Operating Systems – Project 2: User Program

Final Report

Khau Lien Kiet
Team 10

2022/11/1

Department of Computer Science & Engineering
Pohang University of Science and Technology (POSTECH)

Table of Contents

I. Process Termination Message & Argument Passing

a. Implementation of Process Termination Message

- Data Structure:
  - **Add thread.h/struct thread/exit_status:** Under the structure of thread in the user program section, it is necessary to initialize the status of one thread to get terminated.

```
struct thread
 {
   //..
#ifdef USERPROG
   //..
  /* Project 2: Process Termination Message section*/
   int exit_status;                    // Stores the status if it exists
   //..
#endif
   //..
```

- Algorithm
  - **Initialization of exit_status:** To implement it, it is necessary to modify *threads/thread.c/init_thread()* by setting the initial number for *exit_status* as -1.

```
static void
init_thread (struct thread *t, const char *name, int priority)
{
//..
 /* Project 2: Process Termination Message section  */
 t->exit_status = -1;
//..
}
```

  - **Add userprog/syscall.c/syscall_exit():** This function is one of the system calls in the section II. However, in terms of this domain, it would print out the message of the thread name and its *exit_status*.

```
void syscall_exit (int status)
{
  thread_current()->exit_status = status;
  /* Project 2: Process Termination Message: When the user program is exited, the status is printed
and return back to the Kernel mode.*/
  printf("%s: exit(%d)\n", thread_current()->name, status);
  thread_exit ();
}
```

b. *Implementation of Argument Passing*

- Data Structure: no new variable or data structure added
- Algorithm
    - **Modify process_execute(const char *file_name):** As stated in the Design Report, there would be a new function called *pars_file_name()* which is added inside the *process_execute()*. However, to optimize it, I directly the implementation of *pars_file_name()* inside *process_execute()*. Basically, the main idea is to extract the file name, which is the first argument of the command line, by parsing with the function *strtok_r()*.

```
void pars_filename (char *cmd)
{
  char *ptr;
  cmd = strtok_r (cmd, " ", &ptr);
}

tid_t
process_execute (const char *file_name)
{
  char *fn_copy;
  tid_t tid;

  /* Make a copy of FILE_NAME.
     Otherwise there's a race between the caller and load(). */
  fn_copy = palloc_get_page (0);
  if (fn_copy == NULL)
    return TID_ERROR;
  strlcpy (fn_copy, file_name, PGSIZE); /* Makes a copy of the entire command line string, args included. */
  /* Project 2: Argument Passing: This function is performed as pars_file_name()*/
  // Its functionality is to extract a file name
  char * ptr;
  char * file_name_parsed = strtok_r((char *)file_name, " ", &ptr);
  // If the file_name_parsed does not have a meaning, it would return an error
  if (file_name_parsed == NULL)
    return TID_ERROR;

  /* Create a new thread to execute FILE_NAME_PARSED. */
  tid = thread_create (file_name_parsed, PRI_DEFAULT, start_process, fn_copy);
```

```
  //..
  return tid;
}
```

- **Modify load (const char *file_name, void **eip, void **esp):** As described in the Design Report, there would be a new function called *pars_argument()*, which is to separate each argument of the command line inside the *argv* list and also return the number of arguments of that command line. Hence, its implementation would be embedded inside function *load()* to optimize its structure.

```
int pars_argument (char *cmd, char **argv)
{
  char *token, *ptr;
  int argc = 0;
  for (token = strtok_r (cmd, " ", &ptr); token != NULL;
  token = strtok_r (NULL, " ", &ptr), argc++)
  {
    argv[argc] = token;
  }
  return argc;
}


bool
load (const char *file_name, void (**eip) (void), void **esp)
{
//..
  /* Project 2: Arugment Passing - This function is the idea of pars_argument()*/
  // The command line passing through the program. Besides extracting the command line with tokening, it
would detect the number of arguments with the argc count.
  char *token, *save_ptr;
  char *argv[128];
  int argc = 0;
  for(token = strtok_r((char *)file_name, " ", &save_ptr); token != NULL; token = strtok_r(NULL, " ",
&save_ptr))
  {
    argv[argc] = token; // Add the token toward the command line argument
    argc++;           // Add up the number of arguments in the command line
  }
```

```
//..
}
```

