# UNIX Shell Project

Assigned : February 17, 2021 2:00pm Due: March 31, 2021 2:00pm

**No Extensions**

The purpose of this assignment is to reinforce, through practical application, the concepts that we are studying in CMPE320 Operating Systems, in particular, process forking, execution, and managing inter-process communication. Unfortunately, this also means we need to exercise things like string parsing, memory allocation, and linked lists. Read this through and thoroughly before coding. I’ve greatly expanded the descriptions of what is needed, added a bunch of sample code and what I hope to be helpful tips throughout the document.

## Requirement 1 - Basic Functionality

50 points

Any shell has to serve as a mechanism to read commands and manage the execution of those commands. Ultimately, your shell will need to read commands as strings, *parse* them into their component parts, and then manage the execution of those parts.

For example, if your shell reads the text “find . -name “first prj” /”, then your shell will need to split up the command into its basic parts, and prepare it for the one of the “exec()” functions, shown in Figure 1. Note that there are really two families here, the “execl” family that takes a *variadic* argument list. *Variadic* simply means a variable number of arguments. This is appropriate when you know the arguments are *compile time*. The “execv” family is looking for an array of arguments, which is where you’ll need to start.

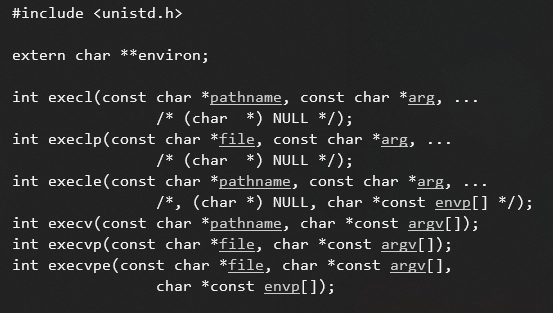


Figure 1 Standard Exec() functions in Linux

So, using the “find” command, you’ll need to split that up into an array of strings as follow. Note that the text inside the double quotes is turned in one argument, no matter what is inside of it:

Table 1 Parsed command line

|  |  |
| --- | --- |
| Argument | Pointer to Text |
| 0 | find |
| 1 | . |
| 2 | -name |
| 3 | first prj |
| 4 | / |
| 5 | NULL |

There are three things to notice in Table 1. First, the command itself is always the first command line argument. This is true in any UNIX shell. If you run any C program, the arguments to main include int argc and char \*\*argv. Because the command name is always the first argument, then argc >= 1, and printing “argv[0]” it will be the name of that command. The second is that the double quotes are used to create a single argument that contains spaces or other special characters. The third is that the argument list is terminated by a NULL pointer.

We’ll look at the parser more in the next section.

Finally, recall that any exec() function is almost always preceded by a fork(), and that the child is what actually performs the exec(). The parent will wait() for the child to complete.

## Parsing Command Lines

This is probably the most complex, intricate, and tedious part of the whole project. This unit of your project has to take a command line as a string of characters and turn it into something that will allow the rest of the parts of your project to perform their tasks. What makes this task even more difficult is the relatively wide variety of input combinations that are possible.

My strongest recommendation to your group is to implement the parser as a *state machine*. In this arrangement, you write your parser to look at one character of the input string at a time, and then, based on the current state, decide what to do next.

For example, suppose we wanted to find all numbers in a list separated by spaces. We know that that any string of digits will be one “number.” So, we can draw this graphically (see Figure 3):



Figure 3 Simple State Machine

The interpretation here is that every time we read a character, we consider what state we are in, and then decide what state to go to next. For example, if we are in the “start” state and read a series of white space (spaces & tabs), then we just eat them up (yum). As soon as we find the first digit, we take the edge to the “digit” state, and keep eating up digits until we hit more white space.

