Chapter 20. I/O Redirection

Table of Contents

There are always three default *files* [1] open, `stdin` (the keyboard), `stdout` (the screen), and `stderr` (error messages output to the screen). These, and any other open files, can be redirected. Redirection simply means capturing output from a file, command, program, script, or even code block within a script (see Example 3-1 and Example 3-2) and sending it as input to another file, command, program, or script.

Each open file gets assigned a file descriptor. [2] The file descriptors for `stdin`, `stdout`, and `stderr` are 0, 1, and 2, respectively. For opening additional files, there remain descriptors 3 to 9. It is sometimes useful to assign one of these additional file descriptors to `stdin`, `stdout`, or `stderr` as a temporary duplicate link. [3] This simplifies restoration to normal after complex redirection and reshuffling (see Example 20-1).

```
   COMMAND_OUTPUT >
      # Redirect stdout to a file.
      # Creates the file if not present, otherwise overwrites it.

      ls -lR > dir-tree.list
      # Creates a file containing a listing of the directory tree.

   : > filename
      # The > truncates file "filename" to zero length.
      # If file not present, creates zero-length file (same effect as 'touch').
      # The : serves as a dummy placeholder, producing no output.

   > filename
      # The > truncates file "filename" to zero length.
      # If file not present, creates zero-length file (same effect as 'touch').
      # (Same result as ": >", above, but this does not work with some shells.)

   COMMAND_OUTPUT >>
      # Redirect stdout to a file.
      # Creates the file if not present, otherwise appends to it.


      # Single-line redirection commands (affect only the line they are on):
      # --------------------------------------------------------------------

   1>filename
      # Redirect stdout to file "filename."
   1>>filename
      # Redirect and append stdout to file "filename."
   2>filename
      # Redirect stderr to file "filename."
   2>>filename
      # Redirect and append stderr to file "filename."
   &>filename
      # Redirect both stdout and stderr to file "filename."
      # This operator is now functional, as of Bash 4, final release.

   M>N
     # "M" is a file descriptor, which defaults to 1, if not explicitly set.
     # "N" is a filename.
     # File descriptor "M" is redirect to file "N."
   M>&N
     # "M" is a file descriptor, which defaults to 1, if not set.
     # "N" is another file descriptor.
```

```
      #==============================================================================

      # Redirecting stdout, one line at a time.
      LOGFILE=script.log

      echo "This statement is sent to the log file, \"$LOGFILE\"." 1>$LOGFILE
      echo "This statement is appended to \"$LOGFILE\"." 1>>$LOGFILE
      echo "This statement is also appended to \"$LOGFILE\"." 1>>$LOGFILE
      echo "This statement is echoed to stdout, and will not appear in \"$LOGFILE\"."
      # These redirection commands automatically "reset" after each line.



      # Redirecting stderr, one line at a time.
      ERRORFILE=script.errors

      bad_command1 2>$ERRORFILE       #  Error message sent to $ERRORFILE.
      bad_command2 2>>$ERRORFILE      #  Error message appended to $ERRORFILE.
      bad_command3                    #  Error message echoed to stderr,
                                      #+ and does not appear in $ERRORFILE.
      # These redirection commands also automatically "reset" after each line.
      #=======================================================================
```

```
   2>&1
      # Redirects stderr to stdout.
      # Error messages get sent to same place as standard output.
        >>filename 2>&1
            bad_command >>filename 2>&1
            # Appends both stdout and stderr to the file "filename" ...
        2>&1 | [command(s)]
            bad_command 2>&1 | awk '{print $5}'   # found
            # Sends stderr through a pipe.
            # |& was added to Bash 4 as an abbreviation for 2>&1 |.

   i>&j
      # Redirects file descriptor i to j.
      # All output of file pointed to by i gets sent to file pointed to by j.

   >&j
      # Redirects, by default, file descriptor 1 (stdout) to j.
      # All stdout gets sent to file pointed to by j.
```

```
   0< FILENAME
    < FILENAME
      # Accept input from a file.
      # Companion command to ">", and often used in combination with it.
      #
      # grep search-word <filename


   [j]<>filename
      #  Open file "filename" for reading and writing,
      #+ and assign file descriptor "j" to it.
      #  If "filename" does not exist, create it.
      #  If file descriptor "j" is not specified, default to fd 0, stdin.
      #
      #  An application of this is writing at a specified place in a file.
      echo 1234567890 > File    # Write string to "File".
      exec 3<> File             # Open "File" and assign fd 3 to it.
      read -n 4 <&3             # Read only 4 characters.
      echo -n . >&3             # Write a decimal point there.
      exec 3>&-                 # Close fd 3.
      cat File                  # ==> 1234.67890
      #  Random access, by golly.



   |
      # Pipe.
      # General purpose process and command chaining tool.
      # Similar to ">", but more general in effect.
      # Useful for chaining commands, scripts, files, and programs together.
      cat *.txt | sort | uniq > result-file
      # Sorts the output of all the .txt files and deletes duplicate lines,
      # finally saves results to "result-file".
```

