

Sistemi Operativi

Appunti di Corso



Università degli Studi di Torino

Dipartimento di Informatica

Oskar Heise

A.A. 2022/23

Introduzione al corso

Obiettivi formativi

Il sistema operativo costituisce l'interfaccia fondamentale tra l'utilizzatore di un computer e il computer stesso. Parte essenziale del curriculum di base di un laureato in informatica è la conoscenza di come il sistema operativo sia in grado di amministrare le varie componenti hardware di cui è composto un computer. Queste modalità di amministrazione devono essere il più possibile trasparenti al generico utilizzatore del computer, ma devono essere conosciute a fondo da ogni specialista del settore. L'insegnamento fornisce dunque una conoscenza di base dell'architettura interna e del funzionamento dei moderni sistemi operativi, e di come, ai fini di garantire un ragionevole compromesso tra efficienza, sicurezza e facilità d'uso, vengono amministrate le risorse fondamentali della macchina su cui il sistema operativo è installato: il processore, la memoria principale e la memoria secondaria.

Per la parte di laboratorio gli obiettivi formativi sono l'apprendimento del linguaggio C, utilizzato per la programmazione nell'ambiente del sistema operativo Unix. La parte di laboratorio mira a fornire allo studente una conoscenza (teorica e pratica) di base sui comandi della shell, sulla gestione dei processi, sugli strumenti di inter-process communication e sulla gestione dei segnali forniti dal sistema, oltre che alcuni rudimenti di programmazione bash.

Risultati dell'apprendimento attesi

Lo studente acquisirà la conoscenza dell'architettura e del funzionamento dei moderni sistemi operativi, e dovrà essere in grado di ragionare sulle prestazioni fornite dal sistema e su eventuali inefficienze. Avrà inoltre appreso i fondamenti della programmazione C e concorrente e la capacità di sviluppare programmi concorrenti in grado di interagire fra loro senza causare anomalie o blocchi del sistema. Infine, avrà una conoscenza di base di come le risorse della macchina (in particolar modo il tempo di CPU, lo spazio di memoria primaria e lo spazio di memoria secondaria) possano essere sfruttate al meglio.

Lo studente dovrà essere in grado di sviluppare programmi scritti nel linguaggio C, e programmi di sistema e concorrenti codificati con i comandi propri dell'ambiente del sistema operativo Unix; dovrà inoltre essere in grado di ragionare su alcuni problemi di comunicazione fra processi e sincronizzazione utilizzando gli strumenti introdotti durante il laboratorio.

Modalità di insegnamento

L'insegnamento è diviso in una parte di teoria e una di laboratorio.

Per la parte di teoria sono previste 48 ore di lezione frontali che seguono il programma riportato più avanti, integrate da casi di studio e da esercitazioni volte ad illustrare l'applicazione pratica dei concetti appena studiati.

La parte di laboratorio è articolata in due moduli da 30 ore ciascuno, il primo avente per oggetto l'insegnamento del linguaggio C, e il secondo rivolto alla programmazione in ambiente UNIX. Le lezioni si svolgono in maniera interattiva e sono corredate da vari esercizi miranti a fornire esempi pratici.

Indice

1 Generalità	5
1.1 Introduzione	6
1.1.1 Organizzazione del Sistema Operativo	6
1.2 Struttura della Memoria	7
1.2.1 Gestione dell'Input-Output	8
1.2.2 Multitasking e Time-Sharing	8
1.3 I Compiti del Sistema Operativo	9
1.3.1 Duplice Modalità di Funzionamento	9
1.3.2 Protezione della Memoria	9
2 Strutture dei Sistemi Operativi	10
2.1 Servizi di Sistema	11
2.1.1 Gestione dei Processi	11
2.1.2 Gestione dei File e File System	11
2.1.3 Macchine Virtuali	12
3 Gestione e Sincronizzazione dei Processi	13
3.1 Concetto di Processo	14
3.2 Diagramma di Accodamento	15
3.2.1 Process Control Block (PCB)	15
3.2.2 CPU Scheduler	15
3.3 Operazioni sui Processi	16
3.3.1 Creazione di un Processo	16
3.3.2 Comunicazione tra Processi	17
3.4 Scheduling della CPU	18
3.4.1 Algoritmi di Scheduling	19
3.5 Scheduling per Sistemi Multi-Core	21
3.5.1 Scheduling in Solaris	22
3.5.2 Scheduling in Windows	22
3.5.3 Scheduling in Linux	22
3.6 Sincronizzazione dei Processi	23
3.6.1 Le Sezioni Critiche	23
3.7 Il Problema delle Sezioni Critiche	24
3.7.1 Sincronizzazione via Hardware	24
3.7.2 Semafori	25
3.8 Esempi di Sincronizzazione	26

INDICE

3.8.1	Produttori-Consumatori con Memoria Limitata	26
3.8.2	Problema dei Lettori-Scrittori	27
3.8.3	Problema dei Cinque Filosofi	28
3.9	Deadlock	28
3.9.1	Caratterizzazione del Deadlock	29
3.10	I Thread	30
3.10.1	Definizione di Thread	30
3.10.2	Architetture Multi-Core	31
3.10.3	Simultaneous Multi-Threading (SMT)	32
4	La Memoria Fisica	33
4.1	Introduzione	34
4.2	Binding	35
4.2.1	Binding degli Indirizzi in Fase di Compilazione	36
4.2.2	Binding degli Indirizzi in Fase di Caricamento in RAM	36
4.2.3	Binding degli Indirizzi in Fase di Esecuzione	36
4.2.4	Spazio degli Indirizzi Logici e Fisici	37
4.3	Le Librerie	37
4.4	Tecniche di Gestione della Memoria Principale	38
4.5	Paginazione della Memoria	40
4.5.1	Traduzione degli Indirizzi	41
4.5.2	I Vantaggi della Paginazione	42
4.5.3	Gli Svantaggi della Paginazione	42
4.5.4	Translation Look-Aside Buffer (TLB)	43
5	La Memoria Virtuale	45
5.1	Introduzione	46
5.2	Tecniche di Gestione della Memoria Virtuale	46
5.2.1	Area di Swap	47
5.2.2	Algoritmi di Sostituzione	48
5.2.3	Allocazione Globale e Locale	51
5.3	Trashing	51
5.3.1	Cause del Trashing	52
5.3.2	Antidoti e Prevenzione al Trashing	52
6	La Memoria di Massa	53
6.1	Struttura del Disco Rigido	54
6.1.1	Scheduling dei Dischi Rigidi	54
6.1.2	Formattazione del Disco	55
6.1.3	Gestione dell'Area di Swap	55
6.2	Sistemi RAID	56
6.3	Memorie a Stato Solido	59
7	Il File System	60
7.1	I File	61
7.1.1	Attributi del File	61
7.1.2	Operazioni sui File	61

INDICE

7.2	Le Directory	62
7.2.1	Protezione	63
7.3	Realizzazione del File System	63
7.3.1	Gestione dello Spazio Libero	65
7.4	Link Unix	65

Capitolo 1

Generalità

In questo capitolo verranno descritte le nozioni base di un Sistema Operativo, dalla sua funzione alla sua organizzazione di base.

1.1 Introduzione

Intuitivamente, possiamo definire un Sistema Operativo un intermediario fra le applicazioni degli utenti e l'hardware. Esso è costituito da programmi che gestiscono le risorse fisiche che compongono il computer, in particolare la memoria primaria, la memoria secondaria, le periferiche e il processore.

Più nel dettaglio il sistema operativo:

- Fornisce gli strumenti per l'uso delle risorse del sistema.
- Alloca le risorse in maniera conveniente.
- Controlla l'esecuzione dei programmi.

Quindi, il SO ha due obiettivi principali:

- Rendere il sistema semplice da usare per l'utente.
- Rendere il sistema sicuro.

In ogni caso, in ogni SO è possibile individuare un "cuore", detto Kernel, che svolge funzioni simili a qualsiasi sistema operativo. Si tratta del sistema operativo vero e proprio, ed è esattamente su questa parte che il corso si concentrerà. Tutti i sistemi operativi hanno Kernel pressocchè simili tra loro.

Inizialmente il termine CPU indicava una singola unità di esecuzione delle istruzioni macchina. In realtà però, un processore di fascia alta può eseguire in parallelo anche 3 o 4 istruzioni macchina, ma sempre appartenenti allo stesso programma. Se c'era più bisogno di maggior potere computazionale, si mettevano assieme più processori che utilizzavano la stessa memoria principale, la RAM. Teoricamente, ancora oggi questi sistemi prendono il nome di sistemi multi-processore, e in particolare la catena più utilizzata di questi sistemi sono detti di tipo UMA (Uniform Memory Access).

Dopo una crescita esponenziale, a partire dai primi anni 2000 però, l'aumento delle prestazioni di un singolo processore rallenta sensibilmente. Gli esperti decidono quindi di mettere assieme più processori, mettendo sopra una "fettina" di silicio due processori, che poi potranno essere sfruttati in parallelo. I due processori implementati assieme cambiano nome: ciascuno viene chiamato core e si comincia a parlare di processore dual core. Questo processore di fatto è in grado di eseguire in parallelo più programmi, uno per ogni processore. Ogni core poi ha due livelli di cache privata (L1 e L2) e un livello di cache condiviso tra tutti i core (L3).

Importante sottolineare che aumentare il numero di core non aumenta le prestazioni della macchina in maniera lineare. Le prestazioni infatti seguono la formula di Amdahl.

1.1.1 Organizzazione del Sistema Operativo

Nella maggior parte del tempo il sistema non fa assolutamente niente. Di tanto in tanto però, le applicazioni che vengono eseguite hanno bisogno del SO e lo "svegliano", in altri casi ancora il SO si "sveglia" sa solo per controllare che non ci siamo problemi. Può anche succedere che il programma che sta girando si stia "comportando male", anche in quel caso interviene il SO. Si dice quindi che il SO è Event Driven, quindi si comporta in base a quello che succede.

Distinguiamo due tipologie di eventi:

- Interrupt: si tratta di eventi di natura hardware, che si manifestano come segnali elettrici inviati da qualsiasi elemento del PC.
- Eccezioni: si tratta di eventi di natura software, cioè causati da uno specifico programma. Queste a loro volta sono divise in:
 - Trap: dovute a malfunzionamenti del programma in esecuzione (es divisioni per zero).
 - System Call: richieste di uno specifico servizio fornito dal SO (es richiesta di eseguire un'operazione su un file).

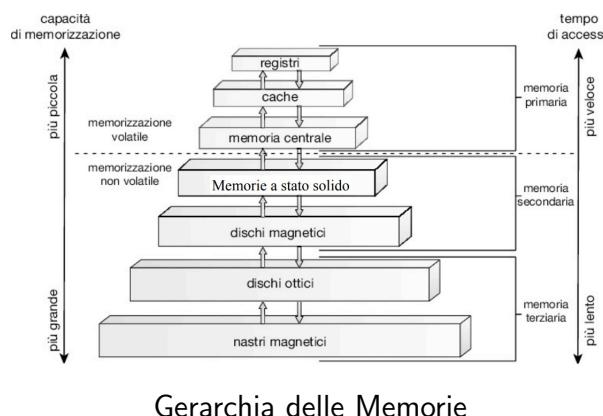
Quando si verifica uno di questi eventi, la CPU reagisce seguendo i seguenti passi:

1. Il valore del PC viene salvato negli appositi registri: in questo modo il programma viene sospeso e potrà essere riavviato quando la gestione dell'evento sarà terminata.
2. Nel PC viene scritto l'indirizzo in RAM della porzione di codice del SO che serve a risolvere l'evento che si è appena verificato. Essenzialmente quando viene acceso il PC, il SO stesso carica in aree della RAM varie porzioni di codice che dovranno entrare in funzione quando si verifica un certo evento. Viene quindi caricato un array noto come vettore delle interruzioni in cui ogni parte del vettore contiene l'indirizzo di partenza di una delle porzioni di codice del SO.

1.2 Struttura della Memoria

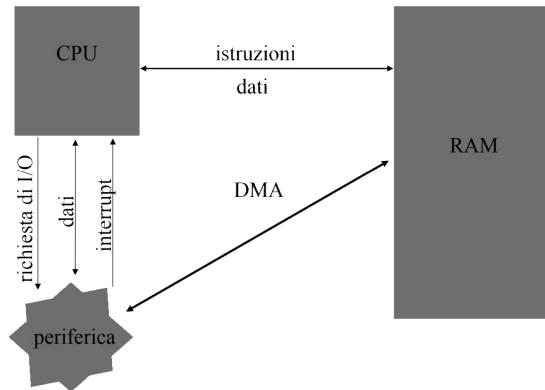
Nel corso ci concentreremo principalmente sulle seguenti due tipologie di memoria: la memoria principale (RAM) che è una memoria temporanea e quella secondaria (di massa) che invece è permanente. Il computer nel suo complesso è poi da immaginare come una gerarchia, e man mano che saliamo dalla piramide le memorie diventano sempre più veloci ma anche decisamente più piccole e più costose.

Da tenere a mente è il concetto di caching, perché ogni componente della gerarchia fa da cache al livello inferiore. Quindi la ram fa da cache all'hard disk, la cache fa da cache per la ram e i registri fanno da cache alla cache.



1.2.1 Gestione dell'Input-Output

Un altro aspetto fondamentale che ha a che fare con la memoria del PC e gli altri dispositivi con cui il processore deve dialogare è la presenza di dispositivi di Input-Output. Ognuno di questi dispositivi è controllato da un componente detto controller, ovvero una sorta di piccolo processore con alcuni registri e una memoria interna, detta buffer. Il SO interagisce con il controller attraverso un apposito software noto come driver. Tutto il procedimento di scambio di informazioni viene effettuato tramite il meccanismo di interrupt, che però è molto inefficiente qualora la quantità di dati è massiccia. È il caso delle comunicazioni tra la memoria secondaria e quella primaria. In questi casi si utilizza un canale apposito chiamato DMA: il SO, istruisce opportunamente il controller con un comando (con scritto cosa deve fare), a quel punto quest'ultimo trasferisce direttamente il blocco in RAM usando il DMA e una volta che si conclude l'operazione avverte il SO mediante un interrupt unico.



Struttura di IO

1.2.2 Multitasking e Time-Sharing

Si tratta di due concetti fondamentali per il corso. Quando lanciamo un programma il SO cerca il codice del programma sull'HDD e "lo fa partire", ma noi utenti non abbiamo nessun bisogno di conoscere la posizione di questo codice né di dove verrà caricato. Dunque, il SO serve a semplificarcici la vita e rendere la macchina efficiente. Può capitare che il programma debba fermarsi temporaneamente (per esempio a causa di un I/O), fino a che l'operazione non si completa quindi il programma non può usare la CPU. È qui che entra in gioco il principio del Multi-Tasking: quando un programma si ferma temporaneamente, il SO ha già in RAM un altro programma a cui assegnare la CPU. Di conseguenza la produttività del PC aumenta drasticamente. Chiarimento: una CPU con un solo core non potrà mai gestire contemporaneamente due programmi, ma è proprio grazie a questi due meccanismi e al rapido cambio di lavoro da un processo all'altro, in frazioni di tempo bassissime, che il processore riesce a gestire più programmi (quasi) all'unisono.

1.3 I Compiti del Sistema Operativo

Un moderno sistema si occupa anche di lanciare nuovi programmi e gestire la terminazione dei vecchi. Ma deve anche sapere qual è il prossimo programma da lanciare, e quindi gestire il cosiddetto CPU Scheduling. Poi vi è il problema della sincronizzazione tra due programmi in comunicazione tra loro e anche della gestione della RAM e della memoria secondaria. Infine, un PC deve anche essere in grado di gestire i file e le cartelle, che sono organizzate in una struttura gerarchica denominata File System.

1.3.1 Duplice Modalità di Funzionamento

Nei processori moderni, le istruzioni macchina possono essere eseguite in due modalità diverse: normale (modalità utente) e di sistema (modalità privilegiata). Il processore è dotato di un bit che determina appunto se l'istruzione in esecuzione è per conto di un programma utente o per conto di un sistema operativo. Questa distinzione viene fatta per evitare che l'utente usi istruzioni delicate che potrebbero danneggiare il computer se usate incautamente.

I programmi utenti tuttavia hanno la possibilità di usare le già citate System Call con cui chiedono al SO di compiere operazioni che l'utente normale non potrebbe eseguire da solo. Ovviamente quando la chiamata al sistema entra in funzione, viene generata un'eccezione e il bit di modalità viene settato automaticamente. Una volta che il SO finisce il da farsi, si restituisce il controllo della CPU al programma utente. Si dice di solito che il processo utente sta eseguendo in Kernel Mode.

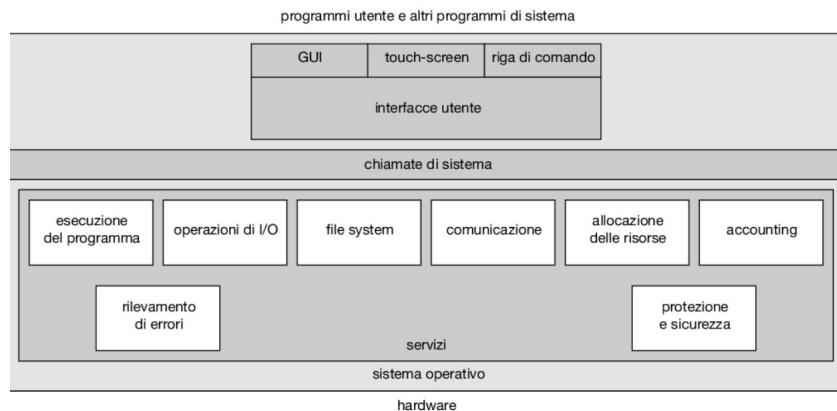
1.3.2 Protezione della Memoria

C'è un'altra importantissima funzione del Sistema Operativo, e sto parlando dell'assicurarsi sempre il controllo della macchina per evitare che un programma vada a leggere e/o modificare i dati di altri programmi, incluso il SO stesso. Una possibile soluzione per proteggere il PC dalla modifica inopportuna di programmi, è il seguente: in due registri appositi della CPU (chiamati Base e Limite) il SO carica gli indirizzi di inizio e fine dell'area di RAM assegnata ad un programma. Ogni indirizzo *I* generato, a questo punto, viene controllato e nel caso in cui non cada nell'intervallo specificato si verifica una Trap.

