OS Project 3 - Team 4

20185083 서영석, 20185141 이준명, 20185010 권강민

1. Introduction

This report covers the implementation tasks of Process Termination Messages, Argument Passing, and System Calls in the Pintos operating system. Firstly, we implemented the argument passing feature. Using the process_execute() function, we parsed the arguments and treated them as program name and arguments, separated by spaces. Secondly, we implemented the system call handling functionality. By implementing the system call handler, we were able to perform appropriate actions based on the system call numbers defined in the system. Additionally, we implemented the functionality to print process termination messages. Upon a process termination, the name and exit code of the terminated process are printed. This report will provide an explanation of the problem scenarios, implementation methods, and outcomes for these three tasks.

2. Overview

A. Argument passing (Problem 2)

The task at hand is to improve the argument passing mechanism in the Pintos operating system. Currently, the arguments are passed as a raw string to the process_execute() function, which poses challenges for proper handling and parsing of command line arguments.

According to the pintos statement, originally the arguments are passed as a space-separated list of words. The first word should represent the program name, followed by the subsequent words representing the arguments. It is important to handle spaces correctly, treating multiple spaces between words as equivalent to a single space.

To tackle this problem, we will modify the process_execute() function to effectively parse the raw string of arguments and separate them into individual words. This requires to develop a method that can appropriately tokenize the string, and store each word in a structured format.

B. System Calls (Problem 3)

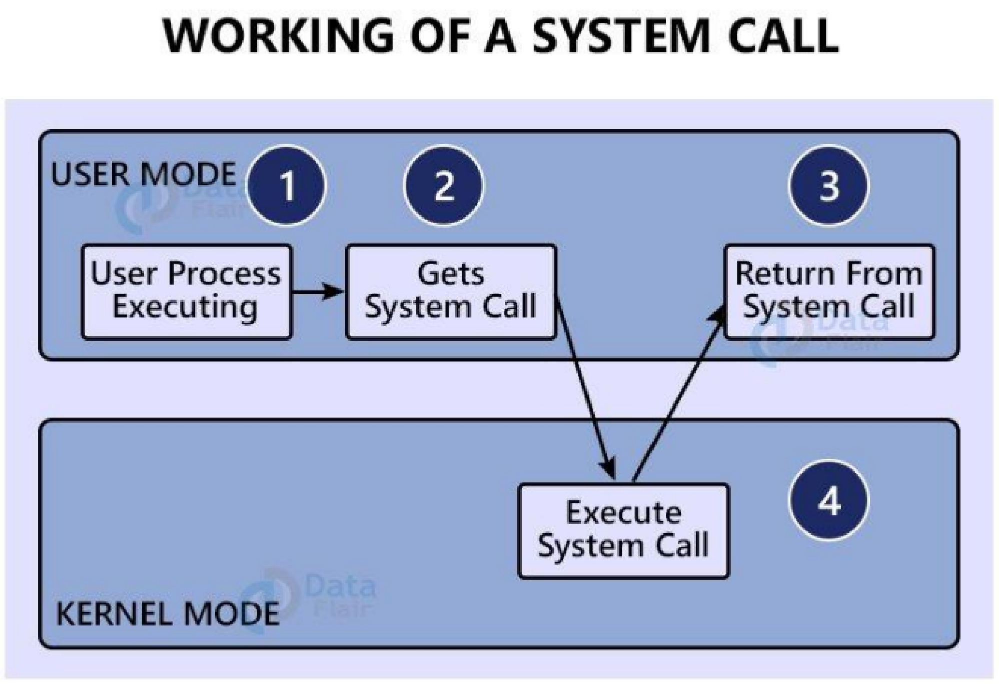


Fig 1. plot about how system call working

This problem focuses on implementing the system call handler in the Pintos operating system. The system call handler is responsible for handling system calls according to the provided requirements. A skeleton implementation of the system call handler can be found in "~/src/userprog/syscall.c". This implementation terminates the process upon handling system calls.

The main objectives of this problem are as follows:

1. Retrieve the system call number and execute the corresponding actions. The system call numbers for each specific system call are defined in `src/syscall-nr.h`.
2. Implement the user-level functions for each system call defined in `/src/lib/syscall.h`. These functions allow user programs to invoke system calls from C programs using inline assembly code. It is important to ensure that these functions work correctly and handle various scenarios.

