Introduzione al Sistema Operativo UNIX

Giuseppe Vitillaro

Consiglio Nazionale delle Ricerche

Centro di Studio per il Calcolo Intensivo in Scienze Molecolari

Dipartimento di Chimica

Università degli Studi di Perugia

e-mail: <peppe@unipg.it>

Perugia, 27/11/1999

Breve storia di UNIX

- Bell Laboratories, 1969:
  - Il progetto MULTICS, per un sistema operativo time sharing multi-utente, termina senza un buon risultato nel 1969.
  - Il gruppo “MULTICS” si trova temporanemente senza un sistema di calcolo interattivo.
  - Ken Thompson e Dennis Ritchie disegnano un nuovo File System che si evolverà in una delle prime versioni del File System UNIX.
  - Thompson scrive un programma che simula il File System proposto e codifica un Kernel preliminare per un GE 645.
  - Allo stesso tempo scrive un gioco, “Space Travel”, in Fortran, per il sistema operativo GECOS (Honeywell 635).

Breve storia di UNIX

- Non soddisfatto ricodifica il programma per il PDP-7.
- Thompson diviene così padrone del PDP-7, ma il suo ambiente di lavoro richiede un cross-assemby tra il GECOS ed il PDP-7.
- Per creare un migliore ambiente di sviluppo Thompson e Ritchie implementano il loro disegno di Sistema Operativo sul PDP-7, che include il gestore dei processi ed un piccolo insieme di utilità.
- A questo punto il nuovo sistema diviene autonomo e non ha più necessità del GECOS come ambiente di sviluppo.
- Al nuovo sistema viene dato il nome di UNIX in contrapposizione (scaramanzia?) al nome MULTICS coniato dal Brian Kernighan.

Breve storia di UNIX

- UNIX è un sistema operativo che è stato scritto da programmatori per essere utilizzato da programmatori.

- Bell Laboratories, 1973:
  - D.M. Ritchie scrive un compilatore per il linguaggio C. Insieme a Thompson riscrivono in C il Kernel di UNIX.
  - Per la prima volta un sistema operativo viene scritto in un linguaggio differente dall'assembler della macchina ospite.
  - Ciò permetterà di "portare" il nuovo sistema su una grande varietà di piattaforme hardware.

Breve storia di UNIX

- Nel 1974 il sistema UNIX viene dato in licenza alle Università ad uso didattico:
  - Il sorgente viene distribuito insieme al sistema: UNIX può essere personalizzato ed adattato alle esigenze dell'ambiente in cui viene installato.
  - Nel 1977 il numero di sistemi UNIX installati raggiunge le 500 unità: 125 in Università
- Fino al 1980 UNIX rimane confinato nell'ambiente delle Università e dei laboratori di ricerca americani collegati alla rete ARPA del Dipartimento della Difesa americano.
- Nel periodo tra il 1978 ed il 1982 i Bell Labs combinano diverse versioni di un singolo sistema, conosciuto come UNIX System III.

Breve storia di UNIX

- L'Università di California, a Berkeley, sviluppa una differente versione di UNIX denominata BSD (Berkeley Software Distribution: 1980 BSD 4.1).

- Nel Gennaio 1983 la AT&T inizia il suo supporto ufficiale per UNIX System V, risultato delle aggiunte di nuove funzioni aggiunte al System III.

  Nasce il SunOS (derivato dal BSD).

- All'inizio del 1984 esistono circa 100.000 installazioni UNIX nel mondo che girano su macchine che vanno dal mainframe al microprocessore.

Breve storia di UNIX

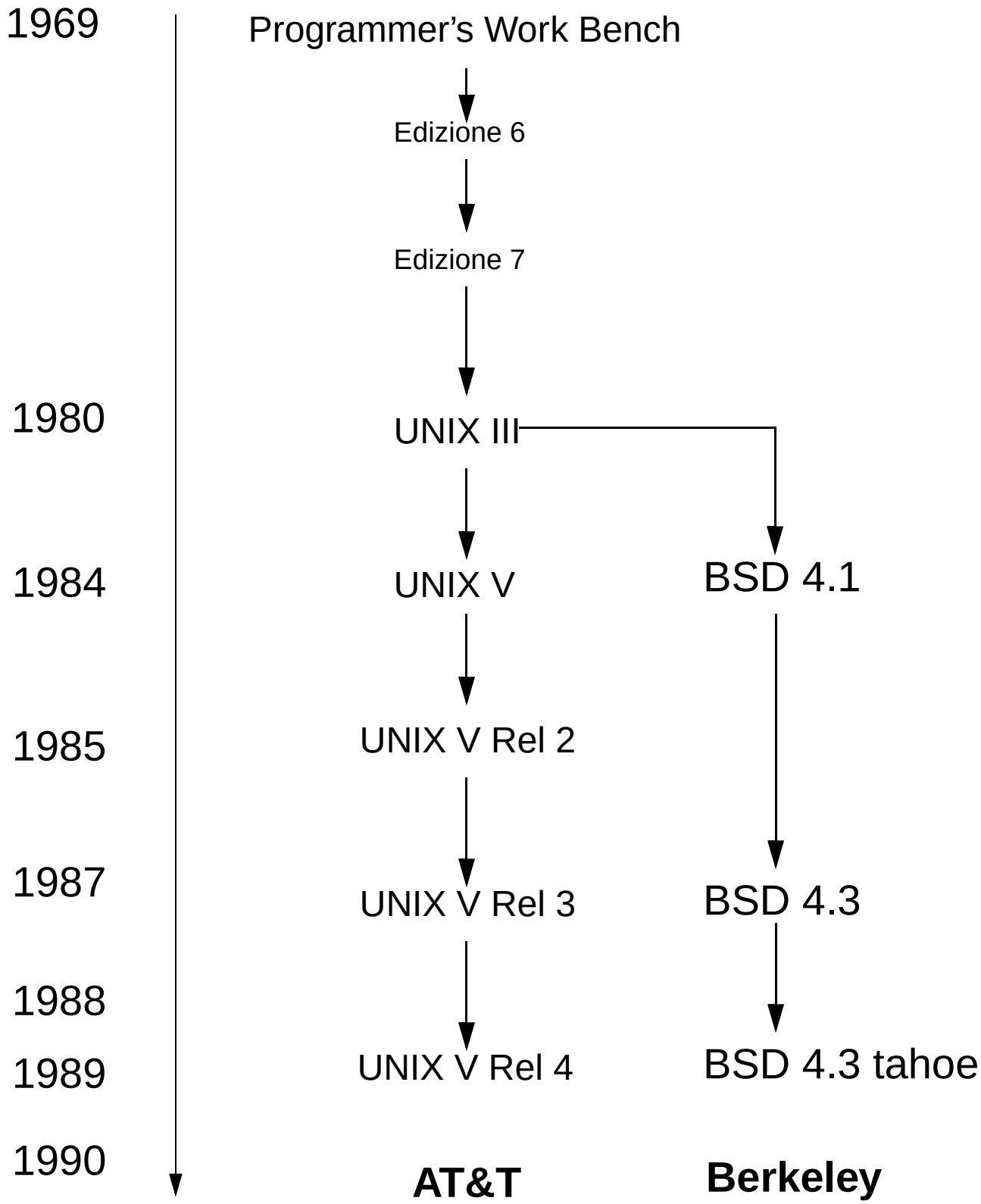
- Con la nascita dei microprocessori Motorola 68000, Zilog Z8000, National 32000 e Intel 80x86 il mercato inizia a diffondersi anche in ambienti come uffici, piccoli studi commerciali, applicazioni domestiche ed hobbistiche.

- Nel 1988 la AT&T e la Sun Microsystems sviluppano congiuntamente il System V.4: si evolverà in UNIXWare e Solaris 2.

- Nel 1993 la Novell acquisisce UNIX da AT&T.

- Verso la metà degli anni ‘90 si risolve il contenzioso legale tra Novell, proprietaria del trademark UNIX, e l’Università di Berkeley, proprietaria della linea di sviluppo BSD.

Albero Genealogico



Una sessione con UNIX

- L’accesso ad un sistema UNIX richiede, tipicamente, la conoscenza di una “userid” e di una “password”:
  - costituiscono la “chiave” di accesso al sistema
  - vengono definite dall’amministratore del sistema.
- Il sistema (comunque esso venga raggiunto) presenta un messaggio di login.
- Fornendo al sistema “userid” e “password” si ottiene l’accesso.
- Un esempio di sessione con un Sistema Operativo UNIX:

Una sessione con UNIX

```
Linux Kernel 2.0.34
Theoretical Chemistry
University of Perugia

login: tuser
Password:
Login incorrect

login: tuser
Password:


You have mail.
$
$
$ date
Wed Mar  3 13:31:29 CET 1999
$
$ who
sabatini tty1     Mar  2 11:02
sabatini ttyp0    Mar  2 11:17 (:0.0)
tuser    ttyp1    Mar  3 13:29 (simbad.thch.unipg.it)
$
$ mail
Mail version 8.1 6/6/93.  Type ? for help.
"/var/spool/mail/tuser": 2 messages 2 new
>N  1 gvt@unipg.it          Wed Mar  3 13:26  17/689   "Test"
&
Message 1:
Date: Wed, 3 Mar 1999 13:26:52 +0100 (CET)
From: Giuseppe Vitillaro <gvt@unipg.it>
To: tuser@sherazade.thch.unipg.it
Subject: Test

  Questa e' una nota di test.

& d
& q
$
$
ctl-D
```

Una sessione con UNIX

- L'esempio precedente mostra una tipica sessione con un sistema Linux. Al messaggio di login (login:) è necessario immettere "userid" e "password" esattamente come sono state fornite dall'amministratore di sistema.
  - UNIX è "**case sensitive**": distingue praticamente sempre tra minuscole e maiuscole.
  - È convenzione comune che le "userid" vengano immesse in **lettere minuscole**.
- Il punto culminante dell'operazione di **login** è la comparsa del **prompt**, usualmente (ove non siano state fatte personalizzazioni) un singolo carattere, il dollaro (**$**) o il percentuale (**%**).

Una sessione con UNIX

- Il prompt viene inviato da un programma chiamato **interprete dei comandi** o più semplicemente **shell**.

- Può essere presente un *Message of The Day* appena prima del prompt oppure la segnalazione che è presente della posta.

- In alcuni casi può venir richiesto il tipo di terminale che state usando (vt100, vt200, etc.).

- Da notare il carattere di *controllo* **ctl-D**: ordina alla shell di chiudere la sessione.

Una sessione con UNIX

- Il prompt della shell segnala che il sistema è pronto a ricevere comandi.

- Per esempio, per ottenere data ed ora dal sistema:

```
$ date
Wed Mar  3 18:18:02 CET 1999
$
```

- Il prossimo comando elenca gli utenti “logged in”:

```
$ who
sabatini tty1     Mar  2 11:02
sabatini ttyp6    Mar  2 12:23
sabatini ttyp7    Mar  2 12:23
tuser    ttyp1    Mar  3 13:29
peppe    ttyp0    Mar  3 13:37
```

Una sessione con UNIX

- Ancora “who” sulla vostra userid:

```
$ who am I
tuser    ttyp1    Mar  3 13:29
$
```

- Questo è il risultato di un errore nell’immissione di un comando:

```
$ whop
bash: whop: command not found
$
```

- Errori di battitura nell’immissione dei comandi possono essere corretti per mezzo di caratteri di controllo che spesso dipendono dal tipo di terminale e di shell utilizzati. I più comuni sono l’*erase* character (spesso su **backspace**) e il *line kill* character (spesso su **ctl-U**).

Una sessione con UNIX

- I caratteri di controllo possono venir visualizzati e modificati con il comando **stty**:

```
$ stty -a
speed 9600 baud; rows 38; columns 82; line = 0;
intr = ^C; quit = ^\; erase = ^?; kill = ^U; eof = ^D
```

- Il kernel del sistema legge i caratteri man mano che li inserite sul terminale, cosicchè potete inserire i comandi alla velocità che vi è più congeniale, anche se i caratteri non vengono visualizzati sul terminale (**type-ahead)**.

- L'esecuzione della maggior parte dei comandi può essere terminata mediante il carattere di interruzione (**intr)**. L'assegnazione più comune è sui tasti **ctl-C, ctl-BREAK.**

Una sessione con UNIX

- Un altro modo (più drastico) di terminare l'esecuzione di un programma è quello di spegnere il terminale dal quale state lavorando o comunque di interrompere la comunicazione con il sistema operativo. Attenzione: le sessioni telnet possono giocare brutti scherzi.

- Il carattere **ctl-S** (XOFF) sospende l'output sul terminale e il carattere **ctl-Q** (XON) lo ripristina.

- Il modo più semplice di concludere una sessione è quello di utilizzare il carattere di controllo **ctl-D** (EOF). È anche il modo più semplice (se non disabilitato) di essere sbattuti fuori senza aver capito perché. Meglio utilizzare il comando **exit**.

Introduzione al File System

- Le informazioni nei sistemi UNIX sono memorizzate in *files*. Ad ogni file viene assegnato un nome, dello spazio per mantenere i dati e delle informazioni che ne consentono l'amministrazione.

- Un file può contenere qualunque tipo di informazione e dal punto di vista del sistema non è altro che una **stringa di bytes**.

- Il File System è organizzato in modo che ciascun utente possa avere i suoi file personali senza dover interferire con i file di proprietà di altre persone. Molti comandi UNIX sono dedicati alla gestione ed amministrazione dei files.

Creazione di files

- La prima cosa necessaria (e forse la più importante in ogni sistema operativo) è imparare a creare e modificare i files.

- Un esempio forse un po' arcaico può essere fornito dall'editor di linea **ed**. È disponibile in ogni sistema UNIX e non necessita di un particolare tipo di terminale (potrebbe ancora oggi funzionare con una telescrivente).

```
$ ed prova                              Esegue l'editor
prova: No such file or directory        il file non esiste!
a                                       comando ed per aggiungere
provate a scrivere qualche cosa
qualunque cosa desideriate
scrivere ...
.                                       termina il modo append
w prova                                 scrive nel file prova
72                                      numero di caratteri
q                                       quit da ed
$
```

Creazione di files

- Il comando a “**append**” indica all’editor di memorizzare il testo che state immettendo nel suo buffer interno. Il “**.**” segnala la fine del testo e deve essere inserito all’inizio della linea.

- Il comando w “**write**” trasferisce il testo dal buffer interno di ed al file chiamato *prova*. Il *filename* può essere una qualunque stringa di vostro gradimento.

- ed risponde indicando il numero di caratteri salvati nel file.

Creazione di files

```
$ ed
a
Questo e’ il file test ...
.
w test
27
q
$
$ ed
a
... e questo e’ il file prova
.
w prova
30
q
$
```

- Il comando **ls** (list) elenca i nomi (non il contenuto dei files):

```
$ ls
prova  test
$
```

Creazione di files

- ls, così come la maggior parte dei comandi UNIX, permette di usare delle **opzioni** (note anche come **flags**) che modificano il comportamento di default del comando.

- Le opzioni seguono il nome del comando sulla “command line” e sono usualmente precedute dal carattere meno “**-**”. Per esempio **ls -l** produce un elenco più dettagliato delle informazioni relative a ciascun file:

```
$ls -l
total 2
-rw-rw-r--   1 tuser    tuser          30 Mar  3 19:20 prova
-rw-rw-r--   1 tuser    tuser          27 Mar  3 19:20 test
$
```

- “total 2” indica quanti blocchi di spazio disco occupano i files elencati (blocchi usualmente da 512 o 1024 bytes).

Creazione di files

La stringa -rw-rw-r-- indica chi ha il permesso di leggere /scrivere il file.

In questo caso il proprietario “**tuser**” ed il suo gruppo possono leggere e scrivere, gli altri possono solo leggere. Il numero “**1**” che segue è il numero di “link” del file.

30 e 27 sono, rispettivamente, il numero di bytes presenti nei files prova e test in accordo con in numeri forniti da ed.

La data e l’ora indicano l’ultima volta che il file è stato **modificato.**

- Le opzioni possono venir raggruppate: **ls -lt** produce lo stesso risultato di **ls -l**, ma ordinato per data. L’opzione **-u** produce informazioni sull’ultima data di accesso al file: **ls -lut**

Creazione di files

produce un long list ordinato per l'utilizzo più recente. L'opzione **-r** inverte l'ordine dell'output. In questo modo **ls -rt** lista i files per ordine di data, ma dal più vecchio al più recente.

```
$ ls -lrut
total 2
-rw-rw-r--   1 tuser    tuser          30 Mar  3 19:20 prova
-rw-rw-r--   1 tuser    tuser          27 Mar  3 19:39 test
$
```

- Ovviamente potete indicare ad ls il nome del file su cui deve operare.

```
$ ls -l test
-rw-rw-r--   1 tuser    tuser          27 Mar  3 19:20 test
$
```

- Le stringhe che seguono il comando sulla command line, come *-l* oppure *test*, sono denominate **argomenti** del comando.

Creazione di files

Gli argomenti sono usualmente opzioni o nomi di files che devono essere usati dal programma che implementa il comando.

In generale, se un comando accetta delle opzioni, esse precedono i filenames, ma possono apparire in qualsiasi ordine.

- Queste sono “consuetudini” a cui tipicamente la maggior parte dei comandi UNIX si uniformano. Non esiste però nessun vincolo di sistema che imponga di seguire queste regole.

  Qualche comando (vedi ad esempio **dd**) possiede le sue idiosincrasie e ciò può, a volte, sconcertare chi si accinge ad usare il sistema per la prima volta.

