Sistemi Operativi

Alessandro Gerotto

Universitá degli studi di Udine

Insegnante *Scagnetto Ivan*

Anno accademico 2021-22

1. Introduzione ai sistemi operativi e Hardware

Un sistema operativo è un **software di base** che gestisce le risorse hardware e software della macchina, fornendo servizi di base ai software applicativi. Un sistema operativo è quindi fortemente **legato all'hardware** del calcolatore su cui gira. Questa è la ragione per cui solitamente il codice dei S.O. è scritto in linguaggi "vicini" alla macchina (a.e., **Assembly, C**).

Esso è composto da più sottosistemi o componenti software: il **kernel,** lo **scheduler**, il **file system**, il **gestore della memoria**, il **gestore delle periferiche**, l'**interfaccia utente** e lo **spooler di stampa**.

Servizi dei SO

1. **Esecuzione dei programmi**: caricamento dei programmi in memoria ed esecuzione;
2. **Operazioni di I/O**: il so deve fornire un modo per condurre le operazioni di I/O, dato che gli utenti non possono eseguirle direttamente;
3. **Manipolazione del file system**: capacità di creare, cancellare, leggere, scrivere file e directory.
4. **Comunicazioni**: scambio di informazioni tra processi in esecuzione sullo stesso computer o su sistemi diversi collegati da una rete.
5. **Individuazione di errori**: garantire una computazione corretta individuando errori nell'hardware della CPU o della memoria, nei dispositivi di I/O, o nei programmi degli utenti;
6. **Allocazione delle risorse**: allocare risorse a pi`u utenti o processi, allo stesso momento;
7. **Accounting**: tener traccia di chi usa cosa, a scopi statistici o di rendicontazione;
8. **Protezione**: assicurare che tutti gli accessi alle risorse di sistema siano controllate;

Protezione hardware

Funzionamento in dual-mode

Un programma mal funzionante non deve in alcun modo intaccare il corretto funzionamento della macchina.

Per questo l'hardware deve fornire un supporto per differenziare almeno tra due modi di funzionamento riconoscibile tramite un **mode bit** che indica in quale modalità si trova:

- **User mode**: alcune operazioni privilegiate non possono essere eseguite;
- **Supervisor mode** (anche **monitor** mode, **system** mode, **kernel** mode).

Protezione dell'I/O

Tutte le istruzioni di **I/O sono privilegiate**. Per questo si deve assicurare che un programma utente non possa mai passare in modo supervisore (per esempio, andando a scrivere nel vettore delle interruzioni).

Protezione della Memoria

Si deve proteggere almeno il **vettore delle interruzioni** e le **routine di gestione degli interrupt**. Si aggiungono due registri che determinano il range di indirizzi a cui un programma può accedere.

La memoria al di fuori di questo range è protetta:

- **registro base:** contiene il primo indirizzo fisico legale;
- **registro limite:** contiene la dimensione del range di memoria accessibile.

Protezione della CPU con il timer

Si vuole impedire che un programma possa **monopolizzare la CPU**. Per questo viene utilizzato il Timer che interrompe la computazione dopo periodi prefissati, in modo che il so riprenda il controllo.

Invocazione del SO

Dato che le istruzioni di I/O sono privilegiate, il programma utente può eseguire operazione di I/O solo attraverso le **system call** (un processo richiede un'azione da parte dell'os, sono un interrupt software):

- Il controllo passa attraverso il vettore di interrupt alla routine di servizio della trap nel sistema operativo, e il mode bit viene impostato a monitor.
- Il so verifica che i parametri siano corretti, esegue la richiesta e ritorna il controllo all'istruzione che segue la system call.
- Con l'istruzione di ritorno, il mode bit viene impostato a user.

System calls

Indica il meccanismo usato da un **processo a livello utente**, per **richiedere un servizio a livello kernel** del sistema operativo del computer in uso.

Essa è disponibile come funzione in quei linguaggi di programmazione che supportano la programmazione di sistema (ad esempio il linguaggio C), oppure come particolari istruzioni assembly.

Esse hanno tre metodi generali per passare i parametri tra il programma e il sistema operativo:

- Passare i parametri **direttamente nei registri**;
- Memorizzare i parametri in una **in memoria**, passando l'indirizzo come parametro in un registro;
- Il programma fa il push dei parametri sullo **stack**, e il sistema operativo ne fa il pop.

Tipologie

Le categorie principali di system call sono: controllo dei processi/thread, gestione dei file e dei file system, gestione dei dispositivi, gestione delle informazioni, comunicazione.

2. Componenti del SO

I componenti comuni dei sistemi operativi sono: la gestione dei processi, la gestione della memoria principale, la gestione della memoria secondaria, la gestione dell'I/O, la gestione dei file, i sistemi di protezione, le connessioni di rete e il sistema di interpretazione dei comandi.

Gestione dei processi

Il sistema operativo è responsabile delle seguenti attività, relative alla gestione dei processi:

- **Creazione** e **cancellazione** dei processi;
- **Sospensione** e resume dei processi;
- Fornire meccanismi per
    - **sincronizzazione** dei processi;
    - **comunicazione** tra processi;
    - evitare, prevenire e risolvere i **deadlock**.

Gestione della Memoria Principale

La memoria principale è un **array di parole** (byte, words…) rapidamente accessibile dalla cpu e dai dispositivi di I/O, ognuna identificata da un preciso **indirizzo**. La memoria principale è una memoria volatile.

Il sistema operativo è responsabile delle seguenti attività relative alla gestione della memoria:

- **Tener traccia** di quali parti della memoria sono correntemente utilizzate, e da chi.
- **Decidere quale processo** caricare in memoria, quando dello spazio si rende disponibile.
- **Allocare e deallocare** spazio in memoria, su richiesta.

Gestione della Memoria Secondaria

La memoria secondaria è una memoria **non volatile** e con **capacità maggiori** rispetto a quella principale in modo da fargli da supporto.

Il sistema operativo è responsabile delle seguenti attività relative alla gestione dei dischi:

- **Gestione** dello **spazio libero**;
- **Allocazione** dello spazio;
- **Schedulazione** dei dischi.

Gestione dell'I/O (Cap. 10)

Il sistema di I/O consiste in

- sistemi di caching, buffering, spooling;
- una interfaccia generale ai gestori dei dispositivi (device driver);
- i driver per ogni specifico dispositivo hardware (controller).

Gestione dei file

Un file è una collezione di **informazioni correlate**, definite dal suo creatore. I file rappresentano **programmi** e **dati**. Il sistema operativo è responsabile di alcune attività connesse alla gestione dei file:

- **Creazione** e **cancellazione** dei file e delle directory;
- Supporto di primitive per la **manipolazione** di file e directory;
- **Allocazione** dei file nella memoria secondaria;
- **Salvataggio** dei dati su supporti non volatili.

Sistemi di protezione

Per Protezione si intende un meccanismo per **controllare l'accesso** da programmi, processi e utenti sia al sistema, sia **alle risorse** degli utenti.

Il meccanismo di protezione deve:

- **distinguere** tra uso autorizzato e non autorizzato;
- fornire un modo per specificare i **controlli** da imporre;
- forzare gli utenti e i processi a **sottostare** ai controlli richiesti.

Connessioni di rete (networking)

Un sistema distribuito è una collezione di processori che **non condividono memoria o clock**. Ogni processore ha una memoria **propria**. L'accesso ad una risorsa condivisa permette:

- **Aumento delle prestazioni** computazionali;
- **Incremento della quantità** di dati disponibili;
- Aumento dell'**affidabilità**.

Sistema di interpretazione dei comandi

Molti comandi sono dati al sistema operativo attraverso **control statement** che servono per

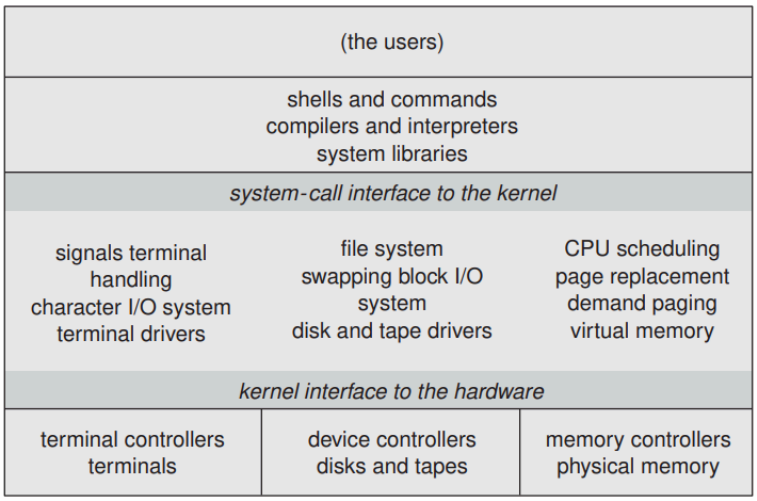
- creare e gestire i processi
- gestione dell'I/O
- gestione della memoria secondaria
- gestione della memoria principale
- accesso al file system
- protezione
- networking

Il programma che legge e interpreta i comandi di controllo ha diversi nomi:

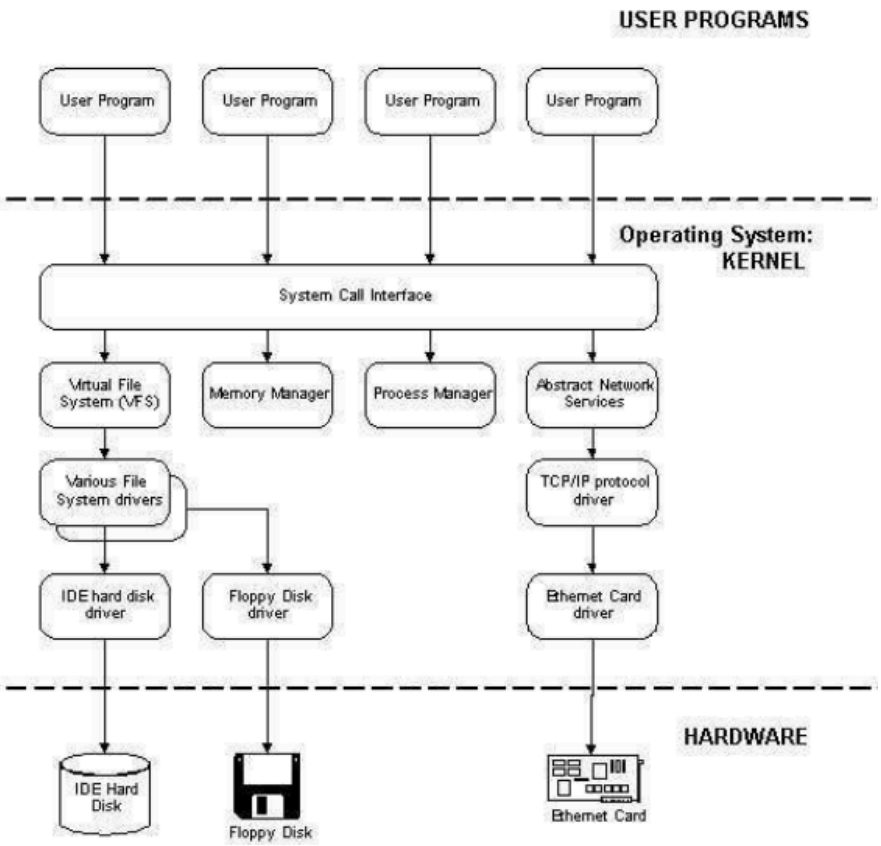
- interprete delle schede di controllo (sistemi batch)
- interprete della linea di comando (DOS, Windows)
- shell (in UNIX)
- interfaccia grafica: Finder in MacOS, Explorer in Windows, gnome-session in Unix…

3. Struttura del SO

Il sistema operativo è diviso in un certo numero di **strati** (livelli) dove ogni strato è costruito su quelli inferiori. Lo strato di base (livello 0) è l'hardware; il più alto è l'interfaccia utente. Gli strati sono pensati in modo tale che ognuno possa utilizzare le funzioni e i servizi solamente di strati inferiori.



Ad esempio: Struttura e stratificazione sistemi operativi Linux



4. Processi, programmi e thread

Programmi

Un programma è un'**entità statica** (rimane immutata durante l'esecuzione) costituita dal **codice oggetto** (traduzione del codice macchina) generato dalla compilazione del codice sorgente.

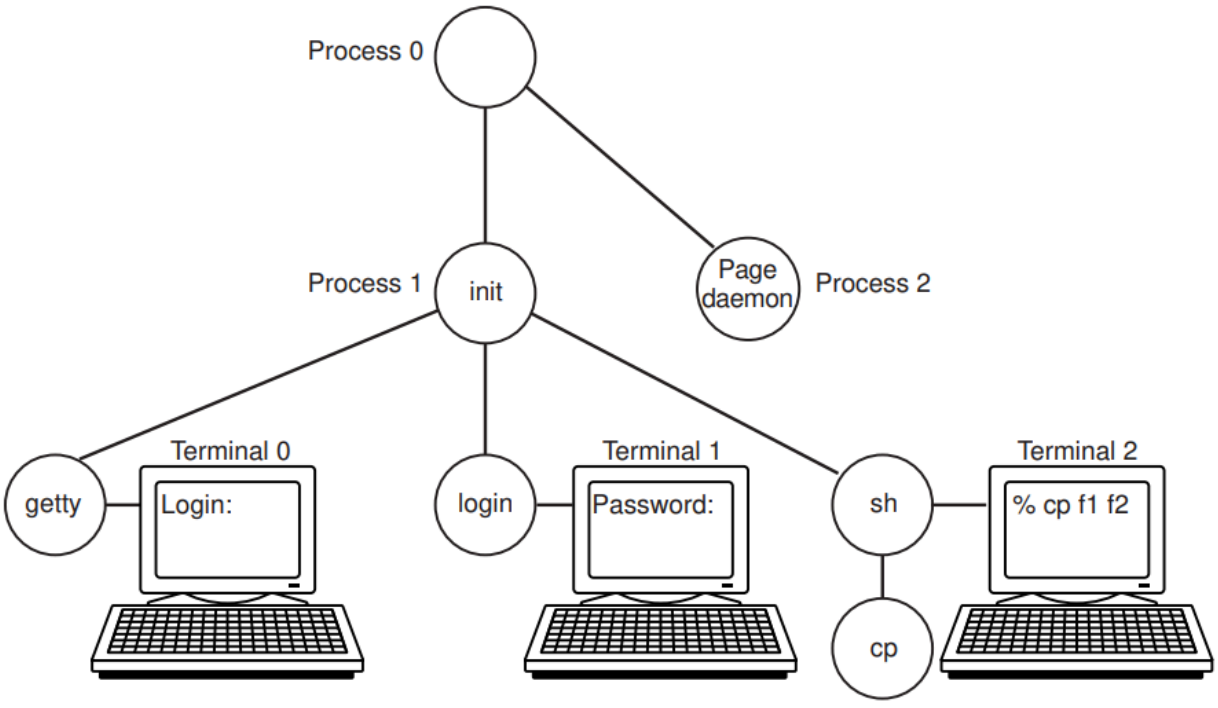
Processi

Il processo è un'**entità dinamica** (dipende dai dati che vengono elaborati e dalle operazioni eseguite su di essi nel ciclo di fetch-execute da parte del processore) caricata su memoria **RAM** generata da un programma (entità statica): identificato da un codice univoco chiamato **PID**. Esso è una **sequenza di attività** (task) controllata da un programma (scheduler) che si svolge su un processore in genere sotto la supervisione del rispettivo sistema operativo.

Il processo è quindi caratterizzato, oltre che dal **codice eseguibile**, da il **tempo di CPU**, **memoria**, **file**, **dispositivi di I/O**, per assolvere il suo compito.

Gerarchia di un processo

In **UNIX** tutti i processi discendono da **init** (PID=1). Se un parent muore, il figlio viene ereditato da init (*Process 0*). Un processo non può diseredare il figlio.



In **Windows** invece non c'è gerarchia di processi: il task creatore ha una handle del figlio.

Creazione di un processo

La generazione dei processi induce una gerarchia, detta **albero di processi**. Un processo viene creato:

- **Al boot** del sistema (intrinseci, daemon);
- Su **esecuzione di una system call** apposita (es., fork());
- Su **richiesta** da parte dell'utente;
- Inizio di un **job batch**;

Terminazione di un processo

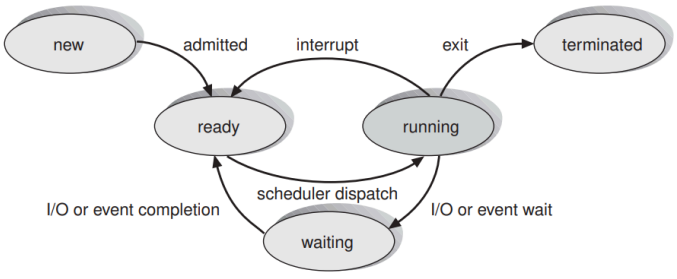
Al momento della terminazione di un processo le risorse del processo sono deallocate dal sistema operativo. Essa può avvenire in diversi modi:

- Terminazione volontaria: I dati di output vengono ricevuti dal processo padre;
- Terminazione involontaria: errore fatale (superamento limiti, operazioni illegali, …)
- Terminazione da parte di un altro processo (uccisione)
- Terminazione da parte del kernel (a.e. il padre termina, e quindi vengono terminati tutti i discendenti: terminazione a cascata)

Stato di un processo in Unix

Ogni processo è accomunato ad uno **stato**. Durante l'esecuzione, i processi cambiano gli stati a seconda di cosa stanno facendo. In generale gli stati possono essere:

- **User running**: esecuzione in modo utente;
- **Kernel running**: esecuzione in modo kernel;
- **Ready to run, in memory**: pronto per andare in esecuzione;
- **Asleep in memory**: in attesa di un evento, processo in memoria;
- **Ready to run, swapped**: eseguibile, ma swappato su disco;
- **Sleeping, swapped**: in attesa di un evento; processo swappato;
- **Preempted**: il kernel lo blocca per mandare un altro processo;
- **Zombie**: il processo non esiste più, il padre attende l'informazione dello stato di ritorno;



PRB: Process Control block

È la **struttura dati** di un processo che contiene le **informazioni essenziali** (pc, aree di salvataggio, stato del processo, un puntatore al processo padre, livello di priorità…) per la gestione del processo stesso.

Durante il **context switch**, è necessario salvare in memoria centrale lo **stato di esecuzione** del processo che viene fermato. Queste informazioni vengono memorizzate proprio nel PCB del processo, e sarà sempre dal PCB che esse verranno ricaricate quando si dovrà proseguire l'esecuzione.

Coda dei processi, ready queue e code dei dispositivi

I processi che sono pronti (in RAM) e attendono di essere eseguiti si trovano in una lista detta **ready queue**. Questa coda generalmente si memorizza come una **lista concatenata**. Il sistema operativo ha anche altre code. L'elenco dei processi che attendono la disponibilità di un particolare dispositivo di I/O si chiama **coda del dispositivo**. Ogni dispositivo ha la sua coda. Durante l'esecuzione, lo **scheduler** sceglie quali processi spostare da una coda all'altra.

Esempio di processo UNIX

Ogni processo UNIX ha uno **spazio indirizzi separato** (non vede le zone di memoria dedicate agli altri processi). Un processo UNIX ha tre segmenti:

- **Stack**: Stack di attivazione delle subroutine. Cambia dinamicamente.
- **Data**: Contiene lo heap e i dati inizializzati al caricamento del programma. Cambia dinamicamente su richiesta esplicita del programma (es., con la malloc).
- **Text**: codice eseguibile. Non modificabile, protetto in scrittura.

Gli Scheduler

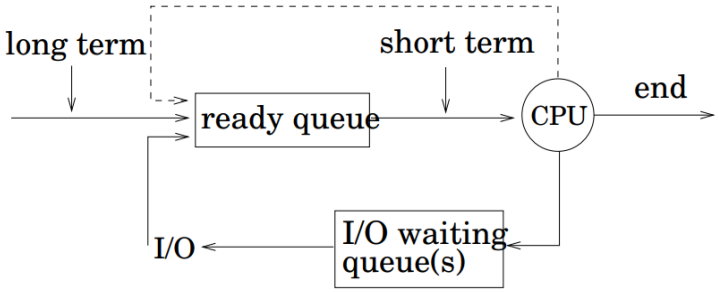
È un componente del so il quale, dato un insieme di richieste di accesso ad una risorsa stabilisce un **ordinamento temporale** per l'esecuzione di tali richieste, privilegiando quelle che rispettano determinati parametri secondo una certa **politica di scheduling**. Generalmente l'obiettivo dello scheduling è quello di:

- **massimizzare il throughput** (produttività dell'intero sistema);
- minimizzare il rapporto tra **tempo di servizio** (tempo che una richiesta impiega per essere soddisfatta) e **tempo di turnaround** (tempo che intercorre tra l'istante in cui la richiesta è generata e quello in cui la richiesta è soddisfatta);
- **evitare la starvation** ("l'attesa eterna" di alcune richieste, verificabile solo in determinate condizioni che non arrivano);
- dare all'utente del sistema la **percezione** che le richieste vengano soddisfatte **contemporaneamente**.

Scheduler di breve e di lungo termine

- Lo scheduler di **lungo termine** seleziona i processi da portare nella ready queue. Viene **invocato raramente** quindi deve essere **lento e sofisticato**;

- Lo scheduler di **breve termine** seleziona quali processi ready devono essere eseguiti. Viene **invocato molto spesso**, quindi deve essere **veloce**.



Processi e Thread

Un thread è una **parte di un processo** che viene eseguita in maniera **concorrente** ed **indipendente** internamente allo stato generale del processo stesso. Un processo ha sempre **almeno un thread** (se stesso). I thread hanno i **propri registri di sistema** che contengono le sue **variabili di lavoro correnti**, uno **stack** che contiene la cronologia di esecuzione e il **contatore** che tiene traccia di quale istruzione eseguire successivamente. Esistono due maggiori differenze tra processi e thread:

- Modalità di condivisione delle risorse: i processi sono di solito fra loro **indipendenti**, i thread di un processo tipicamente **condividono** le medesime informazioni di **stato**, la **memoria** ed altre **risorse di sistema**.
- Il **meccanismo di attivazione**: la creazione di un nuovo processo è sempre costosa per il sistema, in quanto devono essere assegnate risorse necessarie alla sua esecuzione. Il thread invece è parte di un processo e quindi una sua nuova attivazione viene effettuata in **tempi ridottissimi** a **costi minimi**.

