

Report

Task1: Argument Passing

- Data structure & Function
 - Modified thread.h

```
struct thread{
    ...
    /* added one attribute */
    struct file *self; // its executable file
    ...
}
```

- Modified process.c

- modified `process_execute`

```
tid_t process_execute(const char *file_name){
    /* added code to get the thread name(without arguments) for
    thread_create() */
}
```

- modified `load`

```
bool load(const char *file_name, void (**eip)(void), void **esp)
{
    /* added code to extract the name for executable */
}
```

- modified `setup_stack`

```
static bool setup_stack(void **esp, char *file_name){
    /* added code to split args and push into stack */
}
```

- Algorithms & Implementation

In first three method mentioned above, we only need to perform splitting on a long char list to get the executable and the args. We mainly leverage function `strtok_r` to split the name. Eg:

```
oken = strtok_r(file_name, " ", &temp_ptr)
```

In this way, we can easily get the splitted part of the file_name.

In the `setup_stack` method, we need also push the args into stack. We performed it follow the below steps:

1. Split the file_name to get the filename part and args part.
2. we calculate the number of args --> argc.
3. we ask for enough room for args and push them into the stack and make alignment for the stack pointer

```
/* the method for allignment looks like this */
while ((int) *esp % 4 != 0) {
    *esp -= sizeof(char);
    char x = 0;
    memcpy(*esp, &x, sizeof(char));
}
```

- Synchronization

As many threads can not manipulate one file at the same time, so we creted a global lock `filesys_lock` . Any operation with file need to first acquire this.

Here we use

```
lock_acquire(&filesys_lock);
lock_release(&filesys_lock);
```

to make sure there is only one process manipulate a file at one time.

- Rationale

This part is relatively easy and the logic for arguement parsing is straightforward. So our algorithm is ok to handle it.

Task2: Process Control Syscalls

- Data structure & Function

Here we only provid the name of funtions that we modified and we will show the modification detail later

- Modified `thread.h`

1. added attributes in thread structure
2. added one more struct

```

struct thread{
    ...
    bool load_success; /* to record if it's child is loaded
successfully */
    struct semaphore load_sema;
    int exit_status; /* store the exit code for a thread */
    struct list children_list; /* the list to store all the children
*/
    struct thread* parent; /* the parent of this thread */
    struct child_process * waiting_child; /* the child that this
thread is waiting*/
}
struct child_process{
    /* this structure is used to store the information of a thread
as a child and stored in parent process's `child_list` */
}

```

- Modified `syscall.c`

1. modified method `syscall_init`, `syscall_handler`
2. added method `get_content`, `exec_process`, `exit_process`, `check_address`, `syscall_exec`, `exec_process`, `syscall_wait`, `process_wait`, `syscall_halt`
3. added one variable

```

/* the array to save syscall handlers */
typedef void (*CALL_PROC)(struct intr_frame*);
CALL_PROC sys_array[21];

```

- Modified `process.c`

1. modified `process_execute`
2. modified `start_process`
3. modified `process_wait`

- Modified `thread.c`

1. modified `init_thread`
2. modified `thread_create`

- Modified `exception.c`

1. modified `kill`

- Algorithms

- Utility method

- **sys_array**

To make the code simple and straightforward, we store all the syscall handler in one array -- `sys_array`. We initialize this array in `syscall_init` like below:

```

sys_array[SYS_WRITE]=syscall_write;

```

- **check_address**

All method call this to check if an address. If the virtual address is NULL or it doesn't point to a valid area. This method will terminate current thread. Otherwise, it will return the physical address.

- **get_content**

All method call this method to get the argument out of stack.

- Syscall Handlers

- **Halt**

The handler for halt is `syscall_halt`.

The halt syscall will shutdown the system. So, we just call `shutdown_power_off` to implement it.

- **Exec**

The handler for exec is `syscall_exec`.

The main idea for implementing `Exec` is to create a new child process and bind it with its parent process.

In the handler we first get the executed file name and check if it is valid. If valid, we call `exec_process`. In `exec_process`, we first split out the name for the executable and check if it exist. If exist, we call `process_execute()` in `process.c` to create and run a new process. In `process_execute()`, the modification is mainly used to keep synchronization. A new process is actually a new thread, we create it in `thread_create` and create the corresponding `child_process` for it to be binded with its parent process.

- **Wait**

The handler for wait is `syscall_wait`

The main idea is that, if a process want to wait for one of its children. This process needs to be blocked until the waited child finished and returned the exit value.

In `syscall.c`, we first get the child tid that the thread want to wait. Then we call `process_wait` in `process.c` to perform wait. In `process_wait`, we first find the child thread by traverse the `children list` of the current thread to make sure that this is a real child and it is not the second time to wait for it and if this child has already stopped. Then after finding it. We implement the *wait* with semaphore(This will be discussed later). To wait a child, the parent thread need to *down* the semaphore of that child. And, when the child exits, it need to *up* its semaphore if there is one thread is waiting for it.

- **Exit**

Although original pintos support for *Exit*, it need to be modified due to the modification of other part.

