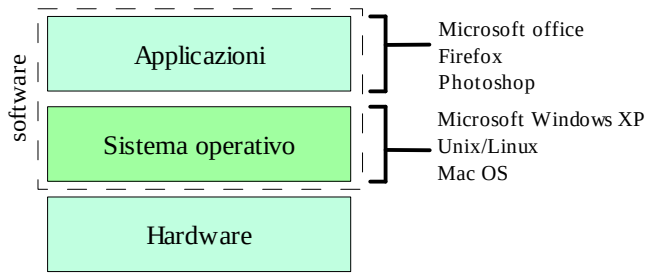
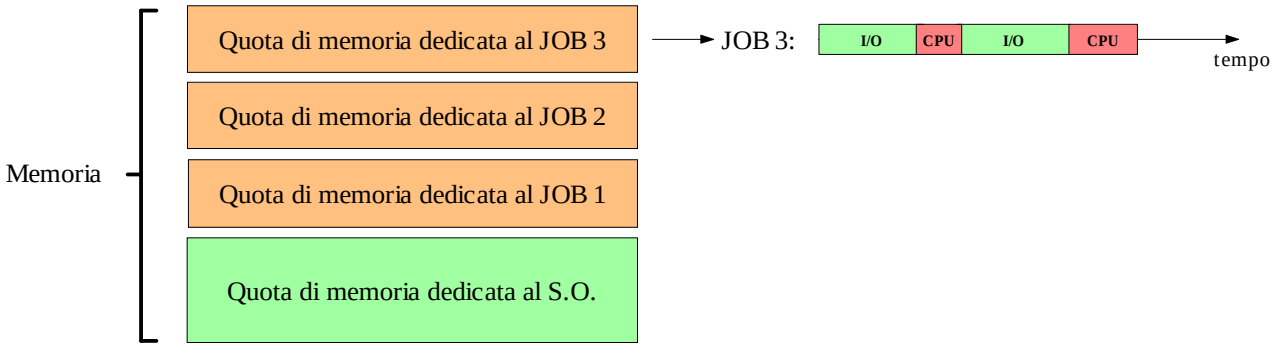
Cos'è un sistema operativo?
Avete mai provato a dare una definizione esauriente alla parola sistema operativo? La definizione che spesso si sente in giro è sempre abbastanza generica, ed inoltre, a seconda di chi vi parla, si sentirà parlare di sistema operativo come gestore delle risorse oppure di sistema operativo come macchina estesa. Evidentemente la definizione dipende dai punti di vista di chi osserva il sistema operativo. Non vi è dubbio, quindi, che un sistema operativo assolve a più di una funzione, pertanto, nel corso di queste note saranno affrontati i principi e le problematiche che caretterizzano il funzionamento, le prestazioni e le funzionalità offerte dai moderni sistemi operativi.
E' ormai risaputo che l'architettura della maggior parte dei calcolatori a livello hardware è di difficile programmazione. Alla logica digitale usata dall'hardware si preferisce così il linguaggio macchina che pur non essendo vicino al linguaggio parlato è almeno di facile manipolazione. Al livello del linguaggio macchina si preferisce, poi, aggiungere un ulteriore strato software che consegna all'entità superiore una ulteriore estensione. E' proprio questa estensione che consegna nelle mani del programmatore una realtà molto più semplice, fatta di operazioni astratte per l'interazione con i dispositivi di I/O (periferiche) e per l'esecuzione contemporanea di più programmi. Questa ulteriore astrazione (al di sopra del livello del linguaggio macchina) è appunto il sistema operativo che quindi ha l'importante compito di estendere l'hardware sottostante altrimenti incontrollabile. Il sistema operativo offre ai programmatori numerosi servizi e l'accesso a tali servizi avviene attraverso istruzioni speciali, le cosiddette system calls. Un altra estensione operata dal sistema operativo è l'implementazione di una visione astratta della memoria secondaria nota come file system. Tale astrazione ha come obiettivo la gestione del disco rigido o di un supporto dati mediante operazioni di lettura e scrittura, in questo modo viene data all'utente la possibilità di archiviare file di dati e programmi. Riassumendo, il sistema operativo è il livello software che interagisce direttamente con l'hardware e che ne estende le potenzialità rendendo possibile l'esecuzione di applicazioni, semplificando l'accesso alle periferiche e offrendo all'utente la possibilità di archiviare dati.



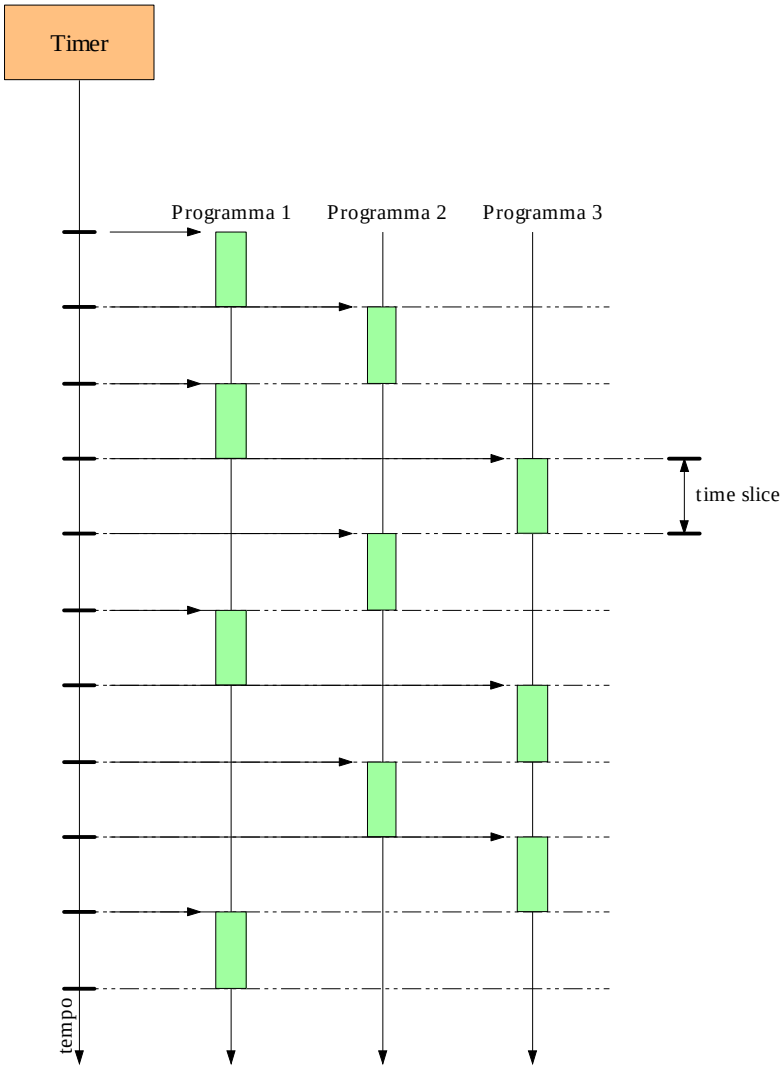
La definizione di sistema operativo come macchina estesa equivale ad una descrizione dall'alto verso il basso del sistema operativo. Procedendo, quindi, ad una descrizione di sistema operativo dal basso verso l'alto si arriva a descrivere un'altra importante funzionalità che vede il sistema operativo come gestore delle risorse. Un calcolatore dispone di molte risorse: tipicamante uno o più processori, un certo quantitativo di memoria principale, memoria di massa, lettori cd-rom e dvd, mouse, tastiera ed altre svariate periferiche. Quanto elencato fa quindi parte dell'hardware di sistema (risorse generalmente condivise tra processi è questo il problema!), il compito del sistema operativo è allora anche quello di essere gestore delle risorse. Infatti, uno o più programmi possono richiederne l'uso ma solo il programma che giunge ad un accordo con il sistema operativo ne riceve i benefici. Quando un calcolatore viene usato da più utenti, la necessità di gestire e proteggere la memoria, i dispositivi di ingresso e di uscita e le altre risorse di sistema è ancora più evidente. Immaginate cosa possa accadere se un programma utente tentasse di dare una sbirciata in memoria ad un altro programma utente! Ed ancora, supponiamo che due programmi accedono contemporaneamente alla stampante locale: una riga stampata potrebbe ad esempio appartenere al primo programma, la successiva al secondo programma (oppure viceversa) e così via... La gestione delle risorse è per questo motivo una altra funzionalità che il sistema operativo offre, sia per allocare le risorse ai vari programmi in esecuzione e sia per proteggere l'hardware da un uso improprio. In base a quanto osservato, un'altra funzionalità offerta da un sistema operativo è quindi

quella di controllore, garantire cioè che i programmi in esecuzione non accedano ad informazioni riservate ad altri programmi in esecuzione o del sistema operativo, compromettendo la riservatezza dei dati oppure il buon funzionamento del sistema operativo.
Per conseguire una buona efficienza il sistema operativo dispone di diverse politiche di gestione dei programmi in esecuzione note come tecniche di schedulazione. Le risorse possono essere condivise rispetto al tempo se i programmi utenti fanno a turno per usarle, oppure possono essere condivise rispetto allo spazio se la memoria è sufficientemente grande da poter ospitare contemporaneamente più processi in esecuzione. La sola rilevazione di un hardware usato male oppure malfunzionante non è sintomo di robustezza per un sistema operativo che deve quindi sviluppare importanti tecniche di protezione dell'hardware. Ad esempio, se il disco rigido ha un settore difettoso il sistema operativo può ricopiare le informazioni residenti su quel settore in un altra parte del disco.
Riassumendo, il sistema operativo è il livello software che interagisce direttamente con l'hardware e ne effettua la gestione, alloca le risorse di sistema ai programmi in esecuzione, previene il malfunzionamento delle risorse ed offre protezione ai programmi in esecuzione vietandone le interazioni dannose.
Nel corso degli anni si sono susseguite diverse generazioni di sistemi operativi. La prima generazione è riconducibile all'uso di valvole e schede a spinotti. L'hardware occupava molto spazio, anche una intera stanza. La programmazione avveniva usando il linguaggio macchina. Un gruppo di personee specializzate disponeva su delle apposite schede particolari spinotti che servivano a controllare le funzioni logiche della macchina.
La scoperta del transistor ed il suo rapido impiego ci conduce alla seconda generazione. I calcolatori diventano adesso affidabili e meno ingombranti al punto tale da poterne iniziare la vendita (in realtà il costo di un calcolatore era talmente alto al punto che solo le grandi aziende di stato e le università potevano sopportarne le spese). Nella seconda generazione si collocano, invece, i sistemi batch. Il termine batch (dall'inglese infornata) viene usato per indicare un particolare modo di gestire le operazioni di un sistema. In altre parole significa che tutta la sequenza di operazioni e dati necessari per svolgere un particolare compito vengono preparati in anticipo e memorizzati su un adeguato supporto (solitamente un blocco di schede perforate). Una procedura (l'antenato del sistema operativo) viene poi eseguita dal sistema che le svolge in un unico blocco, cioè senza che sia necessario, o possibile, un intervento umano prima che essa sia terminata. Il termine è quindi contrapposto alla modalità interattiva, che si ha quando l'utilizzatore accede direttamente ad un terminale o ad una interfaccia dell'elaboratore e segue passo passo le varie fasi dell'operazione.
L'integrazione del transistor all'interno di circuiti integrati riduce ancor di più le dimensioni dei calcolatori, siamo adesso alla terza generazione. Nella precedente generazione è emersa la necessità di tenere sempre impegnata la costosissima CPU. Per questo motivo i calcolatori della terza generazione hanno preferito la multiprogrammazione allo spooling. La multiprogrammazione ha effettivamente aumentato le prestazioni dei sistemi operativi sfruttando meglio la CPU, il concetto è il seguente: mentre un programma deve attendere la fine di un operazione di I/O un altro programma può essere eseguito in parallelo. A tale vantaggio si aggiunge tuttavia il problema di far convivere diversi programmi residenti in memoria. Un programma utente è rappresentabile graficamente da sequenze di operazioni di I/O alternate a periodi di CPU burst.



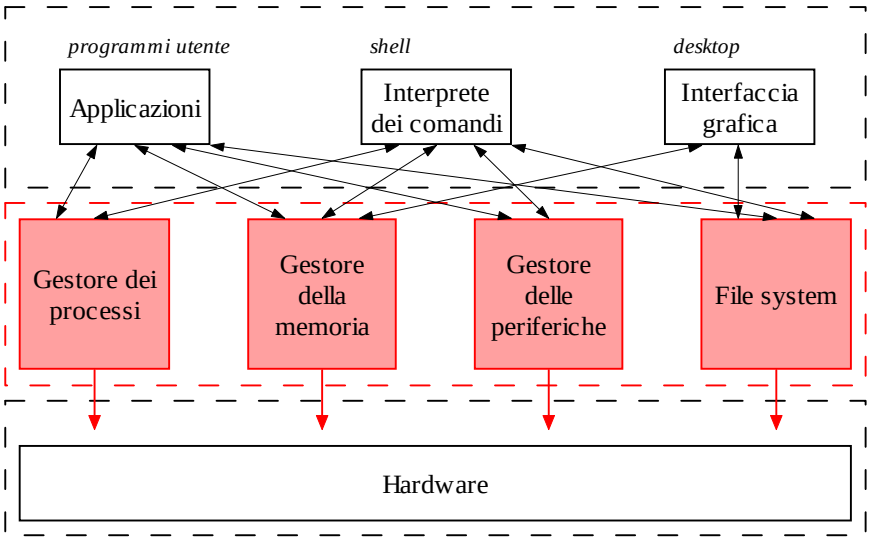
Ad ogni passaggio della CPU, da un programma in esecuzione ad un altro, il sistema operativo deve

salvare lo stato del programma (stack pointer, program counter e status register) se vuole in seguito poterne consentire la ripresa ricaricando i valori salvati.
I sistemi operativi con il passare del tempo hanno preferito la tecnica del time sharing alla multiprogrammazione. Nei sistemi a time sharing la CPU viene assegnata a turno ai vari programmi in esecuzione. Un programma usa la CPU solo nel quanto di tempo assegnatogli, lo slot temporale (talvolta detto time slice) è uguale per tutti i programmi in esecuzione. Un dispositivo hardware, il timer o clock di sistema, invia dei segnali di interrupt alla CPU che quindi si libera del programma in esecuzione (salvando i registri della CPU come detto prima) e garantisce che ciascun programma non occupi mai la CPU più a lungo di un time slice. Il segnale scandito dal clock viene interpretato dal programma in esecuzione come una sollecitazione forzata al rilascio della risorsa CPU, può anche capitare che il programma in esecuzione termini i propri calcoli prima del time slice.



Struttura e organizzazione di un S.O.
Esistono diversi modi per classificare un sistema operativo, tuttavia la classificazione più naturale è quella che classifica il sistema operativo in base ai suoi processi ed ai loro compiti (in seguito verranno date ulteriori nozioni per adesso è sufficiente sapere che un processo è un programma in esecuzione). Ai vari processi vanno assegnate le risorse di sistema e ciò va fatto in maniera tale da non compromettere la normale esecuzione dei successivi processi. Per questo motivo l'assegnazione delle risorse di sistema deve rispettare delle regole ben precise, sarà compito del gestore dei processi allocare le risorse di sistema utili ai processi. La memoria centrale del sistema è anch'essa una risorsa condivisa tra i vari processi e va pertanto partizionata (in seguito vedremo che il concetto di memoria virtuale darà al processo l'illusione di disporre di più memoria). Il gestore della memoria si occupa, appunto, dell'assegnazione della memoria. Analizzando, quindi, tutte le

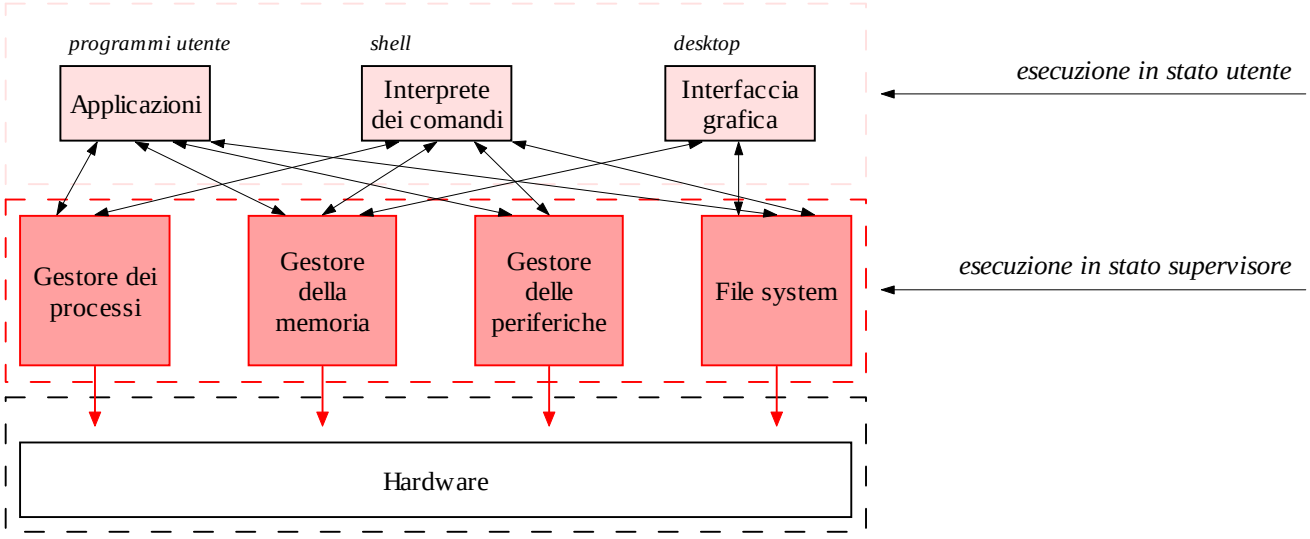
risorse di sistema scopriamo che per ognuna di esse è necessario un gestore che si faccia carico della loro assegnazione. Ciò avviene anche per le periferiche di sistema, il gestore delle periferiche si occupa di risolvere i problemi di accesso alle periferiche nonchè le eventuali contese di accesso verso le stesse. Una speciale periferica è la parte del sistema operativo che si occupa della gestione del file system il quale fornisce una struttura dati per file e programmi e si occupa così della gestione della memoria di massa.



L'interprete dei comandi e l'interfaccia grafica non fanno parte del sistema operativo anche se molto spesso vengono fornite assieme ad esso. Nel corso degli anni sono state adottate diverse strutture organizzative per sistemi operativi, tra queste le più importanti sono: organizzazione monolitica, organizzazione a livelli, organizzazione a macchine virtuali, organizzazione exokernel ed organizzazione client/server.

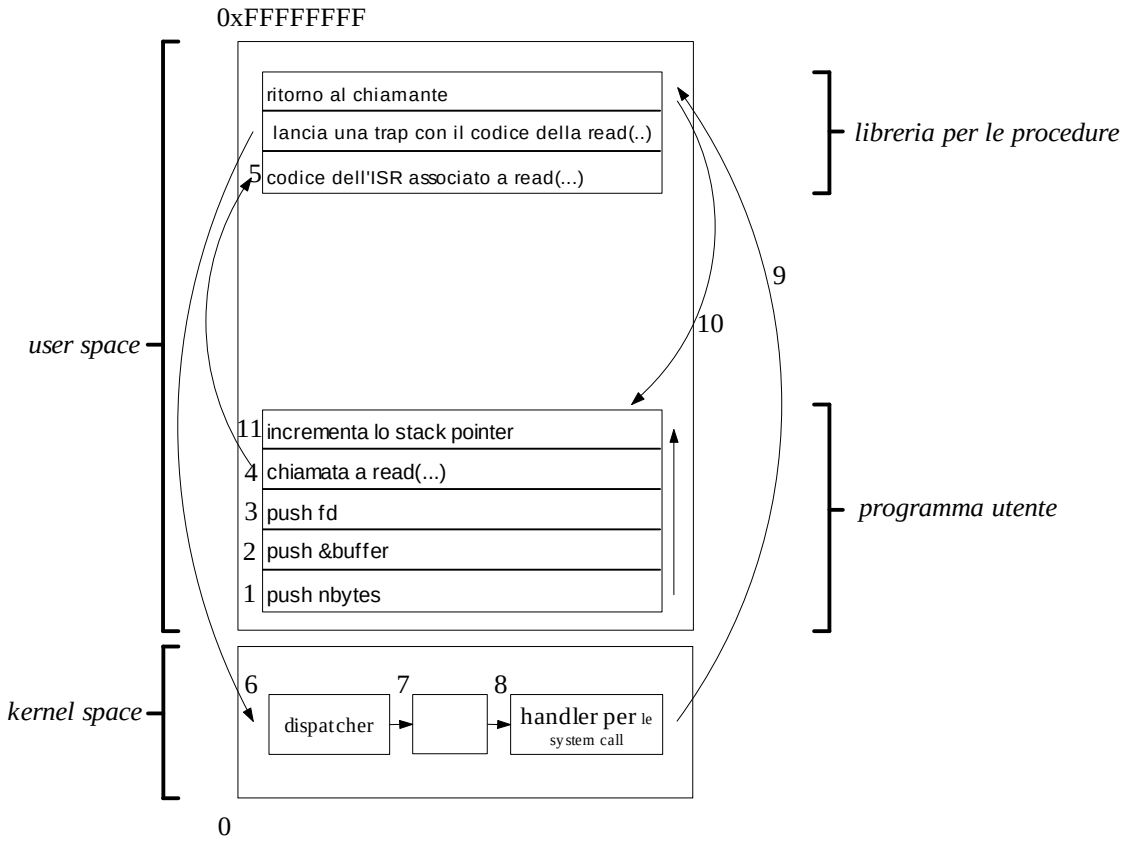
Organizzazione monolitica
L'organizzazione monolitica è quella usata dalla maggior parte dei sistemi operativi. La struttura consiste nell'assenza di una struttura! Il sistema operativo è scritto come un insieme di funzioni e procedure capaci di dialogare e scambiare dati. In questo modo ogni procedura può chiamare un altra procedura se ne ha bisogno. Quando si adotta questa struttura per realizzare il sistema operativo si compilano dapprima tutte le singole procedure e in un secondo momento si procede a collegarle tra di loro per formare un unico programma. E' questa la tipica struttura adottata nei sistemi Unix/Linux e Windows. Tuttavia, anche se è assente una vera struttura è possibile intravedere una netta suddivisione nei servizi offerti dal sistema operativo. Anzitutto, il sistema operativo, ammette due modalità di funzionamento: la prima è la modalità di funzionamento in stato utente, la seconda è la modalità di funzionamento in stato supervisore. Tale distinzione consente la protezione dell'hardware, infatti, la modalità supervisore agisce direttamente sull'hardware mentre alla modalità utente viena data la possibilità di agire solo su un sottoinsieme delle risorse di sistema. La modalità supervisore è quindi programmata per interagire con l'hardware e poichè ne conosce tutti gli aspetti, solitamente, è abbastanza robusta da poterne comandare le azioni. All'utente abituale, invece, che mai conoscerà alla precisione tutti gli aspetti architetturali con cui funziona l'hardware sottostante non è concessa la possibilità di una interazione diretta. Il sistema operativo si limita a fornire alla modalità utente svariate chiamate di sistema, le system calls.
Lo stato supervisore o anche stato kernel è quella modalità di funzionamento che permette l'accesso a tutte le risorse di sistema. Le system calls sono usate da tutte le applicazioni in esecuzioni allo stato utente per chiedere aiuto al sistema operativo. Si tratta essenzialmente di sottoprogrammi particolari, diversi dalle normali jump a subroutine, che invece necessitano di un meccanismo hardware per la gestione basato su interruzioni. Le system calls, quando invocate, portano il sistema dallo stato utente allo stato supervisore che gode di maggiori privilegi e che generalmente manda a sua volta in esecuzione un pezzo del sistema operativo.

Le system calls possono essere bloccanti oppure non bloccanti. Si parla di system calls bloccanti quando l'esecuzione del processo che ha invocato la system calls viene bloccata in attesa del risultato prodotto dall'elaborazione della system calls chiamata. Le system calls non bloccanti possono invece avere seguito senza bloccare un processo. Il sistema operativo può interrompere l'esecuzione di un programma utente per effettuare operazioni di gestione, l'interruzione avviene attraverso il meccanismo delle interruzioni hardware. Ad esempio, il sistema operativo potrebbe assegnare la CPU ad un nuovo processo quando l'interruzione hardware (le cosiddette interrupt) generata dal clock di sistema indica al processo in esecuzione che il time slice è scaduto.
Le system calls, invece, vengono attivate da interruzioni software (le cosiddette trap). Le interrupt sono asincrone rispetto al programma in esecuzione mentre le trap sono sincrone con il programma in esecuzione perchè causate proprio da quest'ultimo che così facendo attira l'attenzione della CPU affinchè il sistema passi dalla modalità utente alla modalità supervisore.
Si immagini ad esempio una istruzione ciclica contenuta all'interno di un processo utente. Supponiamo poi che l'istruzione ciclica (magari perchè mal formulata) non giunga mai al termine dell'iterazione. Sotto queste ipotesi, in assenza di un segnale di clock, il processo utente impegnerà il proprio time slice anche quando quest'ultimo è ormai scaduto da tempo. Invece, in presenza del segnale di clock, come più volte anticipato, è possibile generare una interruzione hardware il cui significato è essenzialmente il passaggio ad un successivo processo (il gestore dei processi metti in atto il passaggio). Stiamo quindi scoprendo che le interruzioni hardware sono essenziali per il corretto funzionamento del sistema operativo poiché evitano che un processo utente impegni sempre e da solo le risorse di sistema. Le trap, quando inserite nel programma utente, consentono l'arresto del programma a favore della system calls invocata. Dunque il sistema operativo conosce due metodi per l'interruzione di un processo che può così avvenire mediante interrupt oppure mediante trap.
Le trap vengono provocate dall'esecuzione di una determinata istruzione all'interno del programma utente in esecuzione. Esse vengono trattate in modo del tutto analogo alle interruzioni. Ciascuna trap ha un identificatore che viene usato per cercare l'indirizzo della prima istruzione del corrispondente interrupt handler (ISR). Le chiamate alle system calls sono realizzate tramite trap. Alla ricezione di un interrupt/trap la CPU passa da user mode a kernel mode: in questo modo l'interrupt handler (che è un componente software del sistema operativo) esegue in kernel mode un pezzo del sistema operativo. Alla istruzione di ritorno dall'interrupt viene ripristinata la vecchia status word (PSW) del programma precedentemente interrotto, il sistema ritorna allo user mode.



La figura che segue mostra tutti i passi che si susseguono alla chiamata della system calls *read( )*:

```
read(fd,buffer,nbytes)
```

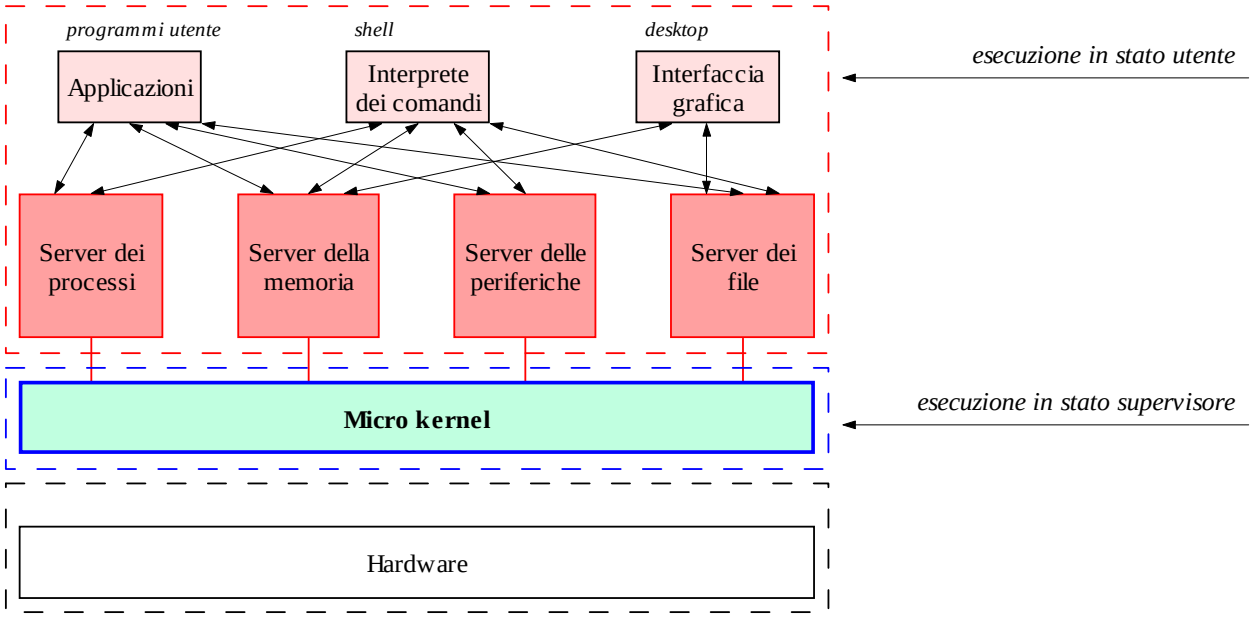


In seguito osserveremo alcune delle più importanti system calls offerte dal sistema Unix.

Organizzazione a livelli
Nell'organizzazione a livelli (adottata per la prima volta da E.W. Djkstra nel lontano 1968) il sistema è appunto strutturato in livelli. Ogni livello è fondato su quello sottostante e lo scambio delle informazioni da un livello ad un altro avveniva attraverso le system calls. Nel sistema operativo THE erano previsto 6 livelli: il livello 0 si occupava dell'allocazione del processore; il livello 1 era responsabile della gestione della memoria, il livello 2 garantiva la comunicazione fra l'operatore ed i processi.; il livello 3 effettuava la gestione dei dati in ingresso e uscita; il livello 4 era rivolto ai programmi utente; il livello 5 ospitava il processo dell'operatore. Una organizzazione assai simile era quella adottata dal sistema operativo MULTICS che anzichè essere organizzato a livelli prevedeva dei cerchi concentrici (in effetti è comunque riconducibile ad una struttura a livelli). I cerchi concentrici più esterni lanciavano una trap quando volevano chiamare una system calls. I parametri inseriti nella chiamata venivano dapprima verficati e se ritenuti idonei si procedeva all'attivazione della system calls.

Organizzazione a macchine virtuali
L'organizzazione del sistema operativo secondo macchine virtuali nasce dall'esigenza dei vecchi utenti del sistema operativo OS/360. Questi ad un certo punto iniziarono a desiderare un sistema di tipo time sharing cosicchè IBM rilasciò, diversi anni dopo, una versione time sharing del sistema operativo OS/360 e che si chiamava TSS/360. Questo sistema operativo non era tuttavia veloce (oltre ad essere voluminoso) e non suscitò particolare interesse presso gli utenti che dovettero così attendere una nuova versione del sistema operativo, il VM/370. Il VM/370 aveva una certa dotazione di hardware, presso i terminali utenti (solitamente un monitor ed alcune risorse di sistema capace anche di supportare il vecchio OS/360) girava, invece, una virtualizzazione dell'hardware vero. Questo concetto è stato poi ripreso in seguito da Sun Microsystem con l'avvento della JVM (una macchina estesa virtuale).

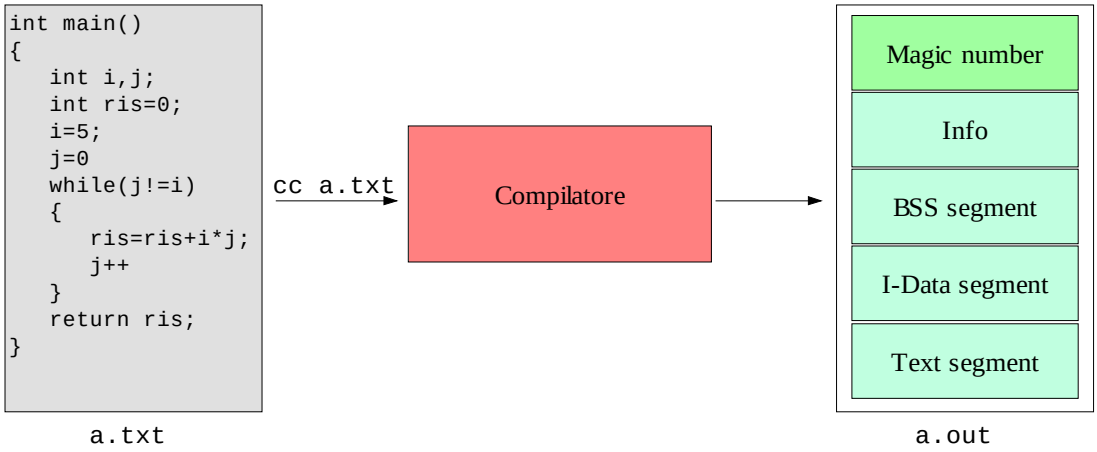
Organizzazione client/server
Altri sistemi operativi adottano una diversa organizzazione detta client/server che cerca di ridurre al minimo le funzionalità di lavoro in modalità supervisore (le system calls) e si basano quindi su un micro kernel detto anche nucleo minimo. Molte funzioni sono realizzate da processi server che sono in esecuzione nella modalità utente. Il nucleo minimo si compone essenzialmente di funzioni di base per la gestione dei processi, essenzialmente orientate alla comunicazione che avviene tramite messaggi di *send( )* e *receive( )*. Questo modello è più sicuro della organizzazione monolitica ma risulta meno efficiente e si adatta bene a sistemi operativi di rete.



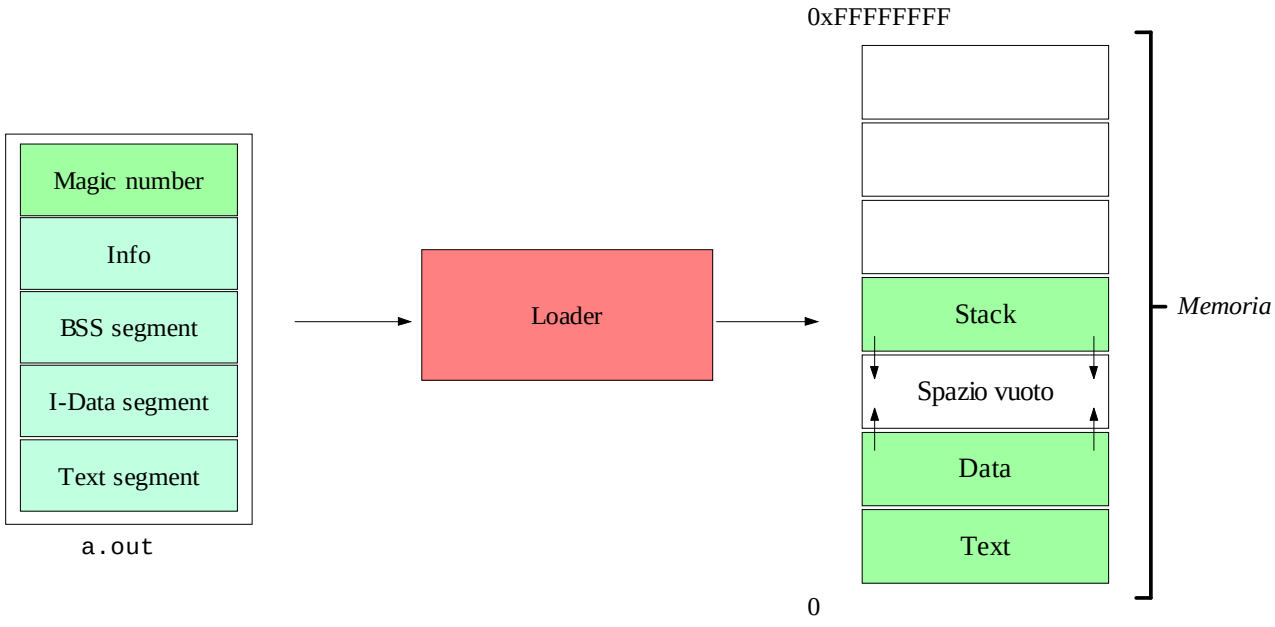
Il kernel si occupa della gestione della comunicazione tra client e server. Il sistema operativo è qui suddiviso in più parti, ognuna di queste cura un solo aspetto del sistema. In questo modo le singole parti sono più maneggevoli e piccole. Questa organizzazione si addice a sistemi operativi in ambito distribuito, infatti se un server invia un messaggio (una richiesta tramite *send( )*) non ha bisogno di sapere se il messaggio viene gestito localmente al sistema oppure su di una macchina remota. In entrambi i casi, infatti, il processo client riceverà comunque una risposta dal processo server. Alcuni esempi di sistemi operativi basati su una organizzazione client/server sono MACH e Minix (derivate da UNIX) e Windows NT 3.0.

Il file eseguibile di Linux
Il programma utente è generalmente scritto in un file di testo ed è formattato seguendo la sintassi offerta dal linguaggio di programmazione adottato. Quest'ultimo è di sicuro più vicino al linguaggio umano e permette di formulare una descrizione sufficientemente idonea a modellare una determinata esigenza. L'operazione di compilazione si base propio sul programma utente ed a partire da questo genere un file eseguibile il cui contenuto dipende fortemente dal sistema oerativo sottostante. Si tratta di un file binario al cui interno vengono mappate tutte le informazini necessarie a rendere eseguibile il programma utente. In altre parole nell'eseguibile troviamo tutte le informazioni necessarie affinchè il programma utente diventi poi un processo.
Nel corso di queste note osserveremo come esempio il file eseguibile di Linux, talvolta detto più brevemente ELF (*execute linux file*). Il compilatore sotto sistema operativo Linux (le cui funzionalità su di un file, opportunamente preparato e scritto ad esempio in linguaggio C, vengono richiamate attraverso il comando `cc programma.c`) si occupa di generare il file eseguibile che è possibile suddividere al suo interno in più sezioni informative. La sezione informativa più in basso è detta *text segment*, in essa troviamo le istruzioni del programma utente. Nella sezione *I-Data* prendono posto tutte le variabili globali inizializzate dal programma, mentre nella sezione *BSS-segment* si trovano le variabili globali non inizializzate. La dimensione di questa sezione dipende fortemente dal tipo di variabile utilizzata dal programma utente. Procedendo dal basso verso l'alto

troviamo una sezione dedicata a più informazioni del programma e per finire vi è una sezione dati contraddistinta da un *magic number*, un numero che identifica il file come eseguibile.



Il programma utente, così come il file eseguibile, risiedono sulla memoria di massa del calcolatore. Il programma utente può essere modificato in ogni momento, il file eseguibile, invece, può essere caricato in memoria per generare un nuovo processo utente. Il loader si occupa di caricare in memoria centrale lo spazio di indirizzamento del file eseguibile.



La sezione *Text* contiene le istruzioni del programma tradotte dal loader in linguaggio macchina, la sezione *Data* contiene le variabili gloabili del programma utente mentre la sezione *Stack* è una pila di frame contenente le chiamate non ancora ritornate. Inizialmente, quando il programma utente viene mandato in esecuzione, sullo stack viene caricata la sola funzione principale *main( )*. Ad ogni funzione chiamata corrisponde un frame aggiunto sullo stack, per ogni funzione che termina si procede alla rimozione di un frame dallo stack. In tal caso l'esecuzione del programma continua a partire dall'indirizzo di ritorno. L'area *Data* destinata alle variabili globali è dedicata sia alle variabili statiche che a quelle dinamiche (*heap*) che quindi possono variare a tempo di esecuzione del programma. Quindi, sia l'area *Data* che l'area *Stack* possono subire una crescita (quest'ultima in base alle chiamate non ancora ritornate) secondo il verso indicato dalle frecce in figura. Tale crescita è agevolata da un opportuna area vuota che funge in questo modo da polmone. Il processo viene mappato in memoria in un area consecutiva di indirizzi, se lo *Stack* oppure l'area *Data* crescono eccessivamente si provvede a rilocare l'intero processo, questa volta con un area vuota

sufficientemente più grande (se possibile).
Quando il processo è in esecuzione (ciò si verifica quando a seguito dell'indirizzamento in memoria al programma viene assegnato anche la risorsa processore) il sistema operativo può intervenire solo se: si verifica un interruzione hardware (il segnale di clock che indica lo scadere del time slice); il processo termina la sua elaborazione; il processo invoca al suo interno una system call.

Il sistema UNIX e le system calls
Per utilizzare i servizi del sistema operativo, il programmatore ha a disposizione una serie di funzioni di libreria, dette system calls che variano da una settantina ad oltre 200, a seconda della versione di UNIX. Il processo di standardizzazione ha come obiettivo la realizzazione di un insieme minimo di interfacce per sistemi operativi. POSIX ad esempio standardizza un nucleo base di system calls, dette anche primitive, in circa 120 interfacce, come sorgenti C.
La scelta operata da POSIX per la standardizzazione delle funzioni di base segue il cosiddetto approccio minimalista, nel senso che vengono scelte o usate le funzioni di tipo "normale" che un applicazione può richiedere. Vengono poi fornite delle estensioni non POSIX che includono primitive e/o opzioni addizionali. POSIX inoltre include anche alcune primitive della libreria standard C, questo ovviamente genera delle sovrapposizioni di funzioni che sono quindi comune ai due standard. Ad esempio la funzione *fopen( )* è standard C e POSIX. Un sorgente C Standard conforme a POSIX può essere eseguito, una volta ricompilato, su qualunque sistema POSIX dotato di un ambiente di programmazione C Standard. Introdotta quindi l'importanza delle system calls procediamo adesso nella loro suddivisione che, essenzialmente, è la seguente:

- Gestione di file e directory;
- Gestione dei processi;
- Comunicazione fra processi;

Per ciascuna categoria affronteremo in seguito lo studio delle principali system calls mostrando, ove possibile, qualche esempio. Tuttavia è bene precisare quanto segue, non è importante imparare a memoria tutte le funzioni standard di POSIX poiché in caso di aiuto l'utente e/o il programmatore può ricorrere all'uso di appositi manuali in linea. Il manuale in linea è il primo supporto alla programmazione. Per richiamare il manuale in linea è sufficiente digitare a linea di comando la parola chiave *man( )* che per i sistemi operativi UNIX e LINUX fornisce l'accesso alla tradizionale documentazione in linea.
Le "*man pages*" sono file formattati in modo speciale, digitando `man nomecomando` verrà visualizzata la pagina del manuale relativa al comando denominato `nomecomando`. Le pagine del manuale vengono raggruppate in sezioni numerate. Ad esempio, è possibile incontrare l'indicazione `man(1)`, la quale indica che il comando è documentato nella sezione 1, di seguito viene mostrato un elenco dei parametri numerici che il comando `man( )` può accettare:

- Sezione 1, comandi utente (viene mostrata solo l'introduzione);
- Sezione 2, chiamate di sistema;
- Sezione 3, chiamate della libreria C;
- Sezione 4, dispositivi (ad esempio hd, sd);
- Sezione 5, formati dei file e protocolli;
- Sezione 6, giochi (introduzione);
- Sezione 7, informazioni su convenzioni, pacchetti, macro;
- Sezione 8, per la manutenzione di sistema;

In realtà ci sono altri comandi utili in fase di documentazione come ad esempio il comando `whatis` e `apropos`, il cui scopo comune è quello di agevolare la ricerca di informazioni all'interno del sistema delle pagine del manuale. `whatis` produce una descrizione molto breve dei comandi di sistema mentre `apropos` viene utilizzato per la ricerca delle pagine di manuale contenenti un dato argomento.

La gestione degli errori
La maggior parte delle system calls restituisce il valore `-1` in caso di errore ed assegna un particolare codice di errore alla variabile globale `errno`. In caso di successo di una system calls la variabile `errno` non viene azzerata, per cui essa contiene sempre l'ultimo errore verificatosi. L'header `errno.h` contiene la definizione dei nomi simbolici dei codici di errore. Aprendo tale file troveremo al suo interno una cosa di questo tipo:

```
#define EPERM     1 /* Not owner */
#define ENOENT    2 /* No such file or directory */
#define ESRCH     3 /* No such process */
#define EINTR     4 /* Interrupt system call */
#define EIO       5 /* I/O errore */
...
```

dove si osserva che al numero di un possibile errore corrisponde un abbreviazione mnemonica ed una descrizione breve. La primitiva `perror()`, così definita:

```
void perror(const char *str)
```

converte il codice in errno in un messaggio, solitamente in inglese, e lo stampa anteponendogli la stringa `str`. Ad esempio:

```
...
fd=open(“noneexist.txt”,RDONLY);
if(fd==-1) perror(“Si è verificato l'errore”);
...

Sul monitor:
Si è verificato l'errore: No such file or directory
```

la primitiva `perror` sta per “print error”.

Apertura di un file: open(…)
Il primo esempio di system calls che vediamo riguarda la funzione che si occupa dell'apertura di un file. Un file per essere usato deve essere aperto. L'apertura di un file comporta tutta una serie di operazioni/istruzioni che essenzialmente hanno il compito di trasferire nella memoria centrale la struttura del file. La funzione open è così definita:

```
int open(const char *path, int oflag, ...);
```

La `open()`:

- Localizza il file nel file system attraverso il suo `pathname`;
- Copia in memoria il descrittore del file (i-node);
- Associa al file un intero non negativo (file descriptor), che verrà usato nelle operazioni di accesso al file, invece del `pathname`.

Poiché un file aperto occupa spazio in memoria esiste un limite massimo di file apribili, come vedremo in seguito la chiusura di un file libera le risorse di sistema, in particolare la memoria che può quindi essere nuovamente impiegata.
Una chiamata ad `open()` restituisce al programma un descrittore di file, si tratta di una chiave (un numero positivo) per accedere alla struttura. Alcuni file standard sono già aperti di default dalla shell, come ad esempio i file di input (tastiera) e file di output (video) che rispettivamente hanno il descrittore di file pari a 0 (tastiera) e ad 1 (video). Infine un ulteriore file, dedicato agli errori, è già aperto per default e ad 'esso si associa il descrittore di file con numero 2. Quindi un successivo file aperto avrà come intero associato al proprio descrittore di file il numero 3. Quindi, riassumendo la primitiva `open()` apre (o crea) il file specificato in `pathname`, secondo la modalità specificata in `oflag`; restituisce il file descriptor con il quale ci si riferirà al file successivamente (oppure

restituisce -1 in caso di errore). I permessi e le modalità con cui aprire un file sono:

`O_RDONLY`	Modalità di sola lettura
`O_WRONLY`	Modalità di sola scrittura
`O_RDWR`	Modalità di lettura e scrittura
`O_APPEND`	Modalità di accodamento al precedente contenuto
`O_CREAT`	Creazione del file

Creazione di un file: `creat(…)`
Per la creazione di un file esiste anche un apposita primitiva così definita:

```
int creat(const char *path, mode_t mode);
```

- Crea un nuovo file normale di specificato `pathname`, e lo apre in scrittura;
- `mode` specifica i permessi iniziali; l'owner è l'effective user-id del processo;
- Se il file esiste già, lo svuota;
- Restituisce il `file descriptor` oppure `-1` in caso di errore;

Chiusura di un file: `close(…)`
Per la chiusura di un file esiste la primitiva `close()`:

```
int close(int fildes)
```

- Chiude il descrittore del file;
- Restituisce l'esito dell'operazione e quindi `0` in caso di successo, `-1` altrimenti;

Lettura di un file: `read(…)`
La lettura di un file è affidata alla primitiva `read()`:

```
ssize_t read(int fildes, void *buf, size_t nbyte);
```

- Legge in `*buf` una sequenza di `nbyte bytes` dalla posizione corrente del file `fildes`;
- Aggiorna la posizione corrente;
- Restituisce il numero di `bytes` effettivamente letti, oppure `-1` in caso di errore;

Scrittura di un file: `write(…)`
La scrittura di un file è affidata alla primitiva `write()`:

```
ssize_t write(int fildes, const void *buf, size_t nbyte);
```

- Scrive nel file `fildes nbyte` bytes da `*buf` a partire dalla posizione corrente;
- Aggiorna la posizione corrente;
- Restituisce il numero di bytes effettivamente scritti, oppure `-1` in caso di errore;

Avendo fornito alcune primitive di sistema è possibile scrivere già un elevato numero di programmi, quello che segue è un esempio che si occupa di copiare lo standard input sullo standard output:

```
/* Copia lo standard input sullo standard output Ver.1.0 */
#include <stdio.h>
#define BUFFSIZE 8192
int main(void)
{
      int n;
      char buf[BUFFSIZE];
      while( (n=read(0, buf,BUFFSIZE))>0)
      {
            if( write(1, buf, n)!=n)
                  perror("ERRORE");
      }
      if(n<0)
            perror("ERRORE nella funzione read(…)");
      exit(0);
}
```

Attenzione, è corretto usare anche `STDIN_FILENO` e `STDOUT_FILENO` definiti in `unistd.h` al posto del descrittore di file `0` per lo standard input (tastiera) ed `1` per lo standard output (video). In questo programma si può notare che lo standard input e output non vengono né aperti e né chiusi!

