

Appunti di

Sistemi Operativi

Rosario Terranova

v 1.0.1

Sommario

Introduzione	3
Cos'è un Sistema Operativo	3
SO come macchina estesa.....	4
SO come gestore di risorse	4
Uno sguardo all'hardware.....	5
Eccezioni.....	8
Moltitudine di SO	10
Struttura di un SO	10
Gestione dei processi.....	14
Cos'è un processo	14
Nascita e morte di un processo	15
Gerarchie dei processi.....	16
Stati dei processi	16
Tabella dei processi.....	17
Thread	18
Comunicazione fra processi	21
Mutua esclusione.....	22
Tecniche di mutua esclusione.....	24
Semafori	26
Monitor	28
Scambio di messaggi tra processi	29
Problemi noti di sincronizzazione dei processi	31
Scheduling	40
Algoritmi di schedulazione.....	42
Scheduling in Windows, Unix e Linux	47
Gestione della memoria.....	49
Compiti del gestore della memoria.....	49
Gestione dei vari tipi di programmazione di sistema	50
Gestione dell'allocazione	51
Memoria virtuale	56
Paginazione	57
Tabelle delle pagine	58
Memoria associativa (TLB)	62
Algoritmi di rimpiazzamento della pagine	64
Allocazione dei frame.....	68
Working set	69
Paginazione su richiesta.....	70

Dimensione della pagina.....	71
Segmentazione.....	73
Gestione della memoria su Linux.....	77
Gestione dei File.....	85
Cos'è un file system	85
Directory	87
Struttura dei file system.....	88
Prestazioni del file system	94
Condivisione di file su un file system	94
File system virtuale (FSV)	95
Gestione blocchi liberi	96
Quote su disco	96
Controlli di consistenza.....	97
File system e log.....	98
Cache del disco e dei file.....	99
Deframmentazione	99
File system storici e contemporanei	100
Scheduling del disco.....	103
Algoritmi di scheduling su disco.....	104
Sistemi RAID	105
Memorie Flash e File System	109
Comandi della Shell Unix.....	110
Struttura di un SO Unix	110
Tipi di comandi.....	111
Comandi di spostamento	112
Comandi di stampa di messaggi a video	113
Comandi di conteggio e ordinamento	114
Comandi per i permessi e proprietà	114
Modalità di utilizzo degli utenti	115
Comandi di compressione file.....	115
Alias per i comandi.....	116
Link del file system.....	117
Modalità di esecuzione	118
Controllo dei jobs	119
Variabili	119
Comandi di ricerca di un file	120
Programmazione in ambiente UNIX	122

Introduzione

Cos'è un Sistema Operativo

Un calcolatore moderno è costituito da uno o più processori, da memoria primaria e secondaria, dischi, ecc. Quindi è un sistema complesso. Scrivere programmi che tengano conto di tutte queste componenti e che le usino correttamente, è estremamente difficile. Per questo motivo, i calcolatori vengono dotati di uno strato software detto **Sistema Operativo** (SO), il cui compito è gestire tutti questi dispositivi e fornire ai programmi utente un'interfaccia semplificata con l'hardware. Ad esempio la gestione dei *registri di dispositivo* (registri che vengono utilizzati per effettuare una scrittura o lettura da una periferica) è molto complicata e quindi contro produttiva da lasciare all'utente o al programmatore che dovrebbe preoccuparsi più di problemi hardware che software.

Obiettivi di un SO

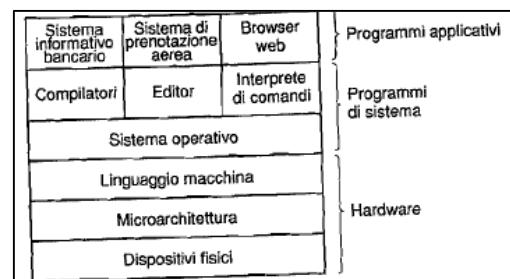
- **Convenienza** per l'utente che vuole qualità nei servizi e maggiore velocità nel rispondere alle richieste;
- Uso **efficiente** delle risorse hardware a disposizione, in modo da avere buone prestazioni nell'esecuzione dei programmi; si verifica scarsa efficienza se un programma non usa le risorse assegnatigli, occorre quindi minimizzare lo spreco di risorse e l'**overhead** (richieste in sovrappiù rispetto a quelle strettamente necessarie per ottenere un determinato scopo);
- Garanzia che altri utenti non siano in grado di interferire con le proprie attività, ovvero la **protezione** contro i malintenzionati.

Funzioni di un SO

- Gestione dei programmi e della CPU con lo **scheduling** (funzione per gestire l'esecuzione dei programmi), che decide di volta in volta a quale programma deve essere concesso l'utilizzo della CPU, sottraendolo allo scadere di un tempo ad un processo e dandolo ad un altro attraverso la prelazione;
- Gestione e allocazione delle risorse delle risorse quando un programma le chiede consultando una tabella delle risorse;
- Sicurezza e protezione.

Posizione del SO rispetto agli altri componenti del computer

Nell'architettura di un calcolatore il SO viene posto al di sopra degli strati dei dispositivi fisici, della microarchitettura e del linguaggio macchina, ma al di sotto dei programmi di sistema e programmi applicativi che non fanno parte del SO stesso ma ne fanno uso.



Portabilità ed espandibilità di un SO

Durante il ciclo di vita di un SO, si possono verificare diversi cambiamenti nei computer e negli ambienti di elaborazione. Per questi motivi, dovrebbe essere semplice implementare il SO su un nuovo computer e successivamente aggiungervi nuove funzionalità. Si parla pertanto rispettivamente di portabilità ed espandibilità.

- La **portabilità** di un SO si riferisce alla facilità con cui il SO può essere implementato su un computer che ha una differente architettura.
- L'**espandibilità** di un SO si riferisce alla facilità con cui le sue funzionalità possono essere migliorate per adattarle a un nuovo ambiente di elaborazione.

I moderni SO sono implementati nella forma di nucleo, detto kernel o microkernel, e costruiscono il resto del SO usando i servizi offerti dal nucleo. Questa struttura fa sì che la portabilità di un SO sia determinata dalle proprietà del suo kernel (o microkernel), mentre l'espandibilità di un SO sia determinata dalla natura dei servizi offerti dal kernel (o microkernel).

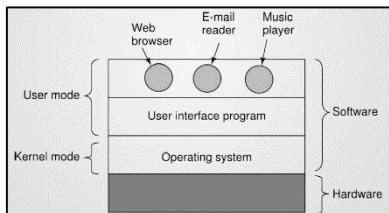
Kernel

Il kernel è l'insieme di routine (funzioni) che costituiscono il cuore del SO. Esso controlla il funzionamento del computer implementando le operazioni note come **funzioni di controllo** (inizializzazione dei programmi e allocazione delle risorse). Inoltre fornisce servizi all'utente. Il kernel risiede in memoria durante il funzionamento del SO ed esegue le istruzioni utilizzando la CPU per implementare le funzioni di controllo ed i servizi. In questo modo la CPU è usata sia dai programmi utente che dal kernel.

Modalità duale di accesso alle risorse

Nei SO che per la protezione fanno uso di meccanismi basati su stati gerarchici di privilegio, vengono chiamati:

- **Kernel mode:** lo stato di privilegio massimo riservato all'esecuzione del kernel. Il codice macchina eseguito in tale modalità ha accesso illimitato alla memoria, all'HW e alle altre risorse;
- **User mode:** uno stato di privilegio caratterizzato da un numero relativamente basso di privilegi verso la memoria, l'HW e altre risorse. In un tale sistema le applicazioni utente solitamente girano in user mode, e passano ad uno stato di maggiore privilegio a seconda delle operazioni che devono eseguire. Ogni processo dispone di una propria memoria virtuale e quindi di un proprio spazio di indirizzamento, che può eventualmente avere sezioni condivise con lo spazio di indirizzamento di altri processi.

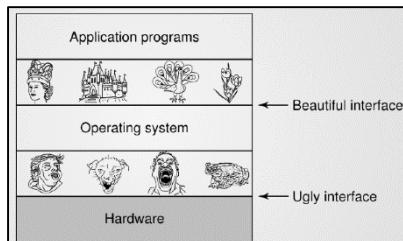


Il SO è (normalmente) quella porzione di SW che viene eseguito in modalità kernel ed ha accesso illimitato all'HW. I compilatori e gli editor vengono eseguiti in modalità utente. Il SW utente non accede direttamente alle risorse HW poiché viene eseguito dalla CPU in user mode che impedisce l'accesso diretto alle risorse nonché manomissioni al SO.

User e kernel mode sono quindi stati dell'hardware e non del software. Il SW utente che necessita dell'HW richiede servizi al SO tramite l'istruzione **trap**, in risposta la CPU passa in kernel mode e fornisce il servizio al SW utente, anche se in modo controllato. Il codice di servizio del SO prima di tornare ad eseguire il SW utente ripristina lo stato della CPU in user mode tramite l'istruzione **return from trap**.

Dare una definizione rigorosa di SO risulta ostico poiché realizza due funzionalità scorrelate: estende la macchina e gestisce le risorse.

SO come macchina estesa



Il SO nasconde al programmatore la verità sull'hardware e gli presenta una semplice serie di file con nome che possono essere letti e scritti senza dover pensare alle modalità fisiche di esecuzione dei comandi. In questo senso la funzionalità del SO è quella di presentare all'utente una macchina estesa o una macchina virtuale che sia più facile da programmare dell'HW sottostante. Il SO fornisce a questo scopo una varietà di servizi di cui i programmi possono usufruire attraverso istruzioni speciali dette chiamate di sistema (*system call*).

SO come gestore di risorse

Da un punto di vista alternativo, il compito del SO è quello di gestire un'allocazione (assegnamento) ordinata di varie risorse ai vari programmi che competono tra loro per usarle. Poniamo come esempio il caso in cui tre diversi programmi vogliono stampare: il SO può risolvere una simile situazione di potenziale caos **bufferizzando** (cioè salvando temporaneamente) i dati destinati alla stampante sul disco.

Condivisione rispetto allo tempo e al spazio

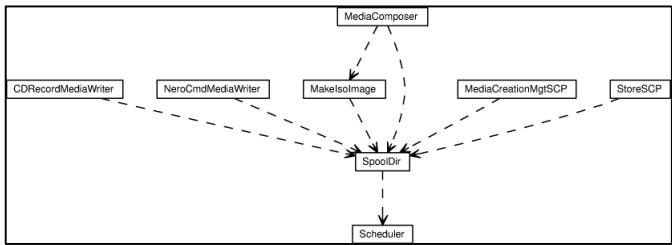
La gestione delle risorse comporta la loro condivisione sotto due aspetti: rispetto al tempo e rispetto allo spazio.

- Quando una risorsa è condivisa rispetto al **tempo**, programmi o utenti diversi fanno a turno per usarla: prima uno di questi usa la risorsa, poi un altro e così via.
- Quando la condivisione della risorsa avviene attraverso lo **spazio**, invece di alternarsi ad ogni programma viene assegnata parte della risorsa. Per esempio la memoria principale è ripartita tra diversi programmi in esecuzione.

Sovraffollamento di richieste di risorse

Conflitti e condivisioni scaturiscono dal **multiprogramming**, cioè più programmi che risiedono in memoria. Il multiprogramming consente, ma non implica, il **multitasking** il quale permette a più programmi di essere eseguiti in parallelo. Un programma in corso di esecuzione è detto **processo** ed è caratterizzato dal codice (istruzioni) e stato (contatore di programma e contenuto della sua memoria). I SO moderni sono multiprogramming, multitasking e multiutente.

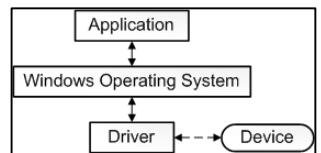
Altro problema scaturisce dall'uso esclusivo di una risorsa, che potrebbe creare un effetto collo di bottiglia (affollamento) se fosse concessa direttamente ai processi che resterebbero in attesa di esecuzione. La soluzione a



spooling è molto utile quando i dispositivi accedono ai dati a una velocità variabile. Il buffer rappresenta una stazione di attesa dove i dati possono rimanere fino a che il dispositivo più lento non riesce a gestire i dati in attesa. Un processo sempre attivo (**deamon**) preleva uno alla volta i job dalla spool e li avvia alla risorsa.

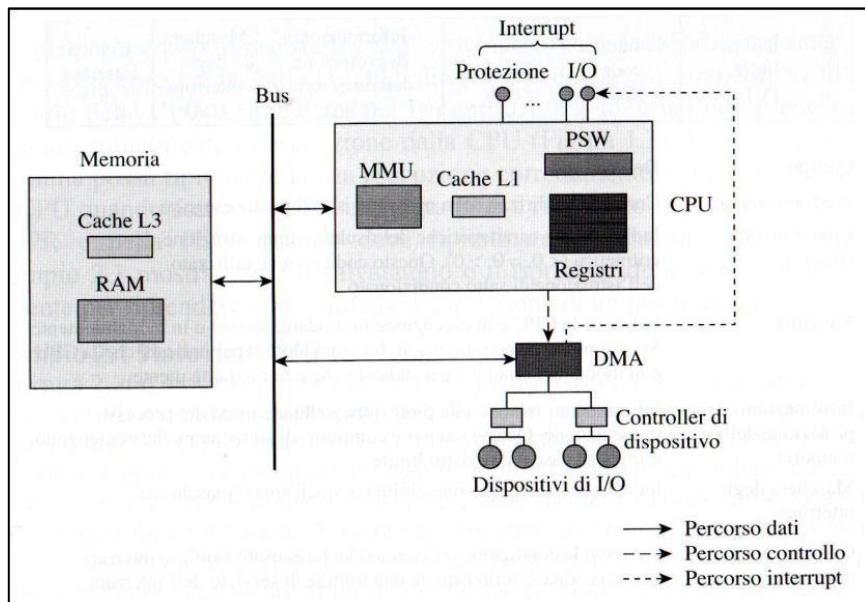
Il SO nel caso dei device (periferiche di I/O) non accede direttamente all'unità fisica ma all'**interfaccia** che mette a disposizione il **controller** associato al device, che ha il compito di ricevere un input da SO e trasformarlo in un input comprensibile al device. Per poter comunicare ed interagire con il controller il SO ha bisogno di SW aggiuntivo chiamato driver. Il driver del device deve essere fornito dal produttore e può trovarsi o fuori il SO (cioè fuori dal kernel, soluzione rara per la difficoltà di pilotare i registri del controller da fuori) oppure dentro il kernel nel quale può entrare in tre modi diversi:

- **Linking statico**: reboot del sistema dopo l'installazione della nuova periferica
- **Inclusione al boot**: in obbedienza a direttiva in file di configurazione (ad esempio in dos config.sys)
- **Installazione on-the-fly a run-time**: ad esempio l'installazione di periferica usb



Uno sguardo all'hardware

La figura sotto mostra lo schema di un computer indicando le unità rilevanti dal punto di vista di un SO. La CPU e la memoria sono direttamente connesse al bus di sistema, mentre i dispositivi di I/O sono connessi al bus attraverso un controller e il DMA.



Central Processing Unit (CPU)

L'unità di elaborazione centrale (CPU) è una tipologia di processore digitale **general purpose** (dispositivi elettronici che non siano dedicati ad un solo possibile utilizzo) la quale coordina in maniera centralizzata tutte le altre unità di elaborazione presenti nelle architetture hardware dei computer di elaborazione delle varie periferiche interne o schede elettroniche (scheda audio, scheda video, scheda di rete).

Il compito della CPU è quello di eseguire le istruzioni di un programma presente in memoria primaria (detta anche centrale, la RAM) dopo averlo prelevato dalla memoria secondaria o di massa, dalla ROM, o da altri dispositivi. Durante l'esecuzione del programma la CPU legge o scrive dati in memoria primaria. Il risultato dell'esecuzione dipende dal dato su cui si opera e dallo stato interno in cui la CPU stessa si trova, e può mantenere traccia delle istruzioni eseguite e dei dati letti (cache).

Nei sistemi attuali con più unità elaborative, sia per eseguire le istruzioni che per eseguire specifiche funzioni autonome, il termine "centrale" ha perso di significato. Non esiste più un organo centrale che gestisca il sistema. Il termine CPU è stato perciò sostituito dal termine *Processor* (processore). Ad esempio si ha il termine *microprocessor* (e non microCPU), *multiprocessor* e *multicore* (e non multiCPU) che sono in antitesi con il concetto di "central processor". Il termine CPU è attualmente obsoleto e non più usato (o raramente usato solo per retaggio storico).

Sono due le caratteristiche della CPU visibili ai programmi utente o al SO:

- **Registri GPR** (noti anche come general-purpose registers o come registri visibili dagli utenti) sono usati per memorizzare i dati, gli indirizzi, gli indici e lo **stack pointer** (SP, puntatore all'indirizzo in cima alla pila della memoria di elaborazione dei processi) durante l'esecuzione di un programma;
- **Registri di controllo PSW** che contengono l'informazione necessaria a controllare il funzionamento della CPU. Per semplicità l'insieme dei registri di controllo verrà chiamato **program status word (PSW)** e i singoli registri verranno individuati come campi della PSW.

Program Status Word (PSW)	
Program Counter (PC)	Indirizzo della prossima istruzione da eseguire
Condition Code (CC, flag)	Caratteristiche del risultato di un'istruzione aritmetica (<0 , $=0$, >0)
Modalità (M)	0 = kernel mode, 1 = user mode
Informazioni sulla Protezione della Memoria (MPI)	Informazioni sulla protezione della memoria del processo in esecuzione
Maschera degli Interrupt (IM)	Indica quali interrupt sono abilitati e quali sono mascherati
Codice Interrupt (CI)	Describe l'evento che ha causato l'ultimo interrupt

I tipi di processori che possiamo trovare in un elaboratore moderno sono:

- **CPU Multithreading (o hyperthreading):** Il multithreading indica il supporto HW da parte di un processore di eseguire più **thread** (processi eseguiti contemporaneamente). Questa tecnica si distingue da quella alla base dei sistemi multiprocessore per il fatto che i singoli thread condividono lo stesso spazio d'indirizzamento, la stessa cache e lo stesso translation lookaside buffer. Il multithreading migliora le prestazioni dei programmi solamente quando questi sono stati sviluppati suddividendo il carico di lavoro su più thread che possono essere eseguiti in apparenza in parallelo. Mentre i sistemi multiprocessore sono dotati di più unità di calcolo indipendenti per le quali l'esecuzione è effettivamente parallela, un sistema multithread invece è dotato di una singola unità di calcolo che si cerca di utilizzare al meglio eseguendo più thread nella stessa unità di calcolo. Le due tecniche sono complementari: a volte i sistemi multiprocessore implementano anche il multithreading per migliorare le prestazioni complessive del sistema
- **CPU Multiprocessore:** Sistema di elaborazione (computer, workstation, server) equipaggiato con 2 o più processori operanti in parallelo in cui le elaborazioni di un processore vengono replicate e controllate da un processore gemello, per garantire l'integrità e l'esattezza dei dati. Vantaggi: throughput, economia di scala, affidabilità, resistenza ai guasti (NoStop)
- **CPU Multicore:** Una CPU composta da 2 o più core, ovvero da più nuclei di processori "fisici" montati sullo stesso package. Il termine Multi core è ovviamente generico e, sebbene adatto a descrivere CPU a 2 soli core o più core, può essere affiancato anche con altri termini specifici della soluzione adottata per la CPU, quali dual core, quad core, octa core o l'ultimo nato hexa core (1, 2, 4, 8 o 16 core). Questo tipo di architettura rispetto alla single core consente di aumentare la potenza di calcolo di una CPU senza aumentare la frequenza di clock di lavoro, a tutto vantaggio del calore dissipato (che diminuisce rispetto al caso di più processori separati) così come l'energia assorbita.
- **GPU (unità di elaborazione grafica o scheda video):** Tipologia particolare di coprocessore (ausiliaria ad un altro processore) che si contraddistingue per essere specializzata nel rendering di immagini grafiche. La GPU è tipicamente implementata come microprocessore monolitico e, da alcuni anni, viene anche implementata assieme alla CPU nel medesimo circuito integrato. Da alcuni anni vengono anche prodotti processori multicore formati da più core di GPU.
- **FPU (unità di calcolo in virgola mobile):** Un tipo di processore che si contraddistingue per essere specializzato nell'esecuzione di calcoli matematici in virgola mobile (es. $0,07824 \times 10^5$). La maggior parte delle operazioni di calcolo svolte dalla FPU sono semplice aritmetica (come l'addizione o la moltiplicazione) ma alcune FPU sono in grado di svolgere anche calcoli esponenziali o trigonometrici (come l'estrazione di radice o il calcolo del seno). Attualmente la FPU è tipicamente implementata come microprocessore monolitico e spesso, quando è un coprocessore matematico della CPU, viene integrata assieme alla CPU nel medesimo circuito integrato.

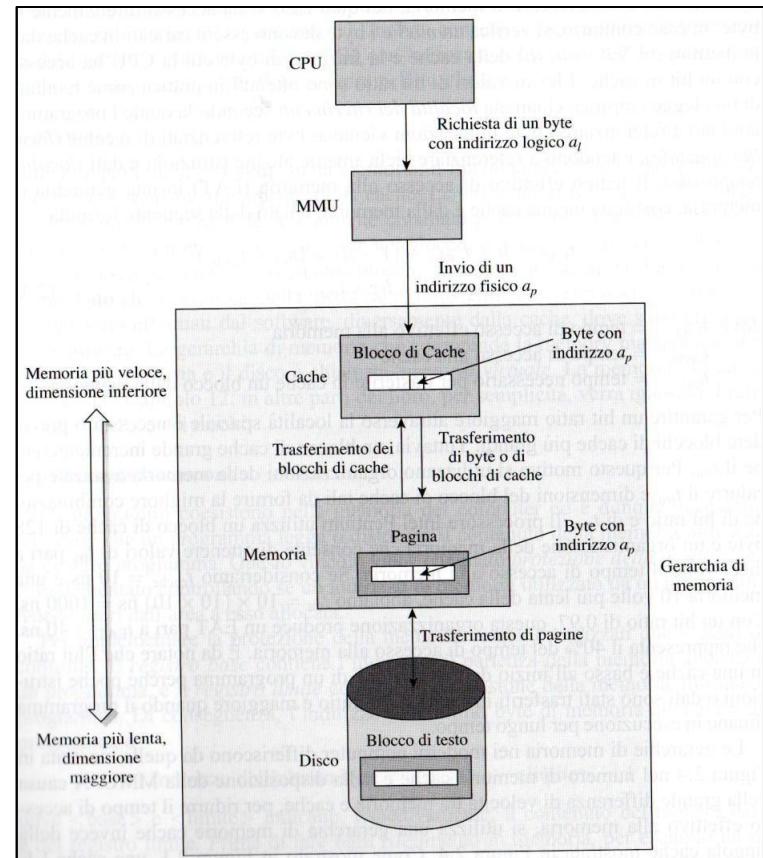
Memory Management Unit (MMU)

Questa unità si occupa di effettuare la traduzione degli indirizzi:

- Un **indirizzo logico** è l'indirizzo usato dalla CPU per fare riferimento ad un dato o ad un'istruzione
- Un **indirizzo fisico** è l'indirizzo in memoria dove risiede il dato o l'istruzione richiesta dalla CPU

La MMU si occupa proprio di "tradurre" un indirizzo logico in un indirizzo fisico. Inoltre, il SO implementa la **memoria virtuale** usando l'allocazione della memoria non contigua e la MMU. La tecnica della memoria virtuale consiste nel creare l'illusione di una memoria più grande della reale memoria installata.

Un computer dovrebbe idealmente contenere una memoria abbastanza capiente e abbastanza veloce, così che gli accessi alla memoria non rallentino la CPU. Tuttavia la memoria veloce è costosa. La soluzione consiste in una gerarchia della memoria che contenga un numero di unità di memoria con differenti velocità. La memoria più veloce è quella di dimensioni più piccole, mentre la memoria più lenta è quella di dimensioni maggiori. La CPU accede solo alla memoria più veloce: se il dato (o l'istruzione) di cui necessita è presente nella memoria più veloce allora viene usato direttamente, altrimenti il dato richiesto viene copiato dalla memoria più lenta in quella più veloce e successivamente utilizzato. Il dato rimane nella memoria più veloce finché non viene rimosso per fare spazio ad altri dati. Questa organizzazione ha, quindi, lo scopo di velocizzare gli accessi ai dati utilizzati di più frequente. Un esempio di **memoria gerarchica** è quello costituito da:



- **Memoria cache**, piccola e veloce, contiene alcune istruzioni e valori di dati cui la CPU ha avuto accesso più di recente. L'hardware della memoria non trasferisce un singolo byte dalla memoria alla cache, ma carica sempre un blocco di memoria di dimensioni standard in un'area della cache chiamata *cache block* o *cache line*. In questo modo, l'accesso a un byte vicino a un byte caricato di recente, può essere effettuato senza accedere nuovamente alla memoria. Per ogni dato o istruzione richiesti durante l'esecuzione di un programma, la CPU effettua una ricerca nella cache. Si verifica un *hit* se i byte richiesti sono presenti nella cache, e in questo caso si ha accesso diretto ai byte; si verifica un *miss* se i byte richiesti non sono presenti, e in questo caso devono essere caricati nella cache dalla memoria. A causa della grande differenza di velocità tra cache e memoria, per ridurre il tempo di accesso effettivo alla memoria, si utilizza una gerarchia di memorie cache invece della singola cache mostrata in figura. Esistono, infatti, cache di vari livelli che consentono di migliorare il tempo effettivo di accesso alla memoria: una cache L1 (cache di livello 1) è montata sul chip della CPU; a questa è solitamente affiancata una cache L2 (cache di livello 2), più lenta ma più capiente della cache L1; normalmente è presente anche una cache L3 ancora più capiente e lenta.
- **Memoria RAM**, più capiente ma più lenta della cache, il funzionamento della memoria centrale è analogo a quello della memoria cache. Le similitudini riguardano il trasferimento di un blocco di byte, solitamente chiamato pagina, dal disco rigido alla memoria RAM. La differenza sta nel fatto che la gestione della memoria e il trasferimento dei blocchi tra memoria e disco sono effettuati dal software, mentre dalla cache sono effettuati dall'hardware.
- **Hard disk**, è la memoria più lenta ma anche quella più capiente.

Periferiche di Input/Output

Una delle operazioni più lente che deve eseguire un sistema è il trasferimento di I/O, cioè il trasferire i dati da o verso una periferica di I/O. Quest'operazione risulta, solitamente, un'operazione molto lenta, che richiede l'intervento della CPU, della memoria (che bufferizza le richieste) e di una periferica di I/O. Visto che il sistema di I/O è il più lento di un computer, la CPU può eseguire milioni di istruzioni nella quantità di tempo richiesta per effettuare un'operazione di I/O; per tale motivo si preferisce utilizzare tra i vari modi esistenti per effettuare le operazioni di I/O quello che non prevede l'intervento della CPU.

Per eseguire operazioni di input/output (I/O) tra la CPU ed un dispositivo di I/O esistono le seguenti modalità:

- **Busy waiting (attesa impegnata):** Tecnica di sincronizzazione per cui un processo o un thread che debba attendere il verificarsi di una certa condizione (per esempio la disponibilità di input dalla tastiera o di un messaggio proveniente da un altro processo) lo faccia verificando ripetutamente (ciclicamente) tale condizione. Nell'ingegneria del software si tende a evitare l'impiego del busy waiting laddove possibile; questa tecnica, infatti, presenta lo svantaggio di impegnare la CPU, in quanto ogni iterazione del ciclo di busy wait comporta l'esecuzione delle istruzioni che costituiscono la verifica della condizione attesa. La tecnica viene invece impiegata frequentemente nella progettazione dell'hardware. La CPU quindi non può eseguire nessun'altra istruzione mentre è in esecuzione un'operazione di I/O con questa tecnica.
- **Interrupt:** Si adotta la tecnica della sospensione del processo e del suo successivo risveglio tramite un segnale specifico. Un interrupt viene generato quando un byte di dati deve essere trasferito dalla periferica di I/O alla memoria, e la CPU esegue la routine di servizio dell'interrupt che gestisce il trasferimento del byte.
- **DMA (Direct Memory Access, "accesso diretto alla memoria"):** Il trasferimento di dati tra la periferica di I/O e la memoria avviene direttamente sul bus, la CPU non è coinvolta. Tale meccanismo permette ad altri sottosistemi, quali a esempio le periferiche, di accedere direttamente alla memoria primaria per scambiare dati, in lettura e/o scrittura, senza coinvolgere la CPU per ogni byte trasferito tramite l'usuale meccanismo dell'interrupt e la successiva richiesta dell'operazione desiderata, ma generando un singolo interrupt per blocco trasferito.

Le operazioni del DMA sono effettuate dal **controller DMA**, ovvero un processore dedicato all'esecuzione delle operazioni di I/O. Diversi dispositivi di I/O della stessa classe sono collegati ad un controller. I vari controller sono poi collegati al DMA. Quando viene eseguita un'operazione di I/O, la CPU "delega" il DMA in modo tale da non essere coinvolta nell'operazione ed essere dunque libera di svolgere altre funzioni. È il DMA, mediante il controller, ad effettuare l'operazione. Al termine dell'operazione, il DMA genera un interrupt di I/O.

La CPU passa all'esecuzione del kernel quando rileva un interrupt; il kernel analizza la causa dell'interrupt e deduce che l'operazione di I/O è stata completata. Inizialmente il controller DMA viene inizializzato specificando la periferica da cui prelevare i dati, l'indirizzo fisico X della memoria in cui memorizzare questi ultimi ed il numero C dei byte complessivi da trasferire. A questo punto il controller della periferica inizia il trasferimento, inviando ogni singolo byte al controller DMA. Quest'ultimo acquisisce il controllo del bus della memoria memorizzando il byte all'indirizzo X; in seguito incrementa l'indirizzo per il prossimo byte e decrementa il contatore C. Quando quest'ultimo raggiunge il valore 0, il trasferimento si è concluso e viene inviato un interrupt alla CPU per segnalare l'evento.

Eccezioni

Durante il normale flusso di lavoro di un SO possono capitare degli eventi che alterano la normale esecuzione di un programma e richiedono l'attenzione del SO. Un'eccezione è il risultato di una qualunque causa che impedisca al processore di eseguire un'istruzione. Le eccezioni si possono dividere nei seguenti tipi:

- **Trap:** accesso non consentito in memoria, divisione per 0, disconnessione dalla rete, errore di trasferimento dati a stampante.
- **Interrupt:** richiesta spesso asincrona e di provenienza esterna al microprocessore che forza il SO a interrompere il programma in esecuzione.
- **System call:** chiamate di un programma ad una funzione del SO, ad esempio per eseguire una operazione su un file o su un processo.

Le eccezioni possono essere quindi inattese, previste o addirittura volute dal programmatore. Al verificarsi di un'eccezione, il processore sospende l'esecuzione del programma corrente e passa ad una **routine predefinita** del SO. Tale routine può avere compiti speciali (ad esempio, un interrupt è spesso usato per ricevere dati dalle periferiche), o può anche terminare il programma nel caso non possa fare nulla per risolvere il problema, come nel caso classico della divisione per zero.

Interrupt

Scopo dell'interrupt è quello di segnalare al SO il verificarsi dell'evento a cui è associato in modo da consentirgli di effettuare le azioni appropriate per gestirlo.

Durante la normale elaborazione, l'istruzione il cui indirizzo è contenuto nel **Program Counter** (registro “puntatore” della CPU la cui funzione è quella di conservare l'indirizzo di memoria della prossima istruzione da eseguire) viene recuperata-decodificata-eseguita (**fetch-decode-execute**); dopo l'esecuzione (**execute**) dell'istruzione si controlla se si è verificato un interrupt durante l'esecuzione dell'istruzione. In caso affermativo esegue un'azione di interrupt che salva lo stato della CPU (cioè il contenuto del PSW e dei GPR), e comincia l'esecuzione in modalità kernel di una routine di servizio dell'interrupt (chiamata anche ISR), caricando i nuovi dati di tale routine nel PSW e nei GPR. A un certo punto, il kernel può ripristinare l'esecuzione del programma interrotto ricaricando lo stato salvato della CPU.

Ad ogni interrupt è associata anche una **priorità**. Se diversi interrupt si verificano nello stesso tempo, la CPU seleziona l'interrupt con priorità più alta, mentre gli altri interrupt restano pendenti finché non verranno selezionati ed elaborati. I tipi di interrupt sono:

- **I/O interrupt**: segnalano completamento o malfunzionamento di una periferica I/O
- **Timer interrupt**: generato dopo uno specifico intervallo di tempo
- **Program interrupt**: causato da errori aritmetici come overflow, violazioni di memoria o volontariamente dagli interrupt software.

Chiamate di sistema

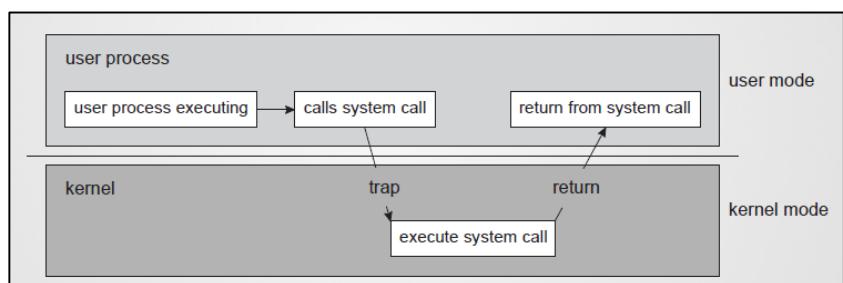
Una chiamata di sistema (in inglese system call) è il meccanismo, usato da un processo a livello utente o livello applicativo, per richiedere un servizio a livello kernel dal SO del computer in uso. Essa, di solito, è disponibile come funzione in quei linguaggi di programmazione che supportano la programmazione di sistema (es. il linguaggio C), oppure come particolari istruzioni assembler. Fondamentale è il passaggio dall'user mode al Kernel mode attraverso una particolare istruzione che si identifica nel trap.

L'interfaccia tra i SO e i programmi utente è definita dall'insieme di chiamate di sistema fornite dal SO. Ai programmi utente viene fornita una procedura di libreria per rendere possibile l'esecuzione di chiamate di sistema.

Ogni calcolatore con una singola CPU può eseguire al massimo una sola istruzione per volta. Se un processo che sta eseguendo un programma utente in user mode ha bisogno di un servizio di sistema (come leggere dati da un file) deve eseguire una trap o un'istruzione che esegua una syscall per trasferire il controllo al SO, il quale riesce a capire cosa vuole il processo chiamante esaminando i parametri, dopodiché esegue la syscall e restituisce il controllo all'istruzione seguente alla syscall. In un certo senso, le syscall entrano a far parte del kernel.

I passi seguiti da un syscall sono:

1. Il programma utente esegue una trap
2. Trap porta la CPU in kernel mode e trasferisce il controllo al SO
3. Il SO determina l'indice k della procedura di servizio Pk richiesta
4. Il SO trova l'indirizzo di Pk nella **dispatch table** e la invoca
5. Il SO torna in modalità utente con una return from trap.



Da quello che abbiamo appena discusso, nel codice di alto livello i servizi di SO vengono invocati attraverso funzioni raccolte in **library** fornite a corredo del SO. Il codice delle librerie si occupa tra l'altro di interagire direttamente con il SO invocandone i servizi. Per questo non usa, in genere, normali chiamate di procedura ma trap e di conseguenza è scritto almeno in parte in assembler.

Moltitudine di SO

Abbiamo diversi tipi di SO per diversi sistemi di elaborazione:

- **Mainframe:** I mainframe sono calcolatori della dimensione di una stanza e si distinguono dai personal computer per la loro capacità di ingresso e di uscita. I SO per mainframe sono fortemente orientati all'elaborazione di molti job per volta, la maggior parte dei quali necessita di una quantità enorme di ingressi/uscite. Esistono tre tipi di servizi: elaborazione batch (elabora senza alcuna interazione con l'utente), di transazioni (manipolano grosse quantità di piccole richieste) e di condivisione rispetto al tempo (che permettono a molti utenti remoti di eseguire contemporaneamente i loro job).
- **Server:** un server può essere un pc molto grande o persino mainframe. I So di tali server permettono di condividere risorse HW e SW.
- **Multiprocessore:** un modo per aumentare la potenza di calcolo è unire più CPU in un unico sistema il quale necessita di So particolari che spesso sono varianti di So per i server con caratteristiche particolari per la comunicazione e la connessione.
- **Per i personal computer:** il compito dei So per i pc è quello di fornire un interfaccia all'utente singolo e che soddisfi le sue necessità.
- **Real time:** i So per i sistemi real time sono caratterizzati dall'avere il tempo come parametro chiave. Se un'operazione deve essere eseguita in un certo momento abbiamo un sistema real time stretto (hard). Se invece un'operazione può essere eseguita in un lasso di tempo comunque piccolo abbiamo un sistema real time lasco (soft).
- **Embedded:** vengono eseguiti su calcolatori che controllano dispositivi che non sono generalmente nati come calcolatori (tv, forni).

Struttura di un SO

Esistono diverse strutture di SO e tra queste esamineremo solo alcune:

Sistemi monolitici

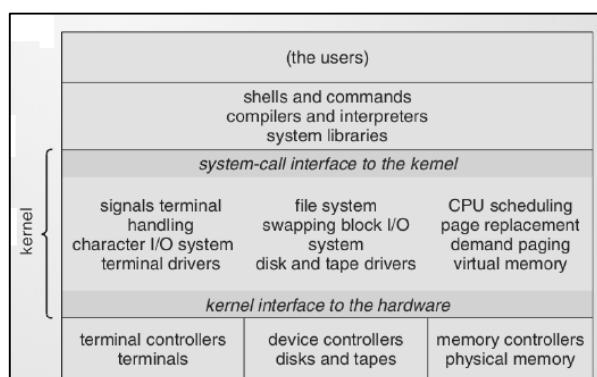
I primi SO avevano una struttura monolitica, secondo cui il SO formava un singolo strato software tra l'utente e la macchina (hardware). L'interfaccia utente consisteva in un interprete dei comandi. Sia l'interprete dei comandi che i processi degli utenti richiamavano le funzioni e i servizi del SO attraverso le chiamate di sistema.

Il SO è scritto come un insieme di procedure, ciascuna delle quali può chiamare un qualunque delle altre. Prima si compilano tutte le singole procedure, o i file che contengono le procedure, che in seguito vengono legate tutte insieme in un unico file oggetto tramite un linker di sistema. Ogni procedura è visibile da ogni altra procedura.

Questo tipo di sistemi aveva un portabilità molto limitata poiché il codice dipendente dall'architettura era presente in gran parte nel SO. Inoltre, nella struttura monolitica tutte le componenti del SO erano in grado di interagire con l'hardware, e questo rendeva complicate e dispendiose (anche in termini economici) le fasi di test e debug a causa del gap semantico, ossia l'assenza di corrispondenza tra la natura delle operazioni necessarie all'applicazione e la natura delle operazioni fornite dall'hardware.

Anche se poi arrivo il supporto HW a tali sistemi e la kernel/user mode, vi erano tanti problemi come un unico kernel con tutto dentro, ogni componente poteva richiamarne un altro, la poca gestibilità nel tempo. Questi problemi portarono alla ricerca di modi alternativi di strutturare un SO.

Esempi di SO con questa architettura: MS-DOS, UNIX.

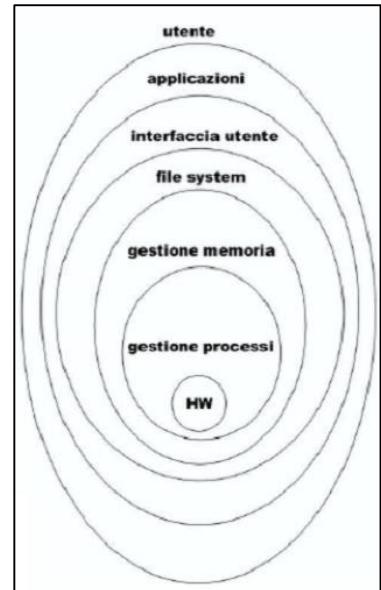


Sistema a livelli (o strati)

Il SO è scritto come una gerarchia di livelli ciascuno dei quali costruito su quello sottostante. Il sistema software è organizzato/suddiviso in layer L_1, \dots, L_n . Il livello L_k può usare il codice dei livelli L_{k-1}, \dots, L_0 ma non idem L_n, \dots, L_{k+1} . Di conseguenza L_h fornisce servizi / una macchina virtuale per L_n, \dots, L_{h+1} . Ogni livello quindi implementa delle funzionalità impiegando quelle fornite da quello inferiore.

La progettazione a livelli dei sistemi operativi utilizzava il principio dell'astrazione per controllare la complessità della progettazione del SO. Questa progettazione vede il SO come una gerarchia di livelli, in cui ogni livello forniva un insieme di servizi al livello superiore ed esso stesso usava i servizi messi a disposizione dal livello inferiore. Ciò voleva dire che nessun livello poteva essere "saltato".

Un'organizzazione del genere semplifica notevolmente le fasi di test, di debug e di modifica di un modulo; è più semplice da sviluppare e controllare (incapsulamento tipo OOP).



L'organizzazione a layer può essere una disciplina che il progettista del SO si dà, ma può essere aggirata (es. dalle applicazioni), oppure imposta dai meccanismi di accesso ai servizi di un layer.

Abbiamo comunque vari problemi di prestazioni dovuti alle chiamate nidificate e al relativo overhead.

Sistemi basati su kernel

Le motivazioni storiche di una struttura del SO basato su kernel risiedono nella portabilità del SO e nella semplicità di progettazione e codifica delle routine non kernel. La portabilità si ottiene inserendo nel kernel le parti del codice del SO dipendenti dall'architettura, mantenendo al di fuori del kernel le parti di codice indipendenti dall'architettura.

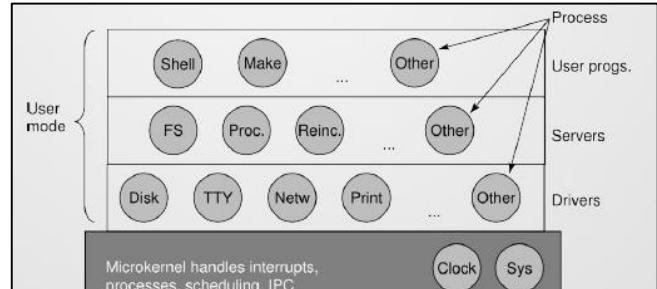
I SO basati su kernel presentano una ridotta espandibilità poiché l'aggiunta di nuove funzionalità può richiedere cambiamenti nelle funzioni e nei servizi offerti dal kernel.

Sistemi basati su microkernel

Abbiamo detto che mettere tutto il codice del SO dipendente all'architettura nel kernel fornisce una buona portabilità. Tuttavia, in pratica, anche i kernel contengono del codice indipendente dall'architettura per funzionare. Questo fa sì che la dimensione del kernel sia elevata allontanando l'obiettivo della portabilità. Inoltre, per incorporare nuove funzionalità spesso è necessario effettuare modifiche al kernel, e ciò porta a poca espandibilità.

Il microkernel fu sviluppato negli anni '90 per superare i problemi relativi alla portabilità, all'espandibilità e all'affidabilità del kernel. Un microkernel è il nucleo essenziale del codice di un SO; è di dimensione ridotta, contiene pochi meccanismi, supporta un piccolo numero di system call e non contiene nessuna politica.

I moduli contenenti le politiche sono implementati come processi server, ovvero semplici processi che non terminano mai; possono essere cambiati o sostituiti senza coinvolgere il microkernel, fornendo in tal modo elevata espandibilità al SO. I processi server e i programmi utente operano al di sopra del microkernel.



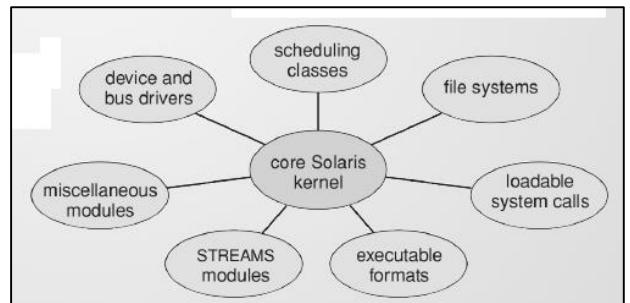
La comunicazione tra moduli avviene attraverso messaggi; si ha un miglior design (componenti piccoli) e migliore stabilità.

Esempi di SO basati su microkernel sono MINIX 3, Match, QNX, Mac OS (Darwin) e Windows NT.

Sistemi con struttura a moduli

La struttura dei SO basati su kernel si è evoluta per compensare alcuni dei suoi svantaggi. Gli elementi fondamentali di tale evoluzione sono i moduli del kernel caricabili dinamicamente e i driver dei dispositivi a livello utente.

In pratica, un kernel base viene caricato in memoria durante la fase di boot, mentre gli altri moduli sono caricati quando le loro funzionalità sono richieste e sono rimossi dalla memoria quando non sono più necessari. In questo modo viene preservata la memoria perché vengono caricati solo i moduli che servono effettivamente. Inoltre anche l'espandibilità viene migliorata in quanto è possibile aggiungere nuove funzioni al SO modificando i moduli già presenti o, meglio ancora, aggiungendone di nuovi.



È un design tipico della programmazione ad oggetti. L'efficienza è data dalla possibilità di ogni modulo di invocare qualunque altro modulo direttamente (quindi non attraverso messaggi ma tramite invocazioni).

Esempi di SO con tale struttura sono Solaris, Linux e Mac OS (ibrido).

Sistemi a macchine virtuali

Questi sistemi furono adottati perché diverse classi di utenti hanno la necessità di differenti tipologie di servizi, dunque, utilizzare un unico SO su di un computer può provocare una scarsa soddisfazione da parte di diversi utenti. Un SO si dice "macchina virtuale" in quanto costituisce una replica virtuale della macchina HW, e non una macchina immaginaria con funzionalità estese rispetto all'HW.

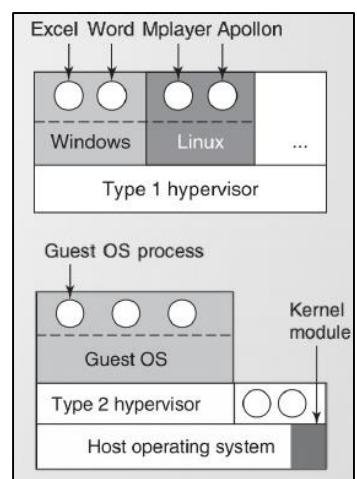
I sistemi operativi basati su macchina virtuale (SO VM) supportavano il funzionamento di diversi SO su un computer simultaneamente, creando una virtual machine per ogni utente e permettendo all'utente di eseguire i suoi programmi sul SO di sua scelta nella virtual machine. Chiameremo ognuno di questi sistemi operativi SO ospite, e chiameremo il SO della macchina virtuale host. Il SO VM realizza il funzionamento concorrente dei SO ospite attraverso un'azione simili alla commutazione dei processi, quindi con una procedura analoga allo scheduling. Quando una virtual machine veniva schedulata, il suo SO organizzava l'esecuzione delle applicazioni degli utenti in esso attive. Ogni processo dispone di una sua replica virtuale della CPU (perfetta, semmai solo più lenta) invece la memoria e I/O non sono replicati virtualmente. Tutti i processi "vedono" la stessa locazione e lo stesso registro del controller del disco.

La distinzione tra modalità kernel e modalità utente della CPU comporta alcune difficoltà nell'uso di un SO VM. Quest'ultimo deve infatti proteggersi dai SO ospite, per cui li deve eseguire con la CPU in modalità utente. In questo modo sia il SO ospite che i programmi utente al suo interno vengono eseguiti in modalità utente, cosa che rende vulnerabile il SO ospite a operazioni non legittime da parte di un processo utente.

Il processo di virtualizzazione viene gestito dall'**Hypervisor**, conosciuto anche come *virtual machine monitor*. Esso è il componente centrale e più importante di un SO VM è un software che deve operare in maniera trasparente senza pesare con la propria attività sul funzionamento e sulle prestazioni dei SO. Svolge attività di controllo al di sopra di ogni sistema, permettendone lo sfruttamento anche come monitor e debugger delle attività dei SO e delle applicazioni in modo da scoprire eventuali malfunzionamenti ed intervenire celermente. L'hypervisor può controllare ed interrompere eventuali attività pericolose, può allocare le risorse dinamicamente quando e dove necessario riducendo in modo drastico il tempo necessario alla messa in opera di nuovi sistemi, isolare l'architettura nel suo complesso da problemi a livello di SO ed applicativo, abilitare ad una gestione più semplice di risorse eterogenee e facilitare collaudo e debugging di ambienti controllati. Esistono due tipi di hypervisor:

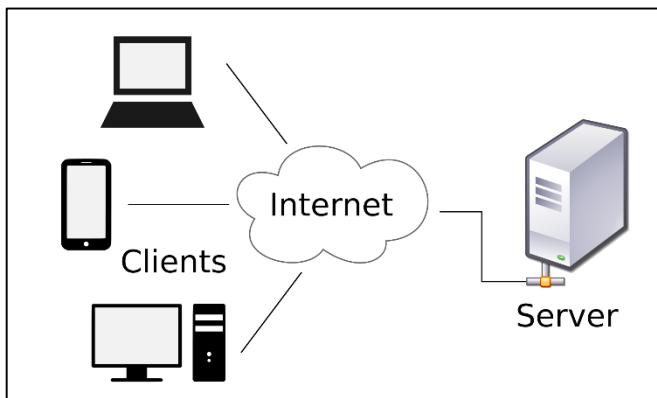
- **Hypervisor di tipo 1:** gira direttamente sull'hardware (es: VMware ESX/ESXi, Microsoft Hyper-V hypervisor);
- **Hypervisor di tipo 2:** è un processo in un SO Host (es: VMware Workstation, VirtualBox)

Discorso differente può essere fatto per la java virtual machine, indipendente dall'hypervisor, è il componente della piattaforma Java che esegue i programmi tradotti in bytecode (linguaggio intermedio tra linguaggio macchina e linguaggio di programmazione) dopo una prima compilazione.



Sistemi basati su client/server

Una tendenza dei moderni SO è quella di sviluppare ulteriormente l'idea di spostare il codice verso livelli superiori, e rimuoverne il più possibile dal SO, lasciando un minimo microkernel. L'approccio comune implementa la maggior parte delle funzioni del SO attraverso processi utente, e per richiedere un servizio, come ad esempio la lettura di un blocco di un file, un processo utente detto il processo client spedisce la richiesta ad un processo server, che poi esegue il servizio e restituisce la risposta. In questo modello il kernel si occupa solo della gestione e della comunicazione tra client e server; dividendo il SO, ciascuna delle quali gestisce solo un aspetto del sistema, come ad esempio la gestione del file, quella dei processi del terminale, o della memoria, ogni parte diventa piccola e maneggevole.



macchina remota. Per ciò che riguarda il client, nei due casi accade sempre la stessa cosa: viene spedita una richiesta, e viene ricevuta una risposta.

Il fatto che un kernel gestisce solo il trasferimento dei messaggi dai client ai server e viceversa, non è completamente realistica: alcune funzioni del SO (come il caricamento dei comandi nei registri dei dispositivi fisici di ingresso / uscita) sono difficili, se non impossibili, da realizzare attraverso programmi che girano nello spazio utente. Ci sono due modi per affrontare il problema:

- Uno è quello di permettere che alcuni processi server critici (come i driver dei dispositivi di ingresso/uscita) vengano eseguiti in modalità kernel, con un accesso completo a tutto l'hardware, ma comunichino ancora con gli altri processi usando il normale meccanismo a messaggi.
- L'altro modo è quello di costruire una quantità minima di meccanismi nel kernel, ma lasciare le politiche di decisione ai processi server che vengono eseguiti nello spazio utente. Ad esempio, il kernel potrebbe riconoscere che un messaggio spedito a un certo indirizzo speciale significa caricare il contenuto del messaggio stesso nei registri di ingresso / uscita di un certo disco, per iniziare un'operazione di lettura. In questo caso, il kernel non controlla nemmeno i byte contenuti nel messaggio, per vedere se sono validi o significativi, ma si limita semplicemente a copiarli alla cieca nei registri di dispositivo del disco. (Ovviamente, bisogna usare un qualche meccanismo per limitare l'uso di questi messaggi solo ai processi autorizzati.) La divisione fra meccanismi e politiche è un concetto importante, che ritornerà più volte nei sistemi operativi in contesti diversi.

Il tempo di overhead è un parametro fondamentale per lo studio delle prestazioni di un SO. Esso rappresenta il tempo medio di CPU necessario per eseguire i moduli del kernel. È spesso fornito in percentuale rispetto al tempo totale di utilizzo della CPU. Si può calcolare con:

$$\text{Overhead \%} = \frac{\text{Tempo CPU per l'esecuzione dei moduli del kernel}}{\text{Tempo totale di utilizzoCPU}}$$

Ovviamente, più il tempo è basso, maggiore sarà la quantità di tempo CPU che si può utilizzare per i processi utente.

Nei sistemi a livelli l'inserimento di più strati implica un sostanziale aumento dell'Overhead, diminuendo l'efficienza del sistema stesso

Inoltre, poiché tutti i servizi vengono eseguiti come processi utente, e non in modalità kernel, non si ha accesso diretto all'hardware, quindi, se ci fosse un baco nella gestione dei file, questo servizio potrebbe bloccarsi, senza causare il blocco dell'intera macchina.

Un altro vantaggio del modello client-server è la sua adattabilità ad essere utilizzato nei sistemi distribuiti: se un client comunica con un server inviandogli un messaggio, non ha bisogno di sapere se il messaggio è gestito localmente sulla sua macchina, o se invece è stato inviato attraverso una rete verso un server su una

Gestione dei processi

Cos'è un processo

Un processo è un programma in esecuzione che utilizza le risorse a esso allocate; occorre però fare una distinzione:

- **Programma:** insieme di istruzioni che specificano un attività; è formato da più processi che interagiscono tra loro per il raggiungimento di un obiettivo comune.
- **Processo:** l'esecuzione dell'attività; è costituito da due parti: una parte *statica*, ovvero il codice; una parte *dinamica*, ovvero le risorse che utilizza. Il codice del programma è costante, mentre la parte dinamica varia nel tempo.

Un SO mantiene in esecuzione un gran numero di processi per ogni istante di tempo. È il kernel che alloca le risorse ai processi e li schedula per l'utilizzo della CPU.

Parti di un processo

Ad ogni processo vengono associati il suo **spazio di indirizzamento** (chiamato **immagine di memoria** o **core image**) e una lista di **locazioni di memoria**. Nello specifico, un processo comprende sei componenti:

- ID (identificativo univoco assegnato dal SO)
- Programma eseguibile (codice del programma)
- Dati (usati durante l'esecuzione del programma, inclusi i dati contenuti nei file)
- Puntatore a stack (contiene i parametri e gli indirizzi di ritorno delle funzioni chiamate durante l'esecuzione)
- Risorse (quelle allocate dal SO)
- Stato della CPU (composto dal contenuto del PSW e dei registri GPR della CPU)

Lo stato della CPU contiene una serie di informazioni molto importanti come la prossima istruzione da eseguire e il contenuto del campo **CC (condition code)**. Inoltre lo stato della CPU cambia man mano che l'esecuzione del programma progredisce.

Periodicamente il SO decide di sospendere un processo e di iniziare ad eseguirne un altro (**prelazione**), perché ad esempio il primo ha esaurito il tempo della CPU (se si tratta di ambiente time sharing).

Program counter (PC)

È un registro della CPU la cui funzione è quella di conservare l'indirizzo di memoria della prossima istruzione (in linguaggio macchina) da eseguire. È un registro puntatore cioè punta a un dato che si trova in memoria all'indirizzo corrispondente al valore contenuto nel registro stesso.

Il program counter è utilizzato nel ciclo fetch-execute che costituisce la dinamica fondamentale nel funzionamento di un computer; tale ciclo è una ripetizione infinita dei seguenti passi:

1. Caricamento dell'istruzione riferita dal program counter;
2. Aggiornamento (incremento) del program counter, in modo che contenga l'indirizzo dell'istruzione successiva;
3. Esecuzione dell'istruzione caricata.

Gestire i processi

Un SO mantiene in esecuzione un gran numero di processi per ogni istante di tempo. È il kernel che alloca le risorse ai processi e li schedula per l'utilizzo della CPU. Gestire i processi significa: crearli, soddisfare le richieste di risorse, schedularli per l'uso della CPU, sincronizzarli per controllare la loro interazione, evitare **deadlock** (situazione in cui due o più processi che interagiscono, a volte possono mettersi in una situazione di stallo dalla quale non possono uscire ad esempio nell'usare una risorsa hardware oppure quando uno aspetta l'altro per un qualche servizio) in modo che non siano in attesa l'un l'altro indefinitamente e terminarli quando hanno concluso l'esecuzione.

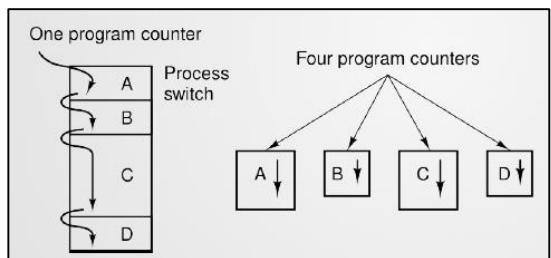
Thread

Anche un thread è un programma in esecuzione ma usa le risorse di un processo, quindi somiglia a un processo in tutti gli aspetti. È possibile che molti thread vengano eseguiti nell'ambito dello stesso processo. Quando un processo viene sospeso tutti i suoi puntatori vanno salvati.

Parallelismo

La CPU, ad ogni singolo istante, esegue un solo programma, nell'arco di un secondo può lavorare su programmi diversi, dando all'utente l'illusione del **parallelismo**. Talvolta, per indicare questo continuo alternarsi tra i vari programmi da parte della CPU si parla di **pseudo-parallelismo**, per distinguerlo dal vero parallelismo hardware dei sistemi multiprocessore (che hanno due o più CPU che condividono la stessa memoria fisica).

Modello a processi



Gli elaboratori moderni usano il modello a processi, dove tutto il SW che può essere eseguito su di un calcolatore, compreso talvolta il SO, è organizzato in un certo numero di processi sequenziali. Concettualmente, ogni processo dispone di una propria CPU virtuale; ovviamente, nella realtà la CPU passa in continuazione da processo all'altro. Questo rapido cambio di contesto, avanti e indietro, viene chiamato **multiprogrammazione**. Quindi ogni singolo processo avrà il proprio program counter virtuale che diventerà reale solo nel momento in cui il processo viene realmente eseguito dalla cpu. Per determinare quando la CPU deve smettere di lavorare per un processo vengono utilizzati **algoritmi di schedulazione**.

Nascita e morte di un processo

I SO hanno bisogno di diverse tecniche per creare e distruggere processi durante il corso delle operazioni.

Creazione dei processi

A provocare la creazione di un processo sono, principalmente, ci sono quattro eventi:

- Inizializzazione del sistema;
- Esecuzione di una syscall per la creazione di un processo, effettuata da un processo in esecuzione;
- Una richiesta, da parte dell'utente, affinché venga creato un nuovo processo;
- Inizio di un job batch (insieme di comandi o programmi, tipicamente non interattivi, aggregati per l'esecuzione, come in uno script o un comando batch; un file batch è un file di testo che contiene una sequenza di comandi per l'interprete di comandi del sistema).

Quando un SO viene lanciato, vengono generalmente creati diversi processi, alcuni dei quali sono processi che vengono eseguiti in primo piano (**foreground**), cioè interagiscono con l'utente ed eseguono il lavoro per loro, mentre altri sono processi che eseguono sullo sfondo (**background**) che non sono associati a particolari utenti ma hanno qualche funzione specifica. I processi che stanno in background per gestire qualche attività come le e-mail, le pagine web, la stampa e così via vengono chiamati demoni o *daemon* (è il kernel che sveglia i servizi dei demoni).

Oltre ai processi creati a tempo di avvio, in seguito se ne possono creare di nuovi. Spesso un processo in esecuzione originerà system call per creare uno o più nuovi processi per aiutarlo a compiere il suo lavoro.

In UNIX esiste una sola system call per creare un nuovo processo: **fork**, che crea una copia esatta del processo chiamante. Dopo la fork, il processo padre e il processo figlio hanno la stessa immagine di memoria, le stesse stringhe di ambiente, gli stessi file aperti. Normalmente il processo figlio esegue poi una **execve** (esegue un programma) o una **syscall** simile per cambiare la sua immagine di memoria ed eseguire un nuovo programma. La ragione di questo procedimento in due passi è di permettere al processo figlio di manipolare i propri descrittori di files, dopo le fork, ma prima della execve, per realizzare la redirezione dello standard input, dello standard output e dello standard error.

Il comando per creare un nuovo processo per un nuovo comando per Win32 è invece **CreateProcess**.

La terminazione dei processi

Il processo termina a causa di una delle seguenti cause:

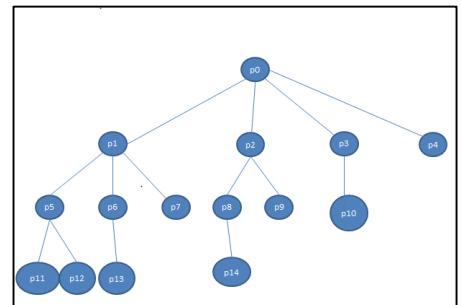
- Terminazione normale (volontaria): quando termina il suo lavoro. In UNIX è **exit**, in Win32 è **ExitProcess**;
- Terminazione con errore (volontaria): quando si tenta di compilare un file inesistente;
- Si verifica un errore fatale (involontaria): dovuto a errore di programmazione; alcuni sono gestibili, altri no;
- È stato ucciso (involontaria): quando un processo esegue una syscall dicendo al SO di uccidere qualche altro processo. In UNIX questa chiamata è **kill**, mentre la funzione Win32 corrispondente è **TerminateProcess**.

In genere la vita dei processi è molto breve, al massimo di qualche minuto. Quando il tempo specificato per un processo è terminato, il SO manda un segnale di allarme al processo, che causa una sospensione momentanea. Quando la procedura di trattamento del segnale è terminata, il processo sospeso viene fatto ripartire esattamente dal punto in cui è stato bloccato. I segnali sono il corrispondente software delle interruzioni hardware e possono essere generati da molte cause.

Gerarchie dei processi

Se un processo può creare uno o più altri processi, questi sono detti processi figli; l'insieme dei processi padri e dei loro processi figli, è detta **gerarchia dei processi** ed in genere non sono molto profonde (arrivano al massimo a tre livelli).

In UNIX, quando un utente invia un segnale dalla tastiera, questo è consegnato a tutti i membri del gruppo di processi (padre e tutti i suoi discendenti) correntemente associato alla tastiera. Individualmente, ogni processo, può catturare il segnale, ignorarlo o eseguire l'azione di default, cioè essere ucciso dal segnale.



Windows, invece, non ha un concetto di gerarchia di processi: tutti i processi sono uguali. Il solo caso in cui esiste qualcosa di simile a una gerarchia di processi si verifica quando un processo viene creato, il genitore riceve un gettone (*token*) speciale, chiamato **handle** o gestore, che può usare per controllare il figlio; d'altra parte il genitore può passare il token a qualche altro processo, invalidando così la gerarchia.

