two

Shell Program

Exercise Goal: You will learn how to write a UNIX shell program. This will give you the opportunity to learn how child processes are created to perform large-grained work and how the parent process can follow up on a child process's work.

Introduction

In Part 1, Section 3.6, you studied how a process is assigned to each login port when the machine is booted up and how a copy of a shell program is executed on behalf of a user that logs in on any port. A *shell*, or *command line interpreter* program, is a mechanism with which each interactive user can send commands to the OS and by which the OS can respond to the user. Whenever a user has successfully logged in to the computer, the OS causes the user process assigned to the login port to execute a specific shell. The OS does not ordinarily have a built-in window interface. Instead, it assumes a simple character-oriented interface in which the user types a string of characters (terminated by pressing the Enter or Return key) and the OS responds by typing lines of characters back to the screen. If the human-computer interface is to be a graphical window interface, then the software that implements the window manager

subsumes the shell tasks that are the focus of this exercise. Thus the character-oriented shell assumes a screen display with a fixed number of lines (usually 25) and a fixed number of characters (usually 80) per line.

Once the shell has initialized its data structures and is ready to start work, it clears the 25-line display and prints a prompt in the first few character positions on the first line. Linux systems are usually configured to include the machine name as part of the prompt. For example, my Linux machine is named kiowa.cs.colorado.edu, so the shell prints, as its prompt string,

```
kiowa>
```

or

```
bash>
```

depending on which shell I am using. The shell then waits for the user to type a command line in response to the prompt. The command line could be a string such as

```
kiowa> ls –al
```

terminated with an enter or return character (in Linux, this character is represented internally by the NEWLINE character, '\n'). When the user enters a command line, the shell's job is to cause the OS to execute the command embedded in the command line.

Every shell has its own language syntax and semantics. In the standard Linux shell, bash, a command line has the form

```
command argument_1 argument_2 ...
```

in which the first word is the command to be executed and the remaining words are arguments expected by that command. The number of arguments depends on which command is being executed. For example, the directory listing command may have no arguments—simply by the user's typing ls—or it may have arguments prefaced by the negative "-" character, as in ls –al, where a and l are arguments. The command determines the syntax for the arguments, such as which of the arguments may be grouped (as for the a and l in the ls command), which arguments must be preceded by a "-" character, and whether the position of the argument is important

Other commands use a different argument-passing syntax. For example, a C compiler command might look like

```
kiowa> cc –g –o deviation  –S main.c inout.c –lmath
```

in which the arguments g, o deviation, S, main.c, inout.c, and lmath are all passed to the C compiler, cc.

The shell relies on an important convention to accomplish its task: The command for the command line is usually the name of a file that contains an executable program, for example, ls and cc (files stored in /bin on most UNIX-style machines). In a few cases, the command is not a filename but rather a command that is implemented within the shell. For example, cd (change directory) is usually implemented within the shell itself rather than in a file in /bin. Because the vast majority of the commands are implemented in files, you can think of the command as actually being a filename in some directory on the machine. This means that the shell's job is to find the file, prepare the list of parameters for the command, and then cause the command to be executed using the parameters.

Many shell programs are used with UNIX variants, including the original Bourne shell (sh), the C shell (csh) with its additional features over sh, the Korn shell, and the standard Linux shell (bash). All have followed a similar set of rules for command line syntax, though each has a superset of features.

Basic UNIX-Style Shell Operation

The Bourne shell is described in Ritchie and Thompson's original UNIX paper [Ritchie and Thompson, 1974]. As described in the previous subsection, the shell should accept a command line from the user, parse the command line, and then invoke the OS to run the specified command with the specified arguments. This command line is a request to execute the program in any file that contains a program, including programs that the user wrote. Thus a programmer can write an ordinary C program, compile it, and have the shell execute it just like it was a UNIX command.

