Sistemi Operativi[1]

Mattia Monga

Dip. di Informatica
Università degli Studi di Milano, Italia
mattia.monga@unimi.it

a.a. 2019/20

Lezione XIV: Unix power tools

File

File

Una sequenza di byte che esistono indipendentemente dall'esecuzione dei programmi (e quindi sono persistenti rispetto all'attivazione dei processi)

Sono identificati da nomi (link nel gergo di Unix) organizzati gerarchicamente in un *file system*.
Come vedremo sono un'astrazione fondamentale nel mondo UNIX: terminale, dispositivi, ecc. sono trattati secondo questa politica.

POSIX Syscall (file mgt)

fd = creat(name, mode)	Obsolete way to create a new file
fd = mknod(name, mode, addr)	Create a regular, special, or directory i-node
fd = open(file, how, ...)	Open a file for reading, writing or both
s = close(fd)	Close an open file
n = read(fd, buffer, nbytes)	Read data from a file into a buffer
n = write(fd, buffer, nbytes)	Write data from a buffer into a file
pos = lseek(fd, offset, whence)	Move the file pointer
s = stat(name, &buf)	Get a file's status information
s = fstat(fd, &buf)	Get a file's status information
fd = dup(fd)	Allocate a new file descriptor for an open file
s = pipe(&fd[0])	Create a pipe
s = ioctl(fd, request, argp)	Perform special operations on a file
s = access(name, amode)	Check a file's accessibility
s = rename(old, new)	Give a file a new name
s = fcntl(fd, cmd, ...)	File locking and other operations

POSIX Syscall (file mgt cont.)

s = mkdir(name, mode)	Create a new directory
s = rmdir(name)	Remove an empty directory
s = link(name1, name2)	Create a new entry, name2, pointing to name1
s = unlink(name)	Remove a directory entry
s = mount(special, name, flag)	Mount a file system
s = umount(special)	Unmount a file system
s = sync()	Flush all cached blocks to the disk
s = chdir(dirname)	Change the working directory
s = chroot(dirname)	Change the root directory

Per fare esperimenti con i file descriptor può essere utile una funzione come la seguente

```
7  #include <sys/types.h>
8  #include <sys/stat.h>
9  #include <fcntl.h>
10 #include <unistd.h>
11 #include <limits.h>
12
13 void lsofd(const char* name){
14         int i;
15         for (i=0; i<_POSIX_OPEN_MAX; i++){
16                 struct stat buf;
17                 if (fstat(i, &buf) == 0){
18                     printf("%s fd:%d i-node: %d\n", name,
                       ↪  i, (int)buf.st_ino);
19                 }
20         }
21 }
```


Esercizio

Creare un file `provaxxx.dat` in un programma che, dopo la creazione, genera un nuovo processo con una `fork`. Scrivere nel file dal processo genitore e figlio, controllando lo stato del file prima e dopo ogni operazione.
Suggerimenti:

- `open("provaxxx.dat", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);`
- `lseek(fd, 0, SEEK_CUR);` per conoscere la posizione corrente del "cursore" del file, cioè il punto in cui verrà scritto (o letto) il prossimo carattere

File


```
23 int main(){
24         pid_t pid;
25         int f,  off;
26         char    string[] = "Hello, world!\n";
27
28         lsofd("padre (senza figli)");
29         printf("padre (senza figli) open *\n");
30         f = open("provaxxx.dat", O_CREAT|O_WRONLY|O_TRUNC, S_IRWXU);
31         if (f == -1){
32                 perror("open");
33                 exit(1);
34         }
35         lsofd("padre (senza figli)");
36         if (write(f, string, (strlen(string))) != (strlen(string)) ){
37                 perror("write");
38                 exit(1);
39         }
40
41         off = lseek(f, 0, SEEK_CUR);
42         printf("padre (senza figli) seek: %d\n", off);
43
44         printf("padre (senza figli) fork *\n");
45         if ( (pid = fork()) < 0){
46                 perror("fork");
47                 exit(1);
48         }
```

