# Sistemi operativi

Un sistema operativo è il sistema che consente l'elaborazione dei dati su un calcolatore. Nella maggior parte dei casi l'utente interagisce solo con il sistema operativo, ma non con la macchina: il SO è quindi l'interfaccia tra i programmi che utilizziamo e la macchina.

I primi SO videro la luce nella seconda generazione di calcolatori (1955-65) ed erano di tipo batch (a lotti): il merito è dovuto all'introduzione dei transistor nell'elettronica ma il loro costo, tuttavia, ne impediva l'utilizzo al di fuori delle università o delle grandi industrie. Il fatto che le schede perforate non dovessero più essere inserite singolarmente dall'utente rese i calcolatori molto più rapidi ed efficienti.

Nella terza generazione di calcolatori, un problema ricorrente consisteva nel fatto che se questi lotti avessero richiesto operazioni molto lunghe e complesse avrebbero tenuto la CPU occupata per molto tempo, impedendo l'utilizzo del calcolatore ad altri utenti o ad altri programmi. Per questo motivo nei sistemi multiutenti ebbe la meglio la logica del time sharing: in questo modo ogni utente aveva la CPU a completa disposizione per intervalli di tempo regolari, e le operazioni eseguite per ciascuno non influenzavano l'esperienza degli altri. L'overhead per la gestione della CPU tuttavia poteva diventare significativo con molti processi attivi oppure con time slice molto piccolo.

| Sistemi batch: | Time sharing: |
|---|---|
| - Massimizzano l'utilizzo del processore<br>- Controllati con un job control language | - Minimizzano i tempi di risposta<br>- Ricevono comandi da terminale |

Con l'ulteriore progresso dell'elettronica, i calcolatori della quarta generazione (1980-1990) sono ormai tutti basati sulla tecnologia VLSI (Very Large Scale Integration), e non di rado si aveva più di un processore da gestire. Per la stessa ragione, i costi si abbassarono drasticamente permettendo così di permettere la nascita delle Workstation e dei cosiddetti Personal Computer, meno potenti ma a disposizione di tutti: i SO che dominavano questo settore furono principalmente MS-DOS (destinato all'utilizzo da parte di un singolo utente per volta) e UNIX.

La quinta generazione riguarda invece i dispositivi mobili quali PDA (Personal Digital Assistant) e telefoni cellulari, che, dovendo essere alimentati da batterie di capacità contenute, erano caratterizzati da memorie limitate, processori non performanti e schermi molto ridotti.

La crescita di reti di PC e di WS ha permesso lo sviluppo di:

- Network Operating Systems
    - l'utente "vede" più calcolatori, può accedere a macchine remote e copiare file; ogni macchina ha il suo sistema operativo locale
- Distributed Operating Systems
    - appare all'utente come un tradizionale sistema monoprocessore anche se è composto da più processori; l'esecuzione di programmi può essere a carico di macchine diverse (anche in parallelo)
    - gli N processori possono condividere o meno clock e/o memoria (loosely o tight coupled)

Inoltre divenne sempre crescente la presenza di sistemi real-time al servizio di una specifica applicazione che ha vincoli precisi nei tempi di risposta: il SO garantisce un tempo massimo entro il quale il programma deve essere mandato in esecuzione a seguito di un evento (gestione di strumentazioni, controllo di processi, gestione di allarmi oppure sistemi transazionali). In generale, si può parlare di sistemi real-time quando i tempi di risposta dalla richiesta al completamento dell'esecuzione del processo è sempre minore del tempo prefissato.

- Strutture dei sistemi operativi

*Gestione dei processi*
Un processo è un programma in esecuzione: ogni processo per soddisfare i compiti richiesti necessita di risorse quali tempo di CPU, memoria, file o dispositivi di I/O. Il sistema operativo è responsabile della creazione, sospensione, ripristino e cancellazione dei processi, con opportuni meccanismi di sincronizzazione e comunicazione tra i singoli (nel caso in cui ce ne siano molteplici in esecuzione allo stesso tempo).

*Gestione della memoria centrale*
La memoria è condivisa dalla CPU e dai sistemi di I/O ed è l'unico indirizzabile direttamente dal processore; non è possibile eseguire operazioni che non siano in memoria. Per migliorare l'utilizzo della CPU e le prestazioni complessive del sistema è necessario quindi poter gestire più programmi in memoria secondo schemi opportuni (che dipendono dall'hardware utilizzato).

*Gestione della memoria secondaria*
Il sistema operativo è responsabile di gestire lo spazio libero, allocare lo spazio e fare lo scheduling del disco. Dato che l'uso del disco è molto frequente la gestione deve essere efficiente vi è stato perciò uno studio molto accurato per la ricerca degli algoritmi migliori.

*Gestione del sistema di I/O*
Il sistema operativo "nasconde" l'hardware all'utente attraverso i driver: l'utilizzatore non deve infatti interessarsi alle caratteristiche fisiche del particolare dispositivo collegato per poterlo utilizzare.

*Gestione dei file*
Allo stesso modo, l'utente è all'oscuro delle particolari caratteristiche dei supporti di memoria (es: nastri, dischi) o dei metodi di accesso ai dati e la velocità del trasferimento. Il SO ha una visione logica uniforme del processo di memorizzazione delle informazioni: ai suoi occhi esiste un solo oggetto. Il file è la sola unità logica di memoria: il SO gestisce infatti la creazione, manipolazione e cancellazione di file e directory.

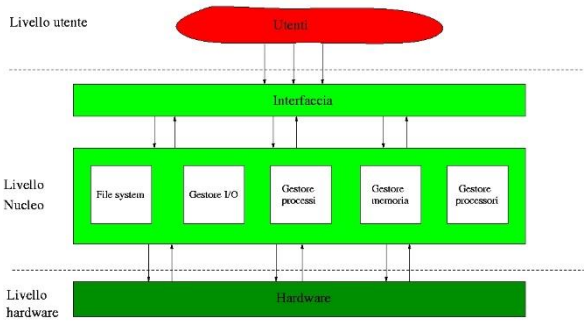
*Gestione delle reti*
Un sistema distribuito è un insieme di unità che non condividono la memoria, i dispositivi periferici o un clock: ogni unità dispone di una propria memoria locale, comunica con le altre tramite reti utilizzando protocolli noti. Dal punto di vista dell'utente, un sistema distribuito è un insieme di sistemi fisicamente separati, organizzato in modo tale da apparire come un unico sistema coerente, con una singola memoria principale ed un unico spazio di memoria di massa. Si rivela utile per file system distribuiti.

L'accesso ad una risorsa condivisa permette di accelerare il calcolo, di aumentare la disponibilità di dati e di incrementare l'affidabilità.

*Gestione delle protezioni*
Un sistema operativo multiutente consente che i processi vengano eseguiti in concorrenza: ogni processo dunque deve essere protetto dalle attività degli altri. Per far ciò occorrono opportuni meccanismi di autorizzazione. La protezione è il meccanismo che controlla l'accesso da parte di programmi, processi o utenti alle risorse di un sistema di calcolo. L'amministratore di sistema possiede tutti i permessi per l'accesso alla memoria; gli unici dati a cui non può accedere sono i dati protetti da crittografia.

*Interprete dei comandi*
Si occupa di interagire direttamente con l'utente e accettare i comandi che vengono inseriti. L'interfaccia può essere di diversi tipi: i più comuni sono la shell e le interfacce grafiche, ma nei primi sistemi multiutente potevano essere accettate solo istruzioni di controllo (successivamente è stata introdotta proprio la shell).

- Interazione col sistema operativo




Un sistema operativo è principalmente costituito da un nucleo (kernel) e dall'interfaccia con l'utente: esso può essere concepito come un programma che viene eseguito al momento del boot e che mantiene il controllo della macchina fino al momento in cui deve cederlo a un processo esterno. Il SO può essere sostituito o può essere possibile aggiornare il kernel: naturalmente, questa operazione è molto delicata.

*System call*

L'interfaccia tra il SO e i programmi degli utenti è definita da un insieme di istruzioni estese, ovvero system calls (chiamate di sistema), che creano, cancellano e usano oggetti software gestiti dal sistema operativo. La loro implementazione può essere diversa a seconda del sistema operativo utilizzato.

I principali tipi di chiamate di sistema sono:

- controllo dei processi
- gestione dei file
- gestione dei dispositivi
- gestione dell'informazione
- comunicazioni.

I programmi dell'utente, quindi, comunicano con il sistema operativo e gli richiedono servizi per mezzo proprio delle chiamate di sistema. A ogni system call corrisponde una procedura di libreria (che il programma può invocare) che ha i seguenti compiti:

- mette i parametri della system call in un sito predefinito (es. registro)
- istanzia un'istruzione TRAP per mandare un interrupt al sistema operativo
- nasconde i dettagli dell'istruzione TRAP
- rende le system call come chiamate di procedura normali (es. da programmi in C)

*Programmi di sistema*

I programmi di sistema offrono un ambiente per lo sviluppo e l'esecuzione dei programmi. Possono essere classificati in programmi per:

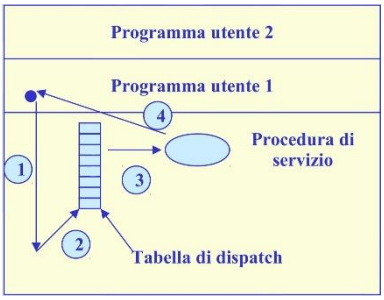
- gestione dei file
- informazioni di stato
- modifica dei file
- ambienti di sviluppo
- caricamento ed esecuzione dei programmi
- comunicazioni
- programmi applicativi

Dal punto di vista dell'utente la maggior parte delle operazioni è vista come esecuzione di programmi, non chiamate di sistema.

- Struttura dei sistemi operativi

*Sistemi monolitici*

Nei sistemi monolitici il sistema operativo è costituito da una collezione di procedure, ognuna delle quali può chiamare qualsiasi altra; l'unica struttura presente sono le system call, che comportano il salvataggio dei parametri e l'esecuzione di una trap speciale, detta kernel call o supervisor call.




Una chiamata di sistema può essere realizzata in questo modo:

1. il programma utente esegue una trap verso il nucleo;
2. il sistema operativo determina il numero del servizio richiesto;
3. il sistema operativo individua e chiama la procedura di servizio;
4. viene restituito il controllo al programma utente.

Un problema relativamente frequente con i primi sistemi operativi consisteva nell'esecuzione non voluta di un ciclo infinito, che comprometteva l'utilizzo della macchina e costringeva l'utente a riavviare il sistema.

I sistemi che condividono risorse devono quindi garantire che programmi non corretti non interferiscano con gli altri processi in esecuzione. L'hardware (mode bit) permette al sistema operativo di operare in due modalità differenti:

- User mode: normale funzionamento dei programmi utente
- Monitor mode (o kernel/system mode): operazioni effettuate dal sistema operativo

Quando avviene un interrupt il controllo passa al kernel mode.

Alcune istruzioni (privileged instructions) possono essere eseguite solo in kernel mode, tra cui tutte le istruzioni di I/O: occorre garantire che nessun programma possa essere eseguito in kernel mode.

Nei sistemi UNIX:


*Struttura a tre strati*

1. Un programma principale che richiama la procedura di servizio richiesta
2. Un insieme di procedure di servizio che eseguono le chiamate di sistema
3. Un insieme di procedure di utilità che forniscono il supporto alle procedure di servizio

*Sistemi multilivello*

Questi tipi di SO sono organizzati in una gerarchia di livelli.
Il 1° sistema con una struttura di questo tipo è il THE, realizzato alla Technische Hogelschool Eindhoven in Olanda da Dijkstra nel 1968 per il computer Electrologica X8. THE ha 6 livelli:

5. Operatore
4. Programmi utente
3. Gestione I/O
2. Comunicazione processo-console
1. Gestione della memoria a tamburo
0. Allocazione della CPU e multiprogrammazione

*Struttura a microkernel*

Si sposta quanto possibile dal kernel allo spazio utente. Le comunicazioni avvengono tramite messaggi.
Vantaggi:

- Estendere un microkernel è più facile

- Maggiore portabilità del sistema operativo
- Più affidabilità (il codice critico nel kernel è minore)
- Più sicuro

Svantaggi:
- Overhead dovuto alle comunicazioni fra spazio utente e kernel

*Modello Client-Server*
Si basa sull'idea di portare il codice ai livelli superiori, lasciando un kernel minimo: l'approccio è quello di implementare la maggior parte delle funzioni di sistema operativo nei processi utente.
- Processo Client: richiede un servizio
- Processo Server: gestisce le richieste del client

Il kernel si occupa di gestire la comunicazione tra client e server.

Il sistema operativo viene suddiviso in più parti, piccole e maneggevoli, ognuna delle quali si riferisce ad un aspetto del sistema (memoria, file, ecc.).

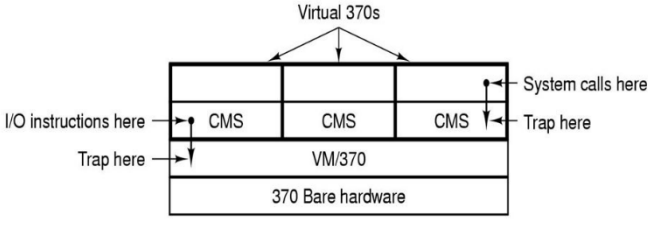
Vantaggi:
- Adattabilità all'utilizzo in sistemi distribuiti
- Alcune funzioni di sistema operativo non sono eseguibili da programmi nello spazio utente
- Due possibili soluzioni:
    - I processi server critici vengono eseguiti in modalità kernel
    - SI crea una divisione tra:
        - meccanismo, che ha dimensioni minime e viene implementato nel kernel
        - decisioni di strategia, lasciate ai processi server nello spazio utente

*Moduli*
I più moderni sistemi operativi implementano un approccio modulare del kernel: questo tipo di approccio è caratterizzato da:
- Programmazione orientata agli oggetti
- Componenti separate che comunicano tra loro attraverso interfacce definite
- Può essere caricato a sistema avviato

Approccio simile ai livelli ma più flessibile.

*Macchine virtuali*
Il sistema VM/370 (1972) si basa sulla separazione netta tra le funzioni di multiprogrammazione e di implementazione della macchina estesa. Il Monitor della macchina virtuale gestisce la multiprogrammazione fornendo più macchine virtuali, ognuna delle quali è una copia esatta dell'hardware.

Diverse macchine virtuali possono supportare sistemi operativi diversi: in questo modo in un ambiente multiutente utenti diversi potevano usare SO differenti. Uno dei sistemi utilizzato per gestire queste specifiche è il CMS (Conversational Monitor System), un sistema interattivo per utenti time-sharing.




