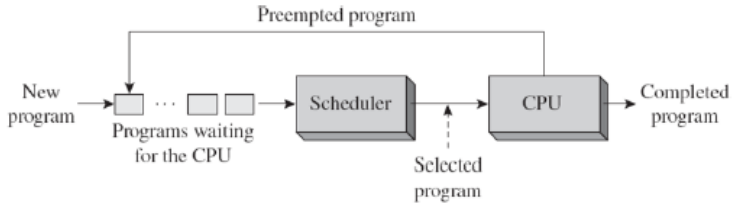
Sistema operativo: insieme di programmi che agisce come intermediario tra gli utenti di un sistema di elaborazione e l'hardware.

Obiettivi e compiti di un S.O.

1. Fornire metodi convenienti per utilizzarlo, è il punto di vista dell'utente. Per conseguire questo obiettivo il SO ha diversi scopi: definisce e rende disponibile una macchina virtuale, rende il software applicativo indipendente dall'HW, facilità la portabilità dei programmi su architetture diverse e permette l'uso delle risorse HW solo tramite chiamate di sistema (system call).
2. Assicurare un uso efficiente delle risorse HD, è il punto di vista del calcolatore. Per far questo, definisce tecniche e strategie con cui assegnare la risorsa, evita conflitti di accesso, massimizza l'uso delle risorse, bilancia l'overhead dovuto al monitoraggio delle risorse e delle prestazioni, protegge da eventuali malfunzionamenti.
3. Prevenire interferenze nelle attività degli utenti.

Da questi obiettivi si ricavano i compiti principali che un SO deve adempiere, ovvero:

- Gestione dei programmi: tramite le cpu ad alte prestazioni si può intervallare l'esecuzione di più programmi. Lo **scheduling** decide quale processo ha la CPU, le quali politiche influenzano le prestazioni e l'uso efficiente della CPU. Il **preemption** (prelazione) significa sottrazione prematura della CPU.



- Gestione delle risorse: allocazione e deallocazione delle risorse possono essere effettuate facendo uso di una tabella delle risorse. Esistono 2 strategie di gestione: **partizionamento delle risorse:** si divide le risorse in partizioni e si decide a priori quali risorse allocare ad ogni programma, in questo caso la tabella ha un rigo per ogni partizione. E' semplice da implementare ma poco flessibile ed estremamente inefficiente. L'altra strategia è **pool (insieme) unico di risorse:** si allocano le risorse prendendole da un insieme, la tabella contiene un rigo per ogni risorsa, il SO consulta la tabella e se la risorsa risulta inutilizzata la usa. Questa strategia comporta un maggiore overhead ma garantisce un uso efficiente delle risorse. **Virtualizzazione delle risorse:** sono risorse fittizie che però permettono di avere a disposizione più risorse di quelle reali. La corrispondenza viene mantenuta dal SO in modo trasparente. Con la virtualizzazione si può: garantire la multiprogrammazione, tramite la memoria virtuale non avere un limite fisico per lo spazio disponibile e per i dispositivi I/O non abbiamo il vincolo di uso in mutua esclusione.
- Sicurezza e protezione: la **sicurezza** riguarda l'uso illegale o interferenze provocate da persone/programmi fuori dal controllo del SO, viene risolto con l'autenticazione. La **protezione** invece sono le stesse situazioni ma eseguite da utenti del SO, si risolve con due modalità di utilizzo del SO, una modalità utente (limitata), una per il sistema operativo (illimitata). La memoria invece viene protetta monitorando chi ne fa accesso per sapere sempre la causa del problema.

Componenti principali di un sistema di elaborazione

I componenti principali sono:

Hardware, Sistema Operativo, Programmi di sistema, Programmi applicativi, Utenti.

Evoluzione storica dei SO

- 1° generazione (1945-1955), caratterizzati da sistemi grandi e costosi. Venivano controllati da un utente programmatore esperto tramite una consolle. I sistemi di I/O erano lettori di schede perforate, unità di nastro e stampanti.
- 2° generazione (1955-1965), l'evoluzione è caratterizzata dall'introduzione dei nastri magnetici. Prima di questi si operava con le schede perforate, quindi il programmatore le consegnava al tecnico il quale aveva il compito di eseguirle, il programmatore tornava poi giorni dopo per sapere

l'esito, provocando un allontanamento tra macchina e programmatore. Grazie ai dischi magnetici invece abbiamo potuto abbattere questa problematica e introdurre la tecnica dello spooling. Otteniamo con esso 2 miglioramenti: si abbattono i tempi di attesa della cpu e si attuano politiche di scheduling. Viene poi introdotta l'idea della multiprogrammazione, cioè l'esecuzione di più programmi insieme.

- 3° generazione (1965-1980), si sposta l'attenzione verso l'uso efficiente delle risorse continuando con la multiprogrammazione. Viene introdotta la memoria virtuale per favorire il multitasking. Con queste innovazioni vengono finalmente realizzati programmi applicativi e quindi introdotte periferiche quali tastiere e mouse, ma ancora troppo lenti per la cpu. Per risolvere questo problema si introducono sistemi **time-sharing**. Questa tipologia di sistemi permettono di sottrarre la cpu dopo un tempo prestabilito di utilizzo. Sistemi **real-time** sono creati per controllare il corretto funzionamento di uno specifico sistema fisico con l'obiettivo che ogni programma termini rispettando i vincoli temporali. Si suddividono in 2 tipologie: hard real-time, garantiscono il completamento di compiti critici in un dato livello temporale. Soft real-time, danno priorità ai task critici fin quando non vengono completati.

- 4° generazione (1980-oggi), si sono sviluppati i **personal computer**, grazie all'abbassamento dei costi dei materiali. Questi nuovi sistemi sono orientati verso l'utente, infatti godono di un'interfaccia utente molto sviluppata. **Sistemi paralleli**, sono sistemi multiprocessore che permettono l'esecuzione di più programmi contemporaneamente, ma condividono le risorse. Ne esistono di due tipi: Symmetric multiprocessing (SMP): ogni processore esegue una copia del so così si ha un parallelismo nell'esecuzione delle diverse cpu. Asymmetric multiprocessing (AMP): ogni processore ha un suo task, ma ne esiste uno master che coordina tutti gli altri allocando il lavoro. **Sistemi di rete e distribuiti,** hanno lo scopo di condividere le risorse distribuendole tra molti processori. Si hanno due tipi: SO di rete, ogni nodo ha il suo SO e i trasferimenti avvengono esplicitamente, poco tolleranti ai guasti e complessi per gli utenti; SO distribuiti, tutti i nodi hanno lo stesso SO, sono più complessi da progettare ma affidabili e tolleranti ai guasti.

Meccanismi di protezione

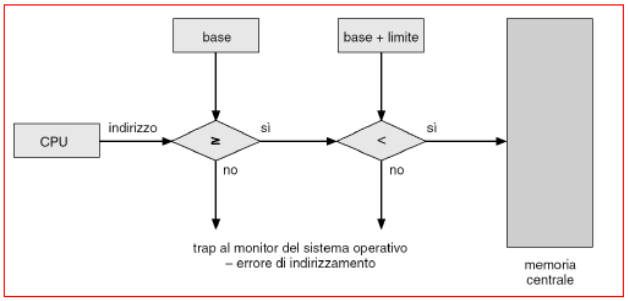
Sono necessari soprattutto nei sistemi multiutenti e multiprogrammati. Dobbiamo infatti garantire che: i programmi non si ostacolino tra di se, il SO non subisca danni da programmi applicativi e i dati di ogni utente protetti da eventuali accessi sbagliati o maliziosi. Per far questo ci sono 2 grandi tipologie di meccanismi: software e hardware.

Protezioni hardware:

- Duplice modalità di funzionamento: il processore ha due diverse modalità con cui possiamo usarlo, una *utente*, per la normale esecuzione dei programmi e alla quale non è permesso accedere liberamente a tutte le risorse di sistema. L'altra modalità è *monitor (supervisore, di sistema, kernel),* si usa per l'esecuzione di servizi richiesti al SO (system call) e non ha alcun limite sulle operazioni effettuabili. Questo meccanismo funziona solo se i programmi utente accedono alle risorse solo tramite system call (con relativo switch di modalità) e dobbiamo anche garantire che nessun programma utente ottenga mai il controllo del calcolatore mentre si trova in modalità monitor.

System call: è una richiesta che un programma fa al kernel attraverso un interrupt software. Ne esistono di vari tipi (controllo dei processi, gestione dei file, gestione dei dispositivi, gestione delle informazioni di sistema, realizzazione delle comunicazioni). Solitamente sono scritte in linguaggio assembler, ma non sono accedute direttamente da programmi, infatti usano una **Application Programming Interface (API)** di più alto livello. I vantaggi di usare una API sono 2: portabilità delle applicazioni, ovvero un programma sviluppato usando una certa API girerà in ogni sistema che la mette a disposizione; facilità d'uso, sono scritte in linguaggio di alto livello più facile da capire e gestire. La system call è implementata associandoli un numero, l'interfaccia che la gestisce ha una tabella indicizzata in base al numero, quindi invoca la system call desiderata e quando termina restituisce lo stato e i valori di ritorno. Non c'è bisogno di sapere come è implementata la system call basta usarla correttamente. Per passare i parametri alle system call ci sono 3 metodi: parametri inseriti nello stack del programma e prelevati dal SO o viceversa; parametri memorizzati in una tabella in memoria e si passa il suo indirizzo in un registro; parametri memorizzati in registri hardware.

- Protezione dell'I/O, tutte le istruzioni sono privilegiate e gestite come system call.
- Protezione della memoria, dobbiamo proteggere perlomeno il vettore degli interrupt per evitare che un programma assuma il controllo del sistema in modalità monitor. Per fare questo usiamo 2 registri: **registro base**, contiene il più piccolo indirizzo fisico destinato al programma, **registro limite**, lunghezza dell'area di memoria riservata al programma. Tutta la memoria che non è compresa in questo intervallo risulta inaccessibile al programma ed è quindi protetta. Solo il SO, tramite istruzioni privilegiate, può modificare i dati presenti nei due registri. Per quanto riguarda la **protezione della memoria cache**, in questo caso il problema si pone perché nella cache di L1 si accede con indirizzi logici, sui quali non è possibile effettuare un controllo con i registri, in quanto più programmi possono usare gli stessi indirizzi logici. Per risolvere quindi il problema ci sono 2 soluzioni: la più semplice è quella di cancellare tutti i dati al momento di un context switch, questo però ha due inconvenienti, il primo riguarda il fatto di cancellare anche dati che non sono del programma terminato, il secondo riguarda l'efficienza, infatti il prossimo programma avrà un sacco di miss. La soluzione più complessa ma anche più intelligente è quella di associare ad ogni blocco un identificativo del programma a cui appartiene, l'accesso al blocco è consentito solo al programma corrispondente all'identificativo.



- Protezione della CPU, per evitare che un programma entri in un loop infinito e monopolizzi la CPU, si usa un timer, scaduto il quale la CPU viene automaticamente sottratta. Il timer si utilizza per implementare quindi sistemi time-sharing e l'istruzione per impostarlo è privilegiata.

Struttura di un SO

Funzioni di un SO

- Gestione dei processi, un processo è un programma in esecuzione, il quale per funzionare ha bisogno di risorse, memoria file e dispositivi di I/O. Il SO ha le responsabilità di creare ed eliminare i processi, di sospenderli e riprendere l'esecuzione. Inoltre il SO deve fornire i meccanismi per la sincronizzazione, la comunicazione e la gestione dei deadlock.
- Gestione della memoria principale, la memoria principale è un vettore di grandi dimensioni di parole o byte, ognuna col proprio indirizzo e rapidamente accessibile, essa è condivisa da CPU e dispositivi di I/O. Il suo contenuto è volatile, ovvero si perde in caso di spegnimento. Il SO ha le seguenti responsabilità in merito: tener traccia di quali parti di memoria sono usate e da chi, decidere quali processi caricare quando ci sono aree disponibili, allocare e deallocare spazio in base alle necessità, proteggere gli accessi alle locazioni.
- Gestione dei file, un file è un insieme di informazioni correlate e definite dal suo creatore. E' un'unità di memorizzazione logica ed astratta, solitamente rappresentano programmi o raccolte di dati. In merito ai file il SO ha le seguenti responsabilità: creazione e cancellazione di file o directory (insiemi di file), supporto di primitive per manipolare file e directory, associazione dei file ai dispositivi di memoria secondaria, backup di file su dispositivi non volatili, controllo degli accessi per garantire una protezione adeguata.
- Gestione della memoria secondaria, siccome la memoria principale è troppo piccola e soprattutto volatile, serve una memoria grande e permanente in cui fare il backup della memoria principale. Il SO ha le seguenti responsabilità: gestione dello spazio libero, allocazione della memoria, scheduling delle richieste di operazione.
- Gestione dei dispositivi di I/O, si compone di un'interfaccia generale per i driver dei dispositivi e un driver per ogni specifico dispositivo. Le responsabilità del SO sono: mascherare la diversità e la complessità dei vari dispositivi, gestire la competizione dei processi, gestire l'I/O delle informazioni (buffering, caching, spooling).

- Sicurezza e protezione, riguarda l'uso illegale o interferenze operate da persone/programmi esterni (sicurezza) o interni (protezione). Per determinare gli accessi si fanno autenticare gli utenti, in questo modo ognuno sarà identificato con un user id e tutti i file e processi suoi saranno associati a quel id. Inoltre esistono anche i group id che permette l'associazione di file o processi a un gruppo di utenti. Sono quindi meccanismi atti a autorizzare l'uso delle risorse, questi meccanismi devono: specificare quali controllo si devono fare, distinguere tra autorizzati e non, fornire metodi per imporre i criteri stabiliti.

Progetto e implementazione

Il principio principale di progettazione è di separare politiche e meccanismi. Le **politiche** ci dicono cosa dobbiamo fare, mentre i **meccanismi** come farlo (per esempio dispatcher fa il context switch, meccanismo, mentre lo scheduler implementa lo scheduling, politica). Questa divisione permette la massima flessibilità in caso di modifiche politiche o aggiunta di componenti hardware. Un altro punto importante riguarda il fatto che un SO, per ammortizzare i costi di implementazione, dovrebbe durare almeno 10 anni, questo lo costringe a doversi adattare ai cambiamenti tecnologici. Risultano quindi importanti due caratteristiche:

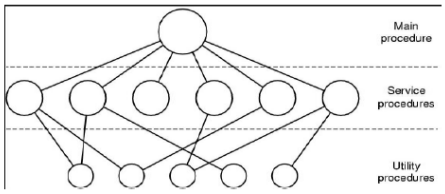
- **Portabilità**, che rappresenta la facilità di adattare software a un altro pc. Questa caratteristica richiede il cambiamento delle parti di codice dipendenti dall'HW, ovvero che riguardano i meccanismi. Il porting è facilitato se la parte dipendente ha dimensione ridotta ed è ben separata da quella indipendente con due interfacce ben definite.

- **Espandibilità**, facilità con cui si possono aggiungere funzionalità al software. Questa caratteristica è necessaria per aggiungere nuovo hardware e fornire nuove funzionalità in risposta alle aspettative degli utenti. Per far questo i sistemi moderni utilizzano una funzionalità **plug-and-play**, con la quale è possibile aggiungere componenti anche durante l'esecuzione del SO. Il SO gestisce l'interrupt causato dall'aggiunta del dispositivo e seleziona il software adatto e lo integra al kernel.

Categorie e modelli strutturali

Si differenziano in base al settore applicativo: Batch multiprogrammati, time-sharing, real-time, distribuiti. Oppure in base all'architettura HW: monoprocessore, multiprocessore, sistemi distribuiti e reti. Il modello strutturale di un SO rappresenta come sono organizzate le sue componenti e come interagiscono.

- Sistemi monolitici, SO costituito da un unico programma che gestisce le interazioni tra le componenti tramite chiamate a procedure. Adatti a sistemi molto semplici e non multiprogrammati. E' presente una sola organizzazione minima: un programma principale chiama una procedura di servizio, le procedure di servizio realizzano le system call. (MS-DOS)



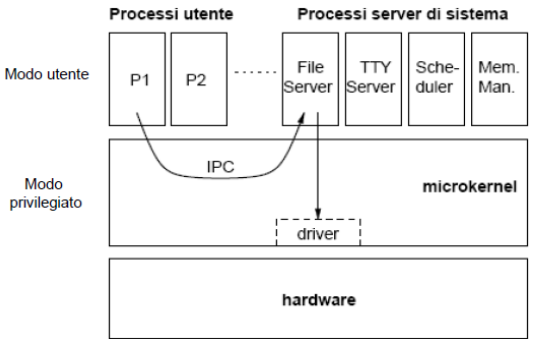
- Sistemi modulari, SO costituito da moduli, ognuno dei quali fornisce una determinata funzionalità. Ogni modulo è caratterizzato da un'interfaccia (specifica le funzionalità offerte) e un corpo (implementazione). Esistono due categorie di moduli: procedure di servizio (direttamente invocabili con system call) e le procedure di utilità (le prime le usano ma non sono visibili agli utenti).

- Sistemi a livelli, l'obiettivo è quello di semplificare il progetto e la verifica del SO riducendo il numero di connessione tra i moduli. Per fare questo si usa una struttura gerarchica a livelli, dove ognuno definisce un tipo di servizio e le modalità per essere utilizzato. La gerarchia prevede che ogni livello può utilizzare solo le funzionalità di quello a lui immediatamente sottostante. Gli svantaggi sono un rallentamento del SO a causa dei livelli e una complessità nella progettazione nel dividere le funzionalità.

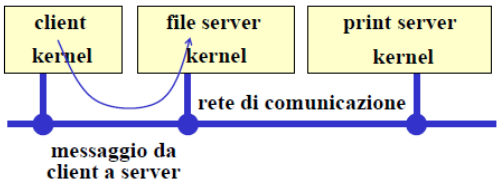
- Sistemi basati su macchina virtuale, si tratta l'hardware e il kernel del SO come se fosse tutto hardware, ovvero si fornisce una macchina virtuale con un'interfaccia simile a quella dell'hardware sottostante. In questo modo permettiamo di avere una MV per ogni utente che si sceglie il SO e esegue i suoi programmi. Questa implementazione si basa sulla condivisione delle risorse HW. I principali vantaggi sono: il funzionamento concorrente dei SO ospite come processi normali di SO, protezione completa delle risorse di sistema, perfetto per l'emulazione e lo sviluppo di un nuovo

SO. Mentre il principale svantaggio è la complessità nel realizzarlo, insieme al costo elevato. Per fronteggiare questo problema spesso si utilizzano MV senza utilizzare un SO MV.

- Sistemi a microkernel, è un'estremizzazione del principio di separazione politiche e meccanismi, infatti il kernel fornisce solo i meccanismi (comunicazione tra processi, gestione della memoria e dei processi minimale, gestione dell'hardware di basso livello) mentre le politiche vengono implementate come processi normali (tramite server). Questa implementazione però riduce le prestazioni, infatti il dover richiedere l'uso di usa risorsa tramite IPC rallenta il sistema. I vantaggi invece sono la grande flessibilità, è più affidabile e sicuro ed è adatto ad ambienti di elaborazione di rete e embedded.



- Sistemi client-server, sono caratterizzati da processi utenti (client) che richiedono un servizio, e i processi di SO (server) che accontentano la richiesta, i due processi possono essere allocati su pc diversi. Un microkernel presente su ciascun sistema gestisce le risorse e le comunicazioni tra i processi.



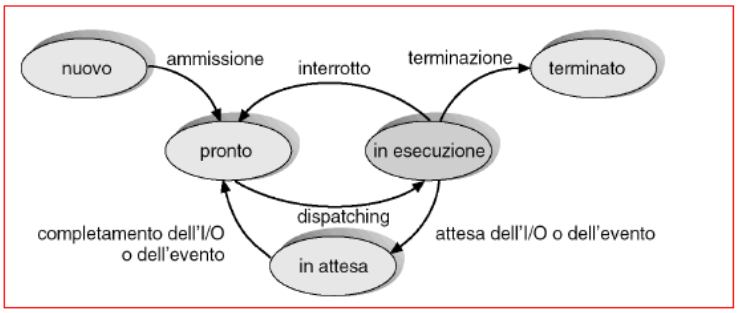
Processi e Thread

Modello concorrente e processi

Un SO consiste in un grande numero di attività eseguite contemporaneamente da parte della CPU e dei dispositivi di I/O. Il modello ci serve per descrive ed analizzare la coesione delle diverse attività. Il modello scelto è quello concorrente, che si basa sul concetto astratto di processo. Un programma è un'entità statica/passiva infatti descrive semplicemente un algoritmo, il processo invece è un'entità dinamica, identifica l'attività dell'elaboratore relativa all'esecuzione di un programma, è l'unità esecutiva di un SO ed è l'unità di allocazione delle risorse necessarie all'esecuzione. Un processo può condizionare o essere condizionato da altri processi (interagente) oppure no (indipendente).