File (cont.)

```
49          if (pid > 0){
50                  lsofd("padre");
51                  printf("padre write & close *\n");
52                  off = lseek(f, 0, SEEK_CUR);
53                  printf("padre seek prima: %d\n", off);
54                  if (write(f, string, (strlen(string))) != (strlen(string)) ){
55                          perror("write");
56                          exit(1);
57                  }
58                  lsofd("padre");
59                  off = lseek(f, 0, SEEK_CUR);
60                  printf("padre seek dopo: %d\n", off);
61                  close(f);
62                  exit(0);
63          }
64          else {
65                  lsofd("figlio");
66                  printf("figlio write & close *\n");
67                  off = lseek(f, 0, SEEK_CUR);
68                  printf("figlio seek prima: %d\n", off);
69                  if (write(f, string, (strlen(string))) != (strlen(string)) ){
70                          perror("write");
71                          exit(1);
72                  }
73                  lsofd("figlio");
74                  off = lseek(f, 0, SEEK_CUR);
75                  printf("figlio seek dopo: %d\n", off);
76                  close(f);
77                  exit(0);
78          }
79  }
80
```


Input e Output

In generale il paradigma UNIX permette alle applicazioni di fare I/O tramite:

Input

- Parametri (`argv`)
- Variabili *d'ambiente* (`envp`, `getenv`)
- File (`open`, `read`, `write`, `close`)
  - Terminale (tty)
  - Device (p.es. il mouse potrebbe essere `/dev/mouse`)
  - Rete (socket)

Output

- Valore di ritorno (`return`)
- Variabili *d'ambiente* (`setenv`)
- File (`open`, `read`, `write`, `close`)
  - Terminale (tty)
  - Device (p.es. lo schermo in modalità grafica potrebbe essere `/dev/fb`)
  - Rete (socket)

Redirezioni

A ogni processo sono sempre associati tre file (già aperti)

- Standard input (Terminale, tastiera)
- Standard output (Terminale, video)
- Standard error (Terminale, video, usato per le segnalazione d'errore)

Possono essere *rediretti*

- `sort < lista` Lo stdin è il file `lista`
- `ls > lista` Lo stdout è il file `lista` (anche: `ls 1> lista`)
- `ls piripacchio 2> lista` Lo stderr è il file `lista`
- `( echo ciao & date ; ls piripacchio ) 2> errori 1>output`

Pipe

La pipe è un canale, analogo ad un file, bufferizzato in cui un processo scrive e un altro legge. Con la shell è possibile collegare due processi tramite una pipe anonima.
Lo stdout del primo diventa lo stdin del secondo
`ls | sort`
`ls -lR / | sort | more`
funzionalmente equivalente a
`ls -lR >tmp1; sort <tmp1 >tmp2; more <tmp2; rm tmp1 tmp2`
Molti programmi copiano lo `stdin` su `stdout` dopo averlo elaborato: sono detti filtri.

Pipe

```
ls | sort
```

```
int main(void){
int     fd[2], nbytes;    pid_t   childpid;
char    string[] = "Hello, world!\n";
char    readbuffer[80];

pipe(fd);
if(fork() == 0){
        /* Child process closes up input side of pipe */
        close(fd[0]);
        write(fd[1], string, (strlen(string)+1));
        exit(0);
} else {
        /* Parent process closes up output side of pipe */
        close(fd[1]);
        nbytes = read(fd[0], readbuffer, sizeof(readbuffer));
        printf("Received string: %s", readbuffer);
}
return(0);
}
```


Pipe (cont.)

```
if(fork() == 0)
{
  /* Close up standard input of the child */
  close(0);

  /* Duplicate the input side of pipe to stdin */
  dup(fd[0]);
  execlp("sort", "sort", NULL);
}
```


Command substitution

Con una pipe è possibile "collegare" lo `stdout` di un programma con lo `stdin` di un altro.
Per usare l'output di un programma sulla riga di comando (cioè come `argv`) di un altro programma, occorre usare la command substitution

```
ls -l $(which sort)
```