Posizionamento su un file: `lseek(…)`
Tale funzione permette di posizionarci su una particolare area del file:

```
off_t lseek(int fildes, off_t offset, int whence);
```

- Sposta la posizione corrente nel file `fildes` di `offset` bytes a partire dalla posizione specificata in `whence`:

  `SEEK_SET` dall'inizio del file;
  `SEEK_CUR` dalla posizione corrente;
  `SEEK_END` dalla fine del file;

- Restituisce la posizione corrente dopo la `lseek`, oppure `-1` in caso di errore;

Questa funzione non effettua alcuna operazione di I/O.

Creazione di un hard link: `link(…)`
Questa funzione consente di accedere ad un file con un'altro nome e da due percorsi diversi poiché genera un collegamento ad un file, è così definita:

```
int link(const char *existing, const char *new);
```

- Crea il nuovo (hard) link `new` al file `existing`;
- Restituisce `0` se ok altrimenti `-1`;

Rimozione di un hard link: `unlink(…)`

```
int unlink(const char *path);
```

- Distrugge il link(hard) `path` e, se si tratta dell'ultimo, dealloca il file;
- Restituisce l'esito dell'operazione, `0` per ok, `-1` altrimenti;

Se qualche processo sta usando il file, viene solo deallocata la directory entry: il file verrà rimosso quando tutti i descrittore di file relativi al file saranno chiusi. In questo modo un eseguibile può fare `unlink` di se stesso e proseguire comunque la esecuzione.

Gestione di directories
Per creare una directory vuota si usa la seguente primitiva:

```
int mkdir(const char *path, mode_t mode);
```

Per rimuovere una directory:

```
int rmdir(const char *path);
```

Per cambiare la working directory:

```
int chdir(const char *path);
```

Permessi di un file: `chmod(…)`
Questa primitiva modifica le modalità e/o i permessi di accesso a file e cartelle, è così descritta:

```
int chmod(const char *path, mode_t mode);
```

dove `chmod` sta per “change mode”:

- Cambia i permessi del file `path`, come specificato in `mode`;
- Restituisce l'esito dell'operazione, `0` per ok `-1` altrimenti;

Duplicazione di file descriptors: `dup(…)`
La primitiva `dup`, crea per un file già aperto, un ulteriore descrittore di file:

```
int dup(int fildes);
```

- Associa al file descriptor `fildes` un file descriptor aggiuntivo (il primo a partire da 0);
- Restituisce il nuovo file descriptor, oppure `-1` in caso di errore;

Tabella delle principali primitive di gestione file

Primitive POSIX	*Descrizione*
`fildes=creat(path, mode)`	Crea un nuovo file
`fildes=open(path, oflag, ...)`	Apre un file
`r=close(fildes)`	Chiude un file aperto
`n=read(fildes,&buf,nbyte)`	Legge i dati da un file nel buffer
`n=write(fildes,&buf,nbyte)`	Scrive i dati da un file nel buffer
`newoffset=lseek(fildes, offset, whence)`	Muove il puntatore di file in un punto del file
`r=link(existing, new)`	Crea un nuovo link a un file
`r=unlink(path)`	Rimuove un link
`r=rename(oldpath,newpath)`	Cambia nome a un file
`r=stat(path, &buf)` `r=fstat(fildes, &buf)`	Fornisce informazioni sullo stato del file
`r=mkdir(path, mode)`	Crea una nuova directory
`r=rmdir(path)`	Rimuove una directory vuota
`r=chdir(path)`	Cambia la working directory
`newfildes=dup(fildes)`	Duplica il file descrittore
`s=pipe(fd_pipe)`	Crea una pipe senza nome
`s=chdir(dirname)`	Cambia la directory dilavoro
`r=chmod(path, mode)`	Cambia le protezioni di un file
`r=chown(path, owner, group)`	Cambia l'owner e/o il gruppo di un file
`r=utime(path, &time)`	Cambia il tempo di accesso e modifica di un file

Primitive per la gestione dei processi
In UNIX ogni processo è creato da un processo “padre” mediante una chiamata di sistema `fork()`. Il processo padre in esecuzione, a seguito di una `fork()`, viene duplicato in un altra area di memoria generando quindi un vero e proprio clone del processo “padre” che tuttavia è ben distinguibile grazie ad un valore restituito appunto dalla primitiva di sistema. Un processo “figlio” può inoltre eseguire un proprio codice o set di istruzioni ereditando dal padre tutti i parametri fin qui utilizzati. Quindi, riassumendo, padre e figlio condividono lo stesso codice, ed il figlio eredita una copia dei dati del padre. La primitiva `fork()` è così definita:

```
pid_t fork(void);
```

dunque, la `fork()` non richiede parametri e restituisce un intero che:

- per il processo figlio vale `0`;

- per il processo padre è un valore positivo che rappresenta il PID del figlio;
- è un valore negativo pari a `-1` in caso di errore;

La `fork()` inoltre, quando invocata, genera le seguenti azioni:

- Allocazione di una nuova process structure nella process table associata al processo figlio e sua inizializzazione (sarà più chiaro in seguito);
- Allocazione di una nuova user structure nella quale viene copiata la user structure del padre;
- Allocazione dei segmenti di dati e stack del figlio nei quali vengono copiati dati e stack del padre;
- Aggiornamento della text structure del codice eseguito (condiviso con il padre) ed incremento del contatore dei processi;

Ecco un esempio di come va usata la primitiva `fork`:

```
...
pid=fork();
if(pid<0)
{
      /* la fork è fallita */
}
else if(pid>0)
{
      /* operazioni del padre */
}
else
{
      /* operazione del figlio*/
}
...
```

Le primitive `wait(…)` e `waitpid(…)`
Si tratta di due primitive che usate in combinazione con la `fork` ne consentono una gestione ottimale, la prima è così definita:

```
pid_t wait(int *statloc);
```

- Sospende il processo chiamante, fino a che uno qualunque dei suoi figli termina;
- Restituisce il PID del figlio terminato, oppure -1 se non ci sono figli;
- Assegna a `statloc` l'exit status del figlio;

mentre:

```
pid_t waitpid(pid_t pid, int *statloc, int options);
```

permette di specificare di quale figlio si attende la terminazione. Un processo può terminare involontariamente ad esempio mediante interruzione da segnali (in seguito saranno trattati i segnali) oppure volontariamente, ciò può avvenire perché il processo può eseguire l'ultima istruzione oppure perché viene fatta una chiamata alla primitiva `_exit`.

```
void _exit(int status);
```

- Termina il processo chiamante;
- Rende disponibile il valore di `status` al processo padre (che lo otterrà tramite `wait`);

La primitiva della C Standard library `exit()` chiama al suo interno la primitiva POSIX `_exit()`.
Quando un processo “figlio”viene terminato passa nello stato di zoombie, occupando un insieme minimale di risorse, esso viene definitivamente rimosso dopo che il padre ha ricevuto il suo stato di terminazione con una `wait()`.

Esempio di un processo che attiva due figli e ne attende la terminazione

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

main()
{
   pid_t pid;
   int status;

   /* genera il primo figlio */
   pid=fork();
   if(!pid)
   {
       /* codice del primo figlio */
       printf("Sono il primo figlio:\nPID=%d\nPPID=%d\n",getpid(),getppid());
       exit(0);
   }
   else
   {
       /* codice del padre */
       pid=fork();
       if(!pid)
       {
       /* codice del secondo figlio */
        printf("Sono il secondo figlio:\nPID=%d\nPPID=%d\n",getpid(),getppid());
        exit(0);
       }
       else
       {
       /* codice del padre */
       /* attende il primo figlio che termina */
        pid=wait(&status);
        printf("Primo figlio terminato:\nPID=%d\nSTATO=%d\n",pid,status);
       /* attende il secondo figlio che termina */
        pid=wait(&status);
        printf("Secondo figlio terminato:\nPID=%d\nSTATO=%d\n",pid,status);
        printf("Programma terminato\n");
       }
   }
}
```

Esecuzione di un programma
La primitiva:

```
exec (pathname, argomenti)
```

- Sostituisce l'immagine del chiamante con il file eseguibile `pathname`, e lo manda in esecuzione passandogli gli `argomenti`.

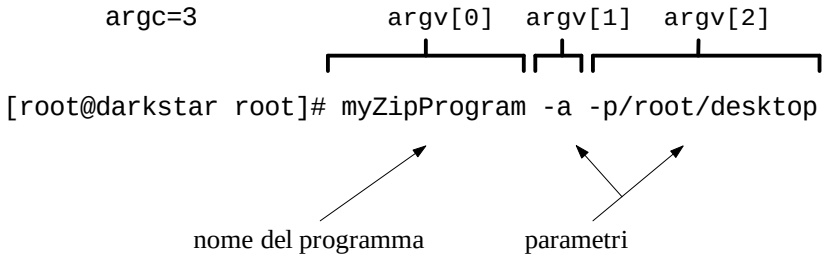
A questo punto accade che i dati del processo padre non sono più visibili al processo figlio. La funzione `exec()` prevede eventuali parametri iniziali che possono essere passati al processo/programma. La funzione `main` è la prima funzione ad essere attivata e quindi può avere dei parametri di ingresso:

```
int main(int argc, char *argc[])
```

I parametri vengono passati come stringhe/sequenze di caratteri. In prima posizione troviamo il nome del programma per cui ad `argc` va sempre sommato uno se si vuole passare un successivo parametro:

```
argv[0]=nome del programma
argv[1]=primo argomento
argc = 2
```

Quindi, `argc` contiene il numero dei parametri che si stanno passando, mentre con `argv[i]` si accede all'i-esimo parametro.



In realtà ci sono diverse versioni di `exec` ed una di queste è la primitiva `execve()`:

```
int execve(const char *pathname, char *const argv[], char *const envp[]);
```

- Esegue il file di `pathname` specificato, passandogli gli argomenti `argv`, e l'environment `envp`.

Quindi, con la primitiva `exec` individuiamo una famiglia di primitive che differiscono tra di loro per il significato dai parametri. Più brevemente possiamo subito dire che con la lettera finale (che precede quindi il comando classico) `l` si vuole denotare la possibilità di passare al processo da lanciare, tramite `exec`, una lista di parametri; con `ve` l'environment e con `p` il path.

La primitiva `times(…)`

```
clock_t times(struct tms *buf);
```

- Restitusce in `buf` alcuni contatori relativi al tempo di CPU usato dal processo e dai suoi figli:

```
struct tms
{
   clock_t tms_utime;   /* user CPU time */
   clock_t tms_stime;   /* system CPU time */
   clock_t tms_cutime;  /* user CPU time of terminated children */
   clock_t tms_cstime;  /* system CPU time of terminated children */
}
```

Struttura della shell di Linux
In ambiente UNIX `init` è addetta all'inizializzazione dei processi generando, mediante primitiva `fork`, un processo per ogni terminale ed in particolare per l'esecuzione del comando `getty`. Tale processo controlla l'accesso al sistema mediante l'esecuzione e quindi mediante `exec` del comando `login`. In caso di accesso corretto `login` esegue la shell, un interprete di comandi. Ogni utente che accede inserisce un nome ed una password che, se corretti, danno la possibilità di interagire con l'interprete. Ad ogni comando digitato la shell genera un processo figlio. Esistono due modalità di comportamento previste da una shell, infatti, ad ogni comando la shell genera un figlio è può:

- mettersi in attesa della terminazione del figlio (esecuzione in foreground);
- continuare l'esecuzione in contemporanea con il figlio (esecuzione in background);

La shell non fa parte del sistemaoperativo anche se ne usa frequentemente le funzionalità, si tratta di un interfaccia rivolta all'utente attraverso la quale invocare le system call. Essa usa un terminale standard di input che è la tastiera del sistema ed un terminale standard di output che è invece il monitor di sistema. Ogni shell si presenta all'utente attraverso un prompt, il segno del dollaro indica che questa è pronta a ricevere un comando, il segno cancelletto, invece, contraddistingue l'utente root da quello abituale. Nella shell di linux è possibile impostare un comando da eseguire in modalità background (la shell dopo una `fork()` genera un processo figlio a cui assegna il compito di portare a termine il comando dell'utente) facendo seguire il carattere & al comando. In questo

modo la shell ripropone all'utente un nuovo prompt, su una nuova riga, offrendo la possibilità di eseguire unaltro comando

Esempio di Shell

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>

main()
{
      pid_t pid, procid;
      int status, k;
      size_t n;
      char buffer[80], prompt[30];
      sprintf(prompt,"myprompt:>");
      write(1,prompt,10);
      bzero(buffer,80);
      while(n=read(0,buffer,80)!=0)
      {
      /* command line processing */
      k=strlen(buffer);
      buffer[k-1]=buffer[k];
      if(strcmp(buffer,"esci")==0) exit(0);
      printf("attivazione processo figlio %s\n",buffer);
      if((pid=fork())==0)
      {
            if(execlp(buffer,(char*)0)==-1) exit(1);
      }
      procid=wait(&status);
      if(status!=0) write(2,"cmd not founf\n",14);
      bzero(buffer,80);
      write(1,prompt,10);
      }
exit(0);
}
```

Comunicazione fra processi
I processi lavorano in aree di memorie separate. La comunicazione fra processi va fatta quindi scambiando dei messaggi e non leggendo negli indirizzi di memoria che contengono le variabili degli altri programmi. Ecco, dunque, la necessità di implementare un meccanismo per la sincronizzazione tra i processi, la tecnica usata da UNIX adotta la soluzione dei segnali. Si tratta di un interruzione software mandata ad un processo attraverso un meccanismo di comunicazione che notifica un evento asincrono al programma. I meccanismo più comuni usati da UNIX per la comunicazione dei processi sono:

- pipes;
- segnali;
- semafori;
- memoria condivisa;
- sockets

Le pipe
Una pipe è un file di dimensione limitata gestito secondo una politica FIFO:

- Un processo produttore deposita dati (e resta in attesa se la pipe è piena);
- Un processo consumatore legge dati (e resta in attesa se la cosa è vuota);

Esistono due tipi di pipes: pipe con nome, create da `mknod` (devono essere aperte con `open`); pipe senza nome, create e aperte da pipe su "`pipe device`" definito al momento della configurazione;
La creazione di pipe senza nome avviene tramite system call:

```
int pipe(int fildes[2]);
```

che:

- crea una pipe, aprendola in lettura e scrittura;
- restituisce l'esito dell'operazione, `0` per ok oppure `-1` in caso di errore;
- Assegna a `fildes[0]` il descrittore di file del lato aperto in lettura, e a `fildes[1]` quello del lato aperto in scrittura.

Questo metodo va bene se utilizzato in concomitanza con la primitiva `fork()` poiché favorisce l'interazione tra i processi che possono così accordarsi su chi dovrà usare `fildes[1]` per la scrittura e `fildes[0] per la lettura`. Un esempio:

```
#include <stdio.h
#include <stdlib.h

int main()
{
      int pid,j,c;
      int piped[2];

      /* Apre la pipe creando due fd memorizzati nell'array piped[] */
      if(pipe(piped)<0)
            exit(1);

      if((pid=fork()<0)
            exit(2);
      else if(pid==0)   /* figlio che ha una copia di piped[]*/
      {
            close(piped[1]);
            for(j=1;j<=10;j++)
            {
                  read(piped[0],&c,sizeof(int));
                  printf("figlio: ho letto dalla pipe il numero %d\n",c);
            }
      exit(0);
      }
      else /* padre */
      {
            close(piped[0]);
            for(j=1;j<=10;j++)
            {
                  write(piped[1],&j,sizeof(int));
                  printf("Padre: ho scritto nella pipe il numero %d\n",j);
                  sleep(1);
            }
      exit(0);
      }
}
```

Segnali

Un segnale è una notifica a un processo a cui viene lanciato a seguito di un evento. Si tratta di un meccanismo che non consente lo scambio di messaggi fra processi ma consente solo lo scambio di segnali codificati, ciò può essere fatto perché ad esempio si verifica un errore in floating point, per notificare la morte di un figlio oppure a causa di una richiesta di terminazione (magari lanciata dall'utente che si è annoiato diaspettare). I segnali possono essere pensati come degli "interrupt software" e possono essere inviati da un processo ad un altro processo, da un processo a se stesso oppure dal kernel a un processo. Ogni segnale è identificato da un numero intero, associato a un nome simbolico e definito nella libreria `signal.h`.

Segnali in POSIX

Segnale	***Significato***	***Default***

Segnale	*Significato*	*Default*
SIGABRT	Abortisce un processo	Termina con dump
SIGALARM	Invia un segnale di “sveglia”	Termina
SIGCHILD	Lo stato di un figlio è cambiato	Ignora
SIGCONT	Continua un processo stoppato	Continua o ignora
SIGFPE	Floating Point Errore	Termina con dump
SIGHUP	Hangup su terminale	Termina
SIGILL	Istruzione di macchina illegale	Termina con dump
SIGINT	L'utente ha usato il tasto DEL del terminale per interrompere il processo	Termina
SIGKILL	Segnale per terminare un processo (non può essere ignorato)	Termina
SIGPIPE	Tentativo di scrivere una pipe che non ha descrittori	Termina
SIGQUIT	L'utente ha usato il tasto di quit del terminale	Termina con dump
SIGSEGV	Riferimento a un indirizzo di memoria non valido	Termina con dump
SIGSTOP	Segnale per stoppare un processo (non può essere ignorato)	Stoppa il processo
SIGTERM	Segnale per terminare un processo	Termina
SIGTSTP	L'utente ha usato il tasto suspend del tel terminale	Stoppa il processo
SIGTTIN	Un processo in background tenta di leggere dal suo terminale di controllo	Stoppa il processo
SIGTTOU	Un processo in background tenta di scrivere sul suo terminale di controllo	Stoppa il processo
SIGUSR1-2	Disponibile per scopi definiti dall'appl.	

Quando un processo riceve un segnale può comportarsi in diversi modi, esso infatti può:

- gestire il segnale con una funzione handler opportunamente definita dal programmatore;
- eseguire un azione predefinita dal S.O. (azione di default mostrata in tabella);
- ignorare il segnale (non sempre possibile);

Nei primi due casi il processo reagisce in modo asincrono al segnale, inoltre, alcuni segnali non possono essere ignorati. Linux ne prevede ben 32! Ed ogni processo può gestire esplicitamente un segnale utilizzando la system calls `signal()`:

```
void(*signal(int sig, void (*func)(int))) (int);
```

dove:

- `sig` è l'intero che individua il segnale da gestire;
- il parametro `func` è un puntatore ad una funzione che indica l'azione da associare al segnale;
- la primitiva ritorna un puntatore a funzione

la primitiva per inviare un segnale è in vece `kill()`:

```
int kill(pid_t pid, int sig);
```

- notifica il segnale `sig` al processo specificato in `pid`;
- restituisce il risultato dell'operazione (`0` se è stato inviato almeno un segnale, `-1` se errore);

Per motivi di protezione, infine, è ragionevole pensare quanto segue: il processo che invia il segnale e quello che lo riceve deve avere lo stesso owner ed inoltre l'owner del processo che invia il segnale deve essere superuser.

Per descrivere i segnali, si rende necessario introdurre il concetto di evento. Un evento, in pratica, è un fatto accaduto all'interno del sistema. Può essere generato dalle periferiche (interrupt, operazioni di I/O), da processi (fra cui quelli di sistema), ecc... Gli avvenimenti indicati con un evento possono essere vari, completamento di un'operazione i I/O, richiesta di chiusura di un processo, scadenza di periodi di tempo (allarme), ecc...
La generazione di un evento asincrono (cioè che può arrivare in qualsiasi momento) può avvenire tramite un segnale rivolto ad un processo. Quindi, con un segnale, viene indicato il verificarsi di un evento asincrono, oppure la necessità di compiere una determinata azione associata al segnale stesso. Vengono considerati eventi asincroni perchè, il processo destinatario del segnale, lo può ricevere solo quando riprende la propria esecuzione (passaggio dello stato di esecuzione da modo kernel a modo utente che sarà descritto in seguito). Un segnale può essere inviato da un processo ad un altro processo, dal kernel ad un processo oppure da un processo a se stesso (funzione `alarm`). In ogni caso, l'invio del segnale passa per il kernel. Infatti, il processo non riceve subito il segnale. Se questo si trova nella modalità utente (il processo è quindi in esecuzione) il kernel lo propaga immediatamente. Se, invece, il processo si trova in modo kernel (in attesa cioè di essere scelto dallo schedulatore dei processi per essere mandato in esecuzione), il segnale gli viene passato al momento della riattivazione da parte dello schedulatore. Vediamo adesso come si può inviare e catturare un segnale. L'header C contenente le funzioni relative alla gestione dei segnali è situato in /usr/include/signal.h. Il segnale voluto può essere mandato con le funzioni `alarm` e `kill` (`kill` non invia necessariamente un segnale per uccidere un processo), mentre può essere catturato per mezzo della funzione `signal`.

Alarm
Come già accennato, la funzione `alarm` si usa per far inviare un segnale al processo che la invoca. Più precisamente, `alarm` permette l'invio del segnale SIGALRM (si tratta di un segnale di sveglia) dopo un certo tempo `t` (specificato come parametro). Un esempio:

```
#include <signal.h>
...
unsigned t; //tempo che deve trascorrere
...
//A partire da questo momento, il processo
//ricevera' un segnale di allarme dopo t secondi.
alarm(t);
...
```

Nel frammento di codice mostrato potrebbe mancare una cosa: il processo deve essere in grado di catturare il segnale per poter avviare una routine utente (catcher). Questa routine può essere attivata tramite la funzione `signal` che modifica il comportamento di default del segnale. La funzione `signal` verrà descritta piu' avanti (essa si occupa di catturare il segnale e di far attuare un certo comportamento al programma che riceve il segnale). Un’ultima nota riguardante `alarm`: usando il valore `0` come tempo, si ottiene l’azzeramento del conteggio eventualmente già avviato (è come resettare un cronometro, qui lo si fa scrivendo `0` in `t`).

Kill

`Kill` si occupa di inviare un segnale ad un processo di cui si conosca il PID, o ad un gruppo di processi con lo stesso PGID (Progress Group IDentification number). Abbiamo visto che è comunque il kernel che, alla fine, invia il segnale al processo. Il kernel, per identificare un processo, può usare sia il PID che il PGID. Per la gestione del PID e del PGID, esistono diverse funzioni come `getpgid()`, `setpgid()`, ecc...(per informazioni provate a digitare man `getpgid`). Tenendo conto che un segnale è identificato da un numero, i parametri accettati da `kill` sono due: il PID del processo che deve ricevere il segnale ed il numero del segnale stesso, entrambi interi. Un esempio:

```
#include <signal.h>
...
int errore=0;
...
//invia il segnale e riporta l'eventuale errore.
errore=kill(40,SIGSEGV);
...
```

Il segnale SIGSEGV (vedi tabella, oppure il file `/usr/include/bits/signum.h`) invia, al processo il cui PID è 40 (numero inventato a caso), l'indicazione di violazione di segmento di memoria (in altre parole, un indirizzo di memoria non valido). Nella variabile `errore` ci sarà `0` se il segnale e' stato inviato correttamente, `-1` altrimenti (nella variabile `errno` ci sarà il codice dell'errore).

Signal

Con la funzione `signal` è possibile catturare i segnali che arrivano ad un processo e attuare quindi un determinato comportamento.Il meccanismo di utilizzo di questa funzione, prevede la conoscenza del numero di segnale e la creazione di una funzione che lo gestisca. Un esempio:

```
#include <signal.h>
...
void gestoreKill()
{
   /* procedura eseguita quando viene ricevuto un segnale di kill */
}
...

int main(void)
{
   /* Assegna al segnale SIGKILL la funzione gestoreKill */
   signal(SIGKILL, gestoreKill);
   ...
}
```

Questo piccolo pseudo-programma cattura il segnale di uccisione del processo e lo gestisce in qualche modo con la funzione `gestoreKill()`. La struttura mostrata, non soddisfa, però, tutte le esigenze: infatti, dopo che un segnale viene catturato, l'azione (meglio, la funzione) ad esso associata, viene riportata a quella di default! Cioè, nel programmino precedente, il segnale viene catturato solo la prima volta. Si può ovviare a questo inconveniente inserendo come prima istruzione, nella funzione `gestoreKill()`, di nuovo una chiamata a `signal()`:

```
#include <signal.h>
...
void gestoreKill()
{
   /* procedura eseguita quando viene ricevuto un segnale di kill */
   signal(SIGKILL,gestoreKill);
   ...
}
```

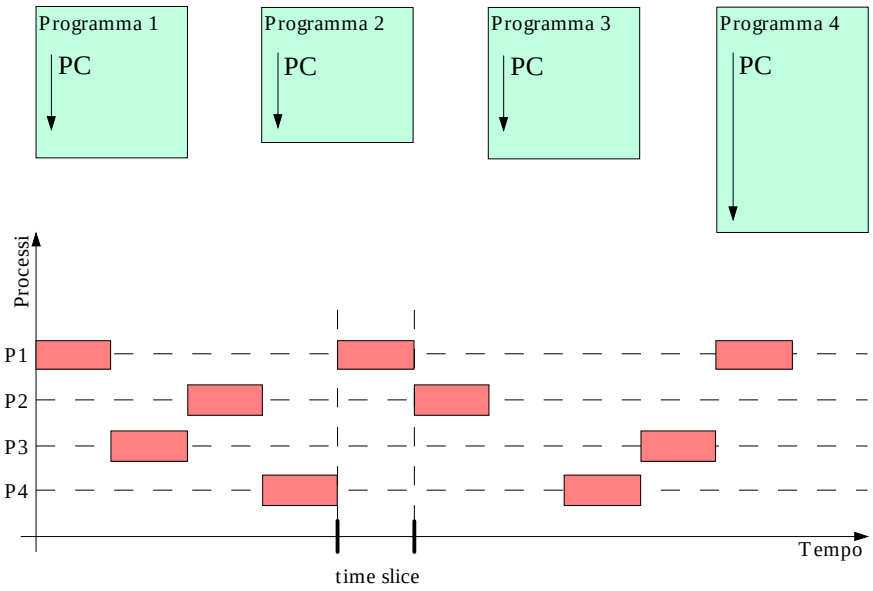
```
...
int main(void)
{
   /* Assegna al segnale SIGKILL la funzione gestoreKill */
   signal(SIGKILL, gestoreKill);
   ...
}
```

Vi è comunque un difetto non eliminabile completamente: il processo termina (azione di default) se interviene un SIGKILL appena prima della chiamata a `signal()` nella funzione `gestoreKill()`. I segnali in arrivo possono essere anche ignorati. Per ignorare un segnale si può usare `signal(segnale,SIG_IGN);` Per ripristinare il segnale, gli si può assegnare una funzione particolare (come visto sopra) oppure si può ripristinare il comportamento di default: `signal(segnale,SIG_DFL);`
`SIG_IGN` e SIG_DFL sono due particolari funzioni. La prima compie un'azione nulla. La seconda invece contiene le azioni di default per i segnali. Il segnale `SIGKILL` non può essere ignorato e di default stabilisce la fine per un processo.

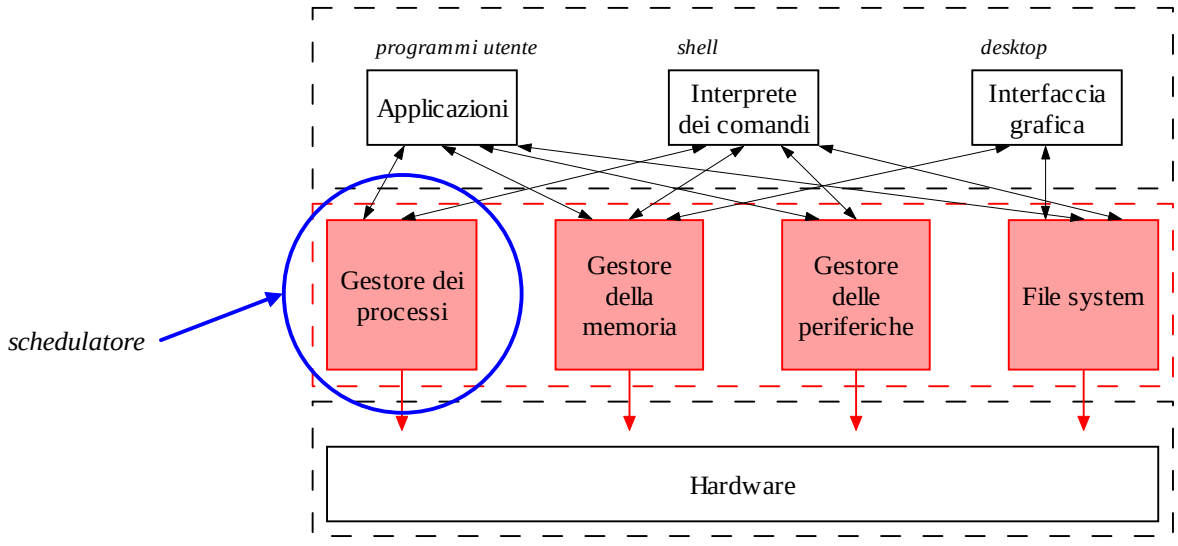
Organizzazione di un S.O. monolitico

Il concetto di processo influisce fortemente sulla struttura e sulla organizzazione del sistema operativo. Tutto il software eseguibile sul calcolatore, compreso anche lo stesso sistema operativo, può essere organizzato in un certo numero di processi. Il file eseguibile di un programma utente contiene al suo interno tutte le informazioni necessarie affinchè il programma utente, residente su memoria di massa, possa essere mandato in esecuzione. Per processo intendiamo quindi un programma in esecuzione (quindi la sua immagine si trova già in memoria centrale) con i propri valori del program counter (PC), registri e variabili locali, lista di file aperti ed altre informazioni. Ad ogni processo viene poi assegnata una CPU virtuale. In realtà la risorsa CPU (risorsa condivisa) è unica e viene per questo motivo assegnata in continuazione da un processo ad un altro. Anche il program counter è unico e nella realtà ogni processo è dotato di un proprio program counter detto per questo motivo program counter logico. Il vero program counter è anche detto program counter fisico. Pertanto, ogni processo in esecuzione implica un aggiornamento del program counter fisico. Il program counter logico del programma da mandare in esecuzione viene allora caricato sul program counter fisico mentre lo stato del precedente program counter fisico (e quindi appartenente al vecchio processo in esecuzione) viene salvato sul program counter logico del vecchio programma

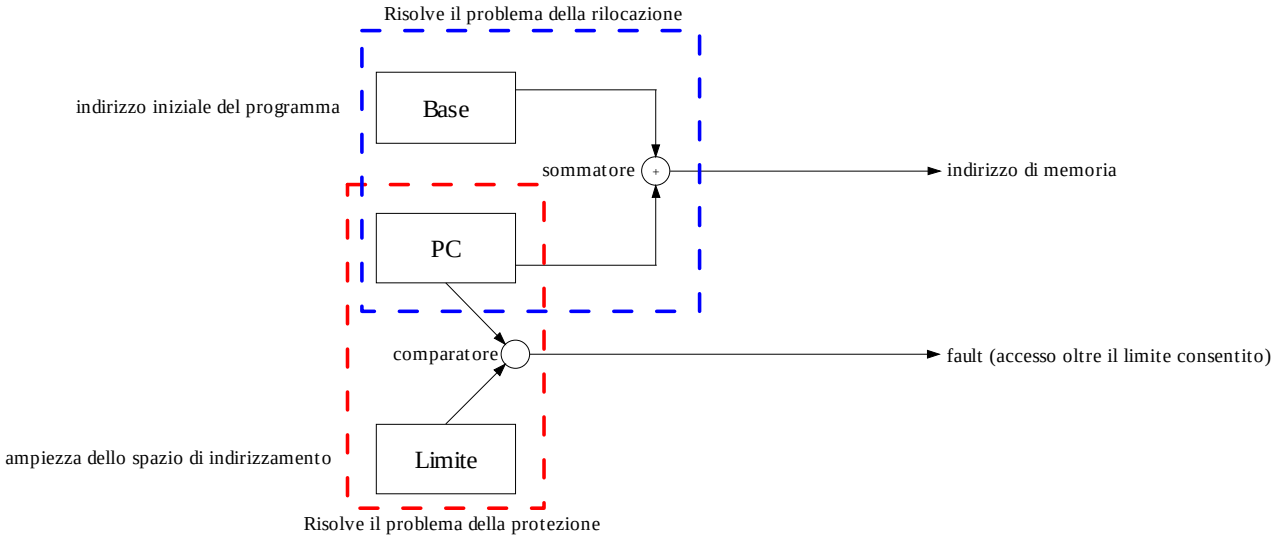
in esecuzione. Quindi, se l'impressione data all'utente è quella in cui si trovano più programmi in esecuzione (multiprogrammazione), nella realtà, in un fissato istante di tempo, è in esecuzione un solo programma alla volta.



Nella figura sopra viene appunto mostrato quanto appena detto: un solo processo è in esecuzione in un quanto di tempo (time slice). La vera multiprogrammazione avviene ad esempio nei sistemi multiprocessori. Il modello a processi sequenziali (quello in figura) è un modello concettuale che permette di esprimere in modo semplice più attività contemporanee. L'attività è confinata all'interno del programma utente ed è descritta dal program counter logico che guida il flusso dati del programma.
Se dunque i processi sono eseguiti in maniera sequenziale occorre un meccanismo in grado di mettere in pausa l'esecuzione di un processo allo scadere del time slice ed assegnare, quindi, la CPU ad un nuovo processo. Il gestore dei processi, talvolta detto schedulatore, si preoccupa di fare quanto appena detto. La distribuzione del processore ai vari processi in attesa deve essere trasparente e bilanciata. Non deve cioè privilegiare alcun processo e trascurarne altri. Nella realtà alcuni processi, ad esempio quelli dedicati al sistema operativo, possono avere una priorità più alta di un processo utente. Ciò è indispensabile per garantire il normale funzionamento del sistema operativo che altrimenti fallirebbe. Lo schedulatore segna, dunque, le attività dei processi attribuendo a quest'ultimi un adeguata priorità.



Se il program counter fisico è aggiornato al program counter logico del programma in esecuzione, la memoria centrale (anch'essa risorsa condivisa) è invece suddivisa tra i vari processi cosicchè lo spazio di indirizzamento è in parte assegnato a processi dedicati al sistema operativo ed in parte assegnato ai processi utenti. La condivisione della memoria principale innesca diverse problematiche, una di queste è ad esempio il livello di protezione che occorre offrire ai processi residenti su memoria principale affinchè un altro processo, sufficientemente malizioso, non acceda ad indirizzi di memoria al di fuori dell'indirizzamento a lui assegnato. Un altro problema è quello della rilocazione: il compilatore assume che lo spazio di indirizzamento del programma inizi da 0 mentre nella realtà il pogramma in memoria occupa generalmente un indirizzo diverso da quello iniziale. L'indirizzo iniziale del programma allocato in memoria varia ogni volta che questo è caricato in memoria (in funzione dei programmi che al momento sono già caricati in memoria).
Vi è quindi la necessità di un meccanismo che si faccia carico della rilocazione, ossia sommare un certo offset all'indirizzo base del programma utente. E' possibile attuare questa soluzione agendo direttamente sul software. In tal caso occorre avvertire il loader di tale esigenza (sarà cioè il loader che aggiungerà l'offset agli indirizzi delle istruzioni da mappare in memoria). Il problema della riservatezza dei dati in memoria è invece risolto agendo sull'hardware del sistema. La soluzione che vedremo permette in realtà di superare entrambi i problemi: la rilocazione e la protezione dei dati.



Il compilatore può assegnare alle istruzioni solo indirizzi relativi, a questi poi, quando il loader proietta in memoria l'immagine del processo, va aggiunto l'offset cosicchè in uscita dal nodo sommatore si ricava il reale indirizzo di memoria.
Nel registro PC va caricato ovviamente il program counter del programma da eseguire. L'accesso in memoria ad un dato oppure ad una istruzione del programma in esecuzione avviene sommando l'indirizzo contenuto nel program counter a quello invece contenuto nel registro Base. Il registro base contiene quindi l'offset da sommare all'indirizzo logico. Questo permette di risolvere in maniera dinamica il problema della rilocazione. Nel registro Limite è invece riportato l'intervallo dello spazio di indirizzamento associato al programma in esecuzione: il confronto dell'indirizzo all'interno del registro program counter con l'ampiezza dell'indirizzamento del programma permette di stabilire se l'accesso in memoria oltrepassa il limite consentito. L'istruzione che oltrepassa l'intervallo degli indirizzi indicato nel registro Limite viene quindi impedita.
In alcuni sistemi operativi, quando il sistema è avviato (fase di inizializzazione), si possono già trovare in memoria tutti i processi che il sistema operativo, poi, una volta caricato anch'esso in memoria, dovrà gestire. Tale scenario si manifesta in alcuni sistemi dedicati come ad esempio nei controller di una centralina elettronica (lì in memoria i processi risiedono in maniera permanente ed il sistema operativo, una volta avviato li esegue dapprima uno per uno effettuando un opportuno test). Nei calcolatori, invece, all'avvio del sistema, la memoria è vuota ed i processi sono generalmente creati attraverso il meccanismo delle chiamate di sistema (ad esempio attraverso la

primitiva `fork()` se siamo in presenza di un sistema UNIX oppure mediante primitiva `CreateProcess()` se siamo in presenza di sistema Windows). In altri casi, invece, quando il sistema è già avviato, un processo può essere creato da un altro processo in esecuzione come ad esempio fa la Shell di Linux quando interpreta un comando. In tal caso il processo viene generato mediante primitiva `exec()`. In altri casi è invece l'utente che decide di creare un processo, ad esempio eseguendo un programma da lui compilato. Riassumendo, gli eventi che provocano la creazione di un processo sono:

- inizializzazione del sistema;
- esecuzione di una chiamata di sistema fatta da un processo in esecuzione per far eseguire un determinato compito ad un altro processo;
- esecuzione di una chiamata di sistema fatta dall'utente;
- l'inizio di un job batch che attiva dei processi ad un ora prestabilita del giorno;

La sola creazione di un processo non è utile se il processo stesso poi non fornisce alcun risultato o elaborato. Dopo che il processo è stato creato esso comincia ad eseguie il compito che gli è stato assegnato, qualunque esso sia. Ad ogni modo, niente dura per sempre, nemmeno i processi che prima o poi termineranno. La terminazione di un processo può essere volontaria (se decisa dal processo stesso) oppure involontaria (se decisa ad esempio dal processo padre o da un evento asincrono al programma). Ci sono essenzialmente quattro eventi che causano la terminazione di un processo:

- la causa più ovvia è la terminazione volontaria del processo che eseguendo l'ultima istruzione a lui assegnata termina (terminazione volontaria);
- un altra causa di terminazione è quella che si ha in presenza di un errore che impedisce al programma di continuare. Tale errore è ad ogni modo rilevato dal programma che decide esso stesso di terminare l'esecuzione. Un gestore dell'errore si fa carico di trattare l'evento, solitamente si provvede ad avvisare l'utente attraverso la stampa di un messaggio (terminazione volontaria);
- una causa di terminazione più seria è quella che si stabilisce in presenza di un errore che però non viene rilevato. L'errore impedisce al programma di funzionare correttamente al punto che esso prima o poi verrà terminato dal sistema operativo. Si tratta essenzialmente di errori di programmazione (divisione per zero, riferimenti a indirizzi di memoria non esatti, istruzioni illegali, etc...) talvolta detti bachi (terminazione involontaria);
- un processo esegue una system call ed indica al sistema operativo quale processo uccidere. Il processo che decide di terminare un altro processo deve ovviamente avere i permessi necessari. In UNIX la system call `kill()` può inviare un segnale di terminazione ad un processo. In ambiente Windows la chiamata `TerminateProcess()` fa altrettanto;

Ad ogni modo, un processo non rimane indefinitivamente in esecuzione. Esso solitamente assume diversi stati ed è per questo motivo interrotto più volte nel corso del suo ciclo di vita. L'interruzione momentanea di un processo può essere causata da:

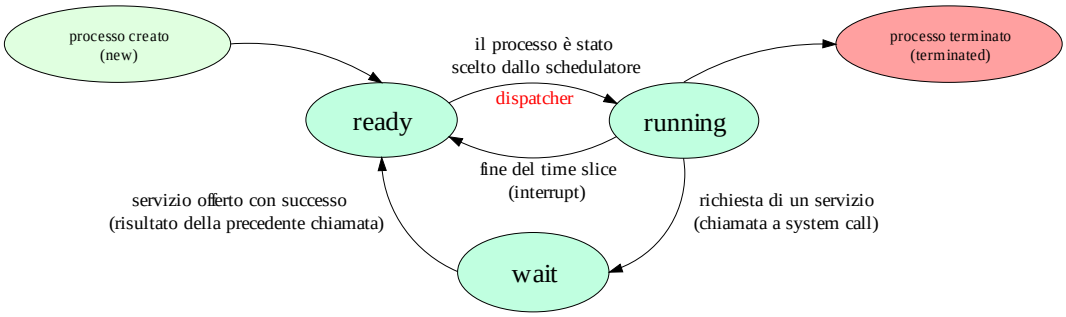
- un segnale di clock che notifica la fine del time slice;
- una chiamata ad una system call fatta dal processo in esecuzione per richiedere un determinato servizio al sistema operativo;

Stati di un processo
In base a quanto finora detto sono già emersi, nel corso della trattazione fin qui fatta, i possibili stati di un processo. Nell'ipotesi di più processi caricati in memoria centrale è possibile attribuire ad ognuno di essi uno dei seguenti stati:

- processo in esecuzione (stato running);
- processo momentaneamente bloccato (stato waiting);

- processo pronto (stato ready);

Un processo in esecuzione (stato running) su di un sistema è l'unico a disporre della CPU, esso pertanto può far avanzare i propri calcoli svolgendoli all'interno del time slice assegnatogli. Tale stato presuppone l'esistenza in memoria dell'immagina del processo. Lo stato waiting è invece dato a tutti quei processi che stanno attendendo l'esito di una richiesta precedentemente lanciata attraverso il meccanismo delle system calls (il processo chiede un servizio al sistema operativo e per questo motivo attende una risposta da una system call). Infine, un processo si dice essere pronto (stato ready) quando attende l'assegnazione del processore (la sua immagine è quindi già in memoria, manca solo il processore per eseguirlo).



Nella realtà sono presenti altri due stati. Un processo può infatti essere stato appena creato, in tal caso esso può allora permanere in questo stato finchè non sono allocate tutte le risorse sufficienti a renderlo schedulabile, cioè idoneo ad una eventuale scelta (è come se il processo creato non si candidasse ad una sua possibile scelta). Talvolta questo stato viene conferito ad un processo per limitare il livello di multiprogrammazione (per non sovrapporre cioè un ulteriore processo a quelli che già litigano fra di loro per l'ascesa alla CPU, il processo creato subentra non appena un processo termina). Il processo segue il suo flusso descritto nel proprio program counter logico e finquando ci saranno ulteriori istruzioni da portare a termine esso percorrerà ciclicamente gli stati ready e waiting. Qualche volta si troverà anche ad essere in stato waiting ma ciò sarà sempre compatibile con le istruzioni assegnate al processo. Una volta eseguita l'ultima istruzione un processo termina portandosi, come si vede dalla figura, in uno stato terminated. Tale stato è raggiungibile anche a causa di una richiesta di `kill()` eseguita da un processo nei confronti del processo in esecuzione (in tal caso il processo termina, involonariamente, prima di raggiungere l'ultima istruzione).
Lo schedulatore è quel compenente del sistema operativo che si occupa di scegliere quale processo mandare in esecuzione (stato running) fra tutti quelli in stato ready. La scelta avviena in base a delle opportune politiche di schedulazione che il sistema operativo si prefigge. In altre parole esso adotterà un determinato algoritmo per stabilire quale processo mandare in esecuzione. Il compito dello schedulatore consiste nella sola scelta (in esso è sintetizzata la logica da applicare alla schedulazione), il dispatcher infatti è un altra componente del sistema operativo che ha il compito di attuare la transizione dal processo corrente al nuovo processo scelto dallo schedulatore. Tale operazione, talvolta ricordata come context switch, inizia salvando nel program counter logico tutte le informazioni necessarie a conservare lo stato del processo appena fermato. Per rendere poi eseguibile il processo scelto dallo schedulatore, l'operazione di context switch si completa caricando sul program counter fisico il program counter logico del nuovo processo scelto
La trattazione fin qui portata avanti è fortemente basata sul modello a processi sequenziali che abbiamo introdotto per agevolare la comprensione sui reali meccanismi di funzionamento di un moderno sistema operativo. Affinchè ciò sia possibile (eseguire cioè un processo alla volta dando l'illusione di eseguirne tanti in parallelo) è necessario implementare una struttura utile a segnare tutte le attività, gli stati e le informazioni idonee alla gestione dei processi (program counter, stack pointer, lista di file aperti, stato del processo, uso della CPU in percentuale, etc...). Tali informazioni sono tenute in una apposita tabella detta tabella dei processi. Essa consiste di diversi record che vanno quindi a descrivere attraverso una struttura complessa le informazioni dette prime.

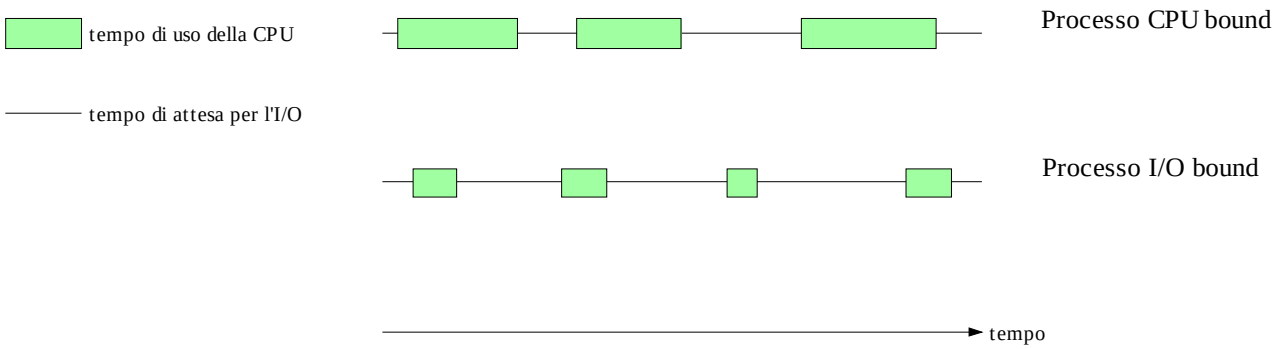
Ogni singola struttura della tabbella dei processi (process table) viene anche detto blocco di controllo del processo (process control block oppure più brevemente PCB).
I tipi ed il numero di campi inseriti in una PCB dipendono dal sistema operativo, ad ogni modo saranno presenti un certo numero di campi dedicati alla gestione del processo, un certo numero di campi dedicati alla gestione della memoria ed infine un certo numero di campi viene destinata alla gestione dei file. Alcuni campi di una tabella dei processi potrebbero ad esempio essere i seguenti:

Gestione dei processi	*Gestione della memoria*	*Gestione dei file*
Registri	Puntatore allo stack	Directory radice
Program counter	Puntatore a dati	Directory di lavoro
Stack pointer	Puntatore al testo	Descrittore dei file
Parola di stato del processo		Lista di file aperti
Stato del processo		
Priorità		
Identificatore del processo		
Processo genitore		
Parametri per la schedulazione		
Segnali		
Tempo di CPU usato		
Tempo di CPU usato dai figli		

Il cambio di contesto che un dispatcher rende operativo è una operazione generalmente molto costosa, nei sistemi reali questo costo si aggira attorno ai diversi millisecondi, ne deriva quindi che un cambio di contesto non deve avvenire con troppa frequenza se non si vuole penalizzare la prestazione dell'intero sistema che altrimenti impegnerebbe gran parte del tempo ad operazioni di context switch. Nei sistemi UNIX ad esempio avviene un contet switch ogni 20 – 200ms.