For example, suppose that you write a C program in a file named main.c and then compile and execute it with shell commands such as

```
kiowa> cc main.c
kiowa> a.out
```

For the first command line, the shell will find the cc command (the C compiler) in the /bin directory and then, when the cc command is executed, pass it the string main.c. The C compiler, by default, will translate the C program that is stored in main.c and write the resulting executable program into a file named a.out in the current directory. The second command line is simply the name of the file to be executed, a.out, without any parameters. The shell finds the a.out file in the current directory and then executes it.

Consider the following steps that a shell must take to accomplish its job.

1. Print a prompt.

   A default prompt string is available, sometimes hardcoded into the shell, for example the single character string %, #, or >. When the shell is started, it can look up the name of the machine on which it is running and prepend this string to the standard prompt character, for example a prompt string such as kiowa>. The shell also can be designed to print the current directory as part of the prompt, meaning that each time that the user types cd to change to a different directory, the prompt string is redefined. Once the prompt string is determined, the shell prints it to stdout whenever it is ready to accept a command line.

2. Get the command line.

   To get a command line, the shell performs a blocking read operation so that the process that executes the shell will be asleep until the user types a command line in response to the prompt. Once the user types the command line (and terminates it with a NEWLINE ('\n') character), the command line string is returned to the shell.

3. Parse the command.

   The syntax for the command line is trivial. The parser begins at the left side of the command line and scans until it sees a whitespace character (such as space, tab, or NEWLINE). The first word is the command name, and subsequent words are the parameters.

4. Find the file.

   The shell provides a set of *environment variables* for each user. These variables are first defined in the user's .login file, though they can be modified at any time by using the set command. The PATH environment variable is an ordered list of absolute pathnames specifying where the shell should search for command files. If the .login file has a line such as

   ```
   set path=(. /bin /usr/bin)
   ```

   then the shell will first look in the current directory (since the first full pathname is "." for the current directory), then in /bin, and finally in /usr/bin. If no file with the same name as the command can be found (from the command line) in any of the specified directories, then the shell notifies the user that it is unable to find the command.

5. Prepare the parameters.

   The shell simply passes the parameters to the command as the argv array of pointers to strings.

6. Execute the command.

The shell must execute the executable program in the specified file. UNIX shells have always been designed to protect the original process from crashing when it executes a program. That is, since a command can be *any* executable file, then the process that is executing the shell must protect itself in case the executable file contains a fatal error. Somehow, the shell wants to launch the executable so that even if the executable contains a fatal error (which destroys the process executing it), then the shell will remain unharmed. The Bourne shell uses multiple processes to accomplish this by using the UNIX-style system calls fork(), execve(), and wait().

- fork()

  The fork() system call *creates* a new process that is a copy of the calling process, except that it has its own copy of the memory, its own process ID (with the correct relationships to other processes), and its own pointers to shared kernel entities such as file descriptors. After fork() has been called, *two* processes will execute the next statement after the fork() in their own address spaces: the parent and the child. If the call succeeds, then in the parent process fork() returns the process ID of the newly created child process and in the child process, fork() returns a zero value.

- execve()

  The execve() system call *changes* the program that a process is currently executing. It has the form

  ```
  execve(char *path, char *argv[], char *envp[])
  ```

  The path argument is the pathname of a file that contains the new program to be executed. The argv array is a list of parameter strings, and the envp array is a list of environment variable strings and values that should be used when the process begins executing the new program. When a process encounters the execve() system call, the next instruction it executes will be the one at the entry point of the new executable file. Thus the kernel performs a considerable amount of work in this system call. It must

  - find the new executable file,
  - load the file into the address space currently being used by the calling process (overwriting and discarding the previous program),
  - set the argv array and environment variables for the new program execution, and

- start the process executing at the new program's entry point.

  Various versions of execve()are available at the system call interface, differing in the way that parameters are specified (for example, some use a full pathname for the executable file and others do not).

- wait()

  The wait() system call is used by a process to block itself until the kernel signals the process to execute again, for example because one of its child processes has terminated. When the wait() call returns as a result of a child process's terminating, the status of the terminated child is returned as a parameter to the calling process.

