SISTEMI OPERATIVI

2011/2012

by OrangeWorld

Nota importante: questi appunti sono stati presi da me a lezione, e integrati ove necessario con materiale reperito su siti web, libri, etc.
Questi appunti non sono assolutamente da ritenere come sostituto di materiale didattico fornito dal professore del corso (libri, dispense etc.), ma come supporto allo studio personale; non sono inoltre né visionati né approvati dal Docente del Corso.
Non escludo che siano affetti da errori e/o imprecisioni, e declino ogni responsabilità sull'uso che si farà di questo fascicolo.

INTRODUZIONE AI SISTEMI DI ELABORAZIONE

Elementi di base

E' caratterizzato da:

- *processore* (controlla le operazioni del computer e svolge le sue funzioni di elaborazione dei dati);
- *memoria principale* (memorizza dati e programmi ed è volatile);
- *moduli I/O* (trasferimento dati da computer ad ambiente esterno);
- *bus* (interconnessioni nel sistema).

Un processore contiene un insieme di registri:

- *registri visibili all'utente*: minimizzano i riferimenti alla memoria principale ottimizzando l'uso di registri; sono disponibili solitamente per dati, indirizzi e codici di condizione. I registri possono essere "dei dati" e "degli indirizzi", ed entrambi possono essere generali o riguardare una determinata funzione (index register →aggiungere un indice a un valore base per ottenere l'indirizzo effettivo | segment pointer →viene usato un registro per memorizzare l'indirizzo base di un segmento della memoria | stack pointer →permette l'uso di istruzioni che non contengono indirizzo come push e pop essendo lo stack una pila). Esistono anche i flag (i bit settati dall'hardware del processore);
- *registri visibili all'utente*: controlla le operazioni del processore e della routine privilegiate del sistema operativo per l'esecuzione dei programmi (PC →indirizzo successiva istruzione da prelevare | IR →istruzione prelevata più recentemente); il registro PSW che contiene informazioni di stato.

Un *programma* è un insieme di istruzioni immagazzinate in memoria. Un istruzione viene eseguita da due operazioni, il *fetch* (prelievo) e l'*execute* (esecuzione):

processore memoria -> il PC memorizza l'indirizzo dell'istruzione successiva da prelevare (alla fine del fetch si incrementa di 1 il PC) | *processore i/o ->* a carica nell'IR | *elaborazione dei dati ->* che verrà interpretata dal processore | *controllo ->* eseguita e effettuati i controlli necessari.
L'accumulatore è il registro che contiene i dati (4 bit per il codice operativo).

1. L'indirizzo della prima istruzione viene caricata in IR;
2. caricata una parola di memoria | 940 →indicano l'indirizzo;
3. Prelievo istruzione successiva;
4. AC = AC(vecchio) + indirizzo nuovo;
5. viene prelevata l'istruzione successiva;
6. il contenuto di AC è già salvata in 941 quindi si incrementa PC.

Le *interruzioni* servono per migliorare l'efficienza dell'elaborazione, poiché i dispositivi esterni sono più lenti del processore. Ogni programma è diviso in tre sezioni: preparazione dell'operazione, esecuzione dell'operazione e conclusione dell'operazione e gestione flag (indica il successo o il fallimento dell'operazione). Quando avviene un'interruzione, il processore si libera per svolgere altre operazioni e quando il dispositivo esterno è pronto ad accettare altri dati, il relativo modulo I/O manda un segnale di richiesta di interruzione al processore (*interrupt handler*). Al ciclo dell'istruzione viene aggiunto il ciclo di interruzione che si attiva solo dopo che verifico che c'è la presenza dell'interrupt. Con le interruzione una parte del tempo dell'esecuzione dell'operazione di I/O si sovrappone all'esecuzione delle istruzioni utente.
Con interruzioni multiple possiamo comportarci in due modi: il primo disabilitando le interruzioni e riattivandole dopo la routine di gestione dell'interrupt, poiché quest'ultimi sono gestiti in ordine sequenziale ma non tenendo conto delle priorità. L'altro metodo consiste nel definire quella con maggiore priorità cosi da interrompere quella a bassa priorità (interrupt interrompono altri interrupt).

Nell'I/O il processo si interrompe quando il modulo I/O è pronto a scambiare dati; il processore gestisce l'interrupt solo se qualcosa gli dice che ci sta un interrupt (prima faceva sempre controlli) e salva i registri del programma eseguito e comincia ad eseguire l'interrupt handler. L'interrupt serve per dire quando è finita l'operazione e si può cominciare a partire con la successiva; non si sta mai in attesa finché la coda non è piena.

Il ***DMA*** (Direct Memory Access) serve per eseguire operazioni frequenti molto onerose e "pesanti"; trasferisce blocchi di memoria da una parte all'altra ogni volta che gli arriva un'interrupt (è un co-processore dell'I/O). Il bus è condiviso e quindi non si sono interrupt perché il processore non salva un contesto e non esegue altre operazioni limitandosi a rimanere in pausa per il ciclo.

Usiamo la memoria RAM per tenere i dati del disco (usata come buffer) cosi se un programma vuole rileggere quel dato è già subito li dentro.

Scendendo verso il basso, diminuisce il costo per bit e la frequenza di accesso alla memoria da parte del processore ma aumenta la capacità e il tempo di accesso.

La tecnica ***disk cache*** permette di aggiungere dei livelli addizionali (come una porzione di memoria principale come buffer) per memorizzare dati temporanei letti su disco. Si raggruppano scritture su disco avendo cosi meno trasferimenti di dati anche se di maggiori informazioni e alcuni dati possono essere referenziati da un programma per far si che la cache software la recuperi più velocemente.

SISTEMA OPERATIVO

E' un'interfaccia tra le applicazioni e l'hardware; contiene le funzionalità di base e controllo l'esecuzione di programmi applicativi (anche in fasi diverse). Queste definizioni riguardano solo una parte del sistema operativo. Non dipende da nessun'altra parte del computer se non dall'hardware. Il sistema operativo gestisce risorse e attività.

Una *risorsa* riguarda ogni cosa per l'esecuzione del programma (CPU time, I/O devices, memoria, codice eseguibile). Il sistema operativo permette la creazione ed esecuzione dei programmi, l'accesso ai dispositivi I/O, controllato ai file e al sistema, rilevazione degli errori e risposta e contabilità.

Un *kernel* parte all'avvio e gestisce le risorse, si trova in memoria principale e contiene le funzioni più usate. Le strutture dati devono essere aggiornate in maniera atomica.

Per fare un sistema operativo bisogna rispettare alcuni parametri come la facilità nell'uso, efficiente (devono usare le risorse di sistema in maniera efficiente come consumare poco e la velocità nell'aprire applicazioni) e possibilità di sviluppare (introducendo nuovi programmi). L'adozione di terminali grafici e a modalità di pagina invece di quelli a linea può influire sulla progettazione e l'evoluzione del sistema operativo, offrendo di conseguenza nuovi servizi e correzioni anche dopo averlo progettato.

I servizi per l'utente che fornisce il sistema operativo sono l'esecuzione del programma, supporto per i programmi di sviluppo e la sicurezza; per quanto riguarda il programma, il sistema operativo permette di gestire le risorse (come memoria e CPU) e accesso ai dispositivi (come i file).

Scheduling: consente di eseguire più programmi contemporaneamente e migliora l'utilizzo del processore.

In *user mode* non c'è la possibilità di eseguire alcune istruzioni macchina e si può

accedere solo ad una parte di memoria. In *system mode* si possono eseguire le istruzioni privilegiate e accedere ad aree di memoria protette (come nella parte de kernel).
L'approccio *uniprogramming* permette di ridurre l'utilizzo della CPU. Nel *multiprogramming* assegniamo alla CPU i vari processi dei programmi con tempi diversi. Il throughput è il numero di lavori eseguiti nell'unità di tempo. Quando i processi sono concorrenti, si usano gli interrupt.
I processi *I/O bound* aspettano i dati che arrivano e usano la CPU per fare richieste verso I/O, mentre i *CPU bound* sono programmi che fanno sopratutto conti. Si favoriscono i processi I/O bound cosi rilasciano la CPU subito mentre la CPU bound non la rilascia subito.
Time sharing: coordina i processi nella CPU e ogni 50 msec interrompe il processo (gestione "a ruota"); usa una singola CPU tra più processi e più utenti (anche se a velocità minore).

Le operazioni più importanti di un sistema operativo sono:
Processi: è un programma in esecuzione sul computer; si può definire anche come un'unità di attività caratterizzato da le risorse che necessità per andare avanti, come le regioni di memoria e i file aperti. È costituito da un programma eseguibile, i dati associati ad esso e il suo contesto di esecuzione (comprende le informazioni necessarie per gestire il processo).
Il thread è il valore del PC dentro la CPU assieme ai registri general purpose; le esecuzioni vengono eseguite uno dopo l'altro. Si può definire anche come un'entità che viene assegnata ed eseguita dal processore e ha senso solo se si associa ad un processo; in un sistema operativo moderno ce ne sono più di uno.
Una CPU può eseguire solo un processo in un certo istante, gli altri sono "congelati" (il loro stato è salvato da qualche altra parte) e per riprenderli si usa l'execution context.

Il PID è il numero che identifica un processo. I processi sono raggruppati in una struttura dati (come ad esempio la lista) e ognuno è costituito da context (struttura dati parte del sistema operativo), data (variabili globali, i dati del programma) e codice.

Gestione della memoria: è caratterizzato da processi isolati, allocazione e gestione automatica, supporto della programmazione (non è detto che sia tutto dentro un eseguibile) come le librerie (.dll per Windows e .so per Linux), protezione e controllo all'accesso e gestione della memoria di massa. Importante è la memoria virtuale (anche se non è strettamente necessaria): gli indirizzi che vede il programmatore non sono quelli fisici e si può usare molto più grande rispetto a quella reale e la memoria reale è usato solo per i processi veramente utili. Questo si può fare perché i sistemi usano il concetto di paging (ogni processo viene diviso in blocchi). La memory managment unit (MMU) è l'unità che realizza la virtualizzazione (si esegue in RAM): il kernel decide quali pagine usare per far partire un processo o quale processo ha bisogno di più o meno memoria. La memoria virtuale è una funzionalità che permette ai programmi di indirizzare la memoria da un punto di vista logico senza preoccuparsi di quella disponibile. Il file system implementa una memorizzazione a lungo termine con l'informazione memorizzata.

Sicurezza:: le principali operazioni di competenza dei sistemi operativi sono il controllo dell'accesso, del flusso dell'informazione e la certificazione.

Gestione di risorse: il kernel fornisce le risorse a chi ne ha bisogno in egual modo. Ci sono differenti classi di processi da gestire in modo diverso. Massimizza il throughput, si minimizza il tempo di risposta. Lo scheduling della CPU procede in due modi modi: short term (contengono i processi nella memoria principale e pronti ad essere eseguiti) e long term (un nuovo "lavoro" aspetta il processo"). Nell' I/O è gestito durante un interrupt. Ci sono tre fattori che devono essere rispettati: uguaglianza dei processi, il tempo di risposta differenziale ed efficienza.

Il sistema operativo gestisce un insieme di code, ciascuna delle quali è semplicemente una lista di processi in attesa di una risorsa.
Il sistema è strutturato come una serie di livelli, in cui vengono separate le sue funzioni a seconda della loro complessità, della scala temporale e del livello di attrazione: ogni livello è in relazione con il successivo livello più basso per eseguire funzioni più primitive e per nasconderne i dettagli, e fornisce servizi allo strato successivo di più alto livello. I livelli più bassi trattano eventi di durata assai più ridotta (interagiscono direttamente con l'hardware), mentre quelli più alti comunicano con l'utente e i tempi solo più lunghi.

In un sistema operativo moderno sono presenti i seguenti ambiti:
Microkernel: è un'architettura che assegna al kernel solo poche funzioni essenziali come la gestione dei diversi spazi di indirizzamento, la comunicazione fra i processi e la schedulazione di base. Gli altri servizi sono forniti da servizi chiamati processi. Questo approccio semplifica l'implementazione, aumenta la flessibilità ed è appropriato per un ambiente distribuito.

Multithread: è una tecnica in cui un processo, eseguendo un'applicazione, viene suddiviso in thread eseguibili simultaneamente. Un thread è un'unità di allocazione del lavoro; si può eseguire sequenzialmente e passare da uno all'altro in qualsiasi momento. Un processo è un insieme di uno o più thread. È utile per quelle applicazioni che effettuano un insieme di operazioni essenzialmente indipendenti che non necessitano di essere serializzati.

Multiprocessore simmetrico (SMP): è un'architettura in cui sono presenti più processori che condividono la stessa memoria e stesse funzioni I/O e sono interconnessi da bus; tutti i processori possono effettuare le stesse funzioni. I vantaggi sono di prestazione (il lavoro viene eseguito parallelamente), disponibilità

(tutti i processori effettuano le stesse funzioni e se si ferma uno non si blocca tutto quanto), crescita incrementale (si possono aggiungere processori) e scalabilità (serie di prodotti con prezzi diversi a seconda del numero di processori). L'esistenza di più processori è nascosta all'utente.

Programmazione orientata agli oggetti: permette di disciplinare il processo di espansione modulare dei kernel di piccole dimensioni; consente ai programmi di modificare un sistema operativo senza distruggere l'integrità del sistema.

PROCESSO

Ogni processo interagisce con il sistema operativo tramite *chiamata di sistema* (***system calls***): è come una chiamata di procedura, ma sono procedure nel kernel e non nel programma e possono inoltre eseguire operazioni "privilegiate"; si usano chiamate particolari per il system calls (speciali istruzioni o software di interrupt). Da un processo (in user mode) con la chiamata di sistema passa in kernel mode per essere eseguita e poi ritorna in user mode.
Le chiamate di sistema si classificano in I/O, allocazione di risorse (comunicazione tra i vari canali, memoria), controlli di processi e gestione di risorse.
Quando viene chiamata una system call, il sistema operativo può eseguirla subito o pospone la richiesta; se chiediamo per esempio di leggere sul disco un blocco di dati, poiché non si può soddisfare subito la richiesta perché i dati non ci sono e per non sprecare la CPU viene eseguito il blocco sul processo. Quando un processo viene messo in blocco si può scegliere quale processo attivare e questa scelta viene effettuata da una parte del kernel che si chiama ***short time cpu scheduler***.
Tra i processi c'è una relazione ad albero: ogni processo ha un genitore. Lo scheduling decide il prossimo processo da eseguire e subito dopo viene eseguito il dispatcher, che ripristina il suo contesto sulla CPU (gli assegna la CPU).

Ciclo di un processo: creato, eseguito (in modalità non privilegiata) e finito (viene de-allocata la memoria).
Creazione: quando un processo è aggiunto, il sistema crea una struttura dati per gestirlo e alloca lo spazio di indirizzamento. È determinato da quattro eventi: in risposta alla presentazione di un job (ambiente batch), quando un nuovo utente cerca di accedere al sistema (ambiente interattivo), per conto di un'applicazione e per sfruttare il parallelismo.

Terminazione: i sistemi devono fornire ai processi un modo per indicare il completamento, per esempio tramite istruzioni Halt, quando spegne il terminale, si esce da un'applicazione, un certo numero di errori o dal processo genitori che l'ha creato.

Quando il sistema operativo crea un nuovo processo, lo introduce nel sistema nello stato not-Running (il processo esiste ed aspetta di essere eseguito): da un momento ad un altro il processo corrente di interrompe e passerà da uno stato Running a uno not-Running mentre uno di quelli in attesa passerà a Running. Ogni processo deve essere rappresentato affinché il sistema operativo possa conservare una traccia (elenco di istruzioni eseguite). Il numero dei processi "Running" è dato dal numero della CPU.

In alcuni casi, alcuni processi not-Running non sono pronti per l'esecuzione mentre altri sono bloccati nell'attesa di operazioni di I/O. Per questo si creano i due stati Ready (un processo pronto per l'esecuzione) e Blocked (un processo non può essere eseguito finché non viene eseguito un altro processo). Lo stato new rappresenta quei processi appena creati ancora non immessi nello stato dei processi eseguibili (associa un identificatore al processo e alloca e crea tabelle necessarie per la sua gestione) e exit quello dei processi terminati.

Se un processo ha un'architettura interna che prevede più thread, si può avere più thread per ciascun core. Se ci sono tanti processi ready si procede per far girare tutti i processi in modo eguale. In Running ce ne può stare uno di processo, in block la maggior parte dei processi (sono in attesa) e in Ready i processi che escono da block. Quando un processo ha terminato o è fallito esce dal sistema.

Per i processi che rimangono in block e consumano memoria si ricorre allo stato "sospeso": si creano i "pronto/sospeso" e i "bloccato/sospeso" per inserire i processi non utilizzati. Le frecce tratteggiate rappresentano casi particolari (il sistema operativo chiede di sospendere un programma in esecuzione).

Le tabelle di memoria conservano traccia sia della memoria reale che di quella virtuale (allocazione e attributi di protezione). Ogni programma contiene una memoria sufficiente per mantenere i dati. L'esecuzione di un programma contiene uno stack usato per conservare traccie delle chiamate di procedura; ogni processo ha degli attributi assegnati dal sistema operativo per il controllo dei processi stessi e si chiama ***process control block***. Tutto quest'insieme fa parte dell'immagine del processo. Quest'ultima è mantenuta come blocco contiguo in memoria secondaria. Per gestire un processo almeno una parte dell'immagine deve essere mantenuta nella memoria principale poiché il sistema operativo deve conoscere la posizione di ciascun processo sul disco. Un'immagine si divide in blocchi detti o segmente (se di lunghezza variabile) o pagine (lunghezza fissa).

PCB: ***process control block*** -> contiene i dati di un processo e tutte le informazioni di cui abbiamo bisogno per interrompere il Running di un processo e riprendere un'esecuzione, creazione e gestione del sistema operativo e il supporto per più processi. Le informazioni del PCB si possono dividere in tre categoria: identificazione del processo (a tutti i sistemi operativi a ciascun processo è assegnato un unico identificatore numerico), informazioni sullo stato del processore (comprendono i contenuti dei registri del processore, poiché quando un processo è interrotto tutte le informazioni dei registri devono essere salvate in modo da poterla ripristinare) e informazioni di controllo del processo (informazioni supplementari per controllare i vari processi). Il PCB può contenere informazioni per la strutturazione dei dati.

L'insieme dei PCB costituisce lo stato del sistema operativo. Alle informazioni del PCV accedono le routine del sistema operativo; i problemi che possono nascere da questa cosa sono nella sicurezza.

In Unix il PCB forma un albero e all'interno viene scritta l'informazione dei processi in coda.

Per creare un processo bisogna:

1) assegnare al nuovo processo un identificatore è unico ed aggiungere una nuova entry nella tabella principale;

2) allocazione della memoria per un processo comprendendo tutti gli elementi dell'immagine del processo (il sistema deve sapere lo spazio necessario per gli indirizzi e per lo stack utente); questi valori sono assegnati per difetto;

3) inizializzazione di un PCB: contiene l'ID del processo. La parte di informazioni sullo stato del processore viene inizializzata con la maggior parte dei valori a zero (tranne il PC e i puntatori dello stack) mentre la parte relativa alle informazioni di controllo del processo è inizializzata basandosi su valori standard per difetto e sugli attributi richiesti per questo processo;

4) settiamo i collegamenti (ogni processo nuovo deve essere messo nella coda dei Ready);

5) creiamo o espandiamo un'altra struttura dati.

Tutto questo viene fanno dalla system call nel kernel. Lo standard POSIX standardizza varie cose in Unix come ad esempio le system call (la FORK di Unix crea un processo figlio).

La struttura dati relativa al PCB consiste in:

Proces Tables: un entra per ogni programma e contiene il minimo numero di informazioni necessarie per attivare un processo.

Memory Tables: allocazioni di memoria principale e secondaria per il processo e protegge gli attributi per l'accesso alla regione di memoria condivisa; inoltre contiene le informazioni necessarie per la gestione della memoria virtuale.

I/O Tables: ogni dispositivo può essere impegnato in un certo processo; è una locazione nella memoria principale inizialmente usata come risorsa o destinatario del trasferimento I/O.

File Tables: i file sono situati nella memoria secondaria; contengono lo stato corrente, gli attributi e molte di queste informazioni sono mantenute nel sistema di gestione dei file.

Un processo può girare in kernel mode o in user mode. Il *mode switch* mette la CPU da kernel a utente o viceversa: da user a kernel è data da un'interrupt o una system call, la CPU passa in modalità "privilegiata", quando si salva lo stato della CPU. Da kernel a user si ha quando il kernel decide di riprendere ed eseguirlo, la CPU passa in modalità non privilegiata oppure quando riprende tutta o parte dello stato della CPU.