La schedulazione dei processi
La multiprogrammazione, ossia la tecnica che prevede l'esecuzione in contemporanea di più processi, genera notevoli problemi di contese nei confronti delle risorse di sistema (processore, memoria e periferiche). Quando più processi, infatti, si trovano in stato ready lo schedulatore sceglie il successivo processo da eseguire. Il vero motore del sistema operativo, in questo caso, non è il solo schedulatore ma come abbiamo già visto è necessaria anche l'azione congiunta del dispatcher affinchè la scelta operata dallo schedulatore diventi effettiva. Lo schedulatore implementa al suo interno una determinata politica di scheduling, questa caretterizza inevitabilmente le prestazioni percepite dagli utenti esterni e l'efficienza nell'utilizzo delle risorse di sistema. Tale politica è quindi descritta da un appropriato algoritmo di schedulazione.
La schedulazione può perseguire obiettivi diversi da sistema a sistema. Nei sistemi batch ad esempio (quelli cioè caratterizzati da un infornata di job da eseguire e visti dal sistema come un unico job su nastro) l'algoritmo di schedulazione non serviva a molto. Tutti i job andavano comunque eseguiti e presentati in uscita su di un unico nastro nello stesso istante di tempo, per questo motivo l'algoritmo di schedulazione consisteva semplicemente nell'esecuzione del successivo job in lettura. Ordinare i vari job in base ad un criterio che ne avrebbe discriminato l'ordine di esecuzione aggiungeva un inutile ritardo (in seguito vedremo che questa cosa non è proprio vera).
Con l'avvento dei sistemi timesharing (più utenti condividevano le risorse ed erano per questo motivo in attesa) il problema della schedulazione ha iniziato a farsi sentire. Un buon algoritmo di schedulazione può in questo caso soddisfare un utente e caratterizzare le prestazioni di un sistema.
Nei personal computer odierni la schedulazione è ancora importante ma non come lo era una volta. Ciò è dovuto da un lato alle caratteristiche delle moderne CPU, di sicuro più veloci e performanti. Inoltre, in un personal computer è ragionevole immaginare uno o pochi processi utenti attivi. Se infatti l'utente è unico questo molto probabilmente sarà preso dall'interazione con uno solo dei programmi utente e se pur esistono altri processi in background è comunque buona norma soddisfare prima le richieste del processo in esecuzione dall'utente. In questo modo si da all'utente la sensazione di un sistema più veloce.
Nei sistemi interattivi invece è importante dare all'utente l'impressione di un sistema assai veloce a

fornire risposte specie se questo si serve di una grafica per l'interattività con l'utente. Immaginate ad esempio cosa possa pensare un utente poco pratico di sistemi operativi nel vedere un messaggio che conferma l'avvenuta ricezione di una mail, precedentemente mandata, giungere con un certo ritardo. Il ritardo potrebbe ad esempio essere causato dal sistema operativo che in quel momento è preso ad aggiornare una finestra appena modificata dallo stesso utente. L'utente pensieroso di leggere la conferma di avvenuta ricezione potrebbe pensare ad un sistema lento quando invece la conferma al suo messaggio è già stata ricevuta e sarà aggiornata sul video al più presto.
Prima di procedere ad analizzare le regole di schedulazione è opportuno effettuare una suddivisione dei processi. Alcuni processi possono spendere la maggior parte del tempo in calcoli, essi si diranno processi CPU bound. Altri processi invece spenderanno la maggior parte del loro tempo in brevi periodi di elaborazione fra due richieste di I/O, tali processi si diranno I/O bound.



Esistono essenzialmente 5 momenti o motivi per schedulare, momenti in cui occorre prendere una decisione circa il processo da mandare in esecuzione:

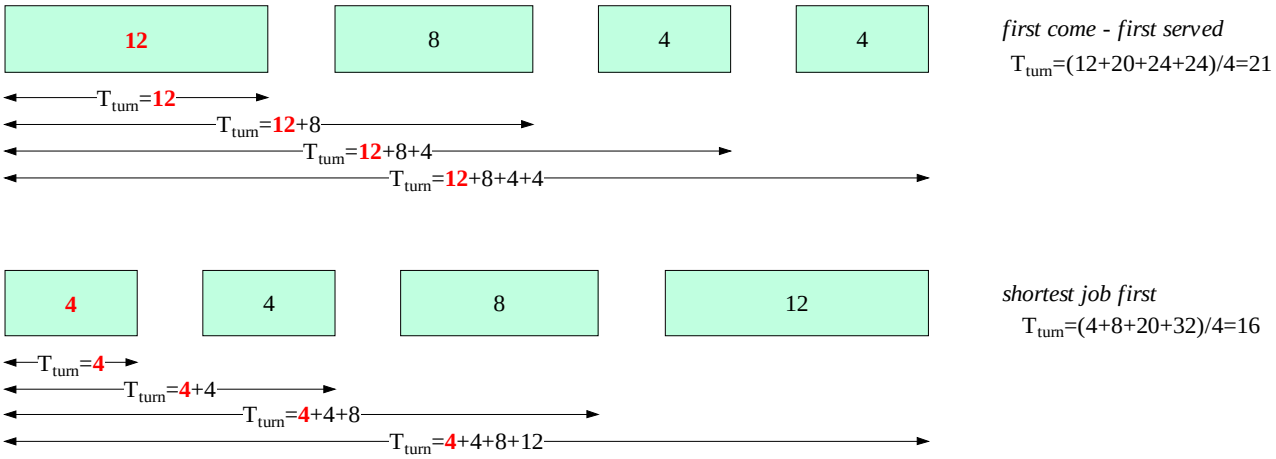
1. Nel momento in cui si crea un processo deve essere presa una decisione di schedulazione nei confronti del processo padre o del processo figlio oppure di altri processi che si trovano in stato ready;
2. Nell'istante in cui un processo termina eseguendo l'ultima istruzione occorre decidere quale processo, fra tutti quelli in stato ready, deve essere eseguito. Se nessun processo esiste sul sistema viene eseguito dal sistema operativo un processo inattivo;
3. Quando un processo in esecuzione si blocca su una operazione di I/O, su un semaforo o per altra ragione (qualsia altra operazione sospensiva) lo schedulatore interviene mandando in esecuzione un altro processo. In alcuni casi il blocco temporaneo del processo potrebbe suggerire allo schedulatore il successivo processo da eseguire. Se ad esempio il processo A si mette in attesa del processo B, la successiva esecuzione del processo B potrebbe sbloccare lo stato del processo A. Purtroppo lo schedulatore non conosce le dipendenza tra processi;
4. Quando una interruzione di I/O di un dispositivo notifica la sua disponibilità un processo precedentemente bloccato dall'attesa di tale dispositivo può essere rimesso in esecuzione oppure può essere eseguito un altro processo fra quelli al momento pronti;
5. Quando giunge l'interruzione hardware del clock di sistema il processo in esecuzione viene messo in pausa ed un diverso processo è mandato in esecuzione (in alcuni sistemi il processo è messo in pausa solo dopo l'arrivo di k segnali di clock).

Gli algoritmi di schedulazione possono essere suddivisi in due categorie in base a come questi trattano le interruzioni di clock. Un algoritmo di schedulazione si dice senza prerilascio (non preemptive) se lo schedulatore una volta scelto il processo da mandare in esecuzione ne attende la terminazione volontaria, anche se il processo è in esecuzione per diverse ore! Il segnale di clock, se presente, è ignorato dal processo oppure, se vogliamo, in caso di interruzione viene sempre eseguito il processo che era in esecuzione.
Un algoritmo di schedulazione si dice invece con prerilascio (preemptive) se il processo è mandato in esecuzione per un fissato tempo (time slice). Se allo scadere del time slice il processo è ancora in esecuzione lo schedulatore interviene sospendendone l'attività ed eseguendo un diverso processo.

Osservare che qualora su di un sistema è assente un segnale di clock l'unica strategia di schedulazione che è possibile mettere in atto è quella senza prerilascio.
L'algoritmo di schedulazione è diverso da sistema a sistema, esistono tuttavia tre ambienti che vale la pena distinguere: sistemi batch, sistemi interattivi e sistemi real time. Ognuno di questi si prefigge particolari obiettivi e richiede per questo motivo un particolare comportamento allo schedulatore. Tuttavia esistono alcune caratteristiche comuni a tutti i sistemi prima citati. Un obiettivo comune è così l'equità: garantire ad ogni processo una aliquota di CPU. Ciò scaturisce da una banale osservazione: due processi simili devono necessariamente ricevere lo stesso trattamento se non si vuole favorie un processo piuttosto che un altro. Non sarebbe corretto assegnare più tempo di CPU ad un processo, certamente ciò non passerebbe inosservato agli utenti più attenti che inizierebbero a protestare nei confronti dello schedulatore. Assicurata quindi l'equità tra tutti i processi occorre poi garantire a questi il rispetto della politica di schedulazione prefissata: lo schedulatore (che ricordiamo essere la parte del sistema operativo che effettua la scelta del processo da mandare in esecuzione) deve verificare che la politica da lui indicata venga messa in atto dal dispatcher. Infine, un altro obiettivo generico è il bilanciamento delle risorse: tenere impegnate tutte le risorse di sistema comporta l'esecuzione di più processi.
Detto ciò occorre adesso caratterizare le pretese di ciascun ambiente detto prima (sistemi batch, sistemi interattivi e sistemi real time). Abbiamo precedentemente detto che in un sistema batch tutto il lavoro eseguito è in relatà composto da più unità di lavoro (i cosiddetti job) la cui esecuzione produce in unico output i risultati dei singoli job. Abbiamo poi detto che in realtà lo schedulatore di un sistema batch ha un compito abbastanza semplice: eseguire ogni volta che termina un job quello immediatamente successivo. Tuttavia i gestori dei grossi sistemi batch desiderano sempre impegnare al massimo la costosissima CPU, essi valuatno la prestazione di uno schedulatore in base a tre parametri: il troughput, il tempo di turnaround e l'uso della CPU. Il troughput è il numero di job eseguiti in un ora, più è alto questo parametro e più sarà contento il gestore del sistema (nella sua testa dirà: "meglio portare a termine 100 job in un ora che terminarne 50..."). Il tempo di turnaround è la durata media di un job ossia il tempo che mediamente l'utente deve aspettare prima che il job venga portato a termine. E' preferibile avere un basso tempo di turnaround, ciò significherebbe attendere un minor tempo prima di poter richiedere un nuovo job. Un algoritmo di schedulazione che massimizza il throughput non è detto che minimizza il tempo di turnaround, infatti: se sul sistema si trovano diversi job, alcuni di breve durata ed altri invece più lunghi, lo schedulatore potrebbe decidere di eseguire per prima i job più piccoli e successivamente quelli più grandi. Se accade quanto appena detto si avrebbe indubbiamente un elevato numero di troughput (più job eseguiti in un ora) a scapito del tempo di turnaround che potrebbe addirittura tendere ad infinito se al sistema vengono sottoposti con una certa frequenza job di breve durata (in tal caso infatti lo schedulatore, in base alla propria politica di schedulazione, manderebbe in esecuzione sempre i job più piccoli non eseguendo mai quelli più grandi!). Un altro parametro usato per misurare le prestazioni di uno schedulatore per sistemi batch si basa sulla percentuale di utilizzo della CPU, si tratta di misurare in che percentuale di tempo la CPU viene usata rispetto al tempo di vita del sistema o alla durata della sessione.
I sistemi interattivi danno più importanza ai tempi di risposta del sistema che sono generalmente correlati in maniera proporzionale alle aspettative degli utenti. Per tempo di risposta si intende l'intervallo di tempo che intercorre tra l'istante di tempo che ha inizio in corrispondenza del comando inviato dall'utente all'istante di tempo che coincide con la notifica dell'evento richiesto o provocato dall'utente. E' fondamentale avere un tempo di risposta quanto più piccolo possibile, l'utente ne sarà sicuramente soddisfatto poichè avrà l'impressione di lavorare con un sistema estremamente veloce. Per questo motivo, su un sistema interattivo, tutte le attività collegate a processi interattivi hanno la precedenza rispetto ai processi collegati ad attività in background. Il tempo di risposta è ad ogni modo proporzionale all'attività richiesta cosicchè se l'utente immagina di aver ordinato un'attività piuttosto pesante al sistema esso stesso avrà pazienza nell'attendere un risultato. Se però il comando impartito può sembrare stupido agli occhi dell'utente allora quest'ultimo si aspetterà in breve tempo una risposta. In alcuni casi lo schedulatore non può fare gran che in altri casi invece, agendo sull'ordine dei processi da mandare in esecuzione, è possibile migliorare il tempo di risposta del sistema.

I sistemi real time sono alquanto complessi ed hanno caratteristiche diverse dai sistemi interattivi e molto spesso sono tenuti a rispettare determinate QoS (quality of service). Il rispetto di queste QoS avviene solitamente osservando delle precise scadenze temporali ciò costituisce per questo motivo l'obiettivo principale di un sistema real time (in alcuni casi l'inossorvenza di una determinata scadenza temporale non è poi molto grave specie se essa si riferisce ad esempio ad un flusso video in ricezione, ovviamente più frame vengono persi e più si degrada la qualità del servizio).
Nei sistemi batch non ci sono utenti impazienti in attesa di risultati, in questi sistemi infatti si prevede una mole di calcolo abbastanza elevata e l'utente è consapevole della sua richiesta di calcolo. Pertanto nei sistemi batch viene usato un algoritmo di schedulazione senza prerilascio e generalmente con un lungo periodo di CPU per ogni processo (i processi sono cioè per la maggior parte CPU bound). Nei sistemi interattivi viene usato un algoritmo di schedulazione con prerilascio, in questo ambiente infatti è essenziale che un processo non usi sempre e da solo la CPU impedendo ad altri processi di andare in esecuzione. Pertanto, con la tecnica del prerilascio viene assegnato prima o poi un slot temporale ad ogni processo. Nei sistemi real time si hanno scadenze più stringenti ed i processi sanno che non possono occupare a lungo la CPU (altrimenti come farebbero a rispettare le QoS), in questi ambienti viene adottato maggiormente la tecnica senza prerilascio (stranamente).

Schedulazione nei sistemi batch
Un algoritmo molto semplice da programmare e da capire è il *first come – first served*. Si tratta di un algoritmo senza prerilascio (lo schedulatore interviene solo a seguito di una interruzione volontaria del processo in esecuzione) che serve il primo processo arrivato. I sistemi siffatti adottano un unica coda per l'attesa in cui possono ospitare un determinato numero di processi, il primo processo che arriva, percorre a vuoto la coda di attesa trovandola libera, ed è anche il processo che per prima viene servito. Quando arrivano altri job questi si mettono in attesa e non appena il processo in esecuzione si blocca lo schedulatore interviene mandando in esecuzione il primo processo in coda mentre il processo bloccato è collocato in fondo alla coda (come se fosse un job appena arrivato). L'algoritmo si dimostra essere sufficientemente equo poichè da a tutti i processi l'opprtunità di giungere, dopo aver percorsa la coda, alla CPU.
La schedulazione *shortest job first* prevede, invece, dapprima l'esecuzione dei job più piccoli (i cosiddetti CPU bound) e successivamente quelli più lunghi (I/O bound). Purtroppo per applicare lo shortest job first è necessario conoscere la durata dei processi affinchè si possa poi decidere quali di questi sia effettivamente il processo più piccolo in termini di durata. Ad ogni modo si dimostra che operando in questo modo si minimizza il tempo medio di turnaround. Si supponga ad esempio di avere quattro job A, B ,C e D, con tempi di esecuzione rispettivamente di 12, 8, 4 e 4 minuti. Se lo schedulatore applica l'algoritmo first come – first served il tempo di turnaround verrebbe ad essere di 21 minuti: (12+20+24+28)/4. Se lo schedulatore si serve dell'algoritmo shortest job first il tempo di turnaround è invece di 16 minuti: (4+8+20+32)/4. Più in generale, nell'ipotesi di quattro job, è possibile esprimere il tempo medio di turnaround come: (4a+3b+2c+d)/4. Questo perchè il tempo impiegato dal primo processo influenzerà inevitabilmente anche la durata dei successivi tempi di turnaround, la durata dell'ultimo processo influenza il proprio tempo di turnaround. Per questo motivo il primo processo da mandare in esecuzione deve essere di breve durata (in questo modo si minimizzano anche le successive sommatorie) mentre l'ultimo ad essere eseguito deve necessariamente essere il processo più lungo.



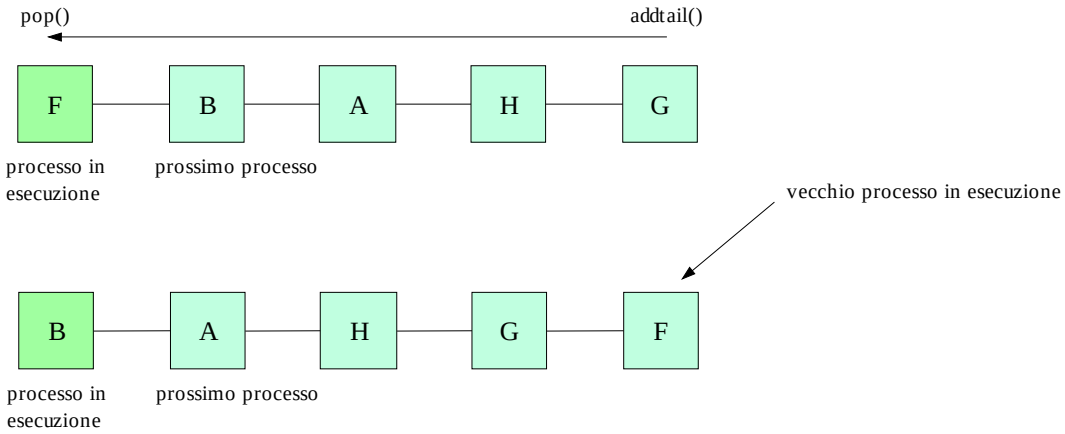
Qualora i job non sono tutti disponibili presso il sistema che sta applicando la schedulazione shortest job first il tempo di turnaround non viene affatto minimizzato. Infatti, nell'ipotesi di cinque processi A, B, C, D e F con tempi di esecuzione rispettivamente di 4, 2, 1, 2, 1 minuti e tempi di arrivo di 0, 0, 3, 3, 3 minuti si ha che: inizialmente sono disponibili i soli processi A e B che ordinati in base alla loro durata, B precede A in virtù di una durata temporale inferiore, contribuiscono ad un tempo di turnaround che è (inizialmente) di (4+6)/2=4 minuti. Qualche istante successivo si rendono disponibili, presso il sistema, altri job ed il tempo di turnaround diventa di ((10+1)+(10+1+1)+(10+1+1+2))/4=12.3 minuti (se i processi erano invece disponibili contemporaneamente presso il sistema essi sarebbero stato lanciati nell'ordine CEBDA con un tempo di turnaround di 7.5 minuti).

Schedulazione nei sistemi interattivi

Così come nei sistemi batch esiste la schedulazione first come – first served, per i sistemi interattivi si parla di schedulazione *round robin*. E' un tipo di schedulazione con prerilascio, ciò significa che ad ogni processo è assegnato un quanto di tempo durante il quale il processo è in esecuzione. Lo schedulatore ha il compito di tenere in una lista tutti i processi in stato ready cosicchè quando allo scadere del quanto di tempo viene forzato il cambio di contesto, il processo in esecuzione è messo in stato wait ed è collocato in fondo alla lista (come se fosse un processo nuovo) mentre il successivo processo nella lista è ora mandato in esecuzione (sempre per un quanto di tempo di CPU).

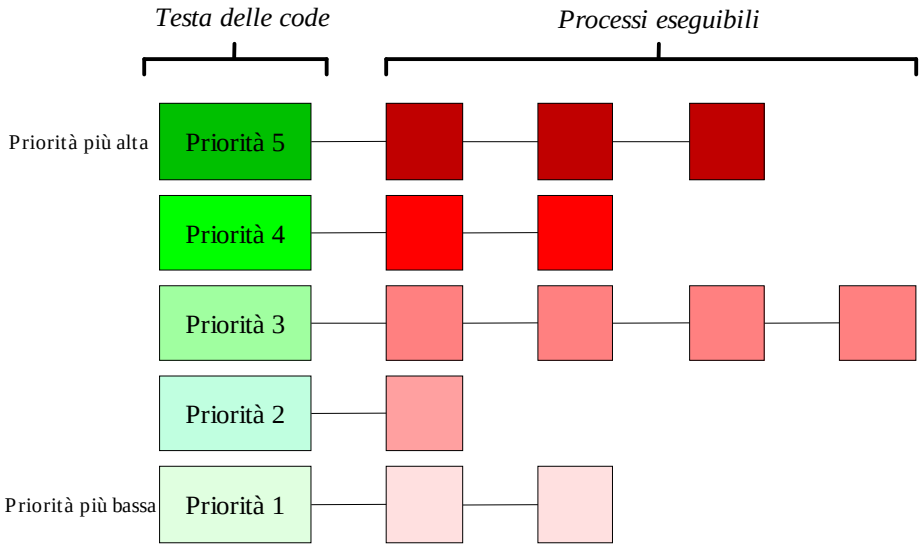
La lista è tipicamente realizzata con puntatori (lista collegata), i nuovi processi sono aggiunti in fondo alla coda (con operazioni tipiche di `addtail()`), i processi in cima alla lista sono estratti mediante operazioni di `pop()` ed aggiunti in coda, quando il quanto di tempo è scaduto, con operazioni di `addtail()`. Questo continua aggiornamento della lista ha l'effetto di far avanzare i processi che sono in stato ready nella coda, prima o poi ognuno avrà assegnato un quanto di tempo per cui l'algoritmo si rileva particolarmente equo. Una importante questione a riguardo della schedulazione round robin è la durata del quanto di tempo. Il cambio di contesto (context switch) richiede un preciso intervallo di tempo (in tale intervallo vengono svolte le operazioni di salvataggio dei registri ed il caricamento dei nuovi valori) e se ciò si ripete con una certa frequenza la CPU potrebbe essere impegnata per la maggior parte del tempo. Se ad esempio le operazioni di context switch richiedono 1ms ed il quanto di tempo è di 4ms, la CPU è per il 20% del tempo occupata in operazioni di context switch! Se, invece, il quanto di tempo è di 100ms la CPU sarà occupata per l'1% del tempo. La scelta del quanto di tempo non è casuale, essa va fatta considerando il tempo medio di turnaround. Infatti, se si decide per un quanto di tempo minore del tempo medio di turnaround, con molta probabilità la CPU vedrà scadere il quanto di tempo prima che il processi termini la sua computazione (in questo caso il processo è ricollocato in fondo alla lista poichè ha bisogno di ulteriori quanti di tempo). Pertanto, assegnare un quanto di tempo troppo breve provocherebbe troppi cambi di contesto nel sistema allungando la durata media di ogni processo (ogni processo dovrà ripercorrere di nuovo tutta la lista!). Una buona strategia è modellare il quanto di tempo in maniera che risulti appena un po più grande del tempo di turnaround, in questo

modo, con molta probabilità, la maggior parte dei processi riuscirà a completare le proprie operazio di calcolo e verranno per questo motivo rimossi dalla lista (terminazione del processo) mentre i processi che hanno ancora bisogno di ulteriori quanti di tempo percorreranno una lista più breve se nel frattempo il numero di processo in stato ready è rimasto invariato.
Notare che un quanto di tempo troppo grande potrebbe offrire la possibilità a tutti i processi di terminare rispettando le scadenze del quanto assegnato ma farebbe crescere inevitabilmente il tempo di percorrenza della lista, in altre parole aumenterebbero i tempi di risposta ed il sistema apparirebbe lento. Ad esempio, nell'ipotesi di un quanto di tempo lungo 100ms ed in presenza di 10ms si ha che: il primo processo in cima alla lista non attende alcun tempo, il secondo dovrà attendere 10ms (il quanto di tempo usato dal primo processo che lo precede), il terzo processo attenderà invece 20ms e così via... L'ultimo processo dovrà attendere i quanti di tempi di tutti i processi che lo precedono, per cui impiegarà 90ms nell'attesa e solo dopo 1s potrà eventualemente (nel caso non debba ripercorrere la lista) essere portato a termine. Un valore generalmente usato per il quanto di tempo si aggira nell'intorno compresso tra i 20ms ed i 50ms (con questi valori il processo di prima, l'ultimo della lista di dieci processi, verrà al massimo portato a termine nel giro di mezzo secondo).



Nella schedulazione (con prerilascio) di tipo round robin si è implicitamente assunto per tutti i processi in stato ready la stessa importanza, non a caso essi vengono collocati in unica coda di attes. Nei sistemi interattivi e multimediali lo scenario è leggermente diverso: ogni processo è contraddistinto da un livello di priorità. Esisteranno allora processi più importanti e processi meno importanti. La *schedulazione con priorità* manda ogni volta in esecuzione (quando cioè scade il quanto di tempo) il processo in stato ready con la priorità più alta.
Per evitare poi che i processi a priorità più alta impegnino sempre e da soli la CPU, lo schedulatore diminuisce la priorità del processo in esecuzione ad ogni tic di clock cosicchè quando la priorità del processo si abbassa fino a non essere quella al momento più alta è necessario effettuare un contet switch a favore del processo con priorità più alta. Un processo può allora sfruttare più quanti di tempo consecutivi se nel quanto di tempo successivo detiene ancora la cima della lista (lista dei processi in stato readu ordinata in base alle priorità. Data poi la natura dei processi (CPU bound oppure I/O bound) è ragionevole assegnare ai processi I/O bound una priorità più alta dei processi CPU bound, il motivo è qui spiegato: i processi I/O bound generalmente impegnano un dispositivo di I/O, si trovano già in memoria e richiedono poco tempo di CPU (se si tratta ad esempio di un processo per la lettura dal disco, molto probabilmente il processo ha in precedenza avviato il motore affinchè questo si portasse a regime, per questo motivo, se la lettura non avviene qualche istante dopo il processo si attiverà affinchè venga ripetuta l'intera procedura di boot della periferica e ciò implicherà un ulteriore attesa). Far aspettare un processo I/O significherebbe tenerlo inutilmente in memoria quando invece può essere eseguito e terminato (liberando memoria) in breve tempo. Altro motivo ancora più importante è che così facendo, eseguendo cioè immediatamente un processo I/O bound, si da all'utente una risposta immediata circa l'input o l'output che intende raccogliere. Per questo motivo, alcuni schedulatori con priorità assegnano ai processi una priorità che è funzione del quanto di tempo a disposizione e del quanto di tempo effettivamente utilizzato, $priorità = f_q / f_{used}$. Se

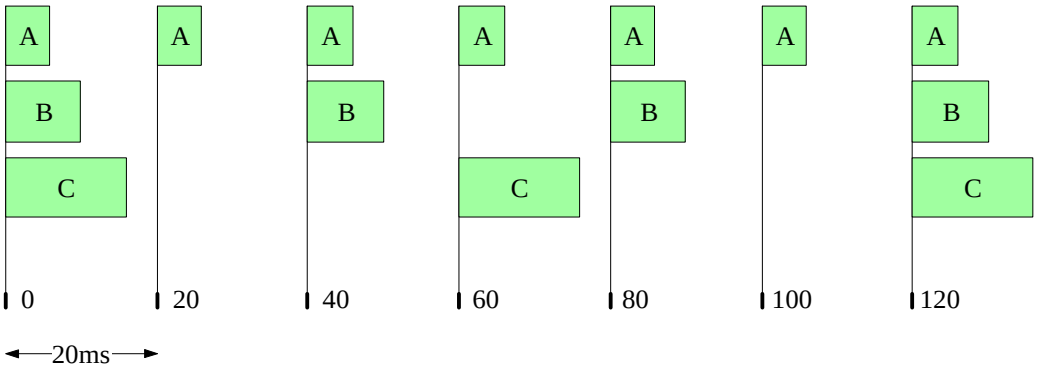
ad esempio un sistema interattivo sta usando un quanto di tempo di 50ms, ad un processo I/O bound che richiede 1ms dei 50ms a sua disposizione verrebbe assegnata una priorità pari a 50. Al contrario, invece, ad un processo CPU bound che sfrutta l'intero quanto di tempo verrebbe assegnata una priorità pari ad 1 (un processo che sfrutta tutto il quanto di tempo è solitamente penalizzato e la sua priorità è quasi sempre diminuità di un unità, se però questa è già la più piccola possibile al processo viene lasciata inalterata la priorità).
Talvolta la schedulazione con priorità può essere fatta su più code, si parla in questo caso di *schedulazione con code multiple*. Esiste una coda di attesa per ciascun livello di priorità previsto dallo schedulatore ed i processi, in base al livello di priorità, sono depositati in una delle suddette code. All'interno di ogni coda, poichè i processi in lista hanno qui la stessa priorità, si usa un algoritmo di schedulazione di tipo round robin (il processo in cima alla lista viene considerato prima degli altri). I meccanismi di aggiornamento dinamico delle priorità effettueranno, di tanto in tanto, delle promozioni cosicchè ad un processo viene data la possibilità di fare carriera. L'algoritmo di schedulazione con code multiple manderà in esecuzione il processo in stato ready in cima alla lista dei processi con priorità massima. Se la suddetta lista è vuota l'algoritmo ripiega sulla lista dei processi con priorità massima-1. Allo scadere del quanto di tempo lo schedulatore si riporta sulla lista a priorità massima (nella speranza di trovare un processo che nel frattempo è stato promosso) e se questa è ancora vuota ritorna sulla lista con priorità massima-1. Se anche la lista con priorità massima-1 è vuota lo schedulatore si porta sulla lista con priorità massima-2 e così via... L'aggiornamento delle priorità è fondamentale a garantire la proprietà di equità fra i processi, solo in questo modo i processi con priorità bassa hanno la possibilità di accrescere, nell'attesa, la propria priorità e per questo motivo avanzare di qualche classe (se ciò non si verifica si è soliti dire che è il processo è morto per starvation).



Esistono poi altri algoritmi di schedulazione per sistemi interattivi. Nello *shortest process next* ,ad esempio, viene stimato il comportamento del processo in base al precedente uso della CPU, il processo con il minor tempo stimato viene quindi mandato in esecuzione. Se la durata di un processo è T_0 e la successiva azione elaborativa ne richede invece T_1, la stima al processo è aggiornata al valore ottenuto dalla media pesata di questi due valori: $aT_0+(1-a)T_1$. La scelta del parametro *a* permette poi di decidere in che percentuale la stima da calcolare dovrà ricordare o dimenticare le precedenti elaborazioni. In alcuni sistemi interattivi si decide di adottare una strategia di schedulazione capace di garantire a tutti i processi una percentuale di CPU. Se ad esempio sul sistema sono in esecuzione *n* processi è molto corretto assegnare ad ogni processo l'*1/n* percento della CPU. A tale proposito vengono dapprima contati i processi in stato ready, quindi si procede all'esecuzione di un processo, almeno per un quanto di tempo di *1/n*. All'arrivo di nuovi processi, per meglio distribuire la CPU, si ripete il calcolo della quantita 1/n.

Schedulazione nei sistemi real time
Nei sistemi real time la reazione del sistema nei confronti di stimoli esterni (sensori che generano eventi) deve avvenire entro un certo istante di tempo. Alcuni sistemi real time hanno vincoli più leggeri, ad essi viene permesso di tanto in tanto di non rispettare le scadenze temporali prefissate e per questo motivo si dicono *soft real time*. Altri sistemi invece sono costretti a reagire rispettando le scadenze temporali che gli vengono chieste, questi sistemi si dicono *hard real time*. Gli eventi sottoposti al sistema possono essere periodici (un flusso video in riproduzione) oppure aperiodici (pilota automatico di un aereoplano). Ci sono diverse regole per stabilire se un algoritmo di schedulazione è più o meno possibile, se ad esempio ci sono *m* eventi periodici e se l'evento *i-esimo* arriva periodicamente ogni P_i sedondi richiedendo C_i secondi di CPU affinchè sià possibile gestire tutti gli eventi deve essere soddisfatta la condizione:

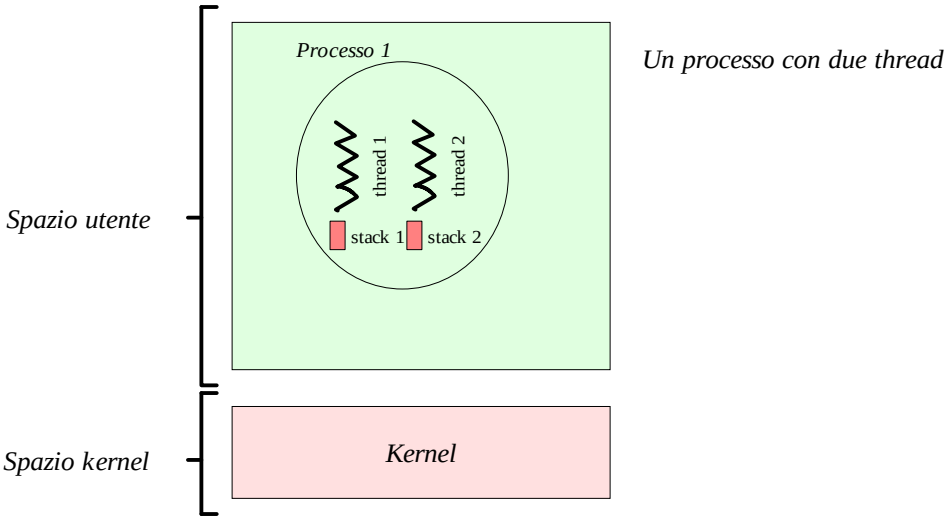
$$\sum_{i=1}^{m} \frac{C_i}{P_i} < 1$$



Nell'esempio, vengono ipotizzati tre processi: il processo A si ripete ogni 20ms e richiede 5ms di CPU, il processo B si ripete ogni 40ms e richiede 10ms di CPU, il processo C si ripete ogni 60ms e richiede 15ms. La sommatori sopra indicata rimane inferiore ad 1, è quindi possibile schedulare contemporaneamente i processi A,B e C. Il sistema real time è quindi schedulabile.

Thread e multithreading
Il concetto di multiprogrammazione visto come l'esecuzione parallela di più processi può essere ancora valido e quindi riciclato all'interno di un nuovo scenario che ci apprestiamo a descrivere, quello relativo cioè ai thread ed al multithreading. Abbiamo finora imparato a comprendere come sia possibile eseguire su di un sistema più processi utenti. Questi alternandosi danno all'utente l'impressione che il sistema sia in grado di svolgere parallelamente le computazioni di più processi. Nella realtà sappiamo bene che ciò non si verifica (in caso di sistema con un solo processore) e l'esecuzione dei processi è confinata all'interno di minuscoli quanti di tempo, tuttavia la multiprogrammazione accresce le prestazioni del sistema e stimola i ricercatori impegnati in questo settore. Vedremo a breve che il concetto di multiprogrammazione non è poi molto diverso dal multithreading. I primi calcolatori erano *singletasking*, data la scarsa potenza di calcolo delle CPU, eseguivano un solo job alla volta, come nei sistemi batch. In seguito i sistemi, disponendo di maggiori capacità in termini di CPU, sono diventati *multitasking* e più job vengono ora eseguiti contemporaneamente. I principali sistemi multitasking sono anche multithreading. L'abbinamento non è casuale, il multithreading (infatti) favorisce il multitasking (più processi in esecuzione). Ecco spiegato il perchè: il multithreading da la possibilità di realizzare più flussi di controllo, tra loro concorrenti, nell'ambito di una stessa applicazione. Ogni thread è caratterizzato, per questo motivo, da un proprio program counter (logico) che ne descrive i comportamenti, uno stack per le chiamate non ancora ritornate, uno stato logico ed alcuni registri di memoria per le variabili. In questo modo un applicazione, quindi un processo, può essere associata all'attività congiunta di uno o più thread ognuno dei quali si occupa di fornire all'applicazione un determinato servizio. In Microsoft Word ad esempio sono contemporaneamente in esecuzione diversi thread. Alcuni di questi si occupano

del controllo dell'ortografia mentre altri sono orientati alla formattazione del testo. Ogni singolo flusso sequenziale di istruzioni che opera all'interno di un processo è detto thread. Più flussi, e quindi più thread, associati ad un solo processo fanno diventare multithreading il processo.
Pertanto, così come la multiprogrammazione si occupa dell'esecuzione parallela di più processi, il multithreading si occupa, invece, dell'esecuzione (concorrente) di più thread in un processo. Il vantaggio di avere più thread in un processo risiede nella possibilità che questi hanno di condividere i dati del processo. Mentre a più processi in esecuzione è vietato l'accesso verso l'indirizzamento di altri processi (anch'essi in esecuzione), più thread di uno stesso processo riescono meglio a condividere tra loro i dati (Fra i vari processi l'interazione è in relatà possibile solo tramite socket, pipe o segnali. Tuttavia sappiamo bene quanto elaborato e lungo possa essere il codice per lo scambio di dati e/o messaggi).



Un processo con due thread

Quando un processo con thread multipli viene eseguito su un sistema con una sola CPU, i thread vengono eseguiti a turno, così come avveniva per i processi. Non esiste alcun meccanismo di protezione fra thread cosicchè un thread può leggere, scrivere ed anche cancellare lo stack di un altro thread. Il linguaggio java si presta bene alla modellazione di processi multithreading. In java ogni thread è un oggetto ed ogni thread è un'istanza della classe `Thread` del package `java.lang.Thread`. Ogni istanza della suddetta classe deve implementare al suo interno un metodo `run()` che sarà quindi eseguito dal thread. In altre parole, nella scrittura di un thread si estende la classe `Thread` ereditando così il metodo `run()` che va riscritto per la applicazione che si vuole creare. In realtà si ereditano diversi metodi, tutti utili alla gestione dei thread, uno di questi è il metodo `start()` che serve ad eseguire il thread (il metodo `run()` implementa il corpo del thread mentre il metodo `start()` da inizio all'esecuzione del metodo `run()`). Esistono due diverse strategie per realizzare un thread. Una prima possibilità è quella che prevede l'estensione della classe `java.lang.Thread` e la sovrascrittura del metodo `run()`. Altra possibilità consiste invece nel far implementare ad una classe l'interfaccia `Runnable`. Quest'ultimo è indubbiamente più vantaggioso poichè in java è possibile estendere una sola classe mentre ad una classe è concessa la possibilità di implementare più interfacce. Se dunque la classe che descrive gli oggetti che intendiamo far diventare thread è già l'estensione di un'altra classe è necessario far implementare alla classe in oggetto l'interfaccia `Runnable` se vogliamo riuscire nei nostri scopi. Ecco un esempio di come realizzare dei thread:

Primo metodo: estensione della classe `Thread`

```
public class TableTennis
{
      public static void main(String[] a)
      {
            Ping thread1=new Ping();
```

```
            thread1.start();
            Pong thread2=new Pong();
            thread2.start();
      }
}

class Ping extends Thread
{
      public void run()
      {
            while(true)
            {
                  system.out.println("Ping");
            }
      }
}

class Pong extends Thread
{
      public void run()
      {
            while(true)
            {
                  system.out.println("Pong");
            }
      }
}
```

In questo esempio la classe `TableTennis` manda in esecuzione due thread. Uno discende dalla classe `Ping`, l'altro invece dalla classe `Pong`. Il thread della classe `Ping` provoca la stampa a video della stringa "`Ping`", quello della classe `Pong` insiste nella stampa della stringa "`Pong`". Entrambi i thread sono dapprima creati, ognuno dalle rispettive classi e successivamente mandati in esecuzione invocando su ciascuno oggetto `Thread` il metodo `run()`. Il risultato è facilmente intuibile, entrambi i cicli di stampa continuano all'infinito. Quando uno dei due thread viene schedulato per un time slice si assiste alla stampa della stringa Ping oppure Pong. Può anche capitare che un thread si avvantaggi rispetto all'altro thread riuscendo ad ottenere la schedulazione per più time slice consecutivi.

```
C:>Documents and Settings\Luca>java TableTennis
Ping
Pong
Ping
Ping
Ping
Pong
Pong
Ping
Ping
Pong
…
```

Secondo metodo: implementazione dell'interfaccia Runnable

```
public class TableTennis
{
      public static void main(String[] a)
      {
            Ping player1=new Ping();
            Pong player2=new Pong();
            Thread thread1=new Thread(player1);
            Thread thread2=new Thread(player2);
```

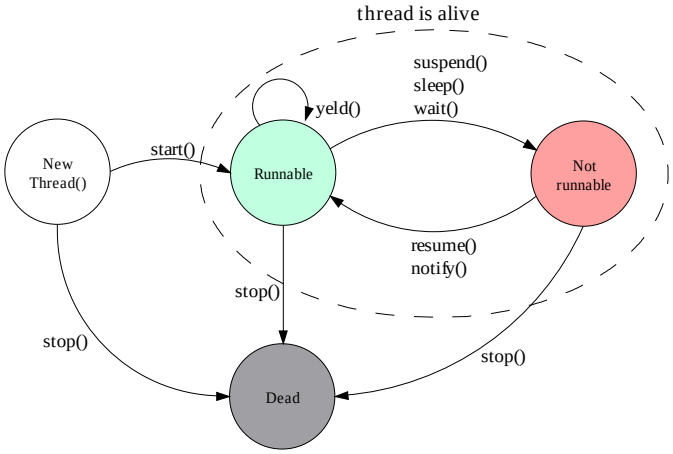
```
            thread1.start();
            thread2.start();
      }
}

class Ping implements Runnable
{
      public void run()
      {
            while(true)
            {
                  System.out.println(“Ping”);
            }
      }
}

class Pong implements Runnable
{
      public void run()
      {
            while(true)
            {
                  System.out.println(“Pong”);
            }
      }
}
```

L’effetto generato sul video è sempre lo stesso, la stampa alternate delle stringhe Ping e Pong. In questo caso, poichè la classe da noi scritta non discende direttamente dalla classe `Thread` il programma principale si preoccupa prima della costruzione degli oggetti `player1` e `player2`, e successivamente della costruzione di due thread (`thread1` e `thread2`) usando il costruttore della classe `Thread`. I thread sono quindi lanciati con il metodo `start()`.
Quando lanciamo un applicazione java vengono eseguite le istruzioni contenure in essa in maniera sequenziale, la prima istruzione, pertanto, partirà dal metodo `main()`. Spesso, soprattutto in fase di debugging, allo scopo di simulare l’esecuzione del programma, il programmatore immagina nella sua mente un cursore che scorre in maniera sequenziale le istruzioni. Talvolta simulerà questo cursore con il suo dito, altri strumenti di debugging, invece, puntano al rigo che sarà eseguito al prossimo ciclo di clok. Detto ciò, sicuri che quanto detto sia sufficientemente familiare, possiamo pensare al thread come al puntatore appena spiegato. Ecco quindi che a runtime esiste almeno un thread in esecuzione, quello cioè che esegue il processo. Il multithreading aggiunge più thread ad un processo, ognuno con assegnati compiti. Un thread è in esecuzione sul processo fintanto che il metodo `run()` non esegue l’ultima istruzione e quindi ritorna. In alcuni casi (così come avveniva per i processi) un thread può terminare prima dell’ultima istruzione, ciò accade ad esempio quando si invocano su di esso i metodi `destroy()`, `stop()` oppure in presenza di eccezioni non catturate.
Altri metodi ereditati dalla classe `Thread` sono:

```
public void start()                                 Lancia il thread
public void run()                                   Esegue il codice
public final void destroy()                         Distrugge il thread
public final void stop()                            Arresta il thread
public final void suspend()                         Sospende il thread
public final void resume()                          Riattiva il thread
public final void sleep(long n)                     Sospende per n msec
public final void setPriority(int newPriority)      Modifica la priorità
public final int get priority()                     Ottieni la priorità
```

```
public static void yield()                    Torna in coda di schedulaz.
public final Boolean isAlive()                E' true se il thread è vivo
```

L'uso di questi metodi può condizionare lo stato attuale di un thread, quest'ultimo quando è stato appena creato con l'apposito costruttore della classe `Thread` viene collocato in uno stato che diremo `New Thread`. Il thread può da questo stato passare allo stato `dead` (chiamata al metodo `stop()`) oppure essere messo nello stato `Runnable` che caratterizza tutti i thread che possono essere mandati in esecuzione (chiamata al metodo `start()`). Ancora una volta, da qui il thread può finire nello stato `dead` (se arriva una chiamata al metodo `stop()` oppure se il metodo `run()` del thread esegue l'ultima istruzione). Il metodo `yeld()` se applicato su di un thread in stato `Runnable` rimanda il thread in coda nella lista dei thread in stato `Runnable`. I metodi `suspend()`, `sleep()` e `wait()` portano un thread `Runnable` in stato `Not runnable`. Da qui, con una certa frequenza che varia da programma a programma, un thread può ritornare a far parte dei thread in stato `Runnable` (chiamate `resume()` e `notify()`) oppure può finire in stato `dead` (chiamata al metodo `stop()`).



La condivisione di una qualunque risorsa è sempre causa dei problemi, anche per i thread il problema è fortemente sentito al punto che esistono diverse soluzioni. I thread condividono tra di loro le variabili locali del processo utente, ed allora, un thread che accede ad una variabile può non aver letto un valore giusto se poco prima un altro thread ha già fatto accesso alla stessa variabile e ne sta ancora facendo uso. In altre parole, il valore letto non è allineato al valore corrente della variabile! Vogliamo qui presentare uno solo dei metodi usati per affrontare il problema della sincronizzazione che sarà affrontato ampiamente in seguito. Java mette a disposizione un interessante modificatore `synchronized` che, se prescritto ad un metodo, prevede che un solo thread alla volta possa eseguire il suddetto metodo. Quando un thread esegue un metodo synchronized viene eseguita una *operazione di lock* sull'oggetto a cui appartiene il metodo e quando il metodo termina viene al contrario eseguita una *operazione di unlock* sull'oggetto. Prima di tale unlock, eventuali altri metodi synchronized non possono avvenire, i thread che provano ad eseguirli vengono per questo motivo messi in attesa. La keyword synchronized può inoltre essere usata anche per proteggere blocchi arbitrari di istruzioni che fanno accesso a variabili condivise.

Implementazione dei thread

Esistono due modi per imlementare i thread, la loro realizzazione può avvenire direttamente nello spazio utente oppure nello spazio kernel. Le due soluzioni, ovviamente, comportano diversi vantaggi e svantaggi che qui cercheremo di affrontare.