Stati di un processo

Durante l'esecuzione un processo cambia stato:

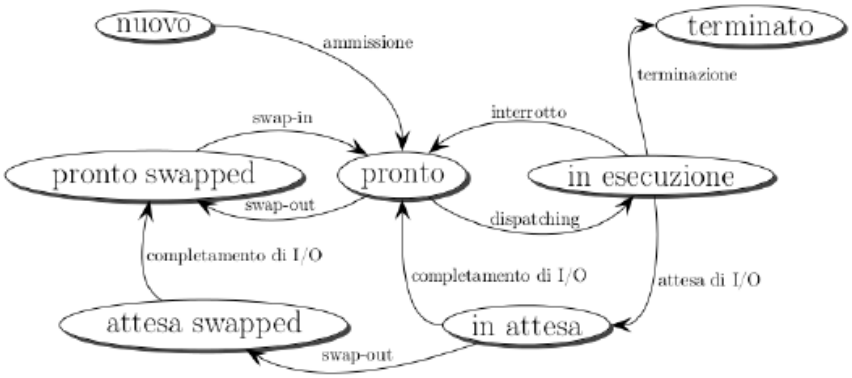


Descrittore di processo

Ogni processo ha una struttura dati associata detta Process Control Block (**PCB**). Tutti i PCB presenti in un sistema sono organizzati in una tabella (tabella dei processi) memorizzata in un'area di memoria principale accessibile solo dal kernel del SO. Quanto un processo è pronto o in attesa, le informazioni relative vengono salvate nel PCB, quando questo passa in esecuzione vengono recuperate. Il PCB deve quindi contenere informazioni sia per quando il processo è in esecuzione che quando non lo è. Una cosa importante da vedere è il **context switch**, effettuato ogni volta che la CPU passa da un processo all'altro. Questa operazione è effettuata dal SO attraverso i seguenti passi: 1. Salvataggio del contesto del processo in esecuzione nel PCB (salvataggio di stato). 2. Inserimento del PCB nelle code di attesa o nei pronti. 3. Caricamento dell'indirizzo del PCB del nuovo processo in esecuzione. 4. Caricamento del contesto del nuovo processo nei registri del processore (ripristino di stato). Il context switch è puro tempo di gestione del sistema quindi è un overhead.

Schedulazione dei processi

Durante l'esecuzione i processi si alternano da una coda all'altra tra processi pronti e in attesa di dispositivi di I/O. Un registro della CPU gestito dal SO punta al PCB del processo in esecuzione. Lo scheduling dei processi riguarda quindi l'attività con la quale si sceglie tra i processi a chi assegnare la CPU e chi spostasse in memoria principale. Esistono 3 tipologie di scheduling: a **breve termine** (sceglie tra i processi pronti quindi interviene quando il processo in esecuzione perde il diritto di usare la CPU), a **medio termine** (trasferisce temporaneamente i processi in memoria secondaria quando la somma delle richieste è maggiore della memoria principale disponibile), **a lungo termine** (sceglie tra quelli in memoria secondaria chi portare in memoria principale, permette quindi il controllo del grado di multiprogrammazione). I processi possono essere **I/O-bound**, ovvero spendono più tempo a fare I/O oppure **CPU-bound**, orientati ai calcoli. Le prestazioni migliori si ottengono con una combinazione equilibrata delle due tipologie di processi e a volte limitando la multiprogrammazione. Quando è presente lo scheduling a medio termine cambia il diagramma degli stati di un processo, infatti va aggiunta la parte dello swap-in e swap-out:



Operazioni sui processi

Le operazioni sui processi sono sostanzialmente 3: creazione, terminazione e interazione. Tutte eseguibili tramite system call.

- **Creazione**, può essere statica (inizializzazione del sistema) oppure dinamica (system call per creare un processo da un altro, richiesta da parte di un utente di creare un nuovo processo). I compiti del SO sono: assegnazione di un identificatore, allocazione di memoria principale, allocazione di altre risorse, inizializzazione e collegamento del PCB con le altre strutture del SO. Un processo *padre* può creare dei processi *figli* che a loro volta ne possono creare altri ecc. Il kernel deve quindi fornire meccanismi per fare in modo che: le risorse possono essere condivise totalmente parzialmente o non condivise; l'esecuzione può essere concorrente o il padre aspetta la terminazione del figlio; l'uso dello spazio degli indirizzi può essere del figlio clone di quello del padre oppure il figlio è caricato in un programma differente.
- **Terminazione**, può avvenire con o senso consenso. Le situazioni tipiche sono: uscita normale (volontaria), fatal error (involontaria) e terminazione forzata da un altro processo (involontaria). In quelle volontarie il processo esegue l'ultima istruzione oppure una exit, grazie ad una wait il padre può ricevere dati spediti dal figlio che esegue una exit, le risorse vengono deallocate dal SO. La terminazione involontaria avviene quando un padre forza la terminazione dell'esecuzione del figlio (abort), questo succede se: il figlio ha ecceduto nell'uso delle risorse, il task a lui assegnato non serve più oppure se il padre sta per terminare (terminazione a cascata).

Processi interagenti

Due o più processi possono interagire tra loro con due schemi diversi:

- Cooperazione, desiderata e prevista. I processi sono logicamente connessi e ciascuno ha bisogno dell'altro. Prevede una sincronizzazione diretta con operazioni eseguite secondo un ordine ben preciso.
- Competizione, non desiderata ne prevista. Ogni processo evolve indipendentemente dall'altro ma si trovano in conflitto per l'uso di risorse. La sincronizzazione è indiretta e se la risorsa è seriale avviene in mutua esclusione.

I vantaggi della cooperazione sono la condivisione di informazioni, la modularità e le prestazioni. I principali modelli di interazioni sono due:

- Ambiente locale, ovvero senza dati ne memoria condivisa. L'ambiente è visto come un insieme di processi ciascuno operante nel locale dove gli altri non possono accedere direttamente. Gli spazi di indirizzi dei processi possono essere separati, l'interazione avviene o con scambio di informazioni temporali (sincronizzazione) oppure tramite invio di messaggi (comunicazione). Gli strumenti usati sono messaggi e primitive quali send e receive. Eventuali problemi di cooperazione e competizione si risolvono con lo scambio di messaggi.
- Ambiente globale, ovvero con condivisione dei dati. L'ambiente è visto come un insieme di processi e oggetti. Richiede la mutua esclusione per l'accesso ai dati condivisi, spazi di indirizzamento condivisi (basta parzialmente) e gli strumenti usati sono semafori e primitive wait, signal, monitor. I problemi di cooperazione e competizione si risolvono con wait signal semafori e monitor.

Comunicazione tra processi

Avviene tramite meccanismi con cui i processi si scambiano i dati (**Inter-Process Communication IPC**). L'IPC fornisce due operazioni: send e receive. Sono entrambi messaggi e se due processi si vogliono mettere in comunicazione prima stabiliscono un canale di comunicazione e poi grazie alle due operazioni dell'IPC si scambiano i messaggi. La realizzazione di un canale può essere logica o fisica, noi analizziamo quella logica.
Comunicazione diretta simmetrica: ogni processo si nomina con l'altro esplicitamente send(P, message) receive(Q, message). Le proprietà sono: canali stabiliti/rimossi automaticamente, un canale è associato ad una sola coppia di processi e quella coppia ha solo quel canale, esso è solitamente bidirezionale. Gli svantaggi sono la limitata modularità, infatti basta cambiare il nome di un processo per non far funzionare più la comunicazione.
Comunicazione diretta asimmetrica: solo il mittente deve specificare il ricevente send(P, message) receive(id, message) ovvero si riceve il messaggio nella variabile message e si registra il nome di chi l'ha mandato in id.
Comunicazione indiretta: i processi comunicano inviando messaggi da/a una porta, quindi ogni porta ha un identificativo unico e i processi che vogliono comunicare devono accedere alla stessa porta. Le primitive sono send(A, message) e receive(A, message) dove A è l'id della porta. Le proprietà sono: un canale è stabilito solo se i processi condividono una porta, una porta può essere condivisa da più canale e ognuno può condividere più porte, un canale può essere unidirezionale e bidirezionale. Si può verificare un problema quando ci sono tipo 3 processi 2 ricevono e 1 invia, chi riceve il messaggio? Le possibili soluzioni sono 3: permettere la condivisione solo di 2 processi per porta, permettere a un solo processo per porta di ricevere, permettere al sistema di selezionare il ricevente in modo arbitrario comunicando al mittente l'id di chi ha ricevuto il messaggio. Di chi è una porta? Di un processo, quindi fa parte dello spazio degli indirizzi, esso è l'unico ricevente e la porta muore quando lui termina. Del SO che offre ai processi alcune operazioni quali creare/rimuovere porte e spedire/ricevere messaggi da/a una porta. Il proprietario della porta può passare il diritto ad un altro.
Comunicazione sincrona & asincrona, lo scambio di messaggi può essere**: bloccante** (sincrono), ovvero all'invio ci si blocca finché il messaggio non arriva a destinazione, oppure la ricezione è bloccata finché non si riceve un messaggio. **Non bloccante** (asincrono), cioè si invia messaggi e si riprende l'attività, il processo ricevente acquisisce un messaggio valido o uno nullo. I messaggi inviati risiedono in una cosa, ovvero nei canali di comunicazione sono associate delle code di messaggi. Per implementarle ci sono 3 metodi: Capacità 0-0 messaggi (il mittente si blocca finché non si riceve il messaggio), Capacità limitata – lunghezza finita (il mittente si blocca quando la cosa è piena), capacità illimitata – lunghezza infinita (il mittente non si blocca mai).

Paradigmi di interazione: produttore-consumatore, client-server

Il socket è l'estremità di un canale di comunicazione identificato con un indirizzo IP e dal numero di una porta. La comunicazione si effettua tra una coppia di socket ovvero tra due processi che comunicano tramite una reta che usa una coppia di socket. Quando si effettua la chiamata di procedura remota (RPC) il sistema invoca lo stub per passarli i parametri (stub: segmento di codice). Lo stub del client si occupa di individuare la porta per comunicare con lo stub del server, effettua il marshalling dei parametri e l'unmarshalling di eventuali risultati. Lo stub del server riceve il messaggio, effettua l'unmashalling dei

parametri, invoca la procedura sul server e fa eventualmente un marshalling dei risultati. Un problema potrebbe essere come fare a determinare la porta del server, ma si risolve con un processo del SO che si occupa di dare la porta giusta.

Thread

Processi e Thread, un processo può essere visto come un elemento che possiede delle risorse più un flusso di controllo dell'esecuzione, quindi uno stato della CPU, questo è il **thread**. La gestione separata delle risorse e del flusso di controllo permette di avere processi multithread che condividono le risorse.
Concetto di Thread, i thread di uno stesso processo risiedono all'interno dello spazio di indirizzamento e condividono sezione di codice, sezione di dati e risorse. Un thread è caratterizzato da: identificatore, stat di esecuzione, spazio di memoria, contesto (registri), stack e descrittore. Come abbiamo detto i processi possono essere **single-thread**, operano indipendentemente e servono per svolgere job non correlati. Oppure possono essere **multithread**, condividendo le risorse, sono adatti a job correlati. L'uso dei thread ha diversi vantaggi: prontezza di risposta (in uno multithread se un thread è bloccato in attesa, un altro può continuare l'esecuzione senza bloccare completamente il processo), condivisione di risorse (la cooperazione tra thread multipli migliora le prestazioni permettendo un'esecuzione parallela), performance (sempre per l'alto grado di condivisione fa in modo che il context switch sia più veloce perché non richiede operazioni di gestione della memoria). Gli svantaggi invece sono: non implementano meccanismi di protezione (non dovrebbe essere un problema perché si suppone che thread che condividono cooperino e non competono), introducono problemi di controllo della concorrenza e sono molto difficili da progettare.
Gestione dei thread, ci sono tre modelli di thread che differiscono per il ruolo che i processi ed il kernel del SO hanno nella creazione e nella gestione.

- Thread a livello utente, la gestione è effettuata tramite una libreria di funzioni a livello utente collegata al codice di ogni processo. Sarà presente una tabella dei thread per ogni processo, il kernel non è a conoscenza dei thread lui schedula solo processi. Quando il thread invoca una funzione di libreria che richiede un evento, la stessa funzione schedula un altro thread per l'esecuzione, se non lo trova invoca una system call "block me". A questo punto il processo sarà sbloccato solo quando un thread si libererà. I principali vantaggi sono: efficienza e flessibilità, dovuti alla sincronizzazione e schedulazione effettuate dalla libreria, e la portabilità garantita dal fatto che il SO gestisce solo i processi e interviene con context switch solo tra processi. Gli svantaggi invece: non c'è prelazione e l'invocazione di una system call bloccante blocca tutti i thread, questo perché il kernel non riconosce la differenza tra thread e processo quindi non si ha scheduling automatico tra i thread. Inoltre non si sfruttano sistemi multiprocessori in quanto il kernel è sequenziale, e non sono molto utili per i processi I/O bound.
- Thread a livello kernel, la gestione è affidata direttamente dal kernel del SO che la supporta con una propria tabella dei thread. Quando si verifica un evento, il kernel salva lo stato corrente della CPU del thread interrotto nel TCB, lo scheduler considera tutti i thread ready e ne seleziona uno per l'esecuzione. Il dispatcher usa il puntatore al PCB del TCB per verificare se il thread selezionato appartiene a un processo differente da quello di prima, nel caso salva lo stato del processo interrotto e carica quello nuovo. I vantaggi sono: esiste uno scheduling della CPU per thread, conveniente per il programmatore (è come programmare per processi), consente il parallelismo all'interno di un processo, è utile per il programmi I/O bound. Gli svantaggi sono: la politica di scheduling è fissata dal kernel, meno efficiente (serve una system call per ogni operazione sui thread) e meno portabile.
- Approccio ibrido, prevede l'uso sia di thread a livello utente che a livello kernel e un metodo per associarli. I vantaggi sono: quelli a livello utente + livello kernel e una maggiore flessibilità. Svantaggio: scarsa portabilità Ecco i diversi modelli:
    - Molti-a-uno, molti thread a livello utente sono associati a un thread a livello kernel. Ridotto overhead di commutazione, esecuzione dei thread in interleaving, blocco del processo in caso di blocco di un thread.
    - Uno-a-uno, ogni thread utente ne ha uno kernel. E' come avere tutti kernel, permette quindi l'esecuzione dei thread in overlapping e la commutazione è effettuata a livello kernel quindi produce un maggiore overhead.

- Molti-a-molti, molti thread a livello utente sono associati a molti a livello kernel, l'implementazione è complessa ma permette un ridotto overhead di commutazione e un'elevata concorrenza e parallelismo.

Scheduling della CPU

Concetti di base

L'obiettivo principale è quello di utilizzare al meglio la CPU, questo si ottiene tenendo più processi in memoria contemporaneamente, facendone eseguire uno alla volta finché non deve attendere un evento. Durante l'esecuzione quindi si alternano momenti di esecuzione di istruzioni (**CPU burst**) e attesa di eventi o operazioni esterne (**I/O burst**). I momenti CPU burst di processi CPU-bound sono caratterizzati da pochi burst CPU molto lunghi, mentre quelli I/O bound ne hanno molti ma corti. Il kernel quindi bada di tenere in memoria un mix appropriato di processi I/O bound e CPU bound. Solitamente si assegna una priorità ai processi, generalmente I/O ha priorità maggiore dei CPU bound. In ogni momento la CPU è assegnata al processo con priorità maggiore, se uno con priorità minore sta eseguendo e uno maggiore vuole la CPU, quello con priorità minore viene interrotto. Se aumentiamo la multiprogrammazione possiamo: aggiungere un programma CPU-Bound, il quale avrebbe priorità più bassa rispetto a quelli di I/O bound e quindi non li influenzerebbe e aumenta l'uso della CPU perché sfrutta gli intervalli in cui i I/O bound fanno operazioni su I/O. Oppure possiamo aggiungere un programma I/O bound, il quale avrebbe una priorità maggiore di tutti quelli ci CPU bound, ridurrendone il progresso, inoltre aumenta l'utilizzo dell'I/O. Il **throughput** è il numero di processi completati da un sistema in un'unità di tempo. Contribuiscono al suo incremento anche i programmi con bassa priorità a patto che hanno possibilità di essere eseguiti. **Scheduling della CPU** a breve termine, decide quale processo mandare in esecuzione tra quelli pronti. La politica a breve termine si contrappone con quella a medio, che sceglie i processi parzialmente eseguiti da spostare in memoria secondaria, e da quella a lungo, che decide i job da trasformare in processi regolando il grado di multiprogrammazione. Lo scheduler interviene: da esecuzione a attesa, da esecuzione a pronto, da attesa a pronto, da esecuzione a terminazione. Quelli senza prelazione fanno solo il 1 e il 4 caso, mentre quelli con intervengono in tutti e 4. Gli scheduler con prelazione hanno maggiore overhead ma offrono anche servizi migliori. Il **dispatcher** è un modulo del SO che dà il controllo della CPU al processo scelto dallo scheduler, ovvero implementa il meccanismo mentre lo scheduler la politica. Il dispatcher quindi si occupa del context switch, l'eventuale passaggio alla modalità utente della CPU e il salto alla giusta istruzione. La latenza di dispatcher è il tempo che esso impiega a interrompere un processo e inizializzarne un altro.

Criteri di scheduling

Frequenza di completamento (throughput), numero di processi completati in un'unità di tempo.
Tempo di completamento, tempo trascorso dall'ingresso di un processo fino al suo completamento.
Tempo di attesa, somma degli intervalli di tempo necessari per terminare un processo.
Tempo di risposta, tempo trascorso dall'invio della richiesta fino alla ricezione della risposta.
I criteri su cui opera un algoritmo di scheduling sono diversi: massimizzare l'uso della CPU, la frequenza di completamento, minimizzare il tempo di completamento, di attesa, di risposta e la varianza del tempo di risposta. Le politiche di gestione invece definiscono i principi con cui il kernel del SO assegna la CPU ai processi, e possono differire per 3 aspetti: senza/con prelazione, senza/con priorità, statiche/dinamiche. Politiche con prelazione, sono in grado di forzare l'interruzione del processo soddisfacendo esigenze di priorità o di equità, e evitando il monopolio della CPU da parte dei processi CPU-bound. Senza prelazione si basano sul rilascio spontaneo della CPU da parte del processo. Politiche senza priorità, ogni processo ha la stessa priorità quindi si basano su strategie di ordinamento FCFS (First come, first served). Con priorità, si dividono i processi secondo l'importanza o la criticità rispetto al tempo di esecuzione. Politiche statiche, ogni processo conserva la propria priorità nel tempo, penalizza i processi a bassa priorità (starvation). Dinamiche, i processi aumentano la priorità a seconda del tempo passato in coda, più aspettano più aumenta, garantendo la CPU a tutti.

Politiche di scheduling

- FCFS (First come first served), i processi ottengono la CPU nello stesso ordine in cui arrivano, è senza prelazione e senza priorità.
- SJF e SRTF (shortes-job-firts e shortes-remaining-time-first), assegna la CPU al processo con tempo più breve. La priorità è data dalla sua lunghezza. SJF non prevede la prelazione, quindi una volta ottenuta la CPU il processo la tiene finché non termina. SRTF invece prevede la prelazione, quindi

mentre un processo è in esecuzione se ne arriva uno più corto quello in esecuzione viene sostituito, questo può provocare starvation. La soluzione per la starvation è l'invecchiamento (aging), ovvero si aumenta la priorità man mano che si aspetta (priorità dinamiche).

- RR (scheduling circolare Round Robin), ogni processo ha un quanto di tempo prestabilito assegnato, una volta terminato viene inserito nella coda dei pronti.
- Scheduling con code multilivello, siccome abbiamo gruppi di processi che per la loro natura hanno esigenze diverse, possiamo per esempio dividere i processi in due gruppi (I/O bound e CPU bound), ognuno con la sua coda e il suo algoritmo di scheduling. A questo punto dobbiamo gestire lo scheduling tra le code, e si può fare con priorità fissa, prima serve tutti quelli di una coda e poi di un'altra (possibile starvation) oppure con quanto di tempo, ogni coda ottiene la CPU per un certo tempo.
- Scheduling con code multilivello con feedback (retroazione), prevede la possibilità di spostare un processo tra le code dei gruppi, potendo così implementare l'invecchiamento. A questo punto ci serve sapere il numero di code, l'algoritmo di scheduling per ognuna, un metodo per sapere quando un processo va spostato da una coda all'altra e un metodo per sapere in quale coda mettere un processo quando richiede un servizio.
- Scheduling per sistemi multiprocessore (omogenee), si attuano quando abbiamo a disposizione più CPU omogenee, in questo caso è indifferente chi esegue il task, quindi si cerca solo di suddividere il carico. Questo però genera un problema, infatti tutte le CPU devono accedere alle strutture del kernel, esistono due schemi di risoluzione: si condividono le strutture con hardware e controlli particolari (symmetric multiprocessing), oppure si fa che solo una CPU master può accedere alle strutture dati e sarà lei che prende le decisioni di scheduling per tutti (asymmetric multiprocessing).
- Scheduling per sistemi in tempo reale, ci sono due categorie: tempo reale stretto (hard real-time), in cui i processi critici vanno eseguiti in un tempo dichiarato e garantito, serve prenotare le risorse e determinare i tempi di risposta. Oppure a tempo reale lasco (soft real-time), dove i processi critici hanno priorità maggiore rispetto agli altri, in questo caso possono coesistere con i sistemi time-sharing a patto di avere dei vincoli sullo scheduler, quali: gestione di processi con priorità, le priorità dei real-time non diminuisce nel tempo e garantire una latenza di dispatcher minima. Per garantire questa latenza minima ci sono due soluzioni: punti di prelazionabilità, ovvero inserire dei punti in cui se c'è un processo con priorità maggiore si può interrompere il processo in corso per fare spazio a quello maggiore. Oppure col kernel prelazionabile, ovvero le strutture dati del kernel sono gestite con semafori in modo che un processo possa essere interrotto in ogni momento.
- Scheduling dei thread, abbiamo due tipi di thread: schedulazione di User Level Thred (ULT), che coinvolge il kernel solo quando si cambia processo, e kernel level thread (KLT), schedulazione che avviene in base ai thread. La differenza tra i due riguarda le prestazioni, infatti: nell'ULT il context switch tra thread dello stesso processo richiede pochissime istruzioni macchina (molto veloce) mentre nel KLT può richiedere il cambio delle mappe di memoria e l'invalidazione della cache (molto lento).