Capitolo 2

Strutture dei Sistemi Operativi

Un sistema operativo mette a disposizione dei propri utenti (e dei loro programmi) molti servizi. Alcuni di questi sono completamente invisibili all'utente, altri invece sono parzialmente visibili e altri ancora vengono usati direttamente dall'utente. Il grado di visibilità però dipende anche dal tipo di utente. In questo capitolo analizzerò le varie interfacce e il loro "rapporto" con l'utente.



Struttura dei Sistemi Operativi

2.1 Servizi di Sistema

I moderni SO offrono interfacce grafiche (GUI) in modo da semplificare la vita agli utenti. I primi a offrirle furono Apple e Microsoft, poi successivamente anche Unix. Ma i servizi di sistema non si limitano all'estetica, esistono diversi servizi come compilatori, posta elettronica, browser, editor di vario tipo.

Ben più importanti all'interno di un Sistema Operativo sono le System Call e da ora in poi chiameremo processi i programmi in esecuzione. Come abbiamo già osservato, quando un processo deve compiere qualche operazione delicata, deve chiedere aiuto al SO, effettuando una chiamata di sistema. Le System Call quindi sono la vera interfaccia tra gli utenti e il sistema operativo.

Poi abbiamo le API che sono un'interfaccia per le applicazioni che utilizzano codice (di solito una libreria) che non appartiene al processo che sto girando. Di solito, un'API è composta da un insieme di chiamate a funzioni/metodi che possono essere effettuate alla libreria dall'applicazione.

2.1.1 Gestione dei Processi

In un dato istante, all'interno del SO sono attivi più processi, anche se uno solo è in esecuzione. Si parla quindi di Processi Concorrenti, perchè questi competono per usare le risorse hardware della macchina:

- La CPU.
- Lo spazio di memoria primaria e secondaria.
- I dispositivi di input e output.

Il SO ha la responsabilità di fare in modo che ogni processo abbia le sue risorse, senza danneggiare gli altri processi. Inoltre deve gestire tutta la vita dei processi, e quindi:

- Creazione e cancellazione dei processi.
- Sospensione e riavvio dei processi
- Sincronizzazione tra i processi.
- Creazione di meccanismi per la comunicazione tra i processi.

Per eseguire un programma questo deve essere caricato in memoria primaria, ma anche i processi vanno caricati in quella stessa area di RAM, il SO quindi deve tenere traccia di dove ha messo ciascun processo e di come distribuire la RAM in base all'evoluzione di un processo.

2.1.2 Gestione dei File e File System

Quasi ogni informazione è contenuta in un File, ovvero una raccolta di informazioni denotata da un nome. Questi sono a loro volta organizzati in una struttura gerarchica detta File System, mediante le cartelle. Il SO quindi è responsabile anche di creare e cancellare i file e le directory, ma anche di fornire strumenti per la gestione di questi ultimi,

2.1.3 Macchine Virtuali

UN moderno SO è anche in grado di trasformare una macchina reale in una sorta di Macchina Virtuale (MV). Questo permette all'utente di usare la MV indipendentemente dall'hardware, può avere a disposizione una CPU, un File System e altre risorse private, e in più questo permette di avere più portabilità di applicazioni tra macchine diverse.

Capitolo 3

Gestione e Sincronizzazione dei Processi

Il processo è l'unità di lavoro del Sistema Operativo, perchè quello che fa principalmente il SO è amministrare la vita dei vari processi che girano sul PC. Possiamo all'incirca dire che un processo equivale ad un programma in esecuzione, ma in realtà è qualcosa di più di un semplice programma, infatti ha una struttura in memoria primaria suddivisa in più parti, che messi insieme prendono il nome di Immagine del processo:

- Codice da eseguire.
- Dati.
- Stack.
- Heap.

Inoltre è corretto dire che attraverso un programma si possono definire più processi, tuttavia la distinzione fondamentale è che un programma è un'entità statica, mentre un processo è un'entità dinamica. Lo stesso programma lanciato due volte può quindi dare origine a due processi diversi.

3.1 Concetto di Processo

Un programma si trasforma in un processo quando viene lanciato. Da quando nasce a quando termina, un processo passa la sua esistenza muovendosi tra un insieme di stati, e in ogni istante della sua vita un processo si trova in un ben determinato stato. Questa cosa viene illustrata dal diagramma di transizione degli stati di un processo:

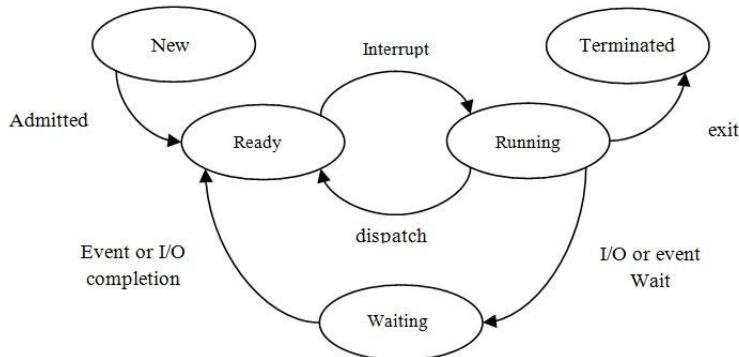


Diagramma di Transizione degli Stati di un Processo

I vari stati in cui un processo può trovarsi sono:

- **New**: il processo è appena nato.
- **Ready**: il processo è pronto per entrare in esecuzione.
- **Running**: la CPU sta eseguendo il codice del processo.
- **Waiting**: il processo ha eseguito una chiamata di sistema ed è fermo in attesa del risultato.
- **Terminated**: il processo è terminato.

Ecco cosa succede generalmente ad un processo:

1. In primo luogo, il processo viene "creato" caricandolo da un dispositivo di archiviazione secondario (disco rigido, CD-ROM, ecc.) nella memoria principale. Successivamente, il process scheduler gli assegna lo stato di "Ready".
2. Mentre il processo è in "Ready", attende che lo scheduler effettui il cosiddetto cambio di contesto. Il cambio di contesto carica il processo nel processore e ne cambia lo stato in "Running", mentre il processo precedentemente "Running" viene conservato nello stato "Ready".
3. Se un processo nello stato "Running" deve attendere una risorsa (ad esempio, attendere l'input dell'utente o l'apertura di un file), gli viene assegnato lo stato "Waiting". Lo stato del processo viene riportato a "Ready" quando il processo non ha più bisogno di aspettare (in uno stato bloccato).
4. Quando il processo termina l'esecuzione o viene terminato dal sistema operativo, non è più necessario. Il processo viene rimosso istantaneamente o viene spostato nello stato "Terminated". Quando viene rimosso, attende solo di essere rimosso dalla memoria principale.

Il context switch richiede tempo perché è composto da molte informazioni, generalmente quindi il cambio di contesto costa qualche centinaio di nanosecondi. Il tempo sprecato è detto Overhead e influisce le prestazioni del sistema.

3.2 Diagramma di Accodamento

Per amministrare la vita dei processi il SO gestisce le code dei processi. Questa non è altro che una lista di PCD mantenuta in una delle aree di memoria che il Sistema riserva per sé stesso. La coda più importante è la Ready Queue(RQ), ovvero l'insieme dei processi nello stato Ready. Dunque la RQ coincide con lo stato Ready del diagramma.

Quando un processo lascia la CPU e nè termina nè va in ready, può fare due cose:

- Device Queue: sono le code dei processi in attesa di un dispositivo I/O.
- Waiting Queue: sono le code dei processi in attesa di un certo evento.

Durante la loro vita quindi i processi vanno di coda in coda.

Possiamo quindi riformulare il diagramma visto in precedenza creando il Diagramma di Accodamento in cui i processi si muovono tra le varie code

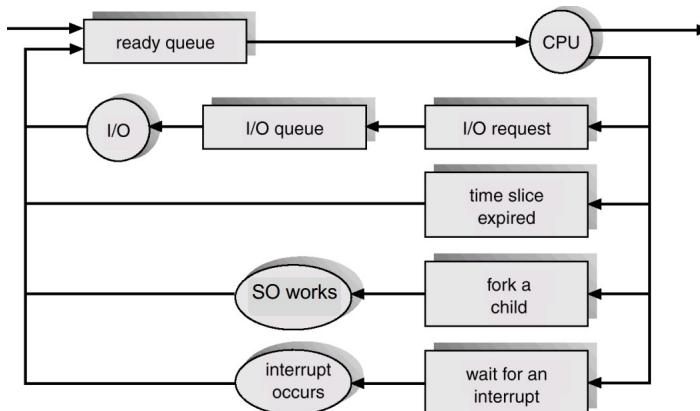


Diagramma di Accodamento

3.2.1 Process Control Block (PCB)

Per amministrare bene i processi, il sistema operativo mantiene una struttura dati chiamata Process Control Block che contiene tutte le informazioni necessarie ad amministrare al meglio i processi. Tra queste informazioni ci sono: ID del processo (PID), stato del processo, indirizzi RAM in cui è stato salvato ecc.

3.2.2 CPU Scheduler

Un componente fondamentale del SO è il CPU Scheduler, detto anche Short Term Scheduler, ovvero una porzione del Sistema Operativo che va a cercare nella coda di Ready a cercare il prossimo processo da mandare in esecuzione. Per evitare l'overhead, deve agire molto velocemente.

3.3 Operazioni sui Processi

Ogni Sistema Operativo possiede una specifica System Call di creazione di un processo e ogni processo è sempre creato a partire da un altro processo usando la System Call relativa. Il processo creatore si chiama Processo Padre, il processo creato è invece detto Processo Figlio.

3.3.1 Creazione di un Processo

Quando nasce un nuovo processo, il SO gli assegna un identificativo unico, ovvero il suo Process-ID (PID), poi recupera dall'hard-disk il codice da eseguire e lo porta in RAM, alloca un nuovo PCB e lo inizializza con le informazioni necessarie e infine inserisce il PCB nella Ready Queue. Ma analizziamo i vari punti attentamente:

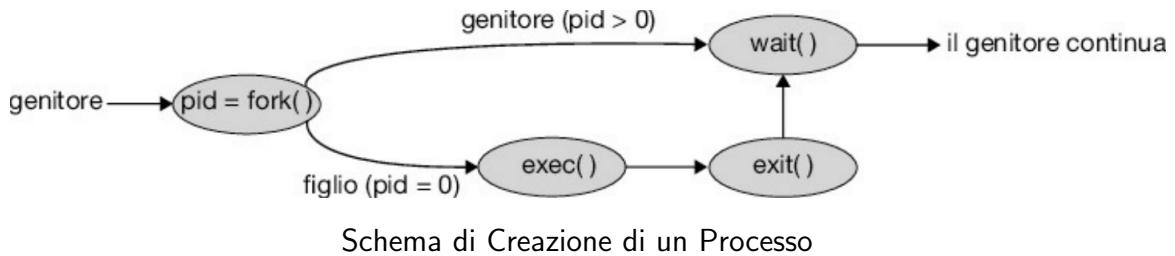
```

1 int main() { /* fig. 3.8 modificata */
2     pid_t pid, childpid;
3     pid = fork(); /* genera un nuovo processo */
4     printf("questa la stampano padre e figlio");
5     if (pid == 0) { /* processo figlio */
6         printf("processo figlio");
7         execlp("/bin/ls", "ls", NULL);
8     } else {/* processo padre */
9         printf("sono il padre, aspetto il figlio");
10        childpid = wait(NULL);
11        printf("il processo figlio      terminato");
12        exit(0);
13    }
14 }
```

1. Quando il Processo Padre invoca la System Call e genera il Processo Figlio, prosegue la sue esecuzione oppure si mette in attesa del completamento dell'esecuzione di quest'ultimo.
2. Una volta che nasce, al processo figlio viene data una copia del codice e dei dati in uso al Processo Padre, oppure al Processo Figlio viene dato un nuovo programma con nuovi dati.
3. Il processo entra nel Sistema Operativo, gli viene allocato un nuovo PCB e al suo interno vengono messe le strutture dati e il codice del Processo Padre.
4. Il SO inizializza il Program Counter del figlio con l'indirizzo successivo alla `fork()`. In una cella di memoria associata alla variabile che riceve il risultato della `fork` viene scritto 0 nel Figlio, mentre nel Padre viene scritto il PID del Figlio. In questo modo sappiamo sempre qual è il processo Parent.
5. Infine il Processo Figlio viene messo in coda di Ready.

Le System Call più importanti sono quindi:

- `fork()`: crea un processo.
- `execp()`: riceve in input un puntatore ad un file contenente codice eseguibile. Il processo che la invoca prosegue eseguendo il codice specificato.
- `wait()`: serve al Padre per farlo attendere la fine della creazione del Figlio. Restituisce al Padre il PID del Figlio appena creato.
- `exit()`: serve per dire al Sistema Operativo di terminare istantaneamente il processo di creazione del figlio.
- `kill()`: serve per uccidere esplicitamente un processo.

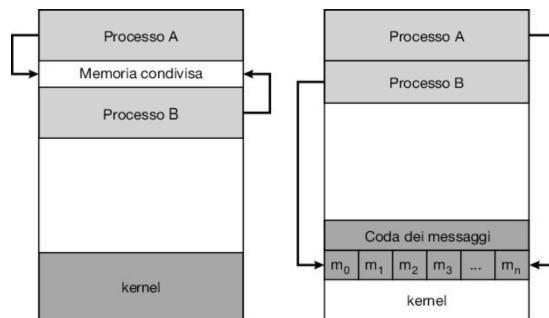


3.3.2 Comunicazione tra Processi

I processi di un sistema concorrono per l'uso delle risorse del computer. Due o più processi sono fra loro:

- Indipendenti: se non influenzano l'un l'altro durante il processo.
- Cooperanti: se si influenzano l'un l'altro per scambiarsi informazioni e portare avanti un'elaborazione divisa in vari processi per ragioni di efficienza.

Ovviamente, se i processi cooperano, il Sistema Operativo deve mettere a disposizione meccanismi di cooperazione e sincronizzazione. Per scambiarsi le informazioni, i processi utilizzano dei meccanismi di Inter-Process Communication (IPC), ovvero opportune System Call in grado di permettere lo scambio dei messaggi o di usare la stessa area di memoria condivisa.



IPC

3.4 Scheduling della CPU

Nella vita di un processo si alternano fasi di uso della CPU (burst di CPU) e fasi di attesa per il completamento di operazioni di I/O (burst di I/O). Chiamo quindi CPU-bound i processi che usano molto la CPU e poco i dispositivi di I/O, e I/O-bound quelli che fanno esattamente il contrario.

Quando un processo utente abbandona la CPU, il SO si sveglia e decide tra tutti i processi pronti per essere eseguiti in coda di Ready qual è il prossimo da mandare in esecuzione. Questo processo si chiama Scheduling della CPU. Ma facciamo dei distinguo in base ai vari casi:

1. Il processo che sta usando la CPU passa dallo stato Running allo stato di Waiting,
2. Il processo che sta usando la CPU termina.

In questi due casi, lo scheduler deve prendere un processo dalla coda di Ready e metterlo in Running. Ma se capita che un processo non lascia mai lo stato di Running possono esserci varie conseguenze:

- Il processo che sta usando la CPU viene obbligato a passare allo stato Ready, per il principio di time-sharing.
- Il processo entra in Ready perchè è appena nato o perchè arriva dallo stato di Waiting. Il SO interviene per due ragioni:
 3. I processi non si spostano da soli ma è il SO che, conoscendo il PCB del processo, si accorge del completamento di un'operazione di I/O e li sposta.
 4. Se il processo è più importante di quello attualmente in esecuzione, il SO toglie quest'ultimo dalla CPU e manda in esecuzione il primo.

Ebbene, quando un SO interviene solo nei casi 1-2 si parla di Scheduling senza diritto di Prelazione (Non-Preemptive Scheduling), se interviene anche nei casi 3-4 si parla invece di Scheduling con diritto di prelazione (Preemptive Scheudling).

Importantissimi sono poi i criteri di Scheduling, che vengono decisi in base ai seguenti principi:

- Massimizzare l'utilizzo della CPU.
- Massimizzare il Throughput, ossia la produttività del sistema.
- Minimizzare il tempo di risposta, ovvero il tempo che intercorre da quando si avvia un processo a quando viene eseguito.
- Minimizzare il Turnaroud time, ossia il tempo medio di completamento di un processo.
- Minmizzare il Waiting time, ossia la somma del tempo passato dal processo in Ready Queue.

Esiste una relazione tra il Turnaround e il Waiting time, infatti, supponendo che un processo non finisca mai in coda di wait, il Turnadound time non è altro che il Running time sommato al tempo in cui il processo è stato in coda di Ready.

3.4.1 Algoritmi di Scheduling

Esistono diversi algoritmi di scheduling. Considerando solamente processi con un unico burst di CPU e nessun burst di I/O, e una generica "unità di tempo", i principali sono i seguenti:

First Come, First Served (FCFS) Si tratta del più semplice da implementare, essenzialmente tratta la coda di ready in modo First in First out (FIFO). Si tratta di un algoritmo non-Preemptive e ha tempi di attesa molto lunghi.