We can further define the state machine with some actions. For example:

|  |  |  |
| --- | --- | --- |
| Current State | Next State | Action |
| WS | DIGIT | Record the position of this character in the string as “start” |
| DIGIT | WS | Use the position as the end of the number, “strndup” from the starting position for this number of characters to “number”, and then call “atoi” to convert it to a number, and then add it to an output list. |

### Unknown Number of Events

One of the tricks with parsing is that we don’t know ahead of time how many “things” we can end up with. If you look at the action description above, I let slip the ugly cat out of the bag – “add it to a list.” Since arrays are fixed in size, they aren’t convenient for this task. Instead, we could create a linked list.

struct number {  
 int number;  
 char \*original\_text;  
 struct list\_head list;  
};

LIST\_HEAD(numbers);  
  
struct number \*curr = malloc(sizeof(struct number));  
curr->original\_text = strndup(string, start, (end-start));  
curr->number = atoi(curr->original\_text);  
list\_add(&curr->list, &numbers);

This is the code that was explored in a prior homework, and if you’re still struggling, you need to reach out to me and we can talk about it.

Of course, one of the issues with this is that as your parser finds things it keeps gobbling up memory. This is fine, but when you are finished with the list of values that you make sure you release all of the memory, your program will fail its *valgrind* tests. Of course, this also means you can use *valgrind* to help you spot places where you need to release the memory you’ve allocated.

The following example shows how to delete all of the elements of a linked list, first removing the node from the list, and then free()-ing any allocated memory.

while(! list\_empty(&numbers) {  
 struct list\_head \*curr = numbers.next;  
 struct number \*entry = list\_entry(curr, struct number, list);  
 list\_del(curr);  
 free(entry->original\_text);  
 free(entry);  
}

Using linked lists like this will enable your parser to handle any number of items. Then, you can write code that *counts* the number of list nodes, *malloc*’s that number of items, and finally loops through each item in the list and copies it to the allocated array.

Of course, when it comes to the *argument list* to the execute function, you’ll want to allocate one extra pointer, and make sure its last one is null.

### Allocating Memory

This brings us to the next topic. By now, all of you should have seen *malloc* and *free*. But, there is a third allocation function:

void \*calloc(size\_t nmemb, size\_t size)

This function takes the number of members and the size of each member allocates the chunk of memory to hold it from the heap, just like malloc. The big differences is that calloc() zeroes out the memory that it allocates, while malloc does not.

### Parser Tedium

My parser worked out to 585 lines (including comments and blank lines). Much of that was just small tedious code derived from the state machine graph I sketched out when I started the project. Stuff like “if (is\_redirect(ch)) state = REDIR”. I tell you this, because the parser ended up being over 1/3 of the entire project’s code (1545 lines). I probably over-engineered it, I usually do. But, there were some things you probably want to do in your own solution.

### Parser Helpful Hints

The first hint has to do with the pipe option. Supporting the UNIX “pipe” operator means running more than one command. I started writing the parser, had basic stuff working, when back to implement pipes, and realized I did it all wrong. I deleted everything and started over.

What I realized later was the pipes effectively turn one command line into multiple sub-commands. In other words, there is a many-to-one relationship between the text string you are parsing the number of commands you may find in the command line. So, now I split the one command line into a linked list of “sub command lines” using the “|” symbol. I don’t store the pipe symbol again after that. After parsing, I have a list of commands to execute. I presume that those commands are connected together by pipes – the output of one command is the input to the next command. No reason to store the pipes. *However, this means that a pipe symbol appearing in a quoted string is no longer allowed.* None of my test cases will generate this type of error.

Next, I realized that for each “sub command line” there is a possible many to one relationship with “tokens.” In this case, a token is just any word from the parser. There are three types of tokens I found, “normal”, “redirect,” and “file name.” I ended up coding the parser to create another linked list of tokens as in the following data structures:

enum token\_types {  
 TOKEN\_NORMAL, TOKEN\_REDIR, TOKEN\_FNAME  
};  
  
struct tokens {  
 char \*token\_text;  
 enum token\_types;  
 struct list\_head list;  
};

After parsing the tokens from the string, then I could build a data structure that contained the information for the execution phase.

### Playing the Data Structures Game

After the parser creates the list of tokens, its job is done. I wrote another function that takes that list and turns it into a representation of a command. This means I played the “data structure game,” where I spent time thinking about what information does the execution unit need, and how would it be best represented there. For example, its clear it will need:

* The command / path name for the call to execv()
* The argument array for the call to execv()

So, the function uses the information the linked list of tokens to fill in these fields of a struct. But there were other things that were needed, so they went into this structure too. Lets call it a “struct command”. Notice that all of the tokens in the tokens list for a sub-command got pulled into one command.

Then, I realized that with multiple sub-commands, one command line of text can turn into an linked list of “struct command” entries.

