

Appunti di sistemi operativi

Cos'è un sistema operativo?

Per sistema operativo si intende quell'insieme di procedure e programmi che permettono di gestire in modo corretto, efficiente e totalmente trasparente al programmatore l'hardware nudo e crudo. Una delle funzioni principali che deve svolgere il sistema operativo è quindi quella di fornire una serie di procedure che permettano di accedere in modo semplice all'hardware dando al programmatore un'idea astratta delle periferiche su cui lavora. Per farsi un'idea di cosa si intende si consideri un disco fisso, un supporto di massa su cui è possibile scrivere file di dati all'interno di un'organizzazione ad albero formata da cartelle e file (detta file-system) in cui si ha una cartella radice contenente file e cartelle che a loro volta possono contenere file e cartelle. Questo in realtà è quello che noi vediamo dell'organizzazione di un disco fisso, ovvero l'interfaccia astratta che l'utente ha per la gestione delle informazioni sul supporto di massa, il programmatore inoltre ha un'ulteriore interfaccia astratta formata da una serie di possibili chiamate di sistema (ovvero appelli alle procedure del sistema operativo di cui si accennava prima) che permettono di creare, leggere e scrivere file; il programmatore sa solamente che esistono dei file organizzati in un file-system, che essi possono essere aperti in lettura, scrittura, append, ecc., che è possibile leggere, scrivere o muoversi all'interno del file e che infine è possibile chiuderlo. Tutto questo però fa riferimento ad un approccio astratto (e sicuramente più comodo) offerto dal SO per l'utilizzo del disco fisso. Se dovessimo analizzare cosa realmente avviene e magari controllare le reali operazioni relative all'hardware dovremmo spedire i dati di controllo della testina all'interfaccia (hardware s'intende) del disco indicando al disco di ruotare, posizionarci sulla traccia contenente la FAT e cercare l'indirizzo del primo blocco del file (ovvero il numero di traccia, settore e faccia) eccetera eccetera; logicamente se un programmatore dovesse prendere in considerazione sempre tutte queste problematiche (per non aggiungere il fatto che qualcosa potrebbe andare storto!!!) probabilmente perderebbe moltissimo tempo e rischierebbe anche di errare e rovinare l'hardware stesso! Risulta ora ovvia l'utilità del sistema operativo, in realtà però questa mansione non è l'unica ma risulta solamente una parte (anche molto piccola) delle problematiche che il sistema risolve: con lo sviluppo ed il diffondersi dei sistemi multiprogrammati e multiutente oggi la gestione della concorrenza fra processi è diventata infatti una delle mansioni principali che il sistema operativo svolge, esso infatti dovrà predisporre dei moduli particolari fatti apposta per gestire l'esecuzione e la concorrenza fra processi su di una macchina considerata monoprocesso (la presenza o meno di più processori non fa differenza perché viene gestita direttamente via hardware). Vedremo quindi ora alcune delle parti che caratterizzano il sistema operativo e che servono principalmente per gestire le risorse (per esempio dischi, memoria, processore, ecc.).

Lo scheduler come gestore della risorsa processore

Come detto in precedenza un sistema operativo deve permettere di gestire l'esecuzione di più processi concorrenti, ma cos'è un processo? Un processo altro non è che l'esecuzione di un programma di qualunque genere (sistema, utente, figlio di un altro processo, ecc.) e deve comprendere quindi tutto ciò che serve all'architettura per la sua esecuzione (per es. uno stack ed uno heap, un program counter, registri, ecc.; tutte queste informazioni sono memorizzate in un blocchetto detto PCB (Process Control Block) che contiene anche altre informazioni relative allo stato del processo, l'archiviazione di tali processi è eseguita tramite una chiave primaria numerica detta PID (Process Identifier)), un processo quindi deve essere gestito come un pacchetto di informazioni che permettono al sistema di poter eseguire una porzione di codice sul processore in modo assolutamente indipendente dagli altri (considerando per ora processi che non comunicano fra di loro). Quali sono quindi le problematiche relative all'esecuzione di più processi su di una macchina singola? Sicuramente la condivisione del processore, ovvero scegliere in ogni momento quale processo deve utilizzare la CPU. Come simulare l'esecuzione concorrente? Semplice, lasciando la CPU ad ogni processo per un breve intervallo di tempo. Lo scheduler esegue esattamente questa operazione, ovvero mette in esecuzione i processi assegnando loro la risorsa processore un po' per ciascuno. A questo punto è d'obbligo distinguere fra due metodologie di scheduling: quello preemptive e quello non preemptive. Ovvero quello che permette l'interruzione di esecuzione di un processo

per favorirne un altro (magari appena arrivato) e quello che invece non interrompe un processo in esecuzione per favorirne altri a meno che non sia scaduto il suo quanto di tempo (ovvero il tempo di CPU assegnato a quel processo dallo scheduler). In qualunque dei due casi comunque si ha il bisogno di poter assegnare ad ogni processo uno stato: abbiamo visto che i processi possono essere in esecuzione (running) oppure essere pronti per essere eseguiti (ready); in realtà esiste un altro stato in cui un processo può trovarsi: lo stato di attesa (waiting), magari tale processo non può essere eseguito perché è in attesa di operazioni di I/O oppure è in attesa che un altro processo termini delle operazioni particolari di cui lui stesso ha bisogno. Il primo problema che ora risolveremo è la creazione di un algoritmo di scheduling efficiente che permetta di tenere ottimali i seguenti parametri: utilizzo della CPU, ovvero il tempo durante il quale la CPU è assegnata ad un processo; throughput, ovvero il numero di processi eseguiti nell'unità di tempo; turnaround, ovvero il tempo totale trascorso dal momento in cui un processo viene messo in esecuzione assieme agli altri ed il momento in cui esso termina la sua esecuzione; waiting time, ovvero il tempo trascorso in attesa della CPU; response time, ovvero il tempo trascorso dal momento in cui il processo è stato messo inizialmente in coda d'attesa (ready) a quando viene messo in esecuzione la prima volta (running). Logicamente si cerca di rendere minimi questi tempi!

L'algoritmo **FCFS (First Come First Served)** è uno dei più semplici non che dei primi studiati per la risoluzione di questa problematica, consiste nel servire per primi i processi arrivati per primi. Quest'algoritmo non è certo dei migliori, crea infatti tempi di attesa anche molto lunghi anche per processi con un CPU burst (tempo di esecuzione effettivo) molto basso, basti pensare al caso in cui un processo di mole elevata arrivi poco prima di alcuni processi molto piccoli (un comando `ls` piuttosto che un comando `ps`), accadrebbe che i processi piccoli e rapidi dovrebbero attendere a lungo prima di essere eseguiti aumentando l'attesa media irrimediabilmente quando sarebbe stato più utile eseguire prima i processi rapidi e solo dopo lasciare la CPU a quello gravoso! Così si sono realizzati altri algoritmi più efficienti.

L'algoritmo **RR (Round-Robin)**, che è di tipo preemptive, risolve parzialmente il problema precedente, esso infatti prevede la definizione della dimensione di un quanto di tempo da assegnare a turno ai processi in attesa nella coda ready (dove vi sono tutti i processi nello stato ready), ogni processo viene eseguito per quel quanto di tempo (viene messo quindi in running) e poi viene bloccato e riaccodato. Logicamente il quanto di tempo va scelto in maniera sensata: un quanto di tempo troppo lungo farebbe degenerare il RR in un FCFS, un quanto di tempo troppo corto rischierebbe di far sprecare il processore per i context-switch (quando si deve far passare il processore dall'esecuzione di un processo ad un altro si devono salvare tutte le informazioni relative al processo precedente come i registri, il program counter e tutte le variabili, inoltre si deve mettere in esecuzione lo scheduler che dovrà accodare questo processo dopo averlo messo in ready e mettere in running il successivo, tutte queste operazioni assieme si chiamano context-switch e richiedono tempo per essere eseguite senza giovare a nessun processo, se è più gravoso il tempo per i context-switch che non quello di esecuzione dei processi si spreca la CPU).

Ancora come prima però può capitare che un processo veramente veloce arrivato dopo e capace di terminare la sua esecuzione in un solo quanto di tempo se non in una frazione venga messo in fondo alla coda di ready e quindi debba attendere un tempo esageratamente elevato. L'algoritmo **SJF (Shortest Job First)** risolve questo problema, esso chiede che processi con un CPU burst time inferiore vengano eseguiti per primi. Si può dimostrare matematicamente che questo algoritmo è quello ottimale, l'unico problema è che non è realizzabile perché non è possibile sapere a priori quale sia il CPU burst time di un processo anticipatamente, un metodo per approssimarne è quello di calcolare questo tempo in modo statistico.