Stati ed operazioni sui thread

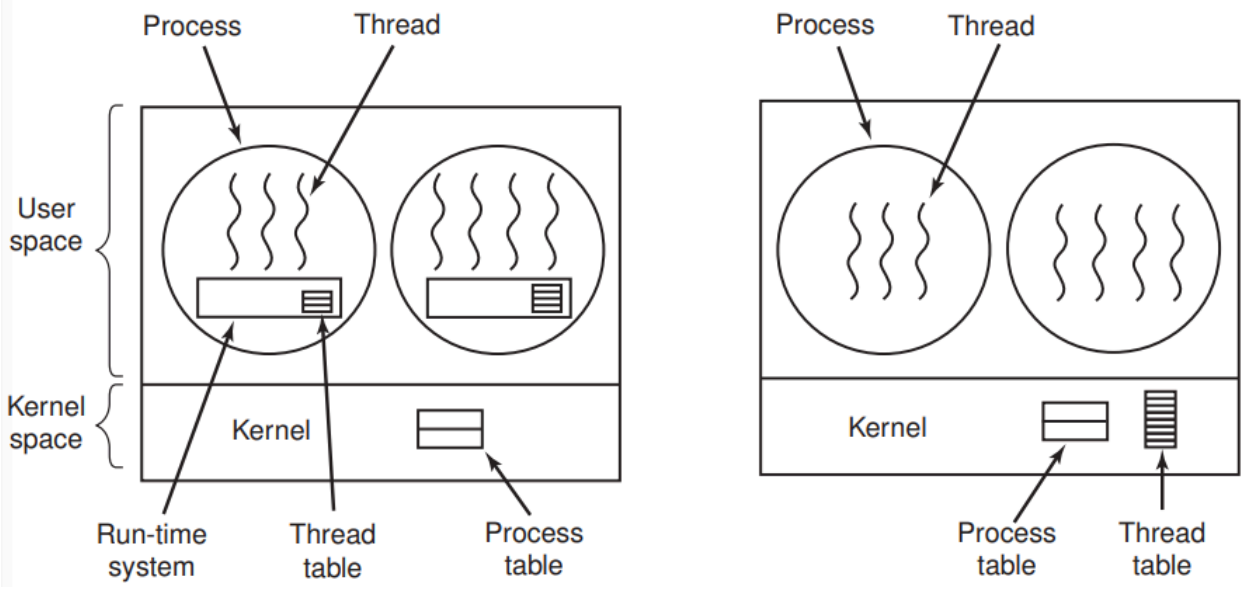
- Stati: **running, ready, waiting, stopped e zombie**.
- Operazioni sui thread:
  - **creazione (spawn)**: un nuovo thread viene creato all'interno di un processo;
  - **blocco**: un thread si ferma, e l'esecuzione passa ad un altro thread/processo;
  - **sblocco**: quando avviene l'evento, il thread passa dallo stato "blocked" already";
  - **cancellazione**: il thread chiede di essere cancellato. Il suo stack e le copie dei registri vengono deallocati.
- Meccanismi per la **sincronizzazione** tra i thread: indispensabili per l'accesso concorrente ai dati in comune.

Implementazione a livello utente e a livello kernel

Esistono due tipi principali di thread:

- Thread a livello utente (thread gestiti dall'utente)

- Thread a livello di kernel (thread gestiti dal so che agiscono sul kernel, un core del sistema operativo).



Multiprogrammazione (multitasking)

Indica la capacità di un software di **eseguire** più programmi **contemporaneamente**: se ad esempio viene chiesto al sistema di eseguire contemporaneamente due processi A e B, la CPU eseguirà per qualche istante di tempo il processo A, poi per qualche istante successivo il processo B, poi tornerà a eseguire il processo A e così via

Il passaggio dal processo A al processo B e viceversa viene definito "commutazione di contesto" (**context switch**). Il tempo di context-switch porta un certo **overhead**; il sistema non fa un lavoro utile mentre passa di contesto.

Esistono principalmente due tipi di multiprogrammazione:

- **Tipo batch**: in cui non c'è interattività con l'utente.
- **Tipo time-sharing**: in cui si permette a più utenti di utilizzare contemporaneamente e interattivamente lo stesso sistema, massimizzando la praticità d'uso, a scapito delle prestazioni.

5. Scheduling della CPU

Lo scheduling della CPU consiste nella **selezione** tra i processi in memoria e pronti per l'esecuzione, quello a cui allocare la CPU. La decisione dello scheduling può avere luogo quando un processo:

1. passa da **running a waiting** (nonpreemptive);
2. passa da **running a ready**;
3. passa da **waiting o new a ready**;
4. **Termina** (nonpreemptive).

Con/senza prelazione

È importante la distinzione tra:

- **scheduling con diritto di prelazione** (scheduling preemptive): lo scheduler può sottrarre il possesso del processore al processo anche quando questo potrebbe proseguire nella propria esecuzione.
- **scheduling senza diritto di prelazione** (scheduling non-preemptive o cooperative): lo scheduler deve attendere che il processo termini o che cambi il suo stato da quello di esecuzione a quello di attesa.

Dispatcher

Il dispatcher è il modulo che **dà il controllo della CPU** al processo selezionato dallo scheduler di breve termine. Questo comporta:

1. **context switch**;
2. passaggio della CPU da modo **supervisore** a modo **user**;
3. **salto alla locazione** del programma utente per riprendere il processo;

La **latenza di dispatch** è il tempo necessario per fermare un processo e riprenderne un altro.

Criteri di valutazione

Esistono vari algoritmi di scheduling che possono essere più indicati in alcuni contesti piuttosto che in altri. La scelta dell'algoritmo da usare dipende da cinque principali criteri:

- **Utilizzo del processore**: devono essere ridotti al minimo i possibili tempi morti della CPU;
- **Throughput**: il numero di processi completati in una determinata quantità di tempo;
- **Tempo di completamento**: il tempo tra la sottomissione di un processo ed il suo completamento;
- **Tempo d'attesa**: il tempo in cui un processo pronto per l'esecuzione rimane in attesa della CPU;
- **Tempo di risposta**: il tempo che trascorre tra la sottomissione del processo e l'ottenimento della prima risposta.

Obiettivi dello scheduling

Un algoritmo di scheduling si pone i seguenti obiettivi:

- **Fairness** (equità): processi dello stesso tipo devono avere trattamenti simili;
- **Balance** (bilanciamento): tutte le parti del sistema devono essere sfruttate;

In più, nei **sistemi batch** il throughput deve essere massimizzato e il Tempo di Turnaround deve essere minimizzato.

Nei **sistemi interattivi**, il tempo di risposta deve essere il minimo possibile per dare l'idea di continuità all'utente e la proporzionalità deve essere rispettata, ossia il tempo di risposta deve essere proporzionale alla complessità dell'azione.

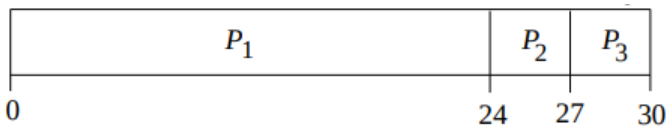
Scheduling First-Come, First-Served (FCFS)

È un tipo di algoritmo FIFO: esegue i processi nello stesso ordine in cui essi vengono sottomessi al sistema. Il primo processo ad essere eseguito è esattamente quello che per primo richiede l'uso della CPU. Quelli successivi vengono serviti non appena questo ha terminato la propria esecuzione, per questo viene definito come un algoritmo **equo**, ovvero senza il rischio di starvation. Questo tipo di algoritmo è **molto semplice** da implementare ma solitamente è anche **poco efficiente**.

Esempio

Infatti, prendiamo ad esempio la sottomissione nell'ordine dei seguenti processi con la seguente durata espressa in millisecondi:

Processo	**Burst time**
P1	24
P2	3
P3	3



Tempi di attesa: P1=0, P2=24, P3=27

Tempo di attesa medio: (0+24+27)/3=17

Si ha allora un **effetto convoglio**, caratterizzato dal fatto che più processi brevi attendono la terminazione di un unico processo più corposo.

Tuttavia se i processi sono **CPU-bound** (processi che sfruttano pesantemente le risorse computazionali del processore, ma non richiedono servizi di ingresso/uscita dati al sistema operativo in quantità rilevanti) e lunghi, il FCFS rimane l'algoritmo migliore.

Scheduling Shortest-Job-first (SJF)

Si associa ad ogni processo la lunghezza del suo prossimo burst di CPU. I processi vengono ordinati e schedulati per tempi crescenti. Gli algoritmi SJF possono essere sia sia.

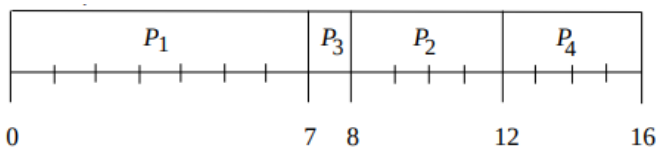
- **non-preemptive (SNPF)**: quando la CPU viene assegnata ad un processo, questo la mantiene finché non termina il suo burst.
- **preemptive (SRTF)**: se nella ready queue arriva un nuovo processo il cui prossimo burst è minore del tempo rimanente per il processo attualmente in esecuzione, quest'ultimo viene prelazionato.

Questo algoritmo risulta essere un ottimale: fornisce il **minimo tempo di attesa** per un dato insieme di processi ma si **rischia la starvation**.

Tuttavia permane un **problema**: come si può conoscere la durata di un ciclo di burst che deve ancora avvenire? Non esiste una risposta a questo quesito ma si può solo **darne una stima** (nei sistemi batch il tempo viene stimato dall'utente e nei sistemi time-sharing si usano i tempi delle precedenti elaborazioni).

Esempio SJF Non-Preemptive

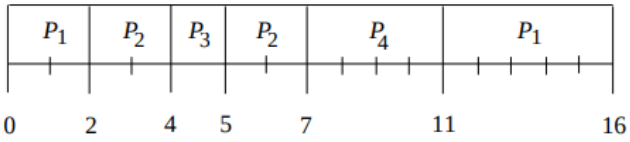
Processo	Arrival time	Burst time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4



Tempo di attesa medio: (0+6+3+7)/4=4

Esempio SJF Preemptive

Processo	Arrival time	Burst time
P1	0.0	7
P2	2.0	4
P3	4.0	1
P4	5.0	4



Tempo di attesa medio: (9+1+0+2)/4=3

Scheduling con priorità

Viene associato un numero intero ad ogni processo. La cpu verrà allocata al processo con il numero più piccolo. La grandezza del numero viene decisa in due modi:

- **internamente**: in base a parametri misurati dal sistema sul processo (tempo di CPU impiegato, file aperti, memoria, interattività, uso di I/O…)
- **esternamente**: importanza del processo, dell'utente proprietario, dei soldi pagati, . . .

Gli scheduling con priorità possono essere preemptive o nonpreemptive.

Ad esempio l'SJF è uno scheduling a priorità, dove la priorità è il prossimo burst di CPU previsto. Tuttavia esiste un problema per questo algoritmo di scheduling, ovvero che i processi a bassa priorità restano bloccati da un flusso continuo di nuovi processi a priorità maggiore (**starvation**).

Per rimediare a questo problema, viene introdotto il concetto di **aging**: con il passare del tempo i processi non eseguiti aumentano la loro priorità.

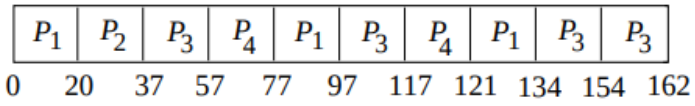
Round Robin (RR)

È un algoritmo con prelazione (**preemptive**) che esegue i processi nell'ordine d'arrivo, come il FCFS, ma in cui ogni processo riceve una piccola unità di tempo di CPU (**quanto**, tipicamente 10-100 millisecondi). Dopo questo periodo, il processo viene prelazionato e rimesso nella coda di ready.

Se ci sono n processi in ready, e il quanto è q, allora ogni processo riceve 1/n del tempo di CPU in periodi di durata massima q. Nessun processo attende più di $(n - 1)q$.

Esempio RR con quanto di 20

Processo	Burst time
P1	53
P2	17
P3	68
P4	24



Prestazioni RR

q grande: degenera nell'FCFS

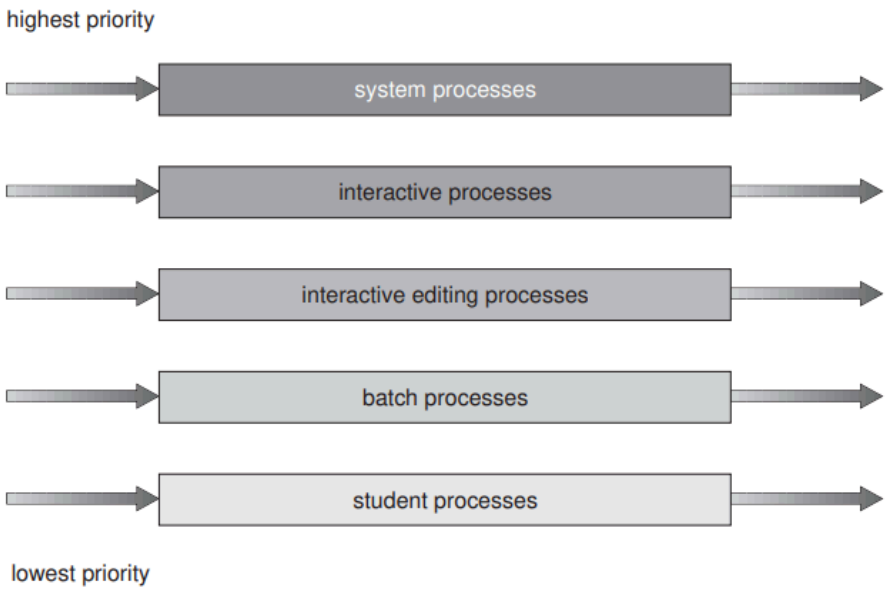
q piccolo: q deve comunque essere grande rispetto al tempo di context switch, altrimenti l'overhead è elevato.

Scheduling con code multiple

La coda di ready viene partizionata in **più code separate** ognuna con un proprio livello di priorità e con un proprio algoritmo di scheduling (a.e. RR per i foreground, FCFS per i background).

Lo scheduling deve avvenire tra tutte le code:

- **Scheduling a priorità fissa**: eseguire i processi di una coda solo se le code di priorità superiore sono vuote (possibilità di starvation).
- **Quanti di tempo per code**: ogni coda riceve un certo ammontare di tempo di CPU per i suoi processi (a.e. 80% ai foreground in RR, 20% ai background in FCFS).



Scheduling con code multiple con feedback

I processi vengono spostati da una coda all'altra, dinamicamente (a.e. per implementare l'aging) Uno scheduler a code multiple con feedback viene definito dai seguenti parametri:

- **numero di code**;
- **algoritmo di scheduling** per ogni coda;
- come determinare **quando promuovere** un processo;
- come determinare **quando degradare** un processo;
- come determinare la coda in cui **mettere un processo che entra** nello stato di ready.

Esempio

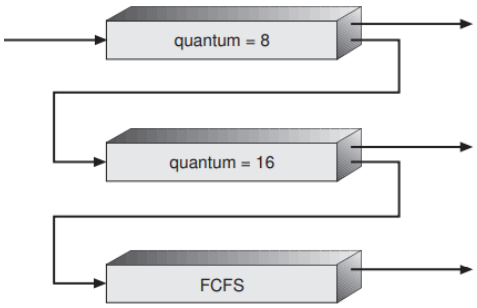
Tre code:

- Q_0 quanto di 8 ms;
- Q_1 quanto di 16 ms;
- Q_2 FCFS.

Un nuovo job entra in Q_0 dove viene servito FCFS con prelazione. Se non termina nei suoi 8 millisecondi, viene spostato in Q_1;

Nella coda Q_1 ogni job è servito FCFS con prelazione, quando Q_0 è vuota. Se non termina in 16 millisecondi, viene spostato in Q_2;

Nella coda Q_2, ogni job è servito FCFS senza prelazione, quando Q_0 e Q_1 sono vuote.



Scheduling garantito

Viene fatta una **promessa** (a.e. se ci sono n utenti connessi, ogni utente riceverà 1/n della potenza di CPU). Per mantenere la promessa fatta 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;
- del 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.

Scheduling a lotteria o a estrazione

Rappresenta una semplice implementazione di uno "scheduling garantito". Esistono un certo numero di "**biglietti**" per ogni risorsa. Ogni processo, al momento della creazione, acquisisce un sottoinsieme di tali biglietti. Successivamente viene estratto casualmente un biglietto, e la risorsa viene assegnata al vincitore.

Più biglietti in processo detiene, più ha una **priorità elevata** dato che ha più probabilità di essere estratto.

Un aspetto utile in questa tipologia di scheduling è il fatto che i biglietti **possono essere passati** da un processo all'altro per cambiare la priorità di un processo.

Scheduling multi-processore

Lo scheduling diventa più complesso quando più CPU sono disponibili. Nei sistemi omogenei il problema non si pone dato che è indifferente su quale processore si eseguirà il prossimo task.

Può comunque essere richiesto che un certo task venga eseguito su un preciso processore (**pinning**). Per garantire le performance, è molto importante bilanciare il carico di lavoro (load sharing).

Per fare ciò tutti i processori selezionano i processi dalla stessa ready queue.

Si pone il **problema di accesso condiviso** alle strutture del kernel:

- **Asymmetric multiprocessing** (AMP): solo un processore per volta può accedere alle strutture dati del kernel (semplifica il problema, ma diminuisce le prestazioni (carico non bilanciato));
- **Symmetric multiprocessing** (SMP): condivisione delle strutture dati. Serve hardware particolare e controlli di sincronizzazione in kernel.

Scheduling Real-Time

Un sistema real-time garantisce un certo tempo di risposta deciso a design-time. L'obiettivo di un sistema real-time è il completamento dei task nel **tempo stabilito**, non il prima possibile. Per questo motivo, un sistema real-time non è un sistema estremamente veloce ma un sistema estremamente prevedibile.

Classificazione

Una prima classificazione dei sistemi real-time riguarda la tollerabilità del **non rispetto delle deadline** temporali:

- Sistema **hard real-time**: il non rispetto delle deadline **non è ammesso**; il mancato raggiungimento di una sola deadline può portare a conseguenze catastrofiche nell'ambiente in cui il sistema opera (a.e. I sistemi di controllo delle centrali nucleari, i sistemi airbag…);
- Sistema **soft real-time**: il mancato rispetto di alcune deadline **è ammesso purché entro certi limiti**; se la deadline è mancata, il sistema può utilizzare il risultato, tipicamente degradando in computer performance senza causare eccessivi problemi (a.e. Un lettore multimediale).

Gli eventi si dividono in

- Eventi **aperiodici**: imprevedibili (a.e. La segnalazione di un sensore);
- Evento **periodici**: avvengono ad intervalli precisi (a.e. codifica audio, video). Dati m eventi periodici, questi sono *schedulabili* se:

  $\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$ con P_i = periodo dell'evento i e C_i = tempo di cpu necessario per gestire l'evento i.

Rate Monotonic Scheduling (RMS)

È un algoritmo di scheduling che appartiene alla categoria di scheduling dei sistemi operativi real-time solo per processi periodici a costo costante. È di natura **preemptive** infatti la priorità viene decisa in base al **tempo di ciclo** dei processi coinvolti. Se il processo ha una durata del lavoro ridotta, ha la priorità più alta. Pertanto, se un processo con la priorità più alta inizia l'esecuzione, prevarrà gli altri processi in esecuzione.

Scheduling Earliest Deadline First (EDF)

È un algoritmo di scheduling che appartiene alla categoria di scheduling dei sistemi operativi real-time adatto anche a processi non periodici. Risulta essere più complesso e costoso di RMS. È un algoritmo di scheduling che funziona a priorità dinamiche a seconda di chi scade prima.

Scheduling di Linux

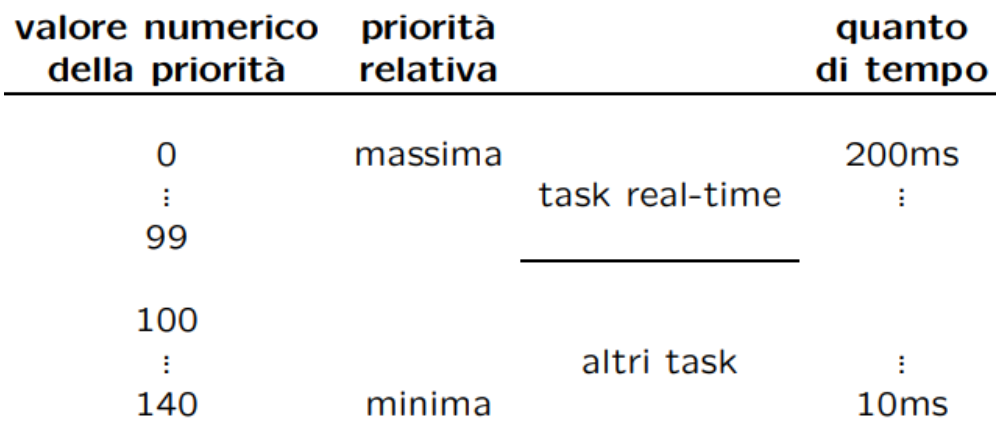
In Linux ad ogni processo è assegnato un **quanto di tempo massimo** per l'esecuzione (tipicamente di 20 clock ticks o 210 ms). La selezione del prossimo processo da eseguire è basata sul concetto di priorità questa può essere

- **dinamica**: per evitare il fenomeno della starvation, coincide con il numero di clock tick.
- **statica:** introdotta per consentire la gestione di processi real-time. Ai processi real-time è assegnata una priorità maggiore di quella assegnata ai processi ordinari.

Di norma Linux prevede uno scheduling con prelazione (preemptive), per cui ad un processo viene tolta la CPU se:

- **esaurisce il quanto** di tempo a sua disposizione;
- un **processo a priorità più alta** è pronto per l'esecuzione.

Algoritmo di scheduling…



…per processi generali

Il tempo è suddiviso in periodi, detti **epoche**, che si ripetono ciclicamente. All'inizio di ogni epoca, a ciascun processo viene assegnato un quanto di tempo. Un'epoca termina quando tutti i processi eseguibili hanno esaurito il loro quanto. Terminata un'epoca, ne inizia un'altra. All'inizio della nuova epoca viene calcolato il **nuovo quanto** di tempo da assegnare a ciascun processo.

