeno 18/30. Il voto di ciascuna prova può essere conservato per gli appelli successivi, viene invalidato dalla partecipazione a una prova successiva.

Domande/ricevimento

Si raccomanda di utilizzare le lezioni frontali come momento privilegiato per domande e discussioni. È possibile utilizzare il forum del corso su didatticaonline. I docenti sono ovviamente disponibili per colloqui privati e ricevimenti di gruppo, da concordare via email.

Parte I

Teoria

Capitolo 1

Definizione e storia

1.1 Che cos'è un sistema operativo?

Un sistema operativo (d'ora in poi SO oppure OS) è un insieme di programmi che fa da intermediario fra l'hardware (processore, memoria, periferiche) e i programmi utente.

- Gestisce e rende efficiente l'uso delle risorse hardware
- Astrae l'hardware esponendo un'interfaccia uniforme per il suo utilizzo (file, allocazione della memoria, rete)

Come fa?

- Modalità “privilegiata” (kernel mode, supervisor mode) con istruzioni macchina precluse ai programmi utente.
- I programmi utente operano in “user mode” e possono invocare funzioni esposte dal SO.

Si osservi che le risorse da allocare includono il processore stesso (i processi competono per lo stesso “esecutore”, oltre che per la memoria e le periferiche).

Il processore “sa” (per mezzo di un flag interno) se si trova in modalità utente o kernel, e il tentativo di eseguire istruzioni privilegiate in user mode può generare una “trap” (interruzione software) che passa il controllo al SO.

In Figura 1.1, il sistema operativo propriamente detto è la fascia che poggia direttamente sull'hardware e che opera in kernel space. Storicamente, un SO è sempre accompagnato e fortemente integrato con numerose librerie in user space (librerie condivise, “shared libraries”) che semplificano le operazioni più comuni e da una o più interfacce dirette verso l'utente (shell testuali o grafiche).

1.2 Storia

In questa sezione possiamo osservare come gli sviluppi tecnologici abbiano portato a macchine sempre più potenti ed economiche, aprendo di continuo nuovi campi applicativi; la conseguenza è stata lo sviluppo di software di sistema sempre più potente, complesso e versatile, sempre accompagnato da nuove idee e concetti che svilupperemo durante il corso.

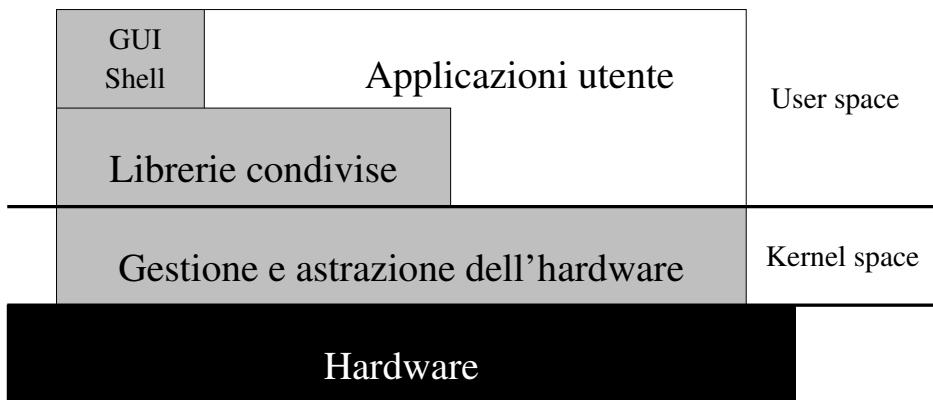
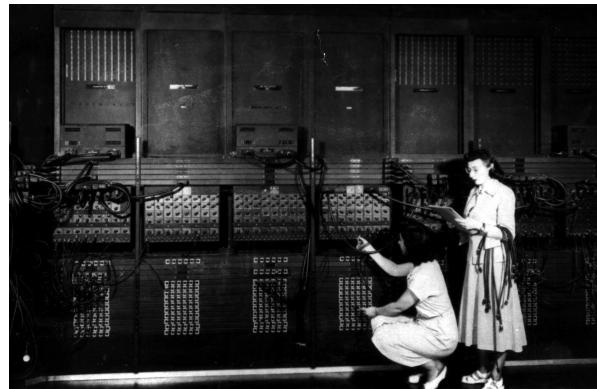


Figura 1.1: Collocazione del SO propriamente detto e delle librerie e programmi utente che lo accompagnano

1.2.1 Prima generazione (1940–1955): le valvole termoioniche



*Two early programmers
(Gloria Ruth Gordon [Bolotsky] and Esther Gerston)
at work on the ENIAC.
US Army photo from the archives of the ARL Library
(US Army Research Laboratory).*

- Elementi attivi: valvole termoioniche (primi prototipi con relé elettromeccanici) → enormi, molto calore generato, estrema fragilità.
- Non c'è S.O.; spesso la programmazione avviene ricablando i circuiti, o agendo su interruttori di una console.
- Sviluppo dei primi sistemi di I/O (schede e nastri perforati, stampanti).

Limitazione Lunghi tempi di impostazione del programma, computer spesso in attesa → Molto tempo di calcolo sprecato.

Tabella 1.1: Struttura di un job FMS. I comandi di controllo hanno il simbolo \$ in prima colonna.

\$JOB	<i>Inizio del job</i>
\$FORTRAN	<i>Invocazione del compilatore</i>
...	
...	<i>Programma FORTRAN</i>
...	
\$LOAD	<i>Carica il programma</i>
\$RUN	<i>Esegui il programma</i>
...	
...	<i>Dati per il programma</i>
...	
\$END	<i>Fine del job.</i>

1.2.2 Seconda generazione (1955-1965): i transistor



IBM 7090 operator's console at the NASA Ames Research Center in 1961, with two banks of IBM 729 magnetic tape drives. (NASA).

- Transistor: aumento radicale dell'affidabilità, della velocità e della densità circuitale → commercializzazione presso grosse banche, università, centri governativi.
- Il S.O. è poco più di un interprete di comandi
 - Fortran Monitor System (IBM) include un interprete di comandi e un compilatore FORTRAN.
 - Le due componenti vengono caricate insieme in memoria.

I sistemi batch

Problema Tempi d'attesa dovuti alla lettura delle schede e alla stampa, potenza di calcolo ancora sottoutilizzata. In Figura 1.2, i tempi di utilizzo della CPU (fascia centrale) sono molto ridotti a causa delle attese per le operazioni di I/O.

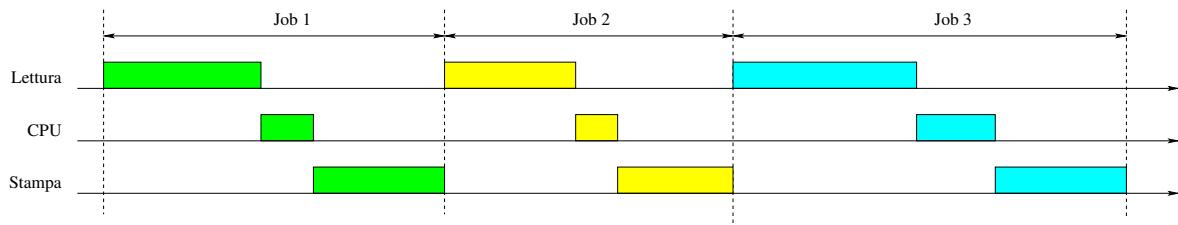


Figura 1.2: Lunghi periodi di inattività per la CPU.

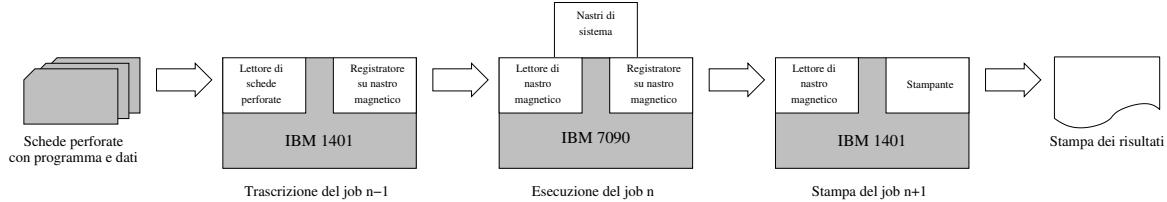


Figura 1.3: Un sistema batch: una pipeline di dispositivi per gestire le operazioni IO in parallelo al calcolo.

Soluzione Una batch pipeline (Figura 1.3):

- un sistema elettromeccanico legge più job da schede perforate e li registra su nastri (più veloci)
- Il mainframe legge e scrive nastri magnetici
- Un altro sistema elettromeccanico stampa il contenuto dei nastri di output.

Le tre fasi (lettura del programma e dei dati, compilazione ed esecuzione, stampa dell'output) avvengono in parallelo su sistemi separati
→ Utilizzo più efficiente della potenza di calcolo.

Ulteriore ottimizzazione (Figura 1.4):

- Moltiplicare i dispositivi di I/O.
- La CPU, costosissima, resta unica e opera sequenzialmente.

1.2.3 Terza generazione (1965–1980): i circuiti integrati

- Circuiti integrati: densità circuitale in crescita, sistemi sempre più complessi
→ Nuove applicazioni (p.es. gestionali).
- Nascita dei minicomputer ($\sim 100.000\$$)
- Il sistema batch non si adatta bene a programmi con più cicli di I/O.

Multiprogrammazione

- Ripartizione delle risorse (memoria) fra più job.
- Mentre un job è in attesa di input, eseguirne un altro.
- Simultaneous Peripheral Operation On Line (**spooling**).

Esempio: **OS/360**, colossale progetto di IBM in continua evoluzione per coprire tutti i dispositivi offerti.

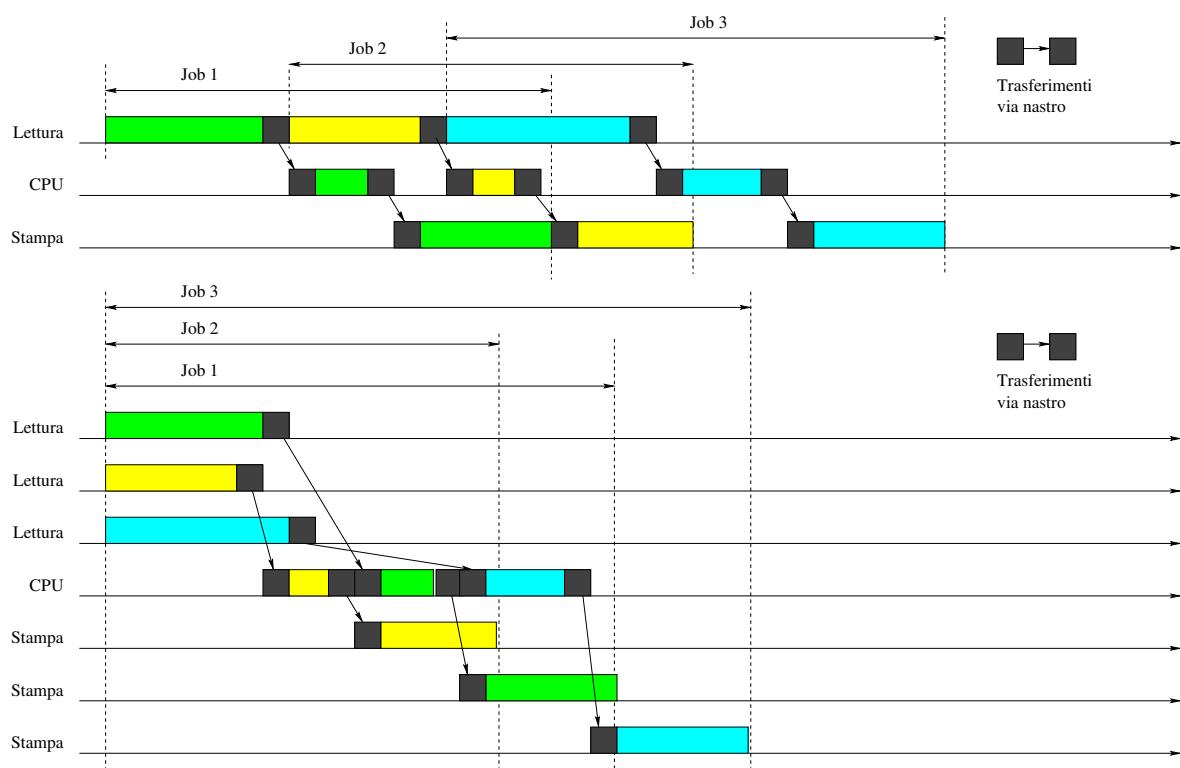


Figura 1.4: Lo scorporo delle operazioni di I/O rende l'uso della CPU più efficiente. Sopra: con un solo dispositivo di input e uno di output; sotto: con più dispositivi di I/O.

Time sharing

- Utenti su terminali online, la CPU passa da un processo all'altro.
- Necessità di proteggere gli ambienti di processi diversi

Esempi:

- Preesistente: CTSS (Compatible Time Sharing System, MIT 1962)
 - Sistema sperimentale (MIT).
 - Hardware: IBM 7094 modificato
- MULTICS (Multiplexed Information and Computing Services, MIT/BellLabs/GE)
 - Potenza di calcolo as-a-service.
 - Ha introdotto numerose idee fondamentali.
 - Troppo complesso, scritto in un linguaggio (PL/I) troppo nuovo e con compilatori non ancora efficienti.
- Unix
 - Nato nei Bell Labs come sistema monoutente (Uniplexed Information and Computing System - vedi MULTICS)
 - Presto evoluto in sistema multitasking, multiutente.
 - Un sistema di licenze del codice sorgente ha portato alla creazione di molte versioni.
 - Standard IEEE POSIX per uniformare i dialetti e permettere interoperabilità.
 - Principali dialetti: System V (AT&T), BSD (Berkeley Software Distribution), Linux (che useremo in laboratorio).
 - Mac OS X si basa su BSD, Android poggia su un sistema Linux.

1.2.4 Quarta generazione (dal 1980): i microprocessori

- LSI (Large Scale Integration), microprocessori (CPU su un singolo chip)
→ microcomputer, personal computer.
- Diffusione dei dischi magnetici (floppy disk)
- CP/M (Control Program for Microcomputers)
 - Sviluppato da Digital Research nel 1977, basato su processori Intel (8080) e loro estensioni (Zilog Z80, in seguito anche Intel 8086 con CP/M86).
 - Dominante per quasi un decennio, soppiantato da MS-DOS.
- MS-DOS (Disk Operating System)
 - Sviluppato da Microsoft nel 1982 per IBM PC basato su processori Intel (8086) e loro successori (80286).
 - Ha dato la stura a tutta la successione di sistemi operativi Microsoft Windows, da Windows 98 in poi.
- Apple Macintosh System Software (in seguito Mac OS)
 - Sviluppato da Apple per MacIntosh nel 1984 su processore Motorola 68000.

- Sostituito da Mac OS X (basato su BSD Unix) nel 1999.
- Primo sistema di successo con interfaccia utente grafica, focus su user friendliness.
- Dialetti Unix per microcomputer
 - Microsoft XENIX, Apple A/UX, IBM AIX, Apple Mac OS X, Linux, FreeBSD...

1.2.5 Quinta generazione (dal 1990?): i dispositivi mobili

- Mobile Computing
 - Symbian OS (Samsung, Sony Ericsson, Motorola, Nokia)
 - Blackberry OS (Research In Motion, 2002)
 - Varie incarnazioni di Windows per sistemi mobili
 - Apple iOS (iPhone, 2007)
 - Android (Google, 2008)
 - * Linux con macchina virtuale Java-like per applicazioni utente.

Capitolo 2

Struttura hardware di riferimento

In questo corso faremo riferimento a un’architettura standard con una (o più) CPU, memoria centrale, periferiche, collegate tra loro da un bus che trasporta dati, indirizzi, vari segnali di controllo. Vedere figura 2.1 per una rappresentazione semplificata.

2.1 La CPU

Contiene la logica di controllo e di calcolo (**ALU**, **FPU**) e un numero limitato di **registri** per contenere dati, indirizzi di memoria, informazioni di stato. Comunica con il resto del calcolatore attraverso un **bus** nel quale espone indirizzi e scrive o legge i relativi dati. Gli indirizzi possono fare riferimento alla memoria (per comunicare con la RAM) o a “porte” di I/O (per le periferiche). La distinzione fra operazioni di lettura e scrittura e tra memoria e porte I/O, come pure la notifica di vari stati, è operata attraverso alcune linee di controllo.

2.1.1 Registri

Ricordiamo almeno i seguenti tipi:

- Uso generale: le istruzioni macchina fanno riferimento a questi registri per caricare dati dalla memoria o dalle periferiche, o per conservare/aggiornare puntatori alla memoria.
- Flag: conservano lo stato del processore, informazioni sull’esito delle operazioni.
- Program counter: punta all’indirizzo di memoria contenente la prossima istruzione da eseguire.
- Stack pointer: punta allo stack di esecuzione (accesso LIFO per memorizzare indirizzi di ritorno delle subroutine, parametri e variabili locali delle chiamate); normalmente associato anche a un base pointer per contrassegnare locazioni importanti all’interno dello stack.

2.1.2 Accesso alla memoria e all’I/O

Tralasciando per il momento il ruolo della MMU (vedi più avanti), le due operazioni principali sono:

- Lettura:
 - La CPU imposta l’indirizzo sull’apposito bus, imposta un bit sul bus di controllo (ad esempio, alza una linea RD) per indicare che si tratta di un’operazione di lettura e che il suo lato del bus dati rimane quindi ad alta impedenza, e per segnalare che l’indirizzo è valido.
 - La RAM risponde impostando il dato desiderato sul bus dati e segnalando che il dato è disponibile attraverso una linea (p.es. DATA) del bus di controllo.

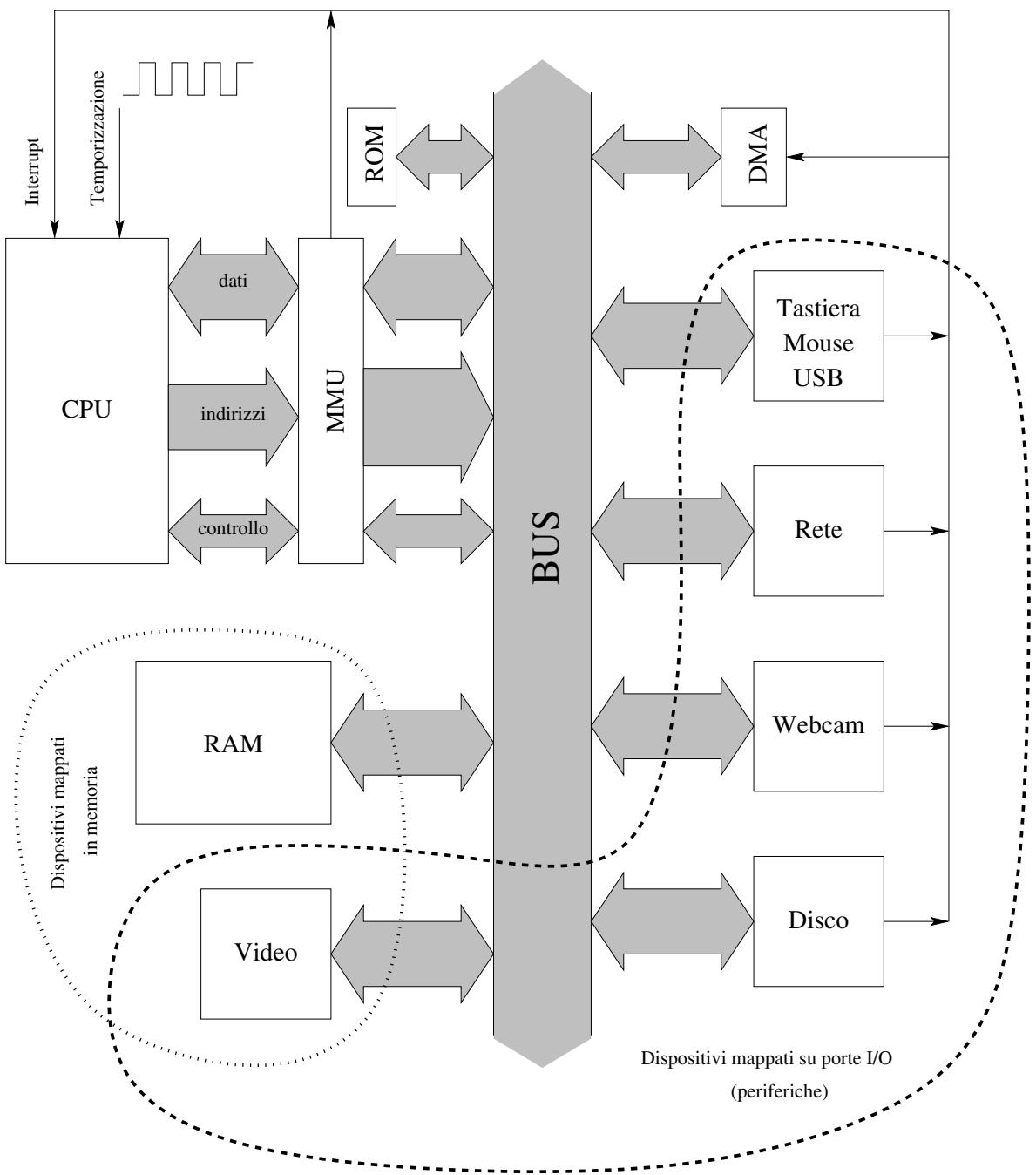


Figura 2.1: Architettura di riferimento.

- La CPU attende la segnalazione che il dato è valido, lo trasferisce a un registro interno, libera il bus degli indirizzi e abbassa la linea di segnalazione.

- Scrrittura:

- La CPU imposta l'indirizzo e il dato da scrivere sugli appositi bus, imposta un bit sul bus di controllo (ad esempio, alza una linea WR) per indicare che si tratta di un'operazione di scrittura, che il suo lato del bus dati è attivo e che il contenuto del bus è valido.
- La RAM risponde copiando il dato sull'indirizzo desiderato e segnala (attraverso una linea del bus di controllo) che il bus può essere liberato.
- La CPU attende la segnalazione e libera il bus.

Alcune segnalazioni di validità del bus indirizzi e dati potrebbero non essere presenti, se il sistema è temporizzato in modo da tenere conto dei ritardi di propagazione attraverso il bus e i circuiti logici. Questa descrizione non tiene nemmeno conto dei diversi livelli di caching che possono servire a velocizzare l'accesso alla RAM “avvicinando” determinate zone di memoria alla CPU.

Da un punto di vista circuitale, le operazioni sulle porte di I/O hanno la stessa struttura; l'unica differenza è il valore su una linea di controllo che specifica quale “spazio di indirizzamento” usare¹. Dal punto di vista del codice macchina, però, le operazioni I/O sono gestite da istruzioni diverse da quelle per l'accesso alla memoria.

2.1.3 L'unità di gestione della memoria

La memoria è strutturata (al netto della cache) come una lunga schiera di valori, individuati da indirizzi numerici consecutivi. Quest'organizzazione piatta, però, è rischiosa se alterniamo più processi: uno di questi potrebbe, intenzionalmente o per errore, corrompere strutture dati non proprie.

Soluzione La CPU non accede direttamente alla memoria: gli indirizzi passano attraverso un dispositivo, la Memory Management Unit (MMU) che, per mezzo di tabelle, li traduce in indirizzi “fisici”.

- Usando tabelle diverse per ogni processo, ciascuno ha un proprio spazio di indirizzamento. Lo stesso indirizzo logico (lato CPU) corrisponde a indirizzi fisici diversi a seconda della tabella usata.
- È possibile segnalare errori se un indirizzo “logico” eccede le dimensioni della memoria.
- La traduzione avviene in hardware, quindi non rallenta (significativamente) il processo.

Spesso la MMU è interna al microprocessore.

Parleremo del ruolo della MMU trattando i processi e la gestione della memoria principale.

2.1.4 Modalità di esecuzione

Un processore moderno (non embedded) ha almeno due modalità di funzionamento:

- Privilegiata (supervisore): tutte le istruzioni sono disponibili.
- Normale (utente): alcune istruzioni sono precluse. In particolare, quelle che riguardano la lettura/scrittura su porte, la configurazione della MMU, l'impostazione di alcuni registri e flag interni, istruzioni particolarmente distruttive come HALT...

¹Spesso però in prossimità della CPU sono collocati controllori che separano il bus di memoria dal bus I/O, ad esempio i chip Northbridge e Southbridge delle architetture Intel

I processori x86 hanno quattro modalità organizzate gerarchicamente (dette anche “protection ring”, identificate dal valore di due bit in un registro), ma la maggior parte dei sistemi usa solo le due estreme ($0 \rightarrow$ supervisore, $3 \rightarrow$ utente).

Il passaggio dalla modalità privilegiata a quella utente avviene tramite la modifica di alcuni Flag, oppure tramite alcune istruzioni come il ritorno da un interrupt (RTI, vedi sotto).

Il passaggio inverso richiede eventi esterni (segnali di interrupt), alcuni tipi di errori (divisione per zero), oppure istruzioni TRAP. Ad ogni evento di questo tipo, il processore passa in modalità privilegiata e salta a una subroutine il cui indirizzo è reperito in una tabella.

2.1.5 Altre linee di controllo

Come visto sopra, il processore può sospendere l'esecuzione di un processo alla ricezione di un segnale di interrupt per passare a gestire compiti non differibili (ad esempio, l'arrivo di un nuovo pacchetto di rete).

Inoltre, trattandosi di un circuito sequenziale, la CPU deve ricevere un segnale di temporizzazione (clock).

2.2 La memoria principale

Solitamente, la CPU ha accesso a una ROM relativamente ridotta nella quale è contenuto il programma di bootstrap (BIOS, Basic Input/Output System), e a una quantità di memoria RAM.

Un altro dispositivo che espone indirizzi di memoria è la scheda video: il display è solitamente configurabile attraverso porte di I/O, ma la schermata da visualizzare è mappata in memoria.

2.3 La memoria di massa e altri dispositivi di I/O

La CPU configura e accede a questi dispositivi leggendo e scrivendo porte I/O. La comunicazione con questi dispositivi può essere:

- sincrona, nel qual caso la CPU legge i dati accedendo a specifiche porte I/O; ad esempio tramite polling; ricordiamo che le istruzioni di I/O sono solitamente privilegiate, quindi un programma utente dovrà invocare una routine di sistema tramite TRAP;
- asincrona: quando un dispositivo possiede dei dati da mettere a disposizione della CPU, invia un interrupt. La CPU passa a una subroutine privilegiata (in quanto invocata da un interrupt fisico) che accede alla porta e legge i dati.

2.4 Il DMA (Direct Memory Access)

Come visto nell'excursus storico, un fattore che limita l'efficienza con cui si può usare la CPU è la necessità di sospenderne il funzionamento per effettuare operazioni di copia fra memoria e dispositivi di I/O (ad esempio il caricamento di programmi o di dati dal disco).

Queste operazioni possono essere svolte efficacemente da un opportuno dispositivo (il DMA) che sfrutta i periodi in cui la CPU non utilizza il bus. Il funzionamento di base è il seguente:

- la CPU imposta, attraverso alcune porte di I/O, un'operazione di copia (tra sezioni di memoria, tra dispositivi di I/O, o dall'uno all'altro), poi passa a fare altro.
- Il DMA è strettamente sincronizzato con la CPU: tra un ciclo di accesso alla memoria (fetch delle istruzioni, lettura di un operando, scrittura di un dato) e l'altro, il DMA utilizza il bus per il trasferimento (interleaving fra le operazioni della CPU e quelle del DMA).

- Al termine del trasferimento, il DMA notifica la CPU tramite polling o interrupt.

Riassumendo, i trasferimenti di dati sono fittamente alternati con i normali cicli di lettura/scrittura della CPU.

Capitolo 3

Funzioni di un sistema operativo moderno

3.1 Gestione dei processi

Un **processo** è un programma in esecuzione.

Per svolgere il proprio compito, un processo deve poter accedere alle risorse (in primo luogo le CPU, poi la memoria, il disco, il display...).

Un processo viene eseguito sequenzialmente, un'istruzione per volta, e compete con altri processi per l'uso delle risorse. Di norma appartiene a un utente, che ne ha richiesta l'esecuzione ed è figlio di un processo che l'ha avviato.

Responsabilità del sistema operativo

- Gestione del ciclo di vita dei processi (creazione, distruzione, sospensione, ripresa).
- Divisione del tempo CPU fra i processi (scheduling) in base alle necessità e alla priorità.
- Sincronizzazione e comunicazione fra processi.
- Fornitura di meccanismi per evitare blocchi reciproci (deadlock).

3.2 Gestione della memoria principale

La memoria principale è veloce, ma è volatile, costosa e limitata. In virtù della sua velocità, è il contenitore di tutto il codice e di tutte le strutture dati su cui operano i processi (per eseguire un processo è prima necessario caricarlo nella memoria principale).

È un contenitore unico e piatto (basta fornire un indirizzo e viene restituito il contenuto), ma la MMU permette di imporre delle regole.

Responsabilità del sistema operativo

- Allocazione e rilascio dello spazio di memoria secondo le necessità dei processi.
- Spostamento verso il disco di aree di memoria raramente usate per liberare spazio per altri processi (memoria virtuale).
- Gestione della frammentazione dello spazio libero in presenza di continue allocazioni e rilasci.

3.3 Gestione della Memoria Secondaria

La memoria secondaria è permanente ed economica, ma lenta.

Costituita da una o più “unità a disco” (dischi magnetici, ma sempre più spesso SSD - solid state disks). Ha una struttura più complessa della memoria principale (tracce, settori, facce, unità di allocazione).

Responsabilità del sistema operativo

- Allocazione dello spazio libero su disco.
- Gestione degli accessi.
- Organizzazione di gruppi di dischi in unità logiche più estese o per aumentare la ridondanza (RAID).

3.4 Gestione dell'I/O

I dispositivi di I/O collegati a un PC sono numerosi ed eterogenei. Ad esempio:

- dispositivi orientati al carattere con un flusso continuo di dati (tastiera, mouse),
- dispositivi orientati ai blocchi, che operano tramite scambio di pacchetti dati (scheda di rete, disco).

Responsabilità del sistema operativo

- Nascondere (astrarre) le caratteristiche fisiche specifiche di un dispositivo offrendo interfacce uniformi a basso livello (device driver).
- Offrire un piccolo numero di interfacce ad alto livello in grado di coprire tutte le modalità di interazione.
- Gestire l'accumulo di dati in attesa che un processo se ne faccia carico (buffering).

3.5 Gestione dei File

A differenza della memoria principale, la memoria di massa può avere strutture molto varie e complesse. Per uniformarne l'uso si ricorre a un'astrazione logica uniforme fra i vari dispositivi, il file.

Un file costituisce una raccolta di informazioni in qualche modo correlate fra loro (programmi, dati).

I file sono organizzati in strutture gerarchiche (directory)

Responsabilità del sistema operativo

- Gestione dei file e delle directory (allocazione, copia, cancellazione, lettura, scrittura).
- Reperimento di un file sulla base di informazioni di accesso (path);
- Backup e ripristino.

3.6 Protezione

Il sistema operativo deve essere in grado di gestire l'accesso alle risorse vista sopra sulla base dell'identità dei processi e degli utenti che li “possiedono”.

Responsabilità del sistema operativo

- Autorizzazione all'accesso
- Fornire strumenti per verificare le politiche di accesso.

3.7 Sistemi Distribuiti

Un “Sistema distribuito” è una collezione di elementi di calcolo che non condividono né la memoria né un clock: le diverse risorse sono connesse tramite una rete e scambiano pacchetti di dati.

Responsabilità del sistema operativo

- Gestione e sincronizzazione delle varie componenti (processi distribuiti, memoria distribuita, filesystem distribuito...).

3.8 Tipi di sistema operativo

Esempi dal Tanenbaum

Mainframe Orientati all'I/O, al trasferimento dati, all'elaborazione di grosse moli di dati.

- Batch processing: elaborazione di job di routine senza supervisione dell'utente (rapporto vendite quotidiano/mensile di una grossa catena di negozi).
- Transaction processing: brevi operazioni da gestire rapidamente (operazione su conto bancario, transazione con carta di credito, aggiornamento dell'archivio dopo una vendita al dettaglio).
- Timesharing: job richiesti dagli utenti da eseguire in “parallelo”

Per quanto apparentemente desueti, hanno ancora mercato presso grosse aziende/banche. Esempio: IBM OS/390, z/OS?

Server Servono più utenti/richieste via rete. Permettono la condivisione di hardware e software. Possono offrire:

- periferiche: stampa, disco;
- tempo di calcolo, o hardware di calcolo specifico (da GPU a quantum computing)
- servizi web.

Esempi: Linux, Windows Server, Solaris...

Multiprocessore Orientati a bilanciare il carico fra più CPU, funzionalità ormai presente anche nei sistemi per PC, spesso multiprocessore o multicore.

Personal computer Enfasi sull'interfaccia utente.
Linux, Mac OS X, Windows, FreeBSD...

Sistemi mobili Bassa potenza di calcolo, memoria più limitata, restrizioni nel consumo energetico. Enfasi sull'interfaccia utente.
Android, iOS.

Embedded Spesso fatti per macchine ancora meno potenti, con la garanzia che l'hardware e soprattutto il software che devono eseguire non cambierà ed è fidato. Spesso completamente in ROM (“firmware”), a volte semplicemente un bootloader.

Embedded Linux, VxWorks...

Real-time Per applicazioni critiche. Offrono garanzie sui tempi di utilizzo delle risorse, ma richiedono altrettante garanzie dal software che mandano in esecuzione.

Capitolo 4

Architettura

4.1 Principi di progettazione

Rientrano nell'ambito dell'ingegneria del software, ma è importante evidenziare alcuni punti.

4.1.1 Requisiti

Ad alto livello, i requisiti sono ovvi.

Dal punto di vista dell'utente UI facile da imparare e da usare, affidabile, sicuro, veloce.

Dal punto di vista dei progettisti e dei programmatore facile da progettare, realizzare e mantenere; efficiente, privo di errori.

Il problema è mappare questi requisiti su una serie di definizioni chiare. Questo passo dipende anche dalla natura del SO: in un SO real-time l'affidabilità ha un senso diverso da quella di un SO utente.

4.1.2 Policy e meccanismi

Ovvero, **cosa** si vuole fare (politica, policy) e **come** si intende farlo (metodo, meccanismo).

Esempio si desidera un meccanismo che periodicamente interrompe l'esecuzione di un processo utente per dedicare tempo CPU ad altri compiti. Questa è una decisione di policy. Il meccanismo attraverso il quale si applica questa policy può essere un timer che genera un interrupt. Il periodo di questo timer è, nuovamente, una decisione di policy.

La distinzione è importante perché le policy possono cambiare (ad esempio, l'interruzione può avvenire con periodi diversi a seconda del contesto) mantenendo inalterati i meccanismi; viceversa, nuovi sviluppi nell'hardware possono portare a implementare nuovi meccanismi in grado di applicare una policy (ad esempio, l'aggiunta di capacità di interrupt in un'architettura che prima accettava sospensioni solamente per I/O).

4.1.3 Realizzazione pratica

I SO sono collezioni di numerosi programmi, generalmente scritti lungo archi di tempo estesi da molte persone.

Inizialmente scritti direttamente in linguaggio macchina (per ragioni di efficienza, e per la stretta associazione fra hardware e sistema operativo dei primi sistemi).

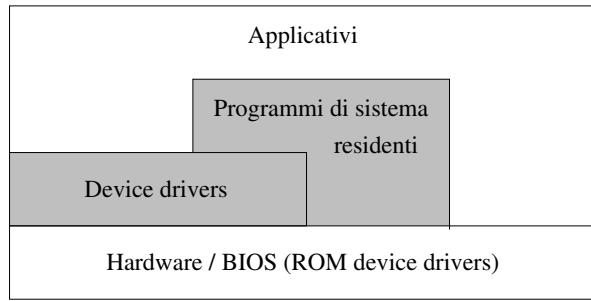


Figura 4.1: Struttura a blocchi di MS-DOS

Ora tipicamente utilizzano un linguaggio ad alto livello (C, C++), spesso con l'uso di linguaggio macchina (assembly) per le routine più critiche. Le ragioni per l'uso di codice ad alto livello sono le stesse applicabili a tutti i programmi:

- Facilità di scrittura, debug, manutenzione;
- portabilità attraverso diverse architetture;
- non ultimo, i compilatori moderni sono ormai molto efficienti nell'ottimizzare il codice, quindi la perdita di efficienza è limitata.

Spesso si inizia con la scrittura ad alto livello; in seguito, operazioni di profiling identificano i colli di bottiglia e si riscrivono le parti più critiche, eventualmente in assembly.

4.2 Struttura dei sistemi operativi

4.2.1 Strutture monolitiche

Alcuni ambienti, ad esempio processori senza meccanismi di protezione, non permettono la realizzazione di SO particolarmente strutturati.

Un esempio è MS-DOS, la cui struttura è riassunta in Fig. 4.1. Le interfacce e il livelli delle funzionalità non sono ben separati a causa della semplicità del processore Intel 8088. I programmi applicativi, se ad esempio devono scrivere su video, hanno più scelte:

- ricorrere a routine residenti installate dal sistema operativo — accesso uniforme e indipendente dall'hardware;
- ricorrere a routine presenti nella ROM della macchina, spesso un'estensione fisicamente installata nella scheda video — accesso efficiente, ma più complesso perché dipendente dall'hardware;
- operare direttamente sull'hardware.

Una struttura simile è applicabile ai primi sistemi Unix, vedi Fig. 4.2: il numero ridotto di strati e di componenti distinte rende il sistema difficile da mantenere, ma riduce notevolmente l'overhead nelle interazioni con l'hardware (meno chiamate prima di arrivare al bare metal).

4.2.2 Strutture a strati

Un principio molto importante nella realizzazione di un sistema operativo è il seguente:

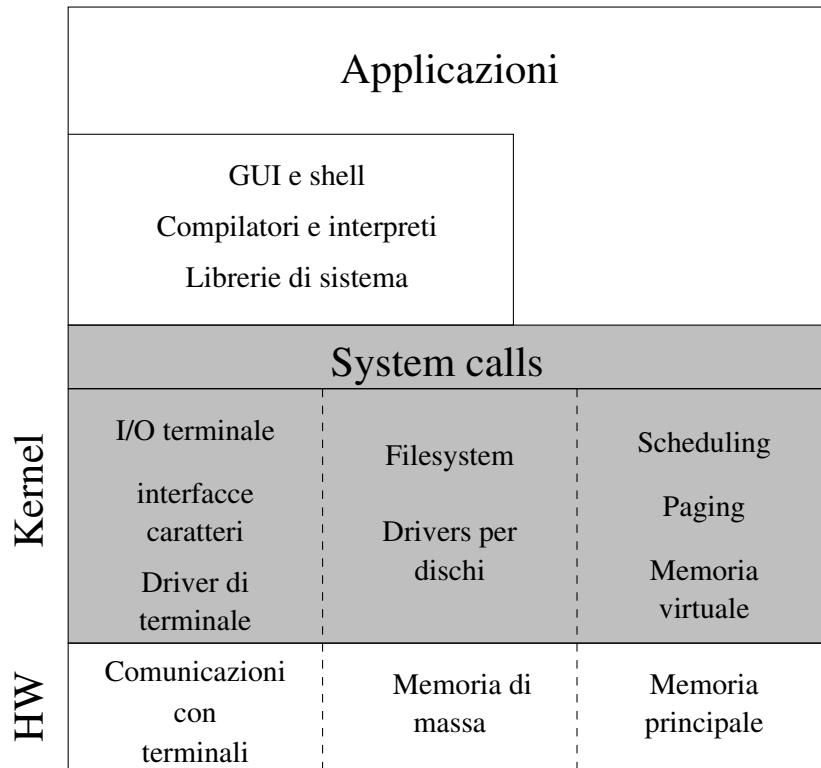


Figura 4.2: Struttura tradizionale di Unix

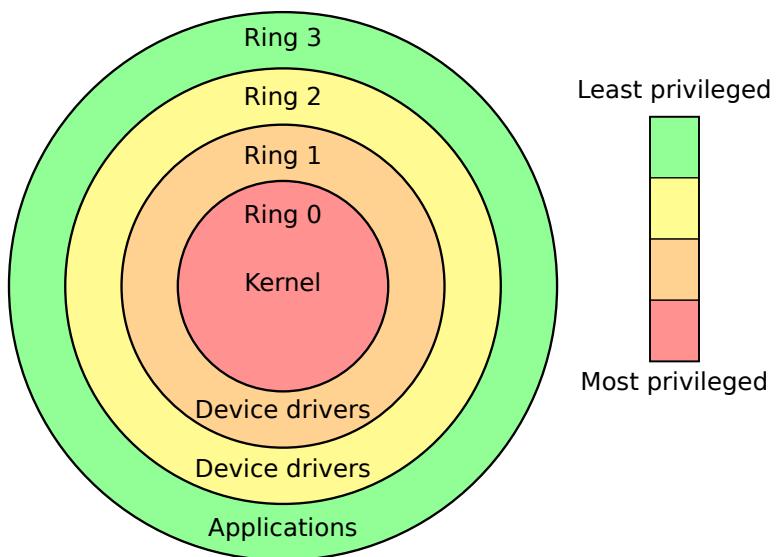


Figura 4.3: I quattro livelli di privilegio di un'architettura Intel (da https://en.wikipedia.org/wiki/Protection_ring).

Principio del minimo privilegio (Principle of Least Privilege, PoLP, o anche Principle of Least Authority, PoLA) *Ogni componente dovrebbe avere il minimo livello di privilegio necessario a svolgere la propria funzione.*

Ad esempio, consideriamo la struttura a quattro livelli (ring) delle architetture Intel mostrata in Figura 4.3. Mentre lo scheduler dei processi dovrà probabilmente avere tutti i privilegi necessari all'accesso alle tabelle di paginazione della memoria, per la scrittura su disco sarà probabilmente necessario solamente l'accesso al sottosistema di I/O. L'uso di un livello di privilegio inferiore garantisce che, anche nel caso di crash della scrittura su disco, gli altri processi resteranno relativamente intatti.

Altre interpretazioni della struttura a strati di protezione pongono l'hardware al livello più interno e il kernel del SO negli strati intermedi.

Ogni strato consiste in una collezione di strutture dati e di routines che possono essere invocate dai livelli superiori (più esterni, meno privilegiati) e che, all'occorrenza, possono invocare codice di livello inferiore (più interno, più privilegiato). Vige il consueto principio di separazione in cui ogni strato non conosce i dettagli di ciò che sta sotto, che vengono nascosti e astratti dal livello immediatamente inferiore.

