TECHNION
Israel Institute
of Technology

Computer
Science
Department

Operating Systems - 234123

Homework Exercise 1 – Wet

Winter 2025

Teaching Assistant in charge:

Ahmad Agbaria

Assignment Subjects & Relevant Course material:

- **Processes and IPC (inter-process communication)**
- **Recitations 1-3 & Lectures 1-3**
- **HW0**

Introduction

A Unix shell is a command-line interpreter or shell that provides a command-line user interface for Unix-like operating systems[1].

There are several well-known shells on Unix systems, with Bash being one of the most widely used. Bash is a command processor that typically runs in a text window, allowing users to interact by typing commands that trigger various actions.

In this assignment, you will implement a 'smash' (small shell) that mimics the behavior of a real Linux shell but supports only a limited subset of Linux shell commands.

Be sure to review the entire document before you start working or asking questions.

Description

For simplicity, you can assume that only up to 100 processes can run simultaneously, and the name of each process can contain up to 50 characters. In the smash.zip file attachment you will find a base code for handling commands (Commands.cpp), a shell code for smash.cpp, and a Makefile. You must complete the code found in `Commands.cpp` and add signal handling. The functions that handle and define the signal handling routine will be in the file `siganls.cpp`.

<u>The program will function as follows:</u>

- The program waits for commands typed by the user and executes them.
- It can execute a limited number of **built-in commands**, which will be listed below.
- When the program receives a command that is not one of the **built-in commands** (referred to as an **external command**), it attempts to run it like a normal shell. The method for running external commands will be described later.
- Whenever an error occurs, a proper error message should be printed, and the program should return to parsing and executing the next command.
- If an empty command is entered, it will be ignored, and the prompt will be shown again on a new line.
- While waiting for the next command to be entered, the program should display the following prompt (this text can be changed using the **chprompt** command):

```
smash>
```

Assumptions

<u>You may assume the following:</u>

- Each command appears on a separate line and cannot exceed **200** characters (you may Not assume max length on specific parameters).
- The number of arguments for each command is up to **20**.
- Any number of spaces (see WHITESPACE macro in `Commands.h`) can be used between words in the same command line and at the beginning of the line unless stated otherwise.
- Your smash should support up to **100** processes running simultaneously.
- File and folder names do not include special characters and are in lowercase.
- I/O redirection and Pipe commands are correctly formatted.
- For this assignment, `new` or `malloc` won't fail, and memory leaks won't be checked.

[1] https://en.wikipedia.org/wiki/Unix_shell

Detailed Description

As mentioned before, there are basically two kinds of commands to be executed by smash: **built-in commands** and **external commands** (there are also **special commands** which we will describe later).

Built-in commands are features provided by the shell for its users. They run from the same code as the shell (same process) and should not be forked and executed separately. Generally, built-in commands are simple and do not execute external executables to achieve their goals. Instead, they usually run some query system call or maintain internal data structures of the shell.

External commands usually require running an external executable. To execute these, the shell runs them in a child process by calling fork and execv afterwards.

What are jobs?

In shell terminology, a job is a process managed by the shell (processes forked by the shell process). Each job has its own job ID, which is assigned by the shell once it's inserted into the jobs list and remains unchanged. Additionally, each job has a process ID (PID) assigned by the kernel. **Note that the job ID, assigned by the shell, is different from the process ID (PID) assigned by the kernel.**

A shell job can be in one of two states:

1) **Foreground**:

   When you type a command in the shell terminal window, the command will start running and cause the shell to be **blocked**, waiting for the command to finish (the shell waits for the [child] process to finish in this case). This is called a foreground job, where the shell waits for the process to complete before accepting new commands.

2) **Background**:

   When an ampersand symbol `&` is typed at the end of a command line, it means that the shell should run this command in the background. This means the job will not occupy the terminal window, and the shell prompt will be displayed immediately, allowing users to type new commands even though the background job is still running. The shell runs the job without waiting for its completion.

What is the jobs list?

Any job that is sent to the background (using the ampersand symbol `&`) is added to the **jobs list**, allowing users to query and manage them later by their associated **job ID**. For example, the user can ask the shell to print all the jobs it controls using the `jobs` command.