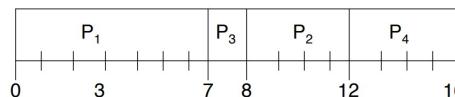
Uso un diagramma di Gantt per rappresentare questa situazione:



First Come, First Served

Shortest Job First (SJF) In questo algoritmo, si esamina la durata del prossimo burst di CPU e si assegna la CPU al processo con il burst di durata minima. Può essere usato in modo Preemptive e non-Preemptive. Si tratta di un algoritmo ottimale, tuttavia, dato che non si conosce la durata del prossimo burst time è impossibile da implementare.

Uso un diagramma di Gantt per rappresentare questa situazione:



Shortest Job First

Priority Scheduling (PS) Si tratta non tanto di un algoritmo singolo ma di un pacchetto di algoritmi. In questo caso associamo una misura di priorità dei processi che può essere di due tipi:

- Interna al sistema: calcolata dal SO sulla base del comportamento di ogni processo.
- Esterna al Sistema: assegnata con criteri esterni dal SO.

A volte però può sorgere il seguente problema: cosa succede se un processo in Ready ha sempre una priorità peggiore rispetto agli altri? In questi casi si dice che il Processo và in Starvation, e per risolvere questo problema interviene il meccanismo di Aging. Il SO aumenta la priorità di un processo anche in base al tempo trascorso in cosa così che prima o poi questo verrà scelto e fatto eseguire.

Round Robin (RR) In questo algoritmo, ogni processo ha a disposizione una data quantità di tempo, chiamata Quanto di tempo. Se entro l'arco di tempo il processo non lascia volontariamente la CPU, viene interrotto dal SO e rimesso in RQ.

Le prestazioni dell'algoritmo dipendono dal Quanto di tempo: se questo è troppo elevato si ha un effetto simile all'FCFS, mentre se è troppo basso aumenta il context switch e l'overhead.

Multi-level Queue Scheduling In questo algoritmo, i processi sono divisi in classi differenti:

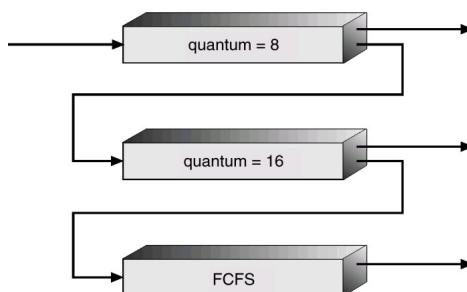
- Foreground: interattivi, ad esempio un editor.
- Background: non interattivi.
- Batch: possono essere differiti, ad esempio di notte perché sono troppo lunghi.

Grazie a questa divisione è possibile inserire i processi in una delle code e gestire appropriatamente in base a ciascuna proprietà. Per esempio si può gestire il Foreground in RR, e gli altri due in FCFS.

Per scegliere fra le code si può avere una proprietà fissa, per esempio gestire prima tutti i processi Foreground e poi gli altri (ma si rischia Starvation) oppure, utilizzando dei tempi per ciascuna coda.

Scheduling a Code Multilivello con Retroazione (MFQS) Si tratta del tipo più generale di algoritmo di Scheduling ed è il più usato nei Sistemi Operativi moderni.

In questo caso, l'assegnamento di un processo a una coda non è fisso, ma questi possono venir spostati dal SO da una coda all'altra per adattarsi alla lunghezza di CPU Burst del processo e gestire ogni coda con lo Scheduling più adatto. Quindi quando un processo nasce viene messo nella coda a probabilità più elevata e riceverà, quando arriva il suo turno, un Quanto di tempo pari a x .

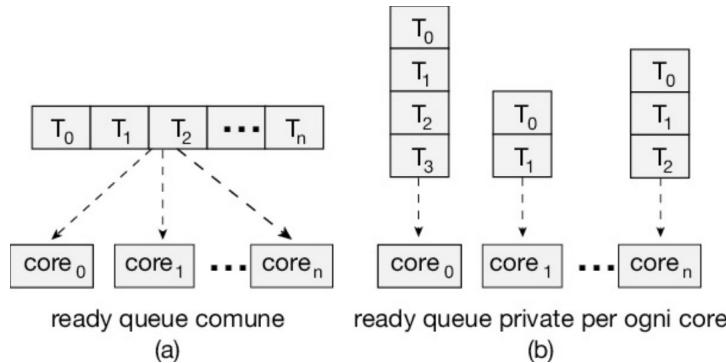


Scheduling a Code Multilivello con Retroazione

Se il processo non usa tutto il tempo che gli è stato dato a disposizione, perché è finito o perché deve ricevere un I/O, quando rientrerà, avrà comunque lo stesso quantitativo di tempo che aveva inizialmente, quindi sempre x . Se invece il processo non termina l'esecuzione ma finisce il tempo a disposizione, al giro successivo verrà messo in una coda con più Quanti di tempo, ma con priorità inferiore.

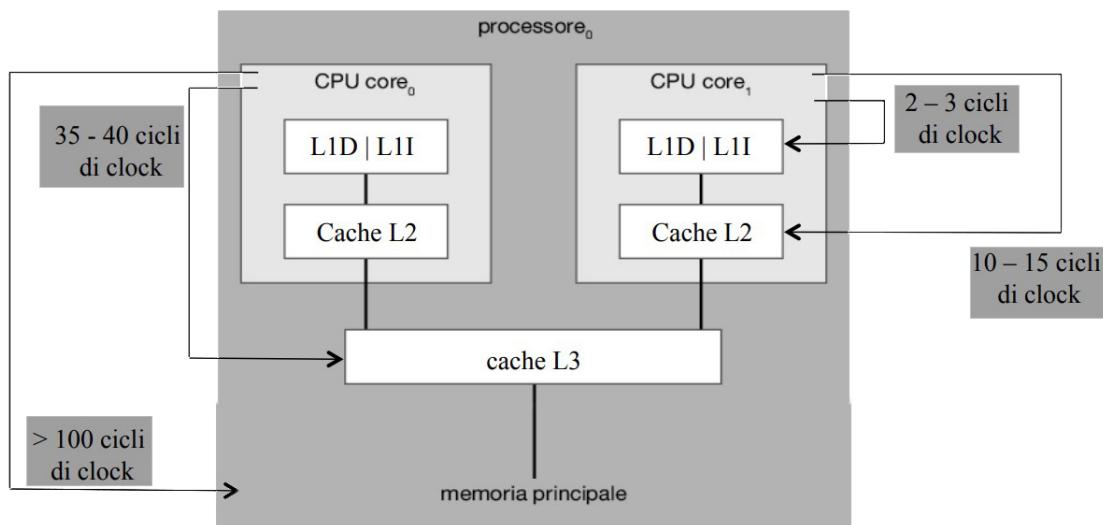
3.5 Scheduling per Sistemi Multi-Core

Come detto in partenza, la maggioranza dei PC moderni hanno molteplici Core, che vedono la stessa memoria principale e condividono un livello di cache. In questo modo, si ottengono prestazioni maggiori, a patto che il SO sappia sfruttare appieno ciascun core. Tutti i SO prevedono in particolare la Multielaborazione Simmetrica (SMP) in cui uno scheduler gira su ciascun core. Quando lo Scheduler si attiva, sceglie uno dei processi presenti nella sua coda di Ready (o in una comune per tutti) e lo manda in esecuzione sul proprio core:



Un aspetto importante è poi il Bilanciamento del Carico, quindi distribuire omogeneamente i processi tra i vari Core. Inoltre si può specificare che un processo non possa spostarsi da un core all'altro, per evitare di perdere dati salvati nella cache privata.

Qui vediamo il costo in Cicli di Clock necessari per accedere a un certo livello di cache:



3.5.1 Scheduling in Solaris

Solaris è un sistema operativo basato su Linux che usa uno Scheduling a priorità con code multiple a retroazione, i cui processi sono divisi in 4 classi distinte:

- Real Time (priorità maggiore).
- Sistema.
- Interattiva.
- Time Sharing (priorità minore).

Quando un processo nasce, si trova nella priorità più bassa, se non diversamente specificato. Dopodichè viene usata una tabella con 60 righe, ognuna che rappresenta una diversa coda di Scheduling e una diversa priorità, da 0 a 59.

La priorità corrente determina il Quanto di tempo che gli verrà assegnato. Notare che priorità e tempo assegnato sono inversamente proporzionali, e notare cosa succede quando vengono esauriti o meno i Quanti: Lo Scheduler calcola la priorità globale di un

Priorità corrente	Quanto di tempo (millisecondi)	Nuova priorità (quanto esaurito)	Nuova priorità (quanto non esaurito)
0	200	0	50
20	120	10	52
30	80	25	53
59	20	49	59

Scheduling Solaris

processo in base alla sua priorità e alla classe a cui appartiene, e ovviamente assegna la CPU al processo con priorità globale più alta.

3.5.2 Scheduling in Windows

Lo Scheduling su Windows è basato su priorità con retroazione e prelazione. Vengono usati 32 livelli di priorità divisi in real-time (da 16 a 31) e gli altri inferiori, tranne lo 0 che è riservato. In genere viene scelto il processo a priorità più alta, se ce ne sono due alla pari si usa RR.

Nel caso in cui il processo va in Waiting la sua priorità viene alzata e l'incremento dipende anche dal tipo di evento atteso: se è un dato dalla tastiera (processo interattivo) si avrà un grosso aumento, per esempio. Al contrario, se la CPU esaurisce il Quanto di tempo, la sua priorità viene abbassata.

3.5.3 Scheduling in Linux

Linux utilizza un algoritmo predefinito chiamato Completely Fair Scheduler (CFS). Questo cerca di distribuire il tempo di CPU a tutti i processi Ready, assumendo che se ci sono n processi Ready, ogni processo deve aspettare esattamente $\frac{1}{n}$ del tempo.

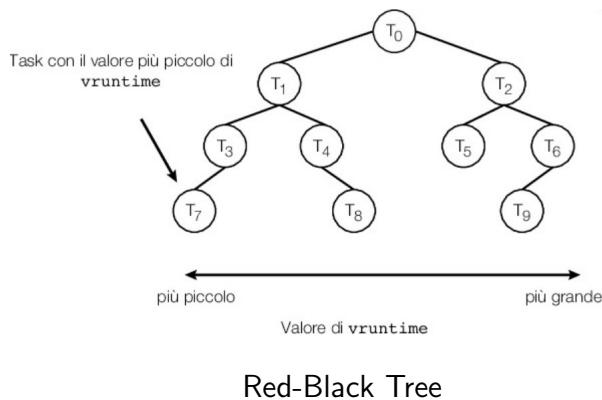
Essenzialmente, ad ogni Context Switch, il CFS ricalcola quanto si deve dare la CPU ad un processo. Siano:

- P.expected_run_time: tempo di CPU che spetta a P.
- P.vruntime: tempo di CPU consumato da P.
- P.due_cputime: tempo di CPU che ancora spetta a P.

Dunque:

$$P.vruntime = P.expected_run_time - P.due_cputime$$

Quindi: la CPU viene data al processo con più basso valore di P.vruntime e questi non vengono più organizzati in code ma diventano nodi di un albero di ricerca binario detto Red-Black Tree (R-B Tree) che permette di cercare, inserire o cancellare uno dei nodi. Queste operazioni, hanno un costo computazionale $O(\log n)$ dove n è il numero dei nodi.



3.6 Sincronizzazione dei Processi

Come già detto in precedenza, più processi devono cooperare tra loro per lavorare al meglio. Tuttavia è importante che l'accesso ai dati che questi processi condividono e modificano non vengano mai lasciati in uno stato inconsistente. I processi infatti devono agire in modo ordinato, cioè sincronizzarsi quando ciascuno di loro vuole accedere ai dati.

3.6.1 Le Sezioni Critiche

Ogni processo ha almeno una porzione di codice, detta Sezione Critica, in cui il processo manipola le variabili condivise. Quando un processo è dentro la propria sezione critica, nessun altro processo può eseguire codice nella sezione critica perché altrimenti userebbe le stesse variabili condivise.

Fino a che P non avrà terminato di eseguire il codice della sua sezione critica, nessun altro processo che deve manipolare con le stesse variabili condivise potrà eseguire codice della propria sezione critica.

3.7 Il Problema delle Sezioni Critiche

Un processo deve "chiedere il permesso" per entrare nella sezione critica con una porzione di codice chiamata Entry Section. Per uscire invece ne usa una detta Exit Section. Quindi un generico processo avrà questa struttura:

```

1 //altro codice
2     //entry section
3     //sezione critica
4     //exit section
5 //altro codice

```

Un'adeguata soluzione al problema della Sezione Critica sono:

1. Mutua esclusione: se un processo P è entrato nella propria sezione critica e non è ancora uscito, nessuno altro processo può entrare.
2. Progresso: se un processo lascia la propria sezione critica, deve permettere a un altro processo di entrare nella propria sezione critica. Se la sezione critica è vuota e più processi vogliono entrare, uno tra questi deve essere scelto in un tempo finito.
3. Attesa limitata: se un processo ha già eseguito la sua Entry Section, deve riuscire ad entrare in quella sezione critica in un certo tempo finito. Quest'ultima condizione garantisce l'assenza di Starvation.

Il progettista del SO deve poi decidere come vanno gestite le sezioni critiche del sistema operativo, e le scelte possibili sono di sviluppare un Kernel che può essere:

- Kernel con diritto di prelazione: un processo in kernel mode può essere interrotto da un altro processo.
- Kernel senza diritto di prelazione: un processo in kernel mode non può essere interrotto da un altro processo.

Il primo tipo di Kernel è molto facile da implementare, poichè basta disattivare gli interrupt quando un processo è in kernel mode. Non bisogna quindi preoccuparsi dell'accesso concorrente alle sezioni critiche del kernel. Sono adatte per le applicazioni real time.

Il secondo tipo di Kernel è più complesso da gestire, poichè deve disabilitare gli interrupt per il tempo necessario al codice del SO di accedere in modo esclusivo ad una qualche struttura dati del SO

3.7.1 Sincronizzazione via Hardware

Esistono soluzioni semplici al problema della Sezione Critica e possono essere ottenute usando specifiche istruzioni macchina:

- *TestAndSet(var1)*: testa e modifica il valore di una cella di memoria
- *Swap(var1, var2)*: scambia il valore di due celle di memoria

3.7.2 Semafori

I semafori sono uno strumento di sincronizzazione che può essere implementato senza busy-waiting. Si tratta di una variabile intera su cui si può operare solamente tramite due operazioni atomiche (operazioni che non devono essere interrotte) che per il momento descrivo come:

```
1 wait (S):           while S <= 0 do no-op;
2                     S = S - 1;
```

```
1 Signal (S):        S = S + 1;
```

Un semaforo può essere visto come un oggetto condiviso da tutti i Processi che devono usarlo per sincronizzarsi tra di loro. La variabile intera S viene denotata come Variabile Semaforica e il suo valore è detto Valore del Semaforo.

Wait e Signal sono i metodi con cui usare il semaforo.

Adesso la soluzione del problema della Sezione Critica per un gruppo di processi è banale. La variabile mutex (mutual exclusion) fa da variabile di lock:

```
1 do{
2     wait(mutex);
3     //sezione critica
4     signal(mutex);
5     //sezione non critica
6 }
```

I semafori possono essere usati per qualsiasi problema di sincronizzazione, non solo in questi casi.

Nel codice qui di sopra abbiamo usato Wait() e Signal() che utilizzano il busy-waiting (quando un Processo che deve attendere una certa condizione lo fa verificando continuamente tale condizione). I semafori implementato attraverso il busy-waiting esistono, e prendono il nome di Spinlock. Per evitare il busy-waiting bisogna farsi aiutare dal SO che mette a disposizione opportune System Call per l'implementazione di Wait e Signal.

Essenzialmente, ogni semaforo è implementato usando due campi: valore e lista di attesa:

```
1 typedef struct {
2     int valore;
3     struct processo *waiting_list;
4 } semaforo;
```

Due System Call sono disponibili per implementare wait e signal:

- Sleep(): toglie la CPU al processo che la invoca e manda in esecuzione uno dei processi in RQ.
- Wakeup(): inserisce il processo nella RQ.

Riprendo il Diagramma di Accodamento e noto il caso Wait for an Interrupt, che avevo ignorato a suo tempo.

Lo posso proprio associare al caso in cui un processo abbandona la CPU "addormentandosi" sul semaforo: Quindi, riassumendo, attraverso i semafori implementati usando

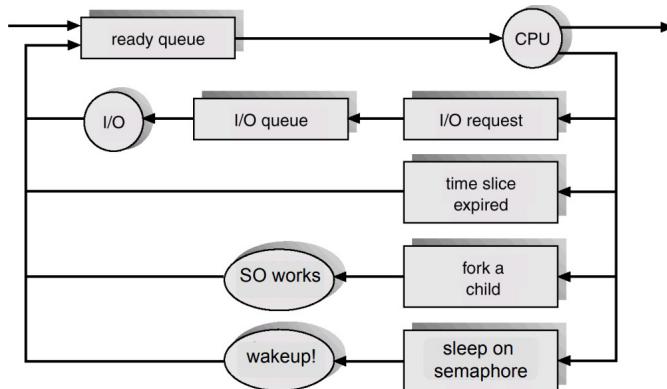


Diagramma di Accodamento

`Sleep()` e `Wait()`, i processi utente possono contenere sezioni critiche arbitrariamente lunghe senza sprecare tempo di CPU e rischiare di dare il controllo della CPU al processo.

`Wait` e `Signal` possono invece essere implementate con busy-waiting perché il tutto avviene per poco tempo e sotto il controllo del SO.

3.8 Esempi di Sincronizzazione

I semafori sono tipologie di sincronizzazione molto primitive, ma in quanto primitive sono anche "rischiose". Il loro problema è che le due operazioni di `Wait` e `Signal` sono indipendenti e quindi possono essere utilizzate in modo errato. Ecco degli esempi di sincronizzazione risolti con i semafori.

