Project 2 Final Report

The project is finished individually by Songlin Yang (student ID:11611519).

Some of ideas come from: https://github.com/davidpeterko/pintOS-User-Programs/

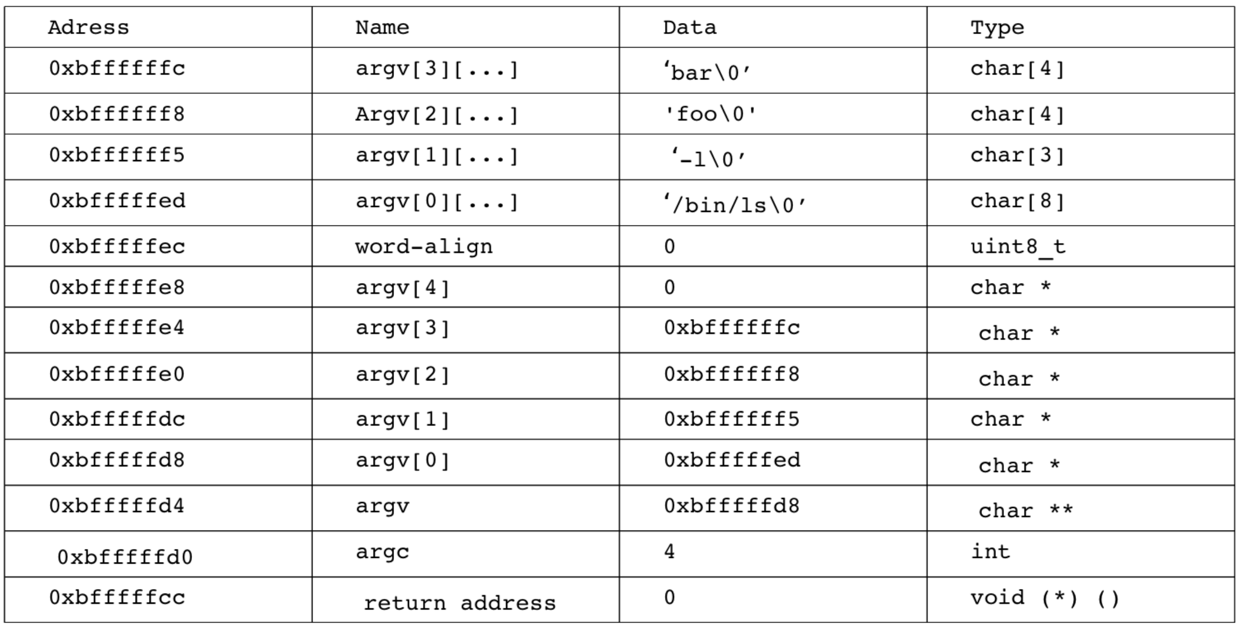
Task1: Argument parsing

1.1 Data structure and functions:

- <thread.h>
    - `struct file *code_file`: Use for store the executable code file.
- <process.c> :
    - `process_execute (const char *file_name)`
    - `load (const char *file_name, void (**eip) (void), void **esp)`
    - `setup_stack (void **esp, char * file_name)`

    These functions are involved closely with our task: argument parsing.

1.2. Algorithm:



Adress	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 (*) ()

Our goal is to set up stack like described above.

Our implementation will be found at `setup_stack (void **esp, char * file_name)`

The argument **file_name** contains all the argument concated by white space. So we need to use `strtok_r` function to split it up.

```
int i = 0;
for(token = strtok_r(fn_copy," ",&save); token!=NULL; token =
strtok_r(NULL," ",&save))
{
    tokens[i] = token;
    i++;
}
```

Then **i** is our desired **argc**.

Push these into the stack according to the order mentioned above, then this part is finished.

1.3 Synchronization

In load method, we need to open the executable file. In order to realize synchronized read. We have to add a global lock when we open the file.

```
lock_acquire(&filesys_lock);
file = filesys_open (t->name);
lock_release(&filesys_lock);
```

Also, when we load the file successfully, we need to make sure that our executable file will be not edited by others.

```
if(success){
    t->code_file = file;
    file_deny_write(file);
}
```

When we load successfully, we have to call **file_deny_write** on the file we opened.

1.4 Rational

I think we put the code related to argument parsing in the method **setup_stack** is a very natual thing.

Task2 Process Control Systemcalls

2.1 Data structure and functions

- <thread.h>
    - `int exit_status`
    - `int load_status`
    - `struct semaphore load_sema`
    - `struct semaphore exit_sema`
    - `struct semaphore wait_sema`
    - `struct list children` Use to store the list of child processes.

- <syscall.c>
    - `static void checkvalid (void *ptr,size_t size)` To check the validity of given address with given size.
    - `static void syscall_handler (struct intr_frame *f UNUSED)` To handle system call.

2.2 Algorithm:

2.2.1 dealing with the invalid memory access.

We gaurantee our memory access's validity in this way: The method **checkvalid** has two argument, the first one is the start address, the second one is the length of memory you want to access.

