TECHNION
Israel Institute
of Technology

Computer
Science
Department

# Operating Systems - 234123

# **Homework Exercise 1 - Wet**

Teaching Assistant in charge:
**Ori Ben-Zur**

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 [Wikipedia].

There are few well-known shells on Unix systems, one of the most used is Bash. Bash is a command processor that typically runs in a text window. Users can interact with the Bash shell by typing commands that cause some actions.

In this assignment, you are going to implement a "**smash**" (small shell) which will behave like a real Linux shell but it will support only a limited subset of Linux shell commands.

**Please read the whole document before you start working.**

## 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 (command.cpp), a shell code for smash.cpp, and a Makefile. You must complete the code found in command.c and add signal handling, the functions that handle and define the signal handling routine will be in the file siganls.cpp.

The program will work as follows:

- The program waits for commands that will be typed by the user and executes them (and back forth).
- The program can execute a small number of built-in commands, the commands will be listed below.
- When the program receives a command that is not one of the built-in commands (we call it an external command), it tries to run it like a normal shell. The way to run external commands will be described later.
- If one of the built-in commands was typed but with invalid parameters, the following error message should be printed:

  **smash error : > "command-line"**

  where command-line is the command line as typed by the user (note that the error message should be wrapped with double quotes). you'll see examples later.
- Whenever an error occurs a proper error message should be printed, and the program should get back to parse and execute the next typed commands.
- While the program is waiting for the next command to be entered, it should print the following text (though it could be changed, see chprompt command):

  **smash>**

Assumptions - you may assume the following:

- Each command appears on a separate line and cannot exceed 80 characters.
- You can assume that the number of arguments is up to 20.
- Any number of spaces can be used between words (in the same command line) and at the beginning of the line.
- Your smash should support up to 100 processes that can run simultaneously.

## 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 can be considered as features that the shell proposes for its users. Built-in commands usually run from the same code of the shell (same process) and they should not be forked and executed. Generally, built-in commands are simple and should not execute some external executable to achieve their goals (in most cases they will just run some query system call or maintain some internal data structures of the shell).

External commands usually require running an external executable. To that end, the shell runs external commands in a child process (i.e., shell will `fork` and `execv` to run these commands).

### What are jobs?

In shell terminology, a job is a process that the shell manages (processes forked by the shell process). Each job has its own job ID (which is assigned to the job by shell once it's inserted to the jobs list and shall not be changed). Since the job is also a separate process (forked by the shell) it also has a process ID (`PID`) in addition to the job ID. Please do not confuse between the job ID, which the shell assigns to the job, and the process ID (`PID`) which is assigned to the process by the kernel.

A shell job can be in one of three available 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 (which runs in the foreground and the shell waits for its completion before allowing the next commands to be typed).

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. Running a job in the background means that it will not occupy the terminal window and the shell prompt will be displayed immediately to allow users to type new commands even though the background job is still running. Running a job in the background means that the shell will run it without waiting for its completion (without using a blocking wait system call). This is called a background job.

3) Stopped:

   Users can stop running jobs in the foreground by pressing Ctrl+Z. Stopped jobs will remain stopped (their execution will be halted/stopped) until they get signaled with SIGOCONT, then they will resume their execution.

### What is the jobs list?

Any job that either (1) sent to the background (using the ampersand symbol *&*) or (2) stopped (by pressing *Ctrl+Z*) will be added to the **jobs list**, for 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 controlled by it via the jobs command.

As you will see later, jobs can be removed from the jobs list due to *fg* command which chooses a job from the jobs list to be run in the foreground (see *fg* command), or because it is simply finished. If a job was removed from the jobs list due to *fg* command, it can be brought back to it with a *Ctrl+Z*.

Please refer to bg, fg, jobs commands and signal handling for further details on how to manipulate and manage the jobs list.

**When should you delete finished jobs from the jobs list?**

Before executing any command, before printing the jobs list (see jobs command), before adding new jobs to the jobs list.

**Assigning the job-id**

The job gets its job-id on its first insertion into the jobs list and shall not be changed from that point forward. The job is given a job id as follows:

$$Job\ id := maximal\ job\ id\ in\ jobs\ list + 1$$

And if the list was empty then $Job\ id := 1$ .

Please make sure you delete all finished jobs before adding a new job to the jobs list.

## Built-in commands:

Your smash should support and implement a limited number of shell commands (features). The commands you need to support will be listed below with a detailed description. Please note that these commands (the built-in commands) should be executed from the smash process (i.e., you should not fork the smash process to run them, but you will run them from smash code).