To achieve goals, proper error handling is required when invalid arguments are passed to system calls. Acceptable options include returning error values (for calls that have return values), returning undefined values, or terminating the process.

C. Process Termination Messages (Problem 1)

We write termination messages by using printf function on exit() system calls after implementing system calls.

D. Sub Problem : Memory validation

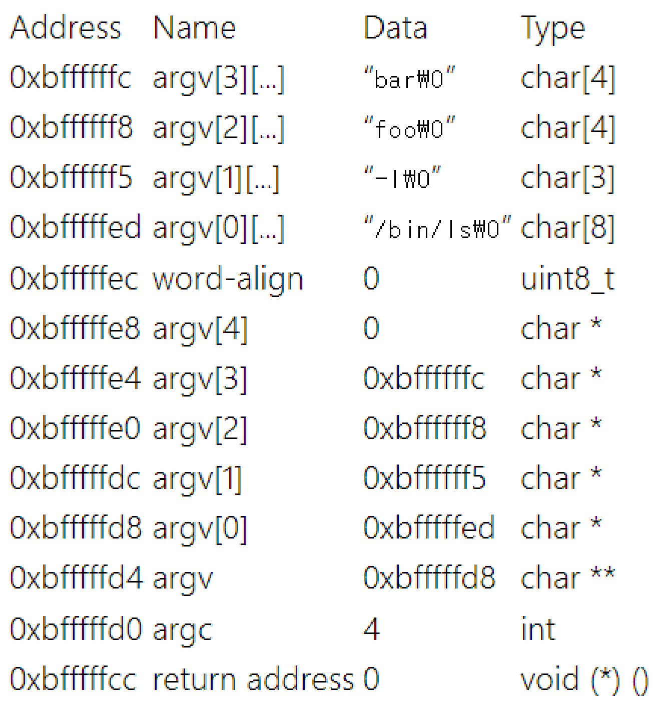
If a user thread attempts to reference an invalid memory area, it is terminated to prevent a system crash. However, when a system interrupt occurs, including during a system call, the code execution switches to the kernel mode. Consequently, if an invalid memory reference happens within a system call, it can lead to a system crash.

Since the only operating system has access rights to the physical address, there is a risk of memory corruption when a user memory passes a physical address as an argument and invokes memory read/write system calls. To remove these risks, it is essential to implement proper checks and validation mechanisms to ensure the integrity and safety of memory operations within the system calls.

3. Code-level development

A. Argument passing (Problem 2)

To correctly invoke system calls at the user-level, it is necessary to implement argument passing as a prerequisite. Therefore, we started by implementing argument passing, which allowed us to successfully parse the raw string using the reference from `lib/string.h`. As a result, we obtained a modified string where spaces were replaced with `\0` characters. This modified string made it easier to implement argument passing.



Address	Name	Data	Type
0xbffffffc	argv[3][...]	“bar₩0”	char[4]
0xbffffff8	argv[2][...]	“foo₩0”	char[4]
0xbffffff5	argv[1][...]	“-l₩0”	char[3]
0xbfffffed	argv[0][...]	“/bin/ls₩0”	char[8]
0xbfffffec	word-align	0	uint8_t
0xbfffffe8	argv[4]	0	char *
0xbfffffe4	argv[3]	0xbffffffc	char *
0xbfffffe0	argv[2]	0xbffffff8	char *
0xbfffffdc	argv[1]	0xbffffff5	char *
0xbfffffd8	argv[0]	0xbfffffed	char *
0xbfffffd4	argv	0xbfffffd8	char **
0xbfffffd0	argc	4	int
0xbfffffcc	return address	0	void (*) ()

Fig 2. structure of process stack of pintos

The provided figure represents the process stack structure attached to the Pintos documentation page. Based on this structure, we realized that we could easily implement argument passing by simply writing the parsed string onto the top of the stack. In summary, the specific steps involved in implement argument passing in stack area are as follows:

1. Convert the space-separated string, such as "filename arg1 arg2," into a modified string using the `strtok_r()` function.
   - During this process, the string transforms from `"filename arg1 arg2"` to `"filename\0arg1\0arg2"`.
   - Simultaneously, create a char pointer array `argv` that points to the first character of each word.
2. Copy the modified string to the top of the stack, and allocate space for the char pointer array just below it after word-aligning.
3. Allocate space below that for a double pointer pointing to `argv` and an integer `argc` representing the number of arguments.

belows blocks are code-level implementations of argument passing.