I principali obiettivi di un algoritmo di scheduling sono diversi a seconda dei sistemi su cui opera, nello specifico:

- Tutti i sistemi, garantire equità, assicurare che la politica non possa essere aggirata e bilanciare il carico di lavoro tra le componenti di sistema.
- Sistemi batch, massimizzare la frequenza di completamento e l'uso della CPU, minimizzare il tempo di completamento.
- Sistemi interattivi, minimizzare il tempo di risposta, soddisfare le aspettative dell'utente.
- Sistemi real-time, rispettare i vincoli temporali.

Valutazione degli algoritmi di scheduling

Per valutare gli algoritmi di scheduling bisogna innanzitutto fissare i criteri base e poi utilizzare un metodo per la valutazione, per esempio:

- Modelli deterministici, usano una valutazione analitica, considerando un carico di lavoro predeterminato si guardano le prestazioni di un algoritmo per quel carico. Vantaggi: è semplice e rapido, svantaggi: non sempre è possibile conoscere a priori il carico e inoltre esso varia spesso nel tempo.
- Modelli con reti di code, si cerca di determinare tramite misurazioni e approssimazioni o tramite stima, la distribuzione di probabilità delle durate dei CPU burst e I/O burst, e gli istanti di arrivo dei

processi. Gli svantaggi sono che le classe di algoritmi analizzabili sono limitate e l'analisi è così complessa che spesso richiede semplificazioni che fanno perdere di significato il risultato.
- Simulazioni, ovvero si programma un modello del sistema e si fanno girare gli algoritmi misurandone le prestazioni. Svantaggi: è molto oneroso, nonostante dia risultati molto precisi.
- Realizzazioni, il modo migliore per testa un algoritmo è quello di inserirlo in un SO e vedere come si comporta durante il suo normale utilizzo, questo però è costoso per le modifiche da apportare al SO e inoltre danneggia gli utenti costretti a subire continue modifiche del SO.

Stallo (deadlock)

I deadlock si verificano quando alcuni processi attendono indefinitamente le azioni da parte di qualche altro processo, ovvero nei seguenti casi: sincronizzazione tra processi (attendono segnali uno dall'altro), comunicazione tra processi (si invia un messaggio solo dopo averne ricevuto uno), condivisione di risorse (i processi attendono che altri rilascino le risorse di cui hanno bisogno). Il SO gestisce il deadlock per la condivisione di risorse. Lo stallo quindi è una situazione che va evitata infatti: impedisce l'avanzamento dei processi in stallo, sottrae definitivamente le risorse a loro assegnate e degrada le prestazioni di sistema.

Modello delle risorse di un sistema

Nel modello concorrente, il sistema è composto da un insieme finito di risorse distribuite a più processi in competizione. Una risorsa è una qualsiasi entità hardware o software che serve per l'esecuzione di un processo. Le risorse sono suddivise in classi, infatti un processo richiede una risorsa di una classe non una specifica. Inoltre le risorse si possono suddividere in tipologie:

- Condivisibili, che si possono usare simultaneamente da più processi. Non condivisibili (seriali), usate in mutua esclusione.
- Statiche, sono assegnate a un processo dalla sua creazione alla sua terminazione. Dinamiche si assegnano durante l'esecuzione.
- Riusabili, possono essere date ripetutamente in uso ai processi. Consumabili, si possono usare una volta sola.

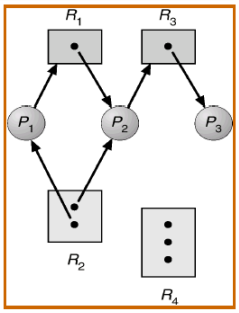
Per utilizzare una risorsa un processo deve effettuare il seguente ciclo di operazioni: richiesta, uso, rilascio. Come abbiamo detto la gestione delle risorse compete al SO, il quale usa un gestore per ogni risorsa, il quale è in grado di riceve e soddisfare le richieste d'uso e il rilascio di tale risorsa. Le richieste e i rilasci avvengono tramite system call. Ogni volta che un processo usa una risorsa il SO controlla che ne abbia fatto richiesta e che non sia già stata assegnata. Il SO ha una tabella dove registra lo stato di ogni risorsa e nel caso in cui fosse assegnata anche il processo che la sta usando. Nel caso in cui un processo richieda una risorsa già assegnata, viene accodato. In questo modo però si può verificare starvation, infatti se un processo fa una richiesta per una risorsa attualmente in uso da un altro processo, finché quest'ultimo non la rilascia il primo rimarrà bloccato, quindi se non garantiamo un rilascio può verificarsi starvation. Per questo motivo gli scopi dello scheduling di una risorsa sono: evitare starvation, minimizzare il tempo di attesa medio e la variabilità tra i tempi di attesa, rispettare le priorità.

Caratterizzazione del deadlock

Uno stallo si verifica solo se si presentano simultaneamente le seguenti 4 condizioni:

1. Mutua esclusione, risorse utilizzabili solo da un processo alla volta.
2. Possesso e attesa, i processi trattengono risorse che già posseggono e ne chiedono altre aggiuntive.
3. Mancanza di pre-rilascio, le risorse già assegnate ai processi non possono essere sottratte, ovvero le può rilasciare solo il processo volontariamente.
4. Attesa circolare, esiste un insieme di risorse {P1…Pn} in attesa tale che P1 attende P2, P2 attende P3..e via finché Pn sta aspettando P1.

Queste condizioni sono necessarie ma non sufficienti affinché uno stallo si verifichi. Per stabilire se un insieme di processi è in stallo si devono analizzare le informazioni sulle risorse allocate e le richieste. Queste informazioni costituiscono lo stato di allocazione e si rappresentano tramite grafo o tramite matrice. **Grafo di allocazione delle risorse,** grafo(V, E) dopo V sono i nodi (ce ne sono di 2 tipi: P, che rappresentano i processi, e R, che rappresentano le risorse) mentre E è l'insieme degli archi (sempre di 2 tipi: di richiesta P->R, di assegnazione R->P). Se il grafo di allocazione non contiene cicli allora non c'è stallo. Se contiene cicli, dipende dal numero di istanze delle risorse, se ogni risorsa ha una sola istanza



si ha deadlock, altrimenti c'è la possibilità ovvero la condizione è necessaria ma non è sufficiente.

Metodi di gestione dei deadlock

Per assicurarsi che un sistema non entri mai in stallo dobbiamo:

- Prevenire i deadlock, quando si scrivono i programmi vengono creati dei metodi vincolanti che impediscono il verificarsi delle 4 condizioni simultaneamente.
- Evitare i deadlock, il SO riceve delle informazioni dai processi sulle risorse che intendono richiedere e per quanto tempo, così possiamo attuare una politica che evita lo stallo.
- Rilevare e ripristinare, se si verificano il SO interviene e recupera uno stato corretto precedente.
- Ignorare il problema assumendo che non si verificano mai.

Prevenire i deadlock: prevenzione statica, ovvero che riguarda la stesura dei programmi con la quale si fa in modo che non si verificano le 4 condizioni simultaneamente. **Mutua esclusione**, è necessaria solo per le risorse seriali, anche se per quelle hardware si può risolvere con la virtualizzazione, mentre per i file condivisi e altre risorse software è importante garantirla per evitare inconsistenze. **Possesso e attesa,** i vincoli da garantire sarebbero, l'allocazione globale delle risorse (un processo le richiede tutte insieme) e la richiesta può essere effettuata solo da processi che non hanno risorse. Questi vincoli però hanno degli inconvenienti, infatti dobbiamo conoscere a priori le esigenze di un processo, le risorse sono scarsamente utilizzate e c'è possibilità di attesa indefinita su risorse molto utilizzate; tutto questo porta a un'inefficienza delle risorse, riduce il grado di multiprogrammazione e l'efficienza della CPU. **Mancanza di pre-rilascio,** si risolve facendo in modo che se un processo richiede risorse che non possono essergli assegnate, allora rilascia tutte anche quelle che già possiede, quindi si crea una lista con tutte le risorse di cui ha bisogno e riprenderà l'esecuzione solo quando saranno tutte disponibili. Gli inconvenienti sono: prerilascio complicato e si adatta solo a risorse il cui stato può essere salvato e ripristinato in seguito. **Attesa circolare, si** definisce una relazione di ordinamento tra tutti i tipi di risorsa, così ogni processo deve chiedere le risorse rispettando tale ordinamento. Ovvero, se Ri<Rj, allora le risorse di tipo Ri vanno richieste prima di quelle Rj, in modo analogo un processo non può richiedere Ri se già possiede Rj.

Evitare i deadlock: prevenzione dinamica, ovvero si usano algoritmi che esaminano dinamicamente lo stato di allocazione e accertano che la condizione di attesa circolare non si verifichi mai. Per far questo necessitiamo di informazioni a priori su come vengono richieste le risorse dai processi, però ipotizzando che i processi una volta ottenute le risorse richieste, terminano rilasciandole tutte, possiamo comunque stabilire se uno stato è sicuro. **Stato sicuro**, uno stato è sicuro se esiste un ordine con cui il sistema può allocare le risorse richieste senza deadlock (sequenza sicura). La sequenza <P1,..Pn> è sicura se per ogni Pi, le richieste di risorse che può farse Pi possono essere soddisfatte usando le risorse attualmente disponibili e le risorse possedute da tutti i processi Pj con j<i. Se il sistema è in uno stato sicuro non si verificano stalli, quindi per evitarli dobbiamo garantire che, l'assegnazione di risorse ai processi sulla base delle loro richieste, non faccia diventare lo stato non sicuro (con deadlock). Si può verificare tramite un *algoritmo basato sul grafo di allocazione delle risorse*, questo metodo è applicabile solo se ogni risorsa ha una sola istanza. Si aggiunge una tipologia nuova di archi, l'arco di prenotazione P-->R (linea tratteggiata), le risorse devono essere prenotate a priori, ovvero prima che un processo vada in esecuzione. Quando un processo richiede effettivamente una risorsa, l'arco di prenotazione diventa di assegnazione se la risorsa è disponibile altrimenti di richiesta. Se questo non comporta un ciclo, l'assegnazione può essere effettuata altrimenti il processo deve attendere. Quando un processo rilascia una risorsa, l'arco di assegnazione è convertito in un arco di prenotazione. *Algoritmo del banchiere*, è applicabile sempre e viene attivato a ogni richiesta e rilascio. Ogni processo deve dichiarare il numero massimo di istanze per ogni risorsa che può aver bisogno durante l'esecuzione, quando ne fa richiesta si deve stabilire se l'assegnazione lascia il sistema in uno stato sicuro, se è così bene, altrimenti il processo deve attendere. Siccome un processo ottiene tutte le risorse richieste, deve restituirle in un periodo di tempo finito. Abbiamo: n= numero dei processi, m=numero di tipi di risorse, Available= vettore di lunghezza m (rappresenta il numero di istanze disponibili per tipo), Max= matrice n x m (rappresenta ogni processo quante risorse di ogni tipo vuole al massimo), Allocation= matrice n x m (rappresenta le risorse allocate a ogni processo), Need= matrice n x m (differenza tra Max e Allocation), Request= matrice n x m (rappresenta le richieste per ogni processo e richiesta). Algoritmo, Work e finish sono due vettori di lunghezza m e n:

1. Inizializzazione: Work=Available Finish[i]=false per i=1,2,…n
2. Cercare un indice i tale che: Finish[i]==false Needi<=Work Se non esiste si passa al punto 4
3. Work= Work+Allocation Finish[i]=true, tornare al punto 2
4. Se per ogni i, Finish[i]==true, allora lo stato è sicuro

Algoritmo del banchiere:

1. Se Request i <= Need i, passare al punto 2, altrimenti sollevare un errore.
2. Se Request i <= Available, passare al punto 3, altrimenti Pi deve attendere.
3. Il sistema assegna al processo le risorse modificando lo stato nel modo seguente:

   Available= Available – Request i

   Allocation i= Allocation i + Request i

   Need i = Need i – Request i

   Si invoca quindi l'algoritmo per verificare che lo stato è sicuro, se lo è le risorse vengono effettivamente assegnate altrimenti si ristabilisce lo stato precedente e il processo attende.

L'algoritmo viene attivato ad ogni richiesta e ad ogni rilascio, quando si fa una richiesta si inserisce nella matrice Request, quando avviene un rilascio si aggiorna lo stato di locazione facendo:

Available= Available + k

Allocation i= Allocation i - k

Need i = Need i + k

Rilevare i deadlock: siccome gli algoritmi che evitano i deadlock considerano il caso pessimo, ovvero che tutti i processi richiedono le risorse residue di cui hanno bisogno contemporaneamente, e sono quindi costosi, riducono il grado di multiprogrammazione e non sono sempre applicabili. Si può permettere che uno stallo si verifichi, bisogna quindi usare un algoritmo che lo rilevi e si applica uno schema per il recupero dell'integrità del sistema, questo provoca un overhead per la memorizzazione delle informazioni, per l'esecuzione dell'algoritmo di rilevamento e per i costi di ripristino. Se ogni tipo di risorsa ha una sola istanza si può verificare lo stallo tramite un semplice grafo di attesa, dove i nodi rappresentano i processi e ci sono archi se Pi aspetta una risorsa attualmente in uso da Pj (nel grafo di allocazione avremo Pi->Rk Rk->Pj). Se nel grafo c'è almeno un ciclo, nel sistema si è verificato uno stallo. Se invece le risorse hanno istanze multiple, si controlla la presenza di un deadlock cercando di costruire sequenze fattibili che associano a ogni processo le richieste che li servono. Abbiamo: n= numero dei processi, m= numero di risorse, Available= vettore di lunghezza m, che indica il numero di istanze disponibili per ciascuna risorsa, Allocation= matrice m x n, indica il numero di risorse di ogni tipo allocate a ogni processo, Request= matrice che indica le richieste correnti per ogni processo. L'algoritmo di rilevamento quindi procede così:

1. Work= Available se Allocationi i ≠ 0 allora Finish[i] = false, altrimenti finish[i]=true
2. Trova un indice i tale che Finish[i]==false && Requesti <= Work Se non c'è vai al punto 4
3. Work=Work+Allocation i Finish[i]=true si torna al passo 2
4. Se finish[i]==false per qualche i allora il sistema è in stallo, se Finish[i]==false significa che Pi è bloccato.

Se un processo Pi ha Allocation i== 0 significa che non possiede risorse assegnate quindi non è in attesa circolare, si pone Finish[i]=true. Se invece Finish[i]==false, al termine dell'algoritmo, allora Pi è in stallo ma se è true potrebbe anche essere comunque in stallo. Infatti potrebbe succedere che non possiede risorse ma è bloccato da risorse assegnate ad un processo coinvolto in un'attesa circolare. Se ipotizziamo, ottimisticamente, che per completare la propria esecuzione Pi non richiede risorse aggiuntive rispetto a Request i, allora la situazione di prima non si verifica. Ma se l'ipotesi non fosse valida si potrebbe verificare, anche se il deadlock sarebbe poi trovato in esecuzioni future dell'algoritmo. Quando va invocato questo algoritmo? La frequenza dipende dalla probabilità di presentarsi un deadlock e quanti processi ne saranno coinvolti. Possiamo quindi decidere di richiamarlo, ad ogni richiesta non immediatamente soddisfacibile (caso estremo), con cadenza periodica oppure quando le prestazioni di degradano. Se si invocasse arbitrariamente non sarebbe possibile capire chi ha provocato lo stallo.

Modalità di ripristino dal deadlock:

Abbiamo due possibilità: informare l'operatore del sistema o effettuare un ripristino automatico (interviene su processi e risorse). Si può quindi **terminare forzatamente i processi**: se si terminano tutti è molto costoso perché poi andranno rieseguiti, allora ne terminiamo uno alla volta fin quando non ci è eliminato lo stallo, ovviamente in questo caso dobbiamo rieseguire più volte l'algoritmo. Si inizia a terminare dal meno costoso da rieseguire, poi in base alla priorità, al tempo che è stato in esecuzione e quanto ancora gli manca, quante e quali risorse ha assegnate e quante ne richiederà. **Rilascio anticipato delle risorse**, si fanno rilasciare preventivamente le risorse fin quando non si elimina il ciclo, si seleziona la "vittima" con un algoritmo e tramite il rollback si riporta in uno stato corretto il processo a cui è stata tolta. Può causare starvation quando si seleziona la vittima sui fattori di costo, quindi si potrebbe finire per selezionare sempre la stessa che quindi non verrà mai eseguita fino in fondo.

Ignorare i deadlock:

Siccome il SO gestisce molte tipologie di risorse, è costoso evitare i deadlock in tutte, quindi si può attuare una prevenzione o semplicemente ignorarli. L'ignorare il problema è fattibile perché i deadlock non si verificano spesso ed è la soluzione meno costosa. In pratica il SO gestisce i deadlock in modo diverso e separatamente per ogni tipo di risorsa, nello specifico:

- Memoria, è prelazionabile quindi non può causare deadlock e viene rilasciata tramite swapping.
- Dispositivi I/O, non si può attuare l'allocazione globale (comporta una scarsa efficienza dei dispositivi) ne l'ordinamento perché in generale non a tutti gli utenti va bene il solito, quindi dove si può si creano dispositivi virtuali.
- File e messaggi tra processi, il SO ne ha troppi quindi non può ordinarli, si risolve facendo gestire i deadlock direttamente ai processi che fanno uso di uno stesso insieme di file o messaggi.
- Blocchi di controllo e strutture dati del kernel, si può fare la politica di ordinamento ma ancora più semplice e fattibile è l'allocazione globale.

Gestione della memoria

Analogie e differenze con la gestione della CPU

I programmi in attesa di essere eseguiti risiedono su disco in forma di file eseguibili, ovvero in formato binario pronti per caricare in memoria tutte le informazioni del programma. Per mandarlo in esecuzione, il programma deve quindi essere spostato in memoria centrale e attivato da un processo a esso dedicato. Quindi analogamente alla CPU in cui non è il processo a chiedere di essere assegnato, qui non è il programma a chiedere di andare in esecuzione, perché questo comporterebbe che esso si trovi già in memoria centrale e stia già eseguendo. Le differenze sono invece che la memoria centrale, diversamente dalla CPU, può essere assegnata a più processi contemporaneamente, esiste quindi l'opportunità di condividere codice e la necessità di utilizzare meccanismi di protezione.

Spazi di indirizzamento logici e fisici

La virtualizzazione della memoria avviene associando ad ogni processo una sua memoria virtuale in cui allocare il programma, i dati e lo stack. Il SO ha il compito di allocare la memoria fisica per fornire il supporto via via alla memoria virtuale dei processi. La memoria virtuale è organizzata in base alle esigenze del processo, mentre quella fisica in funzione di obiettivi di efficienza complessiva del sistema. Esistono quindi 2 tipologie di indirizzi: logico (virtuale), generato dalla CPU e fisico, visto dall'unità di memoria e caricato nel memory address register (MAR). La corrispondenza tra indirizzi logici e fisici è gestita da meccanismi HW/SW. Una copia dello spazio virtuale di ciascun processo è allocata in memoria secondaria, ma generalmente solo una porzione dello spazio virtuale è allocato in memoria fisica, il processo avanza finché accede a porzioni presenti in memoria fisica, sta quindi al SO allocare dinamicamente e in modo trasparente la memoria.