Multiple instances of input and output redirection and/or pipes can be combined in a single command line.

```
command < input-file > output-file
# Or the equivalent:
< input-file command > output-file   # Although this is non-standard.

command1 | command2 | command3 > output-file
```

See Example 16-31 and Example A-14.

Multiple output streams may be redirected to one file.

```
ls -yz >> command.log 2>&1
#  Capture result of illegal options "yz" in file "command.log."
#  Because stderr is redirected to the file,
#+ any error messages will also be there.

#  Note, however, that the following does *not* give the same result.
ls -yz 2>&1 >> command.log
#  Outputs an error message, but does not write to file.
#  More precisely, the command output (in this case, null)
#+ writes to the file, but the error message goes only to stdout.

#  If redirecting both stdout and stderr,
#+ the order of the commands makes a difference.
```

Closing File Descriptors

n<&-

Close input file descriptor *n*.

0<&-, <&-

Close `stdin`.

n>&-

Close output file descriptor *n*.

1>&-, >&-

Close `stdout`.

Child processes inherit open file descriptors. This is why pipes work. To prevent an fd from being inherited, close it.

```
# Redirecting only stderr to a pipe.

exec 3>&1                              # Save current "value" of stdout.
ls -l 2>&1 >&3 3>&- | grep bad 3>&-    # Close fd 3 for 'grep' (but not 'ls').
#              ^^^^   ^^^^
exec 3>&-                              # Now close it for the remainder of the script.

# Thanks, S.C.
```

For a more detailed introduction to I/O redirection see Appendix F.

Notes

[1] By convention in UNIX and Linux, data streams and peripherals (device files) are treated as files, in a fashion analogous to ordinary files.

[2] A *file descriptor* is simply a number that the operating system assigns to an open file to keep track of it. Consider it a simplified type of file pointer. It is analogous to a *file handle* in **C**.

[3] Using `file descriptor 5` might cause problems. When Bash creates a child process, as with exec, the child inherits fd 5 (see Chet Ramey's archived e-mail, SUBJECT: RE: File descriptor 5 is held open). Best leave this

particular fd alone.

Prev | Home | Next
Here Strings | Up | Using *exec*

Appendix F. A Detailed Introduction to I/O and I/O Redirection

written by Stéphane Chazelas, and revised by the document author

A command expects the first three file descriptors to be available. The first, *fd 0* (standard input, `stdin`), is for reading. The other two (*fd 1*, `stdout` and *fd 2*, `stderr`) are for writing.

There is a `stdin`, `stdout`, and a `stderr` associated with each command. **`ls 2>&1`** means temporarily connecting the `stderr` of the **ls** command to the same "resource" as the shell's `stdout`.

