# CS/SE 3377 Sys. Prog. in Unix and Other Envs.
# Project 1
# Due: March 22nd, 2024

This project may be done individually or with a partner. I strongly recommend working with a partner. Sharing your work with anyone other than your partner is strictly prohibited. Do not look for a solution on the net or any other outside sources. Do not use, even partially, others' work. **Any act of plagiarism will be reported to the Dean of OSC.**

You can develop the project in cs1/cs2/cs3. Compile the code in any of the cs* machines. However, do not test/execute in cs* machines. Use any of the {net01, …, net45} machines for program execution. You may create a large number of processes due to a bug (more on how to handle forkbomb later). You do not want to slow down cs* machines by executing them. CSTech does not take kindly if you do this. If you are caught executing your program in any of the cs* machines, you will be given a ticket. For every ticket you receive, we will deduct 5 points from your final score.

## Simple Shell (sish)

### Background

In this project, you will implement a command line interpreter or shell. The shell should operate in this basic way: when you type in a command (in response to its prompt), the shell creates a child process that executes the command you entered and then prompts for more user input when it has finished.

The shell you implement will be like, but much simpler than, the one (bash) you run every day in Linux. You can find out which shell you are running by typing echo $SHELL at a prompt. A typical shell provides many functionalities and features. For this project, you need to implement only some as specified below.

### Part 1: The Simple Shell

1. **Your shell executable should be named sish.** Your shell's source code should be in sish.c,
2. **The shell should run continuously and display a prompt when waiting for input.** The prompt should be EXACTLY sish> . Note the space after greater than sign. Example with a command:

```
sish> ls -l
```

3. **Your shell should read a line from stdin one at a time.** This line should be parsed out into a *command* and *all its arguments*. In other words, tokenize it.

- You may assume that the only supported delimiter is the whitespace character (ASCII character number 32).
- You do not need to handle "special" characters. Do not worry about handling quotation marks, backslashes, and tab characters. This means your shell will be unable to support arguments with spaces in them. For example, your shell will not support file paths with spaces in them.
- You may set a reasonable maximum on the number of command line arguments, but your shell should handle input lines of any length.

4. **After parsing the command, your shell should execute it.** A command can either be a reference to an executable OR a built-in shell command (see Part 2). For Part 1, just focus on running executables, and not on built-in commands.
   - Executing commands that are not shell built-ins and are **just** the executable name (and not a full path to the executable) is done by invoking `fork()` and then invoking `execvp()`.
   - You may **NOT** use the `system()` function, as it just invokes the `/bin/sh` shell to do all the work.

## Part 2: Implement Built-in Commands: `exit, cd, history`

- `exit` - Simply exits your shell after performing any necessary clean up.
- `cd [dir]` - Short for "change directory" and will be used to change the current working directory of your shell. Do not worry about implementing the command line options that the real cd command has in Bash. Just implement cd such that it takes a single command line argument: The directory to change to. To change directory, use the chdir() system call with the argument supplied by the user. If chdir fails, it is an error.

- `history [-c] [offset]` - Like the Bash built-in `history` command, but much simpler.
  - `history` (without arguments) displays the last 100 commands the user ran, with an offset next to each command. The offset is the index of the command in the list, and valid values are 0 to 99, inclusive. 0 is the oldest command. Do not worry about saving this list to a file; just store it in memory. Once more than 100 commands are executed, remove the oldest entry from the list to make room for the newer commands. Note that history is also a command itself and therefore should also appear in the list of commands. If the user ran invalid commands, those should also appear in the list.
  - `history -c` clears the entire history, removing all entries. For example, running `history` immediately after `history -c` should show `history` as the sole entry in the list.
  - `history [offset]` executes the command in history at the given offset. Print an error message of your choosing if the offset is not valid.
  - Example output for built-in `history`:

```
sish> cd /home/3377
sish> ls
my_file.txt
sish> history
```

```
0 cd /home/3377
1 ls
2 history
sish> history 1
my_file.txt
sish> history
0 cd /home/CS4348
1 /bin/ls
2 history
3 history 1
4 history
sish> history -c
sish> history
0 history
sish>
```

**Part 3: Pipes**

- Augment your shell to be capable of executing a sequence of programs that communicate through a pipe. For example, if the user enters `ls | wc`, your program should fork the two programs, which together will calculate the number of files in the directory. For this you will need to replace `stdin` and `stdout` with pipe file descriptors using `dup2`.
- While this example shows two processes communicating through a pipe, your shell should support pipes between multiple processes, not just two.
- You **need not** support built-in commands to work with pipes.

**Hints**

For reading lines of input, you should use getline(). This allows you to obtain arbitrarily long input lines with ease. To parse the input line into constituent pieces, you might want to use strtok() (or, if doing nested tokenization, use strtok_r()). Read the man page (carefully) for more details.

Remember to get the basic functionality of your shell working before worrying about all the error conditions and end cases. For example, first get a single command running (probably first a command with no arguments, such as ls).
Next, add built-in commands. Finally, think about pipe commands. Each of these requires a little more effort on parsing, but each should not be too hard to implement.

**Other Requirements**

Error handling is an important concept in system programming: a shell can't simply fail when it encounters an error; it must check all parameters before it trusts them. In general, there should be no circumstances in which your C program will core dump, hang indefinitely, or prematurely terminate. Therefore, your program must respond to all input in a reasonable manner; by "reasonable", we mean print an understandable error message and either continue processing or exit, depending upon the situation.

So, check the input for errors. Check the return values of function calls. Display appropriate error messages. You will lose some points if you have not checked the return values of system/library call, for errors.

**Testing**

Make sure you compile your program as follows:
gcc sish.c –o sish -Wall -Werror -std=gnu99

There should not be any error messages and warning during compilation.

Sample testcases are in directory ~sxa173731/3377/testcases/p1_testcases. You should also write your own test cases and test thoroughly. Writing good testcases is an important skill you need to do well in your career.

Your program may end up creating too many processes than intended due to a bug (fork bomb). To prevent yourself from being locked out of the system limit the number of processes you can create by adding this line to.bashrc file in your home directory.

ulimit –u 100

In case you reach the limit of 100, then login through a different terminal and kill sish and all the processes created by sish. Use commands "ps –u" and kill pid.

**Hand-in Instructions**

Name your program file as sish.c and submit it on elearning.

If you have worked with a partner, only one of you should submit the file. But both of you should upload a text file named PARTNER.txt which should contain partners' names and netids.

**Grading Policy**

Source code should be structured well with adequate comments clearly showing the different parts and functionalities implemented.

The TA will compile the code and test it in one of cs* machines. If your program does not compile in cs* machines, you will get 0 points.

You may also be called by the TA to demonstrate your code. If you are not able to explain the working of your code, you will not be given any points.

If you are working as a group, both the partners should contribute equally. If you didn't contribute, you will get 0 points.

Rubric:

Code review - 35 points. Testcases - 55 points. Style and comments - 10 points