

Sistemi Operativi

Alessandro Trigolo

29 febbraio 2024

Indice

I Gestione dei Processi	9
1 Processi	9
1.1 Allocazione in memoria	9
1.1.1 Process Control Block (PCB)	9
1.2 Stati di un processo	10
1.2.1 Context switch	11
1.2.2 Creazione di un processo	11
1.2.3 L'albero dei processi in Linux	13
1.2.4 Terminazione di un processo	13
1.3 Comunicazione tra processi (IPC)	13
1.3.1 Memoria condivisa	14
1.3.2 Passaggio di messaggi	14
2 Threads	15
2.1 Concorrenza e parallelismo	16
2.1.1 Tipi di parallelismo	16
2.1.2 Legge di <i>Amdahl</i>	17
2.2 Modelli multithreading	18
2.2.1 Many-to-One	18
2.2.2 One-to-One	18
2.2.3 Many-to-Many	18
2.3 Librerie di thread	19
2.4 Threading implicito	20
2.4.1 Modello fork-join	20
2.4.2 Thread pools e OpenMP	20
2.5 Problematiche	21
2.5.1 Semantica exec e fork	21
2.5.2 Segnalazione ed eliminazione	21
3 CPU Scheduling	23
3.1 Nozioni fondamentali	23
3.2 Algoritmi non preemptive	24
3.2.1 First-Come First-Served (FCFS)	24
3.2.2 Shortest-Job-First (SJF)	25
3.2.3 Stima del <i>CPU burst time</i>	25
3.3 Algoritmi preemptive	26
3.3.1 Shortest-Remaining-Time-First (SRTF)	26
3.3.2 Round Robin (RR)	27
3.3.3 Reattività	28
3.4 Scheduling con priorità	29
3.4.1 Scheduling con priorità e RR	30
3.4.2 Coda multilivello	30
3.5 Scheduling multiprocessore	31
3.5.1 Symmetric multiprocessing (SMP)	31
3.5.2	31

3.6	Scheduling real-time	31
3.7	Valutazione di un algoritmo	31
II	Sincronizzazione dei Processi	32
4	Sincronizzazione	32
4.1	Sezione critica	32
4.1.1	Requisiti	33
4.1.2	Soluzioni inefficienti	33
4.2	Soluzione di Peterson	33
4.2.1	Architetture moderne	34
4.3	Sincronizzazione via hardware	35
4.3.1	Test and set	35
4.3.2	Compare and swap	35
4.3.3	Variabili atomiche	36
4.4	Mutex lock e Semafori	36
4.4.1	Waiting queue	38
4.5	Monitor	38
4.5.1	Struttura e implementazione	39
4.5.2	Variabile di condizione	40
4.6	Problemi comuni della sincronizzazione	40
4.6.1	Buffer limitato	40
4.6.2	Problema dei lettori e degli scrittori	41
4.6.3	Problema dei 5 filosofi	42
5	Deadlocks	45
5.1	Nozioni fondamentali	45
5.1.1	Caratterizzazione	45
5.1.2	Grafo risorsa-allocazione	46
5.2	Avoidance	48
5.2.1	Safe state	48
5.2.2	Algoritmo sul grafo risorsa-allocazione	49
5.2.3	Algoritmo del banchiere	50
5.3	Detection	52
5.3.1	Istanza singola	52
5.3.2	Istanze multiple	53
5.3.3	Deadlock recovery	54
III	Gestione della Memoria	55
6	Memoria principale	55
6.1	Introduzione	55
6.1.1	Protezione	56
6.1.2	Binding	56
6.1.3	Memory-Management Unit (MMU)	57
6.1.4	Caricamento e collegamento dinamico	58
6.2	Primi modelli di allocazione	58

6.2.1	Allocazione contigua	58
6.2.2	Allocazione a partizione fissa	59
6.2.3	Allocazione a partizione variabile	60
6.2.4	Problema della frammentazione	61
6.3	Paginazione (<i>paging</i>)	62
6.3.1	Frammentazione interna	63
6.3.2	Traduzione degli indirizzi	64
6.3.3	Allocazione nei frames liberi	65
6.4	Page table	66
6.4.1	Translation Look-aside Buffer (TLB)	66
6.4.2	Bit di validità	68
6.4.3	Page table gerarchica	69
6.4.4	Page table con tabella <i>hash</i>	70
6.4.5	Page table invertita	71
6.5	Swapping	72
6.5.1	Swapping con paginazione	73
6.5.2	Swapping nei dispositivi mobili	73
6.6	Segmentazione (<i>segmentation</i>)	73
6.6.1	Segment table	74
6.6.2	Modello ibrido	75
7	Memoria virtuale	76
7.1	Introduzione	76
7.1.1	Spazio degli indirizzi virtuali	76
7.1.2	Memoria condivisa	77
7.2	Demand Paging	77
7.2.1	Page fault	77
7.2.2	Pure demand paging	79
7.2.3	Performance e ottimizzazione	80
7.2.4	Prepaging	81
7.3	Page replacement	81
7.3.1	FIFO	82
7.3.2	Algoritmo ottimale	82
7.3.3	LRU	83
7.3.4	Second-chance (clock)	84
7.3.5	Algoritmo counting	85
7.3.6	Ottimizzazione	86
7.4	Allocazione dei frames	86
7.4.1	Allocazione globale e locale	86
7.4.2	Richiesta delle pagine	87
7.5	Thrashing	87
7.5.1	Modello Working-set	88
7.5.2	Frequenza dei page fault (PFF)	89
7.6	Allocare la memoria del kernel	89
7.6.1	Buddy system	90
7.6.2	Slab allocation	90

IV Gestione dello storage 92

8 Memoria di massa	92
8.1 Tipi di memoria secondaria	92
8.1.1 HDD	92
8.1.2 NVM	93
8.1.3 Memoria volatile	94
8.1.4 Nastro magnetico	94
8.1.5 Dispositivi di memorizzazione esterna	94
8.2 Indirizzamento	95
8.3 HDD scheduling	95
8.3.1 FCFS	95
8.3.2 SSTF	96
8.3.3 SCAN e C-SCAN	96
8.3.4 Scelta dell'algoritmo	97

Elenco delle figure

1	Spazio in memoria allocato per il processo dal sistema operativo.	9
2	Rappresentazione del contenuto di un generico PCB.	10
3	Lista concatenata che mantiene tutti i PCB dei processi (task) in Linux.	10
4	Gli stati della vita di un processo.	11
5	Il context switch.	12
6	Rappresentazione delle 3 <i>system calls</i> fondamentali.	12
7	L'albero dei processi generato da <code>systemd</code>	13
8	Il modello di memoria condivisa per IPC.	14
9	Il modello di memoria condivisa per IPC.	15
10	Esempio di concorrenza tra 4 processi.	16
11	Esempio di parallelismo in un sistema dual core.	16
12	Esempio di parallelismo di dati.	16
13	Esempio di parallelismo di compiti.	17
14	Il grafico che descrive lo <i>speedup</i> a seconda della percentuale di codice seriale.	17
15	Il modello di <i>multithreading</i> molti a uno.	18
16	Il modello di <i>multithreading</i> uno a uno.	19
17	Il modello di <i>multithreading</i> molti a molti.	19
18	Rappresentazione grafica del modello fork-join per la risoluzione di un task.	20
19	Lo scheduler entra in gioco nel passaggio da ready a running.	23
20	Diagramma di <i>Gantt</i> dell'algoritmo FCFS.	24
21	Diagramma di <i>Gantt</i> dell'algoritmo SJF.	25
22	Grafico che indica come viene stimato il burst di un processo.	26
23	Diagramma di <i>Gantt</i> dell'algoritmo SRTF.	27
24	Diagramma di <i>Gantt</i> dell'algoritmo RR.	28
25	Diagramma di <i>Gantt</i> dell'algoritmo basato puramente sulla priorità. .	29
26	Diagramma di <i>Gantt</i> dello scheduling con priorità unito all'algoritmo RR per i processi con lo stesso grado di urgenza.	30
27	Ci sono diverse modi per gestire n threads con n cores.	31
28	Creazione di due processi figli con lo stesso PID.	32
29	La struttura di astrazione del monitor.	39
30	Caption	40
31	Disposizione dei 5 filosofi.	42
32	Rappresentazione grafica di un grafo che presenta una situazione di deadlock.	46
33	Rappresentazione grafica di un grafo senza deadlock.	47
34	Piccolo <i>decision tree</i> utile per capire se è presente un deadlock oppure no.	47
35	Differenza tra situazione unsafe e safe.	49
36	Un grafo composto da tutti e tre i tipi di archi.	50
37	L'algoritmo nega la richiesta di T_1 per R_2	50
38	Un grafo <i>resource-allocation</i> e il suo rispettivo <i>wait-for graph</i>	53
39	La gerarchie delle memorie nel calcolatore.	55
40	Limite inferiore e limite superiore dello spazio del processo in memoria.	56
41	Procedura di controllo di un indirizzo mediante il SO.	56
42	I tre tempi di binding.	57
43	Procedura di controllo di un indirizzo mediante la MMU.	58

44	Allocazione contigua dei processi in memoria.	59
45	I registri utilizzati per il binding tra logico e fisico.	59
46	La memoria divisa in partizioni fisse per i processi.	59
47	Rappresentazione di una partizione variabile in memoria.	60
48	In quale allocazione di memoria verrà inserito il nuovo processo? . . .	60
49	Processo di compacting, una soluzione alla frammentazione esterna. .	61
50	Il funzionamento ad alto livello della paginazione.	62
51	La rimozione di un processo e l'inserimento di un altro con la pagina- zione.	63
52	Il processo di conversione da indirizzo logico a indirizzo fisico.	64
53	Esempio di una paginazione.	65
54	Il processo di allocazione delle pagine in nuovi frames liberi.	65
55	I due registri necessari per tenere traccia della page table in memoria. .	66
56	Il processo di traduzione effettuato con l'ausilio della TLB.	67
57	Implementazione del valid-invalid bit sulla tabella delle pagine.	68
58	Struttura gerarchica della page table.	69
59	L'accesso in memoria attraverso una struttura gerarchica della page table.	69
60	Accesso alla memoria tramite una page table implementata attraverso una hash table.	70
61	Rappresentazione di una <i>clustered</i> page table.	71
62	Accesso in memoria attraverso la tabella delle pagine invertita.	71
63	Processo di <i>swap-in</i> e <i>swap-out</i> dal backing store.	72
64	Rappresentazione del processo di swapping solo su determinata pa- gine di un processo.	73
65	Il funzionamento ad alto livello della segmentazione.	74
66	processo di traduzione di un indirizzo logico a fisico mediante la segment table.	74
67	Rappresentazione di un modello che combina paginazione e segmen- tazione.	75
68	Alcune pagine del processo sono in memoria, altre sono sul disco. . . .	76
69	La memoria virtuale fornisce la possibilità di condividere delle librerie. .	77
70	Illustrazione del funzionamento del demand paging nel caso di due processi, A e B.	78
71	Generazione di un page fault.	78
72	Gli steps necessari per prelevare la pagina dallo storage secondario . .	79
73	Dato che P1 deve scrivere sulla pagina C, questa viene copiata.	80
74	Funzionamento generico di un algoritmo di rimpiazzo.	81
75	Comportamento dell'algoritmo di rimpiazzo FIFO.	82
76	Una piccola illustrazione dell'anomalia di Belady.	83
77	Funzionamento teorica dell'algoritmo ottimale.	83
78	Funzionamento dell'algoritmo LRU.	84
79	Il funzionamento del second-chance algorithm.	85
80	Questa tecnica assicura la presenza di frames liberi in memoria al fine di soddisfare eventuali nuove richieste.	87
81	Il fenomeno del thrashing.	88
82	Il working-set.	88
83	Il controllo della frequenza dei page fault	89
84	La relazione tra working-set e page fault rate	90

85	L'approccio di allocazione buddy system con l'allocatore <i>power-of-2-allocator</i>	90
86	La slab allocation del kernel.	91
87	Schema ad alto livello di un disco rigido.	92
88	Un blocco con delle pagine valide e delle pagine non valide.	94
89	Funzionamento dell'algoritmo FCFS.	96
90	Funzionamento dell'algoritmo SSTF.	96
91	Funzionamento dell'algoritmo SCAN.	97
92	Funzionamento dell'algoritmo C-SCAN.	97

Elenco dei codici

1	Soluzione di Peterson	34
2	Utilizzo <code>test_and_set</code>	35
3	Utilizzo di <code>compare_and_swap</code>	36
4	Esempio di una variabile atomica <code>increment()</code>	36
5	Struttura del semaforo con <i>waiting queue</i>	38
6	Utilizzo di semaforo con <i>waiting queue</i>	38
7	Struttura del monitor	39
8	Problema del buffer limitato	41
9	Risoluzione del problema dei filosofi con i semafori	42
10	Risoluzione del problema dei filosofi con i monitor	43
11	Classica situazione di deadlock	45

Parte I

Gestione dei Processi

1 Processi

Iniziamo dalle basi. Un **processo** è un programma in esecuzione, è un'istanza del programma che viene eseguita sulla CPU. Possiamo infatti avere diverse istanze dello stesso programma, ognuna che viene eseguita indipendentemente dall'altra. Possiamo quindi dire che il programma, ovvero il file eseguibile (.exe) è qualcosa di passivo mentre il processo è qualcosa di attivo.

1.1 Allocazione in memoria

Andando un po' più in dettaglio, quando il programma è in esecuzione, questo viene eseguito in maniera sequenziale. Al processo, una volta che è eseguito, viene dedicato dello spazio in memoria dal sistema operativo. Come è possibile osservare nella figura 1, la memoria messa a disposizione dal sistema operativo è suddivisa in diverse zone, ciascuna con un particolare compito. Prima di tutto, il codice sorgente

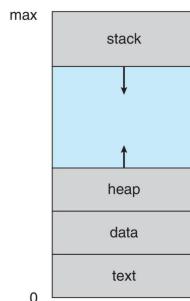


Figura 1: Spazio in memoria allocato per il processo dal sistema operativo.

del programma viene caricato nella zona **text**. Dopo di che, nella parte dedicata ai dati (**data**) vengono salvate generalmente le variabili globali, che permangono per tutta la vita del processo. Sono infine presenti due parti: lo **stack** e l'**heap** che crescono in direzione opposta. Lo stack contiene dati temporanei come variabili locali mentre l'heap è utilizzato al fine di allocare la memoria dinamicamente durante la vita del programma¹.

1.1.1 Process Control Block (PCB)

Ad ogni processo che è mandato in esecuzione è assegnata una particolare struttura dati dal sistema operativo, ovvero il *Process Control Block* (figura 2). Il PCB contiene diverse informazioni riguardanti il processo, in particolare:

1. Lo **stato** del processo;

¹Come abbiamo visto con C++, heap e *freestore* sono quasi dei sinonimi.

2. Informazioni sul **program counter**, in particolare è importante sapere se il processo è fermato temporaneamente e poi fatto ripartire più tardi;
3. Valori dei registri, utili nel caso in cui un processo venga messo in pausa;
4. Altre informazione riguardanti lo scheduling della CPU (vedi capitolo 3), come la priorità del processo;
5. Informazioni per la gestione della memoria e dell'I/O.

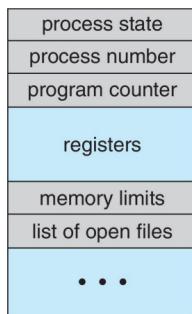


Figura 2: Rappresentazione del contenuto di un generico PCB.

In particolare, in Linux, nel PCB di un processo (che in Linux è chiamato **task**) sono presenti le seguenti informazioni: **pid** (numero assegnato al particolare processo), puntatori al processo genitore (che vedremo saranno utili nella fase di creazione di un processo), puntatori ai processi figli e altre informazioni come la lista dei file aperti. Quando un nuovo processo è creato in Linux, le sue informazioni sono detenute in una lista concatenata (*doubly-linked list*) dove ogni nodo della lista è il PCB di un processo specifico (figura 3). Al fine di andare a modificare delle informazioni del

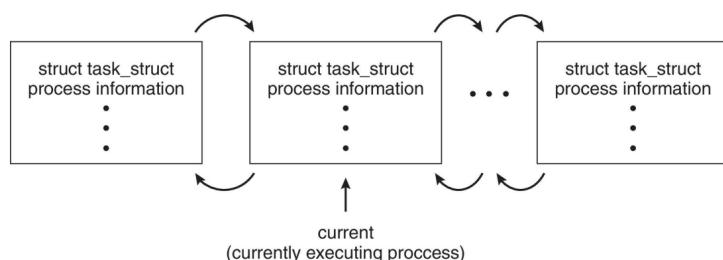


Figura 3: Lista concatenata che mantiene tutti i PCB dei processi (task) in Linux.

processo (come lo stato corrente) il sistema operativo scorre la lista e, dopo aver selezionato il PCB del processo desiderato, andrà a modificare il campo.

1.2 Stati di un processo

Durante l'intera vita del processo, questo passa in diversi stati (figura 4). I principali sono:

1. **New**: il processo è appena stato creato;

2. **Ready**: il processo è pronto per essere eseguito, quindi non è ancora stato assegnato al processore e sta aspettando l'assegnazione;
3. **Running**: dopo essere stato associato alla CPU il processo inizia ad essere eseguito. Come vedremo nel capitolo 3 e successivi, il processo può essere interrotto e quindi ritorna allo stato ready;
4. **Waiting**: se il processo deve aspettare qualche input da esterno si mette in attesa e una volta che riceve l'input ritorna nello stato ready;
5. **Terminated**: una volta che il processo finisce la sua esecuzione, questo ovviamente termina e viene rimosso dalla CPU.

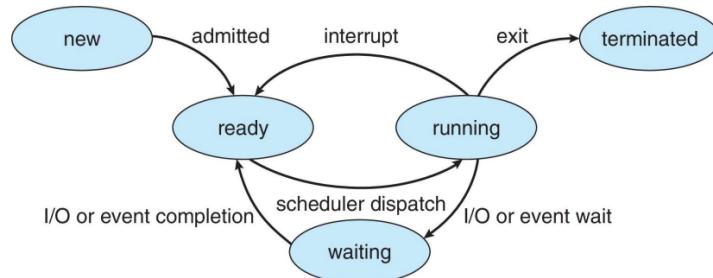


Figura 4: Gli stati della vita di un processo.

Lo stato *ready* e lo stato *wait* contengono delle code dove i processi attendono di essere eseguiti ovvero la **ready queue** e la **wait queue** che non sono altro che delle liste concatenate. Le liste contengono i PCB dei processi: il sistema operativo tiene traccia del primo e dell'ultimo processo nella lista al fine di riuscire a implementare le due code. Possiamo inoltre suddividere i processi in due macro categorie:

- ◊ **CPU bound** che sono i processi che hanno un uso massiccio della CPU;
- ◊ **I/O bound**, ovvero processi che spendono la maggior parte del loro tempo in una situazione di wait per leggere o scrivere sulle periferiche.

1.2.1 Context switch

Quando un processo A viene rimosso dalla CPU, nel caso in cui sia stato interrotto per far spazio ad un altro processo B, è necessario salvare l'informazione del processo A in modo tale da poterlo sostituire con il processo B per poi, in un secondo momento, riuscire a ricaricare il processo A. Questa operazione è detta **context switch** (figura 5) e viene effettuata in pochi microsecondi. Ciò nonostante se è effettuata in maniera molto frequente durante l'esecuzione di diversi processi può causare molto spreco di tempo: il context switch può quindi generare un **overhead** che va sicuramente preso in considerazione negli algoritmi di scheduling (capitolo 3). È importante notare che il context switch tipicamente richiede anche un aggiuntivo utilizzo della memoria, che andremo a discutere nel capitolo 6 quando discuteremo di *paginazione* e *swapping*.

1.2.2 Creazione di un processo

Al fine di creare un processo ce ne deve sempre essere uno iniziale (*parent*) che genera il nuovo (*child*). Ogni nuovo processo ha un identificativo, il **pid**, che distingue

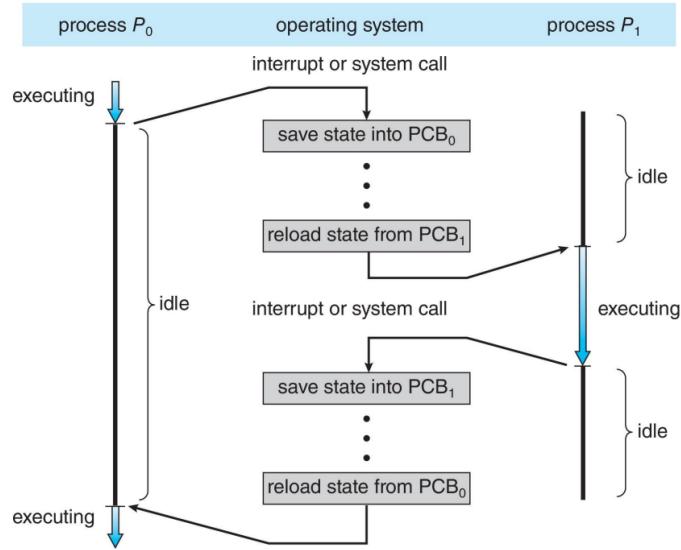


Figura 5: Il context switch.

univocamente il processo creato. Al momento della creazione è possibile specificare alcune opzioni al fine di creare il processo child in un determinato modo. Prima tra tutte è l'opzione di **condivisione di risorse**, dove si può specificare se il figlio condivide le stesse risorse del genitore, un sottoinsieme oppure si può specificare che il figlio non condivide alcuna risorsa con il *parent*. Inoltre si possono specificare le opzioni di **esecuzione**: si specifica se il figlio e il genitore possano essere eseguiti in maniera concorrente oppure se il *parent* deve aspettare il termine dell'esecuzione del *child*. Infine si può anche specificare lo **spazio degli indirizzi**, in particolare si sceglie se il figlio crea una copia identica della memoria utilizzata dal genitore oppure se carica un programma completamente nuovo.

Vediamo ora un esempio di creazione di un processo in UNIX (figura 6). Questo sistema operativo fornisce tre particolari *system calls*:

- ◊ **fork()**: questa *system call* non fa altro che creare un processo. Il processo parent, dopo aver chiamato la funzione `fork()` viene duplicato. La funzione inoltre ritorna un valore intero, se questo valore è maggiore di zero vuol dire che ci troviamo all'interno del processo genitore; se invece il valore è zero vuol dire che il codice eseguito è all'interno del child. L'unico modo per distinguere se il processo è parent o child è attraverso il valore di ritorno di `fork()`.

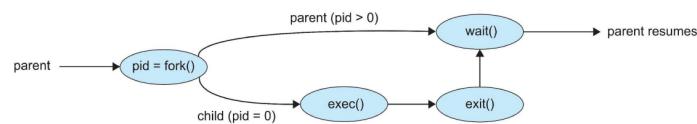


Figura 6: Rappresentazione delle 3 *system calls* fondamentali.

- `exec()`: è una funzione utilizzata dal processo figlio nel caso in cui è necessario far partire un processo completamente diverso dal parent;
- `wait()`: è una system call utilizzata dal genitore al fine di aspettare il termine dell'esecuzione del figlio.

1.2.3 L'albero dei processi in Linux

Come fa il sistema operativo a generare tutti i processi di cui ha bisogno? Ci deve sempre essere un processo iniziale, un programma all'inizio da cui tutti si genera. In particolare in Linux il primo processo da cui tutto è generato è chiamato `systemd` ed è il processo padre di tutti gli altri processi, quello il quale pid vale 1. Da `systemd`, *forkando* processo dopo processo vengono generati tutti i processi necessari all'avvio del sistema, come il terminale, generando quindi un albero (figura 7).

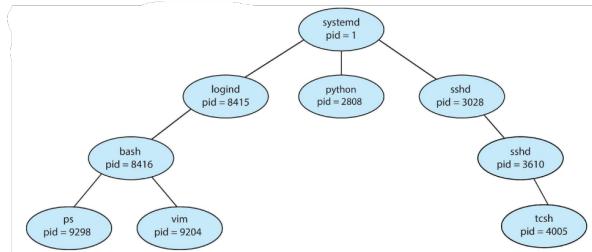


Figura 7: L'albero dei processi generato da `systemd`.

1.2.4 Terminazione di un processo

Naturalmente, un processo può anche terminare. La terminazione del processo può essere spontanea (attraverso la *system call exit*) e quindi il processo viene deallocated dal sistema operativo, oppure il processo può essere terminato dal genitore attraverso la *system call abort*. Questo di solito avviene quando il processo figlio supera il limite delle risorse allocate, quando la task che sta completando non è più richiesta oppure nel momento in cui il genitore termina e di conseguenza il sistema operativo termina anche i figli.

Come abbiamo visto in precedenza, esiste una funzione (`wait()`), che serve per evitare che il processo parent termini prima del processo child: la funzione infatti obbliga il parent ad aspettare che il child termini. Inoltre, se al termine di un processo child non c'è nessun processo parent che stava aspettando il termine del child, ci troviamo davanti ad un **processo zombie**. Infine, se il processo parent termina senza aspettare la terminazione del child, quest'ultimo è chiamato **orfano** e verrà terminato dal sistema operativo.

1.3 Comunicazione tra processi (IPC)

Passiamo ora a discutere i diversi modi per comunicare tra diversi processi. In alcuni casi avremo a che fare con processi indipendenti, altre volte invece necessiteremo di

processi **cooperanti**. Per questi ultimi è necessario un modello di *Inter-Process Communication*, chiamata anche **IPC**. In questo paragrafo ci occuperemo di due modelli: comunicazione tramite memoria condivisa e tramite il passaggio di messaggi.

1.3.1 Memoria condivisa

Il primo modello di cui ci occuperemo è la tecnica di memoria condivisa. In questo modello, come possiamo notare anche dalla figura 8, l'unica cosa di cui si fa carico il sistema operativo è l'assegnazione di una memoria che è condivisa tra i processi A e B. Ciò significa che la comunicazione è molto veloce tra i due processi in quanto non

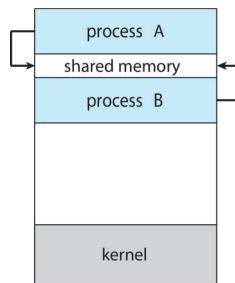


Figura 8: Il modello di memoria condivisa per IPC.

c'è nessun intermediario tra i due. Allo stesso tempo però è più facile che si generino errori come la sovrascrittura di valori e in genere problemi di sincronizzazione con i dati (come la *race condition*, vedi capitolo 4) in quanto il sistema operativo non ha alcun tipo di controllo sulla memoria secondaria.

Partiamo ora da un esempio di questo modello al fine di ottenere informazioni più dettagliate: stiamo infatti parlando del problema **Producer - Consumer**. Ipotizziamo quindi che due processi abbiano dello spazio in memoria condiviso; la comunicazione attraverso questa memoria può avvenire in due modi:

- ◊ **unbounded**, ovvero che il produttore continua a generare dati da mettere nell'area condivisa, e che il consumatore continua ad utilizzare quei dati fino a che non finiscono (al più attende la creazione di altri dati).
- ◊ **bounded**, dove si ha un **buffer** che entrambi devono aspettare: il consumer attende che ci siano dati nel buffer e il producer aspetta nel momento in cui il buffer è pieno.

1.3.2 Passaggio di messaggi

In questo secondo caso invece il sistema operativo si prende carico di gestire la coda dei messaggi (**message queue**) che vengono scambiati tra i due processi (figura 9). In questo caso è il kernel che fa da intermediario tra i due e di conseguenza la velocità di comunicazione sarà ridotta dall'**overhead**. Allo stesso tempo però il kernel garantisce la sincronizzazione e la correttezza tra i messaggi scambiati e di conseguenza è un modello più sicuro.

Discutiamo ora più dettagliatamente questo modello. In particolare, sono fornite due operazioni fondamentali: `send(message)` e `receive(message)`. Ciò nonostante la comunicazione di tali messaggi fa sorgere diversi dubbi e domande legate alla

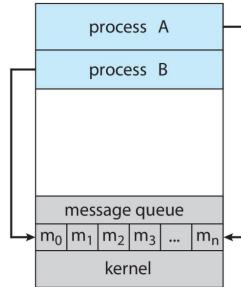


Figura 9: Il modello di memoria condivisa per IPC.

progettazione: come sono stabilite le connessioni? qual è la sua capacità? è unidirezionale o bidirezionale? la dimensione del messaggio è fissa o variabile? un collegamento è associato solo a 2 processi o a più? Per rispondere a queste domande andiamo a vedere l'implementazione di questo modello che può essere di due tipi.

Quando si parla di comunicazione **diretta**, si intende che i processi specifichino esplicitamente il destinatario del messaggio: `send(Q, message)` e `receive(P, message)`. Nella comunicazione **indiretta** invece si ha a che fare con un **buffer**, chiamato anche porta, il quale ha un ID unico tra gli altri. In questa implementazione due processi si possono parlare solo se condividono questo buffer (**mailbox**). Se più processi condividono la stessa mailbox si ha quindi un link che collega diversi processi, inoltre attraverso questa porta il collegamento è bidirezionale in quanto un processo può sia spedire un messaggio che riceverlo. Con la comunicazione indiretta le primitive però sono diverse: una volta creata la porta (chiamiamola A, per comodità), al fine di scambiare i messaggi è necessario utilizzare le operazioni `send(A, message)` e `receive(A, message)`.

2 Threads

Partiamo con il distinguere un thread da un processo. Il thread è un **filo di esecuzione**: in un processo ci possono essere diversi thread i quali possono avere dei diversi *pattern* per ciascuna esecuzione.

Perché?

I thread sono entità più semplici rispetto ai processi e sono quindi più facili da gestire. Grazie ai thread si ottengono diversi vantaggi:

- ◊ Il sistema è più **recettivo**;
- ◊ Dato che le risorse sono condivise è più facile gestirle;
- ◊ Richiede meno risorse rispetto alla creazione di un processo;
- ◊ Può utilizzare tutti i *core* messi a disposizione dal sistema (*multicore programming*).