By convention, a command reads its input from fd 0 (`stdin`), prints normal output to fd 1 (`stdout`), and error ouput to fd 2 (`stderr`). If one of those three fd's is not open, you may encounter problems:

```
bash$ cat /etc/passwd >&-
cat: standard output: Bad file descriptor
```

For example, when **xterm** runs, it first initializes itself. Before running the user's shell, **xterm** opens the terminal device (/dev/pts/<n> or something similar) three times.

At this point, Bash inherits these three file descriptors, and each command (child process) run by Bash inherits them in turn, except when you redirect the command. Redirection means reassigning one of the file descriptors to another file (or a pipe, or anything permissible). File descriptors may be reassigned locally (for a command, a command group, a subshell, a while or if or case or for loop...), or globally, for the remainder of the shell (using exec).

`ls > /dev/null` means running **ls** with its fd 1 connected to `/dev/null`.

```
bash$ lsof -a -p $$ -d0,1,2
COMMAND PID     USER   FD   TYPE DEVICE SIZE NODE NAME
 bash    363 bozo        0u   CHR  136,1         3 /dev/pts/1
 bash    363 bozo        1u   CHR  136,1         3 /dev/pts/1
 bash    363 bozo        2u   CHR  136,1         3 /dev/pts/1


bash$ exec 2> /dev/null
bash$ lsof -a -p $$ -d0,1,2
COMMAND PID     USER   FD   TYPE DEVICE SIZE NODE NAME
 bash    371 bozo        0u   CHR  136,1         3 /dev/pts/1
 bash    371 bozo        1u   CHR  136,1         3 /dev/pts/1
 bash    371 bozo        2w   CHR    1,3       120 /dev/null


bash$ bash -c 'lsof -a -p $$ -d0,1,2' | cat
COMMAND PID USER   FD   TYPE DEVICE SIZE NODE NAME
 lsof    379 root    0u   CHR  136,1         3 /dev/pts/1
 lsof    379 root    1w  FIFO    0,0      7118 pipe
 lsof    379 root    2u   CHR  136,1         3 /dev/pts/1


bash$ echo "$(bash -c 'lsof -a -p $$ -d0,1,2' 2>&1)"
COMMAND PID USER   FD   TYPE DEVICE SIZE NODE NAME
 lsof    426 root    0u   CHR  136,1         3 /dev/pts/1
 lsof    426 root    1w  FIFO    0,0      7520 pipe
 lsof    426 root    2w  FIFO    0,0      7520 pipe
```

This works for different types of redirection.

Exercise: Analyze the following script.

```
#! /usr/bin/env bash

mkfifo /tmp/fifo1 /tmp/fifo2
while read a; do echo "FIFO1: $a"; done < /tmp/fifo1 & exec 7> /tmp/fifo1
exec 8> >(while read a; do echo "FD8: $a, to fd7"; done >&7)

exec 3>&1
(
 (
  (
   while read a; do echo "FIFO2: $a"; done < /tmp/fifo2 | tee /dev/stderr \
   | tee /dev/fd/4 | tee /dev/fd/5 | tee /dev/fd/6 >&7 & exec 3> /tmp/fifo2

   echo 1st, to stdout
   sleep 1
   echo 2nd, to stderr >&2
   sleep 1
   echo 3rd, to fd 3 >&3
   sleep 1
   echo 4th, to fd 4 >&4
   sleep 1
   echo 5th, to fd 5 >&5
   sleep 1
   echo 6th, through a pipe | sed 's/.*/PIPE: &, to fd 5/' >&5
   sleep 1
   echo 7th, to fd 6 >&6
   sleep 1
   echo 8th, to fd 7 >&7
   sleep 1
   echo 9th, to fd 8 >&8

  ) 4>&1 >&3 3>&- | while read a; do echo "FD4: $a"; done 1>&3 5>&- 6>&-
 ) 5>&1 >&3 | while read a; do echo "FD5: $a"; done 1>&3 6>&-
) 6>&1 >&3 | while read a; do echo "FD6: $a"; done 3>&-

rm -f /tmp/fifo1 /tmp/fifo2


# For each command and subshell, figure out which fd points to what.
# Good luck!

exit 0
```