Il *process switch* assegna alla CPU un differente processo; un cambio di processo può verificarsi ogni volta che il sistema operativo ottiene il controllo dal processo corrente in esecuzione e funziona in kernel-mode. Modifica la struttura dati del sistema operativo: un processo viene dislocato in Ready o block a seconda dell'utilizzo. Le tabelle di memoria della CPU con il cambiamento di processo si stravolgono, non vedono più la locazione di memoria del processo 1 ma quella del processo 2.
Avviene uno switch quando:

1) interruzione di clock (il sistema operativo determina se il processo corrente in esecuzione sia stato eseguito per il quanto di tempo massimo permesso e se affermativo il processo passa allo stato ready e ne è selezionato un altro);
2) system call (un processo utente in esecuzione può effettuare una richiesta di I/O e questa chiamata trasferisce il controllo ad una routine (parte del sistema operativo. L'uso di una system call comporta il passaggio del processo utente allo stato Blocked).
3) I/O interrupt (il sistema operativo determina quale operazione di I/O sia avvenuta. Nel caso ci siano processi che attendono, il sistema operativo li passa nello stato Blocked o Ready (Blocked-Suspend →Ready-Suspend), quindi decide se riprendere l'esecuzione del processo corrente nello stato Running oppure passare a un

altro processo Ready con priorità più elevata).

4) Trap (determina se un errore sia fatale o meno e in caso affermativo il processo in esecuzione passa allo stato di Exit e avviene un cambio di process).

5) Fault di memoria (il processore incontra un riferimento a un indirizzo di memoria virtuale, per una parola che non si trova in memoria principale. Il sistema operativo carica il blocco di memoria contenente il riferimento dalla memoria secondaria a quella principale; soddisfatta la richiesta di caricamento, che è un'operazione di I/O, il sistema operativo effettua un cambio per riprendere l'esecuzione di un altro processo: il processo che ha generato il fault di memoria è posto nello stato Blocked e, caricato in memoria il blocco richiesto, quel processo passa allo stato Ready).

Quando il processore incontra un'interruzione, per prima cosa salva il contesto del programma in esecuzione, imposta il PC e effettua il passaggio da modalità utente a quella kernel in modo che l'elaborazione dell'interruzione attraverso un programma apposito possa contenete istruzioni privilegiate.

Un cambio di modo non determina un cambio di processo, ma viceversa si perché implica cambiamenti sostanziali. Per effettuare un completo cambio di processo:

- Si salva il contesto del processore compresi i registri;
- Si aggiorna il PCB corrente e dello stato del processo corrente nello stato Running e occorre aggiornare tutte le altre informazioni rilevanti.
- Si sposta il PCB nella coda appropiata.
- Si seleziona un altro processo per l'esecuzione.
- Si aggiorna il PCB del processo selezionato (si passa allo stato Running) e delle strutture dati per la gestione della memoria.
- Ripristino del contesto del processore relativo al processo che aveva precedentemente abbandonato lo stato Running ricaricando i vecchi valori degli indirizzi.

Non-process kernel (a): il kernel è eseguito al di fuori di ogni processo: il kernel ha il suo spazio di memoria (non è un processo del sistema operativo qualunque ed è inefficiente poiché si devono riconfigurare le memory table) e implementa trucchi (tricks) per accedere alle immagini del processo ed è obsoleta; questa zona di memoria serve per far si che il sistema operativo controlli le chiamate o ripristinare processi utenti interrotti, altrimenti potrebbe completare l'operazione di salvataggio dell'ambiente del processo, fare lo scheduling e sceglierne un altro. È applicabile solo ai programmi utenti.

L'*esecuzione con i processi utente (b):* il kernel appare nella struttura di memoria di ogni processo e non è necessaria la riconfigurazione della memory table e quindi è più efficiente (il kernel ha bisogno solo della modalità switch). Si esegue quasi tutto il software del sistema operativo nel contesto di un processo utente. Quando c'è una system call o un interrupt il kernel è già pronto che sono gestite da uno stack, il processo è in modalità kernel e il codice e i dati del sistema operativo sono nello spazio degli indirizzi condiviso e sono comuni a tutti i processi utente. Ogni processo ha una sua immagine che contiene il kernel stack, il kernel program e il kernel data; quest'ultimi due sono condivide da tutte le immagini, poiché il kernel mode ha bisogno di leggerle e scriverle. Quando ci sono interruzioni si passa in modalità kernel e il controllo passa al sistema operativo. Questo cambio di modo però non modifica il processo corrente: questo è utile perché se il sistema operativo completando il lavoro stabilisce che un processo corrente dovrebbe continuare l'esecuzione, allora un cambio di modo ripristina il programma interrotto. Quindi un programma utente è stato interrotto per utilizzare una routine del sistema operativo e quindi ripristinarlo senza dove ricorrere ai cambi di processo.

Sistemi operativi basati sui processi *(c)*: si può implementare il sistema operativo come una serie di processi di sistema; il software che è parte del kernel è eseguito in kernel mode cosi le sue principali funzioni sono organizzate come processi separati. I vantaggi sono che si impone una disciplina della progettazione dei programmi con interfacce minimali e alcune funzioni non critiche sono implementate come processi separati (per esempio un programma che registra il livello delle risorse) e alcuni dei sistema operativi possono essere affidati a processori dedicati migliorando le prestazioni. Per lo scambio viene sfruttato una piccola quantità di codice.

UNIX

Unix esegue la maggior parte del sistema operativo nell'ambiente di un processo utente e sfrutta due categorie sia due modalità che due categorie di processi, cioè utente e kernel: i processi di sistema eseguono codice di sistema operativo in modalità kernel per eseguire funzioni amministrative, mentre un processo utente lavora per eseguire funzioni di utilità. Unix impiega due stati Running per indicare se il processo è eseguito in modalità utente oppure kernel ed effettua una distinzione fra lo stato di Ready e di preempted (prerilasciato) anche se formano un'unica cosa. Quando un processo è in esecuzione in modalità kernel, non può essere prerilasciato e ciò rende UNIX inutilizzabile per l'elaborazione in tempo reale.

In UNIX ci sono due processi "unici": processo o e processo 1. Il primo è una struttura dati predefinita creato quando parte il sistema e gestisce il trasferimento sul disco; il secondo invece è generato dal primo e viene chiamato init, cioè è l'antenato di tutti i processi che si creano (crea per i nuovi processi un processo utente).

L'immagine del processo in UNIX è caratterizzato da un contesto al livello utente (contiene tutti gli elementi fondamentali di un programma utente ed è diviso in aree testo di sola lettura per memorizzare le istruzioni, e dati), un contesto dei registri (le informazioni di stato del processore sono memorizzate qui) e un contesto al livello di sistema (contiene le informazioni rimanenti di cui il sistema operativo ha bisogno per gestire i processi).

La creazione dei processi in UNIX è fatta mediante system call al sistema dette fork(); il sistema operativo, quando c'è una richiesta di fork, esegue le seguenti funzioni in modalità kernel:

- alloca uno spazio nella tabella dei processi per il nuovo processo;
- assegna un ID di processo al figlio (unico nel sistema);
- fa una copia dell'immagine del processo genitore, ad eccezione della memoria condivisa;

- incrementa i contatori dei file del genitori per registrare che ora anche il nuovo processo possiede questi file;
- assegna al processo figlio lo stato Ready;
- ritorna l'identificatore ID del figlio al genitore e il valore 0 al figlio.

Sia il genitore che il figlio eseguono la stessa porzione di codice, ma la differenza sta nel valore di ritorno della fork (0 se viene eseguito il processo figlio ed esegue un salto all'opportuno programma utente per continuare l'esecuzione, altrimenti se è diverso da 0 è in esecuzione un programma genitore e la linea di esecuzione continua). Quando il kernal ha compiuto queste funzioni, può rimanere nel processo genitore (il controllo ritorna al modo utente nel punto di chiamata della fork), trasferire il controllo al processo figlio (il processo figlio comincia ad eseguire dallo stesso punto del codice del genitore, cioè dalla chiamata della fork) oppure ad un altro processo (genitore e figlio lasciati nello stato di Ready.

GESTIONE DELLA MEMORIA

La gestione della memoria deve:

ricollocare: la memoria principale disponibile è di solito condivisa tra un certo numero di processi e non si può sapere in anticipo dove un programma deve essere allocato; per questo si permette alle operazioni di trasferimento da e per la memoria secondaria di muovere il programma nella memoria principale. Il sistema operativo gestisce la memoria ed è responsabile del caricamento del processo nella memoria principale; inoltre il processore si deve occupare dei riferimenti alla memoria all'interno del programma. L'hardware e il software del sistema operativo devono essere in grado di tradurre i riferimenti alla memoria trovati nel codice del programma in indirizzi fisici effettivi di memoria, riflettendo l'allocazione corrente del programma nella memoria principale. La memoria viene allocata dinamicamente su richiesta. La memoria viene gestita da più processi divisi.

proteggere: ogni processo deve essere protetto contro interferenze indesiderate da parte di altri processi, siano esse accidentali o intenzionali; perciò i programmi degli altri processi non dovrebbero avere riferimenti a locazioni di memoria di un processo senza permessi. Poiché la locazione di un programma nella memoria principale è sconosciuta, è impossibile controllare gli indirizzi assoluti a tempo di compilazione. Tutti i riferimenti alla memoria generati da un processo devono essere controllati a tempo di esecuzione per assicurarsi che si riferiscano solo allo spazio di memoria allocato per quel processo. Questi requisiti devono essere soddisfatti più dall'hardware che dal software poiché il sistema operativo non può conoscere tutti i riferimenti di memoria che un programma fa.

condivisione: i processi che cooperano ad uno stesso compito possono avere bisogno di condividere l'accesso alla stessa struttura dati; quindi il sistema per la gestione della memoria deve permettere l'accesso controllato ad aree condivise della memoria senza compromettere la protezione.

organizzazione logica: i programmi sono organizzati in moduli, di cui alcuni non modificabili (read only o execute only). Il fatto che il sistema operativo e l'hardware si possono occupare dei programmi utente e dei dati ha come vantaggi i moduli possono essere scritti e compilati indipendentemente dando ad ogni modulo un grado di protezione diverso ed essere condivisi.
organizzazione fisica: la memoria è organizzata in principale e secondaria: la prima permette un accesso rapido ma ad un costo relativamente alto ed è volatile, mentre la seconda è più lenta, ma meno costosa e non è volatile. L'organizzazione del flusso tra queste due memorie è fondamentale (è meglio non assegnata al programmatore, ma fatta dal sistema).
Ci sono varie tecniche della gestione della memoria che non coinvolgono la memoria virtuale; il compito principale del gestore della memoria è portare i programmi nella memoria principale perché il processore li esegua coinvolgendo la memoria virtuale, a sua volta basata su segmentazione e paginazione. Le tecniche più semplici che non la coinvolgono sono la partizione, la paginazione semplice e la segmentazione semplice.
Partizionamento fisso: si suppone che il sistema operativo occupi una porzione fissa della memoria principale e che il resto della memoria principale sia disponibile per essere utilizzata da molti processi: per gestire la memoria lo schema più semplice è la partizione. Con il partizionamento fisso possiamo far uso di partizioni di uguali dimensioni: qualunque processo la cui dimensione è minore o uguale alla dimensione della partizione può essere caricato in qualunque partizione disponibile e se tutte sono piene e nessun processo è nello stato Ready o nello stato Running, il sistema operativo può trasferire un processo fuori da una qualunque partizione e caricarvici un altro processo (lo schedulatore decide quale processo trasferire). Le difficoltà che si presentano usando partizioni fisse della stessa dimensione sono che ci possono essere programmi troppo grandi per la partizione (un programmatore deve fare uso di overlay, cioè viene caricata una parte del programma sulla memoria principale e che poi va a sostituire una parte del programma), facendo si che quando c'è bisogno di un

modulo non presente in memoria, il programma dell'utente deve caricare quel modulo nella partizione dedicata al programma sovrapponendolo con altri programmi, e l'utilizzazione della memoria principale è inefficiente per il fatto che un programma occupa un'intera partizione (indipendentemente dalle dimensioni di esso) e quindi c'è rischio di spreco di memoria se il programma occupa uno spazio più piccolo rispetto alla partizione (frammentazione interna).
Finché c'è una partizione disponibile, un processo può esservi caricato e poiché tutte le partizioni hanno le stesse dimensioni, non ha importanza quale partizione è usata; se tutte le partizioni fossero occupate da processi che non sono Ready, uno di questi processi deve essere trasferiti per far posto ad un nuovo processo. Usando partizioni di diversa dimensione, è possibile scegliere tra due metodi di assegnazione: il primo consiste nell'assegnare ogni processo alla partizione più piccola che lo contenga (è necessaria una coda di schedulazione per ogni partizione per mantenere i processi trasferiti all'esterno destinati a quella partizione) minimizzando la frammentazione all'interno di una partizione comportando però che c'è il rischio che partizioni non vengano utilizzate; il secondo consiste nello scaricare la partizione più piccola che lo contenga, ma se sono tutte occupate si deve scaricare un processo sulla memoria secondaria (si possono dare priorità o preferenze al trasferimento). L'uso di partizioni con diversa dimensione fornisce una certa flessibilità e richiedono un sovraccarico minimo al software del sistema operativo; ha degli svantaggi come il fatto che il numero di partizioni limita il numero di processi attivi nel sistema e che le dimensioni della partizione sono predefinite al momento della generazione del sistema e quindi i piccoli job non utilizzeranno efficientemente lo spazio della partizione.
Partizionamento dinamico: si usano partizioni in numero e lunghezza variabile; quando un processo nella memoria principale, gli è allocata tanta memoria quanta richiesta e mai di più. Quando un processo è più grande di un blocco disponibile, viene sostituito con un altro processo che va a finire in memoria principale; questo metodo comporta ad avere tanti piccoli buchi in memoria portando cosi la memoria ad essere sempre più

frammentata e l'utilizzo a peggiorare (frammentazione esterna). Per questo si ricorre alla compattazione nella quale il sistema operativo sposta i processi in modo tale che essi siano contigui e tutta la memoria libera sia riunita in un blocco; ha un elevato costo computazionale e implica una capacità di rilocazione dinamica (sposta un programma da una regione ad un'altra senza invalidare i riferimenti alla memoria. Esistono tre algoritmi di allocazione, nella quale ognuno si limita a scegliere tra i blocchi liberi di memoria principale che sono uguali o più grandi del processo che deve essere immesso in memoria; tutti e tre dipendono necessariamente dall'esatta sequenza dei trasferimenti dei processi e dalla loro dimensione:

- ***best fit***: sceglie il blocco che è più vicino in dimensione alla richiesta;
- ***first- fit***: scandisce la memoria dall'inizio e sceglie il primo blocco disponibile sufficientemente grande;
- ***next- fit***: scandisce la memoria partendo dalla locazione dell'ultima allocazione e sceglie il blocco disponibile successivo e sufficientemente grande.

Ci sarà un momento in cui tutti i processi residenti nella memoria principale saranno bloccati e la memoria sarà insufficiente per un ulteriore processo anche dopo la compattazione: per evitare spreco di tempo nell'attesa in cui un processo attivo si sblocchi, il sistema operativo tirerà fuori dalla memoria uno dei processi per fare spazio ad un nuovo processo o ad un processo in stato Ready-Suspend.

Buddy System: si trova su Unix ed è un compromesso tra il partizionamento fisso e quello dinamico: nel buddy system i blocchi di memoria disponibili hanno dimensione 2^k con k compreso tra due valori L e U (2^U è la dimensione dell'intera memoria disponibile e di solito è usato per allocare pagine per i processi). Il buddy system è un sistema che alloca abbastanza efficientemente.

La teoria de buddy system parte dal fatto di avere una quantità di memoria potenza di 2^U e la richiesta è di una certa quantità s ed è in grado di gestire blocchi di celle di log2 s bytes. Se la richiesta di dimensioni s si trova tra $2^{i-1} < s <= 2^i$, il blocco di

lunghezza 2_i è allocato; un blocco da 2_i può essere diviso in due uguali "buddies" di 2^{i-1} bytes; se non troviamo un blocco delle dimensioni necessarie (più grande o uguale a s), li divido finché non arrivo alla taglia giusta.

Il buddy system mantiene una lista L_i (i=1,...U) di blocchi da 2^i non allocati: le operazioni sono la divisione (rimuove il buco da $L_{(i+1)}$ li divide e li mette nei due buddies di dimensione 2^i) e il coalescing (rimuove due buddies non allocate da L_i e li riunisce in un singolo blocco nella lista in $L_{(i+1)}$. Nell'esempio, allochiamo lo spazio di memoria più vicino alla richiesta (la potenza di due subito superiore ai byte richiesti). Due blocchi si definiscono buddy se due blocchi derivati da un taglio della stessa dimensione si possono riunire tornando al blocco precedente al taglio.

L'algoritmo di allocazione non si basa sull'albero, ma sulle liste perché dovrebbero essere più facili da gestire. Nei sistemi moderni, la memoria virtuale basata sulla paginazione e segmentazione è superiore del buddy system, il buddy trova applicazione nei sistemi paralleli come un mezzo efficiente per l'allocazione e il rilascio dei programmi eseguiti in parallelo (una forma modificata di buddy è usata nel kernel di Unix). In presenza di una richiesta di allocazione di dimensioni K tale che $2^{i-1} < K <= 2^i$ si usa un algoritmo ricorsivo per trovare un blocco libero di dimensioni 2^i:

```
procedure prendibuco (i);
begin
    if i = U + 1 then fallimento;
    if i_list vuota then begin
        prendibuco (i+1);
        dividi il buco in due compagni;
        metti i compagni nella lista i_list
    end;
    prendi il primo buco in i_list
end
```

Get_hole: prendi un blocco libero (di input gli passo i e significa che chiedo un blocco 2^i e come precondizione i<u e se non rispettata non gli posso passare il blocco) e come output un blocco c di dimensioni 2^i (come post-condizione Li con deve contenere c e nessuna lista deve contenere i blocchi sovrapposti). Se L_i contiene qualcosa, il blocco c è il primo blocco libero in L_i, lo rimuovo e lo ritorno come blocco destinato all'allocazione. Se L_i è vuota, mi cerco un blocco vuoto di grandezza 2^{i+1} ricorsivamente, divido b in due buddies b1 e b2 (b è grande il doppio di quello che ci serve) e viene messo in L_i. Per rappresentare tutte le liste libere senza la malloc o kmallor ci sono due modi: 1) utilizzare spazi liberi per tenere le strutture dati 2) tenere strutture dati tipo array in zone di memoria che possono stare dentro o fuori il MB (riferendoci all'esempio di prima).

Rilocazione: quando si usa uno schema di partizionamento fisso, un processo è sempre assegnato alla stessa partizione, nel quale la partizione selezionata al momento del caricamento di un nuovo processo è la stessa in cui allocarlo ogni volta che rientra in memoria; quando un processo per la prima volta è caricato, tutti gli indirizzi relativi in memoria presenti nel codice sono sostituiti dagli indirizzi assoluti della memoria principale, determinati dall'indirizzo base del processo caricato. Nel caso di partizioni di uguale dimensione e nel caso di una sola cosa di processi per partizioni diverse, un processo può occupare partizioni diverse nel corso della sua vita. Quando un'immagine del processo è creata per la prima volta, è caricata in una partizione della memoria principale; in seguito il processo può essere trasferito fuori dalla memoria, ma quando è successivamente reinserito, può essere assegnato ad una partizione diversa dalla precedente e lo stesso accade nel partizionamento dinamico. Le locazioni delle istruzioni e dei dati cui un processo fa riferimento non sono fisse, ma cambieranno ogni volta che il processo è caricato in memoria o spostato. Gli indirizzi in un programma possono essere *logici* (riferimenti a una locazione di memoria, indipendenti dall'assegnazione attuale dei dati in memoria e devono essere

tradotti in indirizzo fisico prima di essere tradotti in memoria), *relativi* (casi particolati di indirizzi logici in cui l'indirizzo è espresso come una locazione relativa ad un punto conosciuto, solitamente l'inizio del programma) e *fisici* (sono le locazioni effettive nella memoria principale). I supporti hardware per la rilocazione sono i *registri base* (viene caricato l'indirizzo iniziale del processo nella memoria principale)e i *registri limite* (indicano la locazione finale del programma); questi valori devono essere impostati quando il programma è caricato in memoria o quando l'immagine del processo è trasferita dal disco. Durante l'esecuzione del processo, si incontrano indirizzi relativi con il contenuto del registro di istruzione, gli indirizzi delle istruzioni che si incontrano nelle istruzioni di trasferimento di controllo e di chiamata, e gli indirizzi dei dati che si incontrano nelle istruzioni di load e store; ciascuno di tali indirizzi passa attraverso due manipolazioni da parte del processore (prima viene sommato il valore contenuto nel registro base all'indirizzo relativo per produrre un indirizzo assoluto e poi l'indirizzo risultante è confrontato con il valore del registro limite e se l'indirizzo è dentro ai limiti è dentro ai limiti l'esecuzione dell'istruzione può procedere, altrimenti si genera un'interruzione al sistema operativo.

