OS lab1 report - Team 16

Part 1: Argument Passing

I. push_argument() implementation details

Step 1:

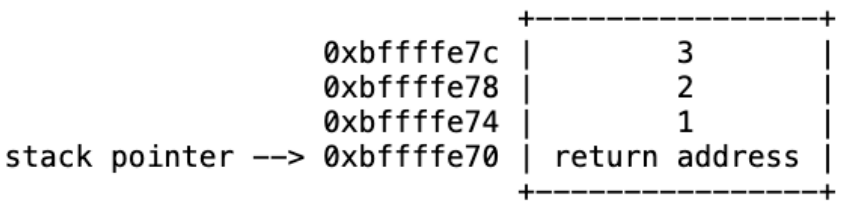
```
for (token = strtok_r (cmdline, " ", &save_ptr); token != NULL;
     token = strtok_r (NULL, " ", &save_ptr)) {
  if(i == 0) {
    thread_current()->cmd = palloc_get_page (0);
    strlcpy(thread_current()->cmd, token, strlen(token)+1);
  }
  tokens[i] = token;
  i++;
}
```

1.Use strtok_r() to disassemble the command line into multiple strings
2.Save the strings to tokens[], for the purpose of pushing onto the stack in ordering

Step 2:

```
int j, len;
for (j = i; j > 0; j--) {
  len = strlen(tokens[j-1]);
  tokensLen += len + 1;
  espChar -= len + 1;
  strlcpy(espChar, tokens[j-1], len+1);
  tokens[j-1] = espChar;
}
```

1.Push arguments onto the stack from right to left
2.Right to left cause we want to push the arguments into the stack in the reverse order as x86 convention, which means the last argument on the top of the stack(higher address)
3. after pushing the arguments into the stack, we use tokens to remember the addresses of the arguments



Step 3:

```
tokensLen = 4 - (tokensLen % 4);
for (j = 0; j < tokensLen; j++) {
  espChar--;
  *espChar = word_align;
}
```

1.Word-align: 4-byte alignment in x86 stack

2.Fill the stack with zeros (padding) so that the pointers pushed later are word-aligned

Step 4:

```
espChar -= 4;
*espChar = 0;
```

add null pointer

Step 5:

```
for (j = i; j > 0; j--) {
  espChar -= 4;
  *((int *)espChar) = (unsigned)tokens[j-1];
}
```

1.Push pointers to each argument

2.the stack then has the address to each argument, since now the values in tokens array are the addresses to the arguments

Step 6:

```
void *tmp = espChar;
espChar -= 4;
*((int *)espChar) = (unsigned)tmp;
/* put argc onto the stack */
espChar -= 4;
*espChar = i;
espChar -= 4;
*espChar = 0;
/* move esp to bottom of stack */
*esp = espChar;
```

1.the goal of this code block is to push the address of the first argument, the number of the arguments and fake return address into the stack

