- # Project 2: User Programs

I. Group Members

- Shiyi Li 11610511@mail.sustech.edu.cn
- Mengwei Guo 11610615@mail.sustech.edu.cn

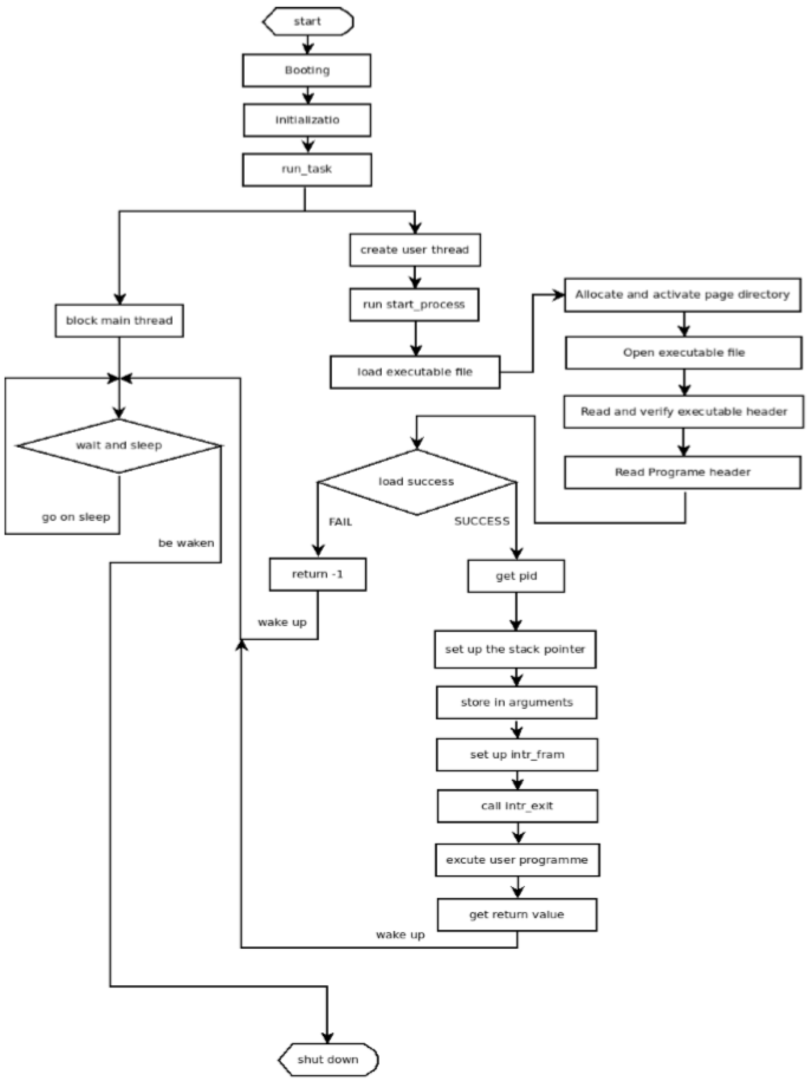
II. Division of labor

- **Task 1**
  - **Code** : Li Shiyi
  - **Report**: Li Shiyi
- **Task 2**
  - **Code** : Li Shiyi & Guo Mengwei
  - **Report** : Li Shiyi & Guo Mengwei
- **Task 3**
  - **Code**: Li Shiyi & Guo Mengwei
  - **Report**: Li Shiyi & Guo Mengwei

III. Implementation

Before we started coding, we had a brief look at pintos itself. As the picture below shown, while running a user program, pintos will follow the steps like:

1. PintOS will start and do the initialization, which includes storage allocation for kernel programs.
2. PintOS will load the ELF executable file and allocate address space for the user program (activate the page directory at the same time).
3. After that, the user program can take argument and push them in the stack of it. **It will be finished in Task 1.**
4. The System call function will read the parameters from the stack by moving the pointer. After executed the executable, it will put the return value in a specific position.
5. The Main process of operating system will wait for the return value of system call, when it has received the return value, the main process will be shut down.