Here are the details of 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 were sent, then the prompt shall be reset to smash. If more than one parameter was sent, then the rest shall be ignored.

Note that this command will not change the prompt in error messages that we will see 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 then they will be ignored.

### 3. pwd command

**Command format:**

```
pwd
```

**Description:**

pwd command has no arguments.

pwd prints the full path of the current working directory. In the next command (cd command) will explain how to change the current working directory.

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 pwd then they will be ignored.

## 4. cd command

**Command format:**

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

**Description:**

Change directory (cd) command receives a single argument <path> that describes the relative or full path to change the current working directory to it. There is a special argument that cd can get which is "-". If "-" was specified as the only argument of cd command then it will change the current working directory to the last working directory. That means, if the current working directory is X, then cd was executed to change the current working directory to Y, and then cd was called (again) with "-" then it should go back and set the current working directory to X.

You may use `chdir` system call to change the current working directory.

**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:

1. unfinished jobs (which are running in the background).
2. stopped jobs (which were stopped by pressing Ctrl+Z while they are running).

the list should be printed in the following format:

if the job was stopped: (see stopped jobs def. above)

[<job-id>] <command> : <process id> <seconds elapsed> (stopped)

else

[<job-id>] <command> : <process id> <seconds elapsed>

where <seconds elapsed> is the seconds elapsed since the job was inserted to the jobs list (hint: think about using `time()` system call and `difftime()` library function).

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

Please make sure you delete all finished jobs before printing the jobs list.

**note:** if the job was added again then the timer should reset.

**Example:**

```
smash> sleep 100&
smash>    sleep 200
^Zsmash: process 30902 was stopped
smash> jobs
[1] sleep 100& : 30901 18 secs
[2]    sleep 200 : 30902 11 secs (stopped)
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, then they will be ignored.

## 6. fg command

**Command format:**

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

**Description:**
fg command brings a stopped process or 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 sends to the required job SIGCONT signal and 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 300&
smash>    sleep 400&
smash> sleep 500&
smash> jobs
[1] sleep 100& : 978 14 secs
[2] sleep 200& : 979 11 secs
[3] sleep 300& : 980 8 secs
[4]    sleep 400& : 981 6 secs
[5] sleep 500& : 982 1 secs
smash> fg 5
sleep 500& : 982
```

**Error handling:**

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
```

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
```

## 7. bg command

**Command format:**

```
bg [job-id]
```

**Description:**

bg command resumes one of the stopped processes in the background. bg command should first print the command line of the job to be resumed and only then to resume it (by sending the SIGCONT signal) and continue running it in the background (i.e., there is no need to `wait` for it).

The job-id argument is an optional argument. If it is specified, then the specific stopped job whose sequence number (as printed in jobs command) should be resumed in the background. If the job-id argument is not specified, then the last stopped job (in the jobs list) should be selected to continue running it in the background (the stopped job with the maximal job-id).

**Side effects:** After resuming a stopped job in the background, the stopped mark should be removed from the jobs list.

**Example:**

```
smash> bg 4
sleep 100 : 30986
```

**Error handling:**

If job-id was specified but does not exist in the jobs list, then the following error message should be reported:

```
smash error: bg: job-id <job-id> does not exist
```

If job-id exists but it is for a job which is already running in the background (not stopped), then the following error message should be reported:

```
smash error: bg: job-id <job-id> is already running in the background
```

If bg was typed with no arguments (without job-id) but jobs list does not contain any stopped job to resume in the background, then the following error message should be reported:

```
smash error: bg: there is no stopped jobs to resume
```

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: bg: invalid arguments
```

## 8. quit command

**Command format:**