- `src/userprog/process.c`

```
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);

+ char * token, save_ptr;
+ int file_name_length = strlen(file_name);
+ char cp_str[file_name_length + 1];
+ strlcpy(cp_str,file_name,file_name_length+1);
  /* Create a new thread to execute FILE_NAME. */
+ token = strtok_r(cp_str, " ",&save_ptr);
  tid = thread_create (token, PRI_DEFAULT, start_process, fn_copy);
  if (tid == TID_ERROR)
    palloc_free_page (fn_copy);
  return tid;
}
```

- `src/userprog/process.c`

```
static void
start_process (void *file_name_)
{
  char *file_name = file_name_;
  struct intr_frame if_;
  bool success;
  /* Initialize interrupt frame and load executable. */
  memset (&if_, 0, sizeof if_);
  if_.gs = if_.fs = if_.es = if_.ds = if_.ss = SEL_UDSEG;
  if_.cs = SEL_UCSEG;
  if_.eflags = FLAG_IF | FLAG_MBS;


+ char *token,*save_ptr;
+ char * tokenArr[128];
+ int tokenNum = 0;
+ int wordSize = 3;
+ int stringLen = strlen(file_name);
+ for(token = strtok_r(file_name, " ",  &save_ptr); token != NULL; token = strtok_r(NULL, " ", &save_ptr)){
+   tokenArr[tokenNum] = token;
+   tokenNum += 1;
+ }
+ wordSize += tokenNum + 1;
+ wordSize += stringLen/4 + 1;
+ tokenArr[tokenNum] = NULL;

  success = load (tokenArr[0], &if_.eip, &if_.esp);

  /* If load failed, quit. */
  if (!success){
    palloc_free_page (file_name);
    thread_exit ();
  }
+ if_.esp -= wordSize * 4;
+ void * string_start = if_.esp+12+4*(tokenNum+1);
+ void * pointer_array_start = if_.esp + 12;
+ *(int *)(if_.esp) = 0;
+ *(int *)(if_.esp+4) = tokenNum;
+ *(int *)(if_.esp+8) = pointer_array_start;
+ memcpy(string_start, file_name,stringLen);
+ *(int *)(pointer_array_start) = string_start;
+ int i = 1;
+ for (; i <tokenNum;i++){
+   *(int *)(pointer_array_start+i*4) = *(int*)pointer_array_start + tokenArr[i]-tokenArr[0];
+ }
+ *(int*)(pointer_array_start+i*4) = 0;
+ palloc_free_page (file_name);


  asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) : "memory");
  NOT_REACHED ();
}
```

B. System Calls (Problem 3)

- `src/userprog/syscall.c`

```
static void
syscall_handler (struct intr_frame *f UNUSED)
{
  /*Memory validation check codes......*/
+ switch (syscall_num){
+   case SYS_HALT : sys_halt(); break;
+   case SYS_EXIT : sys_exit(f); break;
+   case SYS_EXEC : sys_exec(f); break;                   /* Start another process. */
+   case SYS_WAIT : sys_wait(f); break;                   /* Wait for a child process to die. */
+   case SYS_CREATE : sys_create(f); break;                 /* Create a file. */
+   case SYS_REMOVE : sys_remove(f); break;                 /* Delete a file. */
+   case SYS_OPEN : sys_open(f); break;                  /* Open a file. */
+   case SYS_FILESIZE : sys_filesize(f);break;               /* Obtain a file's size. */
+   case SYS_READ : sys_read(f); break;                   /* Read from a file. */
+   case SYS_WRITE : sys_write(f); break;                 /* Write to a file. */
+   case SYS_SEEK: sys_seek(f); break;                   /* Change position in a file. */
+   case SYS_TELL: sys_tell(f); break;                    /* Report current position in a file. */
+   case SYS_CLOSE: sys_close(f); break;                 /* Close a file. */

+    default :{
+       printf ("Unimplemented system call!\n syscall num : %d\n", syscall_num);
+       thread_exit ();
+   }
+ }
}
```

we implemented 13 system calls on `syscall.c` with these functions. These are too much changes, so we do not attach all add/deleted lines on this report. Rather, we write changes for prior functions.