Scopo dei processi figli

Soltanamente, un processo crea uno o più figli in modo tale da delegare ad ognuno di essi una parte del suo lavoro; questa tecnica prende il nome di **multitasking** e presenta tre benefici:

1. **Speedup dell'elaborazione** (maggiore velocità di esecuzione): diminuzione del tempo di esecuzione dell'applicazione grazie alla creazione di processi figli. Se il processo primario non creasse processi figli, eseguirebbe le operazioni sequenzialmente; invece creando i processi figli, questi eseguono concorrentemente le operazioni.
2. **Priorità per le funzioni critiche**: molti SO consentono a un processo genitore di assegnare priorità ai processi figli. Un'applicazione real-time può assegnare una priorità alta a un processo figlio che deve eseguire una funzione critica in modo tale da soddisfare i suoi requisiti di risposta.
3. **Proteggere un processo genitore dagli errori**: il kernel può terminare un processo figlio in caso di errore, ma il padre resta protetto e può avviare un'azione di recupero.

Per facilitare l'uso dei processi figli, il kernel fornisce funzioni per creare un processo figlio e assegnargli una priorità, terminare un processo figlio e determinare lo stato di un processo figlio. Inoltre permette la condivisione, la comunicazione e la sincronizzazione dei processi figli. Solo il processo padre può controllare o accedere ai processi figli ma mai viceversa.

Stati dei processi

Sebbene il processo sia un'entità indipendente, con un proprio programma e un proprio stato interno, i processi spesso devono interagire tra di loro. I tre stati (+ due addizionali) in cui si può trovare un processo sono:

- In esecuzione (**running**): quando sta veramente usando la CPU in quell'istante;
- Pronto (**ready**): il processo richiede l'uso della CPU per continuare la sua esecuzione, tuttavia non è ancora stato eseguito il **dispatch**, ovvero non è stato impostato il controllo della CPU (o l'accesso alle risorse) per il processo schedulato in ready;
- Bloccato (**blocked**): non può ottenere la CPU anche se questa non ha niente da fare, poiché è in attesa di qualche evento esterno;
- Nuovo (**new**): processo appena creato, aspetta l'allocazione delle risorse per entrare nello stato di ready;
- Terminato (**terminated**): l'esecuzione del processo è stata completata correttamente o è stata terminata dal kernel.

Un computer che ha una sola CPU può avere un solo processo nello stato di running, ma più processi negli stati blocked e ready

Transizione tra gli stati

La transizione tra gli stati descritti avviene grazie al seguente diagramma a 4 stati:

NB: un processo può entrare negli stati ready, running e blocked anche più volte, mentre può entrare nello stato di terminazione una sola volta.

- (NEW) → READY: un nuovo processo entra nello stato ready dopo che le risorse richieste sono state allocate. In questo stato ci sono tutti i processi che si trovano in memoria centrale. NB: per andare in esecuzione, il processo deve essere caricato in memoria centrale.
- READY → RUNNING: un processo può andare nello stato running solo se in precedenza si trovava nello stato ready e ci va non appena viene completato il dispatching. In pratica un processo va in esecuzione non appena gli viene passato il controllo della CPU. Dallo stato running può raggiungere tutti gli altri stati.
- RUNNING → TERMINAZIONE: un processo passa dallo stato running a quello di terminazione quando viene portata a termine l'esecuzione del programma. Le cause che portano alla terminazione di un programma sono molteplici (es: auto-terminazione, terminazione richiesta dal processo padre, eccesso di utilizzo di una risorsa, condizioni anomale durante l'esecuzione, interazione non corretta con altri processi). Quando un processo è terminato, vengono liberate tutte le risorse che gli erano state assegnate.
- RUNNING → READY: un processo passa dallo stato running a quello ready quando viene prelazionato poiché il kernel decide di schedulare un altro processo (es: un processo a priorità più alta va nello stato ready oppure la time-slice del processo si esaurisce).
- RUNNING → BLOCKED: un processo passa dallo stato running a quello blocked quando effettua una system call per richiedere l'uso di una risorsa o quando rimane in attesa di un evento. Un processo bloccato si trova in memoria, ma non necessariamente in memoria centrale (ad esempio in memoria swap).
- BLOCKED → READY: quando la richiesta del processo viene soddisfatta o quando si verifica l'evento, il processo passa dallo stato blocked a quello ready.

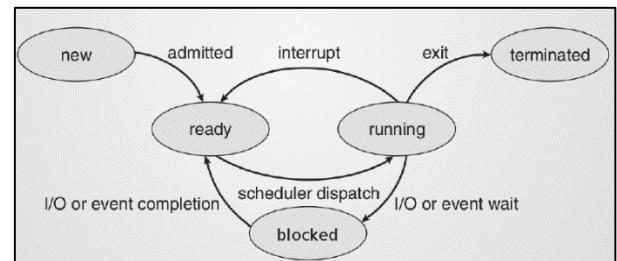
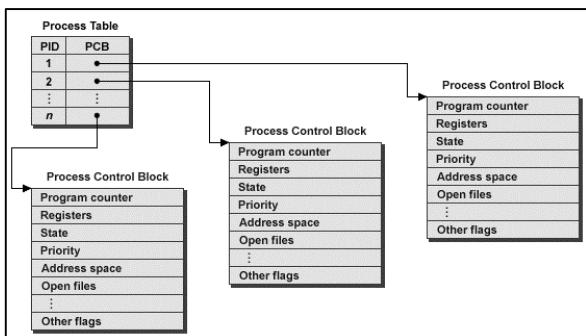


Tabella dei processi

In molti SO tutte le informazioni relative ai processi, tranne il contenuto del suo spazio di indirizzamento, sono memorizzate in una tabella di SO chiamata **tabella dei processi** o process table, che è una lista di strutture, una per ogni processo attualmente esistente. Quando un processo viene interrotto, una funzione si occupa del salvataggio del contesto, cioè salva lo stato del processo e della CPU relativo al processo in questa tabella.



Ogni elemento salvato in questa tabella è chiamato **Process Control Block (PCB)**, che contiene informazioni sullo stato del processo, il suo program counter, stack pointer, allocazione della memoria, stato dei suoi file aperti, informazioni necessarie per l'addebito dei costi e per la schedulazione e qualunque altra cosa debba essere salvata quando il processo passa da uno stato di esecuzione ad uno stato di pronto o bloccato, in maniera che possa essere fatto ripartire più tardi come se non fosse mai stato fermato.

Ad ogni classe di dispositivi I/O viene associata una locazione, spesso vicina alla parte bassa della memoria, chiamata **interrupt vector** che contiene l'indirizzo della procedura di gestione delle interruzioni.

Context switch

Per poter rimuovere dall'esecuzione un processo (ad esempio in seguito a prelazione), prima di metterne in esecuzione uno nuovo il SO deve salvare in memoria una serie di informazioni sullo stato corrente del processo, che saranno ripristinate quando poi esso verrà rimesso in esecuzione.

Questo permette a più processi di condividere una stessa CPU, ed è utile quindi sia nei sistemi con un solo processore, perché consente di eseguire più programmi contemporaneamente, sia nell'ambito del calcolo parallelo, perché consente un migliore bilanciamento del carico.

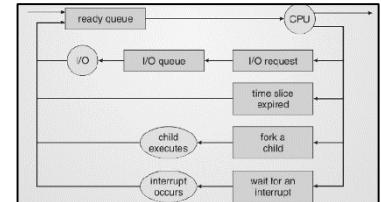
Gestione degli interrupt per il passaggio di processo	
#n passo	Azione
1	Salvataggio dei registri PC e del PSW nello stack attuale
2	Caricamento dal vettore degli interrupt dell'indirizzo della procedura associata
3	Salvataggio registri nel PCB e impostazione di un nuovo stack;
4	Interrogazione dello scheduler per sapere con quale processo proseguire
5	Ripristino dal PCB dello stato di tale processo (registri, mappa memoria)
6	Ripresa nel processo corrente

Supponiamo che sia in esecuzione il processo utente 3, quando si verifica un'interruzione dal disco: il program counter di questo processo, la parola di stato del programma e magari uno o più registri vengono messi sullo stack corrente dall'HW dedicato alle interruzioni e la CPU salta all'indirizzo specificato nell'interrupt vector. Questo è tutto quello che fa l'HW; da qui in poi è tutto in mano al SW. Azioni come il salvataggio dei registri o l'inizializzazione dello stack pointer, vengono svolte da una piccola routine in linguaggio assembler. Quando questa routine termina, chiama una procedura C per terminare il resto del lavoro per lo specifico tipo di interrupt.

Quando questa ha svolto il suo compito, viene richiamato lo scheduler per vedere qual è il prossimo processo da eseguire. Dopodiché il controllo viene passato al codice in linguaggio assembler perché vengano caricati i registri e la mappa di memoria per il nuovo processo corrente. Tutte queste azioni di salvataggio e ripristino vengono chiamate **context switching**.

Coda dei processi pronti e code dei dispositivi

Fanno parte dei processi di scheduling, e sono l'insieme dei processi in attesa di una risposta da una periferica I/O. Sono strutture collegate sui PCB.



Thread

La commutazione tra i processi all'interno di un'applicazione genera un elevato overhead dovuto alla quantità di informazione da salvare e ripristinare ad ogni commutazione. Per questo motivo i progettisti dei SO hanno sviluppato un modello alternativo di esecuzione, chiamato **thread**, che favorisce la concorrenza all'interno di un'applicazione con un ridotto overhead.

Un thread rappresenta un modello alternativo di esecuzione di un programma che usa le risorse di un processo, senza quindi richiedere altre risorse proprio per non generare overhead. Un processo può contenere più thread, ciascuno dei quali evolve in modo logicamente separato dagli altri thread.

Un processo crea un thread mediante una system call. Il thread non possiede risorse proprie, quindi non ha un contesto; viene eseguito usando il contesto del processo e in tal modo accede alle risorse del processo.

Modello a thread

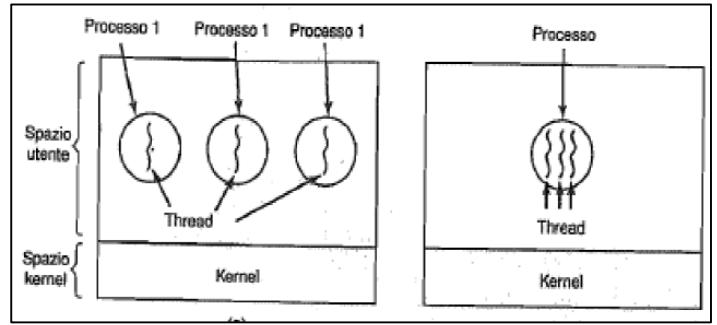
Il thread ha un program counter, dei registri che mantengono le variabili di lavoro correnti, ed ha uno stack che contiene lo storia dell'esecuzione. Sebbene un thread debba essere in esecuzione in qualche processo, il thread e il suo processo sono concetti diversi e possono essere trattati separatamente. I processi vengono usati per **raggruppare risorse**, mentre i thread sono entità schedulate per **l'esecuzione nella CPU**.

Quello che i thread aggiungono al modello a processi è il permettere molte esecuzioni nell'ambiente di un processo, in larga misura indipendenti una dall'altra. Avere più thread che vengono eseguiti in parallelo in un solo processo è analogo ad avere vari processi che sono in esecuzione in parallelo su un singolo calcolatore, solo che nel primo caso condividono uno spazio di indirizzamento, mentre nel secondo caso i processi condividono la memoria fisica.

Un thread è caratterizzato da: PC, registri, stack e stato; condivide tutto il resto, e non adotta una politica di protezione della memoria.

Multithreading

Poiché i thread hanno solo alcune delle proprietà dei processi, a volte vengono chiamati processi leggeri o **lightweight process**. Il termine **multithreading** è utilizzato per descrivere la situazione in cui ad un solo processo sono associati più thread. Se si hanno tre processi e in ognuno di questi si ha un singolo thread, allora i thread operano in spazi di indirizzamento diversi; mentre se si ha un singolo processo e questo contiene diversi thread, questi condividono lo stesso spazio di indirizzamento.

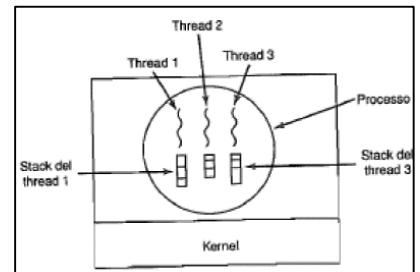


Quando un processo con thread multipli viene eseguito su un sistema con una sola CPU, i thread vengono eseguiti a turno; la CPU passa rapidamente avanti e indietro per i thread. Thread diversi in un processo non sono indipendenti come processi diversi, perché tutti i thread hanno esattamente lo stesso spazio di indirizzamento, cioè condividono anche le stesse variabili globali. Dal momento che ogni thread può accedere ad ogni indirizzo di memoria dello spazio di indirizzamento del processo, un thread può leggere, scrivere o persino cancellare completamente lo stack di un altro thread: **non c'è protezione** tra i thread perché è impossibile realizzarla e non dovrebbe essere necessaria. Oltre alla condivisione dello spazio di indirizzamento, tutti i thread **condividono** lo stesso insieme di files aperti, processi figli, allarmi eccetera. Quindi quando si hanno dei processi correlati fra loro, è meglio utilizzare un unico processo con diversi thread, invece se i processi non sono correlati è preferibile utilizzare diversi processi.

Quando è presente il **multithreading**, normalmente i processi partono con solo un thread presente, che ha la capacità di creare nuovi thread richiamando una procedura dalla libreria, come ad esempio *thread_create* e un parametro che tipicamente specifica il nome della procedura che il nuovo thread deve eseguire. Non è necessario né possibile specificare qualcosa di nuovo sullo spazio di indirizzamento del thread, dal momento che questo, automaticamente, è in esecuzione nello spazio di indirizzamento del thread che l'ha creato (lo spazio di indirizzamento è condiviso).

Forma dei thread

Come in un processo tradizionale (cioè con un solo thread), un thread può essere in uno dei qualsiasi di diversi stati: running, ready, blocked o terminated. È importante capire che ogni thread ha il proprio stack. Lo stack di ciascun thread contiene un elemento per ogni procedura chiamata ma non ancora conclusa, con le variabili locali alla procedura e gli indirizzi di ritorno da usare quando la chiamata di procedura è conclusa.



Vantaggi dei thread

Abbiamo già visto che il primo vantaggio dei thread rispetto ai processi consiste nel minor overhead relativamente alla creazione e alla commutazione. Ci sono però anche altri vantaggi che, in varie situazioni, fanno preferire l'uso dei thread all'uso dei processi.

Uno di questi è quello di avere una **comunicazione più efficiente**. In pratica, visto che i thread (diversamente dai processi) condividono lo spazio di indirizzamento del processo genitore, possono comunicare tra loro attraverso dati condivisi anziché mediante messaggi, evitando in questo modo l'overhead di comunicazione dovuto alle system call.

Con i thread il **cambio di contesto è molto più veloce** in quanto è relativo ad una minore quantità di informazioni da commutare.

Un altro vantaggio rispetto ai processi è la **progettazione semplificata**. Infatti l'uso dei thread può semplificare la progettazione e la codifica delle applicazioni che servono le richieste concorrentemente (es: prenotazioni dei voli online).

La creazione e la terminazione dei thread è più efficiente rispetto alle medesime operazioni sui processi; tuttavia, il suo overhead può causare un decadimento delle prestazioni del server nel caso in cui i client effettuassero un numero elevato di richieste. Per questo motivo si ricorre ad un'organizzazione chiamata **thread pool** che evita questo overhead: consiste nel riutilizzare i thread invece di distruggerli dopo aver soddisfatto le richieste evitando così l'overhead dovuto alla creazione e alla terminazione dei thread.

Codifica per l'utilizzo dei thread

I thread assicurano la correttezza dei dati condivisi e la sincronizzazione. Però ciò avviene se l'applicazione che utilizza i thread è codificata in *thread safe*; ciò non accade se è codificata in *thread unsafe*. Nelle applicazioni thread safe i dati globali sono protetti dall'utilizzo della mutua esclusione, dunque non è possibile produrre risultati inconsistenti.

Operazioni tipiche sui thread

- *thread_create*: un thread ne crea un altro;
- *thread_exit*: il thread chiamante termina;
- *thread_join*: un thread si sincronizza con la fine di un altro thread;
- *thread_yield*: il thread chiamante rilascia volontariamente la CPU.

I thread permettono prestazioni efficienti con core *hypertreading* (abilitati a gestire più thread allo stesso tempo) e soprattutto con sistemi **multicore**; con un sistema single-core abbiamo una esecuzione *interleaved*, ovvero i thread vengono elaborati in coda ciclicamente a poco a poco.

4 thread da gestire su sistema single core (pseudo parallelismo)

T1 -> T2 -> T3 -> T4 -> T1 -> T2 -> T3 -> T4 -> ...

4 thread da gestire su sistema multi core (parallelismo puro)

core 0 T1 -> T3 -> T1 -> T3 -> ...

core 1 T2 -> T4 -> T2 -> T4 -> ...

Progettare programmi che sfruttino le moderne architetture multicore non è banale. I principi base sono:

- Separazione dei task;
- Bilanciamento;
- Suddivisione dei dati;
- Dipendenze dei dati;
- Test e debugging.

Livelli di thread

Tre sono i modelli di thread utilizzati: thread di livello kernel, thread di livello utente e thread ibridi. Ognuno di essi ha differenti implicazioni sull'overhead della commutazione, sulla concorrenza e sul parallelismo.

- **Thread di livello kernel (modello 1-a-1):** Un thread di livello kernel è implementato dal kernel, dunque la creazione, la terminazione e il recupero dello stato di un thread kernel sono effettuati mediante system call (praticamente tutti i moderni SO). Inoltre c'è unica tabella dei thread del kernel. L'accesso da parte di un thread ad una area dello spazio di indirizzamento che non è attualmente paginata, non implica il blocco di tutti gli altri thread dello stesso processo; dunque un thread che non rilascia spontaneamente la CPU può comunque bloccare gli altri thread dello stesso processo. Questo modello può essere utilizzato solo se supportato nativamente dal SO.

VANTAGGI: un thread di livello kernel è come un processo eccetto che ha una quantità inferiore di informazioni di stato; un thread su chiamata bloccante non intralzia gli altri;

SVANTAGGI: la commutazione dei thread è effettuata dal kernel dunque viene generato overhead anche se il thread interrotto e il thread selezionato appartengono allo stesso processo; cambio di contesto più lento (richiede trap); la creazione e la distruzione sono più costose (numero di thread kernel tipicamente limitato, possibile riciclo).

- **Thread di livello utente (modello 1-a-molti):** I thread di livello utente sono implementati da una libreria di thread, che viene linkata al codice del processo. In questo tipo di thread il kernel non viene coinvolto ed è la libreria stessa che gestisce l'alternanza dell'esecuzione dei thread nel processo. In questo modo, il kernel non è a conoscenza della presenza dei thread di livello utente in un processo; il kernel vede solo il processo.

VANTAGGI: la sincronizzazione e la schedulazione dei thread non sono implementate dal kenel (quindi non richiedono trap), e lo scheduling è personalizzato; utile se non c'è supporto da parte del kernel ai thread.

SVANTAGGI: usando thread di questo tipo, il kernel non conosce la differenza tra thread e processo, per cui se un thread si bloccasse su una system call, il kernel bloccherebbe il processo genitore. Vi è anche la possibilità di non rilascio della CPU.

- **Modello dei thread ibrido (modello multi-a-molti):** Un modello del genere implementa sia i thread di livello utente che i thread di livello kernel e anche un metodo per associare i thread di livello utente ai thread di livello kernel (assegnazione decisa dal programmatore). Come risultato si possono ottenere differenti combinazioni caratterizzate da ridotto overhead di commutazione dei thread di livello utente ed elevata concorrenza e parallelismo dei thread di livello kernel. Esso prevede un certo numero di thread del kernel, e ognuno di essi viene assegnato ad un certo numero di thread utente (eventualmente uno), tramite un assegnazione decisa dal programmatore.

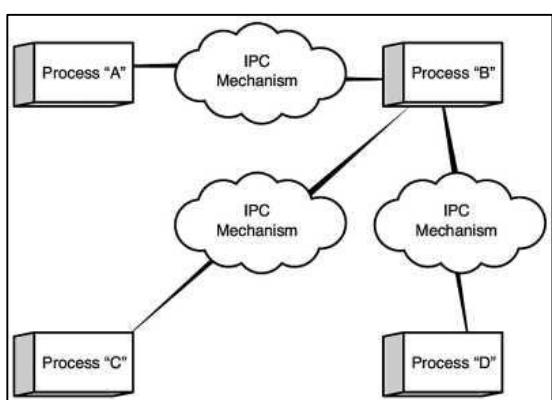
La libreria di thread crea thread utente in un processo e, a ogni thread utente, associa un thread control block (TCB). Il kernel crea i thread kernel in un processo e, a ogni thread kernel, associa un kernel thread control block (KTCB). Ci sono tre metodi per associare i thread di livello utente ai thread di livello kernel.

1. molti a uno – il kernel crea un singolo thread kernel nel processo e tutti i thread utente creati nel processo sono associati con l'unico thread kernel.
2. uno a uno – ogni thread utente è mappato permanentemente su un thread kernel.
3. uno a molti – ad ogni thread utente è permesso di essere mappato su differenti thread kernel in momenti diversi.

Thread nei SO moderni

Quasi tutti i SO supportano i thread a livello kernel (Windows, Linux, Solaris, Mac OS X,...) Il Supporto ai thread utente avviene attraverso apposite librerie: green threads su Solaris; GNU portable thread su UNIX; fiber su Win32.

Comunicazione fra processi



Processi collegati che cooperano delegando del lavoro ad altri spesso hanno bisogno di comunicare fra loro in modo ben strutturato e senza utilizzare interruzioni per sincronizzare le loro attività. Questa comunicazione è detta **InterProcess Communication (IPC)**.

Sincronizzazione tra processi

I concetti chiave della sincronizzazione dei processi sono essenzialmente due: la **sincronizzazione per l'accesso ai dati** e la **sincronizzazione per il controllo**. La prima riguarda l'uso della mutua esclusione per salvaguardare la consistenza dei dati condivisi, mentre la seconda riguarda l'uso delle operazioni atomiche per il coordinamento delle attività dei processi.

Scopo della comunicazione

Soltanamente un'applicazione è composta da vari processi che interagiscono tra loro per il raggiungimento di un obiettivo comune. Questi processi sono detti processi concorrenti o processi **intercomunicanti**. Due o più processi possono interagire fra loro secondo due modalità: cooperazione e competizione.

- Due processi **cooperano** se ciascuno ha bisogno dell'altro per procedere con le proprie operazioni.
- Due processi **competono** se entrano in conflitto sulla ripartizione delle risorse.

In entrambi i casi occorre predisporre meccanismi di sincronizzazione e comunicazione che permettano al processo di gestire la cooperazione e la competizione.

Processi concorrenti

Col termine processi concorrenti ci si riferisce a processi il cui comportamento è influenzato dalla contemporanea presenza di altri processi. I processi intercomunicanti sono processi concorrenti che condividono dati o che coordinano le loro attività. I processi che non interagiscono tra loro sono detti processi indipendenti. La sincronizzazione dei processi riguarda i processi intercomunicanti e consiste nell'individuare le tecniche usate per ritardare e ripristinare i processi e per implementare le interazioni tra i processi stessi.

Interazione tra processi

Il modo in cui i processi interagiscono può essere classificato sulla base del grado di conoscenza che hanno dell'esistenza degli altri processi.

- *Processi che non si vedono tra loro*: sono processi indipendenti che non sono fatti per collaborare. In questo caso il SO deve preoccuparsi della gestione della competizione per le risorse.
- *Processi che vedono gli altri processi indirettamente*: sono processi che non conoscono necessariamente il nome degli altri processi, ma condividono con loro l'accesso a qualche oggetto, come un buffer di I/O. Tali processi effettuano cooperazione nel senso che condividono un oggetto comune.
- *Processi che vedono gli altri processi direttamente*: Sono processi che possono comunicare direttamente fra loro per nome, e che sono progettati per lavorare insieme; anche questi processi effettuano cooperazione.

Tipi di comunicazione

I processi possono risiedere sullo stesso computer o essere distribuiti su una rete. Nel caso specifico di processi distribuiti, i meccanismi di comunicazione fra processi sono in effetti **protocolli di rete**. Si deve notare, tuttavia, che non tutti i protocolli di rete sono meccanismi di IPC; ai livelli bassi della gerarchia OSI, infatti, il concetto di processo non compare, e i protocolli si limitano a farsi carico del trasferimento di dati fra computer. L'esempio forse più noto di protocollo di rete per lo scambio di informazioni fra processi è TCP; per un esempio di IPC a livello di linguaggio di programmazione si pensi a RMI di Java. Di norma, i meccanismi di IPC per processi distribuiti permettono la comunicazione anche fra processi residenti sulla stessa macchina.

In senso lato si può intendere come meccanismo di comunicazione fra processi anche la semplice **clipboard** che consente a un utente di copiare e incollare informazioni da una finestra a un'altra, o l'uso di file, che un processo scrive e un altro legge; tuttavia, si parla di IPC in senso stretto solo per quei meccanismi che possono essere usati dal software senza intervento manuale umano e che non memorizzano i dati su memorie di massa

Mutua esclusione

Tecnica usata per gestire l'accesso ordinato ai dati condivisi.

Corse critiche

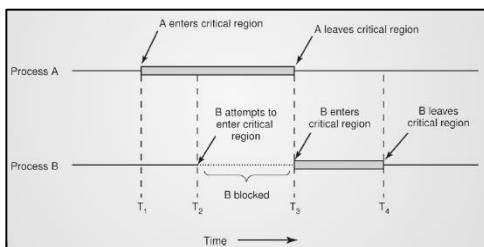
In alcuni SO, i processi che lavorano insieme possono condividere una parte di memoria comune, che ciascuno può leggere o scrivere. Situazioni nelle quali due o più processi stanno leggendo o scrivendo un qualche dato condiviso ed il risultato dipende dall'ordine in cui vengono eseguiti i processi, vengono dette corse critiche o **race condition**. In pratica si ha una race condition quando due processi accedono contemporaneamente alla stessa parte di memoria.

Quando un processo vuole stampare un file, aggiunge il nome del file in una directory speciale detta directory di spool (già incontrata prima). Un altro processo, il demone della stampante, controlla periodicamente se ci sono file da stampare, e in tal caso li stampa e rimuove i loro nomi dalla directory. Immaginiamo che la nostra directory di spool abbia un gran numero di elementi, numerati 0, 1, 2, ... Immaginiamo anche che ci siano due variabili condivise, *out*, che punta al prossimo file da stampare, e *in*, che punta al prossimo elemento libero della directory. Ad un certo istante, gli elementi da 0 a 3 sono vuoti (i file sono già stati stampati) e gli elementi da 4 a 6 sono pieni (con i nomi dei file accodati per la stampa), quando, più o meno simultaneamente, i processi A e B decidono di accodare un file per la stampa. Il processo A legge *in* e ne memorizza il valore, 7, in una variabile locale chiamata *primo_slot_libero*; proprio in quell'istante arriva un'interruzione dal clock, quindi la CPU decide che il processo A è rimasto in esecuzione per abbastanza tempo e passa al processo B, che legge *in* e ottiene 7. Anche B memorizza il valore nella sua variabile *primo_slot_libero*: a questo istante, entrambi i processi pensano che il primo slot disponibile sia 7. Il processo B continua la sua esecuzione, memorizza il nome del suo file nell'elemento 7 e aggiorna a 8, poi passa a fare altre cose. Prima o poi, il processo A va di nuovo in esecuzione, dal punto esatto in cui era stato interrotto, accede a *primo_slot_libero*, vi trova 7, e scrive il nome del suo file nell'elemento 7, cancellando il nome che il processo B vi aveva appena scritto: poi calcola *primo_slot_libero+1*, che è 8, e assegna 8 a *in*. Ora la spooler directory è internamente consistente, e quindi il demone della stampante non noterà nulla di sbagliato, ma il processo B non otterrà mai niente in uscita. È stata quindi svolta una corsa critica tra i due processi per A e B per accaparrarsi la CPU, e in questo caso specifico B ha perso l'uso della risorsa provocando un disservizio.

Per evitare le race condition bisogna assicurarsi che solo un processo alla volta possa accedere e modifica i dati in comune. Questa condizione richiede una forma di sincronizzazione tra processi.

Mutex con sezioni critiche

Per prevenire i guai che coinvolgono memoria condivisa, abbiamo bisogno della mutua esclusione (mutual exclusion o mutex), un qualche modo per assicurarsi che quando un processo sta usando una variabile o un file condiviso, gli altri processi saranno impossibilitati a fare la stessa cosa.



Il termine mutex (contrazione dell'inglese mutual exclusion, mutua esclusione) indica un procedimento di sincronizzazione con il quale si impedisce che più processi accedano contemporaneamente agli stessi dati in memoria o ad altre risorse soggette a race condition (o corsa critica). Questo concetto riveste importanza fondamentale nella programmazione parallela e soprattutto per i sistemi di transazione.

La mutua esclusione viene implementata usando le **sezioni critiche (o regioni critiche)**: queste sono sezioni del codice che accedono ad una risorsa condivisa ma che possono essere eseguite da un solo processo per volta, per non generare errori o malfunzionamenti della risorsa a cui fa riferimento il codice. In pratica, se un processo P1 sta eseguendo una sezione critica ed un processo P2 vuole eseguire quella sezione critica, P2 dovrà attendere finché P1 non termini l'esecuzione della sua sezione critica.

Condizioni per il mutex

Affinché sia possibile la mutua esclusione si devono soddisfare le quattro **condizioni di Dijkstra**:

1. Due processi non devono mai trovarsi contemporaneamente all'interno della sezione critica
2. Non si deve fare alcuna ipotesi sulle velocità e sul numero delle CPU
3. Nessun processo in esecuzione fuori dalla sua sezione critica può bloccare altri processi
4. Nessun processo deve aspettare indefinitivamente per poter entrare nella sua sezione critica.

Tecnica di realizzazione del mutex

Per realizzare l'esclusione reciproca si assegna ad un oggetto o porzione di programma (sezione critica) un elemento che va sempre controllato prima che un processo o thread possa eseguire istruzioni sull'oggetto stesso. Se un processo o thread sta già accedendo all'oggetto, tutti i successivi devono aspettare che il primo finisca. Gli oggetti mutex utilizzati per coordinare processi diversi devono essere di per sé accessibili a tutti i processi coinvolti, il che implica necessariamente l'uso di memoria condivisa, oppure gestita direttamente dal SO.

Per realizzare ciò, la procedura di utilizzo di una risorsa critica deve essere strutturata nelle seguenti fasi:

(1) Richiesta → (2) Sezione Critica → (3) Rilascio

All'avvio, mediante la fase di richiesta, il processo verifica se un altro processo sta utilizzando la sezione critica. Al termine, mediante la fase di rilascio, il processo segnala che la risorsa critica utilizzata è libera e dunque utilizzabile da un altro processo.

Implementazioni

L'implementazione più comune dei mutex fa uso di monitor, ma lo stesso risultato si può ottenere anche per mezzo di semplici lock o semafori. È spesso possibile migliorare la tecnica di accesso con l'ausilio di lock read/write che consentono un numero illimitato di accessi in lettura ma uno solo in scrittura. Questa tecnica è impiegata soprattutto per regolare l'accesso ai file e alle banche dati. Per implementare un mutex in maniera efficiente è necessario che il SO offra uno scheduler adatto. Senza questa predisposizione, ed in particolare su molti sistemi operativi real-time, bisogna ricorrere a **spinlock** che però riducono l'efficienza del multitasking poiché utilizzano il processore durante le attese.

Supporto

Alcuni linguaggi di programmazione offrono la tecnica mutex come parte del linguaggio stesso, in particolare Ada, Java e i linguaggi di programmazione .NET. Per quasi tutti gli altri linguaggi esistono librerie che implementano il sistema mutex. Questo può essere integrato come parte dell'API o dell'ambiente runtime.

Problematiche

L'esclusione reciproca comporta il rischio di **deadlock**, situazione in cui più task si bloccano vicendevolmente e nessuno può più proseguire (**starvation**). Il problema dei filosofi a cena è un esempio di questa circostanza. Esistono algoritmi speciali per raggiungere questo inconveniente (algoritmo di Peterson, algoritmo di Dekker) che si può facilmente evitare ponendo cura alla programmazione.

Tecniche di mutua esclusione

Esistono varie proposte per ottenere la mutua esclusione, alcune usano però l'attesa attiva (busy waiting).

Disabilitazione delle interruzioni

La soluzione più semplice è di permettere a ciascun processo di disabilitare le interruzioni non appena entra nella sua sezione critica, in modo tale che non possa essere prelazionato mentre è nella sezione critica e di riabilitarle non appena ne esce. Tale tecnica usa un approccio HW, ovvero la sincronizzazione tra processi viene implementata utilizzando istruzioni macchina speciali fornite dall'architettura.

Questo approccio ha però dei difetti: e se un processo disabilitasse le interruzioni e non le riabilitasse più? Potrebbe essere la fine del sistema. In più, in un elaboratore multiprocessore, con due o più CPU, la disabilitazione delle interruzioni avrebbe effetto solo sul processore che esegue l'istruzione *disable*, mentre le altre continuerebbero l'esecuzione e potrebbero accedere alla memoria condivisa. D'altra parte, però, è spesso conveniente che lo stesso kernel disabiliti le interruzioni per poche istruzioni, mentre sta aggiornando variabili o liste.

Per questi motivi, i SO implementano le sezioni critiche e le operazioni atomiche attraverso istruzioni indivisibili fornite dai computer, insieme a variabili condivise chiamate variabili di *lock*.

Variabili di lock

È una soluzione di tipo SW che consiste nell'allocare all'interno della memoria condivisa (in maniera tale da essere visibile ad ogni processo) una certa variabile (o parola o bit) che assume la funzione di "lucchetto" (**lock**) per l'accesso ad una certa risorsa condivisa. Ogni processo può accedere al suo valore e controllare se essa è "chiusa" o "aperta", cioè se tale flag è settato a 0 o a 1. Quando un processo vuole entrare nella sua sezione critica, controlla prima la sua variabile di lock e se vale 0, la mette a 1 ed entra, altrimenti aspetta che sia 0.

Supponiamo però che un processo legga la variabile di lock e veda che contiene 0, ma prima che possa metterla a 1, viene schedulato un altro processo che va in esecuzione e setta la variabile di lock a 1. Quando il primo processo torna in esecuzione, imposterà, a sua volta, la variabile a 1 e i due processi saranno contemporaneamente nella loro sezione critica, non garantendo la mutua esclusione.

Alternanza stretta

Si utilizza una variabile **turno**, inizialmente posta a 0 che tiene traccia del processo al quale tocca entrare nella sezione critica. Inizialmente, il processo 0 legge turno (=0) ed entra nella propria sezione critica; anche il processo 1 trova turno a 0 ed entra in un piccolo ciclo, testando in continuazione turno per vedere quando sarà uguale a 1. Il testare continuamente una variabile si dice **busy waiting** (in pratica, c'è un ciclo while con il quale il processo controlla se qualche processo è in sezione critica per lo stesso dato; in caso negativo, entra in sezione critica; in caso affermativo, continua a ciclare finché l'altro processo termina).

Un lock che usa l'attesa attiva si dice **spin lock**. Quando il processo 0 lascia la sezione critica, pone turno a 1 per permettere al processo 1 di entrare. Il processo 1 entra e termina rimettendo il turno a 0. Adesso però il processo 0 non ha bisogno di entrare nella sua area critica, ma è il suo turno; il processo 1 ha di nuovo bisogno di entrare in sezione critica ma il turno è di processo 0 che non lo rilascia perché esegue codice non critico. Questa proposta non è ottima se un processo è molto più lento dell'altro. Inoltre, questa situazione viola la terza condizione, ossia, un processo è bloccato da un altro che non è nella propria sezione critica.

```
while (true) do
    while (turn != 0) do
        nothing
    critical_region()
    turn = 1
    noncritical_region()
```

```
while (true) do
    while (turn != 1) do
        nothing
    critical_region()
    turn = 0
    noncritical_region()
```

Algoritmo di Peterson

La soluzione di Peterson dice che prima di usare le variabili condivise (prima di entrare nella sezione critica), ciascun processo richiama la funzione *entra_nella_regione* con il proprio numero di processo (0 o 1) come parametro, il che può far sì che il processo debba aspettare, se è il caso, fintanto che risulti sicuro entrare nella sezione critica. Dopo avere finito di lavorare sulle variabili condivise, il processo chiama *lascia_la_regione* per indicare che ha finito.

Vediamo come funziona: inizialmente nessun processo è nella regione critica, poi il processo 0 chiama *entra_nella_regione*, indica il suo interesse impostando il proprio elemento nel vettore e mette turno a 0; ora se il processo 1 tentasse di entrare rimarrebbe in attesa. Se entrambi i processi dovessero chiamare la *entra_nella_regione* quasi contemporaneamente, memorizzerebbero il proprio numero di processo nella variabile turno e qualunque cosa venga memorizzata, il secondo valore memorizzato è quello che conta, mentre il primo viene sovrascritto e perso.

```

int N=2
int turn
int interested[N]

function enter_region(int process)
    other = 1 - process
    interested[process] = true
    turn = process
    while (interested[other] = true and turn = process) do
        nothing

function leave_region(int process)
    interested[process] = false

```

I problemi di questo algoritmo sono che c'è ancora busy waiting, non funziona con più di 2 processi e può avere malfunzionamenti sui moderni multi-processori a causa del riordino degli accessi alla memoria centrale.

Istruzione Test-and-Set-Lock (TSL)

Per evitare race condition nell'impostazione della variabile di lock, viene utilizzata un'operazione indivisibile per la lettura e la chiusura, l'istruzione TSL. L'istruzione TLS viene usata per scrivere in una locazione di memoria e restituire il suo vecchio valore come una singola operazione atomica (non interrompibile).

Se diversi processi possono accedere alla stessa area di memoria, e se un processo sta eseguendo una TLS, nessun altro processo può iniziare un'altra TLS finché il primo processo non ha terminato la propria.

In dettaglio, quest'algoritmo richiede un piccolo aiuto anche dall'HW. Molti calcolatori hanno un registro RX, nel quale TSL mette il contenuto di una parola di memoria lock, e poi memorizza un valore diverso da 0 all'indirizzo di memoria di lock. Le operazioni di lettura e memorizzazione della parola sono garantite indivisibili: nessun altro processore può accedere alla parola finché l'istruzione non è finita.

enter_region: TSL REGISTER,LOCK CMP REGISTER,#0 JNE enter_region RET	leave_region: MOVE LOCK,#0 RET
---	---

La CPU che esegue l'istruzione TSL blocca il bus di memoria per impedire che altre CPU accedano alla memoria. Per usare l'istruzione TSL, useremo una variabile condivisa, lock, per coordinare l'accesso alla memoria condivisa. Quando lock è a 0, qualunque processo può metterla a 1 usando l'istruzione TSL e poi leggere o scrivere nella memoria condivisa; quando ha finito, il processo mette lock di nuovo a 0 utilizzando una normale istruzione move.

Per garantire la mutua esclusione, la prima istruzione copia il vecchio valore di lock su un registro e mette il valore di lock a 1, dopodiché il vecchio valore di lock viene confrontato con 0; se non è uguale a 0, il blocco era già impostato, quindi il programma torna all'inizio e continua a controllare il valore. Per risolvere il problema della sezione critica, un processo chiama *entra_nella_regione* che fa attesa attiva finché il blocco non è libero, poi acquisisce il controllo e termina; dopo la sezione critica, il processo chiama *lascia_la_regione* che memorizza uno 0 in lock.

TSL si può considerare atomica in ambiente mono CPU (atomica rispetto a interrupt), ma non in ambiente multi CPU (interleaving cicli memoria), e ancora sì se blocca il bus. L'istruzione TSL è una istruzione non privilegiata e quindi eseguibile anche in modalità utente e non solo kernel.

Istruzione XCHG (eXCHaGe)

Le CPU Intel x86_64 mettono a disposizione l'istruzione XCHG, che scambia due operandi con un'unica istruzione non interrompibile. La procedura di accesso alla funzione critica diventa dove indice è una locazione di memoria inizializzata a 1. Ancora busy waiting.

Sospensione e risveglio

Sia la soluzione di Peterson che quella che usa TSL, sono corrette, ma entrambe hanno il difetto di richiedere busy waiting, il che può provocare dei comportamenti inaspettati.

Consideriamo un calcolatore con due processi, H con priorità alta e L con priorità bassa; le regole di schedulazione sono tali che H viene mandato in esecuzione non appena si trova in ready. Ad un certo istante, mentre L si trova

nella propria regione critica, H passa nello stato di ready. H comincia quindi l'attesa attiva, ma poiché L non viene mai scelto quando H è in esecuzione, L non ha mai la possibilità di lasciare la sezione critica, così H cicla all'infinito. Questa situazione si chiama problema dell'**inversione di priorità**.

Per risolvere il problema dello spin lock (attesa attiva), esistono delle primitive di comunicazione fra processi che li bloccano. Una delle coppie di primitive più semplici, è formata dalla **sleep** e dalla **wakeup**: sleep è una syscall che provoca il blocco del processo chiamante, cioè, che lo sospende finché un altro processo non lo risveglia. La chiamata wakeup ha un parametro, che rappresenta il processo che deve essere risvegliato. In alternativa, sia la sleep che la wakeup hanno un parametro, un indirizzo di memoria che serve ad accoppiare le sleep con le wakeup.

Un rimedio veloce per risolvere dunque gli algoritmi sopra descritti è quello di utilizzare dei bit di attesa della sveglia o **wakeup waiting bit**. Quando viene spedita una sveglia ad un processo che è già sveglio, questo bit viene messo a 1, altrimenti il segnale andrebbe perso. Più tardi quando il processo cercherà di sospendersi, se il bit di attesa è 1, lo mette a 0 ma non si sospenderà. Nel caso in cui si debbano gestire più di due processi, però, si dovrebbero utilizzare più bit di attesa, il problema dunque rimane.

Semafori

Un semaforo è una particolare struttura di sincronizzazione. Consiste in una variabile intera condivisa **S**, a valori non negativi, che può essere soggetta solo alle seguenti operazioni:

1. Inizializzazione: il semaforo viene inizializzato con un valore intero e positivo;
2. Operazione indivisibile down (wait): il semaforo viene decrementato. Se, dopo il decremento, il semaforo ha un valore negativo, il processo viene sospeso e accodato, in attesa di essere riattivato da un altro processo.
3. Operazione indivisibile up (signal): il semaforo viene incrementato. Se ci sono processi in coda, uno dei processi in coda (il primo nel caso di accodamento FIFO) viene tolto dalla coda e posto in stato di ready (sarà perciò eseguito appena schedulato dal SO).

Il principio fondamentale è il seguente: due o più processi possono comunicare attraverso semplici segnali, in modo tale da sincronizzarsi tra loro.

Quando un processo effettua una **down** su un semaforo, controlla se $S > 0$

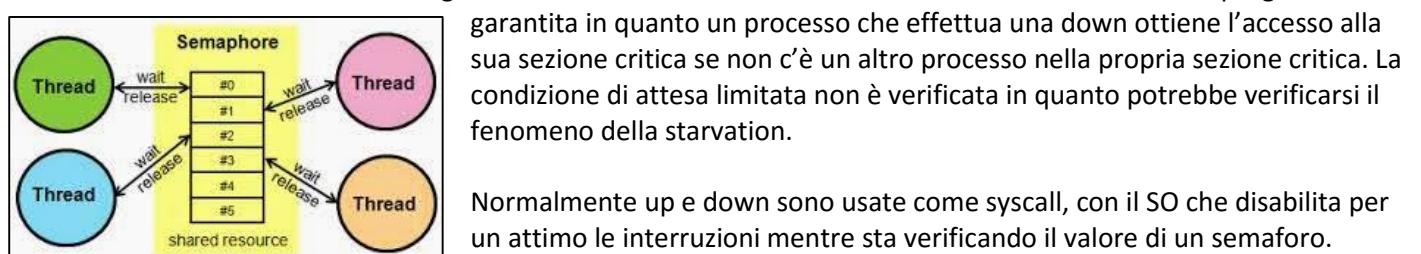
- Caso affermativo: decremente il valore del semaforo (S torna a 0) e consente al processo di proseguire la sua esecuzione nella sezione critica. Quando il secondo processo effettua una down si blocca sul semaforo perché il valore è 0; bisogna aspettare che il primo processo effettui una up una volta finito.
- Caso negativo: sospende il processo sul semaforo.

Quando un processo effettua un **up** su un semaforo, controlla se ci sono processi bloccati sul semaforo

- Caso affermativo: viene attivato un processo sospeso sul semaforo
- Caso negativo: viene incrementato S a 1; questo valore permette ad un processo, che successivamente effettua una down, di entrare immediatamente nella sua sezione critica.

Le operazioni suddette sono corrette se non vengono interrotte da altri processi. Supponiamo invece che due processi eseguano contemporaneamente l'operazione down su un semaforo che ha valore 1. Dopo che il primo processo ha decrementato il semaforo da 1 a 0, il controllo passa al secondo processo, che decrementa il semaforo da 0 a -1 e si pone in attesa. A questo punto il controllo torna al primo processo che, siccome il semaforo ha ora un valore negativo, si pone in attesa. Il risultato è che entrambi i processi si sono bloccati, mentre il semaforo a 1 avrebbe consentito a uno dei due task di procedere.

La condizione di mutua esclusione è garantita visto che il semaforo è inizializzato a 1. La condizione di progresso è



Al termine delle operazioni da parte di tutti i processi, il valore che ci aspettiamo di vedere per il semaforo è lo 1.

Semafori binari (mutex)

In alcune occasioni può essere utilizzata una versione semplificata dei semafori, detta **mutex** (semaforo binario). Un semaforo binario è un particolare semaforo usato per implementare la mutua esclusione. Esso è inizializzato a 1 ma può assumere solo i valori 0 e 1 durante l'esecuzione del programma. Le operazioni down e up sono differenti rispetto a quelle eseguite su un semaforo normale:

- L'istruzione "S = S - 1" è sostituita dall'istruzione "S = 0"
- L'istruzione "S = S + 1" è sostituita dall'istruzione "S = 1"

Ogni volta che un processo ha bisogno di accedere ai dati condivisi, acquisisce il mutex (*mutex_lock*); quando l'operazione è terminata, il mutex viene rilasciato (*mutex_unlock*), permettendo ad un altro processo di acquisirlo per eseguire le sue operazioni.

Il numero contenuto nel semaforo rappresenta il numero di risorse di un certo tipo disponibili ai processi. Il semaforo mutex è utilizzato per la mutua esclusione, mentre l'altro uso dei semafori è per garantire la **sincronizzazione** (conteggio risorse).

I **semafori vuoti e pieni** sono necessari per garantire se certe sequenze di eventi si verifichino o no in un certo ordine.

Concorrenza limitata

La concorrenza limitata implica che un'operazione possa essere eseguita da al più c processi (con $c \geq 1$). La concorrenza limitata viene implementata inizializzando un semaforo S a c . Il valore del semaforo rappresenta:

- Il numero di accessi consentiti (risorse libere) se ≥ 0
- Il numero di processi in attesa se < 0 (in valore assoluto)

I semafori utilizzati per implementare la concorrenza limitata vengono denominati semafori contatore.

Futex (fast user space mutex)

I mutex in user-space sono molto efficienti ma lo spin lock può essere lungo. Su linux viene usata la syscall Futex.

Un futex consiste in una coda di attesa di kernelspace che è collegata a un numero intero nella userspace. Più processi o thread operano sull'intero interamente in userspace (utilizzando le operazioni atomiche per evitare di interferire uno con l'altro), e ricorrono alle costose syscall solamente per richiedere operazioni sulla coda di attesa (per esempio per svegliarsi processi in attesa, o per mettere il processo corrente sulla coda di attesa).

Una lock correttamente futex-based programmata non userà chiamate di sistema, tranne quando il lock è sostenuto; poiché la maggior parte delle operazioni non richiedono arbitrarietà tra processi, questo non avverrà nella maggior parte dei casi.

I semafori utilizzati per implementare la concorrenza limitata vengono denominati **semafori contatore**.

Come si ottiene la mutua esclusione

La mutua esclusione si ottiene quindi: per processi utente (che stanno sopra il SO) li otteniamo con i semafori. Per i processi kernel invece possiamo usare soluzioni HW come TSL e interrupt oppure SW come la soluzione di Peterson.

Essendo down e up due funzioni primitive, un loro uso scorretto porterebbe a problemi di correttezza:

- Se si sostituisse alla down un up, cioè avviene up->CS -> up, più processi contemporaneamente potrebbero accedere alle proprie sezioni critiche
- Se si sostituisse alla up una down, cioè avviene down -> CS -> down, un processo, all'uscita dalla sua sezione critica, rimarrebbe bloccato così come gli altri processi che vorrebbero accedere in sezione critica, generando quindi una situazione di deadlock

Proprio per evitare tale soluzione si usa la tecnica dei monitor.

Supponiamo di avere 3 processi che condividono una variabile x e che i loro pseudo-codici siano i seguenti:

P1: wait(S) x=x-2 signal(T) wait(S) x=x-1 signal(T)	P2: wait(R) x=x+2 signal(T) wait(R)	P3: wait(T) if (x<0) signal(R) wait(T) print(x)
---	---	---

Determinare l'output del processo P3 assumendo che il valore iniziale di x è 1 e che i 3 semafori abbiano i seguenti valori iniziali: S=1, R=0, T=0.

P2 e P3 attendono su due variabili (R e T) che sono inizialmente poste a zero, dunque vengono sospesi fino a una successiva signal che li faccia uscire dalla coda di attesa. P1, invece, attende sulla variabile S, che inizialmente è posta a uno, dunque non entra in sospensione, ma semplicemente pone il semaforo associato a 0, e continua l'esecuzione portando x, dal valore 1 al valore $1-2=-1$. Adesso P1 sblocca un processo dalla coda di T (cioè P3), con signal(T), e torna immediatamente ad attendere su S (su cui, almeno in questo codice, non verrà mai più fatta una signal, e che quindi resterà in attesa per sempre...). Il processo sbloccato (P3) controlla il valore di x; lo trova pari a -1, e $-1 < 0$, quindi entra nel ramo then dell'if, eseguendo signal(R), che sblocca un processo dalla coda di R (P2), e torna in attesa su T. Il processo sbloccato dalla coda di R (P2), porta x da -1 a $-1+2=1$ e immediatamente sblocca un processo dalla coda di T. L'unico che era presente in tale attesa era P3 da pochissimo fa, che, adesso, può finalmente fare la print(x): siccome x contiene 1, P3 stamperà 1, e questa è la risposta.

Monitor

Un monitor è un costrutto di sincronizzazione ad alto livello su alcuni linguaggi (es. C#, Java) che consiste in collezioni di procedure, variabili e strutture dati che vengono raggruppate insieme in un tipo speciale di modulo o package. È un tipo astratto di dato (variabili + procedure) con garanzia di mutua esclusione e vincolo di accesso ai dati.

Un'istanza di un tipo monitor può essere utilizzata da due o più processi o thread per rendere mutuamente esclusivo l'accesso a risorse condivise. Il vantaggio nell'utilizzo del monitor deriva dal fatto che non si deve codificare esplicitamente alcun meccanismo per realizzare la mutua esclusione, giacché il monitor permette che un solo processo sia attivo al suo interno. I processi possono chiamare le procedure di un monitor ogni volta che vogliono ma non possono accedere direttamente alle strutture dati del monitor da procedure dichiarate al di fuori del monitor stesso. I monitor possiedono un'importante proprietà che li rende utili per ottenere la mutua esclusione: ad ogni istante, un solo processo può essere attivo in un monitor.

Struttura

Un monitor tipo è formato da:

- Variabili locali, i cui valori definiscono lo stato di un'istanza del tipo monitor;
- Un blocco d'inizializzazione o una procedura d'inizializzazione dei valori dei dati locali di un'istanza del tipo (detto anche **costruttore**).
- Un insieme di corpi di procedure o funzioni che realizzano le operazioni del tipo.

Le variabili locali sono dichiarate come *private*, ovvero sono accessibili solo dalle procedure del monitor. Quando un monitor viene istanziato, è avviato il blocco d'inizializzazione oppure dev'essere invocata la sua procedura d'inizializzazione, in modo analogo al modo in cui viene invocato un metodo costruttore di una classe di un linguaggio di programmazione ad oggetti quando viene istanziato un oggetto, oppure, effettivamente in quel modo, se il monitor è una classe di un oggetto. Un processo entra in un monitor invocando una delle sue procedure.

All'interno del monitor può essere attivo un solo processo per volta, sicché, quando un processo invoca una procedura, la richiesta viene accodata e soddisfatta non appena il monitor è libero.

```
monitor conto_corrente {
    double saldo:= 0.0

    procedure preleva(double importo) {
        if importo < 0.0 then error "L'importo del prelievo deve essere un numero positivo"
        else if saldo < importo then error "Fondi insufficienti per il prelievo"
        else saldo:= saldo - importo
    }

    procedure versa(double importo) {
        if importo < 0.0 then error "L'importo del versamento deve essere un numero positivo"
        else saldo:= saldo + importo
    }

    double function saldo() {
        return saldo
    }
}
```

Nell'esempio a destra, si ha la garanzia che la transazione aggiorni correttamente il saldo del conto corrente

Variabili di condizione

Sebbene i monitor forniscano una maniera semplice per ottenere la mutua esclusione, grazie alla loro proprietà, questo non è sufficiente, perché abbiamo bisogno di un modo per bloccare i processi quando non possono proseguire. La soluzione sta nell'introduzione di **variabili di condizione**, accessibili solo dall'interno del monitor.

Una variabile di condizione è una variabile con l'attributo condizione ed è associata a una condizione nel monitor. Le uniche due operazioni eseguibili su una variabile di condizione sono wait e signal:

- La **wait** sospende l'esecuzione del processo chiamante sulla condizione c; il monitor diventa disponibile per gli altri processi
- La **signal** riattiva un processo sospeso sulla condizione c; se i processi sospesi sono molti, ne sceglie uno; se non ce ne sono, non fa niente.

Queste due operazioni sono diverse da quelle dei semafori: se un processo in un monitor effettua una signal e non c'è nessun processo bloccato su quella variabile di condizione, il segnale viene perso. Inoltre, la wait e la signal del monitor non sono contatori.

Funzionamento

Quando una procedura di monitor scopre che non può continuare, chiama una wait su una qualche variabile di tipo condizione, diciamo pieni. Questa azione blocca il chiamante e permette l'ingresso a un altro processo precedentemente bloccato, il quale può svegliare il partner sospeso chiamando una signal sulla variabile di tipo condizione su cui è in attesa. Se si fa una signal su una variabile di tipo condizione su cui sono sospesi diversi processi, soltanto uno di essi, determinato dallo schedulatore del sistema, verrà risvegliato.

La wait deve avvenire prima della signal. In Java i metodi sincronizzati differiscono dai monitor tradizionali in modo sostanziale: Java non ha variabili di tipo condizione, ma ha due procedure, wait e notify che sono le equivalenti delle sleep e wakeup, eccetto che quando vengono usate in metodi sincronizzati, non sono soggette a corse critiche. In teoria il metodo wait può essere interrotto e la gestione di questo caso è lasciata al codice che lo circonda; poi Java richiede che la gestione delle eccezioni sia esplicita.

Inoltre i monitor non sono sempre implementabili perché dipendono dal linguaggio ospite (infatti la stragrande maggioranza di linguaggi non supporta i monitor) servirebbe così un'estensione ai compilatori esistenti in maniera tale da riconoscere i monitor e naturalmente questo non è fattibile.

Caratteristiche principali

Riassumendo, le caratteristiche principali di un monitor sono le seguenti:

1. Le variabili locali sono accessibili solo dalle procedure del monitor e non dalle procedure esterne;
2. Un processo entra nel monitor chiamando una delle sue procedure;
3. Solo un processo alla volta può essere in esecuzione all'interno del monitor; ogni altro processo che ha chiamato il monitor è sospeso, nell'attesa che questo diventi disponibile.

Le chiamate delle operazioni sono servite in ordine FIFO per soddisfare la proprietà di attesa limitata.

Inoltre può contenere le dichiarazioni di speciali dati di sincronizzazione chiamati variabili di condizione su cui possono essere effettuate solo le operazioni wait e signal.

Semafori vs monitor

- Ad ogni semaforo è associata una coda di processi in attesa sul semaforo.
- Ad ogni variabili di tipo condition è associata una coda su cui attendono i processi sospesi
- L'operazione down su un semaforo è sospensiva solo se il semaforo non è positivo
- L'operazione wait su un monitor è immediatamente sospensiva

Scambio di messaggi tra processi

Abbiamo detto che i monitor non sono sempre implementabili perché dipendono dal linguaggio ospite. Per aggirare questo problema usiamo la tecnica del **message passing**.

Questo metodo di comunicazione tra processi usa due primitive, **send** e **receive**, che, come i semafori e a differenza dei monitor, sono chiamate di sistema invece che costrutti del linguaggio. Come tali, possono essere facilmente messe in procedure di libreria:

- ***send(destinazione, &messaggio)***: spedisce un messaggio ad una determinata destinazione
- ***receive(sorgente, &messaggio)***: riceve un messaggio da una determinata sorgente (o da ANY, qualunque sorgente, se al ricevente non interessa una particolare sorgente).

Se non è disponibile nessun messaggio, il ricevente potrebbe bloccarsi finché non ne arriva uno; in alternativa, può terminare immediatamente con un codice d'errore.

I sistemi a scambio di messaggi però hanno molti problemi di progetto che non si presentano con i semafori o i monitor, in particolare se i processi che comunicano sono su macchine diverse connesse attraverso una rete, ad esempio, i messaggi possono essere persi dalla rete. Per prevenire la perdita di messaggi, il mittente e il destinatario possono concordare che non appena un messaggio viene ricevuto, il destinatario spedisce indietro uno speciale **messaggio di acknowledgement** (ACK, conferma dell'avvenuta ricezione); se il mittente non ha ricevuto la conferma entro un certo intervallo di tempo, ritrasmette il messaggio.

Consideriamo ora cosa succede se il messaggio stesso è stato ricevuto correttamente, ma la conferma viene persa. Il mittente ritrasmetterà il messaggio, così che il destinatario lo riceverà due volte: è essenziale che il destinatario possa distinguere un nuovo messaggio dalla ritrasmissione di uno vecchio. Di solito questo problema viene risolto mettendo numeri di sequenza consecutivi in ogni messaggio originale; se il destinatario ottiene un messaggio che porta lo stesso numero di sequenza del messaggio precedente, saprà che è un duplicato che può essere ignorato.

I sistemi a scambio di messaggi devono anche trattare il problema dell'assegnazione dei nomi ai processi, in modo che il processo specificato in una chiamata send o receive non sia ambiguo. Anche l'autenticazione è un problema nei sistemi a scambio di messaggi: come fa il cliente a dire se sta comunicando con il file server reale, e non con un impostore? È importante per la sicurezza del sistema.

Send bloccanti e non bloccanti

Le due primitive possono essere bloccanti o non bloccanti. Consideriamo la primitiva *send*, ci sono due possibilità:

- Il processo mittente si blocca finché il ricevente non riceve il messaggio
- Il processo mittente continua la sua esecuzione subito dopo aver inviato il messaggio

Analogamente, per quanto riguarda un processo che effettua una *receive*, può accadere che:

- Il processo riceve il messaggio inviato dal mittente e prosegue la sua esecuzione
- Se non ci sono messaggi in arrivo il processo si blocca in attesa di messaggi o continua l'esecuzione rinunciando a ricevere il messaggio

Possiamo sintetizzare queste situazioni distinguendo tra send e receive bloccanti e non bloccanti:

- Una send bloccante blocca il processo mittente finché il messaggio da inviare non viene consegnato al processo destinatario; questa metodologia di message passing prende il nome di scambio di messaggi sincrono
- Una send non bloccante consente a un mittente di proseguire la propria esecuzione dopo aver effettuato una chiamata send, senza preoccuparsi dell'immediata consegna del messaggio; questa metodologia prende il nome di scambio di messaggi asincrono In entrambe le metodologie, la send è tipicamente una send non bloccante.

Message passing sincrono e asincrono

Il **message passing sincrono** fornisce alcune proprietà per i processi utente e semplifica le azioni del kernel. Un processo mittente ha la garanzia che il messaggio inviato venga consegnato prima di poter continuare la propria esecuzione. Questa caratteristica semplifica la progettazione dei processi concorrenti. Il kernel consegna il messaggio immediatamente se il processo destinatario ha già effettuato una chiamata receive per ricevere un messaggio; altrimenti, blocca il processo mittente finché il ricevente non effettua una receive. Tuttavia, l'uso delle send bloccanti presenta una controindicazione, può cioè ritardare un processo mittente in alcune situazioni.

Il **message passing asincrono** migliora la concorrenza tra i processi mittente e destinatario consentendo al processo mittente di continuare la propria esecuzione. Per realizzare questa metodologia, il kernel esegue il buffering del messaggio: quando un processo effettua una send, il kernel alloca un buffer nell'area di sistema e copia il messaggio

nel buffer. In questo modo, il processo mittente è libero di accedere all'area di memoria che conteneva il testo del messaggio. Tuttavia, questa organizzazione presenta due svantaggi:

- Spreco di memoria; coinvolge un sostanziale impiego di memoria per i buffer quando molti messaggi sono in attesa di essere consegnati
- Spreco di CPU; un messaggio deve essere copiato due volte: una volta nel buffer di sistema quando viene effettuata la send e, successivamente, nell'area di messaggio del destinatario al momento della consegna.

Denominazione diretta e indiretta

Una problematica molto importante nello scambio di messaggi è l'identificazione per nome (*Naming*) dei processi, cioè la denominazione dei processi mittente e destinatario nelle chiamate send e receive. I nomi dei processi mittente e destinatario, o sono indicati esplicitamente nelle istruzioni send e receive, o sono dedotti dal kernel in un altro modo. Abbiamo due possibili schemi per specificare i processi nelle send e nelle receive.

Con l'indirizzamento diretto i processi mittente e destinatario dichiarano il proprio nome. Tale tecnica può essere utilizzata in due modi.

- Nella tecnica basata sui nomi simmetrici sia il mittente che il destinatario specificano i rispettivi nomi; in questo modo, un processo può decidere da quale processo ricevere un messaggio. Tuttavia deve conoscere il nome di ogni processo che vuole inviargli messaggi; cosa difficoltosa quando, ad esempio, i processi di applicazioni differenti vogliono comunicare.
- Nella tecnica basata sui nomi asimmetrici il destinatario non fornisce il nome del processo da cui vuole ricevere un messaggio; il kernel inoltra un messaggio inviatogli da qualche processo.

Con l'indirizzamento indiretto i processi non specificano i rispettivi nomi nelle istruzioni send e receive. In questo caso i messaggi non viaggiano direttamente dal mittente al destinatario, ma sono mandati ad una struttura dati condivisa che si compone di code che contengono contemporaneamente i messaggi. Questa struttura prende il nome di **mailbox** e possiede tre caratteristiche:

1. Ha un nome unico
2. Il proprietario della mailbox è tipicamente il processo che l'ha creata. Solo il proprietario del processo può ricevere i messaggi da una mailbox.
3. Ciascun processo che è a conoscenza del nome della mailbox gli può inviare messaggi (al processo utente della mailbox). In questo modo, i processi mittenti e destinatari utilizzando il nome di una mailbox, piuttosto che i rispettivi nomi, nelle istruzioni send e receive

Le relazioni tra mittenti e destinatari possono essere uno a uno, uno a molti, molti a molti. L'associazione dei processi con le mailbox può essere statica o dinamica. Spesso una mailbox è associata staticamente ad un solo processo in particolare se la relazione tra mittente e destinatario è di tipo uno a uno. Mentre se la relazione è di tipo uno a molti o molti a molti allora si utilizzano mailbox con associazioni dinamiche.

