Project2: User Programs

PintOS

陈震雄

2025 年 3 月 15 日

武汉大学

Table of contents

Introduction

Introduction

Now that you've worked with Pintos and are becoming familiar with its infrastructure and thread package, **it's time to start working on the parts of the system that allow running user programs.**

The base code already supports loading and running user programs, but no I/O or interactivity is possible. In this project, you will enable programs to interact with the OS via system calls.

You will be working out of the userprog/ directory for this assignment, but you will also be interacting with almost every other part of Pintos. We will describe the relevant parts below.

Preparation

Problem1

```
Kernel panic in run: PANIC at ../../threads/vaddr.h:84 in vtop(): assertion `is_kernel_vad
dr (vaddr)' failed.
Call stack: 0xc002a296 0xc002ce51 0xc002d5c6 0xc002c7f4 0xc0021b53 0xc0021c74 0xc002114b 0
xc0023360 0xc0023746 0xc0021cae 0xc0020c70 0xc00202df
Translation of call stack:
0xc002a296: debug_panic (.../../lib/kernel/debug.c:38)
0xc002ce51: vtop (.../../../threads/vaddr.h:86)
0xc002d5c6: pagedir_activate (...../userprog/pagedir.c:230)
0xc002c7f4: process_activate (...../userprog/process.c:128)
0xc0021b53: thread_schedule_tail (..../../threads/thread.c:699)
0xc0021c74: schedule (..../../threads/thread.c:726)
0xc002114b: thread_yield (..../../threads/thread.c:325)
0xc0023360: sema_up (...../../threads/synch.c:124)
0xc0023746: lock_release (...../../threads/synch.c:276)
0xc0021cae: allocate_tid (..../../threads/thread.c:738)
0xc0020c70: thread_init (..../../threads/thread.c:102)
0xc00202df: pintos_init (...d/../../threads/init.c:91)
```

paging_init () is called after thread_init(). So the assertion is false.

Problem2

```
1   // thread.c
2   static bool schedule_started;
3
4   void thread_start (void) {
5     schedule_started = true;
6     ...
7   }
8
9   void thread_yield (void) {
10    if (!schedule_started)
11      return;
12    ...
13  }
```

```
FAIL tests/userprog/sc-bad-sp
Kernel panic in run: PANIC at ../../threads/thread.c:322 in thread_yield(): assertion `!in
tr_context ()' failed.
Call stack: 0xc002a2af 0xc0021126 0xc0023379 0xc002709c 0xc00223d1 0xc00225f1 0xc0026e3b 0
xc00269ab 0xc00265c5 0xc0020357
Translation of call stack:
0xc002a2af: debug_panic (.../../lib/kernel/debug.c:38)
0xc0021126: thread_yield (..../../threads/thread.c:324)
0xc0023379: sema_up (...../../threads/synch.c:124)
0xc002709c: interrupt_handler (...d/../../devices/ide.c:517)
0xc00223d1: intr_handler (..../threads/interrupt.c:367)
0xc00225f1: intr_entry (threads/intr-stubs.S:38)
0xc0026e3b: issue_pio_command (...d/../../devices/ide.c:414)
0xc00269ab: identify_ata_device (...d/../../devices/ide.c:275)
0xc00265c5: ide_init (...d/../../devices/ide.c:154)
0xc0020357: pintos_init (.../../../threads/init.c:126)
```

thread_yied() can’t be called in intr_context().

Problem3

```
// synch.c
void sema_up (struct semaphore *sema) {
    ...
     if (intr_context ())
        intr_yield_on_return ();
    else
        thread_yield ();
    ...
}
```

pintos -v -k -T 360 –qemu
–filesys-size=2
-p tests/userprog/no-vm/multi-oom
-a multi-oom
–
-q -f run **multi-oom**

