Final Report for Project 2: User Programs

Group Members

- 黄玉安 11610303@mail.sustech.edu.cn
- 陈闽 11612602@mail.sustech.edu.cn

1 Task 1: Argument Passing

1.1 Data structures and functions

1.1.1 Data structures

- <thread.h>
  - Add new `structure process` to the .h file.
  - `#define MAXCALL 20`
  - `#include "kernel/list.h"`
  - `typedef void(*CALL_PROC)(struct intr_frame*);`

```
*/
struct process
{
  /*father will wait in this semaphore, the address is given by its parent*/
  struct semaphore* be_wait;
  struct semaphore wait_anyone; // to implement wait(-1)
  /* to implement wait child load file completely, father need to wait child
process loaded completely */
  struct semaphore wait_child_load;
    /* to implement child wait parent process_execute completely, child need
to wait father process process_execute completely*/
  struct semaphore wait_father_execute;

  struct thread* father;

  struct list child;
  tid_t pid;
  bool is_loaded;  // indicate whether process load successfully, since
execuable file may not found or have error

  struct file* this_file; // store the excutable file itself
  int rtv;              // return value of this thread(process).
};
```

Add return value of process in struct proccess (`rtv`). Other member in struct process will be used in task 2 and task 3.

1.1.2 Functions

The functions involved in this process is

- <process.c> :
  - `tid_t process_execute (const char *file_name)`
  - `static void start_process (void *file_name_)`
  - `bool load (const char *file_name, void (**eip) (void), void **esp)`
  - `static bool setup_stack (void **esp, char* file_name);`
  - `int process_wait (tid_t child_tid)`
- <syscall.c>
  - `void syscall_init (void)`
  - `static void syscall_handler (struct intr_frame *f UNUSED)`
  - `int syscall_WRITE(struct intr_frame *f); /* Write to a file. */`
  - `int syscall_EXIT(struct intr_frame *f)`

We put the code for return exit status in `syscall_EXIT` .

`process_wait(tid_t child_tid)` and `syscall_write(struct intr_frame *f)` are prerequisite functions to print to stdout.

The tokenize process of original argument(type char*) is implemented by calling `streak_r` with separator, and two char* input.

1.2 Algorithms

1.2.1 Analysis

The following steps of how pintos create a new userprog process is:

1. calling method `process_execute` in file <process.c>
2. in `process_execute`, invoke `thread_create` in <thread.c> and passing a function pointer of `start_process` to it. The operation system will ask for a correspinding room of memory.
3. In `start_process`, the function will initialize interrupt frame and load executable by calling `load` in <process.c> with argument of raw file name and `eip,eps` of interrupt frame we initialized.
4. In `load`, it loads an ELF from the splited `file_name` passed in. Stores the executable's entry point into `*eip` and its initial stack pointer into `*esp` by calling function `setup_stack`.
5. In `setup_stack`, the function push arg from input to the stack assign to the process. In the order of table showed in guideline.

1.2.2 Implementation

1.2.2.1 Split and Pass the arguments

The `file_name` passed into function `process_execute (const char *file_name)` include both executable file and the argument. Therefore, the first thing we need to do is to split the executable file name and the argument.

- <process.c/process_execute> Split the thread name and save it in the variable `char * thread_name`

  ```
  char *file_name_only;
  char *save_ptr;
  file_name_only = malloc(strlen(file_name)+1);
  strlcpy (file_name_only, file_name, strlen(file_name)+1);
  file_name_only = strtok_r (file_name_only," ",&save_ptr);
  ```

  Then invoke `thread_create` in <thread.c> to create a child threadand passing a function pointer of `start_process` to it. The operation system will ask for a correspinding room of memory.

  `tid = thread_create (file_name_only, PRI_DEFAULT, start_process, fn_copy);`

- <process.c/start_process>

  The function will initialize interrupt frame and load executable by calling `load` in <process.c> with argument of raw file name and `eip,eps` of interrupt frame we initialized.

  `success = load (file_name, &if_.eip, &if_.esp);`

- <process.c/load>

  Split the executable file name and open the file. Loads an ELF from the splited `file_name` passed in. Stores the executable's entry point into `*eip` and its initial stack pointer into `*esp` by calling function `setup_stack`.

- <process.c/setup_stack> In this method, we split the `argv` , count the `argc` and then push them in the correct order according to the document.
  - First tokenize the `file_name`
  - Iterate the token and count the `argc` (argument count)
  - ask for the coresponding size of room for `argv[]`
  - push the element of argv, word align. Also make sure the `argv[argc]` is a null pointer. Push the address of `argv` and finally a return address.

