Input and Output Redirection

You can use your knowledge of processes to alter the behavior of programs by exploiting the fact that open file descriptors are preserved across calls to `fork` and `exec`. The next example involves a *filter program* — a program that reads from its standard input and writes to its standard output, performing some useful transformation as it does so.

Try It Out Redirection

Here's a very simple filter program, `upper.c`, that reads input and converts it to uppercase:

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

int main()
{
    int ch;
    while((ch = getchar()) != EOF) {
        putchar(toupper(ch));
    }
    exit(0);
}
```

When you run the program, it does what you expect:

```
$ ./upper
hello THERE
HELLO THERE
^D
$
```

You can, of course, use it to convert a file to uppercase by using the shell redirection

```
$ cat file.txt
this is the file, file.txt, it is all lower case.
$ ./upper < file.txt
THIS IS THE FILE, FILE.TXT, IT IS ALL LOWER CASE.
```

What if you want to use this filter from within another program? This program, `useupper.c`, accepts a filename as an argument and will respond with an error if called incorrectly.

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    char *filename;

    if (argc != 2) {
        fprintf(stderr, "usage: useupper file\n");
```

```
            exit(1);
        }

        filename = argv[1];
```

You reopen the standard input, again checking for any errors as you do so, and then use `execl` to call `upper`.

```
        if(!freopen(filename, "r", stdin)) {
            fprintf(stderr, "could not redirect stdin from file %s\n", filename);
            exit(2);
        }

        execl("./upper", "upper", 0);
```

Don't forget that `execl` replaces the current process; if there is no error, the remaining lines are not executed.

```
        perror("could not exec ./upper");
        exit(3);
    }
```

How It Works

When you run this program, you can give it a file to convert to uppercase. The job is done by the program `upper`, which doesn't handle filename arguments. Note that you don't require the source code for `upper`; you can run any executable program in this way:

```
$ ./useupper file.txt
THIS IS THE FILE, FILE.TXT, IT IS ALL LOWER CASE.
```

The `useupper` program uses `freopen` to close the standard input and associate the file stream `stdin` with the file given as a program argument. It then calls `execl` to replace the running process code with that of the `upper` program. Because open file descriptors are preserved across the call to `execl`, the `upper` program runs exactly as it would have under the shell command:

```
$ ./upper < file.txt
```

Threads

Linux processes can cooperate, can send each other messages, and can interrupt one another. They can even arrange to share segments of memory between themselves, but they are essentially separate entities within the operating system. They do not readily share variables.

There is a class of process known as a *thread* that is available in many UNIX and Linux systems. Though threads can be difficult to program, they can be of great value in some applications, such as multithreaded database servers. Programming threads on Linux (and UNIX generally) is not as common as using multiple processes, because Linux processes are quite lightweight, and programming multiple cooperation processes is much easier than programming threads. Threads are covered in Chapter 12.