…per i processi real-time

Per i processi **real-time** non vale la suddivisione del tempo in epoche. Infatti è assegnato un livello di **priorità statica** che sarà sempre maggiore a quelli di processi non real-time. Per questo un processo ordinario può essere eseguito solo se non ci sono processi real-time pronti per l'esecuzione. I processi real-time possono appartenere ad una di due categorie:

- *SCHED_FIFO*: processi real-time con **quanto illimitato**. Essi **lasciano la CPU solo** se:
    - si bloccano;
    - terminano;
    - un processo a priorità più alta passa nello stato di pronto;
- *SCHED_RR*: processi real-time soggetti a scheduling di tipo **Round Robin**.

6. Cooperazione dei processi

I processi si dividono in:

- I Processi **indipendenti:** non possono modificare o essere modificati da un altro processo;
- I processi **cooperanti**: possono modificare o essere modificati dall'esecuzione di altri processi (a.e. caso produttore-consumatore dove un produttore produce un'informazione che viene poi usata da un processo consumatore). Poter avere cooperazione tra processi implica diversi vantaggi:
    - **Condivisione** delle informazioni;
    - Aumento della computazione (**parallelismo**);
    - **Modularità**;
    - **Praticità** implementativa/di utilizzo.

Nel momento in cui due processi devono comunicare, bisogna tenere conto di alcune cose:

- Come può un processo passare informazioni ad un altro?
- Come evitare accessi inconsistenti a risorse condivise?
- Come sequenzializzare gli accessi alle risorse secondo la causalità?

Race conditions

È un fenomeno che si presenta nei sistemi concorrenti quando, in un sistema basato su processi multipli, il **risultato finale** dell'esecuzione di una serie di processi **dipende dalla temporizzazione** o dalla sequenza con cui vengono eseguiti.

Spesso, una situazione di corsa è un **effetto indesiderato**. In questo caso, essa è denominata **corsa critica** per il sistema.

Evitare il busy wait

Le soluzioni basate su spinlock portano a:

- **busy wait**: alto consumo di CPU;
- **inversione di priorità**: un processo a bassa priorità che blocca una risorsa può essere bloccato all'infinito da un processo ad alta priorità in busy wait sulla stessa risorsa.

Idea migliore:

- quando un processo deve **attendere** un evento, il pid viene **spostato in wait** (sleep());
- quando l'**evento avviene**, il pid viene **spostato in ready** (wakeup(pid)).

Soluzioni

Per evitare il verificarsi di situazioni di corsa quando si impiegano memorie, file o risorse in condivisione, sono stati studiati diversi algoritmi che prevedono la **mutua esclusione** (se il processo P sta eseguendo la sua sezione critica, allora nessun altro processo può eseguire la propria sezione critica).

Se più processi hanno la possibilità di accedere ad una risorsa in modalità:

- **scrittura**: importante prevedere ed includere l'utilizzo di algoritmi,
- **lettura**: i processi non possono influenzare lo stato della risorsa, quindi non è fisicamente possibile la realizzazione di situazioni di corsa e l'utilizzo di algoritmi che prevedono la mutua esclusione non serve.

Soluzioni hardware

Il processo in corso può **disabilitare tutti gli interrupt** hardware all'ingresso della sezione critica, e riabilitarli all'uscita:

- soluzione semplice; garantisce la mutua esclusione;
- pericolosa: il processo può non riabilitare più gli interrupt, acquisendo la macchina;
- può allungare di molto i tempi di latenza;
- non si estende a macchine multiprocessore (a meno di non bloccare tutte le altre CPU).

Soluzioni software

Prima di usare le variabili condivise (regione critica) ciascun processo chiama una *enter_region()* con il proprio numero di processo (0 o 1). Dopo aver finito di lavorare con le variabili condivise, il processo chiama una *exit_region()* per permettere a un altro processo di entrare .

```
While (TRUE){
        enter_region();
                Sezione critica
        exit_region();
                Sezione non critica
}
```

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

Alternanza stretta

È un esempio di **busy wait**: si continua a testare un valore finché non cambia (spin lock). Semplice da implementare ma può portare a consumi inutili alla cpu. Risulta ottimale in caso di attese molto brevi.

Algoritmo di Peterson (1981)

È un algoritmo che definisce *exit_region() e enter_region()*. Esempio:

- Nessun processo è nella sezione critica, il processo 0 chiama la *enter_region* indicando il suo interesse mettendo a *true* il proprio elemento nell'array;
- Se il processo 1 non è interessato, *enter_region* termina subito;
- Se 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.

Algoritmo del fornaio

È un algoritmo che definisce *exit_region() e enter_region()*. Prima di entrare nella sezione critica, ogni processo **riceve un numero**. Chi ha il numero **più basso** entra nella sezione critica. Eventuali conflitti vengono risolti da un ordine statico: Se i processi Pi e Pj ricevono lo stesso numero: se i < j, allora Pi è servito per primo.

Istruzioni Test&Set

```
enter_region:
    TSL REGISTER,LOCK          | copy lock to register and set lock to 1
    CMP REGISTER,#0            | was lock zero?
    JNE enter_region           | if it was non zero, lock was set, so loop
    RET | return to caller; critical region entered


leave_region:
    MOVE LOCK,#0               | store a 0 in lock
    RET | return to caller
```

Semafori e Mutex

Viene definita una **variabile intera** *s* come semaforo. Supporta **due operazioni**

- ***up(s)*** che incrementa *s* (mettendo uno dei processi in attesa nello stato ready);

  s.value := s.value - 1;

  If s.value < 0

  Togli un processo P da S.L;

  sleep();

- ***down(s)*** che attende (spostando il processo in uno stato di wait) finché è maggiore di 0 per poi decrementarlo.

  s.value := s.value + 1;

  If s.value <= 0

  Togli un processo P da S.L;

  wakeup(P);

I mutex sono **semafori** con **due soli possibili valori**: bloccato o non bloccato. Supportano due primitive: *mutex_lock* e *mutex_unlock*.

L'uso dei semafori può portare a **deadlock**, situazioni in cui due o più processi sono in attesa indefinita di eventi che possono essere causati solo dai processi stessi in attesa. Programmare con i semafori è molto delicato e **prono ad errori**, difficilissimi da debuggare.

Monitor

Un monitor è un tipo di **dato astratto** che fornisce funzionalità di mutua esclusione:

- collezione di dati privati e funzioni/procedure per accedervi;
- processi possono chiamare le procedure ma non accedere alle var locali;
- un solo processo alla volta può eseguire codice di un monitor.

Il programmatore **raccoglie** quindi i **dati condivisi** e tutte le sezioni critiche relative in un monitor; questo risolve il problema della mutua esclusione.

```
monitor example
    integer i;
    condition c;

    procedure producer( );
    .
    .
    .
    end;

    procedure consumer( );
    .
    .
    .
    end;
end monitor;
```

Scambio di messaggi

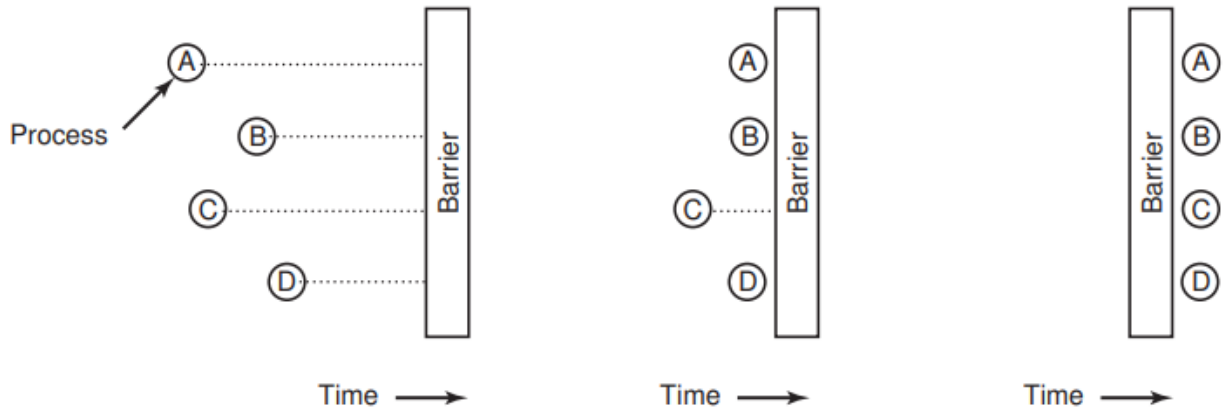
Comunicazione **non basata su memoria condivisa** con controllo di accesso. Basato su due primitive:

- ***send(destinazione, messaggio)***: spedisce un messaggio ad una certa destinazione; solitamente non bloccante;
- ***receive(sorgente, &messaggio)***: riceve un messaggio da una sorgente; solitamente bloccante (fino a che il messaggio non è disponibile).

Tuttavia spesso risulta un metodo poco affidabile (a.e. reti) e necessita di **appositi protocolli** fault-tolerant, richiede un modo per autenticare i due partner e necessita di avere la certezza che i canali che comunicano non siano intercettati. In termini di efficienza, se la comunicazione avviene sulla stessa macchina, il passaggio di messaggi è sempre più lento rispetto ad altri metodi.

Barriere

Meccanismo di sincronizzazione per gruppi di processi. Ogni processo alla fine della sua computazione, chiama la funzione **barrier()** e si sospende. Quando tutti i processi hanno raggiunto la barriera, la superano tutti assieme (si sbloccano).



I grandi classici

Esempi paradigmatici di programmazione concorrente. Presi come testbed per ogni primitiva di programmazione e comunicazione.

- Produttore-Consumatore a buffer limitato
- I Filosofi a Cena
- Lettori-Scrittori
- Il Barbiere che Dorme

I filosofi a cena (1965)

n filosofi seduti attorno ad un tavolo rotondo con n piatti di spaghetti e n forchette (bastoncini). (nell'esempio, n = 5)

- Mentre pensa, un filosofo non interagisce con nessuno
- Quando gli viene fame, cerca di prendere le bacchette più vicine, una alla volta.
- Quando ha due bacchette, un filosofo mangia senza fermarsi.
- Terminato il pasto, lascia le bacchette e torna a pensare.

Problema: programmare i filosofi in modo da garantire:

- **assenza di deadlock**: non si verificano mai blocchi. Può accadere nel momento in cui tutti i filosofi prendono contemporaneamente la forchetta alla loro sinistra.
- **assenza di starvation**: un filosofo che vuole mangiare, prima o poi mangia.

Un modo per garantire il corretto funzionamento dell'algoritmo è quello di

- introdurre un **semaforo mutex** per proteggere la sezione critica;
- Tenere traccia dell'intenzione di un filosofo di mangiare. Un filosofo ha tre stati (THINKING, HUNGRY, EATING), mantenuto in un **vettore state**. Un filosofo può entrare nello stato EATING solo è HUNGRY e i vicini non sono EATING.

Lettori e scrittori

Un **insieme di dati** (es. un file, un database…), deve essere **condiviso** da processi lettori e scrittori. Sono necessari alcuni accorgimenti:

- Due o più lettori possono accedere contemporaneamente ai dati;
- Ogni scrittore deve accedere ai dati in modo esclusivo.

Implementazione con i semafori:

- Tenere conto dei lettori in una variabile condivisa, e fino a che ci sono lettori, gli scrittori non possono accedere.
- Da maggiore priorità ai lettori che agli scrittori.

Il barbiere che dorme

In un negozio c'è **un solo barbiere**, **una sedia** da barbiere e **n sedie per l'attesa**.

- Quando non ci sono clienti, il barbiere dorme sulla sedia.
- Quando arriva un cliente, questo sveglia il barbiere se sta dormendo.
- Se la sedia è libera e ci sono clienti, il barbiere fa sedere un cliente e lo serve.
- Se un cliente arriva e il barbiere sta già servendo un cliente, si siede su una sedia di attesa se ce ne sono di libere, altrimenti se ne va.

Problema: programmare il barbiere e i clienti filosofi in modo da garantire assenza di deadlock e di starvation.

Un modo per garantire il corretto funzionamento dell'algoritmo è quello di

- Utilizzare **tre semafori**
    - *Customers*: clienti in attesa;
    - *Barbers*: conta i barbieri in attesa;
    - *Mutex*: per la mutua esclusione
- Ogni cliente prima di entrare nel negozio **controlla se ci sono sedie libere**; altrimenti se ne va.

7. Deadlock

Indica una situazione in cui **due o più processi** o azioni **si bloccano a vicenda**, aspettando che uno esegua una certa azione che serve all'altro e viceversa. Questa situazione **non può essere risolta**, ma si può **prevenire.** Applicazioni che sono tipicamente soggette agli stalli sono le **basi di dati** oppure i **sistemi operativi** che gestiscono l'accesso contemporaneo a file e a dispositivi di I/O di diversi processi. I deadlock si possono verificare solo se sono presenti contemporaneamente alcune condizioni, dette anche di **Havender**:

- **Mutua esclusione**: almeno una delle risorse del sistema deve essere 'non condivisibile' (ossia usata da un processo alla volta oppure libera).
- **Accumulo incrementale**: i processi che possiedono almeno una risorsa devono attendere prima di richiederne altre (già allocate ad altri processi).
- **Impossibilità di prelazione**: solo il processo che detiene la risorsa può rilasciarla.
- **Attesa circolare**: esiste un gruppo di processi {P0,P1,...,Pn} per cui P0 è in attesa per una risorsa occupata da P1, P1 per una risorsa di P2, ecc. Pn per una risorsa di P0.

Grafo di allocazione risorse

Una **risorsa** è una componente del sistema di calcolo a cui i processi possono accedere in modo esclusivo, per un certo periodo di tempo. Si dividono in:

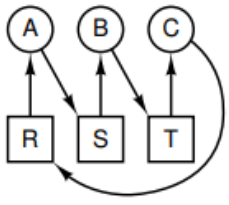
- **Risorse prerilasciabili**: possono essere tolte al processo allocante, senza effetti dannosi. Esempio: memoria centrale;
- **Risorse non prerilasciabili**: non possono essere cedute dal processo allocante, pena il fallimento dell'esecuzione. Esempio: stampante. I deadlock si hanno con le risorse non prerilasciabili.

Protocollo di utilizzo di una risorsa:

- **richiedere** la risorsa se non disponibile, ci sono diverse alternative (attesa, attesa limitata…);
- **usare** la risorsa;
- **rilasciare** la risorsa.

Le quattro condizioni viste sopra si modellano con un grafo orientato, detto **grafo di allocazione delle risorse**: un insieme di vertici V e un insieme di archi E.

- V è partizionato in due tipi:
  - P = {P1, P2, … , Pn}, l'insieme di tutti i processi del sistema.
  - R = {R1, R2, …, Rm}, l'insieme di tutte le risorse del sistema.
- archi di richiesta: archi orientati Pi→Rj
- archi di assegnamento (acquisizione): archi orientati Rj→Pi



Uno **stallo** è un **ciclo nel grafo** di allocazione delle risorse. Esempio di deadlock: *A requests R, B requests S, C requests T, A requests S, B requests T, C requests R*. Quindi se il grafo:

- **non contiene dei cicli:** è impossibile che si verifichi un deadlock;
- **contiene dei cicli**:
  - c'è deadlock se c'è solo una istanza per tipo di risorsa
  - c'è possibile deadlock se ci sono più istanze per tipo di risorsa.

I grafi di allocazione risorse sono uno strumento per **verificare** se una sequenza di allocazione porta ad un deadlock. Ad esempio:

- Il FCFS è una politica “safe”, ma insoddisfacente per altri motivi.
- Il round-robin in generale non è safe.

Gestione dei deadlock

In generale ci sono quattro possibilità:

1. **Ignorare il problema**, fingendo che non esista.
2. **Permettere l’entrata** in un deadlock, **riconoscerlo** e quindi **risolverlo**.
    a. **Riconoscimento**: tramite un algoritmo di ricerca dei cicli per grafi orientati (l’algoritmo può essere richiamato ad ogni richiesta di risorse o ogni tot minuti o quando la cpu scende sotto una certa soglia)
    b. **Risoluzione**: può essere di tre tipi:
        i. **Prelascio**: Togliendo una risorsa allocata ad uno dei processi in deadlock;
        ii. **Rollback**: Inserire nei programmi dei *check-point* (con tutto lo stato dei processi), quando si scopre un deadlock, si ritorna ad uno dei checkpoint con conseguente rilascio delle risorse allocate da allora in poi.
        iii. **Terminazione**: termina tutto. Equivale ad un rollback iniziale.
3. Cercare di **evitare dinamicamente** le situazioni di stallo conoscendo alcune **info a priori**:
    a. Il numero massimo di risorse di ogni tipo di cui avrà bisogno durante la computazione;
    b. Un algoritmo si assicura che non ci siano mai code circolari;
    c. Stato di allocazione delle risorse allocate.

    Stato sicuro

    È uno stato in cui **è possibile allocare tutte le risorse** richieste da un processo senza che quest'ultimo comporti un deadlock con un altro processo.

    Algoritmo del banchiere ($O(n^2m)$)

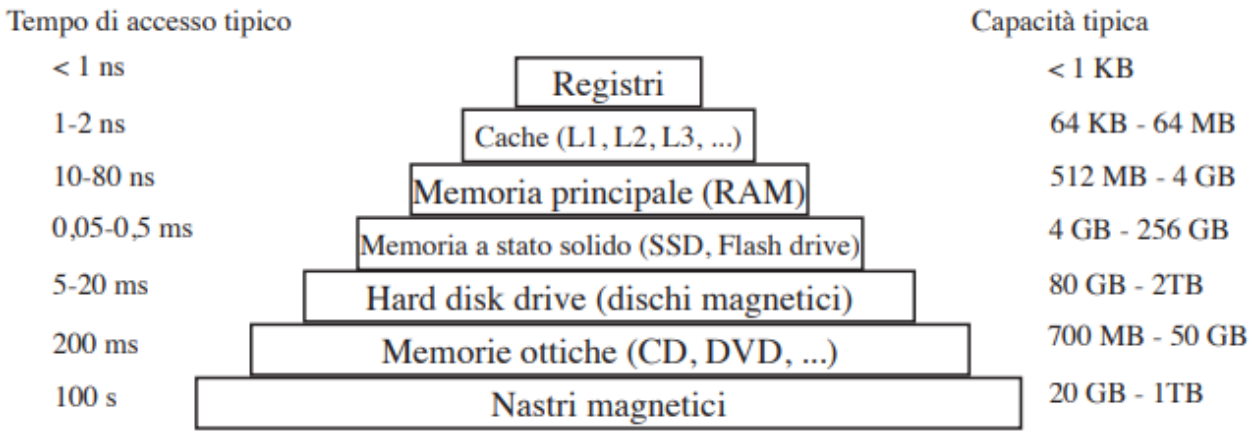
    Dove *n* è il numero di processi e *m* il numero di risorse. Ad ogni richiesta, l’algoritmo controlla se le risorse rimanenti sono sufficienti per soddisfare la massima richiesta di almeno un processo (si troverebbe in uno **stato sicuro** allocando tutte le risorse di cui ha bisogno?);

    I processi sono visti come dei **clienti** che possono richiedere credito presso la banca (fino ad un certo **limite** individuale) e le risorse allocabili sono viste come il **denaro**. È chiaro che il sistema, come la banca, non può permettere a tutti i clienti di raggiungere il loro **limite** di credito **contemporaneamente**, poiché in tal caso la banca fallirebbe (e il sistema non potrebbe allocare risorse a sufficienza, causando un deadlock).

4. **Assicurare** che il sistema non possa mai entrare **mai** in uno stato di **deadlock**
    a. Negando una delle quattro condizioni necessarie.

8. Gestione della memoria

Esiste una **gerarchia della memoria** gestita dal gestore della memoria:



Attivazione di un programma

Ogni programma, per poter essere eseguito, deve essere portato (almeno in parte) in memoria ram e posto nello **spazio indirizzi** di un processo. L'insieme dei **programmi su disco in attesa** di essere portati in memoria ram per essere eseguiti è inserito in una coda chiamata **coda di input**.

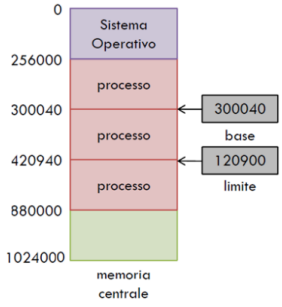
Librerie aggiuntive

Tutto ciò può essere realizzato completamente a **livello di linguaggio** o di programma senza l'intervento del sistema operativo. Tuttavia il so può fornire delle librerie collegate all'esecuzione tramite il **linking dinamico** (a.e. i .dll su windows). Nell'eseguibile si inseriscono piccole porzioni di codice, dette **stub**, che servono per localizzare la routine. Alla prima esecuzione, si carica il segmento se non è già presente in memoria, e lo stub viene rimpiazzato dall'**indirizzo della routine** e si salta alla routine stessa.

Monoprogrammazione e multiprogrammazione

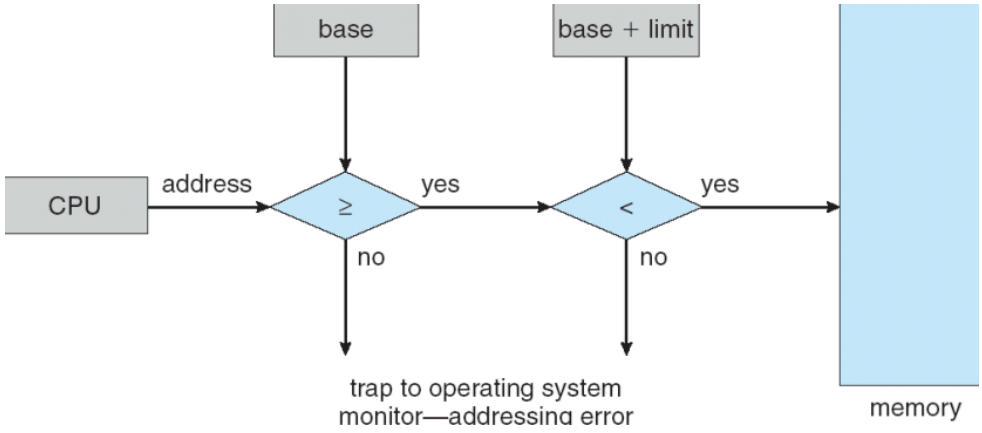
Nei sistemi **monoprogrammati**, viene eseguito **un processo** alla volta. Nei sistemi **multiprogrammati** più processi sono **contemporaneamente** pronti in ram per l'esecuzione. I processi di sistema devono coesistere tra di loro e con il so nello stesso spazio di indirizzamento fisico.
Sono quindi necessarie:

- **Condivisione della memoria**;
- **Separazione degli spazi di indirizzamento** (serve garantire che ogni processo acceda solo alla sua area). Si utilizzano due registri:
    - **Registro base:** contiene il più piccolo indirizzo fisico;
    - **Registro limite**: contiene la dimensione dell'intervallo ammesso.