### Testing, Testing, Testing

Testing throughout the development cycle is all but required to ensure success. Testing in C language is a little more tricky than in other languages like Java. The test-code and the code-under-test both have to be compiled together.

This is where its is extremely important to structure your project into separate “.c” and “.h” files for each of the different functional areas / testing areas.

You can then create a “test” unit that exercises the code in each of the different sub-parts of your project. For example, when I was finishing up the parser, I developed some tests like:

int test\_split1( )  
{

char \*cmd = “test one two three”  
 char \*expected[] = { “test”, “one”, “two”, three”;

return run\_test(cmd, expected);  
}

Not only did I keep chipping away at the low-level parser until it was totally working, but then I was also able to *valgrind* the test code and make sure it didn’t leak memory. Testing this small portion of the project was crucial to success; and it let me focus my debugging efforts on a few lines of a code at a time.

## Requirement 2 - Shell Redirection

20 points

After the ability to read commands, the next one is to read commands that include the “redirection” operators:

* command > output   
  Runs command, but its “stdout” is directed to an output file. Prior to writing data, the output file is created if it doesn’t exist, and is truncated to a zero-length file if it does.
* command >> output  
  Runs command, but its “stdout” is directed to an output file. Prior to writing data, the output fiile is created if it doesn’t exist, and it *appended* to if it already does.
* command < input  
  Runs command, but its “stdin” is directed to come from an input file. Prior to running the program, the file is opened. If it doesn’t exist, the command is not run.

This part can seem like magic. First, remember that you’ve already seen this in homework. Second, your shell only has to open the file and “close” / “dup” so that the child process’s standard in / out are connected to the file rather than your terminal. Your shell doesn’t read or write these files, it just opens them. When the command that is being executed runs it inherits the connection of standard in/out, so if the child redirects them, the executed process inherits them. Then, when the executed command performs a “read” to read from standard input, or a “write” to standard output, that reads and writes to the files you’ve opened. Your shell has nothing to do with it after that. You don’t even need to close them, the executed command will do that.

### Parsing Redirection

Your parser has to read these values from the command line and somehow communicate that information (play the data structures game).

Your parser should also enforce the rule that standard input can only be directed once, and standard output can only be redirected once. Any violation is an illegal or malformed command line and should not be executed.

For example:

|  |  |
| --- | --- |
| ls -l > output | Valid – redirects standard out to the output file |
| cat < input > output | Valid – redirects standard input to the input file, and standard out to the output file. |
| ls -l > output > output2 | Illegal, two standard outs |
| ls -l > output >> output2 | Illegal, still two standard outs |

## Requirement 3 - Internal Commands

30 points

UNIX shells support commands that allow you to interact with the shell’s internal state, and for which no external process can “see.” These internal commands are handled by the shell itself, and not through the usual “fork/exec” mechanism.

You will need to handle the following internal commands:

|  |  |
| --- | --- |
| Command / Format | Purpose |
| setenv name value | Sets the environment variable with *name* to the indicated value. |
| getenv | Gets the entire environment and displays it to standard out. |
| getenv name | Get the value of the environment variable named *name* and display it to standard out. |
| unsetenv name | Unset / delete the environment variable named *name* |
| cd | Execute the chdir() system call to enter the “home” directory as stored in the HOME environment variable |
| cd pathname | Execute the chdir() system call to enter the directory indicated by pathname |
| pwd | Execute the “getcwd()” system call to get the current working directory and print it |
| exit | Exit the shell – don’t just call exit(), but perform memory clean-up so you can *valgrind* your shell code. |

Throughout the years I’ve used this or similar projects, I’ve seen all manner of “interesting” attempts to implement this. My recommendation is that you consider using *function pointers* and *table driven code*.

