

copied and the name of the copied file. The program will then create an ordinary pipe and write the contents of the file to be copied to the pipe. The child process will read this file from the pipe and write it to the destination file. For example, if we invoke the program as follows:

```
filecopy input.txt copy.txt
```

the file `input.txt` will be written to the pipe. The child process will read the contents of this file and write it to the destination file `copy.txt`. You may write this program using either UNIX or Windows pipes.

Programming Projects

Project 1—UNIX Shell and History Feature

This project consists of designing a C program to serve as a shell interface that accepts user commands and then executes each command in a separate process. This project can be completed on any Linux, UNIX, or Mac OS X system.

A shell interface gives the user a prompt, after which the next command is entered. The example below illustrates the prompt `osh>` and the user's next command: `cat prog.c`. (This command displays the file `prog.c` on the terminal using the UNIX `cat` command.)

```
osh> cat prog.c
```

One technique for implementing a shell interface is to have the parent process first read what the user enters on the command line (in this case, `cat prog.c`), and then create a separate child process that performs the command. Unless otherwise specified, the parent process waits for the child to exit before continuing. This is similar in functionality to the new process creation illustrated in Figure 3.10. However, UNIX shells typically also allow the child process to run in the background, or concurrently. To accomplish this, we add an ampersand (&) at the end of the command. Thus, if we rewrite the above command as

```
osh> cat prog.c &
```

the parent and child processes will run concurrently.

The separate child process is created using the `fork()` system call, and the user's command is executed using one of the system calls in the `exec()` family (as described in Section 3.3.1).

A C program that provides the general operations of a command-line shell is supplied in Figure 3.36. The `main()` function presents the prompt `osh->` and outlines the steps to be taken after input from the user has been read. The `main()` function continually loops as long as `should_run` equals 1; when the user enters `exit` at the prompt, your program will set `should_run` to 0 and terminate.

This project is organized into two parts: (1) creating the child process and executing the command in the child, and (2) modifying the shell to allow a history feature.

```

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

#define MAX_LINE 80 /* The maximum length command */

int main(void)
{
    char *args[MAX_LINE/2 + 1]; /* command line arguments */
    int should_run = 1; /* flag to determine when to exit program */

    while (should_run) {
        printf("osh>");
        fflush(stdout);

        /**
         * After reading user input, the steps are:
         * (1) fork a child process using fork()
         * (2) the child process will invoke execvp()
         * (3) if command included &, parent will invoke wait()
         */
    }

    return 0;
}

```

Figure 3.36 Outline of simple shell.

Part I— Creating a Child Process

The first task is to modify the `main()` function in Figure 3.36 so that a child process is forked and executes the command specified by the user. This will require parsing what the user has entered into separate tokens and storing the tokens in an array of character strings (`args` in Figure 3.36). For example, if the user enters the command `ps -ael` at the `osh>` prompt, the values stored in the `args` array are:

```

args[0] = "ps"
args[1] = "-ael"
args[2] = NULL

```

This `args` array will be passed to the `execvp()` function, which has the following prototype:

```
execvp(char *command, char *params[]);
```

Here, `command` represents the command to be performed and `params` stores the parameters to this command. For this project, the `execvp()` function should be invoked as `execvp(args[0], args)`. Be sure to check whether the user included an `&` to determine whether or not the parent process is to wait for the child to exit.