

INDICE

1 – INTRODUZIONE.....	8
1.1 GENERALITÀ.....	8
1.1.1 COS’È UN SISTEMA OPERATIVO.....	8
1.1.2 COMPONENTI DI UN SISTEMA DI CALCOLO.....	8
1.1.3 SISTEMI MAINFRAME.....	9
1.1.4 CONFIGURAZIONE DELLA MEMORIA PER UN SISTEMA DI LOTTI.....	9
1.2 STRUTTURA DEL SISTEMA OPERATIVO.....	10
1.2.1 SISTEMA MULTIPROGRAMMATO.....	10
1.2.2 SISTEMI A PARTIZIONE DEL TEMPO D’ELABORAZIONE.....	10
1.2.3 SISTEMI DA SCRIVANIA.....	11
1.2.4 SISTEMI PARALLELI (o MULTIPROCESSORE).....	11
1.2.5 SISTEMI DISTRIBUITI.....	11
1.2.6 BATTERIE DI SISTEMI (SISTEMI CLUSTER).....	12
1.2.7 SISTEMI DI ELABORAZIONE IN TEMPO REALE.....	12
1.2.8 SISTEMI PALMARI.....	12
1.3 AMBIENTI D’ELABORAZIONE.....	12
2 – STRUTTURE SISTEMICHE DI CALCOLO.....	13
2.1 GENERALITÀ.....	13
2.2 FUNZIONAMENTO DI UN SISTEMA DI CALCOLO.....	13
2.2.1 FUNZIONI COMUNI DEI SEGNALI DI INTERRUZIONI.....	14
2.2.2 GESTIONE DELL’INTERRUZIONE.....	14
2.2.3 ACCESSO DIRETTO ALLA MEMORIA.....	16
2.2.4 STRUTTURA DELLA MEMORIA.....	17
2.2.5 GERARCHIE DELLE MEMORIE.....	17
2.7 ARCHITETTURE DI PROTEZIONE.....	19
3 – STRUTTURE DEI SISTEMI OPERATIVI.....	21
3.1 GESTIONE DEI PROCESSI.....	21
3.2 GESTIONE DELLA MEMORIA CENTRALE.....	22
3.3 GESTIONE DELLA MEMORIA DI MASSA.....	22
3.3.1 GESTIONE DEI FILE.....	22
3.3.2 GESTIONE DEL SISTEMA DI I/O.....	22
3.3.3 GESTIONE DELLA MEMORIA SECONDARIA.....	23
3.3.4 CACHE.....	23
3.4 RETI (SISTEMI DISTRIBUITI).....	23
3.5 SISTEMA DI PROTEZIONE.....	24
3.6 INTERPRETE DEI COMANDI.....	24
3.7 SERVIZI DI UN SISTEMA OPERATIVO.....	24
3.8 CHIAMATE DEL SISTEMA.....	26
3.9 STRUTTURA DEL SISTEMA OPERATIVO.....	27
3.9.1 ESECUZIONE NELL’MS-DOS.....	27
3.9.2 SISTEMA OPERATIVO UNIX.....	28
3.9.3 PROGRAMMI DI SISTEMA.....	29
3.9.4 IL METODO STRATIFICATO.....	29
3.9.5 STRUTTURA STRATIFICATA DELL’OS/2.....	30
3.9.6 MICRKERNEL (ORIENTAMENTO A MICRONEUCLEO).....	31

3.9.7 MACCHINE VIRTUALI.....	31
3.9.8 MACCHINA VIRTUALE JAVA.....	32
3.10 PROGETTAZIONE E REALIZZAZIONE DI UN SISTEMA OPERATIVO.....	33
3.10.1 SCOPI DELLA PROGETTAZIONE.....	33
3.10.2 MECCANISMI E CRITERI.....	33
3.10.3 REALIZZAZIONE.....	33
3.10.4 GENERAZIONE DEI SISTEMI OPERATIVI.....	34
4 – PROCESSI.....	34
4.1 CONCETTO DI PROCESSO.....	34
4.1.1 PROCESSO.....	34
4.1.2 STATO DEL PROCESSO.....	35
4.1.3 BLOCCO DI CONTROLLO DEI PROCESSI.....	35
4.2 SCHEDULING DEI PROCESSI.....	36
4.2.1 CODE SCHEDULING.....	37
4.2.2 SCHEDULER.....	38
4.2.3 CAMBIO DI CONTESTO.....	39
4.3 OPERAZIONI SUI PROCESSI.....	40
4.3.1 CREAZIONE DI UN PROCESSO.....	40
4.3.2 TERMINAZIONE DI UN PROCESSO.....	41
4.4 COMUNICAZIONE TRA PROCESSI.....	41
4.4.1 SISTEMI A MEMORIA CONDIVISA.....	42
4.4.2 SISTEMI A SCAMBIO DI MESSAGGI.....	43
4.4.3 NAMING.....	44
4.4.4 SINCRONIZZAZIONE.....	45
4.4.5 CODE DI MESSAGGI.....	45
4.5 COMUNICAZIONE NEI SISTEMI CLIENT-SERVER.....	45
4.5.1 SOCKET.....	45
4.5.2 CHIAMATE DI PROCEDURE REMOTE.....	46
4.5.3 INVOCAZIONE DEI METODI REMOTI.....	46
5 – THREADS.....	47
5.1 INTRODUZIONE.....	47
5.1.1 MOTIVAZIONI DEI THREAD.....	47
5.1.2 VANTAGGI.....	48
5.2 PROGRAMMAZIONE MULTICORE.....	48
5.3 PREEMPTION (PRELAZIONE).....	49
5.4 MODELLI DI SUPPORTO AL MULTITHREADING.....	49
5.4.1 MODELLO DA MOLTI A UNO.....	49
5.4.2 MODELLO DA UNO A UNO.....	50
5.4.3 MODELLI DA MOLTI A MOLTI.....	50
5.5 LIBRERIE DI THREAD.....	51
5.5.1 PTHREADS.....	51
5.5.2 THREAD IN WINDOWS.....	52
5.5.3 THREAD JAVA.....	53
5.6 PROBLEMATICA DI PROGRAMMAZIONE MULTITHREAD.....	53
5.6.1 CHIAMATE DI SISTEMA FORK() ED EXEC().....	53
5.6.2 CANCELLAZIONE DEI THREAD.....	54
5.6.3 GESTIONE DEI SEGNALI.....	54
5.6.4 DATI SPECIFICI DEI THREAD.....	54
5.7 GRUPPI DI THREAD.....	54

6 – SCHEDULING DELLA CPU.....55

6.1 CONCETTI FONDAMENTALI.....	55
6.1.1 CICLICITÀ DELLE FASI DI ELABORAZIONE E DI I/O.....	55
6.1.2 SCHEDULER DELLA CPU.....	56
6.1.3 SCHEDULING CON PRELAZIONE.....	56
6.1.4 DISPATCHER.....	57
6.2 CRITERI DI SCHEDULING.....	57
6.3 ALGORITMI DI SCHEDULING.....	58
6.3.1 SCHEDULING IN ORDINE DI ARRIVO.....	58
6.3.2 SCHEDULING SHORTEST-JOB-FIRST.....	59
6.3.3 SCHEDULING CON PRIORITÀ.....	60
6.3.4 SCHEDULING CIRCOLARE.....	62
6.3.5 SCHEDULING A CODE MULTIPLE.....	63
6.3.6 SCHEDULING A CODE MULTILIVELLO CON RETROAZIONE.....	64
6.4 SCHEDULING PER SISTEMI MULTIPROCESSORE.....	65
6.4.1 APPROCCI ALLO SCHEDULING PER MULTIPROCESSORI.....	65
6.5 SCHEDULING REAL-TIME DELLA CPU.....	66
6.6 VALUTAZIONE DEGLI ALGORITMI.....	66

7 – SINCRONIZZAZIONE DEI PROCESSI.....67

7.1 INTRODUZIONE.....	67
7.2 ESEMPIO: MEMORIA LIMITATA – SOLUZIONE CON MEMORIA CONDIVISA.....	67
7.3 PROBLEMA DELLA SEZIONE CRITICA.....	70
7.4 ALGORITMI.....	71
7.4.1 ALGORITMO 1.....	71
7.4.2 ALGORITMO 2.....	71
7.4.3 ALGORITMO 3.....	72
7.4.5 ALGORITMO DEL FORNAIO.....	72
7.5 ARCHITETTURE DI SINCRONIZZAZIONE.....	73
7.6 SEMAFORI.....	74
7.6.1 SEZIONE CRITICA DI N PROCESSI.....	74
7.6.2 IMPLEMENTAZIONE DEI SEMAFORI.....	75
7.6.3 SEMAFORI COME STRUMENTI GENERALI DI SINCRONIZZAZIONE.....	76
7.6.4 STALLO DEI PROCESSI E ATTESA INDEFINITA.....	76
7.6.5 TIPI DI SEMAFORI.....	77
7.6.6 REALIZZAZIONE DI S COME SEMAFORO BINARIO.....	77
7.7 PROBLEMI TIPICI DI SINCRONIZZAZIONE.....	78
7.7.1 PRODUTTORE/CONSUMATORE CON MEMORIA LIMITATA.....	78
7.7.2 PROBLEMA DEI LETTORI-SCRITTORI.....	79
7.7.3 PROBLEMA DEI CINQUE FILOSOFI.....	80
7.8 REGIONI CRITICHE.....	81
7.9 MONITOR.....	82
7.10 SINCRONIZZAZIONE IN LINUX.....	83
7.11 SINCRONIZZAZIONE IN WINDOWS.....	83

8 – STALLO DEI PROCESSI.....84

8.1 MODELLO DEL SISTEMA.....	84
8.2 CARATTERIZZAZIONE DELLE SITUAZIONI DI STALLO.....	84
8.2.1 CONDIZIONI NECESSARIE.....	85
8.2.2 GRAFO DI ASSEGNAZIONE DELLE RISORSE.....	85

8.3 METODI PER LA GESTIONE DELLA SITUAZIONE DI STALLO.....	87
8.4 PREVENIRE LE SITUAZIONI DI STALLO.....	87
8.4.1 MUTUA ESCLUSIONE.....	87
8.4.2 POSSESSO E ATTESA.....	87
8.4.3 ASSENZA DI PRELAZIONE.....	88
8.4.4 ATTESA CIRCOLARE.....	88
8.5 EVITARE LE SITUAZIONI DI STALLO.....	89
8.5.1 STATO SICURO.....	89
8.5.2 ALGORITMO CON GRAFO DI ASSEGNAZIONE DELLE RISORSE.....	89
8.5.3 ALGORITMO DEL BANCHIERE (da leggere, non lo chiede nei dettagli).....	90
8.5.3.1 ALGORITMO DI VERIFICA DELLA SICUREZZA.....	91
8.5.3.2 ALGORITMO DI RICHIESTA DELLE RISORSE.....	91
8.5.3.3 ESEMPIO.....	92
8.6 RILEVAMENTO DELLE SITUAZIONI DI STALLO.....	93
8.6.1 ISTANZA SINGOLA DI CIASCUN TIPO DI RISORSA.....	93
8.6.2 PIÙ IstanTE PER CIASCUN TIPO DI RISORSA.....	94
8.6.3 USO DELL'ALGORITMO DI RILEVAMENTO.....	95
8.7 RIPRISTINO DI SITUAZIONI DI STALLO.....	95
8.7.1 RIPRISTINO DEI PROCESSI.....	96
8.7.2 PRELAZIONE DELLE RISORSE.....	96
8.8 APPROCCIO COMBINATO PER LA GESTIONE DELLO STALLO.....	96

9 – GESTIONE DELLA MEMORIA..... **97**

9.1 INTRODUZIONE.....	97
9.1.1 ASSOCIAZIONE DEGLI INDIRIZZI.....	97
9.1.2 SPAZI DI INDIRIZZI LOGICI E FISICI.....	98
9.1.3 CARICAMENTO DINAMICO.....	99
9.1.4 LINKING DINAMICO (COLLEGAMENTO DINAMICO).....	100
9.1.5 SOVRAPPOSIZIONE DI SEZIONI (OVERLAY).....	100
9.1.6 AVVICENDAMENTO DEI PROCESSI (SWAPPING).....	100
9.2 ALLOCAZIONE CONTIGUA DELLA MEMORIA.....	101
9.2.1 PROTEZIONE DELLA MEMORIA.....	102
9.2.2 ALLOCAZIONE DELLA MEMORIA.....	102
9.2.3 FRAMMENTAZIONE.....	103
9.3 PAGINAZIONE.....	104
9.3.1 METODO DI BASE.....	104
9.3.2 ARCHITETTURA DI PAGINAZIONE.....	106
9.3.3 PROTEZIONE DELLA MEMORIA.....	107
9.4 STRUTTURA DELLA TABELLA DELLE PAGINE.....	108
9.4.1 PAGINAZIONE GERARCHICA.....	108
9.4.2 TABELLA DELLE PAGINE DI TIPO HASH.....	110
9.4.3 TABELLA DELLE PAGINE INVERTITA.....	110
9.4.4 PAGINE CONDIVISE.....	111
9.5 SEGMENTAZIONE.....	112
9.5.1 ARCHITETTURA DI SEGMENTAZIONE.....	112

10 – MEMORIA VIRTUALE..... **113**

10.1 INTRODUZIONE.....	113
10.2 PAGINAZIONE SU RICHIESTA.....	114
10.3 COPIATURA SU SCRITTURA.....	116
10.4 ASSOCIAZIONE DEI FILE ALLA MEMORIA.....	116

10.5 SOSTITUZIONE DELLE PAGINE.....	117
10.6 ALGORITMI DI SOSTITUZIONE DELLE PAGINE.....	118
10.6.1 ALGORITMO FIFO (FIRST-IN-FIRST-OUT).....	118
10.6.2 SOSTITUZIONE OTTIMALE DELLE PAGINE.....	119
10.6.3 SOSTITUZIONE DELLE PAGINE USATE MENO RECENTEMENTE (LRU).....	120
10.7 SOSTITUZIONE DELLE PAGINE PER APPROXIMAZIONE A LRU.....	121
10.7.1 ALGORITMO CON SECONDA CHANCE.....	122
10.7.2 SOSTITUZIONE DELLE PAGINE BASATA SU CONTEGGIO.....	123
11 – INTERFACCIA DEL FILE SYSTEM.....	123
11.1 CONCETTO DI FILE.....	123
11.1.1 ATTRIBUTI DEI FILE.....	123
11.1.2 OPERAZIONI SUI FILE.....	124
11.2 METODI DI ACCESSO.....	125
11.3 STRUTTURA DELLA DIRECTORY E DEL DISCO.....	126
11.3.1 DIRECTORY A UN LIVELLO.....	127
11.3.2 DIRECTORY A DUE LIVELLI.....	127
11.3.3 DIRECTORY CON STRUTTURA AD ALBERO.....	128
11.3.4 DIRECTORY CON STRUTTURA A GRAFO ACICLICO.....	129
11.3.5 DIRECTORY CON STRUTTURA A GRAFO GENERALE.....	130
11.4 MONTAGGIO DI UN FILE SYSTEM.....	131
11.5 CONDIVISIONE DI FILE.....	132
11.5.1 PROTEZIONE.....	132
11.5.2 CONTROLLO DEGLI ACCESSI.....	132
12 – REALIZZAZIONE DEL FILE SYSTEM.....	133
12.1 STRUTTURA DEL FILE SYSTEM.....	133
12.1.1 STRUTTURE DEL FILE SYSTEM CHE SI MANTENGONO IN MEMORIA.....	134
12.1.2 FILE SYSTEM VIRTUALI.....	134
12.2 REALIZZAZIONE DELLE DIRECTORY.....	135
12.2.1 LISTA LINEARE.....	135
12.2.2 TABELLA HASH.....	135
12.3 METODI DI ASSEGNAZIONE.....	136
12.3.1 ALLOCAZIONE CONTIGUA.....	136
12.3.2 ALLOCAZIONE CONCATENATA.....	137
12.3.3 ALLOCAZIONE INDICIZZATA.....	138
12.4 GESTIONE DELLO SPAZIO LIBERO.....	140
12.4.1 VETTORE DI BIT.....	140
12.4.2 LISTA CONCATENATA.....	140
12.5 EFFICIENZA E PRESTAZIONI.....	141
12.5.1 I/O SENZA BUFFER CACHE UNIFICATA.....	141
12.5.2 I/O CON BUFFER CACHE UNIFICATA.....	141
12.6 RIPRISTINO.....	142
12.7 NFS.....	142
12.7.1 PROTOCOLLO DI MONTAGGIO.....	143
12.7.2 PROTOCOLLO NFS.....	143
12.7.3 ARCHITETTURA NFS.....	143
12.7.4 TRADUZIONE DEI NOMI DI PERCORSO.....	144
12.7.5 FUNZIONAMENTO REMOTO.....	144

13 – MEMORIA SECONDARIA E TERZIARIA.....145

13.1 STRUTTURA DEI DISPOSITIVI DI MEMORIZZAZIONE.....	145
13.1.1 DISCHI MAGNETICI.....	145
13.1.2 DISCHI A STATO SOLIDO.....	145
13.2 STRUTTURA DEI DISCHI.....	146
13.2.1 PARTIZIONAMENTO DI UN HARD DISK.....	147
13.2.2 FRAMMENTAZIONE E DE-FRAMMENTAZIONE.....	147
13.3 SCHEDULING DELL'HARD DISK.....	148
13.3.1 SCHEDULING IN ORDINE DI ARRIVO - FCFS.....	148
13.3.2 SCHEDULING - SSTF.....	149
13.3.3 SCHEDULING - SCAN.....	149
13.3.4 SCHEDULING – C-SCAN.....	150
13.3.5 SCHEDULING LOOK.....	150
13.3.6 SCELTA DI UN ALGORITMO DI SCHEDULING.....	151
13.4 STRUTTURE RAID.....	151
13.4.1 LIVELLI RAID.....	152

1 – INTRODUZIONE

1.1 GENERALITÀ

1.1.1 COS'È UN SISTEMA OPERATIVO

È un insieme di programmi che funge da intermediario tra un utente e gli elementi fisici di un calcolatore (hardware). Deve assegnare le risorse necessarie a un programma per poterlo rendere eseguibile. Bisogna però considerare che, contemporaneamente, il sistema operativo deve fare tante altre cose. Proprio per questo non possiamo dare una vera e propria definizione netta.

Obiettivi di un sistema operativo:

- Eseguire programmi d'applicazione (veri e propri programmi che vengono lanciati dall'utente) e rendere più facile la risoluzione dei problemi che gli utenti devono affrontare;
- Rendere il sistema conveniente da utilizzare.

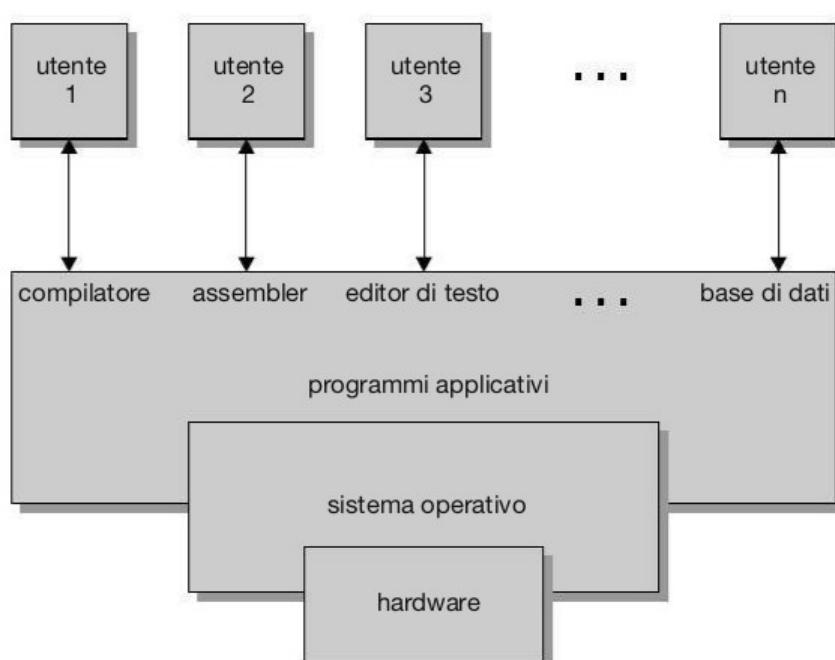
Tutte queste operazioni devono garantire un utilizzo efficiente dei dispositivi fisici.

1.1.2 COMPONENTI DI UN SISTEMA DI CALCOLO

Un sistema di elaborazione si può suddividere in quattro componenti:

- **Dispositivi fisici (hardware)**: sono lo strato di livello più basso. Forniscono le risorse fondamentali per tutto il resto dell'elaborazione (CPU, memoria, I/O);
- **Sistema operativo**: controlla e coordina l'uso dei dispositivi da parte dei programmi d'applicazione per gli utenti;
- **Programmi d'applicazione**: sono i programmi veri e propri che l'utente lancia (editor di testo, fogli di calcolo, compilatori e browser Web);
- **Utenza**: si tratta dello strato più alto. Più essere rappresentata da persone, macchine o altri calcolatori.

Tutto quello che abbiamo detto possiamo visualizzarlo con questo schema:



Detto questo, possiamo definire un sistema operativo come:

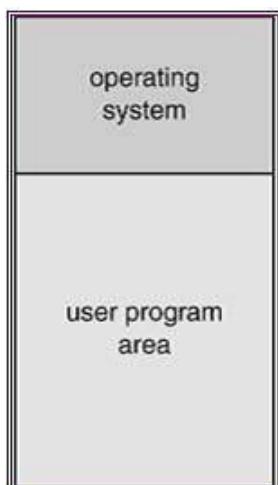
- Assegnatore di risorse: gestisce e assegna risorse.
- Programma di controllo: controlla l'esecuzione dei programmi. Se, ad esempio, durante l'esecuzione un programma prende il sopravvento, non deve bloccare il resto della macchina. Il sistema operativo deve essere in grado di bloccare quel programma.
- Nucleo (kernel): tutte queste operazioni descritte, vengono svolte perché all'interno del computer c'è qualcosa che permette l'operatività del sistema operativo. Questo qualcosa è il *kernel*. È l'unico programma che funziona sempre.

1.1.3 SISTEMI MAINFRAME

Sono sistemi principali (es. un centro di calcolo) il cui obiettivo è:

- Riduzione del tempo di elaborazione raggruppando insieme lavori con requisiti simili. Questo perché, se siamo in grado di raggruppare lavori con requisiti simili, siamo in grado di trattarli in maniera omogenea.
- Trasferimento automatico del controllo da un lavoro a quello successivo. Deve essere in grado di accodare una serie di processi, quindi deve sapere come iniziare e terminare un processo per iniziare quello successivo.
- Monitor residente: la capacità di realizzare un pacchetto di funzionalità che rendano sempre attiva la capacità di tenere sotto controllo il sistema.

1.1.4 CONFIGURAZIONE DELLA MEMORIA PER UN SISTEMA DI LOTTI

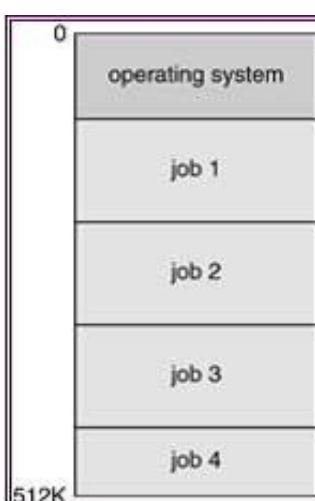


In questo caso abbiamo un blocco in cui c'è il sistema operativo e poi l'area per programmi utenti. Questa visione ci facilita molto la configurazione della nostra memoria in un sistema in cui da una parte c'è tutto il sistema operativo e dall'altra l'area per programmi utente.

Una peculiarità è che il sistema operativo si trova nella parte migliore della memoria. Bisogna scegliere il posto migliore perché, se il sistema operativo funziona male, fa funzionare male anche gli altri programmi.

1.2 STRUTTURA DEL SISTEMA OPERATIVO

1.2.1 SISTEMA MULTIPROGRAMMATO



Vediamo che il sistema operativo si trova nella parte alta (zona privilegiata). L'idea su cui si fonda questa tecnica è la seguente: il sistema operativo tiene contemporaneamente in memoria centrale diversi job. Nella memoria i job possono essere migliaia, nello schema ne sono stati messi solo alcuni. Più sono messi in basso e più aumenta la loro complessità dal punto di vista dell'indirizzamento (diventerà più lento arrivarcì).

Questo significa che, per arrivare ad esempio al "job 4", bisogna comporre un numero più complesso rispetto al "job 1".

Tutte queste cose appena descritte, vengono prese in considerazione dal sistema operativo. Le caratteristiche di un sistema multiprogrammato sono:

- Tutti i lavori che entrano nel sistema sono mantenuti in un gruppo che consiste di tutti i processi d'elaborazione presenti nel disco che attendono il caricamento nella memoria centrale;
- Gestione della memoria: il sistema deve ripartire la memoria tra i vari lavori in entrata;
- Scheduling della CPU: se in un dato momento più lavori sono pronti per essere eseguiti, il sistema deve scegliere quello da eseguire;
- Allocazione delle risorse

1.2.2 SISTEMI A PARTIZIONE DEL TEMPO D'ELABORAZIONE

La partizione del tempo d'elaborazione (**time sharing** o **multitasking**) è un'estensione logica della multiprogrammazione.

Permette la comunicazione diretta tra utente e sistema. L'utente impedisce le istruzioni direttamente al sistema operativo oppure a un programma, attraverso una tastiera, un mouse, un touch pad o un touch screen, e attende una risposta immediata tramite un dispositivo di output. Il tempo di risposta dovrebbe perciò essere breve, in genere meno di un secondo.

Un sistema operativo time sharing permette a più utenti di condividere contemporaneamente il calcolatore. Poiché le azioni e i comandi eseguiti in un sistema time sharing sono tendenzialmente brevi, a ciascun utente basta poter usare una piccola parte del tempo di calcolo della CPU. Il sistema passa rapidamente da un utente all'altro, quindi ogni utente ha l'impressione di disporre dell'intero calcolatore, che in realtà è condiviso da molti utenti.

Per assicurare a ciascun utente una piccola frazione del tempo di calcolo, un sistema operativo time sharing si avvale dello scheduling della CPU e della multiprogrammazione. Ciascun utente dispone di almeno un proprio programma in memoria.

1.2.3 SISTEMI DA SCRIVANIA

Questi sono:

- Personal computer: calcolatori utilizzati da un singolo utente;
- Dispositivi I/O: tastiera, mouse, schermo, piccole stampanti;

In questi sistemi ci deve essere una comodità e prontezza d'uso per l'utente. Devono essere in grado di adottare le tecnologie sviluppate per i grandi sistemi e possono funzionare con sistemi operativi diversi.

1.2.4 SISTEMI PARALLELI (o MULTIPROCESSORE)

Sono sistemi con più unità di elaborazione, con più CPU in stretta collaborazione. Un sistema parallelo è un sistema sul quale concorrono diversi processi in modo parallelo: questo avviene quando sono dipendenti tra di loro.

Sono sistemi strettamente connessi, infatti condividono il bus e talvolta il clock di sistema, la memoria e i dispositivi periferici. I vantaggi di questi sistemi sono:

- Maggiore capacità elaborativa (throughput): Aumentando il numero di unità d'elaborazione è possibile svolgere un lavoro maggiore in meno tempo.
- Economia di scala: I sistemi multiprocessore possono costare meno rispetto a più sistemi monoprocesso, poiché vengono condivisi periferiche, memorie di massa e sistemi di alimentazione.

- Incremento dell'affidabilità: Se le funzioni si possono distribuire adeguatamente tra più unità d'elaborazione, un guasto di una di esse non blocca il sistema, semplicemente lo rallenta; ciascuna delle unità d'elaborazione rimanenti assume su di sé una parte del lavoro che svolgeva l'unità d'elaborazione guasta; l'intero sistema non si ferma, ma funziona a una velocità ridotta.
La capacità di continuare a offrire un servizio proporzionale alla quantità di hardware ancora in funzione è detta degradazione controllata (graceful degradation). Taluni sistemi si spingono oltre la degradazione controllata e sono chiamati tolleranti ai guasti (fault-tolerant) perché continuano a funzionare ugualmente nonostante il guasto di un qualunque singolo componente.

I sistemi attualmente in uso sono di due tipi. Possono impiegare la:

- Multielaborazione simmetrica (SMP): ciascuna unità di elaborazione esegue un'identica copia del sistema operativo. Si possono eseguire molti processi contemporaneamente senza causare un rilevante calo delle prestazioni. È supportato dalla maggior parte dei moderni sistemi operativi.
- Multielaborazione asimmetrica (AMP): a ogni processore si assegna un compito specifico. Un processore principale controlla il sistema, gli altri attendono istruzioni dal processore principale oppure hanno compiti predefiniti. Questo schema definisce una relazione gerarchica; il processore principale organizza e assegna il lavoro ai processori secondari. È più comune nei sistemi di grandi dimensioni.

1.2.5 SISTEMI DISTRIBUITI

I sistemi distribuiti si basano sulle reti per realizzare le proprie funzioni. Una rete si può considerare, in parole semplici, come un canale di comunicazione tra due o più sistemi.

In questo sistema ciascuna unità di elaborazione ha la propria memoria locale e comunica con le altre per mezzo di linee di comunicazione di vario genere.

I suoi vantaggi sono:

- Condivisione risorse;
- Maggiore velocità;
- Affidabilità;
- Comunicazione;

Le reti si classificano secondo le distanze tra i loro nodi. Abbiamo:

- Local Area Network (LAN)
- Wide Area Network (WAN)
- Metropolitan Area Network (MAN)

Questi sistemi operativi possono essere di diversa natura:

- Client-Server: sono sistemi nei quali c'è un'asimmetria. Ci sono alcune macchine che fungono da servizio e altre che fungono da cliente.
- Peer-to-Peer: sono sistemi che coinvolgono punti della rete specifici.

1.2.6 BATTERIE DI SISTEMI (SISTEMI CLUSTER)

Sono un altro tipo di sistemi multiprocessore, basati sull'uso congiunto di più CPU, ma differiscono dai sistemi multiprocessore descritti nel paragrafo precedente per il fatto che sono composti di due o più calcolatori completi – detti nodi – collegati tra loro. Forniscono un'elevata disponibilità.

Si dividono in:

- **Batterie asimmetriche (asymmetric clustering)**: un calcolatore rimane nello stato di attesa attiva (*hot standby mode*) mentre l'altro esegue le applicazioni.
- **Batterie simmetriche (symmetric clustering)**: due o più calcolatori eseguono le applicazioni e allo stesso tempo si controllano reciprocamente; in questo modo si ottiene una maggiore efficienza, poiché si utilizzano meglio le risorse, ma ciò richiede che siano disponibili più applicazioni da eseguire.

1.2.7 SISTEMI DI ELABORAZIONE IN TEMPO REALE

Sono spesso impiegati nella gestione dei dispositivi di controllo per applicazioni specifiche (controllo di esperimenti scientifici, sistemi di controllo industriale, rappresentazione d'immagini in medicina...). Presenta vincoli di tempo fissati e ben definiti entro i quali si deve effettuare l'elaborazione.

Esistono due tipi di sistemi d'elaborazione in tempo reale:

- **Sistemi d'elaborazione in tempo reale stretto (hard real-time)**: memoria secondaria limitata o assente; dati sono memorizzati nella memoria a breve termine, o memoria di sola lettura (ROM).
- **Sistemi d'elaborazione in tempo reale debole (soft real-time)**: utilità limitata nel controllo industriale della robotica; utile in applicazioni (multimedia, realtà virtuale) che richiedono caratteristiche avanzate del sistema operativo.

1.2.8 SISTEMI PALMARI

Ci riferiamo fondamentalmente ai nostri telefoni cellulari. A causa delle piccole dimensioni, la maggior parte dei sistemi palmari dispone di:

- Memoria limitata
- Unità d'elaborazione lente
- Schermi piccoli

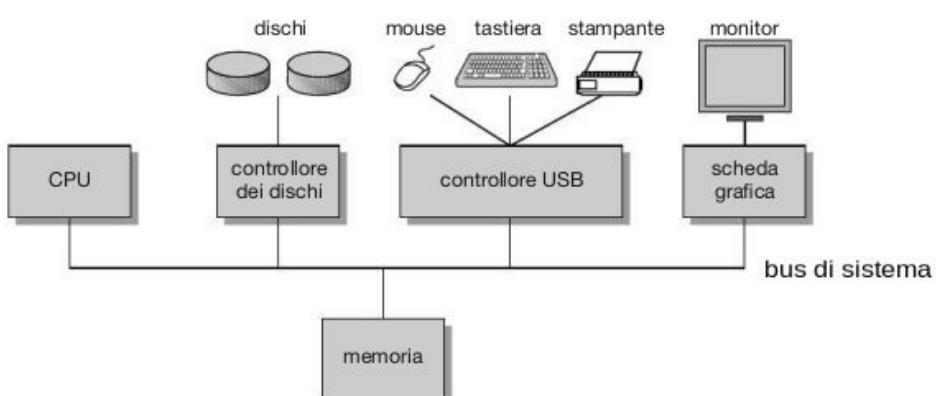
1.3 AMBIENTI D'ELABORAZIONE

- Elaborazione tradizionale.
- Elaborazione basata sul web: molte delle applicazioni che girano su rete hanno necessità di essere elaborate da diverse tipologie di piattaforme. L'elaborazione non viene elaborata sulla nostra macchina ma la sua esecuzione avviene su un server le cui parti possono trovarsi anche in posti diversi.
- Dispositivi d'elaborazione integrati.

2 – STRUTTURE SISTEMICHE DI CALCOLO

2.1 GENERALITÀ

Vediamo la struttura di un sistema di calcolo:



Un moderno calcolatore general-purpose è composto da una o più CPU e da un certo numero di controllori di dispositivi connessi attraverso un canale di comunicazione comune (*bus*) che permette l'accesso alla memoria condivisa dal sistema. Un controllore (*controller*) non è altro che un'interfaccia.

Queste unità hanno diverse velocità di trattamento dei dati (es. il disco tratta i dati più velocemente della stampante). Questi dispositivi, per comunicare tra di loro, necessitano di un'interfaccia che li metta d'accordo e che ci sia la possibilità di compensare le differenze di velocità e di compensare le regole di trattamento dei dati da parte di questi dispositivi. Da qui vediamo l'importanza dell'esistenza dei *controller*.

2.2 FUNZIONAMENTO DI UN SISTEMA DI CALCOLO

- I dispositivi di I/O e la CPU possono operare in modo concorrente: importante non confondere la concorrenza con il parallelismo. Questo perché, il parallelismo che noi possiamo avere dell'elaborazione delle informazioni riguarda essenzialmente dispositivi che sono tra di loro presumibilmente indipendenti, quindi non c'è bisogno che un dispositivo, per trattare i dati, debba comprendere cosa stia facendo l'altro dispositivo; invece qualora i dispositivi del nostro sistema di calcolo sono immersi tutti in un'unico sistema, dovranno tra di loro *gareggiare* per accaparrarsi le risorse (che non sono infinite).
- Ciascun controller si occupa di un particolare tipo di dispositivo fisico (per esempio, unità disco, dispositivi audio e unità video).
- Ciascun controller ha un buffer locale: abbiamo detto che ogni dispositivo ha una propria velocità, quindi è importante che un certo dispositivo abbia la possibilità di memorizzare (e quindi far transitare) temporaneamente i dati in modo tale che poi, con calma, li lavora.
- La CPU sposta i dati da/verso la memoria principale da/verso i buffer locali.
- L'I/O avviene dal dispositivo al buffer locale del controller.
- Quando un controller prende in carico i dati e li elabora, dove avvertire della fine e dell'inizio di questo processo. Diremo che vengono lanciati dei segnali (tramite un bus) di inizio e fine che vanno sotto il nome di **interruzioni**.

2.2.1 FUNZIONI COMUNI DEI SEGNALI DI INTERRUZIONI

Un segnale di interruzione deve causare il trasferimento del controllo all'appropriata procedura di servizio dell'evento a esso associato. L'architettura di gestione delle interruzioni deve anche salvare l'indirizzo dell'istruzione interrotta. Che significa: quando riceve un segnale d'interruzione, la CPU interrompe l'elaborazione corrente e trasferisce immediatamente l'esecuzione a una locazione fissa della memoria. Di solito, questa locazione contiene l'indirizzo iniziale della procedura di servizio dell'interruzione. Una volta completata l'esecuzione della procedura richiesta, la CPU riprende l'elaborazione precedentemente interrotta.

Un **segnale d'eccezione (trap)** può essere causato da un programma in esecuzione a seguito di un evento eccezionale oppure a seguito di una richiesta specifica effettuata da un programma utente. Un moderno sistema operativo è guidato dalle interruzioni (*interrupt driven*).

2.2.2 GESTIONE DELL'INTERRUZIONE

La cosa fondamentale che deve fare il sistema operativo è di memorizzare l'indirizzo di ritorno nella pila (*stack*) di sistema: se c'è un'interruzione dobbiamo ricordarci, quando l'interruzione sarà completata, in quale punto dell'algoritmo dobbiamo riprendere l'elaborazione.

Inoltre, il sistema operativo deve determinare quale tipo di interruzione si è verificato. Ne possiamo avere di due tipi:

- Polling: si tratta di un'interrogazione ciclica. Significa che ciclicamente vengono interrogati i dispositivi per chiedere a ognuno di essi se per caso deve effettuare un'operazione di input o output di dati. Questo criterio è semplice da realizzare ma non è conveniente, perché c'è una "perdita di tempo" sostanziale nell'interrogazione di ogni dispositivo, soprattutto se uno di questi non ha neanche nulla da dire. Possiamo dunque dire che ha un grande dispendio di energia. Ci sono, però, delle situazioni in cui il polling è estremamente conveniente quando abbiamo alte situazioni di traffico.
- Vectored Interrupt System: è, come il polling, un metodo per rilevare se è occorso un cambiamento all'interno del sistema, ma non ha bisogno di continui controlli per saperlo.

Ovviamente devono essere disposti dei segmenti di codice che trattano le singole interruzioni che possono essere diversificate.

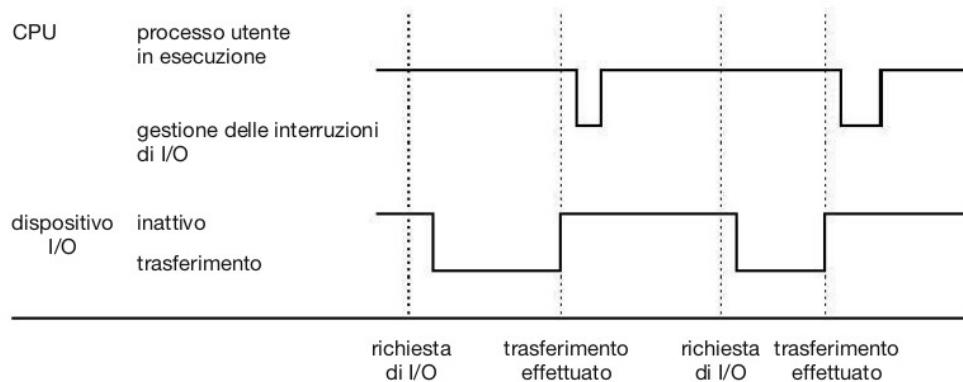


Figura 1.3 Diagramma temporale delle interruzioni per un singolo processo che emette dati.

Una volta iniziata l'operazione di I/O, si restituisce il controllo al processo utente solo dopo il completamento dell'operazione di I/O.

- L'istruzione **wait** sospende la CPU fino al successivo segnale di interruzione;
- Nei calcolatori che non prevedono un'istruzione del genere, si può generare un ciclo di attesa;
- Si ha al più una richiesta pendente alla volta.

Una volta iniziata l'operazione di I/O, si restituisce immediatamente il controllo al processore utente, senza attendere il completamento dell'operazione di I/O.

L'operazione **wait** è una chiamata di sistema (*System Call*). Una System Call è una richiesta al sistema operativo di consentire al programma utente di attendere il completamento dell'operazione.

Ovviamente il sistema operativo deve avere a disposizione una tabella di stato dei dispositivi, che contiene elementi per ciascun dispositivo di I/O che ne specificano il tipo, l'indirizzo e lo stato.

Il sistema operativo individua il controllore del dispositivo che ha emesso il segnale di interruzione, quindi accede alla tabella dei dispositivi, risale allo stato in cui il dispositivo si trova e modifica l'elemento nella tabella indicando l'occorrenza dell'interruzione.

Vediamo, graficamente, le modalità con cui questo scambio di informazioni sono effettuate:

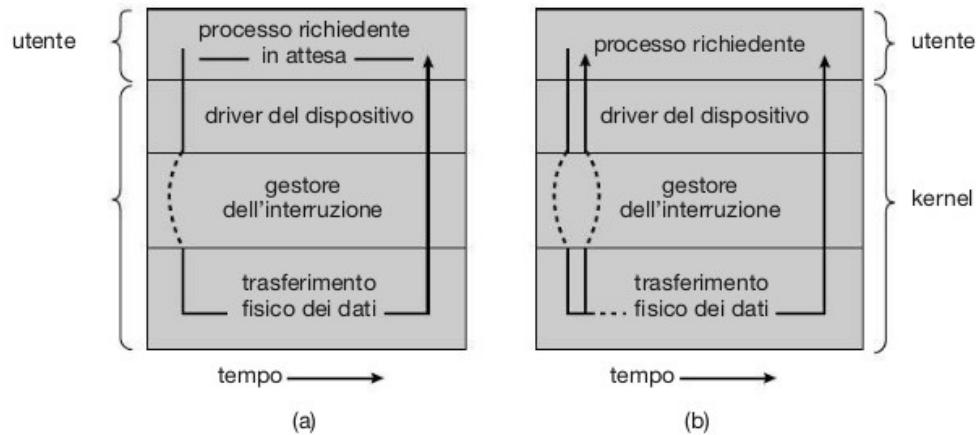


Figura 13.8 Due metodi per l'I/O; (a) sincrono e (b) asincrono.

- **Sincrono:** è un criterio che richiede che i dispositivi che intervengono nella comunicazione devono essere tutti disponibili perché devono gestire questo trasferimento di dati.
- **Asincrono:** lo scambio dei dati non necessariamente richiede la co-partecipazione dei dispositivi.

Vediamo ora, graficamente, la tabella di stato dei dispositivi:

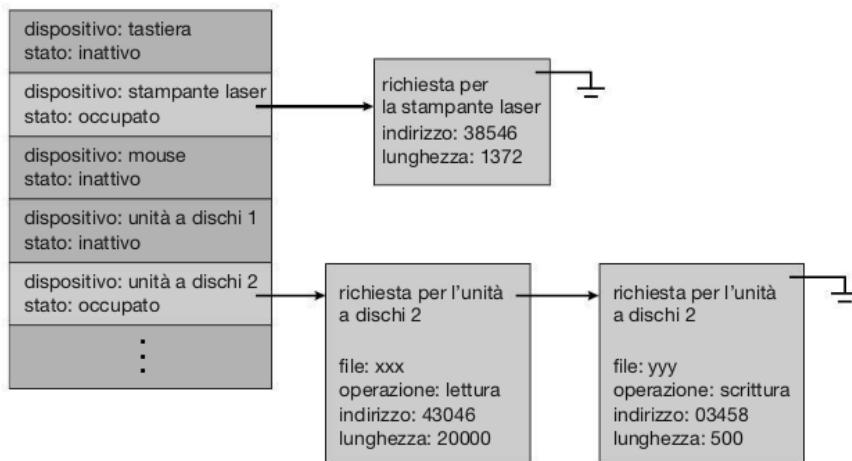


Figura 13.9 Tabella dello stato dei dispositivi.

2.2.3 ACCESSO DIRETTO ALLA MEMORIA

Con la sperimentazione si è visto che spesso e volentieri non è realmente importante coinvolgere anche la CPU per lo scambio dei dati. Dal punto di vista del modello teorico, la CPU dovrebbe governare tutti i processi che stanno nel sistema ma, se così fosse, la CPU sprecherebbe tantissimo tempo laddove potrebbe fare altre cose. Questo perché ci sono dei dati che per esempio devono essere semplicemente copiati nella memoria e quindi non c'è bisogno che la CPU si occupi di questo processo, che potrebbe avvenire direttamente. Questo avviene soprattutto per dispositivi particolarmente veloci.

Questo processo prende il nome di **DMA** (*Direct Access Memory*). Il trasferimento richiede una sola interruzione per ogni blocco di dati trasferito, piuttosto che per ogni byte.

2.2.4 STRUTTURA DELLA MEMORIA

Abbiamo di base la memoria centrale che contiene i dati ed è direttamente accessibile alla CPU. Idealmente, si vorrebbe che sia i programmi sia i dati da essi trattati risiedessero in modo permanente nella memoria centrale. Questo non è possibile per i seguenti due motivi:

- la capacità della memoria centrale non è di solito sufficiente a contenere in modo permanente tutti i programmi e i dati richiesti.
- la memoria centrale è un dispositivo di memorizzazione *volatile*, che perde il proprio contenuto quando l'alimentazione elettrica viene spenta o si interrompe.

Per queste ragioni la maggior parte dei sistemi elaborativi comprende una memoria secondaria come estensione della memoria centrale. La caratteristica fondamentale di questi dispositivi è la capacità di conservare in modo permanente grandi quantità di informazioni.

Il dispositivo più comunemente impiegato a questo scopo è l'unità a disco magnetico, adoperato per la memorizzazione sia di programmi sia di dati. La maggior parte dei programmi (applicativi e di sistema) è mantenuta in un disco sino al momento del caricamento nella memoria. Molti programmi fanno uso del disco come sorgente e destinazione delle informazioni elaborate.

Occorre tuttavia specificare che la struttura descritta (composta da registri, memoria centrale e unità a disco) rappresenta semplicemente una delle possibili configurazioni del sistema di memorizzazione di un calcolatore. Esistono altri tipi di memorie, per esempio le memorie cache, i CD-ROM e i nastri magnetici.

Vediamo lo schema funzionale di un disco:

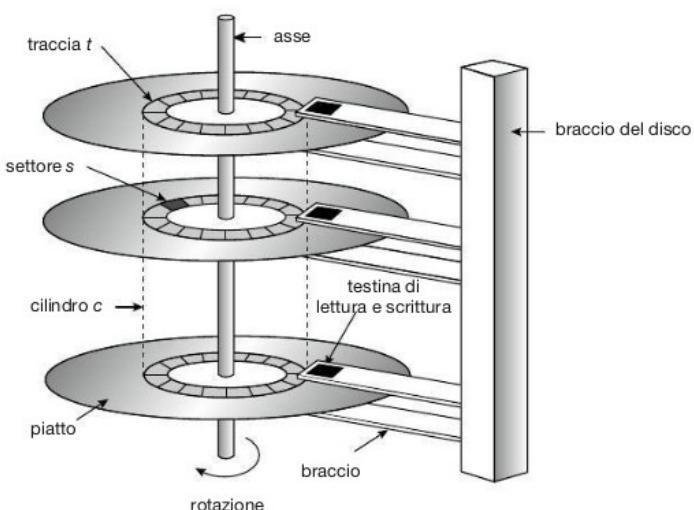


Figura 10.1 Schema di un disco a testine mobili.

2.2.5 GERARCHIE DELLE MEMORIE

I componenti di memoria di un sistema di calcolo possono essere organizzati in una struttura gerarchica in base a:

- Velocità;
- Costo;
- Volatilità.

Nella gerarchia delle memorie, appartengono anche le **cache**: copia temporanea di informazioni un'unità più veloce; la memoria centrale si può considerare una cache per la memoria secondaria.

La gerarchia basata sulla velocità è questa:

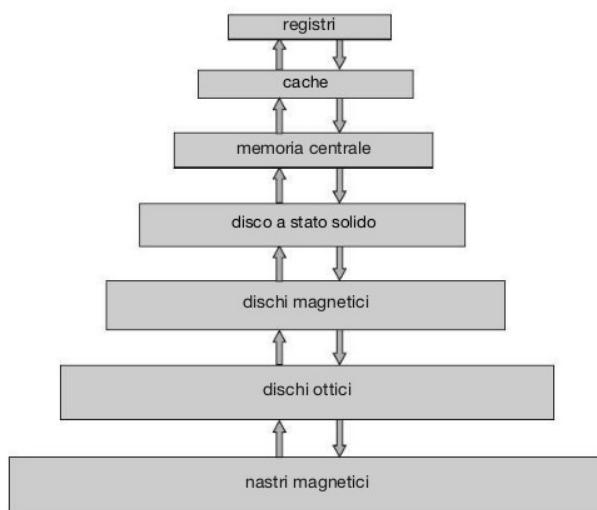


Figura 1.4 Scala gerarchica dei sistemi di memorizzazione.

- Registri: celle di memoria che si trovano all'interno della CPU. Anche se è una piccolissima zona, comunque transitano i dati e quindi possiamo considerarla come memoria. È l'oggetto capace di trattare i dati alla più alta velocità possibile, proprio perché si trova all'interno della CPU.
- Cache: in questo caso è la cache del processore. È una memoria molto piccola che permette di trattare i dati ad altissima velocità.
- Memoria centrale: è la vera e propria RAM.
- Disco di stato solido (disco RAM): un disco che possiamo configurare nella nostra RAM. Nel nostro PC abbiamo tanta RAM e molto spesso tanta RAM non ci serve. Possiamo quindi approfittarne per configurare parte di questa RAM come disco. Infatti il sistema operativo assegna a un pezzo della memoria centrale una lettera e la assegna a una logica di gestione simile, dal punto di vista dell'utente, a quella di un Hard-Disk. Quindi i dati possono essere scritti su un Hard-Disk virtuale (in questo caso il disco RAM) e possono essere elaborati molto più velocemente.
- Dischi ottici: sono ad esempio i disci ROM. Più lenti dei dischi magnetici.

Questa gerarchia non è completa, perché ci sono tantissimi altri dispositivi che andrebbero presi in considerazione.

Ogni dispositivo ha un buffer locale che permette di far transitare i dati. Questo comporta che, in realtà, uno stesso dato può esistere in vari punti. Vediamo questo schema:

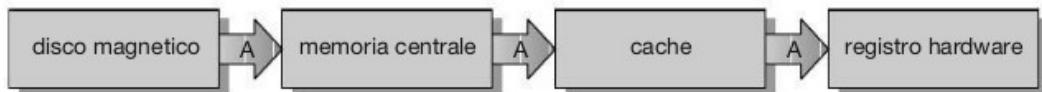


Figura 1.12 Migrazione di un intero A da un disco a un registro.

Cosa succede: il programma presente nel disco magnetico, viene copiato nella memoria centrale; per essere eseguito deve essere copiato nella cache del processore; i dati poi vengono riscritti volta per volta nel registro hardware. Tutto questo (semplificato in 4 stati, ma possono essere anche di più), come abbiamo detto prima, comporta il fatto che uno stesso dato è presente in più punti e potrebbe darci grosse difficoltà. Questo viene chiamato problema di coerenza della cache.

2.7 ARCHITETTURE DI PROTEZIONE

Quello che vogliamo fare in particolare è proteggere la CPU. Proteggerla, ad esempio, da un uso sconsiderato di un processo. Per questo la CPU può lavorare in duplice modo (*dual mode*). Questo non solo protegge la CPU, ma anche l'I/O e la memoria.

Questo duplice modo di funzionamento sta a denotare un segnale che sta a siboleggiare se i dati, che in quel momento sono trattati dalla CPU, sono del sistema operativo o se sono delle istruzioni dell'utente. I due modi di funzionamento sono dunque:

- Modo d'utente (user mode): la funzione corrente si esegue per conto di un utente;
- Modo di sistema (monitor mode, detto anche modo del supervisore, modo monitor o modo privilegiato): l'istruzione corrente si esegue per conto del sistema operativo.

Vediamo questo schema per capire il funzionamento:

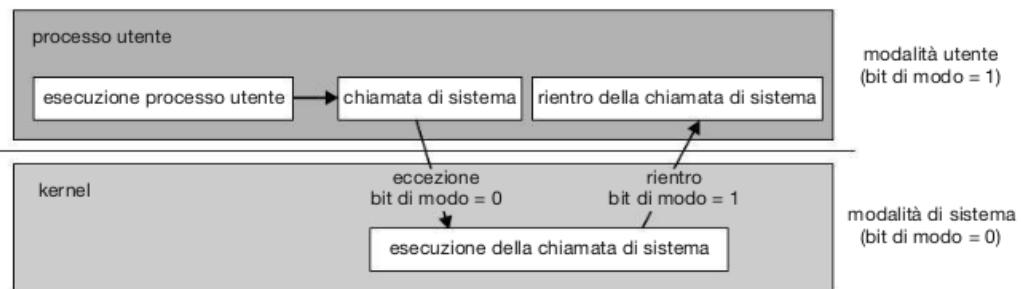


Figura 1.10 Transizione da modalità utente a modalità di sistema.

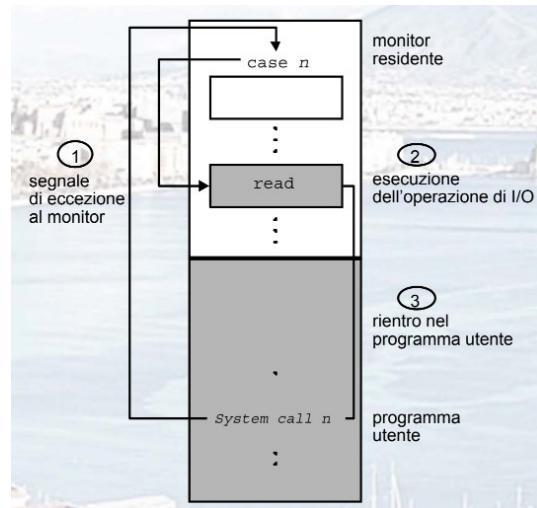
Il bit di modo (mode bit) indica quale modo è attivo: di sistema (0) o d'utente (1).

Ogni volta che si verifica un'interruzione o un'eccezione di passa dal modo d'utente al modo di sistema.

La CPU consente l'esecuzione di istruzioni privilegiate (privileged instruction) soltanto nel modo di sistema. Con istruzioni privilegiate si intendono istruzioni che non possono essere interrotte dall'utente.

Tutte le istruzioni di I/O sono istruzioni privilegiate. Questo porta a una protezione totale dell'I/O, in quanto è necessario evitare che l'utente possa in qualsiasi modo ottenere il controllo del calcolatore quando questo si trova in stato di sistema.

Vediamo con questo schema in che modo avviene l'esecuzione di una chiamata di I/O:



È importante, come abbiamo già detto, proteggere anche la memoria. Lo si fa mettendo degli indici che costituiscono l'inizio di una procedura e denotando la sua lunghezza. Così facendo sappiamo esattamente dove si trova (per conosciamo la base e quando è lunga).

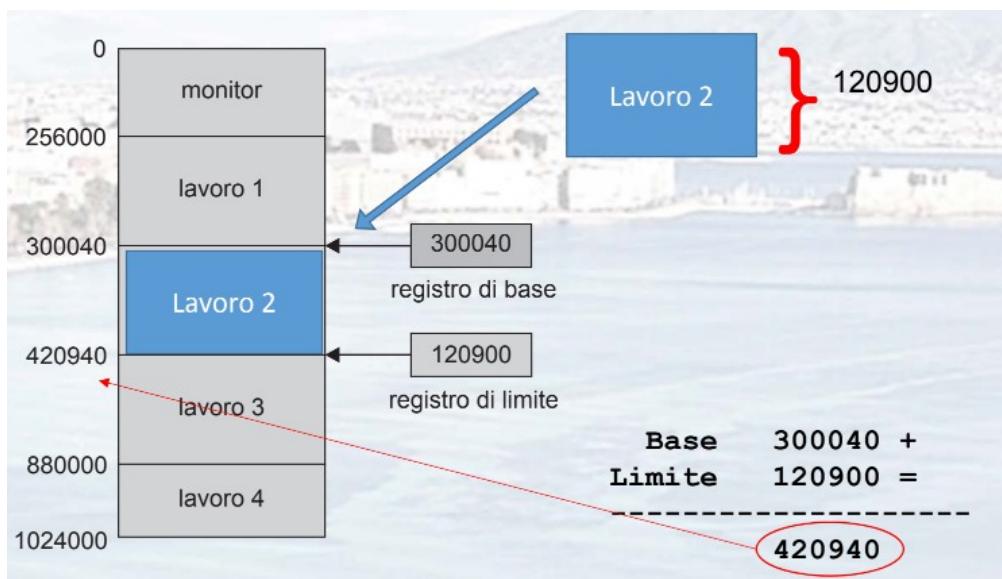
Formalizziamo il tutto: è necessario fornire protezione della memoria almeno per il vettore delle interruzioni e le relative procedure di servizio contenute nel codice del sistema operativo.

Questo tipo di protezione si realizza impiegando due registri che contengono l'intervallo degli indirizzi validi cui un programma può accedere:

- Registro di base: contiene il più basso indirizzo della memoria fisica al quale il programma dovrebbe accedere;
- Registro di limite: contiene la dimensione dell'intervallo.

Le aree di memoria al di fuori dell'intervallo stabilito sono protette.