Quando un **processo utente** cerca di accedere ad un indirizzo, la cpu lo **confronta** con i valori dei due registri:

- Se l'operazione è lecita, tutto prosegue;
- Se l'operazione non è lecita:
    - Il controllo passa al so (che non ha limiti)
    - Passaggio dalla modalità utente a quella kernel.



Indirizzamento

I programmi utente passano attraverso più stadi prima di essere eseguiti, e in tali stadi gli indirizzi cambiano la loro rappresentazione. Esistono **diversi spazi degli indirizzi**:

- Implementazione: indirizzi simbolici;
- Compilazione: il compilatore associa indirizzi simbolici ad indirizzi relativi (binding);
- Caricamento: gli indirizzi relativi vengono trasformati in indirizzi assoluti.

L'associazione di istruzioni e dati ad indirizzi di memoria può essere eseguita in diverse fasi:

- Fase di **compilazione**: vengono generati indirizzi di riferimento a dove il codice dovrà risiedere in memoria durante l'esecuzione;
- Fase di **caricamento**:
    - **Statico**: intero programma e tutti i suoi dati caricati in ram;
    - **Dinamico**: si carica una porzione di programma (procedura) solo quando richiamata.
- Fase di **esecuzione:**

Spazi di indirizzi fisici e logici

In un sistema dotato di memoria virtuale, il processore e i programmi si riferiscono alla ram con **indirizzi logici** (virtuali) che vengono **tradotti in indirizzi fisici** reali da una unità apposita, la **MMU** o memory management unit che in genere è incorporata nei processori. Il **programma utente** vede solamente gli indirizzi logici; non vede mai gli indirizzi reali, fisici.

Tecniche di gestione della memoria

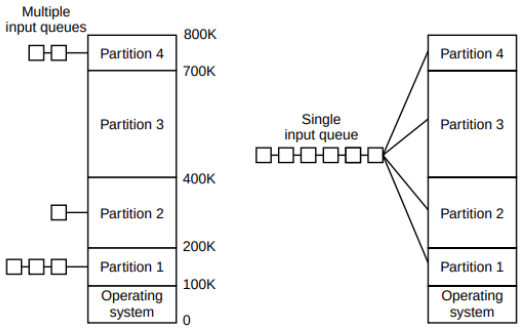
Allocazione contigua

La memoria è divisa in almeno due partizioni:

- **Sistema operativo residente**
- **Spazio per i processi utente**: formato da celle consecutive.

Partizionamento statico o fisso

La memoria disponibile è divisa in **partizioni di dimensione** (uguale o non) **fissa**. Nel momento in cui arriva un processo, il so decide una partizione tra quelle libere e la assegna completamente (porta a **frammentazione interna**, parte della partizione infatti non viene usata). Ogni partizione ha la propria **coda di input**.



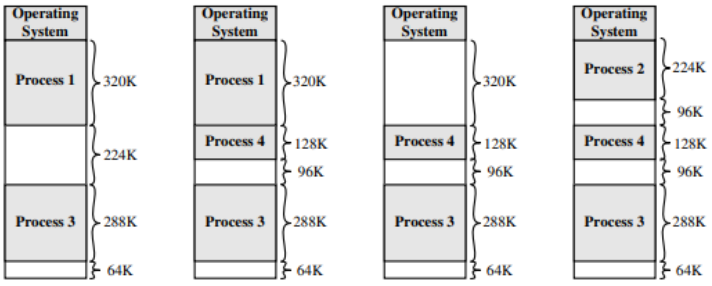
Come soddisfare una richiesta di dimensione n?

- **First-fit**: per ogni buco, il primo che ci entra;
- **Best-fit**: : il più grande che ci entra. Penalizza i job piccoli.

Partizionamento dinamico

La dimensione delle partizioni viene decisa al **runtime**. Non è soggetto a frammentazione interna, invece può portare a frammentazione esterna. La memoria risulta quindi composta da:

- **Processi;**
- **Hole**: blocchi di memoria liberi.



Non c'è frammentazione interna, ma c'è **frammentazione esterna** dato che è possibile che ci sia memoria libera sufficiente per un processo ma che non sia contigua. Tuttavia la frammentazione esterna si può ridurre con la **compattazione** (a.e. agglomerante tutte le locazioni vuote in un'unica più grande).

Come soddisfare una richiesta di dimensione n?

- **First-fit**: Alloca il primo buco sufficientemente grande
- **Next-fit**: Alloca il primo buco sufficientemente grande a partire dall'ultimo usato.
- **Best-fit**: Alloca il più piccolo buco sufficientemente grande. Deve scandire l'intera lista (a meno che non sia ordinata). Produce il più piccolo buco di scarto.
- **Worst-fit**: Alloca il più grande buco sufficientemente grande. Deve scandire l'intera lista (a meno che non sia ordinata). Produce il più grande buco di scarto

Allocazione non contigua

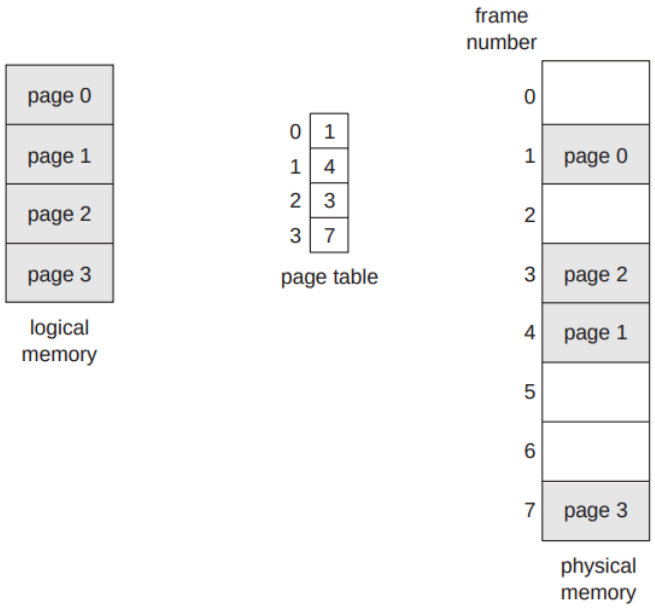
È possibile assegnare ad un programma aree di memoria separate:

Paginazione

Tecnica attraverso la quale il sistema operativo:

- **Suddivide la ram** in *frame* di dimensione fissa (una potenza di 2, tra 512B e 16MB). Il numero di page fault cala all'aumentare del numero di frame disponibili sul sistema;
- **Suddivide la memoria logica** in *pagine* della stessa dimensione;
- **Collega** gli indirizzi logici (pagine, in memoria logica) con gli indirizzi fisici (frame, in memoria ram) attraverso una **page table**.

Per eseguire un programma di *n* pagine, servono *n* frame liberi in cui caricare il programma:

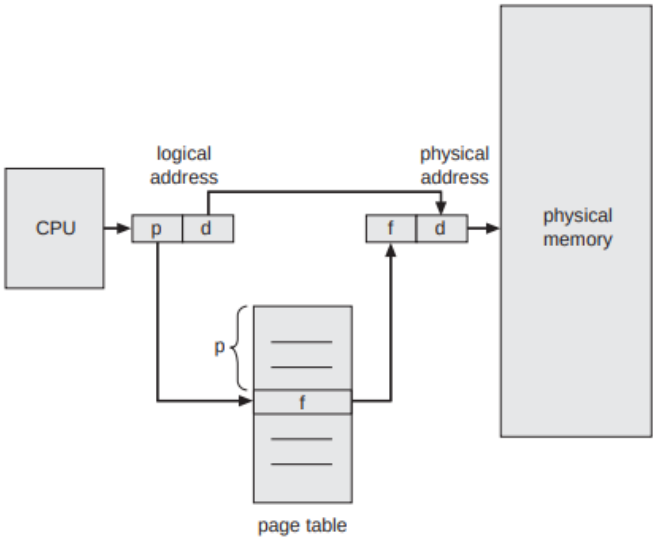


Alcune pagine di memoria possono essere disponibili per più processi (per permettere ai processi di comunicare tra di loro). Con questa tecnica **non c'è frammentazione esterna**, e la **frammentazione interna risulta ridotta**.

Schema di traduzione degli indirizzi

L'indirizzo generato dalla cpu viene diviso in:

- **Numero di pagina *p***: usato come indice in una **page table** (che contiene il numero del frame corrispondente);
- **Offset di pagina *d***: combinato con il numero del frame contenuto nella page table, fornisce l'indirizzo fisico da inviare alla memoria ram.



Protezione

La protezione della memoria è implementata associando:

- Un **bit di protezione** ad ogni frame (quindi in ram);
- Un **valid bit** collegato ad ogni entry nella page table:
  - **Valid**: legale accedere alla pagina (la pagina è nello spazio logico del processo);
  - **Invalid**: segment violation (la pagina non è nello spazio logico del processo).

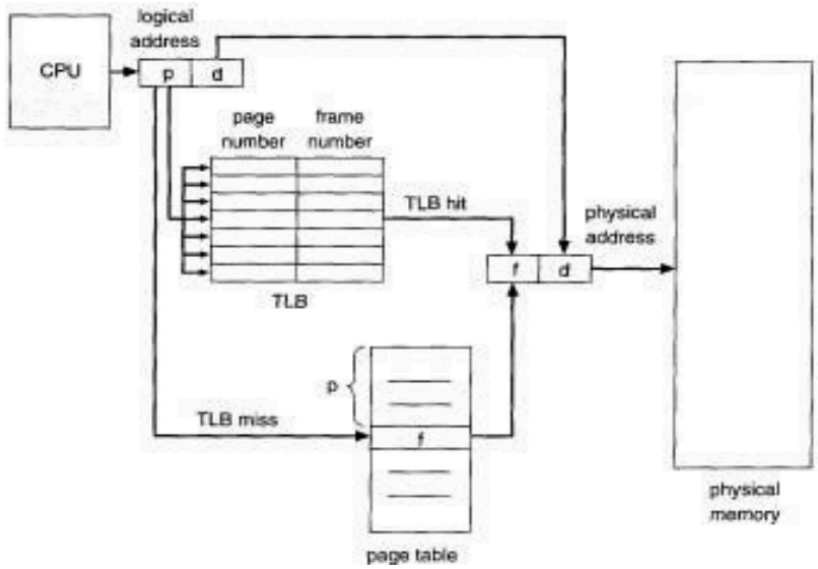
Implementazione della page table

Ci sono due modalità:

- Page table nei **registri della MMU:** costoso al context switch e impossibile se il numero di pagine è elevato;
- Page table in **memoria ram**: grande consumo di memoria, ogni accesso ai dati richiede 2 accessi alla memoria che possono essere ridotti implementando una cache dedicata (**registri associativi TLB**) per le entry della page table.

TLB ed Effective access time

Il TLB è una memoria di piccole dimensioni e molto veloce. Ogni entry del TLD consiste in due sezioni: un **tag** ed un **valore**. Tipicamente il numero di entry è basso dato che è una memoria molto costosa (da 64 a 1024).



La percentuale di volte in cui un particolare numero di pagina viene trovato nel TLB viene definito **hit ratio**. A.e un hit ratio del 80% significa che la pagina desiderata viene trovata nel TLB l'80% delle volte.

- Se la ricerca nel TLB impiega 20 ns e l'accesso in memoria ne impiega 100 ns, allora in caso di TLB hit ci si impiega 120 ns.
- Se il numero di pagina non viene trovato nel TLB (20 ns), bisogna accedere alla memoria per trovare la page table e il frame number (100 ns) e successivamente accedere al byte desiderato in memoria (100 ns) per un totale di 220 ns.

Per definire l'**effective access time**:

$EAT = \epsilon + \alpha t + (1 + \alpha)(2t)$

- Nel caso descritto sopra:
  $EAT = 20 + 0.80(100) + (1 - 0.80)(200) = 140ns$ e quindi si soffre di un rallentamento del 40% (passando dai 100 ns ai 140 ns).
- Nel caso descritto sopra ma usando un hit ratio del 98%
  $EAT = 20 + 0.98(100) + (1 - 0.98)(200) = 122ns$ e quindi si soffre di un rallentamento del 22% (passando dai 100 ns ai 122 ns).

Algoritmi per rimpiazzare una pagina

- **Firs-In-First-Out (FIFO)**: Si rimpiazza la pagina da più tempo in memoria.
- **Ottimale (OPT o MIN)**: Si rimpiazza la pagina che non verrà usata per il periodo più lungo. Algoritmo che porta al minore numero di page fault e non soffre dell'anomalia di Belady;
- **Least Recently Used (LRU)**: Si rimpiazza la pagina che da più tempo non viene usata. Può essere implementato:

- A **contatori** (molto dispendioso): la MMU ha un contatore che viene incrementato ad ogni accesso in memoria, ogni entry nella page table ha un registro in cui si copia il valore del contatore al momento in cui si fa riferimento ad una pagina. Quando si deve liberare un frame, si cerca la pagina con il registro più basso.
- A **stack** (costoso in termini di tempo): si basa su uno stack di numeri di pagina in una lista double-linked. Quando si fa riferimento ad una pagina, la si sposta in cima allo stack. Quando si deve liberare un frame, la pagina da swappare è quella sul fondo dello stack.

L'algoritmo LRU **può essere approssimato** in diversi modi:

- **Reference bit**: si associa un bit 0 ad ogni pagina, nel momento in cui si fa riferimento ad una pagina, il bit viene settato ad 1. Si rimpiazza la pagina che ha il bit 0.
- **Not frequently used (NFU)**: ad ogni pagina si associa un contatore e ad intervalli regolari si somma per ogni entry il valore del contatore a quello del reference bit.
- **Aging**: ad ogni pagina, si associa un array di bit inizialmente uguale a zero. Ad intervalli regolari, si shifta gli array di tutte le pagine immettendo i bit di riferimento, che vengono settati a 0. Si rimpiazza la pagina che ha il numero più basso nell'array.
- **Clock**: se una pagina è stata molto usata di recente,probabilmente verrà usata pesantemente anche prossimamente. Se la pagina candidata ha un reference bit
  - 0: rimpiazza;
  - 1: bit a 0, lascia la pagina in memoria e passa alla prossima pagina in memoria.

  Se i reference bit sono tutti 1, l'algoritmo diventa un FIFO.

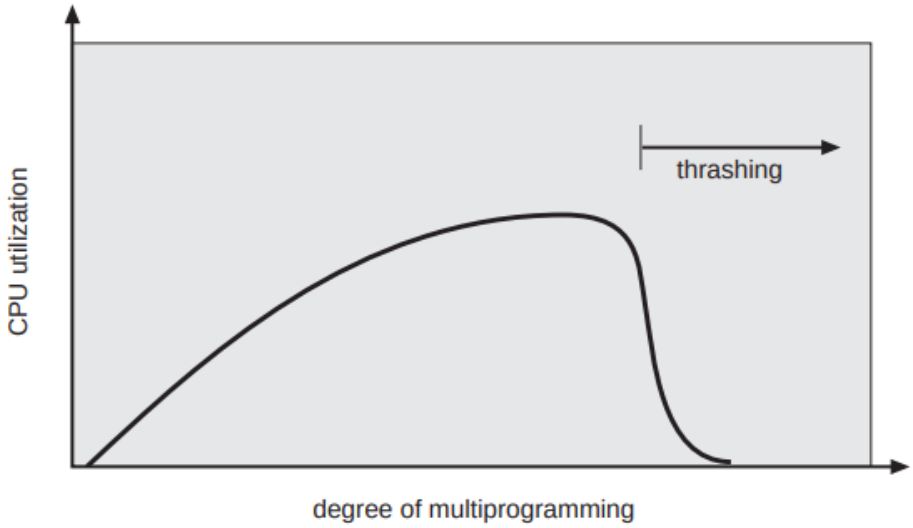
- **Clock migliorato**: usato in macOS tradizionale. Si usano due bit per ogni pagina:
  - r=0, d=0: non usata recentemente, non modificata (buona);
  - r=0, d=1: non usata recentemente, ma modificata (meno buona);
  - r=1, d=0: usata recentemente, non modificata (probabilmente verrà riusata);
  - r=1, d=1: usata recentemente, modificata (molto usata).

  Si cerca una pagina con (0,0) da sostituire. Se non la trova, cerca una pagina (0,1) azzerando il reference bit, se ancora la pagina non viene trovata, si torna all'inizio.

Trashing

Si verifica quando la memoria virtuale **scambia** rapidamente dati per dati su disco rigido. Questo può essere causato dal fatto che un processo non ha "abbastanza" pagine (a.e. ad alti livelli di **multiprogrammazione**).

Man mano che la memoria principale viene riempita, è necessario scambiare e rimuovere pagine aggiuntive dalla memoria virtuale. Lo scambio provoca un tasso molto elevato di accesso al disco rigido.



Working set per impedire il trashing

Il working set è un concetto che **definisce la quantità di memoria** richiesta da un processo in un determinato intervallo di tempo. Alla creazione di un nuovo processo, questo viene ammesso nella coda ready **solo** se ci sono frame liberi sufficienti per coprire il suo working set.

Se il totale dei **frame richiesti** è **maggiore** del numero di **frame fisici disponibili** (avverrebbe trashing), allora si **sospende** uno dei processi per liberare la sua memoria per gli altri (diminuire il grado di multiprogrammazione).

Principio di località

Una **località** è un principio di pagine che vengono utilizzate attivamente assieme dal processo. Il processo, durante l'esecuzione **migra** da una parte all'altra

Algoritmi di allocazione dei frame

Esistono diversi modi di allocare i frame quando arriva un processo:

- **Allocazione libera**: dare ad ogni processo tutti i frame che vuole;
- **Allocazione equa**: stesso numero di frame ad ogni processo;
- **Allocazione proporzionale**: dare un numero di frame in proporzione a:
    - Dimensione del processo;
    - Priorità;

Esempio: due processi da 10 e 127 pagine, su 62 frame:

$$\frac{10}{127+10} * 62 \approx 4 \text{ e } \frac{127}{127+10} * 62 \approx 57$$

Inoltre, l'allocazione **varia** al variare del **livello di multiprogrammazione**: se arriva un terzo processo da 23 frame:

$$\frac{10}{127+10+23} * 62 \approx 3 \text{ e } \frac{127}{127+10+23} * 62 \approx 49 \text{ e } \frac{23}{127+10+23} * 62 \approx 8$$

Paginazione a più livelli

Per ridurre l'occupazione della page table, si **pagina la page table** stessa. Solo le pagine effettivamente usate sono allocate in memoria RAM (la conversione dell'indirizzo logico in indirizzo fisico può necessitare di diversi accessi alla memoria).

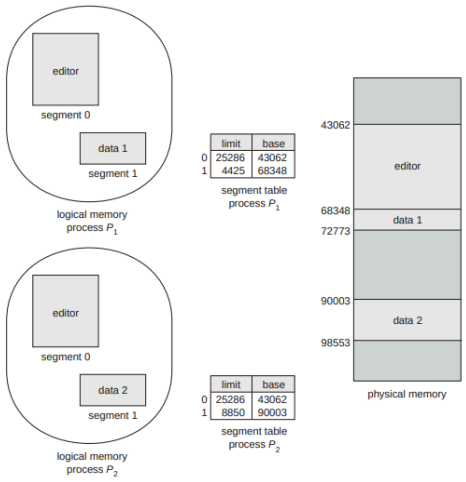
Segmentazione

Tecnica attraverso la quale un programma, prima di essere eseguito, **viene diviso** in **unità logiche** di memoria di dimensione diversa (segmenti).

Ad esempio. un programma può essere diviso in:

- Un segmento contenente le procedure chiamate più spesso;
- Un segmento per le variabili globali;
- …

Ogni segmento può cambiare la sua dimensione durante l'esecuzione (a.e. lo stack).



Protezione

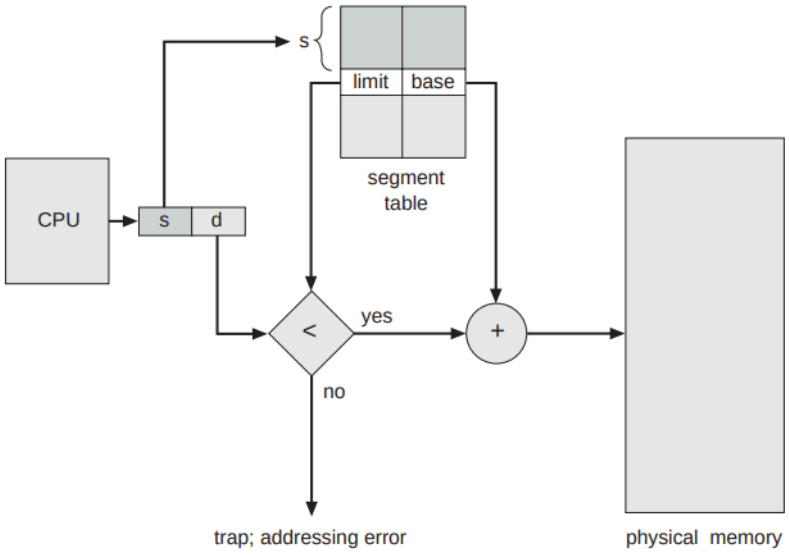
La protezione della memoria è implementata associando ad ogni entry della segment table:

- Dei **privilegi** di read/write/execute;
- Un **valid bit**:

Schema di traduzione degli indirizzi

L'**indirizzo logico** consiste in una coppia *segment-number, offset* che viene mappato nel corrispettivo indirizzo fisico tramite una **segment table** composta da varie **entry** a loro volta composte da:

- **Base**: indirizzo fisico di inizio del segmento;
- **Limit**: lunghezza del segmento.