Aspetti caratterizzanti

I meccanismi per la gestione della memoria sono caratterizzati da 4 paramenti principali:

- Rilocazione degli indirizzi (statica o dinamica)
- Allocazione della memoria (contigua o non contigua)
- Organizzazione dello spazio di memoria virtuale (unico o segmentato)
- Caricamento dello spazio virtuale in memoria fisica (unico o a domanda)

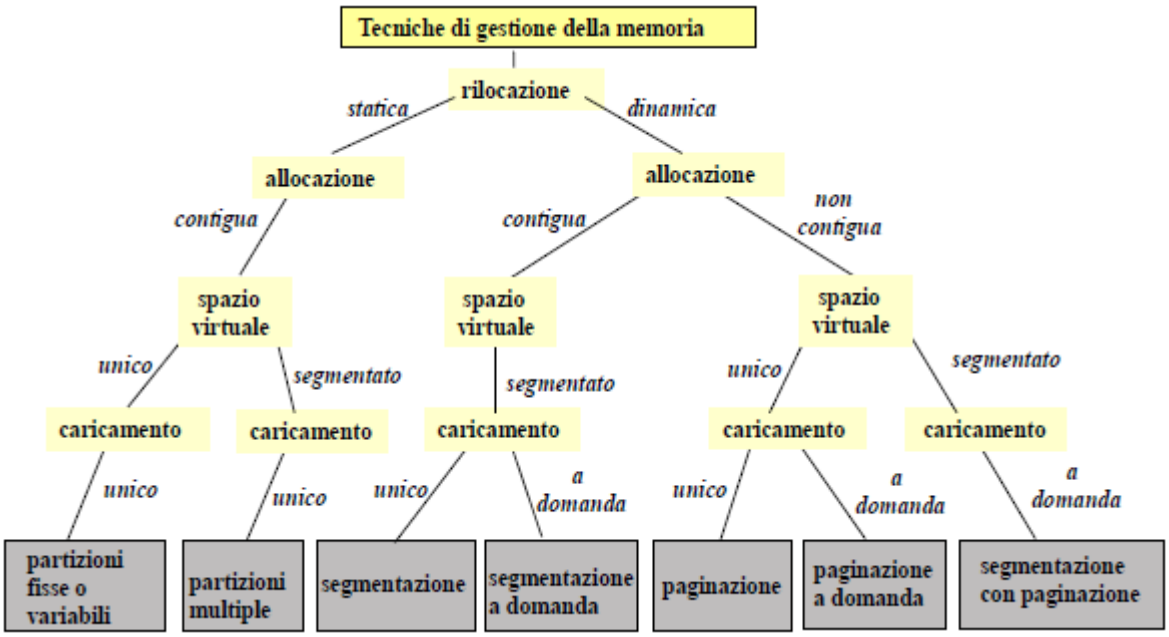
Questi parametri dipendono dall'architettura HW del processore e dalle scelte di progettazione del SO.

Rilocazione degli indirizzi, mette in corrispondenza indirizzi virtuali a indirizzi fisici, ovvero implementa una funzione, di rilocazione, che permette di calcolare a quale indirizzo fisico corrisponde un dato indirizzo logico. Si può compiere in diverse fasi:

- Compilazione, quando la locazione di memoria in cui va caricato un programma è conosciuta a priori. Se la locazione cambia il codice va ricompilato.
- Caricamento, quando la locazione non è conosciuta a priori, il compilatore genera codice rilocabile mentre il caricatore traduce gli indirizzi rilocabili in indirizzi assoluti ogni volta che il programma sarà caricato in memoria.
- Esecuzione, si utilizza quando un software ha bisogno di essere spostato durante l'esecuzione, è la più usata ma ha bisogno di hardware dedicato (MMU) perché la traduzione via software sarebbe inefficiente.

Come abbiamo detto la rilocazione può avvenire in modo statico, effettuata quindi in fase di compilazione o caricamento, a run-time gli indirizzi logici e fisici sono gli stessi e la CPU genera indirizzi fisici; la gestione è piuttosto semplice ma non permette di ricaricare il processo in un'area

di memoria differente, perché prima dello swap-out potrebbe aver prodotto delle informazioni di ritorno da chiamate di procedura riferendosi a indirizzi fisici, la soluzione è quella di ritardare la fase di rilocazione. L'altro modo è dinamico, eseguita in fase di esecuzione, gli indirizzi logici e fisici differiscono. La MMU, interfaccia CPU e memoria centrale ed effettua la rilocazione dinamica. Quando viene schedulato un processo e il dispatcher fa il context switch, carica il valore del registro base e quello limite, la MMU controlla che l'indirizzo generato dalla CPU sia minore del registro limite, se è così ci somma quello base e genera l'indirizzo fisico altrimenti lancia un'eccezione, così facendo protegge il SO e la memoria.
Allocazione della memoria fisica, può essere contigua o non contigua. Se la rilocazione è statica allora l'allocazione deve per forza essere contigua, infatti si lavora con indirizzi fisici quindi la CPU terminata un'istruzione prenderà l'indirizzo di quella dopo che è esattamente l'indirizzo fisico successivo. Se la rilocazione invece è dinamica questo vincolo non sussiste, infatti lavorando con indirizzi logici la CPU una volta terminata l'esecuzione di un'istruzione prenderà l'indirizzo logico dell'istruzione successiva, ma non è detto che questo indirizzo logico sia successivo a quello fisico su cui stavamo lavorando.
Organizzazione dello spazio di memoria virtuale, può essere unico, ovvero che il linker alloca tutti i moduli componenti un programma in indirizzi virtuali contigui, oppure segmentato, cioè non è detto che il primo indirizzo di un modulo e quello successivo sono contigui. Quando la rilocazione è statica il caricatore usa una tabella contenente gli indirizzi fisici iniziali di ciascun segmento, quando è dinamica la MMU usa i registri base e limite contenuti in una tabella con tanti elementi quanti sono i segmenti.
Caricamento dello spazio virtuale in memoria fisica, unico quando un programma e i suoi dati sono interamente caricati in memoria fisica, comporta un vincolo in quanto la memoria virtuale deve essere < o uguale a quella fisica, inoltre l'assegnazione e il rilascio della memoria avvengono solo durante la creazione e la terminazione dei processi. Questo metodo permette la rilocazione statica degli indirizzi. Se invece il caricamento avviene a domanda, il programma e i dati non sono necessariamente caricati per intero, anche qui abbiamo un vincolo, ovvero che la rilocazione sia dinamica, inoltre la memoria fisica varia nel tempo sia in quantità che in posizione, le assegnazioni e le revoche possono avvenire un numero arbitrario di volte, ma la memoria virtuale può essere maggiore di quella fisica. Se il codice e i dati necessari per l'esecuzione di un programma non vengono trovai in memoria si genere un'interruzione, e il SO la gestisce caricando gli elementi che servono. In questo modo si possono eseguire processi la cui somma delle dimensioni degli spazi virtuali supera la dimensione della memoria fisica, avendo così un migliore utilizzo della memoria. Questa gestione prevede l'esistenza di un'area di scambio in memoria secondaria, quindi la memoria virtuale = memoria fisica + memoria di scambio (swap-area). Il collegamento di una procedura è postposto fino alla fase di esecuzione. Senza il collegamento dinamico, i programmi dovrebbero contenere una copia della libreria all'interno della propria immagine, mentre col collegamento dinamico i processi condividono la stessa copia. In questo caso l'immagine eseguibile include uno stub, cioè una piccola porzione di codice che si usa per localizzare la procedura in memoria, quindi questo stub sostituisce se stesso con l'indirizzo della procedura e la esegue. Il SO si deve occupare di controllare che la procedura invocata sia in memoria o altrimenti provvedere a metterla.



Swapping

A seconda di come viene gestita la memoria, un processo durante la sua esecuzione può risiedere temporaneamente in memoria ausiliaria (swap out) per poi essere riportato in memoria centrale (swap in). La memoria ausiliaria (swap area), è un disco ad accesso veloce e diretto sufficientemente grande per memorizzare le copie delle immagini di memoria di tutti i processi. Quando lo scheduler della CPU decide quale processo eseguire, il dispatcher eventualmente ricaricare in memoria principale il processo precedentemente swappato. Se la rilocazione non è dinamica, si deve ricaricare il processo esattamente nello stesso spazio di memoria che occupava in precedenza. Il tempo di swap è quasi tutto dovuto al trasferimento, quindi direttamente proporzionale alla quantità di memoria spostata. Una variante dello swapping è roll out, roll in, che si adatta a algoritmi di scheduling basati sulla priorità; in questo caso un processo a bassa priorità è scambiato in modo che uno ad alta possa essere caricato ed eseguito. Un problema che si può presentare è quando si scarica un processo che ha I/O pendenti, quando il processo che viene caricato al suo posto va in esecuzione potrebbe darsi che l'operazione di I/O tenta di accedere alla memoria del nuovo processo (pensando che invece era quello vecchio che aveva fatto richiesta). Per risolvere questo problema possiamo: non scaricare processi con I/O pendenti o eseguire I/O solo in appositi buffer del SO.

Memoria partizionata

La memoria fisica è divisa in due parti: una per i SO residente e una destinata ad ospitare i programma utente. Esistono due tecniche di partizionamento: a partizioni fisse (spazio virtuale unico) o a partizioni multiple (spazio virtuale segmentato).

Partizioni fisse, caratterizzata da una rilocazione statica, allocazione contigua, spazio virtuale unico e caricamento tutto insieme. Quando lo spazio virtuale di un processo deve essere caricato in memoria, bisogna trovare un'area di memoria contigua sufficientemente grande per contenerlo. La memoria destinata ai processi utente è suddiviso in un numero fisso di partizioni con dimensione fissate, decise in fase di installazione del SO. Ogni partizione può ospitare un solo processo alla volta. Il SO si tiene una tabella con tutte le partizioni disponibili e quelle occupate. Quando un processo termina o viene spostato (swap out) la sua partizione diventa disponibile per altri processi. La rilocazione degli indirizzi è per forza statica, mentre l'allocazione della memoria fisica può anche essere dinamica, in questo caso si mantiene una coda di processi che si alternano nell'uso della partizione tramite lo swapping. Comporta un basso overhead ma un uso estremamente inefficiente della memoria, infatti soffre di **frammentazione interna**, ovvero si alloca più memoria di quella necessaria (infatti raramente i processi hanno le stesse dimensioni della partizione, la differenza tra i due comporta uno spreco di memoria partizionata). Inoltre un altro problema che ha è la mancanza di flessibilità, dovuta al fatto che le partizioni e le loro dimensioni sono decise in fase di installazione e quindi limitano il grado di multiprogrammazione e la dimensione massima di un processo caricabile.

Partizioni variabili, le caratteristiche (numero e dimensione) delle partizioni sono decise dinamicamente in base alle necessità, inizialmente la memoria per i processi utente costituisce un'unica partizione, poi via via verrà divisa in più parti a seconda dell'esigenze dei processi. Il SO

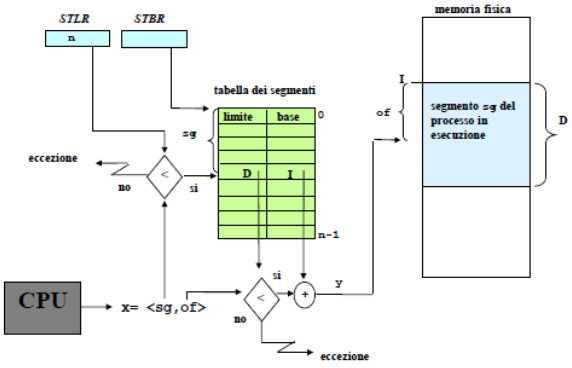
quando deve caricare un processo in memoria centrale cerca una partizione disponibile sufficientemente grande, se non c'è può aspettare che si crei oppure scorrere la lista finché non trova un processo di dimensioni giuste per le partizioni disponibili.
Quando hai più partizioni che possono soddisfare la richiesta si usano delle assegnazioni dinamiche della memoria.

- First-fit, si alloca la prima partizione abbastanza grande.
- Best-fit, si alloca la partizione più piccola sufficiente a contenerlo, comporta l'analisi di tutte le partizioni a meno che non siano ordinate per dimensione crescente. Inoltre produce partizioni inutilizzate molto piccole.
- Worst-fit, si alloca la partizione disponibile più grande. Anche in questo caso vanno analizzate tutte a meno che non siano ordinate per dimensione decrescente, produce partizioni disponibili più grandi.

Il gestore della memoria centrale si mantiene una lista aggiornata della partizioni disponibili, nella lista sono contenuti l'indirizzo iniziale e la dimensione. Questa lista viene mantenuta ordinata per dimensioni crescenti/decrescenti in caso di algoritmo best-fit o worst-fit, oppure per indirizzi crescenti per l'algoritmo first-fit. I vantaggi di questa implementazione sono la flessibilità e l'eliminazione del problema della frammentazione interna. Lo svantaggio è che soffre di **frammentazione esterna**, ovvero le partizioni disponibili sono tutte di dimensione inferiore alla quantità di memoria richiesta, sebbene la loro somma soddisferebbe la richiesta. Una soluzione potrebbe essere quella di compattare le partizioni libere, ma non è applicabile perché richiede la rilocazione dinamica, l'altra soluzione è rinunciare all'assegnazione di memoria contigua. La protezione è implementabile con supporto HW, mentre la condivisione non è realizzabile in quanto lo spazio virtuale è unico e quindi richiede allocazione contigua.
Partizioni multiple, si usa questa configurazione per permettere la condivisione di codice, infatti anche se ogni segmento è allocato in locazioni contigue, può essere allocato in maniera indipendente dagli altri segmenti del programma. I vantaggi sono appunto la condivisione di codice, la riduzione della frammentazione esterna e l'agevolazione dell'allocazione dei processi (perché si richiederanno partizioni più piccole anziché una grande).

Segmentazione

La segmentazione si prefigge come obiettivo di risolvere il problema della frammentazione esterna, come abbiamo già visto, per risolvere questo problema, necessitiamo di una rilocazione degli indirizzi dinamica. Questo permette anche una maggiore libertà nella struttura del programma, infatti adesso non deve essere più suddiviso in codice, stack e dati ma può essere visto come le unità logiche che lo compongono (programma, procedure, funzioni..). Per poter effettuare la traduzione da indirizzi logici a fisici occorre sapere a quale segmento appartiene un determinato indirizzo. Per fare questo l'indirizzo logico è diviso in due <sg, of>, sg rappresenta il numero del segmento, of è lo scostamento dall'inizio del segmento. Inoltre si tiene in memoria fisica una tabella dei segmenti contenente l'indirizzo base e la dimensione di ciascuno. STBR, contiene l'indirizzo della locazione fisica in cui risiede la tabella dei segmenti, STLR, contiene il numero di segmenti del processo. Questa gestione comporta una perdita d'efficienza, in quanto ogni volta dobbiamo accedere alla memoria 2 volte: una per la tabella dei segmenti e una per l'informazione voluta. La soluzione è che la MMU contiene dei registri associativi detti TLB, in cui sono memorizzati il numero dei segmenti e i corrispettivi valori base e limite. Ogni volta che la CPU genera un indirizzo logico la traduzione prevede la ricerca in parallelo nel TLB, se non ha successo si ricerca in memoria. Ogni registro può contenere un valore per identificare lo spazio virtuale del processo autorizzato in modo da garantire protezione. Nel PCB di ciascun processo c'è un campo che contiene due informazioni: l'indirizzo della tabella dei segmenti del processo, e il numero dei segmenti del suo spazio virtuale. Questi due valori sono usati dalla CPU per inizializzare i registri STBR e STLR. I vantaggi della segmentazione sono: la protezione, infatti si fanno ben 3 controlli: sg<= di STLR, of <= dimensioni del segmento, controllo dei diritti di accesso. Un altro vantaggio è la condivisione, infatti come abbiamo detto è l'obiettivo principale di questa gestione, si ottiene a

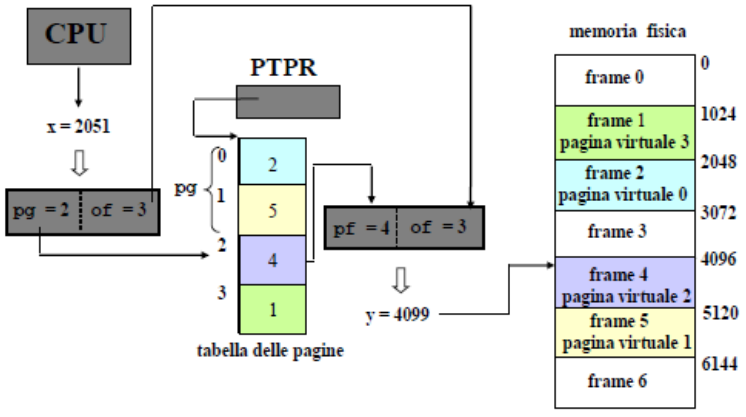


patto che i segmenti condivisi abbiano lo stesso indice negli spazi virtuali di tutti i processi che li usano.

Segmentazione a domanda, lo spazio virtuale di un processo non è totalmente caricato in memoria fisica, quindi lo swapping riguarda i singoli segmenti e non l'intero spazio virtuale. In questo modo è possibile schedulare la CPU anche per un processo che non ha segmenti in memoria centrale. La rilocazione degli indirizzi in questo caso è più complessa, infatti se si genere un indirizzo il cui segmento non è in memoria si verifica un'interruzione segment-fault. Per ogni nuovo segmento nella tabella si usa uno specifico bit di controllo (bit di presenza) posto a 1 se il segmento è presente in memoria (in questo caso i registri base e limite contengono valori significativi), posto a 0 se non è presente (se si genera un indirizzo con questo segmento c'è segment-fault). Se quando vogliamo caricare un segmento non c'è abbastanza spazio, anche dopo un'eventuale compattazione, dobbiamo swappare un segmento dello stesso processo o di altri processi. Per fare questo spostamento si usano degli algoritmi di rimpiazzamento che usano 2 ulteriori bit assegnati a ogni segmento della tabella dei segmenti, il bit di controllo M (bit di modifica), posto a 1 se è stato modificato, bit di controllo U(bit di uso), posto a 1 se è stato usato dopo essere stato caricato in memoria.

Paginazione

In questa gestione abbiamo la rilocazione dinamica ma lo spazio virtuale è unico, l'allocazione in memoria fisica può essere sia contigua che non. Per eliminare totalmente il problema della frammentazione, bisognerebbe poter allocare in memoria fisica, in locazioni non necessariamente contigue, informazioni i cui indirizzi virtuali sono contigui, con la rilocazione dinamica sarebbe possibile ma servirebbe una tabella delle corrispondenze delle stesse dimensioni della memoria virtuale. Il compromesso è che le locazioni dello spazio virtuale sono allocate in gruppi di dimensioni fisse contigue allocati indipendentemente l'uno dall'altro. Lo spazio virtuale è suddiviso in blocchi di indirizzi virtuali di dimensioni fisse, **pagine.** Mentre lo spazio fisico è suddiviso in blocchi di indirizzi fisici delle stesse dimensioni, **frame**. Per tradurre un indirizzo virtuale in fisico serve una tabella delle corrispondenze (tabella delle pagine), ogni pagina viene allocata in un frame e pagine consecutive non devono essere necessariamente allocati in frame consecutivi. La tabella delle pagine è posseduta da ogni processo e risiede in memoria centrale. Inoltre esistono 2 registri PTPR, contenente l'indirizzo iniziale della memoria in cui c'è la tabella delle pagine, e PTLR, che contiene il numero delle pagine del processo. Se manca il PTLR le tabelle delle pagine di tutti i processi hanno la stessa lunghezza, quindi per protezioni della memoria si associa un bit di validità, per essere sicuri che la pagina è nello spazio degli indirizzi logici del processo che la sta richiedendo. Per tradurre gli indirizzi a run-time generati dalla CPU bisogna determinare la pagina a cui un indirizzo appartiene. Per far questo, un indirizzo logico è diviso in due componenti <pg, of>, dove pg è il numero di pagina (nonché il quoziente della divisione tra l'indirizzo e la dimensione delle pagine), e of che è lo scostamento dall'inizio della pagina (resto della divisione indirizzo per dimensione). Se la dimensione è una potenza di 2, (2^y) si possono prendere direttamente i bit più e meno significativi, nello specifico: of rappresenta gli y bit meno significativi di x (combinato con l'indirizzo base del frame diventerà l'indirizzo fisico), pg è costituito dai restanti bit di x (sarà l'indice nella tabella delle pagine). Questo metodo di traduzione però comporta una perdita d'efficienza in quando ogni volta che viene generato un indirizzo, dobbiamo accedere alla memoria 2 volte: una per la tabella delle pagine e una per l'informazione che vogliamo. La soluzione è nei registri associativi contenuti nella MMU, cioè i TLB. Per la traduzione quindi attuiamo una ricerca in parallelo nei TLB, che contengono una parte di pagine con relativo indirizzo iniziale del frame, e volendo un valore per identificare lo spazio virtuale del processo autorizzato, per garantire protezione. Nel descrittore di ogni processo si aggiungono quindi 2 campi: l'indirizzo di memoria della tabella delle pagine e il numero di pagine del suo spazio virtuale. Questi due valori, durante il context switch, si usano per inizializzare i registri PTPR e PTLR. Il gestore della memoria fisica mantiene una tabella dei frame, dove sono indicati i frame liberi e quelli occupati, in questo caso è presente l'id del processo a cui è allocato e l'indice della sua pagina virtuale ospitata. Quando un



processo deve essere caricato in memoria si richiedono tanti frame quante pagine ha, se non c'è abbastanza spazio si fa lo swap out di altri processi fino a crearglielo. La dimensione delle pagine è molto importante, infatti al suo diminuire aumenta la dimensione della tabella delle pagine, ma al suo aumentare aumenta la frammentazione interna. Ogni elemento della tabella può avere anche dei bit di validità presenza e di controllo, per garantire la protezione. La condivisione in questo caso è problematica, infatti una pagina non individua un elemento logico del programma e inoltre si dovrebbe comunque rispettare il vincolo che le pagine condivise dovrebbero occupare le stesse posizioni nei rispettivi spazi virtuali.
Paginazione a domanda, caratterizzata da un caricamento a domanda, quindi un processo non è caricato per intero. Vengono usati gli stessi 3 bit di controllo della segmentazione: P, M e U. Quando si crea un processo il suo spazio virtuale è tutto nella swap area, quindi la tabella delle pagine inizialmente ha tutti i bit P a 0, con i campi destinati all'indirizzo del frame contenenti indirizzi della swap area. Quando si tenta di tradurre un indirizzo con pagina non presente si verifica un page-fault, la routine carica la pagina e si riesegue la traduzione, nel caso in cui non c'è abbastanza posto per il caricamento si invoca un algoritmo di rimpiazzamento. Il rimpiazzamento di una pagina prevede: cercare nella swap area un posto disponibile, nel quale inserire il contenuto della pagina da rimpiazzare, si trasferisce il contenuto a tale indirizzo dalla tabella dei frame si ricava l'identificativo del processo a cui abbiamo spostato la pagina e quindi li aggiorniamo la tabella delle pagine (per mettere il bit di presenza a 0), aggiorno poi la tabella delle pagine fisiche. Il rimpiazzamento di pagine risulta quindi più semplice rispetto alla segmentazione, perché le pagine hanno tutte le stesse dimensioni (non si dovranno mai scaricare 2 pagine per farne posto a 1, cosa che invece succedeva coi segmenti). Inoltre non c'è bisogno di trasferire su disco la pagina scelta per il rimpiazzamento se il relativo bit di modifica è posto a 0. Possiamo inoltre bloccare lo scaricamento di alcune pagine, inserendo un bit di controllo, lock, che ne impedisce appunto lo scaricamento. La scelta della pagina da rimpiazzare è molto importante, infatti bisogna limitare i page-fault per l'efficienza della gestione della memoria e per il sistema (thrashing è lo stato in cui la CPU si occupa solo di spostamenti dalla swap area). Se chiamiamo p la probabilità che la pagina manchi (quindi 1-p è la probabilità che la pagina sia presente), il **tempo di accesso effettivo**= (1-p) x tempo di accesso alla memoria + p x tempo di gestione dei page-fault. A sua volta la gestione dei page-fault è la somma di: servizio di eccezione + eventuale scaricamento di una pagina + caricamento della pagina richiesta + riavvio del processo. **Algoritmi di rimpiazzamento**, si usano sia per il segment che il page fault e sono i seguenti:

- Ottimo, si sceglie una pagina che sicuramente non sarà mai più usata in futuro o comunque usata tra più tempo. E' irrealizzabile, infatti non si può conoscere il carico futuro, ma è un buon metro di paragone.
- FIFO, si scarica la prima pagina caricata in ordine di tempo. La realizzazione è molto semplice, infatti la tabella è organizzata in modo da formare una coda FIFO e si scarica l'ultimo elemento della coda, l'inconveniente è che non sempre è la pagina che non sarà necessaria nell'immediato futuro. Inoltre soffre dell'anomalia di Belady, ovvero più frame abbiamo e più aumentano i page-fault (questo si verifica nel passaggio da 3 frame a 4).

