Programming Assignment 1 Creating a SHELL Arthur
Due: Thursday, 9/29/05 (1 ½ weeks)

You are to develop a command line interpreter that behaves like a SHELL. Your SHELL will provide a prompt (>>) and then accept commands in the following format:

```
>>  program [arg1 arg2 ... argn] [< filename] [> filename] [&]
```

The following Shell semantics are representative variants of the command line:

```
>> abcd
```
execute abcd and wait until it completes before providing another prompt

```
>> abcd arg1 arg2 arg3
```
pass the 3 arguments to abcd at instantiation time

```
>> abcd &
```
execute abcd, but do not wait for abcd to finish before returning the “>>” prompt

```
>> abcd < filename
```
redirect STDIN for abcd

```
>> abcd > filename
```
redirect STDOUT for abcd

Clearly, combinations of the above are also appropriate. The most complex command line argument would be of the form:

```
>> abcd arg1 arg2 arg3 arg4 < Ifile > Ofile &
```

What semantics are associated with the above shell command?

Other Shell requirements include

- The shell will loop continuously to accept user commands. It will terminate when “exit” is typed.

- The shell will accept program path specification that start with a “/”, “./” or “../”, or simply the name of the executable, e.g., abcd. When the latter is encountered the Shell will search for the executable according the values in the environment variable PATH. This value must be retrieved using the UNIX system call “getenv()”

  Upon finding the executable, your Shell will print the full path name (starting with “/”) of where the executable is located.

Your Shell must also handle the case of the executable not being found. The UNIX system call “access()” or variant thereof, might be helpful here.

- Your Shell must use the UNIX system calls “fork()” and the “execv ()” to instantiate and reconfigure the new process.

- When dictated by the command line, your Shell must use the UNIX system call “waitpid(pid, NULL, 0)” to wait for completion of the newly instantiated process.

- When redirecting I/O a sequence of commands like might help:

```
close(1)
fid = open (filevbl, O_WRONLY | O_CREAT);
……
close(1);
```

  I will discuss how the above assists in redirecting I/O in class.

  If you are **creating** an output file through I/O redirection, the file must be created with access permissions minimally set to “read” for the owner.

Additional requirements and clarification will be made in class.

Assumptions that you can make:

- No command line will be longer than 100 characters
- At least one blank space will always precede and follow each argument and the “<” and the“>”; at least one blank space will precede the “&”
- All arguments to the program to be executed will be numeric

You will turn in to me **in class** (a) a listing of the program, (b) a listing of the output of your program. You will ftp to the “incoming” directory on arthur.cs.vt.edu a **tar’ed** and **gzipped** file containing: the program source code, a compiled (**static** executable) version of the source code, file(s) containing the output of your program, a makefile, and a README file containing your name, SSN and a description how to invoke your Shell, e.g., just type its name. Finally, name your gzipped file LnameFM##.tar.gz, e.g., arthurjd01.tar.gz.

For additional insights only, read the Lab Exercise at the end of Chapter 3.

In Class addt'l

(1) argv[0] should hold binary name, eg. a.out, not the full path name to it

(2) Print out all paths searched for any invocation form
    searched ….
    Searched….
    Found but not executable   or   found

(3) can use string manipulation functions in string.h… they only take char *.

CONSIDER adding filename completion "ls –f" TAB => gives all possible file names starting with f.