Operating Systems – Project 2: User Program

Design Report

Khau Lien Kiet

Team 10

2022/10/14

Department of Computer Science & Engineering

Pohang University of Science and Technology (POSTECH)

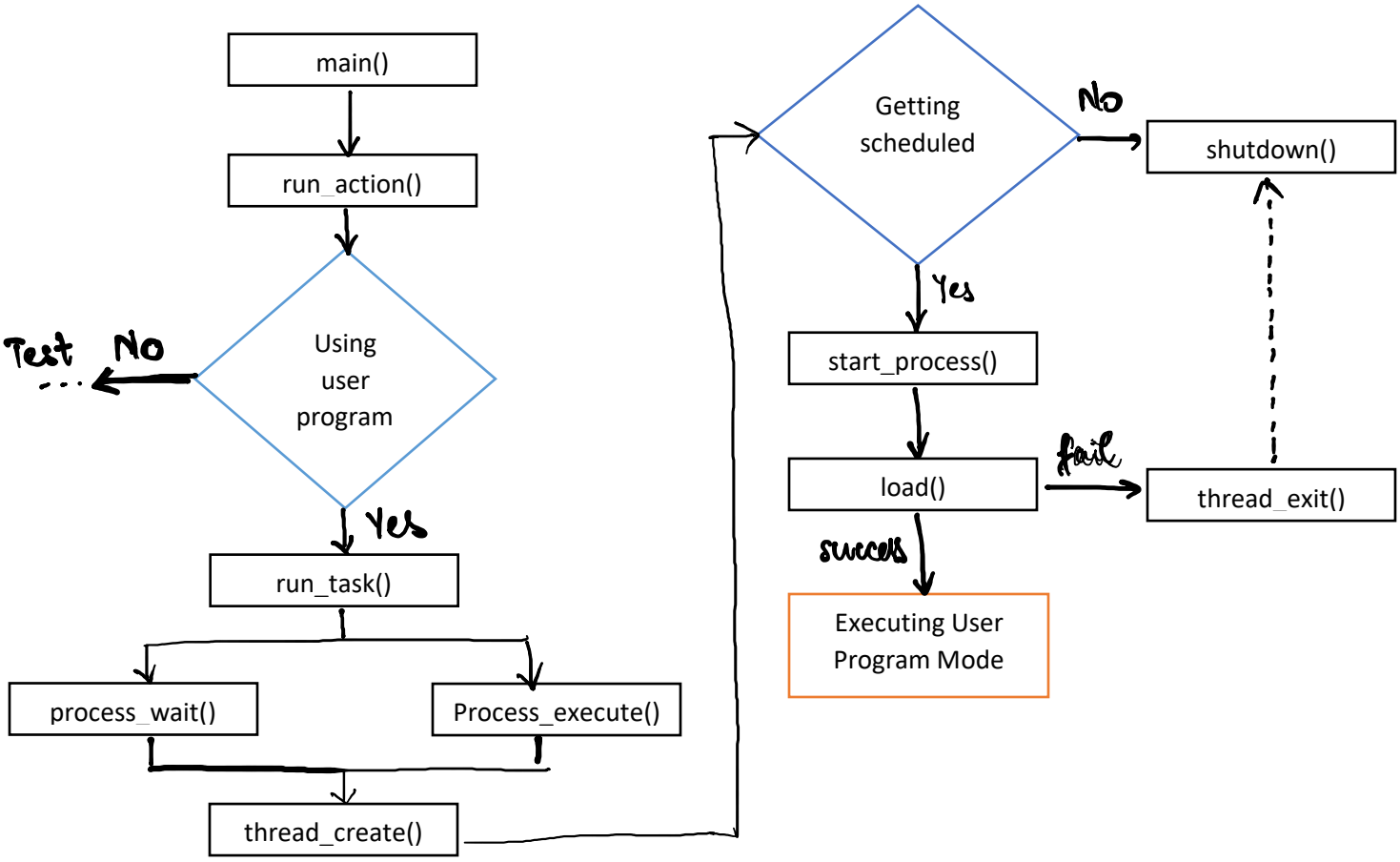
Table of Contents

I. Process Execution

a. Analysis of current system

- Data Structure's background
  - **threads/interrupt.h/struct intr_frame:** interrupt stack frame is a data structure that Kernel stores the user's registers to the Kernel stack. To switch from the Kernel mode (Operating systems) to the User mode, it is necessary to call *intr_exit* (interrupt exit). On the other hand, if you want to switch from User mode to Kernel mode, it is essential to call *int* (interrupt). ***(explained below)***
  - **argv:** starting address of the argument
  - **argc:** number of the argument in the command line
  - **if_.eip:** starting entry point of the function after executing *load()* function
  - **if_.esp:** stack top pointer on the user stack, which is needed to execute
- Procedure (explained by functions and algorithms)

Big picture:



- Pre-execution:
  - **threads/init.c/run_actions(char **argv):** The function would receive the given starting address of the argument, which is being parsed. Within the array, it would iterate to maintain each action in that command line. During the process of *run_actions()*, it could check whether the action is performed valid or not. Therefore, if valid, function is executed with the actual task and simultaneously the user program would run.

```
static void
run_actions (char **argv)
{
  /* An action. */
  struct action
    {
      char *name;                       /* Action name. */
      int argc;                         /* # of args, including action name. */
      void (*function) (char **argv);   /* Function to execute action. */
    };

  /* Table of supported actions. */
  static const struct action actions[] =
    {
      {"run", 2, run_task},
#ifdef FILESYS
#endif
      {NULL, 0, NULL},
    };
  while (*argv != NULL)
    {
      const struct action *a;
      int i;
      /* Find action name. */
      for (a = actions; ; a++)
        if (a->name == NULL)
          PANIC ("unknown action `%s' (use -h for help)", *argv);
        else if (!strcmp (*argv, a->name))
          break;
      /* Check for required arguments. */
      for (i = 1; i < a->argc; i++)
        if (argv[i] == NULL)
          PANIC ("action `%s' requires %d argument(s)", *argv, a->argc - 1);

      /* Invoke action and advance. */
      a->function (argv);
      argv += a->argc;
```