Vantaggi di una mailbox

L'uso di una mailbox ha i seguenti vantaggi:

- Anonimato del destinatario, infatti una mailbox consente al mittente di non conoscere l'identità del destinatario; inoltre se il SO consente di cambiare la proprietà di una mailbox dinamicamente, un processo può prontamente rilevare un altro servizio;
- Classificazione dei messaggi, un processo può creare diverse mailbox e usare ognuna per ricevere i messaggi di un tipo specifico consentendo una semplice classificazione dei messaggi.

Problemi noti di sincronizzazione dei processi

Di seguito verranno proposte alcune soluzioni ai problemi tipici riguardanti la sincronizzazione tra processi.

Problema del produttore-consumatore

Due processi si condividono un buffer di dimensioni fisse. Il produttore deposita informazione e il consumatore le preleva. Il problema nasce quando il produttore deve depositare un elemento e il buffer è già pieno. Per tener conto del numero di elementi presenti nel buffer si utilizza un contatore **cont**; il codice del produttore deve prima controllare il valore di cont e se questo è N, allora deve sospendersi, altrimenti deposita. Analogamente, il consumatore deve prima controllare se N sia maggiore di zero e poi preleva.

Può avvenire comunque la corsa critica: gli accessi alla variabile cont non sono regolamentati; potrebbe accadere che il buffer sia vuoto e il consumatore abbia appena letto cont per vedere se era 0; in quel momento lo scheduler decide di sospendere temporaneamente il consumatore e manda in esecuzione il produttore; il produttore inserisce un elemento nel buffer, incrementa cont e verifica che vale 1. Rendendosi conto che cont era poco prima a 0 e che quindi il consumatore era sospeso, il produttore chiama wakeup per risvegliare il consumatore. Sfortunatamente, il consumatore non è ancora logicamente sospeso, quindi il segnale di sveglia va perso. Quando il consumatore torna in esecuzione, verificherà che il valore di cont che aveva letto prima, ossia 0, e si sospenderà. Prima o poi il buffer ed anche il produttore si sospenderà ed entrambi resteranno sospesi per sempre.

Produttore-consumatore mediante sezioni critiche

La figura mostra una soluzione al problema dei produttori-consumatori utilizzando le sezioni critiche.

```
function producer()
    while (true) do
        item = produce_item()
        if (count = N) sleep()
        insert_item(item)
        count = count + 1
        if (count = 1)
            wakeup(consumer)

function consumer()
    while (true) do
        if (count = 0) sleep()
        item = remove_item()
        count = count - 1
        if (count = N-1)
            wakeup(producer)
        consume_item(item)
```

FUNZIONAMENTO:

Il produttore utilizza la variabile booleana PRODOTTO per interrompere il ciclo while dopo aver prodotto un elemento. Il consumatore usa la variabile booleana CONSUMATO per interrompere il ciclo while dopo che ha consumato un elemento.

Il produttore controlla ripetutamente se esistono buffer vuoti. Non appena ne trova uno, inserisce l'elemento nel buffer e imposta a true la variabile PRODOTTO. Il consumatore controlla ripetutamente se esistono buffer pieni. Non appena ne trova uno estrae l'elemento dal buffer e imposta a true la variabile CONSUMATO.

PROBLEMI:

Questa soluzione ha due problemi:

- Visto che entrambi gli accessi al buffer si trovano in sezioni critiche, allora un solo processo per volta, produttore o consumatore, può accedere al buffer in ogni istante di tempo anche se abbiamo a disposizione più buffer
- Entrambi i processi vanno in attesa attiva quando controllano se ci sono buffer pieni e buffer vuoti

Produttore-consumatore mediante semafori

La figura mostra come i semafori possono essere usati per implementare una soluzione del problema produttore-consumatore con n buffer, un processo produttore e un processo consumatore.

```
int N=100
semaphore mutex = 1
semaphore empty = N
semaphore full = 0

function producer()
    while (true) do
        item = produce_item()
        down(empty)
        down(mutex)
        insert_item(item)
        up(mutex)
        up(full)

function consumer()
    while (true) do
        down(full)
        down(mutex)
        item = remove_item()
        up(mutex)
        up(empty)
        consume_item(item)
```

Il pool di buffer è rappresentato da un array di buffer con un singolo elemento all'interno. Vengono dichiarati due semafori, *pieno* e *vuoto*. I valori dei semafori *vuoto* e *pieno* indicano il numero di buffer, rispettivamente, vuoti e pieni, per cui sono inizializzati, rispettivamente, a n e a 0. *prod_ptr* e *cons_ptr* sono usati come indici dell'array *buffer* e sono inizializzati a 0.

FUNZIONAMENTO:

Il *produttore* aspetta un buffer vuoto con una *wait(vuoto)*. Quando sono disponibili buffer vuoti, inserisce un elemento nel buffer e aggiorna l'indice *prod_ptr*. Dopo aver completato l'operazione di inserimento effettua una *signal(pieno)* per permettere al consumatore di entrare nella sua sezione critica.

Il *consumatore* aspetta un buffer pieno con una *wait(pieno)*. Quando sono disponibili buffer pieni, estrae un elemento dal buffer e aggiorna l'indice *cons_ptr*. Dopo aver completato l'operazione di estrazione effettua una *signal(vuoto)* per permettere al produttore di entrare nella sua sezione critica.

CONSIDERAZIONI:

L'uso dei segnali *wait* e *signal* permette di evitare le attese attive poiché i semafori sono utilizzati per controllare i buffer pieni e vuoti, per cui un processo sarà bloccato se non trova un buffer vuoto o pieno come richiesto.

Il livello di concorrenza in questo algoritmo è 1, a volte viene eseguito un produttore, altre volte viene eseguito un consumatore. L'algoritmo scritto in questo modo assicura che i buffer siano utilizzati in ordine FIFO.

Produttore-consumatore mediante monitor

La figura mostra una soluzione al problema produttori-consumatori mediante l'utilizzo dei monitor. Segue grossomodo lo stesso approccio della soluzione che utilizza i semafori. Nella parte superiore c'è un tipo monitor *Bounded_buffer_type*. Il pool di buffer è rappresentato da un array di buffer. *prod_ptr* e *cons_ptr* sono usati come indici dell'array *buffer* e sono inizializzati a 0. La variabile *pieno* indica il numero di buffer pieni. *buff_full* e *buff_empty* sono variabili di condizione.

FUNZIONAMENTO:

Nella procedura del *produttore*, *produce*, un produttore controlla se tutti i buffer sono pieni, in questo caso (se *pieno=n*) esegue una *buff_empty.wait*. Altrimenti entra in sezione critica, inserisce un elemento in un buffer vuoto ed incrementa il numero di buffer pieni.

Analogamente, nella procedura del *consumatore*, *consumma*, un consumatore esegue una *buff_full.wait* se *pieno=0*, cioè se tutti i buffer sono vuoti. Altrimenti entra in sezione critica, estrae un elemento da un buffer pieno ed incrementa il numero di buffer vuoti.

I consumatori ed i produttori in attesa sono attivati, rispettivamente, dalle istruzioni *buff_full.signal* e *buff_empty.signal* che si trovano all'interno delle procedure *produce* e *consumma*.

Nella sequenza di inizializzazione del monitor, la variabile *pieno* e gli indici dell'array *prod_ptr* e *cons_ptr* sono inizializzati al valore 0.

```
monitor pc_monitor
    condition full, empty;
    integer count = 0;

    function insert(item)
        if count = N then wait(full);
        insert_item(item);
        count = count + 1;
        if count = 1 then signal(empty)

    function remove()
        if count = 0 then
            wait(empty);
        remove = remove_item();
        count = count - 1;
        if count = N-1 then signal(full)

function producer()
    while (true) do
        item = produce_item()
        pc_monitor.insert(item)

function consumer()
    while (true) do
        item = pc_monitor.remove()
        consume_item(item)
```

CONSIDERAZIONI:

Sopra c'è il codice inerente al processo produttore che ovviamente invoca la procedura *produce*, e il codice inerente al processo consumatore che ovviamente invoca la procedura *consumma*.

Produttore-consumatore mediante scambio di messaggi

Supponiamo che tutti i messaggi abbiano la stessa dimensione e che i messaggi spediti ma non ricevuti siano automaticamente bufferizzati dal SO. Il consumatore inizia spedendo N messaggi vuoti al produttore: non appena il produttore ha un elemento da dare al consumatore, prende un messaggio vuoto e ne spedisce indietro uno pieno. In questo modo, il numero totale di messaggi nel sistema rimane costante nel tempo, in modo da memorizzarli in una porzione di memoria predefinita. Se il produttore lavora più velocemente del consumatore, i messaggi finiranno per essere tutti pieni, in attesa del consumatore; il produttore sarà bloccato, in attesa del ritorno di messaggi vuoti. Se è il consumatore ad essere più veloce, allora accade il viceversa: tutti i messaggi saranno vuoti, in attesa che il produttore li riempia e il consumatore sarà bloccato, in attesa di un messaggio pieno.

Vediamo come vengono indirizzati i messaggi; un modo è quello di assegnare ad ogni processo un indirizzo unico e di indirizzare i messaggi ai processi: un modo alternativo è quello di inventare una nuova struttura dati, chiamata **mailbox** (letteralmente, cassetta postale), che è un posto per bufferizzare un certo numero di messaggi, di solito specificato quando la mailbox viene creata. Quando si usano le mailbox, i parametri indirizzo nelle chiamate send e receive sono mailbox e non processi. Quando un processo cerca di spedire un messaggio ad una mailbox piena, viene sospeso finché un messaggio non viene tolto da quella mailbox, facendo spazio per uno nuovo.

Nel problema del produttore-consumatore, sia il produttore che il consumatore dovrebbero creare mailbox abbastanza grandi da contenere N messaggi. Il produttore dovrebbe spedire messaggi contenenti i dati alla mailbox del consumatore e il consumatore dovrebbe spedire messaggi vuoti alla mailbox del produttore.

Usando mailbox il meccanismo di bufferizzazione risulta chiaro: la mailbox di destinazione contiene i messaggi che sono stati spediti al processo destinatario, ma che non sono stati ancora accettati. L'altro estremo rispetto alle mailbox è l'eliminazione della bufferizzazione. Seguendo questo approccio, se la send viene chiamata prima della receive, il processo mittente viene bloccato finché non viene eseguita la receive e, a quel punto, il messaggio può essere copiato direttamente dal mittente al destinatario, senza nessuna bufferizzazione intermedia. In maniera analoga, se la receive viene chiamata per prima, il destinatario viene bloccato finché non viene eseguita la send.

```
function producer()
  while (true) do
    item = produce_item()
    receive(consumer, msg)
    build_msg(m,item)
    send(consumer, msg)
```

```
function consumer()
  for N times do
    send(producer, msg)
    while (true) do
      receive(producer, msg)
      item=extract_msg(msg)
      send(producer, msg)
      consum(item)
```

Questa strategia è spesso conosciuta come **rendez-vous (appuntamento)** ed è più facile da implementare rispetto ad uno schema con messaggi bufferizzati, ma è meno flessibile, poiché il mittente e il destinatario sono forzati ad andare in esecuzione a passi sincronizzati. Lo scambio di messaggi è usato comunemente nei sistemi di programmazione parallela. Un sistema di scambio di messaggi molto conosciuto, ad esempio, è MPI (Message Passing Interface, interfaccia per lo scambio di messaggi), che è ampiamente usata per i calcoli scientifici.

Problema del lettore-scrittore

Problema classico che modella l'accesso ad un database. C'è un processo lettore che può esclusivamente leggere i dati, e un processo scrittore può modificare o aggiornare i dati. Il primo lettore che ottiene l'accesso alla base di dati fa una down sul semaforo db. I lettori successivi incrementano un contatore rc; via via che i lettori escono, decrementano il contatore e l'ultimo esegue una up su rc, permettendo ad uno scrittore bloccato di accedere alla base di dati.

Finché c'è un regolare arrivo di lettori, questi avranno immediato accesso, mentre lo scrittore rimarrà sospeso finché non ci sono più lettori. Se arriva un nuovo lettore, supponiamo ogni 2 secondi e ogni lettore impiega 5 secondi a fare il suo lavoro, lo scrittore non avrà mai accesso alla base di dati.

Per evitare questo problema, quando un lettore arriva e uno scrittore sta aspettando, il lettore viene sospeso dietro lo scrittore, invece di essere ammesso immediatamente. Lo svantaggio di questa soluzione è che realizza meno concorrenza e, quindi, peggiori prestazioni.

Lettore scrittore con priorità ai lettori

Per evitare race condition tutte le modifiche ai contatori vengono effettuate all'interno di sezioni critiche implementate usando un semaforo binario chiamato `&mutex`.

FUNZIONAMENTO:

Questa soluzione utilizza:

- La variabile condivisa `NUM_LETTORI` per memorizzare il numero di lettori ad ogni istante di tempo. Inoltre, grazie a questa variabile è possibile determinare qual è il primo lettore ad accedere ai dati equal è l'ultimo a rilasciare i dati condivisi
- Un semaforo `MUTEX` che serve per evitare inconsistenze quando si modifica il valore di `NUM_LETTORI`
- Un semaforo `DATI` che serve per evitare che lettori e scrittori accedano contemporaneamente ai dati

Sia `MUTEX` che `DATI` sono semaforo binari inizializzati a 1.

Il lettore blocca il `MUTEX` per modificare e incrementare `NUM_LETTORI`. Se è il primo lettore, ferma gli scrittori mediante una `wait` sul semaforo `DATI`. Dopo aver sbloccato il `MUTEX` accede in lettura ai dati. Quindi riblocca il `MUTEX` per modificare `NUM_LETTORI`. Se è l'ultimo lettore, sblocca uno degli scritto r imediante una `signal` sul semaforo `DATI`. Quindi rilascia il `MUTEX`. Lo scrittore non fa altro che bloccare i lettori quando esso sta scrivendo. Quando ha terminato risveglia un lettore o un altro scrittore.

<pre> program lettori_scrittori; var numlettori: integer; mutex, dati: semaforo (:=1); procedure lettore; begin repeat wait (mutex); numlettori++; if numlettori = 1 then wait (dati); signal (mutex); LEGGI; wait (mutex); numlettori--; if numlettori = 0 then signal (dati); signal (mutex); forever end; </pre>	<pre> procedure scrittore; begin repeat wait (dati); SCRIVI; signal (dati); forever end; begin numlettori := 0; parbegin lettore; scrittore; end. </pre>
---	---

CONSIDERAZIONI:

In questa soluzione i lettori hanno la priorità: quando un lettore inizia ad accedere ai dati, i lettori possono mantenere il controllo dell'area dati finché c'è un lettore attivo, quindi gli scrittori rischiano un'attesa perenne. In pratica, fino a che ci sono processi che vogliono leggere i dati, gli scrittori devono rimanere in attesa.

Lettore scrittore con priorità ai scrittori

In questa soluzione ogni lettore che vuole leggere è accettato, a meno che uno scrittore chiede di entrare: in questo caso viene inibito l'ingresso ai lettori successivi.

FUNZIONAMENTO:

Questa soluzione usa:

- Due variabili `NUM_LETTORI` e `NUM_SCRITTORI` per tenere traccia del numero di lettori e del numero di scrittori
- I semafori binari `MUTEX1` e `MUTEX2` per evitare inconsistenze quando si modificano rispettivamente le variabili `NUM_SCRITTORI` e `NUM_LETTORI`
- Il semaforo binario `UNO_ALLA_VOLTA` per far sì che un solo lettore alla volta possa accodarsi su lettura
- Il semaforo binario `LETTURA` che viene utilizzato dagli scrittori per bloccare i lettori
- Il semaforo binario `SCRITTURA` che viene utilizzato dai lettori per impedire che lettori e scrittori accedano contemporaneamente ai dati

In pratica, ogni lettore che vuole leggere è accettato a meno che uno scrittore chiede di entrare: in questo caso viene inibito l'ingresso ai lettori successivi. Il primo lettore inibisce gli scrittori ma consente ad altri lettori di leggere. Il primo scrittore inibisce sia i lettori che gli scrittori.

```

program lettori_scrittori;
var numlettori, numscrittori: integer;
    &mutex1, &mutex2: semaforo (:=1);
    &unoAllaVolta: semaforo (:=1);
    &scrittura, &lettura: semaforo (:=1);

procedure lettore;
begin
repeat
    wait (unoAllaVolta);
    wait (lettura);
    wait (&mutex2);
    numlettori++;
    if numlettori = 1
        then wait (&scrittura);
    signal (&mutex2);
    signal (lettura);
    signal (unoAllaVolta);
LEGGI;
    wait (&mutex2);
    numlettori--;
    if numlettori = 0
        then signal (&scrittura);
    signal (&mutex2);
    forever
end;

procedure scrittore;
begin
repeat
    wait (&mutex1);
    numscrittori++;
    if numscrittori = 1
        then wait (&lettura);
    signal (&mutex1);
    wait (&scrittura);
SCRIVI;
    signal (&scrittura);
    wait (&mutex1);
    numscrittori--;
    if numscrittori = 0
        then signal (&lettura);
    signal (&mutex1);
    forever
end;
begin
    numlettori, numscrittori := 0;
    parbegin
        lettore;
        scrittore
    pend
end.

```

SPIEGAZIONE LETTORE:

Quando un processo lettore vuole leggere effettua una wait su *UNO_ALLA_VOLTA*, dopodiché effettua la stessa operazione su *LETTURA*.

A questo punto non deve far altro che incrementare il valore di *NUM_LETTORI*, proteggendo la modifica di questa variabile bloccando il mutex.

Durante questa operazione, il processo controlla anche se è il primo lettore:

- Se lo è, blocca eventuali processi scrittori che vogliono scrivere, ma permette ad altri processi lettori di leggere in modo concorrente gli stessi dati
- Se non lo è, non fa nulla e continua eseguendo le operazioni successive

Una volta letti i dati, il processo deve decrementare il valore di *NUM_LETTORI*, proteggendo la modifica riboccando il mutex.

SPIEGAZIONE SCRITTORE:

Quando un processo scrittore vuole scrivere non deve far altro che incrementare il valore di *NUM_SCRITTORI* (bloccando il mutex) e controllare se è il primo scrittore:

- Se lo è, allora deve bloccare tutti gli eventuali processi lettori che vogliono leggere i dati
- Se non lo è, continua eseguendo le operazioni successive

Prima di poter scrivere, lo scrittore deve effettuare una wait su *SCRITTURA* in modo tale da controllare se c'è un altro processo scrittore che sta scrivendo.

Una volta effettuata la scrittura, occorre decrementare il valore di *NUM_SCRITTORI* (bloccando il mutex) e controllare se il processo è l'ultimo scrittore. Se lo è, sblocca uno dei lettori in attesa.

UTILITA' DEL SEMAFORO UNO_ALLA_VOLTA:

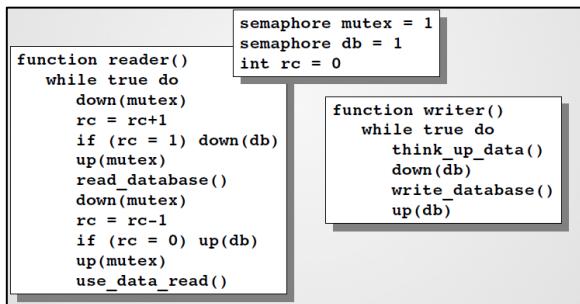
Questo semaforo torna utile quando ci sono uno o più lettori che stanno leggendo e viene accettato un processo (o più di uno) che vuole effettuare una scrittura. Supponiamo che vi siano 3 lettori che stanno leggendo, 2 scrittori che vogliono scrivere e 2 lettori che vogliono leggere ma che arrivano dopo gli scrittori.

Quando il primo dei due scrittori entra, blocca i due processi lettori che vogliono leggere mediante l'operazione *wait (lettura)*, portando il valore del semaforo a 0.

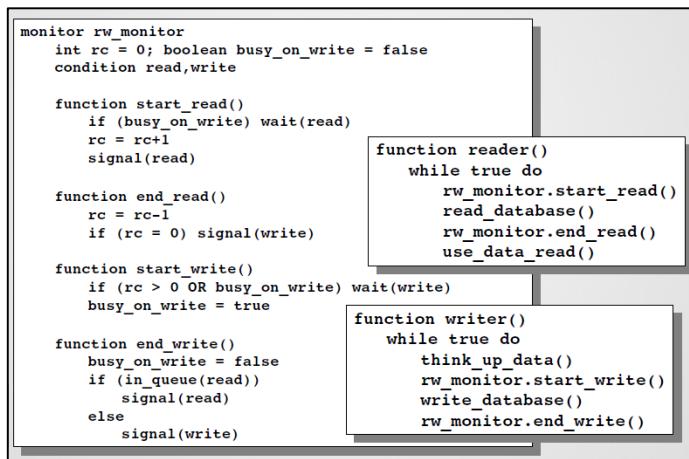
In questo modo, il primo lettore che vuole leggere, effettua la wait su *UNO_ALLA_VOLTA* e assegna valore 0 a questo semaforo, dopodiché si blocca sull'operazione *wait (lettura)* perché il suo valore è già 0. Per questo motivo, tutti i successivi processi lettori si bloccheranno sul semaforo *UNO_ALLA_VOLTA* perché il valore di quest'ultimo è 0.

In questo modo, non appena l'ultimo dei 3 processi lettori termina di leggere, sblocca il primo scrittore che scrive, dopodiché anche il secondo scrittore scrive, e soltanto ora i due lettori verranno sbloccati e potranno leggere i dati.

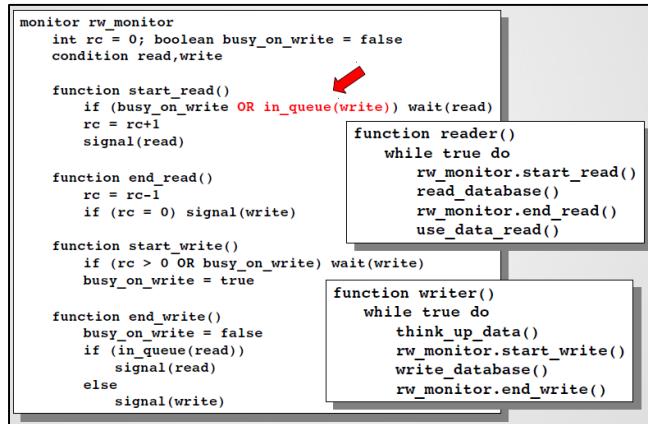
Lettore e scrittore mediante semafori



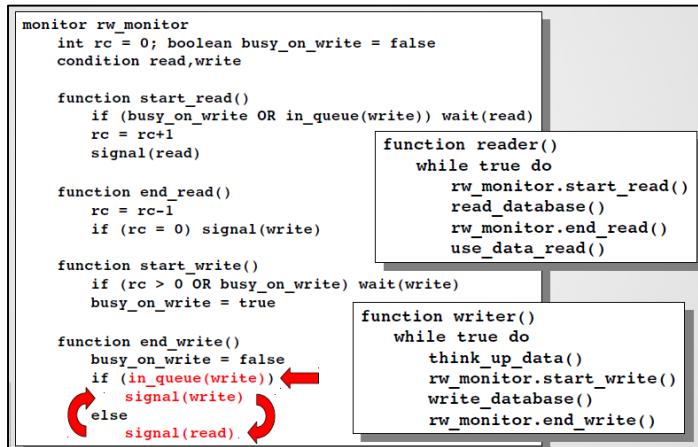
Lettore e scrittore mediante monitor (soluzione 1)



Lettore e scrittore mediante monitor (soluzione 2)



Lettore e scrittore mediante monitor (soluzione 3)

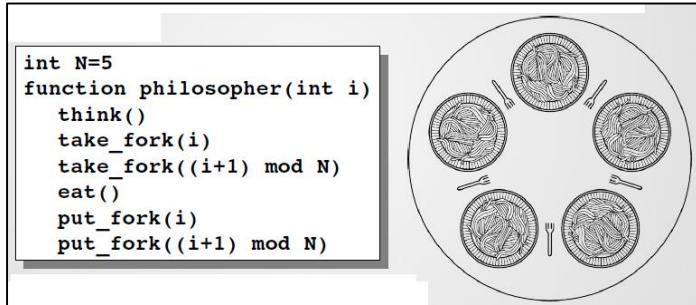


Problema dei filosofi a cena

Cinque filosofi sono seduti attorno ad un tavolo tondo e ciascun filosofo ha un piatto di spaghetti. Gli spaghetti sono così scivolosi che per mangiarli ogni filosofo deve avere due forchette e fra ogni coppia di piatti vi è una forchetta.

La vita dei filosofi alterna periodi in cui essi pensano ad altri in cui mangiano. Quando un filosofo comincia ad avere fame, cerca di prendere possesso della forchetta che gli sta a sinistra e di quella che gli sta a destra, una alla volta ed in un ordine arbitrario. Qualora riesca a prendere entrambe le forchette, mangia per un pò e, successivamente, depone le forchette e continua a pensare.

La condizione di correttezza nel sistema dei filosofi a cena è che un filosofo affamato non dovrebbe attendere indefinitamente quando decide di mangiare.



```

var   successful : boolean;
repeat
    successful := false;
    while (not successful)
        if entrambe le forchette sono disponibili then
            prendi le forchette una alla volta;
            successful := true;
        if successful = false
        then
            block (Pi);
            { mangia }
            posa entrambe le forchette;
        if il vicino di sinistra è in attesa della sua forchetta di destra
        then
            activate (vicino di sinistra);
        if il vicino di destra è in attesa della sua forchetta di sinistra
        then
            activate (vicino di destra);
            { pensa }
        all'infinito
  
```

Un filosofo controlla la disponibilità delle forchette in una sezione critica nella quale prendere anche le forchette. Per questo motivo non si possono verificare race condition.

Questa struttura assicura che almeno alcuni filosofi possono mangiare a ogni istante di tempo e previene il verificarsi di deadlock.

Un filosofo che non riesce a prendere entrambe le forchette allo stesso tempo si blocca. Verrà riattivato quando uno dei suoi vicini posa una forchetta condivisa; pertanto il processo bloccato deve controllare nuovamente la disponibilità delle forchette. Questo è lo scopo del ciclo *while*- Tuttavia, il ciclo causa una condizione di attesa attiva.

Soluzione ovvia: la procedura *prendi_forchetta* aspetta fino a che la forchetta specificata è disponibile e ne prende possesso. Sfortunatamente, la soluzione ovvia è sbagliata. Supponete che tutti e cinque i filosofi prendano la loro forchetta sinistra nello stesso istante: nessuno di loro sarà più in grado di prendere la forchetta di destra e, di conseguenza, ci sarà uno stallo. Potremmo modificare il programma in modo tale che dopo aver preso la forchetta di sinistra, il programma esegua un controllo per vedere se la forchetta di destra è disponibile. Se non lo è, il filosofo rimette a posto quella di sinistra, aspetta per un po' e poi ripete l'intero processo. Anche questa proposta non funziona, sebbene per un motivo diverso.

Con un po' di sfortuna, tutti i filosofi potrebbero cominciare l'algoritmo contemporaneamente, prendere le loro forchette di sinistra, accorgersi che la forchetta di destra non è disponibile, rimettere giù la forchetta di sinistra, aspettare, riprendere la forchetta di sinistra simultaneamente e continuare così per sempre. Una situazione come questa, nella quale tutti i programmi continuano ad essere in esecuzione per un tempo indefinito, ma nessuno di essi compie dei veri progressi, è nota come starvation (letteralmente, morte per fame).

Adesso si potrebbe pensare: "E se i filosofi aspettassero un tempo casuale anziché un tempo fisso fra un tentativo e l'altro di accedere alle forchette, la probabilità che tutto possa andare avanti a bloccarsi anche per una sola ora sarebbe molto bassa." Questa osservazione è vera e in quasi tutte le applicazioni riprovare più tardi non è un problema. Comunque, in alcune applicazioni si vuole una soluzione che funzioni sempre e che non possa fallire a causa di una serie sfortunata di numeri casuali. La soluzione ultima appare la seguente: prima di prendere possesso delle forchette un filosofo dovrebbe fare una down su mutex (semaforo). Dopo aver deposto le forchette dovrebbe fare una up su mutex. Da un punto di vista pratico, questa soluzione presenta un problema: solo un filosofo alla volta

può mangiare. La soluzione seguente presenta, invece, il massimo grado di parallelismo per un numero arbitrario di filosofi. Usa un vettore, affamato, per tenere traccia se un filosofo sta mangiando, pensando o sia affamato. Un filosofo può portare nello stato in cui mangia se nessuno dei suoi vicini sta mangiando.

Filosofi a cena con semafori

```

int N=5; int THINKING=0
int HUNGRY=1; int EATING=2
int state[N]
semaphore mutex=1
semaphore s[N]={0,...,0}

function philosopher(int i)
    while (true) do
        think()
        take_forks(i)
        eat()
        put_forks(i)

function left(int i) = i-1 mod N
function right(int i) = i+1 mod N

function test(int i)
    if state[i]==HUNGRY and state[left(i)]!=EATING and state[right(i)]!=EATING
        state[i]=EATING
        up(s[i])
    
```

```

function take_forks(int i)
    down(mutex)
    state[i]=HUNGRY
    test(i)
    up(mutex)
    down(s[i])

function put_forks(int i)
    down(mutex)
    state[i]=THINKING
    test(left(i))
    test(right(i))
    up(mutex)
    
```

Filosofi a cena con monitor

Questa situazione è priva di situazioni di stallo ma un filosofo può attendere indefinitamente.

Per codificare questa soluzione si devono distinguere i tre diversi stati in cui può trovarsi un filosofo. A tale scopo si introduce la seguente struttura di dati:

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

Occorre ricordare che, per mangiare, un filosofo deve prendere entrambe le forchette ai suoi lati. Per questo motivo, un filosofo può mangiare solo se i suoi vicini non stanno mangiando.

```

int N=5; int THINKING=0; int HUNGRY=1; int EATING=2

monitor dp_monitor
int state[N]
condition self[N]

function take_forks(int i)
    state[i] = HUNGRY
    test(i)
    if state[i] != EATING
        wait(self[i])

function put_forks(int i)
    state[i] = THINKING;
    test(left(i));
    test(right(i));

function test(int i)
    if ( state[left(i)] != EATING and state[i] = HUNGRY
        and state[right(i)] != EATING )
        state[i] = EATING
        signal(self[i])
    
```

```

monitor fc {
    enum {pensa, affamato, mangia} stato [5];
    condition auto [5];

    prende (int i) {
        stato[i] = affamato;
        verifica(i);
        if (stato[i] != mangia)
            auto[i].wait();
    }
    void posa (int i) {
        stato[i] = pensa;
        // verifica i commensali di destra e sinistra
        verifica((i + 4) % 5);
        verifica((i + 1) % 5);
    }
    void verifica (int i) {
        if( (stato[(i + 4) % 5] != mangia) &&
            (stato[i] == affamato) &&
            (stato[(i + 1) % 5] != mangia) )
            stato[i] = mangia ;
            auto[i].signal () ;
    }
    void codice di inizializzazione() {
        for (int i = 0; i < 5; i++)
            stato[i] = pensa;
    }
}
    
```

FUNZIONAMENTO:

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

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

Inoltre occorre impiegare la seguente struttura dati

```
condition auto [5];
```

dove il filosofo i può ritardare se stesso quando ha fame ma non riesce ad ottenere le forchette di cui ha bisogno. A questo punto si può descrivere la soluzione: la distribuzione delle forchette è controllata dal monitor fc . Ogni filosofo prima di cominciare deve invocare l'operazione $prende$; ciò può determinare la sospensione del processo filosofo. Completata con successo questa operazione, il filosofo può mangiare; in seguito, il filosofo invoca $pensa$. Il filosofo i deve invocare in sequenza le operazioni $prende$ e $posa$:

```
fc.prende(i); ... mangia ... fc.posa(i);
```

SPIEGAZIONE:

Il filosofo i -esimo richiama la procedura *PRENDE*. Qui cambia il suo stato in *affamato* per poi richiamare la procedura *VERIFICA*. Nella procedura *VERIFICA*, il filosofo controlla se i suoi vicini stanno mangiando e se egli stesso vuole mangiare, cioè se il suo stato è *affamato*.

Se vuole mangiare e se i suoi vicini non stanno mangiando, allora prende le forchette e mangia. Viceversa, il filosofo non mangia, cioè il suo stato rimane inalterato. Quindi, la procedura verifica, serve per modificare lo stato del filosofo i -esimo in *mangia* nel caso in cui si verifichino le condizioni elencate precedentemente.

A questo punto, il filosofo ritorna nella procedura *PRENDE* e controlla se il suo stato è cambiato. Se è *mangia* allora passa direttamente alla procedura *POSA* per posare le forchette. Se non è *mangia* allora si blocca sulla variabile di condizione *auto* in quanto non è riuscito a prendere le forchette per mangiare.

Dopo aver eseguito la procedura *PRENDE*, un filosofo esegue la procedura *POSA*. In questa procedura, il filosofo cambia il suo stato in *pensa*, dopodiché verifica prima se il suo vicino di sinistra può e vuole mangiare, poi verifica le stesse cose per il suo vicino di destra.

Un filosofo bloccato sulla variabile di condizione, viene risvegliato da uno dei suoi vicini, quando uno di questi richiama la procedura *VERIFICA* passando alla procedura stessa, il valore i del filosofo bloccato.

Scheduling

Quando un calcolatore è multiprogrammato, ha spesso diversi processi che competono per l'utilizzo della CPU. Questo fenomeno si verifica quando più processi si trovano nello stato di *ready* contemporaneamente e vi è una sola CPU a disposizione. La parte del SO che si occupa di stabilire quale deve essere il prossimo processo ad utilizzare la CPU si chiama **schedulatore**.

Lo scheduler è un componente di un SO (ovvero un programma) che implementa un **algoritmo di scheduling** il quale, dato un insieme di richieste di accesso ad una risorsa (tipicamente l'accesso alla CPU da parte di un processo da eseguire), stabilisce un ordinamento temporale per l'esecuzione di tali richieste, privilegiando quelle che rispettano determinati parametri secondo una certa politica di scheduling, in modo da ottimizzare l'accesso a tale risorsa e consentire così l'espletamento del servizio/istruzione o processo desiderato.

Esso è quella parte di codice che si occupa di selezionare il processo che passerà dallo stato *ready* allo stato *running*. Per effettuare questa scelta, lo scheduler considera una lista di richieste in attesa di essere elaborate e ne seleziona una per l'elaborazione. Una volta che una richiesta viene elaborata dalla CPU, essa può essere completata oppure può essere prelazionata (interrotta e riportata nella lista delle richieste in attesa). In entrambi i casi, lo scheduler seleziona la prossima richiesta da elaborare.

Politica di scheduling

L'attenzione posta su alcuni parametri piuttosto che su altri differenzia la cosiddetta politica di scheduling all'interno della gestione dei processi dando vita a code di priorità: ad esempio lo scheduler può eseguire le richieste in base al loro ordine di arrivo (*FIFO*), oppure dare precedenza a quelle che impegnano per meno tempo la risorsa (*SNPF*); possono esistere politiche che si basano su principi statistici o sulla predizione per individuare un ordinamento delle richieste che si avvicini il più possibile quello ottimale.

Termini dello scheduling

- **Tempo di arrivo:** istante in cui un utente invia un job o un processo
- **Tempo di ammissione:** istante in cui il sistema comincia a considerare un job o un processo per lo scheduling
- **Tempo di completamento:** istante in cui un job o un processo è terminato
- **Deadline:** istante entro il quale un job o un processo deve essere terminato per rispettare il requisito di risposta di un'applicazione real-time
- **Tempo di servizio:** il tempo totale di CPU e di I/O richiesto da un job o un processo o sottorichiesta per completare la sua operazione
- **Prelazione:** deallocazione forzata della CPU da un job o un processo
- **Priorità:** una regola discriminante usata per selezionare un job o un processo quando molti job o processi attendono l'elaborazione

Funzionamento

Nei vecchi sistemi, nei quali i dati erano conservati in forma di immagini delle schede perforate in un nastro magnetico, il processo di schedulazione era molto semplice perché bastava seguire la sequenza di dati scritti sul nastro. Poiché la CPU era una risorsa scarsa in quel periodo, il processo di schedulazione assumeva un ruolo fondamentale quando si passava all'utilizzo di sistemi time-sharing.

Con l'evoluzione tecnologica e l'avvento dei personal computer la schedulazione è cambiata in maniera radicale: in un personal computer la maggior parte del tempo c'è solo un processo attivo e quindi lo scheduler non ha molto da lavorare. In secondo luogo i calcolatori sono diventati talmente veloci nel corso dello sviluppo tecnologico che la maggior parte dei programmi sono vincolati all'inserimento dei dati da parte dell'utente che alla loro rapidissima elaborazione. Quando passiamo però a postazioni di lavoro e server di fascia in rete la situazione diventa più delicata in quanto più processi si contendono la CPU in maniera continua.

Oltre a scegliere il processo giusto da eseguire lo scheduler deve anche fare un uso efficiente della CPU poiché il cambio di processo è costoso. Se dovessimo riassumere il cambio di processo in punti distinti se ne identificano 6:

1. Passaggio da modalità utente a modalità kernel;
2. Salvare lo stato del processo corrente ed i dati nei relativi registri in modo da poterli ricaricare in seguito;
3. Salvare la mappa di memoria(i bit di riferimento ai dati memorizzati);
4. Scegliere un nuovo processo da eseguire;
5. Caricare tutti i dati necessari all'esecuzione del processo nei registri di memoria;
6. Avviare il nuovo processo.

Di solito la CPU lavora per un po' senza fermarsi, quindi esegue una chiamata di sistema per leggere da file e per scrivere su un file. Quando la chiamata di sistema è completata la CPU esegue nuovi calcoli fino a che ha bisogno di ulteriori dati o di scriverne altri e così via.

CPU e I/O Bound

Alcuni processi chiamati **CPU Bound** utilizzano il processore per la maggior parte del tempo al solo scopo di eseguire calcoli. Altri processi chiamati **I/O bound** utilizzano il processore per la maggior parte del tempo per fare operazioni di Input e Output. I processi di tipo CPU bound tengono occupato il processore per un tratto molto maggiore rispetto all' altro tipo di processi. Per questo motivo il fattore chiave diventa il tratto di utilizzo della CPU e non la lunghezza dell'attesa dei processi di I/O.

Quando interviene lo scheduler

Esistono dunque un sacco di situazioni che richiedono la schedulazione: quando viene generato un nuovo processo sia il processo padre che quello figlio si trovano nello stato di pronto e quindi sta allo scheduler l'esecuzione dell'uno o dell'altro in maniera indifferente. In altri casi il processo di schedulazione inizia il suo lavoro quando un processo termina e si deve determinare il successivo. Si deve quindi scegliere uno dei processi che si trova nello stato di pronto.

Se nessun processo si trova in questo stato viene eseguito un processo "vuoto" fornito dal sistema. Un processo può essere interrotto a causa di un semaforo o per un'operazione di I/O. Deve essere presa una decisione di schedulazione quando si verifica un'interruzione in I/O; se essa proviene da un dispositivo di I/O che ha completato il suo lavoro, il processo che era stato bloccato in attesa di quest'operazione è ora pronto per essere eseguito. È compito dello scheduler decidere se debba essere eseguito il processo che è appena passato nello stato pronto, il processo che era in esecuzione al momento in cui si è verificata l'interruzione, o un terzo processo. Deve essere presa, in fine, una decisione di schedulazione ogni ad ogni interruzione di clock dipendente esclusivamente dall'hardware del sistema. In base a ciò distinguiamo due tipologie di algoritmi di schedulazione.

Algoritmi preemptive e non preemptive

È importante la distinzione tra scheduling con diritto di prelazione (scheduling preemptive) e scheduling senza diritto di prelazione (scheduling non-preemptive o cooperative).

- **Preemptive (prerilascio):** lo scheduler può sottrarre il possesso del processore al processo anche quando questo potrebbe proseguire nella propria esecuzione
- **Non preemptive (senza prerilascio):** lo scheduler deve attendere che il processo termini o che cambi il suo stato da quello di esecuzione a quello di attesa o di pronto, a seguito, ad esempio, di una richiesta di I/O oppure a causa di un segnale di interruzione (interrupt).

Dispatcher

In informatica il dispatcher è un modulo del SO che passa effettivamente il controllo della CPU ai processi scelti dallo scheduler a breve termine. Poiché si attiva a ogni context switch, il dispatcher dovrebbe essere quanto più rapido possibile. Il tempo richiesto dal dispatcher per fermare un processo e avviare l'esecuzione di un altro è noto come latenza di dispatch.

Tipi di scheduling

Diversi ambienti di lavoro richiedono algoritmi di schedulazione differenti. Vale la pena elencarne i tre tipi più utilizzati e descriverne le caratteristiche: batch, interattivo, real-time. Gli obiettivi comuni di tali sistemi sono l'equità nell'assegnazione della CPU e il bilanciamento nell'uso delle risorse.

- Nei **sistemi batch** non ci sono processi impazienti in attesa di una risposta veloce. Quindi sono accettabili sia algoritmi non preemptive, sia quelli preemptive con un periodo lungo per ogni processo. Questo approccio riduce gli scambi tra processi, migliorando le prestazioni. Per i sistemi batch sono importanti il throughput, il tempo di turnaround e l'uso della CPU. Politiche usate: FCFS, SJF (SNPF e SRTF).
- Nei **sistemi interattivi** il prerilascio è essenziale per evitare che un processo si impossessi della CPU. Per i sistemi interattivi è importante il tempo di risposta. Politiche usate: Round-Robin, Priorità, Code Multiple, HRRN.
- Nei **sistemi real-time** il prerilascio non è sempre necessario perché i processi sanno che non possono essere eseguiti per lunghi periodi di tempo e normalmente fanno il loro lavoro e si bloccano in fretta. Come obiettivi tali sistemi hanno il rispetto delle scadenze e la prevedibilità dei processi.

Algoritmi di schedulazione

Un algoritmo di schedulazione, in generale, deve soddisfare determinati requisiti:

- Equità (processi simili devono ricevere trattamenti simili);
- Applicazione delle politiche di sistema (esempio: se la politica del sistema in uso stabilisce che i processi di controllo possono essere eseguiti quando vogliono essi devono poter prendere la precedenza sugli altri);
- Si devono tenere tutte le parti del sistema occupate (se si utilizzano contemporaneamente la CPU ed i sistemi di I/O "mescolando per bene i processi", si risparmia in tempo ed efficienza del sistema);
- Si deve tenere conto di una certa proporzionalità, un fattore che si occupa delle incidenze dei tempi di risposta di ogni processo.

First-Come First-Served (FCFS, sistemi batch)

Il più semplice degli algoritmi di schedulazione è certamente il "primo arrivato, primo servito" (coda FIFO), con il quale ai processi viene assegnata la CPU in ordine in cui è stata richiesta. Un primo processo prende la CPU per tutto il tempo che gli occorre ed i processi successivi vengono messi su un'unica coda in attesa che la CPU si liberi.

Questo algoritmo soddisfa tutti i requisiti necessari per un buon algoritmo di schedulazione, è semplice da capire; infatti basta eliminare elementi dalla testa della coda per eseguirli, ed aggiungere elementi in fondo alla coda per metterli in attesa del loro turno. D'altro canto questo procedimento presenta il grosso svantaggio della gestione fra operazioni di tipo CPU bound ed operazioni di I/O. Se per ogni operazione di CPU bound dovrebbero essere eseguite X operazioni di I/O, i tempi di esecuzione sarebbero molto più lunghi rispetto a quelli di un algoritmo che alterna tutti i processi in brevi spazi temporali. È non preemptive.

FUNZIONAMENTO:

I processi sono schedulati nell'ordine in cui giungono al sistema, cioè il primo processo ad essere eseguito è quello che per primo ha richiesto la CPU. I processi successivi vengono schedulati con lo stesso criterio non appena il processo in esecuzione completa le sue operazioni. In pratica, i processi ready sono organizzati come una coda FIFO e i processi che richiedono la CPU vengono inseriti alla fine di questa coda.

Abbiamo i seguenti processi da elaborare con l'algoritmo FCFS:

Processo	Arrivo	Durata
P1	0	7
P2	2	4
P3	4	1
P4	5	4

P1 arriva al s0 e andrà subito in elaborazione e ci rimarrà fino al compleimento in s7. P2 arriva a s2 e attenderà i 5s del rimanente completamento di P1 per andare in esecuzione ed essere completato all's9. Così via per gli altri, dove P3 e P4 attendono entrambi s7 e vengono completati in P3=8, P4=11.
 Tempi di attesa: P1=0, P2=5, P3=7, P4=7 (media 4,75)
 Tempi di completamento: P1=7, P2=9, P3=8, P4=11 (media 8,75)

Shortest Job First (SJF, sistemi batch)

Questo algoritmo prevede come requisito la conoscenza anticipata dei tempi di esecuzione di ogni processo. Si utilizza dunque in sistemi nei quali tutte le operazioni eseguite sono abituali e se ne possono stimare i tempi in maniera precisa.

Poiché viene data precedenza ai processi con minore tempo di esecuzione, questo algoritmo risulta particolarmente efficiente quando si hanno già tutti i processi in stato di pronto. È non preemptive.

FUNZIONAMENTO:

Seleziona il processo in attesa che userà la CPU per minor tempo. Se due processi hanno lo stesso tempo di esecuzione, verrà applicato lo scheduling FCFS.

In pratica, le richieste brevi tendono a ricevere prima l'uso della CPU.

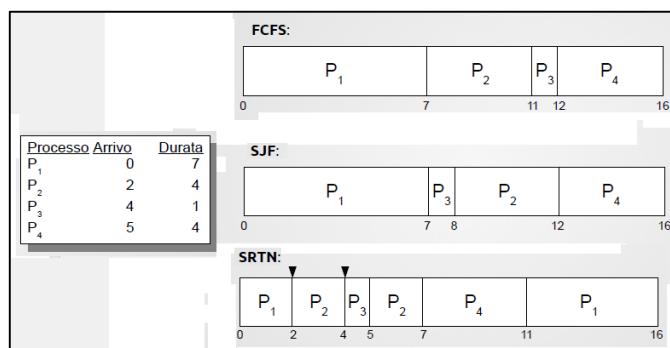
Abbiamo i seguenti processi da elaborare con l'algoritmo SJF:

Processo	Arrivo	Durata
P1	0	7
P2	2	4
P3	4	1
P4	5	4

P1 arriva al s0 e andrà subito in elaborazione e ci rimarrà fino al compleimento in s7. In quest'ultimo istante sono arrivati tutti gli altri processi, si sceglie quindi quello con durata minore che è P3 e viene eseguito, e così via.
 Ordine di elaborazione: P1, P3, P2, P4
 Tempi di attesa: P1=0, P2=6, P3=3, P4=7 (media 4)
 Tempi di completamento: P1=7, P2=10, P3=8, P4=11 (media 8)

Shortest Remaining Time Next (SRTN, sistemi batch)

È una variazione dell'algoritmo SJF descritto precedentemente con l'aggiunta del prerilascio. In questo caso lo scheduler sceglie sempre il processo con minor tempo di esecuzione residuo. Ogni nuovo processo viene confrontato con il tempo residuo del processo corrente, e se inferiore diventa il nuovo processo in esecuzione.



Supponiamo di avere i seguenti cinque processi con relative durate e tempi di arrivo:

processi	P1	P2	P3	P4	P5
durata	7	5	2	4	4
tempo di arrivo	0	3	4	6	9

Qual è il processo che sarà schedulato per ultimo?

Abbiamo detto che l'SRTN esegue prima i processi più veloci; essendo preemptive, nel momento in cui sta elaborando un processo ne arriva uno più veloce, sospende quello in corso e passa al più veloce. Quindi lo schema di esecuzione sarà (in grassetto, il processo a cui passa l'esecuzione):

t=0	t=3	t=4	t=6	t=9
P1=7	P1=4, P2=5	P1=3, P2=5, P3=2	P1=3, P2=5, P3=0, P4=4	P1=0, P2=5, P3=0, P4=4, P5=4

t=13	t=17	t=22
P1=0, P2=5, P3=0, P4=0, P5=4	P1=0, P2=5, P3=0, P4=0, P5=0	P1=0, P2=0, P3=0, P4=0, P5=0

Il processo schedulato per ultimo sarà P2.

Scheduling Round Robin (RR, sistemi interattivi)

L'algoritmo di scheduling RR (round-robin) è un particolare algoritmo di tipo preemptive che esegue i processi nell'ordine d'arrivo, come il FCFS, ma esegue la prelazione del processo in esecuzione, ponendolo alla fine della coda dei processi in attesa, qualora l'esecuzione duri più di un intervallo di tempo stabilito, e facendo proseguire l'esecuzione al successivo processo in attesa; tale intervallo di tempo viene detto **quanto**. Se durante questo quanto il processo termina, la CPU viene assegnata ad un nuovo processo, se invece non si arriva alla conclusione della procedura, la si sospende al termine del suo quanto, la si fa andare alla fine della coda e si assegna la CPU ad un altro processo.



L'unica questione interessante nel round robin è la durata del quanto: il passaggio da un processo ad un altro richiede una certa quantità di tempo per gli aspetti amministrativi: salvare e caricare registri e mappe di memoria, aggiornare varie tabelle e liste, svuotare e ricaricare la cache, eccetera. Circa il 20% di tempo della CPU verrà quindi sprecato in overhead di gestione se utilizziamo dei quanti troppo lunghi. Per ottimizzare il lavoro della CPU potremmo utilizzare dei quanti di 100 millisecondi. D'altro canto però assegnare un quanto troppo breve causerebbe l'esecuzione di troppi cambi in un piccolo intervallo di tempo. Si ricorre quindi ad un compromesso fra le due misure utilizzando quanti di dimensioni che vanno dai 20 ai 25 millisecondi.

Abbiamo i seguenti processi in coda con relativa durata in millisecondi, e quanto di tempo stabilito di 20 ms:

Processi	P1	P2	P3	P4
Durata (millisecondi)	30	15	60	45

Verranno eseguiti nel seguente ordine:

- P1 (interrotto dopo 20 ms, ne rimangono altri 10)
- P2 (termina la propria esecuzione perché dura meno di 20 ms)
- P3 (interrotto dopo 20 ms, ne rimangono altri 40)
- P4 (interrotto dopo 20 ms, ne rimangono altri 25)
- P1 (termina la propria esecuzione perché necessitava di meno di 20 ms)
- P3 (interrotto dopo 20 ms, ne rimangono altri 20)
- P4 (interrotto dopo 20 ms, ne rimangono altri 5)
- P3 (termina la propria esecuzione perché necessitava di esattamente 20 ms)
- P4 (termina la propria esecuzione)

Le prestazioni di quest'algoritmo sono dunque influenzate dal tempo medio d'attesa sebbene consenta a tutti i processi di ottenere il controllo della CPU ed evita quindi il problema dell'attesa indefinita (starvation).

È inoltre da tenere in considerazione l'impatto dovuto ai frequenti context switch effettuati. È necessario quindi calcolare correttamente la durata ottimale del quanto di tempo per far sì che l'incidenza dei cambi di contesto sia abbastanza limitata come anche i tempi di attesa. Si può stabilire che, per un funzionamento ottimale, le sequenze di operazioni di CPU dovrebbero essere più brevi del quanto di tempo stabilito in circa l'80% dei casi.

- **Tempo di attesa di un processo:** In generale, con n processi e un quanto di $q \text{ ms}$, ogni processo avrà diritto a circa $1/n$ della CPU e attenderà al più

$$(n - 1)q \text{ ms}$$

- **Calcolo della percentuale dell'overhead per la gestione dell'interlacciamento dei processi:** l'overhead associato al context switch può essere espresso dalla formula

$$\frac{C}{q + C}$$

Dove q è un quanto di $q \text{ ms}$, e C è il tempo $C \text{ ms}$ del context switch.

Supponiamo di utilizzare un algoritmo di scheduling preemptive come il Round-Robin: assumendo di avere un context switch effettuato in 5 ms e di usare quanti di tempo lunghi 50 ms, a quanto ammonta la percentuale di overhead per la gestione dell'interlacciamento dei processi?

$$\frac{5}{50+5} = 0,09 \cdot 100 = 9\%$$

Schedulazione con priorità (sistemi interattivi)

La schedulazione round robin ipotizza implicitamente che tutti i processi abbiano la stessa importanza, ma spesso le persone che gestiscono calcolatori multiutente hanno idee diverse su questo argomento. La necessità di prendere in considerazione fattori esterni porta alla schedulazione con **priorità**, la cui idea base è banale: ad ogni processo viene assegnata una priorità e viene concessa l'esecuzione al processo eseguibile con priorità più alta.

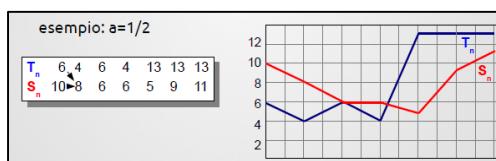
Per evitare che i processi ad alta priorità rimangano indefinitamente attivi, lo scheduler può diminuire la priorità del processo ad ogni ciclo di clock del processore. Le priorità possono essere assegnate ai processi in maniera statica o dinamica. Spesso è conveniente raggruppare i processi in classi di priorità ed utilizzare la schedulazione con priorità fra le classi, facendo eseguire tutti i processi della classe con priorità più alta per poi passare alla classe con priorità subito inferiore.

Shortest Process Next (SPN, sistemi interattivi)

È la versione per sistemi interattivi dello SJF. Nei sistemi interattivi i processi seguono lo schema secondo il quale attendono un comando, lo eseguono, ne attendono un altro e così via. Un approccio utilizzato è quello di realizzare delle stime in base al comportamento passato e passare in esecuzione il processo che ha tempo stimato più breve. Questa tecnica viene chiamata **invecchiamento** (aging) ed è applicabile in tutti quei sistemi dove è possibile fare previsioni basandosi su valori precedenti.

Il problema di tale algoritmo è quindi stimare e trovare la durata del prossimo **burst** di CPU (tempo di occupazione della CPU); la stima viene fatta sui burst precedenti tramite l'equazione

$$S_{n+1} = S_n(1 - a) + T_n a$$



Altri algoritmi di scheduling

- **Scheduling garantito:** Divide il tempo di utilizzo di una risorsa tra gli n processi presenti, stabilendo una percentuale di utilizzo che viene fatta rispettare. La priorità di un processo in attesa dipende dall'utilizzo complessivo precedente della risorse rapportato a 1/n.
- **Scheduling a lotteria:** Tecnica utile per la condivisione di una risorsa in maniera probabilisticamente equa. I 'biglietti' sono distribuiti a tutti i processi che condividono una risorsa (ad esempio il tempo di CPU). Quando lo scheduler deve selezionare un processo a cui attribuire la CPU, viene scelto a caso un biglietto e il processo che lo possiede ottiene la risorsa. Per migliorare le prestazioni, ai processi più importanti possono essere assegnati biglietti extra, per aumentare la loro probabilità di vincere. Inoltre i processi possono scambiarsi i biglietti se lo desiderano per avvantaggiare i processi con i quali cooperano per raggiungere un obiettivo comune.
- **Scheduling fair-share:** Tutte le politiche di scheduling viste finora si occupano di fornire servizi equi ai processi piuttosto che agli utenti o alle loro applicazioni. Se le applicazioni creano un diverso numero di processi, un'applicazione che impiega più processi è probabile che riceva maggiore attenzione dalla CPU rispetto ad un'applicazione che utilizza meno processi. La nozione di fair share risponde proprio a questa esigenza. Una fair share assicura un equo utilizzo della CPU da parte degli utenti o delle applicazioni. In pratica, gli utenti o le applicazioni che usano un numero più alto di processi ricevono meno risorse.

- **Scheduling a code multiple (o multilivello):** Questo scheduling combina lo scheduling basato su priorità (è una variante di esso) e quello round-robin per fornire una buona combinazione tra prestazione del sistema e tempi di risposta. Uno scheduler multilivello utilizza varie code di processi pronti, e ad ogni classe è associata una priorità. Alla coda con priorità più alta viene assegnato un *quanto* (quantità di tempo per usare la CPU) più piccolo, mentre alle code con priorità via via minore viene assegnato un quanto sempre più grande. La coda con priorità più bassa avrà il quanto più grande.

Il tipo di scheduling multilivello è un algoritmo con prelazione, con priorità e statico. Come funzionamento abbiamo detto che vengono definite classi di priorità. I processi della classe più alta vengono eseguiti per un quanto, quelli della classe successiva per due quanti, quelli della classe seguente per quattro quanti e così via.

Ogni qualvolta che un processo rimane in esecuzione per tutto il quanto, viene abbassato di una classe. Se, per esempio, un processo A ha bisogno di 100 quanti, inizialmente gli viene assegnato un quanto, dopodiché viene schedulato un altro processo. Quando A riottiene la CPU, gli vengono assegnati 2 quanti, poi 4, poi 8, poi 16, poi 32, poi 64 quanti. In pratica ottiene la CPU per sette volte anziché per cento come sarebbe avvenuto con il round-robin.

Un processo in testa a una coda è selezionato dallo scheduler solo se le code con priorità più alta sono vuote.

Questo tipo di scheduling è di tipo preemptive, quindi un processo può essere prelazionato se arriva un processo con priorità più alta; in questo caso, il processo interrotto viene aggiunto alla fine della coda a cui apparteneva.

Per ottenere un'efficienza ottimale, si dà una priorità elevata ai processi I/O bound, mentre si dà una priorità bassa ai processi CPU bound.

In questa politica di scheduling vengono utilizzate priorità statiche, per questo motivo non può prevenire la starvation dei processi a bassi livelli di priorità.

- **Scheduling a code multiple con feedback:** Questo scheduling evita la starvation promuovendo un processo ad un livello di priorità più alto se ha speso 3 secondi nel suo attuale livello di priorità senza essere schedulato. Esso usa i seguenti metodi di implementazione della promozione:

- *Promozione di un processo al massimo livello di priorità:* Con questo tipo di promozione si tende ad avere presto tutti i processi al livello massimo. Infatti più processi vanno al livello massimo, più essi bloccano altri processi che quindi andranno a loro volta al livello massimo. In pratica la soluzione degenera in FCFS.
- *Promozione di un processo al successivo livello di priorità:* Questa sembra una politica più equilibrata: può non essere sufficiente a permettere l'esecuzione di un processo, ma allora quest'ultimo verrà ulteriormente promosso.

Scheduling dei thread

- **Thread utente:**
 - ignorati dallo scheduler del kernel;
 - per lo scheduler del sistema run-time vanno bene tutti gli algoritmi non-preemptive visti;
 - possibilità di utilizzo di scheduling personalizzato.
- **Thread del kernel:**
 - o si considerano tutti i thread uguali, oppure;
 - si pesa l'appartenenza al processo;
 - lo switch su un thread di un processo diverso implica anche la riprogrammazione della MMU.

Scheduling su sistemi multiprocessore

- Possibili approcci:
 - **multielaborazione asimmetrica:**
 - uno dei processori assume il ruolo di **master server**;
 - **multielaborazione simmetrica (SMP):**
 - **coda unificata** dei processi pronti o **code separate** per ogni processore/core.
- Politiche di scheduling:
 - presenza o assenza di **predilezione per i processori**:
 - **predilezione debole** o **predilezione forte**;
 - **bilanciamento del carico:**
 - necessaria solo in presenza di code distinte per i processi pronti;
 - **migrazione guidata** o **migrazione spontanea**;
 - possibili **approcci misti** (Linux e FreeBSD);
 - bilanciamento del carico vs. predilezione del processore.

Scheduling in Windows, Unix e Linux

Scheduling in Windows 8

Windows usa uno scheduler preemptive basato sui thread con classi di priorità. Tali priorità sono gestite dalle api Win32, i quali comandi sono:

- **SetPriorityClass**: imposta la classe per l'intero processo;
- **SetThreadPriority**: imposta la priorità relativa per un singolo thread;

Tali api permettono di specificare:

- Priorità di un processo (6 livelli diversi)
- Priorità di un thread all'interno di un processo (7 livelli diversi)

priorità relativa	classi di priorità					
	real-time	high	above normal	normal	below normal	idle priority
time-critical	31	15	15	15	15	15
highest	26	15	12	10	8	6
above normal	25	14	11	9	7	5
normal	24	13	10	8	6	4
below normal	23	12	9	7	5	3
lowest	22	11	8	6	4	2
idle	16	1	1	1	1	1

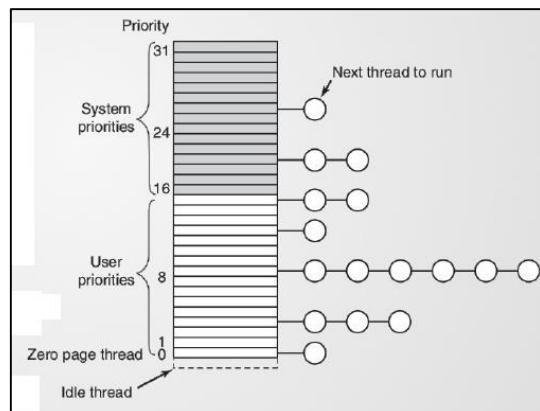
L'algoritmo usato seleziona per priorità più alta, con round-robin interno alla classe. Le priorità sono dinamiche, euristiche, ovvero:

- Utilizzo del quanto di tempo;
- Ripresa da un I/O da tastiera/mouse o da I/O su disco;
- Processi in primo piano con quanto di tempo più ampio;

Vi è il problema dell'inversione di priorità (autoboost).

I thread tipicamente entrano a priorità 8. La priorità viene elevata (problema dell'inversione di priorità) se:

- Viene completata una operazione di I/O (+1 disco, +2 linea seriale, +6 tastiera, +8 scheda audio ...)
- Termina l'attesa su un semaforo, mutex
- L'input nella finestra di dialogo associata al thread è pronto



La priorità viene abbassata se un thread usa tutto il suo quanto (-1). Se un thread non ha girato per un tempo maggiore di una soglia fissata, allora passa per 2 quanti a priorità 15 (serve a gestire potenziali inversioni di priorità).

Scheduling in Unix

Unix è un SO time-sharing puro che usa una politica di scheduling multilivello adattivo, quindi viene associata ad ogni processo una priorità, che in Unix è di tipo numerica: valori alti corrispondono a basse priorità e viceversa. Vengono schedulati i processi con priorità maggiore (cioè col valore vicino allo 0).

Ad esempio, in Unix 4.3, le priorità variano nell'intervallo 0-127. I processi in modalità utente hanno una priorità tra 50 e 127, mentre quelli in modalità kernel hanno priorità tra 0 e 49.

In Unix, viene usata la seguente formula per variare la priorità di un processo:

$$\text{Priorità processo} = \text{priorità base per processi utente} + f + \text{valore nice}$$

Dove:

- f è il tempo di CPU usato recentemente
- *nice* è una system call che permette di modificare la priorità del processo chiamante

Con l'uso del secondo fattore dell'equazione, cioè f , viene assicurata un'equa divisione del tempo di CPU tra i gruppi di processi (*scheduling fair share*). Infatti, la f è relativa all'uso della CPU, più è alto il suo valore, minore sarà la priorità del processo. Quindi se un processo ha usato la CPU recentemente, vedrà la sua priorità diminuire.

