UNIVERSITY OF SAN FRANCISCO | College of Arts and Sciences

<NAME OF YOUR PROGRAM/DEPARTMENT/MAJOR OR MINOR>

ASSESSMENT REPORT
ACADEMIC YEAR 2018 – 2019

I. LOGISTICS

1. **Please indicate the name and email of the program contact person to whom feedback should be sent (usually Chair, Program Director, or Faculty Assessment Coordinator).**

   EJ Jung, ejung2@usfca.edu, Chair and Faculty Assessment Coordinator of CS dept.

2. Please indicate if you are submitting report for (a) a Major, (b) a Minor, (c) an aggregate report for a Major & Minor (in which case, each should be explained in a separate paragraph as in this template), (d) a Graduate or (e) a Certificate Program

(a) CS Major

3. Please note that a Curricular Map should accompany every assessment report. Has there been any revisions to the Curricular Map?

No changes were made.

II. MISSION STATEMENT & PROGRAM LEARNING OUTCOMES

1. Were any changes made to the program mission statement since the last assessment cycle in October 2018? Kindly state "Yes" or "No." Please provide the current mission statement below. If you are submitting an aggregate report, please provide the current mission statements of both the major and the minor program

Mission Statement (Major/Graduate/Certificate):

No changes were made.

Students who graduate with a Bachelor of Science (B.S.) degree in Computer Science will be prepared for both graduate school and for software development careers. The curriculum provides a solid base in computer science fundamentals that includes software design and development, problem solving and debugging, theoretical and mathematical foundations, computer systems, and system software.

2. **Were any changes made to the program learning outcomes (PLOs) since the last assessment cycle in October 2017? Kindly state "Yes" or "No." Please provide the current PLOs below. If you are submitting an aggregate report, please provide the current PLOs for both the major and the minor programs.**

   PLOs (Major/Graduate/Certificate):

   No changes were made.

   - THEORY: Explain and analyze standard computer science algorithms and describe and analyze theoretical aspects of various programming languages.
   - APPLICATION: Apply problem-solving skills to implement medium- and large-scale programs in a variety of programming languages.
   - SYSTEMS: Describe the interactions between low-level hardware, operating systems, and applications.
   - PROJECT: Demonstrate effective communication and organization as part of a team of software developers or researchers collaborating on a large computer program.

3. **State the particular Program Learning Outcome(s) you assessed for the academic year 2018-2019.**

PLO(s) being assessed (Major/Graduate/Certificate):

- SYSTEMS: Describe the interactions between low-level hardware, operating systems, and applications.

III. METHODOLOGY

Describe the methodology that you used to assess the PLO(s).

Mastery in Operating Systems is defined by achieving the learning outcomes listed below. Those students who successfully pass this class should being able to do the following:

1. Configure a Linux-based operating system and work from the shell
2. Understand and evaluate operating system implementations
3. Understand the implementation of fundamental OS structures, including threads, processes, synchronization, system calls, scheduling, virtual memory, and file systems
4. Develop and debug systems software

The assignment specification for Project 2 is included, as it spans items 2, 3, and 4 above by requiring students to build a fundamental part of UNIX-based Operating Systems.

The final grade for this course is computed based on the following breakdown:

Projects: 50%

Homework and Labs: 20%

Quizzes: 30%

IV. RESULTS & MAJOR FINDINGS

What are the major takeaways from your assessment exercise?

To assess mastery, we split the students into four groups:

1: "complete mastery of the outcome"
2: "mastered the outcome in most parts",
3: "mastered some parts of the outcome"
4: "did not master the outcome at the level intended."

The table below lists how many students fell into each category for each assignment group:

Fall 2018 (One Section, 31 students)

Description	**1**	**2**	**3**	**4**
Labs	23	6	0	2
Project 1	12	11	5	3
Project 2	10	14	4	3
Project 3	4	14	11	2
Project 4	9	10	6	6
Quizzes	6	9	8	8
Final Grade	7	17	3	4

Spring 2019 (Two Sections, 48 students)

Description	1	2	3	4
Labs	38	3	3	4
Project 1	26	11	7	4
Project 2	32	11	2	3
Project 3	29	11	6	2
Project 4	19	5	21	3
Quizzes	8	14	22	4
Final Grade	20	14	9	5

Interpretation of the results:

- One of the major changes from Fall 2018 to Spring 2019 was the inclusion of an interactive test framework to allow students to check their project grade as they worked. This led to a substantial increase in the number of students reaching complete mastery since they were able to see where their implementations went wrong and correct them before turning in the projects. As a result, the number of students achieving complete mastery of the class overall is higher.
- In general, labs are graded lightly but require attendance. Low performance on the labs corresponds to attendance issues, which also generally correspond with low mastery.
- In Spring of 2019, the scope of Project 3 was increased while the scope of Project 4 was decreased; many students with high grades opted to not fully finish Project 4 (explaining the abnormally high number of students in category 3)

V. CLOSING THE LOOP

1. Based on your results, what changes/modifications are you planning in order to achieve the desired level of mastery in the assessed learning outcome? This section could also address more long-term planning that your department/program is considering and does not require that any changes need to be implemented in the next academic year itself.

