



UNIVERSITÀ DEGLI STUDI DI PAVIA

Facoltà di Scienze MM. FF. NN., Medicina e Farmacia
Corso di Laurea Triennale in Biotecnologie

Corso di Analisi Matematica e Informatica
ANNO ACCADEMICO 2004–2005

Principi di funzionamento degli elaboratori elettronici

Andrea Rappoldi¹

Quarta parte

19 Struttura del sistema operativo UNIX

Il sistema operativo UNIX è strutturato su più strati, come mostrato in Fig. 19. Ciascuno strato si pone ad un diverso livello di interfaccia tra l'utente e la macchina.

Al livello superiore si trovano generalmente le *applicazioni*, ovvero i programmi utilizzati dall'utente per svolgere il proprio lavoro di elaborazione dati (acquisizione, analisi, calcolo, ecc.). Al di sotto di questo si trova il livello corrispondente alla *shell di comandi*, che rappresenta l'interfaccia tramite la quale vengono resi disponibili ed eseguiti i comandi di interazione con il *kernel*, il quale costituisce la parte principale e più importante del sistema operativo, di cui costituisce il vero e proprio nucleo o "nocciolo".

Il kernel provvede infatti al controllo di tutti i processi e alla gestione delle periferiche della macchina; provvede a regolare gli accessi alla macchina da parte degli utenti, e fornisce inoltre indicazioni e statistiche sull'utilizzo della macchina e delle sue risorse mediante opportuni strumenti di monitoraggio.

Il kernel, a sua volta, realizza le funzioni di controllo e gestione della macchina avvalendosi di alcune sue proprie componenti specifiche.

In particolare, lo *scheduler*, cui è già stato dato accenno nel Cap. 17, è responsabile della gestione della CPU, facendo in modo che questa possa venire utilizzata, a turno, da tutti i processi concorrenti.

Il *File System* rappresenta il sistema di gestione dello spazio disco (memoria di massa), fornendo una suddivisione logica e gerarchica dei volumi disponibili, in modo da rendere il più possibile facile ed immediato l'accesso alle informazioni volute.

¹Istituto Nazionale di Fisica Nucleare – via Bassi, 6 – 27100 Pavia
e-mail: Andrea.Rappoldi@pv.infn.it

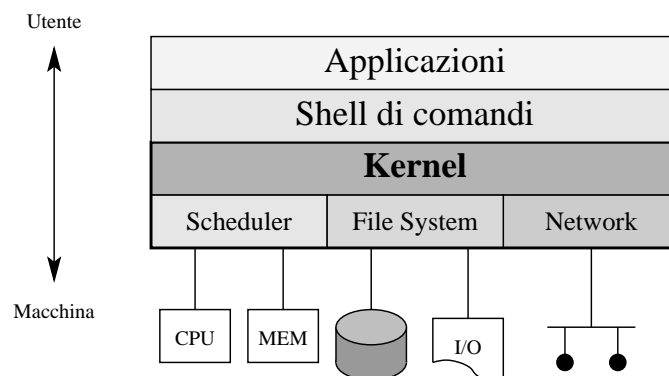


Figura 19: La struttura del sistema operativo UNIX può venire rappresentata da più livelli. L'utente interagisce normalmente con il livello superiore, corrispondente alle applicazioni, al di sotto del quale si trova il livello della shell di comandi, che funge da interfaccia verso il kernel, che rappresenta la parte principale del sistema operativo. Il kernel provvede ad amministrare le risorse della macchina tramite lo scheduler, che presiede al controllo del tempo di CPU e della disponibilità di memoria, il File System, che gestisce lo spazio disco e le operazioni di I/O, ed il sottosistema di Network a cui è demandato il controllo delle comunicazioni.

Il sottosistema di *Network* provvede alla gestione delle comunicazioni e del trasferimento di dati fra macchine diverse, fornendo metodi di indirizzamento ed accesso ai dati che risultino il più possibile efficienti e sicuri.

20 Gestione dei processi

Un processo, la cui definizione è già stata data nel Cap. 17, rappresenta una singola entità di esecuzione di un programma, a cui sono state assegnate le risorse necessarie.

In Unix ogni processo viene identificato da un opportuno identificatore denominato PID (Process IDentity), che è costituito da un valore numerico *univocamente* determinato al momento della sua creazione, così che non vi possano essere mai due processi con lo stesso PID.

Ogni processo presenta inoltre associati i seguenti parametri:

- PPID (Parent Process IDentity): identificatore del processo "genitore", ovvero de processo che ha generato il processo in esame;
- UID (User IDentity): identificatore dell'utente che risulta "proprietario" del processo;
- PRI (PRIority): livello di priorità assegnato al processo; la scala delle priorità è decrescente, per cui il valore 20 corrisponde solitamente alla priorità di base, i valori < 20 indicano un priorità più elevata, mentre i valori > 20 indicano priorità inferiori;
- CPU: tempo di macchina utilizzato dal processo;

- **STAT (STATus)**: indica lo stato del processo, che può essere *Running*, *Sleeping*, *Waiting*, ecc.;
- **RSS (ReSident Size)**: indica lo spazio di memoria RAM occupata dal processo;
- **Command**: è il nome del comando o del programma associato al processo.

Il comando **ps** (Process Status) consente di visualizzare la lista dei processi presenti sul sistema e, con l'aggiunta di opportune estensioni (ad es. **ps -Al**) consente di visualizzare, per ogni processo, le informazioni sopra indicate.

Alcuni dei parametri associati ad un processo, quali la priorità, possono venir modificati in qualunque momento, sia ad opera dell'utente (nel caso in cui voglia modificare il comportamento del processo), sia ad opera del kernel, come risultato di un'azione di controllo sui processi.

20.1 Organizzazione dei processi

In un sistema Unix i processi sono organizzati secondo uno schema gerarchico.

Infatti, ogni processo può generare altri processi, denominati *child*, i quali si trovano ad "ereditare" le caratteristiche del genitore, quali ad esempio UID e priorità (PRI), per cui l'insieme di tutti i processi si trova ad avere una struttura simile ad un albero genealogico. La struttura gerarchica dei processi è tale per cui nessun processo child può venir generato con più privilegi o con maggior priorità del processo genitore.

Un processo child può venir generato in modo indipendente dal genitore, oppure no. Nel primo caso il processo è di tipo *detached* e può continuare ad esistere anche dopo che il genitore ha terminato la propria esistenza. Nel secondo caso, invece, quando il genitore giunge a termine, anche il processo child viene terminato.

Durante la procedura di avvio del sistema operativo viene creato un processo speciale, denominato **init**, che ha $PID = 1$ e $UID = 0$ (corrispondente all'utente privilegiato "root"), il quale provvede a generare tutti gli altri processi associati alle diverse componenti del sistema operativo, secondo uno schema prestabilito. Tali processi, pertanto, hanno la caratteristica di avere $PPID = 1$.

Compito dello scheduler è quello di mandare in esecuzione, a turno, tutti i processi che sono in attesa essere eseguiti, in base alla loro priorità, così da dare precedenza ai processi con priorità più elevata (ovvero con valore di PRI più basso).

