

Sistemi Operativi 1

A.A. 2022-2023

Docente: A. Finzi

Dispense tratte dalle Slide del corso di Informatica a cura dello studente **S. Cerrone**

Sommario

1. Introduzione.....	9
Che cos'è un Sistema Operativo.....	9
<i>Definizione di sistema operativo</i>	9
Organizzazione di un sistema di calcolo	10
<i>CPU (Central Processing Unit)</i>	10
Gestione delle interruzioni	11
<i>Ciclo di I/O.....</i>	13
Caratteristiche del Sistema Operativo.....	13
Attività del sistema operativo	14
<i>Modalità Operativa Duale.....</i>	14
<i>Protezione CPU</i>	15
<i>Protezione Memoria.....</i>	15
Gestione Processi	15
Gestione Memoria.....	16
<i>Gestione Archiviazione</i>	16
<i>Strutture di Memoria.....</i>	17
<i>Cache</i>	18
<i>Sistemi di I/O.....</i>	18
2. Strutture dei Sistemi Operativi.....	19
Servizi di un SO	19
Chiamate di sistema	20
<i>Relazione tra API e Chiamata</i>	21
<i>Passaggio di parametri.....</i>	21
<i>Tipi di chiamate di sistema</i>	22
Programmi di sistema.....	23
Progettazione e realizzazione di un SO	23
<i>Implementazione</i>	23
Struttura del Sistema Operativo.....	24
<i>Struttura semplice</i>	24
<i>Metodo stratificato.....</i>	25
<i>Microkernel.....</i>	25
<i>Moduli.....</i>	25
<i>Struttura ibrida</i>	26
3. Processi.....	27
Concetto di processo	27
<i>Processo.....</i>	27
<i>Stato del processo.....</i>	27
<i>Descrittore di processo</i>	27
<i>Thread.....</i>	28
<i>Rappresentazione dei processi di Linux</i>	29
Scheduling dei processi	29
<i>Code di scheduling</i>	29
<i>Scheduler</i>	30
<i>Context Switch.....</i>	31
Operazioni sui processi.....	32
<i>Creazione di un processo</i>	32
<i>Terminazione di un processo</i>	34

Comunicazione tra processi.....	35
<i>Sistemi shared memory</i>	36
<i>Sistemi message passing</i>	37
Comunicazione nei sistemi client-server.....	39
<i>Socket</i>	39
<i>Chiamate di procedure remote</i>	40
<i>Pipe</i>	41
4. Thread	44
Introduzione	44
<i>Motivazioni</i>	44
<i>Vantaggi</i>	45
<i>Programmazione multicore</i>	45
Modelli di programmazione multithread.....	46
<i>Modello Many-to-One</i>	46
<i>Modello One-to-One</i>	47
<i>Modello Many-to-Many</i>	47
<i>Modello Two-Level</i>	47
Librerie dei thread	47
<i>Pthreads</i>	48
<i>Thread in Win32</i>	49
<i>Thread Java</i>	49
Questioni di programmazione multithread.....	49
<i>Chiamate di sistema fork ed exec</i>	49
<i>Cancellazione</i>	50
<i>Signal handling</i>	50
<i>Thread pool</i>	51
<i>Thread specific data</i>	52
<i>Attivazione dello scheduler</i>	52
Thread di Linux	53
5. Scheduling della CPU	54
Concetti fondamentali.....	54
<i>Ciclicità delle fasi d'elaborazione e di I/O</i>	54
<i>Scheduler della CPU</i>	54
<i>Scheduling preemptive</i>	54
<i>Dispatcher</i>	55
Criteri di scheduling.....	55
Algoritmi di scheduling	56
<i>First-Come, First-Served (FCFS) Scheduling</i>	56
<i>Shortest-Job-First (SJF) Scheduling</i>	57
<i>Scheduling con priorità</i>	58
<i>Round Robin (RR)</i>	59
<i>Scheduling a code multiple</i>	61
<i>Code multiple con feedback</i>	61
<i>Real-Time CPU Scheduling</i>	62
<i>Rate Monotonic Scheduling</i>	62
<i>Earliest Deadline First Scheduling (EDF)</i>	63
Scheduling dei Thread	64
<i>Pthread Scheduling API</i>	64

Scheduling per sistemi multiprocessore.....	66
<i>Soluzioni di scheduling per multiprocessori.....</i>	66
<i>Predilezione per il processore</i>	66
<i>Bilanciamento del carico</i>	67
<i>Processori multicore</i>	67
<i>Virtualizzazione e scheduling.....</i>	67
Esempi di sistemi operativi.....	68
<i>Scheduling di Solaris</i>	68
<i>Scheduling di Windows XP.....</i>	68
<i>Scheduling di Linux</i>	69
Valutazione degli algoritmi.....	69
<i>Modelli deterministici</i>	70
<i>Modelli di code e formula di Little</i>	70
<i>Simulazioni.....</i>	71
<i>Implementazione</i>	72
6. Sincronizzazione dei Processi	73
Introduzione	73
Problema della sezione critica.....	74
Soluzione di Peterson	75
Hardware per la sincronizzazione.....	76
Semafori	78
<i>Uso dei semafori.....</i>	78
<i>Realizzazione</i>	79
<i>Deadlock e Starvation.....</i>	80
<i>Inversione di priorità.....</i>	81
Problemi tipici di sincronizzazione	81
<i>Produttori e consumatori con bounded-buffer.....</i>	82
<i>Problema dei lettori-scrittori</i>	82
<i>Problema dei cinque filosofi</i>	84
Monitor.....	85
<i>Uso del costrutto monitor.....</i>	86
<i>Soluzione al problema dei cinque filosofi per mezzo di monitor</i>	87
<i>Realizzazione di un monitor per mezzo di semafori</i>	89
<i>Ripresa dei processi all'interno di un monitor</i>	89
<i>Monitor in Java</i>	90
Esempi di sincronizzazione	91
<i>Sincronizzazione in Solaris</i>	91
<i>Sincronizzazione in Windows XP.....</i>	91
<i>Sincronizzazione dei processi in Linux.....</i>	92
<i>Sincronizzazione in Pthreads</i>	92
7. Deadlocks	95
Modello del sistema	95
Caratterizzazione delle deadlock.....	95
<i>Condizioni necessarie.....</i>	95
<i>Deadlock con mutex Posix</i>	96
<i>Grafo di assegnazione delle risorse</i>	96
Metodi per la gestione delle situazioni di deadlock.....	98
Prevenire le situazioni di deadlock.....	99

<i>Mutua esclusione</i>	99
<i>Possesso e attesa</i>	99
<i>Impossibilità di prelazione</i>	99
<i>Attesa circolare</i>	100
Evitare le situazioni di deadlock	100
<i>Stato sicuro</i>	100
<i>Algoritmo con grafo di assegnazione delle risorse</i>	101
<i>Algoritmo del banchiere</i>	101
Rilevamento delle situazioni di stallo	104
<i>Istanza singola di ciascun tipo di risorsa</i>	104
<i>Più istanze di ciascun tipo di risorsa</i>	104
<i>Uso dell'algoritmo di rilevamento</i>	106
Ripristino da situazioni di deadlock	106
<i>Terminazione di processi</i>	106
<i>Prelazione su risorse</i>	107
8. Memoria Centrale	108
Introduzione	108
<i>Dispositivi essenziali</i>	108
<i>Binding di indirizzi</i>	109
<i>Spazi di indirizzi logici e fisici a confronto</i>	110
<i>Caricamento dinamico</i>	111
<i>Linking dinamico e librerie condivise</i>	112
Swapping	112
Allocazione contigua della memoria	113
<i>Rilocazione e protezione della memoria</i>	114
<i>Allocazione della memoria</i>	114
<i>Frammentazione</i>	116
Paging	116
<i>Metodo di base</i>	116
<i>Architettura di paging</i>	119
<i>Protezione</i>	121
<i>Pagine condivise</i>	121
Struttura della tabella delle pagine	122
<i>Paginazione gerarchica</i>	122
<i>Tabelle delle pagine di tipo hash</i>	124
<i>Tabella delle pagine invertita</i>	125
Segmentazione	126
<i>Metodo di base</i>	126
<i>Architettura di segmentazione</i>	127
Un esempio: Pentium Intel	127
<i>Segmentazione in Pentium</i>	128
<i>Paging in Pentium</i>	129
9. Memoria Virtuale	131
Introduzione	131
Paginazione su richiesta	132
<i>Concetti fondamentali</i>	133
<i>Prestazioni della paginazione su richiesta</i>	135
Copy on write	135

Sostituzione delle pagine.....	136
<i>Schema di base</i>	136
<i>Algoritmo di sostituzione FIFO</i>	138
<i>Algoritmo ottimale</i>	139
<i>Algoritmo LRU</i>	139
<i>Sostituzione delle pagine per approssimazione a LRU</i>	140
<i>Sostituzione delle pagine basata su conteggio</i>	142
<i>Algoritmi Page Buffering</i>	142
<i>Applicazioni e sostituzione della pagina</i>	143
Allocazione dei frame	143
<i>Numero minimo di frame</i>	143
<i>Algoritmi di allocazione</i>	144
<i>Allocazione globale e allocazione locale</i>	145
<i>Accesso non uniforme alla memoria</i>	145
Thrashing (paginazione degenere)	146
<i>Cause del thrashing</i>	146
<i>Modello working set</i>	148
<i>Frequenza dei page fault</i>	149
Allocazione di memoria nel kernel	150
<i>Sistema buddy</i>	151
Slab allocator	151
<i>Layout memoria x86</i>	152
Altre considerazioni.....	153
<i>Prepaging</i>	153
<i>Dimensione delle pagine</i>	153
<i>TLB reach</i>	154
<i>Tabella delle pagine invertita</i>	154
<i>Struttura dei programmi</i>	155
<i>I/O interlock</i>	156
<i>Esempi tra i sistemi operativi</i>	156
10. Interfaccia del File System	159
Concetto di file	159
<i>Attributi dei file</i>	159
<i>Operazioni sui file</i>	160
<i>Tipi di file</i>	162
<i>Struttura dei file</i>	162
Metodi di accesso	163
<i>Accesso sequenziale</i>	163
<i>Accesso diretto</i>	163
<i>Altri metodi d'accesso</i>	164
Struttura della directory e del disco	164
<i>Struttura della memorizzazione di massa</i>	165
<i>Generalità sulla directory</i>	165
<i>Single-Level Directory</i>	166
<i>Two-Level Directory</i>	166
<i>Tree-Structured Directory</i>	167
<i>Acyclic-Graph Directory</i>	168
<i>General Graph Directory</i>	168
Montaggio di un file system	170

Condivisione di file	170
<i>Utenti multipli</i>	171
<i>File system remoti</i>	171
<i>Semantica della coerenza</i>	171
Protezione	172
<i>Tipi d'accesso</i>	173
<i>Controllo degli accessi</i>	173
11. Realizzazione del File System	175
Struttura del file system	175
Implementazione del file system.....	176
<i>Partizioni e montaggio</i>	178
<i>File system virtuali</i>	179
Implementazione delle directory	180
<i>Lista lineare</i>	180
<i>Tabella hash</i>	181
Metodi di allocazione	181
<i>Allocazione contigua</i>	181
<i>Allocazione concatenata</i>	182
<i>Allocazione indicizzata</i>	184
<i>Scelta del metodo di allocazione</i>	186
Gestione dello spazio libero	187
<i>Vettore di bit o BitMap</i>	187
<i>Lista Concatenata</i>	187
<i>Raggruppamento</i>	187
<i>Conteggio</i>	187
<i>Efficienza e prestazioni</i>	188
<i>Efficienza</i>	188
<i>Prestazioni</i>	188
Ripristino.....	190
<i>Verifica della coerenza</i>	190
<i>File system con log delle modifiche</i>	191
<i>Altre soluzioni</i>	192
<i>Copie di riserva e recupero dei dati</i>	192
12. Memoria di Massa	194
Struttura dei dispositivi di memorizzazione	194
<i>Hard Disk</i>	194
<i>Dispositivi NVM</i>	195
<i>Nastri magnetici</i>	196
Struttura dei dischi e mapping degli indirizzi	196
Connessione dei dischi	197
<i>Direct Access Storage</i>	197
<i>Network-Attached Storage</i>	197
<i>Cloud Storage</i>	198
<i>Storage Area Network</i>	198
Scheduling del disco	198
<i>FCFS</i>	199
<i>SSTF</i>	199
<i>SCAN</i>	200

<i>C-SCAN</i>	201
<i>C-LOOK</i>	201
<i>Selezionare un algoritmo di Disk-Scheduling</i>	202
<i>NVM scheduling</i>	203
Gestione dell'unità a disco	203
<i>Formattazione del disco</i>	203
<i>Boot block</i>	204
<i>Bad block</i>	205
Gestione dello Swap-Space	206
<i>Uso dello swap space</i>	206
<i>Collocazione dello swap space</i>	206
<i>Gestione dello swapping in Linux</i>	207
Strutture RAID	208
<i>Mirrored Disk</i>	208
<i>Disk Striping</i>	208
<i>Livelli RAID</i>	209
<i>Altre caratteristiche</i>	210
<i>Estensioni</i>	210
<i>Problemi connessi a RAID</i>	210
13. Sistemi di I/O	212
Architetture e dispositivi di I/O	212
<i>Polling</i>	212
<i>Interrupt</i>	212
<i>Accesso diretto alla memoria (DMA)</i>	213
Interfaccia di I/O per le applicazioni.....	214
<i>Dispositivi con trasferimento a blocchi o a caratteri</i>	214
<i>Orologi e temporizzatori</i>	214
<i>I/O bloccante e non bloccante</i>	215
Sottosistema per l'I/O del kernel.....	215
<i>Scheduling</i>	215
<i>Memorizzazione transitoria</i>	216
<i>Code e uso esclusivo dei dispositivi</i>	216
<i>Gestione degli errori</i>	217

N.B.: Per la preparazione dello scritto conviene vedere le slide del prof che sono molto più sintetiche poiché lui vuole risposte brevi con magari qualche illustrazione.

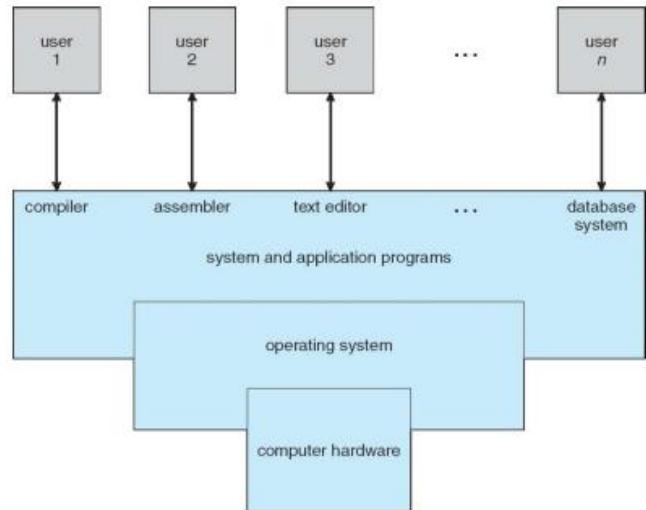
1. Introduzione

Che cos'è un Sistema Operativo

Un SO è un programma (molto complesso) che gestisce le risorse di un calcolatore e fa da intermediario tra l'utente di un calcolatore e l'hardware del calcolatore. Come obiettivi un SO deve: gestire l'esecuzione dei programmi utente; semplificare l'interazione dell'utente con il calcolatore; rendere efficace ed efficiente l'utilizzo del calcolatore; utilizzare l'hardware in modo efficiente.

Un sistema di calcolo si può suddividere in quattro componenti:

- **Hardware** (dispositivi fisici): composti da CPU (unità centrale d'elaborazione), dalla memoria e dai dispositivi I/O, forniscono al sistema le risorse di calcolo fondamentali.
- **Sistema Operativo**: rappresenta il primo strato software. Controlla e coordina l'uso dei dispositivi da parte dei programmi applicativi per gli utenti.
- **Programmi applicativi**: definiscono il modo in cui si usano queste risorse per la risoluzione dei problemi computazionali degli utenti.
- **Utenti**: persone, altri processi, altri calcolatori, etc...



Un sistema di calcolo si può anche considerare come l'insieme dei suoi elementi fisici, programmi e dati. Il sistema operativo offre gli strumenti per impiegare in modo corretto queste risorse; non compie operazioni di per sé utili, ma definisce semplicemente un ambiente nel quale altri programmi possono lavorare in modo utile.

In generale, non si dispone di una definizione completa ed esauriente di sistema operativo. I sistemi operativi esistono poiché rappresentano una soluzione ragionevole al problema di realizzare un sistema di calcolo che si possa impiegare in modo utile, per eseguire i programmi utenti e facilitare la soluzione dei problemi degli utenti, ed è proprio per questo scopo che sono stati costruiti i calcolatori; ma, poiché un calcolatore di per sé non è molto facile da utilizzare, sono stati sviluppati i programmi applicativi. Questi programmi sono diversi tra loro, ma richiedono alcune funzioni comuni, per esempio il controllo dei dispositivi di I/O. Tali funzioni comuni, di controllo e assegnazione delle risorse, sono state racchiuse in un unico insieme coerente di programmi: il sistema operativo.

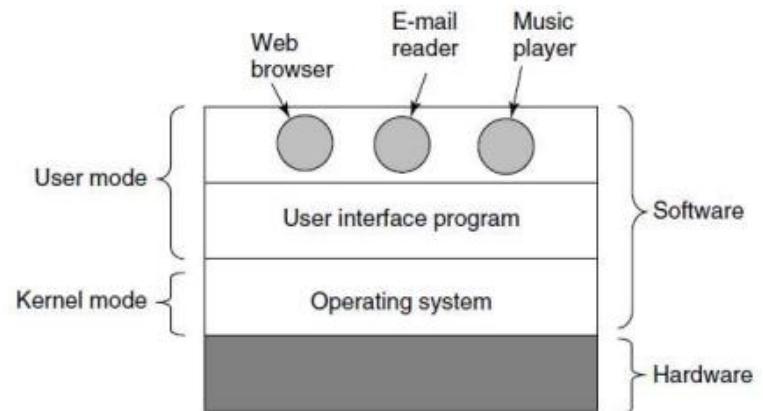
- Il SO è un allocatore di risorse
 - Gestisce tutte le risorse del calcolatore
 - Decide tra richieste conflittuali per l'assegnazione corretta ed efficiente delle risorse
- Il SO è un programma di controllo
 - Controlla l'esecuzione dei programmi evitando errori e usi impropri del calcolatore
- Il SO come macchina estesa
 - Fornisce un'astrazione fornendo un ambiente coerente ed uniforme per l'esecuzione dei programmi

Definizione di sistema operativo

Non si dispone nemmeno di una definizione universalmente accettata di che cosa faccia parte di un sistema operativo. Un punto di vista semplice è considerare che esso comprenda tutto quello che il rivenditore fornisce quando gli si richiede "il sistema operativo". Tuttavia, una definizione più comune è quella secondo

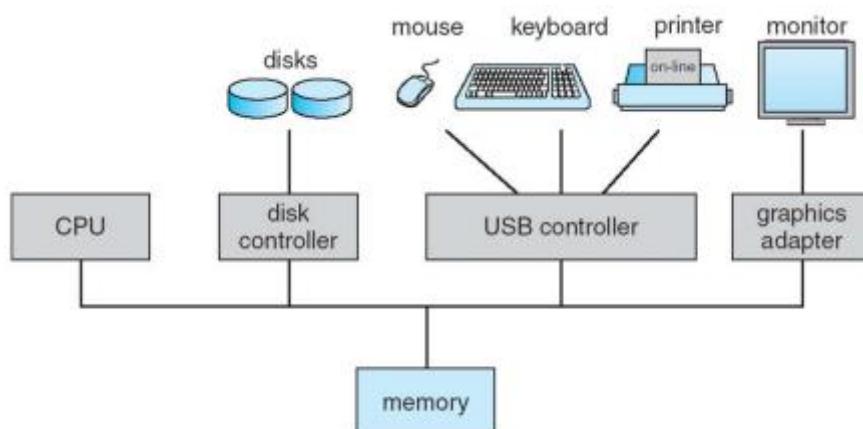
cui il sistema operativo è il solo programma che funziona sempre nel calcolatore, generalmente chiamato **kernel** (nucleo). Tutto il resto (ciò che non è kernel) se non è un **programma di sistema** (unità associate al SO ma non fanno parte del kernel) è un **programma applicativo** (tutti i programmi non correlati al funzionamento del sistema).

Il kernel gestisce le risorse essenziali (la CPU, la memoria, le periferiche, etc...) mentre tutto il resto, anche l'interazione con l'utente, è ottenuto tramite programmi eseguiti dal kernel, che accedono alle risorse hardware tramite delle richieste a quest'ultimo. Nei moderni calcolatori il SO è il solo programma ad avere il completo accesso all'hardware (modalità privilegiata: **kernel mode**), gli altri programmi vengono eseguiti dal kernel (modalità protetta: **user mode**).



Organizzazione di un sistema di calcolo

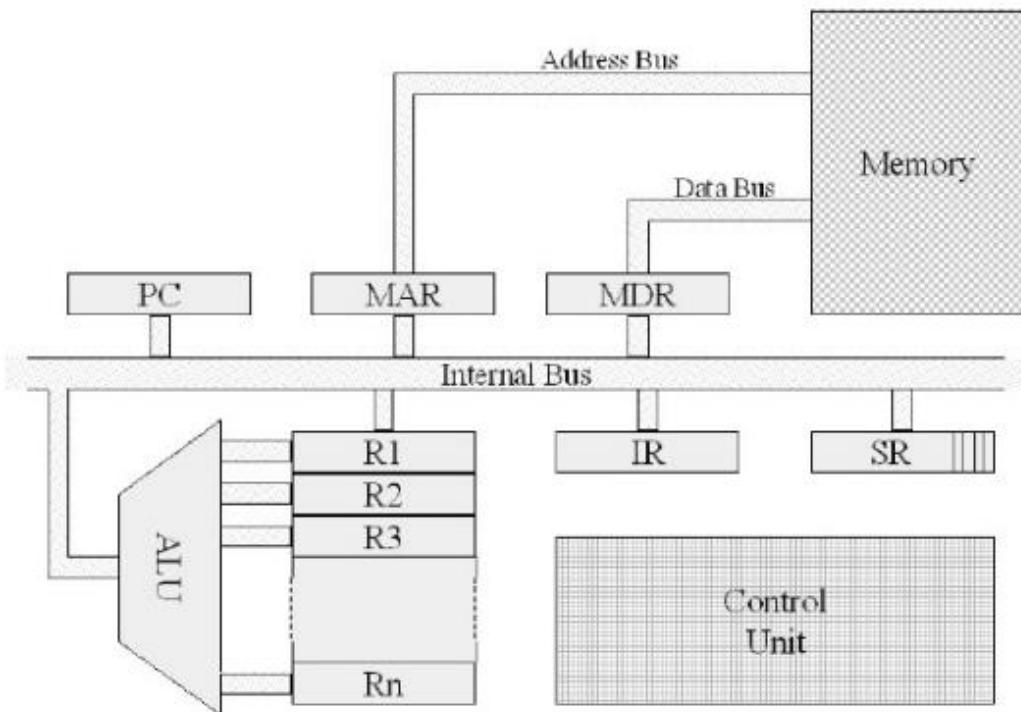
Il sistema operativo è strettamente legato all'architettura del calcolatore. Un moderno calcolatore d'uso generale è composto da una CPU e da un certo numero di controllori di dispositivi connessi (driver) attraverso un canale di comunicazione comune (bus) che permette l'accesso alla memoria condivisa dal sistema. Ciascuno di questi controllori si occupa di un particolare tipo di dispositivo fisico (per esempio, unità a disco, dispositivi audio e unità video). La CPU e questi controllori possono operare in modo concorrente, contendendosi i cicli d'accesso alla memoria. La sincronizzazione degli accessi alla memoria è garantita dalla presenza di un controllore di memoria.



CPU (Central Processing Unit)

Esegue le istruzioni della memoria: ogni CPU ha un **set di istruzioni** che può eseguire ciclicamente (fetch, decode, execute) e dei **registri** (program counter, stack pointer, instruction register, status register, memory address register, memory data register).

- **CPU:** componente hardware che esegue le istruzioni
- **Processore:** un chip fisico che contiene una o più CPU
- **Unità di calcolo (core):** unità di elaborazione di base della CPU
- **Multicore:** che include più unità di calcolo sulla stessa CPU
- **Multiprocessore:** che include più processori



Un evento è di solito segnalato da un'interruzione dell'attuale sequenza d'esecuzione della CPU, che può essere causata da un dispositivo fisico o da un programma. Nel primo caso si parla di segnale d'interruzione o, più brevemente, interruzione (**interrupt**); si tratta di segnali che i controllori dei dispositivi e altri elementi dell'architettura possono inviare alla CPU, di solito attraverso il bus di sistema. Nel secondo caso si parla di segnale di eccezione o, più brevemente, eccezione (**exception** o **trap**), che può essere causata da un programma in esecuzione a seguito di un evento eccezionale, riconosciuto tramite l'architettura della CPU (per esempio un errore: una divisione per zero o un accesso alla memoria non valido); oppure a seguito di una richiesta specifica effettuata da un programma utente per ottenere l'esecuzione di un servizio del sistema operativo, attraverso una speciale istruzione detta chiamata di sistema (**system call**) o chiamata supervisore (**supervisor call**, SVC).

Gestione delle interruzioni

I dispositivi I/O e la CPU sono in esecuzione concorrente:

- Ogni **controllore di dispositivo** è responsabile di un tipo di dispositivo
- Ogni controllore ha un **buffer locale**
- La CPU porta dati dalla/alla memoria ai/dai buffer locali
- I/O porta dati dal dispositivo ai buffer locali del controller
- Ogni controllore informa la CPU che ha finito le sue operazioni attraverso il meccanismo delle interruzioni (**interrupt**)

Le interruzioni permettono di gestire le operazioni concorrenti sovrapponendo CPU e operazioni di I/O così da evitare il **busy waiting** (attesa attiva).

Un sistema operativo moderno è guidato dalle interruzioni, le quali trasferiscono il controllo ad una procedura di servizio dell'interruzione ed una volta eseguita la routine di servizio il controllo ritorna all'operazione interrotta.

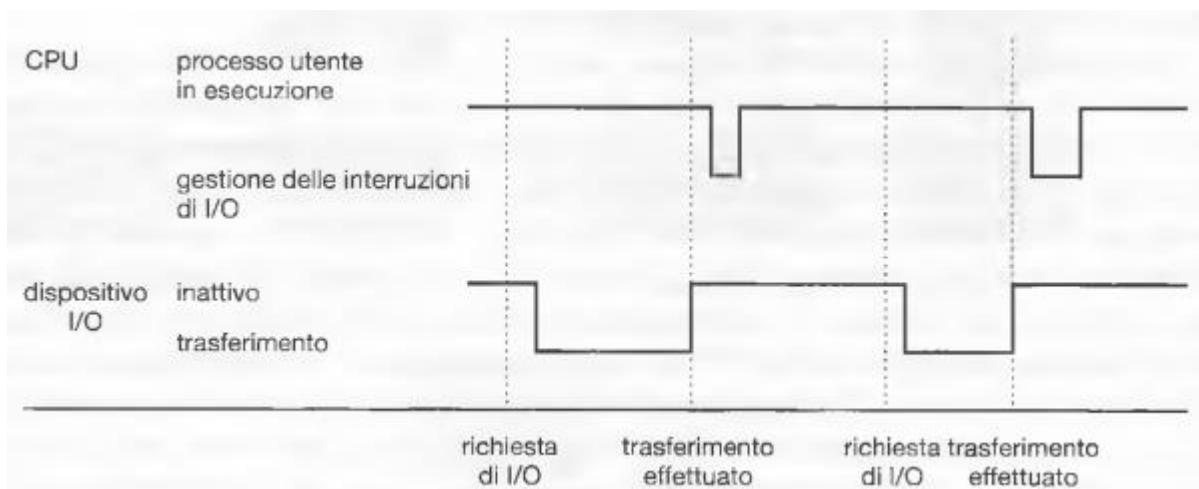
Le interrupt non è l'unica modalità con cui gli eventi generati dai dispositivi I/O possono essere gestiti dalla CPU, infatti esiste anche il polling:

- Con il **polling**, la CPU tiene traccia delle comunicazioni dei dispositivi di I/O interrogandoli ad intervalli regolari

- Con **interrupt**, il dispositivo di I/O interrompe la CPU comunicando ad essa che ha bisogno di andare in esecuzione

Forniamo un diagramma temporale delle interruzioni per un singolo programma che invia dati in output:

- Per avviare I/O la CPU carica opportuni registri del controllore I/O
- La CPU continua il processamento ed intanto il controllore I/O avvia le operazioni sul proprio buffer
- Quando il controllore ha terminato invia l'**interrupt**
- La CPU controlla l'interruzione periodicamente.
- La CPU determina il tipo di interruzione e la gestisce, una volta gestita la CPU riprende l'elaborazione interrotta
 - **Gestore interruzione** invoca la routine di servizio



Il dispositivo I/O invia un segnale su **Interrupt Request Line (IRQ)**. La CPU controlla la IRQ per ogni istruzione.

Per servire la richiesta la CPU:

- Salva le informazioni di stato dell'operazione interrotta (minimo contesto: registri e program counter)
- Trova l'indirizzo della procedura di servizio (**interrupt handler**)
 - Riceve l'**indice** dell'interruzione e lo confronta con il vettore delle interruzioni
 - **Vettore delle interruzioni**: fornisce direttamente indirizzo procedura di servizio (nei processori intel da 0 a 31 i non mascherabili, da 32 a 255 i mascherabili)
- Passa il controllo alla procedura di servizio fornendone l'indirizzo iniziale
 - La procedura di servizio viene eseguita (salva il resto dello stato modificato)
 - La procedura di servizio esce (**interrupt handler exit**)
- Il controllo torna all'operazione interrotta (recuperando lo stato esecutivo)

Nei SO moderni il **controllore hardware di interruzione** fornisce meccanismi di interruzione sofisticati:

- Posticipare la gestione dell'interruzione durante fasi critiche
- Modo efficiente per lanciare il corretto gestore per un dispositivo
- Gestione multilivello (alta priorità, bassa priorità)
- Due tipi di interruzione (*nonmaskable* e *maskable*) su due linee di interrupt
 - **Non mascherabile**: errori irreversibili, alta priorità
 - **Mascherabile**: utilizzata dai controllori di dispositivo, può essere spenta dalla CPU durante le operazioni critiche
- Di solito più servizi che elementi nell'Interrupt vector

- **Interrupt chaining:** numero di interrupt punta ad una lista di interrupt handler

Ciclo di I/O

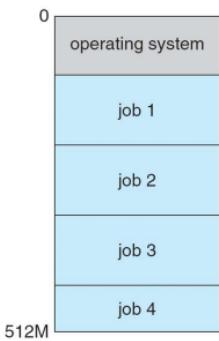
Dopo l'avvio di operazioni di I/O ci sono due modalità di gestione:

- Il controllo torna al programma utente **dopo il completamento I/O**
 - Istruzione *wait* mette in idle (attesa attiva) la CPU fino al prossimo interrupt
 - Ciclo di wait attivo (compete per l'accesso in memoria)
 - Una richiesta pendente alla volta (vantaggio: si conosce il dispositivo che ha richiesto)
- Il controllo torna al programma utente **senza aspettare il completamento**
 - **Tavola dello stato dei dispositivi** entry per ogni I/O che indica tipo, indirizzo e stato
 - Il SO usa tavola dei dispositivi I/O per determinare il dispositivo che ha richiesto l'interruzione, verifica lo stato del dispositivo e modifica la entry per indicare che ha servito l'interruzione.

Caratteristiche del Sistema Operativo

Fra le più importanti caratteristiche dei sistemi operativi vi è la multiprogrammazione.

In generale, un singolo utente non è in grado di tenere costantemente occupata la CPU e i dispositivi di I/O: la **multiprogrammazione** consente di aumentare la percentuale d'utilizzo della CPU, organizzando i lavori in modo tale da mantenerla in continua attività. L'idea su cui si fonda questa tecnica è la seguente: il sistema operativo tiene contemporaneamente in memoria centrale diversi lavori (vedi figura a destra).



Dato che, in genere, la memoria centrale è troppo piccola per contenere tutti i programmi da eseguire, questi vengono collocati inizialmente sul disco in un'area apposita, detta **job pool**, contenente tutti i processi in attesa di essere allocati nella memoria centrale. L'insieme dei programmi caricati in memoria è generalmente un sottoinsieme dei lavori contenuti nel job pool. Il sistema operativo ne sceglie uno tra quelli contenuti nella memoria e inizia l'esecuzione: a un certo punto potrebbe trovarsi nell'attesa di qualche evento, come il completamento di un'operazione di I/O. In questi casi, in un sistema non multiprogrammato, la CPU rimarrebbe inattiva. In un sistema con multiprogrammazione, invece, il sistema operativo passa semplicemente a un altro lavoro e lo esegue. Quando il primo lavoro ha terminato l'attesa, la CPU ne riprende l'esecuzione. Finché c'è almeno un lavoro da eseguire, la CPU non è mai inattiva.

Un sistema con multiprogrammazione fornisce un ambiente in cui le risorse del sistema (per esempio CPU, memoria, dispositivi periferici) sono impiegate in modo efficiente, ma non rappresenta un sistema d'interazione con l'utente. La partizione del tempo d'elaborazione (**time sharing o multitasking**) è un'estensione logica della multiprogrammazione; la CPU esegue più lavori commutando le loro esecuzioni con una frequenza tale da permettere a ciascun utente l'interazione col proprio programma durante la sua esecuzione.

Un sistema di calcolo interattivo permette la comunicazione diretta tra utente e sistema. L'utente impartisce le istruzioni direttamente al sistema operativo oppure a un programma, attraverso una tastiera o un mouse, e attende una risposta immediata. Il tempo di risposta dovrebbe perciò essere breve, in genere meno di un secondo.

Un sistema operativo a partizione del tempo d'elaborazione permette a più utenti di condividere contemporaneamente il calcolatore. Poiché le azioni e i comandi eseguiti in un sistema a partizione del tempo 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 a partizione del tempo d'elaborazione si avvale dello **scheduling della CPU** e della multiprogrammazione. Ciascun utente dispone di almeno un proprio programma in memoria. Un programma caricato in memoria e predisposto per la fase d'esecuzione è noto come processo. Normalmente un processo, durante la sua esecuzione, impegna la CPU per un breve periodo di tempo prima di richiedere operazioni di I/O o di terminare; tali operazioni possono essere interattive, cioè i risultati sono inviati a uno schermo a disposizione dell'utente, che immette dati tramite una tastiera, un mouse, o altro. I tempi di tali operazioni risentono della lentezza del lavoro umano; l'immissione di dati e comandi tramite una tastiera, per esempio, è limitata dalla velocità di battitura dell'utente, che, per elevata che sia, è sempre molto bassa rispetto ai tempi d'elaborazione di un calcolatore: sette caratteri al secondo sono tanti per una persona, ma pochissimi per un calcolatore. Anziché lasciare inattiva la CPU, durante l'immissione interattiva il sistema operativo commuta rapidamente la CPU al programma di un altro utente.

Attività del sistema operativo

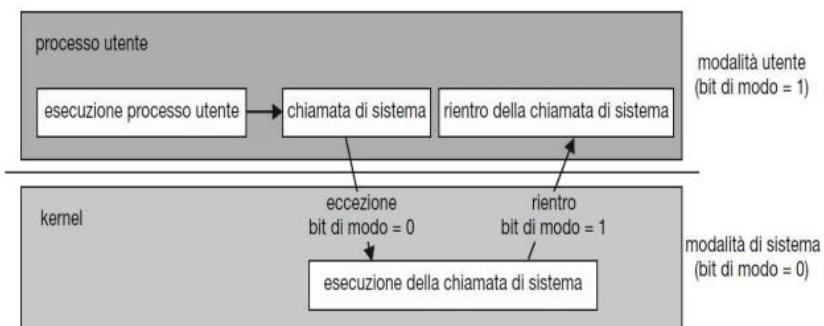
Dal momento che il sistema operativo e gli utenti condividono la dotazione di programmi e dispositivi dell'elaboratore, è necessario assicurarsi che un errore provocato da un programma utente non possa arrecare danno ad altri programmi. In un ambiente condiviso, infatti, l'errore di un solo programma ha un effetto negativo potenziale sugli altri processi. Se, per esempio, un processo rimanesse bloccato in un ciclo infinito, potrebbe essere impedito il corretto funzionamento di molti altri processi. I sistemi multiprogrammati, inoltre, sono esposti a rischi più sottili; si pensi a un programma viziato da errori, capace di modificare interamente un altro programma, i suoi dati e persino lo stesso sistema operativo.

Senza un'efficace protezione da errori come quelli menzionati, il calcolatore è costretto a eseguire un solo processo per volta, a meno di non voler considerare sospetti tutti i dati in uscita. Un sistema operativo progettato in maniera soddisfacente deve evitare che un programma viziato da errori (oppure volutamente dannoso) possa indurre un'esecuzione scorretta di altri programmi.

Modalità Operativa Duale

Per garantire il corretto funzionamento del sistema è necessario distinguere tra l'esecuzione di codice del sistema operativo e di codice definito dall'utente. Il metodo seguito da molti sistemi operativi consiste nel disporre di specifiche caratteristiche dell'architettura del sistema che consentono di gestire differenti modalità di funzionamento.

Sono necessarie almeno due diverse modalità: **user mode** (modalità utente) e **kernel mode** (modalità di sistema). Per indicare quale sia la modalità attiva, l'architettura della CPU deve essere dotata di un bit, chiamato appunto bit di modalità (hardware): di kernel (0) o utente (1). Questo bit consente di stabilire se l'istruzione corrente si esegue per conto del sistema operativo o per conto di un utente. Quando l'elaboratore agisce per conto di un'applicazione utente, il sistema è in modalità utente. Tuttavia, allorquando l'applicazione utente rivolga una richiesta di servizio al sistema operativo (tramite una chiamata di sistema), per soddisfare la richiesta questi deve passare dalla modalità utente alla modalità di sistema. Ciò è esemplificato dalla figura sottostante.



All'avviamento del sistema, il bit è posto in modalità di sistema. Si carica il sistema operativo che provvede all'esecuzione dei processi utenti in modalità utente. Ogni volta che si verifica un'interruzione o un'eccezione si passa dalla modalità utente a quella di sistema, cioè si pone a 0 il bit di modo. Perciò quando il sistema operativo riprende il controllo del calcolatore si trova in modalità di sistema. Prima di passare il controllo al programma utente, il sistema ripristina la modalità utente riportando a 1 il valore del bit.

La duplice modalità di funzionamento (dual mode) consente la protezione del sistema operativo rispetto al comportamento degli utenti e viceversa. Questo livello di protezione si ottiene definendo come **istruzioni privilegiate** le istruzioni di macchina che possono causare danni allo stato del sistema. Poiché la CPU consente l'esecuzione di queste istruzioni soltanto nella modalità di sistema, se si tenta di far eseguire in modalità utente un'istruzione privilegiata, la CPU non la esegue, ma la tratta come un'istruzione illegale inviando un segnale di eccezione al sistema operativo.

Protezione CPU

Occorre assicurare che il sistema operativo mantenga il controllo dell'elaborazione, cioè 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, programmabile affinché invii un segnale d'interruzione alla CPU a intervalli di tempo specificati, che possono essere fissi o variabili. Un **timer variabile** di solito si realizza mediante un generatore di impulsi 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.

Prima di restituire all'utente il controllo dell'esecuzione, il sistema assegna un valore al timer. Se esso esaurisce questo intervallo genera un'interruzione che causa il trasferimento del controllo al sistema operativo, che può decidere se gestire le interruzioni come un errore fatale o concedere altro tempo al programma. Ovviamente, anche le istruzioni usate dal sistema per modificare il funzionamento del timer si possono eseguire soltanto in modalità sistema.

La presenza di un timer garantisce quindi che nessun programma utente possa essere eseguito troppo a lungo. Una tecnica semplice consiste nell'impostare un contatore con un valore pari al tempo concesso al programma per la propria esecuzione. Il timer genererà un'interruzione ogni secondo e il contatore sarà decrementato di 1; fintanto che il valore resta positivo, il controllo ritorna al programma utente; quando il contatore raggiunge un valore negativo, il sistema operativo termina l'esecuzione del programma per il superamento del tempo a esso assegnato.

Protezione Memoria

Occorre anche proteggere la modalità di esecuzione privilegiata poiché sovrascrivendo il vettore delle interruzioni e le operazioni di servizio si potrebbe trasferire la modalità privilegiata all'utente. Duque, occorrono meccanismi di protezione hardware della memoria.

Per stabilire gli indirizzi di memoria a cui un programma può accedere si possono utilizzare due registri:

- **Registro base**: più piccolo indirizzo accessibile
- **Registro limite**: dimensione del range di indirizzi

Gestione Processi

Un processo è un programma in esecuzione. Il programma è una **entità passiva** (come il contenuto di un file memorizzato in un disco), il processo è una **entità attiva** (come il program counter).

Per svolgere i propri compiti, un processo necessita di alcune risorse, tra cui tempo di CPU, memoria, file e dispositivi di I/O. Queste risorse si possono attribuire al processo al momento della sua creazione, oppure si possono assegnare durante l'esecuzione. Oltre alle diverse risorse fisiche e logiche assegnate a un processo

durante la sua creazione, si possono considerare anche alcuni dati d'inizializzazione da passare di volta in volta al processo stesso. Per esempio, un processo che ha lo scopo di mostrare su uno schermo lo stato di un file, riceve il nome del file ed esegue le appropriate istruzioni e chiamate di sistema che gli consentono di ottenere e mostrare sullo schermo le informazioni desiderate; quando il processo termina, il sistema operativo riprende il controllo delle risorse impiegate dal processo.

L'esecuzione di un processo deve essere sequenziale: la CPU esegue le istruzioni del processo una dopo l'altra, finché il processo termina; infatti, i processi **single-threaded** hanno un **program counter** (locazione prossima istruzione da eseguire). Un processo **multi-threaded**, invece, possiede più program counter, ognuno dei quali punta all'istruzione successiva da eseguire per un dato thread. Tipicamente i sistemi hanno in esecuzione molti processi, per più utenti, in esecuzione concorrente su una o più CPU.

Il sistema operativo è responsabile delle seguenti attività connesse alla gestione dei processi. Dunque, il SO fornisce diversi meccanismi per la gestione dei processi, in particolare:

- creazione e cancellazione dei processi utenti e di sistema;
- sospensione e ripristino dei processi;
- fornitura di meccanismi per la sincronizzazione dei processi;
- fornitura di meccanismi per la comunicazione tra processi;
- fornitura di meccanismi per la gestione delle situazioni di stallo (**deadlock**).

Chiamata	Descrizione
<code>pid = fork()</code>	Crea un processo figlio identico al genitore
<code>pid = waitpid(pid, &statloc, options)</code>	Attende la terminazione di un processo figlio
<code>s = execve(name, argv, environp)</code>	Sostituisce il processo corrente con quelli di un altro programma contenuto in un file specificato
<code>exit(status)</code>	Termina il processo in esecuzione e restituisce lo stato

Process management

Gestione Memoria

La memoria è un magazzino di dati velocemente accessibile ed è condivisa dalla CPU e da alcuni dispositivi di I/O. La CPU legge le istruzioni dalla memoria centrale durante il ciclo di prelievo delle istruzioni, oltre a leggere e scrivere i dati nella memoria centrale durante il ciclo d'accesso ai dati. Generalmente la memoria centrale è l'unico ampio dispositivo di memorizzazione a cui la CPU può far riferimento e accedere in modo diretto.

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

- tenere traccia di quali parti di memoria sono attualmente usate e da chi;
- decidere quali processi (o parti di essi) e dati debbano essere caricati in memoria o trasferiti altrove;
- assegnare e revocare lo spazio di memoria secondo le necessità

Gestione Archiviazione

Per facilitare gli utenti, il sistema operativo fornisce un'interfaccia logica uniforme per la memorizzazione delle informazioni. Esso, cioè, prescinde dalle caratteristiche fisiche dei dispositivi di memorizzazione, definendo un'unità logica di archiviazione, il file. Il sistema operativo associa i file a supporti fisici, e vi accede tramite i dispositivi di memorizzazione delle informazioni.

La gestione dei file è uno dei componenti più visibili di un sistema operativo. I calcolatori possono registrare le informazioni su molti mezzi fisici diversi; i più diffusi sono il nastro magnetico, il disco magnetico e il disco ottico. Ciascuno ha caratteristiche proprie e una propria organizzazione fisica, ed è controllato da un

dispositivo, come un'unità a disco o a nastro, avente anch'esso caratteristiche proprie: velocità, capacità, rapidità nel trasferimento dei dati, metodi d'accesso (diretto o sequenziale).

Un **file** è una raccolta d'informazioni correlate definite dal loro creatore. Comunemente, i file rappresentano programmi, sia sorgente sia oggetto, e dati. I file di dati possono essere numerici, alfabetici, alfanumerici o binari; la loro forma può essere libera, come nei file di testo, oppure rigidamente formattata, per esempio in campi fissi. Il concetto di file è chiaramente molto generale. Il sistema operativo realizza il concetto astratto di file gestendo i mezzi di memoria di massa, come nastri e dischi, e i dispositivi che li controllano. 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).

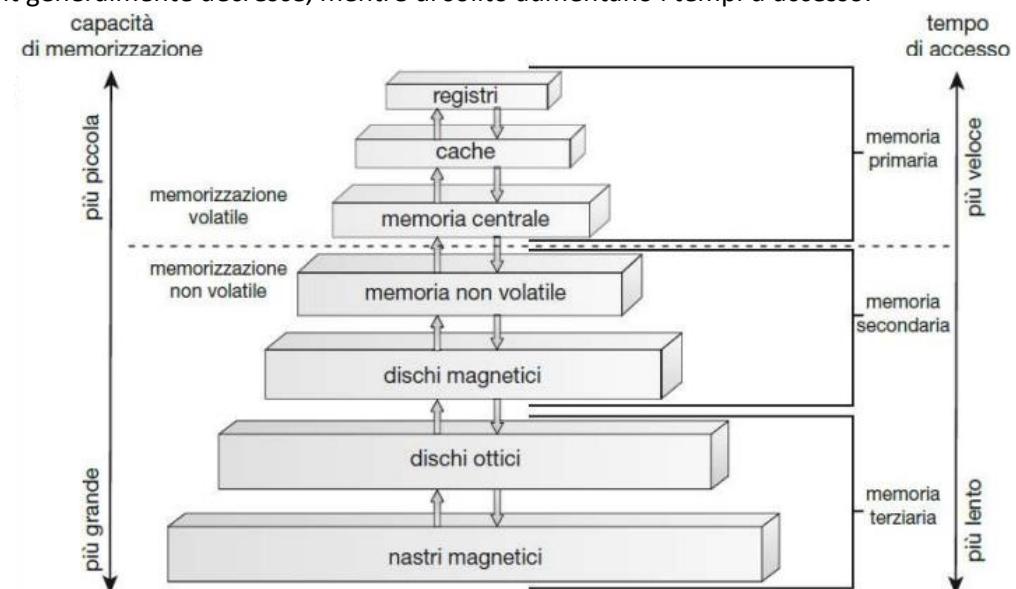
Il sistema operativo è responsabile delle seguenti attività connesse alla gestione dei file:

- creazione e cancellazione di 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.

Strutture di Memoria

Lo scopo principale di un sistema di calcolo è quello di eseguire programmi. Durante l'esecuzione, i programmi, insieme con i dati cui accedono, devono trovarsi nella memoria centrale. Giacché 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 di calcolo 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, le procedure di ordinamento e gli editor. Questi programmi rimangono nel disco fino al momento del caricamento in memoria e si servono del disco sia come sorgente sia come destinazione delle loro computazioni; per tale ragione è fondamentale che la registrazione nei dischi sia gestita correttamente. Il sistema operativo è responsabile delle seguenti attività connesse alla gestione dei dischi: gestione dello spazio libero; assegnazione dello spazio; scheduling disco.

I sistemi di memorizzazione di un calcolatore si possono organizzare in modo gerarchico secondo la velocità e il costo. I livelli più alti rappresentano i dispositivi più rapidi, ma più costosi. Scendendo nella gerarchia, il costo per bit generalmente decresce, mentre di solito aumentano i tempi d'accesso.



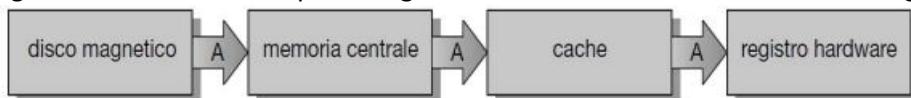
Il frequente uso della memoria secondaria impone una sua gestione efficiente. Infatti, l'efficienza complessiva di un calcolatore può dipendere dalla velocità del sottosistema di gestione dei dischi e dagli algoritmi che lo gestiscono. Vi sono però svariati frangenti in cui tornano utili dispositivi di memorizzazione più lenti, meno costosi e talora più capienti della memoria secondaria. Le copie di riserva dei dischi di sistema, i dati usati raramente e gli archivi a lungo termine sono alcuni esempi. Le unità a nastro magnetico, i CD e i DVD, sono tipici dispositivi di memoria terziaria. I media (nastri e i dischi ottici) comprendono i formati monoscrivibili WORM (*write once, read many times*) e riscrivibili RW (*read and write*). La memoria terziaria ha scarso impatto sulle prestazioni del sistema, ma deve pur essere gestita. Alcuni sistemi si assumono tale onere direttamente, mentre altri lo delegano a programmi applicativi.

Cache

Il concetto di cache è un principio importante valido a diversi livelli (hardware, SO, software). Il caching è un meccanismo che velocizza alcune operazioni; infatti, di norma le informazioni sono mantenute in unità di memoria come la memoria centrale; al momento del loro uso si copiano temporaneamente in un'unità più veloce: la cache. Quando si deve accedere a una particolare informazione, innanzitutto si controlla se è già presente all'interno della cache; in tal caso si adopera direttamente la copia contenuta nella cache, altrimenti la si preleva dalla memoria centrale e la si copia nella cache, poiché si suppone che questa informazione presto servirà ancora.

Data la capacità limitata di questi dispositivi, la **gestione della cache** è un importante problema di progettazione. Da un'attenta selezione delle dimensioni e dei criteri di aggiornamento della cache può conseguire un notevole incremento delle prestazioni del sistema.

Il movimento delle informazioni tra i vari livelli della gerarchia può essere sia implicito sia esplicito, secondo l'architettura e il sistema operativo che la controlla. Per esempio, il trasferimento dei dati tra la cache e i registri della CPU è di solito svolto dall'architettura del sistema senza alcun intervento del sistema operativo. Diversamente, il trasferimento dei dati dai dischi alla memoria è di solito gestito dal sistema operativo. Di seguito forniamo un esempio di migrazione di un intero *A* da un disco a un registro:



Sistemi di I/O

Uno degli scopi di un sistema operativo è nascondere all'utente le caratteristiche degli specifici dispositivi. In UNIX, per esempio, le caratteristiche dei dispositivi di I/O sono nascoste alla maggior parte dello stesso sistema operativo dal sottosistema di I/O, che è composto delle seguenti parti:

- un componente di gestione della memoria comprendente la gestione delle regioni della memoria riservate ai trasferimenti di I/O (*buffer*), la gestione delle cache e la gestione asincrona delle operazioni di I/O e dell'esecuzione di più processi (*spooling*);
- un'interfaccia generale per i driver dei dispositivi;
- i driver per gli specifici dispositivi.

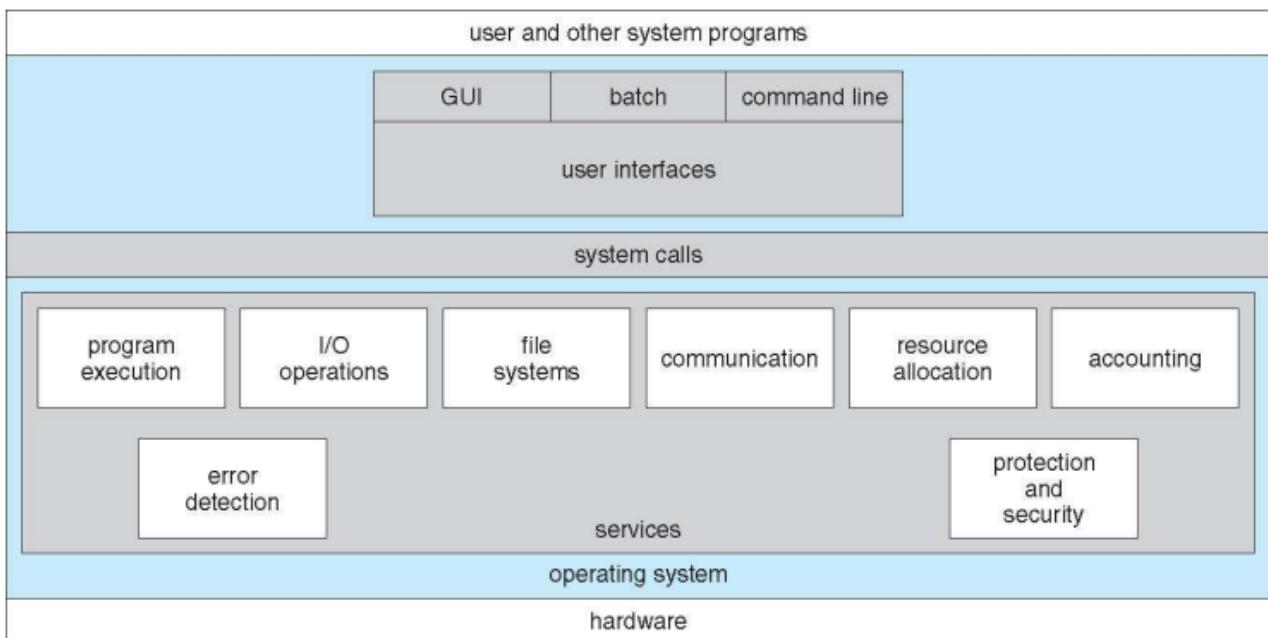
Soltanto il driver del dispositivo conosce le caratteristiche dello specifico dispositivo cui è assegnato.

2. Strutture dei Sistemi Operativi

Servizi di un SO

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.

Forniamo prima di tutto una panoramica dei servizi del sistema operativo e delle loro correlazioni:



Ogni insieme di servizi offre funzionalità utili all'utente.

- **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 e modificarli come ad esempio un programma apposito. Un'interfaccia batch, invece, prevede che comandi e relative direttive siano codificati nei file, eseguiti successivamente a lotti. La forma senz'altro più diffusa è l'interfaccia grafica con l'utente (GUI), ossia un sistema grafico a finestre dotato di un dispositivo puntatore 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 anormale (c'è un 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**: 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 programmi, infine, devono poter gestire 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:** I processi possono scambiare informazioni, sullo stesso computer o tra computer in una rete. La comunicazione si può realizzare tramite una memoria condivisa o attraverso lo scambio di messaggi, in questo caso il sistema operativo trasferisce pacchetti d'informazioni tra i vari processi.
- **Rilevamento d'errori:** Il sistema operativo deve essere sempre capace di rilevare eventuali errori che possono verificarsi nella CPU e nei dispositivi di memoria, quali 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; 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. Naturalmente sistemi operativi differenti reagiscono agli errori e vi pongono riparo in modi diversi. Eventuali funzionalità di debug (cioè, degli strumenti che permettano l'analisi, per esempio, del software che ha causato un errore) aumentano di molto le possibilità dei programmati e degli utenti di usare il sistema efficientemente.

Esiste anche un'altra serie di funzioni del sistema operativo che non riguarda direttamente gli utenti, ma assicura il funzionamento efficiente del sistema stesso. Sistemi con più utenti possono guadagnare in efficienza condividendo le risorse del calcolatore tra i diversi utenti.

- **Assegnazione delle risorse:** Se sono in corso più sessioni di lavoro di utenti o sono contemporaneamente in esecuzione più processi, il sistema operativo provvede all'assegnazione delle risorse necessarie a ciascuno di essi. Alcune di queste risorse, come i cicli di CPU, la memoria centrale e la registrazione in file, possono avere un codice di assegnazione speciale, mentre altre, come i dispositivi di I/O, possono avere un codice di richiesta e di rilascio più generale.
- **Contabilizzazione dell'uso delle risorse:** È possibile registrare quali utenti usino il calcolatore, segnalando quali e quante risorse impieghino.
- **Protezione e sicurezza:** I proprietari di informazioni memorizzate in un sistema di calcolo multiutente o in rete possono voler controllare l'uso di tali informazioni. Più processi non correlati e in esecuzione concorrente 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 l'obbligo d'identificazione da parte di ciascun utente, di solito attraverso parole d'ordine che permettono l'accesso alle risorse; si estende alla "difesa" dei dispositivi di I/O dai tentativi d'accesso illegali e provvede al loro rilevamento.

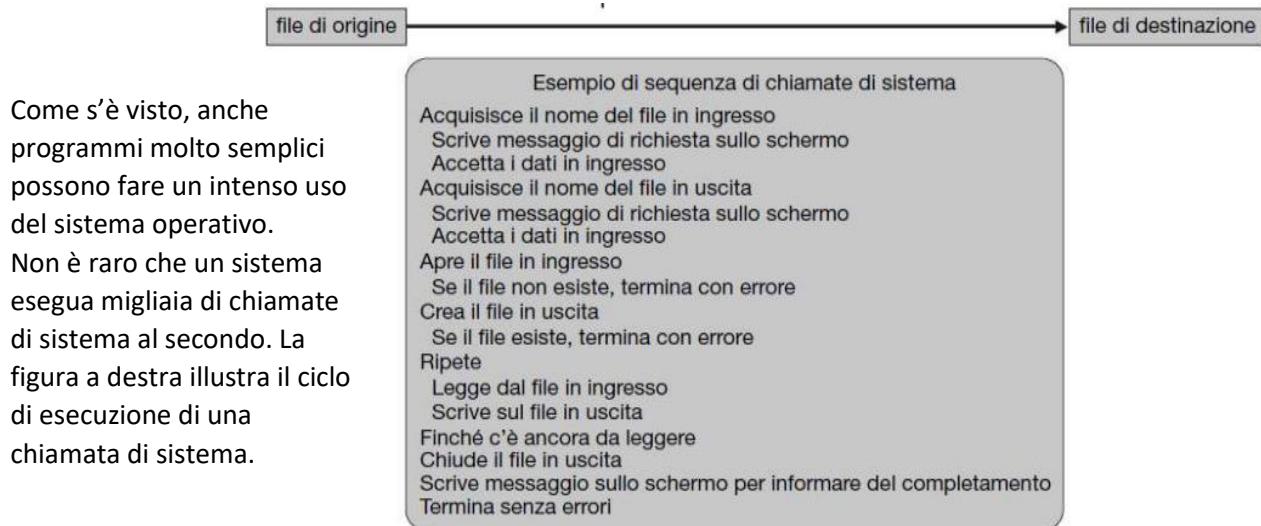
Chiamate di sistema

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

Prima di illustrare come le chiamate di sistema vengano rese disponibili da parte del sistema operativo, consideriamo come esempio la scrittura di un semplice programma che legga i dati da un file e li trascriva in un altro. La prima informazione di cui il programma necessita è costituita dai nomi dei due file: il file in ingresso e il file in uscita. Questi file si possono indicare in molti modi diversi, secondo la struttura del sistema operativo. Una volta ottenuti i nomi, il programma deve aprire il file in ingresso e creare il file di destinazione. Ciascuna di queste operazioni richiede un'altra chiamata di sistema e può andare incontro a condizioni d'errore. Per esempio, quando il programma tenta di aprire il file in ingresso, può scoprire che non esiste alcun file con quel nome, oppure che l'accesso al file è negato. In questi casi il programma deve

scrivere un messaggio nello schermo della console (altra sequenza di chiamate di sistema) e quindi terminare in maniera anormale la propria elaborazione (ulteriore chiamata di sistema). Se il file in ingresso esiste, è necessario creare il file di destinazione. È possibile che esista già un file col nome indicato per il file di destinazione; questa situazione potrebbe causare l'interruzione del programma (una chiamata di sistema) o la cancellazione del file esistente (un'altra chiamata di sistema) e la creazione di uno nuovo.

Una volta predisposti i due file, si entra in un ciclo che legge dal file in ingresso (una chiamata di sistema) e scrive nel file di destinazione (altra chiamata di sistema). Ciascuna lettura (*read*) e ogni scrittura (*write*) deve riportare informazioni di stato relative alle possibili condizioni d'errore. Infine, una volta copiato tutto il file, il programma può chiuderli entrambi (altre chiamate di sistema), inviare un messaggio alla console (più chiamate di sistema) e infine terminare normalmente (ultima chiamata di sistema).



Relazione tra API e Chiamata

La maggior parte dei programmati, tuttavia, non si dovrà mai preoccupare di questi dettagli: infatti, gli sviluppatori usano in genere un'interfaccia per la programmazione di applicazioni (**API**, *application programming interface*). Essa specifica un insieme di funzioni a disposizione dei programmati, e dettaglia i parametri necessari all'invocazione di queste funzioni, insieme ai valori restituiti.

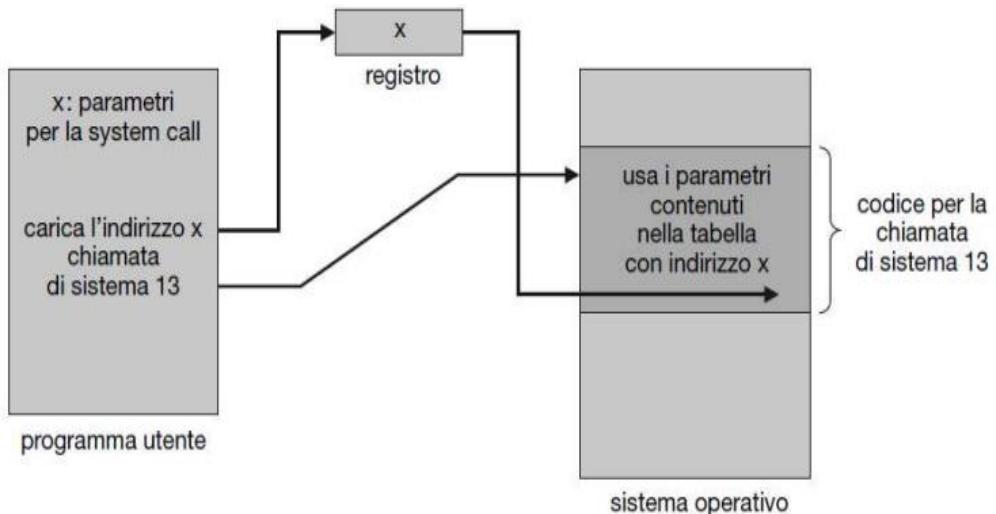
Il cosiddetto sistema di supporto all'esecuzione (run-time support system) di un linguaggio di programmazione, ossia l'insieme di funzioni strutturate in librerie incluse nel compilatore, fornisce nella gran parte dei casi un'interfaccia alle chiamate di sistema rese disponibili dal sistema operativo, che funge da raccordo tra il linguaggio e il sistema stesso. L'interfaccia intercetta le chiamate di sistema invocate dalla API, e richiede effettivamente la chiamata necessaria. Di solito, ogni chiamata di sistema è codificata da un numero; il compilatore mantiene una tabella delle chiamate di sistema, cui si accede usando questi numeri come indici. L'interfaccia alle chiamate di sistema invoca di volta in volta la chiamata richiesta, che risiede nel kernel del sistema, e passa al chiamante i valori restituiti dalla chiamata di sistema, inclusi quelli di stato. Il chiamante non ha alcuna necessità di conoscere alcunché sull'implementazione della chiamata di sistema o del suo ciclo esecutivo: gli è sufficiente riconoscere la API e il risultato dell'esecuzione della chiamata di sistema operativo. Ne consegue che la gran parte dei dettagli relativi alle chiamate di sistema sono nascosti al programmatore dalla API, e gestiti dal sistema di supporto all'esecuzione.

Passaggio di parametri

Le chiamate di sistema si presentano in modi diversi, secondo il calcolatore in uso. Spesso sono richieste maggiori informazioni oltre alla semplice identità della chiamata di sistema desiderata. Il tipo e l'entità esatti delle informazioni variano secondo lo specifico sistema operativo e la specifica chiamata di sistema. Per ottenere l'immissione di un dato, per esempio, può essere necessario specificare il file o il dispositivo

da usare come sorgente e anche l'indirizzo e l'ampiezza dell'area della memoria in cui depositare i dati letti. Naturalmente, il dispositivo o il file e l'ampiezza possono essere impliciti nella chiamata di sistema.

Per passare parametri al sistema operativo si usano tre metodi generali. Il più semplice consiste nel passare i parametri in registri; si possono però presentare casi in cui vi sono più parametri che registri. In questi casi generalmente si memorizzano i parametri in un blocco o tabella di memoria e si passa l'indirizzo del blocco, in forma di parametro, in un registro (figura in basso). E il metodo seguito dal sistema operativo Linux. Il programma può anche collocare (*push*) i parametri in una pila da cui sono prelevati (*pop*) dal sistema operativo. Alcuni sistemi operativi preferiscono i metodi del blocco o della pila, poiché non limitano il numero o la lunghezza dei parametri da passare.



Tipi di chiamate di sistema

- Controllo dei processi
 - terminazione normale e anormale
 - caricamento, esecuzione
 - creazione e arresto di un processo
 - esame e impostazione degli attributi di un processo
 - attesa per il tempo indicato
 - attesa e segnalazione di un evento
 - assegnazione e rilascio di memoria
- Gestione dei file
 - creazione e cancellazione di file
 - apertura, chiusura
 - lettura, scrittura, posizionamento
 - esame e impostazione degli attributi di un file
- Gestione dei dispositivi
 - richiesta e rilascio di un dispositivo
 - lettura, scrittura, posizionamento
 - esame e impostazione degli attributi di un dispositivo
 - inserimento logico ed esclusione logica di un dispositivo
- Gestione delle informazioni
 - esame e impostazione dell'ora e della data
 - esame e impostazione dei dati del sistema
 - esame e impostazione degli attributi dei processi, file e dispositivi
- Comunicazione
 - creazione e chiusura di una connessione
 - invio e ricezione di messaggi
 - informazioni sullo stato di un trasferimento
 - inserimento ed esclusione di dispositivi remoti
- Protezione
 - controllo accesso a risorse
 - gestione dei permessi
 - gestione accesso utenti

Programmi di sistema

I programmi di sistema, conosciuti anche come 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, ridenominano, 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 o informazioni di stato. Altri, più complessi, forniscono informazioni dettagliate su prestazioni, accessi al sistema e debug. In genere mostrano le informazioni tramite terminale, o tramite altri dispositivi per l'uscita dei dati, o, ancora, all'interno di una finestra della GUI. Alcuni sistemi comprendono anche registri, al fine di archiviare e poter poi consultare informazioni sulla configurazione del sistema.
- **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, programmi per la correzione degli errori e interpreti dei comuni linguaggi di programmazione, come C, C++, Java, Visual Basic e PERL, sono spesso forniti insieme con il sistema operativo.
- **Caricamento ed esecuzione dei programmi:** Una volta assemblato o compilato, per essere eseguito, un programma deve essere caricato in memoria. Il sistema può mettere a disposizione caricatori assoluti, caricatori rilocabili, editor dei collegamenti (*linkage editor*) e caricatori di sezioni sovrapponibili di programmi (*overlay loader*). Sono necessari anche i sistemi d'ausilio all'individuazione e correzione degli errori (*debugger*) per i linguaggi d'alto livello o per il linguaggio macchina.
- **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 accedere a calcolatori remoti, di trasferire file da un calcolatore a un altro.

Progettazione e realizzazione di un SO

Per progettare un SO bisogna distinguere la **politica** (cosa il sistema deve fare) dal **meccanismo** (come lo deve fare). Esempio: il timer serve per la protezione della CPU; il timer è un meccanismo ma la lunghezza del timer è una politica.

La separazione della politica dal meccanismo è molto importante ai fini della flessibilità. I criteri sono soggetti a cambiamenti di luogo o di tempo. Nei casi peggiori il cambiamento di un criterio può richiedere il cambiamento del meccanismo sottostante. Sarebbe preferibile disporre di meccanismi generali: in questo caso un cambiamento di criterio implicherebbe solo la ridefinizione di alcuni parametri del sistema.

Specificare e progettare un sistema operativo è un compito altamente creativo per l'ingegneria del software.

Implementazione

Una volta progettato, un sistema operativo va realizzato. Tradizionalmente i sistemi operativi si scrivevano in un linguaggio assembly, attualmente si scrivono spesso in linguaggi di alto livello come il C o il C++. Di solito sono un mix dove le parti più basso livello sono in assembly mentre il corpo principale in C/C++.

I vantaggi derivanti dall'uso di un linguaggio di alto livello, o perlomeno di un linguaggio orientato in modo specifico allo sviluppo di sistemi, 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.

Struttura del Sistema Operativo

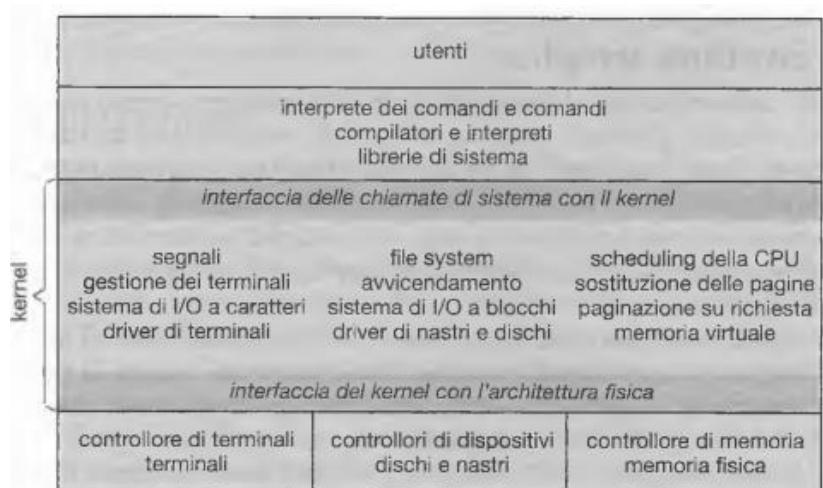
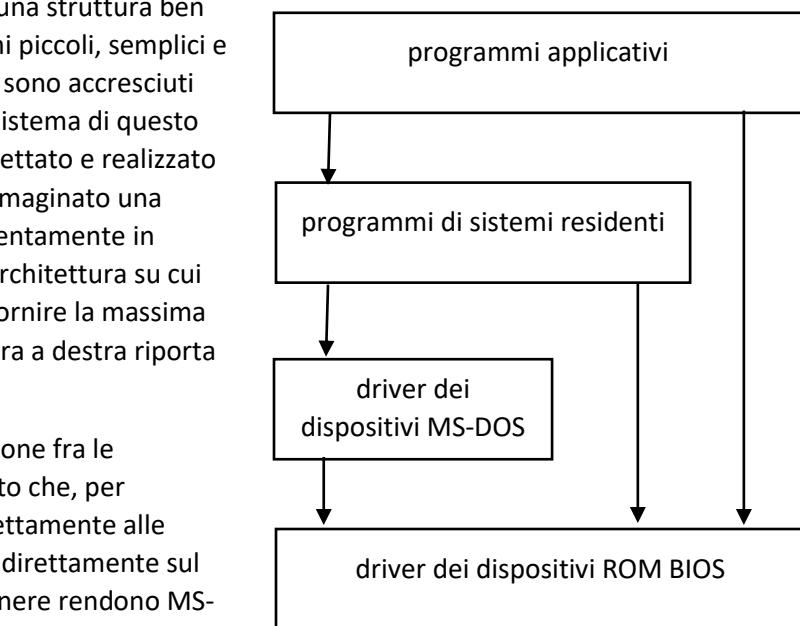
Affinché possa funzionare correttamente ed essere facilmente modificato, un sistema vasto e complesso come un sistema operativo moderno va progettato con estrema attenzione. Anziché progettare un sistema monolitico, un orientamento diffuso prevede la sua suddivisione in piccoli componenti; ciascuno deve essere un modulo ben definito del sistema, con interfacce e funzioni definite con precisione.

Struttura semplice

Molti sistemi commerciali non hanno una struttura ben definita; spesso sono nati come sistemi piccoli, semplici e limitati, e solo in un secondo tempo si sono accresciuti superando il loro scopo originale. Un sistema di questo tipo è l'MS-DOS, originariamente progettato e realizzato da persone che non avrebbero mai immaginato una simile diffusione. Non fu suddiviso attentamente in moduli poiché, a causa dei limiti dell'architettura su cui era eseguito, lo scopo prioritario era fornire la massima funzionalità nel minimo spazio. La figura a destra riporta la sua struttura.

In MS-DOS non vi è una netta separazione fra le interfacce e i livelli di funzionalità, tanto che, per esempio, le applicazioni accedono direttamente alle routine di sistema per l'I/O, scrivendo direttamente sul video e sui dischi. Libertà di questo genere rendono MS-DOS vulnerabile agli errori e agli attacchi dei programmi utenti, fino al blocco totale del sistema.

Anche il sistema UNIX originale è poco strutturato, a causa delle limitazioni dell'hardware disponibile al tempo della sua progettazione. Il sistema consiste di due parti separate, il kernel e i programmi di sistema. A sua volta, il kernel è diviso in una serie di interfacce e driver dei dispositivi, aggiunti ed espansi nel corso dell'evoluzione di UNIX. La figura in basso mostra i diversi livelli di UNIX: tutto ciò che sta al di sotto dell'interfaccia alle chiamate di sistema e al di sopra dell'hardware costituisce il kernel. Esso comprende il file system, lo scheduling della CPU, la gestione della memoria, e le altre funzionalità del sistema rese disponibili tramite chiamate di sistema. Tutto sommato, si tratta di un'enorme massa di funzionalità diverse combinate in un solo livello. È questa struttura monolitica che

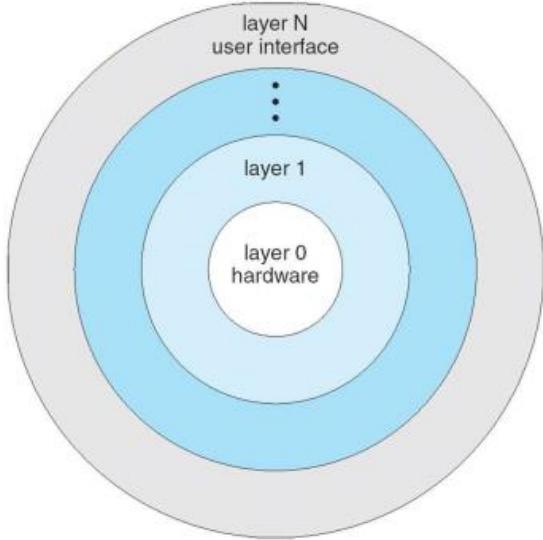


rendeva difficile l'implementazione e la manutenzione.

Metodo stratificato

Il sistema è suddiviso in un certo numero di livelli o strati: il più basso corrisponde all'hardware (*layer 0*), il più alto all'interfaccia con l'utente (*layer N*). Si veda figura.

Lo strato di un sistema operativo è una realizzazione di un oggetto astratto, che incapsula i dati e le operazioni che trattano tali dati. Un tipico strato di sistema operativo (chiamato *M*) è composto da strutture dati e da un insieme di routine richiamabili dagli strati di livello più alto. Lo strato *M*, a sua volta, è in grado di invocare operazioni dagli strati di livello inferiore. Il vantaggio principale offerto da questo metodo è dato dalla semplicità di progettazione e dalle funzionalità di debug. Gli strati sono composti in modo che ciascuno usi solo funzioni (o operazioni) e servizi che appartengono a strati di livello inferiore. Questo metodo semplifica il debugging e la verifica del sistema.



Microkernel

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. In generale, un microkernel offre i servizi minimi di gestione dei processi, della memoria e di comunicazione.

Lo scopo principale del microkernel è fornire funzioni di comunicazione tra i programmi client e i vari servizi, anch'essi in esecuzione nello spazio utente. 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: i nuovi servizi si aggiungono allo spazio utente e non comportano modifiche al kernel. Poiché è ridotto all'essenziale, se il kernel deve essere modificato, i cambiamenti da fare sono ben circoscritti, e il sistema operativo risultante è più semplice da adattare alle 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.

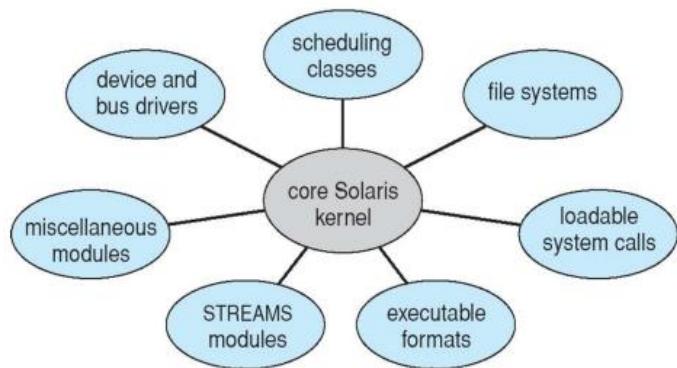
Purtroppo, i microkernel possono incorrere in cali di prestazioni dovuti al sovraccarico indotto dall'esecuzione di processi utente con funzionalità di sistema. La prima versione di Windows NT, basata su un microkernel stratificato, aveva prestazioni inferiori rispetto a Windows 95. La versione 4.0 di Windows NT risolse parzialmente il problema, spostando strati dal livello utente al livello kernel, e integrandoli più strettamente. Questa tendenza fece sì che, al tempo della progettazione di Windows XP, l'architettura di NT era ormai monolitica.

Moduli

Forse il miglior approccio attualmente disponibile per la progettazione dei sistemi operativi si fonda su tecniche della programmazione orientata agli oggetti per implementare un kernel modulare. In questo contesto, il kernel è costituito da un insieme di componenti fondamentali, integrati poi da funzionalità

aggiunte dinamicamente durante l'avvio o l'esecuzione. Questa strategia, che impiega moduli caricati dinamicamente, è comune nelle implementazioni moderne di UNIX, come Solaris, Linux e Mac OS X. La struttura di Solaris, per esempio, illustrata dalla figura, si incentra su un kernel dotato dei seguenti sette tipi di moduli caricabili:

1. Classi di scheduling.
2. File system.
3. Chiamate di sistema caricabili.
4. Formati eseguibili.
5. Moduli STREAMS.
6. Varie.
7. Driver dei dispositivi e del bus.



Questa organizzazione lascia la possibilità al kernel di fornire i servizi essenziali, ma permette anche di implementare dinamicamente certe caratteristiche. È possibile aggiungere driver per dispositivi specifici, per esempio, o gestire file system diversi tramite moduli caricabili. Il risultato complessivo somiglia ai sistemi a strati, perché ogni parte del kernel ha interfacce ben definite e protette. E tuttavia più flessibile dei sistemi a strati tradizionali, perché ciascun modulo può invocare funzionalità di un qualunque altro modulo. Inoltre, come nei sistemi basati su microkernel, il modulo principale gestisce solo i servizi essenziali, oltre a poter caricare altri moduli e comunicare con loro. E tuttavia, rispetto ai sistemi orientati a un microkernel, l'efficienza è superiore, perché i moduli sono in grado di comunicare senza invocare le funzionalità di trasmissione dei messaggi.

Struttura ibrida

La maggior parte dei sistemi operativi moderni non sono in realtà un modello puro bensì combinano più approcci per soddisfare le esigenze di prestazioni, sicurezza e usabilità.

Il sistema operativo Mac OS X di Apple adotta una struttura ibrida; è organizzato in strati, uno dei quali contiene il microkernel Mach. Gli strati superiori comprendono gli ambienti esecutivi delle applicazioni e un insieme di servizi che offre un'interfaccia grafica con le applicazioni. Il kernel si trova in uno strato sottostante, ed è costituito dal microkernel Mach e dal kernel BSD. Il primo cura la gestione della memoria, le chiamate di procedure remote (RPC), la comunicazione tra processi (IPC) - compreso lo scambio di messaggi - e lo scheduling dei thread. Il secondo mette a disposizione un'interfaccia BSD a riga di comando, i servizi legati al file system e alla rete, e la API POSIX, compreso Pthreads. Oltre a Mach e BSD, il kernel possiede un kit di strumenti connessi all'I/O per lo sviluppo di driver dei dispositivi e di moduli dinamicamente caricabili, detti estensioni del kernel nel gergo di Mac OS X.

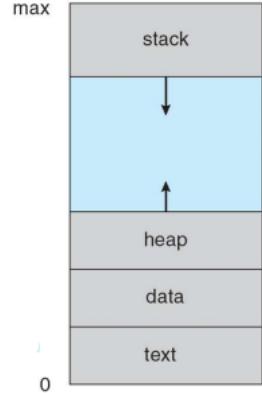
3. Processi

Concetto di processo

Una questione che sorge dall'analisi dei sistemi operativi è la definizione delle attività della CPU. Un **sistema a lotti (batch)** esegue **lavori (job)**, mentre un sistema a partizione del tempo (*time shared*) esegue **programmi utenti o task**. In questi appunti job e processo sono usati in modo quasi intercambiabile (sebbene si preferisca il termine di processo).

Processo

Informalmente, un processo è un programma in esecuzione. È qualcosa di più del codice di un programma, talvolta noto anche come **sezione di testo**: comprende l'attività corrente, rappresentata dal valore del **program counter** (contatore di programma) e dal contenuto dei registri della CPU; normalmente comprende anche il proprio **stack**, contenente a sua volta i dati temporanei, come i parametri di un metodo, gli indirizzi di rientro e le variabili locali, e una **sezione di dati** contenente le variabili globali. Un processo può includere uno **heap**, ossia della memoria dinamicamente allocata durante l'esecuzione del processo. La struttura di un processo in memoria è illustrata in figura.

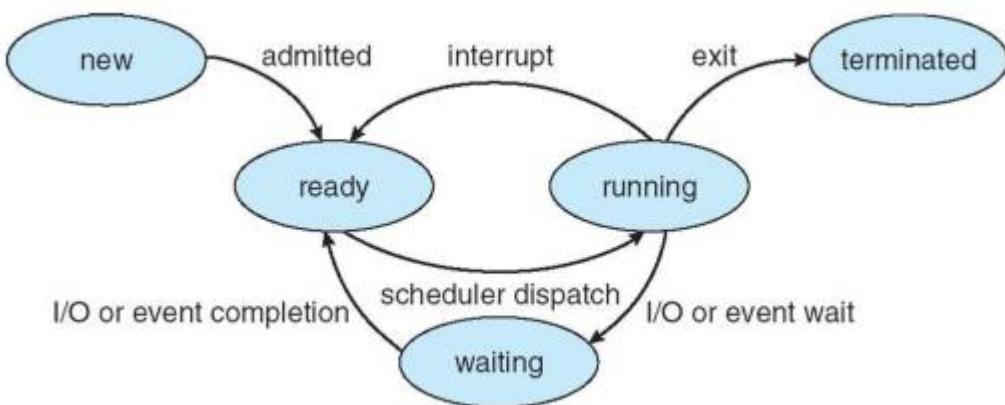


Sottolineiamo che un programma di per sé non è un processo; bensì, un'entità **passiva**, come il contenuto di un file memorizzato in un disco, mentre un processo è un'entità **attiva**, con un program counter che specifica qual è l'istruzione successiva da eseguire e un insieme di risorse associate. Un programma diventa un processo allorquando il file eseguibile che lo contiene è caricato in memoria (ad uno stesso programma possono corrispondere diversi processi tutti con una propria stack di esecuzione).

Stato del processo

Un processo durante l'esecuzione è soggetto a cambiamenti di **stato**, definiti in parte dall'attività corrente del processo stesso. Ogni processo può trovarsi in uno tra i seguenti stati.

- **new**: Si crea il processo
- **running**: Un'unità d'elaborazione esegue le istruzioni del relativo programma
- **waiting**: il processo attende che si verifichi qualche evento (come il completamento di un'operazione di I/O o la ricezione di un segnale)
- **ready**: il processo attende di essere assegnato ad un'unità d'elaborazione
- **terminated**: il processo ha terminato l'esecuzione

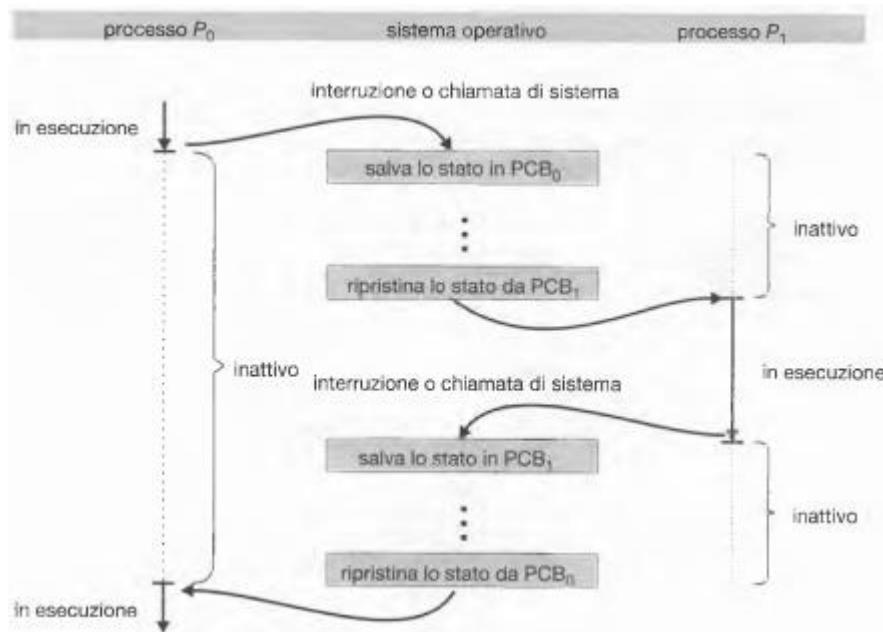


Descrittore di processo

Il sistema operativo mantiene una **tabella dei processi**. Ogni processo è rappresentato nel sistema operativo da un **blocco di controllo di un processo** anche conosciuto come PCB (*process control block*) o

TCB (*task control block*). Un TCB contiene molte informazioni connesse a un processo specifico, tra cui le seguenti.

- **Stato del processo:** Lo stato può essere: new, ready, running, waiting, terminated.
- **Program counter:** Il contatore di programma contiene l'indirizzo della successiva istruzione da eseguire per tale processo.
- **Registri di CPU:** I registri variano in numero e tipo secondo l'architettura del calcolatore. Essi comprendono accumulatori, registri d'uso generale e registri contenenti informazioni relative ai codici di condizione. Quando si verifica un'interruzione della CPU, si devono salvare tutte queste informazioni insieme con il contatore di programma, in modo da permettere la corretta esecuzione del processo in un momento successivo (la figura rappresenta come la CPU può essere commutata tra i processi)



- **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 si possono esprimere attraverso i valori 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 il tempo d'uso della CPU e il tempo reale 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 TCB si usa semplicemente come deposito per tutte le informazioni relative ai vari processi.

Thread

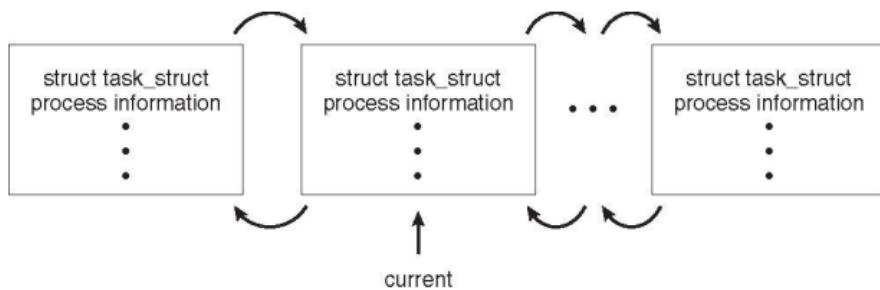
Il modello dei processi illustrato fin qui sottintende che un processo è un programma che si esegue seguendo un unico percorso d'esecuzione, detto thread. Se un processo sta, per esempio, eseguendo un programma di elaborazione di testi, l'esecuzione avviene seguendo una singola sequenza di istruzioni; quindi il processo può svolgere un solo compito alla volta. Tramite lo stesso processo, per esempio, un utente non può contemporaneamente inserire caratteri e verificare la correttezza ortografica di quel che sta scrivendo. In molti sistemi operativi moderni si è esteso il concetto di processo introducendo la possibilità d'avere più percorsi d'esecuzione, in modo da permettere che un processo possa svolgere più di un compito alla volta.

Rappresentazione dei processi di Linux

Il TCB nel sistema operativo Linux è rappresentato dalla struttura *task_struct* in linguaggio C, contenente una descrizione completa del processo, compreso il suo stato, informazioni sullo scheduling e sulla gestione della memoria, la lista dei file aperti e puntatori al processo padre e agli eventuali figli. Alcuni dei campi sono i seguenti:

```
pid_t pid; /* identificatore del processo */
long state; /* stato del processo */
unsigned int time_slice /* informazioni per lo scheduling */
struct files_struct *files; /* lista dei file aperti */
struct mm_struct *mm; /* spazio degli indirizzi del processo */
```

In Linux l'insieme dei processi è rappresentato da una lista doppiamente linkata di *task_struct* e il kernel mantiene un puntatore di nome *current* al processo attualmente in esecuzione:



Scheduling dei processi

L'obiettivo della multiprogrammazione consiste nel disporre dell'esecuzione contemporanea di alcuni processi in modo da massimizzare l'utilizzo della CPU. L'obiettivo della partizione del tempo 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. Nei sistemi monoprocesso non vi sarà mai più di un processo in esecuzione: gli altri dovranno attendere finché la CPU sia nuovamente disponibile.

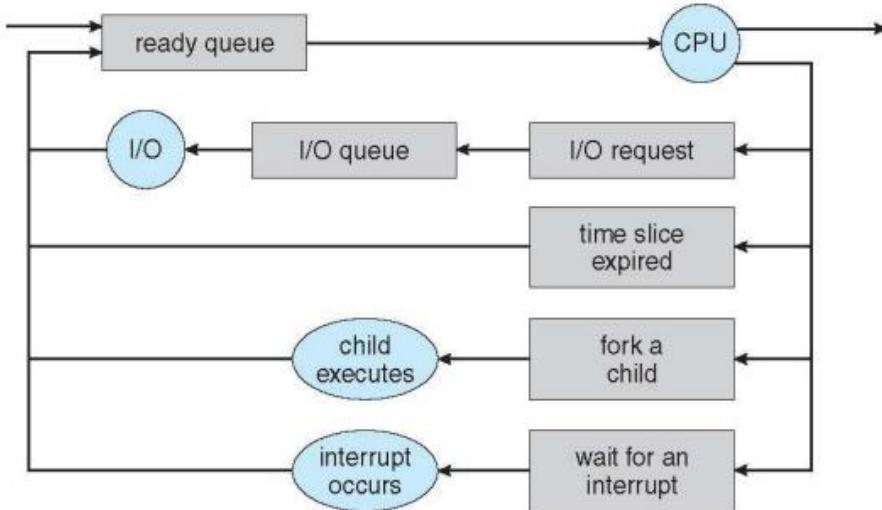
Code di scheduling

Ogni processo è inserito in una **coda di processi**, 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 **ready queue** (coda dei processi pronti). Questa coda generalmente si memorizza come una lista concatenata: un'intestazione della ready queue contiene i puntatori al primo e all'ultimo PCB dell'elenco, e ciascun PCB è esteso con 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. Una richiesta di I/O può essere diretta a un dispositivo condiviso, come un disco. 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 **device queue** (coda del dispositivo); ogni dispositivo ha la propria coda.

