Università degli Studi di Pisa
Dipartimento di Ingegneria dell'Informazione:
Elettronica, Informatica, Telecomunicazioni

Note sul sistema operativo Unix

Andrea Domenici
Scuola Superiore di Studi Universitari e di Perfezionamento
S. Anna

SEU – Servizio Editoriale Universitario di Pisa

Indice

Capitolo 1

Introduzione

Unix is a lot more complicated of course – the typical
Unix hacker never can remember
what the PRINT command is called this week...
– E. Post, "Real Programmers Don't Use PASCAL"

Lo Unix è un sistema operativo *multitasking*, *time-sharing*, e *multiutente*. È particolarmente orientato alla produzione di software (infatti è stato definito "*fabbrica di software*"), per cui dispone di una vasta scelta di strumenti di supporto all'attività dei programmatori:

- editori;
- traduttori;
- gestione del codice sorgente e delle librerie;
- elaborazione di file;
- elaborazione di testi;
- comunicazioni fra gli utenti.

Un aspetto importante di questa impostazione del sistema operativo è la *componibilità* delle applicazioni, cioè l'esistenza di meccanismi che permettono ad applicazioni diverse e progettate indipendentemente l'una dall'altra di comunicare agevolmente fra di loro, purché siano state realizzate seguendo certi semplici criteri di modularità.

Questa dispensa si basa sulle esercitazioni del corso di Calcolatori Elettronici svolte nell'AA. 1992–1993. È un ausilio per chi frequenta le esercitazioni e ovviamente non può sostituire i libri di testo ed i manuali, ma è piuttosto una prima introduzione ad alcuni aspetti del sistema operativo.

Si è cercato di usare un linguaggio il piú possibile chiaro e preciso, ma, data la natura informale del lavoro ed i limiti di spazio, molti termini sono stati definiti in modo approssimativo, altri sono stati usati prima di essere definiti, ed altri ancora sono stati considerati già noti.

Non si fa riferimento ad alcuna particolare versione dello Unix, ma bisogna tener presente che le varie implementazioni, sebbene sostanzialmente simili, spesso differiscono in numerosi dettagli, che possono essere importanti nell'uso del calcolatore. Il lettore è invitato a provare personalmente ed osservare le differenze.

1.1 Componenti del sistema

Possiamo vedere il sistema operativo come formato da due componenti principali: il *nucleo* (*kernel*), che interagisce con lo hardware, incluse le memorie di massa e i dispositivi di ingresso/uscita, e gestisce i processi ed i file, e le *applicazioni*, che si rivolgono al nucleo per ottenere i servizi richieste dalle loro funzioni. Le applicazioni si possono suddividere in tre classi: i) *interpreti di comandi*, ii) *programmi di sistema*, e iii) *programmi di utente*. Gli interpreti di comandi (*shell*) sono i programmi che permettono all'utente di interagire col sistema, di solito scrivendo comandi su un terminale. I programmi di sistema sono i vari strumenti messi a disposizione dal sistema operativo, ed i programmi utente sono quelli prodotti dagli utenti per i propri scopi. Questa suddivisione è soltanto pragmatica, poiché ogni utente può scrivere un programma (per esempio un interprete di comandi o un editore), usarlo come quelli forniti dal sistema, e metterlo a disposizione degli altri utenti. In principio, il nucleo è la sola parte del sistema operativo che non può essere

sostituita da programmi utente, ma naturalmente esistono dei meccanismi di protezione che permettono solo ad utenti privilegiati di modificare le parti essenziali del sistema.

L'interfaccia fra il nucleo e le applicazioni è costituito dalle *chiamate di sistema*. Il programmatore vede le chiamate di sistema come funzioni in linguaggio C.

1.2 Il file system

Il file system di Unix è *gerarchico*, cioè permette di raggruppare logicamente i file secondo i criteri piú convenienti per l'amministratore del sistema e per i singoli utenti. Un gruppo di file è un *directory*, e un directory viene implementato come un file contenente l'elenco dei file appartenenti al directory. Ovviamente un directory può contenerne altri, per cui il file system ha una struttura ad albero. Il file system comprende anche dei file *speciali*, che rappresentano i dispositivi di ingresso/uscita ed altre risorse assimilabili; questo permette di trattare in modo uniforme i dati provenienti sia dai file ordinari che da dispositivi o da processi. Quindi, i file sono di tre tipi:

ordinari contengono dati (p.es., programmi e testi);
directory sono elenchi di file;
speciali rappresentano dispositivi di ingresso/uscita ed altre risorse.

Un comando (o in generale un processo) che deve accedere a un file deve conoscerne la posizione nel file system, che viene indicata da un *path* (*percorso* o *cammino*). Un path è un'espressione formata da nomi di file separati dal carattere '`/`'. L'ultimo nome di un path può essere un file di qualsiasi tipo (directory, file ordinario, o speciale), gli altri (se presenti) devono essere nomi di directory.

In ogni istante, ad ogni processo è associato un *directory corrente*, e un path usato nell'esecuzione del processo (che può essere, per esempio, un comando del sistema operativo) viene interpretato

a partire dal directory corrente, a meno che il path non cominci col carattere '/'. In questo caso, il cammino inizia dalla *radice* (*root*) del file system, cioè dal directory al livello piú alto. Il simbolo '/', da solo o all'inizio di un path, è il nome del directory radice. Un path che inizia dalla radice si dice *assoluto*, altrimenti *relativo*. Il simbolo '.' rappresenta il directory corrente, e il simbolo '..' rappresenta il directory *padre* (*parent*), cioè il directory di livello immediatamente superiore.

1.3 Accesso al sistema e protezione

Un sistema operativo multiutente deve permettere alle persone autorizzate di usare le risorse disponibili evitando che qualche utente ne danneggi altri, ma permettendo la condivisione di informazioni. Lo Unix dispone di un sistema di protezione adatto a questi scopi, che sfrutta un insieme di informazioni associate a ciascun utente e a ciascun file.

Ogni utente del sistema ha un *ID di utente* (*user-ID*), un *nome di login*, e una *password*. L'ID di utente è un numero usato dal sistema operativo per identificare l'utente, il nome di login e la password sono due stringhe usate dall'utente per accedere al sistema. La password in genere non è obbligatoria, ma è sconsigliabile farne a meno.

Ogni utente appartiene ad almeno un *gruppo*, identificato da un nome e un numero (*group-ID*).

1.3.1 Protezione dei file

Gli identificatori di utente e di gruppo sono usati per la *protezione* dei file, e in generale delle risorse del sistema, da accessi indesiderati. Quando un utente crea un file, è *proprietario* (*owner*) del file. Per ogni file, sono definiti dei permessi d'accesso nei confronti del proprietario, dei membri dello stesso gruppo del proprietario, e di tutti gli altri utenti. Esiste un utente privilegiato, il *superuser*, che

ha tutti i diritti su qualsiasi file. Il nome di login del superuser è `root` e il suo ID è 0 (la sua password è segreta).

Quando un utente esegue un programma (in particolare, un comando di sistema), al processo che esegue il programma vengono associati gli ID di utente e di gruppo dell'utente stesso. Se questo processo deve compiere un'operazione (lettura, scrittura, o esecuzione) su un file, il sistema operativo controlla che il file abbia il permesso per l'utente. Per esempio, il file eseguibile dell'editor `vi` appartiene all'utente `root` (il superuser) ed è eseguibile da chiunque. Quando un utente dà il comando `vi unix.tex`, la shell attiva un processo che ha gli ID dell'utente e può leggere e modificare – per esempio – il file `unix.tex` se tale file dà all'utente gli opportuni permessi.

In determinate condizioni, è necessario che a un utente vengano attribuiti temporaneamente i diritti del superuser, o di un altro utente. Questo avviene in situazioni come la modifica della password da parte dell'utente: chi vuole cambiare la propria password deve scrivere nel file `/etc/passwd`, che contiene le password cifrate di tutti gli utenti e ovviamente ha i permessi di scrittura solo per il superuser. Nei casi di questo tipo si usa il meccanismo dell'*ID effettivo*. Gli ID *reali* di utente e di gruppo sono quelli citati piú sopra, e sono associati all'utente in modo permanente, mentre gli ID effettivi di utente e di gruppo vengono assunti temporaneamente nel corso dell'esecuzione di determinati programmi, e sono uguali agli ID del proprietario di tali programmi. Nel caso del cambiamento di password, questo avviene mediante il comando `passwd`, che appartiene al superuser. Quando `passwd` viene invocato da un un utente, il comando non viene eseguito con lo user-ID dell'utente, ma con quello del superuser, ottenendo cosí il permesso di scrittura su `/etc/passwd`. Un file eseguibile può assumere l'ID del proprietario come ID effettivo soltanto se gli è stata associata la proprietà di *Set User ID*.

1.3.2 Accesso al sistema

Il nome di login e la password vengono usati durante il *login*, cioè l'apertura di una sessione di lavoro. Quando un terminale è pronto ad essere usato per una sessione, di solito mostra sullo schermo il messaggio:

```
... login:
```

L'utente può allora scrivere il suo nome di login, seguito da ritorno carrello. Il sistema risponde con:

```
password:
```

dopo di che l'utente scrive la password, che non appare sullo schermo. A meno di errori, appare il *prompt*, e l'utente può cominciare a lavorare. Ad ogni utente è assegnato un directory, chiamato *home directory*, che di solito è chiamato col nome di login dell'utente ed è il directory corrente all'inizio della sessione.

Per terminare la sessione, si scrive il carattere '`^D`' (Control-D) o il comando `logout`.

L'insieme di informazioni e di risorse associate a ciascun utente viene indicato genericamente con la parola *account*, anche se questa si riferisce piú specificamente al calcolo di eventuali canoni per l'uso del sistema.

1.4 Alcuni comandi

Elenchiamo alcuni comandi fra i piú comuni, senza entrare in dettaglio. Per l'uso e la sintassi di questi comandi, si vedano i manuali, in particolare il manuale in linea (scrivere "`man` *nome_comando*" da terminale).

ls	(*list*) Elenca i file.
mkdir	(*make directory*) Crea un directory.
cd	(*change directory*) Cambia il directory corrente.

pwd (*print working directory*) Scrive il nome del directory corrente.

rmdir (*remove directory*) Toglie un directory (vuoto) dal file system.

more, pg Visualizza un file di testo interattivamente, una schermata per volta.

cp (*copy*) Copia file.

mv (*move*) Cambia nome e/o sposta file nel file system.

rm (*remove*) Cancella file (*senza chiedere conferme*).

lp, lpr (*line printer*) Stampa.

find Ricerca di file.

grep (*global search for regular expression and print*) Ricerca di stringhe in un insieme di file.

ed, vi, emacs Editori. L'editore `ed` lavora su singole linee, mentre `vi` ed `emacs` sono editori *full screen*. L'editore `emacs` non è standard, ma è abbastanza diffuso.

man Manuale in linea. Provare `man man`.

apropos Ricerca di informazioni. Si usa scrivendo `apropos` *parola_chiave*, dove la parola chiave descrive un argomento (per esempio `print`, `sort`, ...), e si ottiene una lista di voci del manuale attinenti all'argomento. Il comando non è disponibile su tutte le installazioni.

mail, mh Posta elettronica. Il comando `mail` apre un ambiente interattivo in cui l'utente può leggere, scrivere e tenere in ordine la propria corrispondenza. Il pacchetto `mh` è un insieme di comandi indipendenti, ognuno dei quali svolge una particolare funzione (per esempio, aggiornare la cassetta della posta, leggere un messaggio, ...).

talk Messaggi in tempo reale. Permette il colloquio fra due utenti.

bc Calcolatore aritmetico.

stty (*set teletype*) Mostra o modifica le caratteristiche del terminale, fra cui il carattere di cancellazione,

la velocità e la parità dei caratteri. Se il terminale si blocca, provare `stty sane` (anche se non si vede nulla), terminando il comando con '`^J`' invece che col ritorno carrello.

1.5 Interprete di comandi

L'interprete di comandi, o *shell*, è un programma la cui esecuzione comincia dopo che l'utente ha completato la procedura di login. Scrive un *prompt* (di solito uno dei caratteri '`$`' o '`%`'), si mette in attesa di un comando, lo esegue e torna in attesa di un nuovo comando, finché l'utente non termina la sessione scrivendo il carattere '`^D`' (Control-D) o il comando `logout`.

La forma piú semplice di comando è costituita dal path di un file eseguibile, eventualmente seguito da argomenti. Se il path contiene solo il nome del file, la shell cerca un file con tale nome in alcuni directory predefiniti. L'elenco dei directory in cui avviene la ricerca può essere modificato dall'utente, come sarà spiegato piú oltre.

Molti comandi possono avere un numero variabile di argomenti. Alcuni argomenti sono chiamati *opzioni* e servono a modificare il comportamento del programma. Di solito le opzioni sono costituite da singole lettere, eventualmente precedute dal carattere '`-`', che possono essere raggruppate variamente. Per esempio, il comando `ls` senza argomenti elenca i file del directory corrente nel formato piú semplice, cioè col solo nome, il comando `ls -l` usa il formato lungo (con data, dimensione, etc.), ed `ls -l docs` elenca in forma lunga i file del directory `docs`.

Nello scrivere comandi e nomi di file, si ricordi che lo Unix è *case sensitive*, cioè distingue le maiuscole dalle minuscole.

Di solito un comando termina con un ritorno carrello. Si possono scrivere piú comandi in una linea separandoli col carattere '`;`'.

1.5.1 Redirezione

La shell riconosce la tastiera ed il video del terminale da cui è stata attivata come, rispettivamente, *ingresso standard* e *uscita standard*. I programmi invocati entro la shell che usano l'ingresso e l'uscita standard, quindi, leggono dalla tastiera e scrivono sul video. Se usiamo un programma che, per esempio, scrive sull'uscita standard ma vogliamo che la sua uscita venga scritta su un file, allora possiamo fare una *redirezione*, cioè informare la shell che i dati in uscita dal programma devono essere dirottati su un file. Per esempio, il comando `ls` scrive sull'uscita standard l'elenco dei file nel directory corrente. Il comando

```
$ ls > elenco
```

scrive l'elenco nel file `elenco`, creandolo se non esiste già e cancellandone il contenuto precedente se esiste.

Esistono varie forme di redirezione:

`>` redirezione dell'uscita standard;
`<` redirezione dell'ingresso standard;
`>>` redirezione dell'uscita standard, concatenandola al contenuto del file;
`2>` redirezione dell'errore standard.

L'*errore standard* è il flusso dove sono inviati i messaggi di errore, e di solito corrisponde allo schermo del terminale.

1.5.2 I pipe

L'uscita standard di un comando può essere inviata all'ingresso standard di un altro comando. I due comandi vengono eseguiti in parallelo e sincronizzati in modo che il secondo aspetti i dati inviati dal primo ed il primo aspetti che i dati già inviati siano stati letti dal secondo. Questo modo di esecuzione si chiama *pipe* o *pipeline* e si specifica separando i due (o piú) comandi col carattere '`|`'. Per esempio, il comando

```
$ who | sort
```

scrive in ordine alfabetico (col comando `sort`) i nomi degli utenti collegati (ottenuti dal comando `who`).

