20.1. Using *exec*

An **exec <filename** command redirects `stdin` to a file. From that point on, all `stdin` comes from that file, rather than its normal source (usually keyboard input). This provides a method of reading a file line by line and possibly parsing each line of input using sed and/or awk.

Example 20-1. Redirecting `stdin` using *exec*

```
#!/bin/bash
# Redirecting stdin using 'exec'.


exec 6<&0          # Link file descriptor #6 with stdin.
                   # Saves stdin.

exec < data-file   # stdin replaced by file "data-file"

read a1            # Reads first line of file "data-file".
read a2            # Reads second line of file "data-file."

echo
echo "Following lines read from file."
echo "-------------------------------"
echo $a1
echo $a2

echo; echo; echo

exec 0<&6 6<&-
#  Now restore stdin from fd #6, where it had been saved,
#+ and close fd #6 ( 6<&- ) to free it for other processes to use.
#
# <&6 6<&-    also works.

echo -n "Enter data  "
read b1  # Now "read" functions as expected, reading from normal stdin.
echo "Input read from stdin."
echo "----------------------"
echo "b1 = $b1"

echo

exit 0
```

Similarly, an **exec >filename** command redirects `stdout` to a designated file. This sends all command output that would normally go to `stdout` to that file.

exec N > filename affects the entire script or *current shell*. Redirection in the PID of the script or shell from that point on has changed. However . . .

N > filename affects only the newly-forked process, not the entire script or shell.

Thank you, Ahmed Darwish, for pointing this out.

Example 20-2. Redirecting `stdout` using *exec*

```
#!/bin/bash
# reassign-stdout.sh

LOGFILE=logfile.txt

exec 6>&1           # Link file descriptor #6 with stdout.
                    # Saves stdout.

exec > $LOGFILE     # stdout replaced with file "logfile.txt".

# ----------------------------------------------------------- #
# All output from commands in this block sent to file $LOGFILE.

echo -n "Logfile: "
date
echo "-------------------------------------"
echo

echo "Output of \"ls -al\" command"
echo
ls -al
echo; echo
echo "Output of \"df\" command"
echo
df

# ----------------------------------------------------------- #

exec 1>&6 6>&-      # Restore stdout and close file descriptor #6.

echo
echo "== stdout now restored to default == "
echo
ls -al
echo

exit 0
```

Example 20-3. Redirecting both `stdin` and `stdout` in the same script with *exec*

```
#!/bin/bash
# upperconv.sh
# Converts a specified input file to uppercase.

E_FILE_ACCESS=70
E_WRONG_ARGS=71

if [ ! -r "$1" ]     # Is specified input file readable?
then
  echo "Can't read from input file!"
  echo "Usage: $0 input-file output-file"
  exit $E_FILE_ACCESS
fi                   #  Will exit with same error
                     #+ even if input file ($1) not specified (why?).

if [ -z "$2" ]
then
  echo "Need to specify output file."
  echo "Usage: $0 input-file output-file"
  exit $E_WRONG_ARGS
fi
```

```
exec 4<&0
exec < $1            # Will read from input file.

exec 7>&1
exec > $2            # Will write to output file.
                     # Assumes output file writable (add check?).

# -----------------------------------------------
    cat - | tr a-z A-Z   # Uppercase conversion.
#   ^^^^^                # Reads from stdin.
#           ^^^^^^^^^^   # Writes to stdout.
# However, both stdin and stdout were redirected.
# Note that the 'cat' can be omitted.
# -----------------------------------------------

exec 1>&7 7>&-       # Restore stout.
exec 0<&4 4<&-       # Restore stdin.

# After restoration, the following line prints to stdout as expected.
echo "File \"$1\" written to \"$2\" as uppercase conversion."

exit 0
```

I/O redirection is a clever way of avoiding the dreaded inaccessible variables within a subshell problem.

Example 20-4. Avoiding a subshell

```
#!/bin/bash
# avoid-subshell.sh
# Suggested by Matthew Walker.

Lines=0

echo

cat myfile.txt | while read line;
                 do {
                   echo $line
                   (( Lines++ ));  #  Incremented values of this variable
                                   #+ inaccessible outside loop.
                                   #  Subshell problem.
                 }
                 done

echo "Number of lines read = $Lines"     # 0
                                         # Wrong!

echo "------------------------"


exec 3<> myfile.txt
while read line <&3
do {
  echo "$line"
  (( Lines++ ));                   #  Incremented values of this variable
                                   #+ accessible outside loop.
                                   #  No subshell, no problem.
}
done
exec 3>&-

echo "Number of lines read = $Lines"     # 8
```

```
echo

exit 0

# Lines below not seen by script.

$ cat myfile.txt

Line 1.
Line 2.
Line 3.
Line 4.
Line 5.
Line 6.
Line 7.
Line 8.
```

Prev Home Next
I/O Redirection Up Redirecting Code Blocks