A parità di priorità, lo scheduler può scegliere il processo da mandare in esecuzione utilizzando degli opportuni algoritmi di scheduling, tra cui ad esempio quello denominato *Round-Robin*, che prevede una scelta a rotazione dei processi in attesa, e quello denominato *FIFO* (First-In-First-Out), basato invece sull'ordine in cui i processi vengono posti nella lista d'attesa. La scelta dell'algoritmo di scheduling è un problema abbastanza delicato, che spesso comporta il raggiungimento di un certo compromesso tra la disponibilità di risorse della macchina (in termini di CPU e di memoria) e le sue prestazioni in termini di tempo di risposta, che assumono una particolare importanza nei casi in cui si debba realizzare un sistema per l'elaborazione in tempo reale, ovvero che debba regire ai segnali esterni entro un tempo prestabilito.

Un processo può venir terminato prima del suo completamento mediante il comando `kill`.

21 Gestione della memoria

Un processo, per poter essere attivo, deve essere "residente" in memoria (RAM), in quanto questa rappresenta, di fatto, l'area di lavoro della CPU.

In certe circostanze, però, quando il numero di processi tende ad essere molto elevato (un sistema multiutente può comportare la presenza di parecchie centinaia di processi), la dimensione della memoria potrebbe risultare insufficiente, per cui risulterebbe impossibile creare nuovi processi e nessuna nuova applicazione potrebbe venir attivata, se non aspettando che qualcuno dei processi già attivi giunga a completamento.

Una tale situazione potrebbe risultare oltremodo pernicioso per il buon funzionamento della macchina, in quanto il sistema operativo stesso potrebbe trovarsi in una situazione in cui alcuni dei suoi processi non dispongono della memoria necessaria per poter svolgere il loro compito di controllo, e tutto il sistema cadrebbe in uno stato di paralisi ("*hang*").

Sono quindi stati ideati alcuni metodi di gestione della memoria particolarmente efficaci, tramite i quali è possibile evitare di giungere ad una situazione di stallo causata da memoria insufficiente. Uno di tali metodi prevede la realizzazione di un sistema di *memoria virtuale*, mentre in un altro caso viene utilizzata un'area di *swap*. Entrambi questi metodi si basano sul fatto di utilizzare lo spazio disco per sopperire alla carenza di memoria RAM e sono eventualmente abbinabili, così che solitamente vengono impiegati insieme.

21.1 Utilizzo della memoria virtuale

Si supponga che ad un certo istante, un determinato processo che sta scrivendo dei dati in memoria abbia raggiunto l'ultimo byte di memoria RAM disponibile, come mostrato in Fig. 20, e abbia bisogno di poter proseguire oltre.

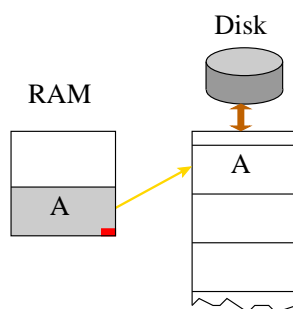


Figura 20: Quando si giunge ad una situazione in cui non vi è più memoria RAM disponibile per la scrittura, il sistema operativo provvede a trasferire su disco il contenuto di una certa area di memoria, indicata con A, rendendola libera per un suo utilizzo ulteriore.

In un sistema basato sull'uso della memoria virtuale, il kernel si accorge di tale evenienza e provvede quindi immediatamente a fermare temporaneamente

il processo in questione (mettendolo in stato di attesa), e quindi inizia un' operazione di trasferimento su disco del contenuto di una certa parte di RAM come indicato in Fig. 20. Ai fini di incrementarne l' efficienza, tale trasferimento viene generalmente effettuato per multipli di una certa quantità, denominata *pagina*, della dimensione tipica di 1 kB.

Al termine di tale operazione l' area di memoria RAM indicata con A risulta copiata su disco, ed è pertanto possibile cancellarne completamente il suo contenuto e ridarla in uso al processo che la stava utilizzando, il quale può continuare a scrivere nella regione di RAM ora indicata con B.

Se durante la scrittura dell' area B viene inoltrata una richiesta, sia da parte dello stesso processo che da parte di un altro, di accedere a dei dati che erano contenuti nell' area A, il kernel provvede innanzitutto a copiare su disco tutto il contenuto dell' area B (come mostrato in Fig. 21a), così che la RAM risulta ora disponibile e può dunque venir riscritta con il contenuto dell' area A, effettuando un' operazione di trasferimento da disco a RAM, come mostrato in Fig. 21b.

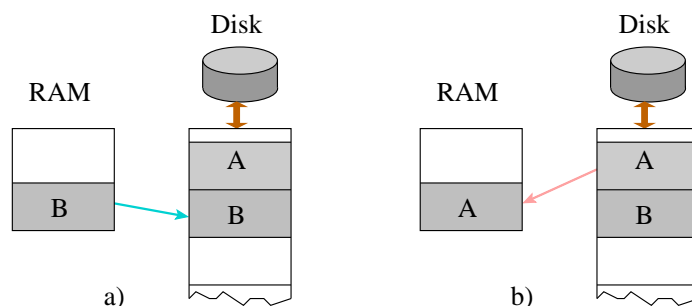


Figura 21: a) Nel caso in cui occorra accedere ad una qualche locazione di memoria dell' area A presente su disco, il sistema operativo provvede innanzitutto al salvataggio dell' area B, trasferendone il contenuto sul disco. b) Una volta salvato su disco il contenuto della RAM, è possibile utilizzare la stessa area di memoria per ospitare il contenuto dell' area A precedentemente copiato su disco, ripristinando la situazione che era stata mostrata in Fig. 20.

In definitiva il kernel provvede automaticamente a trasferire su disco le pagine di memoria RAM che non servono, per poter avere un lo spazio libero su cui trasferire il contenuto delle pagine di RAM (già salvate su disco) che vengono richieste di volta in volta.

È da notare che se la RAM viene utilizzata solo in lettura (e quindi non viene modificata) è possibile creare spazio su di essa semplicemente riscrivendoci sopra (a patto che il suo contenuto sia già stato precedentemente salvato su disco), mentre se anche uno solo dei byte di una certa pagina della RAM viene modificato, è necessario procedere a salvare su disco il contenuto di tutta la pagina, prima di potervi riscrivere sopra, per evitare di perdere tale modifica.

Questo sistema di gestione della memoria può sembrare piuttosto complicato, ma se viene usata l' accortezza di effettuare i trasferimenti in pagine di una certa dimensione il tutto risulta abbastanza veloce ed efficiente, ed il kernel riesce di fatto a gestire la memoria in modo del tutto "trasparente" ai processi, i quali in definitiva risultano avere a disposizione uno spazio di memoria *virtuale* molto più esteso della reale memoria *fisica*.

Ovviamente, dato che i tempi di accesso al disco sono molto più lunghi dei tempi di accesso alla memoria RAM, l'uso della memoria virtuale penalizza le prestazioni del sistema e non è assolutamente indicato per i sistemi di tipo real-time, dove è buona pratica evitarne l'utilizzo.

21.2 Utilizzo dell'area di swap

Questo metodo si basa sul fatto che, in mancanza di memoria RAM disponibile, i processi che non sono attivi possono venir trasferiti su disco.

Si prenda in considerazione la situazione mostrata in Fig. 22, dove la memoria RAM risulta completamente occupata, per via dei processi residenti A, B, C, D, E ed F.