Una comune rappresentazione utile alla descrizione dello scheduling dei processi è dato da un **diagramma di accodamento** come quello illustrato nella figura seguente. Ogni riquadro rappresenta una coda. Sono presenti due tipi di coda: la ready queue e un insieme di device queue. I cerchi rappresentano risorse che servono le code, le frecce indicano il flusso di processi nel sistema.



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 (running), si può verificare uno di questi 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 e attenderne al termine;
- Il processo può essere rimosso forzatamente dalla CPU a causa di un'interruzione, ed essere reinserito nella coda dei processi pronti.

Nei primi due casi, al completamento della richiesta di I/O o al termine del processo figlio, il processo passa dallo stato di waiting allo stato ready ed è nuovamente inserito nella ready queue. Un processo continua questo ciclo fino al termine della sua esecuzione: a questo punto viene allontanato da tutte le code, rimosso il suo PCB e revocate le varie risorse.

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**. Spesso, in un sistema batch, accade che si sottopongano più processi di quanti se ne possano eseguire immediatamente. Questi lavori si trasferiscono in dispositivi di memoria secondaria, generalmente dischi, dove si tengono fino al momento dell'esecuzione (spooling). Il **job scheduler** (scheduler a lungo termine), sceglie i lavori da questo insieme e li carica in memoria affinché siano eseguiti. Il **CPU scheduler** (scheduler a breve termine) fa la selezione tra i job 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. Il CPU scheduler seleziona frequentemente un nuovo processo per la CPU. Il processo può essere in esecuzione solo per pochi millisecondi prima di passare ad attendere una richiesta di I/O. Poiché spesso si esegue almeno una volta ogni 100 millisecondi, lo scheduler a breve termine deve essere molto rapido per non sprecare tempo (se ad esempio impiegasse 10 millisecondi per decidere quale processo eseguire nei 100 successivi si userebbe il $\frac{10}{100+10} = 9\%$ del tempo di CPU per il solo scheduling).

Il job scheduler, 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 richiamata solo quando un processo abbandona il sistema. A

causa del maggior intervallo che intercorre tra le esecuzioni, il job scheduler 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. È fondamentale che lo scheduler a lungo termine selezioni una buona combinazione di processi I/O bound e CPU bound. Se tutti i processi fossero con prevalenza di I/O, la coda dei processi pronti sarebbe quasi sempre vuota e lo scheduler a breve termine resterebbe pressoché inattivo. Se tutti i processi fossero con prevalenza d'elaborazione, la coda d'attesa per l'I/O sarebbe quasi sempre vuota, i dispositivi non sarebbero utilizzati, e il sistema verrebbe nuovamente sbilanciato.

Esistono sistemi in cui lo scheduler a lungo termine può essere assente o minimo. Per esempio, i sistemi a partizione del tempo, come UNIX e Microsoft Windows, sono spesso privi di scheduler a lungo termine, e si limitano a caricare in memoria tutti i nuovi processi, gestiti dallo scheduler a breve termine. La stabilità di questi sistemi dipende dai limiti fisici degli stessi, come un numero limitato di terminali disponibili, oppure dall'autoregolamentazione insita nella natura degli utenti umani: quando ci si accorge che il rendimento della macchina scende a livelli inaccettabili si possono semplicemente chiudere alcune attività o la sessione di lavoro.

In alcuni sistemi operativi come quelli a partizione del tempo, si può introdurre un livello di scheduling intermedio, detto **scheduler a medio termine**. L'idea alla base di un tale scheduler è che a volte può essere vantaggioso eliminare processi dalla memoria (e dalla contesa attiva 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 **swapping**. Il processo viene rimosso e successivamente caricato in memoria dallo scheduler a medio termine. Lo swapping 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.

Context Switch

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 esegua 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 la registrazione dello stato del processo vecchio e il caricamento dello stato precedentemente registrato del nuovo processo. Questa procedura è nota col nome di **context switch** (cambio di contesto). Nell'evenienza del context switch, il sistema salva nel suo PCB il contesto del processo subentrante, salvato in precedenza. Il context switch è un **overhead**; infatti, comporta un calo delle prestazioni, perché il sistema esegue solo operazioni volte alla corretta gestione dei processi, e non alla computazione. Il tempo necessario varia da sistema a sistema, dipendendo dalla velocità della memoria, dal numero di registri da copiare, e dall'esistenza di istruzioni macchina appropriate (in generale si tratta di qualche millisecondo).

La durata del context switch dipende molto dall'architettura; per esempio alcune CPU offrono più gruppi di registri, quindi il context switch prevede la semplice modifica di un puntatore al gruppo di registri corrente.

Naturalmente, se il numero dei processi attivi è maggiore di quello dei gruppi di registri disponibili, il sistema rimedia copiando i dati dei registri nella e dalla memoria, come prima.

Operazioni sui processi

Nella maggior parte dei sistemi i processi si possono eseguire in modo concorrente, e si devono creare e cancellare dinamicamente; a tal fine il sistema operativo deve offrire un meccanismo che permetta di creare e terminare un processo.

Creazione di un processo

Durante la propria esecuzione, un processo può creare numerosi nuovi processi tramite un'apposita chiamata di sistema (`create_process`). Il processo creante si chiama processo **padre** (o anche genitore), 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 e la famiglia Windows, identifica un processo per mezzo di un numero univoco (solitamente un intero), detto **pid** (*process identifier*; ovvero, identificatore del processo). Nei sistemi UNIX si può ottenere l'elenco dei processi tramite il comando `ps`.

In generale, per eseguire il proprio compito, un processo necessita di alcune risorse (tempo d'elaborazione, memoria, file, dispositivi di I/O). Quando un processo crea un sottoprocesso, quest'ultimo può essere in grado di ottenere le proprie risorse direttamente dal sistema operativo, oppure può essere vincolato a un sottoinsieme delle risorse del processo padre. Il processo padre 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 padre, si può evitare che un processo sovraccarichi il sistema creando troppi sottoprocessi.

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

- 1) Il processo padre continua l'esecuzione in modo concorrente con i propri processi figli.
- 2) Il processo padre attende che alcuni o tutti i suoi processi figli terminino.

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

- 1) Il processo figlio è un duplicato del processo padre
- 2) Nel processo figlio si carica un nuovo programma.

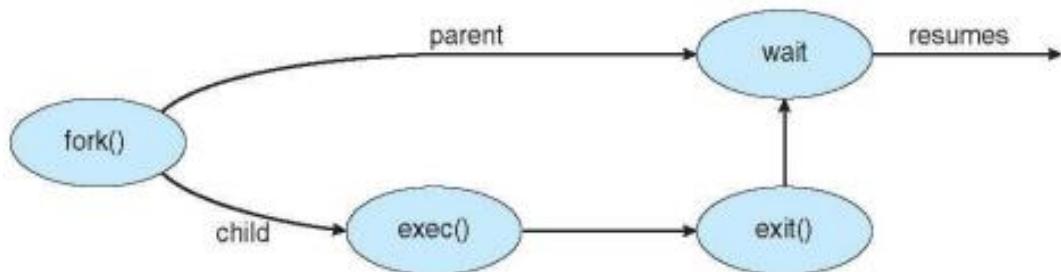
Nel sistema operativo UNIX, per esempio, ogni processo è identificato dal proprio pid. Un nuovo processo si crea per mezzo della chiamata di sistema `fork()`, ed è composto di una copia dello spazio degli indirizzi del processo padre. Questo meccanismo permette al processo padre di comunicare senza difficoltà con il proprio figlio. Entrambi i processi (padre e figlio) continuano l'esecuzione all'istruzione successiva alla chiamata di sistema `fork()`, con una differenza: la chiamata di sistema `fork()` restituisce il valore zero nel nuovo processo (il figlio), ma nel padre restituisce il PID (diverso da zero) del processo figlio. Tramite il valore riportato del PID, si può distinguere il processo padre dal 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. La chiamata di sistema `exec()` carica in memoria un file binario, cancellando l'immagine di memoria del programma contenente la stessa chiamata di sistema `exec()`, quindi avvia la sua esecuzione. In questo modo i due processi possono comunicare e quindi procedere in modo diverso. Il processo genitore può anche generare più processi figli, oppure, se durante l'esecuzione del processo figlio non ha nient'altro da fare, può invocare la chiamata di sistema `wait()` per rimuovere se stesso dalla coda dei processi pronti fino alla terminazione del figlio.

Esempio di programma scritto in C per sistema UNIX delle chiamate descritte precedentemente:

```
1  /* creazione di un processo separato
2   utilizzando la chiamata di sistema fork() */
3 #include <sys/types.h>
4 #include <stdio.h>
5 #include <unistd.h>
6
7 int main() {
8     pid_t pid;
9
10    // genera un nuovo processo
11    pid = fork();
12
13    if (pid < 0) { // errore
14        fprintf(stderr, "generazione del nuovo processo fallita");
15        return 1;
16    }
17    else if (pid == 0) { // processo figlio
18        execvp("/bin/ls", "ls", NULL);
19    }
20    else { // processo padre
21        // il genitore attende il completamento del figlio
22        wait(NULL);
23        printf("il processo figlio ha terminato");
24    }
25
26    return 0;
27 }
```

Si ottengono due processi distinti ciascuno dei quali è un'istanza d'esecuzione dello stesso programma. Il valore assegnato alla variabile pid è zero nel processo figlio, e un numero intero maggiore di zero nel processo padre. Usando la chiamata di sistema execvp() (una versione della chiamata exec() dove il processo figlio sovrappone il proprio spazio d'indirizzi con il comando /bin/ls di UNIX che si usa per ottenere l'elenco del contenuto di una directory). Impiegando la chiamata di sistema wait(), il processo padre attende che il processo figlio termini. Quando ciò accade, il processo genitore chiude la propria fase d'attesa dovuta alla chiamata di sistema wait() e termina usando la chiamata di sistema exit(). Questo passaggio è illustrato nella seguente figura.



Esempio di un tipico esercizio d'esame:

```
- int main(void) {
    int i;
    for (i = 0; i < 2; i++) {
        if (fork() > 0) {
            printf("padre %d\n", i);
        } else {
            printf("figlio! %d\n", i);
        }
    }
    sleep(10);
    return 0;
}
```

- 1) Qual è l'output di questo programma?
- 2) Quanti processi vengono creati?
- 3) Di chi è figlio ciascun processo creato?

Soluzione:

- 1) Un possibile output è il seguente (può cambiare in base allo scheduling):
padre 0 padre 1 figlio! 0 padre 1 figlio! 1 figlio! 1
- 2) Il numero di processi creati è 4. Il ramo del padre crea due figli e ogni figlio creerà un suo figlio.
- 3) Per rispondere basta rappresentare l'albero dei processi:

Proc principale

```
padre 0              figlio! 0  
padre1 figlio! 1      padre1 figlio! 1
```

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 alcuni dati al processo padre, che li riceve attraverso la chiamata di sistema `wait()`. Tutte le risorse del processo, incluse la memoria fisica e virtuale, i file aperti e le aree della memoria per l'I/O, sono liberate dal sistema operativo.

La terminazione di un processo si può verificare anche in altri casi. Un processo può causare la terminazione di un altro per mezzo di un'opportuna chiamata di sistema. Generalmente solo il padre del processo che si vuole terminare può invocare una chiamata di sistema di questo tipo, altrimenti gli utenti potrebbero causare arbitrariamente la terminazione forzata di processi di chiunque. Occorre notare che un padre deve conoscere le identità dei propri figli, perciò quando un processo ne crea uno nuovo, l'identità del nuovo processo viene passata al processo padre.

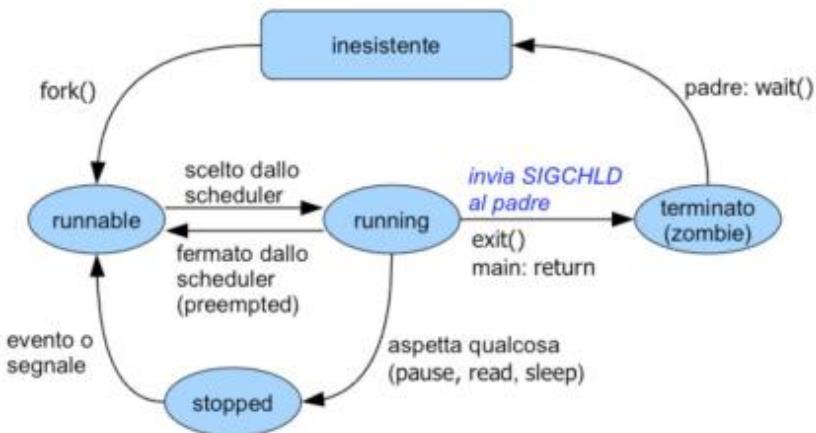
Un processo padre può porre termine all'esecuzione di uno dei suoi processi figli per diversi motivi, tra i quali i seguenti.

- 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 padre sia stata normale o anormale. Si parla di **terminazione a cascata** (*cascading termination*), una procedura avviata di solito dal sistema operativo.

Per illustrare un esempio di esecuzione e terminazione di un processo, si consideri che nel sistema operativo UNIX un processo può terminare per mezzo della chiamata di sistema `exit()`, e il suo processo padre può attendere l'evento per mezzo della chiamata di sistema `wait()`. Quest'ultima riporta l'identificatore di un processo figlio che ha terminato l'esecuzione, sicché il genitore può stabilire quale tra i suoi processi figli l'abbia terminata.

Se un processo genitore termina, tutti i suoi processi figli sono affidati al processo `init`, che assume il ruolo di nuovo padrone, cui i processi figli possono riportare i risultati delle proprie attività. Se non c'è padrone in `waiting` il figlio diventa **`zombie`** mentre se il padrone termina prima del figlio senza invocare `wait()` il figlio è **`orphan`**.



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.

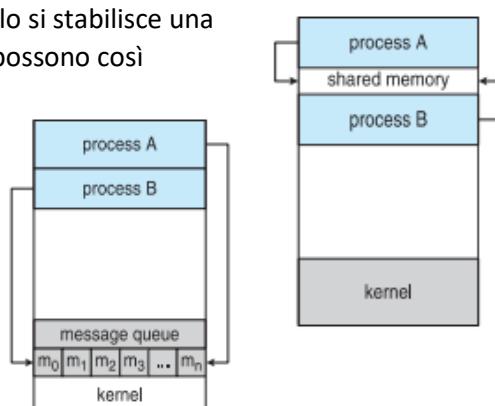
Un ambiente che consente la cooperazione tra processi può essere utile per diverse ragioni.

- **Condivisione d'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. Un'accelerazione di questo tipo è ottenibile solo se il calcolatore dispone di più elementi capaci di attività d'elaborazione (come CPU o canali di I/O).
- **Modularità**: Può essere necessaria la costruzione di un sistema modulare, che suddivide le funzioni di sistemi 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 stampa e compilazione.

Per lo scambio di dati e informazioni i processi cooperanti necessitano di un meccanismo di **comunicazione tra processi, IPC (interprocess communication)**. I modelli fondamentali della comunicazione tra processi sono due:

- **Shared memory** (a memoria condivisa): in questo modello si stabilisce una zona di memoria condivisa dai processi cooperanti, che possono così comunicare scrivendo e leggendo da tale zona.
- **Message passing** (a scambio di messaggi): in questo modello la comunicazione ha luogo tramite scambio di messaggi tra i processi cooperanti.

Nei sistemi operativi sono diffusi entrambi i modelli; a volte coesistono in un unico sistema. Lo scambio di messaggi è utile per trasmettere piccole quantità di dati, non essendovi bisogno di evitare conflitti. La memoria condivisa massimizza



l'efficienza della comunicazione, ed è più veloce dello scambio di messaggi, che è solitamente implementato tramite chiamate di sistema che impegnano il kernel; la memoria condivisa, invece, richiede l'intervento del kernel solo per allocare le regioni di memoria condivisa, dopo di che tutti gli accessi sono gestiti alla stregua di ordinari accessi in memoria che non richiedono l'assistenza del kernel.

Sistemi shared memory

La comunicazione tra processi basata sulla condivisione della memoria richiede che i processi comunicanti allochino una zona di memoria condivisa, di solito residente nello spazio degli indirizzi del processo che la alloca: gli altri processi che desiderano usarla per comunicare dovranno annetterla al loro spazio degli indirizzi. Il tipo e la collocazione dei dati sono determinati dai processi, e rimangono al di fuori del controllo del sistema operativo. I processi hanno anche la responsabilità di non scrivere nella stessa locazione simultaneamente.

Per illustrare il concetto di cooperazione tra processi, si consideri il problema del produttore/consumatore; tale problema è un usuale 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, e il consumatore consumerne un'altra. I due processi devono essere sincronizzati in modo tale che il consumatore non tenti di consumare un'unità non ancora prodotta. Sono utilizzati due tipi di buffer:

- **Unbounded-buffer** (buffer illimitato): non pone limiti alla dimensione del buffer. Il consumatore può trovarsi ad attendere nuovi oggetti, ma il produttore può sempre produrre.
- **Bounded-buffer** (buffer limitato): presuppone l'esistenza di una dimensione fissa del buffer. In questo caso, il consumatore deve attendere se il buffer; viceversa, il produttore deve attendere se il buffer è pieno.

Consideriamo più attentamente come il buffer limitato sia utilizzabile per la condivisione di memoria tra processi. Le variabili seguenti risiedono in una zona di memoria condivisa sia dal produttore sia dal consumatore.

```
#define BUFFER_SIZE 10
typedef struct {
    ...
} item;

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

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

Il buffer è pieno se $((in + 1) \% \text{BUFFER_SIZE}) == \text{out}$

Processo produttore:

```
item next_produced: // elemento appena prodotto

while (true) {
    // produce un elemento in next_produced
    while (((in + 1) \% BUFFER_SIZE) == out)
        ; // non fare nulla
    buffer[in] = next_produced;
    in = (in + 1) \% BUFFER_SIZE;
}
```

Processo consumatore:

```
item next_consumed; // elemento da consumare

while (true) {
    while (in == out)
        ; // non fare nulla
    next_consumed = buffer[out]
    out = (out + 1) % BUFFER_SIZE;

    // consuma l'elemento in next_consumed
}
```

Questo metodo ammette un massimo di BUFFER_SIZE – 1 oggetti contemporaneamente presenti nel buffer. Di seguito proponiamo una versione che permetta la presenza contemporanea di BUFFER_SIZE oggetti.

Produttore:

```
item next_produced;
while (true) {
    // produco l'oggetto
    while (isBufferFull);

    in = (in + 1) % BUFFER_SIZE;
    buffer[in] = next_produced;
    size++;
}
```

Consumatore:

```
item next_consumed;
while (true) {
    while (isBufferEmpty);

    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    size--;
    // consumo l'oggetto
}
```

Una questione ignorata dalla precedente analisi è il caso in cui sia il produttore sia il consumatore tentano di accedere al buffer concorrentemente.

Sistemi message passing

Lo scambio di messaggi è un meccanismo che permette a due o più processi di comunicare e di sincronizzarsi senza condividere lo stesso spazio degli 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 (cioè, “invia messaggio”) e receive (cioè, “ricevi messaggio”). 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 è più complessa, mentre la programmazione utente risulta semplificata. Compromessi di questo tipo si riscontrano spesso nella progettazione dei sistemi operativi.

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. In questo paragrafo non si tratta della realizzazione fisica del canale (come la memoria condivisa, i bus o le reti) ma della sua realizzazione logica. 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.

Le questioni legate a ciascuna di tali caratteristiche vengono illustrate in seguito.

Nominazione. 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:

- $\text{send}(P, \text{messaggio})$, invia *messaggio* al processo *P*;
- $\text{receive}(Q, \text{messaggio})$, riceve, in *messaggio*, un 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. Per poter comunicare, il trasmittente e il ricevente devono nominarsi a vicenda (esiste anche una versione in cui solo il trasmittente nomina il ricevente). Questo schema ha lo svantaggio di una limitata modularità; infatti, se cambia il nome di un processo bisogna fare un riesame di tutti i processi e cambiare i riferimenti con il vecchio nome.

Con la **comunicazione indiretta**, i messaggi si inviano a delle **porte** (dette anche *mailbox*), che li ricevono.

Una porta 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 porte. Due processi possono comunicare solo se condividono una porta. Le primitive si definiscono come segue:

- $\text{send}(A, \text{messaggio})$, invia *messaggio* alla porta *A*;
- $\text{receive}(A, \text{messaggio})$, riceve, in *messaggio*, un messaggio dalla porta *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 porta;
- 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 porta.

A questo punto, si supponga che i processi P_1, P_2 e P_3 condividano la porta *A*. Il processo P_1 invia un messaggio ad *A*, mentre sia P_2 sia P_3 eseguono una receive da *A*. Sorge il problema di sapere quale processo riceverà il messaggio. La soluzione dipende dallo schema prescelto:

- si può fare in modo che un canale sia associato al massimo a due processi;
- si può consentire l'esecuzione di un'operazione receive a un solo processo alla volta;
- si può consentire al sistema di decidere arbitrariamente quale processo riceverà il messaggio (il messaggio sarà ricevuto da P_2 o da P_3 , ma non da entrambi). Il sistema può anche definire un algoritmo per selezionare quale processo riceverà il messaggio (specificando cioè uno schema, detto *round robin*, secondo il quale i processi ricevono i messaggi a turno) e può comunicare l'identità del ricevente al trasmittente.

Sincronizzazione. La comunicazione tra processi avviene attraverso chiamate delle primitive send() e receive(). Ci sono diverse possibilità nella definizione di ciascuna primitiva. 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 processo ricevente, o la porta, 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 oppure un valore nullo.

È possibile anche avere diverse combinazioni di send() e receive() tra quelle illustrate sopra. Se le primitive send() e receive() sono entrambe bloccanti si parla di **rendezvous** tra mittente e ricevente. È

banale dare soluzione al problema del produttore e del consumatore tramite le primitive bloccanti `send()` e `receive()`. Il produttore, infatti, si limita a invocare `send()` e attendere finché il messaggio sia giunto a destinazione; analogamente, il consumatore richama `receive()`, bloccandosi fino all'arrivo di un messaggio.

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 messaggio viene copiato oppure si tiene un puntatore a quel messaggio. 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.

Il caso con capacità zero è talvolta chiamato sistema a scambio di messaggi senza memorizzazione transitoria (*no buffering*); gli altri due, sistemi con memorizzazione transitoria automatica (*automatic buffering*).

Comunicazione nei sistemi client-server

Abbiamo già visto come i processi possano comunicare usando memoria condivisa e scambio di messaggi. Tali tecniche sono utilizzabili anche per la comunicazione client-server. In questo paragrafo considereremo altre tre strategie: Socket, chiamate di procedura remota (RPC) e invocazione di metodi remoti (RMI).

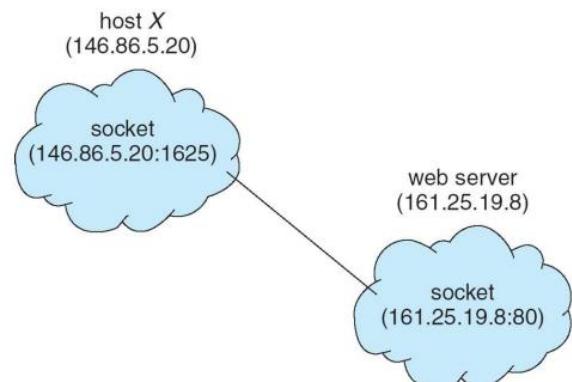
Socket

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. 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, se accetta la connessione proveniente dalla socket del client, si stabilisce la comunicazione. I server che svolgono servizi specifici stanno in ascolto a porte note (i server telnet, ad esempio, usano la porta 23, i server ftp la porta 21 mentre i server web la porta 80). Tutte le porte al di sotto del valore 1024 sono considerate note e si usano per realizzare servizi standard.

Quando un processo client richiede una connessione, il calcolatore che lo esegue assegna una porta specifica, che consiste di un numero arbitrario maggiore di 1024. Si supponga per esempio che un processo client presente nel calcolatore X con indirizzo IP 146.86.5.20 voglia stabilire una connessione con un server Web (in ascolto alla porta 80) all'indirizzo 161.25.19.8; il calcolatore X potrebbe assegnare al client, per esempio, la porta 1625. La connessione sarebbe composta di una coppia di socket: (146.86.5.20:1625) nel calcolatore X e (161.25.19.8:80) nel server Web.

La figura a destra mostra questa situazione.

La consegna dei pacchetti al processo giusto avviene secondo il numero della porta di destinazione.



Tutte le connessioni devono essere uniche; quindi, se un altro processo, nel calcolatore *X*, vuole stabilire un'altra connessione con lo stesso server Web, riceve un numero di porta maggiore di 1024 e diverso da 1625. Ciò assicura che ciascuna connessione sia identificata da una distinta coppia di socket.

Un indirizzo IP speciale 127.0.0.1 (*loopback*) si riferisce al sistema in cui il processo gira.

Forniamo un esempio di codice in C che implementi una comunicazione client-server Unix tramite socket:

```
/* SERVER IN ASCOLTO */
int fd1, fd2;
struct sockaddr_in indirizzo;

indirizzo.sin_family = AF_INET;
indirizzo.sin_port = htons(5200);
indirizzo.sin_addr.s_addr = htonl(INADDR_ANY);

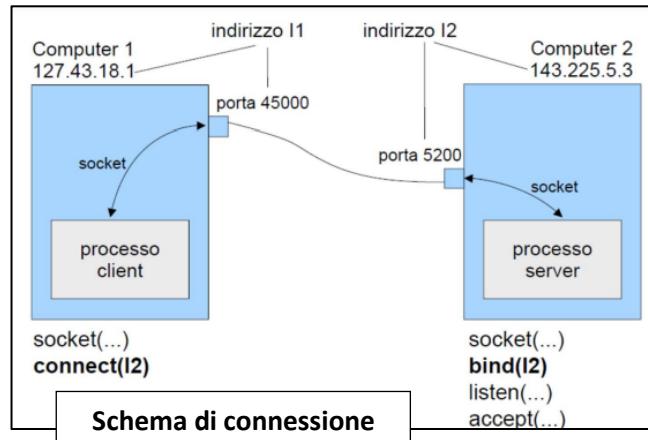
fd1 = socket(PF_INET, SOCK_STREAM, 0);
bind(fd1, (struct sockaddr *) &indirizzo, sizeof(indirizzo));

listen(fd1, 5);
fd2 = accept(fd1, NULL, NULL)
...
close(fd2);
close(fd1);

/* CLIEC CONNESSO */
int fd;
struct sockaddr_in indirizzo;

indirizzo.sin_family = AF_INET;
indirizzo.sin_port = htons(5200);
inet_aton("143.225.5.3", &indirizzo.sin_addr);

fd = socket(PF_INET, SOCK_STREAM, 0);
connect(fd, (struct sockaddr *) &indirizzo, sizeof(indirizzo));
...
close(fd);
```



Nota: Il linguaggio Java offre un'interfaccia alle socket più semplice e dispone di una ricca libreria di strumenti di rete. Infatti, Java prevede tre tipi differenti di socket: quelle orientate alla connessione (TCP) sono realizzate con la classe *Socket*; quelle prive di connessione (UDP) usano la classe *DatagramSocket*; il terzo tipo di socket è basato sulla classe *MulticastSocket*; si tratta di una sottoclasse della classe *DatagramSocket* che permette l'invio simultaneo dei dati a diversi destinatari (*multicast*).

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. Poiché in un sistema distribuito si eseguono i processi su sistemi distinti, per offrire un servizio remoto occorre impiegare uno schema di comunicazione basato sullo scambio di messaggi. Contrariamente a quel che accade nella funzione IPC, i messaggi scambiati per la comunicazione RPC sono ben strutturati e non semplici pacchetti di dati. Si indirizzano a un demone di RPC, in ascolto a una porta del sistema remoto, e contengono un identificatore della funzione da eseguire e i parametri da inviare 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 dei pacchetti di messaggi. 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 di rete che può fornire. Se un processo remoto richiede un servizio, indirizza i propri messaggi alla porta corrispondente; per esempio, per permettere ad altri di ottenere un elenco dei suoi attuali utenti, un sistema deve possedere un demone in ascolto a una porta, per esempio la porta 3027, che realizzi una siffatta RPC. Qualsiasi sistema remoto può ottenere l'informazione richiesta, vale a dire l'elenco degli utenti, inviando un messaggio RPC alla porta 3027 del server; i dati si ricevono in un messaggio di risposta.

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, assegnando un **segmento di codice di riferimento (stub)** alla parte client. Esiste in genere un segmento di codice di riferimento per ogni diversa procedura remota. Quando il client lo invoca, il sistema delle RPC richiama l'appropriato segmento di codice di riferimento, passando i parametri della procedura remota. Il segmento di codice di riferimento 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. Il segmento di codice di riferimento quindi trasmette un messaggio al server usando lo scambio di messaggi. Un analogo segmento di codice di riferimento nel server riceve questo messaggio e invoca la procedura nel server; se è necessario, riporta i risultati al client usando la stessa tecnica.

Una questione da affrontare riguarda le differenze nella rappresentazione dei dati nel client e nel server. Per risolvere questo problema, molti sistemi di RPC definiscono una rappresentazione dei dati indipendente dalla macchina. Uno di questi sistemi di rappresentazione è noto come **rappresentazione esterna dei dati (external data representation, XDR)**. Nel client la strutturazione dei parametri riguarda la conversione dei dati, prima di inviarli al server, dal formato della specifica macchina nel formato XDR; nel server, i dati nel formato XDR si convertono nel formato della macchina server.

Pipe

Una pipe agisce come canale di comunicazione tra processi. Le pipe sono state uno dei primi meccanismi di comunicazione tra processi (IPC) nei sistemi UNIX e generalmente forniscono ai processi uno dei metodi più semplici per comunicare l'uno con l'altro, sebbene con qualche limitazione. Quando si implementa una pipe devono essere prese in considerazione quattro questioni.

- 1) La comunicazione permessa della pipe è unidirezionale o bidirezionale?
- 2) Se è ammessa la comunicazione a doppio senso, essa è di tipo *half duplex* (i dati possono viaggiare in un'unica direzione alla volta) o *full duplex* (i dati possono viaggiare contemporaneamente in entrambe le direzioni)?
- 3) Deve esistere una relazione (del tipo padre-figlio) tra i processi in comunicazione?
- 4) Le pipe possono comunicare in rete o i processi comunicanti devono risiedere sulla stessa macchina?

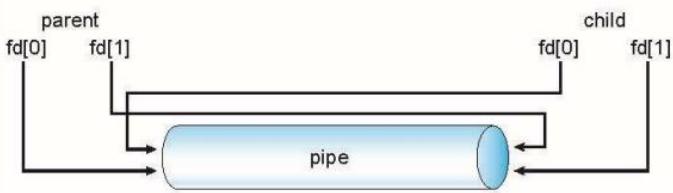
Le **pipe ordinarie** permettono a due processi di comunicare secondo la modalità standard produttore-consumatore. Il produttore scrive a una estremità del canale (l'estremità dedicata alla scrittura, o **write-end**) mentre il consumatore legge dall'altra estremità (l'estremità dedicata alla lettura, o **read-end**). Le pipe ordinarie sono quindi unidirezionali, perché permettono la comunicazione in un'unica direzione. Se viene richiesta la comunicazione a doppio senso devono essere utilizzate due pipe, ognuna delle quali manda i dati in una differente direzione.

Nel sistema UNIX le pipe ordinarie sono costruite utilizzando la funzione `pipe(int fd[])`. Essa crea una pipe alla quale si può accedere tramite i descrittori del file (*file descriptor*) `fd[]`; dove `fd[0]` è l'estremità dedicata alla lettura, mentre `fd[1]` è l'estremità dedicata alla scrittura. Il sistema UNIX considera una pipe

come un tipo speciale di file; si può così accedere alle pipe tramite le usuali chiamate di sistema `read()` e `write()`.

Non si può accedere a una pipe al di fuori del processo che la crea. Solitamente un processo padre crea una pipe e la utilizza per comunicare con un processo figlio generato con il comando `fork()`. Il processo figlio eredita i file aperti dal processo padre.

Dal momento che la pipe è un tipo speciale di file, il figlio eredita la pipe dal proprio processo padre. La figura illustra la relazione del descrittore di file `fd` rispetto ai processi padre e figlio.



Nel seguente programma il processo padre crea una pipe e in seguito esegue una chiamata `fork()`, generando un processo figlio. Ciò che succede dopo la chiamata `fork()` dipende da come i dati fluiscano nel canale. In questo caso, il padre scrive sulla pipe e il figlio legge da essa. È importante sottolineare come sia il processo padre sia il processo figlio chiudano inizialmente le estremità inutilizzate del canale.

```
1 #include <sys/types.h>
2 #include <stdio.h>
3 #include <string.h>
4 #include <unistd.h>
5
6 #define BUFFER_SIZE 25
7
8 int main(void) {
9     char w_msg[BUFFER_SIZE] = "Greetings";
10    char r_msg[BUFFER_SIZE];
11    int fd[2];
12    pid_t pid;
13
14    // crea la pipe
15    if (pipe(fd) == -1) {
16        fprintf(stderr, "Pipe failed");
17        return 1;
18    }
19
20    // crea tramite fork un processo figlio
21    pid = fork();
22    if (pid < 0) { // errore
23        fprintf(stderr, "Fork failed");
24        return 1;
25    }
26    if (pid > 0) { // processo padre
27        // chiude l'estremità inutilizzata della pipe
28        close(fd[0]);
29
30        // scrive sulla pipe
31        write(fd[1], w_msg, strlen(w_msg) + 1);
32        close(fd[1]);
33    }
}
```

```

34     -    else { // processo figlio
35         // chiude l'estremità inutilizzata della pipe
36         close(fd[1]);
37
38         // legge dalla pipe
39         read(fd[0], r_msg, BUFFER_SIZE);
40         printf("read %s", r_msg);
41         close(fd[0]);
42     }
43
44     return 0;
45 }
```

Si noti bene che le pipe convenzionali richiedono una relazione di parentela padre-figlio tra i processi comunicanti, sia in UNIX sia in Windows. Ciò significa che queste pipe possono essere utilizzate soltanto per la comunicazione tra processi in esecuzione sulla stessa macchina.

Named pipe. Le named pipe costituiscono uno strumento di comunicazione molto più potente; la comunicazione può essere bidirezionale, e la relazione di parentela padre-figlio non è necessaria. Una volta che si sia creata la named pipe, diversi processi possono utilizzarla per comunicare. In uno scenario tipico una named pipe ha infatti diversi scrittori. In più, le named pipe continuano a esistere anche dopo che i processi comunicanti sono terminati. Sia UNIX sia Windows mettono a disposizione le named pipe, nonostante ci siano grandi differenze nei dettagli dell'implementazione. Nei sistemi UNIX le named pipe sono dette FIFO. Una volta create, esse appaiono come normali file all'interno del file system. Una FIFO viene creata mediante una chiamata di sistema mkfifo() e viene poi manipolata con le usuali chiamate di sistema open(), read(), write() e close(); essa continuerà a esistere finché non sarà esplicitamente eliminata dal file system. Nonostante i file FIFO permettano la comunicazione bidirezionale, l'unica tipologia di trasmissione consentita è quella half duplex. Nel caso in cui i dati debbano viaggiare in entrambe le direzioni, vengono solitamente utilizzate due FIFO. Per utilizzare le FIFO i processi comunicanti devono risiedere sulla stessa macchina: se è richiesta la comunicazione tra più macchine devono essere impiegate le socket.

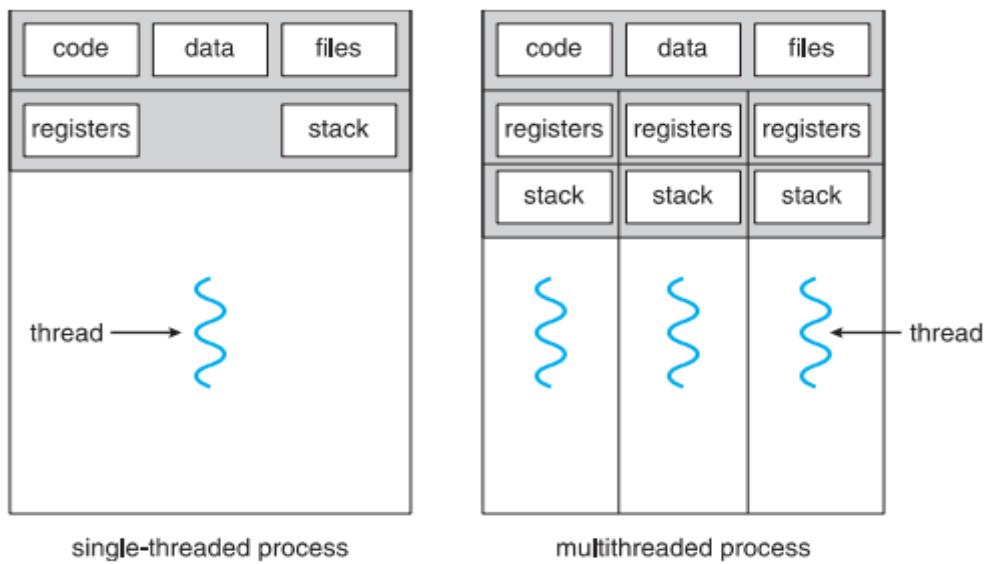
Pipe nella pratica. Le pipe sono usate abbastanza spesso dalla riga di comando di UNIX in situazioni nelle quali l'output di un comando serve da input per un secondo comando. Ad esempio, il comando ls di UNIX elenca il contenuto di una directory. Per directory particolarmente grandi, l'output può scorrere su diverse schermate. Il comando more gestisce l'output mostrando solo una schermata alla volta; l'utente deve premere la barra spaziatrice per muoversi da una schermata all'altra. Istituendo una pipe tra i comandi ls e more (in esecuzione come singoli processi) si fa in modo che l'output di ls venga inviato all'input di more, permettendo così all'utente di vedere il contenuto di una grande directory una schermata alla volta. Dalla riga di comando si può costruire una pipe utilizzando il carattere | . Il comando completo è quindi ls | more. In questo scenario, il comando ls funge da produttore, e il suo output è consumato dal comando more.

4. Thread

Introduzione

Un thread è l'unità di base d'uso della CPU e comprende un identificatore di thread (ID), un program counter, un insieme di registri, ed uno stack. Condivide con gli altri thread che appartengono allo stesso processo la sezione del codice, la sezione dei dati e altre risorse di sistema, come i file aperti e i segnali.

Un processo tradizionale, chiamato anche **heavyweight process** (processo pesante), è composto da un solo thread. Un processo multithread è in grado di lavorare a più compiti in modo concorrente. La figura successiva mostra la differenza tra un processo tradizionale, a singolo thread, e uno multithread (si noti che ogni thread nell'ambiente multithread ha un proprio insieme di registri e di stack che non condivide).

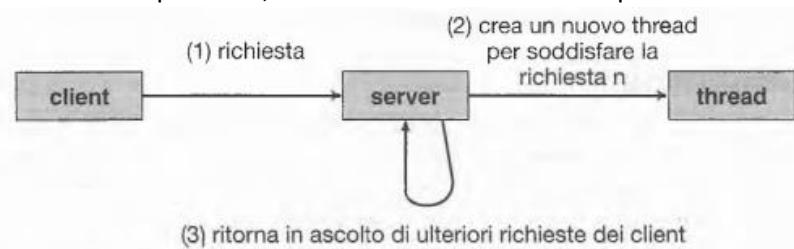


Motivazioni

Molti programmi per i moderni PC sono predisposti per essere eseguiti da processi multithread. Dì solito, un'applicazione si codifica come un processo a sé stante comprendente più thread di controllo. In alcune situazioni, una singola applicazione deve poter gestire molti compiti simili tra loro.

Per esempio, se un server web fosse single thread sarebbe in grado di soddisfare un solo client alla volta. Una soluzione è eseguire il server come un singolo processo che accetta richieste e quando ne riceve una, il server crea un processo separato per eseguirla ma tale soluzione è molto onerosa sia a livello dei tempi che dei costi.

Generalmente, per raggiungere lo stesso obiettivo è più conveniente impiegare un processo multithread. Nel caso del server Web, l'adozione di questo metodo porterebbe alla scelta di un processo multithread: il server genererebbe un thread distinto per ricevere eventuali richieste dei client; alla presenza di una richiesta, anziché creare un altro processo, si creerebbe un altro thread per soddisfarla.



La gestione di server multiprocesso è molto più leggera se implementata tramite thread (il server risulterà meno pesante).

I thread hanno un ruolo primario nei sistemi che impiegano le RPC (*remote procedure call*); si tratta di un sistema che permette la comunicazione tra processi, fornendo un meccanismo di comunicazione simile alle normali chiamate di funzione o procedura. Di solito, i server RPC sono multithread; quando riceve un messaggio, il server delega la gestione a un thread separato, in questo modo può gestire diverse richieste in modo concorrente. Infine, molti kernel di SO sono ormai multithread, con i singoli thread dedicati a specifici servizi; per esempio, la gestione dei dispositivi periferici o delle interruzioni. Linux usa un thread a livello kernel per la gestione della memoria libera del sistema.

Vantaggi

I vantaggi della programmazione multithread si possono classificare rispetto a quattro fattori principali.

- 1) **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 medio all'utente.
- 2) **Condivisione delle risorse.** I processi possono condividere risorse soltanto attraverso tecniche come la memoria condivisa o il passaggio di messaggi. Queste tecniche devono essere esplicitamente messe in atto e organizzate dal programmatore. Tuttavia, i thread condividono d'ufficio la memoria e le risorse del processo al quale appartengono. Il vantaggio della condivisione del codice consiste nel fatto che un'applicazione può avere molti thread di attività diverse, tutti nello stesso spazio d'indirizzi.
- 3) **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.
- 4) **Scalabilità.** I vantaggi della programmazione multithread aumentano notevolmente nelle architetture multiprocessore, dove i thread si possono eseguire in parallelo (uno per ciascun processore). 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.

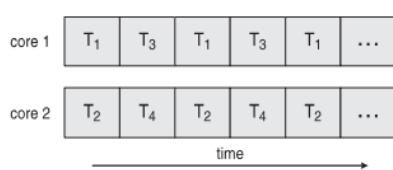
Programmazione multicore

Una tendenza recente nel progetto dell'architettura dei sistemi consiste nel montare diverse unità di calcolo (**core**) su un unico processore (un processore multicore); ogni unità appare al sistema operativo come un processore separato. La programmazione multithread offre un meccanismo per un utilizzo più efficiente di questi processori e aiuta a sfruttare al meglio la concorrenza. Si consideri un'applicazione con quattro thread. In un sistema con una singola unità di calcolo "esecuzione concorrente" significa solo che l'esecuzione dei thread è stratificata nel tempo, o come anche si dice, **interfogliata (interleaved)**, perché la CPU è in grado di eseguire un solo thread alla volta. 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 ciascuna unità di calcolo.

Esecuzione concorrente su sistema single-core:



Parallelismo su sistema multi-core:



N.B.: Differenza tra parallelismo e concorrenza è una tipica domanda d'esame.

- **Parallelismo:** calcolo che avviene contemporaneamente
- **Concorrenza:** diversi processi si alternano su una stessa cpu

La tendenza verso i sistemi multicore ha messo sotto pressione i progettisti di sistemi operativi e i programmati di applicazioni, affinché entrambi utilizzino al meglio unità di calcolo multiple. I progettisti di sistemi operativi devono scrivere algoritmi di scheduling che utilizzano diverse unità di calcolo per permettere un'esecuzione parallela come quella mostrata nella figura precedente. Per i programmati di applicazioni, la sfida consiste nel modificare programmi esistenti e progettare nuovi programmi multithread per trarre vantaggio dai sistemi multicore. In generale, possiamo individuare nelle cinque aree seguenti i principali obiettivi della programmazione dei sistemi multicore.

- 1) **Separazione dei task.** Consiste nell'esaminare le applicazioni al fine di individuare aree separabili in task distinti e concorrenti che possano essere eseguiti in parallelo su unità di calcolo distinte.
- 2) **Bilanciamento.** Nell'identificare i task eseguibili in parallelo, i programmati devono far sì che i vari task eseguano compiti di mole e valore confrontabili. In alcuni casi si verifica che un determinato task non contribuisca al processo complessivo tanto quanto gli altri; in questi casi per eseguire il task può non valere la pena utilizzare un'unità di calcolo separata.
- 3) **Suddivisione dei dati.** Proprio come le applicazioni sono divise in task separati, i dati a cui i task accedono, e che manipolano, devono essere suddivisi per essere utilizzati da unità di calcolo distinte.
- 4) **Dipendenze dei dati.** I dati a cui i task accedono devono essere esaminati per verificare le dipendenze tra due o più task. In esempi in cui un task dipende dai dati forniti da un altro, i programmati devono assicurare che l'esecuzione dei task sia sincronizzata in modo da soddisfare queste dipendenze.
- 5) **Test e debugging.** Quando un programma funziona in parallelo su unità multiple, vi sono diversi possibili flussi di esecuzione. Effettuare i test e il debugging di programmi concorrenti è per natura più difficile rispetto al caso di applicazioni con un singolo thread.

Legge di Amdahl. Guadagno teorico in performance con l'aggiunta di un core per un'applicazione che ha sia componenti seriali che paralleli. La formula è definita come segue (S è la porzione seriale mentre N indica il numero di core usati per il processamento):

$$\text{speedup} \leq \frac{1}{S + \frac{1-S}{N}}$$

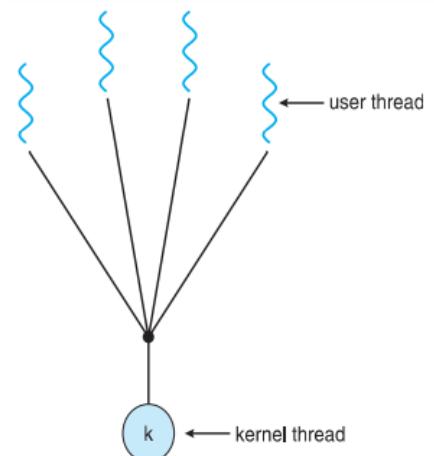
Ad esempio, se l'applicazione è al 75% parallela e 25% seriale, passando da 1 a 2 core si ha uno speedup di 1.6 volte (con $N \rightarrow \infty$ lo speedup tende a $\frac{1}{S}$). La porzione seriale di un'applicazione ha effetto molto marcato sul guadagno di performance con un core aggiuntivo.

Modelli di programmazione multithread

I thread possono essere distinti in **user thread** (thread a livello utente) e **kernel thread** (thread a livello kernel): i primi sono gestiti senza l'aiuto del kernel; i secondi, invece, sono gestiti direttamente dal sistema operativo. In ultima analisi, deve esistere una relazione tra gli user thread e i kernel thread.

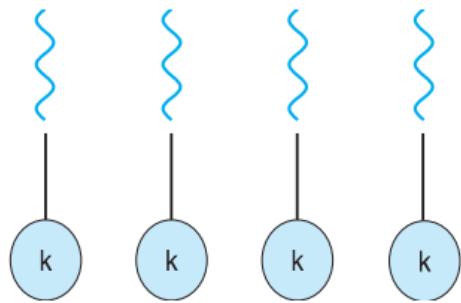
Modello Many-to-One

Il modello Many-to-One fa corrispondere molti thread a livello utente a un singolo thread a livello kernel. Poiché si svolge nello spazio utente, la gestione dei thread risulta efficiente, ma 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 multiprocessore; pochi sistemi usano questo approccio (Solaris ne è un esempio).



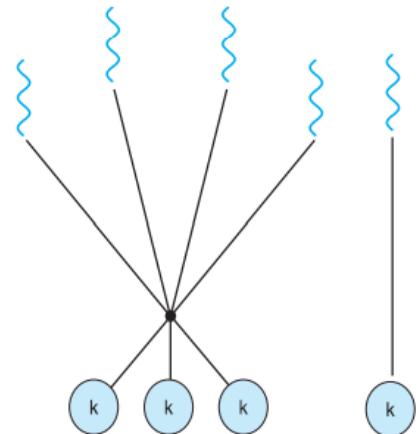
Modello One-to-One

Il modello One-to-One 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 compromettere le prestazioni di un'applicazione, la maggior parte delle realizzazioni di questo modello limita il numero di thread gestibili dal sistema. I sistemi operativi Linux, insieme alla famiglia dei sistemi operativi Windows, adottano il modello da uno a uno.



Modello Many-to-Many

Il modello Many-to-Many 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 (un'applicazione potrebbe assegnare un numero maggiore di thread a livello kernel a un'architettura multiprocessore rispetto quanti ne assegnerrebbe a una con singola CPU). 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.



Modello Two-Level

Una diffusa variante del modello da molti a molti mantiene la corrispondenza fra più thread utente con un numero minore o uguale di thread del kernel, ma permette anche di vincolare un thread utente a un solo thread del kernel.

Librerie dei thread

La libreria dei thread fornisce al programmatore una API per la creazione e la gestione dei thread. I metodi con cui implementare una libreria dei thread sono essenzialmente due. Nel primo, la libreria è collocata interamente a livello utente, senza fare ricorso al kernel. Il codice e le strutture dati per la libreria risiedono tutti nello spazio degli utenti. Questo implica che invocare una funzione della libreria si traduce in una chiamata locale a una funzione nello spazio degli utenti e non in una chiamata di sistema. Il secondo metodo consiste nell'implementare una libreria a livello kernel, con l'ausilio diretto del sistema operativo. In questo caso, il codice e le strutture dati per la libreria si trovano nello spazio del kernel. Invocare una funzione della API per la libreria provoca, generalmente, una chiamata di sistema al kernel.

Attualmente, sono tre le librerie di thread maggiormente in uso: Pthreads di POSIX, Win32 e Java. Pthreads, estensione dello standard POSIX, può presentarsi sia come libreria a livello utente sia a livello kernel. La libreria di thread Win32 è una libreria a livello kernel per i sistemi Windows. La API per la creazione dei thread in Java è gestibile direttamente dai programmi Java. Tuttavia, data la peculiarità di funzionamento della JVM, quasi sempre eseguita all'interno di un sistema operativo che la ospita, la API di Java per i thread è solitamente implementata per mezzo di una libreria dei thread del sistema ospitante. Perciò, i thread di Java sui sistemi Windows sono in effetti implementati mediante la API Win32; sui sistemi UNIX e Linux, invece, si adopera spesso Pthreads.

Pthreads

Col termine Pthreads ci si riferisce allo standard POSIX che definisce la API per la creazione e la sincronizzazione dei thread. Non si tratta di una **realizzazione**, ma di una **definizione** del comportamento dei thread; i progettisti di sistemi operativi possono realizzare le API così definite come meglio credono. Sono molti i sistemi che implementano le specifiche Pthreads; fra questi, Solaris, Linux, Mac OS X, e Tru64 UNIX. Per i vari sistemi Windows, sono disponibili implementazioni **shareware** di dominio pubblico.

Come esempio dimostrativo definiamo un programma in C che computa in un ambiente multithread la somma dei primi N interi non negativi in un thread separato, in simboli: $sum = \sum_{i=0}^N i$.

```
1 #include <pthread.h>
2 #include <stdio.h>
3
4 int sum; // questo dato è condiviso dai thread
5 void *runner(void *param); // il thread
6
7 -int main(int argc, char *argv[]) {
8     pthread_t tid; //id del thread
9     pthread_attr_t attr; // insieme di attributi del thread
10
11 -    if (argc != 2) {
12         fprintf(stderr, "usage: a.out <integer value>\n");
13         return -1;
14     }
15 -    if (atoi(argv[1]) < 0) {
16         fprintf(stderr, "%d must be >= 0\n", atoi(argv[1]));
17         return -1;
18     }
19     // reperisce gli attributi predefiniti
20     pthread_attr_init(&attr);
21     // crea il thread
22     pthread_create(&tid, &attr, runner, argv[1]);
23     // attende la terminazione del thread
24     pthread_join(tid, NULL);
25
26     printf("sum = %d\n", sum);
27 }
28
29 // il thread assume il controllo da questa funzione
30 -void *runner(void *param) {
31     int upper = atoi(param);
32     sum = 0;
33
34     int i;
35     for (i = 1; i <= upper; i++)
36         sum += i;
37
38     pthread_exit(0);
39 }
```

Nei programmi Pthreads, i nuovi thread sono eseguiti 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 i valori globali di *sum*.

Tutti i programmi che impiegano la libreria Pthreads devono includere il file d'intestazione pthread.h. La dichiarazione di variabili pthread_t *tid* specifica l'identificatore per il thread da creare. Ogni thread ha un insieme di attributi che includono la dimensione della pila e informazioni di scheduling. La dichiarazione pthread_attr_t *attr* riguarda la struttura dati per gli attributi del thread, i cui valori si assegnano con la chiamata di funzione pthread_attr_init(&*attr*). Poiché non sono stati esplicitamente forniti valori per gli attributi, si usano quelli predefiniti. La chiamata di funzione pthread_create crea un nuovo thread. Oltre all'identificatore del thread e ai suoi attributi, si passa anche il nome della funzione da cui il nuovo thread inizierà l'esecuzione, in questo caso la funzione runner(), e il numero intero fornito come parametro alla riga di comando e individuato da argv[1].

A questo punto il programma ha due thread: il thread iniziale (o genitore), in main(); e il thread che esegue la somma (o figlio), in runner(). Dopo aver creato il secondo, il primo thread attende il completamento del secondo chiamando la funzione pthread_join(). Il secondo thread termina quando s'invoca la funzione pthread_exit(). Quando il thread che esegue la somma termina, il thread iniziale produce in uscita il valore condiviso *sum* della sommatoria.

Thread in Win32

Nella API Win32 i nuovi thread si generano tramite la funzione CreateThread(); che, proprio come in Pthreads, accetta una serie di attributi del thread come parametri. Tali attributi includono le informazioni sulla sicurezza, la dimensione della pila e un indicatore (flag) per segnalare se il thread debba avere inizio nello stato d'attesa.

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 caratteristiche 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 fare l'override del suo metodo run(). L'alternativa, usata più comunemente, consiste nella definizione di una classe che implementi l'interfaccia *Runnable* (tenuta a definire almeno il metodo run()). Il codice che implementa run() sarà eseguito in un thread distinto.

La creazione di un oggetto di classe Thread non equivale a generare un nuovo thread: è il metodo start() che avvia effettivamente il nuovo thread. L'invocazione del metodo start() ha il duplice effetto di:

- 1) allocare la memoria e inizializzare un nuovo thread nella JVM
- 2) chiamare il metodo run(), cosa che rende il thread eseguibile dalla JVM.

Questioni di programmazione multithread

In questo paragrafo si affrontano alcune problematiche legate ai programmi multithread.

Chiamate di sistema fork ed exec

Abbiamo già descritto l'uso della chiamata di sistema fork() per la creazione di un nuovo processo tramite la duplicazione di un processo esistente. In un programma multithread la semantica delle system call fork() ed exec() cambia: se un thread in un programma invoca fork(), il nuovo processo potrebbe, in generale, contenere un duplicato 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 che già conosciamo: 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(), potrebbe essere utile una duplicazione di tutti i thread del processo genitore.

Cancellazione

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 annullati. Una situazione analoga potrebbe verificarsi quando un utente preme il pulsante di terminazione di un programma di consultazione del Web per interrompere il caricamento di una pagina.

Un thread da cancellare è spesso chiamato **target thread** (thread bersaglio). La cancellazione di un thread bersaglio può avvenire in due modi diversi:

1. **cancellazione asincrona**. Un thread fa immediatamente terminare il thread bersaglio;
2. **cancellazione deferred** (differita). Il thread bersaglio può periodicamente controllare se deve terminare, in modo da riuscirvi in modo opportuno.

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. Quest'ultimo caso è particolarmente problematico se si tratta di cancellazione asincrona. Il sistema operativo di solito si riappropria delle risorse di sistema usate da un thread cancellato, ma spesso non si riappropria di tutte le risorse. Quindi, la cancellazione di un thread in modo asincrono potrebbe non liberare una risorsa necessaria per tutto il sistema.

La cancellazione deferred invece funziona tramite un thread che segnala la necessità di cancellare un certo thread bersaglio; la cancellazione avviene soltanto quando il thread bersaglio verifica se debba essere o meno cancellato. Questo metodo permette di programmare la verifica in un punto dell'esecuzione in cui il thread sia cancellabile senza problemi. Nella libreria Pthreads questi punti si chiamano **cancellation point**.

Signal handling

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:

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

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. I segnali sincroni s'inviano allo stesso processo che ha eseguito l'operazione causa del segnale (questo è il motivo per cui si chiamano sincroni).

Quando un segnale è causato da un evento esterno al processo in esecuzione, tale processo riceve il segnale in modo asincrono. Esempi di segnali di questo tipo sono la terminazione di un processo richiesta con specifiche combinazioni di tasti oppure la scadenza di un timer. Di solito un segnale asincrono s'invia a un altro processo.

Ogni segnale si può gestire in due modi:

1. tramite un default handler (gestore predefinito di segnali);
2. tramite un user-defined signal handler (gestore di segnali definito dall'utente).

Per ogni segnale esiste un **default handler** che il kernel esegue quando deve gestire il segnale. La gestione predefinita è sostituibile da una funzione di **user-defined signal handler**, richiamata per gestire il segnale. Sia i segnali sincroni sia quelli asincroni sono gestibili in modi diversi: alcuni si possono semplicemente ignorare (per esempio, il ridimensionamento di una finestra); altri si possono gestire terminando l'esecuzione del programma (per esempio, un accesso illegale alla memoria).

Poiché nell'inviare un segnale è sufficiente fare riferimento al processo interessato, per i processi a singolo thread il signal handling (gestione dei segnali) è semplice. Per i processi multithread si pone il problema del thread cui si deve inviare il segnale. In generale esistono le seguenti possibilità:

1. inviare il segnale al thread cui il segnale si riferisce;
2. inviare il segnale a ogni thread del processo;
3. inviare il segnale a specifici thread del processo;
4. definire un thread specifico per ricevere tutti i segnali diretti al processo.

Il metodo per recapitare un segnale dipende dal tipo di segnale. I segnali sincroni, per esempio, si devono inviare al thread che ha generato l'evento causa del segnale e non ad altri thread nel processo. Se si tratta di segnali asincroni la situazione non è invece così chiara; alcuni segnali asincroni, come il segnale che termina un processo (come <control><C>), si devono inviare a tutti i thread. La maggior parte delle versioni multithread del sistema operativo UNIX permettono che per ciascun thread si indichino i segnali da accettare e quelli da bloccare. Quindi, alcuni segnali asincroni si potrebbero recapitare soltanto ai thread che non li bloccano.

Tuttavia, poiché i segnali vanno gestiti una sola volta, di solito un segnale è recapitato solo al primo thread che non lo blocca. La funzione UNIX per recapitare i segnali è `kill(aid_t aid, int signal)`, dove *aid* specifica il processo a cui recapitare il segnale *signal*. La API Pthreads POSIX, però, dispone anche della funzione `pthread_kill(pthread_t tid, int signal)` che permette di specificare il thread (*tid*) cui recapitare il segnale.

Thread pool

Riponiamoci nello scenario di un server Web multithread in cui per ogni richiesta ricevuta, il server crea un thread distinto per fornire il servizio richiesto. Nonostante la creazione di un thread distinto sia molto più vantaggiosa della creazione di un nuovo processo, 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 attivi in modo concorrente nel sistema. Un numero illimitato di thread potrebbe esaurire le risorse del sistema, come il tempo di CPU o la memoria. L'impiego della **thread pool** (gruppi di thread) 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 una pool (gruppo) in cui attendano il lavoro che gli sarà richiesto. Quando un server riceve una richiesta, attiva un thread della pool (se ce n'è uno disponibile) e gli passa la richiesta; dopo aver completato il suo lavoro, il thread rientra nella waiting pool (gruppo d'attesa). Se la pool non contiene alcun thread disponibile, il server attende fino al rientro di un thread. I vantaggi offerti sono i seguenti:

1. di solito il servizio di una richiesta tramite un thread esistente è più rapido, poiché elimina l'attesa della creazione di un nuovo thread;
2. 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.

Il numero di thread di una pool 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. Architetture più sofisticate possono correggere dinamicamente il numero di thread di una pool secondo schemi d'uso.

Thread specific data

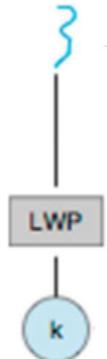
Uno dei vantaggi principali della programmazione multithread è dato dal fatto che i thread appartenenti allo stesso processo ne condividono i dati. Tuttavia, in particolari circostanze, ogni thread può necessitare di una copia privata di certi dati, chiamati dati specifici di thread. Per esempio, in un sistema per transazioni si può svolgere ciascuna transazione tramite un thread distinto e un identificatore unico per ogni transazione. Per associare ciascun thread al relativo identificatore si possono usare dati specifici dei thread.

Attivazione dello scheduler

Un'ultima questione da affrontare in merito ai programmi multithread riguarda la comunicazione tra la libreria del kernel e la libreria per i thread, che può rendersi necessaria nel Two-level model e in quello Many-to-Many. È proprio grazie a questa forma di coordinamento che il numero dei thread nel kernel è modificabile dinamicamente, con l'obiettivo di conseguire le migliori prestazioni.

Molti sistemi che implementano o il modello da molti a molti o quello a due livelli collocano una struttura dati intermedia tra i thread del kernel e dell'utente. Questa struttura dati, nota come processo leggero o LWP (acronimo di *Lightweight Process*) è mostrata nella figura a destra. Dal punto di vista della libreria di thread a livello utente, LWP si presenta come un processore virtuale a cui l'applicazione può richiedere lo scheduling di un thread a livello dell'utente.

Ciascun LWP è associato a un thread del kernel, e sono proprio i thread del kernel che il sistema operativo pone in esecuzione sui processori fisici. Se un thread del kernel si arresta (mentre attende il completamento di un'operazione di I/O, per esempio) anche LWP si blocca. L'effetto a catena risale fino al thread a livello utente associato a LWP, che si blocca anch'esso.



Per un'efficiente esecuzione un'applicazione può aver bisogno di un numero imprecisato di LPW. Si consideri un processo con prevalenza di elaborazione eseguito da un singolo processore. In questa situazione è eseguibile solo un thread per volta, dunque un LWP è sufficiente. Un'applicazione con prevalenza di I/O potrebbe, tuttavia, richiedere l'esecuzione di molteplici LWP. Di solito, è necessario un LWP per ogni chiamata di sistema concorrente bloccante. Supponiamo, per esempio, che giungano allo stesso tempo cinque richieste differenti per la lettura di file. Sono necessari cinque LWP, nel caso che tutte le richieste risiedano nel kernel in attesa di essere completate. Se un processo ha soltanto quattro LWP, la quinta richiesta deve attendere che uno degli LWP sia rilasciato dal kernel.

Uno dei modelli di comunicazione tra la libreria a livello utente e il kernel è conosciuto come **attivazione dello scheduler**. Il suo funzionamento è il seguente: il kernel fornisce all'applicazione una serie di processori virtuali (LWP), mentre l'applicazione esegue lo scheduling dei thread dell'utente sui processori virtuali disponibili. Inoltre, il kernel deve informare l'applicazione se si verificano determinati eventi, seguendo una procedura nota come **upcall**. Le upcall sono gestite dalla libreria dei thread mediante un apposito gestore, eseguito su un processore virtuale. Una situazione capace di innescare una upcall si verifica quando il thread di un'applicazione è sul punto di bloccarsi. In questo caso il kernel, tramite una upcall, informa l'applicazione che un thread è prossimo a bloccarsi, e identifica il thread in oggetto. Il kernel, quindi, assegna all'applicazione un nuovo processore virtuale. L'applicazione esegue un gestore della upcall su questo nuovo processore: il gestore salva lo stato del thread bloccante e rilascia il processore virtuale su cui era stato eseguito. Il gestore della upcall pianifica allora l'esecuzione di un altro thread sul processore virtuale che si è appena liberato. Quando si verifica l'evento atteso dal thread bloccante, il kernel fa un'altra upcall alla libreria dei thread per comunicare che il thread bloccato è nuovamente in condizione di essere eseguito. Il gestore di questa upcall necessita anch'esso di un processore virtuale: il kernel può creare uno ex novo, o sottrarlo a un thread utente per prelazione. L'applicazione contrassegna il thread fino ad allora bloccato come pronto per l'esecuzione, ed esegue lo scheduling di un thread pronto per l'esecuzione su un processore virtuale disponibile.

Thread di Linux

Linux offre la chiamata di sistema `fork()` per duplicare un processo, e prevede inoltre la chiamata di sistema `clone()` per generare nuovo thread. Tuttavia Linux non distingue tra processi e thread, impiegando generalmente al loro posto il termine **task** (operazione) in riferimento al flusso del controllo di un programma. Quando `clone()` è invocata, riceve come parametro un insieme di flag (indicatori), al fine di stabilire quante e quali risorse del task genitore debbano essere condivise dal task figlio. Alcuni di questi flag sono illustrati nello schema seguente.

Flag	significato
<code>CLONE_FS</code>	Condivisione delle informazioni sul file system
<code>CLONE_VM</code>	Condivisione dello stesso spazio di memoria
<code>CLONE_SIGHAND</code>	Condivisione dei gestori dei segnali
<code>CLONE_FILES</code>	Condivisione dei file aperti

Per esempio, qualora `clone()` riceva i flag `CLONE_FS`, `CLONE_VM`, `CLONE_SIGHAND` e `CLONE_FILES`, il task genitore e il task figlio condivideranno le medesime informazioni sul file system (come la directory attiva), lo stesso spazio di memoria, gli stessi gestori dei segnali e lo stesso insieme di file aperti. Adoperate `clone()` in questo modo è equivalente a creare thread, dal momento che il task genitore condivide la maggior parte delle proprie risorse con il task figlio. Tuttavia, se nessuno dei flag è impostato al momento dell'invocazione di `clone()`, non si ha alcuna condivisione, e la funzionalità ottenuta diventa simile a quella fornita dalla chiamata di sistema `fork()`.

Questa condivisione a intensità variabile è resa possibile dal modo in cui un task è rappresentato nel kernel di Linux. Per ogni task, nel kernel esiste un'unica struttura dati (e precisamente, struct `task_struct`). Questa struttura, invece di memorizzare i dati del task relativo, utilizza dei puntatori ad altre strutture dove i dati sono effettivamente contenuti: per esempio, strutture dati che rappresentano l'elenco dei file aperti, le informazioni per la gestione dei segnali e la memoria virtuale. Quando si invoca `fork()`, si crea un nuovo task insieme con una copia di tutte le strutture dati del task genitore. Anche quando s'invoca la chiamata `clone()` si crea un nuovo task, ma anziché ricevere una copia di tutte le strutture dati, il nuovo task punta a queste o a quelle strutture dati del task genitore, a seconda dell'insieme di flag passati a `clone()`.

5. Scheduling della CPU

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.

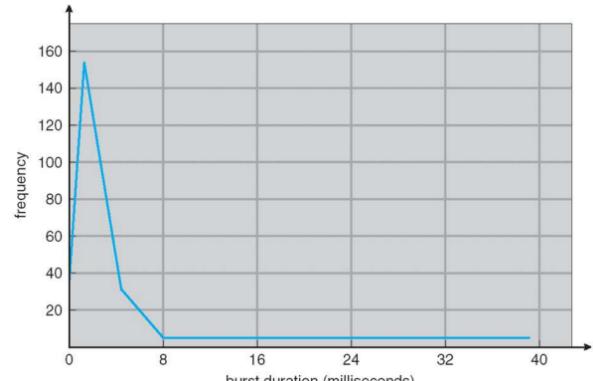
L'idea della multiprogrammazione è relativamente semplice. Un processo è in esecuzione finché non deve attendere un evento, generalmente il completamento di qualche richiesta di I/O; durante l'attesa, in un sistema di calcolo semplice, la CPU resterebbe inattiva, e tutto il tempo d'attesa sarebbe sprecato. 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.

Ciclicità delle fasi d'elaborazione e di I/O

Il successo dello scheduling della CPU dipende dall'osservazione della seguente proprietà dei processi: l'esecuzione del processo consiste in un ciclo **CPU burst** (ciclo di elaborazione svolto dalla CPU) e in un ciclo I/O burst (ciclo 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 (una "raffica") di operazioni d'elaborazione svolte dalla CPU (CPU burst), seguita da una sequenza di operazioni di I/O (I/O burst), quindi un'altra CPU burst, di nuovo una I/O burst e così via. L'ultima sequenza di operazioni della CPU si conclude con una richiesta al sistema di terminare l'esecuzione.

Le durate delle sequenze di operazioni della CPU sono state misurate, e sebbene varino molto secondo il processo e secondo il calcolatore, la loro curva di frequenza è simile a quella illustrata nella figura. La curva è generalmente di tipo esponenziale, con molte brevi sequenze di operazioni della CPU, e poche sequenze di operazioni della CPU molto lunghe. Un programma con prevalenza di I/O (I/O bound) produce generalmente molte sequenze di operazioni della CPU di breve durata. Un programma con prevalenza d'elaborazione (CPU bound), invece, produce poche sequenze di operazioni della CPU molto lunghe. Queste caratteristiche possono essere utili nella scelta di un appropriato algoritmo di scheduling della CPU.



Scheduler della CPU

Ogniqualvolta la CPU passa nello stato d'inattività, il sistema operativo sceglie per l'esecuzione uno dei processi presenti nella coda dei processi pronti. In particolare, è lo **short-term scheduler** (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 coda dei ready non è necessariamente una coda FIFO. Tuttavia, concettualmente tutti i processi della coda dei processi pronti sono posti nella lista d'attesa per accedere alla CPU. Generalmente gli elementi delle code sono i PCB (*process control block*) dei processi.

Scheduling preemptive

Le decisioni riguardanti lo scheduling della CPU si possono prendere nelle seguenti circostanze:

1. Un processo passa dallo stato running a waiting (per esempio, richiesta di I/O o funzione wait)

2. Un processo passa dallo stato running a ready (per esempio quando si verifica un interrupt)
3. Un processo passa da waiting a ready (per esempio, al completamento di un'operazione di I/O)
4. Un processo termina

I casi 1 e 4 in quanto tali non comportano alcuna scelta di scheduling; a essi segue la scelta di un nuovo processo da eseguire, sempre che ce ne sia almeno uno nella coda dei processi pronti per l'esecuzione. Una scelta si deve 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 diritto di prelazione (**nonpreemptive**); altrimenti, lo schema di scheduling è con diritto di prelazione (**preemptive**). Nel caso dello scheduling senza diritto di 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 waiting.

Sfortunatamente lo scheduling con diritto di prelazione presenta un inconveniente. Si consideri il caso in cui due processi condividono dati; mentre uno di questi aggiorna i dati si ha la sua prelazione in favore dell'esecuzione dell'altro. Il secondo processo può, a questo punto, tentare di leggere i dati che sono stati lasciati in uno stato incoerente dal primo processo. Sono quindi necessari nuovi meccanismi per coordinare l'accesso ai dati condivisi (vedi [Sincronizzazione dei processi](#)).

La capacità di prelazione si ripercuote anche sulla progettazione del kernel del sistema operativo. Durante l'elaborazione di una chiamata di sistema, il kernel può essere impegnato in attività in favore di un processo; tali attività possono comportare la necessità di modifiche a importanti dati del kernel, come le code di I/O. Se si ha la prelazione del processo durante tali modifiche e il kernel (o un driver di dispositivo) deve leggere o modificare gli stessi dati, si può avere il caos. Alcuni sistemi operativi, tra cui la maggior parte delle versioni dello UNIX, affrontano questo problema attendendo il completamento della chiamata di sistema o che si abbia il blocco dell'I/O prima di eseguire un cambio di contesto. Quindi, il kernel non può esercitare la prelazione su un processo mentre le strutture dati del kernel si trovano in uno stato incoerente.

Dispatcher

Un altro elemento coinvolto nella funzione di scheduling della CPU è il **dispatcher** (è praticamente l'esecutore delle decisioni prese dallo scheduler); si tratta del modulo che passa effettivamente il controllo della CPU ai processi scelti dallo scheduler a breve termine. Questa funzione riguarda il cambio di contesto, il passaggio all'user mode e il salto alla giusta posizione del programma utente per riavviarne 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 nota come **latenza di dispatch**.

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. Per il confronto tra gli algoritmi di scheduling della CPU sono stati suggeriti molti criteri. 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 può variare dal 40 per cento, per un sistema con poco carico, al 90 per cento, per un sistema con utilizzo intenso.

- **Throughput** (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 throughput. Per processi di lunga durata questo rapporto può essere di un processo all'ora, mentre per brevi transazioni è possibile avere una produttività di 10 processi al secondo.
- **Turnaround time** (tempo di completamento). Considerando un processo particolare, un criterio importante può essere relativo al tempo necessario per eseguire il processo stesso. L'intervallo che intercorre tra la sottomissione del processo e il completamento dell'esecuzione è chiamato turnaround time, ed è la somma dei tempi passati nell'attesa dell'ingresso in memoria, nella coda dei processi pronti, durante l'esecuzione nella CPU e nelle operazioni di I/O.
- **Waiting time** (tempo di 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 coda dei processi pronti. Il tempo d'attesa è la somma degli intervalli d'attesa passati nella coda dei processi pronti.
- **Response time** (tempo di risposta). In un sistema interattivo il tempo di completamento può non essere il miglior criterio di valutazione: spesso accade che un processo emetta dati abbastanza presto, e continua a calcolare i nuovi risultati mentre quelli precedenti sono in fase d'emissione. Quindi, un'altra misura di confronto è data dal tempo che intercorre tra la sottomissione 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 d'emissione. Generalmente il tempo di completamento è limitato dalla velocità del dispositivo d'emissione dei dati.

È auspicabile aumentare al massimo utilizzo e produttività della CPU, mentre il tempo di completamento, il tempo d'attesa e il tempo di risposta si devono ridurre al minimo. Nella maggior parte dei casi si ottimizzano i valori medi; in alcune circostanze è più opportuno ottimizzare i valori minimi o massimi, anziché i valori medi; per esempio, per garantire che tutti gli utenti ottengano un buon servizio, può essere utile ridurre il massimo tempo di risposta.

Per i sistemi interattivi, come i sistemi a tempo ripartito, alcuni analisti suggeriscono che sia più importante ridurre al minimo la varianza del tempo di risposta anziché il tempo medio di risposta (praticamente mantenendo costante il "ritardo" si abitua l'utente all'attesa). Un sistema il cui tempo di risposta sia ragionevole e prevedibile può essere considerato migliore di un sistema mediamente più rapido, ma molto variabile. Tuttavia, è stato fatto poco sugli algoritmi di scheduling della CPU al fine di ridurre al minimo la varianza.

Algoritmi di scheduling

Lo scheduling della CPU riguarda la scelta dei processi presenti nella coda dei processi pronti cui assegnare la CPU. Descriviamo di seguito alcuni dei tanti algoritmi di scheduling della CPU.

First-Come, First-Served (FCFS) Scheduling

Lo scheduling FCFS è il più semplice nonché il meno efficiente algoritmo di scheduling della CPU. Con questo schema la CPU si assegna al processo che la richiede per primo. La realizzazione del criterio FCFS si fonda su una coda FIFO. Quando un processo entra nella coda dei processi pronti, si collega il suo PCB all'ultimo elemento della coda. Quando è libera, si assegna la CPU al processo che si trova alla testa della coda dei processi pronti, rimuovendolo da essa.

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:

<u>Process</u>	<u>Burst Time</u>	
P_1	24	
P_2	3	
P_3	3	0

Se i processi arrivano nell'ordine P_1, P_2, P_3 e sono serviti in ordine FCFS, si ottiene il risultato illustrato dal precedente **diagramma di Gantt** (la realizzazione di Gantt chart dati i processi e i loro tempi è una tipica domanda d'esame), un istogramma che illustra una data pianificazione includendo i tempi d'inizio e fine di ogni processo partecipante.

Il tempo di attesa è 0 millisecondi per il processo P_1 , 24 millisecondi per P_2 e 27 per P_3 . Quindi il tempo di attesa medio è $\frac{0+24+27}{3} = 17$ millisecondi.

Se i processi arrivassero nell'ordine P_2, P_3, P_1 , il tempo di attesa medio sarà di soli $\frac{6+0+3}{3} = 3$ millisecondi.



Quindi, il tempo medio d'attesa in condizioni di FCFS non è in genere minimo, e può variare sostanzialmente al variare della durata delle sequenze di operazioni della CPU dei processi.

Se consideriamo un processo CPU-bound dietro tanti processi I/O bound si ha il cosiddetto effetto convoy (effetto convoglio): tutti i processi attendono che un lungo processo liberi la CPU, 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).

Shortest-Job-First (SJF) Scheduling

Un criterio diverso di scheduling della CPU si può ottenere con l'algoritmo SJF. Questo algoritmo associa a ogni processo la lunghezza della successiva sequenza di operazioni della CPU (**burst time**). Quando è disponibile, si assegna la CPU al processo che ha il più breve burst time. Se due processi hanno lo stesso burst time si applica lo scheduling FCFS.

Come esempio si consideri il seguente insieme di processi, con burst time misurato in millisecondi:

<u>Process</u>	<u>Burst Time</u>	
P_1	6	
P_2	8	
P_3	7	
P_4	3	

Avremo il seguente diagramma di Gantt secondo SJF:

P_4	P_1	P_3	P_2				
0	3	9	16	24			
24							

Il tempo di attesa medio è di $\frac{3+16+9+0}{4} = 7$ millisecondi. Usando lo scheduling FCFS, il tempo d'attesa sarebbe di 10,25 millisecondi (considerando il seguente ordine di arrivo: P_1, P_2, P_3, P_4). Spostando un processo breve prima di un processo lungo, il tempo d'attesa per il processo breve diminuisce più di quanto aumenti il tempo d'attesa per il processo lungo. Di conseguenza, il tempo d'attesa medio diminuisce.

La difficoltà reale implicita nell'algoritmo SJF consiste nel conoscere la durata della successiva richiesta della CPU. Un possibile metodo consiste nel tentare di approssimare lo scheduling SJF: se non è possibile conoscere il burst time della CPU, si può cercare di stimare il suo valore; è probabile, infatti, che sia simile ai precedenti. Quindi, calcolando un valore approssimato della lunghezza, si può scegliere il processo con la più breve fra tali lunghezze previste.

Il CPU burst generalmente si ottiene calcolando la media esponenziale delle effettive lunghezze delle precedenti sequenze di operazioni della CPU. La media esponenziale si definisce con la formula seguente. Denotando t_n la lunghezza dell' n -esima sequenza di operazioni della CPU e con τ_{n+1} il valore previsto per la successiva CPU burst, con α tale che $0 \leq \alpha \leq 1$ si definisce

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

Il valore di t_n contiene le informazioni più recenti; τ_n registra la storia passata. Se $\alpha = 0$, allora $\tau_{n+1} = \tau_n$, e la storia recente non ha effetto; si suppone, cioè, che le condizioni attuali siano transitorie; se $\alpha = 1$, allora

$\tau_{n+1} = t_n$ e ha significato solo la più recente sequenza di operazioni della CPU: si suppone, cioè, che la storia sia vecchia e irrilevante. Più comune è la condizione in cui $\alpha = \frac{1}{2}$, valore che indica che la storia recente e la storia passata hanno lo stesso peso. Il τ_0 iniziale si può definire come una costante o come una media complessiva del sistema. Nella figura è illustrata una media esponenziale con $\alpha = \frac{1}{2}$ e $\tau_0 = 10$.

Per comprendere il comportamento della media esponenziale, si può sviluppare la formula per τ_{n+1} sostituendo in τ_n , in modo da ottenere

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha t_{n-1} + \cdots + (1 - \alpha)^j \alpha t_{n-j} + \cdots + (1 - \alpha)^{n+1} \tau_0$$

Poiché sia α sia $(1 - \alpha)$ sono minori o uguali a 1, ogni termine ha peso inferiore a quello del suo predecessore.

'algoritmo SJF può essere sia con prelazione sia senza prelazione. La scelta si presenta quando alla coda dei processi pronti arriva un nuovo processo mentre un altro processo è ancora in esecuzione. Il nuovo processo può avere una successiva sequenza di operazioni della CPU più breve di quella che resta al processo correntemente in esecuzione. Un algoritmo SJF con prelazione sostituisce il processo attualmente in esecuzione, mentre un algoritmo SJF senza prelazione permette al processo correntemente in esecuzione di portare a termine la propria sequenza di operazioni della CPU. (Lo scheduling SJF con prelazione è talvolta chiamato **scheduling shortest-remaining-time-first**).

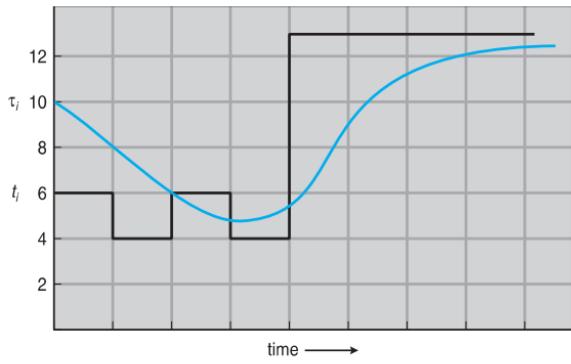
Come esempio, si considerino i quattro processi seguenti:

Process	Arrival Time	Burst Time	
P_1	0	8	
P_2	1	4	
P_3	2	9	
P_4	3	5	

All'istante 0 si avvia il processo P_1 , poiché l'unico che si trova nella coda. All'istante 1 arriva il processo P_2 . Il tempo necessario per completare il processo P_1 (7 millisecondi) è maggiore del tempo richiesto dal processo P_2 (4 millisecondi), perciò si ha la prelazione sul processo P_1 sostituendolo col processo P_2 . Il tempo d'attesa medio per questo esempio è $\frac{(10-1)+(1-1)+(17-2)+(5-3)}{4} = 6,5$ millisecondi. Con uno scheduling SJF senza prelazione si otterebbe un tempo d'attesa medio di 7,75 millisecondi.

Scheduling con priorità

L'algoritmo SJF è un caso particolare del più generale algoritmo di scheduling per 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. Un algoritmo SJF è semplicemente un algoritmo con priorità dove ad un maggiore burst time corrisponde una minore priorità, e viceversa.



CPU burst (t_i)	6	4	6	4	13	13	13	...	
"guess" (τ_i)	10	8	6	6	5	9	11	12	...

Occorre notare che la discussione si svolge nei 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 si è ancora stabilito se 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. Di seguito forniamo un rapido esempio dove a numeri bassi corrispondono priorità alte:

<u>Process</u>	<u>Burst Time</u>	<u>Priority</u>	
P_1	10	3	
P_2	1	1	P_2 P_5
P_3	2	4	0 1 6
P_4	1	5	
P_5	5	2	Tempo di attesa medio = 8.2 msec

Lo scheduling per priorità può essere sia con prelazione sia senza prelazione. Quando un processo arriva alla coda dei processi pronti, si confronta la sua priorità con quella del processo attualmente in esecuzione. Un algoritmo di scheduling per priorità e con diritto di prelazione sottrae la CPU al processo attualmente in esecuzione se la priorità dell'ultimo processo arrivato è superiore. Un algoritmo di scheduling senza diritto di prelazione si limita a porre l'ultimo processo arrivato alla testa della coda dei processi pronti.

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

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

Round Robin (RR)

Nell'algoritmo Round Robin (anche detto scheduling circolare) ciascun processo riceve una piccola quantità fissata del tempo della CPU q , chiamata **quanto di tempo** o porzione di tempo, che varia generalmente da 10 a 100 millisecondi; la coda dei processi pronti è trattata come una coda circolare. Lo scheduler della CPU scorre la coda dei processi pronti, assegnando la CPU a ciascun processo per un intervallo di tempo della durata massima di un quanto di tempo.

Per realizzare lo scheduling RR si gestisce la coda dei processi pronti come una coda FIFO. I nuovi processi si aggiungono alla fine della coda dei processi pronti. Lo scheduler della CPU individua il primo processo dalla coda dei processi pronti, 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 l'effettiva esecuzione del processo.

A questo punto si può verificare una delle 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 coda dei processi pronti; oppure la durata della sequenza di operazioni della CPU del processo attualmente in esecuzione è più lunga di un quanto di tempo; in questo caso si raggiunge la scadenza del quanto di tempo e il timer invia un segnale d'interruzione al sistema operativo, che esegue un cambio di contesto, aggiunge il processo alla fine della coda dei processi pronti e, tramite lo scheduler della CPU, seleziona il processo successivo nella coda dei processi pronti. 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:

<u>Process</u>	<u>Burst Time</u>	
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.

Nell'algoritmo di scheduling RR la CPU si assegna a un processo per non più di un quanto di tempo per volta. 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 coda dei processi pronti. L'algoritmo di scheduling RR è pertanto con prelazione.

Se nella coda dei processi pronti esistono n processi e il quanto di tempo è pari a q , ciascun processo ottiene $\frac{1}{n}$ -esimo del tempo di elaborazione della CPU in frazioni di, al massimo, q unità di tempo. Ogni processo non deve attendere per più di $(n - 1)q$ unità di tempo. Per esempio, dati cinque processi e un quanto di tempo di 20 millisecondi, ogni processo usa 20 millisecondi ogni 100 millisecondi.

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 (indefinito), il criterio di scheduling RR si riduce al criterio di scheduling FCFS. Se il quanto di tempo è molto breve (per esempio, un microsecondo), il criterio RR si chiama **condivisione della CPU** (*processor sharing*) e teoricamente gli utenti hanno l'impressione che ciascuno degli n processi disponga di una propria CPU in esecuzione a $\frac{1}{n}$ della velocità della CPU reale.

Riguardo alle prestazioni dello scheduling RR, occorre tuttavia considerare l'effetto del context switch. Dato un solo processo della durata di 10 unità di tempo, se il quanto di tempo è di 12 unità, il processo impiega meno di un quanto di tempo; se però il quanto di tempo è di 6 unità, il processo richiede 2 quanti di tempo e un cambio di contesto; e se il quanto di tempo è di un'unità di tempo, occorrono nove cambi di contesto, con proporzionale rallentamento dell'esecuzione del processo.

Anche il tempo di completamento (turnaround time) dipende dalla dimensione del quanto di tempo: il tempo di completamento medio di un insieme di processi non migliora necessariamente con l'aumento della dimensione del quanto di tempo. In generale, il tempo di completamento medio può migliorare se la maggior parte dei processi termina la successiva sequenza di operazioni della CPU in un solo quanto di tempo. Per esempio, dati tre processi della durata di 10 unità di tempo ciascuno e un quanto di una unità di tempo, il tempo di completamento medio è di 29 unità. Se però il quanto di tempo è di 10 unità, il tempo di completamento medio scende a 20 unità. Aggiungendo il tempo del cambio di contesto, con un piccolo quanto di tempo, il tempo di completamento medio aumenta poiché sono richiesti più cambi di contesto.

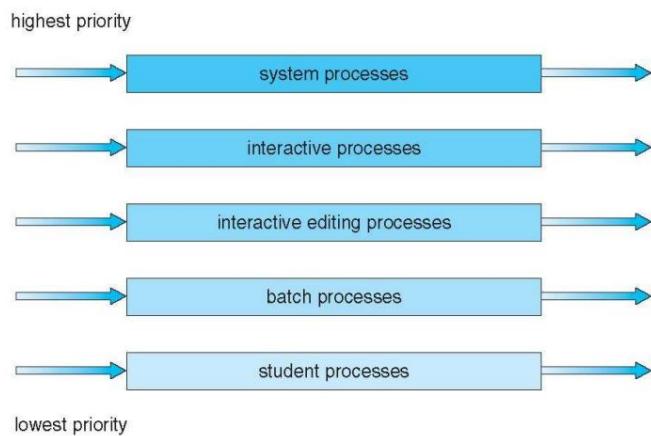
E bene comunque che il quanto di tempo sia maggiore del tempo necessario al cambio di contesto: d'altro canto, è opportuno che lo scarto non sia eccessivo: anche se il quanto di tempo è molto ampio, il criterio di scheduling RR tende al criterio FCFS. Empiricamente si può stabilire che l'80 per cento delle sequenze di operazioni della CPU debba essere più breve del quanto di tempo.

N.B.: Domanda tipica d'esame: Con RR si può avere starvation? Risposta: No.

Scheduling a code multiple

E stata creata una classe di algoritmi di scheduling adatta a situazioni in cui i processi si possono classificare facilmente in gruppi diversi. Una distinzione diffusa 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. Questi due tipi di processi hanno tempi di risposta diversi e possono quindi avere diverse necessità di scheduling. Inoltre, i processi che si eseguono in primo piano possono avere la priorità, definita esternamente, sui processi che si eseguono in sottofondo.

L'algoritmo di scheduling a code multiple (multilevel queue scheduling algorithm) suddivide la coda dei processi pronti in code distinte (vedi figura). 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. 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 per priorità fissa e con prelazione. Per esempio, la coda dei processi in foreground può avere priorità assoluta sulla coda dei processi in background.



Esiste anche la possibilità di impostare i quanti di tempo per le code. Per ogni coda si stabilisce una parte del tempo d'elaborazione della CPU, suddivisibile a sua volta tra i processi che la costituiscono.

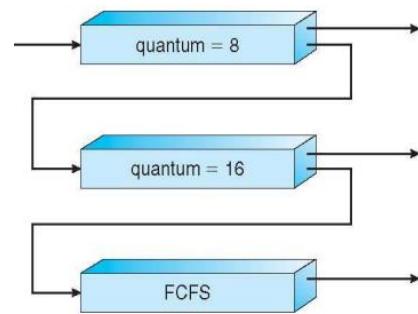
Nell'esempio precedente, si può assegnare l'80 per cento del tempo d'elaborazione della CPU alla coda dei processi in primo piano, per lo scheduling RR tra i suoi processi; mentre per la coda dei processi in sottofondo si riserva il 20 per cento del tempo d'elaborazione della CPU, da assegnare ai suoi processi sulla base del criterio FCFS.

Code multiple con feedback

Lo scheduling a code multiple con feedback permette ai processi di spostarsi fra le code. L'idea consiste nel separare i processi che hanno caratteristiche diverse nelle sequenze delle operazioni della CPU. Se un processo usa troppo tempo di elaborazione della CPU, viene spostato in una coda con priorità più bassa. Questo schema mantiene i processi con prevalenza di i/o e i processi interattivi nelle code con priorità più elevata. Analogamente, si può spostare in una coda con priorità più elevata un processo che attende troppo a lungo. In questo modo si attua una forma d'invecchiamento che impedisce il verificarsi di un'attesa indefinita.

Si consideri, per esempio, uno scheduler a code multiple con retroazione con tre code, numerate da 0 a 2, come nella figura a destra. Lo scheduler fa eseguire tutti i processi presenti nella coda 0; quando la coda 0 è vuota, si eseguono i processi nella coda 1; analogamente, i processi nella coda 2 si eseguono solo se le code 0 e 1 sono vuote. Un processo in ingresso nella coda 1 ha la prelazione sui processi della coda 2; un processo in ingresso nella coda 0, a sua volta, ha la prelazione sui processi della coda 1.

All'ingresso nella coda dei processi pronti, i processi vengono assegnati alla coda 0 e ottengono un quanto di tempo di 8 millisecondi; i processi che non terminano entro



tale quanto di tempo vengono spostati alla fine della coda 1. Se la coda 0 è vuota, si assegna un quanto di tempo di 16 millisecondi al processo alla testa della coda 1, ma se questo non riesce a completare la propria esecuzione, viene sottoposto a prelazione e messo nella coda 2. Se le code 0 e 1 sono vuote, si eseguono i processi della coda 2 secondo il criterio FCFS.

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; anche i processi che necessitano di più di 8 ma di non più di 24 millisecondi (coda 1) vengono serviti rapidamente. I processi più lunghi finiscono nella coda 2 e sono serviti secondo il criterio FCFS all'interno dei cicli di CPU lasciati liberi dai processi delle code 0 e 1.

Generalmente uno scheduler a code multiple 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.

La definizione di uno scheduler a code multiple con retroazione costituisce il più generale criterio di scheduling della CPU, che nella fase di progettazione si può adeguare a un sistema specifico.

Sfortunatamente corrisponde anche all'algoritmo più complesso; la definizione dello scheduler migliore richiede infatti particolari metodi per la selezione dei valori dei diversi parametri.

Real-Time CPU Scheduling

Suddividiamo i sistemi real time in due categorie:

- **Soft real-time system:** non ci sono garanzie su quando un processo verrà schedulato, solo processi critici preferiti ai non critici.
- **Hard real-time system:** task servito prima di una deadline

Importante per un sistema real time è la latenza di un evento (tempo trascorso tra evento e risposta del sistema), due tipi di latenze incidono sulla prestazione:

1. **Interrupt latency:** tempo dall'arrivo dell'interrupt allo start della routine di servizio.
Durante gli update delle strutture del kernel gli interrupt sono disabilitati, in sistemi real time, va minimizzato questo tempo
2. **Dispatch latency:** tempo di scheduling per fare lo switch di un altro processo.
Fase di conflitto e fase di dispatch del processo.
Fase di gestione del **conflitto** durante la dispatch latency;
 - a. Prelazione di qualunque processo in kernel mode
 - b. Rilascio delle risorse occupate dai processi a bassa priorità quando richieste dai processi ad alta qualità

Nel real-time scheduling lo scheduler deve supportare preemptive e priority-based scheduling ma questo garantisce solo il soft real-time. Infatti, per hard real time devono anche essere garantite le deadlines.

Rate Monotonic Scheduling

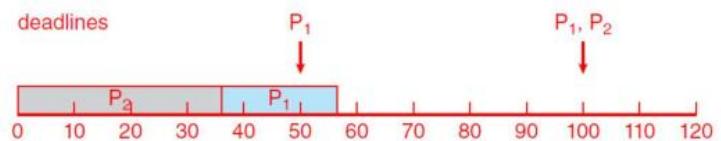
Consideriamo task periodici che richiedono la CPU a intervalli costanti. Ottenuta la CPU il task ha un tempo di processo t , deadline d , e periodo p con $0 \leq t \leq d \leq p$. Il **rate** di un task periodico sarà $\frac{1}{p}$.

In base alla deadline dichiarata dal processo, lo scheduler può ammettere o meno il processo; in caso di ammissione lo scheduler porterà a termine il processo prima della deadline dichiarata.

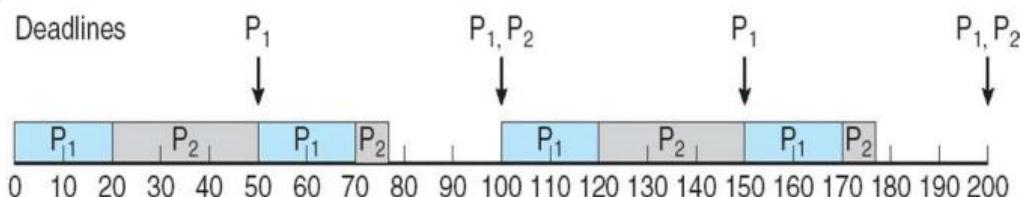
Nel rate monotonic scheduling la priorità viene assegnata in base all'inverso del periodo (ovvero al rate). Periodi brevi avranno priorità alta mentre periodi lunghi corrispondono priorità basse. Prendiamo ad esempio due processi: P_1 con periodo 50 e tempo di processamento 20; P_2 con periodo 50 e tempo di processamento 35. Per entrambe la deadline sarà quella di terminare prima del prossimo periodo.

L'uso di CPU per P_1 è di $\frac{20}{50} = 0.4$, mentre per P_2 è di $\frac{35}{100} = 0.35$. Dunque, l'uso totale della CPU è di 0.75.

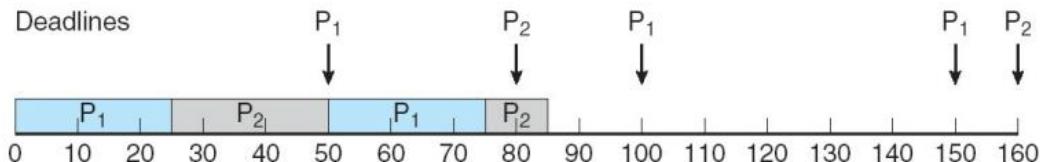
Assumendo P_2 con priorità più alta di P_1
risulterà una deadline persa per P_1 nonostante
l'uso totale di CPU sia inferiore del 100%.



Se assumiamo lo scheduling RMS e dunque sarà P_1 ad avere priorità più alta di P_2 entrambe le deadline saranno rispettate:



Ma anche con il rate monotonic scheduling esistono casi in cui non si riesce a schedulare, ad esempio se P_1 ha periodo 50 e CPU burst 25 mentre P_2 ha periodo 80 e CPU burst 35 risulterà un utilizzo di CPU pari a $\frac{25}{50} + \frac{35}{80} = 0.94$ ma P_2 non riuscirà a rispettare la sua deadline:



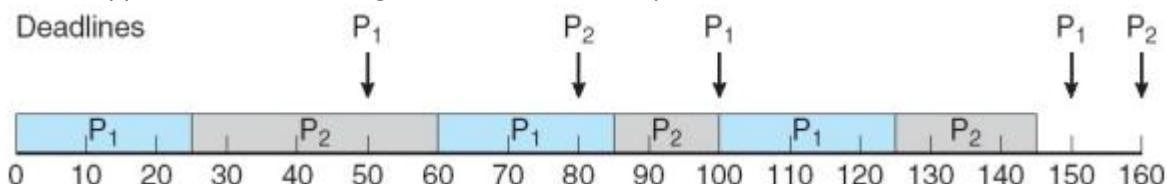
La causa di ciò è che il RMS non sfrutta completamente la CPU, l'utilizzo diminuisce con il numero di processi secondo tale formula: $N \left(2^{\frac{1}{N}} - 1 \right)$. Dunque, con 1 processo posso sfruttare il 100% della CPU, con due processi l'83% e con al crescere di N solo il 69%.

Si può notare che questo algoritmo si basa su priorità statiche; ovvero, decido a priori la priorità del processo e questa rimane fissa fino alla fine.

Earliest Deadline First Scheduling (EDF)

Nello scheduling EDF le priorità sono assegnate secondo le deadline: prima la deadline, più alta è la priorità; più lontana è la deadline, più bassa è la priorità (le priorità sono dinamiche, variano nel tempo).

Poniamoci nello stesso esempio precedente con P_1 di periodo 50 e CPU burst 25 e P_2 di periodo 80 e CPU burst 35. Applicando lo scheduling EDF riusciremmo a rispettare tutte le deadline:



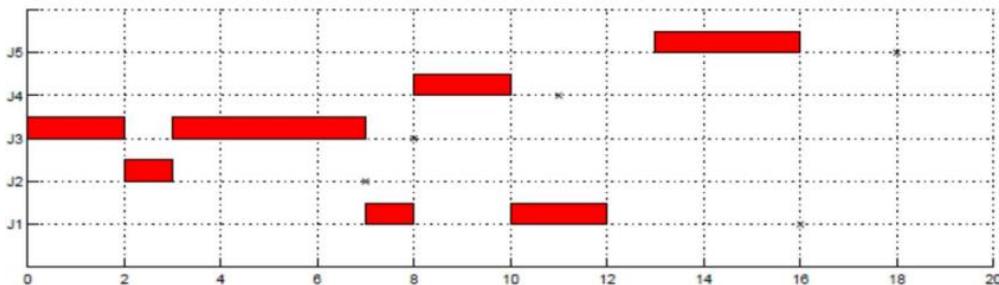
Questo tipo di scheduling non richiede la periodicità dei processi e neppure un CPU burst costante; ogni processo deve dichiarare la deadline quando diventa runnable. L'EDF è teoricamente ottimo (se utilizzo sotto il 100% di CPU rispetta sempre la deadline) però va considerato il context switch e il costo di gestione dell'interrupt.

Forniamo un ulteriore esempio fornendo la soluzione di un tipico esercizio d'esame:

- Tracciare il Gantt secondo EDF verificando se le scadenze sono rispettate (scheduling ammissibile):

Processo	Tempo di arrivo	Esecuzione	Scadenza
J_1	0	3	16
J_2	2	1	7
J_3	0	6	8
J_4	8	2	11
J_5	13	3	18

- Soluzione:



Scheduling dei Thread

La prima distinzione fra thread a livello utente e a livello kernel riguarda il modo in cui è pianificata la loro esecuzione. Nei sistemi che impiegano il modello [many-to-one](#) e il modello [many-to-many](#), la libreria dei thread pianifica l'esecuzione dei thread a livello utente su un LWP libero; si parla allora di **process-contention scope** (ambito della competizione ristretto al processo), in breve **PCS**, perché la contesa per aggiudicarsi la CPU ha luogo fra thread dello stesso processo. In realtà, affermando che la libreria dei thread pianifica l'esecuzione dei thread a livello utente associandoli agli LWP liberi, non si intende che il thread sia in esecuzione su una CPU; ciò avviene solo quando il sistema operativo pianifica l'esecuzione di thread del kernel su un processore fisico. Per determinare quale thread a livello kernel debba essere eseguito da una CPU, il kernel esamina i thread di tutto il sistema; si parla allora di **system-contention scope** (ambito della competizione allargato al sistema), in breve **SCS**. Quindi, nel caso di SCS, tutti i thread del sistema competono per l'uso della CPU. I sistemi caratterizzati dal modello [one-to-one](#) (quali Windows XP e Linux) pianificano i thread unicamente sulla base di SCS.

Se l'ambito della competizione è ristretto al processo, lo scheduling è solitamente basato sulle priorità: lo scheduler sceglie per l'esecuzione il thread con priorità più alta. Le priorità dei thread a livello utente sono stabilite dal programmatore, e la libreria dei thread non le modifica; alcune librerie danno facoltà al programmatore di cambiare la priorità di un thread. Si noti che quando l'ambito della competizione è ristretto al processo si è soliti applicare la prelazione al thread in esecuzione, a vantaggio di thread con priorità più alta; tuttavia, se i thread sono dotati della medesima priorità, non vi è garanzia sulla ripartizione del tempo.

Pthread Scheduling API

Per specificare l'ambito della contesa Pthreads usa i valori seguenti:

- `PTHREAD_SCOPE_PROCESS` pianifica i thread con lo scheduling PCS
- `PTHREAD_SCOPE_SYSTEM` pianifica i thread tramite lo scheduling SCS

Nei sistemi che si avvalgono del modello da many-to-many, la politica PTHREAD_SCOPE_PROCESS pianifica i thread a livello utente sugli LWP disponibili. Il numero di LWP viene stabilito dalla libreria dei thread, che in qualche caso si serve delle attivazioni dello scheduler. La seconda politica, PTHREAD_SCOPE_SYSTEM, crea una corrispondenza di ciascun thread a livello utente, un LWP a esso vincolato, realizzando così una corrispondenza secondo il modello da many-to-one.

Lo IPC di Pthread offre due funzioni per appurare e impostare l'ambito della contesa:

- `pthread_attr_setscope(pthread_attr_t *attr, int scope)`
- `pthread_attr_getscope(pthread_attr_t *attr, int scope)`

Il primo parametro per entrambe le funzioni è un puntatore agli attributi del thread. Il secondo parametro della funzione `pthread_attr_setscope()` riceve uno dei valori PTHREAD_SCOPE_PROCESS oppure PTHREAD_SCOPE_SYSTEM, che stabiliscono l'ambito della contesa. Qualora si verifichi un errore, ambedue le funzioni restituiscono valori non nulli.

Presentiamo di seguito un programma Pthread che determina l'ambito della contesa in vigore e lo imposta a PTHREAD_SCOPE_PROCESS; quindi crea cinque thread distinti, che andranno in esecuzione secondo il modello di scheduling SCS.

