UNIVERSITY OF SAN FRANCISCO | College of Arts and Sciences

<Computer Science/Bridge program>

ASSESSMENT REPORT
ACADEMIC YEAR 2020 – 2021

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).**

   Alark Joshi, apjoshi@usfca.edu, Chair of CS dept.

   EJ Jung, ejung2@usfca.edu, 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

(d) Graduate

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

No. The curricular map is attached.

II. MISSION STATEMENT & PROGRAM LEARNING OUTCOMES

1. **Were any changes made to the program mission statement since the last assessment cycle in October 2020? 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**

   No changes were made.

The mission of the MS in Computer Science Bridge program is:

To prepare students for Master's in Computer Science at USF who are changing fields from non-computer science backgrounds and to give students who do not have a computer science background enough knowledge to do basic software development.

2. **Were any changes made to the program learning outcomes (PLOs) since the last assessment cycle in October 2020? 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.**

No. Each learning outcome is marked with which AY it was evaluated. Please note that the MSCS Bridge program started in AY 17-18, so we do not have any assessment history before then.

Students who pass the bridge program and proceed to the MS in Computer Science will be able to:

- Application: Implement medium- and large-scale programs in a variety of programming languages. (evaluated in AY18-19)
- Theory: Explain and analyze standard computer science algorithms (evaluated in AY 17-18)
- Systems: Describe the interactions between low-level hardware, operating systems, and applications (evaluated in AY20-21)

3. **State the particular Program Learning Outcome(s) you assessed for the academic year 2020-2021.**

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).

We used a direct method. CS 521 Systems Programming is a required course for the Bridge students. (Please note that CS 521 ran as CS 686 Special Topics in Computer Science in Spring 2021 while the course number was getting approved in Curriculog.)

We used the second programming project in CS 521 to assess if students met this learning outcome. In this programming project, the students not only described but also demonstrated their understanding of the interactions between low-level hardware, operating systems and applications by implementing a program that relies on such interactions, using system calls, do low-level I/O, launch applications and facilitate communication between them via pipes. The project description and the rubric are attached.

IV. RESULTS & MAJOR FINDINGS

What are the major takeaways from your assessment exercise?

Level	Percentage of Students
Complete Mastery of the outcome	71% (12/17)
Mastered the outcome in most parts	24% (4/17)
Mastered some parts of the outcome	5% (1/17)
Did not master the outcome at the level intended	0% (0/17)

The vast majority (16 out of 17) of the students mastered the outcome in most parts. 1 student only mastered some parts of this outcome at the time of this project, but upon further investigation we learned that the student was able to master the outcome later in the semester and demonstrated the mastery in the exam. (Please note that the learning outcome expects the student to be able to describe, not necessarily implement, so the exam is a direct method as well.) This student was able to pass the course and is doing well in the MSCS courses. This is not uncommon in the bridge program, where students with non-CS background learn to code for the first time and some students take longer to reach the mastery. CS faculty is committed to supporting such students in our Bridge program.

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.

It is assuring that the students who pass this course fulfills the program's learning outcome properly and continue to the Master's program. This is the third cohort of the Bridge program, and students who successfully finished our Bridge program are making good progress in their Master's Program. Our first cohort recently graduated and many of them were hired before graduation. We will continue to track their achievements and revise the curriculum if necessary.

The report is shared with the CS faculty, especially with the department chair, the graduate program director and the graduate program manager to keep them informed with the learning outcome achievement.

2. What were the most important suggestions/feedback from the FDCD on your last assessment report (for academic year 2019-2020, submitted in December 2020)? How did you incorporate or address the suggestion(s) in this report?

In AY 2019-2020, we used alternative assessment checking on how the CS faculty and the students are adapting to the remote courses. We used the feedback from the survey to increase the online social activities to foster the sense of belonging, and the activities were well-received.

Project 2: Command Line Shell (v 1.0)

Starter repository on GitHub: https://classroom.github.com/a/_x-WbepJ

The outermost layer of the operating system 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.

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 (run `echo $PATH` to see the directories in your PATH). The `execvp` system call will do most of this for you. 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:

```
[🙂]─[1]─[mmalensek@gamestop:~/P2-malensek]$ ./hello
Hello world!

[🙂]─[2]─[mmalensek@gamestop:~/P2-malensek]$ ls /usr
bin  include  lib  local  sbin  share  src

[🙂]─[3]─[mmalensek@gamestop:~/P2-malensek]$ echo hello there!
hello there!

[🙂]─[4]─[mmalensek@gamestop:~/P2-malensek]$ ./blah
crash: no such file or directory: ./blah

[🤮]─[5]─[mmalensek@gamestop:~/P2-malensek]$ cd /this/does/not/exist
chdir: no such file or directory: /this/does/not/exist

[🤮]─[6]─[mmalensek@gamestop:~/P2-malensek]$
```