- **threads/init.c/run_task(char **argv):** Under the user program, it would be executed under the task-level. Hence, while receiving the argument in the process, this function is called *process_execute()* ***(detailed explanation below)*** and *process_wait()* – which is waiting for a child process to be finished executing. However, regarding the *process_wait()* in the naïve PINTOS, it would not perform any action instead of simply returning -1 to wait for a thread get terminated and return a termination status.

```
/* Runs the task specified in ARGV[1]. */
static void
run_task (char **argv)
{
  const char *task = argv[1];

  printf ("Executing '%s':\n", task);
#ifdef USERPROG
  process_wait (process_execute (task));
#else
  run_test (task);
#endif
  printf ("Execution of '%s' complete.\n", task);
}
```

```
int
process_wait (tid_t child_tid UNUSED)
{
  return -1;
}
```

- Main execution:
  - **userprog/process.c/process_execute(const char *file_name):** To contribute to the procedure of process execution, this function would receive a name of executable file. Also, by calling function *thread_create()* ***(detailed explanation below),*** function would create a new thread. At the moment of new thread is created, it would continue executing. Under the code-level interpretation, it would

load the given file name to the user program and execute a new thread. To prevent the race condition, it is required to copy the *file_name* and thus the thread creation request is shown up. Therefore, it would pass the *file_name* to the *thread_create()* to create a thread for getting executed. Otherwise, it would return the error to indicate the failure of creating a new thread. Particularly, in the *thread_create()*'s passing argument, there would be a function argument *start_process()* ***(detailed explanation below)*** to run the program.

```
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(). */
  //...

  /* Create a new thread to execute FILE_NAME. */
  tid = thread_create (file_name, PRI_DEFAULT, start_process, fn_copy);
  //...
  return tid;
}
```

- **threads/thread.c/thread_create(…):** According to PINTOS project 1, this function has been used to implement toward the thread-level system. However, when it comes to the current project, we would utilize it on a different purpose related to the user program execution. Particularly, it would create the new thread structure toward the *struct thread* and thus initialize it. Then, it would not only allocate to the kernel stack but also receive the *start_process()* ***(detailed description below)*** as function argument to put the value of instruction counter to the entry point. Finally, it is required to be added to the *ready_list* by *thread_unblock()*.

```
tid_t
thread_create (const char *name, int priority,
               thread_func *function, void *aux)
{
  struct thread *t;
  struct kernel_thread_frame *kf;
  struct switch_entry_frame *ef;
  struct switch_threads_frame *sf;
  tid_t tid;

  ASSERT (function != NULL);

  /* Allocate thread. */
  t = palloc_get_page (PAL_ZERO);
  if (t == NULL)
    return TID_ERROR;

  /* Initialize thread. */
  init_thread (t, name, priority);
  tid = t->tid = allocate_tid ();

  /* Stack frame for kernel_thread(). */
  kf = alloc_frame (t, sizeof *kf);
  kf->eip = NULL;
  kf->function = function;
  kf->aux = aux;

  //...

  /* Add to run queue. */
  thread_unblock (t);

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

From the code section, *thread_create()* would receive **name* (function name needed to execute), priority, **function* (function name), and auxiliary parameter. In terms of its mechanism, when the function *thread_create()* is called, it would allocate the single page which size of 4K in the Kernel space. Indeed, under the address space, it would initialize the thread structure afterwards and *tid (thread ID)* would be allocated since each thread has its unique ID. When it comes to the stack frame for the Kernel thread, some parameters could be initialized such as *eip*, function to run, or parameters for that function. Lastly, Kernel is putting the thread to the *ready_list* by *thread_unblock()*.

- **userprog/process.c/start_process(void *file_name):** While the thread is created and put into the *ready_list*, it is the time to start a process. From the code section, beside initializing the interrupt frame, there would be a *load()* function ***(detailed explanation below)***. Indeed, its main functionality is to load the program with a *file_name* (binary file) from disk to memory and thus setting an initial pointer. Looking further inside the *load()* function's argument, *load()* function needs to receive the location/entry point of instruction needed to execute as well as the stack top pointer of the user stack. Indeed, if the program performs *load()* function successfully, it would start the user process by jumping to the user program by *interrupt_exit()*. Otherwise, it would call *thread_exit()* to clean up the memory space which is allocated for starting a process before then and thus let the thread quit.

```
static void
start_process (void *file_name_)
{
  char *file_name = file_name_;
  struct intr_frame if_;
  bool success;

  /* Initialize interrupt frame and load executable. */
  //...
  success = load (file_name, &if_.eip, &if_.esp);

  /* If load failed, quit. */
  palloc_free_page (file_name);
  if (!success)
    thread_exit ();

  /* Start the user process by simulating a return from an
     interrupt, implemented by intr_exit (in
     threads/intr-stubs.S).  Because intr_exit takes all of its
     arguments on the stack in the form of a `struct intr_frame',
     we just point the stack pointer (%esp) to our stack frame
     and jump to it. */
  asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) : "memory");
  NOT_REACHED ();
}
```

- **userprog/process.c/load(…):** Generally, when naïve PINTOS runs *load()*, it would read the binary file from disk to memory as well as initialize some segments in the physical memory (data segment, text segment, stack segment) for a process to run.
  Under the visualization of computer organization through *load()* function, when it is called, Kernel would create a page table for the given threads and then open the file to read the *ELF header* (contained the information how the file is organized). Firstly, store the *ELF header* into the memory. Next, with the usage of the *ELF header*, *load()* could parse the file, load the data section into the data segment in memory, and load the text section into the text segment in memory. Then, it would create a stack for a process and initialize it. Indeed, inside the *struct thread*, it did contain the pointer to the page table, which consists of *pagedir* to point toward the data structure of data, text, stack field.