Per quanto riguarda la possibilità di implementare i thread nello spazio utente va subito detto che in tal caso il kernel, non sapendone l'esistenza, schedulerà processi anzichè thread, come del resto è abituato a fare. Questa soluzione, allora, si addice a tutti quei sistemi operativi che non supportano nativamente l'utilizzo dei thread. I thread nello spazio utente, infatti, sono realizzati da funzioni di libreria che girano in spazio utente (`thread_create()`, `thread_exit()` e `thread_wait()`). Ogni

processo schedulato dal sistema operativo e contenente al suo interno diversi thread ha quindi bisogno di una tabella per i thread per segnarne l'attività svolta. Si tratta di una tabella simile a quella usata per schedulare i processi ma, diversamente da questa, molto più snella. Essa conterrà le proprietà caratterizzanti di ciascun thread in esecuzione sul processo: program counter di ogni thread, stack pointer e registri di variabili. Tale tabella è quindi aggiornata ad ogni cambio di contesto, da un thread ad un altro (come ad esempio avviene quando il thread invoca il metodo `yeld()` che oltre a collocare il thread chiamante in fondo alla lista dei thread in stato Runnable, quando questo ha terminato un elaborazione, può in alcuni casi procedere esso stesso a salvare in tabella le informazioni del thread e attirare l'attenzione dello schedulatore affinchè venga quindi attivato un successivo thread). Quando un thread deve fare qualcosa che potrebbe bloccarlo localmente (ad esempio attendere le operazioni di un altro thread e poi ripartire) esso stesso chiama una procedura che a run time controlla se è il caso di bloccare il thread (se infatti il thread deve attendere un altro thread è opportuo fermarlo), di salvare quindi i suoi valori nella tabella dei thread e di attivarne immediatamente un altro. Se il sistema dispone di apposite chiamate per salvare e ricaricare i valori di registro, l'intero scambio dei thread può avvenire in una manciata di istruzioni. Queste operazioni potrebbero essere ad esempio essere abbinate al metodo `yeld()`.
Notare che la creazione di un thread così come il cambio di contesto sono operazioni locali allo spazio utente e pertanto non necessitano di uno switch allo spazio kernel, queste possono quindi avvenire in maniera molto più veloce! Ogni processo dispone di una propria tabella per i thread. La schedulazione dei thread avviene a run time nello spazio utente ed è solitamente anch'essa supportata da una libreria. Questo permette (se il programmatore lo desidera) di avere il proprio algoritmo di schedulazione. Fin qui i possibili vantaggi di una implementazione nello spazio utente dei thread, osserviamo adesso i principali svantaggi. Un grosso problema, che ci costringerà ad adottare un compromesso, è l'eventuale chiamate ad una system call bloccante che un thread può invocare nel corso delle sue istruzioni. Quando un thread esegue una chiamata di sistema e si blocca in attesa di ricevere un servizio l'intero processo a cui appartiene viene bloccato (Ad esempio, in un server web una eventuale lettura da disco fatta con una `read()` blocca tutti gli altri thread!). Come implementare le chiamate di sistema bloccanti?
Una prima, banale ma possibile, soluzione potrebbe richiedere la modifica di tutte le chiamate di sistema bloccanti in chiamate non bloccanti. Questo implicherebbe una pesante modifica al sistema operativo ed a molti programmi utente. E' una soluzione possibile ma inaccettabile dal momento che uno dei principali obiettivi che hanno spinto per l'introduzione dei thread è stata la possibilità di farli funzionare su qualunque sistema esistente senza apportare modifche sugli stessi.
Nel caso in cui è possibile sapere in anticipo se una chiamata si bloccherà (in alcuni sistemi UNIX esiste una chiamata di sistema `select()` che permette di dire al chiamante se una eventuale chiamata bloccante, ad esempio una `read()`, si bloccherà oppure potrà avere immediatamente un seguito nel caso in cui i dati siano già disponibili) si può attuare un interessante soluzione. Le chiamate di sistema bloccanti possono essere sostituie con nuove chiamate che prima eseguono una `select()` e solo dopo, quando cioè la chiamata può avvenire senza blocchi, lanciano la chiamata di sistema. Nel caso in cui la funzione `select()` indica al thread chiamante che la chiamata di sistema che intende invocare non può avvenire (ad esempio perchè i dati richiesti dalla `read()` non sono ancora disponibili nel buffer) viene mandato in esecuzione un nuovo thread. Il codice che è messo attorno alla chiamata di sistema e che esegue i controlli tramite la funzione `select()` si dice *jacket* o *wrapper*.
Un altro problema può invece nascere dalla gestione della memoria, in particolare dai *fault di pagina* che studieremo meglio in avanti. In poche parole, un processo può essere in esecuzione su di un sistema anche quando l'immagine del processo non è interamente in memoria. Se il processo richiede una istruzione non ancora in memoria (*faul di pagina*) il sistema operativo si attiva affinchè l'istruzione venga effettivamente portata in memoria caricandola dal disco. Nel tempo impiegato alla ricerca dell'istruzione nell'immagine del processo in memoria del disco ed al caricamento della suddetta istruzione, l'intero processo è bloccato. Nel frattempo potrebbe andare in esecuzione un altro thread ma ciò non può avvenire poichè il kernel del sistema (che sta implementando i thread nello spazio utente), non sapendo dell'esistenza dei thread, blocca l'intero processo fintanto che la lettura dal disco non rende disponibile l'istruzione invocata. Altro problema con i thread a livello

utente è che se un thread inizia l'esecuzione delle proprie istruzioni nessun altro thread può essere eseguito successivamente se il primo thread in esecuzione non lascia volontariamente la CPU. Il segnale di clock adottato dal sistema per rendere possibile la schedulazione dei processi secondo la modalità round robin non è disponibile per i thread e qualora lo fosse richiederebbe una maggiore complessità alla programmazione dei thread.
Quando, invece, il kernel del sistema supporta nativamente i thread non occorre un sistema a run time per l'esecuzione dei thread. Qui i thread sono creati da primitive di sistema (non da primitive locali allo spazio utente), il kernel ha poi, oltre alla tabella dei processi, una tabella per i thread di sistema. Quando un processo utente vuole creare un thread invoca una chiamata di sistema che effettua la creazione del thread ed aggiorna la tabella dei thread attivi sul sistema. Anche quando un thread utente deve essere terminato si verifica nell'ordine dapprima una chiamata di sistema da parte del processo utente e successivamente (in spazio kernel) viene decisa la sua terminazione. Quando un thread si blocca su una chiamata di sistema bloccante, il kernel che adesso è a conoscenza dell'esistenza dei thread può decidere di eseguire un diverso thread. Siccome le procedure per la creazione e la terminazione dei thread sono implemetate nel kernel come chiamate di sistema esse richiedono generalmente più tempo per la realizzazione ed allora alcuni sistemi adottano una strategia alternativa. Quando un thread è distrutto si aggiorna la tabella dei thread marcando quest'ultimo come non più eseguibile. Tuttavia le strutture dati relative al thread continuano ad essere mantenute vive dal kernel cosicchè quando giungono richieste di creazione per nuovi thread si decide di riciclare dapprima i thread al momento non eseguibili e solo dopo, quando nessun thread non eseguibile è disponibile, si procede alla reale creazione del thread. In questo modo si risparmia un po del tempo richiesto dal sistema per la creazione dei thread. Anche il problema relativo ai fault di pagina può qui essere scongiurato. Il kernel, infatti, può verificare l'attivatà dei thread anche quando questi sono in attesa del caricamento delle istruzioni da loro richieste esso può quindi decidere, nell'attesa, di eseguire altri thread.
Avendo ora una chiara situazione, sia della prima implementazione che della seconda, è ora possibile mettere a confronto le due realizzazioni, citando per una e per l'altra i vantaggi e gli svantaggi che esse comportano:

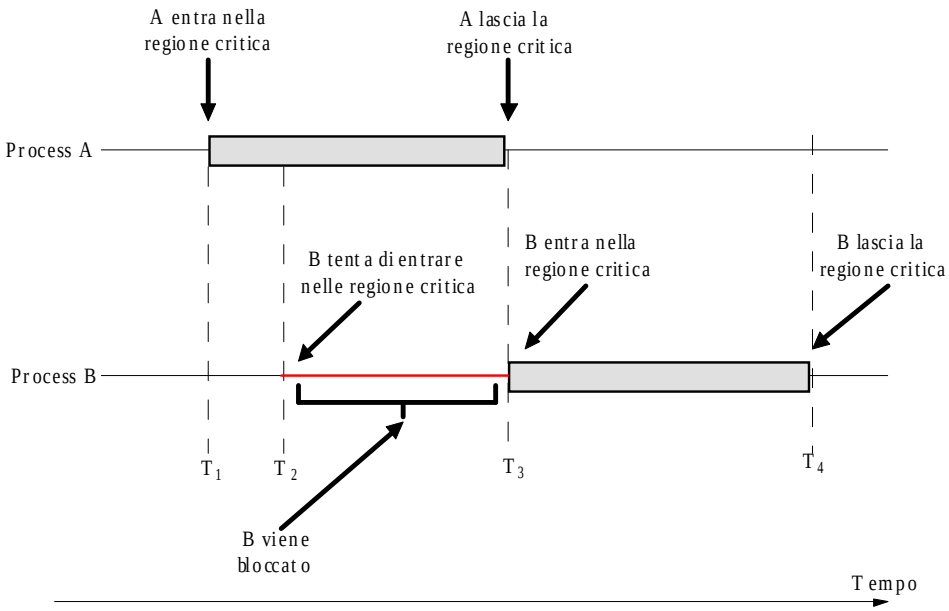
- Un grosso vantaggio a favore dell'implementazione dei thread in spazio utente è dunque la velocità con cui i thread possono essere creati e la rapidità con cui può avvenire un cambio di contesto fra thread non essendoci il passaggio allo spazio kernel che è invece necessario se i thread sono implementati nel kernel;
- Ed ancora, a favore del primo metodo (implementazione in spazio utente) c'è la possibilità di implementare un proprio algoritmo di schedulazione piuttosto che sottostare a quello implementato dal sistema in spazio kernel. Anche i vecchi sistemi possono trarre vantaggi dall'uso dei thread implementandoli nello spazio utente;
- Dalla gestione delle system call è emerso un importante problema nell'implementazione dei thread in spazio utente risolto mediante costruzione di codice wrapper attorno alle chiamate di sistema. Nell'implementazione in spazio kernel, invece, il sistema può subito intervenire mandando in esecuzione un diverso thread (in tal caso il sistema non richiede system call non bloccanti);

In realtà esiste anche una terza possibile implementazione per i thread che, siccome cerca di sfruttare i vantaggi delle precedenti implementazioni viste, è detta, per questo motivo, ibrida. Essa consiste nella definizione di N thread nello spazio utente e di $M \leq$ N thread in spazio kernel. Il sistema a run time decide quali thread mandare in esecuzione e a tale proposito mappa i thread da lui scelti sui thread creati nello spazio utente *(operazione di mapping)*.

Problemi di sincronizzazione fra thread e programmazione conccorrente
In alcuni sistemi i processi, così come possono fare i thread, possono interagire tra di loro per scambiarsi dati e/o sincronizzarsi per eseguire azioni secondo una sequenza corretta. Un esempio reale è la directory speciale per stampa detta *spooler directory*: un demone della stampante controlla con una certa frequenza la cartella e se ci sono nomi di file da stampare li invia alla stampante

affinchè il processo di stampa venga ultimato. Successivamente i nomi dei file vengono rimossi. La spooler directory è nella sostanza una struttura dati particolare, molto simile ad un array, capace di memorizzare in ogni sua cella l'intero nome di un file. La struttura si completa poi di due variabili condivise con tutti i processi. Una variabile punta al prossimo file da stampare (variabile *out*) mentre l'altra tiene memoria dell'indice dell'array immediatamente disponibile, il primo indice libero nell'array (variabile *in*).
Supponiamo adesso l'esistenza di due processi che diremo A e B. Il processo A è in esecuzione sul sistema quando ad un certo istante di tempo decide di mettere nello spooler directory un file da stampare. A tale proposito, accedendo alla suddetta cartella, ricava (leggendo le variabili condivise) la posizione nell'array della prima cella libera. Tuttavia, ancora prima di riuscire ad inserire il nome del file che intende stampare, lo schedulatore decide che il processo A è stato a lungo in esecuzione e ne sospende temporaneamente le attività. Subentra un diverso processo, il processo B, che come il processo A è desideroso di stampare un file. Per questo motivo anche il processo B accede alle variabili condivise della spooler directory, ricava l'indice della prima cella libera e ripone in essa il nome del file che intende stampare. A questo punto il demone della stampante aggiornerà le variabili della cartella condivisa e darà inizio, appena possibile, al processo di stampa del documento.
Quando il processo A ritorna in esecuzione, le istruzioni riprendono dal punto in cui il processo era stato interrotto. Pertanto, siccome il processo A aveva già letto le variabili condivise (in particolar modo l'indice della prima cella libera nell'array) procede spedito nella scrittura del nome del file che intende stampare. In altre parole il processo A sovrascrive con il proprio nome di file quello precedentemente scritto dal processo B. In questo scenario appena descritto l'utente del processo B attenderà inutilmente la stampa del proprio file. Quando due o più processi stanno leggendo o scrivendo un dato condiviso ed il risultato finale dipende fortemente dall'ordine con cui vengono eseguiti i processi si dice di essere in presenza di *race condition* o più semplicemente di *corse critiche*. Tutto quello che qui diremo a proposito delle corse critiche nonchè le soluzioni che analizzeremo è valido sia per il modello a processi che per quello basato sui thread.
Come più volte sottolineato nel corso di queste note, la condivisione di una risorsa è fonte di problemi che meritano un attenta gestione. Una soluzione naturale al problema è vietare l'uso contemporaneo di una risorsa condivisa se questa è già utilizzata da un altro processo. Nel problema della spooler directory la corsa critica si verificava perchè il processo B accedeva alle variabili condivise prima che il processo A terminasse il loro impiego.
Non sempre due processi danno luogo a problemi di corse critice, ciò si verifica solo quando esistono dei potenziali punti di interferenza tra i due processi. Tali interferenze sono costituite dalle porzioni di codice che accedono ai dati condivisi e che solitamente vengono dette sezioni critiche. Pertanto, affinchè non ci siano corse critiche è necessario non eseguire mai contemporaneamente le regioni critiche di due o più processi. Una regione critica deve essere eseguita in mutua esclusione rispetto a tutte le altre. Sebbene la mutua esclusione ci permetta di evitare corse critiche non riusciamo in questo modo a far cooperare assieme due o più processi in maniera efficace. Infatti, per avere una buona cooperazione tra processi è opportuno soddisfare a quattro requisiti:

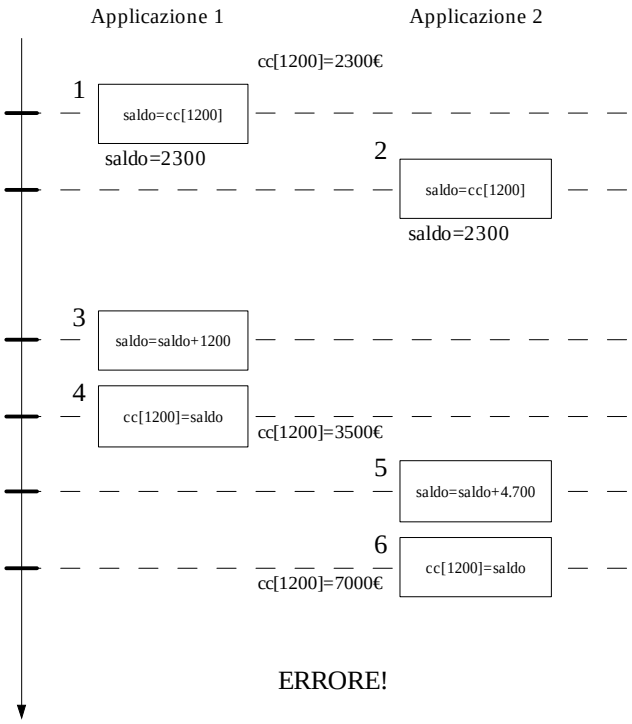
- due processi non devono mai trovarsi contemporaneamente nelle rispettive regioni critiche;
- non vanno fatte ipotesi sul numero di CPU e sulle loro velocità;
- un processo in esecuzione che non attraversa una regione critica non deve bloccare gli altri processi;
- un processo non deve aspettare indefinitivamente prima di entrare nella regione critica;



Un altro esempio di mancata sincronizzazione è il seguente: supponiamo due thread di due applicazioni simili che seguono contemporaneamente la procedura `effettuaVersamento()` così definita:

```
public int effettuaVersamento(int numeroCC, int versamento)
{
   if(versamento>0)
   {
      saldo=CC[numeroCC];
      saldo=saldo+versamento;
      CC[numeroCC]=saldo;
      return salso;
   }
   else return -1;

}
```

I thread delle due applicazioni condividono il vettore `CC[]` che contiene i saldi di tutti i conto correnti mentre la variabile `saldo` è locale alle procedure e quindi diversa nei due thread, utile a calcolare il nuovo valore del saldo di conto corrente. Supponiamo adesso una eventuale singolarità: un applicazione bancaria decide di versare sul numero di conto corrente una certa somma maturata, nello stesso istante di tempo, in un altra sede della stessa banca, viene deciso di fare la stessa cosa, un debitore effettua un versamento a favore di un cliente (il cui numero di conto corrente è proprio quello usato nell'altra sede della banca dove nello stesso istante di tempo si sta aggiungendo la somma di danaro maturata nel tempo). Ecco allora una possibile sequenza di azioni tra loro non sincronizzate:



La mutua esclusione in questo caso avrebbe scongiurato l'interferenza esistente nelle due procedure delle due applicazioni, oltre ad evitare numerose proteste da parte del cliente nei confronti della banca. Esistono diverse possibilità per ottenere la mutua esclusione:

Disabilitazione delle interruzioni
Un processo che entra in una sezione critica disabilita le interruzioni e le riabilita solo quando lascia la regione critica. Avendo disabilitato le interruzioni il processo è sicuro di accedere alla regione critica in mutua esclusione. In tal caso infatti non può avvenire il cambio di contesto da un processo ad un altro ed il processo che entra in regione critica deve esso stesso lasciare volontariamente la CPU. Questa soluzione non è efficiente, un processo potrebbe approfittarne per monopolizzare l'uso della CPU. Inoltre, la disabilitazione delle interruzioni funziona solo se il sistema ha un unica CPU. La diabilitazione delle interruzioni è infatti locale alla CPU che esegue l'istruzione `disable()` e se il sistema è multiprocessore altri processi possono comunque essere eseguiti in parallelo.
In alcuni casi la disabilitazione delle interruzioni può comunque essere una soluzione valida, ciò potrebbe ad esempio essere fatto dal kernel in fase di aggiornamento della tabella dei processi. In questo modo lo schedulatore non accede ad una tabella dei processi inconsistente.

Mutua esclusione con attesa attiva
L'attesa attiva consiste nel testare in continuazione il valore di una variabile, è possibile un approccio sia software che hardware. L'utilizzo di variabili di lock costituiscono il primo approccio software: un processo che desidera entrare nella sua regione critica controlla dapprima il valore di una variabile di lock che se vale 0 permette al processo di accedere alla regione critica, se la variabile di lock è 1 il processo aspetta che questa diventi 0 (attesa attiva). Altre soluzioni software sono l'alternanza stretta e la soluzione semplificata di Peterson.
Nell'alternanza stretta una variabile intera indica il turno del processo che può andare in regione critica. Il processo 0 prima di entrare nella sezione critica controlla il valore della variabile `turno` e se la trova pari a 0 va in regione critica. Anche il processo 1 legge la variabile `turno` ma quest'ultimo trovondola diversa da 1 entra in un ciclo di attesa (legge in continuazione la variabile `turno`, attesa attiva) che spreca inutili cicli di clok. Quando il processo 0 esce dalla sezione critica aggiorna la variabile `turno` ad 1 ed il processo 1 (che sta leggendo in continuazione la variabile `turno`) può adesso andare in regione critica. All'uscita della regione critica il processo rimette la variabile `turno` a 0. Questo provoca un alternanza stretta, con attesa attiva, fra i processi 0 ed 1. Si

tratta di una situazione inaccettabile dal momento che un processo più lento può ritardare notevolmente l'altro (ritardandone ad esempio l'accesso alla regione critica). Un esempio di codice che implementa l'alternanza stretta fra due processi è il seguente:

```
            Processo 0                              Processo 1

...                                     ...
while(true)                             while(true)
{                                       {
   while(turno!=0) /* loop */              while(turno!=1) /* loop */
   regione_critica();                       regione_critica();
   turno=1;                                 turno=0;
   regione_non_critica();                   regione_non_critica();
}                                       }
```

L'altra soluzione software per ottenere la mutua esclusione è la soluzione semplificata di Peterson con attesa attiva che combina l'uso delle variabili di lock con l'alternanza stretta dei processi, eccone un esempio:

```
#define FALSE 0
#define TRUE 1
#define N 2                                 /* numero di processi */

int turno                                   /* A chi tocca? */
int interessato[N];                         /* vettore inizializzato a tutti 0 */

void entra_nella_regione(int processo)      /* processo può valere 0 oppure 1 */
{
   int altri;                               /* numero dell'altro processo */
   altri=1-processo;                        /* l'altro processo */
   interessato[processo]=TRUE;              /* mostrati interessato */
   turno=processo;                          /* imposta il flag */
   while(turno==processo && interessato[altri]==FALSE)
}

void lascia_la_regione(int processo)
{
   Interessato[processo]=FALSE;
}
```

La struttura di un processo sarà pertanto:

```
/* Processo i-simo */
...
while(fine_lavoro!=TRUE)
{
   sezione_non_critica();
   entra_nella_regione(i);
   /* regione critica */
   ...
   lascia_la_regione(i);
}
```

Prima di usare le variabili condivise che appartengono per questo motivo ad una regione critica di interferenza, ciascun processo esegue una chiamata alla procedura `entra_nella_regione(int processo)` con il proprio numero di processo. Tale chiamata può determinare una eventuale attesa se l'altro processo è già in regione critica.
Finita la manipolazione delle variabili condivise il processo esce dalla regione critica effettuando una chiamata alla procedura `lascia_la_regione(int processo)` con il proprio numero di

processo. Una soluzione hardware al problema della mutua esclusione con attesa attiva è l'uso di una particolare istruzione, l'istruzione *test and set lock* TSL (verifica ed imposta il blocco). Molti sistemi multiprocessori adottano questa soluzione. L'istruzione:

```
TSL   RX,LOCK
```

mette nel registro `RX` il contenuto della parola di memoria `LOCK` e memorizza un valore diverso da zero all'indirizzo di memoria di `LOCK`. Le operazioni di lettura e memorizzazione della parola sono garantite indivisibili (operazione atomica). Quando la variabile `LOCK` è a zero qualunque processo può metterla ad 1 fissandola con l'istruzione TSL e può poi leggere o scrivere nella memoria condivisa. Quando il processo ha finito rimette la variabile `LOCK` a zero. Ad esempio:

```
entra_nella_regione:
      TSL   REGISTRO,LOCK          | copia LOCK inREGISTRO e imposta LOCK ad 1|
      CMP   LOCK,#0                | confronta lock con 0 |
      JNE   entra_nella_regione    | se non è 0 LOCK è già stata impostata |
      RET                          | ritorna al chiamante |

Lascia_la_regione:
      MOVE   LOCK,#0               | imposta LOCK a 0 |
      RET                          | ritorna al chiamante |
```

Pertanto, una soluzione al problema della mutua esclusione consiste nel chiamare, prima di entrare nella regione critica, la procedura `entra_nella_regione` che procura un attesa attiva al chiamante se `LOCK` è già stata impostata. All'uscita della regione critica il processo chiama la procedura `lascia_la_regione` che mette a 0 la variabile di `LOCK`.

Mutua esclusione senza attesa attiva

Le soluzioni finora viste per ottenere la mutua esclusione nell'accesso a variabili condivise hanno l'inconveniente dell'attesa attiva. Il processo, oppure il thread, che non riesce ad accedere alla regione critica effettua in continuazione la lettura di una variabile di lock. Ovviamente ciò comporta un inutile spreco di cicli di CPU. La mutua esclusione può essere ugualmente raggiunta anche senza provocare un attesa attiva al processo che tenta l'accesso alla regione critica. L'idea di base è semplicemente bloccare un processo se questo non può accedere ad una regione critica. Tutto ciò oltre che essere ragionevole (del resto se il processo non può entrare nella regione critica perchè tenerlo attivo inutilmente?) ci consente di risparmiare cicli di CPU che in questo modo posso essere assegnati ad altri processi. Per evitare l'attesa attiva si è allora pensato alla presenza di due nuove primitive realizzate come system call: la primitiva `sleep()` blocca il processo chiamante, la primitiva `wakeup(P)` sveglia il processo `P`.

L'introduzione di queste nuove primitive non risolve completamente il problema, possono infatti verificarsi pericolose corse critiche (come quella che avveniva nell'esempio della spooler directory). Il classico esempio del produttore-consumatore ci permetterà di apprezzare meglio quanto solitamente potrebbe accadere (per la legge di Murphy è bene ricordare che se una cosa potrebbe andare storta questa finirà prima o poi per andare storta, quindi, meglio analizzare e capire il problema piuttosto che spostarlo o posticiparlo). Nel problema del produttore-consumatore modelliamo con un linguaggio di programmazione due entità, il produttore ed il consumatore appunto. Il produttore condivide con il consumatore un buffer avente una capacità limitata di `N` elementi. Il compito del produttore è quello di inserire elementi nel buffer (dopo averli creati) quando l'attuale capienza del buffer è stata azzerata (l'inserimento degli elementi non può comunque superare `N`). Il compito del consumatore è quello di prelevare gli elementi inseriti dal produttore, se presenti. Per tenere il conto degli elementi presenti nel buffer abbiamo bisogno di una variabile di conteggio che per noi è `contatore`. Pertanto, il produttore controllerà dapprima se `contatore==N`, in caso affermativo non saranno necessari altri elementi perchè il buffer è pieno ed il produttore si sospenderà da solo con una chiamata alla procedura `sleep()`, al contrario, invece, depositerà nel buffer l'elemento prodotto. Il consumatore, invece, controlla dapprima se la variabile

contatore==0, in caso affermativo si sospenderà da solo con una chiamata alla procedura sleep() non essendoci elementi da prelevare nel buffer, altrimenti preleverà dal buffer un elemento e successivamente lo impegnerà in qualche sua operazione. La cosa più importante da tenere sott'occhio è il fatto che entrambi i processi controllano se è il caso di svegliare l'altro processo. Un esmepio del codice del problema del produttore-consumatore è il seguente:

```
#define N 100
int contatore=0;

/* Processo produttore */
void produttore(void)
{
   int elemento;
   while(TRUE)
   {
      elemento=produciElemento();
      if(contatore==N) sleep();
      inserisciElemento(elemento);
      contatore=contatore+1;
      if(contatore==1) wakeup(consumatore);
   }
}

/* Processo consumatore */
void consumatore(void)
{
   int elemento;
   while(TRUE)
   {
      if(contatore==0) sleep();
      elemento=estraiElemento();
      contatore=contatore-1;
      if(contatore==N-1) wakeup(produttore);
      consumaElemento(elemento);
   }
}
```

Una corsa critica che si può verificare è la seguente: il processo consumatore ha rilevato che contatore è uguale a 0 e sta per eseguire la chiamata alla primitiva sleep(). Tuttavia ancora prima di effettuare la sleep() il processo consumatore viene schedulato dal sistema essendo stato a lungo in esecuzione (è scaduto il time slice a lui assegnato). Il processo produttore che ha un elemento pronto verifica dapprima il valore della variabile contatore e trovandola pari a 0 inserisce l'elemento nel buffer, quindi incrementa contatore ad 1. Poichè contatore è adesso pari ad 1 il processo produttore esegue una chiamata a wakeup(consumatore) sul consumatore, ma poichè il processo consumatore in realtà non è ancora stato bloccato (è stato schedulato prima che lo potesse diventare!) la sveglia viena persa (le sveglie non si accumulano!). Quando successivamente il processo consumatore riprende la sua esecuzione riesce a completare la chiamata alla sleep() (precedentemente interrotta) ed andrà a dormire per sempre. La sveglia è stata infatti già mandata dal produttore ed è andata persa quando il processo è stato schedulato.
Se il problema potrebbe sembrare la perdita della sveglia si può allora optare per una soluzione che tenta di memorizzare il numero di sveglie cosicchè queste non vengono perse quando il processo consumatore non è ancora stato bloccato. E' facile intuire che ciò sposterebbe solo il problema piuttosto che risolverlo, i numeri di bit che infatti posso decidere di dedicare al conteggio delle sveglie rimane comunque limitato così come l'incombenza del problema appena visto.
I semafori costituiscono uno strumento di sincronizzazione fra thread/processi. Si tratta di un tipo di dato a cui sono associate distinte operazioni atomiche. In un semaforo una variabile valore effettua il conteggio delle sveglie salvate, pertanto, se valore=0 nessuna sveglia è stata salvata; se valore è un qualunque numero positivo una o più sveglie sono già arrivate e sono state salvate. Quando si

genera un semaforo si può assegnare a `valore` un valore iniziale mediante la procedura `init(valore_iniziale)`. In seguito possono essere invocate sul semaforo due azioni:

- `semaforo.down()`: controlla che `valore` sia maggiore di zero, se così è decrementa `valore` di 1 altrimenti sospende il processo chiamante con una `sleep()` senza completare, per il momento, la chiamata a `down()`;
- `semaforo.up()`: incrementa `valore` di 1 e se uno o più processi erano sospesi (incapaci di completare la `down()`) uno di essi (scelto a caso) viene svegliato dal sistema per completare la `down()`;

Le procedure `up()` e `down()` devono essere implementate come azioni atomiche. Esse potrebbero essere implementate all'interno del sistema operativo come system call. Per garantire, poi, che vengano eseguite in maniera atomica si potrebbe pensare di adottare la disabilitazione degli interrupt (cosa che in questo caso è possibile dal momento che è il sistema operativo a disabilitare gli interrupt e non un processo) oppure se il sistema di calcolo è multiprocessore si può usare la soluzione basata sull'istruzione TSL. In quest'ultimo caso l'attesa attiva è accettabile dal momento che si tratta del solo codice della `up()` o della `down()`. L'implementazione in java di un semaforo ricorre ai metodi `wait()` e `notify()`, gli equivalenti di `sleep()` e `wakeup()`, eccone un esempio:

```
public class Semaforo
{
      private int inAttesa;   /* Thread in attesa */
      private int stato;      /* Stato del semaforo, 0=rosso */

    /* Costruttore */
    public Semaforo(int stato_iniziale)
    {
      stato=stato_iniziale;
      inAttesa=0;
    }

    public synchronized void down()
    {
      if(stato==0)
      {
            inAttesa++;
            try
            {
                  wait();
            }
            catch(InterruptedException eccezione)
            {
                  inAttesa--;
            }
      }
      else stato--;
    }

    public synchronized void up()
    {
      if(inAttesa>0)
      {
            inAttesa--;
            notify();
      }
      else stato++;
    }
}
```

I semafori che vengono settati ad un valore iniziale per la variabile `stato_iniziale` pari ad 1 si dicono semafori binari. Avendo introdotto i semafori possiamo adesso provare a risolvere il

problema del produttore-consumatore. Nella soluzione che propongo ho usato tre semafori: `mutex` (per garantire l'accesso in mutua esclusione all'array `buffer`, si tratta di un semaforo binario); `spazi_vuoti` (semaforo la cui variabile `stato`, inizializzata a 100, indica le celle ancora vuote nell'array); `spazi_pieni` (semaforo la cui variabile `stato`, inizializzata a 0, indica le celle dell'array già riempite).
Il semaforo `spazi_vuoti` indicherà al processo produttore il momento in cui fermare la produzione. Infatti, un eventuale `down()` su questo semaforo avrà l'effetto di decrementare la variabile `stato` (inizializzata a 100, capienza massima del buffer) finquanto questa non raggiunge il valore 0 (nella realtà il processo produttore si fermerà prima poichè verrà schedulato dal sistema e subentrerà il processo consumatore che inizierà a prelevare gli elementi dal buffer, se presenti). In tal caso (lo si può vedere dal codice del semaforo) il processo produttore si fermerà sulla `wait()`.
Il semaforo `spazi_pieni` indicherà al processo consumatore la presenza di elementi nell'array che quindi saranno prelevati. Questi, se assenti, possono causare la sospensione del processo consumatore (che effettuando una `down` sul semaforo `spazi_pieni` si fermerà sulla `wait()`).
Il semaforo `mutex` è usato, invece, per ottenere la mutua esclusione tra il processo produttore ed il processo consumatore. Il processo produttore che fa una `down()` sul semaforo `mutex` non viene bloccato e va oltre (accedendo al buffer per depositare un elemento) se lo stato di tale semaforo ne permette l'esecuzione (il produttore trova, cioè, `stato=1`). Se, invece, il processo consumatore sta in quel momento accedendo al buffer, una precedente `down` sul semaforo `mutex` farà aspettare il processo produttore. I semafori `spazi_pieni` e `spazi_vuoti` sono, in questo caso, usati per ottenere la sincronizzazione fra i due processi. Essi garantiscono che certe sequenze di eventi si verifichino oppure no in un certo ordine. In questo caso assicurano che il produttore si fermi quando il buffer è pieno e che il consumatore si fermi quando il buffer è vuoto.

```
import java.lang.Thread.*;

public class Produttore_Consumatore
{
	public static void main(String args[])
	{
		for(int i=0;i<buffer.length;i++) buffer[i]=0;
		P.start();
		C.start();
	}

	static int buffer[]=new int[10];
	static Semaforo mutex=new Semaforo(1);
	static Semaforo spazi_vuoti=new Semaforo(10);
	static Semaforo spazi_pieni=new Semaforo(0);
	static Produttore P=new Produttore();
	static Consumatore C=new Consumatore();


	static class Produttore extends Thread
	{
		public void run()
		{
			int elemento;
			while(true)
			{
				elemento=(int)(Math.random()*10);
				spazi_vuoti.down();
				mutex.down();
				buffer[spazi_pieni.stato]=elemento;
				System.out.println("Produttore:aggiunto "+elemento);
				mutex.up();
				spazi_pieni.up();
			}
		}
```

```
	}

	static class Consumatore extends Thread
	{
		public void run()
		{
			int elaborazione=0;
			while(true)
			{
				spazi_pieni.down();
				mutex.down();
				elaborazione=elaborazione+buffer[spazi_pieni.stato];
				System.out.println("Consumatore: "+elaborazione);
				mutex.up();
				spazi_vuoti.up();
			}
		}
	}
}
```

Ora il problema del produttore-consumatore è risolto, tuttavia, in fase di programmazione emergono diverse difficoltà che meritano una certa attenzione. La programmazione con i semafori non è mai perfetta è può dare luogo a fenomeni di stallo detti *deadlock*. Quando tutti i processi si bloccano si ha un deadlock, il programma rimane bloccato per sempre senza tra l'altro svolgere alcun lavoro.
Un esempio di deadlock è ad esempio quello che si potrebbe verificare se in fase di programmazione del problema produttore-consumatore scambiamo l'ordine delle chiamate a `down()` nel processo produttore (in altre parole facciamo prima `down()` su `mutex` e poi `down()` su `spazi_vuoti`). Ecco cosa succede: il produttore accede per prima cosa al buffer condiviso in mutua esclusione facendo una `down()` su `mutex`. Quindi vede se il buffer è vuoto, in caso affermativo inserisce un elemento, se il buffer è invece già pieno il produttore ha chiesto inutilmente l'accesso al buffer, non solo! Esso, infatti, rimarrà bloccato sulla `down()` fatta sul semaforo `spazi_vuoti` non essendoci spazi liberi per l'inseririmento dell'elemento prodotto e non rilasciando il buffer non darà la possibilità al processo consumatore di consumare un elemento del buffer per fare spazio all'elemento nuovo. Anche il processo consumatore rimarrà quindi bloccato, così come il processo produttore. In definitiva l'intero programma è bloccato, si è verificato un deadlock!

```
	static class Produttore extends Thread
	{
		public void run()
		{
			int elemento;
			while(true)
			{
				elemento=(int)(Math.random()*10);
				mutex.down();                          /* DEADLOCK */
				spazi_vuoti.down();                    /* DEADLOCK */
				buffer[spazi_pieni.stato]=elemento;
				System.out.println("Produttore:aggiunto "+elemento);
				mutex.up();
				spazi_pieni.up();
			}
		}
	}
```

Per semplificare la programmazione ed agevolare la concorrenza fra i programmi è stata introdotta una nuova primitiva chiamata *monitor*. Un monitor è l'insieme di variabili, strutture dati e collezione di procedure/metodi condensati in unico tipo. Nei monitor deve essere garantita la mutua esclusione fra le procedure appartenenti al tipo definito, queste poi devono garantire l'accesso alle strutture dati. I processi che usano i monitor non accedono direttamente alla variabile o alla struttura dati condivisa, essi si limitano a farlo attraverso le procedure che il monitor mette a disposizione.

Quando un processo invoca un metodo di un monitor questo a sua volta (sempre il monitor) controlla che nessun'altro processo è al momento attivo nel monitor. Solo in caso affermativo può continuare la procedura, in caso contrario il processo è infatti sospeso. Per permettere la sospensione del processo si fa ricorso a variabili di tipo condizione ed all'uso di due operazioni dette `wait()` e `signal()`. Pertanto, se una procedura invocata scopre che un altra procedura del monitor è già in esecuzione questa si sospende con una chiamata a `wait()` fatta su una variabile di tipo condizione. In questo modo, quando una `signal()` fatta ad una variabile di tipo condizione ne sblocca lo stato la precedente procedura può riprendere. In altre parole la `signal(X)` sveglia uno dei processi in coda su `X`. Se nessun processo è in attesa la `signal()` viene persa. Questo obbliga ad un ordine ben preciso delle chiamate: una `wait()` deve avvenire prima di una `signal()`. Ma allora le primitive viste per implementare un semaforo, `sleep()` e `wakeup()`, sono simili alle primitive usate per l'implementazione dei monitor, `wait()` e `signal()`?
In effetti se pur simili queste primitive hanno una sottile ma importante differenze. Il problema delle primitive `sleep()` e `wakeup()` era dovuto al fatto che mentre un processo provava a sospendersi (ma poi veniva schedulato, ricordate?) l'altro provava a svegliarlo (e non ci riusciva perchè il processo non si era ancora sospeso!), con i monitor viene invece garantita l'atomicità delle procedure (una sola procedura è in esecuzione, vedremo che in java ciò richiederà dei metodi synchronized). In questo caso una `wait()` avrà la certezza di terminare. La soluzione del problema del produttore-consumatore può essere trovata nell'utilizzo dei monitor. Una struttura dati di tipo monitor modella il buffer, le variabili dedicate alla disponibilità del buffer nonchè la prima posizione vuota nell'array e quella utile alla lettura di un elemento. Al monitor si accompagnano poi due metodi synchronized per le operazioni di inserimento e rimozione di un elemento. La procedura per l'inserimento è esclusivamente usata dal processo produttore, quella relativa alla rimozione, invece, è interamente dedicata al processo consumatore.
La procedura `inserisciElemento()` verifica dapprima la disponibilità di elementi nel `buffer`, se questo è pieno la procedura viene sospesa (chiamata a `sospendiAttivita()`), in caso contrario si procede inserendo un elemento nuovo nel `buffer`, aggiornando il valore di `prossima_posizione_vuota` e della variabile `disponibilita`. Quindi si sveglia il consumatore se questo era sospeso.
La procedura `rimuoviElemento()` verifica dapprima la disponibilità di elementi nel `buffer`, se questi sono assenti la procedura non può continuare e per questo motivo viene sospesa. In caso contrario si procede leggendo un valore dal `buffer`, aggiornando il valore di `prossima_posizione_elemento` e della variabile `disponibilita`. Quindi si sveglia il produttore se questo era sospeso.

```
public class Produttore_Consumatore
{
```

```
static myMonitor myBuffer=new myMonitor(10);   /* Istanzia un monitor */
static produttore p=new produttore();          /* Istanzia un prod. */
static consumatore c=new consumatore();        /* Istanzia un cons. */
public static void main(String argr[])
{
      p.start();  /* Avvio il thread produttore */
      c.start();  /* Avvio il thread consumatore */
}
static class produttore extends Thread
{
      public void run()
      {
            int elemento;
            while(true)
            {
                  elemento=(int)(Math.random()*10);
                  myBuffer.inserisciElemento(elemento);
                  System.out.println("Produttore: add["+elemento+"]");
            }
      }
}
static class consumatore extends Thread
{
      public void run()
      {
            int elemento;
            while(true)
            {
                  elemento=myBuffer.rimuoviElemento();
                  System.out.println("Consumatore: read["+elemento+"]");
            }
      }
}
static class myMonitor
{
      private final int capienza;
      private int buffer[];
      private int disponibilita;
      private int prossima_posizione_vuota;
      private int prossima_posizione_elemento;
      public myMonitor(int capienzaMassimaBuffer)
      {
            capienza=capienzaMassimaBuffer;
            buffer=new int[capienza];
            disponibilita=0;
            prossima_posizione_elemento=0;
            prossima_posizione_vuota=0;
      }
public synchronized void inserisciElemento(int nuovoElemento)
{
      if(disponibilita==capienza) sospendiAttivita();
      buffer[prossima_posizione_vuota]=nuovoElemento;
      prossima_posizione_vuota=(prossima_posizione_vuota+1)%capienza;
      disponibilita=disponibilita+1;
      if(disponibilita==1) notify();
}
public synchronized int rimuoviElemento()
{
     int elemento;
     if(disponibilita==0) sospendiAttivita();
     elemento=buffer[prossima_posizione_elemento];
     prossima_posizione_elemento=(prossima_posizione_elemento+1)%capienza;
     disponibilita=disponibilita-1;
     if(disponibilita==capienza-1) notify();
```

```
            return elemento;
      }
      private void sospendiAttivita()
      {
            try
            {
                  wait();
            }
            catch(InterruptedException eccezione)
            {
                  System.out.println("Eccezione: "+eccezione);
            }
      }
   }
}
```

Esercitazione con i thread: assegnazione della risorsa cpu
Realizzare una classe processo che deve effettuare un numero fissato di elaborazioni. Per ogni elaborazione il processo thread:

- chiede l'uso della risorsa di calcolo che gli è stata assegnata (un oggetto della classe cpu);
- ottenuta la cpu la mantiene impegnata per un numero random di millisecondi;
- rilascia la risorsa cpu al termine dell'elaborazione;

La classe processo deve avere come campi dell'istanza il numero di esecuzioni che il thread dovrà effettuare e la risorsa cpu che gli è stata assegnata. La risorsa di calcolo è un oggetto della classe cpu ed è caratterizzata da un numero massimo di thread contemporaneamente in esecuzione. Scrivere due metodi synchronized che regolano l'accesso ed il rilascio della risorsa di calcolo. Quindi, testare con un programma principale la validità delle precedenti classi istanziando una risorsa di calcolo ed assegnondo la stessa a più thread della classe processo.

```
public class myCPU
{
    private int maxThread;
    private int currentThread;
    public myCPU(int maxThreadEseguibili)
    {
         maxThread=maxThreadEseguibili;
         currentThread=0;
    }
    public synchronized boolean accedi(String nomeProcesso)
    {
      if(currentThread<maxThread)
      {
         currentThread++;
         System.out.println("Thread "+nomeProcesso+": cpu["+currentThread+"]");
         return true;
      }
      else return false;
    }
    public synchronized void rilascia(String nomeProcesso)
    {
      if(currentThread>0)
      {
         currentThread--;
         System.out.println("Thread "+nomeProcesso+": exit["+currentThread+"]");
      }
    }
}
public class processo extends Thread
{
```

```
      private int numAzioni;
      private myCPU processore;
      private String nomeProcesso;
      public processo(String nomeProcesso,int numAzioni,myCPU processore)
      {
            this.nomeProcesso=nomeProcesso;
            this.numAzioni=numAzioni;
            this.processore=processore;
      }
      public void run()
      {
            while(numAzioni!=0)
            {
                  if(processore.accedi(nomeProcesso)==true)
                  {
                        try
                        {
                        sleep((int)(Math.random()*100));
                        processore.rilascia(nomeProcesso);
                        numAzioni--;
                        }
                        catch(InterruptedException eccezioneA)
                        {
                              System.out.println("Eccezione: "+eccezioneA);
                        }
                  }
                  else
                  {
                        try
                        {
                              sleep(2000);
                        }
                        catch(InterruptedException eccezioneB)
                        {
                              System.out.println("Eccezione: "+eccezioneB);
                        }
                  }
            }
            System.out.println("Processo "+nomeProcesso+": quit");
      }
}

public class testCPU
{
    public static void main(String arg[])
    {
      myCPU processore=new myCPU(2);
      processo p1=new processo("smss.exe",5,processore);
      processo p2=new processo("firefox.exe",7,processore);
      processo p3=new processo("bdserv.exe",3,processore);
      processo p4=new processo("acroread.exe",4,processore);
      processo p5=new processo("iexplore.exe",3,processore);
      p1.start();
      p2.start();
      p3.start();
      p4.start();
      p5.start();
    }
}
```

Esercitazione con i thread: il problema dei lettori e dei scrittori

Altro problema classico per la programmazione concorrente è quello dei lettori e scrittoi. Il problema prevede che *m* processi lettori ed *n* processi scrittori possano richiedere l'accesso ad una struttura dati condivisa, un buffer di memoria ad esempio. Ai lettori viene concesso l'accesso contemporaneamente ad altri lettori poichè questi non modificano il buffer di memoria ma si limitano a leggerne i valori. L'accesso per uno scrittore, invece, deve avvenire in mutua esclusione rispetto ad un altro processo scrittore e/o lettore. Implementare in java una soluzione al problema facendo uso di una variabile condivisa ed utilizzando i semafori.

Prima soluzione (con starvation per i processi scrittori)

```
public class lettore extends Thread
{
	private static int numLettori=0;
	private static Semaforo mutex=new Semaforo(1);
	private static int nextID=0;
	private int ID;
	public Semaforo semaforoRisorsa;
	private int numAzioni;
	public int buffer[];

	public lettore(int numAzioni,Semaforo x,int[] risorsaCondivisa)
	{
		ID=nextID;
		nextID++;
		this.numAzioni=numAzioni;
		semaforoRisorsa=x;
		buffer=risorsaCondivisa;
	}
	public void run()
	{
		while(numAzioni!=0)
		{
			mutex.down();
			numLettori++;
			if(numLettori==1) semaforoRisorsa.down();
			mutex.up();
			leggi(buffer);
			mutex.down();
			numLettori--;
			if(numLettori==0) semaforoRisorsa.up();
			mutex.up();
			numAzioni--;
		}
		System.out.println("Lettore ["+ID+"]: operazioni terminate...");
	}
	public void leggi(int[] risorsaCondivisa)
	{
		System.out.println("Lettore ["+ID+"]: lettura["+
		risorsaCondivisa[((int)(Math.random()*10))]+"]");
	}
}
```

```
public class scrittore extends Thread
```

```
{
	private static int nextID=0;
	private int ID;
	private int numAzioni;
	public Semaforo semaforoRisorsa;
	public int buffer[];

	public scrittore(int numAzioni,Semaforo x,int[] risorsaCondivisa)
	{
		this.numAzioni=numAzioni;
		buffer=risorsaCondivisa;
		semaforoRisorsa=x;
		ID=nextID;
		nextID++;
	}
	public void run()
	{
		while(numAzioni!=0)
		{
			semaforoRisorsa.down();
			scrivi(buffer);
			semaforoRisorsa.up();
			numAzioni--;
		}
		System.out.println("Scrittore ["+ID+"]: operazioni terminate...");
	}
	private void scrivi(int[] risorsaCondivisa)
	{
		int elemento=(int)(Math.random()*100);
		risorsaCondivisa[(int)(Math.random()*10)]=elemento;
		System.out.println("Scrittore ["+ID+"]: scrittura effettuata ["
		+elemento+"]");
	}
}

public class testLettoriScrittori
{
	public static void main (String[] args)
	{
		Semaforo semaforoRisorsaCondivisa=new Semaforo(1);
		int buffer[]=new int[10];
		scrittore s1=new scrittore(10,semaforoRisorsaCondivisa,buffer);
		scrittore s2=new scrittore(8,semaforoRisorsaCondivisa,buffer);
		lettore l1=new lettore(20,semaforoRisorsaCondivisa,buffer);
		lettore l2=new lettore(15,semaforoRisorsaCondivisa,buffer);
		lettore l3=new lettore(22,semaforoRisorsaCondivisa,buffer);
		lettore l4=new lettore(16,semaforoRisorsaCondivisa,buffer);
		lettore l5=new lettore(19,semaforoRisorsaCondivisa,buffer);
		lettore l6=new lettore(10,semaforoRisorsaCondivisa,buffer);
		s1.start();
		l1.start();
		l2.start();
		l3.start();
		s2.start();
		l4.start();
		l5.start();
		l6.start();
	}
}
```

La soluzioni qui proposta soffre di un problema noto come *starvation*, morte per fame. Il problema è il seguente: finquanto ci sarà un processo lettore che è intenzionato a leggere dal buffer di memoria, un processo scrittore che effettuerà una `down()` sul semaforo `semaforoBuffer` rimarrà

qui bloccato in attesa. Quindi se arrivano con una certa frequenza delle richieste di lettura da diversi processi letrori un processo scrittore non accederà mai al buffer per scriverci (*startvation*). Una soluzione più ragionevole è quella che mette un processo scrittore in attesa dei soli processi lettori che al momento stanno già leggendo sul buffer, tutti gli altri processi lettori dovranno aspettare i processi scrittori. In altre parole, un nuovo lettore non può acquisire un elemento dal buffer se vi è uno scrittore in attesa. Ecco quindi il codice che risolve il problema della starvation, in grassetto sono segnate le istruzioni aggiunte al precedente codice.

