Laboratorio di Sistemi Operativi

Alessandro Valenti

21 febbraio 2005

Indice

1 Introduzione al corso **3**
- 1.1 Svolgimento delle lezioni . . . 3
- 1.2 Obiettivi del corso . . . 3
- 1.3 Verifica dell'apprendimento . . . 4
- 1.4 Consigli agli Studenti . . . 4

2 Concetti di base **5**
- 2.1 Modularità e Processi . . . 6
- 2.2 Gli utenti del sistema . . . 6
- 2.3 Accesso ai comandi . . . 6
- 2.4 Directory corrente . . . 8

3 Il filesystem **11**
- 3.1 File e Directory . . . 12
- 3.2 Permessi associati ai file . . . 13
- 3.3 Bit speciali . . . 14
- 3.4 Altre informazioni sui file . . . 15

4 Alcuni comandi **17**
- 4.1 Man Pages . . . 18
- 4.2 touch . . . 18
- 4.3 mkdir . . . 19
- 4.4 chown . . . 19
- 4.5 chmod . . . 19
- 4.6 cat . . . 20

5 Usiamo la shell **21**
- 5.1 La ridirezione . . . 21
  - 5.1.1 Standard Error . . . 22
  - 5.1.2 Ridirezioni fra descrittori . . . 23
- 5.2 Pipeline . . . 23
- 5.3 Gli shell script . . . 24
  - 5.3.1 Le variabili di shell . . . 25
  - 5.3.2 Esportazione di variabili . . . 26
  - 5.3.3 Gli argomenti della linea di comando . . . 26
- 5.4 Elaborazione condizionale . . . 27
  - 5.4.1 Il comando test . . . 28
- 5.5 Wildcard e pathname expansion . . . 29
  - 5.5.1 Il ciclo for . . . 30
  - 5.5.2 L'operazione di quoting . . . 31

5.6 Command substitution 31

Capitolo 1

Introduzione al corso

Il corso di *Laboratorio di Sistemi Operativi* è un insegnamento facoltativo previsto nei Corsi di Laurea triennali di Ingegnaria Informatica, Ingegneria Elettronica ed Ingegneria della Telecomunicazioni.

Il superamento del corso consente allo Studente di acquisire 3 crediti.

La finalità del corso é quella di presentare una serie di esercizi ed esempi di programmazione che consentano allo Studente di comprendere la struttura e l'uso di un sistema operativo. Nello specifico, ma in modo non limitativo o pregiudiziale, si farà particolare riferimento a sistemi operativi conformi allo standard IEEE POSIX.

1.1 Svolgimento delle lezioni

Le lezioni, ad eccezione della prima settimana, si terranno presso il Laboratorio Base ed avranno un carattere fondamentalmente pratico. Rispetto agli scorsi anni è stata eliminata l'ora teorica in aula a vantaggio di un migliore sfruttamento del Laboratorio che sarà suddiviso in tre turni di due ore ciascuno.

Ad ogni Studente che intenda frequentare il corso di **Laboratorio di Sistemi Operativi** è richiesta la compilazione di un questionario su web in modo da poter ottenere, nei limiti del possibile, l'assegnazione del turno preferito. Indipendentemente dal turno, detta iscrizione è necessaria per permettere la creazione dell'account (coppia username e password) di accesso ai pc del laboratorio.

I 30 pc del Lab. Base sono provvisti di un dual-boot che consente agli Studenti di avviare il sistema operativo open source Linux; lo stesso account potrà essere utilizzato dagli Studenti per esercitarsi anche sulle workstation Sun del LICA con sistema operativo Solaris.

Lo svolgimento dell'attività didattica è sincronizzato ed abbinato al corso di **Sistemi Operativi** in modo che gli studenti possano applicare in laboratorio concetti già visti e studiati teoricamente.

1.2 Obiettivi del corso

Gli Studenti che decideranno di frequentare il corso di *Laboratorio di Sistemi Operativi* avranno la possibilità di sperimentare in pratica i concetti appresi dal corso di *Sistemi Operativi* acquisendo la capacità di programmare un sistema Unix tramite il linguaggio di scripting della shell ed il linguaggio c.

Si farà, in particolare, riferimento all'evoluzione **bash** della storica Bourne shell ed alla versione *improved* del vi ribattezzata **vim**.

Gli obiettivi prefissati per il corso di *Laboratorio di Sistemi Operativi*, che lo studente raggiungerà seguendo le lezioni in laboratorio ed esercitandosi sui temi proposti, sono i seguenti:

- capacità d'uso di macchine unix, della bash e dell'editor vim
- conoscenza della shell di unix e delle modalità di realizzazione di shell script
- padronanza dei sistemi di programmazione concorrente orientati ai processi e conoscenza dei principali metodi di sincronzzazione e comunicazione
- capacità di realizzare programmi in c utilizzando la libreria standard
- padronanza di unix come host di sviluppo, utilizzo del tool make

1.3 Verifica dell'apprendimento

La valutazione degli Studenti viene fatta dalla commissione esaminando la soluzione ad un compito del corso di *Sistemi Operativi* proposto in laboratorio. Ogni Studente potrà pertanto ottenere la valutazione del corso di *Laboratorio di Sistemi Operativi* contestualmente a quella del corso di *Sistemi Operativi*.

Ovviamente, per effetto dei diversi criteri di valutazione, il voto di *Laboratorio di Sistemi Operativi* potrà essere sensibilmente differente da quello di *Sistemi Operativi* sul medesimo svolgimento.

La soluzione dei compiti proposti consiste in alcuni file di testo che l'Esaminando dovrà produrre con l'editor vi/vim riutilizzando a suo piacimento parti di esercitazioni o esempi che potrà consultare durante lo svolgimento.

Ogni compito si compone di due parti:

- la parte shell
- la parte c

La parte shell richiede la realizzazione di uno script interpretabile dalla Bourne shell o dalla bash. La correzione di *Laboratorio di Sistemi Operativi* consente e raccomanda l'uso dei neologismi introdotti dalla bash mentre la correzione di *Sistemi Operativi* considera l'incompatibilità con la Bourne shell come un errore.

La parte c richiede la realizzazione di uno o più file sorgenti in linguaggio c corredati dal relativo makefile che, in seguito al processo di compilazione e linking, producano un file eseguibile rispondente alle specifiche del testo.

L'assenza di almeno un file richiesto o la presenza di errori sintattici negli script o nei sorgenti c invalidano la correzione degli elaborati.

1.4 Consigli agli Studenti

Le ore di lezione saranno suddivise in due parti, quella iniziale di 4 settimane relativa alla programmazione in shell e quella conclusiva di 5 settimane relativa alla programmazione c.

La frequenza della seconda parte di lezioni richiede la conoscenza di base del linguaggio c che gli Studenti dovrebbero aver acquisito da precedenti insegnamenti propedeutici. Si consiglia pertanto di verificare la propria conoscenza del linguaggio c e, eventualmente, affiancare allo studio delle prime 4 settimane di shell un ripasso dei concetti fondamentali di programmazione.

Gli Studenti che volessero approfondire a casa lo studio e gli esercizi proposti, possono installare una qualsiasi distribuzione di Linux, anche non recentissima. Considerate che un *vecchio* pc classe 486 con 16M di ram è sufficiente per tutti gli esercizi proposti.

Capitolo 2

Concetti di base

Per quanto si rimandi al corso di *Sistemi Operativi* per ogni approfondimento sulla struttura di un sistema operativo, ritengo opportuno inserire una breve introduzione sull'argomento che potrà essere di aiuto agli Studenti nella comprensione dei lavori successivi.

Un sistema operativo è, a tutti gli effetti, una porzione di software alla quale si delegano una serie di attività legate alla gestione delle risorse di un sistema programmato. Ogni personal computer, ad esempio, viene spesso equipaggiato con un sistema operativo direttamente dal costruttore o dall'installatore in quanto la maggior parte degli utenti si limiterà ad installare dei pacchetti software aggiuntivi già pronti per l'esecuzione delle più comuni attività informatiche.

Gli utenti più esperti, invece, possono provvedere alla sostituzione o all'installazione di altri sistemi operativi o, addirittura, alla realizzazione di programmi in grado di essere eseguiti senza sistema operativo.

L'inserimento di un sistema operativo è quindi una attività discrezionale del sistemista che, tipicamente, dovrà prendere in considerazione pregi e difetti di entrambe le possibilità.

Un PC è, ad esempio, un sistema programmato discretamente complesso che richiederebbe al programmatore che decidesse di utilizzarlo senza sistema operativo una grossa attività di sviluppo per utilizzarne tutte le potenzialità garantendo, nel contempo, la compatibilità con qualsivoglia configurazione. L'entità del lavoro sarebbe invece molto minore per sistemi programmati *dedicati* o *embedded*, quali, ad esempio, sistemi di controllo industriale.

Il ruolo del sistema operativo non si limita solo alla gestione centralizzata delle periferiche, attività questa che viene spesso demandata ad altri componenti software (bios e driver), ma coinvolge anche tutte le funzioni di messa in esecuzione e sincronizzazione di altri moduli. Esistono sistemi operativi in grado di gestire, ad esempio, la coesistenza di più attività concorrenti (*task*) nella stessa CPU.