```
bool
load (const char *file_name, void (**eip) (void), void **esp)
{
  struct thread *t = thread_current ();
  struct Elf32_Ehdr ehdr;
  struct file *file = NULL;
  off_t file_ofs;
  bool success = false;
  int i;

  /* Allocate and activate page directory. */
  t->pagedir = pagedir_create ();
  if (t->pagedir == NULL)
    goto done;
  process_activate ();
  //...
  /* Read and verify executable header. */
  if (file_read (file, &ehdr, sizeof ehdr) != sizeof ehdr
      || memcmp (ehdr.e_ident, "\177ELF\1\1\1", 7)
      || ehdr.e_type != 2
      || ehdr.e_machine != 3
      || ehdr.e_version != 1
      || ehdr.e_phentsize != sizeof (struct Elf32_Phdr)
      || ehdr.e_phnum > 1024)
    {
      printf ("load: %s: error loading executable\n", file_name);
      goto done;
    }
  //...
  /* Set up stack. */
  if (!setup_stack (esp))
    goto done;
  /* Start address. */
  *eip = (void (*) (void)) ehdr.e_entry;

  success = true;
```

- *Problem statement & Brief solution:* After creating a user process, it cannot run the process in the user program since it always exits out of the program. In the naïve PINTOS, it does not have mechanism to tokenize the command line argument. Indeed, it just passes entire command line for process execution, which leads to the not working. By the project description, we must implement process termination process, and argument passing. Within it, in terms of the process termination process, when the user process is terminated, it is necessary to print the name of the process and the exit code. Afterwards, regarding the argument passing, generally, we just parse the arguments within some functions created and push them to the user stack. Therefore, it is necessary to initialize/set up the stack in the user stack with the proper parameters before jumping to the executed function. If there would no user stack setting in the implementation, there is no destination to restore the user process's register from the interrupt frame *(intr_frame),* which is resided in the Kernel stack.

b. Solution

i. Implementation of process termination process:

When the user process finishes it executing, it is required to prompt out the name of the process and exit code. Importantly, when the Kernel thread is terminated or halt system call is called, it is not allowed to print the above message.

- **Add userprog/syscall.c/syscall_exit(status)**: To implement it, create a new function in *system_exit()* to perform the prompting termination message included the name of the process and the exit code. In addition, to avoid the above edge cases, *thread_exit()* should be called below the *printf()* function.

ii. Implementation of argument passing

From the current naïve PINTOS, aforementioned, it takes the entire command line as the program file name. Thus, our main implementation is to parse the command line into words to get the program name toward process_execute() and also pass the number of the arguments in the command line toward the

start_process() and initialize the stack. Therefore, we would have various modifications and additions toward the naïve PINTOS:

- **Add pars_file_name():** This function would result in the *file_name* that would input through the *process_execute(),* particularly passing into the *thread_create()*'s function argument.
- **Modify userprog/process.c/process_execute():** Add the newly created function to parse the command line and thus receive the program name.

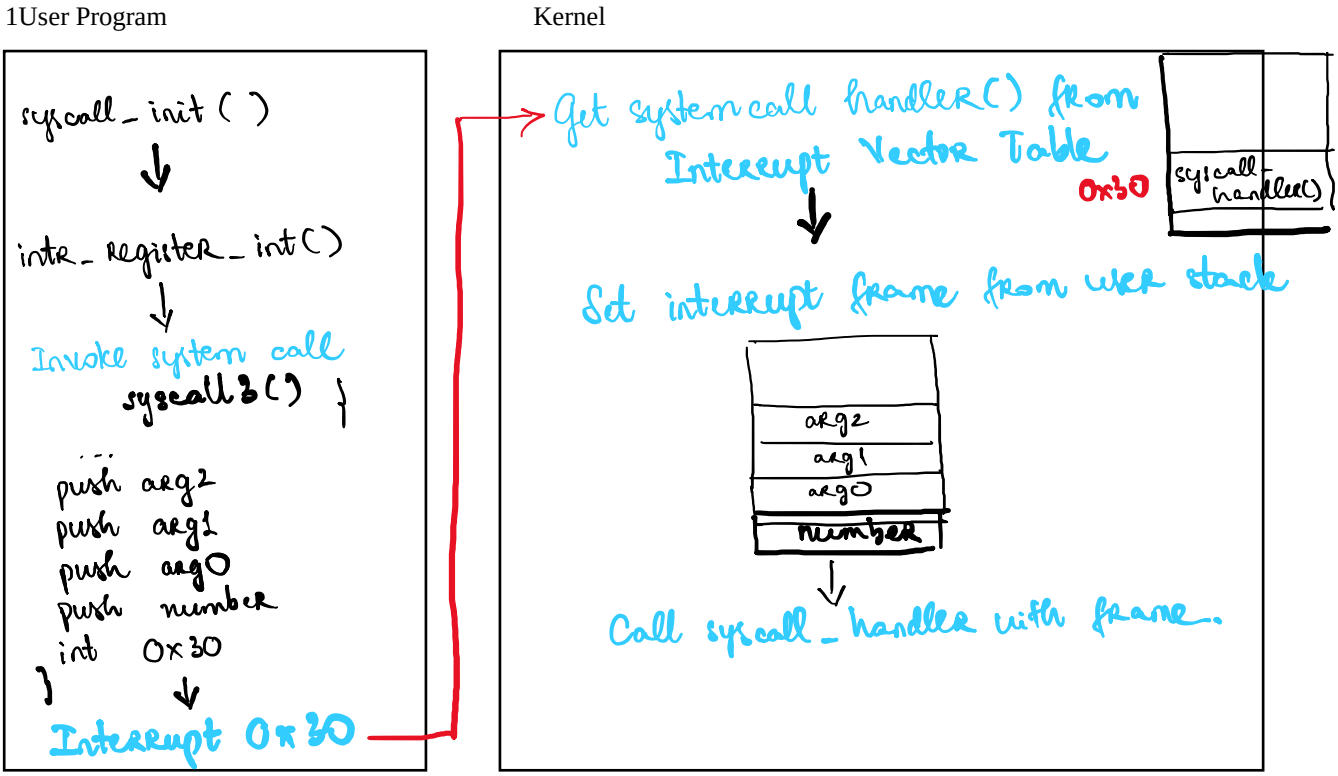
- **Add pars_argument():** This function would return in the number of arguments in the command line
- **Add init_user_stack():** With the current PINTOS, there is no mechanism to provide the initialization of user stack with an argument before switching to the user program and executing the function. Therefore, it is the reason to add this function and input toward the *start_process()*. To implement it, we should move down the *esp* by decrement by certain integer numbers to perform some operation. Indeed, inside the user stack, it is essential to push arguments (push the parsing strings from the right to left in the command line, align the padding by 4 bytes, push the address of the parsing strings), push *argc* (starting address of the first parsing strings) and *argv* (number of arguments in the command line), and push the return address which is contained the next instruction.
- **Modify userprog/process.c/start_process():** Inside this function, we would implement the *pars_arguments()* and *init_user_stack()* if the *load()* function is performed successfully.

II. System Call

a. Analysis of current system

- Description: Before implementing the system call toward the user program in the naïve PINTOS, it is required to explain what the system call is. According to the lecture, system call is a programming interface for some services (i.e. create a process, access the memory, or file, etc.) provided by operating systems. In addition, it would allow the user program to use some certain resources protected by the kernel, which has more privileged than the user mode. By the system calls, they could run its task on the kernel mode through *int* (interrupt) and thus return to the user mode if the system call finishes executing. Therefore, when the system call is called, the execution mode for that thread obtains a higher priority since it is switched toward the special mode – Kernel privilege.
- Procedure (explained by functions and argument)

Big picture



- **Initialization:** When it comes to the main section, **userprog/syscall.c/syscall_init()** is made to initialize the system calls, which is used for the user program. By then, **threads/interrupt.c/intr_register_int()** is called to interrupt for indicating that it is the time to switch from the user program to the kernel mode since we are now generating the *int* – interrupt that allows the transformation from the user mode to the kernel mode. Looking inside the passing argument from the function *intr_register_int()*, it is passed *the register internal interrupt - 0x30* (the interrupt number represented as system call), thus afterwards invoking the *userprog/syscall.c/syscall_handler()* ***(detailed explanation below)*** inside the Kernel. Therefore, it succeeds to initialize the switching between user to kernel option.

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