Push args

```
// process.c
static void push_arguments (const char* cmdline_tokens[], int argc, void **esp)
{
  ASSERT(argc >= 0);
  int i, len = 0;
  void* argv_addr[argc];
  /* push argvs into **esp */
  for (i = argc - 1; i >= 0; i--) {
    len = strlen(cmdline_tokens[i]) + 1;
    *esp -= len;
    memcpy(*esp, cmdline_tokens[i], len);
    argv_addr[i] = *esp;
  }
  /* word align to 4 bytes. */
  *esp = (void*)((unsigned int)(*esp) & 0xfffffffc);
  /* push null, indicating the end of argv. */
  *esp -= 4;
  *((uint32_t*) *esp) = 0;
  /* push argvs address into **esp. */
  for (i = argc - 1; i >= 0; i--) {
    *esp -= 4;
    *((void**) *esp) = argv_addr[i];
  }
  /* push **argv (addr of stack, esp). */
  *esp -= 4;
  *((void**) *esp) = (*esp + 4);
  /* push argc. */
  *esp -= 4;
  *((int*) *esp) = argc;
  /* push ret addr. */
  *esp -= 4;
  *((int*) *esp) = 0;
}
```

```
static void
start_process(void *pcb_)
{
  ...
  /* Tokenize the command line. */
  char *token;
  char *save_ptr;
  int cnt = 0;
  for (token = strtok_r(file_name, "␣", &save_ptr);
      token != NULL;
      token = strtok_r(NULL, "␣", &save_ptr))
      cmdline_tokens[cnt++] = token;
  /* If load succeeds, push arguments to the stack. */
  if (success) {
    push_arguments(cmdline_tokens, cnt, &if_.esp);
  }
  ...
}
```

Now, PintOS can accept user program's args.

Task 1: Process Termination Messages

Exercise 1.1

Whenever a user process terminates, because it called exit or for any other reason, print the process's name and exit code, formatted as if printed by printf ("%s: exit(%d)\n", ...);.

The name printed should be the full name passed to process_execute(), omitting command-line arguments.

Do not print these messages when a kernel thread that is not a user process terminates, or when the halt system call is invoked. The message is optional when a process fails to load.

Aside from this, don't print any other messages that Pintos as provided doesn't already print. You may find extra messages useful during debugging, but they will confuse the grading scripts and thus lower your score.

Print exit message formatted as "%s: exit(%d)\n" with process name and exit status when process is terminated.

Exercise 1.1

```
1   // syscall.c
2   static void
3   syscall_handler (struct intr_frame *f UNUSED)
4   {
5       ...
6       case SYS_EXIT:
7       {
8         int exitcode;
9         memread_user(f->esp + 4, &exitcode, sizeof(exitcode));
10        sys_exit(exitcode);
11        NOT_REACHED();
12        break;
13      }
14      ...
15  }
16  void
17  sys_exit(int status UNUSED)
18  {
19    printf("%s: exit(%d)\n", thread_current()->name, status);
20
21    thread_exit();
22  }
```

Task 2: Argument Passing

Exercise 2.1

Currently, process_execute() does not **support passing arguments** to new processes. You need to implement it in this task.

Implement this functionality, by extending process_execute() so that instead of simply taking a program file name as its argument, it divides it into words at spaces.

The first word is the program name, the second word is the first argument, and so on. That is, process_execute("grep foo bar") should run grep passing two arguments foo and bar.

Within a command line, multiple spaces are equivalent to a single space, so that process_execute("grep foo bar") is equivalent to our original example.

You can impose a reasonable limit on the length of the command line arguments. For example, you could limit the arguments to those that will fit in a single page (4 kB). (There is an unrelated limit of 128 bytes on command-line arguments that the pintos utility can pass to the kernel.)

Task 3: Accessing User Memory

Exercise 3.1

As part of a system call, the kernel must often access memory through pointers provided by a user program.

The kernel must be very careful about doing so, because the user can pass a null pointer, a pointer to unmapped virtual memory, or a pointer to kernel virtual address space (above PHYS_BASE).

All of these types of invalid pointers must be rejected without harm to the kernel or other running processes, by terminating the offending process and freeing its resources.