When these three system calls are used, here is the code skeleton that a shell might use to execute a command.

```
if(fork() == 0) {
  // This is the child
  // Execute in same environment as parent
    execvp(full_pathname, command->argv, 0);

} else {
  // This is the parent—wait for child to terminate
    wait(status);
}
```

Putting a Process in the Background

In the normal paradigm for executing a command, the parent process creates a child process, starts it executing the command, and then waits until the child process terminates. If the "and" ("&")operator is used to terminate the command line, then the shell is expected to create the child process and start it executing on the designated command but not have the parent wait for the child to terminate. That is, the parent and the child will execute *concurrently*. While the child executes the command, the parent prints another prompt to stdout and waits for the user to enter another command line. If the user starts several commands, each terminated by an "&", and each takes a relatively long time to execute, then many processes can be running at the same time.

When a child process is created and started executing on its own program, both the child and the parent expect their stdin stream to come from the user via the keyboard and for their stdout stream to be written to the character terminal display. Notice that if multiple child processes are running concurrently and

all expect the keyboard to define their stdin stream, then the user will not know which child process will receive data on its stdin if data is typed to the keyboard. Similarly, if any of the concurrent processes write characters to stdout, those characters will be written to the terminal display wherever the cursor happens to be positioned. The kernel makes no provision for giving each child process its own keyboard or terminal (unlike a windows system, which controls the multiplexing and demultiplexing through explicit user actions).

I/O Redirection

A process, when created, has three default file identifiers: stdin, stdout, and stderr. If it reads from stdin, then the data that it receives will be directed from the keyboard to the stdin file descriptor. Similarly, data received from stdout and stderr are mapped to the terminal display.

The user can redefine stdin or stdout whenever a command is entered. If the user provides a filename argument to the command and precedes the filename with a left angular brace character ,"<," then the shell will substitute the designated file for stdin; this is called redirecting the input from the designated file.

The user can redirect the *output* (for the execution of a single command) by preceding a filename with the right angular brace character, ">," character. For example, a command such as

```
kiowa> wc < main.c > program.stats
```

will create a child process to execute the wc command. Before it launches the command, however, it will redirect stdin so that it reads the input stream from the file main.c and redirect stdout so that it writes the output stream to the file program.stats.

The shell can redirect I/O by manipulating the child process's file descriptors. A newly created child process inherits the open file descriptors of its parent, specifically the same keyboard for stdin and the terminal display for stdout and stderr. (This expands on why concurrent processes read and write the same keyboard and display.) The shell can change the child's file descriptors so that it reads and writes streams to files rather than to the keyboard and display.

Each process has its own file descriptor table in the kernel (called fileDescriptor in this discussion, but labeled differently in the source code); for more on the file descriptor, see Part 1, Section 7.2. When the process is created, the first entry in this table, by convention, refers to the keyboard and the second two refer to the terminal display.

Next, the C runtime environment and the kernel manage the symbols stdin, stdout, and stderr so that stdin is always bound to fileDescriptor[0] in the kernel table, stdout is bound to fileDescriptor[1], and stderr to fileDescriptor[2].

You can use the close() system call to close any open file, including stdin, stdout, and stderr. By convention, the dup() and open() commands always use the earliest available entry in the file descriptor table. Therefore a code fragment such as the following:

```
fid = open(foo, O_WRONLY | O_CREAT);
close(1);
dup(fid);
close(fid);
```

is guaranteed to create a file descriptor, fid, to duplicate the entry and to place the duplicate in fileDescriptor[1] (the usual stdout entry in the process's file descriptor table). As a result, characters written by the process to stdout will be written to the file foo. This is the key to redirecting both stdin and stdout.