Per esempio, al posto di far eseguire 4 processi differenti è molto meglio eseguire un processo con 4 thread differenti: in questo modo si evita di allocare in memoria 4 volte le stesse risorse che, grazie all'utilizzo dei thread, sono allocate solo una volta.

2.1 Concorrenza e parallelismo

Quando parliamo di **multicore programming** è necessario fare una netta distinzione tra il significato di concorrenza e parallelismo. Con **parallelismo** si intende che un sistema è in grado di preformare più di un compito in maniera simultanea (tipico dei sistemi multicore). Con **concorrenza** si intende la possibilità di far progredire più di un compito (non in maniera simultanea).

Come possiamo vedere della figura 10, quando parliamo di concorrenza ci riferiamo ad un singolo core che esegue a frammenti più thread diversi.

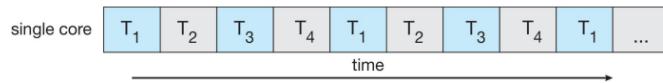


Figura 10: Esempio di concorrenza tra 4 processi.

Quando invece facciamo riferimento al parallelismo (figura 11) indichiamo la capacità del sistema di effettivamente riuscire ad eseguire parallelamente diversi thread. Si osserva che le due pratiche non sono esclusive: notiamo che il core 1 esegue

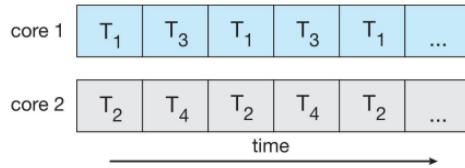


Figura 11: Esempio di parallelismo in un sistema dual core.

in maniera concorrente T_1 e T_3 , mentre il core 2 esegue in maniera concorrente T_2 e T_4 .

2.1.1 Tipi di parallelismo

Possiamo dividere i parallelismi in due tipi. Il primo, chiamato **data parallelism** (figura 12) implica un sottoinsieme preciso di dati sia distribuito per ogni core. In altre parole, avendo a che fare un un largo database, lo si suddivide in parti e ciascuna viene assegnata ad un core. Tale core potrà operare solo in quella porzione di dati

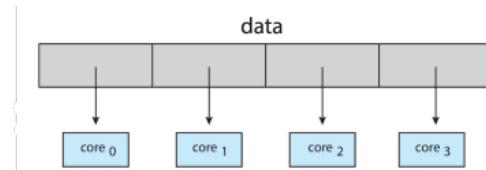


Figura 12: Esempio di parallelismo di dati.

Il secondo tipo di parallelismo è detto **task parallelism**, rappresentato in figura 13. Questo tipo di parallelismo concede la memoria condivisa a ciascun core solo che

ogni core ha un compito ben preciso: un core sarà ottimizzato per la scrittura, un altro sarà più veloce in lettura e così via.

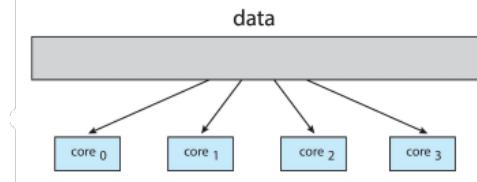


Figura 13: Esempio di parallelismo di compiti.

2.1.2 Legge di *Amdahl*

La legge di *Amdahl* è una funzione che mette in relazione due variabili importanti:

1. S , ovvero la percentuale di codice che non può essere parallelizzato, ovvero codice **seriale** (di conseguenza il numero di codice che può essere parallelizzato è $1 - S$);
2. N , che rappresenta il numero di core disponibili.

Con questi due dati abbiamo la possibilità di calcolare lo *speedup* attraverso la seguente formula:

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

Osservando la formula osserviamo che se se la percentuale di codice seriale tende a zero e il numero di core tende a infinito, lo *speedup* sarebbe infinito. Questa però è una situazione utopica: non esistono casi in cui si è privi di codice seriale.

Osserviamo quindi il seguente grafico (figura 14) che mostra lo *speedup* all'aumentare del numero di core, essendo a conoscenza della percentuale di codice seriale. Notiamo che quando la percentuale di codice seriale è circa la metà non ha impor-

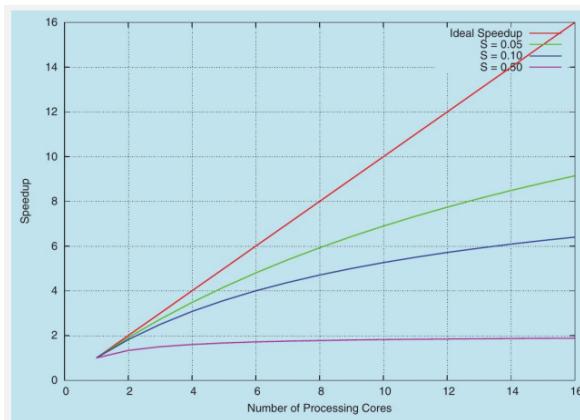


Figura 14: Il grafico che descrive lo *speedup* a seconda della percentuale di codice seriale.

tanza il numero di core del sistema: lo speedup rimarrà pressoché invariato. Anche solo con il 5% di codice seriale notiamo un forte abbassamento rispetto allo speedup ideale. È evidente quindi che l'aumento di core non causa l'aumento di speedup, soprattutto in una situazione dove il codice seriale è molto.

2.2 Modelli multithreading

È importante fare una distinzione tra due classi principali di thread: **user threads** e **kernel threads**. La principale differenza tra i due è che i kernel threads hanno molti più privilegi rispetto a quelli utente. Esistono quindi dei modelli che cercano di associare agli user threads i kernel threads al fine di sfruttare al meglio il principio di *multithreading*.

2.2.1 Many-to-One

In questa prima architettura, il kernel mette a disposizione solo un thread che è collegato e deve soddisfare tutte le richieste di tutti gli user threads (figura 15). È

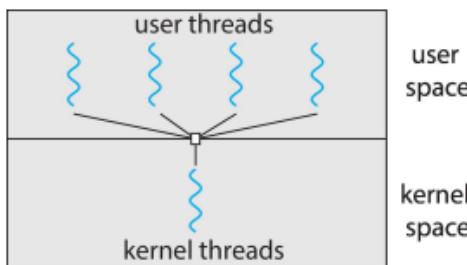


Figura 15: Il modello di *multithreading* molti a uno.

evidente che l'efficienza di questo modello non è il suo forte: sono presenti diversi thread utente ai quali deve rispondere solamente un thread del kernel. Basta pesare al fatto che se il kernel thread è in attesa di un input esterno, tutti gli user thread sono bloccati (**collo di bottiglia**). Questo modello è infatti poco utilizzato dato che non sfrutta le potenzialità del *multicore*.

2.2.2 One-to-One

In questa architettura (figura 16), quando viene creato uno user thread, il suo rispettivo kernel thread viene creato; così facendo esiste un kernel thread associato ad ogni user thread. Ad ogni modo ci possono essere alcune restrizioni in modo da evitare la creazione di troppi kernel threads (e quindi limitare la creazione di user threads). È comunque evidente che questo modello è sicuramente più **efficace** del modello precedente in quanto fornisce la possibilità di sfruttare a pieno un sistema multicore.

2.2.3 Many-to-Many

L'ultimo modello è un buon compromesso tra il modello *Many-to-One* e il modello *One-to-One*. Il modello molti a molti (figura 17) fornisce diversi vantaggi ed è più

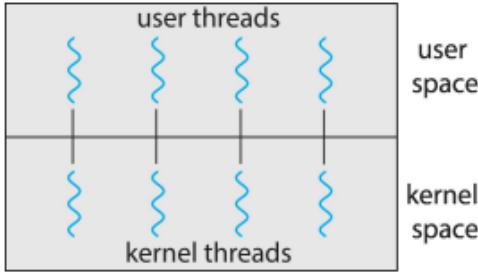


Figura 16: Il modello di *multithreading* uno a uno.

flessibile rispetto ai primi due. Non esiste infatti la corrispondenza univoca, generalmente si hanno più user threads che fanno riferimento ad alcuni kernel threads (è come se una sala da 20 clienti fosse gestita da 5 camerieri). È possibile infatti

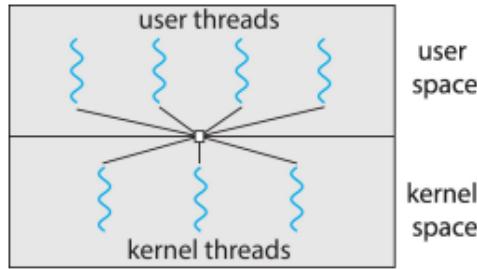


Figura 17: Il modello di *multithreading* molti a molti.

controllare in maniera più efficace i kernel threads e questo rende il modello *Many-to-Many* molto **robusto**. Ad ogni modo la sua implementazione è molto più complessa rispetto ai modelli precedenti.

Per questo, a volte, si fa riferimento ad un *ibrido* tra il modello *Many-to-Many* e il modello *One-to-One*: stiamo parlando del **Two-level model** che consente sia la corrispondenza *1 a 1* che la corrispondenza *N a M*.

2.3 Librerie di thread

In questa piccola sezione ci interessiamo alle librerie disponibili per la creazione e la gestione di threads. Queste possono essere di due tipi:

- ◊ Librerie in **user space**, dove i thread sono gestiti completamente a livello utente (come nel caso di Pthreads);
- ◊ Librerie di tipo **kernel-level** che sono supportate dal sistema operativo e si appoggiano al kernel attraverso le *system calls*; questo comporta un livello maggiore di complessità nel kernel ma il programmatore ha meno da implementare.

Spendiamo due parole su **POSIX Threads**. Questa, non è propriamente una libreria ma è un insieme di specifiche (non implementazioni!) che aiuta con la creazione e

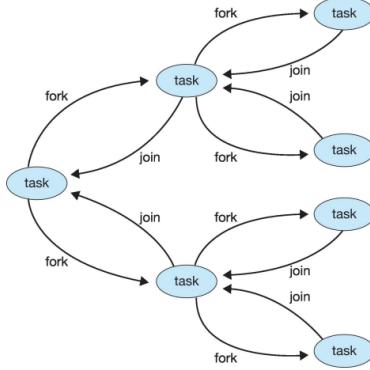


Figura 18: Rappresentazione grafica del modello fork-join per la risoluzione di un task.

la gestione di thread. POSIX Threads fornisce specifiche sia a livello di utente che a livello di kernel.

2.4 Threading implicito

Cerchiamo ora di cambiare approccio: attraverso le librerie l'approccio era esplicito, stava al programmatore l'implementazione e/o la gestione dei thread. Nel momento in cui si fa riferimento a più threads, diventa sempre più difficile controllare la correttezza del codice. Ecco che si è pensato ad un altro approccio: il **threading implicito**. Con questo approccio i thread sono gestiti maggiormente dal compilatore oppure da librerie *run-time* le quali si occupano di creare e gestire i thread. In questa sezione vedremo alcuni dei metodi utilizzati per ottenere il threading implicito.

2.4.1 Modello fork-join

Questo tipo di modello si ispira alla creazione di processi (paragrafo 1.2.2). In questo modello, diversi threads sono divisi e, una volta che sono terminati, vengono uniti (figura 18). La divisione dei threads è una scelta presa completamente dalla libreria. Come possiamo notare dalla figura 18, in questo caso, la libreria sceglie un task da far risolvere ad un thread dove, eventualmente verrà spartita in altri due thread (che vengono appositamente creati) e così via fino a che il task non sia completamente risolto.

2.4.2 Thread pools e OpenMP

Altri due modelli molto importanti sono *thread pools* e OpenMP. Nel primo caso la libreria mette a disposizione un numero di thread (*pool* di thread) che attendono un task da risolvere. In questo modo, dato che i thread sono già pronti, non si spreca tempo per l'effettivo processo di creazione. Inoltre il problema della limitazione dei thread è risolto in quanto una volta creati quelli per la *pool*, non vengono creati altri thread.

Quando parliamo di OpenMP invece parliamo di una serie di **direttive** per il compilatore e un insieme di librerie per C, C++ e FORTRAN. Questo modello fornisce tutte le risorse per la condivisione della memoria tra i thread e i parallelismi. Essendo delle direttive, queste devono essere proprio scritte nel codice; per esempio: `#pragma omp parallel`.

2.5 Problematiche

Come vedremo in questo paragrafo, anche i thread portano a delle problematiche che devono essere gestite come, per esempio, l'interruzione di tali e il modo di implementare l'eliminazione di un thread.

2.5.1 Semantica exec e fork

Il primo dubbio che è necessario chiarire è il comportamento durante la chiamata della funzione `exec()`: se si fa una chiamata alla funzione `exec()`, vengono rimpiazzati tutti i thread del processo oppure solo il thread su cui è stata chiamata `exec?` Generalmente la funzione `exec` cancella il processo in esecuzione e, di conseguenza, tutti i suoi thread.

Il secondo dubbio, riguarda la funzione `fork()`: se si fa una chiamata a `fork()` ad un processo multithread, si effettua una copia a tutti i thread del processo oppure solo al thread su cui è stata chiamata la funzione? La risposta a questa domanda è che dipende dall'obiettivo della funzione `fork()`:

- ◊ Se la funzione deve cambiare subito il codice attraverso `exec()`, non vale la pena copiare tutti i thread se tanto si che verranno eliminati.
- ◊ Se invece il nuovo processo deve supportare anch'esso il multithreading, allora ha senso effettuare una coppia di tutti i thread e non solo del thread su cui è stata chiamata la funzione

2.5.2 Segnalazione ed eliminazione

I segnali sono usati per notificare un processo che un determinato evento è accaduto. Tali segnali però debbono essere gestiti: ecco che emerge la figura del **signal handler** che può essere di **default** oppure **user-defined**, ovvero definito dall'utente. Generalmente ogni segnale ha il suo specifico *default handler* che è utilizzato anche dal kernel. Cosa succede però nel caso in cui il sistema è multi-threading? Ci sono diverse opzioni:

1. Spedire il segnale al thread ad esso compatibile;
2. Propagare il segnale ad ogni thread del processo;
3. Mandare il segnale ad dei thread specifici del processo;
4. Assegnare ad uno specifico thread il compito di ricevere i segnali degli altri thread.

È inoltre importante riuscire a gestire la **cancellazione** dei thread. Questi però devono attivare la possibilità di essere cancellati: in altre parole, se un thread ha la cancellazione disabilitata non potrà essere cancellato fino a che non viene riattivata. Nel momento in cui la cancellazione è attivata, il thread può essere cancellato attraverso due tecniche:

- ◊ **Asincrona**, ovvero il thread termina immediatamente;
- ◊ **Deferred** che significa *posticipata*. Può ritornare utile nel momento in cui il thread sta compiendo un'operazione delicata (chiusura di un file) e non ha la possibilità di terminare immediatamente.

3 CPU Scheduling

In questa sezione ci occupiamo di tutti gli aspetti che concernono lo scheduling del processore, ovvero la pratica secondo la quale, attraverso dei precisi criteri, si sceglie quale processo all'interno della *ready queue* verrà eseguito.

3.1 Nozioni fondamentali

Prima di iniziare a discutere di scheduling vero e proprio è necessario avere chiari alcuni concetti importanti.

Burst. La prima è la nozione di burst. Chiamiamo **CPU burst** il periodo di tempo nel quale un processo esegue operazioni all'interno della CPU; diversamente, l'**I/O burst** è il tempo che il processo spende interfacciandosi con le periferiche di input e output. In particolare, un processo che occupa per molto tempo la CPU si dice *CPU bounded*, mentre nel caso di I/O si parla di un processo *I/O bounded*. In questo capitolo ci occupiamo solo di CPU burst.

CPU scheduler. Il CPU scheduler è il responsabile dell'organizzazione e dell'ordinamento della **coda dei processi** (figura 19). Le decisioni su quale processo deve essere eseguito prima stanno a lui. Questo tipo di decisioni avvengono generalmente quando un processo:

1. Passa dallo stato *running* a *waiting*;
2. Passa da *running* allo stato *ready*;
3. Passa dallo stato *waiting* allo stato *ready*;
4. Termina.

Osserviamo che per i punti 1 e 4 non è presente una vera e propria decisione: un nuovo processo deve essere messo in esecuzione. Questo non vale per i punti 2 e 3 dove si fa una decisione.

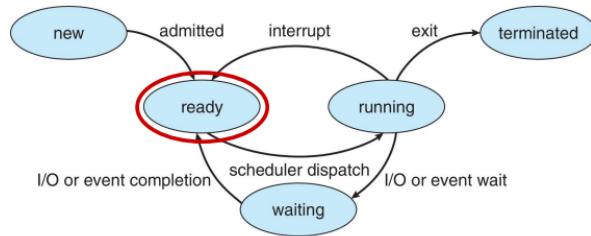


Figura 19: Lo scheduler entra in gioco nel passaggio da ready a running.

Preemption. Ci sono due macro gruppi di scheduling: preemptive e non. Uno scheduling è detto **non preemptive** nel momento in cui un processo non può essere fermato. In altre parole, quando la CPU è assegnata ad un processo, questo la utilizza fino a che non è terminato. Differentemente, uno scheduling è detto **preemptive** quando un processo può essere fermato per dare precedenza ad un altro per poi essere

fatto ripartire: questo tipo di scheduling è sicuramente più performante e moderno; è infatti utilizzato nei sistemi operativi più diffusi come Windows, MacOS, Linux e altri. Ciò porta comunque a delle situazioni indesiderate come le **race conditions**: poniamo il caso di avere due processi che condividono dei dati; immaginiamo che il primo processo stia aggiornando questi dati ma allo stesso tempo il secondo processo li stia utilizzando. Questo è un problema dato che il processo due sta utilizzando dei dati che non sono consistenti dato che sono in fase di aggiornamento dal processo uno. Come vedremo nel capitolo 4, questo è un problema di sincronizzazione che va risolto.

Dispatcher. Nel momento in cui lo scheduler ha scelto quale processo verrà eseguito, il dispatcher si occupa di cambiare il processo nella CPU. In particolare viene effettuato il *context switch* (1.2.1), passa in *user mode* e va alla giusta locazione del programma per iniziare la sua esecuzione. Durante tutto ciò la CPU però non lavora: è importante quindi minimizzare questa latenza e fare in modo che non se ne effettui un numero troppo elevato al fine di mantenere un alta percentuale di utilizzo della CPU.

3.2 Algoritmi non preemptive

In questo paragrafo ci occupiamo dei primi algoritmi non preemptive, ovvero gli algoritmi che non fermano i processi che sono in esecuzione.

3.2.1 First-Come First-Served (FCFS)

Questo è l'algoritmo più banale da implementare: il primo processo che entra nella coda sarà anche il primo ad essere eseguito. Questo algoritmo ha un approccio praticamente identico al **FIFO** (*First In First Out*). Attraverso un semplice esempio, possiamo osservare che questo algoritmo non è efficiente. Poniamo che entrino nella coda 3 processi: P_1 , di durata 24 unità di tempo (in genere millisecondi) e P_2 e P_3 con durata di esecuzione 3. Osservando la figura 20 è banale notare che se P_1 fosse

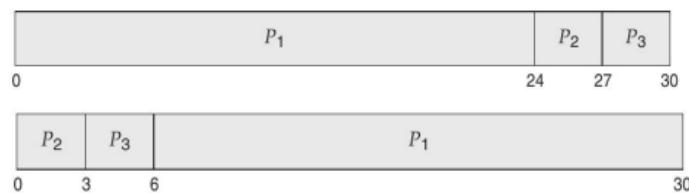


Figura 20: Diagramma di *Gantt* dell'algoritmo FCFS.

arrivato in coda per ultimo, i processi P_2 e P_3 avrebbero aspettato meno. Possiamo dimostrarlo anche in maniera più matematica, attraverso dei brevi calcoli. Nel primo caso P_1 ha aspettato 0 prima di essere eseguito, P_2 ha aspettato l'esecuzione di $P_1 = 24$ mentre P_3 ha aspettato l'esecuzione di $P_1 + P_2 = 24 + 3 = 27$. Se provassimo a fare una media del tempo di attesa otteniamo:

$$\langle T \rangle = \frac{0 + 24 + 27}{3} = 17$$

Nel secondo caso invece la situazione migliora notevolmente in quanto P_2 aspetta 0, P_3 aspetta solamente l'esecuzione di $P_2 = 3$ e infine P_1 attende l'esecuzione di $P_2 + P_3 = 3 + 3 = 6$. Attraverso la stessa espressione matematica otteniamo che l'attesa media in questo caso diventa:

$$\langle T \rangle = \frac{6 + 0 + 3}{3} = 3$$

Diversi sono i problemi di questo algoritmo. Prima di tutto può generare il **convoy effect** (effetto convoglio): se viene eseguito un processo *I/O bounded*, tutti gli altri processi in coda devono aspettare che questo si "sblocchi" generando quindi un rallentamento generale. In secondo luogo, questo algoritmo di scheduling dipende dall'ordine di entrata dei processi e di conseguenza **non** è nemmeno possibili analizzare in modo **deterministico** le prestazioni dell'algoritmo.

3.2.2 Shortest-Job-First (SJF)

Come abbiamo notato dalla figura 20, se i processi più bervi sono eseguiti prima, il tempo medio di attesa $\langle T \rangle$ si abbassa. Implementiamo quindi un algoritmo che dia la precedenza ai processi con il tempo di esecuzione più breve tra quelli che sono presenti in coda. In questo algoritmo stiamo assumendo che siamo a conoscenza del CPU burst time di ciascun processo: si osserva che stiamo facendo un'ipotesi, spesso questo dato non è a disposizione. Se tutti i CPU burst sono conosciuti, l'algoritmo fornisce il minor tempo medio di attesa di un insieme finito di processi.

Il funzionamento di questo algoritmo è molto semplice: in base ai processi arrivati in coda, questi, prima di essere eseguiti, vengono riordinati in base al loro tempo di burst. Per esempio, poniamo di avere in coda un processo P_1 con un burst time di 6, P_2 da 8, P_3 da 7 e P_4 da 3. Osservando la figura 21 notiamo che i processi

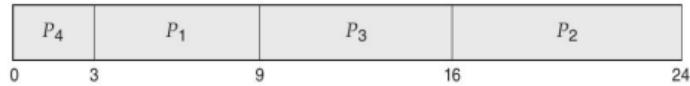


Figura 21: Diagramma di Gantt dell'algoritmo SJF.

sono stati riordinati in modo tale da minimizzare il tempo medio d'attesa. In questo esempio il processo P_4 attende 0, il processo P_1 attende l'esecuzione di $P_4 = 3$, il processo P_3 aspetta la conclusione di $P_4 + P_1 = 3 + 6 = 9$ e infine il processo P_2 aspetta $P_4 + P_1 + P_3 = 3 + 6 + 7 = 16$. Ecco che il tempo di attesa medio diventa:

$$\langle T \rangle = \frac{3 + 16 + 9 + 0}{4} = 7$$

Anche in questo caso però se durante l'esecuzione è in coda un processo con un burst molto alto e entrano solo processi con un burst basso, è probabile che si verifichi una situazione di attesa perenne (chiamata **starvation**); vedremo, nel corso di questo capitolo, come ciò può essere evitato (3.4).

3.2.3 Stima del CPU burst time

Come abbiamo affermato poco fa, quasi mai il burst time è a disposizione. Si è quindi trovato un modo per stimare al meglio il burst time di un processo, in base ai processi

che sono stati eseguiti in precedenza. Il metodo utilizzato è chiamato **exponential averaging** il quale, in essenza, dà peso maggiore ai processi eseguiti da poco tempo e, pian piano, più i processi sono remoti, meno influenza hanno sulla stima. Le variabili in gioco nella formula sono:

- ◊ t_i indica la durata del CPU burst del processo i -esimo, dove $i \in [0, n]$, dove n indica il numero di processi;
- ◊ τ_{n+1} rappresenta la stima, la predizione (*guess*), che si calcola sul processo che si sta per eseguire;
- ◊ α che è un coefficiente che indica quanto pesa la storia dei processi. Quando questo coefficiente è basso, la storia recente non conta, mentre quando è alto, la storia recente ha più peso (con $\alpha = 1$ si ha che solo l'ultimo processo influisce sulla stima). Generalmente si utilizza il valore $\alpha = \frac{1}{2}$.

Nel caso generale, con n processi si ha che per stimare il burts dell' $n + 1$ -esimo:

$$\begin{aligned}\tau_{n+1} = & \alpha t_n + (1 - \alpha) \alpha t_{n-1} + \dots \\ & +(1 - \alpha)^2 \alpha t_{n-2} + \dots \\ & +(1 - \alpha)^{n+1} \tau_0\end{aligned}$$

Osserviamo ora la figura 22 che rappresenta come viene effettuata la *guess* in base alla storia dei processi ($\alpha = .5$). La prima colonna composta dalla coppia $(^6_{10})$ è la

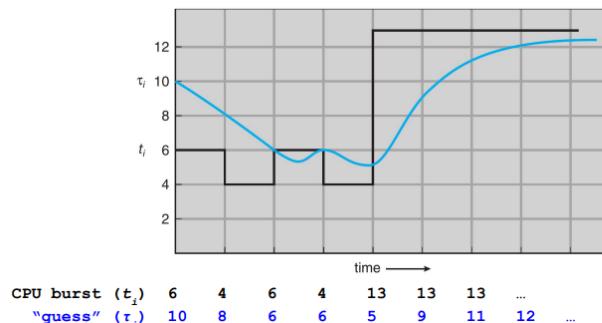


Figura 22: Grafico che indica come viene stimato il burst di un processo.

colonna "base", la partenza del nostro grafico. Con questi due dati, si calcola la seconda colonna $(^4_8)$, dove $8 = \frac{10+6}{2}$; a questo punto si può calcolare la terza colonna $(^6_6)$, dove il secondo $6 = \frac{4+8}{2}$. Così facendo si è in grado di calcolare una buona stima per il CPU burst time del processo corrente.

3.3 Algoritmi preemptive

In questo secondo paragrafo discutiamo invece di due algoritmi preemptive, ovvero algoritmi che possono fermare l'esecuzione di un processo per favorirne un altro.

3.3.1 Shortest-Remaining-Time-First (SRTF)

Il primo algoritmo che andremo a discutere è la versione preemptive del SJF: in questo caso viene servito per primo il processo al quale manca minor tempo per essere

completato. Quindi se si sta eseguendo un processo P e arriva un processo Q il quale burst time è minore rispetto al burst time che manca a P per terminare, quest'ultimo viene fermato per dare la precedenza a Q . Una volta terminata l'esecuzione di Q , il processo P riprende da dove era stato fermato in precedenza. Prendiamo in considerazione i seguenti 4 processi:

Processo	Tempo di arrivo	Stima burst time
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Osservando la figura 23, cerchiamo di capire come si comporta questo algoritmo nel momento in cui i 4 processi sono inseriti all'interno della coda. Al tempo zero

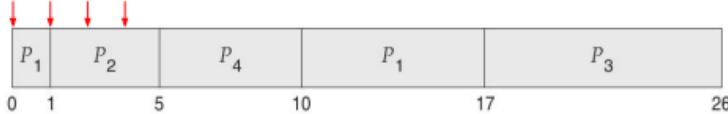


Figura 23: Diagramma di Gantt dell'algoritmo SRTF.

arriva in coda il processo P_1 , di durata 8. Al tempo 1 arriva in coda il processo P_2 che ha una durata di 4. A P_1 rimangono ancora $8 - 1 = 7$ unità di tempo prima di terminare, mentre a P_2 ne servono solo 4. P_1 viene quindi fermato e P_2 inizia la sua esecuzione (si effettua un *depatching*). Al tempo 2 e al tempo 3 sono aggiunti alla coda P_3 e P_4 i quali però hanno un burst time maggiore rispetto a P_2 che quindi termina l'esecuzione al tempo $1 + 4 = 5$. A questo punto rimangono P_1 , P_3 e P_4 . Viene eseguito P_4 in quanto il suo burst (5) è minore rispetto a quello di P_1 (7) e P_3 (9). Terminata l'esecuzione di P_4 inizia quella di P_1 e poi quella di P_3 .

Osserviamo che con questo algoritmo può capitare che sia in esecuzione un processo con un tempo di burst molto elevato e che poi continuino ad arrivare dei processi con un tempo di burst ridotto. In questo caso, il processo con il tempo maggiore verrebbe sempre interrotto dagli altri processi e non riuscirebbe mai a terminare andando quindi in una situazione di **starvation**. Come vedremo, una soluzione è questo problema è fornita dallo scheduling con priorità (3.4).

3.3.2 Round Robin (RR)

Passiamo ora ad un algoritmo un po' più sofisticato ed elegante: stiamo parlando del Round Robin. Alla base di questo algoritmo c'è il **quanto** di tempo (generalmente tra i 10 e i 100 millisecondi): ogni processo all'interno della coda, ha diritto ad essere eseguito per 1 quanto di tempo alla volta. Di conseguenza, ogni quanto di tempo viene effettuato un *context switch* e si prosegue ad un altro processo nella coda: si continua così in maniera ciclica finché ciascun processo viene eseguito completamente lasciando spazio ai nuovi.

Osserviamo che se $q \rightarrow \infty$ si ha un comportamento FIFO, molto simile all'algoritmo First-Come First-Served. Allo stesso tempo però q deve comunque essere maggiore del tempo che ci si impiega per effettuare un context switch (ordine

dei μs), altrimenti la CPU viene sprecata solo per fare i context switch al posto di effettivamente eseguire i processi.

Consideriamo il caso in cui $q = 4$ e in coda sono presenti i tre processi utilizzati nell'algoritmo FCFS:

Processo	Stima burst time
P_1	24
P_2	3
P_3	3

Il comportamento del Round Robin, in questo caso, è illustrato nella figura 24. Os-

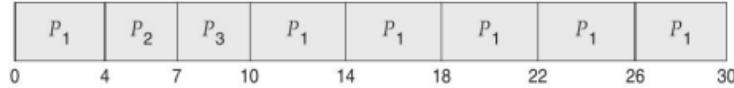


Figura 24: Diagramma di *Gantt* dell'algoritmo RR.

serviamo che il RR, per il processo P_1 , ogni $q = 4$, si ferma per fare spazio agli altri processi mentre, per i processi P_2 e P_3 , che durano meno di un quanto, quando terminano il Round Robin, non aspetta la scadenza del quanto per eseguire un altro processo ma comincia subito, che è un comportamento ragionevolmente ovvio.