Il comando `cat` *`file`* scrive il contenuto di un file sull'uscita standard, e spesso viene usato per immettere un file nell'ingresso di una pipeline.

1.5.3 Esecuzione in background

Quando termina l'esecuzione di un comando, l'interprete riscrive il prompt, permettendo all'utente di scrivere un altro comando. È possibile far sí che la shell accetti nuovi comandi mentre un comando precedente è ancora in esecuzione, eseguendoli in modo concorrente. Questo è particolarmente utile quando un comando ha un tempo di esecuzione molto lungo e non richiede un'interazione con l'utente, come nel seguente esempio:

```
$ cc lunghissimo.c &
7809
$ ls
```

Il carattere '`&`' alla fine del comando di compilazione indica che la shell deve eseguire tale comando in *background*, cioè in parallelo (o, piú precisamente, in modo concorrente) con i successivi comandi dati dall'utente. Il numero scritto dall'interprete (*PID*) identifica il processo che esegue il comando. Per sapere quali processi sono attivi, si usa il comando `ps`:

```
$ ps
  PID TT STAT   TIME COMMAND
 7751 co S      0:08 /usr/bin/X11/twm
 7800 p1 S      0:02 vi unix.tex
 7810 p2 S      0:00 sh
 7812 p2 R      0:00 ps
```

La prima colonna dà il numero del processo, la seconda il terminale da cui il processo è stato attivato, la terza lo stato del processo (`R`, *running*: in esecuzione; S, *sleeping*: sospeso da meno di 20 secondi), la quarta il tempo di esecuzione e la quinta il comando corrispondente.

Per abortire un processo in background o attivato da un altro terminale si usa il comando `kill -9` *`PID`*. Per abortire un processo in *foreground* (cioè non in background) si scrive il carattere '`^C`'.

Per sapere il nome del terminale che si sta usando, si usa il comando `tty`.

1.5.4 Generazione dei nomi di file

È possibile riferirsi a insiemi di file usando delle *maschere* o *modelli* (*pattern*), cioè delle parole (in questo caso nomi di file) formate da caratteri ordinari e da *metacaratteri* (*metacharacter* o *wildcard*). Questi ultimi rappresentano le parti della maschera che possono essere sostituite da altri caratteri per formare nomi di file completi. Per esempio, il comando

```
$ ls *.c
```

ottiene la stampa su terminale di tutti i nomi dei file nel directory corrente che finiscono con '`.c`', poiché il metacarattere '`*`' può essere sostituito da qualunque sequenza di caratteri che non siano spaziature.

L'azione di sostituire uno o piú metacaratteri con delle stringhe di caratteri ordinari si chiama *espansione* o *globbing*.

I metacaratteri che si possono usare sono:

`*` sostituisce zero o piú caratteri;
`?` sostituisce un solo carattere;
`[, ]` denotano *classi* di caratteri. Per esempio, l'espressione `file[0-9]` sostituisce i nomi `file0`, `file1`, ..., `file9`.

Per usare uno di questi caratteri senza che venga espanso, ma venga usato letteralmente come un carattere ordinario, occorre

"proteggerlo" usando altri caratteri speciali. Un singolo carattere può essere protetto preponendogli il carattere '\', uno o piú caratteri si possono racchiudere fra coppie di caratteri '' o "". Fra questi tre modi di proteggere caratteri dall'espansione ci sono delle differenze che saranno trattate in seguito.

Il carattere '.' viene trattato in modo speciale quando è il primo carattere del nome di un file, poiché non può essere usato per espandere metacaratteri. Inoltre, i file il cui nome inizia con '.' non vengono elencati dal comando `ls`, a meno che non si usi l'opzione '`-a`'. Questa convenzione permette di nascondere dei file la cui presenza è richiesta in un directory ma a cui non si deve accedere direttamente. Per esempio, il file `.profile` contiene dei comandi per configurare la sessione di lavoro che vengono eseguiti automaticamente dopo il login.

1.5.5 Permessi di accesso

Ad ogni file sono associati gli identificatori del proprietario e di un gruppo a cui appartiene il proprietario. Al file sono assegnati dei permessi di accesso (lettura, scrittura, ed esecuzione) per il proprietario, per gli altri membri del gruppo, e per tutti gli altri utenti. Per esempio, un utente potrebbe assegnare a se stesso i diritti di lettura e scrittura su un file, dare il diritto di lettura al resto del gruppo, e negare ogni diritto agli altri. L'insieme dei permessi di accesso di un file è chiamato il *modo* del file, e viene mostrato dal comando `ls -l` con una sequenza di dieci caratteri. Il primo carattere è il tipo del file (ordinario, directory, o dispositivo di ingresso/uscita), e gli altri nove sono i diritti di lettura (`r`), scrittura (`w`) ed esecuzione (`x`) per il proprietario, il gruppo, e gli altri. Il seguente esempio:

```
$ ls -l unix.tex
-rw-r--r--  1 andrea   95435 Sep 29 15:58 unix.tex
```

mostra che `unix.tex` è un file ordinario (`-`), con permessi di lettura e scrittura per il proprietario (`rw-`) e permessi di sola lettura per il gruppo e per gli altri (`r--`).

Per cambiare il modo di un file si usa il comando `chmod` *`modo file`*. Il modo può essere dato in forma assoluta o simbolica. La forma assoluta è un numero ottale di quattro cifre. La prima cifra controlla la proprietà di Set User ID e altre modalità d'uso del file che non tratteremo, le altre cifre corrispondono ai diritti per il proprietario, il gruppo, e gli altri. I tre bit rappresentati da ciascuna cifra sono i diritti di lettura, scrittura, ed esecuzione. Per esempio, il modo `0666` dà i permessi di lettura e scrittura ad ogni utente. La forma simbolica è una stringa in cui il primo carattere specifica gli utenti a cui si applica la modifica (`a`, *all*: tutti; `u`, *user*: proprietario; `g`, *group*: gruppo; `o`, *others*: altri), il secondo carattere rappresenta la modifica (`+`: aggiungere un diritto; `-`: togliere un diritto), ed il terzo rappresenta un diritto (`r`, `w`, `x`). Si veda il manuale per altre forme possibili e per le opzioni del comando.

Capitolo 2

Il file system

Il sistema operativo presuppone che il file system sia contenuto in una *memoria di massa*, tipicamente costituita da dischi magnetici. Qualunque sia il supporto fisico, la memoria di massa è vista dal nucleo come un insieme di *blocchi* di dimensione fissa (per esempio, 512 byte), ognuno individuato da un indirizzo fisico dipendente dal dispositivo in cui si trova. Nel caso di unità a disco, l'indirizzo di un blocco è costituito dai numeri di cilindro, di traccia e di settore. Un programmatore, invece, vede i file dello Unix come sequenze di byte. Una delle funzioni del nucleo è quindi di stabilire la corrispondenza fra questi diversi modi di strutturare l'informazione.

2.1 Gli inode

I dati contenuti in un file ordinario o in un directory vengono memorizzati in un numero sufficiente di blocchi generalmente non contigui. Ad ogni file è associato un *inode*, che è una struttura dati contenente le informazioni necessarie al nucleo per gestire il file:

- identificatore del proprietario
- identificatore del gruppo
- tipo del file
- permessi di accesso

- tempi di accesso
- contatore dei link
- dimensione del file
- indirizzi logici dei blocchi costituenti il file

Il contatore dei link è il numero di nomi distinti con cui ci si può riferire ad un file, come sarà spiegato piú oltre. Osserviamo inoltre che l'inode non contiene alcun nome del file.

L'indirizzo logico di un blocco è il suo numero d'ordine. Il driver del disco provvede a convertirlo nell'indirizzo fisico.

Gli inode dei file residenti su un disco sono raggruppati in una tabella (*ilist*) memorizzata sul disco stesso. Il numero d'ordine di un inode (*inumber*) nella lista viene usato dal nucleo (insieme al numero di device del disco) per riferirsi all'inode e quindi al file. Quando un utente o programmatore si riferisce a un file mediante un path, questo viene convertito dal nucleo in un numero di inode. Il nucleo copia in memoria gli inode dei file su cui deve operare, li aggiorna se necessario, e ricopia su disco gli inode aggiornati.

2.2 I directory

Un directory è un file contenente una sequenza di strutture, talvolta chiamate *link*, formate ciascuna dal nome di un file e dal numero di inode corrispondente. I directory permettono quindi di associare i nomi dei file visibili dall'utente e dai suoi programmi agli inumber usati dal filesystem. È possibile che piú nomi, eventualmente in piú di un directory, si riferiscano allo stesso inode, e quindi allo stesso file. Il contatore di link contenuto nell'inode è quindi il numero di nomi (path) con cui ci si può riferire al file. Naturalmente, un file può avere piú nomi ma ad ogni nome corrisponde un solo file. Il comando `ln` (*link*) aggiunge a un directory un link per un file, creando un "sinonimo" per tale file, che può risiedere in un altro directory. Il comando `mv` toglie un link e ne crea uno nuovo. Il comando `rm` toglie un link da un directory, scrivendo 0 nel campo dell'inumber. Se questo link è l'ultimo che si riferisce ad un file,

l'inode ed i blocchi dati del file vengono liberati, cioè resi disponibili per altri file.

Dato il ruolo dei directory nella struttura del file system, il loro uso è soggetto ad alcune restrizioni. Infatti, un link a un directory può essere creato solo dal superuser, e un directory può essere modificato soltanto dal nucleo. Qualsiasi utente, compreso il superuser, può modificare un directory unicamente attraverso le apposite chiamate di sistema (ed i comandi che le usano).

Il permesso di lettura su un directory permette di leggerlo, il permesso di scrittura permette di aggiungere o togliere dei link, e il permesso di esecuzione è la possibilità di cercarvi dei file. Il seguente esempio mostra l'effetto dei vari tipi di permesso su un directory. Il comando '`for ...done`' crea tre file chiamati `1`, `2`, `3` nel directory `junk`.

```
$ mkdir junk
$ for i in 1 2 3
> do
> echo hi > junk/$i
> done
$ ls -ld junk
drwxr-xr-x  2 andrea        512 Feb 12 18:55 junk
$ ls -l junk
total 5
-rw-r--r--  1 andrea          3 Feb 12 18:55 1
-rw-r--r--  1 andrea          3 Feb 12 18:55 2
-rw-r--r--  1 andrea          3 Feb 12 18:55 3
$ chmod -r junk
$ ls junk
junk: Permission denied
$ ls -l junk
junk: Permission denied
$ cd junk
$ pwd
/usr/users/andrea/junk
```

```
$ ls -l
.: Permission denied
$ cd ..
$ chmod +r junk
$ chmod -x junk
$ ls junk
1       2       3
$ ls -l junk
junk/1: Permission denied
junk/2: Permission denied
junk/3: Permission denied
total 0
$ cd junk
junk: bad directory
$
```

Il comando `ls` senza opzioni deve leggere il directory, che – ricordiamo – è un file contenente nomi di file con i rispettivi inode, quindi ha bisogno del permesso di lettura ma non del permesso di esecuzione. Il comando `ls -l`, che deve stampare informazioni quali la dimensione ed il modo dei file, deve accedere agli inode elencati nel directory, per cui ha bisogno del permesso di esecuzione. Anche il comando `cd` ha bisogno del bisogno del permesso di esecuzione. Il permesso di esecuzione per un directory è quindi la possibilità di usare le informazioni ivi contenute.

2.3 Struttura di un disco

Un'unità a disco può essere divisa logicamente in piú *partizioni* o *volumi*, che sono viste dal sistema operativo come dispositivi distinti. D'ora in poi useremo il termine "disco" per indicare un disco fisico o una partizione.

Come si è visto, il file system ha una struttura ad albero, la cui radice è il directory '`/`'. Ogni disco contiene un sottoalbero del filesystem globale, e quindi si può parlare del *file system di un disco*

per indicare la parte del filesystem globale contenuta in un particolare disco. Anche questo file system parziale ha una struttura ad albero, e la sua radice corrisponde a un nodo interno del file system globale.

Ogni disco è diviso in quattro aree:

- il *blocco di boot*
- il *superblocco*
- la *tabella degli inode*
- i *blocchi dati*

L'area dei blocchi dati comprende i blocchi allocati ai file e i blocchi liberi, che sono organizzati in una lista. Gli indirizzi dei blocchi allocati, come si è visto, sono negli inode dei rispettivi file, mentre i puntatori alla lista dei blocchi liberi si trovano nel superblocco, insieme ad altre informazioni necessarie alla gestione del disco, quali il numero di blocchi e di inode disponibili. La tabella degli inode contiene, in particolare, l'inode del directory radice del disco, in una posizione predeterminata e nota al nucleo. Il blocco di boot contiene il codice di bootstrap per l'avvio del sistema, oppure è vuoto.

Il comando `mkfs` (*make file system*) crea un file system su un disco, inizializzando la struttura sú descritta.

Osservazione La frase *file system* ha quindi tre significati:

1. La parte del sistema operativo che gestisce i file
2. l'insieme di file ordinari e directory visti dall'utente
3. l'organizzazione di un disco.

2.4 I file speciali

I file speciali rappresentano dispositivi di ingresso/uscita oppure *pipe* e *code FIFO*, due meccanismi di comunicazione fra processi che vedremo in seguito. Nel sistema Unix ci si può quindi riferire a un dispositivo come se fosse un file ordinario. Anche i file speciali sono rappresentati da inode nel file system, ma gli inode in questo

caso non contengono indirizzi di dati su disco. L'inode del file speciale corrispondente a un dispositivo contiene (oltre ai campi proprietario, gruppo, tipo, etc.) due numeri detti numero *major* e numero *minor*. Il major viene usato dal nucleo per identificare il tipo di dispositivo (per esempio linea seriale, disco, etc.) ed il minor identifica un particolare dispositivo di un certo tipo.

I dispositivi sono divisi in due categorie: quelli che sono visti come sequenze di blocchi di dimensione fissa, detti *dispositivi a blocchi*, e quelli organizzati in altro modo, chiamati convenzionalmente dispositivi *a caratteri* o *raw*. I dispositivi che possono contenere un file system sono, come si è visto, dispositivi a blocchi. I dispositivi a caratteri possono essere di varia natura, come linee seriali o dispositivi di rete. Alcuni dispositivi possono avere sia un interfaccia a blocchi che uno a caratteri.