Vantaggi

In virtù della separazione di competenze, ogni funzionalità — una volta individuato il suo strato di privilegio — non ha bisogno di conoscere dettagli implementativi degli strati inferiori: si semplifica dunque la realizzazione e la manutenzione dei moduli.

La separazione in strati facilita anche le operazioni di verifica della correttezza: si parte dal livello più interno, che poggia direttamente sull'hardware (che dal punto di vista di un programmatore è sempre corretto), e si verificano le sue funzionalità che, essendo a basso livello, rispondono a specifiche semplici. La verifica di una funzionalità a livello più elevato può presumere quindi la correttezza dei livelli più interni, e si procede per induzione.

Svantaggi

Il sistema presenta un overhead potenzialmente maggiore rispetto a un sistema monolitico: per scalare al livello più interno servono più passaggi.

La collocazione di un componente potrebbe non essere semplice: l'accesso al disco potrebbe essere, come in un esempio precedente, a un livello meno privilegiato dell'accesso alla memoria, ma se si desidera operare uno scambio di memoria virtuale l'operazione di scrittura su disco dev'essere invocata dal gestore della memoria principale.

A causa di questi problemi, alcuni sistemi operativi (Linux, Windows) utilizzano a volte il livello più esterno e quell più interno.

4.2.3 Sistemi a microkernel

La maggior parte del codice all'interno di una routine di gestione di un componente, ad esempio il filesystem, non richiede direttamente operazioni privilegiate. Ad esempio, è necessario manipolare strutture dati, e questo può essere fatto in userspace, in obbedienza al PoLA. Solo quando queste strutture sono pronte a essere trasferite all'I/O diventa necessario passare a un livello privilegiato.

Questo porta a una filosofia apparentemente opposta al livello a strati: tutto il codice possibile è piazzato a livello utente. Come si vede in Fig. 4.4, il kernel viene ridotto a una piccola collezione di componenti essenziali (lo scheduling, la gestione della memoria), riducendosi per l'appunto a un microkernel. Una componente importante è quella che coordina i vari componenti a livello utente attraverso l'astrazione del **message passing**. Ogni operazione che richiede il ricorso a moduli di sistema comporta la comunicazione fra moduli attraverso quest'interfaccia.

Il precursore dei sistemi a microkernel è **Mach** (CMU, 1985 circa), e i principi sono utilizzati anche da GNU/Hurd (mai completato perché soppiantato da Linux) e da MINIX (creato da Tanenbaum per il suo corso). Anche il kernel di Mac OS X, Darwin, utilizza parti di Mach.

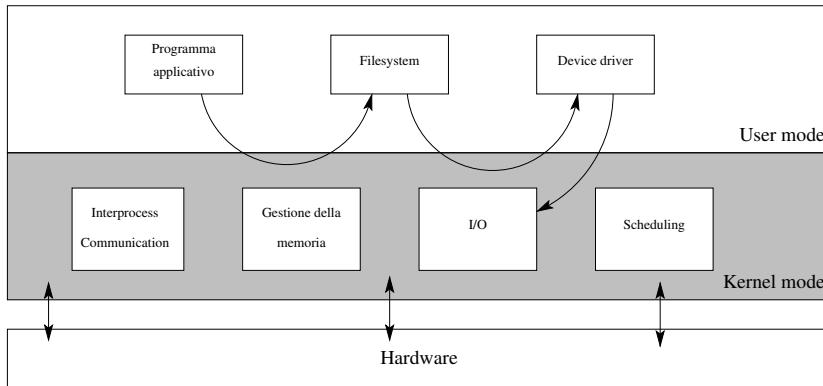


Figura 4.4: Architettura a microkernel.

Vantaggi

I sistemi a microkernel sono molto più protetti da errori di sistema: ad esempio, il gestore del filesystem non opera in modalità privilegiata, quindi può causare meno danni in caso di errore.

La maggiore modularità consente di estendere il sistema operativo più facilmente: la realizzazione di un componente non comporta l'inserimento di funzioni all'interno del codice sorgente del SO e la sua ricompilazione, ma la scrittura di un processo utente che utilizza una particolare interfaccia di comunicazione. L'inserimento e la modifica delle componenti può dunque avvenire anche “a caldo”, durante il funzionamento della macchina.

Svantaggi

L'evidente svantaggio è un overhead ancora maggiore, con un numero potenzialmente elevato di entrate e uscite dalla modalità privilegiata per ogni chiamata di sistema.

4.2.4 Sistemi a moduli

Un modo per conservare alcuni vantaggi dell'architettura a microkernel (estensibilità, inserimento a caldo), rinunciando però alla protezione dello usermode, è quello di implementare le funzionalità in **moduli**, componenti software eseguiti in kernelmode, e che poggiano su un'interfaccia uniforme di chiamate di sistema. Linux utilizza questo sistema principalmente per i device driver e i filesystem, e installa dinamicamente i moduli relativi all'hardware installato sulla macchina.

Anche Windows e Mac OS X sono modulari.

Vantaggi

L'impronta di memoria di un SO modulare è ridotta, perché solamente il codice necessario a gestire l'hardware installato viene caricato in memoria (o può addirittura venire caricato al primo utilizzo).

Svantaggi

Come al solito, overhead dovuto alle varie indirezioni necessarie ad accedere a un modulo non direttamente linkato al kernel.

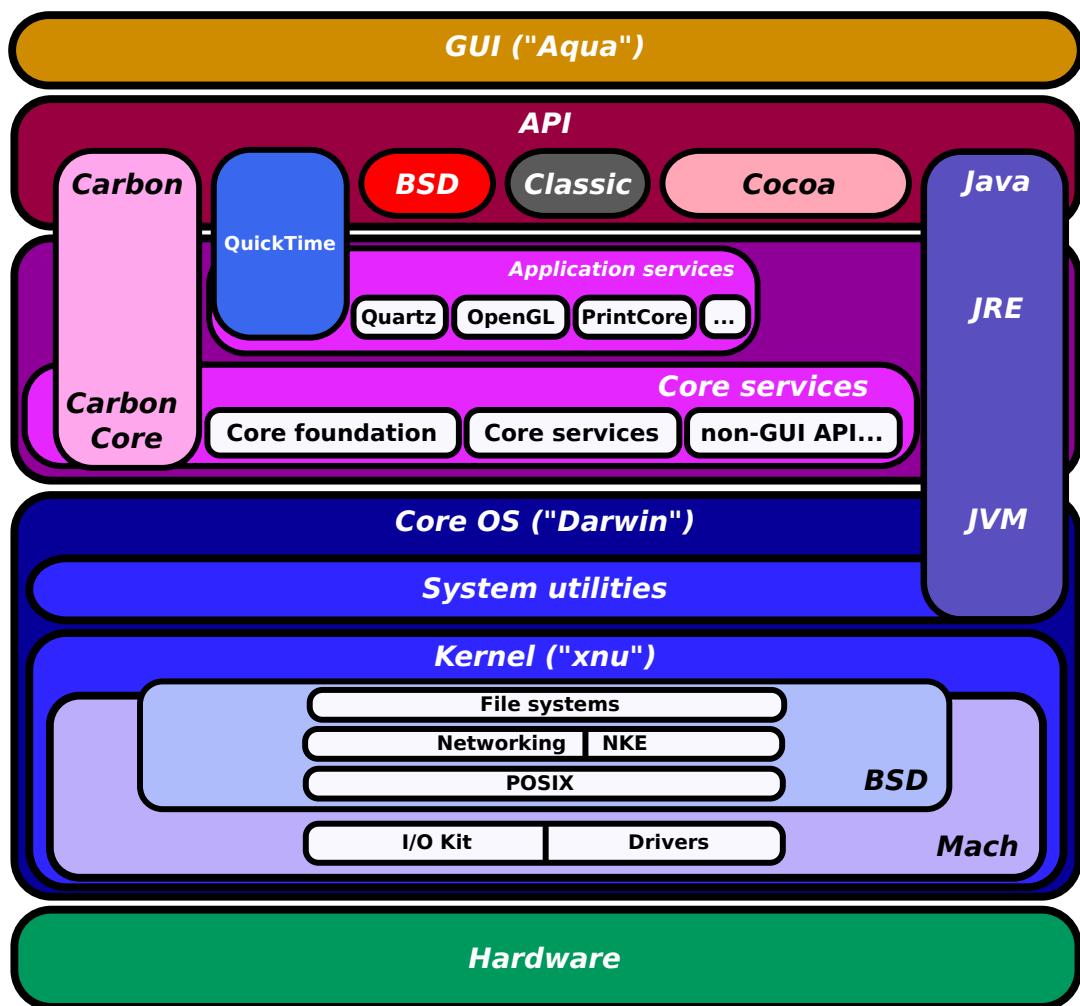


Figura 4.5: Architettura di Mac OS X (da https://en.wikipedia.org/wiki/Architecture_of_macOS).

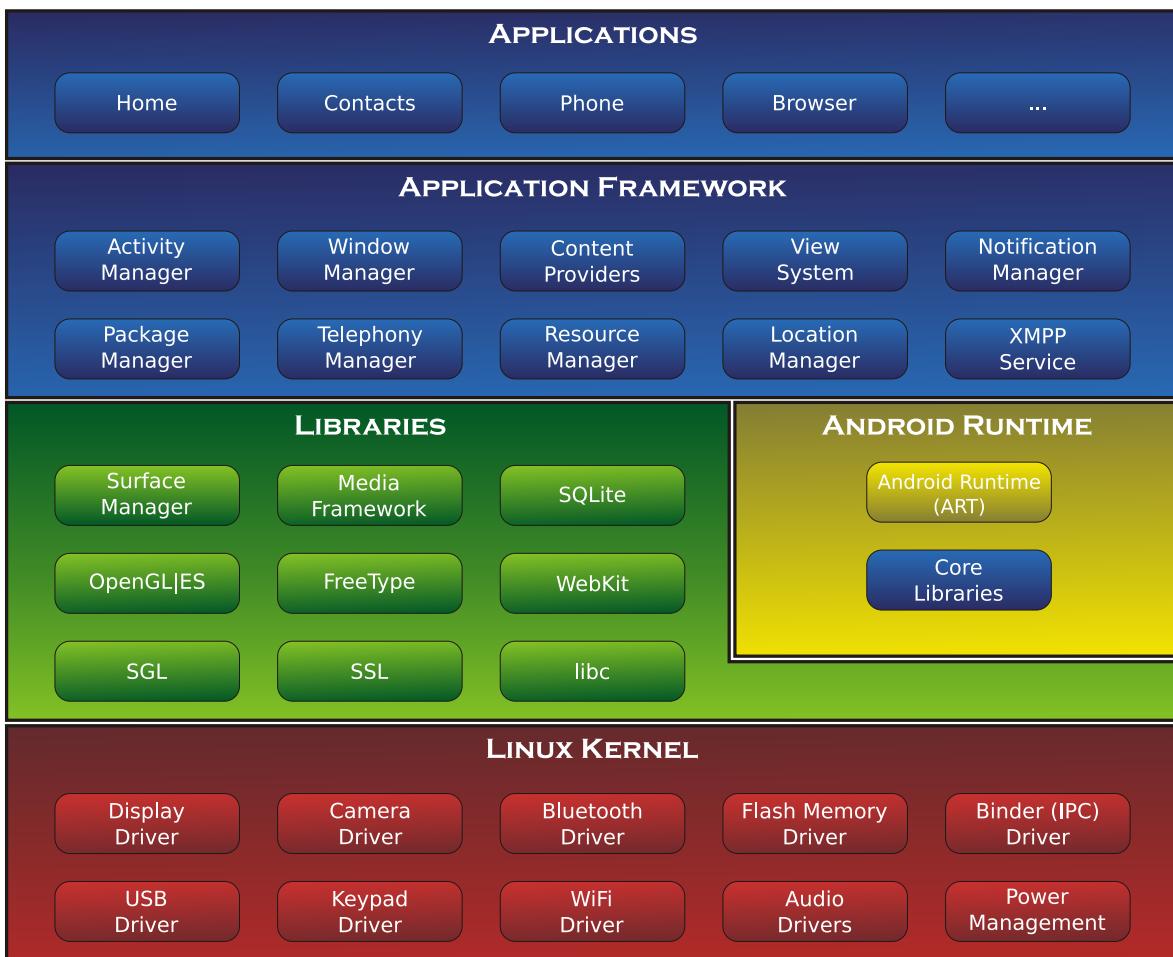


Figura 4.6: Architettura di Google Android (da [https://en.wikipedia.org/wiki/Android_\(operating_system\)](https://en.wikipedia.org/wiki/Android_(operating_system))).

4.3 Esempi

4.3.1 Mac OS X

Vedere Figura 4.5. La GUI (Aqua) poggia su un insieme di API ad alto livello come BSD, Cocoa (interfaccia verso Objective C e Swift), Java, che connettono uno strato di servizi userspace.

Il kernel (Darwin) utilizza moduli BSD, gestiti però attraverso un microkernel Mach che include le parti privilegiate del sistema (I/O, device drivers, scheduling).

4.3.2 Android

Vedere Figura 4.6. Android, anch'esso un sistema stratificato, mira ad offrire un ricco insieme di framework per l'uso di un insieme (ampiamente standardizzato, ma in continua evoluzione) di componenti hardware. Si basa su di un kernel linux per la gestione delle funzionalità di base (processi, memoria, driver dei dispositivi), espanso per la gestione del consumo energetico.

Le applicazioni poggiano su una macchina virtuale Java-like sviluppata da Google.

4.4 Macchine virtuali

Concetto nato negli anni '60: offrire agli utenti di un sistema in timesharing ambienti operativi completamente isolati tra loro. Riscoperto negli ultimi vent'anni per molti usi diversi:

- Nei data center, mettere a disposizione di un gran numero di utenti una macchina completa disponendo di un numero più limitato di server.
- Offrire un ambiente completamente isolato (sandbox) per applicazioni rischiose.
- A livello personale, poter eseguire programmi creati per un ambiente operativo diverso da quello installato, o per testare un prodotto su più ambienti operativi.

4.4.1 Simulazione dell'hardware

Il problema di eseguire più macchine virtuali sulla stessa macchina fisica consiste principalmente nell'evitare tutte le possibili collisioni fra sistemi "convinti" di avere pieno ed esclusivo accesso all'hardware. È necessario un "Virtual Machine Monitor" (VMM, o ipervisore, hypervisor) che, a livello privilegiato, simuli l'hardware messo a disposizione di ciascun SO.

- La CPU può venire simulata, oppure allocata ai diversi OS come fossero processi separati:
 - la simulazione permette di eseguire OS previsti per una CPU diversa da quella fisica: ogni istruzione viene simulata dalla CPU ospitante, a prezzo di una notevole perdita di efficienza;
 - se invece la CPU fisica è compatibile col sistema ospitato, il sistema operativo ospitato può essere eseguito direttamente, anche se a un livello di privilegio inferiore al dovuto; è compito del VMM intercettare tutti i tentativi di esecuzione di istruzioni privilegiate (installando le opportune trap) ed emularne il funzionamento; questo secondo approccio causa dunque una leggera perdita di efficienza, ma la velocità resta paragonabile a quella della macchina fisica.
- La RAM è gestita allocando un grosso blocco di memoria per ciascun OS, il quale poi opera all'interno del blocco in modo esclusivo.
- Il disco è gestito assegnando a ciascun OS una partizione di un disco fisico, oppure un grosso file.
- La rete è gestita attraverso apposite interfacce virtuali (TUN/TAP) trattate dalla macchina fisica con opportune azioni di NAT e firewalling.
- Il display, se si tratta di una macchina desktop, è simulato da una finestra, oppure da una schermata separata in un sistema di display virtuali.
- ...

4.4.2 Tipi di macchine virtuali

Figura 4.7 presenta i due modelli più diffusi di virtualizzazione.

Tipo 1 Il tipo 1 (a sinistra) è un programma a basso livello, che opera a livello privilegiato direttamente sull'hardware (ovviamente poggiando su driver di dispositivo). La macchina è completamente dedicata alla virtualizzazione, che può operare in modo efficiente ripartendo in modo statico tutte le risorse fra gli ospiti. Alcuni esempi sono Oracle VM Server, VMWare ESXi, Xen.

Tipo 2 Il tipo 2 (a destra) è un programma che opera a livello utente simulando l'hardware (a parte la CPU e la RAM, se possibile). Di efficienza leggermente inferiore, perché poggia a sua volta sul sistema operativo della macchina fisica, permette di operare su un sistema virtuale anche su un PC utente. Alcuni esempi: Parallels, QEMU, Oracle VirtualBox, VMWare Workstation.

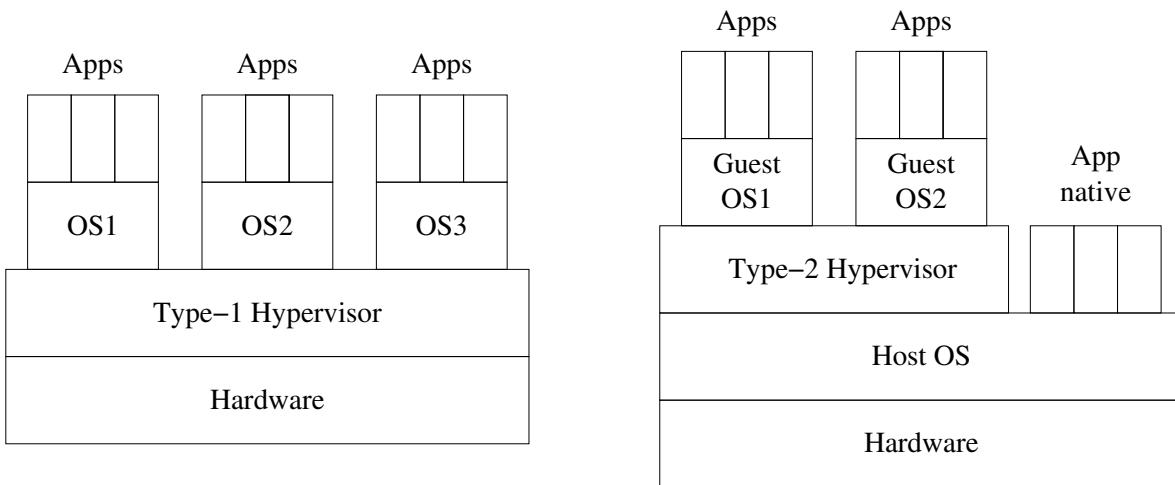


Figura 4.7: Ambienti di virtualizzazione: ipervisori di tipo 1 (a sinistra) e di tipo 2 (a destra).

Ambienti software virtuali

Un altro tipo, meno spinto, di virtualizzazione è quello della macchina virtuale Java (Java Virtual Machine, JVM) o dell'equivalente progetto di Microsoft, il framework .NET (“dot-net”). In questo caso il sistema operativo resta quello fisico, e la macchina simula un ambiente operativo più contenuto offrendo API standardizzate per l'accesso alle risorse ed emulando una CPU per l'esecuzione del processo.

Contenitori

Alcuni sistemi operativi (in particolare tutte le ultime versioni di Linux, Windows Mac OS X) prevedono l'esistenza di istanze userspace separate chiamate “contenitori” (containers) o “zone”, “partizioni”, “celle”¹...

I processi in esecuzione all'interno di un contenitore lo vedono come l'intero ambiente operativo, e possono accedere esclusivamente alle risorse ad esso assegnate.

Alcuni vantaggi di questo approccio:

- Possibilità di eseguire processi in ambienti isolati (sandbox) senza avviare un intero sistema operativo.
- Possibilità di eseguire programmi che richiedono versioni diverse delle librerie di sistema o di alcuni servizi rispetto a quelle installate nel sistema fisico.
- Basso overhead dovuto all'appoggio sul SO reale.
- Mantenimento dell'intero stato di un insieme di programmi in un ambito circoscritto, quindi una maggiore facilità nel salvataggio di immagini per backup e nella migrazione dei servizi tra server.

L'esempio più comune è **Docker**, che usiamo in laboratorio.

¹L'argomento non appare nei libri di testo consigliati; vedere https://en.wikipedia.org/wiki/OS-level_virtualization e le note di laboratorio.

Capitolo 5

Processi e thread

In un sistema in multiprogrammazione è necessario identificare le diverse attività in esecuzione sulla macchina. Nei sistemi batch abbiamo parlato di *job*; nei sistemi attuali l’analoga entità viene detta **processo** (*process*). I due termini sono pressoché interscambiabili (ad esempio, si parla comunemente di *job scheduling*, forse perché è più breve).

5.1 Definizione

Un **processo** è un *programma in esecuzione*. Come si vede in Figura 5.1, un processo è mappato su un’area della memoria principale formata da una sezione di codice (“text”), una sezione contenente i dati statici (“data”), poi da due sezioni dinamiche che contengono la memoria allocabile per le variabili globali (“heap”, tradizionalmente in crescita dal basso) e la pila per gli indirizzi di ritorno delle chiamate di funzione, i parametri e le variabili locali (“stack”, tradizionalmente agli indirizzi più alti della memoria del processo).

Due processi (ad esempio, due istanze di uno stesso browser) possono condividere lo stesso codice, ma consistono comunque di sequenze di esecuzione separate.

Lo heap non è mai condiviso fra processi (con l’eccezione di sezioni di memoria condivisa; inoltre veremo presto i *thread*). Lo stack, ovviamente, non è **mai** condiviso.

La Figura 5.2 riporta un piccolo programma C (in alto) e parte della sua traduzione in assembly (in basso) ottenuta compilando il codice con `gcc` utilizzando le opzioni `-O0` per disattivare le ottimizzazioni e `-S` per generare il codice assembly:

```
gcc -O0 -S esempio.c
```

Osserviamo, in particolare:

- Alcuni valori costanti, come il valore della costante 10.0 assegnata alla variabile globale `p` e la stringa `"%d %f\n"` usata nella funzione `printf`, sono memorizzati nella sezione `.data`; osservare che il linguaggio assembly utilizza solo tipi interi, quindi la combinazione di bit che rappresenta il valore float 10.0 è riportata come se rappresentasse un intero (1092616192).
- L’array `x` da 100 numeri float è memorizzato nello heap, individuato dalla sezione `.bss` (“block starting symbol”) del codice.
- Per trovare la variabile locale `i`, invece, dobbiamo andare a vedere il codice vero e proprio (la sezione `.text` sulla parte destra in figura): lo stack viene ampliato di 16 byte (con una sottrazione `subq` sullo stack pointer `rsp`, perché cresce verso il basso), e la variabile `i` occupa i 4 byte più alti (indirizzo `-4(%rbp)`).
- Il codice principale termina con una `ret`

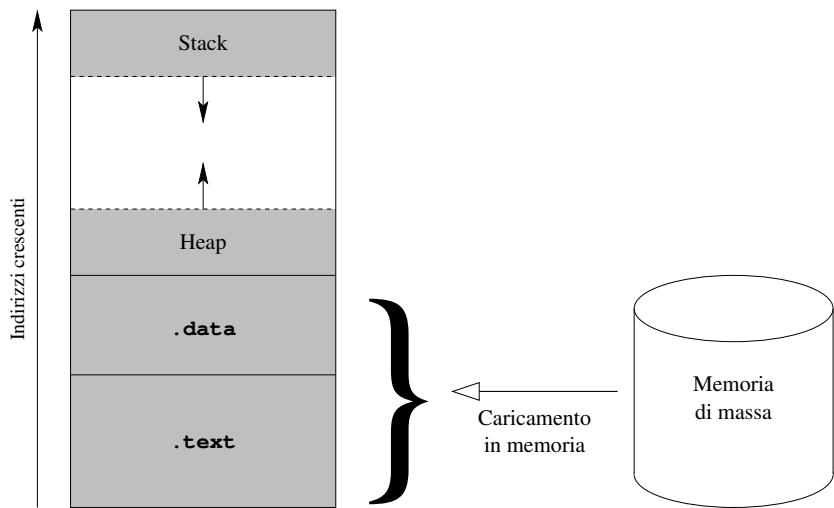


Figura 5.1: Immagine di un processo in memoria (caso semplice).

5.2 Le informazioni di stato di un processo

Un processo può essere in esecuzione, in attesa di input, eccetera. La Figura 5.3 mostra una possibile interpretazione degli stati di un processo e delle loro transizioni. Notare che i nomi non sono standardizzati.

- Created — processo in corso di istanziazione
- Waiting — in attesa di essere preso in carico da una CPU
- Running — in corso di esecuzione
- Blocked — in attesa di un evento esterno (I/O)
- Terminated — completato (o interrotto).

In un sistema in multiprogrammazione, un processo si alterna molto rapidamente fra gli stati Waiting e Running, finendo occasionalmente nello stato Blocked se richiede un input. In Figura 5.4 vediamo tre processi che competono per una singola CPU. In ogni istante, al più un processo è nello stato Running, mentre gli altri sono nello stato Waiting o Blocked.

Il periodo di tempo in cui un processo rimane nello stato Running è detto **quanto**. Un compito del sistema operativo è di suddividere il tempo CPU in quanti e assegnare ciascun quanto a un processo.

5.2.1 Il process control block

Per poter sospendere e riprendere l'esecuzione di un processo a piacimento, la CPU deve poter ricordare l'intero contesto di un processo. Chiamiamo **Process Control Block** (PCB) la struttura che contiene queste informazioni di contesto. Alla sospensione di un processo in esecuzione, la CPU deve salvare nel PCB almeno le seguenti informazioni, che andranno ripristinate all'ingresso successivo nella stato Running:

- I registri della CPU, il program counter, lo stack pointer, i flag, ecc.
- Le informazioni dello scheduler (vedremo più avanti): in quale coda si trova il processo, la priorità, eccetera.

```
#include <stdio.h>
```

```
// Variabili globali -> HEAP
float p = 10.0;
float x[100];

int main() {
    int i; // variabile locale -> STACK
    for (i = 0; i < 100; i++) {
        x[i] = i / p;
        // Stringa di formato -> HEAP
        printf("%d %f\n", i, x[i]);
    }
    return 0;
}
```

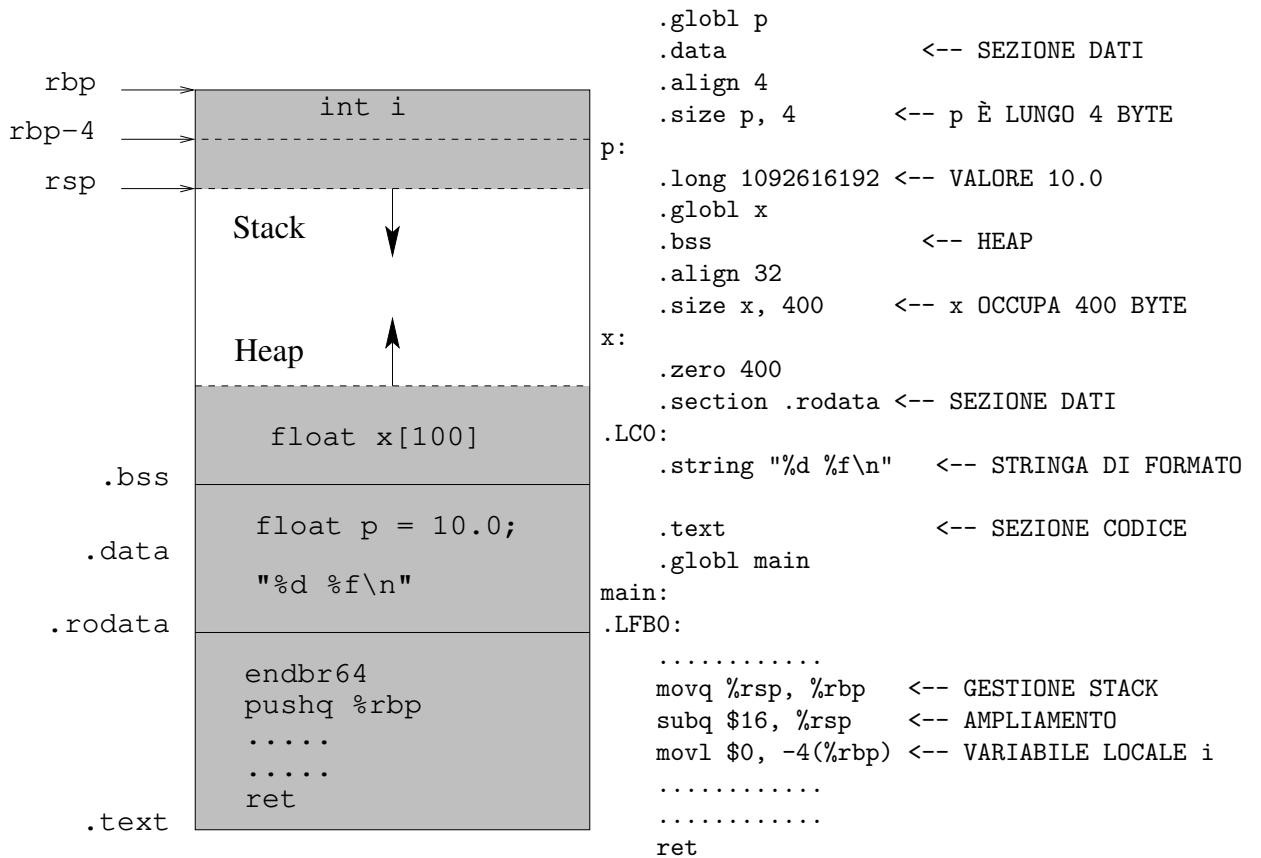


Figura 5.2: Un piccolo programma C (sopra) tradotto in assembly (sotto).

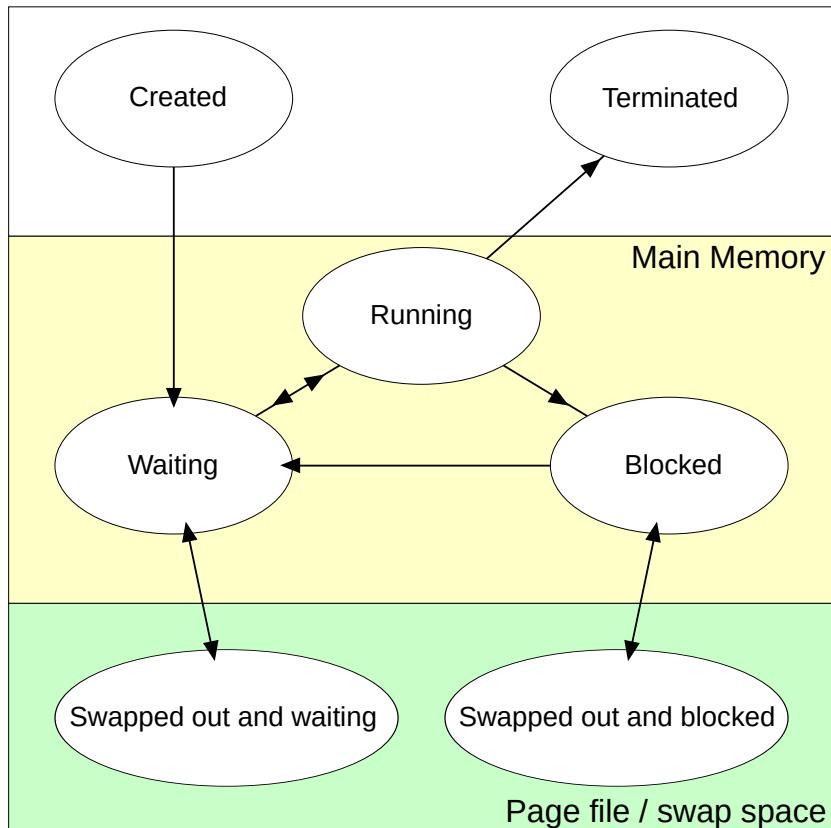


Figura 5.3: Gli stati di un processo (diagramma semplificato, da https://en.wikipedia.org/wiki/Process_state).

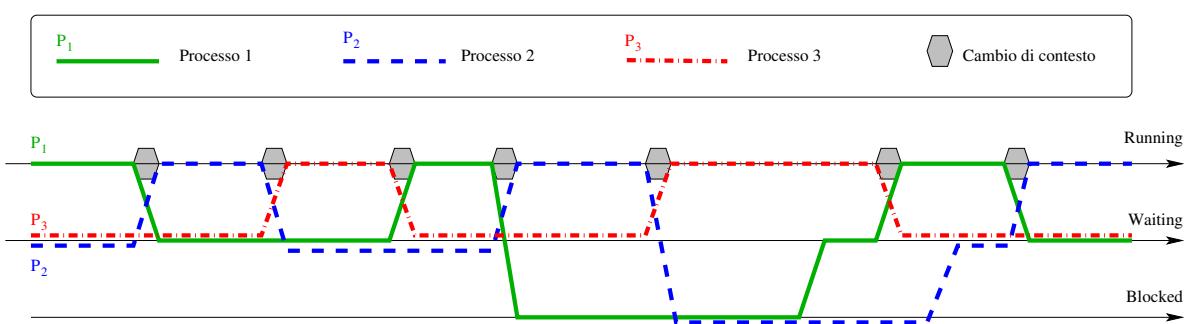


Figura 5.4: Evoluzione degli stati di tre processi concorrenti.

- Informazioni sui blocchi di RAM allocati al processo.
- Informazioni sull'I/O: dispositivi e file allocati al processo.
- Informazioni sui permessi: identificatore dell'utente “proprietario” del processo (utile per stabilire i permessi di accesso), identificatore del processo “genitore”…
- Informazioni statistiche: tempo CPU cumulativo, tempo di orologio…

Vedremo più avanti che Sempre in riferimento alla Figura 5.4, nel momento in cui (per una richiesta di I/O o alla scadenza del suo quanto di tempo) la CPU, che entra in modalità supervisore, deve smettere di eseguire un processo ed eseguire il **dispatch** di un nuovo processo. In dettaglio:

- Un interrupt segnala il termine del quanto di tempo allocato al processo attualmente in stato Running, oppure il processo avvia una richiesta di I/O; in entrambi i casi la CPU passa in modalità privilegiata.
- Parte l'operazione di cambio di contesto (**context switch**):
 - Il contesto del processo in esecuzione viene salvato nel relativo PCB.
 - Il PCB viene accodato alla lista dei processi pertinente.
 - Un altro PCB viene recuperato dalla lista dei processi in stato Waiting.
 - Il contesto (registri CPU, program counter...) del nuovo processo viene ripristinato sulla base delle informazioni contenute nel nuovo PCB.
- La CPU si riporta in modalità utente e continua ad eseguire il nuovo processo.

Queste azioni, che occupano tempo CPU, sono eseguite nelle zone grigie della figura.

5.3 Lo scheduling

Un sistema può arrivare ad avere centinaia, se non migliaia, di processi in esecuzione. Supponendo (per ora) di disporre di una sola CPU, uno solo di questi processi si trova nello stato Running; i contesti di tutti gli altri sono memorizzati nei rispettivi PCB. Ogni volta che si impone un cambio di contesto, il sistema deve decidere quale processo riportare in stato Running.

5.3.1 Le code di processi

Per fare questo, i PCB sono normalmente organizzati in code. Un esempio è in Figura 5.5. Il processo collegato al PCB₄, correntemente in stato Running, sta terminando il suo periodo. Lo scheduler lo rimette in stato Waiting e sposta il PCB₄ nella coda dei processi in stato Waiting. Preleva uno dei PCB attualmente in coda, PCB₇, e lo utilizza per ripristinare il contesto del relativo processo.

Inoltre, un processo attualmente bloccato in attesa di input dal disco `/dev/sda1` può essere sbloccato in quanto i dati dtiprogrammazione del sistema.i cui era in attesa sono tornati disponibili, e il PCB₁₂ viene riportato alla coda di Waiting.

Naturalmente, i PCB non vengono spostati *fisicamente* fra le code, ma si utilizzerà una struttura a liste concatenate.

Dal punto di vista del singolo processo, la sua “storia” dalla creazione alla rimozione può essere rappresentata dal diagramma di Fig. 5.6. In figura possiamo vedere altre ragioni per l'accodamento di un processo, ad esempio la sua attesa che termini un processo figlio (si vedrà in laboratorio), oppure l'attesa di un interrupt che segnali un evento esterno o di orologio.

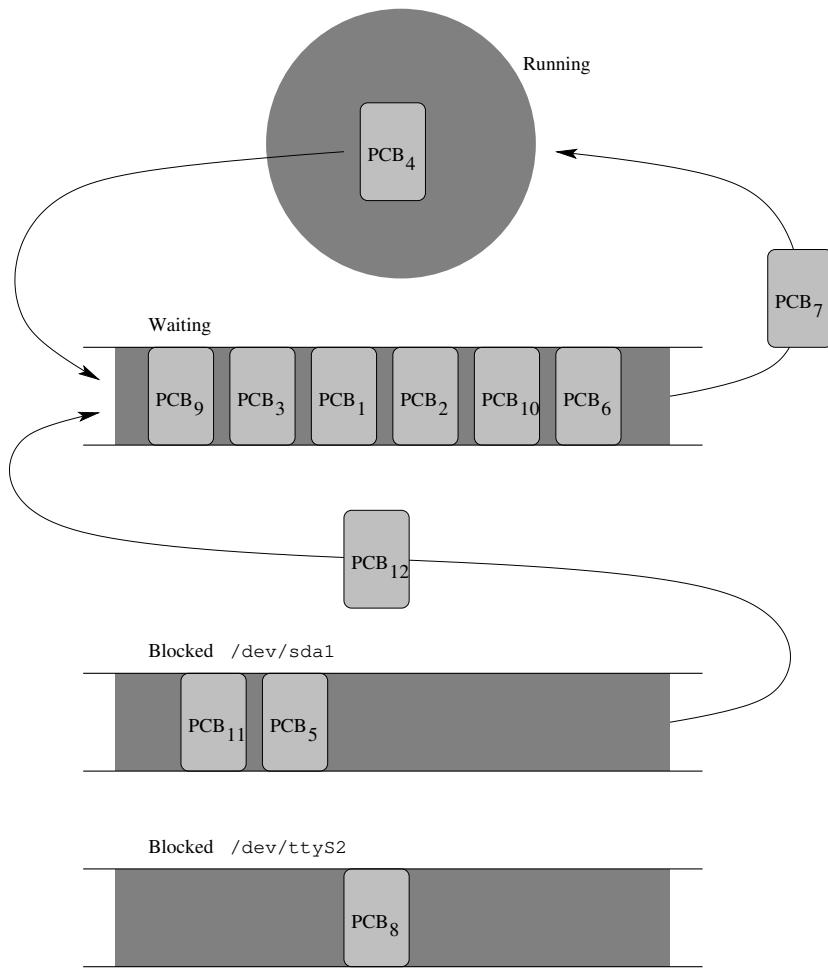


Figura 5.5: Gestione delle code dei processi.

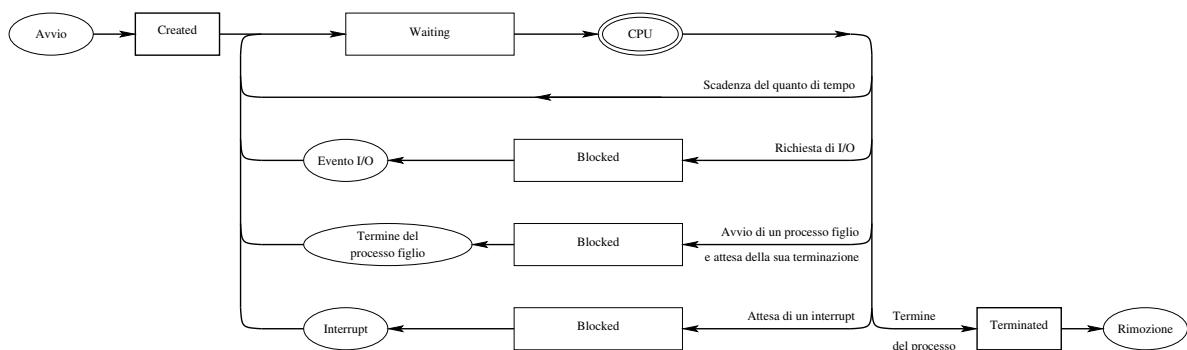


Figura 5.6: Il ciclo di vita di un processo attraverso lo spostamento fra le varie code.

5.3.2 Lo scheduler

Le code dei processi non sono strutture FIFO rigide. Infatti, la scelta di quale processo eseguire in un determinato istante può dipendere da vari fattori, inclusa la sua criticità, il tempo già trascorso in coda, il possesso di risorse che tengono occupati altri processi, e così via.

La selezione dei processi da eseguire è operata da un componente del SO detto **scheduler**. I suoi obiettivi sono:

- massimizzare l'uso della CPU evitando tempi morti;
- passare rapidamente da un processo all'altro in modo che tutti si vedano eseguiti senza soluzione di continuità;
- privilegiare processi critici per il sistema;
- ...

Distinguiamo due tipi di scheduler:

- A breve termine: sceglie, tra i processi in stato Waiting, il prossimo da portare in stato Running. Invocato di frequente (millisecondi), dev'essere veloce e non consumare molta CPU.
- A lungo termine: sceglie quali processi avviare e portare in memoria principale. Invocato meno di frequente, determina il grado di multiprogrammazione del sistema (numero di processi in memoria).

Osserviamo che nei SO più diffusi, dove il grado di multiprogrammazione è molto elevato, lo scheduler a lungo termine è (praticamente) assente: tutti i processi sono caricati in memoria (eventualmente virtuale).

Uno scheduler a lungo termine può basare la sua decisione su quali processi avviare sulla base del loro utilizzo di CPU. In particolare, distinguiamo due tipi "estremi" di processo:

- processi che utilizzano molto I/O ("I/O-bound"),
- processi che effettuano invece molti calcoli ("CPU-bound").

Lo scheduler a lungo termine dovrà in generale bilanciare i due tipi di processi (molti processi I/O-bound lasciano la CPU inutilizzata, molti processi CPU-bound lasciano inutilizzati i dispositivi e richiedono spesso l'intervento dello scheduler a breve termine e molti cambi di contesto).

Talora, la componente che si occupa di selezionare il processo da spostare in memoria virtuale è detta scheduler *a medio termine*.

5.4 Operazioni sui processi

5.4.1 Creazione

A esclusione del processo iniziale (`init` in molti sistemi Unix), un processo viene creato da un processo "genitore" (parent process). La Figura 5.7 illustra come esempio la catena che va dal processo iniziale (`systemd`, il cui ID è 1 e che non ha genitore) fino al processo `ps` usato per stampare la lista dei processi nel terminale.

Esistono tre primitive (chiamate di sistema) principali per la gestione dei processi.

Duplicazione di un processo (fork) Quando un processo deve avviare un altro, invoca la primitiva `fork()` che ne duplica l'immagine in memoria e il PCB, creando un processo figlio identico al padre. A questo punto, i due processi sono in esecuzione concorrente. Il figlio viene posto nella coda Waiting mentre il padre prosegue per il suo quanto di tempo CPU.