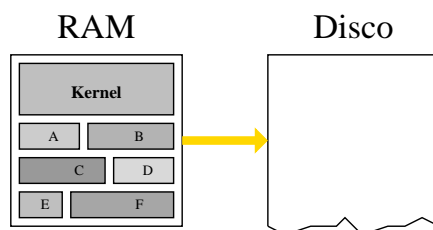


Figura 22: La memoria RAM risulta completamente occupata dai processi A, B, C, D, E ed F in essa residenti. (Si noti che il kernel, essendo un processo sempre attivo, occupa una parte di memoria permanentemente riservata ad esso). Quindi, per poter creare un nuovo processo, il kernel deve liberare una parte della RAM, individuando i processi che non sono attivi e provvedendo a trasferire tutto il loro contenuto sul disco.

Se ora dovesse giungere al kernel la richiesta di creazione di un nuovo processo, questa non potrebbe venir soddisfatta per mancanza di memoria disponibile. In tal caso il kernel provvede ad individuare (tramite lo scheduler) quali fra i processi residenti in memoria è in stato di attesa o dormiente, e quindi provvede a rimuovere uno o più di essi dalla RAM, trasferendo tutto il loro contenuto in un'area disco appositamente dedicata, denominata *area di swap* (in quanto l'operazione di trasferimento in blocco di un processo prende il nome di *swap-out*).

Nell'esempio mostrato si suppone che i processi B e D non siano attivi, e risultano quindi trasferibili su disco. Effettuata tale operazione, risulta disponibile un certo spazio in memoria RAM, per cui ora il kernel può procedere con la creazione del nuovo processo, indicato in Fig. 23 con la lettera G.

Nel caso in cui uno di processi presenti nell'area di swap (B o D) debba venir riattivato, il kernel deve provvedere ad operare il trasferimento in senso opposto (effettuando un'operazione di *swap-in*) ricopiandone tutto il suo contenuto in RAM, e qualora in essa non vi fosse lo spazio sufficiente, deve prima provvedere ad individuare uno o più processi il cui stato sia tale da consentirne il trasferimento su disco, onde poter liberare una quantità sufficiente di memoria.

Naturalmente, quando un processo termina, tutto lo spazio di memoria che occupava (sia su RAM che su disco) viene completamente rilasciato e reso disponibile per altri processi.

Anche in questo caso valgono le considerazioni già fatte per la memoria virtuale, in base alle quali l'uso dell'area di swap è da considerarsi una soluzione

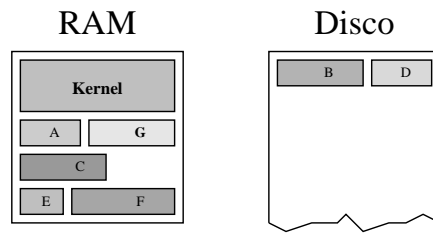


Figura 23: In seguito al trasferimento su disco dei processi B e D, si è resa disponibile un'area di memoria sufficiente alla creazione del nuovo processo G.

alla scarsità di RAM che presenta l'inconveniente di indurre forti rallentamenti nella gestione dei processi e dovrebbe, per quanto possibile, venire evitata dotando il sistema di una RAM di dimensione adeguata.

Nel malaugurato caso in cui la RAM risulti completamente occupata e non vi sia più spazio disponibile nemmeno nell'area di swap, il kernel non è più in grado di gestire i processi ed il sistema va in *hang*, divenendo del tutto inutilizzabile. Tali circostanze possono generalmente venir risolte solo applicando un segnale di Reset al sistema causandone un riavvio forzato.

L'area di swap deve essere di dimensione tale da garantire di poter ospitare un numero adeguato di processi. Tipicamente si segue la regola pratica di riservare all'area di swap una dimensione pari a 2 ÷ 3 volte la dimensione della memoria RAM.

Questo causa talvolta dei problemi in certi sistemi tipo Linux, dove si ha un limite massimo di 2 GB per l'area di swap, mentre la dimensione della RAM può essere superiore ad 1 GB. In tali casi è possibile definire ed utilizzare più aree di swap. Questa è una soluzione spesso consigliata per migliorare le prestazioni di un sistema, specialmente nel caso in cui le diverse aree di swap si trovino su dischi diversi. In questo caso, infatti, è possibile beneficiare del fatto che l'accesso a dischi diversi può avvenire in parallelo.

È evidente inoltre che il kernel, in quanto processo sempre attivo, è sempre residente in RAM, come evidenziato nelle Figg. 22 e 23.

22 Gestione dello spazio disco: il filesystem

I dischi magnetici costituiscono uno dei dispositivi più utilizzati per realizzare la memoria di massa. Dal punto di vista dell'utilizzo, essi presentano la caratteristica abbastanza peculiare di possedere un'organizzazione dei dati tipicamente *bidimensionale*, come schematizzato in Fig. 24.

Per contro, l'organizzazione della memoria RAM, che rappresenta la vera e propria area di lavoro della CPU, è di tipo *lineare*, in quanto è sufficiente una sola coordinata – l'indirizzo – per individuare e localizzare qualunque byte dei dati in essa contenuti.

Quindi, per poter utilizzare in modo efficiente le capacità di memorizzazione offerte dai dischi magnetici è necessario disporre di un opportuno sistema di conversione, che consenta alla CPU di poter accedere in modo semplice ad una qualunque parte dei dati presenti su disco.

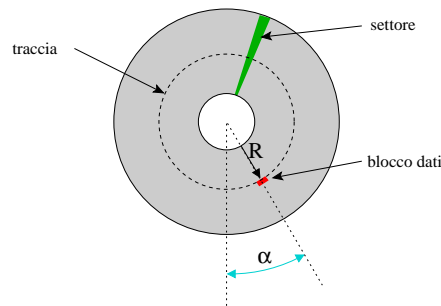


Figura 24: Ogni blocco di dati viene localizzato sulla superficie magnetica di un disco mediante due coordinate: la distanza R dal centro, che individua una determinata traccia del disco, e dall'angolo α che ne individua un determinato settore.

Inoltre, per sfruttare al meglio l'uso dei dischi come unità di archiviazione dei dati, si richiedono generalmente le seguenti caratteristiche:

- raggruppamento dei dati e delle informazioni in entità individuali e facilmente identificabili;
- organizzazione *strutturata* di tali entità, con struttura possibilmente *gerarchica*;
- facilità di manipolazione dei dati, con possibilità di creazione, cancellazione, spostamento, modifica;
- *accessi controllati*, onde consentire l'accesso agli archivi dati da parte di più utenti (su sistemi multi-utente), garantendone al contempo la sicurezza e la riservatezza;

Queste richieste, unitamente alla necessità di disporre di un facile metodo di indirizzamento dei dati, hanno portato alla realizzazione dei sistemi di gestione dei dischi generalmente indicati con il termine *filesystem*.

22.1 Struttura di un filesystem

Il *filesystem* può essere visto come il sottosistema del sistema operativo che si occupa della gestione dello spazio disco (organizzato in tracce ed in settori), e ne fornisce una visione organizzata in *files*.

Il file è una collezione di dati o informazioni, e rappresenta di fatto l'unità di base minima con cui possono venire gestiti e manipolati i dati su disco. Normalmente un file può essere costituito da un documento prodotto con un programma di scrittura, dal testo di un programma (codice sorgente), dal codice binario di un'applicazione o di un programma eseguibile, da immagini grafiche, ecc.