Visualizzazione di files

- Ora che abbiamo creato dei files nasce il problema di visualizzarne il contenuto a video. Esistono molti programmi che svolgono questo compito. Una possibilità, forse un po' eccessiva, consiste nell'usare ancora una volta ed:

```
$ ed test
27
1,$p
Questo e' il file test ...
q
$
```

- ed inizia riportando il numero di caratteri letti dal file test nel buffer interno. Il comando **1,$p** richiede ad ed di visualizzare sul terminale le righe di testo presenti nel file.

Visualizzazione di files

- Ovviamente esistono modi più semplici e meno onerosi di inviare il contenuto di un file sul nostro terminale.

  Il comando **cat** (forse uno dei più utilizzati, in qualche modo paragonabile al comando **type** dell'MSDOS) svolge esattamente questa funzione.

```
$ cat test
Questo e' il file test ...
$ cat prova
... e questo e' il file prova
$ cat test prova
Questo e' il file test ...
... e questo e' il file prova
$
```

- I files elencati come argomenti vengono *concatenati* (in inglese *Catenate* una abbreviazione di "concatenate") sul terminale.

Visualizzazione di files

Nulla viene inserito tra un file e l'altro: **cat** si limita ad emettere come output le righe dei due files in sequenza.

In generale i comandi UNIX tendono a seguire questa regola e ciò è molto importante: permette ai comandi UNIX di "cooperare" allo scopo di ottenere sequenze di comandi complesse.

- Non c'è nessun problema con files brevi, ma **ctl-S** e **ctl-Q** possono aiutare a visualizzare files più lunghi su un terminale "veloce".

  Il comando **more** permette di visualizzare un file una pagina alla volta.

Operare sui files

- Per operare sui files è necessario essere in grado almeno di copiare, cancellare e rinominare un file.

- Per ottenere una copia del file “test” creato nella sezione precedente:

```
$ ls -l
total 2
-rw-rw-r--   1 tuser    tuser          30 Mar  3 19:20 prova
-rw-rw-r--   1 tuser    tuser          27 Mar  3 19:20 test
$ cp test nuovo
$
$ ls -l
total 3
-rw-rw-r--   1 tuser    tuser          27 Mar  4 11:38 nuovo
-rw-rw-r--   1 tuser    tuser          30 Mar  3 19:20 prova
-rw-rw-r--   1 tuser    tuser          27 Mar  3 19:20 test
$
```

- Il comando “**cp test nuovo**” (copy) copia il file “test” nel file “nuovo”. Il comando “ls” ci ha permesso di verificare che l’operazione è andata a buon fine.

Operare sui files

- Rinominiamo il file “nuovo”. Nei sistemi UNIX viene utilizzato il paradigma di **muovere** un file all’interno del file system:

```
$
$ mv nuovo n_test
$
$ cat nuovo
cat: nuovo: No such file or directory
$
$ cat n_test
Questo e’ il file test ...
$
$
```

- L’esempio precedente ci mostra che ora il file “nuovo” non esiste più. Al suo posto esiste il file “n_test” con lo stesso *contenuto*.

  Attenzione: se il ***file target*** esiste già il comando **mv** lo rimpiazzerà con il ***file source*** ed il suo contenuto verrà perso.

Operare sui files

- Cancelliamo ora la copia del file “test” il cui ultimo nome è “n_test”. Il comando usato per cancellare è **rm** (remove):

```
$
$ ls -l
total 3
-rw-rw-r--   1 tuser    tuser          27 Mar  4 11:38 n_test
-rw-rw-r--   1 tuser    tuser          30 Mar  3 19:20 prova
-rw-rw-r--   1 tuser    tuser          27 Mar  3 19:20 test
$ rm n_test
$
$ ls -l
total 2
-rw-rw-r--   1 tuser    tuser          30 Mar  3 19:20 prova
-rw-rw-r--   1 tuser    tuser          27 Mar  3 19:20 test
$
```

- Abbiamo cancellato il file “n_test” con il comando “**rm n_test**” ed il comando “ls” ci mostra la nuova situazione.

- Il comando “rm” può cancellare un elenco di files, forniti come argomenti.

Operare sui files

Possiede un opzione “**-i**” (interactive) che richiede conferma prima di eseguire la cancellazione di ogni file:

```
$ cp test test1
$ cp test test2
$
$ rm -i test1 test2                (opzione -i)
rm: remove ‘test1’? y              (conferma la cancellazione)
rm: remove ‘test2’? n              (la annulla)
$
$ ls -l
total 3
-rw-rw-r--   1 tuser    tuser          30 Mar  3 19:20 prova
-rw-rw-r--   1 tuser    tuser          27 Mar  3 19:20 test
-rw-rw-r--   1 tuser    tuser          27 Mar  4 11:52 test2
$
```

- **cp**, **mv** ed **rm** costituiscono l’*alfabeto* che permette di operare sul file system eseguendo le operazioni **fondamentali** sui files.

Files e Directories

- Abbiamo usato i **Files** UNIX per alcuni semplici esperimenti. Ma che cosa è un file?

  Dal punto di vista del sistema un file è:

  “Una sequenza di bytes”

- I files UNIX non possiedono una struttura predefinita (non esiste il concetto di **record**).

  È probabilmente la struttura più semplice che si possa pensare.

- Il contenuto di un file ha significato solo per i programmi che lo utilizzano.

- I files sono in genere memorizzati su supporto magnetico.

Files e Directories

- Ad ogni file è associato un **filename**: il nome del file. È una stringa di caratteri che può essere scelta in quasi assoluta libertà. Il senso comune suggerisce però di evitare caratteri che non sono visualizzabili sul terminale e i metacaratteri usati dalla shell o da altri programmi di utilità (* . - \ etc.).

- La lunghezza massima del filename è di **255** caratteri sotto **Linux**, ma dipende in effetti dalla implementazione. In alcuni vecchie versioni di UNIX System V il massimo era di **14** caratteri.

- Ricordate sempre che UNIX distingue lettere maiuscole e minuscole. Il filename “test” è quindi differente da “TEST”, “Test”, etc.

Files e Directories

```
$
$ cat > TEST
Questa e‘ una riga del file TEST.
$ (CTL-D)
$ ls -l
total 4
-rw-rw-r--   1 tuser    tuser          34 Mar  4 13:20 TEST
-rw-rw-r--   1 tuser    tuser          30 Mar  3 19:20 prova
-rw-rw-r--   1 tuser    tuser          27 Mar  3 19:20 test
-rw-rw-r--   1 tuser    tuser          27 Mar  4 11:52 test2
$
$ cat test
Questo e’ il file test ...
$
$ cat TEST
Questa e‘ una riga del file TEST.
$
```

- È probabile che in ambiente multi-utente si verifichino omonimie, ovvero che più utenti si trovino ad usare lo stesso filename per files differenti.

- UNIX risolve questo problema raggruppando i files in **directories**.

Files e Directories

Una **directory** è un file (con particolari caratteristiche) che funge da catalogo e contiene informazioni relative ad altri files.

- Una directory può contenere files ordinari oppure altre directories: viene così a costituirsi una struttura **gerarchica**. In modo più naturale possiamo immaginarla come un **albero** di directories e files.

  È possibile attraversare l'albero per individuare ogni file presente nel sistema, partendo dalla *radice* (**root**) e muovendosi lungo i rami.

- Ogni utente possiede una directory *personale*, la **home** che contiene, tipicamente, solo files che gli appartengono. Di norma, subito dopo il

Files e Directories

login vi trovate posizionati nella vostra “home directory.

È ovviamente possibile cambiare la directory di lavoro (**current directory**), ma la vostra *home* rimane la stessa.

- Partiamo dal comando **pwd**: print working directory. Visualizza il nome della current directory:

```
$
$ pwd
/u1/tuser
$
```

Il risultato del comando vi informa che siete nella directory **tuser** che a sua volta è contenuta nella directory **u1** posizionata nella **root directory**.

Files e Directories

La radice viene convezionalmente identificata dal carattere “***/***”. Il carattere “/” (barra o slash) separa anche i componenti del nome (**pathname**). Il limite di 255 (o 14) caratteri si applica solo ad ogni singolo componente, mentre il pathname può raggiungere lunghezze uguali o superiori a 1024 caratteri.

Un filename non *dovrebbe* contenere mai il carattere slash ***/*** (tranne errori ed omissioni).

- Tutti i comandi che abbiamo esaminato fino ad ora possono agire su un “filename” o su un “pathname”.

  Il comando “**ls -l /u1/tuser**” otterrà un long list della directory “/u1/tuser” (la home directory

Files e Directories

dell'utente "tuser" che ci ha accompagnato fino ad ora).

```
$
$ ls -l /u1/tuser
total 4
-rw-rw-r--   1 tuser    tuser          34 Mar  4 13:20 TEST
-rw-rw-r--   1 tuser    tuser          30 Mar  3 19:20 prova
-rw-rw-r--   1 tuser    tuser          27 Mar  3 19:20 test
-rw-rw-r--   1 tuser    tuser          27 Mar  4 11:52 test2
$
```

- Ed ora un elenco della directory **/u1** che contiene le "home directories" degli utenti di questo sistema:

```
$
$ ls -l /u1
total 21
drwx------   2 condor   condor       1024 Sep 18 09:06 condor
drwxr-xr-x   6 root     root         1024 Aug 13  1998 ftp
drwxr-xr-x   5 root     root         1024 Aug 13  1998 httpd
drwxr-xr-x   2 root     root        12288 Aug 10  1998 lost+found
drwx------   7 peppe    thch         1024 Mar  3 13:56 peppe
drwxr-xr-x  28 sabatini thch         3072 Mar  4 13:44 sabatini
drwxrwxr-x   2 root     nobody       1024 May 11  1998 samba
drwx------   2 tuser    tuser        1024 Mar  4 13:42 tuser
$
```

Files e Directories

- Un list della directory radice (da un Linux RedHat 5.1):

```
$ ls -l /
total 68
drwxr-xr-x   2 root     root         2048 Aug 14  1998 bin
drwxr-xr-x   2 root     root         1024 Feb 16 10:34 boot
drwxr-xr-x   3 root     root        21504 Mar  2 11:02 dev
drwxr-xr-x   4 root     root        16384 Jan  1  1970 dos
drwxr-xr-x  24 root     root         3072 Mar  4 13:01 etc
lrwxrwxrwx   1 root     root            3 Aug 13  1998 home -> /u1
drwxr-xr-x   4 root     root         2048 Aug 10  1998 lib
drwxr-xr-x   2 root     root        12288 Aug 10  1998 lost+found
drwxr-xr-x   2 root     root         1024 May  7  1998 misc
drwxr-xr-x   4 root     root         1024 Aug 10  1998 mnt
dr-xr-xr-x   5 root     root            0 Mar  2 13:11 proc
drwxr-xr-x   3 root     root         1024 Feb 28 04:24 root
drwxr-xr-x   3 root     root         2048 Oct 16 12:16 sbin
drwxrwxrwt   8 root     root         1024 Mar  4 11:11 tmp
drwxr-xr-x  10 root     root         1024 Mar  3 13:23 u1
drwxr-xr-x   4 root     root         1024 Aug 24  1998 u2
drwxr-xr-x  20 root     root         1024 Mar  2 10:01 usr
drwxr-xr-x  16 root     root         1024 Mar  2 10:02 var
$
```

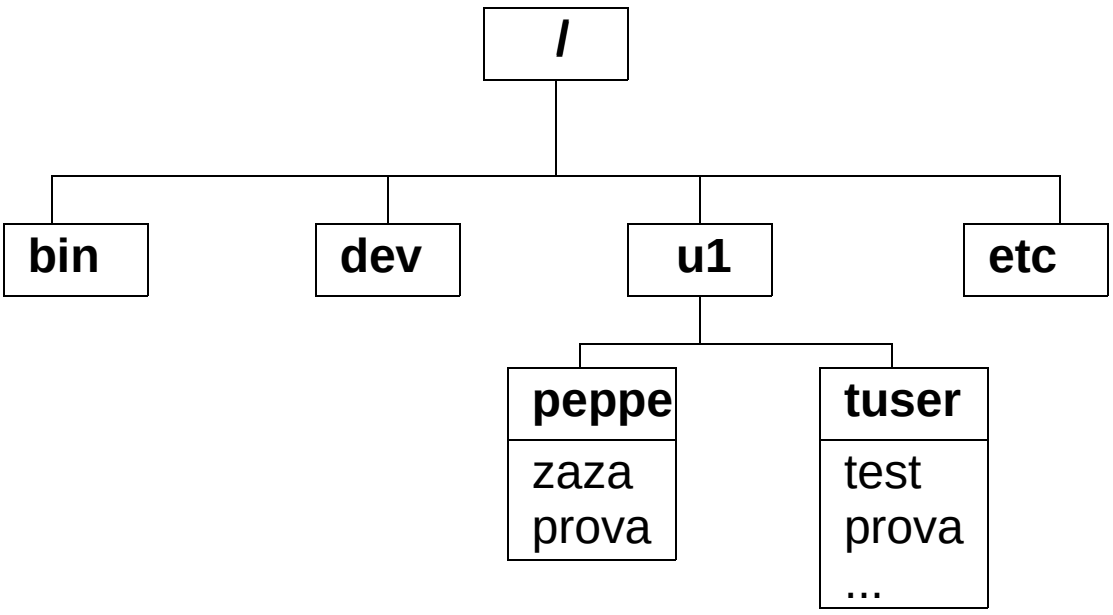
- Il **pathname** rappresenta il nome completo di un file e lo identifica **univocamente**.

Files e Directories

- Comandi come **cat** o **ls** possono agire tanto su pathnames che su filenames: in effetti è una caratteristica della **primitiva di sistema** *open()*.

```
$
$ cat /u1/tuser/prova
... e questo e' il file prova
$
```

- Il File System è organizzato come un *albero rovesciato*.



Files e Directories

- I **pathnames** iniziano a diventare interessanti quando i files su cui state lavorando non si trovano nella *current directory*.

  Per esempio i vostri colleghi possono accedere al file “test” con il comando:

  cat /u1/tuser/test

  Analogamente potreste accedere al file “prova” con il comando: **cat /u1/peppe/prova**.

- Da notare i due significati del carattere “***/***”.

  Indica la **root** directory e funge da separatore tra i filenames componenti di un pathname.

- Il prossimo passo è quello di cambiare la propria *current directory*: il comando **cd**

Files e Directories

(*change directory)* svolge esattamente questa funzione.

```
$
$ cd /u1/ftp
$
$ pwd
/u1/ftp
$
$ cd
$
$ pwd
/home/tuser
$
```

- Partendo dalla **home directory** dell'utente "tuser" (/u1/tuser) il comando cd ci ha permesso di spostarci nella directory /u1/ftp.

  Il comando cd privo di argomenti ci ha fatto ritornare nella home directory (/home/tuser equivalente a /u1/tuser).

Files e Directories

```
$ mkdir esempio
$ ls -la
total 8
drwx------   3 tuser    tuser        1024 Mar  4 16:57 .
drwxr-xr-x  10 root     root         1024 Mar  3 13:23 ..
-rw-rw-r--   1 tuser    tuser          55 Mar  4 13:42 .profile
-rw-rw-r--   1 tuser    tuser          34 Mar  4 13:20 TEST
drwxr-xr-x   2 tuser    tuser        1024 Mar  4 16:57 esempio
-rw-rw-r--   1 tuser    tuser          30 Mar  3 19:20 prova
-rw-rw-r--   1 tuser    tuser          27 Mar  3 19:20 test
-rw-rw-r--   1 tuser    tuser          27 Mar  4 11:52 test2
$ cd esempio
$ pwd
/home/tuser/esempio
$ cd ..
$ pwd
/home/tuser
$ rmdir esempio
$ ls
TEST   prova  test   test2
$ cd .
$ pwd
/home/tuser
$
```

- La directory “**..**” rappresenta la directory *padre* di qualsiasi directory, ovvero la directory che è di un livello più vicina alla radice.

  Invece “**.**” punta **sempre** alla directory stessa.

Files e Directories

- Abbiamo introdotto due nuovi comandi:
  - **mkdir**

    crea una nuova directory
  - **rmdir**

    rimuove una directory esistente (purché vuota)

UNIX File System

- Il File System è uno dei componenti chiave del sistema UNIX: in effetti è stato implementato per primo.

- La maggior parte dei comandi UNIX hanno come *target* un file e buona parte dei suoi sottosistemi possono essere controllati attraverso dei files (anche una stampante o una linea di comunicazione).

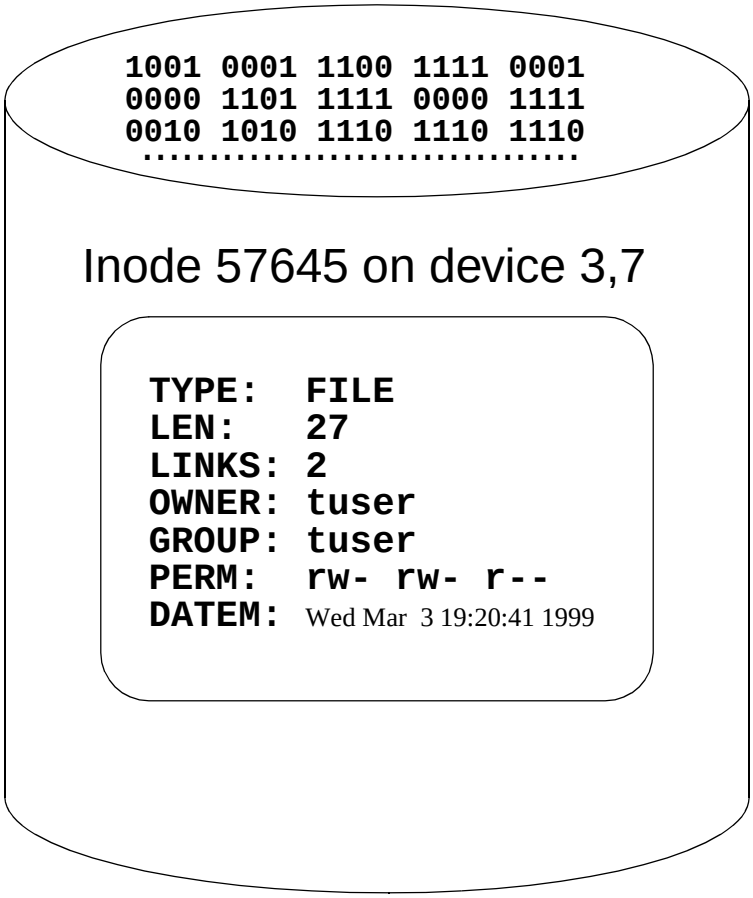
- Per UNIX il contenuto di un file è una "sequenza di bytes". Il sistema non sovrappone strutture al contenuto dei files: si limita a gestire le strutture dati che ne garantiscono l'accesso. MSDOS *interpreta*, per i file di **tipo testo,** i caratteri CR,LF e CTRL-Z.