```
1 #include <pthread.h>
2 #include <stdio.h>
3 #define NUM_THREADS 5
4
5 -int main(int argc, char *argv[]) {
6     int i, scope;
7     pthread_t tid[NUM_THREADS];
8     pthread_attr_t attr;
9
10    // ottiene gli attributi di default
11    pthread_attr_init(&attr);
12
13    // per prima cosa appura lo scope dello scheduling
14    if (pthread_attr_getscope(&attr, &scope) != 0)
15        fprintf(stderr, "impossibile appurare lo scope\n");
16    else {
17        if (scope == PTHREAD_SCOPE_PROCESS)
18            printf("PTHREAD_SCOPE_PROCESS");
19        else if (scope == PTHREAD_SCOPE_SYSTEM)
20            printf("PTHREAD_SCOPE_SYSTEM");
21        else
22            fprintf(stderr, "Valore non ammesso.\n");
23    }
24
25    // imposta l'algoritmo di scheduling a PCS o SCS
26    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);
27
28    //genera i thread
29    for (i = 0; i < NUM_THREADS; ++i)
30        pthread_create(&tid[i], &attr, runner, NULL);
31
32    //aspetta la terminazione di tutti i thread
33    for (i = 0; i < NUM_THREADS; ++i)
34        pthread_join(tid[i], NULL);
35 }
```

Scheduling per sistemi multiprocessore

Fin qui la trattazione ha riguardato i problemi inerenti lo scheduling della CPU in un sistema a processore singolo; se sono disponibili più unità d'elaborazione, anche il problema dello scheduling è proporzionalmente più complesso. Si considerano i sistemi in cui le unità d'elaborazione sono, in relazione alle loro funzioni, identiche (**sistemi omogenei**): si può usare qualunque unità d'elaborazione disponibile per eseguire qualsiasi processo presente nella coda.

Soluzioni di scheduling per multiprocessori

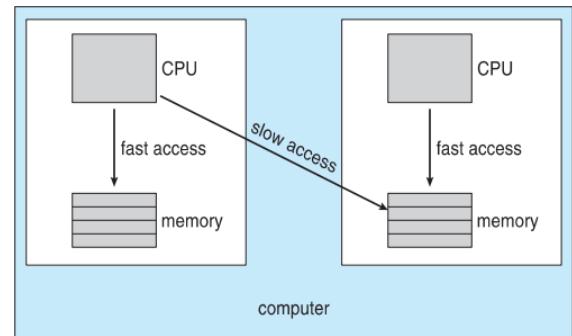
Una prima strategia di scheduling della CPU per i sistemi multiprocessore affida tutte le decisioni, l'elaborazione dell'I/O e le altre attività del sistema a un solo processore, il cosiddetto master server. Gli altri processori eseguono soltanto il codice dell'utente. Si tratta della **multielaborazione asimmetrica** (*asymmetric multiprocessing*), 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 coda 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.

Predilezione per il processore

Si consideri che cosa accade alla memoria cache dopo che un processo sia stato eseguito da uno specifico processore: i dati che il processore ha trattato da ultimo permangono nella cache e, di conseguenza, i successivi accessi alla memoria da parte del processo tendono a utilizzare spesso la memoria cache. Ecco allora che cosa succede se un processo si sposta su un altro processore: i contenuti della memoria cache devono essere invalidati sul processore di partenza, mentre la cache del processore di arrivo deve essere nuovamente riempita. A causa degli alti costi di svuotamento e riempimento della cache, molti sistemi SMP tentano di impedire il passaggio di processi da un processore all'altro, mirando a mantenere un processo sullo stesso processore che lo sta eseguendo. Si parla di predilezione per il processore (**processor affinity**), intendendo con ciò che un processo ha una predilezione per il processore su cui è in esecuzione.

La predilezione per il processore può assumere varie forme. Quando un sistema operativo si propone di mantenere un processo su un singolo processore, ma non garantisce che sarà così, si parla di predilezione debole (**soft affinity**). In questo caso è possibile che un processo migri da un processore all'altro. Alcuni sistemi, per esempio Linux, dispongono di chiamate di sistema con cui specificare che un processo non debba cambiare processore; in tal modo, si realizza la predilezione forte (**hard affinity**).

L'architettura della memoria principale di un sistema può influenzare le questioni relative alla predilezione. La figura mostra un'architettura con accesso non uniforme alla memoria (NUMA) in cui la CPU ha un accesso più rapido ad alcune zone di memoria rispetto ad altre. Di solito una situazione di questo tipo si ha in sistemi dove sono presenti diverse schede, ognuna con CPU e memoria proprie. Le CPU su una scheda possono accedere alla memoria sulla stessa scheda con meno ritardo rispetto a quella su schede diverse. Se lo scheduler della CPU di un sistema operativo e gli algoritmi di allocazione della memoria lavorano insieme, allora un processo con predilezione per una determinata CPU può essere allocato nella memoria residente sulla stessa scheda in cui è montata la CPU. Questo esempio mostra anche come i sistemi operativi non siano definiti e implementati in maniera tanto nitida quanto è descritto nei libri di testo. Le linee di demarcazione



tra le componenti di un sistema operativo, lungi dall'essere nette, sono sfumate; opportuni algoritmi instaurano poi connessioni fra le varie parti allo scopo di ottimizzare le prestazioni e l'affidabilità.

Bilanciamento del carico

Sui sistemi SMP è importante che il carico di lavoro sia distribuito equamente tra tutti i processori per sfruttare appieno i vantaggi della multielaborazione. Se ciò non avviene, alcuni processori potrebbero restare inattivi mentre altri verrebbero intensamente sfruttati con una coda di processi in attesa. Il bilanciamento del carico (**load balancing**) tenta di ripartire il carico di lavoro uniformemente tra tutti i processori di un sistema SMP. Bisogna notare che è necessario, di norma, solo nei sistemi in cui ciascun processore detiene una coda privata di processi passibili di esecuzione. Nei sistemi che mantengono una coda comune, il bilanciamento del carico è sovente superfluo: un processore inattivo passerà immediatamente all'esecuzione di un processo dalla coda comune eseguibile dei processi. Va ricordato, tuttavia, che in quasi tutti i sistemi operativi attuali predisposti per la SMP, ogni processore è effettivamente dotato di una propria coda di processi eseguibili.

Il bilanciamento del carico può seguire due approcci: la migrazione guidata (**push migration**) e la migrazione spontanea (**pull migration**). La prima prevede che un processo apposito controlli periodicamente il carico di ogni processore: nell'ipotesi di una sproporzione, riporterà il carico in equilibrio, spostando i processi dal processore saturo ad altri più liberi, o inattivi. La migrazione spontanea, invece, si ha quando un processore inattivo sottrae ad un processore sovraccarico un processo in attesa. I due tipi di migrazione non sono mutuamente esclusivi, e trovano spesso applicazione contemporanea nei sistemi con bilanciamento del carico.

Una singolare proprietà del bilanciamento del carico è di annullare, spesso, il vantaggio derivante dalla predilezione del processore. Il vantaggio di poter eseguire un processo dall'inizio alla fine sul medesimo processore è che in tal modo la memoria cache contiene i dati necessari per quel processo. Con la migrazione dei processi necessaria al bilanciamento del carico, questo vantaggio si perde.

Processori multicore

Tradizionalmente i sistemi SMP hanno reso possibile la concorrenza tra thread con l'utilizzo di diversi processori fisici. Tuttavia, la tendenza recente nel progetto di processori è di inserire più core (unità di calcolo) in un unico chip fisico, dando origine a un processore multicore. Ogni core ha i registri che le servono per conservare informazioni sul suo stato e appare dunque al sistema operativo come un processore fisico separato. I sistemi SMP che usano processori multicore sono più veloci e consumano meno energia dei sistemi in cui ciascun processore è costituito da un singolo chip.

I processori multicore possono complicare i problemi relativi allo scheduling. Le ricerche hanno permesso di scoprire che quando un processore accede alla memoria, una quantità significativa di tempo trascorre in attesa della disponibilità dei dati. Questa situazione, nota come **stallo della memoria** (memory stall), può verificarsi per varie ragioni, come ad esempio la mancanza dei dati richiesti nella cache. Per rimediare a questa situazione, molti dei progetti hardware recenti implementano dei core multithread in cui due o più thread hardware sono assegnati a un singolo core. In questo modo, se un thread è in situazione di stallo in attesa della memoria, il core può passare il controllo a un altro thread.

La figura mostra un processore a due thread nel quale l'esecuzione dei thread 0 e 1 sono interfogliate (**interleaved**). Dal punto di vista del sistema operativo, ogni thread hardware appare come una CPU in grado di eseguire un thread software. In un sistema a due thread con due core il sistema operativo vede dunque quattro CPU.

Virtualizzazione e scheduling

Un sistema dotato di virtualizzazione, anche se a singola CPU, agisce spesso come un sistema multiprocessore. La virtualizzazione software offre una o più CPU virtuali a ogni macchina virtuale in

esecuzione sul sistema, e quindi pianifica l'utilizzo della CPU fisica condivisa dalle macchine virtuali. Le sostanziali differenze tra le varie tecnologie di virtualizzazione rendono difficile sintetizzare gli effetti della virtualizzazione sullo scheduling. In generale, comunque, la maggior parte degli ambienti di virtualizzazione ha un sistema operativo ospitante e diversi sistemi ospiti. Il sistema operativo ospitante crea e gestisce le macchine virtuali, e ogni macchina virtuale ha un sistema operativo ospite installato con applicazioni in esecuzione su di esso. Ogni sistema operativo ospite può essere messo a punto per usi specifici, per particolari applicazioni e utenti e anche per il tempo ripartito (time sharing) o le operazioni real-time.

Ogni algoritmo di scheduling del sistema operativo ospite che fa assunzioni sulla quantità di lavoro effettuabile in un tempo prefissato verrà negativamente influenzato dalla virtualizzazione. Si consideri un sistema operativo a tempo ripartito che prova a suddividere il tempo in intervalli di 100 millisecondi, per offrire agli utenti un tempo di risposta ragionevole. All'interno di una macchina virtuale questo sistema operativo è alla mercé del sistema di virtualizzazione per quanto riguarda il tempo di CPU che gli viene assegnato. Un intervallo di tempo fissato in 100 millisecondi può diventare nella realtà molto più lungo dei 100 millisecondi di tempo di CPU virtuale. A seconda di quanto è occupato il sistema, i 100 millisecondi possono anche diventare un secondo di tempo reale o più, con il risultato che il tempo di risposta per gli utenti che lavorano sulla macchina virtuale sarà insoddisfacente. Gli effetti su un sistema real-time, poi, sarebbero catastrofici.

I risultato finale di questi livelli di scheduling è che i singoli sistemi operativi virtualizzati sfruttano a loro insaputa solo una parte dei cicli di CPU disponibili, mentre effettuano lo scheduling come se sfruttassero tutti i cicli fisicamente disponibili. Tipicamente, l'orologio all'interno di una macchina virtuale fornisce valori sbagliati, perché i contatori impiegano più tempo a scattare di quanto farebbero su una CPU dedicata. La virtualizzazione può quindi vanificare i benefici di un buon algoritmo di scheduling del sistema operativo ospitato sulla macchina virtuale.

Esempi di sistemi operativi

Procediamo ora nella descrizione dei criteri di scheduling per i sistemi operativi Solaris, Windows XP e Linux. Sarà bene tenere presente che, per Solaris e Linux, lo scheduling in questione è relativo ai thread a livello kernel. Si rammenti inoltre che Linux non distingue tra processi e thread; di conseguenza, relativamente allo scheduler di Linux, useremo il termine task.

Scheduling di Solaris

Solaris utilizza uno scheduling dei thread basato sulle priorità in cui ogni thread appartiene ad una delle sei seguenti classi: Tempo ripartito, Interattivo, Real Time, Sistema, Ripartizione equa, Priorità Fissa. All'interno di ciascuna classe vi sono priorità e algoritmi di scheduling differenti.

Ogni classe di scheduling include una scala di priorità. Tuttavia, lo scheduler converte le priorità specifiche della classe in priorità globali e sceglie per l'esecuzione il thread con la priorità globale più elevata. La CPU esegue il thread prescelto finché (1) si blocca, (2) esaurisce la propria frazione di tempo, o (3) è soggetto a prelazione da un thread con priorità più alta. Se vi sono più thread con la stessa priorità, lo scheduler utilizza una coda circolare RR.

Scheduling di Windows XP

Il sistema operativo Windows XP compie lo scheduling dei thread servendosi di un algoritmo basato su priorità e prelazione. Lo scheduler assicura che si eseguano sempre i thread a più alta priorità. La porzione del kernel che si occupa dello scheduling si chiama dispatcher. Una volta selezionato dal dispatcher, un thread viene eseguito finché non sia sottoposto a prelazione da un altro thread a priorità più alta oppure termini, esaurisca il suo quanto di tempo o esegua una chiamata di sistema bloccante, per esempio un'operazione di I/O. Se un thread d'elaborazione in tempo reale, ad alta priorità, entra nella coda dei processi pronti per l'esecuzione mentre è in esecuzione un thread a bassa priorità, quest'ultimo viene

sottoposto a prelazione. Ciò realizza un accesso preferenziale alla CPU per i thread d'elaborazione in tempo reale che ne hanno necessità.

Per determinare l'ordine d'esecuzione dei thread il dispatcher impiega uno schema di priorità a 32 livelli. Le priorità sono suddivise in due classi: la classe variable raccoglie i thread con priorità da 1 a 15, mentre la classe real-time raccoglie i thread con priorità tra 16 e 31 (esiste anche un thread, per la gestione della memoria, che si esegue con priorità 0). Il dispatcher adopera una coda per ciascuna priorità di scheduling e percorre l'insieme delle code da quella a priorità più alta a quella a priorità più bassa, finché trova un thread pronto per l'esecuzione. In assenza di tali thread, il dispatcher fa eseguire un thread speciale detto idle thread.

Scheduling di Linux

Prima della versione 2.5, il kernel di Linux impiegava, per lo scheduling, una variante dell'algoritmo tradizionale di UNIX. Quest'ultimo, però, comporta due problemi: non fornisce strumenti adeguati per i sistemi SMP, e risponde male al crescere dell'ordine di grandezza del numero dei task nel sistema. A partire dalla versione 2.5, il kernel offre un algoritmo di scheduling che gira in tempo costante a prescindere dal numero di task nel sistema. Il nuovo scheduler è anche migliorato nei confronti della SMP: permette infatti la gestione della predilezione per il processore, rende possibile il bilanciamento del carico, e offre inoltre strumenti volti a garantire equità ai task interattivi.

Lo scheduler di Linux ricorre a un algoritmo di scheduling con prelazione, basato sulle priorità, con due gamme di priorità separate: un intervallo real-time che va da 0 a 99 e un intervallo detto nice compreso tra 100 e 140. Queste due gamme sono poi tradotte all'interno di una scala globale di priorità, in cui i valori numericamente più bassi rappresentano priorità più alte.

A differenza degli scheduler di molti altri sistemi, Linux assegna ai task con priorità più alta porzioni di tempo più cospicue e a quelle dalla priorità bassa quanti di tempo più brevi.

Un task pronto per l'esecuzione è considerato eseguibile dalla CPU se non ha ancora consumato per intero il proprio quanto di tempo. Allorché esaurisce tale quanto, il task è considerato scaduto, e non può più essere posto in esecuzione finché tutti gli altri task non abbiano consumato la propria porzione di tempo. Il kernel elenca i task pronti per l'esecuzione in una struttura dati detta runqueue (coda di esecuzione). Per garantire la possibilità di SMP, ciascun processore mantiene la propria coda di esecuzione e opera una pianificazione autonoma.

Linux implementa lo scheduling real-time come definito dallo standard POSIX. I task real-time ricevono priorità statiche, mentre gli altri task hanno priorità dinamiche, basate sul loro valore nice, cui si aggiunge o si sottrae il valore 5. Il grado di interattività di un task determina se il valore 5 sarà sommato al valore nice o sottratto da esso. Tale grado di interattività è determinato dal tempo trascorso dal task in attesa prima che il suo I/O sia disponibile. I task maggiormente interattivi hanno, in genere, tempi di attesa più lunghi; poiché lo scheduler dà loro preferenza, la probabilità che essi ottengano un bonus vicino a -5 è alta. Questi adattamenti hanno l'effetto di attribuire priorità più alte per i task interattivi. Al contrario, i task con tempi di attesa più brevi sono spesso a prevalenza di elaborazione, e dunque vedranno diminuire la loro priorità.

La priorità dinamica di un task è ricalcolata nel momento in cui abbia esaurito la propria porzione di tempo e debba passare dall'array attivo a quello scaduto. Così, quando i due array si scambiano i ruoli, tutti i task del nuovo array attivo avranno ricevuto nuove priorità, con i relativi quanti di tempo.

Valutazione degli algoritmi

Ci si può chiedere come scegliere un algoritmo di scheduling della CPU per un sistema particolare. Come abbiamo visto ci sono molti algoritmi di scheduling, ciascuno dotato dei propri parametri; quindi, la scelta di un algoritmo può essere abbastanza difficile.

Il primo problema da affrontare riguarda la definizione dei criteri da usare per la scelta dell'algoritmo. Una volta definiti i criteri di selezione, è necessario valutare gli algoritmi considerati. Di seguito si descrivono i vari metodi di valutazione.

Modelli deterministicici

Fra i metodi di valutazione sono di grande importanza quelli che rientrano nella classe della valutazione analitica. La **valutazione analitica**, secondo l'algoritmo dato e il carico di lavoro del sistema, fornisce una formula o un numero che valuta le prestazioni dell'algoritmo per quel carico di lavoro.

La definizione e lo studio di un modello deterministico è un tipo di valutazione analitica, che considera un carico di lavoro predeterminato e definisce le prestazioni di ciascun algoritmo per quel carico di lavoro.

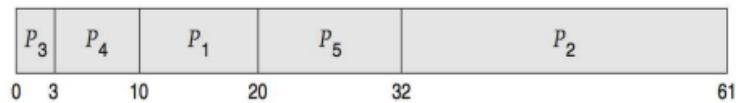
Si supponga, per esempio, di avere il carico di lavoro illustrato di seguito; i cinque processi si presentano al tempo 0, nell'ordine dato, e la durata delle sequenze di operazioni della CPU è espressa in millisecondi:

Process	Burst Time
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

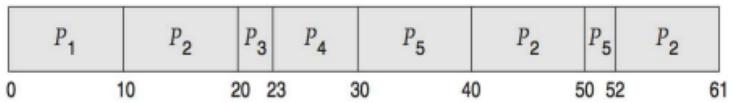
FCS è 28ms:



Non-preemptive SJF è 13ms:



RR è 23ms:



Si può stabilire con quale fra gli algoritmi di scheduling FCFS, SJF e RR (quanto di tempo = 10 millisecondi) per questo insieme di processi si ottenga il minimo tempo medio d'attesa

È importante notare come, in questo caso, il criterio SJF fornisca come risultato un tempo medio d'attesa minore della metà del tempo corrispondente ottenuto con lo scheduling FCFS; l'algoritmo RR fornisce un risultato intermedio tra i precedenti.

La definizione e lo studio di un modello deterministico è semplice e rapida; i risultati sono numeri esatti che consentono il confronto tra gli algoritmi. Nondimeno, anche i parametri devono essere numeri esatti e i risultati sono applicabili solo a quei casi. Il suo impiego principale consiste nella descrizione degli algoritmi di scheduling e nella presentazione d'esempi. Nei casi in cui si possono eseguire ripetutamente gli stessi programmi e si possono misurare con precisione i requisiti d'elaborazione dei programmi, i modelli deterministicici sono utilizzabili per scegliere un algoritmo di scheduling.

Modelli di code e formula di Little

In molti sistemi i processi eseguiti variano di giorno in giorno, quindi non esiste un insieme statico di processi (e di tempi) da usare nei modelli deterministicici. Si possono però determinare le distribuzioni di CPU burst e I/O burst, poiché si possono misurare e quindi approssimare, o più semplicemente stimare. Si ottiene una formula matematica che indica la probabilità di una determinata sequenza di operazioni della CPU. Comunemente questa distribuzione è di tipo esponenziale ed è descritta dalla sua media.

Analogamente, è necessario fornire anche la distribuzione degli istanti d'arrivo dei processi nel sistema. Da queste due distribuzioni si può calcolare la produttività media, l'utilizzo o il tempo d'attesa medi, e così via, per la maggior parte degli algoritmi.

Il sistema di calcolo si descrive come una rete di server, ciascuno con una coda d'attesa. La CPU è un server con la propria coda dei processi pronti, e il sistema di I/O ha le sue code dei dispositivi. Se sono noti l'andamento degli arrivi e dei servizi, si possono calcolare l'utilizzo, la lunghezza media delle code, il tempo medio d'attesa e così via.

Questo tipo di studio si chiama **analisi delle reti di code** (*queueing-network analysis*).

Si consideri il seguente esempio:

Sia n la lunghezza media di una coda, escluso il processo correntemente servito, detti W il tempo medio di attesa nella coda e λ l'andamento medio d'arrivo dei nuovi processi nella coda; si prevede che, nel tempo W durante il quale un processo attende nella coda, raggiungano la coda λW nuovi processi. Se il sistema è stabile, il numero dei processi che lasciano la coda deve essere uguale al numero dei processi che vi arrivano; quindi,

$$\text{Formula di Little: } n = \lambda W$$

Tale formula è utile soprattutto perché è valida per qualsiasi algoritmo di scheduling e distribuzione d'arrivi.

La formula di Little è utilizzabile per il calcolo di una delle tre variabili, quando sono note le altre due. Per esempio, sapendo che ogni secondo arrivano 7 processi (in media), e che normalmente nella coda ne sono presenti 14, si può calcolare che il tempo medio d'attesa per ogni processo è di 2 secondi.

L'analisi delle reti di code può essere utile per il confronto degli algoritmi di scheduling, ma presenta alcuni limiti. Attualmente le classi di algoritmi e distribuzioni trattabili sono piuttosto limitate. Inoltre, poiché può essere difficile lavorare con la matematica di distribuzioni e algoritmi complicati, spesso, affinché siano trattabili matematicamente, si definiscono in modo irrealistico le distribuzioni d'arrivo e servizio.

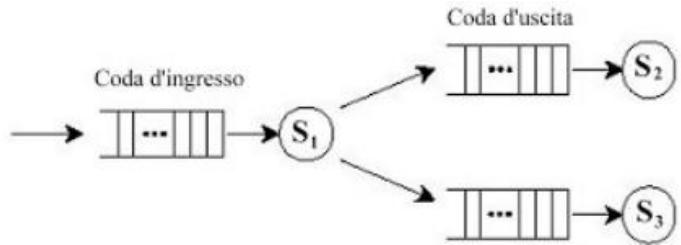
Generalmente è necessario stabilire anche un numero di presupposti indipendenti che possono non essere precisi. Per poter calcolare una risposta, le reti di code spesso si limitano ad approssimare un sistema reale, rendendo discutibile la precisione dei risultati ottenuti.

Simulazioni

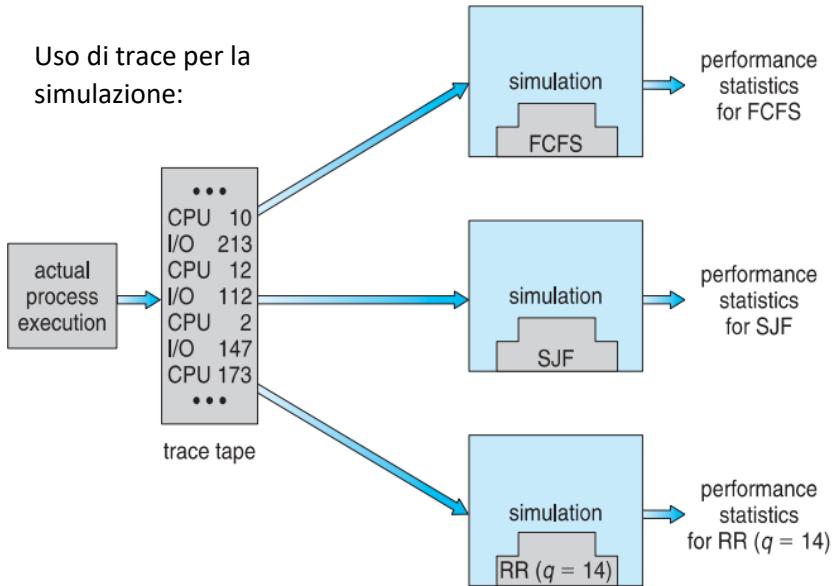
Per riuscire ad avere una valutazione più precisa degli algoritmi di scheduling ci si può servire di simulazioni. Le simulazioni implicano la programmazione di un modello del sistema di calcolo; le strutture dati rappresentano gli elementi principali del sistema. Il simulatore dispone di una variabile che rappresenta un clock; con l'aumentare del valore di questa variabile, il simulatore modifica lo stato del sistema in modo da descrivere le attività dei dispositivi, dei processi e dello scheduler. Durante l'esecuzione della simulazione si raccolgono e si stampano statistiche che indicano le prestazioni degli algoritmi.

I dati necessari per condurre la simulazione si possono ottenere in vari modi. Il metodo più diffuso impiega un generatore di numeri casuali, programmato per creazione di processi, durata delle sequenze di operazioni della CPU, arrivi, conclusioni e così via; in modo conforme alle rispettive distribuzioni di probabilità, definibili matematicamente oppure in modo empirico. Se la distribuzione deve essere definita in modo empirico, si fanno misure sul sistema reale in esame, e si usano i risultati per definire la distribuzione effettiva degli eventi nel sistema reale.

Tuttavia, una simulazione condotta secondo la distribuzione può non essere precisa, a causa delle relazioni esistenti tra eventi successivi nel sistema reale. La distribuzione relativa alle frequenze, infatti, si limita a indicare quanti eventi di una data categoria si verificano, senza fornire informazioni sul loro ordine. Per rimediare a questo problema si può sottoporre il sistema reale a un controllo continuo, con la registrazione della sequenza degli eventi effettivi; in questo modo si ottiene un cosiddetto **trace tape** (vedi figura successiva), che poi si usa per condurre la simulazione. Si tratta di uno strumento eccellente che permette



di confrontare gli algoritmi rispetto allo stesso insieme di dati reali, e con cui si possono ottenere risultati molto precisi.



Poiché spesso richiedono diverse ore del tempo d'elaborazione, le simulazioni possono tuttavia essere molto onerose. Una simulazione dettagliata dà risultati molto precisi, ma richiede anche una gran quantità di tempo, e molto spazio di memoria per la registrazione degli eventi. Inoltre, la progettazione, la codifica e la messa a punto di un simulatore possono essere un compito assai impegnativo

Implementazione

Persino una simulazione ha dei limiti per quel che riguarda la precisione. L'unico modo assolutamente sicuro per valutare un algoritmo di scheduling consiste nel codificarlo, inserirlo nel sistema operativo e osservarne il comportamento nelle reali condizioni di funzionamento del sistema.

Il problema principale di questo metodo è il suo costo: le spese non sono dovute solo alla codifica dell'algoritmo e alle modifiche da fare al sistema operativo affinché possa gestire l'algoritmo con le sue strutture dati, ma anche alle reazioni degli utenti a fronte di costanti modifiche del sistema operativo. Agli utenti non interessa la realizzazione di un migliore sistema operativo ma solo di eseguire i processi; dunque, un sistema operativo che si trovi sempre in fase di modifica e messa a punto non aiuta gli utenti a svolgere il loro lavoro.

Gli algoritmi di scheduling più flessibili sono quelli che possono essere tarati dagli amministratori del sistema o dai suoi utenti in modo da adattarsi a una specifica gamma di applicazioni. Per esempio, le macchine impegnate in applicazioni grafiche all'avanguardia avranno necessità di scheduling diverse da quelle di un server web o di un file server. Tali scheduler si dicono che possono essere modificati per-sito o per-sistema.

Un altro approccio è di usare delle API appropriate per modificare la priorità di processi e thread.

6. Sincronizzazione dei Processi

Introduzione

Nel [Capitolo 3](#) è stato descritto un modello di sistema costituito da un certo numero di processi sequenziali cooperanti o thread, tutti in esecuzione asincrona e con la possibilità di condividere dati. Tale modello è illustrato attraverso l'esempio del produttore/consumatore, che ben rappresenta molte situazioni che riguardano i sistemi operativi. In particolare, si è descritto come un buffer limitato sia utilizzabile per permettere ai processi la condivisione della memoria.

Torniamo al concetto di buffer limitato. Come è stato sottolineato, la nostra soluzione consente la presenza contemporanea di non più di DIM_BUFFER – 1 elementi. Si supponga di voler modificare l'algoritmo per rimediare a questa carenza. Una possibilità consiste nell'aggiungere una variabile intera inizializzata a 0 (counter), che si incrementa ogni volta s'inserisce un nuovo elemento nel buffer e si decrementa ogni volta si preleva un elemento dal buffer. Il codice per il processo **produttore** si può modificare come segue:

```
1 -while (true) {
2     // produce un elemento in next_produced
3     while (counter == BUFFER_SIZE);
4     // non fare nulla
5     buffer[in] = next_produced;
6     in = (in + 1) % BUFFER_SIZE;
7     counter++;
8 }
```

Il codice per il processo **consumatore** si può modificare come segue:

```
1 -while (true) {
2     while (counter == 0);
3     // non fare nulla
4     next_consumed = buffer[out]
5     out = (out + 1) % BUFFER_SIZE;
6     counter--;
7     // consuma un elemento in next_consumed
8 }
```

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 counter 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 counter e potrebbe essere 4, 5 o 6! Ovviamente il solo risultato corretto è counter == 5, che si ottiene se si eseguono separatamente il produttore e il consumatore.

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

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

Analogamente, l'istruzione counter-- si può codificare come

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

dove register1 e register2 sono registri locali della CPU (possono essere lo stesso registro fisico).

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

Una di queste sequenze è

S0: producer register1 = counter	{register1 = 5}
S1: producer register1 = register1 + 1	{register1 = 6}
S2: consumer register2 = counter	{register2 = 5}
S3: consumer register2 = register2 - 1	{register2 = 4}
S4: producer counter = register1	{counter = 6 }
S5: consumer counter = register2	{counter = 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 S4 e S5 si giungerebbe allo stato errato in cui counter == 6.

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 condizione richiede una forma di sincronizzazione dei processi.

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**, in cui il processo può modificare variabili comuni, aggiornare una tabella, scrivere in un file e così via. Quando un processo è in esecuzione nella propria sezione critica, non si deve consentire a nessun altro processo di essere in esecuzione nella propria sezione critica. Quindi, l'esecuzione delle sezioni critiche da parte dei processi è mutuamente esclusiva nel tempo. Il problema della sezione critica si affronta progettando 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 **entry section** (sezione d'ingresso). La sezione critica può essere seguita da una **exit section** (sezione d'uscita), e la restante parte del codice è detta **remainder section** (sezione non critica).

Il codice di destra mostra la struttura generale di un tipico processo P_i . La entry section e l'exit section sono state inserite nei riquadri per evidenziare questi importanti segmenti di codice.

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

1. **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.
2. **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 remainder section 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.
3. **Bounded waiting** (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.

```

do {
    entry section
    critical section
    exit section
    remainder section
} while (true);

```

Si suppone che ogni processo sia eseguito a una velocità diversa da zero. Tuttavia, non si può fare alcuna ipotesi sulla **velocità relativa** degli n processi.

Le due strategie principali per la gestione delle sezioni critiche nei sistemi operativi prevedono l'impiego di:

1. **Kernel preemptive.** Permette che un processo funzionante in modalità di sistema sia sottoposto a prelazione, rinviandone in tal modo l'esecuzione. I kernel con diritto di prelazione rivelano particolari difficoltà di progettazione quando sono destinati ad architetture SMP, poiché in tali ambienti due processi nella modalità di sistema possono essere eseguiti in contemporanea su processori differenti.
2. **Kernel nonpreemptive.** Non consente di applicare la prelazione a un processo attivo in modalità di sistema: l'esecuzione di questo processo seguirà finché lo stesso esca da tale modalità, si blocchi o ceda volontariamente il controllo della CPU. In sostanza, i kernel senza diritto di prelazione sono immuni dai problemi legati all'ordine degli accessi alle strutture dati del kernel, visto che un solo processo per volta impegnava il kernel.

I kernel preemptive dovrebbero essere preferiti a quelli nonpreemptive poiché i primi sono più adatti alla programmazione real time, dal momento che permettono ai processi in tempo reale di far valere il loro diritto di precedenza nei confronti di un processo attivo nel kernel. Inoltre, i kernel preemptive possono vantare una maggiore prontezza nelle risposte, data la loro scarsa propensione a eseguire i processi in modalità di sistema per un tempo eccessivamente lungo, prima di liberare la CPU per i processi in attesa.

Soluzione di Peterson

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 la soluzione di Peterson funzioni correttamente su tali sistemi. Tuttavia, tale soluzione rappresenta un buon algoritmo per il problema della sezione critica che illustra alcune insidie legate alla progettazione di programmi che soddisfino i tre requisiti di mutua esclusione, progresso e attesa limitata.

La soluzione di Peterson si applica a due processi, P_0 e P_1 ognuno dei quali esegue alternativamente la propria sezione critica e la sezione rimanente. Indicheremo con P_i uno dei due processi e con P_j l'altro.

La soluzione di Peterson richiede che i processi condividano i seguenti dati:

- **int turn;** segnala di chi sia il turno d'accesso alla sezione critica; quindi, se $turn = i$, il processo P_i è autorizzato a eseguire la propria sezione critica.
- **boolean flag[2];** array che indica se un processo sia pronto a entrare nella propria sezione critica. Per esempio, se $flag[i] == \text{true}$, P_i lo è.

Analizziamo ora il seguente algoritmo:

```
- do {  
    flag[i] = true;  
    turn = j;  
    while (flag[j] && turn == j);  
    /*  
     * critical section  
    */  
    flag[i] = false;    exit section  
    /*  
     * remainder section  
    */  
} while (true);
```

entry section

P_0 : $\text{flag}[0]=\text{true}$
 P_1 : $\text{flag}[1]=\text{true}$
 P_1 : $\text{turn}=0$
 P_1 : $\text{while}(\text{flag}[0] \&\& \text{turn}=0)$;
 P_0 : $\text{turn}=1$
 P_0 : $\text{while}(\text{flag}[1] \&\& \text{turn}=1)$;
...

Per accedere alla sezione critica, il processo P_i assegna innanzitutto a $flag[i]$ il valore *true*; quindi attribuisce a *turn* il valore *j*, conferendo così all'altro processo la facoltà di entrare nella sezione critica. Qualora entrambi i processi tentino l'accesso contemporaneo, all'incirca nello stesso momento sarà assegnato a *turn* sia il valore *i* che il valore *j*. Soltanto uno dei due permane: l'altro sarà immediatamente

sovrascritto. Il valore definitivo di *turn* stabilisce quale dei due processi sia autorizzato a entrare per primo nella propria sezione critica. Informalmente, il primo che dice "entri lei" (la condizione del while) sarà il primo ad entrare poiché l'altro ricambia la gentilezza.

Dimostriamo ora la correttezza di questa soluzione provando le 3 proprietà che devono essere garantite per una soluzione al problema della sezione critica:

1. **Mutua esclusione:** Ogni P_i accede alla propria sezione critica solo se $flag[j] == false$ oppure $turn == i$. Si noti anche che, se entrambi i processi sono eseguibili in concomitanza nelle rispettive sezioni critiche allora entrambi i flag sono posti a true ma *turn* può valere 0 o 1 oppure 0 ma non entrambi. Pertanto, uno dei processi (poniamo P_i) deve aver eseguito con successo l'istruzione while, mentre P_j aveva da eseguire almeno un'istruzione aggiuntiva ($turn == j$). Tuttavia, poiché da quel momento, e fino al termine della permanenza di P_j nella propria sezione critica, restano valide le asserzioni $flag[j] == true$ e $turn == j$, ne consegue che la mutua esclusione è preservata.
2. **Progresso**
3. **Bounded waiting**

Per dimostrare le ultime due proprietà, osserviamo come l'ingresso di un processo P_i nella propria sezione critica possa essere impedito solo se il processo è bloccato nella sua iterazione while, con le condizioni $flag[j] == true$ e $turn == j$; questa è l'unica possibilità. Qualora P_j non sia pronto a entrare nella sezione critica ($flag[j] == false$) e P_i può accedere alla propria sezione critica. Se P_j ha impostato $flag[j]$ a true e sta eseguendo il proprio ciclo while, $turn == i$, oppure $turn == j$. Se $turn == i$, P_i entrerà nella propria sezione critica. Se $turn == j$, P_j entrerà nella propria sezione critica. Tuttavia, al momento di uscire dalla propria sezione critica, P_j reimposta $flag[j]$ a false, consentendo a P_i di entrarvi. Se P_i imposta $flag[j]$ a true, deve anche attribuire alla variabile *turn* il valore i . Poiché tuttavia P_i non modifica il valore della variabile *turn* durante l'esecuzione dell'istruzione while, P_i entrerà nella sezione critica (**progresso**) dopo che P_j abbia effettuato non più di un ingresso (**bounded waiting**).

Hardware per la sincronizzazione

Molti sistemi forniscono supporto hardware per implementare il codice della sezione critica. In generale, si può affermare che qualunque soluzione al problema richiede l'uso di un semplice strumento detto **lock**. Il corretto ordine degli accessi alle strutture dati del kernel è garantito dal fatto che le sezioni critiche sono protette da lock. In altri termini, per accedere alla propria sezione critica un processo deve acquisire il possesso di un lock, che restituirà al momento della sua uscita. Si veda la figura a destra.

```
do {
    acquire lock
    critical section
    release lock
    remainder section
} while (TRUE);
```

Iniziamo presentando alcune semplici istruzioni hardware disponibili in molti sistemi e mostrando come queste possano essere efficacemente utilizzate per risolvere il problema della sezione critica. Le funzionalità hardware possono rendere più facile il compito del programmatore e migliorare l'efficienza del sistema.

In un sistema dotato di una singola CPU tale problema si potrebbe risolvere semplicemente se si potessero interdire le interruzioni mentre si modificano le variabili condivise. In questo modo si assicurerebbe un'esecuzione ordinata e senza possibilità di prelazione della corrente sequenza di istruzioni; non si potrebbe eseguire nessun'altra istruzione; quindi, non si potrebbe apportare alcuna modifica inaspettata alle variabili condivise. E questo l'approccio seguito dai kernel senza diritto di prelazione. Sfortunatamente questa soluzione non è sempre praticabile; la disabilitazione delle interruzioni nei sistemi multiprocessore può comportare sprechi di tempo dovuti alla necessità di trasmettere la richiesta di disabilitazione delle interruzioni a tutte le unità d'elaborazione. Tale trasmissione ritarda l'accesso a ogni sezione critica

determinando una diminuzione dell'efficienza. Si considerino, inoltre, gli effetti su un orologio di sistema aggiornato tramite le interruzioni.

Per questo motivo 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. Anziché discutere una specifica istruzione di una particolare architettura, è preferibile astrarre i concetti principali che stanno alla base di queste istruzioni.

L'istruzione TestAndSet() si può definire come segue:

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

Questa istruzione è eseguita **atomicamente**, cioè come un'unità non soggetta a interruzioni; quindi, se si eseguono contemporaneamente due istruzioni TestAndSet(), ciascuna in un'unità d'elaborazione diversa, queste vengono eseguite in modo sequenziale in un ordine arbitrario. Se si dispone dell'istruzione TestAndSet(), si può realizzare la mutua esclusione dichiarando una variabile booleana globale *lock*, inizializzata a false. La struttura del processo P_i è illustrata come segue:

```
- do {  
    while (TestAndSet(&lock));  
    /*  
     *      sezione critica  
     */  
    lock = false;  
    /*  
     *      remainder section  
     */  
} while (true);
```

L'istruzione CompareAndSwap(), definita dal codice seguente, restituisce il valore originale del parametro in *value* e setta la variabile *value* al valore *newValue* ma solo se *value == expected*; cioè, lo swap avviene solo con questa condizione. Come l'istruzione TestAndSet(), è anch'essa eseguita atomicamente.

```
- int CompareAndSwap(int *value, int expected, int newValue) {  
    int temp = *value;  
    if (*value == expected)  
        *value = newValue;  
    return temp;  
}
```

Se si dispone di un intero condiviso *lock* inizializzato a 0 la soluzione usando CompareAndSwap() è:

```
- do {  
    while (CompareAndSwap(&lock, 0, 1) != 0);  
    /*  
     *      critical section  
     */  
    lock = 0;  
    /*  
     *      remainder section  
     */  
} while (true);
```

Se *lock == 0* restituisce 0 e setta 1, esce dal loop.
Se *lock == 1* restituisce 1 e rimane 1, rimane nel loop.

Questi algoritmi soddisfano il requisito della mutua esclusione, ma non quello dell'attesa limitata. Inoltre, le soluzioni precedenti sono complicate e generalmente inaccessibili ai programmatore di applicazioni.

I progettisti di SO forniscono strumenti software per risolvere i problemi della sezione critica ed il più semplice è il **mutex lock** (*mutual exclusion*). Un mutex protegge una sezione critica prendendo `acquire()` un lock e poi rilasciandolo `release()` (tali chiamate sono solitamente implementate con istruzioni hardware atomiche).

Definiamo le chiamate `acquire()` e `release()`; si noti che *available* è una variabile booleana che indica se il lock è disponibile o no:

```
- acquire() {  
    while (!available);  
    ... // busy wait  
    available = false;  
}  
  
- release() {  
    available = true  
}
```

La struttura del processo con mutex è illustrata come segue:

```
- do {  
    acquire lock  
    /*  
     * critical section  
     */  
    release lock  
    /*  
     * remainder section  
     */  
} while (true);
```

Semafori

Un **semaforo** S è una variabile intera cui si può accedere, escludendo l'inizializzazione, solo tramite due operazioni atomiche predefinite: `wait()` e `signal()`. Le definizioni classiche di `wait()` e `signal()` in pseudocodice sono le seguenti:

```
- wait(S) {  
    while (S <= 0);  
    ... //non-op  
    S--  
}  
  
- 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. Inoltre, nel caso della `wait(S)` si devono eseguire senza interruzione anche la verifica del valore intero di S ($S \leq 0$) e la sua possibile modifica ($S--$).

Si noti che i semafori hanno un valore numerico illimitato (anche detti semafori contatore) e quindi sono una versione più sofisticata dei mutex lock il cui valore è 0 o 1 (prendono anche il nome di semafori binari).

Uso dei semafori

I semafori contatore trovano applicazione nel controllo dell'accesso a una data risorsa presente in un numero finito di esemplari. Il semaforo è inizialmente impostato al numero di esemplari disponibili. I processi che desiderino utilizzare un esemplare della risorsa invocano `wait()` sul semaforo, decrementandone così il valore; i processi che restituiscono un esemplare della risorsa, invece, invocano `signal()` sul semaforo, incrementandone il valore. Quando il semaforo vale 0, vengono allocati tutti gli

esemplari della risorsa, e i processi che ne richiedano l'uso dovranno bloccarsi fino a che il semaforo non ritorni positivo.

I semafori sono utilizzabili anche per risolvere diversi problemi di sincronizzazione. Si considerino, per esempio, due processi in esecuzione concorrente: P_1 con un'istruzione S_1 e P_2 con un'istruzione S_2 . Si supponga di voler eseguire S_2 solo dopo che S_1 è terminata. Questo schema si può prontamente realizzare facendo condividere a P_1 e P_2 un semaforo comune, $synch$, inizializzato a 0, e inserendo nel processo P_1 le istruzioni S_1 seguito da $\text{signal}(synch)$ e nel processo P_2 l'istruzione $\text{wait}(synch)$ prima di S_2 .

Poiché $synch$ è inizializzato a 0 è evidente che P_2 esegue P_1 solo dopo che quest'ultimo ha eseguito $\text{signal}(synch)$, che si trova dopo S_1 .

Realizzazione

Il principale svantaggio della definizione di semaforo (dunque anche del mutex) è che richiede una condizione di attesa attiva (**busy waiting**). Mentre un processo si trova nella propria sezione critica, qualsiasi altro processo che tenti di entrarvi si trova sempre nel ciclo del codice della sezione d'ingresso. Chiaramente questa soluzione costituisce un problema per un sistema con multiprogrammazione, poiché la condizione d'attesa attiva spreca cicli della CPU che un altro processo potrebbe sfruttare in modo produttivo. Questo tipo di semaforo è anche detto **spinlock**, perché i processi "girano" (spiri) mentre attendono al semaforo (i semafori spinlock hanno però il vantaggio di non richiedere cambio di contesto nel caso in cui un processo sia fermo in attesa). Tali cambi di contesto possono essere piuttosto costosi, in termini di tempo. Ne consegue che i semafori spinlock sono utili quando i lock sono applicati per brevi intervalli di tempo: infatti, trovano frequente applicazione nei sistemi multiprocessore, dove un processo gira su un processore mentre un altro thread esegue la propria sezione critica su un altro processore).

Per superare la necessità dell'attesa attiva, si possono modificare le definizioni delle operazioni $\text{wait}()$ e $\text{signal}()$: quando un processo invoca l'operazione $\text{wait}()$ 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 cambia lo stato del processo nello stato d'attesa. 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 $\text{signal}()$ su S da parte di qualche altro processo. Il processo si riavvia tramite un'operazione $\text{wakeup}()$ che modifica lo stato del processo da attesa a pronto. Il processo entra nella coda ready.

Per realizzare i semafori secondo quel che s'è detto si può definire il semaforo come una struttura del linguaggio C:

```
-typedef struct {
    int valore;
    struct processo *lista;
} semaforo;
```

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

L'operazione $\text{wait}()$ del semaforo si può definire come segue:

```
-wait(semaforo *S) {
    S->valore--;
    if (S->valore < 0) {
        aggiungi questo processo a S->lista;
        block();
    }
}
```

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

```
- signal(semaforo *S) {
    S->valore++;
    if (S->valore <= 0) {
        togli un processo P da S->lista;
        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 di base.

Occorre notare che, mentre la definizione classica di semaforo ad attesa attiva è tale che il valore del semaforo non è mai negativo, tale definizione può condurre a valori negativi. Se il valore del semaforo è negativo, la sua dimensione è data dal numero dei processi che attendono a quel semaforo. Ciò avviene a causa dell'inversione dell'ordine del decremento e della verifica nel codice dell'operazione wait().

La lista dei processi che attendono a un semaforo si può facilmente realizzare inserendo un campo puntatore in ciascun PCB (process control block). Ogni semaforo contiene un valore intero e un puntatore a una lista di PCB. Per aggiungere e togliere processi dalla lista assicurando un'attesa limitata si può usare una coda FIFO, della quale il semaforo contiene i puntatori al primo e all'ultimo elemento. In generale si può usare qualsiasi criterio d'accodamento; il corretto uso dei semafori non dipende dal particolare criterio adottato.

I semafori devono essere eseguiti in modo atomico. Si deve garantire che nessuno dei due processi possa eseguire operazioni wait() e signal() contemporaneamente sullo stesso semaforo. Si tratta di un problema di accesso alla sezione critica, e in un contesto monoprocesso lo si può risolvere semplicemente inibendo le interruzioni durante l'esecuzione di wait() e signal(). Nei sistemi con una sola CPU, infatti, le interruzioni sono i soli elementi di disturbo: non vi sono istruzioni eseguite da altri processori. Finché non si riattivino le interruzioni, dando la possibilità allo scheduler di riprendere il controllo della CPU, il processo corrente continua indisturbato la sua esecuzione.

Nei sistemi multiprocessore è necessario disabilitare le interruzioni di tutti i processori, perché altrimenti le istruzioni dei diversi processi in esecuzione su processori distinti potrebbero interferire fra loro. Tuttavia, disabilitare le interruzioni di tutti i processori può non essere cosa semplice, e causare un notevole calo delle prestazioni. E per questo che (per garantire l'esecuzione atomica di wait() e signal()) i sistemi SMP devono mettere a disposizione altre tecniche di realizzazione dei lock (per esempio, gli spinlock).

È importante rilevare che questa definizione delle operazioni wait() e signal() non consente di eliminare completamente l'attesa attiva, ma piuttosto di rimuoverla dalle sezioni d'ingresso dei programmi applicativi. Inoltre, l'attesa attiva si limita alle sezioni critiche delle operazioni wait() e signal(), che sono abbastanza brevi; se sono convenientemente codificate, non sono più lunghe di 10 istruzioni. Quindi, la sezione critica non è quasi mai occupata e l'attesa attiva si presenta raramente e per breve tempo. Una situazione completamente diversa si verifica con i programmi applicativi le cui sezioni critiche possono essere lunghe minuti o anche ore, oppure occupate spesso. In questi casi l'attesa attiva è assai inefficiente.

Deadlock e Starvation

La realizzazione di un semaforo con coda d'attesa può condurre a situazioni in cui ciascun processo di un insieme di processi attende indefinitamente un evento (l'esecuzione di un'operazione signal()) che può essere causato solo da uno dei processi dello stesso insieme. Quando si verifica una situazione di questo tipo si dice che i processi sono in **deadlock** (stallo).

Per illustrare questo fenomeno si consideri un insieme di due processi, P_0 e P_1 , 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>	Si supponga che P_0 esegua $\text{wait}(S)$ e quindi P_1 esegua
<code>wait(Q);</code>	<code>wait(S);</code>	$\text{wait}(Q)$; eseguita $\text{wait}(Q)$, P_0 deve attendere che P_1
<code>...</code>	<code>...</code>	esegua $\text{signal}(Q)$; analogamente, quando P_1 esegue
<code>signal(S);</code>	<code>signal(Q);</code>	$\text{wait}(S)$, deve attendere che P_0 esegua $\text{signal}(S)$.
<code>signal(Q);</code>	<code>signal(S);</code>	Poiché queste operazioni $\text{signal}()$ non si possono
		eseguire, i due processi sono in deadlock.

Plausibile domanda di esame: Come si potrebbe risolvere la situazione?

Qui si potrebbe avere un $\text{signal}(S)$ dopo $\text{wait}(Q)$ in P_1 e $\text{signal}(Q)$ dopo $\text{wait}(S)$ in P_0 .

Un insieme di processi è in deadlock se **ciascun processo** dell'insieme attende un evento che può essere causato solo da un altro processo dell'insieme. In questo contesto si considerano principalmente gli eventi di acquisizione e rilascio di risorse; tuttavia, anche altri tipi di eventi possono produrre situazioni di deadlock.

Un'altra questione connessa alle situazioni di deadlock è quella dello **starvation** (attesa indefinita).

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.

Inversione di priorità

Nello scheduling dei processi si possono incontrare difficoltà ognqualvolta un processo a priorità più alta abbia bisogno di leggere o modificare dati a livello kernel utilizzati da un processo, o da una catena di processi, a priorità più bassa. Visto che i dati a livello kernel sono tipicamente protetti da un lock, il processo a priorità maggiore dovrà attendere finché il processo a priorità minore non avrà finito di utilizzare le risorse. La situazione si complica ulteriormente se il processo a priorità più bassa ha dovere di prelazione su un processo a priorità più alta. Assumiamo, ad esempio, che vi siano tre processi L , M e H , le cui priorità seguono l'ordine $L < M < H$. Assumiamo che il processo H richieda la risorsa R alla quale sta accedendo il processo L . Usualmente il processo H resterebbe in attesa che L liberi la risorsa R .

Supponiamo però che M diventi eseguibile, con prelazione sul processo L . Avviene indirettamente che un processo con priorità più bassa, il processo M , influenzi il tempo che H attenderà in attesa della risorsa R .

Questo problema è noto come inversione della priorità. Dato che **l'inversione di priorità** si verifica solo su sistemi con più di due priorità, una delle soluzioni è limitare a due il numero di priorità. Tuttavia, questa soluzione non è accettabile nella maggior parte dei sistemi a uso generale. Solitamente questi sistemi risolvono il problema implementando un **protocollo di ereditarietà delle priorità**, secondo il quale tutti i processi che stanno accedendo a risorse di cui hanno bisogno processi con priorità maggiore ereditano la priorità più alta finché non finiscono di utilizzare le risorse in questione. Quando hanno terminato, la loro priorità ritorna al valore originale. Nell'esempio discusso in precedenza, un protocollo di ereditarietà delle priorità avrebbe permesso al processo L di ereditare temporaneamente la priorità di H , impedendo così al processo M di prelazionare la sua esecuzione. In un tale caso, una volta che il processo H avrà terminato con la risorsa R , rinuncerà alla priorità ereditata da H assumendo di nuovo la priorità originale. Poiché R sarà a questo punto disponibile, il processo H , e non il processo M , sarà il successivo processo eseguito.

Problemi tipici di sincronizzazione

In questo paragrafo s'illustrano diversi problemi di sincronizzazione come esempi di una vasta classe di problemi connessi al controllo della concorrenza. Questi problemi sono utili per verificare quasi tutte le nuove proposte di schemi di sincronizzazione. Nelle soluzioni che proponiamo ai problemi s'impiegano i semafori.

Produttori e consumatori con bounded-buffer

Il problema dei produttori e consumatori con memoria limitata si usa generalmente per illustrare la potenza delle primitive di sincronizzazione. In questa sede si presenta uno schema generale di soluzione, senza far riferimento a nessuna realizzazione particolare.

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 *empty* e *full* conteggiano rispettivamente il numero di posizioni vuote e il numero di posizioni piene nel buffer. Il semaforo *empty* si inizializza al valore n , mentre il semaforo *full* si inizializza al valore 0.

Riportiamo di seguito la struttura generale del processo produttore e del processo consumatore:

```
-do { // PRODUTTORE
    ...
    // produce un elemento in next_produced
    ...
    wait(empty);
    wait(mutex);
    ...
    // inserisci in buffer l'elemento in next_produced
    ...
    signal(mutex);
    signal(full);
} while (true);

-do { // CONSUMATORE
    wait(full);
    wait(mutex);
    ...
    // rimuovi un elemento dal buffer e mettilo in next_consumed
    ...
    signal(mutex);
    signal(empty);
    ...
    // consuma l'elemento contenuto in next_consumed
    ...
} while (true);
```

È interessante notare la simmetria esistente tra il produttore e il consumatore. Il codice si può interpretare nel senso di produzione, da parte del produttore, di posizioni piene per il consumatore; oppure di produzione, da parte del consumatore, di posizioni vuote per il produttore.

Problema dei lettori-scrittori

Si supponga che una base di dati da condividere tra numerosi processi concorrenti. Alcuni processi possono richiedere solo la lettura del contenuto dell'oggetto condiviso, mentre altri possono richiedere un aggiornamento, vale a dire una lettura e una scrittura, dello stesso oggetto. Questi due processi sono distinti, e si indicano chiamando **lettori** (o readers) quelli interessati alla sola lettura e **scrittori** (o writers) 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 alla base di dati condivisa. Questo problema di sincronizzazione è conosciuto come **problema dei lettori-scrittori** (readers-writers problem). Da quando tale problema fu enunciato, è stato usato per verificare quasi tutte le nuove primitive di sincronizzazione. 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 si fonda sul presupposto 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 e quella del secondo possono condurre a uno stato di starvation, degli scrittori, nel primo caso; dei lettori, nel secondo. Per questo motivo sono state proposte altre varianti. Noi presenteremo una soluzione del primo problema (quindi senza occuparci della starvation).

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

```
semaforo mutex, rw_mutex;
int read_count;
```

I semafori *mutex* e *rw_mutex* sono inizializzati a 1; *read_count* è inizializzato a 0. Il semaforo *rw_mutex* è comune a entrambi i tipi di processi (lettura e scrittura). Il semaforo *mutex* si usa per assicurare la mutua esclusione al momento dell'aggiornamento di *read_count*. La variabile *read_count* contiene il numero dei processi che stanno attualmente leggendo l'insieme di dati. Il semaforo *rw_mutex* 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.

Il seguente blocco di codice illustra la struttura generale di un processo scrittore:

```
-do {
    wait(rw_mutex);
    ...
    // esegui l'operazione di scrittura
    ...
    signal(rw_mutex)
} while (true);
```

Mentre, la struttura generale di un lettore è definita come segue:

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

Occorre notare che se uno scrittore si trova nella sezione critica e n lettori attendono di entrarvi, si accoda un lettore a *rw_mutex* e $n - 1$ lettori a *mutex*.

Inoltre, se uno scrittore esegue *signal(rw_mutex)* si può riprendere l'esecuzione dei lettori in attesa, oppure di un singolo scrittore in attesa.

La scelta è fatta dallo scheduler

Le soluzioni al problema dei lettori-scrittori sono state generalizzate su alcuni sistemi in modo da fornire **lock di lettura-scrittura**. Per acquisire un tale lock, è necessario specificare la modalità di scrittura o di lettura: se il processo desidera solo leggere i dati condivisi, richiede un lock di lettura-scrittura in modalità lettura; se invece desidera anche modificare i dati, lo richiede in modalità scrittura. È permesso a più processi di acquisire lock di lettura-scrittura in modalità lettura, ma solo un processo alla volta può avere il lock di lettura-scrittura in modalità scrittura, visto che nel caso della scrittura è necessario garantire l'accesso esclusivo.

I lock di lettura-scrittura sono massimamente utili nelle situazioni seguenti:

- Nelle applicazioni in cui è facile identificare i processi che si limitano alla lettura di dati condivisi e quelli che si limitano alla scrittura di dati condivisi.
- Nelle applicazioni che prevedono più lettori che scrittori. Infatti, i lock di lettura-scrittura comportano in genere un carico di lavoro aggiuntivo rispetto ai semafori o ai lock mutex, compensato però dalla possibilità di eseguire molti lettori in concorrenza.

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.



Il problema dei cinque filosofi è considerato un classico problema di sincronizzazione, non certo per la sua importanza pratica, e neanche per antipatia verso i filosofi da parte degli informatici, ma perché rappresenta una vasta classe di problemi di controllo della concorrenza, in particolare i problemi caratterizzati dalla necessità di assegnare varie risorse a diversi processi evitando situazioni di deadlock e starvation.

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 `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 (true);
```

Questa soluzione garantisce che due vicini non mangino contemporaneamente, ma è insufficiente poiché non esclude la possibilità che si abbia una situazione di deadlock. Si supponga che tutti e cinque i filosofi abbiano fame contemporaneamente e che ciascuno tenti di afferrare la bacchetta di sinistra; tutti gli elementi di *bacchetta* a diventano uguali a zero, perciò ogni filosofo che tenta di afferrare la bacchetta di destra entra in deadlock.

Di seguito sono elencate diverse possibili soluzioni per tali situazioni di deadlock:

- solo quattro filosofi possono stare contemporaneamente a tavola;
- un filosofo può prendere le sue bacchette solo se sono entrambe disponibili (occorre notare che 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.

Si noti che qualsiasi soluzione soddisfacente per il problema dei cinque filosofi deve escludere la possibilità di situazioni di starvation, in altre parole che uno dei filosofi muoia di fame (da qui il termine *starvation*), una soluzione immune alle situazioni di stallo non esclude necessariamente la possibilità di situazioni di starvation.

Monitor

Benché i semafori costituiscano un meccanismo pratico ed efficace per la sincronizzazione dei processi, il loro uso scorretto può generare errori difficili da individuare, in quanto si manifestano solo in presenza di particolari sequenze di esecuzione che non si verificano sempre.

L'impiego di contatori nell'ambito della soluzione al problema dei produttori e consumatori rappresenta un esempio di tali errori. In quella circostanza, il problema di sincronizzazione appariva solo sporadicamente, e anche il valore del contatore si manteneva entro limiti ragionevoli, essendo sfasato tutt'al più di 1. Ciononostante, le soluzioni di questo tipo restano inaccettabili, ed è per ottenere soluzioni soddisfacenti che sono stati inventati i semafori.

Neanche l'uso dei semafori, purtroppo, esclude la possibilità che si verifichi qualche errore di sincronizzazione. Per capire perché, analizziamo la soluzione al problema della sezione critica. Tutti i processi condividono una variabile semaforo *mutex*, inizializzata a 1. Ogni processo deve eseguire *wait(mutex)* prima di entrare nella sezione critica e *signal(mutex)* al momento di uscirne. Se questa sequenza non è rispettata, può accadere che due processi occupino simultaneamente le rispettive sezioni critiche. Esaminiamo le difficoltà che possono insorgere (si noti che tali difficoltà possono insorgere anche nel caso che un solo processo abbia delle pecche. L'inconveniente può nascere da un involontario errore di programmazione o essere causato dalla negligenza del programmatore).

- Supponiamo che un processo capovolga l'ordine in cui sono eseguite le istruzioni *wait()* e *signal()*, in questo modo:

```
signal (mutex) ;  
...  
// sezione critica  
...  
wait (mutex) ;
```

In questa situazione, numerosi processi possono eseguire le proprie sezioni critiche allo stesso tempo, infrangendo il requisito della mutua esclusione. Questo errore può essere scoperto solo qualora diversi processi siano attivi simultaneamente nelle rispettive sezioni critiche. Si osservi che tale situazione potrebbe non essere sempre riproducibile

- Ipotizziamo che un processo sostituisca signal(mutex) con wait(mutex), cioè che esegua

```

wait(mutex);
...
// sezione critica
...
wait(mutex);

```

Si genera, in questo caso, un deadlock.

- Si supponga che un processo ometta wait(mutex), signal(mutex), o entrambi. In questo caso si viola la mutua esclusione oppure si genera un deadlock.

Questi esempi chiariscono come sia facile incorrere in errori allorché i programmatore utilizzino i semafori in maniera scorretta, nel tentativo di risolvere il problema delle sezioni critiche.

Per rimediare a questi errori, i ricercatori hanno sviluppato costrutti con un linguaggio ad alto livello. Un costrutto fondamentale di sincronizzazione ad alto livello (il tipo **monitor**) è descritto nel sottoparagrafo successivo.

Uso del costrutto monitor

Un tipo, o tipo di dato astratto, incapsula i dati privati mettendo a disposizione dei metodi pubblici per operare su tali dati. Il tipo monitor presenta un insieme di operazioni definite dal programmatore che, all'interno del monitor, sono contraddistinte dalla mutua esclusione. Il tipo monitor contiene anche la dichiarazione delle variabili i cui valori definiscono lo stato di un'istanza del tipo, oltre ai corpi delle procedure o funzioni che operano su tali variabili. La sintassi di un monitor è mostrata nel codice seguente:

```

monitor nome {
    // dichiarazioni di variabili condivise

    procedure p1(...) { ... }

    procedure p2(...) { ... }

    ...

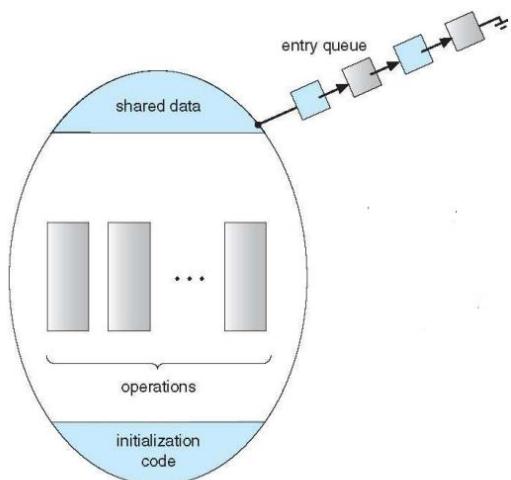
    procedure pn(...) { ... }

    // codice d'inizializzazione (...) {...}
}

```

La rappresentazione di un tipo monitor non può essere usata direttamente dai vari processi. Pertanto, una procedura definita all'interno di un monitor ha accesso unicamente alle variabili dichiarate localmente, situate nel monitor, e ai relativi parametri formali. In modo analogo, alle variabili locali di un monitor possono accedere solo le procedure locali.

Il costrutto monitor assicura che all'interno di un monitor possa essere attivo un solo processo alla volta, sicché non si deve codificare esplicitamente il vincolo di mutua esclusione (vedi figura a destra). Tale definizione di monitor non è abbastanza potente per esprimere alcuni schemi di sincronizzazione, sono perciò necessari ulteriori meccanismi

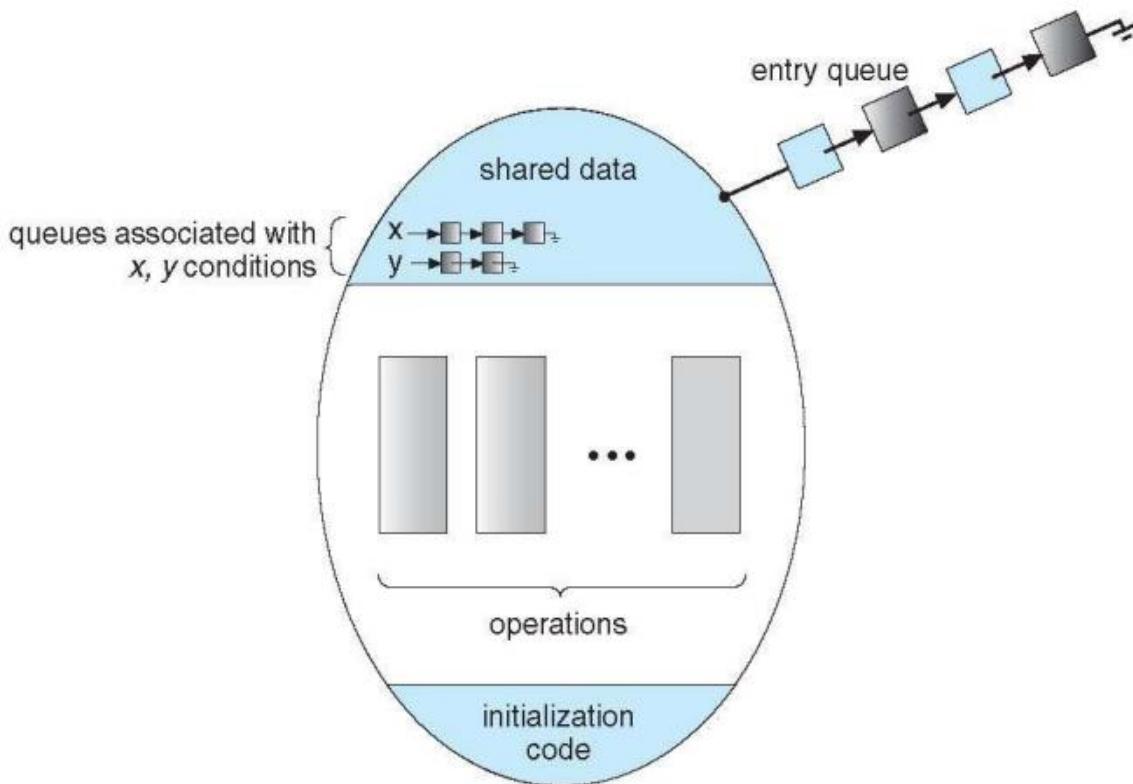


che, in questo caso, sono forniti dal costrutto condition.

Un programmatore che deve scrivere un proprio schema di sincronizzazione può definire una o più variabili condizionali: 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 finché un altro processo non invochi l'operazione $x.signal()$; che risveglia esattamente un processo sospeso. Se non esistono processi sospesi l'operazione signal() non ha alcun effetto, vale a dire che lo stato di x resta immutato, come se l'operazione non fosse stata eseguita; la situazione è descritta nella figura sottostante. Tutto ciò contrasta con l'operazione signal() associata ai semafori, poiché questa influisce sempre sullo stato del semaforo.



Si supponga, per esempio, che quando un processo P invoca l'operazione $x.signal()$, esista un processo sospeso Q associato alla variabile x di tipo condition. Chiaramente, se al processo sospeso Q si permette di riprendere l'esecuzione, il processo segnalante P è costretto ad attendere, altrimenti P e Q sarebbero contemporaneamente attivi all'interno del monitor. Occorre in ogni modo notare che, concettualmente, entrambi i processi possono continuare l'esecuzione. Sussistono quindi due possibilità:

1. **Signal and wait.** P attende che Q lasci il monitor o attenda su un'altra variabile condition.
2. **Signal and continue.** Q attende che P lasci il monitor o attenda su un'altra variabile condition.

Si possono fornire argomenti ragionevoli a favore dell'uno o dell'altra opzione. Da un lato, visto che P era già in esecuzione all'interno del monitor, il secondo metodo appare più ragionevole. D'altro canto, se si lascia proseguire il thread P , la condizione attesa da Q potrebbe non valere più al momento in cui quest'ultimo riprende l'esecuzione. Il linguaggio Concurrent Pascal ha scelto un compromesso: quando il thread P esegue l'operazione signal(), lascia subito il monitor; pertanto, Q riprende immediatamente l'esecuzione.

Soluzione al problema dei cinque filosofi per mezzo di monitor

Illustriamo quindi i concetti relativi al costrutto monitor presentando una soluzione esente da stallo del problema dei cinque filosofi. La soluzione impone però il vincolo che un filosofo possa prendere le sue

bacchette solo quando siano entrambe disponibili. Per codificare questa soluzione si devono distinguere i tre diversi stati in cui può trovarsi un filosofo. A tale scopo si introduce la seguente struttura dati:

```
enum{pensa, affamato, mangia} stato[5];
```

Il filosofo i può impostare la variabile $stato[i] = mangia$ solo se i suoi due vicini non stanno mangiando:

$$((stato[(i + 4)\%5] \neq mangia) \&\& (stato[(i + 1)\%5] \neq mangia))$$

Inoltre, occorre impiegare la seguente struttura dati: condition $self[5]$; dove il filosofo i può ritardare se stesso quando ha famo, ma non riesce a ottenere le bacchette di cui ha bisogno.

A questo punto si può descrivere la soluzione al problema dei cinque filosofi. La distribuzione delle bacchette è controllata dal monitor fc (vedi codice seguente).

```
1 -monitor fc {
2     enum {pensa, affamato, mangia} stato[5];
3     condition self[5];
4
5     - prende(int i) {
6         stato[i] = affamato;
7         verifica(i);
8         if (stato[i] != mangia)
9             self[i].wait();
10    }
11
12    - posa(int i) {
13        stato[i] = pensa;
14        verifica((i + 4) % 5); // testa a sinistra
15        verifica((i + 1) % 5); // testa a destra
16    }
17
18    - verifica(int i) {
19        if ((stato[(i + 4) \% 5] != mangia) \&\&
20            (stato[i] == affamato) \&\&
21            (stato[(i + 1) \% 5] != mangia))
22        {
23            stato[i] = mangia;
24            self[i].signal();
25        }
26    }
27
28    - codice di inizializzazione() {
29        for (int i = 0; i < 5; ++i)
30            stato[i] = pensa;
31    }
32 }
```

Ciascun filosofo, prima di cominciare a mangiare, deve invocare l'operazione $prende()$; ciò può determinare la sospensione del processo filosofo. Completata con successo l'operazione, il filosofo può mangiare; in seguito, il filosofo invoca l'operazione $posa()$ e comincia a pensare. Il filosofo i deve quindi chiamare le operazioni $prende()$ e $posa()$ nella seguente sequenza: $fc.prende(i); \dots fc.posa(i)$;

È facile dimostrare che questa soluzione assicura che due vicini non mangino contemporaneamente e che non si verifichino situazioni di deadlock. Occorre però notare che è possibile lo starvation (un filosofo può attendere all'infinito). La soluzione di questo problema è lasciata come esercizio.

Realizzazione di un monitor per mezzo di semafori

A questo punto si considera la possibilità di realizzare il meccanismo del monitor usando i semafori. A ogni monitor si associa un semaforo *mutex*, inizializzato a 1; un processo deve eseguire *wait(mutex)* prima di entrare nel monitor, e *signal(mutex)* dopo aver lasciato il monitor.

Poiché un processo che esegue una *signal()* deve attendere finché il prossimo risvegliato si metta in attesa o lasci il monitor, si introduce un altro semaforo, *next*, inizializzato a 0, a cui i processi che eseguono una *signal()* possono autosospendersi. Per contare i processi sospesi al semaforo *next*, si usa una variabile intera *next_count*. Quindi, ogni procedura esterna di monitor *F* si sostituisce col seguente codice:

```
wait (mutex) ;  
...  
// corpo di F  
...  
if (next_count > 0)  
    signal (next) ;  
else  
    signal (mutex) ;
```

In questo modo si assicura la mutua esclusione all'interno del monitor.

A questo punto si può descrivere la realizzazione delle variabili *condition*. Per ogni variabile *x* di tipo *condition* si introducono un semaforo *x_sem* e una variabile intera *x_count*, entrambi inizializzati a 0.

Operazione *x.wait()*:

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

Operazione *x.signal()*:

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

Ripresa dei processi all'interno di un monitor

A questo punto si discute il problema dell'ordine di ripresa dei processi all'interno di un monitor. Se più processi sono sospesi alla condizione *x*, e se qualche processo esegue l'operazione *x.signal()*, è necessario stabilire quale tra i processi sospesi si debba riattivare per primo. Una semplice soluzione consiste nell'usare un ordinamento FCFS, secondo cui il processo che attende da più tempo viene ripreso per primo. Tuttavia, in molti casi uno schema di scheduling di questo tipo non risulta adeguato; in questi casi si può usare un costrutto di **conditional wait** (attesa condizionale) della forma *x.wait(c)*; dove con *c* si indica un'espressione intera che si valuta al momento dell'esecuzione dell'operazione *wait()*. Il valore di *c*, chiamato **priority number** (numero di priorità), viene poi memorizzato col nome del processo sospeso. Quando si esegue *x.signal()*, si riprende il processo cui è associato il numero di priorità più basso.

Per comprendere questo nuovo meccanismo, si consideri il seguente monitor:

```
-monitor ResourceAllocator {
    boolean busy;
    condition x;
    - void acquire(int time) {
        if (busy)
            x.wait(time);
        busy = true;
    }
    - void release() {
        busy = false;
        x.signal();
    }
    - codice di inizializzazione() {
        busy = false;
    }
}
```

Per accedere alla risorsa in questione il processo deve rispettare la sequenza
 $R.acquire(t);$
...
 $R.release();$
dove R è un'istanza di tipo ResourceAllocator.

Il suddetto monitor ha il compito di assegnare una particolare risorsa a processi in competizione. Quando richiede l'assegnazione di una delle sue risorse, ogni processo specifica il tempo massimo per il quale prevede di usare la risorsa. Il monitor assegna la risorsa al processo con la richiesta di assegnazione più breve.

Sfortunatamente il concetto di monitor non può garantire che la precedente sequenza d'accesso sia rispettata. In particolare, può accadere quanto segue:

- un processo può accedere alla risorsa senza prima ottenere il permesso d'accesso;
- una volta che ne ha ottenuto l'accesso, un processo può non rilasciare più la risorsa;
- un processo può tentare di rilasciare una risorsa che non ha mai richiesto;
- un processo può richiedere due volte la stessa risorsa, senza rilasciarla prima della seconda richiesta.

Una possibile soluzione del problema precedente prevede l'inclusione delle operazioni d'accesso alle risorse all'interno del monitor ResourceAllocator. Tuttavia, adottando questa soluzione, per lo scheduling delle risorse si userebbe l'algoritmo di scheduling del monitor anziché quello desiderato.

Monitor in Java

Per la sincronizzazione dei thread Java fornisce un meccanismo affine a quello del monitor. Ciascun oggetto, in Java, ha associato un singolo lock. Quando si dichiara un metodo synchronized, per invocare il metodo su un oggetto occorre possedere il lock dell'oggetto.

Il codice che segue dichiara metodoSicuro() come synchronized:

```
-public class Classe {
    ...
    public synchronized void metodoSicuro() { ... }
    ...
}
```

Si supponga di creare un'istanza di Classe, nel modo seguente: Classe sc = new Classe();

Per richiamare il metodo sc.metodoSicuro() è necessario possedere il lock dell'oggetto istanza sc. Se il lock è già proprietà di un thread diverso, il thread che invoca il metodo dichiarato synchronized si blocca ed è collocato nella lista d'attesa per il lock. La lista d'attesa è formata dall'insieme di thread che attendono la disponibilità del lock. Se, al momento dell'invocazione di un metodo synchronized dell'istanza, il lock è disponibile, il thread chiamante diviene il proprietario del lock dell'oggetto e può

accedere al metodo. Il lock ritorna disponibile quando il thread termina l'esecuzione del metodo; un thread in lista d'attesa, quindi, è selezionato come nuovo proprietario del lock.

Java offre inoltre i metodi `wait()` e `notify()`, che funzionano analogamente alle istruzioni `wait()` e `signal()` per imonitor. Nella versione 1.5 la macchina virtuale Java comprende (tra gli altri meccanismi per la concorrenza) la API del package `java.util.concurrent` che mette a disposizione semafori, variabili condizionali e semafori mutex.

Esempi di sincronizzazione

Solaris, Windows XP e Linux sono esempi di sistemi operativi moderni che offrono vari meccanismi come semafori, mutex, spinlock e variabili condizionali per il controllo dell'accesso ai dati condivisi. La API Pthreads fornisce supporto a mutex e variabili condizionali.

Sincronizzazione in Solaris

Per regolare l'accesso alle sezioni critiche, Solaris mette a disposizione semafori mutex adattivi, variabili condizionali, semafori, lock di lettura-scrittura e i cosiddetti tornelli (turnstiles).

Un **adaptive mutex** (mutex adattivo) protegge l'accesso a ogni elemento critico di dati; in un sistema multiprocessore si attiva come un semaforo ordinario realizzato come uno spinlock. Se i dati sono soggetti a lock e quindi già in uso, nel mutex adattivo si possono verificare due situazioni: se i dati sono posseduti da un thread correntemente in esecuzione in un'altra unità d'elaborazione, il thread che ha fatto la nuova richiesta d'accesso entra in uno stato d'attesa attiva (spinlock), mentre aspetta la rimozione del lock, poiché è probabile che il thread in possesso dei dati termini la propria elaborazione in breve tempo; viceversa, se quest'ultimo non si trova nello stato d'esecuzione, il thread richiedente si sospende nello stato d'attesa fino alla rimozione del lock. Solaris adotta l'adative mutex solo per segmenti di codice molto corti; per segmenti di codice più lunghi il sistema ricorre all'impiego di semafori o **variabili condizionali**.

I lock di **lettura-scrittura** si usano per proteggere i dati cui si accede spesso, e di solito per la sola lettura. In tali circostanze essi sono più efficienti dei semafori, poiché più thread possono leggere i dati in modo concorrente, mentre un semaforo avrebbe imposto la serializzazione di questi accessi. Poiché la sua realizzazione introduce un costo aggiuntivo, anche i lock di lettura-scrittura si applicano solo alle sezioni di codice lunghe.

Solaris utilizza i cosiddetti **turnstiles** (tornelli) per ordinare la lista dei thread che attendono di otte' nere un mutex adattivo o un lock di lettura-scrittura. Un turnstile è una struttura a coda contenente i thread che attendono il rilascio di un lock.

Sincronizzazione in Windows XP

Il sistema operativo Windows XP ha un kernel multithread che offre anche la gestione di applicazioni per le elaborazioni in tempo reale e di architetture multiprocessore. Quando il kernel di Windows XP accede a una risorsa globale in un sistema con singola CPU, disabilita temporaneamente le interruzioni aventi procedure di gestione che potrebbero accedere alla stessa risorsa globale. In un sistema multiprocessore, si protegge l'accesso alle risorse globali con i semafori ad attesa attiva (spinlock).

