A Simple Shell

This exercise can be solved on any UNIX system.

Starting with the code in Section 2.4, design and implement a simple, interactive shell program that prompts the user for a command, parses the command, and then executes it with a child process. In your solution you are required to use `execv()` instead of `execvp()`, which means that you will have to read the `PATH` environment variable, then search each directory in the `PATH` for the command file name that appears on the command line.

● Background ●

The OS exports its functionality as the system call interface. All non-OS software uses OS services by making function calls such as `read()` and `fork()` on the system call interface. Human users such as a batch computer operator or an interactive user also need to interact with the OS so they can run programs, inspect the collection of files, and so on. Should the OS provide a specialized human-computer interface just for this purpose? In modern computer systems, the OS does not include such an interface. Instead, there are one or more command line interpreter programs that use the conventional system call interface to invoke OS services, and which export an “operator’s console” to a user (see Figure 2.11). Since a command line interpreter is just an application program, programmers can create their own if they don’t like the ones delivered with the OS.

♦ FIGURE 2.11 The Shell Command Line Interpreter

The shell command line interpreter is an application program that uses the system call interface to implement an operator’s console. It exports a character-oriented human-computer interface, and uses the system call interface to invoke OS functions.
The original UNIX developers were the first to adopt this technique for constructing a command line interpreter [Ritchie and Thompson, 1974]. They called their command line interpreter the shell program, a name that has stuck (and is now used to refer to any program that provides a human–OS interface). The inspiration for the name is that the shell program provides a protective cover over the OS, much like a shell protects an oyster.

The simplest shell programs are text-based programs (more complex shell programs use graphics with point-and-select interfaces). A text-based shell assumes a screen display with a fixed number of lines (usually 25) and a fixed number of characters (usually 80) per line. The user interacts with the OS by typing a string of characters (terminated with the “enter” or “return” key) to the shell, and the OS responds by printing lines of characters back to the shell display.

When a user logs onto a system, a shell program is started to handle the interaction. Once the shell has initialized its data structures and is ready to start work, it clears the 25-line display, and then prints a prompt in the first few character positions on the first line. Sometimes the shell is configured to include the machine name as part of the prompt. My Linux development machine is named kiowa.cs.colorado.edu and I use the bash shell, so my shell prints

```
kiowa$
```

as its prompt string. (My BSD workstation uses the C shell, so its prompt is `pawnee%`.) 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 by an `enter` or `return` character (in UNIX, the character is represented internally by the `NEWLINE` character, `'\n'`). When the user enters a command line, the shell program makes the appropriate system calls to execute the command that appears on the command line.

Every shell has its own language syntax and semantics. In conventional UNIX shells a command line has the form

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

where the command to be executed is the first word in the command line and the remaining words are arguments expected by that command. As discussed in Section 2.4, the number of arguments depends on which command is being executed. For example, the directory listing command can be used with no arguments—by typing “`ls`,” or it may have arguments prefaced by the “`-`” character, as in “`ls -al`” where “a” and “1” are arguments. Each command uses its own syntax for interpreting an argument. For example, a C compiler command might look like

```
kiowa$ cc -g -o deviation -S main.c inout.c -lm a b c
```

where the arguments “`g`”, “`o`”, “`deviation`”, “`S`”, “`main.c`”, “`inout.c`”, and “`lm`” are all being passed as parameters to the C compiler, “`cc`.” That is, the specific command determines which of the arguments may be grouped (like the “`a`” and “`1`” in the `ls` command), which arguments must be preceded by a “`-`” symbol, whether the position of the argument is important, and so on.
The shell relies on an important convention to accomplish its task: The command is usually the name of a file that contains an executable program. For example, `ls` and `cc` are the names of files (stored in `/bin` on most UNIX-style machines). In a few cases, the command is not a file name, but is actually a command that is implemented within the shell; for example `cd` ("change directory") is usually implemented within the shell rather than in a file. Since the vast majority of the commands are implemented in files, think of the command as actually being a file name 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.

There is a long line of shell programs used with UNIX variants, including the original Bourne shell (`sh`), the C shell (`csh`) with its additional features over `sh`, the Korn shell, and so on, to the standard Linux shell (`bash`—meaning Bourne-Again shell). All of these shells follow a similar set of rules for command line syntax, although each has its own special features. The `cmd.exe` shell for Windows uses its own similar, but distinct, command language.

**Basic UNIX-style Shell Design**

A shell could use many different strategies to execute the user's computation. The basic approach used in modern shells is the one described in Section 2.4—to create a new process (or thread) to execute any new computation. For example, if a user decides to compile a program, the process interacting with the user creates a new child process. The first process then directs the new child process to execute the compiler program.

This idea of creating a new process to execute a computation may seem like overkill, but it has a very important characteristic. When the original process decides to execute a new computation, it protects itself from any fatal errors that might arise during that execution. If it did not use a child process to execute the command, a chain of fatal errors could cause the initial process to fail, thus crashing the entire machine.

The UNIX paradigm for executing commands is illustrated in Figure 2.12. Here, the shell has prompted the user with the `%` character and the user has typed "`grep first f3`." This command means the shell should create a child process and cause it to execute the `grep` string search program with parameters `first` and `f3`. (The semantics of `grep` are that the first string is to be interpreted as a search pattern and the second string is a filename.)