UNIX File System

- Per UNIX i files sono dei contenitori della **stream** di dati utente. A ciascun file sono collegate delle informazioni caratteristiche:
  - tipo (device, link, directory, ...)
  - lunghezza
  - numero di collegamenti (links)
  - utente proprietario
  - gruppo di appartenenza
  - permessi di accesso
  - date di modifica, uso ed aggiornamento
- Ogni file è univocamente determinato da un numero: l'**i-node** (o inode). Questo numero individua una struttura dati ove sono

UNIX File System

memorizzate le informazioni necessarie alla gestione del file.



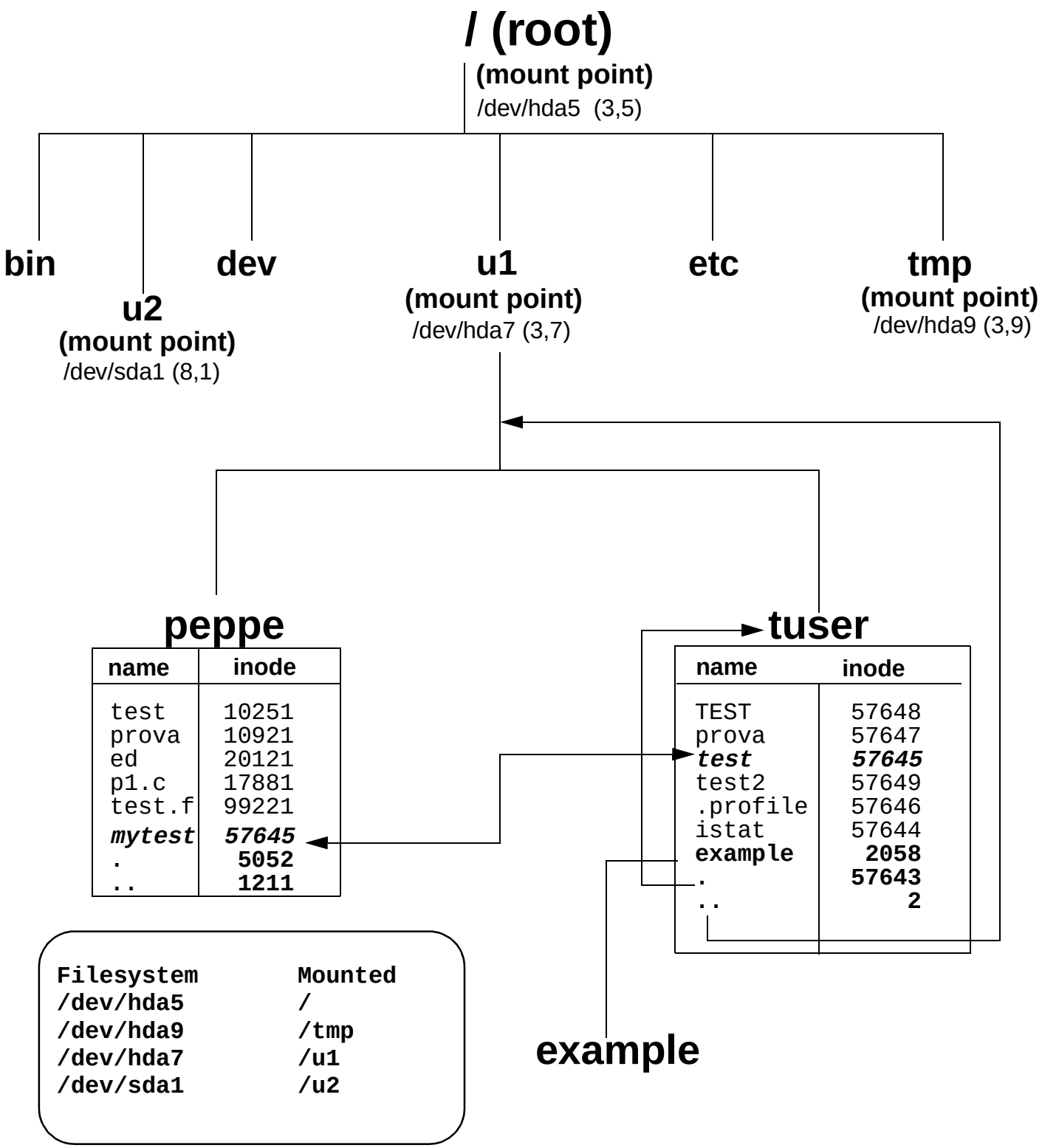
UNIX File System

- Il File System UNIX è organizzato come un *albero rovesciato*. Si parte dalla **root** e si percorre l'albero verso le *foglie*, i files.

- Le directories sono (ancora una volta) files che contengono le associazioni tra i nomi e gli inodes. L'utente può accedere al *contenuto* di una directory, può leggere direttamente il catalogo.

- Più file systems possono essere "*montati*" contemporaneamente. Il file system che funge da **radice** viene montato durante il **bootstrap** (e in genere non può essere *smontato* fino allo **shutdown**).

UNIX File System



UNIX File System

- La directory è un **indice** che permette al sistema di associare un nome al corrispondente inode.

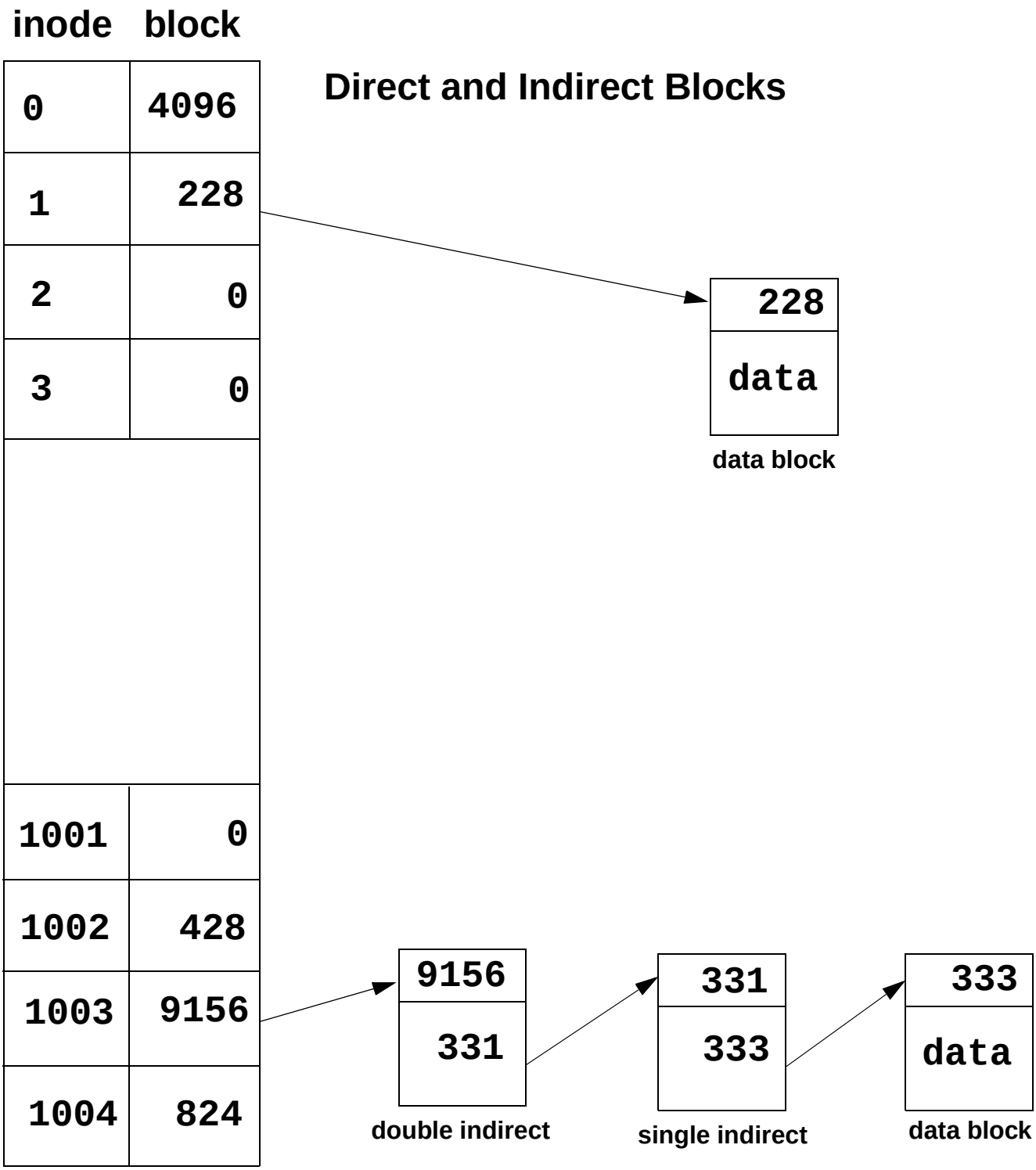
  Ad uno stesso inode possono essere collegati (**linked**) più filenames. Si parla di links:

```
$ ln TEST l_TEST
$
$ ls -li
total 15
  57648 -rw-rw-r--   2 tuser    tuser          34 Mar  4 13:20 TEST
   2058 drwxr-xr-x   2 tuser    tuser        1024 Mar  4 17:56 example
  57644 -rwxr-xr-x   1 tuser    tuser        9215 Mar  4 17:17 istat
  57648 -rw-rw-r--   2 tuser    tuser          34 Mar  4 13:20 l_TEST
  57647 -rw-rw-r--   1 tuser    tuser          30 Mar  3 19:20 prova
  57645 -rw-rw-r--   1 tuser    tuser          27 Mar  3 19:20 test
  57649 -rw-rw-r--   1 tuser    tuser          27 Mar  4 11:52 test2
$
```

  Il comando **ln** permette di *collegare* (di creare un link) tra due filenames.

- I pathnames possono essere **assoluti** (dalla radice) o **relativi** (dalla directory corrente).

UNIX File System



UNIX File System

- I files UNIX sono accessibili solo attraverso un ridotto numero di **system calls**:
  - open
  - read
  - write
  - ioctl
  - lseek
  - close

  sono le primitive “fondamentali”.

- Anche i devices (dischi, porte asincrone e parallele, nastri, etc.) seguono lo stesso modello. Sono accessibili come **files speciali** (in genere sotto /dev).

UNIX File System

- Per ogni file, sono definiti:
  - un ***owner*** o proprietario
  - un ***gruppo*** di appartenenza

    un insieme di *bits* indicatori dei diritti di accesso:

    `--owner--  --group--  --others--`

    `r w x  r w x  r w x`
  - Ciascun bit identifica il *permesso* di leggere o esaminare (r), scrivere o creare/cancellare (w), eseguire o attraversare (x).
  - Possono essere indicati in “ottale” o in forma simbolica (comando **chmod**).

UNIX File System

Ad esempio 555 (cioè 101 101 101) per [r-x r-x r-x]

```
$ chmod 555 test
$ ls -l test
-r-xr-xr-x   1 tuser    tuser          27 Mar  3 19:20 test
$ chmod u=r,g=,o=r test
$ ls -l test
-r-----r--   1 tuser    tuser          27 Mar  3 19:20 test
$ chmod u= test
$ ls -l test
-------r--   1 tuser    tuser          27 Mar  3 19:20 test
$ cat test
cat: test: Permission denied
$ chmod u+r,g+r test
$ ls -l test
-r--r--r--   1 tuser    tuser          27 Mar  3 19:20 test
$ chmod 644 test
$ ls -l test
-rw-r--r--   1 tuser    tuser          27 Mar  3 19:20 test
$ chmod g+w test
$ ls -l test
-rw-rw-r--   1 tuser    tuser          27 Mar  3 19:20 test
$
```

- La primitiva di sistema open() confronta userid e groupid correnti dell'utente e li confronta con proprietario e gruppo del file prima di consentire l'accesso al file.

UNIX File System

- I devices sono files di tipo particolare (categoria dei **files speciali**).

```
$ ls -l /dev/hda[0-9]
brw-rw----   1 root     disk       3,   1 May  5  1998 /dev/hda1
brw-rw----   1 root     disk       3,   2 May  5  1998 /dev/hda2
brw-rw----   1 root     disk       3,   3 May  5  1998 /dev/hda3
brw-rw----   1 root     disk       3,   4 May  5  1998 /dev/hda4
brw-rw----   1 root     disk       3,   5 May  5  1998 /dev/hda5
brw-rw----   1 root     disk       3,   6 May  5  1998 /dev/hda6
brw-rw----   1 root     disk       3,   7 May  5  1998 /dev/hda7
brw-rw----   1 root     disk       3,   8 May  5  1998 /dev/hda8
brw-rw----   1 root     disk       3,   9 May  5  1998 /dev/hda9
$
$ ls -l /dev/sda[0-3]
brw-rw----   1 root     disk       8,   1 May  5  1998 /dev/sda1
brw-rw----   1 root     disk       8,   2 May  5  1998 /dev/sda2
brw-rw----   1 root     disk       8,   3 May  5  1998 /dev/sda3
```

- I due numeri **major** e **minor** identificano in modo univoco il device driver del sistema operativo che gestisce un particolare device connesso al sistema. Interessanti sono /dev/**zero** e /dev/**null.**

UNIX File System

- Le directories sono **files speciali** che possiedono le seguenti proprietà:
  - possono essere lette (non sotto Linux: “**L**INUX **I**s **N**ot **U**ni**X**”) come i files ordinari;
  - possono essere *scritte* solo usando opportuni comandi (come mkdir, rmdir, rm -r, etc.);
  - i permessi hanno un significato diverso:
    - **r** è possibile elencare il contenuto
    - **w** è possibile creare o cancellare files
    - **x** è possibile attraversare la directory.
- Ancora una volta il comando ls può essere di aiuto per comprendere la struttura e le caratteristiche delle directories:

```
$ ls -1i
  34956 doc1.1
  34957 doc1.2
  34958 doc2.2
```

UNIX File System

Il flag **-i** del comando ls permette evidenziare la fondamentale struttura di **indice** della directory in esame. In pratica per ogni filename è memorizzato l'**inode** che identifica il file.

- È abbastanza facile (se non è già disponibile) implementare un comando che estrae da un inode le informazioni in esso contenute:

```
$ istat doc1.1
Inode 34956 on device 3/7       File
Protection: rw-r--r--
Owner: 501(tuser)               Group: 501(tuser)
Link count:     1       Length 1566 bytes

Last updated:   Tue Mar 16 11:16:46 1999
Last modified:  Fri Mar  5 19:41:39 1999
Last accessed:  Tue Mar 16 11:16:19 1999
```

UNIX File System

- In un sistema UNIX in cui le directories possono essere lette come files ordinari (per esempio AIX 4.2.1):

```
rs5 /rs5/u0/peppe/example(40)-> ls -1i
30721 doc1.1
30722 doc1.2
30723 doc2.2
```

```
rs5 /rs5/u0/peppe/example(41)-> od -d .
0000000  30720 11776 00000 00000 00000 00000 00000 00000
0000020  01024 11822 00000 00000 00000 00000 00000 00000
0000040  30721 25711 25393 11825 00000 00000 00000 00000
0000060  30722 25711 25393 11826 00000 00000 00000 00000
0000100  30723 25711 25394 11826 00000 00000 00000 00000
0000120
```

```
rs5 /rs5/u0/peppe/example(42)-> od -c .
0000000    x  \0   .  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000020  004  \0   .   .  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0000040    x 001   d   o   c   1   .   1  \0  \0  \0  \0  \0  \0  \0  \0
0000060    x 002   d   o   c   1   .   2  \0  \0  \0  \0  \0  \0  \0  \0
0000100    x 003   d   o   c   2   .   2  \0  \0  \0  \0  \0  \0  \0  \0
0000120
```

risulta evidente la *struttura* estremamente semplice dell'indice contenuto in una directory.