Shell Pipes

The *pipe* is the main IPC mechanism in uniprocessor Linux and other versions of UNIX. By default, a pipe employs asynchronous send and blocking receive operations. Optionally, the blocking receive operation may be changed to be a nonblocking receive (see Section 2.1.5 for details on invoking nonblocking read operations). Pipes are FIFO (first-in/first out) buffers designed with an API that resembles as closely as possible the file I/O interface. A pipe may contain a system-defined maximum number of bytes at any given time, usually 4KB. As indicated in Figure 2.2, a process can send data by writing it into one end of the pipe and another can receive the data by reading the other end of the pipe.

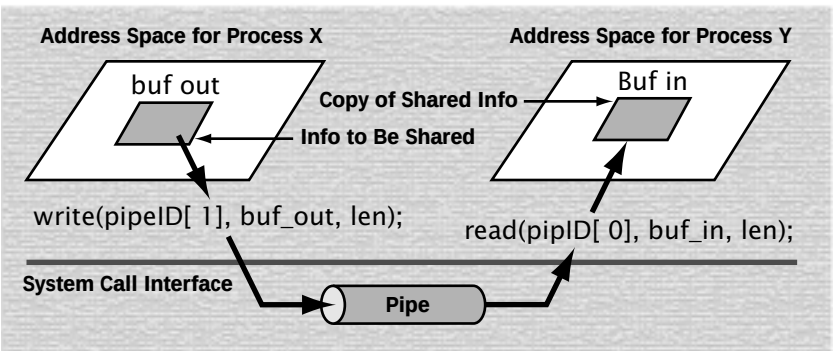

Figure 2.2
Information Flow Through a Pipe

A pipe is represented in the kernel by a file descriptor. A process that wants to create a pipe calls the kernel with a call of the following form.

```
int pipeID[2];
...
pipe(pipeID);
```


The kernel creates the pipe as a kernel FIFO data structure with two file identifiers. In this example code, pipeID[0] is a file pointer (an index into the process's open file table) to the read end of the pipe and pipeID[1] is file pointer to the write end of the pipe.

For two or more processes to use pipes for IPC (interprocess communication), a common ancestor of the processes must create the pipe prior to creating the processes. Because the fork command creates a child that contains a copy of the open file table (that is, the child has access to all of the files that the parent has already opened), the child inherits the pipes that the parent created. To use a pipe, it needs only to read and write the proper file descriptors.

For example, suppose that a parent creates a pipe. It then can create a child and communicate with it by using a code fragment such as the following.

```
...
pipe(pipeID);
if(fork()  ==  0) { /* The child process */
  ...
  read(pipeID[0], childBuf, len);
  /* Process the message in childBuf */
  ...
} else { /* The parent process */
  ...
  /* Send a message to the child */
  write(pipeID[1], msgToChild, len);
  ...
}
```

A pipe is used in place of a mailbox. In a mailbox, the asynchronous send operation is a normal write() system call on the write end of the pipe (pipe descriptor pipe_id[1]) and the read() operation is a blocking read on the read end of the pipe (pipe descriptor pipe_id[0]).

Pipes enable processes to copy information from one address space to another by using the UNIX file model. The pipe read and write ends can be used in most system calls in the same way as a file descriptor. Further, the information

written to and read from the pipe is a byte stream. UNIX pipes do not explicitly support messages, though two processes can establish their own protocol to provide structured messages. Also, library routines are available that can be used with a pipe to communicate via messages.

Figure 2.3 illustrates how pipes can be used in Linux to implement concurrent processing between processes A and B (executing PROC_A and PROC_B, respectively), where A performs some computation ("compute A1"), sends a value x to B, performs a second phase of computation, reads a value y from B, and then iterates. Meanwhile, B reads the value x, performs the first phase of