3.3.3 Reattività

Cerchiamo ora di capire il vantaggio che forniscono gli algoritmi di scheduling preemptive (in particolare il RR) rispetto agli algoritmi non preemptive. Prendiamo come esempio i seguenti processi:

Processo	Burst time
P_1	6
P_2	3
P_3	1
P_4	7

Poniamo ora che il quanto di tempo q sia 4. Calcoliamo ora il turnaround time medio tra questi processi: P_1 viene eseguito per 4 unità, dopo di chè viene fermato (gliene rimangono 2) e viene eseguito P_2 che termina ($T_{P_2} = 4 + 3 = 7$); viene quindi eseguito P_3 che termina anche lui ($T_{P_3} = 7 + 1 = 8$). A questo punto inizia l'esecuzione di P_4 che viene fermato dopo 4 unità (ne rimangono ancora 3) e viene fatto ripartire P_1 che termina ($T_{P_1} = 8 + 4 + 2 = 14$) e infine viene fatto terminare anche P_4 ($T_{P_4} = 14 + 3 = 17$). Per trovare il turnaround time medio si effettua la media dei 4 turnaround trovati.

$$\langle T \rangle = \frac{T_{P_1} + T_{P_2} + T_{P_3} + T_{P_4}}{4} = \frac{14 + 7 + 8 + 17}{4} = \frac{46}{4} = 11.5$$

Osserviamo però che se avessimo utilizzato l'algoritmo SJF (vedi 3.2.2) il turnaround time medio è 8 che è minore rispetto a quello fornito da RR. Ciò significa che utilizzare un algoritmo pre-emptive, in termini di tempistiche, non è necessariamente

la scelta migliore, è semplicemente un altro modo per schedulare l'esecuzione dei processi, ma non ne garantisce il miglioramento della prestazione. Allora perchè utilizzare questi algoritmo? Perchè migliora la reattività (**responsiveness**) del sistema. Ipotizziamo di avere un coda moltissimi processi che stanno in esecuzione. Un algoritmo non preemptive li esegue uno ad uno (secondo determinati criteri, più o meno efficienti) ma tutti i processi devono sempre stare in attesa che uno termini. Il RR invece garantisce che tutti i processi vengano eseguiti per almeno un certo quanto q di tempo: è quindi un algoritmo più **equo** rispetto agli algoritmi non preemptive.

3.4 Scheduling con priorità

Fino ad ora abbiamo trattato i processi in modo equo se non per la stima del *burst time*. Introduciamo ora una seconda informazione, la **priorità** che non è altro che un numero che indica quanto sia importante (urgente) l'esecuzione di un determinato algoritmo. La priorità può essere sia legata al CPU burst time ma può anche essere legata ad altri aspetti.

Arriva però un problema: la **starvation**. Anche in questo caso, come nel SRTF, può capitare che i processi che hanno una priorità di grado molto basso non vengano mai eseguiti in quanto sono sempre presenti processi con una priorità più alta. Con l'introduzione della priorità però, se un processo è da troppo tempo in coda, si aumenta di un grado la priorità al fine da mandarlo in esecuzione. Questa tecnica rappresenta allegoricamente l'invecchiamento (**aging**) del processo nella coda di attesa.

Vediamo ora un primo esempio di scheduling con priorità puro. Abbiamo a che fare con i 5 processi seguenti:

Processo	Stima burst time	Priorità
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Osserviamo che noi consideriamo come numero più basso la priorità più alta (di conseguenza P_2 è il processo con priorità più alta e P_4 quello con priorità più bassa). Detto ciò procediamo con il diagramma di Gantt (figura 25). Notiamo infatti



Figura 25: Diagramma di *Gantt* dell'algoritmo basato puramente sulla priorità.

che i processi sono eseguiti in ordine in base alla loro priorità: P_2, P_5, P_1, P_3 e P_4 . Ovviamente, se durante l'esecuzione di P_1 (che ha priorità 3) fosse arrivato in coda un algoritmo con priorità 2, P_1 sarebbe stato interrotto per favorire l'esecuzione del nuovo processo. Inoltre, nel caso in cui due processi abbiano la stessa priorità si segue la dinamica FIFO, ovvero il primo processo che entra nella coda viene eseguito per primo rispetto ai processi con medesima priorità

3.4.1 Scheduling con priorità e RR

Cerchiamo ora di raffinare un po' di più l'algoritmo fondendo la priorità con l'algoritmo di Round Robin. In particolare, nel momento in cui si hanno più processi che hanno lo stesso livello di priorità, al posto di seguire un approccio FIFO, si utilizza il Round Robin, garantendo quindi una maggiore reattività ai processi. Partiamo dal seguente set di processi:

Processo	Stima burst time	Priorità
P_1	4	3
P_2	5	2
P_3	8	2
P_4	7	1
P_5	3	3

Come è possibile osservare dalla figura 26, notiamo che il processo 4, essendo che è l'unico processo con priorità 1, viene eseguito per primo e non viene interrotto da nessun altro processo. Dopo di che si effettua l'algoritmo RR con $q = 2$ sui processi 2

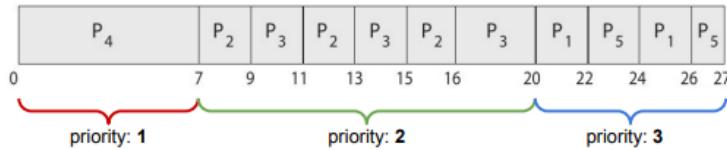


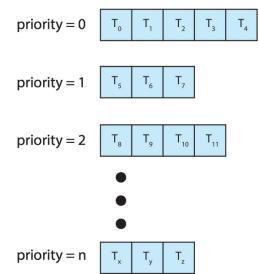
Figura 26: Diagramma di Gantt dello scheduling con priorità unito all'algoritmo RR per i processi con lo stesso grado di urgenza.

e 3 in quanto hanno la stessa priorità. Infine, si fa lo stesso procedimento che con i processi 1 e 5 che hanno priorità 3.

3.4.2 Coda multilivello

Il fatto che per ogni grado di priorità venga eseguito il Round Robin ci porta a creare uno scheduling multilivello dove ogni livello corrisponde ad un grado di priorità. Di conseguenza nell'esecuzione viene prima eseguito il primo livello attraverso il RR; dopo di che si passa al secondo livello e così via fino all'ultimo grado di priorità. In particolare, le priorità più alte sono assegnati a processi che hanno un bisogno **real-time** (come per il controllo di un braccio robotico) che poi sono seguiti dai processi di sistema etc.

Questo ci porta al caso più complesso: immaginiamo che i processi non solo debbano essere inseriti nella coda giusta ma che questi debbano anche essere in grado di sposarsi da una coda all'altra e quindi cambiando il loro grado di priorità. Stiamo infatti parlando delle **Multilevel Feedback Queue** (MFQ).



3.5 Scheduling multiprocessore

Fino ad ora abbiamo discusso di algoritmi tenendo conto del fatto che il sistema disponesse di un singolo processore; sappiamo bene però che nei sistemi moderni ormai una situazione del genere non si verifica più, con sistemi multicore.

Come possono essere gestiti diversi threads all'interno di queste architetture multiprocessore? Nel caso del **Symmetric multiprocessing (SMP)**

Finisci introduzione
CPU scheduling 2

3.5.1 Symmetric multiprocessing (SMP)

Partiamo dalla situazione più semplice, nel casi in cui abbiamo a che fare con un'architettura SMP. In questo caso infatti abbiamo *cores* che vengono gestiti in modo simmetrico (vedi paragrafo 2.1) e, disponendo di n corse, l'unico problema da risolvere è come questi possano andare a gestire n thread. Come è infatti possibile notare dalla figura 27, un modo può essere quello di utilizzare una *ready-queue* comune a tutti i cores oppure quello di creare una coda apposita ad ogni core per gestire i processi. Entrambe le soluzioni sono più che lecite anche se ad oggi i sistemi operativi

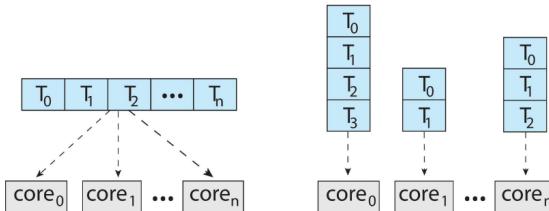


Figura 27: Ci sono diverse modi per gestire n threads con n cores.

moderni tendono ad usare la seconda in quanto la prima soluzione, essendo si che si ha a che fare con una coda condivisa è complicato gestire la condivisione di tale risorsa tra i cores (problemi di *race conditions*).

3.5.2

3.6 Scheduling real-time

3.7 Valutazione di un algoritmo

Finisci CPU scheduling:
parte 2

Parte II

Sincronizzazione dei Processi

4 Sincronizzazione

Come abbiamo visto nel capitolo 1, i processi possono essere eseguiti sia in parallelo che in concorrenza, la quale non è altro che un modo per far apparentemente girare due processi in maniera parallela quando in realtà stanno condividendo lo stesso *core* del sistema. Un processo può quindi essere interrotto in qualunque momento da un algoritmo di scheduling (come il *Round Robin*) per fare spazio ad un altro processo. Cosa succede però se viene interrotto in un momento in cui sta accedendo nella memoria condivisa con un altro processo e, lasciando spazio all'altro, vengono modificati dei dati? Capiremo, in questa sezione, che l'obiettivo da raggiungere è quindi la **cooperazione** tra processi in modo tale che non si verifichino situazioni spiacevoli.

4.1 Sezione critica

Poniamo ora di avere a che fare con due processi i quali effettuano due `fork` simultanei (figura 28). Supponiamo di avere una variabile, chiamata `next_available_pid` che

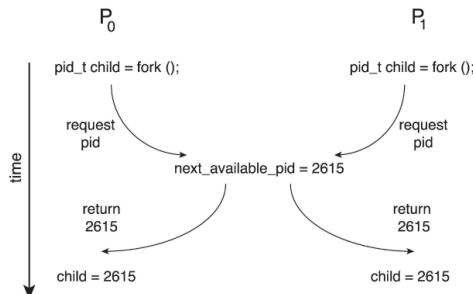


Figura 28: Creazione di due processi figli con lo stesso PID.

contiene il primo PID disponibile. Se i due processi, in questo caso P_0 e P_1 , fanno accesso alla variabile in maniera simultanea, verranno creati due figli con lo stesso PID. I due processi hanno fatto un accesso in maniera non esclusiva alla stessa zona di memoria condivisa generando quindi un problema.

Quello che abbiamo appena visto non è altro che un esempio di **critical section problem**. Generalizzando, possiamo dire che ogni processo è composto di un segmento di codice che deve avere un **accesso esclusivo** in quanto ha un accesso in scrittura dove aggiorna tabelle, modifica variabili o scrive su files. Di conseguenza si cerca di non interrompere un processo nel momento in cui questo è nella sua **sezione critica**.

4.1.1 Requisiti

È stato stilato un elenco di 3 punti di requisiti che devono essere rispettati affinché non vengano generate situazioni dove un processo viene interrotto durante la sua sezione critica.

1. **Mutua esclusione:** se il processo P sta eseguendo la sua parte critica nessun altro processo deve interromperlo;
2. **Progresso:** bisogna garantire che nessun processo rimanga in un'attesa perenne mentre aspetta la conclusione della parte critica di un altro processo. In altre parole, l'esecuzione della sezione critica di un processo non può essere posticipata in maniere indeterminata;
3. **Attesa limitata:** deve esistere un limite massimo per cui un processo deve concludere la sua parte critica.

4.1.2 Soluzioni inefficienti

Una prima idea banale è quella di **disabilitare gli interrupt** nel momento in cui un processo (un *thread*) sta eseguendo la sua parte critica. Questa però è una soluzione poco elegante e anche poco funzionale. Si rischia infatti di far aspettare molto altri processi i quali necessitano di eseguire la loro sezione critica (starvation).

Si è quindi pensato ad una soluzione un po' più elegante, ma che comunque è una forma embrionale rispetto a ciò che vedremo. Stiamo parlando di una semplice **soluzione software** tra due processi dove entrambi condividono una variabile *booleana* *turn* che indica di chi è il turno per eseguire la sezione critica.

```
1 | while(true){  
2 |     while(turn == j); /* attendo che sia il mio turno */  
3 |     /* SEZIONE CRITICA */  
4 |     turn = j; /* rimando il turno all'altro processo */  
5 |     /* sezione rimasta */  
6 | }
```

Con questa soluzione, è rispettata la richiesta di mutua esclusione ma i punti (2) e (3) non sono rispettati: il processo j potrebbe anche durare un'ora e ci sarebbe un'ora di attesa. Non sono quindi i rispettati i limiti di attesa e nemmeno il progresso dei processi.

4.2 Soluzione di Peterson

Una prima soluzione un po' più raffinata è quella di Peterson. In questa soluzione i due processi condividono due variabili:

- ◊ *turn* che è la variabile che indica di quale processo è il turno;
- ◊ *flag[2]* che è un array di due valori booleani che indicano se il processo i -esimo è pronto o meno per entrare nella sezione critica.

Come possiamo notare dal seguente segmento di codice, sono stati aggiunti alcuni controlli.

Codice 1: Soluzione di Peterson

```

1 | while (true){
2 |     flag[i] = true;
3 |     turn = j;
4 |     while (flag[j] && turn == j); /* aspetto fino a che non sono
   |         pronto e fino a che j non ha finito */
5 |     /* SEZIONE CRITICA */
6 |     flag[i] = false;
7 |     /* sezione rimasta */
8 |

```

Osserviamo che possiamo intendere l'esecuzione della parte critica di un processo come una galleria a senso unico alternato: se un processo deve entrarci ma al suo interno ce n'è già un altro attende, altrimenti ci entra. L'esecuzione in "parallelo" è quindi mantenuta dato che un processo può essere molto lontano dalla galleria mentre il secondo la sta attraversando. Con questo modello, quindi, tutti e 3 i requisiti sono rispettati.

4.2.1 Architetture moderne

Ciò nonostante, questa soluzione diventa obsoleta per i sistemi moderni *multicore* e *multithread*. Questo perché nelle architetture moderne il compilatore si prende la libertà di riordinare e riorganizzare il codice. Vediamo un esempio di due thread (assumiamo che le variabili condivise siano *x* e *flag*):

```

1 | /* THREAD 1 */
2 | while(!flag);
3 | print x
4 |
5 | /* THREAD 2 */
6 | x = 100;
7 | flag = true;

```

Una volta eseguito il codice, ci aspettiamo che l'output sia 100. Però, se le istruzioni nel thread 2 vengono invertite, il risultato è completamente diverso, perché il thread 1 viene eseguito prima che la variabile *x* venga cambiata a 100.

```

1 | /* THREAD 2 */
2 | flag = true;
3 | x = 100;

```

Al fine di fare in modo che la soluzione di Peterson funzioni anche nelle architetture moderne si introduce la **memory barrier**: questa è un'istruzione che rende le modifiche effettuate in memoria visibili a tutti i processori (*cores*). Questo tipo di modello di memoria si dice *strongly ordered*, ovvero una modifica in memoria è propagata immediatamente a tutti i core; si contrappone al *weakly ordered* dove una modifica in memoria non è propagata istantaneamente.

Con l'introduzione delle memory barrier ora la soluzione di Peterson rimane valida. Questo perché quando viene invocato un `memory.barrier()` il sistema si assicura che tutti i load e store in memoria vengano completati prima di ogni altro

load e store. Quindi anche se il codice venisse riordinato, la memory barrier si assicura che le modifiche in memoria siano completate.

4.3 Sincronizzazione via hardware

Tra le diverse soluzioni che abbiamo visto, la più gettonata rimane comunque l'implementazione di particolari **istruzioni hardware** che permettono di modificare il contenuto di una *word* in memoria oppure di effettuare uno *swap* di due word.

4.3.1 Test and set

La prima delle due istruzioni HW che discuteremo è la `test_and_set` dove prende un input booleano e lo memorizza in una variabile temporanea `tmp`; in secondo luogo il valore in ingresso viene settato a `true` e infine la variabile temporanea viene restituita. Lo pseudocodice di quest'istruzione è il seguente (ricordiamo che sono istruzioni a livello HW, quindi il codice è solo a scopo concettuale):

```

1 | boolean test_and_set(boolean *target){
2 |     boolean tmp = *target; /* salvo il valore di target */
3 |     *target = true; /* setto a TRUE */
4 |     return tmp;
5 |

```

Si osserva che dopo l'esecuzione dell'istruzione, il valore di `target` è sempre `true` e viene ritornato il vecchio valore della variabile in ingresso. Ricordiamo infine che è un'istruzione **atomica**, ciò significa che non può essere interrotta.

Vediamo ora come questa istruzione possa esserci utile per l'esecuzione della sezione critica di un processo:

Codice 2: Utilizzo `test_and_set`

```

1 | while(true){
2 |     while(test_and_set(&lock)); /* lock = true, attendo per la
3 |         risolrsa, ora e' occupato.*/
4 |         /* lock = false, posso cominciare a eseguire la parte critica
5 |             perche' si e' liberato; metto lock = true*/
6 |         /* SEZIONE CRITICA */
7 |         lock = false; /* lock e' libero */
8 |         /* sezione rimanente */
9 |

```

4.3.2 Compare and swap

La seconda istruzione HW che andiamo a discutere si chiama `compare_and_swap` e, come per la precedente, anch'essa è un'istruzione atomica. Come vedremo, questa istruzione, ha ben 3 variabili in ingresso:

- ◊ `value`, che indica il valore da modificare;
- ◊ `expected`, che indica il valore che ci si aspetta contenga `value`;
- ◊ `new_value`, ovvero il nuovo valore con cui vogliamo cambiare `value`.

Osserviamo ora lo pseudocodice dell'istruzione:

```
1 | int compare_and_swap(int *value, int expected, int new_value){  
2 |     int tmp = *value;  
3 |     if (*value == expected)  
4 |         *value = new_value;  
5 |     return temp; /* ritorna vecchio valore di value */  
6 | }
```

Notiamo che, a differenza di `test_and_set`, in questo caso il valore di `value` viene modificato solo nel momento in cui coincide con il valore che ci aspettiamo: questo viene modificato con il valore inserito (`new_value`). Capiamo ora come questa istruzione possa essere utilizzata per la gestione della sezione critica e come questa sia più **flessibile** rispetto alla precedente.

Codice 3: Utilizzo di `compare_and_swap`

```
1 | while(true){  
2 |     while(compare_and_swap(&lock, 0, 1) != 0);  
3 |     /* lock = 1 = true, continuo ad aspettare */  
4 |     /* lock = 0 = false, viene ritornato 0, quindi esco dal while, e  
5 |        occupo lo spazio, lock = 1 */  
6 |     /* SEZIONE CRITICA */  
7 |     lock = 0;  
8 |     /* sezione rimasta */  
9 | }
```

4.3.3 Variabili atomiche

Istruzioni come `compare_and_swap` sono utilizzate per comporre dei blocchi per comporre altri oggetti di sincronizzazione. Uno tra questi oggetti è la variabile atomica che fornisce degli aggiornamenti elementare a dei dati primitivi. Segue infatti l'esempio di `increment()` dove il valore intero `v` viene incrementato senza interruzioni.

Codice 4: Esempio di una variabile atomica `increment()`

```
1 | void increment(atomic_int *v){  
2 |     int temp;  
3 |     do{  
4 |         temp = *v;  
5 |     } while (temp != compare_and_swap(v, temp, temp+1));  
6 | }
```

4.4 Mutex lock e Semafori

Le soluzioni che abbiamo visto in precedenza erano intricate e spesso non erano accessibili da applicazioni esterne; inoltre creano ulteriori complicazioni in sistemi multithreading. A questo proposito i progettisti di sistemi operativi hanno costruito diversi *tool* al fine di risolvere il problema della sezione critica.

Il primo che andremo a vedere è il **mutex lock** (MUTual EXclusion lock) che è una variabile booleana che indica se il **lock**, ovvero la sezione critica è disponibile o meno. Questa è protetta da due istruzioni atomiche di acquisizione e rilascio:

- ◊ `acquire()` che blocca la CS²;
- ◊ `release()` che rilascia il lucchetto e quindi la CS è liberata.

Spesso queste sono implementate via HW attraverso istruzioni come `compare_and_swap`.

Come possiamo notare dallo pseudocidce sottostante, ci troviamo in una situazione di **busy waiting** in quanto il thread continua ad entrare nel ciclo fino a che non acquisisce il lucchetto. È evidente che stiamo sprecando CPU, che potrebbe essere usata per altri thread.

```

1 | while(true){
2 |   acquire();
3 |   /* SEZIONE CRITICA */
4 |   release();
5 |   /* sezione rimanente */
6 | }
```

A questo proposito, questa pratica è raffinata tramite i **semafori**, che non sono altro che una **variabile intera** (`S`) accessibile, anche in questo caso attraverso due operazioni atomiche: `wait()` e `signal()`.

```

1 | wait(S){
2 |   while (S<=0); /* busy wait */
3 |   S--;
4 | }
5 |
6 | signal(S){
7 |   S++;
8 | }
```

Se la variabile `S` può assumere solo il valore di 0 o 1, si parla di semafori **binari** (ovvero dei mutex lock), altrimenti si fa riferimento ai **counting semaphores**.

Attraverso i semafori abbiamo la possibilità di fare eseguire una parte di processo prima che un altro processo inizi. Siano P_1 e P_2 due processi che contengono il segmento S_1 ed S_2 ; per fare in modo che S_1 sia eseguito prima di S_2 possiamo utilizzare i semafori.

```

1 | P1:
2 |   S1; /* eseguo S1 */
3 |   signal(synch); /* dico a P2 che ho finito */
4 |
5 | P2:
6 |   /* busy wait */
7 |   wait(synch); /* aspetto che P1 abbia finito */
8 |   S2; /* eseguo S2 */
```

Possiamo notare però che anche in questo caso P_2 è in una fase di busy waiting.

²CS sta per *Critic Session*, ovvero sezione critica

4.4.1 Waiting queue

Per evitare che accada è necessario implementare una **waiting queue**:

Codice 5: Struttura del semaforo con waiting queue

```
1 | typedef struct {
2 |     int value;
3 |     struct process *list; /* indica la prossima entry nella lista */
4 | } semaphore
```

Qui abbiamo a disposizione altre due operazioni per implementare la sincronizzazione tra processi:

- ◊ `sleep()` che mette a dormire il processo o il thread e lo inserisce nella waiting queue;
- ◊ `wakeup()` che rimuove il primo processo della coda e lo inserisce nella *ready queue*, coda che contiene i processi pronti per essere eseguiti.

A questo punto possiamo ridefinire le due operazioni basi del semaforo: `wait` e `signal`.

Codice 6: Utilizzo di semaforo con *waiting queue*

```
1 | wait(semaphore *s){
2 |     S->value--; /* decremento il valore del semaforo */
3 |     if (S->value < 0){
4 |         aggiungi processo in S->list;
5 |         sleep(); /* evito il busy waiting, mi tolgo dalla CPU e aspetto
6 |                     che signal() mi svegli dalla coda */
7 |     }
8 |
9 | }
10 |
11 | signal(semaphore *S){
12 |     S->value++; /* incremento il valore del semaforo */
13 |     if (S->value <= 0){
14 |         rimuovi processo in S->list;
15 |         wakeup();
16 |     }
17 | }
```

Bisogna però fare attenzione a utilizzare `wait` e `signal` nel modo corretto: non si può prima invocare `signal` e poi `wait` oppure invocare `wait` due volte di seguito. Questo utilizzo incorretto delle due operazioni può generare errori.

4.5 Monitor

Cerchiamo ora di risolvere il problema dei semafori: le operazioni di `wait` e `signal` erano lasciate al programmatore. Questa scelta ha come unica conseguenza la generazione di errori da parte dell'utilizzatore (che, come vedremo più avanti prendono il nome di **deadlock**, capitolo 5). Cerchiamo quindi di implementare un'astrazione dei semafori al fine di limitarne i danni. Queste astrazioni di alto livello sono proprio i **monitor** i quali garantiscono l'esecuzione di un processo alla volta.

4.5.1 Struttura e implementazione

La struttura di astrazione del monitor è formata da una variabile condivisa da tutti i processi, da del codice di inizializzazione per il monitor e da tante funzioni o procedure (ogni processo ha le stesse procedure).

Codice 7: Struttura del monitor

```

1 monitor name{
2     /* dichiarazione della variabile condivisa */
3     codice di inizializzazione (...) { ... }
4
5     procedura F1 (...) { ... }
6     procedura F2 (...) { ... }
7     /* ... */
8     procedura Fn (...) { ... }
9 }
```

Come possiamo osservare dalla figura 29, concettualmente, il monitor raggruppa tutti i processi attraverso dei dati condivisi in una coda.

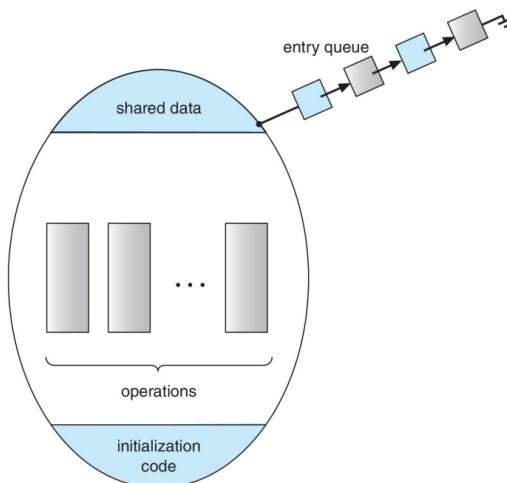


Figura 29: La struttura di astrazione del monitor.

Per implementare un monitor ci appoggiamo sui semafori, in particolare su un mutex:

```

1 semaphore mutex = 1; /* semaforo binario = mutex */
2
3 wait(mutex) /* aspetto che il mutex si liberi per occuparlo */
4     /* corpo della funzione Fi */
5 signal(mutex) /* libero il mutex **/
```

È quindi necessario modificare ogni procedura del monitor bloccando e poi rilasciando il mutex. In questo modo, a differenza dei semafori, le operazioni di wait e signal

sono già implementate all'interno delle procedure che eseguono quelle operazioni, al posto del programmatore. Un esempio di utilizzo completo è presente nella sezione 4.6.3.

4.5.2 Variabile di condizione

Oltre alla struttura base, illustrata nel paragrafo precedente, possiamo anche inserire le *condition variables* (rappresentazione in figura 30) che possono eseguire le stesse due operazioni dei semafori: `wait` e `signal`. In questo caso però le due operazioni

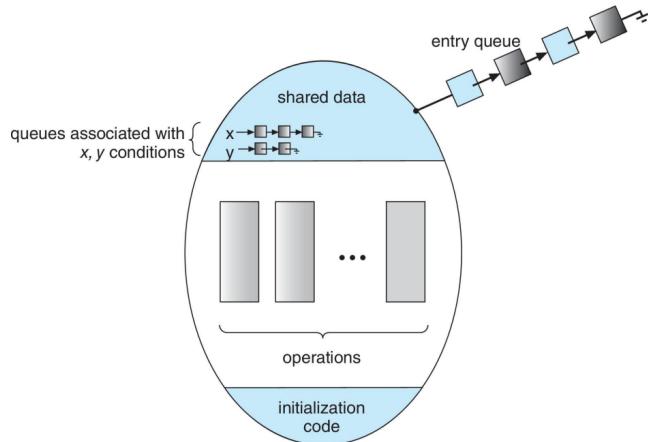


Figura 30: Caption

sono modificate e non portano ad errori se utilizzate male. Ricordiamo che, nel caso dei semafori, se `signal` fosse stato invocato senza aver prima invocato il corrispettivo `wait`, avrebbe erroneamente incrementato il contatore. In questo caso invece, se non è presente alcun processo in `wait`, allora `signal` non fa nulla.

Emerge ora un piccolo dubbio. Se ci sono diversi processi che sono in `wait`, quando viene effettuato un `x.signal()`, quale di questi processi viene eseguito per prima? Notiamo che ci troviamo in una situazione molto simile allo scheduling (capitolo 3) solo che, in questo caso non è più a livello di sistema operativo ma è a livello di monitor e di accesso alla sezione critica. Per risolvere questo problema ci possono essere algoritmi come FCFS oppure a livello di priorità: chi ha la priorità più alta viene fatto eseguire prima. Spesso la priorità è definita in base al tempo di utilizzo della sezione critica, ma si fa presente che se entrano nella coda processi con un tempo di utilizzo molto breve, i processi con un utilizzo lungo possono essere in attesa perenne ed andare in **stravation**. Ecco che non viene più rispettato il terzo requisito, ovvero che l'attesa di un processo per la sezione critica deve essere limitata (vedi paragrafo 4.1.1).

4.6 Problemi comuni della sincronizzazione

4.6.1 Buffer limitato

Il primo problema che andiamo a discutere è il problema del buffer limitato. Abbiamo a disposizione un buffer che contiene n elementi, ovvero il numero di processi in coda

per accedere alla loro sezione critica. Disponiamo inoltre di tre semafori:

- ◊ `mutex` che consente l'accesso al buffer in maniera esclusiva (inizializzato ad 1);
- ◊ `full` che segnala in numero di elementi contenuti all'interno del buffer (inizializzato a 0);
- ◊ `empty` che indica la quantità di spazi disponibili all'interno del buffer, sarebbe $n - full$ (inizializzato ad n).

Analizziamo la soluzione del problema, che è molto simile al problema del produttore e consumatore.