Seconda soluzione (senza starvation per i processi scrittori)

```
public class lettore extends Thread
{
	private static int numLettori=0;
	private static Semaforo mutex=new Semaforo(1);
	private static int nextID=0;
	private int ID;
	public Semaforo semaforoRisorsa;
	public Semaforo attendiScrittore;
	private int numAzioni;
	public int buffer[];

	public lettore(int numAzioni,Semaforo x,int[] risorsaCondivisa,Semaforo y)
	{
		ID=nextID;
		nextID++;
		this.numAzioni=numAzioni;
		semaforoRisorsa=x;
		attendiScrittore=y;
		buffer=risorsaCondivisa;
	}
	public void run()
	{
		while(numAzioni!=0)
		{
			attendiScrittore.down();
			attendiScrittore.up();
			mutex.down();
			numLettori++;
			if(numLettori==1) semaforoRisorsa.down();
			mutex.up();
			leggi(buffer);
			mutex.down();
			numLettori--;
			if(numLettori==0) semaforoRisorsa.up();
			mutex.up();
			numAzioni--;
		}
		System.out.println("Lettore ["+ID+"]: operazioni terminate...");
	}
	public void leggi(int[] risorsaCondivisa)
	{
		System.out.println("Lettore ["+ID+"]: lettura["+
		risorsaCondivisa[((int)(Math.random()*10))]+"]");
	}
}




public class scrittore extends Thread
{
```

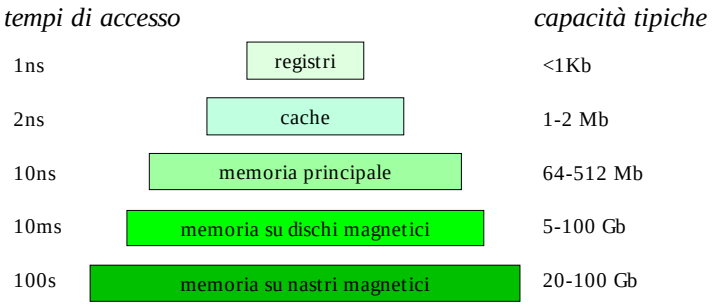
```
	private static int nextID=0;
	private int ID;
	private int numAzioni;
	public Semaforo semaforoRisorsa;
	public Semaforo attendiScrittore;
	public int buffer[];

	public scrittore(int numAzioni,Semaforo x,int[] risorsaCondivisa,Semaforo y)
	{
		this.numAzioni=numAzioni;
		buffer=risorsaCondivisa;
		semaforoRisorsa=x;
		attendiScrittore=y;
		ID=nextID;
		nextID++;
	}
	public void run()
	{
		while(numAzioni!=0)
		{
			attendiScrittore.down();
			semaforoRisorsa.down();
			scrivi(buffer);
			semaforoRisorsa.up();
			attendiScrittore.up();
			numAzioni--;
		}
		System.out.println("Scrittore ["+ID+"]: operazioni terminate...");
	}
	private void scrivi(int[] risorsaCondivisa)
	{
		int elemento=(int)(Math.random()*100);
		risorsaCondivisa[(int)(Math.random()*10)]=elemento;
		System.out.println("Scrittore ["+ID+"]: scrittura effettuata ["
		+elemento+"]");
	}
}

public class testLettoriScrittori
{
	public static void main (String[] args)
	{
		Semaforo semaforoRisorsaCondivisa=new Semaforo(1);
		int buffer[]=new int[10];
		scrittore s1=new scrittore(10,semaforoRisorsaCondivisa,buffer);
		scrittore s2=new scrittore(8,semaforoRisorsaCondivisa,buffer);
		lettore l1=new lettore(20,semaforoRisorsaCondivisa,buffer);
		lettore l2=new lettore(15,semaforoRisorsaCondivisa,buffer);
		lettore l3=new lettore(22,semaforoRisorsaCondivisa,buffer);
		lettore l4=new lettore(16,semaforoRisorsaCondivisa,buffer);
		lettore l5=new lettore(19,semaforoRisorsaCondivisa,buffer);
		lettore l6=new lettore(10,semaforoRisorsaCondivisa,buffer);
		s1.start();
		l1.start();
		l2.start();
		l3.start();
		s2.start();
		l4.start();
		l5.start();
		l6.start();
	}
}
```

Il semaforo `attendiScrittore` aggiunto ad entrambi i processi ha l'effetto di fermare i processi

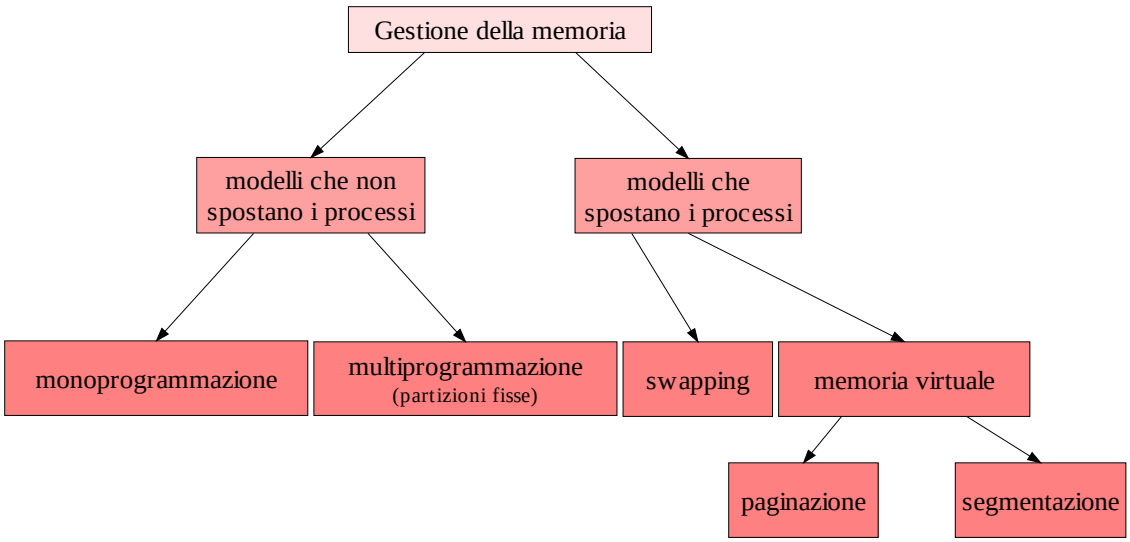
lettori. Infatti, se nessuno scrittore è in attesa le due istruzioni `attendiScrittore.down()` ed `attendiScrittore.up()` vengono rapidamente svolte da un processo lettore. Al contrario, invece, se un processo scrittore è interessato alla scrittura farà dapprima una chiamata ad `attendiScrittore.down()` e metterà in attesa i successivi lettori, lasciando in esecuzione quelli che già hanno avuto accesso alla risorsa.

La gestione della memoria

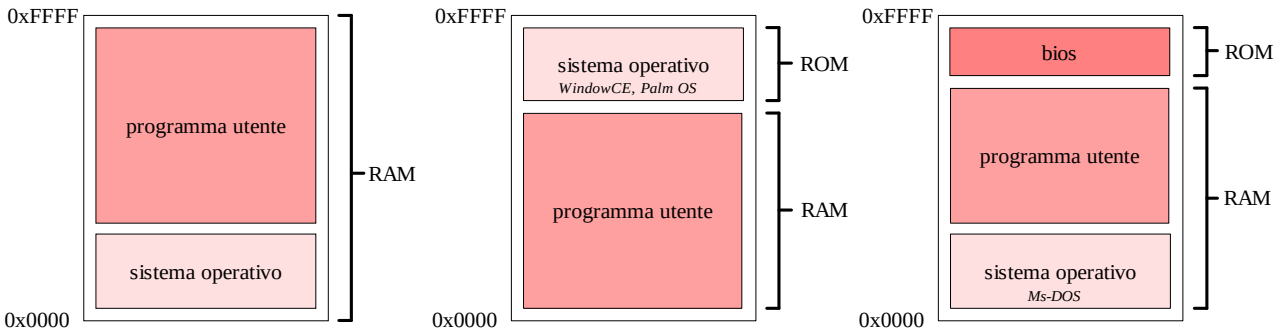
Negli ultimi anni la programmazione è cresciuta molto, ce ne rendiamo conto osservando le dimensioni medie dei programmi. Questi adesso hanno più pretese in termini di memoria e di computazione, tuttavia, a tale aumento non corrisponde un aumento alla pari delle memorie che pure hanno avuto e tuttora stanno avendo una profonda e significativa evoluzione. Ogni risorsa condivisa che si rispetti necessita di un gestore per l'allocazione della risorsa e la risoluzione delle contese e la memoria non fa eccezioni in questo caso. Il gestore della memoria è quella parte del sistema operativo che si occupa dell'allocazione della memoria ai processi richiedenti e della deallocazione degli indirizzi di memoria non più utilizzati. A queste due funzioni che, se vogliamo, descrivono i comportamenti più estremi del gestore della memoria (associate agli eventi creazione/esecuzione di un processo e sua terminazione) se ne aggiunge uno intermedio: il gestore della memoria deve talvolta concordare lo scambio fra la memoria principale e la memoria di massa (disco fisso per molti) quando la memoria principale non è sufficientemente grande per ospitare tutti i processi in esecuzione in memoria.
Idealmente una memoria dovrebbe essere veloce (la velocità è intesa come velocità di scrittura e lettura), capiente (in pochi anni si è passati a memorie principali dell'ordine di qualche decina di Mb a qualche Gb), non volatile (il contenuto della memoria rimane cioè fissato in memoria se l'alimentazione elettrica al sistema viene sospesa) ed anche poco costosa. Le tecnologie del momento non permettono di soddisfare contemporaneamente a tutti i requisiti (quando anche i primi tre requisiti raggiungono un livello soddisfacente in termini di prestazione è il prezzo ad allontanare l'utente) e di conseguenza la maggior parte dei sistemi per l'elaborazione adotta una gerarchia di memoria. La filosofia che è alla base consiste nel portare verso il vertice di tale gerarchia le informazioni più utilizzate dalla porzione di programma in esecuzione cosicchè nella maggior parte dei casi il tempo di accesso coincide con quello delle memorie più rapide.



I modelli di gestione della memoria si sono evoluti con il passare del tempo, adesso i vecchi modelli vengono ugualmente riciclati e applicati ai nuovi dispositivi e/o periferiche per cui vale la pena studiarli ancora. Inutile dire che un buon algoritmo di gestione della memoria può caratterizzare l'efficienza del gestore della memoria e ciò verrà quindi riflesso sulle prestazioni complessive del sistema operativo. I sistemi di gestione della memoria possono essere suddivisi in due categorie: gestori della memoria che non spostano i processi dalla memoria principale una volta iniziata l'esecuzione e gestori della memoria che spostano un proceso in esecuzione dalla memoria principale alla memoria di massa. Nella figura che segue è possibile vedere le suddivisioni applicate ai modelli di gestione della memoria che qui tratteremo.

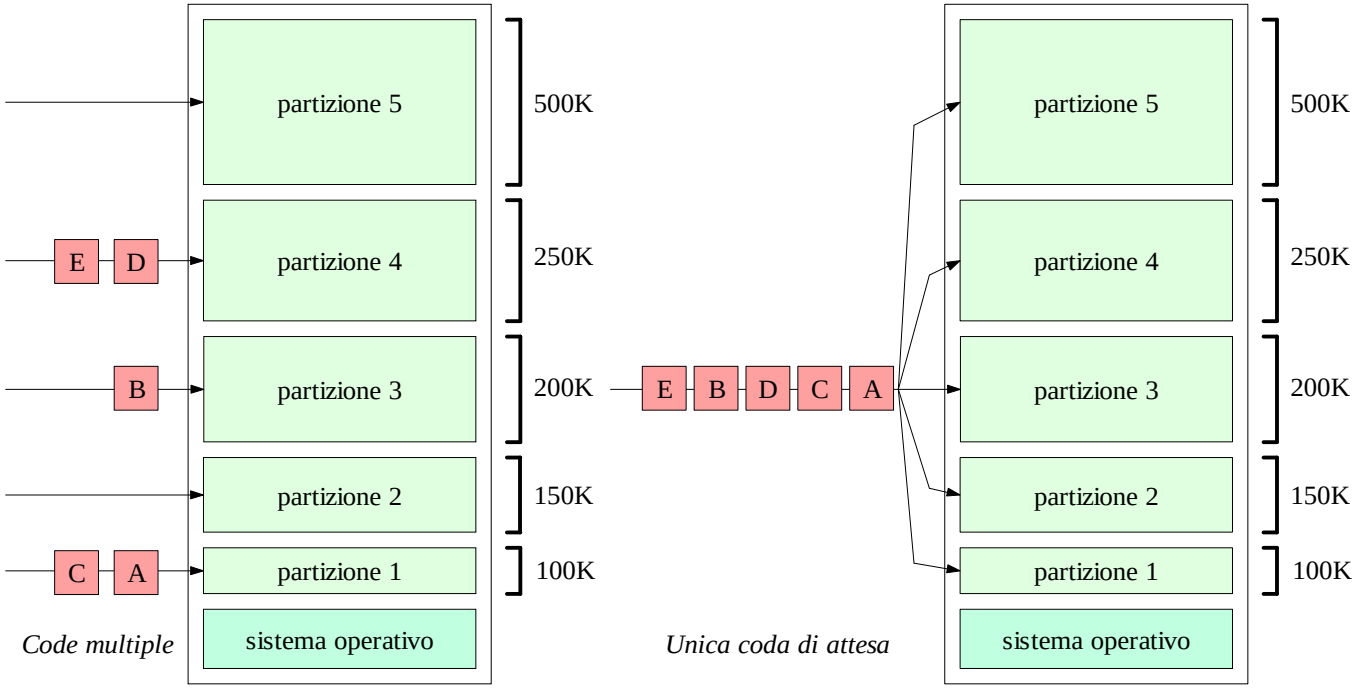


Modelli che non spostano i processi in esecuzione
Il modello più semplice è quello orientato alla monoprogrammazione (ormai in disuso a causa dell'affermarsi della multiprogrammazione), il sistema esegue un solo programma alla volta e per questo motivo l'area di memoria è condivisa dal processo in esecuzione e dal sistema operativo. Talvolta il sistema operativo può occupare un area ROM della memoria e lasciare la restante parte di memoria (non ROM) all'esecuzione dei processi utenti (tale soluzione è ad esempio adottata nei sistemi embedded e nei palmari). Uno scenario alternativo ai primi due appena visti consiste nel collocare nell'area ROM della memoria i gestori dei dispositivi di input/output e di suddividere la restante parte tra il sistema operativo ed il processo in esecuzione (questa soluzione è ad esempio impiegata da MS-Dos, la porzione di memoria ROM è detta BIOS e sta per binary input/output system).



La monoprogrammazione, fortunatamente, è solo un ricordo del passato adesso utile per alcuni dispositivi (il concetto di riciclo e riusabilità è sempre esistito in informatica), tuttavia la possibilità di implementare la multiprogrammazione è davvero a pochi passi. L'area di memoria da allocare ai processi in esecuzione può essere suddivisa in partizioni fisse di diverse dimensioni (ad esempio, un area di memoria principale da 1 Mb può essere suddivisa in quattro partizioni rispettivamente di 100, 200, 300 e 400 Kb). Tali partizioni si possono ad esempio generare in fase di inizializzazione del dispositivo oppure sotto i comandi impartiti dall'utente. Ogni partizione può ospitare un solo programma in esecuzione. Sono possibili due alternative o varianti, implementare un unica coda di attesa per ogni partizione di memoria oppure allestire un unica coda di attesa per tutti i processi. La prima strategia (una coda di attesa per ognuna delle partizioni di memoria) penalizza i processi in attesa su una coda quando in memoria esistono partizioni di memoria più grandi ma vuote. Nel secondo caso, invece, i processi sono tutti nella stessa coda di attesa: il processo in testa alla coda è il primo ad essere allocato (se ciò è possibile compatibilmente con l'attuale stato della memoria), per esso si sceglierà la partizione di memoria che più si avvicina alle dimensioni del processo. Qualora non ci fosse spazio a sufficienza il gestore della memoria cerca di allocare le partizioni di

memoria rimaste ai successivi processi in coda. A seguito della terminazione di un processo, il gestore della memoria potrebbe procedere cercando nella coda di attesa il processo che meglio riempie il buco appena lasciato. In tal caso il gestore della memoria trascurerà i processi di dimensione più piccola che quindi attendono più tempo. Una valida alternativa è invece quella di colmare le partizioni di memoria appena liberate con il processo in coda più piccolo. Questo avvantaggia i processi piccoli che sono quasi sempre di tipo interattivo e quindi meritevoli di un servizio migliore se si vuole dare all'utente una buona impressione circa le prestazioni del sistema.



La multiprogrammazione tiene impegnata, tra l'altro, di più la cpu e ciò non può che apportare un incremento delle prestazioni del sistema (sostanzialmente verranno svolti più job). In un modello per la multiprogrammazione, nell'ipotesi di *n* processi indipendenti, si quantifica l'utilizzo della cpu mediante la seguente formula:

$$Utilizzo\ della\ CPU = 1 - p^n$$

dove *p* è la percentuale di attesa media per l'input/output, *n* numero di processi è anche detto grado di multiprogrammazione. Con questo modello il proprietario di un sistema potrebbe decidere se una spesa di memoria aggiuntiva sia vantaggiosa oppure no. Supponiamo ad esempio che un sistema disponga di 256 Mb di memoria principale e che il sistema operativo ne richieda 64 Mb. Rimangono, quindi, 192 Mb di memoria principale liberi ed essendo di 20 Mb la dimensione media dei processi in esecuzione ciò significa che il grado di multiprogrammazione è di 9 processi (192/20). I processi sono, poi, nell'85% dei casi in attesa di operazioni di input/output per cui la percentuale di utilizzo della cpu si aggira intorno al 76%:

$$Utilizzo\ della\ CPU = 1 - 0.7^9 = 0.76$$

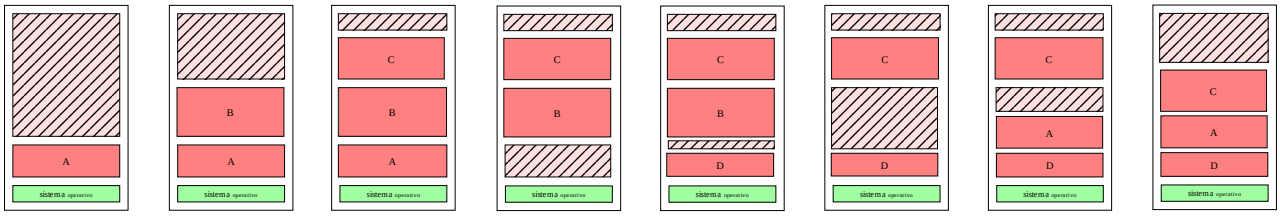
Un aggiunta di 256 Mb di memoria principale comporta la possibilità di eseguire altri 12 processi in multiprogrammazione (256/20), in altre parole il grado di multiprogrammazione è adesso di 21 processi (i precedenti 9 processi sommati agli attuali 12 processi aggiunti) per cui la percentuale di utilizzo della cpu è adesso del 99%:

$$Utilizzo\ della\ CPU = 1 - 0.7^{21} = 0.99$$

Il proprietario del sistema ha, speso bene i suoi soldi.

Modelli che spostano i processi in esecuzione

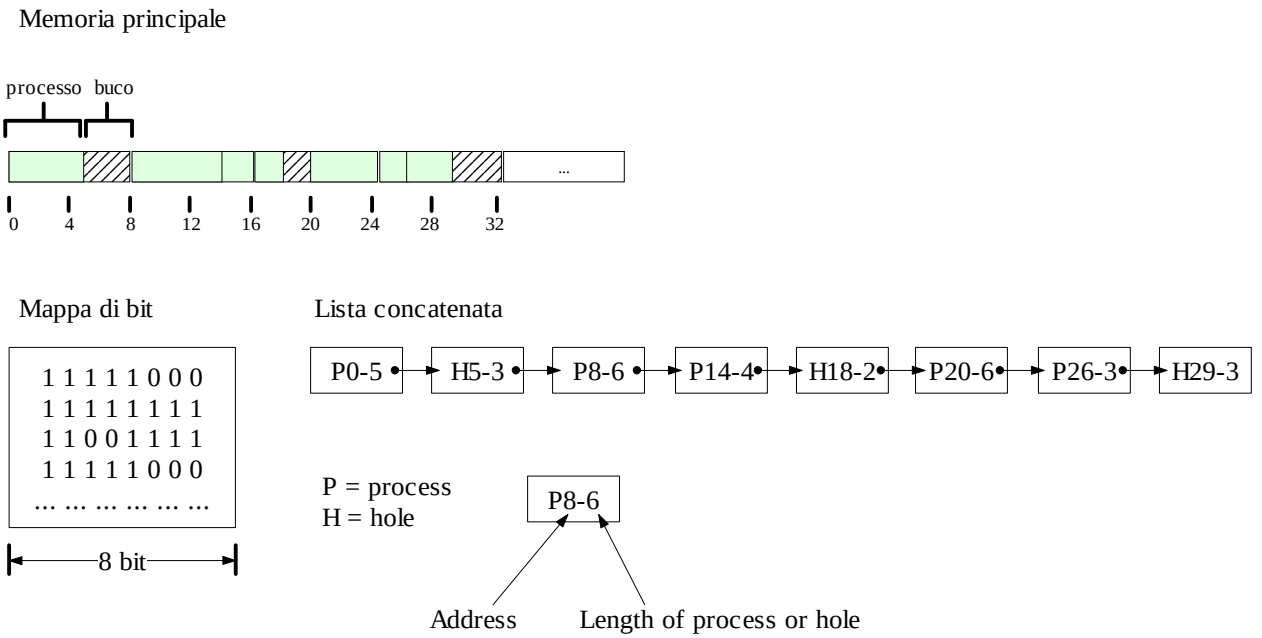
Quando la memoria principale di un sistema è sufficientemente grande da poter ospitare più processi in esecuzione non c'è motivo di adottare un modello diverso dal precedente. Tuttavia la memoria non è infinita ed in alcuni sistemi può addirittura non bastare lo spazio che questa mette a disposizione dei processi. Per questo motivo esiste un diverso modello per la gestione della memoria che è orientato allo spostamento dei processi in esecuzione che, quando occorre spazio libero, vengono temporaneamente scaricati dalla memoria principale al disco fisso. Appartengono a questo modello la tecnica dello swapping e quella della memoria virtuale (paginazione e segmentazione). Nello swapping un processo è interamente caricato in memoria (non ci sono partizioni fisse); quindi viene eseguito per un certo intervallo di tempo ed infine è scaricato dalla memoria principale al disco fisso. Anche i processi creati successivamente al primo vengono allocati interamente in memoria, tuttavia se lo spazio rimasto è insufficiente il processo non può essere caricato in memoria e per questo motivo viene ucciso. Quando un processo in memoria principale termina (eseguendo l'ultima istruzione oppure perchè ucciso da un altro processo) gli indirizzi di memoria ad esso allocati, e quindi lo spazio utilizzato, ritornano ad essere disponibili per le successive allocazioni. Questo genera diversi buchi in memoria che è possibile compattare in unico buco. Tuttavia la compattazione richiede tempo e spesse volte non viene invocata (se ad esempio un sistema dispone di 512 Mb di memoria principale e servono 30 ns per ricopiare 8 byte occorreranno 16 secondi per effettuare la compattazione della memoria!).



creazione di A | creazione di B | creazione di C | sospensione di A | creazione di D | sospensione di B | ripresa di A | compattazione

Altro problema dello swapping (il primo è dunque la compattazione) è la quantità di memoria che deve essere allocata per i processi. Se i processi in esecuzione sul sistema non hanno una crescita (occupano cioè sempre lo stesso spazio) in termini di spazio utile al processo ad essi viene allora allocato l'esatto quantitativo di memoria richiesto. Se, invece, i processi, come spesso accade, subiscono una crescita dell'area dati oppure dell'area stack si decide di allocarli in memoria principale assegnando per essi un quantitativo di memoria in più. Se durante l'esecuzione del processo lo spazio di memoria che intercorre fra l'area dati e l'area stack si esaurisce il processo deve essere rilocato in un buco di memoria sufficientemente grande da contenerlo. Se tale buco non esiste in memoria il processo viene scaricato dalla memoria principale al disco fisso, in attesa che lo spazio necessario si renda disponibile. In alternativa il processo può anche essere ucciso.

Ed ancora, altro problema dello swapping è la gestione degli spazi liberi di memoria. Quando un processo deve essere allocato in memoria va individuata per questo la giusta collocazione in memoria. Gli indirizzi di memoria vuoti, per questo motivo, possono essere ricordati da apposite strutture dati, alcune di queste sono le mappe di memoria e le liste concatenate. Nelle mappe di memoria la memoria principale è vista come una grande matrice, ogni cella rappresenta una singola unità di allocazione. E' importante capire che una piccola unità di allocazione genererebbe una mappa di memoria assai più grande di una che invece è stata costruita scegliendo un unità di allocazione più grande. Tuttavia a seguito di un allocazione di memoria ad un processo la prima mappa di bit (quella cioè con l'unità di allocazione più piccola) riesce a coprire meglio lo spazio richiesto dal processo poichè riduce al minimo gli spazi parzialmente inutilizzati dall'ultima cella allocata. In una mappa di bit le unità di allocazione già impegnate sono segnate con il simbolo 1, quello invece libere sono contraddistente dal simbolo 0.



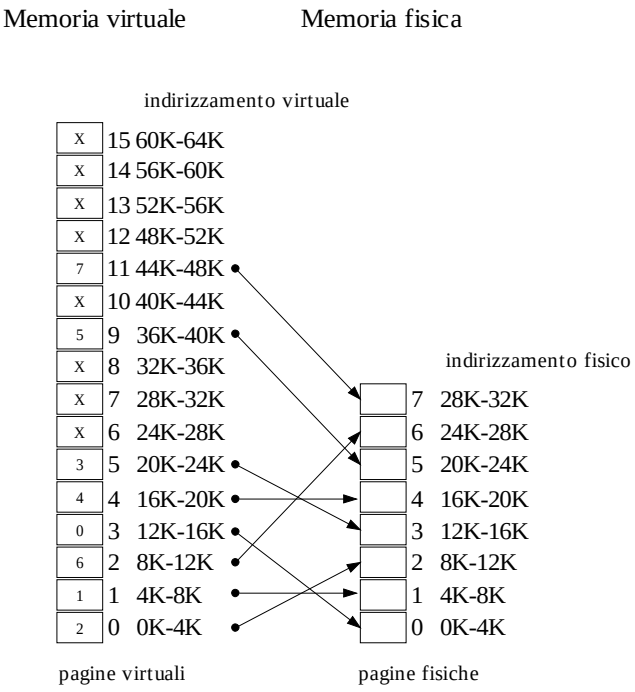
Lo stato della memoria può essere fotografato bene anche da una lista concatenata. Ogni elemento della lista può fare riferimento ad un processo (P) oppure ad un buco(H), esso indicherà inoltre l'indirizzo di partenza del processo o del buco in memoria e la lunghezza impiegata. Pertanto nella mappa dei bit l'allocazione in memoria di un nuovo processo si traduce nella ricerca di un certo numero di elementi 0 consecutivi (se questi esistono); nella lista concatenata la ricerca comporta lo scorrimento di tutta la lista fino a trovare il giusto buco in memoria. In entrambi i casi (mappa oppure lista), quindi, deve essere analizzata tutta la struttura dati prima di ricavarne lo spazio libero. Per questo motivo può risultare utile organizzare la lista concatenata in diverso modo, ad esempio tenendo separate le liste dedicate ai processi e quelle dedicate ai buchi (magari anche ordinandoli per dimensione). Esistono così diversi algoritmi per l'allocazione della memoria principale ad un processo creato oppure ricaricato dal disco:

- *first fit*, nel first fit (primo posto sufficiente) il gestore della memoria scorre la lista fino a quando non trova un buco abbastanza grande per poterci mettere un processo (lo spazio che dal buco trovato rimane viene rimesso nella lista dei buchi);
- *next fit*, nel next fit (prossimo posto sufficiente) l'algoritmo memorizza la posizione del buco precedentemente trovato ed inizia da questo la ricerca di un nuovo buco di memoria da allocare;
- *best fit*, nel best first (miglior posto sufficiente) l'algoritmo scorre tutta la lista concatenata per poi decidere quale sia il miglior buco trovato che più si addice al processo;
- *worst fit*, nell'algoritmo worst fit si decide di allocare al processo il peggior buco esistente in memoria cosicchè lo spazio liberato che rimane dall'allocazione può essere ancora utile alle successive allocazioni;
- *quick fit*, mantiene le liste separate organizzandole in base alle partizioni di memoria più richieste;

L'alternativa allo swapping è la tecnica della memoria virtuale. Il programma viene suddiso in tanti bocconi detti *overlay*, alcuni di questi stanno in memoria princpale altri invece si trovano sul disco fisso e sono caricati solo quando servono. Quindi la dimensione combinata di programma, dati e stack può eccedere la dimensione della memoria principale (detta anche memoria fisica). Sebbene il lavoro necessariao al caricamento e scaricamento venisse in realtà svolto dal sistema operarivo, il compito di dividere il programma in parti (overlay) veniva fatto dal programmatore. La tecnica della paginazione svolge in autamatico ed in modo trasparente la stessa funzione.

La memoria virtuale paginata

Molto tempo fa i programmatori dovettero fare i conti con un importante problema, far eseguire ad un sistema un programma che richiede più memoria di quella che il sistema è in grado di offrire. La soluzione che si trovò passava attraverso l'overlay (suddivisione in più sezioni) del programma sicchè solo le sezioni che al momento sono usate dal programma vengono caricate in memoria principale. Il concetto di memoria virtuale intende dare al processo la sensazione di disporre di una memoria principale assai capiente. Tuttavia la soluzione dell'overlay era solo una tecnica intermedia, ben presto all'overlay (complicato da gestire e svolto dal programmatore) si preferì (il programmatore preferì questa scelta) la tecnica della paginazione. Essa svolge in autamatico, in modo completamente trasparente per il programmatore, la stessa funzione dell'overlay. La compilazione di un programma utente genera la sua immagine in memoria sul disco fisso, questa dovrebbe poi essere proiettata in memoria quando il programma viene eseguito. Gli indirizzi generati dal programma e residenti sul disco fisso si dicono indirizzi virtuali o logici e formano lo spazio di indirizzamento virtuale o logico. Quelli, invece, della memoria principale si dicono indirizzi fisici e formano per questo motivo lo spazio di indirizzamento fisico.
In presenza di un modello per la gestione della memoria orientato alla tecnica della memoria virtuale l'invocazione di una certa istruzione all'indirizzo virtuale non corrisponde alla lettura dell'omonimo indirizzo fisico (nel caso in cui questo coincidesse con l'altro), in altre parole è necessaria una traduzione dell'indirizzo virtuale in indirizzo fisico. Un unità di gestione della memoria detta MMU (memory management unit) si occupa appunto della suddetta traduzione (mappa gli indirizzi virtuali sugli indirizzi della memoria fisica).
Affinchè sia possibile l'esecuzione di un processo che abbia uno spazio di indirizzamento virtuale più grande di quello fisico è poi necessaria una frammentazione di entrambi gli spazi di indirizzamento che solitamente avviene suddividendo gli indirizzamenti (sia virtuale che fisico) in più pagine (le pagine dei due indirizzamenti devono avere la stessa dimensione). Per questo motivo è necessario fissare la dimensione di una pagina (generalmente da 512 byte a 64 Kb) che solitamente è un multiplo della potenza di 2. Talvolta occorre trovare un buon compromesso nella dimensione di una pagina facendo in modo che questa sia capace di generare un numero intero di pagine (in questo modo non viene sprecato spazio). Le pagine dell'indirizzamento virtuale si dicono pagine virtuali o logiche, quelle dell'indirizzamento fisico si dicono pagine fisiche o frame. Supponendo un programma utente con uno spazio di indirizzamento di 64 Kb ed una memoria principale di 32 Kb e fissata la dimensione di una pagina, sia logica che fisica, di 4 Kb ritroveremo nella memoria del sistema la seguente situazione:



L'MMU, dato un indirizzo virtuale (a cui corrisponde una pagina virtuale), deve scoprire in che

pagin fisica il frame virtuale è stato mappato. Se il frame virtuale non è in memoria principale esso dovrà causare una trap di tipo *page fault* (mancanza di pagina) che verrà opportunamente gestita dal sistema operativo (tramite il *page fault handler*). L'MMU traduce effetivamente gli indirizzi virtuali in indirizzi fisici, eccone alcuni esempi. L'istruzione:

```
MOV         REG,0
```

viene tradotta dall'MMU nell'istruzione:

```
MOV         REG,8192
```

Infatti l'indirizzo virtuale `0` è mappato in memoria principale alla pagina fisica `2` (vedere l'immagine) ed essendo ogni pagina di 4096 Kb (4 Kb) questa corrisponderà all'indirizzo fisico `8192` (2·4096). L'istruzione:

```
MOV         REG,8192
```

Viene tradotta dall'MMU nell'istruzione:

```
MOV         REG,24576
```

Infatti l'indirizzo virtuale `8192` è alla pagina virtuale `2` (8192/4096) che a sua volta è mappata alla pagina fisica `6` ed essendo ogni pagina di 4096 Kb (4 Kb) questa corrisponde all'indirizzo fisico `24576` (6·4096). Un interassente esempio di traduzione è la seguente istruzione:

```
MOV         REG,20500
```

che viene tradotta dall'MMU nell'istruzione:

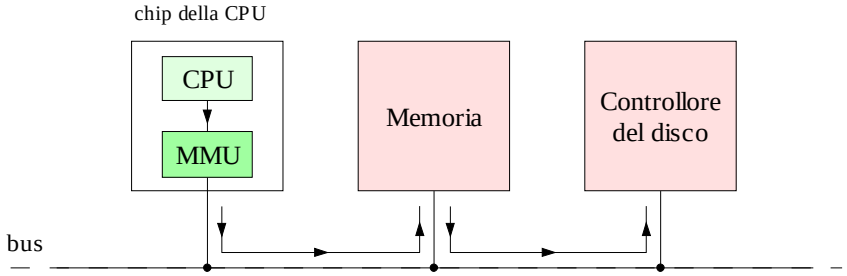
```
MOV         REG,12308
```

Infatti l'indirizzo virtuale `20500` è alla pagina virtuale `5` (20500/4096 che non è un numero intero, occorrerà aggiungere uno spiazzamento di 20 indirizzi) che a sua volta è mappata alla pagina fisica `3` ed essendo ogni pagina di 4096 Kb (4 Kb) questa corrisponde all'indirizzo fisico `12288` (3·4096) a cui però occorre aggiungere lo spiazzamento dei 20 indirizzi di offset. Pertanto l'indirizzo esatto è `12308` (12288+20). Un caso davvero singolare è la seguente istruzione:

```
MOV         REG,32780
```

che accede alla pagina virtuale `8` che non è mappata in memoria fisica (la X contraddistingue nella figura le pagine non in memoria principale). L'MMU si accorge dell'anomalia e fa in modo che la CPU esegua una trap al sistema operativo, tale passaggio è detto *fault di pagina* oppure *page fault*. Il sistema operativo, siccome la pagina appena invocata serve al programma, sceglie una delle pagine fisiche poco usate e la rimette sul disco fisso, quindi alloca la pagina fisica appena liberata alla pagine virtuale invocata dal programma e fa ripartire l'istruzione interrotta. Abbiamo precedentemente detto che la pagina, oppure il frame, deve essere dimensionata affinchè risulti un multiplo della potenza di 2. Il motivo di tale scelta è da ricercare proprio nell'operazione di traduzione dell'MMU. Dato, infatti, un indirizzo virtuale da tradurre si usa scomporre il numero binario che lo rappresenta in due pezzi: i primi bit di tale indirizzo fanno riferimento alla pagina virtuale, i restanti bit si riferiscono invece all'offset dell'istruzione all'interno della pagina virtuale. Se l'indirizzamento virtuale è tale da comprendere 16 pagine virtuali, ognuna di 4 Kb, occorreranno allora 4 bit per indirizzarle. Ed ancora, se una pagina è grande 4096 bit l'offset di una istruzione (l'offset individua l'indirizzo iniziale dell'istruzione) può dunque variare tra 0 e 4096 bit. In definitiva occorreranno $\log_2 4096=12$ bit per l'indirizzamento dell'offset. Gli indirizzi adottati dal sistema considerato in figura si compongono allora di 16 bit: 4 bit indirizzano alla pagina virtuale e

12 bit indirizzano all'offset dell'istruzione. Questo permette all'MMU di effettuare velocemente la traduzione degli indirizzi. Infatti, fissato l'offset dell'istruzione all'interno della pagina (ed essendo questa delle stesse dimensioni di quella virtuale) è necessario aggiornare i soli primi bit, quelli cioè che fanno riferimento alla pagina virtuale. L'MMU, pertanto, scambierà questi bit con quelli della pagina fisica e manterrà inalterato l'offset. Ad esempio, l'indirizzo `8196` che si trova alla pagina virtuale `2` con uno spiazzamento di 4 indirizzi ha un indirizzo binario pari a `0010000000000100` (dove i primi 4 bit `0010` indicano la pagina virtuale `2` e `000000000100` sono i 4 indirizzi di offset) che deve essere mappato alla pagina fisica `6`. L'MMU, quindi, sostituisce i primi 4 bit (`0010`) con i 3 bit della pagina fisica `110` (la memoria fisica ha 8 pagine fisiche che necessitano pe la numerazione di 3 bit), l'indirizzo tradotto sarà 110000000000100.



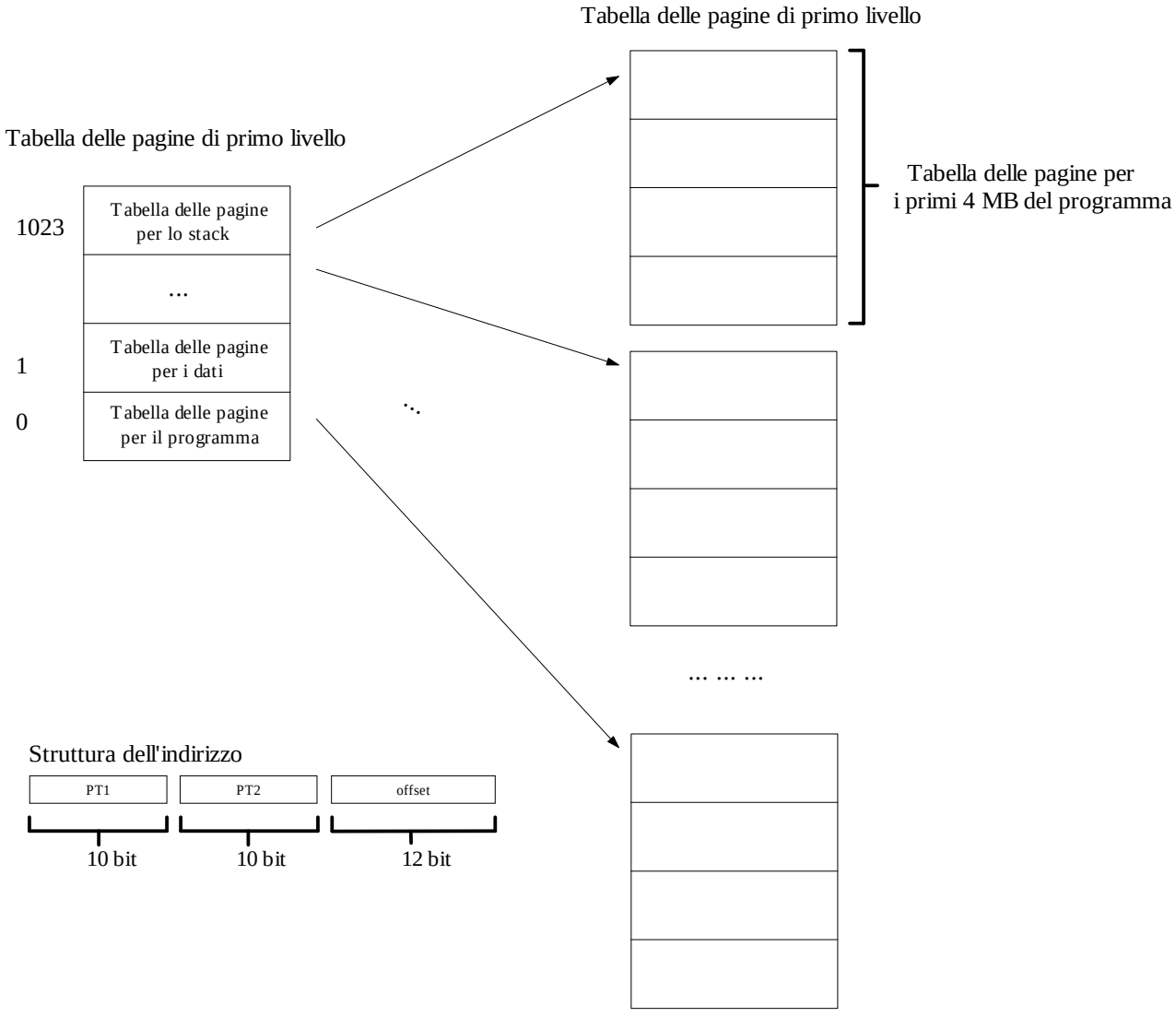
La tabella delle pagine

La traduzione offerta dall'MMU, così come vista nell'esempio precedente in figura, è basata nella realtà su una tabella delle pagine. L'MMU vi accede prelevando dall'indirizzo virtuale il numero di pagina virtuale e ne ricava, in uscita, il numero di pagina fisica (l'offset all'interno della pagina rimane invariato). Un bit, detto bit di validità, per ogni riga della tabella delle pagine indica all'MMU se la pagina virtuale è presente in memoria principale. In caso affermativo è possibile prelevare dalla tabella il numero di pagina fisica.

Range di indirizzi virtuali	Pagina virtuale	Validità	Frame
0K- 4K	0	1	2
4K – 8K	1	1	1
8K – 12K	2	1	6
12K – 16K	3	1	0
16K – 20K	4	1	4
20K – 24K	5	1	3
24K – 28K	6	0	
28K – 32K	7	0	
32K – 36K	8	0	
36K – 40K	9	1	5
40K – 44K	10	0	
44K – 48K	11	1	7
48K – 52K	12	0	
52K – 56K	13	0	
56K – 60K	14	0	
60K – 64K	15	0	

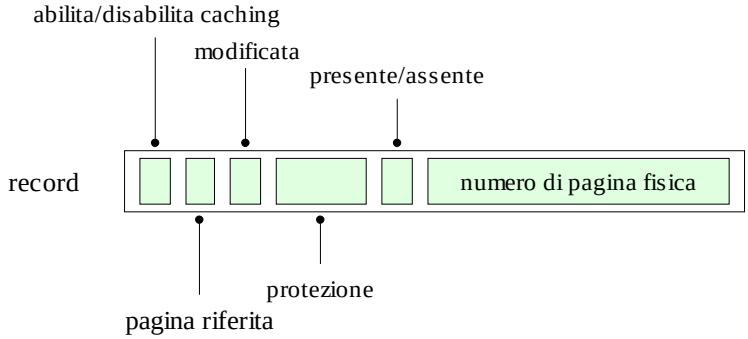
Le dimensioni della tabella delle pagine di un processo possono essere molto grandi, dipende dalle dimensioni del programma. Ed ancora, dinanzi ad una grande tabella delle pagine la traduzione degli indirizzi deve comunque mantenersi ragionevolmente veloce se non si vogliono degradare le prestazioni del sistema. Ad esempio, nell'ipotesi di un programma utente con spazio di indirizzamento di 32 bit (con 32 bit si hanno 2^{32} indirizzamenti, 4294967296 indirizzi) e pagine virtuali di 4 Kb la tabella delle pagine, dovendo avere un riferimento per ogni pagina virtuale, dovrà avere 1048576 righe (occuperà 32 Mb!). Ogni processo ha bisogno di una tabella delle pagine, nei sistemi di calcolo di ultima generazione dotati di maggiore memoria principale è possibile tenere in memoria una o più tabelle delle pagine di queste dimensioni, occorre tuttavia una soluzione migliore se non si vuole saturare l'intera memoria. Un processo, infatti, per funzionare, ha bisogno

di trovare in memoria l'intera tabella delle pagine. Qualora sia presente in memoria un solo processo per volta (alcuni sistemi funzionano così) bisogna anche considerare il problema del cambio di contesto. Occorre in tal caso salvare sul disco l'attuale tabella delle pagine e caricare quella del processo scelto dallo schedulatore per l'esecuzione. Una attenta osservazione ha permesso di risolvere il problema delle dimensioni della tabella delle pagine in memoria principale: un processo, solitamente, in esecuzione fa riferimenti in memoria ad una ristretta area della tabella delle pagine (ciò è vero solo quando il processo è in esecuzione da tempo). Questo permette ad un processo di essere in esecuzione anche senza l'intera tabella delle pagine. L'idea quindi è quella di dividere la tabella delle pagine in tanti pezzi e di caricare in memoria solo quelli usati dal processo.
Ad esempio, un indirizzo virtuale di 32 bit può essere suddiviso in un campo PT1 (process table di primo livello) da 10 bit, in un campo PT2 (process table di secondo livello) da 10 bit e in un offset di 12 bit. Con un offset di 12 bit si ha una pagina di 4096 bit, la tabella del processo dovrà pertanto avere $2^{10} \cdot 2^{10} = 2^{20} = 1048576$ righe, una per ogni pagina. Ed ancora, con 32 bit è possibile indirizzare 4 Gb di programma, nell'ipotesi ad esempio che il programma necessiti di 12 Mb: 4 Mb per il programma, 4 Mb per i dati e 4 Mb per lo stack (supponendo che fra l'area dati e lo stack vi è un buco di memoria non utilizzato). Quando alll'MMU si presenta un indirizzo virtuale esso estrae dapprima il campo PT1 e lo usa come indice per accedere alla tabella di più alto livello ricavando il numero di pagina fisica del secondo livello; poi impiega il campo PT2 ed accedendo alla tabella di secondo livello estrae la pagina fisica richiesta; quindi considera l'offset per orientarsi all'interno della pagina trovata.



Il modello delle pagine a più livelli (solitamente negli schemi a livelli usati non si va oltre il

secondo o terzo livello) permette di tenere in memoria delle pagine solo le pagine utili all'esecuzione del processo. Si tratta di un vantaggio significativo, meglio caricare (ad ogni cambio di contesto) una parte della tabella dei processi piuttosto che l'intera tabella.
Il record di una tabella delle pagine (in altre parole ogni sua riga) è organizzato diversamente da sistema a sistema, tuttavia è possibile riassumere tutte le informazioni in esso comuni dal momento che questi sono pressapoco gli stessi (a volte può cambiare anche solo l'ordine con cui sono memorizzate le informazioni, in altri casi invece possono esserci delle differenze di significato per i bit del record, in altri casi infine può effettivamente sussistere una differenza di bit aggiunti in un record). Per questo motivo anche le dimensioni del record possono essere diverse da sistema a sistema, ma solitamente vengono impiegati 32 bit. Il campo dati più importante del record è ovviamente quello che fa riferimento al numero di pagina fisica. Le righe della tabella delle pagine suddividono l'immagine in memoria del programma in base alla dimensione scelta per la pagina (ad esempio la riga 0 si occupa degli indirizzi di memoria che vanno da 0K a 4K se 4KB è la dimensione di una pagina, la riga 1 si occupa degli indirizzi di memoria che vanno da 4K a 8K e così via). L'MMU accede alla tabella delle pagine con il numero di pagina virtuale e ne vuole ricavare dapprima la sua presenza in memoria e successivamente, in caso affermativo, il numero di pagina fisica. Per questo motivo il record della tabella delle pagine prevede un bit di validità che se trovato ad 1 autorizza l'MMU a leggere il numero di pagina fisica, in caso contrario si verifica un page fault ed il sistema operativo si attiva affinchè la pagina venga resa disponibile in memoria per far ripartire l'istruzione che la richiedeva. Ogni pagni è poi accompagnata da un set di bit di protezione, questi dicono all'MMU i permessi rilasciati per ogni pagina presa in memoria. Tali permessi nella forma base si riducono ad unico bit che contraddistingue con uno dei due simboli la possibilità di accedere alla pagina in lettura, nell'altro caso invece dicono all'MMU se alla pagina si può accedere in scrittura. In sistemi più di alto livelo i bit di protezione possono addirittura essere 3, ciascuno dei quali abilita o disabilita la lettura, la scrittura e l'esecuzione della pagina.
Le pagine virtuali del processo che sono mappate nella memoria fisica possono, durante l'esecuzione del processo, subire delle modifiche (se il bit di protezione lo consente) ed allora per indicare al gestore della memoria che una pagina è stata modificata si usa aggiungere nel record della tabella delle pagine un ulteriore bit detto bit di modifica. Questo bit, come vedremo a breve, è estramente utile al gestore della memoria per capire quali sono le pagine che il processo sta usando. In caso di page fault il gestore della memoria sacrifica una pagina non modificata per fare posto ad una nuova pagina (la pagina non modificata viene sovrascritta dalla nuova pagina poichè la copia in memoria della pagina sovrascritta è ancora valida non essendo stata modificata, una pagina modificata richiede al gestore della memoria di salvare dapprima la pagina sul disco e successivamente si può caricare in essa la nuova pagina). Esiste poi un altro bit utile al gestore della memoria per applicare un algoritmo disostituzione della pagina, si tratta del bit che indica se una pagina è stata riferita (sia in lettura che in scrittura). Talvolta nel record è anche presente un bit per abilitare/disabilitare il caching della pagina. In alcune circostanze la pagina di memoria può essere mappata sulla memoria o sui registri di una periferica anzichè sulla memoria principale. Ed allora, per indicare al gestore della memoria che la pagina mappata in memoria non è quella più aggiornata (ad esempio perchè la periferica modifica i dati prima nei suoi registri e poi li ripone a proprio piacimento in memoria) si usa il suddetto bit di caching.