I files, specie se di dimensioni rilevanti, possono anche occupare zone non contigue del disco, come mostrato in Fig. 25.

Il kernel provvede in ogni caso a gestire il filesystem in modo trasparente, nascondendo i dettagli del modo in cui i vari files sono memorizzati su disco, ma rendendone disponibile il loro contenuto nel modo più semplice. Durante le

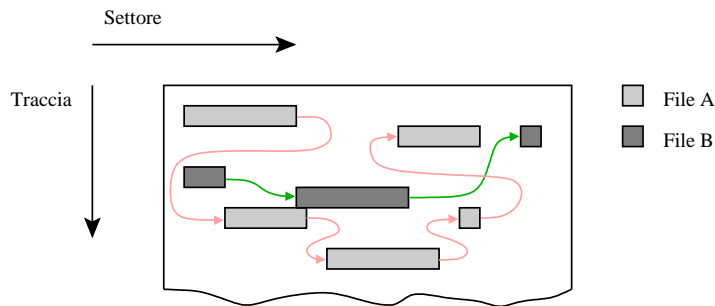


Figura 25: I files possono occupare aree di disco anche non contigue. È compito del filesystem fare in modo che l'utente non veda i dettagli di come i files sono disposti sul disco, ma possa accedere al loro contenuto nel modo più diretto.

operazioni di scrittura nel filesystem, il kernel cerca di utilizzare, se possibile, regioni di disco contigue, ma qualora questo non sia possibile, può utilizzare regioni situate in posizioni differenti (come mostrato Fig. 25), avendo cura di memorizzare nel filesystem stesso tutti i riferimenti necessari per "collegare" le varie parti che compongono ciascun file.

Ogni file viene univocamente individuato all'interno del filesystem mediante un descrittore detto *inode*, il quale si compone di più parti. In particolare, l'*inode number* è un indice numerico che identifica ciascun file presente nella struttura del filesystem.

Tuttavia risulterebbe alquanto scomodo e difficoltoso, per l'utente, dover ricorrere all'*inode number* per riferirsi ai files, per cui a tale indice viene associato per comodità un nome simbolico, la cui scelta è puramente arbitraria ed a discrezione dell'utilizzatore, benché vi siano di norma alcune regole da rispettare (dovute per lo più al fatto che vi sono alcuni caratteri speciali che non possono venir utilizzati per comporre il nome dei files).

L'utilizzo dell'*inode* viene solitamente limitato alle varie funzioni speciali del kernel relative alla gestione diretta del filesystem.

22.2 Organizzazione del filesystem

L'organizzazione di un filesystem è solitamente di tipo *gerarchico*, in quanto i files vengono idealmente raggruppati in una struttura ad albero, come mostrato in Fig. 26.

In tale struttura esistono dei file speciali, denominati *directory*, che sono di fatto dei contenitori di altri files, in modo del tutto equivalente alle "cartelle" o "folder" che si trovano in ambiente Windows, mentre i files ordinari corrispondono ai "documenti".

Nei filesystem utilizzati in ambienti Unix, il livello più elevato (indicato come livello 0 in Fig. 26) è costituito da un'unica directory che contiene l'intero albero, e viene pertanto denominata "radice" o *root* ed indicata con "/". Inoltre, come mostrato in Fig. 27, vi sono alcune particolari directory dal nome prefissato, che vengono utilizzate per contenere i files specifici del sistema operativo ed i files appartenenti agli utilizzatori del sistema.

Tra queste si possono citare:

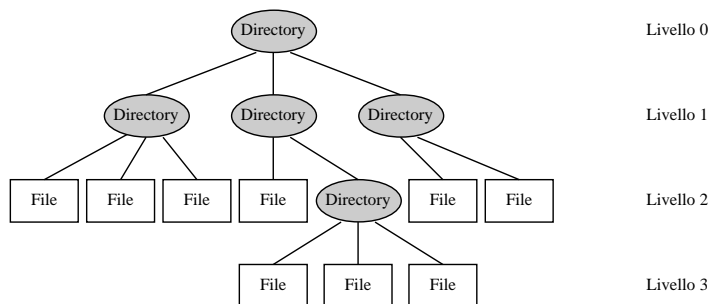


Figura 26: *Tipica struttura ad albero di un filesystem ad organizzazione gerarchica. Le directory, rappresentate con degli ovali, sono file speciali che possono contenere altri files (tra cui altre directory), idealmente situati ad un livello inferiore della struttura. Al livello 0 vi è necessariamente un' unica directory che contiene l' intero albero.*

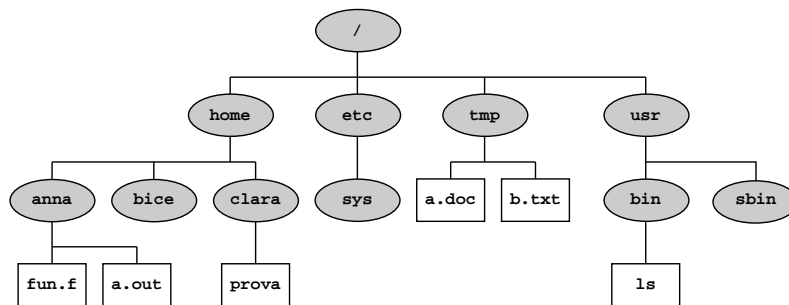


Figura 27: *Tipica struttura ad albero dei filesystem di un sistema Unix. I nomi utilizzati per le directory di livello più elevato, utilizzate per i files specifici del sistema operativo, costituiscono uno standard de facto.*

`/` denominata *root*, costituisce la "radice" dell' intera struttura ad albero secondo la quale sono organizzate tutte le directories del filesystem.

`usr` usata principalmente per contenere tutti i files strettamente legati al sistema operativo, tra cui i vari programmi, le procedure di comandi, ed i files di documentazione.

`etc` utilizzata per contenere tutti i files relativi alla configurazione del sistema.

`tmp` utilizzata come directory temporanea, sempre disponibile per poter contenere i files temporanei richiesti da alcuni programmi, limitamente al periodo del loro impiego. Normalmente, il contenuto di tale directory viene periodicamente cancellato in modo automatico, onde evitarne la saturazione.

`var` destinata a contenere i files che vengono continuamente aggiornati o modificati dal sistema operativo, quali ad esempio i *log files* che riportano i vari messaggi relativi al funzionamento del sistema.

home usualmente utilizzata come livello di partenza per contenere le varie directories associate agli account individuali. Così, ad esempio, lo spazio disco riservato all' utente **anna** risulta associato alla directory `/home/anna`.

Ogni file presente sul filesystem viene identificato in modo univoco mediante il suo percorso "assoluto", costituito dal nome del file preceduto dalla sequenza dei nomi di tutte le directories che ne individuano la sua posizione all' interno dell' albero a partire dalla radice, inserendo eventualmente il carattere "/" per delimitare i vari campi. Così, ad esempio, facendo riferimento alla Fig. 27, al file `fun.f` dell' utente **anna** corrisponde il percorso:

```
/home/anna/fun.f
```

mentre il comando `ls` (utilizzato per fare la lista dei files) è associato al file binario individuato da:

```
/usr/bin/ls
```

Tuttavia, per semplificare l' accesso ai files, e per evitare di dover specificare ogni volta il loro percorso assoluto (che può anche risultare costituito da una stringa molto lunga e complessa) si suole definire la cosiddetta *working directory*, costituita dal nome della directory rispetto alla quale – se non diversamente specificato – sono da riferirsi i nomi dei files. La working directory è definibile e modificabile in qualunque momento, utilizzando l' apposito comando `cd` (change directory) seguito dal nome della directory che deve divenire quella di uso corrente, mentre il suo valore può essere visualizzato mediante il comando `pwd` (print working directory).