Task 1: Argument Passing

- ### Requirement Analysis

  While executing an executable, it always requires arguments to control the running of the program. For example, the file name, the waiting time, and the path. The pintOS follows the style of parameter passing in C program, which is set two default parameters for the main function: **int argc** and **char* argv[].** argc generation the number of table arguments, and argv is an array of pointers passed in as strings. The placement of the parameters should be on the top of the stack of the user program stack. The stack, will expand space from high position to low position.

- ## Data structures

  We don't use any special data structures here, the stack of user program has been designed well, all we need to do is implementing the features of this stack.

- ## Relative Function

  process.c

  - process_execute (const char *file_name): get first parameter as name of thread.
  - get_parameter_num(char * file_name): get number of parameters.
  - setup_stack (void **esp, char * file_name): split parameters and push them into stack

- ## Algorithm & implementation

  - **The time to pass the argument**

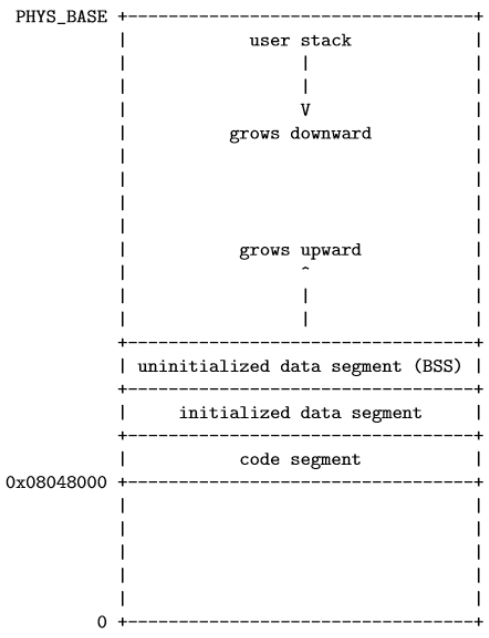
    After the the `load()` function has finished in function `start_process()`, we can write the arguments to the top of stack. The load() function itself, will call the `setup_stack` to split the argument string and put them on the top of stack.

  - **Decode the arguments string**

    After we get the argument number by calling `get_parameter_num()`.Predefined function `strtok_r` to is used to separate out one argument (a string). We can this function in C. The series of argument strings will be copied to the user stack in preparation for the next step. It should be mentioned that the order of characters in string is opposite to the increasing direction of the stack.

  - **Push the arguments in the user stack**

    In the `setup_stack` method, we will get the arguments contexts and the argc in the previous argument string analysis. After that, we need to ask for space for the arguments and move the stack pointer **esp**. Finally, the esp will point to the address of putting the return value of system. (We also need to make sure that argv[argc] is point to null.)



- ## Synchronization

  Everything in PintOS is represented by file, so while we run a specific syscall call, the user program shouldn't be modified (The file operation system call in Task 3 can do that) by other processes to prevent the changing of results. So we add a lock named **system_file_lock** in **thread.h** to implement the synchronization while loading the file.

Task 2: Process Control Syscalls

- ### Requirement Analysis

  In this task, we are required to implement several system calls about the process control, which include **halt()**, **exit()**. **exec()** and **practice().**

- ### Data structures

  In `struct thread`

  - `bool load_success`: check whether child process load successfully.
  - `semaphore child_lock`: record whether it need to wait for its child to execute.
  - `int return_value`: status when a thread need to exit.
  - `struct list *child_list`: store all its child thread.
  - `struct thread *parent`: current thread's parent thread.

  - `struct child_thread *waiting_child`: child thread that current thread is waiting for to execute.

    Add `struct child_thread`

  - `int tid`: tid of child thread.
  - `struct list_elem child_elem`: elem of child thread.
  - `int return_value`: return value of child thread.
  - `bool is_waited`: whether child thread is been waited by its parent thread.
  - `struct semaphore child_wait_lock`: whether child thread is waiting for other thread.

- ## Relative Function

  syscall.c

  - sys_exit(int status): syscall of exit
  - sys_exec(char *filename): syscall of exec
  - sys_practice(): syscall of practic

  thread.c

  - thread_exit(): exit a thread
  - thread_find_file(int fd): return the file pointer by searching it in the current thread based on its fd value.

  process.c

  - check_addr(const void *vaddr): check whether the address in stack is valid
  - process_wait(int child_tid) : wait for the indicated thread.