Jobs can be removed from the **jobs list** by the `fg` command, which brings a job from the background to the foreground, or because the job is finished. Refer to the `fg` and `jobs` commands, and signal handling for more details on managing the jobs list.

Deleting Finished Jobs:

Finished jobs should be deleted from the **jobs list** (1) before executing any command, (2) before printing the **jobs list** (jobs command), and (3) before adding new jobs to the **jobs list**.

Assigning job ID:

A job receives its **job ID** upon its first insertion into the **jobs list**, which remains unchanged. The **job ID** is assigned as follows: `Job ID = maximal job ID in jobs list + 1` If the list is empty, then `Job ID = 1`.

Built-in commands

Your smash should support and implement a limited number of shell commands (features). These commands should be executed from the smash process itself, meaning you should not fork the smash process to run them. Instead, they should be run directly from the smash code.

Below are the details of the built-in commands that your smash should support:

1. chprompt command (warm-up)

Command format:

```
chprompt  [new-prompt]
```

Description:

chprompt command will allow the user to change the prompt displayed by the smash while waiting for the next command.

If no parameters are provided, the prompt shall be reset to smash. If more than one parameter is provided, only the first parameter will be used, and the rest will be ignored.

Note that this command will not change the prompt in error messages (covered later).

Example:

```
smash> chprompt Jeffry_Epstein_didn't_kill_himself
Jeffry_Epstein_didn't_kill_himself> chprompt
smash>
```

2. showpid command

Command format:

```
showpid
```

Description:

showpid command prints the smash PID.

Example:

```
smash> showpid
smash pid is 30903
smash>
```

Error handling:

If any number of arguments were provided with this command, they will be ignored.

3. pwd command

Command format:

```
pwd
```

Description:

pwd command has no arguments.

The pwd command prints the full path of the current working directory. In the next section, we will learn how to change the current working directory using the `cd` command.
You may use getcwd system call to retrieve the current working directory.

Example:

```
smash> pwd
/home/ayalafos/234123/homeworks/smash/build
smash>
```

Error handling:

If any number of arguments were provided with this command, they will be ignored.

4. cd command

Command format:

```
cd <new-path>
```

Description:

The cd (Change Directory) command takes a single argument <path> that specifies either a relative or full path to change the current working directory to.
There's a special argument, `-`, that cd can accept. When `-` is the only argument provided to the cd command, it instructs the shell to change the current working directory to the last working directory.
For example, if the current working directory is X, and then the cd command is used to change the directory to Y, executing `cd -` would then set the current working directory back to X, executing it again would set the current directory to Y.
You may use chdir system call to change the current working directory.
If no argument is provided, this command has no impact.

Example:

```
smash> cd -
smash error: cd: OLDPWD not set
smash> cd dir1 dir2
smash error: cd: too many arguments
smash> cd /home/ayalafos
smash> pwd
/home/ayalafos
smash> cd ..
smash> pwd
/home
smash> cd -
smash> pwd
/home/ayalafos
smash>
```

Error handling:

If more than one argument was provided, then cd command should print the following error message:

```
smash error: cd: too many arguments
```

If the last working directory is empty and `cd -` was called (before calling cd with some path to change current working directory to it) then it should print the following error message:

```
smash error: cd: OLDPWD not set
```

If chdir() system call fails (e.g., <path> argument points to a non-existing path) then perror should be used to print a proper error message (as described in **Error Handling** section).

5. jobs command

Command format:

```
jobs
```

Description:

jobs command prints the **jobs list** which contains the unfinished jobs (Those running in the background).

The list should be printed in the following format: [<job-id>] <command>, where <command> is the original command provided by the user (including aliases as discussed later).

The jobs list should be printed in a sorted order w.r.t the job-id.

Make sure you delete all finished jobs before printing the jobs list.

Example:

```
smash> sleep 100&
smash> sleep 200&
smash> jobs
[1] sleep 100&
[2] sleep 200&
smash>
```

Note: sleep is an external command, we'll see external commands in the next section.

Error handling:

If any number of arguments were provided with this command, they will be ignored.

6. fg command

Command format:

```
fg [job-id]
```

Description:

fg command brings a process that runs in the background to the foreground.

fg command prints the command line of that job along with its PID (as can be seen in the example) and then waits for it (hint: waitpid), which in effect will bring the requested process to run in the foreground.