B.1 File descriptor

To facilitate access to files within a process, the operating system manages files using file descriptors. For the purpose of OS-level file descriptor management, a `File Descriptor Table (FDT)` has been defined in `src/threads/thread.h`. When accessing a file, a file pointer is mapped to an entry in this table. It's important to note that file descriptor allocation follows a convention where 0 is reserved for `STDIN_FILENO`(standard input) and 1 is reserved for `STDOUT_FILENO`(standard output). Therefore, when assigning file descriptor indices, an offset of 2 must be added.

- `src/threads/thread.h`

```
struct thread
  {
    //other thread information......
+   struct file * FDT[128];         //file descriptor array to store file struct.
    //......other thread information
  };
```

- `src/userprog/syscall.c`

```
void sys_open(struct intr_frame *f){
  char* fileName = stack_pop(f, 1);
  if(fileName == NULL||isAddressUpperPHYS(fileName,0)){
    sys_unexpected_exit();
  }
  struct file * result = filesys_open(fileName);
  if(result == NULL){
    storeReturnVal(f, -1);
    return;
  }

  struct file ** FDT = thread_current()->FDT;
  int i = 0;
  for(; i < 128; i++){
    if(FDT[i] == NULL){
      FDT[i] = result;
      storeReturnVal(f, i+2);
      return;
    }
  }
}
void sys_close(struct intr_frame *f){
  int fileIndex = stack_pop(f, 1);
  struct thread * current_thread = thread_current();
  struct file ** FDT = &current_thread->FDT;
  if(fileIndex < 2 || fileIndex > 130){
    sys_unexpected_exit();
  }
  else if(FDT[fileIndex-2] == NULL){
    sys_unexpected_exit();
  }
  else{
    file_close(FDT[fileIndex-2]);
    FDT[fileIndex-2] = NULL;
  }
}
```

B.2 Wait system call

The `wait()` system call plays a crucial role in the program execution process, so we provide short describe about `wait()` system call. To implement Wait, we modified the thread struct to include storage for the parent and child process information. This modification allowed us to track the parent and child relationships. Additionally, we made the necessary updates in the `thread_create` and `init_thread` functions to store the parent and child information.

- `src/threads/thread.h`

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

+   struct list_elem child_list_elem;//list_elem for parent child list
+   struct list child_list;         //list of child
+   int exit_status;                //exit_status store to pass parent
+   bool is_parent_waiting;         //store whether parent is waiting this process or not

+   struct file * FDT[128];         //file descriptor array to store file struct.
+   struct file* selfFile;
    /* Shared between thread.c and synch.c. */
    struct list_elem elem;              /* List element. */

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir;                  /* Page directory. */
#endif
    /* Owned by thread.c. */
    unsigned magic;                     /* Detects stack overflow. */
  };
```

- `src/threads/thread.c`

```
...

tid_t
thread_create (const char *name, int priority,
               thread_func *function, void *aux)
{
  /* ...Initialize codes.*/
  init_thread (t, name, priority);
  tid = t->tid = allocate_tid ();

+ struct thread *parent_thread = thread_current();
+ t->parent_tid = parent_thread->tid;
+ list_push_back (&parent_thread->child_list, &t->child_list_elem);

  old_level = intr_disable ();

  kf = alloc_frame (t, sizeof *kf);
  kf->eip = NULL;
  kf->function = function;
  kf->aux = aux;
  /* remain codes...*/
}


static void
init_thread (struct thread *t, const char *name, int priority)
{
  ASSERT (t != NULL);
  ASSERT (PRI_MIN <= priority && priority <= PRI_MAX);
  ASSERT (name != NULL);


  memset (t, 0, sizeof *t);
  t->status = THREAD_BLOCKED;
  strlcpy (t->name, name, sizeof t->name);
  t->stack = (uint8_t *) t + PGSIZE;
  t->priority = priority;
  t->magic = THREAD_MAGIC;
  list_push_back (&all_list, &t->allelem);
  /*parent, child process initialize*/
+ list_init(&(t->child_list));
+ t->exit_status = -1;
+ t->is_parent_waiting = false;
}
...
```

In order for the parent process to be able to know the exit state of its child process, we made changes to the thread reaping logic to ensure that the child process is not reaped immediately during scheduling time if the parent thread still exists. Unlike the original behavior, we modified the `thread_schedule_tail` function to only perform reaping if the `parent_thread` is not present. To check for the existence of the parent, we implemented a function that finds the process based on the tid in the `all_elem` list. Additionally, we also implemented the `thread_reap` function to allow the parent to reap the child thread in the future. These changes enable the parent process to maintain visibility and control over the exit state of its child process.

- `src/threads/thread.c`