Paginazione: il partizionamento fisso e variabile usano la memoria in modo inefficiente poiché provocano rispettivamente una frammentazione interna e una esterna. Si supponga che la memoria principale sia partizionata in piccoli blocchi di uguale dimensione fissa e che ogni processo sia a sua volta diviso in piccoli pacchetti della stessa dimensione chiamati *pagine* che può essere assegnato al blocco di memoria disponibile (*frame*); lo spazio di memoria sprecato per ogni processo è dovuto alla frammentazione interna e non esterna limitata ad una porzione dell'ultima pagina del processo. l'idea è quella di partizionare la memoria (sia del processo che fisica) in tante parti tutte della stessa grandezza (pagine); i processi hanno le pagine, quelle fisiche le chiamiamo frame (sia frame che pagine sono numerate). Ogni

processo ha una page table e contiene la locazione dei frame per ogni pagine nel processo e l'indirizzo di memoria si compone di un numero di pagine. Nel caso di una partizione semplice, un indirizzo logico rappresenta la locazione di una parola all'inizio del programma e il processore lo traduce in indirizzo fisico (fatta dall'hardware del processore che accede alla page table del processo corrente e in presenza di un indirizzo logico la page table viene usata per produrre un indirizzo fisico). La memoria virtuale fa tutto quello che fa la paginazione ma questi frame non sono tutti i frame del processo perché qualcuno può stare sul disco. È simile al partizionamento fisso, ma con la paginazione le partizioni sono abbastanza piccole e un programma può occupare più partizioni (anche non contigue). La paginazione è invisibile al programmatore.

Segmentazione: i programmi e i dati ad essi associati sono divisi in un certo numero di segmenti (non devono essere tutti di una stessa lunghezza, ma per loro c'è una lunghezza massima); un indirizzo logico che usa la segmentazione è composto da il numero del segmento e un offset. La segmentazione è simile al partizionamento dinamico: tutti i segmenti vengono caricati in memoria per eseguirlo, ma differiscono nel fatto che un programma può utilizzare più di una partizione (anche non contigue) eliminando cosi la frammentazione interna (ma non quella esterna anche se è minore). La segmentazione è visibile ed è conveniente per organizzare programmi e dati. Il programmatore o il compilatore assegneranno il programma e i dati a segmenti diversi e i programmi possono essere ulteriormente frammentati in altri segmenti con lo svantaggio dell'attenzione che si deve date alla dimensione massima del segmento. Non esistono relazioni tra indirizzo logico e fisico. Uno schema semplice per la segmentazione usa una tavola dei segmenti per ogni processo e una lista dei blocchi liberi nella memoria principale. Ogni entry indica l'indirizzo iniziale del segmento corrispondente nella memoria principale e fornirà anche la lunghezza del segmento per assicurare che non siano usati indirizzi non validi. Quando un processo diventa

Running, l'indirizzo sulla tavola dei segmenti è caricato in un registro speciale usato dall'hardware del gestore della memoria. Si consideri un indirizzo di n + m bit (n sono il numero del segmento -sinistra-, m sono l'offset -destra-): prima si estrae il numero del segmento dagli n bit più a sinistra dell'indirizzo logico e si usa il numero del segmento come un indice nella tabella dei segmenti del processo per trovare l'indirizzo fisico iniziale del segmento. Si confronta l'offset con la lunghezza del segmento e se è maggiore della lunghezza l'indirizzo non è valido; l'indirizzo fisico desiderato è la somma dell'indirizzo fisico iniziale del segmento con l'offset.

ESERCIZI SU PROCESSI E GESTIONE DELLA MEMORIA

ESERCIZIO

Considera N processi che non eseguono system call (cpu bound). Ciascun processo ha bisogno per concludere l'esecuzione di 1 s di CPU. Nel sistema non ci sono processi. Gli N processi sono inizialmente in stato "ready". Il time slice per tutti è di 3 s.

1) Quante volte verrà eseguito il dispatcher prima che tutti gli N processi terminano l'esecuzione?

Risposta: N volte perché il dispatcher viene eseguito ogni volta hce termina un processo per assegnare le ad un nuovo processo.

2) Supponi che il time slice sia per tutti di 100 ms. Quante volte verrà eseguito il dispatcher prima che tutti gli N processi terminino l'esecuzione?

Risposta: servono 10 * 100 ms per far terminare un processo; quindi per N processi ci servirà 10N.

3) Quanti sono i process switch?

Risposta: 10*N chiamate al dispatcher (= 10* N process switch)

4) Quanti sono i mode switch?

Risposta: 10*N * 2 (ogni passaggio da kernel mode a user mode).

Workdlow:

1.Processo in esecuzione in User Mode

2.Mode switch (per 100 ms) in Kernel Mode

3.Eseguire dispatcher

4.Mode Switch

5.Eseguo un altro processo.

ESERCIZIO

Supponi che nel sistema ci siano due processi P1 e P2. P1 fa N chiamate di sistema, ciascuna bloccante. P2 non fa alcuna chiamata di sistema e terminerà dopo P1. Supponi che la CPU venga restituita a P1 non avendo P1 va in stato ready (preemption). Conta i mode switch e i process switch nel modello "kernel execution within process" fino al soddisfacimento dell'n-esima chiamata di sistema.

Risposta: numero process switch: 2N | numero mode switch: 4N.

ESERCIZIO BUDDY SYSTEM

Spazio di memoria da allocare 1 MB. Inizialmente nessun blocco allocato.
Viene richiesto un blocco A di 64 Kb. Quante volte viene richiamata la procedura get_hole()?
Com'è l'albero che rappresenta sistema dopo l'allocazione? Cosa contengono le liste Li?

Risposta:

get_hole (64K)

(richiama) get_hole (128K)

(richiama) get_hole (256K)

(richiama) get_hole (512K)

(richiama) get_hole (1M)

ESERCIZIO

Considera due indirizzi di due blocchi b1 e b2 della stessa grandezza che vengono identificati dall'indirizzo del primo byte. Dai un metodo basato sulla rappresentazione binaria di b1 e b2 per capire se sono due buddies.

Risposta:

i = 20 → 1 Mb

i = 19 → 512 Kb

i = 18 → 256 Kb

i = 17 → 128 Kb

U = 20 → 2^20 = 1 Mb = memoria totale. Un buddy viene identificato da Ui bit di b1 e b2.

Si riconosce un buddy se facendo uno shift a destra sia di b1 che di b2 e poi confrontandoli risultano uguali (si fa il confronto tra i padri).

Per esempio: 000 e 001 sono buddy? Entrambi facendo lo shift a destra fanno 00 quindi sono buddy.

Coalescing significa riunire due buddy in un unico blocco. Esistono due strategie: riunirli quando si chiede più memoria di quella che si ha (lazy) oppure appena ho due blocchi liberi.

ESERCIZIO

Si hanno 32 bit per l'indirizzo fisico e 32 bit per l'indirizzo logico con frame di 4 Kb.

1) Quanti bit si usano per l'offset?

Risposta: 12 bit perché il frame è di 4 Kb (2^12).

2) Con quanti bit identifico un frame?

Risposta: Se si ha un sistema a 32 bit la memoria totale è di 2^32, cioè 4 Gb. Se si usano 12 bit per l'offset, ne rimangono 20 per il frame.

3) Con quanti bit identifico una pagina?

Risposta: 20 bit.

4) Quante pagine abbiamo nello spazio di indirizzamento logico?

Risposta: 2^20.

5) Se un processo usa tutte le pagine quanto è grande la page table?

Risposta: Se usa tutte le pagine è grande 2^20.

MEMORIA VIRTUALE

La struttura della memoria virtuale è caratterizzata in regioni (uno spazio di indirizzo di un processo è diviso in regioni per codice, datti e condivisione); possono accedere alla memoria usando linguaggi macchina, creazione e cambiamento di regioni (tramite system call) e condividere (librerie, threads, fork). Per esempio digitando cat/proc/pid/maps si può chiedere al kernel informazioni sulle regioni. Ogni processo ha un'immagine logica e un indirizzamento logico. La fork crea un figlio identico al padre, cioè condivide gli stessi frame.

I vantaggi di andare in memoria virtuale sono:

- posso far girare processi che occupano di più della memoria;
- la parte dedicata ai processi è tipicamente minore della somma degli spazi dei processi;
- se ho tanti processi, è più facile avere un processo ready e fare un buon uso della CPU (è possibile lanciare più processi);
- resident set: è la parte di processo che sta nella memoria principale in un certo instante.

La paginazione è il tipico supporto hardaware: il riferimento alla memoria è traslato dinamicamente dentro l'indirizzo fisico nel run time (il flag di not present mi genera un interrupt se vado a leggere una pagina che non è più in RAM). Un processo può essere caricato in memoria principale e scaricato in modo da occupare diverse regioni di memoria in diversi momenti durante l'esecuzione e può essere suddiviso in un certo numero di pezzi: se entrambe queste caratteristiche sono presenti, è necessario che tutte le pagine o tutti i segmenti di un processo siano nella memoria principale durante l'esecuzione (se il pezzo che contiene l'istruzione successiva e il pezzo che contiene la successiva locazione dei dati sono nella memoria principale, allora

l'esecuzione può procedere). Quando un processo comincia l'esecuzione, tutto prosegue per il verso giusto finché i riferimenti a memoria sono in locazioni presenti nel resident set. Se il processore incontra un indirizzo logico che non è nella memoria principale, genera un'interruzione, che indica un fallimento dell'accesso in memoria; il sistema operativo mette il processo interrotto in uno stato blocked e prende il controllo. Il sistema operativo emette una richiesta di lettura da disco per sapere l'indirizzo logico che ha causato il fallimento e successivamente viene mandato in esecuzione un altro processo mentre l'I/O di disco viene eseguito. Una volta caricato, viene generato un interrupt di I/O restituendo il controllo al sistema operativo che riporta allo stato ready il processo bloccato. Più processi possono essere mantenuti nella memoria principale poiché si caricano solo alcuni pezzi del processo.

Il supporto per le pagine che non stanno in memoria ed è gestita da un interrupt chiamato page fault; quest'ultimo è generato quando un accesso ad un indirizzo di memoria non è in memoria e mette il processo nello stato di block. Il processo aspetta la pagina dal disco (che equivale ad una richiesta di I/O Block), il processo che genera la pagina sta in uno stato di blocco, un altro processo sta in una fase di scheduling e ci sta un interrupt quando l'I/O ha completato e fa si che il processo vada nello stato ready. I page fault si dividono in major (quando un input da disco è necessario) e minor (quando l'input da disco non è necessario).

Thrashing: quando la memoria fisica è troppo piccola rispetto al processo di memoria in corso, bisogna evitare il thrashing (il sistema non fa altro di stare in attesa di pagine da leggere da disco e non può fare nulla). Tutta la memoria principale è occupata dai pezzi dei processi, in modo che il processore e il sistema operativo abbiano accesso diretto a più processi possibili, e quando il sistema operativo carica un pezzo di processo nella memoria principale, deve prima toglierne un altro (se ne toglie uno che sta per essere usato, dovrà recuperarlo immediatamente, producendo questo fenomeno).

Fetch policy: determina quando una pagine dovrebbe essere portata in memoria; prepaging prende molte delle pagine di cui ha bisogno: se la predizione è buona, le pagine sono pronte in memoria quando ne hanno bisogno (è raramente usato); demond paging prende solo le pagine nella memoria quando ne ha realmente bisogno come il salvataggio di memoria, in molte page fault durante la partenza di un processo (è molto usato); vengono allocate zero pagine, solo page fault vengono allocate; i file eseguibili non vengono allocati, ma solo tramite i page fault.

Placement Policy: determina dove nella memoria reale, una parte di processo ha bisogno di stare; è importante nei sistemi di sola segmentazione (vede gli approcci delle allocazioni di memoria e le frammentazioni esterne).

Evition Policy: la startegia usata dal sistema operativa per scegliere le pagine da levate nell'RS nei processi; una buona pagina da evitare non sarà accessibile in un futuro vicino; la strategia per evitare è il modo in cui il sistema operativo usa la predizione del futuro; c'è una lunga ricerca nel passato. Il trashing è presente.

Cleaning Policy: prima evitare di modificare pagine prima che sono scritte nel disco; una pagina è scritta solo quando è stata selezionata per la sostituzione; il precleaning avviene quando le pagine sono scritte in lotti prima della scelta dei sostituti oppure sono inattive.

Il sistema si vuole tenere un certo di frame liberi per riassegnarli; i frame di pagine evitate sono aggiunte ad uno di due liste, una per le pagine libere per essere riassegnate (free-clean list) e una una prima da ripulire e poi riassegnare(free-dirty list); i frame che si trovano nel clean sono pronte per essere riassegnate, mentre quelli presenti nel dirty vengono scritti in lotti e messi nella lista clean. La strategia di cleaning determina quando una pagina modificata deve essere scritta nella memoria secondaria. Il cleaning a richiesta fa si che una pagina è scritta nella memoria secondaria solo quando è stata scelta per essere sostituita (minimizza il numero delle pagine scritte, ma significa che un processo che ha fault di pagina può dover aspettare due trasferimenti di pagine per poter essere sbloccato diminuendo cosi

l'uso del processore). Quella di precleaning scrive le pagine modificate prima che i loro frame siano necessari, cosi le pagine possono essere scritta a gruppi (rimane nella memoria principale fino a che l'algoritmo di sostituzione di pagine non ordina che sia rimossa e consente la scrittura delle pagine in gruppi rischiando cosi di scrivere su disco centinaia di pagine solo per scoprire che la maggioranza di esse è stata ancora modificata prima di essere sostituita).
Un approccio migliore utilizza buffer per pagine: cleaning solo per pagine che sono sostituibili dissociando le operazioni di cleaning e sostituzione. Con l'uso di buffer per pagine le pagine sostituite possono essere inserite in due liste: quella delle pagine modificate e non modificate (le pagine della prima possono essere periodicamente scritte in batch e scritte nella seconda). Una pagina non modificata può essere richiamata se il processo fa riferimento ad essa o persa quando il suo frame è assegnato ad un'altra pagina.

Load Control (controllo del carico): determina il numero dei processi che sarà residente nella memoria principale a livello di multiprogrammazione. La strategia del controllo del carico è critica per una gestione efficiente della memoria poiché se in memoria sono residenti pochissimi processi in un certo momento, ci saranno molte occasioni in cui tutti i processi sono bloccati e si passa molto tempo a trasferirli sul disco. Se ci sono troppi processi residenti, la dimensione del resident set di ogni processo sarà inadeguata e avverranno frequenti fault (si verificherà il trashing).

L'uso del processore cresce perchè ci sono meno probabilità che tutti i processi siano bloccati. Quando si raggiunge un punto in cui il resident set media risulta inadeguato facendo si che la frequenza dei fault cresca drasticamente e l'uso del processore collassi. Per risolvere il problema si può far uso di un algoritmo di working set con il quale si incorpora implicitamente il controllo del carico eseguendo solo quei processi i cui resident set sono sufficientemente grandi: fornendo un resident set

della dimensione richiesta per ogni processo attivo la strategia determina automaticamente e dinamicamente il numero dei processi attivi.

Sospensione del processo: se il livello di multiprogrammazione deve essere ridotto, uno o più processi residenti deve essere sospeso (quello con priorità più bassa, quello che provoca fault, l'ultimo processo attivato, quello con il più piccolo resident set, quello più grande o quello con la più grande finestra di esecuzione rimanente).
La memoria virtual e il disk caching hanno come obiettivo comune quello di prendere in memoria principale solo i dati o i programmi che sono realmente utili, cioè con accesso frequente mentre differiscono per il loro dominio di azione (la memoria virtuale su processi, pagine e segmenti mentre l'altro sui files). Il disk cache ha bisogno di prendere lo stesso tipo di decisione della memoria virtuale; molti file files sono usati da più processi i quali condividono anche le pagine.
La primitiva è una soluzione comune di memory-mapped files: un processo chiede di vedere una parte della memoria e viene fatta una system call che crea una regione per leggerlo. Il page fault prende in memoria quello di cui ha bisogno e il cleaning scrive sul disco quello che è cambiato (lettura e scrittura non sono eseguite come la system call, ma come un accesso alla memoria del processo che risulta più veloce).

MEMORIA VIRTUALE (supporto hardware)

L'hardware deve supportare la paginazione o la segmentazione; il sistema operativo deve gestire il movimento delle pagine e i segmenti tra la memoria secondaria e la memoria principale e decide qual è la migliore pagine da evitare.
Ogni processo ha la sua page table e quando tutte le sue pagine sono caricate nella memoria principale, la page table per un processo è creata e caricata nella memoria principale; questo strumento è necessario quando si consideriamo uno schema di memoria virtuale basato sulla paginazione associando un'unica page table ad ogni processo. Solo alcune pagine di un processo possono stare nella memoria principale e quindi è necessario un bit per ogni entry della page table per indicare se la pagina corrispondente è presente (P) o meno nella memoria principale: se il bit indica che la pagina è in memoria, allora l'entry contiene anche il numero di frame di quella pagina.
Si usa un altro bit di controllo poiché il bit di modifica (M) indica se la pagina corrispondente è stata alterata da quando è stata caricata nella memoria principale (Non cambiamento = il frame non è stato scritto sul disco quando la pagina è evitata).

Struttura della page table

Per leggere una parola dalla memoria si deve fare la traduzione di un indirizzo virtuale (o logico se si compone di un numero di pagine e offset) usando una tabella delle pagine; poiché quest'ultima è di lunghezza variabile (secondo la dimensione del processo) non si può tenere nei registri, ma deve essere in memoria principale per accedervi.
Quando un processo è in esecuzione, un registro ne conserva l'indirizzo virtuale usato come indice in quella tabella per cercare il corrispondente numero di frame, combinarlo con l'offset dell'indirizzo virtuale e produrre l'indirizzo real desiderato.
Per ogni processo è presente una tabella, ognuno può occupare enormi quantità di memoria virtuale. Se la quantità di memoria dedicata alle tabelle è troppo

grande,motli schemi di memoria virtuale memorizzano le page table nella memoria virtuale invece che in memoria principale, facendo si che le tabelle siano soggette alla paginazione e quando un processo è in esecuzione, almeno una parte della sua page table è in memoria principale.

Si può far uso di uno schema a due livelli per organizzare grandi tabelle delle pagine: si usa una directory di pagine, in cui ognuna entry punta ad una page table: un processo può essere composta dalla moltiplicazione tra la lunghezza della directory delle pagine e la massima lunghezza di una page table.

L'*inverted page table* è un approccio alternativo all'uso di tabelle delle pagine ad uno o due livelli: la parte contenente il numero della pagina di un indirizzo virtuale è mappato in una tavola hash usando una semplice funzione di hashing. La tavola hash contiene un puntatore alla tabella delle pagine invertita che contiene le entry della page table. C'è quindi solo un'entry nella tavola hash e una nella inverted page table per ogni memoria reale invece che una per pagina virtuale: una porzione fissa della memoria reale è occupata dalla tavola qualunque sia il numero dei processi o delle loro pagine virtuali del sistema (se più indirizzi virtuali hanno la stessa tavola hash, si applica una lista).

La tabella ha 4 campi:

1. numero di pagina
2. id del processo
3. i bit di controllo
4. campo di concatenazione.

Il numero del frame è il valore hash f calcolato. Tale informazione non è quindi tra i campi della tabella.

L'accesso alla inverted page table viene fatto per verificare che f sia un valore corretto.

Per ogni riga f della IPT (e quindi per ogni frame f) i campi sono process id del processo

p a cui il frame f è assegnato, numero della pagina di p contenuta in frame f, chain (in caso di collisioni si scorre una lista di frames, control bits).
Difatti, la tabella funziona come una tabella hash concatenata. Nel caso in cui una riga è
già utilizzata, l'ultimo campo rimanda ad un altra riga e cosi via, finché non si raggiunge
quella cercata (che pertanto avrà il campo di concatenazione vuoto o nullo).
Pro: mantiene in memoria un numero di entry pari al numero di frame della memoria fisica (le entry contengono l'id della pagina). Si applica una funzione hash al numero della pagina per calcolare il numero del frame. L'accesso alla tabella può confermare che
tale frame contiene la pagina cercata o meno. Le collisioni sono gestite mediante la tecnica del chaining. Spesso è sufficiente un solo accesso a memoria.
Contro: in caso di collisioni si deve percorrere la catena. Il numero di accessi a memoria non è costante.

