Purpose:

The purpose of this project is to create a shell using system commands in which we send the commands to our MINIX OS..

Available Resources:

For this assignment we are given two source files lex.c and shell.c. We will be modifying the shell.c file to have a working shell. We were also given handler sample code from ecommons which contains useful code for learning about how to deal with waiting for processes.

Design:

The design implementation for the shell is to get input from standard input and parse to see what we need to send to MINIX exactly. After parsing the input, we create a child process using the fork system call function. We know that pid given from fork will be 0 if we are looking at a child process and non zero if we aren't. In the new process created, we use the system call execvp to pass the arguments to minix to execute the shell command along with its arguments. In the parent process we wait for the new process to complete with the wait system call, which is done using waitpid. Then we start processing the next command passed to the shell. It's important that we deal with zombie processes with the waitpid system call. Before getting another command, a while loop runs using the waitpid function as its parameter [while (waitpid(-1,&status,WNOHANG))]. The -1 parameter indicates that we are listening to any signal and the WNOHANG parameter indicates that we don't hang if waitpid does not receive a signal. Using this system call we are able to continuously listen and take care of zombie processes without stalling the program

Design of functions:

Our plan is to predominantly use the infinite while loop in the main for the structure of our code. Initially, we parse the command line to figure out what exactly we are looking at. We have to first check to see if the command given is "exit" or NULL in which the shell quits or reprints the prompt respectively. Afterwards, we need to check where all of our important symbols are, this is done in a for loop that checks each argument to see if it is equal to one of these characters: &, <, or >. For each of the characters, we set an appropriate flag accordingly. This way, we can check for each flag after we fork the process. With the exception of a bit of error handling, forking the process comes directly afterwards and the returned value is saved inside of pid. We then check to see if pid is either 0 or not. In the case that it is, we check each flag if they have been set. It could be that all of them are set at once so we have to make sure to check each of the flags individually. For the "&", we call SIGNAL with the parameters SIGCHLD and SIG_IGN. This system call sends a SIGCHLD signal to the parent process and tells the waiting parent process to ignore the signal sent creating a background process. For either of the arrows, we would attempt to open the file and throw an error if we can't, then we redirect in the direction of the arrow. When we use the file as an input we must make sure that we set freopen to read and

close stdin. When using a file for output we need to write and close stdout. While pid is still equal to 0, but after we have done all of our flag checks, we then call execvp and throw an error if it returns, which it shouldn't. Also for each case, we must write over the character (<,>, or &) with a null terminating character so that we can send only the comman to the MINIX OS If pid is non-zero, we check to see if the '&' flag is on. If the flag is on nothing needs to be done, else if the flag is not on we wait for the child process to finish executing before retrieving more commands.

New variables:
int pid : variable that stores the return status of fork()
int input_redirect: flag for determining whether to read from stdin or a file
int output_redirect: flag for determining whether to write to stdout or a file
int background_process: flag for determining if a process should execute in the background
char *input_file: array to store the input file name if input_redirect flag is set
char *output_file: array to store the output file name if output_redirect flag is set
int size: variable to store size of the argument list

Testing:

Testing was an incremental process and followed the timeline of the various requirements of the assignment. All testing was done in MINIX using a makefile and standard UNIX commands. The first step was to build an internal exit command and test if it worked when we ran the shell and attempted to exit.

Secondly we tested to see if commands with and without arguments could execute properly. Both these requirements could be handled with the same code and did not require separate tests.

Next we tested background processes. This required us to also see if non-background processes blocked until the process finished. We created a crunch function that counted from 1 to 4 million and tested it as a foreground process as well as background process.

Fourth, we tested I/O redirection which required the use of all our flags. The beginning tests were simple, only using one file redirect at a time. After we got both working, we tested to see if multiple redirects would work.

Finally after getting all six requirements to work independently, we ran various tests to see if multiple requirements could be handled simultaneously.