Si decide allora di prendere questa decisione basandosi sull'immediato passato, ovvero col working set (insieme delle pagine virtuali riferite dal processo in un certo lasso di tempo) e al principio di località dei riferimenti (l'insieme di lavoro cambia in maniera graduale).

- LRU, si scarica la pagina non acceduta da più tempo. Si può implementare in due modi:
    - Con contatori, la CPU ha un registro contatore che si incrementa ad ogni accesso, ogni frame ha un campo in cui copiare il contenuto di questo contatore, si scarica la pagina con il contatore più basso. Di solito si usa il clock di sistema come contatore.
    - Con stack, si mantiene uno stack con i numeri di pagina in una lista a doppio collegamento, quando si riferisce una pagina si mette in cima, si scarica quindi la pagina il cui numero è in fondo allo stack.

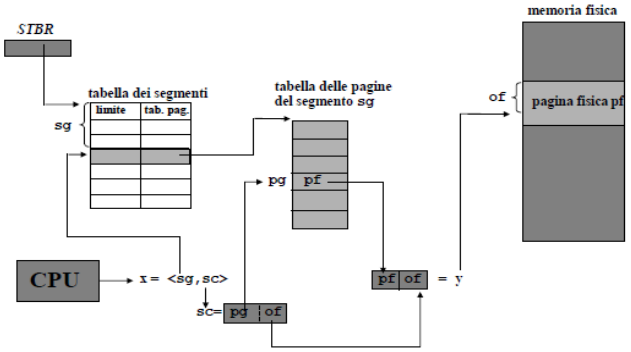
  Questo algoritmo non soffre dell'anomalia di Belady, sperimentalmente è l'algoritmo migliore ma la realizzazione è costosa sia come HW che come SW. Ricorriamo quindi a delle sue approssimazioni:
- Second-chance (clock algorithm), ogni frame ha un bit di uso U, quindi si dividono in due gruppi, U=0, non usati recentemente, U=1, usati recentemente. Si sceglie con tecnica FIFO un frame per cui U=0, altrimenti sempre con tecnica FIFO uno con U=1, si gestisce la tabella come un array circolare, quindi si mantiene nella variabile vittima l'indice della pagina fisica successiva a quella che è stata rimpiazzata per ultima, quando si verifica un page-fault, se il frame il cui indice è in vittima ha U=0, si rimpiazza, altrimenti lo poniamo a 0, si incrementa vittima passando all'elemento successivo. Nel

peggiore dei casi facciamo un giro completo dell'array circolare, quindi scorriamo tutta la tabella dei frame.

- Second-chance raffinato, si usa anche il bit di modifica quindi, se c'è un frame che ha 0 a entrambi i bit (uso e modifica), scarichiamo quello, altrimenti preferiamo scaricare quelli che comunque hanno bit di modifica a zero per non dover copiarla in memoria secondaria.

Per ridurre il numero dei page-fault possiamo adottare diverse strategie, quali: i criteri di rimpiazzamento, ovvero globale (se la pagina si sceglie indipendentemente dal processo a cui appartiene), locale (se si sceglie solo tra quelle del processo che ha generato il page-fault); si usa una preallocazione di alcune pagine di un processo appena viene creato, può essere un'allocazione omogenea (a prescindere dalla dimensione o priorità si allocano sempre lo stesso numero di pagine), oppure proporzionale (più è grande o importante più pagine si preallocano); oppure si mantengono sempre dei frame liberi per velocizzare il page-fault. Un'altra cosa da tenere sotto controllo per il buon funzionamento del sistema è il thrashing, infatti quando accade abbassa l'uso della CPU, quindi lo scheduler aumenta il grado di multiprogrammazione, il che provoca un peggioramento del thrashing e quindi un ulteriore rallentamento della CPU, entrando in un circolo vizioso dannoso per il sistema. Questo fenomeno si verifica se la somma delle dimensioni dei working set dentro cui si muovono i processi, è superiore alla dimensione della memoria fisica. Per controllarlo si fa una valutazione approssimativa delle pagine virtuali che caratterizzato il working set del processo, e si fissa un tasso accettabile di mancanza pagine. Per valutare le pagine del working set, mettiamo per ciascuna pagina n bit di uso, uno per ciascuno degli ultimi n intervalli di tempo. Ad intervalli regolai si controllano i bit di suo, se quando c'è un page-fault tutte le pagine appartengono al working set, vuol dire che l'insieme si sta espandendo e quindi occorre dare più frame al processo, però se la memoria è satura invece di rimpiazzare una pagina, si scaricano direttamente tutte. Per stabilire un tasso accettabile invece, si procede dicendo: se il tasso è troppo basso è possibile che al processo siano stati allocati troppi frame, se è troppo alto ne ha bisogno di altri.

Segmentazione con paginazione



Si usa uno spazio virtuale segmentato e si alloca la memoria in modo non contiguo. La tabella dei segmenti, invece di contenere l'indirizzo base di un segmento, contiene l'indirizzo base della tabella delle pagine per quel segmento. La traduzione degli indirizzi può generare vari tipi di interruzione: segment-fault, se il bit di presenza P nel descrittore del segmento è a 0 (la tabella delle pagine non è in memoria), oppure page-fault, se il bit di presenza P nel descrittore della pagina è a 0 (pagina non in memoria.

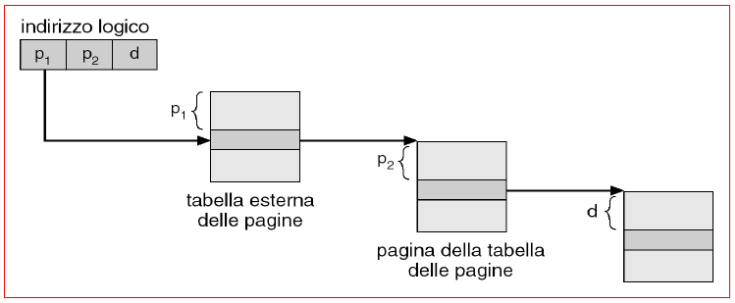
Gestione degli spazi virtuali

Il context switch con la rilocazione dinamica, è più costoso, infatti per cambiare funzione di rilocazione bisogna commutare le informazioni presenti nella MMU e invalidare i registri associativi TLB. Per alleggerire quindi il context switch sono stati introdotti i thread. I meccanismi di rilocazione creano inoltre diversi problemi riguardanti le seguenti funzioni:

- Condivisione delle informazioni, la porzione condivisa deve essere allocata nelle stesse posizioni negli spazi virtuali di tutti i processi interessati.
- Spazio virtuali degli indirizzi generati da funzioni del SO, ovvero quando si invoca una funzione del SO la CPU genera degli indirizzi di istruzioni, questi indirizzi quando c'è rilocazione dinamica si riferiscono ad uno spazio virtuale riservato al SO. Questo genera un appesantimento sulle chiamate di sistema (si commuta lo stato di esecuzione, una system call commuta le informazioni nella MMU, si invalidano i registri TLB e si cambiano le tabelle delle pagine/segmenti aumentando i miss). Inoltre si appesantisce il passaggio dei parametri quando un processo chiamante deve passare un proprio indirizzo come parametro alla funzione chiamata, questo perché i due indirizzi risiedono in spazi virtuali separati. La soluzione si è potuta attuare nei sistemi che permettono una memoria virtuale maggiore di quella fisica, infatti in questi sistemi è possibile allocare il SO in tutti gli spazi virtuali. In questo modo lo spazio virtuale risulta suddiviso in due partizioni: quella superiore che ospita il SO e quella inferiore per codice e dati del processo. In questo modo si risolve il problema

dell'appesantimento ma la tabella delle pagine e dei segmenti è enorme. La soluzione è strutturare la tabella delle pagine oppure suddividere la tabella dei segmenti.

Paginazione a due livelli: La tabella delle pagine, composta da 2^{20} elementi di 4 byte, è suddivisa in 2^{10} porzioni consecutive, ciascuna di 2^{10} elementi di 4 byte (ogni porzione è 2^{10} kb) le porzioni costituiscono le tabelle delle pagine di 2° livello e possono essere allocate in memoria fisica in modo contiguo e solo se necessario. Quando il processo è in esecuzione si mantiene in memoria fisica una tabella di 1° livello (page directory) con 2^{10} elementi, uno per ogni porzione. Il suo indirizzo è mantenuto nel registro PDAR e ogni suo elemento mantiene un bit di presenza per la tabella di 2° livello relativa e, eventualmente, il suo indirizzo fisico. Perciò l'indice di pagina di 2^{10} bit è diviso così: i 10 bit più significativi sono il numero di pagina (DR) usati per accedere alla page directory; i 10 bit rimanenti sono lo scostamento di pagina (STP).



Schema di traduzione degli indirizzi:

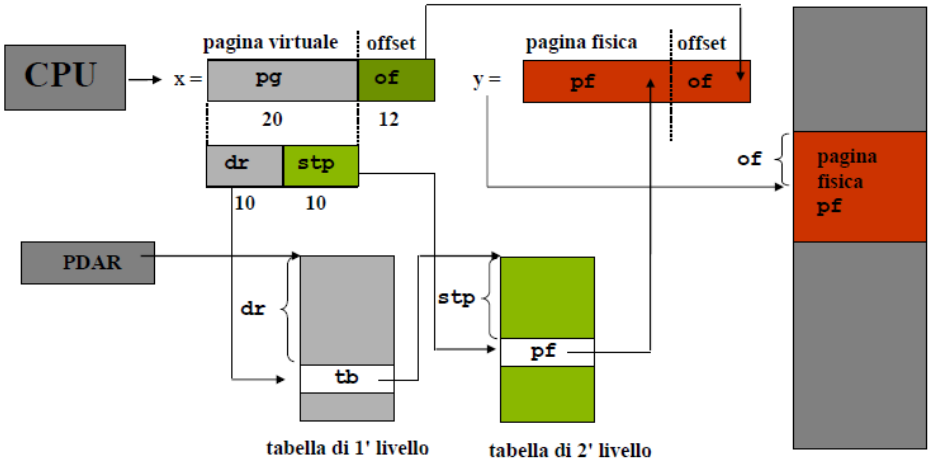
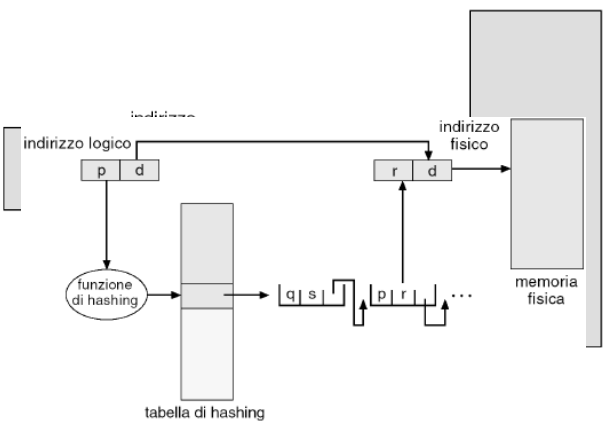


Tabella delle pagine di tipo hash: il numero di pagina virtuale è usato come argomento di una funzione hash. Ogni elemento della tabella hash contiene una lista concatenata di elementi, ciascuno dei quali è un descrittore di pagina, che hanno lo stesso valore della funzione hash. Il numero della pagina virtuale viene confrontati con gli elementi della lista, se si trova il valore corrispondente, si estrae l'indice del frame.

Tabella delle pagine a gruppi: è una tecnica adatta a spazi di indirizzamento a 64 bit: ciascun elemento della tabella delle pagine contiene i riferimenti a frame corrispondenti ad un gruppo di pagine virtuali. Poiché lo spazio degli indirizzi di molti programmi non è arbitrariamente sparso su frame isolati ma distribuito per "raffiche di riferimenti", le tabelle delle pagine a gruppi sfruttano bene questa caratteristica.

Tabella delle pagine invertita: si usa una sola tabella delle pagine per tutto il sistema, che ha un elemento per ogni pagina fisica della memoria centrale. Ciascun elemento contiene l'indirizzo virtuale della pagina logica memorizzata in quella posizione di memoria fisica e l'identificativo del processo a cui appartiene la pagina. Questo schema consente di ridurre la quantità di memoria centrale necessaria per memorizzare ogni tabella, ma aumenta il tempo di ricerca quando si fa



riferimento ad una pagina. Si può usare un hashing per limitare la ricerca ad uno o al più a pochi elementi nella tabella delle pagine.

Suddivisione della tabella dei segmenti: nei sistemi con segmentazione, la tabella dei segmenti di un processo viene suddivisa in due tabelle distinte: una per lo spazio di memoria virtuale superiore del processo (che ospita il so) che non viene mai commutata; e l'altra per tradurre gli indirizzi virtuali del processo stesso. Essa è un segmento dello spazio di sistema (perché è una struttura dati del so) e può essere paginata (nei sistemi segmentati con paginazione).

File system

I dispositivi di memoria secondaria garantiscono ai processi l'accesso a grandi insiemi di dati e a informazioni che sopravvivono alla terminazione dei processi o alla mancanza di alimentazione elettrica. Dal lato hardware sono quindi necessarie periferiche con grande capacità e persistenza; dal lato software astrazioni e meccanismi che consentano la rappresentazione, l'archiviazione e l'accesso ai dati immagazzinati nella memoria secondaria. Questo compito è svolto da una componente del SO detta **file system:** i processi e gli utenti vedono la memoria secondaria attraverso la struttura astratta rappresentata dal esso. La sua interfaccia è costituita dalle system call, che consentono agli utenti di usare la memoria secondaria (accedere e archiviare le informazioni), mentre la sua realizzazione è costituita dalle strutture dati e dagli algoritmi per la gestione della memoria secondaria da parte del SO.

INTERFACCIA DEL FILE SYTEM

Il file system

Il file system realizza i concetti astratti di **file** (unità logica di memorizzazione), **directory** (insieme di file e directory) e **partizione** (insieme di file associato ad un dispositivo fisico o a parte di esso), le cui caratteristiche sono indipendenti dal dispositivo fisico usato. L'interfaccia verso l'utente è formata da file, directory e relative system call, più comandi/interfaccia grafica realizzati richiamando quest'ultime.

I file

Il file è un contenitore per un insieme di informazioni correlate. Dal punto di vista del so, il file è l'unità di memoria logica formata da un insieme di informazioni correlate. Dal punto di vista dell'utente è la più piccola porzione logica di memoria secondaria: i dati si possono scrivere in memoria soltanto all'interno di essi. Gli scopi dei file sono: memorizzare grandi quantità di dati, supportare la creazione di dati permanenti e fornire uno strumento semplice per consentire a più processi di comunicare e/o condividere informazioni. Un file è formato da una sequenza di record logici, il cui significato è definito dal creatore o dall'utente. I record logici sono impacchettati in blocchi fisici (trattati in maniera unitaria dall'i/o sul disco), tutti della stessa dimensione. La struttura del file dipende dal tipo, per esempio: i file di testo sono sequenze di caratteri organizzati in righe e pagine, i programmi sorgente sono sequenze di procedure e funzioni, ciascuna composta da dichiarazioni e istruzioni; i programmi oggetto sono sequenze di byte organizzate in blocchi comprensibili al linker e gli eseguibili sono sequenze di sezioni di codice che il caricatore può trasferire in memoria. Tale struttura deve corrispondere alle aspettative del programma che dovrà manipolarlo. Ogni file ha i seguenti attributi, memorizzati in un descrittore del file.

- Nome, un nome simbolico che serve per riferirlo.
- Identificatore, etichetta unica che identifica il file all'interno del file system (lo usa il sistema).
- Tipo, necessario per sistemi che gestiscono tipi di file diversi; a volte è un suffisso del nome (estensione).
- Locazione, puntatore al dispositivo e alla posizione del file in esso.
- Dimensione, dimensione corrente del file.
- Protezione, informazioni per il controllo delle operazioni sul file (lettura, scrittura, esecuzione).
- Ora, data: relative a creazione, ultima modifica o ultimo uso (utili per protezione e controllo).
- Proprietario, in un so multiutente indica l'utente proprietario.

Il descrittore di file (FCB) deve essere persistente ed è quindi mantenuto in memoria secondaria. I FCB dei file appartenenti a una directory, organizzati opportunamente, costituiscono la struttura (o tabella) della directory.

Operazioni sui file, il SO deve consentire di effettuare operazioni online sui file. Le operazioni base sono 6:

- Creazione: comporta il reperimento dello spazio fisico nel file system e la creazione del relativo elemento nella directory.
- Scrittura: comporta la gestione del puntatore alla locazione interna al file in cui deve avvenire la successiva lettura/scrittura.
- Lettura: come la scrittura.
- Riposizionamento (seek) in un file: comporta l'assegnazione di un nuovo valore al puntatore interno al file.
- Cancellazione: comporta il rilascio dello spazio fisico occupato ed eventualmente l'eliminazione dell'elemento relativo della tabella.
- Troncamento: comporta la modifica della lunghezza del file.

Altre operazioni si possono ottenere combinando. Le tipiche system call sono: create, delete, open, close, read, write, append, seek, get attributes, set attributes, rename.
La maggior parte delle operazioni sui file richiedono la ricerca nelle tabelle delle directory (che stanno sul disco) dell'elemento associato al file. Per migliorare l'efficienza, il SO mantiene in memoria una **tabella dei file aperti** in cui vengono inseriti gli elementi delle directory relativi ai file che sono attualmente n uso, e copia temporaneamente porzioni dei file in memoria principale. A questo scopo si usano due operazioni: **open(Fi)** eseguita sempre per prima, cerca nelle tabelle delle directory l'elemento corrispondente a Fi e ne trasferisce il contenuto nella tabella dei file aperti; eventualmente copia (parzialmente) il file in memoria principale; e **close(Fi)** eseguita per ultima, sposta il contenuto dell'elemento corrispondente a Fi dalla tabella dei file aperti alla appropriata tabella delle directory del disco; se è il caso, copia il file in memoria secondaria. In un ambiente in cui più processi possono aprire un file simultaneamente, ogni processo ha anche una propria tabella dei file aperti.