Aggiornare la IPT: Lo fa il s.o.:
- calcolo frame h1 tramite hash leggendo riga ipt per h1;
- se h1 è libero aggiorno riga h1 nell'ipt: #page, #frame, pid, e setto chain=0, altrimenti scelgo un altra riga h2 (riapplicando hash) e vedo la stessa cosa: se h2 libero aggiorno riga h2 in ipt com sopra, ma poi setto chain =1 nella riga h1 di ipt, altrimenti continuo iterativamente (posso avere liste di collegamenti lunghe).

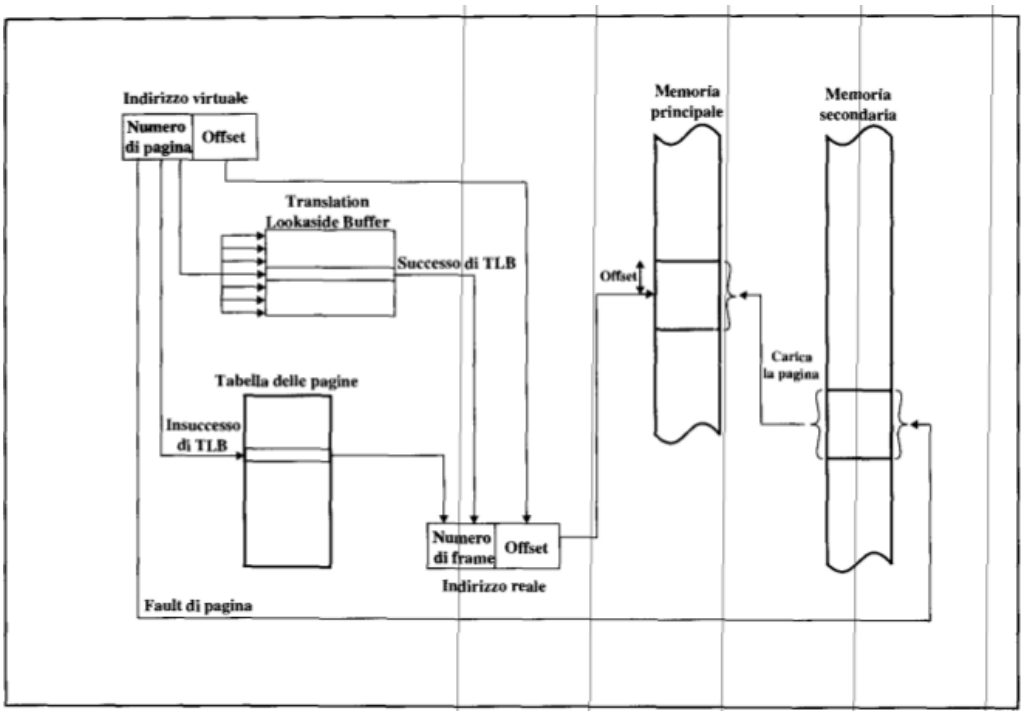
Leggere la IPT: Lo fa il s.o.:
- calcolo hash per h1;
- vedo riga h1 nell'ipt: se #page e pid sono corretti allora leggo il #frame e accedo a ram, altrimenti seguo il campo chain fino a trovare pid e #page corretti; se

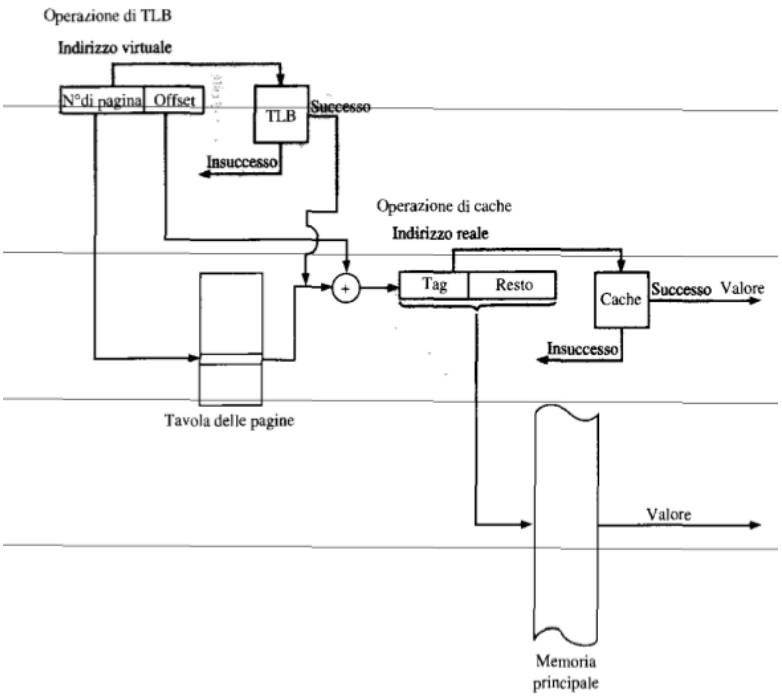
non li trovo vuol dire che la page non è in ram e va caricata -->faccio PAGE FAULT al so per comunicarglielo.

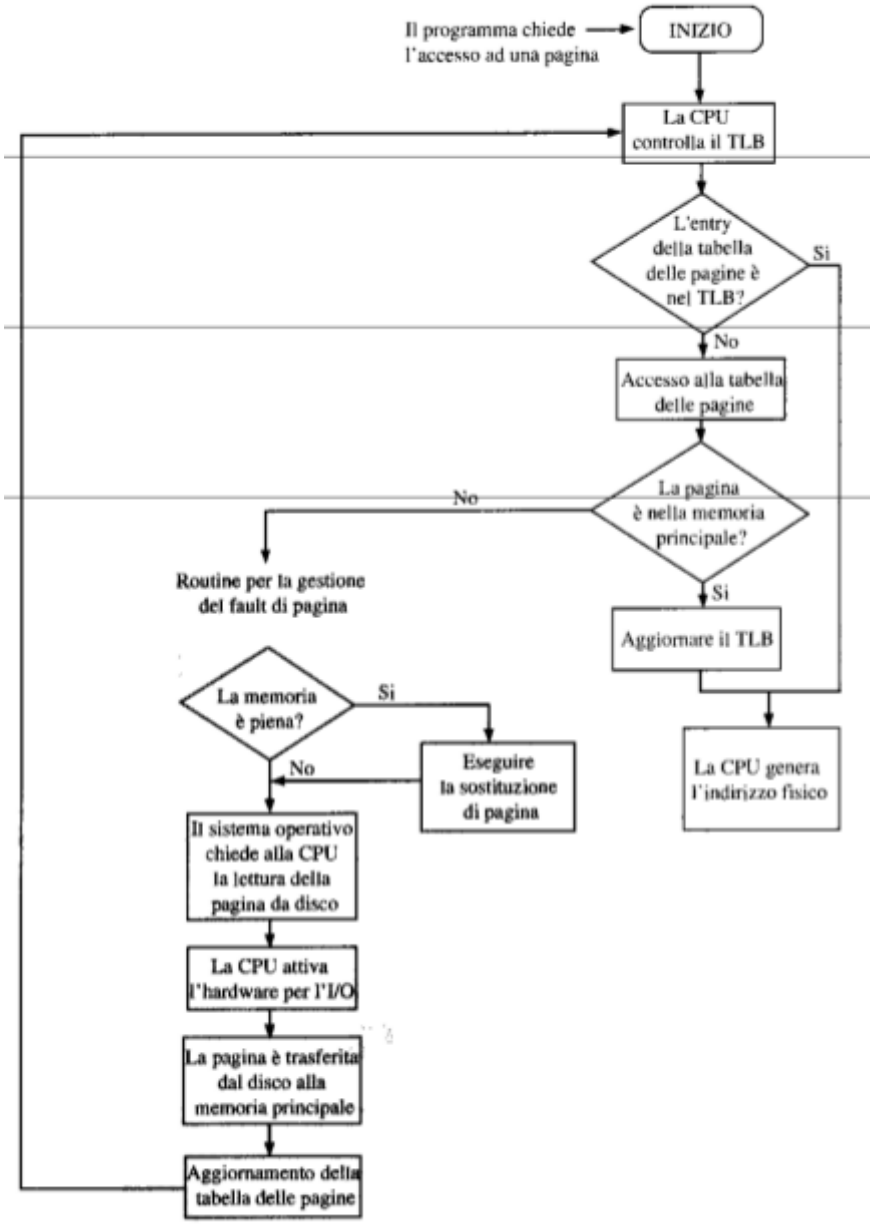
MAS e SAS: Multiple addressing space e single addressing space. Nel modello MAS ciascun processo ha un proprio spazio di indirizzamento mentre nel modello SAS lo spazio di indirizzamento è uno solo. In altre parole, nel modello MAS lo stesso indirizzo virtuale ha
un significato che dipende dal processo (spazi di indirizzamento separato) mentre in SAS
lo stesso indirizzo ha sempre lo stesso significato, è quindi possibile accedere alla memoria degli altri processi con un semplice accesso semplificando la condivisione dei dati.
Le inverted page tables sono usate sopratutto in sistemi SAS.

Translation Lookaside Buffer: ogni riferimento a memoria virtuale può causare due accessi alla memoria fisica: uno per andare a prendere l'entry della tabella delle pagine e uno per andare a prendere il dato desiderato, raddoppiando così l'accesso alla memoria: la translation lookaside buffer (TLB) è una speciale cache per le entry della tabella delle pagine che sono state utilizzate più frequentemente. Dato un indirizzo virtuale, il processo per prima cosa esaminerà il TLB: se l'entry della tabella desiderata è presente ("present bit" = 1), allora si recupera il numero di frame e si forma l'indirizzo reale, mentre se l'entry alla tabella delle pagine non è presente ("present bit" = 0), cioè ha fallito il TLB, allora il processore usa il numero di pagina come indice nella tabella delle pagine, per esaminare la corrispondente entry della tabella delle pagine, poiché la pagina desiderata non è nella memoria principale, e avviene un fault di pagina (fallimento dell'accesso alla memoria).
Il TLB contiene solo alcune entry nell'intera tabella delle pagine, non possiamo semplicemente usare un indice nel TLB dato dal numero di pagina: occorre quindi che

ogni entry nel TLB contenga sia il numero della pagina sia l'entry completa della tabella delle pagine. Quando viene generato un indirizzo fisico in forma di tag e resto si consulta la cache per vedere se il blocco contenente la parola è presente: se è così la parola è restituita alla CPU, altrimenti si recupera la parola dalla memoria principale.






Il programma chiede l'accesso ad una pagina
INIZIO
La CPU controlla il TLB
L'entry della tabella delle pagine è nel TLB?
Si
No
Accesso alla tabella delle pagine
La pagina è nella memoria principale?
No
Si
Routine per la gestione del fault di pagina
Aggiornare il TLB
La memoria è piena?
Si
No
Eseguire la sostituzione di pagina
La CPU genera l'indirizzo fisico
Il sistema operativo chiede alla CPU la lettura della pagina da disco
La CPU attiva l'hardware per l'I/O
La pagina è trasferita dal disco alla memoria principale
Aggiornamento della tabella delle pagine

Dimensione della pagina: è importante la scelta della dimensione della pagina: più è piccola la dimensione della pagina, minore è la frammentazione interna (che deve essere ridotta per ottimizzare l'uso della memoria principale) e maggiore è il numero delle pagine richieste per processo (con tabelle delle pagine più grandi). Quest'ultima conseguenza risulta negativa, poiché parte della tabella delle pagine dei processi attivi devono essere nella memoria virtuale e non in quella principale, provocando un doppio fault di pagina per un singolo riferimento alla memoria (per caricare la porzione desiderata della tabella delle pagine e per caricare la pagina del processo).

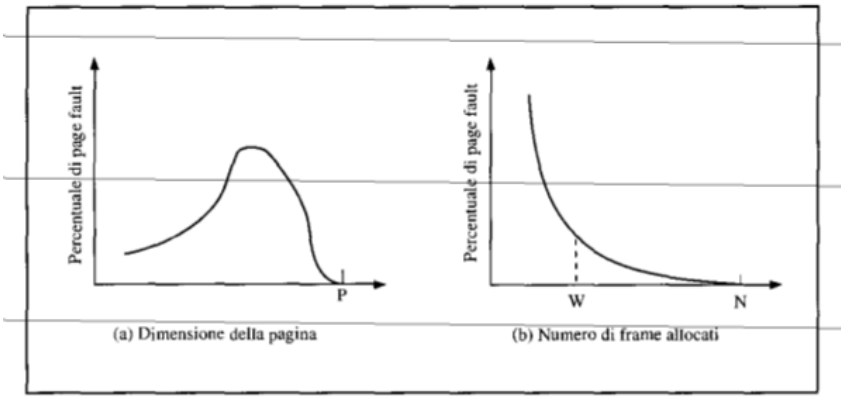


Figura 8.9 *Comportamento tipico di un programma durante la paginazione*

Le caratteristiche fisiche della maggior parte dei dispositivi per la memoria secondaria favoriscono una dimensione di pagine più grande per un più efficiente trasferimento dei blocchi di dati.

Quando cresce la memoria principale, aumenta anche lo spazio degli indirizzi usato dalle applicazioni. Le tecniche orientate agli oggetti incoraggiano l'uso di molti piccoli programmi e moduli di dati con riferimenti sparsi su un numero relativamente alto di oggetti, in un periodo relativamente breve; le applicazioni multithread provocano repentini cambiamenti nel flusso di istruzioni e riferimenti di memoria sparsi.

Più la dimensione di memoria dei processi cresce, più la località decresce, più diminuiscono le possibilità di accesso al TLB (per un dato TLB), diventando così un collo di bottiglia per le prestazioni. Per migliorare le prestazioni del TLB, si deve usare una versione di LB più grande con più entry: la dimensione del TLB interagisce con gli altri aspetti della progettazione di hardware come la cache della memoria principale e il numero di accessi a memoria per ciclo di istruzioni, anche se la dimensione non cresce rapidamente quanto quella della memoria principale;

l'alternativa di usare pagine di dimensioni più grandi non conviene perchè porta al peggioramento delle risorse. L'uso di pagine con dimensioni diverse favorisce la flessibilità necessaria per usare il TLB.

Segmentazione: permette al programmatore di vedere la memoria come un insieme di segmenti o spazi di indirizzamento multipli: i segmenti possono essere di dimensione dinamica e i riferimenti a memoria sono indirizzi della forma (numero di segmento, offset).

I vantaggi di questa organizzazione sono vari:

- semplifica la gestione di strutture dati che crescono, poiché si può assegnare un segmento alla struttura dati e il sistema operativo espanderà o restringerà il segmento secondo i casi, spostandolo se c'è necessità di più memoria;
- permette la modifica e la ricompilazione indipendente dei programmi, senza richiedere che si rifaccia il link con tutto l'insieme dei programmi per ricaricarli;
- si presta alla protezione poiché in ogni segmento il programmatore o amministratore vi può fornire privilegi.

Ogni processo ha la sua tabella di segmenti che viene creata e caricata nella memoria principale quando tutti i segmenti sono caricati; ogni entry della tabella contiene l'indirizzo di partenza del segmento corrispondente nella memoria principale e la lunghezza del segmento. Solo alcuni segmenti di un processo possono essere presenti nella memoria principale: è necessario quindi un bit per ogni entry della tabella dei segmenti per indicare se il segmento corrispondente è presente o meno nella memoria principale (se il segmento è in memoria, l'entry contiene anche l'indirizzo di partenza e la lunghezza di quel segmento). Il modify bit indica se i contenuti del segmento corrispondente sono stati alterati dopo l'ultimo caricamento in memoria: se non ci sono stati cambiamenti, non serve scrivere il segmento sul disco; altri bit di controllo possono riguardare la protezione e la condivisione. Il meccanismo di base per leggere una parola dalla memoria richiede la traduzione di un indirizzo virtuale o logico

(formato dal numero del segmento e dall'offset) in un indirizzo fisico usando una tabella dei segmenti.

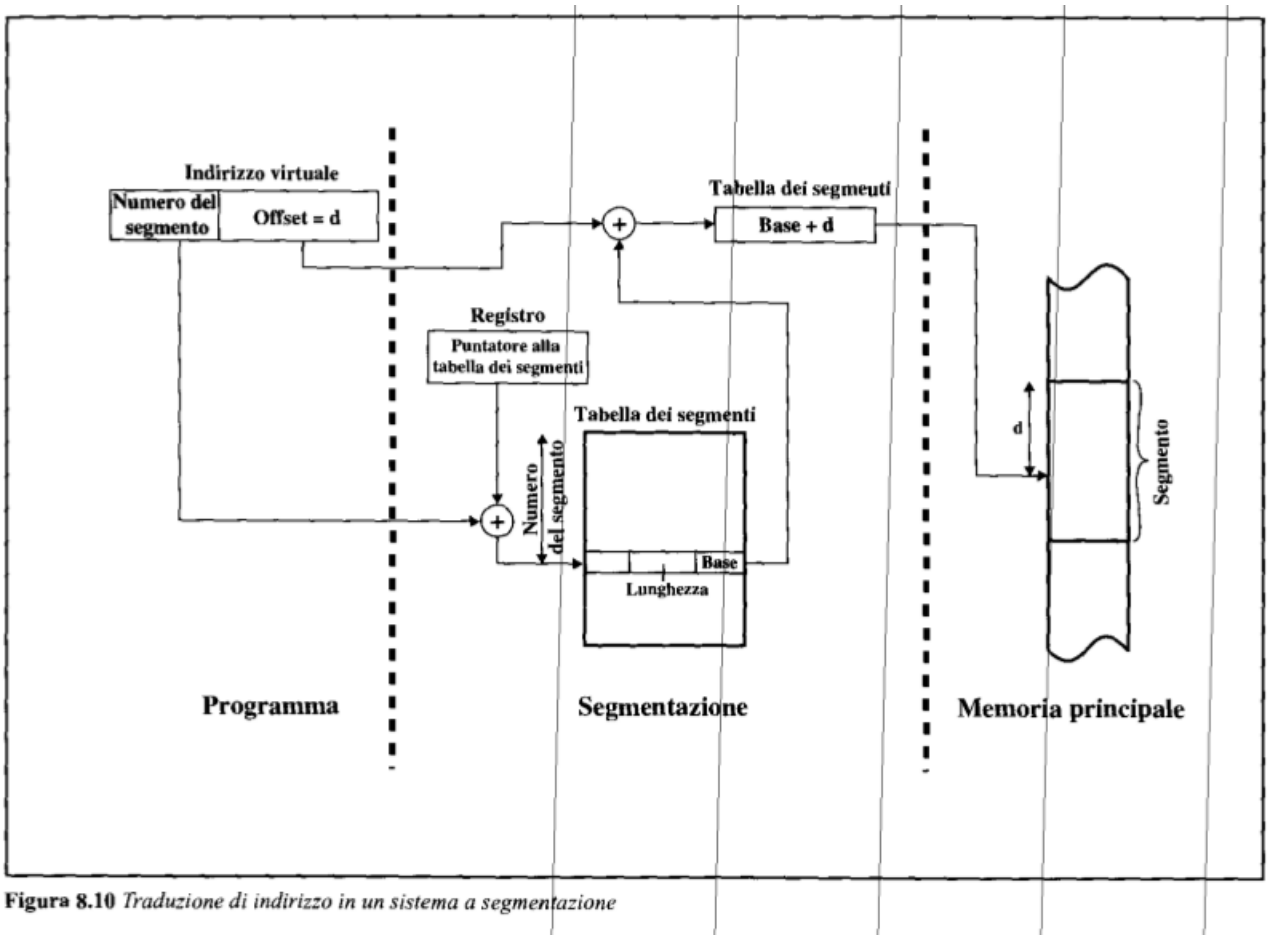


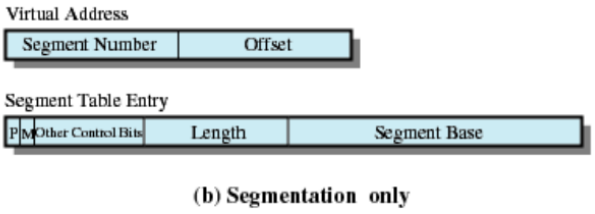
Figura 8.10 *Traduzione di indirizzo in un sistema a segmentazione*

Sia la paginazione che la segmentazione hanno dei punti a favore:

– la paginazione è trasparente al programmatore, elimina il problema della frammentazione interna e consente quindi l'uso più efficiente della memoria principale;

– la segmentazione è visibile al programmatore e ha dei vantaggi come la capacità di gestire strutture dati dinamiche, la modularità e il supporto per la condivisione e la protezione.