Per la sincronizzazione fuori dal kernel, il sistema operativo offre gli oggetti **dispatcher**, che permettono ai thread di sincronizzarsi servendosi di diversi meccanismi, inclusi mutex, semafori, eventi e timer. I dati condivisi si possono proteggere richiedendo che un thread entri in possesso di un mutex prima di potervi accedere, e rilasci il mutex al completamento dell'elaborazione di quei dati. Gli **eventi** sono un meccanismo di sincronizzazione utilizzabile in modo simile alle variabili condizionali; cioè, possono notificare il verificarsi di una determinata condizione a un thread che l'attendeva. Infine, i timer sono usati per informare un thread (o più di uno) della scadenza di uno specifico periodo di tempo.

Gli oggetti dispatcher possono essere nello stato signaled o nello stato nonsignaled. Uno stato signaled indica che l'oggetto è disponibile e che un thread che tentasse di accedere all'oggetto non sarebbe bloccato; uno stato nonsignaled indica che l'oggetto non è disponibile e che qualsiasi thread che tentasse di accedervi sarebbe bloccato.

Sincronizzazione dei processi in Linux

Prima della versione 2.6, Linux adoperava un kernel senza prelazione; ciò significa che non consentiva di applicare la prelazione ai processi eseguiti in modalità di sistema; neppure nel caso che processi con priorità più alta fossero pronti per l'esecuzione. Ora, per contro, il kernel di Linux ha adottato compiutamente il procedimento della prelazione, cosicché i task attivi nel kernel possono essere sottoposti a prelazione.

Il kernel di Linux si serve di spinlock e semafori (nonché della variante lettore-scrittore di questi due meccanismi) per implementare i lock a livello kernel. Sulle macchine SMP, il meccanismo fondamentale è lo spinlock; il kernel è progettato in modo da mantenere attivi gli spinlock solo per brevi periodi di tempo. Sulle macchine monoprocesso, gli spinlock sono inadatti, e si ricorre all'abilitazione e inibizione del diritto di prelazione nel kernel. Su tali macchine, in pratica, anziché attivare uno spinlock, il kernel inibisce la prelazione; anziché rimuovere lo spinlock, abilita la prelazione.

Il modo impiegato da Linux per abilitare e inibire il diritto di prelazione nel kernel è di particolare interesse; si basa su due semplici chiamate di sistema, `preempt_disable()` e `preempt_enable()`. Va detto che non è possibile sottoporre il kernel a prelazione se un task attivo nella modalità di sistema possiede un lock. Per aggirare l'ostacolo, ogni task nel sistema possiede una struttura, `thread_info`, in cui un contatore, `preempt_count`, indica il numero dei lock attivi nel sistema. Quando un lock entra in funzione, `preempt_count` aumenta di uno, mentre diminuisce di uno quando ne viene rimosso. Qualora il valore di `preempt_count` per il task in esecuzione sia maggiore di zero, sarebbe rischioso sottoporre a prelazione il kernel, dato che il task possiede un lock. Se il valore è zero, il kernel può subire l'interruzione (assumendo che non vi siano chiamate in sospeso a `preempt_disable()`).

Gli spinlock, insieme all'abilitazione e inibizione della prelazione, sono utilizzati nel kernel solo quando si ricorre per breve tempo a un lock (o all'inibizione della prelazione del kernel). Quando vi sia necessità di mantenere un lock attivo più a lungo, è opportuno utilizzare i semafori.

Sincronizzazione in Pthreads

La API Pthreads fornisce lock mutex, variabili condizionali e lock di lettura-scrittura per la sincronizzazione dei thread; è a disposizione dei programmati e non fa parte di questo o quel kernel. I lock mutex rappresentano, in ambiente Pthreads, la tecnica di sincronizzazione fondamentale. La loro finalità è di proteggere le sezioni critiche del codice: un thread che sia in procinto di entrare in una sezione critica si appropria del lock, quindi, al momento di uscirne, lo rimuove. Le variabili condizionali si comportano in Pthreads in maniera molto simile a quanto descritto nel paragrafo sui [Monitor](#). I lock di lettura-scrittura hanno un funzionamento analogo al meccanismo descritto al sottoparagrafo [Problema dei lettori-scrittori](#). Molti sistemi predisposti per l'uso di Pthreads, inoltre, offrono semafori, sebbene non rientrino nello standard Pthreads; appartengono, invece, all'estensione POSIX SEM. Tra le altre estensioni della API Pthreads figurano gli spinlock, malgrado non tutte le estensioni siano ritenute portabili da un implementazione a un'altra.

Lock mutex di Pthreads

Il brano di codice seguente illustra come si possano sfruttare i lock mutex forniti dalla API di Pthreads per

proteggere una sezione critica:

```
#include <pthread.h>
pthread_mutex_t mutex;

/* crea il lock mutex */
pthread_mutex_init(&mutex, NULL);

/* acquisisce il lock mutex */
pthread_mutex_lock(&mutex);

/*** sezione critica ***/

/* rimuove il lock mutex */
pthread_mutex_unlock(&mutex);
```

Pthead utilizza il tipo di dati pthread_mutex_t per i lock mutex. Un mutex è generato con la funzione pthread_mutex_init(&mutex,NULL), in cui il primo parametro è un puntatore al mutex. Passando NULL come secondo parametro, il mutex ha gli attributi di default. Il mutex è attivato e disattivato con le funzioni pthread_mutex_lock() e pthread_mutex_unlock(). Se il lock mutex non è disponibile quando è invocata pthread_mutex_lock(), il thread chiamante rimane bloccato finché il proprietario del lock invoca pthread_mutex_unlock(). Tutte le funzioni mutex restituiscono il valore 0 in assenza d'errori e un valore non nullo altrimenti.

Semafori di Pthreads

Pthreads fornisce due tipi di semafori: con o senza nome. Per questo progetto, impiegheremo semafori senza nome. Il codice seguente mostra come si dà origine a un semaforo:

```
#include <semaphore.h>
sem_t sem;

/* Crea il semaforo e lo inizializza a 5 */
sem_init(&sem, 0, 5);
```

La funzione sem_init(), che crea e inizializza il semaforo, accetta tre parametri:

1. un puntatore al semaforo;
2. un flag indicante il livello di condivisione;
3. il valore iniziale del semaforo

Nell'esempio precedente, passare il flag 0 significa stabilire che questo semaforo è condivisibile unicamente dai thread appartenenti al medesimo processo che ha creato il semaforo. Un valore non nullo consentirebbe l'accesso al semaforo anche da parte di altri processi. In questo esempio, il valore iniziale del semaforo è inizializzato a 5.

Nel paragrafo [Semafori](#) abbiamo descritto le classiche operazioni sui semafori wait() e signal(). Pthreads richiama le operazioni wait() e signal() rispettivamente nell'ordine sem_wait() e sem_post(). L'esempio di codice illustrato nel codice seguente genera un semaforo binario mutex con valore iniziale 1 ,

e ne illustra l'uso nel proteggere una sezione critica:

```
#include <semaphore.h>
sem_t mutex;

/* crea il semaforo */
sem_init(&mutex, 0, 1);

/* acquisisce il semaforo */
sem_wait(&mutex);

/*** sezione critica ***/
/* restituisce il semaforo */
sem_post(&mutex);
```

7. Deadlocks

Modello del sistema

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

Se un processo richiede un'istanza relativa a un tipo di risorsa, l'assegnazione di qualsiasi istanza di quel tipo può soddisfare la richiesta. Se ciò non si verifica significa che le istanze non sono identiche e le classi di risorse non sono state definite correttamente. Un sistema può, per esempio, avere due stampanti; se a nessuno interessa sapere quale sia la stampante in funzione, le due stampanti si possono definire come appartenenti alla stessa classe di risorse; se, però, una stampante si trova al nono piano e l'altra al piano terra, allora le due stampanti si possono considerare non equivalenti, e per definire ciascuna delle due può essere necessario ricorrere a classi di risorse distinte.

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.

1. **Request.** 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.
2. **Use.** Il processo può operare sulla risorsa (se, per esempio, la risorsa è una stampante, il processo può effettuare una stampa).
3. **Release.** Il processo rilascia la risorsa.

La richiesta e il rilascio di risorse avvengono tramite chiamate di sistema, come illustrato nel [Capitolo 2](#).

Caratterizzazione delle deadlock

In una situazione di deadlock, 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 le caratteristiche del problema della deadlock.

Condizioni necessarie

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

1. **Mutual exclusion** (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.
2. **Hold and wait** (Possesso e attesa): Un processo in possesso di almeno una risorsa attende di acquisire risorse già in possesso di altri processi.
3. **No preemption** (Impossibilità di prelazione): Non esiste un diritto di prelazione sulle risorse, vale a dire che una risorsa può essere rilasciata dal processo che la possiede solo volontariamente, dopo aver terminato il proprio compito.
4. **Circular wait**: 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 .

Deadlock con mutex Posix

Vediamo come si possa incorrere in situazioni di stallo in un programma multithread di Pthread che usi i lock mutex. La funzione `pthread_mutex_init()` inizializza un semaforo mutex su cui non è attivo un lock. I lock mutex sono acquisiti e restituiti per mezzo di `pthread_mutex_lock()` e `pthread_mutex_unlock()`, rispettivamente.

Se un thread tenta di acquisire un mutex già impegnato, l'invocazione di `pthread_mutex_lock()` blocca il thread finché il possessore del mutex non invochi `pthread_mutex_unlock()`.

Il brano di codice seguente genera due lock mutex:

```
pthread_mutex_t first_mutex;
pthread_mutex_t second_mutex;

pthread_mutex_init(&first_mutex, NULL);
pthread_mutex_init(&second_mutex, NULL);
```

Sì creano poi due thread, di nome *thread_one* e *thread_two*, che possono accedere a entrambi i lock mutex. Sono eseguiti dalle funzioni `do_work_one()` e `do_work_two()`, rispettivamente, come mostrato nei blocchi di codice seguenti:

```
// thread_one esegue in questa funzione
void *do_work_one(void *param) {
    pthread_mutex_lock(&first_thread);
    pthread_mutex_lock(&second_thread);
    /*
     * Fa qualcosa
     */
    pthread_mutex_unlock(&second_thread); // thread_two esegue in questa funzione
    pthread_mutex_unlock(&first_thread);

    pthread_exit(0);
}

// thread_two esegue in questa funzione
void *do_work_two(void *param) {
    pthread_mutex_lock(&second_thread);
    pthread_mutex_lock(&first_thread);
    /*
     * Fa qualcosa
     */
    pthread_mutex_unlock(&first_thread);
    pthread_mutex_unlock(&second_thread);

    pthread_exit(0);
}
```

In questo esempio, *thread_one* tenta di acquisire i lock mutex nell'ordine (1) *first_mutex*, (2) *second_mutex*, mentre *thread_two* tenta di acquisire (1) *second_mutex*, (2) *first_mutex*. Il deadlock è possibile nell'ipotesi che *thread_one* acquisista *first_mutex*, mentre *thread_two* acquisisca nello stesso istante *second_mutex*.

Si noti che il deadlock, pur essendo possibile, non si verifica se *thread_one* è in grado di acquisire e rilasciare entrambi i lock prima che *thread_two* tenti a sua volta di impossessarsene.

L'esempio evidenzia un problema importante per la gestione del deadlock: è difficile identificare e sottoporre a test i deadlock che si verificano solo in determinate condizioni.

Grafo di assegnazione delle risorse

Le situazioni di deadlock 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.

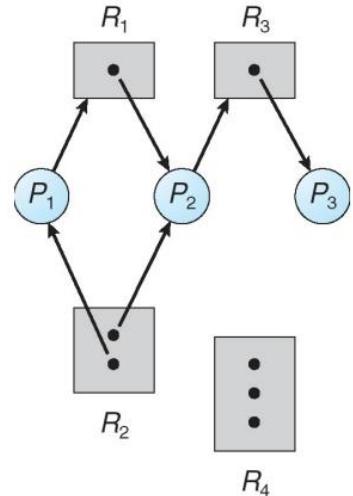
Un arco diretto dal processo P_i al tipo di risorsa R_j si indica $P_i \rightarrow R_j$, e significa che il processo P_i ha richiesto un'istanza del tipo di risorsa R_j , e attualmente attende tale risorsa. Un arco diretto dal tipo di risorsa R_j al processo P_i si indica $P_i \rightarrow R_j$ e significa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

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. Occorre notare che un arco di richiesta è diretto soltanto verso il rettangolo R_j , mentre un arco di assegnazione deve designare anche uno dei puntini 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.

Nel grafo di assegnazione delle risorse nella figura a destra è illustrata la seguente situazione.

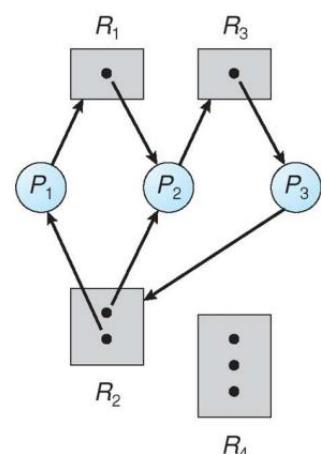
- Insiemi P , R ed E :
 - $P = \{P_1, P_2, P_3\}$
 - $R = \{R_1, R_2, R_3, R_4\}$
 - $E = \{P_1 \rightarrow R_1, P_2 \rightarrow R_2, R_1 \rightarrow P_2, R_2 \rightarrow P_1, R_2 \rightarrow P_2, R_3 \rightarrow P_3\}$
- Istanze delle risorse:
 - un'istanza del tipo di risorsa R_1
 - due istanze del 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, è facile mostrare che, se il grafo non contiene cicli, nessun processo del sistema subisce una deadlock; se il grafo contiene un ciclo, può sopravvenire una deadlock (non è detto che avvenga, quindi il ciclo è condizione necessaria ma non sufficiente).

può sopravvenire uno stallo. Se ciascun tipo di risorsa ha esattamente un'istanza, allora l'esistenza di un ciclo implica la presenza di una deadlock; se il ciclo riguarda solo un insieme di tipi di risorsa, ciascuno dei quali ha solo un'istanza, si è verificata una deadlock. Ogni processo che si trovi nel ciclo è in deadlock. In questo caso l'esistenza di un ciclo nel grafo è una condizione necessaria e sufficiente per l'esistenza di una deadlock. Se ogni tipo di risorsa ha più istanze, un ciclo non implica necessariamente una deadlock. In questo caso l'esistenza di un ciclo nel grafo è una condizione necessaria ma non sufficiente per l'esistenza di una deadlock.

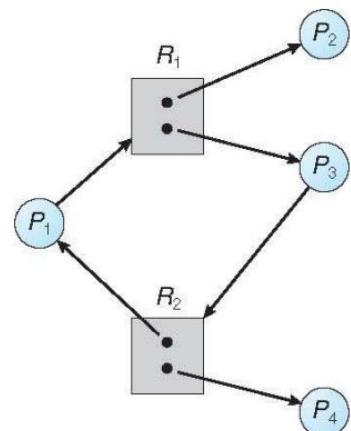
Per spiegare questo concetto conviene ritornare al grafo di assegnazione delle risorse della figura in alto a destra. Si supponga che il processo P_3 richieda un'istanza del tipo risorsa R_2 . Poiché attualmente non è disponibile alcuna istanza di risorsa, si aggiunge un arco di richiesta $P_3 \rightarrow R_2$ al grafo, com'è illustrato nella figura di fianco. A questo punto nel sistema ci sono due cicli minimi: $P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$ e $P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$.



I processi P_1 , P_2 e P_3 sono in deadlock: il processo P_2 attende la risorsa R_3 , posseduta dal processo P_3 ; quest'ultimo, invece, attende il processo P_1 o P_2 rilasci la risorsa R_2 ; inoltre il processo P_1 attende che il processo P_2 rilasci la risorsa R_1 .

Si consideri ora il seguente grafo di assegnazione delle risorse. Anche in questo esempio c'è un ciclo: $P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$. In questo caso però, non si ha alcuna deadlock: 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.

Per concludere, l'assenza di cicli nel grafo di assegnazione delle risorse implica l'assenza di situazioni di stallo nel sistema. Viceversa, la presenza di un ciclo non è sufficiente a implicare la presenza di uno stallo nel sistema. Questa osservazione è importante ai fini della gestione del problema delle situazioni di deadlock.



Metodi per la gestione delle situazioni di deadlock

Essenzialmente, il problema delle situazioni di deadlock si può affrontare impiegando tre metodi:

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

Quest'ultima è la soluzione usata dalla maggior parte dei sistemi operativi, compresi UNIX e Windows. Nel seguito sono spiegati brevemente tutti questi metodi. Inoltre, tali approcci possono essere combinati, in modo da permettere la selezione del migliore per ciascuna classe di risorse del sistema.

Per assicurare che non si verifichi mai una deadlock, il sistema può servirsi di metodi di prevenzione o di metodi per evitare tale situazione. [Prevenire le situazioni di deadlock](#) significa far uso di metodi atti ad assicurare che non si verifichi almeno una delle condizioni necessarie.

Per [evitare le situazioni di deadlock](#) 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 si può decidere se una richiesta di risorse da parte di un processo si può soddisfare o si debba invece sospendere. 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.

Se un sistema non impiega né un algoritmo per prevenire né un algoritmo per evitare gli stalli, tali situazioni possono verificarsi. In un ambiente di questo tipo il sistema può servirsi di un algoritmo che ne esamini lo stato, al fine di stabilire se si è verificato uno stallo e in tal caso ricorrere a un secondo algoritmo per il ripristino del sistema.

Se un sistema non garantisce che le situazioni di deadlock non possano mai verificarsi e non fornisce alcun meccanismo per la loro individuazione e per il ripristino del sistema, situazioni di deadlock possono avvenire senza che ci sia la possibilità di capire cos'è successo. In questo caso la presenza di situazioni di deadlock non rilevate può causare un degrado delle prestazioni del sistema; infatti, la presenza di risorse assegnate a processi che non si possono eseguire determina lo deadlock di un numero crescente di processi che richiedono tali risorse, fino al blocco totale del sistema che dovrà essere riavviato manualmente.

Prevenire le situazioni di deadlock

Questo metodo è descritto nei particolari, con una trattazione di ciascuna delle quattro condizioni necessarie.

Mutua esclusione

Deve valere la condizione di mutua esclusione per le risorse non condivisibili: una stampante non può essere condivisa da più processi. Le risorse condivisibili, invece, non richiedono l'accesso mutuamente esclusivo, perciò non possono essere coinvolte in una deadlock. I file aperti per la sola lettura sono un buon esempio di risorsa condivisibile; è ovviamente consentito l'accesso contemporaneo da parte di più processi. 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.

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 richiederne altre 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 usate può trovarsi nella condizione di attenderne indefiniteamente la disponibilità, poiché almeno una delle risorse di cui necessita è sempre assegnata a qualche altro processo.

Impossibilità di prelazione

La terza condizione necessaria prevede che non sia possibile avere la prelazione su risorse già assegnate. Per assicurare che questa condizione non persista, 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 (cioè 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 che sta richiedendo.

In alternativa, quando un processo richiede alcune risorse, si verifica la disponibilità di queste ultime: 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 su 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.

Questo protocollo è adatto a risorse il cui stato si può salvare e recuperare facilmente in un secondo tempo, come i registri della CPU e lo spazio di memoria, mentre non si può in generale applicare a risorse come le stampanti e le unità a nastri.

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 un ordine crescente di numerazione per le risorse richieste da ciascun processo. Ciò significa che un processo può richiedere inizialmente qualsiasi numero di istanze di un tipo di risorsa, ad esempio R_i , e dopo di che il processo può richiedere istanze del tipo di risorsa R_j se e solo se $f(R_j) > f(R_i)$ con $f: R \rightarrow \mathbb{N}$ (praticamente associa alla risorsa un numero naturale).

Evitare le situazioni di deadlock

Gli algoritmi differiscono tra loro per la quantità e il tipo di informazioni richieste. Il modello più semplice e più utile richiede che ciascun processo dichiari il numero massimo delle risorse di ciascun tipo di cui necessita. Data un'informazione a priori per ogni processo sul massimo numero di risorse richiedibili per ciascun tipo, si può costruire un algoritmo capace di assicurare che il sistema non entri in deadlock. Questo algoritmo definisce un metodo per evitare la deadlock, ed 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.

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 può ancora fare sì che si possono soddisfare impiegando le risorse attualmente disponibili più le risorse possedute da tutti i P_i non sono disponibili immediatamente, allora P_i può attendere che tutti i P_j abbiano finito, e a quel punto P_i può ottenere tutte le risorse di cui ha bisogno, completare il compito assegnato, restituire le risorse assegnate e terminare. Quando P_i 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 deadlock. Viceversa, uno stato di deadlock è uno stato non sicuro; comunque non tutti gli stati non sicuri sono stati di deadlock. Uno stato non sicuro potrebbe condurre a una deadlock. Finché lo stato rimane sicuro, il sistema operativo può evitare il verificarsi di stati non sicuri e di deadlock. In uno stato non sicuro il sistema operativo non può impedire ai processi di richiedere risorse in modo da causare una deadlock: ciò che accade negli stati non sicuri dipende dal comportamento dei processi.

Per illustrare meglio quel che si è detto sopra, si consideri un sistema con 12 unità a nastri magnetici e 3 processi: P_0, P_1 e P_2 . Il processo P_0 può richiedere 10 unità a nastri, il processo P_1 può richiederne 4 e il processo P_2 può richiedere fino a 9. Supponendo che all'istante t_0 il processo P_0 possieda 5 unità a nastri, e che i processi P_1 e P_2 ne possiedano 2 ciascuno, restano libere 3 unità a nastri.

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

All'istante t_0 il sistema si trova in uno stato sicuro. La sequenza $\langle P_1, P_0, P_2 \rangle$ soddisfa la condizione di sicurezza, poiché al processo P_1 si possono assegnare immediatamente tutte le unità a nastri richieste, che saranno poi restituite (a questo punto sono disponibili 5 unità a nastri), quindi il processo P_0 può avere tutte le unità a nastri richieste e restituirle (il sistema ha 10 unità a nastri disponibili) e infine il processo P_2 potrebbe avere tutte le sue unità a nastri e restituirle (sono disponibili tutte e 12 le unità a nastri).

Un sistema può passare da uno stato sicuro a uno stato non sicuro. Si supponga che all'istante t_1 il processo P_2 richieda un'ulteriore unità a nastri e che questa gli sia assegnata: il sistema non si trova più nello stato

sicuro. A questo punto, si possono assegnare tutte le unità a nastri richieste soltanto al processo P_1 . Al momento della restituzione, il sistema avrà solo 4 unità a nastri disponibili. Poiché al processo P_0 sono assegnate 5 unità a nastri, ma il numero massimo è 10, il processo può richiederne altre 5, ma poiché queste non sono disponibili il processo P_0 deve attendere. Analogamente, il processo P_2 può richiedere altre 6 unità a nastri ed è costretto ad attendere; il risultato è una situazione di deadlock. L'errore è stato commesso nel soddisfare la richiesta di un'ulteriore unità a nastri fatta dal processo P_2 . Se P_2 avesse atteso il termine di uno degli altri processi e il conseguente rilascio delle sue risorse, la situazione di deadlock si sarebbe potuta evitare.

Dato il concetto di stato sicuro, si possono definire algoritmi che permettano di evitare le situazioni di deadlock. L'idea è semplice: è sufficiente assicurare che il sistema rimanga sempre in uno stato sicuro.

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 reclamo (*claim edge*). Un **arco di reclamo** $P_i \rightarrow R_j$ 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 reclamo $P_i \rightarrow R_j$ 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 reclamo $P_i \rightarrow R_j$. Occorre sottolineare che le risorse devono essere reclamate a priori nel sistema. Ciò significa che prima che il processo P_j inizi l'esecuzione, tutti i suoi archi di reclamo devono essere già inseriti nel grafo di assegnazione delle risorse. Questa condizione si può indebolire permettendo l'aggiunta di un arco di reclamo $P_i \rightarrow R_j$ al grafo solo se tutti gli archi associati al processo P_i sono archi di reclamo.

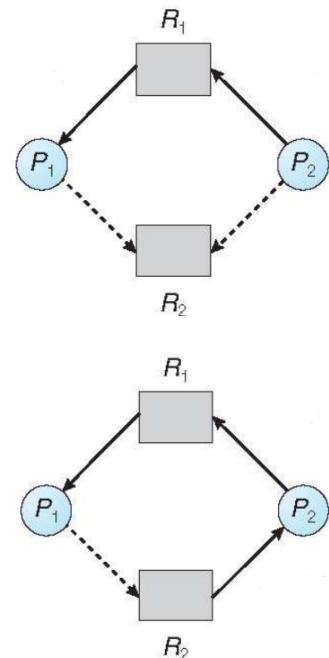
Si supponga che il processo P_i richieda la risorsa R_j . La richiesta si può soddisfare solo se la conversione dell'arco di richiesta $P_i \rightarrow R_j$ nell'arco di assegnazione $R_j \rightarrow P_i$ non causa la formazione di un ciclo nel grafo di assegnazione delle risorse. Occorre ricordare che la sicurezza si controlla con un algoritmo di rilevamento dei cicli, e che un algoritmo per il rilevamento di un ciclo in questo grafo richiede un numero di operazioni dell'ordine di n^2 , dove con n si indica il numero dei processi del sistema.

Se non esiste alcun ciclo, l'assegnazione della risorsa lascia il sistema in uno stato sicuro. Se invece si trova un ciclo, l'assegnazione conduce il sistema in uno stato non sicuro, e il processo P_i deve attendere che si soddisfino le sue richieste.

Per illustrare questo algoritmo si consideri il grafo di assegnazione delle risorse della figura a destra. Si supponga che P_2 richieda R_2 . Sebbene sia attualmente libera, R_2 non può essere assegnata a P_2 , poiché, com'è evidenziato nella figura in basso, quest'operazione creerebbe un ciclo nel grafo e un ciclo indica che il sistema è in uno stato non sicuro. Se, a questo punto, P_1 richiedesse R_2 si avrebbe una deadlock.

Algoritmo del banchiere

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 qui descritto, 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**. Questo nome è stato scelto perché l'algoritmo si potrebbe impiegare in un sistema bancario



per assicurare che la banca non assegna mai tutto il denaro disponibile, poiché, se ciò avvenisse, non potrebbe più soddisfare le richieste di tutti i suoi clienti.

Quando si presenta, un nuovo processo deve dichiarare il numero massimo delle istanze di ciascun tipo di risorsa di cui necessita. 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.

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:

- **Available.** Un vettore di lunghezza m indica il numero delle istanze disponibili per ciascun tipo di risorsa; $available[j] = k$, significa che sono disponibili k istanze del tipo di risorsa R_j .
- **Max.** Una matrice $n \times m$ definisce la richiesta massima di ciascun processo; $max[i, j] = k$ significa che il processo P_i può richiedere un massimo di k istanze del tipo di risorsa R_j .
- **Allocation.** Una matrice $n \times m$ definisce il numero delle istanze di ciascun tipo di risorsa attualmente assegnate a ogni processo; $allocation[i, j] = k$ significa che al processo P_i sono correntemente assegnate k istanze del tipo di risorsa R_j .
- **Need.** Una matrice $n \times m$ indica la necessità residua di risorse relativa a ogni processo; $need[i, j] = k$ 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 $need[i, j] = max[i, j] - available[i, j]$.

Col trascorrere del tempo, queste strutture dati variano sia nelle dimensioni sia nei valori.

Per semplificare la presentazione dell'algoritmo del banchiere, si usano le seguenti notazioni: supponendo che X e Y siano vettori di lunghezza n , si può affermare che $X \leq Y$ se e solo se $X[i] \leq Y[i]$ per ogni $i = 1, 2, \dots, n$. Ad esempio, se $X = (1, 7, 3, 2)$ e $Y = (0, 3, 2, 1)$, allora $Y \leq X$.

Tutte le righe delle matrici *available* e *need* sono considerabili vettori e si possono chiamare rispettivamente $available_i$ e $need_i$. Il vettore $available_i$ specifica le risorse correntemente assegnate al processo P_i , mentre il vettore $need_i$ specifica le risorse che il processo P_i può ancora richiedere per completare il suo compito.

Algoritmo di safety (verifica della sicurezza). L'algoritmo utilizzato per scoprire se il sistema è o non è in uno stato sicuro si può descrivere come segue.

1. Siano $work$ e $finish$ vettori di lunghezza rispettivamente m e n , e inizializza $work = available$ e $finish[i] = \text{false}$, per $i = 1, 2, \dots, n - 1$;
2. Cerca un indice i tale che valgano contemporaneamente le seguenti relazioni:
 - a. $finish[i] == \text{false}$
 - b. $need_i \leq work$
 se tale i non esiste, esegue il passo 4
3. $work = work + allocation_i$ e $finish[i] = \text{true}$ vai al passo 2
4. Se $work[i] == \text{true}$ per ogni i , allora il sistema è in uno stato sicuro (*safe state*).

Per determinare se uno stato è sicuro tale algoritmo può richiedere un numero di operazioni dell'ordine di $m \times n^2$.

Algoritmo di richiesta delle risorse. Si descrive ora l'algoritmo che determina se le richieste possano essere garantite in uno stato sicuro. Sia $request_i$, il vettore delle richieste per il processo P_i . Se $request_i[j] == k$, allora il processo P_i richiede k istanze del tipo di risorsa R_j . Se il processo P_i fa una richiesta di risorse, si svolgono le seguenti azioni:

1. se $request_i \leq need_i$, esegue il passo 2, altrimenti riporta una condizione d'errore, poiché il processo ha superato il numero massimo di richieste;
2. se $request_i \leq available$ vai al passo 3, altrimenti P_i deve attendere poiché le risorse non sono disponibili;
3. simula l'assegnazione al processo P_i delle risorse richieste modificando come segue lo stato di assegnazione delle risorse:
 - a. $available = available - request_i$
 - b. $allocation_i = allocation_i + request_i$
 - c. $need_i = need_i - request_i$

Se lo stato di assegnazione delle risorse risultante è sicuro la transazione è completa e al processo P_i si assegnano le risorse richieste. Tuttavia, se il nuovo stato è non sicuro, P_i deve attendere $request_i$ e si ripristina il vecchio stato di assegnazione delle risorse.

Un esempio. Illustriamo infine 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 "istantanea" (il contenuto della matrice $need$ è definito come $max - available$):

	<u>Allocation</u>	<u>Max</u>	<u>Available</u>	<u>Need</u>
	$A \ B \ C$	$A \ B \ C$	$A \ B \ C$	$A \ B \ C$
P_0	0 1 0	7 5 3	3 3 2	P_0 7 4 3
P_1	2 0 0	3 2 2		P_1 1 2 2
P_2	3 0 2	9 0 2		P_2 6 0 0
P_3	2 1 1	2 2 2		P_3 0 1 1
P_4	0 0 2	4 3 3		P_4 4 3 1

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 $request_1 = (1,0,2)$. Per stabilire se questa richiesta si possa esaudire immediatamente si verifica la condizione $request_i \leq available$ (vale a dire $(1,0,2) \leq (3,3,2)$), che risulta vera, quindi, supponendo che questa richiesta sia stata soddisfatta, si ottiene il seguente nuovo stato:

	<u>Allocation</u>	<u>Need</u>	<u>Available</u>
	$A \ B \ C$	$A \ B \ C$	$A \ B \ C$
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 $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ rispetta il requisito di sicurezza. Quindi si può soddisfare immediatamente la richiesta del processo P_1 .

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 finché non siano disponibili le risorse. Inoltre, una richiesta di $(0,2,0)$ da parte di P_0 non si potrebbe soddisfare neanche se le risorse fossero disponibili, poiché lo stato risultante sarebbe non sicuro.

N.B.: è tipico un esercizio d'esame che richieda di verificare se lo stato è safe o chiedere cosa verrebbe se... (nulla di complicato). In realtà qualsiasi algoritmo visto a lezione è candidato ad essere domanda d'esame.

Rilevamento delle situazioni di stallo

Se un sistema non si avvale di un algoritmo per prevenire o evitare situazioni di stallo, è possibile che si verifichi effettivamente. In tal caso il sistema deve fornire i seguenti algoritmi:

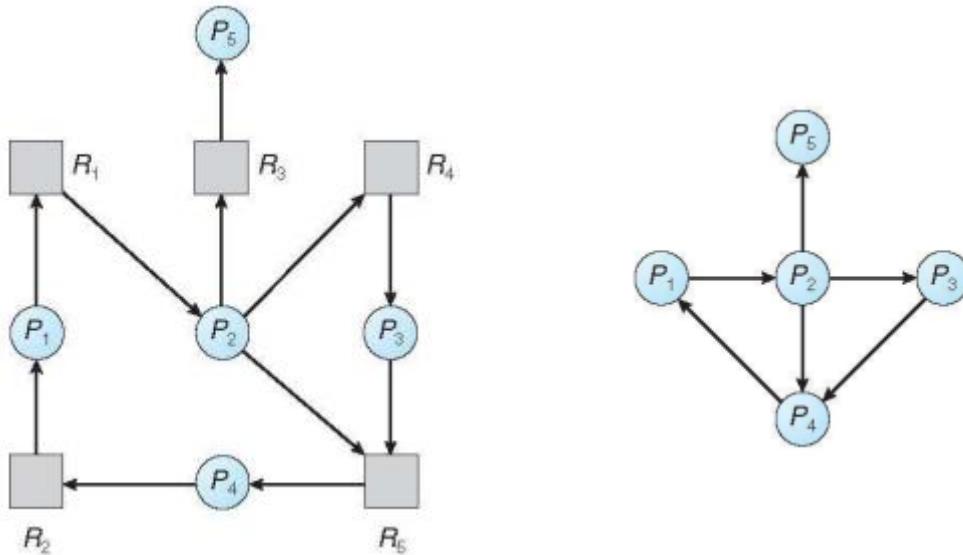
- un algoritmo che esamini lo stato del sistema per stabilire se si è verificato uno stallo;
- un algoritmo che ripristini il sistema dalla condizione di stallo.

Nell'analisi seguente sono trattati i suddetti argomenti che riguardano sia sistemi con una sola istanza di ciascun tipo di risorsa, sia sistemi con più istanze.

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** (wait-for), 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_i ha bisogno. Un arco $P_i \rightarrow P_j$ esiste nel grafo d'attesa se e solo se il corrispondente grafo di assegnazione delle risorse contiene due archi $P_i \rightarrow R_q$ e $R_q \rightarrow P_j$ per qualche risorsa R_q . Nella figura successiva sono illustrati un grafo di assegnazione delle risorse (a sinistra) e il corrispondente grafo d'attesa (a destra).



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 conservare il grafo d'attesa e invocare periodicamente un algoritmo che cerchi un ciclo all'interno del grafo. L'algoritmo per il rilevamento di un ciclo all'interno di un grafo richiede un numero di operazioni dell'ordine di n^2 , dove con n si indica il numero dei vertici del grafo.

Più istanze di ciascun tipo di risorsa

Lo schema con grafo d'attesa non si può applicare ai sistemi di assegnazione delle risorse con più di una istanza. Il seguente algoritmo di rilevamento di situazioni di stallo è, invece, applicabile a tali sistemi. Esso si serve di strutture dati variabili nel tempo, simili a quelle adoperate nell'[algoritmo del banchiere](#).

- **Available**: vettore di lunghezza m che indica il numero delle istanze disponibili per ciascun tipo di risorsa.
- **Allocation**: matrice $n \times m$ che definisce il numero delle istanze di ciascun tipo di risorse correntemente assegnate a ciascun processo.
- **Request**: matrice $n \times m$ che indica la richiesta attuale di ciascun processo. Se $request[i,j] = k$, 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 *available* e *request* si trattano come vettori e, nel seguito, sono indicate rispettivamente come $available_i$ e $request_i$. L'algoritmo di rilevamento descritto indaga su ogni possibile sequenza di assegnazione per i processi che devono ancora essere completati.

1. Siano $work$ e $finish$ vettore di lunghezza rispettivamente m e n , inizializza $work = available$, per $i = 1, 2, \dots, n$, se $allocation_i \neq 0$, allora $finish[i] = \text{false}$, altrimenti $finish[i] = \text{true}$;
2. Cerca un indice i tale che valgano contemporaneamente le seguenti relazioni:
 - a. $finish[i] == \text{false}$
 - b. $request_i \leq work$
 se tale i non esiste, esegue il passo 4.
3. $work = work + allocation_i$
 $finish[i] = \text{true}$
 torna al passo 2
4. Se $finish[i] == \text{false}$ per qualche i , $0 \leq i < n$, allora il sistema è in deadlock, inoltre, se $finish[i] == \text{falso}$, il processo P_i è in deadlock.

Tale algoritmo richiede un numero di operazioni dell'ordine di $m \times n^2$ per controllare se il sistema è in deadlock.

Può meravigliare che le risorse del processo P_i siano richieste (passo 3) non appena risulta valida la condizione $request_i \leq work$ (passo 2.b). Tale condizione garantisce che P_i non è correntemente coinvolto in una deadlock, quindi, assumendo un atteggiamento ottimistico, si suppone che P_i non intenda richiedere altre risorse per completare il proprio compito, e che restituisca presto tutte le risorse. Se non si rispetta l'ipotesi fatta, si può verificare una deadlock, che sarà rilevato quando si richiamerà nuovamente l'algoritmo di rilevamento.

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 istante, 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:

	<u>Allocation</u>			<u>Request</u>			<u>Available</u>		
	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>	<i>A</i>	<i>B</i>	<i>C</i>
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 deadlock. Infatti eseguendo l'algoritmo per la sequenza $\langle P_0, P_2, P_3, P_1, P_4 \rangle$, risulta $finish[i] == \text{true}$ per ogni i . Si supponga ora che il processo P_2 richieda un'altra istanza di tipo C . La matrice *request* viene modificata come segue:

	<u>Request</u>		
	<i>A</i>	<i>B</i>	<i>C</i>
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 deadlock. Anche se si possono reclamare 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 una deadlock composto dai processi P_1, P_2, P_3 e P_4 .

Uso dell'algoritmo di rilevamento

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

1. *frequenza* (presunta) con la quale si verifica una deadlock;
2. *numero* dei processi che sarebbero influenzati da tale deadlock.

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

Le situazioni di deadlock si verificano solo quando qualche processo fa una richiesta che non si può soddisfare immediatamente; può essere una richiesta che chiude una catena di processi in attesa. Il caso estremo è quello nel quale l'algoritmo di rilevamento si usa ogni volta che non si può soddisfare immediatamente una richiesta di assegnazione. In questo caso non si identifica soltanto il gruppo di processi in deadlock, ma anche il processo che ha “causato” la deadlock, anche se, in verità, ciascuno dei processi in deadlock è un elemento del ciclo all'interno del grafo di assegnazione delle risorse, quindi tutti i processi sono, congiuntamente, responsabili della deadlock. Se esistono tipi di risorsa diversi, una singola richiesta può causare più cicli nel grafo delle risorse, ciascuno dei quali è completato dalla richiesta più recente ed è causato da un processo identificabile.

Naturalmente, l'uso dell'algoritmo di rilevamento per ogni richiesta aumenta notevolmente il carico nei termini di tempo di calcolo. Un'alternativa meno dispendiosa è quella in cui l'algoritmo di rilevamento s'invoca a intervalli meno frequenti, per esempio una volta ogni ora, oppure ogni volta che l'utilizzo della CPU scende sotto il 40 per cento, poiché una deadlock può rendere inefficienti le prestazioni del sistema e quindi causare una drastica riduzione dell'utilizzo della CPU. Non è conveniente richiedere l'algoritmo di rilevamento in momenti arbitrari, poiché nel grafo delle risorse possono coesistere molti cicli e, normalmente, non si può dire quale fra i tanti processi in deadlock abbia “causato” la deadlock.

Ripristino da situazioni di deadlock

Una situazione di deadlock si può affrontare in diversi modi. Una soluzione consiste nell'informare l'operatore della presenza della deadlock, in modo che possa gestirlo manualmente. L'altra soluzione lascia al sistema il ripristino automatico dalla situazione di deadlock. Una deadlock si può eliminare in due modi: il primo prevede semplicemente la terminazione di uno o più processi per interrompere l'attesa circolare; il secondo esercita la prelazione su alcune risorse in possesso di uno o più processi in deadlock.

Terminazione di processi

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

- **Terminazione di tutti i processi in deadlock.** Chiaramente questo metodo interrompe il ciclo di deadlock, ma l'operazione è molto onerosa; questi processi possono aver già fatto molti calcoli i cui risultati si annullano e probabilmente dovranno essere ricalcolati.
- **Terminazione di un processo alla volta fino all'eliminazione del ciclo di deadlock.** Questo metodo causa un notevole carico, poiché, dopo aver terminato ogni processo, si deve impiegare un algoritmo di rilevamento per stabilire se esistono ancora processi in deadlock.

Procurare la terminazione di un processo può essere un'operazione tutt'altro che semplice: se il processo si trova nel mezzo dell'aggiornamento di un file, la terminazione lascia il file in uno stato scorretto; analogamente, se il processo si trova nel mezzo di una stampa di dati, prima di stampare un lavoro successivo, il sistema deve reimpostare la stampante riportandola a uno stato corretto.

Se si adopera il metodo di terminazione parziale, dato un insieme di processi in stallo occorre determinare quale processo, o quali processi, costringere alla terminazione nel tentativo di sciogliere la situazione di

deadlock. Analogamente ai problemi di scheduling della CPU, si tratta di scegliere 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:

1. la priorità dei processi;
2. il tempo trascorso dalla computazione e il tempo ancora necessario per completare i compiti assegnati ai processi;
3. la quantità e il tipo di risorse impiegate dai processi (ad esempio, se si può avere facilmente la prelazione sulle risorse);
4. la quantità di ulteriori risorse di cui i processi hanno ancora bisogno per completare i propri compiti;
5. il numero di processi che si devono terminare;
6. il tipo di processi: interattivi o a batch.

Prelazione su 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.

1. **Selezione di una vittima.** Occorre stabilire quali risorse e quali processi si devono sottoporre a prelazione. Come per la terminazione dei processi, è necessario stabilire l'ordine di prelazione allo scopo di minimizzare i costi. I fattori di costo possono includere parametri come il numero delle risorse possedute da un processo in deadlock, e la quantità di tempo già spesa durante l'esecuzione da un processo in deadlock.
2. **Rollback** (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 dal quale essere riavviato. Poiché, in generale, è difficile stabilire quale stato sia sicuro, la soluzione più semplice consiste nel terminare il processo e quindi riavivarlo. Certamente, è più efficace ristabilire un precedente stato sicuro del processo che sia sufficiente allo scioglimento della situazione di deadlock, ma questo metodo richiede che il sistema mantenga più informazioni sullo stato di tutti i processi in esecuzione.
3. **Starvation.** È necessario assicurare che non si verifichino situazioni di starvation, occorre cioè garantire che le risorse non siano sottratte sempre allo stesso processo. In un sistema in cui la scelta della vittima avviene soprattutto secondo fattori di costo, può accadere che si scelga sempre lo stesso processo; in questo caso il processo non riesce mai a completare il suo compito; si tratta di una situazione di starvation che si deve affrontare in qualsiasi sistema concreto.

Plausibili domande orali di questo capitolo:

- Definizione di deadlock
- Condizioni per avere un deadlock
- Grafi di allocazione delle risorse e cicli (dato un grafo, vi è una deadlock?)
- Prevenzione dei deadlock in base alle condizioni di deadlock
- Metodi di evitamento del deadlock per risorse singole (grafo delle allocazioni) e multiple (algoritmo del banchiere)
- Metodi di deadlock detection per risorse singole e multiple
- Metodi di deadlock recovery

8. Memoria Centrale

Introduzione

Come abbiamo visto nel [Capitolo 1](#), la memoria è fondamentale nelle operazioni di un moderno sistema di calcolo; consiste in un ampio vettore di parole o byte, ciascuno con il proprio indirizzo. La CPU preleva le istruzioni dalla memoria sulla base del contenuto del program counter; tali istruzioni possono determinare ulteriori letture (*load*) e scritture (*store*) in specifici indirizzi di memoria.

La memoria vede soltanto un flusso d'indirizzi di memoria, e non sa come sono generati (program counter, indicizzazione, riferimenti indiretti, indirizzamenti immediati e così via), oppure a che cosa servano (istruzioni o dati). Di conseguenza, è possibile ignorare come un programma genera un indirizzo di memoria, e prestare attenzione solo alla sequenza degli indirizzi di memoria generati dal programma in esecuzione.

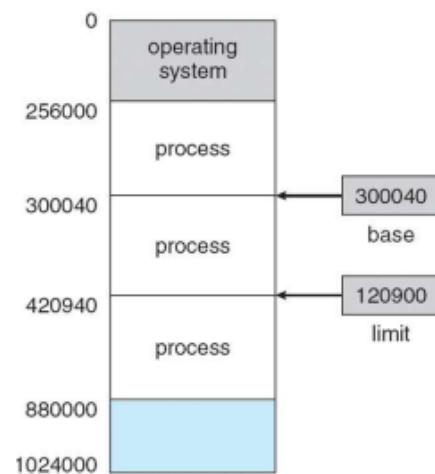
Dispositivi essenziali

La memoria centrale e i registri incorporati nel processore sono le sole aree di memorizzazione a cui la CPU può accedere direttamente. Vi sono istruzioni macchina che accettano gli indirizzi di memoria come argomenti, ma nessuna accetta gli indirizzi del disco. Pertanto, qualsiasi istruzione in esecuzione, e tutti i dati utilizzati dalle istruzioni, devono risiedere in uno di questi dispositivi per la memorizzazione ad accesso diretto. I dati che non sono in memoria devono essere caricati prima che la CPU possa operare su di loro.

I registri incorporati nella CPU sono accessibili, in genere, nell'arco di un clock (un ciclo dell'orologio di sistema). Molte CPU sono capaci di decodificare istruzioni ed effettuare semplici operazioni sui contenuti dei registri alla velocità di una o più operazioni per ciclo. Ciò non vale per la memoria centrale, cui si accede attraverso una transazione sul bus della memoria. Nei casi in cui l'accesso alla memoria richieda molti clock, il processore entra necessariamente in **stallo** (*stall*), poiché manca dei dati richiesti per completare l'istruzione che sta eseguendo. Questa situazione è intollerabile, perché gli accessi alla memoria sono frequenti. Il rimedio consiste nell'interposizione di una memoria veloce tra CPU e memoria centrale. Un buffer di memoria, detto cache, è in grado di conciliare le differenti velocità (vedi sottoparagrafo [Cache](#)).

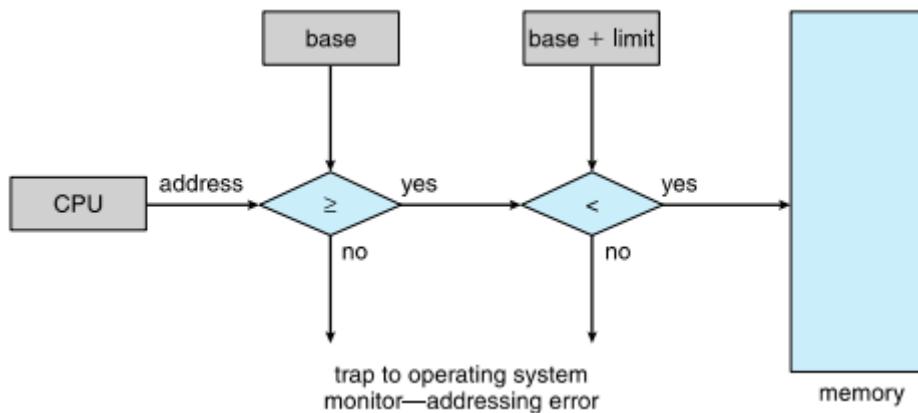
Non basta prestare attenzione alle velocità relative di accesso alla memoria fisica: è anche necessario proteggere il sistema operativo dall'accesso dei processi utenti, e salvaguardare i processi utenti l'uno dall'altro. Tale protezione deve essere messa in atto a livello dei dispositivi; come si vedrà lungo tutto il capitolo, essa può essere realizzata con meccanismi diversi. In questo paragrafo evidenziamo una delle possibili implementazioni.

Innanzitutto, bisogna assicurarsi che ciascun processo abbia uno spazio di memoria separato. A tal fine, occorre poter determinare l'intervallo degli indirizzi a cui un processo può accedere legalmente, e garantire che possa accedere soltanto a questi indirizzi. Si può implementare il meccanismo di protezione tramite due registri, detti **registri base** e **registri limite**, come illustrato nella figura a destra. Il registro base contiene il più piccolo indirizzo legale della memoria fisica; il registro limite determina la dimensione dell'intervallo ammesso. Ad esempio, se i registri base e limite contengono rispettivamente i valori 300040 e 120900, al programma si consente l'accesso alle locazioni di memoria di indirizzi compresi tra 300040 e 420939, estremi inclusi.



Per mettere in atto il meccanismo di protezione, la CPU confronta ciascun indirizzo generato in modalità utente con i valori contenuti nei due registri. Qualsiasi tentativo da parte di un programma eseguito in modalità utente di accedere alle aree di memoria riservate al sistema operativo o a una qualsiasi area di

memoria riservata ad altri utenti comporta l'invio di un segnale di eccezione che restituisce il controllo al sistema operativo che, a sua volta, interpreta l'evento come un errore fatale (figura sottostante).



Questo schema impedisce a qualsiasi programma utente di alterare (accidentalmente o intenzionalmente) il codice o le strutture dati, sia del sistema operativo sia degli altri utenti.

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 nella modalità di sistema, 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 la medesima operazione ai programmi utenti.

Grazie all'esecuzione nella modalità di sistema, il sistema operativo ha la possibilità di accedere indiscriminatamente sia alla memoria a esso riservata sia a quella riservata agli utenti. Questo privilegio consente al sistema di caricare i programmi utenti nelle aree di memoria a loro riservate; di generare copie del contenuto di queste regioni di memoria (*dump*) a scopi diagnostici, qualora si verifichino errori; di modificare i parametri delle chiamate di sistema, e così via.

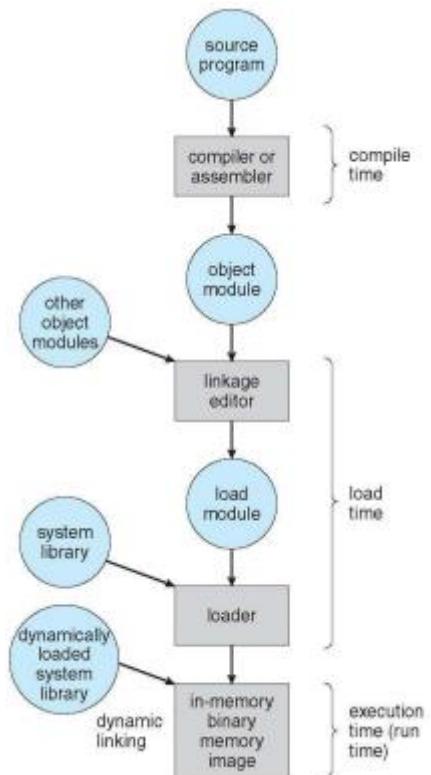
Binding di indirizzi

In genere un programma risiede in un disco in forma di un file binario eseguibile. Per essere eseguito, il programma va caricato in memoria e inserito all'interno 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*).

La procedura normale consiste nello scegliere uno dei processi appartenenti alla coda d'ingresso e nel caricarlo in memoria. Il processo durante l'esecuzione può accedere alle istruzioni e ai dati in memoria. Quando il processo termina, si dichiara disponibile il suo spazio di memoria.

La maggior parte dei sistemi consente ai processi utenti di risiedere in qualsiasi parte della memoria fisica, quindi, anche se lo spazio d'indirizzi del calcolatore comincia all'indirizzo 00000, il primo indirizzo del processo utente non deve necessariamente essere 00000. Quest'assetto influenza sugli indirizzi che un programma utente può usare. Nella maggior parte dei casi un programma utente, prima di essere eseguito, deve passare attraverso vari stadi, alcuni dei quali possono essere facoltativi (figura a destra), in cui gli indirizzi sono rappresentabili in modi



diversi. Generalmente gli indirizzi del programma sorgente sono simbolici (per esempio, contatore). Un compilatore di solito associa (***bind***) questi indirizzi simbolici a indirizzi rilocabili (per esempio, “14 byte dall’inizio di questo modulo”). L’editor dei collegamenti (*linkage editor*), o il caricatore (*loader*), fa corrispondere a sua volta questi indirizzi rilocabili a indirizzi assoluti (per esempio, 74014). Ogni binding rappresenta una corrispondenza da uno spazio d’indirizzi a un altro.

Generalmente, il binding di istruzioni e dati a indirizzi di memoria si può compiere in qualsiasi fase 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. I programmi per MS-DOS nel formato identificato dall’estensione .COM sono collegati al tempo di compilazione.
- **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 il binding finale degli indirizzi alla fase del caricamento. Se l’indirizzo iniziale cambia, è sufficiente ricaricare il codice utente per incorporare il valore modificato.
- **Esecuzione.** Se durante l’esecuzione il processo può essere spostato da un segmento di memoria a un altro, si deve ritardare il binding degli indirizzi fino alla fase d’esecuzione. Per realizzare questo schema sono necessarie specifiche caratteristiche dell’architettura; questo argomento è trattato nel sottoparagrafo successivo. La maggior parte dei sistemi operativi d’uso generale impiega questo metodo.

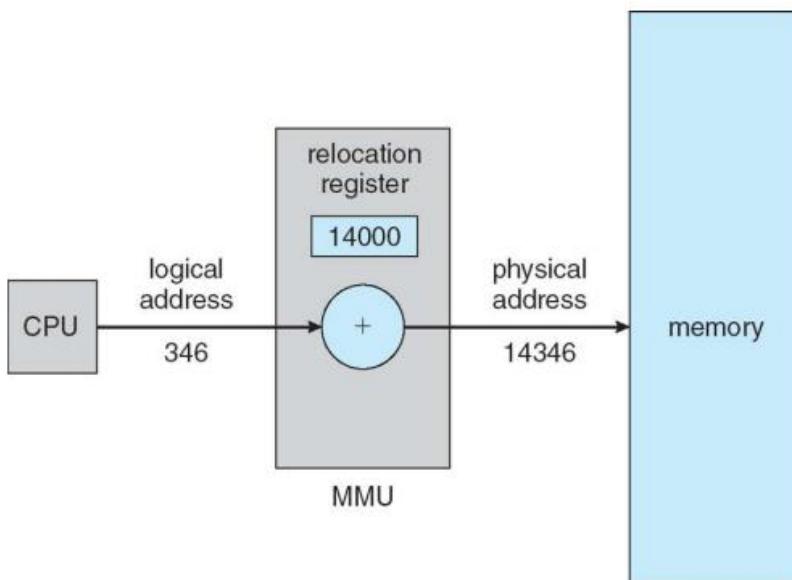
Una gran parte di questo capitolo è dedicata alla spiegazione di come i vari tipi di binding degli indirizzi si possano realizzare efficacemente in un calcolatore e, inoltre, alla discussione delle caratteristiche dell’architettura appropriate alla realizzazione di queste funzioni.

Spazi di indirizzi logici e fisici a confronto

Un indirizzo generato dalla CPU di solito si indica come **indirizzo logico**, mentre un indirizzo visto dall’unità di memoria, cioè caricato nel **registro dell’indirizzo di memoria** (*memory address register*) di solito si indica come **indirizzo fisico**.

I metodi di binding degli indirizzi nelle fasi di compilazione e di caricamento producono indirizzi logici e fisici identici. Con i metodi 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; in questo testo si usano tali termini in modo intercambiabile. 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.

Il binding nella fase d’esecuzione dagli indirizzi virtuali agli indirizzi fisici è svolto da un dispositivo detto **memory-management unit** (unità di gestione della memoria), in breve **MMU**. Com’è illustrato nella figura della pagina seguente, 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**.



Ad 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 corrisponde alla locazione 14346. Quando il sistema operativo MS-DOS, eseguito sulla famiglia di CPU Intel 80x86, carica ed esegue processi impiega quattro registri di rilocazione.

Il programma utente non considera mai gli indirizzi fisici reali. Il programma crea un puntatore alla locazione 346, lo memorizza, lo modifica, lo confronta con altri indirizzi, tutto ciò semplicemente come un numero. 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.

In questo caso esistono due diversi tipi di indirizzi: gli indirizzi logici (intervallo $[0, \text{max}]$) e gli indirizzi fisici ($[r + 0, r + \text{max}]$ con r valore di base). L'utente genera solo indirizzi logici e "pensa" che il processo sia eseguito nelle posizioni da 0 a max. Il programma utente fornisce indirizzi logici che, prima d'essere usati, si devono far corrispondere a indirizzi fisici.

Il concetto di spazio d'indirizzi logici associato a uno spazio d'indirizzi fisici separato è fondamentale per una corretta gestione della memoria.

Caricamento dinamico

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 in memoria secondaria in un formato di caricamento rilocabile. Si carica il programma principale in memoria e quando, durante l'esecuzione, una procedura deve richiamarne un'altra, si controlla innanzitutto che sia stata caricata, altrimenti si richiama il caricatore di collegamento 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 che non si adopera non viene caricata. 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 intervento 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.

Linking dinamico e librerie condivise

Alcuni sistemi operativi consentono solo il **linking statico**, in cui le librerie di sistema del linguaggio sono trattate come qualsiasi altro modulo oggetto e combinare dal loader (caricatore) nell'immagine binaria del programma in formato eseguibile. Il concetto di linking dinamico è analogo a quello di caricamento dinamico. Invece di differire il caricamento di una procedura fino al momento dell'esecuzione, si differisce il linking. Questa caratteristica si usa soprattutto con le librerie di sistema, per esempio le librerie di procedure del linguaggio. Senza questo strumento tutti i programmi di un sistema dovrebbero disporre all'interno dell'immagine eseguibile di una copia della libreria di linguaggio (o almeno delle procedure cui il programma fa riferimento). Tutto ciò richiede spazio nei dischi e in memoria centrale.

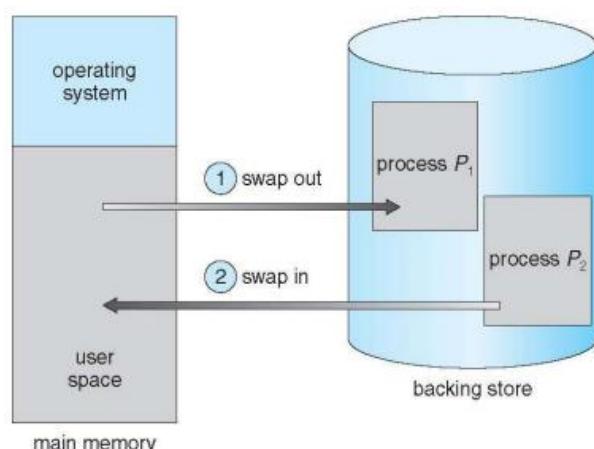
Con il linking dinamico, invece, per ogni riferimento a una procedura di libreria s'inserisce all'interno dell'immagine eseguibile una piccola porzione di codice di riferimento (*stub*), che indica come localizzare la giusta procedura di libreria residente in memoria o come caricare la libreria se la procedura non è già presente. Durante l'esecuzione, il codice di riferimento controlla se la procedura richiesta è già in memoria, altrimenti provvede a cariarla; in ogni caso tale codice sostituisce se stesso con l'indirizzo della procedura, che viene poi eseguita. In questo modo, quando si raggiunge nuovamente quel segmento del codice, si esegue direttamente la procedura di libreria, senza costi aggiuntivi per il linking dinamico. Con questo metodo tutti i processi che usano una libreria del linguaggio si limitano a eseguire la stessa copia del codice della libreria.

Questa caratteristica si può estendere anche agli aggiornamenti delle librerie, per esempio, la correzione di errori. Una libreria si può sostituire con una nuova versione, e tutti i programmi che fanno riferimento a quella libreria usano automaticamente quella. Senza il linking dinamico tutti i programmi di questo tipo devono subire una nuova fase di linking per accedere alla nuova libreria. Affinché i programmi non eseguano accidentalmente nuove versioni di librerie incompatibili, sia nel programma sia nella libreria si inserisce un'informazione relativa alla versione. È possibile caricare in memoria più di una versione della stessa libreria, ciascun programma si serve dell'informazione sulla versione per decidere quale copia debba usare. Se le modifiche sono di piccola entità, il numero di versione resta invariato; se l'entità delle modifiche diviene rilevante, si aumenta anche il numero di versione. Perciò, solo i programmi linkati con la nuova versione della libreria subiscono gli effetti delle modifiche incompatibili incorporate nella libreria stessa. I programmi collegati prima dell'installazione della nuova libreria continuano ad avvalersi della vecchia libreria. Questo sistema è noto anche con il nome di librerie condivise.

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.

Swapping

Per essere eseguito, un processo deve trovarsi in memoria centrale, ma si può trasferire temporaneamente in **memoria ausiliaria** (backing store) da cui si riporta in memoria centrale al momento di riprenderne l'esecuzione. Si consideri, per esempio, un ambiente di multiprogrammazione con un algoritmo circolare (round-robin) per lo scheduling della CPU. Trascorso un quanto di tempo, il gestore di memoria scarica dalla memoria il processo appena terminato e carica un altro processo nello spazio di memoria appena liberato;



questo procedimento è illustrato nella figura e si chiama **swapping** (avvicendamento dei processi in memoria, in breve avvicendamento o scambio). Nel frattempo lo scheduler della CPU assegna un quanto di tempo a un altro processo presente in memoria. Quando esaurisce il suo quanto di tempo, ciascun processo viene scambiato con un altro processo. In teoria il gestore della memoria può swappare i processi in modo sufficientemente rapido da far sì che alcuni siano presenti in memoria, pronti per essere eseguiti, quando lo scheduler della CPU voglia riassegnare la CPU stessa. Anche il quanto di tempo deve essere sufficientemente lungo da permettere che un processo, prima d'essere sostituito, esegua quantità ragionevole di calcolo.

Una variante di questo criterio di swapping dei processi s'impiega per gli algoritmi di scheduling basati sulle priorità. Se si presenta un processo con priorità maggiore, il gestore della memoria può scaricare dalla memoria centrale il processo con priorità inferiore per fare spazio all'esecuzione del processo con priorità maggiore. Quando il processo con priorità maggiore termina, si può ricaricare in memoria quello con priorità minore e continuare la sua esecuzione (questo tipo di swapping è talvolta chiamato **roll out, roll in**).

Normalmente, un processo che è stato scaricato dalla memoria si deve ricaricare nello spazio di memoria occupato in precedenza. Questa limitazione è dovuta al metodo di associazione degli indirizzi. Se l'associazione degli indirizzi logici agli indirizzi fisici della memoria si effettua nella fase di assemblaggio o di caricamento, il processo non può essere ricaricato in posizioni diverse. Se l'associazione degli indirizzi logici agli indirizzi fisici della memoria si compie nella fase d'esecuzione, un processo può essere riversato in uno spazio di memoria diverso, poiché gli indirizzi fisici si calcolano nella fase d'esecuzione.

Lo swapping dei processi richiede una memoria ausiliaria. Tale memoria ausiliaria deve essere abbastanza ampia da contenere le copie di tutte le immagini di memoria di tutti i processi utenti, e deve permettere un accesso diretto a dette immagini di memoria. Il sistema mantiene una **ready queue** (coda dei processi pronti) formata da tutti i processi pronti per l'esecuzione, le cui immagini di memoria si trovano in memoria ausiliaria o in memoria. Quando lo scheduler della CPU decide di eseguire un processo, richiama il dispatcher, che controlla se il primo processo della coda si trova in memoria. Se non si trova in memoria, e in questa non c'è spazio libero, il dispatcher scarica un processo dalla memoria e vi carica il processo richiesto dallo scheduler della CPU, quindi ricarica normalmente i registri e trasferisce il controllo al processo selezionato.

In un siffatto sistema di swapping, il tempo di context switch (cambio di contesto) è abbastanza elevato.

Occorre notare che la maggior parte del tempo di swapping è data dal tempo di trasferimento. Il tempo di trasferimento totale è direttamente proporzionale alla quantità di memoria interessata. In un calcolatore con 4 GB di memoria centrale e un sistema operativo residente di 1 GB massima dimensione possibile per un processo utente è di 3 GB. Tuttavia possono essere presenti molti processi utenti con dimensione minore, per esempio 100 MB. Lo scaricamento di un processo di 100 MB può concludersi in 2 secondi, mentre per lo scaricamento di 3 GB sono necessari 60 secondi. Perciò sarebbe utile sapere esattamente quanta memoria sia effettivamente usata da un processo utente e non solo quanta questo potrebbe teoricamente usarne, poiché in questo caso è necessario scaricare solo quanto è effettivamente utilizzato, riducendo il tempo di swapping. Affinché questo metodo risulti efficace, l'utente deve tenere informato il sistema su tutte le modifiche apportate ai requisiti di memoria; un processo con requisiti di memoria dinamici deve impiegare chiamate di sistema (request memory e release memory) per informare il sistema operativo delle modifiche da apportare alla memoria.

Allocazione contigua della memoria

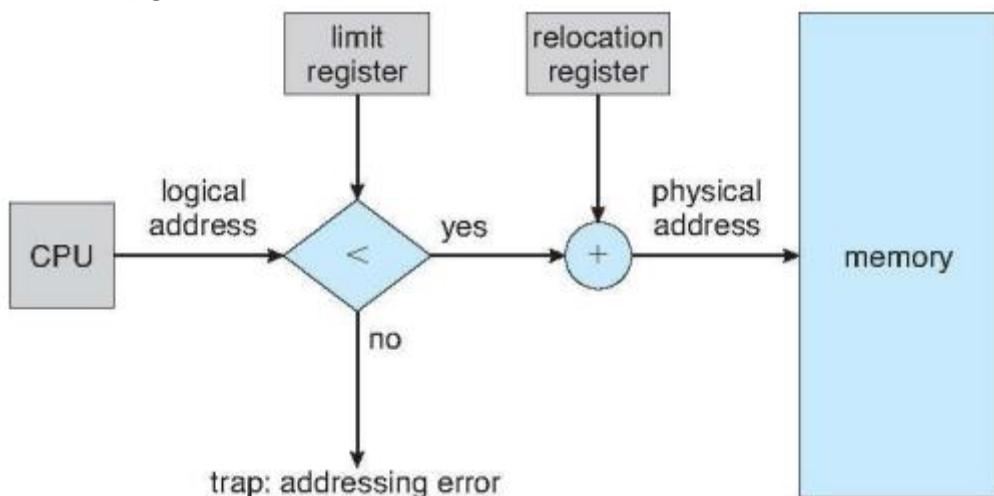
La memoria centrale deve contenere sia il sistema operativo sia i vari processi utenti; perciò, è necessario assegnare le diverse parti della memoria centrale nel modo più efficiente. In questo paragrafo si tratta l'allocazione contigua della memoria.

La memoria centrale di solito si divide in due partizioni, una per il sistema operativo residente e una per i processi utenti. Il sistema operativo si può collocare sia in memoria bassa sia in memoria alta. Il fattore che incide in modo decisivo su tale scelta è generalmente la posizione del vettore delle interruzioni. Poiché si trova spesso in memoria bassa, i programmati collocano di solito anche il sistema operativo in memoria bassa. Per questo motivo 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 contigua di memoria.

Rilocazione e protezione della memoria

La protezione della memoria si può realizzare usando un registro di rilocazione (descritto in “[Spazi di indirizzi logici e fisici a confronto](#)”) con un registro limite (descritto in “[Dispositivi essenziali](#)”). Il registro di rilocazione contiene il valore dell'indirizzo fisico minore; il registro limite contiene l'intervallo di indirizzi logici, per esempio *rilocazione* = 100040 e *limite* = 74.600. Con i registri di rilocazione e limite, ogni indirizzo logico deve essere minore del contenuto del registro limite; la MMU fa corrispondere dinamicamente l'indirizzo fisico all'indirizzo logico sommando a quest'ultimo il valore contenuto nel registro di rilocazione (figura successiva).



Quando lo scheduler della CPU seleziona un processo per l'esecuzione, il dispatcher, durante l'esecuzione del cambio di contesto, carica il registro di rilocazione e il registro limite con i valori corretti. Poiché si confronta ogni indirizzo generato dalla CPU con i valori contenuti in questi registri, si possono proteggere il sistema operativo, i programmi e i dati di altri utenti da tentativi di modifiche da parte del processo in esecuzione.

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.

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. Con il 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 (metodo, detto mft, ormai in disuso). Il metodo descritto di seguito, detto mvt, è una generalizzazione del metodo con partizioni fisse e si usa soprattutto in ambienti d'elaborazione a batch.

Nello schema a partizione fissa 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 **hole** (buco). La memoria contiene una serie di hole 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 ogni dato istante è sempre disponibile una lista delle dimensioni dei blocchi liberi e della coda d'ingresso. Il sistema operativo può ordinare la coda d'ingresso secondo un algoritmo di scheduling. La memoria si assegna ai processi della coda finché si possono soddisfare i requisiti di memoria del processo successivo, cioè finché esiste un blocco di memoria (o hole) disponibile, sufficientemente grande da accogliere quel processo. Il sistema operativo può quindi attendere che si renda disponibile un blocco sufficientemente grande, oppure può scorrere la coda d'ingresso per verificare se sia possibile soddisfare le richieste di memoria più limitate di qualche altro processo.

In generale, è sempre presente un insieme di hole di diverse dimensioni sparsi per la memoria. Quando si presenta un processo che necessita di memoria, il sistema cerca nel gruppo un hole di dimensioni sufficienti per contenerlo. Se è troppo grande, l'hole viene diviso in due parti: si assegna una parte al processo in arrivo e si riporta l'altra nell'insieme degli hole. Quando termina, un processo rilascia il blocco di memoria, che si reinserisce nell'insieme degli hole; se si trova accanto ad altri hole, si uniscono tutti gli hole adiacenti per formarne uno più grande. A questo punto il sistema deve 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 hole liberi. Le soluzioni sono numerose. I criteri più usati per scegliere un hole libero tra quelli disponibili nell'insieme sono i seguenti (sono anche tipiche domande orali).

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

Con Fuso 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.

Frammentazione

Entrambi i criteri first-fit e best-fit di allocazione della memoria soffrono di **frammentazione esterna**: quando si caricano e si rimuovono 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 hole. 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 esterna è un problema. La sua gravità dipende dalla quantità totale di memoria e dalla dimensione media dei processi. L'analisi statistica dell'algoritmo che segue il criterio di scelta first-fit, per esempio, rileva che, pur con una certa ottimizzazione, per n blocchi assegnati, si perdono altri $0,5n$ blocchi a causa della frammentazione, ciò significa che potrebbe essere inutilizzabile un terzo della memoria. Questa caratteristica è nota con il nome di **50 percent rule** (regola del 50 per cento).

La frammentazione può essere sia interna sia esterna. Si consideri il metodo d'allocazione con più partizioni con un buco di 18.464 byte. Supponendo che il processo successivo richieda 18.462 byte, assegnando esattamente il blocco richiesto rimane un hole di 2 byte. Il carico necessario per tener traccia di questo hole è sostanzialmente più grande dell'hole stesso. Il metodo generale prevede di suddividere la memoria fisica in blocchi di dimensione fissa, che costituiscono le unità d'allocazione. Con questo metodo la memoria assegnata può essere leggermente maggiore della memoria richiesta. La **frammentazione interna** consiste nella differenza tra questi due numeri; la memoria è interna a una partizione, ma non è in uso.

N.B.: Concetti di frammentazione esterna e interna sono domande orali molto frequenti.

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 fatta nella fase di assemblaggio o di caricamento; è possibile solo se la rilocazione è dinamica e si compie nella fase d'esecuzione. Se gli indirizzi sono rilocati dinamicamente, la rilocazione richiede solo lo spostamento del programma e dei dati, e quindi la modifica del registro di rilocazione in modo che rifletta il nuovo indirizzo di base. Quando è possibile eseguire la compattazione, è necessario determinarne il costo. Il più semplice algoritmo di compattazione consiste nello spostare tutti i processi verso un'estremità della memoria, mentre tutti gli hole vengono spostati nell'altra direzione formando un grosso hole di memoria. Questo metodo può essere assai oneroso.

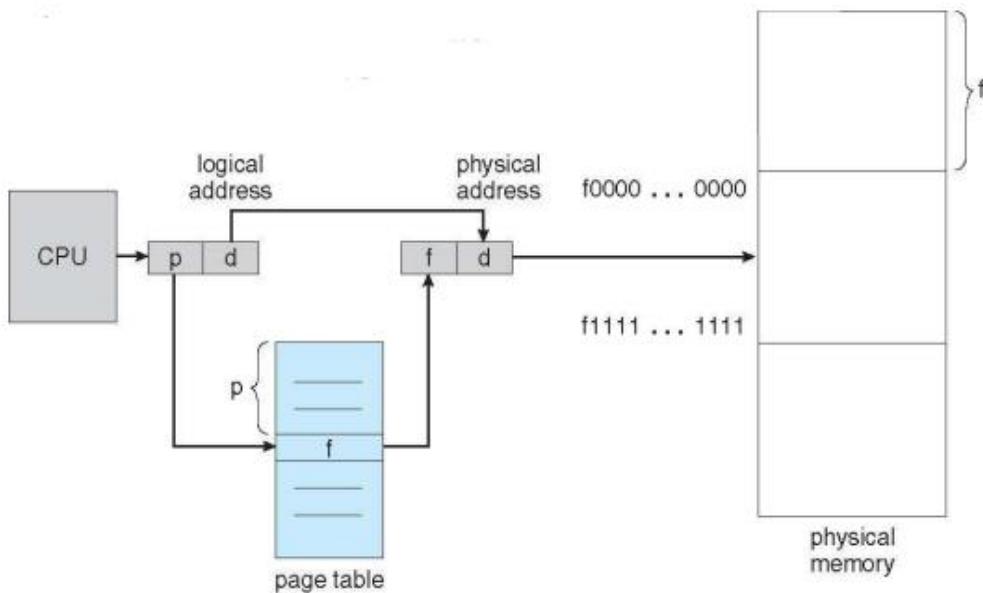
Paging

Il paging (paginazione) è un metodo di gestione della memoria che permette che lo spazio degli indirizzi fisici di un processo non sia contiguo. Elimina il gravoso problema della sistemazione di blocchi di memoria di diverse dimensioni in memoria ausiliaria, una questione che riguarda la maggior parte dei metodi di gestione della memoria analizzati.

Metodo di base

Il metodo di base per implementare il paging consiste nel suddividere la memoria fisica in blocchi di dimensione costante, detti anche **frame** (pagine fisiche), e nel suddividere la memoria logica in blocchi di pari dimensione, detti **page** (pagine). Quando si deve eseguire un processo, si caricano le sue page nei frame disponibili, prendendole dalla memoria ausiliaria, divisa in blocchi di dimensione fissa, uguale a quella dei frame della memoria.

L'architettura d'ausilio al paging è illustrata nella seguente figura; ogni indirizzo generato dalla CPU è diviso in due parti: **page number** (p), e un **page offset** (d , offset = di quanto mi devo spostare).



p serve come indice per la **page table**, contenente l'indirizzo di base in memoria fisica di ogni page. 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. La figura a destra illustra il modello di paginazione della memoria.

La dimensione di una page, così come quella di un frame, è definita dall'architettura del calcolatore ed è, in genere, una potenza di 2 compresa tra 512byte e 16MB. La scelta di una potenza di 2 come dimensione della page facilita notevolmente la traduzione di un indirizzo logico nei corrispondenti p e d .

Se la dimensione dello spazio degli indirizzi logici è 2^m e la dimensione di una page è di 2^n unità di indirizzamento (byte o parole), allora gli $m - n$ bit più significativi di un indirizzo logico indicano il page number, e gli n bit meno significativi indicano il page offset. L'indirizzo logico ha quindi la seguente forma:

- dove p è un indice della page table e d è l'offset all'interno della page indicata da p

The diagram illustrates the mapping between logical memory and physical memory through a page table.

logical memory:

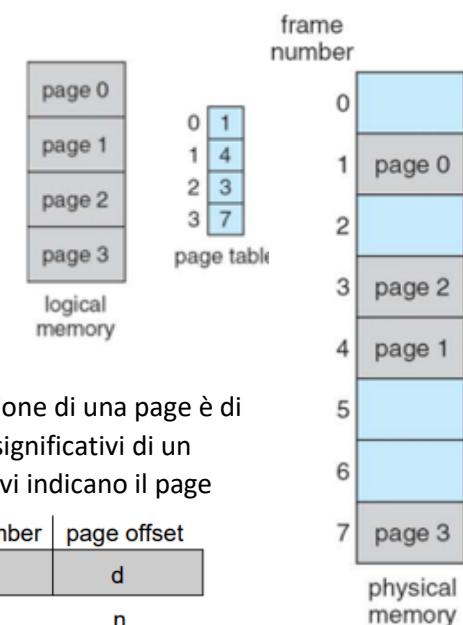
0	a
1	b
2	c
3	d
4	e
5	f
6	g
7	h
8	i
9	j
10	k
11	l
12	m
13	n
14	o
15	p

page table:

0	5
1	6
2	1
3	2

physical memory:

0	
4	i
5	—
6	k
8	m
9	n
10	o
12	p
16	
20	a
21	b
22	c
23	d
24	e
25	f
26	g
28	h



Come esempio concreto si consideri la memoria illustrata nella figura a destra; con page di 4 byte e una memoria fisica di 32byte (8 pagine), si mostra come si faccia corrispondere la memoria vista dall'utente alla memoria fisica. L'indirizzo logico 0 è la page 0 con offset 0. Secondo la page table, la page 0 si trova nel frame 5. Quindi all'indirizzo logico 0 corrisponde l'indirizzo fisico $20 (= (5 * 4) + 0)$. All'indirizzo logico 3 (page 0, offset 3) corrisponde l'indirizzo fisico $23 (= (5 * 4) + 3)$. Per quel che riguarda l'indirizzo logico 4 (page 1, offset 0), secondo la page table, alla page 1 corrisponde il frame 6, quindi, all'indirizzo logico 4 corrisponde l'indirizzo fisico $24 (= (6 * 4) + 0)$. All'indirizzo logico 13 corrisponde l'indirizzo fisico 9.

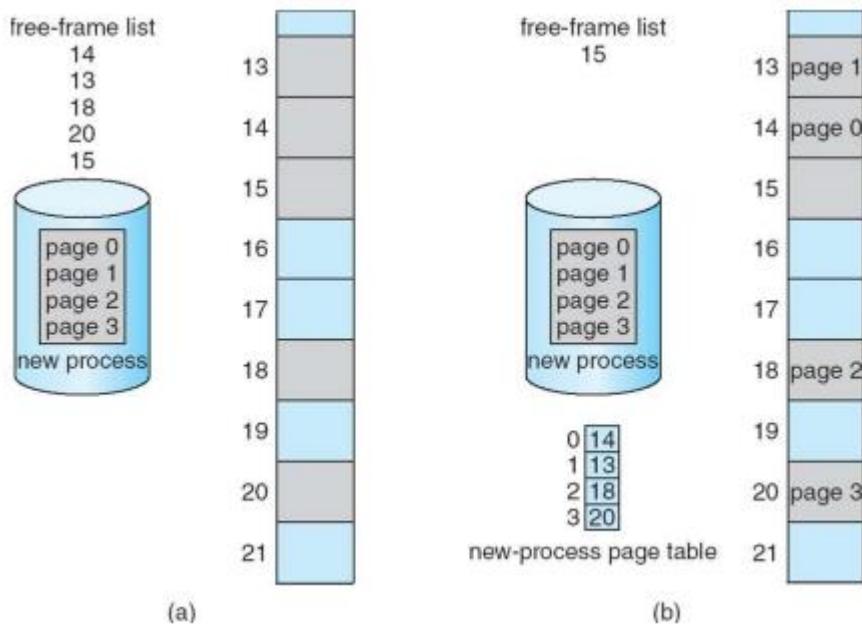
Si può notare che il paging non è altro che una forma di rilocazione dinamica: a ogni indirizzo logico l'architettura di paging fa corrispondere un indirizzo fisico. L'uso della page table è simile all'uso di una tabella di

registri base (o di rilocazione), uno per ciascun frame.

Con il paging si può evitare la frammentazione esterna (si ha quindi solo quella interna): qualsiasi frame libero si può assegnare a un processo che ne abbia bisogno; tuttavia si può avere la frammentazione interna. I frame si assegnano come unità. Poiché in generale lo spazio di memoria richiesto da un processo non è un multiplo delle dimensioni delle page, l'ultimo frame assegnato può non essere completamente pieno. Se, per esempio, le page sono di 2048 byte, un processo di 72766 byte necessita di 35 page più 1086 byte. Si assegnano 36 frame, quindi si ha una frammentazione interna di $2048 - 1086 = 962$ byte. Il caso peggiore si ha con un processo che necessita di n page più un byte: si assegnano $n + 1$ frame, quindi si ha una frammentazione interna di quasi un intero frame.

Se la dimensione del processo è indipendente dalla dimensione della page, ci si deve aspettare una frammentazione interna media di mezza page per processo. Questa considerazione suggerisce che conviene usare page di piccole dimensioni; tuttavia, a ogni elemento della page table è associato un carico che si può ridurre aumentando le dimensioni delle page. Generalmente la dimensione delle page cresce col passare del tempo, come i processi, gli insiemi di dati e la memoria centrale; attualmente la dimensione tipica delle page è compresa tra 4 KB e 8 KB; in alcuni sistemi può essere anche maggiore.

Quando si deve eseguire un processo, si esamina la sua dimensione espressa in page. Poiché ogni page del processo necessita di un frame, se il processo richiede n page, devono essere disponibili almeno n frame che, se ci sono, si assegnano al processo stesso. Si carica la prima page del processo in uno dei frame assegnati e s'inserisce il numero del frame nella page table relativa al processo in questione. La page successiva si carica in un altro frame e, anche in questo caso, s'inserisce il numero del frame nella page table e così via (figura in basso: (a) prima l'allocazione, (b) dopo l'allocazione).



Un aspetto importante della paginazione è la netta distinzione tra la memoria vista dall'utente e l'effettiva memoria fisica: il programma utente vede la memoria come un unico spazio contiguo, contenente solo il programma stesso; in realtà, il programma utente è sparso in una memoria fisica contenente anche altri programmi. La differenza tra la memoria vista dall'utente e la memoria fisica è colmata dall'architettura di traduzione degli indirizzi, che fa corrispondere gli indirizzi fisici agli indirizzi logici generati dai processi utenti.

Poiché il sistema operativo gestisce la memoria fisica, deve essere informato dei relativi particolari di 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 blocchi (o 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.

Architettura di paging

Ogni sistema operativo segue metodi propri per memorizzare le tabelle delle pagine. La maggior parte dei sistemi impiega una tabella delle pagine per ciascun processo. Il PCB contiene un puntatore alla page table.

L'architettura d'ausilio alla 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 paging, questi registri devono essere costruiti in modo da operare a una velocità molto elevata. Tale efficienza è determinante, poiché ogni accesso alla memoria passa attraverso il sistema di paging. Il dispatcher della CPU ricarica questi registri proprio come ricarica gli altri, ma le istruzioni di caricamento e modifica dei registri della tabella delle pagine sono privilegiate, quindi soltanto il sistema operativo può modificare la mappa della memoria.

L'uso di registri per la tabella delle pagine è efficiente se la tabella stessa è ragionevolmente piccola, nell'ordine, per esempio, di 256 entry (elementi). La maggior parte dei calcolatori contemporanei usa comunque tabelle molto grandi, per esempio di un milione (2^{20}) di entry, quindi non si possono impiegare i registri veloci per realizzare la tabella delle pagine; quest'ultima si mantiene in memoria principale e un **page-table base register** (PTBR, registro di base della tabella delle pagine) punta alla tabella stessa. Il cambio delle page table richiede soltanto di modificare questo registro, riducendo considerevolmente il tempo dei context switch.

Questo metodo presenta un problema connesso al tempo necessario di accesso a una locazione della memoria utente. Per accedere alla locazione i , occorre far riferimento alla tabella delle pagine usando il valore contenuto nel PTBR aumentato del numero di pagina relativo a i , perciò si deve accedere alla memoria. Si ottiene il numero del frame che, associato all'offset di pagina, produce l'indirizzo cercato; a questo punto è possibile accedere alla posizione di memoria desiderata. Con questo metodo, per accedere a un byte occorrono due accessi alla memoria (uno per l'entry table e uno per il byte stesso), quindi l'accesso alla memoria è rallentato di un fattore 2 (ritardo nella maggior parte dei casi intollerabile).

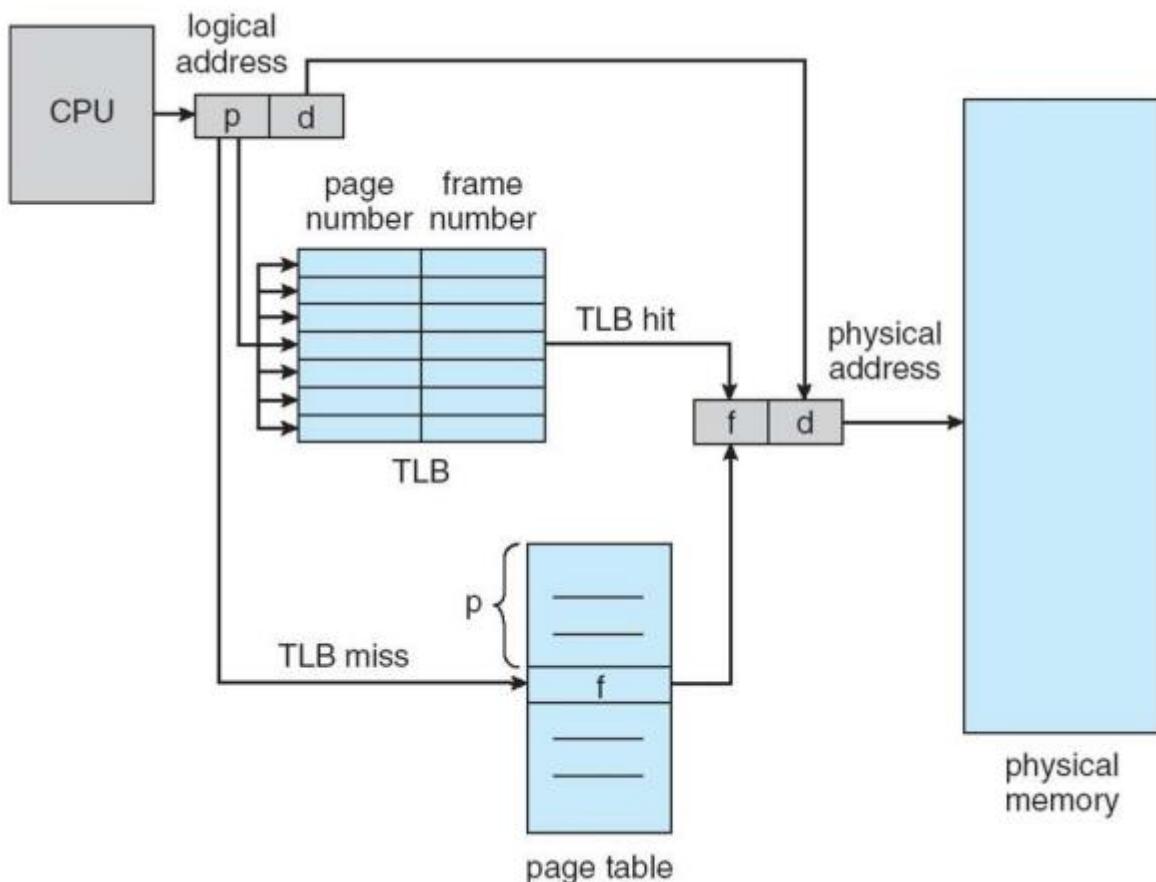
La soluzione tipica a questo problema consiste nell'impiego di una speciale, piccola cache di ricerca veloce, detta **TLB** (*translation look-aside buffer*). La TLB è una memoria associativa ad alta velocità in cui ogni sua entry consiste in due parti: una chiave, o un indicatore (**tag**) e un valore. Quando si presenta una entry, la memoria associativa lo confronta contemporaneamente con tutte le chiavi; se trova una corrispondenza, riporta il valore correlato. La ricerca è molto rapida, ma le memorie associative sono molto costose. Il numero degli elementi in una TLB è piccolo, spesso è compreso tra 2^6 (= 64) e 2^{10} (= 1024).

La TBL si usa insieme con le page table nel modo seguente: la TLB contiene una piccola parte degli entry della page table; 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 **TLB miss**, si deve consultare la page table in memoria. Il numero del frame così ottenuto si può eventualmente usare per accedere alla memoria (vedi figura nella pagina seguente). Inoltre, inserendo i numeri della pagina e del frame nella TLB, al riferimento successivo la ricerca sarà molto più rapida.

N.B: Cosa succede con paging TLB è una tipica domanda d'esame (devi spiegare la figura successiva).

Alcune TLB memorizzano gli **address-space identifier** (ASID, identificatori dello spazio d'indirizzi) in ciascuna entry della TLB. Un ASID identifica in modo univoco ciascun processo e si usa per fornire al processo corrispondente la protezione del suo spazio d'indirizzi. Quando tenta di trovare i valori corrispondenti ai numeri delle pagine virtuali, la TLB assicura che l'ASID per il processo attualmente in esecuzione

corrisponda all'ASID associato alla pagina virtuale. La mancata corrispondenza dell'ASID viene trattata come una TLB miss.



Il tempo di ricerca in TLB può essere minore del 10% del tempo di accesso in memoria. La percentuale di volte che un numero di pagina si trova nella TLB è detta **hit ratio** (tasso di successo). Una hit ratio dell'80% significa che il numero di pagina desiderato si trova nella TLB nell'80% dei casi. Se la ricerca nella TLB richiede 20nanosecondi e sono necessari 100nanosecondi per accedere alla memoria, allora, supponendo che il numero di pagina si trovi nella TLB, un accesso alla memoria richiede 120nanosecondi. Se, invece, il numero non è contenuto nella TLB (20 nanosecondi), occorre accedere alla memoria per arrivare alla page table e al numero del frame (100nanosecondi), quindi accedere al byte desiderato in memoria (100nanosecondi); in totale sono necessari 220nanosecondi. Per calcolare il **tempo effettivo di accesso** (può essere domanda d'esame) occorre tener conto della probabilità dei due casi:

$$EAT = 120 * 0,80 + 220 * 0,20 = 140\text{nanosecondi}$$

In simboli, sia ε l'unità di tempo per la ricerca in TLB, sia α = hit ratio e sia T_a l'unità di tempo per un accesso in memoria; l'EAT (effective access time) si definisce come segue:

$$EAT = (T_a + \varepsilon)\alpha + (2T_a + \varepsilon)(1 - \alpha)$$

Esercizio svolto (è esempio d'esercizio d'esame):

- Sia dato un sistema di paginazione con una tabella delle pagine che risiede in memoria con un TLB con un hit ratio del 90%. Se per un accesso in memoria occorrono 200nanosecondi, quanto tempo occorrerà per ottenere il dato relativo a un indirizzo logico? Si supponga trascurabile il tempo di ricerca della entry nella page table.
 - Caso hit: tempo di accesso 200ns
 - Caso miss: tempo di accesso 400ns

- $EAT = 0.9 * 200 + 0.1 * 400 = 220\text{ns}$

Protezione

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.