- **Modify userprog/syscall.c/setup_stack**(): When finishing loading by *load()* function, it would try to initialize the user stack for those command line input. Its idea is the same as *init_user_stack()* as illustrated in the Design Report but it is now integrated inside the modification of *setup_stack()*. To recall its implementation, we would decrement the *esp* pointer by certain integer to put the content inside the user stack. First, we would initialize a list of address inside the stack. Secondly, we would create enough space for adding the argument string inside the stack as well as its address. Indeed, it is essential to put those argument strings from right to left (descending order). Next, to guarantee the space would be the factor of 4, it is necessary to align the stack and push *NULL* to pad. Also, we also push the address of those argument string inside the argument list *(arg[])*. Then, it is required to push inside the stack with the starting address of the first parsing strings *(argc)* and the number of arguments in the command line *(argv)*. Lastly, we could push the fake address to return the user stack.

```
static bool setup_stack (void **esp, int argc, char *argv[])
{
  uint8_t *kpage;
  bool success = false;
  int i;

  kpage = palloc_get_page (PAL_USER | PAL_ZERO);
  if (kpage != NULL)
    {
      success = install_page (((uint8_t *) PHYS_BASE) - PGSIZE, kpage, true);
      if (success)
      {
        /* Project 2: This implementation is performed as init_user_stack() as
        illustrated in the Design Report 2 */
        *esp = PHYS_BASE - 12;

        // Initialize the list of adddress that is added inside the stack
        uint32_t * arg[argc];

        //Add the argument string in the descending order
        for(i = argc - 1; i >= 0; i--)
        {
```

```
        // Create enough space for argument string to be added inside the user stack as well as its address
inside the array
          *esp = *esp - sizeof(char)*(strlen(argv[i])+1);
          memcpy(*esp, argv[i], sizeof(char)*(strlen(argv[i])+1));
          arg[i] = (uint32_t *)*esp;
        }
        // Allign the stack to be the size of 4 and Push NULL
        *esp -=  4;
        **(uint32_t **)esp = 0;

        // Push the address inside the arg[i]
        *esp -=  4;
        for(i = argc - 1; i >= 0; i--)
        {
          **(uint32_t **)esp = arg[i];
          *esp -=  4;
        }
```

II. System Call

a. Implementation of System Call Handler

- Data Structure: No new variable and data structure added
- Algorithm
    - **Add userprog/syscall.c/address_validation(const void *addr):** This function returns the boolean value, which indicate the address if it is verified or not. Without harming to the kernel and other running processes, this function could detect the invalidity of pointers and let the "bad" process get terminated. To implement it, there are several conditions to check whether the addr below the *PHYS_BASE* or above the *STACK_BOTTOM* or its address is the user address. If it does not satisfy those conditions and its address is *NULL*, we call *syscall_exit()* to get terminated that process.

```
#define PHYS_BASE 0xc0000000
#define STACK_BOTTOM 0x8048000
....
/* Project 2: System Call - This function is to check which value is in the kernel space for some arguments, which is
inside the STACK_BOTTOM and PHYS_BASE. Also, it would verify the user virtual address or the address is NULL
or not. If it is validated, we could call system call to get terminated for exiting that thread.*/
void address_validation (const void *addr)
{
  if(!is_user_vaddr(addr) || addr == NULL || addr < (void *) STACK_BOTTOM || addr > PHYS_BASE)
    syscall_exit(-1);
}
```

    - **Add userprog/syscall.c/copy_argument(struct intr_frame *f, int *args, int count):** This function is to divide the argument from the user stack and add them inside the *args[]* list. Its main functionality to copy the argument from the user stack to the kernel. By implementing it, it is required to check the address passing in whether valid or not. If it does not satisfy the required conditions inside the *address_validation()* function, it would get terminated. Lastly, those arguments would be passed inside that array.

```
/* Project 2: System Call - This function is to extract and add them inside the array.
The main functionality is to get the argument from the user stack to kerenl. Inside we have to check the address is
valid or not in order to get the accurate arguments. */
static void copy_arguments (struct intr_frame *f, int *args, int count)
{
  int i;
  for (i = 0; i < count; i++)
    {
      int *ptr;
```

```
      ptr = (int *) f->esp + i + 1;
      address_validation(ptr);
      args[i] = *ptr;
    }
}
```

- **Modify syscall_handler(struct intr_frame *f):** After receiving the interrupt from the user program, there would appear a notification of the Kernel to call *syscall_handler()* to handle the system call from the user program. To implement inside its function, it is necessary to validate those address by *address_validation()* and copy the arguments by *copy_argument()* function inside each case of system calls. Therefore, it is necessary to create a list *args[]* to hold the stack arguments got from the system call's input. Inside each case of system call, it is required to use the system call number to let the syscall handler notice which case that it needs to tackle. Then, it should have a return value for that system call into the *eax* register after executing those system calls. Indeed, inside this function, initialize a *phys_page_ptr*, which is a converted pointer to the physical page, that is used for check its address valid or not for some types of system calls below ***(detailed implementation in II.b).***

```
* Project 2: When invoking to using a user program, the system call handles would
tackle a system call from a user program. */
static void syscall_handler (struct intr_frame *f UNUSED)
{
    address_validation(f->esp); // Check the process is valid or not. If not, it would be terminated
    int args[3]; // Create a list to hold the stack arguments which gets input from the system call
    //This pointer would be directly pointed to the physical page. This one is used to check the converted address is
valid or not.
    void * phys_page_ptr;

    /* The main implementation is to integrate the system call number for system call handlers in a particular case.
Generally, its return value should be saved in the eax register */
    switch(*(int *) f->esp)
    {
      case SYS_HALT:
        /* It calls syscall_halt() to shutdonw PINTOS */
        syscall_halt();
        break;

      case SYS_EXIT:
```

```
        /* It calls syscall_exit() to terminate the current user program and return the
        status toward the Kernel */
        copy_arguments(f, &args[0], 1);
        syscall_exit(args[0]);
        break;

//..
}
```