```
void
intr_register_int (uint8_t vec_no, int dpl, enum intr_level level,
                   intr_handler_func *handler, const char *name)
{
  ASSERT (vec_no < 0x20 || vec_no > 0x2f);
  register_handler (vec_no, dpl, level, handler, name);
}
```

- **System call invoke:** Before entering the kernel area, some *syscall* functions would be executed based on the number of arguments. When it comes to this case, ***lib/user/syscall.c/syscall3()*** is executed. Hence, it would push the three arguments as well as the number of the *syscall argument number* in the user stack. Under the computer organization understand, while generating the interrupt to get into the OS level, there would exist a pointer pointing toward the *Interrupt Vector Table*, where is contained the *syscall_handler()*. Indeed, the address of the pointer pointing is the same as the *syscall_handler()*, which is *0x30*. Therefore, it is officially getting invoked from the interrupt from the user mode toward the kernel one.

```
/* Invokes syscall NUMBER, passing arguments ARG0, ARG1, and
   ARG2, and returns the return value as an `int'. */
#define syscall3(NUMBER, ARG0, ARG1, ARG2)                \
        ({                                                \
          int retval;                                     \
          asm volatile                                    \
            ("pushl %[arg2]; pushl %[arg1]; pushl %[arg0]; "  \
             "pushl %[number]; int $0x30; addl $16, %%esp"    \
             : "=a" (retval)                              \
             : [number] "i" (NUMBER),                     \
               [arg0] "r" (ARG0),                         \
               [arg1] "r" (ARG1),                         \
               [arg2] "r" (ARG2)                          \
             : "memory");                                 \
          retval;                                         \
        })
```

- **System call handler:** Aforementioned, system call handler is executed whenever there is a notice of the interrupt. In addition, it is taking charge of handling the system call requested from the user program. However, there is no further implementation for ***userprog/syscall.c/syscall_handler()*** in the naïve PINTOS.