Figure 2.3
Linux Pipes

```
int A_to_B[2], B_to_A[2];
main(){
   pipe(A_to_B);
   pipe(B_to_A);
   if (fork()==0) { /* This is the first child process */
      close(A_to_B[0]);
      close(B_to_A[1]);
      execve("prog_A.out", ...);
      exit(1);  /* Error–terminate the child */
   }
   if (fork()==0) { /* This is the second child process */
      close(A_to_B[1]);
      close(B_to_A[0]);
      execve("prog_B.out", ...);
      exit(1);  /* Error–terminate the child */
   }
/* This is the parent process code */
   wait(...);
   wait(...);
}

proc_A(){
   while (TRUE) {
      <compute A1>;
      write(A_to_B[1], x, sizeof(int)); /* Use this pipe to send info */
      <compute A2>;
      read(B_to_A[0], y, sizeof(int)); /* Use this pipe to get info */
   }
}

proc_B(){
   while (TRUE) {
      read(A_to_B[0], x, sizeof(int)); /* Use this pipe to get info */
      <compute B1>;
      write(B_to_A[1], y, sizeof(int)); /* Use this pipe to send info */
      <compute B2>;
   }
}
```

computation, writes a value to A, performs more computation, and then iterates. A process that does not intend to use a pipe end should close so that end-of-file (EOF) conditions can be detected.

A *named pipe* can be used to allow unrelated processes to communicate with each other. Typically in pipes, the children inherit the pipe ends as open file descriptors. In named pipes, a process obtains a pipe end by using a string that is analogous to a filename but that is associated with a pipe. This allows any set of processes to exchange information by using a *public pipe* whose end names are filenames. When a process uses a named pipe, the pipe is a system-wide resource, potentially accessible by any process. Just as files must be managed so that they are not inadvertently shared among many processes at one time, named pipes must be managed, by using the file system commands.

Reading Multiple Input Streams

As with any file, the read end of a pipe, a file descriptor, or a socket can be configured, with an IOCTL() call, to use nonblocking semantics. After the call has been issued on the descriptor, a read on the stream returns immediately, with the error code set to EAGAIN. Also, read() will return a value of 0, thereby indicating that it did not read any information into the buffer. Alternatively, the program can determine whether read() succeeded by checking the length value to see if it is nonzero.

Figure 2.4 illustrates the use of the ioctl() to switch the read end of a pipe from its default blocking behavior to nonblocking behavior. You can apply this technique to any file descriptor, including stdin.

The select() command allows a process to poll all of its open input streams to determine which contain data. It then can execute a normal blocking read operation on any stream that contains data. Note that if multiple processes are reading the pipe and each uses a select(), then a race condition can result. If you decide to use this approach in your solution, use the man page to learn more about select().

Problem Statement

Part A

Write a C program that will act as a shell command line interpreter for the Linux kernel. Your shell program should use the same style as the Bourne shell for running programs. In particular, when the user types a line such as

identifier [identifier [identifier]]

Figure 2.4
Nonblocking read Example

```
#include       <sys/ioctl.h>
int errno;     /* For nonblocking read flag */
...
main() {
  int pipeID[2];
  ...
  pipe(pipeID);
/* Switch the read end of the pipe to the nonblocking mode */
  ioctl(pipeID[0], FIONBIO, &on);
  ...
  while(...) {
  /* Poll the read end of the pipe */
    read(pipeID[0], buffer, BUFLEN);
    if (errno !=EAGAIN){
    /* Incoming info available from the pipe—process it */
      ...
    } else {
    /* Check the pipe for input again later—do other things */
      ...
    }
  }
  ...
}
```

your shell should parse the command line to build argv. It should search the directory system (in the order specified by the PATH environment variable) for a file with the same name as the first identifier (which may be a relative filename or a full pathname). If the file is found, then it should be executed with the optional parameter list, as is done with sh. Use an execv() system call (rather than any of the execl() calls) to execute the file that contains the command. You will need to become familiar with the Linux fork()and wait() functions and the execv() family of system calls (a set of interfaces to the execve() system call).

Part B

Add functionality to the shell from Part A so that a user can use the "&" operator as a command terminator. A command terminated with "&" should be executed concurrently with the shell (rather than the shell's waiting for the child to terminate before it prompts the user for another command).

Part C

Modify your shell program from Part A so that the user can redirect the stdin or stdout file descriptors by using the "<" and ">" characters as filename pre-

fixes. Also, allow your user to use the pipe operator, "|", to execute two processes concurrently, with stdout from the first process being redirected as the stdin for the second process. Your solution may optionally handle only redirection or only pipes in one command line (rather than both options in the same command line).