Address	Name	Data	Type
0xbffffffc	argv[3][...]	bar\0	char[4]
0xbffffff8	argv[2][...]	foo\0	char[4]
0xbffffff5	argv[1][...]	-l\0	char[3]
0xbfffffed	argv[0][...]	/bin/ls\0	char[8]
0xbfffffec	word-align	0	uint8_t
0xbfffffe8	argv[4]	0	char *
0xbfffffe4	argv[3]	0xbffffffc	char *
0xbfffffe0	argv[2]	0xbffffff8	char *
0xbfffffdc	argv[1]	0xbffffff5	char *
0xbfffffd8	argv[0]	0xbfffffed	char *
0xbfffffd4	argv	0xbfffffd8	char **
0xbfffffd0	argc	4	int
0xbfffffcc	return address	0	void (*) ()

1.3 Synchronization

In <process.c/start_process>

```
lock_acquire(&file_read_write_lock); // ,make sure only one process read this file
struct thread* t = thread_current();
lock_release(&file_read_write_lock);
sema_up( cur->proc.be_wait);
sema_up( &father->proc.wait_anyone );
```

Also, according to the **Task3**, while a user process is running, the operating system must ensure that nobody can modify its executable on disk. `file_deny_write(file)` denies writes to this current-running files.

1.4 Rationale

In this task, we split the input arguments and pass them into function `load` to push the arguments into the stack in the corret order. We also implement lock operations to ensure that nobody can operates on the file.

2 Task 2: Process Control Syscalls

2.1 data Structure:

add two structure **process** and **process_node**:

```
struct process_node
{
   /* father will wait in this semaphore, allocated by father and give the
poiner to child */
  struct semaphore father_wait;
  struct list_elem child_elem; // list elem for child list.
  tid_t pid;

  int rtv;  // return value of this thread(process).
};
```

add member in struct `thread`:

```
//==========add new struct to store process information============
    struct process proc;  // the process this thread belongs to
    struct process_node* node; // the node will be used by its father in its
child list
```

2.2 Algorithm

syscall framework:

use a array to store all system call's handle function with arguments `intr_frame* f`, by looking `f->esp` can known the syscall number, then it can get the corresponding handle function.

```
 int No = *(int *)(f->esp);
  if (No >= MAXCALL || No < 0)
  {
    process_exit_with_status(-1);
  }
  if (pfn[No] == NULL)
  {
    process_exit_with_status(-1);
  }
  f->eax = pfn[No](f); // put return value of syscall in f->eax, default
return value is 0
```

use a function pop_stack() to get the arguments in stack pointer, each time we should check whether the get address is valid:

the invalid address has three case:

- null pointers,
- invalid pointers(which point to unmapped memory locations)
- pointers to the kernel's virtual address space.

It may be the case that a 4-byte memory region (like a 32-bit integer) consists of 2 bytes of valid memory and 2 bytes of invalid memory, if the memory lies on a page boundary.

halt:

just call `shutdown_power_off();`

exit:

I have noted that the return value of process will put in struct `process`, it is done in `void process_exit_with_status(int status)`.

After we get the return value of the process, we need to released all resource in `process_exit`,

in function `process_exit`.

- load failure: the exit process may even load executable file fail, so I have a member `is_loaded` to indicate it which I have mentioned in data Structure part. When load fail, do not print exit code, and do not close the executable file `this_file`, and the return value of this process is 0 (default value).
- load success: we need to do all things I have mentioned in load failure case. And wait up parent process since he may be wait. I use semaphore to implement then. The semaphore `wait_ anyone` was used to case when parent called wait(-1).

In both case above, we need to release all resource this process owned, including opened file, and released all child (make child's father be NULL, unless child may wait up their father by sema_up semaphore `&father->proc.wait_anyone` which may be dirty (father process has been destroyed).

When process exit, it should copy its return value in `process_node->rtv`, otherwise there may have memory leakage when parent invoke wait to get child's return value;

exec:

just get its expected return value and invoke `process_exit_with_status(rtv)`.

wait:

There have many case. This syscall is the most complicated one.

- when parent come to wait earlier:

  waited in children's semaphore

  - It is really this process's child:

    after waited child exit, remove this child in its child list. The return value was get by process_node which has the target `child_tid`.

  - It not my child or have waited before:

    since wait will remove this child, so it can't not found this child in its child list

- when parent come to later than child call exit:
  - just read find child's process_node and get its return value (`child_pro_node->rtv`).

2.3 Synchronization

when doing syscall wait, parent need to wait in a semaphore until child called wait or child exit before. Using a semaphore `child_pro -> to_wait` to implement this synchronization.

```
  sema_down(to_wait);
```

And in child exit, up this semaphore.

```
//=======wait up waited father if any=======
     struct thread* father = cur->proc.father;
     if(father!=NULL && father->status != THREAD_DYING){
       sema_up( cur->proc.be_wait);
       sema_up( &father->proc.wait_anyone );
     }
```

2.4 Rationale

In this task, I implement some system call which is the basis for the whole project. The wait and exit is very complecated since it has lots of synchronization problem. Child may return first or parent exit first.