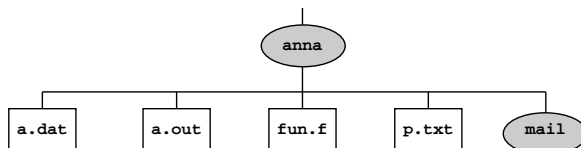


Figura 28: Particolare della struttura di files e directories relative all' utente **anna**.

Ad esempio, supponendo di dover lavorare per un certo tempo sui files relativi all' account **anna** schematizzati in Fig. 28 (cosa che risulta evidentemente usuale per l' utente **anna**), è possibile utilizzare i comandi seguenti²:

```
% cd /home anna      (modifica la working directory)
% pwd                (visualizza la working directory)
/home/anna
```

per cui in tutti i comandi successivi i files possono venire indicati mediante il loro percorso "relativo", ovvero riferito alla working directory `/home/anna`. Dunque la lista dei files presenti nella working directory, prodotta con il comando `ls -F` è del tipo:

```
% ls -F
a.dat      a.out*      fun.f      mail/      p.txt
```

²Qui e nel seguito il carattere "%" indica che il sistema operativo è in attesa di comandi, e pertanto ogni linea che inizia con "%" mostra un comando dato dal terminale, mentre le linee che non iniziano con tale carattere si riferiscono ai messaggi di risposta che compaiono sul monitor del terminale.

in cui il comando `ls (list)` è stato utilizzato con l'opzione `-F` per fornire alcune informazioni supplementari sui files elencati. In particolare, se il nome di un file è seguito da un'asterisco (`a.out*`) significa che si tratta di un file eseguibile (in formato binario o in forma di procedura di comandi), mentre con il simbolo `"/` viene evidenziato il nome dei files di tipo directory (`mail/`).

Il nome dei files può venir modificato mediante l'apposito comando `mv (move)`, il quale può anche essere utilizzato per spostare il file da un filesystem ad un altro. Così ad esempio il comando

```
% mv a.dat b.dat
```

ha l'effetto di cambiare il nome del file `a.dat`, modificandolo in `b.dat`, senza però modificarne il contenuto. È da notare che l'identificatore del file (il suo inode) rimane lo stesso, come si può evidenziare usando l'opzione `-i` del comando `ls`:

```
% ls -i
33023 a.dat          32965 pippo          32857 pluto          32948 prog.f

% mv a.dat b.dat
% ls -i
33023 b.dat          32965 pippo          32857 pluto          32948 prog.f
```

Invece, quando il comando `mv` viene usato per spostare il file su un filesystem diverso, il sistema operativo provvede prima a creare una nuova copia del file (a cui viene assegnato un nuovo inode) e quindi rimuove il file originale (solo nel caso in cui l'operazione di copia si concluda con successo). Si supponga ad esempio di avere le due directory `/dati` e `/programmi` associate a due diversi filesystem, con presenti i files indicati:

```
ls -i /dati
16780 a.dat          17200 b.dat          232173 c.dat          35539 d.dat
ls -i /programmi
66448 prog.f
```

il seguito all'esecuzione del comando `mv /dati/d.dat /programmi/z.dat` la situazione diventa la seguente:

```
ls -i /dati
16780 a.dat          17200 b.dat          232173 c.dat
ls -i /programmi
66448 prog.f          48612 z.dat
```

dove si nota chiaramente che il file `d.dat` inizialmente presente sul filesystem `/dati` (ed identificato dall' inode number 35539) è stato trasferito sul filesystem `/programmi` e qui identificato dall' inode number 48612, a cui viene associato il nome `z.dat`.

La cancellazione di un file viene effettuata con il comando `rm (remove)`, che ha la particolarità di provvedere solamente alla rimozione dal filesystem dell' inode corrispondente al file, senza cancellarne realmente il suo contenuto. Pertanto questo che significa che in seguito all'esecuzione del comando `rm`, lo spazio disco precedentemente occupato dal file ritorna disponibile per la scrittura di

eventuali altri files, ma fintanto che non viene effettuata nessuna operazione di scrittura su disco, il contenuto del file rimosso si trova ancora presente sul disco, benchè risulti del tutto inaccessibile ai normali comandi di manipolazione dei files (quali `ls`, `mv`, `cp`, `rm`, ecc.), in quanto si sono perse tutte le informazioni di collegamento contenute nel suo inode, necessarie per ricostruire la disposizione del file all' interno del filesystem.

Occorre dunque ricordare che la cancellazione di un file effettuata con il comando `rm` ha solo l' effetto di rendere nuovamente disponibile lo spazio disco che il file occupava, senza cancellarne effettivamente il contenuto, che viene perso definitivamente solo se in seguito, per effetto di successive operazioni di scrittura su disco, l' area precedentemente occupata dal file viene sovrascritta con dei nuovi contenuti.

22.3 Caratteristiche principali dei files

L' organizzazione dei filesystem comunemente utilizzati nei sistemi Unix prevede l' esistenza di particolari caratteristiche associate ad ogni file. Tra queste, le principali sono:

- il *tipo* di file, che può essere: ordinario, directory, link, ecc.;
- la *maschera di protezione*, che determina le regole di accesso al file, stabilendo eventuali restrizioni;
- il *proprietario* del file, identificato da *user* e *gruppo*, che ne individua l' appartenenza;
- la *dimensione*, generalmente espressa in byte.

Tali caratteristiche possono essere facilmente visualizzate utilizzando il comando `ls` con l' opzione `-l`, che produce la lista dei files in formato esteso (long) più ricco di informazioni, come nell' esempio seguente:

```
% ls
-rw-r----- 1 cl562094 matem          32 Jan  4 17:52 a.dat
-rwxr-xr-x   1 cl562094 matem       6771 Jan  4 17:53 a.out
drwxrwxr-x   2 cl562094 matem       4096 Jan  4 17:51 mail
-rw-rw-r--   1 cl562094 matem        173 Jan  4 17:53 prova.f
```

in cui le informazioni risultano disposte principalmente su sette colonne, evidenziate in Fig. 29, il cui significato viene ora analizzato.

1. La prima colonna indica la maschera di protezione del file, che risulta composta da 10 caratteri. Il primo carattere indica il tipo di file e può assumere i seguenti valori:

- per indicare un file di tipo *normale*
- d per indicare un file di tipo *directory*
- l per indicare un *link simbolico*

1	2	3	4	5	6	7
-rw-r-----	1	cl562094	matem	32	Jan 4 17:52	a.dat
-rwxr-xr-x	1	cl562094	matem	6771	Jan 4 17:53	a.out
drwxrwxr-x	2	cl562094	matem	4096	Jan 4 17:51	mail
-rw-rw-r--	1	cl562094	matem	173	Jan 4 17:53	prova.f

u	g	o				
r = read			utente	gruppo		
w = write			proprietario			
x = execute					dimensione	
						nome file
					data e ora	

tipo: -	= file normale
d	= directory
l	= link

Figura 29: Attributi dei files visualizzabili con il comando `ls -l`. I 9 caratteri che costituiscono la maschera di protezione, riportata nella prima colonna, si riferiscono alle modalità di accesso ai files da parte del proprietario del file (`u` = user), di altri utenti dello stesso gruppo (`g` = group), e tutti gli altri utenti (`o` = others).