Vantaggi:
- Ogni parte è più semplice, più flessibile e più facile da mantenere
- Una macchina virtuale permette una completa protezione delle risorse di sistema
- Rende facile lo sviluppo di un sistema operativo (le normali operazioni non vengono bloccate)

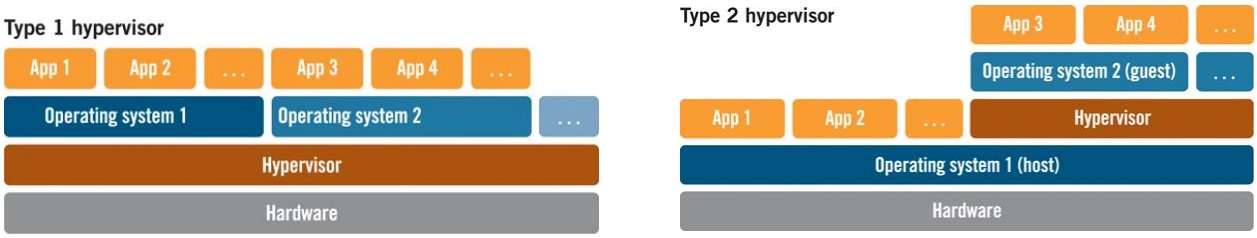
Svantaggi:
- L'isolamento delle macchine virtuali rende complessa la condivisione di risorse

*Modalità 8086*

Un approccio simile è utilizzato nei sistemi Windows con processore Pentium, che può funzionare in modalità virtuale 8086: in tale stato la macchina si comporta come un processore 8086 (indirizzamento a 16 bit e limite di memoria a 1M) e permetteva, nei sistemi che la utilizzavano (Windows, OS2, NT), di eseguire vecchi programmi MS-DOS.

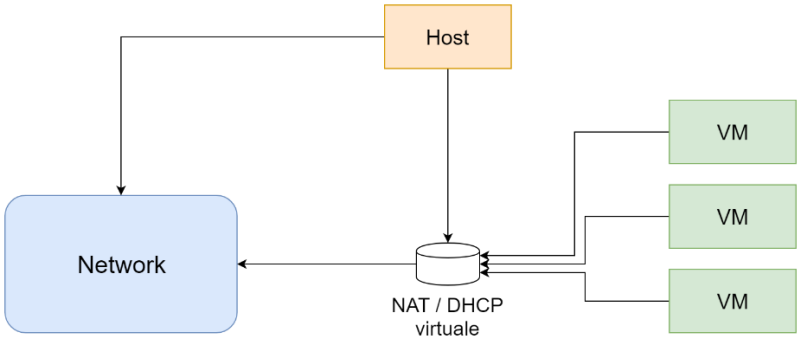
Finché eseguono istruzioni normali lavorano direttamente sull'hardware della macchina, nel momento in cui fanno richieste al sistema operativo o istruzioni dirette di I/O, l'istruzione viene intercettata dalla macchina virtuale.

*Hypervisor*




Sistema utilizzato da tutte le macchine virtuali odierne: l'utente deve solo preoccuparsi di avviare la macchina virtuale (solitamente costituito da un singolo file di grandi dimensioni).

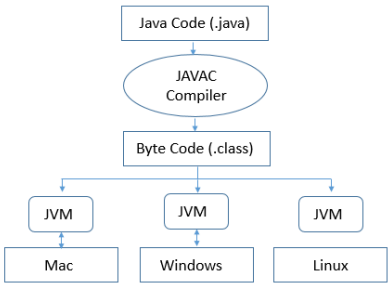
*Rete virtuale VMware*




*Java Virtual Machine*

È un calcolatore astratto che consiste di:

- Un caricatore delle classi
- Un verificatore delle classi, che controlla gli accessi alla memoria
- Un interprete, che può essere un programma che interpreta gli elementi del bytecode uno alla volta o un compilatore istantaneo (just in time - JIT).




Gestisce inoltre la memoria procedendo in modo automatico alla sua ripulitura (Garbage Collection – recupero della memoria non utilizzata).

- Panoramica sulla virtualizzazione

Le istruzioni vengono divise in tre gruppi:

- Istruzioni privilegiate (generano una trap solo se si è in user mode)
- Istruzioni di controllo (modificano le risorse di sistema, p.e. I/O)
- Istruzioni il cui risultato dipende dalla configurazione del sistema

Condizioni sufficienti perché una architettura possa essere virtualizzata (**condizioni di Popek e Goldberg**):

- Le istruzioni di controllo devono essere un sottoinsieme delle istruzioni privilegiate
    - Tutte le istruzioni che possono influenzare l'esecuzione del Virtual Machine Monitor devono generare una trap ed essere gestite dal VMM stesso
    - Le istruzioni non privilegiate possono essere eseguite nativamente.

*Binary rewriting*

È il metodo attraverso il quale le istruzioni vengono riscritte le istruzioni virtuali in istruzioni reali.

- Vengono esaminate le istruzioni del flusso del programma (a tempo di esecuzione)
- Vengono individuate le istruzioni privilegiate
- Vengono riscritte queste istruzioni con le versioni emulate
- Permette la virtualizzazione in spazio utente
- Più lento
- Si deve usare il caching delle locazioni di memoria
- Le prestazioni tipiche vanno dall'80% al 97% di una macchina non virtualizzata

Può essere implementato con i meccanismi dei debugger come i breakpoint

*Tipi di virtualizzazione*

- Emulazione
    - Permette l'esecuzione di un SO su una CPU completamente differente
    - Poco efficiente
- Virtualizzazione piena
    - Esegue copie di SO completi
- Paravirtualizzazione
    - Il sistema operativo ospitato deve essere modificato

*Paravirtualizzazione*

La paravirtualizzazione presenta le seguenti caratteristiche:

- Se un'istruzione del sistema guest genera una trap, questo ne deve gestire le conseguenze
- Il sistema guest deve essere modificato
- Concettualmente simile al binary rewriting, ma il rewriting avviene a tempo di compilazione
- Quando un'applicazione genera una systemcall, questa viene intercetta dal SO (guest)
- Quando il SO guest prova ad eseguire istruzioni privilegiate, il VMM "intrappola" (traps) l'operazione e le esegue correttamente
- Quindi i SO guest effettuano Hypercall per interagire con le risorse del sistema

## Introduzione ai processi

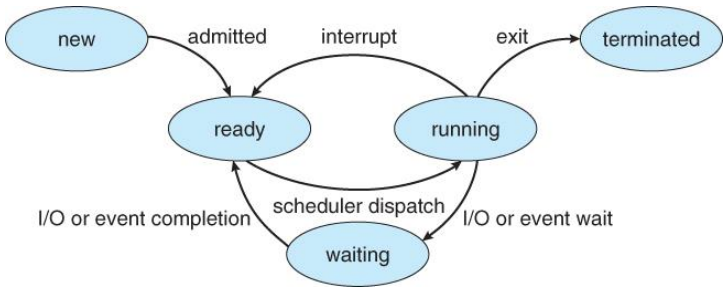
Come già anticipato, un SO può permettere ad un utente di creare processi (lanciare programmi), interallacciare l'esecuzione di diversi processi (per massimizzare l'uso del processore e contemporaneamente fornire un tempo di risposta ragionevole), allocare loro risorse e permettere le comunicazioni fra di essi.

Il processo, quindi, è il concetto principale di ogni Sistema Operativo e consiste nell'astrazione di un programma in esecuzione. I moderni calcolatori eseguono più azioni contemporaneamente (eseguire un programma, leggere un file di dati, stampare ecc.) grazie alla multiprogrammazione: essa garantisce un pseudoparallelismo, poiché consente un rapido cambiamento nell'operato della CPU tra vari programmi, mentre in parallelo lavorano le periferiche. (vedere slide 4)

Con questo modello, tutto il software eseguibile su un calcolatore (eventualmente compreso il sistema operativo) è organizzato in un certo numero di processi sequenziali, la cui durata (per ogni computazione) non è stabilita a priori, non è uniforme e non è identica ad ogni esecuzione: per questo motivo i processi non devono essere programmati con assunzioni a priori rispetto al tempo.

*Stati di un processo*

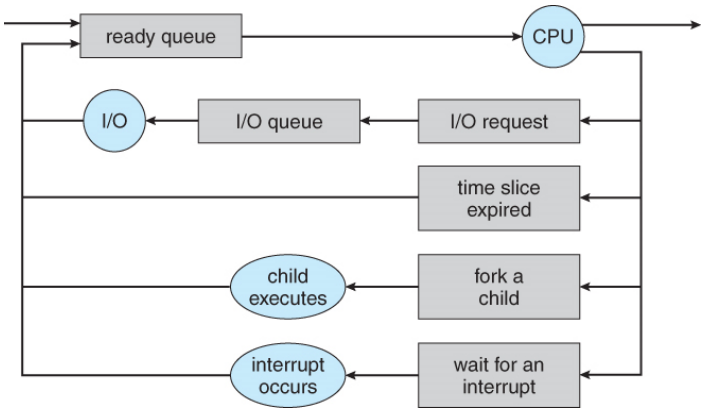
1. Quando un processo in esecuzione chiede un servizio di I/O al sistema operativo si blocca in attesa del risultato
2. Al termine del time slice il controllo torna al SO; lo scheduler decide a chi affidare la CPU
3. Uno dei processi in attesa ottiene la CPU
4. Quando un processo ha completato le operazioni di I/O viene rimesso dal SO nella coda dei processi in attesa.




Con un modello di questo tipo il livello più basso del sistema operativo è lo scheduler e gli altri processi sono al di sopra di esso: la gestione delle interruzioni e i dettagli sulla gestione dei processi sono dunque lasciati all'interno dello scheduler.

*Implementazione dei processi*

Per implementare il Modello del Processo, il SO ha un array di strutture detto **tabella dei processi** con un entry (PCB Process Control Block) per ogni processo: ogni entry è caratterizzato da:




- stato del processo
- program counter
- stack pointer
- allocazione della memoria
- stato dei suoi file aperti
- informazioni sullo scheduling
- informazioni circa i salvataggi necessari per il mantenimento della consistenza.

I campi nella tabella riguardano la gestione dei processi, della memoria e del file system.

*Creazione dei processi*

La creazione di un processo richiede i seguenti passaggi:

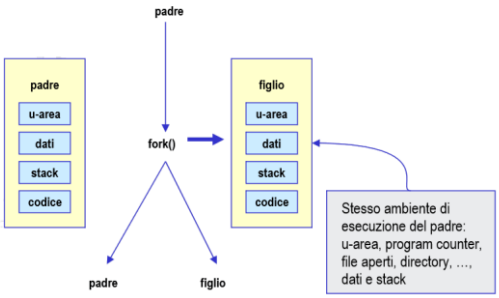
- Esecuzione di un nuovo job batch
- Collegamento di un nuovo utente
- Creazione di un servizio (esempio: stampa)
- Creazione di un processo da parte di un altro processo.

La shell generalmente non fa parte del sistema operativo, ma svolge la funzione di interprete dei comandi. Al momento del login viene avviata una shell, che è considerato un dispositivo di I/O: inizialmente stampa il prompt, poi, all'immissione di un comando, la shell crea un processo figlio che lo esegue.

Il processo padre crea processi figli, che a loro volta possono creare altri processi: a questo punto si possono verificare diversi scenari:

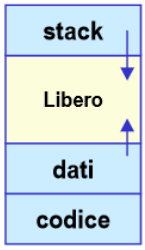
- Condivisione di risorse
    - Padre e figli condividono tutte le risorse
    - I figli condividono un sottoinsieme delle risorse del padre
    - Padre e figli non condividono alcuna risorsa

- Esecuzione
  - Padre e figli sono in esecuzione in modo concorrente
  - Il padre attende che i figli terminino (MS-DOS)
- Spazio degli indirizzi
  - Il figlio è un duplicato del padre
  - Il figlio è un nuovo programma




*Processi UNIX*

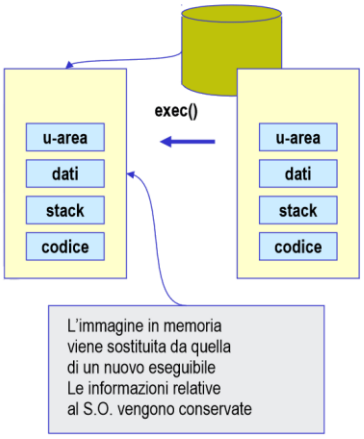
In Unix (standard Posix) un processo figlio viene creato con una chiamata di sistema fork, che duplica l'immagine del padre, creando un processo identico.
Un processo UNIX è costituito da quattro parti principali che costituiscono la sua immagine:




- **u-area**: dati relativi al processo di pertinenza del sistema operativo (tabella dei file, working directory, ...)
- **dati**: dati globali del processo
- **stack**: pila del processo
- **codice**: il codice eseguibile.

La fork restituisce il valore 0 al processo figlio e il pid del figlio al processo genitore (che lo identifica). Il codice non modificabile viene condiviso tra genitore e figlio: se si vuole modificare il codice, uso una exec.

exec(pathname, argomenti)
- sostituisce l'immagine del chiamante con il file eseguibile pathname e lo manda in esecuzione passandogli gli argomenti (NB: non crea un nuovo processo).




Un processo speciale (init) viene avviato al boot e legge un file di configurazione che avvia i terminali; tutti i processi appartengono perciò ad un albero che ha init come radice (per visualizzarlo esiste il comando pstree -p).

*Terminazione di un processo*

Quando il processo ha eseguito la sua ultima istruzione, la sua esecuzione termina (comando exit o anche return per la funzione main) e le risorse del processo sono liberate dal sistema operativo. È anche possibile passare informazioni al processo padre utilizzano il comando wait. Il processo padre può porre termine ad alcuni processi figli (abort, condizioni di errore) nei casi in cui:

- il processo figlio ha richiesto troppe risorse
- il compito affidatogli non è più necessario
- il processo padre termina (un utente si scollega)

Alcuni S.O. non permettono ai figli di sopravvivere al padre e tutti i figli sono fermati con una terminazione a cascata.

*Sostituzione di codice*

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

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

*La famiglia exec*

exec è in realtà una famiglia di primitive:

int execl(const char *path, const char *argv0, …);
int execle(const char *path, const char *argv0, … /*, char * const *envp[] */);
int execlp(const char *path, const char *argv0, …);
int execv(const char *path, char * const *argv[]);

int execve(const char *path, char * const *argv[], char * const *envp[]);
int execvp(const char *path, char * const *argv[]);
int execvpe(const char *path, char * const *argv[], char * const *envp[]);

...l (list: i parametri sono una lista di argomenti della funzione, si aggiunge sempre NULL in coda)
...le (environment)
...lp (path) fa riferimento alla variabile di shell $PATH

...v (vector: i parametri sono memorizzati in un vettore, con un elemento terminale NULL)
...ve (environment)
...vp (path) fa riferimento alla variabile di shell $PATH

- Scheduling dei processi

Lo **Scheduler** è la parte del sistema operativo che decide quale processo eseguire per primo se più processi sono in attesa in un certo, e l'algoritmo che utilizza è chiamato **algoritmo di scheduling**.