The job-id argument is an optional argument. If it is specified, then the specific job which its job id (as printed in jobs command) should be brought to the foreground. If the job-id argument is not specified, then the job with the maximal job id in the jobs list should be selected to be brought to the foreground.

Side effects: After bringing the job to the foreground, it should be removed from the jobs list.

Example:

```
smash> sleep 100&
smash> sleep 200&
smash> sleep 500&
smash> jobs
[1] sleep 100&
[2] sleep 200&
[3] sleep 500&
smash> fg 3
sleep 500& 901
```

Error handling:
If the syntax (number of arguments or the format of the arguments) is invalid then an error message should be printed as follows:

smash error: fg: invalid arguments

If job-id was specified with a job id that does not exist, then the following error message should be reported:

smash error: fg: job-id <job-id> does not exist

If fg was typed with no arguments (without job-id) but the jobs list is empty then the following error message should be reported:

smash error: fg: jobs list is empty

7. quit command

Command format:

```
quit [kill]
```

Description:
quit command exits the smash. Only if the kill argument was specified (which is optional) then smash should kill (by sending SIGKILL signal) all of its unfinished jobs and print (before exiting) the number of processes/jobs that were killed, their PIDs and command-lines (see the example for output formatting)

Note: You may assume that the kill argument, if present, will appear first.

Example:

```
smash> quit kill
smash: sending SIGKILL signal to 3 jobs:
30959: sleep 100&
30960: sleep 200&
30961: sleep 10&
Linux-shell:
```

Error handling:

If any number of arguments (other than kill) were provided with this command, they will be ignored.

8. Kill command

Command format:

```
kill -<signum> <jobid>
```

Description:
Kill command sends a signal whose number is specified by <signum> to a job whose sequence ID in jobs list is <job-id> (same as job-id in jobs command) and prints a message reporting that the specified signal was sent to the specified job (see example below).

Example:

```
smash> kill -9 1
signal number 9 was sent to pid 30985
smash>
```

Error handling:
If the syntax (number of arguments or the format of the arguments) is invalid, then an error message should be printed as follows:

smash error: kill: invalid arguments

If job-id was specified with a job id which does not exist, then the following error message should be reported:

smash error: kill: job-id <job-id> does not exist

If the kill system call fails (includes cases where the signal number falls outside the expected range), then report the failure using `perror` (as described in **Error Handling** section).

9. alias command

Command format:

```
alias <name>='<command>'
```

Description:

The alias command is used to create an alias, which is a shortcut that allows a string to be substituted for a command. This can include the command itself and its parameters, but it cannot replace parameters alone.

Where <name> is a case-sensitive, user-defined string that can include only letters (a-z, A-Z), numbers (0-9), and underscores (_). It must not be an internal reserved keyword of the shell (eg., `quit`, `lisdir` etc…)

If no arguments are provided, list all current aliases, each on separate line.

Example:

```
smash> alias ll='ls -l'
smash> ll
total 0
-rw-r--r-- 1 user user 0 Jan 1 00:00 file.txt
smash> alias s100='sleep   100  '
smash> alias
ll='ls -l'
s100='sleep   100  '
smash> alias s='sleep'
smash> s 3
```

After 3 seconds:

```
smash>
```

Note: an alias can also appear within **Special Commands**, which we will discuss later.

Error handling:

If the alias name conflicts with an existing alias or a reserved keyword, then the following error message should be reported:

smash error: alias: <name> already exists or is a reserved command

If the syntax is invalid (Including non-legal <name>), then an error message should be printed as follows:

smash error: alias: invalid alias format

Notes:

- No need to check if <command> is legal.
- You may use the following regex expression to validate format: ^alias [a-zA-Z0-9_]+='[^']*'$
- Recursive aliases will not be tested.

10. unalias command

Command format:

```
unalias <name_1> <name_2> …
```

Description:

The unalias command removes the specified aliases separated by spaces.

Example:

```
smash> alias ll='ls -l'
smash> unalias ll
smash>
```

Error handling:
If one or more of the provided alias names do not exist, the deletion process stops **at the first invalid occurrence**, then the following error message is reported:

smash error: unalias: <name> alias does not exist

where <name> is the name of the first non-existing alias.
If no arguments are provided, then an error message should be printed as follows:

smash error: unalias: not enough arguments

Built-in commands in background:

In this environment, built-in commands should disregard the `&` symbol, so running a command with `&` to send it to the background won't work with built-in commands. This means that commands like `pwd&` and `pwd` would have **the same effect**.