La progettazione e la realizzazione di alcuni programmi viene notevolmente semplificata se associata all'uso di sistemi operativi che consentono attività concorrenti detti, comunemente, sistemi operativi *multitasking*.

Il multitasking viene comunemente ottenuto alternando su base temporale l'esecuzione dei vari task; in altre parole ogni task ha a propria disposizione una porzione del tempo CPU. Una volta scaduta la prozione temporale a disposizione del task corrente, il contesto del task (memoria e stato dei registri CPU) viene congelato e sostituito con quello del nuovo task da eseguire che, a sua volta, era stato congelato precedentemente.

La determinazione dei *turni* di esecuzione è appannaggio dello **scheduler** che, di fatto, cerca di distribuire nel *miglior modo* il carico di lavoro.

Si noti, solo per completezza, che tale suddivisione delle risorse su base temporale detta *time-slicing*, non è l'unico modo di ottenere il multitasking (*preemptive*, *event-driven*, ecc.).

Nell'utilizzo *storico* di unix il multitasking assume la particolare interpretazione di multiprocesso in quanto ogni programma messo in esecuzione diviene un processo. Un processo è una

entità astratta propria del sistema operativo che ha l'*illusione* di essere l'unico programma in esecuzione ed è associato all'utente che lo ha messo in esecuzione.

Nei sistemi programmati che presuppongono una interazione con l'utente, inoltre, il sistema operativo fornisce, direttamente o tramite eseguibili esterni, una interfaccia di controllo che consenta di impartire comandi e di effettuare interrogazioni; questa interfaccia viene comunemente indicata con il termine *shell*.

La conoscenza di unix, pertanto, richiede una discreta padronanza dei concetti di utente e processo che lo studente otterrà attraverso una serie di esercitazioni relative alla programmazione, prima utilizzando la shell e, in seguito, il linguaggio c.

2.1 Modularità e Processi

L'aspetto forse più interessante di unix è l'estrema modularità che contraddistingue tutti gli oggetti software che lo compongono. Il sistema operativo è di fatto costituito da un modulo software abbastanza ridotto che ne costituisce il kernel; ogni attività collaterale viene demandata ad opportuni programmi che si interfacciano al kernel mediante *chiamate di sistema* o *system calls*

Anche una operazione apparentemente semplice come l'accesso di un utente al sistema coinvolge diversi componenti come indicato nella diapositiva in figura 2.1. A differenza di altri sistemi operativi, infatti, il kernel di unix non provvede direttamente all'interazione con l'utente ma si limita ad avviare un processo particolare di inizializzazione che rimarrà sempre attivo fino allo spegimento o *shutdown*; questo processo viene identificato con il nome **init** e, essendo il primo processo messo in esecuzione, sarà identificato con '0'.

Il processo init, a sua volta, provvede ad avviare altri processi secondo una sequenza definita in un opportuno file di configurazione; fra questi si trovano una o più occorrenze di processi *getty* ognuno dei quali prende controllo di un *terminale*.

Attraverso l'identificazione dell'utente, il processo getty ed il successivo processo *login* consentiranno l'accesso ai comandi tramite un interprete detto **shell**.

2.2 Gli utenti del sistema

Anche prescindendo dagli aspetti (seppur importanti) della sicurezza, ogni sistema Unix/Posix richiede l'abbinamento di ogni attività o processo ad un utente. L'accesso di un utente è verificato dal sistema di login che, tipicamente, provvede a verificare la correttezza di una coppia di stringhe dette *username* e *password*.

Il sistema provvede ad abbinare un identificativo numerico (uid) ad ogni utente che contraddistinguerà anche ogni processo avviato *a nome* dell'utente stesso.

In altri termini, ogni processo è univocamente associato ad un utente. In funzione delle caratteristiche di protezione dell'utente, al processo saranno o meno concessi dei diritti di accesso su file o direttori.

Esiste un particolare utente caratterizzato dall'avere uno uid=0, detto *root*, che può vantare ogni diritto di accesso e che, pertanto, viene utilizzato solo per attività di amministrazione o manutenzione. Cercate di vincere la tentazione ad utilizzare l'account di root per l'uso normale del vostro sistema.

2.3 Accesso ai comandi

L'accesso di un utente provoca l'avvio di un processo sulla macchina che esegue un programma detto shell; la shell è un interprete di comandi che rappresenta l'interfaccia alle richieste dell'utente ed identifica il proprio stato di *attesa comandi* con una stringa detta **prompt**.

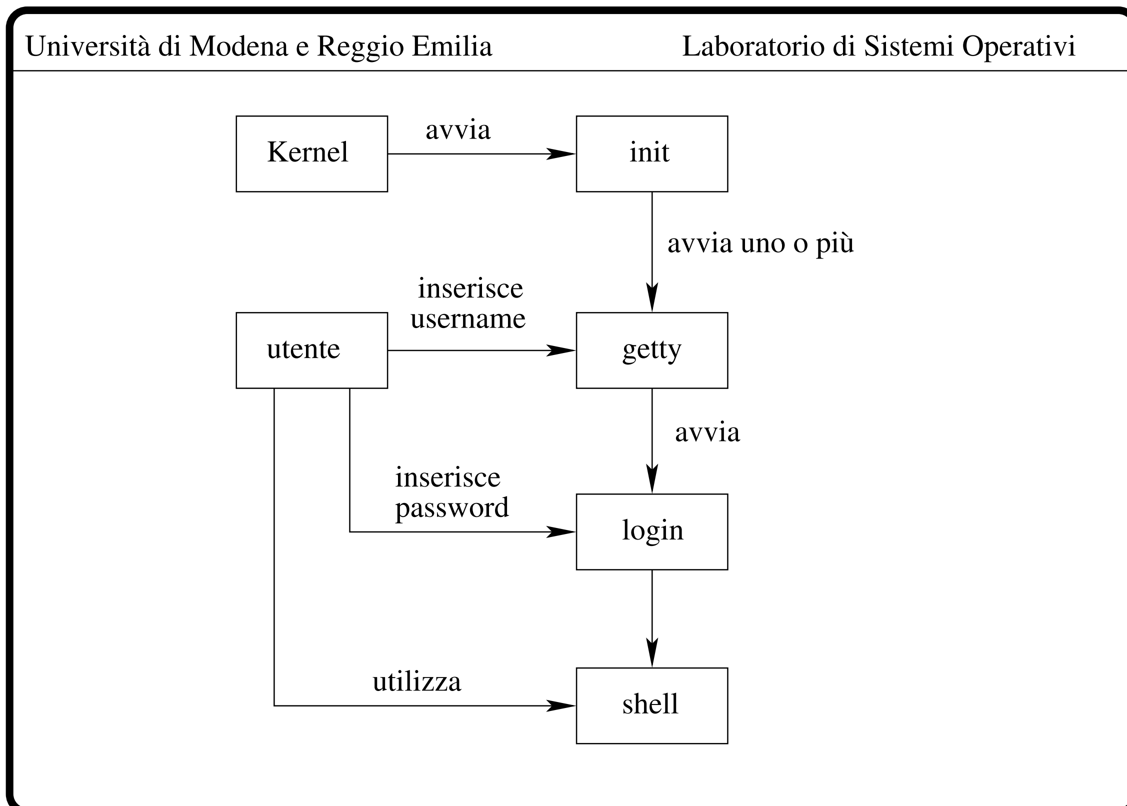

Figura 2.1: init

All'apparire del prompt, l'utente avanza le proprie richieste alla shell digitando una *command line*. Questa è una sequenza di caratteri terminata da un ritorno a capo o *newline*.

Una command line può essere composta da più elementi separati da un carattere particolare che, normalmente[1], è lo spazio. Il primo elemento della command line corrisponde ad un comando della shell o al nome di un file eseguibile, gli elementi successivi rappresentano gli argomenti del comando.

Come primo esempio di command line, chiederemo alla shell che abbiamo ottenuto sul sistema l'esecuzione di un file eseguibile di nome **ls**; questo programma mostra la lista dei file presenti nella directory corrente[2].

```
$ ls
```

I primi due caratteri della linea mostrata, ovvero il *dollaro* e lo spazio successivo, rappresenta il prompt; questo può variare da macchina a macchina ma, normalmente, viene terminato con il carattere dollaro per gli utenti comuni e con il carattere cancelletto per l'utente root.

Noi digiteremo solo i caratteri **l** e **s** seguiti da un ritorno a capo. Fatto questo, la shell provvederà ad interpretare la command line mettendo di fatto in esecuzione il programma *ls* di cui otterremo il risultato a video.