Ogni processo è unico ed imprevedibile, quindi per evitare tempi troppo lunghi di esecuzione di un processo molti sistemi operativi prevedono un Timer (o Clock): ad ogni interrupt del clock il sistema operativo decide se il processo può continuare oppure se deve essere sospeso per permettere l'esecuzione di un diverso processo presente in memoria. La strategia che consente di sospendere temporaneamente processi che sono logicamente eseguibili viene detta Preemptive scheduling (scheduling con diritto di prelazione): questo tipo di strategia si contrappone al metodo di esecuzione a completamento (tipico dei sistemi batch) o non preemptive scheduling (senza diritto di prelazione), con cui lo Scheduler deve attendere che il processo termini o che cambi il suo stato da quello di esecuzione a quello di attesa o di pronto (a seguito, ad esempio, di una richiesta di I/O o di un interrupt).

Lo Scheduler della CPU seleziona fra i processi in memoria quelli pronti per l'esecuzione e assegna la CPU a uno di essi. La decisione di scheduling della CPU può avvenire quando un processo:

1. Passa da in esecuzione a in attesa (non-preemptive)
2. Passa da in esecuzione a pronto (preemptive)
3. Passa da in attesa a pronto (preemptive)
4. Termina (non-preemptive)

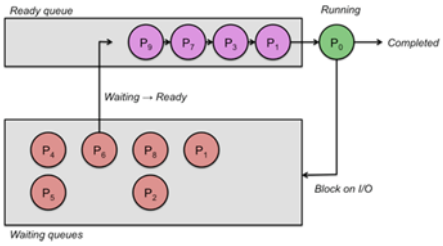
Il Dispatcher è il modulo che dà il controllo della CPU al processo selezionato dallo scheduler e coinvolge:

- Il cambio di contesto
- Il passaggio al modo utente
- Il salto alla giusta posizione del programma utente
- Il tempo necessario è chiamato latenza di dispatch (dovrebbe essere il più piccolo possibile)

Un buon algoritmo di scheduling dovrebbe soddisfare i seguenti criteri:

- Equità: ogni processo deve avere a disposizione una corretta quantità di tempo di CPU
- Efficienza: la CPU dovrebbe essere utilizzata il 100% del tempo
- Tempo di completamento: deve essere il più piccolo possibile
- Tempo di risposta: deve essere minimizzato per gli utenti interattivi
- Tempo di attesa: il tempo trascorso nella coda di attesa deve essere minimizzato
- Throughput: occorre massimizzare il numero di job processati per unità di tempo

Essendo il tempo di CPU finito alcuni di questi obiettivi non sono compatibili fra di loro.




*First Come First Served (FCFS o FIFO)*

L'ordine di esecuzione è pari all'ordine di arrivo: se un processo lungo parte, gli altri processi dovranno attendere tempi più lunghi per poter essere eseguiti.

*Shortest job first (SJF)*

Algoritmo particolarmente indicato per l'esecuzione batch dei job per i quali i tempi di esecuzione sono conosciuti a priori; lo scheduler dovrebbe utilizzare questo algoritmo quando nella coda di input risiedono job di uguale importanza. SJF garantisce sempre il minimo tempo medio di risposta e sarebbe utile estenderlo all'esecuzione dei processi interattivi: se per i processi interattivi si considera l'esecuzione di ogni comando come un singolo job, si ottiene il minimo tempo di risposta eseguendo il più breve per primo.

- Problema: determinare quale tra i processi eseguibili è il più breve
- Soluzione: uso di stime basate sul comportamento passato ed esecuzione del processo con il minor tempo di esecuzione stimato

Esistono due diversi schemi:

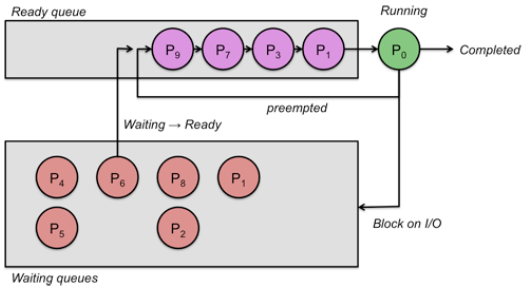
- Non-preemptive: il processo in esecuzione non può essere interrotto
- Preemptive: se arriva un nuovo processo con un CPU burst atteso più breve del tempo rimanente per il processo corrente, il nuovo processo ottiene la CPU.
  Il criterio è noto anche come Shortest-Remaining-Time-First (SRTF)

L'Invecchiamento (o aging) è la tecnica utilizzata per la stima del valore successivo in una serie basata sul calcolo della media pesata del valore corrente misurato e la stima precedente. Ad esempio:

$T_{st} = a\ T_{mis} + (1\text{-}a)\ T_{st}$ con a = 0.5: $T_0$, $T_0/2 + T_1/2$, $T_0/4 + T_1/4 + T_2/2$, $T_0/8 + T_1/8 + T_2/4 + T_3/2$

*Scheduling round robin*

Ad ogni processo viene assegnato un intervallo di tempo di esecuzione prefissato denominato **quanto** (o **time slice**): se al termine di questo intervallo di tempo il processo non ha ancora terminato l'esecuzione, l'uso della CPU viene comunque affidato ad un diverso processo. Anche in questo caso, ogni processo ha uguale priorità di esecuzione.



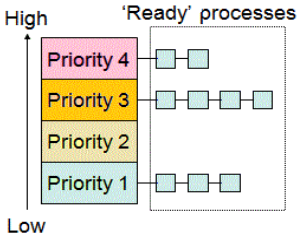

*Context Switch*

Il passaggio dell'esecuzione da un processo ad un altro richiede tempo per il salvataggio ed il caricamento dei registri e delle mappe di memoria, aggiornamento di tabelle e liste: tale operazione viene chiamata **context switch**. La durata del quanto di tempo necessaria:

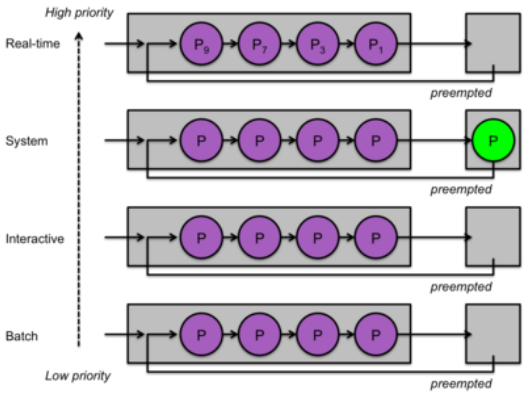
- non deve essere troppo breve per evitare molti context switch tra i processi o la riduzione dell'efficienza della CPU
- non deve essere troppo lunga per evitare tempi di risposta lunghi a processi interattivi con tempo di esecuzione breve.

*Scheduling a priorità*

Ad ogni processo viene assegnata una priorità, in modo tale che la CPU possa eseguire prima il processo con la priorità maggiore. Per evitare che processi con una priorità alta vengano eseguiti indefinitamente, lo scheduler decrementa la priorità del processo in esecuzione ad ogni interrupt del clock; avviene un context switch quando la priorità del processo è minore di quella del processo successivo con priorità più alta.




Le priorità possono essere assegnate staticamente o dinamicamente: è utile raggruppare i processi in **classi di priorità** e utilizzare scheduling a priorità tra le classi e scheduling round robin all'interno di una classe.




*Code multilivello*

La coda dei processi pronti può essere divisa in code separate:

- Processi in foreground (interattivi)
- Processi in background (batch)

Ogni coda può utilizzare propri algoritmi (es: Foreground: round robin, Background: FCFS). Occorre decidere anche uno scheduling fra le code, che possono avere priorità fissa o un time slice stabilito, con ogni coda avente una sua percentuale (es: 80% processi interattivi, 20% batch).
Le code multiple possono essere statiche, in cui ogni processo può appartenere solo a una coda, o dinamiche, dove un processo può spostarsi fra una classe e l'altra e in particolare:

- I processi nella classe più alta vengono eseguiti per un quanto, quelli nella seconda per due quanti, quelli nella successiva per quattro quanti e così via
- Quando un processo ha utilizzato i quanti ad esso assegnati, viene passato alla classe inferiore

Con questo metodo i processi lunghi scendono nelle code di priorità per dare la precedenza all'esecuzione dei processi interattivi brevi.

*Scheduling garantito*

Un approccio di scheduling completamente differente è quello di fare promesse reali all'utente e poi lasciare che si gestiscano.

**Promessa**: *se ci sono n utenti connessi, ogni utente riceverà 1/n della potenza di CPU*

Per mantenere la promessa il sistema deve tenere traccia di:

- quanto tempo di CPU un utente ha utilizzato per i suoi processi dopo la procedura di login e anche quanto tempo è passato dal login
- tempo di CPU che spetta a ogni utente (il tempo passato dal login diviso per n).

La priorità sarà calcolata in base al rapporto tra il tempo di CPU effettivamente utilizzato da un utente e il tempo che gli sarebbe spettato: l'algoritmo consiste nell'eseguire il processo con il rapporto minore finché il suo rapporto raggiunge quello del più vicino competitore.

Questa idea può essere applicata ai sistemi real-time, nei quali ci sono vincoli di tempo stretti da rispettare, perché consente di eseguire prima il processo che rischia maggiormente di non rispettare le scadenze.

*Scheduling ad estrazione*

Ad ogni processo si danno un certo numero di biglietti della lotteria (tempo di CPU), che sostituiscono le priorità: in questo modo il sistema reagisce velocemente ai cambiamenti e consente ai processi di scambiarsi i biglietti.

*Real time scheduling*

Due tipi:

- Hard real-time system:
    - I processi critici terminano entro un tempo stabilito
    - Viene scelto il processo con scadenza più vicina
    - Spesso i processi sono periodici
- Soft real-time system:
    - I processi critici hanno una priorità maggiore degli altri

*Scheduling a due livelli*

Se la memoria principale è insufficiente, alcuni processi eseguibili devono essere mantenuti su disco: in questo caso lo scheduling dei processi comporta situazioni con tempi di switching molto diversi per i

processi che risiedono su disco e i processi in memoria. Un modo pratico per gestire questa situazione è l'utilizzo di uno scheduler a due livelli con due tipi di scheduler:

- **Scheduler di medio termine**
  - Si occupa degli spostamenti dei processi tra memoria e disco: rimuove i processi che sono stati in memoria per un tempo sufficiente e carica in memoria i processi che sono stati su disco a lungo;
- **Scheduler di breve termine**
  - Si occupa dell'esecuzione dei processi che sono effettivamente in memoria

Soprattutto nei sistemi batch si parla di un terzo tipo di scheduler:

- **Scheduler di lungo termine**
  - Sceglie quali lavori (job) mandare in esecuzione

I criteri di decisione dello scheduler di medio termine sono:

- il tempo passato dall'ultimo spostamento da o in memoria
- quanto tempo di CPU è stato assegnato al processo
- la grandezza del processo (non è conveniente spostare processi piccoli)
- priorità del processo

*Scheduling di Windows 2000*: usa un algoritmo basato su priorità e prelazione

- 32 livelli di priorità (una coda per ogni livello)
- una classe a priorità variabile (1-15) e una classe "real time" (16-31)
- la classe 0 è usata solo dal thread di gestione della memoria

Il "dispatcher" sceglie il processo (thread) a priorità più alta pronto per l'esecuzione; se nessuno è pronto viene eseguito lo "idle thread" (un ciclo idle del sistema per ogni processore logico).

Se un processo a priorità variabile esaurisce il suo tempo, gli viene tolta la CPU e la sua priorità viene abbassata.
Se rilascia spontaneamente la CPU la priorità viene innalzata (soprattutto se è in attesa di un evento da tastiera).
Vengono distinti processi in primo piano dai processi in background.
Un processo può essere interrotto da un nuovo processo "Real time" a più alta priorità.

*Scheduling di Linux*: usa tre diversi criteri di scheduling

- FIFO in tempo reale
- Round robin in tempo reale
- Time sharing

I processi della prima classe sono interrotti solo da processi della stessa classe a più alta priorità, mentre i processi della seconda sono eseguiti per un quanto di tempo per essere poi rimessi in coda.

Dalla versione 2.4 i processi normali sono eseguiti in base ai loro crediti.

- Ad ogni interruzione generata dal clock il processo in esecuzione perde un credito
- Se i suoi crediti sono 0 viene sostituito da un nuovo processo pronto
- Se nessun processo pronto dispone di crediti, i crediti di tutti i processi sono aggiornati secondo la formula: crediti = crediti/2 + priorità

Un processo cede la CPU quando un processo a più alta priorità prima bloccato diventa pronto.
La priorità normale in genere è posta a 20 quanti (ticks) cioè circa 210 ms.
Dopo una fork() ogni processo eredita metà dei crediti.

Dalla versione 2.6 si hanno 140 livelli di priorità (0 – 99 per i processi real time, 100 – 139 per i processi normali).

StaticPriority = 100 + nice + 20

$$quanto = \begin{cases} (140 - SP) * 20 & per\ SP \leq 120 \\ (140 - SP) * 5 & per\ SP \geq 120 \end{cases}$$

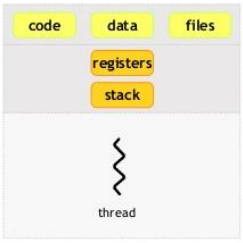
| *Priorità* | Massima | Alta | Normale | Bassa | Minima |
|---|---|---|---|---|---|
| *Nice* | -20 | -10 | 0 | 10 | 19 |

L'algoritmo di scheduling può essere uno di quelli visti precedentemente (round robin, a priorità ecc.).

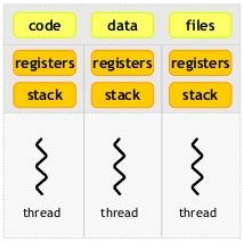
- Thread

Molti sistemi operativi forniscono il supporto per definire sequenze di controllo multiple all'interno di un singolo processo: queste sequenze di controllo sono solitamente chiamate **thread** o processi leggeri.

Vantaggi:
- Tempo di risposta
- Condivisione delle risorse
    - Es: tastiera disponibile a più processi
- Economia
- Uso di più unità di elaborazione



Single Threaded Process



Multi Threaded Process

*Thread a livello utente*

L'utente può gestire i thread utilizzando le opportune librerie fornite dalle API (POSIX, Win32) o dai linguaggi di programmazione (Java).