Esercizi

1. Scrivere una *pipeline* di comandi che identifichi il solo processo con il PPID piú alto (`ps, sort, tail`)
2. Ottenere il numero totale dei file contenuti nelle directory `/usr/bin` e `/var` (`ls, wc, expr`)
3. Si immagini di avere un file contenente il sorgente di un programma scritto in un linguaggio di programmazione in cui i commenti occupino intere righe che iniziano con il carattere `#`. Scrivere una serie di comandi per ottenere il programma senza commenti. (`grep`)
4. Ottenere la somma delle occupazioni dei file delle directory `/usr/bin` e `/var` (`du, cut`)

Alcuni programmi d'uso frequente

Prog. (sez. man)	Descrizione
`ls (1)`	list directory contents
`echo (1)`	display a line of text
`touch (1)`	change file timestamps
`sleep (1)`	delay for a specified amount of time
`rm (1)`	remove files or directories
`cat (1)`	concatenate files and print on the standard output
`man (1)`	an interface to the on-line reference manuals
`test (1)`	check file types and compare values
`sort (1)`	sort lines of text files
`date (1)`	print or set the system date and time
`less (1)`	file perusal filter for crt viewing
`which (1)`	locate a command
`ps (1)`	report a snapshot of the current processes.
`tail (1)`	output the last part of files
`wc (1)`	print the number of newlines, words, and bytes in files
`test (1)`	check file types and compare values
`grep (1)`	print lines matching a pattern
`cut (1)`	remove sections from each line of files
`du (1)`	print disk usage

Link

- "A Brief Introduction to Unix (With Emphasis on the Unix Philosophy)", Corey Satten `http://staff.washington.edu/corey/unix-intro.pdf`
- `http://en.wikipedia.org/wiki/Unix_philosophy`
- "The UNIX Time-Sharing System", Ritchie; Thompson `http://www.cs.berkeley.edu/~brewer/cs262/unix.pdf`

Link

- `http://www.gnu.org/software/coreutils/manual/coreutils.html`

Shell e file system

- Ogni processo (compresa la shell stessa) ha associata una *directory di lavoro* (working directory), che può essere cambiata col comando (interno alla shell) `cd`
- I programmi fondamentali per operare sul file system

ls (1)	list directory contents
cp (1)	copy files and directories
rm (1)	remove files or directories
mv (1)	move (rename) files
mkdir (1)	make directories
rmdir (1)	remove empty directories
df (1)	report file system disk space usage
du (1)	estimate file space usage
pwd (1)	print name of current/working directory

Permessi

A ogni file vengono associati dei *permessi*, che definiscono le azioni permesse sui dati del file

- **Read:** leggere il contenuto del file o directory
- **Write:** scrivere (cambiare) il file o directory
- **eXecute** eseguire le istruzioni contenute nel file o accedere alla directory

R	W	X	
1	1	0	6
1	0	1	5
1	0	0	4
1	1	1	7

I permessi possono essere diversi per 3 categorie di utenti del sistema:

- **User:** il "proprietario" del file
- **Group:** gli appartenenti al gruppo proprietario
- **All:** tutti gli altri

Agire sui permessi

- Cambiare il proprietario
  - `chown utente[:gruppo] file`
- Cambiare il gruppo
  - `chgrp gruppo file`
- Cambiare i permessi
  - `chmod 755 file`
  - `chmod +x file`
  - `chmod a=rw file`
  - `chmod g-x file`
- (per creare un utente: `adduser`)

Il bit SUID

Il proprietario di un processo in esecuzione è normalmente *diverso* dal proprietario del file contente un programma (e diverso ad ogni esecuzione)

- effective UID bit: il processo assume come proprietario il proprietario del file del programma
- SUID `root`
- `chmod 4555 file`
- `chmod u+s file`

find

Per selezionare file con determinate caratteristiche si usa `find`
`find percorso predicato`
Seleziona, nel sottoalbero definito dal percorso, tutti i file per cui il predicato è vero
Spesso usato insieme a a `xargs`
`find percorso predicato| xargs comando`
funzionalmente equivalente a
`comando $(find percorso predicato)`
ma evita i problemi di lunghezza della riga di comando perché xargs si preoccupa di "spezzarla" opportunamente.