Codice 8: Problema del buffer limitato

```

1  /* Prodotto */
2  while (true){
3      /* produco un elemento */
4      wait(empty); /* aspetto che ci sia spazio all'interno del buffer,
5                     ovvero che ci sia almeno una locazione libera (empty > 0) */
6      wait(mutex); /* ora che c'e' un posto libero, richiedo l'accesso
7                     alla sezione critica */
8      /* SEZIONE CRITICA: aggiungo il codice nel buffer */
9      signal(mutex); /* libero la sezione critica */
10     signal(full); /* incremento di 1 il numero di elementi nel buffer
11 }
12
13 /* Consumatore */
14 while(true){
15     wait(full); /* aspetto che ci siano >0 elementi nel buffer */
16     wait(mutex); /* aspetto l'accesso esclusivo nel buffer */
17     /* SEZIONE CRITICA: rimuovo l'elemento dal buffer */
18     signal(mutex); /* libero la sezione critica */
19     signal(empty); /* incremento il numero di locazioni libere */
20 }
```

Osserviamo che i comandi `wait(empty)` e `wait(mutex)` all'interno di produttore **non** possono essere invertiti. Se prima infatti viene bloccato il `mutex`, il consumatore non sarà più in grado di liberare il buffer che, se è pieno, porta ad un'attesa perenne il produttore in `wait(empty)`: siamo ricaduti in una situazione di *deadlock*.

4.6.2 Problema dei lettori e degli scrittori

Il secondo problema è detto dei lettori e degli scrittori. In questo caso: (1) più lettori possono leggere lo stesso dataset (che può essere anche un file su disco) contemporaneamente (per definizione, non possono modificarlo) e (2) lo scrittore deve poter scrivere sul dataset senza che quest'ultimo venga letto. Solo nel momento in cui ha terminato (mutua esclusione), allora i lettori possono effettuare nuovamente l'accesso simultaneo.

Non ci soffermiamo sul codice, ci limitiamo a dire che è sufficiente utilizzare due semafori:

- ◊ `rw_mutex`, semaforo binario inizializzato ad 1, che indica che la risorsa è occupata dallo scrittore;

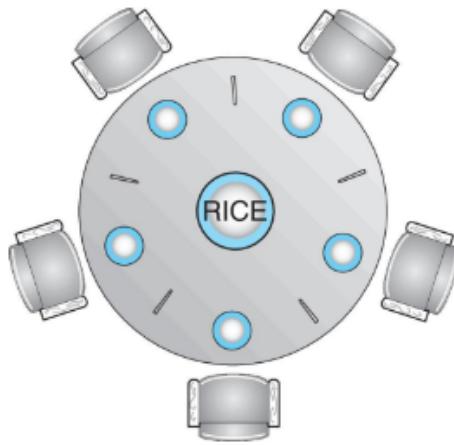


Figura 31: Disposizione dei 5 filosofi.

- ◊ `mutex`, anch'esso un semaforo binario che segnala che la risorsa è in lettura da almeno un lettore;
- ◊ `read_count`, un intero che conta il numero di lettore che stanno leggendo la risorsa.

In questo caso i problemi generati sono due. Il primo è che c'è la possibilità che lo scrittore non riesce mai ad effettuare l'accesso, entra quindi in una fase di attesa perenne. Il secondo problema che emerge è che quando lo scrittore ha effettuato l'accesso, nessun lettore è in grado di accedere: anche in questo caso si può ricadere in una fase di *starvation*.

4.6.3 Problema dei 5 filosofi

Il problema dei 5 filosofi, rimane il più conosciuto (e il più importante). Un filosofo, può pensare o mangiare. Facendo riferimento alla figura 31, osserviamo che i filosofi sono disposti in un tavolo rotondo che ha al centro una scodella di riso (che allegoricamente indica il dataset, ovvero la risorsa condivisa). Alla destra e alla sinistra di ciascun filosofo è presente una bacchetta: questa rappresenta il semaforo in quanto se un filosofo smette di pensare e inizia a mangiare, non può farlo se il filosofo alla sua sinistra o alla sua destra sta mangiando, in quanto almeno una delle due bacchette sono occupate.

Proviamo a dare una prima soluzione a questo problema di sincronizzazione con i semafori.

Codice 9: Risoluzione del problema dei filosofi con i semafori

```

1  while(true){
2      wait(chopstick[i]); /* attende la bacchetta a sinistra */
3      wait(chopstick[ (i + 1) % 5]); /* attende la bacchetta a destra */
4      /* SEZIONE CRITICA: mangia dalla ciotola di riso */
5      signal(chopstick[i]); /* rilascia la bacchetta a sx */

```

```

6 |     signal(chopstick[ (i + 1) % 5]); /* rilascia la bacchetta a dx */
7 |     /* pensa */
8 |

```

Sembrerebbe tutto ottimo, ma cosa succede se tutti e 5 i filosofi prendono la loro bacchetto sinistra allo stesso tempo? Entrano in una situazione di attesa infinita (deadlock) un quanto tutte le loro bacchette destre sono occupate dal filosofo alla loro destra. Proviamo a risolvere questo nuovo problema attraverso i monitor.

Codice 10: Risoluzione del problema dei filosofi con i monitor

```

1 monitor DiningPhilosophers{ /* creo la struttura astratta */
2     enum { THINKING, HUNGRY, EATING } state[5]; /* per ogni filosofo
3         e' definito lo stato in cui si trova */
4         condition self[5]; /* condition variable */
5
6     void test(int i){
7         if ((state[(i + 4) % 5] != EATING) /* se il vicino di sinistra
8             non sta mangiano */ &&
9             (state[i] == HUNGRY) /* se ho fame */ &&
10            (state[(i + 1) % 5] != EATING)) /* se il vicino di destra
11                non sta mangiando */ {
12                state[i] = EATING; /* mi metto a mangiare */
13                self[i].signal();
14            }
15        }
16
17        void pickup(int i){ /* acquire */
18            state[i] = HUNGRY;
19            test(i); /* se posso, mi metto a mangiare */
20            if (state[i] != EATING) /* se non sto mangiando */
21                self[i].wait(); /* mi metto in attesa*/
22        }
23
24        void putdown(int i){ /* release */
25            state[i] = THINKING; /* mi rrimetto a pensare */
26            /* controllo che il mio vicino sinistro e destro non siano in
27                attesa */
28            test((i + 4) % 5);
29            test((i + 1) % 5);
30        }
31
32    initialization_code(){ /* all'inizio tutti i filosofi stanno
33        pensando */
34        for (int i = 0; i < 5; i++)
35            state[i] = THINKING;
36    }
37

```

Dopo aver implementato questa struttura astratta è semplicemente necessario utilizzare le due operazioni pickup() e putdown() al fine di fare in modo che tutti i

filosofi mangino sincronizzati. Ad ogni modo, anche in una situazione così raffinata la **starvation** è possibile, nel caso in cui un filosofo si metta a mangiare e non smetta più.

```
1 | DiningPhilosophers.pickup(i);  
2 | /* SEZIONE CRITICA (mangio) */  
3 | DiningPhilosophers.putdown(i);
```

5 Deadlocks

In questo capitolo cercheremo di approfondire i deadlocks: capiremo sotto quali condizioni si giungerà ad una soluzione di deadlock e i diversi algoritmi per la gestione dei deadlocks.

Modello di sistema

Prima di discutere effettivamente dei deadlock è importante fare un piccola premessa. In questo capitolo avremo a che fare con sistemi che possono essere composti da m tipi di risorse generiche: queste potrebbero essere dei dispositivi I/O, dati su disco, delle allocazioni in memoria, la CPU oppure dei semafori generici. Queste risorse vengono definite con la notazione R_m . Inoltre, ogni tipo di risorsa può essere **singolo**, come un slot del DVD sul computer, oppure **multiplo** come la presenza di diverse porte USB oppure molti buffer in memoria.

5.1 Nozioni fondamentali

Ripassiamo innanzitutto il concetto di deadlock. Prendiamo il caso in cui due semafori S_1 ed S_2 , entrambi inizializzati a 1, siano condivisi da due thread T_1 e T_2 . Osservando lo pseudo-codice seguente, si ha una classica situazione di deadlock. Se T_1 e T_2 sono eseguiti contemporaneamente, T_1 occupa subito S_1 e T_2 occupa subito S_2 . Dopo di che T_1 aspetta S_2 che però è occupato da T_2 il quale è in attesa di S_1 , occupato da T_1 . Siamo quindi in una situazione di stallo dove entrambi i thread stanno aspettando l'altro e nessuno dei due termina l'esecuzione

Codice 11: Classica situazione di deadlock

```
1  /* T1 */
2  wait(S1); /* occupa S1 */
3  wait(S2); /* aspetta per S2 */
4
5  /* T2 */
6  wait(S2); /* occupa S2 */
7  wait(S1); /* attende S1 */
```

5.1.1 Caratterizzazione

Possiamo osservare che si devono verificare 4 condizioni affinché si arrivi ad una situazione di deadlock.

1. **Mutua esclusione:** banalmente, ci deve essere almeno una risorsa che sia utilizzata da un thread alla volta;
2. **Hold and wait:** deve esistere almeno un thread che sta cercando di accedere alla risorsa che è già occupata e, di conseguenza, deve attendere;
3. **No preemption:** una risorsa può essere rilasciata volontariamente dei thread. Le risorse non possono quindi essere "rubate" da altri thread e, di conseguenza, un thread non può essere spezzato durante la sua fase critica;

- Attesa circolare: è il requisito più importante, deve verificarsi una situazione come quella dei cinque filosofi (paragrafo 4.6.3), ovvero che un thread attende la conclusione di un altro thread che ... che aspetta la conclusione del thread iniziale.

Se almeno una di queste quattro condizioni **non** è **verificata** allora non si è in una situazione di deadlock.

5.1.2 Grafo risorsa-allocazione

Uno strumento molto importante, utile per visionare i diversi casi di deadlock è il grafo risorsa-allocazione. Questo grafo è composto, come ogni grafo, da vertici e archi. I vertici si suddividono in due tipi: il primo tipo rappresenta la risorsa R_m mentre il secondo tipo indica il thread T_n . Il grafo è **diretto**, ciò significa che gli archi hanno una direzione. In particolare $R \rightarrow T$ indica che il thread T detiene/occupa la risorsa R , mentre $T \rightarrow R$ segnala che il thread T è in attesa della risorsa R . Inoltre, si osserva che generalmente i vertici che rappresentano una risorsa multipla contengono tanti pallini (\bullet) quante sono le istanze di quella risorsa disponibili (come le 5 porte USB).

Prendiamo come esempio la figura 32. Possiamo notare che rispetta tutte "regole" del grafo risorsa-allocazione. In particolare possiamo affermare che R_1 ed R_3 sono risorse singole mentre R_2 ed R_4 sono risorse multiple (con, rispettivamente, 2 e 4 istanze). Inoltre possiamo dire che ci troviamo in una situazione di *deadlock*. Perché?

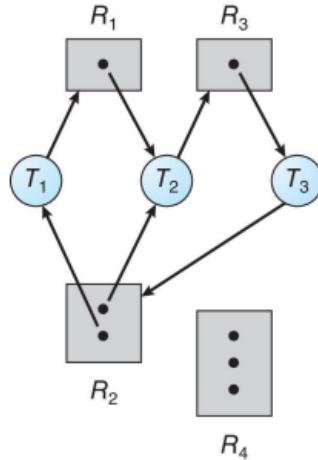


Figura 32: Rappresentazione grafica di un grafo che presenta una situazione di deadlock.

Partiamo da R_2 , questa è occupata da T_1 e T_2 ; T_1 è in attesa di R_1 la quale è occupata da R_2 che, a sua volta, sta aspettando che R_3 venga liberata da T_3 il quale sta attendendo la liberazione due R_2 , occupata per l'appunto da T_1 e T_2 . È proprio una situazione di deadlock: a primo impatto possiamo dire ciò perché è presente un **ciclo**. Osservando però la figura 33, capiamo che il fatto che ci sia un ciclo non implica che ci sia per forza una situazione di deadlock. Se osserviamo infatti la figura 33 riconosciamo che

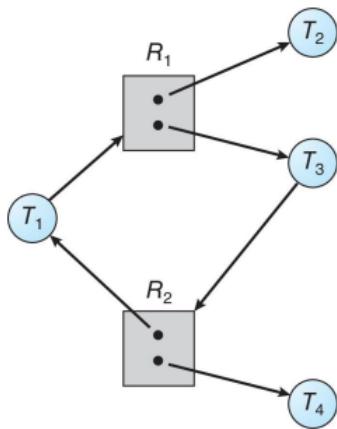


Figura 33: Rappresentazione grafica di un grafo senza deadlock.

prima o poi il thread T_2 oppure T_4 rilascerà la risorsa e di conseguenza le richiesta degli altri thread saranno soddisfatte.

Riassumiamo ora queste informazioni nel seguente diagramma:

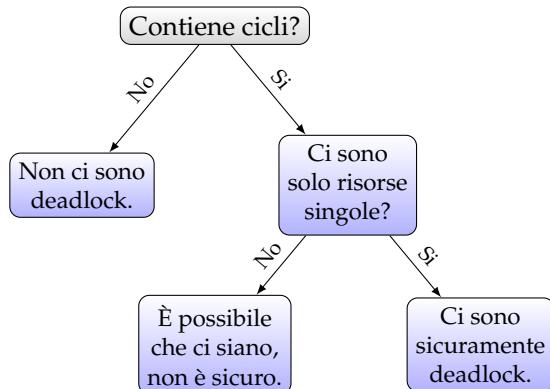


Figura 34: Piccolo *decision tree* utile per capire se è presente un deadlock oppure no.

Livelock

Oltre al deadlock esiste anche una seconda situazione di stallo chiamata *livelock*. In questo caso i due processi non sono completamente in una situazione di stallo ma non riescono a terminare la loro esecuzione. Possiamo allegoricamente spiegare queste situazione con un corridoio dove ci sono due persone, A e B, che vanno in direzione opposta. Ad un certo punto, nel tempo, si incontrano, allora, per non scontrarsi, A si sposta, ma contemporaneamente lo fa anche B. A questo punto A si sposta di nuovo, ma lo fa anche B. I due processi quindi si ostacolano l'un l'altro non riuscendo a terminare.

Prevention

Come vedremo in questo capitolo, ci sono diversi metodi finalizzati alla gestione dei deadlock. Il primo metodo che andiamo a discutere è quello di "prevenzione". In essenza, questo metodo si occupa di **invalidare** una delle quattro **condizioni** necessarie che generano una situazione di deadlock (vedi paragrafo 5.1.1). Anticipiamo già che questo approccio non è assolutamente efficiente e non è utilizzato.

1. La prima condizione, ovvero quella di mutua esclusione raramente può essere invalidata. Proprio perché la sincronizzazione tra processi nasce proprio per far sì che solo un processo abbia accesso ad una risorsa, la mutua esclusione è fondamentale. Altrimenti si incombe in situazioni dove più processi sono, per esempio, in scrittura, sullo stesso file.
2. Per quanto riguarda l'*hold and wait*, in questo caso si cerca di fare in modo che un processo che detiene una risorsa non ne può richiedere un'altra (si rimuove l'*hold* dalla definizione). Di conseguenza si fa in modo che un thread, prima di essere eseguito, faccia una richiesta contemporanea a tutte le risorse di cui ha bisogno lasciano tutti gli altri processi ad aspettare. Si entra in una situazioni di inefficienza e, talvolta, c'è il rischio che un processo vada in **starvation**.
3. Affine di invalidare il punto tre si aggiunge la *preemption*, ovvero che i processi possono essere interrotti al fine di lasciare la risorsa ad altri processi. Anche in questo caso però si possono verificare situazioni assurde: si pensi all'utilizzo di una stampante, un thread non può fermare la stampa e iniziare a stampare. Si ha anche in questo caso una soluzione poco ottimale.
4. Solo nel quarto punto, dove si cerca di evitare l'attesa circolare si ha effettivamente una soluzione sensata. Nella pratica si cerca di eliminare la circolarità imponendo dei vincoli particolare ai processi, per esempio un **ordine** di esecuzione.

5.2 Avoidance

Il secondo approccio che andremo a discutere più a fondo è una sorta di miglioramento della prevenzione. In questo caso si cerca proprio di **evitare** di ricadere in situazioni che possono portare al deadlock, è una prevenzione molto più *strict*.

Affinché questo approccio funzioni però deve essere fornita un'informazione importante da parte del processo, ovvero il numero massimo di risorse per tipo che il processo necessita durante la sua esecuzione. Per esempio, 6 risorse di tipo A, 5 risorse di tipo B e 2 risorse di tipo C.

5.2.1 Safe state

Il safe state, ovvero lo **stato sicuro**, è una situazione in cui esiste un ordine di esecuzione di thread secondo il quale tutti i thread possono essere completati uno dopo l'altro. Quando un processo fa la richiesta per le risorse, prima di concederle, l'algoritmo (che vedremo in seguito) controlla se dopo la cessione delle risorse si è in una situazione ancora safe. Se sì, allora le risorse vengono occupate dal processo, altrimenti il processo deve aspettare il termine di alcuni thread.

Nella primo grafo della figura 35 si è in una condizione non sicura in quanto non esiste una sequenza di esecuzione dei processi dove non si verifichi una situazione di deadlock. Nel secondo grafo invece si è in una situazione safe in quanto esiste una

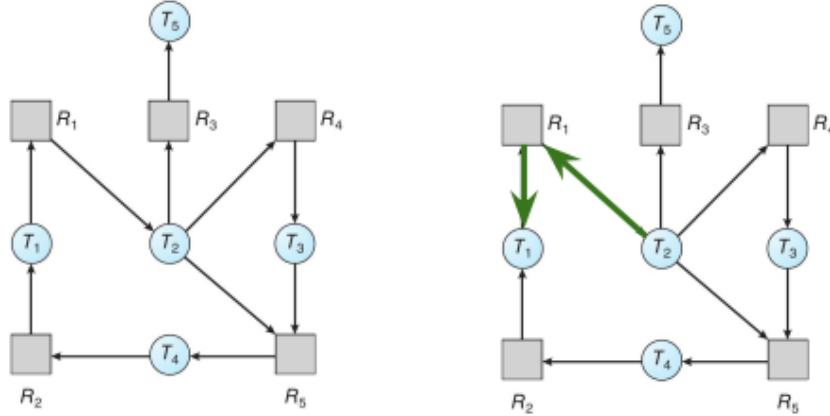
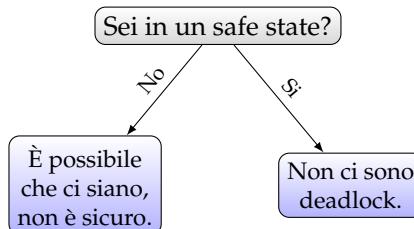


Figura 35: Differenza tra situazione unsafe e safe.

sequenza dove tutti i processi possono essere eseguiti senza entrare in una condizione di deadlock: $T_1 \rightarrow T_4 \rightarrow T_3 \rightarrow T_5 \rightarrow T_2$ infatti è una sequenza che permette il completamento dell'esecuzione dei threads. Anche in questo caso possiamo riassumere tutto in un piccolo *decision tree*:



Ricordiamo infine che gli algoritmi che vedremo in questo paragrafo al fine di prevenire i deadlock **evitano** di entrare in **unsafe state**, rimanendo quindi in safe state.

5.2.2 Algoritmo sul grafo risorsa-allocazione

Iniziamo con un algoritmo che funziona solo nel caso di **risorse singole**, ovvero nel caso in cui si ha solo un pallino (●) nel grafo, e quindi ogni tipo di risorsa ha una sola istanza. Introduciamo ora la nozione di **claim edge** ovvero un arco tratteggiato che indica che il processo T_i può richiedere la risorsa R_j . Questi archi diventano **request edge** nel momento in cui si effettua la richiesta e poi si trasformano a loro volta in **assignments edge** nel momento in cui stanno utilizzando la risorsa. Osservando il grafo in figura 36 notiamo che $T_1 \rightarrow R_2$ è un claim edge, $T_2 \rightarrow R_1$ è un request edge e $R_1 \rightarrow T_1$ è un assignment edge.

Nella situazione presentata in figura 36, si rischia di essere in una situazione *unsafe*. Questo perché se R_2 è assegnata al thread T_2 nel caso in cui il thread T_1 effettui

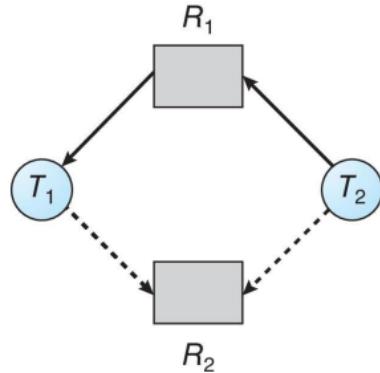


Figura 36: Un grafo composto da tutti e tre i tipi di archi.

la richiesta per R_2 si è in una situazione di deadlock. Possiamo infatti osservare che si ha la presenza di un ciclo e che tutte le risorse abbiano un'istanza, facendo riferimento allo schema in figura 34, si ha che si è sicuramente in una situazione di deadlock. Ponendo infatti che R_2 venga assegnata a T_2 , nel momento in cui T_1 effettui una richiesta, l'algoritmo la nega, in modo tale da non essere in una situazione di deadlock.

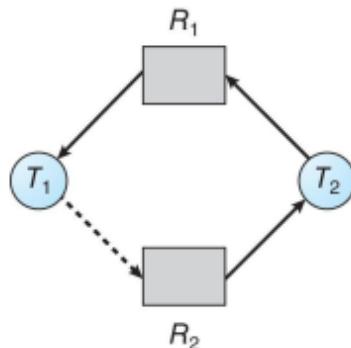


Figura 37: L'algoritmo nega la richiesta di T_1 per R_2

5.2.3 Algoritmo del banchiere

Passiamo ora ad una situazione più reale, quella in cui esistono più risorse con più istanze ciascuna. L'algoritmo a cui si fa affidamento è detto *Banker's algorithm*, ovvero l'algoritmo del banchiere.

Innanzitutto, definiamo con n il numero di processi ed m il numero di tipi di risorse; definiamo poi tre matrici e un'array, strutture sul quale l'algoritmo si appoggia:

- ◊ L'array **available** (disponibili) che indica quante istanze di risorse sono disponibili per ogni tipo. Per esempio, la risorsa j ha a disposizione 3 istanze, e questo $\forall j \in [0, m]$.

- ◊ La prima matrice, **max** che indica il massimo numero di istanze della risorsa *j*esima che il processo *i*esimo ha al più bisogno ($\forall i \in [0, n]$).
- ◊ La seconda matrice **allocated** (allocato), che indica quante istanze della risorsa *j*esima il processo *i*esimo ha già allocato.
- ◊ La terza matrice, **need** (bisogno), che non è altro che la differenza tra *max* e *allocated*; questa matrice indica di quante altre istanze della risorsa *j*esima il processo *i*esimo potrebbe, al più, aver bisogno.

Per capire l'algoritmo al meglio si può tranquillamente trascurare lo pseudocodice e partire subito con un **esempio**. Abbiamo $n = 5$ thread, da T_0 a T_4 , e 3 tipi di risorse: A, con 10 istanze, B, con 5 e C, che ha a disposizione 7 istanze. Al tempo iniziale t_0 , la situazione è la seguente:

Threads	Allocated			Max		
	A	B	C	A	B	C
T_0	0	1	0	7	5	3
T_1	2	0	0	3	2	2
T_2	3	0	2	9	0	2
T_3	2	1	1	2	2	2
T_4	0	0	1	4	3	3

Un esempio di lettura della tabella è il seguente: T_0 detiene un'istanza di tipo B e zero istanze di tipo A e C. Può richiedere un massimo di 7 istanze di tipo A, 5 istanze di tipo B e 3 istanze di tipo C.

Con questi dati possiamo ora calcolare le due strutture che ci mancano: *Available* e *Need*, che rispettivamente indicano le risorse che sono disponibili (per ogni tipo) e il numero di risorse che un determinato thread può richiedere. Osserviamo che per calcolare il vettore di risorse disponibili è necessario sottrarre alle istanze iniziali la somma delle risorse occupate dai threads. Per esempio, nel caso delle istanze disponibili della risorsa A abbiamo che sono $10 - (2 + 3 + 2) = 3$, occupate rispettivamente da: T_1 , T_2 e T_3 . per calcolare invece la matrice *Need*, per ogni cella della matrice si sottrarre il valore di *Max* al valore di *Allocated* nella cella $[i, j]$.

			<i>Need</i>		
			A	B	C
			7	4	3
<i>Available</i>			1	2	2
A	B	C	3	3	2
6	0	0	0	1	1
4	3	1			

Abbiamo ora tutte le risorse necessarie per iniziare a eseguire l'algoritmo. Iniziamo scorrendo la matrice *Need* e comparandola con il vettore *Available*. Partiamo dal thread T_0 : con le risorse disponibili $[3, 3, 2]$, il thread ha la possibilità di terminare? No, perché per terminare necessita, al più, di 7 istanze di A, 4 istanze di B e 3 istanze di C. Passiamo ora al secondo thread: T_1 può terminare con le risorse disponibili? Sì, perché tutte le risorse di cui ha bisogno sono \leq rispetto alle risorse disponibili. A questo punto allora si spostano le risorse necessarie a T_1 per terminare, ovvero $[1, 2, 2]$:

<u>Threads</u>	<u>Allocated</u>			<u>Max</u>			<u>Need</u>			<u>Available</u>
	A	B	C	A	B	C	A	B	C	
T_0	0	1	0	7	5	3	7	4	3	
T_1	3	2	2	3	2	2	0	0	0	A
T_2	3	0	2	9	0	2	6	0	0	B
T_3	2	1	1	2	2	2	0	1	1	C
T_4	0	0	1	4	3	3	4	3	1	

Una volta terminato il thread T_1 , le risorse vengono rilasciate e il vettore *Available* va aggiornato, sommando alle risorse disponibili le risorse che sono appena state rilasciate da T_1 .

$$\begin{array}{ccc} \text{Available} \\ \hline \text{A} & \text{B} & \text{C} \\ 5 & 3 & 2 \end{array}$$

A questo punto ci sono abbastanza risorse per terminare un altro processo come, per esempio T_3 oppure T_4 . Così facendo, mano che i thread vengono eseguiti e terminati vengono rilasciate abbastanza risorse per completare quelli più onerosi. Una sequenza esempio dell'ordine con il quale i thread sono completati può essere: $\{T_1, T_3, T_4, T_2, T_0\}$.

Può succedere che durante l'esecuzione dei thread possono arrivare altre richieste che devono essere accettate o meno. Per esempio se prima dell'esecuzione T_4 avesse fatto una richiesta $[3, 3, 0]$, la richiesta sarebbe stata rifiutata dato che le risorse disponibili erano minori rispetto alla richiesta.

5.3 Detection

L'ultimo approccio che andiamo a discutere è il rilevamento del deadlock: si va alla ricerca di tale e, una volta trovato, si passa al salvataggio, alla **recovery**. Anche in questo caso, come per l'approccio di *avoidance* (paragrafo 5.2) sono presenti due algoritmi, uno studiato per avere solo un'istanza per risorsa e il secondo che copre anche il caso in cui ci siano più istanze per risorse.

5.3.1 Istanza singola

In questo primo caso si fa riferimento ad un secondo grafo particolare chiamato **wait-for graph**. È una modifica del *resource-allocation graph* dove si evidenzia il fatto che un thread sta aspettando il termine dell'esecuzione di un altro thread: non sono infatti presenti vertici che rappresentano risorse a ci sono solo nodi che indicano i thread. In figura 38 si osserva la traduzione di un grafo risorsa allocazioni in grafo *wait-for*. Si nota che nel secondo grafo si accetta ancora di più la presenza di cicli e, dato che si tratta di risorse con istanza singola, la presenza di un ciclo ha l'immediata conseguenza di generare una situazione di deadlock.

L'algoritmo di cui stiamo parlando infatti, per rilevare un deadlock, si limita a creare un *wait-for graph* e controllare la presenza di cicli³. Una volta trovato il ciclo si è

³Da studente che ha seguito il corso "Dati e Algoritmi" penso che si utilizzino algoritmi basati su BFS oppure DFS.

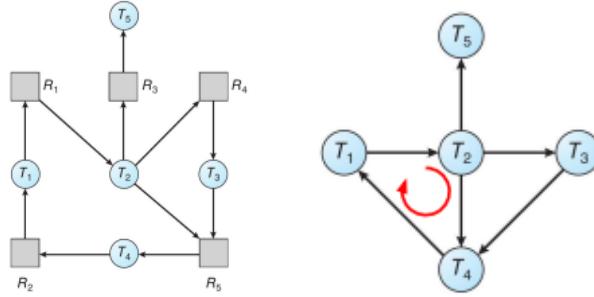


Figura 38: Un grafo *resource-allocation* e il suo rispettivo *wait-for graph*

sicuri di aver trovato il deadlock e si passa alla *deadlock recovery* (vedi paragrafo 5.3.3). Possiamo quindi notare che nella figura 38 si è in una condizione di deadlock dato che il thread T_1 sta attendendo il termine di T_2 che è in attesa di T_4 che a sua volta sta aspettando T_1 . Una situazione analoga è presente anche con i thread T_1, T_2, T_3, T_4 .

5.3.2 Istanze multiple

Nel caso invece delle istanze multiple, anche in questo algoritmo si fa affidamento a diverse strutture dati, proprio come nell'algoritmo del banchiere. Avremo quindi a che vedere con n processi che hanno a che fare con m tipi di risorse che interagiranno con due matrici e un vettore.

- ◊ Il vettore *Available*, proprio come nell'algoritmo del banchiere, tiene traccia di quante istanze sono libere per ogni tipo di risorsa.
- ◊ La prima matrice *Allocated* memorizza il numero di risorse che ogni processo detiene.
- ◊ La seconda matrice, **Request**, indica quante risorse il thread sta richiedendo al fine di terminare.

Come nel caso dell'algoritmo del banchiere, non ci soffermiamo sullo pseudocodice ma partiamo subito con un esempio. Poniamo di avere cinque threads, da T_0 a T_4 e di avere anche in questo caso tre tipi di risorse: A, con 7 istanze, B con 2 istanze e C con 6 istanze. Al tempo iniziale t_0 ci troviamo nella seguente situazione:

Threads	Allocated			Request			<i>Available</i>
	A	B	C	A	B	C	
T_0	0	1	0	0	0	0	
T_1	2	0	0	2	0	2	A B C
T_2	3	0	3	0	0	0	0 0 0
T_3	2	1	1	1	0	0	
T_4	0	0	2	0	0	2	

Dalla tabella *Request* possiamo notare che sia T_0 che T_2 possono completare senza attendere altri processi dato che non richiedono nulla. Una volta terminati T_0 e T_2 tutti gli altri thread possono man mano terminare. Non siamo quindi in una situazione di deadlock.