Support reading from and writing to user memory for system calls. In either case, you need to make sure not to ”leak” resources. An invalid user pointer will cause a ”page fault”

Basic mem access funcs.

```
1  // syscall.c
2  static int32_t
3  get_user (const uint8_t *uaddr)
4  {
5    /* check that a user pointer `uaddr` points below PHYS_BASE*/
6    if (! ((void*)uaddr < PHYS_BASE))
7      return -1; /**< invalid memory access */
8
9    /* as suggested in the reference manual, see (3.1.5) */
10   int result;
11   asm ("movl␣$1f,␣%0;␣movzbl␣%1,␣%0;␣1:"
12       : "=&a" (result) : "m" (*uaddr));
13   return result;
14 }
15 static bool
16 put_user (uint8_t *udst, uint8_t byte)
17 {
18   /* check that a user pointer `udst` points below PHYS_BASE */
19   if (! ((void*)udst < PHYS_BASE))
20     {
21       return false;
22     }
23   int error_code;
24
25   /* as suggested in the reference manual, see (3.1.5) */
26   asm ("movl␣$1f,␣%0;␣movb␣%b2,␣%1;␣1:"
27       : "=&a" (error_code), "=m" (*udst) : "q" (byte));
28   return error_code != -1;
29 }
```

More user mem access funcs and page_fault.

```
1   // syscall.c
2   /** check a single byte is in user mem. */
3   static void
4   check_user (const uint8_t *uaddr)
5   {
6     /* check uaddr range or segfaults */
7     if(get_user (uaddr) == -1)
8       fail_invalid_access();
9   }
10  static bool
11  validate_user_string(const char *uaddr)
12  {
13    const char *p = uaddr;
14    size_t max_len = 1024;
15    while (max_len--) {
16      int c = get_user((const uint8_t *)p);
17      if (c == -1) return false;
18      if (c == 0) return true;
19      p++;
20    }
21    return false;
22  }
```

```
1   static int
2   fail_invalid_access(void)
3   {
4     if (lock_held_by_current_thread(&filesys_lock))
5       lock_release (&filesys_lock);
6     sys_exit (-1);
7     NOT_REACHED();
8   }
9   // exception.c
10  static void
11  page_fault (struct intr_frame *f)
12  {
13      ...
14      /* (3.1.5) a page fault in the kernel merely
15      sets eax to 0xffffffff and copies its former
16      value into eip */
17     if(!user) {
18        f->eip = (void *) f->eax;
19        f->eax = 0xffffffff;
20        return;
21     }
22     ...
23  }
```

Task 4: System Calls

Overview

```
/** Syscall Functions definition. */
void sys_halt (void);
void sys_exit (int);
pid_t sys_exec (const char *cmdline);
int sys_wait (pid_t pid);
bool sys_create (const char* filename, unsigned initial_size);
bool sys_remove (const char* filename);
int sys_open (const char* file);
void sys_close (int fd);
int sys_filesize (int fd);
void sys_seek(int fd, unsigned position);
unsigned sys_tell(int fd);
void sys_close(int fd);
int sys_read(int fd, void *buffer, unsigned size);
int sys_write(int fd, const void *buffer, unsigned size);
```

Exercise 4.1

Implement the system call handler in userprog/syscall.c.

The skeleton implementation we provide ”handles” system calls by terminating the process.

It will need to retrieve the system call number, then any system call arguments, and carry out appropriate actions.

Exercise 4.2

Implement the following system calls. (13 in all for this lab)

The prototypes listed are those seen by a user program that includes lib/user/syscall.h. (This header, and all others in lib/user, are for use by user programs only.)

System call numbers for each system call are defined in lib/syscall-nr.h

halt exit exec wait create remove open filesize read write seek tell close

Data structs

```
// thread.h
struct thread {
        struct process_control_block *pcb;  /**<  Process Control Block */
    struct list child_list;             /**<  List of children processes of this thread,
                                            each elem is defined by pcb#elem */
    struct list file_descriptors;       /**<  List of file_descriptors the thread contains */
    struct file *executing_file;        /**<  The executable file of associated process. */
    ...
}
// process.h
typedef int pid_t;
#define PID_ERROR         ((pid_t) -1)
#define PID_INITIALIZING  ((pid_t) -2)
struct process_control_block {
    pid_t pid;                                  /**< The pid of process */
    const char* cmdline;                        /**< The command line of this process being executed */
    struct list_elem elem;                      /**< element for thread.child_list */
    bool waiting;                               /**< indicates whether parent process is waiting on this. */
    bool exited;                                /**< indicates whether the process is done (exited). */
    bool orphan;                                /**< indicates whether the parent process has terminated before. */
    int32_t exitcode;                           /**< the exit code passed from exit(), when exited = true */
    /* Synchronization */
    struct semaphore sema_initialization;/**< the semaphore used between start_process() and process_execute()*/
    struct semaphore sema_wait;         /**< the semaphore used for wait() : parent blocks until child exits*/
};
/** File description. */
struct file_desc
{
    int id;                                     /**< Identify file. */
    struct list_elem elem;                      /**< element for thread.file_descriptors */
    struct file* file;                          /**< file object. */
};
// syscall.c
struct lock filesys_lock;
```