Vantaggi:
- La commutazione fra i thread non richiede l'intervento del nucleo
- Lo scheduling dei thread è indipendente da quello del nucleo
- Lo scheduling può essere ottimizzato per la specifica applicazione
- Sono indipendenti dal sistema operativo in quanto implementati come libreria

Problemi:
- Le chiamate di sistema sono bloccanti
- Non si sfrutta un eventuale parallelismo hardware

*Thread a livello del nucleo*

I thread a livello del nucleo sono gestiti direttamente dal sistema operativo (Windows, Solaris, Linux, Tru64 UNIX, Mac OS X)

Vantaggi:
- In un sistema multiprocessore il nucleo può assegnare thread dello stesso processo a CPU diverse
- In caso di chiamate di sistema si blocca il singolo thread, non l'intero processo

Problemi:
- La commutazione è costosa: non vale più uno dei vantaggi dell'uso dei thread

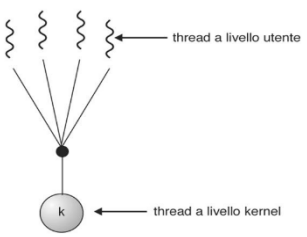
*Problemi di schedulazione*

*Esempio*: Processo A: 1 thread Processo B: 100 thread

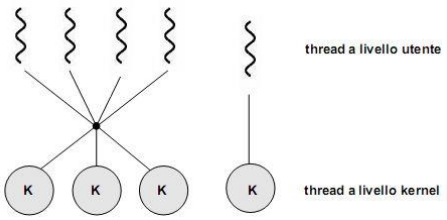
- Thread utente: A e B ottengono lo stesso tempo macchina
    - Ogni thread di B un centesimo dell'unico di A
- Thread di sistema: ogni thread ottiene lo stesso tempo
    - A ottiene un centesimo del tempo di B

*Modelli di programmazione multi-thread*

- Molti a uno
    - Molti thread di livello utente sono mappati in un singolo thread del nucleo
    - Esempi: Solaris Green Thread, GNU Portable Thread
- Uno a uno



    - Ad ogni thread di livello utente corrisponde un singolo thread del nucleo
    - Esempi: Windows NT/XP/2000, Linux, Solaris 9 e successivi
- Molti a molti
    - Consente a molti thread di livello utente di essere mappati in molti thread del nucleo
    - Permette al sistema operativo di creare un numero sufficiente di thread a livello nucleo
    - Esempi: Windows NT/2000 con la libreria ThreadFiber, Solaris 8 e precedenti
- Modello a due livelli
    - È simile al modello molti-a-molti, ma permette ad un thread utente di essere associato ad un thread del nucleo
    - Esempi: IRIX, HP-UX, Tru64 UNIX, Solaris 8 e precedenti




*Problemi di programmazione*

- Semantica di fork() ed exec()
    - Lavorano su tutti i thread o solo su quello chiamante?
- Cancellazione di thread
    - Cosa succede se un thread viene interrotto prima della sua fine naturale?
    - Due diversi approcci:
        - Cancellazione asincrona: il thread interessato viene fermato immediatamente
        - Cancellazione differita: ogni thread controlla periodicamente se deve terminare
- Trattamento di segnali
    - I segnali sono usati in UNIX per comunicare ad un processo il verificarsi di un particolare evento
    - Tutti i segnali seguono lo stesso schema: il segnale è generato da un particolare evento, viene inviato ad un processo e poi viene gestito
    - Chi gestisce il segnale?
        - Il segnale viene inviato al thread a cui si riferisce
        - Ogni thread riceve il segnale
        - Solo alcuni thread ricevono il segnale
        - Esiste un thread particolare che gestisce tutti i segnali
- Gruppi di Thread
    - Si creano un certo numero di thread che attendono di lavorare
    - Vantaggi:
        - È più veloce usare un thread esistente che crearne uno nuovo
        - Limita il numero di thread esistenti
- Dati specifici dei thread
    - Permette ad ogni thread di avere dei propri dati, per esempio associando un identificatore diverso a ciascuno di essi
    - Attenzione quando si usano gruppi di thread

*Windows XP*

- Modello uno a uno
- Ogni thread contiene
    - Un thread ID
    - Un insieme di registri
    - Pile separate per il modo utente ed il modo nucleo
    - Dati privati
- Registri, pile, e memoria sono definiti come il contesto dei thread

*Linux*

- Nella terminologia di Linux si parla di task invece che di thread
- La creazione viene fatta attraverso la chiamata di sistema clone()
- clone() permette ad un task figlio di condividere lo spazio di indirizzi del task genitore (processo)
- La fork diventa un caso particolare della clone

*Java*

I thread di Java sono gestiti dalla JVM e possono essere creati in due modi:

- Estendendo la classe Thread
- Implementando l'interfaccia Runnable

Costruttori:

- Thread(threadName): crea un thread con il nome specificato
- Thread(): crea un thread con un nome di formato predefinito (Thread-1, Thread-2, ...)

Metodi:

- run(): "fa il lavoro effettivo" del thread e deve essere sovrascritto nelle sottoclassi
- start(): lancia l'esecuzione del thread, permettendo quindi il proseguimento dell'esecuzione del chiamante
    - Richiama il metodo run
    - È un errore chiamare due volte start riguardo allo stesso thread

Priorità dei thread:

Ogni applicazione o applet Java è multithread. La priorità dei thread va da 1 a 10:

- Thread.MIN_PRIORITY = 1
- Thread.NORM_PRIORITY = 5 (default)
- Thread.MAX_PRIORITY = 10

Ogni nuovo thread eredita la priorità di chi lo ha creato.

Timeslice: Ogni thread ottiene un quanto del tempo del processore per l'esecuzione; al termine il processore passa al prossimo thread di pari priorità (schedulazione di tipo round-robin)

Metodi per controllare la priorità:

- setPriority(int priorityNumber)
- getPriority()

Metodi per il controllo dello scheduling

- yield() viene rilasciata volontariamente la CPU

- Comunicazione tra processi

L'accesso concorrente a dati condivisi può risultare in inconsistenza dei dati stessi: per poter garantire la consistenza sono necessari meccanismi per coordinare l'esecuzione dei processi cooperanti.

*Problema produttore-consumatore* (bounded buffer problem)
Consideriamo due processi che condividono un buffer comune di dimensione fissata: il processo Produttore mette le informazioni nel buffer, mentre il processo Consumatore le preleva.

Ciò che accade è il verificarsi di una race condition.

Una operazione atomica è una operazione che si completa senza interruzioni. Le istruzioni contatore++ e contatore-- devono essere eseguite in modo atomico, ma in realtà quando vengono tradotte in linguaggio macchina sono costituite da più parti. Se sia il produttore che il consumatore tentano di aggiornare il buffer allo stesso tempo le istruzioni assembler possono risultare intercalate: la sequenza effettiva dipende quindi dalla schedulazione dei due processi.

In queste condizioni il programma potrebbe non terminare mai.

*Comunicazione interprocesso*
In generale una **race condition** è una situazione in cui due o più processi stanno leggendo o scrivendo qualche dato condiviso e il risultato finale dipende da chi esegue e quando. Per prevenire le race conditions i processi concorrenti devono essere sincronizzati.

Il problema della sezione critica si verifica quando N processi competono per usare dati condivisi e ognuno ha un segmento di codice (**sezione critica**) in cui i dati condivisi sono utilizzati: è necessario garantire che quando un processo sta eseguendo la sua sezione critica, nessun altro processo possa entrarvi.

La scelta di appropriate operazioni primitive per ottenere la mutua esclusione è uno dei problemi fondamentali della progettazione di un sistema operativo: a tal proposito sono state formalizzate alcune condizioni da rispettare per avere una buona soluzione:

1. Due processi non devono essere mai simultaneamente in una sezione critica
2. Nessuna assunzione a priori può essere fatta a proposito della velocità o del numero di CPU
3. Nessun processo in esecuzione fuori dalla sua sezione critica può bloccare altri processi
4. Nessun processo deve aspettare a oltranza per entrare nella sua sezione critica

Per ottenere la mutua esclusione sono state proposte varie soluzioni: ognuna può essere distintiva per qualche implementazione di sistema operativo. Mentre un processo è occupato ad aggiornare la memoria condivisa nella sua regione critica, nessun altro processo entrerà nella sua regione critica a "causare guai".

Prima di usare le variabili condivise (regione critica) ciascun processo chiama una *enter_region()* con il proprio numero di processo (0 o 1) (può dover aspettare fino a che risulti sicuro entrare nella regione critica). Dopo aver finito di lavorare con le variabili condivise, il processo chiama una *leave_region()* per permettere a un altro processo di entrare

*Variabili di lock*
Quando un processo vuole entrare nella sua regione critica esegue un test su una singola variabile lock, condivisa da tutti i processi e inizializzata a 0.

- Se lock è 0 (false), il processo la imposta ad 1 ed entra nella sua regione critica
- Se lock è 1 (true), il processo attende finché diventa 0

Se il lock è 1 significa che un processo sta utilizzando la sua regione critica e bisogna quindi attendere che essa venga rilasciata; al contrario se il lock è 0 è possibile accedere immediatamente alla regione critica in quanto nessun processo la sta occupando.

Problema: due processi possono entrare contemporaneamente nella loro regione critica

- Il processo 1 fa il test di lock (è falso)
- Il processo 1 viene interrotto
- Il processo 2 fa il test di lock (è falso)
- Il processo 2 pone lock a vero
  …
- Il processo 1 riprende

*Alternanza stretta*
Il codice implementa la stretta alternanza nell'esecuzione della regione critica da parte di due processi.

**Attesa attiva** (o **busy waiting**): test continuo su una variabile nell'attesa di un certo valore. Dovrebbe essere evitato perché spreca tempo di CPU

Problemi:

- Questa soluzione viola la terza condizione per la mutua esclusione, infatti un processo può essere bloccato da un altro che non sta eseguendo la sua regione critica
- Se un processo si blocca allora anche l'altro è fermo
- La soluzione può essere generalizzata a più processi, rendendo ancora più critico il problema precedente

*Peterson*
Nessun processo è nella sezione critica e il processo 0 chiama la enter_region, indicando il suo interesse mettendo a true il proprio elemento nell'array. Mette turn a 1 e se il processo 1 non è interessato, enter_region termina subito; se invece il processo 1 chiama la enter_region, rimane in attesa fino a che interested[0] è false (cioè quando il processo 0 chiama la leave_region).

Se entrambi i processi chiamano la enter_region modificano il valore in turn (ma uno viene perso): se il processo 1 scrive per ultimo nella variabile turn, quando entrambi i processi arrivano al ciclo while il processo 0 non esegue il ciclo ed entra immediatamente in sezione critica, al contrario il processo 1 esegue il ciclo rimanendo in attesa di entrare nella sezione critica.

*Disabilitazione degli interrupt*
Ogni processo disabilita tutti gli interrupt dopo essere entrato nella sua regione critica e li riabilita prima di lasciarla. Se gli interrupt sono disabilitati nessun interrupt del clock può essere attivato: la disabilitazione delle procedure di interrupt qualche volta è utile se a carico del SO (non può essere lasciata all'utente), ma non è un buon approccio come meccanismo generale per risolvere il problema della mutua esclusione.

*Istruzione TSL* (Test and Set Lock)
Istruzione che copia il contenuto della parola di memoria in un registro e quindi memorizza un valore diverso da zero in quell'indirizzo di memoria; le due operazioni sono indivisibili. In modo equivalente si può usare una istruzione SWAP che scambia il valore fra due memorie in maniera atomica.

*Sleep and Wakeup*
La soluzione di Peterson e l'utilizzo dell'istruzione TLS sono corrette ma comportano l'attesa attiva che comporta uno spreco di tempo di CPU e il problema dell'inversione di priorità (attesa di un processo a bassa priorità). Soluzione: uso di system call.

Le system call sleep e wakeup bloccano i processi invece di sprecare tempo di CPU quando non possono eseguire la loro regione critica.

- Sleep: system call che blocca il chiamante finché un altro processo lo "risveglia"
- Wakeup: system call che ha come unico parametro il processo da "risvegliare"

Se il produttore vuole mettere un nuovo elemento nel buffer pieno, viene messo in attesa per essere risvegliato quando il consumatore ha prelevato qualche elemento; se il consumatore vuole rimuovere un elemento dal buffer vuoto, viene messo in attesa per essere risvegliato quando il produttore ha messo qualcosa nel buffer. Tuttavia, si ha ancora presenza di corse critiche in cui entrambi possono rimanere dormienti a oltranza.

*Semafori*

Dijkstra introduce una nuova proposta per la soluzione dei problemi relativi alla sincronizzazione tra processi e alle condizioni di gara tramite l'utilizzo di un **semaforo**, cioè una variabile intera per il conteggio del numero di wakeup pendenti. Il semaforo avrà valore 0 se non è stato salvato alcun wakeup, mentre avrà un valore positivo se ci sono wakeup pendenti.

down (P) e up (V): generalizzazioni di sleep e wakeup le cui operazioni devono essere atomiche (ossia vengono eseguite come un'azione singola ed indivisibile)

*Monitor*

La realizzazione di programmi con l'utilizzo di semafori può portare facilmente ad errori subdoli quali il **deadlock**, una situazione in cui più processi sono definitivamente bloccati e non può essere eseguito alcun lavoro. Una soluzione che consente la realizzazione di programmi corretti in modo semplice è l'utilizzo di una primitiva di sincronizzazione di più alto livello denominata **monitor**, cioè una collezione di procedure, variabili e strutture dati raggruppate in un pacchetto di tipo speciale. I processi possono richiamare le procedure contenute in un monitor ma non possono accedere direttamente alle sue strutture dati interne da procedure dichiarate esternamente al monitor.

I monitor sono costrutti di linguaggio di programmazione quindi il compilatore gestisce diversamente le chiamate alle procedure del monitor dalle chiamate ad altre procedure

I monitor sono utili per ottenere la mutua esclusione: infatti in ogni istante solo un processo può essere attivo nel monitor. Affinché due processi non entrino contemporaneamente nella loro regione critica è sufficiente inserire le loro sezioni critiche nelle procedure del monitor, poiché la realizzazione della mutua esclusione nei monitor è demandata al compilatore e non al programmatore (minore possibilità di errore).

*Problema*: come bloccare i processi quando non possono procedere?
*Soluzione*: variabili condizione con due operazioni su di esse (wait e signal):

- Wait: operazione su una variabile condizione che blocca il processo chiamante
- Signal: operazione su una variabile condizione che risveglia il relativo processo

Una signal deve essere l'ultima operazione in una procedura di un monitor per evitare che due processi siano attivi contemporaneamente nel monitor (cioè nella sezione critica).

Molti linguaggi (ad esempio C e Pascal) non prevedono i monitor e i semafori e le primitive utilizzate per l'implementazione non sono applicabili in un ambiente distribuito, dove più CPU, ognuna con la relativa memoria privata, sono connesse da una rete locale. Una soluzione può essere il passaggio di messaggi.

Passaggio di messaggi

Questo metodo di comunicazione tra processi utilizza due primitive send e receive, system call che possono essere inserite in procedure di libreria come:

- send(destinazione, &messaggio)
- receive(sorgente, &messaggio)

Con il passaggio di messaggi sono possibili molte varianti per l'indirizzamento dei messaggi: in generale è necessario assegnare ad ogni processo un indirizzo unico ed indirizzare i messaggi ai processi.

Mailbox: struttura dati speciale che consente di bufferizzare un numero di messaggi definito alla creazione della mailbox. I parametri di indirizzamento nelle chiamate send e receive sono le mailbox e non i processi.