F	S	UID	PID	PPID	C	PRI	NI	ADDR	SZ	WCHAN	TTY	TIME	CMD
4	S	0	1	0	0	80	0	-	42447	-	?	00:00:02	systemd
...													
4	S	1000	2721	1	0	80	0	-	5169	ep_pol	?	00:00:01	systemd
...													
0	R	1000	6897	2721	0	80	0	-	205493	do_pol	?	00:00:08	gnome-terminal-
0	S	1000	6905	6897	0	80	0	-	3302	do_wai	pts/0	00:00:00	bash
...													
4	R	1000	7809	6905	0	80	0	-	3068	-	pts/0	00:00:00	ps

Figura 5.7: Estratto dall'esecuzione del comando `ps -el`: la catena dal processo iniziale (PID = 1) al processo `ps` (PID = 7809). L'ID del processo è riportato alla colonna PID, quello del genitore è indicato alla colonna PPID.



Figura 5.8: Il ciclo di base di una shell dei comandi: esempio di creazione di processo ed esecuzione sincrona in cui la shell crea e attende l'esecuzione di un processo figlio che carica un comando richiesto dall'utente.

Sostituzione del processo (exec) A questo punto il figlio ha di fronte due possibilità: proseguire con lo stesso codice del padre, o sostituire la propria immagine con quella di un altro processo. Quest'ultima alternativa è gestita dalla primitiva `exec()` che “carica” in memoria un nuovo processo al posto di quello corrente. Il processo genitore, ovviamente, non è toccato da questa sostituzione.

Attesa del completamento del processo figlio (wait) Il processo genitore, invece, può proseguire nell'esecuzione concorrente, oppure può attendere che il figlio termini invocando la primitiva `wait()`. Ad esempio la shell, dopo aver invocato un comando, attende la sua terminazione prima di ripresentare il prompt.

Il ciclo di attesa del comando ed esecuzione, tipico di una shell, è esemplificato dallo pseudocodice di Figura 5.8 e dimostra l'uso delle tre primitive. Dopo aver atteso un comando dell'utente, il processo shell si sdoppia invocando la `fork()`. A questo punto, sia il processo shell che il suo figlio proseguono con l'istruzione successiva.

Per “rompere la simmetria” (è importante che i due processi facciano cose diverse, ovviamente), si guarda al valore restituito dalla chiamata alla `fork()`. Il padre riceve il PID del figlio, mentre il figlio riceve zero.

Il processo genitore invoca la `wait()` sul figlio, mettendosi volontariamente in stato Blocked.

Il processo figlio rimpiazza la propria immagine in memoria (attualmente uguale al padre) con quella del comando da eseguire. Così facendo, il codice viene interamente sostituito da quello relativo al comando, quindi il figlio non si trova più nel loop infinito del genitore.

Copy-on-write Osserviamo che l'immagine del processo in memoria può occupare molto spazio, e che spesso la sua copia durante la `fork` è pressoché inutile, visto che l'immagine potrebbe essere

immediatamente sostituita da un'altra tramite chiamata `exec`. La soluzione è la tecnica *copy-on-write*: immediatamente dopo la `fork`, le pagine di memoria sono fisicamente le stesse per i due processi (genitore e figlio). La MMU punta i due spazi di indirizzamento verso le stesse pagine fisiche. Le operazioni di scrittura in memoria vengono però intercettate e causano la duplicazione della pagina fisica.

In Figura 5.8, ad esempio, dopo la `fork` il processo padre non scrive nulla, quindi non causa nessuna copia fisica della memoria. Solo la `exec` del figlio causa la creazione di una nuova pagina di memoria (ovviamente necessaria, se si vuole lanciare un nuovo eseguibile).

5.4.2 Terminazione di un processo

Un processo può terminare per varie ragioni:

- Il processo termina la propria esecuzione (invocando la primitiva `exit()` o equivalente).
- Il processo è terminato dal padre
 - se il padre termina il SO potrebbe non permettere ai figli di continuare.
- Il processo è terminato da un altro processo con sufficienti privilegi
 - invocazione della primitiva `kill()`
 - comando `kill` invocato dalla shell.
- Il processo è terminato dal SO
 - eccessivo uso di risorse
 - errori di protezione (istruzioni privilegiate)
 - accesso ad aree di memoria non consentite...

5.5 Il concetto di thread

Finora abbiamo identificato nello stesso oggetto, il processo, due aspetti distinti:

- Il possesso delle risorse (memoria, file)
- L'uso della CPU (esecuzione), contraddistinto dallo stato, dai valori dei registri, ecc.

Le due caratteristiche sono però indipendenti: possiamo immaginare più unità di esecuzione che condividono il codice, la memoria e le altre risorse, ma che operano in modo concorrente fra loro. Distinguiamo dunque:

- i processi sono le unità di possesso delle risorse. In particolare, un processo è associato a
 - uno spazio di indirizzamento,
 - un elenco di risorse.
- i **thread** sono le unità di utilizzo della CPU: uno stesso processo può essere associato a più thread. In particolare, un thread è associato a
 - uno stato di esecuzione,
 - un insieme di registri CPU (in particolare un program counter),
 - uno stack,

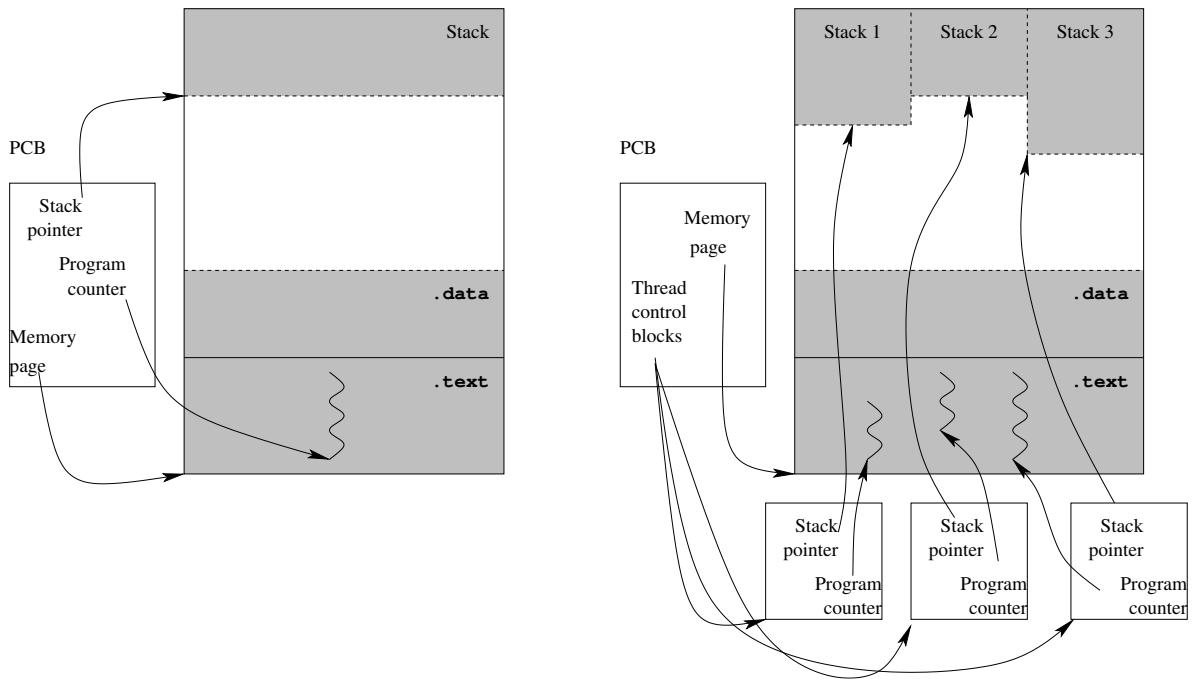


Figura 5.9: Un processo a thread singolo (sinistra) e a thread multiplo (destra).

Questa distinzione è ormai presente in tutti i sistemi operativi moderni, ed è fondamentale per l'esecuzione efficiente di molti servizi ad elevato parallelismo come, ad esempio, i server web che devono gestire un numero elevato di richieste concorrenti, ma che non possono permettersi l'overhead necessario a generare un nuovo processo per ogni richiesta.

In Figura 5.9 vediamo un esempio di esecuzione multithreaded. In uno stesso processo (identificato dal PCB) possiamo avere più flussi di esecuzione concorrenti, ciascuno con i propri registri e il proprio stack, che condividono la stessa pagina di memoria. Il contesto specifico di ciascun thread è conservato in un “thread control block”.

5.5.1 Vantaggi

Velocità di risposta Se un processo gestisce più richieste in sequenza, una richiesta particolarmente laboriosa tiene bloccato il programma. Una soluzione che multithreaded permette a ogni richiesta di essere soddisfatta indipendentemente dallo stato delle altre.

Condivisione delle risorse I thread di uno stesso processo vedono lo stesso spazio di indirizzamento e condividono la memoria. Con le opportune cautele, dunque, è possibile condividere informazioni e scambiare messaggi semplicemente scrivendo valori in variabili globali.

Economicità Creare nuovi processi è costoso perché richiede la duplicazione delle risorse, memoria in primis. Viceversa, la creazione di un thread è più snella. Ad esempio, in Solaris (un dialetto Unix) la creazione di un thread è trenta volte più veloce della creazione di un processo.

Anche il cambio di contesto fra due thread dello stesso processo è più rapido (di circa cinque volte) perché non richiede la ripaginazione della MMU.

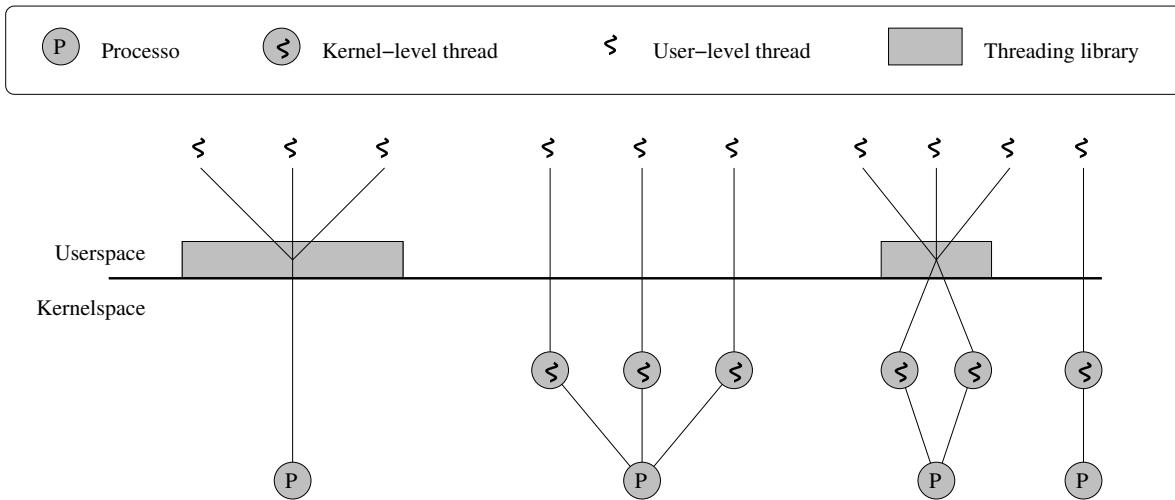


Figura 5.10: Diversi modelli di threading: user-level (sinistra), kernel-level (centro), a due livelli (destra).

Programmazione multiprocessore Grazie al multithreading, uno stesso processo può utilizzare più di un processore semplicemente delegando thread diversi a processori separati¹.

5.5.2 Tipi di thread

Il supporto al multithreading può essere fornito dal kernel, che può dedicare le proprie capacità di scheduling e di context switching. In questo caso si parla di **kernel-level threading**, normalmente gestito dallo scheduler.

In alternativa, il multithreading può essere supportato da apposite librerie a livello utente (**user-level threading**). In questo caso il kernel vede solo un processo monolitico, e la gestione dei thread è delegata all'applicazione. Il vantaggio di questa soluzione, oltre all'applicabilità a SO che non supportano nativamente la funzionalità, è la possibilità di passare da un thread all'altro senza l'overhead dello scheduler; per contro, l'applicazione utente dev'essere costruita appositamente, il blocco di un thread può tener fermo l'intero processo e non è possibile mandare in esecuzione thread di uno stesso processo in parallelo su più processori.

In Figura 5.10, vediamo a destra il caso del threading a livello utente: il kernel gestisce un processo, il resto è demandato a una libreria in userspace. Al centro, i thread del programma utente sono gestiti da thread a livello kernel. A destra, un modello misto in cui un pool (talora piccolo) di thread a livello kernel viene usato per gestire un numero potenzialmente elevato di thread a livello utente. In questo caso, un thread utente può optare per essere associato a un thread kernel esclusivo (se ne ha i privilegi).

5.6 Comunicazioni fra processi

Per quanto ogni processo costituisca un'unità separata, può essere necessario far collaborare più processi fra loro, per almeno due motivi:

- Una stessa applicazione può essere suddivisa in più processi che devono coordinarsi per varie ragioni:
 - Modularità — si desidera che ciascuna funzionalità sia gestita in modo separato dalle altre;

¹Un processo utente non può scegliere esplicitamente quali core allocare, ma il SO cerca di tenere bilanciato il carico operando una scelta (quasi) ottimale.

- Parallelizzazione — in un sistema multiprocessore si vuole velocizzare un calcolo facendolo eseguire in parallelo;
- Protezione — in un'applicazione molto complessa si desidera che il fallimento di un componente non causi la terminazione di tutto il programma;
- Più applicazioni possono dover condividere una stessa informazione (ad esempio contenuta in un file).

La cooperazione richiede comunicazione, e un sistema operativo mette a disposizione molte forme di comunicazione fra processi (**interprocess communications**, IPC) che si possono suddividere in due categorie principali: condivisione di memoria (shared memory) e passaggio di messaggi (message passing).

Due osservazioni:

- Quelli che stiamo per descrivere sono solo meccanismi per scambiare informazioni, e non risolvono tutti i possibili problemi di sincronizzazione fra processi cooperanti (o in competizione). Più avanti studieremo anche alcuni metodi di sincronizzazione.
- La comunicazione fra thread dello stesso processo è molto più semplice in quanto questi condividono le risorse, quindi è sufficiente che il processo dichiari strutture dati globali.

5.6.1 Memoria condivisa

La condivisione della memoria è utile per mettere in comune grandi quantità di dati e strutture dati complesse.

Una regione di memoria condivisa risiede nello spazio di indirizzamento del processo che ne richiede la creazione; gli altri processi che ne vogliono fare uso devono disporre di un meccanismo per inserirla anche nel proprio spazio, aggirando in qualche modo i meccanismi di protezione (ad esempio, agendo sulla MMU).

Fatto questo, i processi possono scambiare informazioni scrivendo e leggendo la memoria.

Problema: nondeterminismo

I processi che condividono memoria devono porre particolare attenzione quando effettuano accessi in scrittura: se due processi devono scrivere nello stesso momento, in generale non è possibile prevedere quale processo prevarrà; se la scrittura riguarda una parola di più byte, potrebbe avvenire a cavallo di un cambio di contesto, oppure su processori diversi, e il risultato finale potrebbe addirittura essere un miscuglio delle due parole.

Il pattern produttore-consamatore

Per esemplificare l'uso della memoria condivisa, consideriamo due processi: uno produce informazioni e le deve mettere a disposizione dell'altro, che le consuma.

Un'implementazione semplice, mostrata in Fig. 5.11, potrebbe consistere in una cella di memoria atta a contenere l'informazione prodotta, con un flag booleano che viene messo a **vero** dal processo produttore quando l'informazione è stata scritta, e viene rimesso a **falso** quando l'informazione è stata usata dal processo consumatore.

In Figura 5.12 vediamo un'implementazione più complessa in cui i due processi condividono un buffer nel quale possono essere raccolti più record di informazione in attesa di essere consumati. Supponiamo che le informazioni consistano di record di dimensione fissa, contenuti nelle celle dell'array **buffer**; il sistema è realizzato come una coda FIFO circolare, condividendo i seguenti elementi:

- la costante **DIMENSIONE_BUFFER** che indica la dimensione del vettore che implementa la coda circolare;

(a) Produttore

```

1. condividi
2.   [ record dato
3.     booleano valido
4.   valido ← falso
5.   ripeti
6.     Produci l'informazione
7.     finché valido
8.       Informazione non ancora prelevata, aspetta
9.       dato ← informazione
10.      valido ← vero

```

(b) Consumatore

```

1.   ripeti
2.     finché non valido
3.       Informazione non ancora disponibile, aspetta
4.       informazione ← dato
5.       valido ← falso
6.       Utilizza l'informazione

```

Figura 5.11: Processo produttore e processo consumatore con un record condiviso.

(a) Produttore

```

1. condividi
2.   [ record buffer[DIMENSIONE_BUFFER]
3.     interi in, out
4.   in ← 0; out ← 0
5.   ripeti
6.     Produci l'informazione
7.     finché (in+1) % DIMENSIONE_BUFFER = out
8.       Buffer pieno, non fare nulla
9.       buffer[in] ← informazione
10.      in ← (in+1) % DIMENSIONE_BUFFER

```

(b) Consumatore

```

1.   ripeti
2.     finché in = out
3.       Buffer vuoto, non fare nulla
4.       informazione ← buffer[out]
5.       out ← (out+1) % DIMENSIONE_BUFFER
6.       Utilizza l'informazione

```

Figura 5.12: Processo produttore e processo consumatore con un buffer circolare in memoria condivisa.

- il vettore `buffer` le cui celle conterranno le informazioni;
- l'indice `in` che rappresenta la prossima cella in cui scrivere l'informazione prodotta;
- l'indice `out` che rappresenta la posizione dell'informazione più vecchia ancora da estrarre.

Dovrebbe essere facile convincersi che il sistema funziona, finché vi sono solo un produttore e un consumatore. Se però più consumatori concorrono per gli stessi dati, le cose si complicano e servono nuove forme di sincronizzazione.

5.6.2 Passaggio di messaggi

Un sistema a passaggio di messaggi (message-passing system) consiste in un canale, messo in genere a disposizione dal sistema operativo, e da alcune primitive che permettono di operarvi. Le due primitive essenziali sono:

- `send()` — invia un messaggio attraverso il canale;
- `receive()` — ricevi un messaggio dal canale.

A differenza della memoria condivisa, in questo caso il sistema operativo assume il compito di sincronizzare i processi che intendono comunicare. Le implementazioni possono differire per molti particolari:

- il modo di identificare il canale;
- comunicazione sincrona o asincrona;
- buffering automatico o esplicito;
- messaggi di dimensione fissa o variabile;
- comunicazione uni- o bidirezionale.

Identificazione del canale

Esistono varie alternative per identificare un canale fra due processi, P e Q , che intendono comunicare tra loro.

ID dei processi (comunicazione diretta) — le due primitive possono richiedere i PID dei due processi: se P intende inviare un messaggio a Q , invoca la funzione `send(Q, messaggio)`. Il processo Q che attende l'informazione invoca `receive(P, buffer)`, dove `buffer` individua un'area di memoria nella quale la primitiva deve scrivere il messaggio ricevuto.

In questo caso, un canale è identificato dalla coppia di ID (P, Q) dei due processi. Il sistema può essere completamente simmetrico; una variante asimmetrica può prevedere la forma `receive(id, buffer)` dove `id` è una variabile: la primitiva resta in attesa di un messaggio da qualunque processo, e ne piazza l'ID nella variabile.

Può esistere un solo canale di comunicazione per ogni coppia di processi.

ID del canale (comunicazione indiretta) — un canale viene creato con un ID univoco k ; due processi che vogliono comunicare devono concordare sul canale da usare, il cui ID viene passato sia alla `send()` che alla `receive()`.

L'identificativo k può essere un numero, un nome, una tupla di informazioni. Il canale può essere visto come una casella postale (mailbox) nella quale i processi depositano e prelevano messaggi.

Una stessa coppia di processi può usare più canali. Uno stesso canale può essere condiviso fra più di due processi, a patto che vi siano delle regole sul prelievo dei messaggi (replicazione verso tutti gli ascoltatori, prelievo da parte di uno solo)

Sincronizzazione

Comunicazione sincrona — Un canale può richiedere che, se un messaggio viene inviato da un processo P , vi sia almeno un processo Q pronto a riceverlo. In caso contrario, la `send()` può fallire, oppure può tenere bloccato P finché Q non esegue la corrispondente `receive()`.

Il vantaggio è che P può sapere con certezza quando il suo messaggio è stato ricevuto.

Allo stesso modo, se Q si mette in ascolto quando P non ha nulla da inviare può rimanere bloccato in attesa, oppure la `receive()` può fallire.

Comunicazione asincrona — Il sistema mette a disposizione un buffer, normalmente FIFO, in grado di contenere un certo numero di messaggi in transito. L'invio e la ricezione nono bloccano i processi (se non in casi estremi).

Il mittente non può sapere (immediatamente) se il messaggio è stato ricevuto; il destinatario preleva il primo messaggio disponibile.

Buffering

La dimensione del buffer per le comunicazioni asincrone può essere un parametro delicato (un buffer consuma memoria).

Oltre al buffer “di transito”, istanziato dal sistema, il ricevente deve predisporre un proprio buffer nel quale la `receive()` piazzerà il messaggio. Normalmente, il ricevente deve comunicare la dimensione massima del proprio buffer di ricezione per evitare che un messaggio troppo grande causi un trabocco o un errore di protezione.

Dimensione dei messaggi

La dimensione dei messaggi può essere definita a priori, nel qual caso un buffer di transito può essere realizzato con un array, oppure variabile, il che richiede qualche cautela in più nella gestione della coda.

Direzionalità

Un canale bidirezionale permette a entrambi i processi di eseguire sia `send()` che `receive()`. Se un canale è unidirezionale, nulla vieta di creare due canali nelle due direzioni.

5.6.3 Esempi di sistemi a passaggio di messaggi

Mentre la gestione della memoria condivisa ha praticamente una sola implementazione, Unix (in particolare lo standard POSIX) prevede varie forme di IPC basate sul passaggio di messaggi.

Pipes (condutture)

- Code FIFO che veicolano un flusso continuo di byte;
- sincrone (scrittura e lettura bloccanti): due processi devono essere rispettivamente in scrittura e all'ascolto per poter procedere;
- unidirezionali: un processo scrive, l'altro legge;
- possono essere create solo fra genitore e figlio.

Pipes con nome (named pipes)

Come sopra, ma:

- istanziabili fra processi arbitrari;
- individuate da un nodo del filesystem (non hanno bisogno di spazio nel filesystem per funzionare, ma i loro nomi risiedono lì);
- bidirezionali.

Code di messaggi (message queues)

- I messaggi sono blocchi di dati (record, struct) accompagnati dalla loro dimensione;
- i messaggi hanno una priorità: il primo ad essere estratto da una coda è il messaggio più vecchio fra quelli che hanno la priorità più elevata.
- un processo può informarsi sullo stato della coda: quanti messaggi contiene, la dimensione massima...

Socket

Molto versatili:

- comunicazione fra processi sulla stessa macchina o fra macchine in rete;
- orientati al flusso di byte o allo scambio di messaggi;
- bidirezionali;
- asincroni.

Capitolo 6

Scheduling

Segue un riassunto di quanto già detto in precedenza:

- In un sistema uniprocessore, solo un processo può essere in esecuzione in un determinato momento. L'obiettivo della multiprogrammazione è di avere un processo in esecuzione in *ogni* momento (massimizzare l'utilizzo della CPU). A questo scopo, più processi risiedono in memoria.
- Ogni processo, nel corso della sua vita, attraversa un'alternanza di periodi in cui deve usare la CPU (**CPU bursts**) e periodi in cui è in attesa di eventi esterni (**I/O bursts**). Se necessario, possiamo distinguere fra processi in cui prevalgono i burst CPU (detti **CPU-bound**) e quelli in cui prevalgono le attese di I/O (detti **I/O-bound**).
- Lo scheduler a breve termine del SO può approfittare dei burst di I/O del processo in esecuzione per sospornerlo e passare a un altro processo, ma può anche (in molti sistemi) interrompere l'esecuzione del processo durante un burst CPU. Per gestire questi passaggi, lo scheduler conserva i process control block in un sistema di code (solitamente a priorità).

6.1 Lo scheduling a prelazione (preemptive)

Lo scheduler entra in gioco nelle quattro seguenti circostanze (già riassunte nelle figure 5.5 e 5.6):

1. un processo in stato Running passa allo stato Blocked (per I/O, oppure per attendere il completamento di un figlio);
2. un processo passa dallo stato Running allo stato Waiting (perché un interrupt ha segnalato la scadenza del suo quanto di tempo);
3. un processo passa dallo stato Blocked allo stato Waiting (completamento di un'operazione di I/O, oppure terminazione del figlio di cui era in attesa);
4. un processo passa dallo stato Running allo stato Terminated.

Nelle circostanze 1 e 4, il corso di azione è chiaro: lo scheduler deve scegliere un altro processo. Se queste due circostanze sono le sole a poter causare un cambio di processo, allora lo scheduler è detto **cooperativo**: in un certo senso, è il processo a decidere quando è il momento di cedere la CPU, ma finché dura il burst non verrà mai interrotto.

Se lo scheduler è in grado di gestire i casi 2 e 3, è detto **a prelazione** (preemptive).

Vantaggi

- In un sistema a prelazione, un processo non può portare al blocco del sistema continuando a usare la CPU senza pause.
- Inoltre, la CPU può essere divisa più equamente fra i processi, indipendentemente dalla durata dei loro CPU burst.

Svantaggi

- Un sistema a prelazione richiede hardware dedicato (un timer che genera interrupt periodici).
- Se due processi condividono memoria, cooperando a mantenere una struttura dati, possono verificarsi delle cosiddette “race conditions”: un processo potrebbe essere interrotto durante una serie di operazioni di scrittura, lasciando la struttura dati in uno stato inconsistente, e un altro processo che tentasse di operare sulla stessa struttura probabilmente fallirebbe.
- Allo stesso modo, la scadenza del quanto potrebbe avvenire durante l'esecuzione di una chiamata di sistema in cui il kernel stesso sta modificando una struttura dati. In tal caso, la prelazione va rimandata fino al completamento della “regione critica” (vedremo più avanti questo termine) nella quale, in casi estremi, è possibile disabilitare gli interrupt. Il rinvio della prelazione, tuttavia, può costituire un problema per i sistemi real-time.

6.1.1 Criteri per lo scheduling

Alcuni dei criteri che possono guidare le decisioni di uno scheduler:

- massimizzare il tempo di utilizzo della CPU;
- massimizzare il *throughput*, il numero di processi completati per unità di tempo;
- minimizzare il *turnaround*, tempo dalla richiesta di avvio al completamento di un processo;
- minimizzare il tempo di attesa passato in stato Waiting (ricordiamo che lo scheduler non ha il controllo sul tempo di blocco);
- minimizzare il tempo di risposta, ovvero il tempo dalla richiesta di avvio al primo feedback ricevuto dall'utente.

Questi obiettivi possono essere in conflitto. Inoltre, trattandosi di tempi che variano da processo a processo, per ciascuno di questi obiettivi è possibile scegliere se ottimizzare:

- il valore medio: scelta ragionevole per tutti gli obiettivi;
- la varianza: ad esempio, per il tempo di risposta;
- i valori estremi (massimo e minimo): può essere importante evitare outliers statistici.

6.2 Algoritmi di scheduling

6.2.1 First Come, First Served (FCFS)

- Caso non-preemptive (senza prelazione).
- Coda Waiting di tipo FIFO: ogni processo viene messo in coda all'avvio, oppure dopo il termine di un I/O burst, viene prelevato il processo in coda da più tempo.

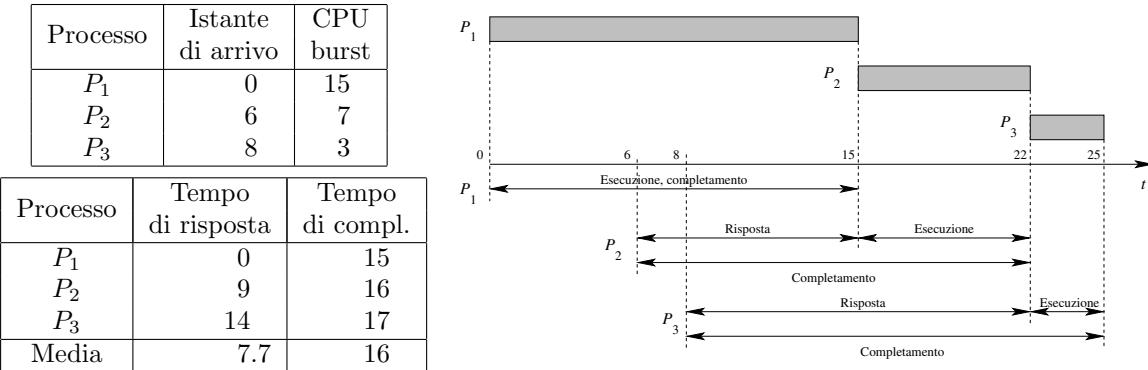


Figura 6.1: Tre processi gestiti da uno scheduler FCFS

Esempio

Figura 6.1, dove assumiamo che un processo dia un primo feedback all'avvio. È chiaro che l'ordine in cui arrivano i processi non è il più vantaggioso.

Vantaggi

- Semplice da realizzare.

Svantaggi

- Effetto carovana: un lungo CPU burst da parte di un processo ritarda tutti gli altri, per quanto brevi.
- Senza prelazione.

6.2.2 Shortest Job First (SJF)

- Con o senza prelazione.
- Si suppone che lo scheduler sia in grado di stimare la durata del prossimo CPU burst.
- Versione senza prelazione: al termine del burst CPU corrente, lo scheduler sceglie il processo con il più breve CPU burst.
- Versione con prelazione: all'arrivo di un nuovo processo P' , se il tempo rimanente del processo P in esecuzione è maggiore del tempo di burst del nuovo processo, allora P viene messo in stato Waiting e P' va in esecuzione al suo posto.

6.2.3 Esempio

Consideriamo l'esempio precedente. All'inizio, quando è arrivato solo P_1 , lo scheduler è forzato a lanciarlo. Al suo termine, però, sarebbe vantaggioso eseguire P_3 , che è più breve, anche se è arrivato dopo. Figura 6.2 illustra il vantaggio ottenuto nella selezione del job più breve. In Figura 6.3, infine, vediamo il comportamento di uno scheduler SJF con prelazione. All'arrivo di P_2 , lo scheduler reputa vantaggioso sospendere P_1 , al cui completamento mancano ancora 9 unità di tempo, a vantaggio di P_2 che ne richiederà solo 7. All'arrivo di P_3 , P_2 subisce la stessa sorte.

Incidentalmente, l'arrivo di processi sempre più corti causa un tempo di risposta immediato. Chiaramente non è sempre così: un processo con un CPU burst lungo dovrà attendere i processi con burst più brevi.

Processo	Tempo di risposta	Tempo di compl.
P_1	0	15
P_2	12	19
P_3	7	10
Media	6.3	14.7

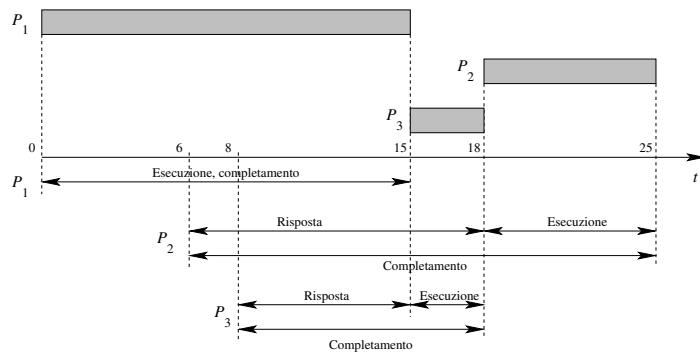


Figura 6.2: I tre processi gestiti da uno scheduler SJF senza prelazione.

Processo	Tempo di risposta	Tempo di compl.
P_1	0	25
P_2	0	10
P_3	0	3
Media	0	12.7

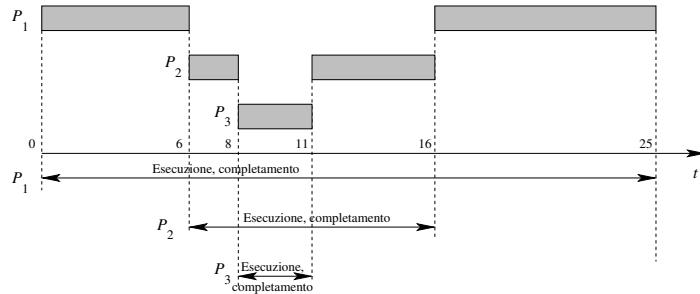


Figura 6.3: I tre processi gestiti da uno scheduler SJF con prelazione (SRTF — Shortest Remaining Time First).

Vantaggi

- SJF è dimostrabilmente ottimo per quanto riguarda il tempo di risposta e di completamento.
- Come abbiamo visto, la versione con prelazione (Shortest Remaining Time First, SRTF) può offrire tempi di prima risposta rapidi.

Svantaggi

- Conoscere il tempo di burst è difficile.
 - Nel caso di scheduling a lungo termine, si può utilizzare un eventuale tempo limite T_{limit} dichiarato dall'utente. Supponendo che il processo venga forzatamente terminato se richiede più di T_{limit} unità, e che SJF penalizza i processi con T_{limit} troppo lungo, l'utente è incoraggiato a stimare con precisione il tempo da dichiarare.
 - Per lo scheduling a breve termine, il tempo di un CPU burst per un processo si può stimare sulla base di una **media mobile** (a decadimento esponenziale) dei CPU burst precedenti dello stesso processo.
- Se t_0, t_1, t_2, \dots sono le durate dei CPU burst di un processo, fissato un parametro $0 \leq \alpha \leq 1$, la durata del primo burst viene stimata in modo grezzo (chiamiamo τ_0 questa stima), se necessario fissandola a un valore costante; le stime successive τ_1, τ_2, \dots dei tempi di burst successivi al primo possono essere calcolate “inseguendo” i tempi misurati in precedenza:

$$\begin{aligned}\tau_0 &= \text{costante}; \\ \tau_1 &= t_0; \\ \tau_i &= \alpha\tau_{i-1} + (1 - \alpha)t_{i-1}, \quad i = 2, \dots, n-1.\end{aligned}$$

Il parametro α rappresenta un fattore di decadimento delle misure precedenti:

- * Se $\alpha = 0$, allora $\tau_i = t_{i-1}$, senza “memoria” per i burst precedenti;
- * Se $\alpha = 1$, allora $\tau_i = \tau_1 = t_0$: le misure successive alla prima non hanno effetto sulla stima.

6.2.4 Scheduling a priorità

- Con o senza prelazione.
- Il prossimo processo scelto dallo scheduler è quello con la priorità più elevata¹.
- Generalizzazione di SJF (nel quale la priorità è data dal tempo di burst).
- Nulla impedisce alle priorità di avere una componente *statica* (“esterna”, ad esempio dichiarata dall’utente) e una componente *dinamica* (“interna”, calcolata dal sistema).

Per esempio, in SJF con prelazione:

- la componente statica è la durata totale del burst CPU (usata in SJF senza prelazione);
- la componente dinamica è il tempo trascorso in esecuzione dal processo;
- la priorità in un determinato istante è data dalla differenza fra le due (tempo di esecuzione rimanente).

Vantaggi

- Adatto a gestire emergenze in tempo reale (basta dare una priorità molto alta al processo che le gestisce).
- Permette di differenziare processi importanti (kernel) da quelli di minore rilevanza (utente).

Svantaggi

- Blocco indefinito (**starvation**): un processo può restare indefinitamente sospeso se continuano ad arrivare processi a priorità più elevata².
 - Possibile soluzione: aumentare la priorità col tempo.
 - Altra soluzione, non deterministica: la priorità determina soltanto la *probabilità* di esecuzione, anche i processi meno prioritari hanno una probabilità residua di ricevere un po’ di CPU di tanto in tanto.

Esempio: algoritmo non-preemptive Higher Response Ratio Next (HRRN) . Seguendo la convenzione (opposta a quella citata in precedenza, ma altrettanto valida) che valori più alti corrispondono a priorità più elevate, il valore di priorità R è dato dal rapporto

$$R = \frac{T_{\text{Waiting}} + T_{\text{Burst}}}{T_{\text{Burst}}} = 1 + \frac{T_{\text{Waiting}}}{T_{\text{Burst}}}. \quad (6.1)$$

A numeratore abbiamo il tempo CPU effettivamente consumato, mentre a denominatore aggiungiamo anche il tempo di waiting (quindi abbiamo il tempo totale trascorso dal processo nel sistema fino a questo istante, a esclusione del tempo trascorso in stato blocked). Il valore va ricalcolato a ogni nuova

¹ non esiste un accordo generale su come ordinare le priorità: in questo caso assumiamo che un valore più basso indichi una priorità più elevata (in Unix si parla di *niceness* del processo).

²Diceria non confermata: nel 1973 al MIT, allo spegnimento definitivo dell’IBM 7094 con CTSS (vedi sezione 1.2.3, pag. 14), è stato trovato un processo che aspettava di essere eseguito dal 1967.

Processo	Tempo di risposta	Tempo di compl.
P_1	0	25
P_2	0	16
P_3	1	8
Media	0.3	16.3

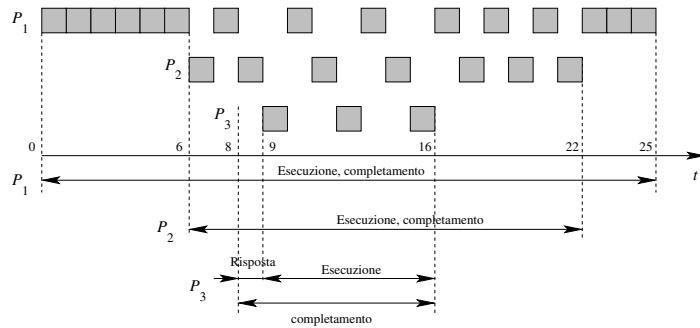


Figura 6.4: I tre processi gestiti da uno scheduler Round Robin con quanto di tempo unitario (tempi di cambio di contesto non considerati).

decisione dello scheduler.

Si noti che un processo che passa molto tempo in stato Waiting accumula una priorità sempre più elevata (cresce il numeratore).

Come esempio, consideriamo nuovamente la situazione in Fig. 6.2 all'istante $t = 15$, quando il processo P_1 è terminato e lo scheduler deve decidere quale altro processo avviare.

Il processo P_2 ha atteso per un tempo $T_{\text{Waiting}_2} = 15 - 6 = 9$, mentre P_3 ha atteso per un tempo $T_{\text{Waiting}_3} = 15 - 8 = 7$. In base a (6.1), le rispettive priorità risultano essere:

$$\begin{aligned} R_2 &= 1 + \frac{T_{\text{Waiting}_2}}{T_{\text{Burst}_2}} = 1 + \frac{9}{7} \approx 2.29, \\ R_3 &= 1 + \frac{T_{\text{Waiting}_3}}{T_{\text{Burst}_3}} = 1 + \frac{7}{3} \approx 3.33. \end{aligned}$$

Il sistema privilegia dunque P_3 , come nel caso SJF. Se il processo P_3 fosse arrivato più tardi, però, non avrebbe avuto il tempo di accumulare una priorità sufficientemente elevata, e sarebbe stato avviato P_2 .

6.2.5 Round-Robin (RR)

- Con prelazione, basato sulla definizione di un quanto di tempo (10 – 100ms).
- Coda di Waiting circolare, FIFO.
- Se ci sono n processi ogni processo ottiene (circa) $1/n$ del tempo di CPU.

Vantaggi

- Sistema equo, con bassi tempi di risposta: se ci sono n processi e il quanto è q , nessun processo resta in coda di attesa per più di $(n - 1)q$ unità di tempo (vedi Fig. 6.4).

Svantaggi

- Sempre da Fig. 6.4 si evince che i tempi di esecuzione e di completamento sono mediamente più elevati.

Problema: come stabilire la durata del quanto q ?

- Troppo lungo: si allungano i tempi di risposta del sistema (attesa prima del primo avvio); inoltre, se i processi terminano il burst prima della scadenza del quanto si ricade in FCFS.

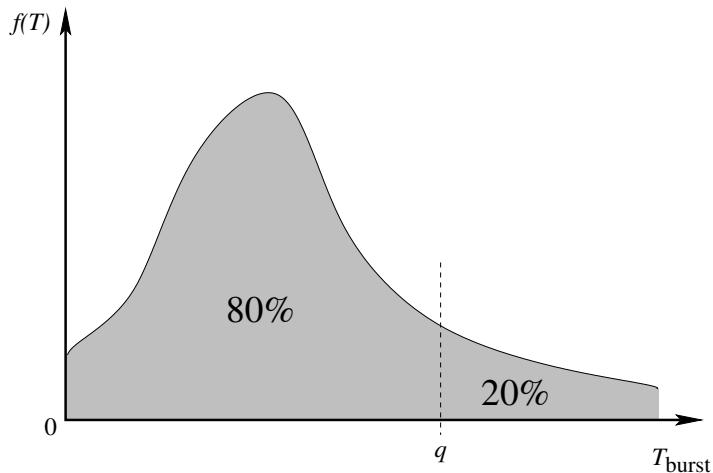


Figura 6.5: Scelta del quanto di tempo q affinché una frazione predefinita dei burst CPU (ad esempio l'80%) non subisca prelazione.

- Troppo corto: troppi cambi di contesto, l'overhead non è più trascurabile.
- Calibrare q in base ai percentili dei tempi di burst (per esempio: fare in modo che l'80% dei burst CPU non sia interrotto da prelazione, Fig. 6.5).

Come si vede da Fig. 6.6, la dipendenza del turnaround time dalla durata di q non è banale. Spesso il modo migliore per determinarlo è una simulazione, o tentativi effettuati su un sistema in esecuzione.

6.2.6 Code multilivello

Se è possibile identificare la tipologia di processo in esecuzione, il sistema potrebbe prevedere l'uso di più code, ad esempio:

- Una coda di processi interattivi (in primo piano, ‘textbf{foreground}’), da gestire con uno scheduler round-robin;
- una coda di processi batch (di sfondo, **background**), da gestire con un algoritmo FCFS.

Lo scheduler si occupa inanzitutto di suddividere il tempo CPU fra le varie code (ad esempio: 70% per la coda di foreground, 30% per quella di background; oppure: la coda di foreground ha priorità assoluta, e i processi di background non ricevono CPU finché ci sono processi in foreground in stato Waiting), assegnando opportunamente i quanti di tempo fra le code. Internamente a ciascuna coda, i processi vengono poi gestiti con l'algoritmo più adeguato³.