Metodi di accesso

I metodi di accesso stabiliscono le modalità con cui i processi possono leggere o scrivere i file. Ciascun metodo presuppone un'organizzazione interna dei file che a questo livello di astrazione sono visti come sequenze di record logici numerati. Tali metodi sono indipendenti dal tipo di dispositivo e dalla tecnica di allocazione dei blocchi di memoria secondaria. I più diffusi sono:

- Accesso sequenziale: si vede ogni file organizzato come una sequenza di record su cui è stabilita una relazione d'ordine. Lettura e scrittura avvengono secondo questo ordine, quindi, per accedere all'i-esimo record bisogna prima accedere ai precedenti (i-1). La posizione del prossimo elemento è mantenuta in un puntatore, aggiornato automaticamente, ma può essere modificato tramite seek(). Le operazioni di lettura/scrittura sono: readnext(F, &V), che assegna il valore del prossimo record logico del file F alla variabile V e aggiorna il puntatore; e writenext(F, V) che scrive nel prossimo record del file F il valore della variabile V e aggiorna il puntatore. E' il modello che fa riferimento ai nastri magnetici e ai cd-rom.
- Accesso diretto: i record logici di cui è composto un file sono accessibili autonomamente e in un ordine qualsiasi tramite un numero di blocco relativo, per questo non è necessario mantenere un puntatore o accedere a tutti i record che precedono quello a cui vogliamo effettivamente accedere. Le operazioni sono: readd(F, N, &V) che legge il record logico Rn dal file F, e salva il valore letto nella variabile V, e writed(F, N, V) che scrive il valore della variabile V nel record Rn del file F. E' il modello che fa riferimento ai dischi.
- Accesso tramite indice: ogni record ha una chiave di accesso e ad ogni file è associata una tabella "indice" che ha un elemento per ogni chiave con associato il riferimento al record corrispondente. Quindi per accedere, si fa una ricerca associativa nell'indice, e non serve neanche in questo caso mantenere un puntatore. Le operazioni sono: readk(F, key, &V) che legge il record di chiave key, dal file F, nella variabile V; e writek(F, key, V) che scrive il valore della variabile V nel record di chiave key del file F.

File mappati in memoria: le tecniche di memoria virtuale possono essere utilizzate per equiparare l'i/o dei file dal disco all'accesso alla memoria centrale. Mappare un file in memoria significa creare un'associazione (tramite una system call map(file, indirizzo)) tra un file, o una sua porzione, e una sezione dello spazio di indirizzamento virtuale di un processo. L'associazione indica al sistema di memoria virtuale che la copia su

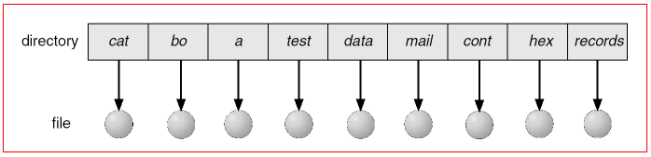
disco delle pagine virtuali del processo su cui è mappato il file sono i corrispondenti blocchi del file su disco e non l'area di swap del processo; quindi tutte le modifiche effettuate ad indirizzi di memoria virtuale associati ai file, verranno riportate sul file dal meccanismo della memoria virtuale. Normalmente un accesso ad un file su disco richiede l'invocazione di una system call, l'uso di un buffer intermedio e l'accesso al disco. Mentre coi file mappati in memoria si ha una maggiore efficienza delle operazioni di i/o: è possibile caricare in memoria solo le parti del file che sono effettivamente usate ad un dato istante; la sezione di memoria mappata su cui si opera sarà a sua volta letta o scritta sul file una pagina alla volta, e solo per le parti effettivamente usate; l'accesso alle pagine del file non ancora caricate avverrà allo stesso modo di quelle nella swap area (tramite preventivo caricamento in memoria). Quando la memoria scarseggia, le pagine che mappano un file verranno salvate automaticamente, così come le pagine dei programmi vengono scritte nell'area di swap, il tutto in modo trasparente al processo. Si ha anche una notevole semplificazione delle operazioni di i/o: i dati da trasferire potranno essere acceduti direttamente nella sezione di memoria mappata (senza usare dei buffer intermedi). Si ha possibilità di condivisione: i processi possono mappare lo stesso file in modo concorrente per consentire la condivisione dei dati. Le dimensioni dei file manipolati saranno inoltre limitate solo dallo spazio di indirizzi virtuali disponibile.
File speciali: sono inclusi nella gerarchia delle directory come qualsiasi altro file, ma in realtà corrispondono a dispositivi di i/o. Possono essere a blocchi (dischi), o a caratteri (tastiere video a caratteri etc). Questo stratagemma permette di fare riferimento ad un dispositivo utilizzando un path name.

Directory
Le directory sono astrazioni che consentono di raggruppare più file. Ad ognuna di esse è associato un descrittore che mantiene i valori degli attributi relativi. I descrittori di tutti i file sono collegati alla directory a cui appartengono e sono organizzati nella struttura (o tabella) della directory. Le operazioni fondamentali sono:

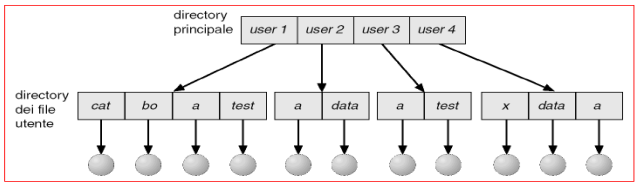
- Ricerca di un file: scorrere la directory alla ricerca dell'elemento associato ad un file.
- Creazione di un file: aggiungere nuovi elementi alla directory.
- Cancellazione di un file: rimuovere elementi dalla directory.
- Elencazione di una directory: elencare i file di una directory e il contenuto dei suoi elementi.
- Ridenominazione di un file
- Attraversamento del file system: navigazione attraverso la struttura logica del file system.

Organizzazione logica delle directory: con l'organizzazione logica si vuole avere: efficienza nella ricerca, flessibilità nel meccanismo di naming dei file (due utenti possono avere due file con lo stesso nome e lo stesso file può avere nomi diversi) e possibilità di raggruppare logicamente i file.

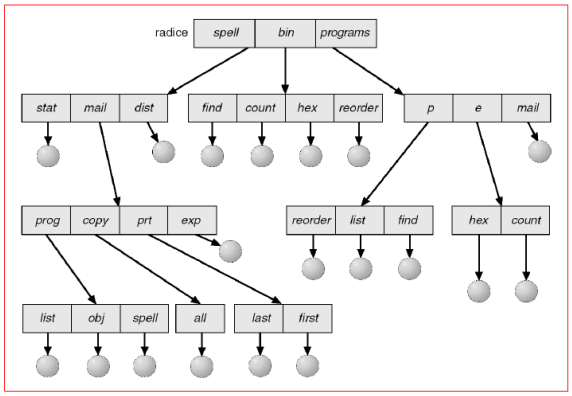
- Directory a singolo livello: è l'approccio più semplice, si ha una sola directory per tutti i file di tutti gli utenti. Si hanno però limiti notevoli all'aumentare del numero dei file o nei sistemi multiutente perché tutti i file devono avere nomi diversi e non si possono raggruppare logicamente.



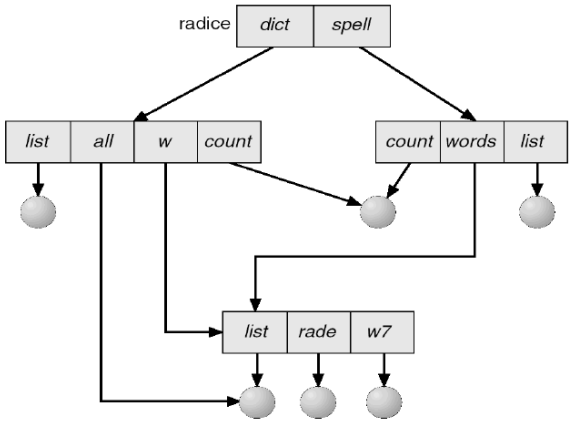
- Directory a due livelli: Ogni utente ha una sua directory separata. Utenti diversi possono avere file con lo stesso nome. E' efficiente nella ricerca ma è necessario specificare un percorso per individuare un file; non c'è capacità di raggruppamento.



- Directory con struttura ad albero: con questa organizzazione si ha ricerca efficiente e capacità di raggruppamento. Introduce il concetto di directory di lavoro corrente (PWD). Il nome di un file/directory è il suo pathname, ovvero il cammino che si deve percorrere per raggiungerlo. Esso può essere assoluto (dalla radice) o relativo (dal PWD). Le operazioni vengono eseguite nella directory corrente.

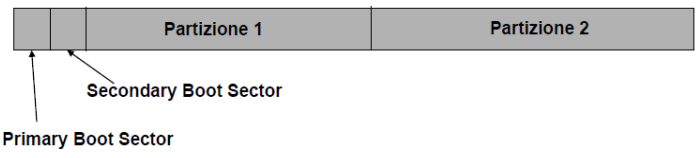


- Directory con struttura a grafo aciclico: si aggiungono archi che permettono di vedere lo stesso file da diverse directory (DAG). In questo modo si possono condividere directory e file; avere 2 o più pathname per lo stesso file, ma non possono esserci più file con lo stesso pathname. Un modo per realizzare la condivisione è usare collegamenti (simbolici o effettivi): un esempio è l'aliasing, ovvero dare più nomi diversi a uno stesso file. Si hanno però alcune complicazioni: per non peggiorare l'efficienza, quando si esplora il file system non si dovrebbe prendere più volte in considerazione una stessa porzione. Si deve inoltre stabilire quando recuperare la memoria in caso si cancelli un file condiviso; infatti se lo si fa subito si rischia di perdere collegamenti pendenti. Si può allora creare un elenco di collegamenti in modo da cancellarli tutti (esso può però prendere molto spazio); oppure utilizzare un contatore dei riferimenti ad un file e si recupera la memoria solo quando il contatore è azzerato. È anche necessario garantire l'assenza di cicli: si possono permettere scollegamenti solo a file e non a sottodirectory, oppure ogni volta che si aggiunge un collegamento si utilizza un algoritmo di controllo dei cicli. Il contatore dei riferimenti potrebbe non annullarsi per un'anomalia dovuta alla possibilità di autoriferimento nella struttura della directory: si usa quindi un algoritmo di garbage collection.



Partizionamento di dischi:

Un disco potrebbe essere usato interamente per un unico file system. Per usarlo anche per altri scopi o per più file system, lo si suddivide in partizioni, ognuna delle quali può contenere un solo file system. Quelle che non ne contengono uno sono prive di struttura logica (raw disk) e sono usate a volte come swap area. Altre partizioni possono contenere un insieme di blocchi che si carica in memoria centrale come un file immagine la cui esecuzione inizia ad una locazione prefissata. Il settore 0 del disco (primary boot sector) contiene delle partizioni ed il master boot record che individua una partizione attiva (contiene il so). Il primo settore di una partizione attiva (secondary boot sector) contiene il codice di boot del so. Le altre partizioni vengono montate come ulteriori file system. In alcuni sistemi le partizioni possono essere più grandi dei dischi così da raggruppare più dischi in un'unica struttura logica: una partizione si può quindi pensare come un disco virtuale. Ogni partizione che contiene un file system contiene anche un indice della partizione o directory del dispositivo che registra le informazioni sui file presenti nel file system.



Montaggio di un file system

Un file system deve essere montato per essere usato. Il montaggio può essere effettuato automaticamente dal SO o esplicitamente dall'utente specificando il nome della partizione, il punto di montaggio e il tipo di partizione.

Condivisione di file
La condivisione risulta molto utile nei sistemi multiutente ma richiede meccanismi di protezione e l'uso di ulteriori attributi dei file/directory: Proprietario (ha maggiore controllo) e gruppo (sottoinsieme di utente che condividono l'accesso al file).

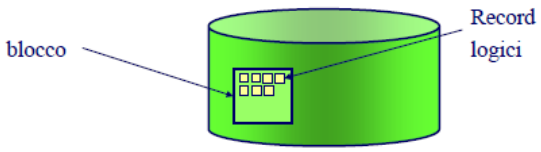
Protezione di file
I meccanismi di protezione hanno 2 ruoli: la rappresentazione delle politiche, cioè la realizzazione delle strutture dati che contengono la specifica dei vincoli di accesso alle risorse; e il controllo degli accessi. Il proprietario/creatore del file deve essere in grado di controllare cosa si può fare e chi può farlo. L'accesso quindi, viene permesso o negato in base al tipo di accesso (autorizzazione) e all'identificativo dell'utente (autenticazione). Le operazioni base sono lettura, scrittura, esecuzione, cancellazione, aggiunta in coda al file e elencazione. Quelle di livello superiore, come copia o ridenominazione sono effettuate tramite le system call. Le politiche di protezione delle risorse possono essere specificate in una matrice di protezione in cui ogni riga corrisponde a un utente, e ogni colonna a una risorsa; all'interno delle caselle ci sono i permessi. Questa matrice non può però essere implementata perché costerebbe troppo. Si usano quindi le seguenti strutture:

- Liste di capability (C-List): per ogni processo P il sistema costruisce una lista di permessi riferiti alle risorse alle quali l'utente a cui appartiene P può accedere (riga della matrice). E' più efficiente della ACL nel controllo degli accessi (in quanto la c-list fa parte delle informazioni locali ad ogni processo quindi la ricerca dei permessi è locale). Però le operazioni come la revoca di tutti i diritti associati ad una risorsa sono costose perché comporta ricerca e modifica di tutte le c-list.

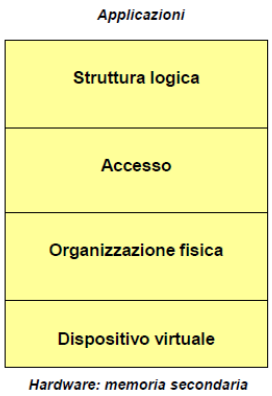
- Liste di controllo degli accessi (ACL): ad ogni risorsa si associa una ACL che per ogni utente specifica le operazioni che quell'utente è autorizzato a fare su di essa (colonna della matrice). Ad ogni richiesta di accesso il SO effettua i controlli necessari. Questo meccanismo è flessibile ma si hanno problemi con liste di utenti non note o che cambiano dinamicamente e i descrittori delle risorse devono essere di dimensioni variabili. In pratica si usa una sua versione condensata.

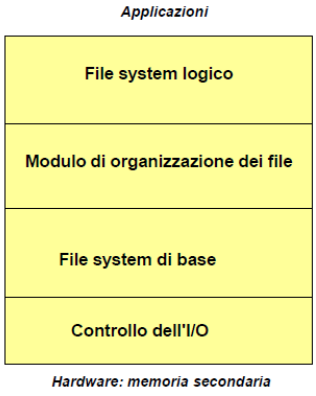
IMPLEMENTAZIONE DEL FILE SYSTEM

Ogni dispositivo di memorizzazione secondaria viene partizionato in **blocchi** (unità di trasferimento nelle operazioni di I/O da/verso il dispositivo) di eguali dimensioni. I processi vedono ogni file come un insieme di **record logici** (unità di trasferimento nelle operazioni di accesso al file). Ogni blocco contiene una sequenza di record logici contigui e la sua dimensione è molto maggiore rispetto a quella del singolo record. Uno dei compiti del file system è stabilire una corrispondenza tra record logici e blocchi.



Struttura del file system
Il file system risiede in un'unità di memorizzazione secondaria ed è opportunatamente strutturato per facilitare lo sviluppo, la manutenzione ed il riuso. Solitamente è organizzato a livelli, ciascuno dei quali si serve delle funzioni dei livelli inferiori per crearne di nuove usate ai livelli superiori. In questo modo si riescono anche a dividere quelli dipendenti dall'hardware da quelli indipendenti. Solitamente sono 4:





La struttura logica fornisce ai processi e agli utenti una visione astratta delle informazioni presenti sul disco che prescinde dalle caratteristiche del dispositivo e dalle tecniche di allocazione e di accesso alle informazioni adottate dal sistema. Comprende le strutture dati per la gestione del file system (struttura delle directory e FCB), ma non il contenuto vero e proprio dei file. Mette a disposizione le operazioni elementari su file e directory tramite system call. L'**accesso** definisce e realizza le operazioni elementari conformemente ai metodi di accesso dei file, di cui conosce il contenuto e controlla l'accesso. **L'organizzazione fisica** realizza i meccanismi di allocazione dei file che allocano i record logici nei blocchi fisici, traduce indirizzi di record logici in indirizzi di bocchi fisici, mantiene la lista dei blocchi fisici liberi, invia comandi all'apposito driver di dispositivo per leggere/scrivere blocchi fisici dal/nel disco, gestisce i buffer di memoria per il trasferimento dei dati e le cache per i metadati del file system usati più frequentemente, e gestisce l'allocazione delle strutture dati necessarie al sistema per la rappresentazione e le realizzazione di file e directory. Infine il **dispositivo virtuale** è costituito dai driver dei dispositivi e dai gestori delle interruzioni; partiziona lo spazio di memoria disponibile come un vettore lineare di blocchi fisici. Interagisce con i controller e si occupa del trasferimento delle informazioni tra memoria centrale e secondaria, un blocco per volta.

Realizzazione del file system
Richiede diverse strutture dati sia nei dischi che in memoria principale.
Nei dischi le principali sono:

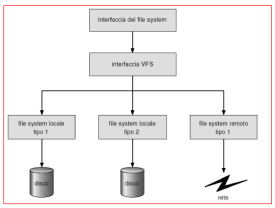
- Boot control block (secondary boot sector): è il primo blocco di un volume e contiene le informazioni su come avviare il SO memorizzato in quel volume (vuoto se non contiene SO).
- Partition/volume control block (indice del volume): contiene numero e dimensione dei blocchi, numero e locazione dei blocchi liberi, numero e locazione dei FCB liberi.
- FCB o descrittori di file (i-node UNIX): mantengono valori degli attributi dei singoli file e informazioni sull'allocazione dei blocchi dei file su disco.
- Strutture delle directory: tabelle in cui gli elementi contengono associazioni tra nime file/sottodirectory e collegamenti a FCB.

Nella memoria centrale i principali sono:

- Tabella di montaggio: informazioni su ciascun volume montato.
- Tabella delle directory: informazioni sulle directory accedute più di recente.
- Tabella generale dei file aperti: contiene una copia dei descrittore di tutti i file aperte e le informazioni sui processi che accedono a tali file.
- Tabella dei file aperti per ciascun processo: per ogni file aperto dal processo, si ha un puntatore all'elemento corrispondente della tabella generale (se l'accesso è sequenziale abbiamo anche un puntatore interno al file).

I dati si caricano in memoria centrale al montaggio del file system o quando si accede a un file/directory, e si eliminano allo smontaggio o all'eliminazione.
Una system call attraversa i diversi livelli a partire dal primo finché non arriva all'ultimo, ovvero il driver, che si occupa di far svolgere al controller del disco su cui si trova il file/directory interessato le operazioni richieste. System call **open(pathname, modo)**: il livello "struttura logica" controlla se il file è giù in uso tramite la tabella generale dei file aperti, se è già aperto aggiunge alla tabella dei file aperti un elemento che punta al corrispondente elemento della tabella generale dei file aperti; altrimenti, avvalendosi dei livelli sottostanti per leggere il contenuto delle strutture di directory che abbiamo nel pathname , alla ricerca del FCB del file: quando si trova viene copiato in un nuovo elemento della tabella generale dei file aperti e

viene aggiunto un elemento alla tabella dei file aperti del processo che punta ad esso. Infine restituisce un indice alla tabella dei file aperti del processo, quindi tutte le operazioni sul file avverranno usando questo indice. System call **read(fd, bytes, &buffer):** il livello "organizzazione fisica" calcola il record logico del file da leggere, individua il corrispondente blocco fisico usando le informazioni nella tabella dei file aperti; si trova quindi la tripla <superficie, cilindro, settore> corrispondente al blocco fisico e si invia la richiesta di lettura al driver (il processo va in waiting); il driver del disco accoglie la richiesta e la accoda alla lista delle richieste per mandarla al controllore del disco non appena sarà possibile. Quando la lettura viene effettuata il controller invia un interrupt gestito dal SO che richiama il driver del disco. Quest'ultimo copierà i dati dal buffer del controller al buffer del processo che li aveva richiesti (il processo va a ready) e andrà a rielaborare una nuova richiesta tra quelle pendenti.

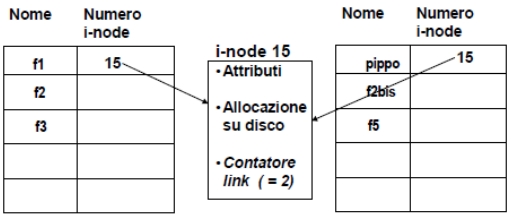


Il file system virtuale è una tecnica basata sugli oggetti che permette al SO di gestire tipi diversi di file system usando la stessa interfaccia. In pratica è un livello aggiuntivo che va tra l'interfaccia del file system ed il file system vero e proprio. Questa svolge due funzioni: separa le chiamante di sistema dall'implementazione delle operazioni corrispondenti; permette la rappresentazione univoca di un file su tutta una rete.

