Creating a new process:

You need to create new process for each command first except for some inbuilt commands, it protects the main shell process from any errors that occur in the new command. Second, it allows for concurrency; that is, multiple commands can be started and allowed to execute simultaneously. You can create new process using fork(). It creates an exact copy of the running process. The new process created has its own address space. Wait() is used along with fork sometimes. Exec function is used to replace current process image with new image.

Built-in Commands:

Whenever your shell accepts a command, it should check whether the command is a **built-in command** or not. If it is, it should not be executed like other programs. No new process will be created for it. Instead, your shell will invoke your implementation of the built-in command. For example:

1. To implement the exit built-in command, you simply call exit(0); in your C program.
2. To implement "cd" (without arguments), you will simply use getenv("HOME") and chdir().
3. To implement pwd, you simply call getcwd().
4. To implement wait you simply call wait(NULL).
5. But to implement ls you have to use fork and exec to /bin/ls.

An example of built in commands:

% cd
% pwd
/afs/cs.wisc.edu/u/m/j/username

Redirection

Redirection is relatively easy to implement: just use **close()** on stdout and then **open()** on a file. With file descriptor, you can perform read and write to a file. Maybe in your life so far, you have only used **fopen()** , **fread()** , and **fwrite()** for reading and writing to a file. Unfortunately, these functions work on **FILE*** , which is more of a C library support.

To work on a file descriptor, you should use **open()** , **read()** , and **write()**system calls. These functions perform their work by using file descriptors. The idea of redirection is to make the stdout descriptor point to your output file descriptor. First of all, let's understand the STDOUT_FILENO file descriptor. When a command "ls -la /tmp" runs, the ls program prints its output to the screen. But obviously, the ls program does not know what a screen is. All it knows is that the screen is basically pointed by the STDOUT_FILENO file descriptor. In other words, you could rewrite printf("hi") in this way: write(STDOUT_FILENO, "hi", 2) .

Batch Mode

There is a batch mode beside an interactive mode in which you display a prompt and the user of the shell will type in one or more commands at the prompt. In batch mode, your shell is started by specifying a batch file on its command line; the batch file contains the same list of commands as you would have typed in the interactive mode.

In batch mode, you should **not** display a prompt. In both interactive and batch mode, your shell should terminates when it sees the exit command on a line or reaches the end of the input stream (i.e., the end of the batch file).

The command line arguments to your shell are to be interpreted as follows:

batchFile: an optional argument. If present, your shell will read each line of the batchFile for commands to be executed. If not present or readable, you should print the one and only error message.

Implementing the batch mode should be very straightforward if your shell code is nicely structured. The batch file basically contains the same exact lines that you would have typed interactively in your shell. For example, if in the interactive mode, you test your program with these inputs:

```
% ./mysh
mysh> ls
some output printed here
mysh> ls > /tmp/ls-out
some output printed here
mysh> notACommand
some error printed here
```

then you could cut your testing time by putting the same input lines to a batch file (for example myBatchFile):

```
ls
ls > /tmp/ls-out
notACommand
```

and run your shell in batch mode:

```
prompt> ./mysh myBatchFile
```

Output should be printed on terminal. Use write instead of printf to write back to terminal. This function is used like this:

```
write(STDOUT_FILENO, cmdline, strlen(cmdline));
```

Some useful functions:

Parsing: For reading lines of input, you may want to look at **fgets().** To open a file and get a handle with type **FILE *** , look into **fopen().** Be sure to check the return code of these routines for errors! (If you see an error, the routine **perror()** is useful for displaying the problem. *But do not print the error message from perror() to the screen. You should only print the one and only error message that I specified above*). You may find the **strtok()**routine useful for parsing the command line (i.e., for extracting the arguments within a command separated by whitespace or a tab or ...). Some have found **strchr()** useful as well.

Executing Commands: Look into **fork** , **execvp** , and **wait/waitpid** . See the UNIX man pages for these functions.

You will note that there are a variety of commands in the exec family; for this project, you must use **execvp** . Remember that if execvp() is successful, it will not return; if it does return, there was an error (e.g., the command does not exist). The most challenging part is getting the arguments correctly specified. The first argument specifies the program that should be executed, with the full path specified; this is straight-forward. The second argument, char *argv[]matches those that the program sees in its function prototype:

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

Note that this argument is an array of strings, or an array of pointers to characters. For example, if you invoke a program with:

foo 205 535 then argv[0] = "foo", argv[1] = "205" and argv[2] = "535".

Important: the list of arguments must be terminated with a NULL pointer; that is, argv[3] = NULL.

Error Handling:

There is a difference between errors that your shell catches and those that the program catches. Your shell should catch all the syntax errors specified in this project page. If the syntax of the command looks perfect, you simply run the specified program. If there is any program-related errors (e.g. invalid arguments to ls when you run it, for example), let the program prints its specific error messages in any manner it desires (e.g. could be stdout or stderr).

MakeFile:

Since you will have more than one files in which your function will be placed but your shell should run by running one file only. This can be done by combining your object files. For example, if you have 3 object files they can be combined into one executable using:

```
mysh: f1.o  f2.o  f3.o
        gcc –o mysh f1.o  f2.o  f3.o
```