Segmentazione con paginazione: MULTICS

Il MULTICS ha risolto il problema della frammentazione esterna paginando i segmenti. A differenza della pura segmentazione, nella **segment table** ci sono gli **indirizzi base delle page table** dei segmenti.

Swapping

Lo swap viene utilizzato per **liberare memoria RAM**: Un processo in esecuzione può essere **temporaneamente rimosso** dalla memoria e caricato in una memoria secondaria (detta backing store o swap area); in seguito può essere riportato in memoria per continuare l'esecuzione. Questa porzione contiene i dati che hanno **minore probabilità** di essere richiesti nel futuro.

Overlay

È una tecnica che permette di **suddividere un programma** di grandi dimensioni in parti abbastanza piccole da essere contenute per intero nella memoria centrale.

Creazione dei processi

Copy on Write (COW)

È una tecnica di ottimizzazione informatica mirante alla **riduzione delle operazioni di duplicazione** delle risorse del sistema (generalmente strutture dati o aree di memoria) attraverso l'eliminazione delle copie non necessarie.

Il sistema può limitarsi a **simulare** l'operazione di duplicazione, mantenendo l'esistenza di un'unica copia e gestendo attraverso di essa tutte le operazioni di lettura destinate ad una qualsiasi delle due. La vera e

propria duplicazione della risorsa può essere posticipata fino al momento in cui l'esistenza di due risorse indipendenti si rende realmente necessaria (modifica dello stato).

Il Copy-on-Write permette al padre e al figlio di **condividere** inizialmente le stesse pagine in memoria.

Memory-Mapped I/O

L'**I/O mappato in memoria** (MMIO) è un metodo per eseguire input/output (I/O) tra la cpu e i dispositivi periferici in un computer.

L'I/O mappato in memoria utilizza lo **stesso spazio di indirizzi** per indirizzare sia la memoria che i dispositivi I/O. Quindi un indirizzo di memoria può fare riferimento a una parte della RAM fisica o invece alla memoria e ai registri del dispositivo I/O.

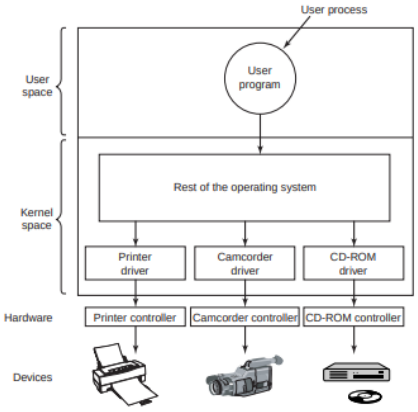
9. Dispositivi I/O

È la parte più difficile da integrare in un sistema operativo, ce ne sono tante e sono tutte diverse. Per far sì che un sistema operativo tratti le periferiche allo stesso modo nonostante le differenze hardware la risposta è l'**astrazione**, l'**incapsulamento** e la **stratificazione** del software.

I driver

Si cerca quindi di **isolare le differenze** tra i dispositivi I/O individuandone pochi tipi generali.

A ciascuno di questi tipi si accede tramite un insieme standard di funzioni che ne costituiscono l' interfaccia, mentre le differenze sono incapsulate in moduli del kernel detti **driver** del dispositivo.



Internamente i driver sono **specifici** per una particolare periferica, mentre all'esterno comunicano con perfetta **integrazione** all'interfaccia standard. I driver possono essere **caricati al boot** oppure **on demand** (come all'inserimento di una chiavetta).

Tuttavia se i driver contengono del **codice malevolo** o non sono ben costruiti, possono portare a problemi di sicurezza e/o compatibilità dato che i driver vengono eseguiti in **spazio kernel**. (per questo Apple non permette di alterare i propri dispositivi o di usarne di terzi).

Classificazioni dei dispositivi di I/O

Grossolanamente esistono tre categorie:

- **Human readable**: orientate all'interazione con l'utente (a.e terminale, mouse, tastiera);
- **Machine readable**: periferiche che operano dentro la macchina (a.e mem di massa, dischi);
- **Comunicazione**: servono per lo scambio di dati tra calcolatori connessi in rete (a.e modem).

Classificazione dal punto di vista informatico:

- **Dispositivi a blocchi o ad accesso casuale**: permettono l'accesso ai blocchi (a.e regioni di memoria) ed è di dimensione costante (a.e dischi);
- **Dispositivi a carattere**: forniscono istante per istante il byte da inviare al calcolatore. Non permettono l'indirizzamento diretto (a.e tastiera, mouse, scheda di rete);
- **Altri dispositivi** che esulano da queste caratteristiche (a.e nastri, timer che emette un clock).

Comunicazione CPU-I/O

- **Approccio con insieme di istruzioni dedicato**: si fa uso di uno spazio separato, questo spazio è molto più ristretto rispetto alla ram, ed è organizzato con istruzioni privilegiate (per fare i/o bisogna sempre passare per il kernel);
- **Approccio mappato in memoria**: invece di avere istruzioni speciali in uno spazio in memoria dedicato, ho una parte della ram marcata per mappare le periferiche i/o. In questo modo è possibile leggere e scrivere in memoria senza passare sempre dal kernel;
- **Approccio ibrido**: fa uso di entrambe le soluzioni (tutte le cpu adottano questo sistema).

Modi di I/O

	Senza interrupt	Con interrupt
trasferimento attraverso il processore	Programmed I/O	Interrupt-driven I/O
trasferimento diretto I/O-memoria		DMA, DVMA

Programmed I/O

(senza interrupt): la cpu manda un comando di i/o e poi attende che l'operazione sia finita, testando lo stato del dispositivo con un loop busy-wait (**polling**). Efficiente solo se la velocità del dispositivo è paragonabile a quella della cpu.

Interrupt-driven I/O

(con interrupt): Il sistema è guidato dagli interrupt. Il processore manda un comando di I/O e il processo viene sospeso. Al termine il processore riprende a svolgere il processo. Se il numero di interrupt è molto alto, si passa in una situazione in cui ci sono poche computazioni e interrupt continui che lasciano poco tempo per elaborazioni utili, la macchina lagga.

Interrupt-driven I/O Con DMA:

I trasferimenti che coinvolgono grosse entità di dati non sono efficienti se effettuati un singolo bit alla volta come abbiamo visto finora, ma è meglio affidarli a processori specifici detti controller di accesso diretto alla memoria (DMA). Il controller DMA opera dunque direttamente sul bus di memoria **senza alcun intervento della CPU**, salvo poi lanciarle un interrupt a trasferimento compiuto. Alcune architetture usano indirizzi fisici per il DMA, altre invece accessi diretti in memoria virtuale (DVMA) che permettono ad esempio di trasferire dati tra due periferiche senza l'intervento del processore.

Vettore degli Interrupt

Un interrupt vector è un indirizzo di memoria del gestore di interrupt, oppure un indice ad un array, chiamato interrupt vector table. La tabella degli interrupt vector contiene gli **indirizzi di memoria dei gestori di interrupt**.

Nei sistemi operativi guidati dalle interrupt, l'interrupt vector è il vettore che contiene gli **indirizzi di tutte le routine di servizio**.

Ogni interrupt ha una sua priorità. Il registro di mascheramento delle interruzioni è un registro di stato, questo significa che gli si abbina 1 valore binario:

- 1: quando deve bloccare i successivi interrupt in quanto hanno una priorità inferiore rispetto all'interrupt che al momento è in exe;
- 0: quando riceve richiesta di interrupt con priorità più alta dell'interrupt che è in exe, così salva il contesto computazionale effettuando un context switching.

Interruzioni precise e imprecise

Quando arriva un interrupt, bisogna salvare lo stato della cpu (i contenuti dei registri) o su una **copia dei registri stessi** (ma in questo caso non è possibile avere interrupt annidati in quanto i registri dopo un po finiscono), o su uno **stack** (che però porta a problemi di page fault e/o overhead per la MMU e la cache).

Un'interruzione è **precisa** se:

- Il PC è salvato in un **posto noto**;
- Tutte le istruzioni **precedenti** a quella puntata dal PC sono state **eseguite completamente**;
- **Nessuna** istruzione **successiva** a quella puntata dal PC **è stata eseguita** (le relative pipeline verranno svuotate dal microcodice della cpu);
- Lo **stato dell'esecuzione** dell'istruzione puntata dal PC è **noto**.

Se una macchina ha interruzioni **imprecise**:

- È difficile riprendere esattamente l'esecuzione in hardware;
- La CPU riversa tutto lo stato interno sullo stack e lascia che sia il SO a capire cosa deve essere ancora fatto;
- Rallenta la ricezione dell'interrupt e il ripristino dell'esecuzione (grandi latenze).

Dispositivi con trasferimento dei dati a blocchi

L'interfaccia dei dispositivi con trasferimento dei dati a blocchi si occupa di tutti gli aspetti necessari per accedere ai **drive di disco** e ad altre periferiche a blocchi. Le periferiche dovrebbero prevedere comandi come *read(), write() o seek()* per l'accesso diretto.

Gli accessi stessi da parte del sistema operativo e delle applicazioni possono sfruttare le interfacce del file system o considerare le periferiche a blocchi come semplici array lineari di blocchi. In questo secondo caso si parla di **I/O grezzo**. Sono caratterizzati da blocchi (512B) che possono essere a loro volta raggruppati in ulteriori blocchi di dimensione maggiore (2-4 KB). Ogni blocco è accessibile tramite la funzione seek, leggibile tramite la read e scrivibile tramite la write;

Mappatura in memoria di dispositivi a blocchi

I file possono essere **mappati in memoria**: si fa coincidere una parte dello spazio indirizzi virtuale di un processo con il contenuto di un file. A.e. un file presente in un disco di memoria (lento), può essere mappato in una zona della ram (veloce). In termini di vantaggi, essendo in ram, è possibile applicare altre funzioni oltre a r/w/s e l'accesso risulta molto **più veloce**. Al termine dell'elaborazione il file mappato viene ricaricato sul disco modificato. A.e. quando si fa l'editing di un video, sarebbe impossibile applicare delle modifiche ad un video lasciandolo su disco, questo perché sarebbe troppo lento.

Dispositivi con trasferimento dei dati a carattere

Le periferiche con trasferimento dati a caratteri sono ad esempio **tastiere mouse e modem**, che producono dati in ingresso sotto forma di **flussi sequenziali di byte** in modo non prevedibile dalle applicazioni. Sono tipici anche di schede audio e stampanti. Questo tipo di periferiche supportano le funzioni *get* (per la lettura di un input) e *put*.

Periferiche di rete

Le prestazioni e le caratteristiche di indirizzamento di una rete informatica sono **molto diverse** da quelle delle operazioni di I/O su disco, dunque la maggior parte dei sistemi operativi fornisce interfacce I/O di rete differenti da quelle viste finora. Tra quelle più diffuse in sistemi come UNIX e Windows c'è quella del socket di rete.

Orologi e temporizzatori

Nella maggior parte dei sistemi vengono utilizzati orologi (**clock**) e temporizzatori (**timer)** per tener traccia dell'ora corrente, del tempo trascorso, o per impostare un intervallo prima di scatenare un interrupt.

I/O bloccante, non bloccante e asincrono

Gli interrupt dati da dispositivi di input/output possono essere di tre tipi:

- **Bloccante**: Le operazioni di I/O bloccanti sospendono l'esecuzione dell'applicazione, che viene spostata nella coda di quelle in attesa finché la chiamata di sistema non è stata completamente eseguita; solo a qual punto l'applicazione potrà tornare nella coda dei processi pronti;
- **Non bloccante**: Gli I/O non bloccanti non prevedono tale sospensione e sono tipici delle interfacce utente di ingresso (tastiera, mouse);
- **Asincrono**: il processo attende che l'operazione sia completata.

Sottosistema di I/O del kernel

Scheduling

Lo scheduling ha il compito di **stabilire in che ordine** le system call devono essere esaudite. Scegliere un buon algoritmo di scheduling **migliora le prestazioni** globali del sistema dato che l'ordine dettato dalle applicazioni è raramente lungimirante. La schedulazione viene implementata mantenendo una **coda di richieste** per ogni periferica e facendovi operare uno schedulatore dedicato.

Buffering

Il buffer è una **zona di memoria** in cui vengono memorizzati temporaneamente i dati **mentre vengono trasferiti** tra due periferiche o tra periferiche e applicazioni. La maggior parte dei buffer è implementata nel software, che in genere utilizza la **RAM** più veloce per archiviare i dati temporanei, a causa del tempo di accesso molto più rapido rispetto alle unità disco rigido. Si usano principalmente per tre motivi:

- **Appianano la differenza di velocità** tra mittente e destinatario di un flusso di dati;
- Fungono da **adattatori** tra periferiche con **blocchi di dimensioni diverse**;
- **Supportano la semantica della copia**, che garantisce che la versione dei dati scritta su un dispositivo sia la stessa versione presente in memoria al momento della chiamata di sistema dell'applicazione.

In casi in cui se ne faccia un **uso eccessivo**, ci potrebbe essere un **calo delle prestazioni** (avverrebbero letture e scritture ripetute le quali avrebbero un costo sia in termini di tempo che in termini di spazio).

Lo **spool** è un buffer su memoria di massa che conserva l'output per una periferica che non può accettare flussi di dati da più processi contemporaneamente, come ad esempio le stampanti.

Caching

Il caching consiste nel **mantenere una copia** dei dati più usati **in una memoria più veloce**. Una cache è sempre una copia di dati esistenti altrove. È fondamentale per aumentare le performance.

Gestione degli errori

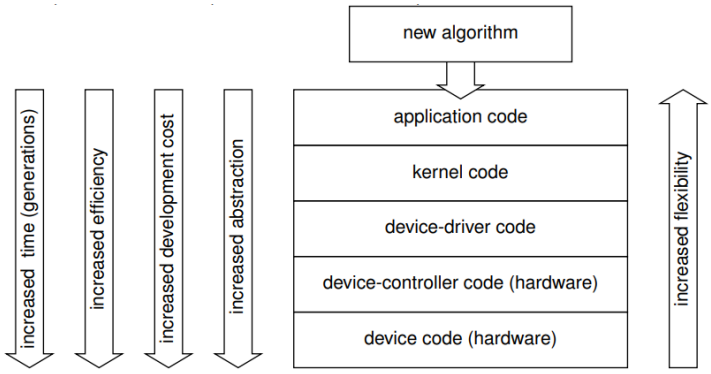
Gli errori si dividono in errori:

- Di **programmazione** (a.e. accedere ad un dispositivo non installato);
- Di **dispositivo** che possono invece essere;
    - **transitori** (es: rete sovraccarica): solitamente il S.O. può (tentare di) recuperare la situazione (es.: richiede di nuovo l'operazione di I/O);
    - **permanenti** (disco rotto).

Le chiamate di sistema segnalano un errore, quando non vanno a buon fine neanche dopo ripetuti tentativi. Spesso i dispositivi di I/O sono in grado di fornire **dettagliate spiegazioni** di cosa è successo (es: controller SCSI). Il kernel può registrare queste diagnostiche in appositi **log di sistema**.

Livelli del software I/O

Per raggiungere gli obiettivi precedenti, si **stratifica** il software di I/O, con interfacce ben chiare garantendo una maggiore modularità.



Driver delle interruzioni

Passi principali da eseguire:

1. **salvare** i registri della CPU (stato corrente),
2. **impostare un contesto** per la procedura di servizio (TLB, MMU, stack. . .),
3. **ack al controllore degli interrupt** (per avere interrupt annidati),
4. copiare la **copia dei registri nel PCB** se non già fatto,
5. eseguire la **procedura di servizio** (che accede al dispositivo),
6. **organizzare un contesto** (MMU e TLB) per il processo successivo,
7. **caricare** i registri del nuovo processo dal suo **PCB**,
8. **continuare** il processo selezionato.

Miglioramento delle performance

Per aumentare le performance di I/O, è necessario:

- **Ridurre** il numero di **context switch** (conserva lo stato, in modo da poter essere ripreso dopo).
- **Ridurre spostamenti di dati** tra dispositivi e memoria, e tra memoria e memoria (a.e. da buffer a buffer).
- **Ridurre gli interrupt** preferendo grossi trasferimenti, controller intelligenti, interrogazione ciclica (se i busy wait possono essere minimizzati).
- Usare canali di **DMA,** o bus dedicati.
- Implementare le primitive in hardware, dove possibile, per **aumentare il parallelismo**.
- **Bilanciare le performance** della CPU, memoria, bus e dispositivi di I/O: il sovraccarico di un elemento comporta l'inutilizzo degli altri.

10. Struttura dei dischi

Il disco rigido è costituito da una serie di **piatti coassiali** che ruotano intorno ad un **asse comune**. Ogni piatto è composto di **materiale plastico** ed è ricoperto su entrambe le facce da uno strato di **materiale ferromagnetico**.

I dischi sono indirizzati come dei **grandi array monodimensionali di blocchi logici**, dove il blocco logico è la più piccola unità di trasferimento con il controller.

Funzionamento: magnetizzazione e smagnetizzazione

Tale strato presenta sulla superficie milioni di areole che possono essere magnetizzate da un campo magnetico. Un'areola magnetizzata corrisponde ad un **1 logico**, un'areola smagnetizzata ad uno **0 logico**.

In prossimità di ciascuna faccia di ogni piatto si affaccia una **testina** di lettura/scrittura che può muoversi in senso radiale mentre il disco ruota a velocità costante. Il compito della testina è **leggere o scrivere** dati sulla superficie di ciascuna faccia.

- **Scrittura**: la testina emette un campo magnetico impulsivo che magnetizza l'areola;
- **Lettura**: la testina passa attraverso l'areola: se la trova magnetizzata, si produce una corrente impulsiva indotta sul circuito della testina e questo corrisponderà ad un 1, altrimenti si avrà assenza di corrente e ciò corrisponderà ad uno 0.

Composizione

Ogni faccia a sua volta è suddivisa in **tracce circolari concentriche** il cui numero dipende dalla capacità del disco.

Ogni traccia è suddivisa in blocchi di bit chiamati **settori**. Un settore contiene 512 bytes ed è individuato mediante un numero a partire da 1. Il numero di settori per traccia è tipicamente 63. Il settore è **la più piccola unità** di lettura/scrittura. Esistono due modi per individuare un settore.

- **Linear Block Addressing:** Fornendo una terna di numeri: il numero della testina, il numero di cilindro e quello del settore;
- **Cylinder Head Sector**: Mediante un solo numero per individuare un settore. In questa tecnica i settori vengono numerati utilizzando la successione 0,1,2....Così i settori della prima traccia e testina (C=0,H=0) andranno da 0 a 62, quelli con C=1, H=0 saranno 63,64,65.....125.

Tempo e modalità di accesso

Un parametro importante dell'HD è il tempo impiegato per accedere ad un determinato settore che non sia sotto la testina. Le case costruttrici forniscono il tempo d'accesso medio in lettura e in scrittura. Esso consta di due parti:

- **Seek Time** : il tempo (medio) per spostare le testine sul cilindro contenente il settore richiesto;
- **Latency Time**: intervallo di tempo tra una richiesta di informazioni dall'unità disco e la disponibilità dell'unità disco a restituire tali informazioni. $60/RPM * 500$

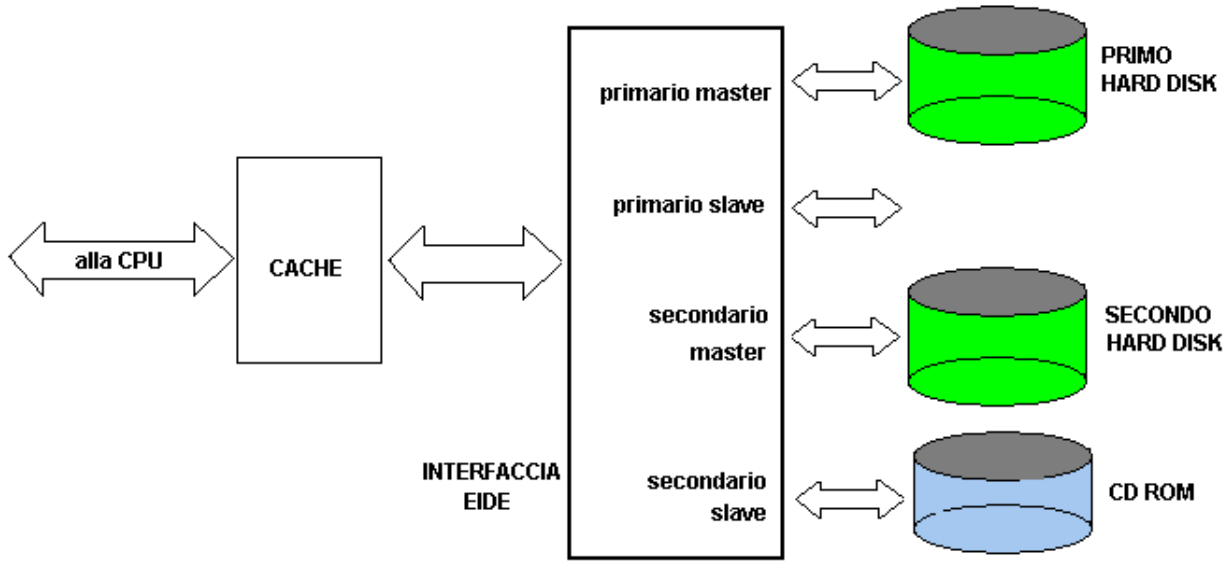
- **Transfer time**:

Esistono diversi tipi di interfacce per HD/CD/DVD: EIDE, SATA, SCSI.

Interfaccia dei dischi rigidi

L'interfaccia per i dischi rigidi comprende:

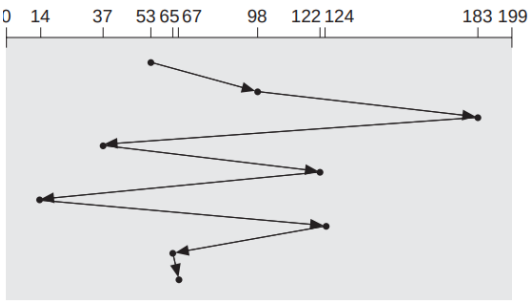
- Il **controller**: ha quattro porte alle quali si possono collegare fino a quattro unità a disco. Le quattro unità vengono indicate con: primaria master, primaria slave, secondaria master, secondaria slave.
- La **cache**: La cache serve per velocizzare le operazioni e impedire il sovraccaricamento della CPU.