Realizzazione delle directory

Ogni directory necessita del collegamento con i descrittori dei file che contiene. Alcuni SO (windows) le realizza come file speciali con contenuto strutturato in modo specifico: tabella i cui elementi contengono nome di un file/sottodirectory, attributi, informazioni sulle allocazioni dei blocchi del file sul disco. L'inconveniente di questa implementazione riguarda il memorizzare i descrittori negli elementi della directory che pone dei problemi per l'implementazione dei link. Quindi, a discapito dell'efficienza usiamo un'altra implementazione che ci permette di realizzare in modo più conveniente i link. Questa implementazione prevede di tenere attributi e informazioni sull'allocazione in una struttura separata e fare in modo che ogni directory contenente un link ad uno stesso file punti all'i-node del file. Usiamo un contatore di link (nell'i-node) così possiamo facilmente implementare la cancellazione di un file con più pathname (la vera cancellazione si effettua solo quando il contatore si azzera, altrimenti si decrementa soltanto). Gli schemi di organizzazione delle directory sono due: lista lineare di elementi, semplice da programmare ma richiede un certo tempo per l'esecuzione delle operazioni; tabella hash: il nome del file viene usato per calcolare un valore hash, gli elementi della directory con lo stesso valore hash sono mantenuti in una lista lineare. Questo schema velocizza la ricerca nella directory e semplifica la creazione e la cancellazione.

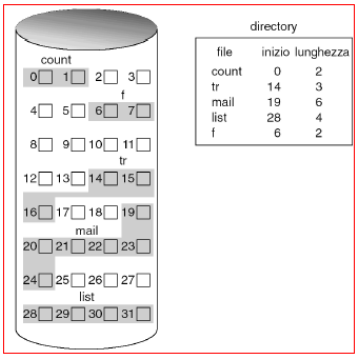
Nome	Attributi	Allocazione file su disco
d1		inizio = Blocco 30
d2		inizio = Blocco 43
f1		inizio = Blocco 110



Metodi di allocazione

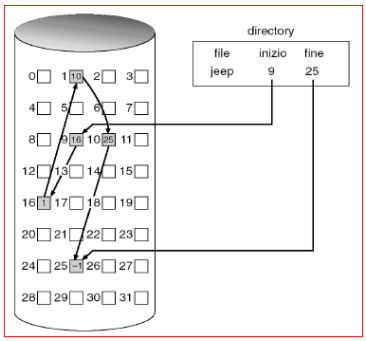
Stabiliscono una corrispondenza tra i record logici contenuti nei file e i blocchi fisici del disco. N_b è il numero di record logici contenuti in un blocco fisico, ottenuto facendo D_b/D_r dove D_b è la dimensione di un blocco e D_r è la dimensione di un record. I metodi più diffusi sono 3:

- Allocazione contigua: ogni file occupa un certo numero di blocchi contigui sul disco. La porzione allocata è definita dall'indice B del primo blocco e dal numero di blocchi (info contenute nel FCB). L'indice del blocco in cui si trova il record logico è: $R_i = B+(i/N_b)$. I vantaggi sono: basso costo della ricerca di un blocco e possibilità di accesso sequenziale e diretto. Gli svantaggi: frammentazione esterna (richiede compattazione o deframmentazione); elevato costo della ricerca dello spazio libero per l'allocazione di un nuovo file (si possono usare best-fit, first-fit, worst-fit); limiti sulle dimensioni dei file. Alcuni SO recenti usano uno schema di allocazione contigua modificato, ovvero viene inizialmente allocata una porzione contigua di spazio disco, poi

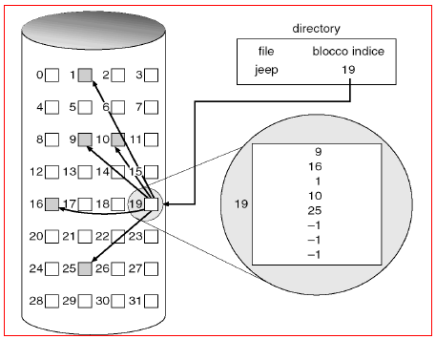


quando la quantità non è più sufficiente si alloca un'estensione, cioè un'altra porzione contigua (un file consiste in una o più estensioni).

- Allocazione a lista: i blocchi sui quali viene mappato un file sono organizzati in una lista concatenata, ovvero ogni blocco contiene un puntatore al successivo. Nell'FCB si mantiene il puntatore al primo blocco. Il record logico $R_i = i/N_b$, ma per trovarne l'indirizzo dobbiamo scorrere la lista fino a lui. Vantaggi: non c'è frammentazione esterna, basso costo della ricerca dello spazio libero per l'allocazione di un nuovo file o l'espansione di uno esistente. Svantaggi: maggior occupazione (ci sono anche i puntatori), l'accesso diretto e la ricerca sono costosi, ha una scarsa robustezza (se un link viene danneggiato non è più possibile accedere ai record dei blocchi successivi, soluzione lista a doppi puntatori o tabella di allocazione dei file). **Tabella di allocazione dei file (FAT)**: descrive la mappa di allocazione di tutti i blocchi, contiene un elemento per ogni blocco del disco il cui valore indica se il blocco è libero oppure occupato, in questo caso contiene l'indice dell'elemento della tabella corrispondente al blocco successivo. La FAT è memorizzata in un'area predefinita del disco ed ha i seguenti vantaggi: incrementa la robustezza, velocizza notevolmente l'accesso ai file tramite copia in memoria principale.



- Allocazione indicizzata: ad ogni file è associato un indice di dimensione prestabilita in cui sono contenuti gli indirizzi dei blocchi in cui è effettivamente allocato il file. Per accedere a un blocco si effettua una ricerca associativa nell'indice. Vantaggi: gli stessi dell'allocazione a lista più la possibilità di accesso diretto e una maggiore velocità di accesso. Svantaggi: richiede memoria per il blocco indice, scarso utilizzo dei blocchi indice (nel caso di file piccoli), overhead per l'accesso al blocco indice, le dimensioni del blocco indice limitano le dimensioni del file. Per evitare la limitazione sulla dimensione dei file alcuni SO usano le seguenti soluzioni: liste concatenate di blocchi indice, più livelli di indice, schema combinato.



Gestione dello spazio libero

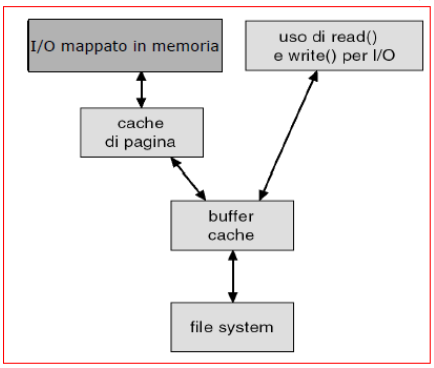
Il SO mantiene una lisa dei blocchi liberi allocabili, questa lista si può realizzare con un vettore di bit con tanti elementi quanti sono i blocchi. Se nell'elemento c'è un 1 significa che il blocco è libero se c'è 0 è occupato. Vantaggi: semplicità nel determinare il primo blocco libero o n blocchi liberi consecutivi. Svantaggi: efficiente solo se il vettore di bit si trova in memoria centrale, tale vettore richiede spazio extra, organizzazione ragionevoli per dischi di dimensioni non elevate. Ci sono altre organizzazioni quali:

- Lista concatenata: i blocchi liberi sono collegati l'uno all'altro, si memorizza solo il primo puntatore ma è difficile trovare blocchi liberi contigui (a meno che la lista non si tenga ordinata)
- Raggruppamento: il primo blocco libero memorizza gli indirizzi di n blocchi liberi successivi; i primi n-1 sono liberi, l'ultimo contiene gli indirizzi di altri n blocchi liberi. Permette di trovare velocemente gli indirizzi di un certo numero di blocchi liberi.
- Conteggio: si mantiene l'indirizzo del primo blocco libero e si usano gli stessi blocchi liberi per realizzare una lista concatenata, ogni blocco mantiene il numero di blocchi liberi consecutivi e l'indirizzo del primo blocco libero successivo a questi.

Efficienza e prestazioni

L'efficienza riguarda gli algoritmi per l'allocazione dei blocchi e la gestione dei blocchi liberi, e la quantità di informazioni in un elemento di directory o in un descrittore (data ultima scrittura, data ultimo accesso). Queste informazioni devono mantenersi aggiornate, quindi quando si scrive/legge un file si deve anche aggiornare un elemento di directory o un descrittore; ciò comporta una lettura nella memoria del blocco, la modifica della sezione e la riscrittura del blocco nel disco (inefficiente per file di frequente utilizzo). Quindi

quando decidiamo di salvare delle informazioni relative ai file, dobbiamo considerare l'impatto che hanno nell'efficienza in quanto dobbiamo poi mantenerle aggiornate. Un altro fattore che influenza l'efficienza è la dimensione del puntatore per l'accesso ai dati (lunghezza massima dei file vs spazio necessario per memorizzare i puntatori). Solitamente i SO usano puntatore di 16 o 32 bit (file di 64kb o 4gb), altri ne usano a 64 bit.
Le prestazioni si migliorano usando parte della memoria centrale come se fosse un disco virtuale (disco RAM), a carico degli utenti, oppure tramite il buffer (disk) cache, una sezione separata della memoria centrale dove sono mantenuti alcuni blocchi del disco, a carico del SO. Il buffer cache funziona gestendo parte della RAM come una cache per blocchi del disco, così si risparmiano costi per accesso al disco. Questo buffer viene gestito completamente in SW; gli algoritmi di rimpiazzamento potrebbero essere gli stessi usati per le pagine in RAM. In questo modo possiamo implementare in modo esatto l'LRU, perché gli accessi al disco sono più lunghi di quelli alla memoria e sono anche meno frequenti. Per i blocchi critici presenti in cache è opportuno, dopo ogni modifica, aggiornare la copia su disco.
Alcune versione di UNIX e di LINUX, hanno il problema della double caching, ovvero l'accesso al disco attraverso il file system usa direttamente il buffer cache; mentre l'I/O mappato in memoria legge i blocchi e li memorizza nel buffer cache, ma deve appoggiarsi anche a una cache delle pagine in quanto il sistema memoria virtuale non può interfacciarsi direttamente con il buffer cache. Per evitare questo spreco di memoria e di cicli della CPU e di I/O, oltre a eliminare il rischio di eventuali incongruenze tra le due cache, si usa un buffer cache unificato.



L'algoritmo di rimpiazzamento LRU non sempre è il migliore, infatti:

- Il metodo usato per l'accesso ai file influenza le prestazioni.
- L'accesso sequenziale può essere ottimizzato (free-behind, rimuove un blocco non appena viene richiesto il successivo; read-ahead, legge il blocco richiesto ed alcuni altri che lo seguono sequenzialmente).

Le scritture dei blocchi possono avvenire in due modi:

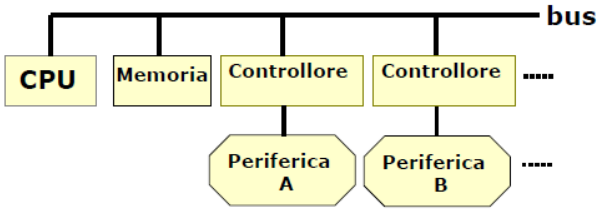
- Scritture sincrone, avvengono nell'ordine in cui le riceviamo, i dati non vengono memorizzati in modo temporaneo nella cache del controllore del disco.
- Scritture asincrone, si memorizzano i dati nella cache del controllore del disco e si restituisce immediatamente il controllo al processo invocante (se si perde corrente o si stacca il dispositivo i dati si perdono)

Per grandi quantità di dati la scrittura risulta più veloce della lettura. Infatti quando dobbiamo scrivere un dato su disco, si memorizzano le pagine sulla cache che funge da buffer mentre il driver del disco riordina la coda in modo da minimizzare gli spostamenti della testina, questo permette di ottenere tempi ottimali per la rotazione. Quindi, a meno che non si chiedono scritture sincrone, un processo scrive nella cache e solo successivamente il sistema trasferirà i dati sul disco in modo asincrono, ciò comporta una scrittura rapidissima. Mentre la lettura viene effettuata con l'I/O che a blocchi fa delle letture anticipate, ma non si giova della modalità asincrona, risultando quindi più lenta.

Gestione dispositivi I/O

Sottosistema di i/o

Insieme dei servizi offerti dal SO per assicurare ai processi un adeguato supporto per l'esecuzione delle operazioni sui dispositivi di I/O. Questo sottosistema fornisce un'interfaccia che nasconde i dettagli dell'hardware e garantisce l'accesso efficiente ai dispositivi gestendo anche eventuali malfunzionamenti. L'hardware dei dispositivi I/O riguarda sia i dispositivi/periferiche non elettroniche, che i controllori dei dispositivi (device controller) ovvero la parte elettronica, che le componenti di connessione, cioè bus



(insieme di linee di connessione) e porte (punti di connessione). Il **controllore** è un circuito elettronico che collega periferiche e bus di sistema (e quindi la CPU). I suoi compiti sono: controllare il funzionamento del dispositivo, attraverso segnali regolati da un protocollo; fornire alla CPU un insieme di registri indirizzabili dalle operazioni di I/O.

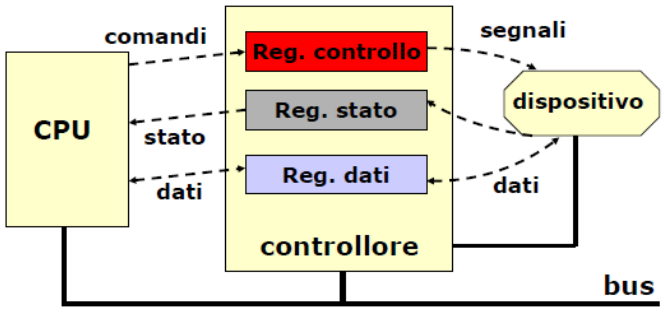
I dispositivi di I/O sono molti e possono differire in molti aspetti, tra i quali: numero di funzioni che possono svolgere, quantità di errori e malfunzionamenti che possono generare, velocità di trasmissione dei dati. Possiamo dunque cercare di classificare i diversi dispositivi secondo alcune loro caratteristiche:

- Sorgente o destinazione dei dati (operatore umano, altre apparecchiature, apparecchiature o utenti remoti).
- Accesso: sequenziale o diretto.
- Condivisione: dedicati (seguono un processo per volta), condivisi (seguono più processi contemporaneamente).
- Direzione di I/O: lettura e scrittura, solo lettura, solo scrittura.
- Organizzazione/trasferimento dei dati: a blocchi, a caratteri.

Per i dispositivi che differiscono per il modo in cui sono organizzati i dati da trasferire abbiamo le seguenti opzioni:

- Dispositivi a blocchi (dischi): registrano i dati in blocchi di dimensione fissa, ogni blocco ha un suo indirizzo e può essere letto e scritto in maniera indipendente. I comandi principali sono read, write, seek. Solitamente si utilizzano tramite un file system.
- Dispositivi a caratteri (tastiere, mouse, stampanti etc): ricevano o inviano stringhe di caratteri senza però poterle strutturare ed indirizzare internamente, i comandi principali sono get e put. Si utilizzano usando librerie che permettono l'accesso a intere sequenze di caratteri per volta.
- Dispositivi speciali (timer): sono quelli che non rientrano in nessuna delle precedenti categorie. Il timer per esempio è un temporizzatore programmabile che si può regolare per provocare un interrupt dopo un certo intervallo di tempo.

Il controllore di un dispositivo di I/O dispone di alcuni registri su cui agisce la CPU. I registri sono indirizzati da comandi di I/O. La comunicazione tra CPU e controllore avviene tramite un bus. I registri del controllore sono i seguenti:



- **Registro di controllo**, consente alla CPU di controllare il funzionamento del dispositivo. Questo registro è di sola scrittura per la CPU che ci inserisce i valori opportuni per richiedere le operazioni (bit di start=1, attiva il dispositivo. Altri bit selezionano l'operazione richiesta. Bit di abilitazione delle interruzioni=1, consente al controllore di fare un interrupt alla CPU alla fine dell'operazione).
- **Registro di stato**, consente al dispositivo di mantenere aggiornato lo stato in cui si trova. Questo registro è di sola lettura per la CPU. (Bit di flag=1, alla fine dell'operazione richiesta, ovvero quando passa da 0 a 1 si può lanciare un interrupt. Bit di errore=1, quando si verifica un errore, possono esserci tanti bit di errore quante sono le possibili cause di malfunzionamento).
- **Registri dati**: buffer del controllore per i dati da trasferire.

L'accesso ai registri può avvenire in diversi modi:

- I/O separato in memoria (port-mapped I/O): esistono specifiche porte (logiche) di IO dedicate al dispositivo. L'accesso avviene tramite speciali istruzioni di I/O che specificano gli indirizzi delle porte.

- I/O mappato in memoria: una parte dello spazio di indirizzi è collegata ai registri dei dispositivi. Questo metodo è efficiente e flessibile (l'accesso ai registri usa le stesse istruzioni dell'accesso alla memoria, permettere di scrivere driver in linguaggio di alto livello). Serve però un accorgimento per la protezione, ovvero si alloca lo spazio di indirizzamento di I/O fuori dallo spazio utente.
- Soluzione ibride (Pentium, controllore della grafica): I registri di controllo e stato si accedono con I/O separato in memoria, mentre il registro dati con l'I/O mappato in memoria.

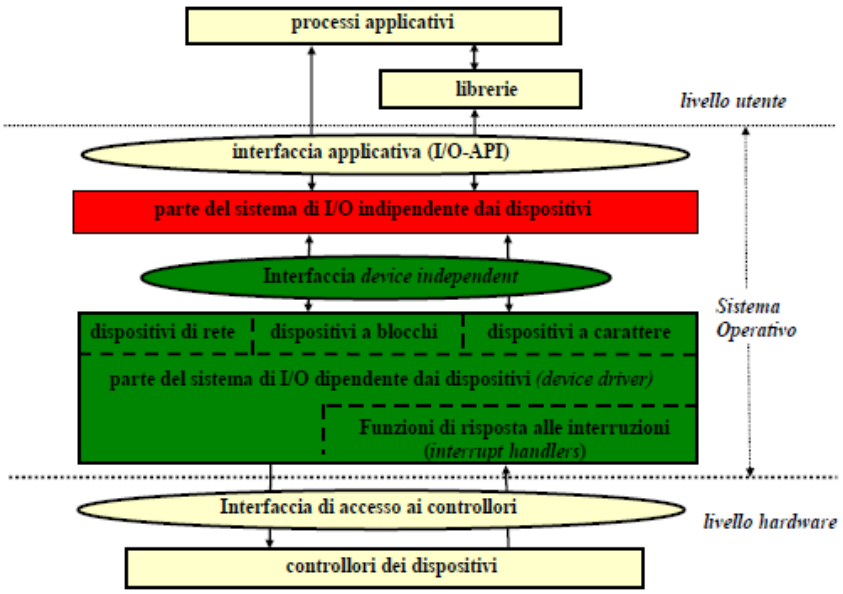
Compiti ed organizzazione logica del sottosistema di I/O

I compiti del sottosistema di I/O sono:

- Nascondere i dettagli hardware dei singoli dispositivi.
- Fornire delle astrazioni dei dispositivi per evitare che si debba conoscere ogni componente hardware per programmare.
- Fornire un'interfaccia di programmazione applicativa per l'ingresso/uscita delle periferiche.
- Naming: definire lo spazio dei nomi con cui identificare i dispositivi, a livello API sono rappresentati da nomi simbolici, mentre a livello di sistema da numeri interi. Questi numeri sono solitamente indici di un array di puntatori a descrittori.
- Gestire malfunzionamenti, alcuni possono essere risolti dal sistema recuperando il corretto funzionamento (ripetendo l'operazione), altri vanno sollevate al processo che ha invocato l'operazione di I/O o trasformate in un messaggio per la consolle (mancanza carta).
- Gestire le interruzioni generate dai dispositivi (completamento di un trasferimento), va garantita la sincronizzazione tra l'attività di un dispositivo e quella del processo che l'ha invocato.
- Buffering: memorizzazione temporaneamente i dati in transito.

Organizzazione logica del sottosistema di I/O:

Il sottoinsieme è logicamente suddiviso in 2 componenti: **device independent** (I/O API), che ha il compito di omogeneizzare le funzioni di accesso ai vari dispositivi; **device dependent**, che ha il compito di nascondere le peculiarità dei dispositivi e dei loro controllori al resto del SO. Questa componente è costituita dai gestori dei dispositivi (e delle interruzioni) che si interfacciano direttamente coi controllori tramite i loro registri, controllandone il comportamento.



Un processo applicativo invoca funzioni fornite dall'I/O API. Queste, poi, invocano funzioni del driver del dispositivo coinvolto che a questo punto si occupa di controllare tutto lo svolgimento dell'operazione da parte del dispositivo.

Componente device independent: i servizi offerti da questo livello rendono più semplice, sicuro ed efficiente l'uso dei dispositivi. In alcuni sistemi i servizi coincidono con quelli offerti dal file system. Alcune funzioni che non dipendono dal tipo di dispositivo e che sono realizzate da questa componente sono: *naming* dei file e dei dispositivi (trattati come file speciali i cui nomi sono inseriti nel file system); *buffering*, disaccoppiamento tra i registri del controllore e la memoria virtuale del processo che ha richiesto il trasferimento; gestione dei malfunzionamenti non gestiti dal driver; protezione e controllo degli accessi; allocazione dinamica dei dispositivi ai processi; *spooling*, conservazione delle richieste per una periferica quando essa può servirne solo una alla volta (stampante).