Altra problematica da risolvere è quella relativa a particolari sistemi detti real-time, dove cioè si richiedono tempi di risposta molto brevi per certi processi privilegiati (si faccia riferimento per esempio ad un sistema di controllo di una centrale nucleare, esso potrebbe essere in time-sharing ed eseguire più processi in contemporanea, fra questi potrebbe esserci quello di controllo delle barre di uranio e quello di refresh dello schermo, logicamente non è pensabile che il processo di refresh dello schermo "rubì" la CPU a quello di controllo delle barre di uranio durante una disputa dello scheduler che deve decidere chi dei due mettere in running! Per gestire situazioni di questo tipo in generali si aggiunge un'ulteriore proprietà ai processi che è la priorità, un valore numerico che definisce chi deve essere messo in stato di running più frequentemente o comunque quale processo deve essere eseguito per primo. L'utilizzo di tale metodologia però alza il polverone su altri problemi non banali: un processo di priorità molto bassa, per esempio, potrebbe non ottenere mai la CPU! Questo fenomeno si chiama starvation; la soluzione a questo problema è l'aging, ovvero l'innalzamento periodico della priorità di quei pro-

cessi che non riescono ad andare in esecuzione in modo da assicurare a tutti che prima o poi vengano messi in running.

In Unix lo scheduler gestisce i processi utilizzando un sistema più complicato, chiamato **multilevel feedback queue scheduling**, che sfrutta tutti gli algoritmi precedentemente visti: vengono organizzate delle code di processi di pari priorità ordinate fra di loro in senso crescente (per priorità s'intende), per i processi ad alta priorità si utilizzano algoritmi di tipo FCFS, per i processi utente invece si utilizzano algoritmi di tipo RR con quanti di tempo differenti che dipendono dalla priorità. Ogni processo può essere spostato da una coda ad un'altra: se per esempio un processo non riuscisse ad essere eseguito potrebbe venir spostato in una coda a priorità più alta, se invece un processo dovesse essere molto oneroso o magari dovesse chiedere troppo spesso operazioni di I/O potrebbe venir spostato in una coda a priorità più bassa (dove i quanti di tempo per il RR sono più lunghi e si risparmiano context-switch).

La sezione critica e la sincronia

In molti casi si ha il bisogno di permettere a più processi di utilizzare delle **risorse condivise**, basti pensare per esempio ad un database distribuito: una struttura di questo tipo memorizza informazioni che devono essere accessibili a più utenti o a più processi per la visione o la loro modifica. Quando risulta indispensabile l'utilizzo di risorse condivise da parte di più processi si crea una problematica relativa alla sezione critica: ma cos'è la sezione critica? Essa è quella parte dei programmi relativi ai processi che condividono la risorsa in cui si fa uso della risorsa stessa e che potrebbe causare un problema di condivisione. Vediamo ora un esempio di cattivo funzionamento della concorrenza a causa della condivisione di una risorsa: consideriamo due processi concorrenti che debbano gestire delle operazioni bancarie su di un conto corrente, fingiamo che P1 sia il processo di accredito e che debba accreditare su di un conto, inizialmente del valore di £ 4.000.000, un milione, mentre il processo P2 deve addebitarne mezzo. A conti fatti il conto corrente alla fine dovrebbe valere £ 4.500.000, vediamo invece cosa potrebbe accadere: il processo P1 legge dalla risorsa condivisa `conto_corrente`, che ipotizziamo essere un file, il valore attuale (4.000.000), successivamente consideriamo il caso in cui P1 perda la CPU, magari perché il suo quanto di tempo è scaduto, e la risorsa processore venga passata al processo P2, esso legge il valore iniziale del conto corrente, esegue l'addebito e scrive sul file il risultato: 3.500.000. Ora il processo P2 termina la sua esecuzione e lo scheduler seleziona il processo P1 e lo mette in running, P1 accredita un milione sul conto corrente letto precedentemente (che era di 4.000.000) e scrive su file il nuovo valore, ovvero $4.000.000 + 1.000.000 = 5.000.000$!!! Il fortunato risparmiatore vede crescere il proprio conto corrente a discapito della banca a causa di un errore di gestione della risorsa condivisa. Perché quest'errore? Perché entrambe i processi erano contemporaneamente all'interno di una porzione di codice che sfruttava la risorsa `conto_corrente`! Tale risorsa (in questo caso un file, ma è facile capire che la stessa cosa potrebbe accadere con molte altre risorse come per esempio una porzione di memoria o una semplice variabile) è condivisa e quindi quando se ne fa utilizzo si entra in sezione critica (attenzione, non solo quando la si legge e quando la si scrive ma anche quando la si elabora accreditando o addebitando), ora è facile vedere anche quali sono le proprietà della sezione critica, ovvero quali sono le condizioni particolari che vorremmo imporre su questa porzione di codice:

1. **Mutua esclusione:** non devono mai esserci più processi all'interno della sezione critica relativa ad una stessa risorsa in contemporanea
2. **No waiting indefinito:** nessun processo può impedire ad un altro processo per un tempo indefinito di entrare nella propria sezione critica
3. **No starvation:** un processo che vuole entrare in sezione critica prima o poi dovrà entrarci

Dovremo elaborare quindi un algoritmo capace di rendere vere queste tre proprietà fondamentali per le sezioni critiche senza, ovviamente, fare alcuna ipotesi sul numero di CPU.

Una prima soluzione è quella della **stretta alternanza**: si utilizza una variabile *turn* che permette di decidere a chi dei processi deve entrare in sezione critica, ogni processo entra in sezione critica solo se non ce ne sono altri controllando questa variabile e quindi alternandosi. Il codice C che segue dà l'idea del funzionamento di questa soluzione:

```
void P1(){
    while (turn!=1);
    sezCrit();
    turn=2}
```

```
void P2(){
    while (turn!=2);
    sezCrit();
    turn=1}
```

Questa soluzione garantisce sicuramente la prima proprietà ma non le altre due: può capitare che un processo non entri in sezione critica (si ipotizzi la presenza di un costrutto if) impedendo anche all'altro di entrare (perché non è stata settata la variabile turn, per esempio). Inoltre si fa utilizzo di un ciclo vuoto (busy-waiting) per emulare l'attesa sprecando però tempo di CPU. Una soluzione che ovvi al primo dei due problemi (al secondo ovvieremo solo alla fine) è quella di utilizzare delle "bandierine", ovvero ogni processo prima di entrare in sezione critica setta la propria flag (bandierina in inglese) per indicare la propria volontà di entrarvi e controlla che l'altro processo non voglia entrare in sezione critica facendo un test sulla sua flag, solo successivamente entra in sezione critica. Questa soluzione però presenta una problematica molto particolare: potrebbe capitare che entrambe i processi settino la propria flag e rimangano in attesa entrambe che l'altro entri in sezione critica, esca e resettì la propria flag. Questa attesa infinita che blocca entrambe i processi prende il nome di deadlock e deriva dal fatto che entrambe i processi sono "scrupolosi" nell'ingresso in sezione critica, ovvero si incaponiscono entrambe nel voler attendere che prima sia l'altro a terminare le operazioni di sezione critica. Di seguito è riportato l'algoritmo appena presentato:

```
void P1(){
    falg[0]=true;
    while (flag[1]);
    sezCrit();
    flag[0]=false;}
```

```
void P2(){
    falg[1]=true;
    while (flag[0]);
    sezCrit();
    flag[1]=false;}
```

In questo caso la presenza delle prime due proprietà c'è, manca la terza! Un algoritmo che riesce a garantire tutte le proprietà e che prende il nome dall'ideatore, che lo ha creato dopo non pochi sforzi ed anni di studio, è l'algoritmo di Peterson: esso sfrutta entrambe i metodi dei precedenti facendo in modo che ogni processo indichi tramite le variabili *turn* e *flag* un qualcosa del tipo "io voglio entrare in sezione critica, se però vuoi entrarvi anche tu: prego!"!!! Questo algoritmo, riassunto nelle seguenti righe di codice, funziona e garantisce tutte e tre le proprietà, ha però un paio di problemi: risulta estremamente complesso, se non irrealizzabile, per più di due processi e sfrutta ancora i cicli di busy-waiting che sprecano la risorsa processore!

```
void P1(){
    flag[0]=true;
    turn=2;
    while (flag[1] && turn==2);
    sezCrit();
    flag[0]=false;}
```

```
void P2(){
    flag[1]=true;
    turn=1;
    while (flag[0] && turn==1);
    sezCrit();
    flag[1]=false;}
```