Possono esistere gerarchie più complesse, ad esempio:

- Processi di sistema, con vincoli di tempo, oppure non interrompibili;
- Processi interattivi, che devono ricevere di frequente del tempo CPU;
- Processi interattivi di editing: come sopra, ma con CPU burst brevi e quindi I/O bound;
- Processi batch: da eseguire sullo sfondo, quando la CPU è libera;

³Questo significa, ad esempio, che i processi della coda di background, eseguiti in FCFS, possono subire prelazione dai processi interattivi, ma mai tra loro: all'interno della coda di background nessun processo acquista tempo CPU se i processi che lo precedono non hanno terminato il loro burst

Processo	Istante di arrivo	CPU burst
P_1	0	6
P_2	0	3
P_3	0	1
P_4	0	7

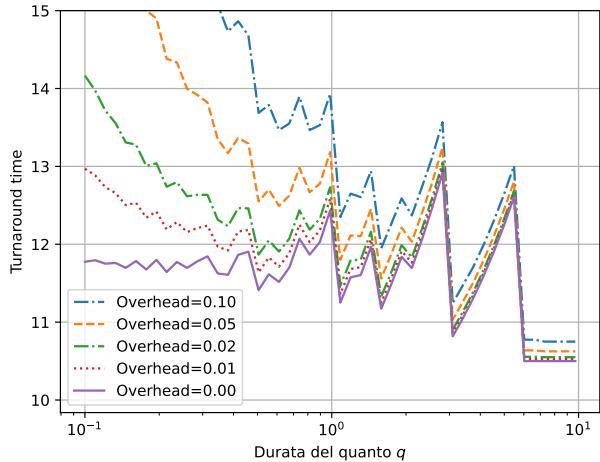


Figura 6.6: Il turnaround time in funzione del quanto q per diversi valori dell'overhead per il cambio di contesto.

- Processi “studente”.

Come detto sopra, le alternative per gestire il tempo CPU da assegnare alle code sono:

- assegnare più o meno quanti di tempo (oppure quanti di tempo più o meno lunghi) a ciascuna coda, secondo la priorità;
- eseguire processi in una coda solo se le code di priorità maggiore sono vuote (tutti i loro processi bloccati in I/O);
- un misto fra i due schemi.

Code multilivello con feedback

È anche possibile inserire un meccanismo di retroazione (feedback) che sposta i job fra le code secondo il loro comportamento osservato. Ad esempio, consideriamo un sistema a tre code:

- Q_0 : round-robin con quanto di tempo $q_0 = 8\text{ms}$;
- Q_1 : round-robin, $q_1 = 16\text{ms}$;
- Q_2 : FCFS.

con le seguenti regole:

- Un processo nella coda Q_i prelaziona il processo in esecuzione, se questo appartiene a una coda di priorità minore Q_j , $j > i$;
- Se è in esecuzione un processo della coda Q_0 e scade il suo quanto di tempo q_0 (quindi se il suo CPU burst non è terminato in tempo), questo viene spostato (retrocesso) alla coda Q_1 .
- Allo stesso modo, un processo della coda Q_1 il cui CPU burst non è ancora terminato entro il quanto q_1 viene spostato nella coda Q_2 .

In questo modo:

- il sistema privilegia processi con burst brevi (minori di 8ms);

- processi con burst minori di $(8 + 16)\text{ms}$ ricevono tempo CPU, seppure con priorità inferiore;
- processi che presentano burst più lunghi sono gestiti a bassa priorità;
- un processo che voglia restare in Q_0 può farlo, a condizione che i suoi burst CPU non superino gli 8ms ; ad esempio, un editor interattivo potrà probabilmente restare in Q_0 .

I sistemi a code multilivello rappresentano probabilmente il caso più generale di scheduler. La loro definizione richiede un certo numero di parametri:

- Quante code, e gli algoritmi di scheduling utilizzati da ciascuna coda;
- le regole per suddividere il tempo CPU fra le varie code;
- le regole per spostare un processo da una coda all'altra.

6.2.7 Processi e applicazioni

Un'ultima osservazione: lo scheduling fra processi privilegia gli utenti che ne mandano molti in esecuzione (che ricevono più CPU); oppure, un'applicazione in grado di suddividere il proprio lavoro in più processi in un sistema round-robin riceve un tempo CPU multiplo rispetto alle applicazioni a singolo processo.

Uno scheduler **fair-share** cerca di ovviare a questo assegnando tempo CPU non direttamente ai processi, ma a *gruppi* di processi individuati, ad esempio, dall'utente che li possiede o dal processo genitore che li ha lanciati.

Il tempo CPU viene suddiviso equamente fra i gruppi, e solo successivamente all'interno di ciascun gruppo il tempo viene assegnato a un processo specifico.

Uno scheduler fair-share può essere modellato come un sistema a code multiple, con una coda per gruppo.

Capitolo 7

Sincronizzazione fra processi

In questa parte, consideriamo il problema in cui più processi devono accedere a una stessa risorsa, ma l'accesso contemporaneo (o quasi, ad esempio in un sistema con prelazione) può causare problemi. Un esempio molto semplice è il seguente, illustrato in Fig. 7.1. Supponiamo che la variabile `i` sia condivisa fra più processi P_1 e P_2 (o fra due thread di uno stesso processo), che al momento valga `i=0`, ed entrambi i processi la debbano incrementare.

Il processo P_1 inizia l'incremento caricando il valore della variabile condivisa (zero) nel registro `eax`; supponiamo ora (possiamo seguire la cosa in Fig. 7.2) che scada il suo quanto di tempo, e che l'esecuzione passi a P_2 : anche P_2 inizia caricando il valore di `i` in `eax` (ovviamente, il *suo* registro `eax`: ogni processo ha il proprio grazie al cambio di contesto, come abbiamo visto). Ora, P_2 incrementa il valore del registro (seconda istruzione) e ne riporta il nuovo valore (uno) all'indirizzo di `i` (terza istruzione). A questo punto, abbiamo `i=1`.

Dopo un po', P_1 riprende il controllo della CPU; il suo contesto viene recuperato, in particolare `eax=0`, e riprende eseguendo la seconda istruzione: incrementa `eax` a 1, e lo memorizza nella memoria condivisa.

In seguito a due incrementi distinti, la variabile `i` è passata da 0 a 1. Se la prelazione non fosse avvenuta *durante* l'incremento, allora `i` varrebbe 2: l'accesso condiviso ha introdotto un errore **non deterministico**, quindi molto complesso da individuare e risolvere, soprattutto perché è molto facile cadere nell'errore di considerare un'istruzione semplice come `i++` come se fosse *atomica* (non prelazionabile).

```
.text
.globl i
.bss
.align 4
.type i, @object
.size i, 4
int i = 0;
...
i++;
    i:
        .zero 4
    ...
        movl i(%rip), %eax # i, i.1_2
        addl $1, %eax      #, _3
        movl %eax, i(%rip) # _3, i
```

Figura 7.1: Una semplice istruzione di incremento (sinistra), apparentemente atomica, si traduce in tre istruzioni assembly.

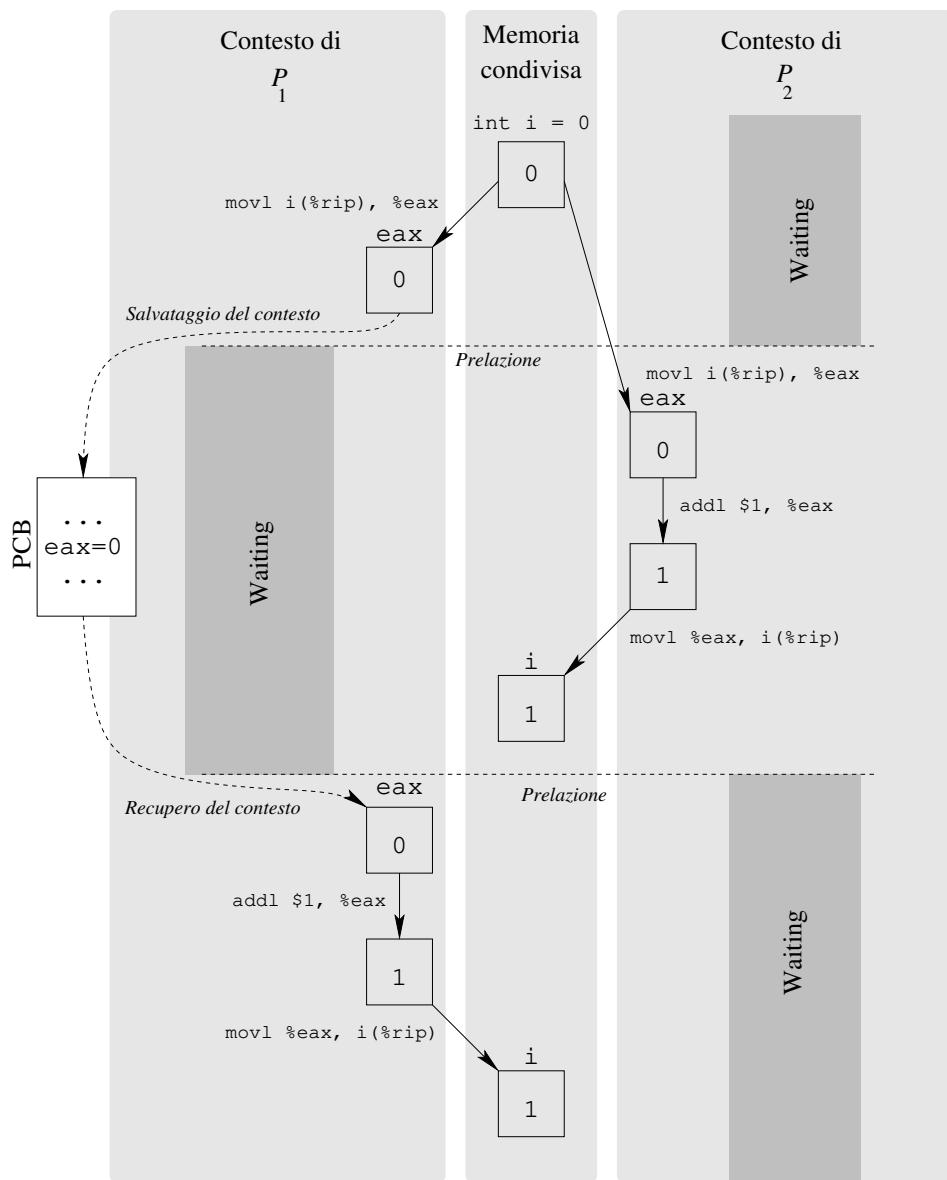


Figura 7.2: La non-atomicità dell'istruzione di incremento

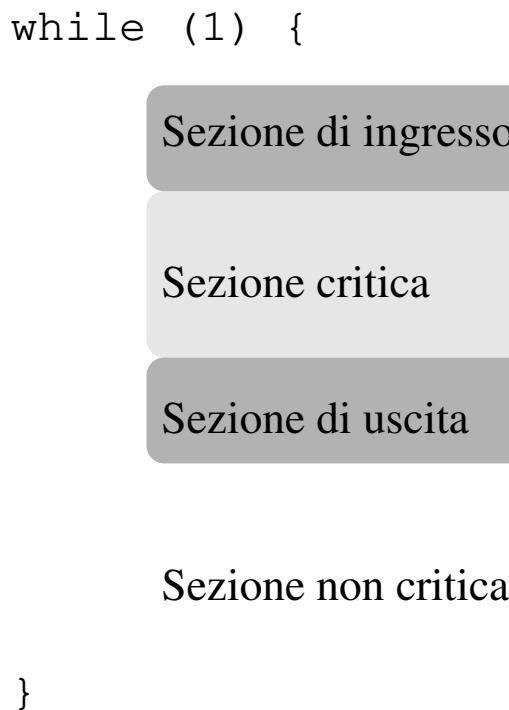


Figura 7.3: Un processo come alternanza fra una “sezione critica” di accesso a una risorsa e codice non critico.

7.1 La regione critica

L'esempio appena descritto dimostra che esistono determinate sezioni di codice, per quanto riguarda l'accesso a una risorsa (ad esempio memoria condivisa), che devono essere eseguite in modo mutuamente esclusivo.

In generale possiamo modellare un processo, dal punto di vista dell'accesso a una risorsa condivisa, come in Fig. 7.3: il processo si trova, di tanto in tanto, ad eseguire una sezione di codice in cui deve garantirsi l'accesso esclusivo a una risorsa condivisa. Il codice di accesso alla risorsa (la **sezione critica**) va protetto con alcune istruzioni iniziali che ne garantiranno l'accesso (la *sezione di ingresso*) e alcune istruzioni finali (la *sezione d'uscita*) che segnalano il termine della sezione per consentire ad altri processi di accedere alla stessa risorsa.

L'accesso alla sezione critica deve soddisfare tre criteri:

- **Esclusione mutua** (mutual exclusion) — in ogni istante, solo un processo può trovarsi all'interno della sezione critica.
Di conseguenza, se questo processo viene interrotto per prelazione, ogni altro processo che voglia accedere alla propria sezione critica dovrà venire bloccato nella sezione di ingresso.
- **Progresso** (progress) — L'ingresso alla sezione critica deve essere deciso solo tra i processi in attesa di entrare, e la decisione su chi entrerà non può essere rinviaata indefinitamente.
- **Attesa limitata** (bounded waiting) — nessun processo deve rischiare di restare bloccato indefinitely nella sezione di ingresso.

Nel seguito assumeremo dunque che il codice che regola l'accesso alla sezione critica consista nella lettura/scrittura di celle di memoria condivisa da parte dei processi.

Distinguiamo i possibili approcci fra soluzioni puramente software e soluzioni “hardware” che utilizzano specifiche istruzioni atomiche messe a disposizione dal processore.

```

1. int turno ← 0                                variabile condivisa
2. processo i:
3.   [ j ← 1 - i                                Indice dell'altro processo
4.   finché (1):
5.     [ finché turno ≠ i;      sezione di ingresso: aspetto che sia il mio turno
6.       Sezione critica
7.       turno ← j                                ezione di uscita: è il turno dell'altro
8.     ] Sezione non critica

```

Figura 7.4: Sincronizzazione fra due processi P_1 e P_2 : i è l'indice del processo e vale 0 oppure 1, mentre j rappresente l'indice dell'altro processo.

```

1. bool ingresso[2] ← { falso, falso }           vettore condiviso
2. processo i:
3.   [ j ← 1 - i                                Indice dell'altro processo
4.   finché (1):
5.     [ ingresso[i] ← vero                  segnalo di voler entrare
6.       finché ingresso[j];                 aspetto che l'altro sia fuori
7.       Sezione critica
8.       ingresso[i] ← falso                ezione di uscita: non sono più dentro
9.     ] Sezione non critica

```

Figura 7.5: Sincronizzazione fra due processi P_1 e P_2 , secondo tentativo.

7.1.1 Soluzioni software

Supponiamo che due processi, P_0 e P_1 , debbano accedere a turno a una sezione critica di codice nella quale agiscono su una risorsa condivisa in modo mutuamente esclusivo.

Una prima soluzione, illustrata in Fig. 7.4, consiste nell'uso di una variabile condivisa `turno` il cui valore indica quale processo potrà entrare nella sezione critica. Ciascuno dei due processi attende il proprio turno nel ciclo di linea 5; al termine della sezione critica, il turno viene passato all'altro processo (linea 7).

Teorema 1. *Il codice di Fig. 7.4 garantisce l'esclusione mutua dei due processi.*

Dimostrazione. Se P_0 si trova nella sezione critica, significa che ha superato la sezione di ingresso, il che è possibile solo se `turno` = 0. Ma il valore di `turno` non cambierà in 1 finché il processo P_0 non uscirà dalla sezione critica, e fino a quel momento il processo P_1 non potrà superare la sezione di ingresso. Idem invertendo gli indici. \square

Purtroppo, il codice di Fig. 7.4 non soddisfa il requisito del progresso: una volta uscito dalla sezione critica, il processo P_i non vi potrà rientrare finché P_1 non sarà entrato a sua volta: se il processo P_j non è più interessato alla risorsa, P_i si trova costretto ad attendere indefinitamente, mentre la decisione di entrare dovrebbe dipendere solo da lui (in quanto unico processo interessato).

Una possibile soluzione al problema del progresso è riportata in Fig. 7.5: ciascuno dei due processi P_i ha a disposizione un flag `ingresso[i]`: ponendolo a `vero`, il processo P_i dichiara la propria intenzione di entrare nella sezione critica. Se P_j non è interessato, il suo flag resta falso, e P_i può entrare.

Purtroppo, il codice presenta il rischio che i due processi superino contemporaneamente (o comunque in stretta successione) la prima istruzione di ingresso (linea 5), prima che uno dei due arrivi al ciclo di attesa (lines 6). Una volta che entrambi i flag sono veri, i due processi restano entrambi in attesa nei cicli, senza potersi sbloccare l'un l'altro. Si dice che i due processi sono in **deadlock**.

Anche invertendo l'ordine delle righe 5 e 6, risolvendo il problema del deadlock, i due processi rischiano di superare entrambi il ciclo di attesa prima di impostare il flag, violando il requisito di esclusione mutua.

<pre> 1. int turno ← 0 2. bool ingresso[2] ← { falso, falso } 3. processo <i>i</i>: 4. ┌───┐ <i>j</i> ← 1 – <i>i</i> valore condiviso 5. ┌───┐ finché (1): vettore condiviso 6. ┌───┐ ingresso[<i>i</i>] ← vero Indice dell'altro processo 7. ┌───┐ turno ← <i>j</i> segnalo di voler entrare 8. ┌───┐ finché ingresso[<i>j</i>] e turno = <i>j</i>; Cedo il turno all'altro processo 9. ┌───┐ Sezione critica aspetto che l'altro sia eventualmente fuori 10. ┌───┐ ingresso[<i>i</i>] ← falso sezione di uscita: non sono più dentro 11. ┌───┐ Sezione non critica └───┘ </pre>
--

Figura 7.6: Combinazione dei due approcci precedenti.

La condizione può essere evitata combinando i due approcci illustrati, come in Fig. 7.6.

Teorema 2. *Il codice di Fig. 7.6 soddisfa tutti i requisiti per l'accesso alla sezione critica.*

Dimostrazione. Osserviamo che ciascun processo scrive solo il proprio flag `ingresso`, mentre può leggere anche quello dell'altro. Ciascun processo, inoltre, può impostare `turno` solamente al valore dell'altro processo (P_0 la può impostare solo a 1 e viceversa). **Esclusione mutua** — Per assurdo, supponiamo che entrambi i processi siano nella sezione critica. Il processo P_i può entrare nella sezione critica se e soltanto se la condizione di linea 8 è falsa, ovvero

$$\text{ingresso}[j] = \text{falso} \quad \text{o} \quad \text{turno} = i. \quad (7.1)$$

Se P_i e P_j fossero entrambi nella sezione critica, però,

$$\text{ingresso}[i] = \text{ingresso}[j] = \text{vera}.$$

Quindi, perché si verifichi la condizione (7.1) si riduce a `turno` = *i*, che non può essere vera per entrambi i valori di *i*.

Progresso e attesa limitata — Supponiamo che P_i voglia entrare nella sezione critica. Se P_j non vuole entrare, allora `ingresso[j] ← falso`, e P_i entra.

Se invece anche P_j vuole entrare e ha impostato `ingresso[j] ← vero`, allora entrambi sono al ciclo di attesa; ma `turno` vale *i* oppure *j*, quindi uno dei due passa ed entra nella sezione critica.

Supponendo che entri prima P_j , questi uscendo imposterà `ingresso[j] ← falso`, lasciando così entrare P_i . Anche se P_i non ricevesse tempo CPU prima del prossimo tentativo di ingresso di P_j , quest'ultimo imposterebbe `turno ← i`. Quindi P_i non deve mai aspettare più di un turno di *j* prima di entrare nella sezione critica. \square

Per il caso generale di N processi, si può usare l'*algoritmo del fornaio* (bakery's algorithm). L'idea è che ogni processo sceglie un numero nella sezione di ingresso, e quello col numero inferiore passa.

Ecco un primo abbozzo in Fig. 7.7: ogni processo ha un numero (se nullo, significa che il processo non è in attesa della sezione critica); nella sezione di ingresso, in linea 4, il processo sceglie un numero maggiore di quelli dei processi già entrati o in attesa. Poi attende il completamento di tutti quelli con numero inferiore (linee 5–6). A questo punto entra, poi per segnalare l'uscita riporta il proprio numero a zero.

Perché l'algoritmo funzioni, però, è necessario correggere due difetti:

- Due processi potrebbero entrare contemporaneamente nella sezione di ingresso, e scegliere lo stesso numero: a questo punto nessuno dei due bloccherebbe l'altro e si violerebbe il requisito dell'esclusione mutua.

Per ovviare a questo problema, è sufficiente che, a parità di numero scelto, la priorità vada al processo con indice inferiore.

```

1. bool numero[N]  $\leftarrow \{ 0, 0, \dots \}$  vettore condiviso
2. processo  $i$ :
3.    $\lceil$  finché (1):
4.     numero[i]  $\leftarrow \max(\text{numero}) + 1$  scelgo il numero
5.     per  $j \leftarrow 0, 1, \dots, N$ 
6.       finché numero[j]  $\neq 0$  e numero[j] < numero[i]; aspetto i processi con numero inferiore
7.       Sezione critica
8.         numero[i]  $\leftarrow 0$  sezione di uscita: non sono più dentro
9.       Sezione non critica

```

Figura 7.7: Una versione semplificata dell'algoritmo del fornaio. L'indice i del processo può variare da 0 a $N - 1$.

```

1. bool ingresso[N]  $\leftarrow \{ \text{falso}, \text{falso}, \dots \}$  vettore condiviso
2. bool numero[N]  $\leftarrow \{ 0, 0, \dots \}$  vettore condiviso
3. processo  $i$ :
4.    $\lceil$  finché (1):
5.     ingresso[i]  $\leftarrow \text{vero}$  Segnalo che sto prendendo un numero
6.     numero[i]  $\leftarrow \max(\text{numero}) + 1$  scelgo il numero
7.     ingresso[i]  $\leftarrow \text{falso}$ 
8.     per  $j \leftarrow 0, 1, \dots, N$ 
9.        $\lceil$  finché ingresso[j]; aspetto che  $j$  finisce di scegliere il numero
10.         $\lceil$  finché numero[j]  $\neq 0$  e (numero[j],  $j$ )  $<$  (numero[i],  $i$ ); aspetto i processi con numero inferiore
11.        Sezione critica
12.          numero[i]  $\leftarrow 0$  sezione di uscita: non sono più dentro
13.        Sezione non critica

```

Figura 7.8: Pseudocodice completo dell'algoritmo del fornaio.

- Un processo P_i potrebbe scegliere un numero, ma subire prelazione prima di scriverlo nella sua cella `numero[i]`. Un altro processo P_j potrebbe a questo punto prendere lo stesso numero ed entrare nella sezione critica, non sapendo dell'intenzione di P_i . Se $i < j$, allo sblocco di P_i questo sarebbe comunque convinto di avere la priorità, ed entrerebbe anche lui.

Per ovviare al problema, possiamo introdurre un flag booleano per ciascun processo. Il flag `ingresso[i]` indica che il processo P_i sta scegliendo il proprio numero. Il codice completo è in Fig. 7.8. Alla linea 10 abbiamo usato l'*ordine lessicografico* per le tuple (`ingresso[i]`, i) così definito:

$$(a_1, b_1) < (a_2, b_2) \quad \text{se e solo se} \quad a_1 < a_2 \quad \text{o} \quad (a_1 = a_2 \quad \text{e} \quad b_1 < b_2).$$

7.1.2 Soluzioni hardware

Una soluzione hardware per ovviare al problema della sezione critica è quello di disabilitare gli interrupt durante la sua esecuzione. Sorgono però alcuni problemi:

- Se la sezione critica è lunga, gli interrupt non possono restare disabilitati troppo a lungo oppure il sistema ne risente.
- Non è necessario bloccare completamente la prelazione, ma solo impedire che altri processi accedano alla sezione critica relativa alla stessa risorsa.

Una soluzione è quella di mettere a disposizione una singola istruzione macchina (quindi non interrompibile) in grado di effettuare più di un'operazione altrimenti non atomica.

Test-and-set

Una possibile istruzione offerta da alcune architetture è la test-and-set, che opera su una cella di memoria booleana. Ecco una possibile implementazione in linguaggio C++, nell'ipotesi che le istruzioni di cui è composta non siano interrompibili:

```
bool test_and_set(bool &var) {
    bool tmp = var;
    var = TRUE;
    return tmp;
}
```

L'accesso a una sezione critica verrebbe così regolato da una variabile booleana condivisa lock:

```
bool lock = FALSE;
while(1) {
    while(test_and_set(lock)); // solo il primo processo che trova lock==FALSE passa,
                               // gli altri la ritrovano subito TRUE.
    // SEZIONE CRITICA
    lock = FALSE;
    // sezione non critica
}
```

Compare-and-swap

L'istruzione `test_and_set` è un caso particolare di un'istruzione più completa:

```
int compare_and_swap(int &var, int obiettivo, int nuovo_valore) {
    int tmp = var;
    if (var == obiettivo) var = nuovo_valore;
    return tmp;
}
```

Swap

Per garantire l'esclusione mutua, è sufficiente anche una swap atomica che scambia i valori di due variabili:

```
void swap(bool &a, bool &b) {
    bool tmp = a;
    a = b;
    b = tmp;
}
```

Il seguente algoritmo garantisce l'esclusione mutua (fintantoché la `swap` è atomica):

```
bool lock = FALSE;
while(1) {
    bool dummy = TRUE;
    while(dummy)
        swap(dummy, lock); // solo il primo processo che trova lock==FALSE passa,
                           // gli altri la ritrovano subito TRUE.
    // SEZIONE CRITICA
    lock = FALSE;
    // sezione non critica
}
```

```

// variabili condivise
bool attesa[N] = {FALSE, FALSE, ... };
bool lock;

while (1) {
    attesa[i] = TRUE; // mi dichiaro in attesa
    // aspetto che qualcuno mi sblocca o che lock si liberi:
    while (attesa[i] && !test_and_set(lock));
    attesa[i] = FALSE; // non sono più in attesa.
    // SEZIONE CRITICA
    // Ora cerca il prossimo processo j in attesa:
    int j = (i+1) % N; while (j != i && !attesa[j]) j = (j+1) % N;
    if (j == i) lock = FALSE; // se nessun processo è in attesa, libero il lock
    else attesa[j] = FALSE; // altrimenti sblocco j
    // Sezione non critica
}

```

Figura 7.9: Accesso a sezione critica da parte di N processi con istruzione `test_and_set` atomica.

```

void signal(int &S) {
    S++;
}

void wait(int &S) {
    while (S==0);
    S--;
}

```

Figura 7.10: Implementazione “concettuale” di un semaforo intero S .

Progresso e attesa limitata

Purtroppo, gli utilizzi di `test_and_set` e di `swap` visti sopra garantiscono solo la mutua esclusione. Se ci sono più processi bloccati nel `while` di attesa, nulla garantisce che tutti verranno prima o poi sbloccati. Un processo P_i potrebbe, in linea di principio, vedersi sempre passare avanti qualche altro processo, rimanendo bloccato indefinitamente (**starvation**).

L'algoritmo di Fig.7.9 garantisce tutti i requisiti per l'accesso a una sezione critica per N processi o thread.

7.2 Semafori

Le soluzioni presentate prima, per quanto funzionali, presentano due problemi:

- non sono banali da realizzare, richiedendo codice ad hoc;
- sono basate su busy waiting, quindi sprecano CPU.

Un **semaforo** (intero) è una variabile intera S a cui si accede attraverso due primitive atomiche fornite dal sistema:

- `signal(S)` ne incrementa il valore di 1;
- `wait(S)` tenta di decrementare il valore di S , ma finché $S = 0$ tiene bloccato il processo.

La Fig. 7.10 ne illustra il concetto. Ovviamente, una realizzazione concreta deve impedire che più processi collidano nel gestire la variabile.

```

void signal(bool &S) {
    S = TRUE;
}
void wait(bool &S) {
    while (!S);
    S = FALSE;
}

```

Figura 7.11: Implementazione “concettuale” di un semaforo binario S .

```

semaforo_binario S = TRUE;
while (1) {
    wait(S);
    // SEZIONE CRITICA
    signal(S);
    // sezione non critica
}

```

Figura 7.12: Utilizzo di base di un semaforo binario S per l’accesso esclusivo a una sezione critica.

Un semaforo *binario* è come sopra (vedi Fig. 7.11), ma può assumere solo due valori, 0 e 1 (oppure vero e falso).

Infine, in Fig. 7.12 vediamo il tipico utilizzo di un semaforo binario a protezione di una sezione critica: la sezione di accesso consiste nella `wait()` in cui il processo resta eventualmente bloccato in attesa che la risorsa si liberi. La sezione di uscita è la `signal()`.

Ora mostriamo che, se si dispone di soli semafori binari, l’implementazione di semafori interi è immediata:

Teorema 3. *I semafori binari hanno lo stesso potere espressivo dei semafori interi.*

Dimostrazione. Consideriamo il codice di Fig. fig:binario-intero: per implementare un semaforo intero utilizziamo un contatore N con la seguente semantica:

- Se $N \geq 0$, allora il valore è quello della definizione di semaforo intero: rappresenta il numero di processi che possono ancora entrare prima che il semaforo blocchi;
- se $N < 0$, allora non ci sono risorse disponibili e $|N|$ processi sono in attesa.

Per coordinare l’accesso atomico a N utilizziamo un primo semaforo, `mutex`, in modo che un solo processo alla volta possa modificare e successivamente leggere la variabile.

Per coordinare l’accesso alla zona critica, utilizziamo un semaforo `accesso` che entra in gioco solamente se il valore di N è negativo, indicando che il processo in esecuzione deve attendere di essere sbloccato da una `signal()` sullo stesso semaforo. \square

7.2.1 Implementazione di un semaforo intero senza busy waiting

Due definizioni:

- Un lock basato su busy waiting è anche detto **spinlock** (in quanto resta bloccato sin un loop).
- Un semaforo binario, inizializzato a vero e utilizzato per controllare un accesso esclusivo viene detto **mutex**.

Osservando l’implementazione di Fig. 7.13, notiamo che:

- il semaforo `mutex` protegge una sezione critica estremamente veloce (un incremento/decremento seguito da un test);

```

struct semaforo_intero {
    semaforo_binario mutex = TRUE;
    semaforo_binario attesa = TRUE;
    int N;
}

void wait(semaforo_intero &S) {
    wait(S.mutex);
    S.N--;
    if (S.N < 0) {
        // Fine delle risorse.
        // il processo deve attendere
        signal(S.mutex);
        wait(S.attesa);
    } else
        // sblocca comunque il mutex
        signal(S.mutex);
}

void signal(semaforo_intero &S) {
    wait(S.mutex);
    S.N++;
    if (S.N <= 0) {
        // C'erano processi in attesa;
        // uno va sbloccato.
        signal(S.mutex);
        signal(S.attesa);
    } else
        signal(S.mutex);
}

```

Figura 7.13: Implementazione di un semaforo intero S utilizzando due semafori binari.

```

void signal(bool &S) {
    S = TRUE;
}

void wait(bool &S) {
    bool tmp = FALSE;
    while (!tmp)
        swap(tmp, S);
}

```

Figura 7.14: Implementazione di un semaforo binario S con primitiva hardware atomica `swap()`.

```

        struct semaforo_intero {
            semaforo_binario mutex = TRUE;
            PCB *coda;
            int N;
        }

void wait(semaforo_intero &S) {
    wait(S.mutex);
    S.N--;
    if (S.N < 0) {
        // Fine delle risorse.
        // il processo deve attendere
        // nella coda di S
        signal(S.mutex);
        inserisci(questo_processo, S.coda);
        sleep();
    } else
        // sblocca comunque il mutex
        signal(S.mutex);
}

void signal(semaforo_intero &S) {
    wait(S.mutex);
    S.N++;
    if (S.N <= 0) {
        // C'erano processi in attesa;
        // il più vecchio presente in coda
        // va svegliato.
        signal(S.mutex);
        PCB *p = estrai(S.coda);
        wakeup(p);
    } else
        signal(S.mutex);
}

```

Figura 7.15: Implementazione di un semaforo intero S utilizzando un semaforo binario e una coda FIFO di attesa.

- il semaforo **attesa**, invece, può comportare attese anche molto lunghe, perché dipende dall'uso che si fa del semaforo a protezione di codice potenzialmente complesso.

Di conseguenza, **mutex** può essere implementato anche con uno spinlock come quello di Fig. 7.14 (con la **swap()** atomica già vista prima). Infatti, le probabilità che il processo subisca prelazione proprio durante l'esecuzione delle due operazioni è molto bassa.

Il semaforo binario **attesa** dovrebbe invece essere sostituito da un meccanismo più efficiente che garantisca, oltre al risparmio di tempo CPU, i requisiti di attesa limitata e progresso che un semplice spinlock non può soddisfare.

Un possibile approccio può essere quello di rimuovere la **wait(attesa)** mettendo il processo in stato Blocked in una coda appositamente realizzata per il semaforo, come in Fig. 7.15:

- **coda** è una lista concatenata di PCB nella quale vengono inseriti tutti i processi in attesa del semaforo S ; è gestita tramite le due primitive **inserisci()** ed **estrai()**.
- la primitiva **sleep()** mette il processo in esecuzione in stato Blocked (attesa di un evento); l'evento è, ovviamente, lo sblocco del semaforo per il processo;
- la primitiva **wakeup()** mette il PCB passato ad argomento nella coda Waiting, pronto ad essere eseguito.

A differenza delle soluzioni viste finora, questa implementazione richiede di essere in kernel mode (accesso ai PCB, addormentamento e risveglio di processi).

Altre possibili alternative al busy waiting:

- Soppressione degli interrupt per bloccare la prelazione: poco pratica in ambienti multiprocessore (ogni processore deve disabilitarli).
- Funzioni kernel non prelazionabili (OK se esiste la possibilità, ma di nuovo il problema sono i sistemi multiprocessore).

```

semaforo_binario S = FALSE;

Processo P:                                Processo Q:

...
Sezione A;                                ...
signal(S); // Segnala che A è completa    wait(S); // Attendi che A sia completa
...                                         Sezione B;
...                                         ...

```

Figura 7.16: Uso di un semaforo binario come barriera: *Q* attende a eseguire la sezione B finché *P* non ha completato la sezione A.

```

semaforo_binario S_P = FALSE;
semaforo_binario S_Q = FALSE;

Processo P:                                Processo Q:

...
signal(S_Q); // Q può passare           ...
wait(S_P); // Posso passare?          signal(S_P); // P può passare
Sezione A_P;                            wait(S_Q); // Posso passare?
...                                         Sezione A_Q;
...                                         ...

```

Figura 7.17: Uso di due semafori binari come barriera mutua: nessuno dei due processi può procedere finché non sono entrambi pronti.

7.2.2 Altri usi dei semafori

Abbiamo già visto due possibili problemi di sincronizzazione risolvibili con l'uso di un semaforo:

- Esclusione mutua da una sezione critica;
- Sincronizzazione nel problema produttore/consumatore.

Segue qualche altro esempio di uso.

Barriere

Supponiamo che un processo *Q* non debba eseguire una sezione di codice B prima che il processo *P* abbia eseguito la sezione A. La soluzione, illustrata in Fig. 7.16 è quella di utilizzare un semaforo binario *S* come “barriera”. È facile verificare che, indipendentemente dall'ordine in cui i due processi arrivano alla barriera, la sezione B non viene eseguita fino al completamento della sezione A.

Un problema più complesso è quello di realizzare una barriera “mutua”: ciascun processo (*P*, *Q*) ha una sezione di codice (*A_P*, *A_Q*) nella quale può entrare solamente quando anche l'altro è pronto. Una possibile soluzione, visibile in Fig. 7.17, consiste in due semafori utilizzati in modo simmetrico. Il primo processo che arriva alla sezione `signal/wait()` deve attendere che arrivi anche il secondo.

Supponiamo che due processi *P* e *Q* possiedano una sezione critica, rispettivamente *A_P* e *A_Q*, nella quale debbono alternarsi. Anche in questo caso, possiamo usare due semafori, uno solo dei quali inizializzato a “vero”. In Fig. 7.18 ogni processo attende il proprio turno e alla fine della propria sezione abilita il turno dell'altro.

```

semaforo_binario S_P = TRUE;
semaforo_binario S_Q = FALSE;

Processo P:                                Processo Q:

while(1) {
    ...
    wait(S_P);    // Posso passare?
    Sezione A_P;
    signal(S_Q); // Ora tocca a Q
    ...
}

while(1) {
    ...
    wait(S_Q);    // Posso passare?
    Sezione A_Q;
    signal(S_P); // Ora tocca a Q
    ...
}

```

Figura 7.18: Uso di due semafori binari per alternare due processi in una sezione critica.

Oltre alla struttura del buffer circolare:

```

semaforo_binario mutex = TRUE; // mutua esclusione per accesso a buffer
semaforo_intero libero = DIMENSIONE_BUFFER; // contatore di posti liberi
                                                // (blocca il produttore se ==0 -- non ce ne sono)
semaforo_intero contenuto = 0; // contatore di dati disponibili
                                // (blocca il consumatore se ==0 -- non c'è contenuto)

```

Processo produttore:

```

while(1) {
    produci un dato;
    wait(libero); // attendi che
                   // ci sia posto
    wait(mutex); // assicurati di
                   // essere solo
    deposita il dato; // sezione critica
    signal(mutex);
    signal(contenuto); // incrementa il
                       // numero di elementi
                       // nel buffer
}

```

Processo consumatore:

```

while(1) {
    wait(contenuto); // attendi che
                     // ci sia qualcosa
    wait(mutex); // assicurati di
                  // essere solo
    preleva il dato; // sezione critica
    signal(mutex);
    signal(libero); // incrementa il
                    // numero di posti liberi
                    // nel buffer
    Utilizza il dato;
}

```

Figura 7.19: Uso di semafori per il problema produttore/consumatore.

```

semaforo_binario mutex_scrittura = TRUE; // mutua esclusione fra scrittori
semaforo_binario mutex_lettura = TRUE; // gestione contatore lettori
int numero_lettori = 0; // contatore di lettori

Processi lettori:
while(1) {
    ...
    wait(mutex_lettura);
    if (++numero_lettori == 1)
        wait(mutex_scrittura);
    signal(mutex_lettura);
    LEGGI I DATI
    wait(mutex_lettura);
    if (--numero_lettori == 0)
        signal(mutex_scrittura);
    signal(mutex_lettura);
}

Processi scrittori:
while(1) {
    ...
    wait(mutex_scrittura);
    MODIFICA I DATI
    signal(mutex_scrittura);
    ...
}

```

Figura 7.20: Uso di due semafori binari per alternare due processi in una sezione critica.

Il problema produttore/consumatore

Già visto, ma Fig. 7.19 illustra una soluzione con semafori; prima di operare sul buffer (protetto da un mutex), i processi mantengono due semafori che fungono da contatori di posti liberi e occupati, e con opportune `wait()` attendono le condizioni per poter operare.

Il problema lettori/scrittori

Esiste una risorsa condivisa (un'area di memoria, oppure un file); alcuni processi sono “lettori” (possono solo leggere l'area), altri processi sono “scrittori” (possono scrivere). Regole:

- Più lettori possono leggere l'area contemporaneamente.
- Un solo scrittore alla volta può scrivere.
- Finché ci sono lettori all'opera, nessuno scrittore può scrivere.

Una soluzione è illustrata in Fig. 7.20. I processi produttori sono molto semplici: proteggono la loro sezione critica con un mutex (`mutex_scrittura`) per garantire che solamente uno di loro acceda. I lettori sono più complessi: proteggono con un altro mutex `mutex_lettura` l'accesso al contatore del numero di lettori nella sezione critica o in attesa di entrarvi. Se un lettore si accorge di essere il primo a entrare cerca di acquisire `mutex_scrittura` finché un eventuale scrittore termina la propria opera. Quando un lettore termina, se è l'ultimo libera un eventuale scrittore in attesa.

Un problema: c'è il rischio di starvation degli scrittori se i lettori continuano a entrare e uscire.

7.2.3 Rischi nell'uso dei semafori

Deadlock

Supponiamo che due o più processi necessitino di due risorse diverse per una sezione di codice, e che ciascuna risorsa sia regolata da un mutex. Per entrare nella sezione critica, dunque, un processo deve acquisire entrambi i semafori con due `wait()` in successione, e rilasciarli alla fine.

Supponiamo ora che, in questo scenario, due processi P e Q acquisiscano i due mutex in ordine diverso, come in Fig. 7.21. Se il processo P subisce prelazione subito dopo la `wait(mutex_A)` e subentra

```

semaforo_binario mutex_A = TRUE; // controllo della risorsa A
semaforo_binario mutex_B = TRUE; // controllo della risorsa B

```

Processo *P*:

```

...
wait(mutex_A);
wait(mutex_B);
SEZIONE CRITICA
signal(mutex_B);
signal(mutex_A);
...

```

Processo *Q*:

```

...
wait(mutex_B);
wait(mutex_A);
SEZIONE CRITICA
signal(mutex_A);
signal(mutex_B);
...

```

Figura 7.21: Un possibile deadlock fra processi che devono acquisire due semafori per procedere in una sezione critica.

il processo *Q* a eseguire la prima `wait(mutex_B)`, entrambi i processi resteranno bloccati sulla seconda `wait()`, in quanto ciascuna risorsa è già stata acquisita dall'altro processo e non verrà liberata se non al termine della sezione critica. I due processi sono in situazione di **deadlock**.

Il problema dei cinque filosofi (dining philosophers)

Problema “accademico”, ma illustra difficoltà reali nella sincronizzazione fra processi.

Cinque filosofi sono seduti intorno a un tavolo. Passano il tempo a pensare, ma di tanto in tanto devono mangiare. Ciascuno ha davanti a sé un piatto di cibo e, alla sua destra e sinistra, due bastoncini, ognuno condiviso col vicino (Fig. 7.22). Per mangiare un filosofo deve avere in mano entrambi i bastoncini, ma li può acquisire solo in sequenza. Al termine del pasto il filosofo rilascia i bastoncini e torna a pensare. È ovvio il parallelo con i processi e l'accesso a una sezione critica che richiede l'acquisizione di risorse.

Soluzione intuitiva Ogni bacchetta è controllata da un semaforo. Ogni filosofo, quando vuole mangiare, acquisisce prima il bastoncino alla sua destra, poi quello alla sua sinistra.

La soluzione è troppo simmetrica: se tutti i filosofi decidono di mangiare contemporaneamente, tutti prendono il bastoncino alla propria destra e restano in attesa indefinita che si liberi quello alla loro sinistra, appena acquisito dal filosofo successivo in senso orario. Si tratta di una generalizzazione del caso di deadlock visto poco fa.