La system call *nice* permette di modificare la priorità del processo chiamante. Visto che può accettare solo un valore ≥ 0 , il terzo parametro dell'equazione non può far altro che diminuire la priorità del processo. Un processo potrebbe invocare una *nice* per favorire altri processi con i quali coopera per il raggiungimento di un obiettivo comune.

Scheduling in Linux

Nel caso specifico di linux esistono due tipi di scheduler, precisamente **l'Input/Output scheduler** e **il Task scheduler**. Il primo si occupa di organizzare tutti i processi che necessitino di gestire del traffico di dati da una periferica, mentre il secondo si occupa della gestione dell'esecuzione vera e proria dei programmi attivi sulla macchina.

Da Linux 2.6 viene adottato lo **scheduler O(1)**, sostituito dalla 2.6.23 dal Completely Fair Scheduler (CFS). Possiamo distinguere tre classi di thread:

- Real-time FIFO (non preemptive);
 - Real-time round-robin (preemptive);
 - Timesharing (preemptive);

Ogni thread ha una priorità nell'intervallo [0,+40]; generalmente all'inizio la priorità di default è 20 (può essere variata con una system call) e un quanto.

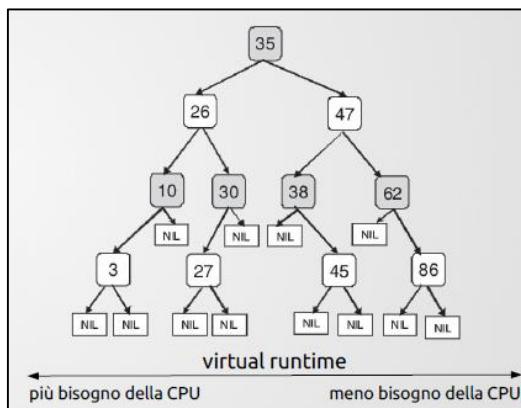
Con lo scheduler O(1) per ogni CPU si ha una runqueue;

- Per ogni priorità: una coda active e una coda expired;
 - Esecuzione del prossimo task a più alta priorità nelle code active fino ad esaurimento del quanto di tempo (i residui non si perdono);
 - Svuotate tutte le code active, c'è uno switch tra active e expired;

I quanti sono lunghi (800ms) per le priorità alte; corti (5ms) per quelle basse; Scalabilità: complessità O(1); Multi-processor/core; Code di attesa e lock.

Lo **scheduler CFS** di Linux implementa l'idea di uno scheduler garantito (virtual runtime)

- Utilizza un albero RB;
 - I timeslice sono dinamici e non predeterminati;
 - Granularità configurabile: bassa (default) (es.desktop); alta: sistemi tipo-batch;
 - Priorità/nice: fattori di decadenza;
 - Schedulazione per gruppo (implementa l'idea del fair-share).



Task su Linux

- concetto universale di **task**
 - astrae il concetto di processo e di thread
 - struttura dati **task_struct**
 - chiamata di sistema **clone**:
 - invocazione: pid = clone (function, stack, sharing flags, args)
 - sharing flags: clone VM / FS / FILES / SIGHAND / PARENT / ...
 - la fork diventa un caso particolare
 - identificativi:
 - **process identifier (PID)**: per retrocompatibilità
 - **task identifier (TID)**: sempre univoco per ogni task

Gestione della memoria

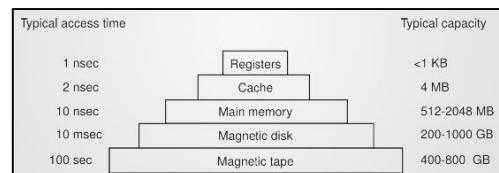
Compiti del gestore della memoria

La memoria di un computer è condivisa da un grande numero di processi, per cui la gestione della memoria è tradizionalmente una parte molto importante di un SO. Le problematiche principali nella gestione della memoria sono l'uso efficiente della memoria, la protezione della memoria allocata a un processo contro gli accessi illegali da parte di altri processi, le prestazioni dei singoli processi e le prestazioni del sistema.

Compito del **gestore della memoria** è quello di tenere conto delle parti di memoria usate e di quelle inutilizzate, di allocare memoria ai processi quando ne hanno bisogno e di deallokarla quando hanno finito, e infine gestire gli scambi fra memoria primaria e secondaria, quando la memoria primaria non è abbastanza sufficiente per contenere tutti i processi.

Componenti della memoria

Come già detto, un calcolatore possiede una **gerarchia di memoria**: registri, cache, memoria RAM, cache del disco, hard disk, supporti di memorizzazione rimovibili, ecc. Lo scopo di una gerarchia di memoria è quello di dare l'illusione di una memoria veloce e grande.



La CPU fa riferimento alla memoria più veloce, la **cache**, quando deve accedere a un'istruzione o a un dato. Se l'istruzione o il dato non è disponibile in cache, viene prelevato dal livello successivo della gerarchia di memoria, che potrebbe essere una cache più lenta oppure la memoria **RAM**. Se l'istruzione o il dato non è disponibile nemmeno al livello successivo della gerarchia, viene prelevato da un livello inferiore e così via.

Per mantenere in memoria un elevato numero di processi, il kernel può decidere di mantenere in memoria anche solo una parte dello spazio di indirizzamento di ogni processo. A tal fine si utilizza la parte della gerarchia della memoria chiamata **memoria virtuale** che si compone della memoria RAM e dell'hard disk. Le parti dello spazio di indirizzamento di un processo non presenti in memoria vengono caricate dal disco quando necessario.

I registri, la memoria cache e la memoria RAM sono **memorie volatili**, cioè non sono utili per memorizzare permanentemente dati e programmi. Per questo scopo si utilizzano i dischi (o i supporti rimovibili).

Allocazione statica e dinamica della memoria (binding)

Un'operazione molto importante nella gestione della memoria è il **binding**, il processo tramite cui si associano gli indirizzi di memoria alle entità di un programma. Esso consiste nello specificare il valore di un attributo. Per esempio, una variabile in un programma ha attributi come il nome, il tipo, la dimensione.

Questa operazione può essere effettuata in 3 momenti diversi:

1. Durante la compilazione
2. Durante il caricamento
3. Durante l'esecuzione

Il momento esatto nel quale viene eseguito il binding può determinare l'efficienza e la flessibilità con cui l'entità del programma può essere utilizzata. In generale possiamo distinguere tra binding statico (*early binding*) e binding dinamico (*late binding*). Il binding statico è un binding eseguito prima dell'esecuzione di un programma (durante la compilazione o durante il caricamento). Il binding dinamico è eseguito durante l'esecuzione di un programma.

Esecuzione dei programmi

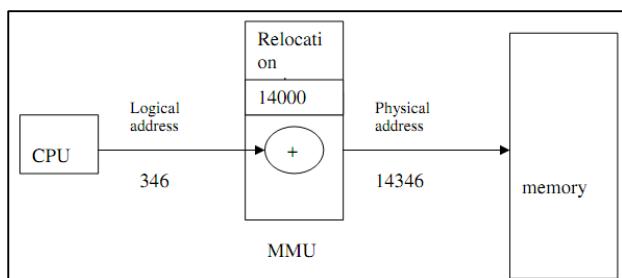
Un programma P scritto in linguaggio L prima di essere eseguito deve subire varie trasformazioni. In particolare deve essere compilato → linkato → eseguito. Ognuna di queste trasformazioni effettua il collegamento delle istruzioni e i dati del programma a un nuovo insieme di indirizzi.

- **Compilazione:** durante la compilazione, le istruzioni del codice sorgente sono tradotte in istruzioni macchina. Viene creato il modulo oggetto.
- **Linking:** durante la fase di linking, il codice delle librerie viene incluso nel modulo oggetto. In pratica, vengono linkati al programma le funzioni che esso utilizzerà. Viene creato il codice binario.
- **Caricamento:** durante il caricamento, il codice binario viene caricato in memoria per poter essere eseguito dalla CPU.

Indirizzi logici e indirizzi fisici

Se gli indirizzi sono generati nelle fasi di compilazione e di caricamento, allora indirizzi logici e indirizzi fisici corrispondono, per cui non è necessario effettuare la traduzione degli indirizzi. Se gli indirizzi sono generati durante l'esecuzione, allora è necessaria la traduzione degli indirizzi. È importante fare la distinzione tra indirizzo logico ed indirizzo fisico.

- Gli **indirizzi logici** sono quegli indirizzi utilizzati dalla CPU; in pratica, il processore fa riferimento alla memoria attraverso gli indirizzi logici.
- Gli **indirizzi fisici** sono quegli indirizzi di memoria dove effettivamente risiede il dato o l'istruzione. L'accesso alla memoria fisica avviene utilizzando gli indirizzi fisici.



La traduzione da indirizzi logici a fisici, attuata durante l'esecuzione dei programmi, viene realizzata in HW dalla MMU attraverso una rilocazione dinamica. I metodi di associazione degli indirizzi nelle fasi di compilazione e di caricamento producono indirizzi logici e fisici identici. Se gli indirizzi sono generati in fase di esecuzione, invece, gli indirizzi logici non coincidono con quelli fisici. In questo caso ci si riferisce agli indirizzi logici col termine di indirizzi virtuali.

Gestione dei vari tipi di programmazione di sistema

Esistono vari tipi di approccio alla gestione della memoria, i più diffusi sono i seguenti.

Monoprogrammazione senza swapping o paginazione

Lo schema di gestione della memoria più semplice possibile è quello di eseguire soltanto un programma alla volta, condividendo la memoria tra il programma ed il SO. Appena l'utente batte il carattere a terminale, il SO carica il programma richiesto dal disco in memoria e lo esegue. I problemi (ovvi) della monoprogrammazione sono che non supporta alcune applicazioni nel quale si rende necessaria la multiprogrammazione e che può supportare utenti multipli solo tramite swapping completo causando carenze nelle prestazioni.

La monoprogrammazione attualmente non viene più utilizzata, a parte alcuni sistemi embedded; la maggior parte dei moderni SO consentono a diversi processi di girare contemporaneamente. In questo caso si parla di multiprogrammazione.

Multiprogrammazione con partizione fisse

La multiprogrammazione consente di avere diversi processi contemporaneamente in memoria; questo comporta che se, ad esempio, un processo è bloccato in attesa di I/O, un altro processo può utilizzare la CPU. Per questo motivo, si incrementano l'efficienza e le prestazioni di una macchina. Inoltre, la multiprogrammazione permette ad un programma di fare uso di due o più processi in modo tale da poter terminare prima le sue operazioni.

Il modo più semplice di realizzare multiprogrammazione è quello di dividere la memoria in n partizioni, magari diverse detto **MFT** (Multiprogramming with a Fixed number of Task, multiprogrammazione con un numero fissato di task). Ogni partizione deve contenere esattamente un processo, quindi il grado di multiprogrammazione è limitato al numero di partizioni. Dal momento che le partizioni sono fisse, tutto lo spazio di una partizione non usato dal processo viene sprecato.

In questo tipo di multiprogrammazione vengono utilizzati due approcci per le code di processi che portano all'allocazione dei processi nelle varie partizioni della memoria.

- **Code separate:** In questo approccio, ciascuna partizione ha una propria coda di ingresso. All'avvio del sistema viene effettuata la suddivisione della memoria in partizioni fisse. Quando arriva un processo, viene messo nella coda di ingresso della partizione più piccola che lo può contenere. Dal momento che le partizioni sono fisse, in questo schema, tutto lo spazio di una partizione non usato dal processo viene sprecato. Lo svantaggio dell'organizzare i processi in ingresso in code separate diventa evidente nel caso in cui la coda per una partizione grande sia vuota ma quella per una partizione piccola sia piena, come nel caso delle partizioni 1 e 3 dell'immagine a sinistra: qui i piccoli processi devono aspettare per essere inseriti in memoria, sebbene la maggior parte della stessa sia libera.

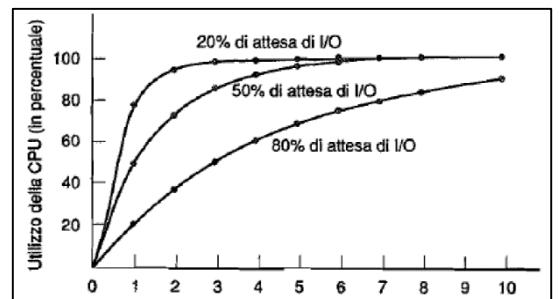
- **Coda unica:** In questo approccio, tutte le partizioni hanno una singola coda di ingresso “comune”. La partizione della memoria viene sempre effettuata all'avvio del sistema. Ogni qualvolta una partizione diventa libera, vi viene caricato il processo più vicino alla testa della coda che può entrare nella partizione, ed esso è quindi mandato in esecuzione. Dal momento che non è desiderabile sprecare una partizione molto grande per far girare un processo piccolo, un'altra strategia è quella di cercare in tutta la coda in ingresso ogni volta che si libera una partizione, e scegliere il job più grande che può entrarci. Questo algoritmo dà poca priorità ai processi piccoli, anche se di solito è preferibile dare a quest'ultimi una priorità alta. Una possibile soluzione è quella di avere sempre a disposizione una partizione piccola che permette ai processi più piccoli di girare senza dover allocare per loro una partizione grande.

Analisi della multiprogrammazione

Usando la multiprogrammazione, si può incrementare l'utilizzo della CPU. Supponiamo che un processo spenda una frazione p del suo tempo in attesa del completamento di operazioni di I/O: con n processi contemporaneamente in memoria, la probabilità che tutti gli n processi stiano aspettando il completamento di un'operazione di ingresso/uscita (nel qual caso la CPU sarebbe inattiva) è p^n . L'utilizzo della CPU è dato dalla formula

$$\text{utilizzo CPU} = 1 - p^n$$

La figura a destra mostra l'utilizzo della CPU come funzione di n , che viene detto **grado di multiprogrammazione**. Dalla figura risulta che se i processi spendono 1'80% del proprio tempo in operazioni di ingresso/uscita, per mantenere lo spreco del tempo di CPU al disotto del 10% bisogna mantenere contemporaneamente in memoria almeno 10 processi. Per maggior precisione e completezza, bisogna sottolineare che il modello probabilistico appena descritto è solo un'approssimazione. Ma con una sola CPU, non possiamo avere tre processi contemporaneamente in esecuzione, per cui un processo che diventa pronto mentre la CPU è occupata dovrà aspettare; quindi i processi non sono indipendenti.



Sebbene il modello di Figura sia semplicistico, può ancora essere usato per fare previsioni specifiche, anche se approssimative, sulle prestazioni della CPU. Supponiamo, ad esempio, che un calcolatore abbia 32 MB di memoria, con un SO che occupi fino a 16MB e con ciascun programma utente che occupi 4 MB: queste dimensioni permettono di avere quattro programmi utente contemporaneamente in memoria. Con un 80 per cento di attesa media per l'I/O, abbiamo un utilizzo della CPU (ignorando il lavoro del SO) del 60 per cento ($1 - 0,8^4$). L'aggiunta di altri 16 MB di memoria permette al sistema di passare da una multiprogrammazione a quattro ad una multiprogrammazione a otto processi, portando la percentuale di utilizzo della CPU all'83 per cento. In altre parole, i 16 MB addizionali di memoria incrementano il throughput (rendimento) del 38 per cento. L'aggiunta di altri 16 MB porterebbe l'utilizzo della CPU solo dall'83 al 93 % migliorando il throughput solo di un altro 12 per cento. Usando questo modello il proprietario del calcolatore potrebbe decidere che la prima aggiunta sia un buon investimento, ma che la seconda non lo sia.

Gestione dell'allocazione

Abbiamo detto che gli indirizzi logici (o virtuali) non corrispondono necessariamente agli indirizzi dove sono effettivamente rese disponibili le informazioni cercate, ossia gli indirizzi fisici, per questo motivo è necessario che sia messo in atto un meccanismo che consenta di mettere in corrispondenza gli indirizzi logici con gli indirizzi fisici. Questo meccanismo è la **rilocazione**.

- La **rilocazione statica** viene eseguita prima che abbia inizio l'esecuzione del programma. Questa politica permette di risparmiare l'overhead dovuto alla traduzione degli indirizzi, ma non è prestante in quanto non consente di cambiare l'area di memoria allocata al programma.
- La **rilocazione dinamica** viene effettuata durante l'esecuzione del programma. Questa politica consente di cambiare l'area di memoria allocata al programma ma soffre di overhead: può essere effettuata sospendendo l'esecuzione del programma, eseguendo la rilocazione e riprendendo l'esecuzione del programma.

Linking

Un **linker** collega insieme i moduli per formare un programma eseguibile. Un **loader** carica un programma o una parte di esso in memoria per l'esecuzione.

- Nel **linking statico** il linker collega tutti i moduli di un programma prima che cominci la sua esecuzione. Se più programmi usano lo stesso modulo di una libreria, ogni programma riceverà una propria copia del modulo, quindi diverse copie del modulo potranno essere presenti in memoria allo stesso tempo se i programmi che usano il modulo vengono eseguiti simultaneamente.
- Il **linking dinamico** viene eseguito durante l'esecuzione di un programma binario. Il linker viene invocato quando, durante l'esecuzione, si incontra un riferimento esterno non assegnato. Il linker collega il riferimento esterno e riprende l'esecuzione del programma.

Assegnazione della memoria

La memoria centrale deve contenere sia il SO sia i vari processi utente, perciò è necessario assegnare le diverse parti della memoria centrale nel modo più efficiente possibile.

Solitamente la memoria centrale si divide in due partizioni, una per il SO e una per i processi utente. Nei moderni SO è richiesto che più processi utente risiedano contemporaneamente in memoria, perciò è necessario considerare come assegnare la memoria disponibile ai processi presenti nella coda di ingresso che attendono di essere caricati in memoria.

Protezione della memoria

Come appena detto, in un sistema multiprogrammato, la memoria principale disponibile è di solito condivisa tra un certo numero di processi. Inoltre esiste una certa difficoltà nel tenere separati i vari processi in memoria. In pratica, occorre assicurarsi che ogni processo abbia uno spazio di memoria separato che non interferisca con gli spazi di memoria degli altri processi. A tal fine occorre determinare l'intervallo degli indirizzi a cui un processo può accedere legalmente, e garantire che possa accedere soltanto a questi indirizzi. La protezione della memoria è implementata mediante due registri di controllo della CPU chiamati registro base e registro limite.

Il registro base contiene l'indirizzo di partenza della memoria allocata a un programma, il registro limite contiene la dimensione della memoria allocata al programma. L'hardware per la protezione della memoria genera un interrupt di violazione di protezione della memoria se un indirizzo di memoria utilizzato nell'istruzione corrente di un processo risiede al di fuori dei limiti degli indirizzi definiti dal contenuto dei registri base e limite.

È il campo del PSW relativo all'informazione di protezione della memoria (MPI) che contiene i registri base e limite. Un processo utente, eseguito con la CPU in modalità utente, non può modificare il contenuto di questi registri poiché le istruzioni per caricare e salvare questi registri sono istruzioni privilegiate.

Problemi di rilocazione e protezione

La multiprogrammazione introduce due problemi essenziali che devono essere risolti: la rilocazione e la protezione.

Quando viene effettuato il link di un programma (il programma principale, le procedure scritte dall'utente e le procedure di libreria vengono combinati in un singolo spazio di indirizzi), il linker deve conoscere l'indirizzo di partenza del programma in memoria.

Per esempio, supponiamo che la prima istruzione sia una chiamata alla procedura che si trova all'indirizzo assoluto 100 nel file binario prodotto dal linker. Se il programma viene caricato nella partizione 1 (all'indirizzo 100K), quell'istruzione salterà all'indirizzo assoluto 100, che sta dentro al SO. Ciò che è necessario è una chiamata all'indirizzo 100K + 100. Se il programma viene caricato nella partizione 2, deve essere eseguito come una chiamata alla locazione 200K+ 100, e così via. Questo problema è noto come il **problema della rilocazione**. Una possibile soluzione è la modifica delle istruzioni quando il programma viene caricato in memoria. I programmi caricati nella partizione 1 aggiungeranno 100K ad ogni indirizzo, i programmi caricati nella partizione 2 aggiungeranno 200K agli indirizzi, e così via. Per effettuare in questo modo la rilocazione durante il caricamento, il linker deve includere nel programma binario una lista, o una mappa di bit (bitmap) che dica quali parole del programma sono indirizzi da rilocare e quali sono codici di operazione, costanti o altre voci che non debbono essere rilocate. La rilocazione durante il caricamento non risolve il problema della protezione; un programma malvagio può sempre costruire una

nuova istruzione e saltarvi. Poiché i programmi in questo sistema usano indirizzi assoluti, invece che indirizzi relativi ad un registro, non c'è nessun modo per impedire ad un programma di costruire un'istruzione che legga o scriva qualunque parola in memoria.

Nei sistemi multiutente non si vuole permettere ai processi di leggere e scrivere la memoria appartenente ad altri utenti. La soluzione che scelse l'IBM per proteggere il 360 fu di dividere la memoria in blocchi di 2KB e di assegnare ad ogni blocco un codice di protezione a 4 bit. La PSW conteneva una chiave a 4 bit; l'hardware del 360 individuava qualunque tentativo da parte di un processo in esecuzione di accedere alla memoria il cui codice di protezione differiva dalla chiave contenuta nella PSW. Poiché soltanto il SO poteva cambiare i codici di protezione e la chiave, ai processi utente veniva impedito di interferire fra di loro e con il SO stesso.

Una soluzione alternativa ad entrambi i problemi della rilocazione e della protezione, è quella di dotare la macchina di due registri hardware speciali, chiamati registro base e limite. Quando un processo viene schedulato, nel registro base viene caricato l'indirizzo di inizio della sua partizione, e nel registro limite viene caricata la lunghezza della partizione. Ad ogni indirizzo di memoria generato viene automaticamente sommato il contenuto del registro base, prima di inviarlo alla memoria; quindi, se il registro base è 100K, un'istruzione CALL 100 viene in realtà convertita in una istruzione CALL 100K+ 100, senza modificare l'istruzione stessa. Gli indirizzi vengono anche confrontati con il registro limite, per assicurarsi che non tentino di indirizzare al di fuori della partizione corrente; l'hardware protegge i registri base e limite per impedire ai programmi utente di modificarli. Uno svantaggio di questo schema è dato dalla necessità di eseguire una somma ed un confronto ad ogni riferimento in memoria.

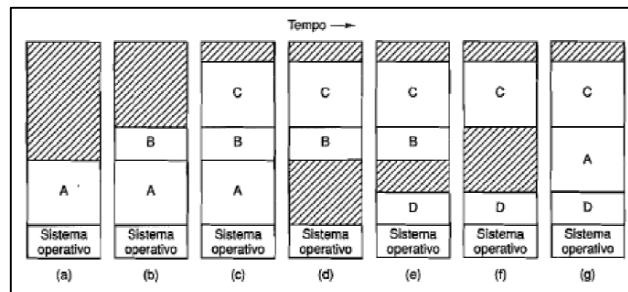
Swapping

Con i sistemi batch, organizzare la memoria in partizioni fisse è facile ed efficiente. Ogni job, quando raggiunge la testa della coda, viene caricato in una partizione e rimane in memoria finché non ha terminato la sua esecuzione.

Con i personal computer orientati alla grafica, i sistemi multiutente e multiprogrammati, la situazione è diversa; a volte non vi è abbastanza memoria principale per mantenere tutti i processi correntemente attivi. Occorre, quindi, trasferire alcuni dei processi in eccesso dalla memoria sul disco, per poi introdurli in memoria successivamente. Possono essere utilizzati due approcci generali di gestione della memoria a seconda dell'hw disponibile:

- **Swapping:** carica interamente ogni processo in memoria, lo esegue e dopo un tot di tempo lo scarica su disco.
- **Memoria virtuale:** consente ai programmi di girare anche quando sono caricati solo parzialmente nella memoria principale.

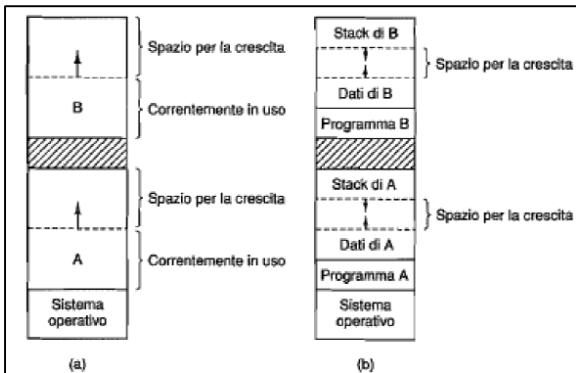
Per quanto riguarda lo swapping, inizialmente il processo A risiede in memoria, poi vengono caricati B e C. Nella figura a destra, il processo A viene scaricato su disco, poi arriva C e B viene scaricato su disco; infine ritorna A. poiché ora A è inserito in una locazione diversa, gli indirizzi contenuti in esso devono essere rilocati dal sw nel momento in cui viene caricato, oppure dall'hw durante l'esecuzione del programma.



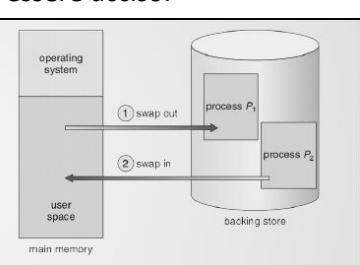
La differenza principale fra le **partizioni fisse** e le **partizioni variabili** è che il numero, la dimensione e la posizione delle partizioni variano dinamicamente nel secondo caso con l'arrivo e la partenza dei processi, mentre nel primo caso sono fisse. La flessibilità data dal non essere legati ad un numero di partizioni che possono essere troppo grandi o troppo piccole migliora l'utilizzo della memoria.

Quando lo swapping crea molti buchi in memoria, è possibile combinarli in un unico grande buco, spostando tutti i processi il più indietro possibile: questa tecnica è nota come **compattazione della memoria** e normalmente non viene eseguita perché richiede troppo tempo alla CPU. Se i processi vengono creati con una dimensione che non cambia mai, allora, l'allocazione è semplice: si alloca esattamente la quantità di memoria necessaria. Se, invece, i segmenti dati dei processi possono crescere, allora si possono avere dei problemi quando il processo tenta di espandersi. Se vicino ad esso si trova un buco, esso può essere allocato al processo; se, invece, il processo è vicino ad un altro processo, quello in crescita deve essere spostato in un buco di memoria abbastanza grande per contenerlo, oppure occorre scaricare uno o più processi sul disco. Se un processo non può crescere o l'area di swap è piena,

allora il processo deve attendere o essere ucciso. Se ci si aspetta che i processi crescano, allora è utile allocare sempre una parte di memoria extra.



In Figura(a) a sinistra vediamo una configurazione di memoria nella quale è stato allocato spazio per la crescita a due processi. Se i processi avessero due segmenti che crescono, per esempio, il segmento di dati, utilizzato come heap per le variabili che sono allocate e rilasciate dinamicamente, ed il segmento stack per le variabili locali normali e gli indirizzi di ritorno, è evidente che si potrebbe utilizzare uno schema alternativo come quello di Figura(b). In questa figura vediamo che ciascun processo ha uno stack che cresce verso il basso, posto in cima alla memoria che gli è stata assegnata, e un segmento di dati posto proprio sopra il testo del programma, che cresce verso l'alto. La memoria che si trova fra i due segmenti può essere usata dall'uno o dall'altro; se si esaurisce, il processo deve essere spostato in un buco abbastanza grande, oppure scaricato sul disco finché non si crea un buco abbastanza grande, oppure deve essere ucciso.



Le operazioni svolte sono in genere:

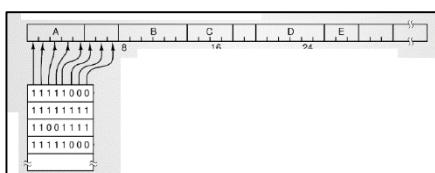
- **Swap-in**, quando si porta un processo dal disco in memoria
- **Swap-out**, quando si porta un processo dalla memoria al disco

Concettualmente lo swap space non è diverso dalla RAM, ma sta su disco (quindi le unità di allocazione sono blocchi, non byte). Per il resto, le tecniche di gestione per RAM sono valide anche per lo swap space. Con due varianti: swap space fisso, allocato alla nascita del processo, usato per tutta la durata del processo e swap space nuovo, allocato a ogni swap-out. In entrambi i casi l'allocazione dello swap space (unica o ripetuta) può sfruttare tecniche e algoritmi per RAM.

Strategie di allocazione

Quando la memoria viene assegnata dinamicamente, il SO deve gestirla. Per decidere la partizione in cui un processo sarà caricato, esistono diverse strategie:

- **Bitmap**: utilizzando una mappa di bit, la memoria viene divisa in unità di allocazione che possono essere lunghe solo poche parole o arrivare a qualche kilobyte; a ciascuna delle unità di allocazione viene associato un bit della mappa, che vale 0 se l'unità è libera e 1 se è occupata. Le dimensioni delle unità di allocazione rappresentano una scelta di progettazione importante. Più piccola è l'unità di allocazione più grande è la mappa di bit; tuttavia anche con un'unità di allocazione di soli 4 byte, 32 bit in memoria richiedono solo 1 bit nella mappa di bit. Una mappa di bit di $32n$ bit userà n bit nella mappa di bit, così la mappa di bit prenderà solo $1/32$ della memoria. Se si sceglie un'unità di allocazione grande, allora la mappa di bit è piccola, ma viene sprecata una quantità significativa di memoria nell'ultima unità quando la dimensione dei processi non è un multiplo esatto dell'unità di allocazione.



Una mappa di bit mette a disposizione una maniera semplice per tenere conto dello stato delle parole di memoria usando una quantità di memoria fissa; infatti la dimensione della mappa di bit dipende solo dalla dimensione della memoria fisica e dalla dimensione dell'unità di allocazione. Il problema principale, quando si utilizza l'allocazione

contigua, sta nel fatto che quando si decide di caricare un processo di k unità il gestore della memoria deve esaminare la mappa di bit per trovare una sequenza consecutiva di k zeri consecutivi. Questa è un'operazione lenta per cui le bitmap sono poco utilizzate.

- **Liste concatenate**: Una soluzione che non richiede un impiego considerevole di memoria è mantenere una lista concatenata dei segmenti di memoria liberi e occupati. Così facendo è sufficiente mantenere in memoria il puntatore al primo blocco della lista per essere in grado di reperire un blocco libero all'occorrenza. Lo svantaggio è che se si rende necessario attraversare la lista occorre leggere ogni singolo blocco, degradando le prestazioni del



sistema in modo considerevole. Ciascun elemento nella lista specifica ha un buco (H) o un processo (P), l'indirizzo di partenza, la lunghezza ed un puntatore all'elemento successivo della lista. Ordinare una lista per indirizzi ha il vantaggio che quando un processo termina o viene scaricato, l'aggiornamento della lista è molto semplice. Un processo che termina ha normalmente due vicini (a meno che non sia l'ultimo o il primo elemento della lista); questi possono essere P o H.

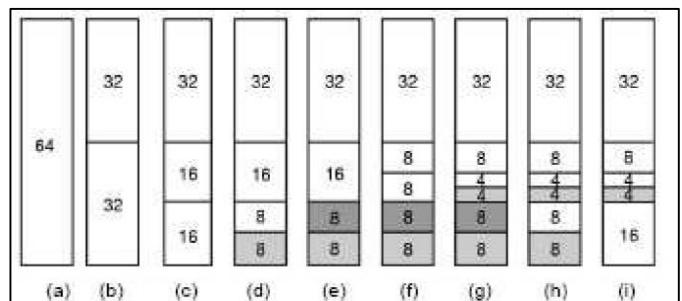
Dal momento che l'elemento della tabella dei processi relativo al processo che termina contiene il puntatore all'elemento della lista che contiene tale processo, risulta più conveniente mantenere una lista doppia. Quando i processi e i buchi vengono mantenuti in una lista concatenata ordinata per indirizzo, si possono usare diversi algoritmi per allocare la memoria ad un processo appena creato o ricaricato dal disco. Gli algoritmi più utilizzati sono:

- **First fit:** Viene utilizzato quando il gestore sa quanta memoria allocare. Il gestore scorre tutta la lista fino a che non trova un buco che sia abbastanza grande; il buco viene quindi diviso in pezzi, uno per il processo e uno per la memoria che non viene usata.
- **Next fit:** Lavora nella stessa maniera del first fit, tranne per il fatto che si ricorda di dove aveva trovato un buco adatto; la prossima volta che viene chiamato riparte da lì.
- **Best fit:** Cerca in tutta la lista e prende il più piccolo dei buchi che possono essere usati. Anziché spezzare un buco grosso che potrebbe essere usato più tardi, il best fit cerca di trovare un buco le cui dimensioni siano vicine a quelle necessarie. Il best fit è più lento del first fit, perché ogni volta scorre tutta la lista ed inoltre dà luogo ad un maggiore spreco di memoria rispetto al first fit e al next fit, poiché tende a riempire la memoria di piccoli buchi.
- **Worst fit:** Per evitare che dopo la divisione di un buco per collocarvi il processo la parte rimanente sia di dimensioni troppo piccole, si potrebbe utilizzare questo algoritmo. Esso prende sempre il buco più grande disponibile, cosicché il buco generato possa avere dimensione abbastanza grande da risultare utile. Però anche questo algoritmo non è buono.

Tutti e quattro gli algoritmi possono essere resi più veloci mantenendo liste separate per H e P. il prezzo pagato per accelerare l'allocazione è la complessità aggiuntiva ed il rallentamento che si hanno quando un blocco di memoria viene deallocated, dal momento che il segmento rilasciato deve essere rimosso dalla lista dei processi ed inserito in quella dei buchi. Se si mantengono liste separate per P e H, la lista degli H può essere ordinata per dimensione, per rendere più veloce il best fit. Quando il best fit esegue la ricerca di un buco partendo da quelli di dimensione più piccole, non appena trova un buco sufficiente, sa che quello è il buco di dimensione più piccola che fa allo scopo. Con la lista dei buchi ordinata per dimensione, best fit e first fit sono ugualmente veloci, mentre il next fit non ha alcun senso.

- **Buddy system:** Questa tecnica di allocazione dinamica della memoria divide la memoria in partizioni per soddisfare una richiesta di memoria nel miglior modo possibile. Questo sistema suddivide ricorsivamente la memoria in due metà finché il blocco ottenuto è grande appena a sufficienza per l'uso, cioè quando un'ulteriore divisione lo renderebbe più piccolo della dimensione richiesta.

L'idea di base per la gestione della memoria è la seguente: inizialmente la memoria si compone di un solo pezzo contiguo, 64 pagine nell'esempio a destra. Al momento di una richiesta di memoria, questa viene arrotondata ad una potenza di due, ad esempio 8 pagine. L'intero pezzo di memoria viene diviso in due come mostrato in (b); se ogni pezzo risulta ancora troppo grande, il pezzo più basso di memoria viene diviso nuovamente in due (c) ed ancora (d). Nell'esempio si è ottenuto un pezzo di dimensione corretta, che viene allocato per il chiamate, come mostrato dal rettangolo ombreggiato in (d). Si supponga adesso che una nuova richiesta di 8 pagine arrivi al sistema: questa può essere soddisfatta immediatamente (e). A questo punto arriva una terza richiesta di 4 pagine: il pezzo più piccolo disponibile viene diviso (f) e la metà più bassa viene concessa (g). Al passo successivo il secondo pezzo di 8 pagine viene rilasciato (h), così come al passo successivo avviene per un altro pezzo di 8 pagine. Poiché due pezzi da 8 pagine, contigui, risultano liberi, essi possono essere uniti



nuovamente per diventare un unico blocco da 16 pagine (i). Buddy system è alla base della gestione della memoria di Linux.

Pro:

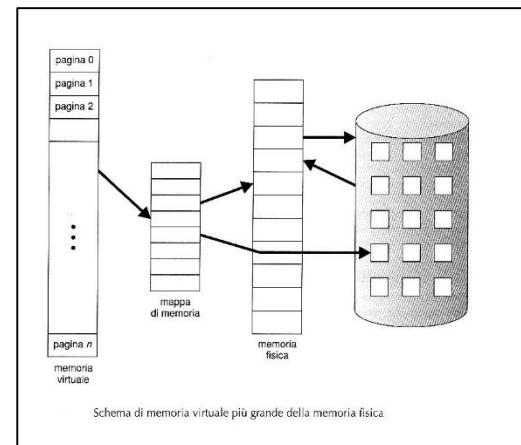
- Risparmio memoria: ogni hole contiene record lista hole
- Allocazione veloce: come in Quick Fit, se non ci vogliono split
- Deallocazione blocco 2k veloce: dall'indirizzo del blocco liberato, si calcola il buddy in O(1), da used bit si vede se anche buddy è un hole, se non lo è, non c'è altro da fare, se no, si fonde (merge) il buddy con quello liberato (per facilitare il merge, le liste di hole saranno doubly linked).

Contro:

- Bad cases (ma peggiorano poco il caso medio): molti split o merge consecutivi
- Frammentazione interna: per allocare 65K serve blocco da 128K quindi spreco "interno" al blocco: 63K (circa pari alla richiesta di 65K).

Memoria virtuale

È un'architettura di sistema capace di simulare uno spazio di memoria primaria maggiore di quello fisicamente presente o disponibile; questo risultato si raggiunge utilizzando spazio di memoria secondaria su altri dispositivi o supporti di memorizzazione, di solito le unità a disco. La memoria centrale fisicamente presente diventa quindi la parte effettivamente utilizzata di quella virtuale, più grande.



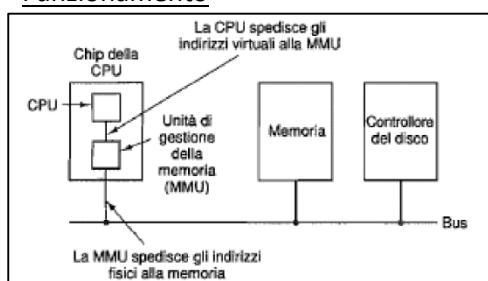
Divisione in overlay

In passato ci si dovette confrontare per la prima volta con programmi che erano troppo grandi per poter essere caricati in memoria centrale; la soluzione generalmente adottata era quella di dividere il programma in parti, dette **overlay**. L'overlay 0 cominciava a girare per primo, e quando aveva finito chiamava un altro overlay. Alcuni sistemi ad overlay erano molto complessi, e permettevano la coesistenza in memoria di più overlay alla volta. Gli overlay venivano mantenuti sul disco e caricati nella memoria dal SO, dinamicamente, quando richiesto. Sebbene il lavoro necessario al caricamento e scaricamento venisse in realtà svolto dal SO, il compito di dividere il programma in parti era del programmatore.

Idea della memoria virtuale

Dividere i grossi programmi in piccole parti modulari era un'attività che portava via tempo. Non ci volle molto prima che qualcuno pensasse ad un modo per delegare al calcolatore l'intero lavoro. Il metodo che fu trovato (Fotheringham, 1961) divenne noto con il nome di **memoria virtuale**. L'idea di base che sta dietro alla memoria virtuale è che la dimensione combinata di programma, dati e stack può eccedere la dimensione della memoria fisica per essi disponibile. Il SO mantiene in memoria le parti che sono in uso in un certo momento, mentre le altre parti vengono mantenute sul disco. Per esempio, un programma da 16MB può girare in una macchina da 4 MB scegliendo in maniera accurata quali 4 MB mantenere in memoria ad ogni istante, con pezzi di programma che vengono caricati e scaricati fra disco e memoria secondo le necessità.

Funzionamento



Quando si usa la memoria virtuale, gli indirizzi virtuali non finiscono direttamente sul bus della memoria. Invece, vanno a finire ad una unità della gestione della memoria (**MMU**, Memory Management Unit), che mappa gli indirizzi virtuali sugli indirizzi della memoria fisica come in figura. Volendo dare dunque una definizione ultima della memoria virtuale possiamo affermare che la memoria virtuale è una caratteristica dell'architettura di un calcolatore che si distingue da quella reale dal fatto che i suoi indirizzamenti sono virtuali.

Indirizzi fisici e virtuali

Degli indirizzi si dicono **fisici** se hanno un unico spazio di indirizzi individuato dall'interfaccia fisica CPU-memoria. In altre parole sono quelli che la macchina mette sul bus degli indirizzi. Degli indirizzi si dicono **virtuali** se hanno spazi di indirizzi visti dal software cioè individuati dal formato di indirizzo che usano le istruzioni.

Traduzione

Osservando quindi un quadro completo della situazione osserviamo che l'implementazione della memoria virtuale coinvolge l'hardware (CPU ed MMU) ed il software (che modifica ed utilizza i vari indirizzi virtuali). La traduzione realizzata dalla MMU è in genere definita dal contenuto di opportuni registri/tabelle, è parziale in quanto non vi è un unico generico modo per tradurre gli indirizzi e modificabile dal software reimpostando registri e tabelle. Se per un indirizzo virtuale non è definita la funzione di traduzione (non esiste riferimento alla RAM) la **CPU causa un'eccezione (trap)** hardware che viene gestita dal SO che corregge la funzione di traduzione comportando uno scambio di informazioni tra disco e RAM l'indirizzo virtuale non in RAM viene mappato su un indirizzo fisico in RAM e il suo contenuto viene copiato all'indirizzo fisico scelto, in seguito, il contenuto dell'indirizzo fisico (se presente) viene salvato sul disco.

Pro e contro

La memoria virtuale ha dei vantaggi quali:

- Permette codice, dati, stack maggiori in RAM;
- Si combina bene con il multiprogramming (genera I/O da/a disco);
- Ogni processo può avere una memoria virtuale privata (ottenendo così protezione e rilocazione);
- Ogni processo può godere di spazi maggiori anche della ram;

Ma ha anche degli svantaggi quali:

- Costo I/O da/a disco;
- Supporto hardware MMU necessario;

Consideriamo un sistema che fa uso di memoria virtuale con le seguenti caratteristiche: uno spazio di indirizzamento virtuale da 1 GB, un numero di pagina virtuale a 22 bit e un indirizzo fisico a 20 bit. Determinare esattamente quanti frame fisici ci sono in memoria.

Spazio di indirizzamento virtuale $1\text{ GB} = 1024^3\text{ byte} = 1073741824\text{ byte}$

Con un numero di pagina virtuale a 22 bit possiamo individuare un numero di pagine pari a: $2^{22}\text{ pagine} = 4194304$

Adesso possiamo calcolare la dimensione di una pagina con $\frac{\text{dimensione indirizzo virtuale}}{\text{numero di pagine}}$ ovvero:

$$\text{dimensione di una pagina} = \frac{1073741824\text{ byte}}{4194304\text{ pagine}} = 256\text{ byte}$$

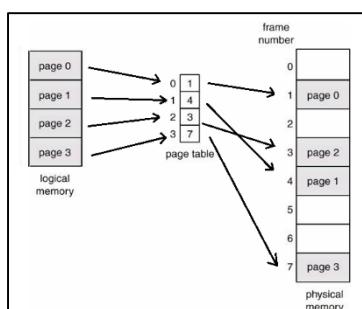
L'indirizzamento fisico è a 20 bit quindi la dimensione totale della RAM è di $2^{20} = 1048576\text{ byte}$

Sapendo che la dimensione di una pagina è 256 byte il numero di frame in memoria RAM è:

$$\frac{1048576}{256} = 4096\text{ frame}$$

Paginazione

Vi sono differenti modalità per ottenere la protezione della memoria. La **segmentazione** e la **paginazione** della memoria sono i metodi più comuni. Queste due tecniche di protezione della memoria impediscono ad un processo di corrompere la memoria di un altro processo in esecuzione contemporaneamente sullo stesso computer. Richiedono usualmente il supporto dell'hardware (ad esempio con una MMU, Memory management unit) e un SO che allochi aree distinte di memoria per i differenti processi e che gestisca l'eventualità nella quale un processo tenti di accedere ad un'area di memoria che non gli compete.



Cos'è la paginazione

Poiché i programmi raramente utilizzano contemporaneamente tutte le parti del proprio codice e dei propri dati, è conveniente implementare la **paginazione**, una tecnica con la quale il SO crea uno spazio di memoria virtuale usando frammenti (o blocchi) di memoria fisica dividendola in piccole porzioni (usualmente di 4 kbytes) chiamate **pagine**.

Ogni pagina può puntare ad un qualsiasi blocco di memoria fisica, e possono anche esserci differenti pagine che puntano allo stesso blocco di memoria fisica. Ad ogni

processo viene assegnata una tabella delle pagine, che specifica a quali pagine il processo ha diritto di accedere. Inoltre, il processo si comporterà come se fosse l'unico processo presente nel sistema e come se disponesse di tutto lo spazio di indirizzamento.

La paginazione permette anche di allocare facilmente nuova memoria per il processo, perché le pagine possono essere mappate ovunque. Parte della memoria di un processo può essere depositata su memoria di massa (**swapping**).

Le pagine (virtuali) e le pagine fisiche hanno sempre la stessa dimensione. Quando un programma tenta di accedere a un indirizzo virtuale, questo viene mandato alla MMU, che secondo la sua funzione di traduzione, ricava l'indirizzo fisico.

Frame

La memoria fisica può memorizzare un numero intero di pagine e viene partizionata in aree o blocchi di memoria, dette **frame**, che hanno la stessa dimensione di una pagina. Quindi, ogni area di memoria (frame) è esattamente della stessa dimensione della pagina, per cui non si crea frammentazione esterna nel sistema. La frammentazione interna può crearsi poiché all'ultima pagina di un processo viene allocato un frame della dimensione di una pagina anche se è più piccolo della dimensione di una pagina

Page fault

Se un programma tenta di utilizzare una pagina non mappata, la MMU si accorge e fa in modo che la CPU esegua un trap al SO; questo passaggio si chiama **page fault**. Il page fault è un'eccezione di tipo trap, generata quando un processo cerca di accedere ad una pagina che è mappata nello spazio di indirizzamento virtuale, ma che non è presente nella memoria fisica, poiché mai stata caricata in quest'ultima o perché, precedentemente, spostata su disco di archiviazione. Tipicamente, il SO tenta di risolvere il page fault rendendo accessibile la pagina richiesta in una locazione della memoria fisica oppure terminando il processo in caso di accesso illegale.

Un page fault è molto sconveniente, poiché la pagina che non viene trovata nella memoria fisica deve successivamente essere cercata (con gli opportuni controlli) e caricata in essa: ciò implica un accesso alla memoria di massa (comunemente un hard disk), che è molto costoso in termini di tempo. Inoltre, un numero eccessivo di pagine mancanti può comportare fenomeni di **thrashing**, ossia di paginazione degenere, con conseguente degradazione delle prestazioni del SO.

Dopo il verificarsi di un page fault, vengono eseguite le seguenti operazioni:

- **Controllo della tabella interna del processo** (solitamente salvata con il *process control block*): se l'accesso alla memoria era illegale, il processo viene arrestato, altrimenti si provvede al caricamento della pagina richiesta, ricercandola nella memoria di massa.
- **Ricerca di un frame libero e caricamento della pagina nella memoria fisica**: se ci sono frames liberi nella memoria fisica (nella quale sono presenti le pagine utilizzate dai processi), la pagina mancante viene immediatamente copiata in essa, e la tabella delle pagine aggiornata; in caso contrario, il SO esegue un **algoritmo di sostituzione delle pagine**, che sceglie la pagina da rimpiazzare ("vittima") secondo determinate politiche di sostituzione e la scambia con la pagina richiesta tramite swap (in pratica, la pagina da rimpiazzare viene copiata sulla memoria di massa, mentre la pagina cercata viene caricata in memoria). Vi sono vari algoritmi di sostituzione delle pagine che vedremo più avanti
- **Riavvio** dell'istruzione che era stata interrotta a causa del page fault.

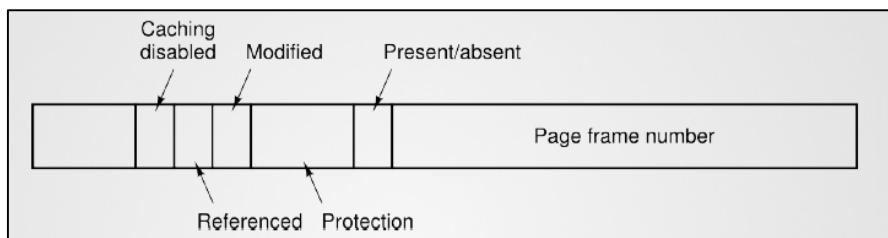
Tabelle delle pagine

Lo scopo della tabella delle pagine è quello di mappare pagine virtuali in pagine fisiche. Da un punto di vista matematico, la tabella delle pagine è una funzione, con il numero di pagina virtuale come argomento e il numero di pagina fisica come risultato. Usando il risultato di questa funzione, il campo che denota la pagina virtuale in un indirizzo virtuale può essere sostituito dal numero della pagina fisica, in modo da formare un indirizzo di memoria fisica. Nonostante questa descrizione semplice, occorre affrontare due problemi principali:

1. La tabella delle pagine può essere molto grande (soprattutto nei calcolatori moderni a 32 o 64 bit)
2. La traduzione deve essere veloce (perché se non lo fosse si annullerebbero i vantaggi di una memoria virtuale creando un collo di bottiglia).

Lo schema più semplice per risolvere i problemi 1 e 2 consiste nell'avere una sola tabella delle pagine realizzata tramite un vettore di registri HW molto veloci, con un elemento per ognuna delle pagine virtuali indirizzato con il numero di pagina virtuale. Quando viene fatto partire un processo, il SO carica i registri con la tabella delle pagine del processo, presa da una copia tenuta nella memoria principale; durante l'esecuzione del processo, non sono più necessari altri riferimenti alla memoria relativi alla tabella delle pagine. I vantaggi di questo metodo consistono nel fatto che è semplice e che non richiede riferimenti alla memoria durante la traduzione; uno svantaggio sta nel fatto che può risultare molto dispendioso (se la tabella delle pagine è grande): il fatto stesso di dover caricare la tabella delle pagine ad ogni cambio di contesto può degradare le prestazioni. All'altro estremo, la tabella delle pagine può risiedere interamente in memoria. Tutto ciò di cui ha bisogno l'hardware è un solo registro che punti all'inizio della tabella in memoria principale. Questo schema permette di cambiare la tabella delle pagine ad ogni cambio di contesto tramite il caricamento di un solo registro; naturalmente ha lo svantaggio di richiedere uno o più riferimenti alla memoria per leggere la tabella delle pagine durante l'esecuzione di ogni istruzione.

Dettaglio su una voce della tabella delle pagine



L'esatta organizzazione di un elemento della tabella delle pagine dipende in maniera pesante dalla macchina, ma il tipo di informazioni presenti è pressappoco lo stesso su tutte le macchine. Le dimensioni variano da calcolatore a calcolatore, ma di solito si usano 32 bit. I campi di una tabella delle pagine sono:

- Il campo più importante è il **numero di pagina fisica**; dopo tutto, lo scopo ultimo della tabella delle pagine è quello di trovare questo numero.
- Poi abbiamo il **bit presente/assente**. Se questo è 1, l'elemento è un elemento valido e può essere usato. Se vale 0, la pagina virtuale cui corrisponde l'elemento non è attualmente presente in memoria. L'accesso di un elemento della tabella delle pagine con questo bit a 0 provoca un fault di pagina.
- I **bit protezione** ci dicono quali tipi di accesso sono permessi. Nella sua forma più semplice questo campo è da un bit, con 0 che indica il diritto di lettura/scrittura e 1 che indica il diritto di sola lettura. Una soluzione più raffinata prevede l'uso di tre bit, ciascuno dei quali abilita o disabilita la lettura la scrittura e l'esecuzione della pagina.
- I **bit modificata/usata** mantengono traccia dell'uso della pagina, se la pagina è stata modificata (cioè è sporca) deve essere di nuovo scritta sul disco, altrimenti (se è pulita) può essere semplicemente abbandonata. Il bit usata viene messo ad 1 ogni qualvolta che si fa riferimento alla pagina e serve ad aiutare il SO a scegliere una pagina da scaricare quando si verifica un fault di pagina.
- Infine l'ultimo bit permette di **disabilitare il caching** della pagina; questa caratteristica è importante per le pagine che vengono mappate su registri di dispositivi anziché sulla memoria.
- Nella pratica esiste anche un **bit di validità** (o di allocazione).

Bit di validità

Supponiamo che la CPU voglia usare un certo dato o una certa istruzione. La MMU, nell'effettuare la traduzione degli indirizzi verifica, facendo uso della tabella delle pagine, se quella pagina è presente o meno in memoria (fa riferimento al campo bit di validità).

Si controlla il numero di pagina, il quale viene usato come indice nella tabella delle pagine, che contiene il numero della pagina fisica corrispondente alla pagina virtuale. Se il bit presente/assente è a 0, si provoca una trap al SO; in dettaglio, la MMU controlla il bit di validità della tabella delle pagine; se è 0 vuol dire che la pagina non è presente in memoria, dunque la MMU genera un interrupt chiamato page fault, che è un interrupt di programma. A causa dell'interrupt viene invocato il gestore della memoria virtuale che carica la pagina in memoria e aggiorna la tabella delle pagine di quel processo.

Il bit di validità è usato proprio per indicare se la pagina è presente o meno in memoria. Si ha page fault quando un processo tenta di usare una pagina non mappata, cioè non presente in memoria. In pratica, MMU e gestore della memoria virtuale interagiscono per decidere quando una pagina di un processo deve essere caricata in memoria.

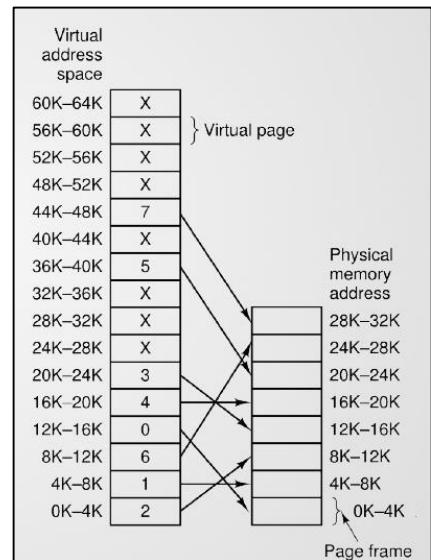
Se invece il bit è a 1, il numero di pagina fisica trovato nella tabella delle pagine viene copiato nei 3 bit di ordine più alto del registro di uscita, insieme ai 12 bit dell'offset, che vengono copiati senza modifiche dall'indirizzo virtuale. Insieme, essi formano un indirizzo fisico di 15 bit; il registro di uscita viene quindi posto sul bus della memoria come indirizzo di memoria fisica.

Esempio di traduzione degli indirizzi virtuali in fisici

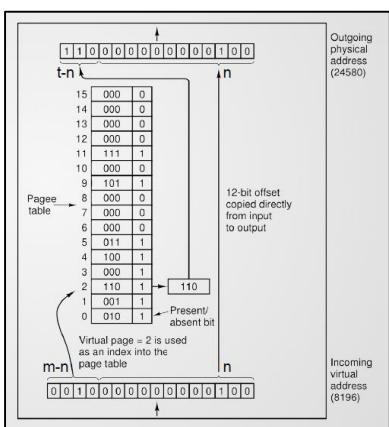
Dalla mappa della MMU nella figura a destra scopriamo di avere una memoria di 64k di spazio di indirizzamento virtuale diviso in pagine da 4k. I numeri contenuti nei rettangoli nelle pagine sono gli indici dei corrispettivi blocchi della memoria fisica.

Ipotizziamo di tradurre l'indirizzo virtuale 8196 (in binario 001000000000100) in indirizzo fisico. Dobbiamo trovare:

- Lo **spazio di indirizzamento virtuale**, dato da 2^m ; dato che abbiamo detto che è 64k, sarà quindi $2^{16} \cong 64k$, quindi $m = 16$
- La **dimensione della pagina**, data da 2^n ; dato che abbiamo detto che è 4k, sarà quindi $2^{12} \cong 4k$, quindi $n = 12$
- **Numero totale di pagine** della tabella delle pagine è 2^{m-n} ; quindi $2^{16-12} = 2^4 = 16$
- Lo **spazio degli indirizzi fisici** è dato da 2^t , il quale è possibile trovare grazie al **numero di frame** dato da 2^{t-n} , $m > t$



Guardiamo quindi la tabella delle pagine di tale esempio:



Troviamo adesso l'indice del **numero di pagina** dell'indirizzo virtuale in input dato dai bit significativi dell'indirizzo virtuale, i quali sono dati da $m - n = 4$ bit; essi sono i bit più a sinistra dell'indirizzo virtuale in input, nell'esempio 0010, il cui risultato in decimale è 2, ovvero il blocco di memoria in cui ci sarà il bit significativo di output, nell'esempio 110, il quale lo scriviamo nell'output.

Adesso troviamo l'offset, ovvero i bit meno significativi, che saranno i 2^n successivi ai bit significativi dopo $m - n$, ovvero nell'esempio i 12 numeri dopo i primi 4, i quali andranno copiati così come sono nell'output.

Troviamo quindi l'indirizzo fisico 110000000000100 = 24580 che è la traduzione di quello di input.

Si supponga che una macchina abbia indirizzi virtuali a 48 bit e indirizzi fisici a 32 bit. (a) Se le pagine sono di 4 KB, quante voci ci sarebbero nella tabella delle pagine se fosse a un solo livello? Si dia una spiegazione. (b) Si supponga che lo stesso sistema abbia un TLB con 32 voci. Inoltre, si assuma che un programma contenga istruzioni che stanno esattamente in una pagina e legga sequenzialmente elementi interi lunghi da un array che si estende su migliaia di pagine. Quanto è indicato il TLB in questo caso?

- (a) Se le pagine sono di 4 KB, servono 12 bit per rappresentare l'offset all'interno di una di esse ($2^{12} = 4096$). Quindi restano $48-12=36$ bit per il numero di pagina virtuale: conseguentemente ci saranno $2^{36} = 64G$ entry nella tabella delle pagine nel caso di una paginazione ad un livello.
- (b) È poco indicato perché la lettura (sequenziale) degli elementi dell'array coinvolge più di 32 voci e quindi si avranno continui fallimenti accedendo al TLB, rendendo di fatto inutile la presenza della memoria associativa se non per la singola voce facente riferimento alla pagina delle istruzioni.

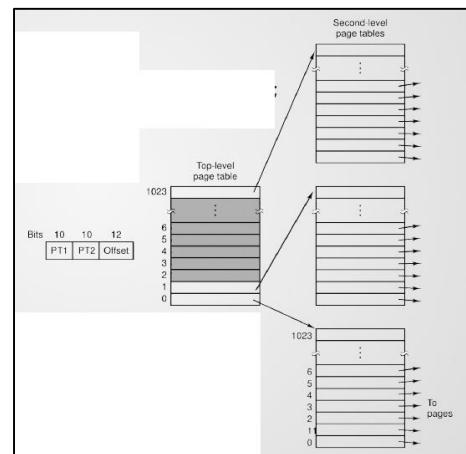
Calcolare il numero di pagine virtuali e offset per una pagina a 4KB e una a 8KB, per ognuno dei seguenti indirizzi virtuali decimali: 20000, 32768 e 60000

La formula è: (indirizzo virtuale)/(4KB*1024)=4,88 → il valore intero "4" è il numero di bit e $0,88 * 4096 =$ l'offset

Tabella delle pagine multilivello

Per risolvere il problema derivante dalla presenza costante in memoria di tabelle delle pagine molto grosse, molti calcolatori usano una tabella delle pagine a più livelli, le quali non mantengono l'intera tabella in memoria, ma fanno una paginazione gerarchica.

La tabella delle pagine multilivello consiste nel suddividere una tabella delle pagine, di grosse dimensioni, in più livelli, evitando di tenere tutti i livelli in memoria per tutto il tempo. In pratica, in questo modo, si effettua una paginazione della tabella dei processi: una tabella delle pagine di alto livello viene utilizzata per accedere alle varie pagine della tabella delle pagine. Se la tabella delle pagine di alto livello è grande, potrebbe essere essa stessa paginata e così via.



Un semplice esempio è mostrato in figura a destra. Abbiamo un indirizzo virtuale da 32 bit che viene diviso in un campo PT1 da 10 bit, un campo PT2 da 10 bit e un campo Offset da 12 bit. Dal momento che gli offset sono da 12 bit, le pagine sono da 4KB e ce ne sono un totale di 2^{20} . Il segreto del metodo delle tabelle a più livelli è di evitare di tenere sempre tutte le tabelle delle pagine in memoria. In particolare, quelle che non risultano necessarie non devono essere tenute in giro. Supponiamo, per esempio, che un processo necessiti di 12 MB, i 4 MB più bassi per il programma, i 4 MB successivi per i dati e gli ultimi 4 MB per la pila. Fra la cima dei dati e il fondo dello stack c'è un gigantesco buco che non viene usato.

Tabella delle pagine invertite

Le tabelle delle pagine tradizionali del tipo descritto precedentemente richiedono un elemento per ogni pagina virtuale, in quanto sono indicizzate da un numero di pagina virtuale.

Tuttavia, visto che i computer da 64 bit diventano sempre più comuni, la situazione cambia drasticamente. Se lo spazio di indirizzamento è ora di 2 byte, con pagine di 4KB, abbiamo bisogno di una tabella delle pagine di 252 elementi, e con 8 byte per ciascun elemento, la tabella risulterà essere di oltre 30 milioni di gigabyte; logicamente utilizzare 30 milioni di gigabyte solo per la tabella delle pagine non è pensabile, né ora e nemmeno per i prossimi anni a venire, di conseguenza, è necessaria una soluzione differente per spazi di indirizzamento virtuali paginati di 64 bit. Una soluzione è la tabella delle pagine invertite: in questa tabella, è presente un elemento per ogni pagina fisica nella memoria reale, invece che un elemento per ogni pagina nello spazio di indirizzamento virtuale. Per esempio, con indirizzi virtuali di 64 bit, una pagina di 4 KB, e 256 MB di RAM, una tabella delle pagine invertite richiede solo 65536 elementi, e questi elementi tengono traccia di quale (processo, pagina virtuale) è localizzato nella pagina fisica.

Sebbene le tabelle delle pagine invertite recuperino una gran quantità di spazio, almeno quando lo spazio di indirizzamento virtuale è più grande della memoria fisica, esse presentano un problema serio: la traduzione da virtuale a fisico diventa molto più difficile. Quando il processo riferisce la pagina virtuale p, l'hardware non può più trovare la pagina fisica utilizzando p come indice nella tabella delle pagine, dovrà invece ricercare nell'intera tabella delle pagine invertite un elemento del tipo (n, p); inoltre, questa ricerca deve essere fatta non solo sui fault di pagina, ma per ogni riferimento di memoria, e scorrere una tabella di 64K per ogni riferimento di memoria non è proprio il modo migliore per rendere la vostra macchina super veloce. Il modo per uscire da questo dilemma è quello di usare il TLB; se il TLB può mantenere tutte le pagine utilizzate di frequente, la traduzione risulterà veloce come se si utilizzassero tabelle delle pagine regolari.

Per quanto riguarda le miss di TLB, però, la tabella delle pagine invertite deve essere controllata dal software; un modo realizzabile per eseguire questa ricerca è quello di avere una tabella hash indicizzata sull'indirizzo virtuale: tutte le pagine virtuali correntemente in memoria che hanno lo stesso valore d'indice sono concatenate insieme. Se la tabella hash ha tanti slot quante pagine fisiche ha la macchina, la catena sarà mediamente lunga un solo

elemento, velocizzando enormemente la traduzione. Una volta trovato il numero di pagina fisica, la nuova coppia (virtuale, fisica) verrà inserita nel TLB.