- **Modify userprog/exception.c/page_fault():** In terms of memory management, there would occur some memory leakage after the process get terminated but it is still held the lock. Therefore, it is necessary to call *syscall_exit()* under the *page_fault()* function if those cases are shown up to prevent memory leaks.

```
static void
page_fault (struct intr_frame *f)
{
//..
  /* Project 2: System Call - This implementation is to prevent the memory leakage.
  At the end of process, all allocated memory would be released. After allocating the memory, it would check to be
sucessfully loaded. If not, that process would be terminated.*/
  syscall_exit(-1);
//..
}
```

b. Implementation of System Call for the user process-level manipulation

- Data Structure:
  - **threads/thread.h/struct thread/list_child_process:** Under the structure of thread, there is a list contained the child process (considered as the sibling)
  - **threads/thread.h/struct thread/elem_child_process:** To keep track each element in the *list_child_process*, there is a variable to control those elements in that list.
  - **threads/thread.h/struct thread/mutex:** It is a semaphore for the parent process. While it is waiting for its children to get terminated, it would sleep inside this *mutex*.

```
struct thread
  {
//..
#ifdef USERPROG
```

```
    //..
   /* Project 2: System call in the process-level*/
    struct list list_child_process;       // List containing each child process which considered as the siblings
    struct list_elem elem_child_process;  // Element for keeping track in the list_child_process.
    struct semaphore mutex;               // While waiting for the child_process, parent process should be sleeped in this
mutex


//..
#endif
    //..
  };
```

- Algorithm:
  - Initialization of new data structure
    - **Modify threads/thread.c/init_thread():** To perform the initialization, we should initialize *list_child_process* with *list_init()*, and assign the initial number of semaphore *mutex* as 0 (performed as conditional variable)

```
static void
init_thread (struct thread *t, const char *name, int priority)
{
//..
  /* Project 2: System Call related to the process-level */
  /* Initialization of list_child_process, mutex semaphore, and the exit_status when the thread
  is initialized.*/
  list_init(&t->list_child_process);
  sema_init(&t->mutex, 0);
//..}
```

  - **Add userprog/syscall.c/syscall_halt():** When *syscall_handler()* notifies the halt system call, it would allocate to *syscall_halt()*. The main implementation is to integrate *shutdown_power_off()* to turn off the PINTOS system.

```
* Project 2: When invoking to using a user program, the system call handles would
tackle a system call from a user program. */
static void syscall_handler (struct intr_frame *f UNUSED)
{
//..
    switch(*(int *) f->esp)
    {
      case SYS_HALT:
```

```
        /* It calls syscall_halt() to shutdonw PINTOS */
        syscall_halt();
        break;
      //..

    }
}
void syscall_halt (void) {shutdown_power_off();}
```

- **Add userprog/syscall.c/syscall_exit():** When *syscall_handler()* notifies the exit system call, it would allocate to *syscall_exit()* by passing the first argument as the status. Its implementation is to get the *exit_status* and print out the thread status when it gets terminated.

```
static void syscall_handler (struct intr_frame *f UNUSED)
{
//..
    switch(*(int *) f->esp)
    {
case SYS_EXIT:
        /* It calls syscall_exit() to terminate the current user program and return the status toward the Kernel */
        copy_arguments(f, &args[0], 1);
        syscall_exit(args[0]);
        break;
      //..

    }
}
void syscall_exit (int status)
{
  thread_current()->exit_status = status;
  /* Project 2: Process Termination Message: When the user program is exited,
  the status is printed and return back to the Kernel mode.*/
  printf("%s: exit(%d)\n", thread_current()->name, status);
  thread_exit ();
}
```

- **Add userprog/syscall.c/syscall_wait():** When *syscall_handler()* notifies the wait system call, it would allocate to *syscall_wait()* by passing the first argument as its pid and return its child *exit_status* toward the *eax* register. Therefore, it is

necessary to modify *process_wait()* for *syscall_wait()* implementation. Indeed, there is an list element to iterate through the *list_child_process*. If there is no child process in the *list_child_process* or *NULL* child process, it would return -1. Also, if there are child processed inside that list, it would check again whether it is a child process for its parent or not. If so, it is required to remove that child process out of the list to perform its execution and put the parent process into sleep by waiting for the *mutex*. After the child process gets terminated, *exit_status* would be returned.

```
static void syscall_handler (struct intr_frame *f UNUSED)
{
//..
      case SYS_WAIT:
       /* It calls the syscall_wait() is to wait for the a child process pid
       to finish executed or exit and receive its child exit status. */
       copy_arguments(f, &args[0], 1);
       f->eax = syscall_wait((pid_t) args[0]);
       break;
      //..
    }
}
int syscall_wait (pid_t pid)
{
  /* Implement process_wait(). If the child process is a valid thread, then disable interupts, and add it to this
list_child_process. */
  return process_wait(pid);
}
int
process_wait (tid_t child_tid UNUSED)
{
  /* Project 2: System Call - Modification for syscall_wait()*/
  struct thread *child_thread = NULL;
  struct list_elem *e; // Element for the list_child_process for iteration
  // If the list_child_process, parent process does not wait and can return */
  if(list_empty(&thread_current()->list_child_process))
    return -1;
  // Check whether its child process or not
  for (e = list_front(&thread_current()->list_child_process); e != NULL; e = e->next)
  {
```

```
    struct thread *t = list_entry (e, struct thread, elem_child_process);
    if (t->tid == child_tid)
    {
      child_thread = t;
      break;
    }
  }
  // If it is not its child process, return out.
  if(child_thread == NULL)
    return -1;
  /* Remove the child from our lists of child threads, so that calling this
  function for a second time does not require additional waiting. */
  list_remove(&child_thread->elem_child_process);
  // Put the parent thread to sleep while waiting for its child thread whose PID was passed in.
  sema_down(&child_thread->mutex);
  // After its child processes is terminated, exit_status could be returned.
  return child_thread->exit_status;
}
```