Note: We will not test **built-in commands** that accept optional parameters using the `[]` notation if the `&` symbol can be used as a valid input to replace the optional parameter. For example, this applies to the `chprompt &` command.

External commands

Besides the built-in commands, the smash should support executing external commands that are not part of the built-in commands. External command is any command that is not a built-in command or a special command.

The smash should execute the external command and wait until the external command is executed.

Command line:

```
<command> [arguments]
```

Where:
`command` is the name of the external command/executable.

The arguments are optional, i.e., if they exist in the external command line then they should be passed as arguments of the external command. (**Reminder:** you may assume that the number of arguments is up to **20**).

How to run:

1) Simple external command: The external command can be provided as a simple command to run some executable with its arguments, for example:

```
a.out arg1 arg2
```

For this type of external commands, you **MUST** run them explicitly using one of the exec family of system calls. In other words, you are not allowed to run these commands by launching an instance of bash and feeding it the command. Note that all the commands you know should work using this mechanism (i.e. sleep, ls,…) as long as they are "simple external commands"

Hint: Consider using a function that searches for and executes commands from the PATH environment variable to handle both executable commands and file paths.

2) Complex external command: On the other hand, the external command could be provided in a complex way, i.e., with the use of one of the following special characters (wildcards): "*" and "?".

For example:

```
rm *.txt
```

For this type of external commands, instead of loading and running the given binary itself, you **MUST** run the external command using bash, which already has the mechanism to parse those complex commands and execute them. This can be done by executing bash (using one of the exec family of functions) with your received <command> [args] in the following form:

```
bash -c "<command> [args]"
```

For example:

```
bash -c "rm *.txt"
```

In this case, smash will run a new instance of bash process while passing ["-c", "rm *.txt"] as the command line arguments to the new bash process. The new instance of bash will run the complex-external-command transparently to you, which will first parse the given command "rm *.txt" and then execute it in a forked child.

note[1]: any external command that contains one of the following special characters ["*", "?"] should be considered as a complex external-command and executed through a new instance of bash as explained above.

note[2]: running bash means calling one of the exec family of functions on "/bin/bash" binary, namely, "/bin/bash" should be provided to the exec system call as the path argument and ["-c", "complex-external-command"] should be provided as the arguments list to the bash binary, which is about to be executed.

note[3]: no, you don't have to learn bash language for this assignment.

<u>External commands in background:</u>

External commands can be executed in the background as in most Linux shells.

If a "&" sign is added to the end of a command line, then this command should be executed in the background.

For example:

```
ls -l &
```

When an external command ends with "&" (ignoring white spaces around it) it should be executed as explained above (in the external commands section) with a minor change that smash should not wait for the command completion.

The command being executed in the background should be added to the **Jobs list** (as we saw `sleep&` example in the jobs command).

Note: The "&" may or may not have white spaces around it.

Special commands

In this section, there are 2 bonus questions. Your maximum grade for this assignment can be 115 points.

1) IO redirection

Your smash code should support simple IO redirection. You can assume, for simplicity, that each typed command could have up to one character of IO redirection followed by a string. IO redirection characters that your smash should support: `>` and `>>`.

How to use redirection features:

- `command **>** output-file`: using the `**>**` redirection character (as in this example) causes the command stdout file-descriptor to be redirected to output-file, and writes any output produced by `command` to the given output file. If the output file does not exist then it creates it, and if it exists then it **overrides** its content.
- `command **>>** output-file`: the same as `>` except that using `>>` will **append** command output to the given output file if it exists. If the output-file does not exist, then `>` and `>>` will produce the same result.

Notes:

- To make the implementation easier for you, your I/O redirection commands won't be tested with commands whose implementation includes the wait system call.
- Unlike the real shell, commands containing I\O directions should ignore the `&` symbol, and they cannot be killed (meaning they will not be tested for receiving signals). This is done for simplification purposes only; a real shell should be able to handle those things.