[1] I criteri di identificazione di una command line e degli argomenti sono qui volutamente semplificati rispetto alla realtà. Allo stato attuale dell'apprendimento è opportuno che lo Studente focalizzi la propria attenzione sugli aspetti fondamentali, rimandando ad un secondo tempo l'analisi dettagliata di ogni possibilità offerta dalla shell di Unix

[2] Torneremo presto sul concetto di directory corrente

2.4 Directory corrente

La semplice esecuzione del comando ls ha sollevato la necessità di approfondire alcuni concetti fondamentali:

- come sono organizzati i file in un sistema Unix, ovvero il **filesystem**
- come vengono *collegati* i dispositivi di input e output ai processi

In prima approssimazione, il filesystem di un sistema POSIX compliant è rappresentabile da una struttura gerarchica avente una unica radice /, detta *root* o *barra*, che contiene un insieme di *file*.

Il concetto di *file* in Unix è estremamente ampio e comprende, oltre ai file *regolari*, un insieme di file speciali fra cui le *directory* che altro non sono se non dei file che contengono un insieme di riferimenti ad altri file.

Dal punto di vista dell'utente, una directory è *assimilabile* ad una suddivisione del filesystem anche se, di fatto, rappresenta solo una *lista di nomi e riferimenti.*

Ogni processo POSIX ha una informazione di stato che indica un file speciale di tipo directory all'interno del filesystem che ne rappresenta la **directory corrente**.

L'accesso ai file elencati nella directory corrente di un processo si ottiene semplicemente indicando il nome del file così come appare nella directory stessa. Questo modo di indicare un file è detto **relativo semplice**.

Supponiamo, ad esempio, che l'esecuzione del comando ls di cui sopra abbia mostrato l'esistenza di un file di nome *uno*. Questo significa che nella directory corrente è presente un riferimento di *nome* **uno** ad un file.

Se chiediamo alla shell di eseguire nuovamente il comando ls con l'argomento **uno**, indichiamo a ls di restringere la lista al solo file uno.

```
$ ls uno
```

Torneremo in seguito sulla definizione di *lista di nomi* che abbiamo dato alla directory; proviamo tuttavia ad aggiungere l'opzione **-i** alla precedente command line ed avremo più chiara la funzione della directory: associa il nome *uno* all'**inode** ove *fisicamente* risiede il file.

```
$ ls -i uno
```

La directory corrente del processo associato alla nostra shell può essere visualizzata con il comando *pwd*. L'output del comando identifica il nome **assoluto** della directory corrente.

Un nome *assoluto* di un file rappresenta[3] un percorso che conduce dalla barra al file. Ogni passaggio di directory viene indicato con il carattere /.

Ogni utente delle macchine Linux del Laboratorio Base, ad esempio, potrà verificare che la directory corrente della propria shell è:

```
$ pwd
/home/n12345
```

dove, ovviamente, 12345 è sostituito dal vero numero tessera. L'output del comando pwd viene letto in questo modo: la directory corrente è elencata con il nome *n12345* nella directory elencata con il nome *home* dalla directory barra.

La directory corrente può essere tipicamente[4] modificata dall'utente mediante il comando **cd**.

Se, ad esempio, invochiamo il seguente comando:

[3] Vedremo in seguito che potrebbe non essere l'unico

[4] a volte questo viene impedito dagli amministratori per esigenze di sicurezza, invocando la shell ristretta

```
$ cd /home
```

la directory corrente della shell viene posizionata in /home; dopo questa impostazione, potremo accedere al file *uno* di cui sopra in uno qualsiasi dei seguenti modi:

- /home/n12345/uno
- n12345/uno

Il secondo modo mette in evidenza un nuovo tipo di nome di file diverso sia dal *nome assoluto* che dal *nome relativo semplice* visti prima: si tratta del **nome relativo** che perde la specifica di semplice.

Capitolo 3

Il filesystem

Una delle principali risorse condivise alle quali può accedere un processo POSIX è rappresentata dall'insieme dei file presenti nel disco[1] o **filesystem** (figura 3.1).

Con il termine filesystem si intende sia il *metodo* che l'*implementazione*, così come spesso avviene fra una classe ed il suo singleton. Il nostro interesse attuale è quello di capire il *metodo* utilizzato dai sistemi POSIX per organizzare i file e, di conseguenza, parleremo di filesystem in tal senso.

Il secondo significato del termine si utilizza, invece, quando si vuole indicare l'oggetto filesystem presente in un disco o una partizione. Esistono diversi comandi fra le utility di sistema che premettono di creare, modificare o montare vari filesystem; fra questi, ad esempio, troviamo *mke2fs* che permette di creare un filesystem di tipo EXT2 in una partizione: il nome del comando è una contrazione per *make ext2 filesystem*, nella quale appare evidente l'uso del termine filesystem nel senso di *implementazione*

Come anticipato al precedente 2.4, il filesystem ha una unica radice detta *barra* o *root*[2] , che contiene, in forma gerarchica, altre sotto directory; in prima approssimazione[3], possiamo pensare all'intera gerarchia come ad un albero avente per radice la barra.

I nomi di directory e la struttura principale dell'albero sono *discretamente* standardizzate fre i vari sistemi POSIX[4]; esistono, tuttavia, dei sistemi che preferiscono riorganizzarne completamente nomi e struttura.

Nella maggioranza dei casi, tuttavia, dovremmo individuare almeno queste sotto directory della barra:

/bin file eseguibili principali di sistema, normalmente eseguibili dagli utenti

/dev file speciali per i dispositivi (o device)

/etc file di configurazione del sistema e dei programmi

/lib librerie di sistema statiche e dinamiche

/tmp file temporanei

/usr file di uso comune per tutti gli utenti come programmi, documentazione, esempi. Spesso /usr viene *montata* in sola lettura per migliorare la velocità di accesso in lettura e la sicurezza; in tal caso la gerarchia /var viene in aiuto.

[1]Un tipico sistema Unix, anche se di piccole dimensioni, utilizza normalmente più dischi che possono eventualmente essere remoti

[2]L'utilizzo di root come appellativo della radice può creare ambiguità con la home directory dell'utente root in /root

[3] In Unix/POSIX è possibile, anzi spesso necessario, utilizzare dei link fra directory. Questi oggetti introducono nel grafo degli anelli tali da impedirne la classificazione come albero

[4]Il Linux Filesystem Hierarchy Standard, FHS, ad esempio

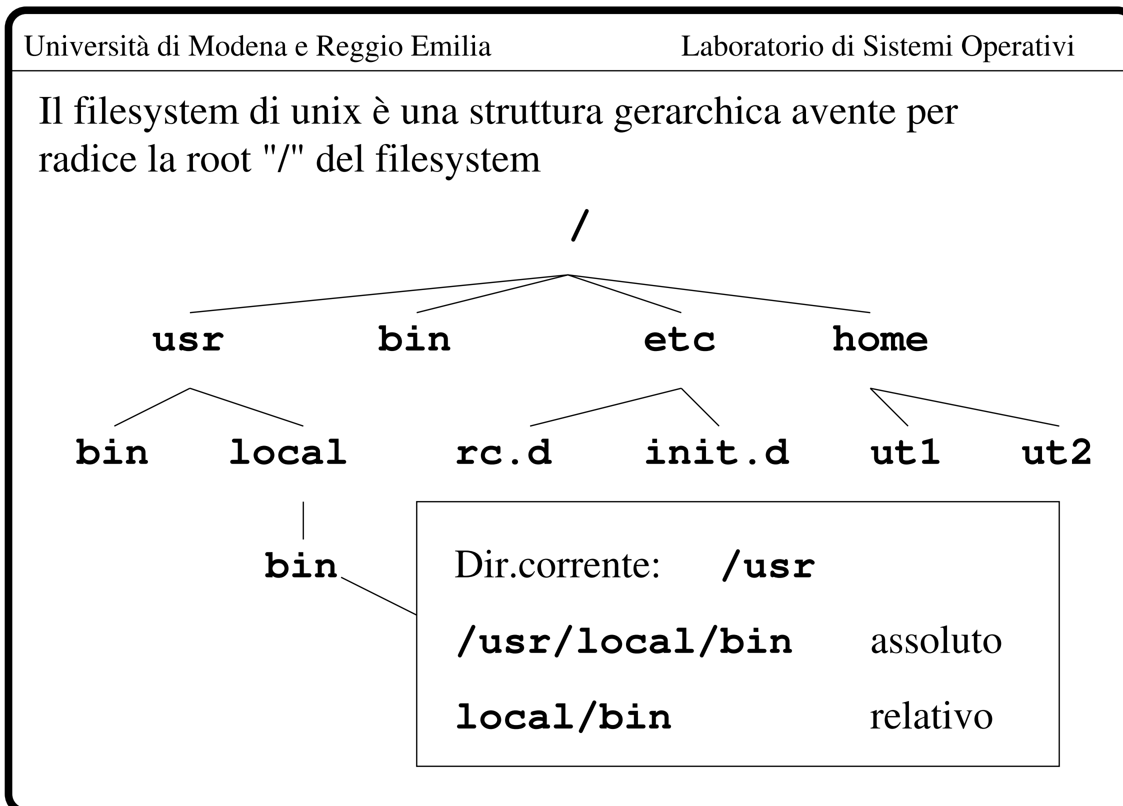

Figura 3.1: Filesystem

/var file di uso comune, come /usr, ma che richiedono l'accesso in scrittura

3.1 File e Directory

La directory è un file *particolare* o meglio, **non regolare**. In Unix, ogni oggetto che trova posto nel filesystem è un file; parleremo di file **regolari** quando questi sono meri contenitori di byte.