Il TLB

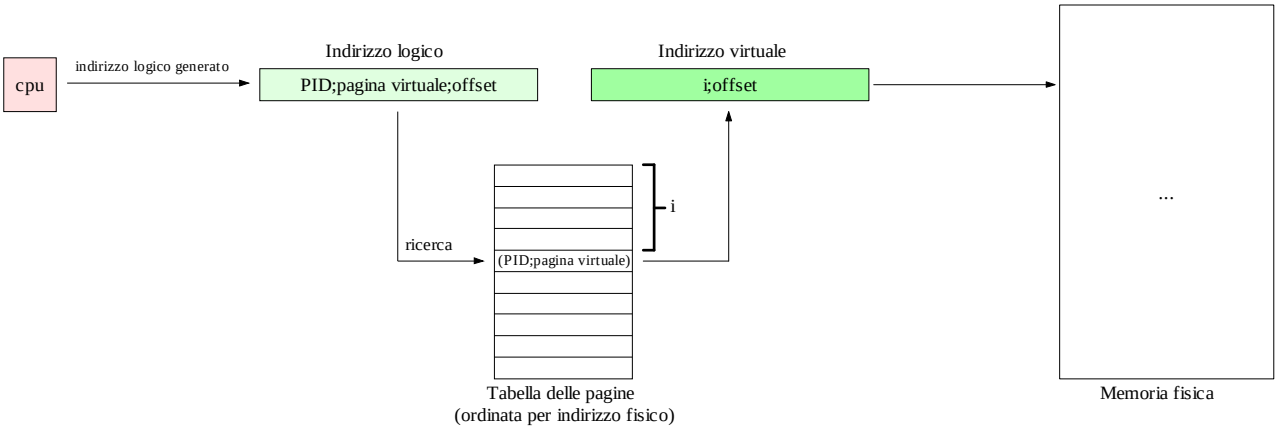
La soluzione ai problemi di spazio e dimensione della tabella delle pagine ed ai tempi di accesso e ricerca è il TLB (translation lookside buffer) detta a volte memoria associativa. La soluzione è stata trovata grazie al fatto che la maggior parte dei programmi tende ad eseguire dei riferimenti ad un piccolo insieme di pagine, così solo una frazione degli elementi della tabella delle pagine viene utilizzata di frequente. Il TLB è dunque un dispositivo hardware che serve a mappare gli indirizzi logici sugli indirizzi fisici senza passare per la tabella delle pagine. Il TLB di solito si trova nell'MMU e ciò favorisce dei buoni tempi di accesso, esso contiene un ristretto numero di elementi della tabella delle pagine. Nell'esempio viene mostrato un TLB per accedelerare la paginazione con la capacità di memorizzazione di 8 record:

Pagina valida	Pagina virtuale	Modificata	Protetta	Pagina fisica
1	140	1	RW-	31
1	20	0	R-X	38
1	130	1	RW-	29
1	129	1	RW-	62
1	19	0	R-X	50
1	21	0	R-X	45
1	860	1	RW-	14
1	861	1	RW-	75

Ogni riga del TLB equivale ad una riga della tabella delle pagine a differenza del campo Pagina virtuale che indica quali pagine logiche ho in memoria e dunque sul chip del TLB. Il tipo di memoria usata per fare un TLB non è di tipo associativo come la memoria principale, solitamente si usa un tipo di memoria con accesso ad indice o chiave. Infatti, alla tabella del TLB si accede con l'indirizzo di pagina virtuale (a cui corrisponde una pagina virtuale che è in questo caso la chiave di accesso), le operazioni di ricerca potrebbero provocare la scansione dell'intera struttura dati ed essere collegate alla collocazione del record nella struttura (se il record che sto cercando è l'ultimo dovrò scorrere l'intera lista) nel caso si utilizzi una memoria associativa. Ed allora la pagina virtuale da cercare viene parallelamente ricavata non appena questa è disponibile impiegando una memoria con accesso su chiave. La la ricerca viene fatta in hardware per cui è estremamente veloce e richiede davvero poco tempo.
Quando all'MMU arriva un indirizzo logico da tradurre l'hardware controlla per prima se il relativo numero di pagina logico è presente nel TLB effettuando un confronto parallelamente al suo arrivo con tutti gli altri elementi del TLB (qui la ricerca è veloce). Se si trova un elemento con lo stesso numero di pagina logica e non si violano i bit di protezione allora il numero della pagina fisica viene direttamente preso dal TLB, senza accedere dunque alla tabella delle pagine. In tal caso si dice che si verifica un evento detto page hit (pagina presa). Se il numero di pagina virtuale risulta presente nel TLB ma si sta accedendo ad una pagina protetta viene generato un errore di protezione. Ed ancora, se la pagina logica non è presente nel TLB l'MMU si accorge di questa mancanza (evento page miss) ed il S.O. è costretto ad effettuare una ricerca nella tabella delle pagine, quindi scarica un elemento del TLB rimpiazzandolo con quello richiesto.
Quindi, in genere, si parla di page fault quando la pagina virtuale è assente nella tabella delle pagine fisiche in memoria principale; si parla di page miss se la pagina virtuale è assente nel TLB.

La tabella delle pagine invertite
Le tabelle delle pagine tradizionali del tipo descritto precedentemente richiedono unelemento per ogni pagina virtuale, in quanto sono indicizzate da un numero di pagina virtuale. Se lo spazio di indirizzamento è di 2^{32} byte, 4096 byte per pagina, allora sono necessari più di un milione di elementi nella tabella delle pagine ($2^{32}/2^{12}$=1048576) che con 84 byte per record occuperebbe 4 MB. 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^{64} byte, con pagine di 4096 byte, abbiamo bisogno di una tabella delle pagine di oltre 30 milioni di gigabyte ($2^{64}/2^{12}=2^{52}$ con 8 byte per ogni record). E' chiaro, quindi, che occorre una soluzione differente per gli spazi di indirizzamento virtuali paginati di 64 bit. Questa soluzione è la tabella delle pagine invertite.
Esisterà in tal caso una sola tabella delle pagine per tutti i processi, questa tabella si compone di un

elemento per ogni pagina reale. In altre parole, per ogni blocco (ricavato dalla suddivisone della memoria fisica in base alla dimensione della pagina) della memoria fisica esiste un riferimento nella tabella delle pagine. Ogni elemento della tabella della pagina contiene, quindi, l'indirizzo virtuale della pagina memorizzata in quella locazione fisica e informazioni sul processo che possiede tale pagina. Per velocizzare la ricerca nella tabella delle pagine si usa una tabella di hash: gli elementi sono ordinati per indirizzo fisico, si accede alla tabella mediante l'associazione generata dall'operazione di hash che in ingresso richiede un identificativo del processo e il numero di pagina virtuale producendo in uscita l'indirizzo fisico. Per aumentare le prestazioni dei sistemi basati su questo schema si può fare ancora fare ricorso all'utilizzo del TLB e memorizzare in esso un certo numero di record.



Algoritmi di sostituzione delle pagine

Al verificarsi di un fault di pagina il S.O. deve scegliere una pagina dalla memoria e rimuoverla per fare posto alla nuova pagina. Si potrebbe scegliere a caso la pagina da rimuovere, tuttavia le prestazioni del sistema risultano migliori se la scelta della rimozione avviene basandosi su alcuni criteri decisionali. Infatti, se si rimuove una pagina che è maggiormente utilizzata si provocherà in seguito un ulteriore rallentamento poiché sarà necessario ricaricare in memoria la pagine che precedentemente era stata rimossa. Inoltre, se la pagine che si vuole rimuovere è stata modificata è allora necessario, prima della rimozione, effettuare un scrittura sul disco che aggiorna dunque la vecchia copia, anche in questo caso si aggiungerà un tempo dovuto alla fase di scrittura, se invece la pagina non è stata modificata si può semplicemente sovrascrivere quest'ultima con la certezza che il contenuto in memoria logica è del tutto uguale. Il problema del rimpiazzamento investe anche altri contesti, come ad esempio la progettazione di calcolatori che usano memoria cache. La memoria cache di 32 o 64 byte tiene memoria dei byte di recente utilizzati, quando la cache è piena si procede alla rimozione di qualche blocco. Altro esempio, leggermente diverso ma simile, è il caso di un server web; il server può tenere traccia delle pagine offerte immettendole in una memoria cache ed offrendole qualora siano richieste. Se la memoria cache risulta essere piena si procede, anche in questo caso, alla rimozione scegliendo dunque la pagina da scaricare (in tal caso tuttavia le pagine saranno sempre le stesse, esse cioè non sono modificate per cui non è necessario riscriverle sul disco).

Algoritmo di rimpiazzamento delle pagine ottimale

Il primo criterio di sostituzione che si può pensare di utilizzare è valido solo a livello teorico poiché si basa sull'ipotesi di conoscere il numero di istruzioni che precedono l'utilizzo di una pagina. Come nella vita reale, questo algoritmo cerca di posticipare gli eventi sgradevoli spostandoli nel futuro. Se una pagina ad esempio non sarà usata per 8 milioni di istruzioni e un'altra pagina non sarà usata per 6 milioni di istruzioni, rimuovere la prima pagina posticipa il page fault. Il problema di questo algoritmo è l'assunzione di tempo che si commette nei confronti di una pagina prima che essa sia utilizzata, cosa che il sistema non può ovviamente prevedere. Questo criterio tuttavia diventa utile se confrontato con altri criteri di rimpiazzamento, esso infatti è usato per misurare la validità di un algoritmo in quanto essendo ideale raccoglie dei buoni risultati.

Algoritmo di rimpiazzamento delle pagine non usate di recente
La maggior parte dei calcolatori associa a ciascuna pagina di memoria due bit, il bit di modifica (bit M) è utile in questo contesto perché può essere testato affinché si possa stabilire se una pagina di memoria è stata modificata. Tra due pagine di memoria è sempre meglio sostituire una pagina non modificata poiché non è necessario effettuare un aggiornamento della copia sul disco essendo questa immutata. Questo metodo è semplice da implementare ma può essere ampliato se si considera anche il bit di pagina riferita (bit R). Questo bit assume significato solo se pari ad 1 ed indica che la pagina è stata riferita in lettura oppure in scrittura. Usando dunque due bit (il bit M ed il bit R) posso definire diverse classi (quattro per la precisione) di utilizzo delle pagine ed agevolare quindi la scelta della rimozione al gestore della memoria.

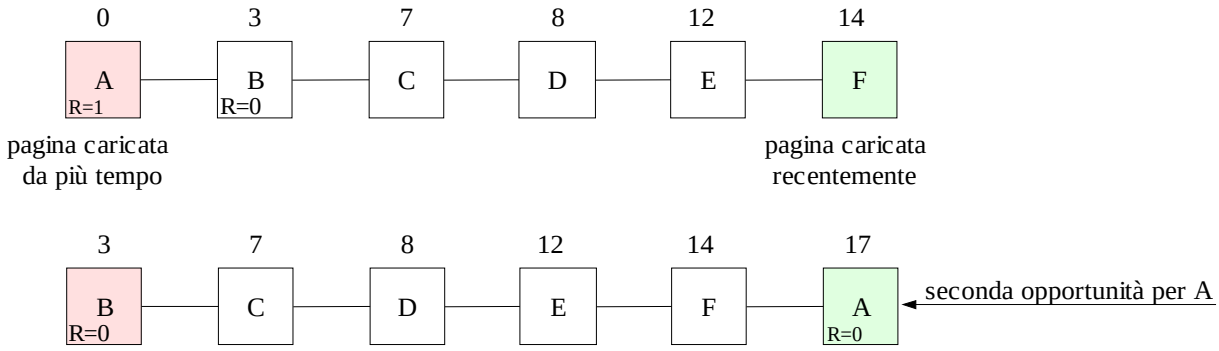
Classe	Info
Classe 0	Non usata, non modificata (R=0;M=0)
Classe 1	Non usata, modificata (R=0;M=1)
Classe 2	Usata, non modificata (R=1;M=0)
Classe 3	Usata, modificata (R=1;M=1)

I bit R ed M sono gestiti da un hardware dedicato, la classificazione delle pagine sopra elencata è assai utile quando si verifica un page-fault, essa consente di discriminare meglio le pagine effettuando una scelta mirata delle pagine da scaricare, vengono per prima scaricate le pagine di classe 0 ed a seguire quelle di classe successiva. Tuttavia è bene precisare che sia i bit R che il bit M vengono azzerati dopo un certo istante di tempo, questo azzeramento è necessario poiché dopo un certo periodo di esecuzione si rischierebbe di avere ad 1 tutti i bit R ed M riferiti alle pagine in memoria (le pagine con il passare del tempo vengono usate per cui i bit potrebbero rimanere bloccati ad 1). Attenzione, l'algoritmo non discrimina le pagine di una stessa classe, in seguito si vedranno algoritmi più discriminanti di quello appena visto.

Algoritmo di rimpiazzamento delle pagine FIFO
Una gestione basata sulla metodologia FIFO si applica a diversi contesti, anche alla gestione delle pagine di memoria. Se il sistema operativo è capace di tenere una lista di tutte le pagine in memoria ordinata dalla pagina più vecchia a quella che per ultima è arrivata è possibile implementare una politica FIFO nella gestione della memoria se, al verificarsi di un fault di pagina, si decide di rimuovere dalla lista delle pagine quella che al momento occupa la testa della lista. Al momento del faul di pagina la pagina in testa alla lista viene rimossa e la nuova pagina viene aggiunta in coda. Tuttavia l'algoritmo di rimpiazzamento con politica FIFO è raramente usato nella sua forma base appena descritta. Esso infatti potrebbe cancellare dalla memoria principale una pagina che è lì da più tempo e fare in questo modo spazio per la nuova pagina da allocare ma la stessa pagina cancellata potrebbe essere quella più usata dal processo in esecuzione che quindi la richiederà immediatamente passata qualche istruzione.

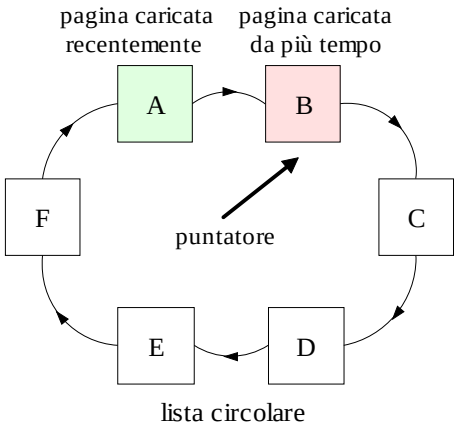
Algoritmo di rimpiazzamento delle pagine della seconda opportunità
Per ovviare al problema descritto sopra della rimozione di una pagina con schema FIFO è possibile implementarne una variante. Qualora si verifica un fault di pagina, la pagina che è in memoria principale da più tempo è anche quella candidata alla rimozione, essa tuttavia non viene sovrascritta se testando il bit R si scopre che la pagina è stata recentemente riferita ed è quindi usata. Pertanto, se il bit R di una pagina vecchia viene trovato ad 1 si decide di concedere alla pagina una seconda opportunità: la pagine non viene sovrascritta ma è collocata in fondo alla lista delle pagine (come se fosse una pagina nuova) e la ricerca continua a partire dalla prossima pagina nella lista.
Per tenere traccia dell'istante di tempo in cui la pagina è stata caricata in memoria principale viene scritto all'interno di un campo appartenente alla pagina il valore che un contatore detiene in virtù dei tick che la cpu compie. Ad ogni tick il contatore incrementa il suo conteggio, quindi, una pagina che viene caricata in memoria entra a far parte della lista e si vede assegnato il valore di conteggio attualmente posseduto dal contatore come valore rappresentativo dell'istante di tempo in cui la pagina stessa è stata caricata. Le successive pagine che vengono caricate in memorie avranno quindi

un differente valore del contatore (una differente copia) e se la pagina viene riferita essa viene trattata come se fosse nuovamente caricata e verrà collocata in fondo alla lista delle pagine con un nuovo valore di conteggio. Quando si verifica un page-fault ed una pagina deve essere rimossa sarà rimossa la pagina che ha la copia più bassa del valore di conteggio ed il bit R=0 (ricordiamo infatti che la copia più bassa del valore di conteggio è assegnata alla pagina che è da più tempo in memoria e se R=1 viene data alla pagina una seconda opportunità).



Algoritmo di rimpiazzamento delle pagine dell'orologio

Dal momento che l'algortimo di rimpiazzamento delle pagine orientato alla seconda opportunità non fa ltro che spostare in continuazione le pagine in fondo alla lista per rendere più veloce le operazioni di ricollocazione all'interno della lista si può anche adottare una lista circolare delle pagine. Un puntatore viene costantemente aggiornato e fatto puntare, per questo motivo, alla pagina che è da più tempo in memoria.



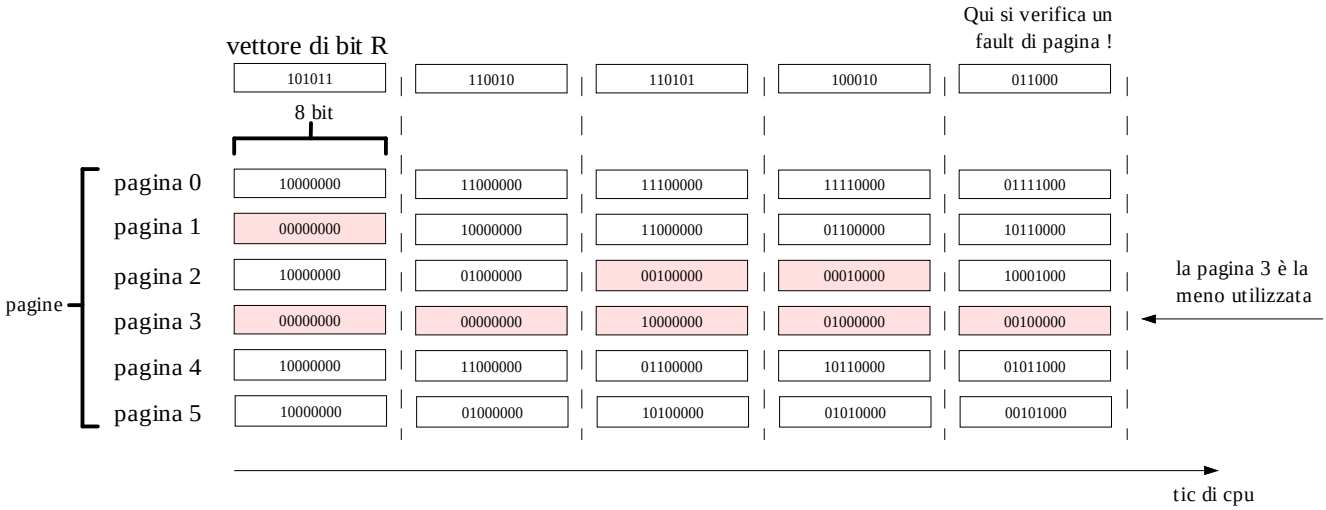
Algoritmo di rimpiazzamento delle pagine LRU

L'algoritmo least recently used (LRU) si basa sul principio di località temporale: le pagine usate di recente saranno riferite di nuovo nell'arco di un breve intervallo di tempo mentre le pagine che non sono state usate per un lunghissimo intervallo di tempo non lo saranno, molto probabilmente, per un altro intervallo di tempo.

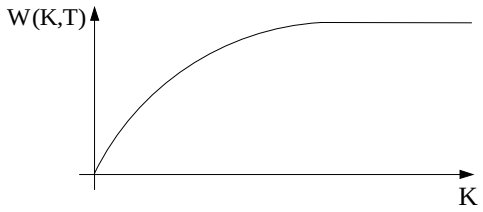
Questa idea può essere realizzata se si pensa di scaricare ad ogni page-fault la pagina che ha il minor tempo di utilizzo. Tuttavia, è bene precisare che se la realizzazione da un punto di vista teorico è possibile tale soluzione rimane comunque non economica. L'algoritmo, dunque, procede in questo modo: ogni pagina di memoria ospita nel record un campo dati in cui viene scritto il tempo di utilizzo della pagina; al verificarsi di un page fault le pagine dotate del più piccolo valore di conteggio saranno candidate alla rimozione.

Per implementare completamente LRU è necessario mantenere in memoria una lista concatenata di tutte le pagine, la pagina recentemente usata occupa la testa della lista, la difficoltà che si riscontra risiede nei continui aggiornamenti da fare alla lista qualora si verifichi un riferimento ad una pagina. Questo algoritmo può essere implementato o comunque approssimato usando dell'hardware

dedicato: l'hardware incrementa il valore del contatore C ad ogni tick di cpu; l'accesso ad una pagina P provoca l'aggiornamento del valore C nel rispettivo record (viene aggiunto alla copia di C il nuovo valore del contatore); ad ogni page fault scarico la pagina con il più piccolo valore di C.
Esiste inoltre una implementazione software dell'algoritmo LRU.
Ogni pagina è dotata di una stringa di un certo numero di bit che colleziona nel tempo i riferimenti alla pagina come dei gettoni di presenza. Ad ogni tick di cpu le pagine riferite ricevono in ingresso un gettone (un bit 1) per ogni riferimento (in assenza di riferimento ricevono un bit 0), i successivi gettoni (quindi altri riferimenti che si susseguono nel tempo) provocano uno shift a destra dei gettoni finora ricevuti. In questo modo, quando si verifica un fault di pagina, è sufficiente contare il numero di gettoni (e quindi il numero di bit 1) per stabilire quale pagina deve essere rimossa dalla memoria principale.



Il modello working set
Affrontiamo adesso altri algoritmi di sostituzione delle pagine di memoria che si basano sul concetto di insieme di lavoro detto appunto working set. Nella paginazione pura dei processi si assiste a numerosi page fault quando un processo viene avviato per la prima volta in memoria. Infatti, un processo appena mandato in esecuzione chiederà irrimediabilmente le pagine relative alle prossime istruzioni e siccome queste non sono ancora state caricate in memoria impegnerà il S.O. affinché le stesse siano rese disponibili al processo che ne sta facendo richiesta. Questa strategia viene detta paginazione a richiesta dal momento che le pagine vengono caricate a richieste piuttosto che anticipatamente. Quindi, esclusa la fase iniziale di un processo che genera diversi page fult si assiste in seguito ad una richiesta quasi ripetitiva delle pagine a lui utili (la maggior parte dei processi in esecuzione fa riferimenti ad un insieme ristretto di pagine detto working set appunto). La prossima idea di gestione della memoria si basa sul concetto di working set: se tutto l'insieme di lavoro si trova già in memoria, magari perchè precaricato anticipatamente, il processo girerebbe senza causare molti fault di pagina. Se la memoria disponibile è troppo piccola per poter contenere tutto l'insieme di lavoro, il processo causerà molti fault di pagina e girerà per questo motivo molto lentamente dal momento che per eseguire un istruzione ci vogliono pochi nanosecondi e per caricare una pagina dal disco ci vogliono 10 millisecondi. Un programma che genera un fault di pagina dopo l'esecuzione di poche istruzioni viene detto in una situazione di trashing. Il working set dipende da T e da K, dove K rappresenta il numero di accessi in memoria e T rappresenta il tempo. Dunque con W(K,T) si rappresenta l'insieme delle pagine che sono state usate/accedute in memoria nell'ultimo istante di tempo T. Notare che il working set ha un andamento diverso al variare di K, tale andamento si assesta in un intorno del working set all'aumentare di K (il processo in tal caso ha raggiunto un certo regime e riferirà sempre alle stesse pagine):



Il concetto di working set può essere adoperato per limitare i numeri di fault di pagina, tutto quello che occorre fare è far trovare in memoria il working set di ogni processo. Per questo motivo alla paginazione a richiesta si preferisce sovente la tecnica della pre-paginazione. Ad ogni istante di tempo T esiste un insieme di lavoro costituito da tutte le pagine utilizzate dai K riferimenti. Dato che l'insieme di lavoro varia lentamente nel tempo si può supporre quali pagine saranno necessarie quando il programma verrà fatto ripartire.
Affinchè sia possibile implementare il concetto di working set in un processo è necessario che il sistema operativo tenga traccia di quali pagine vi appartengono; disporre di questa informazione permette di implementare da subito un interessante algoritmo per il rimpiazzamento delle pagine: al verificarsi di un page fault si seleziona la pagina che non appartiene all'insieme di lavoro e la si scarica. Mi occorre tuttavia un criterio per stabilire quale e in che modo sostituire le pagine. Ridefinisco a tale proposito la definizione di working set ed al posto di K uso τ che mi indica il numero di pagine lette/accedute nell'ultimo istante di tempo (tick del clock). Per ogni pagina esiste un contatore che fissa il tempo dell'ultimo accesso segnandolo nel record di ciascuna pagina (un altro contatore segna invece l'età virtuale della pagina). Al verificarsi di un page fault si leggono le pagine, tutte quelle con R=1 riceveranno come aggiornamento del tempo di ultimo acceso il tempo correntemente segnato dal contatore. Se R=0 la pagina non è stata, invece, riferita nell'ultimo tick di clock e può essere candidata alla rimozione. Tuttavia per stabilire la sua rimozione si calcola dapprima la sua età pari alla differenza fra il suo tempo virtuale di presenza in memoria (attenzione non il tempo di ultimo accesso!) ed il tempo segnato dal contatore: se l'età della pagina è più alta di τ la pagina non è più nel working set e può essere eliminata, se l'età è invece minore o uguale a τ la pagina corrente è ancora nell'insieme di lavoro e sarà, almeno per ora, risparmiata.

Algoritmo di rimpiazzamento delle pagine con orologio WSclock
Si tratta dell'algoritmo dell'orologio già visto in precedenza e qui adattato al concetto di working set. Siccome l'algoritmo prevede, ad ogni fault di pagina, nella sua forma base, l'analisi di tutta lista si decide di implementare l'algoritmo in una lista circolare di pagine fisiche. Questa è inizialmente vuota ma si riempe ben presto dopo aver caricato la prima pagina e quelle successive, assieme ai bit R ed M si inserisce poi il tempo di ultimo accesso (una copia del tempo virtuale corrente). Un puntatore viene quindi fatto puntare ad un elemento della lista. Al verificarsi di un fault di pagina, se il bit R è 1 la pagina viene considerata nel working set (tant'è che è stata recentemente riferita) ed il tempo di ultimo accesso viene aggiornato con una nuova copia del tempo virtuale corrente. Se il bit R è 0 la pagina puntata non è tra quelle recentemente usate dal processo, tuttavia potrebbe comunque appartenere al working set e per questo motivo si procede dunque a stabilire la sua età. Se l'età della pagina è maggiore di τ allora la pagina è fuori il working set e viene candidata alla rimozione. Essa sarà rimossa dalla memoria principale solo se anche il bit M è 0 (la pagina non è stata cioè modificata). Se infatti si trovasse M=1 la pagina necessiterebbe di una scrittura sul disco (per rendere aggiornata anche la copia sul disco). Notare che la suddetta pagina è temporaneamente risparmiata dalla rimozione, essa è tuttavia segnata come pagina da schedulare appena possibile e se il puntatore dell'orologio ritorna su di essa questa volta può essere sovrascritta senza alcun timore.
Si possono poi verificare alcune singolarità: se la lancetta torna al punto di partenza non trovando una buona pagina da rimpiazzare (in altre parole se durante il primo giro di sondaggi non si verificano le condizioni R=0, M=1 ed età>τ) l'algoritmo avrà sicuramente schedulato delle pagine da copiare sul disco per cui troverò al successivo giro di lancetta almeno una pagina con bit M=0; Se nessuna delle pagine nella lista è stata schedulata per la copia sul disco tutte le pagine avranno un età tale da appartenere al working set per cui dopo il primo giro non troverò una pagina da scartare: in tal caso la cosa più semplice da fare è rilasciare una qualiasi delle pagine non modificate

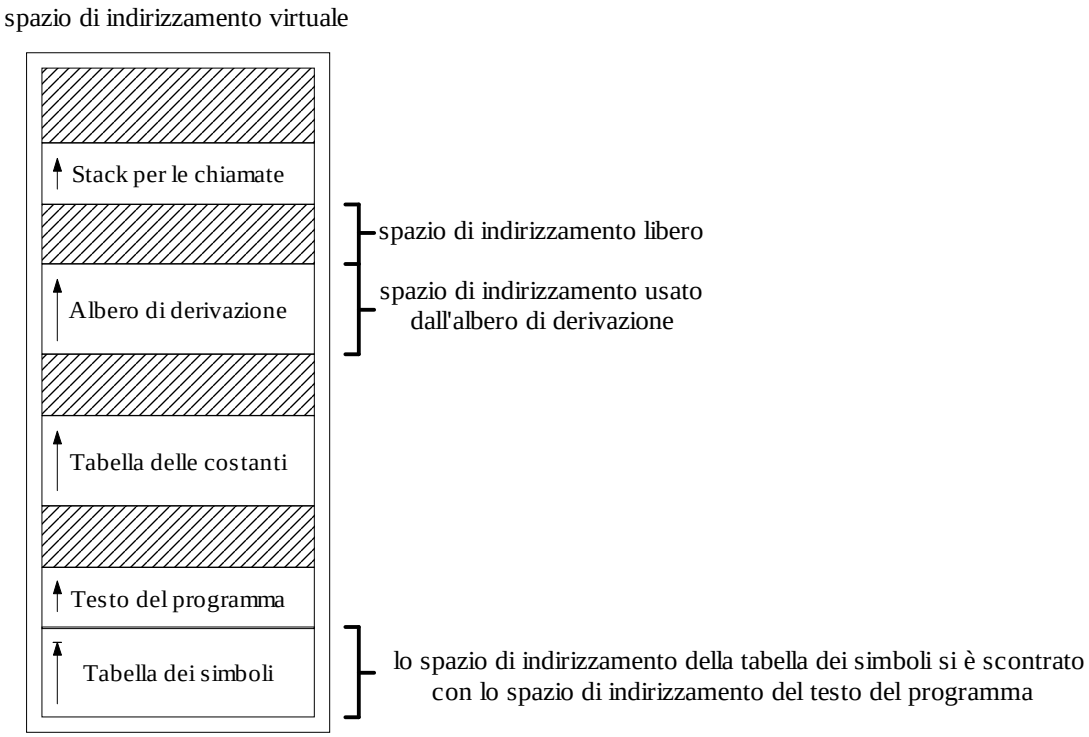
(magari memorizzando durante il giro della lancetta una di questa cosicchè si dispone già della sua locazione).

Riepilogo degli algoritmi di sostituzione

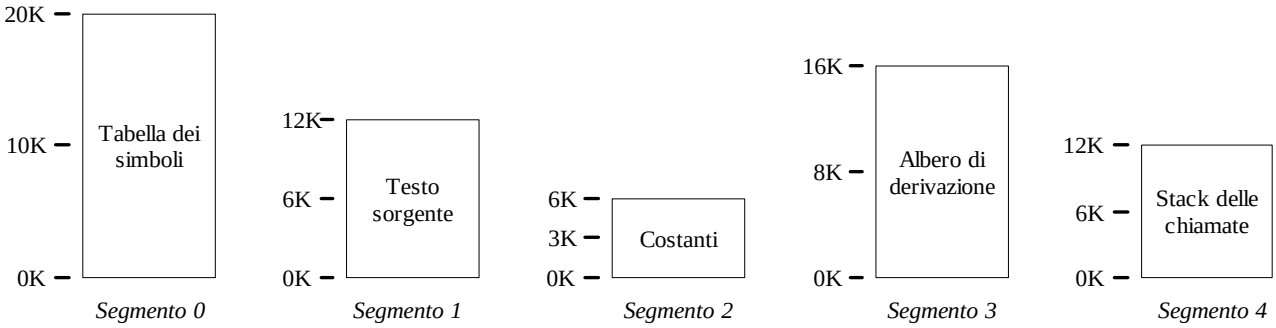
Tipo di algoritmo	Commento
Algoritmo ottimale	Non realizzabile a causa dell'assunzione temporale con dovrebbero avvenire i riferimenti alle pagine ma comunque utilizzato in fase di benchmark;
Algoritmo NRU (not recently used)	Si basa sulla distinzione delle classi generate dalla combinazione dei bit R ed M. Non discrimina le pagine di una stessa classe!
Algoritmo FIFO	Raramente usato nella sua forma base poiche può cancellare anche una pagina datata che però è in uso!
Algoritmo della seconda possibilità	E' una variante dell'algoritmo FIFO, considera i bit R per dare ad una pagina datata una seconda possibilità ricollocandola in fondo alla lista;
Algoritmo dell'orologio	Limita le operazioni di rimozione ed inserimento in lista basandosi su una lista collegata ad anello per cui è meno costoso dell'algoritmo con seconda possibilità;
Algoritmo LRU (last recently used)	Eccellente algoritmo ma va implementato in hardware dal momento che richiede un timer per conteggiare la vita di una pagina;
Algoritmo basato sull'età	Approssima bene all'LRU realizzandolo in software mediante stringhe di bit a gettoni;
Algoritmo basato sul working set	Basato su un interesante intuizione ma costoso da realizzare, è un algoritmo di pre-paging;
Algoritmo basato sul working set clock	Sfrutta bene l'idea del working set abbinandola ad una struttura dati circolare. E' buono ed efficiente;

La segmentazione

Si è soliti dire che la memoria paginata è unidimensionale tant'è che essa definisce sulla memoria fisica un unico spazio di indirizzamento che va da 0 ad un qualche massimo (il valore massimo dipende da quanto è grande la memoria fisica). La segmentazione, invece, è una tecnica ortogonale alla paginazione per la gestione dello spazio di indirizzamento dei processi. Essa infatti definisce più di uno spazio di indirizzamento, ognuno dei quali si dice essere un segmento.
In uno spazio di indirizzamento unidirezionale, così come lo è una memoria paginata, un processo di compilazione genera in fase di esecuzione svariate tabelle e strutture dati (tabella dei simboli, tabella delle costanti, testo del programma, albero di derivazione del programma e stack per le chiamate) che vengono proiettate in memoria principale l'una a ridosso dell'altra. Nella realtà si usa distanziare queste strutture dati con delle aree vuote di memoria cosicchè esse hanno la possibilità di crescere durante la compilazione. Tuttavia, pur inserendo le suddette aree vuote di memoria, può capitare che una struttura dati, crescendo, urti quella che la precede (la tampona). Ciò ad esempio accade quando nel programma ci sono troppe variabili e queste, man mano che la compilazione va avanti, vanno ad aggiungersi alla tabella dei simboli. Prima o poi la tabella dei simboli si scontrerà con la struttura dati che la precede:



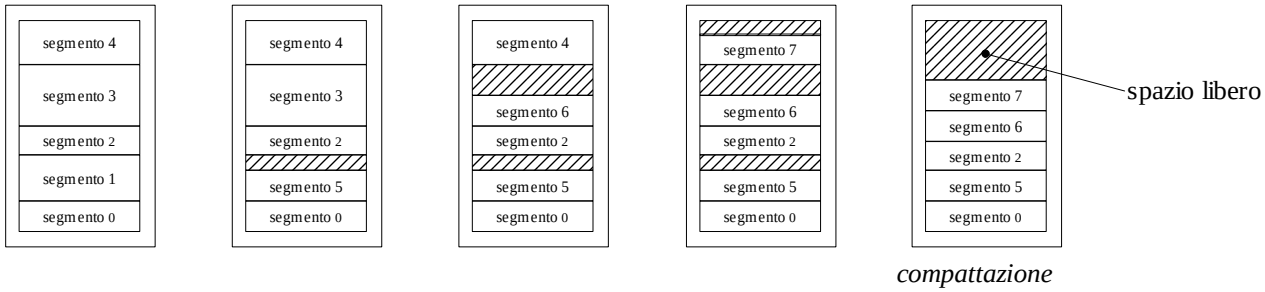
Quando si verifica lo scontro fra lo spazio di indirizzamento della tabella dei simboli e lo spazio di indirizzamento del testo del programma il compilatore potrebbe avvisare l'utente sulla sua impossibilità a procedere oltre, a causa delle troppe variabile nel programma (una gran bella figura). Una soluzione, invece, potrebbe essere quella di assegnare gli indirizzamenti liberi alle tabelle che ne hanno bisogno. Quest'ultima soluzione rimette nelle mani del programmatore la gestione dello spazio di indirizzamento così come i programmatori lo avevano assegnato al gestore della memoria quando la gestione degli overlay portava loro tempo prezioso. La soluzione che invece è adoperata è appunto la segmentazione della memoria. Ogni segmento consiste di una sequenza lineare di indirizzi che vanno da 0 ad un qualche massimo. Ovviamente in memoria esisteranno più segmenti ed allora ognuno di essi avrà un numero che lo identifica. Le lunghezze dei segmenti possono essere cambiate anche a run-time per cui un segmento può crescere quando in esso si allocano variabili oppure diminuire quando le stesse variabili vengono deallocate. Qui la crescita di una struttura dati all'interno del segmento può avvenire con la certezza di non causare scontri con altre strutture dati.



Un segmento può contenere una procedura, una struttura dati, una matrice ma di solito non contiene unmisto di tipi diversi. A differenza della paginazione, la segmentazione non è trasparente al programmatore che per accedere alla memoria principale deve formulare un indirizzo (nelle istruzioni dellinguaggio macchina) suddiviso nella coppia: numero di segmento; offset all'interno del segmento. Ci sono diversi vantaggi che una memoria segmentata porta con se. Oltre a facilitare la gestione delle strutture dati che crescono e/o decrescono, la memoria segmentata comporta che:

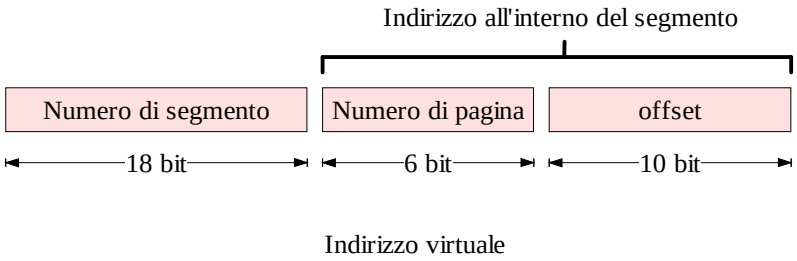
- se la procedura del segmento K viene modificata e quindi ricompilata è necessario ricompilare solo il segmento K piuttosto che l'intero spazio di indirizzamento del programma come invece accade nella memoria paginata. Con una memoria unidirezionale tutte la procedure sono impacchettate una a ridosso dell'altra, per cui se una di queste cambia fa slittare in avanti gli indirizzi di tutte le altre procedure (la segmentazione velocizza la gestione della memoria);
- più segmenti in memoria principale possono condividere una libreria molto più agevolmente piuttosto che fornire la suddetta libreria ad ogni processe (la segmentazione agevola la condivisione);
- ogni segmento è dotato di un livello di protezione. Ancora una volta la condivisione di una cosa comporta delle regole di accesso che disciplinano l'uso. Qui il problema è ancora più sentito poichè il programmatore adesso potrebbe conoscere il contenuto del segmento e uindi utilizzarlo senza alcun scrupolo. Per ogni segmento è possibile definire quali operazioni sono ammesse sulle informazioni in esso contenute: read, write, read & write, read & execute sono alcuni esempo dei permessi concessi ad un segmento. Sono questi livelli di protezione che permettono ad un utente di scoprire gli errori di programmazione e gli abusi, eventuali, di altri utenti maliziosi;

Osservare che nella memoria paginate il problema della protezioe non è assai sentito almeno quanto lo è per la segmentazione. Nella paginazione il contenuto di una pagina è casuale. La gestione della memoria in un siste,a che permetta la segmentazione dovrà occuparsi di allocare i segmenti di ciascun processo in memoria. Esisterà pertanto una tabella dei segmenti per ciascun processo che conterrà le informazioni sull'allocazione di ciascun segmento in memoria (oltre alle informazioni dulla protezione di ciascun segmento). Nell'implementazione della segmentazione pura, dopo che il sistema avrà lavorato per un pò si verifica il fenomeno della frammentazione esterna (dovuto alla allocazione ed alla deallocazione dei segmenti) che può (di tanto in tanto) essere limitato con la compattazione della meoria:



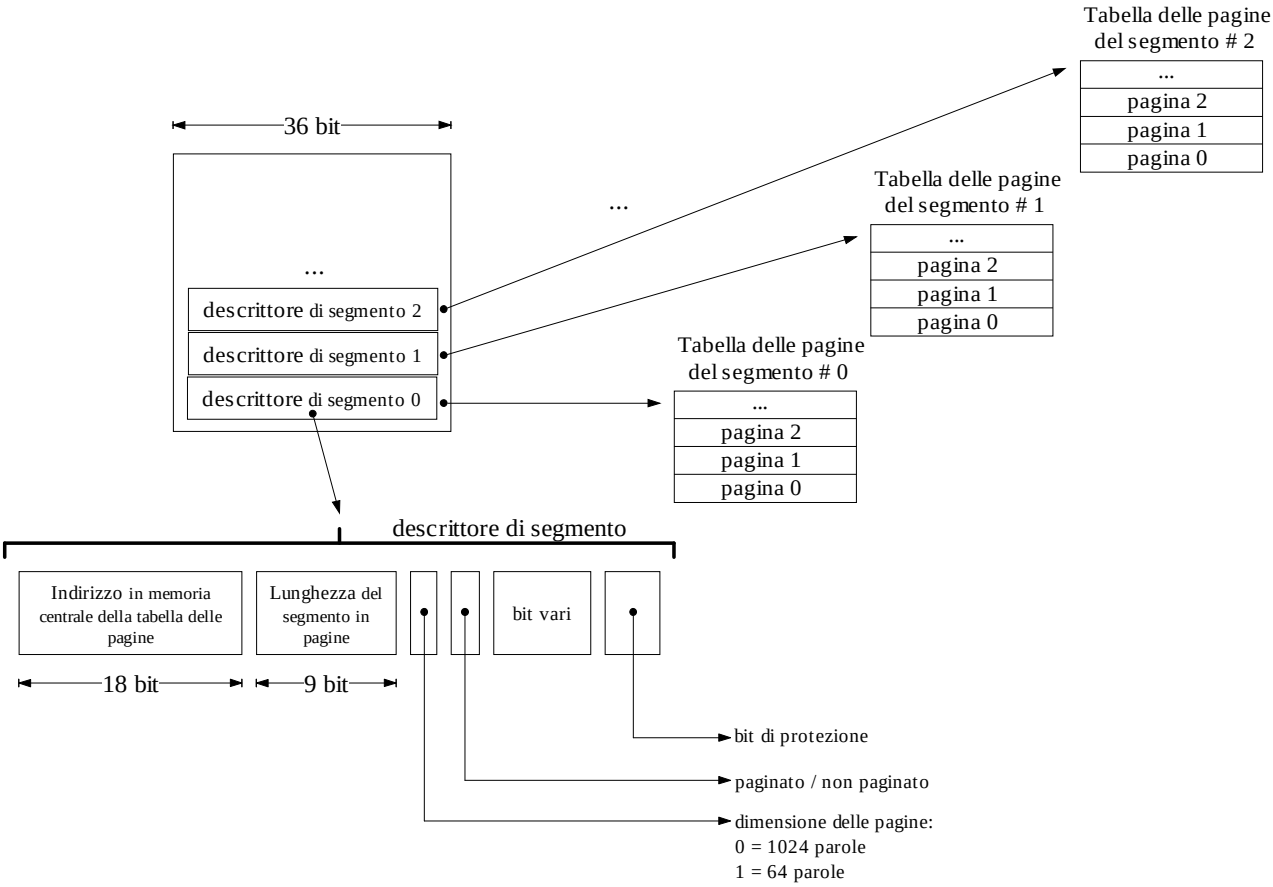
Considerazione	Paginazione	Segmentazione
Il programmatore deve sapere che viene usata questa tecnica?	NO	SI
Quanti spazi di indirizzamento lineari ci sono?	1	MOLTI
Lo spazio totale di indirizzamenteo può eccedere la dimensione della memoria fisica?	SI	SI
Le procedure e i dati possono essere distinti e proiettati separatamente?	NO	SI
Le tabelle a dimensione variabile possono essere gestite semplicemente?	NO	SI
Viene facilitata la condivisaione delle procedure fra gli utenti?	NO	SI
Perchè sono state inventate?	Per ottenere uno spazio di indirizzamento lineare molto grande senza dover comprare altra memoria fisica.	Per permettere di dividere programmi e dati in spazi di indirizzamento indipendenti e per aiutare protezione e condivisione.

E' possibile combinare segmentazione e paginazione: la segmentazione paginata consiste nel suddividere ciascun segmento in pagine di dimensione fissa e gestire una tabella delle pagine per ciascin segmento. Alla tabella delle pagine di un segmento si accede, quindi, con il numero di segmento (che lo identifica). Ogni record della tabella delle pagine del segmento è un descrittore di segmento di 36 bit (in MULTICS).
La dimensione delle pagine di unsegmento è di 1024 parole (il alcuni sistemi come MULTICS i segmenti sono paginati anche con unità più pissole, ed esempio 64 parole). Un indirizzo MULTICS si compone di due blocchi: un blocco fa riferimento al numero del segmento mentre l'latro si riferisce all'indirizzo interno del segmento che a sua volta è composto da un numero di pagina e da un offset all'interno della pagina:



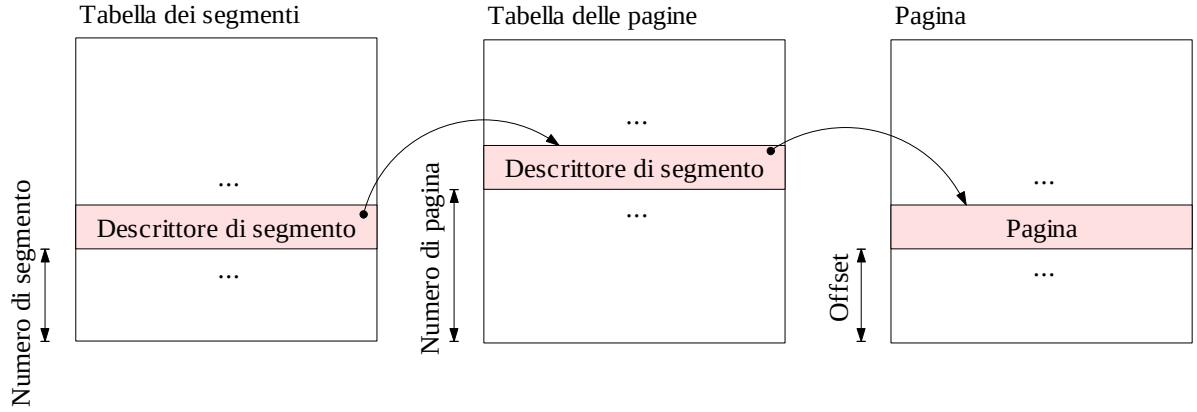
Indirizzo virtuale

Pertanto, in un sistema MULTICS esistono 2^{18} segmenti (262144) ed ognuno di essi è suddiviso in 2^6 pagine (64) da 2^{10} parole (1024) oppure da 64 parole.



Quando si verifica un riferimento alla memoria il gestore della memoria esegue le seguenti operazioni:

(a) dall'indirizzo virtuale viene estratto il numero di segmento che servità per accedere alla tabelle dei segmenti (quella contenente tutti i descrittori dei segmenti);
(b) si verifica dapprima se la tabella del segmento è già in memoria. Se è così la si localizza altrimenti si genera un fault di segmento;
(c) si usa il numero di pagina scritto nell'indirizzo virtuale per accedere ad una delle pagine del segmento;
(d) si aggiunge, infine, all'indirizzo di partenza della pagina l'offset scritto nell'indirizzo virtuale e si costruisce l'indirizzo fisico;



I file system

I processi, ma anche i thread, finora ampiamente discussi nel corso di queste stesse note hanno bisogno di memorizzare e di rintracciare le informazioni a loro utili e/o precedentemente elaborate. Essi, infatti, assolvono ad un importante compito: manipolano l'informazione per conto dell'utente, che impartisce loro dei comandi. La manipolazione dell'informazione avviene esclusivamente nello spazio di indirizzamento concesso in memoria fisica al processo e, in ogni caso, la capacità di memoria non può mai eccedere lo spazio di indirizzamento virtuale, quello cioè disponibile sul disco. Se per alcune applicazioni lo spazio riservato in memoria fisica può rivelarsi sufficiente, per alcune applicazione, invece, esso è addirittura troppo limitato (pensate ad esempio a tutte quelle applicazioni che interagiscono con le basi di dati ed a quanto siano estese queste basi di dati che devono quindi essere memorizzate sulla memoria virtuale e caricate a pezzi sulla memoria fisica). Ed ancora, l'informazione manipolata nella memoria fisica e poi scaricata sulla memoria virtuale deve avere un importante caratteristica, deve essere duratura nei confronti del processo. In altre parole un processo può terminare ma l'informazione da esso prodotta deve essere disponibile ad altri processi oppure alla stessa applicazione terminata e quindi riavviata successivamente. L'informazione deve essere conservata anche quando l'elaboratore si guasta oppure viene spento. Molto più brevemente possiamo, quindi, concludere dicendo che l'informazione va conservata e deve essere eliminata solo quando l'utente lo fa presente al sistema operativo.

L'informazione, come si sa, rimuove le incertezze all'interlocutore che la consulta. In alcuni casi essa può essere di tipo esclusivo, valida cioè solo per un solo utente. Tuttavia, è assai frequente anche la condivisione dell'informazione: più processi devono avere la possibilità di accedere alle informazioni in maniera concorrente (ecco a cosa serve la multiprogrammazione). La questione dell'informazione, delle sue caratteristiche in termini di spazio occupato, permanenza e condivisione viene gestita da strutture dati detti file. I file memorizzano l'informazione, ne permettono la condivisione e forniscono un adeguato spazio di indirizzamento. Quella parte del sistema operativo che si occupa dei file è detta file system.

Il file system fornisce una struttura dati utile alla gestione dei file, implementandoli e fornendo loro alcune caratteristiche come: il nome del file, il nome del proprietario, estensione, protezione ed alcune operazioni di gestione. Ogni sistema operativo crea nel proprio file system una propria astrazione di file (lo implementa cioè in un certo modo), per questo motivo esistono diversi file system ed ognuno di questi è gestito da un sistema operativo.

La prima caratteristica di un file system che osserveremo è la denominazione dei file, essa permette di rintracciare l'informazione univocamente quando il nome dato ad un file è unico in tutto il sistema di elaborazione. Un processo può creare un file ed assegnare a questo un determinato nome, quindi può iniziare a scriverne le informazioni che esso dovrà contenere. In una successiva esecuzione, lo stesso processo, può così richiamare il file precedentemente creato (adesso il processo conosce il nome del file poichè esso stesso lo ha creato oppure è l'utente che suggerisce al processo un determinato nome) e riprendere la manipolazione dell'informazione. Non esistono regole precise circa i nomi dei file, ogni sistema adottando un proprio file system è vincolato anche dalle regole che questo impone all'intero sistema. Ad ogni modo, tutti i file system di sistemi operativi consentono l'utilizzo di stringhe di caratteri come nome per file. Tali stringhe possono poi essere lunghe da uno ad otto caratteri e di recente sono ammesse anche stringhe lunghe 255 caratteri (in seguito vedremo come ciò si realizza). Alcuni sistemi consentono di inserire nel nome del file anche i caratteri speciali ed i simboli di interpunzione, altri sistemi ammettono invece solo i caratteri dell'alfabeto. I caratteri che compongono un nome possono poi essere riconosciuti nelle due forme di scrittura possibili, quella cioè in minuscolo ed in maiscolo sicchè ci sono sistemi che fanno differenza tra nomi scritti interamente in maiuscolo e/o minuscolo (come i sistemi UNIX che per questo motivo sono anche detti sensitive case) mentre altri sistemi ritengono uguali i nomi scritti nelle due possibili varianti (come ad esempio fanno i sistemi MS-DOS e quelli della famiglia WINDOWS).