I 9 caratteri successivi rappresentano tre gruppi di 3 caratteri che indicano il tipo di accesso consentito al proprietario del file (primi tre caratteri, indicati con "u"), ad altri utenti appartenenti allo stesso gruppo cui appartiene il proprietario del file (secondo gruppo di tre caratteri, indicati con "g"), e a tutti gli altri utenti che non si configurino nelle due precedenti categorie (ultimi tre caratteri, indicati con "o"). Il significato dei tre caratteri di ciascun gruppo è il seguente:

primo carattere:

- r indica possibilità di accesso in lettura
- nega l' accesso in lettura

secondo carattere:

- w indica possibilità di accesso in scrittura (o cancellazione)
- nega l' accesso in scrittura

terzo carattere:

- x indica possibilità di accesso in esecuzione
- nega l' accesso in esecuzione

Dunque nell' esempio mostrato sopra, il file `a.dat` risulta essere di tipo normale (-), accessibile in lettura e scrittura da parte del proprietario (`rw-`), accessibile in sola lettura da parte di altri utenti dello stesso gruppo del proprietario (`r--`) e non accessibile in nessun modo da parte di utenti diversi dai precedenti (`---`).

Il file `a.out` risulta invece avere le seguenti protezioni: accesso in lettura, scrittura ed esecuzione da parte del proprietario (`rwx`), accesso in lettura ed esecuzione da parte di utenti dello stesso gruppo od estranei (`r-x r-x`).

Il file `mail`, infine, è di tipo directory (`d`) e consente pieno accesso al proprietario ed agli altri utenti dello stesso gruppo (`rwx rwx`) e accesso in lettura ed esecuzione da parte di tutti gli altri utenti (`r-x`).

2. La seconda colonna indica una proprietà dei file abbastanza particolare (numero di link), che non conviene descrivere in questo contesto.
3. La terza colonna indica il nome del proprietario del file, che nell' esempio mostrato sopra risulta essere `c1562094`.
4. La quarta colonna riporta il nome del gruppo di cui il proprietario fa parte (`matem`). Pertanto, le protezioni indicate dal quinto, dal sesto e dal settimo carattere della prima colonna (contrassegnati con la lettera "g"), si applicano a tutti gli utenti, diversi dal proprietario del file, che appartengono a tale gruppo.
5. La quinta colonna indica la dimensione del file espressa un byte.
6. Nella sesta colonna viene riportata la data e l' ora in cui il file è stato creato o ha subito le ultime modifiche (ad es. `Jan 4 17:53` indica le ore 17:53 del 4 Gennaio dell' anno corrente).
7. L' ultima colonna, infine, riporta il nome completo del file.

Alcune delle caratteristiche dei files sopra mostrate sono modificabili mediante degli opportuni comandi, tra cui:

`chmod` (change mode): consente di modificare le modalità di accesso al file per ciascuno dei tre tipi di utente (`user`, `group`, `others`);

`chown` (change owner): consente di modificare il proprietario del file (`user` e `group`);

`touch`: consente di modificare data e ora del file, impostando il valore corrente, senza modificare il contenuto del file.

23 Utilizzo dello spazio disco

Al fine di utilizzare al meglio lo spazio disponibile sui dischi, si ricorre usualmente alla sua suddivisione fisica in *partizioni*, mentre dal punto di vista logico e funzionale viene impiegata la struttura di *directory* già accennata nel Par. 22.2.

23.1 Partizioni

Una partizione è costituita da una porzione contigua di spazio disco, sulla quale viene generalmente creata una struttura di filesystem, ovvero un sistema organizzato di files, come descritto nel Cap. 22.

Ogni partizione è definita in modo da utilizzare tutto lo spazio disco compreso tra due derminate tracce del disco (Fig. 24), di cui una costituisce la traccia di inizio, mentre l' altra rappresenta la traccia di fine della partizione, come mostrato in Fig. 30.

Ogni filesystem può utilizzare solo lo spazio disco della partizione su cui è stato creato, e non può in alcun modo superare i limiti imposti dalla traccia iniziale e finale. Le partizioni si comportano dunque come dei "compartimenti stagni" che confinano ogni filesystem nel proprio spazio riservato, impedendo che, in caso di errori o malfunzionamenti, si possa giungere al completo esaurimento dello spazio disco.

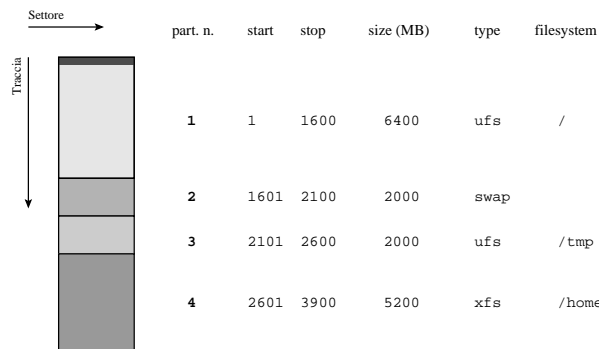


Figura 30: Schema della suddivisione dello spazio disco in partizioni. Ogni partizione, identificata da un numero progressivo, occupa tutto lo spazio contiguo compreso tra la traccia di inizio (**start**) e la traccia di fine (**stop**), per cui la sua dimensione (**size**) dipende in definitiva dal numero di tracce occupate. Su ogni partizione viene creata una struttura di filesystem che può essere di vario tipo, così come per l' area di swap che richiede un filesystem di tipo speciale, mentre normalmente viene utilizzato il tipo **ufs** (*Unix Filesystem*) o **xfs** (*Extended Filesystem*). La sottile parte superiore (in colore più scuro) indica la parte riservata per la partition table.

Il partizionamento del disco, inoltre, consente di poter avere contemporaneamente a disposizione sullo stesso disco più filesystem di tipo diverso. Ad esempio, è possibile in certi casi installare su uno stesso disco due sistemi operativi diversi (es. Linux e Windows), e fare in modo che quando il computer sta funzionando con un sistema operativo possa comunque accedere ai files presenti sulle partizioni dell' altro.

La Fig. 30 mostra un esempio di partizionamento di un disco, in cui per ogni partizione vengono mostrati: il numero identificativo della partizione, il numero della traccia di inizio, il numero della traccia di fine, la dimensione, il tipo di filesystem ed il nome della directory ad essa associata.

La definizione delle partizioni è contenuta nella *partition table*, che è la tabella che contiene sia i valori numerici corrispondenti alle tracce di inizio e di fine di ogni partizione, sia le caratteristiche costruttive del disco (la cosiddetta "geometria") – numero di tracce, numero di settori, dimensione delle tracce e dei settori – grazie alle quali il sistema operativo è in grado di indirizzare correttamente ogni singolo blocco di dati presente sul disco. La partition table si trova generalmente contenuta nei primi settori del disco (che possono venir acceduti in modo diretto), evidenziati in Fig. 30 con un colore più scuro.