```
quit [kill]
```

**Description:**

quit command exits the smash. If the kill argument was specified (which is optional) then smash should kill (by sending SIGKILL signal) all of its unfinished and stopped jobs before exiting.

If the kill option was specified then it should print (before exiting) the number of processes/jobs that were killed, their PIDs and command-lines (see the example for output formatting)

**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 arguments were specified other than kill then they will be ignored.

## 9. Kill command (bonus 5 points)

**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 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 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 the kill system call fails, then report the failure using perror (as described in Error Handling section).

## Built-in commands in background:

Unlike the real shell, all built-in commands should **ignore** the & symbol, meaning they can't be run in the background as it would make things a bit easier for you.

Therefore, for example, pwd& and pwd would yield the same effect.

## 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 (see special commands).

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 using…

**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.

### External commands in background:

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 regular 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).

# Special commands:

## 1) Pipes and IO redirection

Your smash code should support simple IO redirection and pipes features. You can assume, for simplicity, that each typed command could have up to one character of pipe or IO redirection. IO redirection characters that your smash should support: “>” and “>>”. Pipe characters that your smash should support are: “|” and “|&”.

How to use these pipes and 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.
- “command1 | command2”: using the pipe character “|” will produce a pipe which redirects command1 stdout to its write channel and command2 stdin to its read channel.
- “command1 |& command2”: using the pipe character “|&” will produce a pipe which redirects command1 **stderr** to the pipe’s write channel and command2 stdin to the pipe’s read channel.

**Note:** In order to make the implementation a bit easier for you, your pipe and I/O redirection commands will only be tested with simple commands, especially not with commands that their implementation includes the wait system call.

**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> 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 pid.txt | grep smash
smash pid is 27882
smash pid is 27882
smash> showpid | grep smash
smash pid is 27882
smash>
```

**Pipes and I/O commands notice:**

Unlike the real shell, commands containing pipes and I\O directions should **ignore** the & symbol, and they cannot be stopped or 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.

## 2) Find-and-replace ("fare") command (optional 5 points)

**Command format:**

```
fare <file-name> <source> <destination>
```

**Description:**

The find-and-replace finds and replaces every instance of the word "source" to the word "destination" and then prints how many instances of the string "source" it had replaced (exact format specified below).

Note that the fare command should scan the file from start to finish.

The word source doesn't have to be separated by spaces from both sides.

If the "fare" command files mid-way, the file should remain as it was before invoking "fare".

Please note that this is a built-in command and not an external command, which means you must implement it inside smash code, i.e. **do not** run any binary with `execv` or `fork` when implementing "fare". Using `execv` or `fork` will cause **points reduction**.

**Example:**

```
smash> echo "world helloworld" > temp.txt
smash> fare temp.txt world bob
replaced 2 instances of the string "world"
smash> cat temp.txt
bob hellobob
```

**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: fare: invalid arguments
```

## 3) setcore command (optional 7 points)

**Command format:**

```
setcore <job-id> <core_num>
```

**Description:**

The setcore command sets the core that the job with the id "job-id" runs on.

Hint: this can be implemented using a system call.

Please note that this is a built-in command and not an external command, which means you must implement it inside smash code, i.e. **do not** run any binary with `execv` or `fork` when implementing "fare". Using `execv` or `fork` will cause **points reduction**.

You may find the "top" and "ps" shell commands helpful for debugging. The column with the header "PSR" specifies the core that the process

**Example**

```
smash> sleep 100 &
smash> sleep 200 &
smash> jobs
[1] sleep 100& : 115 9 secs
[2] sleep 200& : 935 6 secs
smash> ps -o pid,psr,command
PID PSR COMMAND
[…]
115 0        sleep 100
935 0        sleep 200
[…]
smash> setcore 1 4
smash> setcore 2 1
smash> ps -F
PID PSR COMMAND
[…]
115 4        sleep 100
935 1        sleep 200
[…]
```

**Error handling:**

If the job doesn't exist, then an error message should be printed as follows:

```
smash error: setcore: job-id <job-id> does not exist
```

If the core number is invalid, then the following error message should be printed:

```
smash error: fare: invalid core number
```

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: fare: invalid arguments
```

## 4) timeout command (optional 7 points)

**Command format:**

```
timeout <duration> <command>
```

**Description:**

sets an `alarm` for 'duration' seconds, and runs the given 'command' as though it was given to the smash directly, and when the time is up it shall send a SIGKILL to the given command's process (unless it's the smash itself).

Notes:

- The alarm setup should be done from the smash process itself so it could receive the alarm signal.
- The killing should be done in the alarm signal handler (SIG_ALRM handler).
- Consider setting up `SIG_ALRM` handler with `sigaction` instead of `signal`, and use `SA_RESTART` flag (think about why is that important, what happens when a process receives a signal while it's waiting?)
- When killing a process after it had timed out, the following should be printed:

  ```
  smash: [command-line] timed out!
  ```

  where command-line command line received, unprocessed, of the process that timed out.
- We suggest you save all your timed commands in a list with a timestamp, duration and pid.

**Example:**

```
smash> timeout 3 sleep 10
```

and after 3 seconds will get:

```
smash> timeout 3 sleep 10
smash: got an alarm
smash: timeout 3 sleep 10 timed out!
smash>
```

**Another Example:** (running in the background)

```
smash> timeout 3 sleep 10 &
smash>
```

and after 3 seconds will get:

```
smash> timeout 3 sleep 10 &
smash> smash: got an alarm
smash: timeout 3 sleep 10 & timed out!
```

## Handling signals:

Your smash should support catching Ctrl+C, Ctrl+Z and `SIG_ALRM` and handle them as will be explained.

Ctrl+C causes the shell to send SIGINT to the process running in the foreground, and Ctrl+Z causes the shell to send SIGTSTP to the process running in the foreground. (note that from the point of view of Linux shell, the smash is the process who is running in foreground and it's not the process you are currently running from your smash, so the SIGINT is supposed to be sent to the smash main process).

Your smash should route these two signals to the running process in the foreground. If there is no process running in the foreground, then your smash should ignore them.

meaning:

Pressing Ctrl+C will send SIGINT to your smash, and your smash should route it (send SIGKILL signal) to the process which is currently running in the foreground, which will kill it.

Pressing Ctrl+Z will send SIGTSTP to your smash, and your smash should route it and stop the process running in the foreground (i.e., send SIGSTOP signal to the process running in foreground).

The stopped process should be added to the jobs list and marked as stopped as explained before (see jobs command).

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

- 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**

When Ctrl+Z is pressed, your smash should do the following:

- print the following message: **smash: got ctrl-Z**
- add the foreground process to the jobs list. If no process is running in the foreground, then nothing will be added to the jobs list.
- send SIGSTOP 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 stopped**

**Example:**

```
smash> sleep 1000
^Csmash: got ctrl-C
smash: process 31951 was killed
smash> sleep 1000
^Zsmash: got ctrl-Z
smash: process 31952 was stopped
smash> jobs
[1] sleep 1000 : 31952 2 secs (stopped)
smash>
```

when receiving a `SIG_ALRM`, your smash should do the following:

- print the following information:

  ```
  smash: got an alarm
  ```

- search which command caused the alarm, send a `SIGKILL` to its process and print:

  ```
  smash: [command-line] timed out!
  ```

## Important notes:

**setpgrp**

Please note that you need to use `setpgrp` system call to change the group ID of all of your forked childs. This is necessary due to the fact that Linux shell may send SIGINT and SIGTSTP (in case Ctrl+C and Ctrl+Z was pressed) to your smash and **all** of his children. This could happen because the Linux shell sends the signals to all the processes that share the same group ID.

When you call fork() the created process (the child process) will have the same group ID as his parent unless you change it through `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 a system call fails then your smash should use 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 then should report about 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

- **Optional and bonus exercises**: This assignment has 3 optional exercises and 1 bonus exercise. You **must** complete one of the optional assignments to get a 100 in this assignment (the optional exercises that are worth 7 points will grant you a bonus of 2 points)
- The assignment will be graded by auto tests. Write your own tests and be sure that your system is working according to the above specification.
- **Pay close attention to the print formats that are given throughout the assignment. The tests are automatic, and you don't want to lose points because you missed a space in the output.**
- Use the same setup you have been required to use for HW0.
- First, try to understand exactly what your goal is.
- Figure out which data structures will serve you in the easiest and simplest way. Feel free to use std::vector and std::list.
- Write your own tests. We will check your assignment with our test program.
- You are not obligated to use the given skeleton at all, although you are required to write your solution in c / c++ and submit a Makefile with it. If you are not using the skeleton you are given, you should make sure that your makefile contains the rule: make smash, which compiles all your files into an executable called smash. If you will not, or your code will not compile with the default makefile you are getting in the skeleton, you will not get a grade!

What does this all mean?

PLEASE USE THE GIVEN SKELETON AND DON'T ADD MORE FILES OR CHANGE THE MAKEFILE

If you will, we will have a headache, but so will you!

We sincerely promise that there is no need to add more files or change the makefile in order to solve the assignment.

- Start working on the assignment **as soon as possible**. The deadline is final, NO postponements will be given.

## Questions & Answers

- The Q&A for the exercise will take place at a public forum Piazza **only**.Please **DO NOT** 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 mandatory, 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 not be answered**.
- 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** folder.

## Late Days

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

https://forms.office.com/r/X8SqhiTNwe

## Submission

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. The following format should be used:

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

**Important Note:** Make the outlined zip structure exactly. In particular, the zip should contain only the X 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