```
        grep "search string" *.c *.h

To search for all files with a certain string in their name in the
current directory and all its subdirectories, do this

        find . -name "*string*" -print

To search all files in the current directory and all subdirectories
for a certain string, do this

        find . -exec grep "string" {} /dev/null \;

The /dev/null is a LINUX guru trick to get grep to print out the file
name in addition to the line containing the matching string.
```

(D) You are to combine the C programs from (A), (B) and (C) that: (1) reads from the first file argument, reverses lowercase and uppercase letters, that is, makes lowercase all uppercase letters and vice versa, and writes out all characters, whether or not reversed; and (2) reads the above output, counts the number of NON-alphabetic characters (that is, those not in the a-z range nor in the A-Z range), and writes out all characters, whether counted or not, into its second file argument.

So that you become familiar with the various LINUX system calls and libraries (fork, execl, pipe, creat, open, close, dup, stdio, exit), you will write this program in a certain contrived fashion. The program will be invoked as ``driver file1 file2''. Both file arguments are required (print an error message on stderr using fprintf if either argument is missing and then exit(1)). The driver program will open the first file and creat the second file, and dup them down to stdin and stdout. The driver program sets up a pipe and then forks two children.

The first child forked will dup the read end of the pipe down to stdin and then execl the program you have written, count, which will read characters from stdin, count the non-alphabetic ones (see /usr/include/ctype.h), and write all characters out to stdout. The count program uses only stdin and stdout, and only the stdio library routines (see stdio.h) for input and output (see getchar and putchar). The program will print out the final count on stderr using fprintf and then exit(0) when it hits EOF in its input.

The second child forked will dup the write end of the pipe down to stdout and then execl the program you have written, convert, which will read characters from stdin, reverse uppercase and lowercase (again, see ctype.h), and write them all out to stdout. Like count, this program uses only stdin and stdout. Also, it uses only the stdio library routines. The program will exit(0) when it hits EOF in its input.

The reason for creating the children in this order (first the one that reads from the pipe, usually called the second in the pipeline, and then second the one that writes to the pipe, usually called the first in the pipeline -- confusing!!) is that LINUX will not let a process write to a pipe if no process has the pipe open for reading; the process trying to write would get the SIGPIPE signal. So we create first the process that reads from the pipe to avoid that possibility.

Meanwhile, the parent process, the driver, will close its pipe file descriptors and then call wait (see /usr/include/sys/wait.h) twice to reap the zombie children processes when they finish. Then the parent will exit(0).

Remember to close all unneeded file descriptors, including unneeded pipe ones, or EOF on a pipe read may not be detected correctly. Check all system calls for the error return (-1). For all error and debug messages, use ``fprintf(stderr, ...)" or use the function perror or the external variable errno (see also /usr/include/errno.h). Declare a function to be of type void if it does not return a value, that is, if it is a subroutine rather than a true function. If you absolutely must ignore the output of a system call or function, then cast the call to void.

Hints and Notes

Program skeleton

- parent opens file1:

```
#include <sys/types.h>
#include <fcntl.h>    /* defines O_RDONLY etc. */

...
fd_in = open(argv[1], O_RDONLY);    /* not fopen */
    /* Check fd_in for -1!!!  If it is, then the file does not
       exist or you do not have read permission. */
```

- parent creat's file2:

```
fd_out = creat(argv[2], 0644);  /* mode = permissions, here rw-r--r-- */
    /* Check fd_out for -1!!!  If it is, then the file could not
       be created */
```

- parent uses close and dup so stdin is now file1 (done like children below)

- parent uses close and dup so stdout is now file2: if dup returns -1, then a problem has occurred

- parent calls pipe system call to create pipe file descriptors

- parent forks a child to read from pipe:

  -- this first child manipulates file descriptors

```
        use close, dup so stdin is read end of pipe
        (see text Figure 1-13, similar to else clause)

    -- this first child execs count
        execl("count", "count", (char *) 0)
        execl overlays the child with the binary compiled program in
        the file given by first argument and the process name becomes
        the second argument
```

- parent forks again a child to write to pipe:

```
    -- this second child manipulates file descriptors
        use close, dup so stdout is write end of pipe
        (similar to if clause in text Figure 1-13)

    -- this second child execs convert
        execl("convert", "convert", (char *) 0)
```