- **Add userprog/syscall.c/syscall_exec():** When *syscall_handler()* notifies the execute system call, It would allocate to *syscall_exec()* by passing the first argument of the command line to execute the program. Before performing *syscall_exec(),* it is necessary to check *phys_page_ptr*, which is the pointer of physical page, whether it is valid or not. If not, *syscall_exit()* would be performed. Otherwise, it would keep going to execute *syscall_exec()* and return the *pid* toward the *eax* register. Inside its implementation, in order to receive the child *tid* as the return value, it would get from *process_execute()* function. Thus, we have to modify *process_execute()* for *syscall_exec()* implementation.

```
static void syscall_handler (struct intr_frame *f UNUSED)
{
//..
   switch(*(int *) f->esp)
   {
     case SYS_EXEC:
       /* It calls syscall_exec() is to allocate the first argument of the cmd line
       to execute the program */
       copy_arguments(f, &args[0], 1);
       phys_page_ptr = pagedir_get_page(thread_current()->pagedir, (const void *) args[0]);
```

```
      args[0] = phys_page_ptr;
      f->eax = syscall_exec((const char *) args[0]);
      break;
    //..
  }
}

pid_t syscall_exec (const char * file)
{
 /* If a file is NULL, return an error. */
 if(!file)
   return TID_ERROR;

 /* Return the PID of the child process */
 pid_t child_tid = process_execute(file);
 return child_tid;
}
```

- **Add new data structure**: Initialize a global variable as a thread structure – *matching_thread* – to look for its thread inside the list. Also, initialize a global variable as tid_t – *current_tid* – to be the tid of thread that is found inside the list.

```
/* Project 2: System Call for user process-level manipulation - syscall_exec() */
/* A global variable - the thread finds for in the thread list  */
static struct thread * matching_thread;

/* A global variable - the thread tid of a thread which finds its own thread list  */
static tid_t current_tid;
```

- **Add function userprog/process.c/get_thread_tid():** Create a new function to be a passing function for *thread_foreach()* ***(explained below).*** Its main functionality is to ensure that the *current_tid* is the same of the thread tid, thus assigning the *matching_thread* as that current thread.

```
/* Project 2: System Call for user process manipulation: A function to be
passed to the thread_foreach() function to find a thread tid.*/
void get_thread_tid (struct thread *t, void * aux UNUSED)
{
 if(current_tid == t->tid)
```

```
  {
    matching_thread = t;
  }
}
```

- **Modify userprog/process.c/process_execute():** In terms of the thread tid, if its tid is an error, it would free the page. Otherwise, it would add that thread inside the *list_child_process*. In terms of the *thread_foreach(),* a function *get_thread_tid()* would be passed in to check its valid thread. Indeed, when managing thread, it is necessary to disable interrupt and enable the interrupt after finishing its execution.

```
tid_t
process_execute (const char *file_name)
{
  //..
  if (tid == TID_ERROR)
  {
    palloc_free_page (fn_copy);
  }
  else
  {
    /* Project 2: System Call for the process-level for the syscall_exec() */
    //If the child process is a valid thread, then disable interupts and add it to this list_child_process.
    current_tid = tid;
    enum intr_level old_level = intr_disable ();
    thread_foreach(*get_thread_tid, NULL);
    list_push_front(&thread_current()->list_child_process, &matching_thread->elem_child_process);
    intr_set_level (old_level);
  }
  return tid;
}
```

c. Implementation of System Call for the file-level manipulation

- Data Structure:
  - **threads/thread.h/struct thread/fd_table**: Under the thread structure, we would allocate File Descriptor Table (FDT) to manipulate the file in one process. This data structure is a list to maintain all elements of the file for one process.
  - **threads/thread.h/struct thread/fd_count:** In addition, there would be an integer to keep track of the file descriptor inside FDT.

```
struct thread
```

```
 {
//..
#ifdef USERPROG
    //..
   /* Project 2: System call in the file-level*/
    struct list fd_table;                // List of file descriptors belonging to this thread.
    int fd_count;                        // An integer available file descriptor.
//..
#endif
    //..
 };
```

- **userprog/syscall.c/struct thread_file:** This data structure is performed as a file descriptor inside FDT, which is considered as one element for the list of file descriptors. Under its structure, there would be initialized three variables as following
    - **file_elem:** The variable is represented as the element of file descriptor inside FDT.
    - **file_addr:** The variable is considered the address pointer which is directly pointed to the file.
    - **fd_index:** The unique index belongs to its *thread_file*, thus it has a relation with *fd_count*.

```
/* Project 2: System Call related to the file-level */
/* Create a data structure of a thread_file to insert one file and their respective file descriptor (id) into the fd_table
which is the list structure for the current thread. */
struct thread_file
{
    struct list_elem file_elem;
    struct file *file_addr;
    int fd_index;
};
```