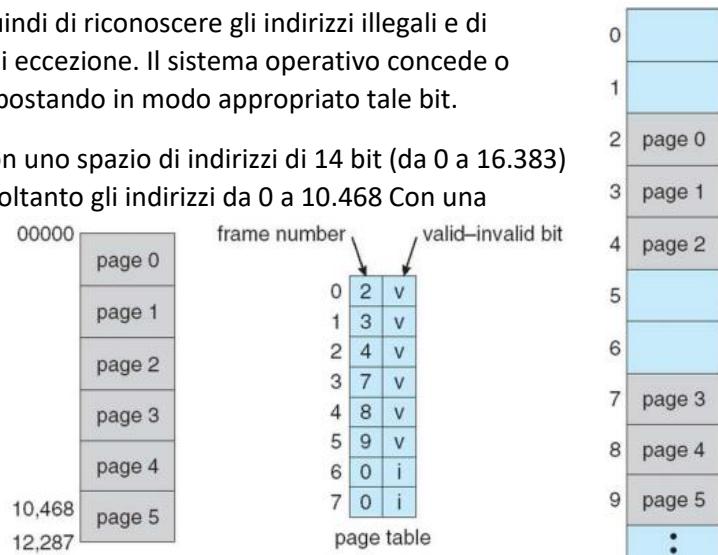
Un bit può determinare se una pagina si può leggere e scrivere oppure soltanto leggere. Tutti i riferimenti alla memoria passano attraverso la tabella delle pagine per trovare il numero corretto del frame; quindi, mentre si calcola l'indirizzo fisico, si possono controllare i bit di protezione per verificare che non si scriva in una pagina di sola lettura. Un tale tentativo causerebbe l'invio di un segnale di eccezione al sistema operativo, si avrebbe cioè una violazione della protezione della memoria.

Di solito si associa a ciascun elemento della tabella delle pagine un ulteriore bit, detto **valid-invalid bit** (bit di validità). Tale bit, impostato a *valid*, indica che la pagina corrispondente è nello spazio d'indirizzi logici del processo, quindi è una pagina valida; impostato a *invalid*, 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 segnale di eccezione. Il sistema operativo concede o revoca la possibilità d'accesso a una pagina impostando in modo appropriato tale bit.

Per esempio, supponiamo che in un sistema con uno spazio di indirizzi di 14 bit (da 0 a 16.383) si possa avere un programma che deve usare soltanto gli indirizzi da 0 a 10.468. Con una

dimensione delle pagine di 2 KB si ha la situazione mostrata nella figura. 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 segnale di eccezione al sistema operativo (riferimento di pagina non valido).



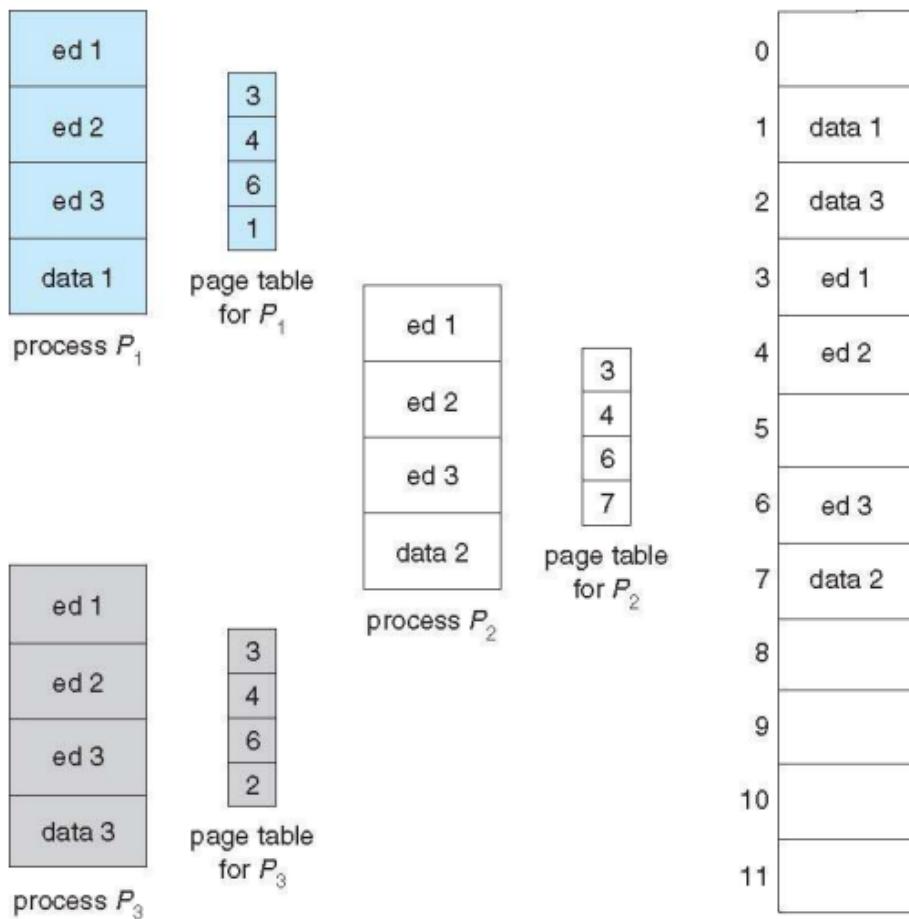
Questo schema ha creato un problema: poiché il programma si estende solo fino all'indirizzo 10.468, ogni riferimento oltre tale indirizzo è illegale; i riferimenti alla pagina 5 sono tuttavia classificati come validi, e ciò rende validi gli accessi sino all'indirizzo 12.287; solo gli indirizzi da 12.288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

Capita raramente che un processo faccia uso di tutto il suo intervallo di indirizzi, infatti molti processi utilizzano solo una piccola frazione dello spazio d'indirizzi di cui dispongono. In questi casi è un inutile spreco creare una tabella di pagine con elementi per ogni pagina dell'intervallo di indirizzi, poiché una gran parte di questa tabella resta inutilizzata e occupa prezioso spazio di memoria. Alcune architetture dispongono di registri, detti **page-table length register** (PTLR, registri di lunghezza della tabella delle pagine), per indicare le dimensioni della tabella. Questo valore si controlla rispetto a ogni indirizzo logico per verificare che quest'ultimo si trovi nell'intervallo valido per il processo. Un errore causa l'invio di un segnale di eccezione al sistema operativo.

Pagine condivise

Un altro vantaggio del paging consiste nella possibilità di condividere codice comune (particolarmente vantaggioso in architetture multiutente).

Se il codice è rientrante, può essere condiviso, come mostra la figura a pagina seguente: un elaboratore di testi, di tre pagine di 50KB ciascuna (l'ampia dimensione delle pagine ha lo scopo di rendere più chiara la rappresentazione), condiviso da tre processi, ciascuno dei quali dispone della propria pagina di dati.



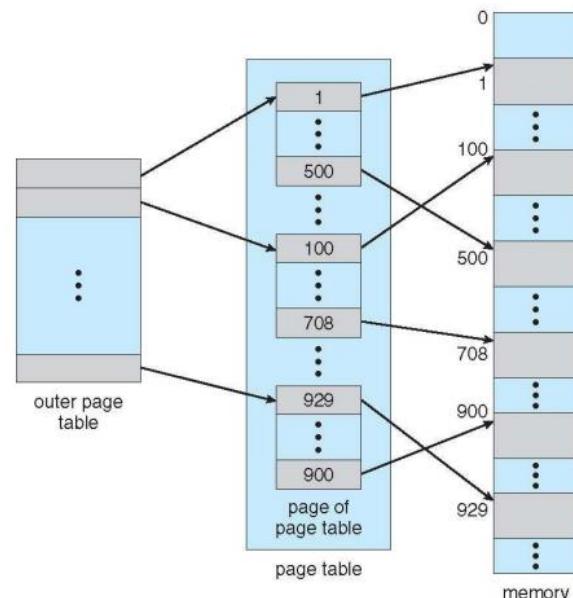
Il **codice rientrante**, detto anche codice puro, è un codice non automodificante: non cambia durante l'esecuzione. Quindi, due o più processi possono eseguire lo stesso codice nello stesso momento. Ciascun processo dispone di una propria copia dei registri e di una memoria dove conserva i dati necessari alla propria esecuzione. I dati per due differenti processi variano, ovviamente, per ciascun processo.

Struttura della tabella delle pagine

In questo paragrafo si descrivono alcune tra le tecniche più comuni per strutturare la tabella delle pagine.

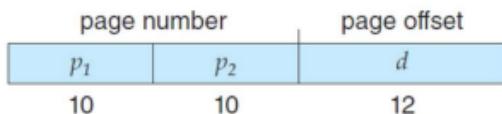
Paginazione gerarchica

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 finirebbe per diventare eccessivamente grande. Si consideri, per esempio, un sistema con uno spazio d'indirizzi logici a 32 bit. Se la dimensione di ciascuna pagina è di 4 KB (2^{12}), la tabella delle pagine potrebbe arrivare a contenere fino a 1 milione di elementi ($2^{32}/2^{12}$). Se ogni elemento consiste di 4 byte, ciascun processo potrebbe richiedere fino a 4 MB di spazio fisico d'indirizzi solo per la tabella delle pagine. Chiaramente, sarebbe

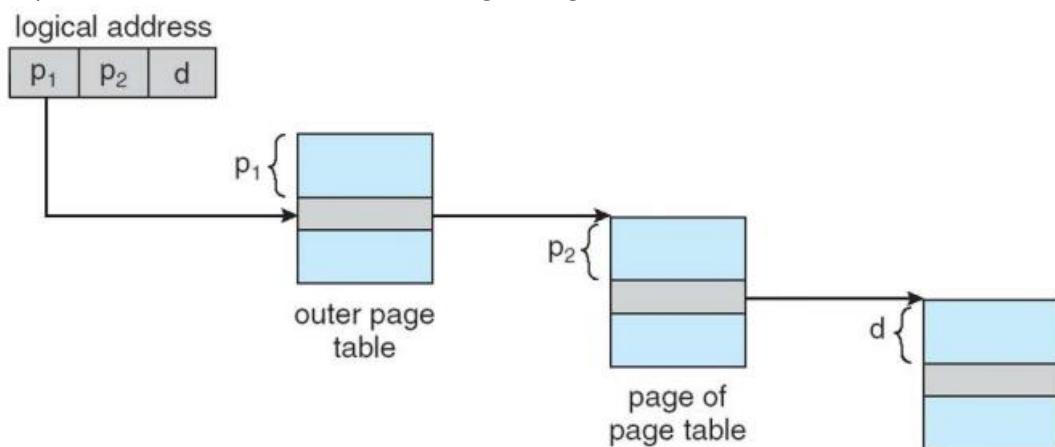


meglio evitare di collocare la tabella delle pagine in modo contiguo in memoria centrale. Una semplice soluzione a questo problema consiste nel suddividere la tabella delle pagine in parti più piccole; questo risultato si può ottenere in molti modi.

Un metodo consiste nell'adottare un algoritmo di paginazione a due livelli, in cui la tabella stessa è paginata (figura precedente). Si consideri il precedente esempio di macchina a 32 bit con dimensione delle pagine di 4 KB. Si suddivide ciascun indirizzo logico 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:

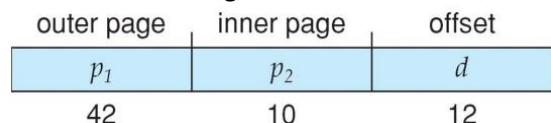


dove p_1 è un indice della tabella delle pagine di primo livello, o tabella esterna delle pagine, e p_2 è l'offset all'interno della pagina indicata dalla tabella esterna delle pagine. Il metodo di traduzione degli indirizzi seguito da questa architettura è mostrato nella figura seguente:



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

Lo schema di paginazione a due livelli non è più adatto nel caso di sistemi con uno spazio di indirizzi logici a 64 bit. Per illustrare questo aspetto, si supponga che la dimensione delle pagine di questo sistema sia di 4KB (2^{12}). In questo caso, la tabella delle pagine conterrà fino a 2^{52} elementi. Adottando uno schema di paginazione a due livelli, le tabelle interne delle pagine possono occupare convenientemente una pagina, o contenere 2^{10} elementi di 4 byte. Gli indirizzi si presentano come segue:



La tabella esterna delle pagine consiste di 2^{42} elementi, o 2^{44} byte. La soluzione ovvia per evitare una tabella tanto grande consiste nel suddividere la tabella in parti più piccole. Questo metodo si adotta anche in alcune CPU a 32 bit allo scopo di fornire una maggiore flessibilità ed efficienza.

La tabella esterna delle pagine si può suddividere in vari modi. Si può paginare la tabella esterna delle pagine, ottenendo uno schema di paginazione a tre livelli. Si supponga che la tabella esterna delle pagine sia costituita di pagine di dimensione ordinaria (2^{10} elementi, o 2^{12} byte); uno spazio d'indirizzi a 64 bit è ancora enorme:

Possibile domanda:

Quanti accessi in memoria ci sono nel caso peggiore?

Risposta: una seconda tabella può portare fino a 4 accessi in memoria per un singolo accesso }

2nd outer page	outer page	inner page	offset
p_1	p_2	p_3	d
32	10	10	12

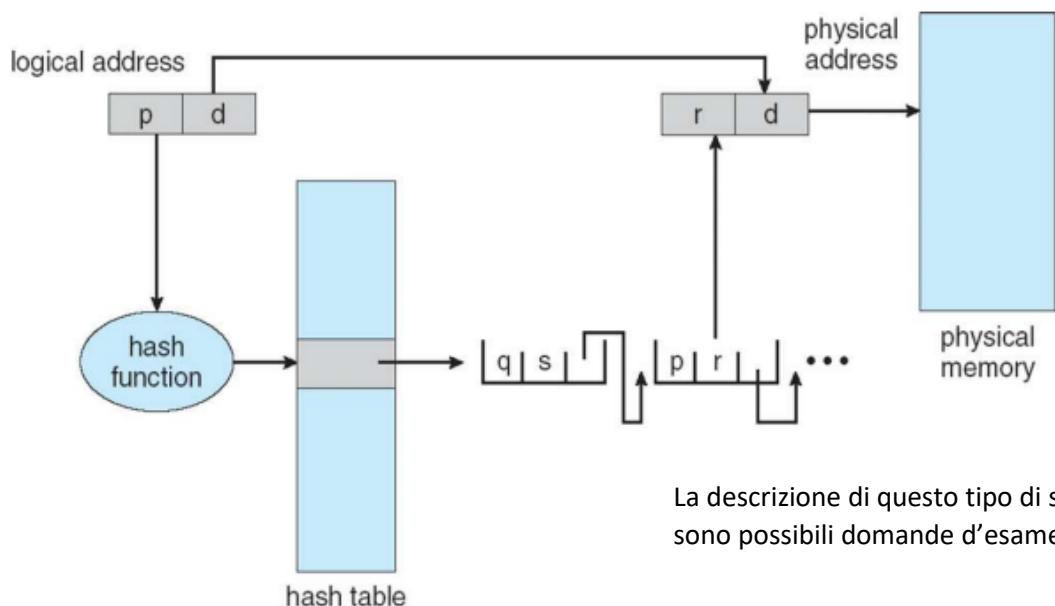
La tabella esterna delle pagine è ancora di 2^{34} byte.

Il passo successivo sarebbe uno schema di paginazione a quattro livelli, in cui si pagina anche la tabella esterna di secondo livello delle pagine. Da questo esempio è possibile capire perché, per le architetture a 64 bit, le tabelle delle pagine gerarchiche sono in genere considerate inappropriate.

Tabelle delle pagine di tipo hash

Un metodo di gestione molto comune degli spazi d'indirizzi relativi ad architetture 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. Per la gestione delle collisioni, 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 (pagina fisica) 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:



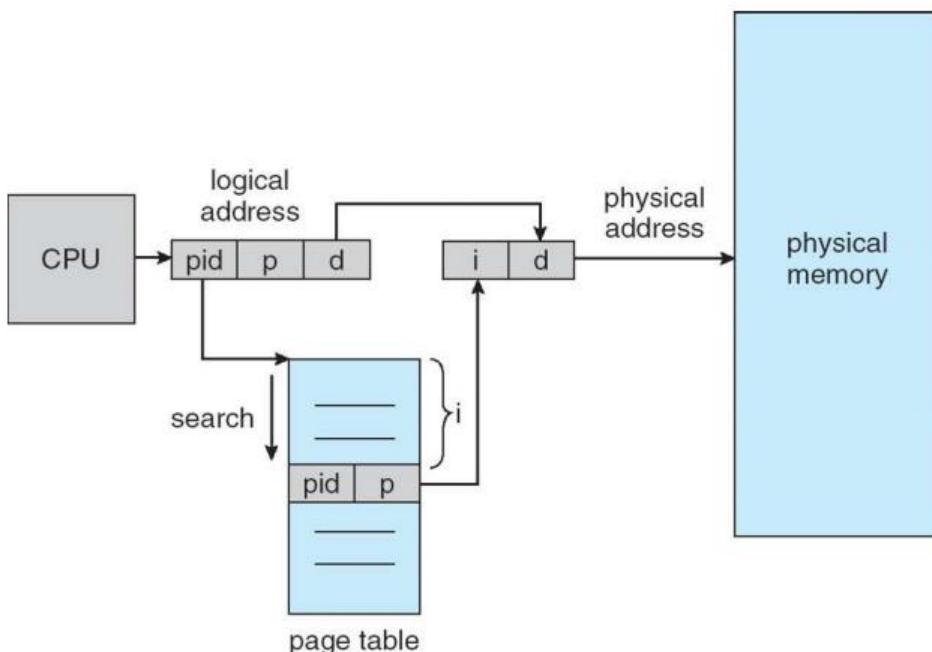
La descrizione di questo tipo di schemi sono possibili domande d'esame.

Le tabelle delle pagine di tipo hash sono particolarmente utili per gli spazi d'indirizzi sparsi, in cui i riferimenti alla memoria non sono contigui ma distribuiti per tutto lo spazio d'indirizzi.

Per questo schema è stata proposta una variante, adatta a spazi di indirizzamento a 64 bit. Si tratta della **clustered page tables** (tabella delle pagine a gruppi), simile alla tabella hash; ciascun elemento della tabella delle pagine contiene però i riferimenti alle pagine fisiche corrispondenti a un gruppo di pagine virtuali contigue (per esempio 16). In questo modo si riduce lo spazio di memoria richiesto. Inoltre, poiché lo spazio degli indirizzi di molti programmi non è arbitrariamente sparso su pagine isolate ma distribuito per raffiche di riferimenti, le tabelle delle pagine a gruppi sfruttano bene questa caratteristica.

Tabella delle pagine invertita

Generalmente, si associa una tabella delle pagine a ogni processo e tale tabella contiene un elemento per ogni pagina virtuale che il processo sta utilizzando, oppure un elemento per ogni indirizzo virtuale a prescindere dalla validità di quest'ultimo. Questa è una rappresentazione naturale della tabella, poiché i processi fanno riferimento alle pagine tramite gli indirizzi virtuali delle pagine stesse, che il sistema operativo deve poi tradurre in indirizzi di memoria fisica. Poiché la tabella è ordinata per indirizzi virtuali, il sistema operativo può calcolare in che punto della tabella si trova l'elemento dell'indirizzo fisico associato, e usare direttamente tale valore. Uno degli inconvenienti insiti in questo metodo è costituito dalla dimensione di ciascuna tabella delle pagine, che può contenere milioni di elementi e occupare grandi quantità di memoria fisica, necessaria proprio 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. Nella figura successiva sono mostrate le operazioni di una tabella delle pagine invertite.



Le tabelle invertite richiedono spesso la memorizzazione di un identificatore dello spazio d'indirizzi ([Architettura di paging](#)) in ciascun elemento della tabella delle pagine, perché essa contiene di solito molti spazi d'indirizzi diversi associati alla memoria fisica; l'identificatore garantisce che una data pagina logica relativa a un certo processo sia associata alla pagina fisica corrispondente.

Ciascun indirizzo virtuale è una tripla del tipo seguente: $\langle \text{process-id}, \text{page-number}, \text{offset} \rangle$. Ogni elemento della tabella delle pagine invertita è una coppia $\langle \text{process-id}, \text{page-number} \rangle$ dove il process-id assume il ruolo di identificatore dello spazio d'indirizzi. Quando si fa un riferimento alla memoria, si presenta una parte dell'indirizzo virtuale, formato da $\langle \text{process-id}, \text{page-number} \rangle$, al sottosistema di memoria. Quindi si cerca una corrispondenza nella tabella delle pagine invertita. Se si trova tale corrispondenza, per esempio sull'elemento i , si genera l'indirizzo fisico $\langle i, \text{offset} \rangle$. In caso contrario è stato tentato un accesso illegale a un indirizzo.

Sebbene questo schema riduca la quantità di memoria necessaria per memorizzare ogni tabella delle pagine, aumenta però il tempo di ricerca nella tabella quando si fa riferimento a una pagina. Poiché la tabella delle pagine invertita è ordinata per indirizzi fisici, mentre le ricerche si fanno per indirizzi virtuali, per trovare una corrispondenza occorre esaminare tutta la tabella; questa ricerca richiede molto tempo.

Per limitare l'entità del problema si può impiegare una tabella hash, che riduce la ricerca a un solo, o a pochi, elementi della tabella delle pagine. Naturalmente, ogni accesso alla tabella hash aggiunge al procedimento un riferimento alla memoria, quindi un riferimento alla memoria virtuale richiede almeno due letture della memoria reale: una per l'elemento della tabella hash e l'altro per la tabella delle pagine. Per migliorare le prestazioni, la ricerca si effettua prima nella TLB, quindi si consulta la tabella hash.

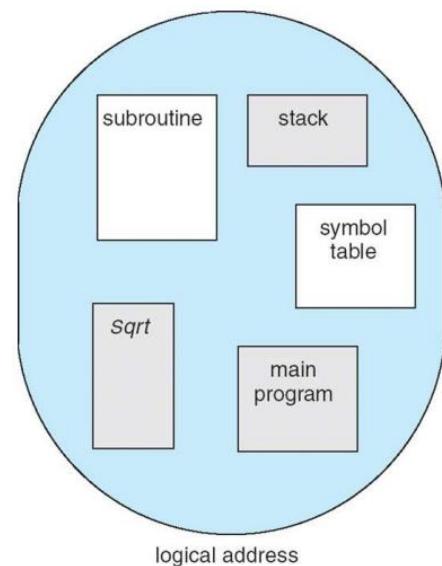
Segmentazione

Un aspetto importante della gestione della memoria, inevitabile alla presenza della paginazione, è quello della separazione tra la visione della memoria dell'utente e l'effettiva memoria fisica. Lo spazio d'indirizzi visto dall'utente non coincide con l'effettiva memoria fisica, ma lo si fa corrispondere alla memoria fisica. I metodi che stabiliscono questa corrispondenza consentono di separare la memoria logica da quella fisica.

Metodo di base

Ci si potrebbe chiedere se l'utente può considerare la memoria come un vettore lineare di byte, alcuni dei quali contengono istruzioni e altri dati. Molti risponderebbero di no. Gli utenti la vedono piuttosto come un insieme di segmenti di dimensione variabile non necessariamente ordinati (figura a destra).

La tipica struttura di un programma con cui i programmati hanno familiarità è costituita di una parte principale e di un gruppo di procedure, funzioni o moduli, insieme con diverse strutture dati come tabelle, matrici, pile, variabili e così via. Ciascuno di questi moduli o elementi di dati si identifica con un nome: "tabella dei simboli", "funzione sqrt()", "programma principale", indipendentemente dagli indirizzi che questi elementi occupano in memoria. Non è necessario preoccuparsi del fatto che la tabella dei simboli sia memorizzata prima o dopo la funzione sqrt(). Ciascuno di questi segmenti ha una lunghezza variabile, definita intrinsecamente dallo scopo che il segmento stesso ha all'interno del programma. Gli elementi che si trovano all'interno di un segmento sono identificati dal loro scostamento, misurato dall'inizio del segmento: la prima istruzione del programma, il settimo elemento della tabella dei simboli, la quinta istruzione della funzione sqrt() e così via.



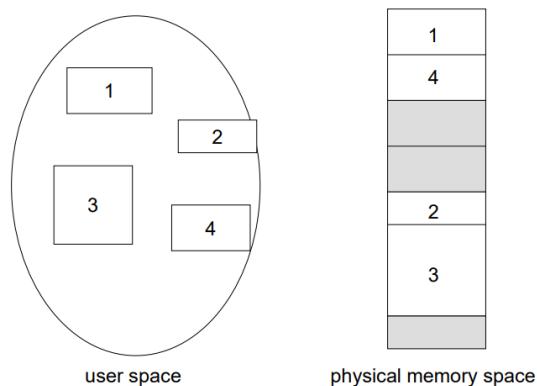
logical address

La **segmentazione** è uno schema di gestione della memoria che consente di gestire questa rappresentazione della memoria dal punto di vista dell'utente. Uno spazio d'indirizzi logici è una raccolta di segmenti, ciascuno dei quali ha un nome e una lunghezza. Gli indirizzi specificano sia il nome sia lo scostamento all'interno del segmento, quindi l'utente fornisce ogni indirizzo come una coppia ordinata di valori: un nome di segmento e uno scostamento. Questo schema contrasta con la paginazione, in cui l'utente fornisce un indirizzo singolo, che l'architettura di paginazione suddivide in un numero di pagine e uno scostamento, non visibili dal programmatore.

Per semplicità i segmenti sono numerati (vedi figura), e ogni riferimento si compie per mezzo di un numero anziché di un nome; quindi un indirizzo logico è una coppia (segment-number, offset).

Normalmente il programma utente è stato compilato, e il compilatore struttura automaticamente i segmenti secondo il programma sorgente. Un compilatore per il linguaggio C può creare segmenti distinti per i seguenti elementi di un programma:

- il codice;
- le variabili globali;
- lo heap, da cui si alloca la memoria;
- le pile usate da ciascun thread;

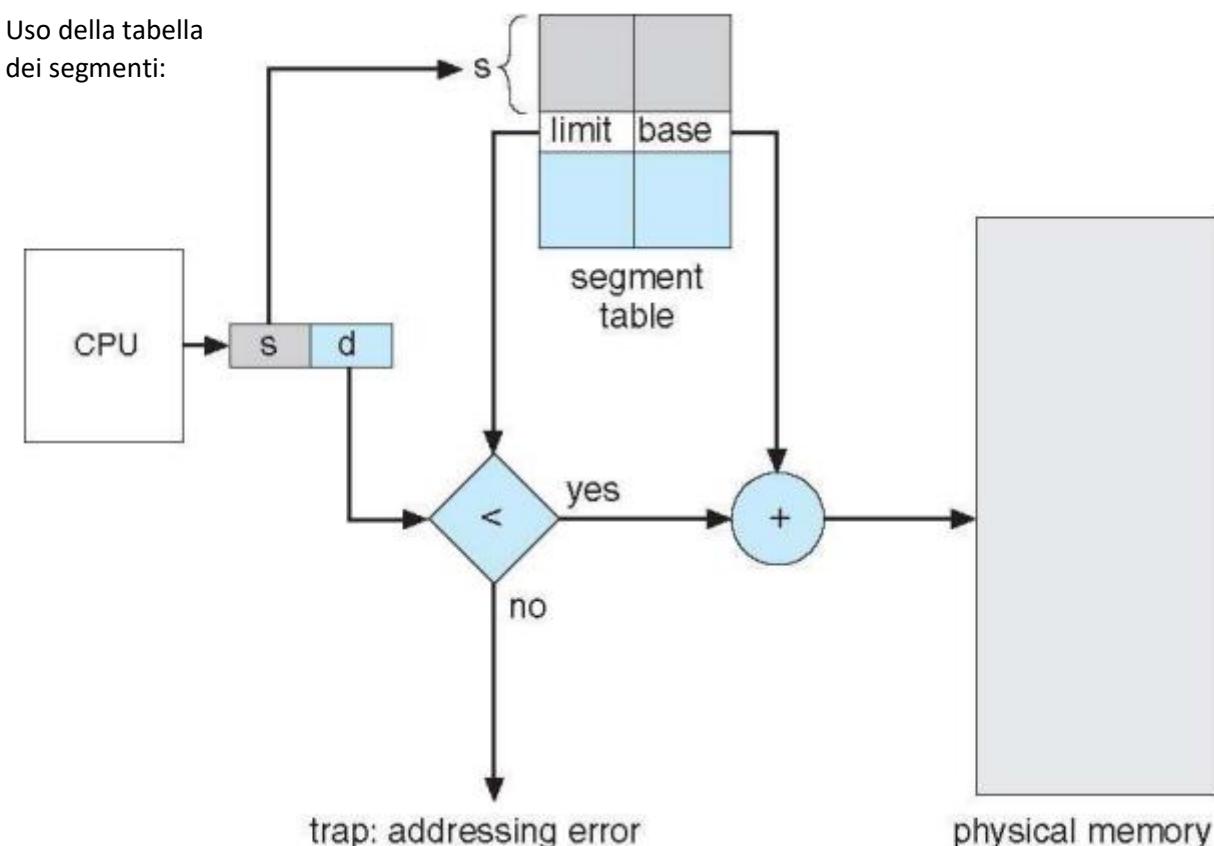


- la libreria standard del C.

Alle librerie collegate dal linker al momento della compilazione possono essere assegnati dei nuovi segmenti. Il loader preleva questi segmenti e assegna loro i numeri di segmento.

Architettura di segmentazione

Sebbene l'utente 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 **tavella dei segmenti**; ogni suo elemento è una coppia ordinata: la **base** del segmento e il **limite** del segmento. La base del segmento contiene l'indirizzo fisico iniziale della memoria dove il segmento risiede, mentre il limite del segmento contiene la lunghezza del segmento.

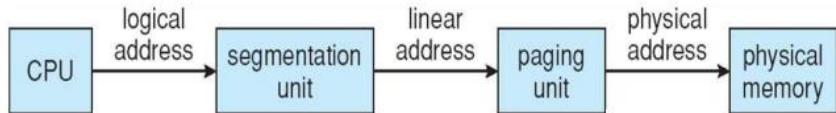


Un indirizzo logico è formato da due parti: un numero di segmento *s* e un offset in tale segmento *d*. Il numero di segmento si usa come indice per la tabella dei segmenti; l'offset *d* dell'indirizzo logico deve essere compreso tra 0 e il limite del segmento, altrimenti s'invia un segnale di eccezione al sistema operativo (tentativo di indirizzamento logico oltre la fine del segmento). Se tale condizione è rispettata, si somma l'offset alla base del segmento per produrre l'indirizzo della memoria fisica dove si trova il byte desiderato. Quindi la tabella dei segmenti è fondamentalmente un vettore di coppie di registri di base e limite.

Un esempio: Pentium Intel

Paginazione e segmentazione presentano vantaggi e svantaggi, tanto che alcune architetture dispongono di entrambe le tecniche. In questo paragrafo prendiamo in esame l'architettura del Pentium Intel che utilizza, oltre alla segmentazione pura, la segmentazione mista a paginazione. Non ci inoltreremo in una trattazione completa della gestione della memoria del Pentium, ma ci limiteremo a illustrare i concetti fondamentali su cui è basata. La nostra analisi si concluderà con una panoramica della traduzione degli indirizzi di Linux nei sistemi Pentium.

In questi sistemi la CPU genera indirizzi logici, che confluiscono nell'unità di segmentazione. Questa produce un indirizzo lineare per ogni indirizzo logico. L'indirizzo lineare passa quindi all'unità di paginazione, la quale, a sua volta, genera l'indirizzo fisico all'interno della memoria centrale. Così, le unità di segmentazione e di paginazione formano un equivalente dell'unità di gestione della memoria (MMU). Il modello è rappresentato nella figura seguente:



Segmentazione in Pentium

Nell'architettura Pentium un segmento può raggiungere la dimensione massima di 4 GB; il numero massimo di segmenti per processo è pari a 16 KB. Lo spazio degli indirizzi logici di un processo è composto da due partizioni: la prima contiene fino a 8 KB segmenti riservati al processo; la seconda contiene fino a 8 KB segmenti condivisi fra tutti i processi. Le informazioni riguardanti la prima partizione sono contenute nella **local descriptor table** (LDT, tabella locale dei descrittori), quelle relative alla seconda partizione sono memorizzate nella **global descriptor table** (GDT, tabella globale dei descrittori). Ciascun elemento nella LDT e nella GDT è lungo 8 byte e contiene informazioni dettagliate riguardanti uno specifico segmento, oltre agli indirizzi base e limite.

Un indirizzo logico è una coppia (selettore, offset), dove il selettore è un numero di 16 bit:

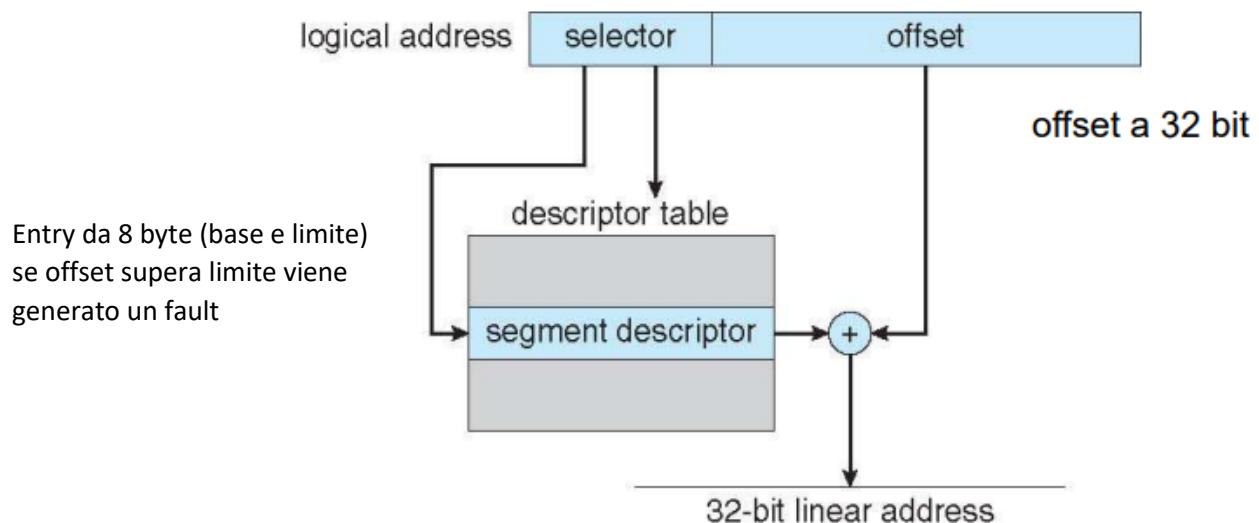
<i>s</i>	<i>g</i>	<i>p</i>
13	1	2

In cui *s* indica il numero del segmento, *g* indica se il segmento si trova nella GDT o nella LDT e *p* contiene informazioni relative alla protezione.

L'offset è un numero di 32 bit che indica la posizione del byte (o della parola) all'interno del segmento in questione.

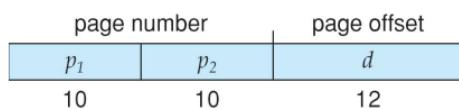
La macchina ha sei registri di segmento che consentono a un processo di far riferimento contemporaneamente a sei segmenti; inoltre possiede sei registri di microprogramma di 8 byte per i corrispondenti descrittori prelevati dalla LDT o dalla GDT. Questa cache evita a Pentium di dover prelevare dalla memoria i descrittori per ogni riferimento alla memoria.

Un indirizzo lineare di Pentium è lungo 32 bit e si genera come segue. Il registro di segmento punta all'elemento appropriato all'interno della LDT o della GDT; le informazioni relative alla base e al limite di tale segmento si usano per generare un indirizzo lineare. Innanzitutto si usa il valore del limite per controllare la validità dell'indirizzo; se non è valido, si ha un errore di riferimento alla memoria che causa l'invio di un segnale di eccezione e la restituzione del controllo al sistema operativo; altrimenti, si somma il valore dello scostamento al valore della base, ottenendo un indirizzo lineare di 32 bit. La figura di seguito illustra tale processo.

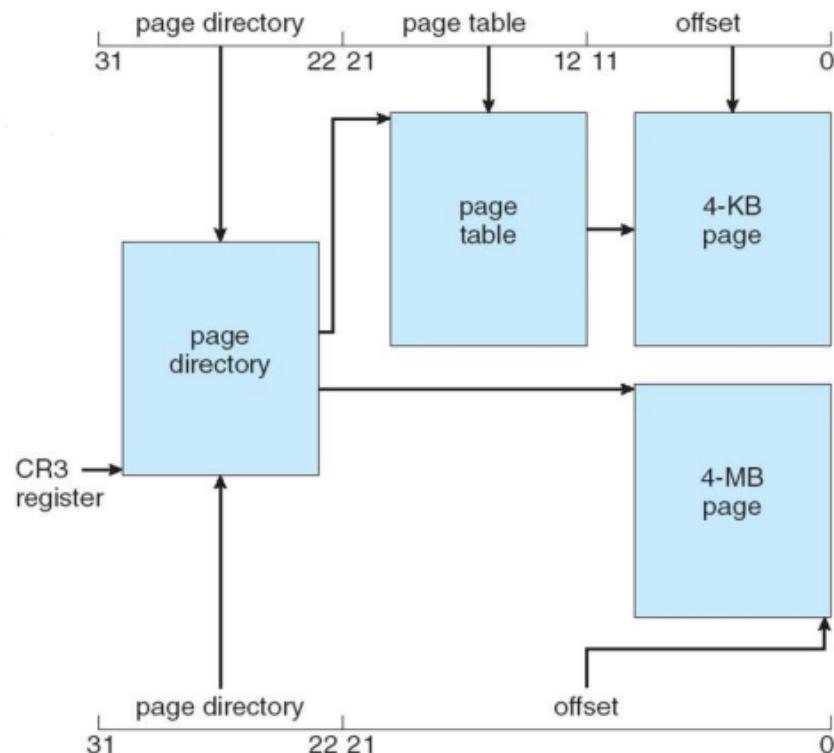


Paging in Pentium

L'architettura Pentium prevede che le pagine abbiano una misura di 4 KB oppure di 4 MB. Per le prime, in Pentium vige uno schema di paginazione a due livelli che prevede la seguente scomposizione degli indirizzi lineari a 32 bit:



Lo schema di traduzione degli indirizzi per questa architettura è mostrato in dettaglio nella figura seguente:



I dieci bit più significativi puntano a un elemento nella tabella delle pagine più esterna, detta in Pentium **directory delle pagine**. Gli elementi della directory delle pagine puntano a una tabella delle pagine interne, indicizzata da dieci bit intermedi dell'indirizzo lineare. Infine, i bit meno significativi in posizione 0-11 contengono lo scostamento da applicare all'interno della pagina di 4 KB cui si fa riferimento nella tabella delle pagine.

Un elemento appartenente alla directory delle pagine è il flag Page Size; se impostato, indica che il frame non ha la dimensione standard di 4 MB, ma misura invece 4 KB. In questo caso, la directory di pagina punta direttamente al frame di 4 MB, scavalcando la tabella delle pagine interna; i 22 bit meno significativi nell'indirizzo lineare indicano l'offset nella pagina di 4 MB.

Per migliorare l'efficienza d'uso della memoria fisica, le tabelle delle pagine del Pentium Intel possono essere trasferite sul disco. In questo caso, si ricorre a un bit in ciascun elemento della directory di pagina, per indicare se la tabella a cui l'elemento punta sia in memoria o sul disco. Nel secondo caso, il sistema operativo può usare i 31 bit rimanenti per specificare la collocazione della tabella sul disco; in questo modo, si può richiamare la tabella in memoria su richiesta.

9. Memoria Virtuale

Introduzione

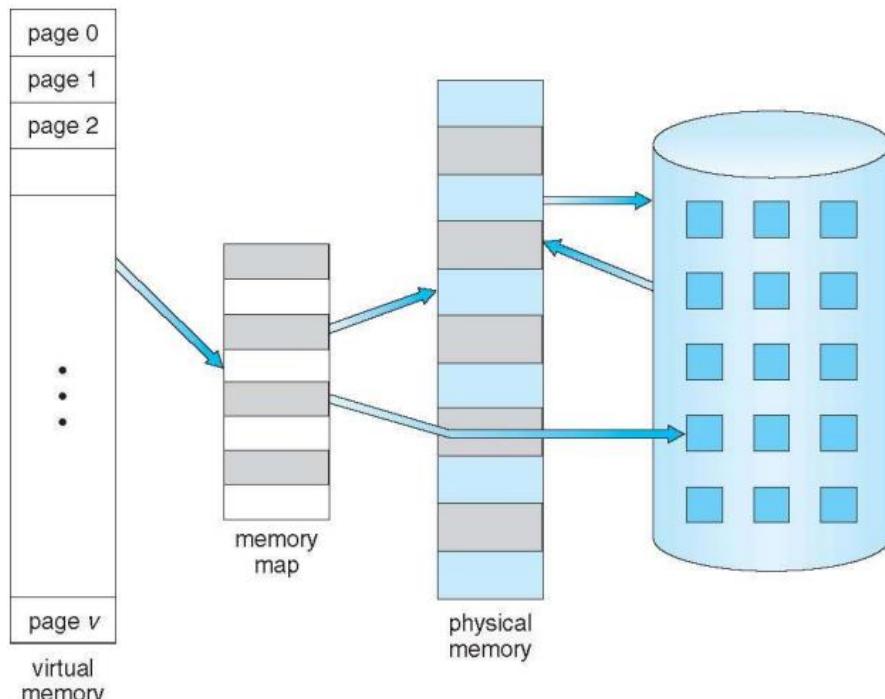
Gli algoritmi di gestione della memoria delineati nel [Capitolo 8](#) sono necessari perché, per l'attivazione di un processo, le istruzioni da eseguire si devono trovare all'interno della memoria fisica. Il primo metodo per far fronte a tale requisito consiste nel collocare l'intero spazio d'indirizzi logici del processo relativo in memoria fisica. Il caricamento dinamico può aiutare ad attenuare gli effetti di tale limitazione, ma richiede generalmente particolari precauzioni e un ulteriore impegno dei programmatori.

La condizione che le istruzioni debbano essere nella memoria fisica sembra tanto necessaria quanto ragionevole, ma purtroppo riduce le dimensioni dei programmi a valori strettamente correlati alle dimensioni della memoria fisica. In effetti, da un esame dei programmi reali risulta che in molti casi non è necessario avere in memoria l'intero programma; ed anche nei casi in cui è necessario disporre di tutto il programma è possibile che non serva tutto in una volta.

La possibilità di eseguire un programma che si trova solo parzialmente in memoria può essere vantaggiosa per i seguenti motivi.

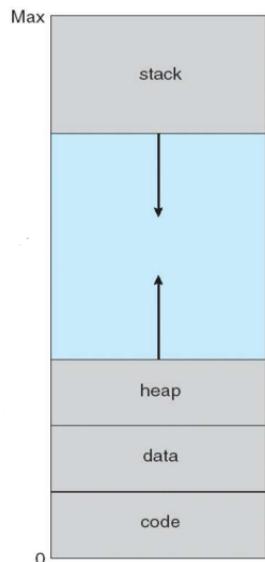
- Un programma non è più vincolato alla quantità di memoria fisica disponibile. Gli utenti possono scrivere programmi per uno spazio degli indirizzi virtuali molto grande, semplificando così le operazioni di programmazione.
- Poiché ogni utente impiega meno memoria fisica, si possono eseguire molti più programmi contemporaneamente, ottenendo un corrispondente aumento dell'utilizzo e della produttività della CPU senza aumentare il tempo di risposta o di completamento.
- Per caricare (o scaricare) ogni programma utente in memoria sono necessarie meno operazioni di I/O, quindi ogni programma utente è eseguito più rapidamente.

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, com'è illustrato nella seguente figura:



La memoria virtuale facilita la programmazione, poiché il programmatore non deve preoccuparsi della quantità di memoria fisica disponibile o di quale codice si debba inserire nelle sezioni sovrapponibili, ma può concentrarsi sul problema da risolvere.

L'espressione spazio degli indirizzi virtuali si riferisce alla collocazione dei processi in memoria dal punto di vista logico (o virtuale). Da tale punto di vista, un processo inizia in corrispondenza di un certo indirizzo logico (per esempio, l'indirizzo 0) e si estende alla memoria contigua, come evidenziato dalla figura a destra. È tuttavia possibile organizzare la memoria fisica in frame di pagine; in questo caso i frame delle pagine fisiche assegnati ai processi possono non essere contigui. Spetta all'unità di gestione della memoria (**MMU**) associare in memoria le pagine logiche alle pagine fisiche. Si noti come allo heap sia lasciato sufficiente spazio per crescere verso l'alto nello spazio di memoria, poiché esso ospita la memoria allocata dinamicamente. In modo analogo, consentiamo alla pila di svilupparsi verso il basso nella memoria, a causa di ripetute chiamate di funzione. Lo spazio vuoto ben visibile (o hole) che separa lo heap dalla pila è parte dello spazio degli indirizzi virtuali, ma richiede pagine fisiche realmente esistenti solo nel caso che lo heap o la pila crescano. Qualora contenga hole, lo spazio degli indirizzi virtuali si definisce sparso. Un simile spazio degli indirizzi è doppiamente utile, poiché consente di riempire gli hole grazie all'espansione dei segmenti heap o pila, e di collegare dinamicamente delle librerie (o altri oggetti condivisi) durante l'esecuzione del programma.



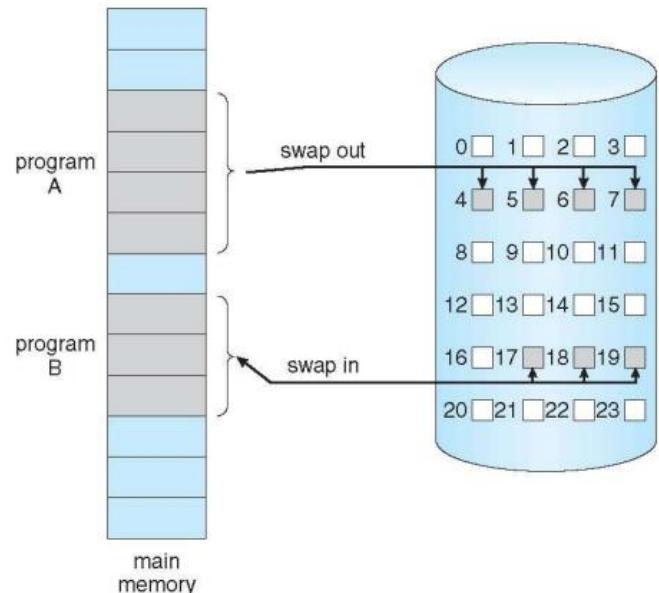
Oltre a separare la memoria logica da quella fisica, la memoria virtuale offre, per due o più processi, il vantaggio di condividere i file e la memoria, mediante la [condivisione delle pagine](#). Ciò comporta i seguenti vantaggi:

- Le librerie di sistema sono condivisibili da diversi processi associando l'oggetto condiviso a uno spazio degli indirizzi virtuali, procedimento detto mappatura. Benché ciascun processo veda le librerie condivise come parte del proprio spazio degli indirizzi virtuali, le pagine che ospitano effettivamente le librerie nella memoria fisica sono in condivisione tra tutti i processi.
- In maniera analoga, la memoria virtuale rende i processi in grado di condividere la memoria. La memoria virtuale permette a un processo di creare una regione di memoria condivisibile da un altro processo. I processi che condividono questa regione la considerano parte del proprio spazio degli indirizzi virtuali, malgrado le pagine fisiche siano, in realtà, condivise.
- La memoria virtuale può consentire, per mezzo della chiamata di sistema `fork()`, che le pagine siano condivise durante la creazione di un processo, così da velocizzare la generazione dei processi.

Paginazione su richiesta

Una strategia per il caricamento in memoria di un eseguibile residente su disco consiste nel caricare le pagine nel momento in cui servono realmente; si tratta di una tecnica, detta **paginazione su richiesta** (*demand paging*), comunemente adottata dai sistemi con memoria virtuale. 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.

Un sistema di paginazione su richiesta è analogo a un sistema paginato con swapping dei processi in memoria; si veda la figura a destra. I processi risiedono in memoria secondaria, generalmente costituita di uno o più dischi. Per eseguire un

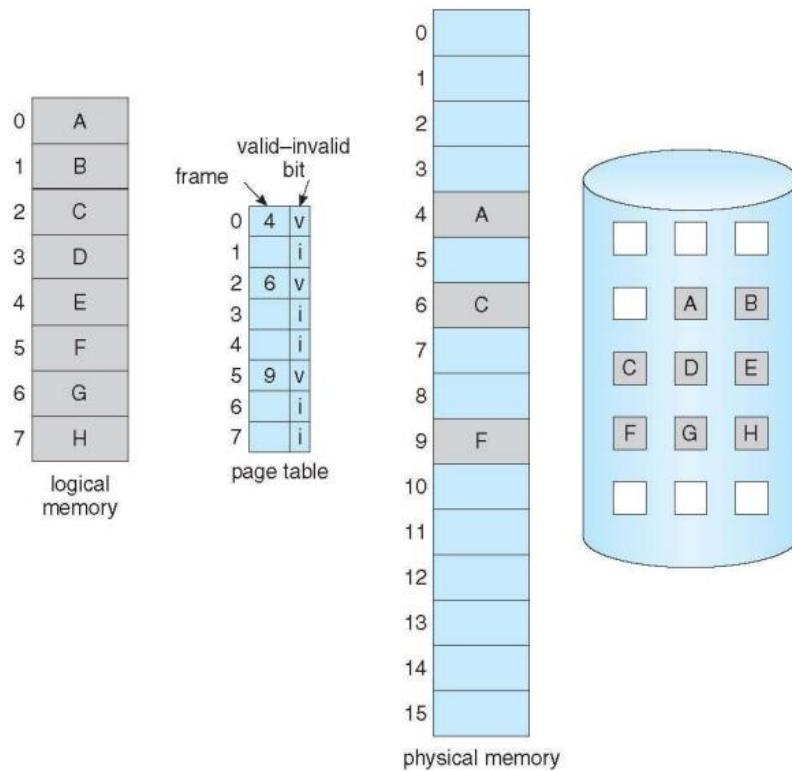


processo occorre caricarlo in memoria. Tuttavia, anziché caricare in memoria l'intero processo, si può seguire un **lazy swapping**: non si carica mai in memoria una pagina che non sia necessaria. Non si manipolano interi processi ma singole pagine dei processi. Nell'ambito della paginazione su richiesta, il modulo del sistema operativo che si occupa della sostituzione delle pagine si chiama **paginatore (page manager)**.

Concetti fondamentali

Quando un processo sta per essere caricato in memoria, il paginatore ipotizza quali pagine saranno usate, prima che il processo sia nuovamente scaricato dalla memoria. Anziché caricare in memoria tutto il processo, il paginatore trasferisce in memoria solo le pagine che ritiene necessarie. In questo modo è possibile evitare il trasferimento in memoria di pagine che non sono effettivamente usate, riducendo il tempo d'avvicendamento e la quantità di memoria fisica richiesta.

Con tale schema è necessario che l'architettura disponga di un qualche 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, descritto nel paragrafo "[Struttura della tabella delle pagine](#)". 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 o contiene l'indirizzo che consente di trovare la pagina nei dischi. Tale situazione è illustrata nella figura seguente:

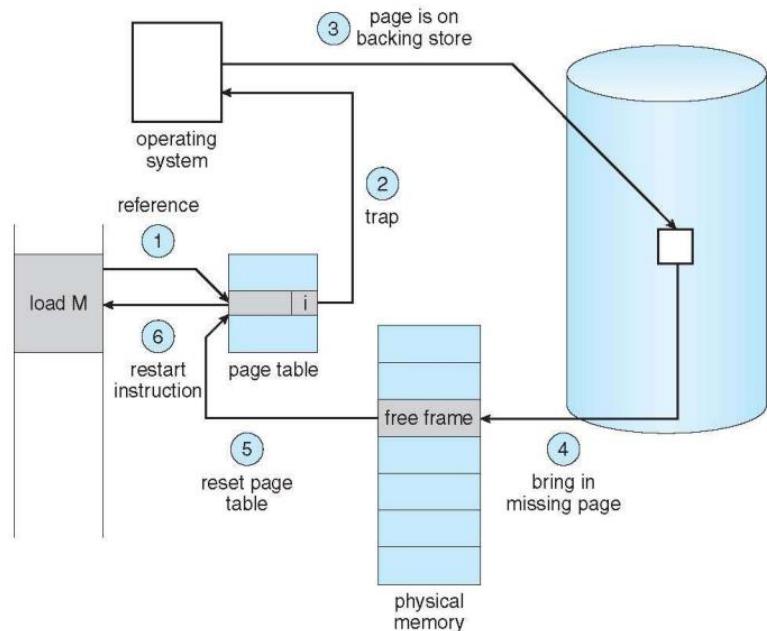


Occorre notare che indicare una pagina come non valida non sortisce alcun effetto se il processo non tenta mai di accedervi. Quindi, se l'ipotesi del paginatore è esatta e si caricano tutte e solo le pagine che servono effettivamente, il processo è eseguito proprio come se fossero state caricate tutte le pagine. Durante l'esecuzione, il processo accede alle pagine residenti in memoria, e l'esecuzione procede come di consueto.

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 una **page fault trap** (eccezione di pagina mancante). L'architettura di paginazione, traducendo l'indirizzo attraverso la tabella delle pagine, nota che il bit è non valido e invia un

segnale di eccezione al sistema operativo; tale eccezione è dovuta a un “insuccesso” del sistema operativo nella scelta delle pagine da caricare in memoria. La procedura di gestione dell’eccezione di pagina mancante (figura a destra) è chiara, e corrisponde ai passi seguenti (N.B.: descrivere in modo schematico i passi per la gestione del page fault è una domanda tipica):

- 1) Si controlla una tabella interna per questo processo; in genere tale tabella è conservata insieme al PCB, allo scopo di stabilire se il riferimento fosse un accesso alla memoria valido o non valido.
- 2) Se il riferimento non era valido, si termina il processo. Se era un riferimento valido, ma la pagina non era ancora stata portata in memoria, se ne effettua l’inserimento.
- 3) Si individua un frame libero, ad esempio prelevandone uno dalla lista dei frame liberi.
- 4) Si programma un’operazione sui dischi per trasferire la pagina desiderata nel frame appena assegnato.
- 5) Quando la lettura dal disco è completata, si modificano la tabella interna, conservata con il processo, e la tabella delle pagine per indicare che la pagina si trova attualmente in memoria.
- 6) Si riavvia l’istruzione interrotta dal segnale di eccezione. A questo punto il processo può accedere alla pagina come se questa fosse già presente in memoria.



È addirittura possibile avviare l’esecuzione di un processo senza pagine in memoria. Quando il sistema operativo carica nel contatore di programma l’indirizzo della prima istruzione del processo, che è in una pagina non residente in memoria, il processo accusa un’assenza di pagina. Una volta portata la pagina in memoria, il processo continua l’esecuzione, subendo assenze di pagine fino a che tutte le pagine necessarie non si trovino effettivamente in memoria; a questo punto si può eseguire il processo senza ulteriori richieste. Lo schema descritto è una paginazione su richiesta pura, vale a dire che una pagina non si trasferisce in memoria finché non sia richiesta.

In teoria alcuni programmi possono accedere a più pagine di memoria all’esecuzione di ogni istruzione (una pagina per l’istruzione e molte per i dati), causando più possibili page fault per ogni istruzione. In un caso simile le prestazioni del sistema sarebbero inaccettabili.

I meccanismi d’ausilio alla paginazione su richiesta che l’architettura del calcolatore deve offrire sono quelli richiesti per la paginazione e l’avvicendamento dei processi in memoria:

- **tabella delle pagine.** Questa tabella ha la capacità di contrassegnare un elemento come non valido attraverso un bit di validità oppure un valore speciale dei bit di protezione;
- **memoria secondaria.** Questa memoria conserva le pagine non presenti in memoria centrale. Generalmente la memoria secondaria è costituita da un disco ad alta velocità detto dispositivo d’avvicendamento; la sezione del disco usata a questo scopo si chiama **swap space**.

Uno dei requisiti cruciali della paginazione su richiesta è la possibilità di rieseguire una qualunque istruzione a seguito di un page fault (è necessario il **restart delle istruzioni** dopo il page fault). Avendo salvato lo stato del processo interrotto (registri, codici di condizione, contatore di programma) a causa della pagina mancante, occorrerà riavviare il processo esattamente dallo stesso punto e con lo stesso stato, eccezion

fatta per la presenza della pagina desiderata in memoria. Nella maggior parte dei casi questa situazione è piuttosto comune: un page fault si può verificare per qualsiasi riferimento alla memoria. Se il page fault si presenta durante la fase di prelievo di un'istruzione, l'esecuzione si può riavviare prelevando nuovamente tale istruzione. Se si verifica durante il prelievo di un operando, l'istruzione deve essere di nuovo prelevata e decodificata, quindi si può prelevare l'operando.

Come caso limite si consideri un'istruzione in tre indirizzi, come ad esempio la somma (*ADD*) del contenuto di *A* al contenuto di *B*, con risultato posto in *C*. I passi necessari per eseguire l'istruzione sono i seguenti:

- 1) prelievo e decodifica dell'istruzione (*ADD*);
- 2) prelievo del contenuto di *A*;
- 3) prelievo del contenuto di *B*;
- 4) addizione del contenuto di *A* al contenuto di *B*;
- 5) memorizzazione della somma in *C*.

Se l'assenza di pagina avviene al momento della memorizzazione in *C*, poiché *C* si trova in una pagina che non è nella memoria, occorre prelevare la pagina desiderata, caricarla nella memoria, correggere la tabella delle pagine e riavviare l'istruzione. Il riavvio dell'istruzione richiede una nuova operazione di prelievo della stessa, con nuova decodifica e nuovo prelievo dei due operandi; infine occorre ripetere l'addizione. In ogni modo la ripetizione è necessaria solo nel caso si verifichi un'assenza di pagina.

Prestazioni della paginazione su richiesta

La paginazione su richiesta può avere un effetto rilevante sulle prestazioni di un calcolatore. Il motivo si può comprendere calcolando il tempo d'accesso effettivo per una memoria con paginazione su richiesta (questo tipo di calcoli potrebbero essere un esercizio d'esame).

Attualmente, nella maggior parte dei calcolatori il tempo d'accesso alla memoria, che si denota *ma*, varia da 10 a 200 nanosecondi. Finché non si verifichino assenze di pagine, il tempo d'accesso effettivo è uguale al tempo d'accesso alla memoria. Se però si verifica un'assenza di pagina, occorre prima leggere dal disco la pagina interessata e quindi accedere alla parola della memoria desiderata.

Supponendo che *p* sia la probabilità che si verifichi un page fault ($0 \leq p \leq 1$), è probabile che *p* sia molto vicina allo zero, cioè che ci siano pochi page fault. Il **tempo d'accesso effettivo** (in breve EAT) è dato dalla seguente espressione: $EAT = (1 - p) * ma + p * t$ (dove *t* è il tempo di gestione del page fault).

Per calcolare l'EAT occorre conoscere il tempo necessario alla gestione di un page fault. Alla presenza di un page fault si eseguono tre operazioni principali: (1) servizio del segnale di page fault trap; (2) lettura della pagina; (3) riavvio del processo. La prima e la terza operazione si possono realizzare, per mezzo di un'accurata codifica, in parecchie centinaia di istruzioni. Ciascuna di queste operazioni può richiedere da 1 a 100 microsecondi. D'altra parte, il tempo di cambio di pagina è probabilmente vicino a 8 millisecondi.⁷

Considerando un tempo medio di servizio dell'eccezione di pagina mancante di 8 millisecondi e un tempo d'accesso alla memoria di 200 nanosecondi, il tempo effettivo d'accesso in nanosecondi è il seguente:

$$EAT = (1 - p)200 + p(8.000.000) = 200 + 7.999.800p$$

il tempo d'accesso effettivo è direttamente proporzionale alla **frequenza di page fault** (anche detto *page fault rate*). Se un accesso su 1000 accusa un'assenza di pagina, il tempo d'accesso effettivo è di 8,2 microsecondi. Impiegando la paginazione su richiesta, il calcolatore è rallentato di un fattore pari a 40. Se si vuole mantenere ragionevole il rallentamento dovuto alla paginazione si può permettere al più un page fault ogni 400mila accessi alla memoria.

Copy on write

Si ricordi che la chiamata di sistema *fork()* crea un processo figlio come duplicato del genitore. Nella sua versione originale la *fork()* creava per il figlio una copia dello spazio d'indirizzi del genitore, duplicando le pagine appartenenti al processo genitore. Considerando che molti processi figli eseguono subito dopo la

loro creazione la chiamata di sistema exec(), questa operazione di copiatura risulta inutile. In alternativa, si può impiegare una tecnica nota come **copy-on-write** (copiatura su scrittura), il cui funzionamento si fonda sulla condivisione iniziale delle pagine da parte dei processi genitori e dei processi figli. Le pagine condivise si contrassegnano 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.

Quando è necessaria la duplicazione di una pagina secondo la tecnica di copy on write, è importante capire da dove si attingerà la pagina libera necessaria. Molti sistemi operativi forniscono, per queste richieste, un gruppo (pool) di pagine libere, che di solito si assegnano quando la pila o il cosiddetto heap di un processo devono espandersi, oppure proprio per gestire pagine da copiare su scrittura. L'allocazione di queste pagine di solito avviene secondo una tecnica nota come **zero-fill-on-demand** (azzeramento su richiesta); prima dell'allocazione si riempiono di zeri le pagine, cancellandone in questo modo tutto il contenuto precedente.

Sostituzione delle pagine

Nelle descrizioni fatte finora, la frequenza (rate) delle assenze di pagine non è stata un problema grave, giacché ogni pagina poteva essere assente al massimo una volta, e precisamente la prima volta in cui si effettuava un riferimento a essa. Tale rappresentazione tuttavia non è molto precisa. Se un processo di 10 pagine ne impiega effettivamente solo la metà, la paginazione su richiesta fa risparmiare l'I/O necessario per caricare le cinque pagine che non sono mai usate. Il grado di multiprogrammazione potrebbe essere aumentato eseguendo il doppio dei processi. Quindi, disponendo di 40 frame, si potrebbero eseguire otto processi anziché i quattro che si eseguirebbero se ciascuno di loro richiedesse 10 blocchi di memoria, cinque dei quali non sarebbero mai usati.

Aumentando il grado di multiprogrammazione, si **sovrassegna** la memoria. Eseguendo 6 processi, ciascuno dei quali è formato da 10 pagine, di cui solo cinque sono effettivamente usate, s'incrementerebbero l'utilizzo e la produttività della CPU e si risparmierebbero 10 frame. Tuttavia è possibile che ciascuno di questi processi, per un insieme particolare di dati, abbia improvvisamente necessità di impiegare tutte le 10 pagine, perciò sarebbero necessari 60 frame, mentre ne sono disponibili solo 40.

Si consideri inoltre che la memoria del sistema non si usa solo per contenere pagine di programmi: le aree di memoria per l'I/O impegnano una rilevante quantità di memoria. Ciò può aumentare le difficoltà agli algoritmi di allocazione della memoria. Decidere quanta memoria assegnare all'I/O e quanta alle pagine dei programmi è un problema rilevante. Alcuni sistemi riservano una quota fissa di memoria per l'I/O, altri permettono sia ai processi utenti sia al sottosistema di I/O di competere per tutta la memoria del sistema.

La **sovrallocazione** (*over-allocation*) si può illustrare come segue. Durante l'esecuzione di un processo utente si verifica un'assenza di pagina. Il sistema operativo determina la locazione del disco in cui risiede la pagina desiderata, ma poi scopre che la lista dei frame liberi è vuota: tutta la memoria è in uso.

Schema di base

La sostituzione delle pagine segue il seguente criterio: se nessun blocco di memoria (frame) è libero si può liverarne uno attualmente inutilizzato. Un frame si può liberare scrivendo il suo contenuto nell'area di swapping e modificando la tabella delle pagine per indicare che la pagina non si trova più nella memoria. Il frame liberato si può usare per conservare la pagina che ha causato l'eccezione. Si modifica la procedura di servizio di page fault in modo da includere la sostituzione della pagina:

- 1) s'individua la locazione nel disco della pagina richiesta
- 2) si cerca un frame libero:
 - a. se esiste, lo si usa,
 - b. altrimenti si impiega un algoritmo di sostituzione delle pagine per scegliere un frame vittima

- c. si scrive il frame vittima nel disco; si modificano nel modo che ne conseguono le tabelle delle pagine e dei blocchi di memoria;
- 3) si scrive la pagina richiesta nel blocco di memoria appena liberato; si modificano le tabelle delle pagine e dei blocchi di memoria;
- 4) si riavvia il processo utente dall'istruzione che ha causato il trap.

Occorre notare che se non esiste alcun frame libero sono necessari **due** trasferimenti di pagina per page fault (uno fuori e uno dentro la memoria).

Questo sovraccarico si può ridurre usando un **dirty bit** (bit di modifica). L'architettura fisica del sistema di calcolo può disporre di un bit di modifica, associato a ogni pagina (o frame), che imposta automaticamente ogni volta che nella pagina si scrive una parola o un byte, indicando che quella pagina è stata modificata.

Quando si sceglie una pagina da sostituire si esamina il suo dirty bit ; se è attivo, significa che quella pagina è stata modificata rispetto a quanto era stata letta dal disco; in questo caso la pagina deve essere scritta nel disco. Se il bit non è attivo, significa che la pagina non è stata modificata da quando è stata caricata nella memoria, quindi non è necessario scrivere nel disco la pagina di memoria: c'è già.

Per realizzare la paginazione su richiesta è necessario risolvere due problemi principali: occorre sviluppare un **algoritmo di allocazione dei frame** e un **algoritmo di sostituzione delle pagine**. Se sono presenti più processi in memoria, occorre decidere quanti frame vadano assegnati a ciascun processo. Inoltre, quando è richiesta una sostituzione di pagina, occorre selezionare i frame da sostituire.

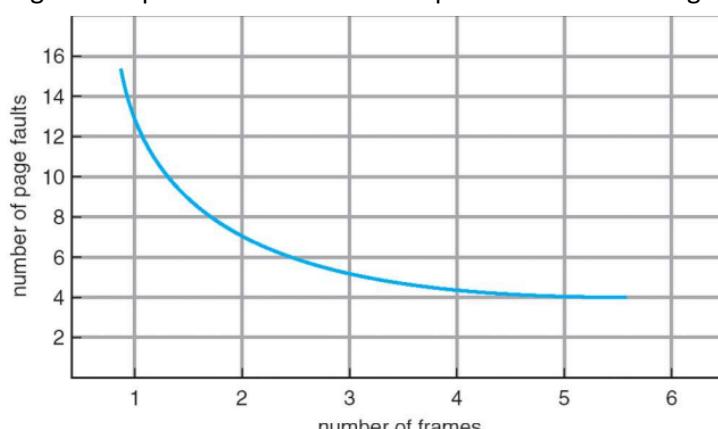
Esistono molti algoritmi di sostituzione delle pagine; probabilmente ogni sistema operativo ha il proprio schema di sostituzione. E quindi necessario stabilire un criterio per selezionare un algoritmo di sostituzione particolare; generalmente si sceglie quello con il minimo **page fault rate** (frequenza di page fault).

Un algoritmo si valuta effettuandone l'esecuzione su una particolare successione di riferimenti alla memoria e calcolando il numero di assenze di pagine. La successione dei riferimenti alla memoria è detta **stringa dei riferimenti**. Questa stringa si può generare artificialmente (generatore casuale di numeri) oppure analizzando un dato sistema e registrando l'indirizzo di ciascun riferimento alla memoria:

- Esaminando un processo si potrebbe ad esempio registrare la seguente successione di indirizzi:
`0100, 0432, 0101, 0612, 0102, 0103, 0104, 0101, 0611, 0102, 0103,
 0104, 0101, 0610, 0102, 0103, 0104, 0101, 0609, 0102, 0105`
- che, a 100 byte per pagina, si riduce alla seguente stringa di riferimenti: 1,4,1,6,1,6,1,6,1,6,1.

Per stabilire il numero di page fault relativo a una particolare stringa di riferimenti e a un particolare algoritmo di demand paging, occorre conoscere anche il numero dei frame disponibili (se è disponibile un solo frame allora avrà tutti fault poiché è necessaria una sostituzione per ogni riferimento).

In genere è prevista un curva come quella illustrata nel seguente grafico:



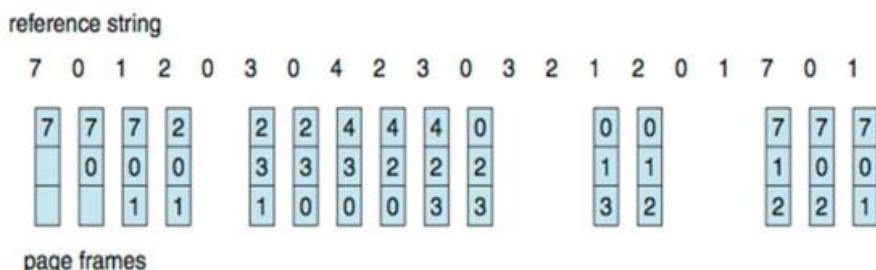
Aumentando il numero dei frame, il numero di assenze di pagine diminuisce fino al livello minimo. Naturalmente aggiungendo memoria fisica il numero dei frame aumenta.

N.B.: i seguenti algoritmi sono spesso domande d'esame. Nella loro illustrazione impiegheremo **sempre** la seguente stringa di riferimenti 7,0,1,2,0,3,0,4,2,3,0,3,0,3,2,1,2,0,1,7,0,1 per una memoria con **tre** frame.

Algoritmo di sostituzione 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. Occorre notare che non è strettamente necessario registrare l'istante in cui si carica una pagina in memoria; infatti si possono strutturare secondo una coda FIFO tutte le pagine presenti in memoria. In questo caso si sostituisce la pagina che si trova nel primo elemento della coda. Quando si carica una pagina in memoria, la si inserisce nell'ultimo elemento della coda.

Nella successione di riferimenti adottata, i nostri tre frame sono inizialmente vuoti. I primi tre riferimenti (7,0,1) accusano ciascuno un'assenza di pagina con conseguente caricamento delle relative pagine nei frame vuoti. Il riferimento successivo (2) causa la sostituzione della pagina 7, perché essa è stata caricata per prima in memoria. Siccome 0 è il riferimento successivo e si trova già in memoria, per questo riferimento non ha luogo alcuna assenza di pagina. Il primo riferimento a 3 causa la sostituzione della pagina 0, che era la prima fra le tre pagine in memoria (0, 1, e 2) da caricare. A causa di questa sostituzione il riferimento successivo, a 0, accuserà un'assenza di pagina. La pagina 1 è allora sostituita dalla pagina 0. Questo processo prosegue come è illustrato in seguito:



Le pagine presenti nei tre frame sono indicate ogni volta che si verifica un'assenza di pagina.

Complessivamente si hanno 15 assenze di pagine (questo è anche un esempio di domanda, dati 3 o 4 riferimenti, quanti page fault vengono generati? Quanto siamo vicini all'ottimo?).

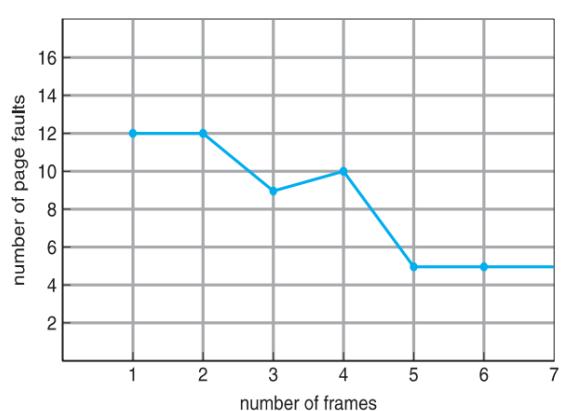
L'algoritmo FIFO di sostituzione delle pagine è facile da capire e da programmare; tuttavia la sue prestazioni non sono sempre buone. La pagina sostituita potrebbe essere un modulo di inizializzazione usato molto tempo prima e che non serve più, ma potrebbe anche contenere una variabile molto usata, inizializzata precedentemente, e ancora in uso.

Occorre notare che anche se si sceglie una pagina da sostituire che è in uso attivo, tutto continua a funzionare correttamente. Dopo aver rimosso una pagina attiva per inserirne una nuova, quasi immediatamente si verifica un page fault trap per riprendere la pagina attiva. Per riportare la pagina attiva in memoria è necessario sostituire un'altra pagina. Quindi, scegliendo una sostituzione errata, aumenta la frequenza di pagine mancanti che rallenta l'esecuzione del processo, ma non vengono causati errori.

Per illustrare i problemi che possono insorgere con l'uso dell'algoritmo di sostituzione delle pagine FIFO, si consideri la seguente successione di riferimenti:

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

Nel grafico è illustrata la curva a delle assenze di pagine in funzione del numero dei frame disponibili. Occorre notare che il numero delle assenze di pagine (10) per quattro frame è maggiore del numero delle assenze di pagine (9) per tre frame. Questo inatteso risultato è noto

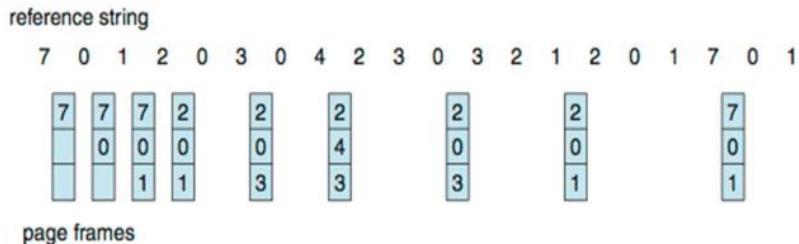


col nome di **anomalia di Belady**, e riflette il fatto che, con alcuni algoritmi di sostituzione delle pagine, la frequenza delle assenze di pagine può aumentare con l'aumentare del numero dei frame assegnati. A prima vista sembra logico supporre che fornendo più memoria a un processo le prestazioni di quest'ultimo migliorino. Si è invece notato che questo presupposto non sempre è vero; l'anomalia di Belady ne è la prova.

Algoritmo ottimale

In seguito alla scoperta dell'anomalia di Belady, la ricerca si è diretta verso un algoritmo ottimale di sostituzione delle pagine. Tale algoritmo è quello che fra tutti gli algoritmi presenta la minima frequenza di assenze di pagine e non presenta mai l'anomalia di Belady. Questo algoritmo esiste ed è stato chiamato **OPT** o **MIN**. È semplicemente si sostituisce la pagina che non si userà per il periodo di tempo più lungo. L'uso di quest'algoritmo di sostituzione delle pagine assicura la frequenza di assenze di pagine più bassa possibile per un numero fisso di frame.

Ad esempio, nella successione dei riferimenti considerata, l'algoritmo ottimale di sostituzione delle pagine produce nove assenze di pagine, come è mostrato nella figura seguente:



I primi tre riferimenti causano assenze di pagine che riempiono i tre blocchi di memoria vuoti. Il riferimento alla pagina 2 determina la sostituzione della pagina 7, perché la 7 non è usata fino al riferimento 18, mentre la pagina 0 viene usata al 5 e la pagina 1 al 14. Il riferimento alla pagina 3 causa la sostituzione della pagina, poiché la pagina 1 è l'ultima delle tre pagine in memoria cui si fa nuovamente riferimento. Con sole nove assenze di pagine, la sostituzione ottimale risulta assai migliore di quella ottenuta con un algoritmo FIFO, dove le assenze di pagine erano 15. Ignorando le prime tre assenze di pagine, che si verificano con tutti gli algoritmi, la sostituzione ottimale è due volte migliore rispetto all'algoritmo FIFO; nessun algoritmo di sostituzione può gestire questa successione di riferimenti a tre blocchi di memoria con meno di nove assenze di pagine.

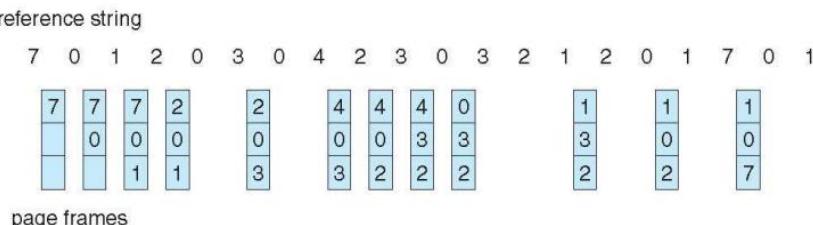
Sfortunatamente l'algoritmo ottimale di sostituzione delle pagine è difficile da realizzare, perché richiede la conoscenza futura della successione dei riferimenti.

Algoritmo 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 è 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*).

La sostituzione LRU associa a ogni pagina l'istante in cui è stata usata per l'ultima volta. Quando occorre sostituire una pagina, l'algoritmo LRU sceglie quella che non è stata usata per il periodo più lungo. Questa strategia costituisce l'algoritmo ottimale di sostituzione delle pagine con ricerca all'indietro nel tempo, anziché in avanti. Infatti, supponendo che S^R sia la successione inversa di una successione di riferimenti S , la frequenza di assenze di pagine per l'algoritmo OPT su S è uguale a quella per l'algoritmo LRU su S^R . Allo stesso modo, la frequenza di assenze di pagine per l'algoritmo LRU su S è uguale a quella per l'algoritmo OPT su S^R .

Il risultato dell'applicazione dell'algoritmo LRU alla successione dei riferimenti dell'esempio è il seguente:

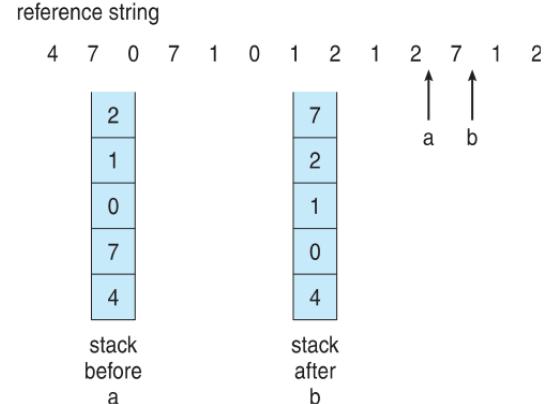


L'algoritmo LRU produce 12 assenze di pagine. Occorre notare che le prime cinque assenze di pagine sono le stesse della sostituzione ottimale. Quando si presenta il riferimento alla pagina 4, però, l'algoritmo LRU trova che, fra i tre blocchi di memoria, quello usato meno recentemente è della pagina 2. Quindi, l'algoritmo LRU sostituisce la pagina 2 senza sapere che sta per essere usata. Quando si verifica l'assenza della pagina 2, l'algoritmo LRU sostituisce la pagina 3, poiché, fra le tre pagine in memoria (0, 3, 4), la pagina 3 è quella usata meno recentemente. Nonostante questi problemi, la sostituzione LRU, con 12 assenze di pagine, è in ogni modo migliore della sostituzione FIFO, con 15 assenze di pagine.

Il criterio LRU si usa spesso come algoritmo di sostituzione delle pagine ed è considerato valido. Il problema principale riguarda la realizzazione della sostituzione stessa. Un algoritmo di sostituzione delle pagine LRU può richiedere una notevole assistenza da parte dell'architettura del sistema di calcolo. Il problema consiste nel determinare un ordine per i frame definito secondo il 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 del momento d'uso, 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 del momento d'uso nella tabella relativa a quella specifica 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. Questo schema implica una ricerca all'interno della tabella delle pagine per individuare la pagina usata meno recentemente (LRU), e una scrittura in memoria (nel campo del momento d'uso della tabella delle pagine) per ogni accesso alla memoria. I riferimenti temporali si devono mantenere anche quando, a seguito dello scheduling della CPU, si modificano le tabelle delle pagine. Occorre infine considerare il superamento della capacità del contatore (*overflow*).
- **Stack.** Un altro metodo per la realizzazione della sostituzione delle pagine LRU prevede la presenza di uno stack dei numeri delle pagine. 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, come illustrato nella figura di fianco. Poiché alcuni elementi si devono estrarre al centro dello stack, la miglior realizzazione è tramite una lista doppiamente linkata, con un puntatore in testa ed uno in coda.



Sostituzione delle pagine per approssimazione a LRU

Sono pochi i sistemi di calcolo che dispongono di un'architettura adatta a una vera sostituzione LRU delle pagine. Nei sistemi che non offrono tali caratteristiche specifiche si devono impiegare altri algoritmi di sostituzione delle pagine, ad 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'architettura 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.

Inizialmente, il sistema operativo azzera tutti i bit. Quando s'inizia l'esecuzione di un processo utente, l'architettura del sistema 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. E questa l'informazione alla base di molti algoritmi per la sostituzione delle pagine che approssimano LRU.

Algoritmo con reference bit supplementari

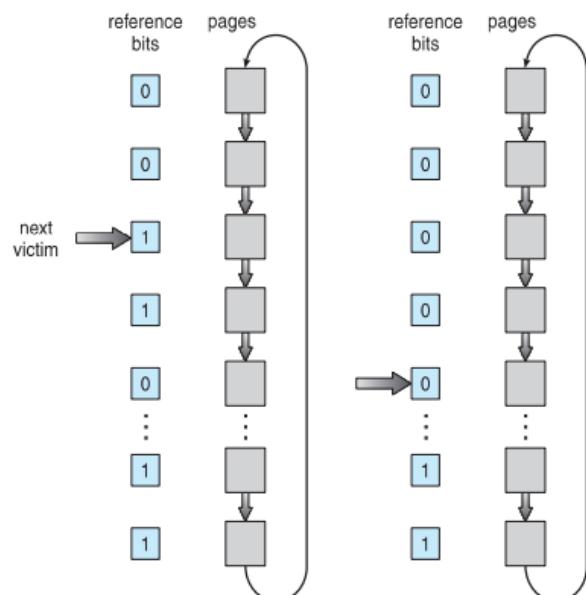
Ulteriori informazioni sull'ordinamento si possono ottenere registrando i bit di riferimento a intervalli regolari. È possibile conservare in una tabella in memoria una serie di bit per ogni pagina. A intervalli regolari, ad esempio di 100 millisecondi, un segnale d'interruzione del timer del sistema trasferisce il controllo al sistema operativo. Questo sposta il bit di riferimento per ciascuna pagina nel bit più significativo della sequenza, traslando gli altri bit a destra di 1 bit e scartando il bit meno significativo.

Il numero dei bit può ovviamente essere variato: si stabilisce secondo l'architettura disponibile per accelerarne al massimo la modifica. Nel caso limite tale numero si riduce a zero, lasciando soltanto il bit di riferimento e definendo un algoritmo noto come algoritmo con seconda chance.

Algoritmo second chance

L'algoritmo di base per la sostituzione con seconda chance è un algoritmo di sostituzione di tipo FIFO. 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 la selezione 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 offerta loro una seconda chance. Inoltre, se una pagina è usata abbastanza spesso, in modo che il suo bit di riferimento sia sempre impostato a 1, non viene mai sostituita.

Un metodo per realizzare l'algoritmo con seconda chance, detto anche a clock, è basato sull'uso di una coda circolare, in cui un puntatore 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 (vedi figura). Una volta trovata una pagina "vittima", la si sostituisce e si inserisce la nuova pagina nella coda circolare nella posizione corrispondente. 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.



Algoritmo enhanced second chance

L'algoritmo con seconda chance descritto precedentemente si può migliorare considerando i bit di riferimento e di modifica come una coppia ordinata, con cui si possono ottenere le seguenti quattro classi:

- 1) (0,0) né recentemente usato né modificato → migliore pagina da sostituire;

- 2) (0,1) non usato recentemente, ma modificato → la pagina non così buona poiché prima di essere sostituita deve essere scritta in memoria secondaria;
- 3) (1,0) usato recentemente ma non modificato → probabilmente la pagina sarà preso ancora usata;
- 4) (1,1) usato recentemente e modificato → probabilmente la pagina sarà presto ancora usata e dovrà essere scritta in memoria secondaria prima di essere sostituita.

Ogni pagina rientra in una di queste quattro classi. Alla richiesta di una sostituzione di pagina, si usa lo stesso schema impiegato nell'algoritmo a orologio, ma anziché controllare se la pagina puntata ha il bit di riferimento impostato a 1, si esaminano le classi cui la pagina appartiene e si sostituisce la prima pagina che si trova nella classe minima non vuota. Si noti che la coda circolare deve essere scandita più volte prima di trovare una pagina da sostituire.

La differenza principale tra questo algoritmo e il più semplice algoritmo a clock è che nel primo si dà la preferenza alle pagine modificate, al fine di ridurre il numero di I/O richiesti.

Sostituzione delle pagine basata su conteggio

Esistono molti altri algoritmi che si possono usare per la sostituzione delle pagine. Ad esempio, si potrebbe usare un contatore del numero dei riferimenti fatti a ciascuna pagina, e sviluppare i due seguenti schemi.

- **Algoritmo lease frequently used** (LFU, sostituzione delle pagine meno frequentemente usate): 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. Il punto debole di questo algoritmo è rappresentato dai casi in cui 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 conteggi a destra di un bit a intervalli regolari, formando un conteggio per Fuso medio con esponente decrescente.
- **Algoritmo most frequently used** (MFU, sostituzione delle pagine più frequentemente usate): 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.

Algoritmi Page Buffering

Oltre a uno specifico algoritmo, per la sostituzione delle pagine si usano spesso anche altre procedure; ad esempio, i sistemi hanno generalmente una pool di frame liberi. Quando si verifica un'assenza di pagina, si sceglie innanzitutto un frame vittima, ma prima che sia scritta in memoria secondaria, si trasferisce la pagina richiesta in un frame libero del gruppo. Questa procedura permette al processo di ricominciare al più presto, senza attendere che la pagina vittima sia scritta in memoria secondaria. Quando nel seguito si scrive la vittima in memoria secondaria, si aggiunge il suo frame alla pool dei frame liberi.

Quest'idea si può estendere conservando una lista delle pagine modificate: ogniqualvolta il dispositivo di paginazione è inattivo, si sceglie una pagina modificata, la si scrive nel disco e si reimposta il suo bit di modifica. Questo schema aumenta la probabilità che, al momento della selezione per la sostituzione, la pagina non abbia subito modifiche e non debba essere scritta in memoria secondaria.

Anche un'altra modifica prevede l'uso di un gruppo di frame liberi ma, in questo caso, per ricordare quale pagina era contenuta in ciascun frame. Poiché quando si scrive il contenuto di un frame in un disco tale contenuto non cambia, se è necessaria, la vecchia pagina è ancora utilizzabile direttamente dalla pool dei frame liberi, prima che sia riusato quel frame. In questo caso non è necessario alcun I/O. Se si verifica un'assenza di pagina occorre controllare se la pagina richiesta si trova nel gruppo dei frame liberi; se non c'è si deve individuare un frame libero e trasferirvi la pagina.

Applicazioni e sostituzione della pagina

In taluni casi, le applicazioni che accedono ai dati tramite la memoria virtuale del sistema operativo non conseguono prestazioni migliori di quelle che il sistema, senza impiegare alcun buffer, potrebbe offrire. Si pensi, quale esempio tipico, a una base di dati che gestisce la memoria e il buffer dell'I/O in modo autonomo. Applicazioni come questa capiscono il funzionamento della memoria e del disco che occupano meglio di quanto possa fare un sistema operativo, che applica algoritmi adatti a un uso generale. Se il sistema operativo adotta un buffer per l'I/O, e così pure fa l'applicazione, la quantità di memoria necessaria per l'I/O sarà inutilmente raddoppiata.

Per risolvere tali problemi, alcuni sistemi operativi permettono a certi programmi di utilizzare una partizione del disco come un array sequenziale di blocchi logici, senza ricorrere alle strutture di dati del file system. Un simile array è anche detto **raw disk** (disco di basso livello), e il relativo I/O è denominato **raw I/O** (I/O di basso livello). Il disco di basso livello salta tutti i servizi del file system, come la paginazione su richiesta dei file in ingresso e in uscita, i lock dei file, il prefetching, l'allocazione dello spazio, i nomi dei file e le directory. Si noti come, sebbene alcune applicazioni siano più efficienti nel gestire i propri servizi specifici di memorizzazione sul disco di basso livello, quasi tutte hanno una resa migliore quando operano con i servizi regolari del file system.

Allocazione dei frame

A questo punto occorre stabilire un criterio per l'allocazione della memoria libera ai diversi processi. Come esempio, è possibile considerare un caso in cui 93 frame liberi si debbano assegnare a due processi.

Il caso più semplice di memoria virtuale è il sistema con utente singolo. Si consideri un sistema monoutente che disponga di 128 KB di memoria, con pagine di 1 KB. Complessivamente sono presenti 128 frame. Il sistema operativo può occupare 35 KB, lasciando 93 frame per il processo utente. In condizioni di paginazione su richiesta pura, tutti i 93 blocchi di memoria sono inizialmente posti nella lista dei frame liberi. Quando comincia l'esecuzione, il processo utente genera una sequenza di page fault. Le prime 93 pagine assenti ricevono i frame liberi dalla lista. Una volta esaurita quest'ultima, per stabilire quale tra le 93 pagine presenti in memoria si debba sostituire con la novantaquattresima, si può usare un algoritmo di sostituzione delle pagine. Terminato il processo, si reinseriscono i 93 frame nella lista dei frame liberi.

Questa strategia è semplice, ma può subire molte variazioni. Si può richiedere che il sistema operativo assegna tutto lo spazio richiesto dalle proprie strutture dati attingendo dalla lista dei frame liberi. Quando questo spazio è inutilizzato dal sistema operativo può essere sfruttato per la paginazione utente. Un'altra variante prevede di riservare sempre tre frame liberi, in modo che quando si verifica un'assenza di pagina sia disponibile un frame libero in cui trasferire la pagina richiesta. Mentre ha luogo il trasferimento della pagina, si può fare una sostituzione, la pagina coinvolta viene poi scritta nel disco mentre il processo utente continua l'esecuzione.

Numero minimo di frame

Le strategie di allocazione dei frame sono soggette a parecchi vincoli. Non si possono assegnare più frame di quanti siano disponibili, sempre che non vi sia condivisione di pagine. Inoltre è necessario assegnare almeno un numero minimo di frame. Naturalmente, col diminuire del numero dei frame allocati a ciascun processo aumenta la frequenza dei page fault, con conseguente rallentamento dell'esecuzione del processo. Esaminiamo quest'ultimo requisito in maggiore dettaglio.

Una delle ragioni per allocare sempre un numero minimo di frame è legata alle prestazioni. Ovviamente, al decrescere del numero dei frame allocati a ciascun processo aumenta la frequenza di mancanza di pagina, con conseguente ritardo dell'esecuzione dei processi. Inoltre va ricordato che, quando si verifica un page fault prima che sia stata completata l'esecuzione di un'istruzione, quest'ultima deve essere riavviata. Di

conseguenza, i frame disponibili devono essere in numero sufficiente per contenere tutte le pagine cui ogni singola istruzione può far riferimento.

Si consideri, ad esempio, un calcolatore in cui tutte le istruzioni di riferimento alla memoria hanno solo un indirizzo di memoria; in questo caso occorre almeno un frame per l'istruzione e uno per il riferimento alla memoria. Inoltre se è ammesso un indirizzamento indiretto a un livello (come nel caso di un'istruzione load presente nella pagina 16 che può far riferimento a un indirizzo della pagina 0, che costituisce a sua volta un riferimento indiretto alla pagina 23) la paginazione richiede allora almeno tre blocchi di memoria per ogni processo. Si consideri che cosa accadrebbe nel caso di un processo che disponga di due soli frame.