Un file di *tipo directory* contiene un elenco di *nomi* e la relativa associazione con un file del filesystem; questo, a sua volta, può essere di tipo directory dando luogo ad una sotto directory della precedente.

Quando invochiamo il comando ls, otteniamo la lista dei nomi di file elencati nella directory corrente. Specificando l'opzione -l otterremo la stessa lista di nomi abbinata ad una serie di informazioni:

```
-rwxr-xr-x    1 valealex users       25843 Jan 20 18:28 lso.tex
```

Vediamo in dettaglio a cosa si riferiscono:

- `-`

  tipo di file

- `rwxr-xr-x`

  diritti associati al file

- `1`

  numero di link

- `valealex`

  identificativo del proprietario

- `users`

  identificativo del gruppo che possiede il file (non necessariamente il gruppo dell'utente proprietario)

- `25843`

  dimensione del file

- `Jan 20 18:28`

  data dell'ultima modifica

- `lso.tex`

  nome associato al file nella directory

Il primo carattere identifica il tipo di file; il segno meno contraddistingue i file regolari mentre il carattere **d** indica una directory, come nella seguente linea prodotta da ls -l:

```
drwxr-xr-x    3 valealex users        4096 Jan 14 14:26 old
```

3.2 Permessi associati ai file

Ogni file contiene delle informazioni aggiuntive a quelle del proprio contenuto che servono al sistema per determinare le operazioni ammissibili ad un dato utente. Come anticipato al precedente 2.1, ogni processo è associato ad un utente; quando il processo richiede al sistema un qualunque accesso ad un file, l'utente viene abbinato all'insieme dei pemessi per determinare se l'accesso richiesto è ammissibile o meno.

Prendiamo ad esempio l'output di ls -l rappresentato in figura 3.2; la sequenza di caratteri che segue il *d* viene letto come una sequenza di informazioni binarie (bit) che viene letta come segue:

bit	carattere	significato
00400	r????????	Se 'r', accesso in lettura consentito al proprietario
00200	?w???????	Se 'w', accesso in scrittura consentito al proprietario
00100	??x??????	Se 'x', accesso in esecuzione consentito al proprietario
00040	???r?????	Se 'r', accesso in lettura consentito a chiunque appartenga al gruppo
00020	????w????	Se 'w', accesso in scrittura consentito a chiunque appartenga al gruppo
00010	?????x???	Se 'x', accesso in esecuzione consentito a chiunque appartenga al gruppo
00004	??????r??	Se 'r', accesso in lettura consentito a tutti i rimanenti utenti
00002	???????w?	Se 'w', accesso in scrittura consentito a tutti i rimanenti utenti
00001	????????x	Se 'x', accesso in esecuzione consentito a tutti i rimanenti utenti

I bit relativi all'accesso in esecuzione hanno un significato differente se applicati a file regolari o a file di tipo directory: nel primo caso controllano effettivamente la possibilità di *mettere in esecuzione* il file, mentre per le directory consentono o meno l'*esplorazione*.

In altre parole, non è possibile impostare la directory corrente di un processo su una directory alla quale non si abbia accesso in esecuzione; questo non impedisce di vederne il contenuto (controllato dall'accesso il lettura) ma dei file contenuti si potrà conoscere solo il nome.

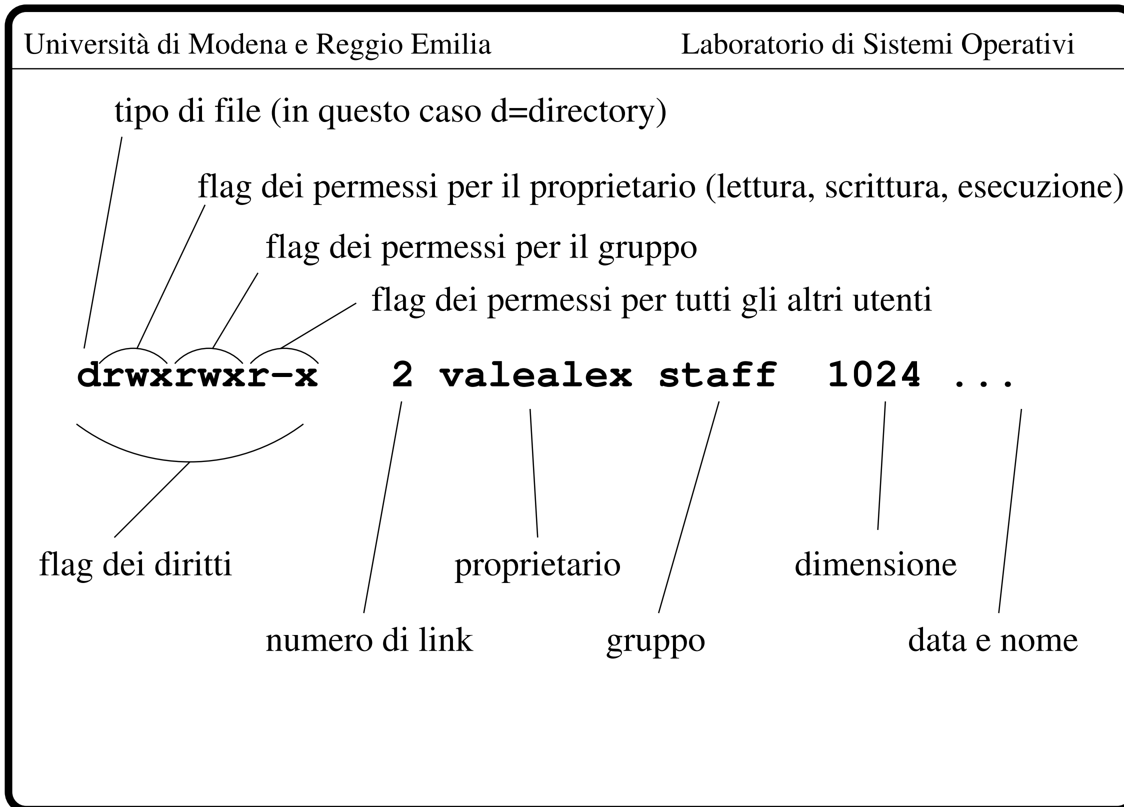

Figura 3.2: Proprietà dei file

3.3 Bit speciali

In aggiunta ai 9 bit relativi ai permessi di lettura, scrittura ed esecuzione, esistono tre bit speciali; come per i bit di esecuzione, anche i tre bit speciali cambiano lievemente significato quando applicati a file regolari o a directory.

bit	carattere	significato
04000	??s??????	Se 's', attiva il flag suid o *set user id*
02000	?????s???	Se 's', attiva il flag sgid o *set group id*
01000	??i??????t	Se 't', attiva lo sticky bit o *restriction deletion flag*

La posizione ed il significato dei tre bit speciali è indicato nella precedente tabella. Per comprendere meglio il reale funzionamento di questi bit, dobbiamo riprendere il concetto di processo anticipato al 2.1:

- ogni processo è associato ad un utente e ad un gruppo
- ogni processo esegue un programma
- il programma è contenuto in un file eseguibile

Quando si esegue un comando, in altre parole, si legge un file eseguibile e lo si *copia* nella memoria del processo. Durante questa fase vengono esaminati i bit speciali del file eseguibile e si eseguono le eventuali *promozioni* espresse dal **suid** o dal **sgid**.

In particolare, l'esecuzione di un file eseguibile con bit suid settato provoca la modifica dell'id utente associato al processo rendendolo uguale a quello del proprietario del file; per il bit sgid avviene lo stesso ma relativamente all'identificativo del gruppo che viene posto uguale a quello del gruppo che possiede il file.

Lo sticky bit, associato ad un file eseguibile, ha più una valenza storica che pratica: esso era utilizzato da alcune versioni di Unix per impedire lo swap della memoria del processo su disco quando il gestore di memoria lo avesse ritenuto opportuno. Il deposito della memoria su disco poteva rappresentare un problema di sicurezza per alcuni eseguibili per cui era necessario impedirlo almeno per certi file critici.

I moderni Unix utilizzano delle strategie di gestione della memoria e dello swap già di per se' idonei alla protezione delle informazioni presenti nei processi soggetti a swap-out per cui, tipicamente, questo bit viene ignorato o utilizzato per scopi specifici.

Per le directory, il significato dei bit speciali *suid* e *sgid* e *sticky* è diverso. Il bit suid è normalmente ignorato.

Il bit sgid è invece utile per marcare delle directory che si vogliono mantenere accessibili ad un gruppo. Normalmente, infatti, quando un utente crea un file vi associa automaticamente il proprio user id e l'id del gruppo principale[5] cui appartiene. Se crea un file o una directory all'interno di una directory con sgid attivo, invece, il gruppo del file creato sarà uguale a quello della directory superiore indipendentemente da quello dell'utente.