Per risolvere il problema in maniera più flessibile ed efficiente è stata introdotta un'istruzione particolare a livello di linguaggio macchina (e quindi direttamente via hardware) che prende il nome di **TSL (Test and Set Lock)**: essa carica in un registro il valore contenuto in una locazione di memoria (cioè una variabile) ed imposta ad un numero diverso da 0 il contenuto di quella locazione. Nonostante siano due operazioni distinte, specialmente in architetture di tipo RISC, si tratta pur sempre di una singola istruzione macchina ed è quindi indivisibile, per fare in modo che sia così anche in sistemi multiprocessore la CPU si impossessa del bus dati impedendo a qualunque altro processore di lavorare sulla memoria. A questo punto, grazie a quest'istruzione, possiamo realizzare le due parti di codice che indicano l'ingresso e l'uscita in sezione critica, quella per l'ingresso dovrà rendere impossibile l'ingresso a qualunque altro processo mentre quella di uscita dovrà rendere nuovamente possibile l'ingresso all'interno del blocco di codice. Di seguito sono riportate (in assembly) queste due procedure che prendono il nome di *enter_crt_sec* e di *leave_crt_sec* che saranno da richiamare una prima e l'altra dopo la sezione critica:

enter_crt_sec:	TSL	reg, lock
	BNEQZ	reg, enter_crt_sec
	RET	
leave_crt_sec:	MOVE	reg, 0
	RET	

Qual è il funzionamento di queste procedure? Lock, che è la variabile che indica se si può entrare in sezione critica o meno, viene copiato in reg da TSL che successivamente lo setta a 1, ora reg contiene, se inizialmente non c'erano altri processi in sezione critica, il valore iniziale di lock che doveva essere 0, in questo modo il salto successivo viene ignorato e la procedura termina (con l'istruzione RET di ritorno da procedura). Se invece dovesse esserci un processo in sezione critica la TSL copierebbe 1 in reg bloccando il processo in un ciclo di busy-waiting all'interno della *enter_crt_sec*. Sarà possibile sbloccare l'attesa solamente con l'esecuzione di una *leave_crt_sec* che imposterà a 0 reg indicando l'uscita da sezione critica del processo che vi era entrato precedente. Una problematica che si incontra tramite l'utilizzo di questo tipo di soluzione è lo spreco di CPU all'interno dei cicli di busy-waiting, per risolvere questo tipo di problematica si sfrutta allora lo scheduler: il controllo dell'esecuzione dei processi è dello scheduler, sarà quindi esso a darci la possibilità di controllare l'esecuzione dei processi, semplicemente si creeranno due operazioni atomiche (l'atomicità può essere garantita tramite l'utilizzo della TSL e tramite la disabilitazione momentanea delle interruzioni sulla CPU) che permettono di mettere in stato di waiting un processo e che permettono di risvegliarlo. Queste operazioni vengono di solito chiamate **SLEEP** e **WAKEUP**. Esse riescono quindi ad "addormentare" processi che attendono di poter entrare in sezione critica e di svegliarli successivamente (dovrà pensarci chi è in sezione critica ad eseguire appena esce una WAKEUP per svegliare chi è in attesa). Questo tipo di soluzione è del tutto accettabile per la problematica della sezione critica, risulta però scomoda per alcune sincronizzazioni fra processi che risultano complicate se non addirittura impossibili! Si faccia riferimento per esempio alla problematica del produttore e del consumatore: un processo produce dati e li mette su di un buffer di dimensioni fisse condiviso con il processo consumatore che preleva i dati dal buffer e li utilizza, nel caso in cui per esempio il buffer dovesse essere vuoto il consumatore si metterà in attesa, tramite una SLEEP, che il produttore gli indichi che il buffer non è più vuoto, per farlo però dovrà eseguire prima il test su di una variabile che gli indica quanti elementi sono contenuti all'interno del buffer, appena fatto il controllo il consumatore potrebbe però venire interrotto dallo scheduler che potrebbe assegnare la CPU al produttore che, prodotti i dati, potrebbe eseguire una WAKEUP per risvegliare il processo consumatore che non era ancora entrato in stato di waiting, ora è chiaro che il processo consumatore entrerà in stato di waiting con la sua SLEEP e non vi uscirà mai perché la WAKEUP è arrivata troppo presto, inoltre quando il buffer sarà pieno anche il processo produttore dovrà fermarsi con lo stesso meccanismo per attendere che il consumatore lo svuoti, così si arriverà in uno stato analogo al deadlock in cui i processi si trovano in stallo. Per risolvere anche i problemi di sincronia si sono create delle particolari variabili che prendono il nome di **semafori**: un semaforo altro non è che una variabile intera su cui si possono eseguire delle operazioni molto simili alla SLEEP ed alla WAKEUP conteggiando però le WAKEUP nel caso avvengano a vuoto come nel caso precedente, quando quindi un processo vorrà utilizzare una SLEEP non entrerà in waiting a meno che il semaforo sia a 0, in caso contrario semplicemente verrà decrementato il semaforo (si utilizza una delle WAKEUP precedentemente ricevute). Le operazioni equivalenti, ma che si eseguono specificamente sui semafori, alla SLEEP ed alla WAKEUP sono: l'operazione P(sem) che testa il semaforo per vedere se vale 0, nel caso in cui entra in waiting, e se non vale 0 lo decrementa; l'operazione V(sem) che testa il contenuto della struttura, di solito una coda, che contiene l'elenco di processi in waiting su quel semaforo e, se questa non è vuota, ne sposta uno in stato ready, altrimenti incrementa il valore del semaforo (indicando la presenza di un'istruzione V andata a vuoto che potrà però svegliare comunque il primo processo che si presenta tramite una P semplicemente impedendogli di passare in waiting). Tramite l'utilizzo dei semafori è possibile gestire in modo corretto sia il problema della sezione critica che quello della sincronizzazione, è chiaro però che dal punto di vista del programmatore i semafori sono particolarmente scomodi da utilizzare nonché fonte di non pochi errori, il problema deriva dal fatto che si utilizza lo stesso strumento (il semaforo) per gestire sia il problema della mutua esclusione all'interno della sezione critica che per gestire la sincronizzazione. La soluzione adottata è stata quella di creare delle strutture più ad alto livello che permettessero di gestire la mutua esclusione all'interno della sezione critica in modo più trasparente dal punto di vista delle strutture e delle metodologie adottate, queste strutture sono i **monitor**: i monitor sono delle porzioni di codice formate da variabili e procedure che elaborano queste variabili alle quali è garantita la

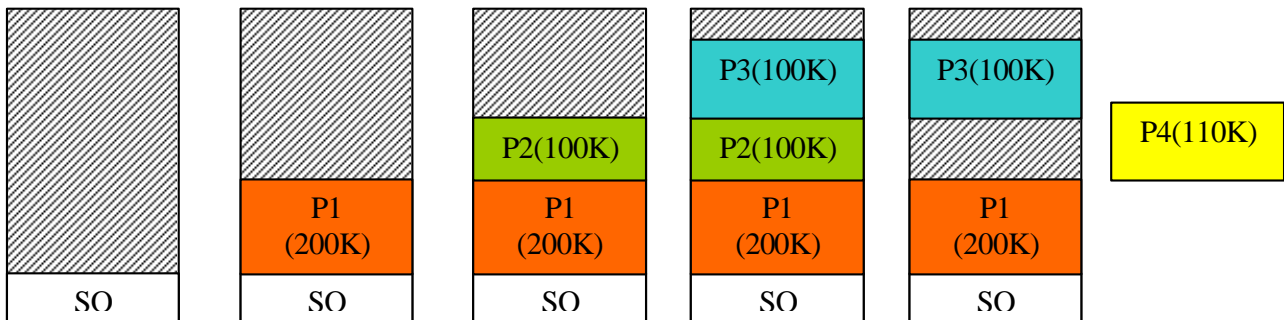
mutua esclusione, una delle prime regole dice che le variabili all'interno di un monitor non possono essere manipolate dall'esterno, in questo modo si garantisce che le sole procedure che possono accedere alle zone di memoria condivise siano quelle del monitor (sottinteso che le zone di memoria condivise sono le variabili dichiarate all'interno del monitor). Inoltre il monitor garantisce che all'interno delle sue procedure non possano esserci più d'un processo attivo in contemporanea, il che garantisce la mutua esclusione senza che il programmatore debba conoscere come questa viene ottenuta (non sempre con i semafori!). Mentre per la gestione della sincronia si utilizzano chiamate di sistema, simili ai semafori, di nome **WAIT** e **SIGNAL** che si occupano di mettere in waiting o in ready un processo.