Il numero minimo di frame per ciascun processo è definito dall'architettura, mentre il numero massimo è definito dalla quantità di memoria fisica disponibile.

Algoritmi di allocazione

Il modo più semplice per suddividere m frame tra n processi è quello per cui a ciascuno si dà una parte uguale, $\frac{m}{n}$ frame. Dati 93 frame e 5 processi, ogni processo riceve 18 frame. I tre frame lasciati liberi si potrebbero usare come pool dei frame liberi. Questo schema è chiamato **equal allocation** (allocazione uniforme).

Un'alternativa consiste nel riconoscere che diversi processi hanno bisogno di quantità di memoria diverse. Si consideri un sistema con frame di 1 KB. Se un piccolo processo utente di 10 KB e una base di dati interattiva di 127 KB sono gli unici due processi in esecuzione su un sistema con 62 frame liberi, non ha senso allocare a ciascun processo 31 frame. Al processo utente non ne servono più di 10, quindi gli altri 21 sarebbero semplicemente sprecati.

Per risolvere questo problema è possibile ricorrere alla **proportional allocation** (allocazione proporzionale), secondo cui la memoria disponibile si assegna a ciascun processo secondo la propria dimensione. Si supponga che s_i sia la dimensione della memoria virtuale per il processo p_i .

Si definisce la seguente quantità:

$$S = \sum_i s_i$$

Quindi, se il numero totale dei frame disponibili è m , al processo p_i si assegnano a_i frame, dove

$$a_i \approx \frac{s_i}{S} * m$$

Naturalmente è necessario scegliere ciascun a_i in modo che sia un intero maggiore del numero minimo di frame richiesti dalla struttura della serie di istruzioni di macchina e in modo che la somma di tutti gli a_i non sia maggiore di m .

Usando la proportional allocation, per suddividere 62 frame tra due processi, uno di 10 e uno di 127 pagine, si assegnano rispettivamente 4 e 57 frame, infatti: $\frac{10}{137} * 62 \approx 4$ e $\frac{127}{137} * 62 \approx 57$.

In questo modo entrambi i processi condividono i frame disponibili secondo le rispettive necessità, e non in modo uniforme.

Sia nell'allocatione uniforme sia in quella proporzionale, l'allocatione a ogni processo può variare rispetto al livello di multiprogrammazione. Se tale livello aumenta, ciascun processo perde alcuni frame per fornire la memoria necessaria per il nuovo processo. D'altra parte, se il livello di multiprogrammazione diminuisce, i frame allocati al processo allontanato si possono distribuire tra quelli che restano.

Occorre notare che sia con l'allocatione uniforme sia con l'allocatione proporzionale, un processo a priorità elevata è trattato come un processo a bassa priorità anche se, per definizione, si vorrebbe che al processo con elevata priorità fosse allocata più memoria per accelerarne l'esecuzione, a discapito dei processi a bassa priorità. Un soluzione prevede l'uso di uno schema di allocatione proporzionale in cui il rapporto dei

frame non dipende dalle dimensioni relative dei processi, ma dalle priorità degli stessi oppure da una combinazione di dimensioni e priorità (**priority allocation**).

Allocazione globale e allocazione locale

Un altro importante fattore che riguarda il modo in cui si assegnano i frame ai vari processi è la sostituzione delle pagine. Nei casi in cui vi siano più processi in competizione per i frame, gli algoritmi di sostituzione delle pagine si possono classificare in due categorie generali: **sostituzione globale** (*global replacement*) e **sostituzione locale** (*local replacement*). La sostituzione globale permette che per un processo si scelga un frame per la sostituzione dall'insieme di tutti i frame, anche se quel frame è correntemente allocato a un altro processo; un processo può dunque sottrarre un frame a un altro processo. La sostituzione locale richiede invece che per ogni processo si scelga un frame solo dal proprio insieme di frame.

Si consideri ad esempio uno schema di allocazione che, per una sostituzione a favore dei processi ad alta priorità, permetta di sottrarre frame ai processi a bassa priorità. Per un processo si può stabilire una sostituzione che attinga tra i suoi frame oppure tra quelli di qualsiasi processo con priorità minore. Questo metodo permette a un processo ad alta priorità di aumentare il proprio livello di allocazione dei frame a discapito del processo a bassa priorità.

Con la strategia di sostituzione locale, il numero di blocchi di memoria assegnati a un processo non cambia. Con la sostituzione globale, invece, può accadere che per un certo processo si selezionino solo frame allocati ad altri processi, aumentando così il numero di frame assegnati a quel processo, purché per altri non si scelgano per la sostituzione i propri frame.

L'algoritmo di sostituzione globale risente di un problema: un processo non può controllare la propria frequenza di assenze di pagine {page-fault rate}, infatti l'insieme di pagine che si trova in memoria per un processo non dipende solo dal comportamento di paginazione di quel processo, ma anche dal comportamento di paginazione di altri processi. Quindi, lo stesso processo può comportarsi in modi piuttosto diversi, ad esempio impiegando 0,5 secondi per un'esecuzione e 10,3 secondi per quella successiva, a causa di circostanze esterne. Con l'algoritmo di sostituzione locale questo problema non si presenta. Infatti l'insieme di pagine in memoria per un processo subisce l'effetto del comportamento di paginazione di quel solo processo. Dal canto suo, la sostituzione locale può limitare un processo, non rendendogli disponibili altre pagine di memoria meno usate. Generalmente, la sostituzione globale genera una maggiore produttività del sistema, e perciò è il metodo più usato.

Accesso non uniforme alla memoria

Fino a questo momento trattando il tema della memoria virtuale abbiamo assunto che le diverse parti della memoria centrale funzionassero tutte allo stesso modo, o almeno che vi si potesse accedere con le stesse modalità. In molti sistemi informatici non è così. Spesso in sistemi con processori multipli un certo processore può accedere ad alcune regioni della memoria più rapidamente rispetto ad altre. Tali differenze nelle prestazioni sono causate dalla modalità di interconnessione tra processori e memoria all'interno del sistema. Frequentemente un tale sistema è costituito da diverse schede madri, ognuna contenente più processori e una parte di memoria. Le schede sono connesse in vari modi, a partire dai bus di sistema fino a connessioni di rete ad alta velocità come *InfiniBand*. Come forse ci si aspetta, i processori di una particolare scheda possono accedere alla memoria della scheda stessa in meno tempo rispetto a quello necessario per accedere ad altre schede del sistema. I sistemi nei quali i tempi di accesso alla memoria variano in modo significativo sono generalmente detti sistemi con **accesso non uniforme alla memoria** (*non-uniform memory access, NUMA*) e, senza eccezioni, sono più lenti dei sistemi nei quali memoria e processori risiedono sulla stessa scheda madre.

Le decisioni su quali frame di pagina memorizzare e dove memorizzarli possono condizionare in modo significativo le prestazioni nei sistemi NUMA. Se, in sistemi del genere ignorassimo le diversità nei tempi di accesso alla memoria, i processori potrebbero dover aspettare molto più a lungo per accedere alla

memoria rispetto al caso in cui gli algoritmi di allocazione della memoria siano modificati per tenere in conto il NUMA. Analoghe modifiche devono essere apportate anche al sistema di scheduling. L'obiettivo di questi cambiamenti è quello di allocare i frame di memoria "il più vicino possibile" al processore sul quale il processo è in esecuzione, dove per "vicino" si intende "con latenza minima", ovvero, di solito, sulla stessa scheda della CPU.

I cambiamenti negli algoritmi consistono nel fatto che lo scheduler tiene traccia dell'ultimo processore sul quale ciascun processo è stato eseguito. Se lo scheduler prova a pianificare ciascun processo sul suo processore precedente, e se il sistema di gestione della memoria prova ad allocare frame per il processo vicino al processore sul quale sta per essere mandato in esecuzione, si otterrà un incremento dei successi di cache e una diminuzione del tempo di accesso alla memoria.

La questione diventa ancora più complicata con l'aggiunta dei thread. Ad esempio, un processo con molti thread in esecuzione potrebbe vedere quei thread pianificati su differenti schede del sistema. Come viene allocata la memoria in questo caso? Solaris risolve il problema creando una entità **Igroup** (gruppi di località) nel kernel. Ogni Igroup raccoglie i processori e la memoria vicini fra loro. In realtà gli Igroup sono ordinati gerarchicamente sulla base del periodo di latenza tra i gruppi. Solaris tenta di pianificare tutti i thread di un processo e di allocare tutta la memoria di un processo nell'ambito di un solo Igroup. Se una tale soluzione non è possibile, per il resto delle risorse necessarie vengono utilizzati gli Igroup più vicini, in modo da minimizzare la latenza complessiva della memoria e massimizzare il grado di successo della cache del processore.

Thrashing (paginazione degenere)

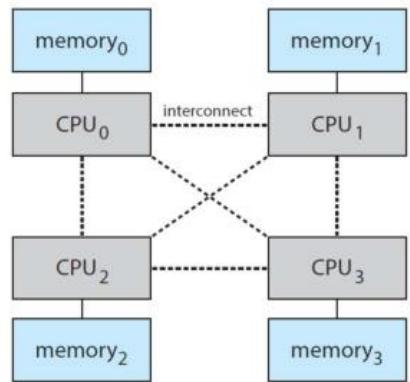
Se il numero dei frame allocati a un processo con priorità bassa diviene inferiore al numero minimo richiesto dall'architettura del calcolatore, occorre sospendere l'esecuzione del processo, e quindi togliere le pagine restanti, liberando tutti i frame allocati. Questa operazione introduce un livello intermedio di scheduling per la gestione dell'entrata e dell'uscita dei processi in memoria centrale.

Infatti, si consideri un qualsiasi processo che non disponga di un numero di frame "sufficiente". Anche se tecnicamente si può ridurre al valore minimo il numero dei frame allocati, esiste un certo (in generale grande) numero di pagine in uso attivo. Se non dispone di questo numero di frame, il processo accusa immediatamente un page fault. A questo punto si deve sostituire qualche pagina; ma, poiché tutte le sue pagine sono in uso attivo, si deve sostituire una pagina che sarà immediatamente necessaria, e di conseguenza si verificano subito parecchi page fault. Il processo continua a subire page fault, facendo sostituire pagine che saranno immediatamente trattate come assenti e dovranno essere riprese.

Questa intensa quanto degenere paginazione (nota come *thrashing*) si verifica quando si spende più tempo per la paginazione che per l'esecuzione dei processi.

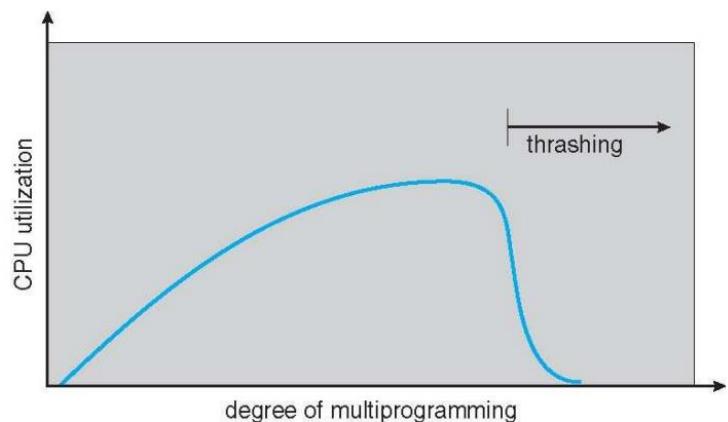
Cause del thrashing

La degenerazione dell'attività di paginazione causa parecchi problemi di prestazioni. Si consideri il seguente scenario, basato sul comportamento effettivo dei primi sistemi di paginazione. Il sistema operativo vigila sull'utilizzo della CPU. Se questo è basso, aumenta il grado di multiprogrammazione introducendo un nuovo processo. Si usa un algoritmo di sostituzione delle pagine globale, che sostituisce le pagine senza tener conto del processo al quale appartengono. Per ora si ipotizzi che un processo entri in una nuova fase d'esecuzione e richieda più frame; se ciò si verifica si ha una serie di page fault, cui segue la sottrazione di nuove pagine ad altri processi. Questi processi hanno però bisogno di quelle pagine e quindi subiscono anch'essi dei page fault, con conseguente sottrazione di pagine ad altri processi. Per effettuare il



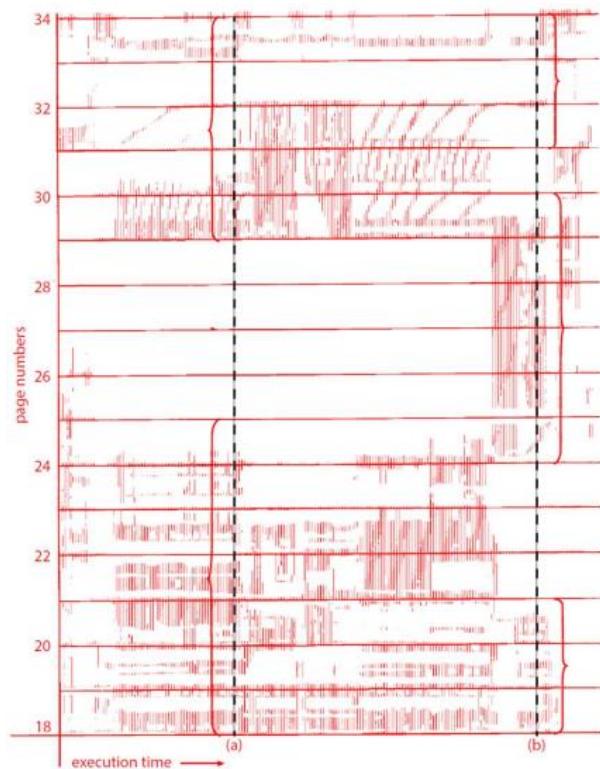
caricamento e lo scaricamento delle pagine per questi processi si deve usare il dispositivo di paginazione. Mentre si mettono i processi in coda per il dispositivo di paginazione, la coda dei processi pronti per l'esecuzione si svuota, quindi l'utilizzo della CPU diminuisce.

Lo scheduler della CPU rileva questa riduzione dell'utilizzo della CPU e aumenta il grado di multiprogrammazione. Si tenta di avviare il nuovo processo sottraendo pagine ai processi in esecuzione, causando ulteriori assenze di pagine e allungando la coda per il dispositivo di paginazione. L'utilizzo della CPU scende ulteriormente, e lo scheduler della CPU tenta di aumentare ancora il grado di multiprogrammazione. L'attività di paginazione è degenerata in una situazione patologica che fa precipitare la produttività del sistema. La frequenza delle assenze di pagine aumenta in modo impressionante, e di conseguenza aumenta il tempo effettivo d'accesso alla memoria. I processi non svolgono alcun lavoro, poiché si sta spendendo tutto il tempo nell'attività di paginazione (vedi figura). Aumentando il grado di multiprogrammazione aumenta anche l'utilizzo della CPU, anche se più lentamente, fino a raggiungere un massimo. Se a questo punto si aumenta ulteriormente il grado di multiprogrammazione, l'attività di paginazione degenera e fa crollare l'utilizzo della CPU. In questa situazione, per aumentare l'utilizzo della CPU occorre ridurre il grado di multiprogrammazione.



Gli effetti di questa situazione si possono limitare usando un **algoritmo di sostituzione locale**, o **algoritmo di sostituzione per priorità**. Con la sostituzione locale, se un processo ricade nell'attività di paginazione degenera, non può sottrarre frame a un altro processo e quindi provocarne a sua volta la degenerazione. Le pagine si sostituiscono tenendo conto del processo di cui fanno parte. Tuttavia, se i processi la cui attività di paginazione degenera rimangono nella coda d'attesa del dispositivo di paginazione per la maggior parte del tempo. Il tempo di servizio medio di un'eccezione di pagina mancante aumenta a causa dell'allungamento medio della coda d'attesa del dispositivo di paginazione. Di conseguenza, il tempo effettivo d'accesso al dispositivo di paginazione aumenta anche per gli altri processi.

Per evitare il verificarsi di queste situazioni, occorre fornire a un processo tutti i frame di cui necessita. Per cercare di sapere quanti frame "servano" a un processo si impiegano diverse tecniche. Il modello dell'insieme di lavoro (working-set), trattato nel sottoparagrafo successivo, comincia osservando quanti siano i frame che un processo sta effettivamente usando. Questo metodo definisce il **modello di località** (*locality model*) d'esecuzione del processo. Il modello di località stabilisce che un processo, durante la sua esecuzione, si sposta di località in località. Una località è un insieme di pagine usate attivamente, com'è illustrato nella figura a destra. Generalmente un programma è formato di parecchie località diverse, che sono sovrapponibili.



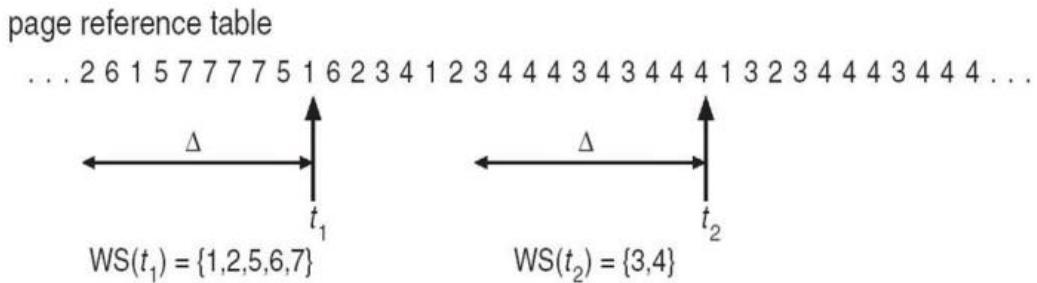
Ad esempio, quando s'invoca una procedura, essa definisce una nuova località. In questa località si fanno riferimenti alla memoria per le istruzioni della procedura, per le sue variabili locali e per un sottoinsieme delle variabili globali. Quando la procedura termina, il processo lascia questa località, poiché le variabili locali e le istruzioni della procedura non sono più usate attivamente.

Le località sono definite dalla struttura del programma e dalle relative strutture dati. Il modello di località sostiene che tutti i programmi mostrino questa struttura di base di riferimenti alla memoria. Si noti che il modello di località è il principio non dichiarato sottostante all'analisi fin qui svolta sul caching. Se gli accessi ai vari tipi di dati fossero casuali, anziché strutturati in località, il caching sarebbe inutile.

Si supponga di allocare a un processo un numero di frame sufficiente per sistemare le sue località attuali. Finché tutte queste pagine non si trovano in memoria, si verificano le assenze delle pagine relative a tali località; quindi, finché le località non vengono modificate, non hanno luogo altre assenze di pagine. Se si assegnano meno frame rispetto alla dimensione della località attuale, la paginazione del processo degenera, poiché non si possono tenere in memoria tutte le pagine che il processo sta usando attivamente.

Modello working set

Come già accennato, il modello dell'insieme di lavoro (**working-set model**) è basato sull'ipotesi di località. Questo modello usa un parametro, Δ , per definire una **working set window** (finestra dell'insieme di lavoro). L'idea consiste nell'esaminare i più recenti Δ riferimenti alle pagine. L'insieme di pagine nei più recenti Δ riferimenti è il **working set**:



Se una pagina è in uso attivo si trova nell'insieme di lavoro; se non è più usata esce dall'insieme di lavoro Δ unità di tempo dopo il suo ultimo riferimento. Quindi, l'insieme di lavoro non è altro che un'approssimazione della località del programma.

Ad esempio, data la successione di riferimenti alla memoria mostrata nella figura precedente, se $\Delta = 10$ riferimenti alla memoria, l'insieme di lavoro all'istante t_1 è $\{1,2,5,6,7\}$. All'istante t_2 l'insieme di lavoro è diventato $\{3,4\}$.

La precisione dell'insieme di lavoro dipende dalla scelta del valore di Δ . Se Δ è troppo piccolo non include l'intera località, se è troppo grande può sovrapporre più località. Al limite, se Δ è infinito l'insieme di lavoro coincide con l'insieme di pagine cui il processo fa riferimento durante la sua esecuzione.

La caratteristica più importante dell'insieme di lavoro è la sua dimensione. Calcolando la dimensione dell'insieme di lavoro, WSS_i , per ciascun processo p_i del sistema, si può determinare la richiesta totale di frame, cioè $D = \sum_i WSS_i$.

Ogni processo usa attivamente le pagine del proprio insieme di lavoro. Quindi, il processo i necessita di WSS_i frame. Se la richiesta totale è maggiore del numero totale di frame liberi ($D > m$), si verifica il thrashing (la paginazione degenera), poiché alcuni processi non dispongono di un numero sufficiente di frame.

Una volta scelto D , l'uso del modello dell'insieme di lavoro è abbastanza semplice. Il sistema operativo controlla l'insieme di lavoro di ogni processo e gli assegna un numero di frame sufficiente, rispetto alle dimensioni del suo insieme di lavoro. Se i frame ancora liberi sono in numero sufficiente, si può iniziare un

altro processo. Se la somma delle dimensioni degli insiemi di lavoro aumenta, superando il numero totale dei frame disponibili, il sistema operativo individua un processo da sospendere. Scrive in memoria secondaria le pagine di quel processo e assegna i propri frame ad altri processi. Il processo sospeso può essere ripreso successivamente. Questa strategia impedisce la paginazione degenera, mantenendo il grado di multiprogrammazione più alto possibile, quindi ottimizza l'utilizzo della CPU.

Poiché la finestra dell'insieme di lavoro è una finestra dinamica, la difficoltà insita in questo modello consiste nel tener traccia degli elementi che compongono l'insieme di lavoro stesso. A ogni riferimento alla memoria, a un'estremità appare un riferimento nuovo e il riferimento più vecchio fuoriesce dall'altra estremità. Una pagina si trova nell'insieme di lavoro se esiste un riferimento a essa in qualsiasi punto della finestra dell'insieme di lavoro.

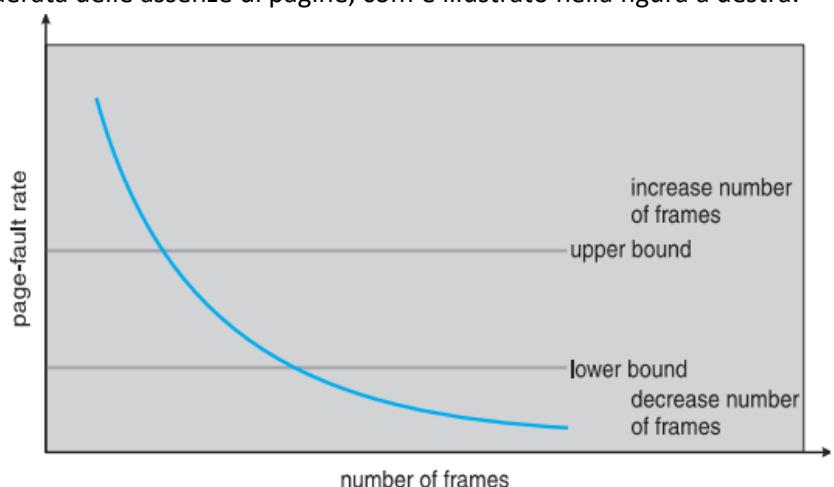
Si supponga, ad esempio, $\Delta = 10000$ riferimenti e che sia possibile ottenere un segnale d'interruzione dal timer ogni 5000 riferimenti. Quando si verifica uno di tali segnali d'interruzione, i valori dei bit di riferimento di ciascuna pagina vengono copiati e poi azzerati. Così, quando si verifica un'assenza di pagina è possibile esaminare il bit di riferimento corrente e i 2 bit in memoria per stabilire se una pagina sia stata usata entro gli ultimi 10000-15000 riferimenti. Se lo è stata, almeno uno di questi bit è attivo. Se non lo è stata, questi bit sono tutti inattivi. Le pagine con almeno un bit attivo si considerano appartenenti all'insieme di lavoro. Occorre notare che questo schema non è del tutto preciso, poiché non è possibile stabilire dove si è verificato un riferimento entro un intervallo di 5000. L'incertezza si può ridurre aumentando il numero dei bit cronologici e la frequenza dei segnali d'interruzione, ad esempio 10 bit e un'interruzione ogni 1000 riferimenti. Tuttavia, il costo per servire questi segnali d'interruzione più frequenti aumenta in modo corrispondente.

Frequenza dei page fault

La strategia basata su **page fault frequency** (PFF, frequenza delle assenze di pagine) è più diretta dell'approccio working set size. Lo scopo è stabilire un tasso "accettabile" di PFF.

Il problema specifico è la prevenzione della paginazione degenera. La frequenza delle assenze di pagine in tale situazione è alta, ed è proprio questa che si deve controllare. Se la frequenza delle assenze di pagine è eccessiva, significa che il processo necessita di più frame. Analogamente, se la frequenza delle assenze di pagine è molto bassa, il processo potrebbe disporre di troppi frame. Si può fissare un limite inferiore e un limite superiore per la frequenza desiderata delle assenze di pagine, com'è illustrato nella figura a destra.

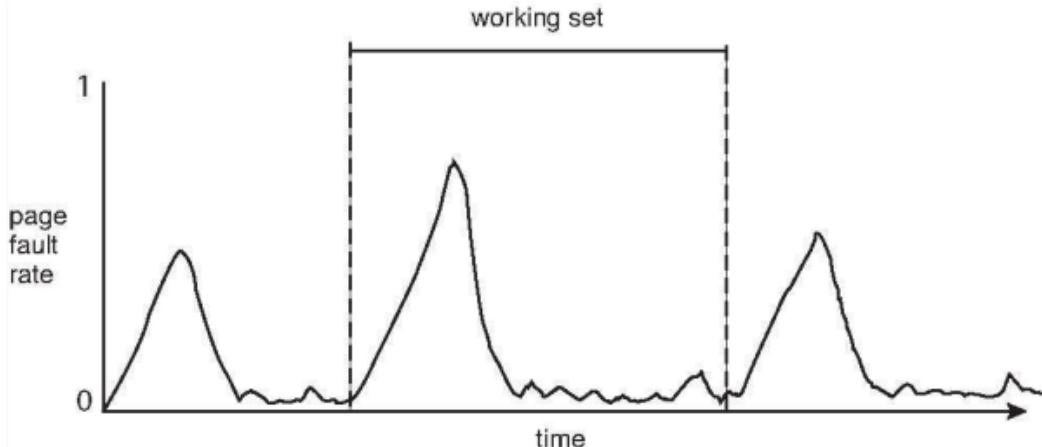
Se la frequenza effettiva delle assenze di pagine per un processo oltrepassa il limite superiore (*upper bound*), occorre allocare a quel processo un altro frame; se la frequenza scende sotto il limite inferiore (*lower bound*), si sottrae un frame a quel processo. Quindi, per prevenire la paginazione degenera, si può misurare e controllare direttamente la frequenza delle assenze di pagine.



Come nel caso dell'insieme di lavoro, può essere necessaria la sospensione di un processo. Se la frequenza delle assenze di pagine aumenta e non ci sono frame disponibili, occorre selezionare un processo e sosponderlo. I frame liberati si distribuiscono ai processi con elevate frequenze di assenze di pagine.

Working set e PFF. Vi è una relazione diretta tra l'insieme di lavoro di un processo e la frequenza di pagine mancanti. L'insieme di lavoro di un processo cambia nel corso del tempo, mentre i riferimenti ai dati e alcune parti del codice sono dislocati dall'una all'altra posizione.

Assumendo memoria sufficiente a contenere l'insieme di lavoro di un processo (ossia, a evitare che la paginazione di un processo degeneri), l'andamento delle pagine mancanti oscillerà, in un dato periodo di tempo, tra picchi e valli. Questa tendenza è illustrata dal seguente grafico:



Un picco nella frequenza di pagine mancanti si verifica alla richiesta di paginazione relativa a una nuova località. Tuttavia, una volta che l'insieme di lavoro interessato sia in memoria, la frequenza di pagine mancanti precipita. Quando il processo entra in un nuovo insieme di lavoro, la frequenza si impenna ancora una volta verso un picco; quando il nuovo insieme di lavoro è caricato in memoria, la frequenza crolla nuovamente. L'intervallo di tempo tra l'inizio di un picco e quello del picco successivo descrive la transizione da un insieme di lavoro a un altro.

Allocazione di memoria nel kernel

Quando un processo eseguito in modalità utente necessita di memoria aggiuntiva, le pagine sono allocate dalla lista dei frame disponibili che il kernel mantiene. Per formare questa lista, si applica in genere uno degli algoritmi di sostituzione delle pagine esaminati nel paragrafo [Sostituzione delle pagine](#); molto verosimilmente, la lista conterrà pagine non utilizzate sparse per tutta la memoria, come abbiamo visto in precedenza. Va inoltre ricordato che, se un processo utente richiede un solo byte di memoria, si ottiene frammentazione interna, poiché al processo viene garantito un intero frame.

Il kernel, tuttavia, per allocare la propria memoria, attinge spesso a una riserva di memoria libera differente dalla lista usata per soddisfare i processi ordinari in modalità utente.

Questo avviene principalmente per due motivi:

- 1) Il kernel richiede memoria per strutture dati dalle dimensioni variabili; alcune di loro corrispondono a meno di una pagina. Deve quindi fare un uso oculato della memoria, tentando di contenere al minimo gli sprechi dovuti alla frammentazione. Questo fattore è di particolare rilevanza, se si considera che, in molti sistemi operativi, il codice o i dati del kernel non sono soggetti a paginazione.
- 2) Le pagine allocate ai processi in modalità utente non devono necessariamente essere contigue nella memoria fisica. Alcuni dispositivi, però, interagiscono direttamente con la memoria fisica, senza il vantaggio dell'interfaccia della memoria virtuale; di conseguenza, possono richiedere memoria che risieda in pagine fisicamente contigue.

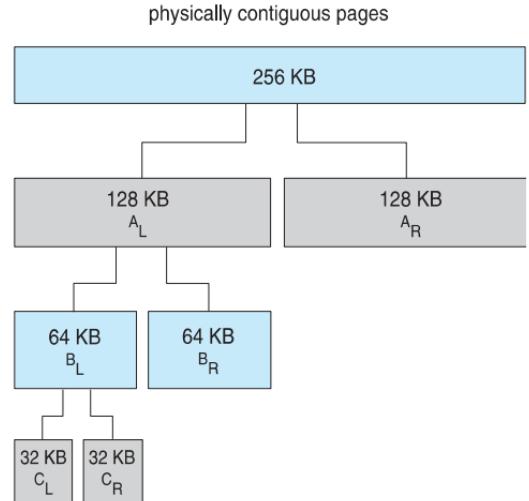
Ci sono due strategie principali per la gestione della memoria libera assegnata ai processi del kernel: il cosiddetto "sistema buddy" e l'allocazione slab (a lastre).

Sistema buddy

Il “sistema buddy” [sistema gemellare) utilizza un segmento di grandezza fissa per l’allocazione della memoria, contenente pagine fisicamente contigue. La memoria è assegnata mediante un cosiddetto **allocatore-potenza-di-2**, che alloca memoria in unità di dimensioni pari a potenze di 2 (4 KB, 8 KB, 16 KB, e via di seguito). Le differenti quantità richieste sono arrotondate alla successiva potenza di 2. Ad esempio, una richiesta di 11 KB viene soddisfatta con un segmento di 16 KB.

Consideriamo un esempio semplice e ipotizziamo che la grandezza di un segmento di memoria sia inizialmente di 256 KB e che il kernel richieda 21 KB di memoria. In primo luogo,

il segmento è suddiviso in due buddy (“gemelli”), che chiameremo A_L e A_R , ciascuno dei quali misura 128 KB. Uno di questi è ulteriormente dimezzato in 2 buddy da 64 KB, diciamo B_L e B_R . Poiché la minima potenza di 2 che supera 21 KB è pari a 32 KB, occorre suddividere ancora B_L , oppure B_R , in due buddy la cui dimensione è 32 KB; chiamiamoli C_L e C_R . Uno di loro è il segmento scelto per soddisfare la richiesta di 21 KB. Lo schema è illustrato nella figura, dove C_L è il segmento allocato per la richiesta di 21 KB.



Questo sistema offre il vantaggio di poter congiungere rapidamente buddy adiacenti per formare segmenti più grandi tramite una tecnica nota come **fusion** (*coalescing*). L’ovvio inconveniente di questo sistema è che l’arrotondamento per eccesso a una potenza di 2 può facilmente generare frammentazione all’interno dei segmenti allocati.

Slab allocator

Una seconda strategia per assegnare la memoria del kernel è detta **allocazione slab**. Uno **slab** (lastra) è composto da una o più pagine fisicamente contigue. Una **cache** consiste di uno o più slab. Vi è una sola cache per ciascuna categoria di struttura dati del kernel: una cache dedicata alla struttura dati che rappresenta i descrittori dei processi, una dedicata agli oggetti che rappresentano i file, un’altra per i semafori, e così via. Ogni cache è popolata da oggetti, istanze della struttura dati del kernel rappresentata dalla cache.

L’algoritmo di allocazione degli slab utilizza le cache per memorizzare oggetti del kernel. Quando si crea una cache, un certo numero di oggetti, inizialmente dichiarati liberi, viene assegnato alla cache. Questo numero dipende dalla grandezza dello slab associato alla cache. Per esempio, uno slab di 12 KB (formato da tre pagine contigue di 4 KB) potrebbe contenere sei oggetti di 2 KB ciascuno. Al principio, tutti gli oggetti nella cache sono contrassegnati come **free**. Quando una struttura dati del kernel ha bisogno di un oggetto, per soddisfare la richiesta dell’allocatore può selezionare dalla cache qualunque oggetto libero; l’oggetto tratto dalla cache è quindi contrassegnato come **used**.

Consideriamo una situazione in cui il kernel richieda all’allocatore delle lastre la memoria per un oggetto rappresentante un descrittore dei processi. Nei sistemi Linux, un descrittore dei processi ha tipo struct *task_struct*, che richiede circa 1,7 KB di memoria. Quando il kernel di Linux crea un nuovo task, richiede alla propria cache la memoria necessaria per l’oggetto di tipo struct *task_struct*. La cache darà corso alla richiesta impiegando un oggetto di questo tipo che sia già stato allocato in uno slab e rechi il contrassegno free. In Linux uno slab può essere in uno dei seguenti stati:

- 1) **full** → tutti gli oggetti dello slab sono contrassegnati come used
- 2) **empty** → Tutti gli oggetti dello slab sono contrassegnati come free
- 3) **partial** → Lo slab contiene un mix di free e used.

Lo slab allocator, per soddisfare una richiesta, tenta in primo luogo di estrarre un oggetto libero da uno slab parzialmente occupato; se non ne esistono, assegna un oggetto libero da uno slab vuoto; in mancanza di slab empty, crea un nuovo slab da pagine fisiche contigue e la alloca a una cache; da tale slab attinge la memoria da allocare all'oggetto. Lo slab allocator offre, essenzialmente, due vantaggi:

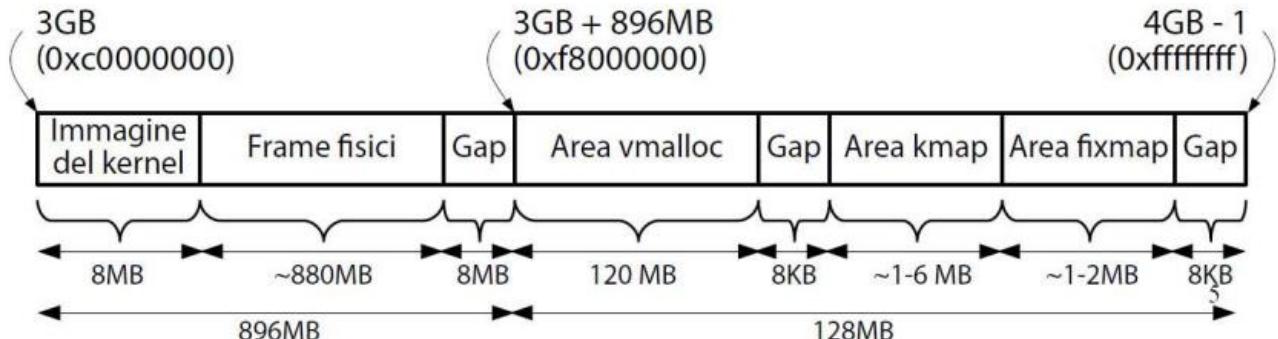
- 1) Annulla lo spreco di memoria derivante da frammentazione. La frammentazione cessa di costituire un problema, poiché ogni struttura dati del kernel ha una cache associata; ciascuna delle cache è composta da un numero variabile di slab, suddivisi in spezzoni di grandezza pari a quella degli oggetti rappresentati. Pertanto, quando il kernel esige memoria per un oggetto, l'allocatore di slab restituisce la quantità esatta di memoria necessaria per rappresentare l'oggetto.
- 2) Le richieste di memoria possono essere soddisfatte rapidamente. La tecnica di allocazione degli slab e si rivela particolarmente efficace quando, nella gestione della memoria, gli oggetti sono frequentemente allocati e deallocati, come spesso accade con le richieste del kernel. In termini di tempo, allocare e deallocare memoria può essere un processo dispendioso. Tuttavia, gli oggetti sono creati in anticipo e possono dunque essere allocati rapidamente dalla cache. Inoltre, quando il kernel rilascia un oggetto di cui non ha più bisogno, questo è dichiarato free e restituito alla propria cache, rendendosi così immediatamente disponibile ad altre richieste del kernel.

Ora Linux ha allocatori di memoria SLOB (per sistemi con limitata memoria) e SLUB (slab ottimizzato).

Layout memoria x86

Nell'architettura x86-32 lo spazio degli indirizzi lineari è diviso in due aree:

- **0-3GB**. Assegnato ad ogni processo (sostituito con context-switch)
- **3-4GB**. Assegnato al kernel, non sostituito



I primi 8MB di memoria (contigui) sono dedicati all'**immagine del kernel**, il codice del kernel è decompresso all'inizio del boot. I successivi 880MB (contigui fisicamente) sono dai frame fisici che possono essere liberi o allocati per oggetti del kernel; i frame liberi sono gestiti dall'allocatore buddy mentre quelli occupati dallo slab allocator.

L'area vmalloc è grande circa 120MB ed è dedicata alle porzioni di codice e dati dei moduli del kernel. I moduli sono attivati e disattivati più volte durante il ciclo di vita del kernel, l'area di memoria successiva non è garantita fisicamente contigua. (`vmalloc()`: allocazione blocco; `vfree()`: deallocazione blocco)

L'area kmap, grande circa 1-6MB è dedicata alla creazione di mappature lineari verso frame fisici oltre il GB. Tali mapping sono di breve durata (es. operazioni di I/O). L'area fixed map contiene pagine di 4KB i cui indirizzi lineari iniziali sono costanti (gestibili dal loader dinamico).

Nell'architettura x86-64, invece, lo spazio degli indirizzi lineari è diviso come segue:

- **0-128TB**. Assegnato ad ogni processo (sostituito ad ogni cambio contesto)
- **128TB-256TB**. Assegnato al kernel, non sostituito

Altre considerazioni

Le due scelte fondamentali nella progettazione dei sistemi di paginazione sono la definizione dell'algoritmo di sostituzione e della politica di allocazione, già analizzate in questo capitolo. Si devono però fare anche molte altre considerazioni.

Prepaging

Una caratteristica ovvia per un sistema di paginazione su richiesta pura consiste nell'alto numero di assenze di pagine che si verificano all'avvio di un processo. Questa situazione è dovuta al tentativo di portare la località iniziale in memoria. La stessa situazione si può presentare anche in altri momenti. Ad esempio, quando si riavvia un processo scaricato nell'area di swapping, tutte le sue pagine si trovano nel disco e ognuna di loro deve essere reinserita in memoria tramite la gestione di un'eccezione di page fault. La **prepaginazione** rappresenta un tentativo di prevenire un così alto livello di paginazione iniziale. Alcuni sistemi operativi, e in particolare Solaris, applicano la prepaginazione ai frame dei file di dimensioni modeste.

In un sistema che usa il modello del working set, ad esempio, a ogni processo si associa una lista delle pagine contenute nel suo insieme di lavoro. Se occorre sospendere un processo a causa di un'attesa di I/O oppure dell'assenza di frame liberi, si memorizza il suo working set. Al momento di riprendere l'esecuzione del processo, al completamento dell'I/O oppure quando si raggiunge il numero di frame sufficiente, prima di riavviare il processo, si riporta in memoria il suo intero working set.

In alcuni casi la prepaginazione può essere vantaggiosa. La questione riguarda semplicemente il suo costo, che deve essere inferiore al costo per l'assistenza dei corrispondenti page fault. Può accadere che molte pagine trasferite in memoria dalla prepaginazione non siano usate.

Si supponga che siano prepagnate s pagine e sia effettivamente usata una frazione $\alpha * s$ di queste s pagine ($0 \leq \alpha \leq 1$). Occorre sapere se il costo degli $\alpha * s$ page fault risparmiati sia maggiore o minore del costo di prepaging di $(1 - \alpha) * s$ pagine non necessarie. Se il parametro α è prossimo allo zero, la prepaginazione non è conveniente; se α tende ad uno, la prepaginazione certamente lo è.

Dimensione delle pagine

Se si devono progettare nuovi calcolatori, occorre stabilire quali siano le dimensioni migliori per le pagine; ovviamente non esiste un'unica dimensione migliore, ma sono invariabilmente potenze di due, in genere comprese tra 4096 (2^{12}) e 4 195 304 (2^{22}) byte.

Un fattore da considerare nella scelta delle dimensioni di una pagina è la dimensione della tabella delle pagine. Per un dato spazio di memoria virtuale, diminuendo la dimensione delle pagine aumenta il numero delle stesse e quindi la dimensione della tabella delle pagine. Per una memoria virtuale di 4 MB (2^{22}), ci sarebbero 4096 pagine di 1024 byte, ma solo 512 pagine di 8192 byte. Poiché ogni processo attivo deve avere la propria copia della tabella delle pagine, conviene che le pagine siano ampie.

D'altra parte, la memoria è utilizzata meglio se le pagine sono piccole. Se a un processo si assegna una porzione della memoria che comincia alla locazione 00000 e continua fino alla quantità di cui necessita, è molto probabile che il processo non termini esattamente sul limite di una pagina, lasciando inutilizzata una parte della pagina finale (frammentazione interna). Supponendo che le dimensioni del processo e delle pagine siano indipendenti è probabile che, in media, metà dell'ultima pagina di ogni processo sia sprecata. Questa perdita è di soli 256 byte per una pagina di 512 byte, ma di 4096 byte per una pagina di 8192 byte. Quindi, per ridurre la frammentazione interna occorrono pagine di piccole dimensioni.

Un altro problema è dovuto al tempo richiesto per leggere o scrivere una pagina. Il tempo di I/O è dato dalla somma dei tempi di posizionamento, latenza e trasferimento. Il tempo di trasferimento è proporzionale alla quantità trasferita, ossia alla dimensione delle pagine; tutto ciò farebbe supporre che

siano preferibili pagine piccole. D'altra parte, per ridurre il tempo di I/O occorre avere pagine di dimensioni maggiori. per ridurre il tempo di I/O occorre avere pagine di dimensioni maggiori. Tuttavia, con pagine di piccole dimensioni si riduce l'I/O totale, poiché si migliorano le caratteristiche di località. Pagine di piccole dimensioni permettono di corrispondere con maggior precisione alla località del programma.

Con pagine di piccole dimensioni è possibile avere una migliore **risoluzione**, che permette di isolare solo la memoria effettivamente necessaria. Se le dimensioni delle pagine sono maggiori, occorre assegnare e trasferire non solo quanto è necessario, ma tutto quel che è presente nella pagina, a prescindere dal fatto che sia o meno necessario. Quindi dimensioni più piccole danno come risultato una minore attività di I/O e una minore memoria totale assegnata.

tato una minore attività di I/O e una minore memoria totale assegnata. D'altra parte, occorre notare che con pagine di 1 byte si verifica un page fault per ciascun byte. Per ridurre il numero di page fault al minimo sono necessarie pagine di grandi dimensioni.

Ricapitolando per pagine di piccole dimensioni abbiamo la proprietà di località, meno frammentazione e meno I/O overhead; mentre, per pagine grandi si ha meno page fault e un page table più piccolo.

TLB reach

Il tasso di successi (**hit ratio**) di una TLB si riferisce alla percentuale di traduzioni di indirizzi virtuali risolte dalla TLB anziché dalla tabella delle pagine. Il tasso di successi è evidentemente proporzionale al numero di elementi della TLB. Tuttavia, la memoria associativa che si usa per costruire le TLB è costosa e consuma molta energia.

Un parametro simile, detto portata della TLB (**TLB reach**), esprime la quantità di memoria accessibile dalla TLB, ed è dato semplicemente dal numero di elementi (quindi è correlato al tasso di successi) moltiplicato per la dimensione delle pagine. Idealmente, la TLB dovrebbe contenere l'insieme di lavoro di un processo; altrimenti, la necessità di ricorrere alla tabella delle pagine per la traduzione dei riferimenti alla memoria farà sì che il processo impieghi in quest'operazione assai più tempo di quello richiesto dalla sola TLB. Se si raddoppia il numero di elementi della TLB, se ne raddoppia la portata; per alcune applicazioni che comportano un uso intensivo della memoria ciò potrebbe rivelarsi ancora insufficiente per la memorizzazione dell'insieme di lavoro.

Un altro metodo per aumentare la portata della TLB consiste nell'aumentare la dimensione delle pagine oppure nell'impiegare diverse dimensioni delle pagine. Se si aumenta la dimensione delle pagine, per esempio da 8 KB a 32 KB, la portata della TLB si quadruplica. Quest'aumento potrebbe però condurre a una maggiore frammentazione della memoria relativamente alle applicazioni che non richiedono pagine così grandi.

L'uso di diverse dimensioni delle pagine richiede però che la gestione della TLB sia svolta dal sistema operativo e non direttamente dall'architettura. Ad esempio, uno dei campi degli elementi della TLB deve indicare la dimensione della pagina fisica cui il contenuto di ciascun elemento fa riferimento. La gestione della TLB svolta dal sistema operativo e non esclusivamente dall'architettura comporta una penalizzazione delle prestazioni. Tuttavia, i vantaggi dovuti all'aumento del tasso di successi e della portata della TLB superano gli svantaggi che derivano dalla riduzione della rapidità di traduzione degli indirizzi. I recenti sviluppi indicano infatti un'evoluzione verso TLB gestite dal sistema operativo e verso l'uso di pagine di diverse dimensioni.

Tabella delle pagine invertita

Nel sottoparagrafo [omonimo](#) a quest'ultimo abbiamo già introdotto il concetto di tabella delle pagine invertita come sistema di gestione delle pagine che consente di ridurre la quantità di memoria fisica necessaria per tener traccia della corrispondenza tra gli indirizzi virtuali e gli indirizzi fisici. Tale riduzione si

ottiene tramite una tabella con un elemento per pagina fisica, indicizzato dalla coppia $\langle id\text{-processo}, page\text{-number} \rangle$.

Poiché contiene informazioni su quale pagina di memoria virtuale è memorizzata in ciascuna pagina fisica, la tabella delle pagine invertita riduce la quantità di memoria fisica necessaria a memorizzare tali informazioni. Tuttavia essa non contiene le informazioni complete sullo spazio degli indirizzi logici di un processo, richieste se una pagina a cui si è fatto riferimento non è correntemente presente in memoria; la paginazione su richiesta richiede tali informazioni per elaborare le eccezioni di pagine mancanti. Per disporre di tali informazioni è necessaria una tabella delle pagine esterna per ciascun processo. Queste tabelle sono simili alle ordinarie tabelle delle pagine di ciascun processo e contengono le informazioni relative alla locazione di ciascuna pagina virtuale.

Contrariamente a quel che potrebbe apparire, Fuso delle tabelle esterne delle pagine non è in contrasto con l'utilità della tabella delle pagine invertita; infatti a loro si fa riferimento solo nel caso di un'assenza di pagina; quindi non è necessario che siano immediatamente disponibili ed esse stesse sono paginate dentro e fuori dalla memoria come è necessario. Sfortunatamente, in questo modo si può verificare un'assenza di qualche pagina dello stesso gestore della memoria virtuale; in tal caso si verifica un'altra assenza di pagina quando esso carica in memoria la tabella esterna delle pagine per individuare la pagina virtuale in memoria ausiliaria (*backing store*). Questo caso particolare richiede un'accurata gestione da parte del kernel del sistema operativo e un ritardo nell'elaborazione della ricerca della pagina.

Struttura dei programmi

La paginazione su richiesta deve essere trasparente per il programma utente; spesso, l'utente non è neanche a conoscenza della natura paginata della memoria. In altri casi, però, le prestazioni del sistema si possono addirittura migliorare se l'utente (o il compilatore) è consapevole della sottostante presenza della paginazione su richiesta.

Come esempio limite, ma esplicativo, ipotizziamo pagine di 128 parole. Si consideri il seguente frammento di programma scritto in C la cui funzione è inizializzare a 0 ciascun elemento di una matrice di $128 * 128$ elementi (si noti che tale array è memorizzato per riga, ovvero nel seguente ordine `data[0][0],data[0][1], ...,data[0][127],data[1][0], ...:`:

```
int i, j;
int [128] [128] data;

for (j = 0; j < 128; ++j)
    for (i = 0; i < 128; ++i)
        data[i][j] = 0;
```

In pagine di 128 parole, ogni riga occupa una pagina, quindi il frammento di codice precedente azzera una parola per pagina, poi un'altra parola per pagina, e così via. Se il sistema operativo assegna meno di 128 frame a tutto il programma, la sua esecuzione causa $128 * 128 = 16384$ page fault. D'altra parte cambiando il codice ordinandolo per colonne si riduce il numero di page fault a 128:

```
int i, j;
int [128] [128] data;

for (i = 0; i < 128; ++i)
    for (j = 0; j < 128; ++j)
        data[i][j] = 0;
```

A volte il programmatore può gestire i seguenti oggetti apportando dei benefici:

- La scelta delle strutture dati può influire sul numero di page fault
- Lo stack che influenza sulla località poiché l'accesso avviene sempre su top
- Una tabella hash diffonde i riferimenti, causando una località non buona

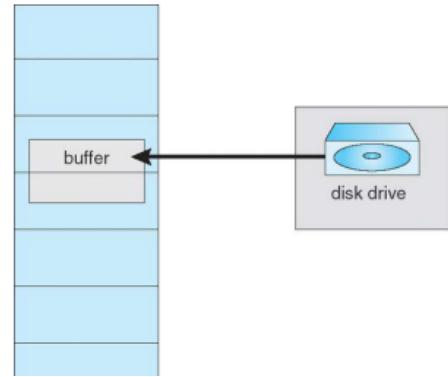
Compiler e loader possono avere un impatto sul paging:

- separazione di codice e dati e la generazione di un codice rientrante significano che le pagine di codice si possono soltanto leggere e quindi non vengono modificate
- il loader può evitare di collocare procedure lungo i limiti delle pagine, sistemando ogni procedura completamente all'interno di una pagina. Le procedure che si invocano a vicenda più volte si possono "impaccare" nella stessa pagina.

I/O interlock

Quando si usa la paginazione su richiesta, talvolta occorre permettere che alcune pagine si possano vincolare alla memoria (*locked in memory*). Una situazione di questo tipo si presenta quando I/O si esegue verso o dalla memoria utente (virtuale).

Spesso il sistema di I/O comprende una specifica unità d'elaborazione; al controllore di un dispositivo di memorizzazione USB, ad esempio, generalmente si indica il numero di byte da trasferire e un indirizzo di memoria per il buffer (si veda a questo proposito la figura). Completato il trasferimento, la CPU riceve un segnale d'interruzione.



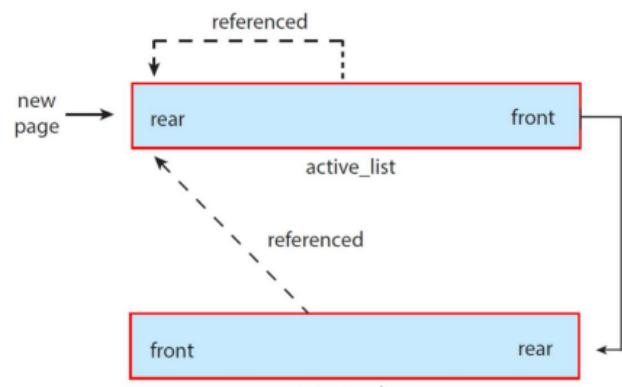
Occorre essere certi che non si verifichi la seguente successione di eventi: un processo emette una richiesta di I/O ed è messo in coda per il relativo dispositivo. Nel frattempo si assegna la CPU ad altri processi che accusano assenze di pagine e, usando un algoritmo di sostituzione globale per uno di questi, si sostituisce la pagina contenente l'indirizzo di memoria per l'operazione di I/O attesa dal primo processo; la pagina viene scaricata dalla memoria. Qualche tempo dopo, quando la richiesta di I/O raggiunge la prima posizione della coda d'attesa per il dispositivo, l'operazione di I/O avviene all'indirizzo specificato, ma questo frame è ora impiegato per una pagina appartenente a un altro processo. Questo problema si può risolvere in due modi. Una soluzione prevede di non eseguire operazioni di I/O in memoria utente, ma di copiare i dati sempre tra la memoria di sistema e la memoria utente. In questo modo l'I/O avviene solo tra la memoria di sistema e il dispositivo di I/O. Per scrivere dati in un nastro, occorre prima copiarli in memoria di sistema, quindi trasferirli all'unità a nastro. Tale copia supplementare può causare un sovraccarico inaccettabile.

Un'altra soluzione consiste nel permettere che le pagine siano vincolate alla memoria. A ogni frame si associa un bit di vincolo (*lock bit*); se tale bit è attivato, la pagina contenuta in tale frame non può essere selezionata per la sostituzione. Seguendo questo metodo, per scrivere dati in un nastro occorre vincolare alla memoria le pagine contenenti tali dati, quindi il sistema può continuare come di consueto. Le pagine vincolate non si possono sostituire. Completato l'I/O, si rimuove il vincolo.

Esempi tra i sistemi operativi

Linux

- Per memoria virtuale Linux usa demand paging con allocazione da lista di free frame
- Global page replacement con clock approx LRU
- Due liste di pagine **active_list** e **inactive_list**, le inactive sono elegibili per essere riusate
 - Un accessed bit per stabilire chi ha avuto un riferimento
 - Nuova pagina nel retro della active list (come ogni pagina con accesso in active)
 - Il bit è periodicamente resettato
 - Last Recently Used emerge sul fronte che retrocede nel retro dell'inactive



- Un demone kswapd periodicamente verifica se occorre liberare memoria, se la memoria libera va sotto soglia scorre la `inactive_list` e libera frame

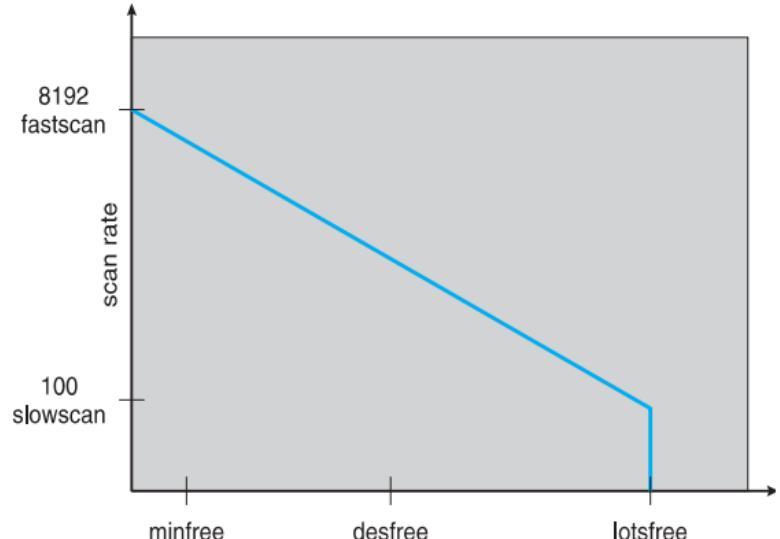
Windows

- Windows 10 supporta 32 o 64 bit (IA-32, IA-64, ARM)
- Windows 11 solo 64 bit v Su 32-bit memoria virtuale per processo è 2GB (estesa a 3GB), memoria fisica fino a 4GB
- Con 64-bit spazio indirizzi virtuale 128 TB e 24 TB di memoria fisica
- Supporta demand paging, copy-on-write, paging, memory compression
- Usa demand paging con **clustering**
 - Il clustering porta in memoria pagine “vicine” alla pagina che ha generato il page fault
 - Nel caso di data page il cluster è di 3 (la pagina richiesta, quella prima e dopo), altrimenti il cluster è 7
- Gestione del working set
 - Ai processi sono assegnati **working set minimum** e **working set maximum**
 - Working set minimum è il numero minimo di pagine che il processo è garantito avere in memoria (50 pagine)
 - Al processo possono essere assegnate tante pagine quanto è il suo working set maximum (345 pagine)
 - I valori possono essere ignorati a meno che non è settato **hard working set limit**
 - Win utilizza un LRU approx. clock con global e local replacement
 - Quando la memoria è sufficiente procede allocando free memory per servire il page fault
 - Quando la quantità di free memory nel sistema va sotto una soglia **automatic working set trimming** è eseguito per recuperare la quantità di free memory
 - Working set trimming rimuove pagine dai processi che hanno pagine in eccesso rispetto al loro working set minimum
 - Preferiti processi grandi e non attivi rispetto a quelli piccolo e attivi
 - Continua finché non si raggiunge la soglia (anche sottraendo sotto il min ws)
 - Trimming sia su processi utente che di sistema

Solaris

- Mantiene una lista di free pages da assegnare ai processi richiedenti
- lostfree → parametro soglia (quantità di free memory) per iniziare il paging (1/64 della dimensione della memoria fisica)
- desfree → parametro soglia di free memory per incrementare il paging
- Scansione 4 volte al secondo per verificare la soglia, se sotto soglia lostfree inizia il processo di **pageout**
- **pageout** scansiona le pagine usando un algoritmo second chance modificato
 - **Front hand**: scansione delle pagine azzerando il reference bit
 - **Back hand**: assegna le pagine non referenziate alla free list e la copia in backing store
 - La distanza tra front e back è di **handspear**
 - Una pagina non riassegnata ma con riaccesso può essere recuperata
- **scanrate** è la frequenza di scansione delle pagine.
 - Varia da slowscan a fastscan: da 100 pagine per secondo a mem-fisica/2 per secondo
- Pageout è chiamata più frequentemente in dipendenza dall'ammontare della free memory disponibile
- Il tempo di scansione dipende da scanrate e handspear
 - Se scanrate è 100 pagine al secondo e handspear è 1024 allora possono passare 10 secondi tra front hand e back hand
 - Nei sistemi reali il tempo tra front e back hand è di pochi secondi

- Pageout fa il check di memoria 4 volte al secondo, se desfree è sotto soglia incrementa a 100 volte al secondo
- Se il SO non riesce a tenere desfree (per 30 sec) allora inizia lo swapping e libera i frame dei processi swappati
- Sotto minfree fa pageout per ogni richiesta di nuova pagina
- Non fa swapping di shared pages, distingue tra pagine per processo e pagine per dati



10. Interfaccia del File System

Concetto di file

I calcolatori possono memorizzare le informazioni su diversi supporti, come dischi, nastri magnetici e dischi ottici. Per rendere agevole l'uso del calcolatore, il sistema operativo offre una visione logica uniforme della memorizzazione delle informazioni; astrae il file dalle caratteristiche fisiche dei propri dispositivi di memoria per definire un'unità di memoria logica. Il sistema operativo associa i file ai dispositivi fisici, di solito non volatili, in modo che il loro contenuto non vada perduto a causa delle interruzioni dell'alimentazione elettrica e dei riavvii del sistema.

Un file è un insieme di informazioni, correlate e 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 secondaria logica; i dati si possono cioè scrivere in memoria secondaria soltanto all'interno di un file. Di solito i file rappresentano programmi, in forma sorgente e oggetto, e dati. I file di dati possono essere numerici, alfabetici, alfanumerici o binari, e non possedere un formato specifico, come i file di testo; oppure essere rigidamente formattati. In genere un file è formato da una sequenza di bit, byte, righe o record il cui significato è definito dal creatore e dall'utente del file stesso. Il concetto di file è quindi estremamente generale

Le informazioni contenute in un file sono definite dal suo creatore e possono essere di molti tipi: programmi sorgente, programmi oggetto, dati numerici, testo, dati contabili, immagini, registrazioni sonore, e così via. Un file ha una struttura definita secondo il tipo: un file di testo è formato da una sequenza di caratteri organizzati in righe, e probabilmente pagine; un file sorgente è formato da una sequenza di procedure e funzioni, ciascuna delle quali è a sua volta organizzata in dichiarazioni seguite da istruzioni eseguibili; un file oggetto è formato da una sequenza di byte, organizzati in blocchi, comprensibile al modulo di collegamento del sistema; un file eseguibile consiste di una serie di sezioni di codice che il caricatore può caricare in memoria ed eseguire.

Attributi dei file

Un file ha altri 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 far 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 e per controllarne l'uso.

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. Un elemento di directory può richiedere più di un kilobyte per contenere queste informazioni per ciascun file. In un sistema con molti file, la dimensione della stessa directory può essere dell'ordine di megabyte. Poiché le directory, come i file, devono essere non volatili, si devono registrare in memoria secondaria e caricare in memoria centrale un po' per volta, secondo le necessità.

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. Le successive considerazioni su ciò che deve fare un sistema operativo per ciascuna di queste sei operazioni di base dovrebbero rendere più semplice osservare come si possano realizzare altre operazioni simili, ad esempio la ridefinizione di un file.

- **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; la discussione sui criteri di allocazione dei file è rimandata al [Capitolo 11](#). Secondariamente, per il file si deve creare un nuovo elemento nella directory in cui registrare il nome del file, la sua posizione nel file system ed eventualmente altre informazioni.
- **Scrittura di un file.** Per scrivere in un file è indispensabile una chiamata di sistema che specifichi il nome del file e le informazioni che si vogliono scrivere. Dato il nome del file, il sistema cerca la sua posizione nella directory. 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 successivo blocco del file. 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 legge o scrive in un file, e la posizione corrente è mantenuta come un puntatore alla posizione corrente del file. Sia le operazioni di lettura sia quelle di scrittura adoperano lo stesso puntatore, risparmiando spazio e riducendo la complessità del sistema.
- **Riposizionamento in 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 a 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.

Queste sei operazioni di base comprendono sicuramente l'insieme minimo delle operazioni richieste per i file. Altre operazioni comuni comprendono l'aggiunta (*Appending*) di nuove informazioni alla fine di un file esistente e la ridefinizione di un file esistente. Queste operazioni primitive si possono combinare per compiere altre operazioni. Sono inoltre necessarie operazioni che consentano a un utente di leggere e impostare i vari attributi di un file.

La maggior parte delle operazioni sopra citate richiede una ricerca dell'elemento associato al file specificato nella directory. Per evitare questa continua ricerca, molti sistemi richiedono l'impiego di una chiamata di sistema `open()` la prima volta che si adopera un file in maniera attiva. Il sistema operativo mantiene una piccola tabella contenente informazioni riguardanti tutti i file aperti (detta, per l'appunto, **tabella dei file aperti**). Quando si richiede un'operazione su un file, questo viene individuato tramite un indice in tale tabella, in questo modo si evita qualsiasi ricerca. Quando il file non è più attivamente usato viene chiuso dal processo, e il sistema operativo rimuove l'elemento a esso associato dalla tabella dei file aperti.

Alcuni sistemi aprono implicitamente un file al primo riferimento e lo chiudono automaticamente quando il processo che lo ha aperto termina. A ogni modo, la maggior parte dei sistemi esige che il programmatore

richieda l'apertura del file in modo esplicito per mezzo di una chiamata di sistema `open()` prima che sia possibile adoperarlo. L'operazione `open()` riceve il nome del file, lo cerca nella directory e copia l'elemento a esso associato nella tabella dei file aperti. La chiamata di sistema `open()` può accettare anche informazioni sui modi d'accesso: creazione, sola lettura, lettura e scrittura, sola aggiunta, etc. Si controllano i permessi relativi al file, e se la modalità d'accesso richiesta è consentita, si apre il file. La chiamata di sistema `open()` riporta di solito un puntatore all'elemento nella tabella dei file aperti; questo puntatore si adopera al posto dell'effettivo nome del file in tutte le operazioni di I/O, evitando così successive operazioni di ricerca e semplificando l'interfaccia delle chiamate di sistema.

La realizzazione delle operazioni `open()` e `close()` è più complicata in un ambiente multiutente dove più utenti possono aprire un file contemporaneamente. Di solito il sistema operativo introduce due livelli di tabelle interne: una tabella per ciascun processo e una tabella di sistema. La tabella di un processo contiene i riferimenti a tutti i file aperti da quel processo; il puntatore alla posizione di ciascun file si trova in questa tabella e specifica la posizione nel file. Si possono includere anche i diritti d'accesso al file e informazioni di contabilizzazione.

Ciascun elemento della tabella associata a ciascun processo punta a sua volta a una tabella di sistema dei file aperti, contenente le informazioni indipendenti dai processi come la posizione dei file nei dischi, le date degli accessi e le dimensioni dei file. Quando un file è già stato aperto da un processo, l'apertura di un file da parte di un processo comporta l'aggiunta di un elemento relativo al file nella tabella dei file aperti del sistema; una `open()` eseguita da un altro processo comporta solamente l'aggiunta di un nuovo elemento nella tabella dei file aperti associata a quel processo, che punta al corrispondente elemento della tabella di sistema. In genere, la tabella dei file aperti ha anche un contatore delle aperture associato a ciascun file, indicante il numero di processi che hanno aperto quel file. Ogni `close()` decremente questo contatore; quando raggiunge il valore zero il file non è più in uso e si elimina l'elemento corrispondente dalla tabella dei file aperti. Riassumendo, a ciascun file aperto sono associate le diverse seguenti informazioni.

- **Puntatore al file.** Nei sistemi che non prevedono lo scostamento come parte delle chiamate di sistema `read()` e `write()`, il sistema deve tener traccia dell'ultima posizione di lettura e scrittura sotto forma di un puntatore alla posizione corrente nel file. Questo puntatore è unico per ogni processo che opera sul file e quindi deve essere tenuto separato dagli attributi del file residenti nel disco.
- **Contatore dei file aperti.** Man mano che si chiudono i file, per evitare di esaurire lo spazio associato alla propria tabella dei file aperti, il sistema operativo deve riutilizzarne gli elementi. Poiché più processi possono aprire uno stesso file, prima di rimuovere l'elemento corrispondente, il sistema deve attendere l'ultima chiusura del file. Questo contatore tiene traccia del numero di `open()` e `close()`, e raggiunge il valore zero dopo l'ultima chiusura, momento in cui il sistema può rimuovere l'elemento della tabella.
- **Posizione nel disco del file.** La maggior parte delle operazioni richiede al sistema di modificare i dati contenuti nel file. L'informazione necessaria per localizzare il file nel disco è mantenuta in memoria, per evitare di doverla prelevare dal disco a ogni operazione.
- **Diritti d'accesso.** Ciascun processo apre un file in una delle modalità d'accesso. Questa informazione è contenuta nella tabella del processo in modo che il sistema operativo possa permettere o negare le successive richieste di I/O.

Alcuni sistemi operativi offrono la possibilità di applicare lock a un file aperto (o a parti di esso). Quando un processo intende proteggere un file dall'accesso concorrente di altri processi, si serve dei lock. L'utilità dei lock dei file emerge nel caso di file condivisi da diversi processi: un file di log, per esempio, può subire modifiche da parte di molti processi.

I lock dei file sono basati su una funzionalità simile ai [lock di lettura-scrittura](#). Un **lock condiviso** è assimilabile, per funzionamento, ai lock di lettura: entrambi consentono a più processi concorrenti di

appropriarsene. Un **lock esclusivo** mostra invece analogie con i lock di scrittura, perché un solo processo per volta può acquisire questo tipo di lock. Si noti bene che non in tutti i sistemi operativi è possibile scegliere tra i due tipi di lock; alcuni sistemi forniscono solamente lock esclusivi dei file.

Inoltre, il sistema operativo può fornire meccanismi di protezione dei file **obbligatori**, oppure **consigliati**. Se un lock è obbligatorio, il sistema operativo impedirà a qualunque altro processo di accedere al file interessato una volta che il suo lock sia stato acquisito.

L'uso dei lock dei file richiede l'osservazione delle stesse accortezze per la sincronizzazione dei processi. Per esempio, i programmati impegnati a sviluppare su sistemi con lock obbligatori devono prestare attenzione a detenere i lock esclusivi solo per l'effettiva durata degli accessi ai file; in caso contrario, bloccheranno anche gli accessi da parte di altri processi. Occorre, inoltre, attuare misure appropriate al fine di evitare che due o più processi entrino in stallo nel tentativo di acquisire i lock per i file.

Tipi di file

Nella progettazione di un file system, ma anche dell'intero sistema operativo, si deve sempre considerare la possibilità o meno che quest'ultimo riconosca e gestisca i tipi di file. Un sistema operativo che riconosce il tipo di un file ha la possibilità di trattare il file in modo ragionevole. Ad esempio, un errore abbastanza comune consiste nel tentativo, da parte degli utenti, di stampare un programma oggetto in forma binaria; di solito questo tentativo porta semplicemente a uno spreco di carta, ma si potrebbe impedire se il sistema operativo fosse informato del fatto che il file è un programma oggetto in forma binaria.

Una tecnica comune per realizzare la gestione dei tipi di file consiste nell'includere il tipo nel nome del file. Il nome è suddiviso in due parti, un nome e una estensione, di solito separate da un punto.

file type	usual extension	function
executable	exe, com, bin or none	ready-to-run machine-language program
object	obj, o	compiled, machine language, not linked
source code	c, cc, java, pas, asm, a	source code in various languages
batch	bat, sh	commands to the command interpreter
text	txt, doc	textual data, documents
word processor	wp, tex, rtf, doc	various word-processor formats
library	lib, a, so, dll	libraries of routines for programmers
print or view	ps, pdf, jpg	ASCII or binary file in a format for printing or viewing
archive	arc, zip, tar	related files grouped into one file, sometimes compressed, for archiving or storage
multimedia	mpeg, mov, rm, mp3, avi	binary file containing audio or A/V information

Struttura dei file

Per il sistema operativo la localizzazione di uno scostamento all'interno di un file può essere complicata. I dischi hanno una dimensione dei blocchi ben definita, stabilita secondo la dimensione di un settore. Tutti gli I/O su disco si eseguono in unità di un blocco (record fisico), e tutti i blocchi hanno la stessa dimensione. E improbabile che la dimensione del record fisico corrisponda esattamente alla lunghezza del record logico

desiderato, che può anche essere variabile. Una soluzione diffusa per questo tipo di problema consiste nell'impaccamento di un certo numero di record logici in blocchi fisici.

Il sistema operativo UNIX, ad esempio, definisce tutti i file semplicemente come un flusso di byte. A ciascun byte si può accedere in modo individuale tramite il suo scostamento a partire dall'inizio, o dalla fine, del file. In questo caso il record logico è un byte. Il file system impacca e de-impacca automaticamente i byte in blocchi fisici (ad esempio 512 byte per blocco) come è necessario.

La dimensione dei record logici, quella dei blocchi fisici e la tecnica d'impaccamento determinano il numero dei record logici all'interno di ogni blocco fisico. L'impaccamento può essere fatto dal programma applicativo dell'utente oppure dal sistema operativo.

In entrambi i casi il file si può considerare come una sequenza di blocchi. Tutte le funzioni di I/O di base operano in termini di blocchi. La conversione da record logici a blocchi fisici è un problema di programmazione relativamente semplice.

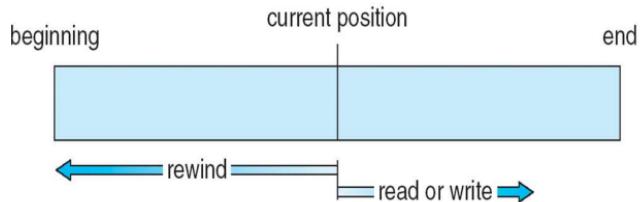
Poiché lo spazio del disco è sempre assegnato in blocchi, una parte dell'ultimo blocco di ogni file in genere è sprecata. Se ogni blocco è composto di 512 byte, a un file di 1949 byte si assegnano quattro blocchi (2048 byte); gli ultimi 99 byte sono sprecati. I byte sprecati, assegnati per la gestione in multipli di blocchi invece che di byte, costituiscono la **frammentazione interna**. Tutti i file system ne soffrono; maggiore è la dimensione dei blocchi, maggiore sarà la frammentazione interna.

Metodi di accesso

I file memorizzano informazioni; al momento dell'uso è necessario accedere a queste informazioni e trasferirle in memoria. Esistono molti metodi per accedere alle informazioni dei file; alcuni sistemi consentono un solo metodo d'accesso ai file; altri, come quelli IBM, offrono diversi metodi d'accesso; la scelta del metodo giusto per una particolare applicazione è un importante problema di progettazione.

Accesso sequenziale

Il più semplice metodo d'accesso è l'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, ad esempio, dagli editor e dai compilatori. Le più comuni operazioni che si compiono sui file sono le letture e le scritture: un'operazione di lettura legge la prima porzione e fa avanzare automaticamente il puntatore del file che tiene traccia della locazione di I/O; analogamente, un'operazione di scrittura fa un'aggiunta in coda al file e avanza fino alla fine delle informazioni appena scritte, che costituisce la nuova fine del file. Un file siffatto si può reimpostare sull'inizio e, in alcuni sistemi, un programma può riuscire ad andare avanti o indietro di n record, con n intero e alcune volte solo per $n = 1$. L'accesso sequenziale è illustrato nella figura a destra. L'accesso sequenziale è basato su un modello di file che si rifà al nastro, e funziona nei dispositivi ad accesso sequenziale così come nei dispositivi ad accesso diretto.



Accesso diretto

Un altro metodo è l'**accesso diretto** (o accesso relativo). 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. Il file si considera come una sequenza numerata di blocchi o record che si possono leggere o scrivere in modo arbitrario: si può ad esempio leggere il blocco 14, quindi il blocco 53 e poi scrivere il blocco 7. Non esistono limiti all'ordine di lettura o scrittura di un file ad accesso diretto.

I file ad accesso diretto sono molto utili quando è necessario accedere immediatamente a grandi quantità di informazioni. Spesso le basi di dati sono di questo tipo: quando si presenta un'interrogazione riguardante un oggetto particolare, occorre stabilire quale blocco contiene la risposta alla richiesta e quindi leggere direttamente quel blocco, ottenendo così le informazioni richieste.

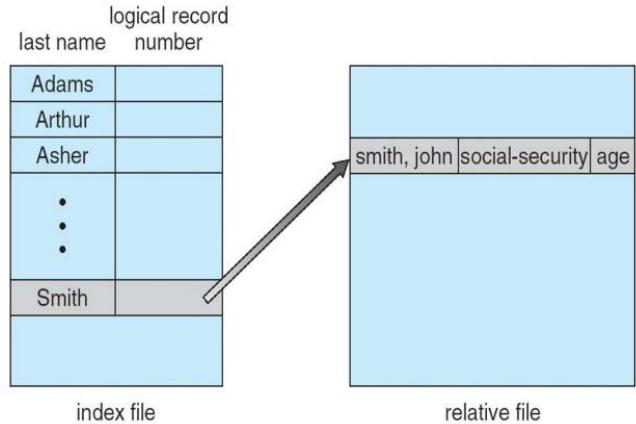
In un sistema di prenotazione di volo, ad esempio, si possono registrare tutte le informazioni su un particolare volo, ad esempio il volo 713, nel blocco identificato da tale numero di volo. Quindi, il numero di posti disponibili per il volo 713 si memorizza nel blocco 713 del file di prenotazione. Per registrare informazioni riguardanti un gruppo più grande, ad esempio una popolazione, si può eseguire la ricerca calcolando una funzione hash sui nomi delle persone, oppure usando un piccolo indice per determinare il blocco da leggere. Per il metodo ad accesso diretto, si devono modificare le operazioni sui file per inserire il numero del blocco in forma di parametro.

Altri metodi d'accesso

Sulla base di un metodo d'accesso diretto se ne possono costruire altri, che implicano generalmente la costruzione di un indice per il file. L'**indice** contiene puntatori ai vari blocchi; per trovare un elemento del file occorre prima cercare nell'indice, e quindi usare il puntatore per accedere direttamente al file e trovare l'elemento desiderato.

Nel caso di file molto lunghi, lo stesso file indice può diventare troppo lungo perché sia tenuto in memoria. Una soluzione a questo problema è data dalla creazione di un indice per il file indice. Il file indice principale contiene puntatori ai file indice secondari, che puntano agli effettivi elementi di dati.

I metodi ad accesso sequenziale indicizzato di IBM (*indexed sequential access method*, ISAM), ad esempio, usa un piccolo indice principale che punta ai blocchi del disco di un indice secondario, e i blocchi dell'indice secondario puntano ai blocchi del file effettivo. Il file è ordinato rispetto a una chiave definita. Per trovare un particolare elemento, si fa inizialmente una ricerca binaria nell'indice principale, che fornisce il numero del blocco dell'indice secondario. Questo blocco viene letto e sottoposto a una seconda ricerca binaria che individua il blocco contenente l'elemento richiesto. Infine, si fa una ricerca sequenziale sul blocco. In questo modo si può localizzare ogni elemento tramite il suo codice con al massimo due letture ad accesso diretto. La figura a destra mostra uno schema simile, com'è realizzato nel VMS con indici e relativi file.



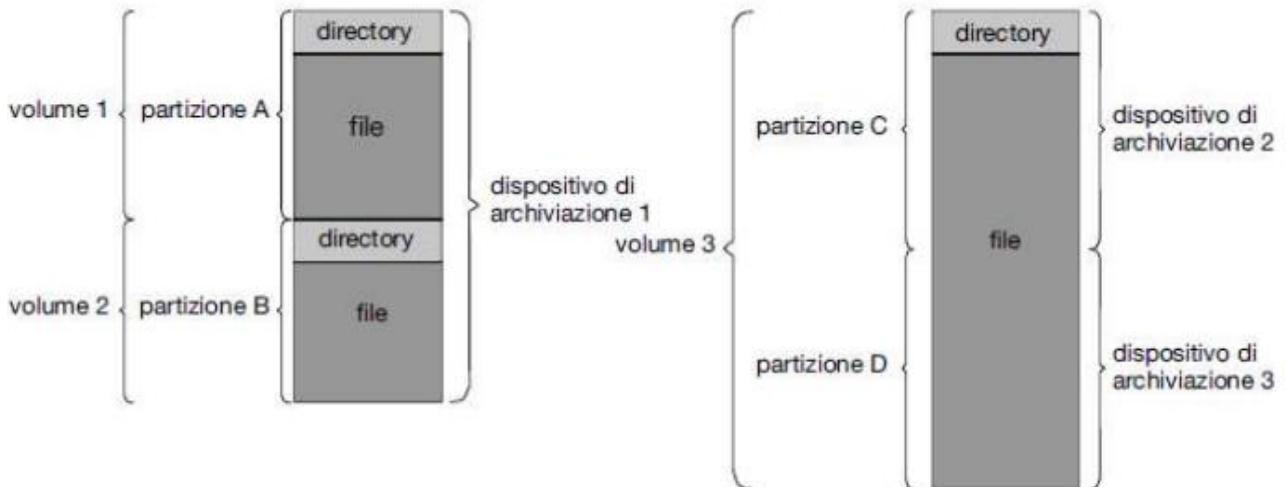
Struttura della directory e del disco

Un dispositivo di memorizzazione può essere interamente utilizzato per un file system, ma può anche essere suddiviso per un controllo più raffinato. Un disco può ad esempio essere partizionato e ogni partizione può contenere un file system. I dispositivi di memorizzazione possono essere raccolti in insiemi RAID che proteggono dal fallimento di un singolo disco. Alcune volte, i dischi sono suddivisi e al contempo raccolti in insiemi RAID.

La suddivisione in partizioni è utile anche per limitare la dimensione dei file system individuali, per mettere sullo stesso dispositivo tipi diversi di file system, oppure per liberare ad altri scopi una parte del dispositivo, come nel caso dello spazio di swap o dello spazio su disco non formattato (**raw**). Le partizioni sono note anche come suddivisioni (**slice**) o minidischi (nel mondo IBM). Un file system può essere installato su ciascuna di queste parti del disco. Ogni entità contenente un file system è generalmente nota come **volume**. Il volume può essere un sottoinsieme di dispositivi, un dispositivo intero o dispositivi multipli.

collegati in RAID. Ogni volume può essere pensato come un disco virtuale. I volumi possono inoltre contenere diversi sistemi operativi, permettendo al sistema di avviare ed eseguire più di un sistema operativo.

Ogni volume contenente un file system deve anche avere in sé le informazioni sui file presenti nel sistema. Tali informazioni risiedono in una **directory del dispositivo o indice del volume**. La directory del dispositivo (in breve directory) registra informazioni, quali nome, posizione e tipo, di tutti i file del volume. La figura seguente mostra la tipica organizzazione dei file system:



Struttura della memorizzazione di massa

Come discusso, un sistema informatico a uso generale dispone di dispositivi di memorizzazione multipli, che possono essere ripartiti nei volumi che contengono i file system. Il numero di file system in un sistema informatico può variare da zero a molti, e ne esistono di diverso tipo.

Consideriamo i tipi di file system del sistema Solaris:

- **tmpfs** - un file system “temporaneo” creato nella memoria centrale volatile e i cui contenuti vengono cancellati se il sistema si riavvia o si blocca;
- **objfs** - un file system “virtuale” (essenzialmente un’interfaccia con il kernel che è simile a un file system) che permette agli strumenti che eseguono il debug di accedere ai simboli del kernel;
- **ctfs** - un file system virtuale che mantiene le informazioni “contrattuali” per gestire quali sono i processi che partono quando il sistema si avvia e quali devono continuare a girare durante il funzionamento del sistema;
- **lofs** - un file system di tipo “loop back” che permette di accedere a un file system piuttosto che a un altro;
- **procfs** - un file system virtuale che presenta le informazioni su tutti i processi presenti come se esse risiedessero su un file system;
- **ufs, zfs** - file system a scopo generico.

I file system dei computer possono quindi essere numerosi. Persino all’interno di un file system è utile dividere i file in gruppi da gestire e sui quali agire.

Generalità sulla directory

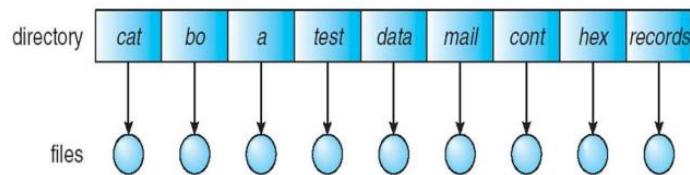
La directory si può considerare come una tabella di simboli che traduce i nomi dei file negli elementi in essa contenuti. Da questo punto di vista, si capisce che la stessa directory si può organizzare in molti modi diversi; deve essere possibile inserire nuovi elementi, cancellarne di esistenti, cercare un elemento ed elencare tutti gli elementi della directory. In questo paragrafo si affronta l’analisi delle appropriate strutture dati utilizzabili per la realizzazione delle directory. 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 ai rispettivi 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 comportare 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. Questo salvataggio consiste nella copiatura di tutti i file in un nastro magnetico; tale tecnica consente di avere una copia di riserva (backup) che sarebbe utile nel caso in cui si dovesse verificare un guasto nel sistema o più semplicemente se un file non è più in uso. In quest'ultimo caso si può liberare lo spazio da esso occupato nel disco, riutilizzabile quindi per altri file.

Single-Level Directory

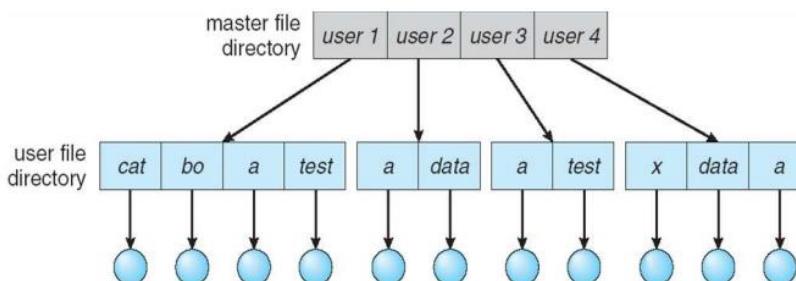
La struttura più semplice per una directory è quella a livello singolo. Tutti i file sono contenuti nella stessa directory, facilmente gestibile e comprensibile (problema del naming e del grouping).



Una directory a livello singolo presenta però limiti notevoli che si manifestano all'aumentare del 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.

Two-Level Directory

Nella struttura a due livelli, ogni utente dispone della propria directory utente. Tutte le directory utente hanno una struttura simile, ma in ciascuna sono elencati solo i file del proprietario. 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 indirizzata con il nome dell'utente o il numero che lo rappresenta, e ogni suo elemento punta alla relativa directory utente.



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. Per creare un file per un utente, il sistema operativo controlla che non ci sia un altro file con lo stesso nome soltanto nella directory di tale utente. Per

cancellare un file il sistema operativo limita la propria ricerca alla directory utente locale, quindi non può cancellare per errore un file con lo stesso nome che appartenga a un altro utente.

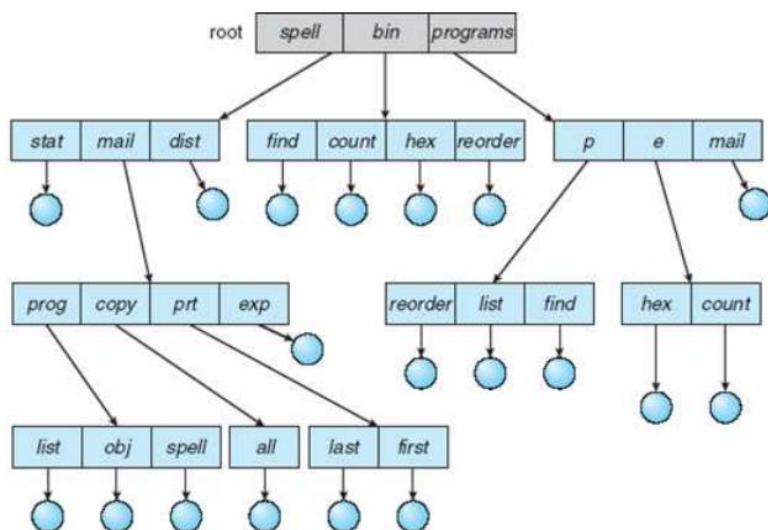
Le stesse directory utente devono essere create e cancellate come è necessario; a tale scopo si esegue uno speciale programma di sistema con nome dell'utente e dati contabili adeguati. Il programma crea una nuova directory utente e aggiunge l'elemento a essa corrispondente nella directory principale. L'esecuzione di questo programma può essere limitata all'amministratore del sistema.

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 gli utenti vogliono cooperare e accedere a file di altri utenti. Alcuni sistemi non permettono l'accesso a file di utenti locali da parte di altri utenti.

Se l'accesso è autorizzato, un utente deve avere la possibilità di riferirsi al nome di un file che si trova nella directory di un altro utente. Per attribuire un nome unico a un particolare file di una directory a due livelli, occorre indicare sia il nome dell'utente sia il nome del file (nome utente e nome del file definiscono il *path name*).

Tree-Structured Directory

La corrispondenza strutturale tra directory a due livelli e albero a due livelli permette di generalizzare facilmente il concetto, estendendo la struttura della directory a un albero di altezza arbitraria.



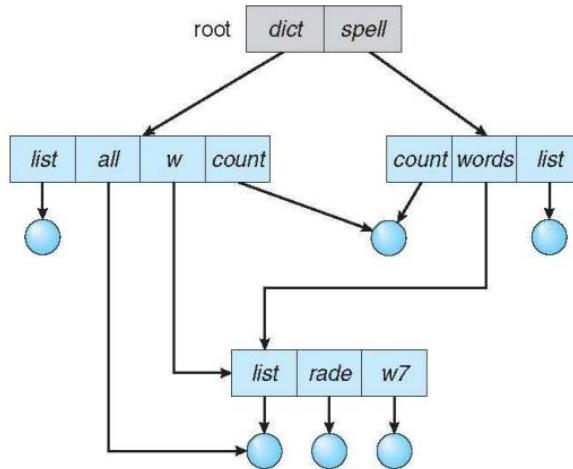
Questa generalizzazione permette agli utenti di creare proprie sottodirectory e di organizzare i file di conseguenza. L'albero ha una root directory, e ogni file del sistema ha un unico nome di percorso.

Una directory, o una sottodirectory, contiene un insieme di file o sottodirectory. Le directory sono semplicemente file, trattati però in modo speciale. Tutte le directory hanno lo stesso formato interno. La distinzione tra file e directory è data dal bit, rispettivamente 0 e 1, di ogni elemento della directory. Per creare e cancellare le directory si adoperano speciali chiamate di sistema.

Normalmente, ogni utente dispone di una directory corrente. La directory corrente (working directory) deve 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.

Acyclic-Graph Directory

La struttura ad albero non ammette la condivisione di file o directory. Un grafo aciclico permette alle directory di avere sottodirectory e file condivisi.



Lo stesso file o la stessa sottodirectory possono essere in due directory diverse. Un grafo aciclico, cioè senza cicli, rappresenta la generalizzazione naturale dello schema delle directory con struttura ad albero.

Il fatto che un file sia condiviso, o che lo sia una directory, non significa che ci siano due copie del file: con due copie ciascun programmatore potrebbe vedere la copia presente nella propria directory e non l'originale; se un programmatore modifica il file, le modifiche non appaiono nell'altra copia. Se invece il file è condiviso esiste un solo file effettivo, perciò tutte le modifiche sono immediatamente visibili. La condivisione è di particolare importanza se applicata alle sottodirectory; un nuovo file appare automaticamente in tutte le sottodirectory condivise.

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.

Una struttura della 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. Questa situazione è simile al problema dell'uso degli alias nei linguaggi di programmazione. Quando si percorre tutto il file system il problema diviene più grave poiché le strutture condivise non si devono attraversare più di una volta.

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 a un file che ormai non esiste più. Sarebbe ancora più grave se i puntatori contenessero gli indirizzi effettivi del disco e lo spazio fosse poi riutilizzato per altri file, poiché i puntatori potrebbero puntare nel mezzo di questi altri file.