```
static void
syscall_handler (struct intr_frame *f UNUSED)
{
  printf ("system call!\n");
  thread_exit ();
}
```

- *Problem Statement & Brief Solution:* Obviously, there is a current gap in the current PINTOS system as there is not any implementation inside the *syscall_handler()* function. Therefore, to implement it, there are some requirements to integrate into our new implementation ***(explained in II.b).*** For a further solution, it is necessary to consider the system calls related to the process as well as the file ***(explained in III)*** that system call handler would tackle with.

b. Solution for implementation of system call

i. Implementation for system call handler

To implement the system call handler, it is necessary to synthesize the fundamental requirements since accessing into the kernel mode is needed to be meticulous if we do not want any error inside the OS. Indeed, it is difficult to notice the error code when executed in OS. Therefore, there are some following requirements to initialize:

(1) Integrate the system call number ***(src/lib/syscall-nr.h)*** *while making the system call handler call the system call*

(2) Verify the validation of address supplied by the user program or pointer whether available or not:

+ Pointer points to the user area, not kernel area (supplied by user to system call)

+ If pointer is not pointed to the valid address, it occurs the page fault.

(3) Copy the arguments from the user stack to the kernel

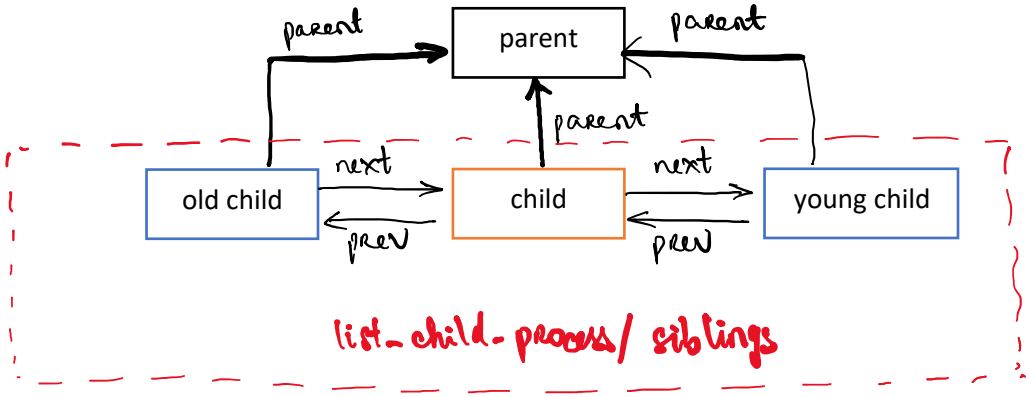
(4) Save the return value of system call in the eax register (stored in the int_frame)

- **Add address_validation():** This function returns the boolean value, which indicates that 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 that "wrong" process terminate. For a further implementation, we could check the user pointer below a variable *PHYS_BASE*. Otherwise, it would be invalid and cause the *page_fault*.
- **Add copy_argument():** This function would divide the argument from the user stack and add them in array. The main functionality is to get the argument from the user stack to the kernel. Also, we need to check the condition of a valid address by checking that the pointer does point to the user area. If it points to the kernel space, it is required to be terminated and exit.
- **Modify userprog/exception.c/page_fault():** Considering the occurrence of page fault, it would lead to that the process dies but it is still held the lock or process gets *malloc()* but it does not free. Hence, the resource would be leaked. To avoid these edge case, it is required to modify the *page_fault().*
- **Modify userprog/syscall.c/system_call_handler():** When adding a new created function *address_validation()* and *copy_argument(),* it is essential to integrate all the above requirements in each *system call number* that system call handler calls toward that particular system call. Inside each case, we could verify the address by *address_validation()* as well as get the argument toward the kernel to proceed. At last, it is required to save the return value of that specific system call ***(detailed implementation below)*** into the *eax* register. For a further consideration, we would implement four system calls associated with the process in the following part.

ii. Implementation of system call related to the process-level: ***halt(), exit(), exec(), wait()***

Before going to the main implementation of those above system call, it is necessary to go through the brief idea of those functions: ***(i) halt():*** *shut down the PINTOS,* ***(ii) exit():*** *Exit the process and print the message " Name of process: exit(status)",* ***(iii) exec():*** *Create a child process that wait till its child process finished creating and loading executable and execute the program associated with the cmd_line* ***(iv) wait():*** *Wait for the child process terminated and retrieve the child's exit status.*

- *Parent-Child relationship*: During the implementation for system call of *wait()* or *exec(),* there is noticeably shown up the correlation between the parent process and the child process. In terms of the *exec(),* a child process is created by the parent process. In addition, during the *wait()* stage, the parent process has to wait for their child process get terminated. Hence, we should provide a data structure to initialize those assignment between the process hierarchy. Therefore, we should initialize that relationship under struct thread
    - **Add struct thread *parent_process:** Pointer to parent process
    - **Add struct list list_child_process:** Pointer to the list of other child process (consider as the "sibling" thread)
    - **Add struct list_elem elem_child_process:** Pointer to the children processes



- **Add sys_halt(void):** In terms of the *sys_halt(),* we could use *shutdown_power(void*) to shutdown PINTOS
- **Add sys_wait(pid):**

  The main functionality of the system call wait() is to wait for a child process pid to finish executed or exit and receive its child exit status. If the pid is available which the child process is still executed, the parent should wait till it gets terminated and thus return the exit status of the terminated child process (passing pid to exit). Otherwise, if the pid is not unavailable (not called exit) which means that thread was terminated by the kernel, it is required to return -1. In addition, when the child process gets terminated, the parent process should deallocate the process descriptor. On the other hand, if the pid does not point to the direct child of the calling process (parent just waits for their own child) or the process calling wait has already waited again, wait() would fail and thus return -1.

  In the naïve PINTOS, *process_wait()* is not implemented indeed. Since *sys_wait()* is the system call *wait()* for the process, it is essential to implement and modify *process_wait().*

  - **Modify/reimplement process_wait():**
    - Using *child_tid* to search for the descriptor of the child process
    - The parent process is blocked until the child process exits. Whenever it exits, the parent should deallocate the descriptor of child process and return the exit status of the child process
    - Under the thread structure, there should contain a boolean field to be denoted as the *exit status*.
    - To synchronize the caller of *process_wait()* to wait for its child termination and the execution of the child process, it is necessary to implement the semaphore to the structure of thread, which is *sema_wait*. At the start, sema value should be initialize to 0 when creating a thread in order to indicate that the child would have its turn to get executed. Also, in the *process_wait(),* it would call *sema_down* for its semaphore *tid* in order to let the process - waiting for this semaphore - need to get block. When it comes to *sema_up()* for unblocking the waiting for process to executed, it might be called in the *process_exit()* function.

- **Add sys_exec(cmd_line):**
  The main functionality of system call exec() in PINTOS is the combination of fork() and exec(). Indeed, besides executing the program by passing the cmd_line, it would create a thread. Whenever the parent process calls exec(), it would wait for its child to be created and load ***executable (not wait for its child to get terminated).*** *If it is created and loaded successfully, the system call would return the pid of the new child process. Otherwise, it would return -1.*
  In the naïve PINTOS, process_execute() has been implemented. Thus, we should analyze that function since it would relate to system call exec, which is sys_exec().
  - *Current relation between parent calling exec and child:* Whenever the parent process is called the *process_execute(),* it would invoke the *start_process()* to allocate the thread structure and create a new thread. Afterward, it would load the binary file for the program to get executed. Then, the parent process is waiting for its child until it ensures the binary file is successfully loaded in the memory. Through the implementation, although *exec()* would return after the child is completely loaded, *tid* parameter could still have the valid value even if the *load()* procedure fails. Therefore, it is required to modify the *process_execute()* afterward
  - **Modify process_execute():**
    - Under the thread structure, there should contain a boolean field to be denoted as the load status for checking the file is successfully loaded or not.
    - To synchronize the caller and the newly created child process due to the *process_execute(),* it is required to add a semaphore for exec() system call in the structure of thread, which is *sema_load*. At the beginning, the sema value should be

initialized to 0 when the creating the thread in order to indicate that thread is now loading. In addition, in *process_execute(),* it is required to call *sema_down* to wait for a successful loading procedure for a child process. When it comes to the *sema_up*, we should declare in the *start_process()* in order to indicate the event that the executable file is loaded completely.

- **Add sys_exit(status):**
  The main functionality of the system call exit() is to terminate the current user program and return the status toward the kernel. If the parent process is waiting for it, this function is called and the status would be returned back.
    - Implement *sys_exit():*
        - At first, it is necessary to save the exit status to the process status at the process descriptor.
        - Before ending its termination, it needs to indicate that the thread waiting for this thread should get executed, which is called *sema_up* for the *sema_load* semaphore.

- Implementation of system call related to the file-level: There is the remaining function related to system call related to the file-level such as *sys_create(), sys_remove(), sys_open(), sys_filesize(), sys_read(), sys_write(), sys_seek(), sys_tell(), sys_close()*. Before implementing it, it is essential to comprehend the **File System Mechanism** under the naïve PINTOS implementation.

III. File System Mechanism

a. Analysis of current system

While using the file, the user program would load from the file system by using the system calls, which would be implemented afterwards, to interrupt for the Kernel mode. Therefore, it is vital to understand the file system mechanism in the naïve PINTOS.

- Data Structure
  - **filesys/inode.c/struct inode & filesys/inode.c/ struct inode_disk:** Under the structure of *inode*, it would include the element in *inode list*, the sector number of disk location, the number of openers which is allowed to remove or write, and *inode* content which is located in the *inode_disk* structure. When it comes to that structure, it would consist of the first data sector and the length of the file on the disk. In addition, it has the *magic* number to identify the structure itself.

```
/* In-memory inode. */
struct inode
  {
    struct list_elem elem;              /* Element in inode list. */
    block_sector_t sector;              /* Sector number of disk location. */
    int open_cnt;                       /* Number of openers. */
    bool removed;                       /* True if deleted, false otherwise. */
    int deny_write_cnt;                 /* 0: writes ok, >0: deny writes. */
    struct inode_disk data;             /* Inode content. */
  };
```

```
/* On-disk inode.
   Must be exactly BLOCK_SECTOR_SIZE bytes long. */
struct inode_disk
  {
    block_sector_t start;               /* First data sector. */
    off_t length;                       /* File size in bytes. */
    unsigned magic;                     /* Magic number. */
    uint32_t unused[125];               /* Not used. */
  };
```

  - **filesys/file.c/struct file:** The structure of *inode* or *inode_disk* are inside the file structure, which is contained the *inode* pointer from the file and also store the current position of the thread. In addition, it includes the boolean *deny_write* to record a success or failure of *file_deny_write()* ***(detailed explanation below)*** to allow or deny from being written on the file.

```
/* An open file. */
struct file
  {
    struct inode *inode;        /* File's inode. */
    off_t pos;                  /* Current position. */
    bool deny_write;            /* Has file_deny_write() been called? */
  };
```

- Current function used
  - **filesys/file.c/file_open(struct inode *inode): This** function receives a given *inode* to open the file. When opening the file, the file would be initialized and allocated dynamically. If successfully allocated, the *file* and *inode* would not *NULL* since they have their pointer pointing towards. Then, it would assign the *inode* value to the file's *inode*, assign 0 to the file position, and set the false boolean for *deny_write* to indicate that it could be written. However, if the procedure of allocation fails, the *inode* is closed and the allocated file is free or deallocated, thus reducing the number of openers in the *inode*.

```
struct file *
file_open (struct inode *inode)
{
  struct file *file = calloc (1, sizeof *file);
  if (inode != NULL && file != NULL)
    {
      file->inode = inode;
      file->pos = 0;
      file->deny_write = false;
      return file;
    }
  else
    {
      inode_close (inode);
      free (file);
      return NULL;
    }
}
```

- **filesys/file.c/file_reopen(struct *file file):** Given a file, a new file would be created by invoking *file_open* function. Thus, the number of openers is increasing.

```
struct file *
file_reopen (struct file *file)
{
  return file_open (inode_reopen (file->inode));
}
```

- **filesys/file.c/file_close(struct file *file):** While this function is called, if the file is pointed to valid address *(file != NULL)*, the file is allowed to write on and close the file. Therefore, it is required to free that file for its deallocation procedure to completely close.

```
void
file_close (struct file *file)
{
  if (file != NULL)
    {
      file_allow_write (file);
      inode_close (file->inode);
      free (file);
    }
}
```

- **filesys/file.c/file_get_inode(struct file *file):** This function is to return the *inode* pointer associated with the file.

```
struct inode *
file_get_inode (struct file *file)
{
  return file->inode;
}
```

- Reading a file
  - **filesys/file.c/file_read (struct file *file, void *buffer, off_t size):** The main functionality is to read from position, which *size* is given to indicate the position reading from a given *buffer*. Next, it would traverse by block to read the file and insert the reading data to the buffer. Also, the *bytes_read* would be added to the file position parameter to indicate the position of reading procedure is pointing currently. Therefore, at the end, it would return the number of reading bytes – *bytes_read.*

```
off_t
file_read (struct file *file, void *buffer, off_t size)
{
  off_t bytes_read = inode_read_at (file->inode, buffer, size, file->pos);
  file->pos += bytes_read;
  return bytes_read;
}
```

- **filesys/file.c/file_reat_at(struct *file file, void *buffer, off_t size, off_t file_ofs):** The main functionality is to read from a given offset and put the data back in a *buffer* by invoking *inode_read_at()*. By this, it is not necessary to add up the reading position instead, but it could still return the number of reading bytes.

```
off_t
file_read_at (struct file *file, void *buffer, off_t size, off_t file_ofs)
{
  return inode_read_at (file->inode, buffer, size, file_ofs);
}
```

- Writing a file
  - **filesys/file.c/file_write (struct file *file, const void *buffer, off_t size):** The main functionality is to write to a position, which *size* is given to indicate the position writing toward a given *buffer*. Next, it would traverse by block to write the file. Also, the *bytes_written* would be added to the file position to indicate the position of writing procedure is pointing currently. Afterwards, it would return the number of writing bytes – *bytes_written.*

```
off_t
file_write (struct file *file, const void *buffer, off_t size)
{
  off_t bytes_written = inode_write_at (file->inode, buffer, size, file->pos);
  file->pos += bytes_written;
  return bytes_written;
}
```

  - **filesys/file.c/file_write_at (struct file *file, const void *buffer, off_t size):** The main functionality is to write to a given offset in *buffer* by invoking *inode_write_at()* function.

```
off_t
file_write_at (struct file *file, const void *buffer, off_t size,
               off_t file_ofs)
{
  return inode_write_at (file->inode, buffer, size, file_ofs);
}
```

- **filesys/file.c/file_deny_write (struct file *file):** This function is to reject the writing on the file. From the current implementation, when setting the *deny_write* to be true, we should invoke *inode_deny_write()* function to increment *inode* parameter.

```
void
file_deny_write (struct file *file)
{
  ASSERT (file != NULL);
  if (!file->deny_write)
    {
      file->deny_write = true;
      inode_deny_write (file->inode);
    }
}
```

- **filesys/file.c/file_allow_write (struct file *file):** This function is to allow the writing on the file. From the current implementation, when setting the *deny_write* to be false, we should invoke *inode_allow_write()* function to decrement *inode* parameter.

```
void
file_allow_write (struct file *file)
{
  ASSERT (file != NULL);
  if (file->deny_write)
    {
      file->deny_write = false;
      inode_allow_write (file->inode);
    }
}
```

- **filesys/file.c/file_length (struct file *file):** This function is to return the length of the file by invoke the *inode_length()* function.

```
off_t
file_length (struct file *file)
{
  ASSERT (file != NULL);
  return inode_length (file->inode);
}
```

- **filesys/file.c/file_seek (struct file *file, off_t new_pos):** This function is to reset and result in the new position to the given position.

```
void
file_seek (struct file *file, off_t new_pos)
{
  ASSERT (file != NULL);
  ASSERT (new_pos >= 0);
  file->pos = new_pos;
}
```

- **filesys/file.c/file_tell (struct file *file):** This function is to return the position of the current file.

```
off_t
file_tell (struct file *file)
{
  ASSERT (file != NULL);
  return file->pos;
}
```

- **filesys/filesys.c/filesys_init(bool format):** The main functionality is to initialize the file system. After getting the file system device by using *block_get_role()* function, it is required to initialize the *inode* list with *inode_init()* and simultaneously initialize the free map. Moreover, to perform the format within initialization of the file system, *do_format()* needs to be called ***(detailed explanation below)***. Afterward, if those above steps are satisfied, opening the free map is to load the free map file.

```
void
filesys_init (bool format)
{
  fs_device = block_get_role (BLOCK_FILESYS);
  if (fs_device == NULL)
    PANIC ("No file system device found, can't initialize file system.");

  inode_init ();
  free_map_init ();

  if (format)
    do_format ();

  free_map_open ();
}
```

- **filesys/filesys.c/filesys_done (void):** This function is to shut down the file system and thus close the server by invoking the *free_map_close()* function.

```
void
filesys_done (void)
{
  free_map_close ();
}
```

- **filesys/filesys.c/filesys_create (const char *name, off_t initial_size):** Given a name and size of the file, it is available to create a file system. When it is called, it would assign its directory *(dir)* through opening a directory from *inode* in the root directory by *dir_open_root()* function. To evaluate it success, it needs to satisfy that we could assign a free map, create an *inode*, and add it to the directory. On the other hand, if it fails, it is essential to release a free map by unassigning it. Afterwards, we can close the directory and return whether it is successful or not.

```
bool
filesys_create (const char *name, off_t initial_size)
{
  block_sector_t inode_sector = 0;
  struct dir *dir = dir_open_root ();
  bool success = (dir != NULL
                  && free_map_allocate (1, &inode_sector)
                  && inode_create (inode_sector, initial_size)
                  && dir_add (dir, name, inode_sector));
  if (!success && inode_sector != 0)
    free_map_release (inode_sector, 1);
  dir_close (dir);

  return success;
}
```

- **filesys/filesys.c/filesys_open (const char *name):** Given a *name* of a file, the main functionality of this function is to return the *file pointer*. Same as *filesys_create()*, firstly, it should assign it directory through opening the root. If the *dir* is existed or *dir != NULL*, we can perform to search the file *name* in directory by *dir_lookup().* Then, to be terminated, it should close the directory through *dir_close()* and thus return the file from an *inode* in its entry.

```
struct file *
filesys_open (const char *name)
{
  struct dir *dir = dir_open_root ();
  struct inode *inode = NULL;

  if (dir != NULL)
    dir_lookup (dir, name, &inode);
  dir_close (dir);

  return file_open (inode);
}
```

- **filesys/filesys.c/filesys_remove (const char *name):** Given a *name* of the file, it would delete the file. Same as two above functions, it would find the directory from *dir_open_root().* To indicate it success, the directory should be not *NULL* and it could perform *dir_remove()* function to delete the *inode* and its entries. When it gets terminated, it would return whether this procedure is successful or not.

```
bool
filesys_remove (const char *name)
{
  struct dir *dir = dir_open_root ();
  bool success = dir != NULL && dir_remove (dir, name);
  dir_close (dir);

  return success;
}
```

- **filesys/filesys.c/do_format (void):** This function is to format the file system by changing to create a new free map.