Attacking the Problem

Organizing a Solution

The exercise introduction generally describes how a shell behaves. It also implicitly provides a plan of attack, summarized here. This plan describes several debugging versions that you can use for Part A, then apply to the other parts as required.

1. Organize the shell to initialize variables and then to perform an endless loop until stdin detects an EOF condition, such as a Ctrl-D character or an exit message. Develop a very simple version that prints the prompt character and then waits for the user to type a command. After it reads the command, it should print it to stdout.
2. Refine your simple shell so that it *parses* the command line. It should determine the strings on the command line and then put them into a char *argv[] array. Also, compute the value for int argc. In this debug version, print the contents of argc and argv[]. You can also detect an exit command that will terminate your shell.
3. In the next debug version, use argv[0] to find the executable file. In this version, simply print the filename.
   - Construct a simple version that can find only command files that are in the current directory.
   - Enhance your program so that it can find command files that are specified with an absolute pathname.
   - Enable your program to search directories according to the string that is stored in the shell PATH environment variable.
4. Create a child process to execute the command.

Part A

Here is a code skeleton for a shell program.

```
int main () {
  ...
```

```
  struct command_struct {
    char *name;
    char *argv[];
    ...
  } *command;
  ...
// Shell initialization
  ...

// Main loop
  while(stdin is not at EOF) {
    print_to_stdout(prompt_string);
    command_line = read_from_stdin();
  // Determine the command name, and construct the parameter list
    ...
  // Find the full pathname for the file
    ...
  // Launch the executable file with the specified parameters
    ...
  }

// Terminate the shell
  ...
}
```

Determining the Command Name and the Parameter List

You should recognize the argv name in the command_struct from writing C programs in your introductory programming classes (and from Exercise 1 in this manual). That is, if you write a program and want the shell to pass parameters (from the command line) to your program, then you declare the function prototype for your main program with a line like this:

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

The convention is that when your executable program (a.out) is executed, the shell builds an array of strings, argv, with argc entries in it. argv[0] points to a string that has the command name a.out, argv [1] points to a string that specifies the first parameter, argv [2] points to a string that specifies the second parameter, and so on. Your program, when executed, reads the argv array to get the strings and then applies whatever semantics it wants to interpret the strings.

For example, your program might be run with a command line of the form

```
a.out foo 100
```

so that when it begins execution, it will have argc set to 3, argv[0] will point to the string a.out, argv [1] will point to foo, and argv [2] will point to the string 100. Your program then can interpret the first parameter (argv [1]) as, say, a filename, and the second parameter (argv[2]) as, say, an integer record count. The shell would simply treat the first word on the command line as a filename and the remaining words as strings.

After your program reads the command line into a string, command_line, it parses the line so as to populate the command_struct fields (name and argv). The explanation in the introduction should suffice for you to design and implement code to parse the command line so that you have the name of the file containing the command in command->name and (a pointer to) the array of pointers to parameter strings in command->argv.

Finding the Full Pathname

The user might have provided a full pathname as the command name word or only a relative pathname that is to be bound according to the value of the PATH environment variable. A name that begins with a "/", "./", or "../" is an absolute pathname that can be used to launch the execution. Otherwise, your program must search each directory in the list specified by the PATH environment variable to find the relative pathname.

Launching the Command

The final step in executing the command is to fork a child process to execute the specified command and then to cause the child to execute that command. The following code skeleton will accomplish this.

```
if(fork() == 0) {
  // This is the child
  // Execute in the same environment as the parent
    execvp(full_pathname, command->argv, 0);

} else {
  // This is the parent; wait for the child to terminate
    wait(status);
}
```

Parts B and C

Getting general code is difficult when more than one special operator is used in one command line, so you should focus on doing the exercise only when “&”, “<”, “>”, or “|” is used and then, not more than one at a time.