Altra problematica da gestire è la possibilità di mettere in comunicazione i processi fra di loro dando quindi la possibilità di scambiarsi informazioni (il che è utile in particolare quando non è possibile condividere porzioni di memoria, il che capita per esempio in una rete ove ogni terminale non sia solo un terminale non intelligente ma sia un calcolatore autonomo). Per poter effettuare operazioni di questo tipo il sistema offre delle primitive chiamate **SEND** e **RECEIVE** che permettono di spedire e di ricevere dati fra un processo ed un altro. Una classica struttura utilizzata per svolgere questo compito è il socket, essa è fatta per architetture di tipo client-server in cui il server si mette in attesa delle richieste del client (richieste di trasmissione o ricezione dati).

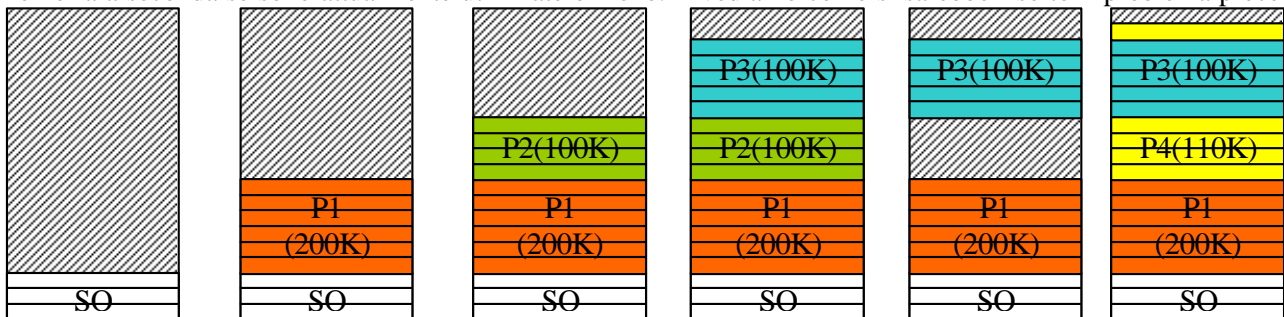
La gestione della memoria

Un'altra risorsa importantissima che deve essere gestita dal sistema operativo è la memoria, essa contiene il SO stesso, i processi (sia il loro codice che i loro dati), deve inoltre essere spartita spesso e volentieri da più utenti che devono poter concorrere per essa e soprattutto devono poter continuare la loro esecuzione. Nei sistemi moderni la memoria si stratifica in: **cache**, **sram/dram/sdram** e di massa (**dischi**). Essa risulta essere sempre più estesa nell'ordine in cui è stata elencata e sempre più costosa nell'ordine inverso, questo porta ad ottimizzare in particolar modo l'utilizzo di memoria cache in modo tale da evitare gli accessi alla ram (che risultano più lenti) e da favorire il processore assicurandogli accessi rapidi alla memoria con costi contenuti. Essendo però la cache molto limitata (mediamente un PC attuale contiene 512KB o 1024KB di cache) si ricorre all'utilizzo della ram, essa è più lenta ma ancora accettabile (si passa dai 10-20 ns della cache ai 60-70 ns della ram), inoltre risulta molto più estesa (attualmente 64-128 MB), in essa sono di solito contenuti, oltre al sistema operativo, tutti i processi attualmente in esecuzione e tutti i loro dati. Infine, quando nemmeno la ram basta per poter memorizzare le informazioni relative ai processi, si ricorre alla cosiddetta **memoria virtuale**: ovvero l'utilizzo del disco come area di swap per la memoria (area in cui scaricare quelle parti di memoria attualmente non utilizzate per liberare dello spazio nella ram). La problematica principale è la gestione di questi tre livelli considerando che un programma, per poter essere eseguito, deve essere residente, assieme ai dati che esso utilizza, all'interno della memoria centrale (ram o cache). Il primo metodo di gestione della memoria richiedeva l'utilizzo di un sistema monoprogrammato, in essa si lasciava il SO nella rom o nella parte bassa della memoria e successivamente si metteva il codice ed i dati dell'unico programma residente ed attivo, esso veniva rilocato (ovvero veniva modificato il suo spazio di indirizzi in modo da renderlo eseguibile nonostante l'inizio del programma non sia all'indirizzo 0) semplicemente sommando agli indirizzi quello di inizio; spesso, essendo il SO fisso e quindi di dimensioni definite, tale operazione veniva svolta direttamente a livello hardware. Tale soluzione è perfettamente funzionante ed efficiente, preclude però ogni strada per la multiprogrammazione! Si è quindi alla ricerca di una soluzione più flessibile che permetta di organizzare più processi all'interno della memoria (fra cui quelli del SO) garantendo a tutti la possibilità di essere messi in stato di running. Ci si organizza allora con i tre livelli sopra descritti tenendo presente però che gli accessi a disco sono un milione di volte più lenti di quelli a cache! Una prima idea banale è stata quella di partizionare la memoria in più parti (segmenti) in modo da permettere ad ogni processo di avere a disposizione la sua parte di memoria ed essere quindi eseguibile assieme agli altri. Il problema della rilocazione viene risolto in questo caso tramite una rilocazione statica: il compilatore (compiler e clinker) crea un codice detto codice-oggetto, ovvero crea il programma considerando come spazio di indirizzi quello che parte da 0 all'inizio del programma; successivamente, quando il programma deve essere caricato in memoria, si attiva un modulo del SO detto loader che ha il compito di caricare in una partizione libera della memoria il programma cambiando tutti gli indirizzamenti in modo che facciano riferimento alla partizione attuale (se per esempio la partizione comincia all'indirizzo 0x7000, questo valore verrà sommato a tutti i campi etichetta delle istruzioni di tipo JUMP e simili). Le problematiche che balzano all'occhio con l'utilizzo di questa

soluzione sono la scarsa flessibilità e lo spreco della risorsa. La scarsa flessibilità è data dal fatto che ogni processo o è in memoria o non lo è, il che richiede di caricare in memoria processi molto grossi quando magari del loro codice in realtà ne serve una porzione molto piccola; per risolvere questo tipo di problema si è dovuta effettuare una modifica alla sorgente: sfruttando la metodologia di programmazione modulare (strutture come funzioni, sottoprogrammi, librerie, oggetti, ecc.) si è pensato di mantenere questa modularità anche all'interno del codice-oggetto in maniera tale da dover caricare all'interno della memoria principale solamente quel modulo attualmente in esecuzione evitando di sprecare memoria con quelli non utilizzati ed evitando completamente di caricare quelli che nell'attuale processo per un motivo o per l'altro non saranno mai utilizzati. La soluzione di prima, ed a maggior ragione questa, però comportano uno spreco spesso non indifferente della risorsa: esso è causato dal continuo caricare e scaricare all'interno della memoria moduli di dimensione differente, si consideri infatti