- ## Algorithm & implementation

  - **HALT**

    Use `shutdown_power_off()` in device to close the system directly.

  - **EXIT: sys_exit()**

    Get the return status of process from stack, delete the process from its parent's `child_list` and then allocate `thread_exit()`. In `thread_exit()`, as the current thread need to exit, its parent thread need to wait for it to exit until current thread exit finish.

- **EXEC: sys_exec()**

  Acquire `system_file_lock` first. Get filename from stack, if filename is null, which means thread has already exit, allocate `thread_exit()` to remove the thread. Else, use `filesys_open()` in `filesys.h` to open the file. If file exists, close the file, release the lock and allocate `process_execute()` in `process.c`.

- **WAIT**

  Read the tid of child thread by moving the esp for 1 potision, then call the function `process_wait()` in process.c, which we implement in the task 1.

- **PRACTICE: sys_practice()**

  Acquire `system_file_lock` first. Get the parameter from stack and and add one on it. Then push it back to the stack and release the lock.

- ### Synchronization

  In `sys_exec()` and `sys_practice()`, it need to operate on file system. In this two method, use `lock_acquire(&system_file_lock)` and `lock_release(&system_file_lock)` to prevent several processes operate on the same file.

Task 3: File Operation Syscalls

- ### Requirement Analysis

  In this task, we are required to implement several system calls about the file operation, which include **create (); remove (); open (); filesize (); read (); write (); seek (); tell (); close ();**

- ### Data structures

  In `struct thread`

  - `struct list *file_list`: store all files that the current thread opens.
  - `struct file *self`: current thread's executable file.

- `int file_count`: number of files it has opened.

  Add `struct file_describer`

- `int fd`: id of file_descirber.
- `struct list_elem file_elem`: elem of file_describer.
- `int file *ptr`: file of file_describer.

- ## Relative Function

  syscall.c

  - syscall_handler(struct intr_frame *f): add syscall function.
  - sys_open(char *filename): open file with the filename.
  - sys_read(int size, void *buffer, int fd): read file with certain fd.
  - sys_write(int size, void *buffer, int fd): write into file with certain fd.
  - sys_close(int fd): close the file with its file_describer has certain fd.

  thread.c

  - thread_exit(): exit a thread
  - thread_find_file(int fd): return the file pointer by searching it in the current thread based on its fd value.

  process.c

  - check_addr(const void *vaddr): check whether the address in stack is valid

- ## Algorithm & implementation

  - **create()**

    Get the filename an size of the file that we want to create from stack and use `check_addr` to check whether these values are valid. Then acquire the `system_file_lock` to prevent other thread to operate on file system. Allocate `filesys_create` in `filesys.h` to create the file and release the lock.

  - **remove()**

Get the filename of the file that we want to remove from stack and use `check_addr` to check whether these values are valid. Then allocate `filesys_remove` in `filesys.h` to remove the file.

- **open()**: sys_open(char *filename)

  Get the filename of the file that we want to remove from stack and use `check_addr` to check whether these values are valid. Then allocate `sys_open` to open file. In `sys_open`, acquire the `system_file_lock` to prevent other thread to operate on file system. Allocate `filesys_open` in `filesys.h` to open the file and release the lock. If the file exists, use a file_describer of record the file and its fd in current thread. Then push it into current thread's `file_list`.

- **filesize()**

  Get the filename of the indicated file by moving the stack pointer, then use `check_addr` to check whether the name is valid. After that, directly use function `file_length` defined in filesys.h. It should be mentioned that this function must be called surrounded by the acquire and release code of `system_file_lock` to make sure other processes won't change the size of file while count the size of file.

- **read()**: sys_read(int size, void *buffer, int fd)

  Get the filename, read buffer size and fd value by moving the stack pointer, then use `check_addr` to check whether the name is valid. We should check the fd value to make sure the fd is 0. When **fd = 0**, it will read from the console. We call function `thread_find_file` here to return the struct file according to the input fd. Then we call the function `file_read` in filesys.h to write the buffer to the indicated file and return the size of read.