3.8.1 Produttori-Consumatori con Memoria Limitata

Uso un buffer circolare di SIZE posizioni in cui i produttori inseriscono i dati e i consumatori li prelevano:

```

1 typedef struct{...} item;
2 item buffer[SIZE]
3 semaphore full, empty, mutex;
4 item nextp, nextc;
5 int in = 0; out = 0;
6 int full = 0; empty = SIZE; mutex = 1;

```

- `full`: conta il numero di posizioni piene del buffer.
- `empty`: conta il numero di posizioni vuote del buffer.
- `mutex`: è un semaforo binario per l'accesso in mutua esclusione del buffer circolare e delle variabili `in` e `out`.
- `in/out`: servono per gestire il buffer circolare.

Il codice di un produttore:

```

1 while(true){
2     ...
3     //produci un item in nextp
4     ...
5     wait(empty);
6     wait(mutex);
7     buffer[in] = nextp;      //inserisce nextp
8     in = (in+1) mod SIZE;   //nel buffer
9     signal(mutex);
10    signal(full);
11 }
```

Il codice di un consumatore:

```

1 while(true){
2     wait(full);
3     wait(mutex);
4     nextc = buffer[out];      //rimuove un
5     out = (out+1) mod SIZE;   //elemento
6     signal(mutex);
7     signal(empty);
8     ...
9     //consuma item in nextc
10    ...
11 }
```

Notare che si usa mutex per gestire in modo mutualmente esclusivo le variabili condivise.

Se ci sono più produttori e il buffer ha almeno due posizioni libere ($\text{empty} \geq 2$), ci possono essere due o più processi che hanno superato la `wait(empty)` e cercano di inserire in buffer un item e di aggiornare la variabile `in`.

Se ci sono più consumatori e il buffer ha almeno due posizioni piene ($\text{full} \geq 2$), ci possono essere due o più processi che hanno superato la `wait(full)` e cercano di prelevare un item dal buffer e di aggiornare la variabile `out`.

3.8.2 Problema dei Lettori-Scrittori

Alcuni processi richiedono solamente la lettura (Lettori), altri invece possono voler modificare il file (Scrittori). Mentre due o più lettori possono accedere al file contemporaneamente, un processo scrittore deve poter accedere al file in mutua esclusione con tutti gli altri processi.

Il codice di uno scrittore:

```

1 wait(scrivi);
2 ...
3 //esegui la scrittura del file
4 ...
5 signal(scrivi);
```

Il codice di un lettore:

```

1 wait(mutex); //mutua esclusione per aggiornare numLettori
2 numLettori++;
3 if numLettori == 1 wait(scrivi); //il primo lettore ferma
//eventuali scrittori
4 signal(mutex);
5 ...leggi il file...
6 wait(mutex);
7 numLettori--;
8 if numLettori == 0 signal(scrivi);
9 signal(mutex);
10

```

3.8.3 Problema dei Cinque Filosofi

Cinque filosofi passano la vita pensando e mangiando, i filosofi condividono un tavolo rotondo con cinque posti e un filosofo per mangiare deve usare due bacchette/risorse. Il dato condiviso è quindi

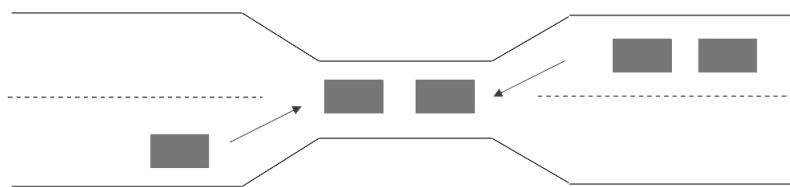
```

1 semaphore bacchetta[5]; //tutti inizializzati a 1

```

3.9 Deadlock

Per Deadlock, o Stallo dei Processi, si intende la situazione in cui ciascun processo in un insieme di n processi ($n \geq 2$) si trova in uno stato di attesa per il verificarsi di un evento che solo uno degli altri processi dell'insieme può provocare. Prendo come esempio questa situazione:



Ponte a una sola corsia

In questa situazione ciascuna posizione di marcia può essere vista come una risorsa. Una situazione di Deadlock può essere risolta se un'auto torna indietro, ovvero se libera una posizione già occupata. Se tuttavia ogni auto attende che l'altra liberi l'unica corsia di marcia si verifica Starvation.

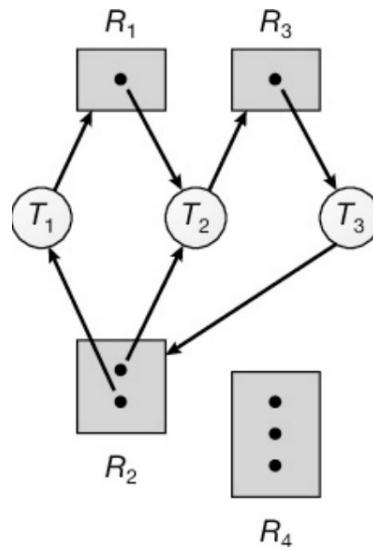
Un sistema Hardware+Software può essere visto come formato da:

- Insieme finito di risorse R (cicli di CU, spazio in memoria, device di I/O).
- Ogni tipo di risorsa è formata da un certo numero di istanze indistinguibili fra loro.
- Insieme di processi P che hanno bisogno di una o più istanze di alcune delle risorse per terminare la computazione.

Posso quindi definire Deadlock di un sottoinsieme di processi del sistema la situazione in cui ciascuno dei n processi P_i è in attesa del rilascio di una risorsa detenuta da uno degli altri processi del sottoinsieme. Si forma quindi una catena circolare per cui: P_1 aspetta P_2 ... aspetta P_n aspetta P_1 . La situazione però non è desiderabile in quanto può bloccare alcune risorse e quindi danneggiare processi non coinvolti nel Deadlock.

3.9.1 Caratterizzazione del Deadlock

Il SO può avvalersi di una rappresentazione detta Grafo di Assegnazione delle Risorse che registra quali risorse sono assegnate a quale processo, e quali risorse sta aspettando ciascun processo. Eccone una rappresentazione in figura:



Grafo di Assegnazione delle Risorse

Infine, è bene tenere a mente tre punti quando si parla di gestione del Deadlock:

1. Prevenire o evitare i Deadlock.
2. Lasciare che il Deadlock si verifichi ma fornire strumenti per la scoperta e il recupero dello stesso.
3. Lasciare agli utenti la prevenzione/gestione dei Deadlock.

3.10 I Thread

Considero due processi che devono lavorare sugli stessi dati, come posso fare se ogni processo ha la propria area dati, e quindi se gli spazi di indirizzamento dei due processi sono separati?

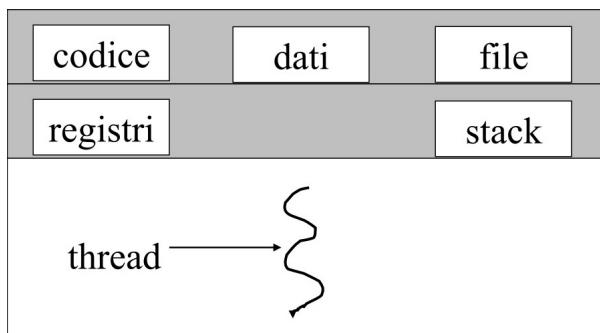
- I due processi possono chiedere al So un'area di memoria condivisa.
- I dati possono essere tenuti in un file che viene acceduto a turno dai due processi.

Tuttavia, sarebbe molto più comodo se due processi potessero lavorare sugli stessi dati senza interpellare il Sistema Operativo e senza usare un file.

Da queste considerazioni nasce il concetto di Thread: un gruppo di Peer Thread è un insieme di processi che condividono lo spazio di indirizzamento (codice e dati).

3.10.1 Definizione di Thread

Rifletto sulla terminologia: un processo P è contraddistinto da un unico Thread (filo) di computazione: questo consiste nella sequenza di istruzioni eseguite che ovviamente può cambiare da un'esecuzione all'altra se per esempio cambiano i dati in input. Un Thread (filo) è una suddivisione di un processo in due o più "fili" o sottoprocessi che vengono eseguiti concorrentemente da un sistema monoprocessore (monothreading), multiprocessore (multithreading) o multicore. Un Thread è composto essenzialmente da



tre elementi: Program Counter, valori nei Registri e Stack. Un processo Multi-Thread (detto anche Task) è composto da più Thread di computazione, detti anche Peer Thread. Ad ogni Peer Thread è assegnata l'esecuzione di codice che è diverso da quello degli altri suoi Peer, e quindi ogni Thread ha uno stato della computazione formato da Program Counter e registri della CPU + Stack. Tuttavia, un insieme di Peer Thread condivide le aree dati.

Ovviamente ci sarà un context-switch tra ogni peer thread, ma questo richiederà molto meno lavoro dal SO poichè richiede solamente il salvataggio del Program Counter e dei Registri, invece non c'è bisogno di cambiare il codice perchè queste informazioni sono condivise. Per questo motivo:

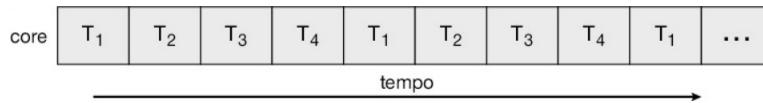
- Heavy-Weight Process (HWP): processi normali.
- Light-Weight Process (LWP): Peer-Thread.

3.10.2 Architetture Multi-Core

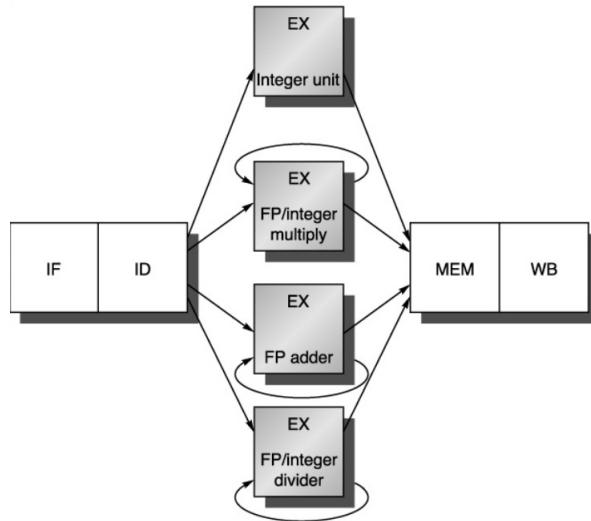
L'uso dei Thread offre diversi vantaggi, tra cui:

- Efficienza: il context-switch è molto meno dispendioso.
- Condivisioni di Dati e Risorse: più thread possono lavorare su dati condivisi in maniera efficiente.
- Architetture multi-core: i thread sono particolarmente adatti per girare su processori multi-core, e ancora più su architetture Multithreaded.

In un processore Single-Core, tutti i Peer-Thread di un Task si alternano in esecuzione esattamente come un insieme di processi, si parla quindi di Multithreading a divisione di tempo:



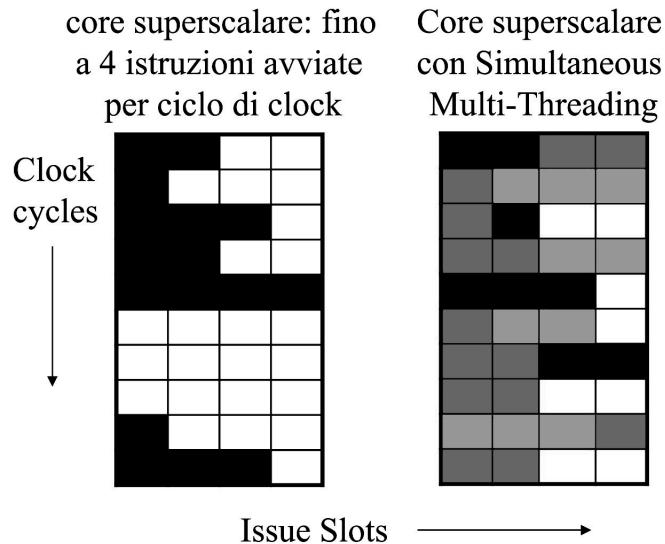
Le architetture Multi-Core invece sono adatte a gestire Task Multi-Threaded, poichè qui i Thread vengono realmente eseguiti contemporaneamente, cioè in parallelo, ciascuno su un distinto core, una tecnica chiamata Multiple Issue. Per poterlo fare ogni core deve essere dotato di più unità funzionali: ALU e Floating Point. Si parla quindi di processi con Architettura Superscalare.



Tuttavia, ogni singolo core, superscalare e multiple issue, non riesce sempre ad avviare in parallelo il massimo numero di istruzioni possibile. Infatti, in un programma le istruzioni non sono indipendenti fra loro, ma in molti casi una istruzione B deve usare il risultato di una istruzione A. Di conseguenza A e B non possono essere eseguite in parallelo: B deve essere eseguita dopo A.

3.10.3 Simultaneous Multi-Threading (SMT)

Per sfruttare questa caratteristica al meglio, ogni core di una CPU moderna è Multi-Threaded: può infatti eseguire in parallelo istruzioni appartenenti a Peer Thread diversi, aumentando così la velocità di esecuzione dei programmi. Si tratta della tecnica del Simultaneous Multi-Threading (SMT).



Ad ogni ciclo di Clock inizia l'esecuzione di un nuovo gruppo di istruzioni. Se si possono eseguire in parallelo istruzioni che appartengono a diversi Peer Thread, la produttività di ciascun core della CPU aumenta.

Capitolo 4

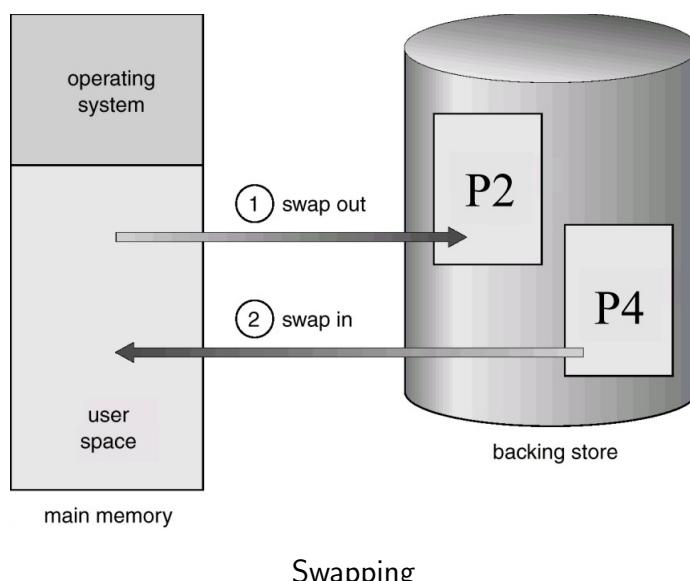
La Memoria Fisica

Abbiamo visto nei precedenti capitoli che i SO tentano di massimizzare le risorse della macchina e l'uso della CPU. Quindi uno degli obiettivi è quello di tenere sempre attiva la CPU, e, di conseguenza, anche di ottimizzare l'utilizzo della memoria.

4.1 Introduzione

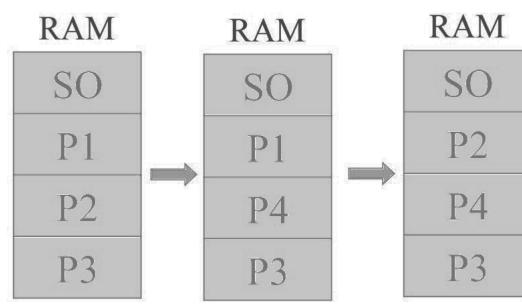
Suppongo che la memoria primaria (RAM) sia completamente occupata da tre processi utente chiamati P1, P2, P3. Se nasce un quadro processo P4 come procedo? Una soluzione banale è aspettare la terminazione di uno dei processi già presenti in RAM, di modo da fare spazio per il quarto. Se uno di questi tre processi fosse in attesa di un I/O, una situazione sarebbe quella di spostarlo temporaneamente in memoria secondaria, in modo da fare posto a P4.

Ma cosa viene spostato esattamente? La chiamiamo Immagine di P: il codice, i dati del processo e lo stack. Questa tecnica viene chiamata Swapping (avvicendamento di processi). L'area del disco in cui il SO copia temporaneamente un processo viene detta Area di Swap.



Swapping

Lo Swapping è raramente utilizzato nei moderni SO poichè è troppo inefficiente, tuttavia mette in luce un importante problema. Poichè infatti un programma possa essere eseguito il suo codice deve trovarsi in memoria primaria. Quando il SO riceve il comando di esecuzione di un programma deve recuperare il codice del programma dalla memoria secondaria e decidere in quale posizione della RAM inserirlo. Ma questo ultimo passaggio non è per nulla banale. Per risolvere questo problema si utilizza il concetto del Binding.



Swapping

4.2 Binding

Quando un programma viene compilato e caricato in Memoria Primaria (MP), per essere eseguito, ad ogni variabile deve essere associato l'indirizzo della locazione di memoria che ne contiene il valore.