I due estremi con l'utilizzo delle mailbox sono rispettivamente:

- produttore e consumatore possono creare mailbox in grado di contenere gli N messaggi
- Rendezvous: nessuna bufferizzazione, il produttore ed il consumatore lavorano forzatamente allo stesso passo.

*Barriera*

I processi si avvicinano ad una barriera e si bloccano per attraversarla solo quando tutti i processi vi arrivano.

*Equivalenze fra primitive*

Sono state proposte diverse primitive per la comunicazione tra processi: utilizzando una qualsiasi di queste primitive è possibile implementare le altre. Si può dimostrare infatti l'equivalenza tra semafori, monitor e messaggi.

*Filosofi a cena* (problema di sincronizzazione - Dijkstra, 1965)

N filosofi stanno seduti intorno a un tavolo. Ciascun filosofo ha un piatto di spaghetti: per mangiarli ogni filosofo ha bisogno di usare due forchette; fra ognuno dei piatti vi è una forchetta. La vita dei filosofi è fatta di periodi alternati in cui essi pensano o mangiano: quando un filosofo comincia ad avere fame, cerca di prendere possesso della forchetta di sinistra e di quella di destra, una alla volta in ordine arbitrario. Quando ha a disposizione entrambe le forchette incomincia a mangiare, quando si è sfamato depone le forchette e riprende a pensare.
Esiste la possibilità di stallo: se tutti i filosofi prendono la forchetta di sinistra nello stesso istante, nessuno di loro sarà in grado di prendere quella di destra.

*Secondo approccio*: Il filosofo prende la prima forchetta e, se non è disponibile la seconda forchetta, il filosofo rimette la prima sul tavolo e aspetta un certo intervallo di tempo; se tutti i filosofi iniziano contemporaneamente con una forchetta, e l'altra non è disponibile, la ripongono, aspettano, riprovano da capo.

**Starvation**: situazione in cui ogni programma è in esecuzione senza che ottenga effettivamente alcun progresso.

*Terza soluzione*: Si fa uso di un semaforo binario:

- prima di prendere possesso delle forchette, un filosofo esegue una down sulla variabile mutex
- dopo aver deposto le forchette, fa una up sulla variabile mutex

La soluzione è corretta, ma costosa in termini di prestazioni: solo un filosofo alla volta può mangiare, mentre con 5 forchette due filosofi potrebbero mangiare contemporaneamente. La soluzione migliore è la seguente: consentire il massimo grado di parallelismo per un numero arbitrario di filosofi impiegando un vettore di stati/semafori per ogni filosofo (un filosofo inizia a mangiare solo se nessuno dei vicini sta mangiando).

*Lettori e scrittori*

Il problema dei "lettori e scrittori" modella l'accesso a un database.

Esempio: un database molto grande con tanti processi in competizione per leggere o scrivere. Molti possono leggere contemporaneamente, ma se un processo sta scrivendo (modifica) nessuno può leggere.

Il primo lettore che ha l'accesso esegue una down sul semaforo db; i lettori successivi incrementano (entrando) e decrementano (uscendo) un contatore L'ultimo a uscire esegue una up sul semaforo lasciando via libera ad uno scrittore Si è data priorità ai lettori, ma ci possono essere soluzioni diverse

*Il barbiere dormiente*

In un negozio di barbiere c'è un unico barbiere, una poltrona da lavoro e n sedie d'attesa per i clienti: se non ci sono clienti, il barbiere si riposa sulla poltrona. Quando arriva un cliente, sveglia il barbiere; se ne arrivano altri, si accomodano sulle sedie d'attesa, fino al loro esaurimento (gli altri vanno).

Problema: come evitare corse critiche? Tre semafori: uno conta i clienti in attesa, uno verifica lo stato di attività del barbiere e un mutex che controlla la mutua esclusione (sulla poltrona) + una variabile che conta i clienti in attesa (controllata da quelli che arrivano).

- Sincronizzazione con Java

Java implementa un meccanismo simile al monitor per garantire la sincronizzazione fra thread. Ogni oggetto ha un lock associato ad esso, e nelle classi possono essere definiti metodi *synchronized*: un solo thread alla volta può eseguire un metodo synchronized e se un oggetto ha più di un metodo sincronizzato, comunque uno solo può essere attivo. NB: non vi è nessun controllo su metodi non sincronizzati.

*Metodi synchronized*

La chiamata di un metodo sincronizzato richiede il "possesso" del lock: se un thread chiamante non possiede il lock (cioè un altro thread ne è già in possesso), viene sospeso in attesa del lock. Il lock è rilasciato quando un thread esce da un metodo sincronizzato.

Un thread può decidere di non proseguire l'esecuzione e chiamare volontariamente il metodo wait all'interno di un metodo sincronizzato (chiamare wait in un contesto diverso genera un errore).

Quando un thread esegue wait(), si verificano i seguenti eventi:

1. Il thread rilascia il lock
2. Il thread viene bloccato
3. Il thread è posto in una coda di attesa
4. Gli altri thread possono ora competere per il lock

Nel momento in cui una condizione che ha provocato una attesa si è modificata si può riattivare un singolo thread in attesa tramite notify. Quando un thread richiama notify():

1. Viene selezionato un thread arbitrario T fra i thread bloccati
2. T torna fra i thread in attesa del lock
3. T diventa eseguibile

*Sincronizzazioni multiple*

notify() seleziona un thread arbitrario, ma non sempre questo è il comportamento desiderato. Java non permette di specificare il thread che si vuole selezionare.

notifyAll() "risveglia" tutti i thread in attesa. In questo modo si permette ai thread stessi di decidere chi deve proseguire. notifyAll() è una strategia conservativa utile quando più thread sono in attesa.

*Metodo sleep*

static void sleep(millisecondi): Il Thread si sospende (non richiede l'uso del processore) per un certo numero di millisecondi, mentre nel frattempo possono andare in esecuzione thread a bassa priorità.
NB: sleep non restituisce il possesso del lock, è un metodo statico della classe Thread. Wait invece è un metodo della classe Object.

*Sincronizzazione di blocco*

Si definisce **scope** di un lock il tempo fra la sua acquisizione e il suo rilascio. Tramite la parola chiave synchronized si possono indicare blocchi di codice piuttosto che un metodo intero: ne risulta uno scope generalmente più piccolo rispetto alla sincronizzazione dell'intero metodo.

*Daemon Threads*
Sono thread che effettuano operazioni ad uso di altri thread (es: garbage collector): sono eseguiti in background, cioè usano il processore solo quando altrimenti il tempo macchina andrebbe perso. A differenza degli altri thread non impediscono la terminazione di una applicazione; quando sono attivi solo thread daemon il programma termina. Occorre specificare che un thread è un daemon prima dell'invocazione di start:

- setDaemon(true);
- isDaemon(): restituisce true se è un daemon thread

*La classe Timer*
La classe java.util.Timer permette la schedulazione di processi nel futuro (esiste anche la classe javax.swing.Timer dal comportamento parzialmente diverso)

Costruttore: void schedule (TimerTask task, long delay, long period)

Permette di eseguire un task periodico dopo un ritardo specificato. Esistono altri costruttori e altri metodi (si veda scheduleAtFixedRate)

## Chiamate di sistema

Una pipe è un file di dimensione limitata gestito come una coda FIFO: un processo produttore deposita dati (e resta in attesa se la pipe è piena), mentre un processo consumatore legge dati (e resta in attesa se la coda è vuota).

Generalmente una pipe viene usata per far comunicare un processo padre con un suo figlio, che ne eredita i file aperti (quindi anche le pipe). La comunicazione generalmente è unidirezionale.

## Deadlock

Esistono potenziali conflitti tra i processi che utilizzano risorse condivise, ragion per cui la maggior parte delle risorse prevede un utilizzo esclusivo: non necessariamente però i deadlock coinvolgono dispositivi di I/O (esempio: database in rete).

I deadlock possono verificarsi quando i processi ottengono accessi esclusivi a dispositivi o oggetti software (in generale si parlerà di **risorse**). Le risorse possono essere di diverso tipo (dispositivi di I/O, memoria, una tupla di un database), ma vengono comunemente classificate in due categorie:

1. Risorse con prerilascio: possono essere tolte ai processi senza problemi (memoria, CPU)
2. Risorse senza prerilascio: se vengono tolte ai processi si ha il fallimento dell'elaborazione (stampanti, lettori di nastro)

In generale i deadlock coinvolgono risorse senza prerilascio.
La sequenza di passi necessari per l'uso di una risorsa è:

1. Richiesta della risorsa
2. Utilizzo della risorsa
3. Rilascio della risorsa

Se la risorsa non è disponibile, il processo viene fatto attendere, in alcuni sistemi il processo viene automaticamente bloccato e poi risvegliato quando la risorsa torna disponibile, in altri è il processo che deve esplicitamente gestire la situazione.

> *Un insieme di processi è in **deadlock** se ogni processo dell'insieme è in attesa di un*

*evento che solo un altro processo appartenente allo stesso insieme può causare.*

Un processo in deadlock non procede mai e non finisce mai la sua esecuzione, inoltre possono bloccare l'intero sistema.

Coffman e altri (1971) hanno dimostrato che le seguenti sono condizioni necessarie affinché esista un deadlock:

1. Mutua esclusione: un solo processo alla volta può utilizzare la risorsa
2. Prendi e aspetta (Hold and Wait): i processi che detengono risorse possono chiederne altre
3. Assenza di prerilascio (No Preemption): le risorse possono essere rilasciate solo dal processo che le detiene, il rilascio non può essere forzato
4. Attesa circolare (Circular Wait): deve esistere una lista circolare di processi/risorse

Holt (1972) ha mostrato che le quattro condizioni possono essere rappresentate da grafi orientati.
I nodi possono essere di due tipi:

- risorse (quadrati)
- processi (cerchi)

Gli archi possono solo connettere nodi di tipo diverso. Necessariamente anche gli archi assumono significato diverso:

- archi uscenti da processi: il processo ha richiesto la risorsa
- archi uscenti da risorse: la risorsa è allocata al processo

*Trattamento del deadlock*
In generale sono utilizzate quattro strategie per trattare i deadlock:

1. Non porsi il problema
2. Individuare il deadlock e risolverlo
3. Prevenire il deadlock in modo dinamico
4. Prevenire il deadlock impedendo una delle quattro condizioni necessarie

*Algoritmo dello struzzo*
Se il deadlock è molto raro, è inutile preoccuparsene: meglio ignorare il problema.

*Prevenzione dei deadlock*
La seconda strategia consiste nell'impedire il deadlock: se si evita una delle quattro condizioni necessarie il deadlock diventa impossibile.

- Mutua esclusione

Se non vi sono risorse assegnate in modo esclusivo ad un singolo processo non vi saranno deadlock. Un possibile approccio è lo **spooling**:

- Occorre un processo daemon (demone) e una directory di spooling
- Per la stampa di un file, il processo genera l'intero file e lo memorizza nella directory di spooling
- Il daemon è l'unico processo ad avere accesso alla directory e ha il compito di avviare la stampa
- Lo spooling viene usato anche nei trasferimenti via rete (posta elettronica, ftp, ...)

*Esempio stampante*: Effettuando lo spooling sulle uscite per la stampante, più processi possono produrre le loro uscite contemporaneamente. Dato che esiste un solo processo che utilizza la stampante fisica e non ha bisogno di altre risorse il deadlock è impossibile, ma non tutti i processi possono essere gestiti attraverso lo spooling (es. la tabella dei processi): rimane comunque la possibilità di deadlock sulla risorsa disco.

- Hold and wait

Occorre evitare che processi che detengono risorse rimangano in attesa di ulteriori risorse. Ci possono essere varie soluzioni:

- Un processo richiede immediatamente tutte le risorse di cui ha bisogno, se non sono disponibili attende. Nascono nuovi problemi:
    - Un processo non sempre sa quello di cui ha bisogno nel corso dell'esecuzione
    - L'uso delle risorse non è ottimizzato (rimangono risorse spesso inutilizzate)
    - Vi è il rischio che un processo che ha bisogno di molte risorse rimanga in attesa per tempi molto lunghi
- Un modo leggermente diverso è quello di imporre che un processo rilasci temporaneamente le risorse detenute prima di una nuova richiesta La terza condizione nessun prerilascio è di fatto inutilizzabile

o Attesa circolare

L'attesa circolare può essere eliminata in vari modi:

- Un processo può richiedere una sola risorsa per volta
- Le risorse sono numerate e possono essere richieste solo secondo l'ordine numerico
- Una variante richiede semplicemente che la nuova risorsa debba avere una etichetta maggiore di tutte quelle detenute

*Evitare il deadlock*

È possibile impedire il verificarsi di un deadlock? Sì se si hanno a disposizione alcune informazioni.
Il modello più semplice e più utile richiede che ciascun processo dichiari il numero massimo di risorse necessarie:

- L'algoritmo esamina dinamicamente lo stato di allocazione delle risorse per garantire che non accada mai una attesa circolare
- Lo stato è definito dal numero di risorse disponibili e allocate e dal numero massimo di risorse richieste

Uno stato si dice *sicuro* se esiste una sequenza di altri stati che porta tutti i processi ad ottenere le risorse necessarie (e quindi terminare) altrimenti viene detto *non sicuro*: per evitare i deadlock è quindi sufficiente evitare gli stati non sicuri.

*Algoritmo del banchiere*

Tratta risorse multiple: ogni processo deve dichiarare il numero massimo di risorse necessarie. Se un processo richiede una risorsa e non è disponibile, rimane in attesa; dopo che un processo ha ottenuto tutte le risorse necessarie, le deve rilasciare in un tempo finito.

L'algoritmo è stato proposto da Dijkstra (1965) imita il comportamento di un banchiere nei confronti delle richieste dei clienti. Uno stato si dice sicuro se esiste una sequenza di altri stati che porta tutti i clienti ad ottenere prestiti fino al loro massimo credito (e quindi terminare).

Una richiesta viene evasa solo se porta in uno stato ancora sicuro, altrimenti il processo deve attendere. L'algoritmo descritto è applicabile ad un sistema con una singola risorsa multipla, ma può essere generalizzato al caso di un sistema complesso con molte classi di risorse. L'algoritmo del banchiere presuppone comunque una conoscenza completa del sistema, ma è un punto di partenza per risolvere i casi concreti.

Per rilevare i deadlock è necessario sviluppare un opportuno algoritmo di rilevamento e definire uno schema di recovery.

*Individuare lo stallo e risolverlo*

Periodicamente si controlla il grafo delle risorse per verificare l'esistenza di un ciclo: se esiste si elimina (almeno) uno dei processi coinvolti.

Un metodo più semplice non controlla nemmeno il grafo: se un processo è bloccato da troppo tempo (es. un'ora) viene rimosso. Esiste comunque il pericolo di effetti collaterali (non sempre è possibile annullare tutti gli effetti di un processo non concluso positivamente).