Vediamo l'uso di un registro di base e un registro di limite tramite questo grafico:



Vediamo l'architettura di protezione degli indirizzi:

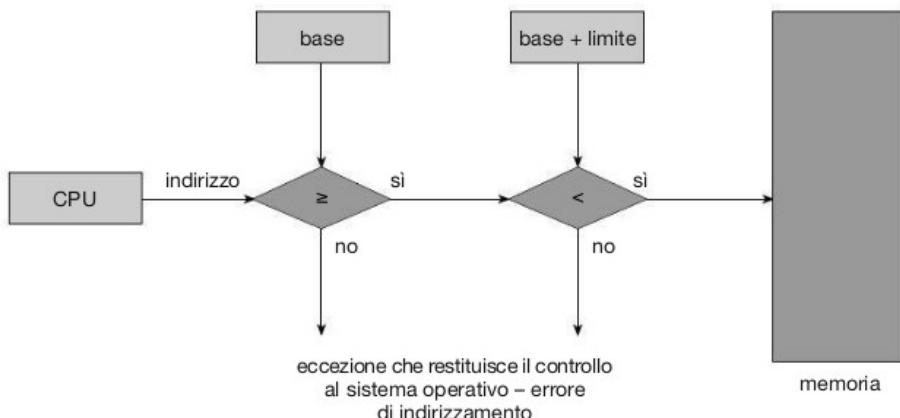


Figura 8.2 Protezione hardware degli indirizzi tramite registri base e limite.

Protezione hardware: solo il sistema operativo può caricare i registri base e limite, grazie a una speciale istruzione privilegiata. Dal momento che le istruzioni privilegiate possono essere eseguite unicamente in modalità kernel, e poiché solo il sistema operativo può essere eseguito in tale modalità, tale schema gli consente di modificare il valore di questi registri, ma impedisce tale operazione ai programmi utenti.

Per la protezione della CPU, occorre assicurare che il sistema operativo mantenga il controllo della CPU, che consiste nell'impedire che un programma utente entri in un ciclo infinito o non richieda servizi del sistema senza più restituire il controllo al sistema operativo. A tale scopo si può usare un **timer** (temporizzatore), programmabile affinché invii un segnale d'interruzione alla CPU a intervalli di tempo specificati, che possono essere fissi. Un **timer variabile** variabile di solito si realizza mediante un clock a frequenza fissa e un contatore. Il sistema operativo assegna un valore al contatore, che si decrementa a ogni impulso e quando raggiunge il valore 0 si genera un segnale d'interruzione.

I temporizzatori si usano comunemente per realizzare la partizione del tempo d'elaborazione (*time sharing*), ma possono essere utilizzati anche per il calcolo dell'ora corrente. Il *load-timer* è un'istruzione privilegiata.

3 – STRUTTURE DEI SISTEMI OPERATIVI

3.1 GESTIONE DEI PROCESSI

Un **processo** è un **programma in esecuzione**. Parliamo di programma quando il codice è passivo, quindi viene scritto ad esempio su un foglio di carta e quindi non è eseguibile; quando poi il programma viene caricato nella macchina ed è pronto per essere eseguito, parliamo di processo. Quindi un programma è un insieme di istruzioni la cui esecuzione genera dei processi. Importante specificare che a un programma possono corrispondere tanti processi.

Per svolgere i propri compiti, un processo necessita di alcune risorse, tra cui tempo di CPU, memoria, file e dispositivi di I/O.

Il sistema operativo è responsabile seguenti attività connesse alla gestione dei processi:

- Creazione e cancellazione dei processi utenti e di sistema;
- Sospensione e ripristino dei processi: un processo, quando va in esecuzione, non occorre necessariamente aspettare la sua terminazione. Quando va in esecuzione, il processo viene più volte fermato perché c'è un altro processo che ha maggiore priorità.
- Fornitura di meccanismi per:
 - Sincronizzazione dei processi;
 - Comunicazione tra processi;

3.2 GESTIONE DELLA MEMORIA CENTRALE

La memoria centrale è un vasto vettore di dimensioni che variano tra le centinaia di migliaia e i miliardi di parole, ciascuna delle quali è dotata del proprio indirizzo. È un magazzino di dati velocemente accessibile ed è condivisa dalla CPU e da alcuni dispositivi di I/O.

La memoria centrale è volatile, e perde le informazioni in caso di guasto del sistema.

Il sistema operativo delle seguenti attività connesse alla gestione della memoria centrale:

- Tenere traccia di quali parti della memoria sono attualmente usate e da che cosa;
- Decidere quali processi si debbano caricare nella memoria quando vi sia spazio disponibile;
- Assegnare e revocare lo spazio di memoria secondo le necessità.

3.3 GESTIONE DELLA MEMORIA DI MASSA

3.3.1 GESTIONE DEI FILE

Un file è una raccolta di informazioni correlate definite dal loro creatore. Comunemente i file rappresentano programmi (sia sorgente sia oggetto) e dati. Nota bene: il nome e l'estensione di un file NON ne garantiscono la tipologia con certezza assoluta. Ad esempio: `prova.exe` potrebbe contenere un testo, mentre `prova.txt` potrebbe essere un eseguibile.

I file sono generalmente organizzati in directory, che ne facilitano l'uso. Infine, se più utenti hanno accesso ai file, si potrebbe voler controllare chi ha la possibilità di accedervi e in che modo (per esempio, lettura, scrittura, aggiunta).

Il sistema operativo è responsabile delle seguenti attività connesse alla gestione dei file:

- Creazione e cancellazione dei file;
- Creazione e cancellazione di directory;
- Fornitura delle funzioni fondamentali per la gestione di file e directory;
- Associazione dei file ai dispositivi di memoria secondaria;
- Creazione di copie di riserva (*backup*) dei file su dispositivi di memorizzazione non volatili.

3.3.2 GESTIONE DEL SISTEMA DI I/O

Il sistema di I/O è composto dalle seguenti parti:

- Un sistema buffer-caching: il sistema operativo deve essere in grado di gestire i sistemi di buffer-caching, cioè di aree tampone che contengono i dati in transito per poter essere elaborati.
- Un'interfaccia generale per i driver dei dispositivi: i driver sono "pezzetti" di

- software che costituiscono le regole di comunicazione con un certo dispositivo.
- I driver per specifici dispositivi.

3.3.3 GESTIONE DELLA MEMORIA SECONDARIA

Poiché la memoria centrale è troppo piccola per contenere tutti i dati e tutti i programmi, e il suo contenuto va perduto se il sistema si spegne, il calcolatore deve disporre di una memoria secondaria a sostegno della memoria centrale.

La maggior parte dei moderni sistemi elaborativi impiega i dischi come principale mezzo di memorizzazione secondaria, sia per i programmi sia per i dati. I dischi contengono la maggior parte dei programmi, compresi i compilatori, gli assemblatori, i word processor e gli editor.

Il sistema operativo deve essere in grado di trasferire i dati dalla memoria centrale al disco e viceversa. Quando parliamo di dischi, parliamo indistintamente di qualunque tipo di disco (SSD o anche di un hardisk). Il sistema operativo è anche responsabile delle seguenti attività connesse alla gestione dei dischi:

- Gestione dello spazio libero;
- Assegnazione dello spazio;
- Scheduling del disco: pianificazione dell'ingresso e dell'uscita dei dati dal disco.

3.3.4 CACHE

Il concetto di cache è un principio importante di un sistema elaborativo. Di norma le informazioni sono mantenute in un sistema di memoria come la memoria centrale; al momento del loro uso si copiano temporaneamente in un'unità più veloce: la cache.

Esistono anche cache che sono interamente gestite dall'hardware del sistema: per esempio la maggior parte dei sistemi è dotata di una cache per la memorizzazione delle istruzioni che presumibilmente saranno eseguite dopo l'istruzione corrente. Senza di essa, la CPU dovrebbe attendere parecchi cicli prima che un'istruzione sia prelevata dalla memoria. Per motivi analoghi, la maggior parte dei sistemi è dotata di una o più cache di dati nella gerarchia delle memorie.

La memoria centrale si può considerare una cache per la memoria secondaria, giacché i dati in essa contenuti devono essere riportati nella memoria centrale per poter essere usati e qui devono risiedere prima di essere trasferiti nella memoria secondaria.

3.4 RETI (SISTEMI DISTRIBUITI)

Per sistema distribuito si intende un insieme di elaboratori fisicamente separati, e con caratteristiche spesso eterogenee, interconnessi da una rete per consentire agli utenti l'accesso alle varie risorse dei singoli sistemi.

Una rete si può considerare, in parole semplici, come un canale di comunicazione tra due o più sistemi. I sistemi distribuiti si basano sulle reti per realizzare le proprie funzioni. Le reti differiscono per i protocolli usati, per le distanze tra i nodi e per il mezzo attraverso il quale avviene la comunicazione.

Un sistema distribuito offre all'utente l'accesso alle varie risorse del sistema. L'accesso a una risorse condivisa permette di:

- Accelerare il calcolo;
- Aumentare la disponibilità dei dati;

- Incrementare l'affidabilità.

3.5 SISTEMA DI PROTEZIONE

Un sistema di protezione è un sistema che deve definire dei meccanismi per controllare l'accesso da parte delle risorse e per garantire che i programmi degli utenti non tendano ad accaparrarsi senza lasciare più risorse.

Il meccanismo di protezione deve:

- Distinguere tra uso autorizzato e non autorizzato;
- Specificare i controllo che devono essere attivati;
- Fornire strumenti di miglioramento dell'affidabilità.

3.6 INTERPRETE DEI COMANDI

È un programma che, una volta eseguito, serve per avere un colloquio diretto con la macchina mediate un'interfaccia testuale.

Molti comandi si impartiscono al sistema operativo attraverso istruzioni di controllo che riguardano:

- Creazione e gestione dei processi;
- I/O;
- Gestione della memoria secondaria;
- Gestione della memoria centrale;
- Accesso al file-system;
- Protezione;
- Reti;

Il programma che legge e interpreta le istruzioni di controllo ha diversi nomi:

- Interprete di schede di controllo (*control-card interpreter*);
- Interprete di riga di comando (*command-line interpreter*);
- *Shell*: si parla di shell quando i sistemi consentono la scelta tra molteplici interpreti dei comandi. In UNIx e Linux, per esempio, l'utente può scegliere tra svariate shell differenti, come la Bourne, la C, la Bourne-again, la Korn, e così via. Sono anche disponibili shell di terze parti e shell gratuite scritte dagli utenti.

La funzione principale dell'interprete dei comandi consiste nel raccogliere ed eseguire il successivo comando impartito dall'utente.

3.7 SERVIZI DI UN SISTEMA OPERATIVO

Un sistema operativo offre un ambiente in cui eseguire i programmi e fornire servizi. Naturalmente, i servizi specifici variano secondo il sistema operativo, ma si possono identificare alcune classi di servizi comuni. Il loro scopo è facilitare il compito dei programmatori di applicazioni.

- Interfaccia con l'utente: quasi tutti i sistemi operativi hanno un'interfaccia con l'utente (UI). Essa può assumere diverse forme. Un'interfaccia a riga di comando (CLI) è basata su stringhe che codificano i comandi, insieme a un metodo per inserirli (per esempio, una tastiera per digitare i comandi in uno specifico formato con determinate opzioni). Un'interfaccia batch (a lotti), invece, prevede che comandi e relative direttive siano codificati in file, che vengono poi eseguiti. La forma senz'altro più diffusa è un'interfaccia utente grafica (GUI), ossia un sistema a finestre dotato di un dispositivo puntatore (per esempio, il mouse) per comandare operazioni di I/o e selezionare opzioni dai menu, insieme a una tastiera per inserire

del testo. certi sistemi offrono alcune o anche tutte queste soluzioni.

- **Esecuzione di un programma:** il sistema deve poter caricare un programma in memoria ed eseguirlo. Il programma deve poter terminare la propria esecuzione in modo normale o anomalo (indicando l'errore).
- **Operazioni di I/O:** un programma in esecuzione può richiedere un'operazione di I/O che implica l'uso di un file o di un dispositivo di I/O. Per motivi di efficienza e protezione, di solito un utente non può controllare direttamente i dispositivi di I/o, quindi il sistema operativo deve offrire mezzi adeguati.
- **Gestione del file system:** il file system riveste un interesse particolare. I programmi richiedono l'esecuzione di operazioni di lettura e scrittura su file, oltre a creare e cancellare file e directory. Essi hanno anche bisogno di creare e cancellare i file, di eseguire la ricerca di un file con un certo nome, e disporre di informazioni relative al file stesso. Alcuni sistemi operativi, infine, gestiscono i permessi di accesso ai file sulla base della proprietà del file interessato. Molti sistemi operativi offrono all'utente la scelta di file system diversi con funzionalità e prestazioni specifiche.
- **Comunicazioni:** in molti casi un processo ha bisogno di scambiare informazioni con un altro processo. ciò avviene principalmente in due modi: tra processi in esecuzione nello stesso calcolatore e tra processi in esecuzione in calcolatori diversi collegati per mezzo di una rete. La comunicazione si può realizzare tramite una memoria condivisa, che permette a due o più processi di leggere e scrivere in una porzione di memoria che condividono, o attraverso lo scambio di messaggi, in questo caso il sistema operativo trasferisce pacchetti d'informazioni in un formato predefinito tra i vari processi (comunemente usato per piccoli dati da scambiare). Questo è sicuramente più semplice perché lo scambio avviene in una "casella" che si trova in una posizione ben precisa; cosa che invece non capita con l'utilizzo di una memoria condivisa perché può trovarsi in una posizione diversa ogni volta.
- **Rilevamento di errori:** il sistema operativo deve essere sempre capace di rilevare e correggere eventuali errori che possono verificarsi nella cPU e nei dispositivi di memoria (come un errore di memoria o un guasto all'alimentazione elettrica), nei dispositivi di I/o (come un errore di parità in un nastro, il guasto di una connessione di rete, la mancanza di carta nella stampante) e in un programma utente (come una divisione per zero, un tentativo d'accesso a una locazione di memoria illegale, un uso eccessivo del tempo di CPU). Per assicurare un'elaborazione corretta e coerente il sistema operativo deve saper intraprendere l'azione giusta per ciascun tipo d'errore. Talvolta l'unica scelta possibile è l'arresto del sistema, altre volte è possibile terminare il processo che è causa d'errore o restituire un codice d'errore a un processo in modo che da solo cerchi di rilevare e correggere l'errore.

Esiste anche un'altra serie di funzioni del sistema operativo che non riguarda direttamente gli utenti, ma assicura il funzionamento efficiente del sistema stesso:

- **Assegnazione delle risorse:** se sono attivi più utenti o sono contemporaneamente in esecuzione più processi, il sistema operativo provvede all'assegnazione delle risorse necessarie a ciascuno di essi. Per esempio, per determinare come utilizzare al meglio la cPU, i sistemi operativi impiegano le procedure di scheduling della CPU, che tengono conto della velocità, dei processi da eseguire, del numero di registri disponibili e di altri fattori.
- **Contabilizzazione dell'uso delle risorse:** vogliamo mantenere traccia di quali utenti usano il calcolatore, segnalando quali e quante risorse impiegano. Questo tipo di registrazione si può usare per contabilizzare l'uso delle risorse, in modo da addebitare il costo agli utenti, oppure per redigere statistiche.
- **Protezione e sicurezza:** i proprietari di informazioni memorizzate in un sistem

elaborativo multiutente o in rete possono voler controllare l'uso di tali informazioni. Quando più processi separati sono in esecuzione concorrente essi non devono influenzarsi o interferire con il sistema operativo. La protezione assicura che l'accesso alle risorse del sistema sia controllato. È importante anche proteggere il sistema dagli estranei. La sicurezza di un sistema comincia con la richiesta d'identificazione da parte di ciascun utente, di solito attraverso password, per permettere l'accesso alle risorse; si estende fino a difendere i dispositivi di I/o (compresi gli adattatori di rete) dai tentativi d'accesso illegali e provvede al loro rilevamento.

3.8 CHIAMATE DEL SISTEMA

Le chiamate del sistema (*System Call*) costituiscono l'interfaccia tra un processo e il sistema operativo. Tali chiamate sono generalmente disponibili sotto forma di routine scritte in C o C++, sebbene per alcuni compiti di basso livello, come quelli che comportano un accesso diretto all'hardware, può essere necessario il ricorso al linguaggio assembly.

Per passare i parametri al sistema operativo si usano tre metodi generali, a seconda del tipo di chiamare:

- Passare i parametri in registri (nel caso in cui si vogliano scambiare pochi dati).
- Memorizzare i parametri in un blocco o tabella di memoria e passare l'indirizzo del blocco, in forma di parametro, in un registro (quando si presentano casi incui vi sono più parametri che registri).
- Collocare (push) i parametri in una pila da cui sono prelevati (pop) dal sistema operativo.

Alcuni sistemi operativi preferiscono i metodi del blocco o dello stack, poiché non limitano il numero o la lunghezza dei parametri da passare. Vediamo questo concetto attraverso questo grafico:

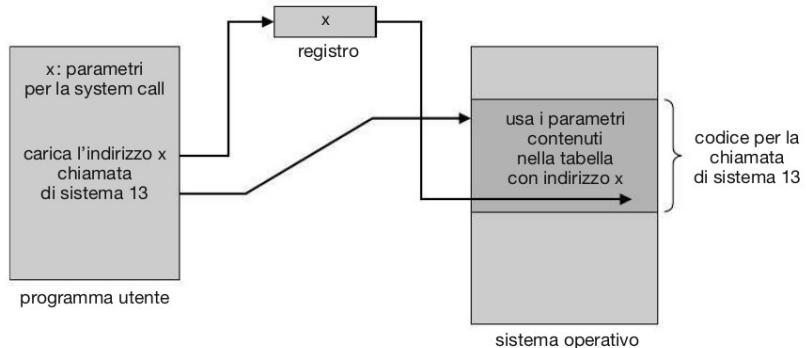


Figura 2.7 Passaggio di parametri in forma di tabella.

Le chiamate di sistema sono classificabili approssimativamente in cinque categorie principali:

- Controllo dei processi.
- Gestione dei file.
- Gestione dei dispositivi.
- Gestione delle informazioni
- Comunicazione

3.9 STRUTTURA DEL SISTEMA OPERATIVO

3.9.1 ESECUZIONE NELL'MS-DOS

MS-DOS sta per Microsoft disk operative system. Era il primo sistema operativo ad alta diffusione.

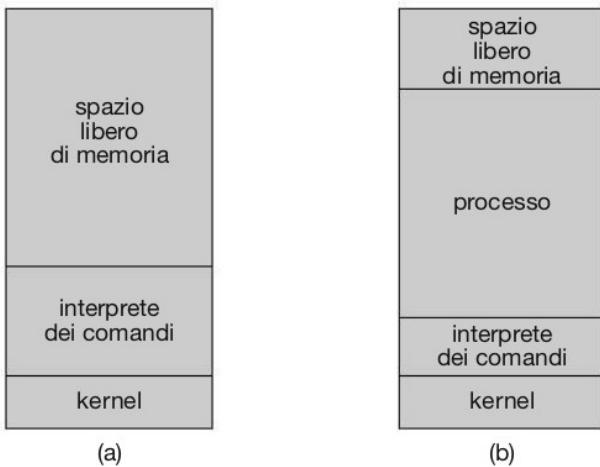


Figura 2.9 Esecuzione nell'MS-DOS. (a) All'avviamento del sistema. (b) Durante l'esecuzione di un programma.

Il sistema operativo MS-DOS era progettato col fine di fornire la massima funzionalità nel minimo spazio. Non era suddiviso in moduli e, anche se dotato di una semplice struttura, le sue interfacce e i livelli di funzionalità non sono ben separati.

Dispone di un interprete di comandi, attivato all'avviamento del calcolatore, e che esegue un solo programma alla volta (Figura (a)). Opera in maniera semplice senza creare alcun nuovo processo; carica il programma in memoria, riscrivendo anche la maggior parte della memoria che esso stesso occupa, in modo da lasciare al programma quanta più memoria è possibile (Figura (b)); quindi imposta il contatore di programma alla prima istruzione del programma da eseguire.

A questo punto si esegue il programma e si possono verificare due situazioni: un errore causa un segnale di eccezione, oppure il programma esegue una chiamata di sistema per terminare la propria esecuzione. In entrambi i casi si registra il codice d'errore in memoria di sistema per un eventuale uso successivo, quindi quella piccola parte dell'interprete che non era stata sovrascritta riprende l'esecuzione e il suo primo compito consiste nel ricaricare dal disco la parte rimanente dell'interprete stesso. Eseguito questo compito, quest'ultimo mette a disposizione dell'utente, o del programma successivo, il codice d'errore registrato.

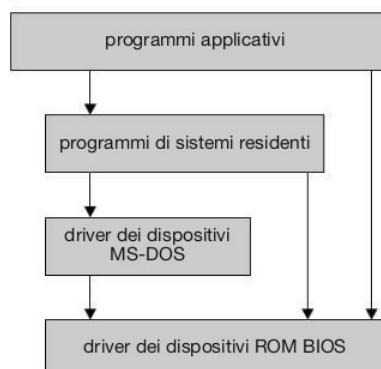


Figura 2.11 Struttura degli strati dell'MS-DOS.

Come possiamo vedere, i programmi applicativi e i programmi di sistemi residenti, riuscivano a colloquiare direttamente con l'hardware della macchina, bypassando completamente il controllo del sistema operativo. Da un lato era un bene perché il sistema era molto veloce; dall'altro era anche un male perché non c'era un controllo del sistema operativo.

3.9.2 SISTEMA OPERATIVO UNIX

Successivo all'MS-DOS è il sistema operativo UNIX, sicuramente più evoluto rispetto al precedente. Inizialmente era limitato dalle funzioni dall'architettura sottostante. È formato da due parti:

- Programmi di sistema;
- Nucleo: tutto ciò che si trova sotto l'interfaccia delle chiamate di sistema e sopra i dispositivi fisici, è il nucleo. Fornisce il file system, lo scheduling della CPU, la gestione della memoria e altre funzioni riguardante il sistema operativo: in un solo livello sono combinate un'enorme quantità di funzioni.

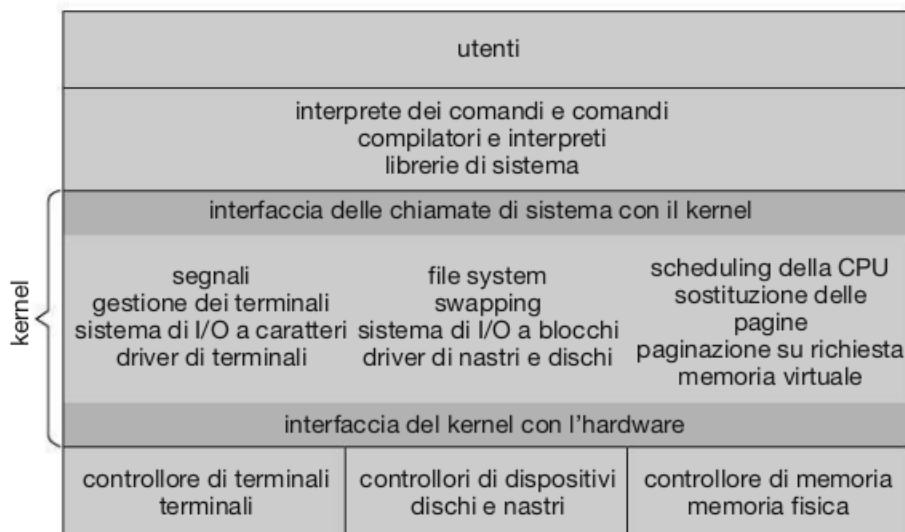


Figura 2.12 Struttura del sistema UNIX.

In questo sistema operativo, quando un utente inizia una sessione di lavoro, il sistema esegue un interprete dei comandi (shell) scelto dall'utente. Quest'interprete è simile a quello dell'MS-DOS nell'accettare i comandi e nell'eseguire programmi richiesti dall'utente. Tuttavia, poiché il UNIX è un sistema multitasking, l'interprete dei comandi può continuare l'esecuzione mentre si esegue l'altro programma, come possiamo vedere da questa figura:

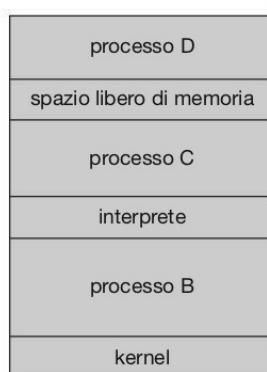


Figura 2.10 Esecuzione di più programmi nel sistema operativo FreeBSD.

3.9.3 PROGRAMMI DI SISTEMA

Un'altra caratteristica importante di un sistema moderno è quella che riguarda l'insieme di programmi di sistema. Detti anche utilità di sistema, offrono un ambiente più conveniente per lo sviluppo e l'esecuzione dei programmi; alcuni sono semplici interfacce per le chiamate di sistema, altri sono considerevolmente più complessi; in generale si possono classificare nelle seguenti categorie:

- **Gestione dei file:** questi programmi creano, cancellano, copiano, rinominano, stampano, elencano e in genere compiono operazioni sui file e le directory.
- **Informazioni di stato:** Alcuni programmi richiedono semplicemente al sistema di indicare data, ora, quantità di memoria disponibile o spazio nei dischi, numero degli utenti e simili informazioni di stato. Altri, più complessi, forniscono informazioni dettagliate su prestazioni, accessi al sistema e debug. In genere mostrano le informazioni su terminale, o tramite altri dispositivi per l'uscita dei dati, o, ancora, all'interno di una finestra della GUI.
- **Modifica dei file:** diversi editor sono disponibili per creare e modificare il contenuto di file memorizzati su dischi o altri dispositivi, oltre a comandi speciali per l'individuazione di contenuti di file o per particolari trasformazioni del testo.
- **Ambienti d'ausilio alla programmazione:** compilatori, assemblatori, debugger e interpreti dei comuni linguaggi di programmazione sono spesso forniti insieme con il sistema operativo oppure disponibili per il download.
- **Caricamento ed esecuzione dei programmi:** una volta assemblato o compilato, per essere eseguito, un programma deve essere caricato in memoria.
- **Comunicazioni:** questi programmi offrono i meccanismi con cui si possono creare collegamenti virtuali tra processi, utenti e calcolatori diversi. Permettono agli utenti d'inviare messaggi agli schermi d'altri utenti, di consultare il Web, d'inviare messaggi di posta elettronica, di effettuare il login su calcolatori remoti, di trasferire file da un calcolatore a un altro.
- **Programmi d'applicazione:** programmi che risolvono problemi comuni o che eseguono operazioni comuni. Comprendono browser web, word processor, fogli di calcolo, sistemi di basi di dati, compilatori, programmi per analisi statistiche e visualizzazioni, oltre che videogiochi.

Per la maggior parte degli utenti, l'interfaccia col sistema operativo è definita dai programmi di sistema piuttosto che dalle effettive chiamate di sistema.

3.9.4 IL METODO STRATIFICATO

In presenza di hardware appropriato, i sistemi operativi possono essere suddivisi in moduli più piccoli e gestibili di quanto non fossero quelli delle prime versioni di MS-DOS e UNIX. Ciò permette al sistema operativo di mantenere un controllo molto più stretto sul calcolatore e sulle applicazioni che lo utilizzano.

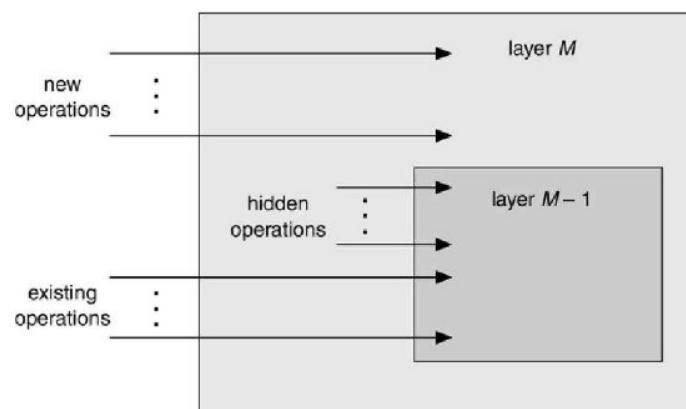
Vi sono molti modi per rendere modulare un sistema operativo. Uno di essi è il metodo stratificato, secondo il quale il sistema è suddiviso in un certo numero di livelli o strati: il più basso corrisponde all'hardware (strato 0), il più alto all'interfaccia con l'utente (strato N).

Il vantaggio principale offerto da questo metodo è dato dalla semplicità di progettazione e di debugging. gli strati sono composti in modo che ciascuno usi solo funzioni (operazioni) e servizi appartenenti a strati di livello inferiore. Questo approccio semplifica il debugging e la verifica del sistema. Il primo strato si può mettere a punto senza intaccare il resto del sistema, poiché per realizzare le proprie funzioni usa, per definizione, solo lo strato hardware, che si presuppone sia corretto. Passando al secondo strato si presume, dopo la messa a punto, la correttezza del primo. Il procedimento si ripete per ogni strato.

Se si riscontra un errore, questo deve trovarsi in quello strato, poiché gli strati inferiori sono già stati corretti; quindi la suddivisione in strati semplifica la progettazione e la realizzazione di un sistema.

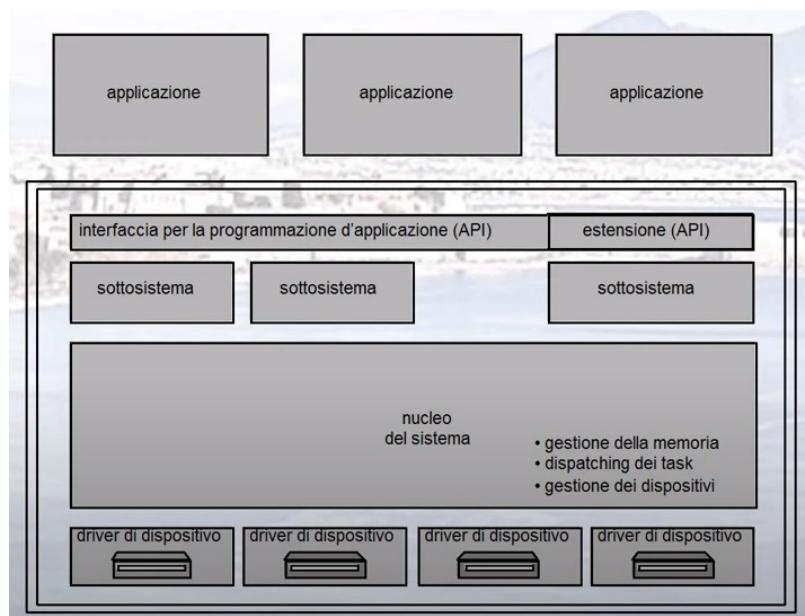
La principale difficoltà del metodo stratificato risiede nella definizione appropriata dei diversi strati. È necessaria una progettazione accurata, poiché ogni strato può sfruttare esclusivamente le funzionalità degli strati su cui poggia. Un ulteriore problema che si pone con la struttura stratificata è che essa tende a essere meno efficiente delle altre; per esempio, per eseguire un'operazione di I/O un programma utente invoca una chiamata di sistema che è intercettata dallo strato di I/O che, a sua volta, esegue una chiamata allo strato di gestione della memoria, che a sua volta richiama lo strato di scheduling della CPU e che quindi è passata all'opportuno dispositivo di I/O. In ciascuno strato i parametri sono modificabili, può rendersi necessario il passaggio di dati, e così via; ciascuno strato aggiunge un overhead alla chiamata di sistema. Ne risulta una chiamata di sistema che richiede molto più tempo di una chiamata di sistema corrispondente in un sistema non stratificato.

Vediamo uno strato di sistema operativo tramite questo grafico:



3.9.5 STRUTTURA STRATIFICATA DELL'OS/2

Anche questo aveva una struttura stratificata.



3.9.6 MICROKERNEL (ORIENTAMENTO A MICRONUCLEO)

Abbiamo visto che, a mano a mano che il sistema operativo UNIX è stato esteso, il kernel è cresciuto notevolmente, diventando sempre più difficile da gestire. Verso la metà degli anni '80 un gruppo di ricercatori della Carnegie Mellon University progettò e realizzò un sistema operativo, Mach, col kernel strutturato in moduli secondo il cosiddetto orientamento a microkernel. Seguendo questo orientamento si progetta il sistema operativo rimuovendo dal kernel tutti i componenti non essenziali, realizzandoli come programmi di livello utente e di sistema. Ne risulta un kernel di dimensioni assai inferiori.

Non c'è un'opinione comune su quali servizi debbano rimanere nel kernel e quali si debbano realizzare nello spazio utente. Tuttavia, in generale, un microkernel offre i servizi minimi di gestione dei processi, della memoria e di comunicazione.

La comunicazione viene realizzata mediante scambio di messaggi. Per accedere a un file, per esempio, un programma client deve interagire con il file server; ciò non avviene mai in modo diretto, ma tramite uno scambio di messaggi con il microkernel.

Uno dei vantaggi del microkernel è la facilità di estensione del sistema operativo: poiché è ridotto all'essenziale, se il kernel deve essere modificato, i cambiamenti da fare sono ridotti, e il sistema operativo risultante è più semplice da portare su diverse architetture. Inoltre offre maggiori garanzie di sicurezza e affidabilità, poiché i servizi si eseguono in gran parte come processi utenti, e non come processi del kernel: se un servizio è compromesso, il resto del sistema operativo rimane intatto.

3.9.7 MACCHINE VIRTUALI

Il concetto di macchina virtuale si sviluppa logicamente dal metodo stratificato. I programmi d'applicazione possono considerare quello che si trova a un livello gerarchico inferiore come se fosse parte della macchina stessa, anche se i programmi di sistema si trovano a un livello superiore.

Ogni volta che usiamo il termine *virtuale* facciamo uso di un concetto di astrazione: la virtualizzazione astrae, ha bisogno di gestire una risorsa in maniera diversa da quella che è la sua reale, fisica, configurazione. Con essa estendiamo la funzionalità strettamente reale, in una più ampia. Un altro vantaggio è che, tramite le macchine virtuali, siamo in grado di poter testare il software. Il contro sta nel fatto che una macchina è più lenta quante più macchine virtuali vengono installate. Bisogna anche però tener presente che, generalmente, le macchine virtuali quando lavorano non sono tutte quante contemporaneamente al massimo carico.

La macchina virtuale è un vero e proprio programma che va in esecuzione e che fa girare in un ambiente protetto un'altra macchina. Con questo processo possiamo far girare su un unico hardware più macchine virtuali. Il sistema operativo crea l'illusione che un processo disponga della propria CPU con la propria memoria (virtuale).

Il calcolatore condivide le risorse in modo da creare macchine virtuali:

- La partizione del tempo d'uso della CPU si può usare sia per condividere la CPU sia per dare l'illusione che gli utenti dispongano di una propria CPU.
- La gestione asincrona delle operazioni di I/O e dell'esecuzione di più processi, unita a un file system, consente di creare lettori di schede e stampanti.
- Un normale terminale di un sistema a partizione del tempo funziona da console d'operatore della macchina virtuale.

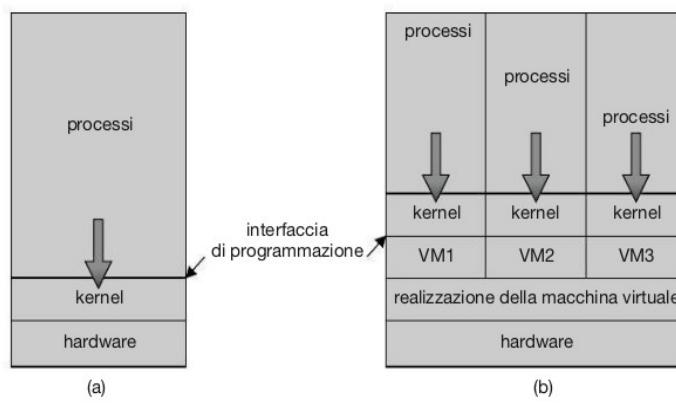


Figura 16.1 Modelli di sistema: (a) semplice; (b) macchina virtuale.

Vediamo bene i pro e i contro delle macchine virtuali:

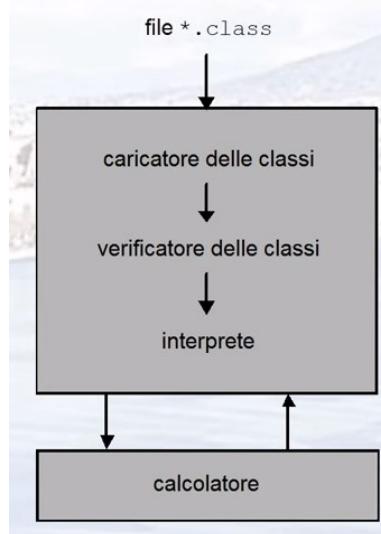
- L'uso delle macchine virtuali protegge completamente le risorse di sistema poiché ciascuna macchina virtuale è isolata dalle altre. Uno svantaggio è che non c'è una condivisione diretta delle risorse.
- Un sistema di macchine virtuali è un perfetto mezzo di ricerca e sviluppo dei sistemi operativi. Lo sviluppo avviene sulla macchina virtuale non su quella fisica, evitando così modifiche che potrebbero causare errori di programmazione in altri punti.
- A una maggiore complessità della macchina da emulare corrisponde una maggiore difficoltà di realizzazione di un'accurata macchina virtuale, e una maggiore lentezza nell'esecuzione

3.9.8 MACCHINA VIRTUALE JAVA

I programmi Java sono *bytecode* indipendente dall'architettura sottostante eseguiti dalla macchina virtuale Java (JVM, Java Virtual Machine). La JVM consiste in:

- Un caricatore di classi.
- Un verificatore di classi.
- Un interprete del linguaggio che esegue il bytecode.

Il compilatore istantaneo Just-In-Time (JIT) ne migliora le prestazioni.



3.10 PROGETTAZIONE E REALIZZAZIONE DI UN SISTEMA OPERATIVO

3.10.1 SCOPI DELLA PROGETTAZIONE

Il primo problema che s'incontra nella progettazione di un sistema riguarda la definizione degli obiettivi e delle specifiche del sistema stesso. Al più alto livello, la progettazione del sistema è influenzata in modo decisivo dalla scelta dell'architettura fisica e del tipo di sistema: a lotti (batch) o time sharing, mono o multiutente, distribuito, per elaborazioni in real-time o general-purpose.

D'altra parte, oltre questo livello di progettazione, i requisiti possono essere molto difficili da specificare, anche se, in generale, si possono distinguere in due gruppi fondamentali:

- **Obiettivi degli utenti**: gli utenti desiderano che un sistema sia utile, facile da imparare e usare, affidabile, sicuro e veloce; questa caratteristiche non sono particolarmente utili nella progettazione di un sistema, poiché non tutti concordano sui metodi da applicare per raggiungere questi scopi.
- **Obiettivi del sistema**: il sistema operativo deve essere di facile progettazione, realizzazione e manutenzione; deve essere flessibile, affidabile, senza errori ed efficiente. Anche in questo caso si tratta di requisiti vaghi, interpretabili in vari modi.

3.10.2 MECCANISMI E CRITERI

Un principio molto importante è quello che riguarda la distinzione tra meccanismi e criteri o politiche (policy). I meccanismi determinano *come* eseguire qualcosa; i criteri, invece, stabiliscono che cosa si debba fare.

La distinzione tra meccanismi e politiche è molto importante ai fini della flessibilità. Le politiche sono soggette a cambiamenti di luogo o di tempo. Nei casi peggiori il cambiamento di una politica può richiedere il cambiamento del meccanismo sottostante. Sarebbe preferibile disporre di meccanismi generali: in questo caso un cambiamento di politica implicherebbe solo la ridefinizione di alcuni parametri del sistema.

3.10.3 REALIZZAZIONE

Una volta progettato, un sistema operativo va realizzato. I primi sistemi operativi erano scritti in linguaggio assembler. oggi, anche se alcuni sistemi operativi sono ancora scritti in linguaggio assembler, la maggior parte è scritta in un linguaggio di alto livello come il c o in linguaggi di livello ancora più alto come il C++. In realtà un sistema operativo può essere scritto in più linguaggi, per esempio usando il linguaggio assembler per i livelli più bassi del kernel, il c per routine di livello superiore e il c, il C++ o linguaggi di scripting interpretati come PERL, Python o gli script di shell per i programmi di sistema. ogni distribuzione di Linux include probabilmente programmi scritti in ciascuno di questi linguaggi.

I vantaggi derivanti dall'uso di un linguaggio di alto livello, o perlomeno di un linguaggio orientato in modo specifico allo sviluppo di sistemi, per implementare un sistema operativo, sono gli stessi che si ottengono quando il linguaggio si usa per i programmi applicativi: il codice si scrive più rapidamente, è più compatto ed è più facile da capire e mettere a punto. Inoltre il perfezionamento delle tecniche di compilazione consente di migliorare il codice generato per l'intero sistema operativo con una semplice ricompilazione. Infine, un sistema operativo scritto in un linguaggio di alto livello è più facile da adattare a un'altra architettura (*porting*).

I soli eventuali svantaggi che possono presentarsi nella realizzazione di un sistema operativo in un linguaggio di alto livello sono una minore velocità d'esecuzione e una maggiore occupazione di spazio di memoria, una questione ormai superata nei sistemi moderni. D'altra parte, benché un programmatore esperto di un linguaggio assembler possa produrre piccole procedure di grande efficienza, per quel che riguarda i programmi molto estesi, un moderno compilatore può eseguire complesse analisi e applicare raffinate ottimizzazioni che producono un codice eccellente.

3.10.4 GENERAZIONE DEI SISTEMI OPERATIVI

Un sistema operativo si può progettare, codificare e realizzare specificamente per una singola macchina; tuttavia, è più diffusa la pratica di progettare sistemi operativi da impiegare in macchine di una stessa classe con configurazioni diverse in vari siti. Il sistema si deve quindi configurare o generare per ciascuna situazione specifica, un processo talvolta noto come generazione del sistema (SYSGEN).

Dopo che un sistema operativo è stato scritto, bisogna renderlo utilizzabile da parte dell'hardware. Ma come fa l'hardware dell'elaboratore a sapere dove si trova il kernel e a caricarlo? La procedura d'avviamento di un calcolatore con il caricamento del kernel è nota come avviamento (booting) del sistema: nella maggior parte dei calcolatori un piccolo segmento di codice, noto come programma d'avvio (bootstrap program) o caricatore d'avvio (bootstrap loader), che individua il kernel, lo carica in memoria e ne avvia l'esecuzione.

4 – PROCESSI

4.1 CONCETTO DI PROCESSO

Il processo è l'unità di elaborazione più semplice, che non va confusa con il programma. Ricorda: ad un programma possono corrispondere anche più processi, perché un programma è la stesura di un algoritmo la cui esecuzione può comportare anche la divisione di più compiti simultanei.

Una questione che sorge dall'analisi dei sistemi operativi è quella di dare un nome alle attività della CPU. Un **sistema batch (lotti)** esegue *job* (lavori), mentre un **sistema time-sharing** esegue *programmi utenti* o *task*. Queste attività sono simili per molti aspetti, perciò sono denominate **processi**.

4.1.1 PROCESSO

Informalmente, un processo è un programma in esecuzione. Comprende l'attività corrente, rappresentata dal valore del **program counter** (è una variabile che contiene l'indirizzo nel quale è stata eseguita l'ultima istruzione) e dal contenuto dei registri della CPU; normalmente comprende anche il proprio **stack**, contenente a sua volta i dati temporanei (come i parametri di una procedura, gli indirizzi di rientro e le variabili locali) e una **sezione di dati** contenente le variabili globali.

Sottolineiamo che un programma di per sé non è un processo; un programma è un'entità *passiva*, come un file su disco contenente una lista di istruzioni (normalmente chiamato file eseguibile), mentre un processo è un'entità *attiva*, con un contatore di programma che specifica qual è l'istruzione successiva da eseguire e un insieme di risorse associate. Un programma diventa un processo allorquando il file eseguibile è caricato in memoria.

4.1.2 STATO DEL PROCESSO

Un processo durante l'esecuzione è soggetto a cambiamenti del suo stato, definito in parte dall'attività corrente del processo stesso. Un processo può trovarsi in uno tra i seguenti stati:

- **Nuovo**: si crea il processo.
- **Esecuzione (running)**: e sue istruzioni vengono eseguite.
- **Attesa (waiting)**: il processo attende che si verifichi qualche evento (come il completamento di un'operazione di I/O o la ricezione di un segnale).
- **Pronto (ready)**: il processo attende di essere assegnato a un'unità d'elaborazione.
- **Terminato**: il processo ha terminato l'esecuzione.

Questi termini sono piuttosto arbitrari e variano secondo il sistema operativo. È importante capire che in ciascuna unità d'elaborazione può essere in esecuzione solo un processo per volta, sebbene molti processi possano essere *pronti* o nello stato di *attesa*.

I diagramma di transizione fra questi stati è:

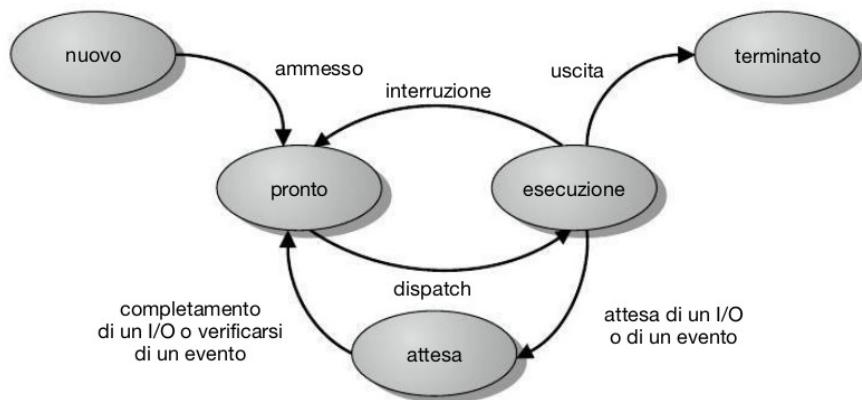


Figura 3.2 Diagramma di transizione degli stati di un processo.

4.1.3 BLOCCO DI CONTROLLO DEI PROCESSI

Ogni processo è rappresentato nel sistema operativo da un **blocco di controllo (process control block, PCB, o task control block, TCB)**. Un PCB contiene molte informazioni connesse a un processo specifico, tra cui le seguenti:

- **Stato del processo**: lo stato può essere: nuovo, pronto, esecuzione, attesa, arresto, e così via.
- **Program Counter**: il contatore di programma contiene l'indirizzo della successiva istruzione da eseguire per tale processo.
- **Registri della CPU**: i registri variano in numero e tipo secondo l'architettura del calcolatore. Essi comprendono accumulatori, registri indice, puntatori alla cima dello stack (*stack pointer*), registri d'uso generale e registri contenenti i codici di condizione (*condition codes*).
- **Informazioni sullo scheduling di CPU**: queste informazioni comprendono la priorità del processo, i puntatori alle code di scheduling e tutti gli altri parametri di scheduling.
- **Informazioni sulla gestione della memoria**: queste informazioni possono includere elementi quali il valore dei registri di base e di limite, le tabelle delle pagine o le tabelle dei segmenti, a seconda del sistema di gestione della memoria usato dal sistema operativo

- **Informazioni di contabilizzazione delle risorse:** queste informazioni comprendono la quota di uso della CPU e il tempo d'utilizzo della stessa, i limiti di tempo, i numeri dei processi, e così via.
- **Informazioni sullo stato dell'I/O:** queste informazioni comprendono la lista dei dispositivi di I/O assegnati a un determinato processo, l'elenco dei file aperti, e così via.

In sintesi, il PCB si usa semplicemente come deposito per tutte le informazioni relative ai vari processi.

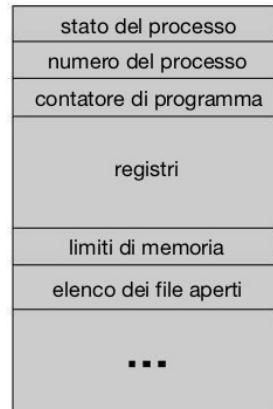


Figura 3.3 Blocco di controllo di un processo (PCB).

La CPU può commutare tra vari processi:

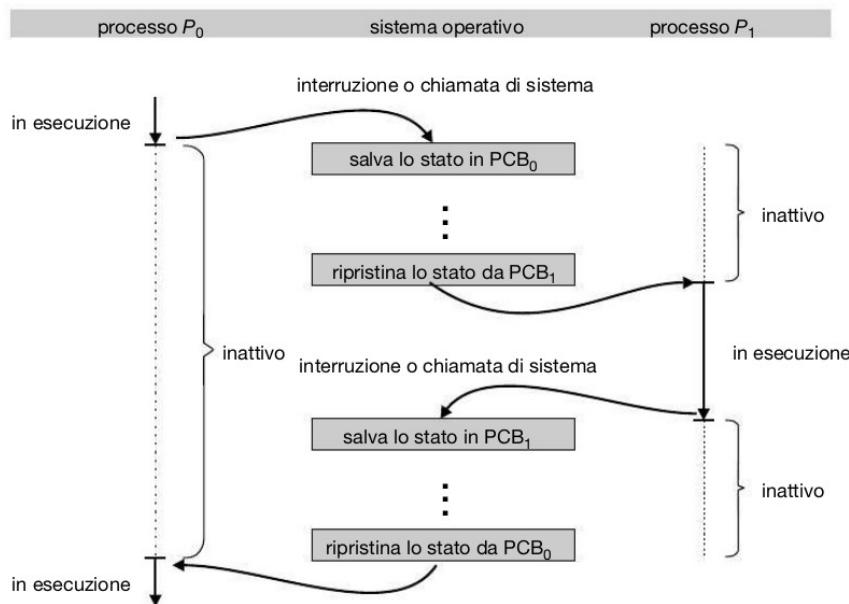


Figura 3.4 La CPU può essere commutata tra i processi.

4.2 SCHEDULING DEI PROCESSI

L'obiettivo della multiprogrammazione consiste nell'avere sempre un processo in esecuzione in modo da massimizzare l'utilizzo della CPU. L'obiettivo del time sharing è di commutare l'uso della CPU tra i vari processi così frequentemente che gli utenti possano interagire con ciascun programma mentre è in esecuzione. Per raggiungere questi obiettivi, lo scheduler dei processi seleziona un processo da eseguire dall'insieme di quelli disponibili.

4.2.1 CODE SCHEDULING

Entrando nel sistema, ogni processo è inserito in una **coda di processi (job queue)**, composta da tutti i processi del sistema.

I processi presenti in memoria centrale, che sono pronti e nell'attesa d'essere eseguiti, si trovano in una lista detta coda dei **processi pronti (ready queue)**. Questa coda generalmente si memorizza come una lista concatenata: un'intestazione della coda dei processi pronti contiene i puntatori al primo e all'ultimo pcB dell'elenco, e ciascun pcB comprende un campo puntatore che indica il successivo processo contenuto nella coda dei processi pronti.

Il sistema operativo ha anche altre code. Quando si assegna la CPU a un processo, quest'ultimo rimane in esecuzione per un certo tempo e prima o poi termina, viene interrotto, oppure si ferma nell'attesa di un evento particolare, come il completamento di una richiesta di I/O. Poiché nel sistema esistono molti processi, il disco può essere occupato con una richiesta di I/O di qualche altro processo, quindi il processo deve attendere che il disco sia disponibile. L'elenco dei processi che attendono la disponibilità di un particolare dispositivo di I/O si chiama **coda del dispositivo**; ogni dispositivo ha la propria coda.

L'appartenenza a una coda non comporta una staticità di appartenenza. Questo significa che anche dopo alcuni istanti un processo può saltare da una coda all'altra in base a quelle che sono le considerazioni e il suo stato.

Una comune rappresentazione dello scheduling dei processi è data da un **diagramma di accodamento**. ogni riquadro rappresenta una coda. Sono presenti due tipi di coda: la ready queue e un insieme di code di dispositivi. I cerchi rappresentano le risorse che servono le code, le frecce indicano il flusso di processi nel sistema.

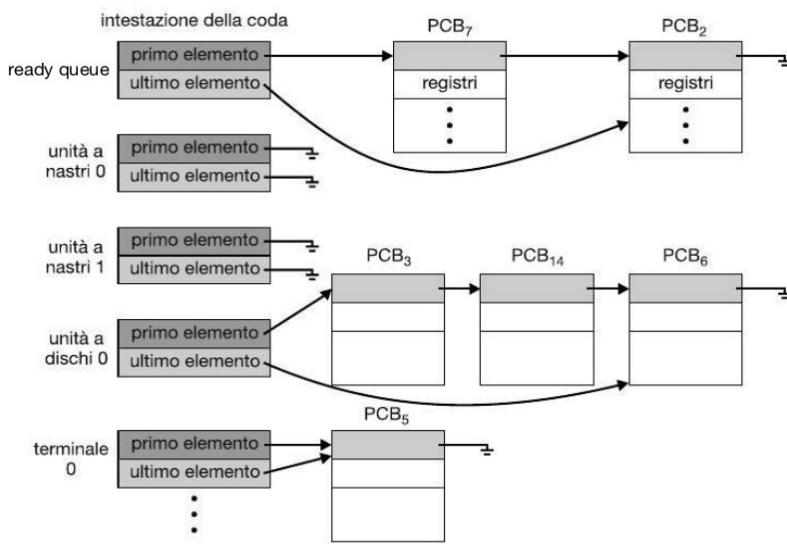


Figura 3.5 Ready queue e diverse code dei dispositivi di I/O.

Un nuovo processo si colloca inizialmente nella ready queue, dove attende finché non è selezionato per essere eseguito (*dispatched*). Una volta che il processo è assegnato alla CPU ed è nella fase d'esecuzione, si può verificare uno dei seguenti eventi:

- Il processo può emettere una richiesta di I/O e quindi essere inserito in una coda di I/O;

- Il processo può creare un nuovo processo figlio e attenderne la terminazione;
- Il processo può essere rimosso forzatamente dalla CPU a causa di un'interruzione, ed essere reinserito nella coda dei processi pronti.

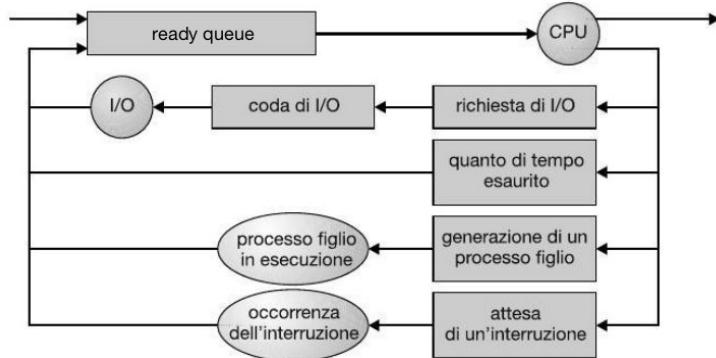


Figura 3.6 Diagramma di accodamento per lo scheduling dei processi.

4.2.2 SCHEDULER

Nel corso della sua esistenza, un processo si trova in varie code di scheduling. Il sistema operativo, incaricato di selezionare i processi dalle suddette code, compie la selezione per mezzo di un opportuno **scheduler**.

Lo **scheduler a lungo termine (job scheduler)**, sceglie i processi da questo insieme e li carica in memoria affinché siano eseguiti. Lo **scheduler a breve termine**, o **scheduler della CPU**, fa la selezione tra i processi pronti per l'esecuzione e assegna la CPU a uno di loro.

Questi due scheduler si differenziano principalmente per la frequenza con la quale sono eseguiti. Lo scheduler a breve termine seleziona frequentemente un nuovo processo per la CPU. Un processo può rimanere in esecuzione solo per pochi millisecondi prima di passare ad attendere una richiesta di I/O. Lo scheduler a lungo termine, invece, si esegue con una frequenza molto inferiore; diversi minuti possono trascorrere tra la creazione di un nuovo processo e il successivo. Lo scheduler a lungo termine controlla il **grado di multiprogrammazione**, cioè il numero di processi presenti in memoria. Se è stabile, la velocità media di creazione dei processi deve essere uguale alla velocità media con cui i processi abbandonano il sistema; quindi lo scheduler a lungo termine si può richiamare solo quando un processo abbandona il sistema. A causa del maggior intervallo che intercorre tra le esecuzioni, lo scheduler a lungo termine dispone di più tempo per scegliere un processo per l'esecuzione.

È importante che lo scheduler a lungo termine faccia un'accurata selezione dei processi. In generale, la maggior parte dei processi si può caratterizzare come avente una prevalenza di I/O, o come avente una prevalenza d'elaborazione. Un **processo con prevalenza di I/O (I/O bound)** impiega la maggior parte del proprio tempo nell'esecuzione di operazioni di I/O. Un **processo con prevalenza d'elaborazione (CPU bound)**, viceversa, richiede poche operazioni di I/O e impiega la maggior parte del proprio tempo nelle elaborazioni.

In alcuni sistemi operativi come quelli in time-sharing, si può introdurre un livello di scheduling intermedio. Questo **scheduler a medio termine** è rappresentato schematicamente in questo modo:

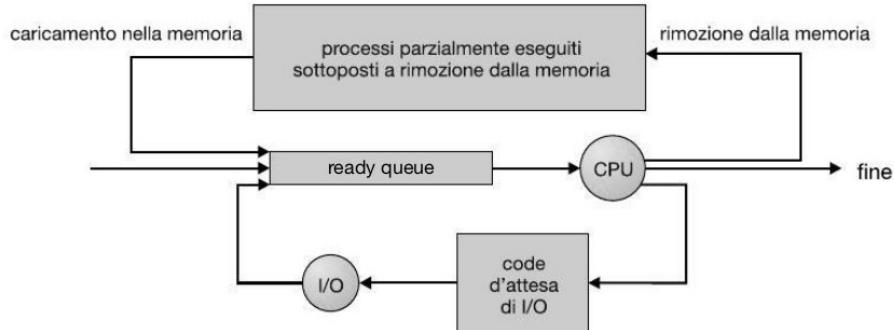


Figura 3.7 Aggiunta di scheduling a medio termine al diagramma di accodamento.