Un ragionamento simile viene fatto per le istruzioni di controllo (salti condizionati e incondizionati), per cui ad ognuno di questi va associato l'indirizzo di destinazione del salto. L'operazione di associazione di variabili e istruzioni agli indirizzi di memoria è detto quindi Binding degli Indirizzi. Alla fine quindi diventerà qualcosa del tipo:

```

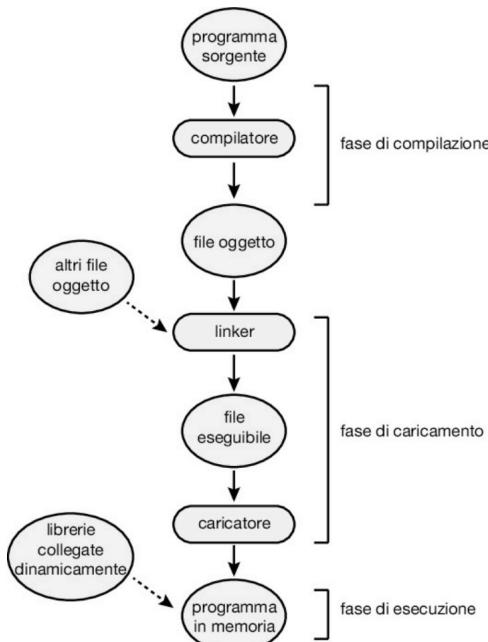
1 // questo
2 counter = counter + 1;
3
4 //diventa questo
5 load(R1, 10456)
6 Add(R1, ]1)
7 store(R1, 10456)

```

Poichè un programma possa essere eseguito deve passare attraverso varie fasi:

- Compilazione
- Caricamento in RAM
- Esecuzione

Eccole riassunte nell'immagine di seguito:



Binding degli Indirizzi

4.2.1 Binding degli Indirizzi in Fase di Compilazione

Se il Binding viene fatto in Fase di Compilazione viene generato Codice Assoluto o Statico. Il compilatore deve conoscere l'indirizzo della cella di RAM a partire dal quale verrà caricato il programma, così da effettuare il Binding degli indirizzi. Se invece vuole metterlo da un'altra parte, bisognerà ricompilare il programma rifacendo il binding degli indirizzi.

4.2.2 Binding degli Indirizzi in Fase di Caricamento in RAM

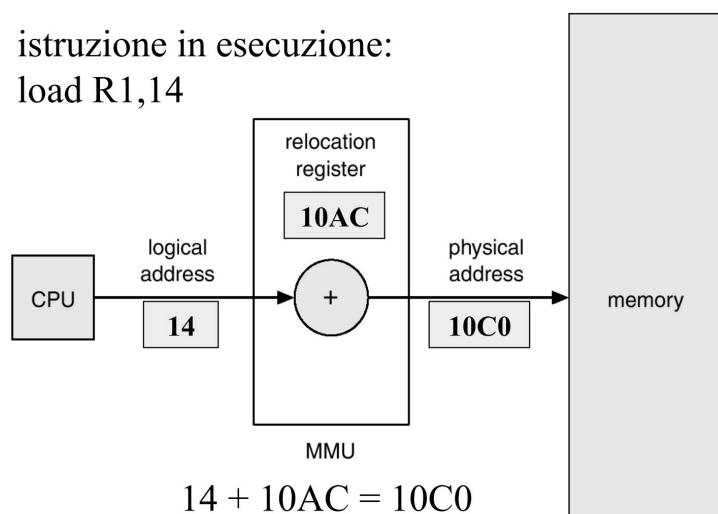
In questo caso si dice che viene generato Codice Staticamente Rilocabile. Il compilatore associa ad istruzioni e variabili degli indirizzi relativi all'inizio del programma, che inizia da un ipotetico 0 virtuale. Gli indirizzi assoluti finali vengono generati in fase di caricamento del codice in MP in base all'indirizzo di MP a partire dal quale è caricato il codice.

Il binding quindi avviene in fase del caricamento in RAM. Se il processo che usa quel codice viene tolto dalla RAM, può caricarlo in una posizione diversa rieffettuando solamente la fase di caricamento.

4.2.3 Binding degli Indirizzi in Fase di Esecuzione

In questo caso si dice che viene generato Codice Dinamicamente Rilocabile. Questo codice usa solamente indirizzi relativi, anche quando si trova in esecuzione. La trasformazione di un indirizzo relativo in uno assoluto avviene nell'istante in cui viene eseguita l'istruzione che usa quell'indirizzo.

Si parla in questo caso di Binding Dinamico degli Indirizzi. Tuttavia abbiamo bisogno di diversi elementi, in particolare un Registro di Rilocazione che serve per trasformare un indirizzo relativo nel corrispondente indirizzo assoluto durante l'esecuzione delle istruzioni. Serve inoltre la Memory Management Unit (MMU) che trasforma gli indirizzi relativi in assoluti usando appunto i registri di rilocazione.



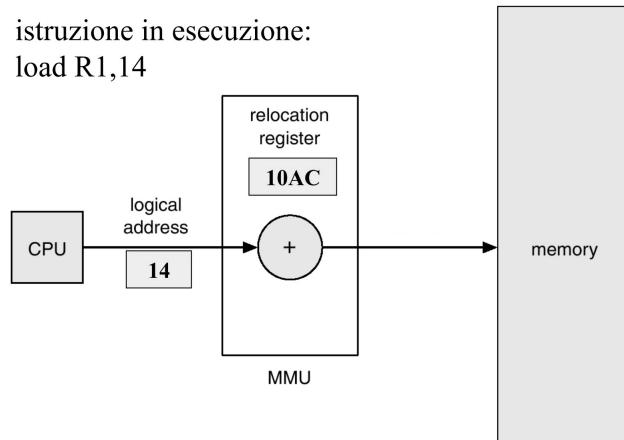
Indirizzi Dinamicamente Rilocabili

Questo è l'approccio utilizzato da tutti i Sistemi Operativi moderni.

4.2.4 Spazio degli Indirizzi Logici e Fisici

Considero Codice Dinamicamente Rilocabile, che da ora in poi sarà lo standard. Qualsiasi indirizzo usato nel codice avrà un valore compreso da 0 e quello dell'ultima cella di memoria occupata dal codice. Chiamo questo insieme di indirizzi Spazio di Indirizzamento Lodico o Virtuale del programma.

Il codice poi viene caricato in RAM ed eseguito, gli indirizzi usati nel codice rimarranno relativi. Parlerò quindi di Indirizzi Logici o Virtuali come quegli indirizzi generati dal processore che "escono" da questo per andare in RAM.



Spazio degli Indirizzi Logici e Fisici

Gli indirizzi però devono essere trasformati in Assoluti per funzionare. Per fare ciò devo passare per il Registro di Rilocazione e otterrò così gli Indirizzi Fisici. Ma allora posso chiamare Spazio di Indirizzamento Fisico l'insieme degli indirizzi fisici utilizzati.

Ho quindi due tipi di indirizzi:

- Indirizzi Logici: vanno da 0 a *max*.
- Indirizzi fisici: vanno da $r + 0$ e $r + max$, dove r è l'indirizzo in memoria a partire dal quale è stato caricato il programma in RAM.

Devo tenere a mente che quando in futuro parlerò di Spazio di Indirizzamento Logico o Fisico, mi riferirò sempre a quello della macchina, non a quello dei programmi singoli della macchina.

4.3 Le Librerie

Le Librerie sono collezioni di codice messo a disposizione dei programmatori per facilitargli la vita. Esistono due tipi di Librerie:

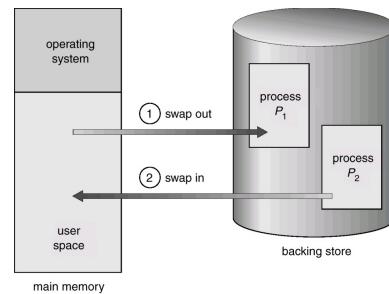
- Librerie Statiche: il codice viene collegato al codice del programma, quindi diventano parte integrante del programma, anche quando non vengono utilizzate.
- Librerie Dinamiche: il codice viene collegato solo nel momento in cui vengono invocate. Sono molto più popolari delle prime.

Le Librerie Dinamiche vengono anche chiamate Librerie Condivise poiché permettono di evitare lo spreco di spazio in memoria primaria.

4.4 Tecniche di Gestione della Memoria Principale

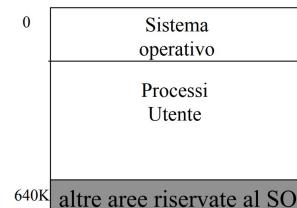
Vedo ora alcune delle tecniche principali di gestione della Memoria Principale. Alcune di queste non sono più in uso ma mi aiuteranno comunque a comprendere meglio concetti più complessi:

Swapping Per Swapping o Avvicendamento dei Processi, si intende il salvare in memoria secondaria l'immagine di un Processo non in esecuzione (Swap Out) e ricaricarla (Swap In) prima di dargli la CPU.

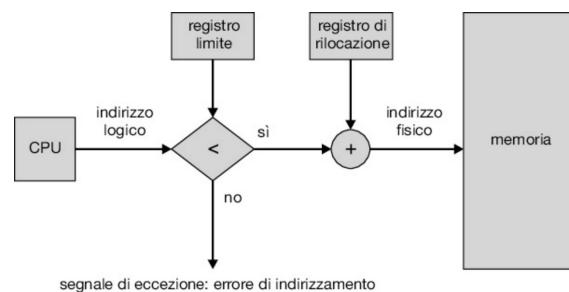


Lo Swapping permette di avere attivi più processi di quanto ne possa contenere lo spazio nella MP. Questa tecnica però non viene usata nei moderni SO, perché è troppo dispendiosa. Tuttavia, l'idea di fondo dello Swapping rimane sostanzialmente valida.

Allocazione Contigua In qualsiasi computer, la memoria principale è divisa in partizioni, assegnate al SO e ai processi. Nel caso più semplice, l'area non assegnata al So viene occupata da un solo processo e in questo caso la protezione della MP coincide con la protezione delle aree di memoria del SO.



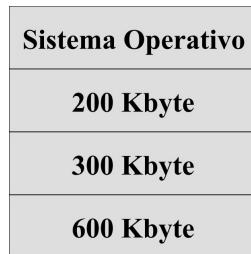
In questo caso viene usato un Registro Limite con cui verrà confrontato ogni indirizzo usato dal processo in esecuzione, che deve essere inferire al valore scritto nel registro limite. Ho bisogno però anche del Registro di Rilocazione per costruire il vero e proprio Indirizzo Fisico.



Allocazione a Partizioni Multiple Fisse La memoria è divisa in Partizioni di dimensione fissa e ogni partizione contiene un unico processo, dall'inizio alla fine dell'esecuzione. Il numero delle partizioni stabilisce il grado di multiprogrammazione e quando un processo termina, il suo posto viene preso da un altro processo.

Questo meccanismo può essere usato per proteggere le varie partizioni ad accessi proibiti.

Quando avviene il Context Switch, il Dispatcher carica nel Registro di Rilocazione l'indirizzo di partenza della partizione del processo a cui viene data la CPU e carica nel registro limite la dimensione di quella partizione:

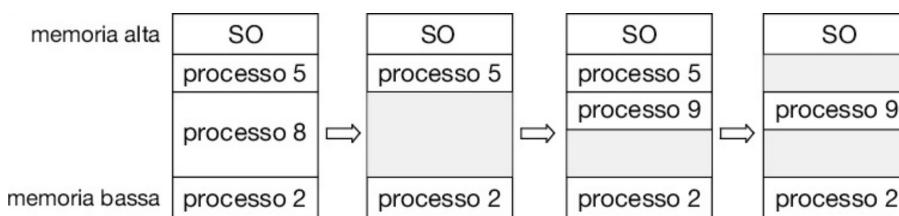


Questa era una tecnica utilizzata nei vecchi sistemi operativi, attualmente non è più in uso poichè presenta troppi svantaggi, in particolare:

- Il grado di multiprogrammazione è limitato dal numero di partizioni previste.
- Difficilmente un processo ha esattamente la dimensione di quella partizione. Se è più piccolo c'è uno spreco, che prende il nome di Frammentazione Interna.
- La Frammentazione Interna causa la Frammentazione Esterna, ovvero lo spreco globale dovuto allo spreco locale.
- Se nasce un processo più grande rispetto alla dimensione massima non può girare.

Allocazione a Partizioni Multiple Variabili La scelta apparentemente migliore è di usare Partizioni a Dimensioni Variabile, dove ad ogni processo viene data una partizione dell'esatta dimensione di quest'ultimo.

Questo però non funziona, poichè quando un processo termina lascia un "buco" in RAM, e un altro processo può occupare una parte dello spazio liberato, come in figura:



Per gestire al meglio questo tipo di Allocazione, il SO deve tenere a mente tutti i buchi liberi e occupati, e aggiornare l'informazione ogni volta che nasce o termina un processo.

Le tre tecniche fondamentali per assegnare un processo ad una partizione è la seguente:

- First Fit: prendo la prima partizione abbastanza grande da poter ospitare un processo.
- Best Fit: prendo la partizione che mi lascia il buco minore possibile.
- Worst Fit: prendo la partizione più grande.

Il Worst Fit è quella che funziona peggio, le altre due sono più o meno le migliori in termini di sprechi, quindi si è deciso di usare First Fit perchè è la più rapida.

Tuttavia, il problema della Frammentazione Esterna rimane, col tempo infatti si creeranno per forza dei buchi che non possono ospitare nessun processo. C'è anche un problema di Frammentazione Interna perchè costa troppo tenere traccia dai buchi molto piccoli.

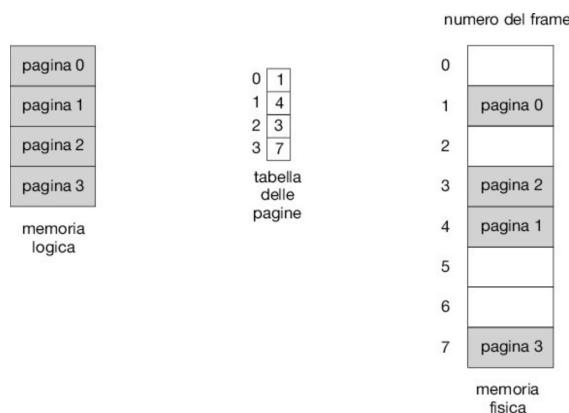
C'è anche un altro problema, perchè dopo molto tempo si arriva ad un momento in cui la memoria è talmente frammentata che diventa inusabile. Devo quindi compattare la memoria, spostando le partizioni occupate di modo che queste siano tutte vicine tra di loro e in modo da creare un solo "grande buco".

4.5 Paginazione della Memoria

Si tratta della tecnica utilizzata dai moderni sistemi operativi. L'idea della Paginazione della Memoria è ammettere che l'area di memoria allocata ad un processo possa essere divisa in diversi pezzi non contigui da loro (esempio latte valigia).

La Memoria Primaria viene divisa in diversi "pezzi" chiamati Frame. La dimensione di questi Frame è sempre una potenza di 2. Lo Spazio di Indirizzamento Logico viene sempre visto dal processo come uno spazio contiguo ma viene diviso in pagine di dimensione identica ai Frame.

Per poter eseguire un processo che occupa x pagine, il SO cerca x frame liberi in cui caricare le pagine. Questi x frame possono anche non essere adiacenti e le pagine possono anche non essere inserite in ordine, questo perchè ad ogni processo infatti viene associata una Tabella delle Pagine (Page Table, PT), ovvero un array che contiene i numero dei frame in cui il SO ha caricato in RAM le pagine del processo.



Nasce però un problema fondamentale, ovvero che gli indirizzi usati dal programma vengono "tagliati in tanti pezzi" e inseriti "a casaccio" nella memoria, sembrano non funzionare più. Per far funzionare tutto, dobbiamo considerare non più gli Indirizzi Logici come indirizzi "lineari" (ossia da 0 all'ultimo byte usato dal programma).

Per risolvere, bisogna trattare gli Indirizzi Logici come delle coppie di valori in cui:

- Il primo elemento della coppia specifica il numero della pagina.
- Il secondo elemento della coppia specifica la posizione (offset) della cella di memoria che vogliamo indirizzare rispetto ad un ipotetico indirizzo.

Dunque un Indirizzo Logico ha ora la forma: (pagina, offset).

4.5.1 Traduzione degli Indirizzi

Un Indirizzo Logico viene tradotto in uno fisico così:

1. Il numero di pagina p viene usato come indice nella Page Table del processo per sapere in quale frame f è contenuta la pagina.
2. Una volta noto il frame f , l'offset d può essere applicato a partire dall'inizio del frame per indirizzare il byte specificato dalla doppia p, d .

Vedo più nel dettaglio come fare: ogni informazione nel computer è una sequenza di bit, per esempio da 12 bit. Adesso però gli indirizzi logici abbiamo detto che sono coppie di valori, quindi devo decidere come separare i bit. La scelta del numero di bit da utilizzare per scrivere il numero della pagina e quello dell'offset dipendono dall'hardware su cui deve girare il SO, che impone:

- Il numero di bit su cui va scritto un indirizzo logico. (pongo per esempio m bit)
- La dimensione di un frame, e quindi di una pagina. (pongo per esempio 2^n byte).

Quindi, dovrò usare n bit per scrivere l'offset all'interno di una pagina/frame, e il numero di bit usati per scrivere il numero di una pagina sarà pari a $m - n$ bit. La dimensione dello spazio di indirizzamento logico è quindi $2^{m-n} \times 2^n = 2^m$ byte.

Ora, definisco più nello specifico l'operazione degli indirizzi logici in indirizzi fisici. Come detto in precedenza, un indirizzo fisico è formato da due parti (p, d):

- Numero di Pagina (p): viene usato come indice per selezionare la entry della PT in cui si trova il numero del frame in cui è caricata la pagina.
- Offset di Pagina (d): viene usato all'interno del frame per spostarsi nel punto esatto del frame specificato dall'indirizzo logico.

Posso però riformulare la stessa cosa in maniera leggermente diversa:

- Numero di Pagina (p): viene usato come indice per selezionare la entry della PT in cui si trova l'indirizzo base del frame.
- Offset di Pagina (d): viene sommato all'indirizzo base del frame per generare l'indirizzo fisico.