una situazione come quella mostrata in figura: si ha una memoria di 512K, il SO occupa i primi 62 lasciandone liberi 450, tre processi P1, P2 e P3 arrivano successivamente occupando 400K e lasciandone liberi 50. Successivamente P2 (che occupa 100K) termina la sua esecuzione e rilascia 100K di memoria libera portando il totale a 150K, eppure il processo P4 che arriva successivamente e vuole essere eseguito non ci riesce nonostante occupi soli 110K perché in realtà la memoria contigua libera arriva al massimo a 100K, ovvero la memoria è frammentata! Individuata la causa di tale fenomeno, il fatto che le partizioni non siano di dimensioni fisse, si ricerca una soluzione che eviti questo tipo di problematica; una prima idea sarebbe quella di partizionare la memoria spazi di dimensione fissa (pagine) evitando così di lasciare “buchi” troppo piccini per essere utilizzati all'uscita dei vari processi, questa soluzione è realizzabile, evita parzialmente il problema della frammentazione ma risulta poco flessibile, dovremmo infatti realizzare partizioni tutte della stessa dimensione ma potrebbe capitare che processi troppo grossi non riescano ad essere contenuti in un'unica partizione o che processi troppo piccoli sprechino quantità veramente esagerate di risorsa; la soluzione migliore (ed attualmente adottata⁹ a questo problema è stata quella di utilizzare entrambe le ultime due, ovvero di affiancare paginazione e segmentazione ottenendo la segmentazione paginata. Questa soluzione prevede la suddivisione della memoria in blocchetti di dimensioni fisse, dette pagine, e di suddividere contemporaneamente anche i processi in moduli caricando all'interno della memoria di ogni processo solamente quei moduli attualmente in esecuzione, questi moduli a loro volta saranno formati da pagine (della dimensione di quelle con cui si è suddivisa la memoria) che potranno essere caricate o meno in memoria a seconda se sono attualmente utilizzate o meno. Rivediamo come si sarebbe risolto il problema prece-



dente in una nuova figura in cui si considerano pagine da 25K: in questo caso quando arriva il processo P4 riesce ad entrare comunque in memoria semplicemente caricando le proprie pagine in posizioni differenti, le prime quattro al posto di quelle che prima erano occupate da P2 e l'ultima in cima! Questa soluzione risulta essere l'ottimale, essa richiede però di affrontare alcuni altri problemi come la paginazione (ovvero scegliere quale pagina sostituire nell'evenienza di un page-fault, ovvero nell'evenienza in cui un processo venga ad aver bisogno

di una pagina che attualmente non è caricata), inoltre bisogna gestire in modo più efficace la rilocazione che risulta complessa se non impossibile da eseguire in modo statico tramite il loader in questo caso (potrebbe avvenire che l'operazione di paginazione risulti più onerosa che non l'esecuzione dei processi, il che porta ad un sottoutilizzo della risorsa processore ed allunga i tempi di risposta in maniera elevata).

Cominceremo proprio dalla problematica relativa alla rilocazione: quando si sposta una pagina di memoria dalla memoria al disco o viceversa per risparmiare spazio in memoria centrale o perché una pagina attualmente non caricata serve per la continuazione di un job, si attiva un modulo chiamato swapper che permette appunto lo svolgersi delle operazioni di swap-in (caricamento da disco di una pagina) e di swap-out (scaricamento su disco di una pagina) sfruttando una partizione particolare del disco chiamata swap-area, per eseguire quest'operazione risulta indispensabile tenere in considerazione la problematica degli indirizzamenti, bisogna considerare quindi che il blocco di codice di un modulo di un processo potrebbe essere spezzato in più pagine e quindi l'indirizzamento ottenuto con la rilocazione statica si complicherebbe, inoltre si dovrebbe effettuare la rilocazione ogni volta che lo swapper viene richiamato (sia che si stia eseguendo uno swap-in che uno swap-out), con la logica conseguenza di un enorme rallentamento del sistema (che in questo modo rischia anche il trashing, ovvero il sistema rischia di sprecare più tempo per la paginazione, ovvero per le operazioni di swap, che non per la reale esecuzione dei processi). Come soluzione a questo problema è stata realizzata la rilocazione dinamica: in pratica si ritarda ulteriormente l'operazione di rilocazione che non verrà più effettuata dal loader durante il caricamento in memoria del codice di un processo (in memoria quindi ci saranno sempre pezzi di codice che fanno riferimento ad un indirizzo di base 0h per la prima istruzione), essa viene effettuata in execution time direttamente via hardware tramite un apposito registro (il cui contenuto viene sommato agli indirizzi di salto nelle jump e simili) che indica lo piazzamento della porzione di codice rispetto all'ipotetico 0; in questa maniera, una volta che la gestione del registro di offset viene realizzata (e non si tratta di nulla di particolarmente complicato dato che il suo contenuto risulta il risultato di una semplice moltiplicazione e sottrazione (le pagine sono di dimensione fissa, una volta che si sa che pagina si sta considerando e da che indirizzo essa è calcolata si è a posto) rimane solamente da realizzare un modulo che sia in grado di definire quale sia la pagina che contiene un certo indirizzo che fa riferimento allo spazio di indirizzamenti virtuali (ovvero che considerano l'indirizzo 0 alla prima istruzione).

Tramite questo tipo di soluzione si è risolta la prima problematica, rimane ora da risolvere il problema della selezione delle pagine da scaricare dalla memoria nel caso di un pagefault (ovvero nel caso in cui un processo si trovi nella condizione di dover eseguire delle istruzioni o far riferimento a delle variabili contenute in pagine attualmente non caricate in memoria centrale e che quindi devono venir caricate assolutamente)! La logica con cui si cerca l'algoritmo è quella di ridurre al minimo la frequenza di pagefault dato che essi rallentano il funzionamento del sistema (e se troppo frequenti portano anche a situazioni di stallo che si esprimono con un crash di sistema, basti pensare a due processi che sprecano il proprio quanto di tempo per rubarsi a vicenda delle pagine di memoria andando avanti così all'infinito). Come prima cosa vediamo quale sarebbe la soluzione ottimale che risulta però non realizzabile perché ipotizza di conoscere quando precisamente nel futuro verrà utilizzata nuovamente ogni pagina, successivamente vedremo alcuni algoritmi che approssimano in modo soddisfacente quello ottimale. L'algoritmo ottimale ha una logica molto semplice: ad ogni pagina si assegna il numero di istruzioni che dovrà aspettare perché essa venga realmente utilizzata e si seleziona, perché venga rimpiazzata, la pagina che deve attendere il maggior numero di istruzioni; una pagina quindi che dovrà attendere 8 milioni di istruzioni verrà rimpiazzata preferibilmente rispetto ad una che deve attendere 4 milioni di istruzioni posticipando il più possibile l'avvento del pagefault. Non essendo ovviamente implementabile quest'algoritmo si tenta di approssimarne sfruttando il calcolo statistico: una prima soluzione tiene in conto la proprietà per cui una pagina non utilizzata di recente probabilmente non verrà mai più utilizzata, si sceglie quindi la pagina che non viene utilizzata da più tempo e si rimuove quella per la sostituzione. Facciamo un esempio pratico, si consideri la seguente sequenza di richiesta di pagine da utilizzare nella memoria:

1 2 3 4 5 6 3 4 1 2 3 4

Ipotizziamo inoltre che vi siano 4 frame di memoria utilizzabili per processo e che siano inizialmente vuoti (logicamente, per impedire cannibalismi fra processi nei riguardi della memoria, conviene assegnare ad ognuno di loro un numero che indica quale numero di frame esso ha a disposizione, più avanti vedremo come determinare questo numero in modo tale da ottenere un funzionamento efficace di tutto il sistema), si ipotizzi inoltre che queste siano le richieste di pagina di un unico processo P1: in queste ipotesi i primi 4 riferimenti alle

pagine 1, 2, 3 e 4 causeranno un pagefault (queste pagine sono inizialmente vuote), con esse il numero totale di pagine occupate raggiunge il massimo, quindi con le seguenti 5 e 6 si genereranno altri 2 pagefault ed esse verranno sostituite alle pagine 1 e 2 che sono state utilizzate meno di recente. I seguenti riferimenti a 3 e 4 non genereranno assolutamente nessun pagefault dato che queste due pagine attualmente sono ancora in memoria, verrà però generato per i riferimenti successivi a 1 e 2 che erano state rimosse ed esse verranno caricate rimpiazzando le pagine 5 e 6 che non sono state utilizzate di recente, i successivi riferimenti a 3 e 4 non genereranno pagefault. Il seguente schema mostra l'evoluzione del sistema (la pagina utilizzata è segnata in grassetto di colore rosso se genera pagefault e blu se non lo genera):

1			
1	2		
1	2	3	
1	2	3	4

5	2	3	4
5	6	3	4
5	6	3	4
5	6	3	4

1	6	3	4
1	2	3	4
1	2	3	4
1	2	3	4

Questo esempio mostra in modo esauritivo il funzionamento dell'algoritmo **LRU (Last Recently Used)**, ovvero quello appena presentato che seleziona le pagine utilizzate meno di recente) mostrando come in questo caso vengano generati 8 pagefault, 6 obbligatori (dato che le pagine utilizzate sono 6 e prima o poi ognuna deve

venir caricata per la sua esecuzione) e 2 causati dal fatto che non tutte le pagine utilizzate possono venir caricate in contemporanea nella memoria centrale. Un altro algoritmo di selezione delle pagine da rimpiazzare è l'algoritmo **FIFO (First In First Out)** il cui funzionamento risulta molto semplice: si seleziona per il rimpiazzo la pagina entrata in memoria per prima fra quelle presenti; lascio al lettore per esercizio il calcolo del numero di pagefault generati con la precedente situazione utilizzando l'algoritmo FIFO (risposta:10).