Possibili soluzioni corrette È necessario “rompere la simmetria”. Alcune possibilità:

- I filosofi in posizione pari (a partire da un “filosofo zero” concordato da tutti) prendono per primo il bastoncino di destra, gli altri quello di sinistra.
- Al più quattro filosofi possono iniziare a mangiare contemporaneamente, il quinto aspetta (usare un semaforo intero inizializzato a 4).
- Passaggio di un token (ma bisogna che partecipino anche i filosofi che stanno pensando).

Bisogna porre attenzione, oltre ai deadlock, anche al rischio di *starvation*: se non si forza in qualche modo la proprietà dell'attesa limitata, uno dei filosofi potrebbe sempre trovarsi in attesa di uno dei bastoncini e non arrivare mai a mangiare.

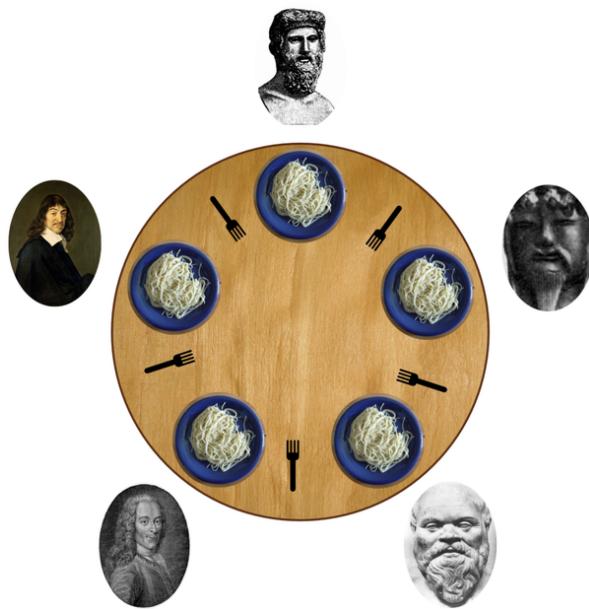


Figura 7.22: Il problema dei cinque filosofi [da Wikipedia]

7.3 Altre primitive di sincronizzazione

I semafori sono oggetti semplici e a basso livello; come abbiamo visto, un loro utilizzo errato può portare a deadlock, e può essere difficile verificare la correttezza di una politica di sincronizzazione che li utilizza.

7.3.1 I monitor

Sono state proposte soluzioni a più alto livello.

Un **monitor** è una classe i cui metodi (“entries”) sono accessibili esclusivamente (un solo processo/thread può trovarsi all’interno di un’entry del monitor in un determinato istante).

Inoltre, un monitor dispone di variabili “speciali”, dette *condizioni*, che operano in modo simile a un semaforo. Una condizione *x* dispone di due primitive, *wait* e *signal*: un processo che invoca *x.wait()* passa in stato Blocked e viene liberato solamente quando un altro processo invoca la corrispondente *x.signal()*.

Dopo l’invocazione di una *wait*, un processo non è più considerato nel monitor, e un altro processo può quindi accedere a un’entry.

Come esempio, vediamo in Fig. 7.23 l’uso di un monitor per regolare il problema produttore/consumatore. Il monitor utilizza due condizioni, *pieno* e *vuoto*. Il produttore attende sulla condizione *pieno* quando il buffer non ha posti liberi, e viene sbloccato da un consumatore che libera un posto: in un certo senso, *pieno.signal()* serve a falsificare la condizione, e sblocca il processo in attesa. Simmetricamente, il consumatore attende con *vuoto.wait()* se vede che non ci sono dati. Quando il produttore inserisce un dato, alla fine sblocca il consumatore in attesa.

Notiamo che l’atomicità delle operazioni di incremento e test è garantita dal fatto che all’interno del monitor non può essere attivo più di un processo: finché un’entry non finisce (o non resta bloccata da una *wait()*), nessun altro thread può agire sulle stesse variabili.

Un ultimo esempio, in Fig. 7.24, è l’implementazione di un semaforo binario per mezzo di un monitor. La condizione *attesa* serve a tenere in coda eventuali processi in attesa che il semaforo si liberi, mentre il fatto che il semaforo sia libero o occupato è codificato nella variabile booleana

Monitor:

```
monitor ProdCons {
    condition pieno, vuoto;
    int numero = 0; // inizializzazione
    entry inserisci(dato) {
        if (numero == DIMENSIONE_BUFFER)
            pieno.wait();
        metti il dato nel buffer;
        if (++numero == 1)
            vuoto.signal();
    }
    entry estrai(&dato) {
        if (numero == 0)
            vuoto.wait();
        preleva il dato dal buffer;
        if (numero-- == DIMENSIONE_BUFFER)
            pieno.signal();
    }
}
```

Processo produttore:

```
while (1) {
    crea il dato;
    ProdCons.inserisci(dato);
}
```

Processo consumatore:

```
while (1) {
    ProdCons.estrai(dato);
    utilizza il dato;
}
```

Figura 7.23: Produttore/consumatore con l'uso dei monitor.

```
monitor SemBin {
    condition attesa;
    bool occupato = FALSE; // inizializzazione
    entry WAIT() {
        // letteralmente: se il semaforo è occupato,
        // mettiti in attesa
        if (occupato)
            attesa.wait();
        occupato = TRUE;
    }
    entry SIGNAL() {
        // Non sono più occupato, libera un eventuale processo in attesa.
        occupato = FALSE;
        attesa.signal();
    }
}
```

Figura 7.24: Implementazione di un semaforo binario con un monitor.

Classe con metodi sincronizzati:

```
class ProdCons {
    Dato buffer = new Dato[DIMENSIONE_BUFFER];
    int numero = 0;
    public synchronized inserisci(Dato d) {
        while (numero == DIMENSIONE_BUFFER)
            wait();
        inserisci d in buffer
        numero++;
        notifyAll();
    }
    public synchronized Dato estrai() {
        while (numero == 0)
            wait();
        estrai dal buffer il Dato d
        numero--;
        notifyAll();
        return d;
    }
}
```

Oggetto condiviso fra i thread:

```
ProdCons pc;
```

Thread produttore:

```
while (1) {
    crea il dato;
    pc.inserisci(dato);
}
```

Thread consumatore:

```
while (1) {
    Dato dato = pc.estrai();
    utilizza il dato;
}
```

Figura 7.25: Produttore/consumatore in Java.

occupato. Di nuovo, l'atomicità delle modifiche e dei test è garantita che non più di un processo per volta può essere in esecuzione all'interno di un'entry del monitor.

Non molti linguaggi però offrono questi costrutti.

7.3.2 La parola chiave synchronized in Java

Un effetto simile ai monitor si ottiene in Java con l'attributo `synchronized`, che può essere associato a un metodo o a un blocco di codice, garantendo l'accesso esclusivo a livello di thread. Un thread può mettersi in attesa all'interno di un blocco sincronizzato invocando il metodo `wait()` in modo simile a un monitor. Il thread può essere risvegliato da una chiamata a `notifyAll()`.

Ecco in Fig. 7.25 un'implementazione parziale in Java del problema produttore/consumatore. I metodi sincronizzati di una stessa classe condividono un mutex (lock) associato all'istanza della classe, quindi non più di un thread può trovarsi in esecuzione in `inserisci()` o `estrai()`, garantendo l'atomicità degli incrementi e dei test. Se un produttore trova il buffer pieno o un consumatore lo trova vuoto, si mette in attesa chiamando `this.wait()` sul lock di istanza. Quando le cose cambiano (un nuovo dato è stato inserito o letto), una chiamata a `this.notifyAll()` sveglia tutti i thread che avevano invocato `this.wait()`. Il primo thread svegliato tornerà a controllare il valore di `numero` e deciderà di proseguire, mentre gli altri torneranno in `wait()`.

7.4 Analisi e prevenzione dei deadlock

Il deadlock è la condizione probabilmente più pericolosa, è il caso di spendere qualche parola in più.

Un deadlock si verifica quando due o più processi si tengono reciprocamente bloccati in attesa di risorse possedute dagli altri processi che partecipano al deadlock. Riassumendo, perché si verifichi un deadlock è necessario che:

- i processi utilizzino risorse che richiedono **accesso esclusivo** (quindi siano costretti ad attendere se queste sono utilizzate da altri);

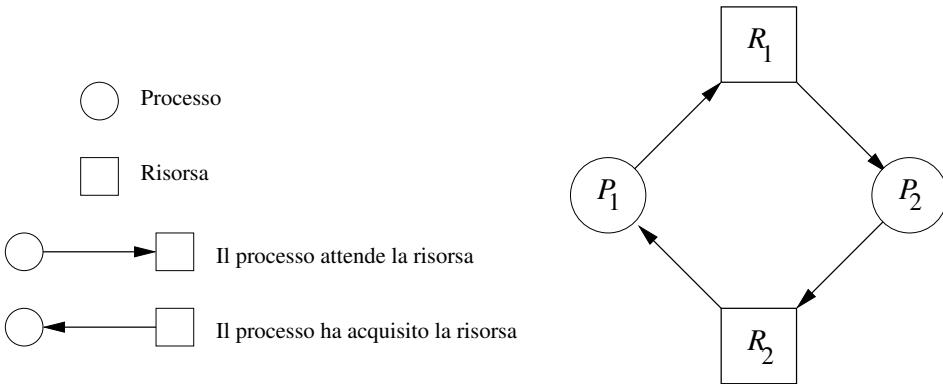


Figura 7.26: Una situazione di deadlock con due processi e due risorse: il processo P_1 attende la risorsa R_1 allocata a P_2 che è in attesa di R_2 allocata a P_1 .

- le risorse possano essere rilasciate solo volontariamente (non vi sia possibilità di “levarle” al processo che le detiene da parte del sistema operativo): questo è necessario se levare una risorsa, ad esempio un file, rischia di lasciarla in uno stato inconsistente;
- due o più processi si trovano in una condizione di **hold and wait**: detengono almeno una risorsa e ne stanno attendendo un’altra.
- esista un ordine di **attesa circolare** in cui il processo P_1 attende una risorsa R_1 posseduta dal processo P_2 che è in attesa della risorsa R_2 ... e questa sequenza si chiuda con una risorsa posseduta da P_1 .

Una situazione di deadlock è rappresentata da un ciclo nel **grafo di allocazione delle risorse** (resource allocation graph, RAG), dove i nodi rappresentano i processi e le risorse, mentre gli archi rappresentano l’attesa di una risorsa e il suo possesso. Fig. 7.26 rappresenta la situazione più semplice di deadlock, causata dal codice già visto in Fig. 7.21.

Possibili strategie per impedire i deadlock:

- impedire che possa verificarsi almeno una delle condizioni necessarie elencate sopra;
- rilevare il deadlock e ripristinare il sistema, ad esempio eliminando uno dei processi coinvolti (approccio tipicamente scelto dai DBMS);
- non fare nulla contando sul fatto che i deadlock sono rari.

7.4.1 Prevenzione

Alcune delle condizioni necessarie non sono negoziabili (ad esempio la mutua esclusione e la non prelazionabilità).

La condizione di “Hold and wait” può essere evitata in due modi:

- Le risorse si allocano solo all’inizio dell’esecuzione (problema: le risorse risultano sottoutilizzate, i processi potrebbero essere in esecuzione perenne);
- È possibile allocare una risorsa solo se non se ne possiedono altre (ma spesso serve usarne più d’una allo stesso tempo)

In generale, le strategie mirate a evitare la situazione di hold and wait causano un sottoutilizzo delle risorse e il rischio di starvation.

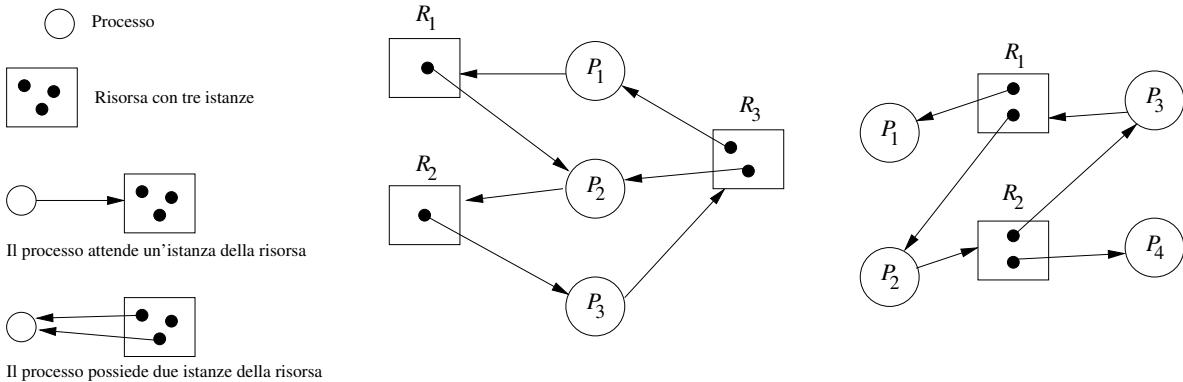


Figura 7.27: Generalizzazione: un RAG con risorse dotate di più istanze. Il grafo di sinistra rappresenta un deadlock, quello di destra no.

La condizione di attesa circolare può essere prevenuta, ad esempio, predisponendo un ordine preciso in cui le risorse possono essere richieste. Si stabilisce un “ordine di priorità” $R_1 > R_2 > \dots > R_n$; un processo può richiedere risorse solo in ordine crescente di priorità. Però gli ordini “naturali” di richiesta delle risorse dipendono dal loro uso, non sono sempre prevedibili a priori.

Il sistema potrebbe, ad ogni richiesta di risorse, analizzare il RAG per evitare cicli. Generalizziamo (Fig. 7.27) il grafo definito in precedenza al caso di risorse che dispongono di più istanze (gestibili, cioè, da semafori interi e non binari). Osserviamo che la presenza di un ciclo, in questo caso, è condizione *necessaria* al deadlock, ma non sufficiente. Ad esempio, l’allocazione di destra in Fig. 7.27 contiene un ciclo, ma non si tratta di un deadlock perché è sufficiente che P_1 o P_4 liberino la loro risorsa e gli altri due processi possono procedere.

7.4.2 Prevenzione dinamica

Definizione 1 (Stato di allocazione). *Lo stato di allocazione di un sistema consiste nella definizione di:*

- numero di istanze disponibili per ogni risorsa;
- numero di istanze allocate ad ogni processo.

Lo stato del sistema può essere descritto dal RAG. Supponiamo ora che un processo sia tenuto a dichiarare, in partenza, la quantità massima di risorse di ciascun tipo che esso richiederà durante la sua vita. Il SO può utilizzare questa informazione epr decidere se il sistema è in uno stato “sicuro” oppure no.

Definizione 2 (Stato sicuro). *Un sistema si trova in uno stato sicuro se, utilizzando le risorse a disposizione, può allocare risorse a ogni processo, in qualche ordine, in modo che ciascuno possa completare la propria esecuzione nel caso peggiore.*

In altri termini, uno satto è sicuro se esiste un ordine P_1, P_2, \dots, P_n tale che P_i può arrivare a completamento utilizzando le risorse libere e quelle detenute dai processi P_j con $i < j$.

Esempio Siano disponibili 12 istanze di una risorsa, con le seguenti allocazioni ai processi:

Processo	Richieste	Possedute
P_1	10	5
P_2	4	2
P_3	9	2

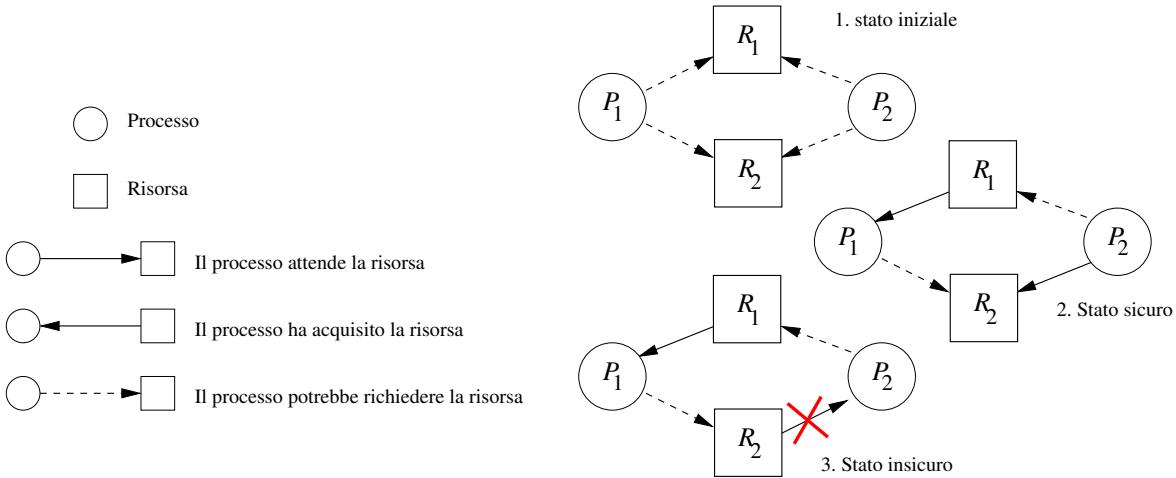


Figura 7.28: Estensione del RAG di Fig. 7.26 con archi di rivendicazione.

In questo momento, 9 istanze sono possedute dai processi e 3 sono libere. Il sistema si trova in uno stato sicuro, perché ci sono abbastanza istanze per soddisfare le richieste di P_2 , che può terminare. Al termine di P_2 le istanze libere diventano 5, il che permette di portare a completamento P_1 , dopodiché resteranno abbastanza risorse per completare P_3 . In altri termini, la sequenza P_2, P_1, P_3 garantisce che tutti i processi arriveranno al termine.

Se però allochiamo un'istanza in più a P_2 , lo stato non è più sicuro:

Processo	Richieste	Possedute
P_1	10	5
P_2	4	2
P_3	9	3

Anche se eseguiamo per primo P_2 , dopo il suo termine le 2 istanze liberate potrebbero non essere sufficienti a portare a termine né P_1 , né P_3 , che potrebbero entrare in un deadlock.

Prevenzione con RAG

Torniamo a considerare il caso in cui ogni risorsa è disponibile in una sola istanza (caso di semafori binari).

Siccome stiamo considerando il caso in cui ogni processo dichiara inizialmente le risorse che richiederà, possiamo estendere il RAG con archi di *rivendicazione*, trattigliati, che indicano il caso peggiore (il processo potrebbe, in futuro, richiedere la risorsa). Si veda Fig.7.28: quando un processo richiede una risorsa, il SO la concede solamente se così facendo non introduce cicli, considerando anche gli archi trattigliati. All'inizio (stato 1 in figura) non ci sono risorse allocate. In seguito (stato 2) P_1 ha chiesto e ottenuto la risorsa R_1 . Se in seguito P_2 richiede R_2 , il SO può constatare che, concedendola, lo stato non sarebbe più sicuro (si creerebbe un ciclo segnalando un futuro potenziale deadlock). Il processo P_2 , quindi, resterà in attesa finché qualche altra variazione dello stato non renderà sicura l'assegnazione.

L'algoritmo del banchiere

Se alcune risorse sono disponibili in più istanze, l'algoritmo precedente non funziona.

Premesse:

- di ogni risorsa R_j conosciamo il numero `available[j]` di istanze disponibili;

```

function stato_sicuro() { // m processi, n risorse
    int available1[m] = available[]; // copia il vettore available
    bool terminato[n] = {FALSE,...,FALSE}; // nessun processo è ancora terminato
    while (terminato[] != {TRUE,...,TRUE}) {
        // Esiste un processo che posso completare con le risorse a disposizione?
        int i;
        for (i = 0; i < n && (terminato[i] || need[i] > available1[]); i++);
        if (i == n) return FALSE
        else {
            terminato[i] = TRUE; // porta a termine il processo i
            available1[] += alloc[i][]; // recupera le risorse allocate
        }
    }
    return TRUE;
}

```

Figura 7.29: Algoritmo del banchiere, funzione di verifica della sicurezza di uno stato: simula l'evoluzione dello stato assumendo che ogni processo richieda il massimo dichiarato, cercando una successione di processi in grado di terminare.

- ogni processo P_j dichiara all'inizio dell'esecuzione il suo fabbisogno massimo $\max[i][j]$ per ogni risorsa R_j ;
- durante l'esecuzione, lo stato è rappresentato da una matrice $\text{alloc}[i][j]$ i cui elementi dicono quante istanza delle risorse R_j sono attualmente allocate al processo P_i ;
- dallo stato si ricava la matrice $\text{need}[i][j]=\max[i][j]-\text{alloc}[i][j]$ che rappresenta quante istanze di R_j potrebbero essere ulteriormente richieste dal processo P_i per poter proseguire.

L'algoritmo, abbastanza immediato, consiste di due funzioni: la prima, rappresentata in Fig. 7.29, rileva la sicurezza di uno stato simulando la conclusione dei processi nel caso peggiore; la seconda utilizza la prima per decidere se allocare subito le risorse richieste da un processo.

Come esempio, consideriamo 5 processi P_1, \dots, P_5 , e tre risorse R_1, R_2, R_3 rispettivamente disponibili in 10, 5 e 3 istanze. Supponiamo che, in un determinato istante, lo stato sia il seguente:

$$\text{available} = \begin{pmatrix} 3 & 3 & 2 \end{pmatrix}, \quad \text{allocation} = \begin{pmatrix} 0 & 1 & 0 \\ 2 & 0 & 0 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix}, \quad \max = \begin{pmatrix} 7 & 5 & 3 \\ 3 & 2 & 2 \\ 9 & 0 & 2 \\ 2 & 2 & 2 \\ 4 & 3 & 3 \end{pmatrix}, \quad \text{need} = \begin{pmatrix} 7 & 4 & 3 \\ 1 & 2 & 2 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}.$$

È facile verificare che il sistema è in uno stato sicuro. Ad esempio, la riga **need** del processo P_2 può essere soddisfatta dalle risorse disponibili (è minore di **available**); una volta liberate le risorse di P_2 , sarà possibile soddisfare P_4 , e così via. La sequenza P_2, P_4, P_5, P_3, P_1 porta a terminare tutti i processi senza mai sfornare sul numero di risorse di ciascun tipo.

Supponiamo però che P_2 richieda $(1, 0, 2)$ (un'istanza di R_1 e due di R_3). Il numero di istanze di ciascun tipo (array **available**) sarebbe sufficiente, ma prima di procedere dobbiamo verificare se lo

```

function assegna_risorse(int i, int richieste[m]) { // Richieste del processo i
    while (TRUE) {
        if (richieste[] > available[]) wait(); // le risorse non bastano
        // Simula l'assegnazione:
        available[] -= richieste[]; // Risorse disponibili
        alloc[i] [] += richieste[]; // Risorse allocate
        need[i] [] -= richieste[]; // possibili richieste ulteriori
        if (stato_sicuro()) {
            ASSEGNA LE RISORSE; break;
        } else { // Ripristino e attendo
            available[] += richieste[];
            alloc[i] [] -= richieste[];
            need[i] [] += richieste[];
            wait();
        }
    }
}

```

Figura 7.30: Algoritmo del banchiere, funzione di allocazione delle risorse richieste. Se l'assegnazione porta a uno stato insicuro, ripristina e attende.

stato raggiunto dopo l'assegnazione

$$\text{available} = \begin{pmatrix} 2 & 3 & 0 \end{pmatrix}, \quad \text{allocation} = \begin{pmatrix} 0 & 1 & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix}, \quad \text{need} = \begin{pmatrix} 7 & 4 & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}$$

è ancora sicuro. Possiamo verificare che la sequenza P_2, P_4, P_5, P_1, P_3 permette a tutti i processi di terminare nel caso peggiore.

Se ora P_1 richiede 2 istanze di R_2 , dopo l'assegnazione lo stato diverrebbe

$$\text{available} = \begin{pmatrix} 2 & 1 & 0 \end{pmatrix}, \quad \text{allocation} = \begin{pmatrix} 0 & \mathbf{3} & 0 \\ 3 & 0 & 2 \\ 3 & 0 & 2 \\ 2 & 1 & 1 \\ 0 & 0 & 2 \end{pmatrix}, \quad \text{need} = \begin{pmatrix} 7 & \mathbf{2} & 3 \\ 0 & 2 & 0 \\ 6 & 0 & 0 \\ 0 & 1 & 1 \\ 4 & 3 & 1 \end{pmatrix}$$

e possiamo verificare che le risorse rimanenti non basterebbero a nessun processo nel caso peggiore in cui venissero richieste in blocco, causando un deadlock.

Capitolo 8

Gestione della memoria

In questa parte esaminiamo le problematiche relative alla gestione della memoria:

- Allocazione ai processi;
- Protezione (come impedire ai processi di leggere/scrivere/eseguire fuori dalle aree loro assegnate);
- Condivisione (come realizzare la memoria condivisa per l'IPC);
- Gestione della memoria virtuale (come fingere di avere più memoria di quella effettivamente disponibile usando il disco).

8.1 Indirizzamento e binding

Per poter essere eseguito, un processo deve trovarsi in memoria (il ciclo di fetch delle istruzioni macchina da eseguire fa sempre riferimento a indirizzi di memoria, e la maggior parte delle istruzioni comporta la lettura/scrittura di dati in RAM).

Il passaggio da “programma” (normalmente memorizzato come file su disco) a “processo” (programma in esecuzione) comporta vari passi. A seconda del contesto, i riferimenti alla memoria possono assumere forme diverse. Con riferimento a Fig.8.1, gli indirizzi di memoria possono assumere forme molto diverse.

- **Codice sorgente** — i riferimenti alla memoria sono simbolici (nomi di funzioni, etichette di codice, nomi di variabile), oppure gestiti attraverso puntatori e riferimenti “opachi” (non ci interessa il loro contenuto).
- **Moduli oggetto, file eseguibili** — se la locazione di caricamento è nota a priori (sistemi non multiprocesso), possono essere anche gli indirizzi finali. Altrimenti (caso più comune, codice rilocabile) sono offset rispetto all'inizio del modulo.
- **Caricamento in memoria** — il loader sistema i riferimenti collegandoli a indirizzi fisici (“binding”). Spesso si fa ricorso a supporto hardware (MMU). Ugualmente, vanno risolti tutti i riferimenti a eventuali librerie condivise (non contenute nel programma se non per piccoli stub di raccordo).

Visto questo, durante l'esecuzione di un processo abbiamo due possibili scenari, come si vede in Fig.8.2:

- Binding statico: gli indirizzi sono stati fissati in fase di compilazione/link (riferimenti assoluti, codice non rilocabile) o in sede di caricamento (codice rilocabile): la CPU emette gli indirizzi fisici di memoria;

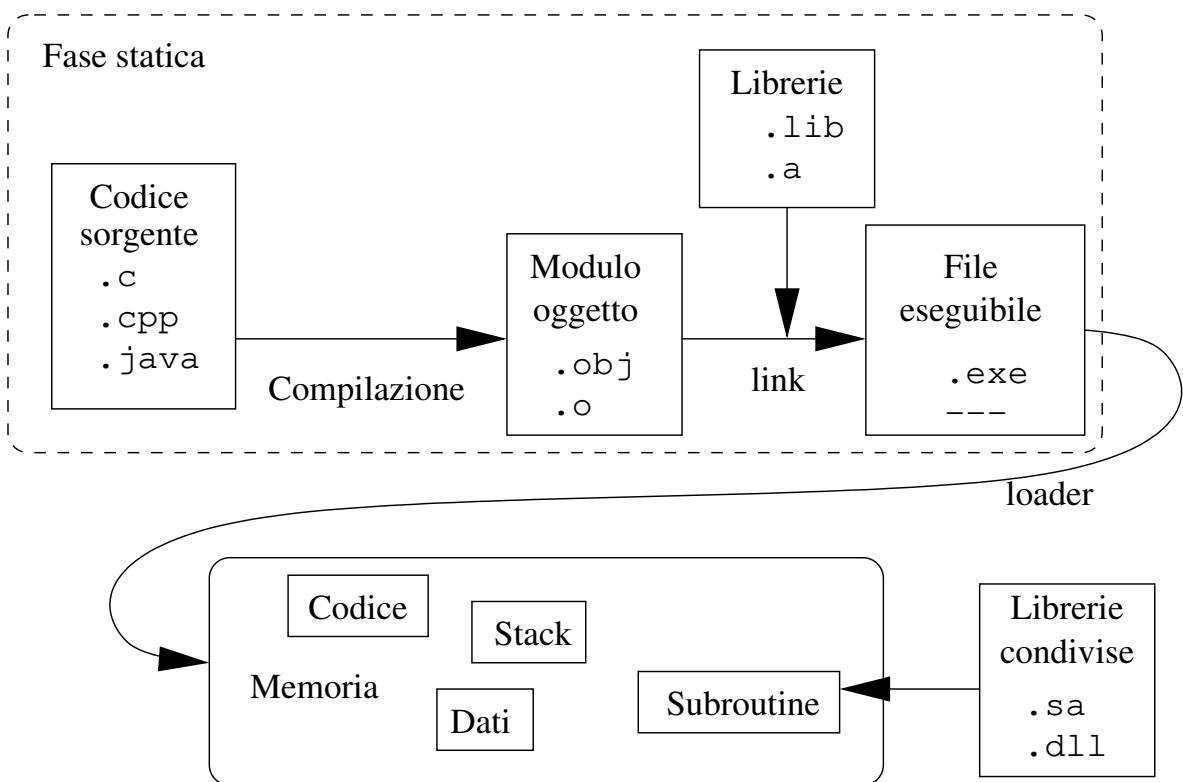
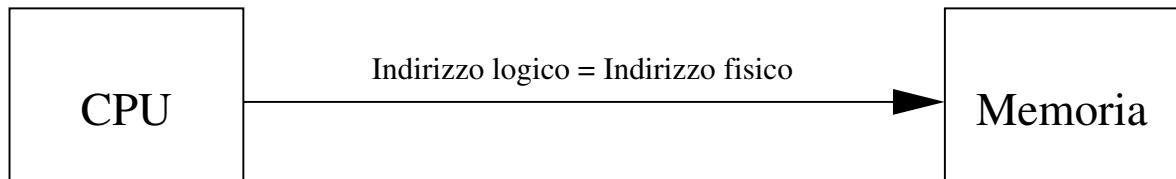


Figura 8.1: Fasi della trasformazione da codice sorgente a processo in esecuzione.

a. Binding statico



b. Binding dinamico

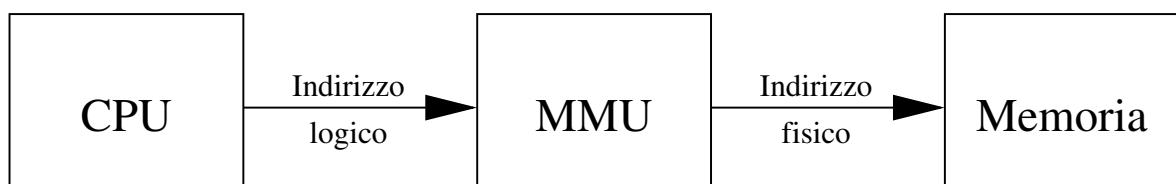


Figura 8.2: Binding statico e dinamico.

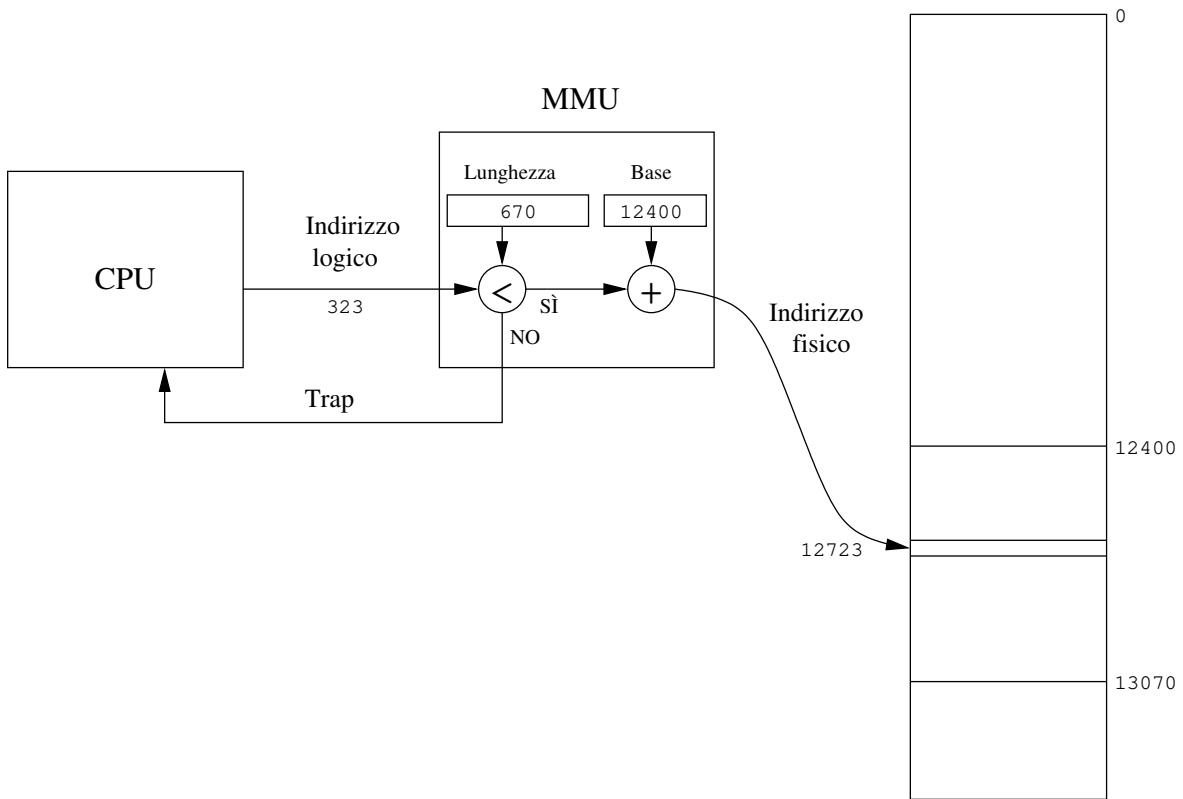


Figura 8.3: Binding dinamico: funzionamento di base della MMU.

- Binding dinamico: la CPU ha una visione della memoria basata su indirizzi “logici”; la traduzione degli indirizzi è demandata a hardware esterno (la Memory Management Unit, MMU) durante l'esecuzione.

Il funzionamento di base della MMU per il binding dinamico è illustrato in Fig. 8.3: nella MMU è precaricato un registro contenente l'indirizzo base della porzione di memoria allocata al processo. Gli indirizzi generati dalla CPU sono interpretati come offset rispetto a questo indirizzo base. Opzionalmente, un registro contenente la lunghezza della partizione di memoria può essere usato per prevenire l'accesso al di fuori dell'area di memoria allocata.

L'impostazione dei registri della MMU avviene all'avvio del processo e ad ogni cambiamento di contesto (i valori dei registri relativi a un processo sono memorizzati nel PCB al pari dei registri interni alla CPU).

Per il momento assumiamo che un processo vada caricato interamente in memoria prima dell'esecuzione.

8.2 Allocazione contigua

Un processo deve vedere la memoria ad esso assegnata come *contigua* (senza salti). Il modo più semplice è di allocare memoria solamente in partizioni contigue.

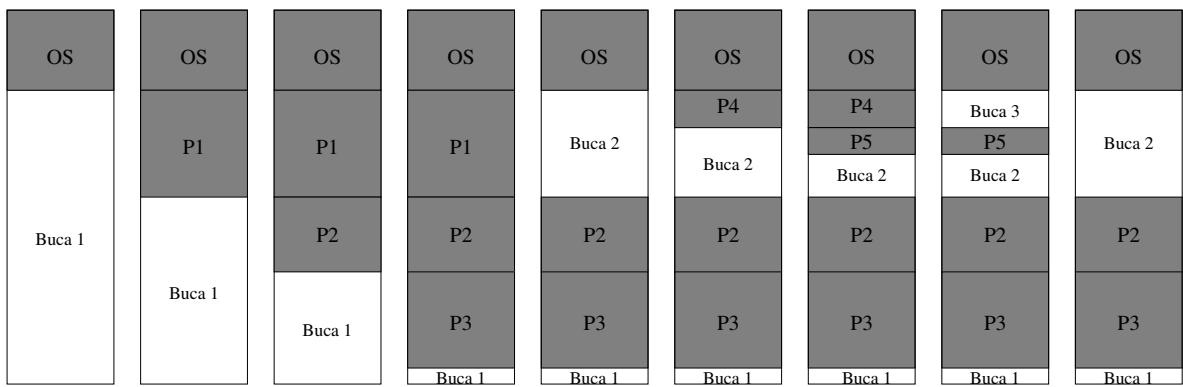


Figura 8.4: Allocazione a partizioni variabili: evoluzione delle buche nel tempo.

8.2.1 Partizioni fisse

La tecnica più semplice consiste nel suddividere a priori la memoria in partizioni di dimensioni predefinite (spesso diverse).

All'avvio, un processo “dichiara” la quantità di memoria di cui ha bisogno e viene caricato in una delle partizioni libere dallo scheduler a lungo termine. La scelta della partizione può seguire vari criteri (ad esempio: si sceglie la partizione più piccola in grado di contenere il processo, oppure la prima partizione nella lista).

Se non c'è posto, il processo viene messo in coda (con varie possibilità: una coda singola, una coda per partizione, coda FIFO o a priorità...) e viene estratto quando qualche processo temrina liberando la propria partizione.

Una volta scelta la partizione, questa viene contrassegnata come occupata e i registri della MMU vengono impostati.

A fronte di una relativa semplicità di implementazione, la tecnica delle partizioni fisse soffre di alcuni svantaggi:

- Il grado di multiprogrammazione è limitato dal numero di partizioni.
- Frammentazione (spreco di memoria):
 - **Frammentazione interna:** all'interno di una partizione c'è dello spazio non richiesto dal processo (a meno che non si trovi una partizione dell'esatta dimensione richiesta), ma non utilizzabile da altri.
 - **Frammentazione esterna:** Pur essendovi abbastanza memoria libera per soddisfare le esigenze di un processo, questa può trovarsi organizzata in partizioni troppo piccole e non contigue.

8.2.2 Partizioni variabili

La memoria non è suddivisa a priori. Ogni volta che un processo parte, si cerca dello spazio libero di dimensioni sufficienti e si alloca al processo la quantità di memoria richiesta.

La memoria è organizzata in un insieme di “buche” (*holes*, zone di memoria disponibili). Con riferimento a Fig. 8.4, inizialmente la memoria è una singola buca.

All'arrivo di un processo, si individua una buca in grado di contenere l'immagine; si alloca la quantità richiesta, e si riduce la buca alla parte ora libera.

Quando un processo termina, la sua memoria viene aggregata alle buche contigue (se ce ne sono), oppure confluiscce in una nuova buca.

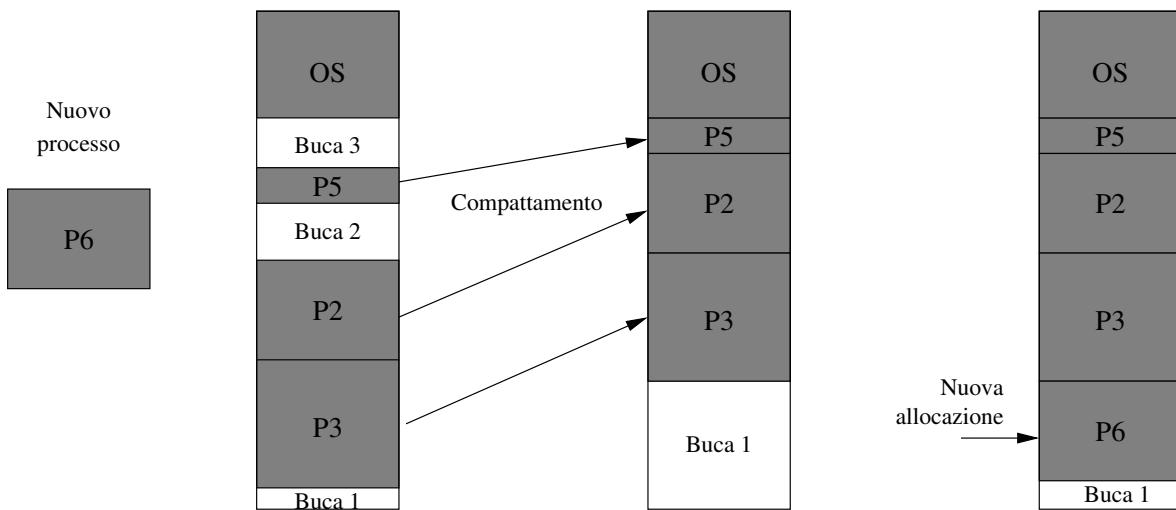


Figura 8.5: Compattamento: riduzione della frammentazione esterna per fare spazio per un nuovo processo.

Le buche sono tipicamente mantenute in una lista concatenata i cui elementi registrano l'indirizzo iniziale e la lunghezza.

La scelta della buca per un nuovo processo può seguire vari criteri:

- **Best-fit:** si sceglie la buca più piccola fra quelle in grado di contenere l'immagine del processo.
- **First-fit:** per non perdere tempo nella ricerca, si assegna la prima buca adatta trovata durante la ricerca.
- **Worst-fit:** si alloca sistematicamente la buca più grande. Per quanto controtuitiva, la tecnica permette di conservare buche relativamente grandi.

La tecnica elimina completamente la frammentazione interna: la memoria assegnata a un processo è esattamente quella richiesta, e il resto rimane disponibile.

Per contro, la frammentazione esterna può raggiungere livelli elevati (come si vede nella penultima fase di Fig. 8.4).

8.2.3 Compattamento

Una possibile soluzione alla frammentazione è il *compattamento* della memoria. Se un processo richiede memoria, e questa è disponibile ma non contigua, il SO può spostare i blocchi di memoria già allocati fino a renderli contigui, come in Fig. 8.5. Lo spostamento di aree di memoria già allocate può essere delicato, ma nel caso in esame si riduce a passaggi abbastanza semplici. Per spostare, ad esempio, l'area di P_5 :

1. Il SO mette il processo P_5 , attualmente in stato Waiting, oppure Blocked, in una coda di attesa specifica per la rilocazione;
2. Incarica il DMA di spostare il contenuto della memoria di P_5 alla nuova posizione; il DMA opera indipendentemente dalla CPU, utilizzando il bus nei momenti in cui gli altri dispositivi (CPU inclusa) non lo stanno usando; durante lo spostamento, la CPU riprende l'attività consueta.
3. Alla fine dello spostamento, il DMA invia un interrupt alla CPU (oppure la CPU effettua di tanto in tanto un polling dei registri DMA).