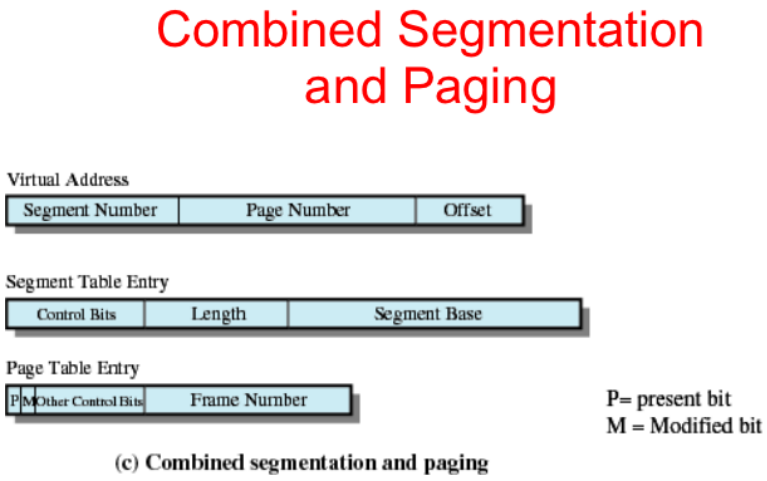
Segment Table Entries



(b) Segmentation only

In un sistema che combina paginazione e segmentazione, uno spazio di indirizzo

utente è diviso in un certo numero di segmenti: ogni segmento è a sua volta suddiviso in pagine di dimensione fissa, lunghe quanto i frame della memoria principale (se un segmento è più corto di una pagina, occupa solo una pagina). Ogni processo ha una tabella dei segmenti e per ogni segmento una tabella delle pagine; durante un esecuzione di un processo, un registro contiene l'indirizzo di partenza della tabella dei segmenti di quel processo (in presenza di un indirizzo virtuale, il processore usa il numero di segmento come indice nella tabella dei segmenti, per trovare la tabella delle pagine del segmento in questione e per cercare il numero del frame ad essa corrispondente che combinato con l'offset dell'indirizzo virtuale, produce l'indirizzo reale desiderato). Il presence bit e il modify bit non sono necessari, perché gestiti a livello di pagina.



Per la condivisione in un sistema a segmentazione, poiché la entry di ogni tabella dei segmenti contiene una lunghezza e un indirizzo di base, un programma non può inavvertitamente accedere ad una locazione di memoria principale il cui indirizzo superi l'indirizzo limite del segmento: per questo è possibile avere un riferimento allo stesso segmento nella tabella dei segmenti di più di un processo. In un sistema a paginazione invece, la struttura delle pagine dei programmi e dei dati non è visibile al programmatore, rendendo i requisiti di protezione e condivisione più difficili.

In un sistema ad anello, un programma può accedere solo a dati che risiedono sullo stesso anello o su un anello con meno privilegi e può chiamare servizi che risiedono sullo stesso anello o su un anello più privilegiato (anelli con numero più basso hanno più privilegi).

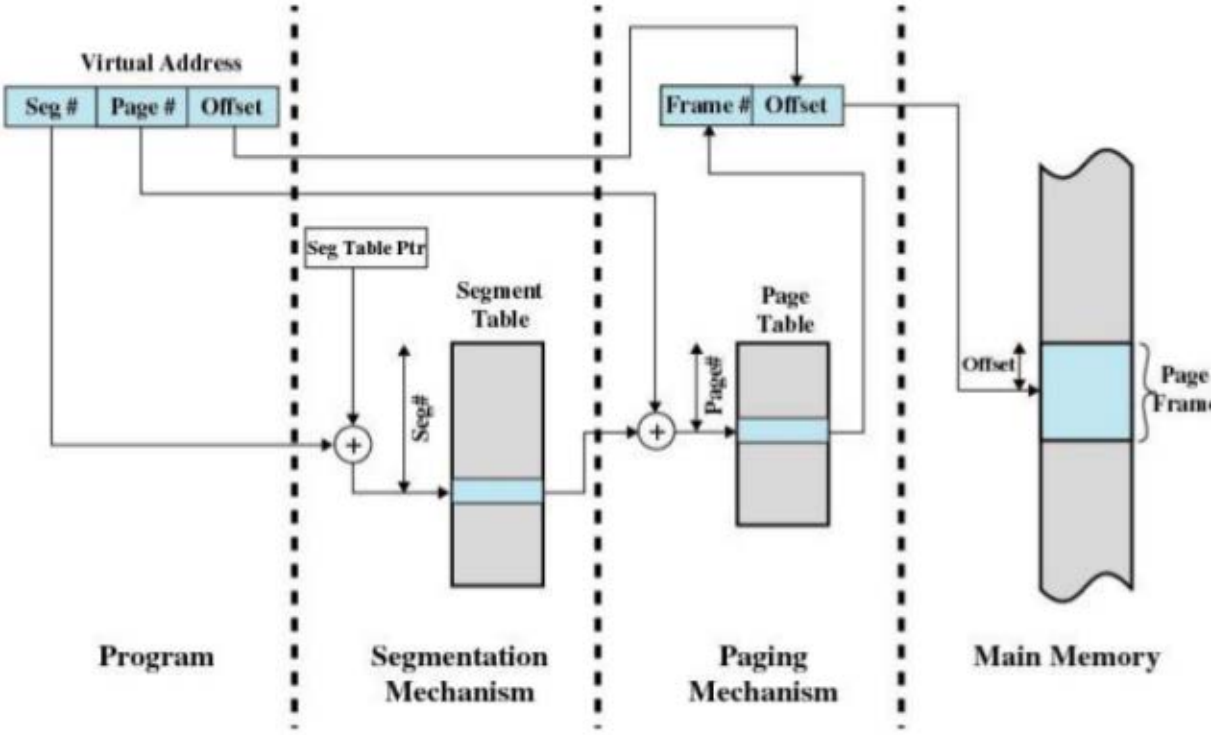


Figure 8.13 Address Translation in a Segmentation/Paging System

Il software del sistema operativo: è importante la gestione della memoria, che può avvenire con l'uso di memoria virtuale, di paginazione, segmentazione o di algoritmi specifici. L'obiettivo è minimizzare la frequenza di fault di pagina, che causano un considerevole sovraccarico software. Al minimo, il sistema operativo deve decidere quali pagine residenti deve sostituire, al costo dell'I/O per lo scambio delle pagine; quindi deve schedulare un altro processo da eseguire durante l'operazione di I/O sulla pagina, causando un cambio di processo e di conseguenza si deve fare in modo che durante l'esecuzione di un programma, la probabilità di riferirsi ad una parola mancante sia minima.

Strategia di fetch: determina quando una pagina deve essere caricata nella memoria principale. Le alternative sono la paginazione a richiesta (una pagina è caricata in memoria solo quando si fa un riferimento ad una locazione su quella pagina) e la prepaginazione (altre pagine, oltre a quella che ha generato il fault di pagina, sono caricate; se le pagine di un processo sono memorizzate in aree contigue della memoria secondaria, è più efficiente caricare un certo numero di pagine contigue tutte in una volta, piuttosto che caricarle una alla volta, anche se è inefficiente se si fa riferimento alla maggior parte delle pagine caricate in più). Quest'ultima tecnica può essere utilizzata quando il processo è avviato per la prima volta (il programmatore dovrebbe indicare le pagine desiderate, sia ogni volta che ci sta un fault di pagina).

Strategia di posizionamento: determina dove deve risiedere un pezzo di processo nella memoria reale. In un sistema a segmentazione, la strategia di posizionamento è un importante problema di progettazione, mentre per la paginazione il posizionamento è irrilevante perché l'hardware di traduzione degli indirizzi e l'hardware per l'accesso alla memoria principale possono eseguire le loro funzioni per ogni combinazione pagina-frame con pari efficienza.

Strategia di sostituzione: riguarda la selezione di una pagina in memoria da sostituire, quando una nuova pagina deve essere caricata; quando tutti i frame della

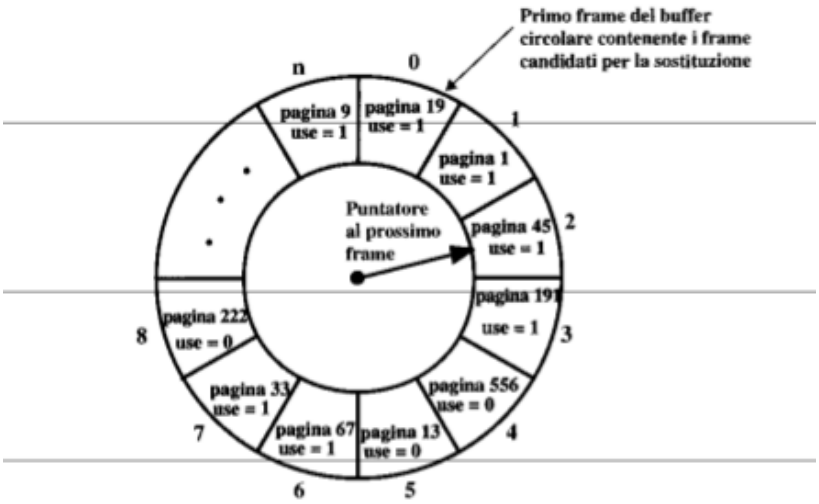
memoria principale sono occupati ed è necessario caricare una nuova pagina per soddisfare un fault di pagina, tale strategia determina quale tra le pagine presenti in memoria deve essere sostituita. Tutte le strategie hanno l'obiettivo di rimuovere la pagina a cui ci si riferirà di meno nel prossimo futuro. Una restrizione a tale strategia deriva dal fatto che alcuni frame della memoria principale possono essere bloccati e la pagina immagazzinata in quei frame non può essere sostituita (il kernel è conservato in queste pagine bloccate); si possono bloccare i frame associando un lock bit ad ogni frame. Alcuni algoritmi per sostituzione sono:

l'algoritmo ottimo: seleziona per la sostituzione la pagina alla quale ci si riferirà dopo il tempo più lungo; il numero di fault page è minimo, però è impossibile implementarlo poiché richiede che il sistema operativo abbia una perfetta conoscenza degli eventi futuri;

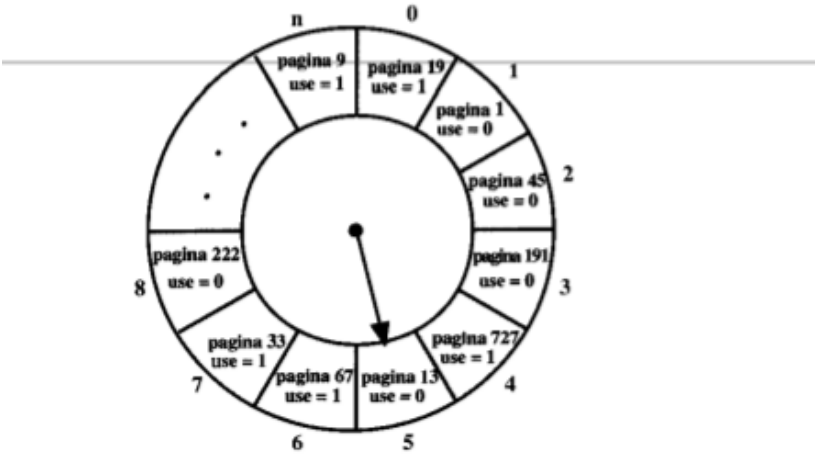
least-recently-used (LRU): sostituisce in memoria la pagina a cui il processo non s è riferito per più tempo; anche qui ci sono problemi per l'implementazione. Se un programma accese ad un intervallo di indirizzi superiori a quelli della memoria allocata per il programma, LRU scarica le pagine nell'ordine di riferimento e perciò incontra un'infinita sequenza di fault di pagina; aumentando la allocazione di memoria, non si riduce la percentuale di fault di pagina fino a che l'intero programma non ha trovato un posto in memoria;

first-in-first-out (FIFO): tratta i frame allocati per le pagine di un processo come un buffer circolare e le pagine sono rimosse utilizzando tutto ciò lo schema del round-robin. Tutto ciò che è richiesto è un puntatore che va in cerchio lungo i frame contenenti le pagine del processo; è quindi una delle strategie più semplici da implementare, poiché si va a sostituire la pagina che è stata più a lungo in memoria (può essere fuori uso). Quest'ultimo ragionamento sarà spesso sbagliato perchè ci saranno spesso porzioni di programmi o dati molto usati durante tutta l'esecuzione di un programma; tali pagine saranno caricate e scaricate dall'algoritmo di FIFO;

strategia dell'orologio: richiede l'associazione di un bit addizionale ad ogni frame, chiamato use bit: quando una pagina è caricata per la prima volta in un frame di memoria, lo use bit di quel frame è fissato a 1. Quando ci si riferirà successivamente alla pagina (dopo il riferimento che ha generato un fault di pagina), il suo user bit sarà fissato a 1; quando una pagina è sostituita, il puntatore punta al frame successivo del buffer quando viene il momento di sostituire una pagina, il sistema operativo scorre il buffer per trovare un frame il cui use bit è fissato a zero (se ne incontra uno con user bit uguale a 1, gli riassegna il valore zero). Se un use bit di qualche frame del buffer ha valore zero all'inizio del ciclo, il primo di tali frame che si incontra è scelto per la sostituzione, il primo di tali frame verrà scelto per la sostituzione; se tutti gli use bit dei frame hanno valore 1, il puntatore assegnerà a tutti il valore 0 facendo un giro completo del buffer fermandosi nella sua posizione iniziale, sostituendo la pagina in quel frame. Qualunque frame con use bit uguale a 1 è evitato dall'algoritmo.



(a) Stato del buffer appena prima della sostituzione di pagina



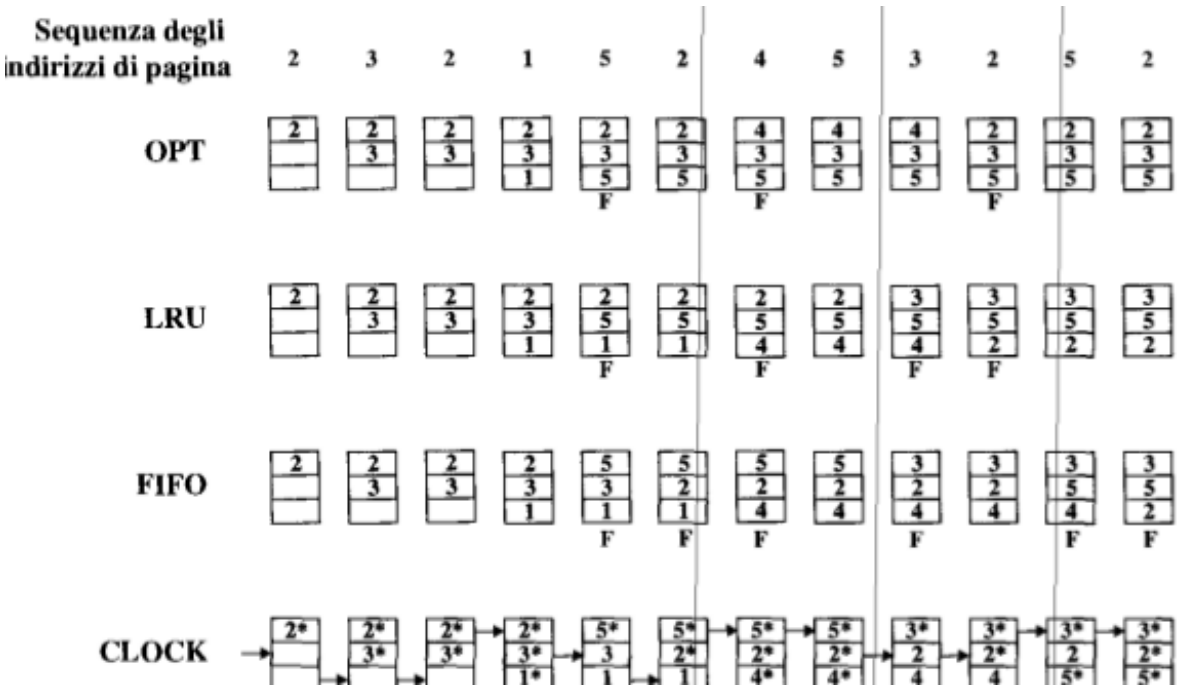
(b) Stato del buffer subito dopo la sostituzione di pagina

L'algoritmo dell'orologio può essere reso più potente incrementando il numero di bit utilizzati: si aggiunge infatti un modify bit ad ogni pagina della memoria principale. Quando si modifica una pagina, essa può essere sostituita solo dopo che è stata

riscritta nella memoria secondaria. I bit quindi possono questi valori

- use e modify = 0 → non utilizzato recentemente, non modificato;
- use = 1, modify = 0 → utilizzato recentemente, non modificato;
- use = 0, modify = 1 → non utilizzato recentemente, modificato;
- use = 1, modify = 1 → utilizzato recentemente, modificato.

Il puntatore scorre durante il buffer senza modificare il valore dello use bit: non appena incontra un frame con use e modify bit uguale a zero (se non è stata usata recentemente, ha il vantaggio che non ha bisogno di essere riscritta nella memoria secondaria), lo seleziona per la sostituzione. Se questo procedimento fallisce, si scorre il buffer cercando un frame che ha sempre use bit uguale a 0, ma con modify bit che sta a 1 e appena lo incontra, verrà selezionato per la sostituzione (durante lo scorrimento assegna allo use bit di ogni frame il valore zero). Se anche questo passo fallisce, il puntatore dovrebbe puntare nuovamente alla posizione di partenza e tutti gli use bit dei frame avranno valore zero, ripetendo successivamente i due passi scritti in precedenza



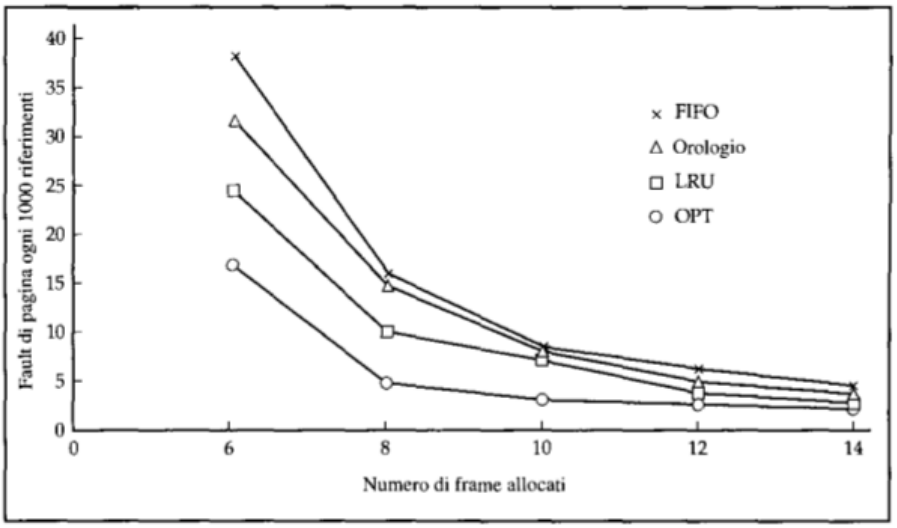


Figura 8.15 *Confronto degli algoritmi di sostituzione locale ad allocazione fissa*