Initialize

```
1   // thread.c
2   static void init_thread (struct thread *t, const char *name, int priority) {
3   ...
4   #ifdef USERPROG
5     list_init(&t->child_list);
6     t->pcb = NULL;
7     list_init(&t->file_descriptors);
8     t->executing_file = NULL;
9   #endif
10  }
11  void syscall_init (void) {
12    lock_init (&filesys_lock);
13    intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");
14  }
```

Initialize

```
// process.c
pid_t process_execute(const char *file_name) {
...
pcb = palloc_get_page(0);
  proc_name = strtok_r(proc_name, "␣", &save_ptr);
  if (pcb == NULL) {
    palloc_free_page(proc_name);
    palloc_free_page(cmd_all);
    return TID_ERROR;
  }

  /* Initial PCB. */
  pcb->pid = PID_INITIALIZING;
  pcb->cmdline = cmd_all;
  pcb->waiting = false;
  pcb->exited = false;
  pcb->orphan = false;
  pcb->exitcode = -1; /**< undefined */

  sema_init(&pcb->sema_initialization, 0);
  sema_init(&pcb->sema_wait, 0);

  /* Create a new thread to execute PROC_CMD. */
  tid = thread_create(proc_name, PRI_DEFAULT,
  start_process, pcb);

  if (tid == TID_ERROR) {
    palloc_free_page(pcb);
    palloc_free_page(proc_name);
    palloc_free_page(cmd_all);
    return TID_ERROR;
  }
```

```

  /* Wait until initialization inside
  start_process() is complete. */
  sema_down(&pcb->sema_initialization);

  if(cmd_all) {
    palloc_free_page (cmd_all);
  }

  /* Process successfully created, maintain child
  process list. */
  if(pcb->pid >= 0) {
    list_push_back (&(thread_current()->child_list),
    &(pcb->elem));
  }
  /* Free proc_name since it's no longer needed. */
  palloc_free_page(proc_name);

  return pcb->pid;
}
```

sys_halt

```
1   // syscall.c
2   static void syscall_handler (struct intr_frame *f UNUSED)
3   {
4       ...
5       case SYS_HALT:
6           {
7             sys_halt();
8             NOT_REACHED();
9             break;
10          }
11      ...
12  }
13
14  void
15  sys_halt(void)
16  {
17    shutdown_power_off();
18  }
```

sys_exit

```
1  // syscall.c
2  static void syscall_handler (struct intr_frame *f UNUSED)
3  {
4      ...
5       case SYS_EXIT:
6          {
7            int exitcode;
8            memread_user(f->esp + 4, &exitcode, sizeof(exitcode));
9            sys_exit(exitcode);
10           NOT_REACHED();
11           break;
12         }
13     ...
14 }
15
16 void
17 sys_exit(int status UNUSED)
18 {
19   printf("%s: exit(%d)\n", thread_current()->name, status);
20
21   /* The process exits.
22     wake up the parent process (if it was sleeping) using semaphore,
23     and pass the return code. */
24   struct process_control_block *pcb = thread_current()->pcb;
25   if(pcb != NULL)
26     {
27       pcb->exited = true;
28       pcb->exitcode = status;
29     }
30
31   thread_exit();
32 }
33 // exception.c change thread_exit () to sys_exit ()
```