Example:

```
smash> showpid > pid.txt
smash> cat pid.txt
smash pid is 27882
smash> showpid >> pid.txt
smash> cat pid.txt
smash pid is 27882
smash pid is 27882
smash> ls -l > ls.txt
smash>
```

2) Pipes (Bonus 10 Points)

You can add pipe functionality to your `smash`. This should work similarly to redirection commands. The pipe characters that your `smash` should support are `|` and `|&`. You may assume only one type of pipe will appear in each command with a single instance.

How to use pipe features:

- `command1 | command2`: Using the pipe character `|` will produce a pipe that redirects `command1`'s `stdout` to its write channel and `command2`'s `stdin` to its read channel.
- `command1 |& command2`: Using the pipe character `|&` will produce a pipe that redirects `command1`'s `stderr` to the pipe's write channel and `command2`'s `stdin` to the pipe's read channel.

Notes:

- As with I/O redirection, pipe commands will be tested with simple commands, won't be run in the background, and won't be sent signals.
- A command won't include both I/O redirection and a pipe.

Example:

```
smash> cat ls.txt
total 140
-rwxrwxrwx 1 root root 6376 Nov 12 10:52 Makefile
-rwxrwxrwx 1 root root 146 Nov 12 14:11 ls.txt
-rwxrwxrwx 1 root root 38 Nov 12 14:11 pid.txt
-rwxrwxrwx 1 root root 82832 Nov 12 13:41 smash
smash> cat ls.txt | grep Makefile
-rwxrwxrwx 1 root root 6376 Nov 12 10:52 Makefile
smash>
```

3) listdir command

Command format:

```
listdir [directory-path]
```

Description:

The listdir command lists all files and subdirectories within the specified directory (including hidden ones) in a tree-like structure with TAB indentation.
Use recursion to display files and directories in a hierarchical format.
The contents will be listed in alphabetical order by name, with directories appearing first.
If no directory path is provided, it lists the contents of the current working directory.

Example:

```
smash> listdir /home/user
documents
    secret files
            .hidden file
    file1.txt
    file2.txt
pictures
    photo1.jpg
    photo2.jpg
compressed.zip
```

Error handling:

If more than one argument is provided, the following error message should be printed:

```
smash error: listdir: too many arguments
```

If the specified directory does not exist or cannot be accessed, `perror` should be used to print a proper error message.

4) whoami command

Command format:

```
whoami
```

Description:

The whoami command displays the current user's username and their home directory.

Example:

```
smash> whoami
johndoe /home/johndoe
```

Error handling:

If any number of arguments were provided with this command, they will be ignored.

Note: You may Not assume that the home directory and the username do share the same name.

5) netinfo command (Bonus 5 Points)

Command format:

```
netinfo <interface>
```

Description:

You can add network analysis functionality to your `smash`. The netinfo command provides basic network configuration information for a specific network interface.

Example:

```
smash> netinfo eth0
IP Address: 192.168.1.100
Subnet Mask: 255.255.255.0
Default Gateway: 192.168.1.1
DNS Servers: 8.8.8.8, 8.8.4.4
```

Error handling:

If no interface is specified, print the following error:

```
smash error: netinfo: interface not specified
```

If the specified interface is invalid or does not exist, print:

```
smash error: netinfo: interface <interface> does not exist
```

Handling signals

Your smash should support catching Ctrl+C signals and handle them as will be explained.

Ctrl+C causes the shell to send SIGINT to the process running in the foreground. Note that from the Linux shell's perspective, the smash is the process running in the foreground, not the process currently running within your smash. Therefore, SIGINT is supposed to be sent to the smash's main process.

Your smash should route this signal to the running process in the foreground. If there is no process running in the foreground, then your smash should ignore it.

When Ctrl+C is pressed, your smash should perform the following actions:

- Print the following message:
  ```
  smash: got ctrl-C
  ```
- Send SIGKILL to the process in the foreground. If no process is running in the foreground, then no signal will be sent.
- Print the following information:
  ```
  smash: process <foreground-PID> was killed
  ```

Example:

```
smash> sleep 1000
^Csmash: got ctrl-C
smash: process 31951 was killed
smash> sleep 1000
smash>
```

Important Notes

setpgrp

Please note that you need to use the setpgrp system call to change the group ID of all your forked children. This is necessary because the Linux shell may send SIGINT (in case of Ctrl+C) to your smash and all its children. This can happen because the Linux shell sends signals to all processes that share the same group ID.