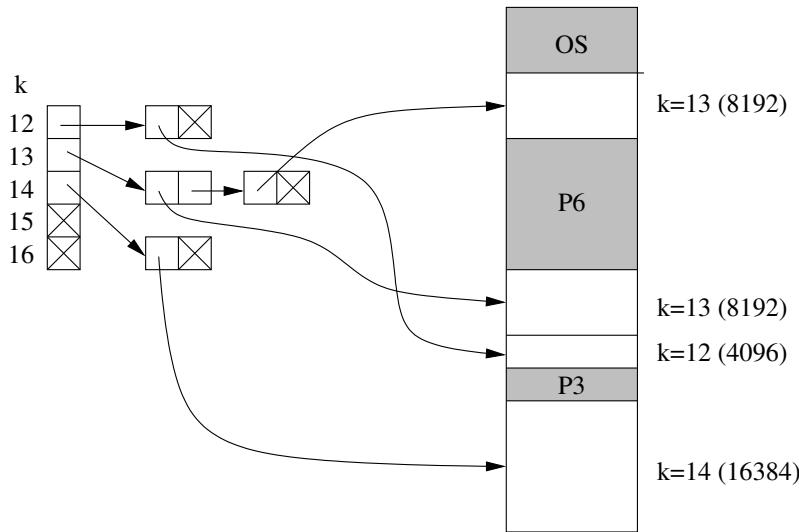


Figura 8.6: Buddy system: esempio con memoria da 64KB, $12 \leq k \leq 16$.

4. Il SO carica il nuovo indirizzo base della partizione di memoria nel PCB di P_5 .
5. Infine, rimette P_5 nella coda dalla quale l'aveva estratto; al prossimo burst CPU, il processo P_5 potrà tornare in esecuzione senza nessun problema, visto che gli indirizzi logici a cui fa riferimento la CPU non sono cambiati.

Naturalmente, la rilocazione richiede tempo, quindi non è necessariamente vantaggiosa rispetto alla semplice strategia di attendere finché non termina un altro processo liberando memoria.

8.2.4 Il Buddy system

Uno degli svantaggi della tecnica a partizioni variabili è la necessità di scorrere la lista delle buche per trovare la più adeguata a un processo.

Un'alternativa è considerare la memoria come una serie di liste di blocchi di dimensione 2^k , con $L \leq k \leq U$ (dove 2^U è tutta la memoria disponibile). All'inizio, l'unico blocco è l'intera memoria, 2^U .

Quando un processo con immagine di dimensione s va caricato in memoria, il SO esplora la lista corrispondente a k tale che $2^{k-1} < s \leq 2^k$ (con L e U come limiti inferiore e superiore) e viene utilizzato un elemento di quella lista. Se la lista k è vuota, si cerca la prima lista non vuota k' (con $k' > k$), e si procede per dimezzamenti successivi.

Quando un processo termina, la sua memoria di dimensione 2^k viene liberata e viene rimessa nella lista opportuna. Se un blocco adiacente ha la stessa dimensione 2^k , i due “buddies” vengono aggregati in un'unica area di dimensione 2^{k+1} .

Il principale vantaggio del sistema è la velocità con cui si può individuare un'area di dimensione adeguata (basta scorrere la sola lista della dimensione opportuna, eventualmente spezzando aree di dimensione maggiore).

La frammentazione esterna è ancora possibile, quella interna è limitata a metà della dimensione dei blocchi richiesti (significativa, ma potrebbe andare molto peggio).

8.3 Paginazione

Si tratta di una tecnica, molto diffusa, per eliminare la frammentazione esterna permettendo allo spazio di indirizzamento di non essere contiguo.

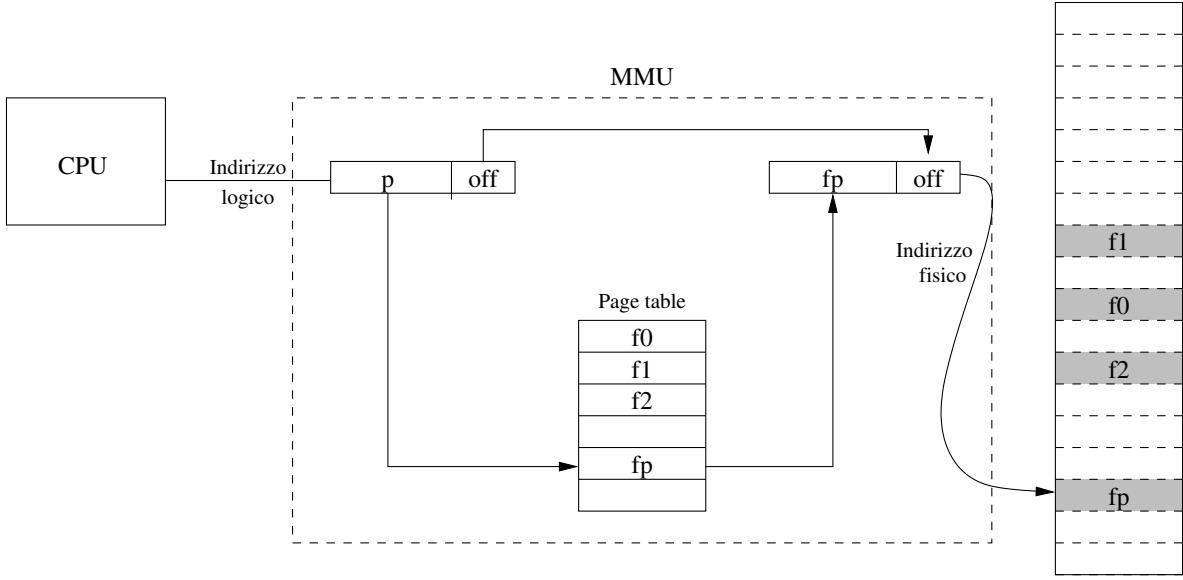


Figura 8.7: Paginazione, funzionamento di base: l'indirizzo logico contiene un numero di pagina (indice consecutivo) e un offset; i frame fisici non sono necessariamente consecutivi.

La memoria fisica viene divisa in molti blocchi, detti *frame*, tutti di uguali dimensioni (dell'ordine dei KB, tipicamente da 1 a 8, sempre una potenza di 2).

Se la dimensione dei frame è f e un processo richiede s bytes, vengono allocati $\lceil s/f \rceil$ frame, non necessariamente contigui. Gli indirizzi di questi frame vengono registrati nella *tabella delle pagine* (page table) associata al processo e gestita dalla MMU.

Dal punto di vista della CPU, i frame allocati sono visti come *pagine*, della stessa dimensione, numerate a partire da 0 (il numero della pagina corrisponde all'indice della tabella).

Come si vede in Fig. 8.7, l'indirizzo logico è suddiviso in due parti: i $\log_2 f$ bit meno significativi rappresentano un offset all'interno della pagina, mentre i bit più significativi rappresentano il numero di pagina, e sono usati come indice della page table.

Problema Se il processo richiede molta memoria, la tabella di traduzione della MMU può diventare molto grande. Con pagine da 4KB, anche uno spazio di indirizzamento da pochi MB richiede migliaia di pagine, quindi una tabella da migliaia di righe. Realizzare una tabella di simili dimensioni all'interno della MMU è proibitivo per almeno due motivi:

- registri hardware più veloci della memoria centrale sono costosi;
- ogni processo richiede la propria tabella, quindi il cambio di contesto costringe il SO a trascrivere tabelle da migliaia (se non più) di righe dal PCB alla MMU, aumentando il tempo di context switch, che noi vogliamo sia trascurabile.

Soluzione Mantenere la tabella in memoria, in uno o più frame allocati per questo scopo al processo. Nella MMU, un registro (Page Table Base Register, PTBR) fa riferimento al frame contenente la tabella (opzionalmente, un corrispondente Length Register PTLR ne registra la lunghezza). In questo modo, la MMU contiene un solo registro da impostare a ogni cambio di contesto. A ogni accesso in memoria da parte della CPU, la MMU effettua un accesso addizionale alla memoria per consultare la page table, trovare il numero di frame associato alla pagina logica richiesta dalla CPU, comporre l'indirizzo fisico corrispondente e finalmente effettuare l'accesso richiesto dalla CPU.

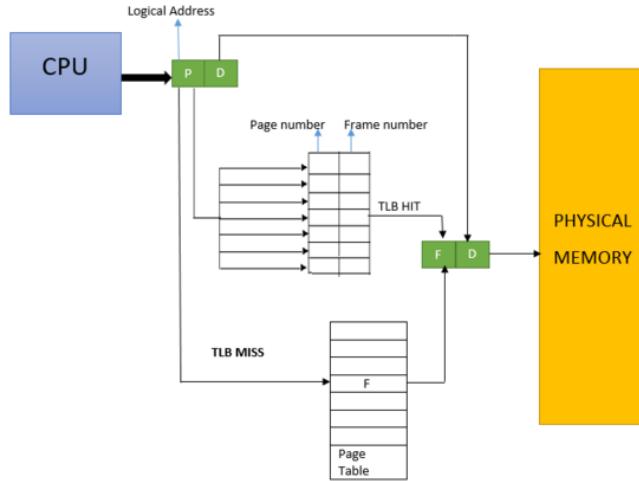


Figura 8.8: Paginazione, utilizzo di un TranslationLookaside Buffer come cache per una page table residente in memoria (da Wikipedia).

Nuovo problema Un accesso alla memoria da parte della CPU richiede ora *due* accessi alla memoria, raddoppiando il tempo richiesto per ogni singola operazione di lettura/scrittura.

Soluzione La MMU mantiene una tabella di dimensioni limitate, detta Translation Lookaside Buffer (TLB, vedi Fig. 8.8). Si tratta di una tabella associativa chiave/valore, contenente in ogni riga un numero di pagina (chiave) e il corrispondente indirizzo del frame (valore); ad ogni richiesta da parte della CPU, la MMU verifica se il numero di pagina è presente nel TLB. Se sì (“page hit”), accede al corrispondente frame; altrimenti (“page miss”), cerca il numero di frame attraverso un accesso in memoria e registra l’associazione pagina/frame nel TLB per usi successivi.

Il tempo di accesso effettivo alla memoria da parte della CPU dipende da alcuni fattori:

- il tempo t_{mem} per effettuare un’operazione di lettura/scrittura a un indirizzo di memoria;
- il tempo t_{tlb} necessario a trovare uno specifico numero di pagina nel TLB, se questo è presente;
- la cosiddetta *page hit ratio* α , la frequenza con cui una pagina richiesta si trova già nel TLB (in altri termini, la probabilità di una page hit).

Sulla base di queste definizioni, il tempo di accesso (medio) effettivo da parte della CPU si può calcolare come

$$EAT = \alpha(t_{tlb} + t_{mem}) + (1 - \alpha)(t_{tlb} + 2t_{mem}).$$

Consideriamo i tempi di accesso tipici in una macchina moderna, approssimati agli ordini di grandezza più vicini:

$$\begin{aligned} t_{mem} &= 100\text{ns} \\ t_{tlb} &= 10\text{ns} \\ \alpha &= 90\% \end{aligned}$$

Allora

$$EAT = 110\text{ns} \cdot 0,9 + 210\text{ns} \cdot 0,1 = 120\text{ns} = 1,2t_{mem}$$

Segue che, se il 90% degli accessi in memoria richiede soltanto l’accesso al TLB, il tempo totale di accesso è di circa il 20% maggiore rispetto al tempo richiesto da un accesso diretto.

8.3.1 Protezione e condivisione

La paginazione facilita la realizzazione di meccanismi di protezione della memoria. A ogni frame è possibile associare vari bit di protezione, ad esempio:

- valid/invalid: la pagina richiesta si trova o no nello spazio di indirizzamento logico del processo;
- read/write: il processo può soltanto leggere o anche scrivere nella pagina;
- execute: la pagina contiene codice eseguibile.

È anche possibile condividere uno stesso frame fra più processi, semplicemente riportando lo stesso indirizzo di frame in più page table diverse. Questo permette a più processi di condividere codice (più istanze dello stesso programma in esecuzione), librerie (condivise), dati (shared memory, visto parlando di IPC).

8.3.2 Dimensione dello spazio di indirizzamento: page table multilivello

Nelle architetture moderne (ad esempio a 64 bit) lo spazio di indirizzamento “virtuale” è enorme: 4GB richiedono un milione di frame da 4KB.

Per questo si rende necessario realizzare tabelle delle pagine a più livelli: l'indirizzo logico emesso dalla CPU viene interpretato come fosse composto da più indici di tabelle delle pagine, seguito da un offset. Ad esempio, se lo spazio di indirizzamento virtuale è a 32 bit e i frame sono da 4KB (offset da 12 bit), un frame può accogliere una page table da $2^{10} = 1024$ elementi (4 byte per frame). I 20 bit più significativi possono essere visti come una coppia di indici da 10 bit. Quindi un indirizzo logico a 32 bit potrebbe essere visto come una terna

$$(p_1, p_2, d)$$

La MMU usa $0 \leq p_1 < 2^{10}$ come indice nella page table di primo livello che punta al frame della page table di secondo livello. Poi usa $0 \leq p_2 < 2^{10}$ come indice in quest'ultima page table, ottenendo l'indirizzo effettivo del frame corrispondente alla pagina richiesta dalla CPU, a cui viene aggiunto l'offset.

Sono possibili più livelli di gerarchia (in pratica fino a 3). Ovviamente, l'uso di page table gerarchiche richiede un maggior numero di accessi in memoria in caso di page miss, aumentando l'EAT.

8.3.3 Page table invertita

Un'altra soluzione alla proliferazione delle page table a più livelli è l'uso di una singola page table “invertita” che mappa ciascun frame sui processi che li possiedono e alle pagine logiche cui sono associati. L'*inverted page table* è una tabella di sistema, con una riga per ogni frame di memoria fisica, contenente la coppia (pid, pagina) alla quale il frame è associato. Un indirizzo logico generato dalla CPU è la terna

$$(\text{pid}, \text{pagina}, \text{offset}).$$

Una tabella unica di sistema ha il vantaggio di non richiedere gestioni complesse e gerarchie multilivello, ma richiede (data la coppia (pid, pagina)) di individuare la riga per conoscere la posizione del frame fisico (vedi Fig. 8.9). Una ricerca sequenziale sarebbe proibitiva, considerato che la tabella ha potenzialmente milioni di righe, e che la ricerca andrebbe svolta ad ogni page miss. È però possibile utilizzare una struttura di supporto come una hash table per “invertire” a sua volta la tabella. Ovviamamente, una volta individuato il frame attraverso la inverted page table, la MMU utilizzerà il consueto meccanismo del TLB per gli accessi successivi.

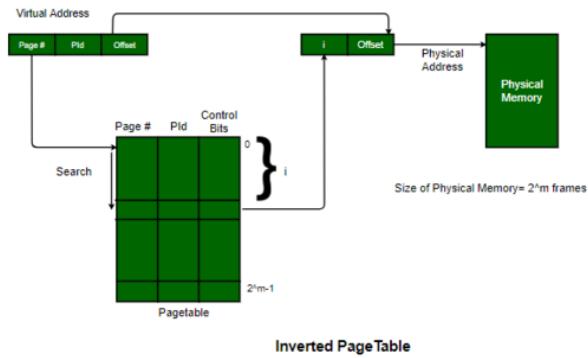


Figura 8.9: Uso dell’Inverted Page Table per individuare il frame fisico associato a una specifica pagina logica di un processo.

8.4 Segmentazione

La segmentazione cerca di mantenere la corrispondenza fra l’allocazione della memoria e la visuale che l’applicazione ha della stessa.

Un processo è visto come una collezione di segmenti in memoria, ciascuno dei quali ha uno scopo specifico:

- Codice principale del processo;
- altre funzioni e procedure;
- dati
- variabili
- stack
- codice di librerie condivise

L’indirizzo logico generato dalla CPU è una coppia (numero di segmento, offset); *in memoria si mantiene una tabella chiamata Segment Table* (vedi fig. 8.10).

Come per la paginazione, la MMU mantiene un Segment Table Base Register (STBR) e un Length Register (STLR), contenuti nel PCB e caricati a ogni cambio di contesto; il TLB è usato per le entry più comuni.

Anche la segmentazione consente di implementare facilmente la protezione degli accessi e la condivisione.

Confronto:

- **Paginazione**

- Vantaggi:
 - * Nessuna frammentazione esterna
 - * L’allocazione dei frame è libera, non richiede attenzioni particolari
 - * Protezione/condivisione
- Svantaggi:
 - * Frammentazione interna (minima)
 - * La vista utente e la struttura fisica sono completamente diverse.

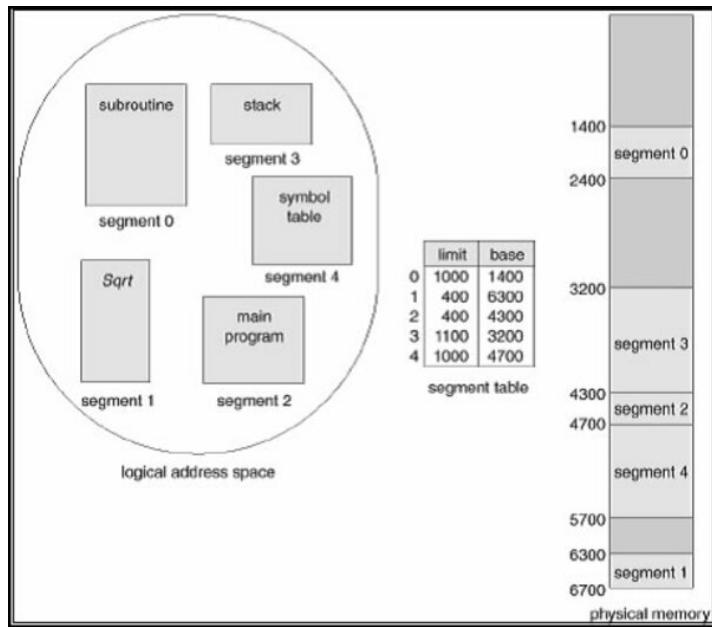


Figura 8.10: Segmentazione e uso di una tabella dei segmenti (dal Silberschatz-Gavin-gagne).

- **Segmentazione**

- Vantaggi:
 - * Consistenza fra vista utente e struttura fisica della memoria
 - * Protezione/condivisione
- Svantaggi:
 - * Sono necessari algoritmi di allocazione dinamica
 - * Frammentazione esterna

8.5 Memoria virtuale

Concetto fondamentale: la memoria principale è limitata, ma non tutti i dati e il codice del processo in esecuzione vanno caricati in memoria allo stesso tempo.

Idea (conseguenza della paginazione): lo spazio degli indirizzi logici (CPU) può essere più grande di quello degli indirizzi fisici (RAM disponibile). all'occorrenza si scambiano ("swap") pagine fra la memoria e il disco.

8.5.1 Paginazione su richiesta (demand paging)

Principio: una pagina viene caricata in memoria solo se necessario, altrimenti i suoi dati sono mantenuti nella memoria di massa. In questo modo, più processi possono risiedere in memoria perché richiedono meno spazio.

A ogni pagina logica (riga della page table) è associato un bit di validità che dice se la pagina risiede in memoria al frame indicato, oppure va caricata. Il bit è consultato durante la traduzione degli indirizzi logici forniti dalla CPU. Se una pagina richiesta ha il bit di validità posto a 0, la MMU genera un interrupt verso il SO (**page fault**); in risposta, il SO (vedi Fig. 8.11):

1. mette il processo in stato blocked su una coda di attesa della pagina richiesta;

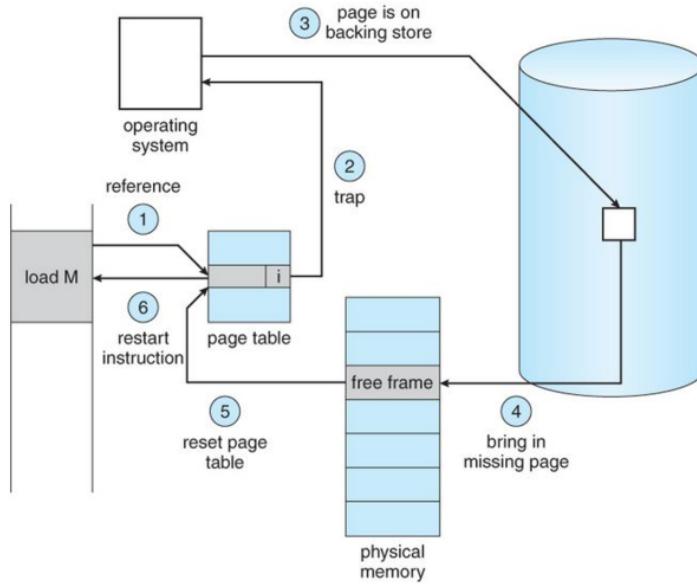


Figura 8.11: Paginazione su richiesta (dal Silberschatz-Gavin-gagne).

2. cerca un frame vuoto in cui caricare la pagina;
3. copia la pagina dal disco nel frame selezionato (questo passo può richiedere tempo; intanto, altri processi vanno in esecuzione);
4. aggiorna la page table;
5. rimette il processo in stato waiting e ripristina l'esecuzione ripetendo l'istruzione che ha causato il page fault.

Se all'avvio di un processo nessuna delle sue pagine è caricata in memoria, si parla di **demand paging puro** e la prima istruzione del processo risulta sempre in un page fault.

Notare che il tempo di caricamento di un frame da disco è costoso (dell'ordine dei millisecondi), quindi si desidera tenere basso il *page fault ratio*. Come stima dei tempi di accesso alla memoria si può fare un discorso simile a quello fatto per i page miss. Siano dati:

- il page fault ratio $0 \leq p \leq 1$;
- il tempo di accesso alla memoria centrale t_{mem} ;
- il tempo di caricamento di un frame da disco t_{pf} .

Allora:

$$EAT = (1 - p)t_{mem} * p t_{pf}.$$

Ad esempio, se $t_{mem} = 100\text{ns}$ e $t_{pf} = 1\text{ms}$, allora

$$EAT = [(1 - p) \cdot 100 + p \cdot 10^6]\text{ns} = 100 \cdot (1 + 9999p)\text{ns}.$$

Segue che per mantenere il peggioramento del tempo di accesso entro il 10% è necessario mantenere $p \leq 10^{-4}$ (un page fault ogni diecimila accessi).

8.5.2 Sostituzione delle pagine

Se non ci sono frame liberi in memoria, è necessario “sacrificare” un frame attualmente presente spostandolo su disco per fare spazio alla pagina richiesta.

La scelta del frame da spostare su disco deve sempre tenere presente l'obiettivo di minimizzazione del page fault ratio.

Osservazione: la necessità di liberare un frame spostandolo su disco prima di caricare quello richiesto comporta un raddoppio del tempo di page fault (bisogna prima scrivere su disco, poi leggere). Per ottimizzare, la page table può contenere un bit “dirty”, posto a zero quando la pagina viene caricata, a modificato in 1 solo se la pagina viene scritta. In questo modo, se un frame viene sacrificato e il suo dirty bit è ancora zero, sappiamo che non è stato modificato rispetto alla sua immagine su disco, e non richiede di essere scritto.

Fra i vari criteri di scelta del frame da sacrificare, possiamo citare:

FIFO (round-robin)

Si rimuove la pagina presente da più tempo, quindi tutti i frame sono rimpiazzati a rotazione. Si tratta di un algoritmo “cieco” che non valuta l'importanza di un frame. Per questo motivo è lontano dall'ottimalità, e soffre della cosiddetta “Anomalia di Bélády”.

Anomalia di Bélády In determinate condizioni, l'algoritmo FIFO può generare più page fault quando ha a disposizione più frame. Come esempio, considerare la seguente sequenza di numeri di pagina:

1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

e verificare come l'algoritmo FIFO la gestisce quando ha a disposizione 3 frame. Ripetere con 4 frame, scoprendo che vengono generati più page fault nonostante si abbia più memoria a disposizione.

Least Recently Used (LRU)

Si rimuove la pagina che non viene usata da più tempo (si assume che il passato sia un'indicazione attendibile di cosa avverrà in futuro).

So osservi che l'algoritmo ideale sarebbe quello che sacrifica il frame che non sarà usato per più tempo nel futuro, ma non possiamo sapere quale sia.

Implementare LRU non è banale, richiede di mantenere aggiornate informazioni a ogni accesso in memoria, e questo può essere proibitivo.

Uso del timer di sistema A ogni frame è associata una variabile (“contatore”) nella quale a ogni accesso viene copiato il valore del timer di sistema. Per scegliere il frame da riutilizzare, si cerca quello con il valore minore del contatore (richiede tempo $O(n)$ se n è il numero dei frame).

Uso di uno stack Si mantiene uno stack di numeri di frame. A ogni accesso, il frame coinvolto viene portato in cima allo stack. Il frame LRU è quello in fondo allo stack. La ricerca del frame da rimpiazzare diventa $O(1)$, ma serve mantenere uno stack (con supporto hardware, se non si vuole sprecare troppo tempo CPU).

Uso di un bit di riferimento Si mantiene un solo bit, posto a zero al caricamento della pagina. Dopo un successivo accesso, il bit viene posto a 1. Per il rimpiazzo, si cerca un frame con il bit di riferimento ancora a zero. Si tratta, ovviamente, di un'approssimazione molto cruda.

Uso di registri a scorrimento Come sopra, ma per ogni frame si mantiene un'intera parola posta a zero al caricamento. A ogni accesso, il bit più significativo viene posto a 1. Periodicamente, la parola viene fatta scorrere verso destra (quindi il suo valore diminuisce). Il frame LRU si troverà tipicamente fra quelli con la parola di valore più basso (ma nuovamente è necessario cercarlo).

Approssimazione Least Frequently Used (LFU) Si mantiene un contatore del numero di riferimenti, incrementato a ogni accesso, e si rimpiazza il frame con il contatore più basso. Si rischia però di sacrificare pagine giovani, che non hanno avuto il tempo di maturare molti accessi.

Approssimazione Most Frequently Used (MFU) Per ovviare, si può sostituire il frame con più accessi, perché si può supporre che sia in memoria da tanto tempo.

Rimpiazzamento “Second Chance” Si usa un bit di riferimento per ogni frame, posto a 1 al caricamento e a ogni accesso. Di base si usa il metodo FIFO (circolare) per la sostituzione, ma se il bit è ancora 1 lo si pone a zero e si passa al frame successivo, offrendo una seconda possibilità.

8.5.3 Allocazione dei frame ai processi

Finora abbiamo considerato la memoria come un unico pool di frame, ma i frame appartengono a processi diversi, ed è possibile migliorare le prestazioni del sistema preallocando un certo numero di frame a ogni processo.

Un processo ha bisogno di un numero minimo di frame per poter essere eseguito (pensare a un’istruzione con due operandi in memoria: sia l’istruzione che i due operandi possono essere a cavallo di due frame, quindi in tutto può generare sei page fault: il processo deve avere a disposizione almeno sei frame, altrimenti non può procedere).

Come al solito, possiamo considerare due schemi di allocazione:

- Fissa: un processo riceve un numero di frame e lo mantiene fino alla fine.
- Variabile: il numero di frame associati può variare durante l’esecuzione.

In caso di page fault, possiamo scegliere in quale contesto selezionare il frame da sostituire:

- Rimpiazzamento locale: il frame vittima deve appartenere allo stesso processo;
- Rimpiazzamento globale: si scaglia la vittima fra tutti i frame (salvo quelli eventualmente bloccati, p.es. di sistema).

Chiaramente, lo schema fisso ha senso solo in un contesto di rimpiazzamento locale.

Allocazione fissa

Dati m frame e n processi, è possibile dividere i frame in parti uguali allocando $\lfloor m/n \rfloor$ frame per ogni processo, oppure allocarli proporzionalmente a un parametro associato al processo (dimensione, priorità).

In questo caso, se il processo P_i ha dimensione (o priorità) s_i , allora riceve $\lfloor ms_i/S \rfloor$ frame, dove $S = \sum_i s_i$.

Allocazione variabile

Si modifica l’allocazione sulla base di qualche criterio:

- Calcolo della working set size;
- Calcolo della page fault frequency.

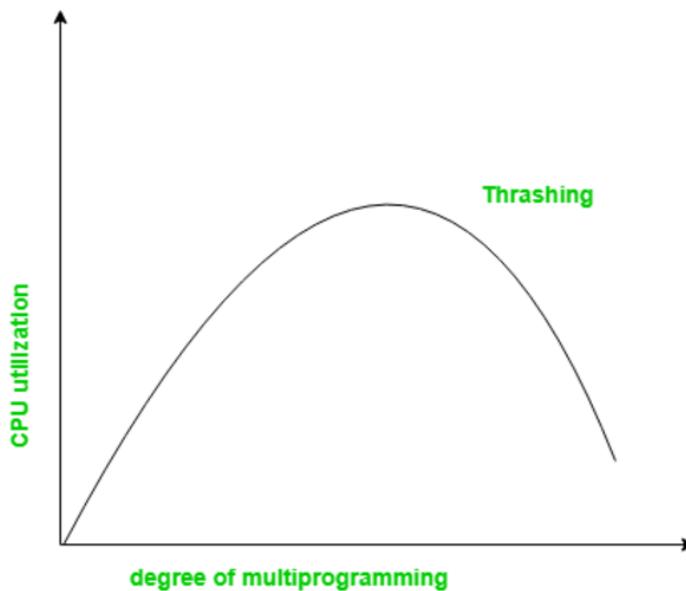


Figura 8.12: Thrashing (da geeksforgeeks.org).

Working set size Ci si basa sul concetto di *località* di un processo: in un certo periodo di tempo, gli accessi in memoria di un processo si localizzano su poche pagine alla volta. Fissato un periodo di tempo Δ , sia $WS_i(t, \delta)$ l'insieme delle pagine visitate dal processo P_i nell'intervallo di tempo $[t - \Delta, t]$. La *working set size* del processo è la dimensione del suo working set:

$$WSS_i(t, \Delta) = |WS_i(t, \Delta)|.$$

È importante scegliere un valore di Δ significativo. Troppo piccolo, non individua correttamente la località di un processo; troppo grande, rischia di coprire varie località ($\Delta \rightarrow \infty$ copre tutte le pagine del processo).

Per misurare il working set, si possono usare dei bit di riferimento associati alle pagine. Periodicamente, il SO li imposta a zero; a ogni accesso alla pagina, il suo bit viene posto a 1. Alla fine del periodo, si contano le pagine con bit posto a 1.

Thrashing Il numero totale di frame richiesti dai processi in esecuzione nel periodo di tempo Δ è $D = \sum_i WSS_i$. Se D è maggiore del numero di frame disponibili, allora il tasso di page fault aumenta. di conseguenza:

- l'utilizzo della CPU scende perché quasi tutti i processi sono in attesa di un frame;
- lo scheduler a lungo termine, vedendo la CPU sottoutilizzata, aumenta il grado di multiprogrammazione aggiungendo processi;
- i nuovi processi rubano ancora più frame.

Il throughput precipita, e i processi avanzano molto lentamente (Fig. 8.12).

Controllo della frequenza dei page fault Una soluzione più stabile consiste nello stabilire due soglie per la page fault ratio: quella superiore rappresenta un tasso inaccettabile e il numero di frame allocati al processo va aumentato; quella inferiore rappresenta un tasso inferiore al necessario (in cui i pochi page fault non hanno influenza sul tempo effettivo di accesso alla memoria), segno che un

processo ha ricevuto più frame del necessario. L'allocazione dei frame cerca di mantenere ogni processo fra queste due soglie.

Naturalmente, per evitare un eccesso di page fault è importante che gli algoritmi rispettino il più possibile una qualche misura di località degli attrezzi. Ad esempio, se una grande matrice è memorizzata per righe, l'accesso per colonne causerà numerosi page fault successivi, perché molti accessi successivi andranno su pagine diverse.

Capitolo 9

Filesystem e memoria di massa

Il filesystem serve a organizzare dati e programmi in strutture persistenti (file, cartelle/directory). Discuteremo tre aspetti:

- l'interfaccia verso le applicazioni: quali funzioni vengono esposte dal SO verso i processi;
- alcune scelte di implementazione;
- considerazioni sui dispositivi fisici e su come i dettagli fisici influiscono sulle prestazioni.

9.1 L'interfaccia del file system

9.1.1 I file

Un file rappresenta un insieme contiguo di dati identificato da un nome. I dati contenuti possono essere di vari tipi (testo, multimedia, documenti, programmi...).

Attributi dei file

Oltre ai dati, dunque, a un file è associato un insieme di attributi che, a seconda del SO, possono consistere in:

- Il nome (o i nomi), solitamente in un formato umanamente leggibile;
- Il tipo; per quanto molti SO siano indifferenti al tipo di dati contenuti, lasciando all'applicazione il compito di interpretarli, in alcuni casi l'informazione di tipo determina la struttura con cui il file è memorizzato nel supporto fisico;
- la posizione: dove si trovano fisicamente i dati nel supporto fisico;
- la dimensione
- informazioni relative alla protezione: chi può leggere, scrivere, eseguire il contenuto;
- marche temporali: creazione, ultima modifica, ultimo accesso...

Operazioni sui file

Su un file si devono poter eseguire alcune operazioni di base, tipicamente operate per mezzo di chiamate di sistema:

- Apertura: ricerca delle informazioni su un file o creazione delle informazioni associate a un nuovo file; impostazione delle strutture in memoria per la successiva gestione;
- Chiusura: esecuzione delle ultime operazioni di modifica eventualmente rimaste in sospeso, liberazione delle strutture in memoria allocate all'apertura;
- Scrittura: aggiunta di dati in coda al file, o sostituzione di dati già esistenti (riscrittura, “rewrite”);
- Lettura: trasferimento di alcuni dati del file in un'area di memoria principale dove il processo potrà leggerli;
- Riposizionamento: indicazione della prossima posizione di lettura/scrittura;
- Troncamento: mantenimento degli attributi, ma rimozione del contenuto;
- Cancellazione: rimozione del file dal sistema.

Metodi di accesso ai file

Non tutte le operazioni sono sempre possibili. In particolare:

- Un file può essere aperto in sola lettura, oppure un processo può non avere diritti di scrittura; in tal caso, le operazioni di scrittura, cancellazione e troncamento non sono possibili.
- Alcuni supporti (ad esempio i nastri) sono intrinsecamente sequenziali. In tal caso, l'operazione di riposizionamento può non essere disponibile, oppure può essere ridotta al solo spostamento all'inizio del file (“rewind”); inoltre, la scrittura può avvenire solo in coda (“append”), mentre non è permesso il rewrite, che potrebbe corrompere i dati che seguono.
- Altri supporti consentono l'accesso “casuale” (arbitrario) a ogni posizione.

Strutture dati in memoria

Come detto, all'apertura di un file il SO ne mantiene lo stato in alcune strutture in memoria.

- Una tabella di sistema in cui, per ogni file, si tiene traccia dei suoi attributi, della posizione su disco, del numero di riferimenti (aperture in corso) da parte dei processi.
- Una tabella per ogni processo contenente un riferimento alla corrispondente voce nella tabella di sistema e alcune informazioni specifiche, ad esempio la posizione corrente di lettura/scrittura. Più processi potrebbero dover accedere contemporaneamente allo stesso file in posizioni diverse, e anche uno stesso processo potrebbe dover mantenere due diverse posizioni aprendo due volte il file (in lettura).

Il conteggio dei riferimenti nella tabella di sistema serve a sapere se ci sono processi interessati al file oppure se, dopo la chiusura da parte di un processo, il file non è più in uso.

9.1.2 Le directory

Una directory è una collezione di nodi contenenti informazioni sui file (o su altre directory), e risiede anch'essa sul supporto di massa. Di norma, ciascun nodo contiene gli attributi elencati in 9.1.1.

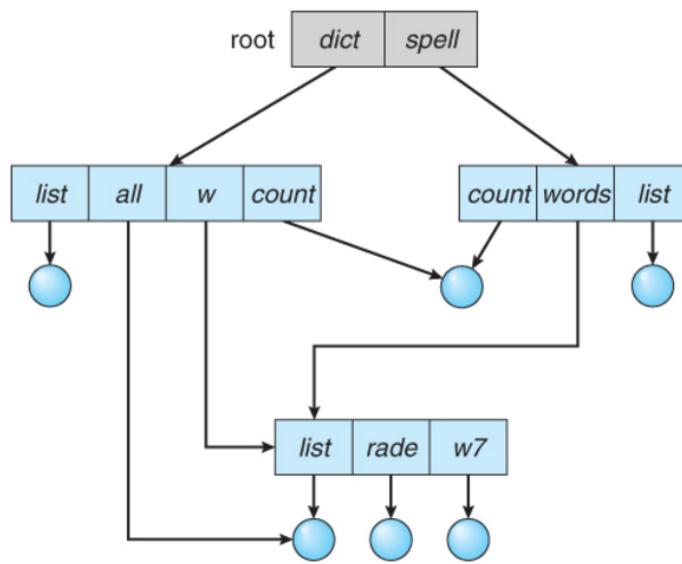


Figura 9.1: Struttura delle directory ad albero aciclico (dal Silberschatz).

Operazioni sulle directory

Le directory devono permettere le seguenti operazioni:

- aggiungere un file (in seguito a creazione o a spostamento da un'altra directory);
- rimuovere un file;
- cambiare gli attributi di un file (ad esempio il nome);
- elencare i file contenuti, cercare un file.

Non tutte queste operazioni saranno sempre possibili, a seconda dei permessi del processo che le esegue.

Struttura delle directory

SO diversi organizzano in diversi modi le directory, in base a criteri di efficienza (rapidità di accesso a un file), a seconda del supporto fisico su cui sono realizzate.

Ad esempio, i sistemi monoutente basati su floppy disk degli anni '70/'80 (CP/M, MS-DOS 1.0) associano a ogni dispositivo fisico una sola directory “piatta”. Altri sistemi (ad esempio Windows) utilizzano strutture ad albero, nelle quali le directory possono essere nodi di altre directory. UNIX aggiunge a questa la possibilità fare apparire lo stesso file come nodo di più directory (hard link), fintantoché il grafo resta aciclico come in Fig. 9.1 (un grafo generale può causare problemi nell’attraversamento). Per strutture più complesse, sia UNIX che Windows prevedono l’uso di link simbolici o “soft”.

Le directory a grafo aciclico hanno molte proprietà utili:

- Utenti diversi possono usare file con gli stessi nomi, semplicemente piazzandoli in directory diverse (metafora della directory come cartella, “folder”).
- Un processo può avere una sua cartella di default su cui operare (“working directory”), con la possibilità di riferirsi ai file con percorsi assoluti (a partire dalla radice) o relativi (a partire dalla posizione corrente);

- File con scopi logici simili possono essere raggruppati in cartelle tematiche (ad esempio `/var/log` sotto UNIX).

Uso di diversi filesystem

Tipicamente, ogni “volume” (dispositivo fisico o sua partizione logica, o raggruppamento RAID di dispositivi fisici in un solo dispositivo logico) ha una sua struttura a directory.

Per poter essere usato, il suo filesystem deve essere “montato”. Sotto Windows, ogni volume ha un proprio nome (di dispositivo fisico, ad esempio C:, oppure volume di rete \\servername\); sotto UNIX, un filesystem ha un proprio punto di montaggio corrispondente a una cartella vuota del filesystem già esistente (ad esempio /mnt/).

I filesystem possono essere condivisi tra più macchine, con opportune precauzioni e protezioni.

Protezione

Il possessore di un file deve poter definire cosa sia possibile farci, e da parte di chi. I permessi tipici riguardano la lettura, la scrittura e l'esecuzione; altri possono essere l'aggiunta in coda, la cancellazione...

I permessi possono essere definiti in due modi:

- Liste d'accesso (“Access Control List”) che definiscono per ogni file o directory chi può fare cosa. Complesse da gestire.
- UNIX raggruppa gli utenti in tre classi rispetto a un file: il proprietario, il gruppo e tutti gli altri. Per ciascuna classe è possibile attivare o disattivare i permessi di lettura, scrittura ed esecuzione.

9.2 Realizzazione

Un filesystem è normalmente realizzato da una gerarchia di strati. A partire dal dispositivo fisico, e per livelli di astrazione crescenti, abbiamo:

- Il dispositivo fisico; normalmente si tratta di un dispositivo a blocchi: l'unità minima di lettura/scrittura è molto più grande del singolo byte. Nei dischi rigidi abbiamo blocchi tipici dell'ordine dei KB (tipicamente 4KB) detti *cluster*. Nel seguito, useremo il termine “blocco” per riferirci a questa quantità. Un blocco è tipicamente identificato da una o più coordinate fisiche all'interno del dispositivo.
- Il controllore di I/O incaricato di comandare il dispositivo, organizzare i trasferimenti DMA e generare gli interrupt per il SO.
- Il filesystem di base, che si occupa di astrarre la lettura/scrittura a livello di blocchi.
- Il modulo di organizzazione dei file, che traduce una posizione all'interno di un file logico nel blocco corrispondente a cui i livelli inferiori devono accedere.
- Il file system logico, che organizza i livelli inferiori in termini di directory e file.
- I processi che utilizzano il file system.

Ciascun livello espone funzionalità verso il livello più astratto, e le traduce in chiamate alle funzionalità di livello meno astratto.

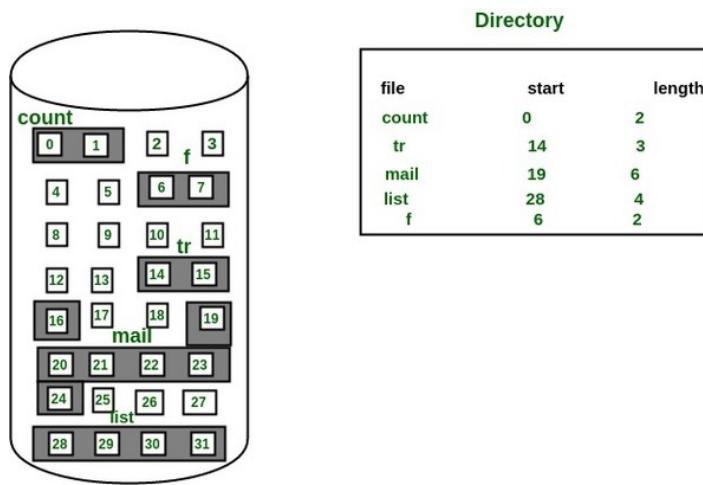


Figura 9.2: Allocazione contigua dei blocchi (da geeksforgeeks.org).