Its handler is `syscall_exit`. In this, it first modify its information stored in its parent as a child like: `exit_status`, `if_waited`. Then call `thread_exit` to exit.

We also need to modify `kill` method in `exception.c` as it just calls `thread_exit` when terminate a thread. We need to change it to `exit_process`.

- Synchronization

- **filesys_lock**

Here also when manipulating files, we acquire the `filesys_lock` first. For example in `exec_process` we need to open the executable file to check its existence. So before we doing this, we first acquire `filesys_lock`.

- **semaphore**

Here, we use semaphore on child thread to manipulate the block and unblock procedure of waiting. The parent thread *down* the semaphore when waiting, and the child *up* the semaphore when exiting.

```
sema_down(&ch->wait_sema);
.....

sema_up(&thread_current()->parent->waiting_child->wait_sema);
```

Also, we use semaphore to record if a child thread is actually started(loaded). In method `process_execute` in file `process.c`, after create a thread, we first *down* the semaphore and in `start_process`, after successfully loading, we *up* the semaphore.

```
sema_down(&thread_current()->load_sema);
.....

sema_up(&thread_current()->parent->load_sema);
```

- Rationale

In this section, we take care of the synchronization problems and we implemented our logic correctly. We believe it will have a good performance.

Task3: File Operation Syscalls

- Data structure & Function

- Modified thread.h

```
/* added new attributes in thread */
struct thread
{
    ...
    struct list opened_files; /* keep the list of opened files for
this thread */
    int fd_count; /* the amount of fd for this thread */
    ...
}
```

- Modified syscall.h

```

/* added a new structure to store the information for one file in
opened_files*/
struct process_file {
    struct file* ptr;
    int fd;
    struct list_elem elem;
};

```

Added `syscall_write`, `syscall_create`, `syscall_open`, `syscall_close`,
`syscall_read`, `syscall_filesize`, `syscall_seek`, `syscall_remove`,
`syscall_tell`, `find_one_file`, `close_single_file`, `close_all_files`

- Algorithms

- Utility Methods

- **find_one_file**

search for a file in the opened_file list of a thread

- **close_single_file**

close one file for a thread

- **close_all_files**

close all files for a thread

- Syscall Handlers

- **create**

Its handler is `syscall_create`. It will first get the name and initial_size of the file and then call `filesys_create` to create a file.

- **write**

Its handler is `syscall_write`. It will first get the fd, size and buffer information. And according to fd, it will choose where to write(stdout or a file)

- **open**

Its handler is `syscall_open`. This syscall will open a file. First it get the file name and check if it is valid. Then it calls `filesys_open` to open the file and get the file pointer. Then it save these information to the current thread(fd_count, opened_files).

- **remove**

Its handler is `syscall_remove`. This syscall will remove a file. First it get the file name and check if it's valid. Then it use `filesys_remove` to remove the file.

- **filesize**

Its handler is `syscall_filesize`. This syscall will return the size of a file. First it will get the name of a file and then use `file_length` to get the file length.

- **read**

Its handler is `syscall_read`. This syscall will perform a read operation. First it will get the size, buffer, fd(file descriptor) information. Then according to fd to judge where to read(stdin or file).

- **seek**

Its handler is `syscall_seek`. This syscall will set the current position in a file to a new position. It first got the file descriptor and the new position then use `file_seek` to set the position.

- **tell**

Its handler is `syscall_call`. This syscall will return the current position in a file. It first gets the file descriptor and then use `file_tell` to get the current position.

- **close**

Its handler is `syscall_close`. This syscall will close one file for a thread. It get the file descriptor and use `close_single_file` to close it.

- Synchronization

Here these syscalls are all concerned with file operation. So we acquire the file lock `sysfile_lock` before performing these operations and release after.

- Rationale

In this part, most critical functions are not written by ourselves but by pintos. So we believe our functions works well.

Questions in PDF

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

Our members are: ZHANG Zhaoxu(ZZX), WANG Yutong(WYT)

Division:

Task1: ZX+WYT

Task2: ZX

Task3: WYT

Report: ZX+WYT

What went well: our division is quite specific and clear, so we can work individually and simultaneously.

Improvement: due to lack of communication, some times we might implement methods with similar functions which is a waste of time.

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

Actually, no these things happened. I guess it might because that, everytime we created one element use `malloc`, we will release it after using.

- 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.
Checked, and we take the naming convention same with Pintos which is separated by `_` and we formatted our code to keep a consistent style.
- Is your code simple and easy to understand?
Yes, I think it's quite easy to understand, we encapsulated same function code into method. And we also made comment under the code.
- If you have very complex sections of code in your solution, did you add enough comments to explain them?
I guess we added enough comment to explain.
- Did you leave commented-out code in your final submission?
Ok I removed them all.
- Did you copy-paste code instead of creating reusable functions?
I encapsulated all duplicate code into functions that I have found.
- Are your lines of source code excessively long? (more than 100 characters)
No. It's too ugly.
- Did you re-implement linked list algorithms instead of using the provided list manipulation?
Absolutely not, I used the list provided by pintos/

Reference

[1] 西安电子科技大学Pintos Project2

[2] CSCI 350: Pintos Guide, Stephen Tsung-Han Sher