*Rilevamento a singola istanza*
Se tutti i tipi di risorse possono avere una sola istanza, allora un ciclo nel grafo delle allocazioni indica un deadlock: un generico algoritmo di rilevamento richiede una complessità dell'ordine di $n^2$ operazioni dove n è il numero di processi del grafo di attesa.

*Rilevamento a istanze multiple*
Se esistono più istanze per ogni tipo di risorsa, si usa un algoritmo simile a quello del banchiere per rilevare il deadlock che richiede una complessità dell'ordine di m * n2 operazioni (m = numero di tipi di risorse, n = numero di processi); la frequenza con cui si attiva l'algoritmo di rilevamento dipende dalla probabilità che si verifichi un deadlock e dal numero di processi coinvolti.

Una volta rilevato il deadlock si puó provare a risolverlo utilizzando la preemption, un rollback oppure terminando dei processi coinvolti.

*Terminazione di processi*
Si può procedere in diversi modi: uccidere i processi coinvolti oppure uccidere solo un processo alla volta fino a che il ciclo di deadlock non si è aperto.

Quale ordine scegliere?
- Priorità del processo
- Tempo di esecuzione trascorso e/o necessario
- Risorse usate/necessarie
- Numero di processi che devono essere terminati
- Processi interattivi o batch

*Starvation*
Un fenomeno simile al deadlock è la **starvation**: un processo non può procedere perché altri processi detengono sempre una risorsa di cui ha bisogno, quindi la richiesta del processo non viene mai soddisfatta.

In linea di principio, il processo potrebbe ottenere la risorsa, ma non l'ottiene perché:
- è a priorità bassa
- la temporizzazione della richiesta della risorsa è errata
- l'algoritmo per l'allocazione delle risorse è costruito male

Due soluzioni possono essere:
- opportunità: ogni processo ha una percentuale adeguata di tutte le risorse
- tempo: la priorità del processo viene aumentata se il processo aspetta per un tempo troppo lungo

## Gestione della memoria

La memoria è una risorsa importante e deve essere gestita attentamente.

Un programmatore vorrebbe una memoria infinita, veloce, non volatile e poco costosa: questi desideri però possono essere solo parzialmente soddisfatti (in quanto sono contraddittori). Si cerca di raggiungere un compromesso sfruttando la **gerarchia di memoria**.

La parte del SO che gestisce la memoria è il *gestore di memoria* i cui compiti sono:
- tenere traccia di quali parti di memoria sono in uso e quali non lo sono
- allocare la memoria ai processi che la necessitano e deallocarla

- gestire lo swapping tra la memoria principale e il disco quando la memoria principale non è sufficientemente grande per mantenere tutti i processi
- in definitiva cercare di sfruttare al meglio la gerarchia di memoria.

Il problema fondamentale da risolvere è il passaggio da programma eseguibile (su memoria di massa) a processo in esecuzione (in memoria di lavoro).

*Address binding*
L'associazione tra istruzioni del programma e indirizzi di memoria può essere stabilita in momenti diversi:

- Al momento della compilazione (Indirizzamento assoluto), se ad un dato momento l'indirizzo del programma deve cambiare occorre ricompilare il programma.
  Uno schema di questo tipo è utilizzato nei programmi .com del DOS
- Al momento del caricamento, se è il loader che fa le traduzioni opportune durante l'esecuzione: il codice generato dal compilatore viene detto *rilocabile*, poiché il programma può essere spostato durante l'esecuzione stessa.

Non sempre è necessario caricare in memoria l'intero pprogramm: con il *Dynamic Loading* una porzione di codice (una funzione C per esempio) è caricata solo se viene effettivamente eseguita.

*Librerie dinamiche*
Alcuni sistemi operativi permettono l'uso delle librerie dinamiche (shared library, file .dll in Windows, .so in Unix): in questo caso le librerie sono linkate solo al momento dell'esecuzione del programma.

Questa tecnica permette numerosi vantaggi:
- I programmi sono molto più piccoli
- L'aggiornamento delle librerie è più semplice (basta sostituirle senza dover ricompilare i programmi)
- Vi può essere la possibilità che se una procedura è utilizzata da più di un processo se ne possa conservare una sola copia effettiva in memoria

*Monoprogrammazione*
Lo schema più semplice di gestione della memoria è quello di avere un solo processo alla volta in memoria e consentire al processo di utilizzarla tutta. Questo schema non è utilizzato (nemmeno sui computer meno costosi) in quanto implica che ogni processo contenga i driver di periferica per ogni dispositivo di I/O che utilizza.

La memoria viene divisa tra il SO e un singolo processo utente. Il SO può essere:

- nella RAM (Random Access Memory) in memoria bassa
- nella ROM (Read Only Memory) in memoria alta
- i driver di periferica nella ROM e il resto del SO nella RAM

Quando il sistema è organizzato in questo modo solo un processo alla volta può essere in esecuzione.

*Multiprogrammazione*
Vantaggi della multiprogrammazione
- rende più semplice programmare un'applicazione dividendola in due o più processi
- fornisce un servizio interattivo a più utenti contemporaneamente
- evita spreco di tempo di CPU
    - dato che la maggior parte dei processi passa gran parte del tempo aspettando che vengano completate azioni di I/O del disco, in un sistema monoprogrammato durante questi intervalli di tempo la CPU non lavorerebbe.

Ipotesi base:

- Supponendo che il processo medio sia in esecuzione per il 20% del tempo che risiede in memoria, con 5 processi in memoria, la CPU dovrebbe essere sempre occupata (assumendo ottimisticamente che i 5 processi non siano mai in attesa di I/O contemporaneamente)

Come organizzare la memoria per poter gestire la multiprogrammazione?
La soluzione più semplice è la **multiprogrammazione con partizioni fisse**, con cui la memoria viene divisa in n partizioni (di grandezza eventualmente diversa).

*Code separate*
Soluzione a partizioni fisse e code separate: quando arriva un job viene messo nella coda di input della più piccola partizione in grado di contenerlo.

Le partizioni sono fisse e quindi solo parzialmente occupate: lo spazio non utilizzato dal processo è perso. Uno svantaggio delle code di input separate si ha quando la coda di input per una partizione piccola è piena, mentre quella per una partizione grande è vuota: in questo caso lo spreco di risorse è elevato.

*Coda unica*
Soluzione a partizioni fisse e coda di input singola: quando si libera una partizione, il job più prossimo all'uscita della coda e con dimensione inferiore alla partizione viene caricato e quindi eseguito.

In queste condizioni si può avere spreco di partizioni grandi per job piccoli: una soluzione può essere cercare in tutta la coda il più grande job che può essere contenuto dalla partizione. Tuttavia, ciò genera discriminazione per i job piccoli perché sprecano spazio (assumendo i job piccoli come interattivi, essi dovrebbero avere il miglior servizio)

1. Soluzione: avere almeno una partizione piccola per consentire l'esecuzione dei job piccoli
2. Soluzione: una regola che consenta di stabilire che un job in attesa di esecuzione non venga ignorato più di k volte. Ogni volta che viene ignorato acquisisce un punto; quando ha k punti non può essere ulteriormente ignorato.

L'algoritmo di scheduling risulta complesso.

*Rilocazione*
Quando un programma viene linkato, il linker deve conoscere l'indirizzo di memoria corrispondente all'inizio del programma; gli indirizzi relativi vanno dunque trasformati in indirizzi assoluti.

Una possibile soluzione è la rilocazione durante il caricamento: occorre definire una lista o bit map che specifichi quali elementi vanno rilocati e quali no. Rimane però il problema della protezione: i programmi, definendo indirizzi assoluti, possono costruire un'istruzione che legge o scrive qualsiasi parola di memoria, il che in sistemi multiutente non è accettabile.

Soluzione adottata dall'IBM per proteggere il 360:

- divisione della memoria in blocchi di 2 Kbyte e assegnamento a ogni blocco di un codice di protezione di 4 bit
- assegnamento al PSW (Program Status Word) di una chiave di 4 bit
- il processo in esecuzione ha la possibilità di accesso alla memoria solo se il codice di protezione corrisponde alla chiave nel PSW
- solo il SO può cambiare i codici di protezione e la chiave
- i processi utente non interferiscono tra loro e con il SO stesso.

Soluzione con registri base e limite:

Quando un processo viene selezionato dallo scheduler per essere eseguito nel **registro base** viene caricato l'indirizzo di inizio della sua partizione, mentre nel **registro limite** viene caricata la lunghezza della partizione. Ad ogni accesso alla memoria:

1. a ogni indirizzo generato viene automaticamente sommato il contenuto del registro base prima di essere effettivamente inviato alla memoria
2. viene eseguito un controllo degli indirizzi anche sul registro limite affinché non vi siano tentativi di accesso alla memoria fuori dalla partizione corrente

I registri base e limite non sono modificabili dall'utente.

*Partizioni variabili*

Nei sistemi multiutente normalmente la memoria è insufficiente per tutti i processi degli utenti attivi: è necessario trasferire l'immagine dei processi in eccesso su disco. Lo spostamento dei processi da memoria a disco e viceversa viene detto **swapping**. Il problema con le partizioni fisse è lo spreco di memoria per programmi più piccoli delle partizioni che li contengono: è meglio utilizzare quindi **partizioni variabili**.

Con le partizioni variabili il numero, la locazione e la dimensione delle partizioni varia dinamicamente mentre con le partizioni fisse questi parametri sono stabiliti a priori.
La flessibilità delle partizioni variabili migliora l'utilizzo della memoria; con le partizioni fisse ci sono problemi con partizioni troppo piccole o troppo grandi.
Le partizioni variabili però complicano, rispetto alle partizioni fisse, la gestione delle operazioni di allocazione e deallocazione della memoria.

Quando si formano troppi buchi in memoria è possibile combinare tutti gli spazi liberi in memoria in un unico grande spazio muovendo tutti i processi in memoria verso il basso: questa tecnica viene chiamata compattazione della memoria, ma generalmente non viene utilizzata perché richiede molto tempo di CPU.

*Allocazione di memoria*

Quanta memoria dovrebbe essere allocata per un processo quando viene creato o viene portato in memoria tramite swapping? Per processi a dimensione fissa l'allocazione della memoria al processo è semplice, in quanto viene assegnata al processo esattamente la memoria che necessita. Se invece si sa che i processi tendono a crescere conviene lasciare spazio a disposizione del processo.

Quando un processo cerca di crescere:

- se il processo è adiacente ad uno spazio libero, questo può essere allocato
- se il processo è adiacente a un altro processo:
  1. può essere spostato in uno spazio di memoria libero sufficientemente grande da contenerlo
  2. uno o più processi dovranno essere trasferiti su disco per creare uno spazio libero abbastanza grande da contenerlo
- se il processo non può crescere in memoria e l'area di swapping su disco è piena, il processo deve aspettare o essere "ucciso"

- Metodi di gestione della memoria

*Gestione con bitmap*

Con questa tecnica la memoria è suddivisa in unità di allocazione: a ognuna corrisponde un bit nella bitmap. La scelta della dimensione dell'unità di memoria è importante perché si riflette sulle dimensioni della bitmap e sull'ottimizzazione dell'occupazione di spazio.

Vantaggio: metodo semplice per tenere traccia delle parole di memoria in una quantità fissata di memoria
Svantaggio: per poter eseguire un processo di k unità, il gestore di memoria deve ricercare k zeri consecutivi nella bitmap. Questa è una operazione lenta per cui le bitmap sono poco utilizzate

*Gestione con liste*
Tecnica con la quale si tiene traccia dei segmenti di memoria allocati e liberi. Per segmento si intende una zona di memoria assegnata ad un processo oppure una zona libera tra due assegnate.

I processi e le zone libere sono tenuti in una lista ordinata per indirizzo in questo modo le operazioni di aggiornamento risultano semplificate. La liberazione di una zona di memoria si risolve in quattro casi possibili: (vedi slides)

Mediamente vi è un numero di segmenti liberi pari a metà del numero di processi; non vi è simmetria tra segmenti liberi e occupati in quanto non vi possono essere segmenti liberi consecutivi. Per semplificare le operazioni può essere conveniente usare una lista a doppia concatenazione.

- Algoritmi di allocazione

*First fit*
Il gestore di memoria scandisce la lista dei segmenti finché trova la prima zona libera abbastanza grande: la zona viene divisa in due parti, una per il processo e una per la memoria non utilizzata. È un algoritmo veloce perché limita al massimo le operazioni di ricerca

*Next fit*
Come il precedente, ma ogni ricerca inizia dal punto lasciato alla ricerca precedente: si può notare però da alcune simulazioni che le prestazioni sono leggermente peggiori rispetto al first fit.

*Best fit*
Il gestore di memoria scandisce tutta la lista dei segmenti e sceglie la più piccola zona libera sufficientemente grande da contenere il processo. Questo algoritmo tuttavia è più lento di first fit e spreca più memoria di first fit e next fit perché lascia zone di memoria troppo piccole per essere utilizzate

*Worst fit*
Per risolvere il problema precedente sceglie la più grande zona libera, ma non presenta comunque buone prestazioni.

Questi 4 algoritmi possono essere velocizzati tenendo separate le liste per i processi e gli spazi liberi: la gestione di due liste separate comporta rallentamento e maggior complessità quando vengono deallocati spazi di memoria, ma ordinando la lista degli spazi vuoti in ordine crescente di dimensione si ottimizza la ricerca per il Best Fit.

Un quinto algoritmo è il Quick Fit che mantiene liste separate per le zone di memoria con dimensioni maggiormente richieste: è molto veloce nella ricerca di uno spazio libero di dimensioni fissate, ma come tutti gli altri è costoso in termini di tempo per l'aggiornamento delle liste successivo alla deallocazione di memoria (tempo di ricerca per verificare possibili merge tra spazi liberi adiacenti).

- Memoria virtuale

Principio base: la dimensione totale di programma, dati e stack può eccedere la quantità di memoria fisica disponibile (per il processo). Il Sistema Operativo mantiene in memoria principale solo le parti del programma in uso e mantiene il resto su disco.

Si citano due approcci diversi: **paginazione** e **segmentazione**.

- Paginazione

In qualsiasi computer, ogni programma può produrre un insieme di indirizzi di memoria: questi indirizzi generati dal programma sono detti **indirizzi virtuali** e formano lo **spazio di indirizzamento virtuale**. Nei computer senza memoria virtuale, l'indirizzo virtuale viene messo direttamente sul bus di memoria, quindi

la parola di memoria fisica con lo stesso indirizzo viene letta o scritta. Quando viene utilizzata la memoria virtuale gli indirizzi virtuali non vengono messi direttamente sul bus di memoria ma vengono mandati alla Memory Management Unit (MMU), un chip che mappa gli indirizzi virtuali sugli indirizzi della memoria fisica.

Lo spazio di indirizzamento virtuale è diviso in *pagine*, mentre le unità nella memoria fisica corrispondenti alle pagine sono detti *frame di pagina.* Pagine e frame hanno sempre la stessa dimensione, normalmente da 512 byte a 16M, i trasferimenti fra la memoria ed il disco avvengono sempre per unità di una pagina.