Per creare un file speciale si usa il comando `mknod` (*make node*). Il seguente comando crea il file speciale `/dev/tty10`, di tipo `c` (a caratteri), con major 39 (per ipotesi, classe delle linee seriali), e minor 8 (l'ottava linea seriale disponibile):

```
$ mknod /dev/tty10 c 39 8
```

2.5 Montaggio di dispositivi

Abbiamo visto che il filesystem globale generalmente viene ripartito fra piú dischi, ognuno dei quali contiene un sottoalbero. Per usare il filesystem presente su un dispositivo bisogna *montare* il dispositivo, cioè comunicare al sistema operativo che: i) il dispositivo contiene un filesystem, e ii) tale file system è il sottoalbero del filesystem globale avente per radice un determinato directory.

Per esempio, immaginiamo che il device `/dev/dsk0` contenga la radice del filesystem globale, con i directory `/bin`, `/dev`, ed `/etc`. Se i file del directory `/usr` sono sul disco `/dev/dsk1`, dobbiamo:

1. creare il directory `/usr`:

   ```
   $ mkdir /usr
   ```

2. montare `/dev/dsk1` su `/usr`:

   ```
   $ mount /dev/dsk1 /usr
   ```

Il comando `umount` esegue l'operazione inversa. Il comando `mount` senza argomenti mostra quali device sono montati, e su quali directory.

Normalmente queste operazioni avvengono automaticamente all'attivazione del sistema, ma possono essere necessarie per manutenzione o per usare supporti sostituibili, come i floppy disk.

2.6 Cenno alla manutenzione del filesystem

Il nucleo mantiene in memoria copie dei blocchi dati richiesti dagli utenti e delle strutture dati del file system, come tabelle di inode e superblocchi. Le modifiche fatte a queste copie vengono riprodotte sui dischi periodicamente, in modo asincrono. A causa del ritardo fra le modifiche al file system e la loro effettiva memorizzazione su disco, degli eventi eccezionali come le cadute di tensione possono compromettere la coerenza del file system: per esempio, un file potrebbe aver cambiato la propria dimensiene, perdendo o guadagnando dei blocchi dati, senza che venga aggiornato l'inode. Per questo è importante eseguire correttamente le procedure di disattivazione del sistema (*shutdown*) prima di spengere un calcolatore.

Il programma `fsck` (*file system check*) analizza un file system individuandone le eventuali inconsistenze e lo riporta a uno stato coerente, cercando di salvare il salvabile. In particolare, se ci sono dei blocchi che non risultano né liberi né appertenenti a qualche file, li raccoglie nel directory `lost+found`.

2.7 Archiviazione dei file

Per archiviare dei file su un supporto magnetico (di solito su nastro) si usa generalmente il comando `tar` (*tape archive*). Ne diamo

qui pochi esempi, rinviando al manuale per le numerose opzioni del comando. Altri comandi usati per l'archiviazione sono `cpio` e `backup`.

- Per creare un archivio su nastro contenente tutti i file del directory `mydir`, nell'ipotesi che l'unità a nastro sia `/dev/rmt0`:

  ```
  $ tar -cvf /dev/rmt0 mydir
  ```

- Per estrarre da un archivio tutti i file di un directory:

  ```
  $ tar -xvf /dev/rmt0 mydir
  ```

- Per elencare i file contenuti in un archivio:

  ```
  $ tar -tvf /dev/rmt0
  ```

  L'archivio può anche essere un file ordinario:

  ```
  $ tar -cvf archiv.tar mydir
  ```

Oppure l'ingresso o l'uscita standard, rappresentati dal carattere '-':

```
$ zcat archiv.tar.Z | tar -xvf - mydir
```

Il file `archiv.tar.Z` è stato ottenuto comprimendo `archiv.tar` mediante il comando `compress`, ed il comando `zcat` riporta il file al formato normale (non compresso) scrivendolo sull'uscita standard.

Capitolo 3

La Bourne Shell

In questo capitolo e nel successivo esaminiamo piú in dettaglio i due interpreti di comandi piú usati, rispettivamente la Bourne shell e la C shell. Queste shell, oltre ad interpretare comandi interattivamente, possono eseguire dei file, detti *script*, contenenti dei comandi. Le shell sono quindi dei linguaggi di programmazione, che dispongono anche di istruzioni strutturate e di variabili analogamente ai linguaggi strutturati come il C o il Pascal.

Uno script deve avere il permesso di esecuzione affinché il suo nome possa essere usato come un comando. Inoltre, si può eseguire uno script passandone il nome come argomento al comando `sh` (per la Bourne shell) o `csh` (C shell). Questi comandi hanno varie opzioni che influenzano l'esecuzione dello script. Per correggere uno script, è particolarmente utile l'opzione `-x`, che fa scrivere su terminale i comandi dello script che vengono eseguiti, ma non nella loro forma originale, bensí in quella interpretata dalla shell, cioè dopo l'espansione dei metacaratteri.

Se nello home directory c'è uno script chiamato `.profile`, viene eseguito automaticamente quando ci si connette al sistema, se la shell di default è la Bourne shell.

3.1 Comandi

I comandi accettati dalla shell assumono diverse forme: un *comando semplice* è una sequenza di parole separate da spazi, una *pipeline* è una sequenza di uno o piú comandi separati da '`|`', ed una *lista* è una sequenza di una o piú pipeline separate da uno dei simboli '`;`', '`&`', `&&`, `||`, eventualmente terminata da '`;`' o '`&`'. Il significato di questi simboli è il seguente:

`;` esecuzione sequenziale;

`&` esecuzione in background;

`&&` esecuzione della pipeline successiva se la precedente ha successo;

`||` esecuzione della pipeline successiva se la precedente fallisce.

Una lista di comandi racchiusa fra '`(`' e '`)`' viene eseguita da un nuovo processo shell (*subshell*), mentre una lista di comandi racchiusa fra '`{`' e '`}`' viene eseguita senza attivare un nuovo processo shell (salvo alcune eccezioni).

Un *comando* strutturato è costruito con i comandi `for`, `case`, `if`, `while`, che vedremo in seguito.

3.2 Ingresso/uscita

Oltre alle redirezioni e i pipe già visti, la shell può inviare il suo stesso input a un comando, come per esempio:

```
$ cat <<fine
> queste due
> righe
> fine
queste due
righe
$
```

Il comando `cat` senza un nome di file copia l'ingresso standard nell'uscita standard, finché non riconosce la condizione di fine file, che da terminale viene simulata scrivendo '`^D`'. In questo esempio, invece, l'espressione `<<fine` specifica che il comando deve leggere l'igresso standard finché non incontra la stringa `fine`. Il carattere '`>`' è il *prompt secondario*, usato dalla shell per indicare che aspetta ulteriori caratteri in ingresso.

3.3 Variabili

Le variabili (o *parametri*) della shell assumono delle stringhe come valori, e non vengono dichiarate. Per assegnare un valore ad una variabile, si usa un comando come nel seguente esempio:

```
$ TERM=vt100
```

Per usare una variabile, bisogna precedere il suo nome col simbolo '`$`':

```
$ echo $TERM
```

Per riconoscere piú facilmente le variabili della shell, ci riferiremo ad esse scrivendone il nome preceduto da '`$`'.

Se vogliamo concatenare a una stringa il valore di una variabile, racchiudiamo il nome della variabile fra '`{`' e '`}`':

```
$ echo $HOME
/usr/users/andrea
$ echo ${HOME}/docs
/usr/users/andrea/docs
```

La variabile predefinita `$HOME` contiene il path dello home directory.

Per scrivere una stringa contenente spaziature o metacaratteri che non devono essere interpretati, si racchiude fra ' ' '.

Esistono alcune variabili predefinite, fra cui:

`$#` numero di argomenti passati allo script

`$-`	opzioni
`$?`	valore restituito dal comando precedente
`$$`	numero di processo
`$!`	numero di processo dell'ultimo comando in background
`$HOME`	home directory
`$PATH`	percorsi di ricerca per i comandi

Le variabili `$0`, ... , `$9` sono chiamate *parametri posizionali*, e rappresentano gli argomenti passati allo script (`$0` è il nome del file). Il paramentro `$*` contiene tutti gli argomenti, separati da spazi.

La variabile `$PATH` contiene l'elenco dei directory in cui la shell ricerca i comandi. Per eseguire un file che si trova in un directory assente da `$PATH`, occorre aggiungerlo a tale variabile.

3.4 Environment

L'*environment* o *ambiente* è l'insieme delle variabili definite in una shell, con i rispettivi valori. Le variabili predefinite di una shell sono visibili ai comandi eseguiti entro tale shell, ma se l'utente ne modifica i valori o ne crea di nuove, i nuovi valori ed i nomi delle nuove variabili non sono visibili se non vengono esportati col comando `export`, come mostra il seguente esempio, in cui viene creata una variabile `$VAR1` assegnandole il valore `xxx` e poi viene creato uno script (usando il comando `cat`) che usa tale variabile. L'esecuzione dello script avviene in una subshell, nel cui ambiente `$VAR1` non è nota se non viene esportata dalla shell in cui è stata definita.

```
$ VAR1=xxx
$ echo $VAR1
xxx
$ cat > prova
echo $VAR1
^D
```

```
$ chmod +x prova
$ prova

$ export VAR1
$ prova
xxx
$
```

Una shell eseguita all'interno di un'altra shell (come quando si esegue uno script) riceve una copia delle variabili esportate e può modificare la sua copia ma non può modificare le variabili della shell in cui è contenuta.

3.5 Generazione di nomi di file

Richiamiamo l'uso dei metacaratteri della shell e le convenzioni di *quoting* (letteralmente, *citazione letterale*) per proteggerli dall'espansione:

`?`	sostituito da qualsiasi carattere singolo
`[abc]`	sostituito da un carattere nell'insieme {`a b c`}
`[a-z]`	sostituito da un carattere nell'intervallo `a ... z`
`*`	sostituito da zero o piu' caratteri qualsiasi
`\`	protegge il carattere successivo
`'...'`	protegge i caratteri delimitati, eccetto `'` stesso
`"..."`	protegge i caratteri delimitati, eccetto `\ $ " ' ` `

3.6 Sostituzione di comandi

L'uscita di un comando può essere convertita in una stringa racchiudendo il comando fra coppie di caratteri ‘‘’’ (accento grave). Per esempio

```
$ echo "Data: `date`"
Data: Mon Mar 15 10:14:53 GMT+0200 1993
```

3.7 Comandi strutturati

3.7.1 If

Il comando `if` presenta le forme incontrate nei linguaggi Pascal-like. Il comando deve essere terminato dalla parola `fi`, e si usa `elif` per scelte multiple. La condizione dell'`if` può essere una qualsiasi lista di comandi: la condizione è vera se l'esecuzione della lista ha successo. Ogni comando restituisce al sistema operativo, al termine dell'esecuzione, un valore intero (*exit status*) che è uguale a zero se il comando termina correttamente. Il successo di una lista dipende dal successo dei comandi componenti.

```
if      lista1
then
        lista2
elif    lista3
then
        lista4
else
        lista5
fi
```

La lista usata come condizione di solito è un comando della forma `[` ***espressione*** `]`, o `test` ***espressione***, dove *espressione* è formata da variabili, stringhe, ed operatori che restituiscono un valore booleano in base al verificarsi di una condizione. Alcuni di questi operatori sono:

`-d`	l'operando è un directory
`-f`	l'operando non è un directory
`-r`	l'operando ha permesso di lettura per l'utente
`-s`	l'operando non è vuoto
`-w`	l'operando ha permesso di scrittura
`-eq`	gli operandi sono due numeri uguali
`=`	gli operandi sono due stringhe uguali

Inoltre, un'espressione può essere formata da sottoespressioni unite dagli operatori

`-a`	congiunzione
`-o`	disgiunzione
`!`	negazione
`(...)`	raggruppamento

3.7.2 For

Il parametro (*parm*) di un'istruzione `for` è una variabile che ad ogni ciclo assume un valore ottenuto da una lista di *parole*.

```
for   parm in parole
do
      lista
done
```

La lista *parole* può essere ottenuta per sostituzione di variabili, metacaratteri, o comandi. Una forma particolare del `for` è la seguente, in cui al parametro vengono assegnati, in ordine, i valori degli argomenti passati allo script, cioè i valori di `$1`, `$2`, ...:

```
for   parm
do
      lista
done
```

3.7.3 While ed Until

L'istruzione `while` è simile alla `while` del Pascal. La condizione di controllo è come nell'istruzione `if`.

```
while  lista1
do
       lista2
done
```

L'istruzione `until` esegue il ciclo se `<lista1>` fallisce.

```
until   lista1
do
        lista2
done
```

3.7.4 Break e Continue

Le istruzioni `break` e `continue` permettono rispettivamente di uscire da un costrutto iterativo o di passare all'iterazione successiva. In caso di cicli annidati, si può saltare ad n livelli di annidamento piú in alto (questo parametro è facoltativo).

```
break n
continue n
```

3.7.5 Case

Il comando `case` ha la seguente forma:

```
case parm
maschere1 ) lista1 ;;
maschere2 ) lista2 ;;
...
esac
```

dove *maschere1*, *maschere2*, ..., sono sequenze di maschere, cioè stringhe contenenti eventualmente metacaratteri, separate da '`|`'. Se *parm* corrisponde a una maschera, viene eseguita la corrispondente lista di comandi. Notare che le maschere piú generiche (per esempio `*`) devono seguire quelle piú specifiche (per esempio `pippo.*`).

3.8 Altri comandi

L'istruzione `read` legge una riga dall'ingresso standard e assegna ciascuna parola trovata nella riga alla variabile nella posizione corrispondente.

```
read variabili
```

L'istruzione `exit` termina l'esecuzione della shell (o subshell) restituendo il valore intero *stato*. Per default, il valore restituito è uguale a quello restituito dall'ultimo comando eseguito.

```
exit stato
```

Il seguente comando esegue il file *script* senza attivare una nuova shell.

```
. script
```

Il comando `trap` fa eseguire un *comando* se viene ricevuto il segnale (v. Sez. 8.5) identificato dal numero *n*:

```
trap comando n
```

Capitolo 4

La C shell

La C shell usa un linguaggio simile a quello della Bourne shell, ma dispone di caratteristiche che facilitano l'uso interattivo: il *job control* permette di controllare l'esecuzione dei comandi inviando segnali dalla tastiera, usando certe combinazioni di tasti; la *storia dei comandi* permette di ripetere dei comandi, eventualmente modificandoli; gli *alias* sono definizioni di nuovi comandi.

Nel seguito tratteremo solo le differenze dalla Bourne shell, tralasciando le caratteristiche comuni.

Ogni script in C shell deve iniziare col carattere '`#`'.

Quando una C shell viene attivata, esegue lo script `.cshrc`, se esiste nello home directory. Al login viene eseguito anche il file `.login`, e al logout il file `.logout`.

4.1 Ingresso/uscita

Per assicurarsi che una redirezione non cancelli inavvertitamente un file, si può usare il comando

```
% set noclobber
```

in seguito al quale la shell tratta come errori i tentativi di ridirigere qualcosa in un file già esistente. Per aggirare questo meccanismo,

si scrivono gli operatori di redirezione seguiti immediatamente da un punto esclamativo.