Il tipo di filesystem determina le modalità con cui il kernel organizza lo spazio disco per ospitare le varie parti che costituiscono i files ed i vari puntatori necessari per ricostruire la loro disposizione (come mostrato in Fig. 25). Il tipo di filesystem più diffusamente adottato su tutti i sistemi Unix è denominato **ufs** (*Unix filesystem*), ma esistono anche altre versioni più elaborate che prevedono alcune estensioni della struttura, come nell' esempio del filesystem denominato **xfs** (*Extended filesystem*).

23.2 Filesystem speciali

In un sistema Unix correttamente configurato sono normalmente presenti alcune directory di uso specifico, necessarie per il buon funzionamento del sistema operativo già descritte nel Par. 22.2, le quali sono solitamente associate a filesystem di tipo standard.

Tuttavia, vengono anche definiti alcuni filesystem speciali (correntemente denominati "pseudo-file system") che consentono di interagire con alcune entità particolari del sistema operativo considerandoli allo stesso modo dei files ordinari. Tra questi vi sono i filesystem:

`/proc` interfaccia utilizzata dal kernel per interagire con i processi. Gli pseudo-file di questo filesystem contengono le informazioni ed i parametri di funzionamento di ogni processo. Ad esempio, la directory `/proc/26218` contiene gli pseudo-files di interfaccia relativi al processo con `PID = 26218`.

`/dev` contiene gli pseudo-files di definizione e di interfaccia delle varie periferiche (devices).

Quest'ultimo viene utilizzato dal kernel per accedere in modo *diretto* a tutti i vari device (dispositivi periferici) connessi al sistema, trattandoli come se fossero dei files ordinari. Questo significa, ad esempio, che un'operazione di scrittura di un byte effettuata su uno pseudo-file contenuto nella directory `/dev` ha come effetto quello di inviare il byte stesso al device corrispondente, mentre in modo del tutto analogo un'operazione di lettura da uno pseudo-device corrisponde ad una richiesta di input.

Alcuni esempi tipici di pseudo-devices sono i seguenti:

```
/dev/tty1   terminale n. 1
/dev/lp0    stampante (Line Printer)
/dev/mouse  mouse
/dev/sda    disco SCSI A (scsi disk a)
/dev/sda1  partizione "1" del disco SCSI A
/dev/hdb    disco IDE B (hard disk b)
/dev/fd0    floppy disk
```

Una caratteristica particolare del sistema operativo Unix è quella di trattare ogni entità (processi, dispositivi di I/O, periferiche) esattamente alla stessa stregua dei files, per cui le relative operazioni di controllo e di ingresso/uscita dei dati si riducono, di fatto, a delle operazioni di lettura e scrittura su uno specifico file o pseudo-file.

24 Gestione dei programmi

Il sistema operativo deve fornire agli utilizzatori di un elaboratore elettronico un mezzo pratico ed immediato che consenta di eseguire sia i propri programmi, realizzati espressamente per la soluzione di specifici problemi di calcolo, sia i comandi necessari per il controllo e la gestione dell'elaboratore stesso e delle sue risorse.

L'interfaccia tra il sistema operativo e l'utilizzatore è costituita dalla *shell*. La shell è di fatto un interprete di comandi (per cui viene molto frequentemente indicata come *shell di comandi*), ovvero uno speciale programma che

viene eseguito come processo indipendente, ed ha la funzione di ricevere i comandi inoltrati dall' utilizzatore (secondo una specifica sintassi), di verificarne l' esattezza formale, e quindi di procedere all' esecuzione del compito richiesto.

Per ogni comando ricevuto, la shell verifica innanzitutto se esiste un programma (ovvero un file eseguibile) con quel nome all' interno di una certa collezione di directory predefinita. Si possono dunque presentare differenti casi:

1. se il programma viene trovato, allora esso viene eseguito, passandogli eventualmente i vari ulteriori parametri presenti nel comando stesso;
2. se non viene trovato nessun programma corrispondente, la shell cerca di interpretarlo come comando "interno" della shell;
3. se anche questo non è possibile, viene generato un messaggio di errore.

Come esempio del primo caso, si consideri il comando:

```
% ls -l
```

In questo caso, la shell trova il file eseguibile `/usr/bin/ls` (normalmente presente in tutti i sistemi Unix)³e quindi lo esegue, producendo la lista in formato esteso (per via della presenza del parametro `-l`) dei files presenti nella directory corrente.

Alternativamente, è possibile passare alla shell la definizione completa di un file eseguibile, specificandolo mediante il suo percorso assoluto (come descritto nel Par. 22.2):

```
% /home/anna/a.exe
```

In tal caso la shell viene istruita ad eseguire direttamente il file indicato (che si suppone esistere), costituito ad esempio da un programma compilato dall' utente.

Il caso 2 può venire esemplificato dal comando:

```
% echo $PATH
/usr/local/bin:/usr/bin:/bin
```

In questa circostanza, dato che non esiste un file eseguibile di nome `echo` né nelle directory predefinite del sistema (`/usr/bin`), né nella directory corrente, la shell esegue il suo proprio comando intrinseco `echo`, che serve per visualizzare il valore della variabile `PATH`⁴ (il carattere "\$" che precede il nome della variabile indica alla shell di considerare non il nome della variabile, ma piuttosto il suo valore).

Come ultimo esempio, si consideri il caso seguente:

```
% boh
boh: Command not found.
```

In tale situazione il nome del comando (`boh`) non corrisponde né al nome di un file eseguibile presente in una delle directory descritte dalla variabile `PATH`, né al nome di un file eseguibile presente nella directory corrente, né tantomeno ad un comando intrinseco della shell, per cui viene generato un messaggio di errore di evidente significato.

³La directory `/usr/bin` è una delle prime directory di base in cui la shell effettua la ricerca dei files eseguibili corrispondenti ai programmi che costituiscono il sistema operativo.

⁴La variabile di shell `PATH` contiene l' elenco delle directory in cui la shell deve effettuare la ricerca dei files eseguibili, come descritto nel caso 1.

24.1 La shell di comandi

Data l'importanza della shell di comandi, che funge da interprete e interfaccia tra il sistema operativo e l'utente, è naturale che siano stati sviluppati diversi tipi di shell, ciascuna delle quali può presentare caratteristiche diverse, in quanto concepita per ottimizzare la realizzazione di determinati lavori.

Le shell più largamente utilizzate sono:

- sh** Bourne shell: tra le prime ad essere ideate, rappresenta una sorta di standard, in quanto è disponibile praticamente su tutti i sistemi Unix;
- bash** Enhanced Bourne shell: rappresenta una versione migliorata della precedente, in quanto offre alcune estensioni e facilitazioni d'uso, come ad esempio la possibilità di poter richiamare i comandi precedenti;
- csh** C shell: così chiamata perché utilizza una sintassi molto simile a quella del linguaggio C, per cui risulta talvolta preferita dai programmatori;
- tcsh** Enhanced C shell: la versione migliorata ed estesa della C shell, con possibilità di richiamo dei comandi.

