Assignment 01

Exercise: A Simple Shell

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(), which means that you will have to read the PATH environment, then search each directory in the PATH for the command file name that appears on the command line.

The code (i.e. source files, header files, Makefile) for this asignment should be sent as a zipped tar file to Engelbert Hubbers not later than Thursday March 4th. On Tuesday March 9th, the shells need to be demonstrated during the exercise hours. Because of the coming holiday, these dates may seem far away. Do not make the mistake to start too late.

This exercise is based upon Lab Exercise 2.1 from Gary Nutt's Operating Systems.

Goals

After this exercise you should:

- Understand how a command line shell works.
- Understand the commands fork, execv and wait.
- Feel comfortable compiling simple C-programs using a Makefile.

Background

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.

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. The number of arguments depends on the command which is being executed.

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 is usually implemeted 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 the cause the command to be executed using the parameters.

A shell could use many different strategies to execute the user's computation. However the basic approach used in modern shells is to create a new process to execute any new computation.

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 protect itself from 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.

Steps

These are the detailed steps any shell should take to accomplish its job (note that the code snippets are just examples; they might not suffice for this assignment, but they should give enough clues):

Printing a prompt. There is a default promptstring, sometimes hardcoded into the shell, e.g. > or other. When the shell is started it can look up the name of the machine on which it is running, and prepend it to the standard prompt character, for example, giving a prompt string such as zaagblad>. The shell can also be designed to print the current directory as part of the prompt, meaning that each time the user employs cd 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.

For example this function prints a prompt:

```
void printPrompt() {

/* Build the prompt string to have the machine name,

 * current directory, or other desired information.

 */

  promptString = ...;

  printf("%s ", promptString);

}
```

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 newline character, the command line string is returned to the shell.

```
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);
}
```

Note: do not use gets in your final code. Use man gets to see why.

Parsing the command. For this assignment your shell doesn't need to be able to handle pipes or redirections.

Finding the file. Before you search for the file you have to figure out whether the command needs to be implemented as an internal command or not.

If it is not an internal command, you will have to read the environment variable PATH. These environment variables are typically set when the shell is started, but can be modified on the commandline. The command you have to use for this depends on the shell you are using.

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 like:

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.

Your solution needs to parse the PATH variable before it begins reading command lines. This can de done with code like this:

```
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_PATHS; 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 PATH 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. This 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
  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;
}
```

Execute the command. A process in UNIX is created using the fork() system call. The parent-child example given in the Introductiecursus UNIX illustrates the idea. Look at these examples and try to understand them. For details of how fork() works, consult the man pages by typing man fork at the command prompt.

Code skeleton

It is good practice to use header files to define constants and types. In particular this means that you have to make decisions on the values of these constants. Here is an example minishell.h:

```
...
#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];
}
```

And here is a skeleton of a very simple shell. For instance, it doesn't distinguish between internal commands and file commands. Your solution needs to be more sophisticated!

```
/* 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 */
  ...
}
```