Gli operatori di redirezione che contengono il carattere '`&`' inviano in uno stesso file sia l'uscita standard che l'errore standard. Esiste una forma di questo tipo anche per il pipe.

Gli operatori di redirezione e di pipeline in C shell sono quindi i seguenti:

```
<          >&          >>!          |
>          >&!         >>&          |&
>!         >>          >>&!
```

4.2 Variabili ed environment

Le variabili della C shell vengono definite ed inizializzate col comando `set` e possono contenere sequenze di parole, ciascuna delle quali può essere selezionata con un indice. Il comando `setenv` definisce una variabile e la esporta alle subshell.

```
% set path=(/bin /usr/bin)
% echo $path
/bin /usr/bin
% echo $path[2]
/usr/bin
% setenv DISPLAY galway:0
```

Oltre alle veriabili di tipo stringa, ci sono variabili booleane e numeriche:

```
% set noclobber
% unset noclobber
% @ num=( 1 + 2 )
```

Il comando `@` serve ad assegnare un valore ad una variabile numerica.

Anche la C shell ha numerose variabili predefinite, alcune delle quali identiche a quelle della Bourne shell, eccetto che sono scritte

in minuscolo. Fra le variabili proprie della C shell, citiamo solo `$<`, che viene sostituita da una riga letta dall'ingresso standard. L'uso di questa variabile sostituisce l'istruzione `read` vista nella Bourne shell.

4.3 Generazione di nomi di file

La C shell usa gli stessi meccanismi di espansione di nomi della Bourne shell. Inoltre, la seguente espressione:

```
{parola1,parola2}
```

sostituisce la stringa `parola1` oppure `parola2`. Per esempio,

```
% ls *.{tex,log}
```

elenca tutti i file con estensione `.tex` o `.log`.

La C shell usa anche le stesse convenzioni per la citazione letterale dei metacaratteri, ma ha dei metacaratteri in piú rispetto alla Bourne shell. Uno di questi è il carattere '`!`', che non viene protetto dall'espansione nemmeno inserendolo fra apici, quindi può essere citato solo usando la barra invertita '\'. Il punto esclamativo viene usato per riferirsi alla *storia dei comandi*, come vedremo piú oltre.

Un'altro carattere interpretato in modo speciale è la tilde '`~`', che da sola viene espansa col percorso dello home directory, mentre, prefissa al nome di login di un utente, forma un'espressione che viene sostituita dal percorso dello home directory di tale utente. Per esempio:

```
% echo ~
/usr/users/andrea
% echo ~paolo
/usr/users/paolo
```

4.4 Comandi strutturati

I comandi strutturati hanno una sintassi abbastanza diversa da quelli della Bourne shell. Nel seguito si dànno schematicamente le forme piú comuni.

4.4.1 If

```
if ( espressione1 ) then
     lista1
else if ( espressione2 ) then
     lista2
...
else
     lista3
endif
```

L'espressione di controllo comprende variabili, stringhe, e:

- operatori sui file come `-d`, `-e`, `-f`, `-o`, etc.
- operatori aritmetici e logici del C
- espressioni del tipo *stringa* `=~` *maschera* e *stringa* `!~` *maschera*
- espressioni del tipo { *comando* }

L'operatore `=~` denota la *corrispondenza* (*match*) fra una stringa ed una maschera, cioè è vero se la stringa si può ottenere espandendo eventuali metacaratteri nella maschera:

```
% if ( tex =~ t* ) echo YES
YES
```

In questo esempio è stata usata una forma abbreviata del comando `if`.

L'operatore `!~` è la negazione di `=~`.

Un'espressione della forma { *comando* } è vera se il comando ha successo:

```
% if ( { ls } ) echo YES
asm.tex       lez10.tex     lez15.tex     lez4.tex
     .....
lez1.tex      lez14.tex     lez3a.tex     lez9.tex
YES
```

4.4.2 Foreach e While

I comandi `foreach` e `while` sono simili a `for` e `while` della Bourne shell:

`foreach` *variabile* `(` *parole* `)`
comandi
`end`

`while (` *espressione* `)`
comandi
`end`

4.4.3 Break e Continue

Queste istruzioni permettono di uscire da un costrutto iterativo o di passare all'iterazione successiva.

`break` `continue`

4.4.4 Repeat

Il comando `repeat` serve a ripetere un comando un certo numero di volte:

```
% repeat 3 echo ciao
```

4.4.5 Switch

Nell'istruzione `switch`, l'espressione di controllo è una *parola* che viene espansa (se contiene metacaratteri, variabili, o comandi fra accenti gravi) e confrontata con le stringhe che costituiscono le etichette dei casi alternativi. Queste etichette a loro volta possono contenere i metacaratteri '`*`', '`?`', '`[`', e '`]`'.

Per uscire da un'alternativa si usa `breaksw` (*non* `break`).

```
switch ( parola )
case stringa1 :
        comandi1
        breaksw
case stringa2 :
        comandi2
        breaksw
...
default :
        comandi
        breaksw
endsw
```

4.5 Job control

Una pipeline o una sequenza di comandi separati da '`;`' è un *job*. Ogni job ha un *numero di job* ed è costituito da uno o piú processi, ognuno dei quali ha un *numero di processo* (*PID*).

Un job può essere in esecuzione in *foreground*, in esecuzione in *background*, o *sospeso* (*stopped*). Un job in background o sospeso non può ricevere né dati né segnali dalla tastiera.

L'esecuzione di un job in foreground può essere modificata in questi modi:

1. si può abortire il job col segnale INTERRUPT, ottenuto col carattere '`^C`';

2. si può abortire il job col segnale QUIT, ottenuto col carattere '`^\`'. In questo modo si ottiene un core dump, cioè la creazione di un file con l'immagine del processo, da analizzare con un debugger;
3. si può sospendere il job col segnale STOP, ottenuto col carattere '`^Z`'. Il job si blocca in attesa di essere riattivato;
4. si può mandare il job in background sospendendolo ('`^Z`') e dando il comando `bg`.

L'esecuzione di un job in background può essere modificata in questi modi:

1. si può riportare il job in foreground col comando `fg`;
2. si può sospendere il job col comando `stop`;
3. si può abortire il job col comando `kill`.

L'esecuzione di un job sospeso può essere modificata in questi modi:

1. si può riportare il job in foreground col comando `fg`;
2. si può mandare il job in background dando il comando `bg`;
3. si può abortire il job col comando `kill`.

Un job in background viene sospeso automaticamente quando aspetta un ingresso da terminale. Deve essere riportato in foreground per ricevere l'ingresso e proseguire.

È possibile far sospendere un job in background anche quando deve scrivere un'uscita su terminale. Questo si ottiene col comando `stty tostop`.

I comandi `bg`, `fg`, `kill`, e `stop` agiscono per default sul job corrente, ma possono avere un argomento della forma `%`n, dove n è un numero di job.

Il comando `jobs` elenca i job sospesi o in background nella shell corrente. Per vedere i processi attivati in altre shell, si usa il comando `ps`, con varie opzioni.

4.6 Storia dei comandi

La C shell può memorizzare i comandi eseguiti dall'utente, che può richiamarli per rieseguirli, eventualmente dopo averli modificati. Per usare questa possibilità, conviene dare il comando

```
$ set history=40
```

per far sí che vengano memorizzati, per esempio, i 40 comandi piú recenti. Il comando `history` scrive la lista dei comandi memorizzati.

Se la lista è, per esempio,

```
1 cd src
2 ls
3 cc pippo.c
4 history
```

possiamo ripetere il comando di compilazione usando le seguenti forme:

```
% !3
% !c
```

Per rieseguire il comando immediatamente precedente:

```
% !!
```

Il comando

```
% ^stringa1^stringa2
```

riesegue il comando immediatamente precedente dopo aver sostituito la stringa `stringa1` con `stringa2`. Esistono altri modi di modificare comandi.

Per avere il numero d'ordine del comando nel prompt, scrivere

```
set prompt="\!% "
```

nel file `.cshrc`.

4.7 Alias

Il meccanismo di alias permette di creare nuovi comandi, di ridefinire comandi esistenti, o di cambiarne il nome. Per esempio:

```
alias rm 'mv \!* ~/.trash'
alias l 'ls -l \!* |more'
alias dir 'ls -l \!* |more'
alias note 'date >> ~/.MEMO; cat >> ~/.MEMO'
```

Il comando `alias` senza argomenti elenca le definizioni degli alias correnti. Il comando `unalias` *nome* cancella la definizione dell'alias per *nome*. Generalmente, gli alias vengono definiti nei file `.login` o `.cshrc`.

Capitolo 5

Make

Scrivere un programma funzionante è un processo che richiede la ripetizione di varie operazioni su un certo numero di file: creare i file sorgente, compilarli, collegarli, provare i programmi, modificare i file sorgente, ricompilare, e cosí via. Compiere tutte queste operazioni manualmente è faticoso, prende molto tempo ed è soggetto a errori. Il lavoro è reso piú complicato dalle *dipendenze* fra i vari file che costituiscono il programma: per esempio, se viene modificato un file di intestazione contenente delle dichiarazioni da includere in alcuni file in C, tutti quei file devono essere ricompilati. Se vengono ricompilati tutti i file, anche quelli che non includono il file di intestazione modificato, si spreca molto tempo per la compilazione, e se si cerca di ricompilare solo i file dipendenti da quello modificato si rischia di dimenticarne qualcuno.

Il programma `make` permette di automatizzare alcuni aspetti della manutenzione del software. Per usarlo, occorre scrivere un file, chiamato `Makefile` o `makefile`, contenente una descrizione delle dipendenze fra i file e delle operazioni richieste per ottenere ciascun file dai programmi da cui dipende. Una volta scritto questo file (ed avendo creato i file sorgente), si dà il comando `make`. Questo verifica prima di tutto se i file da produrre (file *obiettivo*, come file oggetto ed eseguibili) esistono, ed eventualmente li crea secondo le istruzioni contenute nel `makefile`. Se i file esistono già, confronta

la loro data di ultima modifica con quella dei file da cui dipendono: se questi ultimi hanno una data di modifica piú recente vuol dire che i file obiettivo non sono aggiornati, e `make` li ricrea. Tutti i file devono risiedere nel directory corrente, cioè quello in cui è stato invocato il comando `make`, però una regola del makefile può eseguire un cambiamento di directory ed eseguire `make`, ricorsivamente, nel nuovo directory.

Il seguente esempio mostra un `makefile` per produrre un file eseguibile chiamato `prog`. Questo si ottiene compilando e collegando i file sorgente `x.c`, `y.c`, e `z.c`, usando la libreria `S`. I file sorgente includono il file di intestazione `defs.h`. Questo `makefile` è formato da quattro *regole*:

```
prog : x.o y.o z.o
        cc x.o y.o z.o -lS -o prog

x.o : x.c defs.h
        cc -c x.c

y.o : y.c defs.h
        cc -c y.c

z.o : z.c defs.h
        cc -c z.c
```

In generale, una regola del makefile comprende una *dipendenza* seguita da uno o piú comandi preceduti da un carattere di tabulazione, che deve essere il primo carattere della riga:

```
obiettivi : prerequisiti
 <tab>     comando
      ....
```

dove gli `obiettivi` sono i nomi di uno o piú file da produrre, i `prerequisiti` sono i file da cui dipendono gli obiettivi, e `<tab>` è il carattere di tabulazione.

Se il comando `make` viene dato con un obiettivo come argomento, viene eseguita la regola corrispondente, altrimenti viene eseguita la prima regola del makefile. Per esempio, il seguente comando ricrea solo `z.o`, se quest'ultimo non è aggiornato:

```
% make z.o
```

Si possono scrivere dipendenze senza prerequisiti. In questo caso, di solito, l'obiettivo non rappresenta un file ma un'azione da eseguire. Se, per esempio, un makefile contiene la regola:

```
clean :
        rm *.o
```

allora si possono cancellare tutti i file oggetto mediante il comando:

```
% make clean
```

Alcune dipendenze sono implicite: per esempio, `make` sa che i file `.o` dipendono dai file `.c`, per cui il file preso come esempio si può semplificare come segue:

```
prog : x.o y.o z.o
        cc x.o y.o z.o -lS -o prog

x.o y.o z.o : defs.h
```

Generalmente, `make` riconosce anche le dipendenze fra file oggetto e file sorgente in linguaggio Fortran (`.f`), assembler (`.s`), Lex (`.l`), e Yacc (`.y`). Comunque è possibile specificare regole riguardanti l'uso di qualsiasi suffisso nei nomi di file.

In un Makefile si possono definire variabili, dette anche macro:

```
OBJS = x.o y.o z.o
LIBS = -lS

prog : ${OBJS}
        cc ${OBJS} ${LIBS} -o prog

${OBJS} : defs.h
```

Si possono usare le parentesi tonde al posto delle graffe. Alcune variabili sono predefinite, ma ovviamente possono essere ridefinite nel makefile dal programmatore. La variabile `${CC}` è uguale, per default, alla stringa `cc`; le variabili `$@` e `$?` rappresentano, rispettivamente, gli obiettivi ed i prerequisiti della regola in cui compaiono.

Nelle regole ci si può riferire anche alle variabili di shell nell'ambiente.

Le variabili cosí definite possone essere cambiate quando si esegue il make. Usando il makefile dell'esempio, il seguente comando fa sí che il programma venga compilato usando la libreria `X` invece della libreria `S`:

```
% make "LIBS = -lX"
```

L'opzione `-f` permette di usare un makefile con un nome diverso da Makefile.

Esistono programmi, quali `imake` e `makedepend` che permettono di trovare automaticamente le dipendenze fra file in linguaggio C. Alcuni compilatori hanno un'opzione con la stessa funzione.

Il comando `make` viene usato spesso per installare pacchetti software. Il `makefile` in questo caso viene distribuito insieme al software e contiene le regole per compilare, creare librerie, installare gli eseguibili nei directory opportuni, e produrre la documentazione, usando programmi per l'elaborazione di testi come `nroff` o LaTeX. Di solito ci sono anche delle regole per cancellare i file non piú necessari. L'uso di variabili migliora la portabilità fra diversi sistemi.

Capitolo 6

SCCS

Nella sezione sul programma `make` si è accennato al lavoro di continua revisione e aggiornamento dei file sorgente. Quando si modifica un file bisogna mantenere la versione precedente, sia che si voglia correggere un errore o che si introducano delle nuove funzionalità. In breve tempo il numero di versioni diventa abbastanza grande da renderne difficile la gestione, specialmente se piú persone lavorano al progetto. In questo caso occorre anche evitare che due programmatori cambino contemporaneamente la stessa versione di un file.