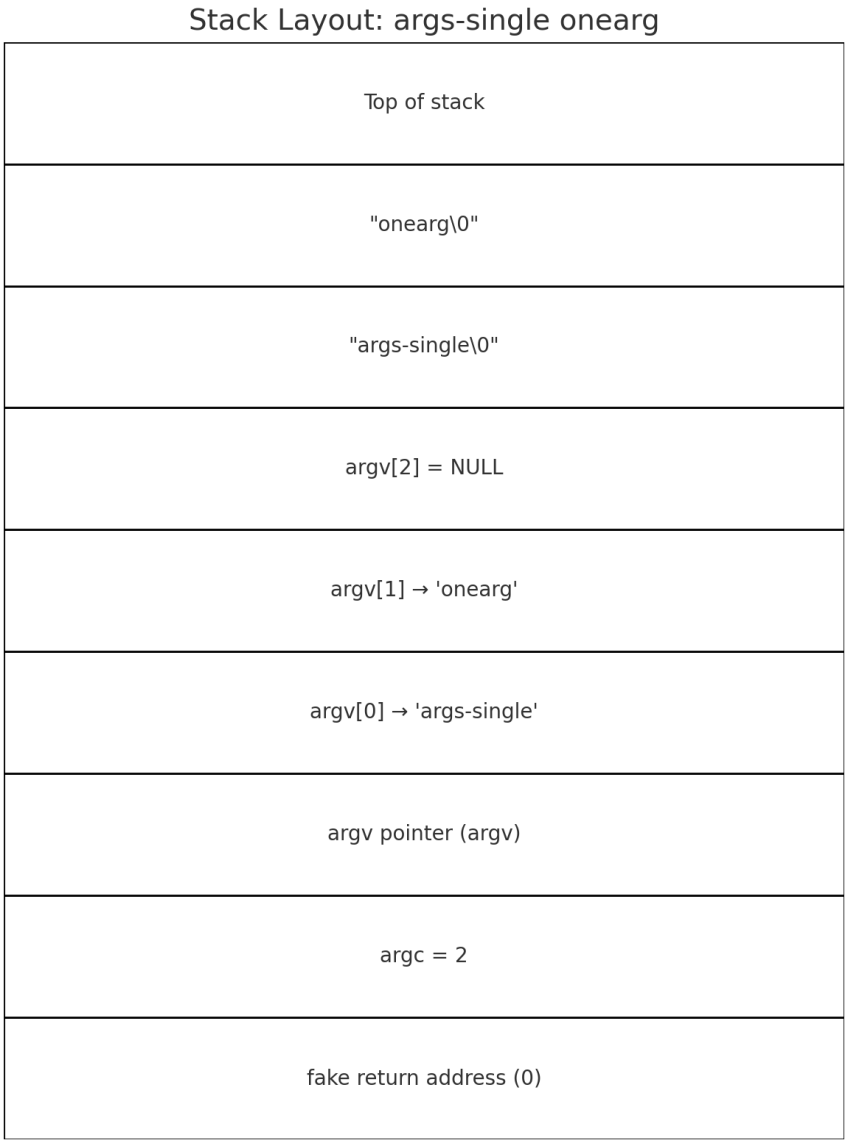
2.esp -> 0: fake return address 0

3.esp+4: the number of arguments

4.esp+8: the address of the first argument (argument 0)