UNIX FIle System

- Alcune note sui permessi di una directory:
  - **non è** il proprietario di un file che determina se un file può essere cancellato, ma l'owner della directory che lo contiene!
  - **non è** il permesso di un file che determina se un file può essere cancellato, ma sono i permessi della directory che lo contiene!

```
$ ls -la .
total 3
drwxr-xr-x   2 tuser    tuser        1024 Mar 16 17:17 .
drwxr-xr-x   7 tuser    tuser        1024 Mar 16 11:16 ..
----------   1 root     root            9 Mar 16 17:17 rootfile
$ ls > rootfile
bash: rootfile: Permission denied
$ rm rootfile
rm: remove 'rootfile', overriding mode 0000? y
$ ls -l
total 0
$
```

  - il permesso **x** di una directory permette di attraversare una directory che non ha permessi di lettura per raggiungere una subdirectory che possiede i permessi corretti:

UNIX File System

```
$ ls -ld example
d--x--x--x   3 tuser    tuser        1024 Mar 16 17:51 example
$ ls -l example
ls: example: Permission denied
$ ls -ld example/ex1
drwxr-xr-x   2 tuser    tuser       1024 Mar 16 17:51 example/ex1
$ ls -l example/ex1
total 148
-rw-r--r--   1 tuser    tuser        6593 Mar 16 17:51 doc2.1
-rw-r--r--   1 tuser    tuser       11949 Mar 16 17:51 doc2.2
-rw-r--r--   1 tuser    tuser       57850 Mar 16 17:51 doc2.3
-rw-r--r--   1 tuser    tuser       71365 Mar 16 17:51 doc2.4
```

Si può facilmente notare che il permesso **r** permette di elencare il contenuto di una directory, mentre il permesso **x** permette di attraversarla anche se sono negati i permessi di lettura.

- La sicurezza, in un sistema UNIX, dipende in modo *critico* da un corretto uso dei permessi: immaginate il risultato del comando "**rm -rf /**", utilizzato da un utente senza i privilegi del super-user, su un file system con directories con permessi come **rwx rwx rwx**!

Comandi utili

- Aggiungiamo qualche altro comando utile ai nostri strumenti. Il comando **wc** (*word count)* conta il numero di righe, di parole e di caratteri presenti in un file.

```
$
$ cat storia.unix
Il gruppo MULTICS si trova temporanemente
senza un sistema di calcolo interattivo.

Ken Thompson e Dennis Ritchie disegnano
un nuovo File System che si evolvera'
in una delle prime versioni del File System UNIX.

Thompson scrive un programma che simula
il File System proposto e codifica un
Kernel preliminare per un GE 645.
$
```

- Applichiamo **wc** al file storia.unix:

```
$
$ wc storia.unix
     10      53     329 storia.unix
$
```

Comandi utili

Il comando ci informa che il file storia.unix contiene:

- 10 linee
- 53 parole
- 329 caratteri

La definizione di “parola” è molto semplice: qualsiasi stringa che non contenga un blank, un tab (0x09) o un ritorno a capo (line feed 0x0a).

wc può operare su più files fornendo i totali:

```
$
$ wc prova test
      1       7      30 prova
      1       6      27 test
      2      13      57 total
$
```

Comandi utili

- Il comando **grep**: ricerca nei files le righe di testo che contengono un *pattern* (il nome proviene dal comando g/*regular-expression*/p di **ed**).

```
$
$ grep Thompson storia.unix
Ken Thompson e Dennis Ritchie disegnano
Thompson scrive un programma che simula
$
```

Ricerchiamo le linee in cui non compare la stringa “Thompson”. Il flag **-v** inverte il senso della ricerca.

```
$ grep -v Thompson storia.unix
Il gruppo MULTICS si trova temporanemente
senza un sistema di calcolo interattivo.

un nuovo File System che si evolvera’
in una delle prime versioni del File System UNIX.

il File System proposto e codifica un
Kernel preliminare per un GE 645.
```

Comandi utili

Alcuni dei comandi più utilizzati

Comando	**Funzione**
ls	Elenca i nomi dei files nella current directory
ls *filenames*	Elenca i files dati come argomenti
ls -t	Elenca in ordine di data: prima il più recente
ls -l	Elenco dettagliato
ls -u	Elenca prima il file acceduto più recentemente
ls -r	Elenco in ordine rovesciato: -rt, -rlt, etc
ed *filename*	edit di linea del file in argomento
cp *file1 file2*	copia *file1* su *file2*, sostituisce *file2* se esiste già
mv *file1 file2*	muove *file1* su *file2*, sostituisce *file2* se esiste già
rm *filenames*	rimuove i files dati come argomenti
cat *filenames*	visualizza il contenuto dei files forniti come argomenti
wc *filenames*	conta le righe, le parole e i caratteri dei files forniti come argomenti

Comandi utili

Alcuni dei comandi più utilizzati

Comando	**Funzione**
wc -l *filenames*	conta le righe dei files
grep *pattern filenames*	stampa ogni linea in cui compare *pattern*
grep -v *pattern filenames*	stampa ogni linea in cui **non** compare *pattern*
sort *filenames*	dispone in ordine alfabetico le linee dei files forniti in argomento
tail *filename*	stampa le ultime 10 linee del file
tail -n	stampa le ultime n linee del file
cmp *file1 file2*	stampa la posizione della prima differenza tra i files
diff *file1 file2*	stampa tutte le differenze tra i files

La Shell

- Abbiamo visto che il punto culminante dell'operazione di login è rappresentato dalla comparsa sul terminale del **prompt**, nel nostro caso il carattere "**$**".

  È il segnale che il sistema è pronto ad accettare i nostri comandi.

- Non state comunicando direttamente con il kernel: i comandi vengono *interpretati* da un programma, un *interprete dei comandi* (simile al COMMAND.COM di MSODS o al CMD.EXE di OS/2 o Windows/NT).

- I programmi che svolgono questa funzione vengono denominati, in ambiente UNIX, **shell**.

La Shell

- La *shell* è un normalissimo programma come *date* o *who,* sebbene svolga un ruolo fondamentale nell'interazione con il sistema.

  Il suo ruolo è quello di tradurre le stringhe, immesse dall'utente, in operazioni che il sistema operativo sia in grado di eseguire.

- Le sue principali funzioni sono:
  - Espansione dei *filenames*
  - Ridirezione dell'input-output
  - Personalizzazione dell'ambiente.

Espansione dei Filenames

- Accade spesso di lavorare con files i cui filenames possono venir raggruppati in qualche modo. Per esempio i sorgenti di un programma C termineranno tutti con *.c* oppure i capitoli di un libro inizieranno tutti con *cap*.

- Supponiamo che il programma C su cui state lavorando si trovi nella subdirectory **source** della vostra home directory e che i files che lo compongono siano *progr.c subr1.c subr2.c subr3.c incl1.h incl2.h*. Nella subdirectory **doc** si trova invece la documentazione relativa al programma suddivisa nei files *doc1.1 doc1.2 doc2.1 doc2.2 doc2.3 doc2.4*, ove i numeri si riferiscono a capitoli e sezioni.

Espansione dei Filenames

- Un modo per stampare la documentazione relativa al secondo capitolo potrebbe essere:

```
$
$ pr doc2.1 doc2.2 doc2.3 doc2.4
$
```

È abbastanza chiaro che se il numero dei capitoli è appena più grande di 4 l'operazione inizia a diventare macchinosa.

- La shell viene in aiuto svolgendo gran parte del lavoro al nostro posto. La soluzione consiste nell'usare i caratteri **wildcard** o **metacaratteri.**

```
$
$ pr doc2.*
$
```

Espansione dei Filenames

Volendo stampare tutta la documentazione:

```
$
$ pr doc*
$
```

- I metacaratteri sostituiscono i filenames, o parti di questi e consentono di creare una corrispondenza tra l'abbreviazione e i nomi completi. Il carattere "*****" può sostituire qualunque stringa, compresa la stringa nulla.

  Il carattere "**?**" sostituisce qualsiasi singolo carattere.

  Le parentesi quadre "**[ ]**" permettono di rappresentare un insieme di caratteri. Per esempio **[a-z]** fa match con qualsiasi lettera minuscola dell'alfabeto.

Espansione dei Filenames

- Il punto cruciale è che l'abbreviazione del filename non è una proprietà del comando **pr**, ma una funzione svolta dalla **shell**.

  La shell interpreta il carattere "*****" come *qualsiasi stringa di caratteri* cosicchè "**doc***" è un pattern che può essere espanso in tutti i filenames che iniziano con la stringa "doc".

- La shell (non il sistema operativo) crea la lista, in ordine alfabetico, dei filenames che fanno *match* con il pattern e la trasforma negli argomenti del comando.

- È una delle differenze più importanti tra UNIX ed MSDOS. Il COMMAND.COM dell'MSDOS non svolge questa funzione per tutti i comandi,

Espansione dei Filenames

ma solo per un ristretto insieme di comandi interni (come dir).

- Si possono utilizzare i **wildcard** con qualsiasi comando per operare su una lista di filenames.

```
$
$ wc doc2*
    202    1061    6593 doc2.1
    366    1923   11949 doc2.2
   1772    9310   57850 doc2.3
   2186   11485   71365 doc2.4
   4526   23779  147757 total
$
```

- Il comando **echo** si rivela particolarmente utile per sperimentare con l'espansione dei metacaratteri. L'unico compito svolto da **echo** consiste nel produrre l'elenco dei propri argomenti.

Espansione dei Filenames

```
$
$ echo Lista di Argomenti
Lista di Argomenti
$
```

- Gli argomenti possono venir generati dall'espansione dei metacaratteri:

```
$
$ echo doc2.*
doc2.1 doc2.2 doc2.3 doc2.4
$
```

La shell ha espanso l'abbreviazione "**doc2.***" nei filenames che corrispondono al secondo capitolo del libro generando gli argomenti "**doc2.1 doc2.2 doc2.3 doc2.4**" per il comando **echo**, esattamente come ha fatto per il comando **wc** nell'esempio precedente.

Espansione dei Filenames

- Se volessimo elencare tutti i files della directory corrente in ordine alfabetico senza usare il comando ls:

```
$
$ echo *
doc1.1 doc1.2 doc2.1 doc2.2 doc2.3 doc2.4
$
```

così come il comando "**pr ***" stamperà tutti i files del libro (in ordine alfabetico).

- **Attenzione**: comandi come **"rm *"** rimuoveranno tutti i files contenuti nella directory corrente senza chiedere conferma.

- Qualche esperimento con i metacaratteri ed il comando echo:

Espansione dei Filenames

- l'uso di "**?**":

```
$
$ echo doc?.2
doc1.2 doc2.2
$
```

- l'uso delle parentesi quadrate "**[ ]**":

```
$
$ echo doc2.[1234]
doc2.1 doc2.2 doc2.3 doc2.4
$
$ echo doc2.[1-4]
doc2.1 doc2.2 doc2.3 doc2.4
$
$ echo doc?.[1-4]
doc1.1 doc1.2 doc2.1 doc2.2 doc2.3 doc2.4
$
$ echo doc[12].[1-4]
doc1.1 doc1.2 doc2.1 doc2.2 doc2.3 doc2.4
$
```

- I metacaratteri trovano corrispondenza solo in files che esistono. In particolare non si

Espansione dei Filenames

possono creare nuovi files usando i metacaratteri.

Per esempio “**mv doc1.* cap1.***” non vi permetterà di rinominare i files sia perché cap1.* non si riferisce a files che esistono già ed anche perché il comando UNIX mv non opera come il comando MSDOS rename.

- In alcuni sistemi UNIX esiste un limite massimo alla lunghezza di un comando. Se troppi files soddisfano il criterio di selezione, la lista degli argomenti (che fa parte del comando) diventa troppo lunga e si incorre in un errore.

Espansione dei Filenames

- I metacaratteri possono essere utilizzati sia nei **pathnames** così come nei **filenames**. Se nella directory /u1/tuser esiste una copia del libro nella directory "**doc.copy**" si possono elencare i files di doc e doc.copy con un solo comando:

```
$
$ ls tuser/doc*/doc1.[12]
tuser/doc.copy/doc1.1  tuser/doc/doc1.1
tuser/doc.copy/doc1.2  tuser/doc/doc1.2
$
```

- **Puntualizziamo ancora una volta**: è la shell che **interpreta** i metacaratteri.

  Qualunque comando invocato dalla shell riceve gli argomenti prodotti dall'espansione dei metacaratteri.

Espansione dei Filenames

- In alcune situazioni potrebbe essere necessario impedire che la shell espanda i metacaratteri: includendo gli argomenti tra apici ’ o doppi apici **"** si ottiene questo risultato.

```
$
$ echo doc?.1
doc1.1 doc2.1
$
$ echo "doc?.1"
doc?.1
$
```

- I due metodi sono differenti: i singoli apici impediscono alla shell l’interpretazione di **tutti** i metacaratteri; i doppi apici le permettono di continuare ad interpretare i caratteri **$**, **‘**, ****.

  Usate i singoli apici ’ a meno che non vogliate esplicitamente che la shell interpreti **$**, **‘**, ****.

Espansione dei Filenames

- Il **backslash ** impedisce che il carattere seguente (e solo quello) venga interpretato dalla shell come un metacarattere:

```
$
$ echo doc?.1
doc1.1 doc2.1
$
$ echo doc\?.1
doc?.1
$
```

Ridirezione dell'Input/Output

- La shell permette di sostituire il terminale con un **file** sia per l'input che per l'output.

  Il comando **ls** elenca i files della directory corrente a terminale, mentre **ls > elenco** produce la stessa lista di files nel file *elenco*.

- Il simbolo **>** ordina alla shell di *connettere* lo **Standard Output** di un comando con il ***filename*** che segue il simbolo stesso.

  Il file verrà creato se non esiste o sarà sovrascritto se esisteva già.

  Nulla viene prodotto sul terminale.

- Questo *disegno* permette di usare cat per **concatenare** i files**.**

Ridirezione dell'Input/Output

```
$
$ wc doc2.*
    202    1061    6593 doc2.1
    366    1923   11949 doc2.2
   1772    9310   57850 doc2.3
   2186   11485   71365 doc2.4
   4526   23779  147757 total
$
$ cat doc2.* > cap2
$
$ wc cap2
   4526   23779  147757 cap2
$
```

- Il simbolo **>>** svolge lo stesso compito, ma piuttosto che riscrivere il file (se esiste già), appende alla fine del file l'output, del comando.

```
$
$ cat test
Questo e' il file test ...
$ cat prova
... e questo e' il file prova
$ cat output
File di output ...
$ cat test prova >> output
$ cat output
File di output ...
Questo e' il file test ...
... e questo e' il file prova
$
```

Ridirezione dell'Input/Output

- Nello stesso modo il simbolo **<** ordina alla shell di *connettere* lo **Standard Input** (la vostra tastiera) con il ***filename*** che segue il simbolo stesso.

  Volendo contare il numero di utenti che sono login nel sistema:

```
$
$ who >temp
$ wc -l <temp
       7
$
```

  oppure contare il numero di files presenti nella current directory:

```
$
$ ls > temp
$ wc -l temp
      13 temp
$
```

Ridirezione dell'Input/Output

o ancora individuare se un vostro collega è login nel sistema:

```
$
$ who > temp
$ grep sabatini < temp
sabatini tty1     Mar  2 11:02
$
```

- I comandi "**cat test**" e **"cat <test"** eseguono lo stesso compito, ma esiste una importante differenza nel modo in cui lo fanno:
  - nel primo caso **test** viene passato al comando **cat** come argomento senza che sia avvenuta nessuna interpretazione
  - nel secondo caso a cat non viene passato nessun argomento; si dà semplicemente il caso che cat sia stato disegnato per leggere dallo Standard Input quando non gli vengono forniti argomenti.

Ridirezione dell'Input/Output

- La maggior parte dei comandi UNIX sono disegnati per operare seguendo questo *schema*:

  "Se l'insieme dei files passati come argomenti è vuoto viene processato lo *Standard Input*"

  È un paradigma da tener ben presente quando si progettano nuovi comandi.

  Ci ha permesso di usare cat come un rudimentale editor (qualche volta può realmente capitare di usarlo proprio così).

Concatenazione di Comandi

- In alcuni degli esempi precedenti abbiamo utilizzato l'output di un programma come input per un altro comando.

  È una situazione che si presenta così di sovente, da aver meritato l'introduzione di uno speciale meccanismo di comunicazione tra i programmi: le **pipelines**.

- Una **pipe** è uno strumento che permette di connettere lo Standard Output di un programma con lo Standard Input di un altro, evitando la necessità di usare un file temporaneo.

  Una **pipeline** è la concatenazione fra due o più programmi attraverso una serie di **pipes**.

Concatenazione di Comandi

- La shell crea una **pipe** tutte le volte che incontra il metacarattere “**|**” (la barra verticale).

```
$
$ who | wc -l
      7
$
$ ls | wc -l
     13
$
$ who | grep sabatini
sabatini tty1     Mar  2 11:02
$
```

- Concatenando i comandi per mezzo delle pipelines si possono risolvere una grande varietà di problemi riutilizzando ciò che è già disponibile senza essere costretti ad implementare (e testare) nuovi programmi.

Concatenazione di Comandi

Ciò richiede che i comandi base siano ragionevolmente *semplici* ed osservino delle regole che permettano loro di **cooperare**.

- I programmi di una pipeline girano **contemporaneamente** (non in sequenza) e ciascuno legge dallo Standard Input man mano che i caratteri si rendono disponibili: nelle pipelines possono essere presenti programmi che interagiscono con l'utente.

- Nessuno dei programmi di una pipeline si rende conto della ridirezione implicita nel meccanismo della pipeline. È la **shell** (ancora una volta) che si occupa di predisporre le cose in modo opportuno.

Concatenazione di Comandi

- Non ci sono limiti al numero di comandi presenti in una pipeline (a parte la lunghezza massima della linea di comando della shell).

  La complessità del comando ottenuto è limitata solo dalla nostra abilità e fantasia.