Threads	Allocated			Request			<i>Avaliable</i>
	A	B	C	A	B	C	
T_0	0	1	0	0	0	0	
T_1	2	0	0	2	0	2	A
T_2	3	0	3	0	0	1	B
T_3	2	1	1	1	0	0	C
T_4	0	0	2	0	0	2	

Mettiamoci però nel caso in cui T_2 debba richiedere una risorsa di tipo C. In questo caso le risorse che vengono rilasciate al termine di T_0 non sono necessarie per terminare gli altri thread e ci troveremo quindi in una situazione di deadlock.

5.3.3 Deadlock recovery

Come dobbiamo comportarci nei momenti in cui rileviamo un deadlock? La soluzione più brutale è quella di terminare tutti i processi che si trovano all'interno del deadlock (come fare un CTRL + C ad ogni processo). Esiste però un altro modo un po' più elegante che fa comunque affidamento alla **terminazione dei processi**: si cerca infatti di terminare ad uno ad uno tutti i thread che compongono il ciclo. L'ordine di terminazione può essere scelto in base alla priorità, in base alle risorse che il thread ha utilizzato o altro.

Una seconda strada invece si basa sul **resource preemption**, ovvero sul rubare delle risorse che sono detenute da un altro processo o thread. In questo caso si sceglie una vittima a cui rubare le risorse e si cerca di far terminare un thread. Si ricorda che anche in questo caso si può arrivare ad una situazione di *starvation* se lo stesso thread è sempre preso come vittima.

Parte III

Gestione della Memoria

6 Memoria principale

Fino ad ora abbiamo discusso di come un sistema operativo gestisce lo scheduling (3), la sincronizzazione (4) e i deadlock (5) sorvolando sempre sulla locazione effettiva su cui questi processi vengono mantenuti, ovvero la memoria. Osserviamo sin dall'inizio che con memoria intendiamo la memoria RAM del calcolatore, le memorie secondarie molto capienti sono i dischi e le memorie di massa, che verranno discusse approfonditamente nel capitolo 8. In particolare in questo capitolo ci occuperemo di dare un'introduzione ai concetti generali, passando poi a discutere il metodo più semplice per risolvere l'utilizzo della memoria, ovvero l'allocazione continua per poi discutere un tecnica più avanzata che tutt'ora si trova nei sistemi operativi moderni, ovvero il *paging*. Scopriremo che saranno necessarie delle strutture ausiliarie, come la tabella delle pagine, e ne studieremo i diversi metodi di implementazione. Infine ci dedicheremo al concetto di *swapping* e a modelli di allocazioni alternativi alla paginazione.

6.1 Introduzione

Come sappiamo, il programma, una volta che viene eseguito, deve essere caricato dal disco in memoria. Solo una volta che è stato caricato in memoria può essere eseguito; questo perché la CPU non ha accesso diretto alla memoria di massa. Sappiamo inoltre che l'accesso della CPU è effettuato seguendo 3 tipi di memorie (figura 39):

- ◊ Registri, che hanno un accesso rapidissimo;
- ◊ RAM, che ha una velocità sicuramente ridotta rispetto ai registri;
- ◊ Cache, che è una piccola memoria che fa da intermediario tra i registri e la RAM ed ha un tempo di accesso molto più veloce della memoria.

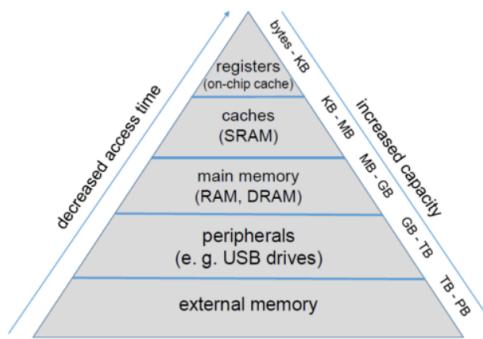


Figura 39: La gerarchie delle memorie nel calcolatore.

6.1.1 Protezione

In generale, se vogliamo avere molti processi che vengono caricati contemporaneamente nell'area della memoria e che vengono eseguiti in modo concorrente, è importante che ci sia una protezione dei processi fornita dal sistema operativo al fine di garantire che ogni processo abbia il suo spazio di memoria senza che vada a collidere con un altro processo. La soluzione più banale, rappresentata in figura 40, è quella di utilizzare due registri che tengono traccia di due valori: l'indirizzo **base**, ovvero l'indirizzo in memoria iniziale da cui il processo parte, e il valore **limite** (*offset*) che indica la massima espansione del processo in memoria. In altre parole, il processo in memoria non può superare l'indirizzo "base + limite" (più avanti in questo capitolo vedremo soluzioni più complesse e raffinate). In questo caso l'unica preoccupazione

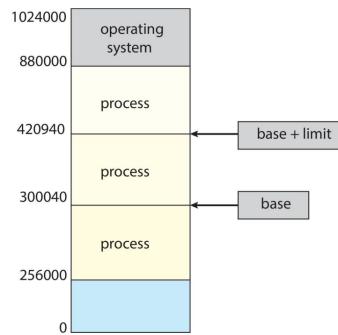


Figura 40: Limite inferiore e limite superiore dello spazio del processo in memoria.

del sistema operativo è controllare, ogni volta che un indirizzo viene generato dalla CPU durante l'esecuzione del programma, se questo indirizzo è compreso tra la base e l'offset fornito dai due registri. Nel caso in cui l'indirizzo è compreso allora la localizzazione di memoria può essere acceduta. Osserviamo che la figura 41 rappresenta un piccolo schema dove viene illustrata questa verifica da parte del sistema operativo.

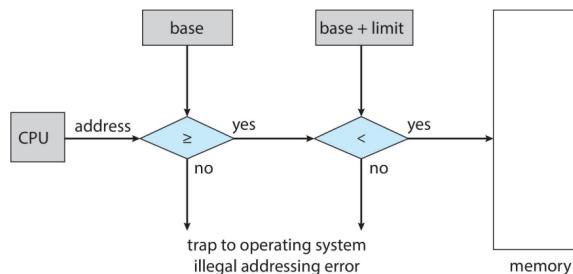


Figura 41: Procedura di controllo di un indirizzo mediante il SO.

6.1.2 Binding

Quando creiamo un processo, di default parte dall'allocazione 0000. Naturalmente non è possibile far partire tutti i processi dalla stessa allocazione altrimenti si genererebbero conflitti che causerebbero problemi non indifferenti. Inoltre ogni processo

che viene eseguito, anche se il suo indirizzo in memoria è diverso 0000, viene eseguito come se la sua prima cella disponibile fosse la 0000. Come è possibile? Il procedimento è chiamato **binding** e mappa due tipi di indirizzi: gli indirizzi **assoluti**, che sono gli effettivi indirizzi nella memoria, e gli indirizzi **rilocabili** che sono gli indirizzi relativi al programma in esecuzione. Per chiarire le idee, facciamo un esempio: poniamo di avere un processo che parte dall'indirizzo in memoria 31400 (indirizzo assoluto) e che stia eseguendo una linea di codice che secondo il processo è all'indirizzo 15. In realtà il programma non sta eseguendo l'indirizzo 15 ma sta eseguendo l'indirizzo assoluto 31415, che è la somma tra l'indirizzo fisico di inizio e l'indirizzo logico (31400 + 15). Questa somma, è infatti detta binding e permette di collegare gli indirizzi logici del processo agli indirizzi fisici della memoria.

Il binding però può avvenire in diversi momenti, come è possibile anche osservare dalla figura 42. Possiamo distinguere principalmente 3 momenti:

1. **Tempo di compilazione:** in questo caso, il compilatore, compilando il codice automaticamente converte gli indirizzi rilocabili in indirizzi assoluti.
2. **Tempo di caricamento:** nel momento di *linking* del programma e quindi si ha un eseguibile che ha ancora gli indirizzi temporanei ma che poi, una volta caricato, verranno sostituiti con gli indirizzi assoluti.
3. **Tempo di esecuzione:** è infine possibile trasformare gli indirizzi rilocabili in indirizzi assoluti durante l'esecuzione del programma. Questo è il metodo tipicamente utilizzato dai sistemi operativi moderni, attraverso tecniche che approfondiremo in questo capitolo.

È quindi fare un'importante distinzione tra due tipi di indirizzi. Lo spazio di indirizzi che è visto dal programma (e quindi è indipendente dalla macchina in cui risiede) è chiamato spazio degli indirizzi **logici** (o **virtuali**, come vedremo nel capitolo 7). D'altro canto, lo spazio degli indirizzi in memoria, ovvero gli indirizzi assoluti, è chiamato spazio degli indirizzi **fisici**.

6.1.3 Memory-Management Unit (MMU)

Parte dei meccanismo utilizzati per gestire la memoria e il binding non sono lasciati solamente al sistema operativo ma sono un ibrido tra hardware e software. Questo hardware è detto *Memory-Management Unit (MMU)* ed aiuta a mappare gli indirizzi logici in indirizzi fisici. Questo modulo era originariamente un chip a parte, tra la CPU e la memoria, con l'avanzare del tempo l'MMU è una parte integrata nella CPU stessa.

Una delle funzionalità dell'MMU è quella di fornire il controllo sull'indirizzo del processo nella memoria. È la stessa funzionalità che implementava il sistema operativo (vedi figura 41) solo che in questo caso il controllo è effettuato autonomamente dell'MMU attraverso un particolare registro chiamato **relocation register**. Il processo è rappresentato nell'illustrazione 43.

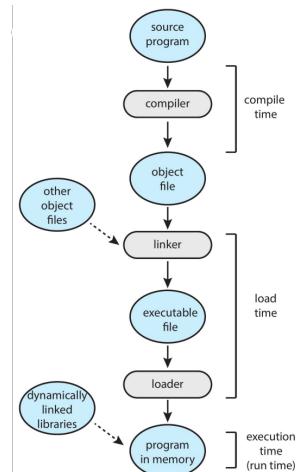


Figura 42: I tre tempi di binding.

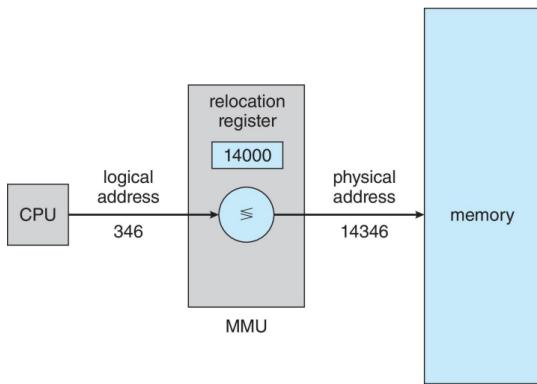


Figura 43: Procedura di controllo di un indirizzo mediante la MMU.

6.1.4 Caricamento e collegamento dinamico

Uno dei concetti fondamentali nell'uso della memoria è il **dynamic loading**. Ciò significa che dal punto di vista del programma non è possibile mantenere tutte le funzioni necessarie del programma sempre in memoria, ma mantenere solo un sottoinsieme che sono al momento necessarie. Nel momento in cui una nuova routine è necessaria la si va a caricare dalla memoria.

Un secondo approccio è quello del *linking*. In particolare il collegamento può essere fatto in due modi: statico e dinamico. Lo **static linking** avviene quando colleghiamo le librerie di sistema al programma e una volta compilato il programma le librerie sono immediatamente integrate nel file eseguibile. A questo si contrappone il **dynamic linking** dove il codice delle librerie non viene caricato nel file eseguibile ma le librerie sono collegate solo durante l'esecuzione del programma. Sono infatti presenti delle **shared libraries**, come le d11 di Windows, che vengono caricate in memoria e condivise a tutti i processi che le necessita e, un volta che nessun processo le utilizza più, queste vengono rimosse dalla memoria. Questo è molto più conveniente rispetto al collegamento statico dove in memoria ci possono essere potenzialmente diversi programmi che hanno una copia della libreria nel codice. Questo significa che la stessa libreria occupa molto più spazio in memoria perché è copiata da diversi processi. Se invece ci fosse stata una libreria dinamica, tutti i processi avrebbero usufruito il codice della stessa in modo tale da risparmiare dello spazio prezioso in memoria.

6.2 Primi modelli di allocazione

Passiamo ora a vedere la storia e i concetti della paginazione della memoria per poi arrivare a discutere dei metodi moderni e tuttora utilizzati.

6.2.1 Allocazione contigua

La prima soluzione che è stata pensata è l'allocazione contigua (figura 44) in memoria: ciò significa avere una parte dedicata al sistema operativo (kernel) e poi, in modo contiguo, inizia il codice per i processi.

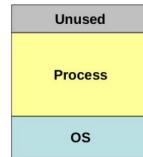


Figura 44: Allocazione contigua dei processi in memoria.

Il modo più semplice per poter implementare questo meccanismo è attraverso il *limit register* e il *relocation register* che abbiamo visto poco fa. Questo infatti permette di proteggere i processi tra di loro, evitando quindi che si sovrappongano in memoria. Osservando infatti l'illustrazione 45 possiamo notare che dalla CPU viene preso l'indirizzo logico (ovvero l'indirizzo relativo per il processo), viene controllato se "sfiora" il limite massimo in memoria; una volta che è stato appurato che l'*upper bound* non è stato superato, il relocation register procede con il *binding*.

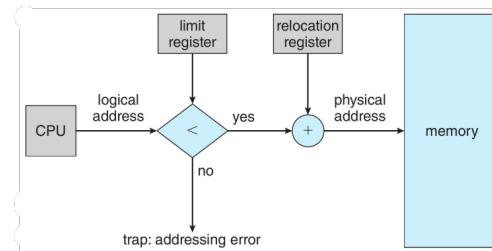


Figura 45: I registri utilizzati per il binding tra logico e fisico.

6.2.2 Allocazione a partizione fissa

Una soluzione un po' più raffinata, è l'allocazione a partizione fisse. Questo tipo di allocazione consiste nel dividere la memoria in un insieme di partizioni di dimensione non variabile e andare ad allocare i processi, non in modo contiguo, ma in una partizione la cui dimensione è adeguata e consona alla dimensione del processo. Osservando la figura 46 notiamo che, innanzitutto le partizioni possono essere di

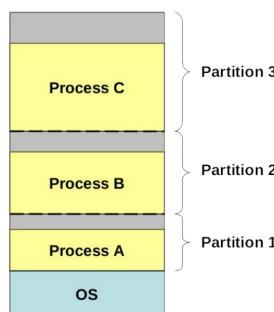


Figura 46: La memoria divisa in partizioni fisse per i processi.

dimensione diversa (l'importante è che non cambi nel tempo) al fine di gestire processi si dimensione diversa. Notiamo già i possibili problemi di questo tipo di modello, primo tra tutti è lo spazio in memoria inutilizzato grigio alla fine di ogni partizione (vedi il paragrafo 6.2.4: frammentazione interna). L'idea però della partizioni fisse è un concetto che, come vedremo verrà ripreso nella paginazione (paragrafo 6.3), presente nei sistemi operativi moderni.

6.2.3 Allocazione a partizione variabile

In alternativa alla partizione fissa, possiamo avere un tipo di allocazione in cui la dimensione della partizione varia a seconda del processo che fa richiesta. Osservando l'illustrazione 47 notiamo che il processo 8, una volta che è terminato lascia un buco ad un secondo processo, il 9 che però occupa meno spazio, di conseguenza la dimensione della partizione è variata. Anche in questo caso, come nella partizione fissa, si può

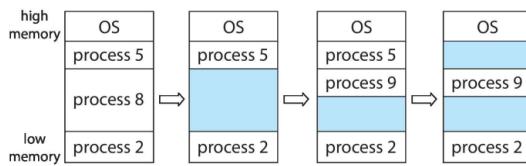


Figura 47: Rappresentazione di una partizione variabile in memoria.

incombere in una frammentazione interna (paragrafo 6.2.4) dato che una volta che il processo 9 ha occupato lo spazio rilasciato dal processo 8, è rimasto comunque un **bucco** in memoria, che non sempre può essere occupato da un altro processo in quanto, magari, non è presente abbastanza spazio contiguo.

Scelta della cella da allocare

Poniamo di essere in una situazione in cui abbiamo alcuni processi sparsi in memoria ed è necessario inserire un nuovo processo, come in figura 48. Quale posto scegliamo?

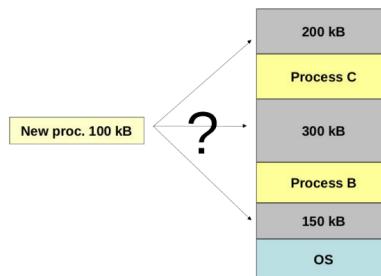


Figura 48: In quale allocazione di memoria verrà inserito il nuovo processo?

La scelta può essere effettuata secondo 3 criteri diversi:

- ◊ *First-fit*: il primo posto disponibile viene occupato dal nuovo processo. Per esempio, se abbiamo la memoria ad allocazione variabile e all'interno di essa

abbiamo un processo B e un processo C, si va a scorrere la memoria e si inserisce il nuovo processo nel primo *hole* disponibile;

- ◊ *Best-fit*: in questo caso si cerca di allocare il nuovo processo nella zona il più si avvicina alla dimensione del processo. Di conseguenza, al fine di allocare il processo in memoria è prima necessario scorrerla tutta con una ricerca **lineare**;
- ◊ *Worst-fit*: infine, con questa tecnica, si sceglie lo spazio con la dimensione peggiore possibile, ovvero la dimensione più grande possibile; ciò significa che il buco lasciato in memoria sarà il più grande possibile. Anche in questo caso è necessario fare una ricerca **lineare**.

6.2.4 Problema della frammentazione

Abbiamo ormai constatato che quando si allocano nuovi processi in memoria, questi lasciano in memoria dei buchi che non sempre possono essere riempiti. In particolare, nel caso della partizione fissa (6.2.2) questo tipo di problema viene chiamato **frammentazione interna**, questo perché è interna alla partizione in quanto il processo allocato non riesce a riempire pienamente lo spazio della partizione (osservare la figura 46). Questo rappresenta ovviamente un problema perché può essere che in memoria ci sia in totale dello spazio disponibile (ovvero la somma di tutti gli spazi grigi in figura) ma essendo non contiguo non permette un ulteriore inserimento del processo in memoria.

Alla frammentazione interna si contrappone la **frammentazione esterna**, che si verifica nel caso dell'allocazione a partizione variabile (6.2.3). Facendo appunto riferimento alla figura 47, può essere che una volta inserito il processo 9 all'interno della memoria si siano generati due buchi che non sono abbastanza grandi al fine di contenere un altro processo. Si è fatto uno studio dove si è notato che con l'allocazione a partizione variabile e il **first-fit**, ogni N blocchi in memoria, se ne perdono la metà a causa della frammentazione esterna: di conseguenza all'incirca 1/3 della memoria è inutilizzato (*50-percent rule*).

Una soluzione al problema della frammentazione esterna è il **compacting** (compattazione), illustrata in figura 49. Poniamo che il processo B termini e venga rimosso dalla memoria; si creerebbe un buco sopra e sotto il processo C che non consentirebbe a processi di grandi dimensione di essere inseriti all'interno della memoria. Allora,

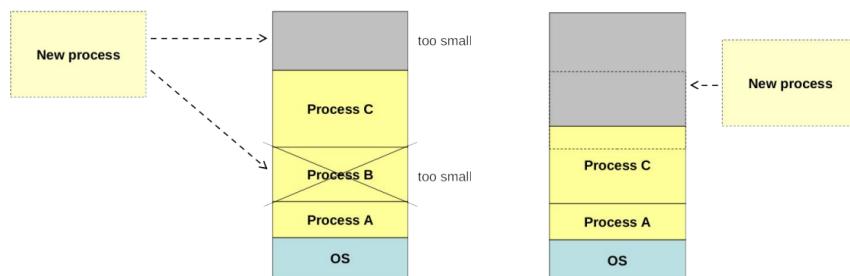


Figura 49: Processo di compacting, una soluzione alla frammentazione esterna.

al posto di lasciare così com'è la memoria, si sceglie di compattare tutti i processi nella parte bassa in memoria al fine di avere un'unica zona di memoria con soli processi (e senza buchi) e una parte in memoria completamente a disposizione dei

nuovi processi. Sembra che una soluzione ottimale ma in realtà comporta alcuni diversi problemi. Primo tra tutti è che bisogna capire come **spostare** dei processi in memoria: fino ad ora venivano inseriti e rimossi ma mai spostati da una zona all'altra. In secondo luogo, se si ripete questo processo per milioni di locazioni in memoria, compattare molti processi diventa un **problema computazionale** non indifferente, soprattutto se si fa ogni volta che un processo in memoria viene rimosso. È quindi evidente che la soluzione sia tutto fuorché ottimale e che quindi i problemi di frammentazione, con questo tipo di allocazioni, persistono.

6.3 Paginazione (*paging*)

Ecco quindi che introduciamo questo nuovo tipo di allocazione, molto più moderno ed elegante, ovvero la **paginazione**. Il concetto alla base di questa tecnica è quello di dividere la memoria in locazioni di dimensione fissa \bar{n} , chiamate **frames**, e dividere lo spazio logico dei processi in blocchi della stessa dimensione \bar{n} . Generalmente, nei sistemi operativi moderni, $\bar{n} \in [512 \text{ Bytes}, 16 \text{ MBytes}]$. In questo modo è quindi possibile dividere i processi in **pagine** e associare ad ogni pagina di ogni processo in esecuzione un frame in memoria, proprio come illustrato nella figura 50. Questa

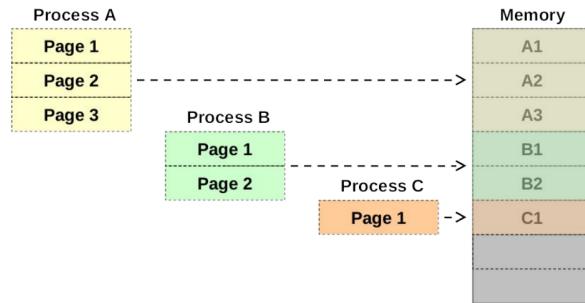


Figura 50: Il funzionamento ad alto livello della paginazione.

tecnica risolve subito il problema della frammentazione esterna e dei buchi. Osservando infatti la figura 51, nel momento in cui il processo B termina l'esecuzione e viene rimosso dalla memoria, dovrebbero rimanere dei buchi tra il processo A e il processo C. Questo buchi però vengono immediatamente colmati dal processo D le quali pagine vengono mappate nei frames che prima erano occupati da B e dal frame seguente al processo C. In questo modo, proprio perché le pagine e i frames sono della stessa dimensione si incastrano perfettamente tra di loro e non lasciano nessun buco.

Se andiamo a dividere lo spazio logico in pagine e la memoria viene divisa in frames, abbiamo bisogno di un supporto che mappa ciascuna pagina del processo ad un frame in memoria. Questo supporto è chiamato **page table** (discussa approfonditamente nel paragrafo 6.4) e si occupa appunto della traduzione di indirizzi logici (pagine) in indirizzi fisici (frames). Si osserva che la page table è una struttura associata ad ogni singolo processo. Di conseguenza due processi hanno due page table differenti.

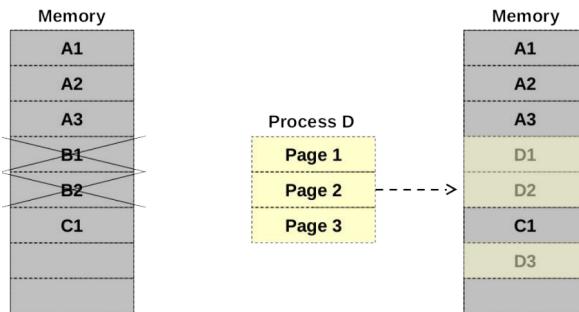


Figura 51: La rimozione di un processo e l'inserimento di un altro con la paginazione.

6.3.1 Frammentazione interna

Osserviamo infine un'ultima cosa: pur avendo eliminato il problema della frammentazione esterna, la **frammentazione interna** persiste: molto spesso infatti un processo non avrà esattamente la dimensione di n pagine, anzi, avrà generalmente una dimensione che è minore della dimensione occupata da n pagine ma comunque più grande di quella occupata da $n-1$ pagine. Di conseguenza l'ultima pagina non sarà mai completamente piena ma avrà dello spazio non occupato. Questo spazio non potrà nemmeno essere utilizzato da altri processi dato che come sappiamo la dimensione della pagina è fissa.

Facciamo un esempio per fissare il concetto. Abbiamo un processo di 10 KBytes che deve essere allocato in memoria. Poniamo che la dimensione delle pagine (e quindi dei *frames*) è di 4 KBytes. Al fine di inserire il processo in memoria sono quindi necessarie 3 pagine, che forniscono uno spazio di $3 * 4$ KBytes = 12 KBytes che è maggiore rispetto ai 10 KBytes del processo: sono quindi sprecati $12 - 10 = 2$ KBytes che non potranno mai essere occupati da altri processi. Al caso migliore lo spreco è di 1 Byte (ovvero la dimensione di 1 cella in memoria), mentre al caso peggiore lo spreco è di $\bar{n} - 1$ Bytes. È ragionevole affermare che invece al **caso medio** si spreca indicativamente la metà di un frame.

Una soluzione a cui si potrebbe intuitivamente pensare è quella di ridurre sempre di più la dimensione delle pagine al fine di sprecare sempre meno memoria. Riducendo però la dimensione delle pagine (e quindi dei frame) significa che abbiamo a che fare con un numero maggiore di pagine: se dimezziamo la dimensione di una pagina avremmo quindi il doppio delle pagine da indirizzare: è evidente che più si riduce la dimensione di un frame, la complessità aumenta. La soluzione migliore non è quindi avere delle pagine estremamente piccole ma riuscire a trovare un compromesso tra la dimensione della pagina e la complessità che ne deriva.

Calcolo della frammentazione interna

Un secondo esercizio che può ritornare utile è quello del calcolo della frammentazione interna durante la paginazione. Poniamo di avere la dimensione della pagina di 2048 Bytes (e quindi lo spazio per l'offset è di 11 bit) e che la dimensione del processo in esecuzione sia 72766 Bytes. Possiamo ora calcolare il numero di pagine occupate dal processo: $72766/2048 = 35.53$. Questo risultato ci dice che il numero

di pagine completamente riempite sono 35 e che la 36esima verrà utilizzata ma solo in parte generando quindi della frammentazione interna. I byte sprecati sono infatti $(36 \cdot 2048) - 72766 = 962$ bytes inutilizzati.

6.3.2 Traduzione degli indirizzi

Al fine di convertire le pagine del processo in frames in memoria è quindi necessario un processo di traduzione. Tale traduzione è effettuata attraverso un'interpretazione dell'indirizzo logico. In particolare, tale indirizzo è diviso in due parti: la prima parte (p), quella dei bit più significativi (ovvero quelli più a sinistra), indica l'indirizzo della pagina mentre la seconda parte (d) indica l'*offset* della cella dall'indirizzo della pagina. La page table si occupa di convertire il numero della pagina nel numero del frame in memoria, mantenendo l'offset invariato.

Osserviamo più nel dettaglio questo processo basandoci sull'illustrazione 52. Prima di tutto il processo fornisce l'indirizzo logico, che, come abbiamo appena visto, è composto da due parti. La parte più significativa, p , indica il numero della pagina, in particolare rappresenta il numero dell'indice della tabella dove è contenuto il frame. Una volta che l'accesso alla page table è stato effettuato, si preleva il frame f

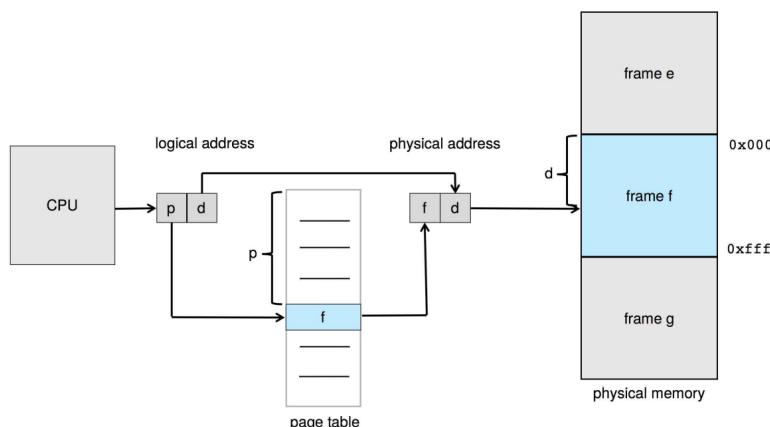


Figura 52: Il processo di conversione da indirizzo logico a indirizzo fisico.

e lo si sostituisce nell'indirizzo logico che diventa un indirizzo fisico. A questo punto non ci resta che andare in memoria nel frame f e aggiungerci l'offset con valore d .

Calcolo di un indirizzo tradotto

Supponiamo di avere lo spazio totale di indirizzo logico di 2^{16} , e quindi un indirizzo logico è 16 bit. Ci viene dato inoltre la grandezza di una pagina, che è 2^{12} . In altre parole, con questa informazione, sappiamo che i bit necessari per l'offset sono 12. A questo punto sappiamo che i bit disponibili per indirizzare le pagine logiche sono $16 - 12 = 4$; di conseguenza abbiamo $2^4 = 16$ pagine logiche disponibili.

A questo punti viene fornito l'indirizzo logico **0011 0000 1011 1001**. Poniamo inoltre che siamo a conoscenza di tutti i valori presenti nella page table del processo. In quale indirizzo fisico viene mappato l'indirizzo dato? La soluzione è molto semplice: l'indirizzo fornito è ovviamente lungo 16 bit; sappiamo inoltre che durante la

conversione vengono modificato solo i bit del numero della pagina, che in questo caso sono i primi quattro verso sinistra, ovvero **0011**. A questo punto scorriamo sulla page table alla cella numero $0011_2 = 3_{10}$ e scopriamo che la pagina viene mappata nel frame **1110**. Procediamo quindi con una banale sostituzione, ottenendo l'indirizzo fisico di valore: **1110 0000 1011 1001**. L'illustrazione 53 è esemplificativa di questo processo.