Il bit sticky, che applicato su una directory assume l'appellativo di *restriction deletion flag*, impedisce a tutti gli utenti di rimuovere o rinominare file presenti nella directory stessa a meno che non ne siano proprietari (o che l'utente non sia proprietario della directory).

Normalmente il bit sticky è presente in directory comuni come /tmp per assicurare ad ogni utente l'integrità dei propri file depositati pur assicurando a tutti l'accesso in lettura e scrittura.

3.4 Altre informazioni sui file

Un file di un sistema POSIX, quindi, è più dei suo semplice contenuto in quanto possiede un insieme di informazioni accessorie che comunemente vengono indicate con il termine **attributi**. Il comando ls -l, dal quale abbiamo iniziato la nostra analisi permette di visualizzarne alcune.

Oltre ai *permessi*, indicati anche come **modo di accesso** o **access mode**, un file possiede alcuni attributi temporali:

atime istante dell'ultimo accesso

mtime istante dell'ultima modifica del contenuto

ctime istante dell'ultima modifica all'i-node, ovvero agli attributi

Nell'output del comando ls -l viene riportato uno dei timestamp del file, che tipicamente è mtime.

[5] Ogni utente può appartenere a più gruppi: il gruppo indicato nel file /etc/passwd

Capitolo 4

Alcuni comandi

Nel capitolo 3 abbiamo utilizzato una delle utility fondamentali disponibili nelle varie installazioni di Unix[1], ovvero **ls**.

Quasi tutte le utility accettano delle opzioni che ne modificano il comportamento secondo regole che tipicamente dipendono dal comando stesso. Vediamo, ad esempio, alcune opzioni accettate dal comando ls:

```
 -C     List files in columns, sorted vertically.

 -F     Suffix each directory name with ‘/’, each FIFO name
with  ‘|’, and each name of an executable with ‘*’.

 -R     Recursively list subdirectories encountered.

 -a     Include files with a name starting with ‘.’ in  the
listing.

 -c     Use the status change time instead of the modifica-
tion time for sorting (with -t)  or  listing  (with
-l).

 -d     List  names of directories like other files, rather
than listing their contents.

 -i     Precede the output for the file by the file  serial
number (i-node number).

 -l     Write  (in single-column format) the file mode, the
number of links to the file, the owner  name,  the
group  name,  the  size of the file (in bytes), the
timestamp, and the filename.

The file types are as follows: -for  an  ordinary
file,  d for  a directory,  b for a block special
device, c for a character special device, l  for a
symbolic link, p for a fifo, s for a socket.
```

[1] anche Linux, OpenBSD, FreeBSD, MacOS-X, Solaris, ecc.

```
By default, the timestamp shown is that of the last
modification; the options  -c  and  -u  select  the
other two timestamps.  For device special files the
size field is commonly replaced by  the  major  and
minor device numbers.

 -q     Output  nonprintable  characters in  a filename as
question marks. (This  is  permitted  to  be  the
default for output to a terminal.)

 -r     Reverse the order of the sort.

 -t     Sort by the timestamp shown.

 -u     Use  the time of last access instead of the modifi-
cation time for sorting (with -t) or listing  (with
-l).

 -1     For single-column output.

 --     Terminate option list.
```

L'elenco sopra riportato è l'output di un altro importante comando cui faremo spesso riferimento: il **man**.

4.1 Man Pages

Anche un sistemista con anni di esperienza può avere qualche difficoltà a ricordare il significato di ogni opzione per tutti i comandi. Per questo viene in aiuto un comando, disponibile praticamente in ogni installazione, che propone una pagina di spiegazione sul programma indicato.

L'invocazione del comando man è tipicamente molto semplice; è sufficiente invocarlo dando come argomento il nome del comando di cui si desidera avere informazioni:

```
man ls
```

L'output di man viene normalmente gestito da un *pager* che consente di eseguire lo scrolling della pagina se questa occupa più linee di quelle visualizzabili a video. In tal caso è possibile usare i tasti freccia per spostarsi e 'q' per uscire.

Un altro comando che normalmente troviamo é **info** che permette di accedere alla documentazione dei pacchetti software sottoforma di ipertesti. Ovviamente, sia man che info non contengono internamente i file di documentazione ma utilizzano quelli che l'installazione di ogni pacchetto deposita in opportune *posizioni* del filesystem.

4.2 touch

Il comando touch serve, fondamentalmente, per modificare l'mtime di un file il cui nome viene passato come argomento; per le prime prove, tuttavia, lo utilizzeremo per creare file regolari vuoti.

Per creare un file di nome pluto, daremo questo comando:

```
touch pluto
```

4.3 mkdir

Un altro importante comando che ci sarà utile nelle esercitazioni è mkdir che permette di creare una directory nel direttorio corrente:

Per creare una directory di nome paperino, daremo questo comando:

```
mkdir paperino
```

4.4 chown

Gli attributi del file relativi al proprietario ed al gruppo possono essere modificati mediante il comando **chown**. Ovviamente, la modifica delle informazioni ha successo solo se il sistema verifica il diritto a compiere tale operazione.

In qualità di utenti non privilegiati non potremo verificare in pratica l'effetto di questo comando nel laboratorio.

4.5 chmod

Un comando che potremo, al contrario, provare in laboratorio è **chmod**; questo permette di modificare l'access mode di un file o di una directory dandoci la possibilità di verificare in pratica quanto detto al precedente capitolo 3

L'invocazione di chmod richiede almeno due argomenti: il primo indica la modifica che si vuole operare sul file mentre il secondo è il nome del file.

Il formato del primo argomento, quello che indica la modifica da compiere sull'access mode, permette di specificare più di una operazione ed è suddiviso in tre parti:

ugoa Un gruppo di uno o più caratteri fra 'u', 'g', 'o' ed 'a'; questi indicano rispettivamente l'utente, il gruppo, gli altri o tutti

+- Un carattere '+' per indicare il set o un carattere '-' per indicare il reset dei bit

rwxst Un gruppo di uno o più caratteri fra quelli elencati ad indicare, rispettivamente, lettura, scrittura, esecuzione, set uer/group id, sticky.

Le operazioni codificate in questo modo mnemonico permettono di agire sull'access mode in modifica, lasciando inalterati i bit che non rientrano nella richiesta; alternativamente, può essere specificato un numero ottale che rappresenta il valore richiesto della maschera di bit.

```
valealex@alex:~/work/unimo/lso/prove$ touch pippo
valealex@alex:~/work/unimo/lso/prove$ ls -l pippo
-rw-r--r--    1 valealex staff          0 Jan 21 09:44 pippo
valealex@alex:~/work/unimo/lso/prove$ chmod 'a+x' pippo; ls -l pippo
-rwxr-xr-x    1 valealex staff          0 Jan 21 09:44 pippo
valealex@alex:~/work/unimo/lso/prove$ chmod 'go-x' pippo; ls -l pippo
-rwxr--r--    1 valealex staff          0 Jan 21 09:44 pippo
valealex@alex:~/work/unimo/lso/prove$ chmod 'u+s' pippo; ls -l pippo
-rwsr--r--    1 valealex staff          0 Jan 21 09:44 pippo
valealex@alex:~/work/unimo/lso/prove$ chmod 0220 pippo; ls -l pippo
--w--w----    1 valealex staff          0 Jan 21 09:44 pippo
valealex@alex:~/work/unimo/lso/prove$ chmod 0620 pippo; ls -l pippo
-rw--w----    1 valealex staff          0 Jan 21 09:44 pippo
valealex@alex:~/work/unimo/lso/prove$ chmod 0640 pippo; ls -l pippo
-rw-r-----    1 valealex staff          0 Jan 21 09:44 pippo
```

Come esercizio, possiamo utilizzare chmod su una directory e verificare gli accessi consentiti dopo ogni modifica.

4.6 cat

Il file è un contenitore di byte; il sistema mette a disposizione alcuni comandi per estrarre o modificare il contenuto di un file invocabili da shell mediante opportune command line.

Uno di questi è il comando **cat** che, se invocato con un argomento, visualizza il contenuto del file il cui nome corrisponde all'argomento stesso:

```
cat README
```

riproduce, a video, il contenuto del file README.

L'invocazione di cat senza argomenti apre un importante argomento di studio: lo standard input. In tal caso, infatti, l'assenza dell'argomento istruisce il comando ad utilizzare, come input, un *canale* predefinito o standard. Questo, tipicamente, coincide con quello della shell che lo ha invocato, ovvero, la tastiera.

La seguente invocazione del comando cat provoca l'eco di ogni linea digitata:

```
cat
```

L'attività del comando cat termina al raggiungimento del fine file; è possibile simulare un fine file digitando ctrl-d da tastiera.

L'utilità di un simile comando sarebbe nulla se non fosse accompagnata da un altro importante strumento: la **ridirezione**.

Se inseriamo, infatti, il carattere ¿ nella command line fra la precedente invocazione ed il nome di un file otteniamo un comportamento interessante:

```
cat >nuovofile
```

Digitiamo alcune linee e completiamo con ctrl-d. Poi proviamo a visualizzare il contenuto di nuovofile; quello che otteniamo è la copia delle linee appena digitate.

