HW #1. Process & IPC

Part A: Get Started

In this part, you will learn the basic usage of Unix API for multi-process programming, including `fork()`, `pipe()`, and more. To get started on Unix multi-process programming, we recommend you first go through the tutorial (i.e. Part A), then compile and execute the example program to know how they work.

[Advice] If you have been familiar with these topics, just skipping this part and go for attacking Part B directly!

Get the example code

You can get all the sample code on GitHub by executing

```
git clone https://github.com/hungys/NCTU_OS_2015_HW1.git
```

We may keep updating the tutorials, sample code and HW spec. To check if your copy is up-to-date, execute `git pull` without cloning again.

Process Creation

```
pid_t fork(void);
```

To create a new process we use the `fork()` system call. The fork system call actually clones the calling process, with very few differences. The clone has a different process id (PID) and parent process id (PPID).

The return value of the `fork()` is the only way that the process can tell if it is the parent or the child. The function returns the **PID of the child** to the parent and **0** to the child. After the function call executed, both parent process and child process will continue to run the next line of code, and you can use the return value to distinguish them.

To see more details about `fork()`, type `man fork` on the terminal.

Example

[Code] example/fork.c

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

int main(int argc, char **argv) {
    pid_t childpid;

    childpid = fork();

    if (childpid < 0) {   /* cannot allocate new process */
        printf("fork failed\n");
    } else if (childpid == 0) {   /* child process */
        printf("(pid=%d) I'm child, I will sleep for 3 seconds\n", getpid());
        sleep(3);
    } else {   /* parent process */
        pid_t pid = getpid();
        printf("(pid=%d) I'm parent, and my child's pid is %d\n", pid, childpid);

        int status;
        waitpid(childpid, &status, 0);   /* wait for child process terminated */
        printf("(pid=%d) My child has been terminated\n", pid);
    }

    return 0;
}
```

Useful Funtions

- `pid_t getpid(void)` : get process id (PID) of calling process
- `pid_t getppid(void)` : get process id (PID) of the parent of calling process, a.k.a parent process id (PPID)
- `pid_t waitpid(pid_t pid, int *status, int options)` : wait for process to change state
    - The value of options is an OR of zero or more of the following constants
    - `WNOHANG` : return immediately if no child has exited
    - `WUNTRACED` : return if a child has stopped
    - `WCONTINUED` : return if a stopped child has been resumed by delivery of `SIGCONT`
    - more... see `man waitpid`

Pipes

Pipes in POSIX allow for a one-way flow of data from one process to another. A pipe is basically a circular buffer that hides in the file system. We use it in a producer-consumer fashion. One process writes to the

pipe, and blocks if the buffer becomes full. Another process reads from the pipe and blcoks if it becomes empty. A read will fail if the producer closes the pipe or dies. And a write will fail if the consumer closes the pipe or dies.

```
int pipe(int pipefd[2]);
```

To create a pipe, We simply call `pipe()` system call in the parent process, by passing it an array of two file descriptors. In Unix, a file descriptor (FD) is an abstract indicator used to access a file or other input/output resource, including `stdin` and `stdout` you are familiar with, and also the pipes and socket connections. All the FDs are maintained in a file descriptor table, you can use `close()`, `dup()`, or `dup2()` to do operation on it.

To see more details about `pipe()`, type `man pipe` on the terminal.

The following example demonstrate how to create a communication channel between the parent process and the child process.

Example

[Code] example/pipe.c

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

int main(int argc, char **argv) {
    pid_t childpid;
    int fd[2];

    pipe(fd); /* create a pipe */
    childpid = fork();

    if (childpid < 0) {
        printf("fork failed\n");
    } else if (childpid == 0) {
        dup2(fd[1], 1);   /* replace stdout */
        close(fd[0]);
        close(fd[1]);

        printf("HelloFromChild(pid=%d)!", getpid());   /* printf now writes to pip
e */
    } else {
        dup2(fd[0], 0);   /* replace stdin */
        close(fd[0]);
        close(fd[1]);

        char buffer[32];
        scanf("%s", buffer);   /* scanf now reads from pipe */
        printf("(pid=%d) Received: %s\n", getpid(), buffer);

        int status;
        waitpid(childpid, &status, 0);
    }

    return 0;
}
```

Useful Functions

These two system calls create a copy of the file descriptor oldfd.

- `int dup(int oldfd)` : uses the lowest-numbered unused descriptor for the new descriptor
- `int dup2(int oldfd, int newfd)` : makes newfd be the copy of oldfd, closing newfd first if necessary

FIFO

Named pipe (a.k.a. FIFO) is same as anonymous pipe, but creating a **special file**. This file makes communication between two **unrelated** processes, which have no parent-child relation, become possible.

```
int mkfifo(const char *pathname, mode_t mode)
```

To use FIFO, we use `mkfifo()` system call to create a named pipe file, which is just a special file in the file-system. And then you can use it just as a regular file with `open()`, `read()` and `write()` system calls.

To see more details about `mkfifo()`, type `man mkfifo` on the terminal.

The following example demonstrate how to use FIFO. After compiling the sample code, you can first use `./fifo -c <fifo_name>` to create a FIFO, then execute `./fifo -r <fifo_name>` and `./fifo -w <fifo_name>` on two terminal seperately. Now these two unrelated processes are able to communicate with each other.

Example

[Code] example/fifo.c

```c
#include <unistd.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>

int main(int argc, char **argv)
{
    if (argc != 3)
    {
        printf("fifo: usage: %s [-c/-w/-r] <fifo_name>\n", argv[0]);
        exit(0);
    }

    if (strcmp(argv[1], "-c") == 0) {
        if (mkfifo(argv[2], 0666) < 0) {   /* create a fifo */
            perror("mkfifo failed");
        } else {
            printf("mkfifo successed\n");
            exit(0);
        }
    } else if (strcmp(argv[1], "-r") == 0) {
        int fd;
        fd = open(argv[2], O_RDONLY);   /* open a special file */
        char buf[1024];
        int len;
        while ((len = read(fd, buf, sizeof(buf))) > 0) {   /* operate as a file */
            printf("Read %d byte(s): %s", len, buf);
            memset(buf, 0, sizeof(buf));
        }
    } else if (strcmp(argv[1], "-w") == 0) {
        int fd;
        fd = open(argv[2], O_WRONLY);
        char buf[1024];
        while (1) {
            memset(buf, 0, sizeof(buf));
            fgets(buf, sizeof(buf), stdin);
            write(fd, buf, strlen(buf));
        }
    }

    return 0;
}
```

Signal Handling

Signals allow one process to communicate the occurance of an event to another process. The number of the signal indicates which event occured. No other information can be communicated via signals. Some common ones include,

- **SIGCHLD:** the signal is sent to a process when a child process terminates, is interrupted, or resumes after being interrupted.
- **SIGCONT:** the signal instructs the operating system to continue (restart) a process previously paused by the SIGSTOP or SIGTSTP signal.
- **SIGINT:** the signal is sent to a process by its controlling terminal when a user wishes to interrupt the process.
- **SIGKILL:** the signal is sent to a process to cause it to terminate immediately.
- **SIGTERM:** the signal is sent to a process to request its termination.
- **SIGTSTP:** the signal is sent to a process by its controlling terminal to request it to stop temporarily.
- **SIGTTIN and SIGTTOU:** the signals are sent to a process when it attempts to read in or write out respectively from the tty while in the background.
- and more on Wikipedia...

```
sighandler_t signal(int signum, sighandler_t handler);
```

The simplest way to do signal handling is using `signal()` system call. If the handler is set to `SIG_IGN`, then the signal is ignored. If the handlr is set to `SIG_DFL`, then the default action associated with the signal occurs. For instance, `signal(SIGTSTP, SIG_IGN)`.

```
int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
```

A more modern way is using `sigaction()`. The following example demonstrate how to handle the SIGINT signal.

Example

[Code] example/signal.c

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

int sigint_counter = 0;

void sigint_handler(int sig) {   /* handler body */
    sigint_counter++;
    if (sigint_counter > 3) {
        printf("Goodbye!\n");
        exit(0);
    } else {
        printf("SIGINT caught, %d more Control-C to exit!\n", 4 - sigint_counter);
    }
}

int main(int argc, char **argv) {
    pid_t childpid;

    struct sigaction sigint_action = {
        .sa_handler = &sigint_handler,
        .sa_flags = 0
    };
    sigemptyset(&sigint_action.sa_mask);
    sigaction(SIGINT, &sigint_action, NULL);

    while(1) {
        sleep(1);
    }

    return 0;
}
```

The sample program will be terminated after the fourth time you press `Control-C`.

Useful Functions

```
int kill(pid_t pid, int sig);
```

`kill()` can be used to send signal to a process. If pid is positive, then signal sig is sent to the process with the ID specified by pid. If pid is less than -1, then sig is sent to every process in the **process group** whose ID is -pid. See more details about it on `man kill`.

The following example shows a basic implementation to kill specific process or process group.

[Code] example/kill.c

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

int main(int argc, char **argv) {
    if (argc < 2) {
        printf("kill: usage: %s <pid>\n", argv[0]);
        exit(-1);
    }

    pid_t pid = atoi(argv[1]);
    if (kill(pid, SIGINT) < 0) {   /* send signal to pid */
        printf("kill: No such process\n");   /* return -1 if error */
    }

    return 0;
}
```

In Part A, we only cover the knowledge you may need in further part of this homework assignment. There are still some topics about Unix IPC not covered in this material; for instance, **message queue**, and **shared memory**.

Part B: Write your own shell

Overview

The shell (e.g. bash) is a program that is so closely associated with the operating system, that most people think of it as part of the operating system. But in reality, all the shell programs are built on top of the programming interface provided by the operating system. In this part, you are asked to write a simple shell called ***mysh*** by yourself, to provide user a command line interface to run programs. This will give you a better sense of the user-system interface used to access these resources.

Although you are not required to implement all the features provided by bash or other commercial products, *mysh* should have the following important functionalities:

- Allow the user to execute programs, from executable files on the file-system, as background (e.g. `./prog &`) or foreground jobs.
- Provide basic job-control functions, including commands `bg`, `fg`, and `kill` for changing the foreground/background status of currently running jobs and job suspension/continuation/termination.
- Allow for the piping of several tasks (e.g. `prog_a | prog_b | prog_c`)

For the detailed specifications and requirements, please read the following parts carefully. To simplify the tasks you need to do, you don't need to implement some functions you may be familiar with but not mentioned in this document, including but not limited to I/O redirection (i.e. token `<` and `>`) and conditional commands (i.e. token `&&` and `||`).

Enviornment

You can do this assignment on any Unix system, but you should ensure your program can be run for demo on Linux workstation provided by NCTU CSCC (i.e. linux1~6.cs.nctu.edu.tw). If you want to develop and test your program on your own virtual machine, we prefer the latest LTS version of Ubuntu (Ubuntu 14.04.3 LTS).

[Important] As for the programming language, you must use **C/C++** to implement your shell. Otherwise you will lose **ALL** the points from this part.

Tasks

You don't need to implement your shell according to the following order of tasks. However, we recommend you to read all the specifications first. We provide a reference architecture for student who has no idea about how to get started.

[Useful Information] It is possible to pass all the tasks with LESS than 400 lines of code!

#1 Shell Prompt

First, mysh should print a welcome message with your student ID when it is launched. The format of welcome message should be `Welcome to mysh by <student ID>!`.

As a good shell, it is useful to provide some information in the shell prompt, including current user and current working directory. The format of prompt should be `<username> in <current working directory>`.

In front of the cursor of user input area, the shell should print `mysh> I`, suppose `I` is the cursor.

Combine the three requirements above, your shell should work as following result:

```
Welcome to mysh by 0456000!
osta in /Users/osta/mysh
mysh>
osta in /Users/osta/mysh
mysh>
......
```

[Hint] You may need to use `getcwd()` for getting current working directory, and `getlogin_r()` for getting current username.

#2 Command Parser

mysh should accept input from the user one line at a time. Your parser need to recognize program name (or command name), arguments, meta-characters including `|` for pipes, and `&` for background jobs.

Blank-space characters should be treated as delimiters, but your shell should be insensitive to repeated blank spaces. It should also be insensitive to blank spaces at the beginning or end of the command line.

Certain characters, known as *meta-characters*, have special meanings within the context of user input. These characters include `&` and `|`. Your shell can assume that these meta-characters cannot occur inside strings representing programs, arguments, or files. Instead they are reserved for use by the shell. Also, you can suppose all the arguments are separated by blank spaces; that is, you don't need to handle the case with **quotation marks** (i.e. "... ...").

The general form of command supported by mysh can be expressed as:

```
progA [argA1 argA2 ... argAN] | progB [argB1 argB2 ... argBN] | ... | progZ [argZ1
 argZ2 ... argZN] [&]
```

You are allowed to generate parser using (f)lex and yacc/bison, although we have simplify the complexity that you should be able to write by your own.

To help you pass this task, before design your parser, please consider the following scenarios carefully:

- Execute a single program: `prog`
- Execute a single program in background: `prog &`
- Execute a single program with arguments: `prog 10 200`
- Execute multiple programs in a pipeline: `progA 10 | progB | progC`
- Execute a command with unnecessary blank-space chracters: `  progA| progB|   progC  &`

#3 Internal Commands

In most cases, the user’s input will be a command to execute programs stored within a file system. We’ll call these external programs.

Your parser should also support several internal commands. If these commands are issued by the user, you should direct the shell to take a particular action itself instead of directing it to execute other programs.

mysh should support the following internal commands:

- `exit` : Kill all child processes, print a goodbye message `Goodbye!`, then exit the shell.
- `cd <path>` : Change the current working directory, and mysh should print an error message if the

destination is not existed.

There are still three commands *mysh* need to support, including `fg`, `bg` and `kill`. Those will be introduced in detail later. For `cd` and `exit`, please refer the following execution example:

```
osta in /Users/osta/mysh
mysh> cd abc
osta in /Users/osta/mysh/abc
mysh> cd xyz
-mysh: cd xyz: No such file or directory
osta in /Users/osta/mysh/abc
mysh> exit
Goodbye!
```

[Hint] You may need to use `chdir()` for changing current working directory. Understand the meaning of return values may be helpful for you to do the error-handling.

#4 Program Execution

The execution of a program is specified by a sequence of delimited strings. The first of these is the name of the executable file, and the others are arguments to be passed to the program. If the command is an error or the executable file named by the first string does not exist, *mysh* should print an error message.

Before each program be executed, mysh should print a message `Command executed by pid=<pid> [in background]` to let user know the program is running in which process.

By default, the external commands or programs are executed in foreground. When a program is executed in foreground, the shell cannot accept any new command until the program is terminated. The foreground process is allowed to take user's input (i.e. stdin) or print any message to the terminal (i.e. stdout). For example,

```
osta in /Users/osta/mysh
mysh> ./calc
Command executed by pid=2649
Please enter an expression: 1+2+3   # 1+2+3 is typed by user
The answer is 6
osta in /Users/osta/mysh
mysh> prog
Command executed by pid=2650
-mysh: prog: command not found
mysh>
```

If the command is end with a `&` token, the program should be run in background. Such process can still print any message to the terminal (i.e. stdout), but has no permission to take user's input (i.e. stdin). For

example,

```
osta in /Users/osta/mysh
mysh> ./counter &
osta in /Users/osta/mysh
mysh> Command executed by pid=2651 in background
count to 1   # the shell can accept next command immediately
count to 2
count to 3
```

Furthermore, *mysh* should be able to execute any programs from the paths defined in environment variable `PATH`.

[Hint]

- You may need to use `fork()`, `execvp()`, `waitpid()` to execute programs.
- You will lose **ALL** the points of this task if you use `system()` to execute the program.

#5 Shell Pipeline

Several program invocations can be present in a single command line, when separated by the shell meta-character `|`. In this case, the shell should fork all of them, chaining their outputs and inputs using **pipes**. For instance, the command

```
progA argA1 argA2 | progB argB1
```

should fork both progA and progB and make the output from progA go to the input of progB. This should be accomplished using a pipe IPC primative.

There is no limitation for the number of programs in a single pipeline. *mysh* should be also insensitive to repeated blank spaces, or even a program followed by a `|` token without any blank space. For example,

```
osta in /Users/osta/mysh
mysh> pwd | cat
Command executed by pid=3855
Command executed by pid=3856
/Users/osta/mysh
hungys in /Users/osta/mysh
mysh> pwd|   cat |cat   # multiple programs and repeated blank spaces
Command executed by pid=3857
Command executed by pid=3858
Command executed by pid=3859
/Users/osta/mysh
mysh>
```

[Hint] You may need to use `pipe()` for creating communication channel between processes, and `dup2()` for redirecting the input and output.

#6 Signal Handling

Through an interaction with the terminal driver, certain combinations of keystrokes will generate signals to your shell instead of appearing within stdin. Your shell should respond appropriately to these signals:

- `Control-C` generates a `SIGINT`. This should not kill your shell. Instead it should cause your shell to kill **ALL** the processes (considering a pipeline) in the current foreground job. If there is no foreground job, it should have no effect.
- `Control-Z` generates a `SIGSTOP`. This should not cause your shell to be suspended. Instead, the default behavior should cause your shell to suspend the processes in the current foreground job and start to accept next command. If there is no foreground job, it should have no effect.

See the following example,

```
osta in /Users/osta/mysh
mysh> ./counter 10   # This program will count from 1 to 10
Command executed by pid=3894
count to 1
count to 2
count to 3
^Costa in /Users/osta/mysh   # ^C means a "Control-C" is pressed
mysh> ps   # ps is an external command
Command executed by pid=3895
  PID TTY           TIME CMD
 3706 ttys001    0:00.03 -bash
 3854 ttys001    0:00.01 ./mysh   # ./counter is terminated
mysh> ./counter 10
Command executed by pid=3896
count to 1
count to 2
count to 3
^Zosta in /Users/osta/mysh   # ^Z means a "Control-Z" is pressed
mysh> ps
Command executed by pid=3897
  PID TTY           TIME CMD
 3706 ttys001    0:00.03 -bash
 3854 ttys001    0:00.01 ./mysh
 3896 ttys001    0:00.00 ./counter   # ./counter is suspended
mysh>
```

In the next task, you will implement the internal commands `bg` and `fg` to make it possible to bring back suspended jobs to background or foreground and continue to run.

[Hint]

- You may need to use `signal()` and `sigaction()` for associating handlers and a signals.
- Understand the meaning of `SIGINT`, `SIGQUIT`, `SIGTSTP`, `SIGTTIN`, `SIGTTOU`, `SIGCHLD` may be helpful.
- The disposition `SIG_DFL` and `SIG_IGN` can be used with `signal()` to associate the default action with the signal, or let the process ignores it.
- Understaing the usage of different flags (e.g. `WNOHANG`, `WUNTRACED` ...) for `waitpid()` may be helpful for you to handle the `Control-Z` case.

#7 Job Control

Before doing this task, you may need to slightly modify your program. From now on, you should consider a single command (may be a pipeline) as a **job**. To bring a job back from suspended status, you may need to notify all the processes in the same pipeline. Instead of maintaining a job/process table, you can simply associate multiple processes with a common group ID, to form a **process group**.

At most, one terminal can be associated with a process group, and such *foreground* process group is the group within a session that currently has access to the controlling terminal (i.e. stdin).

mysh should support basic job control functions. In this task, you need to add three internal commands, including `bg`, `fg` and `kill`.

- `fg <pid>` : Bring the process with specific pid to foreground. If the process was previously stopped, it should now be running. Your shell should wait for a foregound process to terminate before returning a command prompt or taking any other action.
- `bg <pid>` : Execute the suspended process with specific pid in the background.
- `kill <pid>` : If pid>0, it will kill the process with specific pid. If pid<0, it will kill all the processes with **process group** ID `-pid`.

See the following example,

```
osta in /Users/osta/mysh
mysh> ./counter 10
Command executed by pid=3894
count to 1
count to 2
count to 3
^Zosta in /Users/osta/mysh   # process is suspended
mysh> fg 3894
count to 4   # process 3894 is now in foreground again
count to 5
......
count to 10
mysh> ./counter 10
Command executed by pid=3895
count to 1
count to 2
count to 3
^Zosta in /Users/osta/mysh   # process is suspended
mysh> bg 3895
osta in /Users/osta/mysh
mysh> count to 4   # process 3805 is now running in background
count to 5
......   # since shell is in foreground, it can accept next command
```

[Hint]

- You may need to use `setpgid()` for setting process group ID, `kill()` for sending signal to specific process or process group, and `tcsetpgrp()` to make specific process group as the foreground process group.
- `SIGCONT` can be used to signal a process or process group to continue to run.

#8 Prevent Zombies

A zombie process is a process that has completed execution (via the exit system call) but still has an entry in the process table. This occurs for child processes, where the entry is still needed to allow the parent process to read its child's exit status: once the exit status is read via the `wait()` system call, the zombie's entry is removed from the process table and it is said to be "reaped". A child process always first becomes a zombie before being removed from the resource table. In most cases, under normal system operation zombies are immediately waited on by their parent and then reaped by the system – processes that stay zombies for a long time are generally an error and cause a resource leak.

In this task, you are asked to modify your program, or add some code to prevent zombie processes. A common way is to insert some code for reaping such processes at appropriate place. To verify your work, you can use `ps` command to check the snapshot of current processes.

```
osta in /Users/osta/mysh
mysh> ./counter 10 &   # the shell will not wait for the process
osta in /Users/osta/mysh
mysh> Command executed by pid=4055 in background
count to 1   # the shell can accept next command immediately
count to 2
count to 3
......
count to 10
ps   # check for process status
Command executed by pid=4056
  PID TTY           TIME CMD   # process 4055 is reaped
 3706 ttys001    0:00.03 -bash
 3854 ttys001    0:00.01 ./mysh
osta in /Users/osta/mysh
mysh>
```

[Hint]

- If you specify -1 as the pid paramter of `waitpid()`, it will wait for any child process.
- The `SIGCHLD` signal is sent to the parent of a child process when it exits, is interrupted, or resumes after being interrupted.

#9 Colorizing Your Shell

Good news! You are almost done your tasks. Now let's do something interesting to make your shell colorful. Please modify your program to colorize the bold texts in the following execution. You can choose whatever color you love, or just follow our suggestion.

- **osta** in **/Users/osta/mysh** # username: cyan, path: yellow
- **mysh>** ./calc # dark gray
- **Command executed by pid=2649** # green
- Please enter an expression: 1+2+3
- The answer is 6
- **osta** in **/Users/osta/mysh**
- **mysh>**

[Hint] Simply add some escape sequences into your `printf()` call, and the task is finished :)

#10 Celebrate!

Well done! You have finished all the tasks and your shell should now meet for all the requirements for this homework. Don't forget to double check all the requirments and grading policy in case you miss anything.

You are free to add any cool features (e.g. show current branch of git repository in the shell prompt) to your

mysh, but please make sure you don't break any functions required by this homework.

Part C: Short Answer Questions

In this part, you are asked to answer some questions about process and IPC. Some of the problems may be highly related to Part B.

1. Briefly explain how do you prevent zombie processes in Task #9 of Part B.
2. Briefly explain what did you do to ensure the signals were sent to the correct processes? For instance, when there is a process running in foreground, the `Control-C` keyboard combination should not terminate the shell.
3. If you are asked to implement I/O redirection (i.e. token `<` and `>`) in *mysh*, how will you design for it? Briefly summarize what you need to do. You don't need to provide the source code.
4. You may heard about several IPC mechanisms including pipe, FIFO, message queue and shared memory. Which mechanism is mainly used in Android framework? If the answer is not listed above, briefly explain the concept of the mechanism you answers.

Grading Policy

- Part A - 0%
  - Just for practice :)

- Part B - 80%
  - Task #1 Shell Prompt - 5%
    - username - 2%
    - current working directory - 3%

  - Task #2 Command Parser - 5%
    - insensitive to repeated blank spaces - 5%

  - Task #3 Internal Commands - 5%
    - `exit` - 2%
    - `cd` - 3%

  - Task #4 Program Execution - 20%
    - foreground execution - 7%
    - background execution - 8%
    - support unlimited amount of program arguments - 5%

  - Task #5 Shell Pipeline - 20%
    - support single pipe (i.e. `A | B`) - 10%
    - support multiple pipes (e.g. `A | B | C | D`) - 5%

        - support unlimited amount of pipes - 5%

    - Task #6 Signal Handling - 10%
        - `SIGINT (Control-C)` - 5%
        - `SIGSTOP (Control-Z)` - 5%

    - Task #7 Job Control - 15%
        - `fg` - 5%
        - `bg` - 5%
        - `kill` - 5%

    - Task #8 Prevent Zombies - 5%
    - Task #9 Colorizing Your Shell - 5%

- Part C - 20%
    - Each question counts 5%

[Hint] Total scores of Part B are **more than 80 points**. That is, you don't need to pass all the tasks to get max score. In case you may lose some points due to bugs, try your best to attack all the tasks.

Deliverables

[Important] Please organize your files correctly before you upload them to e-campus, otherwise you will lose **10 points** for penalty.

- Part B
    - A **Makefile**
    - Source files that compile, by typing `make`, into an executable of name `mysh`
    - Optionally, a file of name `README` that contains anything you wish to point out to us.
    - Put all the files into a directory named `mysh`
    - **No need to write a report**

- Part C
    - A **pdf** file containing all your answers
    - Name your pdf file as `HW1_<STUDENT ID>.pdf`

- Put all the files in `HW1_<STUDENT ID>.zip`, then upload it to e-campus.

Contact Us

Some commonly asked questions will be listed on QA.md. If you still have any question about this homework, feel free to e-mail the TA, or knock the door of EC618.

- TA: 洪聿昕 (Yu-Hsin Hung)
- E-mail: hungys.cs04g@nctu.edu.tw

Don't forget to attach your **name** and **student ID** in the e-mail, and name the subject as `[OS] HW1 Question (<STUDENT ID>)`.