We will continue to include interactive grading in OS curriculum and also encourage students to attend the lab sessions for the immediate feedback from the instructor and the TAs.

2. What were the most important suggestions/feedback from the FDCD on your last assessment report (for academic year 2016-2017, submitted in October 2017)? How did you incorporate or address the suggestion(s) in this report?

- the Theory PLO is really two as it describes two different assessment processes. If the algorithms and the theoretical aspects of various programming languages are inextricably linked, then I would review both as part of a single assessment activity. However, if they are really different and are assessed differently then, I would split this PLO into two different PLOs.

  The algorithmic analysis and programming languages are indeed very much linked together. For example, copying a list in one programming language takes as many steps as the size of the list, while that in another programming language takes only one step. Understanding such difference is part of the learning outcome of CS major.

ADDITIONAL MATERIALS

(Any rubrics used for assessment, relevant tables, charts and figures should be included here)

The project 2 specification including the rubric and the test cases used for grading are attached. Please note that passing each test case adds a point to a student's score.

Project 2: Command Line Shell (v 1.0)

Starter repository on GitHub: https://classroom.github.com/a/OhBlq5yL

The outermost layer of the operating system kernel is called the *shell*. In Unix-based systems, the shell is generally a command line interface. Most Linux distributions ship with `bash` as the default (there are several others: `csh`, `ksh`, `sh`, `tcsh`, `zsh`). In this project, we'll be implementing a shell of our own – see, I told you that you'd come to love command line interfaces in this class!

You will need to come up with a name for your shell first. The only requirement is that the name ends in 'sh', which is tradition in the computing world. In the following examples, my shell is named `crash` (Cool Really Awesome Shell) because of its tendency to crash.

The Basics

Upon startup, your shell will print its prompt and wait for user input. Your shell should be able to run commands in both the current directory and those in the PATH environment variable. Use `execvp` to do this. To run a command in the current directory, you'll need to prefix it with `./` as usual. If a command isn't found, print an error message:

```
┌─[🙂]─[0]─[mmalensek@glitter-sparkle:~/P2-malensek]
└─╼ ./hello
Hello world!

┌─[🙂]─[1]─[mmalensek@glitter-sparkle:~/P2-malensek]
└─╼ ls /usr
bin  include  lib  local  sbin  share  src

┌─[🙂]─[2]─[mmalensek@glitter-sparkle:~/P2-malensek]
└─╼ echo hello there!
hello there!

┌─[🙂]─[3]─[mmalensek@glitter-sparkle:~/P2-malensek]
└─╼ ./blah
crash: no such file or directory: ./blah

┌─[🤮]─[4]─[mmalensek@glitter-sparkle:~/P2-malensek]
└─╼ cd /this/does/not/exist
chdir: no such file or directory: /this/does/not/exist

┌─[🤮]─[5]─[mmalensek@glitter-sparkle:~/P2-malensek]
└─╼
```

Prompt

The shell **prompt** displays some helpful information. At a minimum, you must include:

- Command number (starting from 0)
- User name and host name: `(username)@(hostname)` followed by `:`
- The current working directory
- Process exit status

In the example above, these are separated by dashes and brackets to make it a little easier to read. The process exit status is shown as an emoji: a smiling face for success (exit code `0`) and a sick face for failure (any nonzero exit code). For this assignment, you are allowed to invent your own prompt format as long as it has the elements listed above. You can use colors, unicode characters, etc. if you'd like. For instance, some shells highlight the next command in red text after a nonzero exit code.