Ora che abbiamo un'idea di che cosa si intende con algoritmo di rimpiazzamento delle pagine è possibile definire gli strumenti che vengono utilizzati per la gestione delle pagine: il primo strumento sono le tabelle delle pagine, ovvero quelle tabelle che permettono di sapere di ogni processo quali sono le pagine da cui è formato, a che indirizzo sono caricate e se sono caricate. Essa si potrebbe quindi organizzare in modo indipendente per ogni processo all'interno di un vettore in cui l'indice indica il numero della pagina, gli elementi del vettore saranno dei record contenenti i campi *indirizzo* e *caricato* di cui quest'ultimo sarà semplicemente un bit. Una gestione di questo tipo però risulta particolarmente onerosa, si considerino infatti processi formati da un numero di pagine particolarmente elevato, si pensi per esempio a Microsoft Word 2000 e si ipotizzi che esso sia della dimensione di 100MB e che le pagine siano di 4KB l'una, si avrebbero quindi circa 25000 pagine, la tabella relativa esclusivamente a questo processo occuperebbe ben $33 \times 25000 = 850000 = 850Kb \approx 100KB$, numeri come questi possono sembrare irrisori, si consideri però che di processi ce ne sono tanti e che quindi se avessimo aperto già solo una decina di applicativi paragonabili a quello dell'esempio un intero MB di memoria centrale sarebbe dedicato unicamente alla tabella delle pagine. La soluzione a questo problema sta nel considerare che non tutte le pagine vengono utilizzate e quindi non tutta la tabella risulta utile, quindi la si divide in blocchi di dimensione variabile, magari si fa riferimento alla segmentazione, e si crea un secondo livello superiore che permette di accedere alle singole sottotabelle; in questo modo per definire una pagina non basteranno più PID (ovvero quel numero che indica il processo cui si fa riferimento) e N° di pagina ma bisognerà indicare: PID che definisce il processo, indice di segmento relativo a quel processo ed indice di pagina all'interno di quel segmento. Altro metodo di memorizzazione delle informazioni relative alle pagine è quello di utilizzare una sola tabella relativa a tutta la memoria in cui si indica per ogni frame a quale processo è relativo e di quel processo a quale N° di pagina corrisponde.

Rimane da risolvere la problematica relativa alla determinazione del numero di pagine assegnabili ad un dato processo, il metodo più efficace adottato è quello di mantenere delle statistiche relative alla frequenza di pagefault che il processo genera ed assegnare inizialmente il numero di pagine a seconda di quanti processi sono attualmente presenti nel sistema o a seconda di previsioni relative a precedenti esecuzioni, successivamente si assegna un valore superiore se i pagefault sono troppo frequenti ed un valore inferiore se sono troppo poco frequenti (la frequenza media accettabile ed i valori di frequenza limite sono definiti in base alle medie calcolate all'interno del sistema su tutti i processi); inoltre è possibile considerare anche le priorità per favorire processi a priorità più alta e sfavorire quelli di priorità meno elevata. Se dovesse capitare che il numero di processi attivi risulta troppo elevato tanto da far salire in modo troppo elevato le medie avvicinandosi pericolosamente al trashing allora si selezionano dei processi vittima (solitamente a priorità bassa) e li si elimina momentaneamente

(scaricandoli interamente su disco) per permettere ad altri di terminare la propria esecuzione e caricarli nuovamente solo quando il sistema risulta più libero.

Per ridurre ulteriormente il numero di pagefault si è preso in considerazione il fatto che ogni processo, quando comincia la sua esecuzione, genera per forza un buon numero di pagefault dovuti al fatto che non vi sono ancora pagine caricate per lui, inoltre mediamente le pagine iniziali di cui un processo ha bisogno sono sempre le stesse, risulta vantaggioso quindi memorizzare all'interno del codice-oggetto i numeri delle prime pagine di cui il processo avrà bisogno in modo da consentire al sistema inizialmente di caricare subito quelle parti del programma che certamente o molto probabilmente serviranno inizialmente evitando così i primi pagefault. L'insieme delle pagine che un processo sta utilizzando e che sono attualmente presenti quindi nella memoria si chiama *working-set*, il *prepaging* (ovvero il caricamento preventivo del *working-set* iniziale di un processo) sfrutta quindi il fatto che all'interno del codice oggetto venga salvato il *working-set* iniziale. Il *working-set* viene salvato anche quando un processo viene momentaneamente sospeso e rimosso dalla memoria in modo tale da permettergli di riprendere la propria esecuzione generando il minor numero possibile di pagefault.

Fin'ora si è fatto riferimento a politiche di paginazione locale, ovvero politiche in cui si considera, nell'evenienza di un pagefault e quindi della ricerca di una pagina da sostituire, solamente il *working-set* del processo che aveva generato il pagefault, in parole povere il processo che aveva bisogno della pagina la caricava al posto di un'altra sua, in realtà esistono anche altri tipi di politiche dette di paginazione globale in cui la scelta non viene ristretta al singolo processo ma a tutti quelli presenti nel sistema, tali algoritmi tendono a penalizzare quei processi che utilizzano troppe pagine generando quindi troppo pochi pagefault. Non analizzeremo questo tipo di politiche.

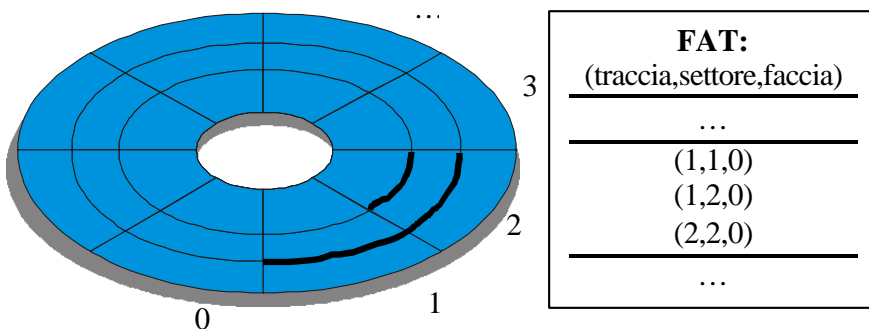
Altre problematiche affrontate per la risoluzione dei problemi relativi alla paginazione o per la sua ottimizzazione sono, per esempio, la realizzazione di demoni di paginazione, ovvero di pezzetti di codice privilegiato del SO che, nel caso si verifichi del *trashing* o comunque ci si trovi in circostanze non ottimali, sia in grado di prendere in mano la situazione e di risolverla (per esempio rimuovendo dei processi dalla memoria). Altra problematica da affrontare è la gestione di pagine condivise e dei diritti di accesso ad ogni pagina: si consideri per esempio il caso in cui più utenti vogliano utilizzare il programma di videoscrittura *vi*, logicamente non ha senso che per ogni utente che lo richiede si carichi in memoria una copia intera di tutto il codice eseguibile, risulterebbe più conveniente caricarlo una volta sola e dividerlo fra tutti gli utenti, è ovvio che il codice eseguibile non dovrà poter essere modificato ma solo letto e quindi le pagine relative ad esso saranno abilitate in sola lettura. Inoltre vi saranno anche delle pagine relative ai dati, quelle dovranno essere distinte per i vari utenti e quindi non dovranno essere condivise. Queste ed altre problematiche ancora sono tutte gestite dal modulo per la gestione ed il paging della memoria.