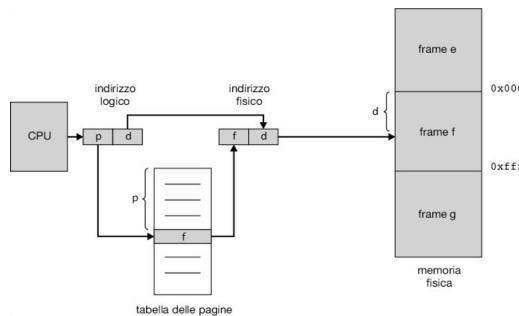
Ora posso mettere assieme tutto quello che ho detto finora per vedere come gli indirizzi logici possono essere visti come dei Valori Lineari, ma possono anche essere visti come coppie di valori.

4.5.2 I Vantaggi della Paginazione

La paginazione implementa automaticamente una forma di protezione dello Spazio di Indirizzamento. Un processo quindi non può far altro che indirizzare all'interno della Memoria Principale celle di memoria che appartengono a sè stesso, non c'è quindi modo al Processo di uscire dal proprio ambito.

La paginazione inoltre evita la frammentazione esterna, perché qualsiasi frame libero può essere usato per contenere una pagina.

Inoltre, la paginazione è una forma di rilocazione dinamica dove ad ogni pagina corrisponde un diverso valore del registro di rilocazione. Qui infatti, il meccanismo è più sofisticato, e ognuna delle Entry della tabella delle pagine corrisponde l'indirizzo di partenza del Frame a cui verrà poi sommato un offset per ricavare l'Indirizzo Fisico. Con questo meccanismo è inutile il controllo del registro limite, perché l'offset non può proprio costruire un indirizzo che vada oltre quello possibile.



Nel corso degli anni la dimensione delle pagine è aumentata. Ovviamente più è grande la pagina, maggiore è la frammentazione interna. Questo però produce tabelle delle pagine più corte, perché tagliamo i processi in meno pezzi.

4.5.3 Gli Svantaggi della Paginazione

Ad ogni processo è associata una tabella delle pagine che ovviamente occupa spazio in RAM, il So deve mantenere anche una Frame Table, di conseguenza deve sapere anche quelli frame sono liberi e quali sono occupati. Ad ogni Context Switch il SO deve poi attivare la tabella delle pagine a cui viene assegnata la CPU.

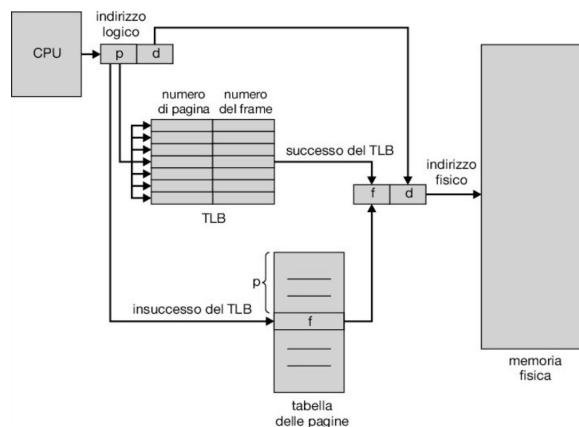
Tutto questo aumenta il lavoro del SO. Il meccanismo di traduzione degli indirizzi da logici a fisici deve essere quindi più efficiente possibile, il vero problema riguarda quindi la gestione della tabella delle pagine. Qualora il numero di pagine per processo è piccolo, possiamo memorizzare la PT nei registri della CPU, ma nei moderdi SO che devono usare migliaia di elementi questo non è possibile. Per forza di cose bisogna memorizzare la PT in RAM. Il problema grave, tuttavia, è che per tradurre un indirizzo occorre passare attraverso una Entry che si trova in memoria primaria. Dunque, il numero di accessi in MP raddoppia, creando grandi problemi dal punto di vista prestazionale.

La soluzione a questo grave problema, consiste nell'utilizzare una tecnica di Caching, quindi mantenere la PT in una cosiddetta Memoria Associativa messa a disposizione della CPU, chiamata Translation Look-Aside Buffer (TLB).

4.5.4 Translation Look-Aside Buffer (TLB)

Il funzionamento di una memoria associativa consiste in tante coppie di Celle chiamate "Chiave" e "Valore". La chiave in ingresso viene confrontata contemporaneamente con tutte quelle disponibili e viene spedito in output il Valore associato alla chiave.

Nella TLB viene caricata una porzione della PT usando il numero di pagina come chiave, e il frame relativo come valore. Se la ricerca ha successo (Hit) bene, se non la si trova (Miss), bisognerà andare a vedere nella Memoria Principale, impiegando però più tempo.



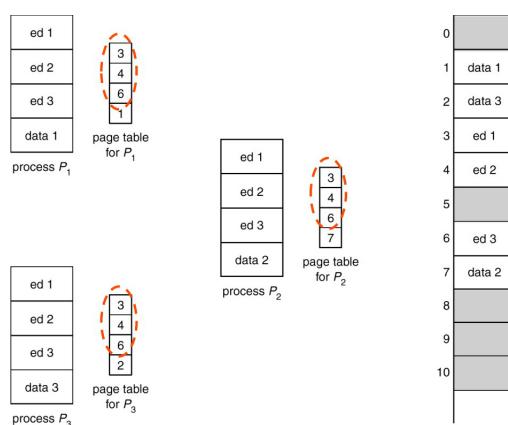
Chiamo Hit Ratio la percentuale di successi (Hit) nell'uso del TLB. Per esempio, suppongo di dover spendere 10 nanosecondi per accedere alla RAM una volta tradotto l'indirizzo, che la degradazione delle prestazioni è il 20% e che l'Hit Ratio sta all'80%:

$$10 \text{ nsec} \times 0,80 + (10 + 10) \text{ nsec} \times 0,20 = 12 \text{ nsec}$$

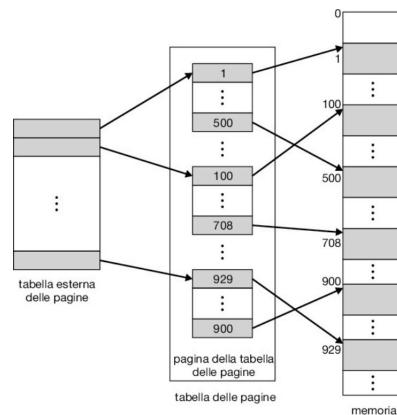
In caso di Miss la coppia pagina-frame mancante viene recuperata attraverso la PT in RAM e viene messa nel TLB, sostituendone un'altra in base al Least Recently Used.

Alcuni processori hanno addirittura due livelli di cache TLB.

La paginazione facilita la condivisione di codice perché questo non cambia durante l'esecuzione (si chiama codice Puro).



Una possibile soluzione al problema dell'allocazione di PT che possono occupare molto spazio consiste nel paginare le PT, quindi spezzarle in pagine memorizzate in frame non necessariamente adiacenti alla RAM. Si parla quindi di paginazione a due livelli: la PT vera e propria la chiamo PT Interna, l'altra PT Esterna.



Capitolo 5

La Memoria Virtuale

I metodi di gestione della Memoria Principale tentano di mettere in memoria più processi possibili. Ovviamente però non è possibile mantenere in RAM più processi di quella che è la dimensione totale.

La Memoria Virtuale è l'insieme di tecniche che permette l'esecuzione di processi in cui il codice non sono caricati completamente in Memoria Primaria.

5.1 Introduzione

La Memoria Virtuale funziona poichè i programmi non hanno bisogno di essere caricati interamente in Memoria Principale per essere eseguiti. L'idea di base è quindi di caricare in RAM un pezzo di programma solo se questo deve effettivamente essere eseguito, e solo quando deve essere eseguito. Allo stesso tempo, cerco in Memoria Principale solo la porzione di dati che effettivamente mi serve.

La conseguenza di ciò è che posso eseguire un processo che sfrutta uno spazio di indirizzamento logico maggiore di quello fisico. Posso inoltre avere contemporaneamente in esecuzione processi che messi assieme occupano più spazio della Memoria Principale disponibile. Un altro vantaggio è che i programmi vengono lanciati più velocemente, poichè non bisogna caricarli completamente in RAM.

Gli svantaggi sono un aumento del traffico tra RAM e HDD, in altre situazioni particolari le prestazioni complessive del sistema possono degradare drasticamente, in un fenomeno chiamato Trashing.

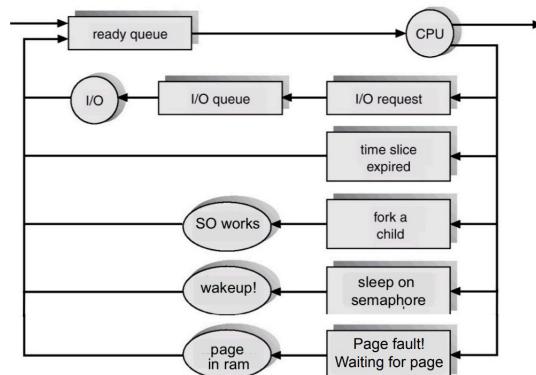
5.2 Tecniche di Gestione della Memoria Virtuale

L'idea di base è portare in MP una pagina del processo che bisogna eseguire solamente nel momento del primo indirizzamento di una locazione (un dato, una istruzione) appartenente alla stessa pagina. Quando la CPU esegue un'istruzione che indirizza una pagina diversa da quella che contiene l'istruzione in esecuzione, e la pagina non è in MP, si dice che il processo ha generato un Page Fault (la pagina manca).

A questo punto il Sistema Operativo deve:

1. Sospendere il processo.
2. Portare in memoria la pagina mancante.
3. Quando arriva il suo turno, far ripartire il processo dal punto in cui era stato sospeso.

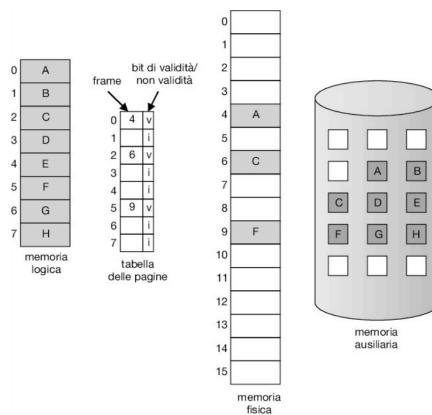
Ora, nel Diagramma di Accodamento, il caso Wait for an Interrupt lo posso associare ad un processo Waiting for Page:



Più in dettaglio, ecco cosa succede quando manca la pagina:

- Il processo viene tolto dalla CPU e messo in uno stato di "Waiting for Page".
- Un modulo del SO chiamato Pager inizia il caricamento della pagina mancante in un frame libero della MP.
- La CPU viene assegnata ad un altro processo.
- Quando la pagina è in MP, il processo viene rimesso in Coda di Ready.

Per sapere se una pagina non è caricata in RAM, il SO usa un Bit di Validità che dice se la pagina associata a quella entry è effettivamente in RAM o meno. Se si tenta di fare riferimento a una pagina non in MP il Bit sarà 0, 1 altrimenti. Ecco una rappresentazione:



Addirittura un processo può essere fatto partire senza nessuna delle sue pagine in Memoria Principale. Alla prima istruzione indirizzata dal PC si genera un Page Fault: il Program Counter punta all'indirizzo di una pagina del processo non in Memoria Principale, in uno schema detto Pure Demand Paging. Alternativamente, il SO può caricare inizialmente in MP almeno una pagina.

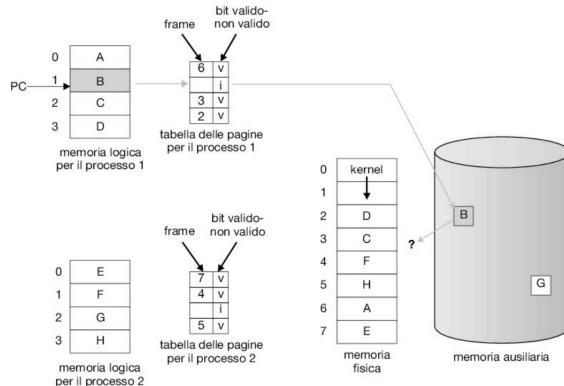
Per implementare la Memoria Virtuale è necessario un certo supporto hardware, in particolare di una Tabella delle Pagine con il Bit di Validità testabile dall'hardware per generare Page Fault. Quindi mentre la paginazione può essere sempre implementata e in qualsiasi sistema, per la MV ho bisogno di un supporto Hardware adeguato.

5.2.1 Area di Swap

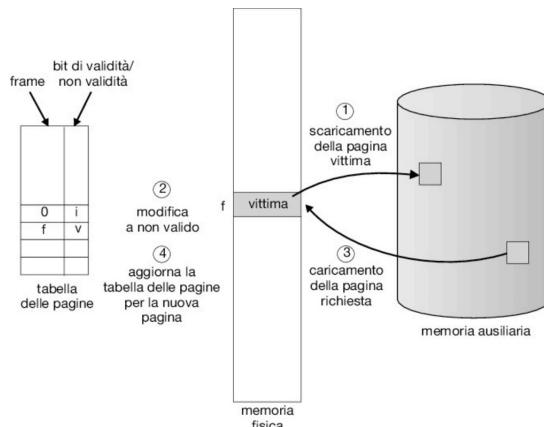
Per funzionare, la MV ha anche bisogno di una opportuna porzione dell'HDD chiamata Area di Swap. Quest'area serve per gestire gli scambi, ed è fatta apposta per gestirli in maniera molto efficiente. All'avvio di un processo, il suo eseguibile viene copiato interamente nell'area di Swap. In alternativa, l'idea è di prelevare le pagine dell'eseguibile o di un eventuale file di dati direttamente dal File System, in modo da ridurre ancora di più i tempi.

L'area di Swap viene usata anche e soprattutto per liberare spazio in Memoria Principale e ospitare le pagine vittime, che devono essere caricate in RAM poiché la loro assenza ha causato un Page Fault.

L'idea della MV è proprio quella di eseguire contemporaneamente processi che insieme occupano più spazio di quello disponibile in RAM:



Dunque se si verifica un Page Fault e tutti i frame della RAM sono occupati, bisogna liberarne uno rimuovendo la pagina che ospita, che prende il nome di Pagina Vittima. Se però questa pagina contiene dati modificati va prima copiata e salvata nell'Area di Swap di modo che poi possa essere recuperata e riportata in MV.



Se invece la Pagina Vittima non è stata modificata non c'è bisogno di salvarla. Viene utilizzato un cosiddetto Dirty Bit associato ad ogni Entry che ci comunica se la pagina relativa è stata modificata o meno.

5.2.2 Algoritmi di Sostituzione

Per valutare l'efficacia di diversi Algoritmi di Sostituzione posso usare delle sequenze di riferimenti in Memoria Principale, che possono essere state generate in modo casuale o dall'esecuzione di programmi reali.

Per esempio, una sequenza di riferimento potrebbe essere la seguente:

10, 7, 4, 5, 6, 1, 10, 4

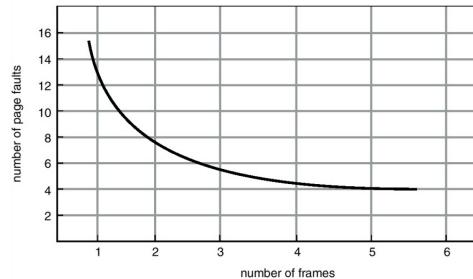
Ossia, durante l'esecuzione delle istruzioni di un processo, la CPU ha generato una sequenza di indirizzi logici che indirizzano qualcosa contenuto nella pagina 10, poi 7 eccetera.

Il numero di Page Fault che genera questa sequenza di riferimento dipende dal numero di frame disponibili. Se abbiamo una memoria con un solo frame disponibile, la sequenza provoca 8 Page Fault, per esempio. Idem se ho questa sequenza:

10, 7, 7, 7, 4, 5, 5, 5, 6, 1, 10, 10, 10, 4

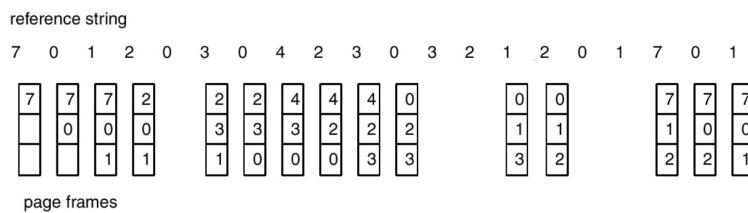
Infatti, dopo aver caricato in Memoria Principale la pagina 7, tutti i successivi riferimenti consecutivi alla stessa pagina non provocano ulteriori Page Fault.

Il numero di Page Fault intuitivamente diminuisce all'aumentare del numero di frame, con qualche rara eccezione:

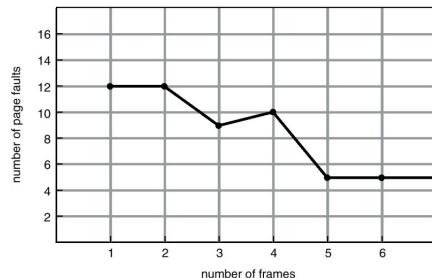


Prima di elencare gli algoritmi, bisogna assumere che ad ogni processo è assegnato un certo numero di frame e che una pagina vittima è selezionata tra le pagine del processo stesso. Si parla quindi di Sostituzione Locale delle Pagine.

Sostituzione delle Pagine secondo l'Ordine di Arrivo (FIFO) Questo algoritmo sceglie come Pagina Vittima quella arrivata da più tempo in Memoria Principale. Si tratta di una algoritmo molto semplice da implementare che tuttavia è molto dispendioso in termini di prestazioni.