Ora, utilizzando gli strumenti visti finora possiamo creare un primo **script**, semplicemente utilizzando i seguenti comandi:

```
cat >unoscript
ls -l
<ctrl-d>
chmod u+x unoscript
./unoscript
```

La progettazione e la realizzazione di script simili a questo primo esempio sarà l'obiettivo della prima parte del corso.

Capitolo 5

Usiamo la shell

Nelle precedenti lezioni abbiamo iniziato a prendere un po' di confidenza con l'interfaccia comandi di un sistema POSIX senza, tuttavia, entrare troppo nel dettaglio di ogni singolo concetto.

Ora dobbiamo approfondire la nostra conoscenza di tali concetti al punto di poterli utilizzare agevolmente nella progettazione e realizzazione degli shell script; iniziamo con uno degli ultimi: la **ridirezione**

5.1 La ridirezione

Nel precedente esempio d'uso del comando *cat* abbiamo visto come sia stato agevole richiedere alla shell di *incanalare* i caratteri prodotti in un file. Questa operazione è nota come ridirezione.

Abbiamo già visto, inoltre, come un processo sia abbinato ad una serie di informazioni di *stato*, fra cui la directory corrente o l'identificativo di utente e gruppo.

Il *canale* nel quale depositare i caratteri prodotti è, ad esempio, una delle informazioni associate ad ogni processo; utilizzando una terminologia più idonea, chiameremo tale canale **standard output** o **stdout**.

Analogamente allo standard output, ogni processo è associato ad un canale di ingresso detto **standard input** o **stdin**. Senza entrare, per ora, nello specifico meccanismo di gestione dei processi, possiamo affermare che i canali sono *ereditari*, ovvero vengono passati da ogni processo agli eventuali processi discendenti.

La shell che utilizziamo normalmente è essa stessa un processo e, di conseguenza, presenta un proprio standard input ed un proprio standard output. È facile intuire come lo standard input sia in *qualche modo* riconducibile alla tastiera e lo standard output al video[1]

Introduciamo ora un concetto nuovo, relativo al modo in cui un processo come quello *della* shell mette in esecuzione un **comando esterno**[2]: ecco cosa avviene:

- Il processo-shell crea un nuovo processo (figlio)
- Il processo-shell attende la conclusione del processo figlio
- Il processo-figlio *esegue* il file eseguibile corrispondente al comando esterno invocato
- Il processo-figlio termina
- Il processo-shell torna *attivo* e presenta il prompt

[1] Con una nomenclatura un po' arcaica, tali dispositivi sono indicati come *telescriventi* o *teletype*. Le corrispondenti entry in /dev, infatti, sono file i cui nomi iniziano o contengono i caratteri **tty**

[2] I comandi interni, ovvero quelli eseguiti dalla shell senza l'intervento di altri file, sono interpretati e codificati all'interno del *programma shell*

Come conseguenza a quanto appena visto, se ne deduce che ogni comando messo in esecuzione dalla shell ha standard output ed input *in comune* con la shell stessa. Un processo può, tuttavia, modificare sia il proprio standard input che il proprio standard output dirigendoli verso *file*[3] differenti.

In particolare, è possibile istruire la shell indicando quali canali alterare ed in quale modo: si parla, in tal caso, di ridirezione.

La ridirezione dello standard output è giaà stata vista nel paragrafo 4.6 aggiungendo il carattere di *maggiore* fra il comando ed un nome di file:

```
cat >unoscript
```

Gli elementi presenti nella command line non sono ne' argomenti ne' opzioni del comando *cat* ma istruzioni dirette alla shell. Questo significa che **non sono necessari** spazi per separare il comando dal simbolo di ridirezione o questo dal nome di file.

La ridirezione in output appena vista distrugge l'eventuale precedente contenuto del file *unoscript*. È possibile richiedere alla shell di eseguire una ridirezione in **append**, semplicemente sostituendo il simbolo di maggiore come nella seguente linea:

```
cat >>unoscript
```

Il comando *cat*, come abbiamo visto, è in grado di *aprire* un file in input nel caso in cui venga invocato con un argomento:

```
cat README
```

In questo caso, è il comando stesso ad eseguire l'apertura del file indicato ed a visualizzarne il contenuto su stdout. Lo stesso comportamento si sarebbe potuto ottenere indicando alla shell di utilizzare *quel* file come stdin:

```
cat<README
```

Le due differenti invocazioni del comando cat sul file README sono funzionalmente indistinguibili ma profondamente distinte a livello di sistema. La prima utilizza una particolare funzionalità messa a disposizione dal comando cat, ovvero quella di leggere dati da un file diverso dal proprio stdin. La seconda, al contrario, non dipende dal particolare comando invocato ma utilizza lo strumento della ridirezione messo a disposizione dalla shell.

Le operazioni di ridirezione possono essere più di una così, ad esempio, possiamo utilizzare il comando cat per copiare il contenuto di un file su un altro file:

```
cat<README>copiadiREADME
```

5.1.1 Standard Error

Il solo stdout potrebbe non essere sufficiente a contenere tutte le informazioni prodotte da un comando; potrebbe, ad esempio, essere conveniente disporre di un canale di uscita dedicato ai messaggi diagnostici o di errore. La maggioranza dei comandi disponibili, infatti, utilizza un terzo *descrittore* di **standard error** o **stderr**.

La *nostra* shell riversa sia lo standard output che lo standard error a video. Questi canali sono tuttavia distinti e la semplice ridirezione in output vista prima devia solamente i caratteri emessi su stdout. Facciamo un esempio:

Supponiamo di *trovarci* in una directory contenente un file di nome "abc": l'esecuzione del comando "ls abc", in tale directory, lista ovviamente il file. Se utilizziamo la ridirezione in output sul file "def", l'esecuzione del comando non produce nulla a video perchè tutto l'output viene trascritto in "def".

[3] Con questa affermazione abbiamo ulteriormente esteso il concetto di file

```
ls abc>def
```

Ora supponiamo di eliminare il file "abc" e di ripetere il comando "ls abc>def":

```
rm abc
ls abc>def
```

Il comando ls non può chiaramente listare il file "abc" e, di conseguenza, il file "def" che contiene l'output risulta vuoto. A video, tuttavia, appare un messaggio di errore generato dal comando ls, simile al seguente:

```
ls: abc: No such file or directory
```

Il descrittore stderr non è infatti stato ridiretto e, di conseguenza, rimane *collegato* al video. Proviamo a riformulare il comando nel seguente modo[4]:

```
ls abc 2>ghi
```

5.1.2 Ridirezioni fra descrittori

Finora abbiamo visto come ridirigere un descrittore su un file. In certe situazioni potrebbe essere conveniente utilizzare un altro descrittore come destinazione di una ridirezione.

Se, ad esempio, si volesse ridirigere sia stderr che stdout su un unico file, la soluzione di eseguire due ridirezioni separate sarebbe da sconsigliare[5]. Al contrarario viene messa a disposizione una forma particolare di ridirezione che utilizza il medesimo descrittore per due canali:

```
ls abc >abc 2>&1
```

Questa ridirezione commuta stdout sul file "abc" poi commuta stderr sull'*attuale* stdout; il risultato è che sia stderr che stdout vengono connessi a "abc". Una eventuale inversione fra le due ridirezioni proudrrebbe un risultato differente:

```
ls abc 2>&1 >abc
```

In questo caso si commuterebbe stderr su stdout *attuale*, ovvero il video, e stdout su "abc". Il risultato sarebbe quello di mantenere i messaggi d'errore a video e solo l'output su "abc".

5.2 Pipeline

La possibilità di commutare i canali di input ed output da e su file rende possibile combinare due comandi elementari per dare vita a command line più complesse; supponiamo, ad esempio, di avere a disposizione un comando che visualizzi un po' di informazioni sui processi attualmente attivi.

Questo comando esiste e si chiama **ps**. Esso ammette un numero considerevole di opzioni ma non permette di specificare il nome di un eseguibile come criterio di ricerca. Questo potrebbe sembrare un problema ed indurre qualche profano a riscrivere[6] ps per includere tale criterio di selezione.

Un sistemista più accorto, al contrario, farebbe semplicemente passare l'output di ps, eventualmente invocato con qualche opzione, in un filtro come *grep*:

---

[4] ora lo spazio fra l'ultimo carattere relativo all'invocazione del comando ed il simbolo di ridirezione "2>" è necessario altrimenti la shell avrebbe potuto interpretare il comando come "ls abc2 > ..."

[5] Le scritture sarebbero operate in modo indipendente con una conseguente possibile sovrapposizione delle parti

[6] o, visto che molto probabilmente si tratta di software Open Source, a modificarne i sorgenti

```
ps -A > tempfile
grep bash < tempfile
rm tempfile
```

Questa semplice sequenza di comandi visualizza le informazioni relative a tutti i processi che stanno eseguendo una copia della shell "bash". Potremmo scriverla in uno script ed utilizzarla tutte le volte che vogliamo.