Tabella dei frame

Il SO tiene traccia dello stato di occupazione di ogni frame fisico attraverso la tabella dei frame. Ogni voce ha uno **stato**, che indica se il frame è occupato o libero; se occupato dice anche da quale processo. Tale tabella viene consultata:

- Ogni volta che viene creato un nuovo processo per creare la relativa tabella delle pagine di quel processo;
- Ogni volta che un processo chiede di allocare nuove pagine.

Memoria associativa (TLB)

Nella maggior parte degli schemi di paginazione le tabelle delle pagine vengono mantenute in memoria, a causa della loro grande dimensione, e questo ha un enorme impatto sulle prestazioni. Consideriamo, per esempio, una istruzione che copi un registro in un altro registro: in assenza di paginazione, l'istruzione esegue un solo riferimento alla memoria, dovuto al prelievo dell'istruzione stessa. Con la paginazione, sono necessari ulteriori riferimenti alla memoria per accedere alla tabella delle pagine; dal momento che la velocità di esecuzione è limitata, in generale, dalla velocità con cui la CPU può prelevare istruzioni e dati dalla memoria, produrre due riferimenti della tabella delle pagine per ogni riferimento in memoria, riduce le prestazioni di 2/3. A queste condizioni, nessuno vorrebbe usare la paginazione.

I progettisti di calcolatori hanno trovato una soluzione che si basa su questa osservazione: la maggior parte dei programmi tende ad eseguire un alto numero di riferimenti ad un piccolo insieme di pagine, e non viceversa. Così solo una piccola frazione degli elementi della tabella delle pagine viene utilizzata frequentemente, il resto non viene usato quasi mai. La soluzione pensata consiste nel dotare i calcolatori di un piccolo dispositivo hardware che serve a mappare gli indirizzi virtuali sugli indirizzi fisici senza passare dalla tabella delle pagine. Il dispositivo, chiamato **TLB** (**T**ranslation **L**ookaside **B**uffer) (o a volte memoria associativa), è illustrato di seguito.

Valid	Virtual Page n.	Modified	Protection	Page frame n. (Repliche campi omonimi PT entry)
1	140	1	RW	31
1	20	0	R X	38
1	130	1	RW	29
0	129	1	RW	50
1	21	0	R X	45

Il TLB è un buffer, cioè una memoria tampone (o, nelle implementazioni più sofisticate, una cache nella CPU), che l'**MMU** (Memory Management Unit) usa per velocizzare la traduzione degli Indirizzi Virtuali. Il TLB possiede un numero fisso di elementi della Page Table, la quale viene usata per mappare gli Indirizzi Virtuali in Indirizzi Fisici. La Memoria virtuale è lo spazio visto da un processo che può essere più grande della memoria fisica (reale). Questo spazio è catalogato in pagine di dimensioni prefissate. Generalmente solo alcune pagine vengono caricate nella memoria fisica in zone dipendenti dalla politica di Page Replacement. La Page Table (generalmente caricata in memoria) tiene traccia di dove le pagine virtuali sono caricate nella memoria fisica. Il TLB è una cache della Page Table, cioè solamente un sottoinsieme del suo contenuto viene memorizzato.

Di solito si trova dentro alla MMU e contiene un piccolo numero di elementi, otto nell'esempio sopra, di solito non più di 64: ciascun elemento contiene informazioni che riguardano una pagina, con le seguenti voci:

- Numero di pagina virtuale;
- Bit per validità della voce della TLB;
- Codice di protezione;
- Dirty bit;
- Numero di frame.

Quando alla MMU arriva un indirizzo virtuale da tradurre, l'hardware controlla dapprima se il relativo numero di pagina virtuale è presente nel TLB confrontandolo simultaneamente (cioè in parallelo) con tutti gli elementi. Se si trova un elemento con lo stesso numero di pagina virtuale (**page hit**) e non si ha violazione dei bit di protezione, il numero della pagina fisica viene direttamente preso dal TLB, senza bisogno di dover accedere alla tabella delle pagine. Se il numero di pagina virtuale risulta presente nel TLB ma l'istruzione sta tentando di scrivere in una pagina

a sola lettura, viene generato un errore di protezione, in maniera del tutto analoga a quanto verrebbe fatto in seguito ad un accesso alla tabella delle pagine. Un caso interessante si ha quando il numero di pagina virtuale non è presente nel TLB. La MMU si accorge della mancanza (**page miss**) ed esegue una normale ricerca nella tabella delle pagine; in seguito, scarica uno degli elementi del TLB e lo rimpiazza con l'elemento della tabella delle pagine appena trovato. Così, se a quella pagina si farà riferimento nell'immediato futuro, si avrà un page hit, piuttosto che un page miss.

Quando si cancella un elemento dal TLB, il bit di modifica viene copiato nella tabella delle pagine in memoria, mentre gli altri valori si trovano già nell'elemento che sta in memoria; quando si carica il TLB dalla tabella delle pagine, tutti i campi vengono presi dalla memoria.

Gestione software del TLB miss

Molte macchine moderne gestiscono le pagine quasi completamente via SW. Su queste macchine, gli elementi del TLB sono caricati esplicitamente dal SO; quando avviene una page miss nel TLB, anziché lasciare che la MMU cerchi nella tabella delle pagine e prelevi il riferimento alla pagina richiesto, genera semplicemente un fault di TLB e rimanda il problema al SO. Il sistema deve trovare la pagina, rimuovere un elemento dal TLB, inserirne uno nuovo, e fare ripartire l'istruzione che era fallita, e certamente, tutto questo deve essere fatto con una manciata di istruzioni poiché le page miss nel TLB accadono molto più frequentemente che i fault di pagina. Sorprendentemente, se il TLB è ragionevolmente grande (ad esempio 64 elementi) in modo da ridurre il tasso di miss, la gestione via software del TLB risulta essere sufficientemente efficiente. Il guadagno principale è quello di avere una MMU più semplice, che libera un considerevole ammontare di spazio nei chip della CPU per la cache e per altre caratteristiche che possono migliorare le prestazioni.

Sono state sviluppate diverse strategie per migliorare le prestazioni su macchine che gestiscono via software il TLB. Un approccio riguarda sia la riduzione delle miss di TLB sia la riduzione del costo di una miss di TLB nel momento in cui si verifica. Per ridurre le miss del TLB, qualche volta il SO può utilizzare la sua intuizione per determinare quali pagine verranno probabilmente utilizzate in seguito e per caricare in anticipo gli elementi per queste pagine nel TLB.

Il modo normale per gestire una miss di TLB, sia nell'hardware che nel software, è quello di andare nella tabella delle pagine e di eseguire operazioni di indicizzazione per localizzare la pagina riferita. Il problema nell'effettuare questa ricerca tramite software sta nel fatto che le pagine che contengono la tabella delle pagine possono non essere presenti nel TLB, e questo causerà altri fallimenti di TLB durante la gestione della miss. Questi fallimenti possono essere ridotti, mantenendo una grande cache software (per esempio 4KB) di elementi di TLB in una locazione fissata la cui pagina è sempre mantenuta nel TLB: controllando inizialmente la cache software, il SO può sostanzialmente ridurre le miss del TLB.

Effective access time (eat)

Indica attraverso una formula il tempo effettivo per l'accesso alla memoria. Facciamo un esempio:

- Tempo di accesso alla memoria = 100 nsec;
- Tempo di accesso alla memoria associativa (TLB) = 20 nsec;

Il tempo effettivo di accesso sarà in questo caso:

- 120 nsec per TLB hit;
- 220 nsec per TLB miss;

Ipotizziamo una TLB ratio (percentuale di successi) dell'80%;

- Tempo (medio) effettivo di accesso: $0.8 \times 120 + 0.2 \times 220 = 140$ nsec

In generale, dati:

- Tempo di accesso alla memoria: α
- Tempo di accesso alla TLB: β
- TLB ratio: ϵ

Avremo:

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

In un sistema che usa paginazione, l'accesso al TLB richiede 150ns, mentre l'accesso alla memoria richiede 400ns. Quando si verifica un page fault, si perdono 8ms per caricare la pagina che si sta cercando in memoria. Se il page fault rate è il 2% e il TLB hit il 70%, indicare l'EAT ai dati.

Convertendo tutti i tempi in ms, abbiamo:

- tempo di accesso al TLB: $150 \text{ ns} = 150 \cdot 10^{-9} \text{ ms}$;
- tempo di accesso alla memoria: $400 \text{ ns} = 400 \cdot 10^{-9} \text{ ms}$;
- tempo di gestione del page fault: 8 ms;
- page fault rate: $2\% = 0,02$;
- TLB hit: $70\% = 0,7$.

Quindi

$$\begin{aligned}
 EAT &= TLB \text{ hit} \cdot (150 \cdot 10^{-9}) + (1 - \text{page fault rate}) \cdot (150 \cdot 10^{-9} + 400 \cdot 10^{-9}) + \text{page fault rate} \cdot \\
 &\quad (150 \cdot 10^{-9} + 8 + 400 \cdot 10^{-9}) \\
 &= 0,7 \cdot (150 \cdot 10^{-9}) + 0,28 \cdot (150 \cdot 10^{-9} + 400 \cdot 10^{-9}) + 0,02 \cdot (150 \cdot 10^{-9} + 8 + 400 \cdot 10^{-9}) \\
 &= 0,7 \cdot (1,5 \cdot 10^{-8}) + 0,28 \cdot (1,5 \cdot 10^{-8} + 4 \cdot 10^{-8}) + 0,02 \cdot (1,5 \cdot 10^{-8} + 8 + 4 \cdot 10^{-8}) \\
 &= 1,05 \cdot 10^{-8} + 1,54 \cdot 10^{-8} + 0,160011 \\
 &= 0,1627 \text{ ms}
 \end{aligned}$$

Algoritmi di rimpiazzamento della pagine

Le due più importanti decisioni che prende il gestore durante il suo funzionamento sono:

- Decidere quale pagina deve essere sostituita quando si verifica un page fault e non ci sono frame liberi in memoria;
- Decidere periodicamente quanta memoria, cioè quanti frame, allocare a ciascun processo

Queste decisioni vengono prese indipendentemente l'una dall'altra. Quando decide di aumentare o diminuire la memoria allocata a un processo, specifica semplicemente il nuovo numero di frame che dovrebbero essere allocati a ciascun processo. Quando si verifica un fault di pagina, il SO deve scegliere una pagina da rimuovere dalla memoria, per fare spazio alla pagina che deve essere caricata.

Perché si sostituisce una pagina

Quando si verifica un page fault, il gestore della memoria virtuale procede sostituendo una **pagina vittima** attraverso un **algoritmo di sostituzione** delle pagine. L'obiettivo di una politica di sostituzione delle pagine è quello di sostituire solo quelle pagine che non si useranno nell'immediato, e quindi di conseguenza minimizzare il numero di page fault in futuro. Se la pagina che deve essere rimossa è stata modificata durante la sua permanenza in memoria, deve essere riscritta sul disco per rendere quella copia aggiornata.

La pagina che deve essere caricata viene semplicemente scritta sopra la pagina destinata ad essere scaricata. Sebbene sia possibile, ad ogni fault di pagina, scegliere a caso una pagina da scaricare, le prestazioni del sistema sono molto migliori quando sceglio una pagina che non viene usata frequentemente, in caso contrario essa dovrà probabilmente essere ricaricata entro breve tempo, provocando ulteriore perdita di tempo.

Algoritmo OPT (ottimale)

Il miglior algoritmo possibile per il rimpiazzamento delle pagine è facile da descrivere ma impossibile da implementare. Funziona così: nel momento in cui avviene un fault di pagina, in memoria si trova un certo insieme di pagine e a una di queste pagine si farà riferimento proprio nella prossima istruzione (la pagina contenente tale istruzione); altre pagine potrebbero non essere utilizzate per altre 10, 100 o forse 1000 istruzioni, ogni pagina può essere etichettata con il numero di istruzioni che saranno eseguite prima che a quella pagina si faccia riferimento per la prima volta.

L'algoritmo ottimale di rimpiazzamento delle pagine dice semplicemente che dovrà essere rimossa la pagina con l'etichetta più alta. Se una pagina non sarà usata per 8 milioni di istruzioni e un'altra pagina non sarà usata per 6 milioni di istruzioni, rimuovere la prima posticipa il fault di pagina che la riporterà in memoria, il più lontano possibile nel futuro. L'unico problema di questo algoritmo è che è irrealizzabile: al momento del fault di pagina, il SO non ha nessun modo di sapere quando si farà riferimento a ciascuna delle pagine. Nonostante ciò, facendo eseguire un programma su un simulatore e mantenendo traccia di tutti i riferimenti alle pagine, è possibile implementare il rimpiazzamento di pagina ottimale alla seconda esecuzione, usando le informazioni sui riferimenti alle pagine raccolte durante la prima esecuzione.

Algoritmo NRU (non usate di recente)

Per permettere al SO di raccogliere statistiche utili sulle pagine usate e su quelle non usate, la maggior parte dei calcolatori dotati di memoria virtuale ha due bit di stato (R e M) associati a ciascuna pagina.

- R viene messo a 1 ogni volta che la pagina viene riferita (letta o scritta)
- M viene messo a 1 ogni volta che una pagina viene modificata.

Questi bit sono contenuti in ogni elemento della tabella delle pagine. Questi bit devono essere aggiornati ad ogni riferimento in memoria, quindi è essenziale che vengano assegnati dall'HW. Quando viene lanciato un processo, entrambi i bit devono essere messi a 0 dal SO; a ogni ciclo di clock il bit R viene azzerato periodicamente, cioè messo a 0, per distinguere le pagine che non sono state utilizzate di recente.

Se l'hardware non ha questi bit, essi possono esser simulati come segue: quando un processo viene avviato, tutti gli elementi della sua tabella delle pagine vengono marcati come non presenti in memoria. Non appena una qualunque pagina verrà richiesta, ci sarà un fault di pagina; il SO, allora, metterà ad 1 il bit R (nelle sue tabelle interne), cambierà l'elemento della tabella delle pagine per puntare alla pagina corretta, con modalità *READ ONLY*, e rieseguirà l'istruzione. Se la pagina viene successivamente scritta, si avrà un altro fault di pagina, che permetterà al SO di mettere a 1 il bit M e di cambiare la modalità della pagina a *READ/WRITE*.

L'algoritmo NRU rimuove una pagina qualsiasi (a caso) dalla classe di numero inferiore; semplicemente ad ogni intervallo viene fatto un controllo: ci sono pagine di classe 0? si, ne scelgo una e la sostituisco, no, passo alla classe 1; ci sono pagine di classe 1? stesso discorso, si passa in caso alla classe 2 ecc.

Quando avviene un page fault, il SO ispeziona tutte le pagine e le divide in quattro categorie:

- Classe 0: non usata, non modificata ($r=0, m=0$)
- Classe 1: non usata, modificata ($r=0, m=1$)
- Classe 2: usata, non modificata ($r=1, m=0$)
- Classe 3: usata, modificata ($r=1, m=1$)

La classe 1, che apparentemente sembra impossibile da ottenere, si ottiene quando il bit R di una pagina di classe 3 viene messo a 0 dopo un ciclo di clock. L'algoritmo NRU rimuove una pagina a caso dalla classe non vuota di numero più basso.

Supponiamo le pagine attualmente in memoria siano le seguenti (senza nessun ordine particolare):

A($r=1, m=0$), B($r=1, m=1$), C($r=0, m=1$), D($r=1, m=1$), E($r=1, m=0$).

I bit indicati tra parentesi sono rispettivamente i bit di referenziamento e di modifica.

Quale sarebbe la pagina selezionata per la sostituzione dall'algoritmo NRU?

La pagina C, che appartiene alla classe 1, più bassa delle altre.

L'attrattiva principale dell'algoritmo NRU è che è facile da capire, abbastanza semplice da implementare, e che dà delle prestazioni che spesso risultano soddisfacenti.

Algoritmo FIFO (first input first output)

Il SO mantiene una lista di tutte le pagine correntemente in memoria, dove la pagina di testa è la più vecchia e la pagina in coda è quella arrivata più di recente. Al momento del fault di pagina, la pagina in testa viene rimossa anche se è la pagina più utilizzata. Tale scelta non è sempre felice: può rimuovere pagine vecchie, ma magari molto usate.

Algoritmo FIFO seconda chance

Una semplice modifica all'algoritmo FIFO evita il problema dovuto allo scaricamento di una pagina molto usata: è quello di controllare il bit R della pagina più vecchia. Se è a 0, la pagina è sia vecchia sia non usata, così può essere immediatamente rimpiazzata.

Se il bit R vale 1, allora la pagina è usata spesso, quindi R viene posto a 0, la pagina viene spostata alla fine della coda delle pagine, ed il suo tempo di caricamento viene aggiornato come se la pagina fosse appena arrivata in memoria. Poi si continua la ricerca fino a trovare la vittima che ha R=0.

$$A(r=1), B(r=1), C(r=0) \rightarrow B(r=1), C(r=0), A(r=0) \rightarrow C(r=0), A(r=0), B(r=1), \rightarrow C = \text{vittima}$$

Ciò che la tecnica della seconda opportunità fa è di cercare una pagina vecchia che non sia stata usata nel precedente periodo del clock. Se tutte le pagine sono state usate, allora la tecnica della seconda opportunità degenera e diventa uguale alla FIFO (nell'esempio sopra, se C fosse stato R=0, allora avrebbe scelto come vittima A).

Supponiamo di avere una coda FIFO delle pagine attualmente in memoria:

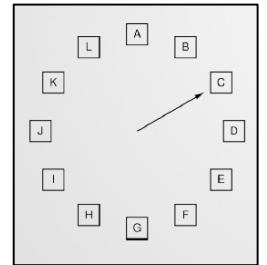
A (r=1, m=0), B (r=1, m=1), C (r=0, m=0), D (r=0, m=1), E (r=1, m=0)

Tale lista è ordinata secondo l'ordine d'arrivo (A è quella più vecchia). I bit indicati tra parentesi sono rispettivamente il bit di referenziamento e il bit di modifica. Quale sarebbe la pagina selezionata per la sostituzione dall'algoritmo della Seconda chance?

La pagina C perché è la prima che nello scorrimento della coda ha R=1.

Algoritmo Clock (dell'orologio)

L'idea dell'algoritmo della seconda chance è buona ma si può implementare in modo più efficiente: un approccio migliore è quello di mantenere le pagine in una lista circolare a forma di orologio, come mostrato in figura; una lancetta punta alla pagina più vecchia. Quando si verifica un fault di pagina, si controlla la pagina puntata dalla lancetta. Se il suo bit R è a 0, la pagina viene scaricata, la pagina nuova viene inserita al suo posto nella disposizione ad orologio e la lancetta viene spostata in avanti di una posizione; se il bit R è a 1, viene messo a zero e la lancetta viene spostata in avanti di una posizione. Questo processo viene ripetuto fino a che si trova una pagina con il bit R messo a 0.



Algoritmo LRU (last recently used)

Probabilmente le pagine più usate di recente lo saranno anche in futuro; idea: rimuovere le pagine meno usate di recente. Quando si verifica un page fault, viene scaricata la pagina usata meno di recente. Sebbene LRU sia realizzabile dal punto di vista teorico, non è economica, poiché per implementarlo completamente è necessario mantenere liste concatenate di tutte le pagine in memoria, con la pagina usata più di recente in testa alla lista; la difficoltà sta nel fatto che la lista va aggiornata ad ogni riferimento alla memoria.

Tuttavia esistono altri modi per implementare LRU con HW speciale. Questo metodo richiede che l'HW sia dotato di contatore C di 64 bit, che viene incrementato automaticamente dopo ogni istruzione; inoltre ogni elemento della tabella delle pagine deve avere un campo abbastanza grande da contenere il contatore. Dopo ogni riferimento in memoria, il valore di C viene memorizzato nell'elemento della tabella delle pagine corrispondente alla pagina appena usata. Quando si verifica un page fault, il SO somma C con R di ogni elemento della page table per trovare quello con valore più basso: la pagina trovata è quella usata meno frequentemente.

0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 3	0 1 2 3
0 1 1 1	0 0 1 1	c 0 0 1	0 0 0 0	0 0 0 0
1 0 0 0	1 0 1 1	1 0 0 1	1 0 0 0	1 0 0 0
2 0 0 0	0 0 0 0	1 1 0 1	1 1 0 0	1 1 0 1
3 0 0 0	0 0 0 0	0 0 0 0	1 1 1 0	1 1 0 0
→pagina 0	→pagina 1	→pagina 2	→pagina 3	→pagina 2
0 0 0 0	0 1 1 1	0 1 1 0	0 1 0 0	0 1 0 0
1 0 1 1	0 0 1 1	0 0 1 0	0 0 0 0	0 0 0 0
1 0 0 1	0 0 0 1	0 0 0 0	1 1 0 1	1 1 0 0
1 0 0 0	0 0 0 0	1 1 1 0	1 1 0 0	1 1 1 0
→pagina 1	→pagina 0	→pagina 3	→pagina 2	→pagina 3

Esiste un altro algoritmo LRU realizzato in HW. Per una macchina con n pagine fisiche, l'HW LRU può mantenere una matrice di $n \times n$ bit, inizialmente tutti a 0; ogni volta che viene riferita la pagina fisica k, l'HW mette prima tutti i bit della riga k a 1, poi tutti i bit della colonna k a 0. In ogni istante, la riga con il più piccolo valore binario è quella usata meno di recente.

Algoritmo NFU (not frequently used)

Implementare l'algoritmo LRU in HW è dispendioso, poche macchine hanno a disposizione tale potenza di calcolo; possiamo simularlo via SW tramite l'algoritmo NFU. L'algoritmo richiede un contatore associato ad ogni pagina, inizialmente a 0; ad ogni interruzione dal clock, il SO esamina tutte le pagine in memoria e per ogni pagina, il bit R viene sommato al contatore.

In effetti, i contatori sono un tentativo di tenere traccia della frequenza con cui si fa riferimento ad ogni pagina; quando si verifica un fault di pagina, viene scelta per il rimpiazzamento la pagina con il contatore più basso. Il problema principale con l'algoritmo NFU è che non si dimentica mai di niente.

Ad esempio, in un compilatore a più passate le pagine che sono state usate pesantemente durante la prima passata, possono continuare ad avere un valore del contatore alto anche nelle passate successive. Infatti, se la prima passata ha un tempo di esecuzione più lungo delle altre, le pagine contenenti il codice per le passate successive avranno sempre un contatore più basso delle pagine della prima passata. Di conseguenza, il SO rimuoverà pagine utili invece

di pagine che non vengono più usate. Fortunatamente una piccola modifica all'algoritmo NFU lo rende capace di simulare piuttosto bene l'algoritmo LRU.

Problema: può erroneamente privilegiare pagine che sono state molto utilizzate in passato ma che invece sono scarsamente usate di recente: queste lo saranno, probabilmente, anche nel prossimo futuro.

Algoritmo di Aging (invecchiamento)

Tale algoritmo è un'modifica dell'NRU e prevede un aumento graduale della proprietà dei processi che si trovano in attesa nel sistema da lungo tempo. Ciò è realizzabile mantenendo un contatore C associato ad ogni pagina caricata in memoria. Tale contatore viene consultato nel momento in cui si deve scegliere quale pagina rimuovere dalla memoria: viene scelta quella con il contatore più basso (il più giovane).

La modifica avviene in due fasi.

1. Ognuno dei contatori C viene shiftato a destra di un bit prima di sommarvi il bit R
2. Il bit di referenziamento R viene sommato al bit più a sinistra (bit più significativo) invece che a quello più a destra.

Supponiamo che dopo il primo ciclo di clock i bit R delle pagine da 0 a 5 abbiano i valori 1, 0, 1, 0, 1 e 1, rispettivamente (la pagina 0 a 1, la pagina 1 a 0, la pagina 2 a 1, eccetera). In altre parole, tra il ciclo 0 ed il ciclo 1, le pagine 0, 2, 4 e 5 sono state usate, ed i loro bit R messi ad 1, mentre quelli delle altre sono rimasti a 0. Dopo che i sei contatori corrispondenti (che erano tutti a zero) sono stati shiftati ed il bit R inserito a sinistra, tali contatori hanno i valori mostrati come in figura: le quattro colonne restanti mostrano i sei contatori dopo i successivi quattro cicli di clock. Quando si verifica un fault di pagina, viene rimossa la pagina con il contatore più basso, cioè si convertono i bit dei contatori in decimale e si prende il numero più basso; quindi nell'esempio viene rimossa la pagina 3 dell'ultimo tick dato che vale 32 in decimale (le altre sono maggiori di 32). Chiaro che una pagina che non è stata usata per, diciamo, quattro cicli di clock, avrà quattro zeri all'estrema sinistra nel contatore, e quindi avrà un valore più basso del contatore di una pagina che non è stata usata per tre cicli di clock.

	R bits for pages 0-5, clock tick 0	R bits for pages 0-5, clock tick 1	R bits for pages 0-5, clock tick 2	R bits for pages 0-5, clock tick 3	R bits for pages 0-5, clock tick 4
Page	1 0 1 0 1 1	1 1 0 0 1 0	1 1 0 1 0 1	1 0 0 0 1 0	0 1 1 0 0 0
0	1000000	1100000	1110000	1111000	0111100
1	0000000	1000000	1100000	0110000	1011000
2	1000000	0100000	0010000	0001000	1000100
3	0000000	0000000	1000000	0100000	0010000
4	1000000	1100000	0110000	1011000	0101100
5	1000000	0100000	1010000	0101000	0010100

Abbiamo però un altro problema relativo: registrando soltanto un bit per ogni intervallo di tempo, abbiamo perso la possibilità di distinguere all'interno di un intervallo di clock i riferimenti avvenuti da quelli accaduti successivamente. La differenza tra l'algoritmo LRU e l'algoritmo dell'invecchiamento è che in quest'ultimo i contatori hanno un numero finito di bit, 8 bit in questo esempio. Supponiamo che due pagine abbiano ciascuna il contatore a 0: tutto ciò che possiamo fare è sceglierne una a caso. In realtà, potrebbe anche succedere che una delle pagine sia stata usata per l'ultima volta 9 cicli fa, e che l'altra sia stata usata l'ultima volta 1000 cicli fa; non abbiamo nessun modo per accorgercene. In pratica, comunque, 8 bit sono generalmente sufficienti se il ciclo di clock è di circa 20 ms: se una pagina non è stata usata in 160 ms, probabilmente non è così importante.

Anomalia di Belady

Essa è un fenomeno che si presenta in alcuni algoritmi di rimpiazzamento delle pagine di memoria per cui **la frequenza dei page fault può aumentare con l'aumento del numero di frame assegnati ai processi**. Ne soffrono gli algoritmi FIFO con alcune combinazioni di richieste di pagina.

Essendo solo gli algoritmi FIFO ad essere affetti da quest'anomalia, (quindi anche seconda chance, clock e NRU che riducono a FIFO) è sufficiente utilizzare i cosiddetti algoritmi a stack per gestire la paginazione. La ricerca verso questo genere di soluzioni ha avuto particolare impulso proprio dopo la scoperta di Nelson e Shadler, orientandosi inizialmente verso la definizione dell'algoritmo ottimale (OPT), definito solo in forma teorica data l'impossibilità di prevedere la successione di riferimenti. Siccome non è possibile implementare OPT su sistemi reali, la ricerca si è concentrata sull'algoritmo LRU (Least Recently Used), che invece sceglie la pagina "vittima" fra quelle utilizzate più indietro nel tempo. Dato il fatto che anche quest'ultimo approccio è eccessivamente dispendioso, si sono individuate approssimazioni come l'algoritmo con seconda chance e l'algoritmo con seconda chance migliorato, entrambi basati sull'utilizzo di bit di validità.

LRU non soffre dell'anomalia per la **proprietà di inclusione**: l'insieme di pagine caricate avendo n frame è incluso in quello che si avrebbe avendo $n+1$ frame

$$B_t(n) \subseteq B_t(n+1) \forall t, n$$

- NFU, aging: godono della proprietà di inclusione (appross. LRU);
- Seconda chance, clock: soffrono dell'anomalia (riducono a FIFO);
- NRU: soffre dell'anomalia (riduce a FIFO).

Riepilogo sugli algoritmi

- OPT: non implementabile, ma utile come termine di paragone;
- NRU*: approssimazione rozza dell'LRU;
- FIFO*: può portare all'eliminazione di pagine importanti;
- FIFO Seconda chance*: un netto miglioramento rispetto a FIFO;
- Clock*: come FIFOSC ma più efficiente;
- LRU: eccellente idea (vicina a quella ottima) ma difficilmente realizzabile se non in hardware;
- NFU: approssimazione software abbastanza rozza dell'LRU;
- Aging: buona approssimazione di LRU con implementazione software efficiente.

* soffre dell'anomalia di Belady

Allocazione dei frame

In un sistema multiprogrammato i frame (pagine di memoria) disponibili fra i processi vengono distribuiti attraverso varie soluzioni detti algoritmi di allocazione:

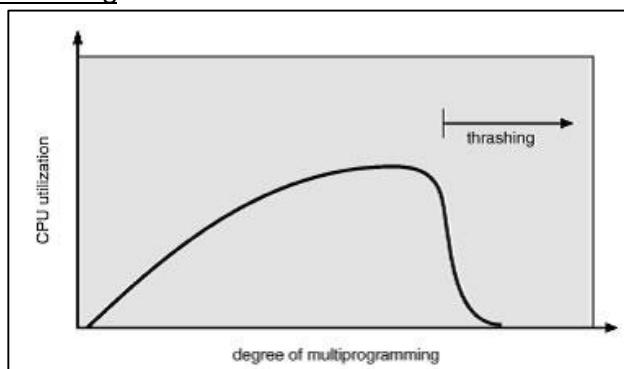
- **Allocazione uniforme (equa)**: lo stesso numero di frame a tutti i processi: n frame, p processi, n/p frame a ciascun processo
- **Allocazione proporzionale**: tiene conto del fatto che i processi hanno dimensioni diverse. Se le dimensioni in pagine di tre processi sono: $P_1 = 4$, $P_2 = 6$, $P_3 = 12$ e ci sono 11 frame disponibili, l'allocazione sarà: $P_1 = 2$, $P_2 = 3$, $P_3 = 6$.
- **Allocazione proporzionale in base alla priorità**: tiene conto del fatto che i processi hanno priorità diverse
 - Al processo i di dimensione s_i assegniamo $\alpha = \frac{s_i}{S \times m}$ frame dove $S = \sum s_i$

Scelta della vittima da rimuovere

Una volta deciso come allocare le pagine, dobbiamo decidere in quale gruppo di pagine (di che processo) dobbiamo scegliere la vittima da rimuovere dalla memoria principale. Anche qui ci sono diverse soluzioni.

- **Allocazione globale**: scegliamo la vittima fra tutte le pagine della memoria principale (di solito, escluse quelle del SO). Potremmo in questo modo portare via una pagina ad un processo diverso da quello che ha generato il page fault.
- **Allocazione locale**: scegliamo la vittima fra le pagine del processo che ha generato page fault. In questo modo il numero di frame per processo rimane costante. Attenzione però, se si danno troppe (relativamente) pagine ad un processo, si può peggiorare la situazione del sistema perché gli altri processi genereranno più page fault.

Thrashing



In generale l'allocazione globale è di solito preferita per sistemi time sharing. Può succedere che avendo pochi frame, ogni processo ha un'alta probabilità di generare un page fault. Per gestire il page fault, una pagina viene tolta dalla memoria primaria, probabilmente ad un altro processo, il quale ha a sua volta un'alta probabilità di generare un page fault. Alla fine tutti i processi sono attivissimi a rubarsi pagine l'un l'altro, per fare page-fault subito dopo! Questo fenomeno si chiama **thrashing** ed è assolutamente da evitare dimensionando opportunamente il numero di pagine per processo. Intuitivamente, il thrashing si verifica quando si tenta di aumentare troppo il grado di multiprogrammazione, in modo da sfruttare al massimo il tempo di CPU e

incrementare il throughput del sistema. Oltre una certa soglia però, i processi passano più tempo a fare page fault che a portare avanti il loro lavoro, e il livello di utilizzazione della CPU, e quindi il throughput, crollano verticalmente!

In definitiva quindi, il thrashing è una sorta di “ingolfamento” del sistema: vogliamo sfruttarlo al meglio “iniettando” più e più processi nel sistema, col risultato che i processi si danneggiano a vicenda. Per risolvere questo problema si può per esempio misurare la frequenza di page-fault, per vedere se è “accettabile” rispetto alle prestazioni che vogliamo ottenere dal sistema. Se la frequenza osservata è troppo bassa, possiamo togliere ai processi qualche frame, e aumentare il grado di multiprogrammazione. Se la frequenza osservata è troppo alta, diminuiamo il grado di multiprogrammazione e ridistribuiamo i frame liberati fra i processi non rimossi.

Diciamo molto superficialmente che sia il sovradimensionamento che il sottodimensionamento di memoria concesso a un processo porta a cali di prestazioni del sistema e scarsa efficienza della CPU. Tuttavia, non è chiaro il modo in cui il gestore della memoria virtuale decide il giusto numero di frame da allocare a ogni processo, ossia il giusto valore di allocazione per ogni processo. Si può scegliere tra assegnare il minimo strutturale numero di frame (set istruzioni, livello di indirizzamento indiretto) o i massimi (l’intera memoria libera).

Esiste un modello più diretto per approcciare il thrashing: tramite monitoraggio della **Page Fault Frequency (PFF)** dei processi; esso è un algoritmo che calcola appunto la frequenza dei page fault, caratteristico dei sistemi in sovraccarico.

Controllo dell’allocazione dei frame

Vengono utilizzati due approcci per controllare l’allocazione dei frame per un processo:

- **Allocazione di memoria fissa:** l’allocazione di memoria per un processo è fissa; di conseguenza, la prestazione di un processo è indipendente dagli altri processi del sistema. Quando si verifica un page fault in un processo, viene sostituita una delle sue pagine. Questo approccio è detto sostituzione di pagina locale;
- **Allocazione di memoria variabile:** l’allocazione di memoria può essere variata in due modi. Quando si verifica un page fault, tutte le pagine di tutti i processi che sono in memoria possono essere prese in considerazione per la sostituzione. Ciò è identificato come sostituzione globale delle pagine. Alternativamente, il gestore della memoria virtuale può rivedere l’allocazione della memoria per un processo periodicamente sulla base della sua località e sul comportamento riguardo ai page fault, ma quando si verifica un page fault esegue una sostituzione locale di pagina.

Nell’allocazione fissa in ambito globale, le decisioni riguardanti l’allocazione di memoria sono eseguite staticamente. La memoria da allocare a un processo è determinata in base ad alcuni criteri quando il processo viene inizializzato. La sostituzione di pagina è sempre eseguita localmente. Questo metodo risente di tutti i problemi connessi a una decisione statica: un sottodimensionamento o un sovradimensionamento di memoria per un processo può influenzare la prestazione del processo stesso e quella del sistema.

Nell’allocazione variabile in ambito globale, l’allocazione per il processo attualmente in esecuzione può diventare troppo grande.

La miglior soluzione tra le tre è senza dubbio l’allocazione variabile in ambito locale che utilizza la sostituzione locale delle pagine, perché il gestore della memoria virtuale determina il giusto valore di allocazione per un processo, di volta in volta.

Working set

Il gruppo di pagine della memoria fisica attualmente dedicato ad un processo specifico in una fase della propria elaborazione è chiamato working set per il processo stesso (es. due array globali, più istruzioni di copia). Il numero di pagine nel working set può aumentare o diminuire, a seconda della disponibilità delle pagine stesse.

Se tutto l’insieme di lavoro fosse in memoria, il processo girerebbe senza causare molti fault di pagina fino al momento in cui passa ad un’altra fase di esecuzione, se la memoria disponibile è troppo piccola per poter contenere tutto l’insieme di lavoro, il processo causerà molti fault di pagina e girerà molto lentamente dal momento che per eseguire un’istruzione ci vogliono pochi nanosecondi e per caricare una pagina dal disco ci vogliono 10 millisecondi. Con un tasso di uno o due istruzioni ogni 10 millisecondi, ci vorrà un’eternità per completare l’esecuzione.

Calcolo del working set

Si approssima usando

- Interrupt periodici
- Bit di referenziamento R
- Un log che conserva la storia di R in base al parametro Δ

Il parametro Δ ci dice il numero di accessi alla memoria.

Variazione del working set

La dimensione del working set può cambiare durante l'esecuzione di un processo in quanto si può verificare una successione di fasi di esecuzione che coinvolgono un numero differente di pagine. Ad esempio, si consideri un programma con due cicli:

1. Il primo ciclo esegue delle operazioni sugli elementi di un piccolo vettore;
2. Il secondo ciclo opera una sommatoria degli elementi di un vettore di dimensione maggiore.

A questo punto si può concludere che il working set del processo sarà limitato (perchè avrà bisogno di accedere a poche pagine) fintanto che eseguirà le istruzioni all'interno del primo ciclo, mentre crescerà (dato che il secondo vettore è di dimensione maggiore e quindi occuperà più pagine) quando il processo passerà ad eseguire le istruzioni del secondo ciclo.

Altri esempi di processi con working set di dimensione variabile sono i compilatori di linguaggi che suddividono in fasi la propria esecuzione (analisi lessicale, parsing, costruzione della tabella dei simboli, ecc.).

Diminuzione dei working set

Il SO diminuisce i working set del processo nei seguenti modi:

- Scrivendo su pagine modificate in un'area dedicata, su di un dispositivo memoria di massa (generalmente conosciuti come spazio di swapping o paging)
- Contrassegnando pagine non modificate come libere (non vi è alcuna necessità di scrivere queste pagine su disco in quanto queste non sono state modificate)

Per determinare i working set appropriati per tutti i processi, il SO deve possedere tutte le informazioni sull'utilizzo per tutte le pagine. In questo modo, il SO, determina le pagine usate in modo attivo (risiedendo sempre nella memoria) e quelle non utilizzate (e quindi da rimuovere dalla memoria). In molti casi, una sorta di algoritmo non usato di recente, determina le pagine che possono essere rimosse dai working set dei processi.

Paginazione su richiesta

Nella forma di paginazione più pura, i processi vengono fatti partire senza che nessuna delle loro pagine sia presente in memoria. Non appena la CPU tenta di prelevare la prima istruzione, avviene un fault di pagina che fa sì che il SO carichi la pagina che contiene la prima istruzione. Rapidamente, si hanno altri fault per le pagine delle variabili globali e dello stack, e dopo un po', il processo dispone della maggior parte delle pagine di cui ha bisogno e comincia a girare con un numero relativamente basso di fault di pagina. Quando un processo viene mandato in esecuzione, il gestore della memoria virtuale carica solo quella porzione che contiene l'indirizzo di start del processo, cioè l'indirizzo dell'istruzione con cui la sua esecuzione comincia. Successivamente, carica altre porzioni del processo solo quando necessarie. Questa tecnica prende il nome di caricamento su richiesta

Questa strategia viene detta di **paginazione a richiesta**, dal momento che le pagine vengono caricate a richiesta, piuttosto che anticipatamente. Poiché non vi è abbastanza memoria per soddisfare tutte le richieste dei fault la maggior parte dei processi non lavora in questo modo: essi mostrano una località dei riferimenti, il che significa che in ognuna delle fasi dell'esecuzione, il processo fa riferimento solo a una frazione relativamente piccola delle sue pagine.

Prepaging

Caricare le pagine prima di lasciar andare in esecuzione il processo viene detto anche **prepaging** (pre-paginazione). Si noti che l'insieme di lavoro cambia nel tempo.

Politica di pulizia

La paginazione funziona bene quando c'è una gran quantità di pagine fisiche libere. Per assicurare un rifornimento abbondante di pagine fisiche libere, molti sistemi di paginazione hanno un processo in background, detto **demone di**

paginazione che è inattivo per la maggior parte del tempo, ma che viene risvegliato ad intervalli fissi per ispezionare lo stato della memoria. Se ci sono troppe poche pagine fisiche libere, il demone di paginazione inizia a selezionare le pagine da scaricare usando l'algoritmo di rimpiazzamento prescelto, e se queste pagine sono state modificate dopo il caricamento, vengono scritte sul disco. Ad ogni buon conto, viene memorizzato quale era il contenuto precedente delle pagine, e nel caso in cui una delle pagine scaricate venga richiesta prima che sia stata sovrascritta, essa viene semplicemente riallocata togliendola dall'insieme delle pagine libere. Mantenere un rifornimento di pagine libere dà prestazioni migliori che allocare tutta la memoria disponibile e poi cercare una pagina da scaricare quando se ne ha la necessità. Il demone di paginazione assicura almeno che tutte le pagine libere siano pulite, così che esse non debbano essere scaricate sul disco in tutta fretta quando vengono richieste.

Un modo per implementare questa tecnica di pulizia è di utilizzare un orologio a due lancette. La prima lancetta è controllata dal demone di paginazione; quando essa punta ad una pagina sporca, la pagina viene scritta su disco e la lancetta avanza; quando punta ad una pagina pulita, avanza semplicemente. La seconda lancetta è utilizzata per il rimpiazzamento delle pagine, come nell'algoritmo standard dell'orologio. Ora, con l'utilizzo del demone di paginazione, la probabilità che la seconda lancetta trovi una pagina pulita è incrementata.

Dimensione della pagina

La dimensione delle pagine è un parametro molto importante del sistema. Se si usano pagine piccole si ha il vantaggio di avere uno spreco di memoria minore (a causa dell'inutilizzo parziale dell'ultima pagina).

Se si usano pagine grandi si ha il vantaggio di avere tabelle delle pagine piccole. Inoltre, un altro vantaggio è legato al fatto che il tempo di trasferimento necessario per trasferire una pagina grande è simile a quello necessario per trasferire una pagina piccola.

Soltamente la dimensione delle pagine va da 1 KB a 64 KB.

Come si calcola la dimensione ottimale di una pagina

Spesso, la dimensione della pagina può essere scelta dal SO. Determinare la dimensione di pagina ottima richiede il bilanciamento di alcuni fattori contrapposti, e come risultato non esiste un ottimo generale.

Per cominciare, vi sono due fattori che spingono per una dimensione piccola della pagina: un segmento scelto a caso, di testo di programma, di dati o della pila, non occupa un numero di pagine intero; in media metà dell'ultima pagina sarà vuota, e lo spazio in più nella pagina viene sprecato. Questo spreco di spazio viene chiamato **frammentazione interna**. Dati n segmenti in memoria e pagine di dimensione di p byte, $np/2$ byte risulteranno sprecati per via della frammentazione interna.

Questo ragionamento depone a favore di una piccola dimensione delle pagine. In generale, una pagina di dimensioni grandi farà tenere in memoria una parte di programma non usato, rispetto ad una pagina di dimensioni più piccole. D'altra parte, pagine di dimensione più piccola implicano la necessità di un maggior numero di pagine per il programma, e quindi tabelle delle pagine più grosse.

I trasferimenti da e per il disco avvengono di solito una pagina alla volta, e la maggior parte del tempo viene speso per il movimento delle testine ed aspettando la rotazione del disco (tempo di seek e di latenza) e di conseguenza, il trasferimento di una pagina di piccole dimensioni richiede circa lo stesso tempo del trasferimento di una pagina di grosse dimensioni. Ci potrebbero volere $64 \cdot 10$ ms per caricare 64 pagine da 512 byte, ma solo $4 \cdot 12$ ms per caricare quattro pagine da 8KB.

Su alcune macchine, la tabella delle pagine deve essere caricata in registri hardware ogni volta che la CPU passa da un processo ad un altro; su queste macchine, avere pagine di piccola dimensione significa che il tempo richiesto per caricare i registri di pagina aumenta man mano che la dimensione delle pagine diminuisce. Inoltre lo spazio occupato dalla tabella delle pagine cresce al decrescere della dimensione delle pagine. Quest'ultimo argomento si può discutere da un punto di vista matematico.

Sia s byte la dimensione media di un processo e p byte la dimensione delle pagine; inoltre, supponiamo che ciascun elemento della tabella delle pagine richieda e byte. Il numero di pagine approssimativamente richiesto per ognuno dei processi è dunque s/p ed esse occupano se/p byte nella tabella delle pagine. La memoria sprecata nell'ultima

pagina per via della frammentazione interna è di $P/2$ byte. Così l'overhead totale dovuto alla tabella delle pagine e alla perdita dovuta alla frammentazione interna è data da:

$$\text{overhead} = \frac{se}{p} + \frac{p}{2}$$

Il primo termine (dimensione della tabella delle pagine) è grande quando la dimensione delle pagine è piccola; il secondo termine (frammentazione interna) è grande quando la dimensione delle pagine è grande, quindi l'ottimo deve stare in qualche punto intermedio. Prendendo la derivata prima rispetto a p e ponendola a zero, otteniamo l'equazione:

$$-\frac{se}{p^2} + \frac{1}{2} = 0$$

Da questa equazione possiamo derivare una formula che dà la dimensione ottima della pagina:

$$p = \sqrt{2 se}$$

Per $s = 1MB$ ed $e = 8\text{ byte}$ per ogni elemento della tabella delle pagine, la dimensione ottima della pagina è 4KB. La maggior parte dei calcolatori in commercio usano dimensioni delle pagine che variano da 512 byte a 64 KB. Un valore tipico utilizzato è 1KB, ma oggi sono più comuni 4KB o 8KB. Poiché le memorie diventano grandi, la dimensione della pagina tende a diventare anch'essa grande (ma non linearmente).

Quadruplicare la dimensione della RAM raramente duplica la dimensione della pagina.

Frammentazioni

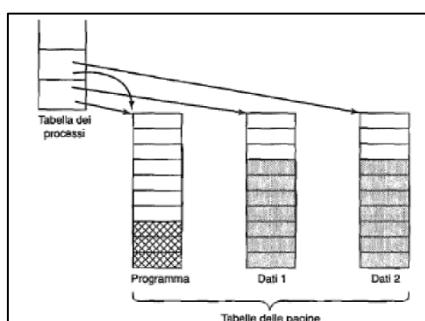
Il fenomeno della frammentazione si verifica quando si inseriscono e si rimuovono processi dalla memoria RAM, oppure file da una memoria di massa (hard disk). Se si considera la memoria RAM, ripetute aggiunte o rimozioni di sequenza di dati di dimensioni diverse all'interno della stessa, comportano una "frammentazione" dello spazio libero disponibile, che quindi non risulta più essere contiguo.

Vengono generalmente individuati due tipi di frammentazione:

- **Frammentazione interna:** si ha frammentazione interna quando viene allocata, ad un processo, più memoria di quanto richiesto dal processo stesso; questo problema si verifica se unallocatore gestisce blocchi di memoria di dimensioni prefissate. Questa frammentazione non influisce sulle prestazioni del sistema ma comporta uno spreco di memoria.
- **Frammentazione esterna:** si ha frammentazione esterna quando un'area di memoria rimane inutilizzata poiché troppo piccola per essere allocata. Si verifica con la gestione delle partizioni dinamiche.

Pagine condivise

In un grosso sistema a multiprogrammazione, è situazione abbastanza comune quella in cui più utenti fanno girare lo stesso programma nello stesso momento. Per evitare di avere più copie della stessa pagina in memoria allo stesso istante risulta chiaramente più efficiente condividere le pagine. C'è un problema, e cioè che non tutte le pagine possono essere condivise; in particolare, le pagine a sola lettura, come le pagine del testo di un programma, possono essere condivise, mentre non possono essere condivise le pagine di dati.

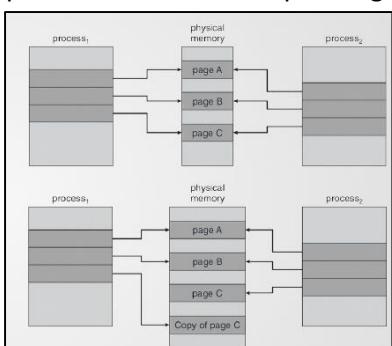


Se vengono utilizzati spazi I e D separati, è abbastanza semplice condividere programmi avendo due o più processi che utilizzano la stessa tabella delle pagine per il proprio spazio- I , e tabelle delle pagine differenti per il proprio spazio- D . Tipicamente in una implementazione che supporta la condivisione in questo modo, le tabelle sono strutture dati indipendenti dalla tabella dei processi; ogni processo ha due puntatori nella sua tabella dei processi: una per la tabella delle pagine dello spazio- I e una per la tabella delle pagine dello spazio- D , come mostrato in figura. Quando lo scheduler sceglie un processo da eseguire, utilizza questi puntatori per localizzare le tabelle delle pagine appropriate e per

impostare la MMU. I processi possono condividere i programmi (o a volte le librerie) anche senza avere gli spazi I e D separati, ma il meccanismo è più complicato.

Quando due o più processi condividono del codice, si verifica un problema con le pagine condivise. Supponiamo che il processo A ed il processo B stiano entrambi facendo girare l'editor e condividano le sue pagine. Se lo scheduler decide di rimuovere il processo A dalla memoria, scaricando tutte le sue pagine e riempendo le pagine fisiche liberate con qualche altro programma, il processo B genererà un alto numero di fault di pagina per ricaricare nuovamente quelle pagine. In maniera del tutto analoga, quando il processo A termina è essenziale poter scoprire che le pagine sono ancora usate in modo che il loro spazio su disco non venga accidentalmente dichiarato libero. Dal momento che la ricerca su tutte le tabelle delle pagine per scoprire se una pagina è ancora in uso risulta di solito troppo costosa, sono necessarie strutture dati particolari che tengano traccia della condivisione delle pagine, specialmente se l'unità di condivisione è la pagina singola (o un insieme di pagine), invece che l'intera tabella delle pagine.

La condivisione dei dati è più complicata della condivisione del codice, ma non è impossibile. In particolare, in UNIX, dopo una chiamata di sistema fork, il processo padre e il processo figlio devono condividere il programma (testo e dati). In un sistema paginato, quello che spesso si fa è fornire a ciascuno di questi processi la propria tabella delle pagine, ed entrambe che puntano allo stesso insieme di pagine, perciò non viene fatta nessuna copiatura delle pagine nel momento in cui si esegue la fork, tuttavia, tutte le pagine dei dati sono mappate in entrambi i processi come *read-only*. Fin tanto che i processi leggono i loro dati, senza modificarli, questa situazione può continuare, ma non appena uno dei processi aggiorna una parola di memoria, la violazione della protezione *read-only* causa una trap al sistema operativo; viene quindi fatta una copia della pagina, in modo che ogni processo ora abbia la sua copia privata. Entrambe le copie vengono quindi impostate a *read-write*, in modo che successive scritture possano



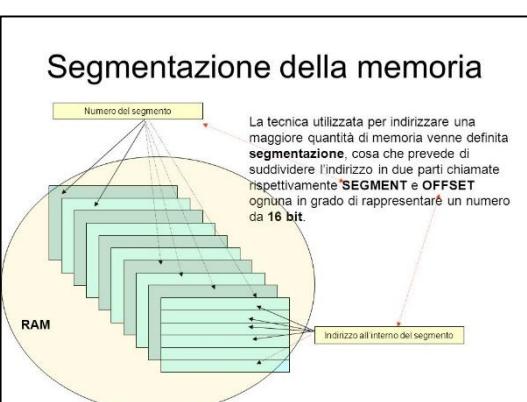
procedere senza ostacoli in entrambe le copie. Questa strategia implica che le pagine che non sono mai scritte (tra cui tutte le pagine del programma) non necessitano di essere copiate, e solo le pagine dei dati che sono veramente scritte necessitano di essere copiate. Questo approccio, chiamato **copy-on-write**, cioè copia quando scrivi, migliora le prestazioni riducendo le copiature (copia finché possibile tutti i tipi di pagine).

In generale, il copy-on-write condivide finché possibile tutti i tipi di pagine (codice e dati). Esiste poi un altro approccio, il **zero-fill-on-demand**, dove come principio base ha che le nuove pagine sono vuote e allocate su richiesta.

Segmentazione

La memoria virtuale è unidimensionale dal momento che gli indirizzi virtuali vanno dall'indirizzo 0 ad un indirizzo massimo. Per molti problemi può essere utile avere uno o più spazi di indirizzamento separati piuttosto che averne uno solo. Questi spazi di indirizzamento vengono chiamati segmenti.

La segmentazione è una comune tecnica di gestione della memoria che suddivide la memoria fisica disponibile in blocchi di lunghezza fissa o variabile detti **segmenti**.



Ciascun segmento consiste di una sequenza lineare di indirizzi da 0 ad un qualche massimo. Segmenti diversi possono avere lunghezze diverse; in più, le lunghezze dei segmenti possono variare durante l'esecuzione. La lunghezza del segmento dello stack può essere allungata quando vi si carica qualcosa e accorciata quando vi si preleva qualcosa' altro.

Nella segmentazione un programma è diviso in vari segmenti di diversa dimensione. Ogni segmento rappresenta un'entità logica del programma come **una funzione** o **una struttura dati**.

Per specificare un indirizzo in questa memoria segmentata o bidimensionale il programma deve fornire un indirizzo formato da due parti: **uno relativo al segmento e l'indirizzo interno del segmento**.

Vantaggi della segmentazione

Una memoria segmentata presenta molti vantaggi: se ciascuna procedura occupa un segmento distinto, con 0 come indirizzo di partenza, il collegamento di procedure compilate separatamente risulta grandemente semplificato. Dopo che tutte le procedure che costituiscono un programma sono state compilate e collegate, una chiamata alla procedura del segmento n userà l'indirizzo composto da due parti (n,0) per indirizzare la parola 0. Se la procedura nel segmento n viene successivamente modificata e ricompilata, non è necessario ricompilare nessun'altra procedura, anche nel caso in cui la nuova versione sia più grande della precedente.

Con una memoria unidimensionale, le procedure sono impacchettate una a ridosso dell'altra, senza alcuno spazio di indirizzamento libero fra di loro. Di conseguenza, cambiare la dimensione di una procedura può provocare modifiche nell'indirizzo di altre procedure, anche se non correlate. Inoltre ogni segmento contiene un unico tipo di oggetto, ossia, normalmente, un segmento non conterrà una procedura e uno stack, per esempio, ma l'uno o l'altro. Quindi ogni segmento può avere la protezione appropriata per l'oggetto che contiene.

La segmentazione permette una facile condivisione del codice, dei dati e delle funzioni di un programma proprio perché questi sono organizzati in segmenti. In pratica, un processo viene visto come un insieme di segmenti, che però sono sparsi in memoria. Per questo motivo indirizzi logici e indirizzi fisici non corrispondono.

Ogni indirizzo logico è rappresentato nella forma (si, bi), dove:

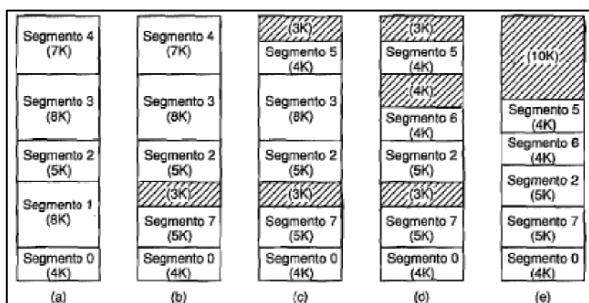
- **si** è l'ID di un segmento
- **bi** è lo scostamento in byte all'interno del segmento

Questa organizzazione permette al programmatore di vedere la memoria come un insieme di segmenti che possono essere di dimensione diversa, e soprattutto dinamica; offre dei vantaggi al programmatore rispetto alla paginazione:

1. Semplifica la gestione di strutture dati che crescono
2. Permette la modifica e la ricompilazione indipendente dei programmi
3. Si presta alla condivisione dei processi
4. Si presta alla protezione

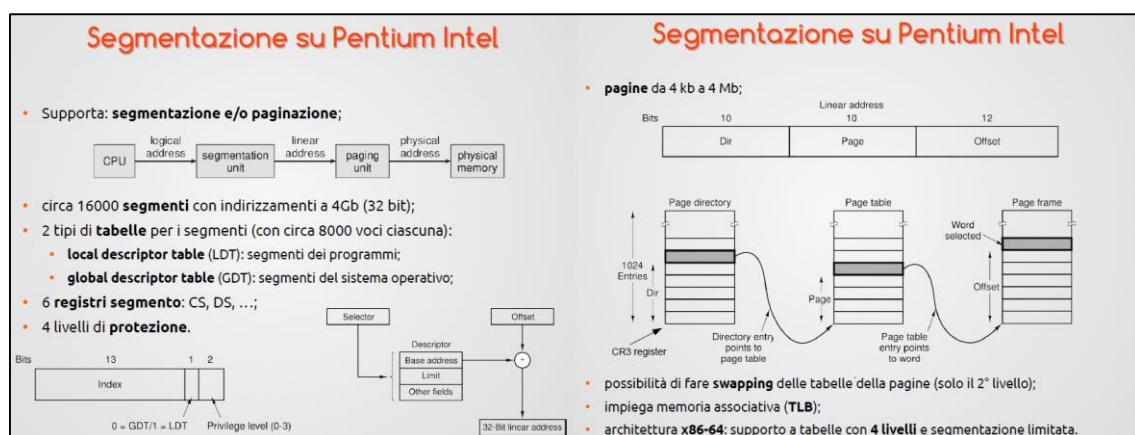
Visto che la dimensione dei segmenti è variabile, allora può essere generata frammentazione esterna. Per questo motivo il sistema deve usare tecniche di riutilizzo della memoria come la first-fit o la best-fit.

Segmentazione pura

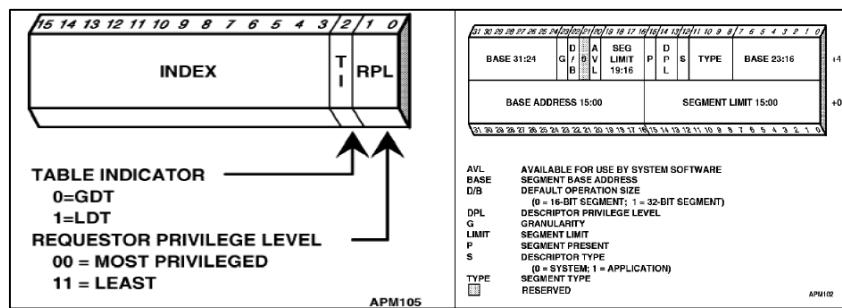


La figura a) mostra un esempio di una memoria fisica che inizialmente contiene cinque segmenti. La figura b) mostra cosa accade se il segmento 1 viene scaricato e al suo posto si mette il 7, più piccolo. Fra il 7 e il 2 si trova un'area non usata; in seguito vi sono altre sostituzioni finché la memoria sarà divisa in un certo numero di parti, alcune contenenti segmenti, altre buchi. A questo fenomeno, detto **checker boarding** o frammentazione esterna che spreca memoria nei buchi, si può porre un rimedio con la compattazione, come visto nella figura e).

Segmenti con paginazione: Pentium Intel



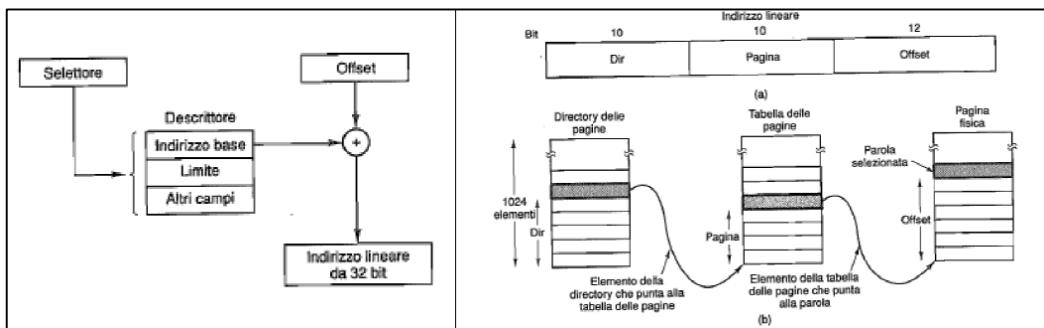
Il Pentium ha 16K segmenti indipendenti, ciascuno dei quali può contenere fino a 1000 miliardi di parole da 32 bit. Il cuore della memoria virtuale del Pentium è dato da due tabelle: la LDT (Local Descriptor Table, tabella dei descrittori locali) e la GDT (Global Descriptor Table, tabella dei descrittori globali). Ciascun programma ha la sua LDT, ma esiste una sola GDT, condivisa fra tutti i programmi del calcolatore. Le LDT descrivono i segmenti locali a ciascun programma, compreso il codice, lo stack, i dati, eccetera, mentre la GDT descrive i segmenti di sistema, compreso lo stesso sistema operativo. Per accedere ad un segmento, un programma Pentium carica dapprima un selettore di quel segmento in uno dei sei registri di segmento della macchina (CS, SS, DS, ES, FS, GS). Durante l'esecuzione, il registro CS contiene il selettore per il segmento del codice ed il registro DS quello per il segmento dei dati. Gli altri registri di segmento sono meno importanti. Ciascun selettore è un numero da 16 bit, come mostrato in Figura 54. Uno dei bit del selettore dice se il segmento è locale o globale (cioè se nella LDT o nella GDT). Tredici altri bit specificano il numero di elemento della LDT o della GDT così che ciascuna di queste tabelle contiene al più 8K descrittori di segmento: Gli altri due bit hanno a che fare con la protezione. Il descrittore 0 è proibito: può essere caricato in un registro di segmento solo per indicare che quel registro di segmento non è disponibile in quel momento. Se usato, causa una trap. Nel momento in cui un selettore viene caricato in un registro di segmento, il corrispondente descrittore viene prelevato dalla LDT o dalla GDT e memorizzato in un registro di microprogramma, in modo che vi si possa accedere velocemente. Un descrittore è formato da 8 byte che includono l'indirizzo di base del segmento, la dimensione e altre informazioni, come illustrato in figura:



Il formato del selettore è stato scelto in maniera intelligente per facilitare la localizzazione del descrittore. Prima si seleziona la LDT o la GDT a seconda del contenuto del bit 2 del selettore, poi si copia il selettore in un registro tampone di microprogramma, e i 3 bit di ordine inferiore vengono messi a 0; infine, vi si somma l'indirizzo della tabella LDT o della GDT per ottenere un puntatore diretto al descrittore. Per esempio, il selettore 72 fa riferimento all'elemento 9 della GDT, che si trova all'indirizzo GDT + 72. Vediamo come, passo passo, una coppia (selettore, offset) venga convertita in un indirizzo di memoria fisica. Non appena il microprogramma sa quale registro di segmento verrà usato, è in grado di trovare il descrittore completo che corrisponde a quel settore nei suoi registri interni; se il segmento non esiste (selettore 0) o in quel momento risulta scaricato, si verifica una trap. In seguito controlla se l'offset va a cadere oltre i limiti del segmento, ed anche in questo caso viene generata una trap. Dal punto di vista logico, nel descrittore ci dovrebbe essere un campo da 32 bit che indica la dimensione del segmento, ma, dal momento che sono disponibili solo 20 bit, si usa uno schema diverso. Se il campo gbit(Granularità) è a 0, il campo Limite rappresenta la dimensione esatta del segmento, fino ad un massimo di 1M. Se il campo è 1, il campo Limite dà la dimensione del segmento in pagine, anziché in byte. La pagina del Pentium è fissata a 4KB, così bastano 20 bit per segmenti che arrivano fino a 2^{32} byte. Supponendo che il segmento sia in memoria e che l'offset cada nell'intervallo dei valori ammessi, il Pentium aggiunge poi all'offset il campo (da 32 bit) Base del descrittore, per formare quello che viene chiamato indirizzo lineare, come mostrato in figura sotto a sinistra.