```
...
+struct thread* find_thread(tid_t target){
+  struct list_elem *e;
+  struct thread * returnThread = NULL;
+  for(e = list_begin(&all_list); e!= list_end(&all_list);e = list_next(e)){
+    struct thread * e_thread = list_entry(e,struct thread,allelem);
+    if(e_thread->tid == target){
+      returnThread = e_thread;
+    }
+  }
+  return returnThread;
+}

...

thread_schedule_tail (struct thread *prev)
{
  struct thread *cur = running_thread ();
  ASSERT (intr_get_level () == INTR_OFF);
  cur->status = THREAD_RUNNING;
  thread_ticks = 0;
#ifdef USERPROG
  process_activate ();
#endif
+ if (prev != NULL && prev != initial_thread && prev->status == THREAD_DYING && find_thread(prev->parent_tid) == NULL)
    {
-     ASSERT (prev != cur);
-     palloc_free_page (prev);
+     thread_reap(prev);//reap when orphan and thread status == dying.
    }
}

...

+void thread_reap(struct thread * target_thread){
+    ASSERT(target_thread != NULL);
+    ASSERT(target_thread != thread_current());
+    ASSERT(target_thread->status == THREAD_DYING);
+    struct thread * parent_thread= find_thread(target_thread->parent_tid);
+    if(parent_thread != initial_thread){
+      list_remove(&target_thread->child_list_elem);
+    }
+    list_remove(&target_thread->allelem);
+
+    palloc_free_page(target_thread);
+}
+static void thread_die_child_reaping(struct thread * target_thread){
+  struct list_elem * e;
+  for(e = list_begin(&target_thread->child_list); e!= list_end(&target_thread->child_list);e = list_next(e)){
+    struct thread * e_thread = list_entry(e,struct thread,allelem);
+    if(e_thread->status == THREAD_DYING){
+      thread_reap(e_thread);
+    }
+  }
+}

...
```

When the `wait()` system call is invoked, the syscall checks if the provided tid belongs to a child of the current process. If the tid corresponds to a child and the child is in the `THREAD_DYING` state, the system reaps the child by collecting its exit status and returns it. If the child is not in the `THREAD_DYING` state, the system sets the `is_parent_waiting` flag of the child thread to true and blocks the parent process.

Subsequently, when the child thread exits, it checks if the parent is waiting for it. If the parent is waiting, it unblocks the parent process. Upon being unblocked, the parent retrieves the child's exit status and returns it. This mechanism ensures that the parent process waits for its child and receives the appropriate exit status once the child has terminated.

- `src/userprog/syscall.c`

```
...
+void sys_wait(struct intr_frame * f){
+  tid_t target_tid = stack_pop(f,1);
+  int return_val = sys_wait_W(target_tid);
+  storeReturnVal(f, return_val);
+}
+int sys_wait_W(tid_t target_tid){
+
+  enum intr_level old_level = intr_disable();
+  struct list_elem * e;
+  struct thread * target_child = NULL;
+  int returnVal = -1;
+  for(e = list_begin(&thread_current()->child_list); e!= list_end(&thread_current()->child_list);e = list_next(e)){
+    struct thread * e_thread = list_entry(e,struct thread,child_list_elem);
+    if(e_thread->tid == target_tid){
+      target_child = e_thread;
+    }
+  }
+if(target_child == NULL || target_child->is_parent_waiting == true){
+    returnVal = -1;
+  }
+  else if(target_child->status == THREAD_DYING){
+    returnVal = target_child->exit_status;
+    thread_reap(target_child);
+  }
+  else if(target_child->status != THREAD_DYING){
+    target_child->is_parent_waiting = true;
+    thread_block();
+    returnVal = target_child->exit_status;
+    thread_reap(target_child);
+
+  }
+  else{
+    printf("unexpected wait condition\n");
+  }
+  intr_set_level (old_level);
+  return returnVal;
+}
...
```

- `src/threads/thread.c`

```
void
thread_exit (void)
{
  ASSERT (!intr_context ());

#ifdef USERPROG
  process_exit ();
#endif
  /* Remove thread from all threads list, set our status to dying,
     and schedule another process.  That process will destroy us
     when it calls thread_schedule_tail(). */
  intr_disable ();
- list_remove (&thread_current()->allelem);
+ struct thread * current_thread = thread_current();
+ struct thread * parent_thread = find_thread(current_thread->parent_tid);
  thread_current ()->status = THREAD_DYING;
+ if(current_thread->is_parent_waiting){
+   thread_unblock(parent_thread);
+ }
  schedule ();
  NOT_REACHED ();
}
```

C. Process Termination Messages (Problem 1)

We make a process to emit termination message by add print statement at exit system call.

- `src/userprog/syscall.c`

```
+void sys_exit(struct intr_frame * f){
+  thread_current()->exit_status = stack_pop(f, 1);
+  printf ("%s: exit(%d)\n", thread_current()->name,thread_current()->exit_status);
+  thread_exit();
+}
```

D. Sub Problem - Memory validation

To implement robust system calls, it is crucial to consider scenarios where insecure memory access occurs at the user-level. Without proper memory access logic in place, users can potentially exploit the system through unauthorized access or perform erroneous write operations that could lead to serious system failures.

According to the Pintos documentation, there are two suggested approaches to address this issue, and we opted to implement safe memory access using the page_fault() function as it seemed more intuitive.

We identified two insecure access scenarios:

- Case 1: When a user attempts to access the invalid address.
- Case 2: When a user provides an argument address that points to an unallocated region.
- Case 3: When a user passes invalid address of stack pointer `%esp`.

By utilizing the page_fault() function, we were able to handle these scenarios and ensure secure memory access.

In both cases, operating system triggering a `page_fault()` error, so we can address insecure access by analyzing condition related each case after the fallback to the `page_fault()` function.

Case 1. Invalid access problem

The first case occurs when process accesses a invalid address. Firstly, we must check whether the user passed the invaild argument which pointing invalid address(ex. user call `write()` system call with not writable address) when the system call is executed in kernel mode. This can be checked with the `!user` condition. This condition check whether error occurs in kernel mode or not.

Secondly, If user attempt to access the kernel area directly, os also occur `page_fault()` error. (ex. user program directly refernce address of kernel area). We can check this case by `is_kernel_vaddr(fault_addr)` condition statement.

If one of the condition statement is true, the OS will excute an exit(-1) to indicate that the access is invalid and terminate. So we can detect case 1 by `!user || is_kernel_vaddr(fault_addr)` condition statement.

Case 2. Unallocated page problem

The second case occurs when there is an attempt to access an unallocated page. This can be checked using the `not_present` variable. The `not_present` variable is used to determine the cause of the page fault from the error code. If a reference is made to a page that is not allocated, the `not_present` variable will be true. In this case, the `sys_unexpected_exit()` function is called to handle the unexpected termination.

- `src/userprog/exception.c`

```
#include "userprog/exception.h"
#include <inttypes.h>
#include <stdio.h>
#include "userprog/gdt.h"
#include "threads/interrupt.h"
#include "threads/thread.h"
+#include "threads/vaddr.h"
+#include "userprog/syscall.h"

...
static void
page_fault (struct intr_frame *f)
{
  bool not_present;  /* True: not-present page, false: writing r/o page. */
  bool write;        /* True: access was write, false: access was read. */
  bool user;         /* True: access by user, false: access by kernel. */
  void *fault_addr;  /* Fault address. */

  /* Obtain faulting address, the virtual address that was
     accessed to cause the fault.  It may point to code or to
     data.  It is not necessarily the address of the instruction
     that caused the fault (that's f->eip).
     See [IA32-v2a] "MOV--Move to/from Control Registers" and
     [IA32-v3a] 5.15 "Interrupt 14--Page Fault Exception
     (#PF)". */
  asm ("movl %%cr2, %0" : "=r" (fault_addr));

  /* Turn interrupts back on (they were only off so that we could
     be assured of reading CR2 before it changed). */
  intr_enable ();

  /* Count page faults. */
  page_fault_cnt++;

  /* Determine cause. */
  write = (f->error_code & PF_W) != 0;
  user = (f->error_code & PF_U) != 0;