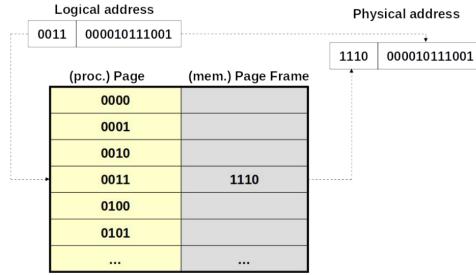


Figura 53: Esempio di una paginazione.

6.3.3 Allocazione nei frames liberi

Nel momento in cui si carica un nuovo processo in memoria, i frames che devono essere riempiti sono, naturalmente, quelli vuoti. Questi sono presenti in una lista (la **free-frame list**) che è detenuta dal sistema operativo al fine di tenere traccia dei frames disponibili. Di conseguenza quando deve essere inserita una pagina in memoria, il sistema operativo prende un frame disponibile dalla free-frame list che viene associato alla pagina. Questa associazione viene quindi inserita nella page table del processo. Nella figura 54 è illustrato questo processo. Inizialmente arriva un nuovo processo che contiene 4 pagine. Queste 4 pagine, in quanto nuove non hanno ad esse associato in frame, e di conseguenza non sono presenti nemmeno sulla tabella delle pagine. Di conseguenza, nel momento in cui la richiesta viene accettata

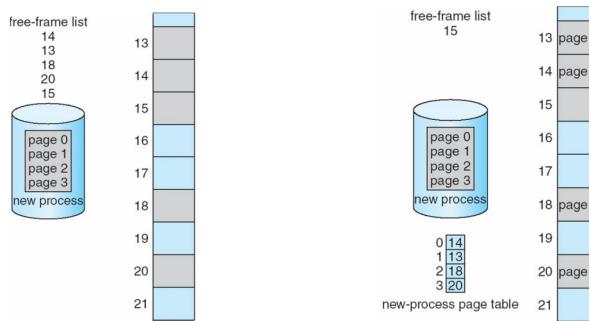


Figura 54: Il processo di allocazione delle pagine in nuovi frames liberi.

dal sistema operativo, alle 4 pagine vengono associati 4 frames liberi che erano presenti nella lista. Dopo l'allocazione infatti notiamo che i frame che erano liberi ora contengono le 4 pagine e non sono più presenti nella free-frame list. Ancora più

importante è però notare che nella pagina delle tabelle del processo si sono aggiunte le nuove associazioni.

6.4 Page table

Entriamo ora più nel dettaglio, e discutiamo in maniera più approfondita la, ormai nota, tabella delle pagine. Come sappiamo questa tabella non è unica in tutto il sistema operativo ma è presente una page table per ogni processo. Tipicamente la tabella ha dimensioni così significative che essa stessa è mantenuta in memoria. Quindi per ogni processo in esecuzione, esiste una sua page table associata che è presente in memoria. In particolare, della page table, il sistema operativo tiene traccia di due importanti valori (vedi figura 55), al fine di riuscire ad associare ogni processo alla sua tabella:

- ◊ *Page-Table Base Register (PTBR)*, che contiene l'indirizzo alla prima cella in memoria da cui parte la tabella;
- ◊ *Page-Table Length Register (PTLR)*, che si occupa di immagazzinare la lunghezza della tabella in memoria.

Ovviamente, nel momento in cui avviene un *context switch* (vedi paragrafo 1.2.1), i due registri vengono modificati. In questo modo si è in grado di associare ad ogni processo la sua personale page table.

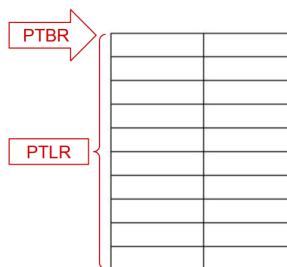


Figura 55: I due registri necessari per tenere traccia della page table in memoria.

6.4.1 Translation Look-aside Buffer (TLB)

Il problema di avere una tabella in memoria è che per accedere ad un indirizzo fisico in memoria, è necessario effettuare due accessi in memoria:

1. Accesso alla page table in memoria per tradurre l'indirizzo logico in un indirizzo fisico;
2. Accedere alla locazione dell'indirizzo fisico fornito dalla tabella delle pagine.

Così facendo il tempo per accedere ad un frame in memoria è raddoppiato. La soluzione a questo problema è fornita dalla **TLB**, che è una speciale **cache** la quale mantiene la porzione della page table che è usata più spesso. In questo modo la CPU non deve sempre accedere alla memoria bensì accede alla TLB che è molto più rapida e quindi consente di risparmiare tempo. Questa cache è infatti implementata fisicamente a livello di CPU e i suoi tempi di accesso sono estremamente ridotti

comparati a quelli della memoria. Si osserva inoltre che, se la velocità di accesso è elevata, lo spazio di memorizzazione è molto basso: generalmente infatti le TLB contengono dalle 64 alle 1024 righe.

Ora che abbiamo inserito la TLB nel processo di traduzione da pagina logica a fisica, vediamo come questo è cambiato. L'illustrazione 56 rappresenta l'intero processo di traduzione. Una volta che arriva l'indirizzo logico dalla CPU si andrà

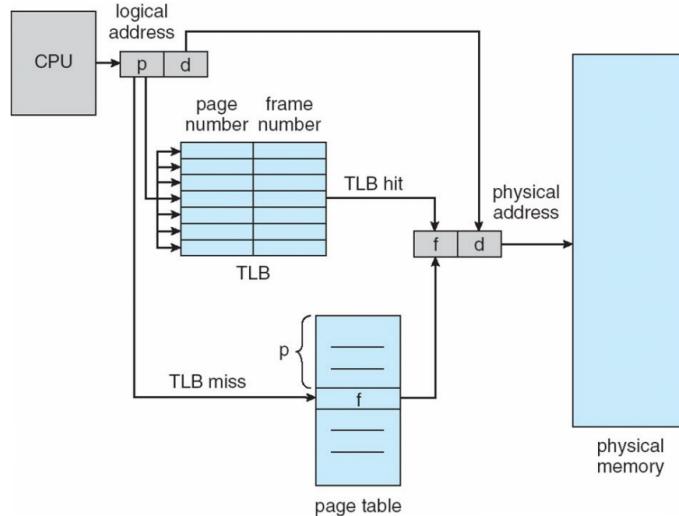


Figura 56: Il processo di traduzione effettuato con l'ausilio della TLB.

innanzitutto a controllare se il numero della pagina logica è presente all'interno della TLB. Nel caso sia presente, si ha un **TLB hit** e si ottiene subito il frame associato alla pagina; di conseguenza si può subito accedere in memoria. Si osserva che la verifica della pagina all'interno della TLB viene effettuata in **parallelo**, di conseguenza tale verifica è praticamente istantanea. Se invece la pagina non è presente nella TLB, si verifica il cosiddetto **TLB miss**. A questo punto è necessario effettuare un primo accesso in memoria al fine di trovare il frame nella page table e, dopo di che, effettuare un secondo accesso al frame allocato.

Va inoltre specificato che a differenza delle page table, la TLB è una unica per tutta il sistema ed è quindi "condivisa" da tutti i processi. Di conseguenza è presente un identificativo, chiamato *address-space identifier (ASID)*, che indica se la particolare entry della TLB fa parte o meno del processo che ne fa richiesta. Questo identificativo serve per fare in modo che i processi accedano solo nei loro indirizzi. Se questa funzionalità non fosse implementata, una soluzione comunque lecita è quella di resettare la TLB ogni volta che un context switch (1.2.1) si genererebbe però dell'overhead.

Si osserva infine che è anche possibile far sì che alcune entry rimangano permanentemente dentro la TLB in quanto, magari, sono delle pagine che vengono richieste quasi sempre. In questo caso si parla infatto di *wired down entries*.

Analisi delle performance con la TLB

Definiamo che **hit ratio** la percentuale secondo la quale si ha un TLB hit. Poniamo di avere un hit ratio di 80%: 80 volte su 100 si verificherà un TLB hit. Supponiamo

inoltre che il tempo di accesso alla memoria sia 10 ns (nanosecondi). Da queste informazioni possiamo evincere che se si ha un TLB hit, il tempo di accesso sarà 10 ns, altrimenti il tempo di accesso si duplicherà diventando 20 ns. A questo punto è possibile calcolare l'**EAT**, ovvero *Effective Access Time*:

$$EAT = 0.8 \cdot 10 + 0.2 \cdot 20 = 12 \text{ ns}$$

Passando però ad un hit ratio più realistico, 99%, il tempo effettivo di accesso diventa:

$$EAT = 0.99 \cdot 10 + 0.01 \cdot 20 = 10.1 \text{ ns}$$

6.4.2 Bit di validità

Uno dei meccanismi dedicati alla protezione della memoria è quello di implementare un bit di validità, chiamato **valid-invalid** bit. Alla page table viene aggiunta una colonna che contiene tale bit di validità. In questo modo, ogni entry della page table avrà un bit che indicherà se quell'associazione sia valida o meno. In particolare il bit si occupa di segnalare se la pagina si trova nello spazio di indirizzi logici del processo. Osservando la figura 57, notiamo che all'interno della page table le pagine 6 e 7 sono invalide proprio perché non sono presenti nello spazio di indirizzi del processo (che si ferma alla quinta pagina).

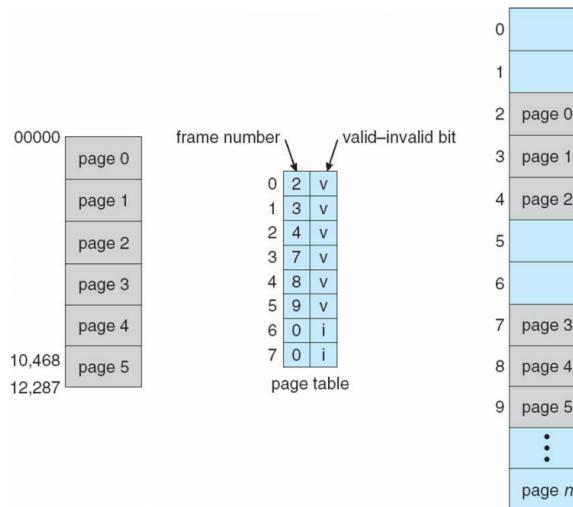


Figura 57: Implementazione del valid-invalid bit sulla tabella delle pagine.

Il problema delle macchine moderne

La page table inserita in maniera continua in memoria con l'ausilio della TLB sembra una soluzione ottima. E lo era un tempo, quando gli indirizzi erano al più di 16 bit. Con l'avvento però di indirizzi da 32 e 64 bit la situazione è però peggiorata perché la dimensione delle pagine è però sempre la stessa e non può cambiare. Poniamo di avere delle pagine da 4KB, che quindi necessitano di 12 bit per l'indirizzamento, rimangono $32 - 12 = 20$ bit per indirizzare tutte le pagine. Ciò significa che è possibile

avere fino a 2^{20} pagine da indirizzare, ovvero avere una tabella con più di un milione di righe. Nei casi ancora più moderni, quando si ha a che fare con indirizzi da 64 bit, si ha la possibilità di avere $2^{52} \approx 4 \cdot 10^{15}$ ovvero 4 biliardi di pagine.

Ricordiamo che alla base della paginazione c'è la necessità di non dover allocare in maniera continua nulla in memoria, di conseguenza è giusto che anche la page table abbia lo stesso trattamento. Ecco quindi che nascono diverse tecniche al fine di riuscire a spezzettare anche la page table all'interno della memoria.

6.4.3 Page table gerarchica

La prima soluzione che si è pensata è quella di creare una page table che indirizzi un altro numero di page tables che a loro volta indirizzano i frames in memoria, proprio come rappresentato nell'illustrazione 58. Come cambia ora il processo di

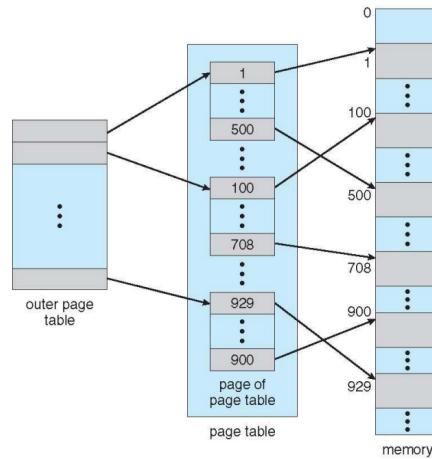


Figura 58: Struttura gerarchica della page table.

traduzione tra una pagina e un frame? Partiamo da un indirizzo a 32 bit e poniamo che le pagine abbiano la dimensione di 4 KB e che quindi necessitino di 12 bit per essere indirizzati (d). A questo punto i restanti 20 bit vengono a loro volta divisi in due parti: la prima parte p_1 punta ad un elemento della *outer page table* e la seconda parte p_2 punta ad un elemento della tabella puntata da p_1 (figura 59). In questo modo

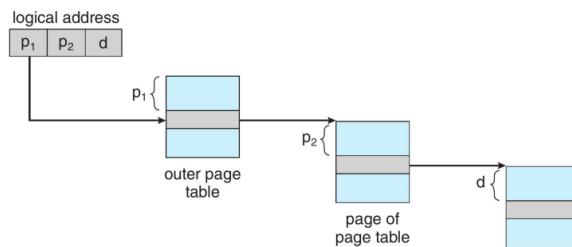


Figura 59: L'accesso in memoria attraverso una struttura gerarchica della page table. però puntualizziamo che, senza l'ausilio di una TLB cache, al fine di raggiungere un

frame in memoria è necessario fare ben 3 accessi ad essa, triplicando così il tempo necessario per accedere al frame.

Nelle architetture a 64 bit invece le soluzioni possono essere 2. Avendo 52 bit a disposizione si sceglie di creare una *outer page table* molto grande, indirizzata con 42 bit, e delle tabelle di secondo livello indirizzate da 10 bit. Una seconda soluzione può essere invece l'inserimento di un terzo livello di page table (**three-level paging scheme**) dove, per esempio, si ha l'*outer page table* indirizzata con 32 bit, e la tabella secondaria e terziaria indirizzate con 1 bit ciascuna. Si osserva che in questo caso il numero di accessi in memoria quadruplica al fine di accedere ad un frames. Proprio perché la struttura gerarchica non si presta molto bene ad architetture con indirizzi maggiori di 32 bit, si sono progettate altre soluzioni.

6.4.4 Page table con tabella hash

Al fine di provare a risolvere il problema che è emerso utilizzando una struttura gerarchica si è pensato di implementare una *hashed page table*: ciò significa convertire il numero della pagina attraverso una *hash function* che restituisce una chiave. Tale chiave è l'indice di una tabella che contiene il frame che stiamo cercando. Questo meccanismo è illustrato in modo chiaro nella figura 60 sottostante. La *hash table*, a

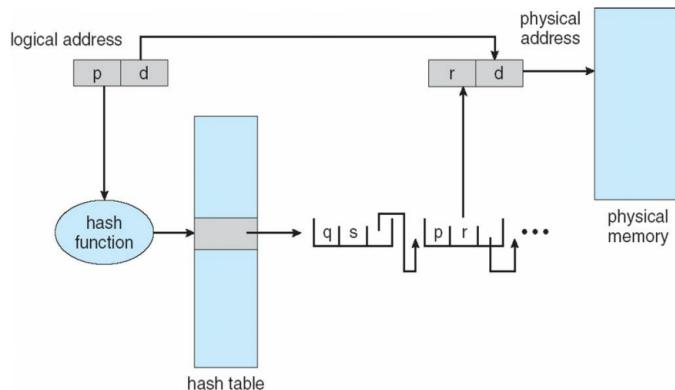


Figura 60: Accesso alla memoria tramite una page table implementata attraverso una hash table.

seconda del range delle chiavi fornito, può essere infatti di dimensione molto ridotta. Ricordiamo inoltre che la hash function per valori di pagine diverse può restituire le stesse chiavi: proprio per questo motivo ad ogni entry della hash table in realtà è presente una lista concatenata che contiene tutte le entry che hanno avuto una collisione. Nel caso della figura 60, il valore della pagina *p* era situato alla seconda posizione della lista.

Introduciamo anche la *clustered page tables* che è una struttura più moderna e migliorata rispetto alla *hashed page table*. In questa struttura, ogni entry della page table, non corrisponde ad una pagina bensì ad un **cluster** di frames, che può arrivare fino a 16. In questo modo è possibile fare riferimento a frames che possono essere sparsi all'interno della memoria. Così facendo i casi con frames sparsi in diverse locazioni sono più facilmente gestibili. Questa struttura è quindi più efficiente rispetto alla classica implementazione con la tabella hash.

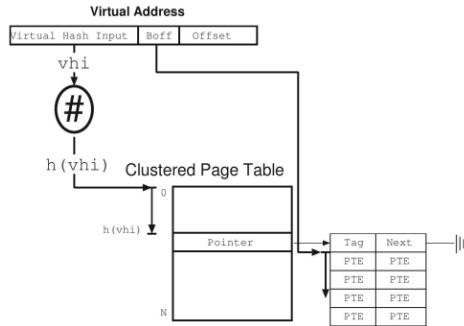


Figura 61: Rappresentazione di una *clustered* page table.

6.4.5 Page table invertita

Una terza soluzione è la cosiddetta tabella delle pagine invertita. Fino ad ora abbiamo detto che ogni processo ha la sua propria tabella delle pagine. Questo è necessario perché ogni processo ha il suo indirizzamento logico a cui può corrispondere un insieme di frames nella memoria. Il problema è che generalmente lo spazio di indirizzo logico non è mai utilizzato completamente. Questo perché tipicamente diverse entry all'interno della page table del processo sono invalide e quindi inutilizzate, ma sono comunque presenti.

Allora perché non fare l'opposto? Invece di avere tante tabelle, di notevoli dimensioni che puntano a frames fisici, si è scelto di utilizzare una **tabella unica** che è indicizzata dai frames fisici reali, e contiene all'interno di ogni elemento la pagina che il frame contiene. Ciò significa che al posto di avere tante tabelle, una per ogni processo, ci ritroviamo con un'unica tabella dove saranno contenuti i frames della memoria fisica e le pagine corrispondenti al processo. Inoltre, nella tabella, sarà necessario aggiungere, per ogni pagina, il numero identificativo del processo, il pid, visto nel capitolo 1, altrimenti non si riuscirebbe a risalire a quale processo appartiene quella pagina. Possono infatti essere presenti due pagine logiche **0110**, ma una appartiene al processo 314 e l'altra magari appartiene al processo 159.

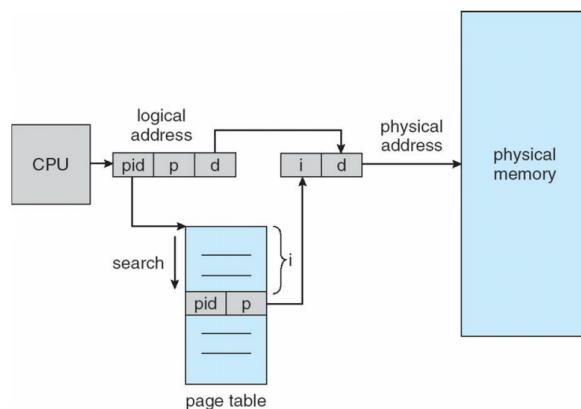


Figura 62: Accesso in memoria attraverso la tabella delle pagine invertita.

La figura 62 mostra il procedimento necessario per accedere ad un frame in memoria. Prima di tutto è necessario effettuare una **ricerca** sulla tabella invertita l'identificativo del processo (pid) e la pagina cercata. Se tale ricerca ha buon fine allora il frame in memoria è presente ed è quindi possibile effettuare l'accesso.

Discutiamo, infine, alcuni vantaggi e svantaggi della tabella delle pagine invertita. Il vantaggio più evidente è che lo spazio occupato dalla tabella invertita è nettamente minore rispetto alla somma dello spazio occupato da ciascuna tabella per i processi in esecuzione. D'altro canto però il tempo di **ricerca lineare** nella tabella invertita genera dell'overhead. Questo però può essere ridotto implementando una tabella di hash per limitare la ricerca all'interno della tabella invertita. È inoltre possibile migliorare le prestazioni tramite l'ausilio della TLB cache.

6.5 Swapping

Per quanto il metodo per paginare la memoria sia efficiente, ci saranno sempre dei casi in cui è necessario rendere dello spazio disponibile in memoria, rimuovendo uno o più processi. Introduciamo ora una tecnica che serve proprio per questo, si chiama *swapping* (su disco o, in generale, sul *backing store*), che è il procedimento secondo il quale si prende un processo dalla memoria e lo si salva temporaneamente sul disco rigido. Immaginiamo di avere in memoria 100 processi ed aver esaurito lo spazio. Per aggiungere il 101 esimo è necessario per forza rimuovere dei processi. Si sceglie infatti di prendere quei processi che sono in attesa (per esempio di un I/O) e che sono fermi in memoria e portarli sul disco, proprio come illustrato nella figura 63. Il tempo di *swap-in* e *swap-out* rimane comunque un tempo significativo dato che

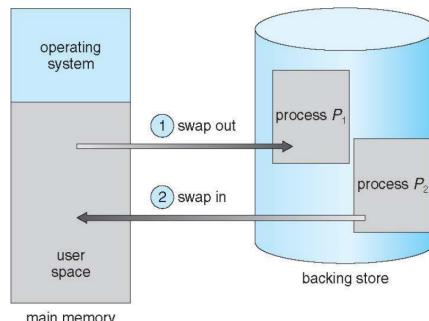


Figura 63: Processo di *swap-in* e *swap-out* dal *backing store*.

il *backing store* è una memoria ad alta capacità ma molto lenta. È quindi necessario tenere conto anche del **tempo di trasferimento** nel momento in cui si scambiano i processi. Ovviamente sarà presente anche una **ready-queue** che contiene tutti i processi presenti nel disco che non sono più in attesa di un evento e che sono quindi pronti per essere inseriti in memoria. Naturalmente al tempo di trasferimento del processo *in / out* dal disco va sommato anche il tempo necessario per effettuare il context switch (1.2.1), ovvero ricaricare lo stato del processo nella CPU.

Sono presenti anche delle varianti dello swapping, dove si va a scegliere quale processo portare fuori in base alla priorità: è un concetto molto simile allo scheduling con priorità che abbiamo visto nel paragrafo 3.4.

6.5.1 Swapping con paginazione

Lo swapping classico, tipicamente non è più utilizzato nei sistemi operativi moderni. Spesso infatti non si fa lo swapping dell'intero processo ma solamente di alcune pagine di quel processo. Ecco quindi che parliamo di *swapping with paging*, rappresentato in figura 64. La scelta delle pagina da rimpiazzare sarà delegata agli algoritmi di *page replacement* (vedi paragrafo 7.3).

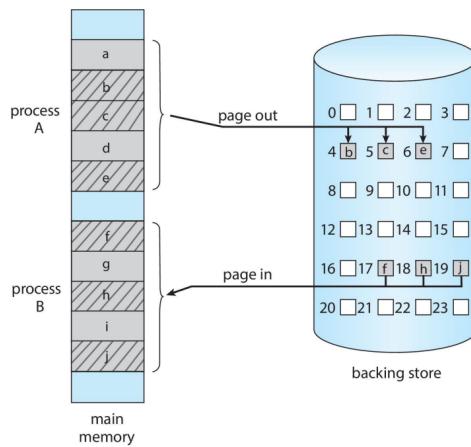


Figura 64: Rappresentazione del processo di swapping solo su determinata pagine di un processo.

6.5.2 Swapping nei dispositivi mobili

Nei sistemi operativi per *mobile devices* la questione è un po' più complicata in quanto effettuare uno swapping è molto dispendioso, sia da un punto di vista energetico, ma soprattutto perché i supporti di memorizzazione dei dispositivi mobili spesso hanno un numero limitato di accessi in scrittura e dopo di ché le performance diminuire. Di conseguenza si cerca sempre di evitare la scrittura sul backing store.

Sono presenti ovviamente dei metodi alternativi. Per esempio **iOS** al fine di liberare memoria domanda a dei processi in esecuzione se qualcuno può volontariamente rilasciare la memoria altrimenti può forzatamente terminare un processo in esecuzione. D'altra parte, **Android** sceglie di terminare applicazioni che generalmente sono dormienti o in background. Nel momento in cui l'applicazione è terminata lo stato dell'applicazione viene comunque memorizzato al fine di essere già pronta per ripartire.

6.6 Segmentazione (*segmentation*)

Introduciamo ora un nuovo modello di allocazione della memoria che è tuttora utilizzato ma non è tanto diffuso quanto la paginazione: stiamo parlando della segmentazione. Quando abbiamo parlato della paginazione, avevamo più volte specificato che la dimensione della pagina è fissa. Nel *segmentation* invece noi permettiamo che la dimensione sia variabile. Non parliamo delle pagine che hanno la stessa dimensione, ma di **segmenti** di dimensione variabile. Osservando la figura 65, notiamo che

i processi sono appunto divisi in segmenti di dimensione variabile. La dimensione

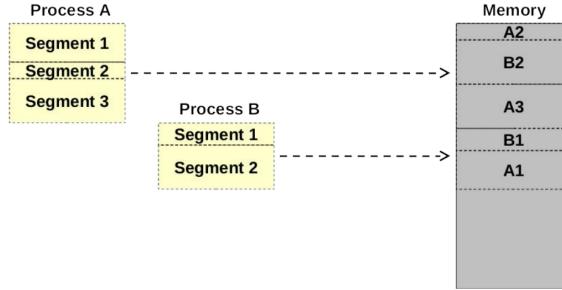


Figura 65: Il funzionamento ad alto livello della segmentazione.

di un segmento è decisa in base all'organizzazione e alla logica del programma stesso: potrebbe essere che alcuni segmenti contengono il `main`, altri contengono delle librerie, altri ancora che contengono le variabili.

6.6.1 Segment table

Al fine di tener traccia di questi segmenti non è più necessaria una tabella della pagine, bensì una **tabella dei segmenti**. Questa contiene tre campi:

1. I primi bit dell'indirizzo logico corrispondono al **numero di segmento**;
2. La **dimensione** del segmento che si trova in memoria;
3. L'indirizzo della prima cella del segmento, ovvero il **base address**.

Osserviamo la figura 66 e discutiamo nel dettaglio il processo di traduzione da un indirizzo logico ad uno fisico. Quando un indirizzo logico viene fornito, questo è diviso in due parti: il numero del segmento e l'offset. A questo punto si prende il

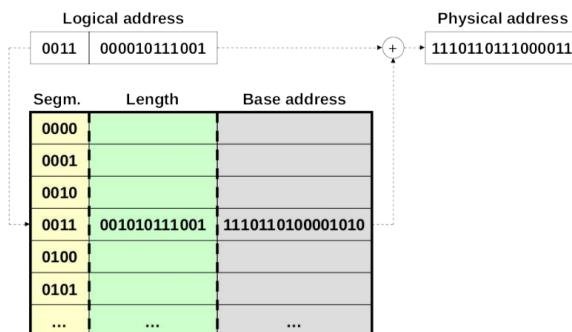


Figura 66: processo di traduzione di un indirizzo logico a fisico mediante la segment table.

numero del segmento, che è l'indice della segment table. A questo punto si controlla che l'offset sia minore della dimensione del segmento in memoria, in modo tale da garantire che il processo non invada celle di altri processi. Una volta che la verifica è andata a buon fine, se prende l'indirizzo base e lo si **somma** con l'offset in modo

da ottenere l'indirizzo in memoria del segmento. Si ricorda che nella paginazione il frame veniva concatenato con l'offset, qui invece si somma l'offset all'indirizzo base.

6.6.2 Modello ibrido

In molti casi è preferibile combinare la paginazione con la segmentazione. Si va così a creare un **page-segmented system**. Osservando la rappresentazione 67, cerchiamo di capire come questi tipi di sistemi funzionino. Innanzitutto l'indirizzo logico (virtuale) è diviso in tre parti: la prima indica la segment table, il secondo indica la page table e il terzo invece rappresenta l'offset. Inizialmente, quando arriva l'indirizzo logico, si accede alla segment table: al suo interno è contenuto l'indirizzo iniziale della page table. A questo punto, utilizzando la seconda parte dell'indirizzo logico

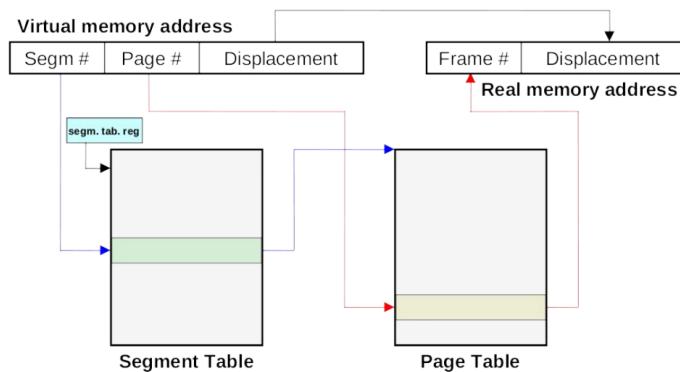


Figura 67: Rappresentazione di un modello che combina paginazione e segmentazione.

si accede alla riga della tabella che contiene l'indirizzo del frame in memoria. Questo frame va quindi concatenato con l'offset.

7 Memoria virtuale

In questo capitolo cerchiamo di capire com'è un sistema operativo che gestisce la memoria attraverso le tecniche che abbiamo visto nel capitolo precedente (6). Ci occuperemo prima di tutto del concetto di *demand paging* e alcune tecniche di ottimizzazione come il *copy-on-write*. In secondo luogo passeremo a discutere di tutti gli algoritmi di rimpiazzo delle pagine (*page replacement*) in memoria. Discuteremo infine il fenomeno del *thrashing* e di alcune soluzioni che adottano i moderni sistemi operativi.