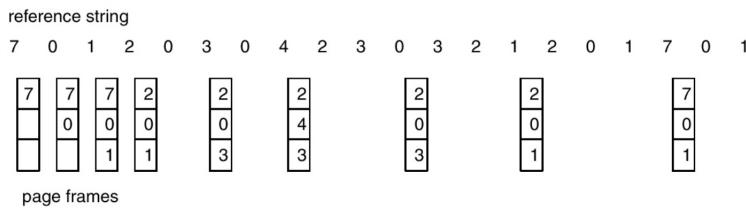


L'algoritmo soffre inoltre della cosiddetta Anomalia di Belady: usando più frame il numero di Page Fault più infatti aumentare:

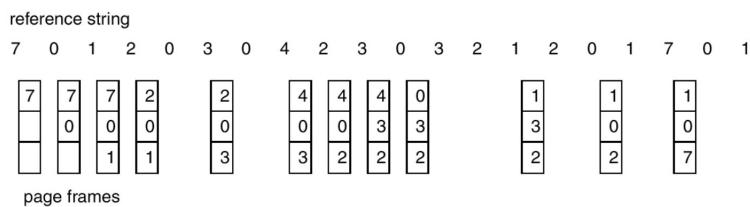


Sostituzione Ottimale delle Pagine (OTP) Si tratta di un algoritmo che non soffre dell'Anomalia di Belady e che soprattutto produce il numero minimo di Page Fault. In questo algoritmo infatti la Pagina Vittima è quella che sarà usata più al di là nel tempo.

Ovviamente però si tratta di un algoritmo non implementabile.

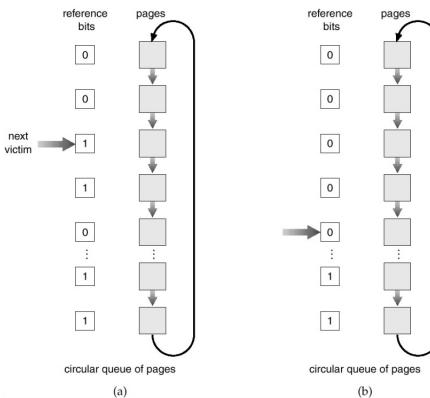


Sostituzione delle Pagine Least Recently Used (LRU) Questo algoritmo sostituisce la pagina che non è stata usata da più tempo. Si tratta di un algoritmo difficile da implementare in modo efficiente perché necessita supporti hardware difficili da implementare nei processori moderni:



Per implementare l'algoritmo si può però utilizzare un Reference Bit, ovvero un bit che va a 1 quando un processo indirizza una certa pagina. In questo modo, in ogni istante conosco quali delle pagine di un processo sono state usate e quali no.

Algoritmo della Seconda Chance In questo algoritmo utilizza FIFO e il Reference Bit. Se quest'ultimo è a 0 allora questo diventa una pagina vittima, se invece è a 1 il SO gli dà una seconda chance e il suo Reference Bit ritorna a 0.



Algoritmo della Seconda Chance Migliorato Se l'Hardware fornisce sia il Reference bit che il Dirty Bit, le pagine possono essere raggruppate in 4 classi:

- (0, 0) nè usata di recente nè modificata: ottima da rimpiazzare.
- (1, 0) usata di recente ma non modificata: meno buona da rimpiazzare.
- (0, 1) non usata di recente ma modificata: ancora meno buona, vù salvata in Memoria Secondaria.
- (1, 1) usata di recente e modificata: la peggiore candidata al rimpiazzamento.

5.2.3 Allocazione Globale e Locale

In che gruppo di pagine possiamo e doviamo scegliere la Vittima da rimuovere dalla RAM? Abbiamo due possibilità:

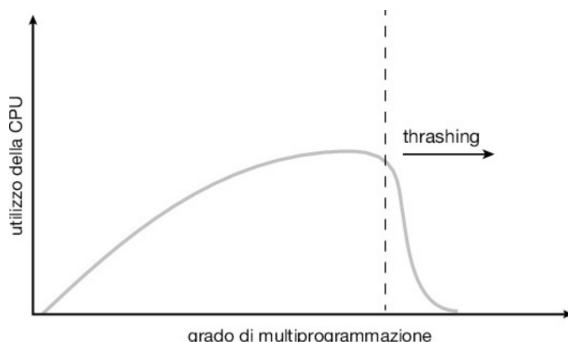
- Allocazione Globale: scelgo la vittima tra tutte le pagine della MP. Il difetto è che il Turnaround è molto influenzato dal comportamento degli altri processi con cui convive.
- Allocazione Locale: scelgo la vittima tra le pagine del processo che ha generato Page Fault. Il difetto è che si danno troppe pagine ad un processo e si può peggiorare il throughput del sistema.

5.3 Trashing

Considero un sistema in cui ogni processo ha a disposizione pochi frame, ossia ogni processo ha in RAM un piccolo numero di pagine rispetto al totale di pagine di cui è composto. Suppongo ora di adottare una Allocazione Globale dei Frame.

Avendo poche pagine in RAM, ogni processo ha un'alta probabilità di generare un Page Fault. Quando questo succede, una Pagina Vittima viene tolta dalla Memoria Principale, probabilmente ad un altro processo. Questo altro processo ha ora in RAM ancora meno pagine di prima e quindi si alza ancora la probabilità di generare un Page Fault. Si crea quindi un circolo vizioso in cui vengono generati continuamente dei Page Fault. Questo fenomeno si chiama appunto Trashing.

Più è alto il grado di multiprogrammazione (in modo da sfruttare il massimo di CPU), e più è più frequente questo problema:

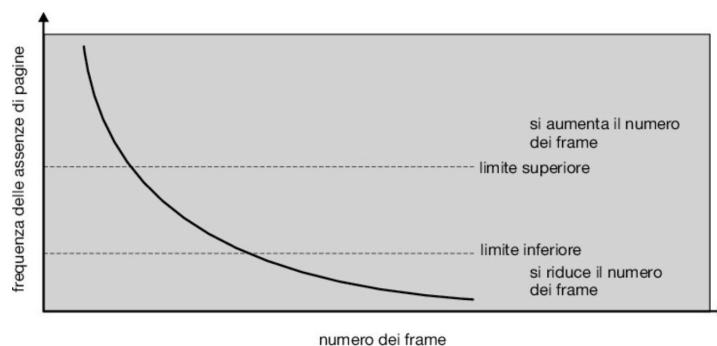


5.3.1 Cause del Trashing

Se il livello di utilizzo della CPU è troppo basso, lo si può alzare permettendo di lanciare un maggior numero di processi. In questo modo però, i nuovi processi cominciano a sottrarre pagine ai processi già presenti. Fino a un certo punto l'aumento dei processi è tollerato, ma solo fino a un certo punto, appunto.

5.3.2 Antidoti e Prevenzione al Trashing

La soluzione è quella di diminuire il grado di multiprogrammazione temporaneamente, in maniera da dare tempo ai processi non rimossi dalla Memoria Principale di terminare correttamente prima di far ripartire gli altri.



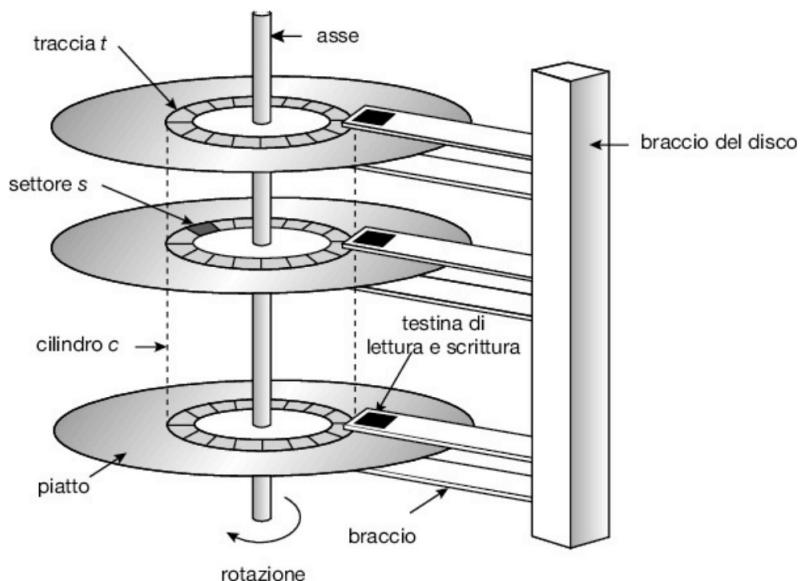
Per prevenire il fenomeno del Trashing, ovviamente la soluzione migliore è quella di dotare il Sistema di una quantità sufficiente di Memoria Principale. Per mitigarlo invece, dovrebbe bastare l'usare una politica di sostituzione locale.

Capitolo 6

La Memoria di Massa

La Memoria di Massa, ovvero l'Hard Disk, è composto da una serie di dischi o piatti sovrapposti. Ognuno di questi piatti è suddiviso in una serie di tracce circolari e ognuna di queste tracce è a sua volta suddivisa in una serie di settori. L'insieme di queste tracce nella stessa posizione sui diversi piatti prende il nome di cilindro.

L'HDD ha anche un "braccio del disco", ovvero un braccio che supporta una testina di lettura scrittura che si muovono assieme sui vari settori del piatto corrispondente.



6.1 Struttura del Disco Rigo

Ogni settore del Disco Rigo memorizza un blocco di dati. I piatti dell'HDD ruotano tutti assieme rispetto al proprio asse e la testina di lettura/scrittura sfiora la superficie ad una distanza di pochi micron.

Questa testina può compiere le sue operazioni su un settore solamente quando si trova esattamente sotto la testina stessa. Dunque, il tempo di accesso ad un settore dipende da due componenti principali:

- Seek Time (tempo di posizionamento): tempo impiegato per spostare la testina nella posizione desiderata.
- Rotation Latency (latenza rotazionale): tempo impiegato dal piatto per portare il settore interessato sotto la testina.

Da un punto di vista logico, un HD può essere visto come un array di blocchi logici.

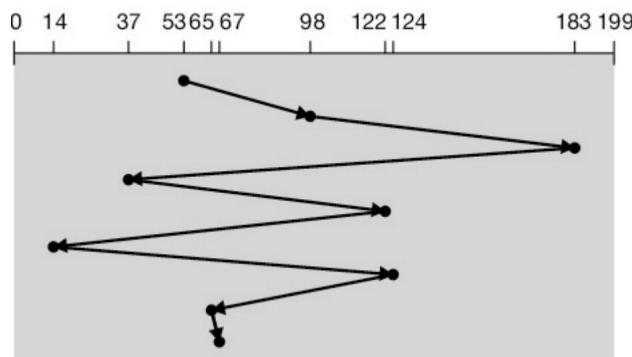
6.1.1 Scheduling dei Dischi Rigidi

In generale, il SO riceve da parte dei processi molte richieste di accesso a dati memorizzati sul disco, e quindi va organizzato il trasferimento delle informazioni in modo da ottimizzare le prestazioni di accesso al disco.

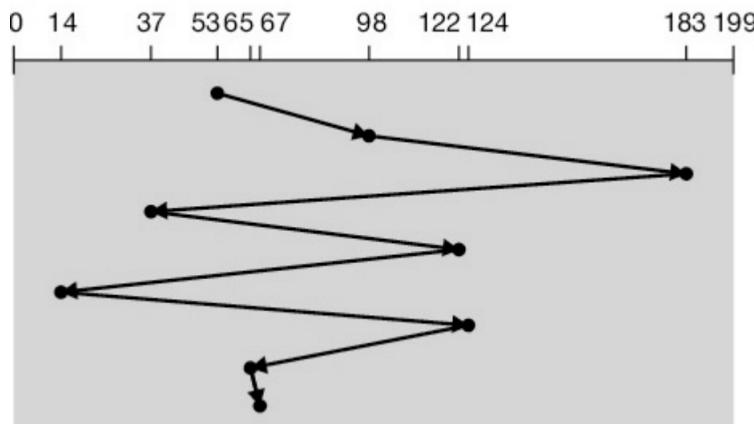
Il Sistema Operativo però non può influenzare la latenza rotazionale del disco, ma può cercare di minimizzare il Seek Time complessivo, ordinando le richieste in modo che le testine si debbano muovere il meno possibile.

Esistono quindi più algoritmi di Scheduling delle richieste di I/O del disco.

Scheduling FCFS Si tratta del classico algoritmo di First Come First Served, quindi il primo processo ad essere eseguito è esattamente quello che per primo richiede l'uso della CPU.



Scheduling C-SCAN Si tratta dell'algoritmo Circular SCAN. Questo fornisce un tempo di attesa più uniforme di altri algoritmi. La testina si muove da un estremo all'altro del piatto servendo le richieste. Quando arriva alla fine, torna immediatamente all'inizio senza servire richieste.



6.1.2 Formattazione del Disco

Per poter essere usato, un disco ha bisogno di fornire un processo di Formattazione a Basso Livello (Formattazione Fisica). Il Sistema Operativo sottopone poi l'HD ad una Formattazione Logica, necessaria per creare e gestire sull'HDD il File System del Sistema Operativo.

Il SO quindi crea la lista dei blocchi liberi e una directory iniziale. Vengono poi riservato sull'HDD le aree che dovranno essere gestite direttamente dal Sistema Operativo:

- Il Boot Block (blocco di avviamento).
- L'area che contiene gli attributi del file.

Il Boot Block contiene il codice necessario per far partire il Sistema Operativo. All'accensione, un piccolo programma contenuto in ROM istruisce il Disk Controller in modo da trasferire il contenuto del Boot Block in RAM. A questo punto il controllo viene trasferito al codice del Boot Block che si occupa di far partire l'intero SO.

6.1.3 Gestione dell'Area di Swap

Durante la Formattazione Logica dell'HDD, il Sistema Operativo riserva a se stesso uno spazio da usare come Area di Swap. Nel caso più semplice, l'Area di Swap può essere un file molto grande all'intero del File System. Windows usa quindi questa soluzione, e lo Swap File si chiama "pagefile.sys".

In maniera alternativa, una porzione specifica dell'HD può essere riservata per l'Area di Swap. Poiché tutto funziona adeguatamente, è altamente preferibile sovradimensionare l'Area di Swap, così che il Sistema Operativo trovi sempre in fretta un'area libera per lo Swap delle pagine e dei segmenti.

6.2 Sistemi RAID

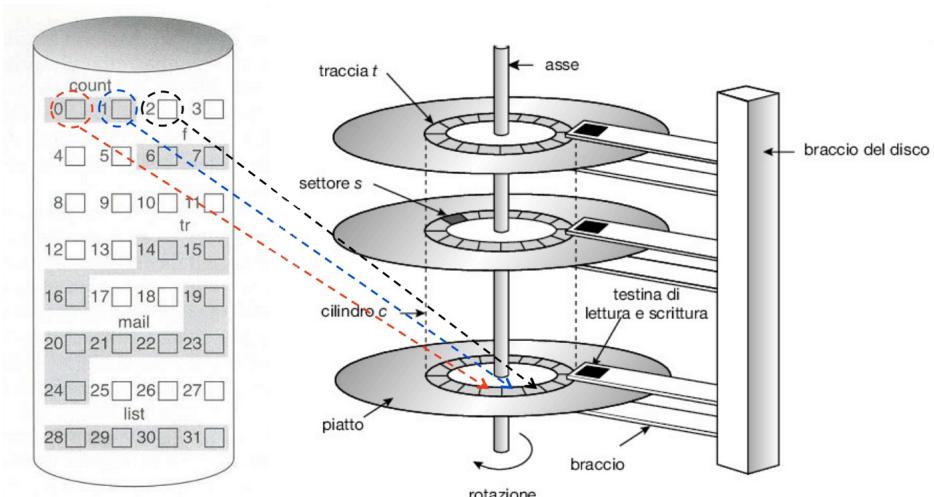
Un Hard Disk, ma anche un SSD, è più lento del processore e della memoria primaria. Il guasto di un HDD è però potenzialmente più dannoso: se infatti si guasta e non c'è un back-up, tutti i dati che contiene vanno persi.

Un Sistema RAID è il modo di configurare la memoria secondaria che aumenta le prestazioni degli Hard Disk e la loro affidabilità. Queste architetture sono fondamentali soprattutto in contesti in cui non ci si può permettere di perdere il servizio fornito, ad esempio in campo finanziario e bancario.

RAID è l'acronimo di Redundant Array of Inexpensive Disks, oppure Redundant Array of Independent Disks. In modo simile alla contrapposizione RISC/CISC fu definita anche la controparte SLED (Single Large Expensive Disk).

Un sistema RAID è composto da un insieme di Hard Disk ma viene visto dal Sistema Operativo come un normale disco singolo, ma più veloce e affidabile di uno SLED. Il controller del RAID gestisce i dischi in modo che il Sistema Operativo li veda come un unico dispositivo di memorizzazione. Le idee di fondo di questo sistema sono due:

1. Distribuire l'informazione su più dischi per parallelizzare le operazioni e migliorare le prestazioni.
2. Duplicare l'informazione memorizzata su più dischi per prevenire le perdite in caso di danneggiamento.



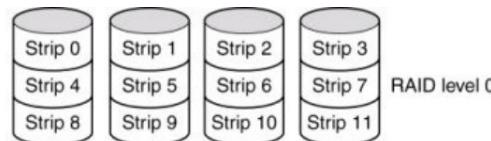
RAID Livello 0 I Sistemi RAID di livello 0 essenzialmente non sono Sistemi RAID, in quanto non c'è una duplicazione dei dati.

In un sistema RAID di livello 0 il disco virtuale (cioè ciò che viene visto dal SO) viene mappato dalla logica del RAID sui settori dei vari dischi in cui è composto il sistema suddividendo i blocchi logici del disco virtuale in Strips di k blocchi consecutivi ciascuna. Quindi, lo Strip 0 contiene i blocchi logici da 0 a $k - 1$, lo Strip 1 da k a $2k - 1$ e così via.