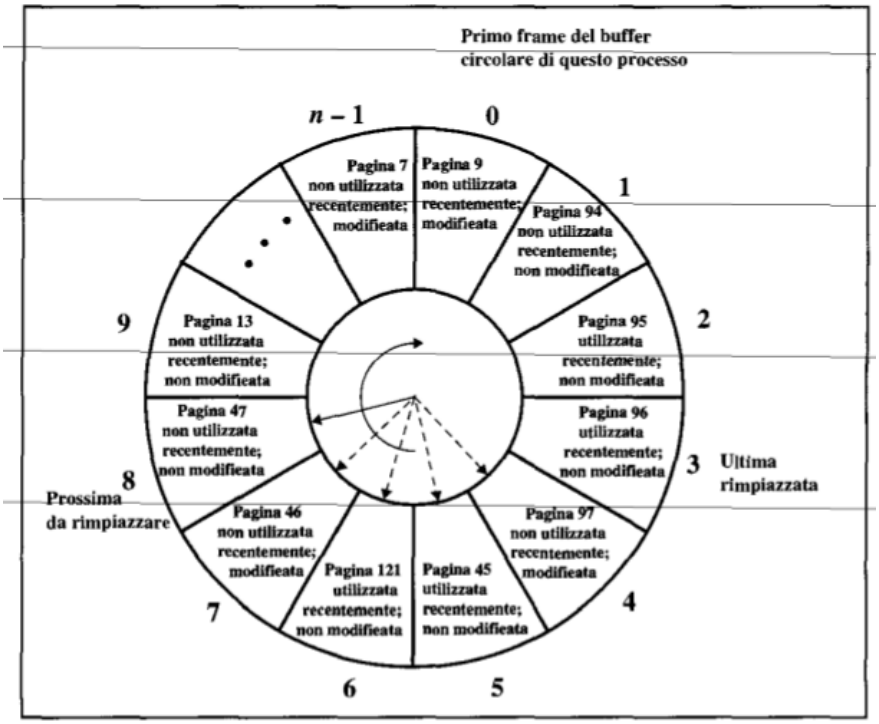
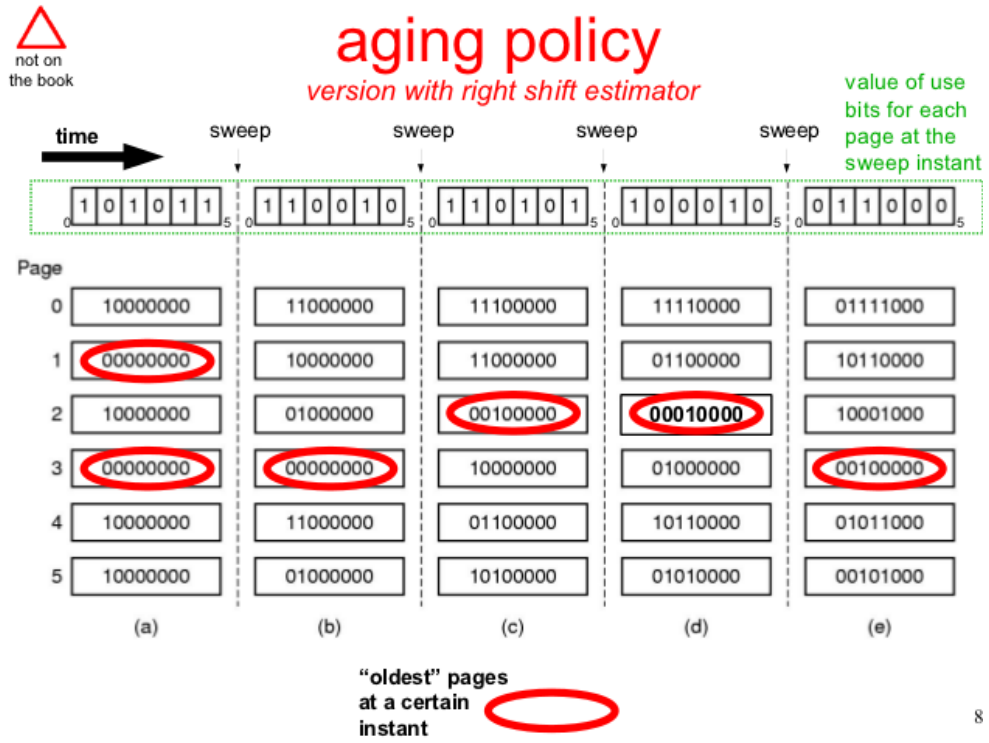


Figura 8.16 *Algoritmo dell'orologio per la sostituzione di pagine [GOLD89]*

aging policy: per ogni periodo si prende un "estimatore d'età": minore è il valore, più vecchia è la pagina. Periodicamente vengono scansionate tutte le pagine: vengono controllati use e modifies bit per ogni pagina e la pulizia di tutti gli use bits per registrare la pagina da usare per la prossima scansione.



Gestione del resident set: con la memoria virtuale gestita a pagine, non è necessario caricare tutte le pagine di un processo nella memoria principale per prepararlo all'esecuzione: il sistema operativo deve decidere quante pagine caricare (quanta memoria principale allocare per un particolare processo. Minore è la quantità di memoria allocata per un processo, maggiore è il numero di processi che può risiedere nella memoria principale in ogni momento (aumenta la possibilità che il sistema operativo trovi almeno un processo Ready in ogni momento, riducendo il tempo perso per il trasferimento). Se un numero relativamente piccolo di pagine di un processo è nella memoria principale, nonostante il principio di località, la frequenza di fault di pagina sarà alta. Oltre una certa dimensione, l'ulteriore allocazione di memoria principale per un processo non cambierà il numero di fault di pagina per quel processo. I sistemi operativi utilizzano due tipi di strategie:

- *allocazione fissa*: assegna un numero fissato di pagine per l'esecuzione deciso al momento del primo caricamento e può essere determinato in base al tipo di processo e alle indicazioni del programmatore; ogni volta che avviene un fault di pagina nell'esecuzione del processo, una delle pagine di quel processo deve essere sostituita dalla pagina necessaria;
- *allocazione variabile*: consente di variare il numero di frame allocati per un processo durante la sua esecuzione; un processo con maggiore numero di fault di pagina, dispone di frame addizionali per ridurre il numero di fault di pagina, mentre per un processo con un numero basso di fault di pagina, gli sarà assegnata una minore locazione di memoria. Anche se è più potente, la difficoltà sta nel richiedere al sistema operativo di giudicare il comportamento dei processi attivi, causando overhead nel software ed è dipendente dai meccanismi hardaware.

Ambiti di sostituzione: sono attivati da fault di pagina dovuti a mancanza di frame liberi. Può essere *locale* (sceglie la pagina da sostituire tra le pagine residenti in memoria del processo che ha generato fault di pagina) e *globale* (considera tutte le pagine non bloccate in memoria principale come candidate alla sostituzione, senza

badare a quale processo appartiene). Un resident set fissato implica una strategia di sostituzione locale: per mantenere fissa la dimensione del resident set, una pagina rimossa dalla memoria principale deve essere sostituita da un'altra pagina dello stesso processo. Una politica di allocazione variabile può utilizzare una strategia di sostituzione globale: la sostituzione di una pagina di un processo nella memoria principale con una pagina di un altro processo fa si che la memoria allocata per un processo aumenti di una pagina e che la memoria allocata per l'altro processo si restringa di una pagina.

Ci possono essere varie combinazioni:

- *Allocazione fissa, ambito globale*: si ha un processo in esecuzione nella memoria principale con un numero fisso di pagine a sua disposizione; quando avviene un fault di pagina, il sistema operativo deve scegliere, tra le pagine del processo residenti in memoria al momento, la pagina da sostituire. Con una politica di allocazione fissa, è necessario decidere in anticipo la quantità di allocazione da assegnare ad un process, in base al tipo di applicazione e alla quantità di memoria richiesta dal programma. Allocazione piccola → maggiori fault di pagina, rallentamento del sistema | Allocazione grande → pochi processi, processore lento.
- *Allocazione variabile, ambito globale*: ci sono processi nella memoria principale, ciascuno con a disposizione un certo numero d frame allocati; il sistema operativo si lascia una lista dei frame liberi. Quando avviene un fault di pagina, un frame libero è aggiunto al resident set di un processo e si carica la pagina: un processo di conseguenza crescerà gradualmente in dimensione il che dovrebbe aiutare a ridurre il numero complessivo di fault di pagina del sistema. Quando però non ci sono frame liberi per la sostituzione, il sistema operativo deve scegliere una pagina in memoria da sostituire fra tutti i frame in memoria, eccetto quelli bloccati, come quelli del kernel, rischiando di levare la pagina al processo meno adatto: per risolvere questo si possono usare un buffer per pagine.

– *Allocazione variabile, ambito locale*: quando un processo è caricato nella memoria principale, si alloca per lui un certo numero di frame (si usano prepaginazione o la paginazione per riempire i frame allocati). Quando avviene un fault di pagina, si seleziona la pagina da sostituire tra quelle del resident set del processo che provoca il fault di pagina; si rivaluta periodicamente l'allocazione fornita al processo e la si aumenta o decrementa per migliorare le prestazioni globali in base alle probabili richieste future dei processi.

Tabella 8.4 *Gestione del resident set*

	Sostituzione locale	**Sostituzione globale**
Allocazione fissa	Il numero di frame allocati per un processo è fisso, la pagina da rimpiazzare è scelta tra i frame allocati per il processo	Impossibile
Allocazione variabile	Il numero di frame allocati per un processo può essere cambiato nel tempo, per mantenere il working set del processo La pagina da sostituire è scelta tra i frame allocati per il processo	La pagina da sostituire è scelta tra tutti i frame disponibili in memoria principale; questo fa sì che la dimensione del resident set del processo possa variare

Il working set definisce un processo in un certo instante.

Delta = parametro per un processo;

t = tempo virtuale (tempo che passa mentre il processo è in esecuzione).

W(t, delta) = insieme di pagine di quel processo a cui ci si è riferiti nelle ultime delta unità di tempo virtuale. Più grande è la dimensione della finestra, più grande è il working set. Se un processo è eseguito oltre le delta unitò di tempo e usa solo una pagina, allora W(t,delta) è uguale a 1; un working set può crescere fino ad essere grande quanto il numero delle N pagine del processo se si accede a molte pagine diverse rapidamente e se la dimensione della finestra lo permette.

Il valore di W quindi è compreso tra 1 e il minimo tra delta e N.

Quando un processo comincia la sua esecuzione, costruisce gradatamente il suo working set mentre fa riferimento a nuove pagine. Prima o poi il processo dovrebbe stabilizzarsi su un certo insieme di pagine e successivi momenti di transizione

riflettono uno spostamento del processo verso una nuova località: durante questa ultima fase, alcune pagine della vecchia località rimangono all'interno della finestra delta, causando una crescita della dimensione del working set e quando la finestra supera questi riferimenti a pagina, la dimensione del working set diminuisce fino a contenere solo le pagine della nuova località. Il working set può essere usato anche per guidare una strategia per la dimensione del resident set che consiste nel controllare il working set di ogni processo rimuovendo periodicamente dal resident set di un processo quelle pagine che non sono presenti; un processo può essere eseguito solo se il suo working set è nella memoria principale, cioè se il suo resident set contiene il suo working set. In questo modo si minimizzano i fault di pagina. I problemi con questa strategia sono che la dimensione e l'appartenenza al working set cambiano nel tempo e sarebbe difficile stimare ad ogni processo la sua dimensione, e il valore ottimale di delta è sconosciuto e variabile. Un'altra strategia è quella di controllare direttamente la dimensione del working set: se la frequenza di fault di pagina è al di sotto di una cerca soglia, l'intero sistema può migliorare assegnando al processo un resident set più piccolo senza danneggiare il processo, mentre se la frequenza di fault di pagina è più della soglia superiore, il processo può beneficiare di un aumento delle dimensioni de resident set provocando meno fault di pagina.
L'*algoritmo di page-fault frequency (PFF)* richiede un use bit da associare ad ogni pagina presente in memoria: quando si accede alla pagina, si assegna il valore 1 allo use bit. Quando avviene un fault di pagina, il sistema operativo prende nota del tempo virtuale trascorso dall'ultimo fault di pagina con un contatore e si definisce una soglia F: se il tempo trascorso è minore della soglia, si aggiunge una pagina al resident set del processo, altrimenti si scartano tutte le pagine con use bit uguale a 0 e di conseguenza si restringe il resident set (si riassegna il valore zero agli use bit delle pagine rimanenti). Il tempo fra i fault di pagina è il reciproco della frequenza di fault di pagina. Le prestazioni non sono buone quando c'è uno spostamento ad una nuova località.

SCHEDULING MONOPROCESSORE

In un sistema in multiprogrammazione i processi sono mantenuti nella memoria principale e ogni processo alternativamente usa il processore o aspetta l'esecuzione dell'I/O o il completamento di un altro evento. Lo scopo dello scheduling è assegnare i processi che devono essere eseguiti al processore o ai processori, nel tempo, in modo da realizzare gli obiettivi del sistema come il tempo di risposta, il throughput e l'efficienza del processore.

Tipi di scheduling:

Scheduling a lungo termine	Si decide di aggiungere un processo all'insieme dei processi che devono essere eseguiti
Scheduling a medio termine	Si decide di aggiungere un processo all'insieme dei processi che sono parzialmente o completamente in memoria
Scheduling a breve termine	Si decide quale processo disponibile sarà eseguito dal processore
I/O scheduling	Si decide quale processo in attesa di I/O sarà gestito da un dispositivo di I/O disponibile

lo scheduling può essere:

- *breve termine*: decide quale processo Ready eseguire e per quanto tempo deve essere eseguito; è chiamato ogni volta che si verifica un evento che può portare alla sospensione del processo corrente o che consente di prerilasciare il processo attualmente in esecuzione in favore di un altro (esempio sono interruzioni di clock e di I/O, chiamate a sistema operativo, segnali).
- *medio termine*: è una parte della funzione di trasferimento su disco dei processi; è basato sul bisogno di gestire la multiprogrammazione e sospende i processi;
- *lungo termine*: quando si crea un processo; decide quale programma aggiungere al sistema per eseguirli e controlla il grado di multiprogrammazione. Una volta inserito un job e o un programma utente, diventa un processo e va ad aggiungersi alla coda dello scheduler a breve termine; lo schedular preleva i processi dalla coda

quando può e deve decidere se il sistema operativo può prendere uno o più processi e quale job accettare e trasformarlo in processo.

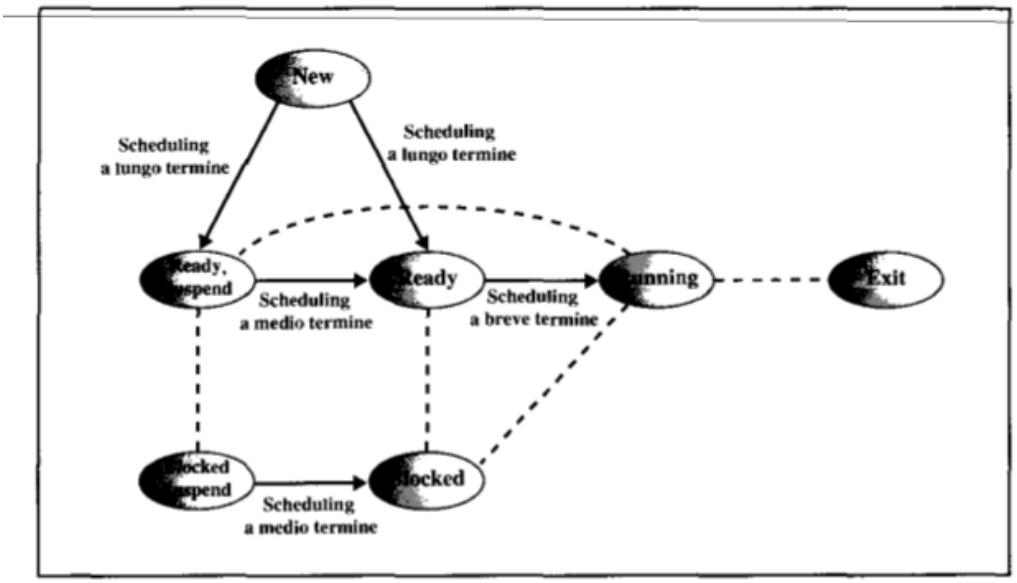
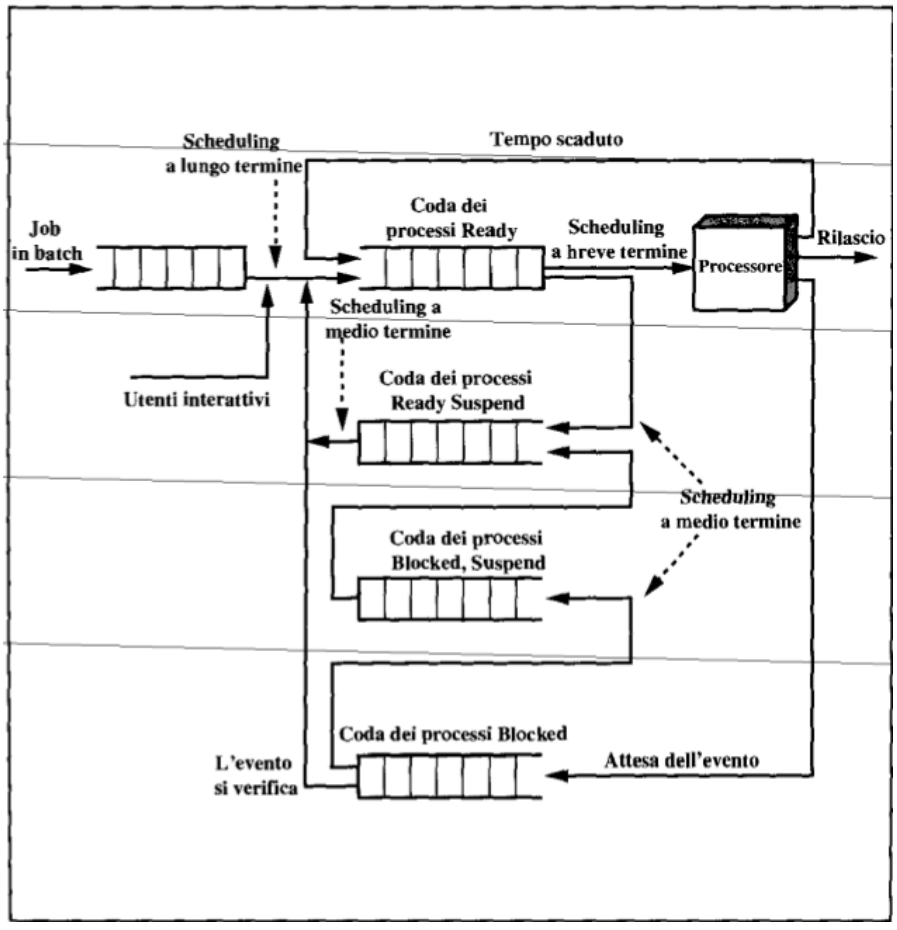


Figura 9.1 *Scheduling e transizioni di stato dei processi*

Lo scheduling influenza le prestazioni del sistema perchè determina i processi che aspetteranno e quelli che precederanno.



Algoritmi di scheduling: criteri dello scheduling a breve termine.

Orientati all'utente, Relativi alle prestazioni	
Tempo di risposta	Per un processo interattivo, questo è il tempo che trascorre dall'invio di una richiesta fino a quando la risposta non comincia ad essere ricevuta. Spesso un processo può cominciare a produrre qualche output per l'utente, mentre continua l'esecuzione. Perciò, dal punto di vista dell'utente, questa è una misura migliore del tempo di turnaround. La disciplina di scheduling dovrebbe tentare di raggiungere bassi tempi di risposta e di massimizzare il numero di utenti interattivi che hanno un tempo di risposta accettabile.
Tempo di turnaround	È l'intervallo di tempo tra l'invio di un processo e il suo completamento. Comprende il tempo di esecuzione e il tempo speso in attesa delle risorse, compreso il processore. Questa è una misura appropriata per i job batch.
Scadenze	Quando si può specificare il termine di completamento di un processo, la disciplina di scheduling può subordinare altri obiettivi oltre a quello di massimizzare la percentuale di scadenze raggiunta.
Orientati all'utente, Altri	
Prevedibilità	Un dato job può essere eseguito nello stesso tempo e allo stesso costo, indipendentemente dal carico del sistema. Un'ampia variazione nel tempo di risposta o nel tempo di turnaround è fastidiosa per l'utente: potrebbe segnalare un'ampia oscillazione nel carico di lavoro del sistema oppure la necessità di adattare il sistema per evitare l'instabilità
Orientati al sistema, Relativi alle prestazioni	
Throughput	La strategia di scheduling dovrebbe tentare di massimizzare il numero di processi completati per unità di tempo. Così si misura la quantità di lavoro eseguito, che dipende chiaramente dalla lunghezza media di un processo, ma che è anche influenzata dalla strategia di scheduling che può avere effetti sull'utilizzazione
Utilizzazione del processore	Questa è la percentuale del tempo in cui il processore è occupato. Per un sistema costoso e condiviso, questo è un criterio significativo, mentre in sistemi a singolo utente e in alcuni altri sistemi, come i sistemi a tempo reale, questo criterio è meno importante.
Orientati al sistema, Altri	
Fairness	In assenza di indicazioni da parte dell'utente, oppure del sistema, i processi devono essere trattati allo stesso modo, e nessun processo deve subire starvation.
Priorità	Quando si assegna una priorità ai processi, la strategia di scheduling dovrebbe favorire i processi a più alta priorità.
Bilanciamento delle risorse	La strategia di scheduling deve tenere impegnate le risorse del sistema, favorendo i processi che sottoutilizzano le risorse più impegnate. Questo criterio coinvolge anche lo scheduling a medio e lungo termine.