7.1 Introduzione

Sappiamo che il codice, al fine di essere eseguito, deve essere presente nella memoria. Quando parliamo di memoria virtuale, parliamo infatti della separazione tra indirizzo logico e fisico. Fino ad ora abbiamo discusso di processi che, anche se divisi in pagine o in segmenti, venivano comunque caricati tutti in memoria. Da adesso introduciamo la possibilità di poter eseguire un processo anche se le sue pagine non sono tutte in memoria, anzi, possono risiedere sul file system (??) o sul *backing store*. Infatti quando un processo è eseguito, non è necessario che l'intero codice del programma sia in memoria, è sufficiente solo un sottoinsieme. Osservando infatti la figura 68, possiamo notare che le pagine nella memoria virtuale non sono tutte presenti nella memoria fisica, bensì alcune risiedono nei *backing store*.

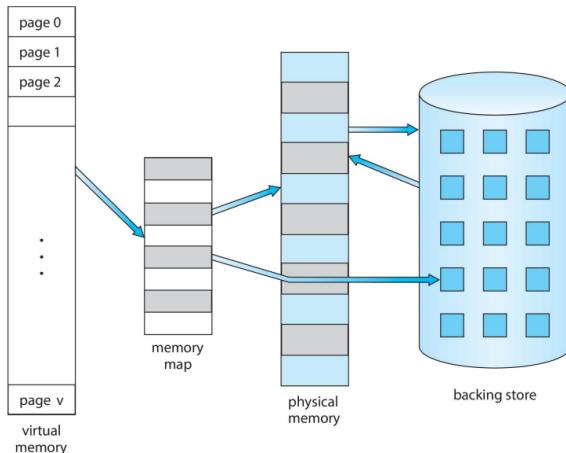


Figura 68: Alcune pagine del processo sono in memoria, altre sono sul disco.

7.1.1 Spazio degli indirizzi virtuali

Quello che nel capitolo precedente abbiamo definito come lo spazio degli indirizzi logici in realtà si chiama più propriamente spazio di indirizzi virtuali. Questo è lo spazio degli indirizzi che il programma vede: per ogni programma quindi questo spazio parte dall'indirizzo zero e consiste in un insieme di indirizzi continui. L'implementazione di questo spazio virtuale può essere effettuata attraverso delle tecniche di paginazione o segmentazione chiamate come *demand paging* e *demand segmentation*.

Il fatto di riuscire ad implementare uno spazio di indirizzi virtuali contigui ci permette di implementare delle strutture che dal punto di vista virtuale sono quelle viste dal processo (come la figura 1 nel capitolo 1).

7.1.2 Memoria condivisa

Tramite la dinamicità fornita dalla memoria virtuale è possibile mappare delle zone in memoria condivise e quindi avere dei riferimenti in processi diversi alla stessa zona in memoria (69). In queste zone di memoria possono essere condivise delle librerie,

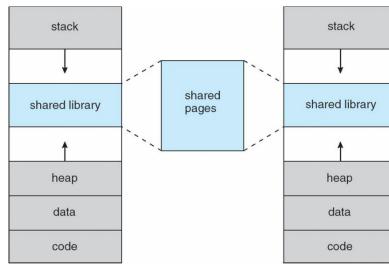


Figura 69: La memoria virtuale fornisce la possibilità di condividere delle librerie.

come nel caso delle dll, viste nel paragrafo 6.1.4, oppure utilizzare questa zona per della comunicazione tra processo, ovvero la IPC, *Inter Process Communication* (1.3).

7.2 Demand Paging

Al fine di essere in grado di trasformare tutti i *virtual address space* in indirizzi fisici caricando solo le pagine che vengono utilizzate e non tutte le pagine del processo si utilizza la tecnica del **demand paging**, illustrato in figura 70. Ciò significa che se abbiamo dei processi A e B che richiedono delle particolari pagine (contenenti codice o dati), si andranno a caricare quelle pagine su domanda dal *secondary storage* (come l'hard disk) alla memoria principale. Dovremo quindi essere in grado di gestire questa richiesta per caricare dalla memoria secondarie le pagine oppure per scaricare le pagine dalla memoria principale, ovvero la memoria RAM. Al fine di riuscirci, ci appoggeremo a dei supporti hardware come il **bit di reference** che indica se la pagina di cui abbiamo bisogno è in memoria o sullo storage secondario. Nel caso in cui la pagina si trova già in memoria si dice infatti che la pagina è **memory resident**, ovvero che è immediatamente disponibile.

7.2.1 Page fault

Per garantire che in memoria ci sia sempre il set di pagine richiesto si avrà bisogno del supporto hardware, in particolare si fa riferimento alle funzionalità della MMU. In particolare, questa fornisce la possibilità di settare il *valid/invalid bit* - nella tabella delle pagine - il quale indica se la pagina che andiamo a richiedere è già presente in memoria o meno. Se la pagina non è in memoria si genera un interrupt chiamato **page fault** che deve essere ovviamente gestito dal sistema operativo.

Osserviamo la figura 71. Abbiamo a che fare con un processo che richiede 8 pagine che però non sono presenti tutte in memoria: dalla *page table* possiamo notare

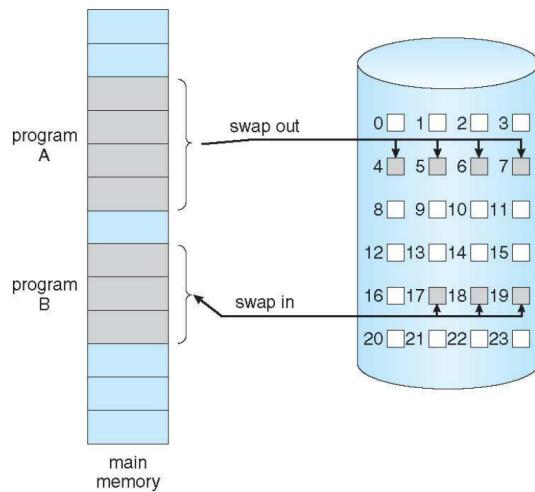


Figura 70: Illustrazione del funzionamento del demand paging nel caso di due processi, A e B.

che la pagina A è in memoria e si trova nel frame 4, la pagina C è associata al frame 6 e infine la pagina F è associata al frame 9. Tutte le altre pagine non sono presenti

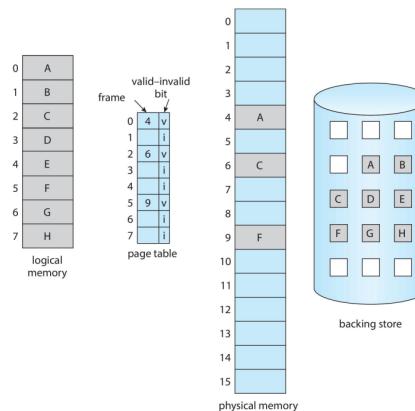


Figura 71: Generazione di un page fault.

in memoria e quindi il loro *invalid bit* è impostato a "v". Per quanto riguarda le altre pagine invece si verificano dei *page fault* dato che non si trovano in memoria ma solo nel *backing store*: possiamo infatti notare che l'*invalid bit* di queste pagine è settato a "i".

Come si comporta il sistema operativo adesso? Basandoci sull'illustrazione 72, possiamo osservare che ci sono 6 steps da compiere. Dopo aver richiesto la pagina (punto 1) e, mediante la page table, aver constatato che non è presente in memoria (punto 2), il sistema operativo cattura il page fault e va a cercare il frame sulla memoria secondaria (punto 3). Assumendo che ci siano sempre degli spazi liberi in

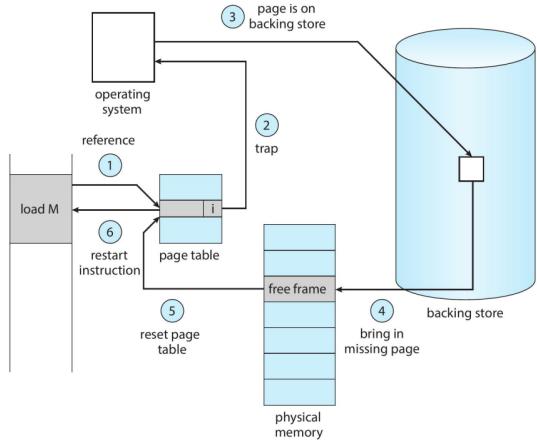


Figura 72: Gli steps necessari per prelevare la pagina dallo storage secondario

memoria, il sistema operativo ne sceglie uno e inserisce il frame che ha trovato sul *backing store* (punto 4). A questo punto aggiorna l'*invalid bit* sulla *page table* e specifica in quale frame si trova la pagina virtuale (punto 5). Infine l'istruzione viene fatta ripartire (punto 6): in questo modo l'istruzione può essere eseguita e il processo può continuare.

7.2.2 Pure demand paging

Appena iniziamo l'esecuzione di un processo, questo deve ancora essere caricato in memoria. Ebbene, il demand paging puro consiste nel caricare solo le pagine del processo che vengono richieste. In altre parole il processo viene caricato in memoria frammento dopo frammento. Questo, per quanto istintivo possa sembrare, è in realtà estremamente **inefficiente**: significherebbe aumentare generare un numero elevatissimo di *page fault* e di conseguenza il sistema operativo dovrebbe passare per il *backing store* molte volte. Fortunatamente però i processi godono di una proprietà chiamata **locality of reference**: questo significa che la maggior parte del codice e dei dati di un processo è raggruppata tutta assieme e non sono sparpagliati nella memoria.

Un approccio sicuramente più efficiente è quello di cercare di caricare il maggior numero di pagine possibili in memoria al fine di minimizzare il numero di *page fault* e quindi di accessi alla memoria secondaria.

Gestione della memoria libera

Fino ad ora abbiamo assunto che ci siano sempre dei frame liberi in memoria ma naturalmente non è così (vedi paragrafo 7.3). Il sistema operativo possiede quindi una lista dei frame che sono liberi, la cosiddetta **free-frame list**, che viene utilizzata per identificare dove poter inserire la nuova pagina. Nel caso non ci siano elementi liberi è necessario liberare dello spazio in memoria attraverso gli algoritmi di rimpiazzo (paragrafo 7.3).

Nel momento in cui dei frame vengono liberati tramite questi algoritmi va ad azzerare il contenuto utilizzato precedentemente, in modo tale da garantire che

il nuovo processo non vada a leggere dati che appartengono al vecchio processo. Questa tecnica è chiamata **zero-fill-on-demand**.

7.2.3 Performance e ottimizzazione

Come abbiamo visto, nel momento in cui viene catturato un page fault il tempo perso a sostituire la pagina desiderata è notevole. Ci sono molte cause che portano le tempistiche ad essere così elevate però, generalmente le principali sono le tre seguenti:

1. La *routine* che gestisce il page fault;
2. Il tempo per leggere la pagina dal disco;
3. Il tempo necessario per far ripartire il processo (ricaricare i dati e i registri nei momenti in cui la pagina è in memoria).

Per valutare le performance del demand paging ci si basa su una variabile chiamata **page fault rate**, che è la frequenza con cui un page fault avviene. Quando questa vale 0, non si verificano mai questi interrupt, quando è 1 si verificano ad ogni nuova esecuzione che viene eseguita. Ovviamente, il nostro obiettivo è quello di minimizzare il più possibile questo fattore. Questo valore lo utilizziamo per calcolare l'**EAT** (*Effective Access Time*), ovvero il tempo di accesso reale, effettivo. Sia p il *page fault rate*, per calcolare l'EAT si utilizza la seguente formula:

$$(1 - p) \cdot \text{Memory Access Time} + \\ p \cdot (\text{Page fault Overhead} + \text{Swap Page Out} + \text{Swap Page In})$$

Al fine di riuscire a diminuire l'EAT si può cercare di ottimizzare alcuni aspetti. Si può cercare di diminuire il tempo di *swap-out* e di *swap-in* dei frame nella memoria oppure si può cercare di copiare l'intero processo sullo swap space. Si possono sfruttare anche altre proprietà dei processi: se le pagine che abbiamo in memoria che devono essere sostituite non sono state modificate è sufficiente eliminare le pagine al posto di andarle a salvare sul disco (tanto sono uguali).

Infine, una tecnica molto importante ed ingegnosa è quella del **Copy-On-Write** (figura 73) dove, nel momento in cui un processo padre viene *forkato* (vedi 1.2.2), il processo figlio condivide le stesse pagine del padre: è molto più efficiente che copiare le stesse pagine due volte in memoria. Le pagine continuano ad essere condivise fino a che tali pagine non devono essere modificate: ecco perché si chiama in questo modo, una pagina viene copiata solo nel momento in cui si deve scrivere su di essa.

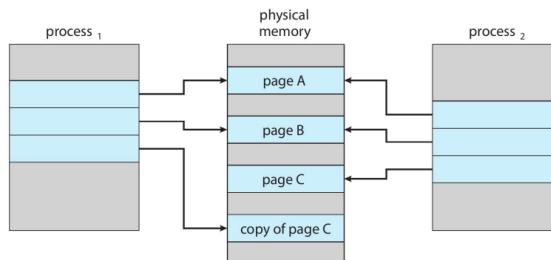


Figura 73: Dato che P1 deve scrivere sulla pagina C, questa viene copiata.

7.2.4 Prepaging

Controposto al *pure demand paging*, dove, dopo aver caricato il processo all'inizio, si caricano le pagine che necessita attraverso i page fault. Ovviamente, c'è anche la possibilità di non caricare solamente una pagina del processo, bensì caricare un insieme di pagine al fine di diminuire la frequenza di page fault e quindi risparmiare più tempo

7.3 Page replacement

Fino ad ora abbiamo dato per scontato che ci fosse sempre dello spazio libero in memoria. Spesso invece non è così e il sistema operativo si trova nella condizione di dover rimuovere una pagina dalla memoria per sostituirla con quella che è stata appena richiesta. Come scegliere la pagina da scartare? È proprio questo il compito di un **algoritmo di rimpiazzo**. Attraverso i *replacement algorithms* ben studiati è quindi possibile avere una grande memoria virtuale in una memoria fisica di dimensioni ridotte.

Nella pratica le operazioni generali di un algoritmo di rimpiazzo sono (vedi anche figura 74):

1. Selezionare un frame vittima dalla memoria e portarlo nel *backing store* (*swap out*);
2. Cambiare il bit nella page table da *valid* a *invalid*;
3. Copiare la nuova pagina dallo storage secondario alla memoria *swap-in*;
4. Aggiornare l'*invalid* bit nella page table a *valid*.

Osserviamo anche che in questi casi è aggiunto un altro bit, chiamato **dirty bit** che segnala se una pagina in memoria è stata modificata o meno. Di conseguenza se la pagina non è stata modificata viene rimossa, altrimenti deve essere copiata nella memoria secondaria.

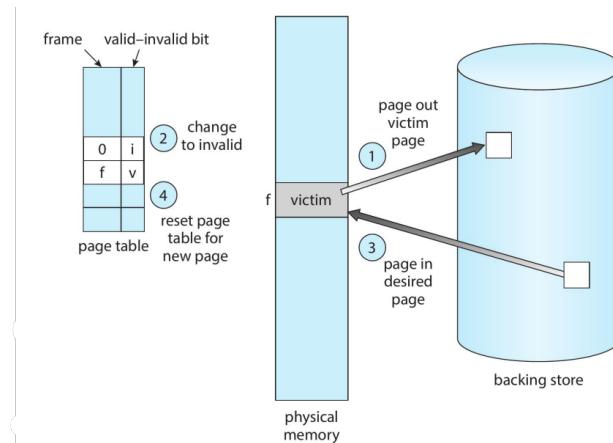


Figura 74: Funzionamento generico di un algoritmo di rimpiazzo.

Ricordiamo che un algoritmo di *page replacement* deve tenere conto di diversi aspetti. Innanzitutto bisogna capire quanti e quali frame devono essere assegnati

per ciascun processo? È meglio avere una distribuzione equa oppure proporzionato all'importanza o alla priorità che hanno? In secondo luogo il compito di un algoritmo di rimpiazzo rimane quello di **ridurre** il **page fault** rate.

7.3.1 FIFO

Il primo algoritmo che vediamo, come per lo scheduling e per altre occasione, è un algoritmo simile ad una coda: *First In First Out*. In altre parole la vittima che si sceglie è il frame che è stato inserito in memoria meno recentemente.

Prendiamo in considerazione la stringa $7 \ 0 \ 1 \ 2 \ 0 \ 3 \ 0 \ 4 \ 2 \ 3 \ 0 \ 3 \ 3 \ 2 \ 1 \ 2 \ 0 \ 1 \ 7 \ 0 \ 1$, dove ogni numero rappresenta la pagina che si sta richiedendo. Supponiamo inoltre, per semplicità, che la nostra memoria sia in grado di contenere solamente 3 frames. Simuliamo le richieste e contiamo il numero di *page fault* che si sono verificati durante le richieste. Commentiamo il comportamento di questo algoritmo

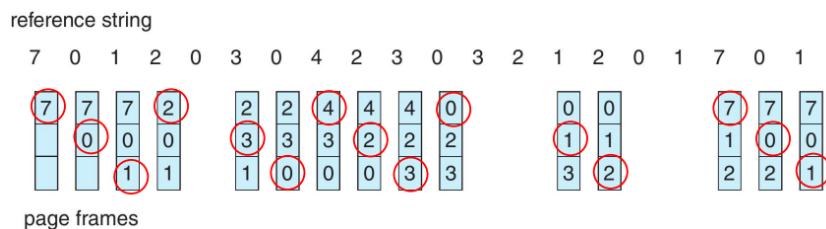


Figura 75: Comportamento dell'algoritmo di rimpiazzo FIFO.

basandoci sulla figura 75. Con le prime 3 pagine si verifica un *page fault* in quanto la memoria da vuota deve riempirsi. Al quarto *timestamp*⁴, quando si ha bisogno della pagina 2, questa non è presente in memoria, si verifica un *page fault* e si scambia pagina 2 con pagina 7 questo perché, tra le tre pagine presenti, è stata la prima ad entrare e quindi è la più "vecchia". Osserviamo che al quinto *timestamp*, si richiede la pagina 0 che però è già presente in memoria e quindi non si verifica alcun *page fault*. Proseguendo così per tutta la stringa possiamo notare che su 20 richieste si verificano 15 *page fault*.

Osserviamo un avvenimento curioso, chiamato **Belady's anomaly**. In modo contro-intuitivo, all'aggiungere di più frames si incrementa il numero di *page fault* anziché andarla a diminuire. Dando un'occhiata alla figura 76 notiamo passando da 3 a 4 frames il numero di *page fault* cresce.

7.3.2 Algoritmo ottimale

Il problema dell'algoritmo FIFO, oltre all'anomalia, è che nel momento in cui rimuoveva un frame dalla memoria non andava a domandarsi quanto questo frame è stato utilizzato e quindi senza tenere conto della storia dei frame. A livello teorico, un algoritmo ottimale rimuoverebbe il frame che verrà utilizzato il più tardi possibile nel futuro. Questo algoritmo è ovviamente utopico dato che conosce tutta la stringa. Nella realtà, ovviamente, non si è in grado di prevedere il futuro però si è in grado di fare delle buone stime.

⁴Con *timestamp* si intende l'istante di tempo che stiamo esaminando.

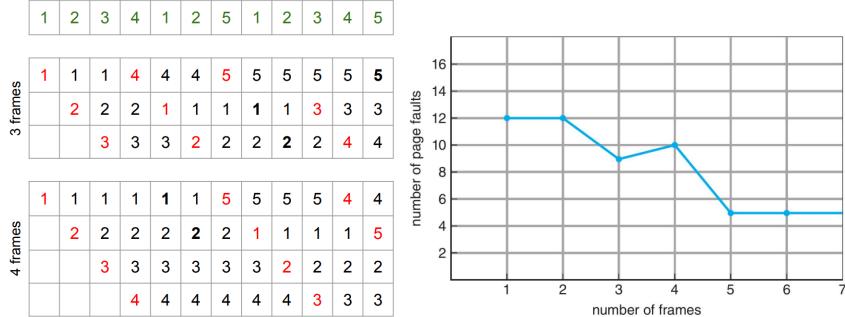


Figura 76: Una piccola illustrazione dell'anomalia di Belady.

Riprendiamo in considerazione la stringa 7 0 1 2 0 3 0 4 2 3 0 3 3 2 1 2 0 1 7 0 1 e cerchiamo di capire come l'algoritmo ottimale rimpiazzi i frames all'interno della memoria (figura 77). Dall'immagine notiamo che le prime tre richieste sono page

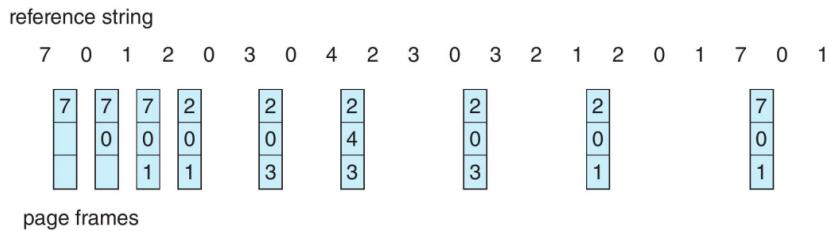


Figura 77: Funzionamento teorica dell'algoritmo ottimale.

fault dato che la memoria da vuota deve riempirsi. Al quarto *timestamp* notiamo che anche in questo caso la pagina 7 viene rimossa perché è quella che viene utilizzata più tardi (verso la fine praticamente). Dopo di che, alla richiesta della pagina 0 non viene rimosso nulla perché verrà utilizzata nel futuro prossimo (più precisamente dopo due *timestamp*). Proseguendo, quando viene richiesta la pagina 3 viene rimossa la pagina 1 dato che rispetto alla pagina 2 e 0 è quella che verrà utilizzata più tardi. Osserviamo che dalle 15 page fault dell'algoritmo FIFO riusciamo ad arrivare a 9.

Dato che questo algoritmo non è realistico, questo serve come **caso limite** o caso ideale verso cui vogliamo che i nostri algoritmi vertano. È impossibile raggiungerci però ci si cerca di avvicinare il più possibile. Inoltre, sappiamo che un algoritmo, basandosi su questa stringa, non potrà avere meno di 9 page fault in quanto il limite è proprio stabilito dall'algoritmo ottimale.

7.3.3 LRU

Dato che non è possibile guardare nel futuro, questo algoritmo guarda il passato, la storia degli utilizzi delle pagine e fa una stima per cercare di capire quale saranno le prossime richieste. È proprio questo il funzionamento dell'algoritmo **LRU**, che sta per *Least Recently Used*, ovvero il frame che è stato utilizzato meno tra quelli presenti nella memoria.

Prendiamo ancora una volta in considerazione la stringa $7 \ 0 \ 1 \ 2 \ 0 \ 3 \ 0 \ 4 \ 2 \ 3 \ 0 \ 3 \ 3 \ 2 \ 1 \ 2 \ 0 \ 1 \ 7 \ 0 \ 1$ e vediamo come questo algoritmo si comporta (figura 78). Come nei due casi precedenti, le prime quattro richieste sono page fault e la quinta

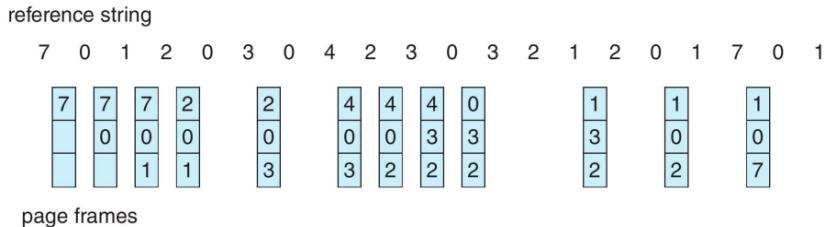


Figura 78: Funzionamento dell'algoritmo LRU.

no. Al sesto *timestamp*, quando viene richiesta la pagina 3 l'algoritmo LRU sceglie di scartare il frame 1, questo perché è stato quello utilizzato meno recentemente rispetto a 0 e 2. A questo punto, la richiesta della pagina 0 non produce alcun page fault però la richiesta successiva, quella per la pagina 4 genera un interrupt che provoca la rimozione della pagina 2 in quanto è stata quella utilizzata meno rispetto alle pagine 0 e 3. Procedendo in questa maniera scopriamo che il numero di page fault è 12, che sono 3 in meno rispetto all'algoritmo FIFO ma rimangono 3 in più rispetto all'algoritmo ottimale.

Com'è possibile implementare questo tipo di algoritmo? Non necessariamente abbiamo a che fare con dei *timestamp* ma possiamo utilizzare un **contatore** mantenuto non dal sistema operativo bensì dall'hardware (come l'MMU), che incrementa il contatore ogni volta che c'è una reference a quella pagina. In questo modo, per scegliere quale pagina deve essere rimpiazzata, si controlla questo contatore e la pagina con il valore minore verrà scartata. Una seconda tecnica per implementare l'algoritmo LRU è attraverso uno **stack**: ogni volta che una pagina viene utilizzata viene impilata in cima allo stack e di conseguenza nel momento in cui si deve scegliere una pagina da rimpiazzare, quella che si troverà nel *bottom* dello stack verrà scelta e scartata. Osserviamo che la seconda soluzione, a differenza del contatore, è una soluzione software piuttosto che hardware.

7.3.4 Second-chance (clock)

Questo algoritmo si basa sull'utilizzo di una **lista circolare** che contiene i riferimenti alle pagine che sono presenti in memoria e ogni qualvolta che di deve rimpiazzare una pagina in memoria con una nuova pagina appena richiesta si va a scorrere questa lista circolare. Ad ogni pagina caricata in memoria è associato un **bit**. Questo bit è impostato come segue:

- ◊ Il bit viene impostato a 1 ogni volta che la pagina in memoria è utilizzata dal processo;
- ◊ Il bit è impostato da 1 a 0 nel momento in cui la pagina presente non coincide con la pagina richiesta;

L'algoritmo quindi, nel momento in cui è necessario effettuare un rimpiazzo, scorre la lista fino a che non trova un bit a zero, e la pagina associata a quel bit viene rimossa. Tutti i bit a 1 che l'algoritmo ha incontrato prima di arrivare allo 0 sono impostati a

zero. In questo modo, se la pagina non viene utilizzata a breve, al prossimo controllo dell'algoritmo questa verrà rimossa dalla memoria.

Facendo quindi riferimento alla figura 79, osserviamo che in memoria sono presenti 6 pagine in memoria. In particolare osserviamo che tutte le pagine tranne la numero 3 hanno il bit impostato ad 1: ciò significa che tutte le pagine, eccetto la terza, nel momento in cui è necessario rimpiazzare una pagina l'algoritmo parte

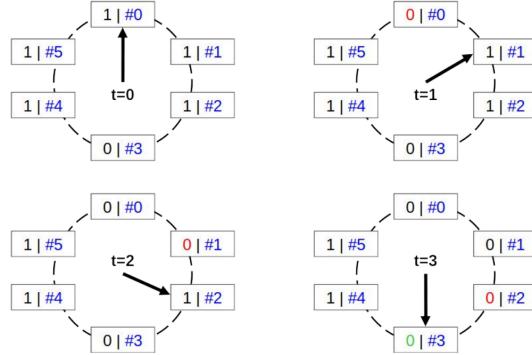


Figura 79: Il funzionamento del second-chance algorithm.

dalla pagina 0 e comincia a scorrere. La pagina 0 è stata utilizzata poco prima perché il bit è a 1: di conseguenza l'algoritmo imposta il bit a zero e procede alla pagina seguente. Sia per la pagina 1 che per la pagina 2 l'algoritmo le lascia in memoria ma il bit diventa zero. Alla pagina numero 3 però il bit è già a zero, di conseguenza tale pagina verrà rimossa al fine di lasciare spazio a quelle nuove.

È infine possibile migliorare questo approccio inserendo un secondo bit, chiamato **modify bit** che indica se la pagina in memoria è stata modificata o meno. Si creano quindi quattro combinazioni possibili, e in base a queste l'algoritmo effettua delle azioni sulla pagina (reference, modify):

- ◊ $(0, 0)$ indica che la pagina non è stata utilizzata recentemente e nemmeno modificata: è quindi la scelta migliore.
- ◊ $(0, 1)$ segnala che la pagina non è stata utilizzata di recente ma è stata modificata, di conseguenza, nel momento in cui la si rimpiazza, è necessario salvare le modifiche nel backing store.
- ◊ $(1, 0)$ indica che la pagina non è stata modificata ma è stata utilizzata di recente.
- ◊ $(1, 1)$ indica che la pagina è sia stata utilizzata di recente che modificata.

7.3.5 Algoritmo counting

Oltre all'algoritmo LRU sono presenti anche altri tipi di approcci che si basano sul numero di utilizzi di una pagina. Le altre due opzioni che vediamo sono algoritmi che vanno a contare il numero di referenze fatte ad ogni singola pagina. A differenza quindi dell'algoritmo second-chance non si ha un bit che indica se la pagina è stata utilizzata di recente ma si ha un vero e proprio **contatore** che tiene traccia del numero di referenze alla determinata pagina in memoria.

Con questa informazione si possono quindi andare a rimuovere le pagine che sono state usate di meno, come nel caso dell'algoritmo **LFU** (*Least Frequently Used*). Con questo approccio però può capitare che una pagina rimanga molto in memoria, anche se inutilizzata: poniamo il caso di una pagina nuova che entra in memoria e viene usata molto spesso. Questo significa che il contatore cresce subito e rimane alto. Anche se la pagina dopo molto tempo non è utilizzata, l'algoritmo non la rimuoverà mai dato che il contatore è molto alto. Si è quindi deciso di avere un check periodico dove si va a dimezzare il valore del contatore alle pagine che non sono utilizzate molto. In questo modo, prima o poi quella pagina in memoria se continua ad essere inutilizzata verrà rimossa appena il contatore diventa sufficientemente piccolo.

Un secondo approccio è quello opposto al LFU: stiamo infatti parlando del **MFU** (*Most Frequently Used*), dove si va a rimuovere la pagina che ha il maggior numero di utilizzi.