- Semaphore for synchronization between writer and multiple readers
    - **userprog/syscall.c/lock:** Lock is a synchronization primitive for the contention of multiple readers to grab the lock for reading in the buffer.
    - **userprog/syscall.c/w_or_r:** This synchronization is to allocate its turn for reader or write to execute inside the buffer.
    - **userprog/syscall.c/read_count:** This variable is number of readers to have it access to read the buffer.

```
/* Synchronization for multiple readers and writer inside the buffer */
struct semaphore lock, w_or_r;
int read_count;
```

- Algorithm
  - Initialization of File Descriptor Table and synchronization
    - **Modify threads/thread.c/init_thread():** Since *fd_table* is a list type, it is necessary to initialize it with *list_init()* in order to become a File Descriptor Table for holding a file inside the list. In addition, we have to initialize *fd_count* as 2 because 0 and 1 are for STDIN and STDOUT, respectively.

```
static void
init_thread (struct thread *t, const char *name, int priority)
{
//..
 /* Project 2: System Call related to the file-level*/
 /* Initialization of the File Descriptor Table which hold the file inside that list
 as well as the available file descriptor count (NOTE: 0 and 1 is for STDIN and STDOUT, respectively.*/
 list_init(&t->fd_table);
 t->fd_count = 2;
//..
}
```

    - **Modify userprog/syscall.c/syscall_init():** To implement the synchronization for avoiding the race condition, it is required to initialize *lock* as 1 and *w_or_r* as 1. Moreover, the initial *read_count* needs to be set as 0.

```
void
syscall_init (void)
{
 intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");
 /* Initialize the lock for the file system. */
 //lock_init(&lock);
 sema_init(&w_or_r,1);
 sema_init(&lock,1);
 read_count = 0;
}
```

- **Add userprog/syscall.c/syscall_create():** When *syscall_handler()* notifies the create system call, it would allocate to *syscall_create()* by passing the first argument as the name of the file and second argument as the initial size of the file. By using the *pagedir_get_page()* function, we could grab the file name and thus pass to *syscall_create()*. Inside the *syscall_create()*, if the file is *NULL* or its address is invalid, it is necessary to get terminated by calling *syscall_exit()*. Otherwise, it would return the boolean by invoking *filesys_create()*.

```
static void syscall_handler (struct intr_frame *f UNUSED)
{
//..
   switch(*(int *) f->esp)
   {
     case SYS_CREATE:
      /* It calls the syscall_create() to create a file which has the first
      argument as the name of the file and second argument as the size of the file.*/
       copy_arguments(f, &args[0], 2);
       phys_page_ptr = pagedir_get_page(thread_current()->pagedir, (const void *) args[0]);
       args[0] = phys_page_ptr;
       f->eax = syscall_create(args[0], (unsigned) args[1]);
       break;
   }
}

bool syscall_create (const char *file, unsigned initial_size)
{
  if (file == NULL)
    syscall_exit(-1);

  return filesys_create(file, initial_size);
}
```

- **Add userprog/syscall.c/syscall_remove():** When *syscall_handler()* notifies the remove system call, it would allocate to *syscall_remove()* by passing the first argument as the name of the file. If the file is *NULL* or its address is invalid, it is required to get terminated by calling *syscall_exit()*. Ortherwise, it would return boolean value by invoking *filesys_remove()*.

```
static void syscall_handler (struct intr_frame *f UNUSED)
{
```

```
//..
   switch(*(int *) f->esp)
   {
     case SYS_REMOVE:
       /* It calls the syscall_remove() to remove a file which name is a file
       that is the first argument passing. */
       copy_arguments(f, &args[0], 1);
       phys_page_ptr = pagedir_get_page(thread_current()->pagedir, (const void *) args[0]);
       args[0] = phys_page_ptr;

       f->eax = syscall_remove((const char *) args[0]);
       break;
     //..
   }
}
bool syscall_remove (const char *file)
{
  if (file == NULL){
    syscall_exit(-1);
  }
  return filesys_remove(file);
}
```

- **Add userprog/syscall.c/syscall_open():** When *syscall_handler()* notifies the open system call, it would allocate to *syscall_open()* by passing the first argument as the name of the file. This function is to open the file associated with path in file. First, by using *pagedir_get_page()* to get a file, if the file is *NULL*, it would invoke *syscall_exit()* directly. However, if it is not, the name of the file would be passed inside the *syscall_open()*. Inside the syscall_open(), we would call *filesys_open()* to store in the FDT with the struct file. By that, it is required to a structure to hold that file inside FDT by using *malloc()* and store the file address as its associated file. Then, increment *fd_count* to have the space for other file descriptor. Lastly, it is essential to push that file descriptor inside the *fd_table*. Therefore, it would return *fd* as the index of that file descriptor.

```
static void syscall_handler (struct intr_frame *f UNUSED)
{
//..
   switch(*(int *) f->esp)
```

```
  {
    case SYS_OPEN:
      /* It calls the syscall_open() to open file associated with the path in file and its argument passing is the name of
the file. */
      copy_arguments(f, &args[0], 1);
      phys_page_ptr = pagedir_get_page(thread_current()->pagedir, (const void *) args[0]);
      if (phys_page_ptr == NULL)
        syscall_exit(-1);

      args[0] = phys_page_ptr;
      f->eax = syscall_open(args[0]);
      break;
    //..
  }
}

/* This function is to open a file with the given name, and returns the fd
  opened by one thread. */
int syscall_open (const char *file)
{
  struct file* f = filesys_open(file);
  struct thread *t = thread_current();

  /* If there is no file created, return -1. */
  if(f == NULL)
    return -1;

  /* The implementation is to create a structure to hold the file/fd,
     for the use of fd_table of the current process. */
  struct thread_file *new_file = malloc(sizeof(struct thread_file));
  new_file->file_addr = f;
  // Increment the fd for the next file
  int fd = t->fd_count;
  t->fd_count++;

  new_file->fd_index = fd;
  list_push_front(&thread_current ()->fd_table, &new_file->file_elem);
```

```
  return fd;
}
```