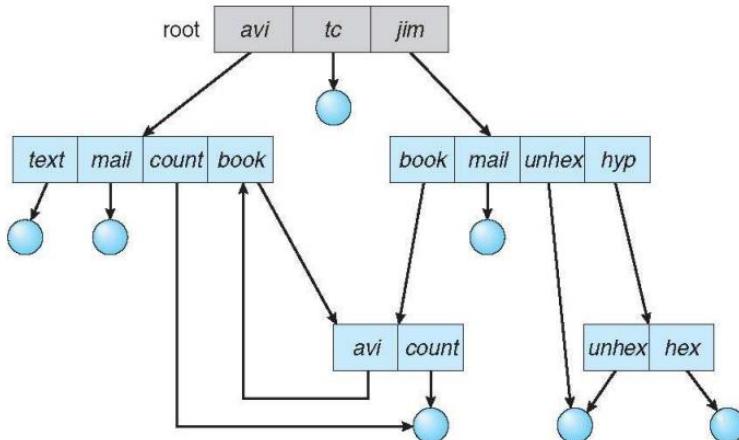
Soluzioni al problema della cancellazione sono i seguenti:

- Backpointers, così si possono cancellare tutti i pointer Problema dei record a dimensione variabile
- Backpointers usando un'organizzazione a daisy chain
- Soluzione entry-hold-count

General Graph Directory

Un serio problema connesso all'uso di una struttura a grafo aciclico consiste nell'assicurare che non vi siano cicli. Iniziando con una directory a due livelli e permettendo agli utenti di creare sottodirectory si crea una

directory con struttura ad albero. Aggiungendo nuovi file e nuove sottodirectory alla directory con struttura ad albero, la natura di quest'ultima persiste. Tuttavia, quando si aggiungono dei collegamenti a una directory con struttura ad albero, tale struttura si trasforma in una semplice struttura a grafo:



Il vantaggio principale di un grafo aciclico è dato dalla semplicità degli algoritmi necessari perattraversarlo e per determinare quando non ci siano più riferimenti a un file. È preferibile evitare un duplice attraversamento di sezioni condivise di un grafo aciclico, soprattutto per motivi di prestazioni. Se un file particolare è stato appena cercato in una sottodirectory condivisa, ma non è stato trovato, è preferibile evitare una seconda ricerca nella stessa sottodirectory, che costituirebbe solo una perdita di tempo.

Se si permette che nella directory esistano cicli, è preferibile evitare una duplice ricerca di un elemento, per motivi di correttezza e di prestazioni. Un algoritmo mal progettato potrebbe causare un ciclo infinito di ricerca. Una soluzione è quella di limitare arbitrariamente il numero di directory cui accedere durante una ricerca.

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 garbage collection per stabilire quando sia stato cancellato l'ultimo riferimento e quando sia possibile riallocare lo spazio dei dischi. Tale metodo implica l'attraversamento del file system, durante il quale si contrassegna tutto ciò che è accessibile; in un secondo passaggio si raccoglie in un elenco di blocchi liberi tutto ciò che non è contrassegnato. Una procedura di marcatura analoga è utilizzabile per assicurare che un attraversamento o una ricerca copriano tutto quel che si trova nel file system una sola volta. L'applicazione di questo metodo a 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. Per sapere quando un nuovo collegamento ha completato un ciclo si possono impiegare gli algoritmi che permettono di individuare la presenza di cicli nei grafi. Dal punto di vista del calcolo, però, questi algoritmi sono onerosi, soprattutto quando il grafo si trova in memoria secondaria. Nel caso particolare di directory e collegamenti, un semplice algoritmo prevede di evitare i collegamenti durante l'attraversamento delle directory: si evitano così i cicli senza alcun carico ulteriore.

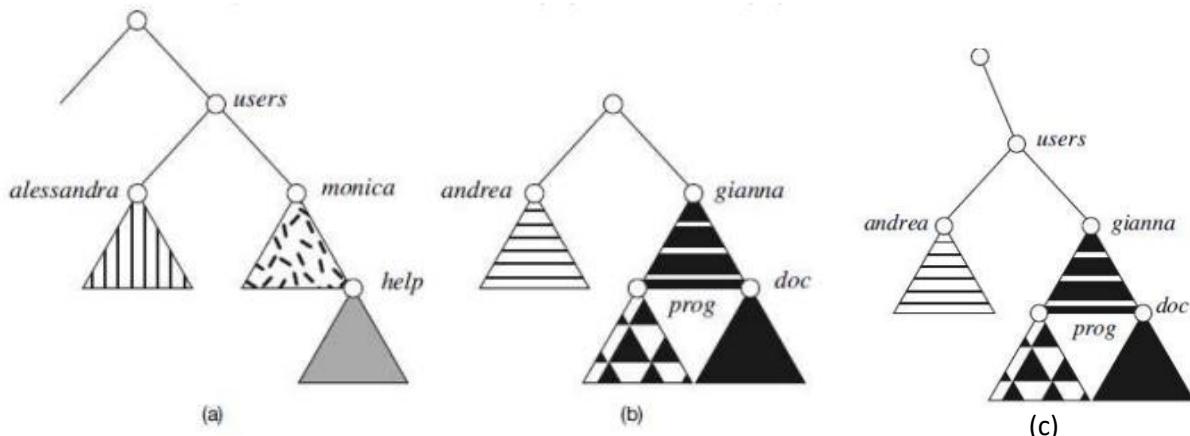
Montaggio di un file system

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 struttura delle directory può ad esempio essere composta di volumi, che devono essere montati affinché siano disponibili nello spazio dei nomi di un file system.

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. Alcuni sistemi operativi richiedono un file system prefissato, mentre altri ispezionano le strutture del dispositivo e determinano il tipo del file system presente. Di solito, un punto di montaggio è una directory vuota cui sarà agganciato il file system che deve essere montato. Ad esempio, in un sistema UNIX, un file system contenente le directory iniziali degli utenti si potrebbe montare come /home: quindi la directory iniziale dell'utente *simone* avrebbe il percorso /home/simone. Se lo stesso file system si montasse come /users il percorso per quella directory sarebbe /users/simone.

Il passo successivo consiste nella verifica da parte del sistema operativo della validità del file system contenuto nel dispositivo. La verifica si compie chiedendo al driver del dispositivo di leggere la directory di dispositivo e controllando che tale directory abbia il formato previsto. Infine, il sistema operativo annota nella sua struttura della directory che un certo file system è montato al punto di montaggio specificato. Questo schema permette al sistema operativo di attraversare la sua struttura della directory, passando da un file system all'altro secondo le necessità.

Per illustrare l'operazione di montaggio, si consideri il file system rappresentato nella figura seguente, in cui i triangoli rappresentano sottoalberi di directory rilevanti.



La figura (a), mostra un file system esistente, mentre nella figura (b) è raffigurato un volume non ancora montato che risiede in /device/dsk. Nella figura (c) si possono vedere gli effetti dell'operazione di montaggio del volume residente in /device/dsk al punto di montaggio /users. Se si smonta il volume, il file system ritorna alla situazione rappresentata nelle figure (a) e (b).

Condivisione di file

In questo paragrafo si considerano diversi aspetti della condivisione dei file, innanzitutto l'aspetto relativo alla molteplicità degli utenti e ai diversi metodi di condivisione possibili. Una volta che più utenti possono condividere file, l'obiettivo diventa estendere la condivisione a più file system, compresi i file system remoti. Infine, ci possono essere diverse interpretazioni delle azioni conflittuali intraprese su file condivisi. Ad esempio, se più utenti stanno scrivendo nello stesso file, ci si può chiedere se il sistema dovrebbe permettere tutte le operazioni di scrittura, oppure dovrebbe proteggere le azioni di ciascun utente da quelle degli altri.

Utenti multipli

Se un sistema operativo permette l'uso del sistema da parte di più utenti, diventano particolarmente rilevanti i problemi relativi alla condivisione dei file, alla loro identificazione tramite nomi e alla loro protezione. Data una struttura della directory che permette la condivisione di file da parte degli utenti, il sistema deve poter mediare questa condivisione. Il sistema può permettere a ogni utente, in modo predefinito, di accedere ai file degli altri utenti, oppure può richiedere che un utente debba esplicitamente concedere i permessi di accesso ai file.

Per realizzare i meccanismi di condivisione e protezione, il sistema deve memorizzare e gestire più attributi di directory e file rispetto a un sistema che consente un singolo utente. Sebbene storicamente siano stati proposti molti metodi per la realizzazione di questi meccanismi, la maggior parte dei sistemi ha adottato (relativamente a ciascun file o directory) i concetti di proprietario (o utente) e gruppo. Il proprietario è l'utente che può cambiare gli attributi di un file o directory, concedere l'accesso e che, in generale, ha il maggior controllo sul file o directory. L'attributo di gruppo di un file si usa per definire il sottoinsieme di utenti autorizzati a condividere l'accesso al file. Ad esempio, il proprietario di un file in un sistema UNIX può fare qualsiasi operazione sul file, mentre i membri del gruppo possono compiere un sottoinsieme di queste operazioni e il resto degli utenti un altro sottoinsieme. Il proprietario del file può definire l'esatto insieme di operazioni che i membri del gruppo e gli altri utenti possono eseguire. Maggiori dettagli sugli attributi che regolano i permessi sono trattati nel paragrafo successivo.

Gli identificatori del gruppo e del proprietario di un certo file o directory (ID) sono memorizzati insieme con gli altri attributi del file. Quando un utente richiede di compiere un'operazione su un file, per verificare se l'utente richiedente è il proprietario del file si può confrontare l'identificatore utente con l'attributo che identifica il proprietario. Analogamente si confrontano gli identificatori di gruppo. Ne risultano i permessi che l'utente ha sul file, e che il sistema considera per consentire o impedire l'operazione richiesta.

Molti sistemi operativi hanno più file system locali, inclusi volumi di un unico disco, o più volumi in diversi dischi connessi al sistema. In questi casi, la verifica degli identificatori e il confronto dei permessi si possono fare facilmente dopo l'operazione di montaggio dei file system.

File system remoti

I metodi con i quali i file si condividono in una rete sono cambiati molto, seguendo l'evoluzione della tecnologia delle reti e dei file. Un primo metodo consiste nel trasferimento dei file richiesto in modo esplicito dagli utenti, attraverso programmi come l'ftp. Un secondo metodo, molto diffuso, è quello del file system distribuito (distributed file system, DFS), che permette la visibilità nel calcolatore locale delle directory remote. Il terzo metodo, il World Wide Web, è, da un certo punto di vista, il contrario del primo. Per accedere ai file remoti si usa un programma di consultazione (browser), e operazioni distinte (essenzialmente un wrapper per l'ftp) per trasferirli.

L'ftp si usa sia per l'accesso anonimo sia per quello autenticato. L'accesso anonimo permette di trasferire file senza essere utenti accreditati nel sistema remoto. Il World Wide Web usa quasi esclusivamente lo scambio di file anonimo. Un DFS comporta un'integrazione molto più stretta tra il calcolatore che accede ai file remoti e il calcolatore che fornisce i file. Tale integrazione incrementa la complessità, illustrata in questo paragrafo.

Semantica della coerenza

La semantica della coerenza è un importante criterio per la valutazione di qualsiasi file system che consenta la condivisione dei file. Si tratta di una caratterizzazione del sistema che specifica la semantica delle operazioni in cui più utenti accedono contemporaneamente a un file condiviso. In particolare, questa semantica deve specificare quando le modifiche ai dati apportate da un utente possano essere osservate da altri utenti. La semantica è tipicamente realizzata come codice facente parte del codice del file system.

La semantica della coerenza è direttamente correlata agli algoritmi di sincronizzazione dei processi del [Capitolo 6](#). Tuttavia, a causa delle lunghe latenze e delle basse velocità di trasferimento dei dischi e delle reti, i complessi algoritmi descritti in tale capitolo di solito non s'impiegano per PI/O su file. Ad esempio, l'esecuzione di una transazione atomica su dischi remoti può coinvolgere molte comunicazioni di rete e molte letture e scritture nei dischi. I sistemi che tentano una così completa serie di funzioni tendono ad avere scarse prestazioni.

Nella trattazione seguente si suppone che una serie d'accessi, cioè letture e scritture, tentati da un utente allo stesso file sia sempre compresa tra una coppia di operazioni `open()` e `close()`. Tale serie d'accessi si chiama sessione di file. Per illustrare questo concetto si descrivono sommariamente qui di seguito alcuni esempi di semantica della coerenza.

Semantica UNIX

- Il file system di UNIX usa la seguente semantica della coerenza.
 - Le scritture in un file aperto da parte di un utente sono immediatamente visibili ad altri utenti che hanno aperto contemporaneamente lo stesso file.
 - Esiste un metodo di condivisione in cui gli utenti condividono il puntatore alla locazione corrente nel file, quindi l'avanzamento del puntatore da parte di un utente influisce su tutti gli utenti che condividono il file. In questo caso il file ha una singola immagine e tutti gli accessi si alternano (intercalandosi) a prescindere dalla loro origine.
- Nella semantica UNIX un file è associato a una singola immagine fisica, accessibile come una risorsa esclusiva. La contesa di quest'immagine singola determina il differimento dei processi utenti.

Semantica delle sessioni

- Il file system Andrew (Andrew file system, AFS) usa la seguente semantica della coerenza:
 - le scritture in un file aperto da un utente non sono visibili immediatamente ad altri utenti che hanno aperto contemporaneamente lo stesso file;
 - una volta chiuso il file, le modifiche apportate sono visibili solo nelle sessioni che iniziano successivamente. Le istanze del file già aperte non riportano queste modifiche.
- Secondo questa semantica, un file può essere temporaneamente associato a parecchie immagini, probabilmente diverse. Di conseguenza, più utenti possono eseguire accessi concorrenti di lettura o scrittura sulla rispettiva immagine del file senza subire ritardi. Non si impone quasi alcun vincolo nella gestione degli accessi.

Semantica dei file condivisi immutabili

- Un altro metodo è quello dei file condivisi immutabili; una volta che un file è stato dichiarato condiviso dal suo creatore, non può essere modificato. Un file immutabile presenta due caratteristiche chiave: il suo nome non si può riutilizzare e il suo contenuto non può essere modificato. Quindi il nome di un file immutabile indica che i contenuti di quel file sono fissi. Come si descrive nel Capitolo 17, la realizzazione di questa semantica in un sistema distribuito è semplice; infatti la condivisione è molto disciplinata poiché consente la sola lettura.

Protezione

La salvaguardia delle informazioni contenute in un sistema di calcolo dai danni fisici (la questione della affidabilità) e da accessi impropri (la questione della protezione) è fondamentale. Generalmente l'affidabilità è assicurata da più copie dei file. Molti calcolatori hanno programmi di sistema che copiano i file dai dischi ai nastri a intervalli regolari, ad esempio una volta al giorno, alla settimana o al mese; quest'operazione di copiatura può essere automatica, o controllata dall'intervento di un operatore. Lo scopo è quello di conservare copie di riserva utili nei casi in cui il file system andasse distrutto a causa di problemi dei dispositivi: errori di lettura o scrittura, cali o cadute della tensione elettrica, roture delle

testine, sporcizia, temperature estreme, atti vandalici, etc. I file possono inoltre essere cancellati accidentalmente, e anche errori di programmazione possono causare la perdita del contenuto dei file.

La protezione si può ottenere in molti modi. Per un piccolo sistema monoutente, la protezione si ottiene rimovendo fisicamente i dischetti e chiudendoli in un cassetto della scrivania oppure in un armadietto. In un sistema multiutente sono necessari altri metodi.

Tipi d'accesso

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**.

Il controllo offerto dai meccanismi di protezione si ottiene limitando i possibili tipi d'accesso. Gli accessi si permettono o si negano secondo diversi fattori, innanzitutto i tipi d'accesso richiesti. Si possono controllare distinte operazioni.

- **Lettura.** Lettura da file.
- **Scrittura.** Scrittura o riscrittura di file.
- **Esecuzione.** Caricamento di file in memoria ed esecuzione.
- **Aggiunta.** Scrittura di nuove informazioni in coda ai file.
- **Cancellazione.** Cancellazione di file e liberazione del relativo spazio per un possibile riutilizzo.
- **Elencazione.** Elencazione del nome e degli attributi dei file.

Si possono controllare anche altre operazioni, come ridenominazione, copiatura o modifica dei file.

Tuttavia, in molti sistemi queste funzioni di livello superiore si possono realizzare tramite un programma di sistema che compie alcune chiamate di sistema di livello inferiore, quindi è sufficiente garantire la protezione a livello inferiore. Ad esempio, la copiatura di un file si può realizzare semplicemente con una sequenza di richieste di lettura; in questo caso un utente con accesso per la lettura di un file può richiederne la copiatura, la stampa o altro.

Sono stati proposti molti meccanismi di protezione. Come sempre, ogni meccanismo presenta vantaggi e svantaggi, e deve essere appropriato alla particolare applicazione che richiede la protezione. Un piccolo calcolatore usato soltanto da pochi membri di un gruppo di ricerca non richiede la stessa protezione del sistema di calcolo di una grande società, usato per operazioni di ricerca, finanza e per il lavoro del personale.

Controllo degli accessi

I problema della protezione comunemente si affronta rendendo l'accesso dipendente dall'identità dell'utente. Più utenti possono richiedere diversi tipi d'accesso a un file o a una directory. Lo schema più generale per realizzare gli accessi dipendenti dall'identità consiste nell'associare una **access-control list** (ACL, lista di controllo degli accessi) a ogni file e directory; in tale lista sono specificati i nomi degli utenti e i relativi tipi d'accesso consentiti. Quando un utente richiede un accesso a un file specifico il sistema operativo esamina la lista di controllo degli accessi associata a quel file; se tale utente è presente nella lista si autorizza l'accesso, altrimenti si verifica una violazione della protezione e si nega l'accesso al file.

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 già nota;

- l'elemento della directory, precedentemente di dimensione fissa, richiede di 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 richiedono tipi di accesso simili.
- **Universo**. Tutti gli altri utenti del sistema.

Il più comune orientamento recente prevede la combinazione delle liste di controllo degli accessi con lo schema di controllo degli accessi per proprietario, gruppo e universo (più facile da realizzare).

Per definire la protezione, data questa più limitata classificazione, occorrono solo tre campi. Ogni campo è formato di un insieme di bit, ciascuno dei quali permette o impedisce l'accesso che gli è associato. Nel sistema UNIX, ad esempio, sono definiti tre campi di tre bit ciascuno: *rwx*, dove *r* controlla l'accesso per la lettura, *w* quello per la scrittura e *x* per l'esecuzione. Un campo distinto è riservato al proprietario del file, al gruppo proprietario e a tutti gli altri utenti. In questo schema, per registrare le informazioni di protezione sono necessari nove bit per file.

11. Realizzazione del File System

Struttura del file system

I dischi costituiscono la maggior parte della memoria secondaria in cui si conserva il file system. Hanno due caratteristiche importanti che ne fanno un mezzo conveniente per la memorizzazione dei file:

- 1) si possono riscrivere localmente; si può leggere un blocco dal disco, modificarlo e quindi scriverlo nella stessa posizione;
- 2) è possibile accedere direttamente a qualsiasi blocco di informazioni del disco, quindi risulta semplice accedere a qualsiasi file, sia in modo sequenziale sia in modo diretto, e passare da un file all'altro spostando le testine di lettura e scrittura e attendendo la rotazione del disco.

Anziché trasferire un byte alla volta, 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. Secondo l'unità a disco, la dimensione dei settori è compresa tra 32 byte e 4096 byte; di solito è pari a 512 byte.

Per fornire un efficiente e conveniente accesso al disco, il sistema operativo fa uso di uno o più file system che consentono di memorizzare, individuare e recuperare facilmente i dati. Un file system presenta due problemi di progettazione piuttosto diversi. Il primo riguarda la definizione dell'aspetto del file system agli occhi dell'utente. Questo compito implica la definizione di un file e dei suoi attributi, delle operazioni permette su un file e della struttura delle directory per l'organizzazione dei file. Il secondo riguarda la creazione di algoritmi e strutture dati che permettano di far corrispondere il file system logico ai dispositivi fisici di memoria secondaria.

Lo stesso file system è generalmente composto da molti livelli distinti. La struttura illustrata nella figura a destra è un esempio di struttura stratificata.

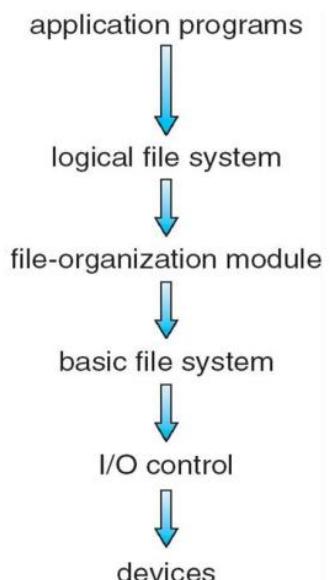
Ogni livello si serve delle funzioni dei livelli inferiori per creare nuove impiegate dai livelli superiori.

Il livello più basso, il **controllo dell'I/O**, costituito dai driver dei dispositivi e dai gestori dei segnali d'interruzione, si occupa del trasferimento delle informazioni tra memoria centrale e memoria secondaria. Un driver di dispositivo si può concepire come un traduttore che riceva comandi ad alto livello, come "recupera il blocco 123", e che emette specifiche istruzioni di basso livello per i dispositivi, usate dal controllore che fa da interfaccia tra i dispositivi di I/O e il resto del sistema. Un driver di dispositivo di solito scrive specifiche configurazioni di bit in specifiche locazioni della memoria del controllore di I/O per indicare quali azioni il dispositivo di I/O debba compiere, e in quali locazioni.

Il **file system di base** deve soltanto inviare dei generici comandi

all'appropriato driver di dispositivo per leggere e scrivere blocchi fisici nel disco. Ogni blocco fisico si identifica col suo indirizzo numerico nel disco, ad esempio unità 1, cilindro 73, superficie 2, settore 10. Questo strato gestisce inoltre buffer di memoria e le cache che conservano vari blocchi del file system, delle directory e dei dati. Un blocco viene allocato nel buffer prima che possa verificarsi il trasferimento di un blocco del disco. Quando il buffer è pieno, il gestore del buffer deve recuperare più spazio di memoria per il buffer oppure deve liberare spazio nel buffer per permettere il completamento di un I/O richiesto. Le cache servono a conservare metadati di file system usati frequentemente, in modo da migliorare le prestazioni. La gestione dei loro contenuti è quindi un punto critico per conseguire prestazioni ottimali del sistema.

Il **modulo di organizzazione dei file** è a conoscenza dei file e dei loro blocchi logici, così come dei blocchi fisici dei dischi. Conoscendo il tipo di allocazione dei file usato e la locazione dei file, può tradurre gli indirizzi dei blocchi logici, che il file system di base deve trasferire, negli indirizzi dei blocchi fisici. I blocchi



logici di ciascun file sono numerati da 0 (o 1) a n ; i blocchi fisici contenenti tali dati di solito non corrispondono ai numeri dei blocchi logici; per individuare ciascun blocco è quindi necessaria una traduzione. Il modulo di organizzazione dei file comprende anche il gestore dello spazio libero, che registra i blocchi non assegnati e li mette a disposizione del modulo di organizzazione dei file quando sono richiesti.

Infine, il **file system logico** gestisce i metadati; si tratta di tutte le strutture del file system, eccetto gli effettivi dati (il contenuto dei file). Il file system logico gestisce la struttura della directory per fornire al modulo di organizzazione dei file le informazioni di cui necessita, dato un nome simbolico di file. Mantiene le strutture di file tramite i **file control block** (FCB, blocchi di controllo dei file), contenenti informazioni sui file, come la proprietà, i permessi, e la posizione del contenuto.

Nei file system stratificati la duplicazione di codice è ridotta al minimo. Il controllo dell'I/O e, talvolta, il codice di base del file system, possono essere comuni a numerosi file system, che poi gestiscono il file system logico e i moduli per l'organizzazione dei file secondo le proprie esigenze. Sfortunatamente, la stratificazione può comportare un maggior overhead del sistema operativo, che può generare un conseguente decadimento delle prestazioni. L'utilizzo della stratificazione e le scelte sul numero di strati da impiegare e sulle loro funzionalità rappresentano una grande sfida per la progettazione di nuovi sistemi.

Implementazione del file system

Per realizzare un file system si usano parecchie strutture dati, sia nei dischi sia in memoria. Queste strutture variano secondo il sistema operativo e il file system, ma esistono dei principi generali. Nei dischi, il file system tiene informazioni su come eseguire l'avviamento di un sistema operativo memorizzato nei dischi stessi, il numero totale di blocchi, il numero e la locazione dei blocchi liberi, la struttura delle directory e i singoli file.

Fra le strutture presenti nei dischi ci sono le seguenti:

- Il **boot control block** (blocco di controllo dell'avviamento), contenente le informazioni necessarie al sistema per l'avviamento di un sistema operativo da quel volume; se il disco non contiene un sistema operativo, tale blocco è vuoto. Di solito è il primo blocco di un volume. Nell'UFS, si chiama blocco d'avviamento (boot block); nell'NTFS, settore d'avviamento della partizione (partition boot sector).
- I **volume control block** (blocchi di controllo dei volumi); ciascuno di loro contiene dettagli riguardanti il relativo volume (o partizione), come il numero e la dimensione dei blocchi nel disco, il contatore dei blocchi liberi e i relativi puntatori, il contatore dei blocchi di controllo dei file liberi e i relativi puntatori. Nell'UFS si chiama superblocco; nell'NTFS si chiama tabella principale dei file (master file table, MFT).
- Le **strutture delle directory** (una per file system) usate per organizzare i file. Nel caso dell'UFS comprendono i nomi dei file e i numeri di *inode* associati. Nel caso dell'NTFS sono memorizzate nella tabella principale dei file (master file table).
- I **file control block** (FCB), contenenti molti dettagli dei file, compresi i permessi d'accesso ai relativi file, i proprietari, le dimensioni e le locazioni dei blocchi di dati. Nell'UFS si chiamano *inode*; nell'NTFS, queste informazioni sono memorizzate all'interno della tabella principale dei file, che si serve di una struttura di base di dati relazionale, con una riga per ciascun file.

Le informazioni tenute in memoria servono sia per la gestione del file system sia per migliorare le prestazioni attraverso l'uso di cache. I dati si caricano al momento del montaggio e si eliminano allo smontaggio. Le strutture che vi possono essere incluse sono di diverso tipo:

- la **tabella di montaggio** interna alla memoria che contiene informazioni relative a ciascun volume montato;

- la **struttura della directory**, tenuta in memoria, contenente le informazioni relative a tutte le directory cui i processi hanno avuto accesso di recente (per le directory che costituiscono dei punti di montaggio, può essere presente un puntatore alla tabella dei volumi);
- la **tabella generale dei file aperti** (livello SO), contenente una copia del blocco di controllo del file per ciascun file aperto, insieme con altre informazioni;
- la **tabella dei file aperti** (livello processo) per ciascun processo, contenente un puntatore all'appropriato elemento della tabella generale dei file aperti, insieme con altre informazioni;
- i **buffer** conservano blocchi del file system durante la loro lettura o scrittura sul disco.

Le applicazioni, per creare un nuovo file, eseguono una chiamata al file system logico, il quale conosce il formato della struttura della directory. Esso crea e alloca un nuovo FCB. Il sistema carica quindi la directory appropriata in memoria, la aggiorna con il nome del nuovo file e con il blocco di controllo associato, e la scrive nuovamente sul disco. Una tipica struttura di blocco di controllo dei file (FCB) è illustrata nella figura a destra.

Alcuni sistemi operativi, compreso UNIX, trattano le directory esattamente come i file, distinguendole con un campo per il tipo che indica che si tratta di una directory. Altri, tra cui il sistema operativo Windows NT, dispongono di chiamate di sistema distinte per i file e le directory e trattano le directory come entità separate dai file. Indipendentemente da questioni strutturali, il file system logico può basarsi sul modulo che si occupa dell'organizzazione dei file per far corrispondere l'I/O su directory a numeri di blocchi di disco, che poi si passano al file system di base e al sistema per il controllo dell'I/O.

file permissions
file dates (create, access, write)
file owner, group, ACL
file size
file data blocks or pointers to file data blocks

Una volta creato un file, per essere usato per operazioni di I/O deve essere aperto. La chiamata di sistema `open()` passa un nome di file al file system. Per controllare se il file sia già in uso da parte di qualche processo, la chiamata `open()` dapprima esamina la tabella dei file aperti in tutto il sistema; in caso affermativo, aggiunge un elemento alla tabella dei file aperti del processo (per ogni processo che stia usando il file) che punta alla tabella dei file aperti in tutto il sistema. L'algoritmo può eliminare significativi rallentamenti. Una volta aperto il file, se ne ricerca il nome all'interno della directory. Alcune porzioni della struttura delle directory sono di solito tenute in memoria per accelerare le operazioni sulle directory. Una volta trovato il file, si copia l'FCB nella tabella generale dei file aperti, tenuta in memoria. Questa tabella non solo contiene l'FCB, ma tiene anche traccia del numero di processi che in quel momento hanno il file aperto.

Successivamente, si crea un elemento nella tabella dei file aperti del processo con un puntatore alla tabella generale e con alcuni altri campi. Questi altri campi possono comprendere un puntatore alla posizione corrente nel file e il tipo d'accesso richiesto all'apertura del file. La `open()` riporta un puntatore all'elemento appropriato nella tabella dei file aperti del processo, sicché tutte le operazioni sul file si svolgeranno usando questo puntatore. Il nome del file potrebbe non essere contenuto nella tabella dei file aperti, visto che, una volta che l'FCB appropriato è stato individuato nei dischi, il sistema non ne ha bisogno. Tuttavia, potrebbe venir memorizzato in una cache per risparmiare tempo sulle aperture successive dello stesso file. Il nome dato all'elemento della tabella è **file descriptor** (descrittore di file) in UNIX e **handle del file** in windows. Finché un file non viene chiuso, tutte le operazioni si compiono sulla tabella dei file aperti usando questo elemento.

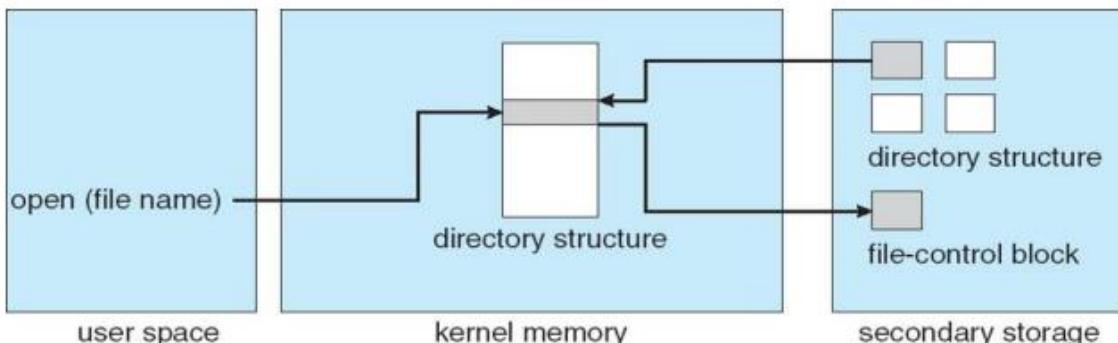
Quando un processo chiude il file, si cancella il relativo elemento nella tabella dei file aperti del processo e si decrementa il contatore associato al file nella tabella generale. Se tutti i processi che avevano aperto il file lo hanno chiuso, si riscrive l'informazione aggiornata sul file nella struttura delle directory nei dischi e si cancella il relativo elemento nella tabella generale dei file aperti.

Alcuni sistemi complicano ulteriormente lo schema descritto, usando il file system come interfaccia per altri aspetti del sistema, come la comunicazione in rete. Ad esempio, nell'UFS, la tabella generale dei file aperti contiene gli ino de e altre informazioni su file e directory, ma contiene anche informazioni simili per le connessioni di rete e i dispositivi. In questo modo si può usare un unico meccanismo per molteplici fini.

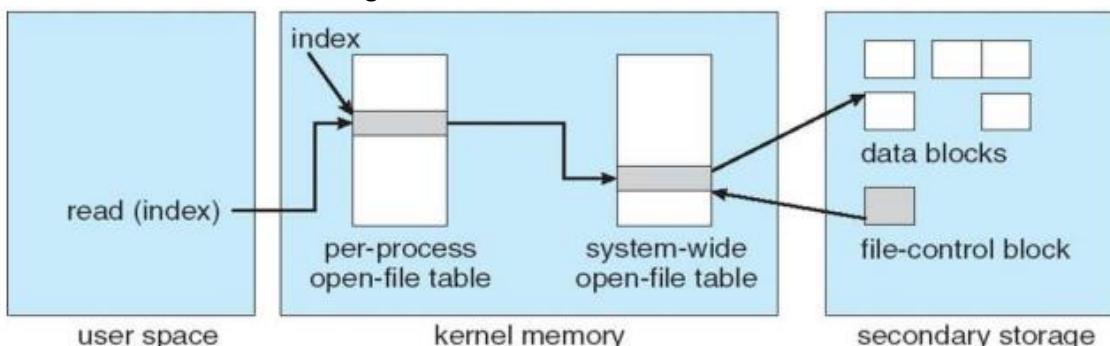
Le questioni concernenti l'uso delle cache per queste strutture non vanno però trascurate; usando questo schema, tutta l'informazione su un file aperto, eccetto i suoi effettivi blocchi di dati, è tenuta in memoria. Il sistema UN IX BSD è noto per il suo uso di cache ovunque sia possibile risparmiare su operazioni di I/O nei dischi. La sua frequenza media di successi nella cache, pari all'85 per cento, dimostra l'utilità di queste tecniche.

Le seguenti figure riassumono le strutture che si usano nella realizzazione di un file system.

Struttura del file system che si mantengono in memoria all'apertura di un file:



Struttura del file che si mantengono in memoria alla lettura di un file:



Partizioni e montaggio

Un disco si può suddividere in più partizioni dove ciascuna partizione è priva di struttura logica (raw partition) se non contiene alcun file system. Se nessun file system è appropriato, si usa un disco privo di struttura logica (raw disk). Il sistema operativo UNIX impiega una partizione priva di struttura per l'area di swap dei processi; per questo scopo usa un formato specifico. Allo stesso modo alcuni sistemi di gestione di basi di dati usano dischi privi di un'ordinaria struttura logica e formattano i dati secondo le proprie necessità. Un disco privo di struttura logica può anche contenere informazioni necessarie per sistemi [RAID](#) di gestione dei dischi, ad esempio le mappe di bit che indicano quali blocchi sono duplicati in altri dischi, e quali sono stati modificati e si devono aggiornare negli altri dischi.

Le informazioni relative all'avviamento del sistema si possono registrare in un'apposita partizione, che anche in questo caso ha un proprio formato, poiché nella fase d'avviamento il sistema non ha ancora caricato i driver di dispositivo del file system e quindi non può interpretarne il formato. Questa partizione consiste piuttosto in una serie sequenziale di blocchi, che si carica in memoria come un'immagine.

L'esecuzione dell'immagine comincia a una locazione prefissata, ad esempio il primo byte. L'immagine d'avviamento può contenere più informazioni di quelle che servono per un singolo sistema operativo.

Nella fase di caricamento del sistema operativo, si esegue il montaggio della **root partition**, che contiene il kernel del sistema operativo e in alcuni casi altri file di sistema. Secondo il sistema operativo, il montaggio degli altri volumi avviene automaticamente in questa fase oppure si può compiere successivamente in modo esplicito. Durante l'operazione di montaggio, il sistema verifica che il dispositivo contenga un file system valido chiedendo al dispositivo di leggere la directory di dispositivo e verificando che la directory abbia il formato corretto. Se così non fosse, è necessaria una verifica della coerenza della partizione e una eventuale correzione, con o senza l'intervento dell'utente. Infine, il sistema annota nella struttura della **tavella di montaggio** in memoria che un file system è stato montato insieme al tipo di file system. I dettagli di questa funzione dipendono dal sistema operativo.

In UNIX, l'operazione di montaggio di un file system si può compiere in qualsiasi directory. Questa funzione si realizza impostando un flag nella copia dell'inode tenuta in memoria di quella directory, che segnala che la directory è un punto di montaggio. Un campo dell'inode punta a un elemento nella tabella di montaggio, che indica quale dispositivo è montato in quella posizione. L'elemento della tabella di montaggio contiene un puntatore al superblocco del file system in quel dispositivo. Questo schema permette al sistema operativo di attraversare facilmente la propria struttura della directory, passando da un file system all'altro secondo le necessità.

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 separate per ciascun tipo di file system. Al contrario, la maggior parte dei sistemi operativi, compreso UNIX, impiega tecniche orientate agli oggetti per semplificare e organizzare in maniera modulare la soluzione. L'uso di queste tecniche rende possibile la realizzazione, nella stessa struttura, di tipi di file system molto diversi tra loro, compresi i file system di rete, come l'NFS. Gli utenti possono accedere a file contenuti in diversi file system nei dischi locali, o anche in file system disponibili tramite la rete.

Per isolare le funzioni di base delle chiamate di sistema dai dettagli di realizzazione si adoperano apposite strutture dati. In questo modo la realizzazione del file system si articola in tre strati principali, riportati in modo schematico nella figura seguente.

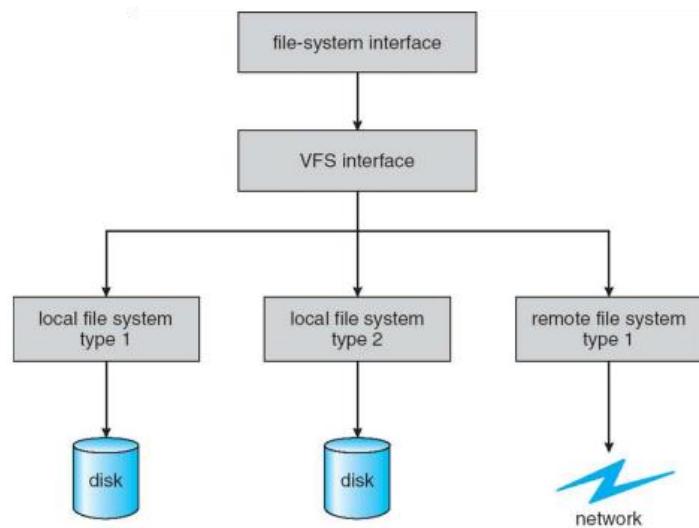
Il primo strato è l'interfaccia del file system, basata sulle chiamate di sistema `open()`, `read()`, `write()` e `close()` e sui descrittori di file.

Il secondo strato si chiama strato del **virtual file system** (VFS) e svolge due funzioni importanti:

- 1) Separa le operazioni generiche del file system dalla loro realizzazione definendo un'interfaccia VFS uniforme.

Nello stesso calcolatore possono coesistere più interfacce VFS, che permettono un accesso trasparente a diversi tipi di file system montati localmente.

- 2) Permette la rappresentazione univoca di un file su tutta la 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. (Gli inode di UNIX sono unici solo all'interno di un singolo file system.) Tale unicità per tutta la rete è richiesta per la gestione del file system di rete. Il kernel contiene una struttura vnode per ciascun nodo attivo, sia che si tratti di un file sia che si tratti di una directory.



Il VFS attiva le operazioni specifiche del file system per gestire le richieste locali secondo i tipi di file system, e invoca le procedure del protocollo NFS per le richieste remote. Gli handle del file si costruiscono secondo i vnode relativi e s'inviano a queste procedure come argomenti. Lo strato che realizza il protocollo NFS è il più basso dell'architettura.

Esaminiamo succintamente l'architettura **VFS di Linux**. I quattro tipi più importanti di oggetti definiti in questo sistema sono:

- l'oggetto **inode**, che rappresenta il singolo file;
- l'oggetto **file**, che rappresenta un file aperto;
- l'oggetto **superblock**, che rappresenta un intero file system;
- l'oggetto **dentry**, che rappresenta il singolo elemento della directory

Per ognuno di questi tipi, VFS specifica un insieme di operazioni da implementare. Ciascun oggetto di uno di questi tipi contiene un puntatore a una tabella di funzioni; questa, alla sua volta, contiene gli indirizzi delle effettive funzioni che implementano le operazioni richieste dalla specifica dell'oggetto. Per esempio, una versione semplificata della API di alcune delle operazioni dell'oggetto file è:

- int open(...) → apre il file.
- ssize_t read(...) → legge dal file.
- ssize_t write(...) → scrive sul file.
- int mmap(...) → mappa il file in memoria.

Ogni implementazione dell'oggetto file per uno specifico tipo di file deve implementare tutte le funzioni specificate nella definizione dell'oggetto file.

Questo schema permette a VFS di eseguire operazioni su uno degli oggetti in questione invocando l'appropriata funzione della tabella delle funzioni dell'oggetto, senza dover conoscere i dettagli dello specifico oggetto. Per esempio, VFS non sa, né vuol sapere, se un certo inode rappresenti un file su disco, un file di directory o un file remoto. L'implementazione appropriata dell'operazione read() per l'oggetto in questione è comunque reperibile sempre nel medesimo punto all'interno della tabella delle funzioni: invocando tale implementazione, VFS può disinteressarsi dei dettagli legati all'effettiva lettura dei dati.

Implementazione delle directory

La selezione degli algoritmi di allocazione e degli algoritmi di gestione delle directory ha un grande effetto sull'efficienza, le prestazioni e l'affidabilità del file system. Per tale ragione è necessario comprendere i vari aspetti di questi algoritmi, trattati di seguito

Lista lineare

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. Per creare un nuovo file occorre prima esaminare la directory per essere sicuri che non esista già un file con lo stesso nome, quindi aggiungere un nuovo elemento alla fine della directory. Per cancellare un file occorre cercare nella directory il file con quel nome, quindi rilasciare lo spazio che gli era assegnato. Esistono vari metodi per riutilizzare un elemento della directory: si può contrassegnare l'elemento come non usato (attribuendogli un nome speciale, come un nome vuoto, oppure un bit d'uso in ogni elemento), oppure può essere aggiunto a una lista di elementi di directory liberi; una terza possibilità prevede la copiatura dell'ultimo elemento della directory in una locazione liberata e la diminuzione della lunghezza della directory. Per ridurre il tempo di cancellazione di un file si può usare anche una lista concatenata.

Il vero svantaggio dato da una lista lineare di elementi di directory è dato dalla ricerca lineare di un file. Le informazioni sulla directory vengono usate frequentemente, e gli utenti avvertirebbero una gestione lenta dell'accesso a tali informazioni. In effetti, molti sistemi operativi impiegano una cache per memorizzare le

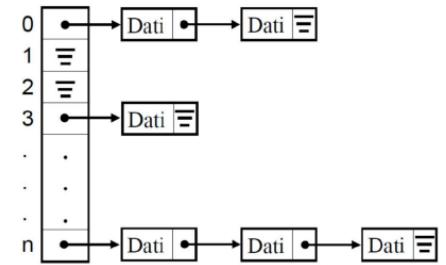
informazioni sulla directory usata più recentemente. La presenza nella cache delle informazioni richieste ne evita la continua rilettura dai dischi. Una lista ordinata permette una ricerca binaria e riduce il tempo medio di ricerca, tuttavia il requisito dell'ordinamento può complicare la creazione e la cancellazione di file, poiché, per tenere ordinata la lista, può essere necessario spostare quantità notevoli di informazioni sulla directory. Un vantaggio della lista ordinata è che consente di produrre l'elenco ordinato del contenuto della directory senza una fase d'ordinamento separata.

Tabella hash

Un'altra struttura dati che si usa per realizzare le directory è la tabella hash. In questo metodo una lista lineare contiene gli elementi di directory, ma si usa anche una struttura dati hash. La tabella hash riceve un valore calcolato usando come operando il nome del file e riporta un puntatore al nome del file nella lista lineare. 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 evitare collisioni, cioè situazioni in cui da due nomi di file si ottiene un riferimento alla stessa locazione.

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.

Alternativamente, ciascun elemento della tabella hash, anziché un singolo valore, può essere una lista concatenata; ciò consente di risolvere le collisioni aggiungendovi il nuovo elemento. Le ricerche vengono alquanto rallentate, poiché la ricerca per nome può richiedere l'attraversamento della lista concatenata degli elementi in collisione della tabella hash; probabilmente tale metodo è comunque più veloce di una ricerca lineare nell'intera directory.



Metodi di allocazione

La natura ad accesso diretto dei dischi permette una certa flessibilità nella realizzazione dei file. In quasi tutti i casi, molti file si memorizzano nello stesso disco. Il problema principale consiste dunque nell'allocare lo spazio per questi file in modo che lo spazio nel disco sia usato efficientemente e l'accesso ai file sia rapido. Esistono tre metodi principali per l'allocazione dello spazio di un disco; può essere infatti contigua, concatenata o indicizzata. Ciascuno di questi metodi presenta vantaggi e svantaggi.

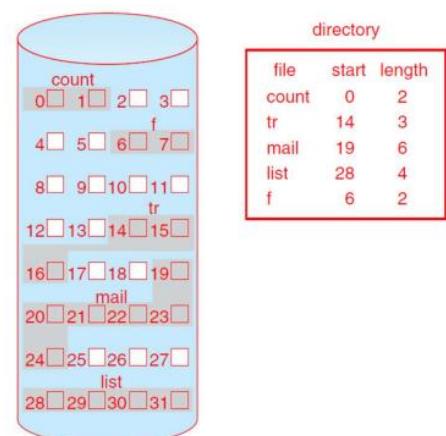
Allocazione contigua

Per usare il metodo di allocazione contigua, ogni file deve occupare un insieme di blocchi contigui del disco. Gli indirizzi del disco definiscono un ordinamento lineare nel disco stesso. Con questo ordinamento l'accesso al blocco $b + 1$ dopo il blocco b non richiede normalmente alcuno spostamento della testina. Se la testina deve essere spostata dall'ultimo settore di un cilindro al primo settore del cilindro successivo lo spostamento avviene su una sola traccia. Quindi, il numero dei posizionamenti (seek) richiesti per accedere a file il cui spazio è allocato in modo contiguo è trascurabile, così com'è trascurabile il tempo di ricerca (seek time), quando quest'ultimo è necessario.

L'allocazione contigua dello spazio per un file è definita dall'indirizzo del primo blocco (inteso come numero di blocco) e dalla lunghezza (espressa in blocchi). Se il file è lungo n blocchi e comincia dalla locazione b , allora occupa i blocchi $b, b + 1, \dots, b + n - 1$.

L'elemento di directory per ciascun file indica l'indirizzo del blocco d'inizio e la lunghezza dell'area assegnata per questo file (vedi figura).

Accedere a un file il cui spazio è assegnato in modo contiguo è facile.



Quando si usa un accesso sequenziale, il file system memorizza l'indirizzo dell'ultimo blocco cui è stato fatto riferimento e, se è necessario, legge il blocco successivo. Nel caso di un accesso diretto al blocco i di un file che comincia al blocco b si può accedere immediatamente al blocco $b + i$. Quindi, sia l'accesso sequenziale sia quello diretto si possono gestire con l'allocazione contigua.

L'allocazione contigua presenta però alcuni problemi; una difficoltà riguarda l'individuazione dello spazio per un nuovo file. La realizzazione del sistema di gestione dello spazio libero, illustrata nel [paragrafo successivo](#), determina il modo in cui tale compito viene eseguito. Si può usare ogni sistema di gestione, anche se alcuni sono più lenti di altri.

I 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. I più comuni criteri di scelta di un buco libero da un insieme di buchi disponibili sono quelli del primo buco abbastanza grande (*first-fit*) e del più piccolo tra i buchi abbastanza grandi (*best-fit*).

Questi algoritmi soffrono della **frammentazione esterna**: assegnando e liberando lo spazio per i file, lo spazio libero dei dischi viene frammentato in tanti piccoli pezzi. La frammentazione esterna si ha ogniqualvolta lo spazio libero è suddiviso in pezzi, e diviene un problema quando il più grande di tali pezzi contigui non è sufficiente a soddisfare una richiesta; la memoria viene frammentata in tanti buchi, nessuno dei quali è abbastanza grande da contenere i dati. Secondo la capacità dei dischi e la dimensione media dei file, la frammentazione esterna può essere un problema più o meno grave.

Una strategia per prevenire la perdita di una quantità significativa di spazio sul disco a causa della frammentazione esterna consiste nel copiare un intero file system su un altro disco o nastro. Quindi si liberava completamente il primo disco creando un ampio spazio libero contiguo. La procedura provvedeva poi a copiare nuovamente i file nel disco, assegnando tale spazio contiguo. Questo schema compatta efficacemente tutto lo spazio libero in uno spazio contiguo, risolvendo il problema della frammentazione. Questa funzionalità può essere eseguita offline (non è possibile usare il volume) o online (peggiora le prestazioni).

Un altro problema che riguarda l'allocazione contigua è la determinazione della quantità di spazio necessaria per un file. Quando si crea un file, occorre trovare e allocare lo spazio di cui necessita. Esiste il problema di conoscere la dimensione del file da creare; in alcuni casi questa dimensione si può stabilire in modo abbastanza semplice, ad esempio quando si copia un file esistente; in generale, tuttavia, non è facile stimare la dimensione di un file che deve contenere dati emessi da un programma.

Se un file riceve poco spazio, può essere impossibile estenderlo. Esistono allora due possibilità. La prima è che il programma utente si possa terminare con un idoneo messaggio d'errore. L'utente deve allora allocare più spazio ed eseguire di nuovo il programma. L'altra possibilità consiste nel trovare un buco più grande, copiare il contenuto del file nel nuovo spazio e rilasciare lo spazio precedente.

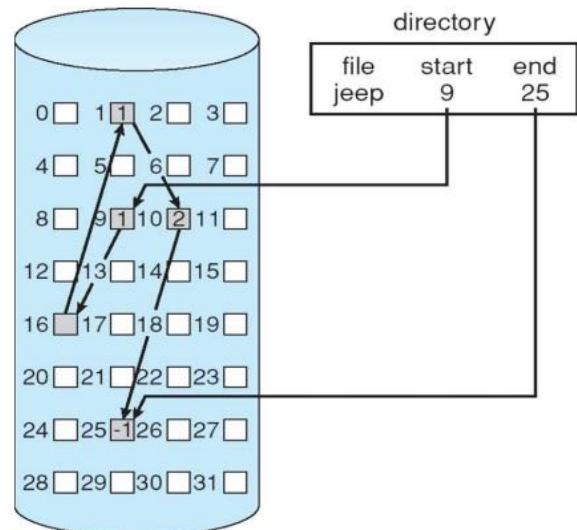
Per ridurre al minimo questi inconvenienti, alcuni sistemi operativi fanno uso di uno schema di **allocazione contigua estesa**: inizialmente si assegna una porzione di spazio contiguo, e se questa non è abbastanza grande si aggiunge un'altra porzione di spazio, un'estensione. La locazione dei blocchi dei file si registra come una locazione e un numero dei blocchi, insieme con l'indirizzo del primo blocco dell'estensione seguente.

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. Ad

esempio, un file di cinque blocchi può cominciare dal blocco 9, continuare al blocco 16, quindi al blocco 1, al blocco 10 e infine terminare al blocco 25 (vedi figura). Ogni blocco contiene un puntatore al blocco successivo. Questi puntatori non sono disponibili all'utente, quindi se ogni blocco è formato di 512 byte e un indirizzo del disco (il puntatore) richiede 4 byte, l'utente vede blocchi di 508 byte.

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. Questo puntatore s'inizializza a *nil* (valore del puntatore di fine lista) a indicare un file vuoto; anche il campo della dimensione s'imposta a 0. Un'operazione di scrittura nel file determina la ricerca di un blocco libero attraverso il sistema di gestione dello spazio libero, la scrittura in tale blocco, e la concatenazione di tale blocco alla fine del file. Per leggere un file occorre semplicemente leggere i blocchi seguendo i puntatori da un blocco all'altro. Con l'allocazione concatenata non esiste frammentazione esterna e per soddisfare una richiesta si può usare qualsiasi blocco libero della lista. Inoltre non è necessario dichiarare la dimensione di un file al momento della sua creazione. Un file può continuare a crescere finché sono disponibili blocchi liberi, di conseguenza non è mai necessario compattare lo spazio del disco.



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. Ogni accesso a un puntatore implica una lettura del disco, e talvolta un posizionamento della testina. Di conseguenza, per file il cui spazio è assegnato in modo concatenato, la funzione d'accesso diretto è inefficiente.

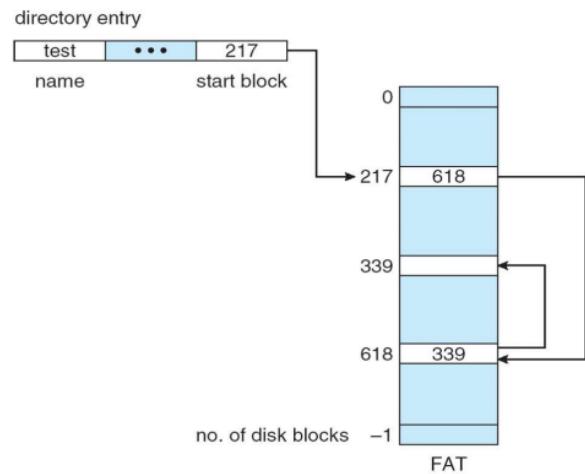
Un altro svantaggio dell'allocazione concatenata riguarda lo spazio richiesto per i puntatori. Se un puntatore richiede 4 byte di un blocco di 512 byte, allora lo 0,78 per cento del disco è usato per i puntatori anziché per le informazioni: ogni file richiede un po' più spazio di quanto ne richiederebbe altrimenti.

La soluzione più comune a questo problema consiste nel riunire un certo numero di blocchi contigui in **cluster** (gruppi di blocchi), e nell'allocare i cluster anziché i blocchi. Ad esempio, il file system può definire cluster di 4 blocchi e operare nel disco soltanto per unità di cluster. Così i puntatori usano una quantità di spazio di disco che si riduce in modo proporzionale al numero di cluster. Questo metodo permette che la corrispondenza tra blocchi logici e blocchi fisici rimanga semplice, ma migliora la produttività del disco: si hanno meno posizionamenti della testina del disco e diminuisce lo spazio necessario per l'allocazione dei blocchi e la gestione della lista dei blocchi liberi. Il costo di questo metodo è dato da un incremento della frammentazione interna, poiché se un cluster è parzialmente pieno si spreca più spazio di quanto se ne sprecherebbe con un solo blocco parzialmente pieno. I cluster si possono usare per ottimizzare l'accesso ai dischi in molti altri algoritmi, quindi s'impiegano nella maggior parte dei sistemi operativi.

Un altro problema 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. Un errore di programmazione del sistema operativo oppure un errore di un'unità a disco potrebbero causare il prelevamento del puntatore errato. Questo errore, a sua volta, potrebbe causare il collegamento alla lista dei blocchi liberi oppure a un altro file. Una soluzione parziale a tale problema consiste nell'usare liste

doppiamente concatenate oppure nel memorizzare il nome del file e il relativo numero di blocco in ogni blocco; questi schemi però sono ancora più onerosi per ogni file.

Una variante importante del metodo di allocazione concatenata consiste nell'uso della **file allocation table** (FAT, tabella di allocazione dei file). Per contenere tale tabella si riserva una sezione del disco all'inizio di ciascun volume; la FAT ha un elemento per ogni blocco del disco ed è indicizzata dal numero di blocco; 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. Questa catena continua fino all'ultimo blocco, che ha come elemento della tabella un valore speciale di fine del file. I blocchi inutilizzati sono indicati nella tabella da un valore 0. L'allocazione di un nuovo blocco a un file implica semplicemente la localizzazione del primo elemento della tabella con valore 0 e la sostituzione del valore di fine del file precedente con l'indirizzo del nuovo blocco; lo 0 è quindi sostituito con il valore di fine del file. Un esempio esplicativo di tale metodo è dato dalla struttura della FAT della figura a destra, dove il file in questione è formato dai blocchi 217, 618 e 339.



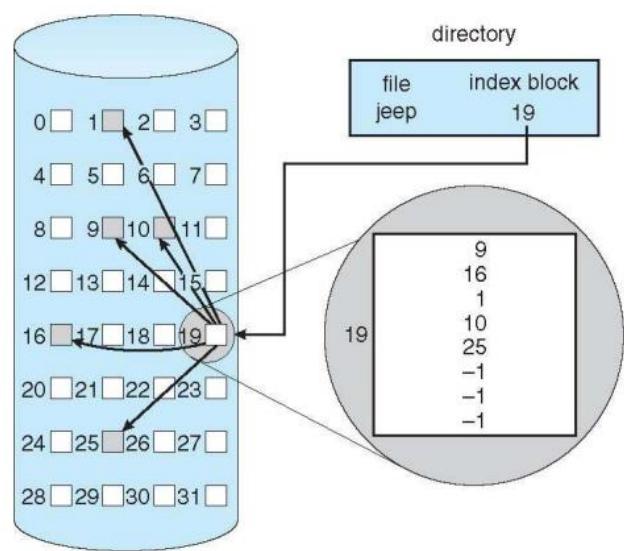
Lo schema di allocazione basato sulla FAT, se non si usa una cache, può causare un significativo numero di posizionamenti della testina. La testina del disco deve spostarsi all'inizio del volume per leggere la FAT e trovare la locazione del blocco in questione, quindi raggiungere la locazione del blocco stesso; nel caso peggiore sono necessari ambedue i movimenti per ciascun blocco. Un vantaggio è dato dall'ottimizzazione del tempo d'accesso diretto, poiché la testina del disco può trovare la locazione di ogni blocco leggendo le informazioni contenute nella FAT.

Allocazione indicizzata

L'allocazione concatenata risolve il problema della frammentazione esterna e quello della dichiarazione delle dimensioni dei file, entrambi presenti nell'allocazione contigua. Tuttavia, 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 ha il proprio blocco indice: si tratta di un array d'indirizzi di blocchi del disco. L' i -esimo elemento del blocco indice punta all' i -esimo blocco del file. La directory contiene l'indirizzo del blocco indice, com'è illustrato nella figura. Per individuare e leggere l' i -esimo blocco occorre usare il puntatore che si trova nell' i -esimo elemento del blocco indice, per poi localizzare e leggere il blocco desiderato.

Una volta creato il file, tutti i puntatori del blocco indice sono impostati *nil*. Quando si scrive l' i -esimo blocco per la prima volta, il gestore dei blocchi liberi fornisce un blocco; l'indirizzo di questo blocco viene inserito nell' i -esimo elemento del blocco indice. Poiché ogni blocco libero del disco può soddisfare

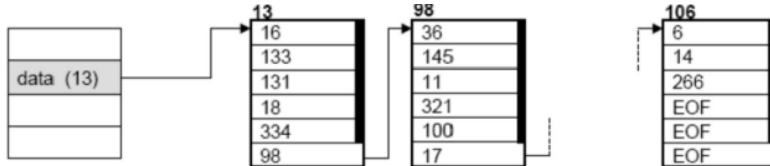


una richiesta di maggiore spazio, l'allocazione indicizzata consente l'accesso diretto senza soffrire di frammentazione esterna.

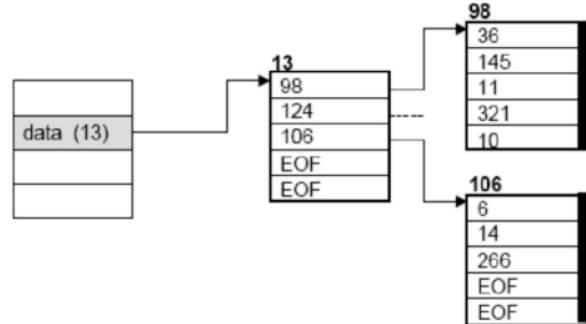
Lo spazio aggiuntivo richiesto dai puntatori del blocco indice è generalmente maggiore dello spazio aggiuntivo necessario per l'allocazione concatenata. Si consideri il comune caso di un file con uno o due blocchi; con l'allocazione concatenata si perde il solo spazio di un puntatore per blocco, complessivamente uno o due puntatori; con l'allocazione indicizzata occorre allocare un intero blocco indice, anche se solo uno o due puntatori sono diversi da *nil*.

Questo punto solleva la questione della dimensione del blocco indice. Ogni file deve avere un blocco indice, quindi è auspicabile che questo sia quanto più piccolo è 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 gestire questa situazione.

- **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. Ad esempio, un blocco indice può contenere una piccola intestazione in cui sono riportati il nome del file e l'insieme dei primi 100 indirizzi del blocco di disco. L'indirizzo successivo, vale a dire l'ultima parola del blocco indice, è *nil* (per un file piccolo) oppure è un puntatore a un altro blocco indice (per un file lungo).

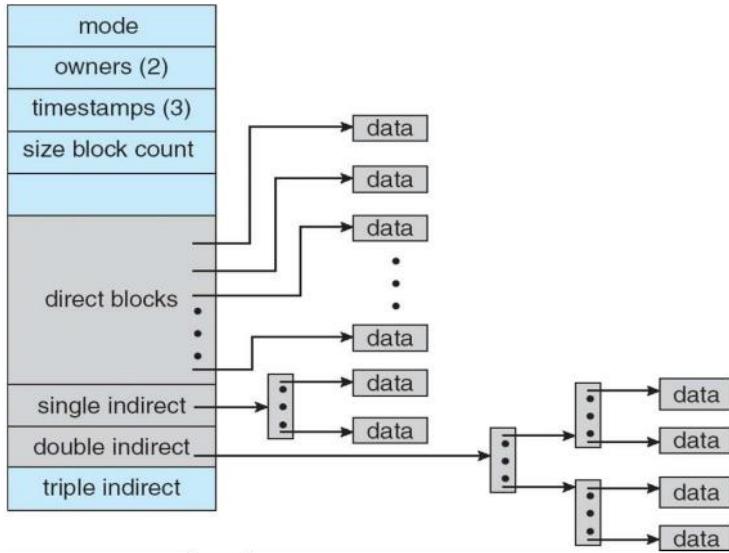


- **Indice a più livelli.** Una variante della rappresentazione concatenata consiste nell'impiego di un blocco indice di primo livello che punta a un insieme di blocchi indice di secondo livello che, a loro volta, puntano ai blocchi dei file. Per accedere a un blocco, il sistema operativo usa l'indice di primo livello, con il quale individua il blocco indice di secondo livello, e con esso trova il blocco di dati richiesto. Questo metodo può continuare fino a un terzo o quarto livello, secondo la massima dimensione desiderata del file. Con blocchi di 4096 byte si possono memorizzare 1024 puntatori di 4 byte in un blocco indice. Due livelli di indici consentono 1.048.576 blocchi di dati, che permettono di avere file sino a 4 GB.



- **Schema combinato:** UNIX UFS. Un'altra possibilità, è la soluzione adottata nell'UFS, 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. Quindi, i dati per piccoli file (non più di 12 blocchi) non richiedono un blocco indice distinto. Se la dimensione dei blocchi è di 4 KB, è possibile accedere direttamente fino a 48 KB di dati. Gli altri tre puntatori puntano a blocchi indiretti. Il primo puntatore di blocco indiretto è l'indirizzo di un blocco indiretto singolo; si tratta di un blocco indice che non contiene dati, ma indirizzi di blocchi che contengono dati. Quindi c'è un puntatore di blocco indiretto doppio contenente l'indirizzo di un blocco che a sua volta contiene gli indirizzi di blocchi contenenti puntatori agli effettivi blocchi di dati. L'ultimo puntatore contiene l'indirizzo di un blocco indiretto triplo. Con questo metodo, il numero dei blocchi che si può allocare a un file supera la quantità di spazio che possono indirizzare i puntatori a file di 4 byte usati da molti sistemi operativi. Un puntatore a file di 32 bit consente di arrivare a soli 232 byte, 4 GB. Molte versioni del sistema operativo UNIX, tra le quali Solaris e l'IBM AIX ora gestiscono puntatori a file sino a 64 bit. Puntatori di questa dimensione permettono di avere file e file system di dimensioni dell'ordine dei

terabyte. Un inode UNIX è mostrato nella figura sottostante.



Scelta del metodo di allocazione

I metodi d'allocazione presentati hanno diversi livelli di efficienza di memorizzazione e differenti tempi d'accesso ai blocchi di dati; entrambi i fattori sono importanti nella scelta del metodo o dei metodi d'allocazione più adatti da impiegare in un sistema operativo.

Prima di scegliere un metodo di allocazione, è necessario determinare il modo in cui si usano i sistemi: un sistema con una prevalenza di accessi sequenziali farà uso di un metodo differente da quello di un sistema con una prevalenza di accessi diretti.

Per qualsiasi tipo d'accesso, l'allocazione contigua richiede un solo accesso per ottenere un blocco. Poiché è facile tenere l'indirizzo iniziale del file in memoria, si può calcolare immediatamente l'indirizzo del disco dell' i -esimo blocco, oppure del blocco successivo, e leggerlo direttamente. Con l'allocazione concatenata si può tenere in memoria anche l'indirizzo del blocco successivo e leggerlo direttamente. Questo metodo è valido per l'accesso sequenziale mentre, per quel che riguarda l'accesso diretto, un accesso all' i -esimo blocco può richiedere i letture del disco. Questo spiega perché l'allocazione concatenata non si dovrebbe usare per un'applicazione che richiede accessi diretti.

Da tutto ciò segue che alcuni sistemi gestiscono i file ad accesso diretto usando l'allocazione contigua, e i file ad accesso sequenziale tramite l'allocazione concatenata. Per questi sistemi, il tipo d'accesso si deve dichiarare al momento della creazione del file. Un file creato per l'accesso sequenziale è un file concatenato e non si può usare per l'accesso diretto. Un file creato per l'accesso diretto è contiguo e consente entrambi i tipi d'accesso, purché se ne dichiari la lunghezza massima al momento della sua creazione. In questo caso, il sistema operativo deve avere strutture dati idonee e algoritmi capaci di gestire entrambi i metodi di allocazione. I file si possono convertire da un tipo all'altro creando un nuovo file del tipo desiderato, nel quale si copia il contenuto del vecchio file; quest'ultimo si può quindi cancellare e il nuovo file rinominare.

L'allocazione indicizzata è più complessa. Se il blocco indice è già in memoria, l'accesso può essere diretto. Tuttavia, per tenere il blocco indice in memoria occorre una quantità di spazio considerevole. Se questo spazio di memoria non è disponibile, occorre leggere prima il blocco indice e quindi il blocco di dati desiderato. Per un indice a due livelli possono essere necessarie due letture del blocco indice. Se un file è estremamente grande, per compiere l'accesso a un blocco che si trovi vicino alla fine del file, prima di leggere il blocco dei dati occorre leggere tutti i blocchi indice per seguire la catena dei puntatori. Quindi le prestazioni dell'allocazione indicizzata dipendono dalla struttura dell'indice, dalla dimensione del file e dalla posizione del blocco desiderato.

Alcuni sistemi combinano l'allocazione contigua con l'allocazione indicizzata, usando quella contigua per i file piccoli (fino a tre o quattro blocchi) e passando automaticamente a quella indicizzata per i file grandi. Poiché generalmente i file sono piccoli, e in questo caso l'allocazione contigua è efficiente, le prestazioni medie possono risultare abbastanza buone.

Gestione dello spazio libero

Poiché la quantità di spazio dei dischi è limitata, è necessario riutilizzare lo spazio lasciato dai file cancellati per scrivere, se è possibile, nuovi file (i dischi ottici a una sola scrittura permettono una sola scrittura in qualsiasi settore e quindi il riutilizzo è fisicamente impossibile). 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. Per creare un file occorre cercare nella lista dello spazio libero la quantità di spazio necessaria e assegnarla al nuovo file, quindi rimuovere questo spazio dalla lista. Quando si cancella un file, si aggiungono alla lista dello spazio libero i blocchi di disco a esso assegnati. A dispetto del suo nome, la lista dello spazio libero potrebbe non essere realizzata come una lista, come vedremo più avanti.

Vettore di bit o BitMap

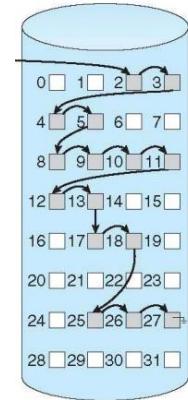
Spesso la lista dello spazio libero si realizza come una mappa di bit, o vettore di bit. Ogni blocco è rappresentato da un bit: se il blocco è libero, il bit è 1, se il blocco è assegnato il bit è 0.

I 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; in effetti, molti calcolatori forniscono istruzioni di manipolazione dei bit utilizzabili con efficacia a tale scopo. Sfortunatamente, i vettori di bit sono efficienti solo se tutto il vettore è mantenuto in memoria centrale, e viene di tanto in tanto scritto in memoria secondaria allo scopo di consentire eventuali operazioni di ripristino; è possibile tenere il vettore in memoria centrale se i dischi sono piccoli, come quelli usati nei microcalcolatori; tale soluzione non è applicabile ai dischi più grandi.

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.

Questo primo blocco contiene un puntatore al successivo blocco libero, e così via. Questo schema non è efficiente; per attraversare la lista è infatti necessario leggere ogni blocco, con un notevole tempo di I/O. Fortunatamente la necessità di attraversare la lista dello spazio libero non è frequente. Di solito il sistema operativo ha semplicemente bisogno di un blocco libero perché possa assegnarlo a un file, quindi si usa il primo blocco della lista. Il metodo che fa uso della FAT include il conteggio dei blocchi liberi nella struttura dati per l'allocazione; non è necessario un metodo separato.



Raggruppamento

Una possibile modifica del metodo della lista dello spazio libero prevede la memorizzazione degli indirizzi di n blocchi liberi nel primo di questi. I primi $n - 1$ di questi blocchi sono effettivamente liberi; l'ultimo blocco contiene gli indirizzi di altri n blocchi liberi, e così via. L'importanza di questo metodo, diversamente dall'ordinaria lista concatenata, è data dalla possibilità di trovare rapidamente gli indirizzi di un gran numero di blocchi liberi.

Conteggio

Un altro orientamento sfrutta il fatto che, generalmente, più blocchi contigui si possono allocare o liberare contemporaneamente, soprattutto quando lo spazio viene allocato usando l'algoritmo di allocazione contigua o attraverso l'uso di cluster. Quindi, anziché tenere una lista di n indirizzi liberi, è sufficiente tenere l'indirizzo del primo blocco libero e il numero n di blocchi liberi contigui che seguono il primo blocco. Ogni elemento della lista dello spazio libero è formato da un indirizzo del disco e un contatore. Anche se

ogni elemento richiede più spazio di quanto ne richieda un semplice indirizzo del disco, se il contatore è generalmente maggiore di 1 la lista complessiva è più corta.

Efficienza e prestazioni

Dopo avere descritto le opzioni di allocazione dei blocchi e di gestione delle directory, è possibile considerare i loro effetti sulle prestazioni e l'efficienza d'uso dei dischi. I dischi tendono di solito a essere il principale collo di bottiglia per le prestazioni di un sistema, essendo i più lenti tra i componenti più rilevanti di un calcolatore. In questo paragrafo si considerano diverse tecniche utili per migliorare l'efficienza e le prestazioni della memoria secondaria.

Efficienza

L'uso efficiente di un disco dipende fortemente dagli algoritmi usati per l'allocatione del disco e la gestione delle directory. Ad esempio, gli inode di UNIX sono assegnati preventivamente in un volume. Anche un disco "vuoto" impiega una certa percentuale del suo spazio per gli inode. D'altra parte, l'allocatione preventiva degli inode e la loro distribuzione nel volume migliorano le prestazioni del file system. Queste migliori prestazioni sono il risultato degli algoritmi di allocatione e di gestione dei blocchi liberi adottati da UNIX, i quali cercano di mantenere i blocchi di dati di un file vicini al blocco che ne contiene X inode allo scopo di ridurre il tempo di ricerca.

Si devono tenere in considerazione anche il tipo di dati normalmente contenuti in un elemento di una directory (o di un *inode*). Di solito si memorizza la *data dell'ultima scrittura* per fornire informazioni all'utente e per determinare se per il file occorre la creazione o l'aggiornamento di una copia di riserva. Alcuni sistemi mantengono anche la *data dell'ultimo accesso* per consentire all'utente di risalire all'ultima volta che un file è stato letto. Per mantenere queste informazioni, ogniqualvolta si legge un file, si deve aggiornare un campo della directory. Questa modifica richiede la lettura nella memoria del blocco, la modifica della sezione e la riscrittura del blocco nel disco, poiché sui dischi si può operare solamente per blocchi (o cluster). Quindi, ogni volta che si apre un file per la lettura, si deve leggere e scrivere anche l'elemento della directory a esso associato. Ciò può essere inefficiente per file cui si accede frequentemente, quindi nella fase della progettazione del file system è necessario confrontare i benefici con i costi rispetto alle prestazioni. In generale, è necessario considerare l'influenza sull'efficienza e sulle prestazioni di ogni informazione che si vuole associare a un file.

Prestazioni

Dopo aver scelto gli algoritmi fondamentali del file system le prestazioni possono essere migliorate in diversi modi. Alcuni controllori di unità a disco contengono una quantità di memoria locale sufficiente per la creazione di una **cache** interna al controllore sufficientemente grande da memorizzare un'intera traccia del disco alla volta. Eseguito il posizionamento della testina, si legge la traccia nella cache del controllore del disco a partire dal settore sotto cui si viene a trovare la testina (riducendo il tempo di latenza). Il controllore trasferisce quindi al sistema operativo tutte le richieste di settori. Quando i blocchi sono trasferiti dal controllore del disco alla memoria centrale, il sistema operativo ha la possibilità di inserirli in una propria cache nella memoria centrale. Alcuni sistemi riservano una sezione separata della memoria centrale come cache del disco, dove tenere i blocchi in previsione di un loro riutilizzo entro breve tempo. Altri sistemi impiegano una cache delle pagine per i file; si tratta di una soluzione che impiega tecniche di memoria virtuale per la gestione dei dati dei file come pagine anziché come blocchi di file system; l'uso degli indirizzi virtuali è molto più efficiente dell'uso dei blocchi fisici di disco.

Alcune versioni di UNIX e Linux prevedono la cosiddetta buffer cache unificata. Per illustrarne i vantaggi si considerino le due possibilità di aprire un file e accedervi: l'uso della mappatura dei file in memoria e l'uso delle ordinarie chiamate di sistema `read()` e `write()`.

Senza una buffer cache unificata, si verifica una situazione simile a quella illustrata nella figura di fianco. In questo caso, le chiamate di sistema `read()` e `write()` passano attraverso la buffer cache. La chiamata con associazione alla memoria richiede l'uso di due cache, la cache delle pagine e la buffer cache. L'associazione alla memoria prevede la lettura dei blocchi di disco dal file system e la loro memorizzazione nella buffer cache. Poiché il sistema di memoria virtuale non può interfacciarsi con la buffer cache, si deve copiare nella cache delle pagine il contenuto del file presente nella buffer cache. Questa situazione è nota come **double caching** proprio perché i dati del file system richiedono un doppio passaggio di cache.

Non solo ciò comporta uno spreco di memoria, ma anche uno spreco di cicli della CPU e di I/O dovuti a un ulteriore trasferimento di dati nella memoria del sistema. Inoltre, eventuali incoerenze tra le due cache possono generare errori nella memorizzazione dei dati nei file.

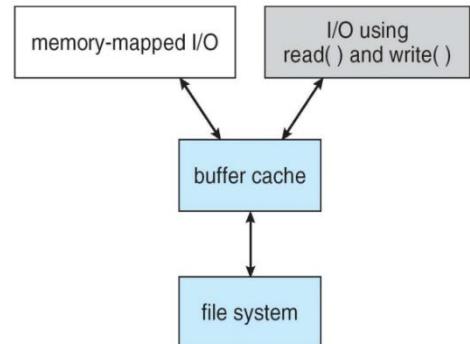
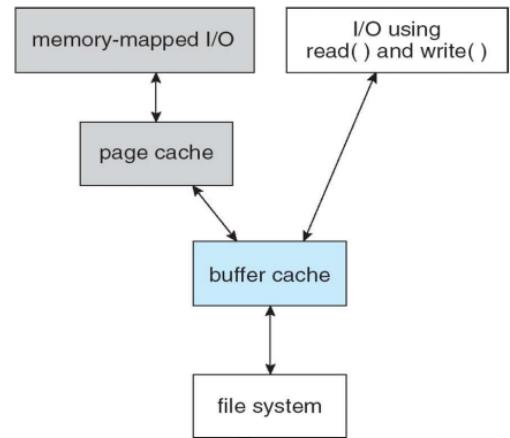
Con una buffer cache unificata, sia l'associazione alla memoria sia le chiamate di sistema `read()` e `write()` usano la stessa cache delle pagine, con il vantaggio di evitare il double caching e di permettere al sistema di memoria virtuale di gestire dati del file system. La figura a destra illustra l'uso della buffer cache unificata.

Indipendentemente dalla gestione delle cache per blocchi di disco oppure per pagine (o per entrambi), l'algoritmo LRU è in generale ragionevole per la sostituzione dei blocchi o delle pagine. Tuttavia, l'evoluzione degli algoritmi di gestione delle cache delle pagine

usati dal sistema operativo Solaris rivela le difficoltà nella scelta di un algoritmo ottimale. Tale sistema operativo permette ai processi e alla cache delle pagine di condividere la memoria inutilizzata; prima della versione 2.5.1, non si facevano distinzioni tra l'allocazione delle pagine a un processo o alla cache delle pagine, con la conseguenza che un sistema che eseguiva molte operazioni di I/O usava la maggior parte della memoria disponibile per la cache delle pagine. A causa dell'alta frequenza delle operazioni di I/O, quando la memoria libera diventa troppo esigua, il modulo di scansione delle pagine sottrae pagine ai processi anziché alla cache delle pagine. In Solaris 2.6 e in Solaris 7 è stata realizzata in forma opzionale la tecnica di paginazione con priorità, secondo la quale il modulo di scansione delle pagine dà la priorità alle pagine dei processi rispetto a quelle della cache delle pagine.

Ci sono altri aspetti che possono influenzare le prestazioni di I/O, come quelli che riguardano la necessità di scritture sincrone o asincrone. Le scritture sincrone avvengono nell'ordine in cui le riceve il sottosistema per la gestione del disco e non subiscono la memorizzazione transitoria. Quindi la procedura chiamante prima di proseguire deve attendere che i dati raggiungano l'unità a disco. Nella maggior parte dei casi si usano scritture asincrone. Nelle scritture asincrone si memorizzano i dati nella cache e si restituisce immediatamente il controllo alla procedura chiamante. Le scritture dei metadati, tra le altre, possono essere sincrone. I sistemi operativi spesso includono un flag nella chiamata di sistema `open()` per permettere a un processo di richiedere che le operazioni di scrittura si eseguano in modo sincrono. I sistemi di gestione delle basi di dati ad esempio usano questa funzione per realizzare le transazioni atomiche, in modo da assicurare che i dati raggiungano la memoria stabile nell'ordine richiesto.

Alcuni sistemi ottimizzano la cache delle pagine adottando, secondo il tipo d'accesso ai file, differenti algoritmi di sostituzione. Le pagine relative a un file da leggere o scrivere in modo sequenziale non si dovrebbero sostituire nell'ordine LRU, infatti la pagina usata più di recente sarà usata nuovamente per ultima, o forse mai. Gli accessi sequenziali si potrebbero invece ottimizzare con tecniche note come rilascio



indietro e lettura anticipata. Il rilascio indietro (free-behind) rimuove una pagina dalla memoria di transito non appena si verifica una richiesta della pagina successiva; le pagine precedenti con tutta probabilità non saranno più usate e quindi sprecano spazio in memoria di transito. Con la lettura anticipata (readahead) 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. Il recupero di questi dati dal disco con un unico trasferimento e la memorizzazione nella cache consentono di risparmiare una quantità di tempo considerevole. La presenza nel controllore di una cache per le tracce non elimina la necessità di adottare la tecnica di lettura anticipata in un sistema multiprogrammato, ciò a causa dell'elevata latenza e del sovraccarico determinato dai tanti piccoli trasferimenti dalla cache per le tracce alla memoria centrale; il ricorso alla lettura anticipata è vantaggioso.

La cache delle pagine, il file system e i driver del disco interagiscono in modi interessanti. Quando i dati vengono scritti su un file del disco, le pagine sono memorizzate nella cache, che qui funge da buffer, mentre il driver del disco ordina la propria coda di dati in uscita in base all'indirizzo sul disco. Queste due azioni consentono al driver del disco di minimizzare gli spostamenti della testina del disco e fanno sì che la scrittura dei dati rispetti i tempi ottimali per la rotazione del disco. A meno che le scritture richieste siano sincrone, un processo, per scrivere sul disco, scrive direttamente nella cache: il sistema trasferisce infatti i dati su disco, in maniera asincrona, quando lo ritiene opportuno. Dal punto di vista del processo utente, le scritture sembreranno estremamente rapide. Durante la lettura, l'I/O a blocchi procede a qualche lettura anticipata; ma, in realtà, le scritture si giovano della modalità asincrona molto più delle letture. Per grandi quantità di dati, quindi, la scrittura su disco tramite il file system è spesso più veloce della lettura, contrariamente a ciò che suggerirebbe l'intuizione.

Ripristino

Poiché i file e le directory sono mantenuti sia in memoria centrale sia nei dischi, è necessario aver cura di assicurare che il verificarsi di un malfunzionamento nel sistema non comporti la perdita di dati o la loro incoerenza, in questo paragrafo vedremo anche come un sistema può essere ripristinato in seguito a un malfunzionamento.

Un crollo del sistema può causare incoerenze tra le strutture dati del file system su disco, come le strutture delle directory, i puntatori ai blocchi liberi e i puntatori agli FCB liberi. Molti file system applicano delle modifiche direttamente a queste strutture. Operazioni comuni come la creazione di un file possono comportare molti cambiamenti strutturali all'interno del file system di un disco. Le strutture delle directory vengono modificate, gli FCB e i blocchi di dati allocati e i contatori liberi per tutti questi blocchi diminuiti. Quando queste modifiche sono interrotte da un crollo del sistema, ne possono derivare incoerenze tra le strutture. Ad esempio, il contatore degli FCB liberi potrebbe indicare che un FCB è stato allocato, ma la struttura della directory potrebbe non avere un puntatore a quel FCB. L'utilizzo della cache che i sistemi operativi adottano per ottimizzare le prestazioni di I/O aggrava questo problema. Alcuni cambiamenti potrebbero andare direttamente sul disco, mentre gli altri possono finire nella cache. Se i cambiamenti nella cache non raggiungono il disco prima che si verifichi un crollo, è possibile che la situazione peggiori ulteriormente. Inoltre, anche i bachi nell'implementazione del file system, i controllori del disco, e persino le applicazioni utente possono indurre errori nel file system. I file system hanno svariati metodi per affrontare queste circostanze, a seconda delle strutture dati e degli algoritmi del file system. Ci occuperemo adesso di questi temi.

Verifica della coerenza

Quale che sia la causa degli errori, un file system deve prima scoprire i problemi e poi correggerli. Per scoprire gli errori vengono esaminati tutti i metadati su ogni file system per verificare la coerenza del sistema. Sfortunatamente, questo procedimento richiederà diversi minuti, o addirittura delle ore, e avverrà tutte le volte che il sistema si avvia. In alternativa, un file system può registrare il suo stato all'interno dei metadati del file system. All'inizio di ogni serie di modifiche dei metadati è impostato un bit di stato per

indicare che i metadati sono in stato di modifica. Se tutti gli aggiornamenti dei metadati si completano con successo, il file system può azzerare quel bit. Se tuttavia il bit dello stato rimane impostato, entra in funzione un verificatore della coerenza.

Il **verificatore della coerenza** confronta i dati delle directory con quelli contenuti nei blocchi dei dischi, tentando di correggere ogni incoerenza. Gli algoritmi di allocazione e di gestione dello spazio libero determinano il genere di problemi che questo programma può riconoscere e con quanto successo riuscirà a risolverli.

File system con log delle modifiche

Spesso nell'informatica si adottano algoritmi e tecnologie anche in aree diverse da quelle per le quali sono stati progettati. E il caso degli algoritmi per il ripristino. Questi algoritmi sono stati applicati con successo al problema della verifica della coerenza, realizzando i log-based transaction-oriented file system, noti anche come **journaling file system**.

Il verificatore della coerenza potrebbe non essere in grado di ripristinare le strutture, con una conseguente perdita di file o addirittura di intere directory; ancora, il verificatore della coerenza potrebbe richiedere l'intervento umano per risolvere i conflitti, il che causa inconvenienti: in mancanza di assistenza da parte di qualcuno, il sistema potrebbe essere inutilizzabile finché una persona non gli indichi come procedere; infine, il verificatore della coerenza sottrae risorse al sistema: per controllare un terabyte di dati possono essere necessarie molte ore.

La soluzione a questo problema consiste nell'applicare agli aggiornamenti dei metadati relativi al file system metodi di ripristino basati sulla registrazione delle modifiche. Sia il file system NTFS sia il Veritas usano questo metodo, che è anche opzionale rispetto al file system UFS nel Solaris 7 e nelle versioni successive. In realtà, sta diventando un metodo comune in molti sistemi operativi.

Fondamentalmente, tutte le modifiche dei metadati si annotano in modo sequenziale in un file di registrazione, detto log. Ogni insieme di operazioni che esegue uno specifico compito si chiama **transazione**. Una volta che le modifiche sono riportate nel file di registrazione, le operazioni si considerano portate a termine con successo (*committed*) e la chiamata di sistema può restituire il controllo al processo utente, permettendogli di proseguire la sua esecuzione. Nel frattempo, si applicano alle effettive strutture del file system le operazioni scritte nel log, e man mano che si eseguono si aggiorna un puntatore che indica quali azioni sono state completate e quali sono ancora incomplete. Quando un'intera transazione è stata completata, se ne rimuovono le annotazioni dal log, che è in realtà un buffer circolare. I **buffer circolari** scrivono fino alla fine dello spazio disponibile, e poi ricominciano dall'inizio, sovrascrivendo i vecchi contenuti. Naturalmente, si devono prendere delle misure per evitare che dati non ancora salvati siano sovrascritti. Il log si potrebbe mantenere in una sezione separata del file system, o anche in un disco separato. È più efficiente, anche se è più complesso, averlo sotto testine di lettura e scrittura separate, poiché si riducono le situazioni di contesa della testina e i tempi di ricerca (*seek time*).

Se si verifica un crollo del sistema, nel log ci potranno essere zero o più transazioni. Le transazioni presenti non sono mai state ultimate nel file system, anche se il sistema operativo le definisce portate a termine con successo, e quindi si devono completare. Le transazioni si possono eseguire a partire dalla posizione corrente del puntatore fino al completamento, e le strutture del file system rimangono coerenti. L'unico problema che si può presentare è il caso in cui una transazione sia fallita (*aborted*), cioè non sia stata dichiarata terminata con successo prima del crollo del sistema. In questo caso, si devono annullare tutti i cambiamenti che erano stati applicati al file system dalla transazione, di nuovo mantenendo la coerenza del file system. Questo ripristino è tutto ciò che è necessario fare dopo un crollo del sistema, eliminando tutti i problemi concernenti la verifica della coerenza.

Un vantaggio indiretto dell'uso dell'annotazione in un disco degli aggiornamenti dei metadati è che gli aggiornamenti sono molto più rapidi di quelli che si applicano direttamente alle strutture dati nei dischi. La ragione di questo miglioramento sta nel vantaggio, dal punto di vista delle prestazioni, dell'I/O ad accesso sequenziale rispetto a quello ad accesso diretto. Le onerose operazioni di scrittura dei metadati ad accesso diretto e sincrono si sostituiscono con molto meno gravose operazioni di scrittura sincrone ma ad accesso sequenziale nell'area di registrazione delle modifiche di un file system con annotazione delle modifiche. I cambiamenti determinati da quelle operazioni si riportano successivamente in modo asincrono nelle strutture appropriate nei dischi attraverso operazioni di scrittura ad accesso diretto. Il risultato complessivo è un significativo guadagno in termini di prestazioni per le operazioni orientate ai metadati, come la creazione e la cancellazione dei file.

Altre soluzioni

Un'altra alternativa alla verifica della coerenza è impiegata dal file system WAFL di Network Appliance e da ZFS di Sun. Entrambi i sistemi non sovrascrivono mai i blocchi con nuovi dati; al contrario, una transazione scrive tutti i cambiamenti di dati e metadati su nuovi blocchi. Quando la transazione viene completata, le strutture di metadati che puntavano alla vecchia versione di questi blocchi sono aggiornate in modo da puntare ai nuovi. Il file system può quindi rimuovere i vecchi puntatori e i vecchi blocchi per renderli nuovamente disponibili. Se i vecchi puntatori e i vecchi blocchi vengono mantenuti, viene creata una snapshot (istantanea), cioè un'immagine del file system prima dell'ultimo aggiornamento. Questa soluzione non dovrebbe richiedere una verifica della coerenza se l'aggiornamento del puntatore è eseguito automaticamente. Ciononostante, il WAFL possiede comunque un controllore della coerenza, perché alcuni tipi di malfunzionamento possono ancora causare un errore nei metadati.

In merito alla coerenza del disco lo ZFS di Sun presenta un approccio ancora più innovativo. ZFS non sovrascrive mai i blocchi, come WAFL, ma è in grado di andare oltre fornendo un riassunto delle verifiche su tutti i metadati e i blocchi di dati. Questa soluzione (se combinata con RAID) assicura che i dati siano sempre corretti. ZFS non possiede quindi un controllore della coerenza.

Copie di riserva e recupero dei dati

Poiché si possono verificare malfunzionamenti e perdite di dati anche nei dischi magnetici, è necessario preoccuparsene e provvedere affinché 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 in altri dispositivi di registrazione di dati, come dischetti, nastri magnetici, dischi ottici 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 riserva.

Ai fine di ridurre al minimo la quantità di dati da copiare, è possibile sfruttare le informazioni contenute nell'elemento della directory associato a ogni file. Ad esempio, se il programma di creazione delle copie di riserva sa quando è stata eseguita l'ultima copia di riserva di un file, e se la data di ultima scrittura di quel file, registrata nella directory, indica che il file da quel momento non ha subito variazioni, non sarà necessario copiare nuovamente il file. Quella che segue è una tipica sequenza di gestione delle copie di riserva.

- Giorno 1. Copiatura nel supporto di backup delle copie di riserva di tutti i file contenuti nel disco; detta **copiatura completa**.
 - Giorno 2. Copiatura su un altro supporto di tutti i file modificati dal Giorno 1; si tratta di una **copiatura incrementale**.
 - Giorno 3. Copiatura su un altro supporto di tutti i file modificati dal Giorno 2.
 - ...
 - Giorno **n**. Copiatura su un altro supporto di tutti i file modificati dal Giorno $n - 1$.
- Ritorno al Giorno 1.

Il nuovo ciclo può comportare la scrittura delle nuove copie di riserva nel primo insieme di supporti di backup, oppure in un nuovo insieme; in questo modo si ha la possibilità di recuperare il contenuto dell'intero disco iniziando le operazioni di recupero dalla copia di riserva completa e proseguendo con le copie di riserva incrementalì. Naturalmente, più grande è n, maggiore sarà il numero di nastri o dischi da leggere per un completo recupero. Un ulteriore vantaggio di questo ciclo di creazione di copie di riserva è la possibilità di recuperare qualsiasi file accidentalmente cancellato durante il ciclo, recuperandolo dalle copie del giorno precedente. La lunghezza del ciclo è un compromesso tra la quantità di supporti per le copie di riserva necessari e il numero di giorni addietro da cui si può compiere un'operazione di recupero. La lunghezza del ciclo è un compromesso fra la quantità di spazio richiesta dalle copie di riserva e il numero di giorni addietro da cui si può compiere un'operazione di recupero. Una possibilità per diminuire la quantità di nastri che è necessario leggere per portare a termine il ripristino è di eseguire inizialmente una copia di riserva completa, per poi eseguire copie dei soli file modificati nei giorni successivi. In questo modo, il ripristino può fondarsi sull'ultima copia incrementale, insieme all'ultima copia completa, senza necessità di altre copie incrementalì. Qui il compromesso da tenere a mente è che il numero di file modificati aumenta giornalmente: ogni nuova copia incrementale, quindi, richiede più spazio.