Lo strumento SCCS (*Source Code Control System*) aiuta a gestire le versioni dei file sorgente di un progetto. L'SCCS funziona come un "segretario" a cui i programmatori affidano i file. Questo segretario provvede a numerare ed archiviare le diverse versioni e tiene traccia delle persone che li usano. Un programmatore che ha bisogno di un file deve chiederlo al segretario, specificando se vuole soltanto leggerlo (per esempio, per compilarlo o stamparlo) oppure modificarlo. In quest'ultimo caso, il segretario può rifiutarsi di consegnare il file quando un altro programmatore ci sta lavorando. Quando un file è stato modificato, viene riconsegnato al segretario che provvede a registrare la nuova versione.

Ogni nuova versione viene memorizzata sotto forma di un *delta*, cioè dell'insieme delle differenze rispetto alla versione precedente.

Quindi il sistema SCCS mantiene una copia del file originale, e, ogni volta che viene registrata una nuova versione, calcola il nuovo delta, lo memorizza e lo identifica. È cosí possibile mantenere la storia completa delle modifiche, ed ottenere qualsiasi versione intermedia, senza per questo tenere su disco le copie integrali di tutte le versioni.

Ad ogni versione corrisponde un identificatore chiamato SID (*Sccs IDentifier*) o numero di versione, formato da due numeri separati da un punto, detti *release* e *livello*. Quest'ultimo viene incrementato automaticamente per ogni nuova versione, mentre il release per default resta uguale a 1, ma il programmatore può modificarlo per identificare cambiamenti abbastanza grossi.

L'SCCS è un pacchetto formato da numerosi comandi, corrispondenti alle varie operazioni da compiere sui file. Per facilitarne l'uso, esiste un programma di interfaccia (*front end*) costituito dal comando `sccs`, a cui si passa come primo argomento il nome di un sottocomando.

L'SCCS ha bisogno di un directory chiamato `SCCS` nel directory di lavoro del progetto. Per creare ed inizializzare questo directory SCCS si può usare questo script:

```
mkdir SCCS save
foreach i (*.[ch])
    sccs admin -i$i $i
    mv $i save/$i
end
```

Lo script crea i directory `SCCS` e `save` nel directory di lavoro, che contiene le versioni originali dei file (in questo caso, con estensione `.c` e `.h`). Quindi invoca il comando `sccs` per inizializzare il directory `SCCS`, creando per ogni file sorgente un *file storico* contenente la versione iniziale ed informazioni amministrative. I file storici non hanno diritti di scrittura nemmeno per il proprietario[1]. Dopo aver

[1]Come fa SCCS a modificare i file storici? Si ricordi la distinzione fra user-ID reale ed effettivo. Si cerchi il file eseguibile `sccs` e si esamini il modo con `ls -l`.

controllato che i file sono stati copiati correttamente in SCCS, il directory `save` può essere cancellato.

6.1 Operazioni sui file

Diamo un elenco delle principali azioni previste dall'SCCS:

- Per aggiungere un nuovo file al directory `SCCS`,

  `% sccs create` *file*

- Per chiedere una copia da usare senza modificarla,

  `% sccs get` *file*

- Per chiedere una copia da modificare,

  `% sccs edit` *file*

- Per restituire una copia modificata,

  `% sccs delta` *file*

  Questo comando chiede di scrivere un commento che viene associato al nuovo delta.
- Per restituire una copia modificata per sbaglio,

  `% sccs unedit` *file*

- Per chiedere una vecchia versione, per esempio 1.2,

  `% sccs get -r1.2` *file*

6.2 Parole chiave e informazioni

I file sorgente possono contenere delle parole chiave di identificazione che vengono espanse quando si ottiene una copia da non modificare. Alcune delle parole chiave sono:

```
%W%    nome del file, versione, data, stringa @(#)
%I%    versione
%G%    data
%Z%    stringa @(#)
```

Conviene mettere queste parole chiave in un commento all'inizio del file sorgente e anche in una variabile del programma, in modo che il file eseguibile contenga l'informazione relativa alla versione.

```
static char sccsid[] = "%W%"
```

Il comando

```
% sccs what file
```

scrive su terminale tutte le linee contenute in *file* che contengono la stringa `@(#)`. Si noti che *file* può essere un file oggetto o eseguibile.

Il comando

```
% sccs prt file
```

elenca su terminale tutti i delta del file con i relativi commenti.

Il comando

```
% sccs info
```

elenca su terminale tutti i file che sono in corso di modifica insieme agli utenti che li hanno richiesti.

6.3 SCCS e Make

Diamo un esempio di Makefile che usa SCCS:

```
SRCS= prog.c prog.h

prog : prog.o
        cc prog.o -o prog
```

```
prog.o : prog.h

sources : $(SRCS)
$(SRCS) :
        sccs get $@
```

dove la variabile del `make` `$@` assume come valori i nomi dei file che devono essere prodotti. L'ultima regola permette cosí di ottenere da SCCS i file sorgente mancanti, usando il comando `make sources`.

Capitolo 7

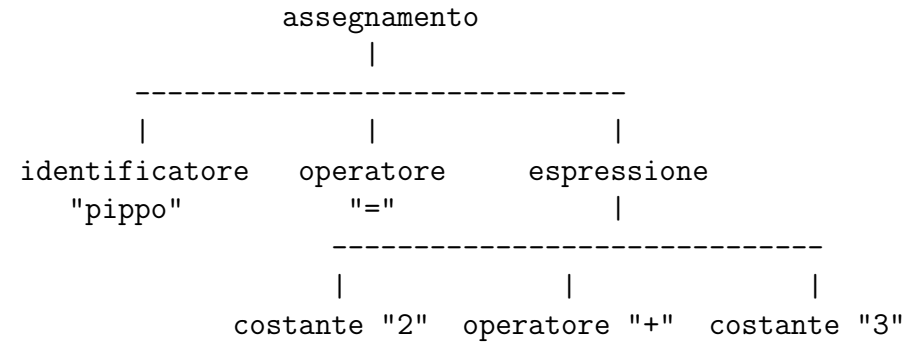
Lex e Yacc

I programmi Lex e Yacc sono due strumenti destinati alla produzione di compilatori ed interpreti, anche se in generale possono essere usati ogni volta che si debba analizzare ed eventualmente trasformare la struttura di un file di testo. Sono i rappresentanti piú evoluti di una famiglia di programmi per la manipolazione di file comprendente i programmi `sed` ed `awk`.

Un compilatore o un interprete deve costruire una rappresentazione del programma partendo dal testo sorgente. Per esempio, l'istruzione

```
pippo = 2 + 3;
```

può essere rappresentata da una struttura di questo tipo:



Questa struttura è un albero i cui nodi interni sono categorie sintattiche (`assegnamento` ed `espressione`), mentre i nodi foglia sono delle coppie formate da un *tipo* (come `operatore`, `identificatore`, `costante`) ed un *valore* ("`=`", "`pippo`", ...). Ogni foglia corrisponde ad un elemento lessicale significativo, detto *token*. La costruzione di questo albero avviene in due fasi: l'*analisi lessicale* riconosce i singoli token isolandoli dalla sequenza di caratteri del file di ingresso, quindi l'*analisi sintattica* riconosce le strutture sintattiche (cioè i diversi tipi di albero corrispondenti ad espressioni ed istruzioni del linguaggio) isolandole dalla sequenza di token prodotta dall'analisi lessicale. Di solito un compilatore alterna fra queste due fasi, costruendo strutture sintattiche man mano che i token vengono riconosciuti.

Un programma che esegue l'analisi lessicale si chiama (ovviamente) *analizzatore lessicale* o *scanner* o *tokenizer*. Un programma per l'analisi sintattica è un *analizzatore sintattico* o *parser*.

I programmi Lex e Yacc servono a generare rispettivamente anlizzatori lessicali ed analizzatori sintattici. L'ingresso di Lex è un file contenente una descrizione dei token da riconoscere, e la sua uscita è una funzione in C, capace di riconoscere i token. Analogamente, l'ingresso di Yacc è una descrizione delle strutture sintattiche, da cui si ottiene una funzione che riconosce tali strutture a partire dai token forniti da un analizzatore lessicale. Il programma generato da Yacc presuppone che l'analizzatore lessicale sia una funzione chiamata `yylex()`, e questo è appunto il nome della funzione prodotta da Lex, quindi è facile combinare i due analizzatori in un unico programma. Comunque i due strumenti possono essere usati indipendentemente l'uno dall'altro.

7.1 Lex

Un sorgente Lex è un file formato da tre sezioni: *dichiarazioni*, *regole*, e *programmi*, delle quali la prima e l'ultima sono facoltative. La sezione per le regole contiene la specifica della grammatica dei

token, e la sezione per i programmi contiene funzioni definite dal programmatore.

La grammatica dei token usa il formalismo delle *espressioni regolari*. Un'espressione regolare descrive una famiglia di stringhe di caratteri: per esempio, l'espressione `a+` descrive tutte le stringhe formate da uno o piú caratteri '`a`', come `a`, `aa`, `aaa`, ... La grammatica consiste in un elenco di espressioni regolari corrispondenti ai vari token. Il programma generato da Lex riconosce le stringhe la cui struttura corrisponde a (*matches*) una delle espressioni regolari. Ad ogni espressione regolare si può associare una *azione* che deve essere eseguita dall'analizzatore lessicale quando l'espressione viene riconosciuta. Un'azione tipica consiste nel restituire un valore intero che codifica il token al programma che chiama l'analizzatore lessicale.

Le sezioni sono separate da righe contenenti solo la stringa `%%`, come nel seguente esempio:

```
DIGIT       [0-9]
%%
{DIGIT}+    myfun("intero");
%%
void myfun( char* s ) { printf("tipo: %s\n", s); };
```

La sezione per le dichiarazioni contiene – in questo caso – la definizione della macro `DIGIT`, che equivale all'espressione `[0-9]`, che nel liguaggio delle espressioni regolari rappresenta i caratteri fra '`0`' e '`9`'. Nella sezione delle regole, ci si riferisce ad una macro scrivendola fra parentesi graffe. La grammatica contiene una sola regola, la cui espressione regolare (equivalente a `[0-9]+`, dopo aver espanso `DIGIT`) riconosce le stringhe formate da almeno una cifra decimale, e la cui azione stampa un messaggio. La sezione dei programmi contiene la definizione della funzione `myfun()`.

7.1.1 Espressioni regolari

Un'espressione regolare è formata da *caratteri testuali* e da *operatori*. I caratteri testuali sono lettere, cifre, simboli eventualmente protetti, e sequenze di escape. Gli operatori sono i seguenti:

`"` ... `"` per delimitare caratteri da interpretare letteralmente;

`\` per interpretare letteralmente il carattere successivo;

`[` ... `]` classi di caratteri, per esempio `[a-z0-9<>_]`; se la classe contiene il carattere '`-`', deve essere il primo o l'ultimo fra le parentesi quadre;

`^` complemento di una classe di caratteri, per esempio `[^abc]` rappresenta tutti i caratteri meno '`a`', `b`', '`c`';

`-` intervallo di caratteri (v. classi di caratteri);

`?` espressione opzionale, per esempio `ab?c` riconosce `ac` e `abc`;

`.` qualsiasi carattere escluso il carattere di nuova linea;

`*` zero o piú volte l'espressione precedente;

`+` una o piú volte l'espressione precedente;

`|` espressioni alternative, per esempio `ab|cd` riconosce `ab` e `cd`;

`^` inizio linea: `^abc` riconosce `abc` solo all'inizio di una linea;

`$` fine linea: `$abc` riconosce `abc` solo alla fine di una linea;

`/` contesto destro: `$abc/de` riconosce `abc` solo se seguita da `de`;

`<` ... `>` condizioni iniziali (*start conditions*): questi caratteri delimitano un identificatore che si può considerare una variabile booleana, il cui valore può essere cambiato da un'azione. Un'espressione regolare preceduta da una condizione iniziale viene usata solo quando la condizione iniziale è vera;

`{` ... `}` ripetizione: `a{2}` riconosce `aa`, `a{2,4}` riconosce `aa`, `aaa`, `aaaa`;

La concatenazione di due stringhe si esprime semplicemente scrivendo una di seguito all'altra le espressioni corrispondenti.

Le precedenze degli operatori sono come segue:

1. `*`, `+`, `?`;
2. concatenazione;
3. ripetizione

4. `^`, `$`;
5. `|`;
6. `/`,
7. `< ... >`;

7.1.2 Azioni

I caratteri di ingresso che non vengono riconosciuti da alcuna espressione regolare sono copiati sull'uscita standard. Altrimenti, quando un'espressione viene riconosciuta si esegue l'azione corrispondente specificata dal programmatore. Un'azione è costituita da una o piú istruzioni C, racchiuse nel secondo caso fra parentesi graffe. In particolare, si può usare lo statement vuoto, formato dal solo carattere '`;`'.

Nel programma generato da Lex sono definite alcune variabili che possono essere usate nelle azioni definite dal programmatore. Fra queste,

`yytext` è un array di caratteri contenente la stringa riconosciuta;
`yylen` è la lunghezza di tale stringa.

Alla fine del file viene invocata automaticamente la funzione `yywrap()` e si esamina il valore restituito: se questo è 1 l'esecuzione termina, se è 0 prosegue. Per default, `yywrap()` si limita a restituire 1, ma è possibile ridefinirla per specificare determinate azioni[1] da eseguire alla fine, come stampare messaggi e statistiche, o per proseguire l'esecuzione, magari avendo aperto un nuovo file da elaborare. Per proseguire l'esecuzione, occorre restituire il valore 0.

7.1.3 Regole ambigue

Può accadere che una stringa possa essere riconosciuta da piú espressioni regolari. Per esempio, la stringa `int` corrisponde sia

[1] Le azioni eseguite alla fine dell'esecuzione a volte vengono chiamate *wrapup*, da cui il nome della funzione.

all'espressione `int` che all'espressione `[a-z]+`. Queste ambiguità vengono risolte secondo due regole:

1. si applica la regola che riconosce la stringa piú lunga;
2. fra regole che riconoscono lo stesso numero di caratteri, si sceglie la prima.

Per esempio, data la grammatica

```
%%
int      { /* 1 */  return( KEYWORD ); }
[a-z]+   { /* 2 */  return( IDENTIFIER ); }
```

se l'ingresso dell'analizzatore contiene la stringa `integer` la sequenza `int` non verrà riconosciuta dalla regola `1`, ma verrà letta col resto della stringa, che sarà riconosciuta (come identificatore) dalla regola `2`, poiché quest'ultima regola in questo caso permette di riconoscere un maggior numero di caratteri. Se l'ingresso contiene la stringa `int` seguita da un carattere che non sia una lettera minuscola, allora tutte e due le regole riconoscono lo stesso numero di caratteri, per cui si applica la regola `1` e la stringa viene riconosciuta come parola chiave.

