Write your own shell.

Submit to eLearning by 11/1 11:30 pm

- source code .c files
- makefile
- picture of your drawing of your state machine design: example of simple state machine

Reads a command from stdin and executes it until EOF.

- Use the lexical analyzer provided. shellex.zip (in project files)
- Use a state machine to keep track of what is coming from getNext()
- Use the two structs and the algorithm provided to execute the line shell

Your shell must handle

- command arguments
- < input redirection
- > stdout open for overwrite
- >> stdout open for append
- >& stderr overwrite
- >>& stderr append
- | pipes zero or more

```
Suggested structs for pipes and so forth, blindly copying this without
understanding and modifying for your use will not work.
This came from a working shell written in C++ and you are using C for
example.  This is one way that works, bash uses similar structs but a
totally different execute for example.

The Command Table
The Command Table is an array of  SimpleCommand structs.
A SimpleCommand struct
contains members for the command and arguments of a single entry in the
pipeline. The
parser will look also at the command line and determine if there is any
input or output
redirection based on symbols present in the command (i.e. < infile, or
> outfile).

a < in | b | c > out
remember when pipes only the first can have input redirection and only the
last can have output redirection so only two for whole line and can be in
the struct Command.
You can have just one stderr for the whole line stored in the struct
Command.

In SimpleCommand the first string in _arguments is the command string
also.
```

Here is an example of a command and the ***Command Table*** it generates:

command

ls -al | grep me > file1

Command Table

SimpleCommmand array:

0:	**ls**	**-al**	**NULL**
1:	**grep**	**me**	**NULL**

IO Redirection:

in: default	**out: file1**	**err: default**

```
// Command Data Structure
// Describes a simple command and arguments
struct SimpleCommand {
        // Available space for arguments currently preallocated
        int _numberOfAvailableArguments;
        // Number of arguments
        int _numberOfArguments;
        // Array of arguments
        char ** _arguments;

};
      struct SimpleCommand * simpleCommand();
        void insertArgument(struct SimpleCommand command, char * argument
); // or global *_currentSimpleCommand instead of parameter

// Describes a complete command with the multiple pipes if any
// and input/output redirection if any.
struct Command {
        int _numberOfAvailableSimpleCommands;
        int _numberOfSimpleCommands;
        SimpleCommand ** _simpleCommands;
        char * _outFile;
        char * _inputFile;
        char * _errFile;
        int _background;
};

// this uses global static to keep track of current commands perhaps not
the best way, works for objects not so well in C
        void print();
```

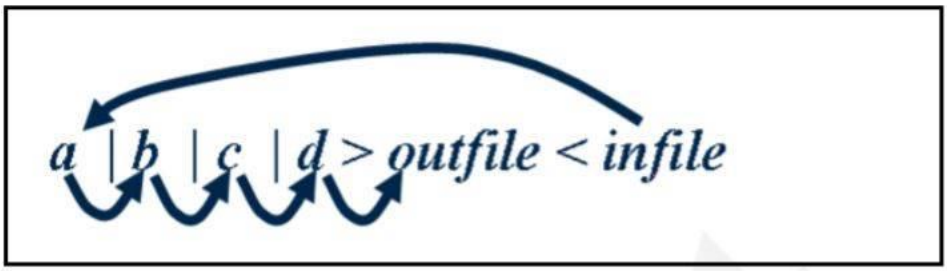
```
        void execute();
        void clear();
        struct Command * command(); // constructor
        void insertSimpleCommand( SimpleCommand * simpleCommand );
        static Command _currentCommand;
        static SimpleCommand *_currentSimpleCommand;
```

The constructor simpleCommand constructs a simple empty command.
The method insertArgument(char * argument) inserts a new argument into the SimpleCommand and enlarges the _arguments array if necessary. It also makes sure
that the last element is NULL since that is required for the exec() system call.