- **write()**: sys_write(int size, void *buffer, int fd)

  Get the filename, read buffer size and fd value by moving the stack pointer, then use `check_addr` to check whether the name is valid. We should check the fd value to make sure the fd is 1. When **fd = 1**, it will print the buffer in the console. We call function `thread_find_file` here to return the struct file according to the input fd. Then we call the function `file_write` in filesys.h to write the buffer and return the size of read. `system_file_lock` is used to avoid race condition here.

- **seek()**

  Get the fd value and position value by moving the stack pointer, then use the function `file_seek` in filesys.h. We call function `thread_find_file` here to return the struct file according to the input fd. `system_file_lock` is used to avoid race condition here.

  - **tell()**

    The implementation of this system call is nearly the same as the seek(). We call the `file_tell` in filesys.h here. We call function `thread_find_file` here to return the struct file according to the input fd. `system_file_lock` is also used to avoid race condition here.

  - **close()**:: sys_close(int fd)

    The first thing we need to do is find the file based on its id. After we get the aimed file, we call three functions in order to close the file. `file_close` function in filesys,h is used to close the file. After that, we call `list_remove` to remove this file from the current thread's file list. In the end, we call function `free` to free the space this file object allocated.

- **Synchronization**

  Synchronization is an important issue for file system system call. In our implementation, we use the most straightforward way. The lock `system_file_lock` is acquired before we call the predefined function and released after the file operation is ended.

IV.Addition questions

- ***A reflection on the project–what exactly did each member do? What went well, and what could be improved?***

  The division of labor has been listed in the pervious part of this report. We met many problem about the pintos environment when we started doing this project. After we solved these problems, we had a tough time while doing the task 1. It's hard to test the results without the argument passing. We are stalled there until the bugs have been fixed. Then the write() system call is proved to be right. The following system calls are kind of easy to implement based on the previous functions. It's pity that we can't pass all the test cases at the end.

- ***Does your code exhibit any major memory safety problems (especially regarding C strings), memory leaks, poor error handling, or race conditions?***

  We haven't met errors related to the memory. Kernel panic happened sometimes but we don't know the trigger of it. We haven't modifies the exception.c in userprog directory so I guess our program may have a poor error handling. For the race condition, we simply used a global lock structure provided in source code and use it to prevent race condition in most of system calls. We also have tried to use semaphore but failed.

- ***Did you use consistent code style? Your code should blend in with the existing Pintos code. Check your use of indentation, your spacing, and your naming conventions.***

  We use consistent code style here.The main modification of codes only happened in four files : **syscall.c**, **process.c**, **thread.c** and **thread.h**. In syscall, we started the process in `syscall_handler()` and implement the complex system calls in separate methods (some easy system calls are not). We followed a same naming style too. For thread.c , we add two attributes to the thread struct but not create another struct. The only function we added `thread_find_file` also followed the naming style in thread.c

- ***Is your code simple and easy to understand?***

  It's maybe true. We added several new methods which can be easy understood based on their names. The structure of the syscall.c is very clear too. Some parts in process.c may be a little bit hard to understand but we add some comments to help the code reviewers.

- ***If you have very complex sections of code in your solution, did you add enough comments to explain them?***

  We did that for the sections that we thought it may be complex.

- ***Did you leave commented-out code in your final submission?***

  We commented some useless codes in some configuration files to make sure pintOS works well. We just simply follow the solutions in stackOverflow and we don't know why we need to fix some of them.

  At first, we put string split and put them into stack function after load() function in start_process(), however, it doesn't work and we find that in load(), it allocate **setup_stack()** to establish stack. So the operation is moved to the tail of function setup_stack().

- ***Did you copy-paste code instead of creating reusable functions?***

  No, we always try to create the reusable but not use duplicated code block in different places. For example , we create function **thread_find_file** in thread.c to implement the file search in current thread's files list. Because of that, we don't need to put the same code block everywhere.

- ***Are your lines of source code excessively long? (more than 100 characters)***

No, there is no such type of line in our source code.

- ***Did you re-implement linked list algorithms instead of using the provided list manipulation*?**

  No, we don't modify the basic structure of default linked list. We add two linked list named **child_list**(child thread list of every thread) and **file_list**(the file description of one thread) in the thread.h by using the provided list manipulation.