9.2.1 Strutture dati

Sul disco

Un volume fisico contiene alcune strutture dati:

- Il blocco di avvio (boot), contenente il codice da caricare per avviare il SO, se questo è presente nel volume;
- il blocco di controllo delle partizioni, che identifica i volumi logici nei quali il dispositivo fisico è suddiviso;
- Le strutture che contengono le directory e i descrittori con gli attributi dei file (che vedremo in seguito)

In memoria

Alcune di queste strutture sono replicate in memoria per velocizzare il riferimento ad esse:

- La tabella delle partizioni montate;
- Le directory a cui si accede più di frequente;
- le tabelle già viste (di sistema e di processo) dei file aperti.

9.2.2 Allocazione dello spazio su disco

Come già visto per la memoria centrale, anche per il disco è necessario definire delle strategie di allocazione dello spazio nella creazione/modifica dei file.

Una complicazione, rispetto alla memoria centrale, è il fatto che, mentre un'area di memoria viene allocata una volta per tutte durante la vita di un processo (se serve altra memoria, il processo allocherà una nuova area), un file può crescere (e inizialmente non occupa spazio), rendendo così difficili strategie come la best fit.

Allocazione contigua

Ogni file occupa una sequenza di blocchi contigui sul disco (fig. 9.2).

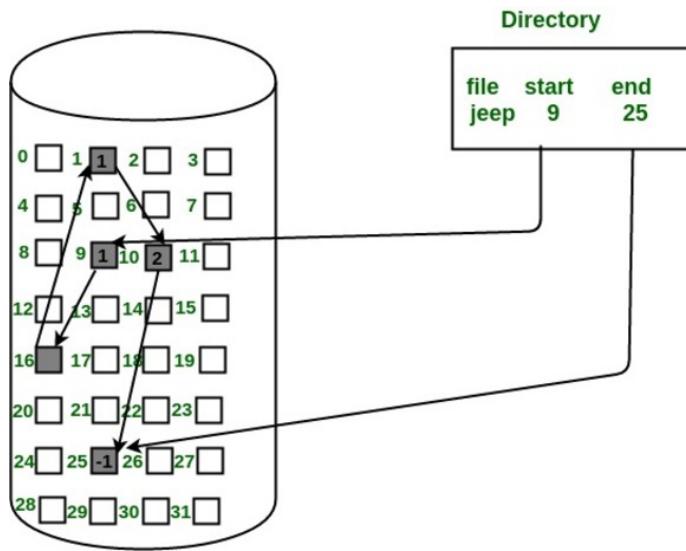


Figura 9.3: Allocazione dei blocchi a lista concatenata (da geeksforgeeks.org).

Vantaggi La directory deve contenere soltanto il blocco di inizio e la lunghezza. Inoltre, blocchi contigui garantiscono un accesso sequenziale più veloce, ma è supportato senza problemi anche l'accesso sequenziale: per accedere alla posizione p , basta caricare il blocco $b_0 + \lfloor p/B \rfloor$, dove B è la dimensione di un blocco e b_0 è il blocco iniziale del file.

Svantaggi Frammentazione esterna, difficile collocare un file non sapendo quanto crescerà, si rende necessario un compattamento periodico, oppure lo spostamento di un file che non ha più spazio.

Allocazione a lista

I blocchi sono organizzati in liste concatenate (Fig. 9.3). Una directory contiene i puntatori al primo e all'ultimo blocco del file (per gli append), ogni blocco contiene un puntatore al blocco successivo.

Vantaggi Creazione ed estensione del file semplice: basta prendere un blocco dalla lista dei blocchi liberi e concatenarlo al file. Nessuna frammentazione esterna, un file non deve essere contiguo.

Svantaggi Non è possibile l'accesso casuale: per sapere la posizione di un blocco bisogna scorrere tutti i precedenti. Blocchi sparsi causano inefficienza. Problemi di affidabilità: se un blocco è corrotto, il resto del file è perduto (a meno di usare liste doppiamente concatenate).

La File Allocation Table (FAT) Una variante dell'allocazione a lista nei sistemi MS-DOS e OS/2: la partizione contiene una tabella (FAT) di indici interi, uno per blocco. Il contenuto della tabella è l'indice del blocco successivo nel file o nella lista dei blocchi liberi. La FAT è tipicamente caricata in memoria, quindi consente l'accesso casuale senza la lettura di tutti i blocchi.

A seconda della dimensione e del numero di blocchi, la FAT è costituita da parole a 12, 16 o 32 bit.

Allocazione indicizzata

Ogni file ha un blocco indice contenente una tabella di indici (index table) dei blocchi fisici che lo costituiscono (Fig. 9.4). La directory contiene l'indirizzo (nel disco) del blocco indice.

La posizione dell' i -esimo blocco del file si trova all' i -esima riga della index table.

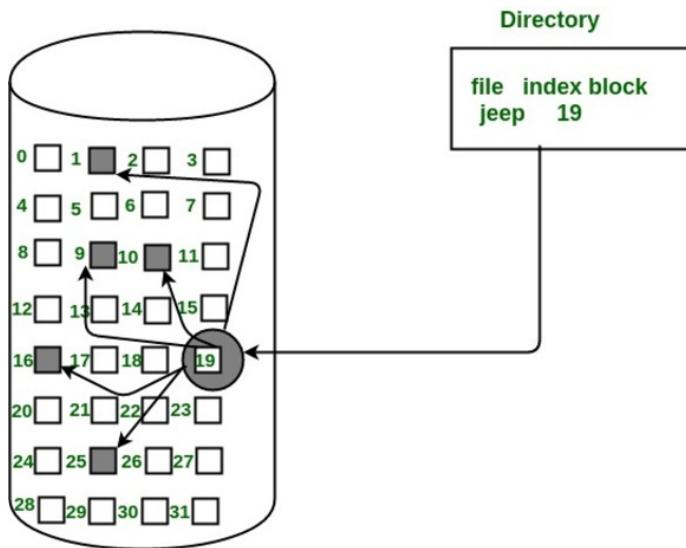


Figura 9.4: Allocazione indicizzata (da geeksforgeeks.org).

Vantaggi Accesso casuale efficiente, nessuna frammentazione esterna

Svantaggi Overhead di occupazione: ogni file richiede un blocco in più per conservare l'indice; overhead di lettura: per accedere a un blocco del file bisogna prima leggere il blocco indice. Dimensioni del file limitate dalla dimensione della tabella: ipotizzando blocchi da $2^{12}B = 4KB$ e parole da 32 bit, un blocco può indicizzarne $2^{10} = 1024$, quindi un file può essere grande fino a $2^{10} \cdot 2^{12}B = 4MB$.

Per indicizzare file di dimensioni maggiori, ci sono due possibilità:

- Tabelle indice concatenate: l'ultima riga di un blocco indice contiene l'indirizzo del blocco indice successivo (l'accesso casuale a grandi file diventa però difficile, perché bisogna scorrere la lista dei blocchi indice).
- Tabelle indice gerarchiche a due, tre o quattro livelli: il nodo della directory punta a una tabella indice le cui righe puntano a tabelle indice.

Con le dimensioni di blocco e parola già considerate, ogni livello della gerarchia di indirezione causa un aumento di un fattore $2^{10} = 1024$ della dimensione massima del file. Una gerarchia a due livelli, dunque, permette di lavorare con file fino a 4GB; la gerarchia a tre livelli permette di gestire file fino a 4TB.

Gli inode di UNIX

UNIX utilizza un sistema combinato in cui alcuni blocchi, detti *inode* (index node) sono raccolti in una tabella predefinita. Ogni nodo di una directory che fa riferimento a un file fisico punta a un inode, la cui struttura è riassunta in Fig. 9.5: la lista di indici è suddivisa in:

- dai 10 ai 12 indici diretti a blocchi dati, che permettono di accedere velocemente a primi blocchi di un file, evitando strutture gerarchiche per file di piccole dimensioni;
- un indice a una lista di primo livello, composta di 128 indici, quindi in grado di puntare fino a 512KB di dati;
- un indice a una lista di secondo livello (fino a 64MB);

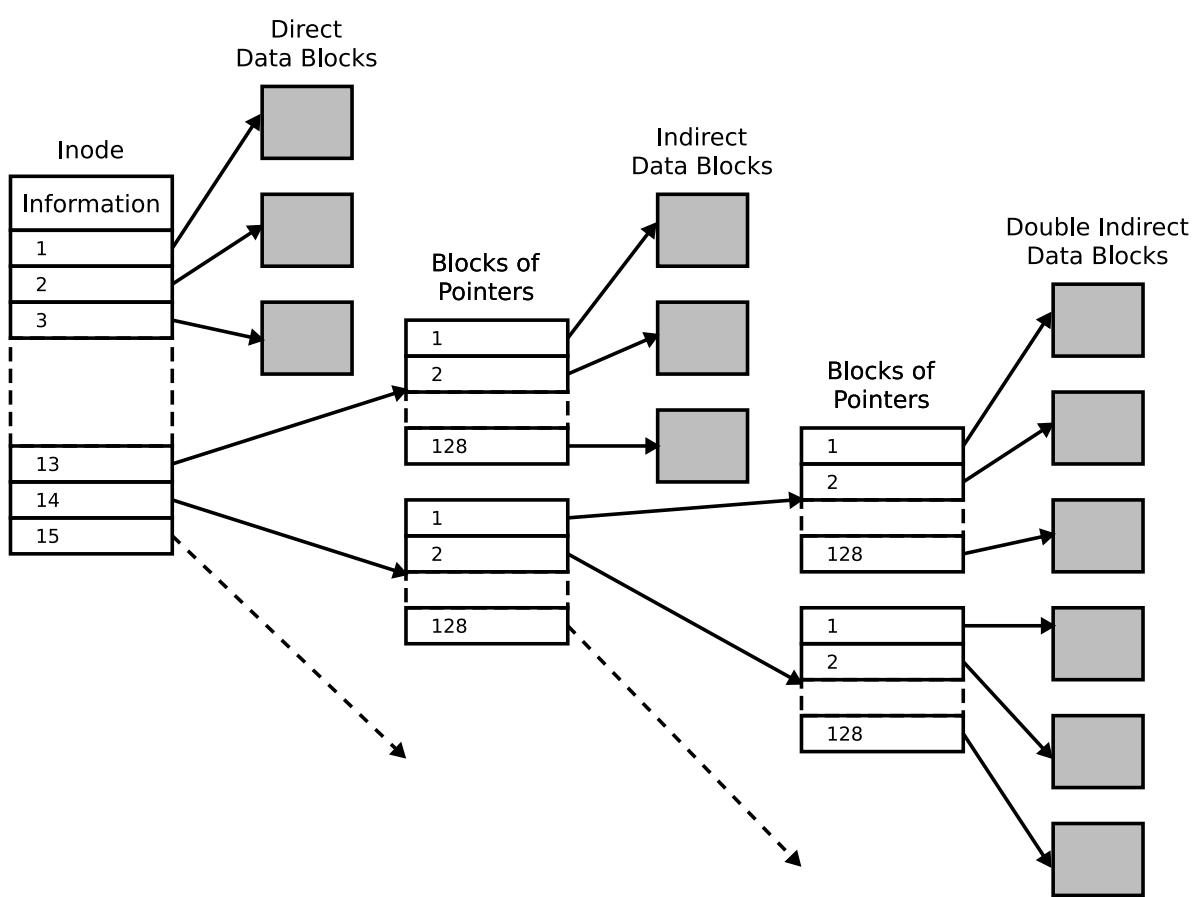


Figura 9.5: Gli inode di UNIX (da Wikipedia).

- un indice a una lista di terzo livello (fino a 4GB).

Chiaramente, questi numeri possono cambiare a seconda delle dimensioni dei blocchi.

Oltre ai puntatori ai blocchi, l'inode contiene tutti gli attributi del file, a eccezione del nome (contenuto nella directory). In questo modo, due directory possono puntare allo stesso inode per rappresentare lo stesso file in tutto e per tutto, tranne il nome.

9.2.3 Realizzazione delle directory

A meno dei casi molto semplificati visti prima (MS-DOS 1.0, CP/M), una directory è un file con un attributo speciale, che il SO interpreta come un elenco di nodi che fanno riferimento a file e ad altre directory. Questi nodi possono essere organizzati come:

- una lista lineare di nomi file e puntatori ai blocchi dati: di facile realizzazione, ma la ricerca del nome di un file richiede tempo lineare; si può ovviare cono una ricerca binaria su lista ordinata (di complessità logaritmica), ma è necessario tenere ordinata la lista quando si inseriscono nuovi nodi;
- una tabella hash, con un migliore tempo di ricerca a prezzo di una maggiore complessità di implementazione e un maggiore spreco di spazio (una tabella hash è efficiente quando è molto più grande del numero di dati che deve memorizzare).

9.2.4 La gestione dello spazio libero

È necessario tenere traccia dei blocchi liberi di un volume. Ci sono vari metodi per farlo.

Vettore di bit

Si può mantenere su disco, in un'area apposita, un array con un bit per ogni blocco. Il bit vale 1 se il blocco è usato, 0 se è libero. Per quanto lo spazio richiesto sia ridotto rispetto alle dimensioni del disco, un disco da $2^{40}\text{B} = 1\text{TB}$ organizzato in blocchi da $2^{12}\text{B} = 4\text{KB}$ contiene $2^{40-12} = 2^{28}$ (più di 256 milioni) blocchi. Se ogni byte ne rappresenta 8, il vettore di bit richiede $2^{28-3}\text{B} = 2^{25}\text{B} = 32\text{MB}$.

Se dunque vogliamo mantenerlo in memoria per efficienza, il vettore occuperà uno spazio non trascurabile.

Lista concatenata

Come fatto per i blocchi appartenenti a un file, anche i blocchi liberi possono essere organizzati in una lista in cui ciascuno punta al successivo, con un puntatore scritto direttamente nel blocco, o indirettamente in una FAT.

Per velocizzare la gestione e accorciare la lista, si possono collegare tra loro solo i blocchi iniziali di sequenze contigue di blocchi liberi, insieme a un contatore che dice di quanti blocchi è composta la sequenza.

Raggruppamento

Simile all'allocazione indicizzata dei blocchi: il primo blocco libero contiene gli indirizzi dei successivi n ; l'ultimo di questi blocchi contiene gli indici degli n successivi, e così via, come si vede in Fig. 9.6.

9.3 Efficienza e prestazioni

Consideriamo per riferimento un tipico disco rigido (Fig. 9.7). Un hard disk è suddiviso in facce (due per disco). Su ciascuna faccia è disposta una testina che lo legge per tracce concentriche. Ogni traccia è suddivisa in settori.

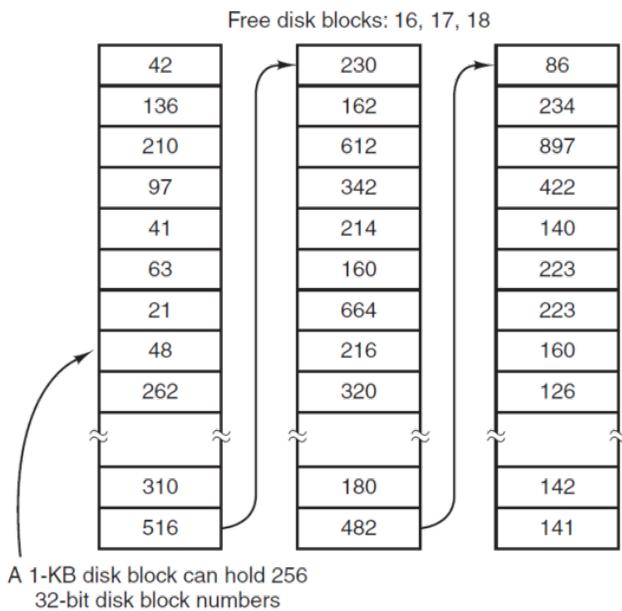


Figura 9.6: Raggruppamento di blocchi liberi: ciascuno di questi tre blocchi liberi contiene gli indirizzi di altri n blocchi. (da gcallah.github.io).

Il settore è la più piccola quantità di dati leggibile e scrivibile sul disco; la dimensione più comune è di 512 byte. L'organizzazione vista finora considera come unità minima di lettura/scrittura un gruppo (blocco, cluster) di settori da 4KB.

I tempi di accesso al disco sono suddivisi In

- Tempo di ricerca (seek time): tempo richiesto dalla testina di lettura/scrittura per posizionarsi sulla traccia corretta, dai 200 microsecondi a un millisecondo fra tracce successive;
- Tempo di latenza (latency time): tempo medio di attesa perché il settore richiesto passi sotto la testina. Per un disco da 7200rpm (giri al minuto), la latenza media è il tempo richiesto a compiere mezzo giro, quindi $0,5 \cdot 60 / 7200 \approx 4\text{ms}$.
- Tempo di trasferimento (transfer time) dipende dalla velocità della RAM di sistema, solitamente 300MB/s.

Come si vede, il tempo per accedere a un blocco su disco è dominato dal primo valore, il seek time.

È dunque importante che, se possibile, i file siano memorizzati su blocchi contigui. I sistemi UNIX cercano anche di mantenere gli inode distribuiti su disco e di utilizzare quelli in prossimità dei blocchi dati a cui si riferiscono.

9.3.1 La cache del disco

Per sfruttare il principio della località già visto nella memoria centrale, il SO dedica una porzione di memoria ai blocchi utilizzati più di frequente. Le problematiche di gestione di questa cache sono le solite:

- Dimensione
- Politiche di rimpiazzamento:
 - Che blocco si elimina se non c'è più spazio nella cache?

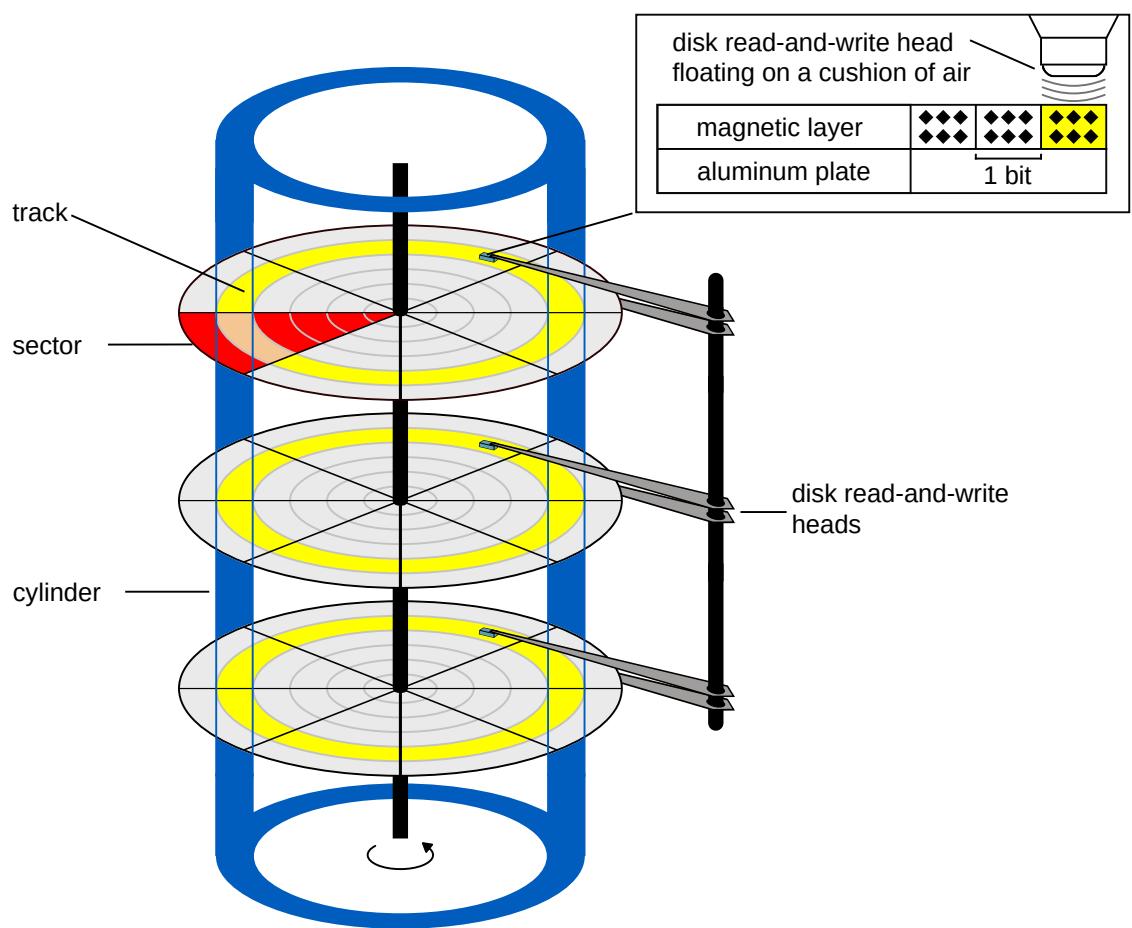


Figura 9.7: Struttura di un disco rigido (facce, tracce, settori) (da Wikipedia).

- LRU, LFU, scelta casuale...
- LRU non è efficiente per l'accesso sequenziale; meglio rilasciare un blocco completamente letto, anche se da poco tempo.
- La cache serve anche a fare da buffer di scrittura: non è economico scrivere ripetutamente un blocco su disco ad ogni modifica di pochi byte.
 - Write-back: scrivere solo quando il blocco va rimosso dalla cache (poco affidabile in caso di crash);
 - Write-through: si scrive sempre a ogni modifica (meno efficiente, in pratica la cache si usa solo in lettura).

Per risolvere i problemi di consistenza dovuti all'uso della cache in scrittura, ogni SO possiede delle utility per controllare la corrispondenza fra le informazioni contenute nelle directory, negli inode e nei blocchi dati alla ricerca di discrepanze, con vari algoritmi, spesso euristici, per risolverle.

I file system journaled

Per ottimizzare la scrittura su disco in presenza di caching, alcuni filesystem (Apple HPFS, Linux Ext4) scrivono tutte le transazioni in scrittura su un log.

Una transazione si considera avvenuta quando è stata scritta sul log, anche se il file system non può essere aggiornato.

Le transazioni vengono scritte in modo asincrono nel file system, e ogni volta che questo avviene la transazione è rimossa dal log.

Se il sistema va in crash, le transazioni non avvenute sono quelle presenti sul log.

In questo modo, tutte le operazioni di scrittura vengono temporaneamente salvate su disco in tracce contigue dedicate al log, ottimizzando i tempi di seek.

9.3.2 Scheduling degli accessi al disco

Abbiamo visto che il tempo dominante negli accessi al disco è il seek time (posizionamento della testina sulla traccia). Di conseguenza, quando più processi richiedono operazioni di lettura/scrittura su disco il SO può decidere di non eseguirle nell'ordine in cui sono pervenute, ma di riordinarle in modo da ottimizzare gli spostamenti della testina.

Il tempo di spostamento dalla traccia tr_1 alla traccia tr_2 è proporzionale alla differenza $|tr_1 - tr_2|$. Se la testina si trova originariamente sulla traccia tr_0 , data una sequenza di tracce richieste $(tr_1, tr_2, \dots, tr_n)$, il tempo totale per visitare le tracce nella sequenza data è dunque

$$T \propto \sum_{i=1}^n |tr_i - tr_{i-1}|.$$

dove la costante di proporzionalità, come abbiamo visto, è dell'ordine dei millisecondi. Ci si pone dunque il problema di individuare la permutazione della sequenza di tracce che minimizza il valore di T .

First Come First Served (FCFS)

La baseline con cui confrontarsi è ovviamente l'algoritmo che mantiene l'ordine di richiesta. In Fig. 9.8 (sinistra) consideriamo il tempo impiegato a servire la seguente sequenza:

- Posizione corrente della testina all'istante iniziale: $tr_0 = 53$;
- Accessi richiesti: $(98, 183, 37, 122, 14, 124, 65, 67)$.

Il tempo totale richiesto è 640 volte il tempo di spostamento da una traccia alla successiva.

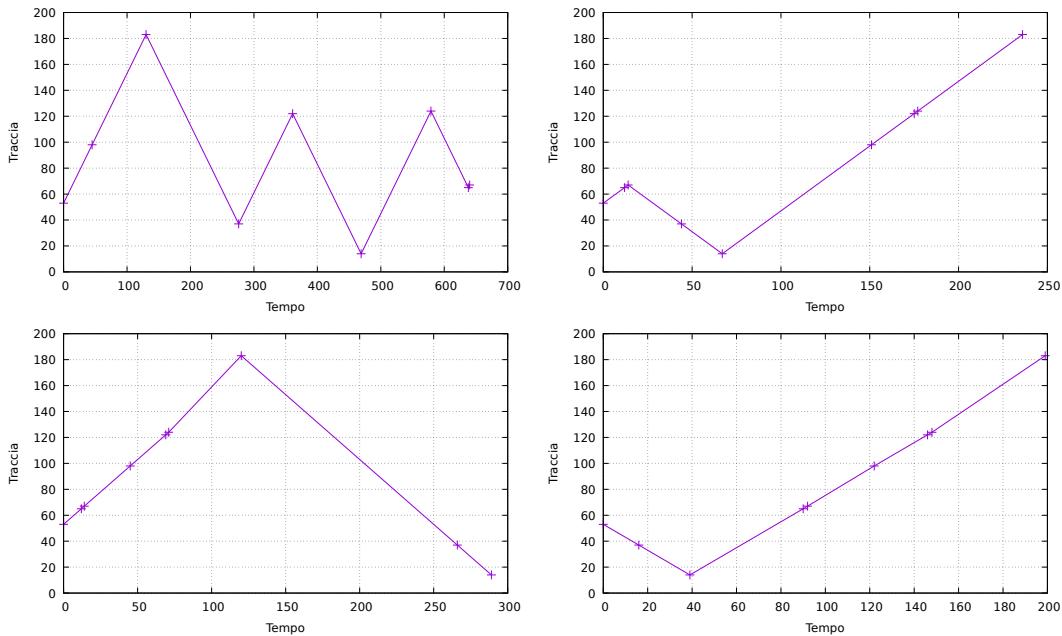


Figura 9.8: Scheduling FCFS (in alto a sinistra), SSTF (in alto a destra), SCAN con partenza ascendente (in basso a sinistra) e SCAN con partenza discendente (in basso a destra) per le richieste di accesso alle tracce del disco rigido.

Shortest Seek Time First (SSTF)

Spostiamo la testina alla traccia più vicina a quella corrente fra quelle finora richieste. Nell'esempio precedente, le tracce vengono dunque visitate nell'ordine seguente:

$$(65, 67, 37, 14, 98, 122, 124, 183),$$

e il tempo di percorrenza complessivo, calcolato come in Fig. 9.8 (destra), è di 183 volte il tempo di spostamento da una traccia alla successiva.

Un problema di questo approccio, per quanto molto più efficiente del precedente, è il rischio di starvation: se continuano ad arrivare richieste per tracce vicine, quelle più lontane dalla testina potrebbero essere servite con un ritardo arbitrariamente alto. Questo vale soprattutto per le tracce più esterne (vicine a 0 o al massimo).

L'algoritmo SCAN

Per ovviare al rischio di starvation, la testina mantiene un bit di stato che dice in che direzione s'è mossa l'ultima volta; finché sono presenti nuove richieste nella stessa direzione, la testina le visita in sequenza. Altrimenti inverte la direzione. Nell'esempio precedente, supponendo che la testina stia servendo tracce crescenti a partire da $tr_0 = 53$, queste verranno servite nell'ordine seguente:

$$(65, 67, 98, 122, 124, 183, 37, 14)$$

Se invece la testina si stava spostando verso le tracce inferiori, l'ordine risulta

$$(37, 14, 65, 67, 98, 122, 124, 183).$$

I due casi sono rappresentati nella parte bassa di Fig. 9.8, e comportano un seek time totale pari rispettivamente a 289 e 199 volte il tempo di spostamento fra tracce consecutive. Ovviamente, l'efficienza

dell'algoritmo SCAN può essere osservata soprattutto in un contesto dinamico nel quale le richieste di accesso continuano ad arrivare.

Parte II

Esercizi

Esercizi

Alcuni degli esercizi presentati in seguito sono stati proposti negli appelli scritti dello scorso anno accademico. In particolare:

- Primo appello (giugno 2021): esercizi 15, 20, 17.
- Secondo appello (luglio 2021): esercizi 1, 18, 19.
- Terzo appello (settembre 2021): esercizi 9, 2, 11

Nota bene — Le tracce di soluzione riportate qui non sono le sole possibili: spesso, gli esercizi sono aperti a varie possibili interpretazioni.

Esercizio 1

Consideriamo i seguenti processi (unità di tempo arbitrarie):

Processo	Istante di arrivo	Tempo di burst CPU
P_1	0	40
P_2	2	20
P_3	5	5

1.1) Calcolare il tempo di turnaround medio per i processi se lo scheduler utilizza l'algoritmo FCFS.

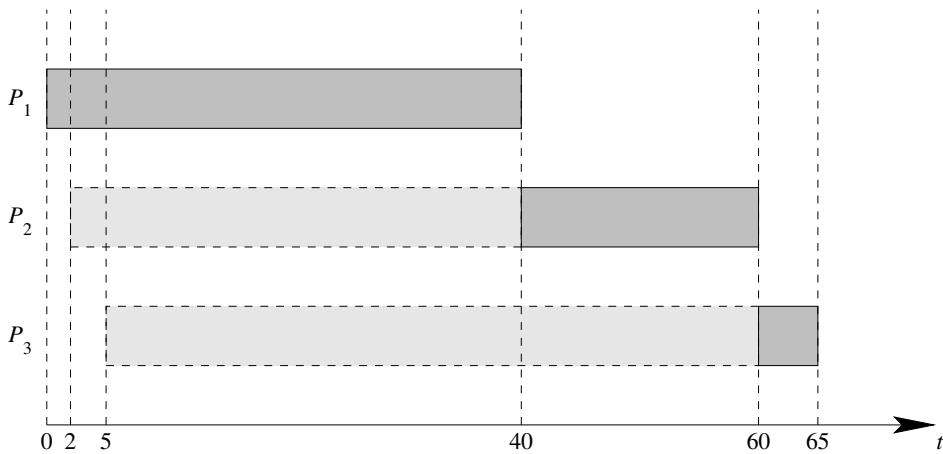
1.2) Calcolare il tempo di turnaround medio per i processi se lo scheduler utilizza l'algoritmo SJF (senza prelazione).

1.3) Verificare se la resa dell'algoritmo SJF nel punto precedente aumenterebbe tenendo la CPU ferma per le prime 5 unità di tempo.

1.4) Calcolare il tempo di turnaround medio per i processi se lo scheduler utilizza l'algoritmo SJF con prelazione (SRJF — shortest remaining job first).

Soluzione 1

1.1) All'istante $t = 0$ il processo P_1 arriva e viene eseguito. Durante la sua esecuzione, agli istanti $t = 2$ e $t = 5$, arrivano i processi P_2 e P_3 , che vengono eseguiti in quest'ordine non appena la CPU è disponibile. Nel diagramma qui sotto, i processi sono rappresentati da un rettangolo chiaro tratteggiato quando sono in attesa di essere eseguiti, da un rettangolo solido scuro mentre sono in esecuzione:



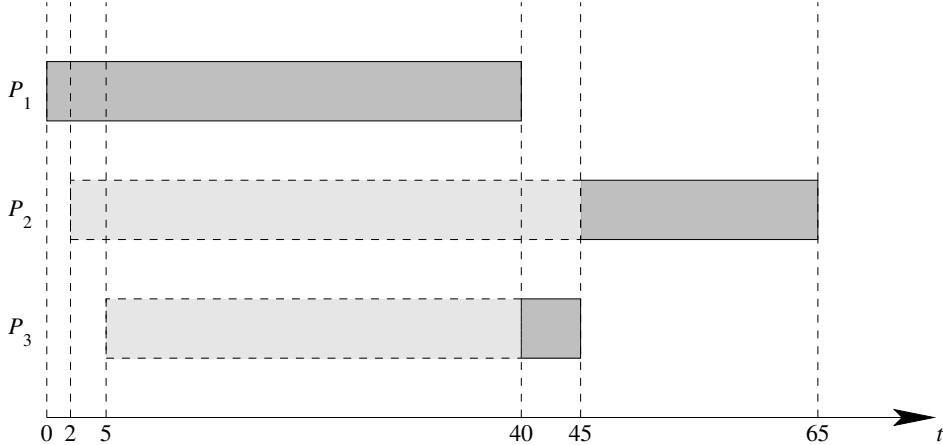
Calcoliamo i tempi di ripsosta come differenza fra l'istante in cui il processo va in esecuzione la prima volta e quello in cui il processo è arrivato nel sistema. I tempi di completamento (turnaround) sono la differenza fra l'istante in cui il processo termina l'esecuzione e l'istante di arrivo:

Processo	Risposta	Completamento
P_1	0	40
P_2	$40 - 2 = 38$	$60 - 2 = 58$
P_3	$60 - 5 = 55$	$65 - 5 = 60$

Il tempo di completamento medio richiesto dall'esercizio è dunque

$$t_{\text{turnaround}} = \frac{40 + 58 + 60}{3} = \frac{158}{3} \approx 52,67.$$

1.2) Il caso SJF è simile al precedente, ma al completamento di P_1 lo scheduler, trovando i due processi P_2 e P_3 in coda, preferisce quello con burst CPU inferiore:



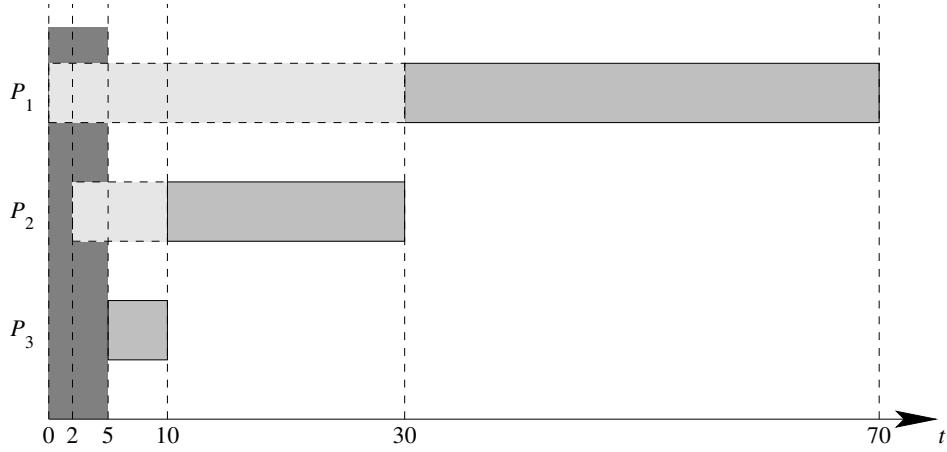
Si noti che la CPU impiega lo stesso tempo di prima a completare l'esecuzione dei tre processi, ma in questo caso uno dei tre processi termina prima, garantendo un tempo di completamento inferiore:

Processo	Risposta	Completamento
P_1	0	40
P_2	$45 - 2 = 43$	$65 - 2 = 63$
P_3	$40 - 5 = 35$	$45 - 5 = 40$

Di conseguenza,

$$t_{\text{turnaround}} = \frac{40 + 40 + 63}{3} = \frac{143}{3} \approx 47,67.$$

1.3) Se lasciamo lo scheduler inattivo per le prime 5 unità di tempo, in attesa di raccogliere processi da eseguire, la situazione è quella in figura:



Il risultato è

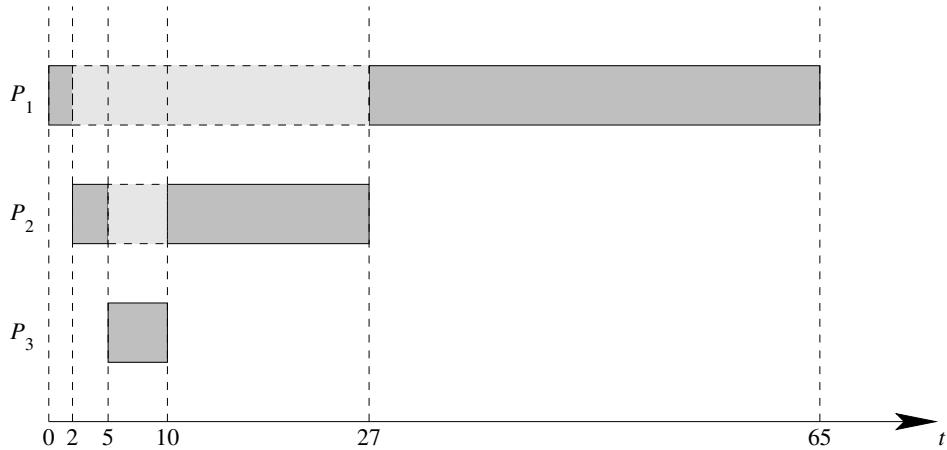
Processo	Risposta	Completamento
P ₁	30	70
P ₂	10 - 2 = 8	30 - 2 = 28
P ₃	5 - 5 = 0	10 - 5 = 5

Di conseguenza,

$$t_{\text{turnaround}} = \frac{5 + 20 + 70}{3} = \frac{95}{3} \approx 31,67$$

con un evidente, notevole vantaggio in termini di tempi di completamento.

1.4) Infine, se consentiamo la prelazione, quando entra il processo P₂, all'istante t = 2, il suo CPU burst è più breve del tempo rimanente per il processo P₁ attualmente in esecuzione, quindi gli viene preferito. Allo stesso modo, P₃ causerà la sospensione di P₂.



Notiamo che, in questo caso (ma non è una proprietà generale dell'algoritmo) i processi vanno subito in esecuzione, quindi per tutti il tempo di attesa è nullo.

Processo	Risposta	Completamento
P_1	0	65
P_2	0	$27 - 2 = 25$
P_3	0	$10 - 5 = 5$

Di conseguenza,

$$t_{\text{turnaround}} = \frac{65 + 25 + 5}{3} = \frac{95}{3} \approx 31,67.$$

Esercizio 2

Si consideri il seguente insieme di processi (unità di tempo arbitrarie; un numero di priorità maggiore indica una priorità più elevata):

Processo	CPU burst	Priorità
P_1	2	2
P_2	1	1
P_3	8	4
P_4	4	2
P_5	5	3

2.1) Illustrare con un diagramma l'esecuzione di questi processi con ciascuno dei seguenti algoritmi di scheduling:

- FCFS;
- SJF;
- con priorità;
- RR con quanto pari a 2 unità di tempo.

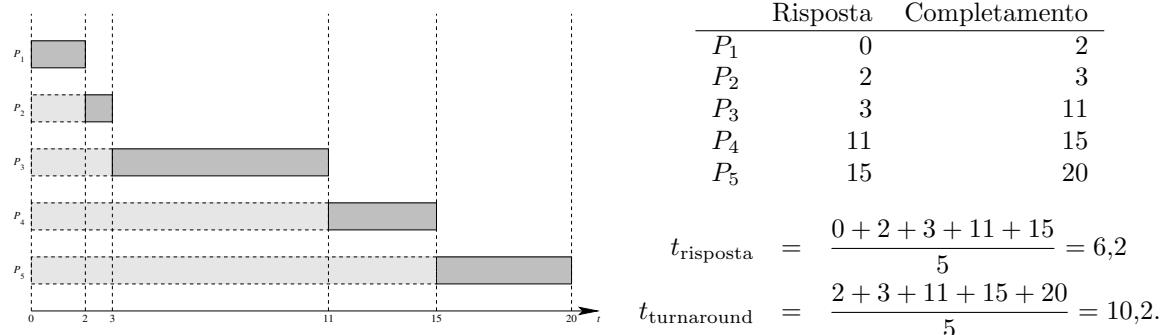
Solo l'ultimo caso prevede prelazione.

2.2) Calcolare i tempi di attesa e di completamento (turnaround) di ciascun processo nei quattro casi considerati.

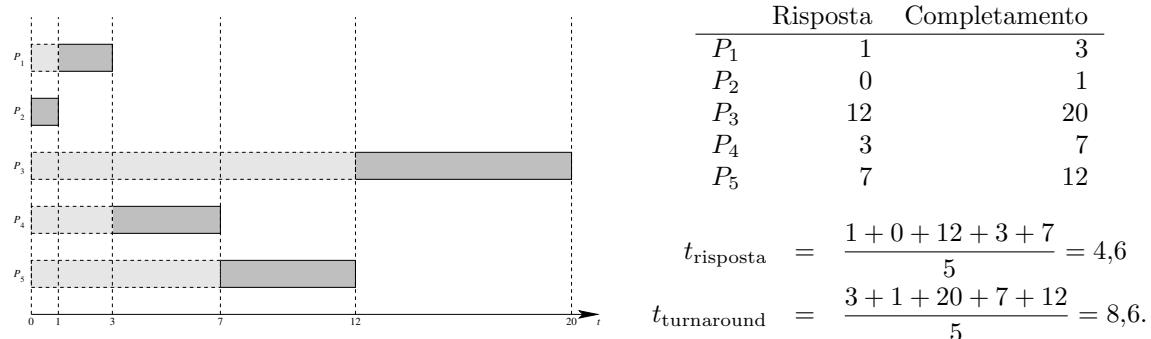
2.3) Quale algoritmo di scheduling ha il minor tempo medio di attesa?

Soluzione 2

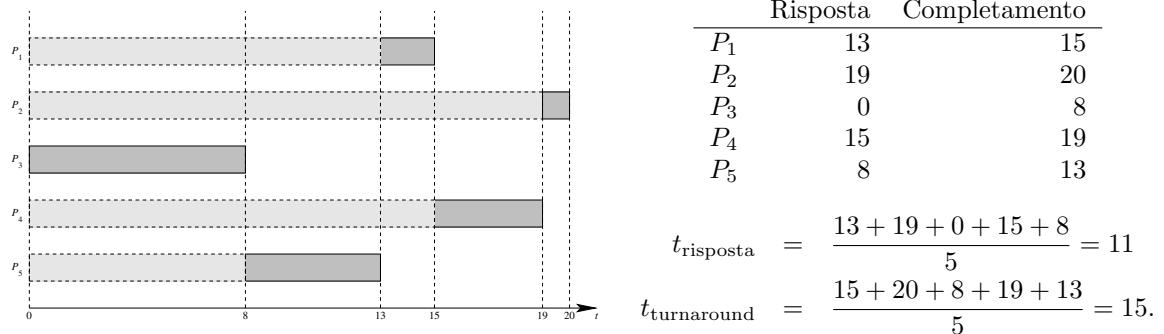
2.1) Per FCFS, assumiamo che i processi, pur arrivando nello stesso istante, siano accodati nell'ordine P_1, \dots, P_5 :



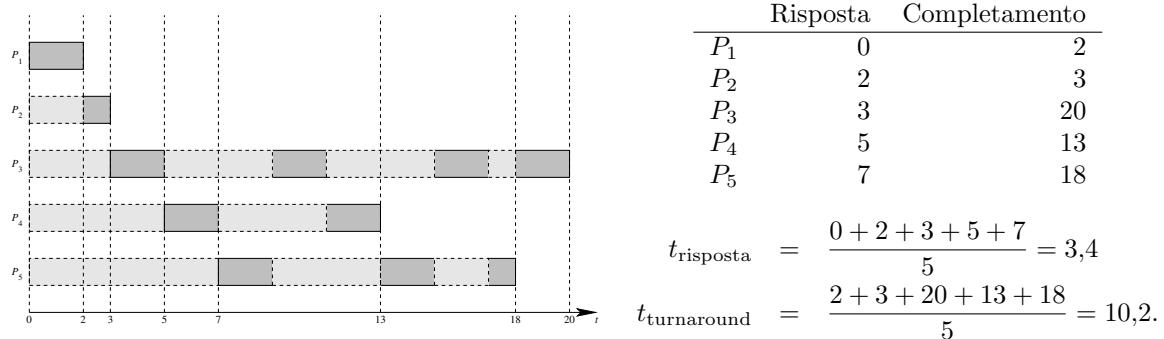
Per SJF, sappiamo che tutti i processi sono noti allo scheduler fin dall'istante $t = 0$:



Per quanto riguarda lo scheduling con priorità, dobbiamo semplicemente eseguire i processi nell'ordine richiesto, a partire dal valore di priorità più elevato. Trattandosi di un ordinamento arbitrario, non possiamo aspettarci una particolare efficacia:



Infine, per la politica Round-robin assumiamo che l'ordine di esecuzione sia FCFS e che il cambio di contesto richieda, come negli altri casi, un tempo trascurabile:



2.2) I calcoli per i tempi medi sono riportati accanto ai relativi diagrammi.