I file system

Uno dei compiti principali del sistema operativo abbiamo visto essere la gestione dell'hardware in modo trasparente rispetto al programmatore, ovvero il programmatore medio che scrive programmi applicativi non si interfaccia direttamente con la macchina fisica ma con un modello astratto di macchina offerto dal SO. Questo avviene in modo particolarmente evidente per quanto riguarda la gestione dei dischi: i dischi, infatti, per poter funzionare correttamente devono essere gestiti in modo rigoroso e non poco complesso tenendo presente le problematiche inerenti allo spostamento della testina, al posizionamento ed organizzazione delle informazioni, alla memorizzazione di tabelle che consentano di ritrovare i dati, ecc. Tutte queste problematiche vengono affrontate dal file system, ovvero dal modulo di gestione del disco che usufruisce del concetto astratto di file come blocco di informazioni indipendente dal resto. Un file system è organizzato appunto in unità logiche chiamati file, essi possono essere principalmente di due differenti tipologie: *directory* o file di dati; i file di dati sono quelli che contengono le informazioni da memorizzare sul supporto fisico, le *directory* invece contengono altre informazioni relative all'organizzazione del file system e permettono di raggruppare altri file (*directory* comprese). Ogni singolo file avrà, oltre che dati riguardanti la dimensione occupata, la data di creazione ed altri attributi, un nome che permette all'utente (o al programmatore) la sua identificazione, inoltre tutta la struttura è contenuta all'interno di un'unica *directory* chiamata radice ed indicata da una slash "/". Per indicare che un dato file è una *directory* e che si vuole indicare un suo contenuto si usa sempre questo simbolo, se si vuole indicare quindi il file gino contenuto nella *directory* pippo contenuta a sua volta nella *directory* pluto si scriverebbe

“/pluto/pippo/gino”. In molti sistemi vengono memorizzate di ogni file anche altre informazioni particolarmente importanti come gli accessi: si possono avere quindi accessi, per esempio, in lettura, oppure in scrittura, oppure in esecuzione, o una qualunque combinazione di questi, inoltre si potrebbero, in un sistema multiutente, avere tipi di accessi differenti a seconda dell’utente che utilizza il sistema (che razzismo: esistono utenti più o meno privilegiati!), oppure si potrebbero addirittura creare gruppi di lavoro differenti aventi ognuno un GID (Group Identifier) ed un livello di accesso differente. In questa parte vedremo come viene organizzato l’intero file system fisicamente sul disco nel tentativo di ottimizzazione del suo utilizzo e dell’utilizzo dei dati. Come prima cosa definiamo quali operazioni si possono effettuare sui file tramite le system-call offerte dal file system: è possibile eseguire operazioni di apertura di un file (ovvero la dichiarazione al sistema operativo della propria intenzione di utilizzare quel file in un certo modo), essa può essere fatta in differenti modalità (sequenziale, casuale, random: quest’ultimo permette di ricercare in modo completamente trasparente al programmatore informazioni all’interno di un file strutturato apposta per questo tipo di operazioni, non è una modalità presente in tutti i sistemi operativi). Esso può essere chiuso (dichiarazione del programmatore della sua intenzione di non utilizzare più il file, è importante chiudere sempre i file perché sono risorse condivise e non sono utilizzabili da altri utenti se aperti in certe modalità). Inoltre, logicamente, vi si possono effettuare operazioni di lettura e scrittura. Se si aprono i file in modalità ad accesso casuale è anche possibile spostare un puntatore ad esso tramite l’operazione di seek. Passiamo ora a vedere come può essere strutturato un file: un file può innanzitutto essere binario o ascii, ovvero un file di caratteri o un file di dati, inoltre possiamo distinguere ulteriormente per la modalità di utilizzo che può essere a carattere (si legge o scrive un carattere alla volta) o a record (si definiscono delle unità logiche di informazione dette record e si tratta il file considerando i record come unità minima di informazioni leggibili o scrivibili).

Ok, finalmente passiamo da quella che è “l’interfaccia” che il SO offre al programmatore per la gestione del disco alla sua reale organizzazione: cominciamo con il dire che l’organizzazione più efficace sarebbe quella di posizionare i file uno dietro l’altro sul disco in modo continuo (cioè senza spezzettare i singoli file in blocchi separati) e scrivere una bella tabella che indica dove si trova ogni file (dando l’indirizzo del blocco di inizio sul disco con la solita tecnica cilindro-settore-faccia, ovvero suddividendo il disco in tracce (che prendono il nome di cilindri non appena si hanno fisicamente più piatti) e queste in settori (pezzettini di traccia), infine si hanno le facce da dover scegliere (attenzione che non è detto che l’indirizzo di faccia sia qualcosa del tipo *superiore/inferiore* dato che i piatti possono essere anche molti), quindi un indirizzo su disco è formato da 3 parametri). Questa soluzione sarebbe ottima perché, una volta determinata la posizione del file, basterebbe un unico accesso a disco per leggerlo o scriverlo dato che esso è continuo. Purtroppo questa soluzione è anche inottenibile perché dovremmo poter sapere a priori quale sia la dimensione del file quando vogliamo scriverlo, inoltre si avrebbe comunque il problema della frammentazione non risolvibile con operazioni di de-frammentazione assolutamente impraticabili perché particolarmente onerose. Quale potrebbe essere quindi una soluzione alternativa che cerchi di approssimare questa? (penso che si sia capito che spesso si determina la soluzione ottima, che di solito è impraticabile, e successivamente se ne cercano altre alternative che siano praticabili ed approssimino il meglio possibile quella ottima. Una prima soluzione sarebbe quella di suddividere l’intero disco in piccole porzioni (con il metodo descritto prima) chiamate cluster e mantenere in uno spazio appropriato, sempre sul disco, le tabelle che permettono di trovare in sequenza tutti i cluster relativi ad un qualunque file. Il disegno seguente

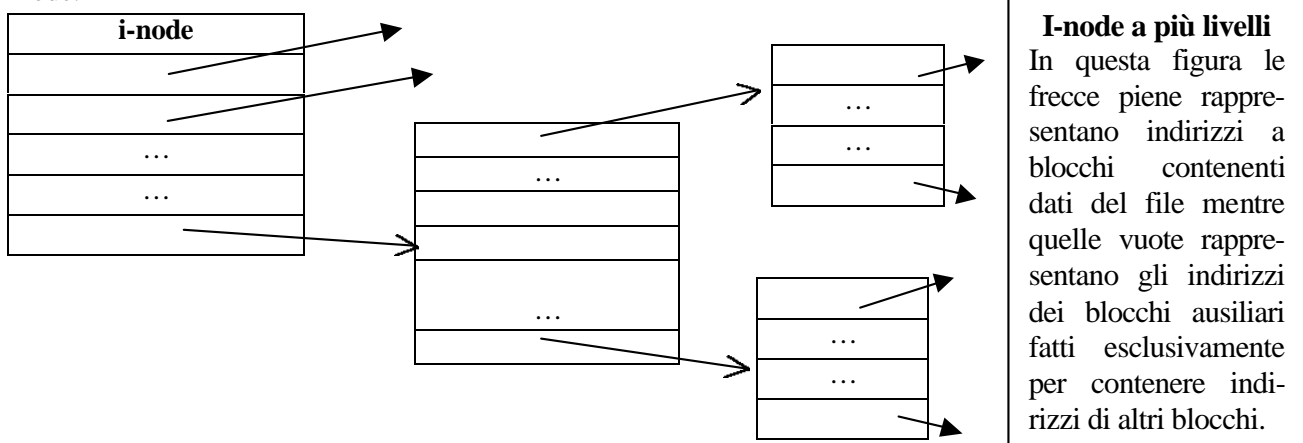


mostra un esempio molto stilizzato di disco e di FAT (File Allocation Table) strutturata come abbiamo detto, in questo esempio si utilizzano 3 cluster per memorizzare un certo file; all’interno della FAT, ad un indice particolare della tabella di cui discuteremo più avanti, si trova l’inizio dei riferimenti al file tramite indirizzamento traccia-settore-faccia, si noti che il file non è contiguo, il che implica mo-

vimenti della testina particolarmente lunghi, in particolare in questo caso sono richiesti almeno 2 rotazioni per

poter leggere o scrivere interamente il file. In realtà però nemmeno 2 sole rotazioni potrebbero bastare: la tabella di allocazione dei file infatti si trova a sua volta su disco, il che implica almeno un accesso per la sua lettura; in realtà proprio la gestione della FAT risulta uno dei punti particolarmente importanti da trattare in questo discorso, infatti essa è residente su disco, ma se la tenessimo costantemente lì dovremmo effettuare un accesso per ogni cluster da leggere solo per sapere dove si trova ed un secondo accesso per leggerlo. Logicamente una cosa simile è inammissibile! Peraltro non è nemmeno possibile pensare di caricarla in memoria, basta un semplice esempio per rendersene conto: si consideri un HD (Hard Disk) di 4GB con cluster da 4KB (assolutamente normale per HD che girano oggi), avremmo 1M cluster di cui tener traccia, se si considera che per avere 1M di indirizzi servono almeno 20 bit tutto ciò significa che come minimo ottimizzando quanto ci pare fino ad utilizzare giusto 20 bit per indirizzo (fra traccia, settore e faccia) ci servirebbero 20MB di memoria! Improprio! Si vuole allora una soluzione intermedia che permetta una gestione flessibile e poco dispendiosa, per implementarla si andrà ad agire direttamente a livello di file-system. Intanto introduciamo un elemento fondamentale di tutta l'organizzazione: l'**i-node**.