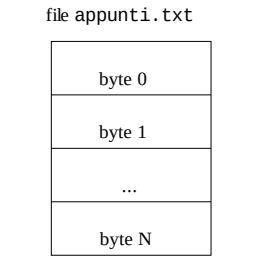
Il nome di un file è poi suddiviso in due pezzi o più pezzi, la prima parte dell'intero nome è il vero nome del file mentre le stringhe di caratteri che seguono il punto si riferiscono all'estensione del file. Nei sistemi con file system MS-DOS un nome intero di file è diviso in un nome di file avente lunghezza da uno ad otto caratteri e da una estensione che invece può essere lunga da una a tre

caratteri. L'estensione che accompagna il nome del file fornisce al sistema operativo delle interessanti informazioni aggiuntive. Abbiamo detto che un file contiene l'informazione, ma come leggerne o manipolarne il contenuto? Quando l'utente decide di mandare in esecuzione un file esso dapprima lo seleziona e successivamente lo manda in esecuzione premendo il tasto di invio (oppure si può anche dare un doppio click quando il puntatore del mouse punta sul nome del file). A questo punto il sistema operativo consulta una lista di associazione tra estensioni di file e programmi utente che hanno l'abilitazione ad eseguire il file che è stato scelto, quindi manda in esecuzione il programma utente abilitato ed apre il file selezionato. In questo caso l'estensione è servita al sistema operativo per rintracciare il programma utente capace di manipolare l'informazione nel file, questo accade ad esempio nei sistemi MS-DOS e WINDOWS.
Nei sistemi UNIX l'estensione del file può essere più articolata, il file può non avere alcuna estensione oppure può averne una o più. L'estensione è infatti (diversamente dai sistemi MS-DOS e WINDOWS) un informazione aggiuntiva data all'utente (e non al sistema operativo). Pertanto mentre un file nominato `appunti.txt` nei sistemi MS-DOS è aperto dal programma che detiene la gestione dei file con estensione `.txt` (ad esempio il programma utente `notepad.exe` oppure `wordpad.exe`) nei sistemi UNIX l'estensione `.txt` suggerisce all'utente il contenuto del file: un file di testo tipicamente codificato in ASCII (sarà l'utente che deciderà, poi, con quale programma utente andarlo ad aprire). Ecco un elenco di alcune estensione ed una loro breve descrzione:

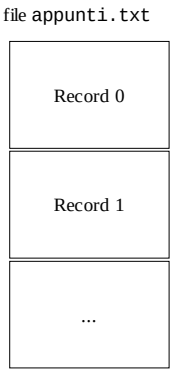
Alcune estensioni	**Descrizione**
.bak	File di backup
.c	File sorgente di un programma scritto in linguaggio C
.hlp	File di aiuto, solitamente una guida al programma utente
.html	File o documento formattato con tag HTML
.jpg	File di immagine codificata con algortimo standard JPG
.mp3	File audio compresso codificato con l'algoritmo MP Ver.3
.mpg	File video codificato con algoritmo standard MPEG
.o	File oggetto compilato ma non ancora collegato
.pdf	File di documento strutturato secondo il formato PDF
.ps	File di documento strutturato secondo il formato PS
.tex	File di input per il programma TEX di UNIX
.txt	File di testo generico codificato in ASCII
.zip	File di archivio compresso codificato con l'algoritmo ZIP
.rar	File di archivio compresso codificato con l'algoritmo RAR

Detto ciò, procediamo nella trattazione dei file occupandoci adesso della struttura dei file. Esistono tre possibili strutture di file:

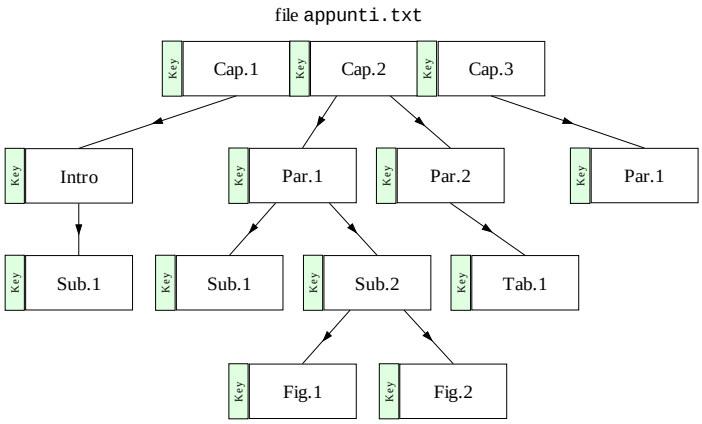
- sequenza di byte non strutturata: il file è visto dal sistema operativo cone una sequenza di byte. Si tratta dell'implementazione più flessibile dal momento che l'utente può inserire nel file qualunque cosa, qualunque byte. Dall'altra parte, tuttavia, il sistema operativo non fornisce alcun supporto ai file: non impedisce i nomi troppo lunghi ai file, non pretende un estensione, non limita e/o evita il contenuto del file. Nonostante ciò è la struttura di file usata dai sistemi UNIX ed MS-DOS;
- sequenza di record: si tratta dell'evoluzione della precedente implemetazione. Il file è una sequenza di record avente una lunghezza fissa. Ogni record ha poi una propria struttura ed organizzazione interna. Per questo motivo la scrittura, oppure la lettura, del file viene indirizzata indicando un numero di record e la colonna (questo sistema era ad esempio utilizzato nei vecchi sistemi con schede perforate in cui ogni scheda 132 colonne ed ognuna di essa aveva la capacità dicontenere 80 caratteri!);
- struttura ad albero: il file è formato da un albero di record non tutti della stessa lunghezza. Ogni record ha poi un campo chiave e l'albero è ordinato rispetto a tale chiave. A differenza della precedente struttura che accede al file accedendo man mano al record successivo nella struttura ad albero l'accesso è assai più veloce: si accede, infatti, ad una parte del file fornendo un opportuna chiave che identifica il blocco di interesse;



sequenza di byte non strutturata

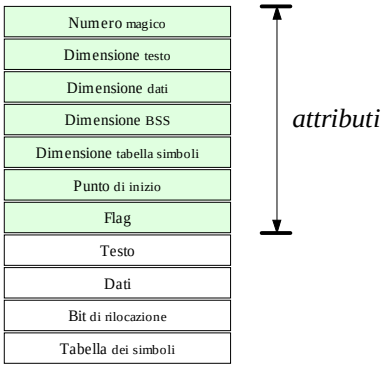


Record

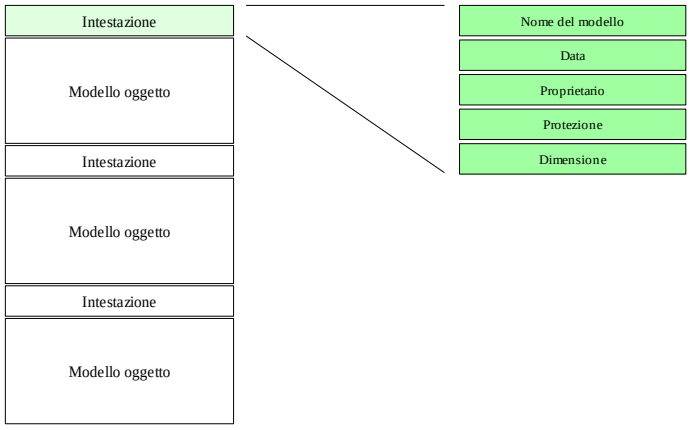


Albero

Ogni sistema di calcolo adotta un file system per modellare l'entità file, un sistema operativo può tuttavia comprendere diversi formati e/o tipi di file. Nei sistemi WINDOWS ed UNIX, ad esempio, esistono file regolari, file speciali e directory. I file regolari sono quelli che contengono l'informazione e sono strutturati secondo le volontà del programmatore che decide quindi l'organizzazione dell'informazione. Per ogranizzare e tenere ordinati i file sono poi necessari particolari file, le directory. Esse conservano e mantengono la struttura del file system, tipicamente ad albero, ed organizzano i file in più sottodirectory. In UNIX, inoltre, esistono file speciali che modellano alcune periferiche di input e di output.
L'informazione nei file regolari può essere formattata secondo lo standard ASCII, in tal caso la stampa del file ne riproduce esattamente il contenuto. In alcuni casi, invece, i file regolari sono file di tipo binario. L'informazione è in tal caso mascherata o codificata nel file e solo il programmatore, oppure il programma utente ne conosce la struttura interna e quindi la collocazione. Ad esempio, in un file binario eseguibile di UNIX prendono posto diverse informazioni aggiuntive oltre all'informazione vera e propria. Il file ha più sezioni informative: intestazione, testo, dati, bit di rilocazione e tabella dei simboli. Nell'intestazione del file binario sono suggerite le dimensioni del testo, qualche flag e/o bit protezione e gli indirizzi di partenza nel file di alcuni blocchi informativi.



file binario eseguibile di UNIX



file binario di tipo archivio di UNIX

In un file binario di tipo archivio sono impacchettati tutti i moduli e le procedure compilate e non ancora collegate. Ogni modulo è preceduto da una intestazione (anche questa in forma binaria). Inutile dire che la stampa di un file binario provoca in uscita delle stampe incomprensibili dal momento che l'informazione è codificata.

L’accesso ad un file può avvenire in due modi:

- accesso sequenziale, è la modalità di accesso prevista ad esempio per i nastri magnetici. Prima di accedere ad un record bisogna leggere tutti quelli che lo precedono. Ciò comporta un enorme tempo di attesa e per questo motivo si usa quasi sempre l’altra modalità di accesso;
- accesso casuale, permette di accedere ad un record dell’informazione senza leggere quelli precedenti. Per fare ciò si fornisce ad una istruzione particolare quale è `seek` l’indirizzo di partenza, in questo modo una successiva operazione di lettura avrà inizio a partire dal suddetto indirizzo;

Ogni sistema prevede per i file una serie di attributi, quelli maggiormenti usati riguardano la data e l’ora di creazione, la dimensione del file, la data e l’ora dell’ultimo accesso al file, alcuni flag per i permessi ed altre informazioni utili al sistema oppure all’utente. Ogni sistema adotta diversi attributi, nella tabella che segue è possibile osservare una lista completa dei possibili attributi di file:

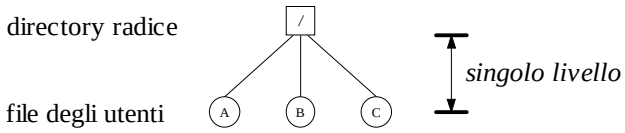
Attributo	Significato
Proprietario	Indica l’utente che ha generato il file
Dimensione corrente	Esprime la dimensione in byte del file
Dimensione massima	Esprime la massima dimensione in byte
Protezione	Decide chi può accedere al file
Password	Parola chiave per l’accesso al file
Flag di lettura	0=lettura e scrittura 1=sola lettura
Flag di sistema	0=file normale 1=file di sistema operativo
Flag di archivio	0=file non copiato 1=file con copia
Flag di file nascosto	0=file normale 1=non mostrare il file
Flag di ASCII	0=file ASCII 1=file binario
Flag di bloccaggio	0=file accessibile 1=file bloccato
Lunghezza di un record	Indica la dimensione di un record in byte
Lunghezza di una chiave	Indica lo spazio occupato da una chiave
Tempo di creazione	Data e ora di creazione del file
Tempo di ultimo accesso	Data e ora dell’ultimo accesso al file
Tempo di modifica	Data e ora dell’ultima modifica al file

Alcune system call che trattano i file sono già state affrontate nel corso di queste stesse note, per maggiore completezza ne riportiamo di seguito un elenco riassuntivo:

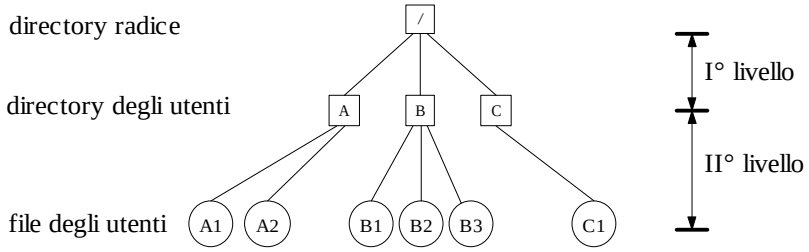
- `create`: crea un file senza dati, predispone tutte le strutture necessarie al file;
- `delete`: elimina un file dopo averne indicato il nome;
- `open`: apre un file in una delle modalità previste, lo scopo della chiamata è la predisposizione in memoria centrale di tutte le strutture dati del file;
- `close`: in alcuni sistemi esiste un limite massimo al numero dei file che possono essere aperti, questa chiamata di sistema chiude il file e libera in questo modo lo spazio in memoria principale;
- `read`: legge un certo numero di byte dal file;
- `write`: scrive i byte nel file sovrascrivendo quelli già presenti nel file;
- `append`: scrive i byte nel file collocandoli in coda a quelli già presenti nel file;
- `get attributes`: restituisce gli attributi del file disponendoli in una apposita struttura dati;
- `set attributes`: modifica un attributo del file;
- `rename`: cambia il nome ad un file;

La sola esistenza dei file non garantisce l’ordine desiderato dall’utente per la memoria virtuale ma al contrario si limita a contenere i file in unica directory che per questo motivo viene detta directory principale o directory radice. In tal caso non possono esistrere due file con lo stesso nome. Le directory sono particolari file che contengono altri file, in questo modo ogni utente può avere la

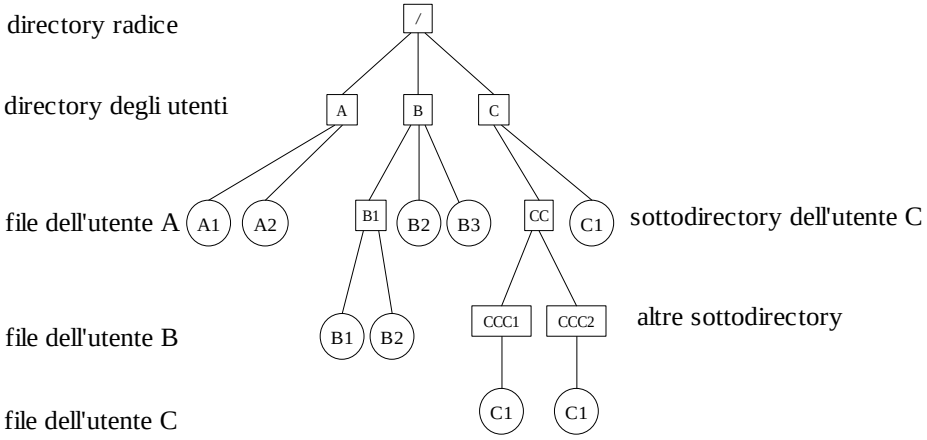
propria directory ed inserire, pertanto, i propri file. La soluzione con un unica directory è più semplice da realizzare e qualora debba essere cercato un file esisterà una sola directory in cui cercare.



La possibilità di generare altre directory per ordinare i file da luogo a diversi livelli di directory: in ogni directory può essere creata un altra directory che per questo motivo si dirà essere una sottodirectory della directory superiore:



La gerarchia a livelli delle directory elimina i conflitti che possono sorgere sui nomi dei file, ogni utente può tuttavia avere l'esigenza di un ulteriore livello di astrazione per le directory, magari perchè vuole un secondo file con lo stesso nome di uno già esistente. Per questo motivo i moderni sistemi consentono una gerarchia di directory che permette di avere un numero arbitrario di sottodirectory:



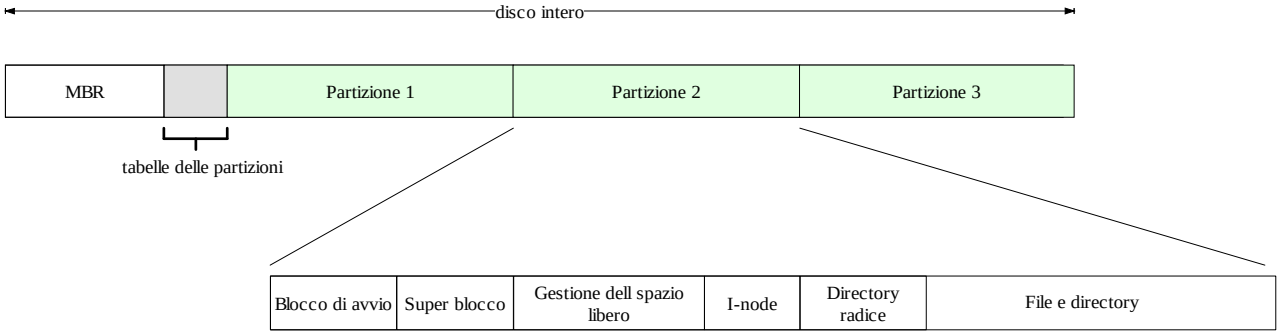
Quando il file system è organizzato su una gerarchia di directory, tipicamente detta ad albero, è necessario una procedura sistematica che permetta di individuare un file. Un primo metodo prevede l'uso del cosiddetto path assoluto: per indicare un file si parte sempre dalla direcotory radice (che per i sistemi UNIX è / mentre per i sistemi MS-DOS è C:\), quindi si elencano tutte le sottodirectory intermedie fino al file (neisistemi MS-DOS il separatore di directory è il simbolo \, nei sistemi UNIX il separatore è il simbolo /). Ad esempio, il path assoluto: `/c/cc/ccc1/c1` indica il file `c1` della directory `ccc1` che è una sottodirectory della directory `cc` che a sua volta è una sottodirectory della directory `c` che, per finire, è una sottodirectory della directory radice.
Un altro metodo per indicare un path di file fa uso del concetto di directory di lavoro. Per directory di lavoro si intende l'attuale directory selezionata dall'utente (nei sistemi UNI ad esempio, l'utente si sposta dalla directory radice alla directory `bin` mediante il comando di change directory `cd bin`). Dopo aver eseguito il precedente comando la directory di lavoro che prima coincideva con la directory radice è la directory `bin`. Nei sistemi UNIX il comando `pwd` restituisce l'attuale directory

di lavoro. Se ad esempio ci troviamo nella directory di lavoro `cc` il pathname relativo `/ccc1/c1` indica il file `c1` della directory `ccc1` che è una sottodirectory della directory di lavoro `cc`. Alcune system call che hanno a che fare con le directory sono state già trattare nel corso di queste stesse note, tuttavia preferiamo riportare per completezza un elenco riassuntivo delle possibili operazioni su directory:

- `create`: crea una directory vuota (`mkdir` per i sistemi UNIX);
- `delete`: elimina una directory;
- `opendir`: apre una directory;
- `closedir`: chiude una directory;
- `readdir`: legge da un elemento il prossimo elemento, è utile a fare una lista di file;
- `rename`: cambia il nome ad una directory;
- `link`: il collegamento permette ad una directory di comparire in altre sottodirectory;
- `unlink`: rimuove il collegamento;

L'utente, che sicuramente userà i file e le directory, si preoccupa di come i file sono chiamati ed in quale path essi vengono memorizzati. Coloro che invece studiano i file e le directory si preoccupano (almeno dovrebbero farlo), invece, di come questi vengono implementati.

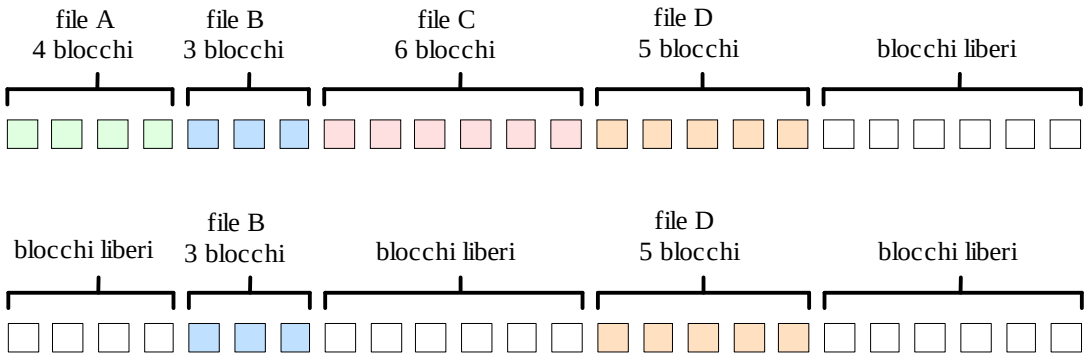
Implementazione di un file system
Il file system può essere visto secondo due punti di vista, quello dell'utente e quello del sistema operativo. L'uente riceve dal file system le astrazioni necessarie per i file e per le directory, a lui spettano le assegnazioni dei nomi per i file e per le directory nonchè la struttura gerarchica per le directory. Il sistema operativo, invece, si prende cura di come implementare in maniera efficiente delle opportune strutture dati, di come deve avvenire la gestione tramite le system call e di come lo spazio sul disco debba essere mantenuto. I file system vengono memorizzati sui dischi, sia essi magnetici oppure ottici come i cd-rom, ogni disco è diviso in partizioni, ed ogni partizione del disco può avere un proprio file system. Il settore 0 di ogni disco è detto master boot record o più brevemente MBR, in esso è contenuto il programma per mandare in esecuzione il sistema operativo. Verso la fine dell'MBR si trovano le tabelle delle partizioni, ogni partizione ha un blocco di avvio ed altri blocchi informativi. Una sola partizione è segnata come attiva per cui quando il sistema viene avviato il BIOS legge ed esegue il programma contenuto nell'MBR che a sua volta individua la partizione attiva ed esegue il blocco di avvio di quest'ultima. Nel blocco di avvio di ogni partizione, solitamente, si trova il loader del sistema operativo. A parte il blocco di avvio che è presente in ogni partizione, la struttura dei blocchi di partizione varia a seconda del file system utilizzato dal sistema.



Per implementare un file in file system sono possibili diverse strategie:

Allocazione contigua
Nell'allocazione contigua lo spazio di indirizzamento dell'intero disco è suddiviso in blocchi e le operazioni di lettura e scrittura interessano uno o più blocchi. Ad ogni file viene assegnato un certo numero di blocchi consecutivi, di questi solo l'ultimo blocco a volte non è interamente riempito (ad

esempio perchè il file non è un multiplo esatto della dimensione del blocco di disco). Per la lettura/scrittura è sufficiente conoscere l'indirizzo del blocco iniziale del file. Ad esempio, se un disco è diviso in blocchi da 1Kb allora un file utente di 100Kb sarà allocato in uno spazio contiguo del disco fisso di 100 blocchi. Se la dimensione dei blocchi sul disco è invece di 2Kb occorreranno 50 blocchi contigui. L'implementazione dei file secondo uno schema a blocchi contigui è senza dubbio la più facile dal punto di visto realizzativo: per tenere traccia dei blocchi di un file è sufficiente conoscere l'indirizzo del primo blocco ed il numero di blocchi che lo compongono;la posizione dei successivi file dipende dalla dimensione del singolo blocco e da quanti blocchi lo precedono; la lettura del file può avvenire poi alla massima velocità poichè i blocchi del file sono contigui, pertanto, per leggere un file è necessaria una sola operazione di posizionamento sul blocco iniziale.
Nonostante la semplicità realizzativa la soluzione dei blocchi contigui presenta alcuni inconvenienti abbastanza significativi. Un primo problema, come già anticipato, è il mancato utilizzo dell'ultimo blocco che sarà sempre parzialmente occupato. Meglio quindi non fare i blocchi del disco troppo grandi, si rischia di inutilizzare gran parte dello spazio di un blocco se il file non è esattamente un multiplo della dimensione del blocco. Inoltre, se il sistema è in esecuzione per un certo periodo di tempo, la rimozione di alcuni file rischia di frammentare il disco. In altre parole i buchi dovuti agli spazi liberati dai file adesso non più utili non sempre coincidono con gli spazi necessari ad allocare i nuovi file. Se questi possono essere in alcuni casi accodati all'ultimo blocco del disco non ancora utilizzato prima o poi sarà necessaria una operazione di compattazione, senza dubbio dispendiosa in termini di tempo. Lo spazio ancora inutilizzato potrebbe allora essere organizzato meglio: i buchi liberi ad esempio potrebbero essere ordinati per dimensionare, in tal caso prima di allocare un nuovo file si dovrebbe dapprima conoscere la dimensione finale del file in maniera tale da avviare una ricerca del buco sul disco. Nonostante questa forte limitazione l'allocazione contigua è ancora usata come ad esempio avviene nella scrittura del file system per cd-rom. In questo caso, davvero singolare, è infatti possibile conoscere a priori le dimensioni dei file che si intendono scrivere sul cd-rom, oltre al fatto che le dimensioni dei file, una volta scritte sul cd-rom non cambieranno mai più.

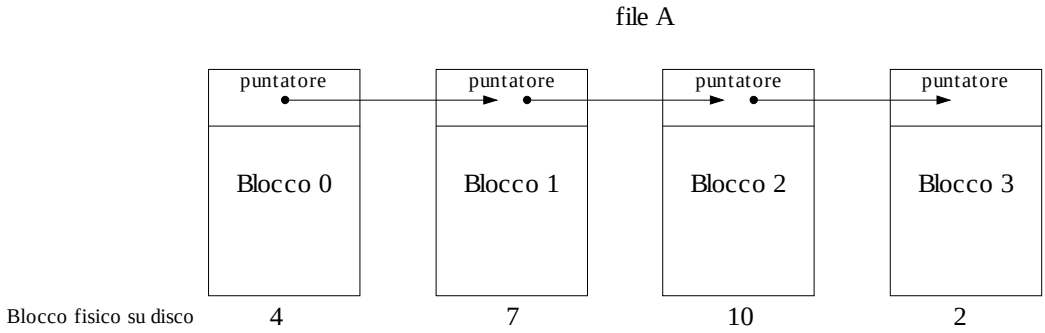


Allocazione con lista concatenata
Nell'allocazione contigua è emerso il problema della frammentazione del disco, così come già fatto per la gestione della memoria principale, è possibile allocare i blocchi del disco secondo una lista concatenta. Ogni blocco ha una parola che viene destinata al puntatore al prossimo blocco, la restante parte del blocco è quindi destinata alla memorizzazione dell'informazione. Così facendo si elimina il problema della frammentazione e si può usare ogni blocco del disco, anche se questo appartiene ad un buco troppo piccolo per l'allocazione contigua. Tuttavia l'allocazione dei blocchi del disco secondo lo schema delle liste concatenate ha alcuni svantaggi:

- se una parola del blocco o comunque una sua frazione è destinata al puntatore al prossimo blocco allora la restante parte da destinare all'informazione non è più una potenza di due. In altre parole lo spazio realmente dedicato all'informazione non coincide con la dimensione del blocco. Si tratta di una complicazione forte per la programmazione dal momento che molti programmi usano leggere e scrivere blocchi di informazione che sono multipli della

potenza di due;

- l'eliminazione dei problemi dovuti alla frammentazione è ulteriormente svantaggiosa poichè l'accesso ad un blocco del file, ad esempio l'n-esimo blocco, avviene solo dopo n-1 accessi al file. In altre parole è necessario scorrere l'intera lista dei blocchi prima di arrivare al blocco desiderato (nell'allocazione contigua la lettura richiedeva, invece, la conoscenza dell'indirizzo del primo blocco del file e la lettura poteva quindi avvenire alla massima velocità vista la contiguità dei file);



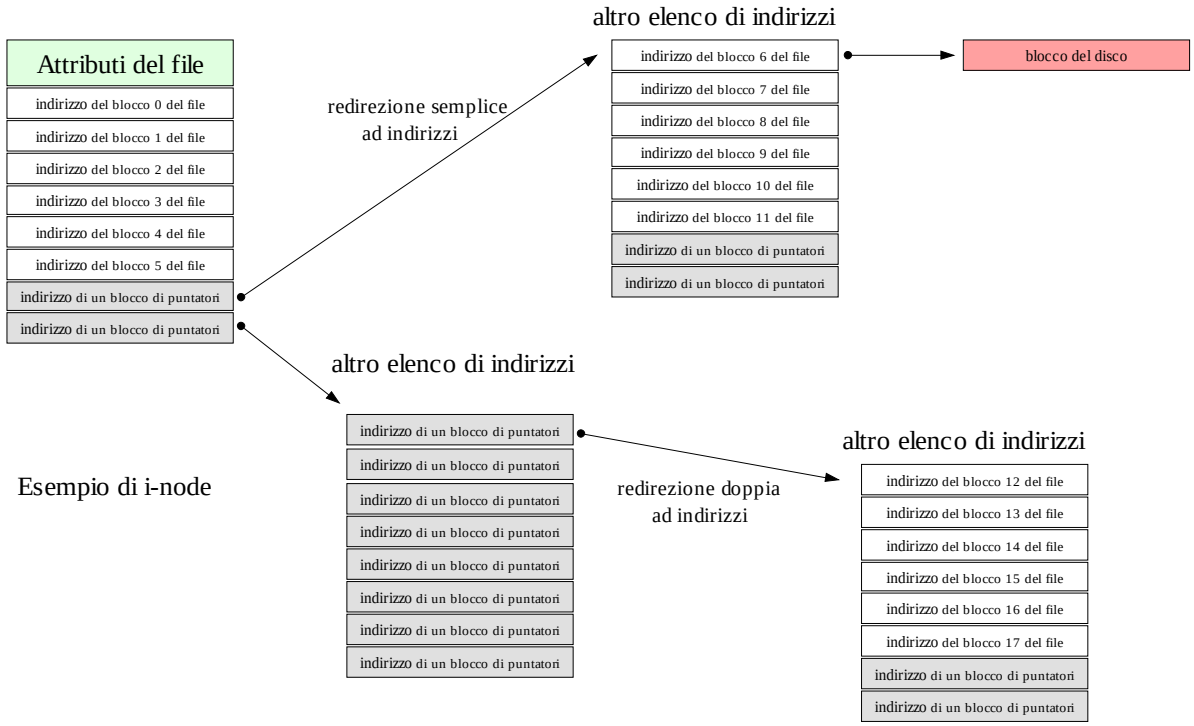
Allocazione con lista concatenata e tabella in memoria

Entambi gli svantaggi analizzati per lo schema di allocazione con lista concatenata possono essere rimossi, la soluzione consiste nel mettere i puntatori di un file al blocco successivo in una tabella. La tabella ha tante righe per quanti sono i blocchi del disco, in corrispondenza di ogni riga (che fa riferimento ad un blocco fisico) si segna il successivo blocco che lo segue. In questo modo è possibile mappare su questa tabella l'allocazione dei blocchi di un file, la terminazione di un file è segnata con un simbolo speciale, ad esempio -1. In questo modo è possibile seguire tutti i blocchi di un file e l'informazione occupa tutto lo spazio messo a disposizione dal blocco. La tabella, poi, risiede in memoria principale, pertanto, le operazioni di scorrimento della lista concatenata fino al blocco di file interessato possono avvenire molto più velocemente. La tabella in memoria è detta file allocation table o più brevemente FAT. Qui lo svantaggio è che la dimensione della tabella è proporzionale alle dimensioni del disco! Se il disco è da 60 GB e se i blocchi sono da 1 KB la FAT avrà $60 \cdot 2^{20}$ righe, uno per ogni blocco del disco che dovrà così avere un puntatore al prossimo blocco di $\log_2 60 \cdot 2^{20} \approx 4$byte. La FAT occuperà per questo motivo almeno 240 MB ($4 \cdot 60 \cdot 2^{20}$ byte). Il file A dell'esempio precedente con lista concatenata è stato qui allocato in una FAT per disco da 16KB con blocchi da 1KB, il file A inizia a partire dal blocco 4 del disco:

0	Blocco non usato
1	Blocco non usato
2	-1
3	Blocco non usato
4	7
5	Blocco non usato
6	Blocco non usato
7	10
8	Blocco non usato
9	Blocco non usato
10	2
11	Blocco non usato
12	Blocco non usato
13	Blocco non usato
14	Blocco non usato
15	Blocco non usato

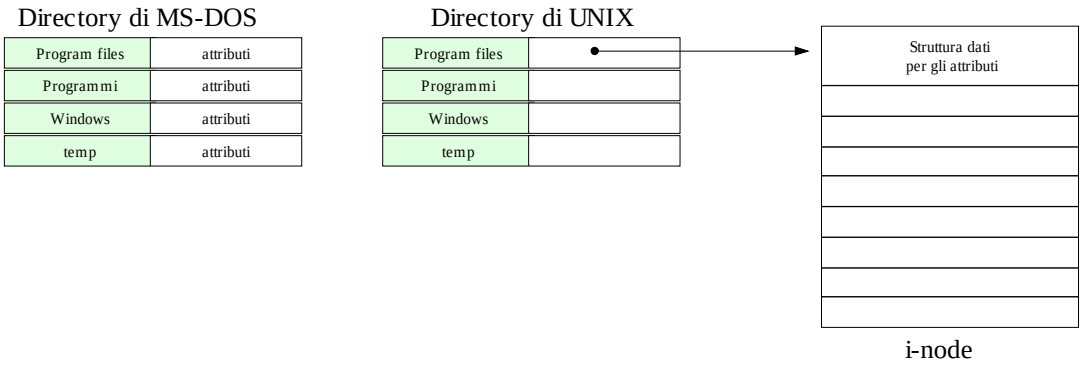
Il blocco 2 del disco non punta a nessun blocco successivo, ed essendo la terminazione del file A è quindi contrassegnato con il simbolo -1.

Allocazione con i-node
L'allocazione dei file basata su i-node permette di tenere traccia dei blocchi appartenenti ad un file associando al file una piccola struttura dati che è appunto l'i-node (index node, nodo indice). L'i-node è una piccola tabella o array in cui scrivere i blocchi di disco associati al file e gli attributi che lo caratterizzano. Siccome il file potrebbe crescere oltre il limite previsto dall'array (oppure potrebbe già impegnare un numero di blocchi del disco il cui elenco è tale da superare i posti a disposizione nell'array) si decide di associare alcune delle celle dell'array ad indirizzi di blocchi del disco che contengono altre di queste strutture dati. L'i-node è assai più vantaggioso dell'allocazione con tabella FAT poichè la struttura dati adoperata è molto più snella (si tratta di un array con un fissato numero di celle), innanzittutto l'array non è proporzionale alle dimensioni del disco come invece è la FAT. Altra cosa vantaggiosa è che l'i-node deve essere caricato in memoria solo quando il file corrispondente viene aperto, la FAT invece deve trovarsi costantemente in memoria.

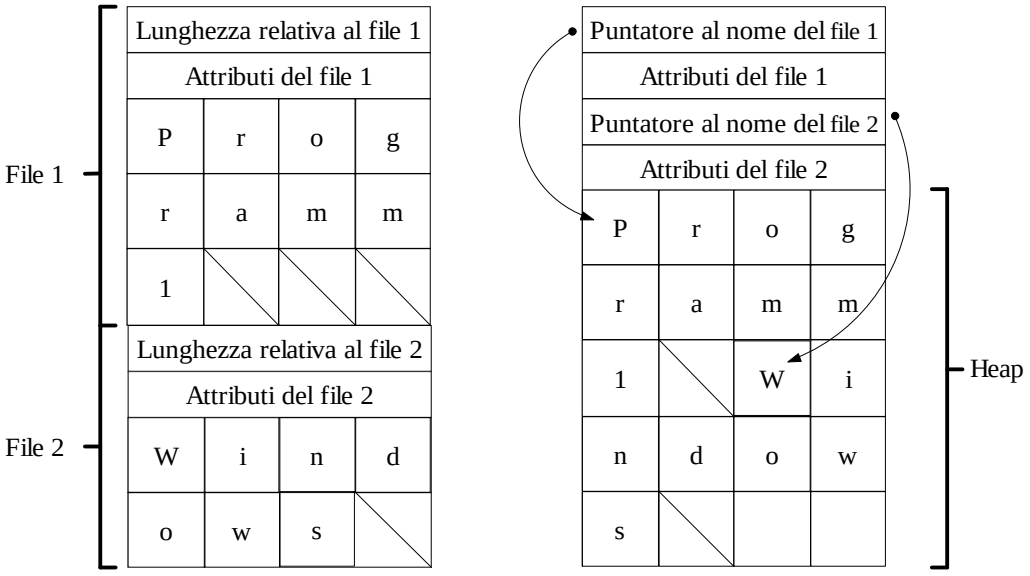


Esempio di i-node

Per implementare una directory in un file system sono possibili due diverse strategie, entrambe si differenziano per come vengono gestiti gli attributi e come avviene la gestione dei nomi lunghi. La directory è il contenitore per i file, quando un file deve essere aperto il sistema operativo lo rintraccia tramite il path di directory che conducono ad esso (se il file è modellato come una collezione contigua di blocchi) oppure cercando il numero di i-node (se il file è modellato con i-node). Gli attributi del file possono essere memorizzati direttamente nella directory che quindi prevede una struttura di elementi aventi una lunghezza fissata per contenerli (nome, dimensione e date), tale approccio è ad esempio seguito nei file system basati su FAT. Nei sistemi che invece usano file system basati su i-node viene seguita una diversa possibilità, gli attributi sono memorizzati nell'i-node, così facendo l'elemento di directory è assai più snello poichè contiene solo il nome del file nella directory ed il suo i-node.



Le directory sono dunque viste come dei normali file. Nei sistemi con FAT esse sono viste come degli "entry point" verso i blocchi iniziali dei file in essa contenuta, ricordiamo infatti che la tabella contiene il nome del file, gli attributi e la locazione del primo blocco. Per i sistemi basati su i-node le directory contengono gli indirizzi dei blocchi con gli i-node dei file contenuti.
Altra questione da trattare è quella che riguarda i nomi lunghi nei sistemi MS-DOS i file hanno una lunghezza che varia da 1 a 8 caratteri ed una estensione opzionale da 1 a 3 caratteri, mentre nei sistemi UNIX Ver.7 sono supportati nomi da 1 a 14 caratteri, incluse le estensioni. Tuttavia è ben nota la possibilità di memorizzare nome più lunghi come attualmente fanno i moderni calcolatori. Una possibile soluzione consiste nel fissare una lunghezza massima per i nomi, ad esempio 255 caratteri, ed usare uno dei due schemi, quelli con FAT oppure i-node.
Se decido di assegnare un certo spazio (ad esempio un certo numero di campi dati di n byte per ogni file, in base alla lunghezza del nome del file saranno necessari un certo numero del suddetto campo dati) nella directory per memorizzare i nomi lunghi dei file posso tuttavia sprecarlo inutilmente se questi poi non lo impegnano effettivamente. Oltretutto, poi, se decido di rimuovere un file rimarrà un buco in tale struttura pari al numero di elementi di lunghezza fissa ad esso assegnato che non sempre coincide con il numero di elementi richiesti dal nuovo file.
Una diversa soluzione è invece quella che prevede la scrittura dei nomi dei file in unico heap della directory: ogni elemento che fa riferimento ad un file inizia con un puntatore all'heap della directory verso il nome del file rappresentato. In questo caso la directory occupa lo spazio che effettivamente richiede e la rimozione di un file provoca l'aggiornamento dei puntatori (operazione che tra l'altro avviene velocemente dal momento che l'elemento di directory è caricato in memoria).

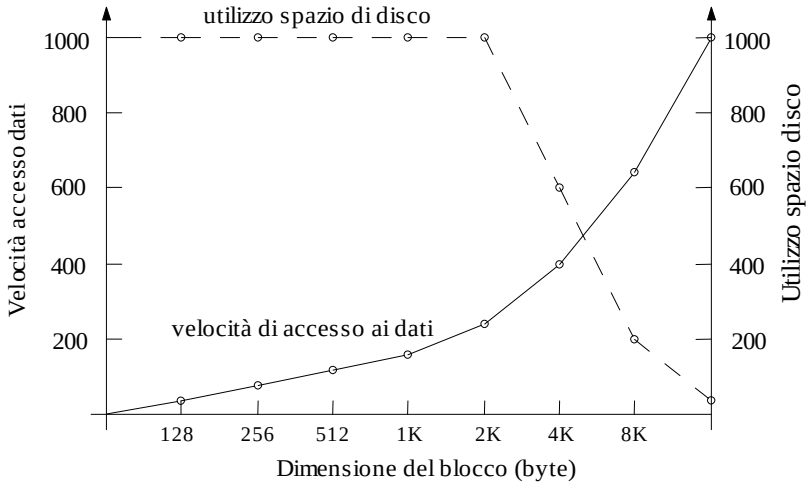


In tutte le configurazioni descritte fino a questo momento, le ricerche di un nome di file avvengono in modo lineare dall'inizio alla fine della directory; per directory estremamente lunghe la ricerca

lineare può essere molto lenta. Un modo per velocizzarla è quella di usare una tabella hash all'interno di ogni directory; se la lunghezza della tabela è *n*, per inserire il nome di un file, si deve calcolare partendo da tale nome un valore da *0* ad *n-1*. Ad esempio dividendolo per *n* e prendendo il resto, oppure sommando le parole che compongono il nome e dividendo la quantità risultante per *n*, oppure qualche oprazione simile. In ogni caso si ispeziona la componente della tabella corrispondente al codice calcolato, se tale componente è libera, si aggiunge un puntatore all'elemento del file in questione, così gli elementi dei file seguono la tabella di hash per rintracciarlo. Se la componente è già in uso, si crea una lista collegata, la cui testa è raggiungibile dalla tabella, composta da tutti gli elementi che corrispondono allo stesso valore di hash.
Da quanto finora detto appaiono evidenti le due possibili strategie per la gestione dello spazio di un disco. I file possono essere memorizzati in blocchi contigui, oppure possono essere divisi in un numero di blocchi non necessariamente contigui. Un compromesso simile si era presentato nella gestione della memoria e trovava le soluzioni nella memoria paginata o nella frammentazione. Una sostanziale differenza risiede nel fatto che memorizzando i file in maniera contigua si presenta il problema di spostare il file qualora esso cresca in dimensioni, nella segmentazione ciò non costituiva un grosso problema poiché la memoria principale è assai più veloce del disco, ragion per cui l'operazione di spostamento dei file avviene con molta più velocità. In base a quanto detto la maggior parte dei file system suddivide i file in blocchi di dimensione fissa che non necessariamente sono tra loro contigui. Una scelta assai importante nella progettazione di un file system è la dimensione del blocco di allocazione. Bisogna trovare il giusto compromesso tra l'overhead dovuto alla frammentazione interna (che richiederebbe blocchi più piccoli) e velocità di trasferimento dei dati che invece richiede blocchi grandi. Infatti, i blocchi piccoli hanno il vantaggio di generare una frammentazione che meglio si adatta alla memorizzazione dei file poiché generalmente solo l'ultimo blocco sarà parzialmente occupato, tuttavia avere un blocco piccolo implica anche una tabella FAT più grande (ricordiamo che la tabella FAT ha tante righe per quanti sono i blocchi del disco). Al contrario invece, un blocco dalle dimensioni grandi permette di recuperare l'informazione nel blocco con poche operazioni di lettura ma produce più spazio sprecato. Le operazioni di lettura/scrittura di un disco sono caratterizzate da un tempo di rotazione (medio poichè varia con la distanza periferica del blocco dal centro del disco) e da un tempo di ricerca del blocco. Ad esempio, se un disco di 20 GB ha una traccia di 131072 byte, un tempo di rotazione di 8.33 ms ed un tempo di ricerca di 10 ms, il tempo necessario a leggere un blocco di K byte è:

$$T = 10 + \frac{8.33}{2} + \frac{K}{131072} \cdot 8.33$$

Per blocchi da 1 KB il tempo è di 14.23 ms, per blocchi da 4 KB il tempo è di 14.42 ms e così via...



La velocità di accesso al file cresce all'aumentare delle dimensioni del blocco (si leggono/scrivono

più informazioni in un colpo solo). Se il file è però troppo grande la velocità di accesso è fortemente condizionata dal tempo di trasferimento (ci vuole più tempo per reperire l'intero file ed il tempo di trasferimento fa abbassare la velocità di accesso). I blocchi piccoli, invece, vanno bene per un buon utilizzo dello spazio del disco. Per questo motivo ogni file system deve arrivare a scegliere un compromesso per la dimensione dei blocchi del disco, tale valore si attesta in un intorno dei 2 KB (come diverse statistiche confermano).
Per tenere traccia dei blocchi liberi si usano due metodi: la lista concatenata dei blocchi liberi e la tecnica della mappa dei bit liberi (come per la memoria principale). Nella lista concatenata ogni blocco contiene una lista di blocchi del disco che sono liberi. Se la dimensione del blocco è di 1024 byte e se per indirizzare ogni blocco occorrono 32 bit (ci sono cioè 2^{32} blocchi sul disco), in ogni blocco ci possono essere 256 numeri di blocchi liberi (ecco il calcolo: $1024 \cdot 8 = 8192$ sono i bit in un blocco; l'indirizzamento ad un blocco libero avviene con 32 bit, in un blocco sono contenuti $8192/32 = 256$ indirizzi di blocchi liberi). La mappa dei bit occupa meno spazio della lista poiché usa 1 bit per blocco contro i 32 bit della lista (32 bit sono i bit usati dalla lista concatenata per indicare in ogni blocco gli indirizzi di altri blocchi liberi). Un blocco libero è segnato nella mappa di bit con il simbolo 1, un blocco già allocato è segnato, invece, com il simbolo 0. Un disco di 20 GB ha $20 \cdot 2^{20}$ blocchi da 1 KB ed una mappa di bit di altrettanti $20 \cdot 2^{20}$ bit ed occuperà, quindi, $20 \cdot 2^{20} / 8 \cdot 2^{10} = 2560$ blocchi da 1 KB (2.5 MB). Tuttavia nel caso in cui il disco sia quasi pieno allora la mappa dei bit sarà quasi completa e richiederà più spazio della lista concatenata.
Se viene scelto il metodo della lista concatenata solo un pezzo della lista sarà portato in memoria principale (in tal caso il sistema operativo disporrà di 256 indirizzi di blocchi liberi), quando un file viene creato si accede a tale blocco per reperire gli indirizzi dei blocchi liberi da assegnare. Quando il suddetto blocco di indirizzi si esaurisce si provvede a leggerne un altro dal disco.
Un inconveniente per la bit-map è che essa va tenuta interamente in memoria. Affinché un utente non usi tutto lo spazio del disco si usa assegnare a ciascun utente una quota sul disco, i sistemi operativi con molti utenti offrono questo meccanismo ed introducono due limiti di superamento, uno detto hard (rigido) e l'altro detto soft (variabile). Si concede all'utente la possibilità di superare il limite soft a patto che alla fine della sessione l'utente rientra nella quota a lui assegnata. Il limite hard è invece invalicabile. Nel tenere traccia dei blocchi liberi si cerca prevenire l'esaurimento della lista dei blocchi liberi anticipando il caricamento in meoria di un ulteriore lista.