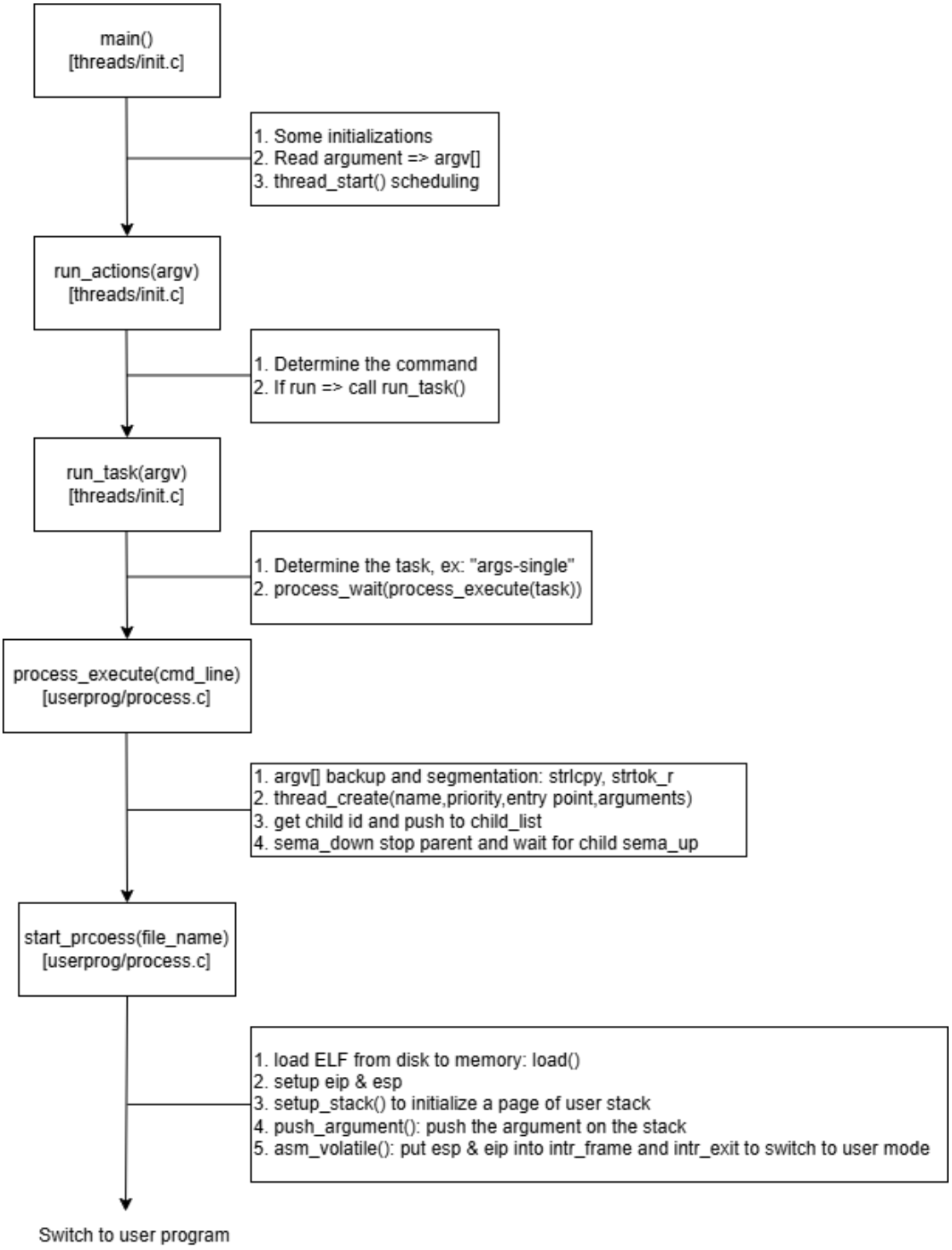
5.Set esp to the bottom of stack

II. Stack layout structured



Stack layout following the high to low address
Take “args-single onearg” as example

III. Function flow



Part 2: System Calls

I. Modifications in threads/thread.h

Below are some new fields added to struct head and their purpose.

- struct list openfiles:

Maintains a list of files opened by the thread. Each thread has its own set of open file descriptors, and this list is used to manage them individually.

- struct file *execfile:

Store the executable file that the thread is running. This is used to deny write access to the executable file while the process is running.

- struct list children:

A list of child processes spawned by this thread. This is used for managing parent-child relationships between processes.

- struct list_elem children:

List element used to link this thread into its parent's children list.

- int next_fd:

Used to assign file descriptors when new files are opened. It ensures that each new file opened by the thread gets a unique descriptor.

- char *cmd:

Stores the first command line arguments used to launch the thread. This is useful for debugging or reloading arguments in the future.

- struct semaphore load_sema:

Used to synchronize between parent and child during process_execute(). The parent waits until the child has loaded its executable.

- struct semaphore wait_sema:

Used to block the parent thread when it calls process_wait() until the child exits.

- struct semaphore exit_sema:

Used to ensure that the child doesn't free resources before the parent collects its exit status.

- int exit_status:

Stores the exit code of the thread so the parent can retrieve it via process_wait().

- int load_sucess:

Indicates whether the child successfully loaded its executable. Used by the parent to decide whether to proceed or fail in process_execute().

II. System call handling in syscall.c

Overall system call handling process:

1. User programs use the int 0x30 instruction to trigger a software interrupt, below is the syscall_init() function for interrupt handling. Whenever int 0x30 is invoked, control is transferred to the kernel's syscall_handler() function. The processor passes an intr_frame *f structure containing the user program's CPU state, including its stack pointer (f->esp) and return register (f->eax).

```
void syscall_init (void)
{
  intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");
  lock_init (&filesys_lock);
}
```

2. Then the syscall_handler() will retrieve the system call number first from the user stack. Before accessing this memory, the kernel ensures that both the system call number (esp[0]) and the first argument (esp[1]) point to valid user memory using the valid_mem_access() function.

```
void *esp = f->esp;
uint32_t *eax = &f->eax;
int syscall_num;
if(!valid_mem_access ( ((int *) esp) ))
  sys_exit (-1);
if(!valid_mem_access ( ((int *) esp)+1 ))
  sys_exit (-1);
syscall_num = *((int *) esp);
```

The implementation of valid_mem_access is described as below. This function checks whether a given user-provided pointer (up) points to valid, accessible user space memory. If it is invalid, it terminates the process immediately with exit_status = -1.

```c
static bool valid_mem_access (const void *up)
{
  struct thread *t = thread_current ();

  if (up == NULL)
    return false;
  if (is_kernel_vaddr (up))
    return false;
  if (pagedir_get_page (t->pagedir, up) == NULL)
    return false;
  uint8_t *check_byteptr = (uint8_t *) up;
  for (uint8_t i = 0; i < 4; i++)
  {
    if (get_user(check_byteptr + i) == -1)
    {
      t -> exit_status = -1;
      thread_exit();
    }
  }
  return true;
}
```

3. Then a switch statement matches the syscall number to the correct kernel implementation:

```c
if (syscall_num >= 0 && syscall_num < MAX_SYSCALL) {
    switch (syscall_num){
      case SYS_HALT:{
        sys_halt();
        break;
      }
      case SYS_EXIT:{
        int status = *(((int *) esp) + 1);
        sys_exit(status);
        break;
      }
      case SYS_EXEC:{
        const char *cmd_line = *(((char **) esp) + 1);
        *eax = (uint32_t) sys_exec (cmd_line);
        break;
      }
      case SYS_WAIT: {
        pid_t pid = *(((pid_t *) esp) + 1);
        *eax = (uint32_t) sys_wait (pid);
        break;
      }
      case SYS_CREATE: {
        const char *file = *(((char **) esp) + 1);
        unsigned initial_size = *(((unsigned *) esp) + 2);
        *eax = (uint32_t) sys_create (file, initial_size);
        break;
      }
```

4. After the syscall handler finishes, control returns to user mode with the result in eax

Our implementation of each system call is described as below.

- sys_exit():

In syscall_handler, the argument is retrieved from the user stack using pointer arithmetic on f->esp.

```
case SYS_EXIT:{
  int status = *((int *) esp) + 1);
  sys_exit(status);
  break;
}
```

After that, the current thread's exit_status field is updated so that the parent can retrieve the value via process_wait(). thread_exit() is then called to cleanly terminate the current thread and release its resources.

```
void sys_exit(int status){
  struct thread *cur = thread_current();
  cur->exit_status = status;
  thread_exit();
}
```

- sys_exec():

In syscall_handler(), the cmdline argument is retrieved from the user stack:

```
const char *cmd_line = *(((char **) esp) + 1);
```

Error handling: we ensure the pointer to the string is safe before it's passed to process_execute().

Then the current process requests to execute another process described by cmdline. The function process_execute() parses the command string and starts the child thread. The return thread ID is passed back to the user via f->eax.

```
pid_t sys_exec(const char *cmdline){
  tid_t child_tid = TID_ERROR;

  if(!valid_mem_access(cmdline))
  {
    sys_exit (-1);
  }
  child_tid = process_execute (cmdline);
  return child_tid;
}
```

- sys_wait():

In syscall_handler(), the pid argument is extracted from the user stack like:

```
case SYS_WAIT: {
  pid_t pid = *(((pid_t *) esp) + 1);
  *eax = (uint32_t) sys_wait (pid);
  break;
}
```

Then sys_wait() directly delegates to process_wait() which handles child management, blocking and synchronization. It returns the child's exit status if successful, otherwise, return -1

```
int sys_wait(pid_t pid){
  return process_wait (pid);
}
```

- sys_create():

In syscall_handler(), two arguments are retrieved from the user stack. (1)file is a pointer to the name of the file to create (2)initialize_size is the number of bytes the file should initially contain.

```
case SYS_CREATE: {
  const char *file = *(((char **) esp) + 1);
  unsigned initial_size = *(((unsigned *) esp) + 2);
  *eax = (uint32_t) sys_create (file, initial_size);
  break;
}
```

Error handling:

- The pointer file is validated inside sys_create() using valid_mem_access(file).
- If the file is NULL, points to kernel space, or unmapped memory, the process is immediately terminated via sys_exit(-1).
- If filesys_create() fails (e.g., due to existing file or file system error), the function returns false.

We use filesys_lock to avoid race conditions, this ensures no other thread modifies the file system while the file is being created.The file creation is performed by filesys_create() in filesys/filesys.c.

```
bool sys_create(const char *file, unsigned initial_size){
   bool retval;
  if(valid_mem_access(file)) {
    lock_acquire (&filesys_lock);
    retval = filesys_create (file, initial_size);
    lock_release (&filesys_lock);
    return retval;
  }
  else
    sys_exit (-1);

  return false;
}
```

- sys_remove():

In syscall_handler(), the argument file is retrieved from the user stack, it retrieves the first argument to remove(file), which is a pointer to the file name string.
As with other syscalls, the kernel ensures esp and esp + 1 are valid before accessing them. After that, the file itself is validated again in the syscall function.

```
case SYS_REMOVE: {
  const char *file = *(((char **) esp) + 1);
  *eax = (uint32_t) sys_remove (file);
  break;
}
```

Error handling:

- If the file pointer is NULL, or points to an invalid or unmapped address, valid_mem_access(file) fails, and the process is terminated.
- If the file does not exist or cannot be deleted (e.g., it is in use or protected), filesys_remove() will return false.

We use filesys_lock to avoid race conditions, this ensures that other threads don't interfere with the remove operation. The file remove is performed by filesys_remove() in filesys/filesys.c.

```
bool sys_remove(const char *file){
  bool retval;
  if(valid_mem_access(file)) {
    lock_acquire (&filesys_lock);
    retval = filesys_remove (file);
    lock_release (&filesys_lock);
    return retval;
  }
  else
    sys_exit (-1);

  return false;
}
```

- sys_open():

In syscall_handler(), the argument file is retrieved from the user stack, this pointer should reference a valid, null-terminated file name string in user space. The memory at esp and esp + 1 is validated using valid_mem_access() before retrieving this pointer.

```
case SYS_OPEN: {
  const char *file = *(((char **) esp) + 1);
  *eax = (uint32_t) sys_open (file);
  break;
}
```

Error handling:

- If the file is NULL or an invalid user pointer (unmapped or pointing to kernel space), it fails the valid_mem_access(file) check and immediately calls sys_exit(-1) to terminate the process.
- If the file does not exist or cannot be opened, filesys_open() returns NULL, and sys_open() returns -1 to the user.

Each thread has a private next_fd counter that tracks the next available file descriptor. A new struct openfile is dynamically allocated using palloc_get_page(0). The fd field is set to the current thread's next_fd, and next_fd is incremented. And the new openfile object is added to the current thread's openfiles list.
Also we use filesys_lock to avoid race conditions, and call the filesys_open() function in filesys/filesys.c.

```
int sys_open(const char *file){
  if(valid_mem_access ((void *) file)) {
    struct openfile *new = palloc_get_page (0);
    new->fd = thread_current ()->next_fd;
    thread_current ()->next_fd++;
    lock_acquire (&filesys_lock);
    new->file = filesys_open(file);
    lock_release (&filesys_lock);
    if (new->file == NULL)
      return -1;
    list_push_back(&thread_current ()->openfiles, &new->elem);
    return new->fd;
  }
  else
    sys_exit (-1);

  return -1;
}
```

- sys_filesize():

In syscall_handler(), the file descriptor is retrieved from the user stack. fd is the first argument to filesize(fd).The memory at esp and esp + 1 is validated before accessing this value using valid_mem_access() to ensure safe memory access.

```
case SYS_FILESIZE: {
  int fd = *(((int *) esp) + 1);
  *eax = (uint32_t) sys_filesize (fd);
  break;
}
```

The function getFile(fd) is used to search for the file associated with the given fd in the current thread's openfiles list. If the file is not found (of == NULL), the syscall returns 0. And the call to file_length() is a file system operation, so it's wrapped with a lock to prevent concurrent access issues. The file size is estimated by the function file_length() in filesys/file.c.

```
int sys_filesize(int fd){
  int retval;
  struct openfile *of = NULL;
  of = getFile (fd);
  if (of == NULL)
    return 0;
  lock_acquire (&filesys_lock);
  retval = file_length (of->file);
  lock_release (&filesys_lock);
  return retval;
}
```

The implementation of get_file() is described below. This function looks up and returns the struct openfile * associated with a given file descriptor fd for the current thread. If the file descriptor is not valid (i.e., not found), it returns NULL.

1. Get the current thread: struct thread *t = thread_current();
   Each thread maintains its own list of open files.
2. Iterate over the open file list: The function goes through each list_elem in t->openfiles.
3. Extract struct openfile * from list element:Using list_entry(e, struct openfile, elem).
4. Compare fd values: If the of->fd matches the input fd, the function returns the matching openfile pointer.
5. If no match is found: Returns NULL.

```
static struct openfile *
getFile (int fd)
{
  struct thread *t = thread_current ();
  struct list_elem *e;
  for (e = list_begin (&t->openfiles); e != list_end (&t->openfiles);
       e = list_next (e))
    {
      struct openfile *of = list_entry (e, struct openfile, elem);
      if(of->fd == fd)
        return of;
    }
  return NULL;
}
```

- sys_read():

In syscall_handler(), the three arguments are retrieved from the user stack. (1)fd: file descriptor to read from (2)buffer: pointer to the memory location to store the data (3)size: number of bytes to read

```
case SYS_READ: {
  int fd = *(((int *) esp) + 1);
  void *buffer = (void *) *(((int **) esp) + 2);
  unsigned size = *(((unsigned *) esp) + 3);
  *eax = (uint32_t) sys_read (fd, buffer, size);
  break;
}
```

Error handling:

- The buffer pointer is validated using valid_mem_access() before being accessed
- If the file descriptor is invalid (i.e., not found in openfiles), the function returns -1.
- For stdin (fd = 0), input_getc() is used to read one byte at a time from the keyboard.

If the fd corresponds to an open file, we locate the file via getFile(fd) and use file_read() to read the requested bytes. Also, we wrap the file operation with filesys_lock to ensure synchronization.

```
int sys_read(int fd, void *buffer, unsigned size){
  int bytes_read = 0;
  char *bufChar = NULL;
  struct openfile *of = NULL;
  if (!valid_mem_access(buffer))
    sys_exit (-1);
  bufChar = (char *)buffer;
  if(fd == 0) {
    while(size > 0) {
      input_getc();
      size--;
      bytes_read++;
    }
    return bytes_read;
  }
  else {
    of = getFile (fd);
    if (of == NULL)
      return -1;
    lock_acquire (&filesys_lock);
    bytes_read = file_read (of->file, buffer, size);
    lock_release (&filesys_lock);
    return bytes_read;
  }
}
```

- sys_write():

In syscall_handler(), the three arguments are retrieved from the user stack. (1)fd: file descriptor to read from (2)buffer: pointer to the memory location to store the data (3)size: number of bytes to read

```
case SYS_WRITE: {
  int fd = *(((int *) esp) + 1);
  const void *buffer = (void *) *(((int **) esp) + 2);
  unsigned size = *(((unsigned *) esp) + 3);
  *eax = (uint32_t) sys_write (fd, buffer, size);
  break;
}
```

Error handling:

- If buffer is invalid (e.g., NULL, kernel address, or unmapped), the system immediately exits
- If getFile(fd) returns NULL (i.e., invalid or closed file descriptor), the function returns 0.

When fd == 1, the function calls putbuf() to write data to the console. To avoid writing large chunks at once, the buffer is broken into chunks of size BUF_MAX.
If fd refers to a regular file, we locate it using getFile(fd) and call file_write() in filesys/filesys.c.

```
int sys_write(int fd, const void *buffer, unsigned size){
  int bytes_written = 0;
  char *bufChar = NULL;
  struct openfile *of = NULL;
  if (!valid_mem_access(buffer)){
    sys_exit (-1);
  }
  bufChar = (char *)buffer;
  if(fd == 1) {
    /* break up large buffers */
    while(size > BUF_MAX) {
      putbuf(bufChar, BUF_MAX);
      bufChar += BUF_MAX;
      size -= BUF_MAX;
      bytes_written += BUF_MAX;
    }
    putbuf(bufChar, size);
    bytes_written += size;
    return bytes_written;
  }
  else {
    of = getFile (fd);
    if (of == NULL)
      return 0;
    lock_acquire (&filesys_lock);
    bytes_written = file_write (of->file, buffer, size);
    lock_release (&filesys_lock);
    return bytes_written;
  }
}
```

- sys_seek():

In syscall_handler(), the two arguments are retrieved from the user stack. (1)fd is the target file descriptor (2)position is the new offset to seek to

```
case SYS_SEEK: {
  int fd = *(((int *) esp) + 1);
  unsigned position = *(((unsigned *) esp) + 2);
  sys_seek (fd, position);
  break;
}
```

The function calls getFile(fd) to find the corresponding open file from the thread's openfiles list. If getFile() returns NULL (invalid or closed file descriptor), the function simply returns without performing any action. And also, to safely update the file's internal position pointer, the file system operation is wrapped with filesys_lock. The file_seek is called from filesys/file.c.

```
void sys_seek(int fd, unsigned position){
  struct openfile *of = NULL;
  of = getFile (fd);
  if (of == NULL)
    return;
  lock_acquire (&filesys_lock);
  file_seek (of->file, position);
  lock_release (&filesys_lock);
}
```

- sys_tell():

In syscall_handler(), the argument fd is retrieved from the user stack.

```
case SYS_TELL: {
  int fd = *(((int *) esp) + 1);
  *eax = (uint32_t) sys_tell (fd);
  break;
}
```

The function uses getFile(fd) to retrieve the corresponding struct openfile *. If the file descriptor is invalid or not found, the function returns 0. The file_tell is a call from filesys/file.c.

```
unsigned sys_tell(int fd) {
  unsigned retval;
  struct openfile *of = NULL;
  of = getFile (fd);
  if (of == NULL)
    return 0;
  lock_acquire (&filesys_lock);
  retval = file_tell (of->file);
  lock_release (&filesys_lock);
  return retval;
}
```

- sys_close():

In syscall_handler(), the argument fd is retrieved from the user stack.

```
case SYS_CLOSE: {
  int fd = *(((int *) esp) + 1);
  sys_close (fd);
  break;
}
```

The function uses getFile(fd) to locate the struct openfile * corresponding to the given file descriptor. If the file descriptor is invalid or already closed (of == NULL), the function returns silently without performing any operation. This structure is maintained per-thread in the openfiles list.

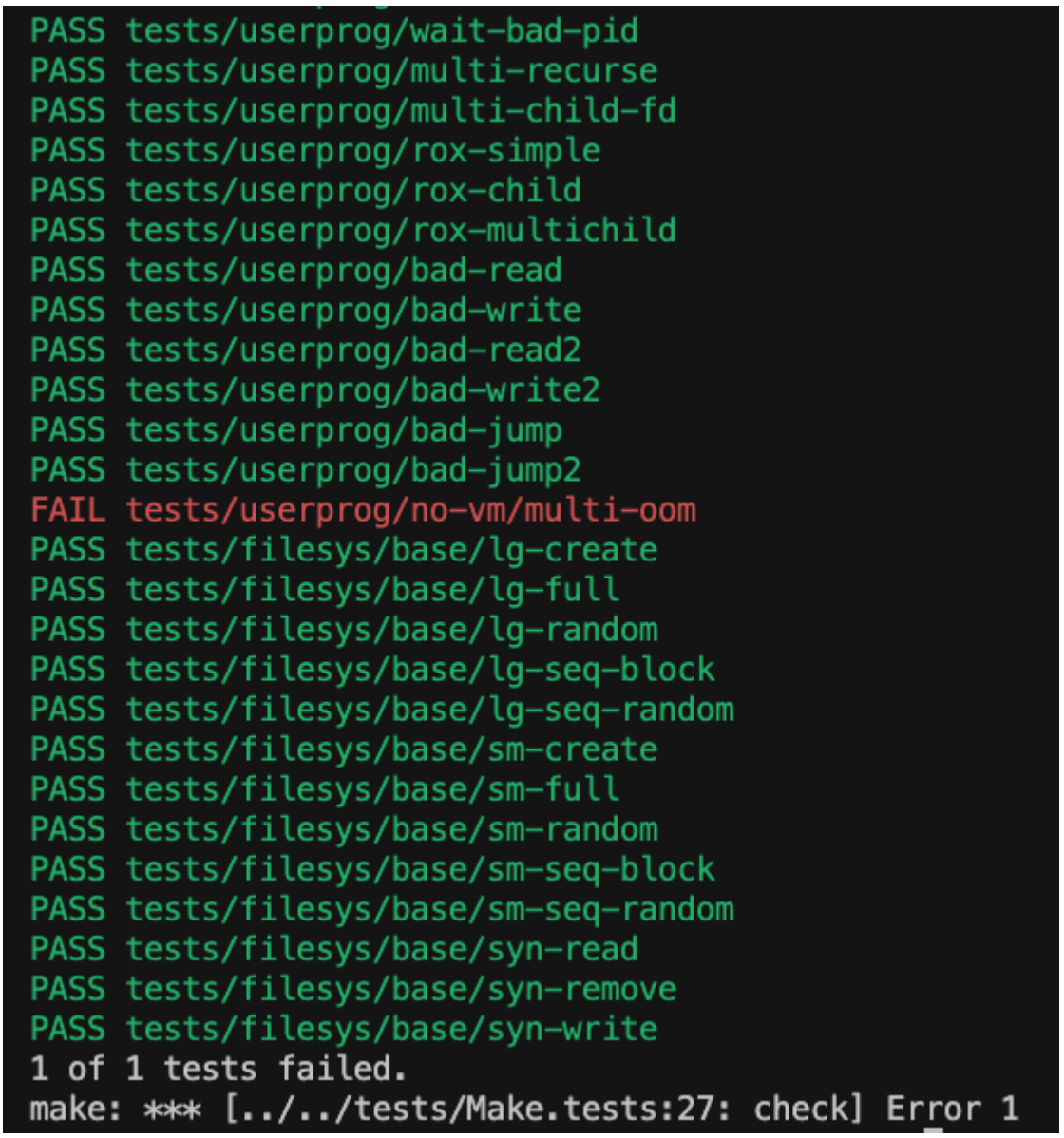
Also added filesys_lock to protect the operation on open resources.