L'i-node (nodo indice) contiene al suo interno tutte le informazioni relative al file in questione, ovvero tutti gli attributi (comprese le informazioni relative ai permessi) e la sequenza di indirizzi relativi ai blocchi (cluster) contenenti i dati del file. Questo vale per file di piccole dimensioni, se infatti ci si trovasse di fronte un file di dimensioni elevate quest'organizzazione risulterebbe molto dispendiosa (dato che si presuppone che l'i-node venga caricato in memoria per risparmiare accessi), in questo caso si aggiunge in fondo all'elenco di indirizzi anche quello di un blocco particolare che contiene a sua volta altri indirizzi di altri blocchi del file. Se anche quest'organizzazione risulta troppo dispendiosa allora l'ultimo indirizzo dell'elenco punta ad un blocco contenente gli indirizzi di altrettanti blocchi (qualche centinaio) contenenti ancora indirizzi relativi ai blocchi del file. Sicuramente la figura seguente spiega in modo migliore il livello d'organizzazione a cui si può arrivare con l'i-node.



Sarà quindi possibile, una volta caricato in memoria un i-node (e non certo i blocchi ausiliari che vengono caricati solo in caso di necessità) accedere ad un file in maniera veloce (si ha già a disposizione l'indirizzo della posizione dei dati sul disco) e senza occupare una quantità esagerata di memoria (solo quella dell'i-node ed eventualmente dei blocchetti ausiliari utilizzati, solo quelli realmente utilizzati attualmente). Rimane un unico dilemma: dove si trovano memorizzati gli i-node? Logicamente sul disco, sono anche loro all'interno di qualche cluster sperduto nell'HD. E come si fa a ritrovarli? Semplice: si utilizza il pathname del file cui si vuole fare riferimento e si accede all'i-node caricandolo direttamente dal file di directory relativa al percorso. Per spiegarmi in maniera migliore farò un esempio: consideriamo un file di nome *pippo.txt* contenuto nella cartella *pluto*, il suo pathname sarà */pluto/pippo.txt*, il che significa che nella directory radice, che ricordo essere un file come gli altri ma strutturato apposta per la gestione del file system, vi sarà all'interno della tabella che esso contiene un record con scritto come nomefile *pluto* e come riferimento l'indirizzo dell'i-node relativo al file di directory */pluto*, in quest'ultimo sarà presente ad un certo punto il record contenente come nomefile *pippo.txt* e come riferimento l'indirizzo dell'i-node del file */pluto/pippo.txt*. Da questo è facile capire che l'i-node deve avere le dimensioni di un cluster per poter essere utilizzato come soluzione, consideriamo l'esempio di prima in cui si avevano cluster di 4KB, considerando che un indirizzo generalmente è di 4 byte (32 bit) allora un

cluster può contenere 1024 indirizzamenti, se consideriamo che in realtà vi sono anche altre informazioni relative agli attributi, i permessi ed altri dati relativi al file, allora potremmo approssimativamente dire che, in casi limite un file che utilizzi un unico i-node non dovrebbe essere composto in questo caso da più di 600 cluster, il che è relativamente buono se si considera che in quel caso occuperebbe 2MB! Il calcolo della dimensione massima di un file in questa casistica viene lasciato al lettore (risposta: ~4.7GB, più del doppio della capacità del disco). Com'è fatta quindi una directory per esempio in UNIX (in DOS è strutturata in modo differente)? Semplice, come file contenente una tabella il cui record generico ha 2 campi: nomefile che è una stringa di lunghezza fissa (255 caratteri nei sistemi moderni, 32 caratteri in alcuni sistemi vecchi) e l'indirizzo dell'i-node. Tutti i dati relativi ai file sono contenuti all'interno dei vari i-node. Se si analizza in modo specifico questo tipo di soluzione ci si rende subito conto del fatto che il file system in questo modo viene organizzato non più come un albero ma come un DAG (Directed Acyclic Graph, grafo diretto aciclico) infatti più file di directory potrebbero contenere un riferimento allo stesso i-node nel caso in cui si intenda condividere, per esempio, lo stesso file fra più utenti. Quest'approccio alla condivisione dei file genera però problematiche non di piccola portata: se per caso uno degli utenti dovesse effettuare dei particolari cambiamenti al file tali da cambiare, per esempio, l'indirizzo dell'i-node (magari perché questo viene riscritto sul disco ma ad un indirizzo differente), l'altro utente non potrebbe più accedervi! Per risolvere il problema della condivisione di file (directory comprese) si è utilizzato il metodo del symbolic-link, ovvero del collegamento simbolico: se il file reale è contenuto all'interno di una directory A e lo si vuole condividere con B allora si metterà all'interno di B un file (il symbolic-link, appunto) contenente il pathname del file reale (che identifica univocamente il file indipendentemente dall'indirizzo del suo i-node) che consentirà all'utente B di accedere al file in modo totalmente trasparente. Altre problematiche che entrambe le due soluzioni comportano è quella di consentire a programmi scritti in modo non accurato e che lavorano su file contenuti in directory in ricorsione di processare lo stesso file più volte. Altre considerazioni da fare riguardano le dimensioni dei blocchi del disco: il problema fondamentale è che gestire blocchi troppo grossi rischia di sprecare molto spazio su disco (perché i file di dimensioni minori di quella del blocco lo occupano comunque tutto), mentre blocchi di dimensioni troppo piccole rischiano di sovraccaricare la gestione del file-system e di costringere ad un numero troppo elevato di accessi al disco per l'elaborazione di file di dimensioni medie. In UNIX si è calcolato che la dimensione media dei file è di 1KB, risulta quindi comodo utilizzare blocchi di 512Byte, 1KB o 2KB (come si può notare sono tutte potenze di 2: questo perché la maggior parte dei programmi utilizza blocchi di dati equivalenti a potenze di 2, il che facilita le operazioni di I/O). Infine si ha la problematica della sicurezza: si pretende di ottenere una gestione sicura del disco impedendo a mal'intenzionati oppure a curiosi o ancora a sbadati di manomettere l'intero sistema (magari appropriandosi del file delle password per crackarle e manomettere il lavoro di altri utenti). Una prima precauzione riguarda l'occupazione del disco: se tutti gli utenti potessero utilizzarne quanto piace loro ben presto non si avrebbe più abbastanza spazio a causa della presenza di molti file inutili, si fissano quindi 2 quote massime per ogni utente, una quota soft ed una quota hard. La quota soft può essere superata durante una sessione di lavoro da parte dell'utente, se l'utente non rimuove dei file per recuperare spazio alla prossima sessione di lavoro verrà avvisato del problema; se ancora non diminuisce lo spazio utilizzato gli verrà impedito in maniera automatica di accedere al proprio account se non per mano dell'amministratore di sistema. Se si tenta invece di superare la quota massima hard il sistema operativo lo impedirà con un errore durante la scrittura del file che tentava di eccedere. Per fare ciò il file-system utilizza dei controlli durante l'operazione di scrittura prelevando le quote massime dal file contenente i dati dell'account e confrontandoli con il numero di blocchi attualmente occupati scritto all'interno di un altro file, quando l'utente sta scrivendo un file e viene aggiunto un blocco questo valore viene incrementato, mentre quando viene rimosso un blocco questo viene decrementato. Per quanto riguarda la sicurezza si assegna un proprietario ad ogni file in modo che solamente il proprietario possa manipolarlo (oppure il gruppo), oltre che la root ovviamente. Questo non protegge da tutti gli attacchi che si possono effettuare al sistema ma sicuramente rende le cose un po' più difficili a chi li vuole effettuare!

Il testo qui pubblicato altro non è che un insieme di appunti relativi al corso di sistemi operativi, il pubblicante declina ogni responsabilità!
Lombardi Gabriele.