Se la paginazione è disabilitata (tramite un bit che si trova in un registro globale di controllo), l'indirizzo lineare viene interpretato come indirizzo fisico e viene direttamente spedito alla memoria per la lettura o la scrittura. Così, con la paginazione disabilitata abbiamo uno schema di segmentazione pura, con l'indirizzo di base di ciascun segmento contenuto nel suo descrittore. Incidentalmente, è permessa la sovrapposizione fra segmenti, probabilmente perché controllare che i segmenti siano disgiunti sarebbe troppo complicato e porterebbe via troppo tempo. D'altra parte, se la paginazione risulta abilitata, l'indirizzo lineare viene interpretato come un indirizzo virtuale e tradotto in indirizzo fisico usando le tabelle delle pagine in maniera del tutto simile a quanto visto negli esempi precedenti. La sola complicazione è che con un indirizzo virtuale da 32 bit e una pagina da 4KB, un segmento può contenere un milione di pagine, per cui si usa uno schema di traduzione a due livelli per ridurre la dimensione delle tabelle delle pagine per segmenti di piccole dimensioni. Ciascun programma in esecuzione ha una directory (indice) delle pagine che consta di 1024 elementi da 32 bit, posta ad un indirizzo contenuto in un registro globale. Ciascun elemento in

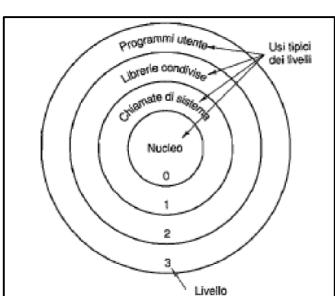
questa directory punta ad una tabella delle pagine, anch'essa contenente 1024 elementi da 32 bit. Gli elementi della tabella delle pagine puntano alle pagine fisiche. Lo schema è illustrato in figura sotto a destra.



In Figura (a) vediamo un indirizzo lineare diviso in tre parti, Dir, Pagina e Offset. Il campo Dir viene usato come indice nella directory delle pagine per individuare un puntatore alla tabella delle pagine opportuna; poi si usa il campo Pagina come indice per accedere alla tabella delle pagine per trovare l'indirizzo della pagina fisica, infine, il campo Offset viene sommato all'indirizzo della pagina fisica per ottenere l'indirizzo della parola o byte che è stata indirizzata. Gli elementi della tabella delle pagine sono da 32 bit ciascuno, di cui 20 bit contengono il numero della pagina fisica. I rimanenti bit contengono i bit che indicano se la pagina è stata modificata o semplicemente usata, mantenuti dall'hardware a vantaggio del sistema operativo; i bit di protezione ed altri bit di utilità. Ciascuna delle tabelle delle pagine ha elementi per 1024 pagine fisiche da 4KB, così ogni tabella delle pagine può indirizzare fino a 4MB di memoria. Un segmento più piccolo di 4MB avrà una directory delle pagine con un singolo elemento, il puntatore alla sua unica tabella delle pagine; in questo modo, l'overhead per i segmenti piccoli è di sole due pagine, contro il milione di pagine che sarebbero state necessarie nel caso di tabella delle pagine ad un solo livello.

Per evitare di eseguire continui riferimenti alla memoria, il Pentium ha un piccolo TLB che traduce direttamente le combinazioni Dir-Pagina usate più di recente in indirizzi di pagina fisica. Solo quando la combinazione richiesta non è presente nel TLB si usa veramente il meccanismo descritto dalla Figura sopra a destra, e si aggiorna il TLB: se le miss su TLB sono rare, le prestazioni sono buone. Vale la pena di notare che se un'applicazione particolare non necessita di segmentazione, ma si accontenta di uno spazio di indirizzamento singolo paginato a 32 bit, ebbene, questo è un modello possibile. Tutti i registri di segmento possono essere caricati con uno stesso selettore, il cui descrittore ha Base = 0 e Limite caricato con il valore massimo. L'offset dell'istruzione sarà poi l'indirizzo lineare, e vi sarà un solo spazio di indirizzi in uso, come accade effettivamente con la paginazione normale.

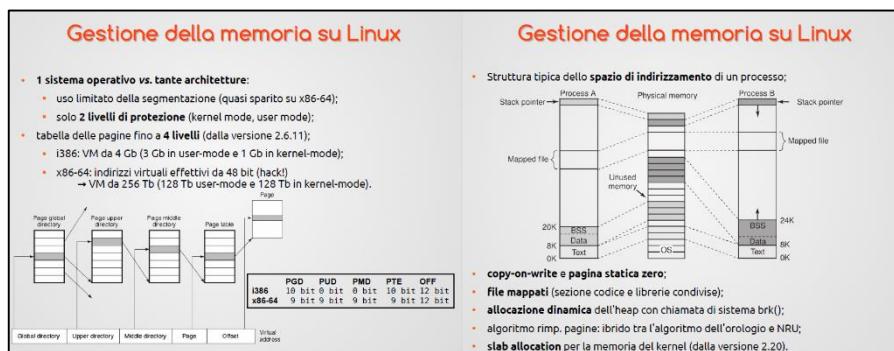
Il Pentium supporta quattro livelli di protezione, con il livello 0 che rappresenta quello con maggiori privilegi e il livello 3 quello con meno. Questi livelli sono mostrati in Figura: ad ogni istante, un programma in esecuzione sta ad un certo livello, indicato da un campo a 2 bit della sua PSW. Anche ciascun segmento nel sistema ha un suo livello.



Finché un programma si limita ad usare segmenti del proprio livello, tutto funziona bene. Sono anche permessi tentativi di accedere a dati di più alto livello, ma tentativi di accedere a dati di livello più bassi sono invece illegali e provocano delle trap. I tentativi di chiamare procedure di un livello diverso (più alto o più basso) sono permessi, ma in un modo controllato con attenzione. Per realizzare una chiamata ad un livello diverso, l'istruzione di CALL deve contenere un selettore, invece di un indirizzo. Questo selettore specifica un descrittore detto call gate (porta di chiamata) che dà l'indirizzo della procedura da chiamare: così non è possibile saltare nel bel mezzo di un segmento di codice arbitrario di livello diverso, e si possono usare solo i punti di accesso (entry point) ufficiali. I concetti di livelli di protezione e di porte di chiamata furono sperimentati per la prima volta in MULTICS, dove venivano visti come anelli di protezione. La Figura 58 suggerisce un uso tipico di questo meccanismo: a livello 0 troviamo il nucleo del sistema operativo, che gestisce l'ingresso/uscita, si occupa della gestione della memoria e di altre questioni critiche. Al livello 1 si trova il gestore delle chiamate di sistema: i programmi utente possono chiamare procedure che si trovano a questo livello per richiedere l'esecuzione delle chiamate di sistema, ma si possono chiamare solo le procedure specifiche che stanno in una lista protetta di procedure. Il livello 2 contiene le procedure di libreria, possibilmente condivise fra molti programmi che si trovano in esecuzione. I programmi utente possono chiamare queste procedure e leggere i loro dati, ma non li possono modificare. Infine, i programmi utente si trovano al livello 3, che ha la protezione minore. Le trap e le interruzioni usano un meccanismo simile alle porte di chiamata: anche loro fanno riferimento a descrittori, piuttosto che ad indirizzi assoluti, e questi descrittori puntano a specifiche procedure da eseguire.

Gestione della memoria su Linux

Non vi è dubbio che l'organizzazione della memoria sia un aspetto fondamentale per le prestazioni dell'intero sistema operativo.



L'implementazione che ne deriva è dunque piuttosto complessa ed estesa, anche a causa del fatto che un sistema operativo moderno richiede numerose funzionalità. Tra queste ricordiamo: la memoria virtuale, che consente di avere processi di dimensione totale superiore alla memoria fisica; l'allocazione e deallocazione efficiente della stessa ai programmi, che implica meccanismi per ridurre la frammentazione; la cache del file system, meglio se di dimensioni variabili secondo la memoria disponibile di volta in volta.

Architettura di base

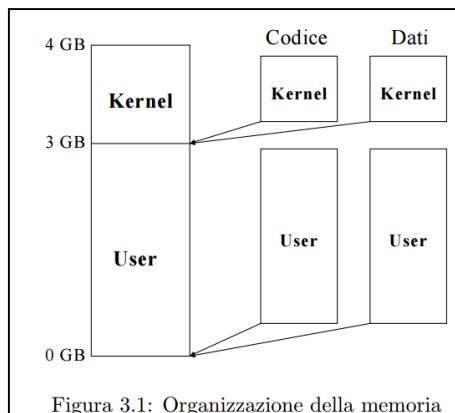
È noto che per realizzare la memoria virtuale vi sono diverse possibili strategie implementative, basate sull'uso della segmentazione e della paginazione. Gli sviluppatori di Linux hanno scelto di usare il meccanismo della segmentazione paginata.

È noto che con questo sistema l'indirizzo virtuale specificato dal programma deve prima attraversare un processo di segmentazione e poi uno di paginazione, per poter essere tradotto in un indirizzo fisico utilizzabile per accedere alle celle di memoria. Nella particolare architettura qui descritta, ossia quella della famiglia x86, è possibile avere il supporto dell'hardware per entrambe le operazioni precedentemente descritte, ottenendo quindi una certa efficienza.

Scendendo più in dettaglio, si può dire che la segmentazione è utilizzata da Linux su questa architettura più per ragioni di implementazione delle protezioni che per una suddivisione dei processi in parti omogenee. Quest'ultima (ossia la divisione dei programmi in parte codice, parte dati, librerie condivise ecc.) è invece ottenuta solamente per via software, e quindi in maniera completamente indipendente dall'architettura.

Segmentazione

I segmenti in Linux sono definiti come si vede dalla figura 3.1.



Il codice di inizializzazione si trova nel file arch/i386/kernel/head.S. Essi vengono resi operativi nel momento del passaggio del processore dal modo reale al modo protetto, istante nel quale si incominciano ad utilizzare i selettori per l'accesso alla memoria.

Poiché essi riguardano tutto il sistema e sono indipendenti dal particolare processo che si sta eseguendo, sono ovviamente posti nella **Global Descriptor Table (GDT)**. Si riporta qui di seguito l'organizzazione di detta tabella, dove ogni riga simboleggia un descrittore di 8 byte: Si notano le righe corrispondenti ai 4 segmenti citati, due per il modo user con DPL (Descriptor Privilege Level) pari a 3, e due per il modo kernel con DPL pari a 0. Sono presenti due righe per ogni modalità in quanto una è utilizzata per gli accessi necessari al caricamento del codice, mentre l'altra è usata per manipolare i dati (ed ha quindi permessi sia di lettura sia di scrittura a differenza della precedente). Inoltre la tabella contiene anche una coppia di descrittori per ogni processo che può girare sul sistema: esse si riferiscono rispettivamente alla Local DescriptorTable (LDT) e al Task State Segment (TSS) di ciascun task. Il posizionamento di questi descrittori nella **GDT** impone dunque che il massimo numero dei processi su architettura x86 sia pari a poco più di 4000, poiché la **GDT** è limitata a 8192 entry; il valore di default è comunque di 512 processi (modificabile tramite una direttiva `#define NR_TASKS 512` nel file `include/linux/tasks.h`). Sono infine presenti alcuni segmenti utilizzati per la gestione delle funzionalità di power saving dei processori x86.

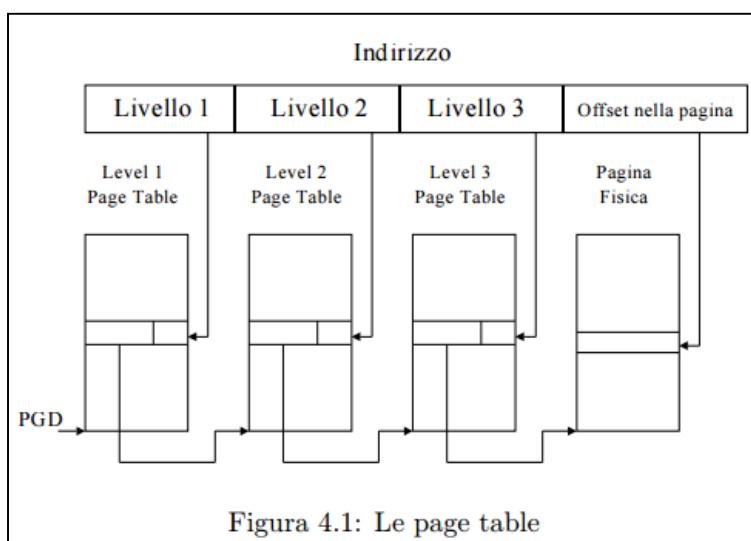
Osservando meglio la figura 3.2, si nota come l'entry relativa alla **LDT** punti normalmente sempre alla stessa struttura dati che contiene un descrittore nullo. Questo è dovuto al fatto che Linux non prevede di utilizzare in maniera sistematica la **LDT** per la gestione dei processi, in quanto si è scelto di usare un unico segmento per i processi, che occupa buona parte di tutta la memoria virtuale disponibile (per essere precisi, in realtà i segmenti sono due, uno per il codice e uno per i dati, sovrapposti). I **segmenti logici** dei programmi (come codice, dati, stack, librerie condivise ecc.) sono invece gestiti via software e i relativi diritti di accesso tramite la protezione offerta a livello di pagine. Questo ha il vantaggio di consentire una portabilità maggiore su altre architetture. La LDT puo` comunque essere modificata tramite l'apposita *system call modify_ldt()* per esigenze particolari (per esempio sull'architettura x86 viene utilizzata dagli emulatori di altri "sistemi operativi", quali il DOS con il DOSEMU, il Windows con Wine e Wabi).

Il TSS è una struttura dati manipolata direttamente dal processore, che questo utilizza per salvare lo stato del processo e per ripristinarlo durante i context switching: è dunque necessario riservarne una diversa per ogni processo.

Paginazione

In Linux l'organizzazione della paginazione è stata virtualizzata, ossia il sistema implementa uno schema generale che viene di volta in volta adattato alle varie architetture.

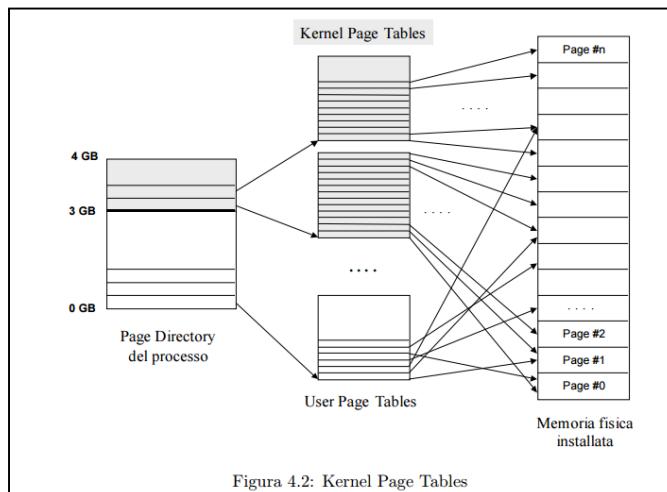
Questo prevede tre livelli di page table: ognuna di esse contiene il numero della pagina corrispondente alla page table del livello successivo. Per tradurre l'indirizzo virtuale in un indirizzo fisico, il processore deve dunque prendere il contenuto di ciascun campo in cui l'indirizzo virtuale è stato suddiviso, convertirlo in un'offset all'interno della pagina fisica che contiene la page table e leggere l'indirizzo della pagina corrispondente alla page table del livello successivo. Questa operazione è ripetuta tre volte, finché si trova l'indirizzo della pagina fisica cui corrisponde l'indirizzo virtuale dato; si utilizza infine l'offset iniziale per accedere all'interno della pagina alla parola di dato richiesta. Nella figura 4.1 si vede uno schema di quanto appena descritto.



Ogni piattaforma su cui Linux è stato portato fornisce le sue particolari macro di traduzione per attraversare le page table: in questo modo il kernel non necessita di conoscere il formato delle stesse o come queste sono organizzate. Il meccanismo funziona talmente bene che Linux può usare lo stesso identico codice di manipolazione delle pagine per tutte le architetture, siano esse per esempio l'Intel x86 che ha due livelli di paginazione oppure l'Alpha o lo Sparc, che invece hanno tre livelli.

In Linux si è scelto di assegnare ad ogni processo una sua page directory propria, il cui indirizzo viene caricato nell'apposito registro del processore ad ogni cambio di contesto.

La parte utilizzata dai processi quando questi operano in kernel mode contiene comunque gli stessi valori per tutti, come è d'altronde logico aspettarsi. Qui di seguito vediamo in dettaglio quanto appena descritto. La page directory (ossia il primo livello di page table) di ogni processo, come si vede dalla figura 4.2, è logicamente suddivisa in due parti.



La prima parte, corrispondente agli indirizzi lineari da 0 GB a 3 GB, contiene le informazioni di traduzione per le pagine utilizzate dal processo quando si trova in “user mode”. La seconda parte, che comprende gli indirizzi lineari da 3 GB a 4 GB, contiene le informazioni usate per la traduzione in “kernel mode” (gli indirizzi corrispondono infatti ai segmenti del kernel). Si può notare la corrispondenza ordinata tra le page table entry e le pagine nella memoria fisica. Le informazioni memorizzate nelle page table hanno infatti la particolarità di far corrispondere ad un certo indirizzo virtuale del segmento del kernel il medesimo indirizzo fisico in memoria. Questa corrispondenza è stabilita, per i primi 4 MB di memoria, nel file `arch/i386/kernel/head.S` in fase di inizializzazione, e completata poi nel file `arch/i386/mm/init.c` per tutte le pagine necessarie a coprire completamente la memoria fisica installata sul particolare sistema su cui il kernel si sta caricando.

Questa scelta è causata dal fatto che è il kernel che gestisce le pagine dei processi in modo user, e se esso dovesse gestire anche le pagine per se stesso, questo comporterebbe una notevole complicazione, se non addirittura l'impossibilità di adempiere al compito.

Page allocation

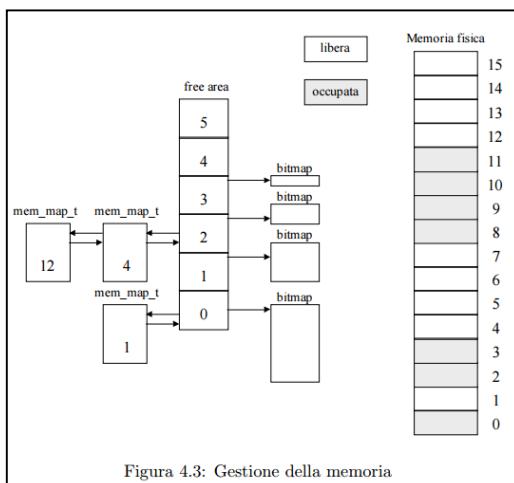
In un sistema le pagine fisiche in memoria devono adempiere a svariati compiti: per esempio poter essere allocate e deallocate, contenere le strutture dati del kernel, e le page table stesse. Il meccanismo di allocazione e deallocazione è fondamentale, in quanto risulta critico per l'efficienza del sottosistema di gestione della memoria virtuale.

L'organizzazione di Linux prevede che tutte le pagine fisiche siano descritte da una struttura dati di nome `mem_map` (definita in `mm/memory.c`), che è un vettore di strutture `mem_map_t` inizializzata durante il caricamento del sistema, nel file `mm/page alloc.c`, funzione `free area init()`. Ogni `mem_map_t` descrive una singola pagina fisica. I campi più importanti sono:

- `count` numero degli utenti della pagina: se è maggiore di uno, allora la pagina è condivisa tra più processi;
- `age` riporta l'età associata alla pagina, ed è usato per decidere se questa è una buona candidata per essere eliminata o portata nello swap;
- `map_nr` il numero della pagina fisica che questa struttura descrive.

La struttura è definita nel file `include/linux/mm.h`. Esiste inoltre un vettore `free area`, utilizzato dalla routine di allocazione delle pagine per trovare quali siano libere. Ogni elemento di questo vettore contiene una mappa di bit tramite la quale si determinano quali siano i blocchi di pagine liberi, oltre che la testa delle liste che descrivono i blocchi da una pagina, i blocchi di due pagine, i blocchi di quattro e così via, in potenze di due. Tutte le pagine libere vengono accodate in liste puntate dal vettore `xfree_area`. Di seguito si illustrano graficamente le strutture dati descritte.

Linux usa un algoritmo particolare per allocare e deallocare le pagine, detto Buddy Algorithm. Innanzitutto la routine alloca solamente blocchi di pagine (di 1, 2, 4, 8 pagine, come descritto in precedenza): essa per prima cosa cerca i blocchi di dimensione corrispondente alla richiesta, e se non ne trova, cerca quelli di dimensione superiore (ossia il doppio), finché tutta la free area è stata passata in rassegna o un blocco di pagine è stato trovato. Se il blocco è più largo della richiesta, esso viene suddiviso finché diventa della dimensione giusta. Grazie alle dimensioni dei blocchi una il doppio dell'altra, quest'operazione è semplice, infatti basta continuare a suddividere i blocchi a metà. I blocchi liberi così generati sono aggiunti alla coda appropriata e il blocco allocato è ritornato al chiamante.



La deallocazione tende a frammentare la memoria in piccoli blocchi, ma la routine cerca di ricombinare le pagine in blocchi più grossi quando ciò è possibile. Anche in questo caso le particolari dimensioni scelte facilitano notevolmente l'operazione. Quando un blocco viene deallocated il codice verifica per prima cosa se quello adiacente è libero: in tal caso li unisce, e questo procedimento si ripete ricorsivamente, fino a che ciò è possibile; così facendo, i blocchi liberi sono i più grossi che la memoria consente di avere. Per maggiori dettagli si può visionare il file `mm/page_alloc.c`, ed in particolare la funzione `free_pages_ok()`.

L'organizzazione della memoria di un processo

Ad ogni processo nel sistema Linux associa una struttura dati di nome `task_struct` (vedi `include/linux/sched.h`), contenente a sua volta un campo `mm` che punta a una `mm struct` (vedi `include/linux/sched.h`), la quale contiene dei puntatori ad un certo numero di strutture di tipo `vm_area_struct` (file `include/linux/mm.h`).

Ogni `vm_area_struct` descrive un'area di memoria virtuale destinata a qualcosa, e contiene tra le altre cose l'indirizzo di inizio e fine, i permessi di accesso da parte del processo e un insieme di operazioni eseguibili sulla memoria, contenute in una `vm_operations_struct` (file `include/linux/mm.h`), che, se definite, Linux utilizza quando deve operare su quella particolare porzione di memoria.

Per avere un esempio di un possibile impiego di dette operazioni, si pensi ad un processo che accede a una zona di memoria la cui corrispondente pagina fisica non è presente: questo comporta un'eccezione di page fault; la routine riconosce la situazione e decide di usare l'operazione associata di nome `nopage`.

Nel caso l'area di memoria si riferisca ad un'immagine eseguibile, `nopage` sarà stata precedentemente inizializzata a puntare ad un'apposita routine in grado di caricare la corrispondente pagina dal file.

Quando un eseguibile viene mappato in memoria per essere successivamente eseguito, viene generato un insieme di `vm_area_struct`, ognuna delle quali rappresenta una parte dell'immagine eseguibile: il codice vero e proprio, le variabili inizializzate, quelle non inizializzate e così via.

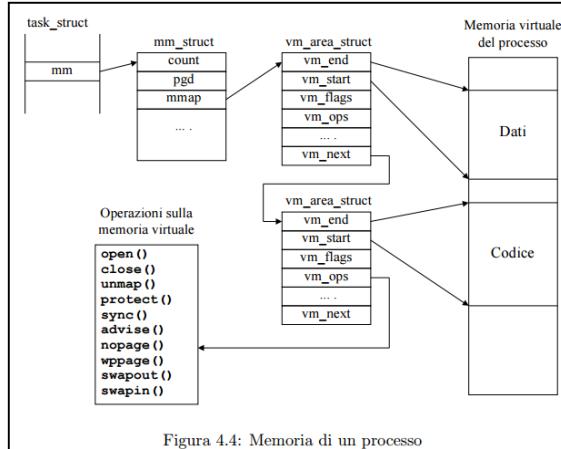


Figura 4.4: Memoria di un processo

Linux supporta un certo numero di operazioni standard sulla memoria virtuale, come si può vedere dalla figura 4.4, e al momento della creazione di ogni `vm_area_struct`, associa loro l'insieme corretto e specifico di operazioni.

Il page fault

La routine del page fault (corrispondente alla trap n. 14 dei sistemi x86), è il principale strumento di gestione della memoria. Infatti Linux fa un uso estensivo dell'eccezione del page fault per assolvere a svariati compiti, e questa organizzazione, come si vedrà, permette anche una notevole efficienza. Il codice sorgente della routine si trova in `arch/i386/mm/fault.c`.

Essa viene richiamata tramite un'eccezione scatenata dal processore quando questo rileva tramite le tabelle di paginazione che la pagina a cui si cerca di accedere non è presente in memoria, oppure l'accesso non è conforme con i permessi previsti.

In tutti i sistemi la routine di page fault per prima cosa ricava in qualche modo l'indirizzo virtuale che ne ha causato l'esecuzione, la presenza o meno della pagina e il tipo di accesso tentato (lettura o scrittura e a quale livello di privilegio). Nei sistemi x86 in particolare, la prima informazione si ricava dal registro CR2 del processore, mentre la seconda da un apposito codice di errore posto sullo stack.

Per poter effettuare una corretta gestione del page fault è necessario che Linux per prima cosa individui la `vm_area_struct` che rappresenta l'area di memoria di cui l'indirizzo virtuale fa parte. Poiché la velocità di ricerca in queste strutture dati è critica per ottenere una buona efficienza a tempo di esecuzione, queste sono organizzate in una struttura ad albero particolare, detta AVL tree (Adelson-Velskii e Landis). Essa è in pratica un albero binario, in cui sono mantenute alcune proprietà, e che permette di effettuare la ricerca di una chiave nell'albero in un tempo $O(\log n)$ al posto del comune $O(n)$ di una normale lista linkata, dove n è ovviamente il numero di aree di memoria virtuale del task (generalmente 6 ma in alcuni casi si possono raggiungere le 3000). Questa caratteristica è dovuta alle proprietà dell'albero, che sono qui brevemente riassunte. Ogni nodo ha tre campi fondamentali: puntatore al figlio sinistro, puntatore a quello destro, e altezza. Nell'albero valgono sempre queste regole:

- L'altezza di un nodo è pari al massimo di quella dei suoi due nodi figli aumentata di una unità;
- La differenza tra l'altezza del figlio sinistro e quella del figlio destro non eccede l'unità;
- Per ogni nodo nell'albero del figlio di sinistra, le chiavi contenute sono minori o uguali di quelle del padre e analogamente per il figlio di destra

Queste proprietà sono mantenute richiamando la funzione di ribilanciamento dell'albero (`avl_rebalance`) ogni volta che questo viene modificato. Per ulteriori dettagli, si consiglia di vedere il file `mm/mmap.c`.

Una volta che è stata effettuata la ricerca in questo albero, si può presentare il caso che non ci sia nessuna struttura dati corrispondente all'indirizzo virtuale: questo significa che il processo ha tentato di accedere ad un indirizzo illegale; il kernel invia dunque al task un segnale di SIGSEGV (Segmentation fault), e se il processo non ha definito un handler per questo segnale, esso viene terminato. Se invece l'indirizzo è valido, occorre verificare se il tipo di accesso è compatibile con quelli permessi per quell'area di memoria. Ne caso in cui non sia così, si ricade nella condizione precedente, e si segnala al processo un errore di memoria.

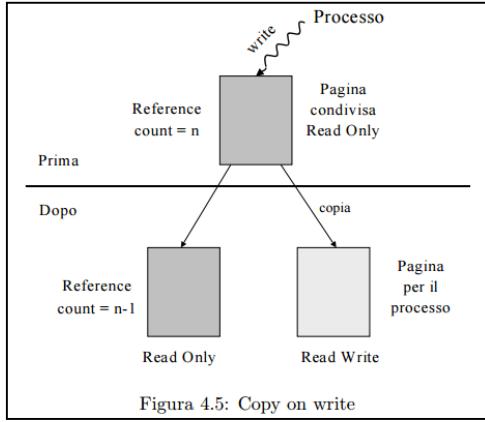


Figura 4.5: Copy on write

Se tutti i controlli hanno dato esito positivo, Linux deve ora servire il page fault. Per prima cosa determina se la pagina è in memoria fisica o no. Per far ciò si testa il bit di presenza, determinando se la pagina manca perché è stata portata nello swap o solamente perché non è mai stata creata. Nel primo caso, se è stata associata un'operazione specifica al puntatore swapin nella `vm_operations_struct` della `vm_area_struct` la si esegue, altrimenti si usa la funzione `swap_in()` di default (si veda la do_swap_page() nel file mm/memory.c).

Nel secondo caso, se tramite il puntatore nopage della struttura vm ops è stata specificata una routine particolare per quest'area di memoria, la si richiama, altrimenti si usa una procedura di default (si veda la funzione `do_no_page()` in mm/memory.c).

Si noti che in entrambi i casi si cerca di condividere la pagina con una esistente se possibile, al fine di allocare una nuova pagina solo se strettamente necessario. Inoltre, se la pagina è condivisa, viene posta comunque read-only.

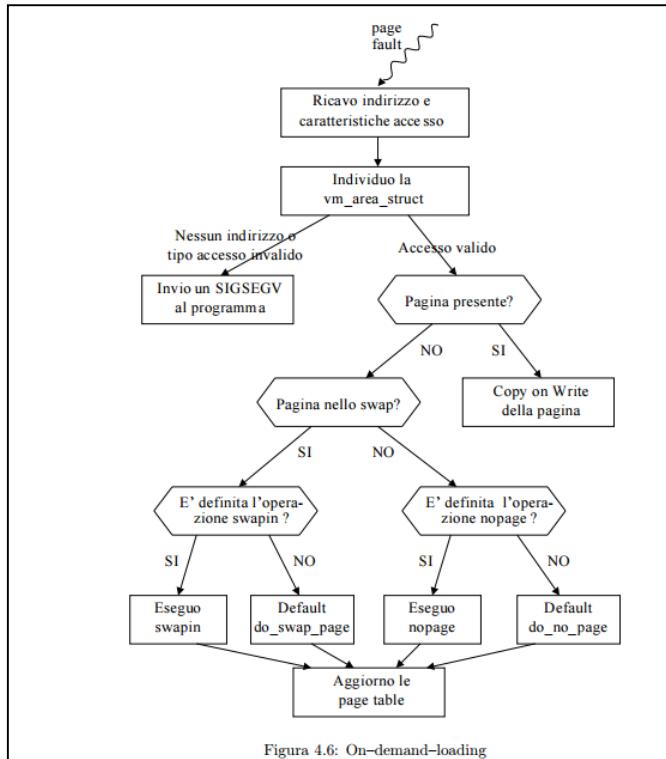
Ora che la pagina è presente in memoria, si aggiorna la rispettiva entry nella page table cosicché il processo possa riprendere la sua naturale esecuzione. Se invece la pagina è presente in memoria fisica ed è stato tentato un accesso in scrittura, significa che essa era protetta dalla scrittura. Poiché i controlli sui permessi sono già stati eseguiti, vuol dire che la pagina è scrivibile ma condivisa per evitare specie di risorse. Di conseguenza se la pagina ha un reference counter maggiore di uno, si ricopia questa in una nuova, si decrementa il contatore della vecchia e si assegna la nuova pagina al processo che ne ha richiesto la scrittura. La situazione è schematicamente illustrata nella figura 4.5.

Quest'ultimo meccanismo è noto come "copy on write" e permette un notevole risparmio di pagine, soprattutto quando vengono lanciate più copie dello stesso eseguibile. In entrambi i casi è ovviamente necessario modificare la page table, perciò devono essere anche aggiornate le loro copie nelle varie cache del sistema (per esempio nel TLB).

On-demand-loading

Il meccanismo di on-demand-loading è molto utile per avere un efficiente esecuzione dei programmi. Quando si richiede l'esecuzione di un'immagine (ossia si lancia un programma), il contenuto della immagine eseguibile deve essere portato nello spazio di indirizzamento virtuale dei processi. Analoga cosa accade nel caso delle librerie condivise che devono essere utilizzate. Il file eseguibile però non viene copiato subito nella memoria fisica, bensì è semplicemente aggiunto nello spazio della memoria virtuale del processo: vengono cioè create delle `vm_area_struct` (la cui organizzazione è già stata vista in precedenza), che ricalcano la suddivisione del programma in codice, dati inizializzati, variabili non inizializzate, stack e così via. Questa aggiunta delle varie parti del file allo spazio di indirizzamento del processo è conosciuta come operazione di "memory mapping".

Non appena il task viene poi selezionato dallo scheduler ed entra in esecuzione, tenta di accedere alla prima pagina della sua immagine eseguibile. Poiché questa non è presente in memoria, l'hardware invoca la routine del page fault che provvede a caricare la pagina dal file del disco, con la sequenza di operazioni vista prima. L'esecuzione quindi può procedere finché si incontrerà un'altra pagina non presente. Questo meccanismo è particolarmente efficiente in quanto carica in memoria solamente le parti di codice che sono effettivamente eseguite dal programma, consentendo un certo risparmio di memoria.



Page cache

Al fine di ottimizzare le prestazioni del sistema, Linux mantiene una page cache che contiene quelle utilizzate più recentemente. Questo meccanismo rende più efficiente l'on demand loading appena visto nel caso delle immagini eseguibili.

Quando una pagina deve essere letta da un file mappato in memoria per prima cosa si interroga la page cache. Se in essa è presente la pagina cercata, è sufficiente restituire al page fault handler semplicemente il puntatore alla pagina; in caso contrario, essa verrà letta dal disco, assieme ad alcune pagine successive, tramite un'operazione di read ahead, ponendo le pagine non richieste nella page cache, pronte per essere rese disponibili al processo se questo sta per esempio accedendo ad esse in modo sequenziale. Col tempo, eseguendo sempre nuovi programmi, la page cache crescerà; quando non vi sarà più posto per le nuove pagine, quelle più vecchie e meno accedute saranno rimosse.

Questo meccanismo permette ai programmi di essere lanciati e ed avere prestazioni discrete anche se il numero di pagine fisiche disponibili è basso. Se necessario Linux è comunque in grado di diminuire le dimensioni di questa page cache.

Swapping

Come in tutti i sistemi di memoria virtuale, quando la memoria fisica disponibile diventa scarsa, Linux deve cercare di liberare delle pagine fisiche. Questa operazione viene affidata ad un apposito demone del kernel, chiamato kswapd. Esso è uno speciale tipo di processo, cioè un thread del kernel, che non ha memoria virtuale e gira in kernel mode: in questo modo può dunque manipolare direttamente la memoria fisica. Inoltre sempre a questo demone è affidato il compito di assicurarsi che ci siano abbastanza pagine libere in memoria per mantenere il sistema efficiente.

Il kswapd viene attivato dal processo del kernel stesso durante il caricamento, e generalmente è in stato dormiente, in attesa di essere risvegliato periodicamente dal kernel swap timer. Ogni volta che questo timer scade, il demone verifica se il numero delle pagine libere sta diventando troppo esiguo. A tal fine si avvale di due variabili, *free_pages_high* e *free_pages_low*, per decidere se liberare delle pagine. Fintanto che il numero di queste è maggiore di *free pages high*, non fa niente. Se invece il numero di pagine è inferiore a *free_pages_high* o peggio a *free_pages_low*, il thread cerca di diminuire le pagine usate dal sistema per esempio riducendo la dimensione del buffer cache e della page cache oppure portando nello swap alcune pagine di memoria. Le due variabili menzionate precedentemente servono per dare un'indicazione al kswapd su quante pagine alla volta deve cercare di liberare prima della successiva riattivazione. In genere il thread continua ad utilizzare lo stesso metodo (riduzione delle cache o swap delle pagine) usato la volta precedente se si è rivelato una buona scelta. Inoltre, se rileva che il numero di pagine è inferiore a *free_pages_low*, dimezza il periodo del timer che lo deve risvegliare.

Il buffer cache e la page cache sono in genere buone riserve a cui attingere pagine da liberare: spesso infatti ne contengono di non strettamente necessarie, che consumano solo la memoria del sistema. Mediamente eliminare pagine da queste cache non richiede scritture sui device fisici (al contrario dell'operazione di swap), ed inoltre ciò non ha effetti troppo dannosi sul sistema se non quello di rendere gli accessi ai device fisici e ai file mappati in memoria un po' più lenti. Se comunque lo scarto delle pagine è fatto con una politica equa, tutti i processi subiranno lo stesso effetto. Per mettere in atto l'operazione si esaminano i blocchi di pagine in maniera ciclica, con il noto algoritmo dell'orologio, che permette di passare in rassegna tutte le pagine prima di ritornare su una precedentemente visitata.

Si noti che poiché nessuna delle pagine che vengono liberate faceva parte della memoria virtuale di alcun processo (esse erano al massimo cached pages), nessuna tabella di paginazione necessita di aggiornamenti.

Diverso è il caso delle pagine a cui i processi accedono tramite la memoria virtuale. Se infatti una di queste viene spostata dalla memoria nel device di swap, la corrispondente tabella di paginazione deve essere aggiornata, ponendo a falso il bit di presenza, e indicando nei restanti bit il device di swap e l'offset nello stesso tramite il quale la pagina potrà essere in seguito recuperata quando questa necessiterà di essere referenziata.

Per decidere quali pagine portare nello swap, un questo caso Linux usa un meccanismo di aging: ogni pagina ha un contatore (che fa parte della struttura *mem_map_t*) che permette al demone di swap di capire se la pagina è una buona candidata o no per l'operazione. Questo contatore viene incrementato di tre unità fino ad un massimo di 20 quando si accede alla pagina, e decrementato di una ad ogni attivazione del demone. In questo modo si seleziona dunque una pagina vecchia, e se la pagina è “dirty”, ossia è stata accedita in scrittura, è necessario copiare effettivamente i suoi dati sul device di swap.

Se la pagina si trova in una regione di memoria virtuale per cui è stata definita una operazione particolare per operare lo swap (funzione *swapout* nella struct *vm_operations_struct* contenuta nella *vm_area_struct*), essa viene richiamata, altrimenti il demone di default alloca una pagina nello swap e vi copia la vecchia, aggiornando di conseguenza la page table interessata. La pagina vecchia viene poi resa libera accodandola nella struttura free area.

Se la pagina non è “dirty”, si può invece liberare senza ulteriori copie, in quanto ne esiste sicuramente già una salvata da qualche parte. Ovviamente questo metodo è molto più efficiente che portare nello swap interi processi, in quanto cerca di mantenere un equilibrio nel sistema con il minimo dispendio di risorse.

Si noti infine che invece le pagine contenenti il codice binario dei programmi eseguibili, essendo per loro natura a sola lettura, non vengono mai spostate sul device di swap, bensì sempre ricaricate dall'immagine su disco o dal buffer cache quando ciò è richiesto. Questo consente di occupare lo spazio destinato allo swap solamente con i dati delle applicazioni, ottenendo così un'efficienza notevole.

Gestione dei File

Cos'è un file system

Un file system (abbreviazione: FS) indica informalmente un meccanismo con il quale i file sono posizionati e organizzati su un dispositivo di archiviazione o una memoria di massa, come un disco rigido o un CD-ROM e, in casi eccezionali, anche sulla RAM.

Più formalmente, un file system è l'insieme dei tipi di dati astratti necessari per la memorizzazione (scrittura), l'organizzazione gerarchica, la manipolazione, la navigazione, l'accesso e la lettura dei dati. Di fatto, alcuni file system (come l'NFS) non interagiscono direttamente con i dispositivi di archiviazione.

I file system possono essere rappresentati sia graficamente tramite file browser sia testualmente tramite shell testuale. Nella rappresentazione grafica (GUI) è generalmente utilizzata la metafora delle cartelle che contengono documenti (i file) ed altre sottocartelle.

Mantenimento delle informazioni

Tutte le applicazioni su calcolatore hanno bisogno di memorizzare e rintracciare informazioni. Durante l'esecuzione, un processo può memorizzare una quantità limitata di informazioni nello spazio degli indirizzi ad esso concesso e, comunque, la capacità di memoria è limitata dalla dimensione dello spazio degli indirizzi virtuali. Per alcune applicazioni questo spazio riservato è adeguato, ma per altre, come prenotazioni aeree, applicazioni bancarie o archivi di grandi aziende, è spesso troppo limitato.

Un secondo problema nel mantenere le informazioni all'interno dello spazio degli indirizzi di un processo è che le informazioni vengono perse quando il processo termina; per molte applicazioni (ad esempio, per le basi di dati) le informazioni devono essere conservate per settimane, mesi o a volte per sempre, per cui è inaccettabile che esse possano essereperse quando il processo termina. Inoltre le informazioni non devono andare perse quando si guasta il computer e si interrompono i processi in corso in quell'istante.

Un terzo problema consiste nel fatto che frequentemente più processi devono poter accedere allo stesso tempo ad una parte delle informazioni. Se abbiamo un elenco telefonico in linea memorizzato nello spazio degli indirizzi di un processo, solo quel processo può accedervi; il modo per risolvere questo problema consiste nel rendere le informazioni stesse indipendenti da ogni altro processo. La soluzione a questo tipo di problemi viene risolta con i file; i dati vengono memorizzati nei dischi rigidi dentro "contenitori" chiamati file. I file devono essere permanenti e quindi indipendenti dalla chiusura di un processo. Possono essere eliminati soltanto da un'azione esplicita dell'utente.

Denominazione dei file

I file sono un meccanismo di astrazione dei dati. Probabilmente la parte più importante ed allo stesso tempo critica dei meccanismi di astrazione è il modo in cui gli oggetti gestiti sono denominati. Quando un processo crea un file, tutti gli altri processi possono utilizzarlo facendo riferimento al nome. Ogni sistema ha regole precise di denominazione dei file; ad ogni modo la maggior parte dei sistemi consentono la ridenominazione dei file con stringhe che vanno da 1 a 255 caratteri. Una differenza sostanziale la troviamo però fra file system di sistemi UNIX e file system di sistemi MS-DOS. Nel primo caso si fa differenza fra lettere maiuscole e minuscole ed i nomi "A" ed "a" fanno riferimento a due file diversi. Nel secondo caso non si fa differenza e con "a" o "A" indichiamo lo stesso file. Molti SO dividono i nomi dei file in due parti separate da un punto; la parte dopo il punto viene chiamata estensione del file. In molti casi, nei sistemi UNIX, l'estensione del file è solo una convenzione, ma diventa fondamentale per applicativi che possono prendere in input solo determinati tipi di file (ad esempio un compilatore C, prende solo file xxxx.c). Nei sistemi WINDOWS invece le estensioni assumono un aspetto fondamentale nell'utilizzo corretto del SO stesso.

Struttura dei file

I file possono essere strutturati in molti modi; sia WINDOWS che UNIX utilizzano lo stesso approccio. In un primo caso, il SO tratta i file come se fossero un insieme di bit garantendo la massima flessibilità. I programmi utente possono inserire nei file qualsiasi cosa senza essere minimamente influenzati dal sistema operativo. In un secondo caso la sequenza di bit rappresentante un file viene raggruppata in record; ogni operazione di lettura restituisce un record ed ogni operazione di scrittura sovrascrive un record o ne aggiunge un altro. Un ultimo caso identifica i file

come un albero di record di svariata lunghezza, ognuno dei quali contiene un campo chiave. In questo modo possono essere effettuati dei salti andando a prendere semplicemente i file con il campo chiave richiesto.

Tipi di file

Diversi sistemi operativi trattano molti tipi di file. Sia UNIX che WINDOWS trattano due tipi di file fondamentali, file regolari e file directory. I primi sono quelli che contengono tutte le informazioni dell'utente mentre i file directory sono tutti quelli che conservano la struttura del file system. Generalmente i file regolari sono di tipi ASCII o binario. I file di tipo ASCII sono sequenze di caratteri e presentano l'enorme vantaggio di poter essere editati con qualsiasi editor di testo; molti programmi utilizzano in input e in output file ASCII. Tutti i file che non sono ASCII sono di tipo binario; la loro visualizzazione è un listato apparentemente incomprensibile ma facilmente interpretabile dalle macchine. Questo tipo di file ha una struttura interna ben definita, decifrabile dalle applicazioni che ne fanno utilizzo; inoltre i file di tipo binario possono essere sottodivisibili in file archivio, formati da procedure e librerie, ed in file eseguibili. Ogni sistema operativo ha un suo formato eseguibile. Alcuni sistemi operativi riescono a leggere più tipi di formati eseguibili.

Accesso ai file

I primi sistemi operativi potevano avere accesso ai file esclusivamente in maniera sequenziale. Tale sistema era l'ideale se pensiamo che le unità di memoria erano dei nastri magnetici che potevano essere riavvolti e riutilizzati allo stesso modo infinite volte. Con l'introduzione dei dischi diventò possibile leggere i bit o i record di ogni file in maniera non ordinata; nascono così i file ad accesso casuale. Questo tipo di file è fondamentale per tutte le applicazioni.

Per far partire la lettura dei dati dal punto giusto si utilizzano due metodi: nel primo metodo ogni operazione di lettura da l'indirizzo di memoria dal quale bisognerà andare a leggere i file. Con il secondo metodo invece, una procedura speciale di nome **seek**, dà la possibilità di scegliere il punto dal quale iniziare la lettura. Dopo di che si procede con la lettura sequenziale dal punto stabilito. In tutti i sistemi operativi moderni i file utilizzati sono ad accesso casuale.

L'organizzazione dei file viene scelta in base alle caratteristiche del dispositivo di I/O che si usa, in modo tale da fornire un accesso efficiente. Ad esempio, un hard disk può accedere direttamente, mediante l'indirizzo, a qualunque record, mentre un drive a nastro può accedere al record solo in modo sequenziale.

- **Organizzazione sequenziale dei file:** Nell'organizzazione sequenziale dei file le informazioni sono memorizzate in ordine crescente o decrescente in base al campo chiave. Di conseguenza l'elaborazione di queste informazioni supporta solo due operazioni:
 - Legge l'informazione (record) successiva (o precedente)
 - Salta l'informazione (record) precedente (o successiva)Un file ad accesso sequenziale viene utilizzato nelle applicazioni se i suoi dati possono essere pre-ordinati convenientemente in ordine crescente o decrescente.
- **Organizzazione diretta dei file:** L'organizzazione diretta dei file fornisce efficienza e convenienza perché permette di accedere alle informazioni(record) in ordine casuale. Di conseguenza l'elaborazione di queste informazioni permette di leggere o scrivere byte senza alcun ordine particolare.
- **Organizzazione indicizzata dei file:** Nell'organizzazione indicizzata dei file un indice aiuta a determinare la posizione di un record a partire dal valore della sua chiave.

Nell'organizzazione indicizzata pura, esiste un indice per ogni record. Un indice di un file è costituito dalla coppia (valore della chiave, indirizzo del disco). Per accedere a un record con chiave k, viene trovato l'elemento indice contenente k tramite la ricerca per indice, e si utilizza l'indirizzo del disco annotato nell'elemento trovato per accedere al record.

L'organizzazione sequenziale indicizzata è un'organizzazione ibrida che combina gli elementi delle organizzazioni dei file indicizzata e sequenziale: esiste un indice per ogni sezione del disco. Per accedere ad un record, si cerca un indice che punta ad una sezione del disco che può contenere il record. Poi si effettua la ricerca sequenziale dei record in questa sezione del disco per trovare il record desiderato. La ricerca ha buon esito se il record è presente nel file; altrimenti avrà esito negativo. Questa organizzazione richiede un indice

molto più piccolo rispetto all'indicizzazione pura poiché l'indice contiene elementi solo per alcuni valori della chiave.

Attributi dei file

Ogni file oltre ad un nome ed ai dati contiene una serie di caratteristiche come la data di creazione, la dimensione etc. Queste caratteristiche, assegnate dal SO, si chiamano attributi di un file. Gli attributi che il SO può attribuire ad un file sono numerosissimi; al momento però non esistono sistemi operativi che li utilizzano tutti.

Tali attributi sono anche conosciuti come **metadati**, ovvero dei dati usati per accedere ai file (dati di controllo).

Operazioni sui file

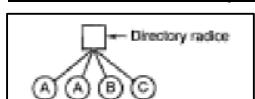
Ogni file nasce con lo scopo di conservare dati ed utilizzarli in momenti successivi. Su di essi quindi deve essere possibile effettuare operazioni per leggerli e scriverli. Di seguito sono indicate le operazioni più frequenti. Il SO può offrire chiamate di sistema per creare, scrivere, leggere, spostare, cancellare e troncare un file.

- **CREATE**: crea un file vuoto e setta degli attributi;
- **DELETE**: chiamata di sistema che elimina un file specificato;
- **OPEN**: prima di utilizzare un file esso deve essere letto. La chiamata OPEN permette al SO di avere a disposizione tutti i dati per la lettura e l'apertura del file;
- **CLOSE**: chiude un file quando tutti i processi in corso non ne fanno più utilizzo risparmiando spazio nella tabella interna;
- **READ**: legge sequenze di bit/byte da un file;
- **WRITE**: i dati vengono scritti alla posizione corrente all'interno del file. Se la posizione corrente è l'ultima disponibile il file viene esteso in dimensione e si procede alla scrittura;
- **APPEND**: chiamata restrittiva del processo WRITE. Serve a scrivere esclusivamente alla fine di un file;
- **SEEK**: utilizzata nei file ad accesso casuale serve per riposizionare il puntatore alla posizione richiesta per l'inizio della lettura;
- **GET ATTRIBUTES**: funzione utilizzata dai SO per estrarre gli attributi di un file;
- **SET ATTRIBUTES**: imposta o modifica gli attributi di un file;
- **RENAME**: rinomina un file;

Directory

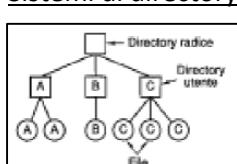
Un elemento molto importante dei file system sono le directory. Una directory contiene le informazioni relative a un gruppo di file. Ogni elemento in una directory contiene gli attributi di un file, come il nome, il tipo, la dimensione, i permessi, ecc.

Sistemi di directory a singolo livello



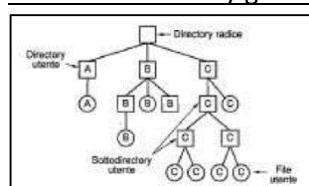
La forma più semplice di sistema di directory è quello di avere un'unica directory che contiene tutti i file; talvolta è chiamata directory radice, ma siccome è l'unica, il nome non è molto importante.

Sistemi di directory a due livelli



Per evitare i conflitti causati da diversi utenti che scelgono gli stessi nomi di file per i loro file personali, il passo successivo è stato quello di dare ad ogni utente una directory privata. In questo modo, i nomi scelti da ogni utente non interferiscono coi nomi scelti da un altro, e non ci sono problemi se lo stesso nome è presente in due o più directory. In questo schema è implicito che quando un utente cerca di aprire un file, il sistema sa di quale utente si tratti in modo da sapere in che directory cercare. Quando questo sistema viene implementato nella sua forma più semplice, gli utenti possono solo accedere ai file nella loro directory.

Sistemi di directory gerarchici



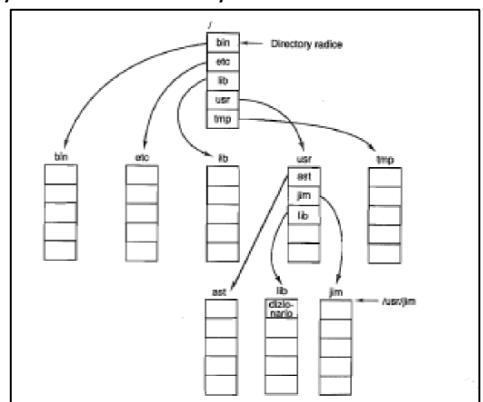
La gerarchia a due livelli elimina i conflitti sui nomi, ma non è soddisfacente per utenti con un grande numero di file; questo capita persino su un PC a singolo utente. Ciò che è necessario è una gerarchia generale (ad esempio un albero di directory). Con questo approccio ogni utente può avere tante directory quante sono necessarie, così i suoi file possono essere raggruppati in modo naturale. Dare agli utenti la possibilità di creare un

numero arbitrario di sottodirectory, fornisce un mezzo di strutturazione molto potente per organizzare il loro lavoro; per questa ragione, quasi tutti i moderni file system sono organizzati in questo modo.

Path Name

Quando il file system è organizzato come un albero di directory; è necessario un qualche modo per specificare i nomi dei file; si usano di solito due metodi differenti. Nel primo metodo ad ogni file viene dato un path name assoluto (nome assoluto) che consiste nel cammino dalla directory radice al file.

L'altro tipo di nome è il path name relativo, che si usa congiuntamente al concetto di directory di lavoro (**working directory**; anche detta directory corrente). Un utente può definire una directory come la directory di lavoro corrente, nel qual caso tutti i path name che non iniziano dalla directory radice sono considerati relativi alla directory di lavoro. Alcuni programmi hanno bisogno di accedere a un file specifico a prescindere dalla directory di lavoro; in quel caso, essi dovranno usare sempre path name assoluti. Ogni processo ha la sua propria directory di lavoro, così quando un processo cambia la sua directory di lavoro e più tardi termina, non ci sono altri processi coinvolti in questo cambiamento, e non ne rimane traccia nel file system. In questo modo è sempre perfettamente sicuro per un processo cambiare la sua directory di lavoro ogniqualvolta ciò è conveniente. D'altra parte se una procedura di libreria cambia la directory di lavoro e non ritorna dove era al momento della sua terminazione, il resto del programma non può funzionare correttamente.



Operazioni sulle directory

1. **CREATE.** Crea una directory vuota, a parte punto e punto punto che sono inseriti automaticamente dal sistema (o, in pochi casi, dal programma *mkdir*)
2. **DELETE.** Cancella una directory. Solo una directory vuota può essere cancellata
3. **OPENDIR.** Le directory si possono leggere; per esempio, per elencare tutti i file di una directory, un programma apre la directory per leggere i nomi di tutti i file che essa contiene. Prima che una directory possa essere letta, essa deve essere aperta, analogamente alla apertura e lettura dei file
4. **CLOSEDIR.** Quando una directory è stata letta, dovrebbe essere chiusa per liberare lo spazio della tabella interna
5. **REaddir.** Questa chiamata restituisce il prossimo elemento in una directory aperta.
6. **RENAME.** Sotto molti aspetti le directory sono come i file, e si possono ridenominare nello stesso modo dei file.
7. **LINK.** È una tecnica che permette a un file di essere presente in più di una directory. Questa chiamata di sistema specifica un file esistente e un path name, e crea un link (collegamento) dal file esistente al nome specificato dal path: in questo modo, lo stesso file può essere presente in più directory.
8. **UNLINK.** Un elemento di directory è rimosso. Se il file che è stato sganciato è presente solo in una directory (il caso normale), esso è rimosso dal file system.

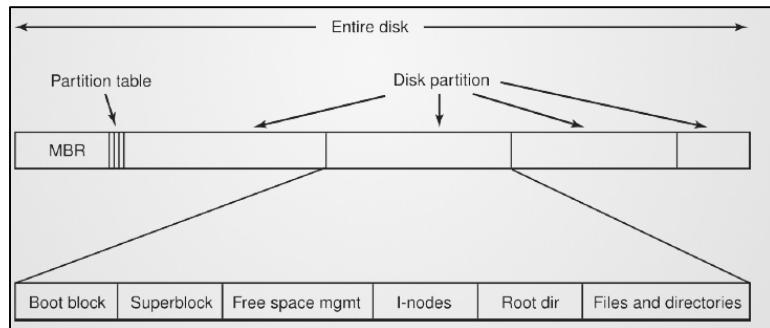
Struttura dei file system

I file system sono memorizzati in dischi; la maggior parte dei dischi possono essere divisi in una o più partizioni, con file system indipendenti per ogni partizione. Il settore del disco è chiamato MBR (record di avvio, Master Boot Record) e viene usato per avviare il computer. La fine del MBR contiene la **tavella delle partizioni**, che contiene il punto di inizio e di fine di ogni partizione; una delle partizioni della tabella è marcata come attiva.

Quando il computer è avviato, il **BIOS** legge ed esegue il programma MBR: la prima cosa che questo programma fa, è localizzare la partizione attiva, leggere il suo primo blocco, chiamato **blocco di avvio (boot block)** ed eseguirlo. Il programma nel blocco di avvio carica il sistema operativo contenuto nella partizione attiva; per uniformità, ogni partizione inizia con un blocco di avvio, anche se non contiene un sistema operativo avviabile, ed inoltre, potrebbe contenerne uno in futuro, quindi tenere pronto un blocco di avvio è comunque una buona idea. Spesso un file system conterrà il **superblocco**, che contiene tutti i parametri chiave relativi al file system. In seguito può esserci qualche informazione circa i blocchi liberi. Dopo di chè può esservi la directory radice, che contiene la cima dell'albero del file system. Infine, il resto del disco contiene tutte le altre directory ed i file.

MBR

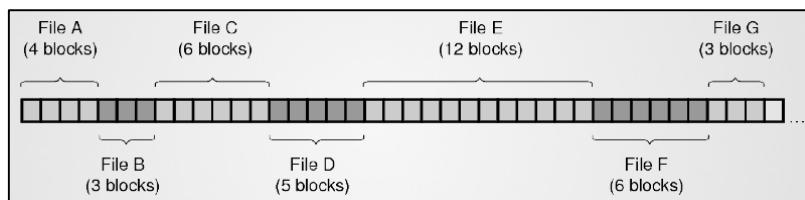
Il **MBR** è un settore dell'hard disk di un computer, noto anche come settore di avvio principale, composto dai primi 512 byte (mezzo kilobyte) del disco, che contiene la sequenza di comandi/istruzioni necessarie all'avvio (boot) del sistema operativo, (tipicamente è il boot manager/boot loader del sistema).



Implementazione dei file

Un disco può contenere molti file system, ognuno nella sua partizione del disco; l'allocazione dello spazio sul disco per ogni file è eseguita dal file system. Ne abbiamo di diversi tipi:

- **Allocazione contigua** (situato nelle immediate vicinanze, prossimo): Lo schema di allocazione più semplice è quello di memorizzare ogni file come un blocco contiguo di dati sul disco; su un disco con blocchi da 1 KB, un file di 50KB sarà allocato in 50 blocchi consecutivi, con blocchi da 2KB, sarà allocato in 25 blocchi. È quindi molto semplice da implementare in quanto basta conoscere l'indirizzo del blocco iniziale e la lunghezza per accedere ai blocchi del file.



L'allocazione contigua dello spazio su disco ha due vantaggi significativi.

- Primo, è semplice da implementare, perché per tener traccia di dove sono i blocchi di un file basta memorizzare due numeri, cioè l'indirizzo su disco del primo blocco e il numero di blocchi del file; dato il numero del primo blocco, il numero di ogni altro blocco può essere calcolato con una semplice addizione.
- Secondo, è molto efficiente in lettura, poiché l'intero file si può leggere dal disco con una singola operazione; è necessaria solo un'operazione di posizionamento (sul primo blocco) e, dopo di ciò, non saranno necessari altri posizionamenti e ritardi di rotazione, quindi si può accedere ai dati alla massima velocità di accesso fornita dal disco.

Quindi, l'allocazione continua è semplice da implementare e dà alte prestazioni. Sfortunatamente l'allocazione contigua ha anche un inconveniente ugualmente significativo: col passare del tempo il disco diventa frammentato, ed è necessario conoscere a priori la dimensione massima del file in fase di creazione.

La traduzione da indirizzo logico a fisico (indicando con B la dimensione del blocco e LA l'indirizzo logico) avviene eseguendo la divisione intera di LA per B: il quoziente va sommato all'indirizzo del blocco iniziale per determinare l'indirizzo del blocco da accedere, mentre il resto indicherà l'offset all'interno di quest'ultimo.

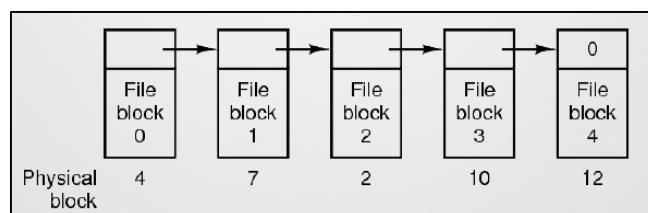
L'allocazione contigua soffre di frammentazione esterna in quanto, dovendo i file occupare una sequenza di blocchi contigui, in seguito a creazioni e cancellazioni di file si può avere una situazione in cui lo spazio libero totale su disco è sufficiente a soddisfare una nuova richiesta, ma non è contiguo, ovvero, i buchi di blocchi liberi contigui sono troppo piccoli per la richiesta in oggetto.

Il file-system ISO-9660 impiegato sui CD-ROM fa uso di una allocazione contigua.

- **Allocazione non contigua:** I moderni file system adottano l'allocazione non contigua della memoria per l'allocazione dello spazio sul disco. In questo approccio, una parte di spazio sul disco è allocata su richiesta, ovvero, quando viene creato un file oppure quando la sua dimensione aumenta a seguito di un'operazione di aggiornamento. Il file system deve risolvere tre problemi per implementare questo approccio:
- Gestione dello spazio libero sul disco: deve tenere traccia dello spazio libero sul disco e allocarlo quando un file richiede un nuovo blocco del disco
 - Evitare movimenti eccessivi della testina del disco: garantire che un file non sia "sparagliato" in diverse parti del disco, poiché ciò causerebbe un movimento eccessivo delle testine del disco durante l'elaborazione del file
 - Accesso ai dati del file: mantenere le informazioni sui file per trovare i blocchi del disco che lo contengono

L'allocazione non contigua può essere concatenata (chiamata anche con liste collegate) o indicizzata.

- **Allocazione a lista concatenata** (o liste collegate): Il secondo metodo per la memorizzazione di file è quello di mantenere ognuno come una lista concatenata di blocchi. In questo modo si possono usare tutti i blocchi del disco, cosa che non era possibile nell'allocazione contigua. D'altra parte l'accesso ai file risulta particolarmente lento.



L'allocazione concatenata risolve il problema della frammentazione e quello della dichiarazione delle dimensioni del file, entrambi presenti nell'allocazione contigua della memoria. Tuttavia, in mancanza di una FAT (tabella di memoria), l'assegnazione concatenata non è in grado di sostenere un efficiente accesso diretto, poiché i puntatori ai blocchi sono sparsi, con i blocchi stessi, per tutto il disco e si devono recuperare in ordine.

In questa allocazione, ogni file è rappresentato da una lista concatenata di blocchi del disco, che possono essere sparpagliati ovunque sul disco.

Ogni blocco del disco ha due campi al suo interno:

- Dati, che contiene, appunto, i dati
- Metadati, che è un campo di tipo link e punta al prossimo blocco

Il campo *Info_di_posizione* dell'elemento della directory punta al primo blocco sul disco del file. Agli altri blocchi si accede seguendo i puntatori dei vari blocchi. L'ultimo blocco del disco contiene un'informazione null nel campo metadati.

Lo spazio libero sul disco è rappresentato da una free list in cui ogni blocco libero contiene un puntatore al successivo. Quando viene richiesto un blocco, viene estratto un blocco dalla free list per poi essere aggiunto alla lista dei blocchi del file. Per cancellare un file, la lista di blocchi del file viene semplicemente aggiunta alla free list.