You will format the current working directory as follows: if the CWD is the user's home directory, then the entire path is replaced with `~`. Subdirectories under the home directory are prefixed with `~`; if I am in `/home/mmalensek/test`, the prompt will show `~/test`. Here's a test to make sure you've handled `~` correctly:

```
┌─[🙂]─[6]─[mmalensek@glitter-sparkle:~]
└─╼ whoami
mmalensek

┌─[🙂]─[7]─[mmalensek@glitter-sparkle:~]
└─╼ cd P2-malensek

# Create a directory with our full home directory in its path:
# **Must use the username outputted above from whoami)**
┌─[🙂]─[8]─[mmalensek@glitter-sparkle:~/P2-malensek]
└─╼ mkdir -p /tmp/home/mmalensek/test

┌─[🙂]─[9]─[mmalensek@glitter-sparkle:~/P2-malensek]
└─╼ cd /tmp/home/mmalensek/test

# Note that the FULL path is shown here (no ~):
┌─[🙂]─[10]─[mmalensek@glitter-sparkle:/tmp/home/mmalensek/test]
└─╼ pwd
/tmp/home/mmalensek/test
```

Scripting

Your shell must support scripting mode to run the test cases. Scripting mode reads

commands from standard input and executes them without showing the prompt.

```
cat <<EOM | ./crash
ls /
echo "hi"
exit
EOM

# Which outputs (note how the prompt is not displayed):
bin  boot  dev  etc  home  lib  lost+found  mnt  opt  proc  root  run  sbin  s
hi

# Another option (assuming commands.txt contains shell commands):
./crash < commands.txt
(commands are executed line by line)
```

You should check and make sure you can run a large script with your shell. Note that the script should not have to end in `exit`.

To support scripting mode, you will need to determine whether stdin is connected to a terminal or not. If it's not, then disable the prompt and proceed as usual. Here's some sample code that does this with `isatty`:

```
#include <stdio.h>
#include <unistd.h>

int main(void) {

    if (isatty(STDIN_FILENO)) {
        printf("stdin is a TTY; entering interactive mode\n");
    } else {
        printf("data piped in on stdin; entering script mode\n");
    }

    return 0;
}
```

When implementing scripting mode, check:

- The return value of `getline`, `fgets`, etc. If you reach `EOF`, exit the shell.
- In the child process, you may need to close `stdin` to prevent infinite loops.

Built-In Commands

Most shells have built-in commands, including `cd` and `exit`. Your shell must support:

- `cd` to change the CWD. `cd` without arguments should return to the user's home directory.
- `#` (comments): strings prefixed with `#` will be ignored by the shell
- `history`, which prints the last 100 commands entered with their command numbers
- `!` (history execution): entering `!39` will re-run command number 39, and `!!` re-runs the last command that was entered. `!ls` re-runs the last command that starts with 'ls.' Note that command numbers are **NOT** the same as the array positions; e.g., you may have 100 history elements, with command numbers 600 – 699.
- `setenv` to set environment variables; the syntax is `setenv variable-name variable-value`.
- `jobs` to list currently-running background jobs.
- `exit` to exit the shell.

Variable Expansion

Your shell's command line parser should support variable expansion denoted by the `$` symbol; for instance, `$VAR` in the command line will be expanded to the value of the `VAR` environment variable. Use `getenv()` to retrieve the values. To evaluate variable replacements, look for the `$` character in each token. To get the basic functionality working, simply replace `$VAR` with its actual value as a single token. To fully implement this functionality, you must also tokenize the variables' values as well (if `VAR` is set to 'hello world', it should be parsed as two tokens).

Signal Handling

Your shell should gracefully handle the user pressing Ctrl+C:

```
┌─[🙂]─[11]─[mmalensek@glitter-sparkle:~]
└─╸ hi there oh wait nevermind^C

┌─[🙂]─[11]─[mmalensek@glitter-sparkle:~]
└─╸ ^C

┌─[🙂]─[11]─[mmalensek@glitter-sparkle:~]
└─╸ ^C

┌─[🙂]─[11]─[mmalensek@glitter-sparkle:~]
└─╸ ^C
```

```
┌─[🙂]─[11]─[mmalensek@glitter-sparkle:~]
└─╼ sleep 100
^C

┌─[🤮]─[12]─[mmalensek@glitter-sparkle:~]
└─╼ sleep 5
```

The most important aspect of this is making sure `^C` doesn't terminate your shell. To make the output look like the example above, in your signal handler you can (1) print a newline character, (2) print the prompt *only* if no command is currently executing, and (3) `fflush(stdout)`.

History

Here's a demonstration of the `history` command:

```
┌─[🙂]─[13]─[mmalensek@glitter-sparkle:~]
└─╼ history
  42 ls -l
  43 top
  44 echo "hi" # This prints out 'hi'

... (commands removed for brevity) ...

 140 ls /bin
 141 gcc -g crash.c
```

In this demo, the user has entered 141 commands. Only the last 100 are kept, so the list starts at command 42. If the user enters a blank command, it should **not** be shown in the history or increment the command counter.