L'idea alla base di un tale scheduler è che a volte può essere vantaggioso eliminare processi dalla memoria (e dalla contesa per la CPU), riducendo il grado di multiprogrammazione del sistema. In seguito, il processo può essere reintrodotto in memoria, in modo che la sua esecuzione riprenda da dove era stata interrotta.

Questo schema si chiama **avvicendamento dei processi in memoria (swapping)**. Il processo viene rimosso e successivamente ricaricato in memoria dallo scheduler a medio termine. L'avvicendamento dei processi in memoria può servire a migliorare la combinazione di processi, oppure a liberare una parte della memoria se un cambiamento dei requisiti di memoria ha impegnato eccessivamente la memoria disponibile.

4.2.3 CAMBIO DI CONTESTO

Come spiegato le interruzioni forzano il sistema a sospendere il lavoro attuale della cpU per eseguire routine del kernel. In presenza di una interruzione, il sistema deve salvare il **contesto** del processo corrente, per poterlo poi ripristinare quando il processo stesso potrà ritornare in esecuzione.

Il contesto è rappresentato all'interno del PCB del processo, e comprende i valori dei registri della cpU, lo stato del processo, e informazioni relative alla gestione della memoria.

In termini generali, si esegue un salvataggio dello stato corrente della cpU, sia che essa esega in modalità utente o in modalità di sistema; in seguito, si attuerà un corrispondente ripristino dello stato per poter riprendere l'elaborazione dal punto in cui era stata interrotta.

Il passaggio della cpU a un nuovo processo implica il salvataggio dello stato del processo attuale e il ripristino dello stato del nuovo processo. Questa procedura è nota col nome di **cambio di contesto (context switch)**. Nell'evenienza di un cambio di contesto, il sistema salva il contesto del processo uscente nel suo PCB, e carica il contesto del processo subentrante, salvato in precedenza. Il cambio di contesto è puro overhead, perché il sistema esegue solo operazioni volte alla gestione dei processi, e non alla computazione. La durata del cambio di contesto dipende molto dall'architettura.

4.3 OPERAZIONI SUI PROCESSI

4.3.1 CREAZIONE DI UN PROCESSO

Durante la propria esecuzione, un processo può creare numerosi nuovi processi. Il processo creante si chiama processo **genitore** (o **padre**), mentre il nuovo processo si chiama processo **figlio**. Ciascuno di questi nuovi processi può creare a sua volta altri processi, formando un **albero** di processi.

La maggior parte dei sistemi operativi (compresi UNIX, Linux e Windows) identifica un processo per mezzo di un numero univoco, solitamente un intero, detto **identificatore del processo** o **pid** (*process identifier*). Il pid fornisce un valore univoco per ogni processo del sistema e può essere usato come indice per accedere a vari attributi di un processo all'interno del kernel.

In generale, quando un processo crea un processo figlio, quest'ultimo avrà bisogno di determinate risorse (tempo d'elaborazione, memoria, file, dispositivi di I/o) per eseguire il proprio compito. Un processo figlio può essere in grado di ottenere le proprie risorse direttamente dal sistema operativo, oppure può essere vincolato a un sottoinsieme delle risorse del processo genitore. Il processo genitore può avere la necessità di spartire le proprie risorse tra i suoi processi figli, oppure può essere in grado di condividerne alcune, come la memoria o i file, tra più processi figli. Limitando le risorse di un processo figlio a un sottoinsieme di risorse del processo genitore si può evitare che un processo sovraccarichi il sistema creando troppi processi figlio.

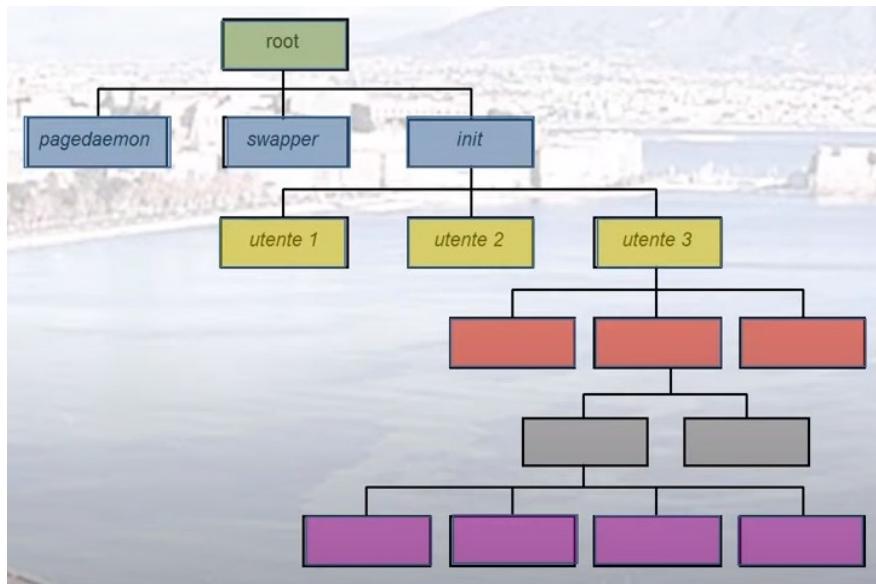
Quando un processo ne crea uno nuovo, per quel che riguarda l'esecuzione ci sono due possibilità:

- Il processo genitore continua l'esecuzione in modo concorrente con i propri processi figli;
- Il processo genitore attende che alcuni o tutti i suoi processi figli terminino.

Ci sono due possibilità anche per quel che riguarda lo spazio d'indirizzi del nuovo processo:

- Il processo figlio è un duplicato del processo genitore (ha gli stessi programma e dati del genitore);
- Nel processo figlio si carica un nuovo programma.

Per illustrare queste differenze, consideriamo dapprima il sistema operativo UNIX. In UNIX, come abbiamo visto, ogni processo è identificato dal proprio identificatore di processo, un intero univoco. Un nuovo processo si crea per mezzo della chiamata di sistema `fork()`, ed è composto di una copia dello spazio degli indirizzi del processo genitore. Questo meccanismo permette al processo genitore di comunicare senza difficoltà con il proprio processo figlio. Generalmente, dopo una chiamata di sistema `fork()`, uno dei due processi impiega una chiamata di sistema `exec()` per sostituire lo spazio di memoria del processo con un nuovo programma.



Albero di processi di un tipico sistema UNIX

4.3.2 TERMINAZIONE DI UN PROCESSO

Un processo termina quando finisce l'esecuzione della sua ultima istruzione e inoltra la richiesta al sistema operativo di essere cancellato usando la chiamata di sistema `exit()`; a questo punto, il processo figlio può riportare un'informazione di stato al processo genitore, che la riceve attraverso la chiamata di sistema `wait()`.

Occorre notare che un genitore deve conoscere le identità dei propri figli per terminarli, perciò quando un processo ne crea uno nuovo, l'identità del nuovo processo viene passata al processo genitore.

Un processo genitore può porre termine all'esecuzione di uno dei suoi processi figli (`abort`) per diversi motivi, tra i quali i seguenti:

- Il processo figlio ha ecceduto nell'uso di alcune tra le risorse che gli sono state assegnate. ciò richiede che il processo genitore disponga di un sistema che esamini lo stato dei propri processi figli.
- Il compito assegnato al processo figlio non è più richiesto.
- Il processo genitore termina e il sistema operativo non consente a un processo figlio di continuare l'esecuzione in tale circostanza.

In alcuni sistemi, se un processo termina si devono terminare anche i suoi figli, indipendentemente dal fatto che la terminazione del genitore sia stata normale o anormale. Si parla di **terminazione a cascata**, una procedura avviata di solito dal sistema operativo.

4.4 COMUNICAZIONE TRA PROCESSI

I processi eseguiti concorrentemente nel sistema operativo possono essere indipendenti o cooperanti. Un processo è **indipendente** se non può influire su altri processi del sistema o subirne l'influsso. chiaramente, un processo che non condivide dati (temporanei o permanenti) con altri processi è indipendente. D'altra parte un processo è **cooperante** se influenza o può essere influenzato da altri processi in esecuzione nel sistema. Ovviamente, qualsiasi processo che condivide dati con altri processi è un processo cooperante.

Un ambiente che consente la cooperazione tra processi può essere utile per diverse ragioni:

- **Condivisione di informazioni**: poiché più utenti possono essere interessati alle stesse informazioni (per esempio un file condiviso) è necessario offrire un ambiente che consenta un accesso concorrente a tali risorse.
- **Accelerazione del calcolo**: alcune attività d'elaborazione sono realizzabili più rapidamente se si suddividono in sottoattività eseguibili in parallelo.
- **Modularità**: può essere utile la costruzione di un sistema modulare, che suddivide le funzioni di sistema in processi o thread distinti
- **Convenienza**: anche un solo utente può avere la necessità di compiere più attività contemporaneamente; per esempio, può eseguire in parallelo le operazioni di scrittura, ascolto di musica e compilazione.

Per lo scambio di dati e informazioni i processi cooperanti necessitano di un meccanismo di **comunicazione tra processi (IPC, interprocess communication)**.

4.4.1 SISTEMI A MEMORIA CONDIVISA

Per illustrare il concetto di cooperazione tra processi si consideri il problema del produttore/consumatore; tale problema è un comune paradigma per processi cooperanti. Un processo **produttore** produce informazioni che sono consumate da un processo **consumatore**.

Una possibile soluzione del problema del produttore/consumatore si basa sulla memoria condivisa. L'esecuzione concorrente dei due processi richiede la presenza di un buffer che possa essere riempito dal produttore e svuotato dal consumatore. Il buffer dovrà risiedere in una zona di memoria condivisa dai due processi. Il produttore potrà allora produrre un'unità mentre il consumatore ne consuma un'altra. I due processi devono essere sincronizzati in modo tale che il consumatore non tenti di consumare un'unità non ancora prodotta.

Si possono utilizzare due tipi di buffer. Quello **illimitato** non pone limiti pratici alla dimensione del buffer. Il consumatore può dover attendere nuovi oggetti, ma il produttore può sempre produrne. Il problema del produttore e del consumatore con **buffer limitato** presuppone una dimensione fissa del buffer in questione. In questo caso, il consumatore deve attendere se il buffer è vuoto; viceversa, il produttore deve attendere se il buffer è pieno.

Consideriamo più attentamente in che modo il buffer limitato illustra la comunicazione tra processi con memoria condivisa. Le variabili seguenti risiedono in una zona di memoria condivisa sia dal produttore sia dal consumatore.

```
#define BUFFER_SIZE 10

typedef struct {
    . . .
} elemento;

elemento buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
```

```

item next_produced;
while (true) {
    /* produce un elemento in next_produced */
    while (((in + 1) % BUFFER_SIZE) == out); /* non fa niente */
    buffer[in] = next_produced;
    in = (in + 1) % BUFFER_SIZE;
}

```

Figura 3.13 Processo produttore con l'utilizzo della memoria condivisa.

Il buffer condiviso è realizzato come un array circolare con due puntatori logici: `in` e `out`. La variabile `in` indica la successiva posizione libera nel buffer; `out` indica la prima posizione piena nel buffer. Il buffer è vuoto se `in == out`; è pieno se `((in + 1) % BUFFER_SIZE) == out`.

Il processo produttore ha una variabile locale `next_produced` contenente il nuovo elemento da produrre. Il processo consumatore ha una variabile locale `next_consumed` in cui si memorizza l'elemento da consumare.

```

item next_consumed;

while (true) {
    while (in == out)
        ; /* non fa niente */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    /* consuma l'elemento in next_consumed */
}

```

Figura 3.14 Processo consumatore con l'utilizzo della memoria condivisa.

4.4.2 SISTEMI A SCAMBIO DI MESSAGGI

Lo scambio di messaggi è un meccanismo che permette a due o più processi di comunicare e di sincronizzarsi senza condividere lo stesso spazio di indirizzi. È una tecnica particolarmente utile negli ambienti distribuiti, dove i processi possono risiedere su macchine diverse connesse da una rete.

Un meccanismo per lo scambio di messaggi deve prevedere almeno due operazioni: `send(message)` e `receive(message)`.

I messaggi possono avere lunghezza fissa o variabile. Nel primo caso, l'implementazione a livello del sistema è elementare, ma programmare applicazioni diviene più complicato. Nel secondo caso, l'implementazione del meccanismo a livello di sistema è più complessa, mentre la programmazione utente risulta semplificata.

Se i processi P e Q vogliono comunicare devono inviare e ricevere messaggi tra loro; deve, dunque, esistere un **canale di comunicazione** (*communication link*), realizzabile in molti modi.

Ci sono diversi metodi per realizzare a livello logico un canale di comunicazione e le operazioni `send()` e `receive()`:

- Comunicazione diretta o indiretta;
- Comunicazione sincrona o asincrona;
- Gestione automatica o esplicita del buffer.

4.4.3 NAMING

Per comunicare, i processi devono disporre della possibilità di far riferimento ad altri processi; a tale scopo è possibile servirsi di una comunicazione diretta oppure indiretta.

Con la comunicazione diretta, ogni processo che intenda comunicare deve nominare esplicitamente il ricevente o il trasmittente della comunicazione. In questo schema le funzioni primitive `send()` e `receive()` si definiscono come segue:

- `send(P, messaggio)`, invia messaggio al processo P;
- `receive(Q, messaggio)`, riceve messaggio dal processo Q.

All'interno di questo schema un canale di comunicazione ha le seguenti caratteristiche:

- Tra ogni coppia di processi che intendono comunicare si stabilisce automaticamente un canale; i processi devono conoscere solo la reciproca identità;
- Un canale è associato esattamente a due processi;
- Esiste esattamente un canale tra ciascuna coppia di processi.

Questo schema ha una simmetria nell'indirizzamento, vale a dire che per poter comunicare, il trasmittente e il ricevente devono nominarsi a vicenda. Una variante di questo schema si avvale dell'asimmetria nell'indirizzamento: soltanto il trasmittente nomina il ricevente, mentre il ricevente non ha bisogno di nominare il trasmittente. In questo schema le primitive `send()` e `receive()` si definiscono come segue:

- `send(P, messaggio)`, invia messaggio al processo P;
- `receive(id, messaggio)`, riceve un messaggio da qualsiasi processo; nella variabile `id` si ottiene il nome del processo con cui è avvenuta la comunicazione.

Entrambi gli schemi, simmetrico e asimmetrico, hanno lo svantaggio di una limitata modularità delle risultanti definizioni dei processi. La modifica del nome di un processo può infatti implicare la necessità di un riesame di tutte le altre definizioni dei processi, individuando tutti i riferimenti al vecchio nome allo scopo di sostituirli con il nuovo.

Con la **comunicazione indiretta** i messaggi s'inviano a delle **porte** o **mailbox**, che li ricevono. Una mailbox si può considerare in modo astratto come un oggetto in cui i processi possono introdurre e prelevare messaggi, ed è identificata in modo unico.

In questo schema un processo può comunicare con altri processi tramite un certo numero di mailbox e due processi possono comunicare solo se condividono una mailbox. Le primitive `send()` e `receive()` si definiscono come segue:

- `send(A, messaggio)`, invia messaggio alla mailbox A;
- `receive(A, messaggio)`, riceve un messaggio dalla mailbox A.

In questo schema un canale di comunicazione ha le seguenti caratteristiche:

- Tra una coppia di processi si stabilisce un canale solo se entrambi i processi della coppia condividono una stessa mailbox;
- Un canale può essere associato a più di due processi;
- Tra ogni coppia di processi comunicanti possono esserci più canali diversi,

ciascuno corrispondente a una mailbox.

4.4.4 SINCRONIZZAZIONE

Lo scambio di messaggi può essere **sincrono** (o bloccante) oppure **asincrono** (o non bloccante).

- **Invio sincrono**: il processo che invia il messaggio si blocca nell'attesa che il processore ricevente, o la mailbox, riceva il messaggio.
- **Invio asincrono**: il processo invia il messaggio e riprende la propria esecuzione.
- **Ricezione sincrona**: il ricevente si blocca nell'attesa dell'arrivo di un messaggio.
- **Ricezione asincrona**: il ricevente riceve un messaggio valido o un valore nullo.

4.4.5 CODE DI MESSAGGI

Se la comunicazione è diretta o indiretta, i messaggi scambiati tra processi comunicanti risiedono in code temporanee. Fondamentalmente esistono tre modi per realizzare queste code:

- **Capacità zero**: La coda ha lunghezza massima 0, quindi il canale non può avere messaggi in attesa al suo interno. In questo caso il trasmittente deve fermarsi finché il ricevente prende in consegna il messaggio.
- **Capacità limitata**: La coda ha lunghezza finita n , quindi al suo interno possono risiedere al massimo n messaggi. Se la coda non è piena, quando s'invia un nuovo messaggio, quest'ultimo è posto in fondo alla coda. Il trasmittente può proseguire la propria esecuzione senza essere costretto ad attendere. Il canale ha tuttavia una capacità limitata; se è pieno, il trasmittente deve fermarsi nell'attesa che ci sia spazio disponibile nella coda.
- **Capacità illimitata**: La coda ha una lunghezza potenzialmente infinita, quindi al suo interno può attendere un numero indefinito di messaggi. Il trasmittente non si ferma mai.

4.5 COMUNICAZIONE NEI SISTEMI CLIENT-SERVER

Fino ad ora ci siamo soffermati su come i processi possano comunicare usando memoria condivisa e scambio di messaggi. Tali tecniche sono utilizzabili anche per la comunicazione in sistemi client/server. Consideriamo qui altre tre strategie: socket, chiamate di procedura remota (RPC) e pipe.

4.5.1 SOCKET

Una socket è definita come l'estremità di un canale di comunicazione. Una coppia di processi che comunicano attraverso una rete usa una coppia di socket, una per ogni processo, e ogni socket è identificata da un indirizzo IP concatenato a un numero di porta. Ad esempio la socket **161.25.19.8:1625** si riferisce alla porta **1625** sul calcolatore **161.25.19.8**.

In generale, le socket impiegano un'architettura client-server; il server attende le richieste dei client, stando in ascolto a una porta specificata; quando il server riceve una richiesta, accetta la connessione proveniente dalla socket del client, e si stabilisce la comunicazione. Tutte le porte al di sotto del valore 1024 sono considerate ben note (*well known*) e si usano per realizzare servizi standard.

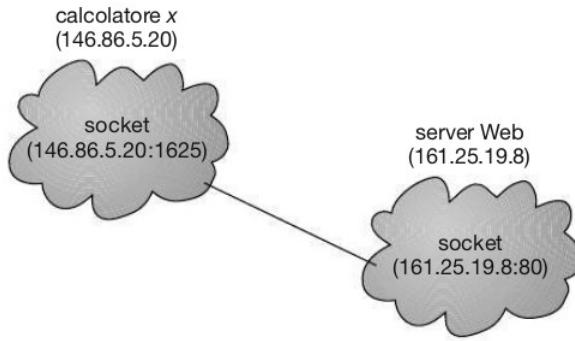


Figura 3.20 Comunicazione tramite socket.

4.5.2 CHIAMATE DI PROCEDURE REMOTE

La RPC è stata progettata per astrarre il meccanismo della chiamata di procedura affinché si possa usare tra sistemi collegati tramite una rete.

Per molti aspetti è simile al meccanismo IPC, ed è generalmente costruita su un sistema di questo tipo. Contrariamente ai messaggi IPC, i messaggi scambiati per la comunicazione RPC sono ben strutturati e non semplici pacchetti di dati. Si indirizzano a un demone RPC, in ascolto a una porta del sistema remoto, e contengono un identificatore della funzione da eseguire e i parametri da passare a tale funzione. Nel sistema remoto si esegue questa funzione e s'invia ogni risultato al richiedente in un messaggio distinto.

La **porta** è semplicemente un numero inserito all'inizio del messaggio. Mentre un singolo sistema ha normalmente un solo indirizzo di rete, all'interno dell'indirizzo può avere molte porte che servono a distinguere i numerosi servizi che può fornire in rete. Se un processo remoto richiede un servizio, indirizza i propri messaggi alla porta corrispondente.

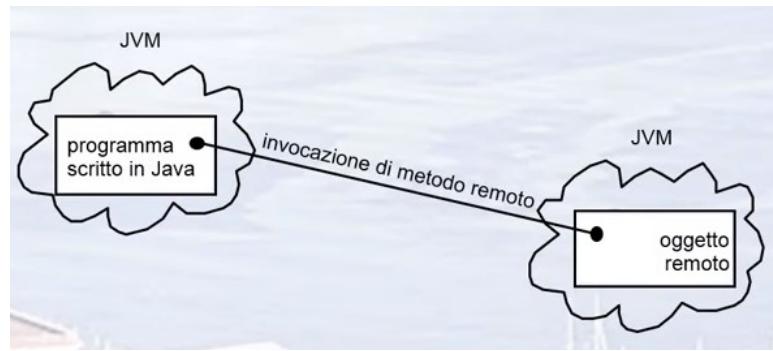
La semantica delle RPC permette a un client di richiamare una procedura presente in un sistema remoto nello stesso modo in cui invocherebbe una procedura locale. Il sistema delle RPC nasconde i dettagli necessari che consentono la comunicazione, fornendo uno **stub** al lato client. Esiste in genere uno stub per ogni diversa procedura remota.

Quando il client la invoca, il sistema delle rpc richiama l'appropriato stub, passando i parametri della procedura remota. Lo stub individua la porta del server e struttura i parametri; la strutturazione dei parametri (*marshalling*) implica l'assemblaggio dei parametri in una forma che si può trasmettere tramite una rete. Lo stub quindi trasmette un messaggio al server usando lo scambio di messaggi. Un analogo stub nel server riceve questo messaggio e invoca la procedura nel server; se è necessario, riporta i risultati al client usando la stessa tecnica.

4.5.3 INVOCAZIONE DEI METODI REMOTI

L'invocazione di metodi remoti (RMI, *remote method invocation*) è una funzione del linguaggio Java simile alla RPC.

Una RMI consente a un thread di invocare un metodo di un oggetto remoto.



5 – THREADS

5.1 INTRODUZIONE

Quasi tutti i sistemi operativi moderni permettono tuttavia che un processo possa avere più percorsi di controllo chiamati *thread*. Un thread è l'unità di base d'uso della CPU e comprende un identificatore di thread (*ID*), un contatore di programma, un insieme di registri, e una pila (*stack*).

Un thred è un singolo flusso sequenziale di controllo all'interno di un processo. Un processo può contenere più thread. Tutti i thread di un processo condividono lo stesso spazio di indirizzamento.

Un processo tradizionale, chiamato anche **processo pesante** (*heavyweight process*), è composto da un solo thread. Un processo multithread è in grado di svolgere più compiti in modo concorrente.

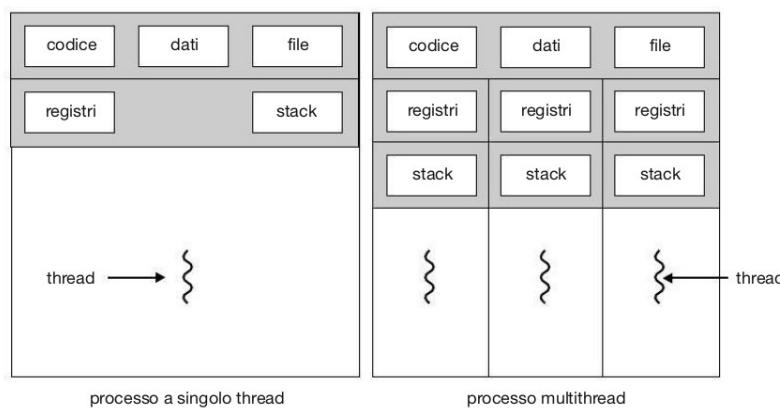


Figura 4.1 Processi a singolo thread e multithread.

5.1.1 MOTIVAZIONI DEI THREAD

È spesso necessario dividere il programma anche in “sotto-compiti” indipendenti.

Un programma può avere diverse funzioni concorrenti:

- Operazioni ripetute nel tempo ad intervalli regolari (es. animazioni);
- Esecuzione di compiti laboriosi senza bloccare la GUI del programma;
- Attesa di messaggi da un altro programma;
- Attesa di input da tastiera o dalla rete;

L'alternativa al multithreading è il polling. Questo, oltre che scomodo da implementare, consuma molte risorse di CPU. È quindi estremamente inefficiente.

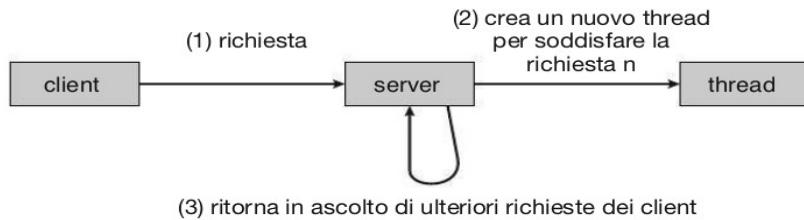


Figura 4.2 Architettura di server multithread.

5.1.2 VANTAGGI

I vantaggi della programmazione multithread si possono classificare in quattro categorie principali:

- **Tempo di risposta:** rendere multithread un'applicazione interattiva può permettere a un programma di continuare la sua esecuzione, anche se una parte di esso è bloccata o sta eseguendo un'operazione particolarmente lunga, riducendo il tempo di risposta all'utente. Questa caratteristica è particolarmente utile nella progettazione di interfacce utente.
- **Condivisione delle risorse:** i processi possono condividere risorse soltanto attraverso tecniche come la memoria condivisa e lo scambio di messaggi. Queste tecniche devono essere esplicitamente messe in atto dal programmatore. Tuttavia, i thread condividono per default la memoria e le risorse del processo al quale appartengono. Il vantaggio della condivisione del codice e dei dati consiste nel fatto che un'applicazione può avere molti thread di attività diverse, tutti nello stesso spazio d'indirizzi.
- **Economia:** assegnare memoria e risorse per la creazione di nuovi processi è costoso; poiché i thread condividono le risorse del processo cui appartengono, è molto più conveniente creare thread e gestirne i cambi di contesto.
- **Scalabilità:** i vantaggi della programmazione multithread sono ancora maggiori nelle architetture multiprocessore, dove i thread si possono eseguire in parallelo su distinti core di elaborazione. Invece un processo con un singolo thread può funzionare solo su un processore, indipendentemente da quanti ve ne siano a disposizione. Il multithreading su una macchina con più processori incrementa il parallelismo.

5.2 PROGRAMMAZIONE MULTICORE

Il progetto dell'architettura dei sistemi consiste nel montare diversi core di elaborazione su un unico chip; ogni unità appare al sistema operativo come un processore separato. Sia che i core appartengano allo stesso chip o a più chip, noi chiameremo questi sistemi multicore o multiprocessore.

Si consideri un'applicazione con quattro thread. In un sistema con un singolo core, "esecuzione concorrente" significa solo che l'esecuzione dei thread è avvicendata nel tempo (*interleaved*), perché la CPU è in grado di eseguire un solo thread alla volta. Il parallelismo di processi e thread viene simulato allocando a ciascuno una frazione del tempo della CPU (*time slice*). Allo scadere di ogni unità di tempo, il sistema operativo opera un cambio di contesto (*context switch*). In questo caso i thread sono processi leggeri perché il cambio di contesto è veloce. Su un sistema multicore, invece, "esecuzione concorrente" significa che i thread possono funzionare in parallelo, dal momento che il sistema può assegnare thread diversi a ciascun core.

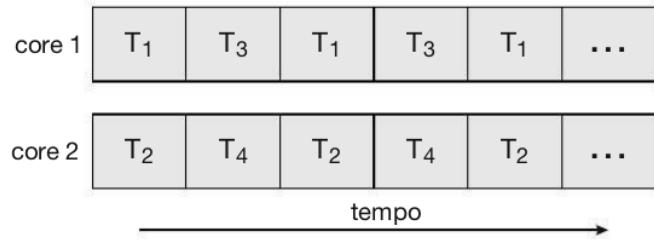


Figura 4.4 Esecuzione parallela su un sistema multicore.

5.3 PREEMPTION (PRELAZIONE)

Quando un processo, un thread, viene eseguito dalla CPU. Dobbiamo fare un'ipotesi: questa esecuzione del thread è un'esecuzione interrompibile? A seconda della risposta a questa domanda possiamo avere due diversi tipi di sistemi:

- **Sistema non preemptive**: il cambio di contesto avviene quando il processo o il thread interrompe la propria esecuzione o volontariamente, o perché in attesa di un evento (input, output);
- **Sistema preemptive**: allo scadere del *time slice* il processo o il thread viene forzatamente interrotto e viene operato il cambio contesto;

5.4 MODELLI DI SUPPORTO AL MULTITHREADING

I thread possono essere distinti in thread a **livello utente** e thread a **livello kernel**: i primi sono gestiti sopra il livello del kernel e senza il suo supporto; i secondi, invece, sono gestiti direttamente dal sistema operativo. Praticamente tutti i sistemi operativi moderni supportano i thread nel kernel, compresi Windows, Linux, mac OS ,X e Solaris.

5.4.1 MODELLO DA MOLTI A UNO

Il modello da molti a uno fa corrispondere molti thread a livello utente a un singolo thread a livello kernel. La gestione dei thread risulta efficiente perché viene effettuata da una libreria di thread nello spazio utente. Tuttavia l'intero processo rimane bloccato se un thread invoca una chiamata di sistema di tipo bloccante. Inoltre, poiché un solo thread alla volta può accedere al kernel, è impossibile eseguire thread multipli in parallelo in sistemi multicore.

Tuttavia, pochissimi sistemi utilizzano ancora questo modello a causa della sua incapacità di trarre vantaggio dalla presenza di più core.

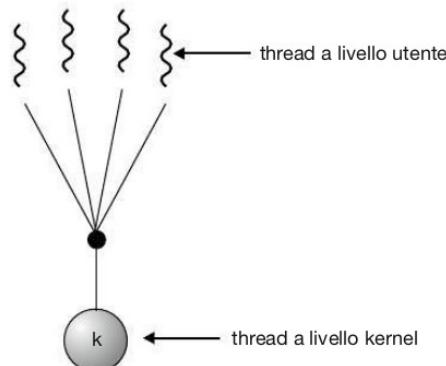


Figura 4.5 Modello da molti a uno.

5.4.2 MODELLO DA UNO A UNO

Il modello da uno a uno mette in corrispondenza ciascun thread a livello utente con un thread a livello kernel. Questo modello offre un grado di concorrenza maggiore rispetto al precedente, poiché anche se un thread invoca una chiamata di sistema bloccante, è possibile eseguire un altro thread; il modello permette anche l'esecuzione dei thread in parallelo nei sistemi multiprocessore.

L'unico svantaggio di questo modello è che la creazione di ogni thread a livello utente comporta la creazione del corrispondente thread a livello kernel, poiché il carico dovuto alla creazione di un thread a livello kernel può sovraccaricare le prestazioni di un'applicazione, la maggior parte delle realizzazioni di questo modello limita il numero di thread supportabili dal sistema.

I sistemi operativi Linux, insieme alla famiglia dei sistemi operativi Windows, adottano il modello da uno a uno.

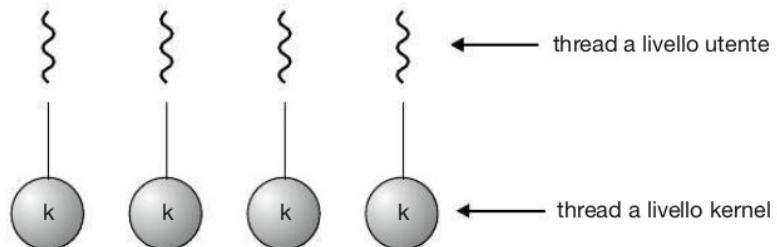


Figura 4.6 Modello da uno a uno.

5.4.3 MODELLI DA MOLTI A MOLTI

Il modello da molti a molti mette in corrispondenza più thread a livello utente con un numero minore o uguale di thread a livello kernel; quest'ultimo può essere specifico per una certa applicazione o per un particolare calcolatore. Consideriamo l'effetto di questo modello sulla concorrenza. Nonostante il modello da molti a uno permetta ai programmati di creare tanti thread a livello utente quanti ne desiderino, non viene garantita una concorrenza reale, poiché il meccanismo di scheduling del kernel può scegliere un solo thread alla volta.

Il modello da uno a uno permette una maggiore concorrenza, ma i programmati devono stare attenti a non creare troppi thread all'interno di un'applicazione (in qualche caso si possono avere limitazioni sul numero di thread che si possono creare). Il modello da molti a molti non ha alcuno di questi difetti: i programmati possono creare liberamente i thread che ritengono necessari, e i corrispondenti thread a livello kernel si possono eseguire in parallelo nelle architetture multiprocessore. Inoltre, se un thread impiega una chiamata di sistema bloccante, il kernel può schedulare un altro thread.

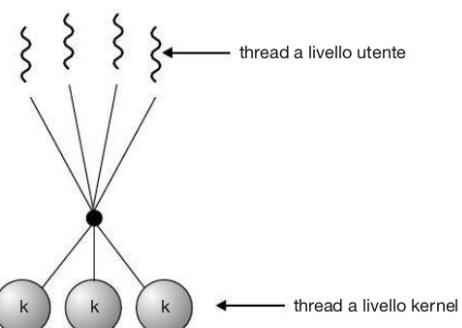


Figura 4.7 Modello da molti a molti.

5.5 LIBRERIE DI THREAD

5.5.1 PTHREADS

Col termine **Pthreads** ci si riferisce allo standard POSIX (IEEE 1003.1c) che definisce una API per la creazione e la sincronizzazione dei thread. Non si tratta di una *implementazione*, ma di una *specifica del comportamento* dei thread; i progettisti di sistemi operativi possono realizzare la specifica come meglio credono. Sono molti i sistemi che implementano le specifiche pthreads; per la maggior parte si tratta di sistemi di tipo UNIX, tra cui Linux, mac OS X e Solaris.

Vediamo un esempio:

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

int sum; //questo dato è condiviso dai thread
void *runner(void *param) //i thread chiamano questa funzione

int main (int argc, char *argv[]){
    pthread_t tid; //identificatore di thread
    pthread_attr_t attr; //insieme di attributi del thread

    if (argc != 2) {
        fprintf(stderr,"usage: a.out <integer value>\n");
        return -1;
    }

    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"%d must be >= 0\n", atoi(argv[1]));
        return -1;
    }

    //reperisce gli attributi predefiniti
    pthread_attr_init(&attr);
    //crea il thread
    pthread_create(&tid,&attr,runner,argv[1]);
    //attende la terminazione del thread
    pthread_join(tid,NULL);

    printf("sum = %d\n",sum);
}

//Il thread comincia l'esecuzione da questa funzione
void *runner(void *param){
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}
```

Il seguente programma è un programma C multithread che esemplifica la API pthreads tramite il calcolo di una sommatoria eseguito da un thread apposito. Nei programmi pthreads, i nuovi thread iniziano l'esecuzione a partire da una funzione specificata. Nel programma in esame si tratta della funzione `runner()`.

All'inizio dell'esecuzione del programma c'è un unico thread di controllo che parte da `main()`; dopo una fase d'inizializzazione, `main()` crea un secondo thread che inizia l'esecuzione dalla funzione `runner()`. Entrambi i thread condividono la variabile globale `sum`.

5.5.2 THREAD IN WINDOWS

Illustriamo la API Windows nel programma C:

```
#include <windows.h>
#include <stdio.h>
DWORD Sum; //il dato è condiviso tra i thread

//il thread viene eseguito in questa funzione separata
WORD WINAPI Summation(LPVOID Param) {
    WORD Upper = *(WORD*)Param;
    for (WORD i = 0; i <= Upper; i++)
        Sum += i;

    return 0;
}

int main(int argc, char *argv[]) {
    WORD ThreadId;
    HANDLE ThreadHandle;
    int Param;

    if (argc != 2) {
        fprintf(stderr,"An integer parameter is required\n");
        return -1;
    }

    Param = atoi(argv[1]);
    if (Param < 0) {
        fprintf(stderr,"An integer >= 0 is required\n");
        return -1;
    }

    //crea il thread
    ThreadHandle = CreateThread(
        NULL, //attributi di sicurezza di default
        0, //dimensione di default dello stack
        Summation, //funzione del thread
        &Param, //parametri alla funzione del thread
        0, //flag di creazione di default
        &ThreadId); //restituisce l'identificatore del thread

    if (ThreadHandle != NULL) {
        //adesso aspetta la fine del thread
        WaitForSingleObject(ThreadHandle, INFINITE);

        //chiude l'handle del thread
        CloseHandle(ThreadHandle);

        printf("sum = %d\n", Sum);
    }
}
```

La tecnica usata dalla libreria Windows per la creazione dei thread può richiamare, per molti versi, quella di Pthreads. Si noti che per utilizzare la api Windows è necessario includere il file d'intestazione windows.h.

Nella API Windows i nuovi thread si generano tramite la funzione `Create Thread()`, che – proprio come in Pthreads – accetta una serie di attributi del thread come parametri. Tali attributi includono le informazioni sulla sicurezza, la dimensione dello stack e un indicatore (*flag*) per segnalare se il thread debba avere inizio nello stato d'attesa. Una volta creato il nuovo thread, il thread iniziale deve attenderne il completamento prima di produrre in uscita il valore di `Sum`, poiché esso è computato dal nuovo thread.

5.5.3 THREAD JAVA

I thread rappresentano il paradigma fondamentale per l'esecuzione dei programmi in ambiente Java; il linguaggio Java, con la propria api, è provvisto di una ricca gamma di funzionalità per la generazione e la gestione dei thread. Tutti i programmi scritti in Java incorporano almeno un thread di controllo – persino un semplice programma, costituito soltanto da un metodo `main()`, è eseguito dalla Jvm come un singolo thread.

In un programma Java vi sono due tecniche per la generazione dei thread. Una è creare una nuova classe derivata dalla classe `Thread` e “sovrascrivere” (*override*) il suo metodo `run()`. L'alternativa, usata più comunemente, consiste nella definizione di una classe che implementi l'interfaccia `Runnable`, definita come segue:

```
public interface Runnable {  
    public abstract void run();  
}
```

Per implementare `Runnable`, una classe è tenuta a definire il metodo `run()`. Il codice che implementa `run()` sarà eseguito in un thread distinto.

5.6 PROBLEMATICA DI PROGRAMMAZIONE MULTITHREAD

5.6.1 CHIAMATE DI SISTEMA `FORK()` ED `EXEC()`

In un programma multithread la semantica delle chiamate di sistema `fork()` ed `exec()` cambia.

Se un thread in un programma invoca la chiamata di sistema `fork()`, il nuovo processo potrebbe, in generale, contenere un duplice di tutti i thread oppure del solo thread invocante. Alcuni sistemi UNIX includono entrambe le versioni.

La chiamata di sistema `exec()` di solito funziona nello stesso modo descritto precedentemente: se un thread invoca la chiamata di sistema `exec()`, il programma specificato come parametro della `exec()` sostituisce l'intero processo, inclusi tutti i thread.

L'uso delle due versioni della `fork()` dipende dall'applicazione. Se s'invoca la `exec()` immediatamente dopo la `fork()`, la duplicazione dei thread non è necessaria, poiché il programma specificato nei parametri della `exec()` sostituirà il processo. In questo caso conviene duplicare il solo thread chiamante. Tuttavia, se la `exec()` non segue immediatamente la `fork()`, il nuovo processo dovrebbe duplicare tutti i thread del processo genitore.

5.6.2 CANCELLAZIONE DEI THREAD

La cancellazione dei thread è l'operazione che permette di terminare un thread prima che completi il suo compito. per esempio, se più thread eseguono una ricerca in modo concorrente in una base di dati e un thread riporta il risultato, gli altri thread possono essere cancellati.

Una situazione analoga potrebbe verificarsi quando un utente preme il pulsante di terminazione di un browser Web per interrompere il caricamento di una pagina. Spesso il caricamento di una pagina è gestito da più thread: ogni immagine è carica da un thread separato; quando l'utente preme il pulsante di terminazione, tutti i thread che stanno caricando la pagina vengono cancellati.

Si presentano difficoltà con la cancellazione nei casi in cui ci siano risorse assegnate a un thread cancellato, o se si cancella un thread mentre sta aggiornando dei dati che condivide con altri thread.

5.6.3 GESTIONE DEI SEGNALI

Nei sistemi UNIX si usano i **segnali** per comunicare ai processi il verificarsi di determinati eventi. Un segnale si può ricevere in modo sincrono o asincrono, secondo la sorgente e la ragione della segnalazione dell'evento. Indipendentemente dal modo di ricezione sincrono o asincrono, tutti i segnali seguono lo stesso schema:

- all'occorrenza di un particolare evento si genera un segnale;
- s'invia il segnale a un processo;
- una volta ricevuto, il segnale deve essere gestito.

Come esempio, un accesso illegale alla memoria o una divisione per zero generano segnali sincroni. In questi casi, se un programma in esecuzione compie le suddette azioni, viene generato un segnale.

5.6.4 DATI SPECIFICI DEI THREAD

Uno dei vantaggi principali della programmazione multithread è dato dal fatto che i thread appartenenti allo stesso processo ne condividono i dati. Tuttavia, in particolari circostanze, ogni thread può necessitare di una copia privata di certi dati, chiamati **dati specifici dei thread** (o **TIS**).

È facile confondere i dati specifici dei thread con le variabili locali. mentre le variabili locali sono visibili solo durante la chiamata di una singola funzione, i dati specifici sono visibili attraverso tutte le chiamate. In un certo senso, i dati specifici assomigliano ai dati statici, con la differenza che i dati specifici sono unici per ogni thread.

5.7 GRUPPI DI THREAD

Nonostante la creazione di un thread distinto sia molto più vantaggiosa della creazione di un nuovo processo, tuttavia un server multithread presenta diversi problemi. Il primo riguarda il tempo richiesto per la creazione del thread prima di poter soddisfare la richiesta, considerando anche il fatto che questo thread sarà terminato non appena avrà completato il proprio lavoro. La seconda questione è più problematica: se si permette che tutte le richieste concorrenti siano servite da un nuovo thread, non si è posto un limite al numero di thread concorrentemente attivi nel sistema. Un numero illimitato di thread potrebbe esaurire le risorse del sistema, come il tempo di CPU o la memoria.

L'impiego dei **gruppi di thread** (*thread pool*) è una possibile soluzione a questo problema. L'idea generale è quella di creare un certo numero di thread alla creazione del processo, e organizzarli in un gruppo (pool) in cui attendano di eseguire il lavoro che gli sarà richiesto. Quando un server riceve una richiesta, attiva un thread del gruppo – se ce n'è uno disponibile – e gli passa la richiesta; dopo aver completato il suo lavoro, il thread rientra nel gruppo d'attesa. Se il gruppo non contiene alcun thread disponibile, il server attende fino a che un thread non si libera.

I vantaggi sono i seguenti:

- Il servizio di una richiesta tramite un thread esistente è più rapido dell'attesa della creazione di un nuovo thread;
- Un gruppo di thread limita il numero di thread esistenti a un certo istante; ciò è particolarmente rilevante per sistemi che non possono sostenere un elevato numero di thread concorrenti.
- Separare il task da svolgere dalla meccanica della sua creazione ci permette di utilizzare diverse strategie per l'esecuzione di tale task.

Il numero di thread di un gruppo si può determinare tramite euristiche che considerano fattori come il numero di CPU nel sistema, la quantità di memoria fisica e il numero atteso di richieste concorrenti da parte dei client.

6 – SCHEDULING DELLA CPU

6.1 CONCETTI FONDAMENTALI

In un sistema monoprocesso si può eseguire al massimo un processo alla volta; gli altri processi, se ci sono, devono attendere che la cpu sia libera e possa essere nuovamente sottoposta a scheduling. L'obiettivo della multiprogrammazione è avere sempre un processo in esecuzione, in modo da massimizzare l'utilizzazione della CPU.

L'idea è relativamente semplice. un processo è in esecuzione finché non deve attendere un evento, generalmente il completamento di qualche richiesta di I/O; durante l'attesa, in un sistema di calcolo semplice, la cpu resterebbe inattiva, e tutto il tempo d'attesa sarebbe sprecato. con la multiprogrammazione si cerca d'impiegare questi tempi d'attesa in modo produttivo: si tengono contemporaneamente più processi in memoria, e quando un processo deve attendere un evento, il sistema operativo gli sottrae il controllo della cpu per cederlo a un altro processo.

Lo scheduling è una funzione fondamentale dei sistemi operativi; si sottopongono a scheduling quasi tutte le risorse di un calcolatore. Naturalmente la cpu è una delle risorse principali, e il suo scheduling è alla base della progettazione dei sistemi operativi.

6.1.1 CICLICITÀ DELLE FASI DI ELABORAZIONE E DI I/O

L'esecuzione del processo consiste in un ciclo d'elaborazione (svolta dalla CPU) e d'attesa del completamento delle operazioni di I/O. I processi si alternano tra questi due stati.

L'esecuzione di un processo comincia con una sequenza di operazioni d'elaborazione svolte dalla CPU (*CPU burst*), seguita da una sequenza di operazioni di I/O (*I/O burst*), quindi un'altra sequenza di operazioni della cpu, di nuovo una sequenza di operazioni di I/O, e così via. L'ultima sequenza di operazioni della CPU si conclude con una richiesta al sistema di terminare l'esecuzione.

Quanto detto è ben schematizzato in questo schema:

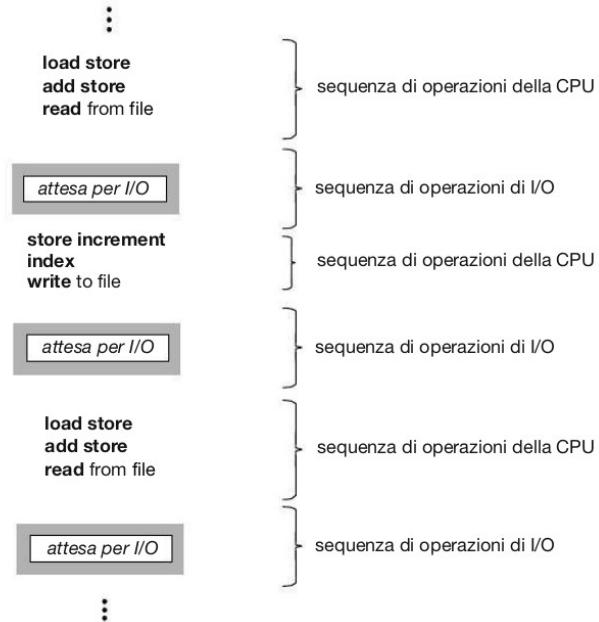


Figura 6.1 Serie alternata di sequenze di operazioni della CPU e di sequenze di operazioni di I/O.

6.1.2 SCHEDULER DELLA CPU

Ogniqualvolta la CPU passa nello stato d'inattività, il sistema operativo sceglie per l'esecuzione uno dei processi presenti nella ready queue. In particolare, è lo **scheduler a breve** termine, o scheduler della CPU che, tra i processi in memoria pronti per l'esecuzione, sceglie quello cui assegnare la CPU.

La ready queue non è necessariamente una coda in ordine d'arrivo (*first-in, first-out* o FIFO). Essa si può realizzare come una coda FIFO, una coda con priorità, un albero o semplicemente una lista concatenata non ordinata. Tuttavia, concettualmente tutti i processi della ready queue sono posti nella lista d'attesa per accedere alla CPU. Generalmente gli elementi delle code sono i *process control block* (PCB) dei processi.

6.1.3 SCHEDULING CON PRELAZIONE

Le decisioni riguardanti lo scheduling della CPU vengono prese nelle seguenti quattro circostanze:

- Un processo passa dallo stato di esecuzione allo stato di attesa (per esempio, richiesta di I/O o invocazione di `wait()` per la terminazione di uno dei processi figli);
- Un processo passa dallo stato di esecuzione allo stato pronto (per esempio, quando si verifica un segnale d'interruzione);
- Un processo passa dallo stato di attesa allo stato pronto (per esempio, al completamento di un'operazione di I/O);
- Un processo termina.

I casi 1 e 4 non danno alternative in termini di scheduling: si deve comunque scegliere un nuovo processo da eseguire, sempre che ce ne sia almeno uno nella ready queue per l'esecuzione. Una scelta si può invece fare nei casi 2 e 3.

Quando lo scheduling interviene solo nelle condizioni 1 e 4, si dice che lo schema di scheduling è **senza prelazione** (*nonpreemptive*) o **cooperativo** (*cooperative*); altrimenti, lo schema di scheduling è con **prelazione** (*preemptive*). Nel caso dello scheduling senza prelazione, quando si assegna la CPU a un processo, questo rimane in possesso della CPU fino al momento del suo rilascio, dovuto al termine dell'esecuzione o al passaggio nello stato di attesa.

6.1.4 DISPATCHER

Si tratta del modulo che passa effettivamente il controllo della cpu al processo scelto dallo scheduler a breve termine. Questa funzione comprende:

- il cambio di contesto;
- il passaggio alla modalità utente;
- il salto alla giusta posizione del programma utente per riaviarne l'esecuzione.

Poiché si attiva a ogni cambio di contesto, il dispatcher dovrebbe essere quanto più rapido è possibile. Il tempo richiesto dal dispatcher per fermare un processo e avviare l'esecuzione di un altro è noto come **latenza di dispatch**.

6.2 CRITERI DI SCHEDULING

Diversi algoritmi di scheduling della cpu hanno proprietà differenti e possono favorire una particolare classe di processi. Prima di scegliere l'algoritmo da usare in una specifica situazione occorre considerare le caratteristiche dei diversi algoritmi. Le caratteristiche usate per tale confronto possono incidere in modo rilevante sulla scelta dell'algoritmo migliore. Di seguito si riportano alcuni criteri:

- **Utilizzo della CPU**: La cpu deve essere più attiva possibile. Teoricamente, l'utilizzo della cpu può variare dallo 0 al 100 per cento. In un sistema reale dovrebbe variare dal 40 per cento, per un sistema con poco carico, al 90 per cento, per un sistema con utilizzo intenso.
- **Produttività**: La CPU è attiva quando si svolge del lavoro. Una misura del lavoro svolto è data dal numero dei processi completati nell'unità di tempo: tale misura è detta produttività (*throughput*).
- **Tempo di completamento**: Dal punto di vista di uno specifico processo, il criterio più importante è il tempo necessario per eseguire il processo stesso. L'intervallo che intercorre tra la sottomissione del processo e il completamento dell'esecuzione è chiamato tempo di completamento (*turnaround time*), ed è la somma dei tempi passati nell'attesa dell'ingresso in memoria, nella ready, durante l'esecuzione nella cpu e nelle operazioni di I/O.
- **Tempo d'attesa**: L'algoritmo di scheduling della cpu non influisce sul tempo impiegato per l'esecuzione di un processo o di un'operazione di I/O; influisce solo sul tempo d'attesa nella ready queue. Il tempo d'attesa è la somma degli intervalli d'attesa passati in questa coda.
- **Tempo di risposta**: Un'altra misura di confronto è data dal tempo che intercorre tra la effettuazione di una richiesta e la prima risposta prodotta. Questa misura è chiamata tempo di risposta, ed è data dal tempo necessario per iniziare la risposta, ma non dal suo tempo necessario per completare l'output.

6.3 ALGORITMI DI SCHEDULING

6.3.1 SCHEDULING IN ORDINE DI ARRIVO

Il più semplice algoritmo di scheduling della cpu è l'algoritmo di **scheduling in ordine d'arrivo** (**scheduling first-come, first-served** o **FCFS**). con questo schema la CPU si assegna al processo che la richiede per primo. La realizzazione del criterio FCFS si basa su una coda FIFO.

Quando un processo entra nella ready queue, si collega il suo PCB all'ultimo elemento della coda. Quando la CPU è libera, viene assegnata al processo che si trova alla testa della coda, rimuovendolo da essa.

Un aspetto negativo è che il tempo medio d'attesa per l'algoritmo FCFS è spesso abbastanza lungo. Si consideri il seguente insieme di processi, che si presenta al momento 0, con la durata della sequenza di operazioni della cpu espressa in millisecondi:

Processo	Durata della sequenza
P_1	24
P_2	3
P_3	3

Se i processi arrivano nell'ordine P_1, P_2, P_3 e sono serviti in ordine FCFS, si ottiene il risultato illustrato nel seguente **diagramma di Gantt**, un diagramma a barre che illustra una data pianificazione includendo i tempi d'inizio e fine di ogni processo partecipante.



Il tempo d'attesa è 0 millisecondi per il processo P_1 , 24 millisecondi per il processo P_2 e 27 millisecondi per il processo P_3 . Quindi, il tempo d'attesa medio è $(0 + 24 + 27)/3 = 17$ millisecondi. Se i processi arrivassero nell'ordine p_2, p_3, p_1 , i risultati sarebbero quelli illustrati nel seguente diagramma di Gantt:



Il tempo di attesa medio è ora di $(6 + 0 + 3)/3 = 3$ millisecondi. Si tratta di una notevole riduzione. Quindi, il tempo medio d'attesa in condizioni di FCFS non è in genere minimo, e può variare grandemente all'aumentare della variabilità dei CPU burst dei vari processi.

Si considerino inoltre le prestazioni dello scheduling FCFS in una situazione dinamica. Si supponga di avere un processo con prevalenza d'elaborazione e molti processi con prevalenza di I/O. Via via che i processi fluiscono nel sistema si può verificare la seguente situazione. Il processo con prevalenza d'elaborazione occupa la CPU.

Durante questo periodo tutti gli altri processi terminano le proprie operazioni di I/O e si spostano nella ready queue, nell'attesa della cpu. Mentre i processi si trovano nella ready queue, i dispositivi di I/O sono inattivi. Successivamente il processo con prevalenza d'elaborazione termina la propria sequenza di operazioni della cpu e passa a una fase di I/O. Tutti i processi con prevalenza di I/O, caratterizzati da sequenze di operazioni della cpu molto brevi, sono eseguiti rapidamente e tornano alle code di I/O, lasciando inattiva la CPU. Il processo con prevalenza d'elaborazione torna nella ready queue e riceve il controllo della CPU; così, finché non termina l'esecuzione del processo con prevalenza d'elaborazione, tutti i processi con prevalenza di I/O si trovano nuovamente ad attendere nella ready queue.

Si ha un **effetto convoglio**, tutti i processi attendono che un lungo processo liberi la CPU, il che causa una riduzione dell'utilizzo della CPU e dei dispositivi rispetto a quella che si avrebbe se si eseguissero per primi i processi più brevi.

L'algoritmo di scheduling FCFS è senza prelazione; una volta che la cpu è assegnata a un processo, questo la trattiene fino al momento del rilascio, che può essere dovuto al termine dell'esecuzione o alla richiesta di un'operazione di I/O.

L'algoritmo FCFS risulta particolarmente problematico nei sistemi in time sharing, dove è importante che ogni utente disponga della cpu a intervalli regolari. permettere a un solo processo di occupare la cpu per un lungo periodo condurrebbe a risultati disastrosi.

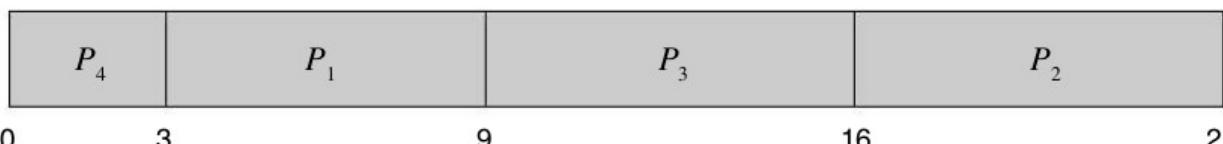
6.3.2 SCHEDULING SHORTEST-JOB-FIRST

Un criterio diverso di scheduling della cpu è l'algoritmo di **scheduling per brevità (shortest-job-first, SJF)**. Questo algoritmo associa a ogni processo la lunghezza della successiva sequenza di operazioni della CPU. Quando è disponibile, si assegna la CPU al processo che ha la più breve lunghezza della successiva sequenza di operazioni della CPU. Se due processi hanno le successive sequenze di operazioni della cpu della stessa lunghezza si applica lo scheduling FCFS.

Come esempio si consideri il seguente insieme di processi, con la durata della sequenza di operazioni della CPU espressa in millisecondi:

Processo	Durata della sequenza
P_1	6
P_2	8
P_3	7
P_4	3

Con lo scheduling SJF questi processi si ordinerebbero secondo il seguente diagramma di Gantt:



Il tempo d'attesa è di 3 millisecondi per il processo P1, di 16 millisecondi per il processo P2, di 9 millisecondi per il processo P3 e di 0 millisecondi per il processo P4. Quindi, il tempo d'attesa medio è di $(3 + 16 + 9 + 0)/4 = 7$ millisecondi.

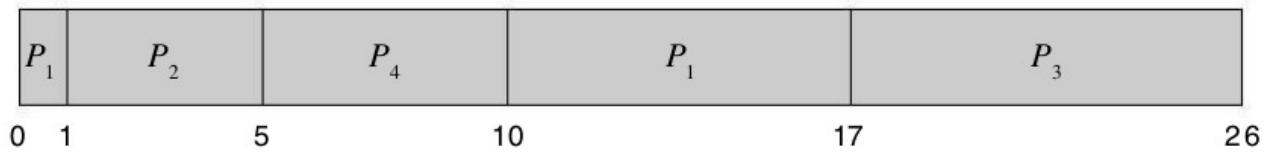
La difficoltà reale implicita nell'algoritmo SJF consiste nel conoscere la durata della successiva richiesta della CPU. Sebbene sia ottimale, l'algoritmo SJF non si può realizzare a livello dello scheduling della CPU a breve termine, poiché non esiste alcun modo per conoscere la lunghezza della successiva sequenza di operazioni della CPU. Un possibile approccio a questo problema consiste nel tentare di approssimare lo scheduling SJF: se non è possibile *conoscere* la lunghezza della prossima sequenza di operazioni della CPU, si può cercare di *predire* il suo valore.

L'algoritmo SJF può essere sia con prelazione sia senza prelazione. La scelta si presenta quando alla ready queue arriva un nuovo processo mentre un altro processo è ancora in esecuzione. Il nuovo processo può richiedere una sequenza di operazioni della CPU più breve di quella che resta al processo correntemente in esecuzione.

Come esempio, si considerino i quattro processi seguenti, dove la durata delle sequenze di operazioni della CPU è data in millisecondi:

Processo	Istante d'arrivo	Durata della sequenza
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Se i processi arrivano alla ready queue nei momenti indicati e richiedono i tempi di CPU illustrati, dallo scheduling SJF con prelazione risulta la sequenza indicata dal seguente diagramma di Gantt:



All'istante 0 si avvia il processo P1, poiché è l'unico che si trova nella coda. All'istante 1 arriva il processo P2. Il tempo necessario per completare il processo P1 (7 millisecondi) è maggiore del tempo richiesto dal processo P2 (4 millisecondi), perciò si effettua la prelazione del processo P1 sostituendolo col processo P2. Il tempo d'attesa medio per questo esempio è $((10 - 1) + (1 - 1) + (17 - 2) + (5 - 3))/4 = 26/4 = 6,5$ millisecondi. Con uno scheduling SJF senza prelazione si otterebbe un tempo d'attesa medio di 7,75 millisecondi.

6.3.3 SCHEDULING CON PRIORITÀ

L'algoritmo SJF è un caso particolare del più generale **algoritmo di scheduling con priorità**: si associa una priorità a ogni processo e si assegna la cpu al processo con priorità più alta; i processi con priorità uguali si ordinano secondo uno schema FCFS.

occorre notare che la discussione si svolge in termini di priorità alta e priorità bassa. Generalmente le priorità sono indicate da un intervallo fisso di numeri, come da 0 a 7, oppure da 0 a 4.095. Tuttavia, non c'è un orientamento comune sull'attribuire allo 0 la

priorità più alta o quella più bassa; alcuni sistemi usano numeri bassi per rappresentare priorità basse, altri usano numeri bassi per priorità alte, il che può generare confusione. In questo testo i numeri bassi indicano priorità alte.

Come esempio, si consideri il seguente insieme di processi, che si suppone siano arrivati al tempo 0, nell'ordine P1, P2 ... P5, e dove la durata delle sequenze di operazioni della CPU è espressa in millisecondi:

Processo	Durata della sequenza	Priorità
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Usando lo scheduling con priorità, questi processi sarebbero ordinati secondo il seguente diagramma di Gantt:



Il tempo d'attesa medio è di 8,2 millisecondi.

Le priorità si possono definire sia internamente sia esternamente. Quelle definite internamente usano una o più quantità misurabili per calcolare la priorità del processo; per esempio sono stati utilizzati i limiti di tempo, i requisiti di memoria, il numero dei file aperti e il rapporto tra la lunghezza media delle sequenze di operazioni di I/O e la lunghezza media delle sequenze di operazioni della CPU. Le priorità esterne si definiscono secondo criteri esterni al sistema operativo, come l'importanza del processo, il tipo e la quantità dei fondi pagati per l'uso del calcolatore, il dipartimento che promuove il lavoro e altri fattori, spesso di ordine politico.

Lo scheduling con priorità può essere sia con prelazione sia senza prelazione.

Un problema importante relativo agli algoritmi di scheduling con priorità è **l'attesa indefinita** (*starvation*). Un processo pronto per l'esecuzione, ma che non dispone della CPU, si può considerare bloccato nell'attesa della CPU. Un algoritmo di scheduling con priorità può lasciare processi a bassa priorità nell'attesa indefinita della CPU. In un sistema con carico elevato, un flusso costante di processi con priorità maggiore può impedire a un processo con bassa priorità di accedere alla CPU.

Una soluzione al problema dell'attesa indefinita dei processi con bassa priorità è costituita dall'**invecchiamento** (*aging*); si tratta di una tecnica di aumento graduale delle priorità dei processi che attendono nel sistema da parecchio tempo.

6.3.4 SCHEDULING CIRCOLARE

L'algoritmo di **scheduling circolare** (**round-robin**, RR) è stato progettato appositamente per i sistemi time sharing; è simile allo scheduling FcFS, ma aggiunge la capacità di prelazione in modo che il sistema possa commutare fra i vari processi.

Ciascun processo riceve una piccola quantità fissata del tempo della CPU, chiamata **quanto di tempo** o **porzione di tempo** (*time slice*), che varia generalmente da 10 a 100 millisecondi; la ready queue è trattata come una coda circolare. Lo scheduler della CPU scorre la ready queue, assegnando la cpu a ciascun processo per un intervallo di tempo della durata massima di un quanto di tempo.

Per implementare lo scheduling rr si gestisce la ready queue come una coda FIFO. I nuovi processi si aggiungono alla fine della ready queue. Lo scheduler della CPU prende il primo processo dalla ready queue, imposta un timer in modo che invii un segnale d'interruzione alla scadenza di un intervallo pari a un quanto di tempo, e attiva il dispatcher per eseguire il processo.

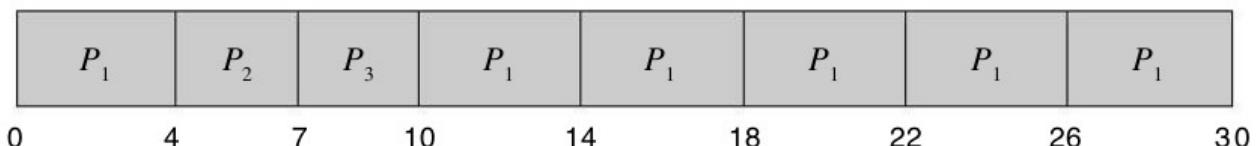
A questo punto si può verificare una delle due seguenti situazioni: il processo ha una sequenza di operazioni della CPU di durata minore di un quanto di tempo, quindi il processo stesso rilascia volontariamente la CPU e lo scheduler passa al processo successivo della ready queue; oppure la durata della sequenza di operazioni è più lunga di un quanto di tempo; in questo caso il timer scade e invia un segnale d'interruzione al sistema operativo, che esegue un cambio di contesto e mette il processo alla fine della ready queue. Lo scheduler quindi seleziona il processo successivo nella ready queue.

Il tempo d'attesa medio per il criterio di scheduling rr è spesso abbastanza lungo.

Si consideri il seguente insieme di processi che si presenta al tempo 0, con la durata delle sequenze di operazioni della CPU espressa in millisecondi:

Processo	Durata della sequenza
P_1	24
P_2	3
P_3	3

Se si usa un quanto di tempo di 4 millisecondi, il processo P_1 ottiene i primi 4 millisecondi ma, poiché richiede altri 20 millisecondi, è soggetto a prelazione dopo il primo quanto di tempo e la CPU passa al processo successivo della coda, il processo P_2 . Poiché il processo P_2 non necessita di 4 millisecondi, termina prima che il suo quanto di tempo si esaurisca, così si assegna immediatamente la cpu al processo successivo, il processo P_3 . Una volta che tutti i processi hanno ricevuto un quanto di tempo, si assegna nuovamente la CPU al processo P_1 per un ulteriore quanto di tempo. Dallo scheduling RR risulta quanto segue:



Calcoliamo ora il tempo di attesa medio per questa sequenza. P1 resta in attesa per 6 millisecondi (10-4), P2 per 4 millisecondi e P3 per 7 millisecondi. Il tempo d'attesa medio è di $17/3 = 5,66$ millisecondi.

Nell'algoritmo di scheduling RR la cpu si assegna a un processo per non più di un quanto di tempo per volta (a meno che questo sia l'unico processo ready). Se la durata della sequenza di operazioni della CPU di un processo eccede il quanto di tempo, il processo viene sottoposto a prelazione e riportato nella ready queue. L'algoritmo di scheduling RR è pertanto con prelazione.

Se nella ready queue esistono n processi e il quanto di tempo è pari a q, ciascun processo ottiene $1/n$ -esimo del tempo di elaborazione della cpu in frazioni di, al massimo, q unità di tempo. Ogni processo non deve attendere per più di $(n - 1) \times q$ unità di tempo per il suo successivo quanto di tempo.

Le prestazioni dell'algoritmo RR dipendono molto dalla dimensione del quanto di tempo. Nel caso limite in cui il quanto di tempo sia molto lungo, il criterio di scheduling RR si riduce al criterio di scheduling FCFS. Se il quanto di tempo è molto breve (per esempio, un millisecondo), il criterio RR può portare a un numero elevato di cambi di contesto. Il quanto di tempo deve essere ampio rispetto alla durata del cambio di contesto.

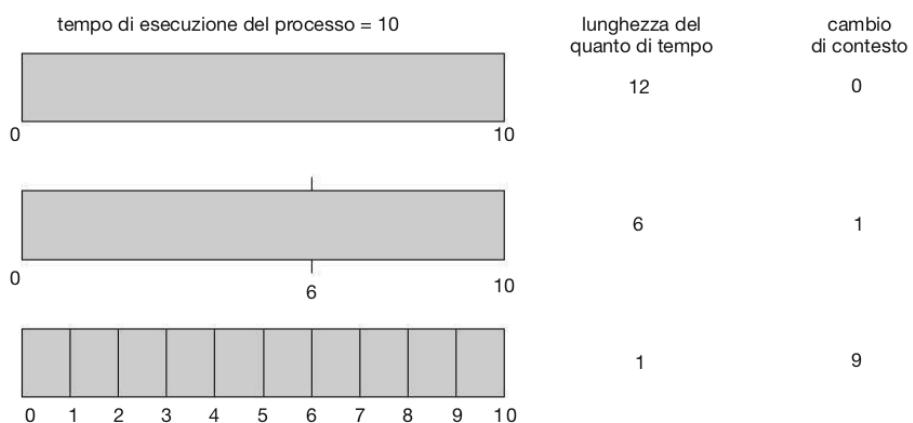


Figura 6.4 Aumento del numero dei cambi di contesto al diminuire del quanto di tempo.

6.3.5 SCHEDULING A CODE MULTIPLE

È stata creata una classe di algoritmi di scheduling adatta a situazioni in cui i processi si possono classificare facilmente in gruppi diversi. Una distinzione comune è per esempio quella che si fa tra i processi che si eseguono in **foreground** (primo piano), o **interattivi**, e i processi che si eseguono in **background** (sottofondo), o **batch** (a lotti).

Un **algoritmo di scheduling a code multilivello** (*multilevel queue scheduling algorithm*) suddivide la ready queue in code distinte.

I processi si assegnano in modo permanente a una coda, generalmente secondo qualche caratteristica del processo, come la quantità di memoria richiesta, la priorità o il tipo di processo. Ogni coda ha il proprio algoritmo di scheduling. Per esempio, per i processi in foreground e i processi in background si possono usare code distinte. La coda dei processi in foreground si può gestire con un algoritmo RR, mentre quella dei processi in background si può gestire con un algoritmo FCFS.

In questa situazione è inoltre necessario avere uno scheduling tra le code; si tratta comunemente di uno scheduling a priorità fissa con prelazione. Per esempio, la coda dei processi in foreground può avere la priorità assoluta sulla coda dei processi in background.

Un'altra possibilità è definire porzioni di tempo per le code. Ad ogni coda si assegna una parte del tempo d'elaborazione della CPU, suddivisibile a sua volta tra i processi che la costituiscono. Nel caso foreground/background, per esempio, si può assegnare l'80 per cento del tempo d'elaborazione della CPU alla coda dei processi in foreground, con scheduling RR tra i suoi processi; mentre per la coda dei processi in background si riserva il 20 per cento del tempo d'elaborazione della CPU, assegnato col criterio FCFS.

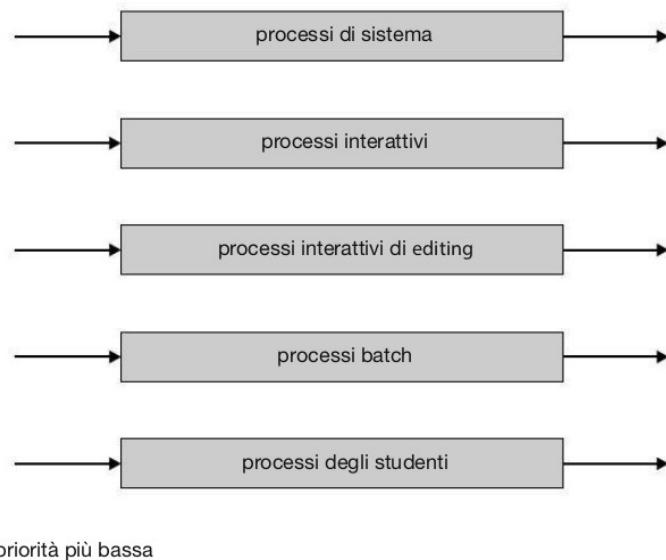


Figura 6.6 Scheduling a code multilivello.

6.3.6 SCHEDULING A CODE MULTILIVELLO CON RETROAZIONE

Di solito in un algoritmo di scheduling a code multilivello i processi si assegnano in modo permanente a una coda all'entrata nel sistema, e non si possono spostare tra le code. Se, per esempio, esistono code distinte per i processi che si eseguono in foreground e quelli che si eseguono in background, i processi non possono passare da una coda all'altra, poiché non possono cambiare la loro natura di processi in foreground o in background. Quest'impostazione è rigida, ma ha il vantaggio di avere un basso carico di scheduling.

Lo **scheduling a code multilivello con retroazione** (*multilevel feedback queue scheduling*), invece, permette ai processi di spostarsi fra le code. L'idea consiste nel separare i processi che hanno caratteristiche diverse in termini di CPU burst. Se un processo usa troppo tempo di elaborazione della cpu, viene spostato in una coda con priorità più bassa. Inoltre, si può spostare in una coda con priorità più elevata un processo che attende troppo a lungo in una coda a priorità bassa. Questa forma di aging impedisce il verificarsi di un'attesa indefinita.

Questo algoritmo di scheduling dà la massima priorità ai processi con una sequenza di operazioni della cpu della durata di non più di 8 millisecondi. I processi di questo tipo ottengono rapidamente la CPU, terminano la propria sequenza di operazioni della CPU e passano alla successiva sequenza di operazioni di I/O;

Generalmente uno scheduler a code multilivello con retroazione è caratterizzato dai seguenti parametri:

- Numero di code;
- Algoritmo di scheduling per ciascuna coda;
- Metodo usato per determinare quando spostare un processo in una coda con priorità maggiore;
- Metodo usato per determinare quando spostare un processo in una coda con priorità minore;
- Metodo usato per determinare in quale coda si deve mettere un processo quando richiede un servizio.

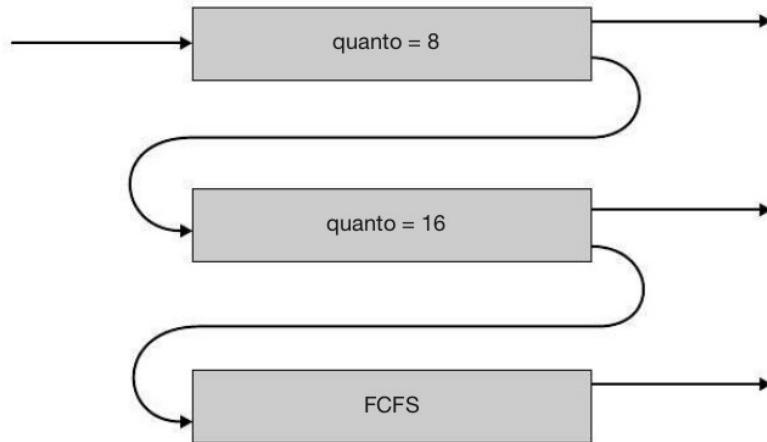


Figura 6.7 Code multilivello con retroazione.

6.4 SCHEDULING PER SISTEMI MULTIPROCESSORE

Fin qui la trattazione ha riguardato lo scheduling della cpu in un sistema a processore singolo. Se sono disponibili più unità d'elaborazione, è possibile la distribuzione del carico (*load sharing*), ma il problema dello scheduling diviene proporzionalmente più complesso.

Si considerano i sistemi in cui le unità d'elaborazione sono funzionalmente identiche (**sistemi omogenei**): si può usare qualunque unità d'elaborazione disponibile per eseguire qualsiasi processo presente nella coda. Anche i multiprocessori omogenei, talvolta, possono incorrere in limitazioni nello scheduling.

6.4.1 APPROCCI ALLO SCHEDULING PER MULTIPROCESSORI

Una prima strategia di scheduling della CPU per i sistemi multiprocessore affida tutte le decisioni, l'elaborazione dell' I/O e altre attività del sistema a un solo processore, il cosiddetto *master server*. Gli altri processori eseguono soltanto il codice utente. Si tratta della **multielaborazione asimmetrica**, che riduce la necessità di condividere dati grazie all'accesso di un solo processore alle strutture dati del sistema.

Quando invece ciascun processore provvede al proprio scheduling, si parla di **multielaborazione simmetrica** (*symmetric multiprocessing*, SMP). In questo caso i processi pronti per l'esecuzione sono situati tutti in una ready queue comune, oppure vi è un'apposita coda per ogni processore. In entrambi i casi, lo scheduler di ciascun processore esamina la coda appropriata, da cui seleziona un processo da eseguire.

6.5 SCHEDULING REAL-TIME DELLA CPU

Lo scheduling della CPU nei sistemi real-time ha peculiarità proprie. In generale, possiamo distinguere tra sistemi real-time soft e sistemi real-time hard.

I **sistemi real-time soft** non offrono garanzie sul momento in cui un processo critico sarà eseguito, ma assicurano solamente che sarà data precedenza a quest'ultimo piuttosto che ad altri processi non critici.

I **sistemi real-time hard** hanno vincoli più rigidi: i task vanno eseguiti entro una scadenza prefissata ed eseguirli dopo tale scadenza è del tutto inutile.

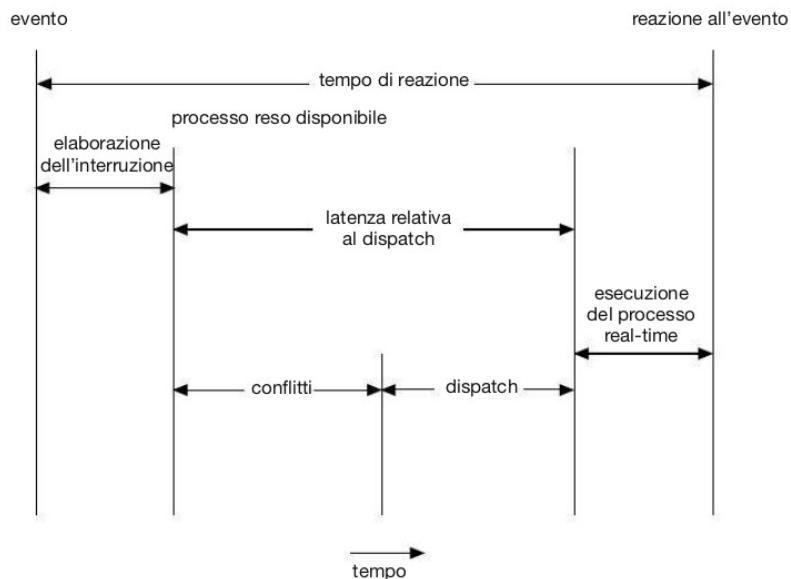


Figura 6.14 Latenza relativa al dispatch.

6.6 VALUTAZIONE DEGLI ALGORITMI

Ci si può chiedere come scegliere un algoritmo di scheduling della cpu per un sistema particolare. Il primo problema da affrontare riguarda la definizione dei criteri da usare per la scelta dell'algoritmo.

Come abbiamo visto, i criteri spesso si definiscono spesso in termini di utilizzo della CPU, tempo di risposta o produttività. per scegliere un algoritmo occorre innanzitutto stabilire l'importanza relativa di queste misure. Tra i criteri suggeriti si possono inserire diverse misure, per esempio le seguenti:

- Rendere massimo l'utilizzo della cpu con il vincolo che il massimo tempo di risposta sia 1 secondo;
- Rendere massima la produttività in modo che il tempo di completamento sia (in media) linearmente proporzionale al tempo d'esecuzione totale.

Una volta definiti i criteri di selezione, è necessario valutare gli algoritmi considerati. Di seguito si descrivono i vari possibili metodi di valutazione.

- **Modellazione deterministica**: un tipo di valutazione analitica, che considera un carico di lavoro predeterminato e definisce le prestazioni di ciascun algoritmo per quel carico di lavoro.
- **Reti di code**: se sono noti le distribuzioni degli arrivi e dei servizi si possono calcolare l'utilizzo, la lunghezza media delle code, il tempo medio di attesa, ecc.

- **Simulazione e realizzazione:** quando si fa la valutazione, conviene parametrare un sistema e farlo girare. Durante l'esecuzione si registra tutto ciò che avviene. In questo modo si ottiene un cosiddetto **trace tape**. Si tratta di uno strumento eccezionale che permette di confrontare gli algoritmi rispetto allo stesso insieme di dati reali, e con cui si possono ottenere risultati molto precisi.

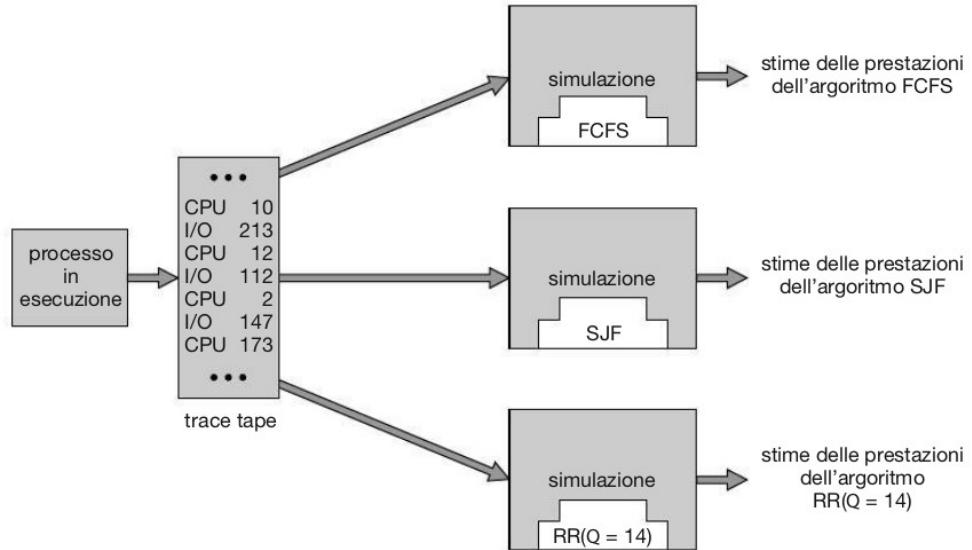


Figura 6.25 Valutazione di algoritmi di scheduling della CPU tramite una simulazione.

7 – SINCRONIZZAZIONE DEI PROCESSI

7.1 INTRODUZIONE

Nei sistemi operativi multi-programmati diversi processi o thread sono in esecuzione asincrona e possono condividere dati.

I processi cooperanti possono:

- Condividere uno spazio logico di indirizzi, cioè codice e dati (Thread);
- Oppure solo dati.

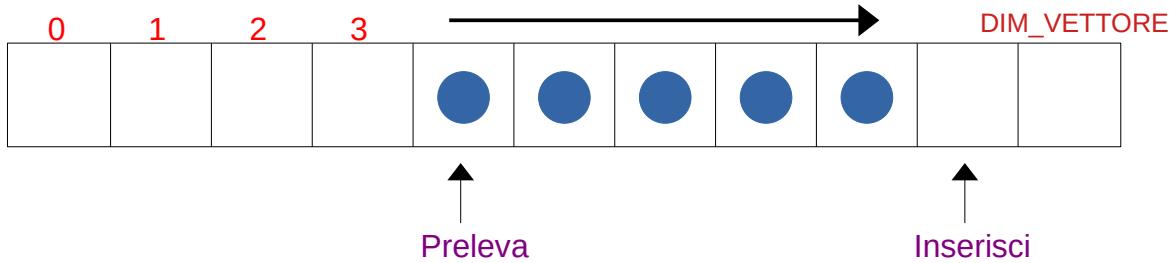
L'accesso concorrente a dati condivisi, se non si adottano politiche di sincronizzazione, può causare incoerenza degli stessi dati. Mantenere la coerenza dei dati richiede meccanismi atti ad assicurare un'ordinata esecuzione dei processi cooperanti.

7.2 ESEMPIO: MEMORIA LIMITATA – SOLUZIONE CON MEMORIA CONDIVISA

Supponiamo di usare una zona di memoria condivisa ed un vettore circolare di grandezza `DIM_VETTORE` e che fa uso di 2 indici per accedere al vettore. Con l'indice `inserisci` puntiamo alla prima casella disponibile del vettore. Con l'indice `preleva` puntiamo alla prima posizione occupata del vettore.

Da tali definizioni si ricava che:

- Se `inserisci == preleva`, allora il vettore è vuoto;
- Se $((inserisci+1) \% \text{DIM_VETTORE}) == \text{preleva}$, allora il vettore è pieno;
- MAX elemento del vettore = `DIM_VETTORE - 1`;



Per come è stato strutturato questo tipo di vettore, non è possibile riempirlo al massimo della capienza, perché rimarrà sempre una locazione vuota. La soluzione del problema dei produttori e dei consumatori con memoria limitata contente la presenza contemporanea del vettore di n-1 elementi.

Si supponga di voler modificare il codice del produttore-consumatore aggiungendo una variabile intera `counter` inizializzata a 0 e che si incrementa ogniqualvolta s'inserisce un nuovo elemento nel buffer e si decrementa ogniqualvolta si preleva un elemento dal buffer.

Andiamo ora ad implementare un po' di codice della seguente situazione:

```
#define BUFFER_SIZE 10

typedef struct {
    ...
} item;

item buffer[BUFFER_SIZE];
int in = 0;
int out = 0;
int counter = 0;
```

Nel caso del processo produttore:

```
item nextProduced;

while (1){
    while (counter == BUFFER_SIZE)
        //do nothing

    buffer[in] = nextProduced;
    in = (in + 1) % BUFFER_SIZE;
    counter++;
}
```

Nel caso del processo consumatore invece:

```
item nextConsumed;

while (1){
    while (counter == 0)
        //do nothing

    nextConsumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    counter--;
}
```

Sebbene siano corrette se si considerano separatamente, le procedure del produttore e del consumatore possono non funzionare altrettanto correttamente se si eseguono in modo concorrente. Si supponga per esempio che il valore della variabile contatore sia attualmente 5, e che i processi produttore e consumatore eseguano le istruzioni `counter++` e `counter--` in modo concorrente. Terminata l'esecuzione delle due istruzioni, il valore della variabile contatore potrebbe essere 4, 5 o 6! Il solo risultato corretto è `counter == 5`, che si ottiene se si eseguono separatamente il produttore e il consumatore.

Le istruzioni `counter++` e `counter--` devono essere eseguite in modo atomico: significa che queste istruzioni non devono essere interrotte durante la loro esecuzione, quindi se si eseguono correttamente senza interruzioni.

Si può dimostrare che il valore di contatore può essere scorretto: l'istruzione `conunter++` si può codificare in un tipico linguaggio macchina, come:

```
register1 = counter
register1 = register1 + 1
counter = register1
```

dove `register1` è un registro locale della CPU. Analogamente, l'istruzione `counter--` si può codificare come:

```
register2 = counter
register2 = register2 - 1
counter = register2
```

dove `register2` è un registro locale della CPU.

L'esecuzione concorrente delle istruzioni `counter++` e `counter--` equivale a un'esecuzione sequenziale delle istruzioni del linguaggio macchina introdotte precedentemente, **intercalate** (*interleaved*) in una qualunque sequenza che però conservi l'ordine interno di ogni singola istruzione di alto livello. Una di queste sequenze è:

T_0 :	<i>produttore</i>	esegue	$registro_1 := \text{contatore}$	{ $registro_1 = 5$ }
T_1 :	<i>produttore</i>	esegue	$registro_1 := registro_1 + 1$	{ $registro_1 = 6$ }
T_2 :	<i>consumatore</i>	esegue	$registro_2 := \text{contatore}$	{ $registro_2 = 5$ }
T_3 :	<i>consumatore</i>	esegue	$registro_2 := registro_2 - 1$	{ $registro_2 = 4$ }
T_4 :	<i>produttore</i>	esegue	$\text{contatore} := registro_1$	{ $\text{contatore} = 6$ }
T_5 :	<i>consumatore</i>	esegue	$\text{contatore} := registro_2$	{ $\text{contatore} = 4$ }

e conduce al risultato errato in cui `counter == 4`; si registra la presenza di 4 elementi nel buffer, mentre in realtà gli elementi sono 5. Se si invertisse l'ordine delle istruzioni in T4 e T5 si giungerebbe allo stato errato in cui `counter == 6`.

Si è arrivati a questo stato non corretto perché si è permesso a entrambi i processi di manipolare concorrentemente la variabile `counter`. Per evitare le situazioni di questo tipo, in cui più processi accedono e modificano gli stessi dati in modo concorrente e i risultati dipendono dall'ordine degli accessi (le cosiddette **race condition**) occorre assicurare che un solo processo alla volta possa modificare la variabile `counter`. Questa garanzia richiede una forma di **sincronizzazione dei processi**.

7.3 PROBLEMA DELLA SEZIONE CRITICA

Si consideri un sistema composto di n processi $\{P_0, P_1, \dots, P_{n-1}\}$ ciascuno avente un segmento di codice, chiamato **sezione critica** (detto anche *regione critica*), in cui il processo può modificare variabili comuni, aggiornare una tabella, scrivere in un file e così via. La caratteristica fondamentale del sistema è che, quando un processo è in esecuzione nella propria sezione critica, non si consente a nessun altro processo di essere in esecuzione nella propria sezione critica.

Il problema della *sezione critica* consiste nel progettare un protocollo che i processi possono usare per cooperare. Ogni processo deve chiedere il permesso per entrare nella propria sezione critica. La sezione di codice che realizza questa richiesta è la **sezione d'ingresso**. La sezione critica può essere seguita da una **sezione d'uscita**, e la restante parte del codice è detta **sezione non critica**.

La seguente figura mostra la struttura generale di un tipico processo P_i . La sezione d'ingresso e quella d'uscita sono state inserite nei riquadri per evidenziare questi importanti segmenti di codice.

```
do {  
    sezione d'ingresso  
    sezione critica  
    sezione d'uscita  
    sezione non critica  
} while (true);
```

Figura 5.1 Struttura generale di un tipico processo P_i .

Una soluzione del problema della sezione critica deve soddisfare i tre seguenti requisiti:

- **Mutua esclusione**: Se il processo P_i è in esecuzione nella sua sezione critica, nessun altro processo può essere in esecuzione nella propria sezione critica.
- **Progresso**: Se nessun processo è in esecuzione nella sua sezione critica e qualche processo desidera entrare nella propria sezione critica, solo i processi che si trovano fuori delle rispettive sezioni non critiche possono partecipare alla decisione riguardante la scelta del processo che può entrare per primo nella propria sezione critica; questa scelta non si può rimandare indefinitamente.
- **Attesa limitata**: Se un processo ha già richiesto l'ingresso nella sua sezione critica, esiste un limite al numero di volte che si consente ad altri processi di entrare nelle rispettive sezioni critiche prima che si accordi la richiesta del primo processo.

7.4 ALGORITMI

7.4.1 ALGORITMO 1

Come variabile condivisa abbiamo `turno`, inizialmente a 0. L'algoritmo è il seguente:

```
do {  
    while (turno != i);  
        sezione critica  
    turno = j;  
        sezione non critica  
} while (1);
```

Questo algoritmo soddisfa la mutua esclusione (perché può entrare un processo alla volta), ma non il requisito di progresso perché diamo per scontato che subito dopo P_0 (che è sicuramente il primo processo perché abbiamo inizializzato `turno` a 0 all'inizio) ci sia P_1 , ma questo non ce lo garantisce nessuno.

7.4.2 ALGORITMO 2

La variabile condivisa è `pronto[2]`, inizialmente a `false`. L'algoritmo è il seguente:

```
do {  
    pronto[i] = true;  
    while (pronto[j]);  
        sezione critica  
    pronto[i] = false;  
        sezione non critica  
} while (1);
```

Anche questo algoritmo soddisfa la mutua esclusione, ma non il requisito di progresso perché i due processi possono essere pronti a entrare nella sezione critica e quindi non si potrà stabilire quale dei due processi dovrà entrare nella propria sezione critica.

7.4.3 ALGORITMO 3

Questo algoritmo mette insieme gli algoritmo 1 e 2. Di conseguenza le variabili condivise saranno la combinazione dei due algoritmi. Il codice è il seguente:

```
do {  
    pronto[i] = true;  
    turno = j;  
    while (pronto[j] && turno == j);  
  
    sezione critica  
  
    pronto[i] = false;  
  
    sezione non critica  
} while (1);
```

Soddisfa tutti i tre requisiti e va a risolvere il problema della sezione critica per i due processi.

7.4.5 ALGORITMO DEL FORNAIO

L'algoritmo precedente, per quanto sia semplice e soddisfi tutti e tre i requisiti, in realtà non è molto efficiente perché può essere applicato solo per 2 processi. Vogliamo ora andare a definire un algoritmo che ci permetta di risolvere il problema della sezione critica per n processi.

Prima di entrare nella sua sezione critica, il processo riceve un numero. Chi detiene il numero più basso entra nella sezione critica.

Se i processi P_i e P_j ricevono lo stesso numero, se $i < j$, allora P_i è servito per primo; altrimenti viene servito P_j . Lo schema di numerazione genera sempre numeri in ordine crescente (es. 1,2,3,3,3,4,5, ...). Il codice è il seguente:

```
do {  
  
    scelta[i] = true;  
    numero[i] = max(numero[0], ... , numero[n-1]) + 1;  
    scelta[i] = false;  
  
    for (j = 0; j < n; j++) {  
        while (scelta[j]);  
        while (numero[j] != 0 &&  
               (numero[j] < numero[i] ||  
                (numero[j] == numero[i] && j < i))  
    }  
  
    sezione critica  
  
    numero[i] = 0;  
  
    sezione non critica  
} while (1);
```

A causa del modo in cui i moderni elaboratori eseguono le istruzioni elementari del linguaggio macchina, quali `load` e `store`, non è affatto certo che l'algoritmo del fornaio funzioni correttamente su tali sistemi. Tuttavia si è scelto di presentarla ugualmente perché rappresenta un buon algoritmo per il problema della sezione critica che illustra alcune complessità legate alla progettazione di programmi che soddisfino i tre requisiti di mutua esclusione, progresso e attesa limitata.

7.5 ARCHITETTURE DI SINCRONIZZAZIONE

Molte delle moderne architetture offrono particolari istruzioni che permettono di controllare e modificare il contenuto di una parola di memoria, oppure di scambiare il contenuto di due parole di memoria, in modo atomico – cioè come un'unità non interrompibile. Queste speciali istruzioni sono utilizzabili per risolvere il problema della sezione critica in modo relativamente semplice.

Vediamo prima l'istruzione `test_and_set()`:

```
boolean TestAndSet (boolean &obiettivo) {  
    boolean valore = obiettivo;  
    obiettivo = true;  
    return valore;  
}
```

La caratteristica fondamentale di questa istruzione è che viene eseguita atomicamente, quindi se si eseguono contemporaneamente due istruzioni `test_and_set()`, ciascuna in un'unità d'elaborazione diversa, queste vengono eseguite in modo sequenziale in un ordine arbitrario.

Se si dispone dell'istruzione `test_and_set()`, si può realizzare la mutua esclusione dichiarando una variabile booleana globale `blocco`, inizializzata a `false`. La struttura del processo P_i è il seguente:

```
do {  
    while (TestAndSet (blocco)) ;  
  
    sezione critica  
  
    blocco = false;  
  
    sezione non critica  
} while (1);
```

L'altra istruzione è `swap()`. Essa agisce sul contenuto di due parole di memoria; come per `test_and_set()`, anch'essa è eseguita in maniera atomica. La sua struttura è la seguente:

```
void Swap(boolean &a, boolean &b) {  
    boolean temp = a;  
    a = b;  
    b = temp;  
}
```

Se si dispone dell'istruzione `swap()`, si può realizzare la mutua esclusione dichiarando una variabile booleana `blocco` e un vettore booleano `waiting[n]`, entrambi settati a `false`.

La struttura del processo P_i è il seguente:

```
do {  
    chiave = true;  
    while (chiave == true)  
        Swap(blocco, chiave);  
  
    sezione critica  
  
    blocco = false;  
  
    sezione non critica  
} while (1);
```

7.6 SEMAFORI

Un **semaforo** S è un strumento di sincronizzazione che non richiede attesa attiva. È una variabile intera cui si può accedere, escludendo l'inizializzazione, solo tramite due operazioni atomiche predefinite: `wait()` e `signal()`.

La definizione di `wait()` in pseudocodice è la seguente:

```
wait(S) {  
    while (S <= 0)  
        //attesa attiva (non fa operazioni)  
    S--;  
}
```

La definizione di `signal()` in pseudocodice è la seguente:

```
signal(S) {  
    S++;  
}
```

Tutte le modifiche al valore del semaforo contenute nelle operazioni `wait()` e `signal()` si devono eseguire in modo indivisibile: mentre un processo cambia il valore del semaforo, nessun altro processo può contemporaneamente modificare quello stesso valore.

7.6.1 SEZIONE CRITICA DI N PROCESSI

Vediamo come si può immaginare di realizzare una sezione critica con n processi impiegando un semaforo. Come dato condiviso abbiamo `semaphore mutex` (inizialmente `mutex = 1`). La struttura del processo P_i è il seguente:

```
do {  
    wait(mutex);  
  
    sezione critica  
  
    signal (mutex);  
  
    sezione non critica  
} while (1);
```

7.6.2 IMPLEMENTAZIONE DEI SEMAFORI

Per superare la necessità dell'attesa attiva si possono modificare le definizioni delle operazioni `wait()` e `signal()` come segue: quando un processo invoca l'operazione `wait()` e trova che il valore del semaforo non è positivo, deve attendere, ma anziché restare nell'attesa attiva può *bloccare* se stesso. L'operazione di bloccaggio pone il processo in una coda d'attesa associata al semaforo e pone lo stato del processo a *waiting*. Quindi, si trasferisce il controllo allo scheduler della CPU che sceglie un altro processo pronto per l'esecuzione.

Un processo bloccato, che attende a un semaforo `S`, sarà riavviato in seguito all'esecuzione di un'operazione `signal()` su `S` da parte di qualche altro processo. Il processo si riavvia tramite un'operazione `wakeup()`, che modifica lo stato del processo da *waiting* a *ready*. Il processo entra nella coda dei processi pronti.

Per realizzare i semafori secondo quel che s'è detto si può definire il semaforo come segue:

```
typedef struct {
    int valore;
    struct processo *L;
} semaforo;
```

A ogni semaforo sono associati un valore intero (`valore`) e una lista di processi (`L`), contenente i processi in attesa a un semaforo; l'operazione `signal()` preleva un processo da tale lista e lo attiva.

L'operazione `wait()` del semaforo si può definire come segue:

```
wait(S) {
    S.valore--;
    if (S.valore < 0)
        aggiungi questo processo a S.L
        block;
}
```

L'operazione `signal()` del semaforo si può definire come segue:

```
signal(S) {
    S.valore++;
    if (S.valore <= 0)
        togli un processo P da S.L
        wakeup(P);
}
```

L'operazione `block()` sospende il processo che la invoca; l'operazione `wakeup(P)` pone in stato di pronto per l'esecuzione un processo `P` bloccato. Queste due operazioni sono fornite dal sistema operativo come chiamate di sistema.

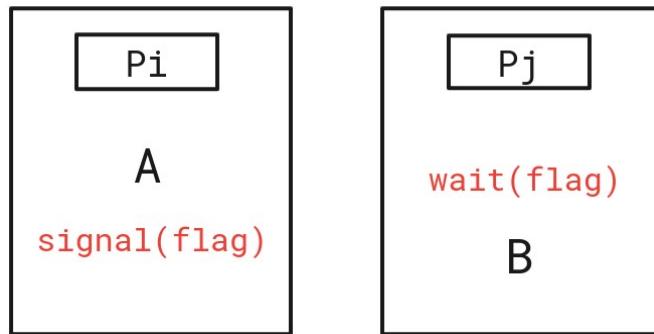
Occorre notare che, mentre la definizione classica di semaforo ad attesa attiva è tale che il valore del semaforo non è mai negativo, questa definizione può condurre a valori negativi. Se il valore del semaforo è negativo, il suo valore assoluto rappresenta il numero dei processi che attendono a quel semaforo.

Ciò è dovuto all'inversione dell'ordine del decremento e della verifica nel codice dell'operazione `wait()`.

7.6.3 SEMAFORI COME STRUMENTI GENERALI DI SINCRONIZZAZIONE

Possiamo anche ipotizzare l'uso dei semafori per un sincronismo molto più generale. Come sappiamo, nel nostro sistema, ci sono tanti processi è molto complicato stabilire quale processo mandare in esecuzione prima e quale dopo. Ci sono però situazioni in cui abbiamo bisogno di sapere quale processo eseguire prima.

Facciamo un esempio: ipotizziamo di avere due processi A e B. Vogliamo che venga eseguito prima A. Per questo scopo usiamo un semaforo `flag` inizializzato a 0. Abbiamo questa situazione:



Iniziamo quindi prima da P_i : viene eseguito il codice A e poi si effettua la `signal(flag)` che incrementerà la variabile `flag` (passa da 0 a 1). A questo punto, se deve andare in esecuzione P_j , questo effettua prima la `wait(flag)`. Ma `flag` vale 1 e la condizione della funzione `wait()` non è rispettata; il processo va avanti e `flag` viene decrementata per poi, infine, eseguire B. Possiamo dunque dire che, avendo inizializzato `flag` a 0, prima viene eseguito A e poi B.

Se volessimo invece fare il caso opposto (far partire prima P_j), non si presenterà la stessa situazione, in quanto la condizione della `wait()` è verificata (`flag` è inizializzata a 0) e quindi rimarrà in loop.

7.6.4 STALLO DEI PROCESSI E ATTESA INDEFINITA

La realizzazione di un semaforo con coda d'attesa può condurre a situazioni in cui due o più processi attendono indefinitamente un evento – l'esecuzione di un'operazione `signal()` – che può essere generato solo da uno dei processi in attesa. Quando si verifica una situazione di questo tipo si dice che i processi sono in **stallo** (*deadlocked*). Per illustrare questo fenomeno si considerino due processi, P0 e P1, ciascuno dei quali ha accesso a due semafori, S e Q, impostati al valore 1:

P_0	P_1
<code>wait(S);</code>	<code>wait(Q);</code>
<code>wait(Q);</code>	<code>wait(S);</code>
⋮	⋮
⋮	⋮
<code>signal(S);</code>	<code>signal(Q);</code>
<code>signal(Q);</code>	<code>signal(S);</code>

Si supponga che P_0 esegua `wait(S)` e quindi P_1 esegua `wait(Q)`; eseguita `wait(Q)`, P_0 deve attendere che P_1 esegua `signal(Q)`; analogamente, quando P_1 esegue `wait(S)`, deve attendere che P_0 esegua `signal(S)`. Poiché queste operazioni `signal()` non si possono eseguire, P_0 e P_1 sono in stallo.

Un insieme di processi è in stallo se ciascun processo dell'insieme attende un evento che può essere causato solo da un altro processo dell'insieme.

Importante notare che se viene eseguito prima tutto P_0 e poi tutto P_1 , la situazione di stallo non si presenta.

Un'altra questione connessa alle situazioni di stallo è quella **dell'attesa indefinita** (nota anche col termine **starvation**). Con tale termine si definisce una situazione d'attesa indefinita nella coda di un semaforo, che si può per esempio presentare se i processi si aggiungono e si rimuovono dalla lista associata a un semaforo secondo un criterio LIFO (last-in, first-out).

7.6.5 TIPI DI SEMAFORI

Si usa distinguere i semafori in due categorie:

- **Semafori contatore**: Il loro valore è un numero intero. Trovano applicazione nel controllo dell'accesso a una data risorsa presente in un numero finito di esemplari. Essendo un valore non binario, è difficile da maneggiare.
- **Semafori binari**: Il loro valore è limitato a 0 o 1. Proprio per questo è più semplice da realizzare.

È possibile comunque implementare un semaforo contatore in termini di un semaforo binario.

7.6.6 REALIZZAZIONE DI S COME SEMAFORO BINARIO

Se abbiamo un semaforo S e vogliamo esprimere mediante semafori binari abbiamo bisogno:

- Strutture dati:
 - semaforo_binario $S1$ e $S2$;
 - int C ;
- Inizializzazione:
 - $S1 = 1$;
 - $S2 = 0$;
 - $C = \text{valore iniziale del semaforo contatore } S$.

Abbiamo bisogno ora di definire le funzioni di `wait()` e `signal()`. La `wait()` sarà definita in questo modo:

```
wait(S1);
C--;
if (C < 0) {
    signal(S1);
    wait(S2);
}
signal(S1);
```

L'operazione `signal()` sarà definita in questo modo:

```
wait(S1);
C++;
if (C <= 0) {
    signal(S2);
} else {
    signal(S1);
}
```

7.7 PROBLEMI TIPICI DI SINCRONIZZAZIONE

7.7.1 PRODUTTORE/CONSUMATORE CON MEMORIA LIMITATA

Sappiamo che i produttori e i consumatori lavorano in maniera concorrente. Il problema nasce dal fatto che il buffer su cui lavorano ha capienza limitata. Quindi bisogna adottare una logica di sincronismo per controllare quando si può produrre e quando si può consumare. Vediamo di trovare un modo per risolvere questo problema.

Si supponga di disporre di una certa quantità di memoria rappresentata da un buffer con n posizioni, ciascuna capace di contenere un elemento. Il semaforo `mutex` garantisce la mutua esclusione degli accessi al buffer ed è inizializzato al valore 1. I semafori `vuote` e `piene` conteggiano rispettivamente il numero di posizioni vuote e il numero di posizioni piene nel buffer. Il semaforo `vuote` si inizializza al valore n ; il semaforo `piene` si inizializza al valore 0. Quindi avremo:

```
int n;
semaforo mutex = 1;
semaforo vuote = n;
semaforo piene = 0;
```

La struttura generale del processo produttore sarà:

```
do {
    ...
    //produce un elemento in appena_prodotto
    ...
    wait(empty);
    wait(mutex);

    ...
    //inserisci appena_prodotto in buffer
    ...
    signal(mutex);
    signal(piene);
} while (1);
```

La struttura generale del processo consumatore sarà invece:

```
do {
    wait(piene);
    wait(mutex);

    ...
    //rimuovi un elemento da buffer e mettilo in da_consumare
    ...
    signal(mutex);
    signal(vuote);

    ...
    //consuma l'elemento contenuto in da_consumare
    ...
} while (1);
```

7.7.2 PROBLEMA DEI LETTORI-SCRITTORI

Si supponga che una base di dati sia da condividere tra numerosi processi concorrenti. Alcuni processi possono richiedere solo la lettura del contenuto della base dati, mentre altri ne possono richiedere un aggiornamento, vale a dire una lettura e una scrittura. Questi due tipi di processi vengono distinti chiamando **lettori** quelli interessati alla sola lettura e **scrittori** gli altri.

Naturalmente, se due lettori accedono nello stesso momento all'insieme di dati condiviso, non si ha alcun effetto negativo; viceversa, se uno scrittore e un altro processo (lettore o scrittore) accedono contemporaneamente alla stessa base di dati, ne può derivare il caos.

Per impedire l'insorgere di difficoltà di questo tipo è necessario che gli scrittori abbiano un accesso esclusivo in fase di scrittura alla base di dati condivisa. Questo problema di sincronizzazione è conosciuto come **problema dei lettori-scrittori**.

Il problema dei lettori-scrittori ha diverse varianti, che implicano tutte l'esistenza di priorità; la più semplice, cui si fa riferimento come al primo problema dei lettori-scrittori, richiede che nessun lettore attenda, a meno che uno scrittore abbia già ottenuto il permesso di usare l'insieme di dati condiviso. In altre parole, nessun lettore deve attendere che altri lettori terminino l'operazione solo perché uno scrittore attende l'accesso ai dati. Il secondo problema dei lettori-scrittori richiede che uno scrittore, una volta pronto, esegua il proprio compito di scrittura al più presto. In altre parole, se uno scrittore attende l'accesso all'insieme di dati, nessun nuovo lettore deve iniziare la lettura.

La soluzione del primo problema dei lettori-scrittori prevede la condivisione da parte dei processi lettori delle seguenti strutture dati:

```
semaforo scrittori = 1;
semaforo mutex = 1;
int numLettori = 0;
```

Il semaforo `mutex` si usa per assicurare la mutua esclusione al momento d'aggiornamento di `numLettori`. Quest'ultima contiene il numero dei processi che stanno attualmente leggendo l'insieme di dati. Il semaforo `scrittori` funziona come semaforo di mutua esclusione per gli scrittori e serve anche al primo o all'ultimo lettore che entra o esce dalla sezione critica. Non serve, invece, ai lettori che entrano o escono mentre altri lettori si trovano nelle rispettive sezioni critiche.

Vediamo la struttura generale di un processo scrittore:

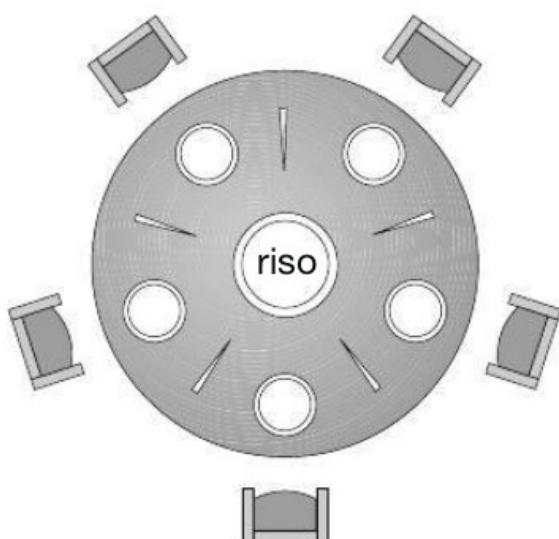
```
do {  
    wait (scrittori);  
    ...  
    //esegui l'operazione di scrittura  
    ...  
    signal (scrittori);  
} while (1);
```

Vediamo ora la struttura generale di un processo lettore:

```
do {  
    wait (mutex);  
    numLettori++;  
    if (numLettori == 1)  
        wait (scrittori);  
  
    signal (mutex);  
    ...  
    //esegui l'operazione di lettura  
    ...  
    wait (mutex);  
    numLettori--;  
    if (numLettori == 0)  
        signal (scrittori);  
  
    signal (mutex);  
} while (1);
```

7.7.3 PROBLEMA DEI CINQUE FILOSOFI

Si considerino cinque filosofi che trascorrono la loro esistenza pensando e mangiando. I filosofi condividono un tavolo rotondo circondato da cinque sedie, una per ciascun filosofo. Al centro del tavolo si trova una zuppiera colma di riso, e la tavola è apparecchiata con cinque bacchette.



Quando un filosofo pensa, non interagisce con i colleghi; quando gli viene fame, tenta di prendere le bacchette più vicine: quelle che si trovano tra lui e i commensali alla sua destra e alla sua sinistra. Un filosofo può prendere una bacchetta alla volta e non può prendere una bacchetta che si trova già nelle mani di un suo vicino. Quando un filosofo affamato tiene in mano due bacchette contemporaneamente, mangia senza lasciare le bacchette. Terminato il pasto, le posa e riprende a pensare.

Una semplice soluzione consiste nel rappresentare ogni bacchetta con un semaforo: un filosofo tenta di afferrare ciascuna bacchetta eseguendo un'operazione `wait()` su quel semaforo e la posa eseguendo operazioni `signal()` sui semafori appropriati. Quindi i dati condivisi sono:

```
semaforo bacchetta[5];
```

dove tutti gli elementi di `bacchetta` sono inizializzati a 1. La struttura del filosofo i è la seguente:

```
do {
    wait(bacchetta[i]);
    wait(bacchetta[i+1] % 5);

    ...
    //mangia

    ...
    signal(bacchetta[i]);
    signal(bacchetta[i+1] % 5);

    ...
    //pensa
    ...
} while (1);
```

Questa soluzione garantisce che due vicini non mangino contemporaneamente, ma è insufficiente poiché non esclude la possibilità che si abbia una situazione di stallo. Si supponga che tutti e cinque i filosofi abbiano fame contemporaneamente e che ciascuno afferri la bacchetta di sinistra; tutti gli elementi di `chopstick` diventano uguali a zero, perciò ogni filosofo che tenta di afferrare la bacchetta di destra entra in stallo. Tali situazioni di stallo possono essere evitate con i seguenti espedienti:

- solo quattro filosofi possono stare contemporaneamente a tavola;
- un filosofo può prendere le sue bacchette solo se sono entrambe disponibili (quest'operazione si deve eseguire in una sezione critica);
- si adotta una soluzione asimmetrica: un filosofo dispari prende prima la bacchetta di sinistra e poi quella di destra, invece un filosofo pari prende prima la bacchetta di destra e poi quella di sinistra.

7.8 REGIONI CRITICHE

È un costrutto di sincronizzazione di alto livello. Sono fondati su una variabile condivisa `v` di tipo `T`, dichiarata in questo modo:

v : shared T

Alla variabile `v` si può accedere solo dall'interno di un'istruzione:

region v when B do S

dove `B` è un'espressione booleana.

Mentre si esegue l'istruzione `S` nessun altro processo può accedere alla variabile.

Regioni che si riferiscono alla stessa variabile condivisa si escludono vicendevolmente.

Quando un processo vuole accedere alla variabile condivisa *v* nella regione critica, si valuta l'espressione booleana *B*: se risulta vera, si esegue l'istruzione *S*, altrimenti il processo rilascia la mutua esclusione ed è ritardato fino a quando *B* diventa vera e nessun altro processo si trova nella regione associata a *v*.

Vediamo un esempio (problema dei produttori e consumatori con memoria limitata):
Dati condivisi:

```
struct vettore {  
    elemento gruppo[n];  
    int contatore, inserisci, preleva;  
}
```

Il processo produttore inserisce appena_prodotto nel vettore condiviso:

```
region vettore when (contatore < n) {  
    gruppo[inserisci] = appena_prodotto;  
    inserisci := (inserisci + 1) % n;  
    contatore++;  
}
```

Il processo consumatore invece rimuove un elemento dal vettore condiviso e lo inserisce in da_consumare:

```
region vettore when (contatore > 0) {  
    da_consumare = gruppo[preleva];  
    preleva = (preleva + 1) % n;  
    contatore--;  
}
```

7.9 MONITOR

È un costrutto di sincronizzazione di alto livello, che consente la condivisione sicura di un tipo di dato astratto fra processi concorrenti. La sua struttura la seguente:

```
monitor nome_monitor {  
    //dichiarazione di variabili condivise  
  
    procedure body P1 (...) {  
        ...  
    }  
  
    procedure body P2 (...) {  
        ...  
    }  
  
    procedure body Pn (...) {  
        ...  
    }  
  
    codice d'inizializzazione(...)  
}
```

Per consentire a un processo di attendere all'interno di un monitor occorre definire una variabile condizionale come:

condition x,y;

Le uniche operazioni eseguibili su una variabile **condition** sono `wait` e `signal`:

- L'operazione `x.wait()`; implica che il processo che la invoca rimanga sospeso fino a che un altro processo non invoca l'operazione `x.signal()`.
- L'operazione `x.signal()` risveglia esattamente un processo sospeso. Se non esistono processi sospesi, l'operazione `signal` non ha alcun effetto.

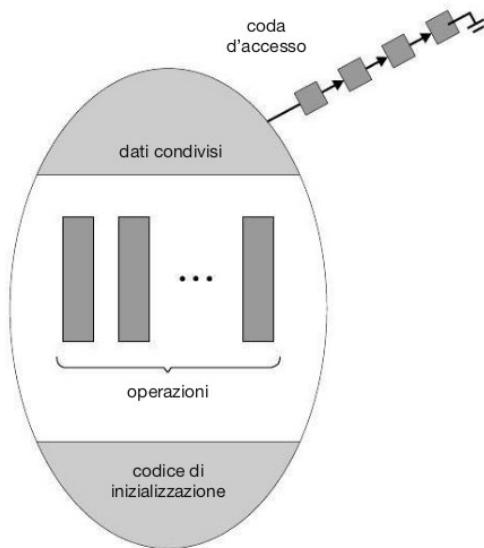


Figura 5.16 Schema di un monitor.

7.10 SINCRONIZZAZIONE IN LINUX

Nelle prime versioni di Linux non era gestita la prelazione. La tecnica di sincronizzazione più semplice nel kernel di Linux era l'intero atomico rappresentato mediante il tipo di dato opaco `atomic_t`.

Le operazioni matematiche che usano interi atomici vengono eseguite senza interruzioni.

Altre tecniche più sofisticate disponibili nel kernel:

- `mutex_lock()`: per entrare nella sezione critica;
- `mutex_unlock()`: in uscita dalla sezione critica.

7.11 SINCRONIZZAZIONE IN WINDOWS

Quando il nucleo di tale sistema operativo accede a una risorsa globale in una sistema a singola CPU, disabilita temporaneamente le interruzioni avendo procedure di gestione che potrebbero accedere alla stessa risorsa globale. In un sistema con più unità di elaborazione, protegge l'accesso alle risorse globali con i semafori ad attesa attiva (*spinlocks*).

Per la sincronizzazione fuori dal nucleo, il sistema operativo offre gli **oggetti dispatcher** che permettono ai thread di sincronizzarsi servendosi di diversi macchinari, inclusi mutex, semafori ed eventi. Gli eventi sono un meccanismo di sincronizzazione che si può usare in modo simile alle variabili condizionali.

8 – STALLO DEI PROCESSI

In un ambiente con multiprogrammazione più processi possono competere per ottenere un numero finito di risorse; se una risorsa non è correntemente disponibile, il processo richiedente passa allo stato d'attesa. In alcuni casi, se le risorse richieste sono trattenute da altri processi, a loro volta nello stato d'attesa, il processo potrebbe non cambiare più il suo stato. Situazioni di questo tipo sono chiamate di **stallo** (*deadlock*).

Un efficace esempio di situazione di stallo si può ricavare da una legge dello stato del Kansas approvata all'inizio del ventesimo secolo, che in una sua parte recita: "Quando due treni convergono a un incrocio, ambedue devono arrestarsi, e nessuno dei due può ripartire prima che l'altro si sia allontanato". (non serve a niente sto coso, ma al prof piace quindi mettiamolo)

Altri esempi possono essere:

- Il sistema dispone di due unità a nastri. P_1 e P_2 possiedono ciascuno una di queste unità e ciascuno richiede un'altra unità a nastri;
- Semafori A e B inizializzati a 1, con la seguente condizione:

P_1	P_2
wait (A) ;	wait (B) ;
wait (B) ;	wait (A) ;

8.1 MODELLO DEL SISTEMA

Un sistema è composto da un numero finito di risorse da distribuire tra più processi in competizione. Le risorse possono essere suddivise in tipi (o classi) differenti, ciascuno formato da un certo numero di istanze identiche. Cicli di CPU, file e dispositivi di I/O (come stampanti e lettori DVD), sono tutti esempi di tipi di risorsa. Se un sistema ha due CPU, il tipo di risorsa CPU ha due istanze. Analogamente, il tipo di risorsa stampante può avere cinque istanze.

Prima di adoperare una risorsa, un sistema deve richiederla e, dopo averla usata, deve rilasciarla. Un processo può richiedere tutte le risorse necessarie per eseguire il compito assegnatogli, anche se ovviamente il numero delle risorse richieste non può superare quello totale delle risorse disponibili nel sistema: un processo non può richiedere tre stampanti se il sistema ne ha solo due.

Nelle ordinarie condizioni di funzionamento un processo può servirsi di una risorsa soltanto se rispetta la seguente sequenza:

- **Richiesta**: Il processo richiede la risorsa; se la richiesta non si può soddisfare immediatamente – per esempio, perché la risorsa è attualmente in possesso di un altro processo – il processo richiedente deve attendere finché non possa acquisire tale risorsa;
- **Uso**: Il processo può operare sulla risorsa (se, per esempio, la risorsa è una stampante, il processo può effettuare una stampa);
- **Rilascio**: Il processo rilascia la risorsa.

8.2 CARATTERIZZAZIONE DELLE SITUAZIONI DI STALLO

In una situazione di stallo i processi non terminano mai l'esecuzione, e le risorse del sistema vengono bloccate impedendo l'esecuzione di altri processi. Prima di trattarne le soluzioni, descriviamo più precisamente le caratteristiche di una situazione di stallo.

8.2.1 CONDIZIONI NECESSARIE

Si può avere una situazione di stallo solo se in un sistema si verificano contemporaneamente le seguenti quattro condizioni:

- **Mutua esclusione**: almeno una risorsa deve essere non condivisibile, vale a dire che è utilizzabile da un solo processo alla volta. Se un altro processo richiede tale risorsa, si deve ritardare il processo richiedente fino al rilascio della risorsa.
- **Possesso e attesa**: Un processo deve essere in possesso di almeno una risorsa e attendere di acquisire risorse già in possesso di altri processi.
- **Assenza di prelazione**: Le risorse non possono essere prelazionate, vale a dire che una risorsa può essere rilasciata dal processo che la possiede solo volontariamente, dopo aver terminato il proprio compito.
- **Attesa circolare**: Deve esistere un insieme $\{P_0, P_1, \dots, P_n\}$ di processi, tale che P_0 attende una risorsa posseduta da P_1 , P_1 attende una risorsa posseduta da P_2 , ..., P_{n-1} attende una risorsa posseduta da P_n e P_n attende una risorsa posseduta da P_0 .

Occorre sottolineare che tutte e quattro le condizioni devono essere vere, altrimenti non si può avere alcuno stallo. La condizione dell'attesa circolare implica la condizione di possesso e attesa, quindi le quattro condizioni non sono completamente indipendenti; tuttavia è utile considerare separatamente ciascuna condizione.

8.2.2 GRAFO DI ASSEGNAZIONE DELLE RISORSE

Le situazioni di stallo si possono descrivere con maggior precisione avvalendosi di una rappresentazione detta **grafo di assegnazione delle risorse**. Si tratta di un insieme di vertici V e un insieme di archi E , con l'insieme di vertici V composto da due sottoinsiemi: $P = \{P_1, P_2, \dots, P_n\}$, che rappresenta tutti i processi del sistema, e $R = \{R_1, R_2, \dots, R_m\}$, che rappresenta tutti i tipi di risorsa del sistema.

Un arco diretto dal processo P_i al tipo di risorsa R_j si indica $P_i \rightarrow R_j$, e significa che il processo P_i ha richiesto un'istanza del tipo di risorsa R_j , e attualmente attende tale risorsa. Un arco diretto dal tipo di risorsa R_j al processo P_i si indica $R_j \rightarrow P_i$, e significa che un'istanza del tipo di risorsa R_j è assegnata al processo P_i . Un arco orientato $P_i \rightarrow R_j$ si chiama **arco di richiesta**, un arco orientato $R_j \rightarrow P_i$ si chiama **arco di assegnazione**.

Graficamente ogni processo P_i si rappresenta con un cerchio e ogni tipo di risorsa R_j si rappresenta con un rettangolo. Giacché il tipo di risorsa R_j può avere più di un'istanza, ciascuna di loro si rappresenta con un puntino all'interno del rettangolo.

Quando il processo P_i richiede un'istanza del tipo di risorsa R_j , si inserisce un arco di richiesta nel grafo di assegnazione delle risorse. Se questa richiesta può essere esaudita, si trasforma immediatamente l'arco di richiesta in un arco di assegnazione, che al rilascio della risorsa viene cancellato.

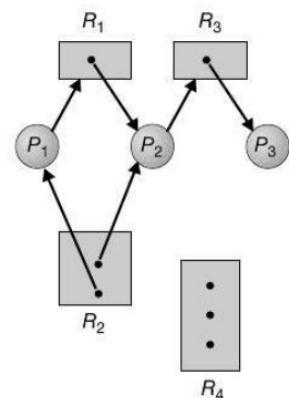


Figura 7.1 Grafo di assegnazione delle risorse.

Nel grafo di assegnazione delle risorse della figura precedente è illustrata la seguente situazione:

- Insiemi P, R ed E:
 - $P = \{P_1, P_2, P_3\}$
 - $R = \{R_1, R_2, R_3, R_4\}$
 - $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_1, R_2 \rightarrow P_2, R_3 \rightarrow P_3\}$
- Istanze delle risorse:
 - Un'istanza del tipo di risorsa R_1
 - Due istanze del tipo di risorsa R_2
 - Un'istanza del tipo di risorsa R_3
 - Tre istanze del tipo di risorsa R_4
- Stati dei processi:
 - Il processo P_1 possiede un'istanza del tipo di risorsa R_2 e attende un'istanza del tipo di risorsa R_1
 - Il processo P_2 possiede un'istanza dei tipi di risorsa R_1 ed R_2 e attende un'istanza del tipo di risorsa R_3
 - Il processo P_3 possiede un'istanza del tipo di risorsa R_3

Data la definizione di grafo di assegnazione delle risorse, si può mostrare che, se il grafo non contiene cicli, nessun processo del sistema subisce uno stallo; se il grafo contiene un ciclo, può sopraggiungere uno stallo.

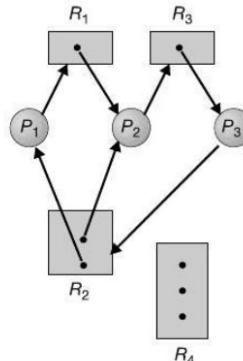


Figura 7.2 Grafo di assegnazione delle risorse con uno stallo.

Il fatto che ci sia un ciclo all'interno del grafo di assegnazione, non implica che ci sia necessariamente uno stallo. Ad esempio:

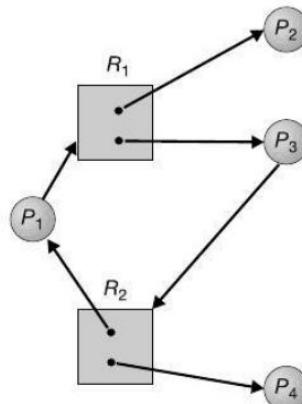


Figura 7.3 Grafo di assegnazione delle risorse con un ciclo, ma senza stallo.

Anche in questo caso c'è un ciclo $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$, ma non c'è uno stallo, in quanto il processo P_4 può rilasciare la propria istanza del tipo di risorsa R_2 , che si può assegnare al processo P_3 , rompendo il ciclo.

8.3 METODI PER LA GESTIONE DELLA SITUAZIONE DI STALLO

Essenzialmente, il problema delle situazioni di stallo si può affrontare in tre modi:

- Si può usare un protocollo per prevenire o evitare le situazioni di stallo, assicurando che il sistema non entri mai in stallo;
- Si può permettere al sistema di entrare in stallo, individuarlo, e quindi eseguire il ripristino;
- Si può ignorare del tutto il problema, fingendo che le situazioni di stallo non possano mai verificarsi nel sistema.

Quest'ultima è la soluzione usata dalla maggior parte dei sistemi operativi, compresi Linux e Windows.

Per assicurare che non si verifichi mai uno stallo, il sistema può servirsi di metodi di prevenzione o di metodi per evitare tale situazione. **Prevenire le situazioni di stallo** significa far uso di metodi atti ad assicurare che non si verifichi almeno una delle condizioni necessarie.

Per **evitare le situazioni di stallo** occorre che il sistema operativo abbia in anticipo informazioni aggiuntive riguardanti le risorse che un processo richiederà e userà durante le sue attività. Con queste informazioni aggiuntive il sistema operativo può decidere se una richiesta di risorse da parte di un processo si può soddisfare o se il processo debba invece attendere. In tale processo di decisione il sistema tiene conto delle risorse correntemente disponibili, di quelle correntemente assegnate a ciascun processo, e delle future richieste e futuri rilasci di ciascun processo.

8.4 PREVENIRE LE SITUAZIONI DI STALLO

Affinché si abbia uno stallo si devono verificare quattro condizioni necessarie; perciò si può prevenire il verificarsi di uno stallo assicurando che almeno una di queste condizioni non possa capitare.

8.4.1 MUTUA ESCLUSIONE

Deve valere la condizione di mutua esclusione: almeno una risorsa deve essere non condivisibile. Le risorse condivisibili, invece, non richiedono l'accesso mutuamente esclusivo, perciò non possono essere coinvolte in uno stallo. I file aperti per la sola lettura sono un buon esempio di risorsa condivisibile; se più processi richiedono l'apertura di un file a sola lettura, possono ottenere un accesso contemporaneo.

Un processo non deve mai attendere una risorsa condivisibile. Ma poiché alcune risorse sono intrinsecamente non condivisibili, non si possono prevenire in generale le situazioni di stallo negando la condizione di mutua esclusione.

8.4.2 POSSESSO E ATTESA

Per assicurare che la condizione di possesso e attesa non si presenti mai nel sistema, occorre garantire che un processo che richiede una risorsa non ne possegga altre. Si può usare un protocollo che ponga la condizione che ogni processo, prima di iniziare la propria esecuzione, richieda tutte le risorse che gli servono e che esse gli siano assegnate.

Questa condizione si può realizzare imponendo che le chiamate di sistema che richiedono risorse per un processo precedano tutte le altre.

Un protocollo alternativo è quello che permette a un processo di richiedere risorse solo se non ne possiede: un processo può richiedere risorse e adoperarle, ma prima di richiedere ulteriori risorse deve rilasciare tutte quelle che possiede.

Entrambi i protocolli presentano due svantaggi principali. Innanzitutto, l'utilizzo delle risorse può risultare poco efficiente, poiché molte risorse possono essere assegnate, ma non utilizzate, per un lungo periodo di tempo. Il secondo svantaggio è dovuto al fatto che si possono verificare situazioni di attesa indefinita. Un processo che richieda più risorse molto utilizzate può trovarsi nella condizione di attendere indefinitamente la disponibilità, poiché almeno una delle risorse di cui necessita è sempre assegnata a qualche altro processo.

8.4.3 ASSENZA DI PRELAZIONE

La terza condizione necessaria prevede che non sia possibile avere la prelazione su risorse già assegnate. Per assicurare che questa condizione non sussista, si può impiegare il seguente protocollo. Se un processo che possiede una o più risorse ne richiede un'altra che non gli si può assegnare immediatamente (e quindi il processo deve attendere), allora si esercita la prelazione su tutte le risorse attualmente in suo possesso. Si ha cioè il rilascio implicito di queste risorse, che si aggiungono alla lista delle risorse che il processo sta attendendo; il processo viene nuovamente avviato solo quando può ottenere sia le vecchie risorse sia quella nuova che sta richiedendo.

In alternativa, quando un processo richiede alcune risorse, se ne verifica la disponibilità: se sono disponibili vengono assegnate, se non lo sono, si verifica se sono assegnate a un processo che attende altre risorse. In tal caso si sottraggono le risorse desiderate a quest'ultimo processo e si assegnano al processo richiedente. Se le risorse non sono disponibili né sono possedute da un processo in attesa, il processo richiedente deve attendere. Durante l'attesa si può avere la prelazione di alcune sue risorse; ciò può accadere solo se un altro processo le richiede. Un processo si può avviare nuovamente solo quando riceve le risorse che sta richiedendo e recupera tutte quelle a esso sottratte durante l'attesa.

8.4.4 ATTESA CIRCOLARE

La quarta e ultima condizione necessaria per una situazione di stallo è l'attesa circolare. Un modo per assicurare che tale condizione d'attesa non si verifichi consiste nell'imporre un ordinamento totale all'insieme di tutti i tipi di risorse e imporre che ciascun processo richieda le risorse in ordine crescente.

Si supponga che $R = \{R_1, R_2, \dots, R_m\}$ sia l'insieme dei tipi di risorse. A ogni tipo di risorsa si assegna un numero intero unico che permetta di confrontare due risorse e stabilirne la relazione di precedenza nell'ordinamento. Formalmente, si definisce una funzione iniettiva, $F: R \rightarrow N$, dove N è l'insieme dei numeri naturali. Se, per esempio, l'insieme dei tipi di risorsa R contiene unità a nastri, unità disco e stampanti, la funzione F si può definire come segue:

$$\begin{aligned}F(\text{unità a nastri}) &= 1 \\F(\text{unità a disco}) &= 5 \\F(\text{stampante}) &= 12\end{aligned}$$

Per prevenire il verificarsi di situazioni di stallo si può considerare il seguente protocollo: ogni processo può richiedere risorse solo seguendo un ordine crescente di numerazione.

Ciò significa che un processo può richiedere inizialmente qualsiasi numero di istanze di un tipo di risorsa, per esempio R_i , dopo di che il processo può richiedere istanze del tipo di risorsa R_j se e solo se $F(R_j) > F(R_i)$. Si noti inoltre che, se sono necessarie più istanze di uno stesso tipo di risorsa, queste istanze devono essere oggetto di **un'unica richiesta**.

8.5 EVITARE LE SITUAZIONI DI STALLO

Un metodo alternativo per evitare le situazioni di stallo consiste nel richiedere ulteriori informazioni sulle modalità di richiesta delle risorse. In un sistema con un'unità a nastri e una stampante, per esempio, il sistema potrebbe aver bisogno di sapere che il processo p intende richiedere prima l'unità a nastri e poi la stampante, prima di rilasciarle entrambe, mentre il processo Q richiederà prima la stampante e poi l'unità a nastri. Una volta acquisita la sequenza completa delle richieste e dei rilasci di ogni processo, il sistema può stabilire per ogni richiesta se il processo debba attendere o meno, per evitare una possibile situazione di stallo futura.

Il modello più semplice e più utile richiede che ciascun processo dichiari il *numero massimo* delle risorse di ciascun tipo di cui necessita. Data questa informazione a priori, si può costruire un algoritmo capace di assicurare che il sistema non entri mai in stallo. Questo algoritmo per evitare lo stallo esamina dinamicamente lo stato di assegnazione delle risorse per garantire che non possa esistere una condizione di attesa circolare. Lo stato di assegnazione delle risorse è definito dal numero di risorse disponibili e assegnate e dalle richieste massime dei processi.

8.5.1 STATO SICURO

Uno stato si dice *sicuro* se il sistema è in grado di assegnare risorse a ciascun processo (fino al suo massimo) in un certo ordine e impedire il verificarsi di uno stallo. Più formalmente, un sistema si trova in stato sicuro solo se esiste una **sequenza sicura**.

Una sequenza di processi $\langle P_1, P_2, \dots, P_n \rangle$ è una sequenza sicura per lo stato di assegnazione attuale se, per ogni P_i , le richieste che pi può ancora fare si possono soddisfare impiegando le risorse attualmente disponibili più le risorse possedute da tutti i P_j con $j < i$. In questa situazione, se le risorse necessarie al processo P_i non sono disponibili immediatamente, allora pi può attendere che tutti i P_j abbiano finito, e a quel punto pi può ottenere tutte le risorse di cui ha bisogno, completare il compito assegnato, restituire le risorse assegnate e terminare. Quando pi termina, P_{i+1} può ottenere le risorse richieste, e così via. Se non esiste una sequenza di questo tipo, lo stato del sistema si dice *non sicuro*.

Uno stato sicuro non è di stallo. viceversa, uno stato di stallo è uno stato non sicuro; tuttavia non tutti gli stati non sicuri sono stati di stallo. Uno stato non sicuro può condurre a uno stallo.

8.5.2 ALGORITMO CON GRAFO DI ASSEGNAZIONE DELLE RISORSE

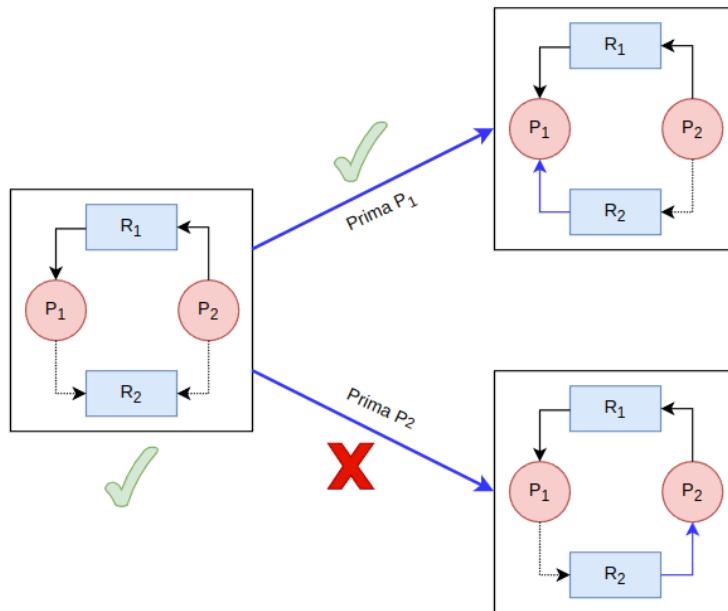
Quando il sistema per l'assegnazione delle risorse è tale che ogni tipo di risorsa ha una sola istanza, per evitare le situazioni di stallo si può far uso di una variante del grafo di assegnazione delle risorse. Oltre agli archi di richiesta e di assegnazione, si introduce un nuovo tipo di arco, l'arco di "rivendicazione" (*claim edge*).

Un **arco di rivendicazione** $P_i \rightarrow R_j$ indica che il processo P_i può richiedere la risorsa R_j in un qualsiasi momento futuro. Quest'arco ha la stessa direzione dell'arco di richiesta, ma si rappresenta con una linea tratteggiata.

Quando il processo P_i richiede la risorsa R_j , l'arco di rivendicazione $P_i \rightarrow R_j$ diventa un arco di richiesta. Analogamente, quando P_i rilascia la risorsa R_j , l'arco di assegnazione $R_j \rightarrow P_i$ diventa un arco di rivendicazione $P_i \rightarrow R_j$.

Occorre sottolineare che le risorse devono essere rivendicate a priori nel sistema. Ciò significa che prima che il processo P_i inizi l'esecuzione, tutti i suoi archi di rivendicazione devono essere già inseriti nel grafo di assegnazione delle risorse.

Vediamo ora, partendo da una certa situazione, cosa può succedere applicando quanto detto sopra:



Leggendo il primo grafico abbiamo:

- $P_2 \rightarrow R_1, R_1 \rightarrow P_1$
- Sia P_1 sia P_2 potrebbero chiedere una risorsa a R_2 .
- Non è presente circolarità.

Con i dati in nostro possesso, non è possibile stabilire chi far partire prima. Se assegnamo prima la risorsa R_2 a P_1 , notiamo che la non circolarità persiste. Diverso invece il discorso se assegnamo R_2 prima a P_2 .

8.5.3 ALGORITMO DEL BANCHIERE (da leggere, non lo chiede nei dettagli)

L'algoritmo con grafo di assegnazione delle risorse non si può applicare ai sistemi di assegnazione delle risorse con più istanze di ciascun tipo di risorsa. L'algoritmo per evitare le situazioni di stallo descritto nel seguito, pur essendo meno efficiente dello schema con grafo di assegnazione delle risorse, si può applicare a tali sistemi, ed è noto col nome di **algoritmo del banchiere**.

Quando si presenta al sistema, un nuovo processo deve dichiarare il numero massimo delle istanze di ciascun tipo di risorsa di cui potrà aver bisogno. Questo numero non può superare il numero totale delle risorse del sistema. Quando un utente richiede un gruppo di risorse, si deve stabilire se l'assegnazione di queste risorse lasci il sistema in uno stato

sicuro. Se si rispetta tale condizione, si assegnano le risorse, altrimenti il processo deve attendere che qualche altro processo ne rilasci un numero sufficiente. Quando un processo ottiene tutte le sue risorse, deve restituirle entro un intervallo di tempo definito.

La realizzazione dell'algoritmo del banchiere richiede la gestione di alcune strutture dati che codificano lo stato di assegnazione delle risorse del sistema. Sia n il numero di processi del sistema e m il numero dei tipi di risorsa. Sono necessarie le seguenti strutture dati:

- **Disponibili:** Un vettore di lunghezza m che indica il numero delle istanze disponibili per ciascun tipo di risorsa; $\text{Disponibili}[j] = k$, significa che sono disponibili k istanze del tipo di risorsa R_j .
- **Massimo:** Una matrice $n \times m$ che definisce la richiesta massima di ciascun processo; $\text{Massimo}[i, j] = k$ significa che il processo P_i può richiedere un massimo di k istanze del tipo di risorsa R_j .
- **Assegnate:** Una matrice $n \times m$ che definisce il numero delle istanze di ciascun tipo di risorsa attualmente assegnate a ogni processo; $\text{Assegnate}[i, j] = k$ significa che al processo P_i sono correntemente assegnate k istanze del tipo di risorsa R_j .
- **Necessità:** Una matrice $n \times m$ che indica la necessità residua di risorse relativa a ogni processo; $\text{Necessità}[i, j] = k$ significa che il processo P_i , per completare il suo compito, può avere bisogno di altre k istanze del tipo di risorsa R_j . Si osservi che $\text{Necessità}[i, j] = \text{Massimo}[i, j] - \text{Assegnate}[i, j]$.

8.5.3.1 ALGORITMO DI VERIFICA DELLA SICUREZZA

L'algoritmo utilizzato per scoprire se il sistema è o non è in uno stato sicuro si può descrivere come segue:

- Siano Lavoro e Fine vettori di lunghezza rispettivamente m e n , si inizializza $\text{Lavoro} = \text{Disponibili}$ e $\text{Fine}[i] = \text{falso}$, per $i = 1, 2, \dots, n - 1$;
- si cerca un indice i tale che valgano contemporaneamente le seguenti relazioni:
 - $\text{Fine}[i] == \text{falso}$
 - $\text{Necessità}_i \leq \text{Lavoro}$
 Se tale i non esiste, si esegue il passo 4;
- $\text{Lavoro} = \text{Lavoro} + \text{Assegnate}_i$
 $\text{Fine}[i] = \text{vero}$
 si va al passo 2;
- se $\text{Fine}[i] == \text{vero}$ per ogni i , allora il sistema è in uno stato sicuro.

8.5.3.2 ALGORITMO DI RICHIESTA DELLE RISORSE

Si descrive ora l'algoritmo che determina se le richieste possano essere soddisfatte mantenendo la condizione di sicurezza.

Sia Richieste_i il vettore delle richieste per il processo P_i . Se $\text{Richieste}_i[j] == k$, allora il processo P_i richiede k istanze del tipo di risorsa R_j . Se il processo P_i effettua una richiesta di risorse, si svolgono le seguenti azioni:

- Se $\text{Richieste}_i \leq \text{Necessità}_i$, si va al passo 2, altrimenti si riporta una condizione d'errore, poiché il processo ha superato il numero massimo di richieste;

- Se $\text{Richieste}_i \leq \text{Disponibili}$, si esegue il passo 3, altrimenti P_i deve attendere poiché le risorse non sono disponibili;
- Si simula l'assegnazione al processo P_i delle risorse richieste modificando come segue lo stato di assegnazione delle risorse:
 $\text{Disponibili} = \text{Disponibili} - \text{Richieste}_i$
 $\text{Assegnate}_i = \text{Assegnate}_i + \text{Richieste}_i$
 $\text{Necessità}_i = \text{Necessità}_i - \text{Richieste}_i$
- Se lo stato di assegnazione delle risorse risultante è sicuro, la transazione è completata e al processo P_i si assegnano le risorse richieste. Tuttavia, se il nuovo stato è non sicuro, P_i deve attendere Richieste_i e si ripristina il vecchio stato di assegnazione delle risorse.

8.5.3.3 ESEMPIO

Illustriamo l'uso dell'algoritmo del banchiere, considerando un sistema con cinque processi, da P_0 a P_4 , e tre tipi di risorse: A, B, e C. Il tipo di risorse A ha 10 istanze, il tipo B ha 5 istanze e il tipo C ha 7 istanze. Si supponga che all'istante T_0 si sia verificata la seguente situazione del sistema:

	Assegnate	Massimo	Disponibili
	A B C	A B C	A B C
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

Il contenuto della matrice Necessità è definito come Massimo – Assegnate:

	Necessità
	A B C
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

Possiamo affermare che il sistema si trova attualmente in uno stato sicuro; infatti, la sequenza $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ soddisfa i criteri di sicurezza. Si supponga ora che il processo P_1 richieda un'altra istanza del tipo di risorsa A e due istanze del tipo C, quindi $\text{Richieste}_1 = (1, 0, 2)$. Per stabilire se questa richiesta si possa soddisfare immediatamente verifichiamo la condizione $\text{Richieste}_1 \leq \text{Disponibili}$ (vale a dire $(1, 0, 2) \leq (3, 3, 2)$), che risulta vera.

A questo punto simuliamo che questa richiesta sia stata soddisfatta, e otteniamo il seguente nuovo stato:

	Assegnate	Necessità	Disponibili
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Occorre stabilire se questo nuovo stato del sistema sia sicuro; a tale scopo si esegue l'algoritmo di verifica della sicurezza da cui risulta che la sequenza $\langle P_1, P_3, P_4, P_2, P_0 \rangle$ rispetta il requisito di sicurezza.

Tuttavia, dovrebbe essere chiaro che, quando il sistema si trova in questo stato, una richiesta di (3, 3, 0) da parte di P_4 non si può soddisfare perché non sono disponibili le risorse. Inoltre, una richiesta di (0, 2, 0) da parte di P_0 non si può soddisfare, anche se le risorse sono disponibili, poiché lo stato risultante sarebbe non sicuro.

L'implementazione dell'algoritmo del banchiere è lasciata come esercizio di programmazione.

8.6 RILEVAMENTO DELLE SITUAZIONI DI STALLO

Se un sistema non si avvale di un algoritmo per prevenire o evitare lo stallo, è possibile che si verifichi una di queste situazioni. In un ambiente di questo genere, il sistema può fornire i seguenti algoritmi:

- un algoritmo che esamini lo stato del sistema per stabilire se si è verificato uno stallo;
- un algoritmo che ripristini il sistema dalla condizione di stallo.

8.6.1 ISTANZA SINGOLA DI CIASCUN TIPO DI RISORSA

Se tutte le risorse hanno una sola istanza si può definire un algoritmo di rilevamento di situazioni di stallo che fa uso di una variante del grafo di assegnazione delle risorse, detta **grafo d'attesa**, ottenuta dal grafo di assegnazione delle risorse togliendo i nodi dei tipi di risorse e componendo gli archi tra i processi.

Più precisamente, un arco da P_i a P_j del grafo d'attesa implica che il processo P_i attende che il processo P_j rilasci una risorsa di cui P_j ha bisogno.

Come in precedenza, nel sistema esiste uno stallo se e solo se il grafo d'attesa contiene un ciclo. Per individuare le situazioni di stallo, il sistema deve *Mantenere aggiornato* il grafo d'attesa e *invocare periodicamente un algoritmo* che cerchi un ciclo all'interno del grafo.

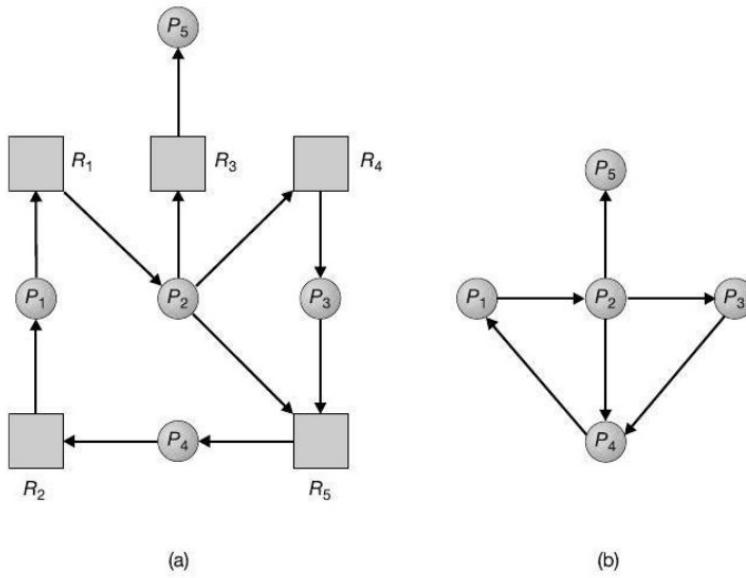


Figura 7.9 (a) Grafo di assegnazione delle risorse; (b) Grafo d'attesa corrispondente.

8.6.2 PIÙ ISTANTE PER CIASCUN TIPO DI RISORSA

Lo schema con grafo d'attesa non si può applicare ai sistemi di assegnazione delle risorse con più istanze di ciascun tipo di risorsa. Il seguente algoritmo di rilevamento di situazioni di stallo è, invece, applicabile a tali sistemi:

- **Disponibili**: vettore di lunghezza m che indica il numero delle istanze disponibili per ciascun tipo di risorsa.
- **Assegnate**: Matrice $n \times m$ che definisce il numero delle istanze di ciascun tipo di risorse correntemente assegnate a ciascun processo.
- **Richieste**: Matrice $n \times m$ che indica la richiesta attuale di ciascun processo. Se $\text{Richieste}[i, j] = k$, significa che il processo P_i sta richiedendo altre k istanze del tipo di risorsa R_j .

Per semplificare la notazione, le righe delle matrici Assegnate e Richieste si trattano come vettori e, nel seguito, sono indicate rispettivamente come Assegnate_i e Richieste_i .

L'algoritmo è il seguente:

- Siano Lavoro e Fine vettori di lunghezza m e n , si inizializza $\text{Lavoro} = \text{Disponibili}$; per $i = 1, 2, \dots, n$, se $\text{Assegnate}_i \neq 0$, allora $\text{Fine}[i] = \text{falso}$, altrimenti $\text{Fine}[i] = \text{vero}$;
- si cerca un indice i tale che valgano contemporaneamente le seguenti relazioni:
 $\text{Fine}[i] == \text{falso}$
 $\text{Richieste}_i \leq \text{Lavoro}$
 se tale i non esiste, si esegue il passo 4.
- $\text{Lavoro} = \text{Lavoro} + \text{Assegnate}_i$
 $\text{Fine}[i] = \text{vero}$
 si torna al passo 2.
- se $\text{Fine}[i] == \text{falso}$ per qualche i , $0 \leq i < n$, allora il sistema è in stallo, inoltre, se $\text{Fine}[i] == \text{falso}$, il processo P_i è in stallo.

Per illustrare questo algoritmo, si consideri un sistema con cinque processi, da P_0 a P_4 , e tre tipi di risorse: A, B, e C. Il tipo di risorsa A ha 7 istanze, il tipo B ha 2 istanze e il tipo C ne ha 6. Si supponga di avere, all'istante T_0 , il seguente stato di assegnazione delle risorse:

	Assegnate	Richieste	Disponibili
	A B C	A B C	A B C
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Il sistema non è in stallo. Infatti eseguendo l'algoritmo troviamo che la sequenza $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ porta a `Fine[i] == vero` per ogni i .

Si supponga ora che il processo P_2 richieda un'altra istanza di tipo C. La matrice Richieste viene modificata come segue:

	Richieste
	A B C
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

Ora il sistema è in stallo. Anche se si possono liberare le risorse possedute dal processo P_0 , il numero delle risorse disponibili non è sufficiente per soddisfare le richieste degli altri processi, quindi si verifica uno stallo composto dai processi P_1, P_2, P_3 e P_4 .

8.6.3 USO DELL'ALGORITMO DI RILEVAMENTO

Per sapere quando è necessario ricorrere all'algoritmo di rilevamento si devono considerare i seguenti fattori:

- frequenza presunta con la quale si verifica uno stallo;
- numero dei processi che sarebbero influenzati da tale stallo.

Se le situazioni di stallo sono frequenti, è necessario ricorrere spesso all'algoritmo per il loro rilevamento. Le risorse assegnate a processi in stallo rimangono inattive fino all'eliminazione dello stallo. Inoltre, il numero dei processi coinvolti nel ciclo di stallo può aumentare.

Non è conveniente richiedere l'algoritmo di rilevamento in momenti arbitrari, perché nel grafo delle risorse posso coesistere molti cicli e, normalmente, non si può dire quale fra i tanti processi in stallo abbia "causato" lo stallo.

8.7 RIPRISTINO DI SITUAZIONI DI STALLO

Una volta rilevato uno stallo, questo si può affrontare in diversi modi. Una soluzione consiste nell'informare l'operatore della presenza dello stallo, in modo che possa gestirlo manualmente. L'altra soluzione lascia al sistema il ripristino automatico dalla situazione di stallo.

8.7.1 RIPRISTINO DEI PROCESSI

Per eliminare le situazioni di stallo attraverso la terminazione di processi si possono adoperare due metodi; in entrambi il sistema recupera tutte le risorse assegnate ai processi terminati:

- **Terminazione di tutti i processi in stallo:** Chiaramente questo metodo interrompe il ciclo di stallo, ma l'operazione è molto onerosa;
- **Terminazione di un processo alla volta fino all'eliminazione del ciclo di stallo:** Questo metodo comporta un notevole overhead, poiché, dopo aver terminato ogni processo, si deve invocare un algoritmo di rilevamento per stabilire se esistono ancora processi in stallo.

Se si adopera il metodo di terminazione parziale, occorre determinare quale processo, o quali processi, in situazione di stallo devono essere terminati. Analogamente ai problemi di scheduling della CPU, si tratta di seguire un criterio. Le considerazioni sono essenzialmente economiche: si dovrebbero arrestare i processi la cui terminazione causa il minimo costo. Sfortunatamente, il termine *minimo costo* non è preciso. La scelta dei processi è influenzata da diversi fattori, tra cui i seguenti:

- La priorità del processo;
- il tempo trascorso in computazione e il tempo ancora necessario per completare il compito assegnato al processo;
- la quantità e il tipo di risorse impiegate dal processo (per esempio, se si può effettuare facilmente la prelazione delle risorse);
- la quantità di ulteriori risorse di cui il processo ha ancora bisogno per completare l'esecuzione;
- il numero di processi che si devono terminare;
- il tipo di processo: interattivo o batch.

8.7.2 PRELAZIONE DELLE RISORSE

Per eliminare uno stallo si può esercitare la prelazione sulle risorse: le risorse si sottraggono in successione ad alcuni processi e si assegnano ad altri finché si ottiene l'interruzione del ciclo di stallo. Se per gestire le situazioni di stallo s'impiega la prelazione, si devono considerare i seguenti problemi:

- **Selezione di una vittima:** occorre stabilire quali risorse e quali processi si devono sottoporre a prelazione.
- **Ristabilimento di un precedente stato sicuro:** occorre stabilire che cosa fare con un processo cui è stata sottratta una risorsa. poiché è stato privato di una risorsa necessaria, la sua esecuzione non può continuare normalmente, quindi deve essere ricondotto a un precedente stato sicuro (*rollback*) dal quale essere riavviato.
- **Attesa indefinita (starvation):** È necessario assicurare che non si verifichino situazioni d'attesa indefinita, occorre cioè garantire che le risorse non siano sottratte sempre allo stesso processo.

8.8 APPROCCIO COMBINATO PER LA GESTIONE DELLO STALLO

La combinazione dei tre approcci di base (*prevenire*, *evitare*, *rilevare*) consente l'uso dell'approccio ottimale per ciascuna delle risorse del sistema.

Partizioni delle risorse in classi ordinate gerarchicamente.

Uso della tecnica appropriata per la gestione dello stato all'interno di ciascuna classe.

9 – GESTIONE DELLA MEMORIA

9.1 INTRODUZIONE

La memoria consiste in un grande vettore di byte, ciascuno con il proprio indirizzo. La Cpu preleva le istruzioni dalla memoria sulla base del contenuto del contatore di programma; tali istruzioni possono determinare ulteriori letture (*load*) e scritture (*store*) in specifici indirizzi di memoria.

Un tipico ciclo d'esecuzione di un'istruzione, per esempio, prevede che l'istruzione sia prelevata dalla memoria; quindi viene decodificata (e ciò può comportare il prelievo di operandi dalla memoria) e poi eseguita sugli eventuali operandi; i risultati si possono scrivere in memoria.

9.1.1 ASSOCIAZIONE DEGLI INDIRIZZI

In genere un programma risiede in un disco sotto forma di un file binario eseguibile. Per essere eseguito, il programma va caricato in memoria e inserito nel contesto di un processo. Secondo il tipo di gestione della memoria adoperato, durante la sua esecuzione, il processo si può trasferire dalla memoria al disco e viceversa. L'insieme dei processi presenti nei dischi e che attendono d'essere trasferiti in memoria per essere eseguiti forma la **coda d'ingresso** (*input queue*).

Nella maggior parte dei casi un programma utente, prima di essere eseguito, deve passare attraverso vari passi, alcuni dei quali possono essere facoltativi.

Generalmente, l'associazione di istruzioni e dati a indirizzi di memoria si può compiere in qualsiasi passo del seguente percorso:

- **Compilazione:** Se nella fase di compilazione si sa dove il processo risiederà in memoria, si può generare **codice assoluto**. Se, per esempio, è noto a priori che un processo utente inizia alla locazione r , anche il codice generato dal compilatore comincia da quella locazione. Se, in un momento successivo, la locazione iniziale cambiasse, sarebbe necessario ricompilare il codice. Come vantaggio c'è il fatto che, avendo già generato il codice, poi non lo si debba fare più. Ma c'è come svantaggio il fatto che, una volta caricato nella memoria centrale, il codice non può essere più spostato (a meno che non si decida di ricompilare il programma).
- **Caricamento:** Se nella fase di compilazione non è possibile sapere in che punto della memoria risiederà il processo, il compilatore deve generare codice rilocabile. In questo caso si ritarda l'associazione finale degli indirizzi alla fase del caricamento. Se l'indirizzo iniziale cambia, è sufficiente ricaricare il codice utente per incorporare il valore modificato. A differenza della fase di compilazione, in questo caso, prima che l'applicazione venga caricata nella memoria centrale, c'è ancora tempo per effettuare eventualmente un ripensamento, una migliore allocazione;
- **Esecuzione:** Se durante l'esecuzione il processo può essere spostato da un segmento di memoria a un altro, si deve ritardare l'associazione degli indirizzi fino alla fase d'esecuzione. per realizzare questo schema è necessario disporre di hardware specializzato. La maggior parte dei sistemi operativi general purpose impiega questo metodo. Il vantaggio sta nel fatto che il programma, pur trovandosi nella memoria (ma non ancora eseguito), c'è la possibilità di adattarsi a una situazione successiva e migliorare la condizione in cui ci si trova.

Tutto ciò può essere rappresentato da questo grafico:

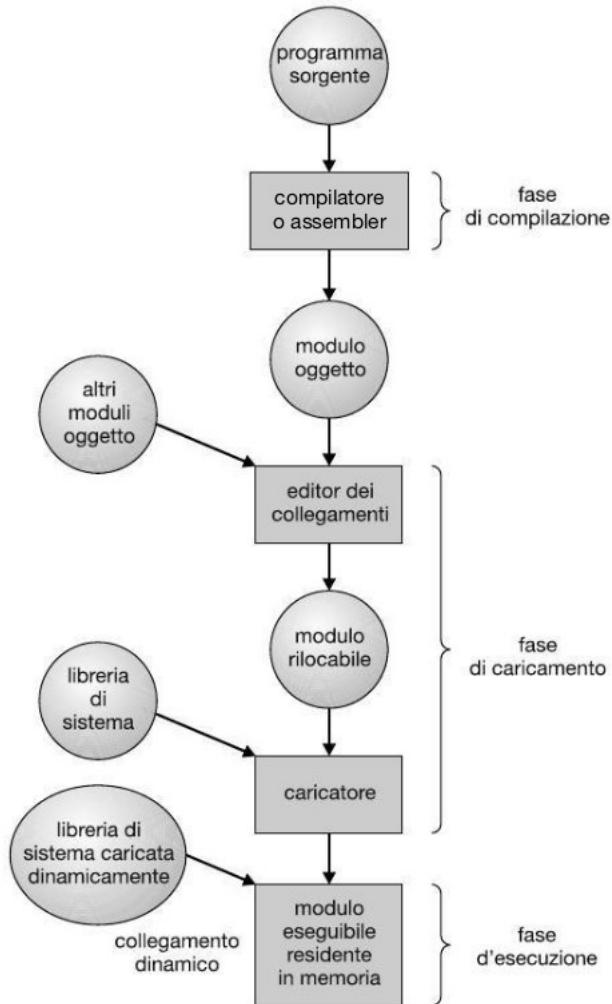


Figura 8.3 Fasi di elaborazione di un programma utente.

9.1.2 SPAZI DI INDIRIZZI LOGICI E FISICI

Un indirizzo generato dalla CPU è normalmente chiamato **indirizzo logico**, mentre un indirizzo visto dall'unità di memoria, cioè caricato nel **registro dell'indirizzo di memoria** (*memory address register*, MAR) è normalmente chiamato **indirizzo fisico**.

I metodi di associazione degli indirizzi nelle fasi di compilazione e di caricamento producono indirizzi logici e fisici identici. Con il metodo di associazione nella fase d'esecuzione, invece, gli indirizzi logici non coincidono con gli indirizzi fisici. In questo caso ci si riferisce, di solito, agli indirizzi logici col termine **indirizzi virtuali**. L'insieme di tutti gli indirizzi logici generati da un programma è lo **spazio degli indirizzi logici**; l'insieme degli indirizzi fisici corrispondenti a tali indirizzi logici è lo **spazio degli indirizzi fisici**. Quindi, con lo schema di associazione degli indirizzi nella fase d'esecuzione, lo spazio degli indirizzi logici differisce dallo spazio degli indirizzi fisici.

L'associazione nella fase d'esecuzione dagli indirizzi virtuali agli indirizzi fisici è svolta da un dispositivo detto **unità di gestione della memoria** (*memory-management unit*, MMU).

Com'è illustrato nella prossima figura, il registro di base è ora denominato registro di rilocazione: quando un processo utente genera un indirizzo, prima dell'invio all'unità di memoria, si somma a tale indirizzo il valore contenuto nel registro di rilocazione. Per

esempio, se il registro di rilocazione contiene il valore 14000, un tentativo da parte dell'utente di accedere alla locazione 0 è dinamicamente rilocato alla locazione 14000; un accesso alla locazione 346 è mappato alla locazione 14346.

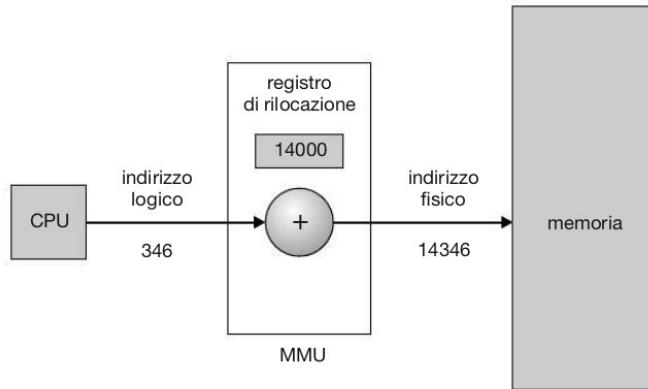


Figura 8.4 Rilocazione dinamica tramite un registro di rilocazione.

Il programma utente non vede mai i reali indirizzi fisici. Il programma crea un puntatore alla locazione 346, lo memorizza, lo modifica, lo confronta con altri indirizzi, tutto ciò sempre come il numero 346. Solo quando assume il ruolo di un indirizzo di memoria (magari in una load o una store indiretta), si riloca il numero sulla base del contenuto del registro di rilocazione. Il programma utente tratta indirizzi logici, l'architettura del sistema converte gli indirizzi logici in indirizzi fisici. La locazione finale di un riferimento a un indirizzo di memoria non è determinata finché non si compie effettivamente il riferimento.

9.1.3 CARICAMENTO DINAMICO

Nella discussione svolta fin'ora, era necessario che l'intero programma e i dati di un processo fossero presenti nella memoria fisica perché il processo potesse essere eseguito. La dimensione di un processo era quindi limitata alle dimensioni della memoria fisica. Per migliorare l'utilizzo della memoria si può ricorrere al **caricamento dinamico** (*dynamic loading*), mediante il quale si carica una procedura solo quando viene richiamata; tutte le procedure si tengono su disco in un formato di caricamento rilocabile.

Si carica il programma principale in memoria e quando, durante l'esecuzione, una procedura deve richiamarne un'altra, controlla innanzitutto che sia stata caricata. Se non è stata caricata, si richiama il linking loader rilocabile per caricare in memoria la procedura richiesta e aggiornare le tabelle degli indirizzi del programma in modo che registrino questo cambiamento. A questo punto il controllo passa alla procedura appena caricata.

Il vantaggio dato dal caricamento dinamico consiste nel fatto che una procedura viene caricata solo quando serve. Questo metodo è utile soprattutto quando servono grandi quantità di codice per gestire casi non frequenti, per esempio le procedure di gestione degli errori. In questi casi, anche se un programma può avere dimensioni totali elevate, la parte effettivamente usata, e quindi effettivamente caricata, può essere molto più piccola.

Il caricamento dinamico non richiede un supporto particolare del sistema operativo. Spetta agli utenti progettare i programmi in modo da trarre vantaggio da un metodo di questo tipo. Il sistema operativo può tuttavia aiutare il programmatore fornendo librerie di procedure che realizzano il caricamento dinamico.

9.1.4 LINKING DINAMICO (COLLEGAMENTO DINAMICO)

Le librerie collegate dinamicamente sono librerie di sistema che vengono collegate ai programmi utente quando questi vengono eseguiti. Con il linking dinamico, per ogni riferimento a una procedura di libreria s'inscrive all'interno dell'eseguibile una piccola porzione di codice di riferimento (**stub**), che indica come localizzare la giusta procedura di libreria residente in memoria o come caricare la libreria se la procedura non è già presente. Durante l'esecuzione, lo stub controlla se la procedura richiesta è già in memoria, altrimenti provvede a caricarla; in entrambi i casi lo stub sostituisce se stesso con l'indirizzo della procedura.

A differenza del caricamento dinamico, il linking dinamico richiede generalmente l'assistenza del sistema operativo. Se i processi presenti in memoria sono protetti l'uno dall'altro, il sistema operativo è l'unica entità che può controllare se la procedura richiesta da un processo è nello spazio di memoria di un altro processo, o che può consentire l'accesso di più processi agli stessi indirizzi di memoria.

Quindi, il linking dinamico si rivela molto utile per le librerie.

9.1.5 SOVRAPPOSIZIONE DI SEZIONI (OVERLAY)

Se il caricamento dinamico offreva una soluzione all'esecuzione dei processi con dimensioni maggiori della memoria centrale, la tecnica dell'overlay consente a un processo di essere più grande della quantità di memoria a esso allocata, ovvero al suo spazio di indirizzamento. Il principio è simile: mantenere in memoria solo le istruzioni e i dati che sono necessari in un certo momento, rimpiazzando man mano quelle più vecchie.

Si può dire che un overlay è una sorta di partizione del programma di partenza, indipendente dagli altri e quindi in grado di compiere autonomamente tutta una serie di operazioni che lo caratterizzano. Quando la sua esecuzione parziale del programma si esaurisce, viene deallocated e al suo posto ne viene caricato un altro.

Gli overlay non richiedono alcun supporto speciale da parte del sistema operativo, è compito del programmatore definirne il numero e le suddivisioni. Questo si rivela però un compito improbo, dato che i programmi in questione sono generalmente molto grandi (per quelli piccoli questa tecnica non è evidentemente necessaria) e suddividerli in parti implica una conoscenza puntigliosa della loro struttura. Inoltre i moduli possono avere dimensioni diverse, con conseguenti sprechi di spazio quando se ne caricheranno alcuni di grandezza più ridotta. Troppe responsabilità al programmatore e sfruttamento della memoria centrale poco efficiente fanno così dell'overlay una tecnica utilizzata solo per sistemi con poca memoria fisica e supporto hardware poco avanzato.

9.1.6 AVVICENDAMENTO DEI PROCESSI (SWAPPING)

Per essere eseguito, un processo deve trovarsi nella memoria centrale. Un processo, tuttavia, può essere temporaneamente tolto dalla memoria centrale e spostato in una **memoria ausiliaria** (*backing store*) e in seguito riportato in memoria per continuare l'esecuzione. Questo procedimento si chiama **avvicendamento dei processi in memoria** – o, più brevemente, **swapping**. Grazie all'avvicendamento dei processi lo spazio totale degli indirizzi fisici di tutti i processi può eccedere la reale dimensione della memoria fisica del sistema, aumentando così il grado di multiprogrammazione possibile.

La memoria ausiliaria è un disco veloce sufficientemente capiente da contenere le copie di tutte le immagini di memoria di tutti i processi; deve permettere un accesso diretto a queste immagini di memoria.

Una variante di questo processo è il **roll out/roll in**, impiegata per gli algoritmi di scheduling basati sulle priorità: il processo con priorità inferiore viene scaricato dalla memoria centrale per fare spazio all'esecuzione del processo con priorità maggiore.

La maggior parte del tempo di avvicendamento è data dal tempo di trasferimento. Il tempo di trasferimento totale è direttamente proporzionale alla quantità di memoria interessata.

Attualmente l'avvicendamento nella sua forma standard si usa in pochi sistemi; richiede infatti un elevato tempo di trasferimento, e consente un tempo di esecuzione troppo breve per essere considerato una soluzione ragionevole al problema di gestione della memoria. Si possono trovare comunque versioni modificate di tecniche di avvicendamento su molti sistemi, es. UNIX, Linux, Windows.

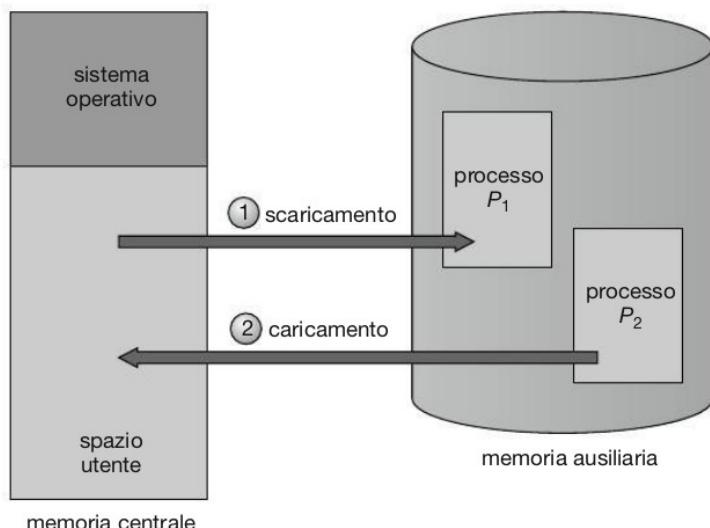


Figura 8.5 Avvicendamento (swapping) di due processi con un disco come memoria ausiliaria.