Task 3: File Operation Syscalls

3.1 Data structures and functions

thread.h

```
struct thread {
    ...
    struct list opened_files;   //all the opened files
    int fd_count;
    ...
};
```

Holding the list of all opened files.

syscall.h

```
struct process_file {
  struct file* ptr;
  int fd;
  struct list_elem elem;
};
```

Store file pointer and file description number, as a list_elem in the `opened_files` of the `struct thread`.

syscall.c

- `static void syscall_handler (struct intr_frame *)`

  Handling the file syscall, going to the specific calls by the values in the stack.

- specific file syscall functions

  Call the appropriate functions in the file system library.

```
int syscall_EXIT(struct intr_frame *f);
int syscall_EXEC(struct intr_frame *f);
int syscall_WAIT(struct intr_frame *f);
int syscall_CREAT(struct intr_frame *f);
int syscall_REMOVE(struct intr_frame *f);
int syscall_OPEN(struct intr_frame *f);
int syscall_FILESIZE(struct intr_frame *f);
int syscall_READ(struct intr_frame *f);
int syscall_WRITE(struct intr_frame *f);
int syscall_SEEK(struct intr_frame *f);
int syscall_TELL(struct intr_frame *f);
int syscall_CLOSE(struct intr_frame *f);
```

- `void pop_stack(int *esp, int *a, int offset)`

  All pop operation on the stack needs to call this function. It will verify if the stack pointer is a valid user-provided pointer, then dereference the pointer.

- `int exec_process(char *file_name)`

  Sub-function invoked by `int syscall_exec()`: split string into tokens and call `process_execute()` with tokens.

- `void exit_process(int status)`

  Sub-function invoked by `int syscall_exit()`: set current thread status by `status`, and update the status of the child(current) process in its parent process. At last, call `thread_exit()`.

- `struct process_file* search_fd(struct list* files, int fd)`

  Find file descriptor and return process file struct in the process file list, if not exist return NULL.

- `void clean_single_file(struct list* files, int fd)`

  Go through the process file list, and close specific process file, and free the space by the file descriptor number.

- `void clean_all_files(struct list* files)`

  Go through the process file list, close all process file and free the space. Do this when exit a process.

3.2 Algorithms

When a syscall is invoked, `void syscall_handler()` handle the process. All arguments are pushed in the stack when a user program doing system operation using `lib/user/syscall.c`. So, we just need to take the parameter from the stack.

1. We pop the syscall number from the stack by the `esp`.
2. we go to the specific syscall functions by the syscall number. For each file syscalls, we need to pop more detailed arguments. If the parameter we pop out is a pointer (such as a `char *`), we also need to verify its usability.
3. Each file syscalls call the corresponding functions in the file system library in `filesys/filesys.c` after acquired global file system lock.
4. The file system lock will be released anyway at last.

3.3 Synchronization

All of the file operations are protected by the global file system lock, which prevents doing I/O on a same fd simultaneously.

```
//syscall_READ
lock_acquire(&file_read_write_lock);
ret = file_read(pf->ptr, buffer, size);
lock_release(&file_read_write_lock);
//syscall_WRITE
lock_acquire(&file_read_write_lock);
ret = file_write(pf->ptr, buffer, size);
lock_release(&file_read_write_lock);


//release_all_file
if(lock_held_by_current_thread(&file_read_write_lock)){
  lock_release(&file_read_write_lock);
}
```

3.4 Rationale

Actually, all the critical part of syscall operations are provided by `filesys/filesys.c`. At the same time, the document warns us avoiding modifying the `filesys/` directory. So the vital aspect is that poping and getting the data in the stack correctly, and be careful not to do I/O simultaneously.

Question

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

        - Task1 :
            - Code : Yu'an Huang
            - Report : Min Chen
        - Task2:
            - Code : Yu'an Huang
            - Report : Yu'an Huang & Min Chen
        - Task3:
            - Code: Min Chen
            - Report: Min Chen
        - Debug
            - Min Chen
            - Yu'an Huang
- Does your code exhibit any major memory safety problems (especially regarding C strings), memory leaks, poor error handling, or race conditions?
    - Yes. We noted that when process_exetuce, it must wait untile child load programe completely and after child load completely, it also need to wait parent complete process_execute since patent need to konw child's status, child can not run too fast. And we also need to remember deallocated memory.
- 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.
    - Our code style is comsistent.
- Is your code simple and easy to understand?
    - Yes, we using variable with meaningful names and using comment on it.
- If you have very complex sections of code in your solution, did you add enough com- ments to explain them?
    - Yes. For example, we have `load` function with lines: 110 and comment lines 17.
- Did you leave commented-out code in your final submission?
    - No.
- Did you copy-paste code instead of creating reusable functions?
    - No.
- Are your lines of source code excessively long? (more than 100 characters)
    - No.