7.1.4 Esempio

Il seguente esempio è una semplice grammatica per riconoscere i token usati in espressioni di assegnamento formate da identificatori, costanti intere ed operatori aritmetici e di assegnamento. Un identificatore è formato da uno o piú caratteri, il primo dei quali è una lettera o una sottolineatura '_' e gli altri posono essere lettere, cifre o sottolineature. Un intero è costituito da una o piú cifre.

```
%%
[a-zA-Z_][a-zA-Z_0-9]*    return( IDENTIFIER );
[0-9]+                    return( INTEGER );
"="                       return( '=' );
"+"                       return( '+' );
```

```
"-"                     return( '-' );
"*"                     return( '*' );
"/"                     return( '/' );
```

I token degli identificatori e le costanti intere sono rappresentati dalle costanti simboliche `IDENTIFIER` ed `INTEGER`, il cui valore può essere definito nel sorgente Lex mediante direttive `#define` o eventualmente nel sorgente Yacc, nel modo che sarà illustrato piú avanti. I token degli operatori, formati da un solo carattere, sono rappresentati dal codice ASCII dei caratteri stessi.

7.2 Yacc

Lo Yacc (Yet Another Compiler-Compiler) ha come ingresso un'insieme di regole che definiscono una grammatica e produce un programma in C (*parser* o *analizzatore sintattico*) capace di riconoscere le strutture descritte dalle regole. A ogni regola può essere associata un'*azione* da eseguire quando la regola viene soddisfatta.

Le regole descrivono strutture sintattiche (p.es. espressioni, istruzioni etc.) come sequenze di *token*. Per estrarre una sequenza di token dalla sequenza di caratteri che costituisce l'ingresso dell'analizzatore sintattico, questo chiama una funzione (*analizzatore lessicale*) che legge i caratteri d'ingresso e riconosce i token. Questa funzione, fornita dal programmatore, può essere generata dal Lex.

7.2.1 File sorgente

Anche i file Yacc sono formati dalle tre sezioni *dichiarazioni*, *regole*, e *programmi*, delle quali la prima e l'ultima sono facoltative. La sezione per le dichiarazioni contiene delle direttive di vario tipo, in particolare per dichiarare token e variabili. La sezione per le regole contiene la specifica della grammatica, e la sezione per i programmi contiene funzioni definite dal programmatore.

7.2.2 Regole

Una regola è la definizione di una struttura sintattica identificata da un *simbolo non terminale*. I token sono detti *simboli terminali*, e sono denotati da costanti simboliche o da caratteri fra apici. In questo caso, il numero intero corrispondente al token è la codifica del carattere stesso. I nomi dei token possono essere dichiarati per mezzo della direttiva `%token` nella sezione delle dichiarazioni, per esempio

```
%token IDENT INT FLOAT OP
```

Una regola ha la forma

```
non_terminale  :  corpo  ;
```

dove il corpo della regola è una sequenza di simboli terminali o non terminali separati da spaziature. Per esempio:

```
stat  :  IF '(' espr ')' stat ;
```

dove `stat` è la struttura definita dalla sequenza composta dai token `IF` e `'('` seguiti dal simbolo non terminale `espr`, dal token `')'` e dal non terminale `stat`.

Fra i simboli non terminali che appaiono in una grammatica, ce n'è uno chiamato *simbolo di partenza* (*start symbol*) che rappresenta la struttura di livello piú alto che deve essere riconosciuta dall'analizzatore sintattico. Il simbolo di partenza viene designato scrivendone il nome in una direttiva `%start`, o piú semplicemente scrivendo la prima regola che lo definisce all'inizio della sezione delle regole.

Quando un simbolo viene definito da piú regole alternative, queste possono essere raggruppate scrivendo tale simbolo una volta sola e separando i corpi col carattere '`|`'. Cosí, le regole

```
stat  :  IF '(' espr ')' stat ;
stat  :  IF '(' espr ')' stat ELSE stat ;
```

si possono scrivere anche come

```
stat  :  IF '(' espr ')' stat
      |  IF '(' espr ')' stat ELSE stat
      ;
```

Le azioni eventualmente associate alle regole sono frammenti in linguaggio C racchiusi fra parentesi graffe:

```
stat  :  IF '(' espr ')' stat
              { printf("Un if semplice.\n"); }
      |  IF '(' espr ')' stat ELSE stat
              { printf("Un if-else.\n"); }
      ;
```

Un'azione può apparire anche entro il corpo di una regola, invece che alla fine:

```
stat  :  IF '('
              { printf("Inizio espressione.\n"); }
         espr
              { printf("Fine espressione.\n"); }
         ')' stat
      ;
```

Una regola può restituire un valore, che può essere letto dalle azioni di altre regole nel cui corpo appare il simbolo definito dalla regola. A questo scopo viene usata una notazione speciale: il simbolo `$$` è una variabile che contiene il valore da restituire quando la regola viene riconosciuta, mentre i simboli `$1`, `$2`, ... sono variabili contenenti i valori restituiti dai simboli che appaiono nel corpo di una regola. Per esempio, nella regola

```
expr  :  term '+' term
              { $$ = $1 + $3; }
      ;
```

il valore di `expr` (`$$`) è la somma dei valori restituiti dalla regola che riconosce `term` a sinistra (`$1`) e a destra (`$2`) del token `'+'`.

7.2.3 Ambiguità

Un insieme di regole sintattiche è *ambiguo* quando esiste qualche sequenza d'ingresso che può essere riconosciuta in piú di un modo. Per esempio, consideriamo le regole

```
stat  :  IF '(' espr ')' stat             /* (1) */
      |  IF '(' espr ')' stat ELSE stat   /* (2) */
      ;
```

La sequenza d'ingresso

```
IF '(' espr1 ')' IF '(' espr2 ')' stat1 ELSE stat2
```

dove `espr1`, `espr2`, `stat1` e `stat2` rappresentano generiche sequenze di token, può essere fatta corrispondere a due strutture diverse:

1. Quando la sequenza
   `IF '(' espr1 ')' IF '(' espr2 ')' stat1` è stata letta, si applica la regola `(1)` alla sequenza
   `IF '(' espr2 ')' stat1`, *riducendola* al simbolo `stat`. La sequenza d'ingresso letta fino a questo punto diventa

   ```
   IF '(' espr1 ')' stat
   ```

   e proseguendo la lettura si ottiene
   `IF '(' espr1 ')' stat ELSE stat2`, a cui si applica la regola `(2)`. La sequenza d'ingresso è stata quindi analizzata come

   ```
   IF '(' espr1 ')' {
           IF '(' espr2 ') stat1
   } ELSE stat2
   ```

   dove le parentesi graffe sono usate come in C per mostrare il raggruppamento.

2. Quando la sequenza
   `IF '(' espr1 ')' IF '(' espr2 ')' stat1` è stata letta, si continua a leggere l'ingresso (operazione chiamata *shift*) per vedere se aggiungendo nuovi token a quelli già letti si può applicare un'altra regola. Quindi, dopo aver letto `ELSE` e `stat2`, si applica la regola `(2)` alla sequenza

   ```
   IF '(' espr2 ')' stat1 ELSE stat2
   ```

   ottenendo la sequenza `IF '(' espr1 ')' stat`, a cui si applica la regola `(1)`. In questo modo, la sequenza d'ingresso è stata analizzata come

   ```
   IF '(' espr1 ')' {
           IF '(' espr2 ') stat1 ELSE stat2
   }
   ```

Quando un'insieme di regole fa sí che per qualche sequenza d'ingresso siano possibili sia un'operazione del tipo 1 che una del tipo 2, si dice che esiste un *conflitto shift/reduce*. Un'altro tipo di ambiguità è il *conflitto reduce/reduce*, che si verifica quando due regole possono essere applicate ad una stessa sequenza d'ingresso.

Per risolvere le ambiguità, il programmatore può cercare di riscrivere la grammatica, magari introducendo nuovi simboli non terminali, oppure fare una di queste cose:

1. dichiarare esplicitamente *precedenza* ed *associatività* di alcuni token;
2. affidarsi alle regole implicite di risoluzione dello Yacc.

Il primo metodo è particolarmente utile nella descrizione di espressioni aritmetiche. Per specificare l'associatività degli operatori si scrivono le direttive `%left` o `%right`, seguite dai token corrispondenti, nella sezione delle dichiarazioni all'inizio del file, e la precedenza degli operatori aumenta "dall'alto in basso" secondo l'ordine testuale delle direttive. Per esempio, data la specifica

```
%left '+' '-'
%left '*' '/'
%%
espr : espr '+' espr
     | espr '-' espr
     | espr '*' espr
     | espr '/' espr
     | IDENT
     ;
```

la sequenza `c*d-e-f*g` viene raggruppata come `((c*d)-e)-(f*g)`.

Il secondo metodo, utile per istruzioni strutturate, si basa su queste regole:

1. in un conflitto shift/reduce, viene scelta l'azione di shift;
2. in un conflitto reduce/reduce, viene applicata la regola che appare per prima in ordine testuale.

7.3 Uso di Lex e Yacc

Per usare un analizzatore lessicale prodotto da Lex, si può seguire questo metodo:

1. scrivere la grammatica per l'analizzatore lessicale, per esempio `gramm.l`, includendovi il file `y.tab.h`, generato dallo Yacc, che contiene le dichiarazioni usate da `yyparse()`. In questo modo le dichiarazioni, fra cui le definizioni dei token, sono rese visibili a
   `yylex()`:

   ```
   %{
   #include y.tab.h
   %}
   %%
       /* sez. regole */
   ```

2. scrivere il sorgente Yacc, per esempio `sintassi.y`, usando la direttiva `%token` per definire i token;
3. generare l'analizzatore lessicale e l'analizzatore sintattico, ottenendo i file `lex.yy.c` (generato dal Lex), `y.tab.c`, e `y.tab.h` (generati dallo Yacc):

   ```
   % lex gramm.l
   % yacc -d sintassi.y
   ```

   L'opzione `-d` crea il file `y.tab.h`.
4. compilare i file ottenuti ai passi precedenti, eventualmente insieme ad altri moduli:

   ```
   % cc main.c y.tab.c lex.yy.c -ll -ly
   ```

Le opzioni `-ll` e `-ly` fanno collegare le librerie di Lex e Yacc. In alcune installazioni queste librerie non ci sono, ed alcune funzioni devono essere fornite dal programmatore.

Se `y.tab.c` viene compilato da solo, la funzione `main()` viene presa dalla libreria dello Yacc. Questa funzione principale si limita a chiamare `yyparse()`, che restituisce 0 o 1 a seconda che il programma riesca a riconoscere il simbolo di partenza o no. Di solito l'analizzatore sintattico fa parte di un programma piú complesso, come un interprete o un compilatore, a cui viene collegato.

Capitolo 8

Primitive Unix

Le *primitive*, o *chiamate di sistema* (*system calls*) sono i servizi elementari offerti dal nucleo, come, per esempio, aprire un file, trasferire dati, generare un processo, e cosí via. Tutte le applicazioni sono costruite a partire dalle primitive, che costituiscono quindi l'interfaccia fra le applicazioni ed il nucleo.

Le primitive sono definite come funzioni del linguaggio C, e nella maggior parte dei casi restituiscono il valore -1 se si verifica un errore. È importante che si controlli sempre se c'è stato un errore, poiché gli errori nell'esecuzione delle primitive dipendono generalmente da condizioni, come la disponibilità di risorse a tempo di esecuzione, indipendenti dalla logica del programma e fuori dal controllo del programmatore. In caso di errore, la primitiva in cui si verifica scrive un codice di errore nella variabile esterna `errno`, dichiarata nel file `<errno.h>`. La funzione di libreria `perror()` legge tale variabile e stampa un messaggio predefinito, insieme ad un ulteriore messaggio definito dal programmatore. Purtroppo, per mancanza di spazio alcuni esempi saranno scritti senza i prescritti controlli sugli errori.

Delle numerose (da 60 a 150) chiamate di sistema, ne tratteremo pochissime, tra le piú fondamentali. La loro descrizione sarà per forza di cose limitata e in certi casi inesatta, per cui si raccomanda di confrontarla con i manuali di riferimento.

8.1 Alcune strutture dati

Il sistema operativo mantiene numerose strutture dati per gestire i processi ed i file. Alcune di queste sono globali ed inizializzate col sistema, altre sono associate ai singoli processi ed inizializzate con essi. Anche le strutture dati associate ai processi, comunque, possono essere modificate dal rispettivo processo solo quando questo esegue in fase nucleo, cioè durante l'esecuzione di primitive.

Fra le strutture globali, consideriamo le seguenti:

tabella dei processi associa a ciascun processo una struttura contenente varie informazioni, fra cui i numeri identificatori ed i puntatori alle aree di memoria usate.

tabella degli inode contiene una copia dell'inode di ogni file a cui faccia riferimento qualche processo; a questa copia vengono aggiunti altri campi di informazioni, fra cui un *contatore dei riferimenti* che viene aggiornato ogni volta che il file viene aperto o chiuso.

tabella dei file ogni volta che viene aperto un file, in questa tabella viene inserita una struttura contenente un riferimento all'inode del file nella tabella degli inode, e il *puntatore del file*, cioè l'indirizzo del byte da accedere alla prossima operazione di ingresso/uscita. Si noti che uno stesso file può essere aperto piú di una volta, e quindi apparire piú di una volta nella tabella.

Ogni processo Unix è formato da tre segmenti: *istruzioni* (detto anche *testo*), *dati utente*, e *dati di sistema*. L'insieme delle informazioni contenute in queste tre aree è il *contesto* del processo. Al contesto appartengono anche le informazioni relative al processo memorizzate nella tabella dei processi.

Fra le informazioni presenti nel contesto ci sono le seguenti:

- *identificatore del processo* (*PID*)
- *identificatore del processo padre*
- *tabella dei descrittori di file*
- *contatore di programma*

Ogni processo viene generato da un processo preesistente[1], detto *padre* (*parent*), mediante la primitiva `fork()`, che vedremo piú oltre. Fra le informazioni associate a ciascun processo c'è anche il PID del padre.

La tabella dei descrittori di file appartiene al segmento dei dati di sistema del processo e contiene dei puntatori alla tabella dei file. Il termine *descrittore di file* viene usato sia per uno di questi puntatori che per la sua posizione (da 0 a 19) nella tabella dei descrittori. Quando sia necessario, si userà il termine *numero di descrittore* per indicare la sua posizione nella tabella. In particolare, il parametro da passare alle chiamate di sistema è sempre un numero di descrittore.

Per convenzione, i processi usano i descrittori 0, 1 e 2 rispettivamente per l'ingresso standard, l'uscita standard e l'uscita standard di messaggi (*standard error*). Al programmatore conviene rispettare questa regola – osservata in particolare dalla shell – per ottenere una piú facile comunicazione fra i processi, ma per il nucleo non esistono descrittori speciali. Un descrittore è semplicemente un numero usato da un particolare processo per riferirsi ad un particolare file, o piú precisamente ad un'*istanza* di un file.

8.2 Gestione dei file

Le operazioni essenziali sui file sono la creazione, l'apertura, la lettura e la scrittura, e la chiusura. Altre operazioni riguardano la ricerca e l'aggiornamento di informazioni sui file, le modifiche dei modi di accesso, l'uso dei directory e dei file speciali, eccetera. Nel-

[1] Il processo 0, progenitore di tutti i processi, appare per magía all'avvio del sistema.

lo Unix, la creazione e l'apertura di un file possono essere compiute da un'unica primitiva.