The constructor command() constructs and empty command that will be populated with the
method insertSimpleCommand(SimpleCommand * simpleCommand).
insertSimpleCommand also enlarges the array _simpleCommands if necessary. The variables _outFile, _inputFile, _errFile will be NULL if no redirection was
done, or the name of the file they are being redirected to.
The variables _currentCommand and _currentCommand are static variables, that is
there is only one for the whole class. These variables are used to build the Command and
Simple command during the parsing of the command.

****
the following is missing error checking and stderr redirection you must add, also you must change lines like
fdin = open(infile,O_READ .. to
open(_currentCommand.infile or _currentCommand->infile depending on how you write your code.

Pipe and Input/Output Redirection in Your Shell
The strategy for your shell is to have the parent process do all the piping and redirection
before forking the processes. In this way the children will inherit the redirection. The parent
needs to save input/output and restore it at the end. Stderr is the same for all processes



In this figure the process a sends the ouptut to pipe 1. Then b reads its input from pipe 1 and
sends its output to pipe 2 and so on. The last command d reads its input from pipe 3 and
send its output to outfile. The input from a comes from infile.
The following code show how to implement this redirection. Some error checking was
eliminated for simplicity.

```
execute(){
 2    //save in/out
 3    int tmpin=dup(0);
 4    int tmpout=dup(1);
 5
 6    //set the initial input
 7    int fdin;
 8    if (infile) {
 9      fdin = open(infile,O_READ);
10    }
11    else {
12      // Use default input
13      fdin=dup(tmpin);
14    }
15
16    int ret;
17    int fdout;
18    for(i=0;i<numsimplecommands; i++) {
19      //redirect input
20      dup2(fdin, 0);
21      close(fdin);
22      //setup output
23      if (i == numsimplecommands - 1){
24        // Last simple command
25        if(outfile){
26          fdout=open(outfile,â€¦â€¦);
27        }
28        else {
29          // Use default output
30          fdout=dup(tmpout);
31        }
32      }
33
34       else {
35          // Not last
36          //simple command
37          //create pipe
38          int fdpipe[2];
39          pipe(fdpipe);
40          fdout=fdpipe[1];
41          fdin=fdpipe[0];
42       }// if/else
43
44       // Redirect output
45       dup2(fdout,1);
```

```
46       close(fdout);
47
48       // Create child process
49       ret=fork();
50       if(ret==0) {
51         execvp(scmd[i].args[0], scmd[i].args);
52         perror(â€œexecvpâ€);
53         _exit(1);
54       }
55     } //  for
56
57     //restore in/out defaults
58     dup2(tmpin,0);
59     dup2(tmpout,1);
60     close(tmpin);
61     close(tmpout);
62
63     if (!background) {
64       // Wait for last command
65       waitpid(ret, NULL);
66     }
67
68  } // execute
```

Lines 3 and 4 save the current stdin and stdout into two new file descriptors using the dup()
function. This will allow at the end of execute() to restore the stdin and stdout the way it was at the beginning of execute(). The reason for this is that stdin and stdout (file descriptors 0
and 1) will be modified in the parent during the execution of the simple commands

```
 3    int tmpin=dup(0);
 4    int tmpout=dup(1);
```

Lines 6 to 14 check if there is input redirection file in the command table of the form
“command < infile”. If there is input redirection, then it will open the file in infile and save it in
fdin. Otherwise, if there is no input redirection, it will create a file descriptor that refers to the
default input. At the end of this block of instructions fdin will be a file descriptor that has the
input of the command line and that can be closed without affecting the parent shell program.

```
 6    //set the initial input
 7    int fdin;
 8    if (infile) {
 9      fdin = open(infile,O_READ);
10    }
11    else {
12      // Use default input
13      fdin=dup(tmpin);
14    }
```

Line 18 is the for loop that iterates over all the simple commands in the command table. This
for loop will create a process for every simple command and it will perform the pipe
connections.
Line 20 redirects the standard input to come from fdin. After this any read from stdin will
come from the file pointed by fdin. In the first iteration, the input of the first simple command
will come from fdin. fdin will be reassigned to a input pipe later in the loop. Line 21 will close
fdin since the file descriptor will no longer be needed. In general it is a good practice to close
file descriptors as long as they are not needed since there are only a few available (normally
256 by default) for every process.

```
16    int ret;
17    int fdout;
18    for(i=0;i<numsimplecommands; i++) {
19      //redirect input
20      dup2(fdin, 0);
21      close(fdin);
```