Ad esempio, nella figura:

- Il file alpha è costituito da due blocchi, il numero 3 e il numero 2;
- Il file beta è costituito da tre blocchi, il numero 4, seguito dal 5, seguito dal 7
- La free list è di tre blocchi, l'1, il 6 e l'8.

VANTAGGI: Il vantaggio principale di questa allocazione è che è sufficiente memorizzare in ogni elemento della directory solamente l'indirizzo su disco del primo blocco, mentre la parte rimanente si può trovare a partire da esso. Questo porta anche ad una lettura sequenziale semplice da effettuare.

SVANTAGGI: Tuttavia ci sono anche degli svantaggi. L'accesso diretto è estremamente lento, perché per arrivare al blocco n occorre necessariamente aver letto gli n-1 blocchi che lo precedono. Un ulteriore

svantaggio riguarda lo spazio richiesto per i puntatori. La soluzione più comune a questo problema consiste nel riunire un certo numero di blocchi continui in gruppi (cluster) e nell'assegnare i gruppi di blocchi anziché i singoli blocchi.

Un altro problema riguarda l'affidabilità. Se si danneggiasse il campo metadati di un blocco, cioè se si danneggiasse il puntatore, i dati successivi al blocco danneggiato potrebbero essere persi. Ci sono delle soluzioni a questo problema, come l'uso di liste doppiamente concatenate, che però sono onerose da implementare.

Facendo uso di una lista concatenata, che ha blocchi da 4kb e un numero di blocco di 32bit, quanti blocchi occorrono per memorizzare un file da 40kb?

Un blocco della lista è di 4KB, ovvero $4 \cdot 2^{10}$ byte.

Ogni blocco della lista concatenata ha un solo puntatore che specifica il numero del blocco successivo.

Questo numero è di 32bit, ovvero di 4byte.

Dunque ogni blocco della lista ha uno spazio per i dati di:

$$(4 \cdot 2^{10} - 4) \text{ byte} = (4 \cdot 1024 - 4) \text{ byte} = (4096 - 4) \text{ byte} = 4092 \text{ byte.}$$

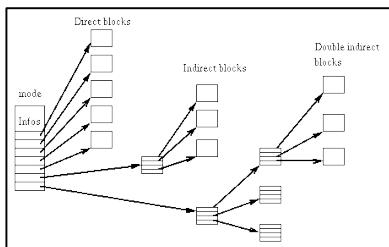
Il file è di 40KB, quindi $(40 \cdot 2^{10}) \text{ byte} = 40960 \text{ byte.}$

La dimensione del file diviso lo spazio disponibile per ogni blocco è: $40960 / 4092 = 10,01.$

Sono serviti poco più di 10 blocchi, cioè 11 blocchi.

- **Allocazione indicizzata (i-node):** Il quarto e ultimo metodo per tener traccia dei blocchi appartenenti a un file consiste nell'associare a ogni file una piccola struttura dati detta i-node (index node, nodo indice), che elenca gli attributi e gli indirizzi del disco dei blocchi del file. A partire dall'i-node è possibile trovare tutti i blocchi del file; il grande vantaggio di questo schema rispetto all'usare una tabella che risiede in memoria è che l'i-node ha bisogno di stare in memoria solo quando il file ad esso corrispondente viene aperto. Se ogni i-node occupa n byte, e se possono essere aperti un massimo di k file alla volta, la memoria totale occupata dall'array che tiene gli i-node dei file aperti è grande soltanto kn , e solo questo spazio dovrà essere riservato.

Un problema con gli i-node è che se ognuno di essi ha spazio per un numero fissato di indirizzi di disco, cosa succede se un file cresce al di là di questo limite? Una soluzione consiste nel riservare l'ultimo indirizzo, non per un blocco dati, ma per l'indirizzo di un blocco contenente altri indirizzi di disco.



L'allocazione indicizzata risolve il problema dell'accesso diretto, presente nell'allocazione concatenata, raggruppando tutti i puntatori in una sola locazione: il blocco indice. Nell'allocazione indicizzata si mantengono tutti i puntatori ai blocchi di un file in una tabella indice chiamata **file map table (FMT)**. In questa tabella sono riportati gli indirizzi dei blocchi del disco allocati a un file.

Ogni file ha il proprio blocco indice (o tabella indice): nella sua forma più semplice, un FMT è un array di indirizzi di blocchi del disco. Ogni blocco ha un solo campo, il campo dati. L'i-esimo elemento del blocco indice punta all'i-esimo blocco del file.

Il campo *Info_di_posizione* dell'elemento della directory relativo a un file contiene l'indirizzo della FMT, cioè punta alla FMT. Una volta creato il file, tutti i puntatori del blocco indice sono impostati a null. Quando le dimensioni del file crescono, viene localizzato un blocco libero e l'indirizzo di questo blocco viene aggiunto alla FMT del file.

VANTAGGI: Questa allocazione permette di accedere direttamente a un blocco del file direttamente dalla FMT senza avere frammentazione esterna. Inoltre l'affidabilità è migliore in quanto il danneggiamento di un elemento della FMT non compromette l'intero file, ma solo una parte di esso.

SVANTAGGI: Il problema principale consiste nella dimensione del blocco indice, cioè della FMT. Se essa è troppo piccola non può contenere un numero di puntatori sufficiente per un file di grandi dimensioni, quindi è necessario disporre di un meccanismo per gestire questa situazione.

Possibili soluzioni:

- **Indice a più livelli:** Una prima soluzione consiste nell'utilizzo di un blocco indice a più livelli.

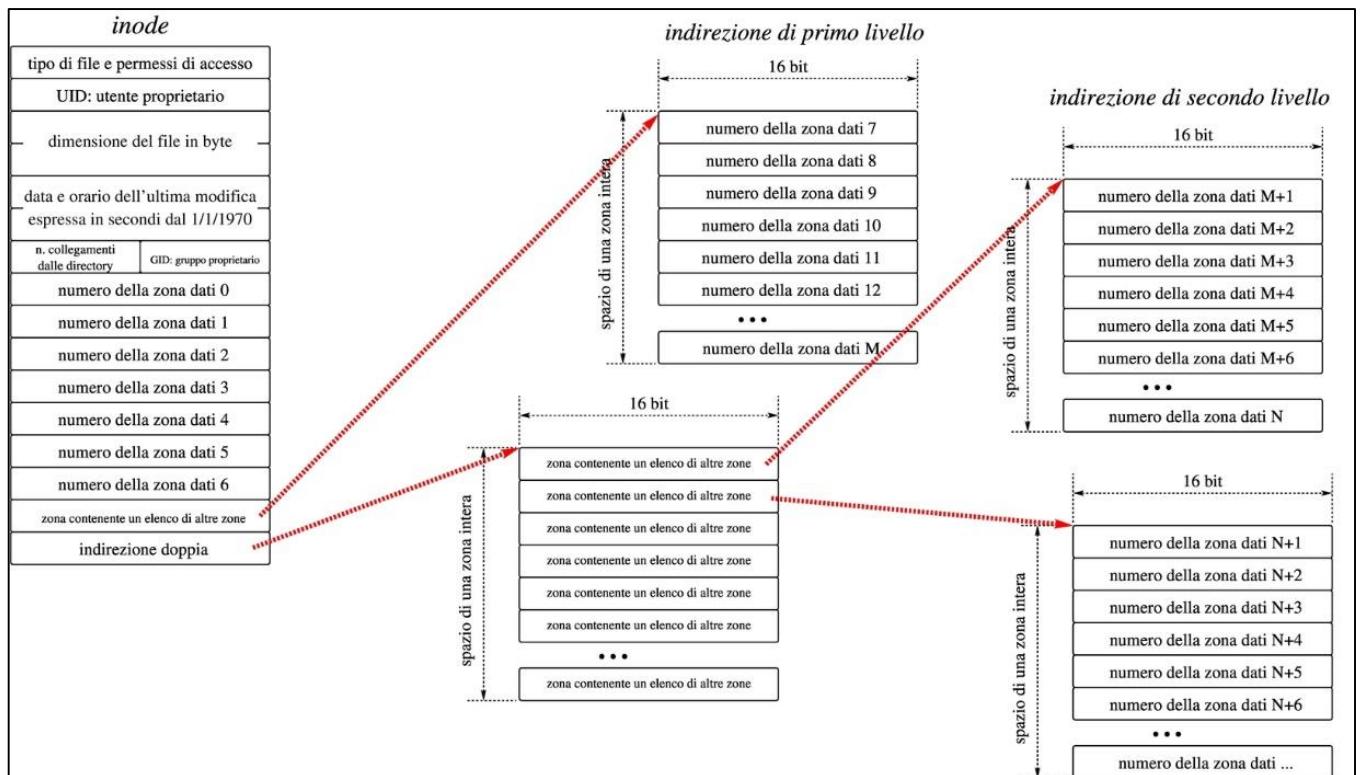
In questa organizzazione, ogni elemento della FMT contiene l'indirizzo di un blocco indice. Un blocco indice non contiene dati; contiene elementi che contengono gli indirizzi dei blocchi dati. Per accedere ai blocchi dati, prima accediamo a un elemento della FMT e otteniamo l'indirizzo di un blocco indice.

Successivamente, accediamo a un elemento del blocco indice per ottenere l'indirizzo di un blocco dati.

- **Organizzazione ibrida:** Alcuni file system implementano un'organizzazione ibrida della FMT che include alcune delle caratteristiche dell'allocazione classica e dell'allocazione indicizzata multilivello.

I primi elementi nella FMT, ad esempio n elementi, puntano a blocchi dati come nell'allocazione indicizzata. Gli altri elementi puntano a blocchi indice.

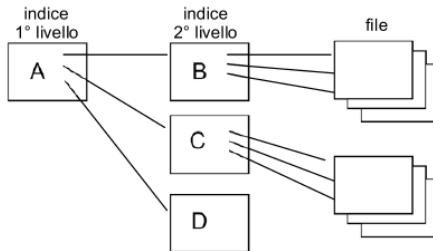
Il vantaggio di questa organizzazione è che piccoli file continuano a essere accessibili in maniera efficiente, poiché la FMT non utilizza i blocchi indice. I file di medie o grandi dimensioni soffrono di un parziale degrado delle prestazioni di accesso a causa dei livelli di indirizzamento.



Supponiamo di utilizzare un file system UNIX basato su i-node particolari che contengono i seguenti campi: 12 indirizzi diretti a blocchi di dati, 1 indirizzo ad un blocco indiretto singolo e 1 indirizzo ad un blocco indiretto doppio. Supponendo di avere numeri di blocchi a 32 bit e blocchi su disco da 1 kb, indicare esattamente la dimensione massima (in kb) supportata da un simile i-node. Esplicitare il calcolo utilizzato.

L'indirizzamento indiretto ha blocchi da 1kb pieno di indirizzi a 32 bit, quindi sono 12 blocchi di indirizzo diretto + 256 di indiretto + 256^2 di indiretto doppio = 65804 blocchi.

Illustrare il meccanismo di allocazione indicizzata dei file. Si faccia riferimento ad un file system con indici a due livelli, dimensione del blocco logico pari a 512 byte e indirizzi di 4 byte. Come si alloca un file di 1 Mb? Quanto blocchi servono? Come si accede al suo 400° blocco? Come si accede al byte 236.448? Qual è la dimensione massima di un file? Quanti blocchi occupa complessivamente?



L'allocazione indicizzata a due livelli segue questo schema:

Se il blocco logico è di 512 byte e gli indirizzi di 4 byte si hanno $512 / 4 = 128$ indirizzi per blocco.

Un file di 1 Mbyte occupa $1M / 512 = 2K = 2048$ blocchi ($1M = 1024 * 1024$)

2048 blocchi richiedono 2048 indici = $2048 / 128 = 16$ blocchi di 2° livello.

Complessivamente il file occupa $2048 + 16 + 1 = 2065$ blocchi.

Il blocco n. 400 è indicizzato dal blocco indice di 2° livello n. 400 / 128 = 3, l'indirizzo è il 400 mod 128 = 16-esimo del blocco.

In generale, l'n-esimo indirizzo occupa i byte da $(n - 1) \times 4$ a $(n - 1) \times 4 + 3$.

Il blocco indice di secondo livello a sua volta è indicizzato dal 4° indirizzo (indirizzo n. 3) dell'indice di 1° livello, che si trova nei byte 12-15 del blocco.

L'accesso al byte 236.448 si risolve così: il byte occupa il blocco $236.448 / 512 = 461$ (contando da 0, quindi è il 462-esimo), e si trova all'offset $236.448 \bmod 512 = 416$. Trovato il blocco si procede come sopra.

La dimensione massima di un file è data dal numero massimo di indirizzi utilizzabili per indicizzarlo.

Nell'indice di 1° livello ci sono al massimo 128 puntatori, quindi ci sono al massimo 128 blocchi indice di 2° livello, ciascuno dei quali contiene 128 puntatori ai blocchi del file, per un totale di $128 \times 128 = 16384$ puntatori, e quindi blocchi del file. Il file può avere una dimensione massima di $16384 \times 512 = 8$ Mbyte (8.388.608 byte), e occupa complessivamente $16384 + 128 + 1 = 16513$ blocchi

Struttura di un i-node

Ogni i-node in un sistema UNIX contiene le seguenti informazioni:

- modo: bit di accesso, di tipo e speciali del file
- UID e GID del possessore,
- dimensione del file in byte,
- timestamp di ultimo accesso (atime), di ultima modifica (mtime), di ultimo cambiamento dell'i-node (ctime),
- numero di link hard che puntano all'inode,
- blocchi diretti: puntatori ai primi 12 blocchi del file,
- primo indiretto: indirizzo del blocco indice dei primi indiretti,
- secondo indiretto: indirizzo del blocco indice dei secondi indiretti,

Implementazione delle directory

Una directory dovrebbe essere costituita da una lista lineare in modo da poter effettuare ricerche lineari per trovare il file richiesto. Tuttavia, questa organizzazione è inefficiente se la directory contiene un elevato numero di elementi.

Per avere una maggiore efficienza, cioè per diminuire notevolmente il tempo di ricerca nelle directory, sono usate organizzazioni che fanno uso di una tabella hash o di un albero B+.

Prima che si possa leggerlo, bisogna aprire un file. Quando si apre un file, il sistema operativo usa il path name definito dall'utente per localizzare l'elemento della directory, e questo fornisce le informazioni necessarie per trovare i blocchi del disco. Un problema strettamente collegato è il posto in cui si dovranno memorizzare gli attributi; ogni file system mantiene gli attributi dei file, ad esempio il proprietario del file e il tempo di creazione, e questi devono essere conservati da qualche parte. Una possibilità ovvia è quella di memorizzarli direttamente nell'elemento di directory e ciò è realizzato in molti sistemi. Per i sistemi che usano i-node, un'altra possibilità è memorizzare gli attributi nell'i-node piuttosto che nell'elemento di directory. In questo caso, l'elemento della directory può essere più piccolo: solo un nome di file e il numero dell'i-node. L'approccio più semplice consiste nel fissare un limite alla lunghezza dei nomi dei file, tipicamente 255 caratteri, se viene rimosso un file, si introduce uno

spazio di lunghezza variabile dentro la directory, e questo spazio può non ~sser~ adatto al prossimo file che verrà aggiunto. Questo problema è lo stesso che abbiamo incontrato nel caso di file contigui. Un altro problema, invece, consiste nel fatto che un singolo elemento relativo ad una directory può allargarsi su più di una pagina, quindi può verificarsi un page fault durante la lettura di un nome di file. Un altro modo per gestire nomi di file di lunghezza variabile è quello di avere gli elementi delle directory di dimensione fissa, e lasciare i nomi dei file tutti insieme in un mucchio (heap) alla fine della directory. In tutte le configurazioni descritte fino a questo momento, le ricerche di un nome di file avvengono in modo lineare dall'inizio alla fine della directory; per directory estremamente lunghe, la ricerca lineare può essere molto lenta. Un modo per velocizzarla è quello di usare una tabella hash all'interno di ogni directory. L'uso delle tabelle di hash ha il vantaggio di consentire ricerche più veloci, ma lo svantaggio di richiedere una gestione molto più complessa; questo metodo può essere preso in considerazione in sistemi dove le directory normalmente contengono centinaia o migliaia di file.

Un modo completamente diverso per velocizzare la ricerca all'interno di directory grandi è quello di mettere in una cache i risultati delle ricerche effettuate; prima di iniziare una ricerca si effettua un controllo per vedere se il nome del file è nella cache, e se c'è, può essere localizzato velocemente, evitando ricerche lunghe. Naturalmente, questo approccio funziona solo se la maggior parte delle ricerche effettuate riguarda un numero relativamente piccolo di file.

Prestazioni del file system

Caching

La tecnica usata più comunemente per ridurre gli accessi al disco è il block cache o il buffer cache. In questo contesto cache è un insieme di blocchi che logicamente appartengono al disco, ma sono mantenuti in memoria per ragioni legate al miglioramento delle prestazioni. Si possono usare vari algoritmi per gestire la cache, ma comunemente si controllano tutte le richieste di lettura per vedere se il blocco necessario è nella cache; in questo caso la richiesta di lettura può essere soddisfatta senza accedere al disco. Se il blocco non è in cache, esso è dapprima letto nella cache, e poi copiato ovunque esso sia necessario; richieste successive per lo stesso blocco potranno essere soddisfatte dalla cache. Quando si deve caricare un blocco in una cache piena, si rimuove qualche blocco, riscrivendolo sul disco se è stato modificato da quando vi era stato messo. Questa situazione è molto simile alla paginazione, e si possono applicare tutti i tipici algoritmi di paginazione descritti nel Capitolo 3, ad esempio quello FIFO (First In First Out), quello detto "seconda possibilità", o quello LRU (Least Recently Used).

Sfortunatamente vi è un inconveniente: se un blocco critico, come un blocco i-node, è letto nella cache e modificato ma non riscritto sul disco, un crash lascerà il file system in uno stato incoerente; se tale blocco i-node è messo alla fine della catena LRU può passare molto tempo prima che raggiunga l'inizio e sia riscritto sul disco. Per mantenere l'integrità del file system, non è desiderabile tenere i blocchi dati in cache troppo a lungo prima della loro scrittura. Se il sistema cade, la struttura del file system non sarà alterata, ma il lavoro di una giorno sarà perduto.

Lettura anticipata dei blocchi

La cache e la lettura anticipata non sono i soli modi per aumentare le prestazioni di un file system; un'altra tecnica importante è ridurre il movimento delle testine del disco.

Ridurre il movimento del braccio del disco

Una seconda tecnica per migliorare le prestazioni del file system è quella di cercare di portare i blocchi nella cache prima che siano effettivamente necessari, per cercare di migliorare la percentuale di hit (di blocchi trovati nella cache). In particolare, molti file vengono letti in modo sequenziale; quando si chiede ad un file system di produrre il blocco k di un file, esso esegue la richiesta, e quando ha finito va a dare un'occhiata nella cache per vedere se il blocco k + 1 è già lì. Se non c'è, fa una richiesta di lettura del blocco k + 1, sperando che quando sarà necessario, sarà già arrivato nella cache, o quanto meno sarà già sulla strada. Naturalmente, questa strategia di lettura anticipata funziona solo per i file che vengono letti in modo sequenziale.

Condivisione di file su un file system

Due o più utenti che vogliono condividere un file, possono farlo usando una FAT, quindi duplicando la lista con i riferimenti ai blocchi, anche se con questo approccio avremo problemi in caso di append di un file, ovvero un file aperto da un utente che sta eseguendo un'operazione (file non disponibile a nessuno se non solo all'utente che ce l'ha aperto); oppure usando gli hard/soft link.

Tipi di link

L'utilizzo dei link è molto diffuso, in quanto rendono più flessibile la struttura del file system, evitando ad esempio duplicazioni di file, ma esistono profonde differenze tra i symbolic link (soft) e gli hard link, vediamole adesso in dettaglio.

Soft link

Il link simbolico (soft link o symlink) punta ad un file o ad una directory esistente, e l'utente potrà quindi fare riferimento a tale file anche mediante il link. Il symlink dipende in tutto e per tutto dal file originale: nel momento in cui viene cancellato il file sorgente, il collegamento non punterà più a nulla. Esso è universale e permette di fare riferimenti al di fuori del file system, anche se comporta un appesantimento nella gestione.

Hard link

Il link vero e proprio o hardlink, invece, ha un comportamento diverso, creando di fatto una nuova entry per lo stesso inode, diventando quindi a tutti gli effetti una nuova porta di accesso al medesimo file, con alcuni vantaggi, tra cui quello di non occupare in alcun modo spazio su disco, ma anche alcuni svantaggi, come quello di dover risiedere sul medesimo file system del file a cui fa riferimento, o per essere precisi all'inode a cui è associato, inoltre non è possibile effettuare hardlink a directory. Esso ha inoltre un contatore di link.

È da notare che anche le caratteristiche del file non vengono in alcun modo alterate dalla creazione di un hardlink, anzi possiamo creare l'hardlink, cancellare il file originale, ricreare il file originale come hardlink e cancellare l'hardlink iniziale, ma l'ora di creazione mostrata da ls rimarrà sempre quella del file originale.

Tutti gli hardlink sono, per definizione, riferimenti allo **stesso** file. Questo vale in qualsiasi file system che li supporta (ext2-3-4, NTFS, btrfs, etc). Quindi modificando un qualsiasi hardlink del file, si modificano tutti gli altri, in questo senso anche il file "originale" è un hardlink, non diverso dagli altri che creerai. I file system FAT e ReFS non supportano gli hard link.

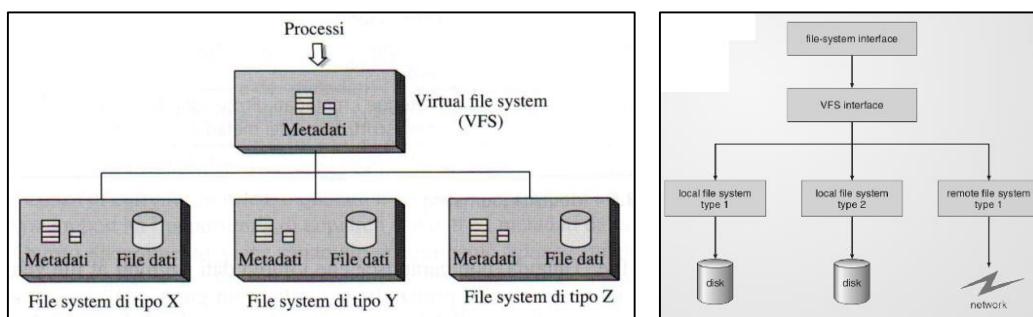
Non è possibile creare un collegamento fisico ad un oggetto residente su un file-system diverso. È sempre possibile creare un collegamento simbolico ad un file di dispositivo speciale. Non crea alcun problema creare un collegamento fisico ad un collegamento simbolico, purché quest'ultimo sia consistente.

File system virtuale (FSV)

Gli utenti richiedono requisiti differenti a un file system, come convenienza, alta affidabilità, risposte veloci e accesso ai file su altri sistemi. Un singolo file system non può fornire tutte queste caratteristiche, per cui un sistema operativo fornisce un file system virtuale (VFS) che facilita l'esecuzione simultanea di diversi file system. In questo modo ogni utente può usare il file system che preferisce.

In pratica un file system virtuale è un livello software che consente a diversi file system di essere in funzione su un computer simultaneamente, in modo che un utente possa scegliere il file system che è più adatto per le sue applicazioni.

Un processo invoca il livello VFS utilizzando comandi generali per l'accesso ai file e il livello VFS redireziona il comando al file system appropriato.



Ciò è implementato da un layer VFS situato tra un processo e un file system. Il layer VFS ha due **interfacce**: un'interfaccia col file system (interfaccia VFS) e un'interfaccia con i processi (interfaccia POSIX). Ogni file system

conforme alle specifiche dell'interfaccia del file system VFS può essere installato per funzionare con il VFS. Questa caratteristica rende facile aggiungere un nuovo file system.

L'interfaccia del VFS con il processo fornisce le funzionalità per eseguire le generiche operazioni sui file (come open, close, read, write) e le operazioni mount e umount sul file system. L'interfaccia con il file system serve per determinare a quale file system appartiene il processo, invocando le operazioni open, close, read e write dello specifico file system.

Il FSV risulta molto utile con i dispositivi rimovibili come le penne USB o i CD/DVD in quanto consente all'utente di montare il file system presente in questi dispositivi nella sua directory corrente e accedere ai file senza preoccuparsi del fatto che i dati sono memorizzati in un formato differente.

Gestione blocchi liberi

1001101101101100
0110110111101011
1010110110110110
0110110110110111
1110110110110111
1101101010001111
0000111011010111
1011101101101111
1100100011101111
~
0111011101110111
1101111011101111

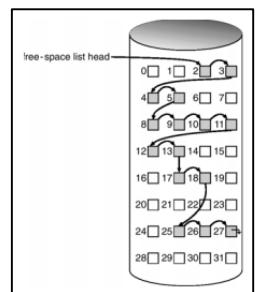
Per memorizzare i blocchi liberi si usa una bitmap (un vettore o mappa di bit) che memorizza n bit per n blocchi. Ogni singolo bit rappresenta un blocco del disco: se il blocco è allocato ad un file il bit è 0, se il blocco è libero il bit è 1. Il vantaggio di questa codifica è essenzialmente l'efficienza nel trovare il primo blocco libero dato che la maggior parte delle CPU più diffuse mettono a disposizione delle istruzioni macchina che forniscono l'offset del primo bit a 1 in una parola. In questo modo il numero del primo blocco libero può essere calcolato come segue:

$$(numero\ di\ bit\ un\ una\ parola) \cdot (numero\ delle\ parole\ con\ tutti\ i\ bit\ 0) + offset\ del\ primo\ bit\ 1$$

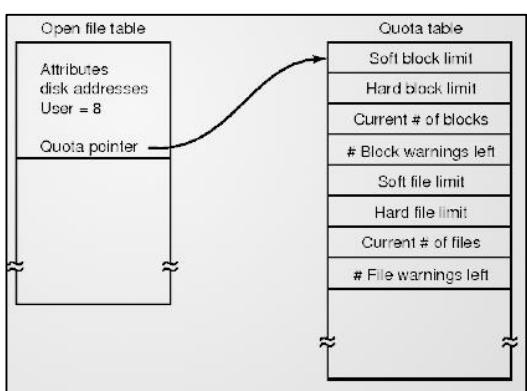
Tale bitmap è comunque relativamente piccola, e come strategie di allocazione in memoria sceglie tra memorizzare tutto in una volta in memoria o un blocco alla volta.

Lo svantaggio è quindi che la bitmap deve essere tenuta in memoria per un utilizzo veloce e quindi può portare ad uno spreco di quest'ultima se il disco è di dimensioni ragguardevoli.

Dato che lo spazio richiesto dalla bitmap potrebbe non essere in memoria. Una soluzione alternativa sarebbe l'uso di liste concatenate (**lista di blocchi liberi**). Tale implementazione richiede più spazio e c'è una bassa efficienza nella ricerca, ma si sfruttano i blocchi liberi e non c'è spreco di spazio.



Quote su disco



Le quote disco controllano la quantità di spazio su disco assegnata agli utenti che condividono le risorse di un solo elaboratore. Per una corretta gestione delle quote, l'amministratore dell'elaboratore deve essere in grado di impostare e rimuovere i limiti sullo spazio su disco e avvertire gli utenti che stanno per raggiungere tali limiti. In alcuni casi l'amministratore potrebbe inoltre decidere di negare ulteriore spazio su disco agli utenti che superano i limiti impostati.

L'assegnazione delle quote di utilizzo, per la stessa unità disco, è usata se l'HD è di grosse dimensioni e se è usato da più persone contemporaneamente. Per configurarlo è necessario disporre di un kernel che lo supporti, come è nel caso del kernel di Debian.

Il sistema delle quote consente di impostare quattro limiti:

- Due limiti (chiamati «soft» e «hard») si riferiscono al numero di blocchi consumati. Se il file system è stato creato con una dimensione dei blocchi di 1 kibibyte, un blocco contiene 1024 byte di uno stesso file. Blocchi non saturi quindi portano a perdite di spazio su disco. Una quota di 100 blocchi, che permette teoricamente la memorizzazione di 102.400 byte, sarà comunque satura con soli 100 file di 500 byte ciascuno, che rappresentano solo 50.000 byte in totale.
- Due limiti (soft e hard) si riferiscono al numero di inode utilizzati. Ogni file occupa almeno un inode per memorizzare le sue informazioni (permessi, proprietario, data e ora dell'ultimo accesso, ecc.). È quindi un limite al numero di file dell'utente.

Un limite «soft» può essere temporaneamente superato, l'utente sarà semplicemente avvertito che sta superando la quota. Un limite «hard» non può mai essere superato: il sistema rifiuterà qualsiasi operazione che causerebbe il superamento della quota.

Controlli di consistenza

Nella gestione dello spazio su disco il sistema operativo assegna i blocchi ai file e, quando questi sono cancellati, recupera lo spazio su disco riportando i blocchi non più in uso nell'elenco dei blocchi liberi. Può succedere che per un crash di sistema, o per un errore di un'applicazione, i blocchi che descrivono lo stato di allocazione dei file contengano informazioni non corrette lasciando il file system in uno stato inconsistente.

Aposite utility possono effettuare dei controlli di consistenza: vediamo una tecnica usata dai programmi di servizio del sistema operativo per controllare la consistenza del file system attraverso la corrispondenza tra i **blocchi** allocati ai diversi file e i blocchi liberi.

Consideriamo un file system con 20 blocchi, numerati da 0 a 19. Supponiamo che nel sistema ci siano 4 file: lo stato di allocazione dei blocchi ai file e l'elenco dei blocchi liberi sono descritti nella tabella a destra. Il programma di controllo costruisce due tabelle: una prima tabella che conta i blocchi occupati e una seconda che conta i blocchi liberi. Ogni tabella contiene una serie di contatori, uno per ogni blocco, inizializzati a 0. Il programma scandisce le mappe di allocazione dei file e aggiorna i contatori incrementandone il valore ogni volta che un blocco è attribuito a un file. La tabella con i contatori dei blocchi liberi è costruita allo stesso modo partendo dalla struttura dati che elenca i blocchi liberi.

Elenco Blocchi	
File 1	0, 1, 17
File 2	6, 9
File 3	10, 11, 18
File 4	12, 16, 2
Blocchi liberi	3, 4, 5, 7, 8, 13, 14, 15, 19

Applicando questo controllo ai dati in tabella si ottengono i due array seguenti

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
1	1	1	0	0	0	1	0	0	1	1	1	1	0	0	0	1	1	1	0
0	0	0	1	1	1	0	1	1	0	0	0	0	1	1	1	0	0	0	1

Blocchi occupati
Blocchi liberi

La consistenza del file system è verificata controllando che per ogni blocco ci sia il valore 1 in uno solo dei due array, perché un blocco deve essere o libero oppure assegnato a un file e, in questo caso, a un solo file.

Con questo tipo di controllo si possono incontrare diverse situazioni anomale. Consideriamo, per esempio, le seguenti tabelle:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
0	0	1	0	0	1	1	0	0	0	1	1	1	1	1	2	0	1	0	0
0	1	0	1	1	0	0	2	1	1	1	0	0	0	0	0	1	0	1	1

↑ ↑ ↑ ↑

Blocchi occupati
Blocchi liberi

- Il blocco 0 è non è allocato ad alcun file ma non compare tra i blocchi liberi. Si tratta di una situazione anomala che ha come unica conseguenza la riduzione dello spazio su disco e che è immediatamente recuperabile dal software di controllo inserendo il blocco 0 tra i blocchi liberi.
- Il blocco 7 compare due volte nell'elenco dei blocchi liberi. Si potrebbe arrivare a una anomalia di questo tipo a causa di una tabella dei blocchi liberi con i seguenti valori non corretti: 3, 14, 5, 7, 8, 7, 13, 15. La situazione è recuperabile rimuovendo una delle due occorrenze di 7 nella lista dei blocchi liberi.
- Anche il caso del blocco 10, dove un blocco è assegnato a un file, ma compare anche tra i blocchi liberi, è recuperabile rimuovendo il blocco dalla lista dei blocchi liberi. Si osservi che questa anomalia può essere la conseguenza dell'anomalia sopra descritta dove un blocco appariva due volte nell'elenco dei blocchi liberi. Questa anomalia, come la precedente, è potenzialmente molto pericolosa, perché può essere alla base alla situazione del blocco 15.
- Il caso del blocco 15 è il più grave perché lo stesso blocco è stato allocato a più di un file. Per correggere l'anomalia, il programma di controllo deve identificare i file ai quali è stato allocato lo stesso blocco, duplicare il blocco 15, copiandolo in un blocco libero, e allocare il nuovo blocco a uno dei due file, sostituendolo al blocco 15.

File system e log

La tendenza in questo momento è di creare dischi sempre più capienti, cache ampie e scritture piccole.

Allora si è avanzata la proposta di un file system alternativo chiamato **Log-structured File System (LFS)**. LFS presenta un nuovo design rispetto ai più tradizionali file system. La differenza più importante è che mentre i file system classici scrivono i file su disco cercando i blocchi tra quelli al momento disponibili, LFS scrive sempre negli stessi blocchi, tutti nello stesso posto (cioè tutti i blocchi da scrivere in un dato momento sono scritti in posizioni adiacenti, indipendentemente dal file di cui fanno parte), in modo che lo stesso blocco di un file, scritto in momenti diversi, esisterà sul disco in posizioni diverse.

Ciò permette la creazione sicura ed asincrona del file (il vecchio indice rimane tra i dati della directory che lo contiene anche in caso di crash), una più veloce scrittura su disco (tutti i blocchi sono scritti insieme, senza necessità di trovare un posto libero), e un recupero istantaneo in caso di arresto del sistema (il filesystem ricomincia dall'ultimo punto di controllo e prosegue, invece di dover essere controllato nella sua totalità per verificarne la consistenza).

I punti salienti di tale file system sono:

- log circolare basato su segmenti;
- ogni segmento può contenere blocchi di dati e i-node;
- si trasformano accessi I/O random in I/O sequenziali;
- necessita di una mappa degli i-node: (# i-node → posizione i-node);
- gestione pulitura vecchi segmenti;

Un utilizzo parziale di tale idea è stata sfruttata per ottenere robustezza tramite il file system con Journaling.

Journaling file system (JFS)

Un file system durante l'esecuzione mantiene in memoria una parte dei file dati e dei metadati, in particolare mantiene in memoria i **file control block**, le file **map table** e le **free list**.

Quando l'esecuzione di un file system viene terminata dall'utente del sistema, il file system copia tutti i dati e i metadati dalla memoria RAM sul disco, in modo che la copia sul disco sia completa e consistente.

Tuttavia, quando si verifica una mancanza di corrente elettrica o quando il sistema viene spento all'improvviso, il file system non ha l'opportunità di copiare i file dati e i metadati sul disco. Questo spegnimento è detto spegnimento sporco e causa una perdita dei dati dei file e dei metadati contenuti in memoria.

Tradizionalmente, i file system utilizzavano delle **tecniche di ripristino** per proteggersi contro la perdita di dati e metadati poiché erano molto semplici da implementare. Per questo motivo, si creavano backup periodici e i file erano ripristinati a partire dalle copie di backup nel momento in cui si verificavano dei malfunzionamenti. La creazione delle copie di backup comportava un piccolo overhead durante il normale funzionamento del sistema. Tuttavia, quando si verifica un malfunzionamento, l'overhead per la correzione delle inconsistenze era elevato ed, inoltre, il sistema non era disponibile durante il ripristino.

Un file system moderno utilizza tecniche di fault tolerance in modo da poter riprendere l'esecuzione velocemente dopo uno spegnimento improvviso. Un **journaling file system** implementa la fault tolerance mantenendo un journal, un diario giornaliero, dove vengono salvate le azioni che il file system si accinge ad eseguire prima di eseguirle effettivamente. Quando l'esecuzione del file system viene ripristinata dopo uno spegnimento improvviso, il file system consulta il journal per identificare le azioni non ancora eseguite e le esegue, garantendo in questo modo la correttezza dei dati dei file e dei metadati.

L'uso di tecniche di fault tolerance genera un elevato overhead. Per questo motivo il JFS offre diverse modalità journaling. Un amministratore di sistema può scegliere una modalità journaling da adattare al tipo di affidabilità necessaria nell'ambiente di elaborazione.

La strategia del JFS è quindi riassumibile in tali punti:

- Le operazioni sui meta-dati sono preliminarmente appuntate in un log che viene poi ripulito a posteriori (subito dopo);
- In caso di ripristino da crash si ripetono le operazioni in log.

I più diffusi file system dotati di journaling sono: NTFS, ext3, ext4, ReiserFS, XFS, Journaled File System (JFS), VxFS, HFS+

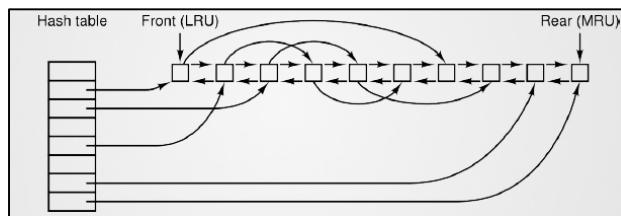
Cache del disco e dei file

Una tecnica generale per velocizzare l'accesso ai dati consiste nell'utilizzare una gerarchia di memoria composta da una parte della memoria e dai file memorizzati sul disco. Nel caso specifico di accesso ai file, per migliorare la prestazione dei dischi si fa spesso uso di una cache del disco o buffer cache.

Il *caching* è la tecnica di mantenere alcuni file in memoria, in modo che vi si possa accedere senza dover eseguire un'operazione di I/O. Il caching riduce il numero di operazioni di I/O necessarie per accedere ai dati memorizzati in un file, migliorando le prestazioni delle attività di elaborazione dei file nei processi e migliorando inoltre le prestazioni del sistema.

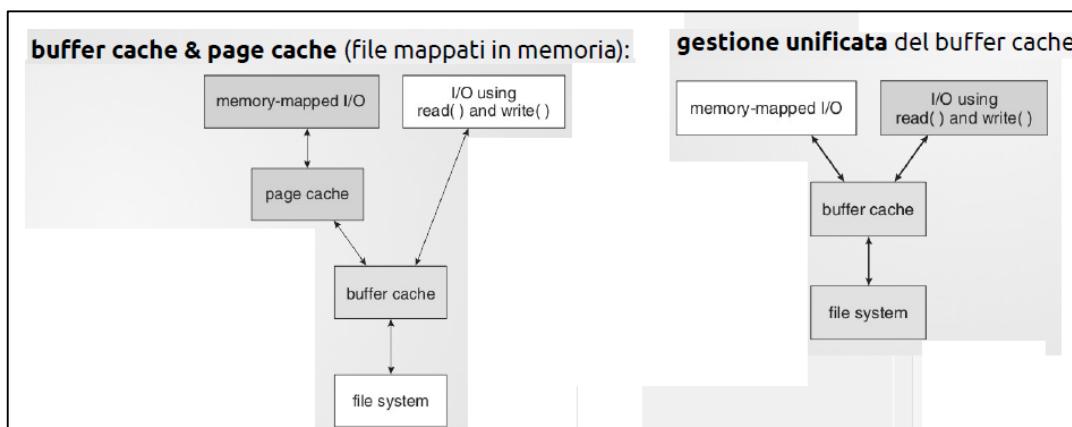
Il file system implementa una cache del disco per ridurre il numero di operazioni di I/O per accedere ai file memorizzati sul disco. Un metodo di accesso implementa una cache di file per ridurre il numero di operazioni di I/O eseguite durante l'elaborazione di un file.

La struttura di tale cache è basata sulle tabelle hash.



Page cache e Buffer cache

Molti sistemi come Linux, Solaris e Windows NT usano una **page cache** che contiene pagine piuttosto che blocchi fisici i disco usando tecniche di memoria virtuale. Alcune versioni di Unix implementano una **buffer cache unificata** che velocizza e ottimizza la lettura di pagine dal disco.



Deframmentazione

La deframmentazione è un'operazione di ottimizzazione dell'archiviazione dei dati nella memoria di massa di un computer. Consiste nel ridurre la frammentazione esterna dei file presenti sulla memoria ristrutturandone l'allocazione e facendo in modo che ciascun file risulti memorizzato in una zona contigua dal punto di vista fisico, permettendo così di ridurre drasticamente i tempi di accesso e lettura dei file.

La deframmentazione ripristina la contiguità dei file rendendo più veloce la loro lettura in quanto il sistema operativo non deve cercare le parti del file sparse per l'hard disk.

La deframmentazione è un'operazione che può essere svolta in modo automatico dal file system durante il suo regolare funzionamento oppure eseguita tramite esplicita richiesta dell'utente tramite l'esecuzione di un programma.

Campo di applicazione

La deframmentazione può essere utilizzata solo per ridurre la frammentazione esterna (frammentazione causata da piccole aree di memoria inutilizzate e non allocate). La frammentazione interna (allocazione eccessiva di memoria per un processo che viene così inutilizzata) non può invece attualmente essere ridotta, a causa del metodo di utilizzo dei file system odierni. L'utilizzo della deframmentazione è essenziale per i file system che utilizzano una tecnica di allocazione file contigua, altrimenti a lungo andare non sarebbe possibile allocare nuovi file anche se ci fosse spazio sufficiente (ma frammentato) sul disco. Nell'allocazione a lista concatenata usata nei moderni file system la deframmentazione serve invece soltanto a velocizzare le operazioni di lettura\scrittura su disco.

Quando un file viene scritto su un hard disk, questo occupa il primo spazio libero (settore) disponibile, ed occupa tanti settori quanti ne servono per contenere l'intero file. Se però lo spazio contiguo è troppo piccolo rispetto alle dimensioni del file, il file viene suddiviso in uno o più pezzi non contigui. Alla fine di ogni catena di settori contigui, viene inserito un riferimento che rimanda al prossimo settore che contiene la restante porzione del file, fino al raggiungimento della fine del file.

Un file, modificato in più occasioni, può aumentare di dimensione e quindi verrà sempre più suddiviso in "frammenti" per poter essere memorizzato nelle parti libere dell'hard disk. Reiterate azioni di questo tipo, anche se del tutto normali, portano ad una progressiva frammentazione dei file, con il risultato di rendere più lento il loro reperimento.

La gestione del file da parte del sistema operativo è trasparente per l'utente, ma l'hard disk è costretto a leggere in punti differenti della faccia del disco, saltando da un settore all'altro, per fornire il file completo, rallentando così le operazioni di input/output.

Funzionamento

Il programma di deframmentazione sposta temporaneamente i cluster (dati sparsi) di uno stesso file in zone libere del disco; successivamente cerca (o libera spostando altri cluster) uno spazio contiguo che possa contenere completamente questo file. Questa operazione ha tempistiche lunghe, in quanto il sistema operativo oltre a dover copiare i file in un'altra posizione del disco, deve riaggiornare tutti i puntatori della lista di ogni file. Nell'allocazione contigua la deframmentazione consiste solamente nel ricompattare tutti i file, eliminando gli spazi vuoti tra essi. Anch'essa è un'operazione molto dispendiosa di tempo.

Problematiche

La problematica maggiore dell'operazione di deframmentazione è la presenza di file che non possono essere spostati. Questi file sono file di paging e di swap, utilizzati dal sistema operativo per la gestione della RAM. Lo spostamento di questi file causerebbe la perdita dei riferimenti per il corretto funzionamento del sistema operativo. Inoltre, poiché il programma di deframmentazione non può spostare sé stesso né file utilizzati da altri programmi, esistono tool che deframmentano il disco rigido prima dell'avvio del sistema operativo, permettendo così la completa deframmentazione.

File system storici e contemporanei

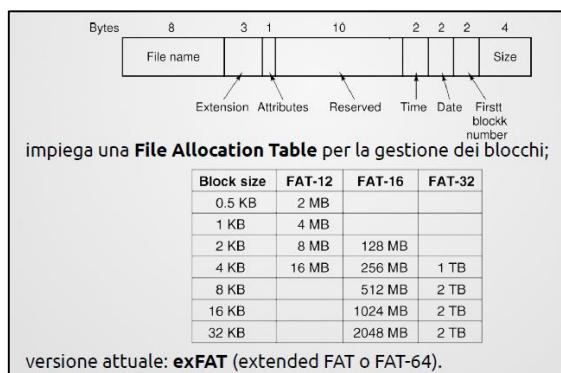
Nel corso della storia informatica, è stata ideata una miriade di file system. I sistemi operativi moderni sono spesso in grado di accedere a diversi file system, spesso semplicemente installando un apposito modulo o driver.

I tipi di file system possono essere classificati in file system per dischi, file system di rete e file system per compiti speciali. Nel gergo comune si è soliti affermare, soprattutto in sistemi GNU/Linux, che il file system viene montato, per consentire al sistema operativo di accedervi per le operazioni di lettura\scrittura.

FAT

La File Allocation Table, in sigla FAT, è un file system sviluppato inizialmente da Bill Gates e Marc McDonald. È il file system primario per diversi sistemi operativi DOS e Microsoft Windows fino alla versione Windows ME. Windows NT e le successive versioni hanno introdotto l'NTFS e mantenuto la compatibilità con la FAT così come molti altri sistemi operativi moderni (Unix, Linux, Mac, ecc.).

La FAT è relativamente semplice ed è supportata da moltissimi sistemi operativi. Queste caratteristiche la rendono adatta ad esempio per i Floppy Disk e le Memory Card. Può anche essere utilizzata per condividere dati tra due sistemi operativi diversi.



Il più grande problema del File System FAT è la frammentazione. Quando i file vengono eliminati, creati o spostati, le loro varie parti si disperdono sull'unità, rallentandone progressivamente la lettura e la scrittura. Una soluzione a questo inconveniente è la deframmentazione, un processo che riordina i file sull'unità. Questa operazione può durare anche diverse ore e deve essere eseguita periodicamente per mantenere le prestazioni dell'unità.

Esistono varie versioni di questo file system, in base a quanti bit sono allocati per numerare i cluster del disco: FAT12, FAT16, FAT32. Esiste anche una versione del FAT16 detta VFAT, che è virtuale cioè non registrato fisicamente sull'hard disk, ma gestito da un software specifico.

NTFS

Sigla di New Technology File System, file system dei sistemi operativi basati su kernel NT.

Questo file system nasce negli anni novanta, quando Microsoft abbandonò lo sviluppo congiunto con IBM del sistema operativo OS/2 e decise di sviluppare in proprio Windows NT (che si presume significhi New Technology). Proprio per questo alcuni degli aspetti presenti nel file system HPFS di OS/2 sono presenti anche in NTFS.

NTFS è un notevole passo avanti rispetto al File Allocation Table (FAT), l'altro file system di Microsoft. Queste le sue principali caratteristiche:

- **Affidabilità** - NTFS è un sistema transazionale (o "Journaled" come si dice nei sistemi operativi Linux e Apple come Mac OS X); questo vuol dire che se un'operazione è interrotta a metà (ad esempio per un blackout) viene persa solo quell'operazione ma non è compromessa l'integrità del file system il quale resta comunque leggibile dal computer.
- **Permessi e Controllo d'Accesso** - a ciascun file o cartella è possibile assegnare dei diritti di accesso (lettura, scrittura, modifica, cancellazione e altri).
- **Nomi lunghi e Unicode** - i nomi dei file e delle cartelle possono essere lunghi fino a 255 caratteri e possono contenere caratteri di tutte le lingue del mondo grazie alla codifica Unicode.
- **Dimensioni e Flessibilità** - La dimensione massima per volume, la dimensione massima per file e il numero massimo di file per volume sono di gran lunga superiori rispetto ai precedenti filesystem di Microsoft: un volume NTFS supporta fino 2³² - 1 file per volume, ha un limite teorico di 264 cluster - 1 e può gestire file che raggiungono i 264 bytes - 1 kb di dimensione; gli ultimi due valori si riducono, tuttavia, in tutte le implementazioni di questo filesystem nei sistemi operativi Windows,[7] rispettivamente a 2³² clusters - 1 e 2⁴⁴ bytes - 64 kb. Di conseguenza - utilizzando cluster di 64 kb - la dimensione massima di un volume NTFS in un sistema Windows è di 256 Tb - 64 kb (che si riduce a 16 Tb - 4kb utilizzando la dimensione standard per cluster di 4 kb). La dimensione massima di un singolo file è di 16 Tb - 64kb contro i 4 Gb di FAT e FAT32. Sono supportati nativamente i volumi sparsi e il mirroring. Sono finalmente disponibili gli hardlink.
- Le **performance** di NTFS sono invece leggermente inferiori a quella di FAT e di FAT32. A partire da Windows 2000, è inoltre possibile montare un volume NTFS come sottodirectory di un altro volume NTFS.

- NTFS permette inoltre di utilizzare trasparentemente delle opzioni di **compressione** (il rapporto di compressione è mediocre, meno di ZIP, ma permette l'accesso immediato a qualunque punto del file) e di crittografia (chiamato anche EFS).
- In NTFS sono stati aggiunti i cosiddetti punti di reparse, ovvero dei meccanismi che consentono le giunzioni (junctions) tra directory, altrimenti impossibili per la struttura del file system.
- **volume** (una o più partizioni o dischi) e **cluster** (unità elementare di alloc.);
- **master file table (MFT)** composta da record (di dimensione fissa);
- **file:** collezione di più attributi di varia natura
 - un attributo può corrispondere ad un metadato del file o ad un flusso;
 - residenti e non residenti;
 - un file può richiedere più record
- directory basata sulla struttura dati **B+ Tree** per ricerche efficienti;
- gestione blocchi liberi basata su **bitmap**;
- **hard-link, soft-link** e montaggio di altri FS (nelle ultime versioni);
- **journaling**;
- **copie shadow** di volumi con tecnica copy-on-write.

NTFS sfrutta un'indicizzazione a 64 bit, sebbene la sua implementazione sia basata soltanto su 32 bit.

La struttura principale di un file system NTFS è la Master File Table (MFT), una tabella strutturata in blocchi (solitamente in record di 1KB) che contiene gli attributi di tutti i file del volume, inclusi i metadati. Tali attributi possono essere attributi residenti quando sono presenti in MFT, oppure, se non memorizzabili a causa del poco spazio, vengono salvati in qualche altra posizione del file system e prendono il nome di attributi non residenti.

Le directory sono memorizzate come file: in ogni file-directory sono presenti degli attributi speciali, memorizzati in ordine lessicografico, che si riferiscono ai file contenuti in tale directory.

I dati veri e propri dei file sono memorizzati in flussi puntati da appositi attributi Data.

EXT

L'Extended File System (ext) fu il primo file system creato specificamente per Linux. Fu progettato da Rémy Card per superare certi limiti del file system Minix. Fu sostituito da ext2 e da xafs, tra i cui ci fu una competizione, vinta da ext2 a causa della sua vitalità.

Il file system ext2 (Second Extended Filesystem) è uno dei file system più diffusi nei vari sistemi operativi open source, in quanto è stato per lungo tempo quello usato dai sistemi GNU/Linux. Attualmente è supportato da tutti i sistemi operativi open source e esistono programmi per utilizzare i volumi ext2 anche da sistemi proprietari come Microsoft Windows (senza supporto Microsoft) e Mac OS X. Benché le sue caratteristiche tecniche lo rendano obsoleto, il suo uso è ancora diffuso per via delle sue buone prestazioni e della grande stabilità che ha dimostrato negli anni. Ext2 non è dotato di nessun tipo di journaling. Inoltre, sebbene nel momento della sua prima scrittura la dimensione massima di un volume fosse di 2GB, ora ext2 supporta volumi di 4TB. Ext2 è stato sostituito da una versione più recente, Ext3 che introduce delle importanti caratteristiche mantenendo la compatibilità completa. L'ultima evoluzione del filesystem (Ext4) introduce modifiche più sostanziali che possono rendere un filesystem Ext4 incompatibile con un filesystem Ext2.

Lo spazio è suddiviso in blocchi e organizzato in **gruppi di blocchi**. Ciò doveva servire a ridurre la frammentazione interna e minimizzare i movimenti della testina del disco durante la lettura di molti dati consecutivi. All'interno di ogni gruppo di blocchi ci sono i superblock, group descriptor, block bitmap, inode bitmap, infine seguiti dai blocchi dei dati. Il superblock contiene informazioni importanti per l'avvio del sistema operativo, per cui vengono fatte copie di backup in ogni gruppo dei blocchi. Solo la prima copia viene utilizzata per l'avvio. Il descrittore del gruppo memorizza il valore del block bitmap, inode bitmap e l'inizio della tabella degli inode per ogni gruppo dei blocchi.

Ext3 è un file system utilizzato su sistemi GNU/Linux e derivato da ext2, rispetto al quale migliora la scrittura su disco rendendo più facile e più veloce leggere i vari file dal disco, inoltre introduce il journaling del file system. Il journaling, già presente in ReiserFS e nelle ultime versioni di NTFS è una caratteristica che permette di evitare che errori e malfunzionamenti hardware (o anche semplici spegnimenti del PC senza chiudere il sistema operativo)

possano danneggiare i dati scritti sull'unità, creando un "diario" (journal) che elenca le modifiche da effettuare sul filesystem.

BTRFS

BTRFS (B-tree FS o "Butter FS") è un file system per Linux di tipo copy-on-write dotato di checksumming (controllo integrità dei dati trasmessi), annunciato da Oracle Corporation nel 2007 e pubblicato sotto la GNU General Public License (GPL).

Nato inizialmente come risposta al filesystem ZFS, è il filesystem scelto da MeeGo, il sistema operativo open source sviluppato da Intel e Nokia per smartphone, netbook e navigatori satellitari. Esso è disponibile in diverse distribuzioni Linux, come ArchLinux dal gennaio 2012, Debian dalla versione 6, Fedora dalla versione 16, Gentoo Linux dal mese di luglio 2010, OpenSUSE a partire dalla versione 12.1, e Ubuntu dalla versione 10.04. La prima comparsa di Btrfs nel kernel Linux risale alla prerelease 2.6.29-rc1 del kernel Linux, mentre la prima versione definita stabile è stata pubblicata nella versione 3.10. Lo sviluppo del filesystem procede tuttora.

Caratteristiche addizionali rispetto ai precedenti FS Linux:

- clonazione di singoli file con copy-on-write;
- sottovolumi;
- snapshot dei sottovolumi con copy-on-write;
- primitive di send/receive su differenze tra snapshot;
- directory basate su strutture dati B-Tree;
- deframmentazione, aumento/riduzione on-line di volumi;
- checksum sui blocchi di dati e dei metadati;
- compressione trasparente dei file;
- supporto a meccanismi tipo-RAID (mirror, striping) all'interno stesso del volume.

In un disco con blocchi di 2 Kbyte (= 2^{11} byte), è definito un file system FAT 16. Ogni elemento ha lunghezza di 2 byte e indirizza un blocco del disco. La copia permanente della FAT risiede nel disco a partire dal blocco di indice 0 e una copia di lavoro viene caricata in memoria principale all'avviamento del sistema operativo. Supponendo che la FAT sia dimensionata in base alla massima capacità di indirizzamento dei suoi elementi si chiede:

- 1) *il numero di blocchi dati indirizzabili dalla FAT.*
- 2) *il numero di byte e di blocchi del disco occupati dalla FAT,*
- 3) *l'indice del primo blocco dati nel disco,*
- 4) *quale capacità (in blocchi e in byte deve avere il disco) per contenere tutti i blocchi dati indirizzabili.*

1). Il numero di blocchi dati indirizzabili dalla FAT è 2^{16}

2). La FAT ha 2^{16} elementi di 2 byte pertanto occupa 2^{17} byte è $2^6 = 64$ blocchi

3). Il primo blocco dati nel disco è quello di indice 64 (i blocchi precedenti sono riservati alla FAT)

4). Per contenere tutti i blocchi indirizzabili, il disco deve avere una capacità di almeno $2^{16} + 26$ blocchi e 227 + 217 byte.

Scheduling del disco

Lo scheduling del disco serve per decidere in fase di lettura (o scrittura) quali blocchi verranno letti prima e quali blocchi verranno letti un secondo momento.

Esistono vari algoritmi di scheduling del disco, che hanno come obiettivi massimizzare il numero di richieste soddisfatte in una unità di tempo (**throughput**), e minizzare il tempo medio di accesso.

In un sistema, soprattutto se multiprogrammato, si vengono a creare varie richieste di I/O su disco che però, tipicamente, possono essere inviate al controller del disco solo una alla volta. Si crea quindi una **coda di richieste pendenti**. Il SO può adottare varie politiche di selezione della prossima richiesta da mandare; si può ottimizzare per:

- Tempo di posizionamento (**seek-time**, tempo che occorre alla testina per posizionarsi sulla successiva traccia del disco per leggere o scrivere i dati);
- **Latenza di rotazione** (una volta che la testina ha raggiunto la traccia corretta (seconda fase) è necessario attendere che il settore desiderato si muova (ruotando) sotto la testina di lettura/scrittura; questo tempo viene chiamato latenza di rotazione o ritardo di rotazione)

Algoritmi di scheduling su disco

Per tutti gli esempi si seguirà il seguente schema:

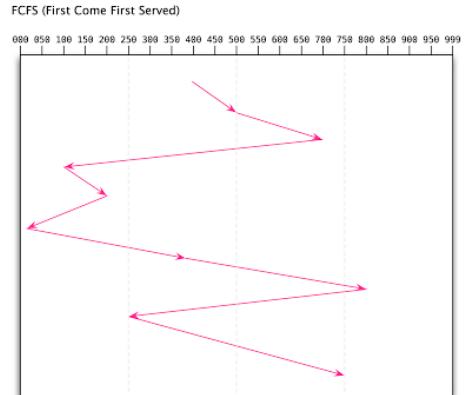
- Dimensione del disco: 1000 cilindri, numerati da 0 a 999
- Coda delle richieste: 500, 700, 100, 200, 10, 390, 800, 250, 750
- Posizione iniziale della testina sul cilindro: 400
- Spostamento traccia da n a n±1: 1ms

FCFS (First Come First Served)

Come dice il nome vengono servite le richieste nell'ordine in cui arrivano.

Nel nostro esempio l'ordine di chiamata è: 500, 700, 100, 200, 10, 390, 800, 250, 750. Quindi: $(500-400) + (700-500) + (700-100) + (200-100) + (200-10) + (390-10) + (800-390) + (800-250) + (750-250) = 3030$. Il tempo per soddisfare le richieste è: 3030ms

Tale algoritmo è equo, semplice da realizzare ma inefficiente.

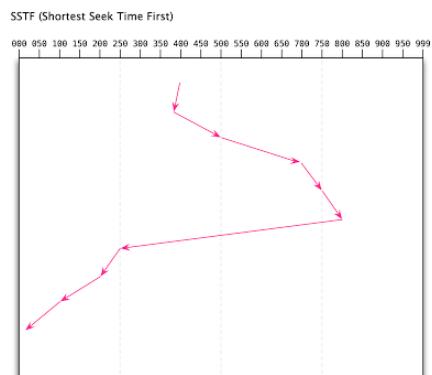


SSTF (Shortest Seek Time First)

Questo algoritmo di scheduling sceglie la richiesta con seek time minore rispetto alla posizione corrente (prima soddisfa quelle vicine alla posizione iniziale della testina sul cilindro).

Nota: Per comodità riordiniamo la lista delle richieste [10, 100, 200, 250, 390, 500, 700, 750, 800]. Nel nostro esempio l'ordine di chiamata è: 390, 500, 700, 750, 800, 250, 200, 100, 10. Il tempo per soddisfare le richieste è: $10+110+200+50+550+50+100+90 = 1210$ ms

Tale algoritmo ha buone prestazioni (non ottimali) ma non è equo (rischio starvation).

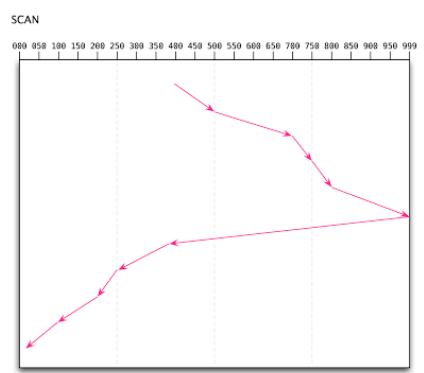


SCAN (scheduling per scansione)

L'algoritmo SCAN serve in una direzione e quando arriva al bordo del disco cambia direzione servendo i rimanenti elementi (detto anche algoritmo dell'ascensore).

Nel nostro esempio l'ordine di chiamata è: 500, 700, 750, 800, (999), 390, 250, 200, 100, 10. Il tempo per soddisfare le richieste è: 1588ms

Tale algoritmo garantisce comunque una attesa massima per ogni richiesta, ma si può fare di meglio.

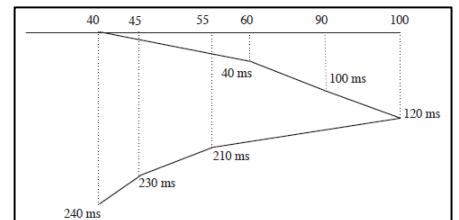


(Es. con tempi di arrivo diversi) Si consideri un disco con un intervallo di tracce da 0 a 100, gestito con politica SCAN. Inizialmente la testina è posizionata sul cilindro 40; lo spostamento ad una traccia adiacente richiede 2 ms. Al driver di tale disco arrivano richieste per i cilindri 90, 45, 40, 60, 55, rispettivamente agli istanti 0 ms, 20 ms, 30 ms, 40 ms, 80 ms. Si trascuri il tempo di latenza. (1) In quale ordine vengono servite le richieste? (2) Il tempo di attesa di una richiesta è il tempo che intercorre dal momento in cui è sottoposta al driver a quando viene effettivamente servita. Qual è il tempo di attesa medio per le cinque richieste in oggetto?

Assumendo una direzione di movimento ascendente all'istante 0, le richieste vengono servite nell'ordine 60, 90, 55, 45, 40

Il tempo di attesa medio per le cinque richieste in oggetto è

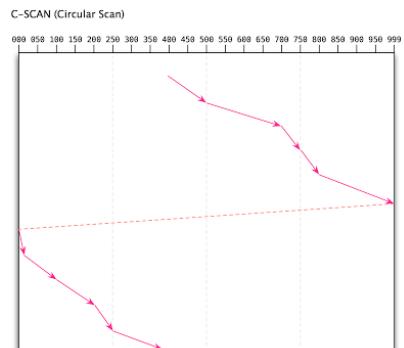
$$\frac{(40-40)+(100-0)+(210-80)+(230-20)+(240-30)}{5} = \frac{0+100+130+210+210}{5} = 130 \text{ ms}$$



C-SCAN (scheduling per scansione circolare)

La testina segue le richieste in ordine (da una parte o dall'altra, in genere si procede seguendo un ordine crescente) finché non arriva all'estremo del disco. Una volta arrivato all'estremo del disco torna all'inizio (senza servire richieste) e riparte. Considera le posizioni come collegate in modo circolare, e arrivato alla fine del disco torna sul primo cilindro senza servire alcuna richiesta.

Nel nostro esempio l'ordine di chiamata è: 500, 700, 750, 800, (999), (0), 10, 100, 200, 250, 390. Il tempo per soddisfare le richieste è: 989ms.

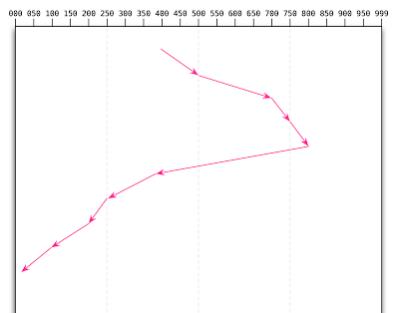


Tale algoritmo garantisce un tempo medio di attesa più uniforme.

