

## PROJECT 2: USER PROGRAMS

---- GROUP ----

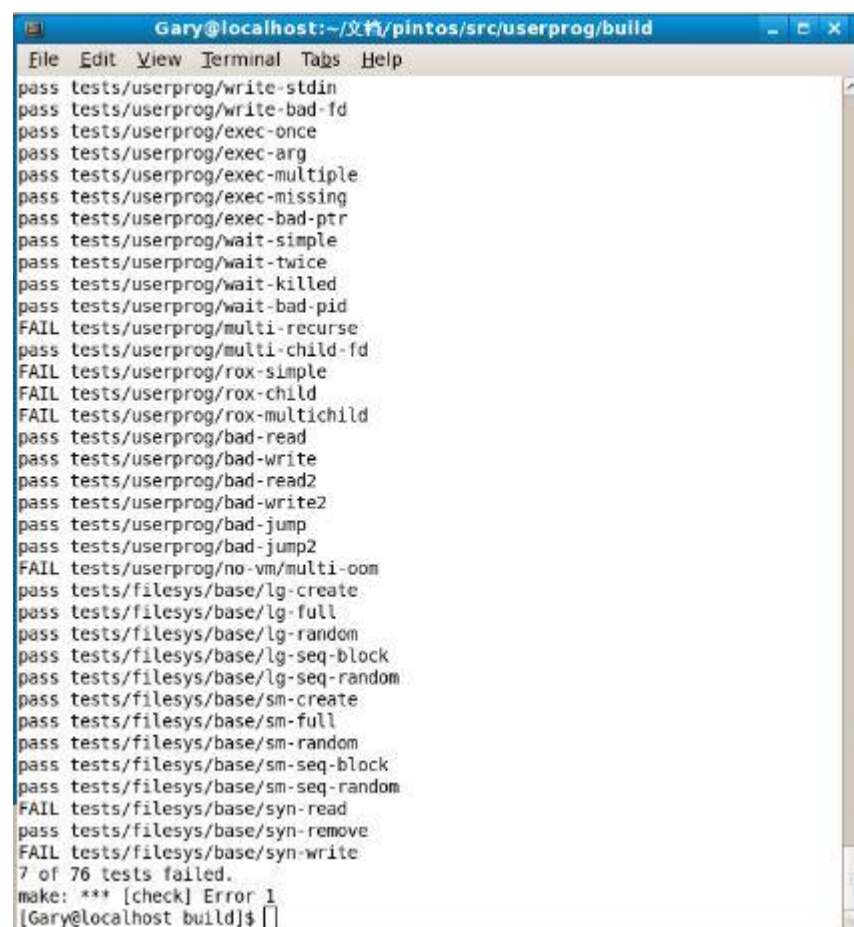
[阮明康<860797144@qq.com>](mailto:860797144@qq.com)

[吴宏昱<hapda@qq.com>](mailto:hapda@qq.com)

[安显琴<1244905436@qq.com>](mailto:1244905436@qq.com)

---- PRELIMINARIES ----

Firstly, we only passed 69 cases out of the 76 cases, here is the screen shot:



```
Gary@localhost: ~/文档/pintos/src/userprog/build
File Edit View Terminal Tabs Help
pass tests/userprog/write-stdin
pass tests/userprog/write-bad-fd
pass tests/userprog/exec-once
pass tests/userprog/exec-arg
pass tests/userprog/exec-multiple
pass tests/userprog/exec-missing
pass tests/userprog/exec-bad-ptr
pass tests/userprog/wait-simple
pass tests/userprog/wait-twice
pass tests/userprog/wait-killed
pass tests/userprog/wait-bad-pid
FAIL tests/userprog/multi-recurse
pass tests/userprog/multi-child-fd
FAIL tests/userprog/rox-simple
FAIL tests/userprog/rox-child
FAIL 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
FAIL 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
FAIL tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
FAIL tests/filesys/base/syn-write
7 of 76 tests failed.
make: *** [check] Error 1
[Gary@localhost build]$
```

In order to store user programs in disk for execution, we need to make  
a simulated disk and copy program:

- |  |                                   |
|--|-----------------------------------|
| 1. pintos-mkdisk fs.dsk 2                      | //create 2MB virtual disk         |
| 2. pintos -f -q                                | //format disk then quit           |
| 3. pintos -q ../../examples/echo -a echo -- -q | //copy file to the disk then quit |
| 4. pintos -q run 'echo x'                      | //run "echo x" then quit          |

## ARGUMENT PASSING

=====

### ---- DATA STRUCTURES ----

We only need to add new data as follow:

The description of each variable has been showed as the comment follows the variable in the graph below:

```
struct thread
{
    /* Owned by thread.c. */
    tid_t tid;          /* Thread identifier. */
    enum thread_status status; /* Thread state. */
    char name[16];      /* Name (for debugging purposes). */
    uint8_t *stack;     /* Saved stack pointer. */
    int priority;       /* Priority. */
    struct list_elem allelem; /* List element for all threads list. */
    struct list_elem elem;   /* List element. */

#ifdef USERPROG
    struct list files; /*list for open files
    int ret_status;    //process's return status
    struct thread *parent; //process's parent thread
    struct semaphore t_sema; //for synchronous processes control
    uint32_t *pagedir; /* Page directory. */
#endif
    unsigned magic;    /* Detects stack overflow. */
};
```

new added members

## ----- ALGORITHMS -----

### 1. Parsing arguments:

**Analysis:** for a file name with several arguments , the name of the file and the arguments are separated by several blank spaces ; so we only need to parse a string( file name or argument) where the blank spaces exist.

```
tid_t
process_execute (const char *file_name)
{
    char *fn_copy,*tmp,*parsed_name;
    tid_t tid;
    struct thread *t;
    struct thread *p=thread_current();
    /* Make a copy of FILE_NAME.
       Otherwise there's a race between the caller and load(). */
    fn_copy = palloccopy_page (0);
    parsed_name = palloccopy_page (0);
    if (fn_copy == NULL || parsed_name == NULL)
        return TID_ERROR;
    strcpy (fn_copy, file_name, PGSIZE);
    strcpy (parsed_name, file_name, PGSIZE);
    parsed_name= strtok_r(parsed_name, " ", &tmp);
    /* Create a new thread to execute FILE_NAME. */
    tid = thread_create (parsed_name, PRI_DEFAULT, start_process, fn_copy);
    palloccopy_free_page (parsed_name);
    if (tid == -1)
        return -1;
    t = get_thread(tid);
    t->parent = p;
    while (t->status == THREAD_BLOCKED) thread_unblock(t);
    sema_down(&t->t_sema);
    if (p->ret_status == -1) tid = -1;
    if (tid == TID_ERROR) palloccopy_free_page (fn_copy);
    return tid;
}
```

### 2. Arrange the elements of argv[] in the right order:

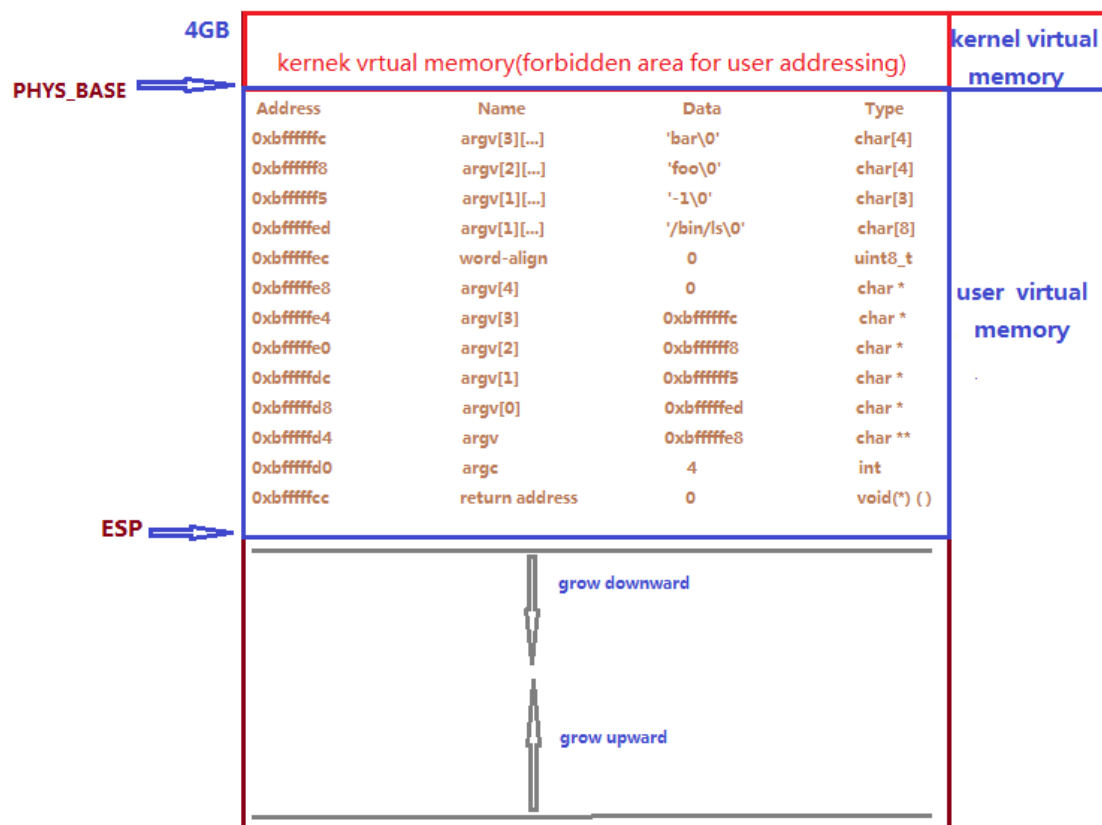
**Analysis:** we parse the string ( file name ) from right to left, token one element at a time, then store it in argv[i], i++; the graph below shows the arrangement:

```

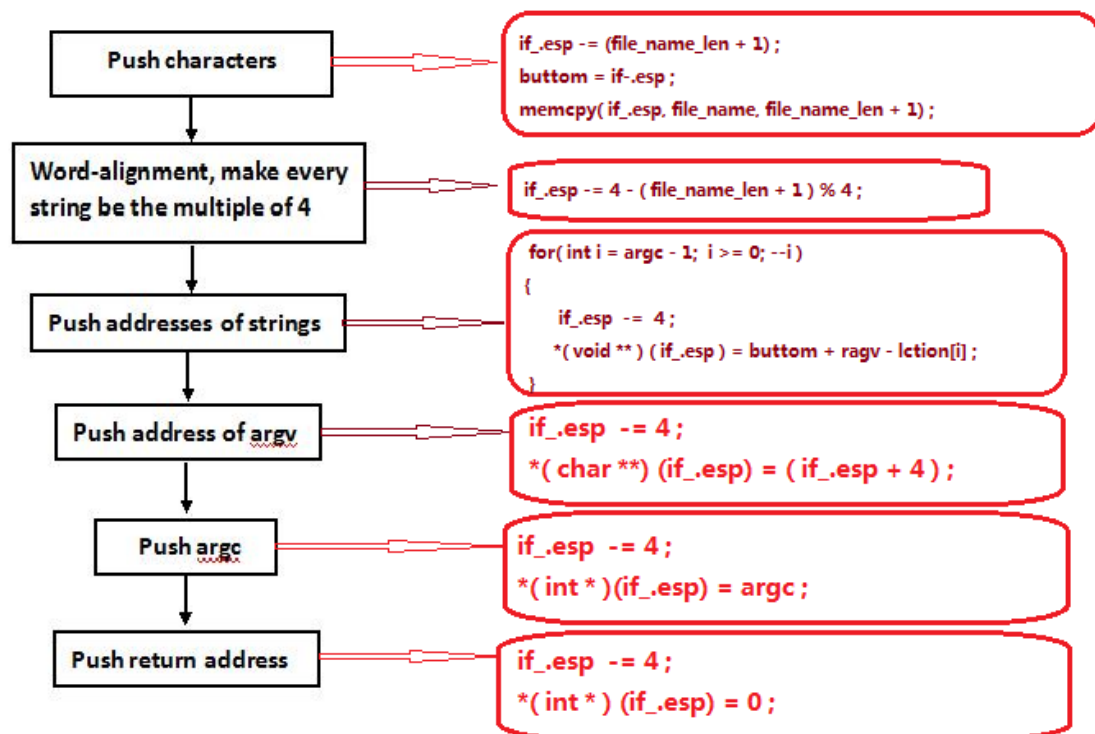
char *usr_argv[argc+1];
int i=0;
for ( token=strtok_r(file_name,"",&save_ptr) ;
      token!=NULL ;
      token=strtok_r(NULL,"",&save_ptr) )
{
    usr_argv[i]=token;
    i++;
}
usr_argv[argc]=NULL;

```

In order to passing arguments into stack successfully, we must first understand how the arguments are stored in the user stack, graph:



The flowchart for putting arguments into stack :



### 3. Avoid overflowing the stack page:

**Analysis:** As the case that stack page hardly happens, we just ignore it, assuming it will never happen.

---- RATIONALE ----

>> A3: Why does Pintos implement `strtok_r()` but not `strtok()`?

**Analysis:**

Prototype for `strtok()`: `char *strtok(char *s, char *delim);`

For the function `strtok()`, the pointer used to store the substrings' addresses is of static type, so it allocates static address space to store the addresses of the substrings, which will cause the thread falls into an unsafe state.

Prototype for `strtok_r()`: `char *strtok_r(char *s, const char *delim, char **ptrptr);`

For the function `stroke_r()`, the user have to provide a local pointer to store the substrngs' addresses, so it is a thread-safe function. We would better choose `strtok_r()` in order to ensure the thread be in safe state.

>> A4: In Pintos, the kernel separates commands into a executable name

>> and arguments. In Unix-like systems, the shell does this

>> separation. Identify at least two advantages of the Unix approach.

Analysis:

Shell is not kernel, actually it's a software provides interface for users, called command parser.

Advantage 1: it provides two styles to parse and execute the users' commands: **interactively** and **automatically**.

Advantage 2: shell is the outermost layer of the operating system, it manages the interactive or non interactive inputs between user and the operating system, parsing the commands , and deals with various outputs sent from the operating system. So shell provides a more powerful , stable and functional mechanism for the communication between user and OS.

## SYSTEM CALLS

=====

>> B1: Copy here the declaration of each new or changed `struct' or

>> `struct' member, global or static variable, `typedef', or

>> enumeration. Identify the purpose of each in 25 words or less.

Analysis: no new or changed 'struct' or other type of variables need to add in this part.

>> B2: Describe how file descriptors are associated with open files.

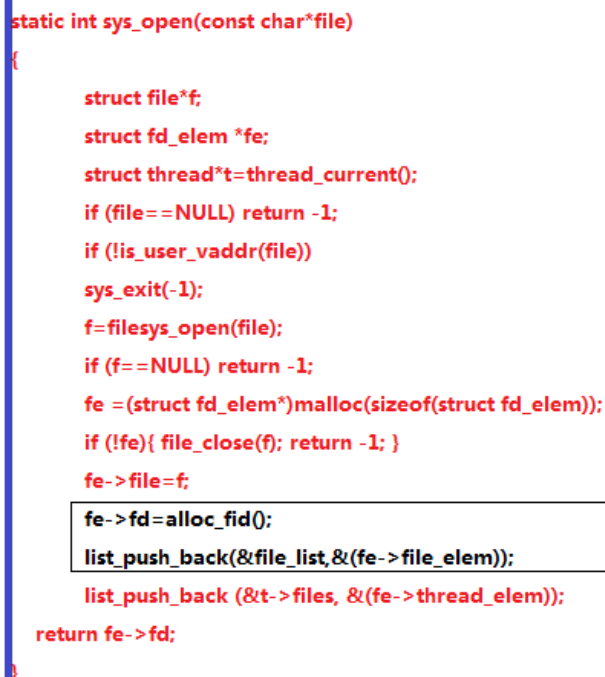
1. File descriptor is a handler associated with a specific file, see the structure of file element:

```
struct fd_elem
{
    //char *file_name;
    int fd;
    struct file *file;
    struct list_elem file_elem;
    struct list_elem thread_elem;
};
```



2. When open new file, allocate a new file descriptor to it:

```
static int sys_open(const char*file)
{
    struct file*f;
    struct fd_elem *fe;
    struct thread*t=thread_current();
    if (file==NULL) return -1;
    if (!is_user_vaddr(file))
        sys_exit(-1);
    f=filesys_open(file);
    if (f==NULL) return -1;
    fe =(struct fd_elem*)malloc(sizeof(struct fd_elem));
    if (!fe){ file_close(f); return -1; }
    fe->file=f;
    fe->fd=alloc_fid();
    list_push_back(&file_list,&(fe->file_elem));
    list_push_back (&t->files, &(fe->thread_elem));
    return fe->fd;
}
```



new file is created, allocate new file-descriptor to it, and put it to the open-file list

3. Since every open file is associated with a unique integer file handler( file descriptor ) , we can access the open file by using the file descriptor as file representative:



>> Are file descriptors unique within the entire OS or just within a single process?

1. First see the definition of the file descriptor:

```
static int fid=1;
```

It is a static integer.

2. Then look at how the file descriptor is allocated to a new value:

```
static int alloc_fid(void)
{
    return ++fid;
}
```

3. Since the file descriptor is a static integer, it is stored in the global section of the stack, which will not be erased during the whole execution period of pintos. So , the file descriptors are unique within the entire OS.

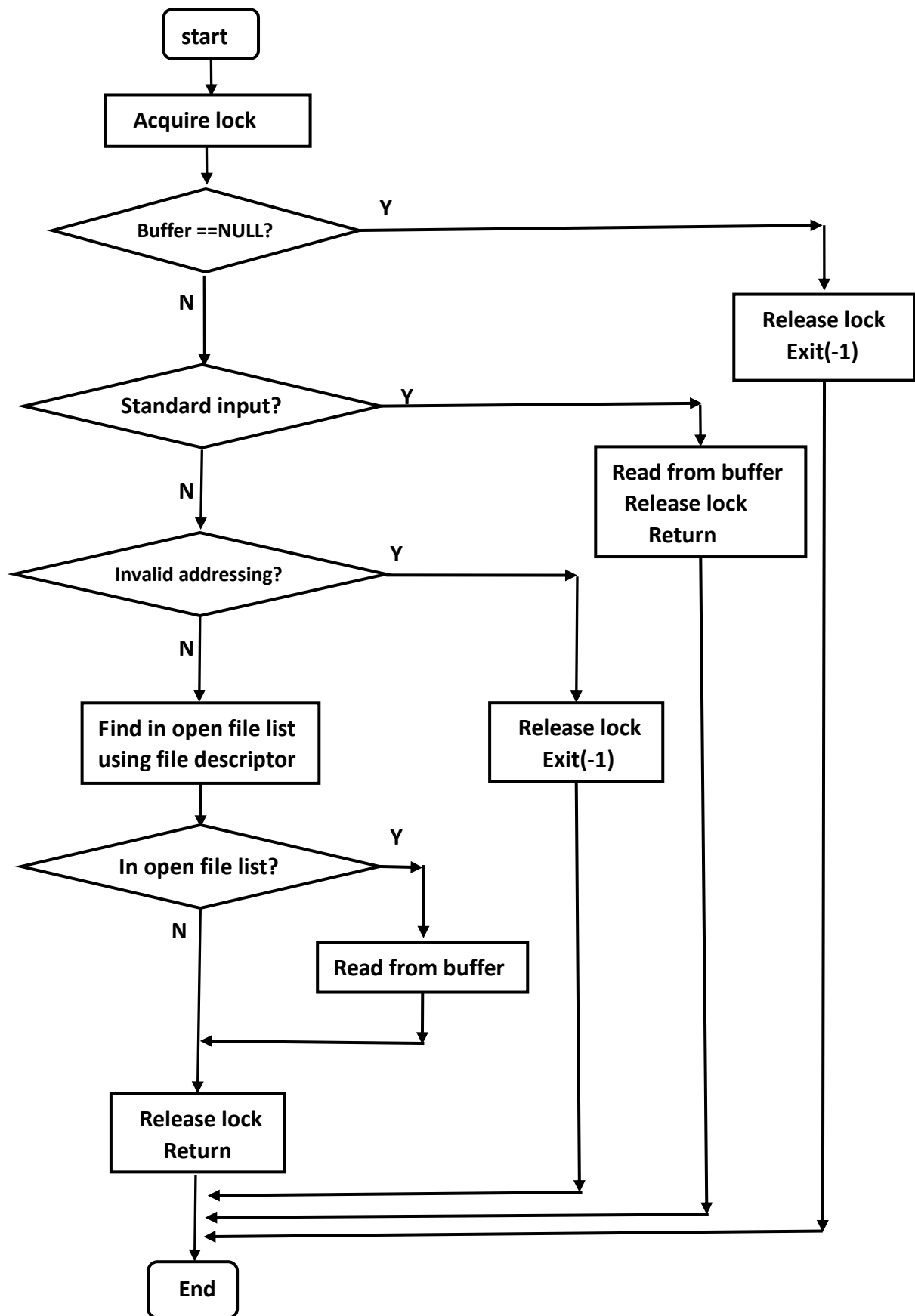
---- ALGORITHMS ----

>> B3: Describe your code for reading and writing user data from the

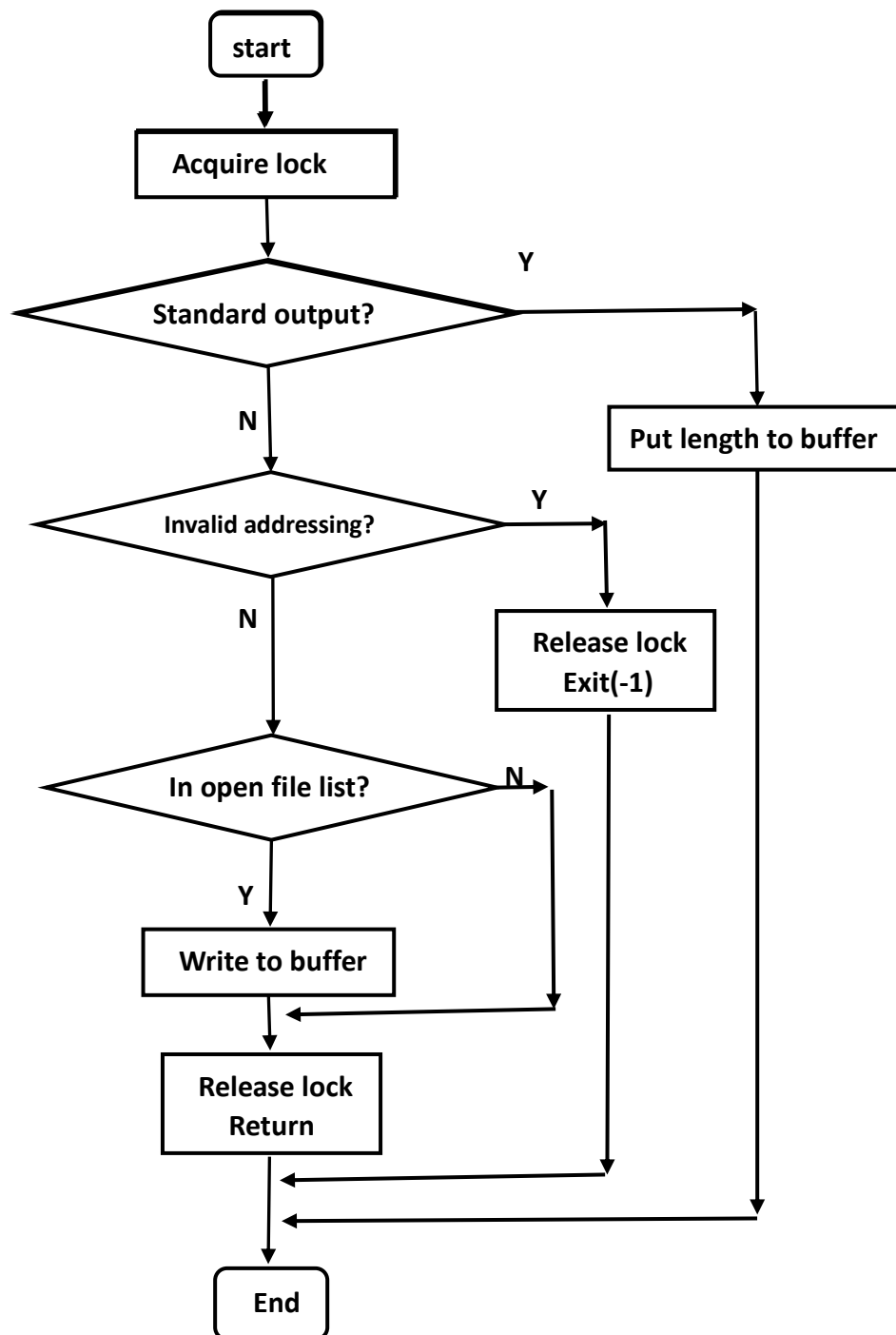
>> kernel.



1. Reading. The flowchart for the code of reading system call is as follow:



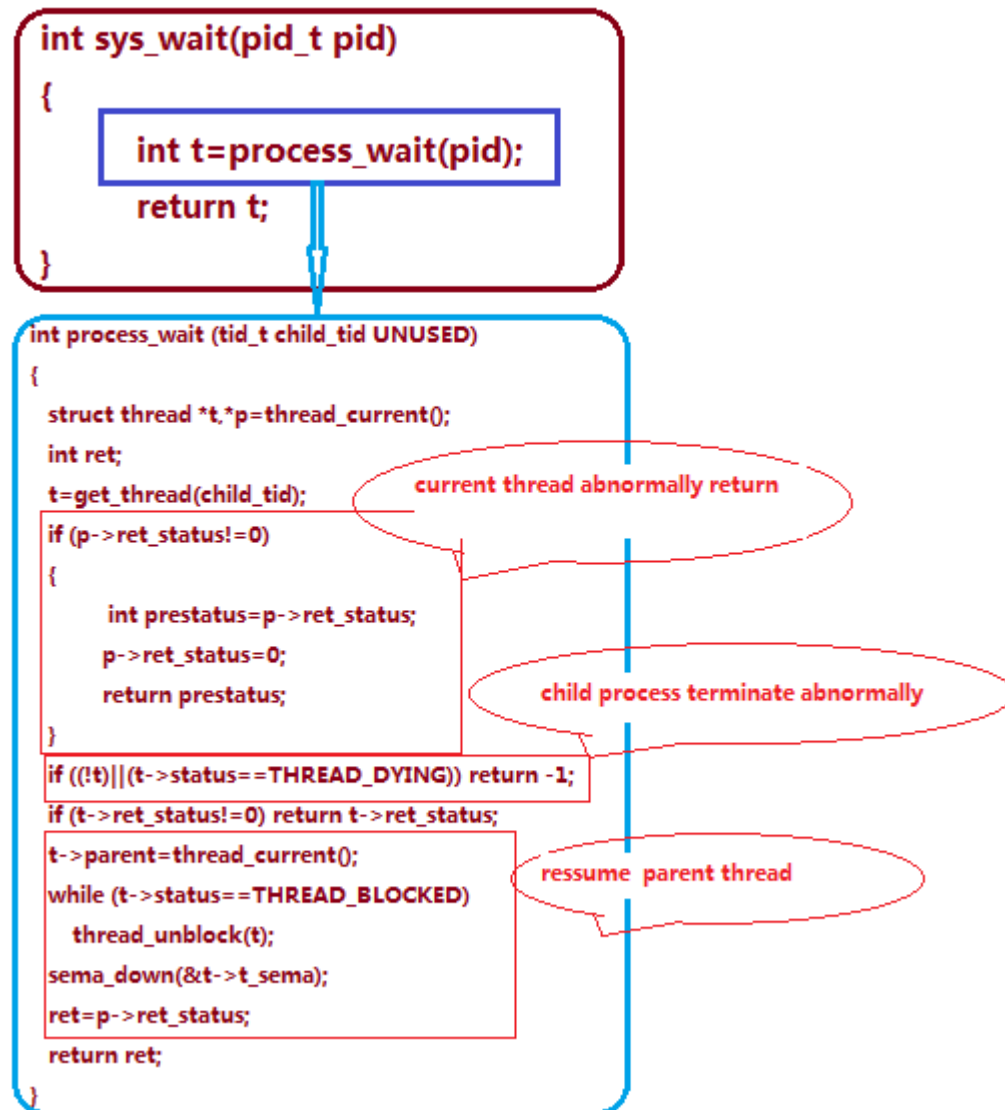
2. Writing: The flowchart for the code of writing system call is as follow:



>> B5: Briefly describe your implementation of the "wait" system call

>> and how it interacts with process termination.

My descriptions have been drawn as follow:



>> B6: Any access to user program memory at a user-specified address

>> can fail due to a bad pointer value. Such accesses must cause the

>> process to be terminated. System calls are fraught with such

>> accesses, e.g. a "write" system call requires reading the system

>> call number from the user stack, then each of the call's three

>> arguments, then an arbitrary amount of user memory, and any of

>> these can fail at any point. This poses a design and

>> error-handling problem: how do you best avoid obscuring the primary

>> function of code in a morass of error-handling? Furthermore, when

>> an error is detected, how do you ensure that all temporarily

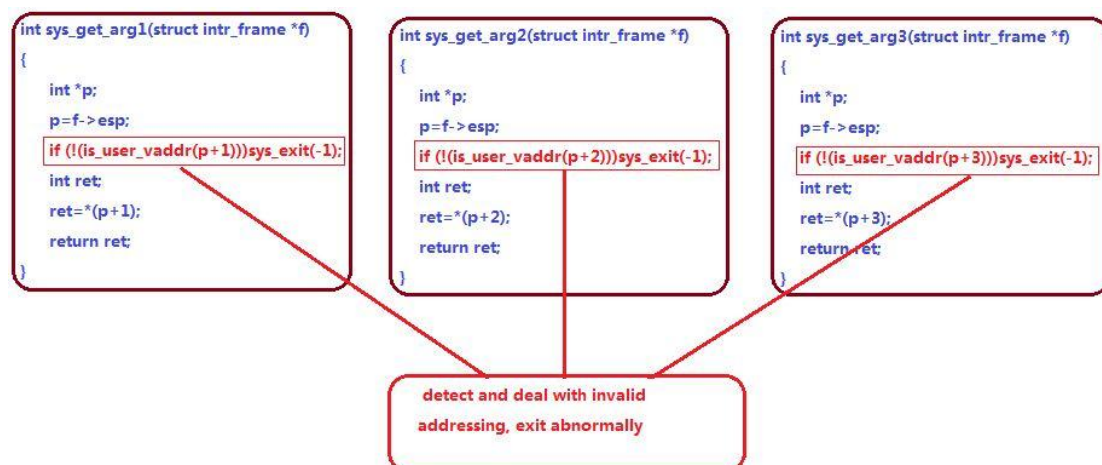
>> allocated resources (locks, buffers, etc.) are freed? In a few

>> paragraphs, describe the strategy or strategies you adopted for

>> managing these issues. Give an example.

Analysis: when the user passes a pointer to the kernel, the pointer may be NULL, which points to an unmapped virtual memory, or points to the kernel virtual memory space, both will cause page fault.

When passing the call's three arguments, the detection and handling of invalid addressing is as follow:



In the function `is_user_vaddr()`, we just check whether the address is under `PHYS_BASE` or not. If it is under `PHYS_BASE`, then it is a valid address, or else is invalid addresses which may cause page faults.

For the system call write, we handle it likes this:

```

int sys_write (int fd, const void *buffer, unsigned length)
{
    struct file *f;
    int ret=-1;
    lock_acquire(&file_lock);
    if (fd==STDOUT_FILENO)  putbuf(buffer,length);
    else if (!is_user_vaddr(buffer)||!is_user_vaddr(buffer+length)) {
        lock_release(&file_lock);
        sys_exit(-1);
    }
    else {
        f=find_file_by_fd(fd);
        if (!f) {
            lock_release(&file_lock);
            return ret;
        }
        ret=file_write(f,buffer,length);
    }
    lock_release(&file_lock);
    return ret;
}

```

when writing a file, both the address of the file's head and tail must be checked, if one of them is invalid, release the lock, and system halts abnormally.