- Il comando che segue ottiene (in un formato adatto alla stampa) l'elenco dei files appartenenti a *root* nella directory /tmp, ordinati per dimensione (dal più grande) e lo memorizza nel file *toremove*:

```
ls -l /tmp | grep root | sort +4nr | pr > toremove
```

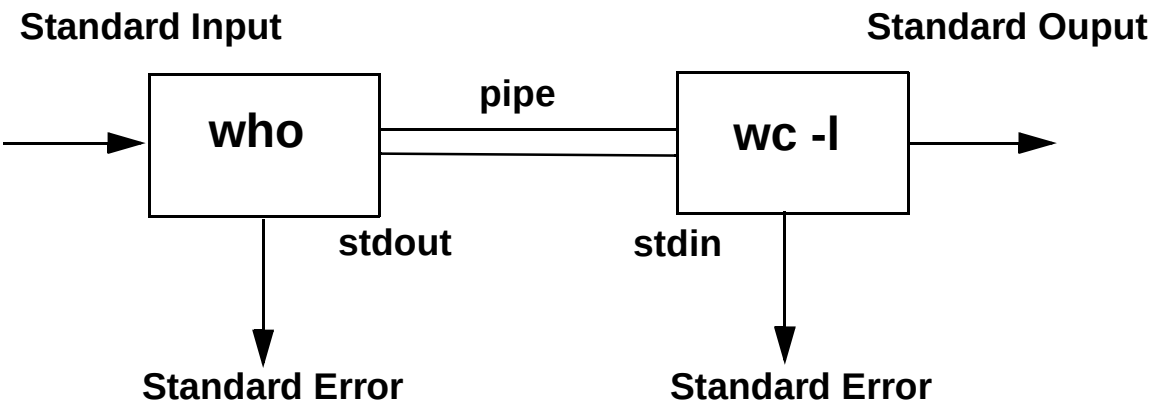
```
99-03-06 12:00                                              Page    1


-rw-r--r--   1 root     root       512000 Mar  6 11:59 test
-rw-r--r--   1 root     root       339744 Mar  1 16:06 log
drwxr-xr-x   2 root     root        12288 Aug 10  1998 lost+found
-rw-r--r--   1 root     root          379 Mar  6 11:59 list
```

Concatenazione di Comandi

- In una semplice immagine il meccanismo di pipe:

who | wc -l



Processi

- Ogni volta che si chiede alla shell di eseguire un comando (con certe eccezioni) ciò conduce alla creazione di un **processo** ed al caricamento in memoria di un programma.

- Un **processo** UNIX è:
  - un programma in esecuzione
  - uno spazio di indirizzi *virtuali*
  - una unità nota al *dispatcher*, un *task* (nei sistemi non multithreaded)

- Tutti i processi sono in esecuzione contemporaneamente attraverso il meccanismo del *time-sharing*.

Processi

- Due comandi eseguiti in sequenza:

```
$
$ date; who
Sat Mar  6 12:47:11 CET 1999
sabatini tty1     Mar  2 11:02
tuser    ttyp0    Mar  6 10:59
$
```

- Un nuovo metacarattere “**&**” viene interpretato dalla shell come l’ordine di eseguire in **background** la linea di comando.

```
$
$ find /u1 -user tuser -type f 2>/dev/null | \
> xargs ls -l | \
> sort +4nr > el &
[1] 13285
$
```

Il comando andrà in esecuzione e noi saremo liberi di svolgere un altro compito. Il numero stampato dalla shell è il **process-id** e può

Processi

essere usato per riferirsi allo specifico processo.

Se una pipeline viene lanciata in background viene stampato solo il process-id dell'ultimo comando.

- È importante distinguere tra processo e programma. Ogni volta che il programma **find** viene lanciato in esecuzione viene creato un nuovo processo. Possono esistere diverse *istanze* dello stesso programma che girano simultaneamente nel sistema.

- Il comando **wait** attende che **tutti** i processi lanciati in background abbiano terminato l'esecuzione.

Processi

- Si può ottenere l'elenco dei processi in esecuzione con il comando **ps** (process status):

```
$
$ ps
  PID TTY STAT TIME COMMAND
12831  p0 S    0:00 /bin/login -h simbad.thch.unipg.it -p
12832  p0 S    0:01 -bash
13283  p0 R    0:00 find /u1 -user tuser -type f
13284  p0 S    0:00 xargs ls -l
13285  p0 S    0:00 sort +4nr
13286  p0 R    0:00 ps
$
```

PID è il process-id, **TTY** è il terminale ad esso associato, **TIME** è il tempo di CPU usato (in minuti e secondi).

- Tutti i processi di *proprietà* di un utente possono essere terminati con il comando **kill**.

Processi

Per esempio il comando **kill 13283** terminerebbe il comando “find” dell’esempio precedente.

E, dopo qualche tempo, ecco il risultato:

```
$ cat el
-rw-r--r--   1 tuser    tuser      147757 Mar  6 11:23 /u1/tuser/doc/cap2
-rw-r--r--   1 tuser    tuser       71365 Mar  5 19:43 /u1/tuser/doc.copy/doc2.4
-rw-r--r--   1 tuser    tuser       71365 Mar  5 19:43 /u1/tuser/doc/doc2.4
-rw-r--r--   1 tuser    tuser       57850 Mar  5 19:43 /u1/tuser/doc.copy/doc2.3
-rw-r--r--   1 tuser    tuser       57850 Mar  5 19:43 /u1/tuser/doc/doc2.3
-rw-r--r--   1 tuser    tuser       11949 Mar  5 19:42 /u1/tuser/doc.copy/doc2.2
-rw-r--r--   1 tuser    tuser       11949 Mar  5 19:42 /u1/tuser/doc/doc2.2
-rwxr-xr-x   1 tuser    tuser        9215 Mar  4 17:17 /u1/tuser/istat
-rw-r--r--   1 tuser    tuser        6593 Mar  5 19:42 /u1/tuser/doc.copy/doc2.1
-rw-r--r--   1 tuser    tuser        6593 Mar  5 19:42 /u1/tuser/doc/doc2.1
-rw-r--r--   1 tuser    tuser        3461 Mar  5 19:41 /u1/tuser/doc.copy/doc1.2
-rw-r--r--   1 tuser    tuser        3461 Mar  5 19:41 /u1/tuser/doc/doc1.2
-rw-r--r--   1 tuser    tuser        2911 Mar  5 20:06 /u1/tuser/.bash_history
-rw-r--r--   1 tuser    tuser        1566 Mar  5 19:41 /u1/tuser/doc.copy/doc1.1
-rw-r--r--   1 tuser    tuser        1566 Mar  5 19:41 /u1/tuser/doc/doc1.1
-rw-r--r--   1 tuser    tuser         379 Mar  6 12:00 /u1/tuser/toremove
-rw-r--r--   1 tuser    tuser         329 Mar  4 19:27 /u1/tuser/storia.unix
-rw-r--r--   1 tuser    tuser         275 Mar  6 11:30 /u1/tuser/temp
-rw-r--r--   1 tuser    tuser          76 Mar  6 11:25 /u1/tuser/output
-rw-rw-r--   1 tuser    tuser          55 Mar  4 13:42 /u1/tuser/.profile
-rw-rw-r--   2 tuser    tuser          34 Mar  4 13:20 /u1/tuser/TEST
-rw-rw-r--   2 tuser    tuser          34 Mar  4 13:20 /u1/tuser/l_TEST
-rw-rw-r--   1 tuser    tuser          30 Mar  3 19:20 /u1/tuser/prova
-rw-rw-r--   1 tuser    tuser          27 Mar  3 19:20 /u1/tuser/test
-rw-rw-r--   1 tuser    tuser          27 Mar  4 11:52 /u1/tuser/test2
-rw-r--r--   1 tuser    tuser           0 Mar  6 13:09 /u1/tuser/el
```

Processi

- I processi sono organizzati, a loro volta, in una struttura gerarchica simile a quella dei file systems. I processi generati da shell commands sono **figli** della shell che, a sua volta, è stata creata da un processo associato alla linea di comunicazione che connette il vostro terminale al sistema.

```
  PID TTY STAT TIME COMMAND
12831  p0 S    0:00 /bin/login -h simbad.thch.unipg.it -p
12832  p0 S    0:02  \_ -bash
13366  p0 R    0:00      \_ ps f


  PID  PPID PRI  NI   SIZE  STA TTY TIME COMMAND
12831 12830   0   0   1568   S   p0  0:00 /bin/login
12832 12831  11   0   1196   S   p0  0:02 -bash
13366 12832  14   0    960   R   p0  0:00 ps f
```

Processi

- Cosa sono i comandi?

  Sono *stringhe* che fanno riferimento a **programmi** memorizzati in normali files.

  Un ls della directory ***/*bin** ci permetterà di

```
-rwxr-xr-x   1 root     root         9032 Apr 24  1998 cat
-rwxr-xr-x   1 root     root        10276 Apr 24  1998 chmod
-rwxr-xr-x   1 root     root        23156 Apr 24  1998 cp
-rwxr-xr-x   1 root     root       257388 Apr 27  1998 csh
-rwxr-xr-x   1 root     root        23028 May  9  1998 date
-rwxr-xr-x   1 root     root         6072 May  9  1998 echo
-rwxr-xr-x   1 root     root        68396 Apr 27  1998 ed
-rwxr-xr-x   1 root     root        69444 May 10  1998 grep
-rwxr-xr-x   1 root     root       162628 Apr 27  1998 ksh
-rwxr-xr-x   1 root     root        12768 Apr 24  1998 ln
-rws--x--x   1 root     root        15588 May  4  1998 login
-rwxr-xr-x   1 root     root        29980 Apr 24  1998 ls
-rwxr-xr-x   1 root     root        72364 Jun 23  1998 mail
-rwxr-xr-x   1 root     root         8268 Apr 24  1998 mkdir
-rwxr-xr-x   1 root     root        25236 May  4  1998 more
-r-xr-xr-x   1 bin      root        12324 May  6  1998 ps
-rwxr-xr-x   1 root     root         5388 May  9  1998 pwd
-rwxr-xr-x   1 root     root        10124 Apr 24  1998 rm
-rwxr-xr-x   1 root     root         4884 Apr 24  1998 rmdir
-rwxr-xr-x   1 root     root       302732 Apr 27  1998 sh
-rwxr-xr-x   1 root     root        25632 May  9  1998 stty
-rwxr-xr-x   1 root     root       394676 May  8  1998 vi
```

  riconoscere nei nomi dei files alcuni dei comandi che abbiamo usato fino ad ora.

Processi

- Esistono eccezioni: alcuni comandi vengono *eseguiti* direttamente dalla shell. L'interprete li traduce direttamente in primitive di sistema.

  Sono noti come **comandi interni** (o anche *builtin commands*).

- Un esempio: il comando **cd** (change directory) non potrebbe essere implementato che come comando interno. La *current directory* è una proprietà del processo in esecuzione: la primitiva *chdir()* non può modificare che la *current directory* del processo che la esegue.

- In altri casi il comando viene implementato direttamente dalla shell per motivi di efficienza.

Processi

- Prima di *eseguire* un comando la shell esegue una ricerca nelle directories elencate nella variabile di **environment** (ambiente) **PATH**.

  Una volta trovato il file che corrisponde al nome del comando, la shell invoca il kernel per iniziarne l'esecuzione. Ciò conduce inevitabilmente alla creazione di un nuovo processo che sarà figlio della shell.

- Diamo una occhiata alla variabile **PATH** del nostro utente di test:

```
$
$ echo $PATH
/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin:.
$
```

Metacaratteri

Metacarattere	Funzione
>	*prog* > *file* dirige lo Standard output su file
>>	*prog* >> *file* appende lo Standard Ouput su file
<	*prog* < *file* Standard input da file
|	*p1*|*p2* pipe tra p1 e p2
*	qualsiasi stringa di zero o più caratteri nell'espansione dei filenames
?	qualsiasi singolo carattere
[ccc]	insieme di caratteri: [a-z] o [12-a]
;	terminatore di comando: *p1*;*p2*
`...`	output di un programma trasformato in lista di argomenti (**backquote**)
\	\c interpreta il carattere c alla lettera
"..."	interpreta il comando alla lettera tranne che per $, `, \
&	lancia il comando in background
$var	valore della variabile di ambiente *var*

Che cosa è UNIX?

- Una parola:
  - un trademark registrato
- Un insieme di specifiche:
  - SVID (System V Interface Description)
  - POSIX
  - ...
- Una implementazione:
  - IBM AIX/6000 4.2
  - SunOS 4.1.3
  - FreeBSD 2.2.8
  - ...

Struttura del sistema UNIX

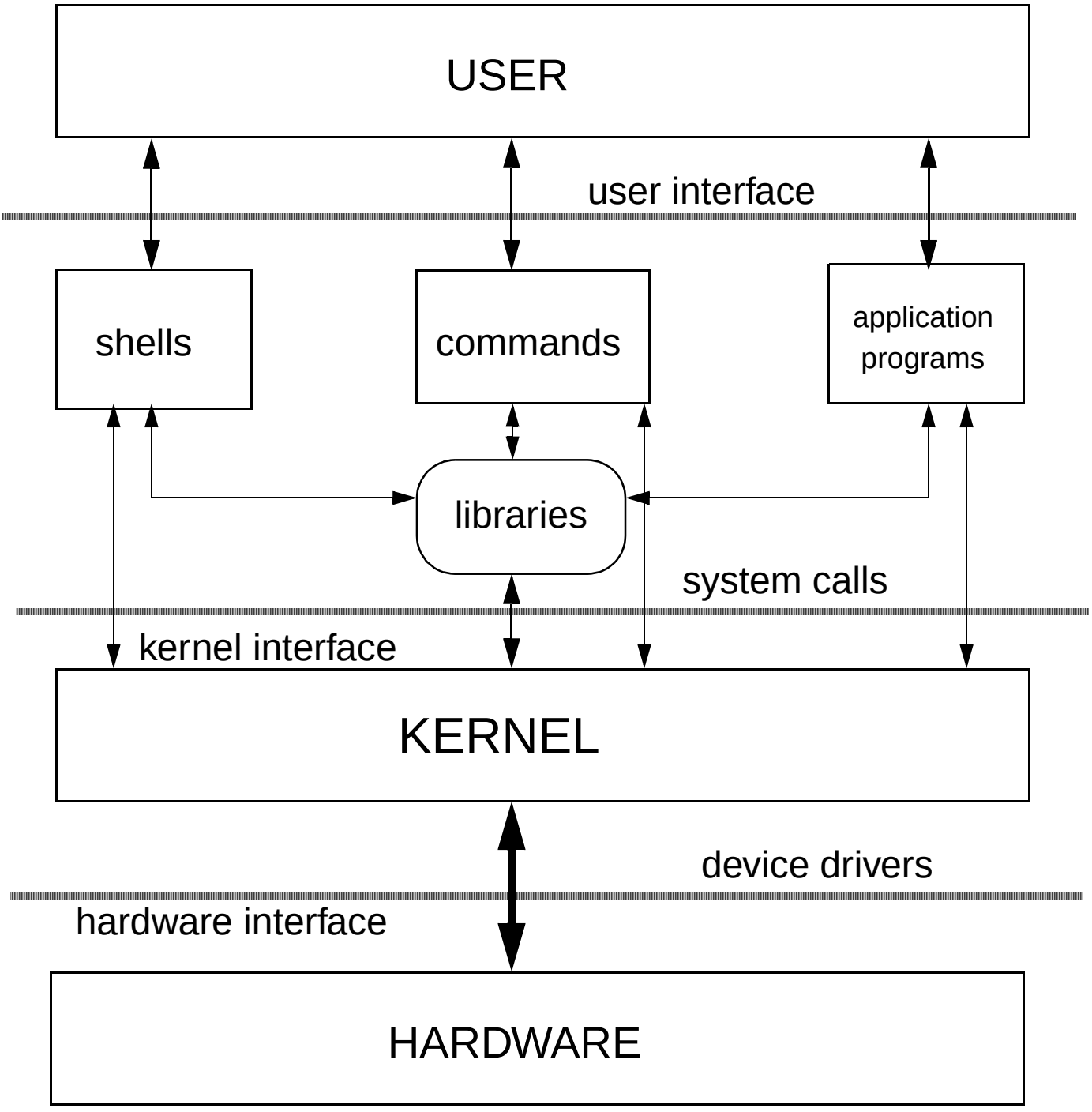
- Il sistema operativo UNIX è costituito da diversi “strati” che implementano funzioni via via più indipendenti dall’hardware.

- Il **kernel** interagisce direttamente con l’hardware (la *bare machine*) e fornisce servizi alle applicazioni.