  +if(not_present){
  +  sys_unexpected_exit();
  +}
  +if (!user || is_kernel_vaddr(fault_addr)) {
  +  sys_unexpected_exit();
  +}
  /* To implement virtual memory, delete the rest of the function
     body, and replace it with code that brings in the page to
     which fault_addr refers. */
@@ -159,3 +166,4 @@ page_fault (struct intr_frame *f)
  kill (f);
}
```

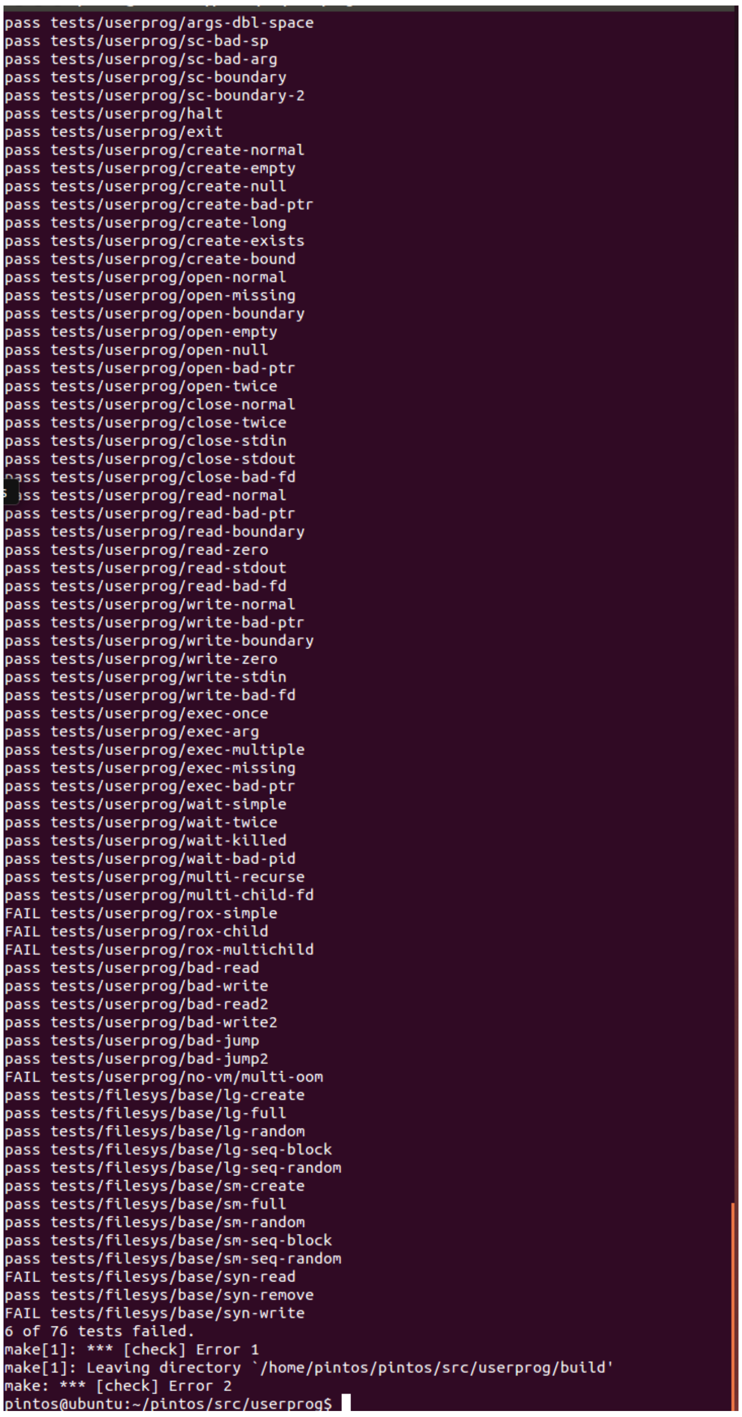
Case 3. Invalid `%esp`

When receiving arguments for executing a system call, the stack may invade the kernel area. Therefore, it is necessary to check the starting address of the stack and the number of arguments by examining `%esp`, and to check whether that the stack does not invade the kernel area. This logic implemented in `src/userprog/syscall.c`

- `src/userprog/syscall.c`