La possibilità di assemblare più comandi fra loro è una dei punti di forza dei sistemi POSIX in quanto ha permesso ai sistemisti di concentrarsi a produrre software semplice e stabile lasciando alla versatilità della shell il compito di esplicitare ogni anche più contorto obiettivo.

Tuttavia, il dover utilizzare un file di appoggio per ogni *passaggio*, è quantomeno scomodo; al suo posto è stato introdotto un meccanismo funzionalmente equivalente e migliorativo in termini di sfruttamento della macchina: la pipe

```
ps -A | grep bash
```

Il vero vantaggio della pipe sarà chiaro quando entreremo nel dettaglio dei processi ma, allo stato attuale, ci basta notare come questo meccanismo renda possibile la realizzazione di catene o **pipeline** complesse. Anche il più visionario dei programmatori avrebbe visto con difficoltà questo possibile uso di ps:

```
ps -A | cut -c24-,1-5 | sed 's/^ *//' | sed 's/^\([0-9]*\)\( \)\(.*\)$/\3 \1/'
| sort | xargs printf '%40s %d\n'
```

5.3 Gli shell script

Saper produrre delle command line potenti è una dalle capacità findamentali del sistemista POSIX. Tuttavia, le sue *creazioni* avrebbero la vita di una sessione di lavoro e dovrebbero essere ricordate o trascritte ogni volta.

Abbiamo già visto come sia possibile creare file che contengano delle command line e metterle in esecuzione dalla shell ed appare ovvio come questo modo di lavorare sia effettivamente più comodo e produttivo. Innanzitutto possiamo utilizzare dei sistemi di editing per i nostri script più potenti dell'interprete shell; in secondo luogo possiamo inserire commenti e strutturare lo script esattamente come faremmo con un linguaggio di programmazione.

La nostra esplorazione del linguaggio di scripting inizia, appunto, dai commenti: La shell ignora ogni linea che inizia con il carattere *cancelletto*; questo ci permette di inserire tutte le linee di commento che desideriamo semplicemente facendole iniziare con questo carattere.

```
# Questo e' un commento
```

Come abbiamo visto in precedenza, in un sistema POSIX è sufficiente settare il bit di eseguibilità ad un file per renderlo eseguibile. Non esiste, pertanto, alcun legame fra nome o estensione di un file ed il suo *tipo*.

I file eseguibili vengono tuttavia *analizzati* per determinarne il tipo, specialmente per distinguere se si tratta di file in linguaggio macchina o di file da interpretare. Questo viene di norma fatto esaminando i primi caratteri del file eseguibile. Un file da interpretare si identifica mediante la sequenza di caratteri

```
#!
```

seguita dal nome assoluto dell'interprete richiesto. Per un file scritto nel linguaggio della shell **bash**, la prima linea deve essere come la seguente:

```
#!/bin/bash
```

La shell, tuttavia, tenta di interpretare ogni file che non dichiara il proprio tipo mediante un interprete di default che, normalmente, coincide con la shell stessa. Per questo motivo siamo riusciti ad eseguire il primo semplice script senza lo **sharp bang**

5.3.1 Le variabili di shell

La shell presenta alcune variabili già definite all'atto del login. Queste, tipicamente, vengono impostate a valori predefiniti dal sistemista o da opportuni file di inizializzazione che ogni utente può personalizzare.

Le variabili sono identificate mediante un *nome* preceduto dal segno dollaro. La shell provvede ad *espandere* ogni occorrenza di variabile in una command line con il proprio valore corrente: se digitiamo, ad esempio, la seguente command line:

```
echo $PATH
```

la shell espande "$PATH" con il suo contenuto attuale:

```
echo /usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games
```

e, di conseguenza, otteniamo su stdout, i seguenti caratteri:

```
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:/usr/games
```

Di consegueza, ogni volta che inseriamo una variabile all'interno di una command line o, il che' è equivalente, di una linea di script, sappiamo che la shell vi sostituirà il valore corrente prima dell'esecuzione.

La variabile $PATH rientra fra quelle che *hanno una particolare valenza* per la shell, cosìcome $PS1, $HOME, ecc. In particolare, questa contiene una lista di directory, date in forma assoluta o relativa, che vengono esplorate alla ricerca dei file eseguibili dei comandi.

Se, ad esempio, abbiamo creato uno shell script di nome **prova.sh**, dobbiamo necessariamente copiarlo in una directory elencata nel $PATH per poterlo eseguire con la seguente command line:

```
prova.sh
```

Se il file non si trova in una delle directory del $PATH abbiamo tre possibilità:

- Esplicitiamo alla shell la posizione del file (es. ./prova.sh)
- Passiamo lo script alla shell mediante ridirezione (es. bash < /home/alesval/prova.sh)
- Modifichiamo la variabile $PATH

La prima possibilità è stata già mostrata in 4.6; la seconda è una estensione del concetto di ridirezione visto prima; la terza richiede la modifica di una variabile di shell.

Per modificare una variabile di shell si utilizza una particolare command line che contiene il nome della variabile (senza il carattere $, altrimenti sarebbe espansa) seguita dal simbolo "=" e dal nuovo valore:

```
variabile=valore
```

Dopo questa command line, sarà possibile utilizzare $variabile sapendo che questo sarà espanso come "valore".

Possiamo ridefinire una variabile con lo stesso meccanismo, eventualmente chiedendo alla shell di riutilizzare il valore precedente che sarà espanso *prima* dell'esecuzione e, quindi, prima della modifica:

```
variabile=nuovo$variabile
```

5.3.2 Esportazione di variabili

Quando la shell esegue un comando, ad esempio una sottoshell per mettere in esecuzione il nostro script, le variabili sono passate al nuovo processo del quale costituiscono l'*environment* o *ambiente*. Tuttavia, la shell mantiene un elenco delle variabili cosìcome erano prima di ogni modifica ed utilizza tale elenco per costruire l'environment dei processi figli.

Se, ad esempio, abbiamo modificato $variabile con una assegnazione come quelle viste sopra, una eventuale sottoshell si troverebbe ancora la variabile $variabile al valore precedente[7]

Se si vogliono *trasmettere* le modifiche fatte su una variabile o la sua creazione ad una eventuale sottoshell, bisogna eseguire un **export** della variabile stessa.

```
variabile=variabile
export variabile
```

Eventualmente questa operazione può essere contestuale alla creazione:

```
export variabile=variabile
```

Non è in **alcun modo possibile**, invece, trasmettere una modifica ad una qualsiasi variabile fatta da una sottoshell alla shell *genitore*, cosìcome è impossibile trasmettere una variabile fra processi distinti. Si ricordi che l'unica cosa condivisa fra processi è il filesystem.

5.3.3 Gli argomenti della linea di comando

Il simbolo $ ha una valenza particolare per la shell. Anteponendo il dollaro ad una sequenza di caratteri, come abbiamo visto, è possibile ottenere l'*espansione* della variabile avente per nome proprio quella sequenza di caratteri.

Esistono, però, caratteri o sequenze di caratteri che vengono trattate in modo particolare dalla shell. Alcune sequenze *dollaro*, ad esempio, consentono l'accesso alla command line di invocazione dello script stesso.

Ogni script, infatti, viene invocato mediante una command line (come ./script.sh) nella quale, eventualmente, possono essere inseriti degli argomenti. Il numero di argomenti presenti nella command line di invocazione è accessibile con la sequenza **$#**.

Con l'ausilio dell'esempio seguente (args.sh), vediamo come sia possibile accedere ai singoli valori degli argomenti con **$1**, **$2**, ecc. o all'intera lista degli argomenti con **$***. **$0**, invece, consente di ottenere l'espansione dell'argomento di indice 0, ovvero il *nome* del comando stesso.

```
#!/bin/bash
echo Ho rilevato $# argomenti sulla command line
echo Quello con indice 0 vale $0
echo Quello con indice 1 vale $1
echo Quello con indice 2 vale $2
echo Quello con indice 3 vale $3
echo La variabile '$*' vale "$*"
echo La variabile \$\* vale \"$*\"
#Ho rilevato 3 argomenti sulla command line
#Quello con indice 0 vale ./args.sh
#Quello con indice 1 vale primo
#Quello con indice 2 vale secondo
#Quello con indice 3 vale terzo
#La variabile $* vale primo secondo terzo
#La variabile $* vale "primo secondo terzo"
```

[7] La bash ha una gestione un po' più articolata delle variabili d'ambiente come PATH; tuttavia negli script è sempre necessario utilizzare un **export** esplicito per non dipendere da particolari settaggi d'ambiente

Lo script sopra riportato contiene anche, sottoforma di commenti, lo standard output di una possibile invocazione con tre argomenti. La presenza della sequenza $* nell'output (in modo non espanso, ovviamente) è stata ottenuta mediante **quoting** con apici singoli e barre inverse (*backslash*).

5.4 Elaborazione condizionale

La shell interpreta in modo particolare anche la sequenza **$?**. Questa rappresenta il valore numerico *restituito* dall'ultimo processo avviato dalla shell.