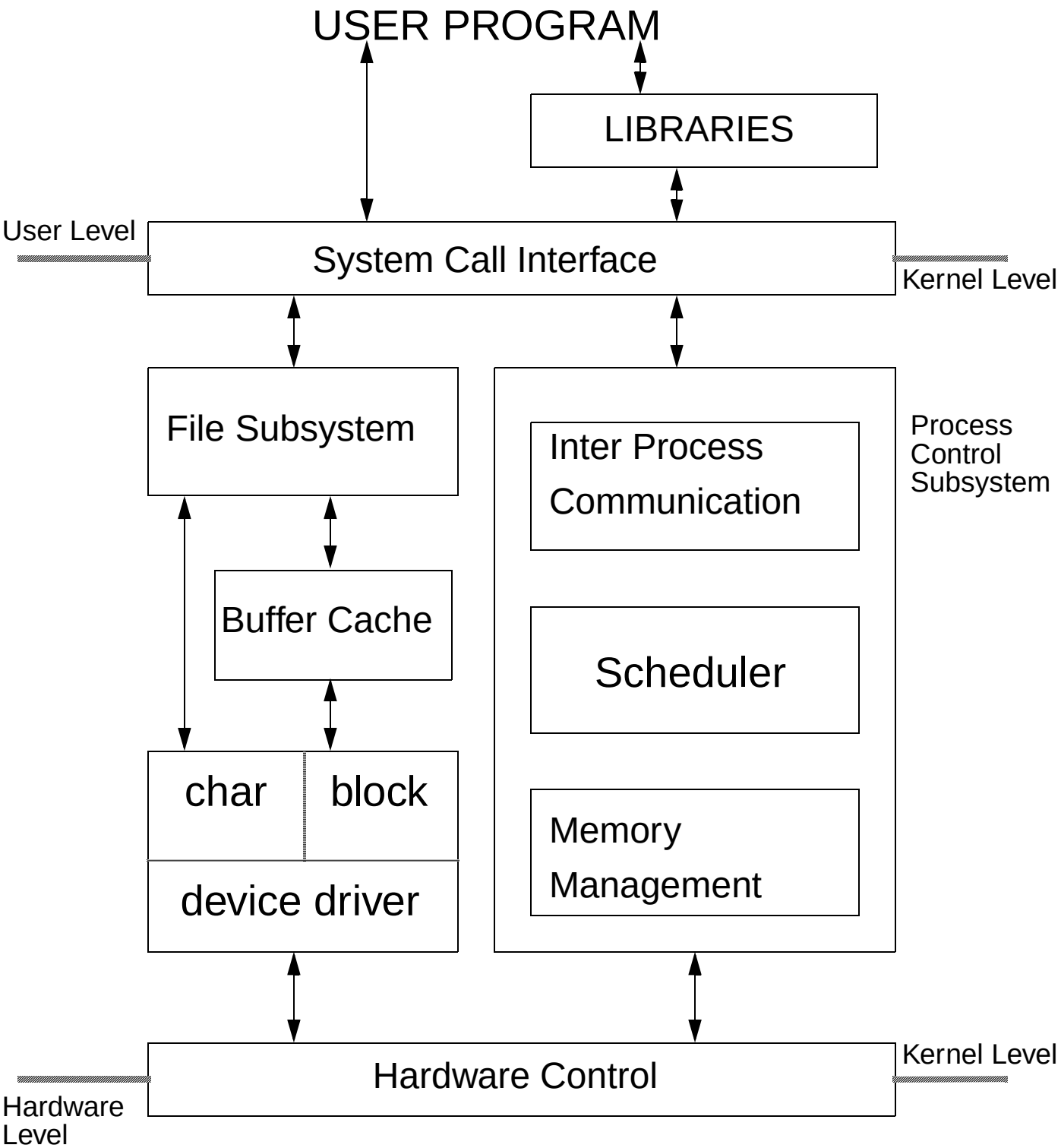
- Le applicazioni interagiscono con il kernel attraverso un insieme predefinito di primitive: le **system calls**.

- UNIX è un sistema operativo **multi-utente** e **multi-tasking**: più utenti possono essere attivi contemporaneamente e ciascuno di loro può far girare simultaneamente più programmi.

Struttura del sistema UNIX



Struttura del sistema UNIX



Processi

- Come abbiamo già visto i processi sono organizzati in una struttura gerarchica: ogni processo è *figlio* di un altro processo, definito *padre*.

- Tutti i processi (contemporaneamente in esecuzione attraverso il meccanismo del *time-sharing*) sono discendenti del primo processo: **init**.

```
$ ps p 1
  PID TTY STAT TIME COMMAND
    1  ?  S    0:02 init
$
```

Ogni processo è identificato univocamente dal suo “process-id”: **pid**.

Il primo processo (non ha padre!) ha, in genere, pid uguale ad **1**.

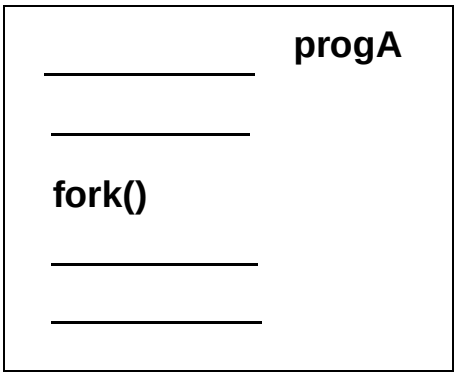
Processi

- I processi realizzano e controllano il multitasking in un sistema UNIX.

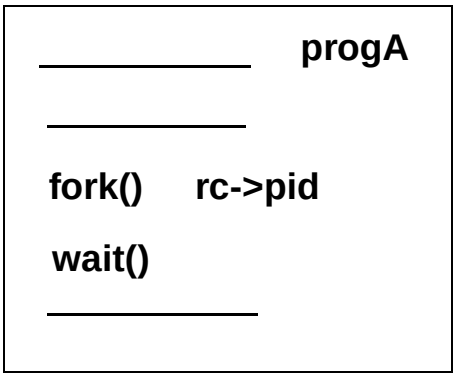
- Mediante l'uso di poche e semplici primitive le applicazioni possono interagire con il kernel per creare, interrompere ed eseguire processi:

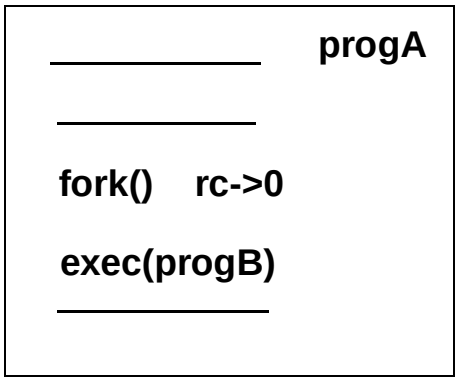
  - fork — creazione
  - exec — esecuzione/load
  - wait — attesa per termine es.
  - kill — invio di un segnale
  - signal — gestione segnali
  - set/getpriority, nice — gestione priorità

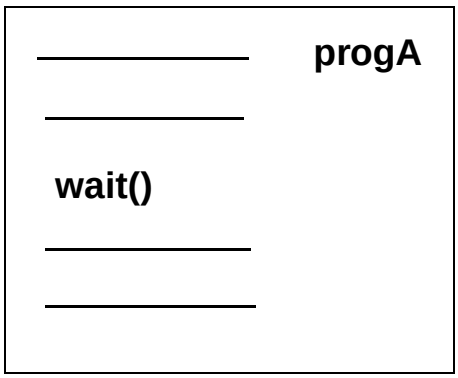
Processi

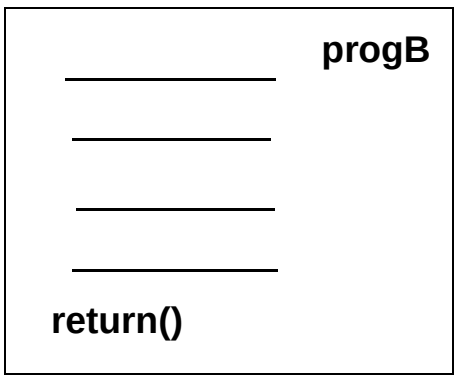


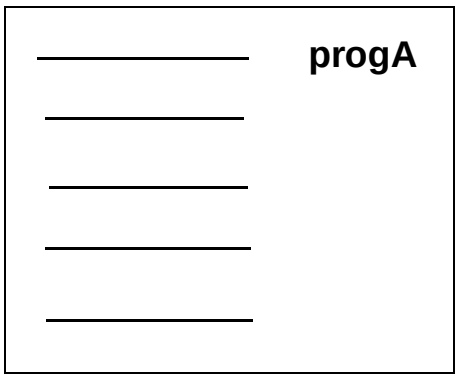

```
rc = fork()
if ( rc != 0 );
    wait()
else
    exec(...);
```











Tutti i processi (tranne init) sono *generati* da una chiamata alla system call fork()

La Shell

- Esistono due filoni principali:
  - Bourne Shell

    la capostipite, **sh**, sviluppata da Steve Bourne, ha avuto successori di successo come **ksh** (Korn Shell), **bash** (GNU Bourne-Again SHell), **zsh** (Z shell).
  - C Shell

    La “C Shell”, **csh**, sviluppata dal Bill Joy (padre anche di **vi**) a Berkeley. Ha implementazioni più moderne come la **tcsh**.
- L’uso di una particolare shell è una *scelta di vita*. Ogni shell ha le sue pecularietà ed il suo linguaggio di programmazione (**script language**).

  La maggior parte delle implementazioni UNIX usa variazioni della Bourne come shell di sistema (esecuzione di /etc/rc, /etc/rc.local).

Environment

- Ogni processo UNIX possiede un **environment**.

  L'environment consiste in una serie di coppie (variabile,valore) mantenute in una *array* all'interno dello spazio di indirizzi virtuali del processo:

  - **getenv** set di una variabile
  - **setenv** get di una variabile
  - **`char **environment`** indirizzo di memoria

- In particolare la shell possiede un environment e lo utilizza per mantenere un insieme di *variabili* che permettono la programmazione e la personalizzazione dell'ambiente.

Environment

- Il comando *interno* **set** permette di visualizzare l'environment della shell:

```
$
$ set
$
HOME=/home/tuser
HOSTNAME=sherazade
HOSTTYPE=i386
IFS=

LOGNAME=tuser
MAIL=/var/spool/mail/tuser
MAILCHECK=60
PATH=/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin:.
PS1=$
PS2=>
TERM=vt220
USER=tuser
...
$
```

Environment

- Alcune variabili vengono interpretate dalla shell:
  - **HOME**: la home directory
  - **PATH**: i pathnames di ricerca dei programmi
  - **TERM**: il tipo di terminale
  - **PS1**: il prompt di primo livello
  - **PS2**: il prompt di secondo livello
  - **LOGNAME**: lo username di login
  - **MAIL**: la mailbox

  e possono essere personalizzate dall'utente.

- L'environment di un processo viene determinato nel momento in cui un programma

Environment

viene caricato in memoria per mezzo di una primitiva di exec.

- Le variabili ambiente acquisiscono un significato (*semantica)* solo nel momento in cui un programma (ad esempio la shell) le interpreta.

- La shell decide quali variabili di environment debbano ricevere i processi figli, creati per mezzo dell'esecuzione dei *comandi.*

Variabili

```
$
$ PATH=$PATH:/usr/games
$ echo $PATH
/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin:.:/usr/gam
es
$ PATH=/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin:.
$ echo $PATH
/bin:/usr/bin:/usr/X11R6/bin:/usr/local/bin:.
$
```

```
$
$ HOME=/tmp
$
$ pwd
/u1/tuser
$ cd
$ pwd
/tmp
$
$ HOME=/home/tuser
$ cd
$ pwd
/home/tuser
$
```

Variabili

- Non tutte le variabili vengono interpretate dalla shell:

```
$ pwd
/home/tuser/bin
$ dir=‘pwd‘
$ cd /usr/local/bin
$
$...
$
$ cd $dir
$ pwd
/home/tuser/bin
$
$ echo $dir
/home/tuser/bin
$
```

- Non tutte le variabili vengono passate ai figli della shell. La shell usa il comando interno **export** per determinare quali coppie debbano essere *preservate* nell’environment:

Variabili

```
$ echo $dir
/home/tuser/bin
$
$ sh
$ echo $dir

$ exit
exit
$
$ export dir
$
$ sh
$ echo $dir
/home/tuser/bin
$ exit
exit
$
```

È ovviamente *conveniente* che le variabili di environment più importanti come PATH, TERM, MAIL, etc. vengano **esportate**.

Personalizzazione dell'Ambiente

- È possibile personalizzare la shell per adattarla alle proprie particolari esigenze.

  Molte caratteristiche possono essere modificate come il **prompt,** il **PATH** di ricerca dei comandi e così via.

- Il comando **stty** permette di modificare il significato di alcuni caratteri del terminale (interpretati dalla shell). Per esempio "**stty erase ^H**" assegna il carattere di cancellazione al **backspace**.

- Una volta definito l'ambiente desiderato è possibile memorizzare una opportuna sequenza di comandi in un file che la shell eseguirà appena prima del prompt.

Personalizzazione dell'Ambiente

- Il filename di questo particolare file (che forse può richiamare l'AUTOEXEC.BAT del MSDOS) dipende dal tipo di shell che state utilizzando:
  - **.profile** sh, bash, ksh
  - **.bash_profile** bash
  - **.login** csh, tcsh
- Notate che i files **dot** (i filenames che iniziano con un punto) non vengono visualizzati, per default, dal comando ls.

  L'opzione **-a** vi permetterà di visualizzarli:

```
$
$ ls -al .[A-z]*
-rw-r--r--   1 tuser    tuser        5763 Mar  8 18:32 .bash_history
-rw-rw-r--   1 tuser    tuser         106 Mar  8 12:11 .profile
$
```

Utenti e Permessi

- Ad ogni file è associato un insieme di permessi che definisce i diritti di accesso degli utenti del sistema.
- Al momento del login viene determinato lo **username** (o login-id) dell'utente: una stringa di caratteri.
- In effetti il sistema trasforma questa stringa in un numero: lo **uid** (userid). Differenti usernames possono essere mappati sullo stesso uid. È lo userid che identifica in modo univoco i diritti di accesso al file system.
- Una tabella, il file **/etc/passwd** contiene le informazioni che identificano gli utenti.

Utenti e Permessi

```
root:EHxmN.Myu3qdg:0:0:root:/root:/bin/bash
bin:*:1:1:bin:/bin:
daemon:*:2:2:daemon:/sbin:
adm:*:3:4:adm:/var/adm:
lp:*:4:7:lp:/var/spool/lpd:
sync:*:5:0:sync:/sbin:/bin/sync
shutdown:*:6:0:shutdown:/sbin:/sbin/shutdown
halt:*:7:0:halt:/sbin:/sbin/halt
mail:*:8:12:mail:/var/spool/mail:
news:*:9:13:news:/var/spool/news:
uucp:*:10:14:uucp:/var/spool/uucp:
operator:*:11:0:operator:/root:
games:*:12:100:games:/usr/games:
gopher:*:13:30:gopher:/usr/lib/gopher-data:
ftp:*:14:50:FTP User:/home/ftp:
nobody:*:99:99:Nobody:/:
sabatini:il8RvM.SUuLj2:252:1001::/home/sabatini:/bin/bash
peppe:YmhRBVKVTaJso:201:1001::/home/peppe:/bin/bash
tuser:2c40.lCbpEDXc:501:501::/home/tuser:/bin/bash
```

- Il file passwd in effetti implementa un piccolo database in cui record sono le righe, i campi sono separati dal carattere “**:**” e seguono il seguente schema:

account:password:UID:GID:GECOS:directory:shell

Utenti e Permessi

- Uno schema analogo viene utilizzato per definire i **gruppi** nel file ***/etc/group***:

```
root::0:root
bin::1:root,bin,daemon
daemon::2:root,bin,daemon
sys::3:root,bin,adm
adm::4:root,adm,daemon
tty::5:
disk::6:root
lp::7:daemon,lp
mem::8:
kmem::9:
wheel::10:root
mail::12:mail
news::13:news
man::15:
games::20:
nobody::99:
users::100:
thch:x:1001:
tuser:x:501:
```

secondo lo schema:

group_name:passwd:GID:user_list

Utenti e Permessi

- Il Kernel confronta lo **uid** di un processo con i permessi di accesso di un file per determinare i privilegi di accesso.

- In genere lo userid "**0**" definisce il *super-user* del sistema. Nella maggior parte dei sistemi UNIX corrisponde allo username **root**.

  Il super-user possiede speciali privilegi:

  - può leggere e modificare tutti i files del sistema prescindendo dallo schema dei permessi;
  - alcuni files presenti nel file system possono essere letti, modificati o eseguiti solo dal super-user;

  che permettono l'*amministrazione* della macchina UNIX.

Utenti e Permessi

```
$
$ ls -l /etc/passwd
-rw-r--r--   1 root     root          802 Mar  3 13:24 /etc/passwd
$ls -l /etc/group
-rw-r--r--   1 root     root          408 Mar  3 13:23 /etc/group
$
$ ls -l /dev/hda?
brw-rw----   1 root     disk       3,   1 May  5  1998 /dev/hda1
brw-rw----   1 root     disk       3,   2 May  5  1998 /dev/hda2
brw-rw----   1 root     disk       3,   3 May  5  1998 /dev/hda3
brw-rw----   1 root     disk       3,   4 May  5  1998 /dev/hda4
```

- Il comando **su** permette di guadagnare i diritti di un qualsiasi utente (quindi anche del super-user) fornendo la password di login (in alcuni il sistemi il suo uso è soggetto a restrizioni per motivi di sicurezza).

- Files come **/etc/passwd** o **/etc/group** presentano immediatamente il problema di fornire agli utenti *alcuni* dei privilegi del super-user in modo controllato.

Utenti e Permessi

- Occorre rispondere ad una semplice domanda: come può un utente, senza i diritti del super-user, modificare la propria password senza interagire direttamente con la persona che amministra il sistema?

```
-rw-r--r--  root    root   /etc/passwd
```

La risposta può essere trovata eseguendo un ls del programma **/usr/bin/passwd** (nella implementazione Linux RedHat 5.1):

```
-r-sr-xr-x  root     bin  /usr/bin/passwd
```

- Notate il permesso **s** al posto del più familiare permesso di esecuzione per l'utente **x**.

Utenti e Permessi

- Il programma /usr/bin/passwd è un programma **set-uid** dell'utente **root** e possiede il permesso di esecuzione **x** per qualsiasi utente.

  Questo significa che il file è eseguibile e che qualsiasi utente può farlo girare come se lo stesse eseguendo con i privilegi del *super-user.*

  Poiché il file /etc/passwd possiede i permessi di accesso in scrittura per l'utente **root** (*il super-user*) gli utenti potranno utilizzarlo per modificare la propria password:

```
$ passwd
Changing password for tuser
(current) UNIX password:
New UNIX password:
BAD PASSWORD: it is too short
New UNIX password:
Retype new UNIX password:
passwd: all authentication tokens updated successfully
$
```

Utenti e Permessi

- Il meccanismo *set-uid* “è un idea semplice ed allo stesso tempo elegante” (patented by Dennis Ritchie).