There are three types of invalid memory access. (1) nullptr (2)try to access kernel space (3) the pointed user space has not been allocated pages.

In order to make sure the start and the end of accessed memory is valid, we check each of these type in the start address and end address.

2.3 Synchronization

2.3.1 load sema

When the parent process call **exec** system call, if the child process didn't load successfully, the parent process should return -1. The problem is that under the current implementation, the parent process has no idea about the load status of the child process. So we introduce a semophore **load_sema** and a varaible **load_status** in thread structure.

Now, when the parent process call **exec** system call, the **exec** system call will call **process_execute** . When we successfully get child process id, there is no guarantee that the process will load successfully, so we let parent process to wait for the end of load procedure of the child process.

```
    sema_down(&child->load_sema);

    if(child->load_status == -1) tid=TID_ERROR;
    else{
        list_push_back(&thread_current()->children, &child->childelem);
    }
```

The child process hold a load_sema, child process will sema up the load sema as soon as it complete the load procedure. If fails, we setup load_status of the child process to -1. So, the parent can be notified whether the child process is loaded successfully. If success, push it back into its child process list. Else return -1.

2.3.2 wait_sema

Wait sema is used for system call **wait**. When a process call **wait** for another process, it must wait until that process finishes. When the waited process exits, it will sema up the wait sema. So the waiting process can continue executing. To handle the two scenarios that we must return -1 immerdiately, we need to maintain a child process list. If the pid is not one of child process's pid, we should return immediately.

2.3.3 exit_sema

exit_sema is quite interesting. Document says "**kernel must still allow the parent to retrieve its child's exit status, or learn that the child was terminated by the kernel.**" So the child process will not actually exit completely. In fact, it will free most of its allocated resource but the thread struct is not completely destroyed. The exit code will be stored, which make its parent process possible to have ability to check its child process's exit code. But if the parent process exits, all of its child process is free to exit since their parent is dead and they don't need to consider the situation that their parent process want to look up its exit status. So, when a process exit, it will sema up all of its child process's exit_sema.

```
void  process_exit (void)
{
...
for (e = list_begin (&cur->children); e != list_end (&cur->children);
     e = list_next (e))
{
    child = list_entry (e, struct thread, childelem);
    sema_up (&child->exit_sema);
}
....

    sema_down(&cur->exit_sema);   // not actually exit, makes its parent
possible to know its exit status.
}
```

The document also says that **The process that calls wait has already called wait on pid. That is, a process may wait for any given child at most once.** So in **process_wait** method, as soon as the parent process is waken up by its child process, the parent process should remove it from its children list and sema up the child process's exit_sema.

```
int process_wait (tid_t child_tid) {
 ...
   list_remove(&child->childelem);
   exit_status = child->exit_status;
   sema_up(&child->exit_sema);
   return exit_status;
   }
```

2.4 Rational

As analysed above, **load_sema**, **wait_sema**, **exit_sema** can work perfectly with our process synchronization issues.

Task3 File Operation Syscalls

3.1 Data structure and functions

- <thread.h>
  - `struct file* file[128]` : To store all files the process opened. Manually set the maximum file the process is able to open as 128.
- <syscall.h>
  - `struct lock filesys_lock` Global file system lock.
- <syscall.c>
  - `static void checkvalidstring(const char *s)` Check the validity of the string. (Memory access aspect)
  - `static int checkfd (int fd)` Check the validity of fd.

3.2 Algorithm

3.2.1 Allocating file descriptor id.

```
static int open(const char * file_name){
  ...

  for (i = 2; i<128; i++)
    {
        if (t->file[i] == NULL){
            t->file[i] = f;
            break;
        }
    }
  ...

}
```

The way to allocate fd is simply tranverse the file array.

3.2.2. Read system call

We should handle the situation that the fd=0 (Standard input), we need to get input from user input in terminal. **input_getc** is used.

```

static int
read(int fd, void* buffer, unsigned size){
    ...
    if(fd == 0) {
    while(size > 0) {
        input_getc();
        size--;
        bytes_read++;
    }
    return bytes_read;
   }

   ...
}
```

3.3.3 Write system call

Also ,we need handle the special situation that fd=1 (standard output). **putbuf** is used.

```
static int
write(int fd, const void* buffer, unsigned size){
    ...

    if(fd == 1){
        while(size > 200){
            putbuf(buffChar,200);
            size -= 200;
            buffChar += 200;
            buffer_write += 200;
        }

        putbuf(buffChar,size);
        buffer_write += size;
        return buffer_write;
    }

    ...
}
```

3.4 Synchronization

All operation related file system is surrounded by

```
lock_acquire(&filesys_lock);

 some file sys operation.

lock_release(&filesys_lock);
```

In **process_exit**, check if the exiting process hold the lock, if so, release it to avoid infinite waiting.

```
    if(lock_held_by_current_thread(&filesys_lock)){
        lock_release(&filesys_lock);
    }
```