9.2 ALLOCAZIONE CONTIGUA DELLA MEMORIA

La memoria centrale deve contenere sia il sistema operativo sia i vari processi utenti; perciò è necessario assegnare le diverse parti della memoria centrale nel modo più efficiente.

La memoria centrale di solito si divide in due partizioni, una per il sistema operativo residente e una per i processi utente. Il sistema operativo si può collocare sia nella parte bassa sia nella parte alta della memoria. Il fattore che incide in modo decisivo su tale scelta è generalmente la posizione del vettore delle interruzioni, poiché si trova spesso nella parte bassa dello spazio degli indirizzi fisici, i programmati collocano di solito anche il sistema operativo nella parte bassa della memoria. Per questo motivo nel seguito prendiamo in considerazione solo la situazione in cui il sistema operativo risiede in quest'area di memoria.

Generalmente si vuole che più processi utenti risiedano contemporaneamente in memoria centrale. Perciò è necessario considerare come assegnare la memoria disponibile ai processi presenti nella coda d'ingresso che attendono di essere caricati in memoria. Con

l'allocazione contigua della memoria, ciascun processo è contenuto in una singola sezione di memoria contigua a quella che contiene il processo successivo.

9.2.1 PROTEZIONE DELLA MEMORIA

Prima di trattare l'allocazione della memoria, dobbiamo soffermarci sul problema della protezione della memoria. Possiamo evitare che un processo acceda alla memoria che non gli appartiene combinando due idee discusse in precedenza. Se abbiamo un sistema con un registro di rilocazione e un registro limite abbiamo già raggiunto il nostro obiettivo. Il registro di rilocazione contiene il valore dell'indirizzo fisico minore; il registro limite contiene l'intervallo di indirizzi logici, per esempio, *rilocazione* = 100.040 e *limite* = 74.600.

Lo schema con registro di rilocazione consente al sistema operativo di cambiare dinamicamente le proprie dimensioni. Tale flessibilità è utile in molte situazioni; il sistema operativo, per esempio, contiene codice e spazio di memoria per i driver dei dispositivi; se uno di questi, o un altro servizio del sistema operativo, non è comunemente usato, è inutile tenerne in memoria codice e dati, poiché lo spazio occupato si potrebbe usare per altri scopi. Talvolta questo codice si chiama **codice transiente** del sistema operativo, poiché s'inserisce secondo le necessità; l'uso di tale codice cambia le dimensioni del sistema operativo durante l'esecuzione del programma.

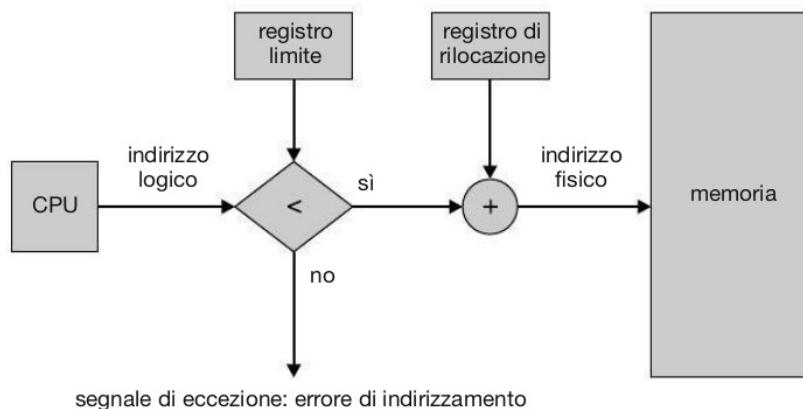


Figura 8.6 Registri di rilocazione e limite.

9.2.2 ALLOCAZIONE DELLA MEMORIA

Uno dei metodi più semplici per l'allocazione della memoria consiste nel suddividere la stessa in **partizioni** di dimensione fissa. Ogni partizione deve contenere esattamente un processo, quindi il grado di multiprogrammazione è limitato dal numero di partizioni. In questo **metodo delle partizioni multiple** quando una partizione è libera può essere occupata da un processo presente nella coda d'ingresso; terminato il processo, la partizione diviene nuovamente disponibile per un altro processo.

Nello schema a partizione variabile il sistema operativo conserva una tabella in cui sono indicate le partizioni di memoria disponibili e quelle occupate. Inizialmente tutta la memoria è a disposizione dei processi utenti; si tratta di un grande blocco di memoria disponibile, un **bucco** (*hole*). Nel lungo periodo, come si vede nel seguito, la memoria contiene una serie di buchi di diverse dimensioni.

Quando entrano nel sistema, i processi vengono inseriti in una coda d'ingresso. Per determinare a quali processi si debba assegnare la memoria, il sistema operativo tiene conto dei requisiti di memoria di ciascun processo e della quantità di spazio di memoria disponibile. Quando a un processo si assegna dello spazio, il processo stesso viene

caricato in memoria e può quindi competere per il controllo della CPU. Al termine, rilascia la memoria che gli era stata assegnata, e il sistema operativo può impiegarla per un altro processo presente nella coda d'ingresso.

In generale, è sempre presente un insieme di buchi di diverse dimensioni sparsi per la memoria. Quando si presenta un processo che necessita di memoria, il sistema cerca nel gruppo un buco di dimensioni sufficienti per contenerlo. Se è troppo grande, il buco viene diviso in due parti: si assegna una parte al processo in arrivo e si riporta l'altra nell'insieme dei buchi. Quando termina, un processo rilascia il blocco di memoria, che si reinserisce nell'insieme dei buchi; se si trova accanto ad altri buchi, si uniscono tutti i buchi adiacenti per formarne uno più grande. A questo punto il sistema può controllare se vi siano processi nell'attesa di spazio di memoria, e se la memoria appena liberata e ricombinata possa soddisfare le richieste di qualcuno fra tali processi.

Questa procedura è una particolare istanza del più generale problema di **allocazione dinamica della memoria**, che consiste nel soddisfare una richiesta di dimensione n data una lista di buchi liberi. Le soluzioni sono numerose. I criteri più usati per scegliere un buco libero tra quelli disponibili nell'insieme sono i seguenti:

- **First-fit**: Si assegna il *primo* buco abbastanza grande. La ricerca può cominciare sia dall'inizio dell'insieme di buchi sia dal punto in cui era terminata la ricerca precedente. Si può fermare la ricerca non appena s'individua un buco libero di dimensioni sufficientemente grandi;
- **Best-fit**: Si assegna il *più piccolo* buco in grado di contenere il processo. Si deve compiere la ricerca in tutta la lista, a meno che questa non sia ordinata per dimensione. Tale criterio produce le parti di buco inutilizzate più piccole.
- **Worst-fit**: Si assegna il buco *più grande*. Anche in questo caso si deve esaminare tutta la lista, a meno che non sia ordinata per dimensione. Tale criterio produce le parti di buco inutilizzate più grandi, che possono essere più utili delle parti più piccole ottenute col criterio best-fit.

Con l'uso di simulazioni si è dimostrato che sia first-fit sia best-fit sono migliori rispetto a worst-fit in termini di risparmio di tempo e di utilizzo di memoria. D'altra parte nessuno dei due è chiaramente migliore dell'altro per quel che riguarda l'utilizzo della memoria ma, in genere, first-fit è più veloce.

9.2.3 FRAMMENTAZIONE

Entrambi i criteri first-fit e best-fit di allocazione della memoria soffrono di **frammentazione esterna**. Caricando e rimuovendo i processi dalla memoria, si frammenta lo spazio libero della memoria in tante piccole parti. Si ha la frammentazione esterna se lo spazio di memoria totale è sufficiente per soddisfare una richiesta, ma non è contiguo; la memoria è frammentata in tanti piccoli buchi.

Questo problema di frammentazione può essere molto grave; nel caso peggiore può verificarsi un blocco di memoria libera, sprecata, tra ogni coppia di processi. Se tutti questi piccoli pezzi di memoria costituissero in un unico blocco libero di grandi dimensioni, si potrebbero eseguire molti più processi.

La frammentazione può essere interna oltre che esterna. Nella **frammentazione interna** la memoria assegnata può essere leggermente maggiore della memoria richiesta; la memoria è interna a una partizione, ma non è in uso.

Una soluzione al problema della frammentazione esterna è data dalla **compattazione**. Lo scopo è quello di riordinare il contenuto della memoria per riunire la memoria libera in un unico grosso blocco. La compattazione tuttavia non è sempre possibile: non si può realizzare se la rilocazione è statica ed è effettuata nella fase di assemblaggio o di caricamento; è possibile solo se la rilocazione è dinamica e si effettua nella fase d'esecuzione.

9.3 PAGINAZIONE

La segmentazione permette che lo spazio degli indirizzi fisici di un processo non sia contiguo. La **paginazione** (*paging*) è un altro schema di gestione della memoria che offre lo stesso vantaggio. Tuttavia, a differenza della segmentazione, la paginazione evita la frammentazione esterna e la necessità di compattazione. Inoltre, la paginazione elimina il gravoso problema della sistemazione di blocchi di memoria di diverse dimensioni in memoria ausiliaria.

L'implementazione della paginazione è frutto della cooperazione tra il sistema operativo e l'hardware del computer.

9.3.1 METODO DI BASE

Il metodo di base per implementare la paginazione consiste nel suddividere la memoria fisica in blocchi di dimensione fissa, detti **frame**, e nel suddividere la memoria logica in blocchi di pari dimensione, detti **pagina**.

L'hardware di supporto alla paginazione è illustrato nella successiva figura; ogni indirizzo generato dalla Cpu è diviso in due parti: un **numero di pagina** (*p*), e un **offset di pagina** (*d*). Il numero di pagina serve come indice per la **tabella delle pagine**, contenente l'indirizzo di base in memoria fisica di ogni pagina. Questo indirizzo di base si combina con l'offset di pagina per definire l'indirizzo della memoria fisica, che s'invia all'unità di memoria.

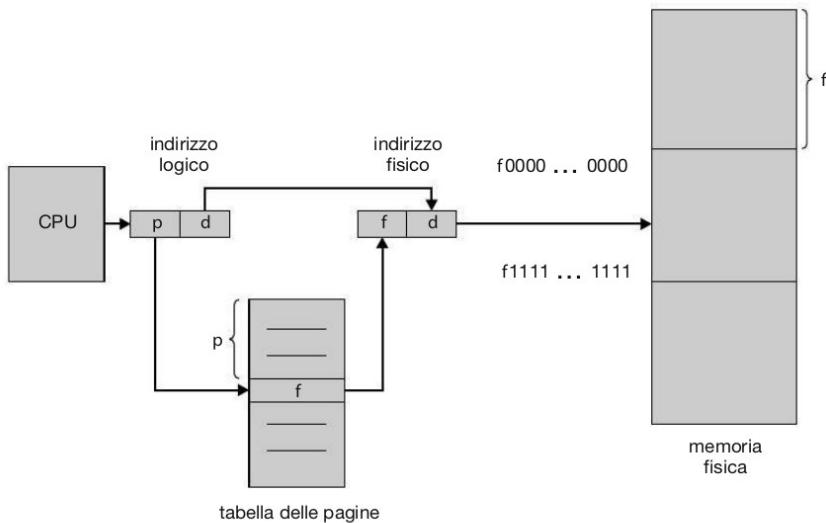


Figura 8.10 Hardware di paginazione.

La dimensione di una pagina, così come quella di un frame, è definita dall'hardware ed è, in genere, una potenza di 2 compresa tra 512 byte e 1 Gb, a seconda dell'architettura. La scelta di una potenza di 2 come dimensione della pagina facilita notevolmente la traduzione di un indirizzo logico nei corrispondenti numero di pagina e offset di pagina.

Per capire meglio il concetto, facciamo un esempio concreto seppur minimo. Consideriamo la memoria illustrata nella seguente figura:

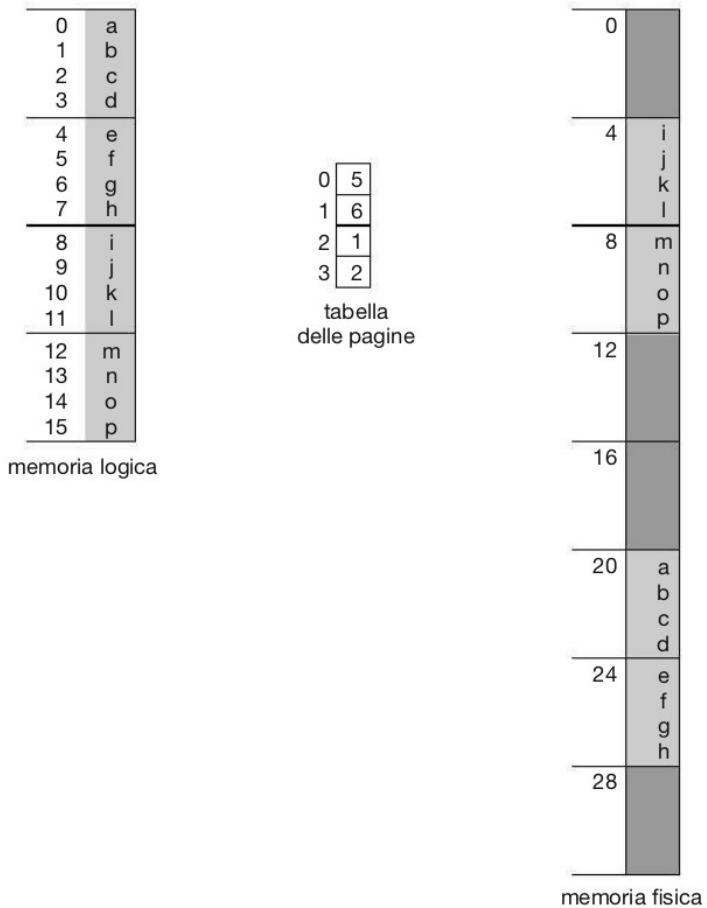


Figura 8.12 Esempio di paginazione per una memoria di 32 byte con pagine di 4 byte.

Qui, nell'indirizzo logico, $n = 2$ e $m = 4$. Utilizzando pagine di 4 byte e una memoria fisica di 32 byte (8 pagine), vediamo come si fa corrispondere la memoria vista dal programmatore alla memoria fisica. L'indirizzo logico 0 è la pagina 0 con offset 0. Secondo la tabella delle pagine, la pagina 0 si trova nel frame 5. Quindi all'indirizzo logico 0 corrisponde l'indirizzo fisico 20 [= $(5 \times 4) + 0$]. All'indirizzo logico 3 (pagina 0, offset 3) corrisponde l'indirizzo fisico 23 [= $(5 \times 4) + 3$]. per quel che riguarda l'indirizzo logico 4 (pagina 1, offset 0), secondo la tabella delle pagine, alla pagina 1 corrisponde il frame 6, quindi, all'indirizzo logico 4 corrisponde l'indirizzo fisico 24 [= $(6 \times 4) + 0$]. All'indirizzo logico 13 corrisponde l'indirizzo fisico 9.

Con la paginazione si evita la frammentazione esterna: qualsiasi frame libero si può assegnare a un processo che ne abbia bisogno; tuttavia si può avere la frammentazione interna.

Poiché il sistema operativo gestisce la memoria fisica, deve essere informato dei dettagli della allocazione: quali frame sono assegnati, quali sono disponibili, il loro numero totale, e così via. In genere queste informazioni sono contenute in una struttura dati chiamata **tabella dei frame**, contenente un elemento per ogni frame, indicante se sia libero oppure assegnato e, se è assegnato, a quale pagina di quale processo o di quali processi.

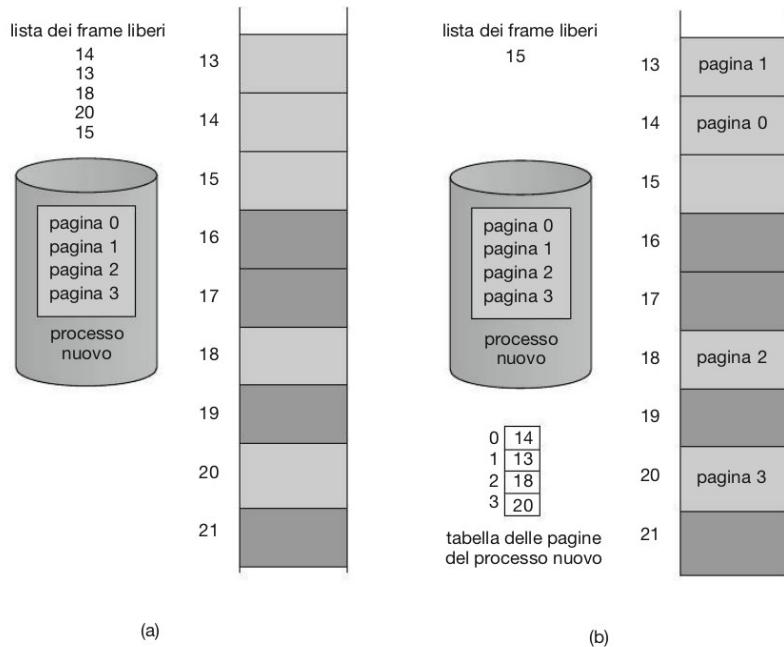


Figura 8.13 Frame liberi; (a) prima e (b) dopo l'allocazione.

9.3.2 ARCHITETTURA DI PAGINAZIONE

L'implementazione hardware della tabella delle pagine si può realizzare in modi diversi. Nel caso più semplice, si usa uno specifico insieme di **registri**, per garantire un'efficiente traduzione degli indirizzi di paginazione, questi registri devono essere realizzati con una logica molto veloce.

L'uso di registri per la tabella delle pagine è efficiente se la tabella stessa è ragionevolmente piccola, nell'ordine, per esempio, di 256 elementi. però la maggior parte dei calcolatori contemporanei usa tabelle molto grandi, per esempio di un milione di elementi, quindi non si possono impiegare i registri veloci per realizzare la tabella delle pagine; quest'ultima viene invece mantenuta nella memoria principale e un **registro di base della tabella delle pagine** (*page-table base register*, **PTBR**) punta alla tabella stessa. Il cambio della tabella delle pagine richiede soltanto di modificare questo registro, riducendo considerevolmente il tempo dei cambi di contesto. Viene inoltre utilizzato il **registro di lunghezza della tabella delle pagine** (*page-table length register*, **PTLR**) che indica la dimensione della tabella delle pagine.

Con questo metodo, per accedere a un byte occorrono *due accessi* alla memoria, uno per l'elemento della tabella delle pagine e uno per il byte stesso. La risoluzione a questo problema consiste nell'impiego di una speciale, piccola cache hardware, detta **TLB** (*translation look-aside buffer*). La TLB è una memoria associativa ad alta velocità in cui ogni elemento consiste di due parti: una chiave (o tag) e un valore.

Quando si presenta un elemento, la memoria associativa lo confronta contemporaneamente con tutte le chiavi; se trova una corrispondenza, riporta il valore correlato. La ricerca è molto rapida e in un hardware moderno è parte della pipeline delle istruzioni: non induce dunque nessuna penalizzazione in termini di prestazioni. Più semplicemente, nel momento in cui si da in input un certo numero di pagina automaticamente, il TLB, subito fa corrispondere un blocco di memoria.

Vediamo un po' come funziona il seguente schema:

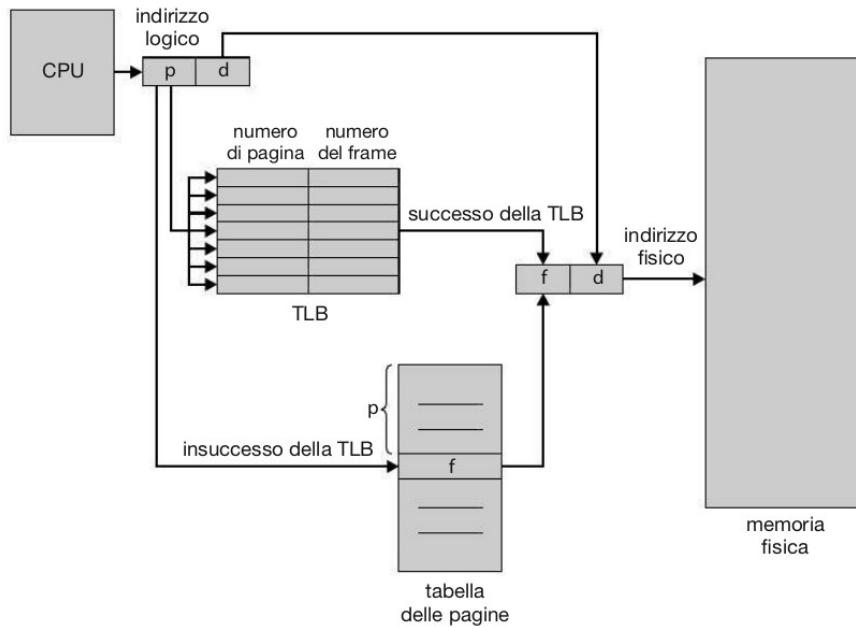


Figura 8.14 Hardware di paginazione con TLB.

La TLB si usa insieme con le tabelle delle pagine nel modo seguente: la TLB contiene una piccola parte degli elementi della tabella delle pagine; quando la CPU genera un indirizzo logico, si presenta il suo numero di pagina alla TLB; se tale numero è presente, il corrispondente numero del frame è immediatamente disponibile e si usa per accedere alla memoria.

Se nella TLB non è presente il numero di pagina, situazione nota come **insuccesso della TLB (TBL miss)**, si deve consultare la tabella delle pagine in memoria. La percentuale di volte che il numero di pagina di interesse si trova nella TLB è detta **tasso di successi (hit ratio)**.

Per il calcolo del **tempo effettivo di accesso** alla memoria supponiamo di avere:

- Lookup associativo pari a ϵ unità di tempo;
- Un tempo di accesso alla memoria pari a 1 microsecondo;
- Tasso di successi pari ad α ;

Il calcolo sarà dunque:

$$EAT : (1 + \epsilon)\alpha + (2 + \epsilon)(1 - \alpha) = 2 + \epsilon - \alpha$$

9.3.3 PROTEZIONE DELLA MEMORIA

In un ambiente paginato, la protezione della memoria è assicurata dai bit di protezione associati a ogni frame; normalmente tali bit si trovano nella tabella delle pagine.

Di solito si associa a ciascun elemento della tabella delle pagine un ulteriore bit, detto **bit di validità**. Tale bit, impostato a *valido*, indica che la pagina corrispondente è nello spazio d'indirizzi logici del processo, quindi è una pagina valida; impostato a *non valido*, indica che la pagina non è nello spazio d'indirizzi logici del processo. Il bit di validità consente quindi di riconoscere gli indirizzi illegali e di notificarne la presenza attraverso un'eccezione. Il sistema operativo concede o impedisce l'accesso a una pagina impostando in modo appropriato tale bit.

Vediamo un esempio concreto:

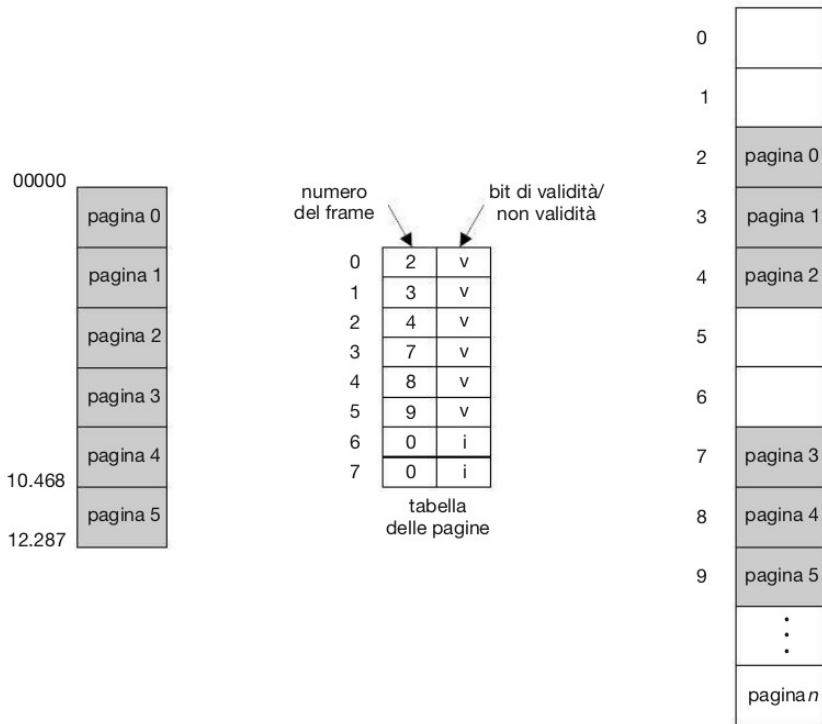


Figura 8.15 Bit di validità (v) o non validità (i) in una tabella delle pagine.

Supponiamo di avere una dimensione delle pagine di 2 KB. Gli indirizzi nelle pagine 0, 1, 2, 3, 4 e 5 sono tradotti normalmente tramite la tabella delle pagine. D'altra parte, ogni tentativo di generare un indirizzo nelle pagine 6 o 7 trova non valido il bit di validità; in questo caso il calcolatore invia un'eccezione al sistema operativo (riferimento di pagina non valido).

Questo schema ha creato un problema: poiché il programma si estende solo fino all'indirizzo 10.468, ogni riferimento oltre tale indirizzo è illegale; i riferimenti alla pagina 5 sono tuttavia classificati come validi, e ciò rende validi gli accessi sino all'indirizzo 12.287; solo gli indirizzi da 12.288 a 16.383 sono non validi. Tale problema è dovuto alla dimensione delle pagine di 2 kb e corrisponde alla frammentazione interna della paginazione.

9.4 STRUTTURA DELLA TABELLA DELLE PAGINE

9.4.1 PAGINAZIONE GERARCHICA

La maggior parte dei moderni calcolatori dispone di uno spazio d'indirizzi logici molto grande (da 2^{32} a 2^{64} elementi). In un ambiente di questo tipo la stessa tabella delle pagine diventa eccessivamente grande. Una semplice soluzione a questo problema consiste nel suddividere la tabella delle pagine in parti più piccole; questo risultato si può ottenere in molti modi.

Un metodo consiste nell'adottare un algoritmo di paginazione a due livelli, in cui la tabella stessa è paginata.

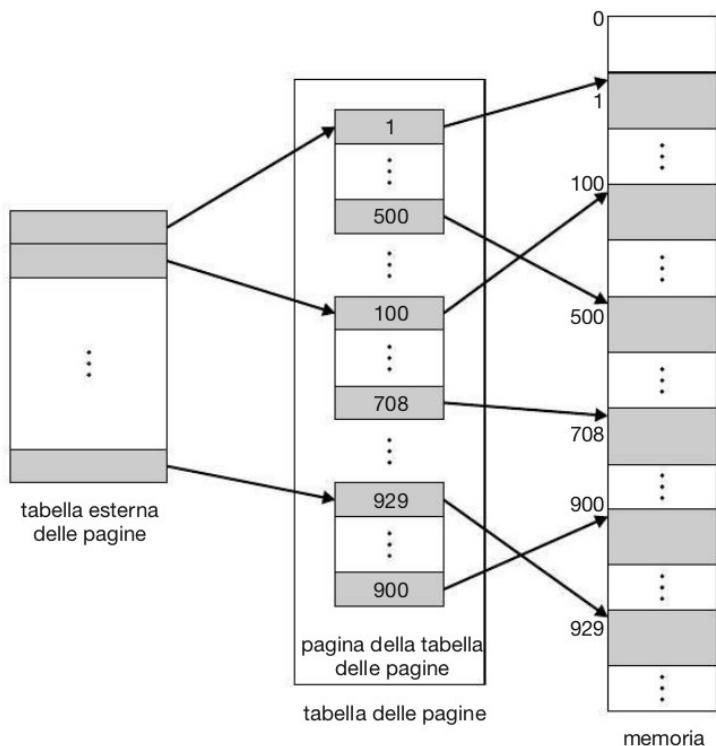


Figura 8.17 Schema di una tabella delle pagine a due livelli.

Si consideri un esempio di macchina a 32 bit con dimensione delle pagine di 4 KB. Ciascun indirizzo logico è suddiviso in un numero di pagina di 20 bit e in un offset di pagina di 12 bit. Paginando la tabella delle pagine, anche il numero di pagina è a sua volta suddiviso in un numero di pagina di 10 bit e un offset di pagina di 10 bit. Quindi, l'indirizzo logico è strutturato come segue:

numero di pagina		offset di pagina
p_1	p_2	d
10	10	12

dove p_1 è un indice della tabella delle pagine di primo livello, o tabella esterna delle pagine, e p_2 è l'offset all'interno della pagina indicata dalla tabella esterna delle pagine.

Poiché la traduzione degli indirizzi si svolge dalla tabella esterna delle pagine verso l'interno, questo metodo è anche noto come **tabella delle pagine ad associazione diretta** (*forward-mapped page table*).

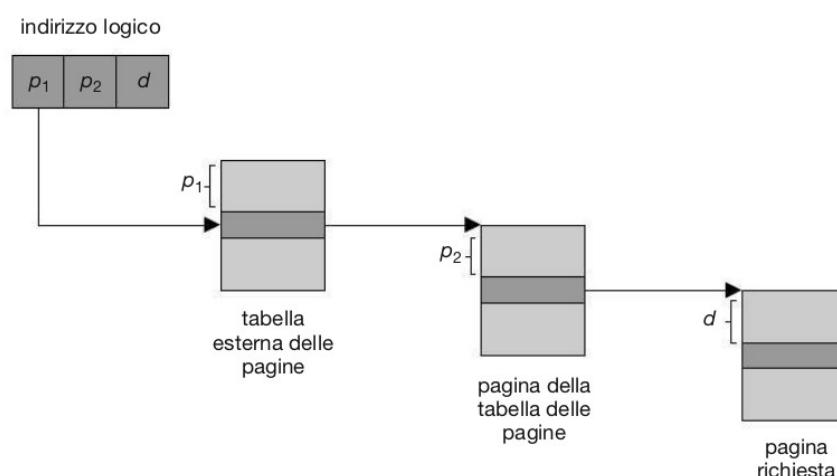


Figura 8.18 Traduzione degli indirizzi per un'architettura a 32 bit con paginazione a due livelli.

9.4.2 TABELLA DELLE PAGINE DI TIPO HASH

Un metodo di gestione molto comune degli spazi d'indirizzi oltre i 32 bit consiste nell'impiego di una **tabella delle pagine di tipo hash**, in cui l'argomento della funzione hash è il numero della pagina virtuale.

Ogni elemento della tabella hash contiene una lista concatenata di elementi che la funzione hash fa corrispondere alla stessa locazione. Ciascun elemento è composto da tre campi: (1) il numero della pagina virtuale; (2) l'indirizzo del frame corrispondente alla pagina virtuale; (3) un puntatore al successivo elemento della lista.

L'algoritmo opera come segue: si applica la funzione hash al numero della pagina virtuale contenuto nell'indirizzo virtuale, identificando un elemento della tabella. Si confronta il numero di pagina virtuale con il campo (1) del primo elemento della lista concatenata corrispondente. Se i valori coincidono, si usa l'indirizzo del relativo frame (campo 2) per generare l'indirizzo fisico desiderato. Altrimenti, l'algoritmo esamina allo stesso modo gli elementi successivi della lista concatenata.

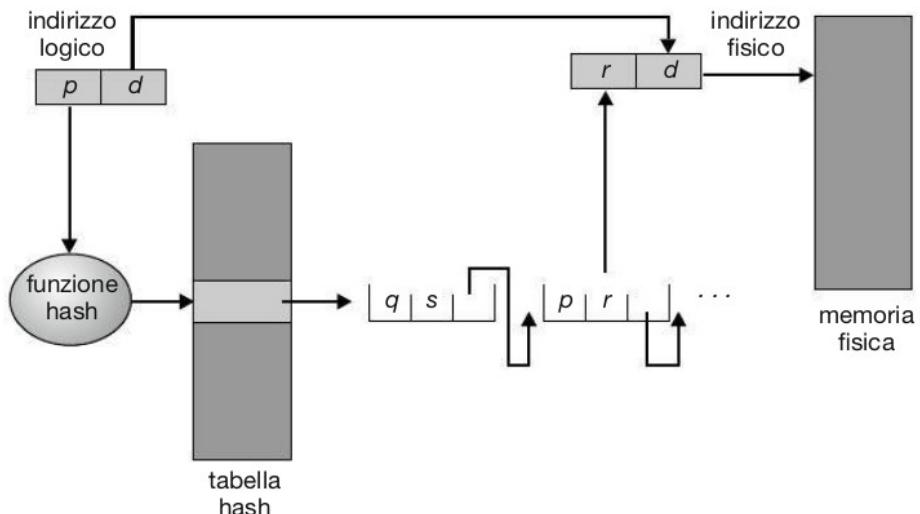


Figura 8.19 Tabella delle pagine di tipo hash.

La funzione hash presenta però degli svantaggi. Ad esempio, soffre del problema delle **collisioni**: si ha una collisione quando si deve inserire nella tabella hash un elemento con chiave u , e nella tabella esiste già un elemento con chiave v tale che $h(u) = h(v)$ [h è la funzione hash].

9.4.3 TABELLA DELLE PAGINE INVERTITA

Uno degli inconvenienti insiti nel metodo standard (in cui si associa una tabella delle pagine a ogni processo e tale tabella contiene un elemento per ogni pagina virtuale che il processo sta utilizzando) è che ciascuna tabella delle pagine può contenere milioni di elementi e occupare grandi quantità di memoria fisica, necessaria solo per sapere com'è impiegata la rimanente memoria fisica.

Per risolvere questo problema si può fare uso della **tabella delle pagine invertita**. Una tabella delle pagine invertita ha un elemento per ogni pagina reale (o frame). Ciascun elemento è quindi costituito dell'indirizzo virtuale della pagina memorizzata in quella reale locazione di memoria, con informazioni sul processo che possiede tale pagina. Quindi, nel sistema esiste una sola tabella delle pagine che ha un solo elemento per ciascuna pagina di memoria fisica.

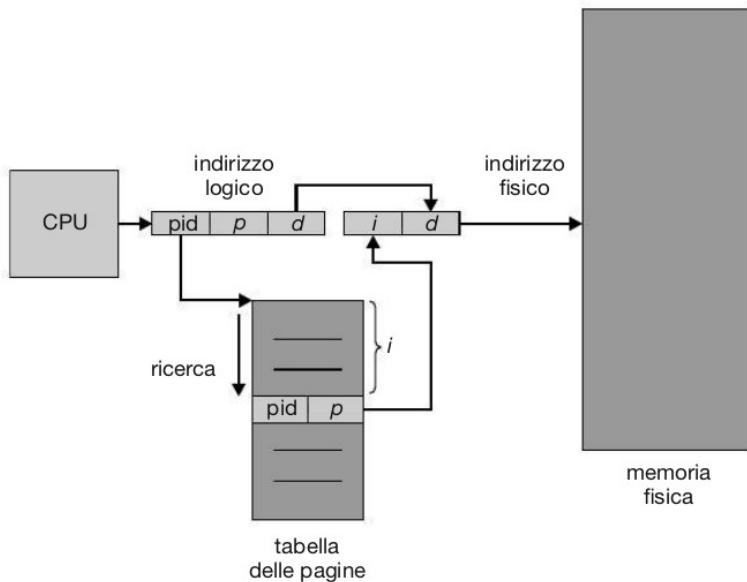


Figura 8.20 Tabella delle pagine invertita.

Vediamo come funziona: ogni processo è identificato da un certo **pid** (*process identifier*), cioè un identificativo univoco dello stesso. Abbiamo dunque una terna: il *pid*, la pagina (*p*) e lo scostamento (*d*). La coppia *pid-p* servirà per cercare l'*i*-esimo elemento.

9.4.4 PAGINE CONDIVISE

- **Codice condiviso:** una copia di codice rientrante condiviso tra i processi (es. editor di testi, compilatori interfacce a finestre). Deve essere collocato nella stessa posizione nello spazio degli indirizzi logici di tutti i processi.

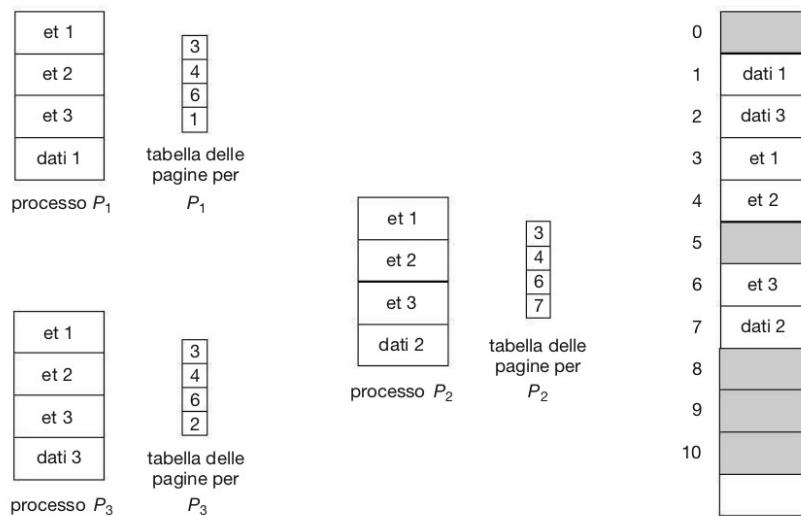


Figura 8.16 Condivisione di codice in un ambiente paginato.

- **Codice privato e dati:** ciascun processo dispone di una propria copia di codice e dati. Le pagine per il codice e i dati possono essere collocate in un qualsiasi punto dello spazio degli indirizzi logici.

9.5 SEGMENTAZIONE

La **segmentazione** è una tecnica di allocazione non contigua dei processi in memoria che differisce dalla paginazione perché i processi sono suddivisi in blocchi, detti **segmenti**, di differente dimensione. Analogamente a quanto succede alle pagine con la paginazione, un processo segmentato è allocato in memoria per segmenti non necessariamente adiacenti.

Con la segmentazione un processo è suddiviso in blocchi secondo criteri logici che riflettono l'organizzazione del software. Per esempio un programma potrebbe essere composto da un certo numero di segmenti secondo la strutturazione dei moduli che lo compongono: un segmento per il programma principale, uno per ogni sottoprogramma, un altro ancora per un blocco di dati e così via. La segmentazione riflette quindi la visione che il programmatore ha del proprio programma (si basa sulla natura dell'oggetto).

Per semplicità di implementazione, i segmenti sono numerati, e ogni riferimento si compie per mezzo di un numero anziché di un nome; quindi un indirizzo logico è una *coppia*:

<numero di segmento, offset>

9.5.1 ARCHITETTURA DI SEGMENTAZIONE

Sebbene il programmatore possa far riferimento agli oggetti del programma per mezzo di un indirizzo bidimensionale, la memoria fisica è in ogni caso una sequenza di byte unidimensionale. per questo motivo occorre tradurre gli indirizzi bidimensionali definiti dall'utente negli indirizzi fisici unidimensionali. Questa operazione si compie tramite una tabella dei segmenti; ogni suo elemento è una coppia ordinata:

- **Base del segmento**: contiene l'indirizzo fisico iniziale della memoria dove il segmento risiede;
- **Limite del segmento**: contiene la lunghezza del segmento.

Nell'architettura di segmentazione sono inoltre presenti un **registro di base della tabella dei segmenti** (*segment-table base register, STBR*) che punta alla tabella dei segmenti in memoria, e il **registro limite della tabella dei segmenti** (*segment-table length register, STLR*) che indica il numero di segmenti usati dal programma. Un numero di segmenti s è valido se $s < STLR$.

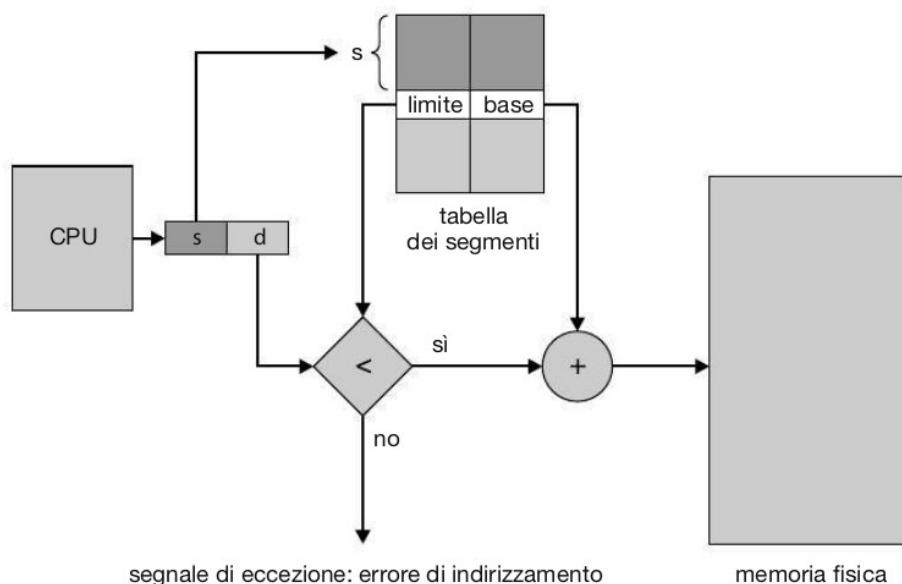


Figura 8.8 Hardware di segmentazione.

Vediamo un esempio per comprendere il concetto. Sono dati cinque segmenti numerati da 0 a 4, posizionati in memoria fisica come indicato. La tabella dei segmenti ha un elemento distinto per ogni segmento, indicante l'indirizzo iniziale del segmento in memoria fisica (la base) e la lunghezza di quel segmento (il limite). per esempio, il segmento 2 è lungo 400 byte e inizia alla locazione 4300, quindi un riferimento al byte 53 del segmento 2 si mappa sulla locazione $4300 + 53 = 4353$. un riferimento al segmento 3, byte 852, si mappa sulla locazione 3200 (la base del segmento 3) + 852 = 4052. un riferimento al byte 1222 del segmento 0 causa la generazione di una eccezione per il sistema operativo, poiché questo segmento è lungo solo 1000 byte.

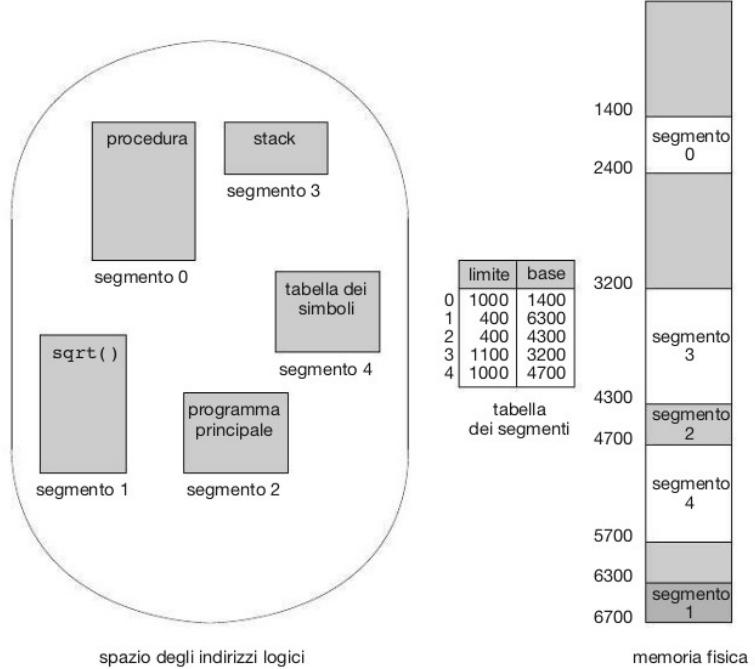


Figura 8.9 Esempio di segmentazione.

10 – MEMORIA VIRTUALE

10.1 INTRODUZIONE

La **memoria virtuale** si fonda sulla separazione della memoria logica percepita dall'utente dalla memoria fisica. Questa separazione permette di offrire ai programmati una memoria virtuale molto ampia, anche se la memoria fisica disponibile è più piccola. La memoria virtuale facilita la programmazione, poiché il programmatore non deve preoccuparsi della quantità di memoria fisica disponibile, ma può concentrarsi sul problema da risolvere con il programma.

I vantaggi della memoria virtuale sono:

- Solo parte del programma deve essere in memoria per l'esecuzione;
- Lo spazio degli indirizzi logici può essere maggiore dello spazio degli indirizzi fisici;
- Permette la condivisione di file e memoria tra diversi processi tramite la condivisione delle pagine;
- Consente anche un miglioramento delle prestazioni durante la creazione dei processi.

La memoria virtuale generalmente si realizza nelal forma di:

- Paginazione su richiesta (*demand paging*);
- Segmentazione su richiesta (*demand segmentation*).

Quello che è stato detto possiamo rappresentarlo in questo modo:

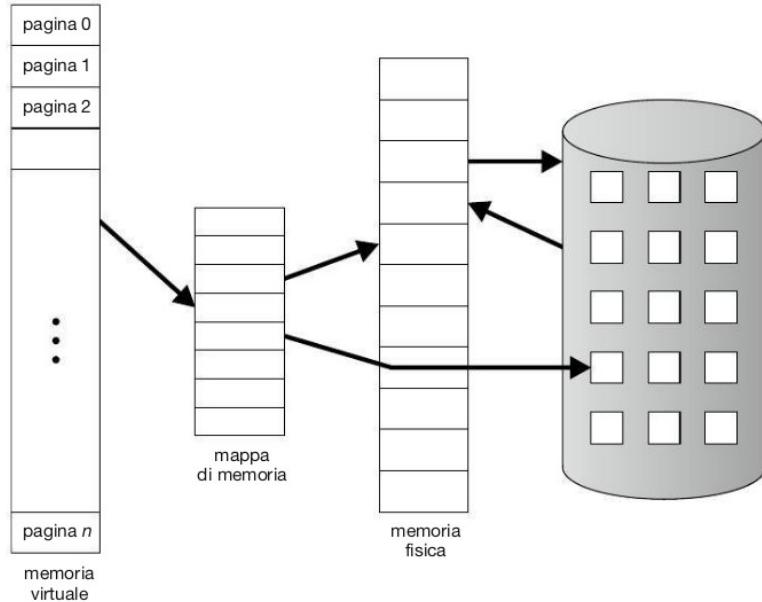


Figura 9.1 Schema che mostra una memoria virtuale più grande di quella fisica.

10.2 PAGINAZIONE SU RICHIESTA

Secondo questo schema, le pagine sono caricate in memoria solo quando richieste durante l'esecuzione del programma: ne consegue che le pagine cui non si accede mai non sono mai caricate nella memoria fisica.

Questo comporta:

- Minore tempo di avvicendamento;
- Minore quantità di memoria fisica;
- Risposta più veloce;
- Più utenti.

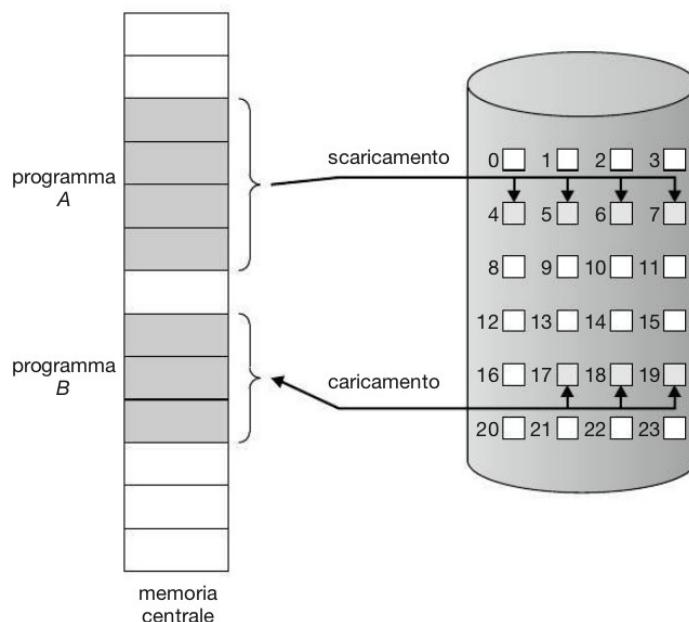


Figura 9.4 Trasferimento di una memoria paginata nello spazio contiguo di un disco.

Con tale schema è necessario che l'hardware fornisca un meccanismo che consenta di distinguere le pagine presenti in memoria da quelle nei dischi. A tal fine è utilizzabile lo schema basato sul bit di validità. In questo caso, però, il bit impostato come "valido" significa che la pagina corrispondente è valida ed è presente in memoria; il bit impostato come "non valido" indica che la pagina non è valida (cioè non appartiene allo spazio d'indirizzi logici del processo) oppure è valida ma è attualmente nel disco.

L'elemento della tabella delle pagine di una pagina caricata in memoria s'imposta come al solito, mentre l'elemento della tabella delle pagine corrispondente a una pagina che attualmente non è in memoria è semplicemente contrassegnato come non valido oppure contiene l'indirizzo della pagina su disco.

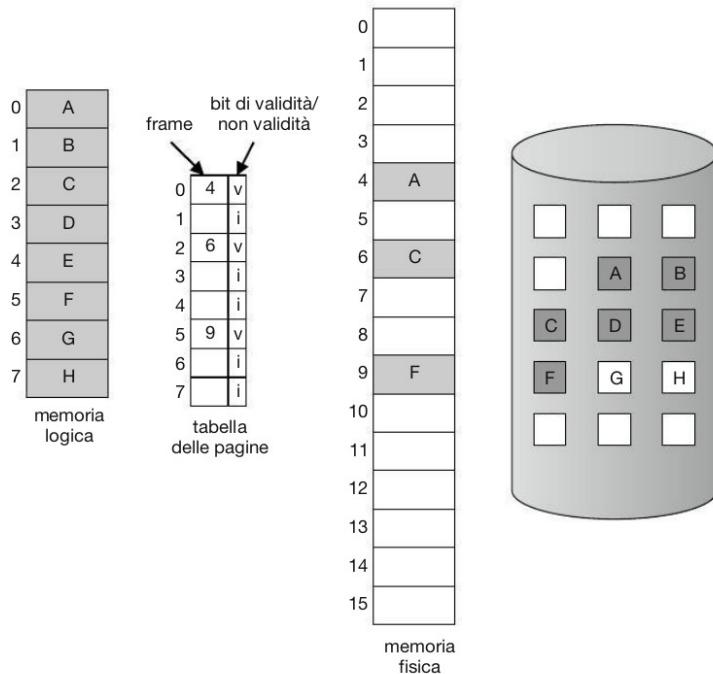


Figura 9.5 Tabella delle pagine quando alcune pagine non si trovano nella memoria centrale.

Che succede se il processo tenta l'accesso a una pagina che non era stata caricata in memoria? L'accesso a una pagina contrassegnata come non valida causa un evento o eccezione di **page fault** (*pagina mancante*). Quindi, l'architettura di paginazione, traducendo l'indirizzo, nota che il bit non è valido e invia un segnale d'eccezione al sistema operativo.

A questo punto si individua un blocco di memoria libero e si trasferisce al suo interno la pagina desiderata. Poi si aggiornano le tabelle, mettendo il bit di validità a 1 e si riavvia l'istruzione che era stata interrotta dal segnale di interruzione.

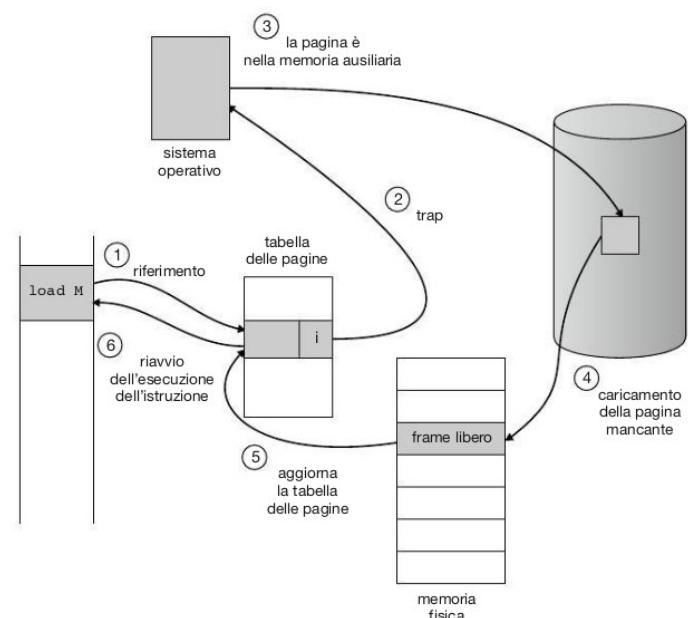


Figura 9.6 Fasi di gestione di un page fault.

10.3 COPIATURA SU SCRITTURA

Il funzionamento si fonda sulla condivisione iniziale delle pagine da parte dei processi genitori e dei processi figli. Le pagine condivise si contrasseggano come pagine da copiare su scrittura, a significare che, se un processo (genitore o figlio) scrive su una pagina condivisa, il sistema deve creare una copia di tale pagina.

Questa tecnica consente una maggiore efficienza poiché solo le pagine modificate sono copiate.

Le pagine libere si trovano in un gruppo (*pool*) di pagine libere che di solito si assegnano quando la pila di un processo deve espandersi oppure proprio per gestire pagine da copiatura su scrittura.

10.4 ASSOCIAZIONE DEI FILE ALLA MEMORIA

L'associazione alla memoria di un file consiste nel permettere che una parte dello spazio degli indirizzi virtuali sia associata logicamente a un file.

L'accesso iniziale al file avviene tramite una normale richiesta di paginazione, che causa un errore di page fault. Tuttavia, una porzione del file, che è pari a una pagina, è caricata dal file system in una pagina fisica (alcuni sistemi possono decidere di caricare porzioni più grandi di memoria). Le operazioni successive di lettura/scrittura sul file si gestiscono come normali accessi alla memoria. (non ho minimamente capito cosa significhi ma lo scrivo lo stesso)

In questo modo si semplifica l'accesso e l'uso dei file, si permette la manipolazione degli stessi attraverso la memoria e si evita il carico dovuto alle chiamate del sistema `read()` e `write()`.

Si può permettere l'associazione dello stesso file alla memoria virtuale di più processi, allo scopo di condividere dati.

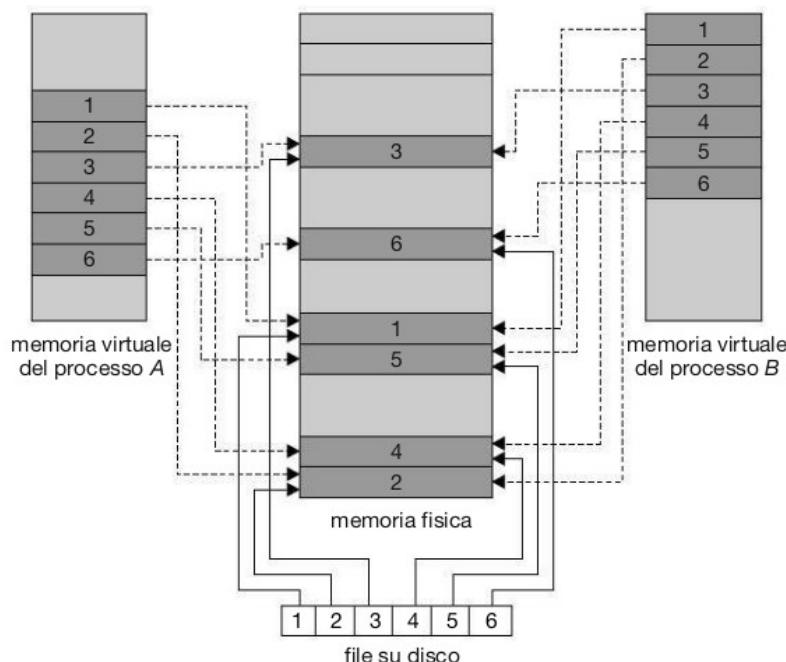


Figura 9.22 File mappati in memoria.

10.5 SOSTITUZIONE DELLE PAGINE

Previene la sovrassegnazione della memoria.

La sostituzione delle pagine segue il seguente criterio: se nessun frame è libero, ne viene liberato uno attualmente inutilizzato. È possibile liberarlo scrivendo il suo contenuto nell'area di swap e modificando la tabella delle pagine (e tutte le altre tabelle) per indicare che la pagina non si trova più in memoria. Il frame liberato si può usare per memorizzare la pagina che ha causato il fault. Si modifica la procedura di servizio dell'eccezione di page fault in modo da includere la sostituzione della pagina:

- s'individua la locazione su disco della pagina richiesta;
- si cerca un frame libero:
 - se esiste, lo si usa;
 - altrimenti si impiega un algoritmo di sostituzione delle pagine per scegliere un **frame vittima**;
 - si scrive la pagina "vittima" nel disco; si di conseguenza le tabelle delle pagine e quelle dei frame;
- si scrive la pagina richiesta nel frame appena liberato; si modificano le tabelle delle pagine e dei frame;
- si riprende il processo utente dal punto in cui si è verificato il page fault.

Occorre notare che, se non esiste alcun frame libero sono necessari due trasferimenti di pagine, uno fuori e uno dentro la memoria. Questa situazione raddoppia il tempo di servizio del page fault e aumenta di conseguenza anche il tempo effettivo d'accesso. Questo sovraccarico si può ridurre usando un **bit di modifica** (*modify bit o dirty bit*). In questo caso l'hardware del calcolatore dispone di un bit di modifica, associato a ogni pagina (o frame), che viene posto a 1 ogni volta che nella pagina si scrive un byte, indicando che la pagina è stata modificata. Quando si sceglie una pagina da sostituire si esamina il suo bit di modifica; se è a 1, significa che quella pagina è stata modificata rispetto a quando era stata letta dal disco; in questo caso la pagina deve essere scritta nel disco. Se il bit di modifica è rimasto a 0, significa che la pagina non è stata modificata da quando è stata caricata in memoria, quindi non è necessario scrivere nel disco la pagina di memoria: c'è già.

La sostituzione delle pagine completa la separazione tra memoria logica e memoria fisica.

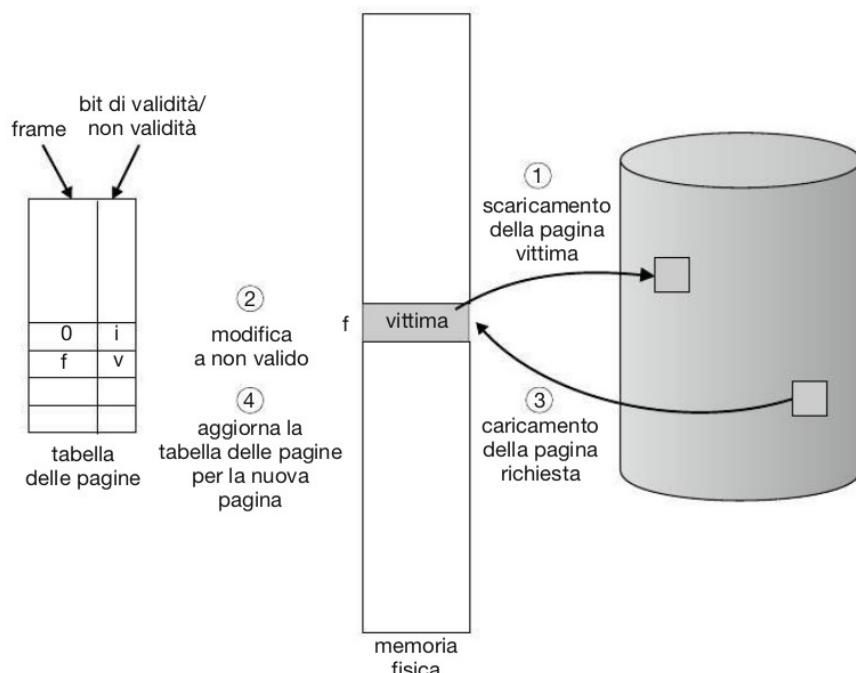


Figura 9.10 Sostituzione di una pagina.

10.6 ALGORITMI DI SOSTITUZIONE DELLE PAGINE

L'obiettivo principale è quello di ottenere la minor frequenza di assenza delle pagine.

L'algoritmo si valuta su una particolare successione (stringa) di riferimenti (**reference string**) e si calcola il numero di assenze di pagine su quella successione. In tutti i nostri esempi la stringa di riferimento è:

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

Esiste un preciso rapporto tra il numero di assenze di pagine (*page fault*) e l'ampiezza dello spazio disponibile (numero di frame): più c'è spazio e più non dobbiamo preoccuparci di togliere e mettere qualcosa.

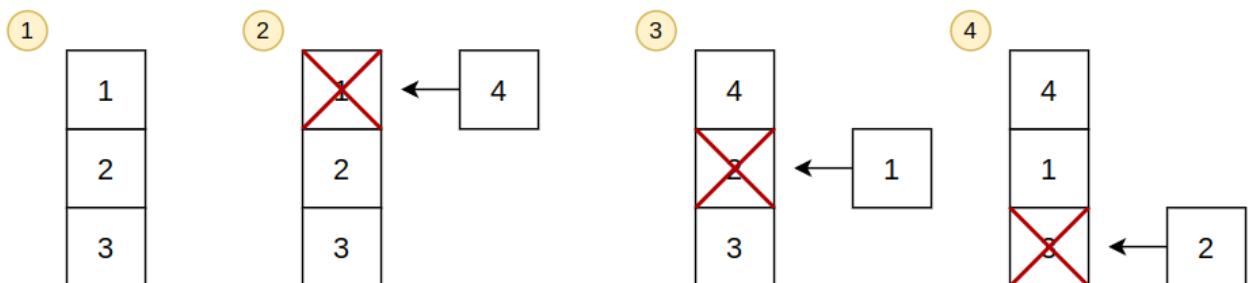
10.6.1 ALGORITMO FIFO (FIRST-IN-FIRST-OUT)

L'algoritmo di sostituzione delle pagine più semplice è un algoritmo FIFO. Questo algoritmo associa a ogni pagina l'istante di tempo in cui quella pagina è stata portata in memoria. Se si deve sostituire una pagina, si seleziona quella presente in memoria da più tempo.

Vediamo un esempio pratico per capirne il concetto. Usiamo la stringa di riferimento mostrata sopra:

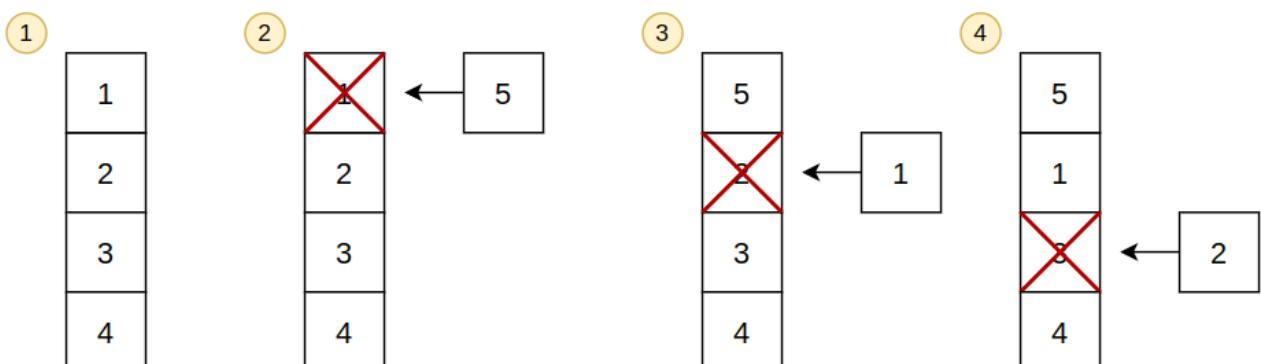
1, 2, 3, 4, 1, 2, 5, 1, 2, 3, 4, 5

Supponiamo di mettere a disposizione 3 blocchi di memoria. Ciò che succederà sarà questo:



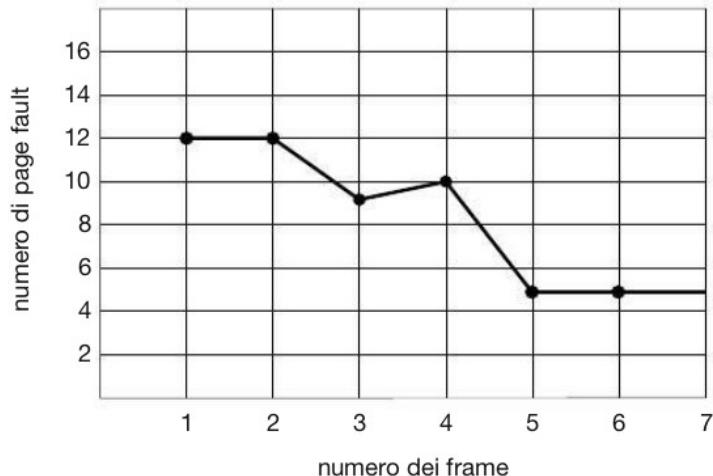
E così via fino a quando non si finirà la sequenza. Alla fine ci ritroveremo con **9 assenze di pagine**.

Supponiamo invece adesso di avere 4 blocchi di memoria:



E così via fino a quando non si finirà la sequenza. Alla fine ci ritroveremo con **10 assenze di pagina**.

Occorre notare che il numero dei page fault (10) per quattro frame è maggiore del numero dei page fault (9) per tre frame. Questo inatteso risultato è noto col nome di **anomalia di Belady**: con alcuni algoritmi di sostituzione delle pagine, il tasso di page fault può aumentare con l'incremento del numero minimo di page fault aumentare del numero dei frame assegnati.



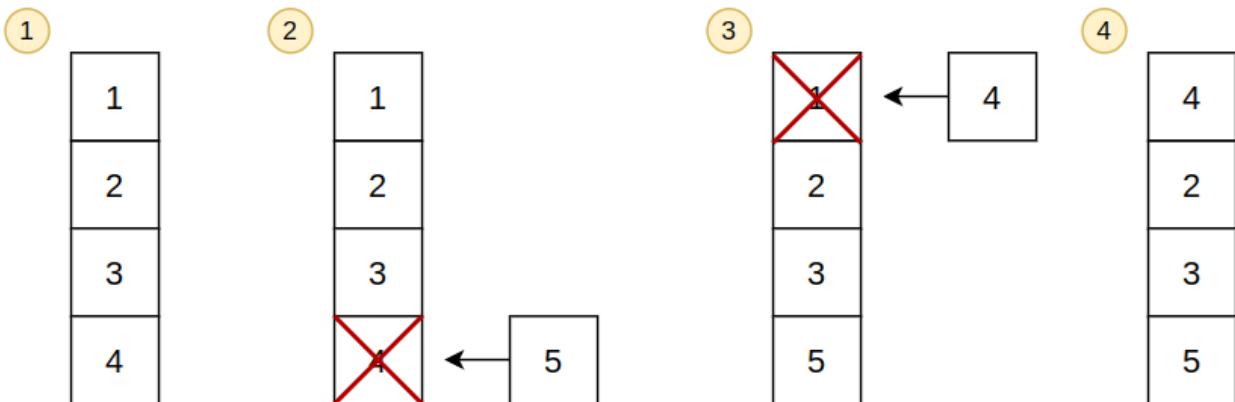
10.6.2 SOSTITUZIONE OTTIMALE DELLE PAGINE

In seguito alla scoperta dell'anomalia di Belady si è ricercato un **algoritmo ottimale di sostituzione delle pagine**. Tale algoritmo è quello che fra tutti gli algoritmi presenta il tasso minimo di page fault e non presenta mai l'anomalia di Belady. Questo algoritmo esiste ed è stato chiamato OPT o MIN. Consiste semplicemente nel: sostituire la pagina che non verrà usata per il periodo di tempo più lungo.

Supponiamo di avere 4 blocchi di memoria e la sequenza di riferimento precedente:

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

A questo punto avremo:



Alla fine del processo, avremo **6 assenze di pagina**.

Sfortunatamente l'algoritmo ottimale di sostituzione delle pagine è difficile da realizzare, perché richiede la conoscenza futura della successione dei riferimenti (una situazione analoga si è riscontrata con l'algoritmo SJF di scheduling della CPU). Quindi, l'algoritmo ottimale si impiega soprattutto per studi comparativi.

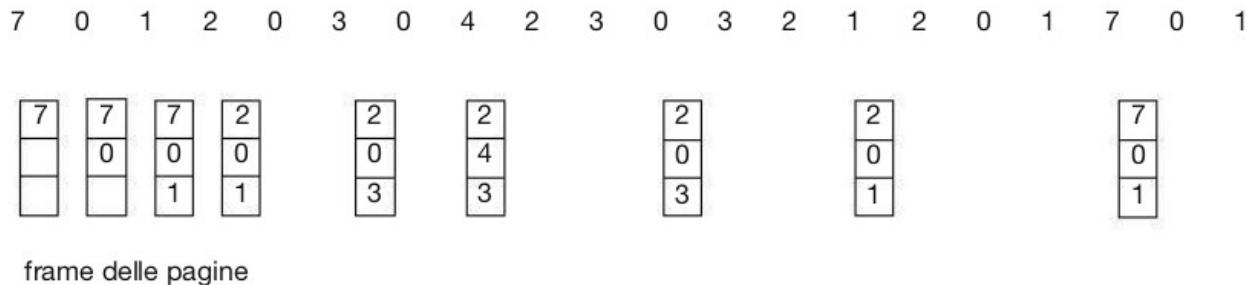
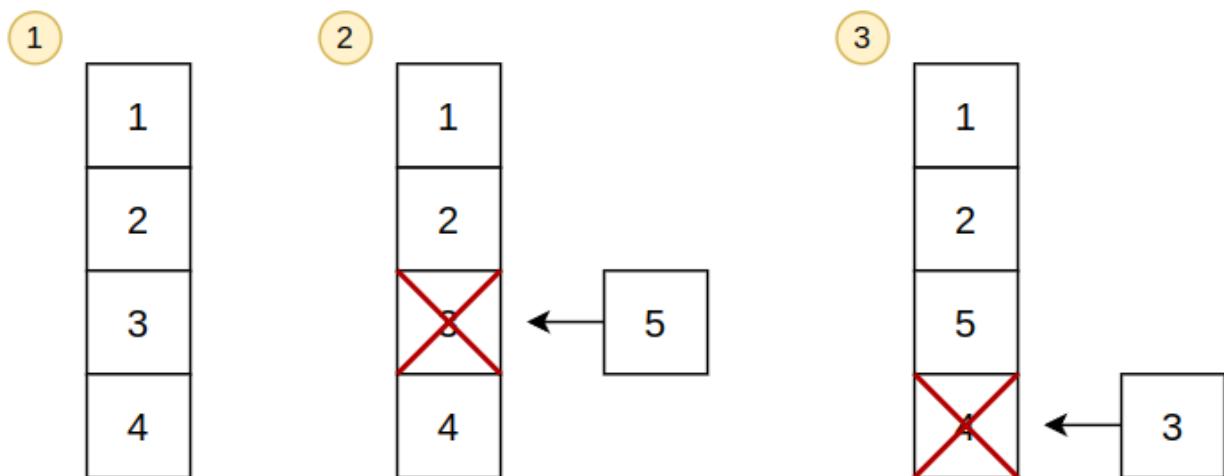


Figura 9.14 Algoritmo ottimale di sostituzione delle pagine.

10.6.3 SOSTITUZIONE DELLE PAGINE USATE MENO RECENTEMENTE (LRU)

Se l'algoritmo ottimale non è realizzabile, è forse possibile realizzarne un'approssimazione. La distinzione fondamentale tra gli algoritmi FIFO e OPT, oltre quella di guardare avanti o indietro nel tempo, consiste nel fatto che l'algoritmo FIFO impiega l'istante in cui una pagina è stata caricata in memoria, mentre l'algoritmo OPT impiega l'istante in cui una pagina sarà usata. Usando come approssimazione di un futuro vicino un passato recente, si sostituisce la pagina che non è stata usata per il periodo più lungo. Il metodo appena descritto è noto come **algoritmo LRU** (*least recently used*).



E così via fino alla fine della sequenza.

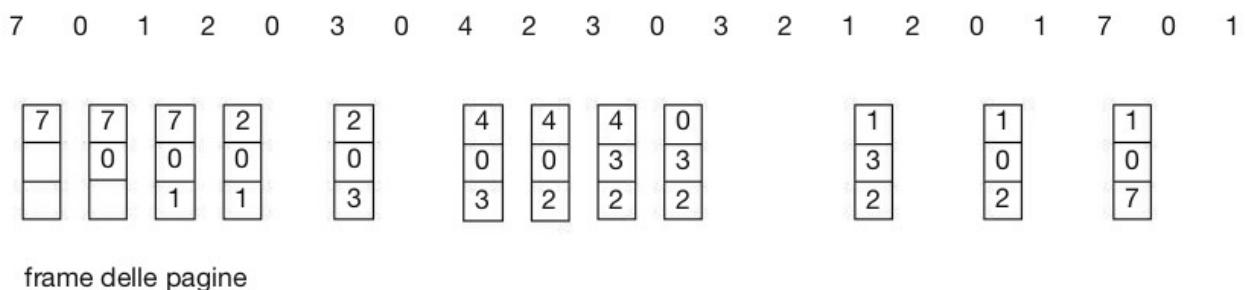


Figura 9.15 Algoritmo di sostituzione delle pagine LRU.

Il problema principale riguarda la sua implementazione. Un algoritmo di sostituzione delle pagine LRU può richiedere una notevole assistenza da parte dell'hardware. Il problema consiste nel determinare un ordine per i frame definito dal momento dell'ultimo uso. Si possono realizzare le due seguenti soluzioni:

- **Contatori**: Nel caso più semplice, a ogni elemento della tabella delle pagine si associa un campo *momento di utilizzo*, e alla CPU si aggiunge un contatore che si incrementa a ogni riferimento alla memoria. ogni volta che si fa un riferimento a una pagina, si copia il contenuto del registro contatore nel campo *momento di utilizzo* nella voce della page table relativa a quella pagina. In questo modo è sempre possibile conoscere il momento in cui è stato fatto l'ultimo riferimento a ogni pagina. Si sostituisce la pagina con il valore associato più piccolo;
 - **Stack**: ogni volta che si fa un riferimento a una pagina, la si estrae dallo stack e la si colloca in cima a quest'ultimo. In questo modo, in cima allo stack si trova sempre la pagina usata per ultima, mentre in fondo si trova la pagina usata meno recentemente. Poiché gli elementi si devono estrarre dal centro dello stack, la migliore realizzazione si ottiene usando una lista doppiamente concatenata, con un puntatore all'elemento iniziale e uno a quello finale. Per estrarre una pagina dallo stack e collocarla in cima, nel caso peggiore è necessario modificare sei puntatori. Ogni aggiornamento è un po' più costoso, ma per una sostituzione non si deve compiere alcuna ricerca: il puntatore dell'elemento di coda punta alla pagina LRU.

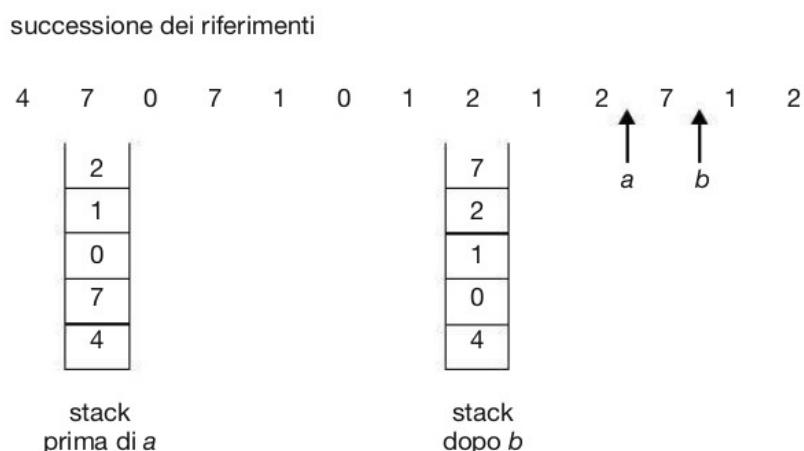


Figura 9.16 Uso di uno stack per registrare i più recenti riferimenti alle pagine.

10.7 SOSTITUZIONE DELLE PAGINE PER APPROXIMAZIONE A LRU

Sono pochi i sistemi di calcolo che dispongono del supporto hardware per una vera sostituzione LRU delle pagine. Nei sistemi che non offrono alcun supporto hardware si devono impiegare altri algoritmi di sostituzione delle pagine, per esempio l'algoritmo FIFO. Molti sistemi tuttavia possono fornire un aiuto: un **bit di riferimento**. Il bit di riferimento a una pagina è impostato automaticamente dall'hardware del sistema ogni volta che si fa un riferimento a quella pagina, che sia una lettura o una scrittura su qualsiasi byte della pagina. I bit di riferimento sono associati a ciascun elemento della tabella delle pagine.

All'inizio, il sistema operativo azzera tutti i bit. Quando s'inizia l'esecuzione di un processo utente, l'hardware imposta a 1 il bit associato a ciascuna pagina cui si fa riferimento. Dopo qualche tempo è possibile stabilire quali pagine sono state usate semplicemente esaminando i bit di riferimento. Non è però possibile conoscere l'ordine d'uso.

10.7.1 ALGORITMO CON SECONDA CHANCE

L'algoritmo di base per la sostituzione con seconda chance è un algoritmo di sostituzione di tipo FIFO. Tuttavia, dopo aver selezionato una pagina, si controlla il bit di riferimento: se il suo valore è 0, si sostituisce la pagina; se il bit di riferimento è impostato a 1, si dà una seconda chance alla pagina e si passa alla successiva pagina FIFO.

Quando una pagina riceve la seconda chance, si azzera il suo bit di riferimento e si aggiorna il suo istante d'arrivo al momento attuale. In questo modo, una pagina cui si offre una seconda chance non viene mai sostituita finché tutte le altre pagine non siano state sostituite, oppure non sia stata data loro una seconda chance. Inoltre, se una pagina è usata abbastanza spesso, da mantenere il suo bit di riferimento impostato a 1, non viene mai sostituita.

Un metodo per implementare l'algoritmo con seconda chance, detto anche a orologio (*clock*), è basato sull'uso di una coda circolare, in cui un puntatore (lancetta) indica qual è la prima pagina da sostituire. Quando serve un frame, si fa avanzare il puntatore finché non si trovi in corrispondenza di una pagina con il bit di riferimento 0; a ogni passo si azzera il bit di riferimento appena esaminato. Una volta trovata una pagina "vittima", la si sostituisce e si inserisce la nuova pagina nella coda circolare nella posizione corrispondente.

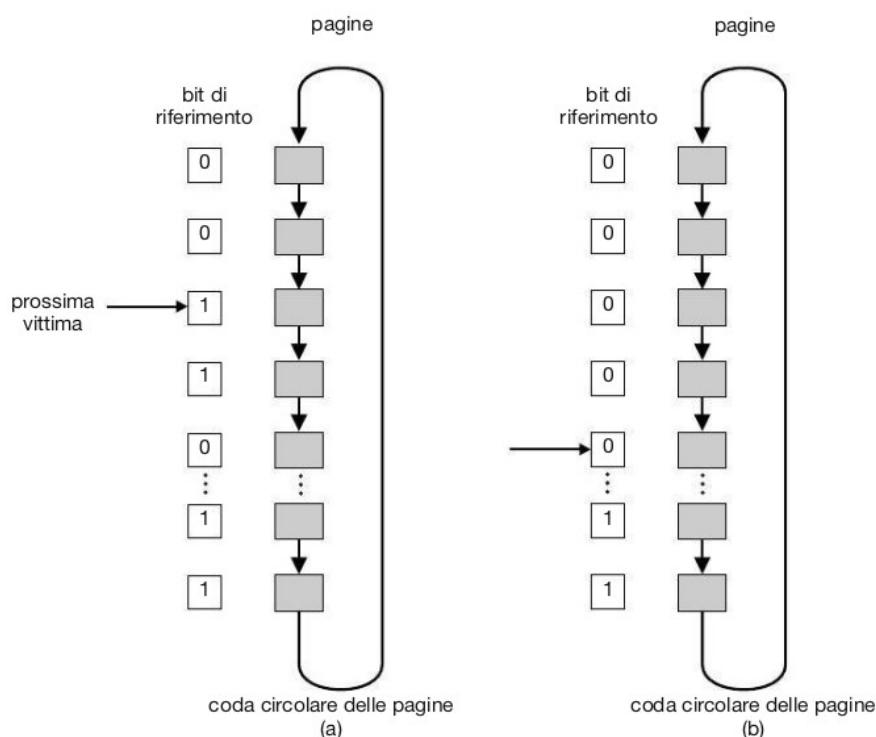


Figura 9.17 Algoritmo di sostituzione delle pagine con seconda chance (orologio).

Si noti che nel caso peggiore, quando tutti i bit sono impostati a 1, il puntatore percorre un ciclo su tutta la coda, dando a ogni pagina una seconda chance. Prima di selezionare la pagina da sostituire, azzera tutti i bit di riferimento. Se tutti i bit sono a 1, la sostituzione con seconda chance si riduce a una sostituzione FIFO.

10.7.2 SOSTITUZIONE DELLE PAGINE BASATA SU CONTEGGIO

Esistono molti altri algoritmi che si possono usare per la sostituzione delle pagine. Per esempio, si potrebbe usare un contatore del numero dei riferimenti fatti a ciascuna pagina, e sviluppare i due seguenti schemi:

- **Algoritmo di sostituzione delle pagine meno frequentemente usate (*least frequently used, LFU*)**: richiede che si sostituisca la pagina con il conteggio più basso. La ragione di questa scelta è che una pagina usata attivamente deve avere un conteggio di riferimento alto. Si ha però un problema quando una pagina è usata molto intensamente durante la fase iniziale di un processo, ma poi non viene più usata. Poiché è stata usata intensamente il suo conteggio è alto, quindi rimane in memoria anche se non è più necessaria. Una soluzione può essere quella di spostare i valori dei contatori a destra di un bit a intervalli regolari, misurando l'utilizzo con un peso esponenziale decrescente.
- **Algoritmo di sostituzione delle pagine più frequentemente usate (*most frequently used, MFU*)**: è basato sul fatto che, probabilmente, la pagina con il contatore più basso è stata appena inserita e non è stata ancora usata.

Le sostituzioni MFU e LFU non sono molto comuni, poiché la realizzazione di questi algoritmi è abbastanza onerosa; inoltre, tali algoritmi non approssimano bene la sostituzione OPT.

11 – INTERFACCIA DEL FILE SYSTEM

11.1 CONCETTO DI FILE

Un **file** è un insieme di informazioni correlate, registrate in memoria secondaria, cui è stato assegnato un nome. Dal punto di vista dell'utente, un file è la più piccola porzione di memoria logica secondaria; i dati si possono cioè scrivere in memoria secondaria soltanto all'interno di un file.

I file di dati possono essere numerici, alfabetici, alfanumerici o binari. I file possono non possedere un formato specifico, come i file di testo, oppure essere rigidamente formattati.

Un file presenta una certa struttura. Ci sono:

- **Strutture semplici**: righe, lunghezze fissa, lunghezza variabile.
- **Strutture complesse**: Documento formattato, relocatable load file.
- Oppure non avere **nessuna struttura**: sequenza di parole, byte.

11.1.1 ATTRIBUTI DEI FILE

Un file ha attributi che possono variare secondo il sistema operativo, ma che tipicamente comprendono i seguenti:

- **Nome**: Il nome simbolico del file è l'unica informazione in forma umanamente leggibile.
- **Identificatore**: Si tratta di un'etichetta unica, di solito un numero, che identifica il file all'interno del file system; è il nome impiegato dal sistema per il file.
- **Tipo**: Questa informazione è necessaria ai sistemi che gestiscono tipi di file diversi.
- **Locazione**: Si tratta di un puntatore al dispositivo e alla locazione del file in tale dispositivo.
- **Dimensione**: Si tratta della dimensione corrente del file (in byte, parole o blocchi) ed eventualmente della massima dimensione consentita.
- **Protezione**: Le informazioni di controllo degli accessi controllano chi può leggere,

scrivere o eseguire il file.

- **Ora, data e identificazione dell'utente:** Queste informazioni possono essere relative alla creazione, l'ultima modifica e l'ultimo uso. Questi dati possono essere utili ai fini della protezione, della sicurezza e del monitoraggio del suo utilizzo.

Le informazioni sui file sono conservate nella struttura della directory, che risiede a sua volta in memoria secondaria. Di solito un elemento di directory consiste di un nome di file e di un identificatore unico, che a sua volta individua gli altri attributi del file.

11.1.2 OPERAZIONI SUI FILE

Un file è un **tipo di dato astratto**. Per definire adeguatamente un file è necessario considerare le operazioni che si possono eseguire su di esso. Il sistema operativo può offrire chiamate di sistema per creare, scrivere, leggere, spostare, cancellare e troncare un file. Esaminiamo ciò che deve fare il sistema operativo per ciascuna di queste operazioni di base:

- **Creazione di un file:** Per creare un file è necessario compiere due passaggi. In primo luogo si deve trovare lo spazio per il file nel file system e poi si deve creare un nuovo elemento nella directory.
- **Scrittura di un file:** Per scrivere in un file viene effettuata una chiamata di sistema che specifica il nome del file e le informazioni che si vogliono scrivere. Dato il nome del file, il sistema ricerca la directory per individuare la posizione del file. Il file system deve mantenere un *puntatore di scrittura* alla locazione nel file in cui deve avvenire l'operazione di scrittura successiva. Il puntatore si deve aggiornare ogniqualvolta si esegue una scrittura.
- **Lettura di un file:** Per leggere da un file è necessaria una chiamata di sistema che specifichi il nome del file e la posizione in memoria dove collocare il blocco del file da leggere. Anche in questo caso si cerca l'elemento corrispondente nella directory e il sistema deve mantenere un puntatore di lettura alla locazione nel file in cui deve avvenire la successiva operazione di lettura. Una volta completata la lettura, si aggiorna il puntatore. Di solito un processo o legge o scrive in un file, e la posizione corrente è mantenuta come un puntatore alla posizione corrente del file specifico del processo. Sia le operazioni di lettura sia quelle di scrittura adoperano lo stesso puntatore, risparmiando spazio e riducendo la complessità del sistema.
- **Riposizionamento di un file:** Si ricerca l'elemento appropriato nella directory e si assegna un nuovo valore al puntatore alla posizione corrente nel file. Il riposizionamento non richiede alcuna operazione di I/O. Questa operazione è anche nota come *posizionamento* o *ricerca (seek)* nel file.
- **Cancellazione di un file:** Per cancellare un file si cerca l'elemento della directory associato al file designato, si rilascia lo spazio associato al file (in modo che possa essere adoperato per altri) e si elimina l'elemento della directory.
- **Troncamento di un file:** Si potrebbe voler cancellare il contenuto di un file, ma mantenere i suoi attributi. Invece di forzare gli utenti a cancellare il file e quindi ricrearlo, questa funzione consente di mantenere immutati gli attributi (a esclusione della lunghezza del file) pur azzerando la lunghezza del file e rilasciando lo spazio occupato.

La maggior parte delle operazioni sopra citate richiede una ricerca nella directory dell'elemento associato al file specificato. Per evitare questa continua ricerca, molti sistemi richiedono l'impiego di una chiamata di sistema `open()` prima che un file venga utilizzato. Quando il file non è più attivamente usato viene *chiuso* (`delete()`) dal processo, e il sistema operativo rimuove l'elemento a esso associato dalla tabella dei file aperti.

11.2 METODI DI ACCESSO

I metodi di accesso sono due:

- **Accesso sequenziale**: le informazioni del file si elaborano ordinatamente, un record dopo l'altro; questo metodo d'accesso è di gran lunga il più comune, ed è usato, per esempio, dagli editor e dai compilatori. Se n è uguale al numero relativo del blocco:

```
read next  
write next  
reset  
no read after last write  
(rewrite)
```

- **Accesso diretto**: In questo caso, un file è formato da elementi logici (record) di lunghezza fissa; ciò consente ai programmi di leggere e scrivere rapidamente tali elementi senza un ordine particolare. Il metodo ad accesso diretto si fonda su un modello di file che si rifà al disco: i dischi permettono, infatti, l'accesso diretto a ogni blocco di file. Se n è uguale al numero relativo del blocco:

```
read n  
write n  
position to n  
    read next  
    write next  
rewrite n
```



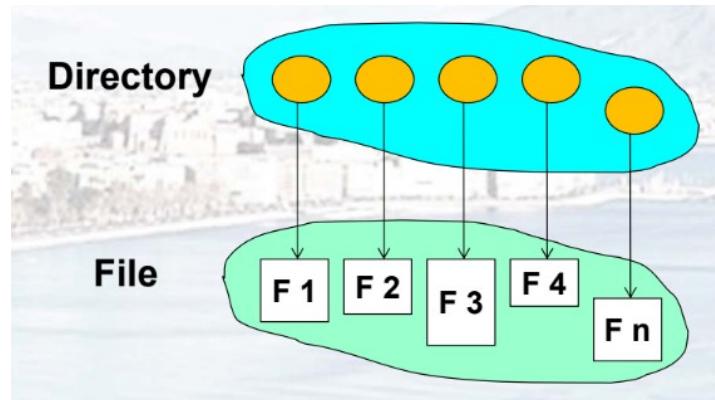
Figura 11.4 File ad accesso sequenziale.

Accesso sequenziale	Realizzazione nel caso di accesso diretto
reset	<code>cp = 0;</code>
<code>read_next()</code>	<code>read cp; cp = cp + 1;</code>
<code>write_next()</code>	<code>write cp; cp = cp + 1;</code>

Figura 11.5 Simulazione dell'accesso sequenziale a un file ad accesso diretto.

11.3 STRUTTURA DELLA DIRECTORY E DEL DISCO

Un insieme di nodi contenenti informazioni su tutti i file.



Sia la directory sia i file risiedono sul disco. Le copie di backup di queste due strutture vengono in generale archiviate su nastro.

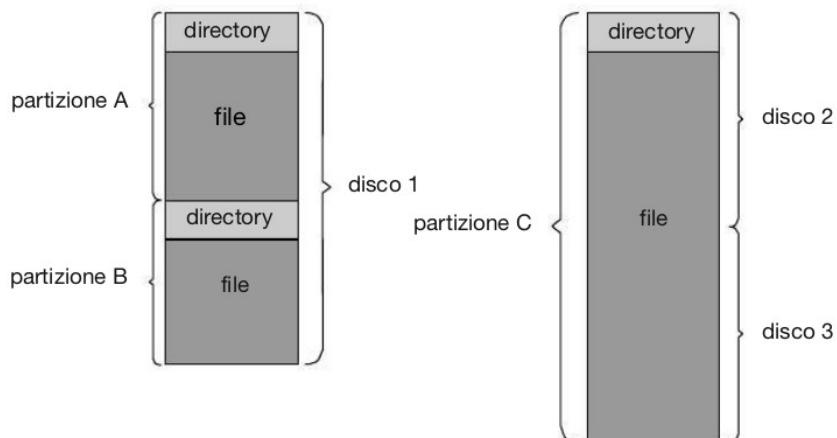


Figura 11.7 Tipica organizzazione di un file system.

I file nella directory hanno le seguenti informazioni:

- Nome;
- Tipo;
- Indirizzo;
- Dimensione corrente;
- Dimensione massima;
- Data dell'ultimo accesso;
- Data dell'ultimo aggiornamento;
- ID del proprietario;
- Informazioni di protezione.

Nel considerare una particolare struttura della directory si deve tenere presente l'insieme delle seguenti operazioni che si possono eseguire su una directory:

- **Ricerca di un file:** Deve esserci la possibilità di scorrere una directory per individuare l'elemento associato a un particolare file. Poiché i file possono avere nomi simbolici, e poiché nomi simili possono indicare relazioni tra file, deve esistere la possibilità di trovare tutti i file il cui nome soddisfi una particolare espressione.
- **Creazione di un file:** Deve essere possibile creare nuovi file e aggiungerli alla directory.
- **Cancellazione di un file:** Quando non serve più, si deve poter rimuovere un file

dalla directory.

- **Elencazione di una directory:** Deve esistere la possibilità di elencare tutti i file di una directory, e il contenuto degli elementi della directory associati a ciascun file nell'elenco.
- **Ridenominazione di un file:** Poiché il nome di un file rappresenta per i suoi utenti il contenuto del file, questo nome deve poter essere modificato quando il contenuto o l'uso del file subiscono cambiamenti. La ridenominazione di un file potrebbe anche permettere la variazione della posizione del file nella directory.
- **Attraversamento del file system:** Si potrebbe voler accedere a ogni directory e a ciascun file contenuto in una directory. Per motivi di affidabilità, è opportuno salvare il contenuto e la struttura dell'intero file system a intervalli regolari.

11.3.1 DIRECTORY A UN LIVELLO

La struttura più semplice per una directory è quella a un livello. Tutti i file sono contenuti nella stessa directory, facilmente gestibile e comprensibile:

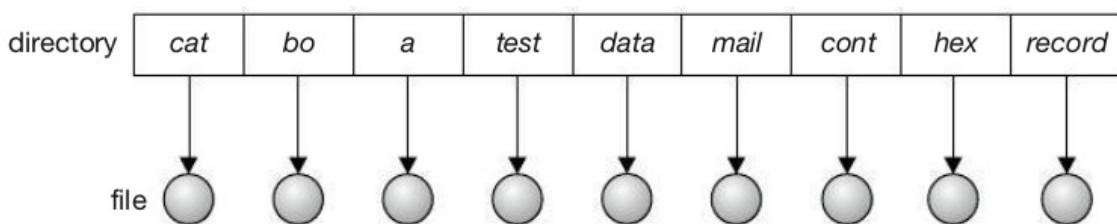


Figura 11.9 Directory a livello singolo.

Una directory a un livello presenta però limiti notevoli quando aumenta il numero dei file in essa contenuti, oppure se il sistema è usato da più utenti. Poiché si trovano tutti nella stessa directory, i file devono avere nomi unici; se due utenti attribuiscono lo stesso nome al loro file di dati, per esempio `test.txt`, si viola la regola del nome unico.

Anche per un solo utente, con una directory a un livello, diventa difficile ricordare i nomi dei file con l'aumentare del loro numero. Non è affatto raro che un utente abbia centinaia di file in un calcolatore e altrettanti file in un altro sistema. In un tale ambiente, sarebbe un compito arduo dover ricordare tanti nomi di file.

11.3.2 DIRECTORY A DUE LIVELLI

Come abbiamo visto, una directory a un livello spesso causa la confusione dei nomi dei file tra diversi utenti. La soluzione più ovvia prevede la creazione di una directory separata per ogni utente.

Nella struttura a due livelli, ogni utente dispone della propria **directory utente** (*userfile directory*, **UFD**). Quando comincia l'elaborazione di un lavoro dell'utente, oppure un utente inizia una sessione di lavoro, si fa una ricerca nella **directory principale** (*master file directory*, **MFD**) del sistema. La directory principale viene indicizzata con il nome dell'utente o il numero di account, e ogni suo elemento punta alla relativa directory utente.

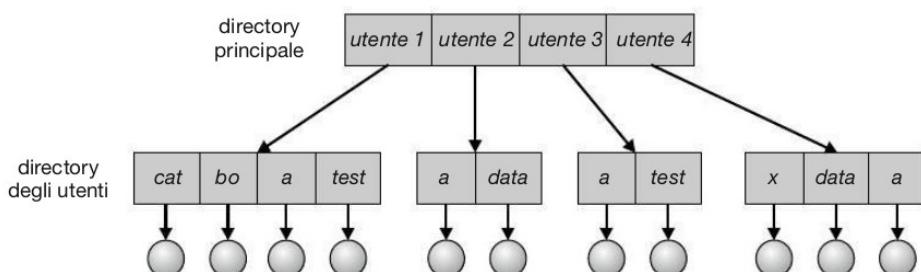


Figura 11.10 Struttura della directory a due livelli.

Quando un utente fa un riferimento a un file particolare, il sistema operativo esegue la ricerca solo nella directory di quell'utente. In questo modo utenti diversi possono avere file con lo stesso nome, purché tutti i nomi di file all'interno di ciascuna directory utente siano unici.

Sebbene risolva la questione delle collisioni dei nomi, la struttura della directory a due livelli presenta ancora dei problemi. In effetti, questa struttura isola un utente dagli altri. Questo isolamento può essere un vantaggio quando gli utenti sono completamente indipendenti, ma è uno svantaggio quando più utenti vogliono cooperare e accedere ai rispettivi file. Alcuni sistemi non permettono l'accesso a file di utenti locali da parte di altri utenti.

Una directory a due livelli si può pensare come un albero di altezza 2. La radice dell'albero è la directory principale, i suoi diretti discendenti sono le directory utente, da cui discendono i file che sono le foglie dell'albero. Specificando un nome utente e un nome di file si definisce un percorso che parte dalla radice (la directory principale) e arriva a una specifica foglia (il file specificato). Quindi, un nome utente e un nome di file definiscono un *nome di percorso* (*path name*).

11.3.3 DIRECTORY CON STRUTTURA AD ALBERO

La corrispondenza strutturale tra directory a due livelli e albero a due livelli può essere facilmente generalizzata, estendendo la struttura della directory a un albero di altezza arbitraria. Questa generalizzazione permette agli utenti di creare proprie sottodirectory e di organizzare i file di conseguenza. Un albero è il più comune tipo di struttura delle directory. L'albero ha una directory radice (*root directory*), e ogni file del sistema ha un unico nome di percorso.

Normalmente, ogni utente dispone di una directory corrente. La **directory corrente** dovrebbe contenere la maggior parte dei file di interesse corrente per il processo. Quando si fa un riferimento a un file, si esegue una ricerca nella directory corrente; se il file non si trova in tale directory, l'utente deve specificare un nome di percorso oppure cambiare la directory corrente facendo diventare tale la directory contenente il file desiderato.

I nomi di percorso possono essere di due tipi: **nomi di percorso assoluti** e **nomi di percorso relativi**. Un *nome di percorso assoluto* comincia dalla radice dell'albero di directory e segue un percorso che lo porta fino al file specificato indicando i nomi delle directory lungo il percorso. Un *nome di percorso relativo* definisce un percorso che parte dalla directory corrente.

Una decisione importante relativa alla strutturazione ad albero delle directory riguarda il modo di gestire la cancellazione di una directory. Se una directory è vuota, è sufficiente cancellare l'elemento che la designa nella directory che la contiene. Tuttavia se la directory da cancellare non è vuota, ma contiene file oppure sottodirectory, è possibile procedere in due modi. Alcuni sistemi non cancellano una directory a meno che non sia vuota; per cancellarla l'utente deve prima cancellare i file in essa contenuti. Se esiste qualche sottodirectory, questa procedura si deve applicare anche alle sottodirectory. Questo metodo può richiedere una discreta quantità di lavoro. In alternativa, come nel comando `rm` di UNIX, si può avere un'opzione che, alla richiesta di cancellazione di una directory, cancelli anche tutti i file e tutte le sottodirectory in essa contenuti.

Il secondo criterio è più comodo, anche se più pericoloso, poiché si può rimuovere un'intera struttura della directory con un solo comando. Se si eseguisse tale comando per sbaglio sarebbe necessario ripristinare un gran numero di file e directory dalle copie di riserva (ipotizzandone l'esistenza).

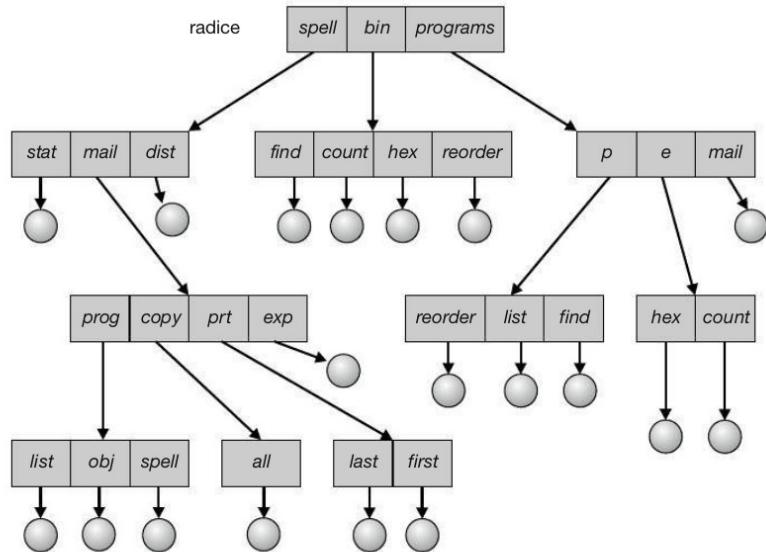


Figura 11.11 Struttura della directory ad albero.

11.3.4 DIRECTORY CON STRUTTURA A GRAFO ACICLICO

La struttura ad albero non ammette la condivisione di file o directory. Un **grafo aciclico** (cioè senza cicli) permette alle directory di avere sottodirectory e file condivisi. Lo stesso file o la stessa sottodirectory possono essere in due directory diverse.

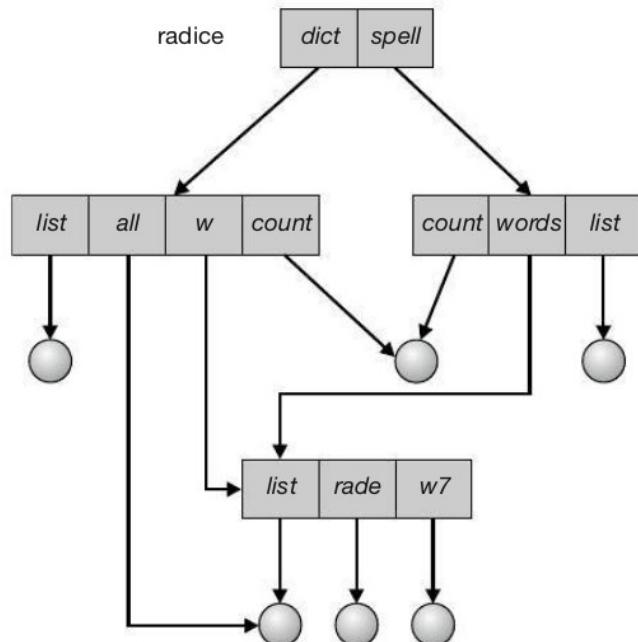


Figura 11.12 Struttura della directory a grafo aciclico.

I file e le sottodirectory condivisi si possono realizzare in molti modi. Un metodo diffuso, esemplificato da molti tra i sistemi UNIX, prevede la creazione di un nuovo elemento di directory, chiamato collegamento. Un **collegamento** (*link*) è un puntatore a un altro file o un'altra directory. Un altro comune metodo per la realizzazione dei file condivisi prevede semplicemente la duplicazione di tutte le informazioni relative ai file in entrambe le directory che lo condividono: i due elementi delle directory sono identici. Si consideri la differenza fra i due approcci. Un collegamento è chiaramente diverso dall'elemento originale della directory. Duplicando gli elementi della directory, invece, la copia e l'originale sono resi indistinguibili: sorge allora il problema di mantenere la coerenza se il file viene modificato.

Una struttura di directory a grafo aciclico è più flessibile di una semplice struttura ad albero, ma anche più complessa. Si devono prendere in considerazione parecchi problemi. Un file può avere più nomi di percorso assoluti, quindi nomi diversi possono riferirsi allo stesso file.

Un altro problema riguarda la cancellazione, poiché è necessario stabilire in quali casi è possibile allocare e riutilizzare lo spazio allocato a un file condiviso. Una possibilità prevede che a ogni operazione di cancellazione seguva l'immediata rimozione del file; quest'azione può però lasciare puntatori sospesi (*dangling*) a un file che ormai non esiste più. Una soluzione a questo problema potrebbe essere la ricerca di tutti i collegamenti e rimuoverli (**backpointers**). Un'altra soluzione prevede la conservazione del file fino a che non siano stati cancellati tutti i riferimenti ad esso.

11.3.5 DIRECTORY CON STRUTTURA A GRAFO GENERALE

Un serio problema connesso all'uso di una struttura a grafo aciclico consiste nell'assicurare che non vi siano cicli. Un problema analogo si presenta al momento di stabilire quando sia possibile cancellare un file. Come con le strutture delle directory a grafo aciclico, la presenza di uno 0 nel contatore dei riferimenti significa che non esistono più riferimenti al file o alla directory, e quindi il file può essere cancellato. Tuttavia, se esistono cicli, è possibile che il contatore dei riferimenti possa essere non nullo, anche se non è più possibile far riferimento a una directory o a un file.

Questa anomalia è dovuta alla possibilità di autoriferimento (ciclo) nella struttura delle directory. In questo caso è generalmente necessario usare un metodo di "ripulitura" (**garbage collection**) per stabilire quando sia stato cancellato l'ultimo riferimento e quando sia possibile riallocare lo spazio dei dischi. La garbage collection di un file system basato su dischi richiede però molto tempo, perciò viene tentata solo di rado.

Inoltre, poiché è necessaria solo a causa della presenza dei cicli, è molto più conveniente lavorare con una struttura a grafo aciclico. La difficoltà consiste nell'evitare i cicli quando si aggiungono nuovi collegamenti alla struttura. Ogni volta che si aggiunge un nuovo collegamento, basta utilizzare un semplice algoritmo di individuazione di cicli per verificare che sia tutto in regola.

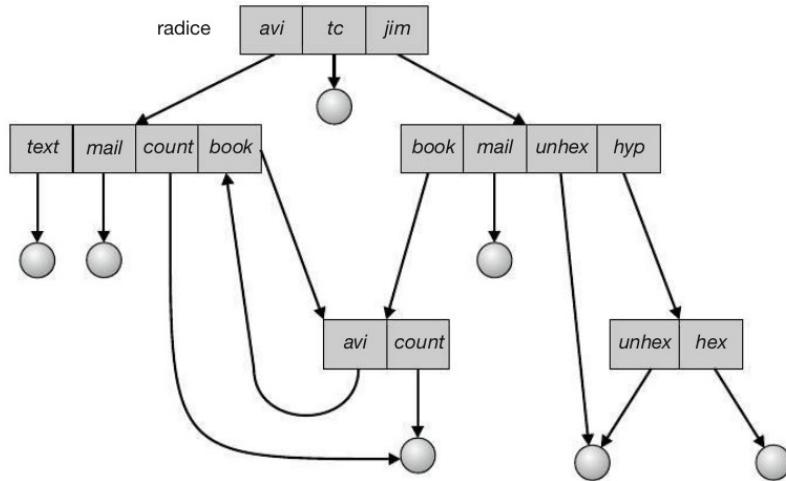


Figura 11.13 Directory a grafo generale.

11.4 MONTAGGIO DI UN FILE SYSTEM

Così come si deve aprire un file per poterlo usare, per essere reso accessibile ai processi di un sistema, un file system deve essere *montato*.

La procedura di montaggio è molto semplice: si fornisce al sistema operativo il nome del dispositivo e la sua locazione (detta **punto di montaggio**) nella struttura di file e directory alla quale agganciare il file system. Di solito, un punto di montaggio è una directory vuota. Il passo successivo consiste nella verifica da parte del sistema operativo della validità del file system contenuto nel dispositivo. Infine, il sistema operativo annota nella sua struttura della directory che un certo file system è montato al punto di montaggio specificato.

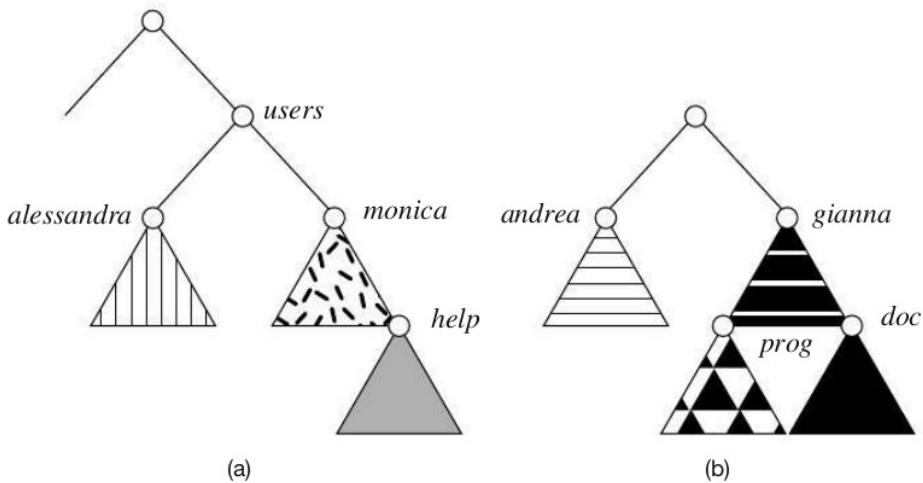


Figura 11.14 File system; (a) sistema esistente; (b) volume non montato.

Per illustrare l'operazione di montaggio, si consideri il file system rappresentato nella Figura 11.14, in cui i triangoli rappresentano i sottoalberi di directory di interesse. La Figura 11.14(a), mostra un file system esistente, mentre nella Figura 10.14(b) è raffigurato un volume non ancora montato che risiede in /device/dsk. A questo punto, si può accedere solo ai file del file system esistente. Nella Figura 11.15 si possono vedere gli effetti dell'operazione di montaggio del volume residente in /device/dsk al punto di montaggio /users.

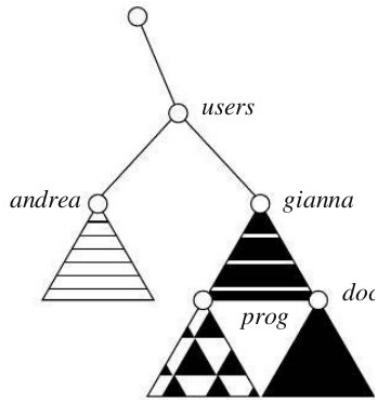


Figura 11.15 Punto di montaggio.

11.5 CONDIVISIONE DI FILE

La possibilità di condivisione di file è particolarmente utile per utenti che vogliono collaborare per ridurre le risorse richieste per raggiungere un certo obiettivo. Quindi, nonostante le difficoltà inerenti alla condivisione, i sistemi operativi orientati agli utenti devono soddisfare questa esigenza.

La condivisione può avvenire attraverso uno **schema di protezione**. Nei sistemi distribuiti, i file sono condivisi attraverso la rete. Il file system di rete (**NFS**, *network file system*) è un metodo comune di condivisione di file.

11.5.1 PROTEZIONE

La necessità di proteggere i file deriva direttamente dalla possibilità di accedervi. I sistemi che non permettono l'accesso ai file di altri utenti non richiedono protezione; quindi si può ottenere una completa protezione proibendo l'accesso. In alternativa si può permettere un accesso totalmente libero senza alcuna protezione. Questi orientamenti sono entrambi eccessivi, quindi non si possono applicare in generale; ciò che serve in realtà è un **accesso controllato**. Si possono controllare diversi tipi di operazione:

- **Lettura**;
- **Scrittura**: scrittura o riscrittura dei file;
- **Esecuzione**: caricamento dei file in memoria durante l'esecuzione;
- **Aggiunta**: Scrittura di nuove informazioni in coda ai file;
- **Cancellazione**: cancellazione dei file e liberazione del relativo spazio per un possibile riutilizzo;
- **Elencazione**: elencazione del nome e degli attributi del file.

11.5.2 CONTROLLO DEGLI ACCESSI

L'approccio più comune al problema della protezione è rendere l'accesso dipendente dall'identità dell'utente. Lo schema più generale per realizzare gli accessi dipendenti dall'identità consiste nell'associare una **lista di controllo degli accessi** (*access-control list, ACL*) a ogni file e directory; in tale lista sono specificati i nomi degli utenti e i relativi tipi d'accesso consentiti.

Questo sistema ha il vantaggio di permettere complessi metodi d'accesso. Il problema maggiore delle liste di controllo degli accessi è la loro lunghezza: per permettere a tutti di leggere un file, la lista deve contenere tutti gli utenti con accesso per la lettura. Questa tecnica comporta due inconvenienti:

- la costruzione di una lista di questo tipo può essere un compito noioso e non gratificante, soprattutto se la lista degli utenti del sistema non è nota a priori;

- l'elemento della directory, precedentemente di dimensione fissa, deve essere di dimensione variabile, quindi anche la gestione dello spazio è più complicata.

Questi problemi si possono risolvere introducendo una versione condensata della lista di controllo degli accessi. Per condensarne la lunghezza, molti sistemi raggruppano gli utenti di ogni file in tre classi distinte:

- **Proprietario**: è l'utente che ha creato il file;
- **Gruppo**: si tratta di un insieme di utenti che condividono il file e hanno bisogno di tipi di accesso simili;
- **Universo**: tutti gli utenti del sistema.

12 – REALIZZAZIONE DEL FILE SYSTEM

12.1 STRUTTURA DEL FILE SYSTEM

Il file system risiede permanentemente nella memoria secondaria, progettata per contenere in modo permanente grandi quantità di dati.

I dischi costituiscono la maggior parte della memoria secondaria in cui si conservano i file system. Quest'ultimo presenta una struttura stratificata, infatti per migliorare l'efficienza dell'I/O, i trasferimenti tra memoria centrale e dischi si eseguono per **blocchi**. Ciascun blocco è composto da uno o più settori. Il **blocco di controllo dei file (FCB, file control block)**, struttura la memorizzazione che contiene informazioni sui file, come la proprietà, i permessi e la posizione del contenuto.

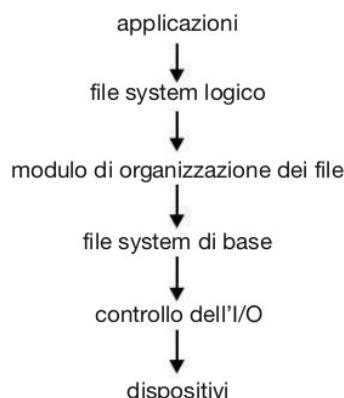


Figura 12.1 File system stratificato.

permessi per il file
data e ora di creazione, di ultimo accesso e di ultima scrittura
proprietario del file, gruppo, ACL
dimensione del file
blocchi di dati del file o puntatori a blocchi di dati del file

Figura 12.2 Tipica struttura di FCB (file-control block).

12.1.1 STRUTTURE DEL FILE SYSTEM CHE SI MANTENGONO IN MEMORIA

Le strutture del file system che si mantengono in memoria sono solo una parte minima rispetto a tutto il file. In una situazione ideale (con una memoria illimitata), caricheremo tutto il file nella memoria secondaria. In particolare vengono caricati soprattutto i **file indice** che indicano dove si trovano i dati del file (una sorta di tabella delle pagine).

Anche in questo caso c'è una stratificazione: spazio dell'utente, memoria del nucleo, memoria secondaria. C'è la possibilità o di scrivere o di leggere un file. Queste operazioni vengono poi gestite dalla memoria del nucleo che si interfaccia con la memoria secondaria.

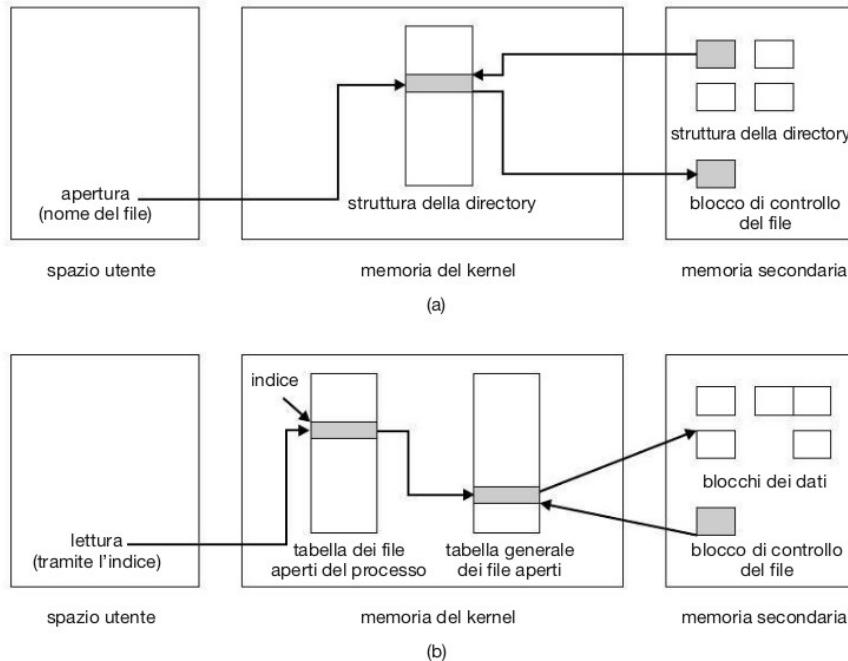


Figura 12.3 Strutture del file system che si mantengono nella memoria; (a) apertura di file; (b) lettura di file.

12.1.2 FILE SYSTEM VIRTUALI

Un metodo ovvio ma non ottimale per realizzare più tipi di file system è scrivere procedure di gestione di file e directory differenti per ciascun tipo di file system. Al contrario, la maggior parte dei sistemi operativi, compreso UNIX, impiega tecniche orientate agli oggetti per semplificare, organizzare e rendere modulare la soluzione. Questa tecnica viene fornita dal **file system virtuale (VFS, virtual file system)** che svolge due funzioni importanti:

- Separa le operazioni generiche del file system dalla loro realizzazione definendo un'interfaccia VFS uniforme.
- Permette la rappresentazione univoca di un file su tutta una rete. Il VFS è basato su una struttura di rappresentazione dei file detta vnode che contiene un indicatore numerico unico per tutta la rete per ciascun file.

Quindi, il VFS distingue i file locali da quelli remoti, e distingue i file locali a seconda del relativo tipo di file system.

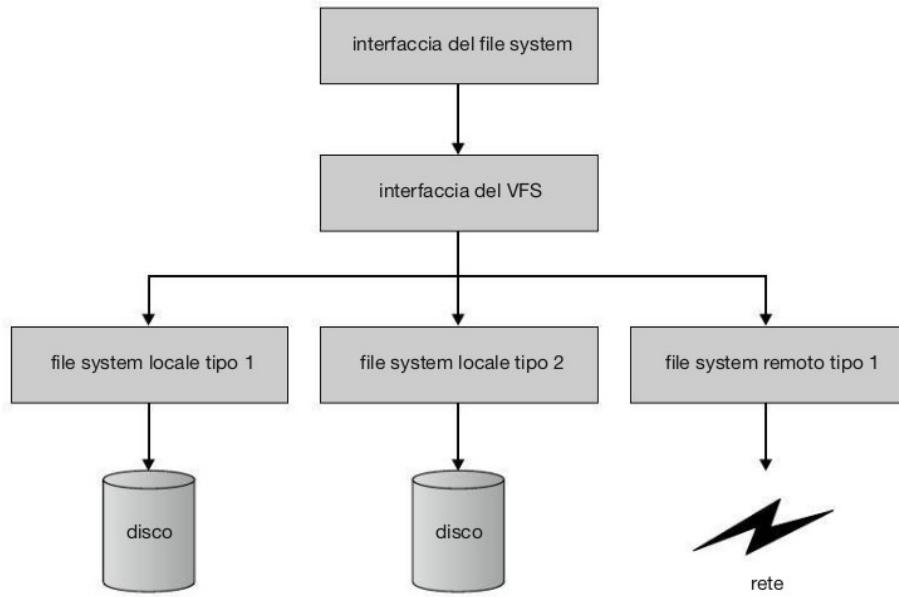


Figura 12.4 Schema di un file system virtuale.

12.2 REALIZZAZIONE DELLE DIRECTORY

12.2.1 LISTA LINEARE

Il più semplice metodo di realizzazione di una directory è basato sull'uso di una **lista lineare** contenente i nomi dei file con puntatori ai blocchi di dati. Questo metodo è di facile programmazione, ma la sua esecuzione è onerosa in termini di tempo.

12.2.2 TABELLA HASH

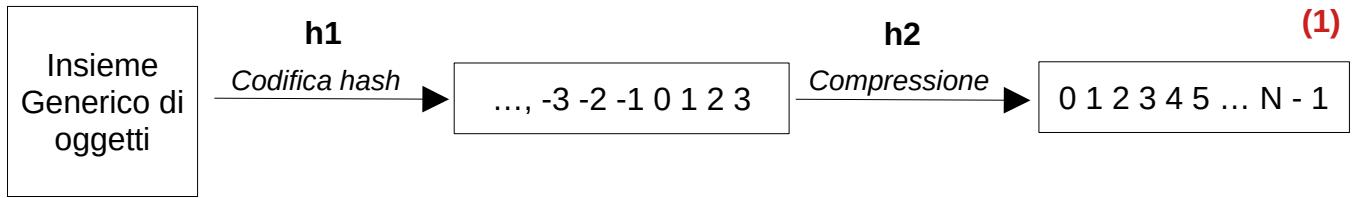
Un'altra struttura dati che si usa per realizzare le directory è la **tabella hash**. In questo caso una lista lineare contiene gli elementi di directory, ma si usa anche una struttura dati hash.

Questa struttura dati può diminuire notevolmente il tempo di ricerca nella directory. L'inserimento e la cancellazione sono abbastanza semplici, anche se occorre prendere provvedimenti per gestire le **collisioni**, cioè situazioni in cui da due nomi di file si ottiene un riferimento alla stessa locazione.

I termine inglese "hash" significa "sminuzzare". Tramite la funzione di hash, quindi, i dati vengono "sminuzzati" e portati a una lunghezza uniforme, indipendentemente dalla dimensione del valore di partenza. Si effettua in due passaggi:

- **Codifica hash**: l'obiettivo principale è quello di generare un valore univoco per un determinato input;
- **Compressione**: è un processo mediante il quale i dati vengono ridotti di dimensione in modo da occupare meno spazio di archiviazione o meno larghezza di banda durante la trasmissione. L'obiettivo principale della compressione è ridurre la quantità di dati necessari per rappresentare l'informazione senza perdita significativa di qualità o informazioni rilevanti.

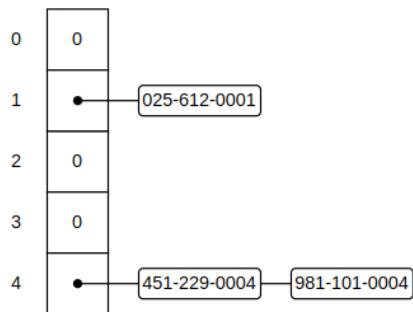
$$h(x) = h_2(h_1(x))$$



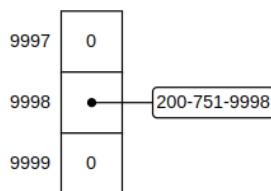
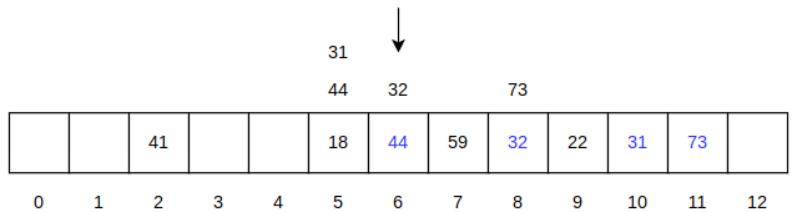
Successivamente bisogna andare a gestire le collisioni. Lo si può fare in due modi:

- **Chaining**: è una tecnica in cui ogni posizione del tabella hash è un puntatore a una lista contenuta di elementi che hanno lo stesso valore di hash. Quando si verifica una collisione, il nuovo elemento viene semplicemente aggiunto alla lista corrispondente. In altre parole, gli elementi con lo stesso valore di hash vengono "concatenati" l'uno all'altro;
- **Open Addressing**: è una tecnica in cui gli elementi con collisioni vengono inseriti direttamente in altre posizioni della tabella hash. Quando si verifica una collisione, invece di utilizzare una lista concatenata, la funzione di hash applicata all'elemento viene modificata per cercare la prossima posizione disponibile nella tabella.

CHAINING



OPEN ADDRESSING



Le maggiori difficoltà legate a una tabella hash sono la sua dimensione, che in genere è fissa, e la dipendenza della funzione hash da tale dimensione.

12.3 METODI DI ASSEGNAZIONE

Esistono tre metodi principali per l'allocazione dello spazio di un disco; può essere contigua, concatenata o indicizzata. Ciascuno di questi metodi presenta vantaggi e svantaggi.

12.3.1 ALLOCAZIONE CONTIGUA

Per usare il metodo di **allocazione contigua**, ogni file deve occupare un insieme di blocchi contigui del disco. L'allocazione contigua dello spazio per un file è definita dall'indirizzo del primo blocco e dalla lunghezza (espressa in numero di blocchi). Se il file è lungo n blocchi e comincia dalla locazione b , allora occupa i blocchi $b, b + 1, b + 2, \dots, b + n - 1$.

Accedere a un file il cui spazio è assegnato in modo contiguo è facile. L'allocazione contigua presenta però alcuni problemi. Una difficoltà riguarda l'individuazione dello spazio per un nuovo file. Il problema dell'allocazione contigua dello spazio dei dischi si può considerare un'applicazione particolare del problema generale **dell'allocazione dinamica della memoria**; il problema generale è, infatti, quello di soddisfare una richiesta di dimensione n data una lista di buchi liberi.

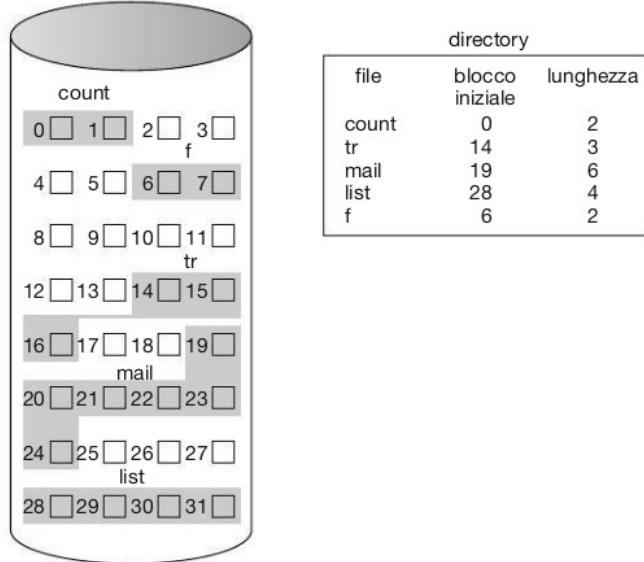


Figura 12.5 Allocazione contigua dello spazio dei dischi.

12.3.2 ALLOCAZIONE CONCATENATA

L'**allocazione concatenata** risolve tutti i problemi dell'allocazione contigua. Con questo tipo di allocazione, infatti, ogni file è composto da una lista concatenata di blocchi del disco i quali possono essere sparsi in qualsiasi punto del disco stesso. La directory contiene un puntatore al primo e all'ultimo blocco del file. Ogni blocco contiene un puntatore al blocco successivo.

Per creare un nuovo file si crea semplicemente un nuovo elemento nella directory. Con l'allocazione concatenata, ogni elemento della directory ha un puntatore al primo blocco del file.

L'allocazione concatenata presenta comunque alcuni svantaggi. Il problema principale riguarda il fatto che può essere usata in modo efficiente solo per i file ad accesso sequenziale. Per trovare l' i -esimo blocco di un file occorre partire dall'inizio del file e seguire i puntatori finché non si raggiunge l' i -esimo blocco. Un altro svantaggio dell'allocazione concatenata riguarda lo spazio richiesto per i puntatori. Un altro problema dell'allocazione concatenata riguarda l'affidabilità. Poiché i file sono tenuti insieme da puntatori sparsi per tutto il disco, s'immagini che cosa accadrebbe se un puntatore andasse perduto o danneggiato.

Una variante importante del metodo di allocazione concatenata consiste nell'uso della **tabella di allocazione dei file** (*file allocation table*, **FAT**). Tale metodo di allocazione dello spazio dei dischi, semplice ma efficiente, era usato nei sistemi operativi MS-DOS. La FAT si usa essenzialmente come una lista concatenata. L'elemento di directory contiene il numero del primo blocco del file. L'elemento della tabella indicizzato da quel numero di blocco contiene a sua volta il numero del blocco successivo del file.

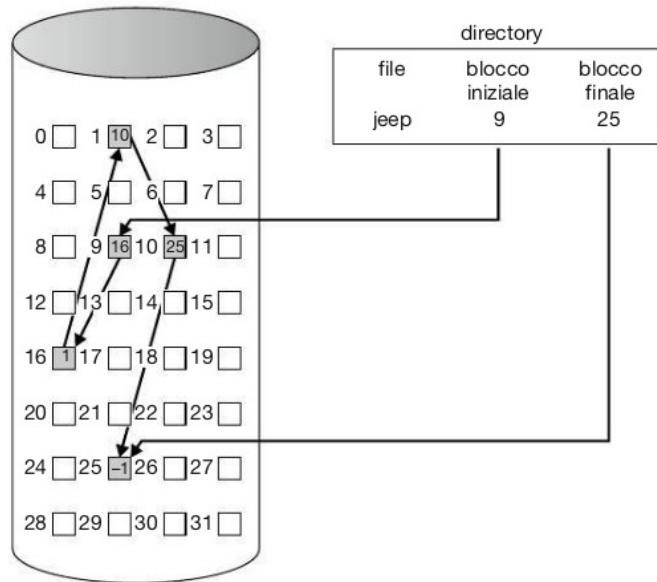


Figura 12.6 Allocazione concatenata dello spazio dei dischi.

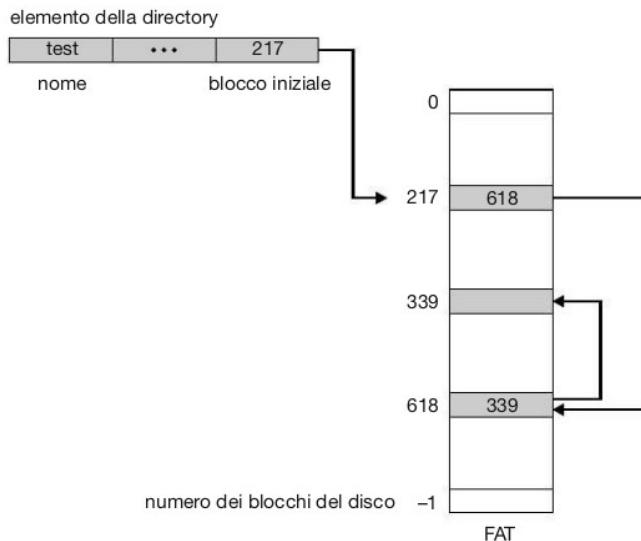


Figura 12.7 Tabella di allocazione dei file.

12.3.3 ALLOCAZIONE INDICIZZATA

In mancanza di una FAT, l'allocazione concatenata non è in grado di sostenere un efficiente accesso diretto, poiché i puntatori ai blocchi sono sparsi, con i blocchi stessi, per tutto il disco e si devono recuperare in ordine. **L'allocazione indicizzata** risolve questo problema, raggruppando tutti i puntatori in una sola locazione: il **blocco indice**.

Ogni file deve avere un blocco indice, quindi è auspicabile che questo sia quanto più piccolo possibile; ma se il blocco indice è troppo piccolo, non può contenere un numero di puntatori sufficiente per un file di grandi dimensioni, quindi è necessario disporre di un meccanismo per la gestione di questa situazione, che può essere:

- **Schema concatenato:** Un blocco indice è formato normalmente di un solo blocco di disco; perciò, ciascun blocco indice può essere letto e scritto esattamente con un'operazione. Per permettere la presenza di lunghi file è possibile collegare tra loro parecchi blocchi indice;

- **Indice a più livelli:** Una variante della rappresentazione concatenata consiste nell'impiego di un blocco indice di primo livello che punta a un insieme di blocchi indice di secondo livello che, a loro volta, puntano ai blocchi dei file. Per accedere a un blocco, il sistema operativo usa l'indice di primo livello, con il quale individua il blocco indice di secondo livello, e con esso trova il blocco di dati richiesto. Questo metodo potrebbe continuare fino a un terzo o quarto livello, a seconda della massima dimensione desiderata del file.
- **Schema combinato:** Un'altra possibilità, utilizzata nei sistemi basati su UNIX, consistente nel tenere i primi 15 puntatori del blocco indice nell'inode del file. I primi 12 di questi 15 puntatori puntano a **blocchi diretti**, cioè contengono direttamente gli indirizzi di blocchi contenenti dati del file. Gli altri tre puntatori puntano a **blocchi indiretti**.

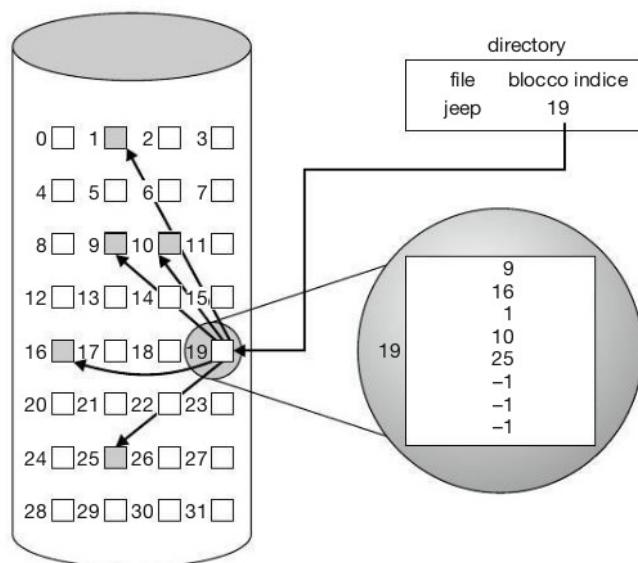


Figura 12.8 Allocazione indicizzata dello spazio dei dischi.

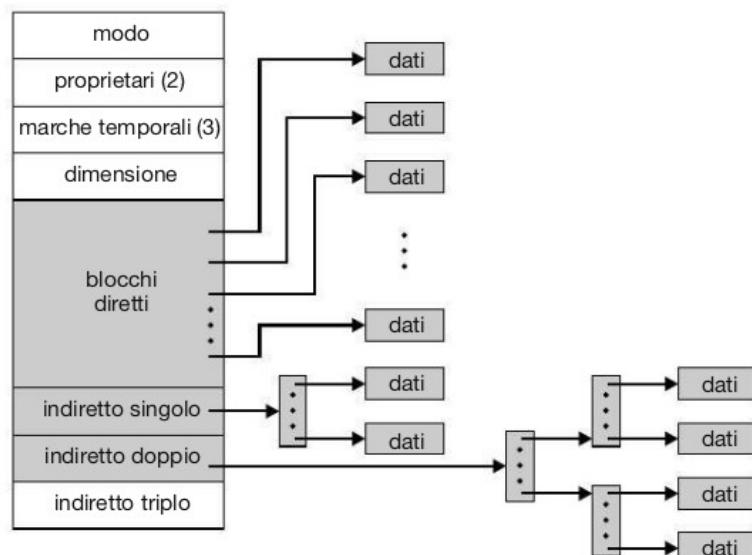


Figura 12.9 Inode di UNIX.

12.4 GESTIONE DELLO SPAZIO LIBERO

Poiché la quantità di spazio dei dischi è limitata, è necessario riutilizzare lo spazio lasciato dai file cancellati per scrivere nuovi file, se possibile. Per tener traccia dello spazio libero in un disco, il sistema conserva una **lista dello spazio libero**; vi sono registrati tutti gli spazi liberi, cioè non allocati ad alcun file o directory.

12.4.1 VETTORE DI BIT

Spesso la lista dello spazio libero si realizza come una **mappa di bit**, o **vettore di bit**. Ogni blocco è rappresentato da un bit: se il blocco è libero, il bit è 1, se il blocco è assegnato il bit è 0.

i consideri, per esempio, un disco dove i blocchi 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 e 27 sono liberi e gli altri sono allocati. La mappa di bit dello spazio libero è la seguente:

00111100111110001100000011100000...

I vantaggi principali che derivano da questo metodo sono la sua relativa semplicità ed efficienza nel trovare il primo blocco libero o n blocchi liberi consecutivi nel disco; Il numero del blocco è dato dalla seguente espressione:

$$(\text{numero di bit per parola}) \times (\text{numero di parole di valore 0}) + \text{offset del primo bit 1}.$$

Le caratteristiche dell'architettura guidano le funzioni del sistema operativo. Sfortunatamente, i vettori di bit sono efficienti solo se tutto il vettore è mantenuto nella memoria centrale, e viene ditanto in tanto scritto nella memoria secondaria allo scopo di consentire eventuali operazioni di ripristino; è possibile tenere il vettore nella memoria centrale se i dischi sono piccoli, come quelli usati nei microcalcolatori; tale soluzione non è applicabile ai dischi più grandi.

12.4.2 LISTA CONCATENATA

Un altro metodo di gestione degli spazi liberi consiste nel collegarli tutti, tenere un puntatore al primo di questi in una speciale locazione del disco e caricarlo in memoria. Il primo blocco libero contiene un puntatore al successivo, e così via.

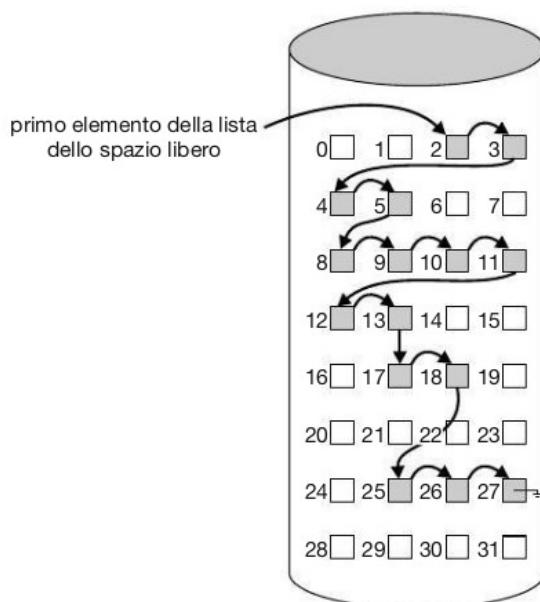


Figura 12.10 Lista concatenata degli spazi liberi su disco.

12.5 EFFICIENZA E PRESTAZIONI

L'uso efficiente di un disco dipende fortemente dagli algoritmi usati per l'allocazione del disco e la gestione delle directory.

Anche dopo aver scelto gli algoritmi fondamentali del file system le prestazioni possono essere migliorate in diversi modi:

- **Cache del disco**: sezione separata della memoria centrale dove tenere i blocchi in previsione di un loro utilizzo entro breve tempo;
- **Rilascio indietro** (*free-behind*) e **lettura anticipata** (*read-behind*): tecniche di ottimizzazione degli accessi sequenziali. Il primo rimuove una pagina dal buffer non appena si verifica una richiesta della pagina successiva; le pagine precedenti con tutta probabilità non saranno più usate e quindi sprecano spazio nel buffer. Nel secondo si leggono e si mettono nella cache la pagina richiesta e parecchie pagine successive: è probabile che queste pagine siano richieste una volta terminata l'elaborazione della pagina corrente.
- Si riserva e si gestisce una sezione della memoria come un **disco virtuale** o **disco RAM**.

12.5.1 I/O SENZA BUFFER CACHE UNIFICATA

Tecniche di memoria virtuale per la gestione dei dati dei file come pagine anziché come blocchi di file system.

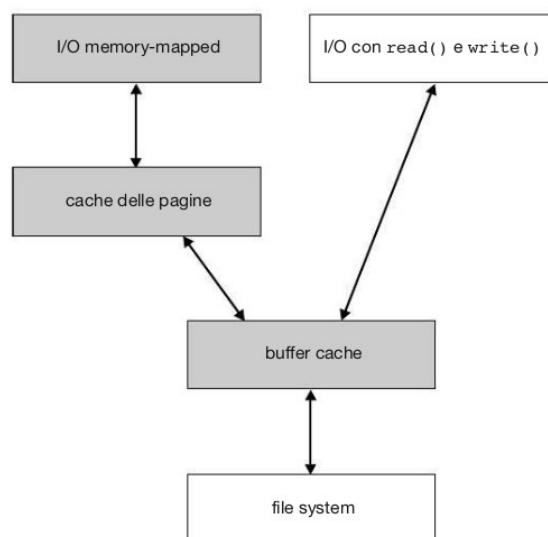


Figura 12.11 I/O senza una buffer cache unificata.

12.5.2 I/O CON BUFFER CACHE UNIFICATA

Con una buffer cache unificata sia l'associazione alla memoria sia le chiamate del sistema `read` e `write` usano la stessa cache delle pagine.

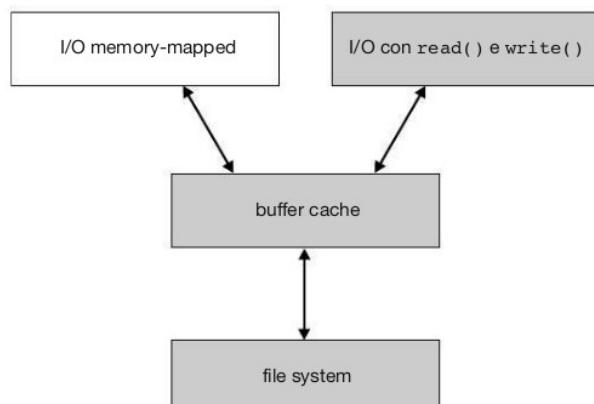


Figura 12.12 I/O con una buffer cache unificata.

12.6 RIPRISTINO

All'interno del file system potrebbero sorgere delle incoerenze. Esistono diversi metodi per affrontare queste circostanze, a seconda della struttura dati e degli algoritmi che utilizzano. Uno è la **verifica della coerenza**, durante la quale si confrontano i dati delle directory con quelli contenuti nei blocchi dei dischi, tentando di correggere ogni incoerenza.

I dischi magnetici sono soggetti a guasti, ed è necessario preoccuparsene e provvedere affinché in tal caso i dati non vadano persi definitivamente. A questo scopo si possono usare programmi di sistema che consentano di fare delle **copie di riserva** (*backup*) dei dati residenti nei dischi su altri dispositivi di memorizzazione, come nastri magnetici o hard disk supplementari. Il ripristino della situazione antecedente la perdita di un singolo file, o del contenuto di un intero disco, richiederà il **recupero** (*restore*) dei dati dalle copie di backup.

12.7 NFS

Un NFS (*Network File System*) è sia una realizzazione sia una definizione di un sistema per l'accesso a file remoti attraverso LAN (o WAN).

Nel contesto dell'NFS si considera un insieme di stazioni di lavoro interconnesse come un insieme di calcolatori indipendenti con un file system indipendenti. Lo scopo è quello di permettere un certo grado di condivisione tra questi file system, su richiesta esplicita, in modo trasparente.

Una directory remota è montata su una directory remota è montata su una directory di un file system locale. La directory montata assume l'aspetto di un sottoalbero integrante del file system locale, e sostituisce il sottoalbero che discende dalla directory locale.

La directory remota si specifica come argomento dell'operazione di montaggio in modo esplicito; si deve fornire la locazione (o il nome del calcolatore) della directory remota. Tuttavia, da questo momento in poi, gli utenti possono accedere ai file della directory remota in modo del tutto trasparente.

Uno degli scopi della progettazione dell'NFS era quello di operare in un ambiente eterogeneo di calcolatori, sistemi operativi e architetture di rete. La definizione dell'NFS è indipendente da questi mezzi e quindi incoraggia altre realizzazioni. Questa indipendenza si ottiene usando primitive RPC costruite su un protocollo di rappresentazione esterna dei dati (XDR, *external data representation*) usato tra due interfacce indipendenti dalla realizzazione.

La definizione dell'NFS distingue tra i servizi offerti da un meccanismo di montaggio e gli effettivi servizi d'accesso ai file remoti.

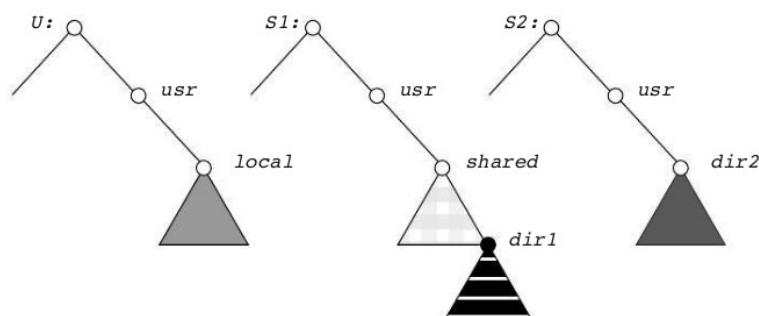


Figura 12.13 Tre file system indipendenti.

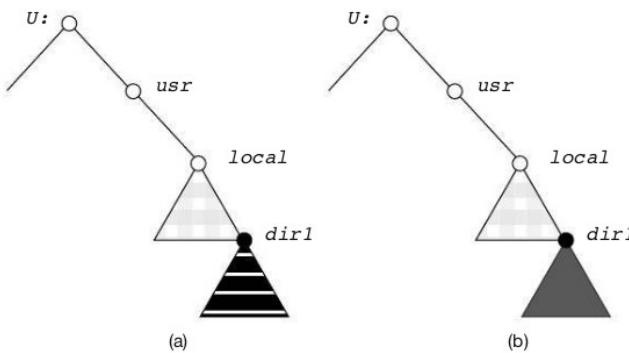


Figura 12.14 Montaggio nell'NFS; (a) montaggio; (b) montaggi a cascata.

12.7.1 PROTOCOLLO DI MONTAGGIO

Il **protocollo di montaggio** stabilisce la connessione logica iniziale tra un server e un client. Un'operazione di montaggio comprende il nome della directory remota da montare e il nome del calcolatore server in cui tale directory è memorizzata.

La richiesta di montaggi si associa alla RPC corrispondente e s'invia al server di montaggio in esecuzione nello specifico calcolatore server. Il server conserva una **lista di esportazione** che specifica i file system locali esportati per il montaggio e i nomi dei calcolatori a cui tale operazione è permessa. La lista può comprendere anche i diritti d'accesso, come la sola scrittura.

Quando il server riceve una richiesta di montaggio conforme alla propria lista di esportazione, riporta al client un **handle del file** da usare come chiave per ulteriori accessi ai file che si trovano all'interno del file system montato. L'handle del file contiene tutte le informazioni di cui ha bisogno il server per identificare un proprio file.

12.7.2 PROTOCOLLO NFS

Offre un insieme di RPC per operazioni su file remoti che svolgono le seguenti operazioni:

- Ricerca di un file in una directory;
- Lettura di un insieme di elementi di una directory;
- Manipolazione di collegamenti e di directory;
- Accesso ad attributi di file.
- Lettura e scrittura di file.

Una caratteristica importante dei server NFS è l'assenza dell'informazione di stato: ogni richiesta deve fornire un insieme completo di argomenti, tra cui un identificatore unico di file e un offset assoluto all'interno del file per svolgere le operazioni appropriate.

I dati modificati si devono riscrivere nei dischi del server prima che i risultati siano riportati al client. Il protocollo NFS non fornisce meccanismo per il controllo della concorrenza.

12.7.3 ARCHITETTURA NFS

Interfaccia del file system UNIX, basata sulle chiamate `open`, `read`, `write` e `close` e sui descrittori di file. Presenta un file system virtuale (VFS, *virtual file system*) che identifica i file locali da quelli remoti e invoca l'appropriata operazione del file system.

Un vantaggio di questa architettura è che il client e il server sono identici; così un calcolatore può essere un client, un server o entrambi.

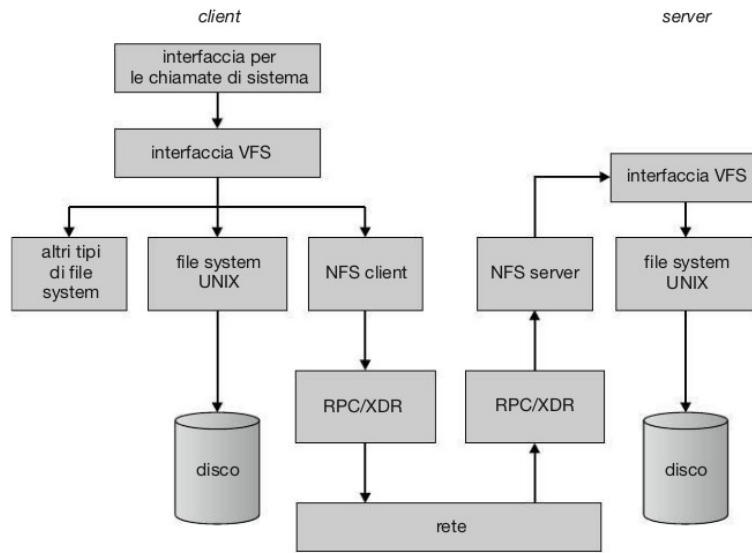


Figura 12.15 Schema dell'architettura dell'NFS.

12.7.4 TRADUZIONE DEI NOMI DI PERCORSO

La traduzione dei nomi di percorso (path-name translation) del protocollo NFS prevede l'analisi di un nome di percorso – per esempio `/usr/local/dir1/file.txt` – al fine di estrarre i nomi delle singole directory, o componenti. La traduzione dei nomi di percorso si compie suddividendo il percorso stesso in nomi di componenti ed eseguendo una chiamata `lookup()` dell'NFS separata per ogni coppia formata da un nome di componente e un *vnode* del directory.

Una cache per la ricerca dei nomi delle directory, nel client, conserva i *vnode* per i nomi delle directory remote; in questo modo si accelerano i riferimenti ai file con lo stesso nome di percorso iniziale. Se gli attributi restituiti dal server non corrispondono agli attributi del *vnode* nella cache, si scarta il contenuto della cache della directory.

12.7.5 FUNZIONAMENTO REMOTO

Tra le normali chiamate del sistema dello UNIX per operazioni su file e le RPC del protocollo NFS esiste una corrispondenza quasi da uno a uno.

Dal punto di vista concettuale, l'NFS aderisce al paradigma del servizio remoto, ma in pratica si usano tecniche di memorizzazione transitoria e cache per migliorare le prestazioni.

Non c'è corrispondenza diretta tra un'operazione remota e una RPC, le RPC prelevano blocchi e attributi del file che memorizzano localmente nelle cache. Le successive operazioni remote usano i dati nella cache, soggetti a vincoli di coerenza.

Esistono due cache: la cache degli attributi dei file (informazioni degli *inode*) e la cache dei blocchi di file.

I client non liberano i blocchi di scrittura differita finché il server non ha confermato che i dati sono stati scritti sui dischi.

13 – MEMORIA SECONDARIA E TERZIARIA

13.1 STRUTTURA DEI DISPOSITIVI DI MEMORIZZAZIONE

13.1.1 DISCHI MAGNETICI

I **dischi magnetici** sono il supporto fondamentale di memoria secondaria dei moderni sistemi elaborativi. Concettualmente, i dischi sono relativamente semplici: i **piatti** dei dischi hanno una forma piana e rotonda come quella dei Cd, con un diametro che comunemente varia tra 1,8 e 3,5 pollici, e le due superfici ricoperte di materiale magnetico; le informazioni si memorizzano registrandole magneticamente sui piatti.

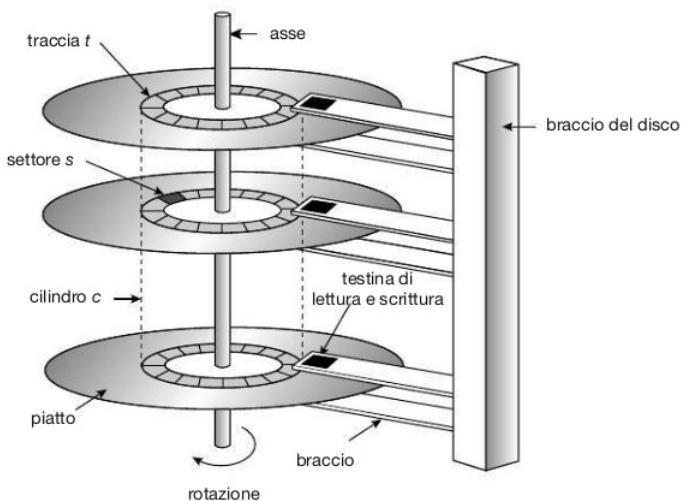


Figura 10.1 Schema di un disco a testine mobili.

Una testina di lettura e scrittura è sospesa su ciascuna superficie d'ogni piatto. Le testine sono attaccate al **braccio del disco** che le muove in blocco. La superficie di un piatto è divisa logicamente in **tracce** circolari a loro volta suddivise in **settori**; l'insieme delle tracce corrispondenti a una posizione del braccio costituisce un **cilindro**. In un'unità a disco possono esservi migliaia di cilindri concentrici e ogni traccia può contenere centinaia di settori.

La velocità di un disco è caratterizzata da due valori: la **velocità di trasferimento**, cioè la velocità con cui i dati fluiscono dall'unità a disco al calcolatore, e il **tempo di posizionamento**, detto anche tempo d'accesso casuale, che consiste di due componenti: il tempo necessario a spostare il braccio del disco in corrispondenza del cilindro desiderato, detto **tempo di ricerca** (seek time), e nel tempo necessario affinché il settore desiderato si porti, tramite la rotazione del disco, sotto la testina, detto **latenza di rotazione**.

13.1.2 DISCHI A STATO SOLIDO

A volte le vecchie tecnologie sono utilizzate in modo nuovo grazie ai cambiamenti economici e all'evolversi delle tecnologie stesse. Un esempio è la crescente importanza dei **dischi a stato solido**, o **SSD**. In breve, un SSD è una memoria non volatile che viene utilizzata come un disco rigido.

Gli SSD hanno le stesse caratteristiche dei dischi rigidi tradizionali, ma possono essere più affidabili, perché non hanno parti in movimento, e più veloci, perché non hanno tempo di ricerca o di latenza. Inoltre consumano meno energia. Tuttavia il costo al megabyte è

superiore rispetto ai dischi rigidi tradizionali, hanno meno capacità rispetto ai dischi rigidi più grandi e possono avere una durata di vita più breve, quindi il loro uso resta un po' limitato.



13.2 STRUTTURA DEI DISCHI

I dischi sono considerati come un grande vettore monodimensionale di **blocchi logici**, dove un blocco logico è la minima unità di trasferimento. Il vettore monodimensionale di blocchi logici corrisponde in modo sequenziale ai settori del disco:

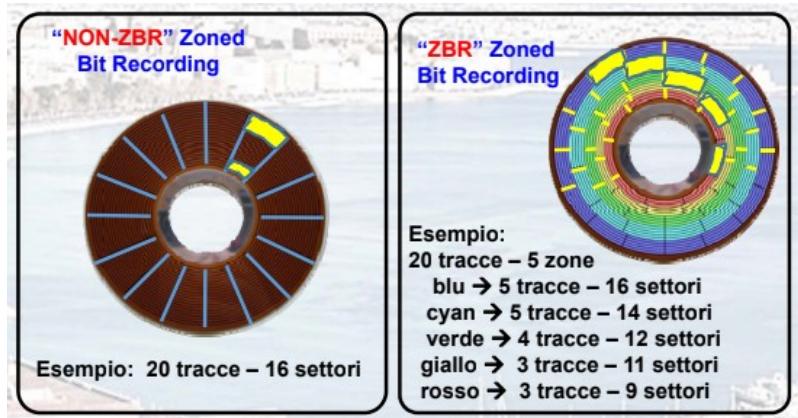
- Il settore 0 è il primo settore della prima traccia sul cilindro esterno;
- La corrispondenza prosegue ordinatamente lungo la prima traccia, quindi lungo le rimanenti tracce del primo cilindro, e così via di cilindro in cilindro, dall'esterno verso l'interno.

Sfruttando questa corrispondenza sarebbe possibile – almeno in teoria – trasformare il numero di un blocco logico in un indirizzo fisico di vecchio tipo, consistente in un numero di cilindro, un numero di traccia concernente quel cilindro, e un numero di settore relativo a quella traccia. In pratica, però, vi sono due motivi che rendono difficile quest'operazione: in primo luogo, la maggior parte dei dischi contiene settori difettosi, ma la corrispondenza nasconde questo fatto sostituendo ai settori malfunzionanti settori sparsi in altre parti del disco; in secondo luogo, il numero di settori per traccia in certe unità a disco non è costante.

Nei supporti che impiegano la **velocità lineare costante** (*constant linear velocity, CLV*) la densità di bit per traccia è uniforme. Più è lontana dal centro del disco, tanto maggiore è la lunghezza della traccia, tanto maggiore è il numero di settori che essa può contenere. Questo metodo si usa nelle unità per CD-ROM e DVD. In alternativa la velocità di rotazione dei dischi può rimanere costante, e la densità di bit decresce dalle tracce interne alle tracce più esterne per mantenere costante la quantità di dati che scorre sotto le testine. Questo metodo si usa nelle unità a disco magnetico ed è noto come **velocità angolare costante** (*constant angular velocity, CAV*). Questo metodo viene usato dai floppy disk.

Su un disco a tracce concentriche, la lunghezza fisica della traccia aumenta all'aumentare della distanza dal centro. Quindi, a densità di dati costante, la capacità di una traccia aumenta assieme alla distanza dal centro. Per gestire tale particolarità si è progettata una nuova strategia nota col nome di **Zone Bit Recording (ZBR)** il cui obiettivo è quello di memorizzare più settori nelle tracce esterne.

Poiché la tipologia usuale degli Hard Disk è di tipo CAV, la strategia ZBR dovrà opportunamente gestire la velocità variabile della lettura e della scrittura: la velocità aumenta nelle tracce esterne.



13.2.1 PARTIZIONAMENTO DI UN HARD DISK

Ci sono molti motivi che inducono a suddividere i dati di un Hard Disk in diverse aree logiche che siano differenziate anche fisicamente in modo da migliorarne la loro gestione:

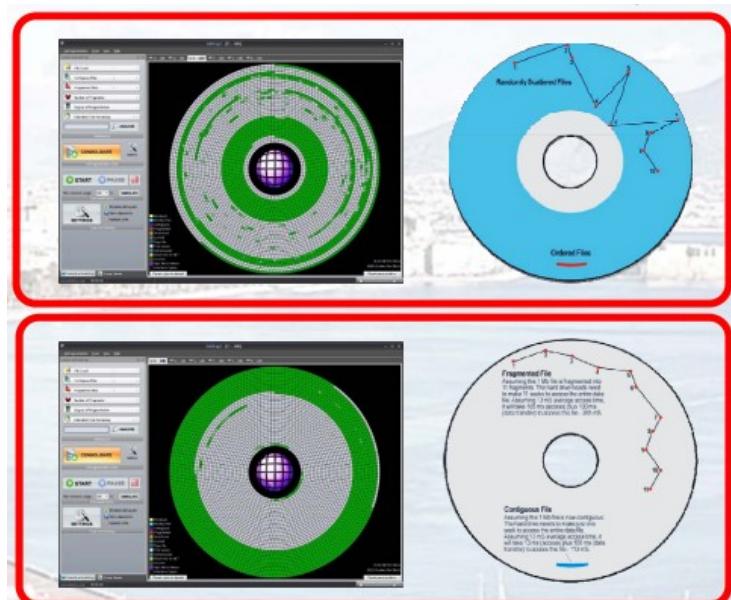
- Grandi capacità di contenimento di un Hard Disk;
- Necessità di separare al massimo aree di dati logicamente molto differenti tra loro;
- Necessità di evitare seek-time troppo elevati.

13.2.2 FRAMMENTAZIONE E DE-FRAMMENTAZIONE

La frammentazione e la de-frammentazione degli hard disk sono concetti correlati all'organizzazione dei dati su un disco rigido.

La frammentazione si verifica quando i dati su un disco rigido vengono memorizzati in modo non contiguo. Quando si scrivono file sul disco, il sistema operativo cerca di trovare spazi liberi consecutivi per posizionare i dati. Tuttavia, man mano che i file vengono aggiunti, modificati o rimossi, gli spazi liberi possono essere frammentati in più parti sparse in tutto il disco. Questo significa che i blocchi di dati di un file possono essere sparsi in diversi settori fisici del disco rigido. La frammentazione può rallentare l'accesso ai dati poiché il disco deve cercare in più posizioni per ricostruire un file.

La de-frammentazione, d'altra parte, è il processo di riorganizzazione dei dati sul disco in modo che i file siano memorizzati in modo contiguo, ovvero in blocchi consecutivi. La deframmentazione viene solitamente eseguita come un'operazione di manutenzione periodica sul disco rigido.



Quando si esegue la de-frammentazione, il software apposito analizza la struttura dei file sul disco e li riorganizza in modo da occupare spazi consecutivi. Ciò riduce la frammentazione e migliora le prestazioni del disco rigido, poiché il sistema operativo può accedere ai dati in modo più efficiente, leggendo blocchi di dati contigui invece di doverli cercare in più posizioni.

Tuttavia, è importante notare che la de-frammentazione potrebbe non essere necessaria o raccomandata su tutti i sistemi. I moderni sistemi operativi e i file system hanno algoritmi di gestione del disco che riducono la frammentazione in modo efficiente. Inoltre, con l'avvento delle unità a stato solido (SSD), la frammentazione ha un impatto minore sulle prestazioni, poiché gli SSD possono accedere ai dati in modo molto più rapido rispetto ai dischi rigidi tradizionali.

13.3 SCHEDULING DELL'HARD DISK

Una delle responsabilità del sistema operativo è quella di fare un uso efficiente dell'hardware. Nel caso delle unità a disco, far fronte a questa responsabilità significa garantire tempi d'accesso contenuti e ampiezze di banda elevate.

Nel caso dei dischi magnetici il tempo d'accesso si può scindere in due componenti principali: il **tempo di ricerca** (seek time), cioè il tempo necessario affinché il braccio dell'unità a disco sposti le testine fino al cilindro contenente il settore desiderato, e la **latenza di rotazione** (rotational latency), e cioè il tempo aggiuntivo necessario perché il disco ruoti finché il settore desiderato si trovi sotto la testina. L'**ampiezza di banda** (bandwidth) del disco è il numero totale di byte trasferiti diviso il tempo totale intercorso fra la prima richiesta e il completamento dell'ultimo trasferimento.

13.3.1 SCHEDULING IN ORDINE DI ARRIVO - FCFS

La forma più semplice di scheduling è, naturalmente, l'algoritmo di servizio secondo l'ordine d'arrivo (*first come, first served, FCFS*). Si tratta di un algoritmo intrinsecamente equo, ma che in generale non garantisce la massima velocità del servizio. Si consideri, per esempio, una coda di richieste per l'unità a disco che riguardino blocchi sui seguenti cilindri (nell'ordine):

98, 183, 37, 122, 14, 124, 65, 67.

Se si trova inizialmente al cilindro 53, la testina dell'unità a disco dovrà prima spostarsi al cilindro 98, poi al 183, 37, 122, 14, 124, 65 e infine al 67, per un movimento totale della testina, misurato in numero di cilindri visitati, di 640 cilindri.

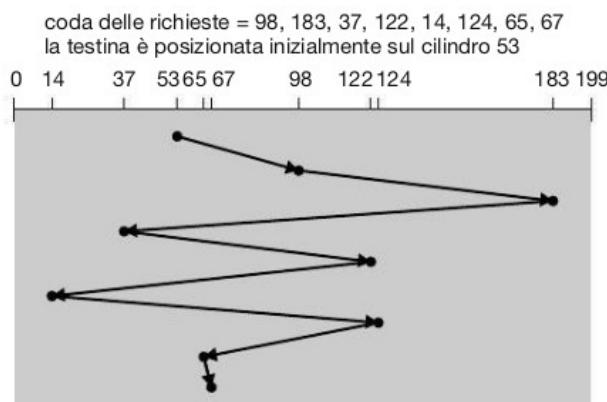


Figura 10.4 Scheduling FCFS.

13.3.2 SCHEDULING - SSTF

Sembra ragionevole servire tutte le richieste vicine alla posizione corrente della testina prima di spostarla in un'area lontana per soddisfarne altre: questa considerazione è alla base dell'**algoritmo di servizio secondo il più breve tempo di ricerca** (*shortest seek time first, SSTF*). SSTF sceglie la richiesta che dà il minimo tempo di ricerca rispetto alla posizione corrente della testina. In altre parole, l'algoritmo sceglie la richiesta relativa ai cilindri più vicini alla posizione della testina.

Se si considera nuovamente la sequenza di richieste dell'esempio sopra, il cilindro più vicino alla posizione iniziale della testina (cioè 53) è il 65, la successiva richiesta più vicina è quella relativa al cilindro 67; da questo cilindro, la richiesta relativa al cilindro 37 è più vicina di quella relativa al cilindro 98, ed è quindi servita per terza. Continuando allo stesso modo, sarà soddisfatta la richiesta relativa al cilindro 14, poi 98, 122, 124 e infine 183.

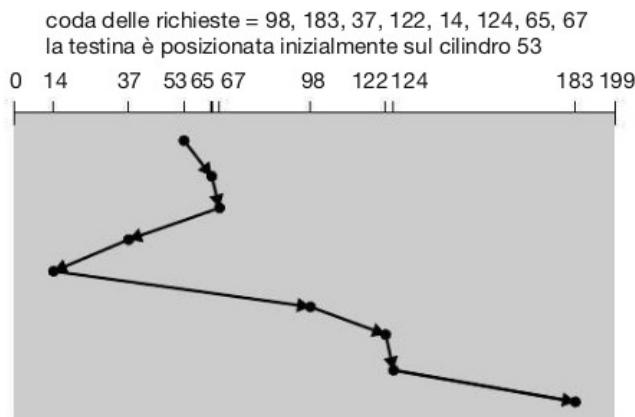


Figura 10.5 Scheduling SSTF.

Lo scheduling SSTF è essenzialmente una forma di scheduling per brevità (*shortest job first, SJF*), e al pari di questo, può condurre a situazioni d'attesa indefinita (starvation) di alcune richieste. Si ricordi infatti che nuove richieste possono giungere in qualunque momento: si supponga di avere due richieste in coda, una per il cilindro 14 e l'altra per il 186, e che mentre si sta servendo la richiesta relativa al cilindro 14, arrivi una nuova richiesta per un cilindro vicino al 14. Questa sarà la prossima richiesta soddisfatta, mentre la richiesta per il cilindro 186 dovrà attendere. La stessa situazione potrebbe ripetersi, perché un'altra richiesta relativa a una posizione in prossimità del cilindro 14 potrà giungere mentre si sta servendo la precedente richiesta: in teoria, un flusso continuo di richieste riferite a posizioni vicine fra loro potrebbe causare l'attesa indefinita della richiesta relativa al cilindro 186.

13.3.3 SCHEDULING - SCAN

Secondo l'**algoritmo SCAN** il braccio dell'unità a disco parte da un estremo del disco e si sposta verso l'altro estremo, servendo le richieste mentre attraversa i cilindri, finché non completa il tragitto: a questo punto, il braccio inverte la marcia, e la procedura continua. Le testine attraversano continuamente il disco nelle due direzioni.

Si consideri ancora l'esempio precedente. prima di poter applicare lo scheduling SCAN alle richieste per i cilindri 98, 183, 37, 122, 14, 124, 65, e 67, oltre la posizione corrente (53), occorre conoscere la direzione del movimento delle testine. Se lo spostamento è nella direzione del cilindro 0, l'unità a disco servirà prima la richiesta 37 e poi la 14; una volta giunto al cilindro 0, il braccio invertirà il movimento verso l'altro estremo del disco, servendo le richieste 65, 67, 98, 122, 124 e 183.

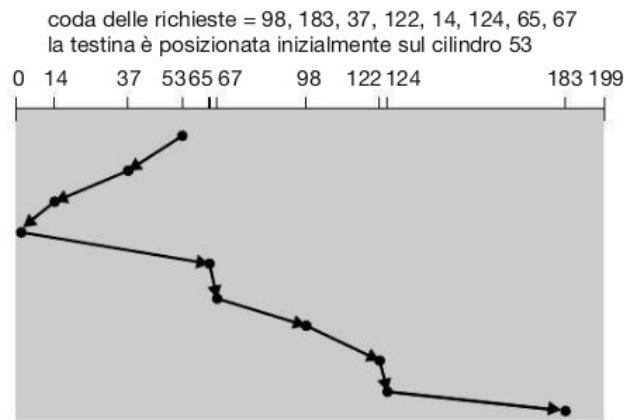


Figura 10.6 Scheduling SCAN.

13.3.4 SCHEDULING – C-SCAN

L'algoritmo **SCAN circolare** (*circular scan, C-SCAN*) è una variante dello scheduling SCAN concepita per garantire un tempo d'attesa meno variabile. Anche l'algoritmo C-SCAN, come lo SCAN, sposta la testina da un estremo all'altro del disco, servendo le richieste lungo il percorso; tuttavia, quando la testina giunge all'altro estremo del disco, ritorna immediatamente all'inizio del disco stesso, senza servire richieste durante il viaggio di ritorno.

L'algoritmo di scheduling C-SCAN, essenzialmente, tratta il disco come una lista circolare, cioè come se il primo e l'ultimo cilindro fossero adiacenti.

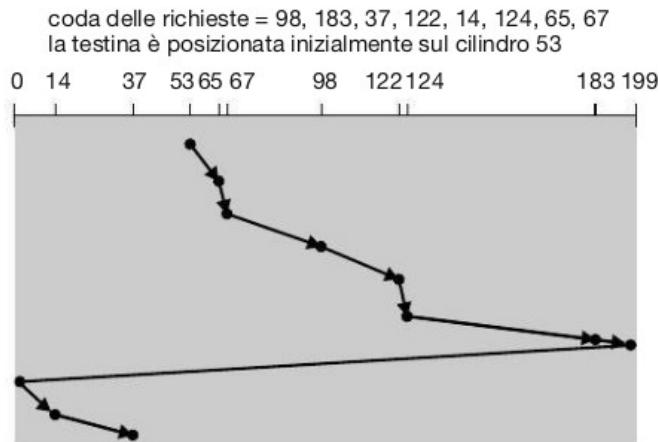


Figura 10.7 Scheduling C-SCAN.

13.3.5 SCHEDULING LOOK

Secondo la descrizione appena data, sia l'algoritmo SCAN sia il C-SCAN spostano il braccio dell'unità attraverso tutta l'ampiezza del disco; in pratica, nessuno dei due algoritmi è implementato in questo modo: più comunemente, il braccio si sposta solo finché ci sono altre richieste da servire in quella direzione, dopo di che cambia immediatamente direzione, senza giungere all'estremo del disco.

Queste versioni dello SCAN e del C-SCAN sono dette **LOOK** e **C-LOOK**, perché "guardano" (in inglese, look) se ci sono altre richieste da soddisfare lungo la direzione attuale prima di continuare a spostare la testina in quella direzione.

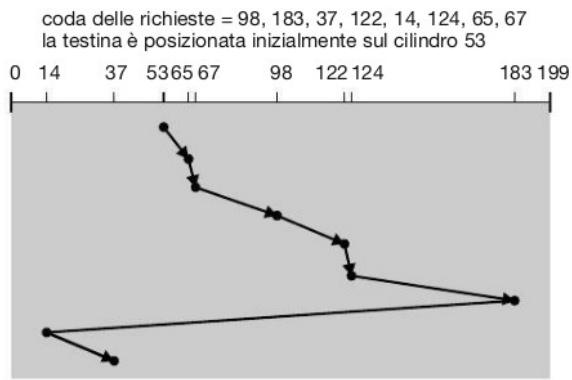


Figura 10.8 Scheduling C-LOOK.

13.3.6 SCELTA DI UN ALGORITMO DI SCHEDULING

Dati tanti diversi algoritmi di scheduling, come scegliere il migliore? Un algoritmo molto comune e naturalmente attraente è l'SSTF poiché aumenta le prestazioni rispetto all'FCFS; lo SCAN e il C-SCAN danno migliori prestazioni in sistemi che sfruttano molto le unità a disco, perché conducono con minor probabilità a situazioni d'attesa indefinita.

Per qualunque algoritmo di scheduling, le prestazioni dipendono comunque in larga misura dal numero e dal tipo di richieste. Per esempio, si supponga che la coda sia costituita in genere di una sola richiesta inevasa: tutti gli algoritmi danno allora luogo allo stesso comportamento, perché hanno una sola scelta possibile di spostamento della testina. In questo caso, tutti gli algoritmi si comportano come l'FCFS.

Le richieste di I/o per l'unità a disco possono essere notevolmente influenzate dal metodo adottato per l'allocazione dei file. Un programma che legga un file allocato in modo contiguo genererà molte richieste raggruppate, con un conseguente limitato spostamento della testina. Un file con allocazione concatenata o indicizzata, d'altro canto, potrebbe includere blocchi sparsi per tutto il disco, e richiedere quindi un maggiore movimento della testina.

A causa di queste complicazioni, l'algoritmo di scheduling del disco dovrebbe costituire un modulo a sé stante del sistema operativo, così da poter essere sostituito da un altro algoritmo qualora ciò fosse necessario; come algoritmo di partenza è ragionevole la scelta dell'SSTF o del LOOK.

13.4 STRUTTURE RAID

L'evoluzione tecnologica ha reso le unità a disco progressivamente più piccole e meno costose, tanto che oggi è economicamente possibile equipaggiare un sistema elaborativo con molti dischi. La presenza di più dischi, qualora si possano usare in parallelo, rende possibile l'aumento della frequenza a cui i dati si possono leggere o scrivere. Inoltre, una configurazione di questo tipo permette di migliorare l'affidabilità della memoria secondaria, poiché diventa possibile memorizzare le informazioni in più dischi in modo ridondante. In questo caso, un guasto a uno dei dischi non comporta la perdita di dati. Ci sono varie tecniche per l'organizzazione dei dischi, note col nome comune di **batterie ridondanti di dischi** (*redundant array of independent disks*, **RAID**), che hanno lo scopo di affrontare i problemi di prestazioni e affidabilità.

Quindi in sintesi, un sistema RAID è una combinazione fra più Hard Disk realizzata allo scopo di migliorare l'affidabilità del sistema, oppure le prestazioni o entrambi.

Le tecniche RAID possono essere realizzate sia in Hardware che in Software. La prima risulta essere più performante.

13.4.1 LIVELLI RAID

È possibile realizzare differenti tipologie di collegamento degli Hard Disk: ogni configurazione di ‘collegamento’ è definita come “LIVELLO RAID x” (con x = 0, 1, 2, ...) ed ogni specifica configurazione offre vantaggi e svantaggi.

- **RAID Level 0:** è la più semplice. Permette la distribuzione automatica dei dati su più dischi ma poiché i dati non sono replicati non garantisce la tolleranza ai guasti in caso di rottura del disco;
- **RAID Level 1:** è la tecnica più diffusa. I dati vengono replicati (*mirroring*) e nel caso di rottura di un disco si passa automaticamente ad un altro disco senza alcuna perdita dei dati;
- **RAID Level 3:** simile alla tecnica di livello 0 ma dedica un hard disk di recupero automatico d’errore mediante il controllo di parità calcolata a livello di byte;
- **RAID Level 4:** i dati vengono suddivisi in blocchi e scritti su diversi dischi, simile al RAID 0. Tuttavia, a differenza del RAID 0, nel RAID 4 viene utilizzato un disco dedicato per archiviare le informazioni di parità, calcolata a livello di blocchi (invece che a livello di byte). Questo disco, noto come disco di parità, memorizza le informazioni necessarie per ricostruire i dati in caso di guasto di uno dei dischi.
- **RAID Level 5:** si usa nel caso di sistemi che richiedono una garanzia di elevata protezione di dati e altre prestazioni. Usa *data striping* completo e un disco per la correzione dell’errore.

Il data striping, anche noto come striping a livello di dati, è una tecnica utilizzata nell’ambito dei sistemi di storage e delle reti di computer per migliorare le prestazioni e la velocità di accesso ai dati. Si tratta di una forma di parallelismo in cui i dati vengono suddivisi in segmenti più piccoli chiamati stripe e distribuiti su più dispositivi di storage o su più dischi all’interno di un sistema RAID

- **RAID Level 6:** RAID 6 è simile a RAID 5, ma fornisce una maggiore ridondanza dei dati. Richiede almeno quattro dischi e utilizza due blocchi di parità per proteggere i dati. Ciò significa che RAID 6 può sopportare il guasto simultaneo di due dischi senza perdita di dati. Tuttavia, RAID 6 richiede più capacità di archiviazione rispetto a RAID 5 poiché utilizza più informazioni di parità.
- **RAID Level 1+0:** RAID 10 combina le caratteristiche di RAID 1 e RAID 0. Richiede almeno quattro dischi, che vengono suddivisi in coppie e configurati in RAID 1. Infine, i due set di RAID 1 vengono combinati in un array RAID 0. RAID 10 offre un’elevata ridondanza e prestazioni, ma richiede un maggior numero di dischi rispetto ad altri livelli RAID.