```
static void
do_format (void)
{
  printf ("Formatting file system...");
  free_map_create ();
  if (!dir_create (ROOT_DIR_SECTOR, 16))
    PANIC ("root directory creation failed");
  free_map_close ();
  printf ("done.\n");
}
```

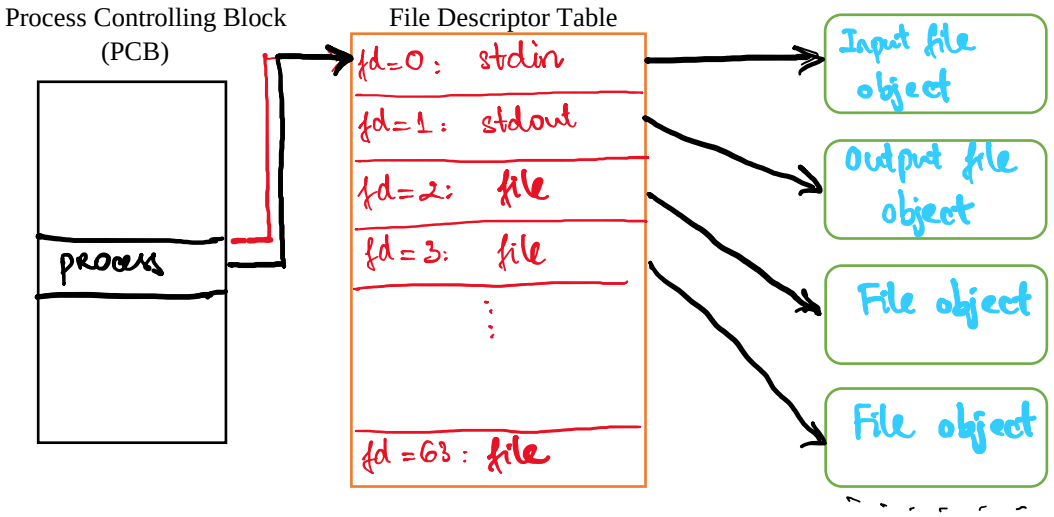
- *Problem statement & Brief Solution:* By those function above, we could thoroughly understand the mechanism of file system by analyzing each function's procedure and functionality. However, if the system calls for the file-level is called, there is no relation to let those system call perform its functionality toward the file system. In addition, if the current running file is modified, the naïve PINTOS could not function properly to address these edgy cases. Hence, it is required to add a new data structure inside the thread structure to manage and control its logic, which is File Descriptor Table (FDT) as stated in the Project 2 Description.

b. Solution for implementation of file-level system call and denying writes to executable files

- Initialization of ***File Descriptor Table***

- Before implementing this idea, it is required to initialize some convention: (i) each process/thread has their own FDT (max size: 64 entries ranging from 0 to 63) (ii) FDT is an array of pointer which is pointed toward the target struct file
- To implement it, we could allocate FDT in the memory, which is adding the pointer inside the thread structure. Indeed, we could notice the index of FDT as *fd*. Therefore, when the thread is created, it would allocate the FDT and initialize the pointer to FDT. While the thread gets terminated, it is necessary to close all files and deallocate the FDT.
- Some conventions
  - *fd = 0* means that it would store the struct file of keyboard/standard input (*stdin*)
  - *fd = 1* means that it would store the struct file of screen/monitor/standard output (*stdout*)
- While the file system is called, system call corresponded to the file system function is executed. Therefore, to avoid the race condition on the file which many system calls are getting executed at the same time, we should implement a global lock and thus initialize it *syscall_init()* function. By this, it could protect the file system related to its code.

Big picture



i. Implementation of file-level system call

- **Add sys_create(const char *file, unsigned initial_size):** This function is to create a file which have size of *initial_size*. Hence, it is required to check whether the given file is suitable or not. The file is invalid and return to error if the file is *NULL* or its address is unavailable by checking *address_validation*() function. If the condition is satisfied, it would return a boolean by invoking *filesys_create()* function.
- **Add sys_remove(constant char *file):** This function is to remove a file which name is *file*. The condition for checking to perform a deletion is the same as the previous function. If the condition is satisfied, it would return boolean by invoking *filesys_remove()* function.
- **Add sys_open(constant char *file):** This function is to open the file associated with the path in "*file*". Firstly, it is required to check the condition whether the file or its address is valid or not. If it is not appropriate, we should return error. On other hand, we can invoke a *filesys_open()* function to store in the FDT with the struct file. At last, it would return the index of FDT (*fd*).
- **Add sys_filesize(int fd):** This function is to return the file size. Before performing it, it needs to check whether the file is in the entry of FDT or not. If there is not located inside, it would return error. Otherwise, it would invoke *file_length()* to return the *size* (in bytes) of the open file.
- **Add sys_read(int fd, void *buffer, unsigned size):** Same as above function, it needs to check whether the index of FDT is valid or not, and the file is valid or not. If it satisfies, we could read the *size* bytes from the open file within the *fd* into *buffer* by invoking function *file_read().* Also, we should be aware that if *fd = 0*, instead of using the same function to read as *file_read()*, it is necessary to have other mechanism to read from keyboard input. At last, the function should return the number of reading bytes – *bytes_read*.
- **Add sys_write():** In this function, we could write the size bytes from *buffer* to open file with index of FDT. Firstly, we should check if the *fd* is available or not. If it is not available, we can return error. Then, check whether *fd* is pointed to the entry of FDT contained the file or not. If it is not, return an error. Next, it is necessary to write a file by invoking the *file_write().* In addition, we should be aware if *fd = 1*, instead of using the same function as *file_write()*, we should concern this edge case to let it write to the console with other mechanism. At last, the function should return the number of writing bytes – *bytes_written*.
- **Add sys_seek(int fd, unsigned position):** This function is to change the target position for reading or writing with the open file with its index *fd*. Besides checking the file is not *NULL*, we could invoke *file_seek()* function to perform afterwards.
- **Add sys_tell(int fd):** This function is to return the position of the open file to be read or written on. If the file is *NULL*, we should return an error. Otherwise, invoke *file_tell()* to return the position as stated.
- **Add sys_close():** This function is to close the index of FDT by invoking *file_close()* function. However, if *fd* points to the *stdin* or *stdout* or fd is controlled by current process, we should return error. Otherwise, we could close the file and reduce the number of *fd* by deallocating in FDT.
- **Modify userprog/exception.c/page_fault():** We should provide a check to it to find out which process is bad. If there exits that type of process, it is necessary to kill that process and result the *exit status (-1)* to indicate the occurrence of page fault.

ii. Implementation of denying writes to executable files

If operating system tries to execute the file that is being modified, the result is unpredictable. Hence, the objective for this implementation is to not allow the file to be modified when it is opened for getting executed. When the file is loaded for executing, it is necessary to deny the write on it. Also, when the file finishes executed, we could allow other to write on the file. Therefore, in this implementation, we just set a permission to write or deny writing.

- **Add a file pointer:** This pointer is added into the thread structure to indicate the status of running file that is allowed to write or not.
- **Modify userprog/process.c/load():** When the file is opened that means the process starts and loads the file, we should provoke *file_deny_write()* to reject the writing on the file.
- **Modify userprog/process.c/process_exit():** When the process ends, we could provoke others to write again by using *file_allow_write()* function