Prompt

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

- Command number (starting from 1)
- 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 or failure to execute the child process). 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@gamestop:~]$ whoami
mmalensek

[🙂]─[7]─[mmalensek@gamestop:~]$ cd P2-malensek

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

[🙂]─[9]─[mmalensek@gamestop:~/P2-malensek]$ cd /tmp/home/mmalensek/te

# Note that the FULL path is shown here (no ~):
[🙂]─[10]─[mmalensek@gamestop:/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
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;
}
```

Since the `readline` library we're using for the shell UI is intended for interactive use, you will need to switch to a traditional input reading function such as `getline` when operating in scripting mode.

When implementing scripting mode, you will likely need to close `stdin` on the child process if your call to `exec()` fails. This prevents 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.
- `jobs` to list currently-running background jobs.
- `exit` to exit the shell.

Signal Handling

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

```
[🙂]─[11]─[mmalensek@gamestop:~]$ hi there oh wait nevermind^C
[🙂]─[11]─[mmalensek@gamestop:~]$ ^C
[🙂]─[11]─[mmalensek@gamestop:~]$ ^C
[🙂]─[11]─[mmalensek@gamestop:~]$ sleep 100
^C
[🤮]─[12]─[mmalensek@gamestop:~]$ 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:

```
[🙂]─[142]─[mmalensek@gamestop:~]$ history
  43 ls -l
  43 top
  44 echo "hi" # This prints out 'hi'

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

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

In this demo, the user has entered 142 commands. Only the last 100 are kept, so the list starts at command 43. If the user enters a blank command, it should **not** be shown in the history or increment the command counter. Also note that the entire, original command line string is shown in the history – not a tokenized or modified string. You should store history commands exactly as they are entered (hint: use `strdup` to duplicate and store the command line string). The only exception to this rule is when the command is a history execution (bang) command, e.g., !42. In that case, determine the corresponding command line and place it in the history (this prevents loops).

I/O Redirection

Your shell must support file input/output redirection:

```
# Create/overwrite 'my_file.txt' and redirect the output of echo there
[🙂]─[14]─[mmalensek@gamestop:~]$ echo "hello world!" > my_file.txt
[🙂]─[15]─[mmalensek@gamestop:~]$ cat my_file.txt
hello world!

# Append text with '>>':
[🙂]─[16]─[mmalensek@gamestop:~]$ echo "hello world!" >> my_file.txt
[🙂]─[17]─[mmalensek@gamestop:~]$ cat my_file.txt
hello world!
hello world!

# Let's sort the /etc/passwd file via input redirection:
[🙂]─[18]─[mmalensek@gamestop:~]$ sort < /etc/passwd > sorted_pwd.txt

# Order of < and > don't matter:
[🙂]─[19]─[mmalensek@gamestop:~]$ sort > sorted_pwd.txt < /etc/passwd

# Here's input redirection by itself (not redirecting to a file):
[🙂]─[20]─[mmalensek@gamestop:~]$ sort < sorted_pwd.txt
```

```
(sorted contents shown)
```

Use `dup2` to achieve this; right before the newly-created child process calls `execvp`, you will open the appropriate files and set up redirection with `dup2`.

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 (use the original command line as it was entered, including the `&` character). 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`.
  - This tells your signal handler the PID of the child process that exited. If the PID is in your jobs list, then it can be removed.

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.

NOTE: your shell prompt output may print in the wrong place when using background jobs. This is completely normal.

The readline library

We're using the `readline` library to give our shell a basic "terminal UI." Support for moving through the current command line with arrow keys, backspacing over portions of the

command, and even basic file name autocompletion are all provided by the library. The details probably aren't that important, but if you're interested in learning more about `readline` its documentation is a good place to start.

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.
- To replace the user home directory with `~`, some creative manipulation of character arrays and pointer arithmetic can save you a bit of 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.

Grading (Rubric)

- 20 pts - Passing the test cases
- 4 pts - Code review: (The test cases were provided on the Github as part of the project template)
  - Prompt, UI, and general interactive functionality. We'll run your shell to test this with a few commands.
  - 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.

Restrictions: you may use any standard C library functionality. Other than `readline`, external libraries are not allowed unless permission is granted in advance (including the GNU `history` library). Your shell may not call another shell (e.g., running commands via the `system` function or executing `bash`, `sh`, etc.). Do not use `strtok` to tokenize input. 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 (3/17)

	PLO1	PLO2	PLO3
Program Learning Outcomes X Courses	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.
Courses or Program Requirement			
CS 514 Accelerated Object Oriented Programming	I/D	D	
SYSTEMS:			
CS 520 Introduction to Parallel Computing		D	D
THEORY:			
Math 501 Discrete Mathematics	D		
CS 545 Data Structures and Algorithms	M	D	
APPLICATIONS:			
CS Elective		M	
CS Practicum: Practical Industry or Research Experience		M	
	Key:		
	I = Introductory		
	D = Developing		
	M = Mastery		