  C’è da aggiungere che è anche un meccanismo molto **pericoloso** per la **sicurezza** del sistema.

  I programmi **set-uid** “devono” essere corretti: un loro *malfunzionamento* può causare severi danni all’integrità di un sistema UNIX (**su** è un programma root set-uid ovviamente!).

  Un utente malizioso può approfittare di un “buco” in un programma set-uid per guadagnare i privilegi del *super-user* e in effetti da quel momento “*possiederà*” la macchina.

  Un sistema UNIX è *sicuro* quanto il **meno** sicuro dei suoi programmi root set-uid.

Utenti e Permessi

- Anche gli utenti non privilegiati possono “approfittare” di questo metodo per concedere i propri privilegi ad altri utenti del sistema:

```
$ id
uid=501(tuser) gid=501(tuser) groups=501(tuser)
$ ls -l myfile
-rw-------   1 tuser    tuser          87 Mar  9 13:20 myfile
$ cp /bin/cat mycat
$ chmod u+s mycat
$ ls -l mycat
-rwsr-xr-x   1 tuser    tuser        9032 Mar  9 13:21 mycat
$
```

```
[peppe@sherazade peppe]$ id
uid=201(peppe) gid=1001(thch) groups=1001(thch)
[peppe@sherazade peppe]$
[peppe@sherazade peppe]$ cat /home/tuser/myfile
cat: /home/tuser/myfile: Permission denied
[peppe@sherazade peppe]$
[peppe@sherazade peppe]$ /home/tuser/mycat /home/tuser/myfile
Questo’ e’ un file posseduto da tuser senza
privilegi di lettura per gli altri utenti.
[peppe@sherazade peppe]$
```

Morale: il privilegio **set-uid** va usato con parsimonia e prudenza.

Utenti e Permessi

- Il permesso **set-uid** (esiste un analogo per il gruppo noto come **set-gid**) va utilizzato solo nei casi in cui sia strettamente **necessario**.

- I programmi **set-uid**, soprattutto quelli posseduti dal super-user, vanno implementati con estrema attenzione ed una cura particolare va dedicata alla loro **implementazione** ed al loro **debugging**.

- Nella maggior parte dei sistemi UNIX il privilegio set-uid non dovrebbe essere **mai** concesso ad una shell script. Esiste un “semplice” meccanismo, noto fin dagli albori dei sistema UNIX, che permette di utilizzarle *immediatamente* per guadagnare i diritti dell’utente che possiede la script.

L’editor vi

- Una delle funzioni più importanti in ogni sistema operativo è quella di **editing**. Imparare a creare e modificare files è forse uno dei passi più importanti dell’interazione con un nuovo ambiente di calcolo.

- L’ambiente UNIX mette a disposizione diversi **editors**:

  - **ed**: editor di linea
  - **ex**: ancora un editor di linea
  - **vi**: the **vi**sual editor

    forse l’editor full screen più diffuso ed usato in ambiente UNIX
  - **emacs**: un formidabile *ambiente* di editing

    è uno dei più ***imponenti*** editor full screen.

L’editor vi

- Due definizioni forse già note:
  - Gli **editor di linea** permettono di lavorare su una linea alla volta di un file. Sono nati per funzionare con terminali di tipo telescrivente.
  - Gli **editor full screen** permettono di aprire una **finestra** sul file di lavoro, sono orientati a terminali video con schermo a cursore indirizzabile e permettono di eseguire modifiche spostando il cursore sulla parte interessata della finestra.
- **ed** è nato insieme a UNIX ed è presente in tutte le piattaforme UNIX (alla stregua del comando ls).
- Sia **ex** che **vi** sono stati scritti da William Joy, presso la University of California - Berkeley e sono due aspetti dello stesso editor.

L’editor vi

vi fornisce una interfaccia visuale ad ex, ma i comandi di ex sono tutti disponibili e si comportano nello stesso modo.

ex è basati sul più semplice ed, ma è dotato di estensioni e di caratteristiche addizionali.

- Per poter usare un editor full screen è ovviamente necessario poter disporre di un terminale video a cursore indirizzabile ed occorre che il sistema lo gestisca correttamente.

  Affinché ciò sia possibile la variabile di environment **TERM** deve essere definita correttamente e il sistema deve possedere la definizione delle sequenze di caratteri necessarie ad indirizzare il cursore.

L’editor vi

- Esistono fondamentalmente due linee per la gestione dei terminali in ambiente UNIX: **termcap** e **terminfo**. Sono concettualmente analoghe. La stringa definita dalla variabile TERM viene confrontata con una serie di definizioni contenute in un database ed una libreria di codice fornisce la necessaria interfaccia.

- Lavoreremo con una variazione dell’editor vi: **vim** (Vi IMproved) in ambiente Linux RedHat 5.1. L’interfaccia terminale è fornita da una libreria termcap (date una occhiata al file **/etc/termcap**). Il terminale sarà fornito da un programma di telnet (in ambiente Windows), in emulazione del terminale DEC vt220.

Invocare vi

- Per lanciare **vi** è sufficiente immettere il comando:

```
$ vi  [ filename ]
```

  Il nome del file può essere omesso. In questo caso **vi** opera su un file il cui filename verrà specificato al momento opportuno.

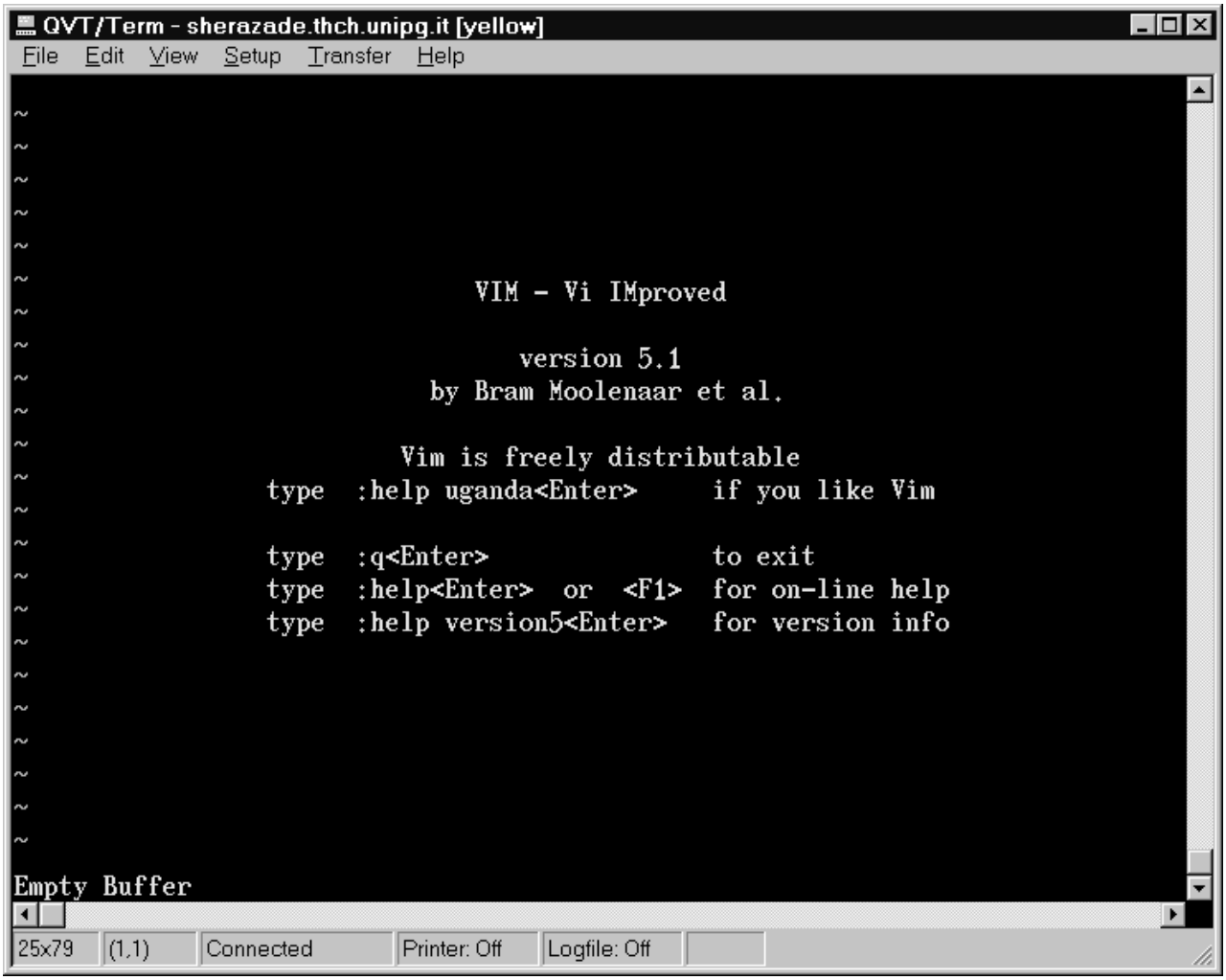
- Se il comportamento della tastiera non è quello atteso oppure si ottiene un messaggio di questo tipo:

```
Visual mode requires more cursor capability
than the terminal provides.
```

  la variabile **TERM** non è definita correttamente oppure il terminale non supporta vi.

Invocare vi

- Lanciando vi (**vim**) dovreste ottenere una schermata di questo tipo:



Il comando ctr-L (*screen redraw*) permetterà di ripulire lo schermo e passare alla configurazione *operativa* di vi.

Invocare vi

- Lanciamo vi in editing sul file “nuovo”:

```
$ vi nuovo
```

Lo schermo verrà cancellato e sostituito da:

```
■
~
~
~
~
~
~
...
~
~
~
~
"nuovo" [New File]
```

- I caratteri “~” (tilde) nella prima colonna a sinistra indicano che non esiste testo nel buffer interno di vi, neppure come linee vuote.

Invocare vi

- La linea dei messaggi nell'ultima riga dello schermo indica il nome e lo stato del file. In questo caso vi segnala che stiamo *creando* un nuovo file, appunto.

- Il comando **ZZ** ci permette di salvare il contenuto del buffer interno (valido solo per la sessione corrente) nel file **nuovo**.

- Tentiamo ora di editare un file che esisteva già:

```
$ vi test
```

Invocare vi

```
Questo e’ il file test ...
~
~
~
~
~
...
~
~
"test" 1 line, 27 characters
```

Il file viene visualizzato ed ancora una volta le linee vuote segnalate. Il messaggio di stato indica che il file esisteva già e fornisce la sua lunghezza in linee e caratteri.

- Il comando “**:**” (due punti) pone vi nello stato di accettare comandi **ex**. Sulla linea di stato (ora linea comandi) sarà possibile utilizzare i comandi dell’editor di linea **ex**.

Invocare vi

Se vogliamo abbandonare il file senza salvare le modifiche possiamo provare con:

```
:q! (RETURN)
```

Per rileggere il file nel buffer interno (senza salvare le modifiche):

```
:e! (RETURN)
```

- Per salvare il buffer interno potete usare il comando ex **w**:

```
:w [filename]
```

Invocare vi

Il comando w (write) può essere utile se non si riesce a salvare il file. Se si ricevono messaggi del tipo:

```
File exists (use ! to override)

'readonly' option is set (use ! to override)
```

è possibile usare il comando **w filename** oppure **w!** filename per *forzare* vi al salvataggio.

Funzioni di Editing

- L’editor vi ha due **modi** (o stati) di funzionamento:
  - Modo **comandi**
  - Modo **inserimento**
- In modo **comandi** (lo stato iniziale dell’editor) vi interpreta i caratteri ricevuti dal terminale come richieste per l’esecuzione di operazioni sul buffer interno.
- In modo **inserimento** vi si comporta come una macchina da scrivere (o una telescrivente): i caratteri ricevuti dal terminale vengono memorizzati nel buffer interno e visualizzati.

Funzioni di Editing

- Un certo numero di comandi fanno passare vi al modo inserimento: il più comune è il comando di insert **i**. Per tornarne al modo comandi è sufficiente usare il tasto **ESC**.

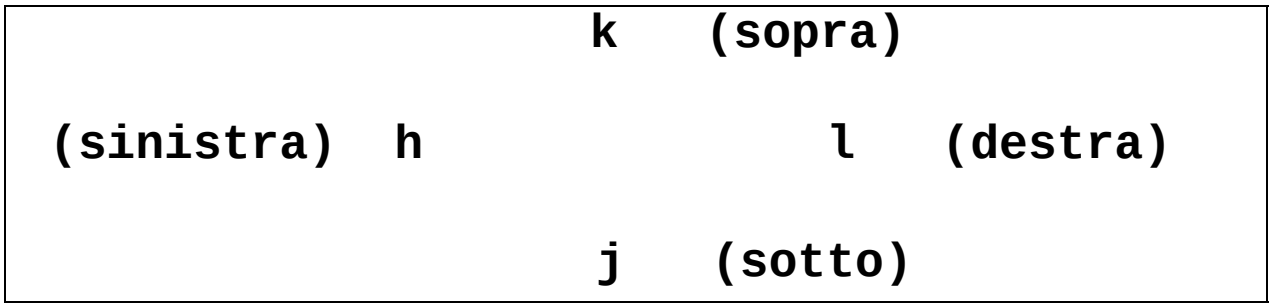
- Il comando ex **set showmode** permette di visualizzare sulla linea di stato il modo in cui sta funzionando vi. Apparirà:

```
-- INSERT --
```

  o una sua variazione sul tema.

- In modo comandi è possibile posizionare il cursore in un qualunque punto del file. I comandi (sempre disponibili) sono:

Funzioni di Editing



Sui terminali che possiedono le *arrow keys* (tasti freccia), le *freccette* svolgono la stessa funzione.

I tasti **RETURN** e **BACKSPACE** svolgono (in genere) la stessa funzione che hanno sulla linea di comando della shell.

- Non è possibile posizionare il cursore oltre una tilde “**~**”. Questo carattere indica una linea della finestra di editing priva di una corrispondente linea di testo nel buffer interno.

Funzioni di Editing

- Un argomento numerico, premesso ad un comando, ne ripete l'esecuzione per un corrispondente numero di volte.

  Il comando **4j** sposta il cursore verso il basso di 4 linee.

- Altri due utili comandi:

0 (zero)	**va ad inizio linea**
$ (dollaro)	**va a fine linea**

- Il comando **i** (insert) passa in modo inserimento ed inserisce il testo dalla posizione corrente del cursore.

- Il comando **a** (append) passa in modo inserimento ed inserisce il testo dalla

Funzioni di Editing

posizione che segue il cursore. Equivale al comando freccia a destra ed i.

Espressioni Regolari

- Le **espressioni regolari** sono nate con l'editor ed. Un buon numero di comandi UNIX (grep, awk, sed, ed, vi, ...) usa le espressioni regolari ogni qualvolta si presenta la necessità di eseguire la ricerca di un *pattern* (pattern matching).

- Sono state concepite per risolvere problemi legati allo sviluppo di linguaggi di programmazione: analizzatori lessicali, interpreti, compilatori, etc.

- Sono uno strumento fondamentale nello sviluppo di utilità o di nuovi comandi nell'ambiente UNIX, spesso uno strumento quotidiano di lavoro.

Espressioni Regolari

```
ls -l | grep '^d'

ls -l | grep '.......rw'
```

Il primo esempio elenca le subdirectories della directory corrente. Il secondo i files che tutti possono leggere e scrivere.

Un modo semplice per cercare gli utenti senza una password:

```
grep '^[^:]*::' /etc/passwd
```

il pattern in questo caso rappresenta qualsiasi stringa che inizia (^) con una stringa che non contiene i due punti ([^:]) ed è seguito da "::".

Esattamente quello che ci aspetteremmo da un record in /etc/passwd che non contiene la password cifrata.