Alcune pagine virtuali non possono avere corrispondenza in memoria fisica: negli attuali circuiti HW un bit presente/assente è utilizzato per tenere traccia se la pagina mappata è sulla memoria fisica oppure no. Nel caso in cui si tenti di accedere a un'area di memoria non mappata, la MMU causa un'eccezione della CPU al SO detta **page fault**. Quando questo accade, il SO sceglie un frame di pagina poco utilizzato, ne salva il contenuto su disco (se necessario), recupera la pagina referenziata e la alloca nel frame appena liberato (fetching della pagina); la mappa viene poi aggiornata e l'esecuzione riparte con l'istruzione bloccata.

*MMU*

Se per esempio l'indirizzo virtuale di 16 bit, viene diviso in un numero di pagina di 4 bit e un offset di 12 bit. Il numero di pagina viene utilizzato come indice nella tabella delle pagine così da ottenere l'indirizzo fisico. Se il bit presente/assente è 0 viene causata una eccezione, mentre se il bit presente/assente è 1 il numero del frame trovato nella tabella delle pagine viene copiato nei 3 bit di ordine alto del registro di output con i 12 bit dell'offset (copiati senza modifiche dall'indirizzo virtuale). Il contenuto del registro di output viene messo sul bus di memoria come indirizzo di memoria fisica.

Lo scopo della tabella delle pagine è mappare le pagine virtuali sui frame. Il modello descritto è semplice, vi sono però dei problemi: la tabella delle pagine può essere infatti molto grande e il mapping (da virtuale a fisico) deve essere veloce.

*Tabella delle pagine*

Il modello più semplice consiste di una singola tabella delle pagine costituita da un array di registri hardware, con un elemento per ogni pagina virtuale indicizzato dal numero di pagina virtuale. Quando viene iniziato un processo, il SO carica la sua tabella delle pagine nei registri.

- Vantaggi: il mapping è immediato, nessun riferimento in memoria
- Svantaggi: il mapping è potenzialmente costoso (se la tabella delle pagine è grande) e il caricamento della tabella delle pagine nei registri ad ogni context switch può alterare le prestazioni (deve esistere una tabella diversa per ogni processo)

Il segreto del metodo a tabelle delle pagine multilivello è evitare di tenere tutte le tabelle delle pagine in memoria per tutto il tempo. Struttura di ogni elemento della tabella:

- Il campo più importante è il numero del frame di pagina
- Il bit presente/assente consente di verificare se la pagina virtuale corrispondente all'elemento è in memoria oppure no
- I bit di protezione contengono le informazioni su quali tipi di accesso sono consentiti (lettura, scrittura ed eventualmente esecuzione di una pagina)
- I bit di pagina modificata e referenziata tengono traccia dell'utilizzo della pagina
- Il bit di caching disabilitato è importante per le pagine che mappano su registri di periferiche invece che in memoria

*Memoria associativa*

Le tabelle delle pagine vengono tenute in memoria per le loro grandi dimensioni, ma ciò può penalizzare fortemente le prestazioni (una singola istruzione può fare riferimento a più indirizzi). I programmi tendono

a fare la maggior parte dei riferimenti a un piccolo numero di pagine, quindi solo una piccola parte degli elementi nella tabella delle pagine vengono letti frequentemente: per questo motivo una possibile soluzione è la **memoria associativa** o TLB (Translation Lookaside Buffer), un dispositivo hardware per mappare gli indirizzi virtuali su indirizzi fisici senza utilizzare la tabella delle pagine.

Ogni elemento nella memoria associativa contiene informazioni su una pagina, in particolare il numero di pagina virtuale, un bit di modifica della pagina e il codice di protezione (permessi di lettura, scrittura, esecuzione della pagina), il numero del frame fisico in cui la pagina è situata e un bit che indica se l'elemento è valido o meno. Questi campi hanno una corrispondenza uno a uno con quelli nella tabella delle pagine.

*Tabelle delle pagine invertite*
Le tabelle descritte richiedono un elemento per ogni pagina virtuale: con uno spazio di indirizzamento di 232 e pagine di 4K sono necessari 220 elementi (almeno 4M di dati, per ogni processo), con 64 bit di indirizzamento la tabella raggiunge la dimensione di milioni di Giga.

La soluzione è la tabella delle pagine inversa: in questo caso la tabella contiene un elemento solo per ogni pagina effettivamente in memoria e ogni elemento contiene la coppia (processo, pagina virtuale). Il problema è la traduzione degli indirizzi virtuali, per cui occorre infatti consultare l'intera tabella e non un singolo elemento: la soluzione è data dall'uso della memoria associativa, ed eventualmente sfruttare metodi di codifica hash nel caso in cui la pagina cercata non sia nella memoria associativa.

- Algoritmi di rimpiazzamento

Per ogni page fault il SO deve scegliere una pagina da rimuovere dalla memoria per creare spazio per la nuova pagina: se la pagina da rimuovere è stata modificata, la copia su disco deve essere aggiornata, se non c'è stata alcuna modifica la nuova pagina sovrascrive quella da rimuovere. Scegliendo la pagina da rimpiazzare con un algoritmo per sostituire pagine poco utilizzate (e non a caso) può portare a un miglioramento delle prestazioni.

*Algoritmo ottimo*
Ad ogni pagina viene assegnata un'etichetta corrispondente al numero di istruzioni che dovranno essere eseguite prima che la pagina venga referenziata: l'algoritmo ottimo consiste nel rimuovere la pagina con l'etichetta maggiore. Tuttavia, non è realizzabile in quanto non è possibile conoscere a priori (quindi al momento del page fault) quali pagine verranno referenziate (non si può prevedere).

Potrebbe essere utile eseguire il programma con un simulatore e tenere traccia dei riferimenti alle pagine e usare l'algoritmo ottimo per una seconda esecuzione utilizzando le informazioni sui riferimenti alle pagine della prima esecuzione. È utile come confronto per la valutazione degli algoritmi di rimpiazzamento pagine.

*Algoritmo FIFO*
Il SO mantiene una lista concatenata di tutte le pagine in memoria con la pagina più vecchia in testa e la pagina più recente in coda. Quando avviene un page fault viene rimossa la pagina in testa alla lista e la nuova pagina viene aggiunta in coda alla lista. L'algoritmo FIFO in questa forma viene utilizzato raramente, in quanto non è detto che la pagina più vecchia sia effettivamente la meno referenziata.

*Algoritmo NRU*
La maggior parte dei computer con memoria virtuale prevede 2 bit per la raccolta di informazioni sull'utilizzo delle pagine:

- il bit R: indica che la pagina è stata referenziata (letta o scritta)
- il bit M: indica che la pagina è stata modificata (scritta)

Con l'algoritmo NRU (Not Recently Used) inizialmente i bit R e M vengono impostati a 0 dal SO: periodicamente (ad esempio ad ogni interrupt del clock), il bit R viene azzerato per distinguere le pagine non referenziate recentemente dalle altre.

Quando avviene un page fault, il SO controlla tutte le pagine e le divide in quattro classi in base al valore corrente dei bit R e M

0. non referenziate, non modificate
1. non referenziate, modificate
2. referenziate, non modificate
3. referenziate, modificate

L'algoritmo NRU rimuove una pagina qualsiasi (a caso) dalla classe di numero inferiore che sia non vuota. Inoltre, è facile da implementare e ha prestazioni che, anche se non ottimali, sono spesso adeguate

*Algoritmo Second Chance*

Con una semplice modifica dell'algoritmo FIFO è possibile evitare il problema della rimozione di pagine molto utilizzate. Viene controllato il bit R della pagina più vecchia: se vale 0, la pagina è sia vecchia che non utilizzata e viene rimpiazzata immediatamente; se vale 1, il bit viene azzerato e la pagina viene messa in coda alla lista come se fosse appena stata caricata in memoria.

L'algoritmo si basa sulla ricerca di una pagina che non sia stata referenziata nel precedente intervallo di clock; se tutte le pagine sono state referenziate degenera nell'algoritmo FIFO puro.

*Algoritmo clock*

L'algoritmo second chance è inefficiente perché sposta costantemente le pagine sulla lista. Un approccio migliore consiste nel tenere tutte le pagine su una lista circolare nella forma di un orologio con una lancetta che punta alla pagina più vecchia.

Quando avviene un page fault:

- se il bit R della pagina vale 0, la pagina puntata dalla lancetta viene rimossa, la nuova pagina viene inserita nell'orologio al suo posto e la lancetta viene spostata in avanti di una posizione
- se il bit R della pagina vale 1, viene azzerato e la lancetta viene spostata in avanti di una posizione

Questo processo viene ripetuto finché non viene trovata una pagina con R = 0. Differisce dall'algoritmo second chance solo per l'implementazione.

*Algoritmo LRU (Least Recently Used)*

È una buona approssimazione all'algoritmo ottimo e si basa sulla seguente osservazione: le pagine molto utilizzate nelle ultime istruzioni saranno probabilmente molto utilizzate nelle successive istruzioni, così come le pagine poco utilizzate recentemente continueranno a non essere utilizzate per molto tempo.

Quando avviene un page fault l'algoritmo rimpiazza la pagina non utilizzata da più tempo. È un algoritmo costoso poiché deve essere mantenuta una lista concatenata di tutte le pagine in memoria (ordinata in base al tempo trascorso dall'ultimo utilizzo), questa lista deve essere aggiornata ad ogni riferimento in memoria. Eseguire, ad ogni istruzione, operazioni di ricerca e manipolazione in una lista concatenata è molto costoso in termini di tempo: ci sono però soluzioni per l'implementazione di LRU con un hardware speciale.

*Algoritmo NFU (Not Frequently Used)*

Una approssimazione dell'algoritmo LRU è l'algoritmo NFU: a ogni pagina viene associato un contatore inizialmente posto a 0 e a ogni clock viene sommato al contatore il bit R. Al momento di un page fault viene rimpiazzata la pagina con contatore minimo.

Il problema è che NFU non dimentica nulla: se una pagina è stata molto utilizzata non verrà più rimossa anche se non più utile. È stata proposta una versione modificata con invecchiamento (aging), che ad ogni clock il contatore scorre a destra introducendo R come bit più significativo.

*Modello a Working Set*

Nel modello più semplice di paginazione le pagine vengono caricate in memoria solo al momento del page fault, si parla di paginazione su richiesta (on demand). La maggior parte dei processi esibiscono la località di riferimenti cioè durante qualsiasi fase dell'esecuzione il processo fa riferimenti ad un piccolo insieme di pagine. L'insieme di pagine che un processo sta correntemente utilizzando viene detto working set.

Più è alto il numero pagine in memoria più è probabile che il working set sia completamente in memoria (e quindi non avverranno page fault). Un programma che causa page fault per ogni piccolo insieme di istruzioni viene detto in thrashing.

Il WS con parametro $\delta$ al tempo t, W($\delta$,t) è l'insieme delle pagine a cui ci si è riferiti nelle ultime $\delta$ unità di tempo. Proprietà: $W(\delta,t) \subseteq W(\delta+1,t)$ $1 \leq |W(\delta,t)| \leq \min(\delta,N)$, con N numero di pagine necessarie al processo.

*Località*

Per località si indica che il prossimo accesso di memoria sarà nelle vicinanze di quello corrente.

- Località spaziale: il prossimo indirizzo differirà di poco
- Località temporale: la stessa cella (o pagina) sarà molto probabilmente riutilizzata nel futuro prossimo

La località è la ragione che giustifica un basso numero di page fault (altrimenti non spiegabile)

Strategia di fetch

Stabilisce quando una pagina deve essere trasferita in memoria. La paginazione su richiesta (demand paging) porta una pagina in memoria solo quando si ha un riferimento a quella: perciò si hanno molti page fault quando un processo parte. Il Prepaging porta in memoria più pagine del necessario ed è quindi più efficiente se le pagine su disco sono contigue.

*Prepaging*

Spesso il thrashing è causato da un numero eccessivo di processi in memoria.

Per risolvere il problema del thrashing molti sistemi a paginazione utilizzano il prepaging prima di eseguire un processo: vengono caricate in memoria le pagine del working set relative al processo. Per implementare il modello Working Set è necessario che il Sistema Operativo tenga traccia di quali pagine sono nel working set: un modo per controllare questa informazione è quello di utilizzare l'algoritmo di invecchiamento. Le informazioni sul working set possono essere utilizzate per migliorare le prestazioni dell'algoritmo clock (questo nuovo algoritmo viene detto wsclock): la pagina viene sostituita solo se non appartiene al working set.

*Algoritmi Locali e Globali*

Gli algoritmi di rimpiazzamento di pagine che usano strategie locali assegnano a ogni processo una quantità fissa di memoria, mentre gli algoritmi globali allocano dinamicamente i frame tra i processi eseguibili. In generale gli algoritmi globali danno risultati migliori, in particolare quando la dimensione del working set varia durante la vita di un processo, tuttavia il sistema deve continuamente decidere quanti frame assegnare a ogni processo.

*Dimensione delle pagine*

La dimensione delle pagine è un parametro del sistema, le motivazioni nella scelta rispondono a esigenze contrastanti.

A favore di pagine piccole:

- Un blocco di memoria non riempirà esattamente un numero intero di pagine, mediamente quindi metà dell'ultima pagina viene sprecata

A favore di pagine grandi:

- con pagine piccole è necessaria una tabella delle pagine grande
- il trasferimento di una pagina piccola da disco richiede quasi lo stesso tempo di una pagina grande

*Interfaccia della memoria virtuale*

In alcuni sistemi avanzati i programmatori hanno un certo controllo sulla mappa di memoria. Un motivo per consentire ai programmatori il controllo sulla mappa di memoria è di consentire a due o più programmi di condividere la stessa memoria: se è possibile dare un nome ad una zona di memoria, un processo può darlo ad un altro processo in modo che quest'ultimo possa inserire la pagina nella sua tabella. La condivisione delle pagine può implementare un sistema a scambio di messaggi ad elevate prestazioni.

*Memory-Mapped File*

Memory-mapped file I/O consente che le operazioni di I/O possano essere trattate come un normale accesso in memoria mappando i blocchi su disco su pagine in memoria. Quando si richiede l'uso di un blocco su disco, questo viene caricato in memoria: l'accesso risulta semplificato in quanto non si usano più le chiamate di sistema read() e write(). Più processi possono condividere in memoria gli stessi file.

- Segmentazione

Con l'approccio precedente la memoria virtuale è unidimensionale (gli indirizzi vanno da 0 all'indirizzo massimo): può essere utile mantenere due o più spazi di indirizzamento separati.

Un compilatore ha molte tabelle che vengono costruite durante la compilazione e che crescono durante la compilazione: in uno spazio di indirizzamento unidimensionale ci possono essere tabelle con molto spazio libero e altre completamente occupate o addirittura che necessitano altro spazio. Ci sono alcune possibili soluzioni ma ciò che effettivamente serve è liberare l'utente dal compito di gestire tabelle che si espandono e contraggono.