7.3.6 Ottimizzazione

Oltre alle performance che si ottengono attraverso un algoritmo di page replacement si può cercare di ottimizzare l'algoritmo anche in altri momenti durante la procedura di rimpiazzo di una pagina. Per esempio è possibile implementare un insieme di **frame sempre liberi** (una sorta di buffer) al fine di allocare subito la pagina richiesta nel momento in cui si verifichino un page fault. Solo dopo aver soddisfatto la richiesta ci si preoccupa di rimuovere una pagina dalla memoria. In questo modo il processo viene immediatamente servito e non viene sprecato tempo prezioso al fine di scambiare le pagine dalla memoria la backing store.

È possibile inoltre implementare una **lista delle pagine modificate**. Oltre al modify bit (7.3.4), questa scelta è opportuna in quanto le pagine modificate contenute al suo interno verranno periodicamente salvate all'interno del backing store. In questo modo, una volta che sono state salvate la lista sarà vuota e il modify bit ritornerà a zero.

7.4 Allocazione dei frames

Facciamo ora anche qualche considerazione riguardante il numero minimo e massimo di frames da allocare a ogni singolo processo. Ci sono diversi modi di allocare i frames per ciascun processo: poniamo di far partire 5 processi e che abbiamo a disposizione 100 frames. Possiamo scegliere di allocare i frames in maniera **uniforme**, dove ogni processo ha a disposizione 20 frames oppure, eventualmente, allocargli 15 per processo e tenere una *pool* di 25 frames liberi. Altrimenti è possibile effettuare un'**allocazione proporzionale**: il numero di frames allocati per processo dipende dalla dimensione e dalla priorità di quest'ultimo. È infatti ragionevole che un processo di grandi dimensioni richieda un numero maggiore di frames rispetto ad un processo di dimensioni minori.

7.4.1 Allocazione globale e locale

Quando andiamo a scegliere il frame da liberare al fine di fare spazio a nuove pagine, si possono scegliere due approcci.

Con l'approccio **globale** il frame da liberare viene scelto tra tutti i frame compresi in memoria, non importa a quale processo appartiene. In generale questo approccio è più performante rispetto al secondo e per questo è più utilizzato.

Con l'approccio **locale** invece, per ogni processo viene definito un insieme di frames da dove è possibile andare a pescare. In questo caso invece le performance sono ridotte in quanto è possibile che ci siano frame liberi al di fuori dell'insieme scelto per il processo ma questi non potranno essere usati proprio perché non compresi nel frame. In questo caso però il vantaggio è che è più consistente nei tempi di risposta dato che sono presenti dei limiti ben definiti per allocare i frames.

7.4.2 Richiesta delle pagine

Al fine di riuscire ad implementare il *global page replacement*, ovvero le classiche politiche di rimpiazzo, è quello di impostare dei limiti inferiori e superiori (*thresholds*) al numero di frames liberi in memoria. Osservando la figura 80, osserviamo che

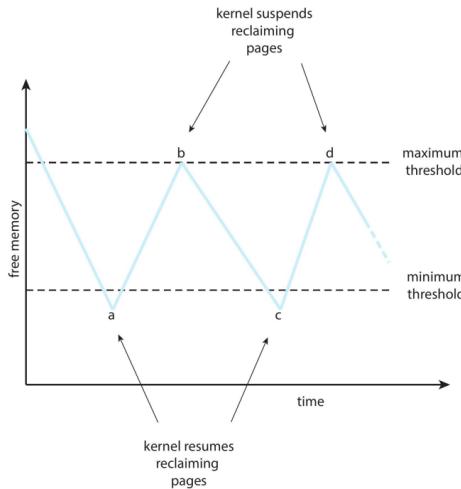


Figura 80: Questa tecnica assicura la presenza di frames liberi in memoria al fine di soddisfare eventuali nuove richieste.

nel momento in cui il numero di frame disponibili scende sotto il limite inferiore, il sistema operativo esegue un algoritmo di page replacement al fine di liberare abbastanza frames da raggiungere il limite superiore. Nel caso in cui il sistema operativo non riesca a liberare la memoria, può scegliere di cambiare algoritmo di page replacement, magari scegliendone uno più *strict*, oppure scegliere proprio di terminare qualche processo, potenzialmente quelli che consumano più memoria.

7.5 Thrashing

Il thrashing è un aspetto importante, soprattutto perché diversi metodi, ormai superati, di gestione della memoria, non sono riusciti a gestire questo problema. La figura 81 contiene uno schema dove viene mostrato l'utilizzo della CPU in relazione al numero di processi presenti in memoria. Teoricamente l'obiettivo è quello di aumentare il numero di processi il più possibile al fine di aumentare l'utilizzo della CPU

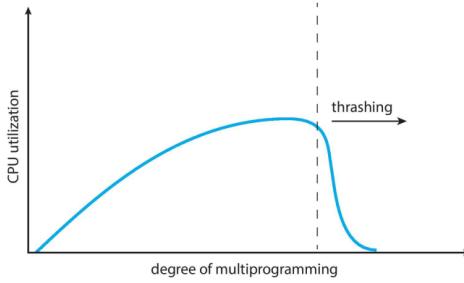


Figura 81: Il fenomeno del thrashing.

il più possibile. Notiamo in realtà che ad un certo punto l'utilizzo della CPU cala drasticamente. Questo generalmente capita quando il numero di processi in memoria è molto alto e quindi il numero di frames disponibili per processo diminuisce. Di conseguenza aumenta il numero di *page fault* (7.2.1) e quindi è necessario effettuare uno *swapping* (6.5). Lo swapping però non è effettuato dal processore, di conseguenza l'utilizzo della CPU diminuisce. Se l'utilizzo diminuisce allora il sistema operativo pensa che il processore sia pronto ad eseguire altri processi. Questi processi vengono quindi caricati in memoria, però non ci sono frames disponibili per loro e di conseguenza è necessario rimpiazzare delle pagine mediante uno swapping. Si entra quindi in un loop dove la CPU non viene utilizzata e tutti i processi non vengono eseguiti dato che si devono sviappare tra di loro.

7.5.1 Modello Working-set

Al fine di cercare di evitare fenomeni di thrashing si può utilizzare una caratteristica che hanno i programmi, chiamata **locality model**. Questa caratteristica dei programmi dice che tipicamente questi sono strutturati in modo tale che istruzioni e dati risiedi, sono raccolti, in zone di memoria **adiacenti**. Cerchiamo quindi di trovare un apporccio che sfrutti al meglio questo principio di località dei programmi.

Iniziamo definendo con Δ la finestra temporale dove avviene un certo numero di riferimenti alle pagine. Chiamiamo inoltre **WS** in *working-set*, ovvero l'insieme di pagine che sono utilizzate all'interno di Δ (vedi figura 82, dove la window è di 10 accessi). Definiamo ora un'altra variabile, la *Working-set size*, **WSS**, che rappresenta

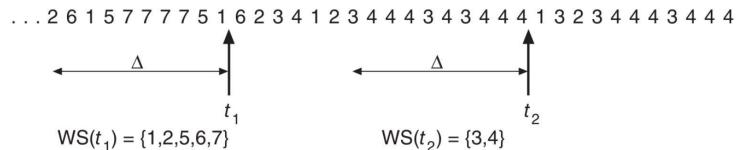


Figura 82: Il working-set.

il numero di pagine utilizzate durante la finestra Δ . In particolare possiamo dire che $WSS(t_1) = 5$ e che $WSS(t_2) = 2$. Con queste informazioni possiamo constatare che:

- ◊ Se Δ è molto piccolo allora non sarà possibile comprendere tutte le pagine che fanno parte della locality;

- ◊ Se Δ è molto grande è molto probabile che verranno comprese, oltre alla locality del processo, anche locality che non sono attive, andando quindi a sprecare memoria.
- ◊ Se $\Delta \rightarrow \infty$ si andrebbero a considerare tutte le pagine del processo.

Chiamiamo ora con $D = \sum WSS_i$, ovvero la somma della dimensione dei working set di tutti i processi. Se D è maggiore del numero dei frames disponibili allora si è in una situazione di thrashing. Ogni sistema operativo tiene traccia di questi WSS_i al fine di evitare situazioni di thrashing, di conseguenza se vede che ci si sta per avvicinare al thrashing si occupa di terminare o mettere in pausa un processo.

7.5.2 Frequenza dei page fault (PFF)

Un'alternativa alla WSS è quella di andare a manipolare la frequenza dei page fault. Questo approccio, a differenza del WSS offre una soluzione meno drastica e cerca di prevenire il verificarsi del thrashing (figura 83). In particolare, se la frequenza dei page fault è molto alta significa che ci sono diversi processi che stanno accedendo alla memoria, di conseguenza è necessario ridurre il numero di processi. Viceversa,

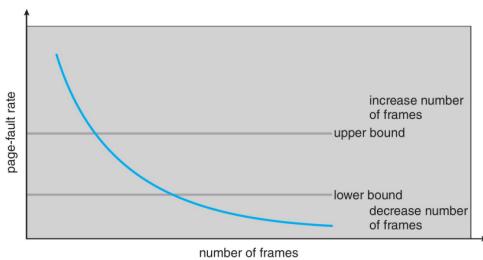


Figura 83: Il controllo della frequenza dei page fault

se la frequenza di page fault è molto bassa, ciò significa che è possibile andare ad aggiungere processi in memoria.

Legame tra WS e PFF

Come è possibile vedere anche dalla figura 84, esiste una relazione tra il working-set e il page fault rate. Notiamo che sono presenti dei picchi di frequenza che piano piano si riducono. Questi picchi rappresentano il cambiamento di locality: quando il processo cambia da una locality ad un'altra, per un certo tempo si genereranno page fault fino a che tutti i frames necessari sono caricati in memoria.

7.6 Allocare la memoria del kernel

Fino ad ora abbiamo visto la gestione della memoria in **modalità utente**. Ovviamente sono presenti anche dei modi dedicati al fine di allocare la memoria da parte del kernel del sistema operativo. Essendo infatti una parte molto importante e delicata, l'accesso in memoria non può essere rallentato a causa di altri processi: ha infatti un accesso privilegiato ed efficiente alla memoria. Sono presenti due strategie al fine di riuscire ad allocare la memoria da parte del kernel: il *buddy system* e la *slab allocation*.

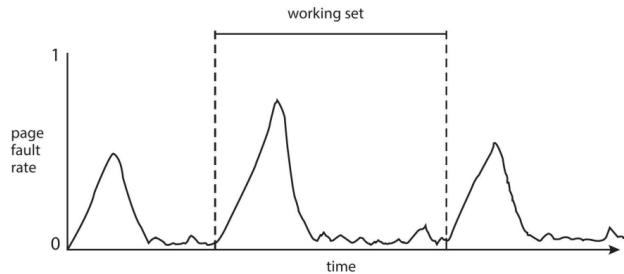


Figura 84: La relazione tra working-set e page fault rate

7.6.1 Buddy system

Con questo approccio la memoria è allocata da un segmento di dimensione fissata che consiste di pagine contigue (figura 85). La memoria viene allocata utilizzando il **power-of-2-allocator**: se, per esempio, abbiamo 256KB di segmento in memoria

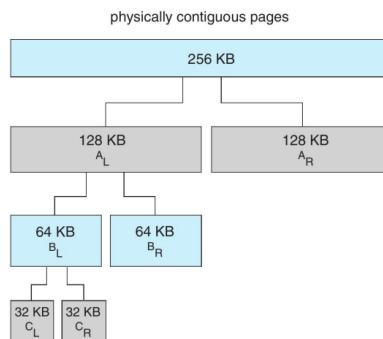


Figura 85: L'approccio di allocazione buddy system con l'allocatore *power-of-2-allocator*.

e il kernel ne richiede 31KB, i 26KB vengono divisi per due fino a che non si arriva al blocco ideale per la richiesta effettuata dal kernel. Se la richiesta fosse stata 33KB, si sarebbe dovuta allocare in un blocco da 64KB. Lo svantaggio sta proprio qui, dove lo spazio in memoria raramente è utilizzato ottimamente in quanto tipicamente le richieste del kernel non sono potenze di due: è il famoso fenomeno della frammentazione interna (6.2.4).

7.6.2 Slab allocation

Il secondo approccio è l'allocazione a "lastre". Questo metodo raggruppa delle pagine in memoria fisicamente contigue in **slab**; inoltre, gruppi di slabs costituiscono una **cache** (figura 86). Ad ogni struttura dati che il kernel utilizza è associata una cache. Sarà quindi presente una cache per gestire la lista dei processi (lista dei PCB 1.1.1), una cache per gestire la lista delle pagine e così via. Una volta che il kernel definisce le strutture che vengono usate, vengono allocate le cache volte a contenerle

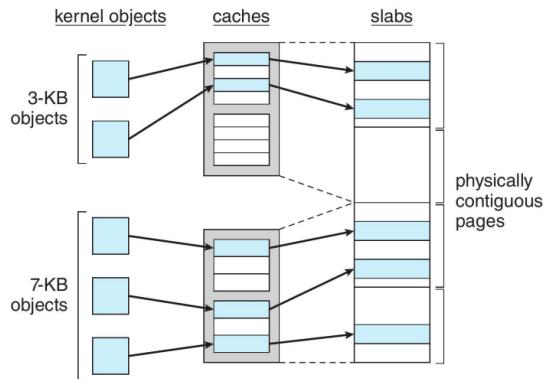


Figura 86: La slab allocation del kernel.

ed, eventualmente, ne incrementa la dimensione nel caso in cui il numero di istanze nelle strutture dati aumenti. Uno dei benefici di questa tecnica è che non si verifica la frammentazione ma comunque l'accesso rimane veloce.

Parte IV

Gestione dello storage

8 Memoria di massa

In questo capitolo iniziamo ad occuparci della gestione della memoria secondaria. Discuteremo innanzitutto il tipo di periferiche con il quale andremo a lavorare per poi passare ad alcuni algoritmi per lo scheduling e l'utilizzo ottimale di tali periferiche. Passeremo poi ad approfondire alcuni aspetti sulla gestione di questi dispositivi e del loro spazio di archiviazione (in particolare dello *swapping*, discusso già nel paragrafo 6.5). Infine discuteremo su alcuni metodi per l'immagazzinamento di molti dati, come le strutture RAID.

8.1 Tipi di memoria secondaria

Iniziamo con il distinguere due tipi di supporti di dati più comuni: stiamo parlando dei dischi rigidi, chiamati anche HDD, e delle schede basate su una tecnologia moderna chiamate NVM

8.1.1 HDD

L'HDD, che sta per *Hard Disk Device*, è quello che comunemente viene chiamato **disco rigido**. Questa unità è composta da diversi dischi magnetici, uno sopra l'altro. Facendo riferimento all'immagine 87, possiamo notare che ogni disco è formato da delle **tracce**, ovvero il cerchio a distanza \bar{R} dal centro. Ogni traccia è composta da diversi **settori**. L'insieme delle tracce a distanza \bar{R} dal centro del disco è detto **cilindro**. Al fine di riuscire a leggere le informazioni dei dischi, questi girano e una "testina"

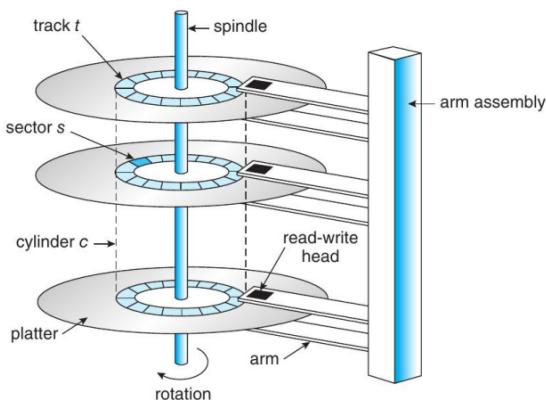


Figura 87: Schema ad alto livello di un disco rigido.

(*arm*, ce n'è una per ogni disco) si appoggia sulla traccia e legge l'informazione.

Essendo uno spostamento fisico, la lettura dal disco è abbastanza lenta in quanto è necessario che la testina raggiunga la traccia del disco necessaria, a questo punto

il disco gira fino a che non si è raggiunta l'informazione cercata. Il tempo di posizionamento sulla traccia (o settore) è detto **seek time** mentre il tempo dovuto alla rotazione del disco è detto **rotational latency**. Possiamo quindi cercare di calcolare la performance di un HDD attraverso delle medie aritmetiche:

- ◊ *average access time*: il tempo medio di accesso si calcola sommando l'*average seek time* e l'*average latency*;
- ◊ *average I/O time* si calcola sommando il valore trovato nel punto precedente a al tempo di trasferimento e all'*overhead* di un potenziale controllore elettrico/pe-riferica: $\text{AVG_Access_Time} + \frac{\text{Amount_to_Transfer}}{\text{Transfer_rate}} + \text{Controller_Overhead}$.

Esercizio

Prendiamo in considerazione la situazione di dover trasferire un blocco da 4KB su un disco che gira a 7200 RPM (*rotation per minute*) con un *average seek time* di 5 millisecondi e un *transfer rate* di 1Gigabit/sec con un *overhead* del controllore di 0.1 millisecondi. Seguendo la formula del secondo punto, dobbiamo trovare *average Access Time* e *Transfer Time* dato che il *Controller Overhead* già lo conosciamo.

- ◊ *Average Access Time* = *average Seek Time* (=5ms) + *Average Latency*. Per calcolare quest'ultima si fa: $\frac{1}{2} \cdot \frac{60}{7200} \cdot 10^3 = 10^3 \cdot \frac{30}{7200}$. Di conseguenza l'*Average Access Time* diventa: $5 + 4.17\text{ms} = 9.17\text{ms}$.
- ◊ *Transfer Time* = $\frac{\text{Amount to Transfer}}{\text{Transfer rate}} = \frac{4\text{KB}}{1\text{Gb/sec}} = \frac{4\text{KB}}{\frac{1}{8}\text{GB/sec}} = \frac{32\text{KB}}{1024^2\text{KB/sec}} = 0.031\text{ms}$

A questo punto sommiamo i due risultati all'*overhead* e otteniamo 9.301ms.

8.1.2 NVM

I dispositivi NVM (*NonVolatile Memory*, ovvero memoria non volatile) sono anche chiamati **SSD**, che sta per *Solid-State Disk*. Questi dispositivi hanno lo stesso compito dei dischi rigidi, ovvero quello di immagazzinare grandi moli di dati. In questo caso cambia il metodo di implementazione di tali dispositivi: non sono infatti presenti parti meccaniche e quindi sono molto più veloci nell'accesso e quindi anche più affidabili nell'**accesso random**.



Questa velocità molto elevata va però a scapito sia della capacità di *storage* che anche della durabilità del dispositivo. Questo perché le SSD sono molto veloci in lettura ma presentano enormi difficoltà e complicazioni in scrittura. Non è infatti possibile sovrascrivere un'informazione ma è possibile solo cancellarla e poi scrivere la nuova informazione. Inoltre la cancellazione è effettuata per **blocchi** di pagine e non pagine singole, di conseguenza, per sovrascrivere una pagina è necessario cancellare l'intero blocco e riscriverlo con la pagina modificata correttamente. La cancellazione inoltre usura il dispositivo: ecco che si cerca di misurare la "vita" di una SSD attraverso i **DWPD** (*Drive Writes Per Day*), ovvero le scritture effettuate in un giorno.

Generalmente ci si trova in una situazione dove in un blocco di pagine sono presenti dei dati che sono validi e dei dati che non lo sono (come in figura 88). Alcuni dati infatti saranno aggiornati mentre degli altri dati sono più vecchi e non hanno

valid page	valid page	invalid page	invalid page
invalid page	valid page	invalid page	valid page

Figura 88: Un blocco con delle pagine valide e delle pagine non valide.

più valore, di conseguenza sono dei candidati per essere rimpiazzati nel caso in cui fosse necessario dello spazio: è quindi presente un algoritmo di **garbage collection** che permette proprio di rimuovere delle pagine datate. Cosa succede però se tutti i blocchi hanno una o più pagine scritte? È necessario di un buffer dove il garbage collector sposta le pagine temporaneamente in modo da liberare lo spazio per poi ricopiare le pagine valide. Tipicamente una parte della NVM (circa il 20%) è lasciata a disposizione del buffer per il garbage collector (*overprovisioning*). Nel momento in cui arriva una nuova richiesta in scrittura e non c'è spazio, il garbage collector andrà a prendere le pagine e salvarle nel buffer, andrà a cancellare il blocco per poi ricopiare le pagine nuove nel blocco.

8.1.3 Memoria volatile

Sono presenti casi, dove al fine di implementare dei file systems, sono necessario le memoria volatili, ovvero memoria che non mantiene salvati i dati quando manca l'alimentazione (ovvero quando il computer si spegne). Stiamo parlando dei **RAM drives**, che sono implementati attraverso memorie mantenute in vita solo quando l'alimentazione è disponibile. Tali memorie possono essere utilizzate per varie operazioni sfruttando il tempo di accesso molto rapido, come l'utilizzo di file temporanei.

8.1.4 Nastro magnetico

Un'altra tecnologia meno comune nei computer di tutti i giorni ma che comunque rimane importante per altri dispositivi, come i server, è il cosiddetto nastro magnetico. Questa è un'altra forma di memorizzazione su cui grandi moli di dati vengono immagazzinati, soprattutto per fare un **backup**. In questi nastri l'accesso ai dati è effettuato in modo **sequenziale**: se dobbiamo accedere ad una particolare zona del nastro è necessario scorrere tutto il nastro fino a che non si raggiunge la zona desiderata. È ovviamente molto inefficiente per un accesso di tipo random ma è efficiente per accessi sequenziali; proprio per questo motivo è ottimo per i backup.

8.1.5 Dispositivi di memorizzazione esterna

Discutiamo ora, brevemente, altre periferiche le quali possono essere collegate temporaneamente attraverso dei connettori (dei *bus*) al computer. Le periferiche più

famosi sono l'ATA, in particolare il SATA. Per invece per quanto riguarda i dispositivi NVM la loro velocità di trasferimento ha stimolato la creazione di connettori con standard più veloci

8.2 Indirizzamento

Nonostante le tecnologie sono implementate diversamente l'una dall'altra, dal punto di vista del computer i dispositivi sono molti simili, anche se i dispositivi fisici sono completamente diversi. Indipendentemente dal fatto che un file venga salvato su una SSD oppure su un HDD, questo verrà salvato sia su una periferica che sull'altra in quanto il computer non nota differenze tra un dispositivo e l'altro. È un concetto molto simile all'indirizzo logico e fisico per i processi, dove questi ultimi vedevano solo i loro indirizzi relativi senza sapere quale spazio in memoria sarebbe stato occupato. Analogamente il sistema operativo salva il file all'interno del dispositivo senza sapere come questo sia fatto. Sarà infatti compito del dispositivo tradurre i segnali dal sistema operativo in operazioni da effettuare nella specifica periferica.

Per esempio, la scrittura su un HDD potrebbe essere rappresentata logicamente come la scrittura su un array unico. Questo array è definito inserendo in maniera contigua tutti i cilindri di una traccia per poi proseguire con il cilindro della traccia più interna. IN questo modo tutti i dati presenti sui vari livelli del disco rigido sono rappresentati mediante un array.

Avere una conoscenza di come il dispositivo funziona è comunque utile per il sistema operativo al fine di riuscire ad ottimizzare le operazioni da effettuare sulla periferica di archiviazione. Queste informazioni sono salvata nella **LBA**, ovvero il *Logical Block Address*.

8.3 HDD scheduling

Tipicamente non c'è mai solo un processo che vuole accedere al disco ma ce ne sono di diversi. Di conseguenza il sistema operativo mantiene delle code che contengono i processi in attesa per l'accesso al disco. Nel momento in cui sono presenti delle code di attesa, sicuramente saranno presenti degli algoritmi che scelgono quale processo andrà ad effettuare l'accesso al disco. In questo caso si può sfruttare il fatto che spesso indirizzi LBA vicini corrispondono anche a indirizzi fisici vicini tra loro.

8.3.1 FCFS

Il primo algoritmo che andiamo a vedere è, di consueto, un algoritmo FIFO. Infatti, proprio come nel CPU scheduling (3), FCFS sta per *First Come First Served*. Con questo algoritmo si ha quindi una coda FIFO dove il primo processo che entra nella coda (e quindi fa la richiesta di accesso) sarà anche il primo ad uscire. Osserviamo la figura 89 e ipotizziamo che arrivi una serie di richieste che corrisponda ad andare nella traccia 98, poi nella traccia 183 e così via fino alla traccia 67. Ipotizziamo di partire dalla traccia 53: andare a soddisfare le richieste in ordine FIFO significa spostare la testina lungo il disco e spostarsi da una traccia all'altra. Possiamo notare subito che questo algoritmo non tiene conto in alcun modo della prossimità delle tracce tra di loro, ma si basa solamente sull'ordine di entrata. Calcolando il numero di tracce che ha dovuto scorrere per completare la coda, scopriamo che il totale è di 640 tracce.

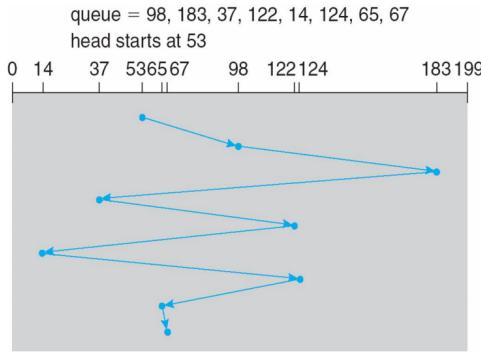


Figura 89: Funzionamento dell'algoritmo FCFS.

8.3.2 SSTF

Un possibile miglioramento è il cosiddetto SSTF, che sta per *Shortest Seek-Time First*: è un algoritmo simile al SJF del CPU scheduling (3.2.2), dove si va a scegliere il processo che ha un burst time più basso. Analogamente in questo caso, si va a scegliere il processo meno distante rispetto alla posizione della testina, di conseguenza si va a soddisfare la richiesta che ci permette di saltare meno tracce. Osservando la figura 90, notiamo che l'ordine di esecuzione è diverso dal precedente in quanto dalla traccia 53 si passa alla 65, poi alla 67 e così via. In questo caso la distanza totale percorsa (in tracce) è di 236, che è un terzo rispetto a quella del FCFS. Proprio come

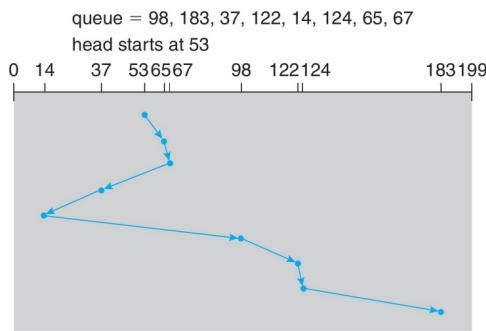


Figura 90: Funzionamento dell'algoritmo SSTF.

nel caso dell'algoritmo SJF ci possono essere dei problemi di **starvation**, ovvero che arrivino sempre richieste vicine tra loro che vengono immediatamente soddisfatte trascurando una richiesta che è più lontana che potenzialmente non verrebbe mai eseguita. Ciò nonostante l'algoritmo rimane migliore rispetto al FCFS però, dato che si possono verificare questo tipo di situazioni, rimane un algoritmo non ottimale.

8.3.3 SCAN e C-SCAN

Al fine di evitare la starvation si è implementato l'algoritmo SCAN. Questo algoritmo è molto semplice: la testina va avanti e indietro lungo il disco e, mentre lo fa, soddisfa

tutte le richieste che arrivano (figura 91). Il problema di questo algoritmo è che si ha uno soddisfacimento non equo delle richieste. Poniamo di trovarci nella traccia 10 e stiamo per arrivare alla 0 per poi tornare indietro. Se arriva una richiesta alla traccia 160 e dopo arrivano delle richieste alla traccia 31, 41, e 59, queste richieste verranno soddisfatte prima della richiesta alla traccia 160 la quale deve attendere più tempo anche se è arrivata prima delle altre.

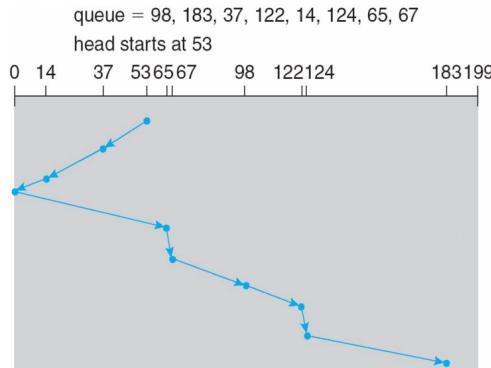


Figura 91: Funzionamento dell'algoritmo SCAN.

Per questo motivo si è implementato un altro algoritmo, molto simile, chiamato **Circular-SCAN** (C-SCAN). Questo algoritmo, al posto di andare avanti e indietro come l'algoritmo SCAN, una volta che arriva alla fine, alla traccia 199, non torna indietro soddisfacendo altre richieste ma va subito alla traccia 0 e ricomincia (figura 92). In questo caso però, anche se le richieste sono soddisfatte in maniere più equa, il tempo di ricerca totale è nettamente maggiore rispetto a quello dell'algoritmo SCAN.

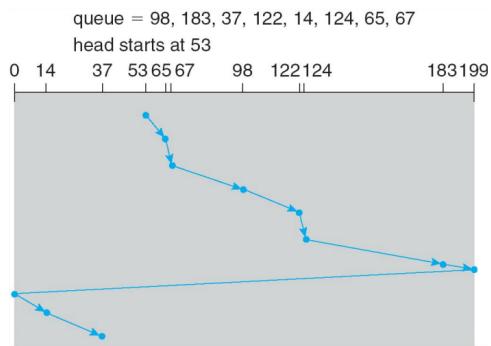


Figura 92: Funzionamento dell'algoritmo C-SCAN.

8.3.4 Scelta dell'algoritmo

La scelta dell'algoritmo da utilizzare dipende sicuramente dal tipo di applicazione che si usano. Quelli discussi in questo capitolo sono gli algoritmi utilizzati più comunemente nei sistemi operativi. Generalmente SSTF è molto più utilizzato rispetto

al FCFS mentre gli algoritmi SCAN e C-SCAN sono utilizzati nei sistemi con un alto carico di dati da memorizzare.