After closing the file, the openfile structure is removed from the current thread's openfiles list. The memory allocated for the struct openfile is released.

```
void sys_close(int fd) {
  struct openfile *of = NULL;
  of = getFile (fd);
  if (of == NULL)
    return;
  lock_acquire (&filesys_lock);
  file_close (of->file);
  lock_release (&filesys_lock);
  list_remove (&of->elem);
  palloc_free_page (of);
}
```

III. Test results

```
PASS tests/userprog/args-none
PASS tests/userprog/args-single
PASS tests/userprog/args-multiple
PASS tests/userprog/args-many
PASS tests/userprog/args-dbl-spac
PASS tests/userprog/sc-bad-sp
PASS tests/userprog/sc-bad-arg
PASS tests/userprog/sc-boundary
PASS tests/userprog/sc-boundary-2
PASS tests/userprog/sc-boundary-3
PASS tests/userprog/halt
PASS tests/userprog/exit
PASS tests/userprog/create-normal
PASS tests/userprog/create-empty
PASS tests/userprog/create-null
PASS tests/userprog/create-bad-pt
PASS tests/userprog/create-long
PASS tests/userprog/create-exists
PASS tests/userprog/create-bound
PASS tests/userprog/open-normal
PASS tests/userprog/open-missing
PASS tests/userprog/open-boundary
PASS tests/userprog/open-empty
PASS tests/userprog/open-null
PASS tests/userprog/open-bad-ptr
PASS tests/userprog/open-twice
PASS tests/userprog/close-normal
PASS tests/userprog/close-twice
PASS tests/userprog/close-stdin
PASS tests/userprog/close-stdout
PASS tests/userprog/close-bad-fd
PASS tests/userprog/read-normal
PASS tests/userprog/read-bad-ptr
PASS tests/userprog/read-boundary
PASS tests/userprog/read-zero
PASS tests/userprog/read-stdout
PASS tests/userprog/read-bad-fd
PASS tests/userprog/write-normal
PASS tests/userprog/write-bad-ptr
PASS tests/userprog/write-boundar
PASS tests/userprog/write-zero
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-bound
PASS tests/userprog/exec-bound-2
PASS tests/userprog/exec-bound-3
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
```



We failed the **multi-oom** test, which recursively executes itself until the child process fails to load.

We observed that our program gets stuck, indicating that it may have entered a deadlock or is waiting indefinitely.

We suspect two possible causes:

First, we may have used semaphores incorrectly — for example, calling sema_down in the wrong location.

Second, our program may not be properly rejecting memory allocations when memory is insufficient.