Due espressioni idiomatiche

Spesso si vuole fare un'operazione per ogni file trovato con find. L'espressione piú naturale sarebbe:

```
for i in $(find percorso predicato); do
  comando $i
done
```

Questa forma presenta due problemi: può eccedere la misura della linea di comando e non funziona correttamente se i nomi dei file contengono *spazi*

Due espressioni idiomatiche

Un'alternativa è

```
find percorso predicato -print0 | xargs -0 -n 1
```

In questo modo (`-print0`) i file trovati sono separati dal carattere 0 anziché spazi e `xargs` è capace di adattarsi a questa forma.
Un'alternativa piú generale che mostra la potenza del linguaggio di shell che non distingue fra comandi e costrutti di controllo di flusso (sono tutti "comandi" utilizzabili in una pipeline)

```
find percorso predicato | while read x; do
  comando $x
done
```

`read x` legge una stringa e la assegna alla variabile *x*.

Esercizi

1. Trovare il file piú "grosso" in un certo ramo
2. Copiare alcuni file (ad es. il cui nome segue un certo pattern) di un ramo in un altro mantenendo la gerarchia delle directory
3. Calcolare lo spazio occupato dai file di proprietà di un certo utente
4. Scrivere un comando che conta quanti file ci sono in un determinato ramo del filesystem

Archivi

Un archivio *archive* è un file di file, cioè un file che contiene i byte di diversi altri file e i relativi *metadati*. (Cfr. con una directory, che è un file speciale, che sostanzialmente contiene solo l'elenco dei file)

- `ar` L'archiviatore classico, generalmente utilizzato per le librerie (provare `ar t /usr/lib/i86/libc.a`
- `tar` Tape archive, standard POSIX
  `tar cvf archivio.tar lista_files`

Gli archivi possono essere compressi con `compress` o, piú comunemente, con `gzip` o `bzip2`
I file `.zip` sono archivi compressi.

Altre utility

Altre utility "standard" di cui è bene conoscere almeno l'esistenza

Prog. (sez. man)	Descrizione
`uniq (1)`	report or omit repeated lines
`cut (1)`	remove sections from each line of files
`tr (1)`	translate or delete characters
`dd (1)`	convert and copy a file
`tee (1)`	read from standard input and write to standard output ...
`sed (1)`	stream editor for filtering and transforming text
`seq (1)`	print a sequence of numbers

Inoltre è molto utile conoscere le espressioni regolari (`man 7 re_format`), usate da `grep`, `sed`, ecc.

Ulteriori utility

Altre utility "standard" di cui è bene conoscere almeno l'esistenza

Prog. (sez. man)	Descrizione
`basename (1)`	strip directory and suffix from filenames
`dirname (1)`	strip non-directory suffix from file name
`uniq (1)`	report or omit repeated lines
`cut (1)`	remove sections from each line of files
`tr (1)`	translate or delete characters
`dd (1)`	convert and copy a file
`stat (1)`	display file or file system status

`cd` invece non è un programma, ma un comando interno della shell (che differenza fa?)

Esercizi

1. Creare un archivio `tar.gz` contenente tutti i file la cui dimensione è minore di 50KB
2. Rinominare un certo numero di file: per esempio tutti i file `.png` in `.jpg`
3. Creare un file da 10MB costituito da caratteri casuali (usando `/dev/random`) e verificare se contiene la parola JOS
4. Trovare l'utente che ha il maggior numero di file nel sistema
5. Trovare i 3 utenti che, sommando la dimensione dei loro file, occupano piú spazio nel sistema.