Algoritmi di scheduling dei dischi

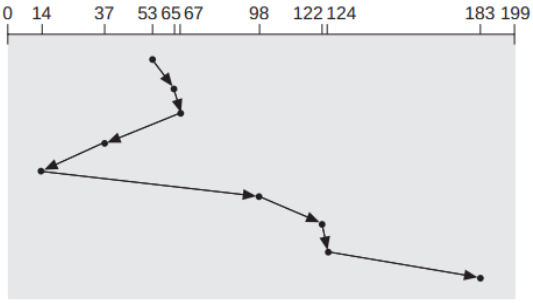
Ci sono molti algoritmi per schedulare le richieste di I/O di disco. Coda di richieste di esempio, su un range di cilindri 0–199: 98, 183, 37, 122, 14, 124, 65, 67 supponendo che la posizione iniziale della testina sia 53.

FCFS



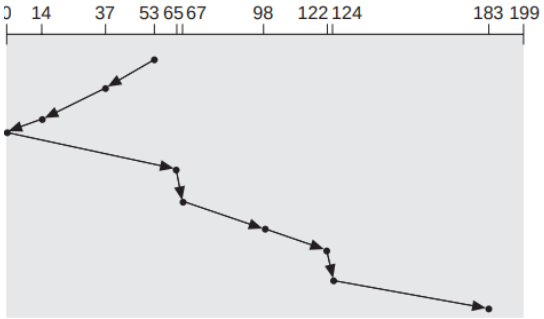
SSTF - Shortest Seek Time First

È molto comune e semplice da implementare, e abbastanza efficiente. Si seleziona la **richiesta con il minor tempo di seek** dalla posizione corrente. SSTF è una forma di scheduling SJF; può causare **starvation**. Sulla coda di esempio: distanza totale di 236 cilindri (36,875% di FCFS).



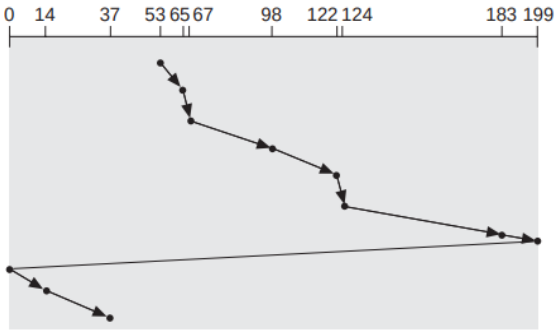
SCAN (o “dell’ascensore”)

Il braccio **scandisce l’intera superficie** del disco, da un estremo all’altro, **servendo** le richieste **man mano**. Agli estremi si inverte la direzione. È ottimo per sistemi con un grande carico di I/O con dischi (si **evita starvation**). Sulla coda di esempio: distanza totale di 236 cilindri.



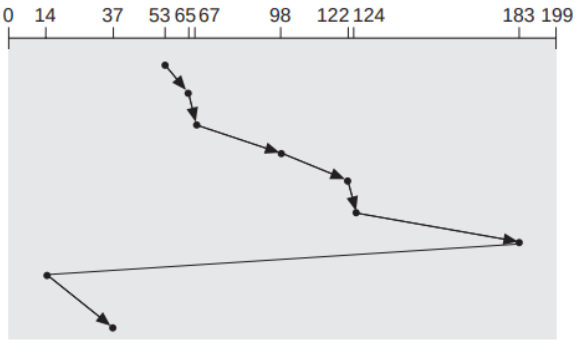
C-SCAN

Garantisce un tempo di attesa più uniforme e equo di SCAN. Tratta i cilindri come in **lista circolare**, e, arrivato alla fine, **ritorna all'inizio** del disco **senza servire** niente durante il rientro. È ottimo per sistemi con un grande carico di I/O con dischi (si **evita starvation**).



C-LOOK

È un miglioramento del C-SCAN dove **il braccio** si sposta solo **fino alla richiesta più estrema**, non fino alla fine del disco, e poi inverte direzione immediatamente.



Gestione dell' area di swap

L'area di swap è parte di disco usata dal gestore della memoria come **estensione della memoria principale**. Gestione dell'area di swap

- **4.3BSD**: alloca lo spazio appena parte il processo per i segmenti text e data. Per lo stack, lo spazio viene allocato man mano che cresce.
- **Solaris 2**: si alloca una pagina sullo stack solo quando si deve fare un page-out, non alla creazione della pagina virtuale.
- **Windows 2000**: Viene allocato spazio sul file di swap per ogni pagina virtuale non corrispondente a nessun file sul file system (es: DLL).

Tecnologia RAID

Acronimo di "Redundant Array of Independent Disks" ovvero insieme ridondante di dischi indipendenti, è una **tecnica di installazione** raggruppata **di diversi dischi rigidi** in un computer che fa sì che gli stessi appaiano e siano utilizzabili come se fossero **un unico volume** di memorizzazione. Scopi del RAID sono:

- aumentare le **performance**;
- rendere il sistema **resiliente alla perdita** di uno o più dischi;
- poterli **rimpiazzare senza interrompere** il servizio.

È una tecnica impiegata nei **server** o nelle **workstation** dove sono richiesti grandi volumi o elevate prestazioni di immagazzinamento di dati. Il RAID si trova comunemente anche nei NAS e, sempre, nei sistemi di storage per architetture blade.

RAID 0 (minimo 2 dischi)

I dati vengono suddivisi in blocchi chiamati **stripe** ed ognuno di questi viene memorizzato in un disco diverso (a.e. Disco 1 contiene i blocchi A1, A3, A5, mentre il disco 2 contiene i blocchi A2, A4, A6). La **performance dell'I/O aumenta** notevolmente dato che il carico di lavoro viene suddiviso fra più dischi.

RAID 1 (minimo 2 dischi)

I dati vengono **duplicati** su due o più dischi diversi. In questo modo anche se se ne guasta uno, i dati rimangono presenti sull'altro. La **performance della lettura dei dati raddoppia**, mentre quella della **scrittura rimane uguale** al caso di un singolo disco. Aspetto negativo: **spreco di spazio**.

RAID 2 (Hamming code ECC: 7 dischi)

Opera a livello di bit utilizzando i codici di Hamming (7,4):

- 4 bit di dati ognuno su uno dei 4 dischi dati
- 3 bit di parità ognuno su un disco di parità.

Corregge errori dovuti all'inversione di un singolo bit e rilevare errori dovuti all'inversione di due bit.

RAID 3 (minimo 3 dischi)

I byte di parità sono registrati su un disco apposito. **Prestazioni molto elevate in lettura/scrittura**. **Ottima tolleranza ai guasti**. In generale si può evadere una singola richiesta per volta.

RAID 4 (minimo 3 dischi)

I dati sono suddivisi in blocchi: ogni blocco viene memorizzato su un disco nell'array (i blocchi di parità sono registrati su un disco apposito). Prestazioni molto elevate in lettura. Ottima tolleranza ai guasti. Il disco di parità è un "collo di bottiglia"

RAID 5 (minimo 3 dischi)

Il principio di funzionamento è quello del RAID 4, ma i blocchi di parità sono memorizzati in modo distribuito sui dischi dell'array. Prestazioni migliorate in scrittura. Il sistema è complesso e costoso da realizzare.

RAID 6 (minimo 3 dischi)

Il principio di funzionamento è quello del RAID 5, ma vengono utilizzati due tipi di controllo d'errori. Grande ridondanza e sicurezza dei dati. In caso di guasto la ricostruzione dei dati è molto lenta a causa del doppio controllo ECC.

RAID composti da più livelli

- RAID 10: un insieme di dischi in mirror (RAID 1) vengono suddivisi in "strisce" in un secondo insieme di dischi (RAID 0).
- RAID 0+1: le immagini di un insieme di dischi a livello 0 vengono replicate in un insieme di dischi a livello 1.

11.File System

Un filesystem è formato dall'**insieme dei file** e delle **directory** e dalla loro organizzazione, ovvero è il meccanismo che **collega** la struttura logica, cioè l'albero delle directory, con quella fisica, cioè i settori sul disco e la metodologia di accesso agli stessi.

I file system possono essere rappresentati:

- **Graficamente** (GUI) tramite file browser dove è generalmente utilizzata la metafora delle cartelle che contengono documenti (i file) ed altre sottocartelle.
- **Testualmente** (tramite shell testuale) tramite il comando *ls* in linux.

I dispositivi di archiviazione, ad esempio i dischi fissi, si presentano al sistema operativo come **array di blocchi** di **dimensione fissa**, tipicamente di 512 byte l'uno. Le operazioni disponibili sono la **lettura** e la **scrittura** di un blocco arbitrario, o talvolta di un insieme di blocchi. Basandosi su questo servizio, il file system rende le risorse di memorizzazione di massa facilmente utilizzabili dagli utenti.

File system logico e fisico

È necessario distinguere tra la struttura logica e quella fisica delle informazioni memorizzate:

- **la struttura fisica** è il modo in cui le informazioni sono scritte fisicamente sulla memoria di massa in modo tale che il sistema possa ritrovarle per poterle leggere e/o modificare;
- **la struttura logica** è il modo in cui l'organizzazione delle informazioni è presentata all'utente.

Attributi dei file (metadata)

Queste informazioni sono solitamente mantenute in apposite strutture (directory) residenti in memoria secondaria e sono:

- **Nome** identificatore del file. L'unica informazione umanamente leggibile. Le regole per denominare i file sono fissate dal file system
- **Tipo** nei sistemi che supportano più tipi di file. Può far parte del nome.
- **Locazione** puntatore alla posizione del file sui dispositivi di memorizzazione.
- **Dimensioni** attuali, ed eventualmente massima consentita.
- **Protezioni** controllano chi può leggere, modificare, creare, eseguire il file.
- **Identificatori dell'utente** che ha creato/possiede il file.
- **Varie date e timestamp** di creazione, modifica, aggiornamento info…

Struttura di un file

A seconda del tipo, i file possono avere struttura:

- **nessuna**: sequenza di parole, byte;
- **sequenza di record**: linee, blocchi di lunghezza fissa/variabile;
- **strutture più complesse**: documenti formattati, archivi (ad albero, con chiavi, …), eseguibili rilocabili (ELF, COFF).

I file strutturati possono essere implementati con quelli non strutturati, inserendo appropriati caratteri di controllo.

La struttura viene imposta da:

- **sistema operativo**: specificato il tipo, viene imposta la struttura e modalità di accesso. Spesso imporre questa struttura porta a conseguenze spiacevoli (a.e. Durante la scrittura di un programma in c, il so permette di compilare il programma. Ma se con l'editor fai una copia di backup, la copia di backup avrà un estensione diversa dal .c e non è più compilabile)
- **utente:** tipo e struttura sono delegati al programma, il sistema operativo implementa solo file non strutturati. Più flessibile.

Operazione sui file

- **Creazione**: allocazione dello spazio sul dispositivo, e collegamento di tale spazio al file system;
- **Cancellazione**: staccare il file dal file system e deallocare lo spazio assegnato al file;
- **Apertura**: copiare alcuni metadati dal disco nella memoria principale, e accedere;
- **Chiusura**: deallocare le strutture allocate nell'apertura;
- **Lettura**: dato un file e un puntatore di posizione, i dati da leggere vengono trasferiti dal media in un buffer in memoria;
- **Scrittura**: dato un file e un puntatore di posizione, i dati da scrivere vengono trasferiti sul media;
- **Append**: versione particolare di scrittura che aggiunge i dati alla fine;
- **Riposizionamento** (seek): per saltare da un punto all'altro di un file;
- **Troncamento**: azzerare la lunghezza di un file, mantenendo tutti gli altri attributi;
- **Lettura dei metadati**: leggere le informazioni come nome, timestamp, ecc.;
- **Scrittura dei metadati**: modificare informazioni come nome, timestamp, protezione, ecc.

Tabella dei file aperti

Per **evitare di accedere continuamente** alle directory, si mantiene in memoria una **tabella dei file aperti** dove ogni elemento descrive un file aperto (file control block). A ciascun file aperto si associa

- **Puntatore al file**: posizione raggiunta durante la lettura/scrittura.
- **Contatore dei file aperti**: quanti processi stanno utilizzando il file.
- **Posizione sul disco**.

Meccanismo di funzionamento

- Alla prima apertura del file, si caricano in memoria i metadati relativi al file aperto;
- Successivamente, ogni operazione viene effettuata riferendosi al file control block in memoria;
- Alla chiusura del file, ogni informazione viene copiata in memoria e il blocco viene deallocato.

Tutto ciò è però soggetto a **problemi di affidabilità** (a.e. Manca la corrente).

Metodo di accesso sequenziale

Un **puntatore** mantiene la **posizione corrente** di lettura/scrittura. Si può accedere **solo progressivamente** (come se si trattasse di un dispositivo a carattere), o riportare il puntatore all'inizio del file.

Operazioni: *read next, write next, reset*. Si utilizza su periferiche a nastri ma va bene anche per file salvati su dischi magnetici.

Metodo di accesso diretto

Il puntatore può essere spostato in **qualunque punto** del file. L'accesso sequenziale viene simulato con l'accesso diretto. Usato per i file residenti su device a blocchi (a.e. dischi).

Operazioni: *read n, write n, seek n* (spostarsi senza fare operazioni r/w)*, read next, write next, rewrite n* dove n = posizione relativa a quella attuale.

Metodo di accesso indicizzato

Prevede due file:

- Il **file vero e proprio**, spesso di grosse dimensione;
- Un **file che contiene solo parte** dei dati, e puntatori ai blocchi (record) del vero file.

L'idea è che i dati siano ordinati secondo un certo ordine. Se si cerca un file preciso, invece di cercarlo tra tutti i file esistenti, si preferisce cercarlo in un secondo file più piccolo ed indicizzato cercando la posizione del dato (ad esempio la **rubrica dei cellulari** permette di saltare direttamente ai cognomi che iniziano con una determinata lettera).

Per questo motivo nei database si creano queste strutture ad indice.

File mappati in memoria

Nel caso di accesso ad un file con grandi dimensioni, risulta più economico **mapparlo completamente** in ram un'unica volta, modificarlo da li e smapparlo alla fine anziché fare ripetuti accessi alla memoria secondaria (lenta). Il cosiddetto accesso dei file mappati in memoria sfrutta infatti un'**unica system call** per copiare tutto in ram. Avendo poche system call, il sistema risulta meno caricato e più leggero.

Directory

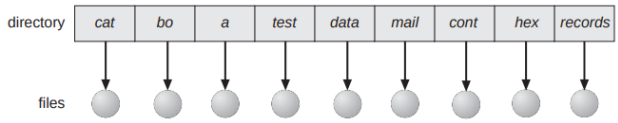
Una directory è una **collezione di nodi** contenente informazioni sui file (**metadati**). Sia la directory che i file risiedono su disco.

Servono per rendere efficiente ed organizzato i file su disco.

Operazioni su una directory: *ricerca di un file, creazione di un file, cancellazione di un file, listing, rinomina di un file, navigazione del file system.*

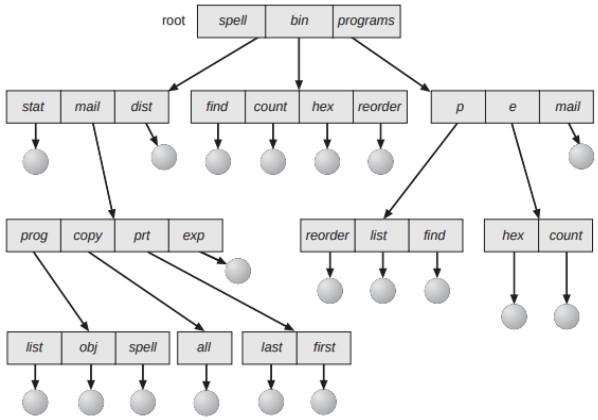
Directory flat

Una sola directory per tutti gli utenti, dove c'è un **unico spazio per tutti** i tipi di file. Ciò porta a problemi vari (mal organizzazione, impossibilità di avere due file con lo stesso nome).



Directory ad albero

Dove si ha **una root**, diversi **nodi** e diverse **foglie** (file).

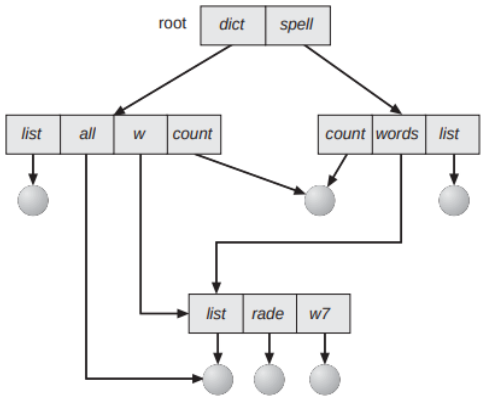


Struttura efficiente per il raggruppamento dei dati (si trovano diversi algoritmi per fare diverse operazioni con gli alberi). Nasce il concetto di **working directory:** cd /home/ger/C e il relativo concetto di **path** (assoluto o relativo).

Directory a grafo aciclico (DAG)

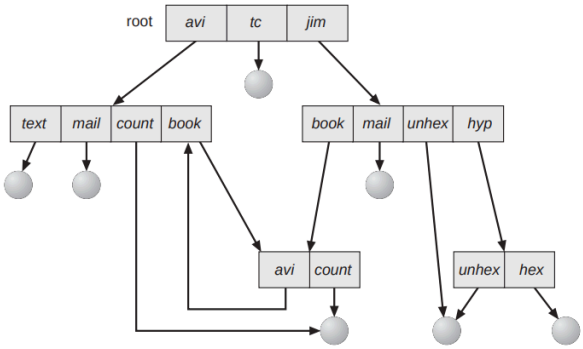
Possibilità di **condivisione dei file**: nascono i **link** o **puntatori** che permettono la condivisione dello stesso file tra due cartelle. È importante garantire l'**assenza di cicli**(i cicli portano loop).

Nasce il problema dei **puntatori sospesi** : quando viene eliminato un file rimangono i puntatori che puntano al nulla (soluzioni: puntatori all'indietro, puntatori a daisy chain e/o contatori di puntatori per ogni file).



Directory a grafo

Permette di avere dei cicli anche se risultano problematici per la **vista** e per la **cancellazione** (si creano frammenti staccati, chiamati garbage. È necessario un garbage collector per eliminarli e risulta essere un'operazione molto costosa).



Unix permette i **link hard a solo le foglie**, in questo modo non ci saranno loop. Un'altra soluzione è che ogni volta che un link viene aggiunto, si verifica l'assenza di cicli (algoritmi costosi).

Protezione dei file system

È importante avere dei meccanismi di protezione in ambienti **multi user** dove si vuole condividere file. creatore o possessore (non sempre coincidono) deve essere in grado di controllare i permessi degli utenti tramite una **matrice di accesso** dove, per ogni coppia (processo,oggetto), associa le **operazioni permesse:**

object / domain	F_1	F_2	F_3	printer
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

Esistono tre **modi di accesso**: **r**ead, **w**rite ed **e**xecute. Per ogni file, esistono quindi tre **classi di utenti**:

	G	R	W	X
Owner access	7	1	1	1
Groups access	6	1	1	0
Public access	1	0	0	1

Modi di accesso e gruppi in UNIX

Per limitare l'accesso ad un gruppo di utenti, si chiede al sistemista di creare un gruppo apposito (G) e di aggiungervi gli utenti:

- Si assegna il modo di accesso al file o directory: $ **chmod 7**61 *game*
- Si assegna il gruppo al file: $ **chgrp** G *game*

In UNIX, il dominio di protezione di un processo viene **ereditato** dai suoi figli, e viene impostato al login (così tutti i processi di un utente girano con il suo UID (id utente) e GID (id gruppo)).

Concedere permessi temporanei

Effective UID e GID (EUID, EGID): due proprietà extra di tutti i processi (stanno nella U-structure e normalmente EUID=UID e EGID=GID). Tutti i controlli vengono fatti rispetto a EUID e EGID. L'utente root può cambiare questi parametri con le system call *setuid(2), setgid(2), seteuid(2), setegid(2)*. l'Effective UID e GID di un processo possono essere cambiati per la durata della sua esecuzione attraverso i bit **setuid** e **setgid**:

- Se setuid bit è attivo, l'EUID di un processo che esegue tale programma diventa lo stesso del possessore del file;
- Se setgid bit è attivo, l'EGID di un processo che esegue tale programma diventa lo stesso del possessore del file.

I real UID e GID rimangono inalterati.

Mounting dei file system

Ogni file system fisico, prima di essere utilizzabile, deve essere **montato** nel file system logico. Il montaggio può avvenire:

- **al boot**, secondo regole implicite o configurabili;
- **dinamicamente**: supporti rimovibili, remoti, ...

Il punto di montaggio può essere

- **fissato** (D:, C:, . . . sotto Windows, sulla scrivania sotto MacOS);

- **configurabile** in qualsiasi punto del file system logico (Unix)

Prima di spegnere o rimuovere il media, il file system deve essere smontato (pena gravi inconsistenze!).

Allocazione

Allocazione contigua

Ogni file occupa un insieme di blocchi contigui sul disco. È una soluzione **semplice** e **facile da implementare** ma porta a **frammentazione esterna** e **interna** nel caso in cui i file devono allocare tutto lo spazio che gli può servire a priori. Altro punto a sfavore è che i file non possono crescere a meno di deframmentazione.

Allocazione concatenata

Ogni file è una **linked list** di blocchi, che possono essere sparpagliati ovunque sul disco. È una soluzione **semplice** che **non porta frammentazione esterna**. Tuttavia **non supporta l'accesso diretto** (seek).

Allocazione indicizzata

Si mantengono tutti i **puntatori ai blocchi** di un file in una **tabella indice**. Supporta l'**accesso random** e **l'allocazione dinamica senza frammentazione esterna**.

Inodes di Unix

Un file in Unix è **rappresentato da un inode** (nodo indice): gli inode sono allocati in numero finito alla creazione del file system. Ogni inode contiene:

- **Modo:** bit di accesso, di tipo e speciali del file;
- **UID e GID** del possessore;
- **Dimensione** del file in byte;
- **Timestamp** di ultimo accesso, di ultima modifica, di ultimo cambiamento dell'inode;
- **Numero di link hard** che puntano a questo inode;
- **Blocchi diretti**: puntatori ai primi 12 blocchi del file;
- **Primo indiretto**: indirizzo del blocco indice dei primi indiretti;
- **Secondo indiretto**: indirizzo del blocco indice dei secondi indiretti;
- **Terzo indiretto**: indirizzo del blocco indice dei terzi indiretti (mai usato!).