Caratteristica comune di ogni shell è quella di poter definire delle *variabili di shell*, utilizzate per memorizzare valori numerici e stringhe alfanumeriche di uso frequente. Alcune variabili vengono automaticamente definite a priori per ogni processo che viene creato dal sistema operativo in seguito all'attivazione di una shell di comandi. Tra queste vi sono ad esempio:

- HOME** contenente il nome della directory di lavoro, definita inizialmente per ogni utente, detta *home directory*. Ad esempio, per l'utente di nome **anna** la variabile **HOME** assume tipicamente il valore `/home/anna`.
- PATH** elenco delle directory in cui la shell effettua la ricerca dei files eseguibili ogni volta che viene dato un comando.
- SHELL** nome della shell attualmente in uso (ad es. `/bin/bash`, `/bin/tcsh`, ecc.)

Una particolarità molto importante delle shell di comandi è data dalla possibilità di definire sequenze di comandi, dette *shell script*, da far eseguire alla shell in modo del tutto analogo alle istruzioni di un programma. Una shell script può contenere qualunque tipo di comando utilizzabile dalla shell, e quindi è possibile usare: comandi del sistema operativo, file eseguibili creati dall'utente e comandi intrinseci della shell (che possono differire a seconda della shell in uso), con la possibilità inoltre di poter definire ed utilizzare variabili di shell e variabili definite dall'utente.

La shell script viene di fatto realizzata scrivendo in un file la sequenza di comandi e di definizioni che la shell deve eseguire, una linea alla volta, interpretando i comandi in esso contenuti.

In pratica, qualunque sequenza di comandi data in modo interattivo (e correttamente eseguita con successo) può venir scritta in forma di file per poter essere eseguita in modo automatico dalla shell.

Si consideri l' esempio seguente, in cui viene mostrato il contenuto del file `delete.csh`:

```
#!/bin/tcsh
# Se il file /home/anna/a.dat esiste, viene cancellato
if (-f /home/anna/a.dat) then
    echo "Deleting file..."
    rm /home/anna/a.dat
else
    echo "File not found"
endif
```

1. La prima linea (`#!/bin/tcsh`) serve a dichiarare che tutti i comandi che seguono verranno eseguiti dalla shell `tcsh` (il cui programma corrispondente si trova appunto in `/bin`).
2. La seconda linea è un semplice commento⁵, che serve solo per descrivere brevemente le operazioni svolte dalla shell script.
3. La terza linea (`if (-f /home/anna/a.dat) then`) viene usata per effettuare un test condizionale: se la condizione specificata tra parentesi è vera (se il file `/home/anna/a.dat` esiste), allora vengono eseguite tutte le linee successive fino a quella con il comando `else`, altrimenti vengono eseguite le linee comprese tra il comando `else` ed il comando `endif`.
4. La quarta linea (`echo "Deleting file..."`) serve per visualizzare sul terminale il messaggio costituito dalla stringa compresa tra le virgolette.
5. La quinta linea provvede alla cancellazione del file `/home/anna/a.dat`.
6. Il comando `else` delimita l' inizio della parte da eseguire nel caso in cui il test specificato alla terza linea dia esito negativo.
7. Viene visualizzato un messaggio sul terminale ("File not found").
8. L' ultima linea (`endif`) rappresenta la fine delle sequenze condizionata dall' esito del test espresso nella terza linea.

Dunque, supponendo che esista il file `/home/anna/a.dat`, una prima esecuzione della shell script `delete.csh`⁶ produce la cancellazione dello stesso, mentre una sua successiva esecuzione non ottiene alcun effetto, in quanto il comando di cancellazione viene saltato in virtù del risultato negativo del test "if", e viene invece stampato un messaggio sul terminale.

```
% delete.csh
Deleting file...

% delete.csh
File not found
```

⁵Il carattere speciale "#" viene utilizzato per denotare i commenti: tutti i caratteri seguenti, fino alla fine della linea, vengono semplicemente trascurati dalla shell, tranne il caso in cui si tratti della prima linea e che questa inizi con "#!".

⁶La shell script viene eseguita semplicemente usando il suo nome come comando, in quanto è proprio compito della shell quello di cercare un file con tale nome e, se questo esiste, eseguirlo.

25 Descrizione dei comandi

In tutti i sistemi Unix è normalmente presente il comando `man` (manual pages) in grado di fornire molte indicazioni utili ed esempi di utilizzo di tutti i comandi presenti ed utilizzabili sul sistema. Tale comando produce la stampa sul terminale di un certo numero di informazioni, generalmente riunite in diverse categorie, tra le quali:

NAME nome del comando con una breve descrizione della sua azione;

SYNOPSIS rappresentazione della corretta sintassi di utilizzo del comando, con gli eventuali parametri ed opzioni;

DESCRIPTION descrizione completa del comando, con eventuale lista di tutte le possibili opzioni ed i possibili parametri, e loro descrizione;

SEE ALSO elenco di altri comandi eventualmente correlati;

BUGS eventuali difetti e/o malfunzionamenti noti.

Il comando `man` può essere chiamato direttamente dandogli come parametro il nome del comando di cui si vogliono ottenere le informazioni, oppure è possibile fare una specie di ricerca per argomenti, per trovare quali sono i comandi inerenti una certa tematica, specificando l'opzione `-k` (key) seguita da una parola chiave.

Nel primo caso si ha, ad esempio:

```
% man rcp
```

```
NAME rcp - remote file copy
```

```
SYNOPSIS rcp [-px] file1 file2 rcp [-px] [-r] file ... directory
```

```
DESCRIPTION Rcp copies files between machines. Each file or directory argument is either a remote file name of the form "name@rhost:path", or a local file name (containing no ':' characters, or a '/' before any ':'s).
```

```
-r If any of the source files are directories, rcp copies each subtree rooted at that name; in this case the destination must be a directory.
```

```
-p The -p option causes rcp to attempt to preserve (duplicate) in its copies the modification times and modes of the source files, ignoring the umask.
```

```
...
```

```
SEE ALSO cp(1), ftp(1), rsh(1), rlogin(1)
```

```
BUGS Doesn't detect all cases where the target of a copy might be a file in cases where only a directory should be legal.
```

Mentre nel secondo caso, volendo ad esempio ricercare tutti i comandi relativi alla parola chiave `"list"`, occorre dare il comando seguente:

```
% man -k list
```

che produce una lista di tutti i comandi nella cui descrizione compare la parola chiave `list`, tra i quali:

...

ls (1) - list directory contents

lsattr (1) - list file attributes on a Linux extended file system

lsmod (8) - list loaded modules

lsof (8) - list open files

lspci (8) - list all PCI devices

...

Il comando `man` è di grandissima utilità, in quanto permette di ottenere in maniera molto rapida ed efficiente tutte le informazioni ed i consigli necessari per il corretto uso non solo dei comandi relativi al sistema operativo, ma anche di tutte le funzioni intrinseche e specifiche del linguaggio C che, come si è detto, è il linguaggio di programmazione con il quale il sistema operativo Unix è realizzato.

Indice

19	Struttura del sistema operativo UNIX	45
20	Gestione dei processi	46
20.1	Organizzazione dei processi	47
21	Gestione della memoria	48
21.1	Utilizzo della memoria virtuale	48
21.2	Utilizzo dell' area di swap	50
22	Gestione dello spazio disco: il filesystem	51
22.1	Struttura di un filesystem	52
22.2	Organizzazione del filesystem	53
22.3	Caratteristiche principali dei files	57
23	Utilizzo dello spazio disco	59
23.1	Partizioni	59
23.2	Filesystem speciali	61
24	Gestione dei programmi	61
24.1	La shell di comandi	63
25	Descrizione dei comandi	65