- **Add userprog/syscall.c/syscall_filesize():** When *syscall_handler()* notifies the filesize system call, it would allocate to *syscall_filesize()* by passing the name of that file. This function is to return the file size. By performing it, it would check whether there is a file inside the FDT of that thread. If not, it would return -1. Otherwise, iterate through the list *fd_table* with the *list_elem*. If the *fd_index* of that file is the same as given *fd*, it would invoke *file_length()* function to return in size of the opening file. However, if there is no file found in FDT, there would return -1.

```
static void syscall_handler (struct intr_frame *f UNUSED)
{
//..
    switch(*(int *) f->esp)
    {
      case SYS_FILESIZE:
        /* It calls the syscall_filesize() to return the file size. It would receive the fd of the file as its argument */
        copy_arguments(f, &args[0], 1);
        f->eax = syscall_filesize(args[0]);
        break;
      //..
    }
}


int syscall_filesize (int fd)
{
  /* Element to iterate through the fd_table. */
  struct list_elem *e;
  /* If there are no files in this thread, return -1 */
  if (list_empty(&thread_current()->fd_table))
    return -1;

  /* If the given fd is opened and used by the current process, return the length of the file*/
  for (e = list_front(&thread_current()->fd_table); e != NULL; e = e->next)
  {
     struct thread_file *t = list_entry (e, struct thread_file, file_elem);
```

```
    if (t->fd_index == fd)
      return file_length(t->file_addr);
  }
  return -1; // No file is found is needed to return -1
}
```

- **Add userprog/syscall.c/check_buffer():** This function is added in order to check whether each memory access to buffer in a valid user space or not. The newly created function would be used for *syscall_read()* and *syscall_write()* to read or write the buffer content. Indeed, this function would receive the buffer and its size in order to ensure the address of a buffer resulted with a validity.

```
/* Project 2: System Call for file-level. This function is to check whether each memory address of a buffer in valid user space or not. It would be used to check the given buffer when calling the syscall_read() and syscall_write(). */
void check_buffer (void *buffer, unsigned size)
{
  unsigned i;
  char *ptr  = (char * )buffer;
  for (i = 0; i < size; i++)
    {
      address_validation(ptr);
      ptr++;
    }
}
```

- **Add userprog/syscall.c/syscall_read():** When *syscall_handler()* notifies the read system call, it would allocate to *syscall_read()* by passing *fd* (index), the *buffer*, and its *length*. Before coming to execute the *syscall_read()* with given length, it is important to check whether its buffer valid or not by *check_buffer()* and also its converted physical page pointer. If that pointer is *NULL*, the process should be terminated. Otherwise, it invokes *syscall_read()*. Importantly, there are two cases to aware inside that system calls, which are *fd = 0* and *fd = 1*. If *fd = 0*, it should be read from the keyboard input by using function *input_getc()*. On the other hand, if *fd =1* or the there are no file descriptor inside FDT, it should be returned. With the remain number of fd, we could perform to iterate through list *fd_table* and read the associate *fd* (index), thus returning the number of reading bytes by using *file_read()*. However, if there is no file read, return -1.
  To enable the synchronization among other readers, it is necessary to implement the idea of reader-writer buffer as described in the lecture. Indeed, one reader grabs the lock once at a time and perform the reading content. If the first reader gets inside the *lock*, it should wait for *w_or_r* if the writer is writing on the buffer. If not, it could grab its turn to read the buffer content and release the l*ock* for other

readers get its turn to read. While all readers read the buffer, the last reader should invoke the writer to get ready for its turn to write the buffer content.

```
static void syscall_handler (struct intr_frame *f UNUSED)
{
//..
    switch(*(int *) f->esp)
    {
      case SYS_READ:
       /* It calls the syscall_read() to read the buffer with given length.
       The passing argument would be fd (index), the buffer, and the buffer length. */
        copy_arguments(f, &args[0], 3);

        /* Ensure the valid buffer is passing*/
        check_buffer(args[1], args[2]);

        phys_page_ptr = pagedir_get_page(thread_current()->pagedir, (const void *) args[1]);
        if (phys_page_ptr == NULL)
          syscall_exit(-1);
        args[1] = phys_page_ptr;
        f->eax = syscall_read(args[0], (void *) args[1], (unsigned) args[2]);
        break;
      //..
    }
}
/* This system call is to read with the given length from the file open as fd into its buffer.
   Therefore, it would return the number of bytes actually read
  - If fd = 0, it should be read from keyboard by using function input_getc()
  - If fd = 1, the file could not be read . */
int syscall_read (int fd, void *buffer, unsigned length)
{
  /* Element to iterate through fd_table. */
  struct list_elem *e;
  /* fd = 0 */
  if (fd == 0)
    return input_getc();
```

```
  /* fd = 1 or the fd_table is empty. */
  if (fd == 1 || list_empty(&thread_current()->fd_table))
    return 0;

  /* If fd is in fd_table, we could perform to read from the file and
  return the number of bytes_read. */
  for (e = list_front(&thread_current()->fd_table); e != NULL; e = e->next)
  {
     struct thread_file *t = list_entry (e, struct thread_file, file_elem);
     if (t->fd_index == fd)
     {
       sema_down(&lock);
       read_count++;
       if (read_count == 1){
         sema_down(&w_or_r);
       }
       sema_up(&lock);
       int bytes_read = file_read(t->file_addr, buffer, length);
       sema_down(&lock);
       read_count--;
       if (read_count == 0){
         sema_up(&w_or_r);
       }
       sema_up(&lock);
       return bytes_read;
     }
  }
  return -1; //Not read the file, return -1
}
```

- **Add userprog/syscall.c/syscall_write():** When *syscall_handler()* notifies the write system call, it would allocate to *syscall_write()* by passing *fd* (index), the *buffer*, and its *length*. The implementation is the same as *syscall_read()* in terms of check the valid buffer or the converted physical page pointer. Inside that system call, there are two cases to be aware of. If *fd = 1*, we could write to STDOUT by using *putbuf()*. On the other hand, if *fd = 0* or there is no file in the *fd_table*, it needs to invoke *syscall_exit()*. With the remain *fd* index, we could perform to iterate through the list *fd_table*, and write on the buffer with the associated *fd*, thus return the number of writing bytes by using *file_write()*.

To enable the synchronization, the writer would wait for *w_or_r* and release it while it finishes writing on the buffer.

```
static void syscall_handler (struct intr_frame *f UNUSED)
{
//..
    switch(*(int *) f->esp)
    {
   case SYS_WRITE:
       /* It calls the syscall_write() to write the buffer with given length.
       The passing argument would be fd (index), the buffer, and the buffer length. */
        copy_arguments(f, &args[0], 3);
        /* Ensure the valid buffer is passing */
        check_buffer(args[1], args[2]);
        phys_page_ptr = pagedir_get_page(thread_current()->pagedir, (const void *) args[1]);
        if (phys_page_ptr == NULL)
          syscall_exit(-1);

        args[1] = phys_page_ptr;
        f->eax = syscall_write(args[0], (const void *) args[1], (unsigned) args[2]);
        break;
      //..
    }
}
/* This system call is to write with given length bytes from buffer to the open file with its fd.
  This function returns the number of bytes actually written
- If fd = 1, it is needed to use putbuf.
- If fd = 0, it should be terminated. */
int syscall_write (int fd, const void *buffer, unsigned length)
{
  /* Element to iterate through fd_table */
  struct list_elem *e;

  /* If fd = 1, then we write to STDOUT. */
  if(fd == 1)
  {
    putbuf(buffer, length);
```

```
    return length;
  }
  /* If fd = 0 (STDIN) or there is no file in fd_table, return 0. */
  if (fd == 0 || list_empty(&thread_current()->fd_table))
    syscall_exit(-1);

  /* If the given id is opened and used by the current process, return
     the number of bytes_written to the file. */
  for (e = list_front(&thread_current()->fd_table); e != NULL; e = e->next)
  {
      struct thread_file *t = list_entry (e, struct thread_file, file_elem);
      if (t->fd_index == fd)
      {
        sema_down(&w_or_r);
        int bytes_written = file_write(t->file_addr, buffer, length);
        sema_up(&w_or_r);
        return bytes_written;
      }
  }
  return -1; //No file is written, return 0
}
```

- **Add userprog/syscall.c/syscall_seek():** When *syscall_handler()* notifies the seek system call, it would allocate to *syscall_seek()* by passing *fd* (index), and position. The main functionality is to change the target postion for reading or writing with the associated *fd*. If there is no file descriptor in *fd_table*, it would be returned. Otherwise, it would iterate through the list *fd_table* and look for the associated *fd_index* with that *thread_file*. Therefore, it would perform *file_seek()* and return afterwards. If there is no file descriptor found inside *fd_table*, it would return outside.

```
static void syscall_handler (struct intr_frame *f UNUSED)
{
//..
    switch(*(int *) f->esp)
    {
      case SYS_SEEK:
        /* It calls the syscall_seek() to change the target position for reading or writing
        the open file with fd */
```

```
      copy_arguments(f, &args[0], 2);
      syscall_seek(args[0], (unsigned) args[1]);
      break;
    //..
  }
}
void syscall_seek (int fd, unsigned position)
{
 /* Element to iterate the fd_table. */
 struct list_elem *e;
 /* If there are no files, then return. */
 if (list_empty(&thread_current()->fd_table))
  return;

 /* If the given fd is in fd_table, we could seek toward that file. */
 for (e = list_front(&thread_current()->fd_table); e != NULL; e = e->next)
 {
   struct thread_file *t = list_entry (e, struct thread_file, file_elem);
   if (t->fd_index == fd)
   {
     file_seek(t->file_addr, position);
     return;
   }
 }
 return; //No file could be seeked,return
}
```