Ogni processo Unix, infatti, è in grado di trasmettere al processo che lo ha messo in esecuzione[8] un numero all'atto della propria terminazione; questo numero si dice, infatti, valore di uscita o **exit value**. Alcuni costrutti sintattici della shell sono pertanto in grado di mettere in esecuzione un comando e di eseguire o meno certe command line in funzione dell'exit value ottenuto.

Le porzioni di codice così ottenute, pertanto, presentano una elaborazione condizionale vincolata al risultato di un processo.

Esistono due comandi particolari che restituiscono un valore sempre vero o sempre falso; in funzione della shell utilizzata e dell'installazione, tali comandi possono essere comandi esterni o interni alla shell. Nel primo caso, l'escuzione del **which** su questi comandi restituisce un nome di file eseguibile; nel secondo caso, tali comandi sono contenuti nella shell sottoforma di **built-in**

A prescindere dalla particolare implementazione di questi comandi, l'exit value segue una particolare convenzione: un exit value **uguale a zero** rappresenta una condizione **vera**; un exit value **diverso da zero** rappresenta una condizione **falsa**.

Il costrutto di programmazione condizionale mostrato nel seguente script è l'**if**. La sintassi prevede che l'if venga seguito dal comando da verificare e dalla keyword **then** nella linea successiva. Nel caso in cui non sia presente la keyword **else** le command line fra **then** e **fi** vengono eseguite solo se la condizione è vera. Nel caso in cui sia presente la keyword **else** le command line fra **then** e **else** vengono eseguite solo se la condizione è vera mentre quelle fra **else** e **fi** solo se la condizione è falsa.

```
#!/bin/bash
if true
then
echo true restituisce vero
else
echo true restituisce falso
fi
if false
then
echo false restituisce vero
else
echo false restituisce falso
fi
which true
which false
true
echo $?
false
echo $?
```

[8] Questa, per ora, è una semplificazione accettabile

```
#true restituisce vero
#false restituisce falso
#/usr/bin/true
#/usr/bin/false
#0
#1
```

5.4.1 Il comando test

Invocare un comando che restituisce, come exit value, un valore costantemente falso o vero non è sicuramente di molta utilità. Conviene, puttosto, disporre di uno strumento in grado di eseguire alcune semplici verifiche relative a stringhe, numeri o informazioni sul filesystem. Questo strumento è rappresentato dal comando **test** del quale troviamo alcuni esempi di invocazione nel seguente script:

```
#!/bin/bash
test 1 = 1
echo $?
test 1 = 2
echo $?
test 1 -eq 1
echo $?
test 1 -eq 2
echo $?
test uno = uno
echo $?
test uno = due
echo $?
test uno -eq uno
echo $?
test uno -eq due
echo $?
#0
#1
#0
#1
#0
#1
#./runtest.sh: line 14: test: uno: integer expression expected
#2
#./runtest.sh: line 16: test: uno: integer expression expected
#2
```

Il comando test può essere invocato, in alternativa, utilizzando la parentesi quadra aperta; in tal caso è convenzione terminare la command line del test stesso con una parentesi quadra chiusa:

```
#!/bin/bash
if test $# -ge 3
then
echo Ho rilevato un numero di parametri idoneo
```

```
else
echo Parametri insufficienti
fi
if [ $# -ge 3 -a $# -le 5 ]
then
echo Ho rilevato un numero di parametri idoneo
else
echo Numero di parametri non idoneo
fi
```

Esistono altri costrutti della shell che utilizzano gli exit value; il **while**, ad esempio, permette di ripetere le command line fra **do** e **done** fintanto che l'exit value del comando che segue la keyword **while** è vera.

Il comando **shift** della shell viene mostrato per la prima volta in questo esempio ma la sua utilità dovrebbe risultare piuttosto ovvia: in seguito all'esecuzione di un comando shift, la lista degli argomenti viene privata del primo elemento e, di conseguenza, viene diminuito di uno il valore si $\$\#$.

```
#!/bin/bash
echo Numero parametri = $#
echo Parametri: $*
while [ $# -gt 2 ]
do
echo eseguo shift
shift
echo I parametri sono: $*
done
echo Ora sono solo 2: $*
#Numero parametri = 6
#Parametri: a b c d e f
#eseguo shift
#I parametri sono: b c d e f
#eseguo shift
#I parametri sono: c d e f
#eseguo shift
#I parametri sono: d e f
#eseguo shift
#I parametri sono: e f
#Ora sono solo 2: e f
```

5.5 Wildcard e pathname expansion

Completiamo il quadro dei caratteri *particolari* interpretati dalla shell introducendo l'asterisco che viene espanso e, quindi, sostituito con l'elenco dei componenti della directory corrente.

In altre parole, e lo script di seguito lo evidenzia, ogni volta che la shell trova un carattere * non *protetto*, questo viene sostituito da una lista di nomi separati da spazi corrispondenti alle entry della directory corrente.

```
#!/bin/bash
# Esempio star
echo *
```

```
echo \"\*\" = '"'*'"'
lista=`echo *`
echo "$lista" = '*'
if test "$lista" = '*'
then
echo nessun file
else
echo alcuni file presenti
fi
echo "*"
echo '*'
echo '$0'
echo "$0"
expr 2 \* 5
#args.sh args.sh~ expansion.sh expansion.sh~ runtest.sh runtest.sh~ se.sh se.sh~ setest.sh set
#"*" = "*"
#args.sh args.sh~ expansion.sh expansion.sh~ runtest.sh runtest.sh~ se.sh se.sh~ setest.sh set
#alcuni file presenti
#*
#*
#$0
#./expansion.sh
#10
```

5.5.1 Il ciclo for

Torneremo rapidamente su alcuni concetti sollevati dallo script appena esposto, come *quoting* e *command substitution*, ma prima vediamo un possibile esempio d'uso della wildcard expansion. Questo utilizza il costrutto del linguaggio shell detto **for**.

Il for permette di realizzare un ciclo di ripetizione di alcune command line con un numero di iterazioni noto a priori e con una particolare sostituzione di variabile. Il concetto sarà più evidente con un esempio:

```
#!/bin/bash
# Esempio for
echo $$ sta esplorando $(pwd)
conta=0
for entry in *
do
if [ -f $entry ]
then
echo $entry file regolare
conta=$(expr $conta + 1)
fi
if [ -d $entry ]
then
echo $entry directory
cd $entry
$0
conta=$(expr $conta + $?)
cd ..
```

```
fi
done
exit $conta
```

Le command line comprese fra le keyword **do** e **done** vengono ripetute un numero di volte esattamente uguale al numero di *parole* che seguono la keyword **in** del ciclo **for**. Ad ogni iterazione la variabile *$entry* assume un valore uguale ad una delle parole stesse.

Nell'esempio, l'elenco di parole è prodotto dall'espansione di *, quindi coincide con l'elenco dei file presenti nel direttorio corrente.

In altre parole, questo costrutto esegue una iterazione per ogni file della directory corrente permettendone l'*esplorazione* del contenuto

5.5.2 L'operazione di quoting

Negli esempi appena riportati si è fatto uso di **quoting**, ovvero quello strumento che permette di inserire in uno shell script o una command line caratteri *particolari* della shell senza che questa provveda ad interpretarli in modo altrettanto particolare.

Il carattere $, ad esempio, ha una valenza particolare per la shell, in quanto rappresenta l'inizio di una variabile o di uno speciale simbolo da espandere. Facendo precedere il dollaro da un carattere *barra inversa* si ritrasforma il $ in un carattere come tutti gli altri.

Questo modo di proteggere un singolo carattere è mostrato nell'esempio iniziale degli argomenti della linea di comando. Nello stesso script, inoltre, è possibile trovare un quoting analogo fatto su una sequenza di caratteri tramite gli apici singoli.

Ogni sequenza di caratteri compresa fra apici singoli viene totalmente inibita all'espansione da parte della shell

L'ultimo metodo di quoting utilizza le dopie virgolette; questo, come indicato nell'esempio appena riportato, non inibisce l'espansione di tutti i caratteri particolari ma, piuttosto, consente di raggruppare zero, una o più parole in un unico elemento della command line.

5.6 Command substitution

Di ogni comando invocabile dalla shell, abbiamo visto come utilizzare l'exit value ma non sappiamo ancora come prelevare il flusso di caratteri emesso sullo standard output. In questo caso dobbiamo utilizzare un nuovo strumento che sostutuisce un comando con il suo standard output: la **command substitution**

In esempio star troviamo l'invocazione del comando echo * compresa fra **apici inversi**; questo è uno dei possibili modi per sostituire, nella command line, un comando con il suo risultato. Dopo la *lettura* della command line da parte della shell, questa viene modificata e, al posto dei caratteri *echo ** troviamo l'ouput di echo stesso, ovvero la lista dei file della directory corrente.

In esempio for, invece, troviamo l'invocazione del comando expr fra $(e). Questo formalismo è analogo al precedente e forza la shell a sostutuire tutta la command line di expr con il suo risultato:

```
conta=$(expr $conta + 1)
conta=1
```