### Internal Commands Table

This is not a requirement, but highly recommended. Using table driven code makes this easy. First, declare a data type to hold the information you need:

typedef struct internal {   
 const char \*name;  
 int (\*handler)(commandline\_t \*cmd);  
} internal\_t;

Then, declare the functions. Note, this could be in a “.h”, and a declaration isn’t a definition.

static int handle\_setenv(commandline\_t \*cmd);  
static int handle\_getenv(commandline\_t \*cmd);  
static int handle\_unsetenv(commandline\_t \*cmd);

Then, declare the table of all known internal commands:

internal\_t internal\_cmds[] = {

{ .name = "setenv" , .handler = handle\_setenv },

{ .name = "getenv", .handler = handle\_getenv },

{ .name = "unsetenv", .handler = handle\_unsetenv },

0

};

And, then, you can write code that searches the table. For example:

int handle\_internal(commandline\_t \*cmd)

{

int i = 0;

while (internal\_cmds[i].name != 0) {

if (!strcmp(internal\_cmds[i].name, cmd->command)) {

return internal\_cmds[i].handler(cmd);

}

i++;

}

return 0;

}

What this code does it iterate over the table entries until it finds a string match. When it does, the “return” line actually *calls* the handler function with the given command structure. Adding/removing internal commands means just tweaking the table. Nothing else changes.

## Requirement 4 – Handling Pipes

20 points

As mentioned previously, commands can be “pipelined” together, such that a command like:

ls -l | grep “\*.c” | less

Means forking and executing three times. But it also requires creating *two pipes* not *three*. One pipe connected the output of ls to the input to grep. Another pipe connecting the output of grep to the input of less.

Commands that are pipelined cannot have the same channel redirected. For example, the following is invalid:

ls -l > output | grep “\*.c”

but

ls -l | grep “\*.c” > output

is valid (the output of grep wasn’t pipelined only its input).

### Helpful Hint

Remember the discussion during parsing about “sub commands”? If you split a command line into sub-command lines by the pipe operator, then you should *parse* out each individual command. You’ll just need to play the data structures game for how do you tell the *execution unit* that you want to pipe instead of redirect (or take the default).

## Requirement 5 – Handling Execution Environment

The execution *environment* consists of a set of variables and values. These are setup for you when you login. For example, your home directory is set in a variable named HOME.

HOME=/home/aa1234

As your change environment variables your shell changes an *internal* environment that it will use to execute future programs. It does not change its environment (or any process above it). Thus, you will need to manage a set of internal environment variables and use them when you execute a process.

### Accessing the Original Environment

The next statement makes it seem like we, the Faculty, have been hiding forbidden knowledge from you. Like we’ve been pulling a giant *Truman Show.*[[1]](#footnote-1)Here goes. The *real* way to write the main task is:

int main(int argc, char \*\*argv, char \*\*envp)

or

int main(int argc, char \*argv[], char \*envp[])

Different syntax for the same result. The argv array is a list of pointers to argc number of strings, one for each argument. The envp array is a list of pointers to strings, one for each environment variable, where the last entry in the array is a NULL.

Consider the following snippet of code in Figure 4 that walks the environment array.

Figure 4 – Walking the environment array

void show\_env(char \*\*envp)  
{  
 int i = 0;  
 while (envp[i] != NULL) {  
 printf(“%s\n”, envp[i]);  
 i++;  
 }   
}

### Saving the Environment for Future Generations

When executing your children, both the execl and execv functions support a form that passes a pointer to an environment array. Thus, to make your own environment for an execution, you can use a snippet similar to the code in Figure 5:

char \*\* make\_env( )  
{  
 char \*\*envp = calloc(4, sizeof(char \*));  
 envp[0] = “HOME=/tmp/bob”;  
 envp[1] = “PATH=/usr/bin:/bin:/sbin”;  
 envp[2] = “USER=bob”;  
 envp[3] = NULL;  
 return envp;  
}

Figure 5 Making an Environment

### Recommendation

The recommendation is to make a function that reads the environment from the envp into a linked list. Implement the *setenv*, *getenv*, and *unsetenv* programs to add/modify, retrieve, and delete entries from that linked list. Finally, make a function that counts the *length* members of the linked list, allocates an array of *length+1* character pointers, then walks the linked list and *strdup* or *malloc/strcpy* each string from the linked list into the new array, and then returns the array. The child can then *exec* using that array.

## Requirement 6 – The Command Prompt

5 points

UNIX shells use the “PS1” environment variable as the prompt that gets displayed to you on the command line. You’ll need to use the same logic. You’ll read your shell’s *internal* environment to get the current value of PS1 each time you display a prompt.

You’ll observe that most shell prompts don’t include a trailing newline (although they could). On the other hand, whenever you use the “printf” function, if you *don’t* have a new line, you may not see the output right away. This is due to *buffering*. Its simply not efficient to write individual characters to the output screen.

