File Descriptors

Jose L. Muñoz, Oscar Esparza, Juanjo Alins, Jorge Mata
Telematics Engineering
Universitat Politècnica de Catalunya (UPC)

Outline

What is a File Descriptor?

- An fd is an integer that is used as index of a kernel-resident data structure containing the details of all **open files**.
- Each process has its own “file descriptor table“.
- The same file can have different file descriptors:
  - We can open a file for reading and get one fd.
  - We can open the same file for writing and get another fd.
  - One of the main elements contained in the file descriptor table is the file pointer, which indicates the current position (for r/w) on a file.

Standard fds I

- There are 3 standard file descriptors which presumably every process should have opened.
- When a process is created using a shell (like Bash), it inherits the 3 standard file descriptors from this shell.
- The `lsof` command shows us the “list of open files”:

```
$ ps
 PID TTY          TIME CMD
7988 pts/3    00:00:00 bash
8839 pts/3    00:00:00 ps
```

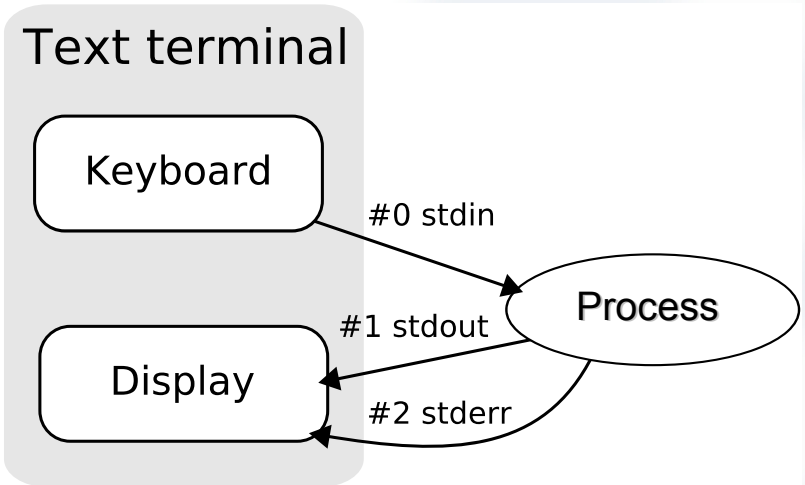
```
$ lsof -a -p 7988 -d0-10
COMMAND  PID     USER    FD TYPE DEVICE NAME
bash    7988 telematics 0r CHR  136,3  /dev/pts/3
bash    7988 telematics 1w CHR  136,3  /dev/pts/3
bash    7988 telematics 2w CHR  136,3  /dev/pts/3
```

- The previous command shows the file descriptors active (up to fd=10) for the process with PID 7988.

Standard fds II

- fd=0 is opened for reading (0r) and fd=1,2 are opened for writing (1w and 2w).

df (integer value)	**Name**
0	Standard Input (STDIN)
1	Standard Output (STDOUT)
2	Standard Error (STDERR)



Outline

Redirecting Output I

- The “output redirection” allows to send STDOUT/STDERR to files different from default ones.
- Examples:

```
$ echo Hello, how are you?
Hello, how are you?
```

```
$ echo Hello, how are you? > file.txt
$ cat file.txt
Hello, how are you?
```

```
$ echo Are you ok? >> file.txt
$ cat file.txt
Hello, how are you?
Are you ok?
```

- You can also redirect the standard error (*fd*=2).

Redirecting Output II

- Example:

```
$ ls -qw
$ ls -qw 2> error.txt
```

- In general:
  - **N**>**file**. It is used to redirect fd=N to a file. If the file exists, is deleted and overwritten. In case file does not exist, it is created.
  - **N**>>**file**. Similar to the first case but opens file in mode append.
  - & >**file**. Redirects STDOUT and STDERR to file.

Outline

Redirecting Input I

- Input redirection allows you to specify a file for reading standard input (*fd=0*).
- The format for input redirection is $<$ *file*.
- Example:

```
1 #!/bin/bash
2 # Our third script, using read for fun
3 echo Please, type a sentence and hit ENTER
4 read TEXT
5 echo You typed: $TEXT
```

- To redirect the input:

```
$ echo hello world > file.txt
$ ./thirdscript.sh < file.txt
Please, type a sentence and hit ENTER
You typed: hello world
$
```

Redirecting Input II

- Let's observe redirections with `lsof`:

```
1 #!/bin/bash
2 lsof -a -p $$ -d0-10
3 echo Now, please type a sentence and hit ENTER
4 read TEXT
5 echo You typed: $TEXT
```

```
$ ./thirdscript.sh < file
COMMAND PID USER  FD  TYPE DEVICE NAME
script  7728 user  0r  REG   8,6  /home/user/file
script  7728 user  1u  CHR 136,3  /dev/pts/3
script  7728 user  2u  CHR 136,3  /dev/pts/3
Now, please type a sentence and hit ENTER
You typed: hola
```

- The variable $ contains the PID of the script in execution.

Outline

Unnamed pipes I

- Unix based operating systems like Linux offer a unique approach to join two commands on the terminal.
- You can take the output of the first command and use it as input of the second command, this is the concept of pipe or "|".

```
$ ls | grep x
```

- This type of pipe is called "unnamed pipe" because the pipe exists only inside the kernel and cannot be accessed by processes that created it, in this case, the bash shell.
- All Unix-like systems include a variety of commands to manipulate text outputs.
- We have already seen some of these commands: `head`, `tail`, `grep` and `cut`.