Uso delle priorità: ad ogni processo è assegnata una priorità e lo scheduler deve sempre scegliere un processo a più alta priorità a discapito di processi con più bassa priorità. Un problema con uno schema di scheduling basato solamente sulla priorità è che i processi con la priorità più bassa possono soffrire di starvation (se sono presenti stabilmente processi Ready a più alta priorità); per superare questo problema, la priorità di un processo può cambiare in dipendenza del tempo di presenza in una coda o del corso della sua esecuzione.

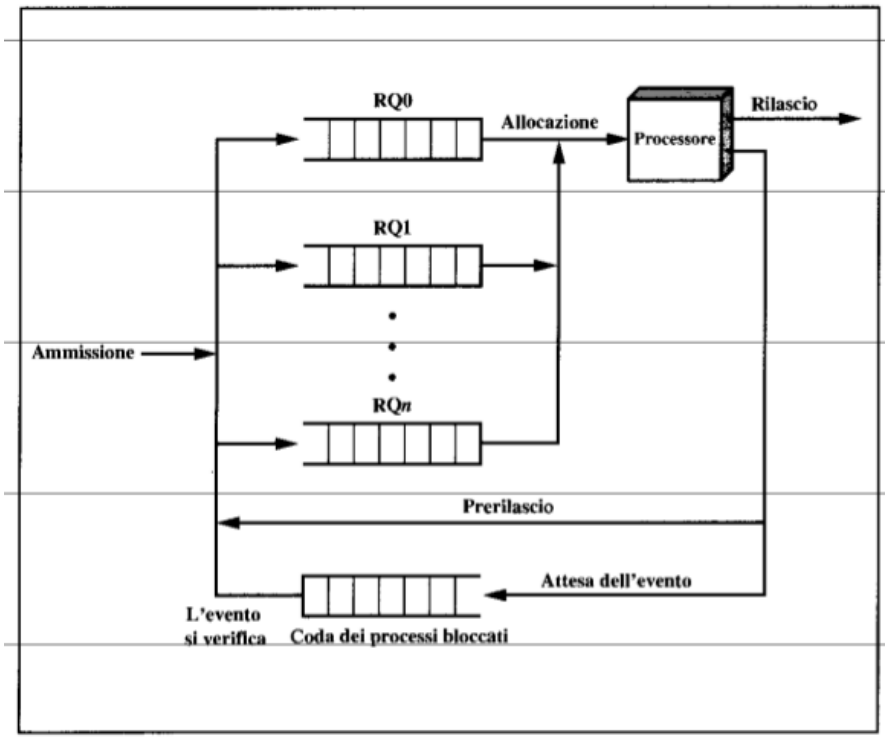


Figura 9.4 *Code di priorità*

Strategie di scheduling alternativo: la *funzione di selezione* determina quale processo tra i processi Ready selezionare per la prossima esecuzione; la funzione può basarsi sulla priorità, sulle richieste di risorsa o sulle caratteristiche di esecuzione del processo. Per l'esecuzione di un processo conta il tempo fin qui speso nel sistema (w), il tempo speso in esecuzione (e) e il tempo totale di servizio richiesto dal processo (s). Il *modo di decisione* specifica gli istanti in cui si esegue la funzione di selezione; può essere senza prerilascio (una volta che il processo è in esecuzione, continua l'esecuzione fino alla terminazione o finchè si blocca in attesa di I/O o se richiede altri servizi) oppure con prerilascio (il processo al momento in esecuzione può essere interrotto e spostato in stato Ready dal sistema operativo quando arriva un nuovo processo, quando un interrupt trasferisce un processo dallo stato Blocked allo stato Ready oppure quando in base all'interruzione di clock). Le strategie con prerilascio provocano un overhead maggiore di quelle senza prerilascio, ma possono fornire un servizio migliore all'intero insieme dei processi, perchè impediscono ad ogni processo di monopolizzare il processore per molto tempo.

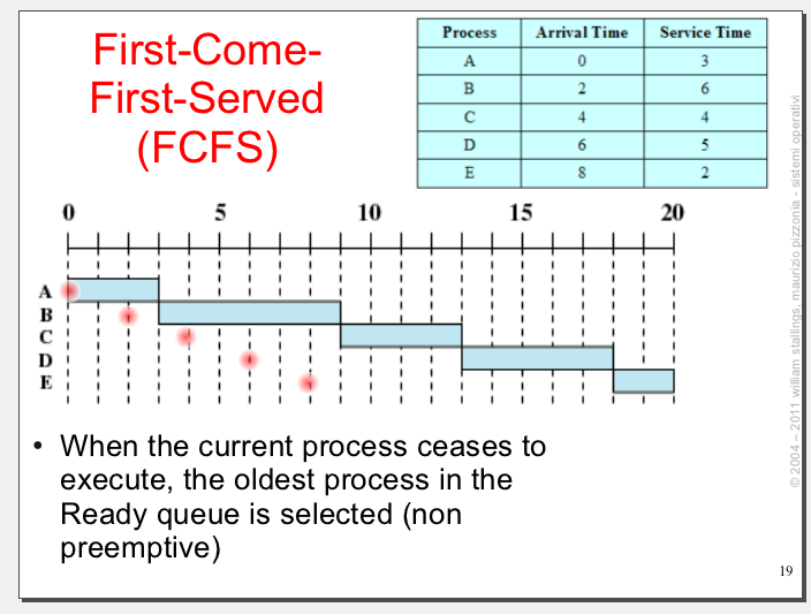
	FCFS	Round Robin	SPN	SRT	HRRN	Feedback
Funzione di selezione	max[w]	costante	min[s]	min[s - e]	$\max\left(\frac{w+s}{s}\right)$	(vedere il testo)
Modalità di decisione	Senza prerilascio	Con prerilascio (allo scadere del quanto di tempo)	Senza prerilascio	Con prerilascio (all'arrivo)	Senza prerilascio	Con prerilascio (allo scadere del quanto di tempo)
Throughput	Non enfatizzato	Può essere basso se il quanto è troppo breve	Alto	Alto	Alto	Non enfatizzato
Tempo di risposta	Può essere alto, specialmente se c'è una grande varianza nel tempo di esecuzione del processo	Fornisce un buon tempo di risposta per i processi brevi	Fornisce un buon tempo di risposta per i processi brevi	Fornisce un buon tempo di risposta	Fornisce un buon tempo di risposta	Non enfatizzato
Overhead	Minimo	Basso	Può essere alto	Può essere alto	Può essere alto	Può essere alto
Effetto sui processi	Penalizza i processi brevi e quelli I/O bound	Trattamento fair	Penalizza i processi lunghi	Penalizza i processi lunghi	Buon bilanciamento	Può favorire i processi I/O bound
Starvation	No	No	Possibile	Possibile	No	Possibile

w = tempo trascorso complessivamente nel sistema, in attesa e in esecuzione

e = tempo trascorso complessivamente in esecuzione

s = tempo di servizio totale richiesto dal processo, compreso e.

First Come First Served (FCFS): non appena un processo diventa Ready, d'inserisce nella coda dei processi Ready; quando il processo al momento in esecuzione si sospende, il più vecchio processo nella coda dei processi Ready è selezionato per l'esecuzione.



Questa tecnica è basata sulla tecnica FIFO e funziona meglio con processi lunghi e ha il problema di favorire i processi dipendenti dal processore (cpu bound) rispetto a quelli I/O bound: se il cpu bound è in esecuzione, tutti gli I/O bound devono attendere o nelle code I/O o nella Ready. Quando la cpu bound esce dalla cpu, quelli I/O diventano Running, ma si bloccano sugli eventi di I/O.

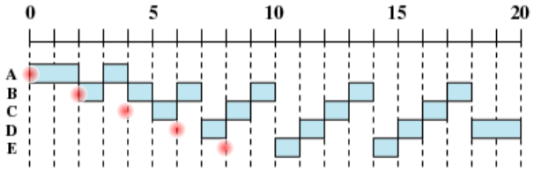
Il criterio di selezione max[w] rappresenta il massimo tempo di attesa in coda, cioè che il primo arrivato è il primo servito. Determino il tempo in cui si sospende il processo e da esso calcolo il tempo che il processo trascorre in coda (turnaround) e posso osservare il turnaround normalizzato che è il rapporto tra il turnaround e il tempo di servizio: più lungo è il tempo di esecuzione di un processo e maggiore è il ritardo assoluto totale che può essere tollerato.

Round Robin: per ridurre le penalità che i processi brevi subiscono con FCFS, è usare il prerilascio basato sul clock: un'interruzione di clock è generata ad intervalli periodici. Quando avviene un'interruzione, il processo in esecuzione al momento è inserito nella coda dei processi Ready e il successivo job Ready è selezionato con il FCFS. Allo scadere dello slot, viene sollevato un interrupt e il dispatcher seleziona il processo successivo nella coda, con una tecnica FCFS, se un processo viene bloccato per l'I/O viene messo nella coda I/O). Se il tempo da usare è molto breve, i processi

brevi passeranno abbastanza velocemente all'interno del sistema, anche se vanno evitati perchè c'è un tempo aggiuntivo di elaborazione dovuto alla gestione dell'interruzione di clock ed all'esecuzione delle funzioni di scheduling ed allocazione. Questo sistema è conveniente in un sistema orientato alle transizioni. La durata del quanto di tempo si decide in base alla grandezza minima del burst dei processi; le considerazioni sono sul fatto che se i CPU birst sono o meno trascurabili in quanto il round robin favorisce comunque i processi CPU bound e se non ho processi CPU bound allora in coda Ready i processi non attendono nulla. Un Round Robin è unfair verso i processi che vanno in blocco prima dello scadere del loro quanto di tempo: quelli I/O bound usano la cpu per poco tempo e poi si bloccano in attesa dell'operazione di I/O, mentre quelli CPU bound usano la cpu per tutto il loro quanto di tempo e non attendono nulla. Lo schema che lo risolve è quello del VRR ovvero mettendo in una coda con maggiore priorità quei processi che sono stati interrotti prima che il loro quanto di tempo sia scaduto. Questi processi opereranno per il tempo rimanente che gli era rimasto prima che venissero interrotti.

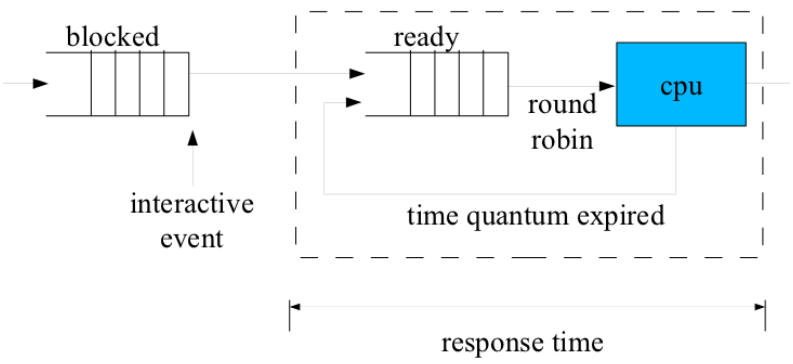
Round-Robin (RR), q=1

Process	Arrival Time	Service Time
A	0	3
B	2	6
C	4	4
D	6	5
E	8	2



- **preemption** based on a timer
- **time quantum** q: each process is allowed to use the processor for the time quantum and then preempted

RR and queues



Shortest Process Next (SPN): è una strategia senza prerilascio nella quale si sceglie come successivo processo da eseguire il processo con il minore tempo esecuzione previsto, facendo passare un processo breve in testa alla coda. La difficoltà sta nello stimare il tempo di esecuzione per ogni processo e i tempi previsti sono minori rispetto ai tempi reali.

$$S_{n+1} = \alpha T_n + (1-\alpha) S_n$$
$$\alpha \in (0,1]$$

Dove alfa è un peso costante che determina il peso relativo dato dalle osservazioni più e meno recenti (per esempio 0,8 considera le ultime 4 osservazioni, 0,2 le ultime 8), Tn è il tempo di esecuzione del processore per la n-esima istanza di quel processo e Sn è il valore previsto per la n-esima istanza. Se sto eseguendo un processo e intanto mi arrivano processi da fare più lunghi di ciò che rimane di quello che sto eseguendo, non mi interessa poiché questo approccio non è preemptive.

Shortest Remaining Time: lo scheduler sceglie sempre il processo che ha il minor tempo di esecuzione rimanente previsto e quando un nuovo processo è inserito nella coda dei processi Ready può avere un tempo rimanente più breve di quello del processo in esecuzione al momento: di conseguenza lo scheduler può dover prerilasciare ogni volta che un nuovo processo diventa Ready. Anche qui lo schedular deve avere la stima del tempo di elaborazione. È migliore rispetto al SPN poiché dà una preferenza immediata ai lavori brevi.

Feedback: se non si può prevedere i tempi di esecuzione dei job, si cercano quelli rimasti più a lungo in esecuzione. Lo scheduling è fatto su basi di prerilascio e si usa un meccanismo di priorità dinamico: quando un processo entra per la prima volta, è posto nella coda Ready, ma dopo la sua prima esecuzione (quando torna in stato Ready) cambia posizione. Ogni volta successiva è spostato nella coda successiva con minore priorità (più è lungo e più scenderà); all'interno di ogni coda si usa il meccanisco FCFS tranne l'ultima che è Round Robin. Ne sono avvantaggiati i processi più veloci e brevi e il feedback è in grado di dare più cpu ai processi I/O bound. Le due varianti (scala di priorità quando scade il suo quanto di tempo e aumenta di

priorità quando va in blocco) non sono soddisfacenti.

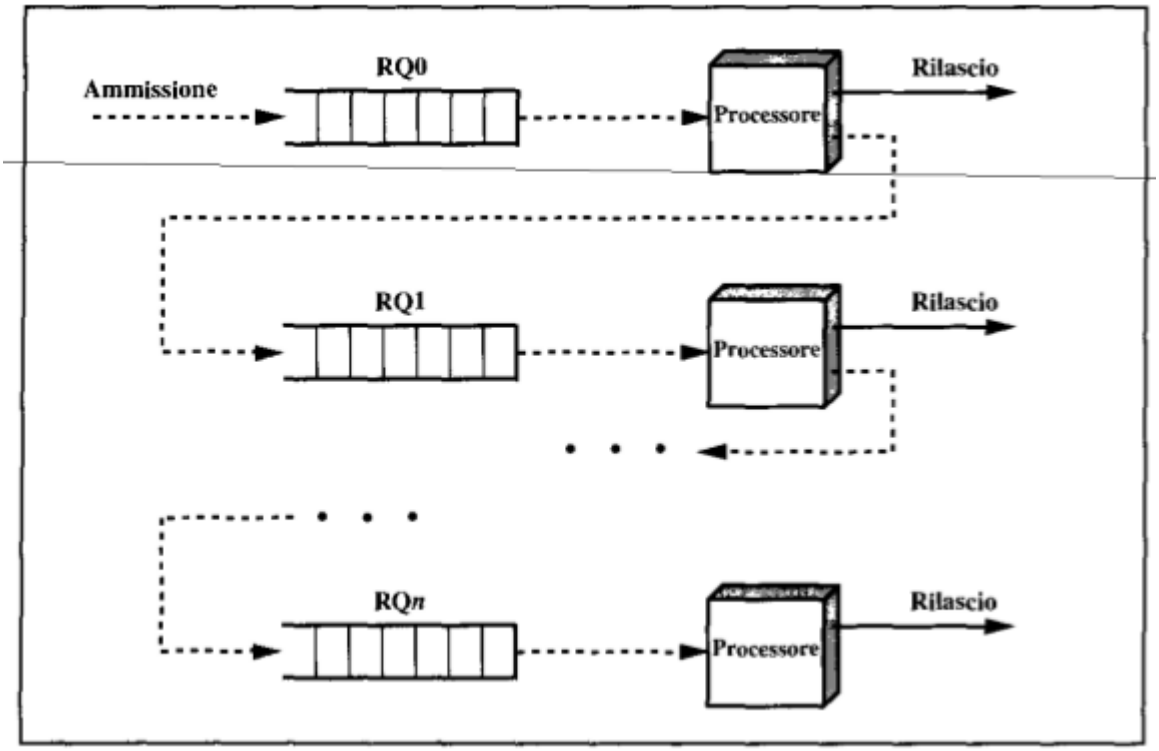


Figura 9.10 *Scheduling con feedback*

Scheduling in Linux: in Linux si ha un feedback con preemption; i processi in kernel-mode possono anch'essi essere interrotti. Il kernel 2.6 è un kernel preemptabel poiché fa una stima del cpu burst, le priorità più alte ce l'hanno i processi con cpu burst più piccoli e il comando nice dice quanto devo essere "gentile e carino" rispetto agli altri processi. Nei processi real-time si ha scheduling FCFS o RoundRobin con quanto di tempo definito dall'utente con priorità e preemption fissato a priori.

SCHEDULING DEL DISCO

Lo scheduling riguardante la gestione del disco è intesa come la scelta di quali porzioni del disco leggere a seconda delle richieste del sistema e dipende in primo luogo dai parametri legati alle prestazioni del disco stesso nell'accesso ai dati in esso memorizzati come il *seek time* (tempo di ricerca, cioè il tempo necessario alla testina per posizionarsi sulla traccia e si calcola con m(costante drive disco)*n(numero tracce)+s(tempo di avvio)), *rotational delay* (ritardo di rotazione, tempo necessario affinchè il settore giusto della traccia transita sotto la testina) e *transfer time* (tempo necessario al trasferimento dei dati, oltre che all'attesa per l'uso del dispositivo e del canale, nel caso questi risultino in altro modo occupati: dati vanno da disco → buffer interno →buffer controller →memoria).

L'obiettivo è di massimizzare il throughput (numero richieste servite nell'unità di tempo), "essere fair" rispetto i vari processi richiedenti, evitare il problema della starvation (due utenti di cui uno va avanti e uno no).

Sono varie politiche di scheduling:

- *Random*: seleziona casualmente le richieste di accesso al disco dalla relativa coda;
- *FIFO*: è FAIR (tutte le richieste sono servite e sono eseguite sequenzialmente secondo l'ordine di arrivo. Va bene per pochi processi in coda e se le richieste richiedono settori vicini, ma ci sono problemi se ho tanti processi e richieste di tracce a caso. Ha il compito di ottimizzare gli obiettivi del sistema operativo, piuttosto che gli accessi al disco;
- *LIFO*: esaudisce la richiesta dell'ultimo arrivato; è UNFAIR (preferisce i processi nuovi arrivati) e da starvation (le richieste più vecchie vengono soddisfatte solo se la coda viene svuotata completamente; è ottimo se il disco è libero;
- *SSTF (shortest service time first)*: sceglie sempre dalla coda la richiesta che richiede il tempo di ricerca (seek) minore; i vantaggi sono ce raggruppa di suo

richieste di tracce contigue, mentre gli svantaggi sono che dà stravation (quando ci sono richieste consecutive per tracce vicine) e unfair (ignora richieste lontane dalla testina;

- *SCAN*: soddisfa tutte le richieste presenti man mano che la testina si muove in una stessa direzione e arrivato all'ultima traccia, torna indietro ripercorrendo la stessa strada e fa le richieste che incontra; lo starvation avviene solo nel caso che ho letture su stessa traccia; non sfrutta la località: quando va in una direzione, se arriva una nuova richiesta di una traccia davanti la testina la fa, altrimenti se la richiesta richiede tracce dietro la testina, saranno eseguite al ritorno; è unfair (favorisce processi che richiedono tracce più esterne o interne e favorisce i nuovi processi in coda anche se le nuove richieste sono eseguite dopo molto tempo); il *C-SCAN è* simile a SCAN, ma limita lo scanning in una sola direzione. Esegue le richieste che incontra, poi arrivato in fondo ritorna all'inizio senza eseguire ulteriori richieste al ritorno e poi riparte, riducendo il ritardo massimo per le nuove richieste; è FAIR poiché non favorisce nessuno e dà starvation poiché si hanno letture su stessa traccia. Il *NstepScan* divide la coda in più sottocode, lavorando su una alla volta come se fosse una coda singola (per N= 1 →FIFO, N molto grande →SCAN tradizionale). *Fscan* lavora su due code: quando lo scan inizia, tutte le richieste si trovano in coda e le ulteriori informazioni vengono inserite nell'altra (la seconda coda viene servita allo svuotarsi della prima).