Gestione dello spazio libero

I blocchi non utilizzati sono indicati in una **lista di blocchi liberi**

- **Vettore di bit** (**bit map**) dove ogni blocco ha 1 bit:
  - **0**: occupato;
  - **1**: libero;
- **Linked list (free list)**;

Affidabilità dei dati

I crash di sistema possono essere causa di perdita di informazioni in cache **non ancora trasferite** al supporto magnetico. Esistono due tipi di affidabilità:

- **Affidabilità dei dati**: avere la certezza che i dati salvati possano venir recuperati.
- **Affidabilità dei metadati**: garantire che i metadati non vadano perduti/alterati

Perdere dei dati è costoso; perdere dei metadati è critico: può comportare la perdita della consistenza del file system (spesso irreparabile e molto costoso).

Possibili soluzioni per aumentare l'affidabilità dei dati

- Aumentare l'**affidabilità dei dispositivi** (RAID).
- **Backup** dei dati dal disco ad altro supporto
    - **dump fisico**: direttamente i blocchi del file system (veloce, ma difficilmente incrementale e non selettivo)
    - **dump logico**: porzioni del virtual file system (più selettivo, ma a volte troppo astratto (a.e. In casi con link, file con buchi...)).

Consistenza del file system

Alcuni blocchi contengono informazioni critiche sul file system (specialmente quelli contenenti metadati). Tali blocchi, in seguito ad un **crash**, possono contenere **informazioni sbagliate**. Per risolvere il problema della consistenza:

- **curare le inconsistenze** dopo che si sono verificate, con programmi di controllo della consistenza (scandisk, fsck): usano la ridondanza dei metadati, cercando di risolvere le inconsistenze. Lenti, e non sempre funzionano.
- **prevenire le inconsistenze**: i journalled file system (a.e. NTFS).

Journaled File System

Il **journaling** è una tecnica utilizzata da molti file system moderni per **preservare l'integrità dei dati** da eventuali cadute di tensione. Quando un applicativo invia dei dati al file system per memorizzarli su disco, questo **prima memorizza le operazioni** che intende fare su un file di **log** e in seguito provvede a effettuare le scritture sul disco rigido, quindi registra sul file di log le operazioni che sono state effettuate.

In caso di caduta di tensione durante la scrittura del disco rigido, al riavvio del sistema operativo il file system non dovrà far altro che **analizzare il file di log** per determinare quali sono le operazioni che **non sono state terminate** e quindi sarà in grado di correggere gli errori presenti nella struttura del file system.

Poiché nel file di log vengono memorizzate solo le informazioni che riguardano la struttura del disco (metadati), un'eventuale caduta di tensione elimina i dati che si stavano salvando, ma **non rende incoerente il file system.**

I più diffusi file system dotati di journaling sono: NTFS, ext3, ext4, ReiserFS, XFS.

12. Sicurezza Sistemi Operativi

Il problema della sicurezza

Bisogna fare in modo di proteggere il sistema da:

- Accessi non autorizzati;
- Modifica o cancellazione di dati fraudolenta;
- Perdita di dati o induzione di inconsistenze accidentali.

Autenticazione

L'identità degli utenti (con relativi **permessi** e **privilegi**) viene determinata per mezzo di meccanismi di **login** che utilizzano **password**. Quando un nuovo utente si connette ad un sistema di calcolo, quest'ultimo deve riconoscerlo. È possibile e consigliato implementare un **log di login** dove si raccolgono i dati inseriti in caso di mancato accesso per credenziali errate.

Un sistema operativo non dovrebbe mai memorizzare le password in chiaro, ma "cifrate" attraverso una **one-way function** (es.: MD5, SHA).

MD5

Questa funzione prende in input una **stringa** di lunghezza arbitraria e ne produce in output un'altra a 128 bit. Il processo avviene molto velocemente e l'output (noto anche come "MD5 Checksum" o "MD5 Hash") restituito è tale per cui è altamente improbabile ottenere con due diverse stringhe in input uno stesso valore hash in output.

Algoritmo di Morris e Thompson

In più, si può utilizzare il meccanismo di **Morris e Thompson**:

- Ad ogni password viene associata (in testa o in coda) una sequenza casuale di n-bit (**salt**, che aumenta lo spazio di ricerca) che viene aggiornato ad ogni cambiamento di password;
- La password ed il numero casuale **vengono concatenati** e poi **cifrati** con la **funzione one-way**;
- In ogni riga del file degli account vengono memorizzati il numero in chiaro ed il risultato della cifratura;

Sistemi One-time Password (OTP)

Meccanismo ideato da Leslie Lamport (1981) che consente ad un utente di collegarsi **remotamente** ad un server **in modo sicuro** anche se il traffico di rete viene intercettato totalmente: si basa sul fatto che data una funzione **one-way** *f,* dato *x* è semplice calcolare *y=f(x)* ma dato *y*, è impossibile risalire ad x.

- l'utente sceglie una password *s* che memorizza ed un intero *n*;
- le password utilizzate saranno $P_1 = f(f(\dots f(s)\dots))$ cioè $P_{i-1} = f(P_i)$;

- il server viene inizializzato con $P_0 = f(P_1)$ (valore memorizzato assieme al nome di login ed all'intero 1);

Autenticazione di tipo challenge-response

Al momento della registrazione di un nuovo utente, esso deve scegliere una **funzione matematica** (ad esempio x*x). Al momento del login, il server invia un **numero casuale** (a.e. 7) e l'utente deve rispondere con il **valore calcolato** dall'algoritmo (a.e. 49). L'algoritmo usato può variare a seconda del momento della giornata, del giorno della settimana ecc…

Per rendere le cose più complesse, l'utente, al momento della registrazione, sceglie una **chiave segreta k** che viene installata manualmente sul server; al momento del login, il server invia un **numero casuale r** al client che **calcola f(r, k)** (dove f è una funzione nota) ed invia il valore in risposta al server; il server ricalcola il valore e lo confronta con quello inviato dal client, consentendo o meno l'accesso.

Autenticazione tramite un oggetto posseduto dall'utente

Molti sistemi/servizi consentono di effettuare l'autenticazione tramite la lettura di una tessera/scheda e l'inserimento di un codice (per prevenire l'utilizzo di una tessera rubata). Le schede si possono suddividere in due categorie:

- **schede magnetiche**: l'informazione digitale (circa 140 byte) è contenuta su un **nastro magnetico** incollato sul retro della tessera;
- **chip card**: contengono un **circuito integrato** e si possono suddividere ulteriormente in:
    - **stored value card**: dotate di una memoria di circa 1 KB;
    - **smart card**: dotate di una CPU a 8 bit operante a 4MHz, 16KB di memoria ROM, 4KB di memoria EEPROM, 512 byte di RAM (memoria per computazioni temporanee), un canale di comunicazione a 9.600 bps. Le smart card quindi sono dei **piccoli computer** in grado di dialogare tramite un protocollo con un altro computer.

Il rischio è che l'oggetto venga perso.

Autenticazione tramite caratteristiche fisiche dell'utente

Per certe applicazioni, l'autenticazione avviene tramite il rilevamento di caratteristiche fisiche dell'utente. Sono previste due fasi:

- **misurazione e registrazione in un database** delle caratteristiche dell'utente al momento della sua registrazione;
- **identificazione dell'utente** tramite rilevamento e confronto delle caratteristiche con i valori registrati;

Il rischio qui è perdere il dito in un incidente e non poter più usare l'impronta per sbloccare e accedere ai dispositivi bloccati.

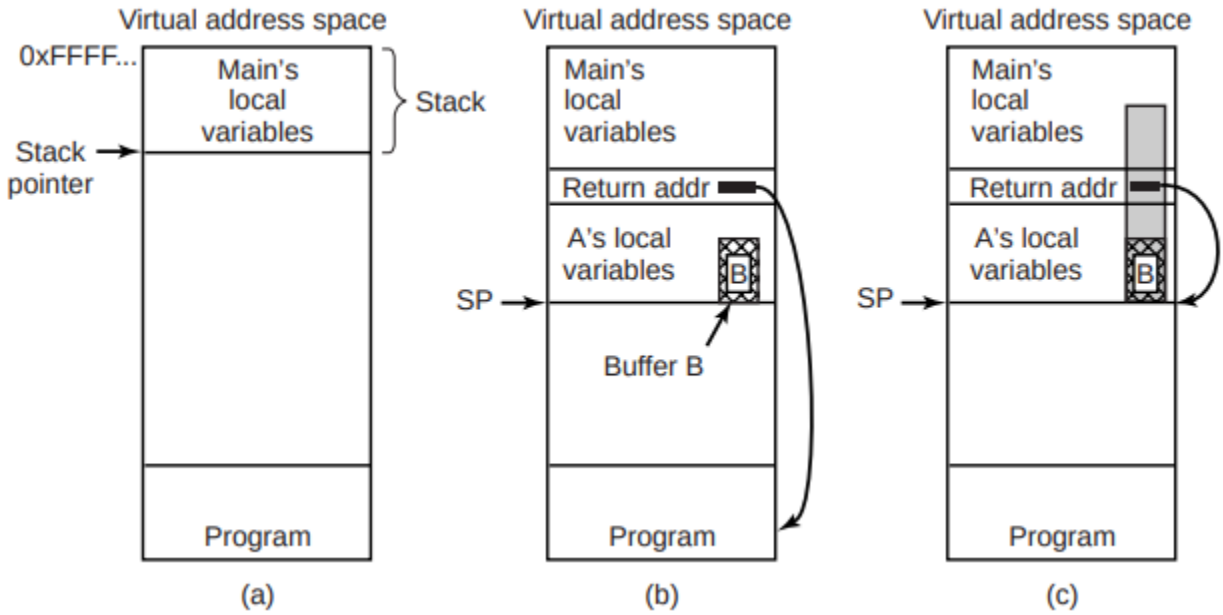
Attacchi dall'interno del sistema

L'attaccante è riuscito ad entrare nel sistema e vuole aumentare il suo grado di priorità per poter avere più potere all'interno del sistema.

Buffer overflow

Smashing the stack for fun and profit - Aleph One

Un buffer può essere visto da unf programmatore C come un **array contiguo di caratteri**. Gli array, come ogni variabile, possono essere dichiarati **statici** (vengono allocati quando vengono caricati) o **dinamici** (allocati al run time nello stack). In questo attacco di buffer overflow, ci si occuperà solo di buffer dinamici basati sullo stack.



(a) Inizio dell'esecuzione del programma;
(b) Chiamata della funzione;
(c) Overflow del buffer B e sovrascrittura dell'indirizzo di ritorno (con redirezione dell'indirizzo).

Composizione dei processi

Un processo è diviso in 3 regioni:

- **Text:** Occupa gli indirizzi di memoria più in basso. Ha dimensioni fisse decise dal programma, include codice e dati di sola lettura dell'eseguibile e ogni tentativo di scrittura viene marcato come **segment violation**.
- **Data:** Corrisponde ad un insieme di dati strutturati (**database**) dell'eseguibile. La dimensione di questa sezione può essere cambiata con la system call **brk(2)**. Se si tenta di superare la memoria massima disponibile, il processo si blocca e viene ri-schedulato per ripartire con una memoria allocata maggiore. La nuova memoria viene aggiunta tra i segmenti data e stack del processo.
- **Stack:** È una **struttura di dati** astratta. Lo stack viene utilizzato anche per **allocare dinamicamente** le variabili locali utilizzate nelle funzioni, per passare parametri alle funzioni e per restituire valori della funzione.

  Un oggetto di tipo stack ha la caratteristica che l'**ultimo elemento** inserito in essa, è il **primo ad essere rimosso** tramite le operazioni implementate dalla cpu:

- **Push**: aggiunge un elemento in cima allo stack;
- **Pop**: riduce di uno la dimensione dello stack, rimuovendo l'ultimo elemento aggiunto (che sta in cima).

La regione dello stack

Uno stack è un **blocco contiguo di memoria** contenente dati. Un registro chiamato **stack pointer** (SP) punta alla cima dello stack. La parte inferiore dello stack ha un **indirizzo fisso**. La sua **dimensione è regolata dinamicamente** dal kernel all'esecuzione.

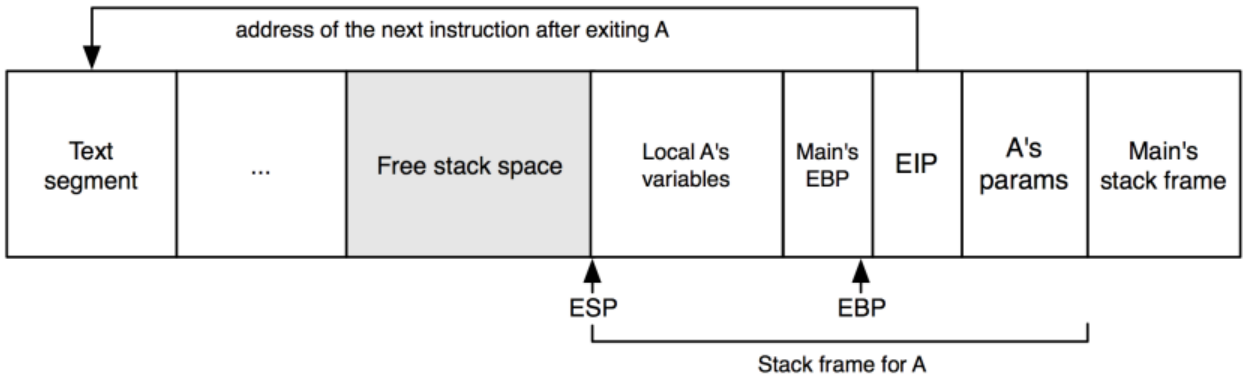
Quando in C viene chiamata una funzione, sullo stack viene allocato un nuovo **stack frame** e l'esecuzione del programma **non è più lineare** dato che ci si muove in indirizzi di memoria **non sequenziali** rispetto a quelli del programma "chiamante". La chiamata di una funzione è come un'istruzione di salto, con la differenza che nel momento in la funzione termina, il controllo va restituito all'istruzione che **seguiva la chiamata**. Viene poi utile, l'uso di un **registro EBP**, (chiamato **frame pointer** (FP)), che viene utilizzato per riferirsi a **variabili presenti nel frame** corrente (alternativamente si può ottenere lo stesso risultato utilizzando un offset sullo SP, anche se porta a problemi dato che, cambiando la dimensione dello stack, anche il campo offset utilizzato per riferirsi ad una variabile dovrebbe cambiare).

Uno **stack frame** contiene quindi:

- i **parametri di una funzione**;
- le **variabili locali** nel corpo della funzione;
- **due puntatori** che sono necessari per **ripristinare la situazione iniziale:**
  - il **saved frame pointer** (SFP): ripristina EBP sul suo valore precedente;
  - l'**indirizzo di ritorno;**

Se una funzione venisse richiamata dall'i**nterno di un'altra**, un nuovo stack frame verrebbe inserito nello stack e il suo indirizzo di ritorno sarebbe l'indirizzo dell'istruzione successiva al richiamo della funzione chiamata, nella funzione chiamante.

A seconda della tipologia dello stack, esso può **crescere verso l'alto** o **verso il basso** (tipologia di stack utilizzata molto dai processori Intel, Motorola, SPARC e MIPS). In questo esempio si userà uno stack che cresce verso il basso. Lo **stack pointer** può essere implementato **puntando all'ultimo indirizzo** o al **prossimo indirizzo libero** (quello in cima, con l'indirizzo più basso). In questo esempio si userà uno stack con SP che punta all'ultimo indirizzo dello stack.



Procedure prolog con esempio

Per capire quello che il programma fa per chiamare la funzione *function()*, si compila il programma con gcc usando il -S per generare il relativo codice assembly (*gcc -S -o example1.s example1.c*).

```
//example1.c:
void function(int a, int b, int c){
   char buffer1[5];
   char buffer2[10];
}
void main(){
   function(1,2,3);
}
```

La chiamata alla funzione è tradotta da:

```
pushl $3
pushl $2
pushl $1
Call function
```

Così facendo i tre argomenti vengono messi nello stack all'indietro e viene chiamata la funzione. L'istruzione di chiamata inserisce nello stack l'instruction pointer (IP). La prima cosa che viene fatta dalla funzione chiamata è il **procedure prolog**:

```
pushl %ebp       @ inserisce EBP (frame pointer) nello stack
movl %esp, %ebp  @ copia lo SP corrente nel EBP, rendendolo il nuovo FP
subl $20, %esp   @ esp = esp - 20, alloca spazio per le variabili locali
```

La prima cosa che una procedura deve fare quando viene chiamata è **salvare il precedente FP** (per ripristinarlo alla fine della procedura, verrà chiamato **SFP**). Successivamente si **copia lo SP nel FP** e **libera lo spazio** per le variabili locali.

Bisogna ricordare che la memoria può essere indirizzata solo in **multipli** della **dimensione di un word** (4 byte o 32 bit). Ad esempio un buffer di 5 byte richiede di avere 8 byte (2 parole) di memoria e un buffer di 10 byte richiede 12 byte (3 parole) e così via. Questo è il motivo per il quale viene sottratto 20 allo sp.

Al momento della chiamata della funzione, lo stack ora sarà così:

```
bottom of                                                    top of
memory                                                       memory
          buffer2       buffer1   sfp   ret   a     b     c
<------   [           ][        ][    ][    ][    ][    ][    ]

top of                                                    bottom of
stack                                                        stack
```

Si noti che l'indirizzo di ritorno (contenuto in *ret*) si trova a 12 byte dopo l'inizio di buffer1 (lunghezza di 2 words). Nel seguente esempio, si sfrutterà questa cosa per fare in modo che dopo il rientro dalla funzione *function(1,2,3)*, il programma non esegua la riga *x=1;* e vada direttamente a quella dopo:

```
void function(int a, int b, int c) {
       char buffer1[5];
       char buffer2[10];
       int *ret;
       ret = buffer1 + 12;
       (*ret) += 8;
}
void main() {
       int x;
       x = 0;
       function(1,2,3);
       x = 1;
       printf("%d\n",x);
}
```

Quello che è stato fatto è aggiungere 12 all'indirizzo di buffer1[]. Questo nuovo indirizzo è dove viene memorizzato l'**indirizzo di ritorno**.

Trojan Horse

Un Trojan Horse è un programma **apparentemente innocuo**, ma contenente del codice in grado di **modificare, cancellare, crittografare file, copiare file** in un punto da cui possano essere facilmente recuperati da un cracker, **spedire** i file direttamente via e-mail (SMTP) o FTP al cracker,...

Solitamente il modo migliore per far installare sui propri sistemi il trojan horse alle vittime è quello di includerlo in un **programma "appetibile"** gratuitamente scaricabile dalla rete. Un metodo per far eseguire il trojan horse una volta installato sul sistema della vittima è quello di sfruttare la variabile d'ambiente **PATH** (elencate le directory dove sono contenuti gli eseguibili):

Se il PATH contiene una lista di directory: */usr/local/bin:/usr/bin:/bin:/usr/X11R6/bin* quando l'utente digita il nome di un programma da eseguire, quest'ultimo viene ricercato per ordine nelle directory elencate nella variabile d'ambiente PATH. Quindi ognuna delle directory elencate in quest'ultima è un ottimo posto per "nascondere" un trojan horse. Di solito i nomi assegnati a tali programmi sono nomi di comandi comuni contenenti dei comuni errori di digitazione (ad esempio *la* invece del corretto *ls*). Questa tecnica di denominazione è molto efficace visto che anche il superutente (root) può compiere degli errori di digitazione dei comandi.

Trap Door

Scorciatoie non necessariamente con fini malevoli che consentono di **bypassare** (ad esempio) il sistema di login (può essere utile nel caso in cui si sta sviluppando qualcosa, per non perdere tempo a fare continuamente il login).

Attacchi dall'esterno del sistema (worm, virus)

Virus

Un virus è un programma che può "**riprodursi**" iniettando il proprio codice a quello di un altro programma. Caratteristiche del virus "perfetto":

- è in grado di **diffondersi** rapidamente,

- è **difficile da rilevare**,
- una volta rilevato, è molto **difficile da rimuovere**.

Essendo un programma a tutti gli effetti, un virus può compiere tutte le azioni accessibili ad un normale programma (visualizzare messaggi, manipolare il file system, generare altri processi ecc.).

Attivit`a di controllo/rilevazione degli attacchi (threat monitoring)

Crittografia

Codice mobile

Meccanismi di protezione

Laboratorio

Shell

La parte del sistema operativo UNIX dedita alla gestione dell'interazione con l'utente è la **shell**, ovvero, un'**interfaccia a carattere** dove l'utente impartisce i comandi al sistema digitando ad in apposito prompt.

Il sistema stampa sullo schermo del terminale eventuali messaggi all'utente in seguito all'esecuzione dei comandi, facendo poi riapparire il prompt, in modo da continuare l'interazione.

Tipi di shell e funzionamento

- **sh**: Bourne shell
- **bash**: Bourne again shell
- **csh**: C shell
- **tcsh**: Teach C shell
- **ksh**: Korn shell

Quando viene invocata una shell, automaticamente al login o esplicitamente:

1. viene **letto un file speciale** nella home directory dello user, contenente informazioni per l'inizializzazione;
2. viene **visualizzato un prompt**, in **attesa** che l'utente invii un comando;
3. se l'utente invia un comando, la shell lo **esegue e ritorna al punto 2**;

Pathname

- **Assoluto**: rispetto a *root/* (a.e. */home/gero/progetto/a*);
- **Relativo**: rispetto alla directory corrente (a.e. *progetto/a* supponendo di trovarsi in */home/gero*).

Comandi utili:

- Present working directory: `> pwd`
- Change directory: `> cd /dir`
- Spostarsi nella dir madre: `> cd ..`

Ricerca di file

`> find <pathnames> <expression>`

Attraversa ricorsivamente le directory specificate in `<pathnames>` applicando le regole specificate in `<expression>` a tutti i file e sottodirectory trovati.

`<expression>` può essere una fra le seguenti:

- Opzione,
- Condizione,
- Azione.

Esempi:

- `> find . -name '*.c' -print` → cerca ricorsivamente partendo dalla dir corrente i file con estensione c e li stampa.
- `> find . -name '*.bak' -ls -exec rm {} \;` → cerca ricorsivamente partendo dalla dir corrente tutti i file con estensione bak, li stampa con relativi attributi (-ls) e li cancella (-exec rm {} \; Il carattere \ serve per fare il "quote" del ;)
- `> find /etc -type d -print` cerca ricorsivamente a partire dalla dir /etc tutte e solo le sottodirectory, stampandole a video.