Unnamed pipes II

- We also have other commands for text pipelines like:
  - `uniq` which displays or removes repeating lines.
  - `sort` which lists the contents of the file ordered alphabetically or numerically.
  - `wc` which counts lines, words and characters.
  - `find` which searches for files.
- The following example shows a compound command with several pipes:

```
$ cat *.txt | sort | uniq > result.txt
```

- `tee` reads STDIN and writes to a file and also to STDOUT.

```
$ ls | tee output.txt | sort -r
```

Process Substitution I

- Commands enclosed in parenthesis are run in a "subshell" (cloned shell).
- An application is to redirect the output of several commands:

```
$ (ps ; ls) >commands.out
```

- Process substitution occurs when you put a "<" or ">" in front of the left parenthesis:

```
$ cat <(ls -l)
```

- In the previous example `cat` has a valid file name to read from.
- Similarly, giving ">(commands)" results in bash naming a temporary pipe, which the commands inside the parenthesis read for input:

Process Substitution II

```
ls|tee >(grep foo|wc>foo.txt)|grep baz|wc>baz.txt
```

Dash

- The dash “-” in some commands is useful to indicate that we are going to use stdin or stdout instead of a regular file.
- Example:

```
$ grep linux doc1.txt | cat doc2.txt -
```

- Dash depends on the context (has other uses).
- To give an example, the command `cd -` means go to the previous directory visited (nothing to do with redirection).

Named Pipes

- Another sort of pipe is the “named pipe”, also called FIFO (First In First Out).
- First bytes entered are the first ones to be removed.
- You can create it:

```
$ mkfifo fifo1
```

- To use it:

```
t1$ echo hello how are you? > pipe1
```

```
t2$ cat < pipe
```

- As you will observe, the output of the command run on the first pseudo-terminal shows up on the second pseudo-terminal.

Outline

Files in Bash I

- In bash, we can manage a total of 9 file descriptors.
- 0 is the standard input, 1 is the standard output, 2 is the standard error and there are six additional fds that take values from 3 to 9.
- Useful for this example:

```
#!/bin/bash
LOGFILE=/var/log/script.log
comand1 >$LOGFILE
comand2 >$LOGFILE
...
```

Files in Bash II

Syntax	Meaning
`exec fd> file`	open file for writing and assign `fd`.
`exec fd>>file`	open file for appending and assign `fd`.
`exec fd< file`	open file for reading and assign `fd`.
`exec fd<> file`	open file for reading/writing and assign `fd`.
`exec fd1>&fd2`	open `fd1` for writing. From this moment `fd1` and `fd2` are the same.
`exec fd1<&fd2`	open `fd1` for reading. From this moment `fd1` and `fd2` are the same.
`exec fd>&-`	close `fd`.
`command >&fd`	write stdout to `fd`.
`command 2>&fd`	write stderr to `fd`.
`command <&fd`	read stdin from `fd`.

Files in Bash III

- Let us illustrate the use of `exec` for reading files by an example:

```
#!/bin/bash
LOGFILE=/var/log/script.log
exec 1>$LOGFILE
cmd1
cmd2
...
```

```
$ exec 3>&1            # create a new fd that points to
                       # the same place as stdout (fd=1)
$ exec 1>script.log    # fd=1 is now redirected to the log file
$ cmd1                 # the output of cmd1 will go into the log file.
$ cmd2                 # the output of cmd2 will go into the log file.
$ exec 1>&3            # now fd=1 points the same file as fd=3
$ cat script.log       # Now we see the output in our terminal
$ lsof -a -p $$ -d0-3 # we have four fd open for this bash
```

Files in Bash IV

- In another example let us assign the file file.log to fd=3 and use this file descriptor number for writing, finally we close the fd.

```
$ exec 3>file.log
$ lsof -a -p $$ -d0,1,2,3
COMMAND  PID  USER   FD   TYPE DEVICE SIZE  NODE NAME
bash    3443 student    0u   CHR 136,35         37 /dev/pts/35
bash    3443 student    1u   CHR 136,35         37 /dev/pts/35
bash    3443 student    2u   CHR 136,35         37 /dev/pts/35
bash    3443 student    3u   REG    3,1    0 86956 /home/student/file.log
$ echo hello >&3
$ cat file.log
hello
$ exec 3>&-
```

Files in Bash V

- Open a file in read mode works similarly:

```
#!/bin/bash
exec 4<&0
exec <restaurant.txt
while read score type phone
do
  echo $score,$type,$phone
done
exec 0<&4
exec 4>&-
```

Files in Bash VI

- Opening in read/write mode

```
$ echo 1234567890 > numbers.txt
$ exec 3<> numbers.txt
$ read -n 4 <&3          # read 4 characters (moves the pointer)
$ echo -n . >&3          # write a dot (without LF)
$ exec 3>&-              # close fd=3
$ cat numbers.txt
1234.67890
```

Files in Bash VII

- You can also do redirections using different fd but only for a single command line (not for all commands executed with the bash).

```
#!/bin/bash
exec 3>student.log                          # Open fd=3 (for bash)
echo "This goes to student.log" 1>&3        # redirects stdout to student.log
                                            # only for this command line
echo "This goes to stdout"
exec 1>&3                                   # redirects stdout to student.log
                                            # for the rest of the commands
echo "This also goes to student.log"
echo "and this sentence, too"
```

- Children processes inherit the opened *fd* of their parent process.
- The children can close an *fd* if it is not going to be used.