Redirection

Your shell must support file output redirection and pipe redirection:

```
# Create/overwrite 'my_file.txt' and redirect the output of echo there:
┌─[🙂]─[14]─[mmalensek@glitter-sparkle:~]
└─╼ echo "hello world!" > my_file.txt

# Pipe redirection:
┌─[🙂]─[15]─[mmalensek@glitter-sparkle:~]
└─╼ cat other_file.txt | sort
(contents shown)

┌─[🙂]─[16]─[mmalensek@glitter-sparkle:~]
```

```
└─▪ seq 100000 | wc -l
10000

┌─[🙂]─[17]─[mmalensek@glitter-sparkle:~]
└─▪ cat /etc/passwd | sort > sorted_pwd.txt
(contents shown)
```

Use `pipe` and `dup2` to achieve this. Your implementation should support arbitrary numbers of pipes, but you only need to support one file redirection per command line (i.e., a `>` in the middle of a pipeline is not something you need to consider).

Background Jobs

If a command ends in `&`, then it should run in the background. In other words, don't wait for the command to finish before prompting for the next command. If you enter `jobs`, your shell should print out a list of currently-running backgrounded processes. The status of background jobs is not shown in the prompt.

To implement this, you will need:

- A signal handler for `SIGCHLD`. This signal is sent to a process any time one of its children exit.
- A non-blocking call to `waitpid` in your signal handler. Pass in `pid = -1` and `options = WNOHANG`.

The difference between a background job and a regular job is simply whether or not a blocking call to `waitpid()` is performed. If you do a standard `waitpid()` with `options = 0`, then the job will run in the foreground and the shell won't prompt for a new command until the child finishes (the usual case). Otherwise, the process will run and the shell will prompt for the next command without waiting.

Each background process should be stored in a job array, with a maximum of 10 background jobs. **NOTE**: your shell prompt output may print in the wrong place when using background jobs. This is completely normal.

Hints

Here's some hints to guide your implementation:

- `execvp` will use the PATH environment variable (already set up by your default shell) to find executable programs. You don't need to worry about setting up the

PATH yourself.

- Check out the `getlogin`, `gethostname`, and `getpwuid` functions for constructing your prompt.
- Don't use `getwd` to determine the CWD – it is deprecated on Linux. Use `getcwd` instead.
- For the `cd` command, use the `chdir` syscall.
- For sizing your arrays, check out `HOST_NAME_MAX`, `PATH_MAX`, and `ARG_MAX`
- To replace the user home directory with `~`, the `strstr` function may be useful. Some creative manipulation of character arrays and pointer arithmetic can save you some work.

Testing Your Code

Check your code against the provided test cases. You should make sure your code runs on your Arch Linux VM. We'll have interactive grading for projects, where you will demonstrate program functionality and walk through your logic.

Submission: submit via GitHub by checking in your code before the project deadline. You **must** include a makefile with your project. As part of the testing process, we will check out your code and run `make` to build it.

Grading

- 25 pts - Passing the test cases.
- 5 pts - Code review:
  - Prompt functionality
  - Code quality and stylistic consistency
  - Functions, structs, etc. must have documentation in Doxygen format (similar to Javadoc). Describe inputs, outputs, and the purpose of each function. **NOTE:** this is included in the test cases, but we will also look through your documentation.
  - No dead, leftover, or unnecessary code.
  - You must include a README.md file that describes your program, how it works, how to build it, and any other relevant details. You'll be happy you did this later if/when your revisit the codebase. Here is an example README.md file.

Extra Credit

- 1 pts - Add support for moving back and forward through the history array with the arrow keys (up arrow moves back one command, down arrow moves forward a command). This should behave similarly to the `bash` history navigation.
- 1 pts - Support for editing the command line with the left/right arrow keys, backspace, etc.
- 2 pts - Tab completion for programs in `$PATH` as well as directories.
- 1 pts - Find a bug in one of the test cases or propose a new test case that wasn't covered in those provided.

Restrictions: you may use any standard C library functionality. External libraries are not allowed unless permission is granted in advance. In particular, you may not use the `readline` library for this project. Your shell may not call another shell (e.g., running commands via the `system` function or executing `bash`, `sh`, etc.). Do not use `strtok`; use your own tokenization implementation. Your code **must** compile and run on your VM set up with Arch Linux as described in class. Failure to follow these guidelines will will result in a grade of **0**.

Changelog

- Initial project specification posted (10/1)