Osserviamo che le primitive qui descritte sono distinte dalle *subroutine* o *funzioni di libreria* studiate nel linguaggio C. Queste ultime, come `fopen()` o `printf()`, servono a dare al programmatore un interfaccia piú comodo e ovviamente sono state scritte usando le primitive. Si ricordi inoltre che le funzioni della libreria standard di ingresso/uscita si riferiscono ai file mediante puntatori a strutture di tipo `FILE`, mentre le primitive usano dei descrittori di tipo `int`.

8.2.1 `open`

```
int open( char *path, int flags, int perms )
```

`path` path del file da aprire.

`flags` intero che specifica il modo in cui si vuole aprire il file. Si usano costanti simboliche predefinite o loro combinazioni mediante l'operatore di *OR* bit a bit ('`|`'):

`O_RDONLY`	lettura
`O_WRONLY`	scrittura
`O_RDWR`	lettura e scrittura
`O_CREAT`	creare se non esiste
	...

Queste costanti sono definite in `<fcntl.h>`.

`perms` diritti di accesso del file, rappresentati da una costante ottale di quattro cifre (usato solo con `O_CREAT`).

Restituisce un numero di descrittore, o -1 se c'è un errore. Per esempio, il seguente frammento apre un file di nome `dati` nel directory corrente, creandolo se necessario, in lettura e scrittura, con diritti di lettura e scrittura per tutti. In caso di errore si stampa un messaggio.

```
if( (fd = open( "dati", O_RDWR|O_CREAT, 0666 )) == -1 )
    perror( "open" );
```

Quando si crea un file, viene allocato un nuovo elemento nella tabella degli inode, e viene aggiornato il directory in cui il file è stato creato.

Quando un file (esistente prima della chiamata o creato da essa) viene aperto, se necessario si alloca il suo inode nella tabella degli inode, quindi viene allocato un nuovo elemento nella tabella dei file, contenente un puntatore all'inode, e poi viene allocato un nuovo descrittore di file nella tabella dei descrittori relativa al processo che ha richiesto l'apertura del file. La `open()` restituisce l'indice del descrittore in quest'ultima tabella.

Naturalmente, le operazioni di creazione ed apertura richiedono che il file e il directory coinvolti abbiano i necessari permessi per il processo.

8.2.2 `close`

```
int close( int fd )
```

Restituisce 0 o, in caso di errore, -1. L'effetto di questa primitiva è di liberare il descrittore del file nella tabella del processo chiamante, e di informare il nucleo che il processo non usa piú il file, decrementando il contatore dei riferimenti. Il nucleo toglie il file dalla tabella dei file quando nessun processo lo usa piú.

8.2.3 `write`

```
int write( int fd, char *buf, unsigned nbytes )
```

`fd`	numero di descrittore di file
`buf`	puntatore ai dati da scrivere
`nbytes`	numero di byte da scrivere

Scrive nel file di descrittore `fd` un numero di byte uguale ad `nbytes` posti in memoria a partire dall'indirizzo `buf`. La scrittura nel file inizia dalla posizione corrente del puntatore del file, che viene poi

incrementato del numero di byte trasferiti. Restituisce il numero di byte scritti o -1.

La `write()`, quando opera su file ordinari, non scrive direttamente sul disco, ma in un *buffer*, che è un'area dati appartenente al nucleo (da non confondere con l'area indirizzata dal parametro `buf`, che appartiene all'utente). In seguito, il meccanismo di gestione dei buffer copia i buffer su disco. La dimensione dei buffer è uguale a quella dei blocchi del disco, per avere trasferimenti più veloci.

Con l'uso dei buffer del nucleo, ogni operazione di ingresso/uscita richiede due trasferimenti: uno fra lo spazio dati dell'utente e lo spazio del nucleo, e uno fra lo spazio del nucleo e il disco. Anche i trasferimenti fra spazio utente e spazio del nucleo, sebbene siano da memoria a memoria, sono molto piú efficienti se eseguiti in blocchi di dimensione pari a quella dei buffer. Se, come accade di solito, l'applicazione richiede trasferimenti in blocchi piú piccoli, conviene usare le funzioni di libreria, che usano un ulteriore livello di buffer nello spazio utente.

8.2.4 `read`

```
int read( int fd, char *buf, unsigned nbytes )
```

`fd`	numero di descrittore di file
`buf`	puntatore ai dati da leggere
`nbytes`	numero di byte da leggere

Legge dal file di descrittore `fd` un numero di byte uguale ad `nbytes` e li copia in memoria a partire dall'indirizzo `buf`. La lettura dal file inizia dalla posizione corrente del puntatore del file, che viene poi incrementato del numero di byte trasferiti. Restituisce il numero di byte letti, 0 se si è raggiunta la fine del file, o -1.

Anche per questa primitiva valgono le considerazioni sull'uso dei buffer fatte a proposito della `write()`.

8.2.5 Altre primitive

Descriviamo piú sinteticamente altre chiamate di sistema relative ai file.

8.2.5.1 `creat`

```
int creat( char *path, int perms )
```

Crea un file vuoto e lo apre in scrittura. Se il file esiste già, lo tronca a lunghezza zero. Le funzioni di questa primitiva possono essere svolte dalla `open()`.

Restituisce il numero di descrittore o -1.

8.2.5.2 `lseek`

```
long lseek( int fd, long offset, int base )
```

Cambia il valore del puntatore al file di descrittore `fd`, permettendo accessi non sequenziali. Se `base` è 0, il puntatore viene posto uguale a `offset`; se `base` vale 1, al valore corrente del puntatore viene sommato `offset`; se, infine, `base` è 2, il puntatore viene posto uguale a `offset` piú la dimensione del file. Poiché il parametro `offset` può essere negativo, quest'ultima opzione permette di riferirsi ai dati partendo dalla fine del file.

La funzione restituisce il nuovo valore del puntatore o -1.

8.2.5.3 `mknod`

```
int mknod( char *path, int mode, int device )
```

Crea un directory o un file speciale. Il parametro `path` è il path del file da creare, e `mode` specifica il tipo di file oltre ai permessi di accesso. Questo parametro si esprime con l'OR bit a bit di costanti definite in `<sys/stat.h>`. Il terzo parametro specifica il numero di dispositivo quando si crea un file speciale di tipo device. Questo

numero è formato dalle due componenti major e minor, contenute in due byte del parametro.

La funzione restituisce 0 o -1, e può essere eseguita solo dal superuser.

8.2.5.4 `stat` ed `fstat`

```
int stat( int fd, struct stat *sbuf )
int fstat( char *path, struct stat *sbuf )
```

Scrive nella struttura di indirizzo `sbuf` le informazioni dell'inode individuato da un path (`path`) o da un descrittore (`fd`). Il tipo della struttura è dichiarato in `<sys/types.h>`.

Restituisce 0 o -1.

8.2.5.5 `ioctl`

```
int ioctl( int fd, int cmd, struct termio *tbuf )
```

Esamina o modifica vari parametri dei dispositivi, fra cui quelli che controllano terminali, come la velocità di trasmissione, il numero di bit per carattere, la parità, e molti altri. Il parametro `cmd` specifica l'azione da eseguire, usando costanti definite in `<sys/ioctl.h>`, e la struttura di indirizzo `tbuf` contiene i valori da leggere o modificare. Il tipo della struttura è dichiarato in `<sys/termio.h>`.

Restituisce 0 o -1.

Mentre le altre primitive sono logicamente indipendenti dal supporto fisico dei file, alla `ioctl()` è stato lasciato il lavoro "sporco" di interagire con la complessità e le particolarità dei vari dispositivi, per cui è una delle primitive la cui definizione varia piú largamente fra le diverse implementazioni.

8.3 Processi

Un processo, com'è noto, è un'istanza di esecuzione di un programma, cioè di un file eseguibile. In Unix, la creazione di un processo

avviene di solito in due tempi: prima si genera un nuovo processo che ha un nuovo identificatore ma è altrimenti una copia esatta del processo (processo *padre*) che lo ha creato, quindi il nuovo processo (*figlio*) carica in memoria il programma che deve eseguire. Per esempio, una shell interattiva è un processo che gestisce il dialogo con l'utente; per eseguire un comando genera un nuovo processo identico a se stessa, e questo processo "si trasforma" nel comando da eseguire. Fra la creazione e la trasformazione, il nuovo processo può eseguire altre operazioni, in particolare quelle necessarie a stabilire dei canali di comunicazione fra il processo padre ed il processo figlio.

8.3.1 `fork`

La chiamata `fork()` esegue il primo passo della creazione di un nuovo processo, cioè produce una copia del processo che la invoca. Si osservi che questa è la vera e propria *creazione* del processo figlio, in quanto produce un nuovo identificatore di processo:

```
int fork()
```

Restituisce al padre il numero di processo del figlio oppure -1 e restituisce 0 al figlio.

Il processo figlio è una copia quasi identica del padre, eccettuati gli identificatori di processo e di processo padre, i tempi di esecuzione, ed il valore restituito da `fork()`. Vengono copiati quasi tutto il segmento dati di sistema, tutto il segmento testo e tutto il segmento dati. Si noti che

- i contatori di programma del padre e del figlio hanno lo stesso valore al ritorno della `fork()`, e quindi puntano a due copie distinte della stessa istruzione.
- la tabella dei descrittori del figlio è la copia della tabella del padre. Quindi il figlio eredita i file aperti dal padre e li può modificare. Anche il puntatore del file è condiviso.

8.3.2 execlp

Il secondo passo nella creazione di un processo può essere eseguito da una chiamata qualsiasi di una famiglia di primitive, dette genericamente *exec*, che si distinguono nel modo di passare gli argomenti al nuovo programma e di interpretare il nome del file. La `execlp()` è probabilmente la piú semplice da usare.

```
int execlp(char *file, char *arg0, char *arg1, ...)
```

Il parametro `file` è il path di un file eseguibile, i parametri successivi sono il nome stesso dell'eseguibile ed eventualmente i suoi argomenti, l'ultimo dei quali deve essere `NULL`.

Restituisce -1 in caso di errore, altrimenti non restituisce niente perché *non ritorna*. Questa primitiva sostituisce i segmenti testo e dati utente di un processo col contenuto del file eseguibile. Questa operazione *non* crea un processo, ma trasforma un processo già esistente, che mantiene il proprio identificatore e quasi tutte le informazioni del segmento dei dati di sistema. In particolare, dopo l'esecuzione di `execlp()` il processo conserva la sua tabella dei descrittori di file.

Le altre primitive *exec* sono `execl()`, `execv()`, `execle()`, `execve()`, ed `execvp()`. È possibile, a seconda delle implementazioni, che solo alcune di esse siano primitive e le altre siano funzioni di libreria.

8.3.3 exit

```
void exit( int status )
```

Termina l'esecuzione del processo e passa al nucleo il byte meno significativo di `status`. Per convenzione, si passa il valore 0 per indicare una terminazione normale. Il padre del processo che ha eseguito `exit()` può esaminare il valore della parte bassa di `status` mediante la chiamata `wait()`.

8.3.4 wait

```
int wait( int *statusp )
```

Il processo che esegue la `wait()` viene sospeso finché non termina l'esecuzione di uno dei figli. Allora la funzione restituisce il PID del figlio, e scrive nella variabile di indirizzo `statusp` lo stato di terminazione. Se il byte meno significativo di questa variabile è 0, allora il byte piú significativo contiene il valore passato dal figlio mediante la `exit()`. Ricordiamo che tale valore si trova nel byte *meno* significativo del parametro della `exit()`. Se il byte basso della variabile di indirizzo `statusp` non è 0, l'altro byte contiene un codice che descrive la ragione per cui il figlio ha terminato.

Non è possibile aspettare un figlio in particolare: il primo dei figli a terminare fa sí che la `wait()` ritorni. Si possono fare piú chiamate `wait()`, e se non ci sono (piú) figli viene restituito immediatamente (senza sospendere il processo) il valore -1.

8.3.5 Esempio

In questo esempio, il processo figlio esegue il comando `echo`, mentre il processo padre termina. Se `fork()` restituisce -1 c'è stato un errore (forse la tabella dei file era piena?); se restituisce 0, è il figlio che sta eseguendo, e quindi viene chiamata `execlp()`; altrimenti, è il padre che esegue, e viene chiamata `exit()`. Si noti la chiamata `perror( exec )`: se il controllo arriva a quel punto, vuol dire che la `execlp()` non ha funzionato, altrimenti le istruzioni successive sarebbero sparite, cancellate dal testo del programma `echo`.

```
main()
{
    switch( fork() ) {
    case -1:
        perror( "fork" );
        exit( 1 );
    case 0:
```

```
            execlp( "echo", "echo", "ciao", NULL );
            perror( "exec" );
            exit( 1 );
        default:
            exit( 0 );
        }
    }
```

8.4 Comunicazione fra processi

Lo Unix dispone di numerosi meccanismi di *comunicazione interprocesso*, fra cui i *semafori*, i *messaggi*, le *code FIFO*, i *socket*, e la *memoria condivisa*. Anche i *segnali* sono una forma rudimentale di comunicazione fra processi. Qui accenneremo soltanto ai pipe, usati anche dalla shell per comporre comandi in pipeline, alle code FIFO, ed ai socket.

8.4.1 `pipe`

Un pipe è una struttura dati gestita dal nucleo come coda. Un processo scrive ad un'estremità della coda, e un altro legge dall'altra. Le due estremità della coda sono viste dai processi come due file, dove si può, rispettivamente, scrivere e leggere con le primitive `write()` e `read()`. Quando operano su un pipe, queste chiamate si sincronizzano in modo che i dati vengano trasferiti correttamente. Un pipe viene creato mediante la chiamata `pipe()`:

```
int pipe( int pfd[2] )
```

La funzione restituisce 0 o -1. Il parametro `pfd` è un array in cui la funzione scrive due numeri di descrittore che corrispondono all'estremità di lettura (`pfd[0]`) e di scrittura (`pfd[1]`) del pipe. Questi descrittori vengono passati rispettivamente a `read()` e `write()` quando bisogna leggere o scrivere. Si può notare che il

processo che crea un pipe ottiene i descrittori relativi sia all'estremità di lettura che a quella di scrittura. Normalmente ne viene usata solo una, e l'altra viene chiusa.

Il nucleo alloca un inode e due elementi della tabella dei file, oltre ai descrittori nella tabella dei descrittori, ma non crea un link in alcun directory, per cui il pipe è un file speciale anonimo, invisibile ai processi che non ne hanno i descrittori.