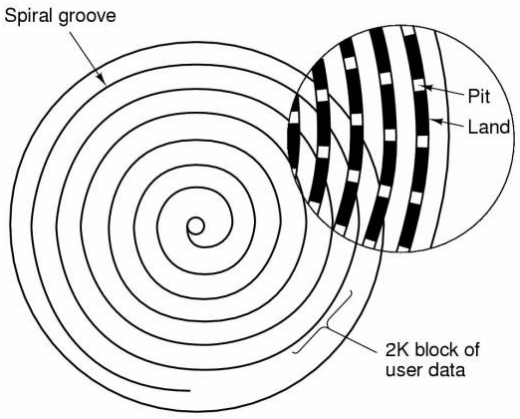
Affidabilità di un file system
Quante copie avete dei vostri file? La perdita dei file e quindi dell'informazione che essi trasportano non è quasi sempre rimediabile e comporta oltretutto un eccessivo tempo di recovery dell'hard disk. A differenza di qualunque altro componente di un sistema, la cui sostituzione può avvenire nel giro di qualche ora rivolgendosi ad un rivenditore, la sostituzione di un hard disk ormai danneggiato se pur possibile non farà contento l'utente che avrà in mente la lista di tutti i file persi. L'inaffidabilità di un file system può essere prevenuta con le copie di backup dei file. Se il danno per un utente privato che perde i propri file è già di per se una catastrofe, figuriamoci allora quanto lo sia per le aziende. Per questo motivo, chi lavora intensamente con un sistema di calcolo e produce grosse quantità di informazione si affida a sistemi di backup dei dati, tipicamente orientati a nastri magnetici. Il backup dei file può avvenire anche una volta al giorno se l'azienda lo ritiene necessario. I motivi che possono indurre al backup possono essere ricondotti a due tipologie di eventi: la prima di queste è senza dubbio dovuta ai disastri ambientali circostanti all'elaboratore (incendi, inondazioni, frane... tutte cose che per fortuna non capitano spesso) che, data la rarità degli eventi, non inducono seriamente l'utente a intraprendere una programmazione di backup dei file; l'altra tipologia di eventi è invece riconducibile alle disattenzioni dell'utente che magari cancella per sbaglio un file utile. Quest'ultimo problema è così frequente che in WINDOWS, quando un file viene rimosso esso non viene affatto cancellato, ma viene spostato in una directory particolare, il cestino, da cui è possibile riprendere il file e ripristinarlo nel path che occupava prima della cancellazione.
Se poi ogni volta occorre fare il backup dell'intero sistema allora il tempo da spendere per questa operazione è davvero tanto, meglio quindi non effettuare il backup dei file che sono rimasti immutati dall'ultima operazione di backup. Se non altro questo accorcia la durata dell'operazione di

backup. Backup di questo tipo si dicono anche dump incrementali. La strategia da seguire è la seguente: ogni settimana si esegue un dump completo dell'intero file system del sistema ed ogni giorno, invece, si esegue un dump incrementale dell'ultimo backup. Se da un lato ciò riduce di molto i tempi di backup (a patto che non molti file vengono modificati tra un giorno ed un altro) l'operazione di ripristino dei dati, qualora sia necessaria, deve seguire tutti i precedenti backup: a partire dal backup completo si aggiornano su di esso tutti i file modificati e contenuti nel backup incrementale che lo segue cronologicamente parlando.
La possibilità di compattare i file di backup è senza dubbio un interessante fonte di attrazione per l'utente che in questo modo potrebbe pensare di risparmiare (oppure far risparmiare alla sua azienda) diversi supporti per backup. Tale possibilità va comunque presa con molta prudenza poichè un eventuale errore sul supporto di backup, che ci può sempre stare, può compromettere la comprensione dell'intero file e non consentire la ricostruzione dei file di backup (l'algoritmo di compattazione usato per i file potrebbe non ricostruire ogni singolo file). In caso di errore sul supporto di backup, senza alcuno algoritmo per la compattazione dei dati, non sarà possibile ricostruire il solo file che sfortunatamente è capitato proprio su quel pezzo di nastro, ad ogni modo meglio quest'ultima ipotesi che la precedente.
Un altra area in cui l'affidabilità di un file system si confronta con altri file system è la coerenza del file system. Molti file system leggono blocchi, li modificano e in un secondo momento li scrivono; se il sistema ha un crash prima che tutti i blocchi modificati siano stati scritti, il file system può trovarsi in uno stato inconsistente. Questo è un problema critico specialmente se alcuni dei blocchi che non sono stati scritti sono blocchi di i-node (avete mai provate a spegnere un sistema UNIX prima che il sistema operativo venga caricato?), blocchi directory o blocchi contenenti la lista dei blocchi liberi! La maggior parte dei sistemi operativi si affida a servizi noti come scandisk (nei sistemi WINDOWS) e fsck (nei sistemi UNIX). Questi programmi vengono attivati dopo un crash del sistema e controllano la coerenza del file system. Per effettuare il controllo della coeranza sui file del file system il sistema operativo si costruisce due liste: nella prima lista verranno segnate le presenze dei blocchi in uso, nella seconda lista i blocchi liberi. Un file consistente avrà un bit di presenza per i blocchi in una o nell'altra lista. Oltre caso, banale, di consistenza appena descritto sono possibili altre tre combinazioni:

- blocco mancante: il blocco del file analizzato non compare nella lista dei blocchi del file (non vi appartiene) e nella lista dei blocchi liberi. Esso occupa solo spazio e per questo motivo il blocco viene aggiunto alla lista dei blocchi liberi;
- bloccoduplicato nella lista libera: nella lista dei blocchi liberi il contatore delle presenze per un blocco ha più di una presenza. La lista dei blocchi liberi viene ricostruita aggiornando il precedente valore ad 1;
- blocco dati duplicato: di tutti i possibili casi è quello più serio. La cosa peggiore che può succedere è che lo stesso blocco dati sia presente in due o più file. Se uno di questi due file fosse rimosso il blocco doppiamente presente sarebbe messo nella lista libera, portando a una situazione in cui lo stesso blocco è contemporaneamente usato e libero, chiaramente inconsistente. Se entrambi i file sono rimossi il blocco sarà messo nella lista libera due volte. L'azione di recuper che il controllore del file system deve eseguire è quella di allocare un blocco libero, copiarvi il contenutodel blocco e inserire la copia in uno dei file; in questo modo non si modifica il contenuto dell'informazione dei file sebbene uno dei due file è inconsistente. L'errore dovrebbe essere segnalato all'utente per permetterne la scelta.

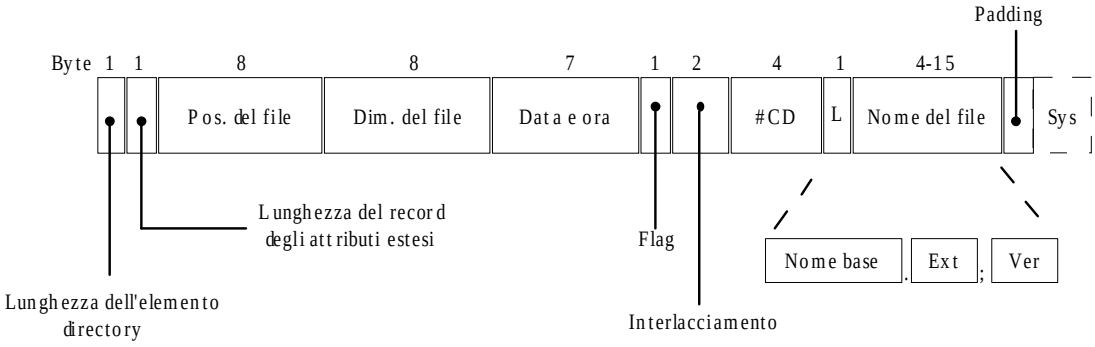
Il file system ISO9660

Come abbiamo precedentemente anticipato lo standard per cd-rom in termini di file system è il file system ISO 9660 che si basa sull'allocazione contigua dei blocchi di file e di directory. Ogni cd-rom che è presente sul mercato osserva il suddetto standard che garantisce quindi la compatibilità e l'interleggibilità per ogni sistema. La differenza sostanziale da un cd-rom con un disco magnetico sta nell'organizzazione interna. Siccome il disco magnetico si affida all'assegnazione dei blocchi per file che non necessariamente deve essere contigua la sua organizzazione interna è basata su una struttura a cilindri concentrici suddivisi in settori che a loro volta sono suddivisi in tracce di K blocchi. I cd-rom, invece, hanno una spirale contigua di blocchi da 2352 byte, il carico informativo effettivo è comunque di 2048 byte (2 KB). Alcuni byte del blocco, infatti, sono destinati a strutture dati informative. La dimensione tipica dei cd-rom in commercio è di 700 MB. In 700 MB è possibile contare $700 \cdot 2^{20}/2352=312077$ blocchi di dati. Tali blocchi possono descrivere un contenuto interamente digitale se usati in ambito informatico e quindi come supporti dati per elaboratori. Oppure, possono descrivere un contenuto interamente analogico del supporto che in questo modo sarà utilizzato per contenuti di tipo audio. Occorrono 75 blocchi per riprodurre 1 solo secondo di audio, pertanto, avendo a disposizione 312077 blocchi è possibile riprodurre da cd-rom 312077/75=4161 secondi che corrispondo a circa 70 minuti di audio (provate a ripetere i calcoli con supporti cd-rom da 800 MB, scoprirete che in tal caso si potranno riprodurre 80 minuti di audio).



Il file system ISO 9660 supporta la gestione di 2^{16}-1 cd in un insieme di file system spezzato su più cd-rom. Ognuno di essi può poi essere partizionato in più volumi logici di dimensioni più piccole. I primi 16 blocchi della spirale di blocchi contigui non sono usati dallo standard che li lascia quindi liberi per i produttori che in questo modo hanno, ad esempio, la possibilità di impegnarli collocando qui un eseguibile per mandare in esecuzione automatica un programma da cd-rom quando quest'ultimo è inserito nel lettore. Sono poi possibili altri usi dei suddetti primi 16 blocchi.

Dopo i primi 16 blocchi inutilizzati troviamo un descrittore primario con informazioni di carattere generale sul cd-rom, in esso si trovano svariati campi dati informativi come: l'identificatore di sistema (da 32 byte); l'identificatore di volume (da 32 byte); l'identificatore del distributore (da 128 byte) e l'identificatore del preparatore dei dati (da 128 byte); Sempre nel descrittore primario si raccolgono altre importanti informazioni come la dimensione dei blocchi logici (di solito 2048 byte ma anche le successive potenze di 2 sono ammesse, 4096 e 8192 byte ad esempio); data di creazione e data di scadenza del cd-rom ed un elemento di directory principale o directory radice che indicherà in quale blocco del cd-rom essa sia stata collocata. Dalla suddetta locazione è poi possibile localizzare il resto del file system. Il produttore e/o chi prepara i dati riempe questi dati con stringhe di sole lettere maiuscole ed alcuni caratteri di tipo numerico. Un elemento di directory sicompone dei diversi campi dati:

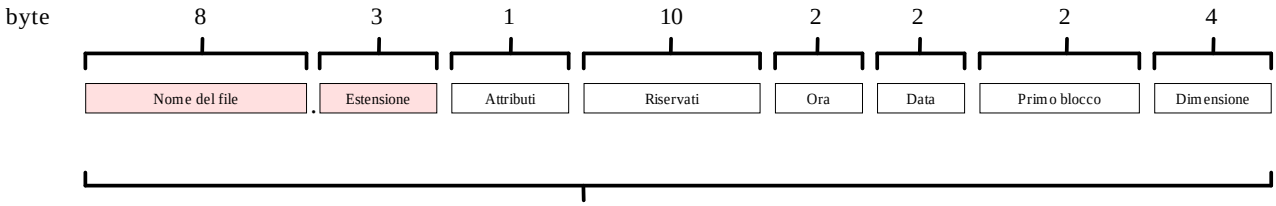


Ogni elemento di directory è di dimensione variabile (solitamente si compone di 10/12 campi dati), l'ultimo è marcato con un bit particolare. Gli stessi campi dati, per dare la possibilità a tutti i sistemi di leggerli, sono poi codificati sia in ASCII che in binario. Ad esempio, un numero di 16 bit userà 4 byte (2 byte in ASCII e altri 2 byte in binario).
Il primo campo dati indica la dimensione in byte dell'elemento directory, questo essendo di 1 byte (8 bit) ne limita la dimensione nell'intervallo da 0 a 255 byte. Lo stesso discorso si ripete per la descrizione della lunghezza dei record per attributi estesi. Se i files non fanno uso degli attributi estesi questo campo dati non è necessario (per questo motivo dicevamo prima che il numero di campi dati che compone un elemento di directory è variabile). Il successivo campo dati nell'elemento di directory sta ad indicare la posizione del file sul cd-rom, in altre parole ne indica il blocco iniziale (con 8 byte (64 bit) è possibile indirizzare 2^{64}-1 blocchi di cd-rom. Anche se ci possono sembrare tanti ricordiamo che in un cd-rom esistono 312077 blocchi e che il file system può essere spezzatto su un numero più di un cd-rom).
Il campo dati relativo alla data e l'ora in cui il cd-rom è stato scritto è di 7 byte, il formato usato per la data è del tipo AAAA-MM-GG mentre per l'ora hh-mm-ss. Nel campo flag si celano diversi bit tra cui quello per indicare al sistema la possibilità di marcare un elemento come non visibile e, quindi, di nasconderlo quando vengono generate le liste di files.
Per l'interlacciamento dei pezzi di file che risultano spezzati su pù cd-rom sono usati 2 byte. Il campo dati successivo di 4 byte (32 bit) indicia il numero di cd-rom che quindi varia tra 2^{32}-1. Il campo dati L indica la lunghezza in byte (quindi compresa tra 0 e 2^{8}-1 byte) del successivo campo dati, quello cioè relativo al nome del file. Quest'ultimo a sua volta si suddivide in un campo dati per il nome, un campo dati per l'eventuale estenzione ed un campo dati per specificare la versione del file (in binario). Il campo dati padding serve, invece, allo stuffing dell'elemento di directory che deve sempre essere un numero pari di byte (ciò favorisce ad esempio un controllo di parità su tutti i byte letti). Infine, l'ultimo campo dati system use (non sempre presente assieme al padding) o più brevemente sys non è specificato dallo standard che lo assegna per usi di sistema. I sistemi UNIX ad esempio usano il suddetto campo per imlementare le cosiddette estensioni Rock Ridge, tali estensioni permettono di rappresentare su cd-rom il file system di UNIX. Le estensioni Rock Ridge prevedono, dunque, nel campo dati system use i seguenti campi: PX per attributi POSIX; PN per numerare i dispositivi (in UNIX ogni dispositivo è modellato con un file); SL per rappresentare i link simbolici che in UNIX usano una variabile di conteggio per stabilire se il file è usato oppure no; NM per i nomi alternativi che in UNIX sono di sicuro più lunghi e articolati su più estensioni; CL per indicare la posizione del figlio in termini di i-node; PL per indicare la posizione del genitore in termini di i-node;RF per specificare la rilocazione di una directory all'interno della gerarchia e aggirare così una forte limitazione al file system ISO 9660 che limita ad 8 directory il massimo livello di profondità nella gerarchia ad albero del file system.; TF per contenere i tre campi dati timestamp solitamente inclusi negli i-node di UNIX.
Anche la comunità di utenti che usava i prodotti Microsoft ha dopo poco avanzato le proprie estensioni per il file systema ISO 9660, le estensioni Joiliet. Tali estensioni permettono: l'utilizzo di nomi lunghi per i file e per le directory; l'utilizzo di un set di carattere Unicode per i nomi di file (utile alla rappresentazione dei nomi per quei paesi che non usano l'; una profondità di struttura delle directory superiore agli otto livelli previsti; l'utilizzo di directory con estensioni.

Il file system di MS-DOS

Il file system di MS-DOS è molto simile, per certi versi ne è l'estensione, al file system di CP/M. La prima versione di MS-DOS (MS-DOS Ver.1.0) era limitata ad una sola directory proprio come faceva CP/M. MS-DOS, inoltre, funziona solo con piattaforme INTEL e non supporta la multiprogrammazione. A partire da MS-DOS Ver.2.0 è stata aggiunta la possibilità di avere un file system gerarchico con una profondità di directory arbitraria. La lettura da un file, mediante apposita system call che genericamente indichiamo come `open`, permette di ottenere un handle (gestore) per il file da manipolare. Nella chiamata di sistema viene specificato il path (sia relativo oppure assoluto) che è analizzato in ogni sua componente fino ad accedere al file (se questo esiste).

Le directory in MS-DOS usano elementi di directory di 32 byte che possono tuttavia avere dimensioni variabili, in esso si trovano informazioni come: il nome del file, gli attributi del file, informazioni di tipo timestamp per le date, il blocco di inizio file e la dimensione espressa in byte del file. Nel campo attributi (8 bit) si trovano alcuni flag nuovi come quello relativo all'archiviazione (indica se il file è stato già archiviato con una copia di backup, anche se non è usato dallo stesso file system permette alle applicazioni utente di gestire questa possibilità), quello relativo alla proprietà di file nascosto (un file nascosto non compare nelle liste dei file quando queste vengono generate per mezzo dell'apposito comando `dir`), quello relativo alla proprietà di file di sistema (un file di sistema non può essere cancellato se l'utente prova a chiamare su di esso il comando `del`).



Elemento di directory di MS-DOS

Il nome di file prevede una lunghezza di otto caratteri più una estensione variabile da uno a tre caratteri. Per il campo dati relativo all'ora si utilizzano 2 byte (16 bit), è pertanto possibile rappresentare 2^{16}-1=65535 valori distinti contro gli 86400 secondi contenuti in 24 ore. Per questo motivo la precisione per il campo dati relativo ai secondi è stata ridotta a ± 2 secondi (ogni bit per il campo dati relativo ai secondi vale 2 secondi). I 2 byte (16 bit) del campo dati riservati alla rappresentazione dell'ora sono così suddivisi:

- 5 bit per i secondi permettono di rappresentare 2^5=32 elementi contro i possibili 60 secondi contenuti in un minuto solare. Quindi, si è scelto di usare 1 bit ogni 2 secondi;
- 6 bit per i minuti permettono di rappresentare 2^6=64 elementi contro i possibili 60 minuti contenuti in un ora solare;
- 5 bit per le ore permettono di rappresentare 2^5=32 elementi contro le possibili 24 ore contenute in un giorno solare;

Anche il campo relativo alla rappresentazione di una data (2 byte=16 bit) è stato suddiviso:

- 5 bit per il giorno permettono di rappresentare 2^5=32 elementi contro i possibili 31 giorni contenuti in un mese solare del calendario;
- 4 bit per il mese permettono di rappresentare 2^4=16 elementi contro i possibili 12 mesi contenuti in un anno solare;
- 7 bit per l'anno permettono di rappresentare 2^7=128 elementi, l'anno base è il 1980. Per questo motivo il massimo anno rappresentabile è il 1980+128=2108;

Per la dimensione in byte del file si usano 4 byte (32 bit), questo significa che un file può essere grande da 0 a 2^{32}=4GB. Tuttavia la dimensione massima è stata comunque limitata a 2 GB. Il file system MS-DOS usa la tabella FAT, per questo motivo nell'elemento di directory sono destinati 2

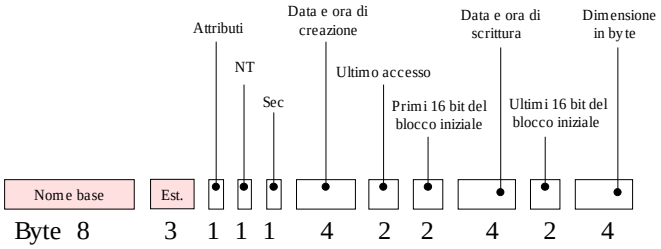
byte (16 bit) per esprimere la posizione del primo blocco. Questo significa che si possono specificare al massimo 2^{16}-1 blocchi (65535 blocchi) che nelle versioni FAT-12 (che usa 12 bit per l'indirizzamento su disco del blocco) usa blocchi che possono essere grandi almeno 512 byte. I blocchi sono qui chiamati cluster.
Per FAT-12 con blocchi da 512 byte è possibile gestire una partizione di $2^{12}\cdot 512$=2 MB (con 2^{12}=4096 voci nella tabella FAT, una per ogni blocco del disco), si tratta di un file system attualmente buono solo per floppy disk.
Con l'avvento degli hard disk il problema della dimensione della partizione limitata ad un certo valore è stato risolto aumentando la dimensione fisica dei blocchi sul disco che possono essere anche di 1, 2 e 4 KB. Così facendo FAT-12 riusciva a gestire al massimo una partizione da $2^{12}\cdot 4096$=16 MB. Gli hard disk più grandi di 16 MB erano partizionati in più unità logiche che MS-DOS nomina con le lettere dell'alafabeto maiuscole (D:\ E:\ F:\ etc...). MS-DOS era in grado di gestire 4 partizioni, pertanto l'hard disk più grande gestibile da MS-DOS era da 64 MB (suddiviso in 4 partizioni da 16 MB).
Per gli hard disk più grandi si pensò ad una nuova versione, FAT-16 (16 bit per l'indirizzamento ai blocchi sul disco). La FAT adesso aveva 2^{16}=65535 voci, una per ogni blocco del disco. Per FAT-16 le dimensioni dei blocchi potevano essere di 2, 4, 8, 16 e 32 KB. La massima partizione gestibile poteva essere grande $2^{16}\cdot 32K$=2 GB. Il massimo numero di partizioni contemporaneamente gestibili da MS-DOS era ancora 4, per questo motivo l'hard disk più grande che MS-DOS poteva gestire con FAT-16 era di 8 GB (suddiviso in 4 partzioni da 2 GB).
Con il sistema operativo WINDOWS 95 è stato fatto debuttare il nuovo file systema, FAT-32 (che in verità usa 28 bit per l'indirizzamento dei blocchi sul disco piuttosto che 32 bit come la sigla lascia intendere). In questo modo la massima partizione gestibile da MS-DOS potrebbe essere di $2^{28}\cdot 32K$=8TB (i blocchi possono ancora essere di 4, 8, 16 e 32 KB), essa è stata tuttavia limitata a 2 TB (2048 GB) in quanto internamente il sistema tiene traccia della dimensione della partizione in settori da 512 byte usando numeri da 32 bit, pertanto $512\cdot 2^{32}$ da al massimo 2 TB.

Dimensione blocco	FAT-12	FAT-16	FAT-32
512 byte	2 MB	/	/
1 KB	4 MB	/	/
2 KB	8 MB	128 MB	/
4 KB	16 MB	256 MB	1 TB
8 KB	/	512 MB	2 TB
16 KB	/	1024 MB	2 TB
32 KB	/	2048 MB	2 TB

Con FAT-2 è allora possibile decidere quale dimensione si preferisce per il blocco fisico. L'amministratore di sistema può in questo modo optare per blocchi più piccoli se i files sull'elaboratore hanno in media una piccola dimensione, in questo modo si limita lo spazio sprecato a causa della frammentazione dell'ultimo blocco allocato.

Il file system di Windows 98

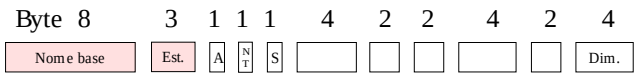
A partire da WINDOWS 98 è stato permesso all'utente l'uso di nomi di file più lunghi dei soliti 8 caratteri che MS-DOS metteva a disposizione. WINDOWS 98 utilizza un file system basato su FAT-32 che quindi permette l'uso dei moderni hard disk ormai sufficientemente capienti. La possibilità di dare ai files nomi lunghi è senza dubbio la caratteristica più importante del file system di WINDOWS 98. Microsoft poteva implementare una tale pensando a nuovi elementi di directory, più lunghi di quelli di MS-DOS e quindi anche più adeguati al suddetto problema. Nonostante ciò non è stata questa la strada percorsa da Microsoft che avendo ancora diversi utenti legati alle precedenti versioni di WINDOWS (come ad esempio WINDOWS Ver.3.1 e WINDOWS 95) ha cercato di mantenere la compatibilità all'indietro con i rispettivi file system e quindi con MS-DOS.
Se dunque la compatibilità all'indietro con le vecchie versioni doveva essere garantita di sicuro il nuovo file system dovrà avere una struttura comune ai vecchi elementi di directory di MS-DOS. La soluzione adottata da Microsoft sfrutta i 10 byte non usati nell'elemento di directory di MS-DOS, ciò favorisce l'aggiunta di altri campi dati.



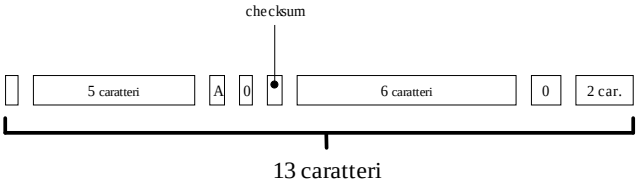
Il campo dati NT (1 byte) è stato aggiunto per favorire la compatibilità anche verso WINDOWS NT in termini di visualizzazione dei nomi di file con lettere maiuscole e minuscole. Il campo dati Sec aumenta la risoluzione della data che con 16 bit non poteva essere efficiente (Ricordate il problema dei secondi? In MS-DOS 1 bit valeva 2 secondi!). Nei 10 byte prima riservati e adesso utilizzati dal file system si trovano anche 4 byte per la data e l'ora di creazione del file; 2 byte per memorizzare l'ultimo accesso (Non l'ora! Solo la data di ultimo accesso!); 2 byte per memorizzare i primi 16 bit del blocco iniziale. Ed i nomi lunghi? Come vengono implementati?
La soluzione adottata da Microsoft consiste nell'associare ad ogni file due nomi, un nome è compatibile con le regole imposte dal vecchio file system, quello di MS-DOS, l'altro nome è invece leggibile per sistemi che adottano il sistema operativo WINDOWS 98. Quando l'utente crea un file assegnando per questo un nome, il sistema operativo verifica dapprima se il nome scelto dall'utente è compatibile con i nomi ammessi da MS-DOS. A tale proposito il sistema operativo si accerta che la lunghezza non superi gli 8 caratteri ammissibili e che il nome sia interamente scritto in maiuscolo. Un nome siffatto è sia compatibile con il vecchio MS-DOSche con WINDOWS 98. Tuttavia, se il nome pernsato dall'utente non risponde ai due requisiti appena citati il sistema operativo si attiva affinchè venga ricavato un secondo nome per il file (il primo nome, quello lungo, è leggibile da WINDOWS 98 ed è ammesso dal file system. A partire da questo si genera il secondo nome, quello per MS-DOS) leggibile anche da MS-DOS. L'algoritmo usato è il seguente: il sistema operativo prende i primi sei caratteri del nome lungo scritto dall'utente e li trasforma in caratteri maiuscoli; quindi aggiunge a questi sei caratteri i simboli ~1. Se il file così generato già esiste il sistema operativo fa seguire ai primi sei caratteri il simbolo ~2 (invece che ~1) e così via... I caratteri spazi inseriti nel nome sono cancellati nella versione MS-DOS del nome mentre alcuni caratteri speciali sono tradotti con il simbolo di underscore. Detto ciò, un file avente nome `appunti di sistemi operativi.txt`, per essere compatibile con i sistemi MS-DOS, è memorizzato nella sua forma estesa (secondo nome) e nella forma ridotta `APPUNT~1.txt` (primo nome). Il nome lungo è comunque memorizzato in più elementi di directory, ogni elemento può contenere 13 caratteri (gli elementi di directory contenenti i vari pezzi del nome sono memorizzati in ordine inverso e sono preceduti da un numero di sequenza. All'ultimo numero di sequenza si aggiunge 64). Gli elementi di directory che seguono il primo utilizzano il campo dati Sec (quello relativo ai secondi che è per loro ridondante) per effettuare una checksum (per un controllo di parità) su 8 bit.
L'esigenza di un campo dati per il controllo di parità è presto spiegato: è cosa ormai ben nota quella

di poter avviare un sistema in entrambe le modalità, quella cioè relativa al sistema operativo WINDOWS 98 e quella invece relativa alla modalità MS-DOS. Se, dunque, in modalità MS-DOS si decidesse di rimuovere un file verrebbe cancellato solamente il primo elemento di directory lasciando ancora sul disco quelli relativi al nome lungo del file (MS-DOS è infatti inconsapevole dell'esistenza degli altri elementi di directory). Gli stessi elementi di directory, poi, potrebbero essere assegnati ad un nuovo file appena creato e verrebbero ad assumere un valore inconsistente nei confronti del nuovo nome del file. Il campo dati per il controllo della parità permette al sistema operativo di scovare questi difetti e di correggerli scrivendo in essi i caratteri associati al vero nome del file (notare qui, che in tal caso il nuovo file creato sotto MS-DOS avrebbe al massimo una lunghezza di 8 caratteri, tale nome coinciderà anche con quello usato da WINDOWS 98).

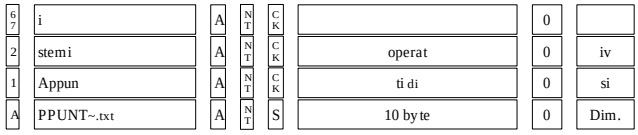
Elemento base di directory



Utilizzo di un elemento di directory per nomi lunghi



Elemento di directory per il file `Appunti di sistemi operativi.txt`



Il file system di Windows NT

NTFS è il file system nativo di WINDOWS NT e supporta, inoltre, i formati FAT e HPFS (HPFS è il file system di IBM per OS/2) per fornire delle possibili forme di migrazione dagli altri sistemi operativi (sebbene Microsoft non incluse il supporto per HPFS in NT Ver.4.0).

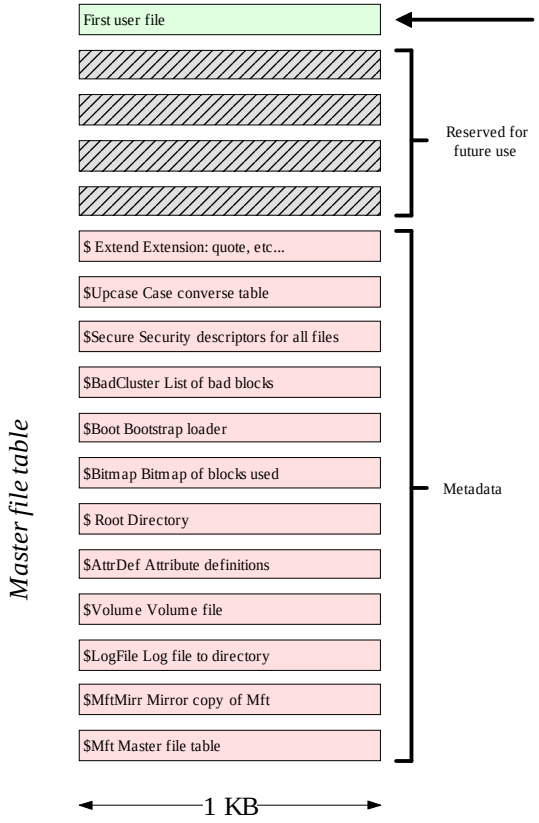
L'obiettivo per cui Microsoft decise di implementare NTFS era quello di superare le limitazioni degli altri due file system, compatibili con NT, e di fornire caratteristiche avanzate che un sistema operativo a livello aziendale richiede. Per esempio, NTFS supporta un sistema di sicurezza a livello di file e directory molto granulare, mentre FAT e HPFS non hanno caratteristiche di sicurezza. In aggiunta, lo schema di allocazione di NTFS può indirizzare efficacemente dischi rigidi di notevoli dimensioni. FAT e HPFS sono invece entrambi limitati dalla dimensione dei dischi. Infine, dei tre file system, NTFS è l'unico che supporta la codifica Unicode per gli ambiti internazionali ed ha caratteristiche per prevenire la corruzione di file e del file system in caso di guasto.

Alla fine degli anni '80 Microsoft progettò NTFS parallelamente allo sviluppo iniziale di Windows NT. Quando l'infrastruttura di base di NTFS fu realizzata e verificata la sua funzionalità, Microsoft diede come direttiva al team che stava sviluppando NT di usare NTFS come file system. Dato che NTFS doveva essere un nuovo file system, progettato da zero, il suo progetto poteva incorporare caratteristiche che potevano superare le limitazioni poste dall'hardware e dai file system dei PC attuali (allo sviluppo di NTFS) e anticipare le richieste degli utilizzatori aziendali del sistema. La cosa più ovvia fu quella di fornire un adeguato supporto ai dischi rigidi che aumentano costantemente le loro dimensioni. Tutti i file system WINDOWS dividono le partizioni dei dischi in unità logiche dette clusters. Il file system FAT usa 16 bit (nella versione classica) per indirizzare clusters, così può indirizzare al più 2^{16} o 65536 cluster diversi. I cluster possono variare in dimensione a seconda della dimensione del disco, ma cluster molto grandi possono portare come risultato a un problema di frammentazione interna oppure parecchio spazio sprecato all'interno del cluster stesso. Per esempio, se un file ha solamente 250 Bytes di dati, esso richiede un intero cluster di spazio allocato su disco il che risulta che più di 15 KB di spazio vanno sprecati in caso di cluster di 16 Kb. Con solo 65536 cluster indirizzabili, un disco FAT con 1KB di spazio per cluster potrebbe indirizzare al massimo un disco di 64 MB. Un disco di 4 GB, per esempio, richiederebbe quindi una dimensione di 64 KB per cluster con i relativi svantaggi che abbiamo visto prima per cluster di così grandi dimensioni. Un disco NTFS invece indirizza i cluster con un indirizzamento a 64 bit. Così anche con 512 Bytes per cluster NTFS non dovrebbe aver difficoltà ad indirizzare dischi con dimensioni che probabilmente non vedremo ancora per anni.

Gli sviluppatori di FAT e HPFS non considerarono il fatto della sicurezza all'interno del file system mentre NTFS usa lo stesso modello di sicurezza di WINDOWS NT. Discretionary access control lists (DACLs) e system access control lists (SACLs), controllano chi può fare operazioni sui file e quando un evento deve essere loggato, le operazioni sono registrate nel formato nativo di WINDOWS NT all'interno del file system NTFS. Il file system FAT usa i caratteri ASCII a 8 bit per nominare i file e le directory. Impiegando quindi set di caratteri ASCII mettiamo una limitazione ai nomi usabili con FAT che equivalgono a quelli inglesi (in generale simbolici). WINDOWS NT ed il file system NTFS usano entrambi il set di caratteri a 16 bit Unicode per i nomi. Questa caratteristica di NTFS permette agli utilizzatori di WINDOWS NT sparsi per il mondo di organizzare i loro file usando la loro lingua madre. Infine FAT non prevede nulla per la salvaguardia dei file e del file system in caso di guasti. Se un sistema va in crash quando stiamo creando, aggiornando file e/o directory la struttura FAT sul disco può diventare inconsistente (si puossono verificare uno dei precedenti casi trattati). La situazione può consistere in una perdita delle informazioni modificate oppure in una totale corruzione del disco e conseguente perdita di parecchie informazioni residenti sul disco. Questo rischio è inaccettabile per il mercato a cui WINDOWS NT è rivolto. Il file system NTFS ha quindi integrato un sistema di logging di transazioni in modo tale che quando una modifica deve essere implementata, NTFS si fa una nota della modifica da fare in un file speciale di log. Se il sistema va in crash, NTFS può esaminare il file di log e usarlo per ripristinare ad uno stato consistente il sistema con il minimo possibile di dati persi.

I vari tool per la formattazione (inizializzazione) dei dischi in NTFS fanno una stima automatica della dimensione delle unità d'allocazione in funzione della dimensione del disco. Queste stime

possono anche essere modificate manualmente dall'amministratore del sistema. Le stime fatte direttamente prendono in considerazione il discorso dello spazio sprecato, frammentazione interna e quindi prestazioni generali e cercano di ottimizzarle tutte con un compromesso.
Le informazioni associate alla gestione del disco sono registrate all'interno del disco come file speciali. I dati registrati all'interno di questi file e tutte le informazioni inerenti all'NTFS all'interno dei file utente e delle directory vengono detti metadata (alcuni file di tipo metadata iniziano con il simbolo $). Quando si inizializza un disco con file system NTFS, esso inserisce al suo interno 11 metadata files. Questi file sono generalmente invisibili quando esploriamo un volume NTFS con i tool classici come ad esempio Explorer oppure Firefox.
In aggiunta al sistema di log per evitare perdite di dati sui volumi NTFS, il file system NTFS protegge i suoi dati su disco con un sistema di firme. Quando in una lettura capita un errore, NTFS identifica il cluster come danneggiato e quindi procede alla rilocazione dei dati presenti su quel cluster e poi all'aggiornamento del metadata file $BADCLUS in modo tale da evitare di riutilizzare lo stesso cluster in futuro. Il metadata file $BITMAP, invece, è un grande array di bit in cui ogni bit corrisponde a un cluster sul disco. Se il bit è off allora il cluster risulta libero altrimenti, è in uso. Questo file è mantenuto per tener traccia dei cluster liberi su disco per l'allocazione di nuovo spazio. Il cuore del file system NTFS è la MFT (master file table). Essa è analoga alla file allocation table nel file system FAT perché MFT mappa tutti i file e le directory sul disco, inclusi i metadata files dell'NTFS stesso. La MFT è divisa in unità discrete chiamate records. In uno o più record, NTFS registra i metadati che descrivono un file o le caratteristiche di una direcory (informazioni sulla sicurezza e altri attributi come file a sola lettura oppure file nascosto) e la loro locazione sul disco. Sorprendentemente la stessa MFT è un file che NTFS mappa usando dei record all'interno della MFT stessa. Questa struttura lascia la possibilità alla MFT di espandersi oppure di restringersi.
I file e le directory sono identificati all'interno della MFT usando i relativi record che descrivono quindi l'inizio dei loro metadati all'interno della MFT stessa. I record sono solitamente di 1KB (come ad esempioavviene per WINDOWS NT 4.0) ma possono essere anche più grandi. Ecco un esempio della MFT:



Il file $MFTMIRR è un file di complemento, in caso di disastri al file system, per la prevenzione di perdita di dati. Esso contiene la copia dei primi 16 record della MFT. NTFS lo registra a metà del

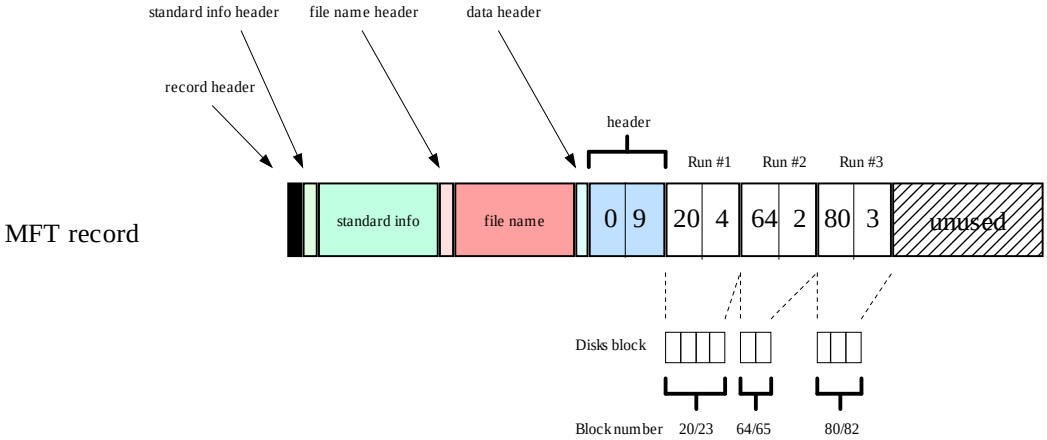
disco circa mentre la MFT è all'inizio dello stesso. Se il file system NTFS ha un problema nella lettura della MFT allora esso si riferisce ad un suo duplicato. La locazione della MFT e della sua copia sono registrate nel boot record del disco (un file da 512 bytes posto all'inizio del disco stesso).
L'accesso ai dati della MFT incide sulle performance di un disco con NTFS, così NTFS cerca delle soluzioni per accedere alla MFT in modo più rapido possibile. Dato che la MFT è un file residente su volume NTFS esso può ingrandirsi e rimpicciolirsi ed anche frammentarsi. Questa frammentazione si verifica perché NTFS non può allocare in anticipo lo spazio che la MFT occuperà. Quando la MFT cresce e qualche altro file sta occupando anche lo spazio che è a ridosso della parte finale della MFT allora NTFS inizia a guardare altrove sul disco per avere dello spazio libero.
L'accesso più veloce si realizza quando vengono fatte delle operazioni sul disco in maniera sequenziale ma in una MFT frammentata ciò non può avvenire ed NTFS ha allora bisogno di più letture per accedere. Questo può portare ad un abbassamento delle performance. Per evitare quanto appena detto il file system NTFS crea una regione di cluster in introno della fine della MFT dove file e directory non possono essere memorizzati. In questo modo si dà la possibilità alla MFT di crescere e/o ridursi senza troppe difficoltà. Infatti, quando lo spazio libero sul disco comincia a scarseggiare, NTFS rilascia un pò dello spazio precedentemente detto. Questo però comporterà per la MFT un rischio di frammentazione in un disco che peraltro è già al limite delle sue capacità (bisogna poi che NTFS non lascia deframmentare a tool esterni la MFT).
La MFT si compone di svariati record, questi contengono alcune informazioni di base che riguardano il record stesso ed espresse sottoforma di header più altri attributi associati invece al file ed alle directory. L'header nel record comprende ad esempio: un numero di sequenza (usati da NTFS per effettuare il controllo di integrità su alcuni campi dati), un puntatore al primo attributo all'interno del record, un puntatore al primo byte libero all'interno del record, il numero di record che appartengono al file etc... NTFS usa gli attributi per immagazzinare tutte le informazioni sui file e le directory. Sul disco gli attributi sono divisi in due componenti logiche: un header ed una parte dati. Nell'header è specificato il tipo di attributo, il nome del file, eventuali flag nonchè la locazione del disco in cui recuperare i pezzi di informazione. Per ottimizzare le prestazioni NTFS registra la parte dati dell'attributo all'interno del record della MFT (quando ciò è possibile) anzichè leggerlo di volta in volta dal cluster. Quando un attributo ha la sua parte dati memoriazzta nella MFT si dice che l'attributo è residente (in caso contrario non residente). Se la parte dati di un attributo è residente nella MFT l'header dell'attributo punta alla posizione dei dati all'interno del record della MFT. Per quanto riguarda gli attributi, invece, NTFS ne prevede 14 tipi diversi.

NTFS Attribute types

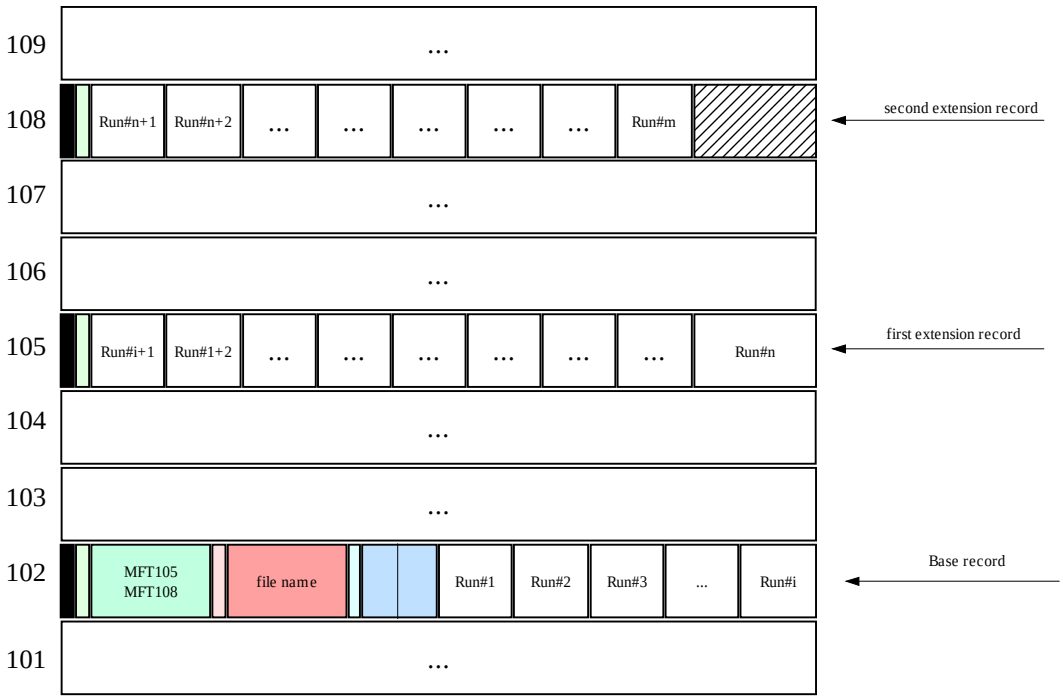
Attribute type	Description
$VOLUME_VERSION	Volume version
$VOLUME_NAME	Disk's volume name
$VOLUME_INFORMATION	NTFS version and dirty flag
$FILE_NAME	File or directory name
$STANDARD_INFORMATION	File timestamps and hidden, system and read-only flags
$SECURITY_DESCRIPTOR	Security information
$DATA	File data
$INDEX_ROOT	Directory content
$INDEX_ALLOCATION	Directory content
$BITMAP	Directory content mapping
$ATTRIBUTE_LIST	Describes non resident attribute headers
$SYMBOLIC_LINK	Unused
$EA_INFORMATION	OS/2 compatibility extended attributes
$EA	OS/2 compatibility extended attributes

Il nome del file, l'attributo per la specifica di attributo residente nella MFT e gli attributi per la sicurezza sono sempre residenti nella record. Le informazioni del record che individuano i pezzi contigui di file si dicono run information (descrivono una sequenza di cluster) e come altre informazioni NTFS hanno dapprima un header che li specifica (l'header indica quale cluster della

parte dati dell'attributo è mappata nella run information). Ogni run information indica un blocco iniziale e la durata in blocchi della run information.

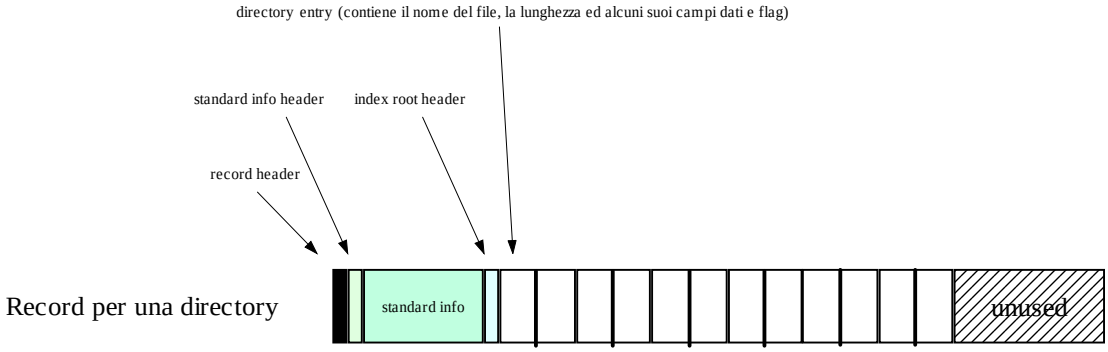


Nella figura è rappresentato un record MFT di un file avente 3 sequenze di run e 9 blochi. Se un file ha troppi attributi che non sono possono essere registrati in un singolo record della MFT, NTFS allora record addizionali e registra la lista degli attributi nel record base. La lista degli attributi punta alla locazione degli attributi nei record addizionali e consiste di un valore per ogni attributo.



Una directory per NTFS è un attributo indice. NTFS usa gli attributi indice per collocare i nomi dei file. Una directory contiene il nome del file ed una copia delle informazioni dell'attributo standard del file (informazioni sulle date e ore). Questo approccio fornisce una spinta sul piano delle prestazioni quando si naviga all'interno delle directory in quanto NTFS non deve accedere alla MFT per trovare le informazioni da visualizzare per ogni file all'interno della directory.
Quando queste informazioni possono essere contenute interamente in un record della MFT, un tipo di attributo, l'index root, descrive la locazione dei valori nel record. Quando una directory cresce, le informazioni necessarie a descriverla possono superare il record della MFT assegnato. In questo caso, NTFS alloca un index buffers per memorizzare informazioni addizionali. L'header dell'attributo indice specifica la locazione di un buffer. Nell'NTFS di WINDOWS NT Ver.4.0 la dimensione di questo buffer era di 4 KB e le informazioni sulla directory all'interno del buffer

erano di lunghezza variabile visto che conteneva nomi di file. Per rendere più efficienti possibili le operazioni sulle directory, NTFS preordina le directory della radice e dei buffer.



Dato che file e directory (inclusi i file di metadati NTFS) cambiano, NTFS scrive delle informazioni nei file log del volume. Il programma `chkdsk` usa questi file di log per rendere le strutture dati NTFS consistenti e per minimizzare la perdita di dati in caso di crash. I file di log possono essere di due tipi: redo ed undo. Redo memorizza le informazioni riguardanti modifiche che devono essere rifatte in caso di guasto al sistema e i dati modificati non sono sul disco. NTFS usa poi una operazione undo (leggendo cosa ripristinare dall'altro file log, quello undo) per fare un rollback delle modifiche fatte che non erano state completate quando il sistema è andato in crash. Se NTFS sta aggiungendo dati ad un file ed il sistema crash tra il tempo in cui NTFS estendeva la lunghezza del file e il tempo in cui scriveva i nuovi dati, il file di log undo dice ad NTFS di accorciare la lunghezza del file alla sua dimensione originaria durante il recupero.

Il file system di UNIX Ver.7

La versione del file system UNIX che tratteremo è la Ver.7 detta anche fast file system ed è la stessa di quella usata precedentemente per spiegare il file system di UNIX basato su strutture dati i-node. FFS limita la lunghezza dei nomi per file ad un massimo di 14 caratteri e dedica un elemento di directory per ogni file o directory occupando 14 byte per il nome e 2 byte per il numero di i-node. La directory è quindi un array regolare di i-node e nomi di file. Con 2 byte nell'elemento directory il massimo numero di file indirizzabile in una directory è di 2^{16}=65535 (64K). Il file system è a forma di albero (capovolto) che con l'aggiunta dei link può essere un grafo aciclico.

La struttura dati più importante di FFS è l'i-node. Esso contiene un certo numero di attributi come la dimensione del file, i tre campi per le date di creazione, ultimo accesso e ultima modifica, il proprietario del file, i relativi permessi ed il contatore per segnare i link che puntano alla directory oppure al file.La struttura dell'i-node è quella già vista, una lista dei primi 10 indirizzi di disco (soluzione buona per i file piccoli) con l'aggiunta di alcuni indirizzi di i-node che puntano ad altri blocchi del file (indirizzamento singolo indiretto, indirizzamento doppio diretto ed indirizzamento triplo indiretto). Quando un file viene aperto il file system deve prendere il nome di file fornito e localozzare i suoi blocchi di disco, ad esempio: