세계 속의 GIST 이공계 대학

Operating System Project 3

Release Date: 2024-04-29

Due Date: 2024-05-13

GIST

Gwangju Institute of Science and Technology

0. Grading Criteria

- This assignment will be graded through the ‘make check’ command, just like Task 2. But only one difference is running ‘make check’ in **‘pintos/src/usrprog’** folder.
- During this task, some tests in the filesys folder can also be passed if a file system call is properly implemented. However, it will not be graded. Grading consists only of make checks in the usrprog folder.

- Tips :

- In this assignment, system calls die or page faults occur frequently. Since this makes debugging difficult, we recommend using GDB.
- Just like the last assignment, don't try to pass all the tests. There is no big minus point if you get a few tests fail. Rather than giving up, aim to pass as many tests as possible.

0. Grading Criteria - Rules

- Problem 1 and 2
  - Report: 1pt (Problem 3 report should be included as well)
  - Test result: # of passed tests / # of tests * 1pt for each problem

- Problem 3
  - Test result: # of passed tests / # of tests * 2pts

  - The score is graded as proportional to the number of tests being passed

- If you have any questions, please mail me at minwook-lee@gm.gist.ac.kr

1. Project #3 Introduction

- In this assignment, you need to enable programs to interact with the OS via **system calls**.
- You will mainly work in the **"*~/pintos/src/userprog*"** directory, but you will also be interacting with almost every other part of pintos
- You can build project 3 on top of your project 2 submission or you can start from fresh build. None of implemented codes in project 2 is required for this assignment.
- In the previous assignment, you compiled your test code directly into kernel, so you had to require certain specific function interfaces within the kernel. From now on, you need to test your operating system by running user programs.

1. Project #3 Introduction

- Running **more than one process** at a time is allowed. Each process consists of a single thread (multithreaded processes are not supported).

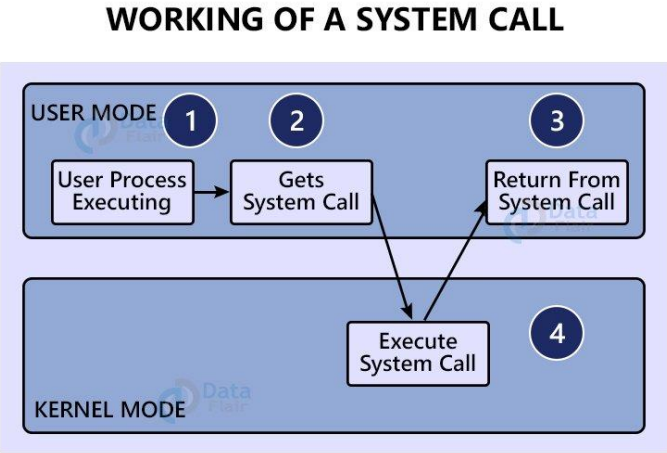
- User programs are written under the illusion that they have the entire machine. This means that when you **load** and **run multiple processes at a time**, you must manage **memory**, **scheduling**, and **other state correctly to maintain this illusion**.

- Do not forget:
  - In *utils/pintos* change line number 259 to /home/**name**/pintos/src/userprog/build/kernel.bin
  - In *utils/Pintos.pm* change line number 362 to /home/**name**/pintos/src/userprog/build/loader.bin

Problem 1.

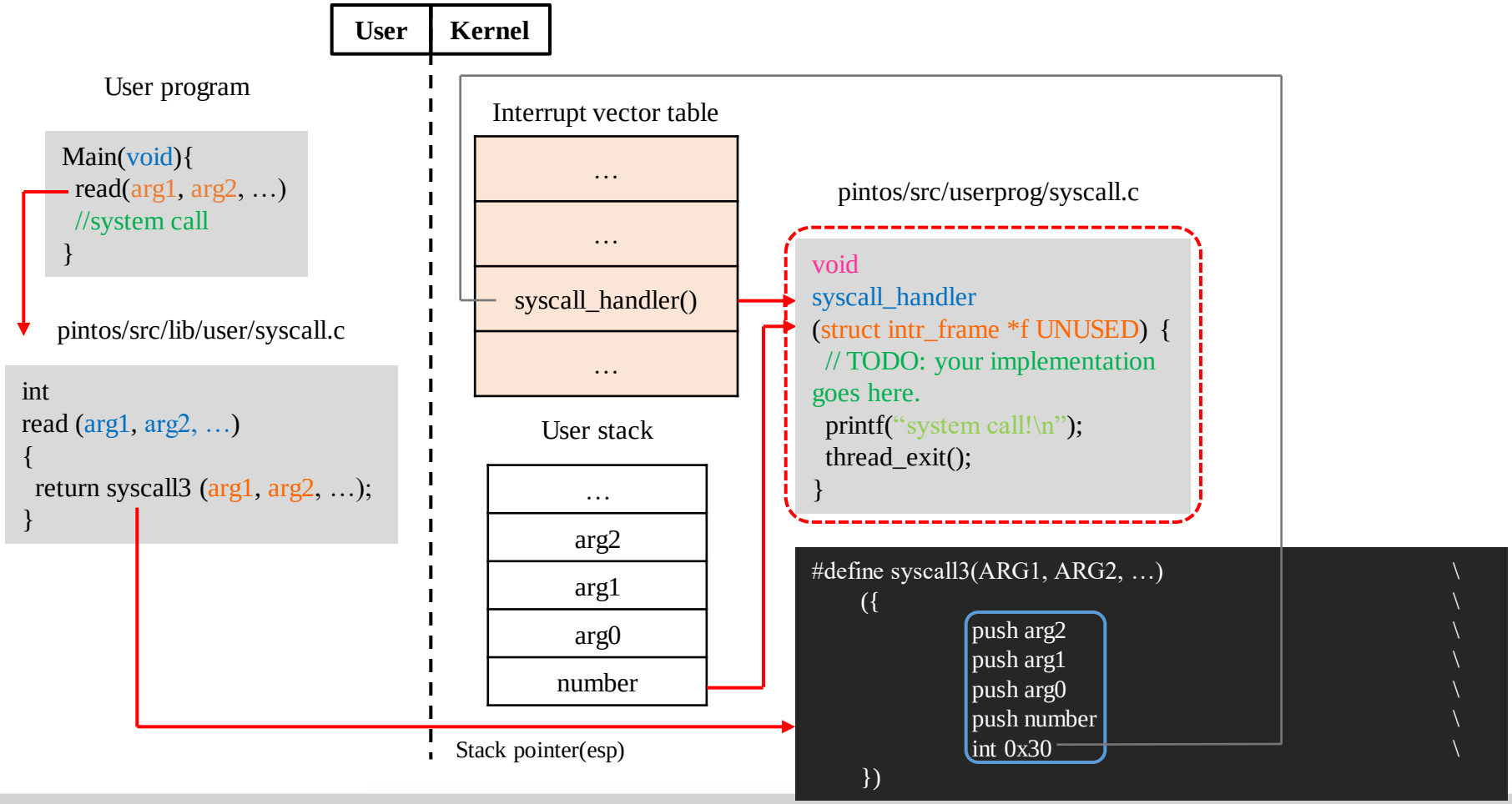
- **Process Termination Messages (required, 1pt)**

- **Requirement:**
  - Printing process termination messages with following rules:
    - Whenever a user process terminates for any reason, print the process's name and exit code, formatted as printed by **printf ("%s: exit(%d)\n", ...);**.
    - The printed name should be the full name passed to process_execute(), omitting command-line arguments.
    - Do not print these messages when a terminated thread is **not a user thread**, or when the **halt system call is invoked**. The message is optional when a process fails to load. Aside from this, do not print any other messages that Pintos as provided does not already print.



Problem 1.

- Problem Manual
  - Following graph shows how the system calls works in PintOS.
  - Red box means the part you should implement for project.



Problem 1.

- Problem Manual: Current state
  - There is no exit() function.
  - The syscall_handler is empty.
  - Process termination messages are not printing.

```
/* the main system call interface */
void
syscall_handler
(struct intr_frame *f UNUSED) {
  // TODO: your implementation goes here.
  printf("system call!\n");
  thread_exit();
}
```

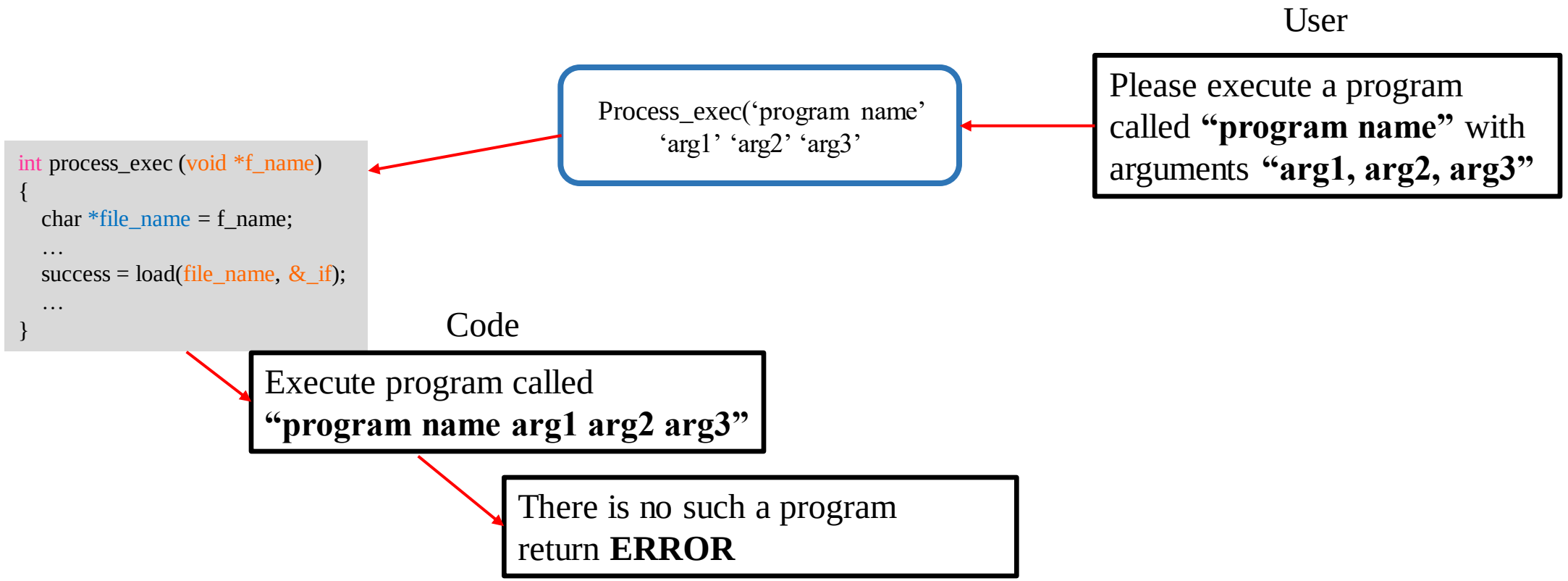
Problem 2.

- **Argument Passing (required, 1pt)**

- **Requirement:**
  - Passing arguments with following rules:
    - Argument passing is done by process_execute() function. <I.E. **process_execute("name arg1 arg2")** >
    - Each words are separated by space. <words == **['name', 'arg1', 'arg2']** >
    - First word is the program name, the second word is the first argument and so on. <program name: **'name'**, 1st argument: **'arg1'**, 2nd argument: **'arg2'**>
    - Multiple spaces are equivalent to a single space.
    - You can impose a reasonable limit on the length of the command line arguments.
    - *(There is an unrelated limit of 128 bytes on command-line arguments that the pintos utility can pass to the kernel.)*
  - You can parse argument strings any way you like. If you're lost, look at **strtok_r()**, prototyped in **'lib/string.h'** and implemented with thorough comments in **'lib/string.c'**.

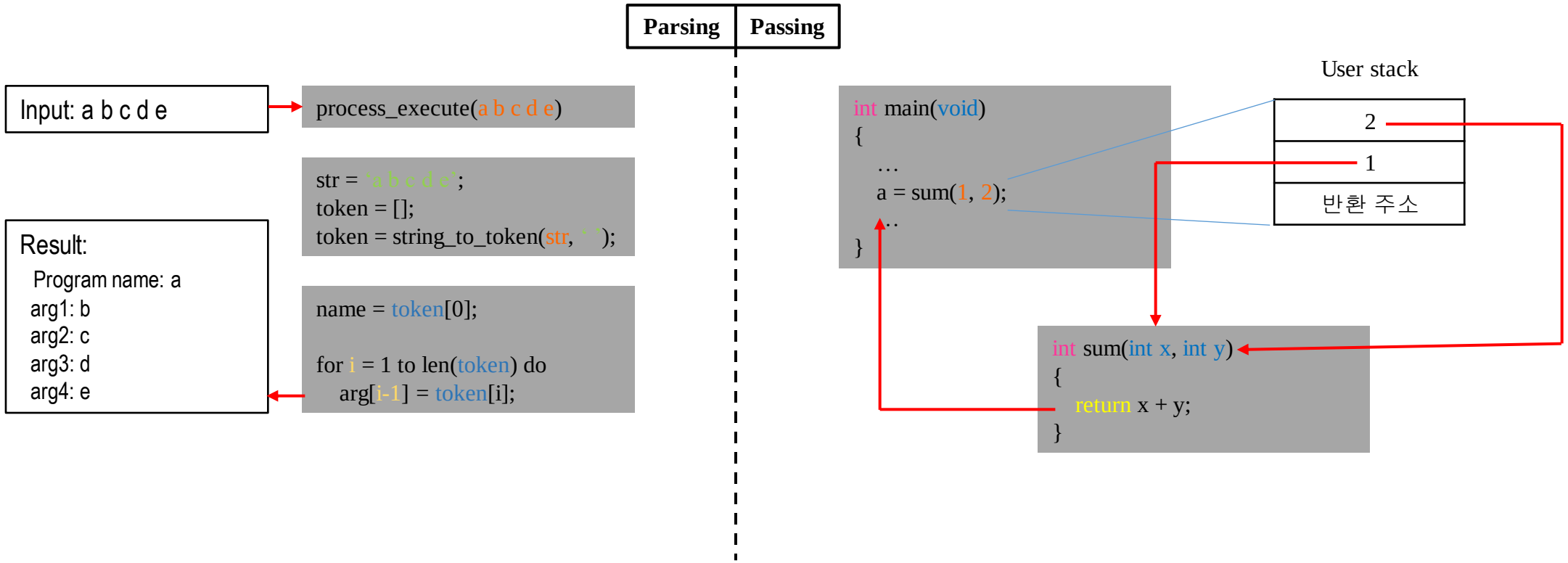
Problem 2.

- Problem Manual: Current state
  - Currently, pintos can't understand program name and arguments separately.
  - Therefore, make pintos **parsing** and **passing** the program name and arguments.



Problem 2.

- Problem Manual



Problem 3.

- **System Calls (required, 2pts)**

- **Requirement (cont.):**
  - **Implement the system call handler with following rules:**
    - There is a skeleton implementation of system call handler in **“~/pintos/src/userprog/syscall.c”**.
    - The skeleton implementation handles system calls by **terminating the process**.
    - Retrieve the system call number and carry out appropriate actions. (system call numbers for each system call are defined in **“~/pintos/src/lib/syscall-nr.h”**)

Problem 3.

- **System Calls (required, 2pts)**

- **Requirement (cont.):**
  - **Implement the system call handler with following rules:**
    - **Implement the following system calls.**
    - Prototypes of those functions are in **"~/pintos/src/lib/user/syscall.h"**. (Include this header file, and all others in **"~/pintos/src/lib/user"** directory, are for use by user programs only.)

```
System Call: void halt (void)
System Call: void exit (int status)
System Call: pid_t exec (const char *cmd_line)
System Call: int wait (pid_t pid)
System Call: bool create (const char *file, unsigned initial_size)
System Call: bool remove (const char *file)
```

```
System Call: int open (const char *file)
System Call: int filesize (int fd)
System Call: int read (int fd, void *buffer, unsigned size)
System Call: int write (int fd, const void *buffer, unsigned size)
System Call: void seek (int fd, unsigned position)
System Call: unsigned tell (int fd)
System Call: void close (int fd)
```

(Check 3.3.4 in https://web.stanford.edu/class/cs140/projects/pintos/pintos_3.html#SEC45 for detailed explanation of the system calls).

Problem 3.

- **Requirement:**
  - Ignore the other syscalls.
  - User-level function for each system call is provided in **“~/pintos/src/lib/user/syscall.c”**. These provide a way for user processes to invoke each system call from a C program. Each uses a little inline assembly code to invoke the system call and (if appropriate) returns the system call's return value.
  - **It is important to emphasize this point: tests will try to break your system calls in many ways**.
  - You need to think of all the corner cases and handle them. The sole way a user program should be able to cause the OS to halt is by invoking the halt system call
  - If a system call is passed an invalid argument, acceptable options include returning an error value (for those calls that return a value), returning an undefined value, or terminating the process.

Problem 3.

- Problem Manual
  - Implement all system calls.

System Call: void halt (void)	Halt: Terminate the OS.
System Call: void exit (int status)	Exit: Terminate the process.
System Call: pid_t exec (const char *cmd_line)	Exec: Execute process with given process name and return that process.
System Call: int wait (pid_t pid)	Wait: Waits for the given child process.
System Call: bool create (const char *file, unsigned initial_size)	Create: Create the file, if success then return TRUE and vice versa.
System Call: bool remove (const char *file)	Remove: Delete the file, if success then return TRUE and vice versa.
System Call: int open (const char *file)	Open: Open the given file.
System Call: int filesize (int fd)	File size: return the size of given file in bytes.
System Call: int read (int fd, void *buffer, unsigned size)	Read: Reads size bytes from the file, open as fd, into buffer.
System Call: int write (int fd, const void *buffer, unsigned size)	Write: Writes size bytes from buffer to the open file fd.
System Call: void seek (int fd, unsigned position)	Seek: Changes the next byte in given file to give position value.
System Call: unsigned tell (int fd)	Tell: returns the position of the next byte in given file.
System Call: void close (int fd)	Close: Close the given file

Please look up *filsys/file.c*. Functions in this file will help your implementation.

5. Report

- Problem, Design, and Implementation (required, 1pt)
- Please submit a report (limit 5 pages, Korean possible, no explicit demerits on exceeding page limits) that includes:
  - Student IDs, names, and team number
  - Each problem should be described in the form of
    - Problem definition
    - Algorithm design
    - Implementation (what you have added to solve the equation and corresponding file names)

6. What to submit?

- Submit a report (**5pages limit**) to TA via email (minwook-lee@gist.ac.kr)and upload implemented source code to your github repository.
- The format of report does not matter if it clearly shows how you implemented the project. (**please clarify which file of your code has been changed!**)

- Reference:

1. https://web.stanford.edu/class/archive/cs/cs140/cs140.1088/projects/pintos/pintos_3.html