Line 23 checks if this iteration corresponds to the last simple command. If this is the case, it
will test in Line 25 if there is a output file redirection of the form “command > outfile” and open
outfile and assign it to fdout. Otherwise, in line 30 it will create a new file descriptor that
points to the default input. Lines 23 to 32 will make sure that fdout is a file descriptor for the
output in the last iteration.

```
23      //setup output
23      if (i == numsimplecommands1){
24        // Last simple command
25        if(outfile){
26          fdout=open(outfile,â€¦â€¦);
27        }
28        else {
29          // Use default output
30          fdout=dup(tmpout);
31        }
32      }
33
34      else {...
```

Lines 34 to 42 are executed for simple commands that are not the last one. For these simple
commands, the output will be a pipe and not a file. Lines 38 and 39 create a new pipe. The
new pipe. A pipe is a pair of file descriptors communicated through a buffer. Anything that is

written in file descriptor fdpipe[1] can be read from fdpipe[0]. IN lines 41 and 42 fdpipe[1] is
assigned to fdout and fdpipe[0] is assigned to fdin.
Line 41 fdin=fdpipe[0] may be the core of the implementation of pipes since it makes the
input fdin of the next simple command in the next iteration to come from fdpipe[0] of the
current simple command.

```
34       else {
35          // Not last
36          //simple command
37          //create pipe
38          int fdpipe[2];
39          pipe(fdpipe);
40          fdout=fdpipe[1];
41          fdin=fdpipe[0];
42       }// if/else
43
```

Lines 45 redirect the stdout to go to the file object pointed by fdout. After this line, the stdin
and stdout have been redirected to either a file or a pipe. Line 46 closes fdout that is no
longer needed.

```
44       // Redirect output
45       dup2(fdout,1);
46       close(fdout);
```

When the shell program is in line 48 the input and output redirections for the current simple
command are already set. Line 49 forks a new child process that will inherit the file
descriptors 0,1, and 2 that correspond to stdin, stdout, and stderr, that are redirected to either
the terminal, a file, or a pipe.
If there is no error in the process creation, line 51 calls the execvp() system call that loads the
executable for this simple command. If execvp succeeds it will not return. This is because a
new executable image has been loaded in the current process and the memory has been
overwritten, so there is nothing to return to.

```
48       // Create child process
49       ret=fork();
50       if(ret==0) {
51         execvp(scmd[i].args[0], scmd[i].args);
52         perror("execvp");
53         _exit(1);
54       }
55     } //  for
```

Line 55 is the end of the for loop that iterates over all the simple commands.

After the for loop executes, all the simple commands are running in their own process and
they are communicating using pipes. Since the stdin and stdout of the parent process has
been modified during the redirection, line 58 and 59 call dup2 to restore stdin and stdout to
the same file object that was saved in tmpin, and tmpout. Otherwise, the shell will obtain the
input from the last file the input was redirected to. Finally, lines 60 and 61 close the temporary
file descriptors that were used to save the stdin and stdout of the parent shell process.

```
57     //restore in/out defaults
58     dup2(tmpin,0);
59     dup2(tmpout,1);
60     close(tmpin);
61     close(tmpout);
```

If the “&” background character was not set in the command line, it means that the shell
parent process should wait for the last child process in the command to finish before printing
the shell prompt. If the “&” background character was set it means that the command line will
run asynchronously with the shell so the parent shell process will not wait for the command to
finish and it will print the prompt immediately. After this, the execution of the command is
done.

```
63     if (!background) {
64       // Wait for last command
65       waitpid(ret, NULL);
66     }
67
68  } // execute
```

The example above does not do standard error redirection(file descriptor 2). The semantics
of this shell should be that all simple commands will send the stderr to the same place. The
example given above can be modified to support stderr redirection.