So, if you want the output to appear without waiting for the buffer to fill, you can “flush” the buffer. You’ll need to do this to make your prompt appear:

printf(“%s”, env\_get(“PS1”));  
fflush(stdout)

## Requirement 7 – Handling Startup Run-Commands

10 points

All UNIX shells look for a startup script and run the commands contained therein. For example, the *Bourne Again Shell(bash)* looks for .profile and .bashrc files on startup. Shells look into system administrator controlled locations (e.g. /etc), and in your home directory (HOME environment variable). This allows you to save commands that should be executed every time you run the shell (i.e. login).

You will do something similar for the Shippensburg University Shell (*sush*). Putting the file in the HOME directory makes it difficult to test. Instead of using this well-known environment variable, we’ll look for an environment variable SUSHHOME. If it is (check with your getenv), then we’ll look in the directory it points for a file named “.sushrc”. It will check the file’s status (see stat system call), and if it is not readable and executable it will ignore it.

The state system call returns the permissions, and then you can use a check like:

if (sb.st\_mode & S\_IRUSR) && (sb.st\_mode & S\_IXUSR)

To decide if the file is readable and executable to the owner. Then, simply open the file for reading, read the file, line by line, and execute each line – just like you would in part 1 if you typed them from the keyboard. In fact, if you’ve done this right, you *could* have a “read commands from file” function that goes into a loop to read strings from the file, calls the parser, and executes them; and closes the file when its finished. You can have another function that “reads commands from the user”, that goes into a loop that “prints a prompt, reads string from stdin, calls the parser, and executes them.

## Requirement 8 – Job Queue

20 points

One of the unique features of the *Shippensburg Shell* is that it will support running commands in the background, one at a time like a printer.

To support this, there are several additional *internal commands*:

|  |  |
| --- | --- |
| Command / Format | Purpose |
| queue *command line* | Enqueue’s the command line for processing. |
| status | Reports the status of jobs in the job queue |
| output *n* | Shows the output of the previously queued, and completed job *n* |
| cancel *n* | Cancels job *n* from the queue. If it is currently running it sends it a “SIGTERM” signal using the kill() system call. |

### Details

The queue command will *enqueue* a job for processing in the background. The *standard input* channel is closed. The job’s standard output is directed to a temporary file created in the “/tmp” directory (see the *mkstemp* library function). No other redirection / pipelining is supported here, but commands can have command line arguments, so “ls -l” is OK, “ls -l > output” is not.

When submitting a job to the job queue, if there is already a job running, then this task will be queued and wait for its turn. If there is no job running, then this job will be released from the job queue.

When the job is finished and exits, the shell will receive a *SIGCHLD* signal alerting it that one of its child processes has finished. Of course, a normal child completion will also trigger SIGCHLD. To receive these signals, you need to create a signal handler and then invoke the “signal” system call.

Your signal handler should scan through the job queue, and use the “waitpid” system call with the WNOHANG option to check the status on each task. Any task that has finished should be marked as completed. If the job that was running is finished, the next job in the job queue should be released and executed like other shell commands.

The output command will show the contents of the temporary file for a completed job. For this, just read the contents of the file to the screen. Then, you can close the file and remove the job from the job queue.

### Example

>queue ls -l /usr

>queue ps -eaf

>queue touch /tmp/bob

>status

0 is complete

1 is complete

2 is complete

>output 0

… <contents of the ls are snipped>

>status

1 is complete

2 is complete

## Requirement 9 – Consistent Error Messages

10 points

Because we will be using the autograder, any error messages must be consistent between projects. To help you, I went through my solution code and extracted every message and put it into a “.h” file (see next page).

Any of the “ERROR\_” messages shall be printed as an error message:

fprintf(stderr, ERROR\_SETENV\_ARG);

Some of these take arguments, such as :

#define ERROR\_INVALID\_CMD "Error could not execute : %s\n" // strerror

What is expected is included as a comment. So, for this type of error message:

fprintf(stderr, ERROR\_INVALID\_CMD, strerror(errno));

Check out the manual entry for the strerror function, but in short, it will take whatever system generated error number and print it as a string in the current language of the user.