Il RAID controller suddivide poi gli strip tra i dischi del sistema secondo la formula:

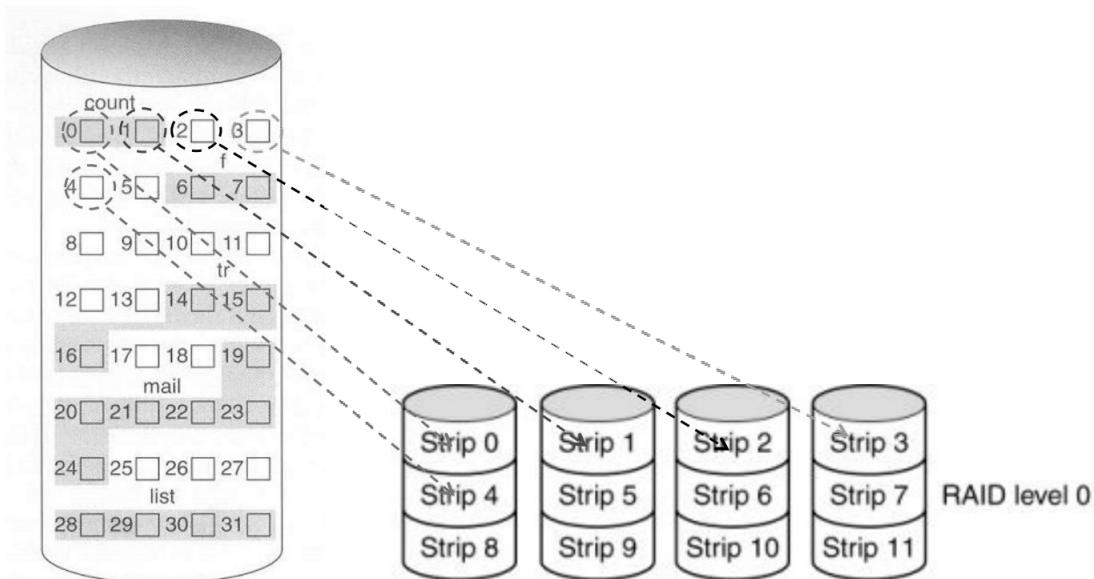
$$\text{numero - strip MOD dischi - nel - sistema}$$

Questa tecnica è nota come Striping:



I Sistemi RAID offrono comunque il livello 0, nel caso in cui si vogliano solamente massimizzare le prestazioni e la capacità del sistema senza aumentarne l'affidabilità.

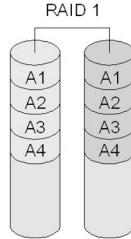
Notare che se $k = 1$, ogni Strip contiene un blocco. Il blocco 0 è mappato sul primo settore del primo disco, il blocco 1 è mappato sul primo settore del secondo disco, e così via:



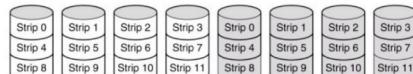
Tuttavia, come mai le prestazioni di un Sistema RAID livello 0 sono migliori di un normale disco che memorizza le stesse informazioni? Questo perché nel primo il controller RAID suddivide la richiesta in 4 letture eseguire in parallelo sui 4 dischi del sistema. Al contrario, su un singolo disco, tutti i settori dei quattro Strip dovranno essere letti in sequenza. Un RAID di livello 0 è quindi tanto più efficiente quanto più è alto il numero di dischi su cui sono suddivisi gli Strip.

Inoltre, l'affidabilità di un sistema RAID di livello 0 è inferiore di quella di un semplice disco SLED perchè il RAID è formato da più dischi e la probabilità che uno si rompa è maggiore e il Mean Time To Failure (MTTF) non può che essere inferiore. Il Livello 0 viene usato in quelle applicazioni che necessitano di alte prestazioni ma senza particolari problemi di affidabilità (audio e video streaming).

RAID Livello 1 (Mirroring) Nel RAID Livello 1, per ogni disco di dati D viene usato un secondo disco di Mirroring che contiene una copia esatta del disco D:

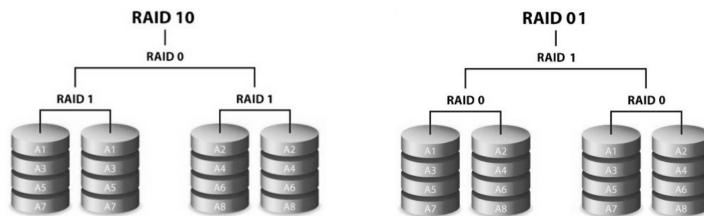


RAID Livello 01 (Striping + Mirroring) Il RAID di Livello 01 usa insieme Striping e Mirroring. Qui, tutti i dati vengono suddivisi in Strip distribuiti su più dischi e ciascun disco viene duplicato. Quando un disco si rompe, il sistema di controllo del RAID si rivolge al disco di mirror per l'accesso ai dati. Nel frattempo, il disco rotto può essere sostituito.



Si tratta della soluzione più costosa ma è questa la soluzione che viene usata per memorizzare dati finanziari e bancari.

RAID Livello 10 (Mirroring + Striping) Si tratta di una variante del precedente, in cui Mirroring e Striping vengono fatti in modo inverso. Le prestazioni del RAID 01 e 10 sono simili, ma quest'ultimo offre alcuni vantaggi in caso di guasti:



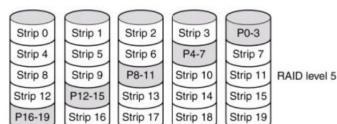
Risalire a un dato perso attraverso la Stringa di Parità Faccio una parentesi per poi capire i livelli successivi. Se ho m stringhe binarie ciascuna di n bit e uno dei dischi si rompe, posso recuperare la stringa persa se avevo salvato la cosiddetta Partità bit a bit delle m stringhe così:

$$\text{parità} = \text{stringa} - 0 \text{ } EX - OR \text{ stringa} - 1 \text{ } EX - OR \dots EX - OR \text{ stringa} - m$$

RAID Livello 4 (Striping con Parità) Il RAID Livello 4 usa lo Striping a livello di blocchi, e calcola uno Strip di parità per poter implementare una eventuale operazione di Recovery.

Questo tipo di RAID risparmia dischi rispetto al RAID 01 e garantisce lo stesso il mantenimento dei dati in caso di guasto di un disco, ma con minore efficienza. Infatti, se lo Strip di un disco viene modificato, vanno letti anche i corrispondenti Strip di tutti gli altri dischi per ricalcolare la parità.

RAID Livello 5 (Striping con Parità Distribuita) Il RAID Livello 5 funziona come il 4, ma riduce il carico sul disco di parità nelle operazioni di scrittura, distribuendo gli strip di parità fra tutti i dischi:



RAID Livello 6 (Striping + Doppia Parità Distribuita) Questo livello è in grado di resistere anche al guasto di due dischi contemporaneamente. Tuttavia, questa soluzione è poco usata poichè la rottura contemporanea di due dischi è un evento estremamente raro.

6.3 Memorie a Stato Solido

Le Memoria a Stato Solido sono memorie permanenti riscrivibili basate sulla tecnologia delle memorie flash. Sono memorie che hanno un limite di circa 100.000 riscritture per pagina (che è più che sufficiente per qualsiasi applicazione ragionevole). Prima di essere riscritta, la pagina deve essere cancellata (appunto con un processo di "flashing").

Un SSD può complementare l'uso di un Hard Disk in due modi:

- Livello di Memoria Cache permanente collocata tra l'HD e la memoria RAM.
- Come un secondo Hard Disk più piccolo ma più veloce.

Dato che l'SSD è un dispositivo ad accesso diretto non necessita di una politica di Scheduling particolare, e quindi si usa il FCFS.

Capitolo 7

Il File System

Il File System fornisce i meccanismi per la memorizzazione e l'accesso ai dati. Esso consiste in due parti:

- Insieme di file.
- Una struttura di Directory, che organizza i file del sistema.

7.1 I File

Un File è un'unità logica di informazione memorizzata permanentemente su un supporto di memoria secondaria e dotato di:

- Nome.
- Posizione logica all'interno del File System.
- Alcuni attributi (dimensione, diritto di accesso ecc.).

I File poi contengono dati di diversa tipologia:

- Dati: possono essere numerici, caratteri, binari.
- Programmi: possono essere sorgenti, eseguibili, linkabili.
- Documenti: possono essere multimediali, omogenei.

Il Sistema Operativo e/o gli applicativi che fanno uso dei file possono riconoscere una loro struttura interna.

7.1.1 Attributi del File

A ciascun File sono poi associati degli Attributi che, tra le altre cose, facilitano l'uso del file e le operazioni che si possono eseguire su di esso:

- Nome simbolico: è l'unica informazione mantenuta in una forma adeguata per gli utenti umani.
- Tipo: necessaria per quei Sistemi Operativi che supportano diversi tipi di file.
- Posizione fisica: indica dove si trova il file all'interno della memoria secondaria.
- Posizione logica: si tratta del pathname del file.
- Dimensione: indica appunto la dimensione corrente del file.
- Permessi di accesso e protezione: informazioni di controllo all'accesso che permette al SO di proteggere il file da abusi.
- Data e ora: può essere la data di creazione, ultima modifica o ultimo accesso.
- Identificazione del proprietario: specifica l'utente proprietario del file.

7.1.2 Operazioni sui File

Le operazioni che possono essere fatte su di un file sono le seguenti:

- Creazione: richiede al Sistema Operativo di trovare lo spazio necessario e poi di creare un accesso al file tramite la directory che lo contiene.
- Lettura e scrittura di un file: il Sistema Operativo deve gestire il puntatore in scrittura e lettura e deve occuparsi di gestire lo spazio sull'HD per ospitare l'eventuale espansione del file.
- Riposizionamento all'interno di un file: per riposizionare nel punto desiderato e per lettere o scrivere partendo da lì.

- Rimozione del file: recupera lo spazio che era prima contenuto dal file sul supporto di memoria.
- Troncamento del file: cancella i dati memorizzati e recupera lo spazio occupato, ma mantiene tutti gli altri attributi del file.

Un file può essere acceduto in due sole modalità:

- Accesso sequenziale: i dati del file vengono letti o modificati in modo sequenziale.
- Accesso diretto: si desidera leggere o modificare un dato posizionato in un punto ben preciso del file.

7.2 Le Directory

Un File System può essere molto grande e occorre quindi un'organizzazione che permetta di accedere a tutti questi dati in tempi ragionevole.

Per farlo, utilizziamo appunto un sistema di Directory che contiene dei file e consente di risalire ad esse.

Le informazioni che possono essere recuperate sono:

- Posizione.
- Dimensioni correnti.
- Data dell'ultimo accesso.
- Data dell'ultima modifica.
- ID del proprietario.
- Protezioni e permessi di accesso.

Le Directory possono assumere una struttura ad albero. Per ovvie ragioni, quella dalla quale si parte prende il nome di Root (radice). Ciascuno utente di un computer ha a disposizione una porzione del File System che può modellare a piacere: la porzione di File System che si dirama a partire dalla cosiddetta Home Directory.

Si deve quindi distinguere tra Pathname assoluto, ovvero definito a partire dalla Root, oppure il Pathname relativo, perchè è definito rispetto alla current directory.

Tuttavia, l'accesso ai file attraverso il Pathname è estremamente inefficiente, perchè può richiedere accessi alla Memoria Secondaria, su cui i file e le directory sono memorizzati. Per evitarlo, il Sistema Operativo chiede ai programmi di aprire i file che vogliono usare e successivamente queste informazioni vengono copiate in Memoria Principale in un Open File Table. Quando un programma chiude, le informazioni del file nella tabella possono essere rimosse.

7.2.1 Protezione

Il proprietario di un file deve poter controllare che cosa si può fare nel suo file e chi lo può fare. Essenzialmente si possono fare operazioni di:

- Read
- Write
- Execute
- Append
- Delete
- List properties

Inoltre, esistono solamente tre classi di utenti:

- Il possessore del file.
- Il gruppo a cui appartiene il possessore del file.
- Tutti gli altri utenti del sistema.

E infine, esistono solamente tre tipologie di protezione:

- Lettura
- Scrittura
- Esecuzione

7.3 Realizzazione del File System

Esistono fondamentalmente tre tipologie di allocazione sull'HDD dei file, e saranno quelle spiegate in questa sezione.

Allocazione Contigua Ogni file è allocato in un insieme di blocchi dell'HDD contigui. Per recuperare le informazioni dei file basterà conoscere la posizione del primo blocco e il numero di blocchi contigui occupati dal file.

Il vantaggio di questa tecnica è la velocità e la semplicità, lo svantaggio è che presenta molti problemi visto che necessita di molti blocchi adiacenti, oltre al fatto che c'è il rischio di frammentazione esterna.

Allocazione Concatenata In questa tecnica costruisco una catena di blocchi contenenti i dati dei file, e ogni blocco conterrà un puntatore a quello successivo. Nell'ultimo blocco viene scritto un numero negativo.

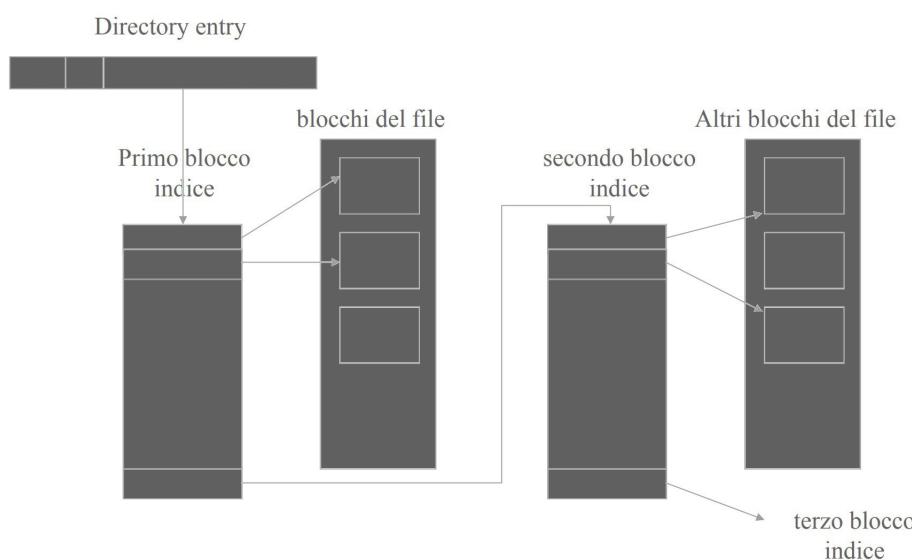
Il vantaggio è che non si crea frammentazione esterna, ma al contempo si genera uno spreco visto che una parte del blocco è impiegata a memorizzare l'indirizzo del blocco successivo.

Allocazione Indicizzata In questa tecnica tengo traccia di tutti i blocchi in cui è contenuto un file scrivendo il loro numero in un altro blocco del disco chiamato Blocco Indice. Per recuperare i dati basta memorizzare, tra gli attributi del file, il numero del suo blocco indice.

Il vantaggio è l'assenza di blocchi contigui e quindi nessuna frammentazione esterna, ma allo stesso tempo un intero blocco indice deve sempre essere utilizzato per memorizzare il numero dei blocchi di dati del file. Se il file è piccolo si genera quindi uno spreco.

Inoltre, che succede se un blocco indice non è sufficiente per memorizzare i numeri di tutti i blocchi dati del file? Ci sono due soluzioni possibili:

- Schema concatenato: L'ultima entry del blocco indice punta a un secondo blocco indice.
- Schema a più livelli: Il blocco indice contiene solo puntatori ad altri blocchi indice.



Una variante dell'Allocazione Indicizzata è usata da Unix: qui a ogni file è associato un i-node (Index Node) che contiene gli attributi del file e l'elenco dei blocchi di dati del file. Questi vengono gestiti direttamente dal Sistema Operativo e i primi blocchi dell'HD vengono appunto usati per contenere questi nodi.

Per tenere traccia di tutti i blocchi di dati del file, ogni i-node contiene lo spazio per memorizzare:

- 10 puntatori diretti a blocchi di dati del file.
- Un puntatore Single Indirect che punta a un blocco indice che conterrà puntatori a blocchi di dati file.
- Un puntatore Double Indirect che punta a un blocco indice che conterrà puntatori a blocchi indice ciascuno dei quali contiene puntatori a blocchi di dati del file.
- Un puntatore Triple Indirect che punta a un blocco indice che conterrà puntatori a blocchi indice ciascuno dei quali conterrà puntatori a blocchi indice ciascuno dei quali contiene puntatori a blocchi di dati del file.

NTFS Un approccio molto diverso è usato nel File System di Windows, chiamato NFTS da New Technology File System.

Ogni file è descritto da un elemento con una dimensione fissa e numerati consecutivamente. Tutti gli elementi di un File System sono contenuti in una Master File Table (MFT): un file memorizzato solo nei primi blocchi del disco e gestito esclusivamente dal Sistema Operativo. Il numero dell'elemento è associato al nome del file ed è detto File Reference.

7.3.1 Gestione dello Spazio Libero

Il Sistema Operativo deve anche tenere traccia di tutti i blocchi liberi del disco. In Unix queste informazioni sono memorizzate nel Superblocco, mentre su Windows nella MFT.

Per creare o estendere un file, si cercano blocchi liberi e si allocano i file, mentre quando viene cancellato i suoi blocchi sono reinseriti nell'elenco dei blocchi liberi.

7.4 Link Unix

In Unix, un file è identificato univocamente dall'Index-Node che contiene tutte le informazioni relative al file: i suoi attributi e in quali blocchi sono memorizzati i suoi dati. Il numero dell'Index-Node rappresenta univocamente un file (esattamente come il PID), ma poichè è scomodo da ricordare, al numero dell'I-Node è associato il nome del file.

mydir:

21	prog.c
128	old_name
11	letter

Hard Link (link fisici) Quindi per concludere, un Hard Link è un collegamento vero e proprio, la sua eliminazione comporta l'eliminazione del file collegato.

Symbolic Link Si tratta di un collegamento semplice, la cui eliminazione non modifica in alcun modo il file.