Un utente potrebbe accorgersi dopo molto tempo che si è perso o si è danneggiato un particolare file. Per questa ragione si è soliti pianificare di tanto in tanto una copiatura completa che sarà conservata in modo permanente; quindi il supporto che la contiene non sarà riutilizzato. E inoltre opportuno conservare tali copie di riserva permanenti lontano dalle copie ordinarie per proteggerle dai vari pericoli, ad esempio un incendio che può distruggere il calcolatore e tutte le copie di riserva. Se il ciclo di creazione delle copie di riserva prevede il reimpiego dei mezzi che le contengono, è anche necessario aver cura di non usarli troppe volte: se dovessero logorarsi, potrebbe essere impossibile ripristinare i dati in essi contenuti.

12. Memoria di Massa

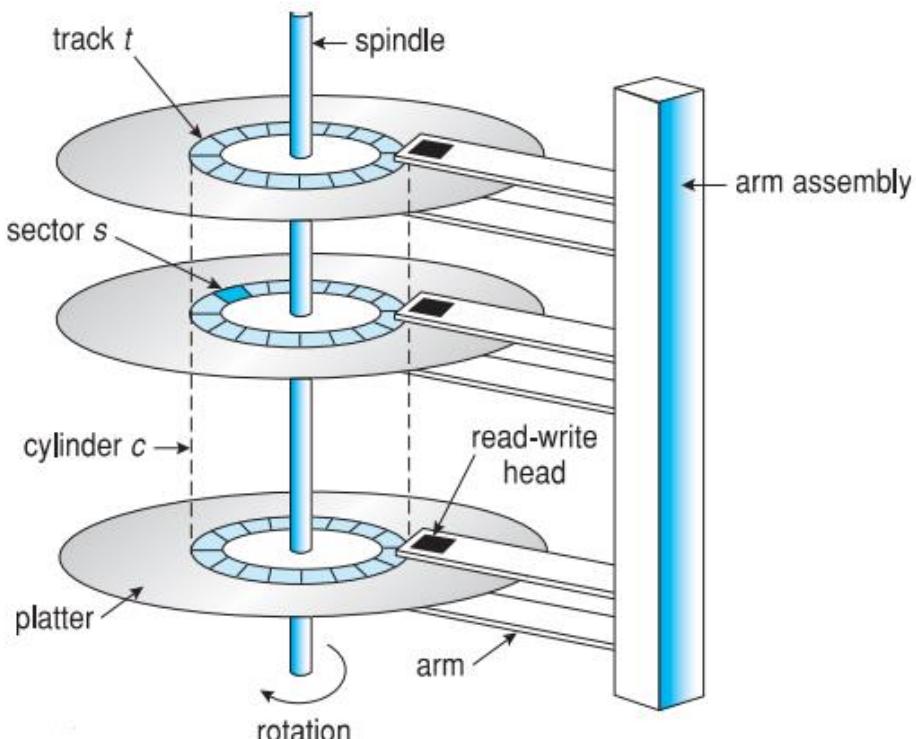
Struttura dei dispositivi di memorizzazione

In questo paragrafo si introduce una presentazione generale della struttura fisica dei dispositivi di memorizzazione secondaria e terziaria.

Hard Disk

I dischi magnetici sono il mezzo fondamentale di memoria secondaria dei moderni sistemi di calcolo.

Concettualmente, i dischi (vedi figura) 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 5,25 pollici, e le due superfici ricoperte di materiale magnetico; le informazioni si memorizzano registrandole magneticamente sui piatti.



Le testine di lettura e scrittura sono sospese su ciascuna superficie d'ogni piatto e 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 (equidistanti dal centro dei piatti) costituisce un cilindro. In un'unità a disco possono esservi migliaia di cilindri concentrici e ogni traccia può contenere centinaia di settori. La capacità di memorizzazione di una comune unità a disco è dell'ordine delle decine di gigabyte.

Quando un disco è in funzione, un motore lo fa ruotare ad alta velocità; la maggior parte dei dischi ruota a velocità comprese tra 60 e 200 giri al secondo. L'efficienza di un disco è caratterizzata da due valori: la **velocità di trasferimento** (*transfer rate*), cioè la velocità con cui i dati fluiscono dall'unità a disco al calcolatore, e il **tempo di posizionamento**, talvolta detto tempo d'accesso diretto, che consiste nel 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**. In genere i dischi possono trasferire parecchi megabyte di dati al secondo e hanno un tempo di ricerca e una latenza di rotazione di diversi millisecondi.

Poiché le testine di un disco sono sospese su un cuscino d'aria sottilissimo (dell'ordine dei micron), esiste il pericolo che la testina urti la superficie del disco; in tal caso, nonostante i piatti del disco siano ricoperti da

un sottile strato protettivo, la testina può danneggiare la superficie magnetica. Tale incidente, detto **crollo della testina**, di solito non può essere riparato e comporta la sostituzione dell'intera unità a disco.

Un hard disk è connesso a un calcolatore attraverso un insieme di fili detto **bus di I/O**; esistono diversi tipi di tale bus, tra i quali i bus EIDE (*enhanced integrated drive electronics*), ATA (*advanced technology attachment*), ATA seriale (*serial ATA*), USB (*universal serial bus*), FC (*fiber channel*) e SCSI. Il trasferimento dei dati in un bus è eseguito da speciali unità di elaborazione dette **controllori**: gli **adattatori** (*adapter*) o **controllori di macchina** (*host controller*) sono i controllori posti all'estremità relativa al calcolatore del bus; i **controllori dei dischi** (*disk controller*) sono incorporati in ciascuna unità a disco.

Hard disk Performance

- Access Latency = Average access time = average seek time + average latency
 - Per dischi più veloci $3\text{ms} + 2\text{ms} = 5\text{ms}$
 - Per dischi lenti $9\text{ms} + 5.56\text{ms} = 14.56\text{ms}$
- Average I/O time = average access time + (quantità da trasferire / velocità di trasferimento) + overhead del controllore
- Per esempio, per trasferire un blocco di 4KB su un disco da 7200 RPM con un average seek time di 5ms, 1Gb/sec di transfer rate con un .1ms overhead del controllore =
 - $5\text{ms} + 4.17\text{ms} + 0.1\text{ms} + \text{transfer time} =$
 - Transfer time = $4\text{KB} / 1\text{Gb/s} = 0.031\text{ ms}$
 - Average I/O time for 4KB block = $9.27\text{ms} + .031\text{ms} = 9.301\text{ms}$

Dispositivi NVM

I dispositivi **NVM** sempre più rilevanti e diffusi. Sono elettrici e non meccanici. Tipicamente composti basati su memoria flash (chip NAND flash) vengono spesso inseriti in contenitori simili a unità disco, e per questa ragione sono chiamati **dischi a stato solido**, o SSD (Solid-State Disk). Un dispositivo NVM può anche assumere la forma di un'**unità USB**. In tutte le sue forme possiamo trattarlo in modo uniforme.



SSD:

- Memoria non volatile usata come un hard drive
 - Non meccanica
 - Implementata con diverse tecnologie
- Può essere più affidabile degli HDD
- Più costosa per MB
- Forse life span più breve
- In generale ha meno capacità di HDD ma più veloce
- Capacità cresce rapidamente ed il prezzo cala, quindi si stanno affermando
- I bus di connessione possono essere troppo lenti
 - connessi direttamente al bus di sistema
- Non parti mobili, non parti meccaniche
 - non seek time e rotational latency

Algoritmi del controllore delle NAND Flash

- Implementate con semiconduttori NAND (transistor MOSFET) che portano nuove sfide a capacità di memorizzazione e affidabilità
- I **semiconduttori NAND** non possono essere sovrascritti direttamente, devono prima essere cancellati. Organizzati in pagine e presentano pagine che contengono dati non validi

- La cancellazione avviene in blocco e prende tempo ($>$ del tempo di scrittura $>$ tempo lettura). Le operazioni possono avvenire in parallelo. Deterioramento dopo ogni cancellazione. Durata misurata in Drive Writes per Day: quanto volte può essere scritta per giorno prima del fallimento (es. 1 TB può avere 5 DWPD nel periodo garantito)
- Algoritmi gestiti dal controller, non dal sistema operativo.
- I controller contiene una tabella: Flash Transalition Layer (FTL) indica quali pagine contengono dati validi
- Per gestire i dati occorrono: algoritmi di garbage collection per liberare blocchi da pagine valide, pagine (circa 20%) sempre disponibili per fare il write (overprovisioning), meccanismi per distribuire le cancellazioni, meccanismi di correzione del dato (se errori continui su una pagina, questa è marcata come bad)

pagina valida	pagina valida	pagina non valida	pagina non valida
pagina non valida	pagina valida	pagina non valida	pagina valida

Nastri magnetici

I nastri magnetici sono stati i primi supporti di memorizzazione secondaria. Pur avendo la capacità di memorizzare in modo permanente un'enorme quantità di dati, queste unità sono caratterizzate da un tempo d'accesso molto elevato rispetto a quello della memoria centrale e dei dischi magnetici. Inoltre il tempo d'accesso diretto dei nastri magnetici (essendo fisicamente ad accesso sequenziale) è migliaia di volte maggiore di quello dei dischi magnetici, e ciò li rende inadatti come supporto di memoria secondaria. Gli usi principali dei nastri sono la creazione di copie di riserva dei dati (backup), la registrazione di dati poco usati e il trasferimento di informazioni tra diversi sistemi di calcolo.

Il nastro è avvolto in bobine e scorre su una testina di lettura e scrittura. Il posizionamento sul settore richiesto può richiedere alcuni minuti, anche se, una volta raggiunta la posizione desiderata, l'unità a nastro può leggere o scrivere informazioni a una velocità paragonabile a quella di un'unità a disco. La capacità varia secondo il particolare tipo di unità a nastro.

Struttura dei dischi e mapping degli indirizzi

I moderni dischi sono considerati come grandi array monodimensionali di blocchi logici, dove un blocco logico è la minima unità di trasferimento. La dimensione di un blocco logico è di solito di 512 byte, sebbene alcuni dischi si possano formattare a basso livello allo scopo di ottenere una diversa dimensione dei blocchi logici su mezzi fisici.

Su hard disk i blocchi logici vengono mappati sequenzialmente in settori; per NVM mapping, invece, si passa da una tupla (*chip, blocco, pagina*) ad un blocco logico.

L'array monodimensionale di blocchi logici corrisponde in modo sequenziale ai settori del disco: il settore 0 è il primo settore della prima traccia sul cilindro più 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.

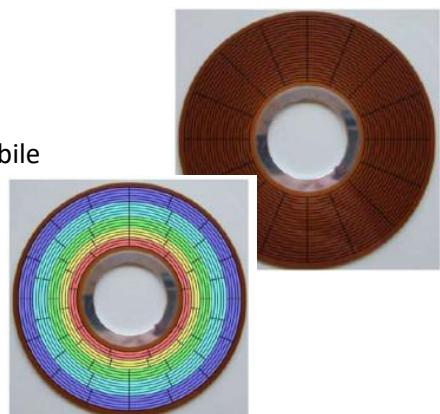
Sfruttando questa corrispondenza sarebbe possibile (almeno in teoria) trasformare il numero di un blocco logico in un indirizzo fisico di vecchio tipo, consistente in un numero di cilindro, un numero di traccia concernente quel cilindro, e un numero di settore relativo a quella traccia. In pratica, però, 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** (CLV, *constant linear velocity*) 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. Spostandosi da aree esterne verso aree più interne il numero di settori per traccia diminuisce. Le tracce nell'area più esterna contengono in genere il 40 per cento in più dei settori contenuti nelle tracce dell'area più interna. L'unità aumenta la sua velocità di rotazione man mano che le testine si spostano dalle tracce esterne verso le tracce più interne per mantenere costante la quantità di dati che scorrono sotto le testine. 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** (CAV, *constant angular velocity*).

Il numero di settori per traccia cresce in conseguenza alla tecnologia dei dischi, e l'area più esterna di un disco di solito contiene centinaia di settori per traccia. Anche il numero di cilindri è andato aumentando; le unità a disco contengono decine di migliaia di cilindri.

Ricapitolando ci sono due tipi di organizzazione del disco:

- Stesso numero di settori per traccia
 - Ampiezza variabile del settore e densità scrittura variabile
 - Velocità lettura costante per traccia
 - Tracce esterne poco utilizzate
- Numero di settori differente per traccia (zone bit recording)
 - Densità scrittura costante
 - Velocità lettura settori variabile
 - Ottimizzazione (più dati nei settori esterni)



Connessione dei dischi

I calcolatori accedono alla memoria secondaria in tre modi: nei sistemi di piccole dimensioni il modo più comune è tramite le porte di I/O (**memoria secondaria connessa alla macchina**); oppure ciò avviene in modo remoto per mezzo di un file system distribuito (**memoria secondaria connessa alla rete**) o ancora con un **dispositivo cloud**.

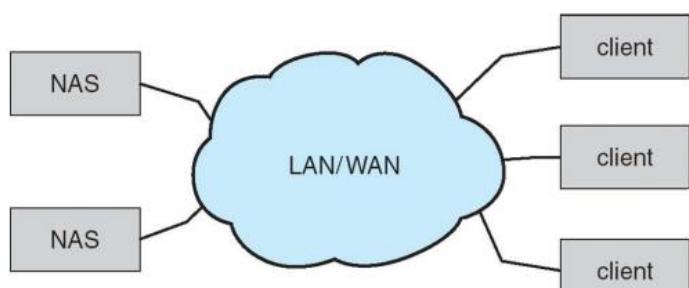
Direct Access Storage

Alla memoria secondaria connessa alla macchina (DAS) si accede dalle porte locali di I/O. Queste porte sono disponibili in diverse tecnologie; i comuni PC impiegano un'architettura per il bus di I/O chiamata IDE o ATA. Quest'architettura consente di avere non più di due unità per ciascun bus di I/O. Le stazioni di lavoro di fascia alta e i server impiegano architetture più raffinate come SCSI e FC (fibre channel).

C'è un gran numero di dispositivi utilizzabili come memoria secondaria connessa alla macchina, tra questi le unità a disco, le batterie RAID, le unità a CD, DVD e a nastri magnetici.

Network-Attached Storage

Un dispositivo di memoria secondaria connessa alla rete è un sistema di memoria speciale al quale si accede in modo remoto per mezzo di una rete di trasmissione di dati (vedi figura). I client accedono alla memoria connessa alla rete (NAS) tramite un'interfaccia RPC, come l'NFS nel caso dei sistemi UNIX o come CIFS nel caso di sistemi Windows. Le chiamate di procedura remota (RPC) sono realizzate per mezzo dei protocolli TCP o UDP sopra una rete IP (di solito la stessa rete locale che porta tutto il traffico di dati ai



client). La memoria secondaria connessa alla rete è normalmente realizzata come una batteria RAID con programmi di controllo che implementano l'interfaccia per le RPC.

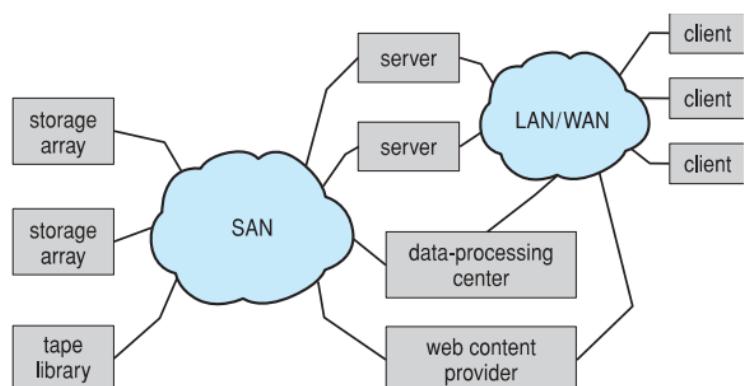
Cloud Storage

Come NAS permette l'accesso tramite rete. Un dispositivo Cloud non accede tramite file system bensì tramite Internet ad un centro remoto di storage. L'accesso tramite API è più robusto rispetto a perdita di connettività. NAS accede assumendo LAN affidabile con protocolli come NFS e CIFS, se la connessione LAN fallisce possono rimanere in hang.

Storage Area Network

Uno svantaggio dei sistemi di memoria secondaria connessa alla rete è che le operazioni di I/O sulla memoria secondaria impegnano banda della rete e quindi aumentano la latenza della comunicazione nella rete. Questo problema può essere particolarmente grave per sistemi client-server di grandi dimensioni: l'ordinaria comunicazione tra i server e i client compete per la banda con la comunicazione tra i server e i dispositivi di memorizzazione.

Una **rete di memoria secondaria (SAN)** è una rete privata (che impiega protocolli specifici per la memorizzazione anziché protocolli di rete) tra i server e le unità di memoria secondaria, separata dalla LAN o WAN che collega i server ai client (vedi figura). La potenza di una SAN sta nella sua flessibilità: si possono connettere alla stessa SAN molte macchine e molte batterie di memoria; la memoria può essere allocata alle macchine



dinamicamente. Uno switch SAN nega o concede alle macchine l'accesso alla memoria secondaria. Questi switch rendono anche possibile la condivisione della memoria di massa da parte di gruppi di server, e il collegamento diretto di più macchine alla memoria di massa.

Un'alternativa emergente è un'architettura specifica per SAN, chiamata InfiniBand, che sono più veloci di NAS, ma hanno meno dispositivi collegati.

Scheduling del disco

Una delle responsabilità del sistema operativo è quella di fare un uso efficiente delle risorse fisiche: nel caso delle unità a disco, far fronte a questa responsabilità significa garantire tempi d'accesso contenuti e ampiezze di banda elevate. Il tempo d'accesso si può scindere in due componenti: il **seek time** (tempo di ricerca), cioè il tempo necessario affinché il braccio dell'unità a disco sposti le testine fino al cilindro contenente il settore desiderato, e la **rotational latency** (latenza di rotazione), e cioè il tempo aggiuntivo necessario perché il disco ruoti finché il settore desiderato si trovi sotto la testina. Il **bandwidth** (ampiezza di banda) è il numero totale di byte trasferiti diviso il tempo totale intercorso fra la prima richiesta e il completamento dell'ultimo trasferimento. Per mezzo della gestione dell'ordine delle richieste di I/O relative al disco si possono migliorare sia il tempo d'accesso sia l'ampiezza di banda.

Ogni volta che devono compiere operazioni di I/O con un'unità a disco, un processo impedisce al sistema operativo una chiamata di sistema. La richiesta contiene diverse informazioni:

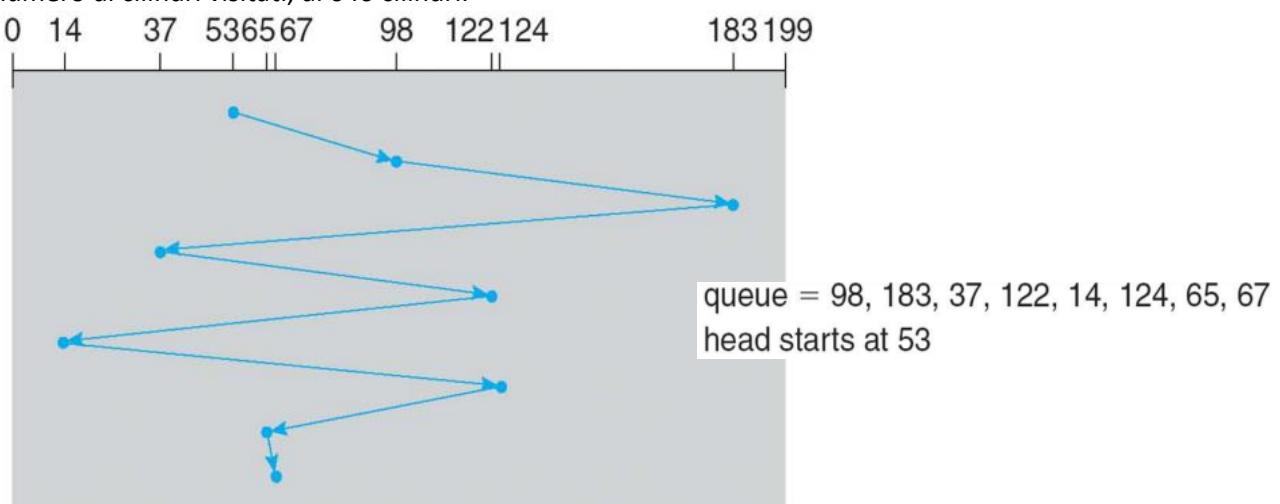
- se l'operazione sia di immissione o di emissione di dati;
- l'indirizzo nel disco rispetto al quale eseguire il trasferimento;
- l'indirizzo di memoria rispetto al quale eseguire il trasferimento;
- il numero di byte da trasferire.

Se l'unità a disco desiderata e il controllore sono disponibili, la richiesta si può immediatamente soddisfare; altrimenti le nuove richieste si aggiungono alla coda di richieste inevase relativa a quell'unità. La coda relativa a un'unità a disco in un sistema con multiprogrammazione può spesso essere piuttosto lunga, sicché il sistema operativo sceglie quale fra le richieste inevase conviene servire prima. Come il sistema operativo affronta tale scelta? Per mezzo di uno dei tanti algoritmi di scheduling che andiamo a presentare di seguito.

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, ad esempio, una coda di richieste per l'unità a disco che dia una lista di cilindri sui quali individuare i blocchi richiesti, nell'ordine seguente:

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 una distanza totale, misurata in numero di cilindri visitati, di 640 cilindri.

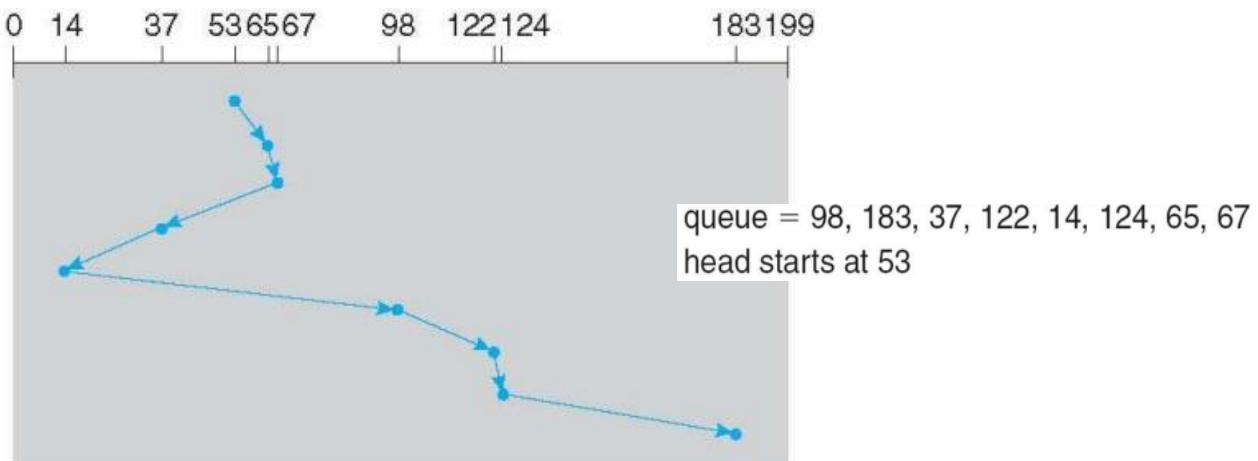


Le defezioni di quest'algoritmo sono palese dal gigantesco salto da 122 a 14 e poi di nuovo a 124: se le richieste per i cilindri 37 e 14 si potessero soddisfare in sequenza, la distanza totale percorsa diminuirebbe notevolmente e le prestazioni migliorerebbero di conseguenza.

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), che sceglie la richiesta che dà il minimo tempo di ricerca rispetto alla posizione corrente della testina; poiché questo tempo aumenta al crescere della distanza dei cilindri dalla testina, l'algoritmo sceglie le richieste relative ai cilindri più vicini alla posizione della testina.

Se si considera nuovamente la sequenza di richieste presa ad 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 (vedi figura successiva). Questo metodo di scheduling implica una distanza totale percorsa di soli 236 cilindri, poco più di un terzo della distanza ottenuta con lo scheduling FCFS di questa coda di richieste: esso porta quindi sostanziali miglioramenti d'efficienza.



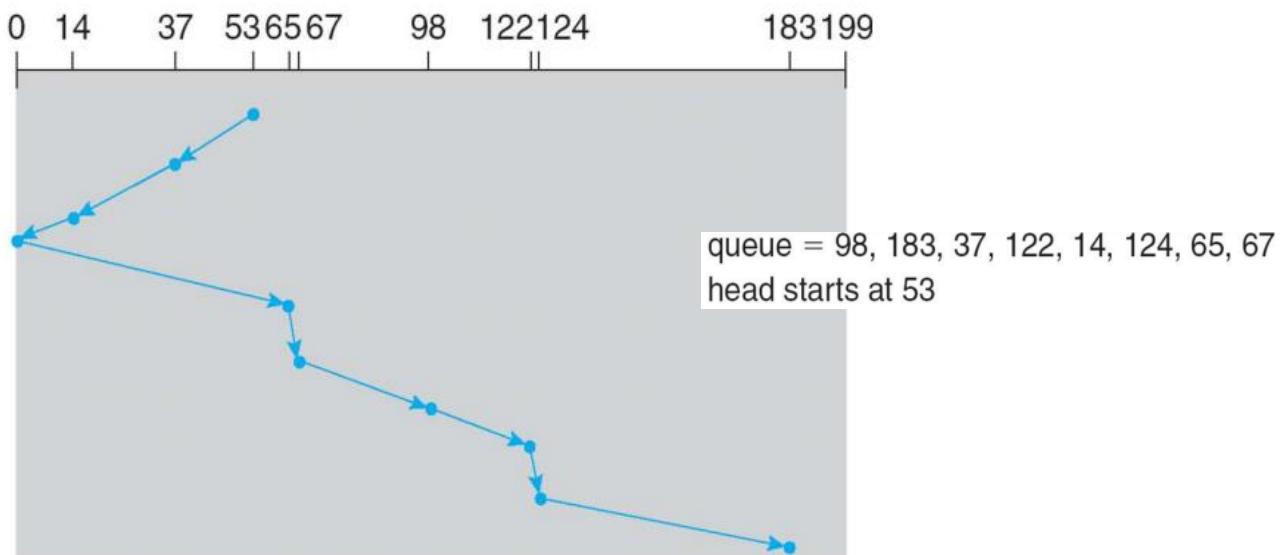
Lo scheduling SSTF è essenzialmente una forma di scheduling per brevità (shortest job first, SJF), e al pari di questo, può condurre a situazioni di starvation di alcune richieste. Si ricordi infatti che nuove richieste possono giungere in qualunque momento: si supponga di avere due richieste in coda, 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 le une vicine alle altre potrebbe causare l'attesa indefinita della richiesta relativa al cilindro 186. Quest'ipotesi diviene sempre più probabile man mano che la coda di richieste si allunga.

Sebbene l'algoritmo SSTF costituisca un miglioramento notevole rispetto al FCFS, esso non è ottimale: si può fare di meglio con uno spostamento dal cilindro 53 al 37, anche se questa non è la minima distanza possibile, e poi al 14, prima di invertire la marcia per servire i 65, 67, 98, 122, 124 e 183. L'adozione di questa strategia riduce la distanza totale percorsa a 208 cilindri.

SCAN

Secondo l'**algoritmo SCAN** (scheduling per scansione) il braccio dell'unità a disco parte da un estremo del disco e si sposta nella sola direzione possibile, servendo le richieste mentre attraversa i cilindri, finché non giunga all'altro estremo del disco: a questo punto, il braccio inverte la marcia, e la procedura continua. Le testine attraversano continuamente il disco nelle due direzioni. L'algoritmo SCAN è a volte chiamato algoritmo dell'ascensore, perché il braccio dell'unità a disco si comporta proprio come un ascensore che serva prima tutte le richieste in salita e poi tutte quelle in discesa.

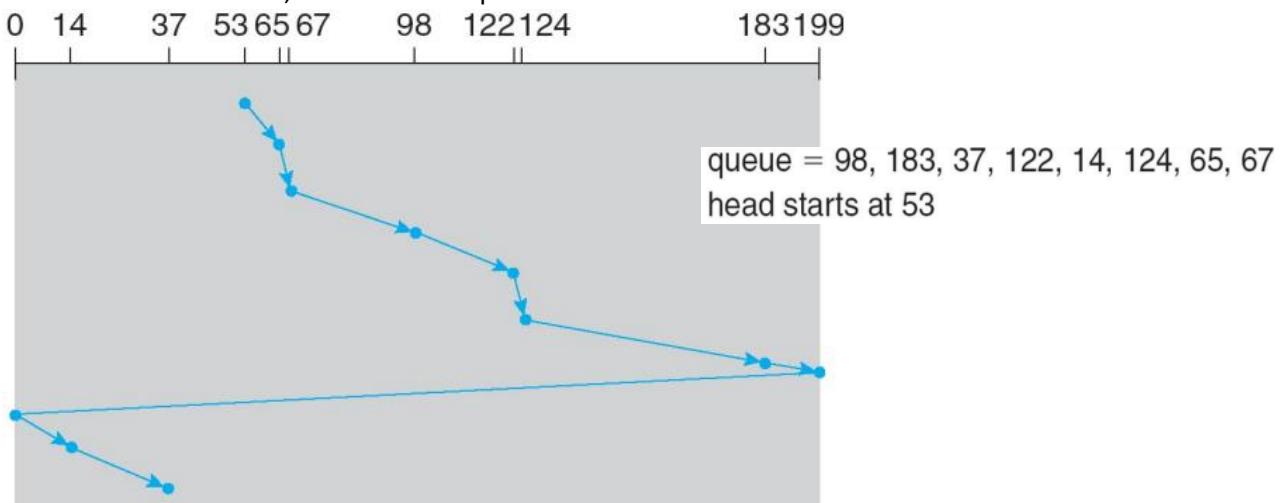
Si consideri ancora l'esempio precedente, prima di poter applicare lo scheduling SCAN alle richieste per i cilindri 98, 183, 37, 122, 14, 124, 65, 67, oltre la posizione corrente (53), occorre conoscere la direzione del movimento delle testine. Se lo spostamento è nella direzione del cilindro 0, l'unità 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 (vedi figura a pagina successiva). Se arriva una nuova richiesta riferita a uno dei cilindri posti davanti alla testina (relativamente alla sua direzione di moto), essa sarà quasi immediatamente soddisfatta; ma se la richiesta è riferita a uno dei cilindri appena sorpassati, essa dovrà attendere fino a che la testina non giunga alla fine del disco, inverta la direzione del moto, e torni indietro.



Assumendo una distribuzione uniforme delle richieste per i cilindri da visitare, si consideri la densità di richieste quando il braccio giunge a un estremo e inverte la direzione del moto: in quel momento, relativamente poche richieste sono riferite a cilindri posti vicino alla testina e davanti a essa, perché i cilindri in questione sono stati recentemente visitati. La massima densità di richieste si riferisce all'altro estremo del disco, e queste richieste sono anche quelle che hanno atteso più a lungo: sembra ragionevole spostarsi lì come prossima mossa. Questa è l'idea dell'algoritmo seguente.

C-SCAN

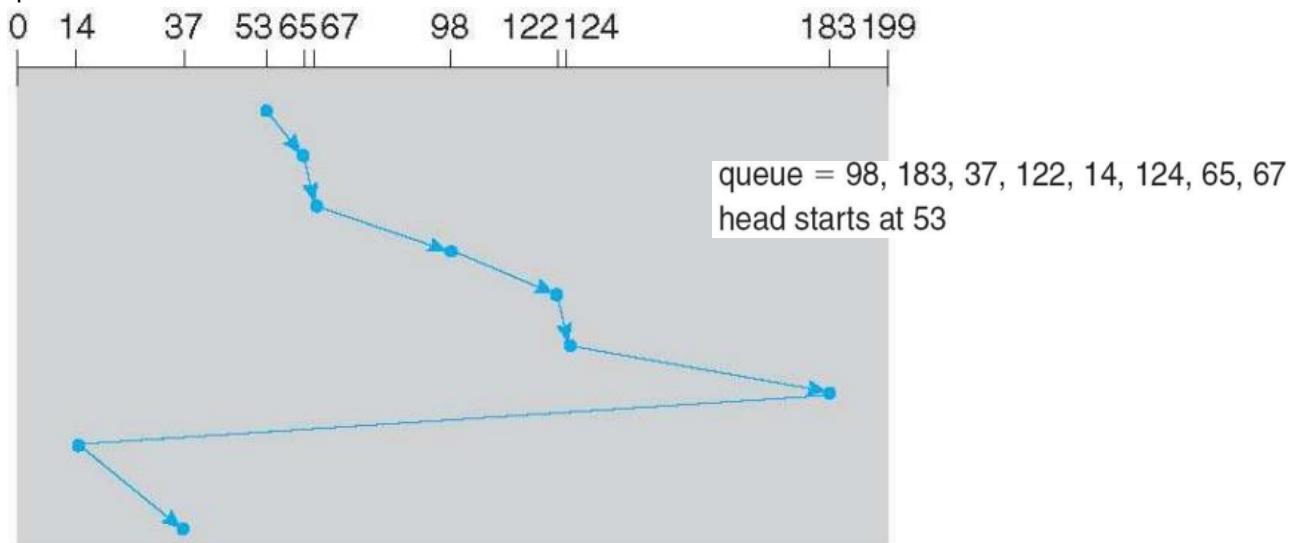
L'algoritmo SCAN circolare (*circular SCAN*, C-SCAN) è una variante dello scheduling SCAN concepita per garantire un tempo d'attesa meno variabile. Anche l'algoritmo C-SCAN, come lo SCAN, sposta la testina da un estremo all'altro del disco, servendo le richieste lungo il percorso; tuttavia, quando la testina giunge all'altro estremo del disco, ritorna immediatamente all'inizio del disco stesso, senza servire richieste durante il viaggio di ritorno (vedi figura). 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.



C-LOOK

Secondo la descrizione appena fatta, sia l'algoritmo SCAN sia il C-SCAN spostano il braccio dell'unità attraverso tutta l'ampiezza del disco; all'atto pratico, nessuno dei due algoritmi è codificato 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.



N.B.: date una serie di indirizzi logici e dato il movimento della testina quanti cilindri attraversa (possibile domanda scritta), e all'orale approfondire tale discussione con la differenza tra indirizzi logici e fisici etc...

Selezionare un algoritmo di Disk-Scheduling

Giacché esistono tanti diversi algoritmi di scheduling, ci si può chiedere come si faccia a 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 una data ma arbitraria lista di richieste si può definire un ordine ottimo di servizio, ma la computazione richiesta può non essere giustificata dal miglioramento in prestazioni rispetto agli algoritmi SSTF o SCAN.

Per qualunque algoritmo di scheduling, le prestazioni dipendono comunque in larga misura dal numero e dal tipo di richieste. Ad esempio, si supponga che la coda sia costituita in genere di una sola richiesta inesiva: tutti gli algoritmi dànno allora luogo allo stesso comportamento, perché hanno una sola scelta possibile relativamente al prossimo 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'assegnazione 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.

Anche la posizione delle directory e dei blocchi indice è importante: poiché ogni file deve essere aperto per essere usato, e visto che l'apertura di un file richiede una ricerca attraverso la struttura delle directory, vi saranno frequenti accessi alle directory. Si supponga che un elemento di directory risieda nel primo cilindro e che i dati del file relativo si trovino nell'ultimo cilindro; la testina dovrà allora percorrere l'intera ampiezza del disco nel caso di apertura del file in questione. Se l'elemento della directory che rappresenta logicamente il file fosse nel cilindro di mezzo, la testina dovrebbe spostarsi al più della metà dell'ampiezza. Anche l'uso della memoria centrale come cache delle directory e dei blocchi indice può contribuire a ridurre i movimenti del braccio dell'unità a disco, in particolare quando si tratta di operazioni di lettura.

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.

Gli algoritmi di scheduling descritti tengono conto solamente del tempo di ricerca, mentre nelle moderne unità a disco la latenza di rotazione può essere lunga quasi quanto il tempo medio di ricerca. Tuttavia è difficile per il sistema operativo adottare una strategia di scheduling che porti a miglioramenti dei tempi di latenza di rotazione, perché le moderne unità a disco non rivelano la posizione fisica dei blocchi logici. I produttori di unità a disco hanno collaborato alla limitazione di questo problema incorporando algoritmi di scheduling all'interno dei controllori contenuti nelle unità a disco: se il sistema operativo invia un gruppo di richieste al controllore, esso può organizzarle in una coda e poi applicare algoritmi di scheduling che riducano sia i tempi di ricerca sia la latenza di rotazione.

Se l'unico elemento di cui tener conto fossero le prestazioni dell'I/O, il sistema operativo scaricherebbe volentieri la responsabilità dello scheduling per il disco sull'apparato elettronico del dispositivo. In pratica, però, il sistema operativo può dover considerare altri vincoli relativi all'ordine in cui si devono servire le richieste: ad esempio, la richiesta di una pagina di memoria virtuale potrebbe avere maggiore priorità rispetto all'I/O delle applicazioni, e le scritture divengono più urgenti delle letture quando la cache sta per esaurire le pagine disponibili. Inoltre, può essere auspicabile mantenere l'ordine naturale delle richieste di scrittura al fine di rendere il file system robusto rispetto ai crolli del sistema: si consideri cosa accadrebbe se il sistema operativoassegnasse un blocco di disco a un file, e un'applicazione scrivesse dati in quel blocco; se il sistema crollasse a questo punto, il sistema operativo potrebbe non essere riuscito a copiare l'*inode* modificato e la nuova lista dello spazio libero sul disco. Per conciliare queste esigenze, il sistema operativo può scegliere di accollarsi la responsabilità dello scheduling del disco e, per alcuni tipi di I/O, fornire le richieste al controllore una alla volta.

NVM scheduling

Lo scheduling HDD deve minimizzare i movimenti meccanici ma nel caso di NVM si usa FCFS poiché non ci sono movimenti meccanici.

Le richieste read sono servite in modo uniforme, ma write in modo non uniforme. Alcuni SO servono le read con FCFS, e accorpando solo richieste write.

I/O possono avvenire in modo random o sequenziale.

- Per HDD meglio sequenziale
- Per NVM random è ok
- input/output operations per second (IOPS) diversi per HDD e NVM
 - Nel caso di accesso random centinaia vs centinaia di migliaia
 - Nel caso di accesso sequenziale più simile
 - Le performance di NVM degradano nel tempo e la scrittura è più lenta della lettura

Gestione dell'unità a disco

Il sistema operativo è anche responsabile di molti altri aspetti della gestione delle unità a disco. In questo paragrafo si discutono l'inizializzazione del disco, l'avviamento del sistema basato sull'unità a disco, e la gestione dei blocchi difettosi.

Formattazione del disco

Un disco magnetico nuovo è tabula rasa: un insieme di uno o più piatti sovrapposti ricoperti di materiale magnetico; prima che possa memorizzare dati, deve essere diviso in settori che possano essere letti o scritti dal controllore. Questo processo si chiama formattazione di basso livello, o formattazione fisica. La formattazione di basso livello riempie il disco con una speciale struttura dati per ogni settore, tipicamente consistente di un'intestazione, un'area per i dati (di solito di 512 byte), e una queue. L'intestazione e la coda contengono informazioni usate dal controllore del disco, ad esempio il numero del settore e un **error correction code** (ECC, codice per la correzione di errori). Quando il controllore scrive dati in un settore nel corso di un'ordinaria operazione di I/O, aggiorna il valore dell'ECC secondo il contenuto dell'area di dati del settore. Quando il controllore legge dati da quel settore, calcola anche l'ECC e lo confronta con il suo valore

memorizzato: se risulta una discrepanza, l'area dei dati del settore non è integra, e il settore del disco potrebbe essere difettoso. L'ECC è un codice per la correzione degli errori: se solo alcuni bit di dati sono stati alterati, esso contiene sufficienti informazioni affinché il controllore possa identificare i bit in questione e ricalcolare il loro corretto valore. Il controllore esegue automaticamente l'elaborazione descritta ogni volta che accede a un settore del disco.

La formattazione fisica dei dischi è eseguita nella maggior parte dei casi dal costruttore come parte del processo produttivo; ciò permette al costruttore di provare il disco, e di instaurare la corrispondenza fra blocchi logici e settori correttamente funzionanti del disco. In molte unità a disco, quando si richiede al controllore di formattare fisicamente il disco, si può anche specificare il numero di byte delle aree di dati comprese fra l'intestazione e la coda di un settore. La scelta è di solito ristretta a poche opzioni, come 256, 512 o 1024 byte. La formattazione in settori più grandi implica la presenza di meno settori su ogni traccia, ma anche meno intestazioni e code, e quindi maggior spazio per i dati veri e propri. Alcuni sistemi operativi gestiscono solo settori di 512 byte.

Per usare un disco come contenitore d'informazioni, il sistema operativo deve ancora registrare le proprie strutture dati nel disco, cosa che fa in due passi. Il primo consiste nel suddividere il disco in uno o più gruppi di cilindri, detti **partizioni**. Il sistema operativo può trattare ogni gruppo come se fosse un'unità a disco a sé stante: ad esempio, una partizione può contenere una copia del codice eseguibile del sistema operativo, mentre un'altra contiene i file degli utenti. Il passo successivo alla suddivisione in partizioni è la **formattazione logica**, cioè la creazione di un file system: il sistema operativo registra nel disco le strutture dati iniziali relative al file system. Le strutture dati in questione possono includere descrizioni dello spazio libero e dello spazio allocato (FAT o inode) e una directory iniziale vuota

Per una maggior efficienza, molti file system accorpano i blocchi in gruppi, detti cluster. L'I/O del disco procede per blocchi, ma l'I/O del file system procede invece per cluster, di modo che l'I/O mutui sempre più caratteristiche dall'accesso sequenziale che non dall'accesso casuale.

Alcuni sistemi operativi danno l'opportunità a certi programmi speciali di impiegare una partizione del disco come un grande array sequenziale di blocchi logici, non contenente alcuna struttura dati relativa al file system, detto disco di basso livello (*ratio disk*); l'I/O relativo si chiama I/O di basso livello (*raw I/O*). Alcuni sistemi per la gestione di basi di dati, ad esempio, preferiscono questo tipo di I/O perché permette di controllare l'esatta posizione nel disco d'ogni informazione trattata. Esso scalca tutti i servizi del file system: gestione delle buffer cache, prelievo anticipato, allocazione dello spazio, nomi dei file, directory, e così via. È possibile rendere certe applicazioni più efficienti includendovi servizi di memorizzazione secondaria specializzati che usino una partizione di basso livello, ma la maggior parte delle applicazioni è in realtà migliore quando usufruisce degli ordinari servizi del file system.

Boot block

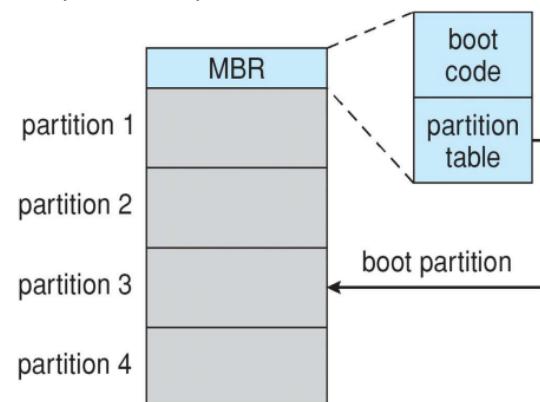
Affinché un calcolatore possa entrare in funzione, ad esempio quando viene acceso o riavviato, è necessario che esegua un programma iniziale; di solito, questo programma d'avviamento iniziale è piuttosto semplice. Esso inizializza il sistema in tutti i suoi aspetti, dai registri della CPU ai controllori dei dispositivi e al contenuto della memoria centrale, quindi avvia il sistema operativo. Per far ciò, il programma d'avviamento trova il kernel del sistema operativo nei dischi, lo carica nella memoria, e salta a un indirizzo iniziale per avviare l'esecuzione del sistema operativo.

Per la maggior parte dei calcolatori, il programma d'avviamento è memorizzato in una **ROM** (*read-only memory*), il che è conveniente, perché la ROM non richiede inizializzazione, e ha un indirizzo iniziale fisso dal quale la CPU può cominciare l'esecuzione ogniqualvolta si accende o si riavvia la macchina. Inoltre, visto che la ROM è a sola lettura, non può essere contaminata da un virus informatico. Il problema, però, è che cambiare il programma d'avviamento richiede in questo caso la sostituzione dei circuiti integrati ROM. A causa di questo inconveniente, molti sistemi memorizzano nella ROM un piccolo **bootstrap loader**

(caricatore d'avviamento) il cui solo compito è quello di caricare da un disco il programma d'avviamento completo. Quest'ultimo si può facilmente modificare: se ne scrive semplicemente una nuova versione nel disco. Il programma d'avviamento completo è registrato in una partizione del disco denominata blocchi d'avviamento, posta in una locazione fissata del disco; un disco contenente una tale partizione si chiama **disco d'avviamento o disco di sistema**.

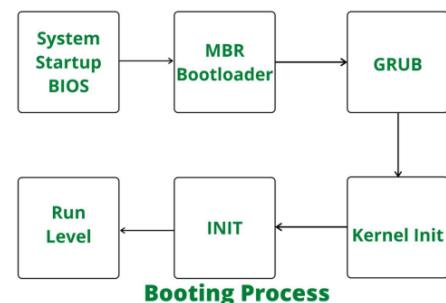
I codice contenuto nella ROM d'avviamento istruisce innanzitutto il controllore dell'unità a disco affinché trasferisca il contenuto dei blocchi d'avviamento nella memoria (si noti che a questo fine non si carica alcun driver di dispositivo), quindi comincia a eseguire il codice. Il programma d'avviamento (*bootstrap program*) completo è più complesso del suo loader, ed è capace di trasferire nella memoria l'intero sistema operativo inizialmente residente in un disco in una locazione non definitivamente fissata, e di avviare il sistema operativo stesso. Il bootstrap program potrà comunque essere breve.

Consideriamo come esempio il processo d'avviamento in Windows 2000. Questo sistema colloca il proprio codice d'avviamento nel primo settore del disco rigido, denominato MBR (master boot record). Esso, inoltre, consente di suddividere il disco rigido in una o più partizioni; in una di loro, detta **partizione d'avviamento**, sono contenuti il sistema operativo e i driver dei dispositivi. La procedura d'avviamento di Windows 2000 inizia con l'esecuzione del codice residente nella memoria ROM del sistema. Questa parte del codice guida il sistema a leggere il codice d'avviamento dall'MBR. Oltre al codice d'avviamento, l'MBR contiene una tabella che elenca le partizioni del disco rigido e un flag che indica da quale partizione si debba avviare il sistema; ciò è illustrato nella figura a destra. Dopo aver identificato la partizione d'avviamento, il sistema legge da tale partizione il primo settore (chiamato **settore d'avviamento**) e svolge le restanti procedure d'avviamento, tra cui il caricamento dei vari sottosistemi e dei servizi del sistema.



In Linux il booting funziona come segue:

- Firmware all'avvio test e lancio del bootloader iniziale
- Bootloader iniziale ha codice nel Master Boot Record (MBR)
- Bootloader di secondo livello (Grub2, systemd-boot)
- Il bootloader presenta un menu di possibili sistemi
- Una volta selezionato il sistema viene caricato il kernel in memoria
- Viene chiamato lo `start_kernel()` che lancia `idle_process`, `scheduler`, `init`



Bad block

Le unità a disco sono strutturalmente portate ai malfunzionamenti perché sono costituite da parti mobili a bassa tolleranza (si ricordi che una testina è sospesa appena sopra la superficie del disco). A volte si può verificare un guasto irreparabile, e l'unità a disco deve essere sostituita: le informazioni contenute nel disco dovranno essere recuperate da una copia di riserva mantenuta separatamente e trasferite nella nuova unità a disco. Più di frequente, uno o più settori divengono malfunzionanti; in effetti, la maggior parte dei dischi messi in commercio contiene già **blocchi difettosi** (*bad block*). Essi sono trattati in diversi modi secondo il controllore e l'unità a disco presenti nel sistema.

Nel caso di dischi semplici come quelli gestiti da un controllore IDE, i blocchi difettosi sono gestiti "manualmente". Unità a disco più complesse come i dischi SCSI in uso nei PC di alto livello e nella maggior parte delle stazioni di lavoro e server, hanno strategie di recupero dei blocchi difettosi più raffinate. Il

controllore mantiene una lista dei blocchi malfunzionanti dell'unità a disco che è inizializzata durante la formattazione fisica eseguita dal produttore, ed è aggiornata per tutto il periodo in cui l'unità a disco è operativa. La formattazione fisica mette anche da parte dei settori di riserva non visibili al sistema operativo: si può istruire il controllore affinché sostituisca da un punto di vista logico un settore difettoso con uno dei settori di riserva inutilizzati. Questa strategia è nota come **sector sparing** (accantonamento di settori). Un'alternativa all'accantonamento dei settori è data da quei controllori capaci di sostituire i settori difettosi tramite la tecnica di sector slipping (traslazione dei settori) che fa slittare i settori.

La sostituzione di un blocco difettoso non è in genere un processo totalmente automatico, perché i dati contenuti nel blocco in questione di solito vanno perduti. Un file che usava quel blocco deve quindi essere riparato (ad esempio, ricopiandolo da un nastro contenente le copie di riserva), e ciò richiede un intervento manuale. Nota che in NVM la gestione è più semplice poiché non c'è seek time da gestire.

Gestione dello Swap-Space

La **gestione dell'area di swapping** è un altro compito di basso livello del sistema operativo. La memoria virtuale usa lo spazio dei dischi come estensione della memoria centrale: poiché l'accesso alle unità a disco è molto più lento dell'accesso alla memoria centrale, l'uso di un'area di swapping, anche chiamata swap space, riduce notevolmente le prestazioni del sistema. L'obiettivo principale nella progettazione e realizzazione di uno swap space è di fornire la migliore produttività per il sistema della memoria virtuale.

Uso dello swap space

Lo swap space è usato in modi diversi da sistemi operativi diversi, in funzione degli algoritmi di gestione della memoria applicati. I sistemi che adottano lo swapping dei processi nella memoria, ad esempio, possono usare lo swap space per mantenere l'intera immagine del processo, inclusi i segmenti dei dati e del codice; i sistemi a paginazione, invece, possono semplicemente memorizzarvi pagine non contenute nella memoria centrale. Lo spazio richiesto dallo swap space per un sistema può quindi variare secondo la quantità di memoria fisica, la quantità di memoria virtuale che esso deve sostenere, e il modo in cui quest'ultima è usata.

Si noti che una stima per eccesso delle dimensioni dello swap space è più prudente di una per difetto, perché un sistema che esaurisca l'area di swapping potrebbe esser costretto a terminare forzatamente i processi o ad arrestarsi completamente: una stima per eccesso spreca spazio dei dischi che si potrebbe usare per i file, ma non provoca altri danni.

Collocazione dello swap space

Le possibili collocazioni per uno swap space sono due: all'interno del normale file system, o in una partizione del disco a sé stante. Se lo swap space è semplicemente un grande file all'interno del file system, si possono usare le ordinarie funzioni del file system per crearla, assegnargli un nome, e allocare spazio per essa. Questo criterio sebbene sia semplice da realizzare è inefficiente: l'attraversamento della struttura delle directory e l'uso delle strutture dati per l'allocazione dello spazio nei dischi richiede tempo, oltre che, almeno potenzialmente, accessi ai dischi aggiuntivi. La frammentazione esterna può aumentare molto i tempi di swapping causando ricerche multiple durante la scrittura o la lettura dell'immagine di un processo. Le prestazioni si possono migliorare impiegando la memoria fisica come cache per le informazioni relative alla posizione dei blocchi, e anche usando strumenti speciali per l'allocazione in blocchi fisicamente contigui del file di swapping, ma il costo dovuto all'attraversamento del file system e delle sue strutture dati permane.

In alternativa, lo swap space si può creare in un'apposita partizione del disco non formattata: in essa non è presente alcuna struttura relativa al file system e alle directory, ma si usa uno speciale gestore dello swap area per allocare e rimuovere i blocchi. Esso adotta algoritmi ottimizzati rispetto alla velocità, e non rispetto allo spazio impiegato, dato che allo swap space (quando viene utilizzato) si accede molto più

frequentemente che ai file system. La frammentazione interna può aumentare, ma questo prezzo da pagare è ragionevole perché i dati nello swap space hanno una vita media molto più breve dei file ordinari, e gli accessi allo swap space sono in genere molto più frequenti.

Poiché lo swap space è inizializzato all'avvio, la frammentazione ha vita breve. Questo metodo assegna una dimensione fisica allo swap space al momento della creazione delle partizioni del disco, e l'aumento delle dimensioni dello swap space deve quindi passare attraverso il ripartizionamento del disco (che implica lo spostamento o l'eliminazione e la sostituzione con copie di riserva delle altre partizioni del disco), o attraverso la creazione di un altro swap space in qualche altra unità a disco del sistema.

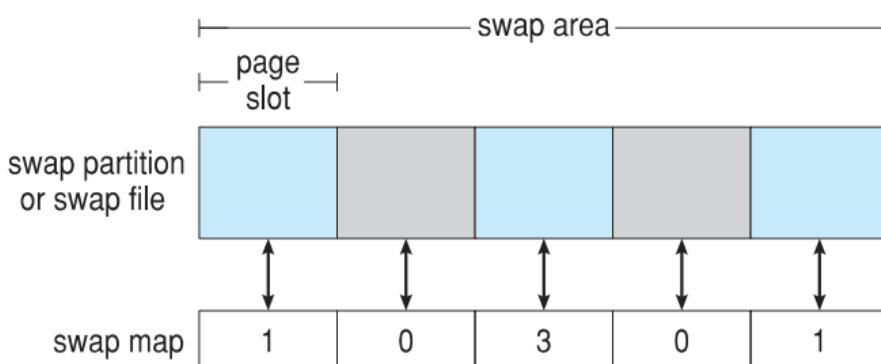
Gestione dello swapping in Linux

Si può comprendere come lo swap space venga applicato ripercorrendo la sua evoluzione e quella della paginazione nei vari sistemi UNIX. Il kernel tradizionale di UNIX implementava inizialmente una tecnica di swapping che spostava interi processi tra le regioni contigue del disco e la memoria. Più tardi, con la disponibilità dei dispositivi per la paginazione, UNIX progredì verso una combinazione di swapping e paging. Per migliorare l'efficienza e assimilare le innovazioni tecnologiche, in Solaris 1 (SunOS) i progettisti cambiarono le tecniche tradizionali di UNIX. Quando un processo va in esecuzione, le pagine contenenti il codice eseguibile sono prelevate dal file system, poste in memoria centrale e, qualora fossero state selezionate per l'espulsione, eliminate. Rileggere una pagina dal file system è più efficiente che scriverla nello swap space e rileggerla da lì: lo swap space è adoperato esclusivamente come area di stoccaggio per le pagine della memoria anonima, di cui fanno parte la memoria allocata per lo stack, quella per lo heap e i dati non inizializzati dei processi.

Ulteriori cambiamenti sono stati introdotti nelle versioni più recenti di Solaris. Il più importante prevede che lo swap space sia allocato solo quando una pagina è espulsa dalla memoria fisica, anziché quando al pagina è creata per la prima volta in memoria virtuale. Questa regola produce un miglioramento delle prestazioni nei calcolatori moderni, i quali, rispetto ai sistemi più vecchi, possono contare su una maggiore quantità di memoria fisica e limitare il ricorso alla paginazione.

Così come per Solaris, lo swap space di Linux è utilizzato soltanto per la memoria anonima o per regioni di memoria condivise tra numerosi processi. Linux permette di istituire uno o più swap space, sia in swap file del file system regolare, sia in una partizione a basso livello dedicata allo swap space (o swap area). Una swap area è formata da una serie di moduli di 4 KB detti **page slot**, la cui funzione è di conservare le pagine scambiate. Ogni area di swap ha una **swap map** associata, ovvero un array di contatori interi, ciascuno dei quali corrisponde a uno slot nello swap space. Se un contatore segna il valore 0, la pagina che gli corrisponde è disponibile. Valori superiori a 0 indicano che lo slot è occupato da una delle pagine scambiate. Il valore del contatore indica il numero di collegamenti alla pagina scambiata; ovvero il numero di processi a cui tale pagina è associata.

Le strutture dati relative allo swapping nei sistemi Linux sono illustrate nella seguente figura:



Strutture RAID

L'evoluzione tecnologica ha reso le unità a disco progressivamente più piccole e meno costose, tanto che oggi è possibile, senza eccessivi sforzi economici, equipaggiare un sistema di calcolo 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 **RAID** (*redundant array of independent [prima era inexpensive] disks*, batterie ridondanti di dischi), che hanno lo scopo di affrontare i problemi di prestazioni e affidabilità.

Mirrored Disk

Consideriamo in primo luogo l'affidabilità. La possibilità che uno dei dischi in un insieme di n dischi si guasti è molto più alta della possibilità che uno specifico disco isolato presenti un guasto. Si supponga che il tempo medio di guasto di un singolo disco sia 100.000 ore. In questo caso, il tempo medio di guasto per un qualsiasi disco in una batteria di 100 dischi sarebbe $100.000/100 = 1000$ ore, o 41,66 giorni; cioè, non molto tempo. Se si memorizzasse una sola copia dei dati, allora ogni guasto di un disco comporterebbe la perdita di una notevole quantità di dati; una frequenza di perdita di dati così alta sarebbe inaccettabile.

La soluzione al problema dell'affidabilità sta nell'introdurre una certa **ridondanza**, cioè nel memorizzare informazioni che non sono normalmente necessarie, ma che si possono usare nel caso di un guasto a un disco per ricostruire le informazioni perse.

Il metodo più semplice (ma anche il più costoso) di introduzione di ridondanza è quello del **mirroring** (copiatura speculare); ogni disco logico consiste di due dischi fisici e ogni scrittura si effettua in entrambi i dischi. Se uno dei dischi si guasta, i dati si possono leggere dall'altro. I dati si perdono solo se il secondo disco si guasta prima della sostituzione del disco già guasto.

Il tempo medio di guasto di un disco con copiatura speculare, dove per guasto s'intende ora la perdita di dati, dipende da due fattori: tempo medio di guasto di un singolo disco e il **tempo medio di riparazione** (mean time to data repair), cioè il tempo richiesto (in media) per sostituire un disco guasto e ripristinarvi i dati. Supponendo che i possibili guasti dei due dischi siano **indipendenti**, vale a dire che il guasto di un disco non sia mai legato a quello dell'altro, se il tempo medio di guasto di un singolo disco è 100.000 ore e il tempo medio di riparazione è di 10 ore, allora il tempo medio di perdita di dati di un sistema con copiatura speculare dei dischi è $\frac{100.000^2}{2 \cdot 10} = 500 \cdot 10^6$ ore, che corrispondono a 57000 anni!

Occorre però notare che i fallimenti non sono indipendenti:

- Problemi di alimentazioni particolarmente problematici durante la scrittura
- Scrivere in momenti diversi le repliche, oppure usare NVM come cache

Disk Striping

L'accesso in parallelo a più dischi può portare vari vantaggi. Con la copiatura speculare dei dischi, la frequenza con la quale si possono gestire le richieste di lettura raddoppia, poiché ciascuna richiesta si può inviare indifferentemente a uno dei due dischi (sempre che entrambi i dischi siano funzionanti, condizione che è quasi sempre soddisfatta). La capacità di trasferimento di ciascuna lettura è la stessa di quella di un sistema a singolo disco, ma il numero di letture per unità di tempo raddoppia.

Attraverso l'uso di più dischi è possibile anche (o alternativamente) migliorare la capacità di trasferimento distribuendo i dati in sezioni su più dischi. Nella sua forma più semplice questa distribuzione, chiamata **data striping** (sezionamento dei dati), consiste nel distribuire i bit di ciascun byte su più dischi; in questo caso si parla di **bit level striping**: ogni i -esimo bit sul drive i -esimo. Ad esempio, se il sistema impiega una batteria

di otto dischi, si scriverà il bit i di ciascun byte nel disco i . La batteria di otto dischi si può trattare come un unico disco avente settori che hanno una dimensione otto volte superiore a quella normale e, soprattutto, che hanno una capacità di trasferimento otto volte superiore.

Il sezionamento non si deve realizzare necessariamente a livello dei bit di un byte: nel **block level striping**, ad esempio, i blocchi di un file si distribuiscono su più dischi; con n dischi, il blocco i di un file si memorizza nel disco $(i \bmod n) + 1$.

Riassumendo, gli obiettivi principali riguardo al parallelismo in un sistema di dischi sono due:

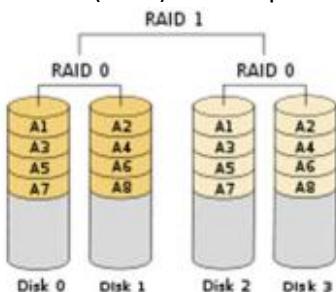
- Incrementare il throughput di piccoli accessi bilanciando il carico; ovvero l'aumento della produttività per accessi multipli a piccole porzioni di dati (cioè accessi a pagine);
- Ridurre il tempo di risposta relativo ad accessi a grandi quantità di dati.

Livelli RAID

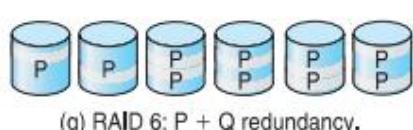
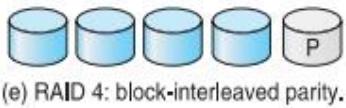
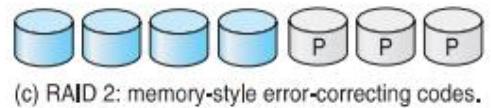
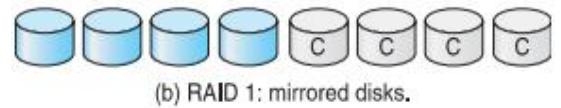
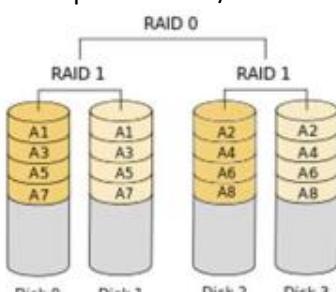
La tecnica di mirroring offre un'alta affidabilità ma è costosa; la tecnica dello striping offre un'altra capacità di trasferimento dei dati ma non migliora l'affidabilità. Sono stati proposti numerosi schemi per fornire ridondanza usando l'idea del sezionamento combinata con i bit di parità. Questi schemi realizzano diversi compromessi tra costi e prestazioni e sono stati classificati in livelli chiamati livelli RAID che la figura a destra mostra graficamente (nella figura, la lettera P indica i bit di correzione degli errori, la lettera C indica una seconda copia dei dati). In tutti i casi riportati nella figura, sono presenti quattro dischi di dati, mentre i dischi supplementari s'impiegano per memorizzare le informazioni ridondanti per il ripristino dai guasti.

Raid (0 + 1) e (1 + 0). Striped mirrors (RAID 1+0) o mirrored stripes (RAID 0+1) forniscono alta performance e alta affidabilità: RAID 0 performance, RAID 1 affidabilità

- RAID 01 (0 + 1) Dati striped e duplicate



- RAID 10 (1 + 0) Prima dati duplicati poi striped sui dischi (almeno 4) Vicino a RAID 0 per performance (usato per sistemi I/O intenso) più ridondanza



Altre caratteristiche

Indipendentemente da come il RAID è implementato si possono aggiungere altri strumenti:

- **Snapshot**
 - vista del file system prima che occorra un insieme di cambiamenti (ad esempio, ad un certo istante di tempo)
- **Duplicazione automatica** delle scritture tra siti separati
 - Per ridondanza e disaster recovery
 - Può essere sincrona o asincrona
- **Dischi di hot spare** (di ricambio)
 - non usati per dati, ma solo in caso di fallimento per replicare il disco fallito e ricostruire il RAID set se possibile
 - Diminuisce il mean time to repair

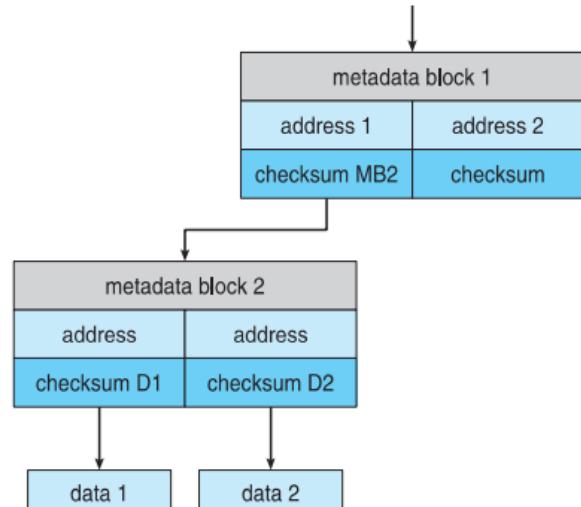
Estensioni

I concetti relativi ai sistemi RAID sono stati generalizzati ad altri dispositivi di memorizzazione, comprese le batterie di nastri e anche alla diffusione dei dati tramite sistemi senza fili (wireless). Con strutture RAID applicate alle batterie di nastri si possono ripristinare i dati anche se uno dei nastri della batteria è danneggiato. Se applicate alla trasmissione dei dati, si divide ogni blocco di dati in unità più piccole che si trasmettono insieme a un'unità di parità; se per qualsiasi ragione una delle unità non viene ricevuta, può essere ricostruita dalle altre. Di solito, con l'uso di unità automatiche dotate di molte unità a nastro si esegue il sezionamento dei dati su tutte le unità per aumentare la produttività e diminuire il tempo di trasferimento dei dati.

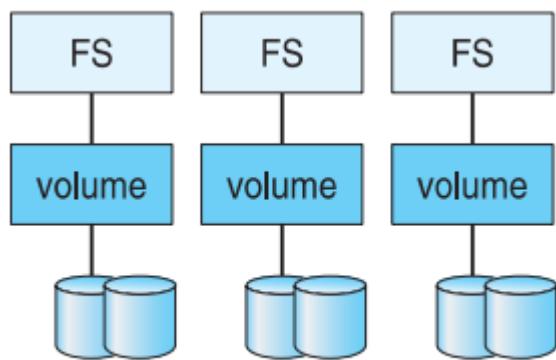
Problemi connessi a RAID

RAID da solo non previene o trova tutti gli errori, solo errori di disk failures. Per risolvere problemi dovuti ai dispositivi e ai programmi (come puntatori che indicano dati sbagliati, operazioni incomplete di scrittura), il file system Solaris ZFS ricorre a una strategia innovativa. Esso applica una somma di controllo (**checksum**) interna a ogni blocco, dati e metadati inclusi. Un'ulteriore funzionalità deriva dalla collocazione delle somme di controllo, che non risiedono nel blocco sottostante a controllo: ciascuna di loro, invece, è memorizzata insieme al puntatore a quel blocco (vedi figura).

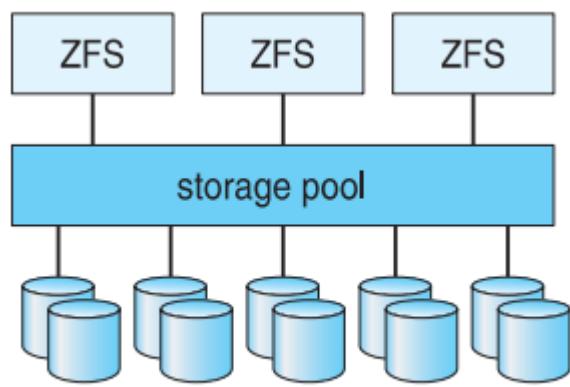
ZFS combina la gestione dei file system e quella dei volumi in una unità in grado di offrire una maggiore funzionalità rispetto a quella permessa dalla tradizionale separazione di tali funzioni. I dischi, o le partizioni di dischi, sono riuniti in gruppi di memorizzazione (pools of Storage) attraverso insiemi RAID. Un gruppo può contenere uno o più file system ZFS. Tutta l'area libera di un gruppo è a disposizione di tutti i file system contenuti all'interno di quel gruppo. Quando i blocchi vengono utilizzati e liberati all'interno del file system ZFS utilizza "malloc" e "free" per allocare e rilasciare memoria in ogni file system. Di conseguenza non ci sono limiti artificiali all'utilizzo della memoria e non sussiste la necessità di ridistribuire i file system tra i volumi né di ridimensionare i volumi. ZFS stabilisce delle quote per limitare la dimensione di un file system e definisce dei parametri per assicurarsi che il file system possa aumentare all'interno di una dimensione specificata, ma queste variabili possono essere sempre cambiate dal proprietario del file system.



La figura (a) illustra i volumi e i file system tradizionali, mentre la figura (b) rappresenta il modello ZFS:



(a) Traditional volumes and file systems.



(b) ZFS and pooled storage.

13. Sistemi di I/O

Architetture e dispositivi di I/O

Un dispositivo comunica con un sistema di calcolo inviando segnali attraverso un cavo o attraverso l'etere e comunica con il calcolatore tramite un punto di connessione (porta), ad esempio una porta seriale. Se uno o più dispositivi usano in comune un insieme di fili, la connessione è detta bus. Un **bus** è un insieme di fili e un protocollo rigorosamente definito che specifica l'insieme dei messaggi che si possono inviare attraverso i fili.

Un **controllore** è un insieme di componenti elettronici che può far funzionare una porta, un bus o un dispositivo.

Una porta di I/O consiste tipicamente di quattro registri: status, control, data-in e data-out.

- Il registro *status* contiene alcuni bit che possono essere letti e indicano lo stato della porta; ad esempio indicano se è stata portata a termine l'esecuzione del comando corrente, se un byte è disponibile per essere letto dal registro data-in, se si è verificato un errore del dispositivo.
- Il registro *control* può essere scritto per attivare un comando o per cambiare il modo di funzionamento del dispositivo. Ad esempio, un certo bit nel registro *control* della porta seriale determina il tipo di comunicazione tra *half-duplex* e *full-duplex*, un altro abilita il controllo di parità, un terzo imposta la lunghezza delle parole a 7 o 8 bit, altri selezionano una tra le velocità che la porta seriale può sostenere
- La CPU legge dal registro data-in per ricevere i dati.
- La CPU scrive nel registro data-out per emettere dati.

Polling

Si assume l'uso di due bit per coordinare la relazione di tipo produttore-consumatore fra il controllore e la CPU. Il controllore specifica il suo stato per mezzo del bit *busy* del registro *status*; pone a 1 il bit *busy* quando è impegnato in un'operazione, e lo pone a 0 quando è pronto a eseguire il comando successivo. La CPU comunica le sue richieste tramite il bit *command-ready* nel registro *command*: pone questo bit a 1 quando il controllore deve eseguire un comando. In questo esempio, la CPU scrive in una porta coordinandosi con un controllore per mezzo dell' **handshaking** (negoziazione) come segue:

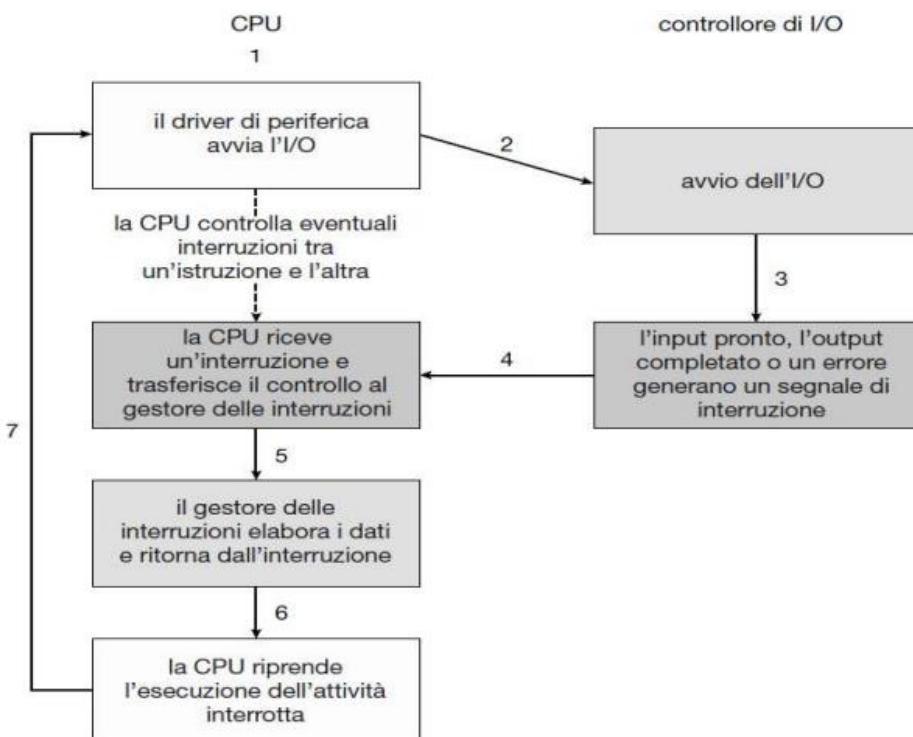
- 1) La CPU legge ripetutamente il bit *busy* fino a che esso non vale 0
- 2) La CPU pone a 1 il bit *write* del registro dei comandi e scrive un byte nel registro data-out
- 3) La CPU pone a 1 il bit *command-ready*
- 4) Quando il controllore si accorge del fatto che il bit *command-ready* è posto a 1, pone a 1 il bit *busy*
- 5) Il controllore legge il registro dei comandi e trova il comando *write*; legge il registro data-out per ottenere il byte da scrivere, e compie l'operazione di scrittura nel dispositivo.
- 6) Il controllore pone a 0 il bit *command-ready*, pone a 0 il bit *error* nel registro *status* per indicare che l'operazione di I/O ha avuto esito positivo, e pone a 0 il bit *busy* per indicare che l'operazione è terminata

La sequenza sopradescritta si ripete per ogni byte. Durante l'esecuzione del passo 1 la CPU è in **busy waiting** o in **polling**: itera la lettura del registro *status* fino a che il bit *busy* assume il valore 0. Se il controllore e il dispositivo sono veloci, questo metodo è ragionevole, ma se l'attesa rischia di prolungarsi, sarebbe probabilmente meglio se la CPU si dedicasse a un'altra operazione (in molte architetture sono sufficienti tre cicli di istruzioni di CPU per interrogare ciclicamente un dispositivo).

Interrupt

La CPU ha un contatto, detto **linea di richiesta dell'interrupt**, del quale la CPU controlla lo stato dopo l'esecuzione di ogni istruzione. Quando rileva il segnale di un controllore nella linea di richiesta

dell'interruzione, la CPU memorizza una piccola quantità di informazioni sullo stato dell'elaborazione corrente (ad esempio il valore del puntatore alle istruzioni) e salta alla **interrupt handler routine** (procedura di gestione delle interruzioni), che si trova a un indirizzo di memoria fissato. Questa procedura determina le cause dell'interruzione, porta a termine l'elaborazione necessaria ed esegue un'istruzione return from interrupt per far sì che la CPU ritorni nello stato in cui si trovava prima della sua interruzione. Il controllore del dispositivo genera un segnale d'interrupt della CPU lungo la linea di richiesta delle interruzioni, che la CPU rileva e recapita al gestore delle interruzioni, che a sua volta evade il compito corrispondente servendo il dispositivo. Nella figura seguente è riassunto il ciclo di I/O indotto da un segnale d'interruzione della CPU.



Accesso diretto alla memoria (DMA)

Per dare avvio a un trasferimento DMA, la CPU scrive nella memoria un comando strutturato per il DMA. Esso contiene un puntatore alla locazione dei dati da trasferire, un altro puntatore alla destinazione dei dati, e il numero dei byte da trasferire. La CPU scrive l'indirizzo di questo comando strutturato nel controllore del DMA, e prosegue con l'esecuzione di altro codice. Il controllore del DMA agisce quindi direttamente sul bus della memoria, presentando al bus gli indirizzi di memoria necessari per eseguire il trasferimento senza l'aiuto della CPU.

La procedura di negoziazione tra il controllore del DMA e il controllore del dispositivo si svolge grazie a una coppia di fili detti DMA-request e DMA-acknowledge. Il controllore del dispositivo manda un segnale sulla linea DMA-request quando una parola di dati è disponibile per il trasferimento. Questo segnale fa sì che il controllore DMA prenda possesso del bus di memoria, presenti l'indirizzo desiderato ai fili d'indirizzamento della memoria e mandi un segnale lungo la linea DMA-acknowledge. Quando il controllore del dispositivo riceve questo segnale, trasferisce nella memoria la parola di dati e rimuove il segnale dalla DMA-request.

Quando l'intero trasferimento termina, il controllore del DMA interrompe la CPU. Quando il controllore del DMA prende possesso del bus di memoria, la CPU è temporaneamente impossibilitata ad accedere alla memoria centrale, sebbene abbia accesso ai dati contenuti nella sua cache primaria e secondaria. Questo fenomeno, noto come sottrazione di cicli, può rallentare le computazioni della CPU; ciononostante l'assegnamento del lavoro di trasferimento di dati a un controllore DMA migliora in generale le prestazioni complessive del sistema evitando che la CPU si sovraccarichi troppo.

Interfaccia di I/O per le applicazioni

Ogni tipo di SO ha le sue convenzioni riguardanti l'interfaccia dei driver dei dispositivi (sfortunatamente per chi li produce). I dispositivi possono differire in molti aspetti:

- **Trasferimento a flusso di caratteri o a blocchi.** Un dispositivo del primo tipo trasferisce dati un byte alla volta, mentre uno del secondo tipo trasferisce un blocco di byte in un'unica soluzione.
- **Accesso sequenziale o diretto.** Un dispositivo del primo tipo trasferisce dati secondo un ordine prestabilito e invariabile, mentre l'utente di un dispositivo ad accesso diretto può richiedere l'accesso a una qualunque delle possibili locazioni di memorizzazione.
- **Dispositivi sincroni o asincroni.** Un dispositivo sincrono trasferisce dati con un tempo di risposta prevedibile, mentre un dispositivo asincrono ha tempi di risposta irregolari o non prevedibili.
- **Condivisibili o riservati.** Un dispositivo condivisibile può essere usto in modo concorrente da diversi processi, mentre ciò è impossibile se il dispositivo è riservato.
- **Velocità di funzionamento.** Può variare da alcuni byte al secondo fino a qualche GB al secondo.
- **Lettura e scrittura, sola lettura o sola scrittura.** Alcuni dispositivi possono emettere e ricevere dati, ma altri consentono solo una delle due possibilità

Dispositivi con trasferimento a blocchi o a caratteri

L'interfaccia per i **dispositivi a blocchi** sintetizza tutti gli aspetti necessari per accedere alle unità a disco e ad altri dispositivi basati sul trasferimento di blocchi di dati. Di solito le applicazioni comunicano con questi dispositivi tramite un file system che funge da interfaccia. Il sistema operativo e certe applicazioni particolari come quelle per la gestione delle basi di dati possono trovare più conveniente trattare questi dispositivi come una semplice sequenza lineare di blocchi. In questo caso si parla di **raw I/O**.

L'accesso ai file associati alla memoria può essere posto a un livello gerarchico immediatamente superiore a quello dei dispositivi a blocchi. Piuttosto che offrire funzioni di lettura e scrittura, un'interfaccia per l'accesso associato alla memoria fornisce la possibilità di usare un'unità a disco tramite un vettore di byte della memoria centrale. La chiamata del sistema che associa un file a una regione di memoria restituisce l'indirizzo di memoria virtuale di un vettore di caratteri che contiene una copia del file. Gli effettivi trasferimenti di dati sono eseguiti solo quando sono necessari per soddisfare una richiesta di accesso all'immagine nella memoria. L'I/O associato alla memoria è efficiente.

La tastiera è un esempio di dispositivo al quale si accede tramite un'interfaccia a **flusso di caratteri**. Le chiamate del sistema fondamentali per le interfacce di questo tipo permettono a un'applicazione di acquisire o inviare un carattere (get o put). Basandosi su quest'interfaccia è possibile costruire servizi aggiuntivi quali l'accesso riga per riga, la correzione. Questo tipo di accesso è conveniente per dispositivi che producono dati spontaneamente (tastiere, mouse, modem) o dispositivi che emettono dati organizzati in modo naturale (stampanti o schede audio).

Orologi e temporizzatori

La maggior parte dei calcolatori ha temporizzatori e orologi che forniscono tre funzioni essenziali: segnare l'ora corrente; segnalare il tempo trascorso; regolare un temporizzatore in modo da avviare l'operazione x al tempo t .

Il dispositivo che misura la durata di un lasso di tempo e che può avviare un'operazione si chiama **temporizzatore programmabile**; si può regolare in modo da attendere un certo tempo e poi generare un segnale d'interruzione, e può anche ripetere questo processo un numero prefissato di volte, generando così interruzioni periodiche. Lo scheduler usa questo meccanismo per generare un segnale d'interruzione che sospende un processo quando il suo quanto di tempo è scaduto. Il sottosistema dell'I/O delle unità a disco lo usa per riversare periodicamente nei dischi il contenuto della buffer cache. In certi casi, simulando orologi virtuali, il sistema operativo può anche gestire un numero di richieste d'uso dei temporizzatori maggiore del numero dei temporizzatori fisici. Per far ciò il kernel mantiene un elenco ordinato

cronologicamente delle interruzioni richieste dagli utenti e dalle proprie procedure, e imposta il temporizzatore per la prima scadenza. Quando il temporizzatore genera il segnale d'interruzione, il kernel avvisa il richiedente, e reimposta il temporizzatore per la prossima scadenza.

I/O bloccante e non bloccante

Un altro aspetto delle chiamate del sistema è la scelta fra I/O bloccante e non bloccante (asincrono).

Quando un'applicazione impiega una chiamata del sistema bloccante, si sospende l'esecuzione dell'applicazione, che passa dalla coda dei processi pronti per l'esecuzione alla coda d'attesa del sistema. Quando la chiamata del sistema termina, l'applicazione è posta nuovamente nella coda dei processi pronti per l'esecuzione in modo che possa riprendere l'esecuzione; solo allora essa riceverà i valori riportati dalla chiamata del sistema. Le operazioni fisiche compiute dai dispositivi di I/O sono in genere asincrone (richiedono un tempo variabile o non prevedibile).

Alcuni processi al livello utente necessitano di una forma non bloccante di I/O. Un esempio è quello di un'interfaccia utente con cui s'interagisce col mouse e la tastiera mentre elabora dati e li mostra su schermo. Una possibile alternativa alle chiamate del sistema non bloccanti è costituita dalle chiamate del sistema asincrone. Esse restituiscono immediatamente il controllo al chiamante, senza attendere che l'I/O sia stato completato. L'applicazione continua a essere eseguita, e il completamento dell'I/O è successivamente comunicato all'applicazione per mezzo dell'impostazione del valore di una variabile nello spazio di indirizzi dell'applicazione o tramite la generazione di un segnale.

Sottosistema per l'I/O del kernel

Il nucleo fornisce molti servizi riguardanti l'I/O, molti sono offerti dal sottosistema per l'I/O del kernel, e sono realizzati a partire dai dispositivi e dai relativi driver.

Scheduling

Fare lo scheduling di un insieme di richieste di I/O significa stabilirne un ordine d'esecuzione efficace; l'ordine in cui si verificano le chiamate del sistema delle applicazioni è raramente la scelta migliore. Lo scheduling può migliorare le prestazioni complessive del sistema, distribuire equamente gli accessi dei processi ai dispositivi e ridurre il tempo d'attesa medio per il completamento di un'operazione di I/O. Ecco un semplice esempio che illustra queste potenzialità. Si supponga che la testina di lettura di un'unità a disco sia vicina alla parte iniziale del disco, e che tra applicazioni impartiscano comandi di lettura bloccanti per quest'unità. L'applicazione 1 richiede la lettura di un blocco che si trova vicino alla parte finale del disco, l'applicazione 2 quella di un blocco vicino alla parte iniziale e l'applicazione 3 quella di un blocco situato nella zona centrale. Il sistema operativo può ridurre la distanza percorsa dalla testina del disco servendo le richieste nell'ordine 2, 3, 1. Simili riordinamenti delle sequenze di servizio delle richieste sono l'essenza dello scheduling dell'I/O.

I progettisti di sistemi operativi realizzano lo scheduling mantenendo una coda di richieste per ogni dispositivo. Quando un'applicazione richiede l'esecuzione di una chiamata del sistema di I/O bloccante, si aggiunge la richiesta alla coda relativa al dispositivo appropriato. Lo scheduler dell'I/O riorganizza l'ordine della coda per migliorare l'efficienza globale del sistema e il tempo medio d'attesa cui sono sottoposte le applicazioni. Il sistema operativo può anche tentare di essere equo, in modo che nessuna applicazione riceva un servizio carente, o può dare priorità a quelle richieste la cui corretta esecuzione potrebbe essere inficiata da un ritardo nel servizio. Ad esempio, le richieste del sottosistema per la memoria virtuale potrebbero necessitare di una priorità maggiore di quelle delle applicazioni.

Lo scheduling dell'I/O è uno dei modi in cui il sottosistema per l'I/O migliora l'efficienza di un calcolatore; un altro è l'uso di spazio di memorizzazione nella memoria centrale o nei dischi, per tecniche di memorizzazione transitoria, uso di cache e di code per la gestione asincrona dell'I/O.

Memorizzazione transitoria

Un **buffer** (memoria di transito) è un'area di memoria che contiene dati mentre essi sono trasferiti fra due dispositivi o tra un'applicazione e un dispositivo. La memorizzazione transitoria si usa per tre ragioni. La prima è la necessità di gestire la differenza di velocità fra il produttore e il consumatore di un flusso di dati. Si supponga, ad esempio, di ricevere un file attraverso un modem e di volerlo memorizzare in un'unità a disco: il modem è circa mille volte più lento del disco, perciò conviene predisporre un'area della memoria centrale per contenere i byte che giungono dal modem. Quando tale area di memoria è piena, si trasferisce il suo contenuto nel disco con un'unica operazione. Poiché quest'operazione di scrittura non è istantanea e il modem ha bisogno di ulteriore spazio per memorizzare i dati in arrivo, è necessario impiegare due aree della memoria di questo tipo: quando la prima è piena, si richiede la scrittura nel disco del suo contenuto e il modem comincia a scrivere nella seconda area di memoria; la scrittura nel disco dovrebbe terminare prima che il modem possa riempirla; cosicché il modem potrà ricominciare a usare la prima area della memoria, mentre si trasferisce nel disco il contenuto della seconda. Questo doppio buffer svincola il produttore dal consumatore, rendendo così meno critico il problema della loro sincronizzazione.

Un secondo uso della memorizzazione transitoria riguarda la gestione dei dispositivi che trasferiscono dati in blocchi di dimensioni diverse. Queste disparità sono particolarmente comuni nelle reti di calcolatori, dove spesso è necessario frammentare e ricomporre messaggi. Quando un mittente spedisce un messaggio molto lungo, esso è spezzato in piccoli pacchetti che si spediscono attraverso la rete; il sistema destinatario provvede a ricostituire in un'apposita memoria di transito l'intero messaggio originario.

Il terzo modo in cui si può impiegare un buffer è per la realizzazione della *semantica delle copie* nell'ambito dell'I/O delle applicazioni. Un esempio ne chiarirà il significato. Si supponga che un'applicazione disponga di un'area di memoria contenente dati da trasferire in un disco: essa richiederà l'esecuzione della chiamata del sistema write, fornendole un puntatore a tale area della memoria e un numero intero che specifica il numero di byte da trasferire. Ci si può chiedere cosa succede se, dopo che la chiamata del sistema restituisce il controllo all'applicazione, quest'ultima modifica il contenuto dell'area di memoria. Ebbene, la **semantica delle copie** garantisce che la versione dei dati scritta nel disco sia conforme a quella contenuta nell'area di memoria al momento della chiamata del sistema, indipendentemente da ogni successiva modifica. Una semplice maniera di realizzare questa semantica consiste nel far sì che la chiamata del sistema write copi i dati forniti dall'applicazione stessa. La scrittura nel disco si compie dalla memoria del kernel, cosicché ogni successivo cambiamento nell'area di memoria per l'I/O dell'applicazione non impedirà che la versione dei dati trasferita nel disco sia quella corretta. In molti sistemi operativi si usa il metodo appena descritto: nonostante esso implichi una diminuzione dell'efficienza di certe operazioni di I/O, la sua semantica è chiara. Lo stesso effetto, tuttavia, si può ottenere più efficientemente tramite un uso intelligente della memoria virtuale e della protezione data dalla copiatura su scrittura delle pagine.

Code e uso esclusivo dei dispositivi

Una coda di file da stampare (*spool*) è un buffer contenente dati per un dispositivo che non può accettare flussi di dati intercalati, ad esempio una stampante. Sebbene una stampante possa servire una sola richiesta alla volta, diverse applicazioni devono poter richiedere simultaneamente la stampa di dati, senza che questi si mischino. Il sistema operativo risolve questo problema filtrando tutti i dati per la stampante: i dati da stampare provenienti da ogni singola applicazione si registrano in uno specifico file in un disco, quando un'applicazione termina di emettere i dati da stampare, si aggiunge tale file alla coda di stampa; quest'ultima viene copiata sulla stampante, un file per volta. In certi sistemi operativi, questa funzione viene gestita da un processo di sistema specializzato (demone), in altri da un thread del kernel. In ogni caso, il sistema operativo fornisce un'interfaccia di controllo che permette agli utenti e agli amministratori del sistema di esaminare la coda, eliminare elementi della coda prima che essi siano stampati, sospendere una stampa in corso, e così via.

Alcuni dispositivi, come le unità a nastro e le stampanti, non possono gestire autonomamente in modo efficiente più richieste concorrenti di I/O da parte di diverse applicazioni. L'accodamento è uno dei modi in cui il sistema operativo può coordinare le emissioni simultanee di dati; un altro è quello di fornire esplicite funzioni di coordinamento. Alcuni sistemi operativi permettono di accedere a un dispositivo in modo esclusivo: un processo può accedere a un dispositivo che non sia già attivo, riservandosene l'uso; e restituirlo al sistema quando non ne ha più bisogno. Altri sistemi impediscono l'apertura di più di una maniglia id file per un dato dispositivo. Molti sistemi operativi forniscono funzioni che permettono ai processi stessi di coordinare l'uso esclusivo dei dispositivi.

Gestione degli errori

Un sistema operativo che usa la protezione della memoria può proteggersi da molti tipi di errori dovuti ai dispositivi o alle applicazioni, cosicché il blocco completo del sistema non è l'ordinaria conseguenza di piccoli difetti tecnici. I dispositivi e i trasferimenti di I/O possono causare errori in molti modi, sia per motivi contingenti, come il sovraccarico di una rete di comunicazione, sia per ragioni "permanenti", come nel caso in cui il controllore dell'unità a disco sia difettoso. I sistemi operativi sono spesso capaci di compensare efficacemente le conseguenze negative dovute a errori generati da cause contingenti. Purtroppo, però, è improbabile che il sistema operativo riesca a compensare gli effetti di errori dovuti a disfunzioni permanenti di qualche componente importante.

Di norma, una chiamata del sistema per l'I/O riporta un bit d'informazione sullo stato d'esecuzione della chiamata, denotando con esso la riuscita o l'insuccesso dell'operazione richiesta. Il sistema operativo UNIX usa una variabile intera detta errno per codificare piuttosto genericamente il tipo d'errore avvenuto; i valori possibili sono un centinaio e denotano errori dovuti ad esempio a puntatori non validi, file non aperti o argomenti oltre i limiti ammessi. Per contro, alcuni tipi di dispositivi possono fornire informazioni assai dettagliate sugli errori, sebbene molti sistemi operativi attuali non siano progettati per passare questi dati alle applicazioni.