The Bourne shell is described in Ritchie and Thompson's original UNIX paper [Ritchie and Thompson, 1973]. The Bourne shell and others 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. When a user passes a command line to the shell, it is interpreted as a request to execute a program in the specified file—even if the file contains a program that the user wrote. That is, a programmer can write an ordinary C program, compile it, and then have the shell execute it just like it was a normal UNIX command. For example, you could write a C program in a file named `main.c`, then compile and execute it with shell commands like

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

The shell finds the `cc` command (the C compiler) in the `/bin` directory, and then passes it the string "`main.c`" when it creates a child process to execute the `cc` program. The C


\section*{FIGURE 2.12 The Shell Strategy}

The shell isolates itself from program failures by creating a child process to execute each command specified by the human user. The \texttt{grep} command is executed by a child of the process executing the shell.

\begin{verbatim}
% grep first f3
\end{verbatim}

Consider the detailed steps that a shell must take to accomplish its job:

\begin{itemize}
  \item **Printing a prompt.** There is a default prompt string, sometimes hardcoded into the shell, e.g., the single character string "$", "#", ">" or other. When the shell is started, it can look up the name of the machine on which it is running, and prepend this string name to the standard prompt character, for example, giving a prompt string such as "kiowa$". The shell can also be designed to print the current directory as part of the prompt, meaning that each time the user employs \texttt{cd} to change to a different directory, the prompt string is redefined. Once the prompt string is determined, the shell prints it to \texttt{stdout} whenever it is ready to accept a command line. For example, this function prints a prompt:

\begin{verbatim}
void printPrompt() {
  /* Build the prompt string to have the machine name,
   * current directory, or other desired information
   */
  promptString = ...;
  printf("%s ", promptString);
}
\end{verbatim}

  \item **Getting 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 blocked until the user types a command line in response to the prompt. When the command has been provided by the user (and terminated with a \texttt{NEWLINE} character), the command line string is returned to the shell.
\end{itemize}
void readCommand(char *buffer) {
    /* This code uses any set of I/O functions, such as those in
    * the stdio library to read the entire command line into
    * the buffer. This implementation is greatly simplified,
    * but it does the job.
    */
    gets(buffer);
}

- **Parsing the command.** This is described in the example in Section 2.4.

- **Finding the file.** The shell provides a set of *environment variables* for each user—this variable is first defined in the user's .login file, although it can be modified at any time with the set command. The PATH environment variable is an ordered list of absolute pathnames that specifies where the shell should search for command files. If the .login file has a line such as

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

the shell will first look in the current directory (since the first pathname is "." for the current directory), then in /bin, and finally in /usr/bin. If there is no file with the same name as the command (from the command line) in any of the specified directories, the shell responds to the user that it is unable to find the command. The solution needs to parse the PATH variable before it begins reading command lines. This is done with

```c
int parsePath(char *dirs[])
{
    /* This function reads the PATH variable for this
    * environment, then builds an array, dirs[], of the
    * directories in PATH
    */
    char *pathEnvVar;
    char *thePath;

    for(i=0; i<MAX_ARGS; i++)
    {
        dirs[i] = NULL;
        pathEnvVar = (char *) getenv("PATH");
        thePath = (char *) malloc(strlen(pathEnvVar) + 1);
        strcpy(thePath, pathEnvVar);

        /* Loop to parse thePath. Look for a ':'
        * delimiter between each path name.
        */
        ...
    }
}
```

The user may have provided a full pathname as the command name word, or only have provided a relative pathname that is to be bound according to the value of the PATH environment variable. If the name begins with a "/", then it is an absolute pathname that can be used to launch the execution. Otherwise, you will have to search each directory in the list specified by the PATH environment variable to find the relative pathname. Each time you read a command, you will need to see if there is an executable file in one of the directories specified by the PATH variable. The lookup() function is intended to serve that purpose:
char *lookupPath(char **argv, char **dir) {
    /* This function searches the directories identified by the dir
     * argument to see if argv[0] (the file name) appears there.
     * Allocate a new string, place the full path name in it, then
     * return the string.
     */
    char *result;
    char pName[MAX_PATH_LEN];

    // Check to see if file name is already an absolute path name
    if(*argv[0] == '/') {
        ...
    }

    // Look in PATH directories.
    // Use access() to see if the file is in a dir.
    for(i = 0; i < MAX.Paths; i++) {
        ...
    }

    // File name not found in any path variable
    fprintf(stderr, "%s: command not found\n", argv[0]);
    return NULL;
}

● ATTACKING THE PROBLEM ●

Begin your solution by reading the UNIX example in Section 2.4. You will have to rewrite
that code to support user interaction. Here is a header file, minishell.h, for your mini-
shell:

...  
#define LINE_LEN  80
#define MAX_ARGS  64
#define MAX_ARG_LEN 16
#define MAX_PATHS  64
#define MAX_PATH_LEN 96
#define WHITESPACE " .\t\n"

#ifndef NULL
#define NULL ...
#endif

struct command_t {
    char *name;
    int argc;
    char *argv[MAX_ARGS];
};

Here is the skeleton of a solution:
/* This is a very minimal shell. It finds an executable in the PATH, then loads it and executes it (using execv). Since it uses "." (dot) as a separator, it cannot handle file names like "minishell.h" */

#include ...
#include "minishell.h"

char *lookupPath(char **, char **);
int parseCommand(char *, struct command_t *);
int parsePath(char **);
void printPrompt();
void readCommand(char *);
...

int main() {
...

  /* Shell initialization */
  ...
  parsePath(pathv); /* Get directory paths from PATH */

  while(TRUE) {
    printPrompt();

    /* Read the command line and parse it */
    readCommand(commandLine);
    ...
    parseCommand(commandLine, &command);
    ...

    /* Get the full pathname for the file */
    command.name = lookupPath(command.argv, pathv);
    if(command.name == NULL) {
        /* "Report error */
        continue;
    }

    /* Create child and execute the command */
    ...

    /* Wait for the child to terminate */
    ...
  }

  /* Shell termination */
  ...
}