LOOK

Il funzionamento è identico all'algoritmo di SCAN. L'unica differenza è che con questo algoritmo non è necessario raggiungere i bordi del disco.

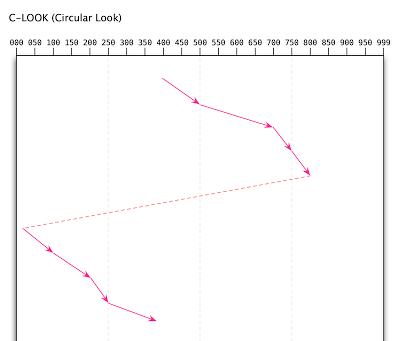
Nel nostro esempio l'ordine di chiamata è: 500, 700, 750, 800, 390, 250, 200, 100, 10. Il tempo per soddisfare le richieste è: 1190ms.



C-LOOK (Circular LOOK)

Questo algoritmo è una variante dell'algoritmo C-SCAN. Con questo algoritmo non è necessario arrivare fino ai bordi del disco.

Nel nostro esempio l'ordine di chiamata è: 500, 700, 750, 800, (790), 10, 100, 200, 250, 390. Il tempo per soddisfare le richieste è: 790ms.



Come scegliere?

- C-LOOK per alto carico;
- LOOK o SSTF per basso carico.

Sistemi RAID

Gli utenti necessitano di dischi con maggiore capienza, accesso ai dati più veloce, tassi di trasferimento maggiori e maggiore affidabilità. Il **RAID** (redundant array of inexpensive disks) è una tecnica di raggruppamento di diversi dischi rigidi collegati ad un computer che li rende utilizzabili, dalle applicazioni e dall'utente, come se fosse un unico volume di memorizzazione.



Tale aggregazione sfrutta, con modalità differenti a seconda del tipo di implementazione, i principi di ridondanza dei dati e di parallelismo nel loro accesso per garantire, rispetto ad un disco singolo, incrementi di prestazioni, aumenti nella capacità di memorizzazione disponibile, miglioramenti nella tolleranza ai guasti.

L'architettura RAID è stata introdotta per migliorare l'affidabilità e le prestazioni del sistema dei dischi in un sistema di calcolo. Infatti, aumentando il numero di dischi fisici presenti in un calcolatore, diminuisce proporzionalmente il tempo medio fra un guasto e l'altro. Quindi il sistema RAID introduce degli schemi di ridondanza dei dati per consentire il funzionamento dei dischi anche in situazioni critiche.

Il RAID è una tecnica tipicamente impiegata nei server o nelle workstation che richiedano grandi volumi o elevate prestazioni di immagazzinamento di dati, per esempio per ospitare una base di dati o una postazione di montaggio di audio o video digitali. Il RAID si trova comunemente anche nei NAS e, sempre, nei sistemi di storage per architetture blade.

Per aumentare le prestazioni bisogna sfruttare il parallelismo anche per l'I/O su disco: servono più dischi indipendenti e bisogna suddividere i dati relativi ad una unità logica (un file, in generale un volume) su più dischi (*striping*).

Funzionamento

Questa tecnologia usa un insieme di dischi rigidi per condividere o replicare i dati e ha come obiettivi principali l'aumento delle prestazioni e una migliore affidabilità. Per fare ciò distribuisce i dati coinvolti in un'operazione di I/O su diversi dischi ed esegue le operazioni di I/O su questi dischi in parallelo. Abbiamo suddivisione dei dati trasparente all'utente, ma aumenta la probabilità che si verifichi un guasto sul volume logico RAID (soluzione: aggiungiamo ridondanza per ottenere migliore affidabilità; sostituzione automatica: dischi spare).

Questa caratteristica può fornire accessi veloci o alti tassi di trasferimento, in base alla configurazione adottata. L'alta affidabilità è ottenuta memorizzando informazioni ridondanti. L'accesso alle informazioni ridondanti non necessita di tempo di I/O aggiuntivo in quanto si può accedere in parallelo sia ai dati che alle informazioni ridondanti.

Da notare che la gestione di un RAID è più dispendiosa della gestione di un unico disco poiché vi sono più dischi da controllare.

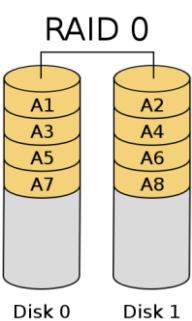
Configurazioni

Esistono diverse configurazioni RAID che utilizzano differenti tecniche di ridondanza e organizzazione. Queste tecniche sono chiamate livelli RAID e, ad oggi, vanno da 0 a 6. Tutti i livelli RAID hanno però delle caratteristiche in comune:

- I vari hard disk configurati in RAID sono visti dal SO come un unico disco
- I dati sono divisi in strisce (*stripes*) distribuite sui vari dischi; una strip è memorizzata su più dischi nella stessa posizione in modo che, sincronizzati i dischi, è possibile leggere una striscia simultaneamente
- La capacità di ridondanza del disco è usata per memorizzare informazioni di parità che permettono il recupero dei dati in caso di guasti

Il RAID può essere implementato sia con hardware dedicato (controllore RAID) sia con software specifico. Nel primo caso, deve essere disponibile almeno un controllore RAID; nei PC desktop, questo può essere una scheda PCI o può essere il controller integrato nella scheda madre. Nel secondo caso, è il SO che gestisce l'insieme di dischi attraverso un normale controller. Questa opzione può essere più lenta di un RAID hardware, ma non richiede l'acquisto di componenti extra.

RAID di livello 0



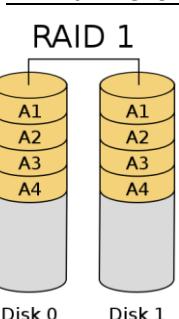
Nel RAID 0 i dati vengono distribuiti su più dischi con la tecnica dello **striping**: i dati sono divisi equamente tra due o più dischi con nessuna informazione di parità o ridondanza. Quindi questa non è proprio una configurazione RAID poiché non usa nessuna memorizzazione ridondante dei dati.

Le operazioni sono svolte in parallelo sui vari dischi; ad esempio, se viene inviato un comando di lettura di un blocco di dati, tale comando si suddivide in più comandi, ognuno per ciascun disco interessato. In questo modo la lettura avviene in parallelo. Il RAID 0 risulta indicato quando le esigenze di performance superano quelle dell'affidabilità.

VANTAGGI: garantisce tassi di trasferimento elevati grazie agli accessi contemporanei ai vari dischi.

SVANTAGGI: ha una scarsa affidabilità. I dati diventano inaccessibili anche nel caso di un singolo disco spento. Inoltre la mancanza di ridondanza causa la perdita dei dati nel caso di malfunzionamento di un disco.

RAID di livello 1



Nel RAID 1 viene creata una copia esatta di tutti i dati su due o più dischi. Questa tecnica prende il nome di **disk mirroring**.

Se un dato viene aggiornato da un processo, una copia di questo dato viene copiata su ogni disco. Ciò aumenta l'affidabilità rispetto al sistema a disco singolo. Infatti, se si verifica un malfunzionamento ad uno dei dischi, un dato può essere recuperato dagli altri dischi.

Il RAID 1 non porta vantaggi nella scrittura ma porta notevoli miglioramenti nella lettura. Infatti, anche nel RAID 1 le operazioni sono svolte in parallelo aumentando, così, le prestazioni in lettura:

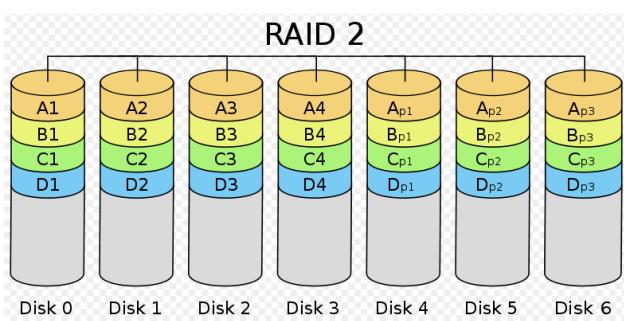
durante una lettura, viene utilizzata la copia che può essere letta prima; inoltre è possibile leggere da un disco mentre l'altro è occupato.

Da notare il fatto che il sistema può avere una capacità massima pari a quella del disco più piccolo.

VANTAGGI: Maggiore affidabilità visto che vengono create tante copie dei dati quanti sono i dischi. Migliori prestazioni in lettura, visto che il dato viene letto dal disco a cui si può accedere prima.

SVANTAGGI: Viene prodotto molto overhead visto che ad ogni aggiornamento di un dato, devono essere aggiornati anche le sue copie presenti sugli altri dischi. Il sistema vede un unico disco, ma con una capienza pari a quella del disco fisico più piccolo.

RAID di livello 2



Nel RAID 2 viene utilizzata la tecnica del **bit striping**: si memorizza ogni bit (invece che i blocchi) di dati ridondanti su un disco differente. Inoltre vengono salvati su altri dischi, i codici per la correzione degli errori. Solitamente si usa il codice di *Hamming* visto che esso è in grado di correggere errori su singoli bit e rilevare errori doppi.

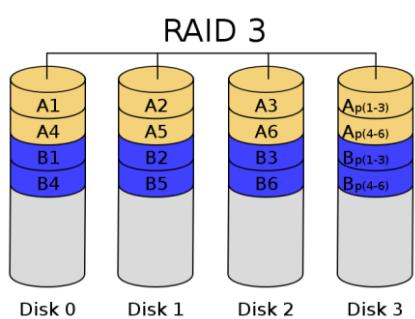
In pratica, una parte dei dischi viene usata per salvare i dati, mentre la restante parte viene usata per salvare solo le informazioni per la correzione degli errori.

In sostanza, il RAID 2 è un RAID 0 maggiormente affidabile. Infatti questo sistema risulta molto efficiente negli ambienti in cui si verificano numerosi errori di lettura o scrittura.

VANTAGGI: Affidabilità molto alta. Velocità di lettura molto alta: visto che i dati sono spezzettati su molti dischi, la possibilità di leggere in parallelo i vari dischi, aumenta notevolmente la velocità di lettura.

SVANTAGGI: Il numero di dischi utilizzati da questo sistema può essere molto alto.

RAID di livello 3



Il RAID 3 non è altro che una semplificazione del RAID 2. Infatti la differenza sostanziale tra i due livelli è che nel RAID 3 viene utilizzato un solo disco per il salvataggio dei codici per la correzione. Questo è possibile sostituendo il codice di Hamming con un nuovo metodo, il **bit di parità**.

Un sistema RAID 3 usa una divisione al livello di byte.

Ogni operazione di I/O richiede di utilizzare tutti i dischi. Questa caratteristica fornisce elevati tassi di trasferimento; tuttavia, non possono essere eseguite operazioni multiple contemporaneamente, ma solo un'operazione di I/O può

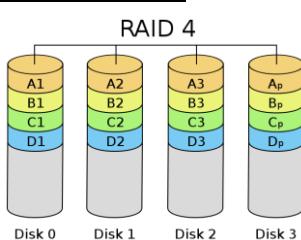
essere eseguita in ogni istante. Nella figura in alto, una richiesta per il blocco A richiederà di cercare attraverso tutti i

dischi; una richiesta simultanea per il blocco B rimarrà invece in attesa.

VANTAGGI: Prestazioni e caratteristiche simili al RAID 2. Elevati tassi di trasferimento. Affidabilità molto alta. Un solo disco viene utilizzato per memorizzare i codici per la correzione degli errori.

SVANTAGGI: Possibilità di eseguire una sola operazione di I/O in ogni istante.

RAID di livello 4



Il RAID 4 è analogo al RAID 3 fatta eccezione per il fatto che usa una divisione a livello di blocchi con un disco dedicato alla memorizzazione dei codici di parità. La differenza sostanziale sta proprio nel modo in cui i dati vengono divisi: nel RAID 3 sono divisi a livello di byte, nel RAID 4 sono divisi al livello di blocchi.

Per le operazioni di lettura molto grandi, cioè quando sono richiesti più blocchi, il RAID 4 si comporta in modo analogo al RAID 3: l'operazione di I/O richiede di utilizzare tutti i

dischi; ciò porta ad un alto tasso di trasferimento, ma anche al fatto di non poter eseguire più operazioni di I/O contemporaneamente.

Per le operazioni di lettura piccole, cioè quando è richiesto un solo blocco, il RAID 4 si comporta diversamente: l'operazione di I/O richiede di utilizzare un singolo disco; ciò porta a bassi tassi di trasferimento, ma anche la possibilità di eseguire più operazioni di I/O contemporaneamente. Nella figura, una richiesta per il blocco A1 potrebbe essere evasa dal disco 1; una richiesta simultanea al blocco B1 dovrebbe aspettare, ma una richiesta per il blocco B2 potrebbe essere servita allo stesso momento dal disco 2.

Un'operazione di scrittura necessita del calcolo delle informazioni di parità tenendo conto di tutti i blocchi memorizzati sui vari dischi. In pratica, quando si vuole scrivere, sono coinvolti tutti i dischi anche se la scrittura riguarda un solo blocco; ciò impedisce al sistema di effettuare operazioni in parallelo.

VANTAGGI: Possono essere effettuate più operazioni di I/O contemporaneamente se le operazioni di lettura sono piccole. Il tasso di trasferimento è più alto se le operazioni di lettura sono grandi, poiché sono coinvolti più dischi. Affidabilità molto alta. Un solo disco viene utilizzato per memorizzare i codici per la correzione degli errori.

SVANTAGGI: C'è la possibilità di effettuare una sola operazione di scrittura alla volta, sia se l'operazione di scrittura è grande, sia se piccola. Il disco di parità è un collo di bottiglia in quanto l'operazione di scrittura impedisce al sistema di effettuare operazioni in parallelo.

Il raid 4 prevede lo striping a livello di blocchi con XOR sull'ultimo disco; quindi usa l'operazione XOR (11=0, 10=1, 01=1, 00=0) per trovare il bit di parità degli stripe dell'ultimo disco.

Dati i seguenti dischi:

Disco 1	Disco 2	Disco 3
0010	1001	101x
1001	0101	110x
0110	0100	001x

Trovare i bit di parità x

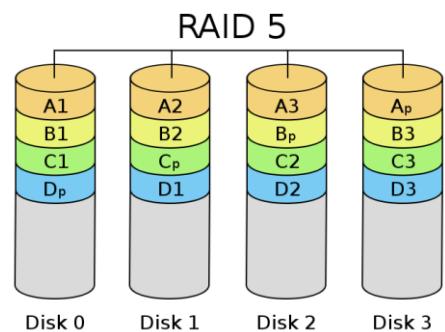
x = 011 dato che con lo xor risultano tali valori

RAID di livello 5

Il RAID 5 è analogo al RAID 4 fatta eccezione per il fatto che i codici di parità sono distribuiti su tutti i dischi. In questo modo, se l'operazione di scrittura interessa un singolo blocco, il sistema ha comunque la possibilità di effettuare altre operazioni di scrittura in parallelo.

Per quanto riguarda la lettura, è tutto come nel RAID 3 e nel RAID 4:

- Operazioni di lettura grandi portano ad un alto tasso di trasferimento ma all'impossibilità di effettuare operazioni in parallelo
- Operazioni di lettura piccole portano a bassi tassi di trasferimento ma alla possibilità di effettuare operazioni in parallelo



VANTAGGI: Possibilità di effettuare scritture in parallelo se queste richiedono un solo blocco, quindi il disco di parità non fa da collo di bottiglia. Possibilità di effettuare letture in parallelo se queste richiedono solo un blocco.

Alti tassi di trasferimento se le operazioni di I/O richiedono più blocchi. Affidabilità molto alta. Un solo disco viene utilizzato per memorizzare i codici per la correzione degli errori.

SVANTAGGI: Possibilità di eseguire una sola operazione di I/O in ogni istante se vengono richiesti più blocchi. Difficile da implementare.

Come il raid 5, il raid 4 prevede lo striping a livello di blocchi, ma con informazioni di parità distribuite; quindi per ricostruire un blocco mancante in RAID-5 (che lavora in modo analogo al RAID-4 ma con le parità distribuite) si deve procedere a fare lo XOR bit a bit degli strip rimanenti:

Dati i seguenti dischi:

101	010	110	100	110
100	111	100	010	110
110	100	110	-	111
111	010	001	101	1100

Trovare gli stripe del disco con il simbolo - .

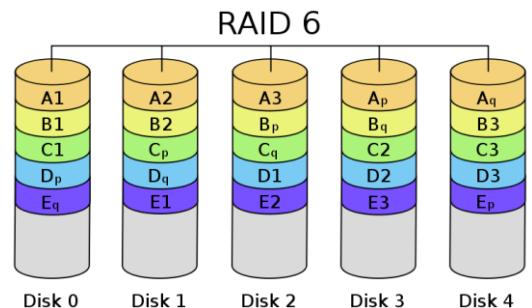
Per ricostruire il blocco mancante, devo fare solamente lo XOR fra gli elementi delle terza (3a) riga; quindi:

$$111 \text{ XOR } 110 = 001 \text{ XOR } 100 = 101 \text{ XOR } 110 = 011$$

RAID di livello 6

Un RAID 6 usa una divisione a livello di blocchi con i dati di parità distribuiti due volte tra tutti i dischi. Risulta più ridondante del RAID 5 e il throughput è leggermente più alto a causa dell'esistenza di un disco in più rispetto al RAID 5.

Supporta il ripristino dal guasto di due dischi.



Memorie Flash e File System

I dispositivi basati su memorie flash (tecnologia NAND) funzionano in modo molto differente dai dischi elettromeccanici. Un blocco si deve **cancellare prima** di poter essere riscritto, e il numero di cancellature (quindi di scrittura) è limitato per ogni blocco. Ogni blocco è composto da **pagine**, infatti l'allocazione è basata su pagine.

I file-system che abbiamo visto non sono stati progettati con questi limiti in mente. I primi tentativi di progettazione sono stati i Flash-Friendly File System (F2FS), e i controller che rimappano i blocchi.

Garbage Collection (GC) sui SSD e operazione TRIM:

- Degrado delle prestazioni nel tempo se non impiegato;
- Richiede adeguamento da parte del SO.

Il TRIM non è la stessa cosa del Garbage Collector; hanno lo stesso scopo (preservare le performance e la vita dell'SSD) ma sono due funzioni diverse. Le finalità sono le stesse (recuperare il degrado prestazionale) ma operano in modo completamente diverso.

Comandi della Shell Unix

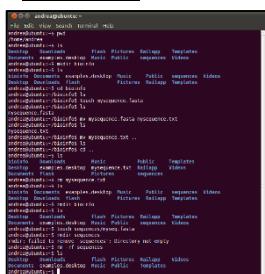
Struttura di un SO Unix

Autenticazione

A differenza del sistema MS-DOS e dei primi sistemi Windows, i sistemi UNIX sono stati da sempre **multi-utente**. La stessa macchina può essere utilizzata da più utenti contemporaneamente (localmente e/o in remoto). Ad ogni utente è associato un **username** ed una **password** che permettono di identificarlo al momento del login. I dati di tutti gli utenti del sistema sono memorizzati all'interno del file **/etc/passwd**. Tutto il sistema è basato sui diritti di accesso: ogni oggetto (file, periferica, processo) ha delle **strutture di accesso** che decidono quali operazioni un dato utente è abilitato a fare. Queste strutture prevedono la suddivisione in **gruppi logici** (con intersezioni) per una maggiore flessibilità.

Ogni utente ha un proprio ambiente di lavoro perfettamente isolato dagli altri. Un utente, in genere, non può interferire nel lavoro degli altri utenti. Esiste un utente particolare, chiamato **root**, che ha i massimi privilegi sul sistema: può fare quello che vuole. In genere `e l'account dell'amministratore di sistema.

Shell Unix



L'interprete dei comandi UNIX è chiamata shell. Sebbene non sia parte del SO, ne utilizza molte caratteristiche e serve quindi come buon esempio di come possano essere usate le syscall. La shell è anche l'interfaccia principale tra un utente seduto al suo terminale ed il SO, a meno che l'utente non stia usando un'interfaccia grafica. Quando l'utente si collega al sistema, viene fatta partire una shell. La shell usa il terminale come ingresso e uscita standard e parte scrivendo un **prompt**, un carattere come il segno di dollaro, che dice all'utente che la shell è in attesa di comando.

Esempio di prompt dei comandi

utente@host:directory\$

Premiamo INVIO per mandare in esecuzione il comando.

Combinazione di tasti utili:

Ctrl-C: interrompe l'esecuzione di un comando

Ctrl-S: blocca l'output che scorre sullo schermo

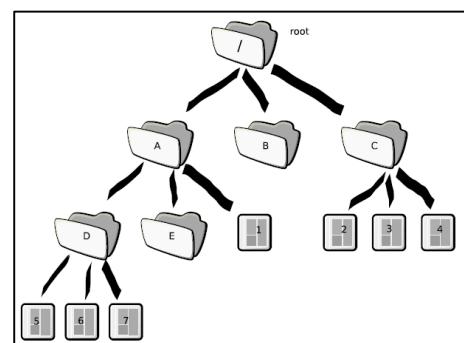
Ctrl-Q: sblocca lo stesso output

File System

Struttura ad albero mantenuta su disco che contiene tutti i dati del SO e dei suoi utenti. Ogni file system ha una directory speciale che contiene tutti gli altri oggetti del filesystem: **root** (radice).

Per lavorare con gli oggetti del file system ci serve poterli identificare in un modo ben preciso. Vengono utilizzati due metodi principali:

- **Path assoluto** di un file: percorso che va dalla radice del filesystem allo stesso. *Esempio: /home/utente/file.txt*
- **Path relativo** di un file: va dalla directory ad esso. *Esempio: utente/documenti/cv.pdf*



In effetti il percorso assoluto di un file è il suo percorso relativo rispetto alla root.

Nella ricostruzione di un path

la cartella . indica la cartella stessa (auto-referenziamento)

la cartella .. indica la cartella genitore

File di dispositivo

File che identificano una periferica del sistema e attraverso essi il SO può interagire con tale periferica. In genere risiedono nella cartella **/dev/**. Ne esistono di due tipi:

- **A caratteri**: le operazioni sul dispositivo vengono fatte per flussi di input/output con il carattere come unità di base. *Esempio*: tastiera, stampante, scheda audio;
- **A blocchi**: si opera con modalità ad accesso casuale e si agisce per blocchi. In genere riguarda i dischi fissi ed i CD/DVD

In genere in un sistema esistono **più di un filesystem**: ne necessita uno per un eventuale CD-ROM inserito nel lettore, uno per ogni partizione sul disco, ecc. Sui sistemi MS-DOS/Windows siamo abituati ad identificare ogni possibile file system con una diversa lettere dell'alfabeto: A:, C:, ... Nei sistemi UNIX tutto questo viene evitato montando i file system in **cascata**. Esiste un file system principale (in genere quello da cui viene caricato il SO) e la sua radice sarà la root di tutto il sistema. Ogni filesystem secondario viene “**montato**” come se fosse un ramo dell'albero principale.

In questo modo si viene a creare un unico grande albero ed ogni file del sistema può essere identificato con un path assoluto che parte dalla root del sistema **/**. In genere i filesystem secondari vengono montati della cartella standard **/mnt/** e quelli delle unità removibili (tipo floppy-disk o chiavette USB) in **/media/**.

Tipico file system Unix

Ecco alcune directory standard che si trovano su tutti i sistemi UNIX:

- **/home/**: ogni utente ha una propria home directory in cui i propri dati e le sue configurazioni dei programmi vengono mantenute (separatamente da quelli degli altri utenti); tutte le home degli utenti stanno in **/home/**;
- **/etc/**: vi risiedono tutti i file di configurazione dei programmi installati compresi gli script di avvio del sistema;
- **/bin/**: contiene i file eseguibili di buona parte delle utility del sistema;
- **/sbin/**: contiene le utility destinate all'amministrazione che solo l'amministratore pu`o utilizzare;
- **/lib/**: contiene le librerie di sistema;
- **/proc/**: una directory virtuale che rappresenta alcuni aspetti interni del S.O., tipo processi, periferiche, ecc.
- **/usr/**: contiene gli eseguibili e i dati di tutto il resto dei programmi installati;
- **/var/**: contiene alcuni database, cache e dati temporanei.

Tipi di comandi

I comandi che si possono invocare da una shell sono di due tipi:

- **Comandi esterni**: si tratta di applicazioni indipendenti dalla shell che sono memorizzate all'interno del file system ed eseguite indipendentemente dal processo della shell; possono andare dalla semplice utility testuale ad vere e proprie applicazioni grafiche complete;
- **Comandi builtin**: vengono eseguiti dalla shell stessa senza coinvolgere processi aggiuntivi. Tipicamente questi comandi vengono usati per manipolare le impostazioni interne della shell e per aumentare l'efficienza dei comandi usati più frequentemente. Fanno parte dei comandi builtin, inoltre, anche tutti i costrutti del linguaggio di **scripting** della shell.

Parametri

Tipicamente un comando può richiedere dei parametri. Si inseriscono dopo il nome del comando separati da uno spazio. Questi parametri possono essere:

- **file**: il percorso (assoluto o relativo) al/ai file su cui il comando deve lavorare;
- **dati**: identificano dei dati da passare ed in genere sono passati sotto forma di stringa con una formattazione opportuna;
- **switch o opzioni**: sono dei parametri che, usualmente, iniziano con un carattere ‘-’. Danno al comando alcune indicazioni sull'esecuzione del suo compito. Spesso la stessa opzione ha più di una sintassi: una breve ed una lunga.
 - Quella breve è in genere del tipo **-h** (dove h `e un singolo carattere).
 - Quella lunga inizia con un doppio trattino **--** ed ha in genere un nome più mnemonico, tipo **--help**.

Spesso le opzioni si possono collassare: al posto di due opzioni **-a -b**, si può spesso utilizzare **-ab**. Alcuni parametri sono opzionali e nella sintassi vengono indicati tra parentesi quadre **[]**. Esempio: **cal [[mese] anno]**.

Aiuto

I comandi UNIX sono tanti e le opzioni sono centinaia. Generalmente ogni comando è capace di fornire una breve descrizione delle sue funzionalità e delle sue principali opzioni quando viene impiegata l'opzione **-h** o **--help**. Per avere più aiuto ci sono altri metodi:

- **apropos stringa**: Visualizza la lista di tutti i comandi che contengono la stringa passata per parametro nella loro descrizione sintetica;
- **whatis nome_comando**: Visualizza la descrizione sintetica del comando passato come parametro;
- **man nome_comando**: Visualizza la pagina del manuale in linea relativa al comando passato come parametro.

Comandi di spostamento

Lista dei file

ls [-l] [-a] [-R] [pathname]

- **[-l]** visualizza informazioni dettagliate
 - flag speciale: directory (d), soft link (l) o file (-)
 - tripla permessi di accesso: possono essere: lettura (r), scrittura (w) e esecuzione; essi riguardano:
 - proprietario file
 - appartenenti al gruppo
 - tutti gli altri.
 - numero di hard link
 - proprietario
 - gruppo
 - dimensioni
 - data e ora di creazione
 - nome del file
- **[-a]** visualizza file nascosti
- **[-R]** visualizza anche il contenuto delle cartelle presenti ricorsivamente
- **[pathname...]**: oggetto del file system su cui viene lanciato il comando

```
$ ls
cartella  esempio2.txt  esempio.txt  script.sh

$ ls -la
drwxr-xr-x  3 mario mario 4096 2005-09-20 12:30 .
drwxr-xr-x  3 mario mario 4096 2005-09-20 12:28 ..
drwxr-xr-x  2 mario mario 4096 2005-09-20 12:30 cartella
-rw-r--r--  1 mario mario   27 2005-09-20 12:30 esempio2.txt
-rw-r--r--  1 mario mario   21 2005-09-20 12:29 esempio.txt
-rwxr-xr-x  1 mario mario   10 2005-09-20 12:30 script.sh

$ ls -R
.:
cartella  esempio2.txt  esempio.txt  script.sh
./cartella:
agenda.txt  eseguimi.sh

$ ls -l cartella/
-rw-r--r--  1 mario mario 21 2005-09-20 12:33 agenda.txt
-rwxr-xr-x  1 mario mario 10 2005-09-20 12:33 eseguimi.sh
```

Le triple di permessi sono appunto 9 caratteri in gruppi di 3 che definiscono i permessi per proprietario, gruppo e altri utenti. Ad esempio nella stringa **rw-r--r-- 1 root suser** abbiamo un file dove il proprietario può leggere e scrivere (rw-) , il gruppo può solo leggere (r-) ed infine tutti quelli che non fanno parte del gruppo possono solo leggere (r-).

Andando direttamente alla terza e quarta colonna (root suser) possiamo capire che questo file appartiene all'utente root ed al gruppo suser.

Metacaratteri

Usati per abbreviare il nome di un file o per specificarne più di uno

- * stringa di 0 o più caratteri
- ? qualunque carattere
- [] singolo carattere tra quelli elencati
- { } stringa tra quelle elencate

```
$ ls
esempio2.txt  esempio3.txt  esempio.txt  script.sh scheda.pdf documento.pdf prova.sh

$ ls esempio*.txt
esempio2.txt  esempio3.txt  esempio.txt

$ ls exemplo?.txt
esempio2.txt  esempio3.txt

$ ls *.{txt,pdf}
documento.pdf  esempio2.txt  esempio3.txt  esempio.txt  scheda.pdf

$ ls /dev/tty[acd][2-5]
/dev/ttya2  /dev/ttya4  /dev/ttyc2  /dev/ttyc4  /dev/ttyd2  /dev/ttyd4
/dev/ttya3  /dev/ttya5  /dev/ttyc3  /dev/ttyc5  /dev/ttyd3  /dev/ttyd5
```

Cambiamento di directory

cd [pathname]

Se non si mette il pathname viene impostata la **home directory** che in genere è **/home/utente**.

Present working directory

pwd

Visualizza il pathname assoluto della directory corrente

Creare cartelle

mkdir [-p] pathname

- [-p] non genera errori se il pathname esiste già

Rimuovere cartelle

rmdir [-p] pathname

Le directory devono essere vuote altrimenti non vengono cancellate

- [-p] rimuovi tutte le directory che contengono il pathname

Copiare dei file

cp [-R] [-i] source dest

- [-R] copia tutto il contenuto di source se è una directory
- [-i] chiede conferma prima di sovrascrivere i file
- source oggetti da copiare in dest

Cancellare dei file

rm [-r] [-i] [-f] pathname

- [-r] se pathname è una directory, elimina ricorsivamente tutti i file e le cartelle all'interno
- [-i] chiede conferma prima di cancellare
- [-f] cancella senza chiede conferma

Spostare dei file

mv source dest

Spostare una directory in un'altra implica lo spostamento di tutto il sottoalbero della directory sorgente

Comandi di stampa di messaggi a video

Redirezione dell'input

Ogni processo ha 3 flussi di dati:

- **Standard input**: da cui prende il suo input; in genere corrisponde alla tastiera;
- **Standard output**: in cui invia il suo output; in genere corrisponde al terminale video;
- **Standard error**: in cui emette gli eventuali messaggi di error; anche qui si usa in genere il terminale.

Attraverso l'invocazione da riga di comando è possibile redirezionare tali flussi su altri file.

- > : redireziona l'output su un file
- >> : redireziona l'output su un file in modalità append;
- < : prende l'input da un file;
- 2> : redireziona lo standard error su un file.

```
$ ls > output.txt
$ cat output.txt
esempio2.txt  esempio3.txt  esempio.txt
$ echo Ciao >> output.txt
esempio2.txt  esempio3.txt
$ cat output.txt
esempio2.txt  esempio3.txt  esempio.txt
Ciao
$ cat < output.txt
esempio2.txt  esempio3.txt  esempio.txt
Ciao
$ man comandochenonesiste
Non c'è il manuale per comandochenonesiste
$ man comandochenonesiste 2> /dev/null
```

Leggere un file

cat [pathname]

Visualizza il contenuto di un file. Se invocato senza pathname prende lo standard input

Stampare messaggio a video

echo [-n] [-e] [stringa]

Stampa a video la stringa passata come parametro

- [-n] non manda a capo il carrello quando ha finito
- [-e] permette l'uso di caratteri speciali
 - \a bell (campanello)
 - \n new line
 - \t tabulazione
 - \\ backslash
 - \nnn il carattere il cui codice ASCII è nnn

```
saro@saro-VirtualBox:~/Desktop$ echo -e "ciao\nmamma"
ciao
mamma
saro@saro-VirtualBox:~/Desktop$ █
```

Stampare messaggio lungo a video

more [pathname]

Come cat con l'eccezione che se l'output è più lungo di una videata di schermo, ad ogni pagina fa una pausa.

Stampare messaggio lungo a scorrimento a video

less [pathname]

Come more ma permette anche di andare su e giù nell'output

Pipeline

Due comandi possono essere messi in **cascata** collegando l'output del primo con l'input del secondo

comando1 | comando2

Comando2 prende come input l'output di comando1. In realtà i due comandi vengono mandati in esecuzione contemporaneamente: il secondo aspetta che arrivi man mano l'output del primo. Si possono mettere in comunicazione anche più di due comandi:

comando1 | comando2 | ... | comando n

Comandi di conteggio e ordinamento

Conteggio

wc [-c] [-w] [-l] [pathname]

Conteggia i seguenti parametri

- [-c] caratteri
- [-w] parole (separate da spazi)
- [-l] righe

Se non si specifica una opzione particolare, vengono riportati tutti e tre i conteggi.

```
$ echo "Ciao a tutti." | wc -c -w
            3      14
$ wc /etc/passwd /etc/fstab /etc/group
 30    41 1317 /etc/passwd
 13    70  655 /etc/fstab
 55    55  728 /etc/group
 98   166 2700 totale
```

Ordinamento

sort [-n] [-r] [-o file] [-t s] [-k s1[,s2]] [pathname]

- [-n] considera numerica (invece che alfabetica) la chiave di ordinamento
- [-r] ordina in modo decrescente
- [-o file] invia l'output su file anziché sul terminale
- [-t s] usa s come separatore di campo
- [-k s1[,s2]] usa i campi da posizione s1 a s2 per l'ordinamento

Testa e coda

head [-c q] [-n q] [pathname]

Mostra di default le prime 10 righe del suo input

- [-c q] mostra solo i primi q byte dell'input
- [-n q] mostra solo le prime q righe dell'input

tail [-c q] [-n q] [pathname]

Come head, ma a partire dalla fine dell'input

```
$ ls -i | sort -r
script.sh
scheda.pdf
prova.sh
esempio3.txt
esempio2.txt
documento.pdf
conta.sh

$ cat /etc/passwd | sort -t: -k 3,3
root:x:0:0:root:/root:/bin/bash
mario:x:1000:1000:Mario,,,:/home/mario:/bin/bash
test:x:1001:1001:Test User,,,:/home/test:/bin/bash
.....
mail:x:8:mail:/var/mail:/bin/sh
news:x:9:news:/var/spool/news:/bin/sh

$ cat /etc/passwd | sort -t: -k 3,3 -n
root:x:0:0:root:/root:/bin/bash
daemon:x:1:1:daemon:/user/sbin:/bin/sh
bin:x:2:2:bin:/bin:/bin/sh
sys:x:3:3:sys:/dev:/bin/sh
....
test:x:1001:1001:Test User,,,:/home/test:/bin/bash
nobody:x:65534:65534:nobody:/nonexistent:/bin/sh

$ ls /bin /sbin /usr/bin /usr/sbin | sort -o lista_comandi.txt
```

Comandi per i permessi e proprietà

Cambiare i permessi

chmod [-R] mode [pathname]

- [-R] applica i permessi in modo ricorsivo alle sottocartelle
- **mode** nuova maschera di permessi; si usa una stringa del tipo:
 - **target**: a chi cambiare i permessi tra
 - user (u)
 - group (g)
 - other (o)
 - all (a) (default quando si omettono gli altri)

Esempio:

```
rw-r----- → 110 100 000 → 640
rwxr-xr-x → 111 101 101 → 755
rw-r----- ↔ u=rw,g=r,o-rwx
rw-rw-rw- + u+x,o-rw → rwxrw----
rw-r--r-- + a-r,u+r → rw-----
rw-r--r-- + +x → rwxr-xr-x
```

- **grant**: aggiunta o rimozione dei permessi con
 - + (aggiunta)
 - - (sottrazione)
 - = gli lascia alterati
- **permission**: è una sottogringa di rwx ed indica i permessi che si vogliono alterare

Si usa un numero ottale di 3 cifre in cui ogni cifra corrisponde a proprietario, gruppo e altri. I diritti sono di lettura, scrittura ed esecuzione. Ognuno è rappresentato da bit 1 per abilitarlo, bit 0 per disabilitarlo.

Cambiare i proprietari

chown [-R] owner [:group] [pathname]

- [-R] esegue ricorsivamente le modifiche di proprietà
- owner è il nuovo proprietario
- group è il nuovo gruppo proprietario

Il comando

chgrp [-R] group [pathname]

Cambia solo il gruppo di proprietari

Modalità di utilizzo degli utenti

Su un sistema UNIX è bene creare uno o più utenti “normali” per le operazioni quotidiane che non richiedono poteri speciali. In questo modo si limitano i rischi di danneggiare il sistema incautamente e, soprattutto, i danni che si possono causare se l’account viene compromesso da un entità estranea.

Si può usare il comando **adduser** per aggiungerne di nuovi ed il comando **passwd** per cambiare la propria password in qualunque momento.

L’utente amministratore **root** dovrebbe essere usato solo quando strettamente necessario per effettuare operazioni di manutenzione, aggiornamento o configurazione del sistema.

Invece di fare il login come root è possibile diventarlo temporaneamente usando il comando **su (super user)**. Si può diventare un qualunque utente (previa autenticazione) utilizzando **su nome_utente** o si può semplicemente eseguire un solo comando come se fossimo un dato utente con la sintassi **su -c “comando argomenti” nome_utente**.

Super user su Debian e Mac OS X

Su alcuni sistemi Linux (Ubuntu e derivate) e su Mac OS X, l’utente speciale root non viene direttamente utilizzato. Esiste un gruppo di utenti amministratori: ognuno può ottenere i pieni diritti di amministrazione all’occorrenza.

Uno di questi utenti può eseguire con diritti superiori un qualunque comando preponendo il suffisso **sudo**. La sintassi completa diventa quindi **sudo comando argomenti**.

A verifica della propria identità, viene richiesta conferma della password dell’utente stesso (non di root, che potrebbe anche non essere impostata).

Tale verifica verrà riproposta solo dopo un certo lasso di tempo

Comandi di compressione file

Compressione dei file con gzip

gzip [-d] [-c] [pathname]

- [-d] decomprime invece che comprimere (si può usare anche il comando **gunzip**)
- [-c] il file compresso viene mandato allo standard output

```
saro@saro-VirtualBox:~/Desktop$ ls -l
total 0
-rw-rw-r-- 1 saro saro 0 ott 1 08:27 a.txt
-rw-rw-r-- 1 saro saro 0 ott 1 08:27 b.txt
saro@saro-VirtualBox:~/Desktop$ chmod u+x a.txt
saro@saro-VirtualBox:~/Desktop$ chmod g+x,o-r b.txt
saro@saro-VirtualBox:~/Desktop$ ls -l
total 0
-rwxrwxr-- 1 saro saro 0 ott 1 08:27 a.txt
-rw-rwx-- 1 saro saro 0 ott 1 08:27 b.txt
```

```
saro@saro-VirtualBox:~/Desktop$ echo "ciao" > testo.txt
saro@saro-VirtualBox:~/Desktop$ ls -l
total 4
-rw-rw-r-- 1 saro saro 5 ott 1 11:17 testo.txt
saro@saro-VirtualBox:~/Desktop$ chmod 000 testo.txt
saro@saro-VirtualBox:~/Desktop$ ls -l
total 4
----- 1 saro saro 5 ott 1 11:17 testo.txt
```

Tale comando comprime, con appositi algoritmi, il contenuto dei file per occupare meno spazio. I file vengono compressi da un tipo di file con estensione **.gz**.

Per vedere il contenuto di un file compresso invece di usare il comando **cat** possiamo usare il comando **zcat**; esistono anche le varianti **zmore** e **zless**.

Esiste un'altra coppia di comandi che utilizzano algoritmi di compressione più evoluti che permettono di ottenere compressioni migliori (a scapito di una maggiore elaborazione e richiesta di memoria): **bzip2** e **bunzip2**. La sintassi è la stessa di gzip, cambia l'estensione che è **.bz2** e i comandi specifici ce diventano **bzcat**, **bzmore**, **bzless**.

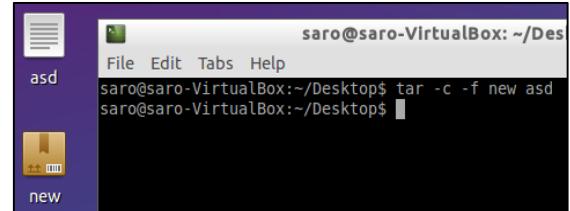
```
$ ls -l config.*  
-rw-r--r-- 1 mario mario 2389 2005-09-23 12:41 config.txt  
$ gzip config.txt  
$ ls -l config.*  
-rw-r--r-- 1 mario mario 813 2005-09-23 12:41 config.txt.gz  
$ gunzip config.txt.gz  
$ ls -l config.*  
-rw-r--r-- 1 mario mario 2389 2005-09-23 12:41 config.txt  
  
$ cat config.txt | bzip2 -c > config.txt.bz2  
$ ls -l config.*  
-rw-r--r-- 1 mario mario 2389 2005-09-23 12:41 config.txt  
-rw-r--r-- 1 mario mario 722 2005-09-23 12:44 config.txt.bz2  
  
$ bzcat config.txt.bz2 > config.txt  
  
$ ls -R /etc/ | gzip | bzip2 | bzcat | zmore  
  
$ cat /dev/cdrom | bzip2 > mio_cdrom.iso.bz2
```

Aggregazione dei file con tar

A differenza dei programmi di compressione ZIP e RAR che hanno come scopo mettere in un archivio compresso più file se non interi alberi del file system, i comandi UNIX di tipo gzip sono **comandi di flusso** che invece hanno lo scopo di comprimere un flusso di dati che nel caso base può semplicemente essere un file. Se vogliamo archiviare (o aggregare) più file o un intero ramo del file system sotto UNIX si usa il comando **tar**.

```
tar [-c|-x|-t] [-z|-j|-J] [-v] [-f archive] [pathname]
```

- [-c] crea un archivio
- [-x] estrae un archivio
- [-t] elenca il contenuto dell'archivio
- [-z | -j | -J] comprime l'archivio con gzip, bzip2 e xz
- [-v] mostra il progresso (modalità verbose)
- [-f archive] specifica l'archivio



L'estensione dei file creati è **.tar**. Inoltre permette di aggregare file o cartelle in un archivio già creato (con estensione .tar). Se il pathname è una cartella allora viene aggiunto anche il contenuto.

Il comando tar accetta anche una sintassi più stringata per le opzioni: si possono omettere i trattini davanti alle opzioni e si possono, ovviamente, raggruppare.

La compressione degli archivi si ottiene facendo “passare” gli archivi .tar attraverso un filtro di compressione (gzip, bzip2, o xz) in modo esplicito (usando una pipe) oppure in modo implicito (utilizzando le opzioni -z,-j o -J).

```
$ tar -c -v -f prova.tar *.tex *.pdf images/  
intro.tex  
shell.tex  
slides.tex  
slides.pdf  
images/  
images/dmi.jpg  
images/unict.gif  
.....  
  
$ tar x -f prova.tar  
  
$ tar cjf prova2.tar.bz2 examples/ images/ *.pdf  
  
$ bzcat prova2.tar.bz2 | tar tv  
drwxr-xr-x mario/mario 0 2005-09-20 12:28:57 examples/  
drwxr-xr-x mario/mario 0 2005-09-23 14:42:28 examples/process/  
-rw-r--r-- mario/mario 21 2005-09-20 12:29:27 examples/process/esempio.txt  
drwxr-xr-x mario/mario 0 2005-09-20 17:40:42 examples/process/esempi/  
.....  
  
$ tar c examples/ images/ *.pdf | gzip > prova3.tar.gz
```

Le estensioni per gli archivi compressi diventano: **.tar.gz**, **.tar.bz2** e **.tar.xz**.

I file vengono inseriti nell'archivio con il path relativo alla directory corrente.

Se non viene specificato il nome dell'archivio si lavora con lo standard input e lo standard output

Sono disponibili tante altre opzioni che permettono anche la modifica e l'aggiornamento degli archivi.

Alias per i comandi

La shell dà la possibilità di definire dei nomi propri (variabili) in modo tale che corrispondano a sequenze arbitrarie di comandi e opzioni

```
alias [name[=value]]
```

```
saro@saro-VirtualBox:~/Desktop$ alias lista=ls
saro@saro-VirtualBox:~/Desktop$ lista
asd new
saro@saro-VirtualBox:~/Desktop$
```

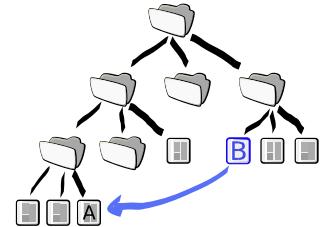
Invocare alias senza parametri visualizza la lista degli alias correntemente attivi. Una invocazione del tipo **alias name** visualizza l'associazione attuale (se ne esiste già una). Per rimuovere un alias si utilizza il comando **unalias name**.

Link del file system

I file system UNIX mettono a disposizione un meccanismo molto potente che permette di riferirsi ad un dato oggetto nel filesystem con più di un riferimento (o **link**). Dato un file **A** nel file system, è possibile collocare in un punto qualunque del file system un link **B** che “punti” ad A.

A livello di applicazione lavorare con A o lavorare con B hanno gli stessi effetti: in effetti operiamo sul medesimo file. Si hanno due nomi per indicare lo stesso oggetto.

Esistono due tipi di link: simbolici e fisici



Link simbolici (soft link)

I link simbolici (detti anche soft link) sono dei file speciali che contengono al loro interno il riferimento al file a cui puntano. In effetti sono dei piccoli file di testo contenente il path assoluto o relativo (alla directory in cui si trova il link simbolico) per raggiungere il file a cui si riferiscono (il file target).

Una invocazione del comando **ls -l** con tutti i dettagli sui file rivela la natura del link simbolico: un flag **l** vicino ai diritti indica che si tratta di un link simbolico; vicino al nome del link simbolico viene indicato il nome del file target.

```
$ ls -l file_?
-rw-r--r-- 1 mario mario 2389 2005-09-23 18:50 file_A
1rwxrwxrwx 1 mario mario     6 2005-09-23 18:48 file_B -> file_A
```

Se viene rimosso il soft link, il file target rimane inalterato. Se si cancella il file target, il soft link diventa **inconsistente**.

Link fisici (hard link)

I link fisici (detti anche hard link) hanno una natura diversa ed agiscono ad un livello del file system più basso.

I file system UNIX rappresentano ogni oggetto (file o cartella) con un **inode** memorizzato in una parte del disco. Se una cartella contiene quel file, avrà un riferimento all'indirizzo dell'inode che memorizza il file (come se fosse un puntatore ad un oggetto).

Un hard link agisce creando, nella cartella che lo contiene, un secondo puntatore al medesimo inode. Ci saranno due puntatori allo stesso inode.

Dato un file e creato un hard link ad esso, a posteriori è impossibile distinguere qual era il nome originale e quale il link quindi si ha una trasparenza totale a livello di applicazione.

Per gestire gli hard link, il filesystem UNIX mantiene per ogni inode un contatore del numero di link fisici che puntano ad esso. Per sapere quanti hard link puntano allo stesso file si può usare il comando **ls**:

```
$ ls -l file_?
-rw-r--r-- 2 mario mario 2389 2005-09-23 18:50 file_A
-rw-r--r-- 2 mario mario 2389 2005-09-23 18:50 file_B
```

Il numero degli hard link è riportato subito dopo i permessi di accesso.

Poiché gli hard link lavorano a livello di inode, si possono creare solo tra file del medesimo filesystem. Con i link simbolici è invece possibile creare link a file che si trovano in filesystem diversi.

Per implementare gli hard link è necessario che il filesystem lo supporti: tutti i filesystem UNIX (ext2, ext3, reiserfs, nfs, ...) lo supportano, quelli di Windows (VFAT,NTFS) no.

Quando si cancella un hard link viene innescato il medesimo meccanismo utilizzato quando si cancella un semplice file:

1. Il SO controlla il contatore dei link all'interno dell'inode del file che si vuole cancellare;
2. Se tale contatore è uguale ad 1, allora viene cancellato sia il link (nella directory) che l'inode stesso (compreso il contenuto del file);
3. Se il contatore è strettamente maggiore di 1, allora esso viene decrementato e viene cancellato solo il riferimento all'inode e quest'ultimo viene preservato.

Si possono creare link simbolici di directory ma non è possibile fare altrettanto con i link fisici. Altrimenti si creerebbero dei **loop** nella gerarchia del filesystem che creerebbero problemi con i programmi che lo esplorano. Ciò non è un problema con i soft link poiché sono distinguibili dalle directory vere.

Creare link tra i file e le directory del file system

In [-s] target [linkpathname]

- [-s] crea un link simbolico invece che uno fisico
- target è il file o directory a cui si farà riferimento
- linkpathname è il pathname del link

Modalità di esecuzione

Esistono varie modalità di esecuzione dei comandi sotto una shell:

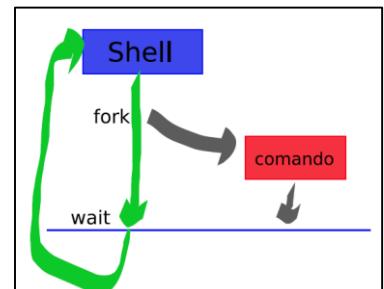
Simple command execution

La più semplice consiste nell'invocazione di un singolo comando (con relativi parametri).

Ogni comando restituisce un **exit status** che rappresenta l'esito della computazione del comando stesso. Mediante l'exit status è possibile controllare la buona riuscita di un comando. L'exit status è un intero:

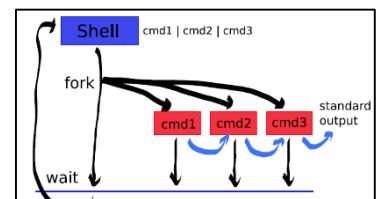
- **0**: esecuzione riuscita con successo;
- **n > 0**: esecuzione fallita;

L'esecuzione di un comando da shell, la sospende temporaneamente fino alla terminazione del comando stesso.



Pipeline

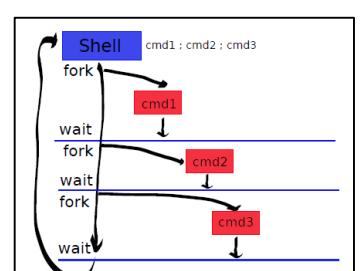
Cascata di processi in cui ognuno riceve l'output del precedente come input. L'output della pipeline corrisponde all'output dell'ultimo processo.



List of command

La lista di comandi permette di eseguire una **sequenza di comandi** come se fossero un solo comando.

La sintassi per l'invocazione è: **comando1 ; comando2...**



L'exit status della sequenza è uguale a quello dell'ultimo comando. Non c'è alcuno scambio di input/output tra i processi.

Nelle sequenze di comandi, oltre al separatore; è possibile utilizzare anche gli operatori condizionali **&&** e **||**. La semantica degli operatori è la seguente:

- **c1 && c2** (and list): c2 viene eseguito se e solo se l'exit status di c1 è 0
- **c1 || c2** (or list): c2 viene eseguito se e solo se l'exit status di c1 è diverso da 0

```
$ cat fileinesistente && echo fatto
cat: fileinesistente: No such file or directory
$ cat fileinesistente || echo errore
cat: fileinesistente: No such file or directory
errore
$ false && comando_inesistente_che_non_verra_mai_eseguito
```

L'exit status di una lista condizionata è uguale all'exit status dell'ultimo comando eseguito (che non è necessariamente l'ultimo comando della lista).

Asynchronous execution

Utilizzando la sintassi **comando &** (notare la e-commerciale alla fine), il comando viene mandato in **esecuzione asincrona** (o più semplicemente in **background**).

In questo caso la shell manda in esecuzione il processo figlio per eseguire il comando e continua la sua attività (non attende la fine del task).

Di default, i processi in esecuzione asincrona prendono il loro input da **/dev/null** (se non specificato).

Un processo in foreground può essere mandato in background interattivamente utilizzando la combinazione di tasti **CTRL-Z**. In effetti il processo viene anche sospeso (non continua la sua esecuzione), viene messo in pausa.

Controllo dei jobs

La shell associa ad ogni comando semplice (o pipeline) uno **job**. Ogni job ha un numero che lo identifica univocamente in un dato istante.

Quando un comando viene eseguito in modo asincrono, la shell dà un output del tipo:

[jobnumber] PID

Dove **jobnumber** è il numero dello job eseguito in background e **PID** è l'identificativo di processo che porta effettivamente avanti il job.

Comandi disponibili:

- **jobs**: visualizza i jobs correntemente attivi ed il loro stato;
- **bg**: manda in background l'esecuzione di uno job;
- **fg**: porta in foreground l'esecuzione di uno job;
- **kill**: invia un segnale ad uno job.

A proposito di kill: con **kill -# PID** si invia il segnale numero # al processo con identificativo PID.

Tra quelli standard: 3 QUIT e 9 KILL.

Variabili

Una variabile è un oggetto che memorizza un valore. Le variabili sono identificate da stringhe alfanumeriche che iniziano con un carattere. L'**assegnamento** di un valore ad una variabile viene effettuata mediante l'operatore di assegnamento **=**. La sintassi esatta è:

nome _variabile=valore

dove prima e dopo del carattere = NON ci deve essere nessuno spazio.

Il valore nullo è un valido assegnamento per una variabile di shell. Per annullare un assegnamento si usa il comando **unset**. Con il comando **set** invece è possibile visualizzare tutte le variabili correntemente definite in una shell (comprese le **variabili d'ambiente**).

Espansione delle variabili

Quando la shell esegue un comando effettua delle espansioni della riga di comando, tra queste abbiamo le **espansioni per i metacaratteri** (che abbiamo già visto) e le **espansioni delle variabili**.

Mettendo il carattere **\$** davanti al nome di una variabile definita, questa viene espansa con il suo valore.

Quindi la sintassi è: **\$nome variabile**, che rappresenta una semplificazione della sintassi generale **\${nome variabile}** che risulta necessaria in alcuni casi.

Variabili speciali e di shell

La shell mette a disposizione alcune variabili speciali:

- **\$?**: exit status del comando più recente eseguito in foreground;
- **\$\$**: PID della shell corrente;
- **\$!**: PID del più recente job eseguito in background.

Così come alcune variabili di shell predefinite:

- **\$USER**: utente corrente;
- **\$HOME**: la home directory associata con l'utente;
- **\$PATH**: la lista delle directory che deve essere ispezionata per trovare i comandi eseguibili;
- **\$PS1**: la stringa corrispondente al prompt corrente.

Comand substitution

Mediante la command substitution è possibile sostituire l'output di un comando con il comando stesso (parlando in termini di espansione della riga di comando).

```
$ ls | wc -w
15

$ PAROLE='ls | wc -w'
$ echo parole conteggiate: $PAROLE
parole conteggiate: 15
```

La sintassi è: **'comando'** (con le apici inverse ottenibili utilizzando il tasto ALT GR e l'apice normale').

Risulta molto utile per inizializzare le variabili.

Quoting

Il meccanismo del **quoting** serve ad eliminare il metasignificato ad alcuni dei caratteri che vengono usati per altri scopi. Il quoting può essere di tre tipi:

- ****(backslash): elimina il metasignificato dal carattere seguente;
- **''** (single quote): elimina il metasignificato da tutti i caratteri contenuti al suo interno;
- **" "** (double quote): elimina il metasignificato da tutti i caratteri contenuti al suo interno ad eccezione di \$, ' e \.

```
$ echo $PATH
/home/mario/bin:/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games
$ echo \$PATH
$PATH

$ echo "Il mio nome è $USER."
Il mio nome è pippo.

$ echo 'Il mio nome è $USER.'
Il mio nome è $USER.

$ echo "Il mio nome è 'whoami' e ho 5$."
Il mio nome è pippo e ho 5$.
```

Comandi di ricerca di un file

Ricerca di un file

find [pathname] [expression]

- pathname è il percorso in cui cerca ricorsivamente i file da esaminare
- expression specifica le regole con cui selezionare il file
 - **opzione**: modifica il comportamento della ricerca (es. `-mount`)
 - **condizione**: condizioni da verificare (es. `-not expr`, `expr1 -or expr2,...`)
 - **azione**: specifica cosa fare quando trova un file (es. `-exec comando {} -print,...`)

Man mano che i file vengono trovati i nomi vengono stampati a video o altro se specificato. Le opzioni sono tante, per dettagli consultare **man find**.

```

$ find . -name '*.sh' -print
./script.sh
./prova.sh
./conta.sh
./cartella/esercizio/eseguimi.sh

$ find /home/mario -name '*.bak' -exec rm {} \;

$ find /etc/ -type d -print
/etc/
/etc/mkinitrd
/etc/network/if-post-down.d
/etc/network/if-pre-up.d
/etc/network/if-up.d
/etc/network/if-down.d
/etc/network/run
/etc/default
/etc/skel
.....

```

Selezionare file

grep [-i] [-l] [-n] [-v] [-w] pattern [filename]

Cerca all'interno dei file specificati con filename le righe che contengono il pattern specificato, indicato da una stringa a da una **regular expression**.

- **[-i]** ignora le differenze tra minuscole e maiuscole
- **[-l]** fornisce la lista dei file che contengono il pattern
- **[-n]** le linee dell'output sono precedute dal numero di linea
- **[-v]** stampa solo le linee che NON contengono il pattern
- **[-w]** vengono restituite solo le linee che contengono il pattern/stringa come parola completa

Esistono due varianti di grep:

- **fgrep** (Fixed General Regular Expression Parser)
- **egrep** (Extended General Regular Expression Parser=

Volendo si possono ottenere i medesimi risultati con le opzioni **-F** e **-G** di grep.

Attraverso le espressioni regolari è possibile specificare dei pattern più complessi della semplice stringa contenuta.

metacarattere	tipo	significato
^	basic	inizio della linea
\$	basic	fine della linea
.	basic	un singolo carattere (qualsiasi)
[str]	basic	un qualunque carattere in str
[^str]	basic	un qualunque carattere non in str
[a-z]	basic	un qualunque carattere tra a e z
\	basic	inibisce l'interpretazione del carattere successivo
*	basic	zero o più ripetizioni dell'elemento precedente
+	ext	una o più ripetizioni dell'elemento precedente
?	ext	zero o una ripetizione dell'elemento precedente

```

saro@saro-VirtualBox:~/Desktop$ ls
asd ciaociao new
saro@saro-VirtualBox:~/Desktop$ ls | grep ^n
new
saro@saro-VirtualBox:~/Desktop$ ls | grep d$ 
asd
saro@saro-VirtualBox:~/Desktop$ ls | grep c.
ciaociao
saro@saro-VirtualBox:~/Desktop$ ls | grep e
new
saro@saro-VirtualBox:~/Desktop$ ls | grep ciao*
ciaociao

```

- **fgrep rossi /etc/passwd**
fornisce in output le linee del file **/etc/passwd** che contengono la stringa fissata **rossi**
- **egrep -nv '[agt]+' relazione.txt**
fornisce in output le linee del file **relazione.txt** che non contengono stringhe composte dai caratteri **a, g, t** (ogni linea è preceduta dal suo numero)
- **grep -w print *.c**
fornisce in output le linee di tutti i file con estensione **c** che contengono la parola intera **print**
- **ls -al . | grep '^d.....w.'**
fornisce in output le sottodirectory della directory corrente che sono modificabili da tutti gli utenti del sistema
- **egrep '[a-c]+z' doc.txt**
fornisce in output le linee del file **doc.txt** che contengono una stringa che ha un prefisso di lunghezza non nulla, costituito solo da lettere **a, b, c**, seguito da una **z**

Programmazione in ambiente UNIX

Segnali di UNIX

I segnali sono un semplice mezzo per notificare ai processi degli eventi asincroni. Si possono vedere come degli interrupt software. A differenza dei messaggi delle code FIFO:

- Un segnale può essere inviato in qualsiasi momento, occasionalmente da un processo, ma spesso dal kernel (ad esempio, per una istruzione illegale);
- Un segnale non viene necessariamente ricevuto e processato; implicitamente la maggior parte dei segnali provoca la terminazione del processo. Eventualmente un processo può decidere di ignorarli o gestirli;
- I segnali non hanno alcun contenuto informativo; in particolare, non si può conoscere il mittente del segnale (potrebbe essere il kernel o un altro processo).

Lista dei segnali principali

Di seguito riportiamo una lista dei segnali principali con relative costanti:

- **SIGHUP (1): hangup.** Un processo riceve questo segnale quando il terminale a cui era associato viene chiuso o scollegato (la finestra viene chiusa o, nel caso di un collegamento remoto, la connessione cade). Spesso molti processi server rileggono il loro file di configurazione alla ricezione di questo segnale;
- **SIGINT (2): Interrupt.** Viene ricevuto da un processo quando l'utente preme la combinazione di tasti di interrupt (solitamente CTRL+C);
- **SIGQUIT (3): Quit.** Simile a SIGINT ad eccezione del fatto che il processo genera un core dump, ovvero un file in cui viene messa l'immagine della memoria del processo al momento in cui si riceve il segnale SIGQUIT. Questa immagine puo' essere utile ai fini del debugging. Solitamente questo segnale viene invocato dalla combinazione di tasti CTRL+;
- **SIGILL (4): Illegal Instruction.** Il processo ha tentato di eseguire una istruzione proibita o inesistente;
- **SIGKILL (9):** questo segnale non può essere intercettato dal processo, che non può fare altro che terminare. È il modo più certo e brutale per "uccidere" un processo;
- **SIGSEGV (11): Segmentation Violation.** Generato quando il processo tenta di accedere ad un indirizzo che ricade fuori dalle aree di memoria allocate;
- **SIGALRM (14): Alarm.** Un segnale che il processo si può auto-inviare alla scadenza di un certo intervallo di tempo;
- **SIGUSR1, SIGUSR2: User defined.** Non hanno un significato preciso e possono essere utilizzati dai processi per implementare un rudimentale protocollo di comunicazione;
- **SIGCHLD (17): Child Death.** Inviato ad un processo quando uno dei suoi figli termina.

Per ulteriori dettagli: **man 7 signal.** Tutte queste costanti sono definite nell'header **signal.h**.

Gestione dei segnali

I processi possono decidere di ignorare o di *armare* (collegare una propria procedura che verrà invocata ogni volta lo si riceve) alcuni segnali. Fanno eccezione i segnali di errore fatale (4,11) o il segnale di kill (9) che non possono in alcun modo essere gestiti e che portano alla terminazione.

Inviare un segnale

Per inviare un segnale ad un altro processo si può utilizzare la chiamata di sistema:

int kill(pid_t pid, int sig)

Dove **pid** è il PID del processo a cui vogliamo inviare il segnale di numero **sig**. La chiamata **kill()** ritorna -1 in caso di errore, 0 in caso di successo.

Un processo può inviare segnali solo ad altri processi che appartengono allo stesso proprietario. Ad esempio, non possiamo uccidere i processi di un altro utente. A tutto questo fa eccezione l'amministratore root, lui può tutto.

Da approfondire: Programmazione in ambientazione Unix, Linguaggio C