Aside from the printing of the prompts, there are no other strings that my shell, or yours, shall display. Further, be careful to emit *messages* to standard out, and *errors* to standard error.

#ifndef MSGS\_H

#define MSGS\_H

// if its an error, then

// fprintf(stderr, ERROR\_SETENV\_ARG, ... )

//

// if its just a message then

// printf(MSG\_STATUS,...)

//

// whats expected is shown after

#define ERROR\_SETENV\_ARG "Error - setenv takes two arguments\n"

#define ERROR\_UNSETENV\_ARG "Error - unsetenv takes one argument\n"

#define ERROR\_GETENV\_ARG "Error - getenv takes 0 or 1 arguments\n"

#define ERROR\_CD\_ARG "Error - cd takes one argument\n"  
#define ERROR\_CD\_NOHOME "Error - cd no home directory\n"

#define ERROR\_PWD\_ARG "Error - pwd takes no arguments\n"

#define ERROR\_EXIT\_ARG "Rrror - exit takes no arguments\n"

#define ERROR\_GETENV\_INVALID "Error - getenv unknown variable %s\n" // variable name

#define ERROR\_QUEUE\_ARG "Error - queue requires at least two arguments\n"

#define ERROR\_OUTPUT\_ARG "Error - output takes one argument\n"

#define ERROR\_OUTPUT\_QUEUED "Error - task %d is still queued." // task # 0, 1, ...

#define ERROR\_OUTPUT\_RUNNING "Error - task %d is still running\n" // task #

#define ERROR\_STATUS\_ARG "Error - status takes 0 arguments\n"

#define MSG\_STATUS\_QUEUED "%d - is queued\n" // task #

#define MSG\_STATUS\_RUNNING "%d is running as pid %d\n" // task #

#define MSG\_STATUS\_COMPLETE "%d is complete\n" // task #

#define ERROR\_CANCEL\_ARG "Error - cancel takes one argument\n"

#define MSG\_CANCEL\_OK "%d is canceled\n" // task #

#define MSG\_CANCEL\_KILL "%d sending kill signal to pid %d\n" // task #, pid\_t

#define ERROR\_CANCEL\_DONE "%d is already finished, use output %d to show results\n"   
// task #, task #

#define ERROR\_EXEC\_INFILE "Error - could not open input file : %s\n"   
// strerror(errno)

#define ERROR\_EXEC\_OUTFILE "Error - could not open output file : %s\n"   
// strerror(errno)  
#define ERROR\_EXEC\_APPEND "Error - could not open output file : %s\n"   
// strerror(errno)  
#define ERROR\_EXEC\_FAILED "Error - could not execute : %s\n"   
// strerror(errno)

#define ERROR\_INVALID\_CMD "Error could not execute : %s\n"   
// strerror(errno)  
#define ERROR\_INVALID\_CMDLINE "Error - malformed command line.\n"

#endif

## Requirement 9 – Building Your Project & Deliverables

10 points

When you submit your project, you shall submit all of your “.c” and “.h” files, and a Makefile. The autograder will scourge away any extra files and report that as a failure. The autograder will attempt to build your project and *it must be a clean build with no errors or warnings*.

A sample makefile is given below. The autograder will attempt a “make all”, and then it will try to run “./sush”. If that fails you will receive 0 points from the autograder.

I will take responsibility to fix actual errors with the autograder, but I will not give any points for failure to follow the instructions in this assignment. You can check your results against the autograder, so there is no reason to give me any excuses here.

OBJS=sush.o tokenizer.o list.o commands.o  
  
all: sush

clean:

rm –f \*.o sush sush-debug

.c.o:

$(CC) $(CFLAGS) -c $<

sush: $(OBJS)  
 $(CC) -o sush $(OBJS)

sush-debug: $(OBJS)  
 $(CC) –g sush-debug $(OBJS)

Also, I will be manually grading these projects as well. The autograder will perform basic tests. I will be manually running your shell, and I will be evaluating your code. The rubric will be available on Gradescope prior to the deadline.

## Requirement 10 – Cannot use “system()” or any such tools

*Checked during code review*

You may not use the “system()” system call or anything like this. This runs a command in a sub-shell, so in effect, you’re circumventing the whole learning process this way.

Doing this will result in a zero for the project.