sys_exit

```
// thread.c
void thread_exit (void) {
  ...
  struct thread *curr = thread_current();
  /* release all locks */
  struct list_elem *e;
  for (e = list_begin (&curr->locks_holding);
  e != list_end (&curr->locks_holding);
  e = list_next (e)) {
    struct lock *lock =
    list_entry(e, struct lock, elem);
    lock_release(lock);
  }
  ...
}
// process.c
void process_exit (void) {
    ...
  while (!list_empty(fdlist))
    {
      struct list_elem *e = list_pop_front (fdlist);
      struct file_desc *desc = list_entry(e,
      struct file_desc, elem);
      file_close(desc->file);
      palloc_free_page(desc); /**< see sys_open(). */
    }
  struct list *child_list = &cur->child_list;
  while (!list_empty(child_list))
    {
      struct list_elem *e = list_pop_front
      (child_list);
      struct process_control_block *pcb;
      pcb = list_entry(e, struct
      process_control_block, elem);
      if (pcb->exited == true)
        {
          /* pcb can freed when it is already
          terminated. */
          palloc_free_page (pcb);
        }
       else
        {
          /* the child process becomes an orphan.
           do not free pcb yet, postpone until the
           child terminates. */
          pcb->orphan = true;
        }
    }

  /* Release file for the executable */
  if(cur->executing_file)
    {
      file_allow_write(cur->executing_file);
      file_close(cur->executing_file);
    }

  /* Unblock the waiting parent process, if any,
  from wait().now its resource (pcb on page, etc.)
  can be freed. */
  bool cur_pcb_orphan = cur->pcb->orphan;
  sema_up (&cur->pcb->sema_wait);

  if (cur_pcb_orphan)
    {
      palloc_free_page (& cur->pcb);
    }
    ...
}
```

sys_exec

```
1   // syscall.c
2   static void syscall_handler (struct intr_frame *f UNUSED)
3   {
4       ...
5         case SYS_EXEC:
6           {
7             void* cmdline;
8             memread_user(f->esp + 4, &cmdline, sizeof(cmdline));
9             int return_code = sys_exec((const char*) cmdline);
10            f->eax = (uint32_t) return_code;
11            break;
12          }
13      ...
14  }
15  pid_t sys_exec(const char *cmdline) {
16    _DEBUG_PRINTF("[DEBUG] Exec : %s\n", cmdline);
17
18    /* check the cmdline is in user mem. */
19    if (!validate_user_string (cmdline))
20      {
21        fail_invalid_access ();
22      }
23
24    /* load() uses filesystem. */
25    lock_acquire (&filesys_lock);
26    pid_t pid = process_execute(cmdline);
27    lock_release (&filesys_lock);
28    return pid;
29  }
```

sys_wait

```
1   // syscall.c
2   static void syscall_handler (struct intr_frame *f UNUSED)
3   {
4       ...
5         case SYS_WAIT:
6           {
7             pid_t pid;
8             memread_user(f->esp + 4, &pid, sizeof(pid_t));
9             int ret = sys_wait(pid);
10            f->eax = (uint32_t) ret;
11            break;
12          }
13      ...
14  }
15  int
16  sys_wait(pid_t pid)
17  {
18    _DEBUG_PRINTF ("[DEBUG] Wait : %d\n", pid);
19    return process_wait(pid);
20  }
```

sys_wait

```
int process_wait (tid_t child_tid UNUSED) {
  struct thread *t = thread_current ();
  struct list *child_list = &(t->child_list);

  /* lookup the process with tid equals
  'child_tid' from 'child_list'. */
  struct process_control_block *child_pcb = NULL;
  struct list_elem *it = NULL;
  if (!list_empty(child_list))
    {
      for (it = list_front(child_list); it !=
      list_end(child_list); it = list_next(it))
        {
          struct process_control_block *pcb = list_entry (
              it, struct process_control_block, elem);

          if(pcb->pid == child_tid)
            { /* OK, the direct child found. */
              child_pcb = pcb;
              break;
            }
        }
    }

  /* if child process is not found, return -1
  immediately. */
  if (child_pcb == NULL)
    {
      _DEBUG_PRINTF("[DEBUG] wait(): child not
found, pid = %d\n", child_tid);
      return -1;
    }
```

```
  if (child_pcb->waiting)
      {
        /* already waiting (the parent already
        called wait on child's pid). */
        _DEBUG_PRINTF("[DEBUG] wait(): child found,
pid = %d, but it is already waiting\n",
        child_tid);
        return -1; /**< a process may wait for any
        fixed child at most once. */
      }
    else
      {
        child_pcb->waiting = true;
      }
    if (! child_pcb->exited)
      {
        sema_down(& (child_pcb->sema_wait));
      }
    ASSERT (child_pcb->exited == true);
    /* remove from child_list. */
    ASSERT (it != NULL);
    list_remove (it);

    /* return the exit code of the child process. */
    int retcode = child_pcb->exitcode;

    /* Now the pcb object of the child process can
    be finally freed. (in this context, the child
    process is guaranteed to have been exited). */
    palloc_free_page(child_pcb);

    return retcode;
}
```