2.3) Osserviamo che, come previsto, RR garantisce un tempo di risposta ridotto (sarebbe stato ancora più basso se il quanto fosse stato di un'unità di tempo), ma tempi di completamento non particolarmente vantaggiosi.

Esercizio 3

In un computer con una CPU è in funzione uno scheduler a tre livelli di priorità (alta=1, media=2, bassa=3). Un processo ha prelazione immediata su quelli di priorità inferiore, e ciascun livello di priorità opera in modalità FCFS.

All'istante iniziale ($t = 0\text{ms}$) arriva un processo L a bassa priorità (3) con un unico CPU burst di 20ms. Ad ogni istante $t_i = (5 + 10i)\text{ms}$, per $i = 0, 1, 2, \dots$, arriva un processo M_i a priorità media (2) e un unico CPU burst di 5ms. All'istante $t = 10\text{ms}$ arriva un processo H ad alta priorità (1) con un unico CPU burst di 15ms.

3.1) Mostrare lo scheduling dei processi per i primi 60ms.

Assumiamo ora che i processi L e H consistano in una sezione critica ininterrotta sulla stessa risorsa non condivisibile e non prelazionabile R . In altri termini, sia S che H sono inclusi fra una `wait(R)` e una `signal(R)`.

3.2) Mostrare lo scheduling dei processi per i primi 60ms. Osservare: (a) il processo H non può più esercitare la prelazione su L ; inoltre, (b) alcuni processi M_i esercitano prelazione su L ritardandone la conclusione e, di fatto, tenendo bloccato H (nonostante quest'ultimo abbia priorità maggiore).

Il fenomeno illustrato al punto 3.2 è detto *inversione di priorità*. Un metodo per ovviare al problema è il seguente: *se un processo a priorità maggiore è in attesa di una risorsa assegnata a un processo di priorità minore, quest'ultimo assume la priorità del primo finché non libera la risorsa*.

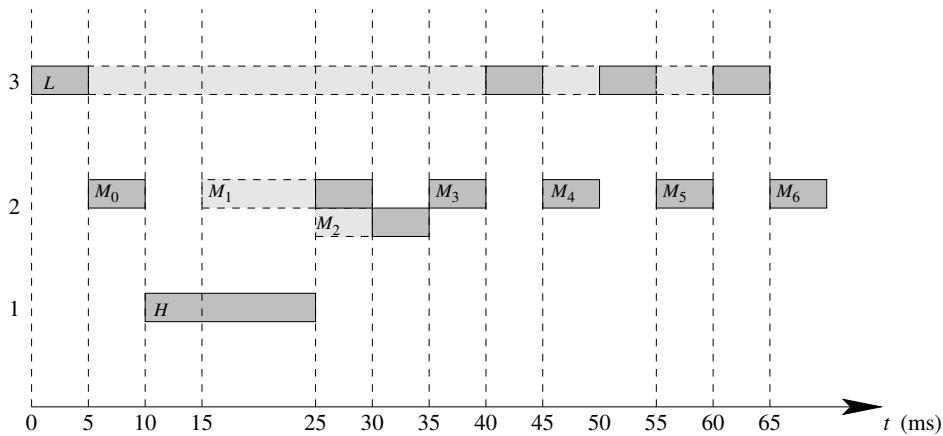
3.3) Mostrare lo scheduling risultante dall'applicazione della regola appena descritta; osservare che il processo H (pur costretto ad attendere il completamento di L) non è più ulteriormente ritardato dai processi M_i , che non possono più interrompere L .

Soluzione 3

Riassumendo in forma tabellare, i processi arrivano nell'ordine seguente:

Processo	Istante di arrivo (ms)	Tempo di burst CPU (ms)	Priorità
L	0	20	3
M_0	5	5	2
H	10	15	1
M_1	15	5	2
M_2	25	5	2
M_3	35	5	2
M_4	45	5	2
M_5	55	5	2
M_6	65	5	2
...			

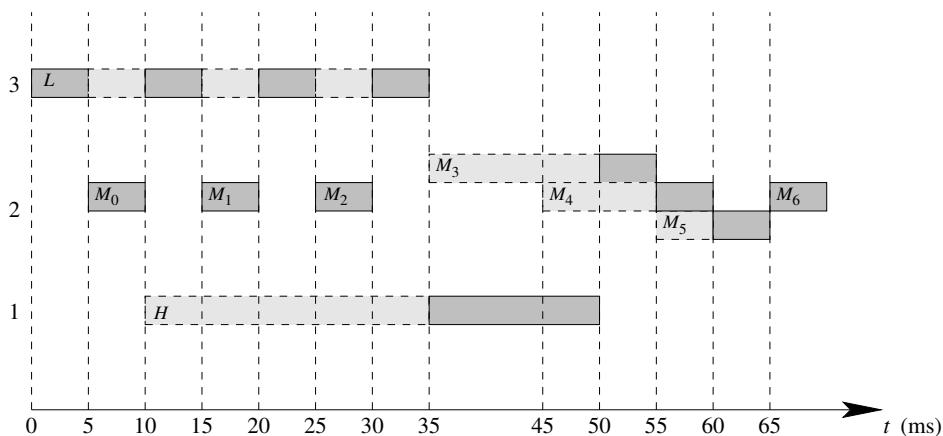
3.1) Facciamo riferimento alla figura:



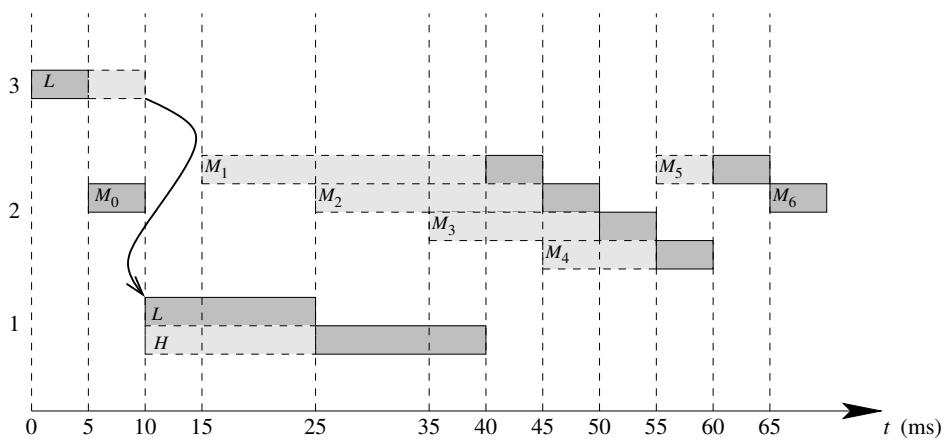
- All’istante $t = 0$ arriva il processo L e lo scheduler lo mette in esecuzione.
- A $t = 5\text{ms}$ arriva il primo dei processi a media priorità, M_0 , quindi lo scheduler mette L in stato waiting (in virtù della sua minore priorità) e M_0 in stato running.
- A $t = 10\text{ms}$ il processo M_0 termina; l’arrivo del processo ad alta priorità H forza lo scheduler a mettere in esecuzione quest’ultimo a discapito di L , che resta in waiting.
- A $t = 15\text{ms}$ arriva M_1 , che viene messo in attesa in quanto la sua priorità è minore di quella del processo in esecuzione.
- A $t = 25\text{ms}$ ccadono tre eventi (in “contemporanea”: possiamo scegliere noi l’ordine in cui considerarli): arriva M_2 , H termina e lo scheduler può mettere in esecuzione un altro processo. Fra i tre in coda (L , M_1 e M_2), supponiamo che venga scelto M_1 in quanto il più vecchio fra quelli più prioritari.
- Per lo stesso motivo, a $t = 30\text{ms}$, quando termina M_1 , lo scheduler mette in esecuzione M_2 preferendolo a L .
- A $t = 35\text{ms}$, M_2 termina, ma arriva subito M_3 , che quindi entra subito in esecuzione.
- A $t = 40\text{ms}$, finalmente il backlog dei processi a media priorità è terminato, e L può tornare in esecuzione.
- Da questo momento in poi, e fino alla terminazione di L a $t = 65\text{ms}$, la CPU eseguirà L sospendendolo all’arrivo dei processi a media priorità.

3.2) La differenza principale rispetto a prima è che, quando all’istante $t = 10\text{ms}$ arriva il processo H , questo non può prendere subito il possesso della CPU (ovvero ne prende possesso per il tempo necessario a eseguire la `wait()` sulla risorsa e viene subito messo in stato di blocco) e deve attendere che L liberi la risorsa, il che (stando al testo dell’esercizio) avverrà solo alla sua conclusione.

Intanto, però, L continua ad essere interrotto dai processi M_0 , M_1 e M_2 che ne ritardano il completamento. Indirettamente, dunque, i processi a priorità intermedia ritardano l’esecuzione di H , anche se questo sarebbe prioritario:



3.3) Se, al momento in cui H richiede la risorsa (quindi al suo stesso arrivo) il processo che la detiene viene “promosso” al suo stesso livello di priorità, allora L non sarà più interrotto dai processi a priorità intermedia e potrà completare più rapidamente, garantendo così un avvio più rapido per H :



Ovviamente, viene ad accumularsi un maggiore backlog di processi intermedi, che andranno smaltiti al termine di **H**.

Esercizio 4

Lo scheduler di un computer a una CPU deve gestire i seguenti processi (unità di tempo arbitrarie):

Processo	Istante di arrivo	CPU burst
P_1	0	3
P_2	2	6
P_3	4	4
P_4	6	5
P_5	8	2

Supponiamo che lo scheduler conosca con precisione la durata del CPU burst di ogni processo. Ovviamente, ad ogni istante di tempo lo scheduler è a conoscenza dei soli processi già arrivati.

4.1) Simulare l'azione dello scheduler sulla base di ciascuna delle seguenti politiche:

- First Come First Served;
- Shortest Job First senza prelazione;
- Highest Response Ratio Next;
- Shortest Job First con prelazione (Shortest Remaining Time First);
- Round-Robin con quanto $q = 1$;
- Round-Robin con quanto $q = 4$.

Assumere che i tempi di cambio di contesto, avvio/terminazione di un processo e di esecuzione dello scheduler siano trascurabili.

4.2) Calcolare i tempi medi di risposta, e di completamento per ciascuna politica.

Esercizio 5

Con riferimento all'esercizio 4:

5.1) Simulare lo scheduler se il tempo di cambio di contesto T_{switch} vale

- $T_{\text{switch}} = 1$;
- $T_{\text{switch}} = 0, 1$.

5.2) Ricalcolare i tempi medi.

Esercizio 6

Con riferimento all'esercizio 4:

6.1) Simulare lo scheduler RR per una durata del quanto molto piccola $q \rightarrow 0$, nell'ipotesi che il tempo di cambio di contesto sia comunque trascurabile rispetto al quanto^a.

6.2) Ricalcolare i tempi medi.

Suggerimento — *Si tratta di un'approssimazione continua: invece di simulare quanti di tempo discreti, possiamo vedere i processi come se fossero in esecuzione contemporaneamente, ciascuno a una velocità ridotta di un fattore pari al numero di processi in esecuzione.*

^aL'ipotesi è valida se $T_{\text{CPU burst}} \gg q \gg T_{\text{switch}}$; ad esempio, quando il CPU burst è dell'ordine dei minuti, con quanti dell'ordine dei decimi di secondo e tempi di cambio di contesto dell'ordine dei millisecondi.

Esercizio 7

Un sistema dispone di 4 risorse diverse, R_1 , R_2 , R_3 e R_4 , ad accesso esclusivo. Sono in esecuzione 3 processi:

- P_1 utilizza le risorse R_1 e R_2 ;
- P_2 utilizza le risorse R_2 , R_3 e R_4 ;
- P_3 utilizza le risorse R_1 e R_4 .

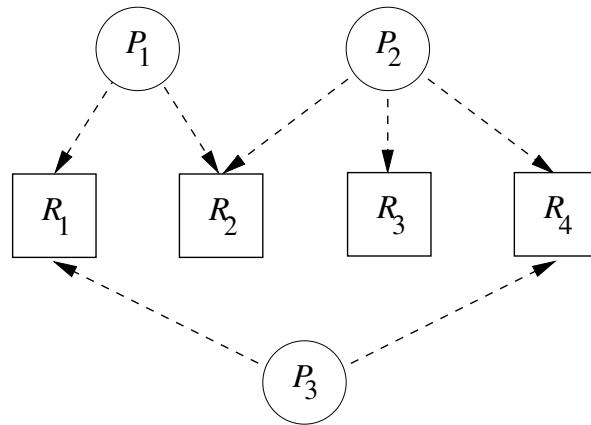
Ciascun processo utilizza le risorse all'interno di una sezione critica controllata attraverso dei semafori (uno per risorsa). Una stessa sezione critica potrebbe richiedere l'uso contemporaneo di due o più risorse diverse.

7.1) Individuare una possibile situazione di deadlock e descriverla utilizzando un grafo di allocazione delle risorse.

7.2) Supponendo che ogni processo debba elencare prima dell'esecuzione le risorse a cui intende accedere, mostrare come il SO può evitare la situazione di deadlock nel caso individuato al punto 7.1.

Soluzione 7

7.1) Esploriamo la situazione costruendo il diagramma di allocazione delle risorse: le righe tratteggiate rappresentano le potenziali richieste da parte dei processi:

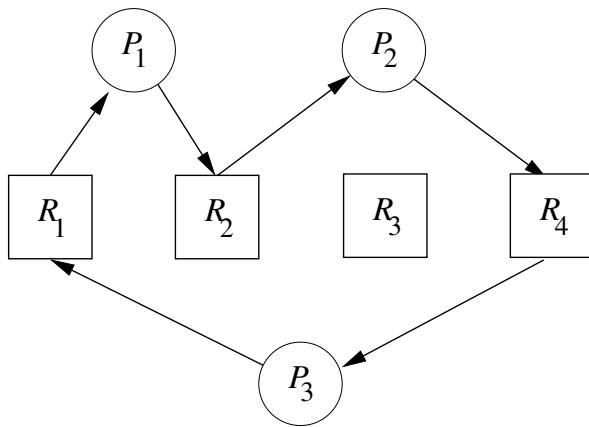


Possiamo osservare che la risorsa R_3 non può avere nessun ruolo in un deadlock, e che per avere un ciclo nel grafo sarà necessario far entrare in gioco tutti e tre i processi.

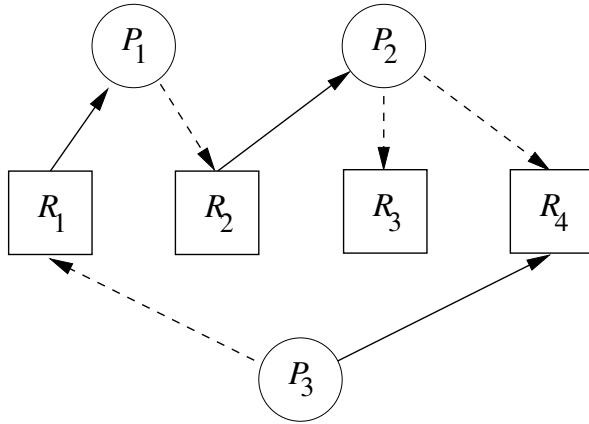
Come esempio, consideriamo la seguente sequenza di azioni:

1. P_1 chiede e ottiene R_1 ;
2. P_2 chiede e ottiene R_2 ;
3. P_3 chiede e ottiene R_4 ;
4. P_1 chiede R_2 , e resta in attesa che P_2 la liberi;
5. P_2 chiede R_4 , e resta in attesa che P_3 la liberi;
6. P_3 chiede R_1 , e resta in attesa che P_1 la liberi.

Possiamo verificare che tutte le condizioni per il deadlock sono verificate, come si vede dal diagramma:



7.2) Supponiamo che il sistema conosca a priori le possibili richieste (le frecce tratteggiate). Quando, al passo 3, P_3 richiede R_4 , il grafico diventa (notare la freccia solida da P_3 a R_4 , che rappresenta la richiesta in corso):



Se il sistema concedesse la risorsa, invertendo la freccia, si formerebbe un ciclo, che rappresenta un potenziale rischio di deadlock (se tutte le frecce tratteggiate si concretizzassero), quindi tiene P_3 in attesa finché l'assegnazione non garantisce uno stato sicuro.

Esercizio 8

Un sistema dispone di 3 tipi diversi di risorse, R_1 , R_2 e R_3 , ciascuna delle quali è disponibile in un certo numero di istanze ad accesso esclusivo. Nel sistema vengono avviati 4 processi, P_1, \dots, P_4 , ciascuno dei quali deve dichiarare, prima dell'esecuzione, il numero massimo di istanze di ciascuna risorsa che potrà allocare contemporaneamente.

- Delle risorse R_1 , R_2 e R_3 sono disponibili rispettivamente 6, 5 e 3 istanze;
- Il processo P_1 potrà allocare contemporaneamente fino a 2 istanze di R_1 , una di R_2 e 2 di R_3 ;
- Per il processo P_2 le istanze allocabili contemporaneamente sono 3, 2 e 1 rispettivamente.
- P_3 può allocare contemporaneamente un'istanza di R_1 e 3 istanze di R_2 e di R_3 ;
- P_4 può allocare contemporaneamente 2 istanze di R_1 e un'istanza di R_3 , mentre non richiede l'uso di R_2 .

8.1) Impostare i valori iniziali delle matrici `max`, `alloc` e `need` e del vettore `available` per l'algoritmo del banchiere sulla base delle informazioni riportate sopra.

Ad un certo punto, durante l'esecuzione dei quattro processi, le istanze allocate risultano essere le seguenti:

$$\text{alloc} = \begin{pmatrix} 1 & 1 & 0 \\ 2 & 1 & 1 \\ 1 & 2 & 1 \\ 1 & 0 & 0 \end{pmatrix}.$$

8.2) Dimostrare che il sistema si trova ancora in uno stato sicuro.

A questo punto, P_1 richiede una nuova istanza di R_1 .

8.3) Mostrare che l'allocazione metterebbe il sistema a rischio di deadlock.

8.4) Individuare un processo da portare a compimento affinché la richiesta di P_1 possa essere soddisfatta in sicurezza.

Esercizio 9

Due processi entrano ed escono ripetutamente da una sezione critica di codice, e sono soggetti al seguente vincolo: sia l'ingresso che l'uscita devono essere contemporanei e sincronizzati. In altri termini: se un processo è pronto a entrare nella sezione critica, deve attendere che anche l'altro lo sia; allo stesso modo, i due processi devono sincronizzare la propria uscita.

Proporre due diverse soluzioni al problema utilizzando due diversi strumenti di sincronizzazione a scelta: semafori, monitor, pipe bloccanti, spinlock...

Esercizio 10

Calcolare il numero di pagina e l'offset di ciascuno dei seguenti indirizzi logici a 16 bit

0x75B1, 0xFAA0, 0x1011

in ciascuno dei seguenti casi:

- 10.1)** pagine da 4KB;
- 10.2)** spazio di indirizzamento fisico a 15 bit suddiviso in 16 frame;
- 10.3)** pagine da 16KB.

Esercizio 11

Un sistema operativo ha indirizzi virtuali a 21 bit e indirizzi fisici a 16 bit. La dimensione della pagina è 2KB.

11.1) Di quante pagine logiche e di quanti frame fisici dispone il sistema?

11.2) Quante pagine si possono indicizzare in una tabella delle pagine a un livello, supponendo di volerla contenere in un singolo frame?

11.3) Quante voci contiene e quanto spazio occupa una tabella delle pagine invertita?

Esercizio 12

Un computer ha uno spazio di indirizzamento fisico a 32 bit e uno spazio di indirizzamento logico da 36 bit, gestito attraverso paginazione su richiesta e memoria di massa. La MMU gestisce pagine da 64KB.

12.1) Se la page table utilizza righe da 8 byte (per indirizzi fisici e varie informazioni di protezione), quante righe potrà contenere, al massimo, una page table a un singolo livello?

12.2) Come verrà suddiviso e tradotto l'indirizzo logico **0x9E45BA82A** in un indirizzo fisico mediante l'uso di una page table a un livello?

12.3) Proporre uno schema di page table a due livelli per il sistema sotto esame: indicare come si potrebbe suddividere l'indirizzo virtuale, quali saranno le dimensioni delle tabelle di primo e secondo livello. Come verrà tradotto lo stesso indirizzo?

12.4) Supponendo che si desideri utilizzare l'intero spazio di indirizzamento virtuale, quanto dovrà essere grande l'area di swap su disco?

Suggerimento — *Ovviamente, per i punti 12.2 e 12.3 potete ipotizzare liberamente il contenuto delle page table.*

Soluzione 12

Innanzitutto, valutiamo le dimensioni degli spazi di indirizzamento.

Uno spazio fisico a 32 bit significa che possiamo indirizzare un massimo di $2^{32}B = 4GB$ di RAM fisica.

Lo spazio logico permette di indirizzare fino a $2^{36}B = 64GB$.

Le pagine sono da 64KB = $2^{16}B$, quindi l'offset all'interno della pagina sarà rappresentato dai 16 bit meno significativi di ciascun indirizzo.

12.1)

$$\text{numero righe} = \frac{\text{dimensione pagina}}{\text{dimensione riga}} = \frac{2^{16}}{2^3} = 2^{13} = 8K (= 8192).$$

12.2) Come detto prima, se l'offset occupa i 16 bit meno significativi (4 cifre esadecimale), allora l'indirizzo individua la pagina di indice **0x9E45B** e l'offset **A82A** all'interno di tale pagina¹

12.3) Osserviamo che il numero di pagina è lungo $36 - 16 = 20$ bit (5 cifre esadecimale). Dobbiamo decidere come separarli in due indici. Sappiamo che ogni livello può utilizzare un massimo di 13 bit per l'indicizzazione (se vogliamo che sia contenuto in un frame). Abbiamo varie scelte:

1. La scelta più efficiente è quella di utilizzare appieno le tabelle di secondo livello, quindi i 20 bit di indice di pagina sarebbero separati in 7 bit (i più significativi) per indicizzare $2^7 = 128$ righe in una tabella di primo livello, e i 13 bit meno significativi per individuare la riga della tabella di secondo livello. L'indice **0x9E45B** sarebbe dunque suddiviso in:

Non strutturato	9		E		4		5		B			
Cifre binarie	1	0	0	1	1	1	0	0	1	0	0	
Strutturato	4		F		0		4		5		B	
	primo livello (7 bit)						secondo livello (13 bit)					

Osserviamo che il fatto di utilizzare un numero di bit non multiplo di 4 per il primo livello causa uno “sfasamento” nell'interpretazione dei valori binari. L'indice nella tabella di primo livello risulta dunque **0x4F**, quello nella tabella di secondo livello è **0x045B**.

2. In alternativa, è possibile optare per una soluzione bilanciata: dato che abbiamo a disposizione 20 bit, ne dedichiamo 10 al primo livello e 10 al secondo. Così facendo, ogni tabella di primo e secondo livello sarà occupata per $2^{10} = 1K = 1024$ righe, un ottavo della dimensione totale (il che non è ottimale, perché si rendono necessarie molte tabelle di secondo livello sottoutilizzate). Gli indici di primo e secondo livello dell'indirizzo di esempio saranno:

¹Notiamo, en passant, che l'indice di pagina **0x9E45B** richiede più di 13 bit, quindi individua una riga non contenuta in una ipotetica tabella a un solo frame.

<i>Non strutturato</i>	9	1	E	4	5	B
<i>Cifre binarie</i>	1 0 0 1 1 1 0 0 1 0 0 0 1 1 0 1 0 1 1 0 1 0 1 1					
<i>Strutturato</i>	2	7	9	0	5	B
	primo livello (10 bit)			secondo livello (10 bit)		

Gli indici di primo e secondo livello risultano dunque 0x279 e 0x05B.

3. Infine, per mantenere l'allineamento con le cifre esadecimali, si potrebbe proporre un indice di secondo livello da 12 bit (corrispondente dunque a un'occupazione di mezzo frame per ogni page table di secondo livello). In questo caso, l'indice di primo livello (gli 8 bit più significativi) è 0x9E, mentre l'indice di secondo livello è 0x45B.

12.4) La dimensione massima dell'area di scambio corrisponde alla dimensione della memoria virtuale, ovvero 64GB.

Se disponiamo di 4GB di memoria centrale, potremmo ridurre lo spazio di scambio a 60GB (considerato che 4GB di frame possono risiedere in memoria), ma in questo modo perderemmo alcune importanti occasioni di ottimizzazione (vedi l'uso del "dirty bit", che presuppone che uno stesso frame si trovi sia in memoria centrale, sia in memoria di massa).

Esercizio 13

Un computer ha uno spazio di indirizzamento reale a 16 bit e uno spazio di indirizzamento virtuale da 20 bit gestito attraverso paginazione su richiesta e memoria di massa. La MMU gestisce pagine da 1KB e contiene un TLB con 4 righe e rimpiazzo FIFO.

Un processo effettua i seguenti accessi in memoria (indirizzi logici):

0x00012 0x2468A 0x00112 0x00502 0x15404 0x75431 0x24411 0x003AB

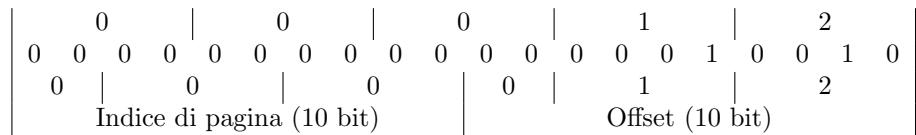
Inizialmente, nessuna pagina è caricata in memoria. Per ogni accesso, indicare se questo risulta in un page hit, un page miss o un page fault; mostrare l'indirizzo fisico generato dalla MMU e il contenuto del TLB dopo che l'accesso è avvenuto (ipotizzare liberamente il numero di frame corrispondente a ciascuna pagina).

Soluzione 13

Osservazione — Questo esercizio, pur non presentando difficoltà concettuali, richiede numerosi passaggi tecnici soggetti a errori. Un tipico esercizio da esame presenterà solo un sottoinsieme di queste difficoltà. Si suggerisce comunque di provare a svolgerlo nella sua interezza per assicurarsi di possedere le nozioni e le capacità tecniche necessarie.

Se la dimensione di pagina è di 1KB = 2^{10} B, allora i 10 bit meno significativi indicano l'offset nella pagina, e gli indirizzi logici saranno composti da 10 bit di pagina e 0 bit di offset.

- Il primo indirizzo, 0x00012, individua l'offset 0x012 della pagina 0x000:

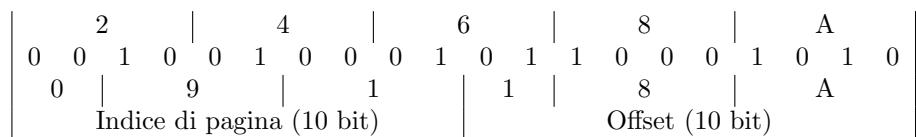


Dato che nessuna pagina è inizialmente caricata in memoria (e il TLB è vuoto), la richiesta causa un **page fault**. Dopo il caricamento della pagina 0 in memoria (supponiamo al frame 0), la page table contiene il riferimento al frame 0 nella riga di indice 0, e il TLB contiene l'associazione pagina-frame per velocizzare i riferimenti successivi:

Page table		TLB	
000	Frame 00	Pagina 000	Frame 00
⋮	...	—	—
	Righe vuote	—	—
	...	—	—

L'indirizzo fisico generato corrisponde dunque all'offset 0x012 (10 bit) del frame 0x00 (6 bit), ovvero 0x0012.

- Il secondo indirizzo, 0x2468A, individua l'offset 0x18A della pagina 0x091:



La pagina 0x091 non è ancora caricata in memoria (la riga corrispondente della page table è vuota), quindi viene generato un altro **page fault** e la pagina è caricata in un nuovo frame (supponiamo il frame 01); quindi il riferimento al frame 01 viene inserito alla riga 0x091 della page table. L'associazione tra la pagina e il frame viene anche registrata nel TLB:

Page table		TLB	
000	Frame 00	Pagina 000	Frame 00
:	...	Pagina 091	Frame 01
091	Frame 01	—	—
:	...	—	—

L'indirizzo fisico generato corrisponde dunque all'offset 0x18A (10 bit) del frame 0x01 (6 bit), ovvero 0x058A, come si può vedere dalla seguente tabella:

Indirizzo base (6 bit)						Offset (10 bit)									
0		1		1		8		A							
0	0	0	0	0	1	0	1	1	0	0	0	1	0	1	0
0					5			8							A

3. Il terzo indirizzo, 0x00112, individua l'offset 112 della pagina 000. La pagina 000 si trova nel TLB (**page hit**), quindi la MMU conosce già l'indirizzo base del frame corrispondente, 0x00, e può generare subito l'indirizzo fisico corrispondente 0x0112.
4. Il quarto indirizzo, 0x00502, individua l'offset 0x102 della pagina 0x001. La pagina non si trova in memoria (**page fault**), quindi viene caricata (supponiamo nel frame 02) e i suoi riferimenti vengono impostati nella page table e in una riga libera del TLB:

Page table		TLB	
000	Frame 00	Pagina 000	Frame 00
001	Frame 02	Pagina 091	Frame 01
:	...	Pagina 001	Frame 02
091	Frame 01	—	—
:	...		

L'indirizzo fisico generato dalla MMU è dunque l'offset 0x102 del frame 0x02, quindi 0x0902.

5. Il quinto indirizzo, 0x15404, individua l'offset 0x004 della pagina 0x055. Nuovamente, la pagina non è presente in memoria (**page fault**); supponiamo che venga caricata nel frame 03. Lo stato risultante è il seguente:

Page table		TLB	
000	Frame 00	Pagina 000	Frame 00
001	Frame 02	Pagina 091	Frame 01
:	...	Pagina 001	Frame 02
055	Frame 03	Pagina 055	Frame 03
:	...		
091	Frame 01		
:	...		

L'indirizzo fisico generato dalla MMU è dunque l'offset 0x004 del frame 0x03, quindi 0x0C04.

6. Il sesto indirizzo, 0x75431, individua l'offset 0x031 della pagina 0x1D5. La pagina non è presente in memoria (**page fault**); supponiamo che venga caricata nel frame 04. Dato che il TLB è pieno, il nuovo riferimento va a sostituire quello contenuto nella prima riga (l'esercizio suppone una sostituzione FIFO):

Page table		TLB	
000	Frame 00	Pagina 105	Frame 04
001	Frame 02	Pagina 091	Frame 01
:	...	Pagina 001	Frame 02
055	Frame 03	Pagina 055	Frame 03
:	...		
091	Frame 01		
:	...		
105	Frame 04		
:	...		

L'indirizzo fisico generato dalla MMU è l'offset 0x031 del frame 0x04, quindi 0x1031.

7. Il settimo indirizzo, 0x24411, individua l'offset 0x011 della pagina 0x091. La pagina è elencata nel TLB (**page hit**). L'indirizzo fisico generato dalla MMU è l'offset 0x011 del frame 0x01, quindi 0x0411.
8. L'ottavo indirizzo, 0x003AB, individua l'offset 0x3AB della pagina 0x000. La pagina non è elencata nel TLB, ma è presente in memoria (**page miss**). Il riferimento al frame 00 viene dunque reinserito nel TLB nella seconda riga:

Page table		TLB	
000	Frame 00	Pagina 105	Frame 04
001	Frame 02	Pagina 000	Frame 00
:	...	Pagina 001	Frame 02
055	Frame 03	Pagina 055	Frame 03
:	...		
091	Frame 01		
:	...		
105	Frame 04		
:	...		

L'indirizzo fisico generato dalla MMU è l'offset 0x3AB del frame 0x00, quindi 0x03AB.

Esercizio 14

Un computer dispone di un sistema di memoria virtuale a paginazione con 4 frame fisici allocabili a 8 pagine virtuali. Per il rimpiazzo delle pagine, il sistema utilizza registri a scorrimento a 4 bit. A ogni accesso a una pagina, il bit più significativo viene posto a 1, e ogni 50ms i registri vengono fatti scorrere verso destra di una posizione. All'istante $t = 0$, subito dopo lo scorrimento, la page table ha il seguente contenuto:

Pagina	Frame	Reference
0		
1		
2	1	0110
3	3	0010
4	0	0101
5		
6	2	0111
7		

14.1) In caso di page fault, quale frame verrebbe rimpiazzato?

Successivamente, la CPU effettua i seguenti accessi (solo il primo accesso di ogni periodo è indicato):

t (ms)	Pagina
10	3
15	1
30	3
55	2
70	5

14.2) Descrivere il contenuto della page table dopo ogni accesso (è sufficiente riportare le righe associate ai frame fisici).

Soluzione 14

14.1) Viene scelto il frame con il valore di riferimento inferiore (il frame 3).

14.2) All'istante $t = 10$ ms il bit di riferimento della pagina 3 viene posto a 1:

Pagina	Frame	Reference
3	3	1010

All'istante $t = 15$ ms, il riferimento alla pagina 1 causa un page fault, e viene sacrificata la pagina 4, recuperando il frame 0; i bit di riferimento vengono azzerati, e il più significativo viene posto a 1:

Pagina	Frame	Reference
1	0	1000
4		

All'istante $t = 30$ ms, il nuovo accesso alla pagina 3 non cambia nulla nella tabella, visto che il bit di riferimento più significativo è già stato posto a 1 in precedenza. All'istante $t = 50$ ms scatta lo scorrimento periodico dei bit di riferimento. Le righe associate a dei frame divengono:

Pagina	Frame	Reference
1	0	0100
2	1	0011
3	3	0101
6	2	0011

All'istante $t = 55$ ms, l'accesso alla pagina 2 causa l'impostazione del suo bit di riferimento:

Pagina	Frame	Reference
2	1	1011

Infine, all'istante $t = 70\text{ms}$ la richiesta della pagina 5 causa un page fault e il sacrificio della pagina 6, che libera il frame 2. La tabella finale risulta dunque:

Pagina	Frame	Reference
0		
1	0	0100
2	1	1011
3	3	0101
4		
5	2	1000
6		
7		

Esercizio 15

Lo spazio di indirizzamento di un sistema a paginazione virtuale utilizza indirizzi fisici a 14 bit (completamente mappati in memoria) e indirizzi logici a 16 bit, ed è organizzato in pagine da 4KB. Si supponga che la page table sia interna alla MMU.

15.1) Indicare le dimensioni del sistema. Quanta memoria fisica e quanta virtuale? Quanti bit vengono usati per l'offset, quanti per il numero di pagina? Quante pagine virtuali sono indirizzabili e quanti frame fisici sono disponibili? Quante righe servono, al massimo, per la page table?

Partendo con la memoria vuota (paginazione su richiesta pura), la CPU effettua i seguenti accessi (da leggere per colonna):

0x3494	0x4414
0x2519	0x3EFB
0x1433	0x2C2A
0x06F8	0x1708
0x36CC	0x0001
0x2122	0x452E

15.2) Descrivere l'evoluzione della page table, supponendo che per il rimpiazzo dei frame il SO utilizzi la tecnica FIFO. Mostrare, in particolare, la traduzione dei primi due indirizzi logici in indirizzi fisici. Qual è la page fault ratio alla fine dei 12 accessi in memoria?

15.3) Ripetere il punto precedente assumendo che il frame di indirizzo inferiore non sia disponibile. Qual è la page fault ratio in questo caso? Si è verificata l'anomalia di Bélády?

15.4) Utilizzare un diverso algoritmo di rimpiazzo nelle condizioni del punto 15.3. Confrontare la page fault ratio.

Esercizio 16

Un disco rigido da 200 tracce ha un seek time di 0,5ms/traccia. Il sistema operativo gestisce una coda di richieste di accesso.

All'istante $t = 0$ la testina si trova alla traccia 100 e sono in coda 4 richieste per le tracce 20, 70, 180 e 100, arrivate in precedenza in quest'ordine.

Successivamente, arrivano le seguenti richieste:

t (ms)	traccia
13	100
20	80
45	150
65	130
100	90

Supponendo che i tempi di latenza e trasferimento siano trascurabili, descrivere l'ordine in cui le richieste sono soddisfatte e in quali istanti di tempo per i seguenti algoritmi di scheduling:

1. FCFS;
2. SSTF;
3. SCAN, supponendo che la testina inizi muovendosi verso le tracce di indice minore;
4. SCAN, supponendo che la testina inizi muovendosi verso le tracce di indice maggiore.

Infine, per ciascun algoritmo registrare la richiesta che ha comportato il maggior tempo di attesa.

Soluzione 16

Nel caso dello scheduling FCFS, il percorso della testina è il seguente:

t (ms)	Traccia di partenza	Traccia di arrivo	Tempo di percorrenza (ms)	Nuove richieste
0	100	20	40	100, 80
40	20	70	25	150, 130
65	70	180	55	90
120	180	100	40	
160	100	100	0	
160	100	80	10	
170	80	150	35	
205	150	130	10	
215	130	90	20	
235	90	(destinazione finale)		

Ecco la soluzione per lo scheduling SSTF:

t (ms)	Traccia di partenza	Traccia di arrivo	Tempo di percorrenza (ms)	Nuove richieste
0	100	100	0	
0	100	70	15	100
15	70	100	15	80
30	100	80	10	
40	80	20	30	150, 130
70	20	130	55	90
125	130	150	10	
135	150	180	15	
150	180	90	45	
195	90	(destinazione finale)		

Osserviamo come, in particolare, la richiesta della traccia 180 sia stata ritardata rispetto allo scheduling FCFS (rischio di starvation delle tracce estreme).

Scheduling SCAN, assumendo che la testina stia inizialmente muovendo verso tracce basse:

t (ms)	Traccia di partenza	Traccia di arrivo	Tempo di percorrenza (ms)	Nuove richieste
0	100	100	0	
0	100	70	15	100
15	70	20	25	80
40	20	80	30	150, 130
70	80	100	10	
80	100	130	15	
95	130	150	10	90
105	150	180	15	
120	180	90	45	
165	90	(destinazione finale)		

Scheduling SCAN, assumendo che la testina stia inizialmente muovendo verso tracce alte:

t (ms)	Traccia di partenza	Traccia di arrivo	Tempo di percorrenza (ms)	Nuove richieste
0	100	100	0	
0	100	180	40	100, 80
40	180	100	40	150, 130
80	100	80	10	
90	80	70	5	
95	70	20	25	90
120	20	90	35	
155	90	130	20	
175	130	150	10	
185	150	(destinazione finale)		

Domande aperte

In questa sezione si presentano alcune domande a risposta aperta, simili a quelle che potrebbero comparire in un compito scritto accanto a esercizi numerici.

Esercizio 17

17.1) Illustrare brevemente gli algoritmi SSTF e SCAN per lo scheduling degli accessi a un disco rigido.

17.2) Mostrare una sequenza di richieste per la quale l'algoritmo SSTF porta a starvation di una delle richieste.

17.3) Fornire una ragione (in termini intuitivi) per la quale SCAN non può portare a starvation di una richiesta. Quale potrebbe essere il caso peggiore (tempo di attesa massimo per una richiesta)?

Suggerimento — *Assumere che il solo tempo non trascurabile sia il seek time. Se può fare comodo, supporre che più richieste alla stessa traccia, se servite consecutivamente, non richiedano tempi aggiuntivi.*

Esercizio 18

Discutere brevemente i vantaggi e gli svantaggi dei tre metodi di allocazione dello spazio disco considerati a lezione (contigua, a lista concatenata, indicizzata) nei seguenti scenari:

18.1) Accesso sequenziale in sola lettura a un solo grande file (esempio: compilazione di un grosso file sorgente);

18.2) Accesso casuale in sola lettura a un solo grande file (esempio: tabella di database);

18.3) Operazioni di scrittura in append su un solo un grande file (esempio: log di un server web molto trafficato);

18.4) Creazione e cancellazione ripetuta di molti piccoli file (esempio: utilities del sistema operativo).

Esercizio 19

Consideriamo un sistema a paginazione su richiesta (on-demand paging), in cui il grado di multiprogrammazione (numero di processi che si alternano nell'uso della CPU) è attualmente 4.

Per ciascuno dei seguenti esiti della misurazione dell'utilizzo della CPU e del disco di swap, illustrare cosa sta verosimilmente accadendo, discutere se la paginazione sia d'aiuto o meno e cosa si può fare per aumentare l'uso della CPU.

19.1) Utilizzo della CPU: 15%; occupazione del disco di swap: 95%.

19.2) Utilizzo della CPU: 90%; occupazione del disco di swap: 5%.

19.3) Utilizzo della CPU: 15%; occupazione del disco di swap: 5%.

Esercizio 20

20.1) Presentare una condizione di deadlock che coinvolga almeno tre processi in attesa di risorse ad accesso mutuamente esclusivo. Rappresentare graficamente la situazione utilizzando un grafo di allocazione delle risorse.

20.2) Descrivere una sequenza di richieste che, a partire da una situazione iniziale a risorse libere, porta alla situazione di deadlock descritta al punto 20.1.

20.3) Descrivere sommariamente (a parole o pseudocodice) un algoritmo per evitare queste situazioni di deadlock. Chiarire su quali assunzioni si basa, e applicarlo alla sequenza descritta al punto 20.2 mostrando come il deadlock viene evitato.