## Requirement 11 – Use Gitlab

*Checked during code review*

Your project must be stored in a gitlab repository that is shared with me. At the conclusion of the project I will be reviewing your branch and commit log. Part of your grade will be for effective use of the repository. The other thing I will be looking for is evidence that your teammates all contributed to different parts of this project.

Yes, we encourage paired programming, but that does not mean one person doing all the work and the other making “helpful suggestions.” That means people working together on one part, different people working together on other parts. That means everyone should be driving the keyboard, committing code, and contributing.

Another advantage to using Gitlab for this project is that you can take advantage of the *issue tracker*, *wiki*, *continuous integration pipeline*, etc. For example, you can set up the gitlab repository that everytime you push code to it automatically does a *make* and runs your test code. Its cool, but don’t get lost in the weeds there.

The real advantage is the issue tracker and planning tools. This will help you stay on track throughout the project.

## Requirement 12 – Code Review

50 points

Follows the coding standard at: <https://users.ece.cmu.edu/~eno/coding/CCodingStandard.html>

|  |  |  |
| --- | --- | --- |
| Item | Description | Importance |
| 1 | Code follows consistent style | HIGH |
| 2 | Are variable declarations commented with meaningful names? | HIGH |
| 3 | Are variables declared with *minimal scope e.g. there should be zero global variables in this project, but can use* static *keyword to limit scope to single compilation units* | MEDIUM |
| 4 | Are functions / procedures commented with meaningful names? | HIGH |
| 5 | Are functions / procedures doing what the comments say they do? | HIGH |
| 6 | Are complex tasks broken down into simpler functions? | HIGH |
| 7 | Are functions declared with *minimal scope* *e.g. functions used through the project are in .h files, helper functions that should not be used elsewhere are* static *to limit scope to single compilation units* | MEDIUM |
| 8 | Is the project broken down in appropriate compilation units? Aptly named “.c” files with only public/shared things appearing in “.h” files. | HIGH |
| 9 | Are complex algorithms and code optimizations documented | HIGH |
| 10 | Are modification logs, authorship attributions kept up to date | LOW |
| 11 | Do comment blocks include appropriate doxygen tags? See coding style. | MEDIUM |
| 12 | Overall, subjective evaluation on the quality of the code | MEDIUM |

## Requirement 13 – Final Project Presentation

*50 points*

Your group will need to record a video demonstration of your project. Your video should be a mixture of a showing your shell working through each of the different requirements, as well as talking about who worked on the different requirements, and walk me through how you implemented the code for that section. Talk about things you’re proud of, things you wish you knew how to do better, things you’ve learned trying to do that section. Things that you think are fragile and if you had more time would like to go back and redo. This is usually called a *post mortem retrospective*, and is an important part of the software development process.

* Your video will be uploaded to D2L and needs to be playable through the D2L viewer.
* You can use free screen recording utilities or record a zoom meeting that your group hosts
* Your video must be between 5 and 15 minutes. Any videos outside this range will get 0 points.

## Requirement 13 – Team Participation

*Assessed through CATME. Up to 20 points individually for the quality of the reviews you give others.  
Based on the CATME results, individual grades for the project can be adjusted or even zeroed out or more severe. Read the rest of this section.*

This is a group project, and I expect everyone to fully participate.

This is a 300-level course. If you’ve gotten this far and cannot participate in a group coding project then you’ve got more problems ahead of you in your career and future studies than getting a bad grade in this course. If that is the case, then it is time to perform an auto-rectal-cranial extraction and step it up, or at least take the burden off your team, admit that you are a failure and step aside.

Further, when I say “participate” I don’t mean bringing your group members coffee and washing their cars while they do the actual work of the project. If that is your level of contribution then as soon as you put your name on the project you’ve committed academic dishonesty and I reserve the right to follow through putative actions laid out in the syllabus. If your team members cover up your laziness or incompetence then they’ve joined you in the academic dishonesty. Given that they risk the same penalties don’t be angry at them when they narc you out.

I guess the strategy here is – fail fast. If you aren’t going to put your effort into this required course in your major then step aside right away, take the F for the project, and do some soul searching. If you tried to play games with your team then expect the F for the course plus any additional penalties the judicial board may apply.

1. It could be worse, we could have been playing *Dumb and Dumber* on you. [↑](#footnote-ref-1)