When you call `fork`, the created process (the child process) will have the same group ID as its parent unless you change it through the `setpgrp` system call. You need to call `setpgrp` right after `fork` in the child code. For example:

```
pid = fork();
if (pid == 0) {
    // Child process code goes here
    setpgrp();
    ...
} else {
    ...
}
```

Error Handling

- If multiple error messages are applicable to the occurred error (e.g., entering the command `fg 8 a` with fewer than 8 jobs), print the first error message specified in the instructions.
- If a system call fails, your smash should use the `perror` function to report the failure. The error message (to be provided to `perror`) should be as follows:

  ```
  smash error: <syscall name> failed
  ```

  Where `syscall name` is the name of the called method to execute the system call. For example, if a call to `fork` failed, it should report the failure this way:

  ```
  perror("smash error: fork failed");
  ```

- **All error messages printed in red throughout this document should be printed to `cerr` and NOT `cout`.**

Important Notes and Tips

- The assignment will be graded by automated tests. Write your own tests to ensure that your system works according to the specifications provided.
- **Pay close attention to the print formats specified throughout the assignment. The tests are automated, and missing a space in the output can cost you points.**
- Use the same setup required for HW0.
- First, try to understand exactly what your goal is.
- Determine which data structures will serve you in the easiest and simplest way. Feel free to use any library for algorithms or data structures, including `std::vector`, `std::list` and `std::string`.
- Do not use `fstream`, `ofstream`, `opendir`, `readdir`, `closedir`, or any similar libraries that allow bypassing the implementation with basic syscalls. **Failure to follow this guideline will result in point deductions.**
- Write your own tests. Your assignment will be checked with our test program.
- You are not obligated to use the given skeleton code, but you must write your solution in C/C++ and submit a Makefile with it. If you do not use the given skeleton, ensure that your Makefile contains the rule `make smash`, which compiles all your files into an executable called `smash`. <u>If your code does not follow these rules, you will not receive a grade!</u>

  We sincerely promise that there is no need to add more files or change the Makefile to complete the assignment.
- Make sure to write your IDs in the provided Makefile.
- Start working on the assignment **as soon as possible**. The deadline is final, and NO postponements will be given.
- **Any form of academic dishonesty will result in strict disciplinary action.**

Questions & Answers

- The Q&A for the exercise will take place at a public forum Piazza **<u>only</u>**. Please **<u>DO NOT</u>** send questions to the private email addresses of the TAs.
- Critical updates about the HW will be published in pinned notes in the piazza forum. These notes are <u>mandatory</u>, and it is your responsibility to be updated.

A number of guidelines to use the forum:

- Read previous Q&A carefully before asking the question; **repeated questions will probably go without answers.**
- Be polite, remember that course staff does this as a service for the students.
- You're not allowed to post any kind of solution and/or source code in the forum as a hint for other students; In case you feel that you have to discuss such a matter, please come to the reception hour.
- When posting questions regarding **hw1**, put them in the **hw1_wet** folder.

Late Days

- Please **<u>DO NOT</u>** send postponement requests to the TA responsible for this assignment. Only the **<u>TA in charge</u>** can authorize postponements. In case you need a postponement, please fill out the attached form:

  https://forms.office.com/r/exT9LqQbeV?origin=lprLink

Submission Instructions

You should create a zip file (use **zip only**, not gzip, tar, rar, 7z or anything else) containing the following files:

1) All source files you wrote **with no subdirectories of any kind.**
2) A file named **submitters.txt** which includes the ID, name and email of the participating students. Use the following format:

```
Linus Torvalds linus@gmail.com 234567890
Ken Thompson ken@belllabs.com 345678901
```

Important Note:

Ensure that the zip file structure is exactly as outlined. The zip should contain only the specified files, **without directories**. You can create the zip by running:

```
zip XXX_YYY.zip <source_files> submitters.txt
```

The zip should look as follows:

```
zipfile -+                    |
Commands.h
Commands.cpp
signals.h
signals.cpp
smash.cpp
Makefile
submitters.txt
              |
              +- other files
```

Important Note:

When you submit, **retain your confirmation code and a copy of the file(s)**, in case of a technical failure. Your confirmation code is **the only valid proof** that you submitted your assignment when you did.

Good luck and have fun!

- The Course Staff