- parent closes both ends of pipe

- parent waits twice (see text Figure 1-10)

```
    -- use wait(&status); instead of waitpid(-1, &status, 0);
```

Before compiling and running driver, remember to:

- in count.c: fprintf(stderr, "final count = %d\n", count);

- cc -o convert convert.c, and

- cc -o count count.c

The two forks form nested if statements.

```
    pipe(...
    if (fork() != 0) {    /* parent continues here */

        if (fork() != 0) {    /* parent continues here */
            close(write end of pipe...
            wait(...
            wait(...
        } else {
            ...            /* second child, writes to pipe */
        }

    } else {
            ...            /* first child, reads from pipe */
    }
```

It is important to check all system calls (open, creat, dup, etc.) for a return value <0, particularly -1, because such a return value means an error has occurred. If you just let your program continue to execute, it will just dump core at some later time and you will not know why.

It is IMPERATIVE that the parent and the first child forked (the one to read the pipe) close their write end file descriptors to the pipe. If they do not do this, then the first child will never get EOF reading from the pipe, even after the second child has sent everything. This is because EOF on a pipe is not indicated until ALL

write file descriptors to the pipe are closed. If the parent and first child fail to close their file descriptors for the write end of the pipe, the program will hang forever (until killed). Furthermore, the parent must close its copy of the file descriptor to the write end of the pipe BEFORE the waits, not after, or deadlock will result. And the first child forked (the one to read the pipe) must close its file descriptor to the write end of the pipe before it starts reading the pipe, not after, or deadlock will result.

How dup works

dup(fd) looks for the smallest unused slot in the process descriptor table and makes that slot point to the same file, pipe, whatever, that fd points to. For example, each process starts as

process descriptor table

slot #	points to
0 (stdin)	keyboard
1 (stdout)	screen
2 (stderr)	screen

If the process does a

```
fd_in = open("file1", ...
fd_out = creat("file2", ...
```

then the table now looks like

process descriptor table

slot #	points to
0	keyboard
1	screen
2	screen
3	file1
4	file2

and fd_in is 3 and fd_out is 4. Now suppose the program does a

```
close(0);
dup(fd_in);
close(fd_in);
```

After the close(0), the table will look like

process descriptor table

slot #	points to
0	-- unused --
1	screen
2	screen
3	file1
4	file2

and after the dup(fd_in), the table will look like

```
                        process descriptor table
                        ------------------------
                        slot #         points to
                        ------         ---------
                             0             file1
                             1            screen
                             2            screen
                             3             file1
                             4             file2

and after the close(fd_in), the table will look like

                        process descriptor table
                        ------------------------
                        slot #         points to
                        ------         ---------
                             0             file1
                             1            screen
                             2            screen
                             3      -- unused --
                             4             file2
```

(E) You are to modify part (D) of your C program. This is to get you familiar with more of the various LINUX system calls and libraries (setjmp, longjmp, getpid, alarm, read, write, kill, signal).

The program will be invoked as ``driver [-n] [file1] [file2]''. All arguments are optional. If there is just one file, then the parent opens it for reading, and dups it down to stdin. If there are two files, then we have the same situation as part (D) above. If there are no files, then the parent leaves its stdin and stdout alone. (The other case could be handled at the command level by invoking the driver as ``driver [-n] >file2''). If the -n argument is present, where n is a positive decimal number, then the first child forked, the one that is going to overlay itself with the count program, passes the entire -n argument to the program count, which overlays it. The first child does this in its execl statement. (It is as if the command ``count -n'' had been entered).

The parent sets up a pipe and forks two children, as in part (D). Before going into its wait loop, waiting for the children to complete, the parent (worried about children getting hung) sets up a signal handler for the SIGALRM signal (see /usr/include/signal.h), using the signal system call. The parent next saves its stack environment, using the setjmp library routine (see /usr/include/setjmp.h). The parent then sets an alarm to go off in 15 seconds, using the alarm system call. Finally, the parent enters its wait loop, as in part (D). The parent looks something like Figure 1.

If both children exit normally (exit(0)) before the alarm goes off, then the parent prints a ``normal children exit'' message on stderr,