La soluzione diretta e generale a problemi di questo tipo è la segmentazione: ogni segmento è una sequenza lineare di indirizzi da 0 a un massimo, la cui lunghezza può variare da 0 a un massimo consentito e può variare durante l'esecuzione. Segmenti diversi possono avere lunghezze diverse.

Per specificare un indirizzo in una memoria segmentata (o bidimensionale), il programma deve fornire un indirizzo costituito da due parti: numero di segmento e indirizzo all'interno del segmento.

Un segmento può contenere una procedura, un array, uno stack o un insieme di variabili ma solitamente non contiene un insieme misto di questi. Un segmento è un'entità logica della quale il programmatore è cosciente e utilizza come una singola entità logica.

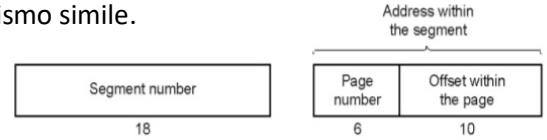
Vantaggi della segmentazione

- Semplifica la gestione di strutture dati che crescono o diminuiscono
- Se ogni procedura occupa un segmento separato, con indirizzo di inizio all'interno del segmento pari a 0, la segmentazione semplifica il linking di procedure compilate separatamente
- Facilita la condivisione di procedure o dati tra alcuni processi
- Dato che ogni segmento è un'entità logica nota al programmatore (procedura, array o stack), segmenti diversi possono avere diversi tipi di protezione

L'implementazione di paginazione e segmentazione ha una differenza fondamentale: le pagine hanno dimensione fissa, i segmenti variabile. Se segmenti vengono continuamente caricati e sostituiti in memoria, dopo un certo tempo si formeranno in memoria delle zone non utilizzate: è il fenomeno del checkerboarding (frammentazione esterna). Una possibile soluzione è la compattazione.

*Segmentazione con paginazione*

La tecnica è stata proposta per sfruttare contemporaneamente i vantaggi della paginazione e della segmentazione. I processori INTEL a 32 bit implementano un meccanismo simile.




- File System

I dati dal punto di vista dell'utente:

- Necessità di memorizzare enormi quantità di informazioni
- Necessità di memorizzare in modo permanente informazioni
- Necessità di accedere contemporaneamente agli stessi dati da parte di più processi
- Necessità di accedere ai dati in maniera ottimizzata I dati dal punto di vista del Sistema Operativo: Il file system

Scopi del file system:

- garantire un accesso permanente, conveniente e consistente
- garantire un uso efficiente delle risorse di memorizzazione

Un file system è l'insieme di algoritmi e strutture dati che realizzano la traduzione tra operazioni logiche sui file e le informazioni memorizzate sui dispositivi fisici (dischi, nastri).

Un file system rappresenta una astrazione unificata dei dispositivi fisici effettivi.

Elementi logici di un file system: file, struttura di directory
Elementi software di un file system: chiamate di sistema, routine di gestione, scheduling, device driver

*File*

Dal punto di vista dell'utente un file è un insieme di dati correlati e associato ad un nome, mentre dal punto di vista del sistema operativo un file è un insieme di byte (eventualmente strutturato).

Il nome è una sequenza (limitata) di caratteri, il cui l'insieme di appartenenza dipende dal sistema operativo: la maggior parte dei sistemi operativi moderni distingue fra lettere maiuscole e minuscole (a volte si usano caratteri UNICODE) e alcuni sistemi operativi dividono il nome in due parti (nome ed estensione) separate da un punto ".". L'estensione permette di classificare il file.

*Struttura di file*

Si distinguono tre tipi diversi di strutture:

- Sequenza di byte (ovvero nessuna struttura)
  - la struttura interna del file è gestita dai programmi applicativi
- Sequenza di record di dimensione fissa
  - le operazioni di lettura restituiscono un record, le operazioni di scrittura sovrascrivono o appendono un record
- Albero di record
  - di lunghezza diversa, caratterizzati da una chiave in base alla quale si ordina l'albero
  - l'operazione base non è ottenere il record successivo, ma un record particolare individuato tramite la chiave

*Tipi di file*

La maggior parte dei file system sono costituiti da directory e file: i **file ordinari** sono costituiti dalle informazioni utilizzate dagli utenti, mentre le **directory** gestiscono la struttura del file system.

In UNIX si hanno anche file speciali a caratteri per gestire i dispositivi di I/O e file speciali a blocchi per modellizzare i dischi. MS-DOS invece riserva alcuni nomi per scopi speciali (con, nul)

I file ordinari sono spesso classificati in file ASCII costituiti da linee di testo e file binari, che hanno generalmente una struttura interna gestita dai programmi.

*Accesso ai file*

Sequenziale:

Usato nei primi sistemi operativi, si basa sul modello di nastro: per accedere ad un dato occorre leggere tutte le registrazioni precedenti

Casuale (Random):

- Implementato dai moderni sistemi operativi, si basa sul modello disco: si può accedere ad ogni dato direttamente

*Directory*

Una directory è spesso essa stessa un file che contiene una voce per ogni file: essa può essere organizzata in due modi:

- ogni voce contiene il nome e gli attributi del file
- ogni voce contiene il nome e un puntatore ad una struttura separata che contiene gli attributi del file.

La struttura del file system risultante può essere di tre tipi:

- Directory unica
    - Tutti i file di tutti gli utenti sono contenuti in una sola directory
    - Facile da implementare, ma impraticabile in ambiente multiutente (causa conflitti sui nomi)
- File System a due livelli
    - I file di utenti diversi sono contenuti in directory separate
    - Possono ancora esistere conflitti sui nomi
- File System ad albero
    - Più directory per ogni utente organizzazione gerarchica
    - In questo modo i conflitti sui nomi minimi e l'organizzazione è più flessibile

*Link*

Sono scorciatoie per accedere a file o directory di frequente, in modo condiviso e attraverso nomi diversi: permettono di avere più di un punto di accesso per lo stesso file o directory, mantenendo le informazioni in un'unica copia.

Il file system NTFS (Windows) permette link simili a UNIX.

*Allocazioni dei file*

Esigenze da soddisfare: accesso veloce ai dati e utilizzo efficiente del disco.

Metodi di allocazione:

- Allocazione contigua
    - Ogni file occupa un insieme contiguo di blocchi su disco, allocati al momento della creazione del file: implementazione semplice (è sufficiente una tabella), ma occorre sapere subito la dimensione del file.

    - Per l'allocazione si usano algoritmi simili a quelli per la gestione di memoria primaria: first fit, best fit. Prestazioni eccellenti lettura e scrittura avvengono tramite un unico blocco.
    - Problemi: frammentazione, espansione dei file
- Allocazione a liste
    - Un file è gestito tramite una lista di blocchi
    - Le directory contengono solo i puntatori al primo blocco
    - Estendere un file è semplice e non esiste frammentazione esterna, ma c'è lentezza di accesso (l'accesso casuale non è semplice)
    - I blocchi su disco devono contenere un puntatore (la dimensione del blocco logico non è una potenza di due)
- Allocazione indicizzata
    - Risolve i problemi precedenti
    - Tutti i puntatori sono memorizzati insieme in un unico blocco (blocco indice), che viene conservato in memoria primaria. L'accesso casuale è ottimizzato in quanto la catena di puntatori è interamente in memoria, ma il blocco indice può raggiungere dimensioni notevoli.

*File Allocation Table*

MS-DOS utilizza l'allocazione indicizzata. Ogni partizione ha una sua FAT con una voce per ogni blocco contenente il numero del blocco successivo; per i blocchi non usati è un puntatore nullo.

Per limitare l'occupazione di memoria i blocchi possono essere di grande dimensione • Struttura delle directory:

*I-Node*

Nel file system UNIX gli attributi dei file sono conservati separatamente dalle directory in una struttura dati chiamata **i-node** (index-node): ogni i-node contiene anche i puntatori ai primi blocchi del file, e se non sono sufficienti uno dei blocchi (blocco a indirezione semplice) è utilizzato per contenere altri indirizzi di blocchi. Se nemmeno questo è sufficiente si utilizza un secondo livello (blocco a indirezione doppia), nei casi estremi si può arrivare ad avere blocchi a indirezione tripla.

*Dimensioni dei blocchi*

La scelta della dimensione dei blocchi è vincolata a diversi parametri:

- Parametri del disco: cilindro, traccia, settore
- Parametri del sistema: dimensione di pagina
- Ottimizzazione dello spazio occupato
    - Se la dimensione media dei file è minore della dimensione del blocco si ha uno spreco di spazio che può essere notevole
- Ottimizzazione del tempo di accesso ai dati

I due parametri considerati hanno esigenze opposte; un altro parametro che può essere utilizzato è la dimensione media dei file utilizzati.

Gestione dei blocchi liberi

Si utilizzano principalmente due tecniche:

- si riservano alcuni blocchi per gestire una lista dei blocchi liberi
- si mantiene una bitmap

*Affidabilità del file system*

Un file system deve essere protetto da danneggiamenti, sia hardware che software. I dischi generalmente contengono settori di riserva, che possono sostituire settori che nel tempo si danneggiano (soluzione a posteriori), ma la soluzione più comune è il backup dei dati, tradizionalmente su nastro.

Consistenza del file system

Per valutare la consistenza dei blocchi si confrontano le liste (o bitmap) dei blocchi liberi e utilizzati.
Quattro casi possibili:

- Nessun errore
- Blocco mancante
- Blocco libero duplicato (con le bitmap non può accadere)
- Blocco utilizzato duplicato

Tipicamente si legge un blocco da disco, si modifica il blocco in memoria e lo si riscrive: se si verifica un crash alcuni blocchi non sono scritti e il file system diventa così inconsistente.

*Prestazioni del file system*
Per migliorare le prestazioni parte dei blocchi su disco sono tenuti in un buffer in memoria (cache di disco), con un'implementazione è analoga alla gestione della paginazione.

Blocchi utilizzati in sequenza dovrebbero essere memorizzati vicino per minimizzare i tempi di posizionamento delle testine (deframmentazione del disco - richiede però molto tempo).

## Input/Output

Si distinguono due categorie di dispositivi.

1. Dispositivi a blocchi
    - Blocco da 512 byte a 64K
    - Ogni blocco può essere letto e scritto indipendentemente dagli altri.
    - Ogni blocco è identificato da un indirizzo
    - I comandi comprendono read, write, seek
    - L'accesso ai file viene fatto tramite mappatura in memoria
    - Ove possibile l'accesso viene fatto a basso livello oppure con un sistema di file system
2. Dispositivi a caratteri
    - Un dispositivo a carattere non è indirizzabile e non ha alcuna primitiva di posizionamento (seek)
    - I comandi comprendono get e put
    - L'editing di linea è possibile mediante librerie ad hoc

Non tutti i dispositivi di I/O ricadono in questa tassonomia (timer, display mappati in memoria): è fondamentale dunque che il S.O. possa gestire l'I/O indipendentemente dal dispositivo.

*I controllori di dispositivo*
Le unità di I/O sono costituite da una componente elettronica e una meccanica.
La parte elettronica è gestita dal controllore di dispositivo o adattatore.

Esempi di controllori:

- IDE (Integrated Drive Electronics)
- (P)ATA ((Parallel) AT Attachment)
- SCSI (Small Computer System Interface)
- SATA (Serial Advanced Technology Attachment)

Spesso i controllori sono integrati sulle schede madri: uso di buffer per blocco o sequenze di caratteri.

Ogni controllore usa un insieme di registri per comunicare con la CPU.

Su alcuni computer (es. 680x0) i registri sono nel normale spazio di indirizzamento della memoria (I/O mappato in memoria), mentre in altri casi si utilizza uno spazio di indirizzamento separato. Esiste anche un approccio misto: il S.O effettua l'I/O scrivendo nei registri del controllore.

*Accesso diretto in memoria (DMA)*
Viene usato per evitare l'I/O programmato e per grandi trasferimenti di dati. Richiede un controller di DMA: si trova concettualmente tra la CPU e la memoria ed è fisicamente è collegato a memoria e CPU tramite bus. Ciò consente di scaricare la CPU dal compito di trasferire dati tra il dispositivo di I/O e la memoria.

*I/O gestito da interrupt*
Al termine di ogni operazione di I/O corrisponde un segnale rilevato dal controllore di interrupt: se non vi sono altre interruzioni in corso la richiesta viene gestita immediatamente, altrimenti viene momentaneamente ignorata.

*Device Driver*
Contiene tutto il software dipendente dal dispositivo:

- Impartisce i comandi al controllore e ne verifica il corretto funzionamento
- Accetta richieste astratte indipendenti dal dispositivo e le traduce in istruzioni dipendenti
- Scrive nei registri del controllore
- Se il gestore è bloccato viene risvegliato da un interrupt.
- Dopo il completamento della operazione di I/O il gestore deve controllare la correttezza dell'operazione.

SW di I/O indipendente dal dispositivo

- Interfacciamento uniforme dei device driver
- Assegnamento dei nomi ai dispositivi
- Protezione dei dispositivi
- Dimensione del blocco indipendente dal dispositivo
- Buffering
- Allocazione della memoria per i dispositivi a blocchi
- Allocazione e rilascio di dispositivi dedicati
- Informazioni sugli errori

*Clock*
I clock o timer servono per gestire il timesharing, per la CPU e per tutta la gestione dei segnali il software per gestire il clock è considerato un device driver. I clock più semplici sono collegati all'alimentazione (una interruzione a ogni ciclo di tensione, 50 o 60 Hz), mentre i clock più complessi sono composti da un oscillatore, un contatore e un registro di caricamento (per quelli programmabili). In genere poi vi è un clock di riserva alimentato a batteria.

L'hardware genera solo una interruzione ad intervalli definiti: qualunque altra attività deve essere gestita via software. Generalmente il driver del clock gestisce:

- L'ora di sistema
- La temporizzazione dei processi
- L'addebito della CPU
- Meccanismi di allarme ed attesa
    i. La gestione degli allarmi può essere fatta tramite una lista ordinata delle richieste di clock pendenti
- Funzioni statistiche

*Ora di sistema*

L'ora di sistema (tempo reale) viene normalmente gestita tramite un contatore che misura i tick a partire da un istante di riferimento (per UNIX 1 gennaio 1970). Con un contatore a 32 bit ed una frequenza di clock di 60Hz in due anni si ha un overflow, per cui in genere si utilizzano due contatori. Contando solo i secondi un contatore è sufficiente per 136 anni, per avere l'equivalente del millenium bug bisogna aspettare fino al ventiduesimo secolo (ma se considero il segno diventa critico il 19 gennaio 2038): in questo caso il numero di tick del secondo corrente viene misurato a parte. Un diverso approccio consiste nel contare i tick dall'istante di avvio del sistema (ma se il sistema rimane attivo più di due anni si ha overflow).