Buffering

Per ogni operazione di trasferimento dati tra dispositivi o tra dispositivi e applicazione, in modo trasparente al richiedente, immagazzina in aree di memoria tampone i dati soggetti al trasferimento che non possono essere ancora memorizzati nella destinazione finale. I suoi scopi sono dunque, compensare le differenze di velocità fra produttore e consumatore, connettere componenti operanti con dati organizzati in modo diverso, supportare la "semantica della copia", ovvero la conformità dei dati nel buffer di un processo con

la loro copia su disco. Un processo applicativo deve leggere ed elaborare una sequenza di blocchi, per evitare troppi context switch sarebbe comodo eseguire in parallelo il trasferimento nel buffer del blocco successivo, questo si effettua col read ahead. Il doppio buffer permette ancora un altro miglioramento, la lettura in un buffer si esegue in parallelo al trasferimento del contenuto dell'altro buffer nell'area di memoria virtuale del processo

Spooling

Si usa in alternativa all'allocazione dinamica (che prevede un gestore del dispositivo a cui indirizzare richieste e rilasci). Lo spooling fa interfacciare il processo su una copia virtuale e privata del dispositivo, realizzata con un file sul disco. Consiste quindi nel memorizzare temporaneamente i dati da inviare ad un dispositivo in un'area di lavoro, da dove un programma apposito può successivamente prelevarli per elaborali. Lo spooling risulta utile perché i vari dispositivi fisici di un computer elaborano i dati a velocità diverse, ed esso fornisce una locazione dove i dati possono essere "posteggiati" in attesa che anche il dispositivo più lento possa usarli. Inoltre lo spooling permette anche a più processi di usare uno stesso dispositivo seriale senza doversi sincronizzare. La più comune applicazione riguarda il **print spooling**: i documenti sono memorizzati in un buffer da dove si leggono da un processo di sistema o del kernel, il quale poi li inserisce nella *coda del dispositivo*, copiata nel dispositivo elemento per elemento. Siccome i documenti sono in un buffer accessibili al software mentre si elaborano il processo utente può effettuare altre operazioni. Dopo di che si crea una "coda di file in attesa di essere serviti" e non una lista di richieste. Questo permette di identificare il problema dell'ordine (scheduling) con cui servire gli elementi della coda, l'efficienza si può quindi migliorare se le richieste possono essere servite in ordine diverso da quello in cui arrivano. Il sosftware di spooling può consentire di assegnare priorità diverse, avvertire di utenti del completamento, distribuire i job tra più dispositivi ecc.

Componente device dependent: la sua principale funzione è quella di fornire i gestori dei dispositivi (driver), offrendo al livello superiore un insieme di funzioni di accesso ai dispositivi.

Gestione di un dispositivo

Per gestire un dispositivo ci sono 3 diverse modalità di interazione tra controllore e gestore:

- Polling o a controllo di programma o a controllo diretto o I/O programmato (PIO)
- A interruzioni (interrupt-driver I/O)
- Accesso diretto in memoria o direct memory access (DMA)

Un processo esterno è una sequenza di azioni fisse (cablate) eseguite da un dispositivo (HW) che ha il suo controllore e che consideriamo come un processore dedicato. Un processo interno invece è quello che attiva il dispositivo, solitamente è il driver per la gestione del dispositivo.

- Gestione di un dispositivo a controllo di programma, il processo interno controlla direttamente la terminazione di ciascuna operazione richiesta, ovvero determina lo stato del dispositivo mediate la ripetuta lettura del bit di flag. Il problema che si può verificare è l'attesa attiva o l'interrogazione ciclica (polling). Questo metodo non è adatto a sistemi multiprogrammati, infatti per utilizzare meglio la CPU, mentre un processo non può proseguire perché attende un I/O, è necessario sospenderlo commutando la CPU.
- Gestione di un dispositivo a interruzione, al dispositivo è associato un semaforo di sospensione (inizializzato a 0); nel processo interno, il bit di flag=0 è sostituito da una wait() sul semaforo (con conseguente sospensione del processo). Quando si è completata l'operazione, il controllore del dispositivo lancia un segnale di interruzione. La CPU controlla la presenza di segnali di interruzione dopo l'esecuzione di ogni istruzione, se ne trova uno salva lo stato corrente e passa all'esecuzione della routine di gestione. La routine attiva l'esecuzione della funzione risposta alle interruzioni da dispositivo, la quale sblocca il processo interno eseguendo una signal() sul semaforo.
    - Gestione delle interruzioni, nei moderi SO, si usano meccanismi raffinati, il tipo di una interruzione è individuato tramite un numero che si può usare come indice nel vettore delle interruzioni, che è una tabella che contiene gli indirizzi delle corrispondenti routine di gestione. Alcune interruzioni sono mascherabili (solitamente quelle generate dai controllori), cioè possono essere disattivate dalla CPU prima dell'inizio dell'istruzione che non deve essere interrotta. Altre interruzioni invece non sono disattivabili (errori non recuperabili). Le interruzioni possono avere delle priorità, quelle alte possono sospendere la gestione di quelle basse. Lo stesso meccanismo delle interruzioni viene usato anche per gestire le eccezioni (divisione per 0, accesso ad indirizzi protetti o inesistenti etc). Un

esempio di interruzioni è dato dall'esecuzione di una system call, solitamente per eseguirle si invocano routine di libreria. La routine controlla i parametri, li assembla in una struttura dati che passa al kernel, ed esegue un'istruzione chiamata **trap** (**interruzione del software**) che ha un argomento che identifica il servizio richiesto. Quando l'interruzione è rilevata, si memorizza lo stato in cui eravamo, si passa al modo supervisore e si esegue la procedura del kernel che realizza il servizio richiesto. Alle trap si assegna una priorità bassa rispetto alle interruzioni generate dai dispositivi I/O. Si può verificare un inconveniente nel trasferimento di n blocchi dati, ovvero il processo verrà bloccato per le n-1 volte precedenti all'ultima in cui potrà effettuare il trasferimento. Per evitare tutte queste interruzioni, forniamo una funzione che accetta come argomento il numero n di blocchi dati da trasferire, così che blocchiamo il processo fino al trasferimento dell'ultimo blocco dato e lo riattiviamo solo una volta terminato il trasferimento. Per ottenere questo la routine di gestione dell'interruzione deve distinguere le interruzioni intermedie da quella finale, con la quale riattivare il processo. Questo è realizzabile grazie al descrittore del dispositivo.

Descrittore e gestore di un dispositivo

Il descrittore di un dispositivo è una struttura dati in memoria che rappresenta il dispositivo ed è accessibile sia al processo interno che alla routine di gestione dell'interruzione. Esso ha un duplice scopo: racchiude al suo interno le informazioni associate al dispositivo e consente la comunicazione tra processo interno e dispositivo o viceversa. I suoi campi sono: indirizzo registro di controllo, indirizzo registro di stato, indirizzi registri dati, semaforo (dato_disponibile), quantità di dati da trasferire (contatore), indirizzo del buffer in memoria (puntatore), esito del trasferimento (esito).

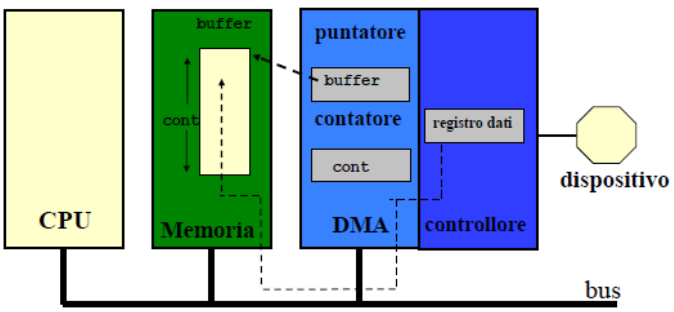
Il gestore di un dispositivo (device driver) è una componente formata da: descrittore del dispositivo e funzioni di accesso al dispositivo (del livello device independent e di risposta alle interruzioni). Il funzionamento tipico di un gestore passa attraverso i seguenti stadi:

1. Inizializza il dispositivo
2. Accetta richieste di operazione e ne controlla la correttezza
3. Gestisce le code delle richieste che non possono essere eseguite subito
4. Sceglie la prossima richiesta da servire e la traduce in una sequenza S di comandi a basso livello da inviare al controllore del dispositivo.
5. Trasmette i comandi S al controllore, eventualmente bloccandosi attendendo il completamento dell'esecuzione di un comando
6. Controlla l'esito di ciascun comando gestendo eventuali errori
7. Invia l'esito dell'operazione ed eventuali dati al richiedente

Un timer invece, si usa per lanciare esecuzioni cadenzate nel tempo. Si utilizza come scheduling della CPU (sistemi time-sharing), per la gestione della data e del tempo (orologio di sistema), attese programmate e ricezione di segnali di time-out. Il driver sarà costituito da: inth(), funzione di risposta alle interruzioni; delay(), primitiva che i processi possono invocare passandogli come argomento un intero che indica la durata dell'attesa richiesta; descrittore, che oltre ai campi normali avrà anche un contatore (la CPU scrive un intero, il timer lo decrementa periodicamente e quando si azzera lancia un interrupt); fine_attesa[], array di semafori su cui si sospendono i processi che invocano delay(); ritardo[], array di interi che indicano i quanti di tempo che devono ancora passare prima che ogni singolo processo possa essere risvegliato.

Gestore di un dispositivo in DMA

Si usa per il trasferimento di grandi quantità di dati, ma serve hardware apposito (DMA controller). Consente di trasferire i dati dal registro dati del controllore direttamente alla memoria centrale (e viceversa) tramite il bus senza intervento della CPU. La CPU non può accedere alla memoria centrale quando il controllore del canale di DMA prende possesso del bus (cycle stealing, sottrazione di cicli). Nel canale DMA sono presenti 2 registri: puntatore e contatore, gestiti via hardware (questi due registri non sono quindi presenti nel descrittore di un dispositivo con DMA). La CPU carica il valore di questi registri e delega il compito di interagire col bus di memoria al controllore DMA. Questo metodo ha i seguenti vantaggi: riduce il tempo di trasferimento di un insieme di



blocchi di dati, elimina la necessità di usare la CPU per eseguire la funzione di gestione delle interruzioni per ciascun blocco letto (nel caso è lanciata un'unica interruzione a fine trasferimento). Come abbiamo detto, il DMA trasferisce i dati usando il bus comune alla CPU, se quest'ultima però vuole utilizzare il bus durante il trasferimento effettuato dal DMA, allora la CPU dovrà aspettare per un ciclo, perché il DMA ha priorità maggiore. Dare priorità maggiore al DMA è molto importante soprattutto per quelli di dispositivi periferici con buffer dati di piccole dimensioni. Inoltre i dispositivi DMA hanno un impatto minore nelle architetture moderne in cui la CPU ha cache di dimensioni tali da soddisfare la maggioranza dei riferimenti in memoria senza dover ricorrere all'uso frequente del bus per accedere alla memoria.

Trasformazione delle richiesti di I/O in operazioni hardware

Come leggere dati da un file in memoria secondaria:

- Determinare quale dispositivo contiene il file
- Tradurre il nome del file nella rappresentazione del dispositivo
- Leggere fisicamente i dati dal dispositivo e scriverli in un buffer
- Rendere i dati disponibili per il processo
- Restituire il controllo a processo

La trasformazione avviene in parte nel file system e il resto nel sottosistema di I/O.

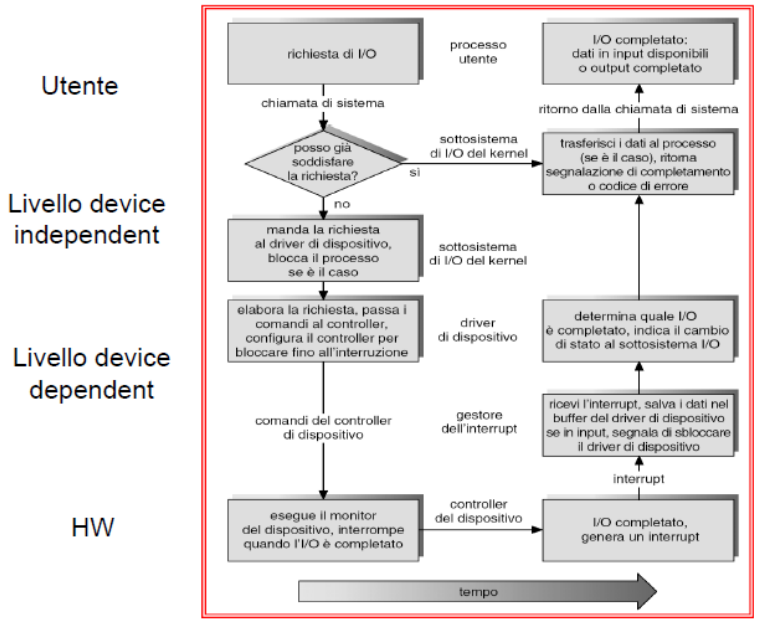
Eseguire una read():

Un processo esegue una system call read() su un file descriptor restituito da una precedente open().

Il codice della system call nel kernel controlla la correttezza dei parametri (se i dati sono già disponibili nel buffer, vengono restituiti al processo invocante e la system call termina, altrimenti il processo invocante è sospeso sulla coda del dispositivo e viene generata una richiesta di I/O).

La richiesta viene prima o poi inviata al driver del dispositivo.

Il driver alloca un buffer nel kernel per ricevere i dati e inserisce i comandi opportuni nei registri del controllore e del dispositivo. Il controllore si assicura che il dispositivo hardware effettui la lettura dei dati e genera un interrupt quando si è finito. Tramite il vettore degli interrupt viene attivato il gestore corrispondente, che memorizza i dati necessari, invia un ack al driver del dispositivo e termina. Il driver riceva l'ack, quindi determina quale richiesta di I/O è stata terminata ed il suo stato, e segnala al sottosistema di I/O del kernel che la richiesta è stata completata. Il kernel trasferisce i dati (o codici di ritorno) nello spazio di memoria dell'invocante e sposta il processo dalla coda di attesa sul dispositivo alla coda dei processi ready. Quando lo scheduler gli riassegnerà la CPU, il processo invocante riprenderà la sua normale esecuzione.



GESTIONE E ORGANIZZAZIONE DEI DISCHI

Compiti del sistema operativo

- Gestione dei dispositivi fisici
- Presentazione alle applicazioni di un'astrazione di macchina virtuale

Per il dischi il SO fornisce due astrazioni: il dispositivo grezzo (rappresentato come un array di blocchi di dati); il file system (directory e file organizzati gerarchicamente). L'efficienza e l'affidabilità dell'intero sistema dipendono da quelle dei dischi.

Struttura fisica dei dischi
Un disco è formato da uno o più piatti rotondo ricoperti da uno strato di materiale magnetico sul quale scriviamo e leggiamo i dati. Inoltre è presente una testina di lettura/scrittura. Quando si effettuano le operazioni la testina rimane fissa mentre il disco ruota sotto di lei. La struttura fisica di un piatto è divisa in *tracce* circolari a loro volta suddivise in *settori*. La formattazione a basso livello (fisica), divide il disco in settori che il controllore può leggere e scrivere, solitamente un settore è caratterizzato da un preambolo, una zona dati e una coda, il preambolo e la coda contengono dati e informazioni utili al controllore. I dischi possono essere divisi in diverse tipologia a seconda di alcune loro caratteristiche:

- Fissi o rimovibili
- Testine fisse (ne abbiamo tante quante sono le tracce), testine mobili (una per disco che si muove in modo radiale)
- Velocità di funzionamento angolare costante o lineare costante.
- Connessione al calcolatore tramite bus o tramite rete

Per quanto riguarda la velocità di funzionamento, nei dischi con velocità angolare costante la velocità di rotazione è fissa. Questo comporta una densità decrescente di bit dalle tracce interne a quelle esterne per mantenere costante la quantità di dati che scorre. Mentre nei dischi con velocità lineare costante la densità dei bit per traccia è uniforme. Quindi al centro avremo meno settori rispetto all'esterno del disco, questo comporta il dover aumentare la velocità di rotazione man mano che ci avviciniamo al centro per mantenere costante la quantità di dati letti.

La connessione al calcolatore tramite bus è curato da controllori di due tipi: gli adattatori (posti all'estremità del bus), controllori dei dischi (incorporati nel disco stesso). Per eseguire un'operazione il calcolatore inserisce un comando nell'adattatore, esso lo invia al controllore del disco, il quale a sua volta agisce sugli elementi meccanici per effettuare il comando. La connessione tramite rete, è un sistema di memoria speciale al quale accediamo in remoto tramite una rete di trasmissione dati. I client accedono alla NAS tramite un'interfaccia RPC. Questa interfaccia è realizzata per mezzo di due protocolli TCP e UDP sopra una rete IP. Un sistema connesso tramite rete è meno efficienze e ha prestazioni inferiori.

Struttura logica dei dischi
A livello più basso di astrazione (grezzo), abbiamo detto che si vedono come array monodimensionali di blocchi logici. Questo array si mappa nei settori del disco in modo sequenziale. Per memorizzare file su disco il SO registra le proprie strutture dati su disco in due fasi: suddivide il disco in uno o più gruppi di cilindri (partizioni), crea un file system (formattazione logica).

Schedulazione degli accessi al disco
L'efficienza di un disco è dovuta da due fattori: **TA e TT.** Il TA (tempo di accesso) sarebbe il tempo necessario per posizionare la testina nel punto giusto (mezzo giro è dell'ordine dei millisecondi). Il TT (tempo di trasferimento) è il tempo necessario per effettuare il trasferimento dei dati relativi a un singolo settore (si approssima con il quoziente della divisione tra tempo per compiere un giro e numero di settori per traccia, è dell'ordine dei microsecondi quindi trascurabile). L'efficienza è quindi dovuta dal TA, per migliorarla possiamo usare algoritmi diversi per determinare in che ordine servire le richieste accodate. I principali algoritmi sono i seguenti: **FCFS** *(servite nell'ordine in cui arrivano),* **SSTF** *(seleziona la richiesta che ha la traccia più vicina a quella in cui siamo),* **SCAN** *(si parte da un'estremità e si va verso l'altra),* **C-SCAN** *(come lo SCAN ma una volta arrivati in fondo si riparte dal 1 dell'estremità opposta, come se fosse circolare),* **LOOK** *(si arriva fino alla richiesta più estrema rispetto alla nostra direzione ma non fino all'estremità, si può fare anche qui la variante circolare,* **C-LOOK***).*

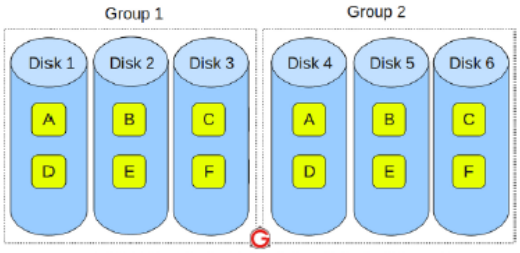
Lo standard RAID
Per migliorare ulteriormente le prestazioni si possono usare più dischi e memorizzare i dati in modo da agire in modo parallelo sui singoli dischi. Questo permette anche di migliorare la robustezza replicando i dati. Il RAID è affidato a 7 livelli che differiscono per il grado di ridondanza dei dati.

- Livello 0 (striping): dati distribuiti su più dischi senza ridondanza.
- Livello 1 (mirroring): tutti i dischi sono duplicati

- Livello 2: organizzazione con codici pe la correzione degli errori.
- Livello 3: organizzazione con bit di parità intercalati
- Livello 4: blocchi di parità intercalati
- Livello 5 (block interleaved parity): se occorrono n dischi per i dati, se ne usano n+1 e per ogni gruppo di n settori consecutivi si crea un settore contenente solo bit di parità.
- Livello 6: simile al 5 ma memorizza ulteriori informazioni ridondanti per poter gestire guasti di più dischi.

Si possono "fondere" più RAID solitamente si usa o RAID 0+1 oppure RAID 1+0.

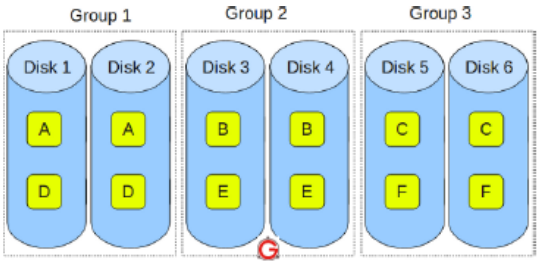
RAID 01: i dischi sono raggruppati in due gruppi e i dati sono distribuiti sui dischi di uno stesso gruppo. I due gruppi sono identici, quindi l'uno è copia dell'altro.



RAID 01 – Blocks Striped. (and Blocks Mirrored)

RAID 10:

I dischi sono raggruppati a coppie (gruppi di 2), ogni gruppo contiene gli stessi dati duplicati.



RAID 10 – Blocks Mirrored. (and Blocks Striped)

La differenza tra le due implementazioni dipende dal livello di tolleranza ai guasti, in ciò RAID 10 è migliore rispetto a RAID01. Infatti, nello 10 con un guasto singolo il disco diventa inaccessibile, ma il suo duplicato è ancora accessibile, mentre in 01 l'intera sezione dei dati diventa inaccessibile, lasciando disponibile solo l'altra sezione.