Comandi per manipolare file e directory

- Creazione directory: `> mkdir NOME`
- Rimozione directory: `> rmdir NOME`
- Copia file *f1* in *f2*: `> cp f1 f2`
- Sposta/rinomina file *f1* in *f2*: `> mv f1 f2`
- Torna byte e # linea dove f1 e f2 differiscono: `> cmp f1 f2`
- Cambiamenti da fare a f1 per renderlo =f2: `> diff f1 f2`
- Listing dei file: `> ls [OPTION]...[FILE]...`

A.e. eseguendo: `> ls -l /bin` si ottiene il seguente output:

```
...
lrwxrwxrwx    1 root     root            4 Dec  5  2000 awk -> gawk
-rwxr-xr-x    1 root     root         5780 Jul 13  2000 basename
-rwxr-xr-x    1 root     root       512540 Aug 22  2000 bash
...
```

Da sinistra a destra abbiamo:

1. **tipo di file** (**-** file normale, **d** directory, **l** link, **b** block device, **c** character device);
2. **permessi** (root, owner, group, world);
3. **numero di hard link** al file;
4. nome del **proprietario** del file;
5. nome dell'insieme di utenti che **possono accedere** al file come gruppo;
6. **grandezza** del file in byte;
7. data di **ultima modifica**;
8. **nome** del file.

Permessi e comando chmod

Linux è un sistema **multiutente**. Per ogni file ci sono 4 categorie di utenti: **root, owner, group, world**. L'amministratore del sistema (root) ha tutti i permessi (lettura, scrittura, esecuzione) su tutti i file. Per le altre categorie di utenti l'accesso ai file è regolato dai permessi:

```
> ls -l /etc/passwd
-rw-r--r--    1 root     root      981 Sep 20 16:32 /etc/passwd
```

Il blocco di caratteri `rw-r--r--` rappresenta i permessi di accesso al file:

- I primi 3 (`rw-`) sono riferiti all'owner;
- Il secondo blocco di 3 caratteri (`r--`) è riferito al group;
- l'ultimo blocco (`r--`) è riferito alla categoria world.

La prima posizione di ogni blocco rappresenta il **permesso di lettura** (r), la seconda il **permesso di scrittura** (w) e la terza il permesso di **esecuzione** (x) (Nota: per "attraversare" una directory, bisogna avere il permesso di esecuzione su di essa).

Un **trattino** (-) in una qualsiasi posizione indica l'**assenza del permesso** corrispondente.

L'owner di un file può **cambiarne i permessi** tramite il comando chmod:

- `> chmod 744 [FILE]` imposta i permessi del file [FILE] a rwxr--r--

  Infatti: rwxr--r-- 111 100 100 = 7 4 4 (leggendo ogni gruppo in ottale)

- `> chmod u=rwx,go=r f1` (produce lo stesso effetto del comando precedente) dove:
  - u rappresenta l'owner,
  - g il gruppo,
  - o il resto degli utenti (world).

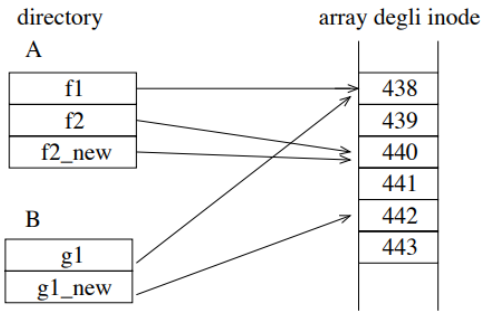
Inoltre: + aggiunge i permessi che lo seguono, - toglie i permessi che lo seguono, = imposta esattamente i permessi che lo seguono. Quindi l'effetto di chmod g+r f1 `e in generale diverso da chmod g=r f1.

Visualizzare il contenuto di un file

- Mostra tutto il contenuto di un file: `> cat [FILE]`
- Mostra il contenuto di un file a blocchi: `> more [FILE]`
- Mostra le ultime righe di un file: `> tail [FILE]`
- Mostra le prime righe di un file: `> head [FILE]`

Inode e link

In UNIX, ad ogni file corrisponde un numero di inode, che è l'**indice in un array memorizzato su disco**. Ogni elemento dell'array contiene le informazioni relative al file (data di creazione, proprietario, dove si trova il contenuto del file su disco, …). Le directory sono tabelle che associano nomi di file a numeri di inode. Ogni entry di una directory è un link.



Creazione di **link (hard)**, supponendo che la directory corrente sia A:

```
> ln f2 f2_new
```

Creazione di un **link simbolico**, supponendo che la directory corrente sia B:

`> ln -s g1 g1_new` un link simbolico è un tipo di file speciale in UNIX; `g1_new è` un file di testo che contiene il pathname di g1.

Metacaratteri della shell Unix

Un metacarattere è un carattere che rappresenta un **insieme di altri caratteri**. Quando l'utente invia un comando, la shell lo scandisce alla ricerca di eventuali metacaratteri, che processa in modo speciale. Una volta processati tutti i metacaratteri, viene eseguito il comando. Esempio:

- `> ls *.java` → Stampa la lista di tutti e soli i file che terminano con .java;
- `> ls prova?` → Stampa i file il cui nome è "prova?" dove "?" sta per un carattere a caso;
- `> ls /dev/tty[234]` → Stampa tutti i file che iniziano con "tty" e terminano con 2, 3 o 4;
- `> mkdir /home/gero/Desktop{a,b,c}` → Crea le directory "a", "b", "c".

Il quoting

Il meccanismo del quoting è utilizzato per **inibire** l'effetto dei **metacaratteri**. I metacaratteri a cui è applicato il quoting perdono il loro significato speciale e la shell li tratta come caratteri ordinari. Ci sono tre meccanismi di quoting:

- Il metacarattere di **escape** "\" inibisce l'effetto speciale del metacarattere che lo segue;
- Tutti i metacaratteri presenti in una stringa racchiusa tra **singoli apici** perdono l'effetto speciale;
- I metacaratteri per l'abbreviazione del pathname presenti in una stringa racchiusa tra **doppi apici** perdono l'effetto speciale.

Redirezione del I/O

Di default i comandi Unix prendono l'input da tastiera (**standard input**) e mandano l'output ed eventuali messaggi di errore su video (**standard output, error**). L'input/output in Unix può essere **rediretto** da/verso file, utilizzando opportuni metacaratteri:

- `>` redirezione dell'output;
- `>>` redirezione dell'output (append);
- `<` redirezione dell'input;
- `<<` redirezione dell'input dalla linea di comando ("here document");
- `2>` redirezione dei messaggi di errore.

Pipe

Il metacarattere "|" (**pipe**) serve per comporre *n* comandi "**in cascata**" in modo che l'output di ciascuno sia fornito in input al successivo. L'output dell'ultimo comando è l'output della pipeline.

Bash: History list

L'history list è un tool che consente di evitare all'utente di digitare più volte gli stessi comandi infatti **memorizza** nell'history list **gli ultimi 500 comandi** inseriti dall'utente. Viene memorizzata nel file .bash history nell'home directory dell'utente al momento del logout (e riletta al momento del login).

II il comando `history` consente di visualizzare la lista dei comandi. Esempio:

`$ history | tail -5` → stampa gli ultimi 5 comandi usati.

Ogni riga prodotta dal comando è detta **evento** ed è preceduta dal **numero dell'evento**. Esempio:

`$ ![NUMERO_EVENTO]` → esegue il comando che corrisponde al numero dell'evento;

`$ !!` → esegue l'ultimo comando digitato;

`$ ![QUALCOSA]` → esegue l'ultimo comando che contiene [QUALCOSA];

`$ !ls:s/al/i` → sostituisci (:s) la stringa "al" con la stringa "i";

Alias

Alias è un comando di shell che permette di **definire altri comandi**. All'uscita dalla shell gli alias creati con il comando alias sono **automaticamente rimossi**. Esempio:

`> alias [NOME]='[COMANDO]'` → digitando `[NOME]` esegue `[COMANDO]`;

Per **rimuovere** uno o più alias:

`> unalias [NOME_ALIAS1] [NOME_ALIAS2]` → rimuove i due alias;

Processi

Ogni processo del sistema ha un **PID** (Process Identity Number). Ogni processo può generare nuovi processi (figli). La radice della gerarchia di processi è il processo **`init`** con PID=1. `init` è il primo processo che parte al boot di sistema.

`> ps` → fornisce i processi presenti nel sistema associati alla shell corrente;

`> ps -af` → tutti i processi del sistema associati ad una shell (`-a`), full listing (`-f`);

`> ps -el` → tutti i processi del sistema non associati ad una shell (`-e`), long listing (`-l`);

`> tty` → terminale corrente

Comandi filtro

I filtri sono una particolare **classe di comandi** che possiedono i seguenti requisiti:

- prendono l'**input dallo standard input** device;
- **effettuano delle operazioni** sull'input ricevuto;
- **inviano il risultato** delle operazioni allo standard output device.

grep, fgrep ed egrep

Restituiscono solo le linee dell'input fornito che contengono un **pattern** specificato tramite espressione regolare o stringa fissata.

sort

Il comando sort prende in input delle **linee di testo**, le **ordina** (tenendo conto delle opzioni specificate dall'utente) e le **invia in output**.

tr

Permette di eseguire operazioni come la **conversione** di lettere minuscole in maiuscole, **cancellazione** della punteggiatura ecc.

Esempi:

- `> tr a-z A-Z` converte le minuscole in maiuscole.
- `> tr -c A-Za-z0-9 ' '` sostituisce i caratteri non alfanumerici con spazi (-c: complemento).
- `> tr -d str` cancella i caratteri contenuti nella stringa `str`.

cut and paste

Il comando **cut** serve a stampare parti di stringhe selezionate da un FILE. Senza FILE, o quando FILE è -, legge lo standard input.

```
> cut [OPTION] [FILE]
```

Indica con l'opzione `-f [NUM]` il campo da estrarre. Esempio con la stringa: "`ABC EFG HIJK`":

- `> cut -f 1` → stampa "ABC";
- `> cut -f 2` → stampa "EFG";

Con l'opzione `-d[CHAR]` si può specificare quale sarà il carattere che delimitano i vari campi (`-f`).
Esempio con la stringa: "`12345:Administration:james:smith`":

- `> cut -d: -f 2` → stampa "Administration";
- `> cut -d: -f 3` → stampa "james";

Stream editor: sed

sed (editor di flusso) è un comando che consente il **filtraggio** e la **manipolazione di testi**. Il comando viene tipicamente utilizzato in pipe in ambiente shell o specificando un file di input da riga di comando.

```
> sed [OPZIONI]... {file regole filtraggio} [file di input]..
```

Esempi:

`> echo 1:2:3:4:5 | sed s/:/-/g` → output: `1-2-3-4-5`

`> sed -n '10,50p' testo.txt` → stampa le righe dalla 10 alla 50

Sostituzione testo con sed

Il formato dell'azione di sostituzione in sed è il seguente:

`> s/expr/new/flags` dove:

- **expr** è l'espressione da cercare,
- **new** è la stringa da sostituire al posto di expr,
- **flags** è uno degli elementi seguenti:
  - **num**: un numero da 1 a 9 che specifica quale occorrenza di expr deve essere sostituita (di default `e la prima),
  - **g**: ogni occorrenza di expr viene sostituita,
  - **p**: la linea corrente viene stampata sullo standard output nel caso vi sia stata una sostituzione)
  - **w** file: la linea corrente viene accodata nel file file nel caso vi sia stata una sostituzione.

Shell script

Gli shell script sono **programmi** interpretati dalla shell, scritti in un linguaggio i cui costrutti atomici sono i **comandi** Unix. I comandi possono essere **combinati** in sequenza o mediante i **costrutti usuali** di un linguaggio di programmazione. La sintassi varia da shell a shell. Uno shell script va scritto in un file utilizzando per esempio il comando **cat o un editor** (vi, emacs, etc). Per poter eseguire lo script, il file deve essere reso **eseguibile**. Lo script viene eseguito invocando il nome del file. Esempio:

```
> cat >prova                    # il file prova viene editato
ls /usr/bin | wc -w            # scrittura del programma
Ctrl-d                         # fine dell'editing
> chmod 700 prova              # dirsize viene reso eseguibile
> ./prova                      # viene invocato il comando prova
459                            # risultato dell'esecuzione
```

Variabili

Le variabili della shell sono **stringhe di caratteri** a cui è **associato** un certo **spazio in memoria**. Le variabili della shell possono essere utilizzate sia sulla linea di comando che negli script. **Non c'è dichiarazione esplicita** delle variabili.

Assegnamento di una variabile (eventualmente nuova): variabile=valore (Importante: non lasciare spazi a sinistra ed a destra dell'operatore =).

Per **accedere** al valore di una variabile si utilizza il $:

```
> x=variabile
```

```
> echo il valore di x: $x
```

Output: il valore di x: variabile

Variabili d'ambiente

Le variabili definite come sopra sono **locali alla shell** o **allo script** in cui sono definite. Per rendere **globale** una variabile (e renderla una variabile d'ambiente) si usa il comando **export**:

```
> export x
```

Esiste un insieme di **variabili di ambiente speciali** definite al momento del login:

- PS1 prompt della shell;
- PS2 secondo prompt della shell; utilizzato a.e. in caso di ridirezione dell'input;
- PWD pathname assoluto della directory corrente;
- UID ID dello user corrente;
- PATH lista di pathname di directory in cui la shell cerca i comandi;
- HOME pathname assoluto della home directory

Parametri

Le variabili `$1, $2, …, $9` sono variabili **speciali associate** al primo, secondo, …, nono parametro passato sulla linea di comando quando viene invocato uno script. Se uno script ha più di 9 parametri, si utilizza il comando shift per fare lo shift a sinistra dei parametri e poter accedere ai parametri oltre il nono. Esempio script di nome `file` che contiene:

```
expr $1 + $2
```

Nel momento in cui si digita sulla shell:

```
> ./file 5 8
```

Da come output: 13

Variabili di stato automatiche

Sono variabili speciali che servono per **gestire lo stato** e sono aggiornate automaticamente dalla shell. L'utente può **accedervi solo in lettura**. Al termine dell'esecuzione di ogni comando unix, viene restituito un valore di uscita, exit status, uguale a:

- 0, se l'esecuzione è terminata con successo;
- diverso da 0, altrimenti (codice di errore).

La variabile speciale `$?` contiene il valore di uscita dell'ultimo comando eseguito.

Altre variabili di stato sono:

- `$?` exit status dell'ultimo comando eseguito dalla shell;
- `$$` PID della shell corrente;
- `$!` il PID dell'ultimo comando eseguito in background;
- `$-` le opzioni della shell corrente;
- `$#` numero dei parametri forniti allo script sulla linea di comando;
- `$*, $@` lista di tutti i parametri passati allo script sulla linea di comando.

Login script

Il login script è uno **script speciale eseguito al login**. Tale script è contenuto in uno dei file `.bash_profile`, `.bash_login`, `.bashrc`, `.profile`, memorizzati nella **home directory** degli utenti.

Il login script contiene alcuni comandi che è utile eseguire al momento del login, come la definizione di alcune variabili di ambiente. Ciascun utente può modificare il proprio login script, ad esempio (ri)definendo **variabili di ambiente** e **alias "permanenti"**. Esiste anche uno **script di login globale** contenuto nel file `/etc/profile` in cui l'amministratore di sistema può memorizzare dei comandi di configurazione che valgano per tutti. Lo script di logout eseguito al momento dell'uscita dalla shell (di login), si chiama solitamente `.bash_logout`.

Controllo di flusso if-then-else

Il comando condizionale:

```
if [CONDITION]
then
      [TRUE]
else
      [FALSE]
ii
```

Esegue il comando `[CONDITION]` e utilizza il suo **exit status** per decidere se eseguire i comandi `[TRUE]` (exit status 0) od i comandi `[FALSE]` (exit status diverso da zero).

Condizioni e comando test

Se la condizione che si vuole specificare non è esprimibile tramite l'exit status di un "normale" comando, si può utilizzare l'apposito comando **test**:

```
test expression
```

che restituisce un **exit status** pari a **0 se expression è vera**, **pari a 1 altrimenti**. Si possono costruire vari tipi di espressioni:

- Espressioni che controllano **se un file possiede certi attributi:**
  - -e f restituisce vero se f esiste;
  - -f f restituisce vero se f esiste ed è un file ordinario;
  - -d f restituisce vero se f esiste ed è una directory;
  - -r f restituisce vero se f esiste ed è leggibile dall'utente;
  - -w f restituisce vero se f esiste ed è scrivibile dall'utente;
  - -x f restituisce vero se f esiste ed è eseguibile dall'utente;
- Espressioni su **stringhe**:
  - -z str restituisce vero se str è di lunghezza zero;
  - -n str restituisce vero se str non è di lunghezza zero;
  - str1 = str2 restituisce vero se str1 è uguale a str2;
  - str1 != str2 restituisce vero se str1 è diversa da str2;
- Espressioni su **valori numerici**:
  - num1 -eq num2 restituisce vero se num1 è uguale a num2;
  - num1 -ne num2 restituisce vero se num1 non è uguale a num2;
  - num1 -lt num2 restituisce vero se num1 è minore di num2;
  - num1 -gt num2 restituisce vero se num1 è maggiore di num2;
  - num1 -le num2 restituisce vero se num1 è minore o uguale a num2;
  - num1 -ge num2 restituisce vero se num1 è maggiore o uguale a num2.
- **Espressioni composte**:
  - exp1 -a exp2 restituisce vero se sono vere sia exp1 che exp2;
  - exp1 -o exp2 restituisce vero se è vera exp1 o exp2;
  - ! exp restituisce vero se non è vera exp;
  - (exp) le parentesi possono essere usate per cambiare l'ordine di valutazione degli operatori (è necessario farne il quoting).

Cicli while

Vengono **eseguiti i comandi** `[COMMANDS]` **finché la condizione** `[CONDITION]` **è vera**. Sintassi:

```
while [CONDITION] do
      [COMMANDS]
done
```

Esempio:

```
while test -e $1 do
sleep 2
done echo file $1 does not exist
exit 0
```

Lo script precedente esegue un ciclo che dura finché il file fornito come argomento non viene cancellato. Il comando che viene eseguito come corpo del while è una pausa di 2 secondi.

Cicli uniti

Vengono **eseguiti i comandi** `[COMMANDS]` **finché la condizione** `[CONDITION]` **è falsa**. Sintassi:

```
until [CONDITION] do
[COMMANDS]
done
```

Esempio di uno script che legge continuamente dallo standard input e visualizza quanto letto sullo standard output, finché l'utente non inserisce la stringa end:

```
until false do
      read firstword restofline
      if test $firstword = end
      then
            exit 0
      else
            echo $firstword $restofline
      fi
done
```

Cicli for

Vengono eseguiti i comandi `[COMMANDS]` **per ogni elemento contenuto in wordlist** (l'elemento corrente è memorizzato nella variabile var). Sintassi:

```
for var in wordlist do
      [COMMANDS]
done
```

Sintassi alternativa solo per shell bash (serve specificarlo includendo `#!/bin/bash` in testa allo script):

```
for (( X=0; X<var; X++)) do
      [COMMANDS]
Done
```

Case selection

L'effetto risultante è che vengono eseguiti i comandi `[COMMAND1]`, `[COMMAND2]`,... a seconda del fatto che string sia uguale a `[EXPRESSION1]`, `[EXPRESSION2]`,... I comandi `[DEFAULT_COMMANDS]` vengono eseguiti soltanto se il valore di string non coincide con nessuno fra `[EXPRESSION1]`, `[EXPRESSION2]`,... I valori `[EXPRESSION1]`, `[EXPRESSION2]`,... possono essere specificati usando le solite regole per l'espansione del percorso (caratteri jolly). Sintassi:

```
case string in
[EXPRESSION1])
      [COMMAND1]
      ;;
[EXPRESSION2])
      [COMMAND2]
      ;;
...
*)
      [DEFAULT_COMMANDS]
      ;;
esac
```

Command substitution

Il meccanismo di command substitution permette di **sostituire** ad un comando o pipeline **quanto stampato sullo standard output** da quest'ultimo. Esempi:

```
> date
Tue Nov 19 17:50:10 2002
> vardata=`date`
> echo $vardata
Tue Nov 19 17:51:28 2002
```

Per operare una command substitution si devono usare gli "**apici rovesciati**" o backquote (`), non gli apici normali (') che si usano come meccanismo di quoting.

Il linguaggio C

Il C è un **linguaggio imperativo** legato a Unix (le chiamate di sistema sono definite come funzioni C infatti qualsiasi linguaggio passa per il C per interfacciarsi con il sistema operativo), adatto all'implementazione di compilatori e sistemi operativi. È stato progettato da D. Ritchie per il PDP-11 (all'inizio degli anni '70). Il C è un linguaggio:

- **Tipato** e **compilato**;
- Ad **alto livello**, ma non "troppo" in quanto fornisce le primitive per manipolare numeri, caratteri ed indirizzi, ma non oggetti composti come liste, stringhe, vettori ecc.
- **Piccolo** perché non fornisce direttamente delle primitive di input/output. Per effettuare queste operazioni si deve ricorrere alla Libreria Standard.

Compilazione

In ambiente Linux e macOS esistono due compilatori C open source e liberamente disponibili: **GCC** (GNU C Compiler) e **Clang**. Per compilare un programma, dopo averlo salvato in un file, ad esempio `programma.c`, si invoca il compilatore:

```
> gcc programma.c -o program
```

Con l'opzione `-o` si può rinominare il file eseguibile.

Il preprocessore

Ogni file C, prima della compilazione, viene **preprocessato**. Il preprocessore trasforma il codice interpretando delle **direttive** dove le direttive sono righe di codice che cominciano con un cancelletto # (a.e. `#include`, `#define`…), seguito dal nome della direttiva vera e propria. Il compilatore vede solo il risultato del preprocessing.

Espressioni booleane

La prima differenza importante rispetto al C++ e al Java è che in C **non esiste il tipo bool**. Infatti il costrutto if e i cicli for e while, si aspettano delle espressioni di tipo **intero** nelle proprie condizioni di controllo:

- Il **valore zero** viene interpretato come **falso**;
- Qualsiasi **valore diverso da zero** viene interpretato come **vero**.