```
syscall_handler (struct intr_frame *f UNUSED)
{
  //uint32_t segment_start = 0x08084000;//TODO
  uint32_t segment_start = PHYS_BASE - PGSIZE;//TODO
  if(f->esp > PHYS_BASE-4){
    sys_unexpected_exit();
  }
  int checkbytes = 0;
  int syscall_num = stack_pop(f,0);
  switch (syscall_num){
    case SYS_HALT : checkbytes = 4; break;
    case SYS_EXIT : ;
    case SYS_EXEC : ;
    case SYS_REMOVE : ;
    case SYS_TELL: ;                    /* Report current position in a file. */
    case SYS_CLOSE: ;
    case SYS_FILESIZE : ;               /* Obtain a file's size. */
    case SYS_OPEN : ;                  /* Open a file. */
    case SYS_WAIT : checkbytes = 8; break;                  /* Wait for a child process to die. */
    case SYS_CREATE : ;                 /* Delete a file. */
    case SYS_READ : ;                   /* Read from a file. */
    case SYS_SEEK: checkbytes = 12;break;                   /* Change position in a file. */
    case SYS_WRITE : checkbytes = 16; break;                 /* Write to a file. */
  }
  if(f->esp > PHYS_BASE-checkbytes){
    sys_unexpected_exit();
  }
...
```

4.Results



```
pass tests/userprog/args-dbl-space
pass tests/userprog/sc-bad-sp
pass tests/userprog/sc-bad-arg
pass tests/userprog/sc-boundary
pass tests/userprog/sc-boundary-2
pass tests/userprog/halt
pass tests/userprog/exit
pass tests/userprog/create-normal
pass tests/userprog/create-empty
pass tests/userprog/create-null
pass tests/userprog/create-bad-ptr
pass tests/userprog/create-long
pass tests/userprog/create-exists
pass tests/userprog/create-bound
pass tests/userprog/open-normal
pass tests/userprog/open-missing
pass tests/userprog/open-boundary
pass tests/userprog/open-empty
pass tests/userprog/open-null
pass tests/userprog/open-bad-ptr
pass tests/userprog/open-twice
pass tests/userprog/close-normal
pass tests/userprog/close-twice
pass tests/userprog/close-stdin
pass tests/userprog/close-stdout
pass tests/userprog/close-bad-fd
pass tests/userprog/read-normal
pass tests/userprog/read-bad-ptr
pass tests/userprog/read-boundary
pass tests/userprog/read-zero
pass tests/userprog/read-stdout
pass tests/userprog/read-bad-fd
pass tests/userprog/write-normal
pass tests/userprog/write-bad-ptr
pass tests/userprog/write-boundary
pass tests/userprog/write-zero
pass tests/userprog/write-stdin
pass tests/userprog/write-bad-fd
pass tests/userprog/exec-once
pass tests/userprog/exec-arg
pass tests/userprog/exec-multiple
pass tests/userprog/exec-missing
pass tests/userprog/exec-bad-ptr
pass tests/userprog/wait-simple
pass tests/userprog/wait-twice
pass tests/userprog/wait-killed
pass tests/userprog/wait-bad-pid
pass tests/userprog/multi-recurse
pass tests/userprog/multi-child-fd
FAIL tests/userprog/rox-simple
FAIL tests/userprog/rox-child
FAIL tests/userprog/rox-multichild
pass tests/userprog/bad-read
pass tests/userprog/bad-write
pass tests/userprog/bad-read2
pass tests/userprog/bad-write2
pass tests/userprog/bad-jump
pass tests/userprog/bad-jump2
FAIL tests/userprog/no-vm/multi-oom
pass tests/filesys/base/lg-create
pass tests/filesys/base/lg-full
pass tests/filesys/base/lg-random
pass tests/filesys/base/lg-seq-block
pass tests/filesys/base/lg-seq-random
pass tests/filesys/base/sm-create
pass tests/filesys/base/sm-full
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
FAIL tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
FAIL tests/filesys/base/syn-write
6 of 76 tests failed.
make[1]: *** [check] Error 1
make[1]: Leaving directory `/home/pintos/pintos/src/userprog/build'
make: *** [check] Error 2
pintos@ubuntu:~/pintos/src/userprog$
```

By implementing the code effectively, we have successfully resolved the majority of the challenges outlined in the assignment. Our solution encompasses the implementation of argument passing, ensuring the proper handling of command line arguments. Additionally, we have developed a system call handler capable of managing the required system calls. We have also incorporated a suitable approach for managing user memory access, prevent any potential insecure access.
Although our solution performed well, we got 6 failures out of the 76 conducted tests. Unfortunately, due to time

constraints, we were unable to extensively debug and conduct further testing to identify the root causes of these failures. Consequently, we were unable to resolve these specific test cases.