Se in coda ho richieste a tracce vicine tra loro, lo schedular può esaudire tutte queste richieste come fosse una richiesta sola. Tutti gli algoritmi visti non risolvono il problema nel quale un processo può mandare tutte le scritture che vuole disinteressandosene, mentre le letture sono di solito bloccanti, cioè quando si richiede si devono aspettare i risultati e quindi per continuare il lavoro un processo deve quindi attendere.

LINUX: i suoi sorgenti sono caratterizzati da 4 disk scheduler di cui uno solo è attivo su un certo dispositivo, e devono essere selezionati in fase di compilazione del

kernel per essere usati.

- Il *noop* è un FIFO con una richiesta di merging (non si usa praticamente mai).
- La *deadline* è uno schedular a scadenza, variante del one-way elevator (le richieste sono marcate con il tempo di arrivo). Si rischia di far fare avanti e indietro alla testina e rendere il processo inefficiente.
- *Anticipatory* è uno scheduler di default: si ferma aspettando che vada tutto avanti prima di fare la successiva operazione (aspetta 6ms).
- *Complete fair queueing* è quello di default oggi e comprende un po' di tutti i precedenti; è un approccio round robin sui processi e tende a schedulare ogni processo per un po' di millisecondi.

RAID

Il RAID È un insieme di dischi fisici visti dal sistema operativo come un unico drive logico; i dati sono distribuiti attraverso i drive fisici di un array. La capacità di ridondanza del disco è usata per memorizzare informazioni di parità, che garantiscono che i dati vengano recuperati in caso di guasto del disco. Permette testine multiple e attuatori che operino simultaneamente consentendo velocità di trasferimento più alte (aumentano però le probabilità di guasto).

RAID 0: i dati utente e i dati del sistema sono distribuiti su tutti i dischi dell'array, potendo eseguire operazioni in parallelo;

RAID 1: si ottiene duplicando tutti i dati per ottenere la ridondanza; una richiesta di lettura può essere servita da uno qualunque dei dischi che contengono i dati richiesti mentre una richiesta di scrittura richiede che entrambe siano aggiornate. Il recupero da un guasto è semplice.

RAID 2: accesso in parallelo (tutti i dischi partecipano ad ogni richiesta di I/O; il codice di correzione di errori viene calcolato sui bit corrispondenti su ogni disco, e i bit di codice sono memorizzati nelle corrispondenti posizionati in dischi di parità multipli. È ugualmente costoso come il RAID 1.

RAID 3: uguale al RAID 2, ma richiede solo un disco ridondante, indipendentemente dalla dimensione dell'array.

RAID 4: ogni disco opera in modo indipendente, così le richieste di I/O possono essere espletate in parallelo; viene calcolata una fetta di parità bit a bit tra tutte le fette corrispondenti in ogni disco di dati, e i bit di parità vengono inseriti nella posizione opportuna della fetta di parità contenuta nel disco di parità.

RAID 5: uguale al RAID 4, ma le fette di parità sono distribuite su tutti i dischi.

COMANDI LINUX

Una *shell* è un programma che permette agli utenti di comunicare con il sistema e comunicare e di avviare altri programmi. Di solito è una shell testuale con cui l'utente interagisce attraverso un terminale tramite un'interfaccia grafica. In UNIX la shell più nota è la Bash; altre shell conosciute sono la KSH (Korn Shell), CSH (C Shell) e TCSH (Tenex C Shell, come la CSH solo con alcune caratteristiche aggiuntive).

CODICI:

- ***man***: è la pagina del manuale (*man man* su terminale). Scrivendo man bash si ha tutto sulla bash.
- ***ls***: mostra il contenuto la directory corrente; ls -l è una versione più lunga (il numero prima del nome utente che compare è il numero di Hard Link, cioè di una voce di directory che condividono lo stesso dato). La **d** all'inizio di ogni riga rappresenta che quel link è una directory, mentre il **-** rappresenta un file. I successivi caratteri sono rappresentati a "triplette" (sono 3) e definiscono i permessi di lettura (**l**), scrittura (**w**) ed esecuzione (**x**) per utente, gruppo e tutti gli altri; le altre colonne indicano l'username, il gruppo, la dimensione in byte, la data di ultima modifica e il nome del file. ls -l -a mostra più cose, in particolare i file o le directory i cui nomi iniziano per punto (i file o directory nascoste); -la è una versione più compatta.
- ***q***: uscire dall'ambiente.
- ***Chmod***: modifica i permessi dei file (r, 4 → lettura | w,2 → scrittura | x,1 → esecuzione).
- ***pwd***: mostra la directory corrente (indirizzo).
- ***./nome_file***: esegue il file.
- **#!/bin/sh**: scrive un programma in bash, i caratteri **#!** informano il sistema che il file contiene una serie di comandi che devono essere passati all'interprete indicato: in questo caso identificano uno script di shell eseguibile. Il percorso successivo ai due simboli indica il percorso del programma che

deve interpretare i comandi contenuti nello script. #! può essere omesso se lo script è formato solamente da una serie di comandi specifici di sistema e non utilizza direttive interne alla shell.

- ***more*** o ***most*** o ***less*** : (pager) vedere un output in una maniera più o meno interattiva (less, more, most); less mostra il contenuto del file, una pagina alla volta.
- ***ls | less***: manda l'output di ls dentro less.
- ***cd***: cambia la directory corrente (cd / vado alla radice della macchina). cd / . identifica la directory stessa, metre cd / .. identifica la directory padre (root).
- ***echo***: fa la stampa.
- ***$***: rappresenta la home.
- ***pathname***: percorso. Si divide in assoluto (inizia con lo / e specifica la posizione di un file a partire dalla radice del file system) e relativo (descrive la posizione a partire da un'altra posizione dell'albero dei file, come la directory corrente senza fare uso quindi di identificatori della radice /).
- ***sudo***: permette di dare comandi da root senza esserlo.
- ***history***: storia dei comandi usati.
- ***ldd /bin /comanda_da_eseguire***: cerca nelle librerie.
- ***cat***: manda in output il contenuto dei file, nell'ordine in cui sono specificati. Senza i nomi dei file, scrive su standard output tutto quello che riceve da standard input finché non viene chiuso (da tastiera si chiude con Ctrl+d).
- ***env***: variabili d'ambiente. Le variabili d'ambiente sono coppie chiave-valore(stringa). Si possono creare (per esempio A=2). Svuotando il path non posso fare alcuni comandi come ls. Se dentro il path non avete un punto, l'eseguibile bisogna chiamarlo con ./nome_eseguibile. Esistono due variabili d'ambiente: esportate (quando si creano un figlio si esportano per averle anche nelle cartelle figlio) e non esportate. export -n a la rende non esportabile.
- ***exit***: torno alla shell padre.
- ***ps***: mostra una lista di processi attivi. ps aux rappresenta tutti i processi del sistema (R = Ready-runnung). Per “uccidere” un'altra bash attiva per esempio di può scrivere kill -9 n°bush (echo $$).

File System di Linux

Il file system ha una struttura gerarchica: tutti gli oggetti sono contenuti all'interno della root. La root non contiene soltanto tutte le directory di una partizione ma tutti gli eventuali file system montati sul sistema. L'utente non può creare lì dentro ma dopo che il root gli ha creato lo spazio l'utente può lavorarci.

- **BIN**: contiene comandi, programmi base e file eseguibili (binari) per tutti gli utenti. I file con la **L** sono dei link simbolici C.
- **SBIN**: contiene gli eseguibili utilizzati per la manutenzione del sistema.
- **ROOT**: è il file system principale da cui derivano tutti gli altri; il suo contenuto deve essere adeguato per l'avvio, il ripristino, il recupero e/o la correzione del sistema.
- **BOOT**: è il kernel di Linux e contiene tutti i file indispensabili al bootstrap del sistema; è una directory statica, non condivisibile. Viene caricato al momento del boot da un bootloader (programma che carica il kernel del sistema operativo) che si chiama grub (prima esisteva Lilo). I*nitrd* contiene i moduli da caricare all'avvio (driver). Memorizza i dati che vengono utilizzati prima che il kernel l'esecuzione di alcuni programmi (userspaces).
- **DEV**: su disco non è un file, ma viene letto “come file”: infatti si dice che contiene file “speciali” che corrispondono a dispositivi hardware o a funzionalità particolari (tramite di essi si può accedere al dispositivo).
- **HOME**: è la directory utente. Qui l'utente può scrivere, cancellare e installare programmi.
- **ETC**: contiene file di configurazione del sistema e dei programmi (spesso si trovano in formato di file di testo).
- **LIB**: contiene le immagini del kernel e delle librerie condivise essenziali necessari per avviare il sistema ed eseguire i comandi sotto /bin/ e /sbin/. Le librerie sono raccolte in routines (sequenze di istruzioni) dei programmi di uso frequente. Si identificano per la loro estensione .so (share object). libc è il c-runtime (ad esempio la printf).

- **LOST+FOUND**: si trovano il risultato di controlli del disco quando ci sono problemi come crash o arresto improvviso. Ci sono i file recuperati (anche se non sono sempre integri).
- **MOUNT**: vengono montati tutti i dispositivi.
- **MNT**: è il punto di mount generico per i file system; il montaggio è il processo di creazione del file systema disposizione del sistema. Dopo il montaggio, i file saranno disponibili sotto il punto di mount.
- **OPT**: contengono i pacchetti o i file statici di grandi dimensioni che non rientrano perfettamente nel file system possono essere collocati in questa cartella.
- **PROC**: informazioni di elaborazione; è un file system virtuale e contiene le informazioni del sistema runtime.
- **SRV**: sito di dati specifici in servizio al sistema. Lo scopo principale è che cosi gli utenti possano trovare il percorso del file dati per un particolare servizio per essere posizionati razionalmente.
- **TMP**: contiene i file temporanei.
- **USR**: è la gerarchia secondaria per i dati utente contenente la maggior parte delle applicazioni dell'utente; è condivisibile in sola lettura dati, quindi non può avere permessi di scrittura se non da amministratore. Contiene la quota maggiore di dati su un sistema.
- **VAR**: contiene i file variabili (ad esempio i file temporanei delle email). Deve contenere i file che il sistema deve essere in grado di scrivere durante il funzionamento.

COMANDI UNIX

- ***cp***: esegue la copia dei file. (-r: ricorsivo(copia le sottodirectory) -v: vede passo passo cosa sta facendo; l'uso del punto indica la directory corrente)
- ***rm***: cancella uno o più file (con -r prima le directory e poi tutte le sotto directory).
- ***mkdir***: crea una directory vuota.
- ***rmdir***: cancella una directory vuota.
- ***touch***: cambia la data di un file.
- ***vi***: editor di testo (sconsigliato).
- ***nano***: editor (poche funzioni).
- ***emacs***: editor di testo.
- ***apt-get install***: nome_pacchetto: installare un pacchetto.
- ***pstree***: processi in albero.
- ***alias***: assegna ad una stringa un certo comando o funzione. Per metterli fissi e non solo per la durata di quelle operazioni, si può modificare il file di testo con i codici che si può aprire scrivendo su terminale *gedit .bashrc*.
- ***kill***: uccide un processo (per esempio specificando il numero del processo).

Il *PID* numero del processo; il numero 1 è *init*: il kernel lo manda in esecuzione durante il bootstrap e ha il compito di portare il sistema in uno stato operativo.

Se sul terminale scrivo: *alias pippo=pwd* assegnerò a “pippo” il comando di pwd.

Il job control (shell) si riferisce alla possibilità di mettere i processi in background (sullo sfondo) e di riportarli in foreground (in primo piano), cioè poter eseguire un processo mentre voi state facendo qualcos'altro, ma mantenendogli il controllo per potergli comunicare qualcosa o poterlo fermare.

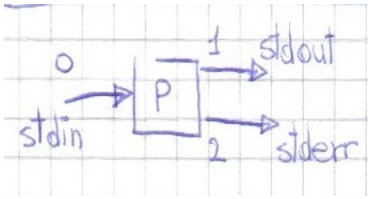
Sotto unix i processi possono ricevere notifiche dall'esterno detti segnali: non contengono dati. Questi segnali possono essere gestiti dal kernel (tipo particolare) come il (***kill -l***)

sigkill -9: uccide definitivamente un segnale;

sigstop: congela il processo;

sighterm: chiede al processo di chiudersi (se il processo si è impallato, non si chiude).

COMANDI UNIX



cat <nome file> <#2> <#3>: mette i file in stdout (schermo) tutti concatenati.

<comando> < pippo: prendo input da file pippo.

<comando> > pippo: output di comando si scrive su file pippo.

<comando> >> pippo: concatenato.

cat 1>pippo 2>pluto: scrive sul file pippo il contenuto di 1 e sul file pluto il contenuto di 2.

grep: seleziona.

awk: linguaggio per “manipolare”.

Cat /etc/bin | sort passwd | tr “:” “_”: prende un carattere : e lo trasforma in _.

Cat /etc/bin | sort passwd | tee pluto: si fa una copia sul file pluto.

ESERCIZI FILTRI

Pipeling: avere l'input di un processo preso dallo standard output di un altro processo (lo standard error non è toccato e va sempre sul terminale)

tr: prendi da input un file e trasforma carattere per carattere (tr " (spazio)" "/n" → trasforma spazi in "a capo").

Sort -n -r -k2 (k2 prendi il secondo campo).

Wc: conta linee parole bite caratteri;

head: da le prime righe (di defoult 10 ma posso decidere per esempio mettendo -2' e da le prime venti) di un input;

tail: da le ultime righe di un input (le stesse regole dell'head);

uniq: analizza l'input e quando incontra due righe consecutive identiche, ne stampa una sola (-c permette inoltre di stampare quante volte è ripetuta una copia); ha senso insieme a sort;

sort |uniq -c| sort -n: mette i comandi in ordine di occorrenza; il primo sort serve perchè altrimenti non è detto che le righe uguali siano vicine.

Sort -u = sort |uniq|

ps aux | wc -c: n° dei processo compreso l'header (per levarlo ***--no-header***).

GREP

grep: trova testo all'interno di un file

[a-z]: trova caratteri, parole.

[A-Z a-z_] [A-Z a-z_0-9]: trova programmi java.

-i ignora Maiuscolo/minuscolo,

-r scende le directory in modo ricorsivo,

-v inverte il senso della ricerca, (con “^$” vicino elimina le righe vuote;

(se si vuole cercare gli spazi bisogna mettere le virgolette)

[a,b,c] uno tra questi caratteri

L'asterisco * serve per far si che per esempio [ab c] permette di ripetere il carattere quante volte vuoi

(abaa)

orange@orange-pc:~$ cat /etc/passwd | grep "i[qzr]*o"

albero di passing: tra caratteri consecutivi ci sta l'operatore concatenazione

conc → i

conc → o

conc → *

asterisco * → [qzr] sarebbe un or tra quei valori

L'ordine deve essere rispettato, cioè OI non viene cercato.

egrep: supporta le parentesi;

(ab)* → abababab

(a | bcd)* → aaa abcd bcd

. = carattere qualsiasi (.* carattere qualsiasi quante volte vuoi ____ orange.*/bin tutto quello che c sta tra

orange e bin) ^: non matcha le ripetizioni

una riga vuota matcha cappelletto dollare

^pippo$:

DATE TUTTE LE RIGHE CHE HANNO NEL 3° CAMPO UN NUMERO PARI

cat /etc/passwd | grep “^[^:]*:[^:]*:[0-9]*[02468]:”

awk: permette di scrivere filtri generici , matcha espressioni regolari.

pattern statement: se manca stampa come grep (sono opzionali).

awk '/pizzonia/': stampa tutte le righe con pizzonia; se aggiungiamo {print “xxx” $} mette delle x prima di pizzonia.

cat <nome file> <#2> <#3>: mette i file .

awk: come primo paramento si aspetta lo script del filtro che deve eseguire (tra apici).

orange@orange-pc:~$ ps aux | awk '/orange/ {print "x " $0}': x spazio riga corrente.

orange@orange-pc:~$ ps aux | awk {s = s+1; print s " " $0}': numera le righe.

orange@orange-pc:~$ ps aux | awk {print $1; print $2}': seleziono primo e secondo campo.

orange@orange-pc:~$ ps aux '$1 == “orange” && $2 > 30000'| awk {print $1; print $2}'

{if ($1 == pizzonia && $2>30000)}} }: seleziono delle righe in base ai contenuti dei campi; lo spazio tra le ultime due parentesi graffe indica che il then non fa nulla.

cat/etc/passwd | awk 'NR==3': mi permette di numerare i record e si aggiorna da solo; BEGIN inizializza in maniera esplicita una variabile e viene eseguita prima dell'elaborazione della prima riga; END viene eseguito dopo l'elaborazione dell'ultima riga.

ps aux | awk 'begin {s=0} {s=s+2 $2} END {print s}': sommare i pid di ps aux.

In aux tutte le variabili hanno tipo stringa (“25” + “1” = “26” oppure “0” = “ “ = “[^0 – 9].*

awk 'begin {print “25” + “a”}': 25.

ctrl -d: end file.

Un frame pulito può essere subitoriassegnato mentre uno sporco deve esse prima scritto sul disco

ps aux | less: RSS dice quanto è la somma dello spazio occupato dei frame in memoria fisica.

VSZ è la somma dello spazio occupato da tutte le regioni. Se entrambi sono uguali a 0, abbiamo

processi che hanno i privilegi del kernel ma sono schedulati come processi.

acho $$: esce il PID del processo (dalla shell);

cd / proc: troviamo tante directory;

cd 17291: il PID che mi è uscito prima con tante funzionalità;

less maps: una riga per ciascuna regione (indirizzo, permesso, “qualcosa, “qualcosa”, /nome).

orange@orange-pc:~$ ps aux | awk '{s=s + $6} END {print s}': 1367284

orange@orange-pc:~$ ps aux | awk 'BEGIN {s=0} $q=="root" {s=s + $6} END {print s}': 0

ps aux | less

campo 6 = RRS e voglio fare la somma del sesto campo.

ps aux | awk '{s = s + $s} end {print s}'

ps aux | awk '{begin{S=0} {if ($1==”root”) {s = s + $6}} end {print s}'

ps aux | awk '{begin{S=0} $1==”root” {s = s + $6}} end {print s}'

ps aux | awk '{begin{S=0} /^root/ {s = s + $6}} end {print s}'

matcho tutti i PID minori di 10:

ps aux | awk '{begin{S=0} $2<10 {s = s + $6}} end {print s}'

matcho con il campo 1:

ps aux | awk '{begin{S=0} $1~/^root/ {s = s + $6}} end {print s}'

NR, number of record → parte da 1 e restituisce il numero della riga:

ps aux | awk '{n = n+1; print n $0}'

stampa il numero della riga e la riga (head: stampa le prime dieci righe)

ps aux | awk '{n = n+1; print n " " NR " " $0}' | head

NF → numero di campi che ha letto e separato

ps aux | awk '{n = n+1; print n " " NR " " $0}' | head

$NF : ultimo campo letto della riga corrente (NF – 1 è la penultima riga)

manda i campi e li manda a capo

ps aux | awk '{for (i=1; i<=NF; i++) print $1}' | less

FS = " " field separator

RS = "/n" record separator

Se voglio specificare il separator:

cat/etc/passwd | awk 'begin {fs=":"} {print $1 " " $3}'

posso stabilire all'esterno una variabile

cat/etc/passwd | awk -v fs=":" {print $1 " " $3}'

cat/etc/passwd | awk -v rs=" " -v fs="\n" {print $1}'

cat/etc/passwd | awk -v rs=" " -v fs="\n" {print $1 " " "2}'

ESEMPI

orange@orange-pc:~$ echo a b c

a b c

orange@orange-pc:~$ echo a{1,2,3}

a1 a2 a3

orange@orange-pc:~$ echo a{1 ,2,3}

a{1 ,2,3}

orange@orange-pc:~$ echo a{1\ ,2,3}

a1 a2 a3

orange@orange-pc:~$ echo {1,2,3}p{a,b,c}

1pa 1pb 1pc 2pa 2pb 2pc 3pa 3pb 3pc

orange@orange-pc:~$ echo ~root

/root

orange@orange-pc:~$ echo $HOME

/home/orange

```
orange@orange-pc:~$ proca1=ciao

orange@orange-pc:~$ proca2=ciao

orange@orange-pc:~$ echo $proca1 $proca2

ciao ciao


orange@orange-pc:~$ echo "cpisadjcd $proca1 jcsipodjci"

cpisadjcd ciao jcsipodjci


orange@orange-pc:~$ echo 'cpisadjcd $proca1 jcsipodjci'

cpisadjcd $proca1 jcsipodjci
```