- **Add userprog/syscall.c/syscall_tell():** When syscall_handler() notiies the tell system call, it would allocate to *syscall_tell()* by passing *fd* (index). The main functionality is to return the position of the open file to be read or written on. The same implementation is as above function with checking the existence of file_descriptor in *fd_table*. If so, it is required to iterate through the list *fd_table* and find for the associated *fd_index*. Therefore, it would perform *file_tell()* which returns the postion of that file. If there is no file descriptor inside found inside *fd_table*, it would return outside.

```
static void syscall_handler (struct intr_frame *f UNUSED)
{
//..
```

```
    switch(*(int *) f->esp)
    {
      case SYS_TELL:
        /* It calls syscall_tell to return the position of the open file to be read or written with its passing argument would
be fd.*/
        copy_arguments(f, &args[0], 1);
        f->eax = syscall_tell(args[0]);
        break;
      //..
    }
}
unsigned syscall_tell (int fd)
{
  /* Element to iterate through fd_table. */
  struct list_elem *e;
  /* If there are no files in fd_table, return immediately, */
  if (list_empty(&thread_current()->fd_table))
    return -1;

  /* If the given id is in the fd_table, call file_tell() and return the position. */
  for (e = list_front(&thread_current()->fd_table); e != NULL; e = e->next)
  {
     struct thread_file *t = list_entry (e, struct thread_file, file_elem);
     if (t->fd_index == fd)
     {
       unsigned pos = file_tell(t->file_addr);
       return pos;
     }
  }
  return -1;
}
```

- **Add userprog/syscall.c/syscall_close():** When *syscall_handler()* notifies the close system call, it would allocate to *syscall_close()* by passing *fd* (index). The main functionality is to close the open file with the associated fd. The same implementation is as above function with checking the existence of file descriptor in *fd_table*. If so, it is required to iterate through the list *fd_table* and find for the associated *fd_index*. Therefore, it would perform *file_close()* which close that open file. In addition, it is necessary to use *list_remove()* for removing the file

element out of the *thread_file* and thus return afterwards. If there is no file descriptor inside found inside *fd_table*, it would return outside.

```
static void syscall_handler (struct intr_frame *f UNUSED)
{
//..
   switch(*(int *) f->esp)
   {
     case SYS_CLOSE:
       /* It calls syscall_close() to close the file with the index fd  */
       copy_arguments(f, &args[0], 1);
       syscall_close(args[0]);
       break;
     //..
   }
}
void syscall_close (int fd)
{
  /* Element to iterate through fd_table. */
  struct list_elem *e;
  /* If there are no files in fd_table, return */
  if (list_empty(&thread_current()->fd_table))
    return;

 /* If the given fd is in fd_table, close the file and remove it from its fd_table. */
  for (e = list_front(&thread_current()->fd_table); e != NULL; e = e->next)
  {
     struct thread_file *t = list_entry (e, struct thread_file, file_elem);
     if (t->fd_index == fd)
     {
       file_close(t->file_addr);
       list_remove(&t->file_elem);
       return;
     }
  }
  return;
```

```
}
```

III. Denying Writes to Executables

- Data Structure: no new variable or data structure is added
- Algorithm
    - **Modify userprog/process.c/load():** When loading is successful, it is essential to deny writing on the file. Therefore, we need to provoke *file_deny_write()* to reject the writing on the file.

```
bool
load (const char *file_name, void (**eip) (void), void **esp)
{
//..
/* Project 2: Denying writes to executable files - When the load is successful, it is
  essential to deny the writting on the file. If there exists some unpredictable behaviors,
  close the file. */
   if (success)
     file_deny_write(file);
   else
     file_close (file);
   return success; /* In either case, return whether or not the executable was loaded correctly.  */
}
```

    - However, to provoke *file_allow_write()* in the *process_exit()*, there is no need to perform that action since my method has dealt with this particular case in system call and synchronization.

IV. Discussion

From the project 2, I have acquired much knowledge regarding the process execution and system call mechanism. When it comes to the big picture of process procedure, it would run through bunch of the functions (i.e. *run_action(), run_task(), process_wait(), process_execute()*) to set up and initialize the thread before its creation. If that thread is getting scheduled, it is required to start a process and load from disk to memory for starting to execute user program mode. By the naïve PINTOS implementation, it is essential to modify an initial stage, which is parsing the file name to let it get ready for thread creation. On the other hand, system calls are the most important part for this project since the program should be switched from user mode to kernel mode for having more privilege of manipulating the user process or its file system. By calling interrupt inside the user program, it would send the signal towards *syscall_handlers()* afterwards. Indeed, under the scope of OS management, this handler would notice which system call is invoked and thus allocating to the newly created function in the project for each *syscall* case. However, there is one thing to consider which is memory leakage when the process is at the end of its procedure or page fault is occurred during system call handling. Therefore, it is vital to modify several part of *page_fault()* in order to prevent those edgy scenarios.