Espressioni Regolari

Carattere	Match
c	qualsiasi carattere che non sia un *metacarattere* fa match con se stesso
\c	il carattere *alla lettera*
^	inizio linea
$	fine linea
.	un singolo carattere
[...]	insieme di caratteri: [a-z], [12-4]
[^...]	non appartiene all'insieme di caratteri
r*	zero o più occorrenze di r
r+	una o più occorrenze di r
r?	zero oppure una occorrenza di r
r1r2	r1 seguita da r2
r1|r2	r1 oppure r2 (egrep)

Nessuna espressione regolare fa mai match con il *newline***:** il pattern "**Capitolo.**" non farà match con le righe in cui la stringa **Capitolo** compare alla fine della linea.

grep

- Il comando **grep** implementa la ricerca di espressioni regolari. Alcuni utili *opzioni* del comando grep:
  - **-v**

    inverte il significato della ricerca; vengono visualizzate tutte e sole le righe che *non* fanno match;
  - **-i** (in alcuni casi **-y**)

    nessuna distinzione tra lettere minuscole e lettere maiuscole (case *insensitive*)
  - **-n**

    visualizza i numeri di linea
  - **-l**

    visualizza solamente i *filenames* dei files che contengono almeno una stringa che soddisfa il match.

grep

- Una espressione regolare fa sempre match con la stringa più *lunga possibile*:

```
grep 'A.*Z'

All of us, including Zippy, our dog
^^^^^^^^^^^^^^^^^^^^^

All of us, including Zippy and Ziggy
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^

All of us, including Zippy and Ziggy and Zelda
^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

- È importante soprattutto per i programmi che eseguono sostituzioni basate su una espressione regolare come ad esempio il *filtro* **sed**.

grep

gres: un comando che può essere utile per

```
#!/bin/sh

if [ $# -lt "3" ]; then
  echo 'Usage: gres pattern replacement file'
  exit
fi

pattern=$1
replacement=$2

if [ -f $3 ]; then
  file=$3
else
  echo $3 is not a file
  exit
fi

sed -e 's/'$pattern'/'$replacement'/' $file
```

sperimentare con le espressioni regolari.

Applicato al file dell'esempio precedente:

```
$ gres "A.*Z" "00" sample
00ippy, our dog
00iggy
00elda
$
```

grep

- Per ottenere un elenco dei messaggi di posta in giacenza nella vostra mailbox:

```
$ grep "^From:" $MAIL
From: Giuseppe Vitillaro <gvt@unipg.it>
From: tuser@sherazade.thch.unipg.it
From: Giuseppe Vitillaro <peppe@unipg.it>
$
```

- O ancora per cercare un nome nella vostra *rubrica telefonica*:

```
#!/bin/sh

phone=$HOME/memo/phone.list

grep -i $* $phone
```

grep

- Una ricerca della primitiva *open*() nei file di include del sistema operativo:

```
$ grep 'open.*(.*oflag.*)' /usr/include/*.h | more

/usr/include/fcntl.h:extern int __open __P ((__const char
*__file, int __oflag, ..
.));
/usr/include/fcntl.h:extern int open __P ((__const char *__file,
int __oflag, ...)
);
$
```

- Un americano (beato lui) potrebbe cercare un numero di telefono con le espressioni regolari:

```
esempi: 287-0522  784-7094

[0-9][0-9][0-9]-[0-9][0-9][0-9][0-9]

[0-9]\{3\}-[0-9]\{4\}
```

awk

- Il comando awk trae il suo nome dalle iniziali dei suoi autori Al **A**ho, Peter **W**einberger e Brian **K**ernighan.

- È un interprete che implementa un *semplice* linguaggio di programmazione che lo rende particolarmente adatto a fungere da ***filtro***.

- Il suo uso è molto simile all'uso del comando grep:

```
awk 'program' filenames

  program
     pattern { action }
     pattern { action }
```

dove pattern può essere una espressione regolare.

awk

- awk legge le linee in input dai *filenames* ed esegue la *action* specificata per ogni *pattern* che soddisfa il match.

- Gli argomenti *pattern* ed *action* sono entrambi opzionali. Se pattern è assente *tutte* le linee soddisfano il match. Quando *action* è assente la azione di default è il comando ***print***: la linea viene inviata allo Standard Output.

  Ecco due versioni *complicate* del comando cat:

```
awk '//' /etc/passwd

awk '{ print }' /etc/passwd
```

  e due del comando grep:

```
awk '/espressione-regolare/'

awk '/espressione-regolare/' '{ print }'
```

awk

- Ovviamente le funzionalità sono più ampie: awk possiede il concetto di **campo**.

  awk assume (per default) che la linea in input abbia una struttura di **record** dove i campi sono separati da *blanks* o *tabs*:

```
$ who
sabatini tty1     Mar  2 11:02
sabatini ttyp1    Mar 10 08:58
tuser    ttyp0    Mar 12 18:08

$ who | awk '{print $1, $5}'
sabatini 11:02
sabatini 08:58
tuser 18:08
```

  I campi sono numerati in ordine progressivo: **$1**, **$2**, ..., **$NF**, dove la variabile NF (*number of fields*) indica il numero di campi disponibili.

  Il separatore di campo può essere facilmente modificato con il flag **-F**.

awk

```
$ cat /etc/passwd
root:*:0:0:root:/root:/bin/bash
bin:*:1:1:bin:/bin:
daemon:*:2:2:daemon:/sbin:
...

$ cat /etc/passwd | awk -F: '{print $1, $6}'
root /root
bin /bin
daemon /sbin
...
```

In questo caso il file /etc/passwd è stato elaborato da awk per produrre in output l'elenco degli utenti e delle loro home directories.

- awk può essere usato con successo tutte le volte che è necessario elaborare files strutturati come /etc/passwd o /etc/group (immaginatene l'efficacia su file di password contenenti *migliaia* di utenti).

awk

- Il campo **$0** rappresenta l'intera linea di testo in input. Ecco un modo semplice per numerare le linee di un testo:

```
$ awk '{ print NR, $0 }' storia.unix
1 Il gruppo MULTICS si trova temporanemente
2 senza un sistema di calcolo interattivo.
3
4 Ken Thompson e Dennis Ritchie disegnano
5 un nuovo File System che si evolvera'
6 in una delle prime versioni del File System UNIX.
```

- **awk** permette di elaborare l'output di un qualunque programma allo scopo di estrarre tutte e sole le informazioni necessarie ad eseguire un determinato compito.

- È particolarmente utile anche in casi in cui sia necessario eseguire elaborazioni *numeriche.*

awk

```
$ df
Filesystem         1024-blocks  Used Available Capacity Mounted on
/dev/hda5              39975   21376    16535     56%   /
/dev/hda8              32171   15675    14835     51%   /var
/dev/hda6             396500  355273    20741     94%   /usr
/dev/hda9              15856    1636    13390     11%   /tmp
/dev/hda1              16582    9654     6928     58%   /dos
/dev/hda7             301340  268555    17213     94%   /u1
/dev/sda1             204219  147577    46096     76%   /u2
$

$ df | awk '{ space=space+$3 } END { print space }'
819746
$
$ df | awk '{ space=space+$2 } END { print space }'
1007667
$
```

```
$
$ df | awk '{
>   size += $2;
>   used += $3;
> }
> END {
>   printf ( "size=%d used=%d %d\%\n",
>             size, used, (used/size)*100 );
> }'
size=1007667 used=819747 81%
$
```

awk al lavoro per produrre un rapporto sulla percentuale di spazio utilizzato nei filesystems.

awk

- Occorre, come peraltro con il comando grep, porre molta attenzione nell'interazione con i metacaratteri interpretati dalla shell.

  Il programma awk (a meno che non sia contenuto in un file ed eseguito con il flag **-f**) va *opportunamente* protetto con i singoli apici a meno che non si desideri esplicitamente che la shell ne interpreti una parte):

```
#!/bin/sh

awk '{ print $'$1' }'
```

```
$ who | field 1
sabatini
sabatini
sabatini
sabatini
tuser
$
```

Cercare un file

- In un *piccolo* sistema UNIX possono essere presenti **migliaia** di files:

```
$ df -i
Filesystem            Inodes   IUsed   IFree  %IUsed Mounted on
/dev/hda5              10368    2550    7818     25%  /
/dev/hda8               8320     378    7942      5%  /var
/dev/hda6             102800   25909   76891     25%  /usr
/dev/hda9               4160      51    4109      1%  /tmp
/dev/hda1                  0       0       0      0%  /dos
/dev/hda7              78128    9982   68146     13%  /u1
/dev/sda1              52832    1587   51245      3%  /u2
```

  La home directory di un utente attivo da qualche anno può raggiungere con facilità un numero di files tra 10.000 e 100.000.

  Trovare il particolare file a cui siamo interessati può diventare un problema *complicato*.

- Il comando **find** possiede le funzionalità necessarie ad eseguire questo importante compito.

Cercare un file

- La pagina del manuale di find ottenuta con il comando **man find**:

```
FIND(1L)                                                    FIND(1L)

NAME
       find - search for files in a directory hierarchy

SYNOPSIS
       find [path...] [expression]

DESCRIPTION
       This manual page documents the GNU version of find.  find searches
       the directory tree rooted at each given file  name  by  evaluating
       the given expression from left to right, according to the rules of
       precedence (see section OPERATORS), until  the  outcome  is  known
       (the  left hand side is false for and operations, true for or), at
       which point find moves on to the next file name.
```

- **find** permette di cercare un *filename* all'interno del sottoalbero del file system UNIX specificato da **path**.

  L'espressione permette di specificare il *criterio* di ricerca del file. È possibile richiedere

Cercare un file

l'esecuzione di un comando sui filenames selezionati.

Il criterio può essere molto semplice: la ricerca di un *particolare* file:

```
$ find . -name "doc1.1"
./doc/doc1.1
./doc.copy/doc1.1
$
```

oppure possiamo specificare un *pattern,* utilizzando gli stessi metacaratteri utilizzati dalla shell:

```
$ find . -name 'doc1.[12]' -print
./doc/doc1.1
./doc/doc1.2
./doc.copy/doc1.1
./doc.copy/doc1.2
$
```

È necessario *proteggere* i metacaratteri per impedire che vengano interpretati dalla shell.

Cercare un file

- Il comando find può essere molto **oneroso** per il sistema se utilizzato su gerarchie che contengono un elevato numero di files o su gerarchie che vengono *montate* attraverso meccanismi di networking (NFS, Samba).

- Può essere l'unico modo per eseguire alcuni compiti fondamentali nell'amministrazione di un sistema UNIX:

```
$ find /bin -perm -u+s -ls
  7105   13 -rwsr-xr-x   1 root     root        12648 May  9  1998 /bin/su
  7118   36 -rwsr-xr-x   1 root     root        35876 May  1  1998 /bin/mount
  7119   18 -rwsr-xr-x   1 root     root        17884 May  1  1998 /bin/umount
  7127   15 -rwsr-xr-x   1 root     root        14340 May  6  1998 /bin/ping
  7144   16 -rws--x--x   1 root     root        15588 May  4  1998 /bin/login
```

Elencare periodicamente i programmi *set-uid* presenti in una gerarchia può essere **vitale** per la sicurezza del sistema.

Cercare un file

```
$ pwd
/home/tuser
$ ls -l ‘find . ! -user tuser‘
-rw-r--r--   1 root     root           0 Mar 13 11:29 ./source/root_file
$
```

Analogamente elencare i files presenti nelle home directories che non appartengono all’utente proprietario della directory può fornire indicazioni su attività *maliziose* da parte degli utenti stessi.

- Il comando find può cooperare con la shell, con grep o awk, con il comando xargs allo scopo di eseguire operazioni di manutenzione sui file systems.

```
$ find . -type f |
> xargs ls -l |
> awk ’{ size += $5 } END { print size }’
473475
$ du -ks .
493     .
```

Cercare un file

Anche in questo caso il comando **echo** può rivelarsi molto utile per comprendere i meccanismi di funzionamento del comando:

```
$ find . -name "cap*" -exec echo {} ;
find: missing argument to ‘-exec’
$
$ find . -name "cap*" -exec echo {} \;
./doc/cap2
$
$ find . -name "cap*" -exec echo mv {} /tmp \;
mv ./doc/cap2 /tmp
$
$ find . -name "cap*" -exec mv {} /tmp \;
$ ls /tmp/cap*
/tmp/cap2
$
```

ed ancora una volta occorre fare attenzione a bloccare l’interpretazione dei metacaratteri da parte della shell. Questo è un problema condiviso da molti comandi: alcuni metacaratteri sono interpretati sia dalla shell che dal comando stesso.

Cercare un file

- Il comando find può operare su:
  - filenames
  - inodes numbers
  - date di aggiornamento, modifica ed accesso
  - numero di links
  - permessi di accesso
  - dimensioni
  - proprietario, gruppo, uid e gid
  - tipo del file: file, directory, socket, etc.
- In pratica find può eseguire selezioni su tutte le informazioni ritornate dalla primitive *stat()* o *fstat()* nella struttura dati **stat** definita nel file di include **/usr/include/sys/stat.h**.

Shell Programming

- La shell non si limita ad interpretare i comandi interattivi: è un vero e proprio **interprete** che implementa un linguaggio di programmazione.

- Tutte le volte che una *sequenza* di comandi si rivela particolarmente utile (soprattutto se è complicata) può essere trasformata in un *programma*, in una **script**.

- L'utente UNIX (per lo meno il tipico utente che avevano in mente i progettisti originali) usa il linguaggio di scripting per *assemblare* vari comandi già esistenti con lo scopo di rendere **automatico** un compito ripetitivo o una sequenza di comandi che deve essere eseguita fuori dall'ambiente interattivo.

Shell Programming

- Pur implementando, in genere, un linguaggio relativamente semplice, l'interazione tra shell ed ambiente di sistema permette di risolvere un gran numero di problemi.

- Vediamo come nasce una script. il compito da risolvere è di semplice definizione:

  "Dato un comando, scandire la variabile di environment **PATH** per determinare (se esiste) il pathname assoluto del programma che verrà eseguito".

- Il primo problema da risolvere è di trasformare la lista di pathnames contenuta nella variabile PATH in un *elenco* di *argomenti*. Occorre tener presente che notazioni come:

Shell Programming

- :elenco
- elenco::elenco
- elenco::

indicano il pathname che corrisponde alla directory corrente, ovvero il pathname “**.**”.

Qualche semplice esperimento interattivo con il comando **echo** e con il filtro **sed** ci permetterà di farci un’idea di come risolvere il problema.

```
$ echo $PATH
:/bin:/home/tuser/bin:/usr/bin
$
$ echo $PATH | sed ’s/:/ /g’
 /bin /home/tuser/bin /usr/bin
$
```

Chiaramente questo primo esperimento non ha risolto il problema del *current path*.

Shell Programming

Per risolverlo dobbiamo essere in grado di trasformare:

- :elenco .:elenco
- elenco::elenco elenco:.:elenco
- elenco: elenco:.

```
$ echo ":/bin" | sed 's/^:/.:/'
.:/bin

$ echo "/bin::/usr/bin" | sed 's/::/:.:/g'
/bin:.:/usr/bin

$ echo "/bin:/usr/bin:" | sed 's/:$/:./'
/bin:/usr/bin:.
```

ed anche per questi tre casi il comando sed ci permette di ottenere una soluzione.

Possiamo ora assemblare le tre sostituzioni in unico comando sed:

Shell Programming

```
$ echo $PATH | sed 's/^:/.:/
>                  s/::/:.:/g
>                  s/:$/:./
>                  s/:/ /g'
. /bin /home/tuser/bin /usr/bin
```

Ora siamo in grado di produrre un elenco di *argomenti* che corrisponde a tutti e soli i pathnames presenti nella variabile PATH.

- Il comando **test** ci permette di determinare le caratteristiche di un file:

```
$ test -x /bin/ls
$ echo $?
0
$ test -x /bin
$ echo $?
0
$ test -f /bin/ls
$ echo $?
0
$ test -f /bin
$ echo $?
1
```

Shell Programming

e quindi il comando:

```
$ test -f /bin/ls -a -x /bin/ls
$ echo $?
0
$ test -f /bin -a -x /bin
$ echo $?
1
```

ci permette di determinare se un file è in effetti un *eseguibile* del nostro file system.

- Per poter portare a termine il nostro compito è sufficiente ora trovare un modo per eseguire il comando test per ciascuno dei pathnames dell'elenco che abbiamo generato.

  Un altro esperimento con il costrutto ***for*** del linguaggio di scripting ci fornirà la parte mancante:

Shell Programming

```
$ for p in a1 a2 a3 a4
> do
>   echo $p
> done
a1
a2
a3
a4
$
```

Usando ora il meccanismo di backquote possiamo trasformare la variabile **PATH** in una lista di argomenti su cui il comando for può lavorare:

```
$ for p in ‘echo $PATH | sed ’s/^:/.:/
>                              s/::/:.:/g
>                              s/:$/:./
>                              s/:/ /g’‘
> do
>   echo $p
> done
.
/bin
/home/tuser/bin
/usr/bin
$
```

Shell Programming

- Come abbiamo visto il comando **test** restituisce **0** (**vero**) o **1** (**falso**) in base al fatto che il file in esame abbia soddisfatto il criterio in esame.

  Il costrutto **if** ci permette di utilizzare questo risultato per decidere se eseguire una sequenza di comandi:

```
$ if test -f /bin/ls -a -x /bin/ls
> then
>   echo "ok"
> fi
ok
$

$ if test -f /bin -a -x /bin
> then
>   echo "ok"
> fi
$
```

  Si deriva facilmente il prossimo passo nella costruzione della nostra script:

Shell Programming

```
$ for p in ‘echo $PATH | sed ’s/^:/.:/
>                             s/::/:.:/g
>                             s/:$/:./
>                             s/:/ /g’‘
> do
>   if test -x $p/ls -a -f $p/ls
>   then
>     echo $p/ls
>    fi
> done
/bin/ls
$
```

Abbiamo praticamente risolto il nostro problema. I comandi (già implementati) **type** o **which** possono aiutarci a verificare se il nostro lavoro è corretto.

- Per ottenere una **script** mancano solo pochi passi:
  - memorizzare la sequenza in un file eseguibile
  - passare alla script, come argomento, il comando

Shell Programming

```
$ cat mywhich
#!/bin/sh

for p in ‘echo $PATH | sed ’s/^:/.:/
                           s/::/:.:/g
                           s/:$/:./
                           s/:/ /g’‘
do
  if test -x $p/$1 -a -f $p/$1
# equivalente a
# if [ -x $p/$1 -a -f $p$1 ]
  then
    echo $p/$1
    exit 0
  fi
done

exit 1
```

e per renderla eseguibile:

```
chmod +x mywhich
```

- Ora abbiamo disponibile (nel nostro PATH) il comando **mywhich**, che può essere eseguito con la sintassi: “**mywhich** command”.

Shell Programming

```
$ mywhich ls
/bin/ls

$ mywhich cat
/bin/cat

$ mywhich mywhich
./mywhich

$ mywhich who
/usr/bin/who

$ mywhich grep
/bin/grep

$ mywhich awk
/bin/awk

$ mywhich sh
/bin/sh

$ mywhich sed
/bin/sed

$ mywhich wich
$ echo $?
1

$ mywhich which
/usr/bin/which
```