Due processi possono comunicare scrivendo e leggendo su un pipe se hanno i relativi descrittori, ma i descrittori possono essere riprodotti solo attraverso il meccanismo di `fork()`, quindi due processi possono comunicare attraverso un pipe solo se sono parenti. Per esempio, se un processo P deve trasmettere informazioni a un altro processo Q, si può seguire il seguente schema:

1. P crea il pipe;
2. P crea un figlio P' (`fork()`), che eredita una copia dei descrittori del pipe;
3. P' chiude l'estremità di scrittura del pipe;
4. P' si trasforma in Q (`execlp()`);
5. P chiude l'estremità di lettura e procede nell'esecuzione.

Si noti che Q riceve una copia dei descrittori, ma *ne ignora il numero di descrittore*, cioè la posizione nella tabella. Infatti sia il segmento testo che il segmento dati di P sono stati sostituiti dalla `execlp()` col testo e i dati di Q, per cui il vettore `pfd[]` – o qualsiasi altra variabile di P – è del tutto inaccessibile, essendo ormai sparito. Il numero di descrittore potrebbe essere passato da P' a Q fra gli argomenti di `execlp()`, come nel seguente esempio:

```
/* TESTO DI P E P' -- VERS. PROVVISORIA */
main()
{
    int pfd[2];
    char fdstr[10];

    pipe( pfd );
    if ( fork() == 0 ) {
```

```
        close( pfd[1] );         /* chiude scrittura */
        sprintf( fdstr, "%d", pfd[0] );
        execlp( "./Q", "Q", fdstr, NULL );
    }
    close( pfd[0] );             /* chiude lettura */
    write( pfd[1], "ciao", 5 );
}

/* TESTO DI Q -- VERS. PROVVISORIA */
main( int argc, char *argv[] )
{
    int fd;
    char s[100];

    fd = atoi( argv[1] );
    read( fd, s, sizeof(s) );
    printf( "%s\n", s );
}
```

Questo metodo *non viene usato* perché ha un grave inconveniente: il programma Q deve essere scritto in modo da ricevere le informazioni sui suoi flussi di ingresso e di uscita attraverso gli argomenti. Questo impedisce di scrivere programmi (come i filtri) che leggono dal descrittore 0 (ingresso standard) e scrivono sul descrittore 1 (uscita standard) lasciando al sistema operativo l'onere di collegare tali descrittori ai file o pipe richiesti di volta in volta.

Un altro metodo, che usa la primitiva `dup()`, permette di evitare questo inconveniente, anche se in un modo alquanto artificioso.

8.4.2 dup

```
int dup( int fd )
```

Crea una copia del descrittore il cui numero è `fd`, e restituisce il numero di descrittore della copia, oppure -1 in caso di errore.

Il numero restituito è il piú basso disponibile nella tabella, quindi se siamo sicuri che il numero 0 è disponibile, allora siamo sicuri

che `dup()` restitusce 0. Per assicurarsi che il numero 0 sia disponibile, basta chiudere il file corrispondente prima di chiamare `dup()`. Analogamente si può ottenere che `dup()` restituisca il numero 1. In questo modo si può comunicare mediante un pipe con un processo programmato per usare i descrittori 0 e 1.

Riscriviamo l'esempio precedente usando la `dup()`:

```
/* TESTO DI P E P' */
main()
{
    int pfd[2];

    pipe( pfd );
    if ( fork() == 0 ) {
        close( 0 );           /* libera desc. 0 */
        dup( pfd[0] );        /* desc. pfd[0] in desc. 0 */
        close( pfd[0] );      /* butta via desc. pfd[0] */
        close( pfd[1] );      /* chiude scrittura */
        execlp( "./Q", "Q", NULL );
    }
    close( pfd[0] );          /* chiude lettura */
    write( pfd[1], "ciao", 5 );
}

/* TESTO DI Q */
main()
{
    char s[100];

    read( 0, s, sizeof(s) );
    printf( "%s\n", s );
}
```

Si noti come questa versione sia molto piú modulare della precedente, in quanto il processo Q si limita a leggere dall'ingresso standard. Questa versione di Q può, per esempio, essere usata come un comando shell (simile a `cat`) senza che l'utente debba specificare il descrittore dell'ingresso. Inoltre, se volessimo sostituire

Q con un altro programma, magari scritto indipendentemente come il comando `cat` stesso, basta sostituirne il path ed il nome nei parametri della `execlp()`, senza modificare il codice che gestisce i descrittori di file.

La shell usa `dup()` per realizzare le pipeline e le redirezioni.

8.4.3 Code FIFO

Le *code FIFO* sono dei pipe che hanno un nome come se fossero dei file ordinari, e sono chiamate anche *pipe con nome* (*named*); quindi i pipe visti prima sono detti anche *pipe anonimi* (*unnamed*). Mentre un pipe anonimo può essere usato solo da processi generati attraverso `fork()` da un antenato comune che ha creato il pipe, una coda FIFO può essere usata, al pari di un file ordinario, da qualsiasi processo. Questo permette una comunicazione – generalmente secondo il modello a scambio di messaggi – fra processi indipendenti.

Per creare una coda FIFO, si usa la chiamata `mknod()` con l'opzione `S_IFIFO`. Il seguente esempio mostra come creare una coda FIFO nel directory `/tmp`, col nome `FIFO1234`, e permessi di lettura e scrittura per tutti:

```
mknod( "/tmp/FIFO1234", S_IFIFO | 0666, 0 );
```

I processi che usano la coda, la aprono in lettura o scrittura con una `open()`, e quindi usano le primitive di lettura e scrittura già viste.

8.4.4 Socket

I *socket* sono un meccanismo per la comunicazione fra processi residenti su nodi di una rete di calcolatori. I concetti relativi alle reti di calcolatori saranno esposti in altri corsi, e qui si dà soltanto una descrizione sommaria dell'uso dei socket dal punto di vista del programmatore di applicazioni distribuite. In particolare, si fa

riferimento ad un modello di comunicazione con *connessione* e *consegna affidabile*. In questo modello, due processi possono dialogare *come se* esistesse un collegamento diretto su cui scambiare messaggi di dimensione arbitraria e in modo affidabile, senza cioè dover provvedere al controllo e alla correzione degli errori di trasmissione.

Un'applicazione *distribuita*, cioè formata da processi cooperanti eseguiti su diversi nodi, può essere strutturata secondo il modello *client/server*, in cui un processo *server* offre un servizio (per esempio, l'accesso a dei file sul nodo in cui risiede), e dei processi *clienti* comunicano col server per ottenerne i servizi. Un socket rappresenta un'estremità di un canale di collegamento, identificata da un *indirizzo*. Si può fare un'analogia col telefono: il socket è come una presa telefonica, e l'indirizzo è come il numero di telefono. Per usare i socket col modello client/server, il server deve creare un socket, che chiameremo *di attesa* ed associargli un indirizzo (che per il momento immaginiamo come un semplice numero identificatore). Quindi il server si mette in attesa di essere chiamato sul socket. Un cliente deve creare il proprio socket, se necessario associargli un indirizzo, e quindi chiamare il server all'indirizzo del server di attesa. Naturalmente, il cliente deve conoscere l'indirizzo (il "numero di telefono") del server. Quando il server riceve la chiamata, la accetta creando un nuovo socket, che chiameremo *di lavoro*. Questo ha lo stesso indirizzo del socket di attesa ed è dedicato al colloquio col cliente, mentre il socket di attesa torna libero e può ricevere chiamate da altri clienti.

Il server ed i clienti "vedono" soltanto i rispettivi socket, che nascondono tutti i meccanismi di livello inferiore che provvedono all'effettiva trasmissione dei dati. Un processo si riferisce ad un socket mediante un *descrittore di socket* che è equivalente ad un descrittore di file, per cui si può usare un socket con le primitive `read()` e `write()`, anche se esistono primitive specifiche per i socket. L'uso di primitive proprie dei socket è invece richiesto per la creazione, l'associazione all'indirizzo, la chiamata e l'accettazione. Comunque, una volta stabilito il collegamento dopo l'accettazione, il descrittore di socket può essere passato a un processo figlio, che

poi può eseguire un programma scritto senza prevederne l'uso in un'applicazione distribuita.

8.4.4.1 `socket`

```
int socket(int af, int type, int protocol)
```

Crea un socket e ne restituisce il numero di descrittore. Il parametro `af` è il formato dell'indirizzo usato dalla famiglia di protocolli (per esempio, la costante `AF_INET` rappresenta il formato usato dalla famiglia di protocolli TCP/IP), `type` rappresenta il tipo di comunicazione (per esempio, `SOCK_STREAM` rappresenta il servizio di consegna affidabile), e `protocol` specifica il particolare *protocollo*, cioè l'insieme di convenzioni usate dalla rete per lo scambio di informazioni, scelto nell'àmbito della famiglia specificata da `af`.

Viene usata sia dal server che dai clienti.

8.4.4.2 `bind`

```
int bind(int sd, struct sockaddr *addr, int len)
```

Associa al socket di descrittore `sd` l'indirizzo codificato in una struttura puntata da `addr` e di dimensione `len`. Il formato dell'indirizzo dipende dal protocollo usato: per esempio, i protocolli della famiglia TCP/IP usano indirizzi formati da un intero di 32 bit che specifica il nodo della rete (*host*) ed uno di 16 bit che identifica il servizio (piú precisamente, la *porta di protocollo*) all'interno del nodo.

Viene usata dal server, ed opzionalmente dai clienti. Se un cliente non lega un indirizzo alla propria porta, l'indirizzo viene assegnato dal software di rete.

8.4.4.3 `listen`

```
int listen(int sd, int backlog)
```

Crea una coda in cui vengono inserite le richieste di connessione col socket di descrittore `sd`, provenienti dai clienti, lunga `backlog` elementi.

Viene usata dal server.

8.4.4.4 `connect`

```
int connect(int sd, struct sockaddr *addr, int len)
```

Richiede la connessione fra il socket locale di descrittore `sd` ed il socket remoto il cui indirizzo è codificato nella struttura puntata da `addr`.

Viene usata dai clienti.

8.4.4.5 `accept`

```
int accept(int sd, struct sockaddr *addr, int len)
```

Sospende il server finché non arriva una richiesta di connessione. Quindi viene creato un nuovo socket (il socket di lavoro), il server riprende l'esecuzione, e la `accept()` restituisce il descrittore del nuovo socket. Inoltre, scrive l'indirizzo del cliente nella struttura puntata da `addr`.

Viene usata dal server.

8.4.4.6 `send` e `recv`

```
int send(int sd, char *buf, int len, int flags)
int recv(int sd, char *buf, int len, int flags)
```

Trasmette/riceve un messaggio di lunghezza `len` da/in un buffer puntato da `buf`, usando il socket di descrittore `sd`. Il parametro `flags` specifica delle operazioni speciali: si possono mandare messaggi *out of band*, che vengono ricevuti immediatamente – sorpassando altri messaggi eventualmente accodati sul socket di destinazione – e causano l'invio di un segnale al ricevente, e si possono leggere messaggi senza estrarli dalla coda. Se queste operazioni

speciali non servono, si possono usare `write()` e `read()` al posto di `send()` e `recv()` o di altre primitive analoghe.

Vengono usate sia dal server che dai clienti.

8.5 Segnali

Sebbene i segnali si possano usare come uno strumento molto limitato per la comunicazione fra processi, il loro vero scopo è quello di gestire situazioni eccezionali, come errori a tempo di esecuzione o richieste di interruzione da parte dell'utente. Quando si verifica una di queste situazioni, il nucleo manda un *segnale*, codificato da un numero intero, al processo. Se questo ha previsto un'azione per rispondere a tale situazione, l'azione viene eseguita, altrimenti il nucleo esegue l'azione di default, che di solito consiste nel terminare il programma. Le azioni previste dai processi generalmente si limitano a chiudere dei file, cancellare file temporanei, e scrivere messaggi all'utente prima di terminare l'esecuzione. Si può anche ignorare il segnale; questo accade, per esempio, nell'esecuzione della shell, che non deve terminare se l'utente preme per sbaglio i tasti di interruzione mentre scrive un comando. Quando un processo predispone un'azione diversa da quella di default, si dice che *intercetta* (*catches*) il segnale.

Il numero di segnali definiti dal sistema dipende dall'implementazione. Ricordiamo i seguenti, definiti in `<signal.h>`:

`SIGHUP` (*hangup*) sconnessione del terminale;
`SIGINT` (*interrupt*) interruzione dall'utente ('`^C`');
`SIGQUIT` (*quit*) interruziene con core dump ('`^\`');
`SIGKILL` (*kill*) terminazione obbligatoria del processo ricevente;
`SIGSEGV` (*segmentation violation*) di solito, errore di indirizzamento;
`SIGTERM` (*software termination*) terminazione del processo ricevente.

Il segnale `SIGKILL` non può essere né ignorato né intercettato.

8.5.1 signal

La primitiva `signal()` serve a specificare al nucleo quale azione deve essere eseguita:

```
void (*signal( int sig, void (*handler)(int) ))(int)
```

Questa celebre dichiarazione vuol dire che `signal()` è una funzione di due argomenti (`sig` e `handler`) che restituisce un puntatore ad una funzione. Il parametro `sig` è un intero che rappresenta il tipo di segnale da intercettare; il parametro `handler` è l'indirizzo della funzione da eseguire per gestire il segnale. Il numero del segnale viene passato alla funzione di indirizzo `handler`. Il valore restituito da `signal()` è l'indirizzo della funzione per la gestione del segnale preesistente alla chiamata di `signal()`; salvando questo indirizzo, è possibile in seguito ripristinare il precedente trattamento del segnale. Invece dell'indirizzo di una funzione, il secondo parametro può essere la costante `SIG_DFL`, per chiedere l'azione di default, o la costante `SIG_IGN`, per ignorare il segnale. Ricordiamo che `SIGKILL` non può essere né ignorato né intercettato.

Nelle versioni di Unix sviluppate a Berkley, la `signal()` è una funzione di libreria basata sulla primitiva `sigvec()`.

8.5.2 kill

Questa primitiva invia un segnale `sig` al processo di identificatore `pid` (questo parametro può assumere valori nulli o negativi per mandare segnali a gruppi di processi).

```
int kill( int pid, int sig )
```

L'uso piú comune di questa primitiva è di far terminare i processi. Si può usare anche per collaudare programmi simulando situazioni eccezionali mediante i corrispondenti segnali.

Restituisce 0 o -1.

Bibliografia

[1] M.J. Bach. *The design of the Unix operating system.* Prentice-Hall, 1986.

[2] D.E. Comer and D.L. Stevens. *Internetworking with TCP/IP.* Prentice-Hall, 1991.

[3] S.J. Leffler, M.K. McKusick, and J.S. Quarterman. *The design and inplementation of the 4.3BSD Unix operating system.* Addison–Wesley, 1990.

[4] J.R. Levine, T. Mason, and D. Brown. *lex & yacc.* O'Reilly & Associates, 1992.

[5] A. Oram and S. Talbott. *Managing Programs with make.* O'Reilly & Associates, 1991.

[6] M.J. Rochkind. *Advaced UNIX programming.* Prentice-Hall, 1985.