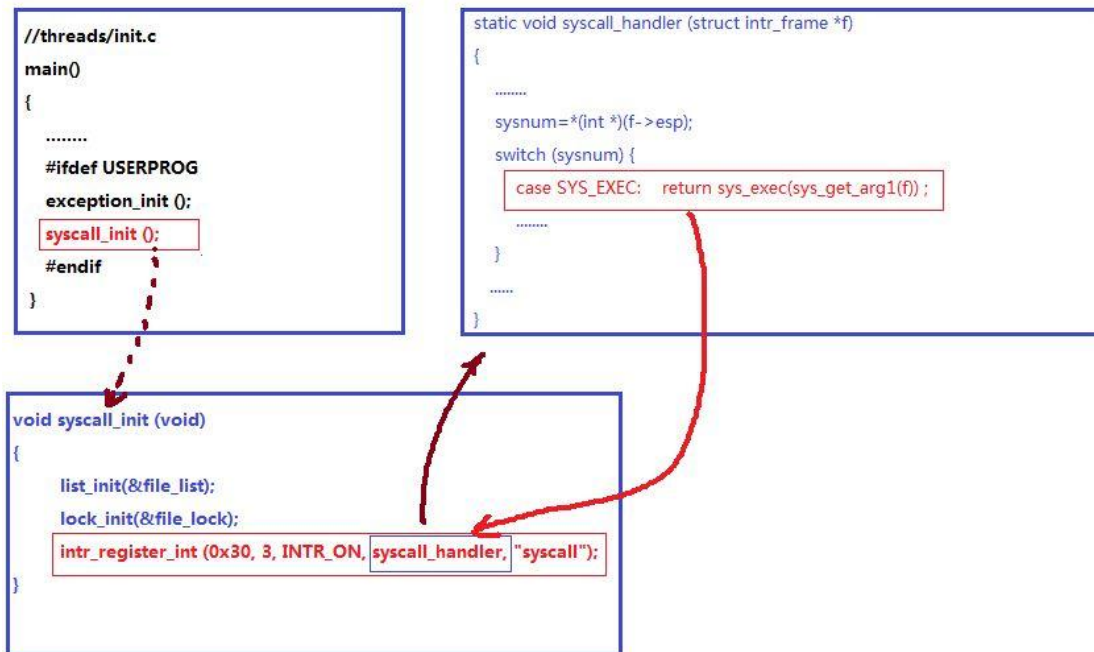
#### ---- SYNCHRONIZATION ----

- >> B7: The "exec" system call returns -1 if loading the new executable
- >> fails, so it cannot return before the new executable has completed
- >> loading. How does your code ensure this? How is the load
- >> success/failure status passed back to the thread that calls "exec"?

#### Analysis:

To ensure that the new executable completely loading before returning, we applied a lock to hold the critical section( the loading of the new executable), the lock will never be released until the new executable has completely loading.

The load success/failure status passed back to the thread which calls 'exec' by calling the system call handler, an example is drew as this:



>> B8: Consider parent process P with child process C. How do you  
 >> ensure proper synchronization and avoid race conditions when P  
 >> calls `wait(C)` before C exits? After C exits? How do you ensure  
 >> that all resources are freed in each case? How about when P  
 >> terminates without waiting, before C exits? After C exits? Are  
 >> there any special cases?

Analysis:

1. When P calls `wait(C)` before C exit, we applied a semaphore with initial value 0.

The structure:

```

struct semaphore sema.value = 0 ;
Wait( C ) ;
Sema_down( sema ) ;
C exits
Sema_up( sema ) ;

```

When P calls `wait( C )` after C exits, we applied a semaphore with initial value 1 ;

```

struct semaphore sema.value = 1;
C exits
Sema_down( sema ) ;
Wait( C ) ;
Sema_up( sema ) ;

```

2. when P terminates before C exits, we applied a semaphore with initial value 0.

The structure:

```
struct semaphore sema.value = 0 ;  
Sema_down( sema ) ;  
C exits  
Sema_up( sema ) ;  
P terminates;
```

when P terminates after C exits, we applied a semaphore with initial value 1,

The structure:

```
struct semaphore sema.value =1 ;  
Sema_down( sema ) ;  
C exits  
Sema_up( sema ) ;  
P terminates;
```

#### ---- RATIONALE ----

>> B9: Why did you choose to implement access to user memory from the

>> kernel in the way that you did?

Analysis: just for obeying the requirement and passing the tests, additionally, it is more easy to understand and implement.

>> B10: What advantages or disadvantages can you see to your design

>> for file descriptors?

Analysis:

Advantage: whenever a file is going to open , allocating a new value to the file descriptor just need to simply increase the static integer( fid ) like ++fid, which represents the file descriptor.

Disadvantage: since the file descriptor is a static integer, the largest number can it holds is 65536. if too many files are opened, since we did not impose any limitation on the number of open files, the file descriptor would possibly crashed.

>> B11: The default tid\_t to pid\_t mapping is the identity mapping.

>> If you changed it, what advantages are there to your approach?

Analysis:

both `tid_t` and `pid_t` are `int`, by default, the mapping is the identity mapping, namely, we can make them a one-to-one mapping, so that the same values in both identify the same process.

If we change it by applying a more complex mapping, we can implement mappings like one kernel thread maps to many user threads or many kernel threads map to many user threads.

## SURVEY QUESTIONS

=====

Answering these questions is optional, but it will help us improve the course in future quarters. Feel free to tell us anything you want--these questions are just to spur your thoughts. You may also choose to respond anonymously in the course evaluations at the end of the quarter.

>> In your opinion, was this assignment, or any one of the three problems

>> in it, too easy or too hard? Did it take too long or too little time?

My opinion: too hard, it took me too much time to handle it!

>> Did you find that working on a particular part of the assignment gave

>> you greater insight into some aspect of OS design?



My opinion: yes, project1 had helped me have a deeper understanding of concept of thread, the execution of kernel thread, the mechanism of synchronization and thread scheduling. In project2, it helped me understand how a program stored in disk is loaded into memory, how to parse the arguments and pass the argument as well as the implementation of system calls.

>> Is there some particular fact or hint we should give students in

>> future quarters to help them solve the problems? Conversely, did you

>> find any of our guidance to be misleading?

My opinion: introduce the pintos project at the beginning of the semester, and end it at the end of the semester, synchronizing the theory teaching.

>> Do you have any suggestions for the TAs to more effectively assist

>> students, either for future quarters or the remaining projects?

My opinion: give more detail tutoring instructions about the usage of linux and make it more clear for us to build up the running environment, instead of helping us by merely sending us a packaged virtual machine, which has already built up the pintos environment.

>> Any other comments?

No