3.5 Rational

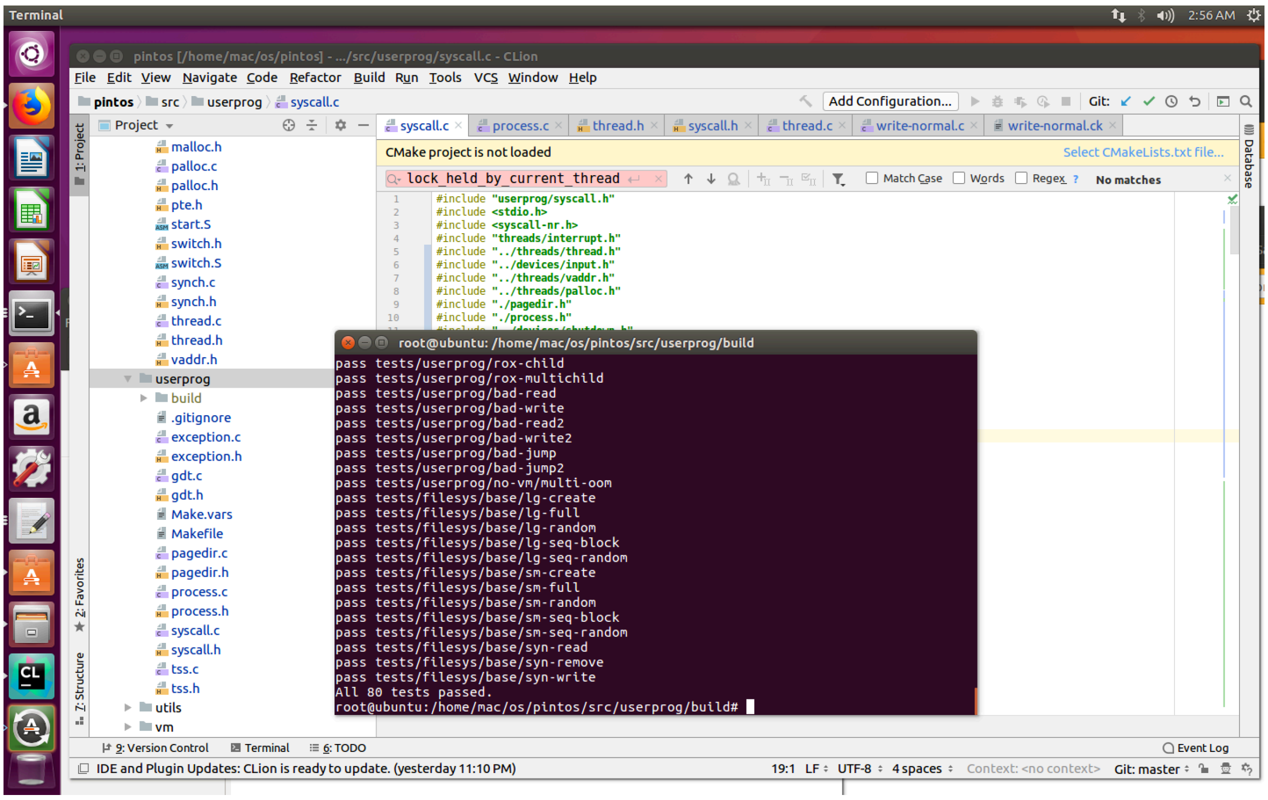
The reason to use file array to store all of the opened file:

(1) Given the fd, it is easy to find the opened file. Just `t->file[fd]`

(2) it is simple to allocate fd id to avoid skipping.

Reflection

Final result:



```
#include "userprog/syscall.h"
#include <stdio.h>
#include <syscall-nr.h>
#include "threads/interrupt.h"
#include "../threads/thread.h"
#include "../devices/input.h"
#include "../threads/vaddr.h"
#include "../threads/palloc.h"
#include "./pagedir.h"
#include "./process.h"
```

```
root@ubuntu: /home/mac/os/pintos/src/userprog/build
pass tests/userprog/rox-child
pass tests/userprog/rox-multichild
pass tests/userprog/bad-read
pass tests/userprog/bad-write
pass tests/userprog/bad-read2
pass tests/userprog/bad-write2
pass tests/userprog/bad-jump
pass tests/userprog/bad-jump2
pass tests/userprog/no-vm/multi-oom
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
All 80 tests passed.
root@ubuntu:/home/mac/os/pintos/src/userprog/build#
```

Question:

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

The code does exhibit some major memory safety problem before I meet the test case multi-oom. It turns out the I forgot to close all open files before process exits. So when the multi-oom test runs, it opens lots of files and finally result in out of memory and fails me.

With C strings, I use **strtok_r** (the thread safety version of **strtok**) method instread of **strtok**. And I also add another **checkvalidstring** method to check every single byte of the C string to see whether it is out of the valid memory.

Finally, I believe all of the issues are solved.