sys_create

```
// syscall.c
static void syscall_handler (struct intr_frame *f UNUSED)
{
    ...
    case SYS_CREATE:
        {
          const char* filename;
          unsigned initial_size;
          bool return_code;
          memread_user(f->esp + 4, &filename, sizeof(filename));
          memread_user(f->esp + 8, &initial_size, sizeof(initial_size));
          return_code = sys_create(filename, initial_size);
          f->eax = return_code;
          break;
        }
    ...
}
bool
sys_create(const char* filename, unsigned initial_size)
{
  bool return_code;
  /* memory validation */
  check_user((const uint8_t*) filename);

  lock_acquire (&filesys_lock);
  return_code = filesys_create(filename, initial_size);
  lock_release (&filesys_lock);
  return return_code;
}
```

sys_remove

```
1   // syscall.c
2   static void syscall_handler (struct intr_frame *f UNUSED)
3   {
4       ...
5       case SYS_REMOVE:
6           {
7             const char* filename;
8             bool return_code;
9             memread_user(f->esp + 4, &filename, sizeof(filename));
10            return_code = sys_remove(filename);
11            f->eax = return_code;
12            break;
13          }
14      ...
15  }
16  bool
17  sys_remove(const char* filename)
18  {
19    bool return_code;
20    /* memory validation */
21    check_user((const uint8_t*) filename);
22
23    lock_acquire (&filesys_lock);
24    return_code = filesys_remove(filename);
25    lock_release (&filesys_lock);
26    return return_code;
27  }
```

sys_open

```
// syscall.c
static void syscall_handler (struct intr_frame *f )
{
    ...
     case SYS_OPEN:
        {
          const char* filename;
          int return_code;
          memread_user(f->esp + 4, &filename,
          sizeof(filename));
          return_code = sys_open(filename);
          f->eax = return_code;
          break;
        }
    ...
}
int
sys_open(const char* file)
{
  /* memory validation */
  check_user((const uint8_t*) file);

  struct file* file_opened;
  struct file_desc* fd = palloc_get_page(0);

  if (!fd) {
    return -1;
  }
```

```

  lock_acquire (&filesys_lock);
  file_opened = filesys_open(file);
  if (!file_opened) {
    palloc_free_page (fd);
    lock_release (&filesys_lock);
    return -1;
  }

  fd->file = file_opened;

  struct list* fd_list = &thread_current ()->file_descript
  if (list_empty(fd_list))
    {
      /* 0, 1, 2 are reserved for stdin, stdout, stderr. *
      fd->id = 3;
    }
  else
    {
      fd->id = (list_entry(list_back(fd_list), struct file
    }
  list_push_back(fd_list, &(fd->elem));
  lock_release (&filesys_lock);

  return fd->id;
}
```

sys_filesize

```
// syscall.c
static void syscall_handler (struct intr_frame *f UNUSED)
{
    ...
    case SYS_FILESIZE:
        {
          int fd, return_code;
          memread_user(f->esp + 4, &fd, sizeof(fd));
          return_code = sys_filesize(fd);
          f->eax = return_code;
          break;
        }
    ...
}
int
sys_filesize(int fd)
{
  lock_acquire (&filesys_lock);

  struct file_desc* file_d;

  file_d = find_file_desc(thread_current(), fd);

  if(file_d == NULL)
  {
    lock_release (&filesys_lock);
    return -1;
  }

  int ret = file_length(file_d->file);
  lock_release (&filesys_lock);
  return ret;
}
```

sys_read

```
// syscall.c
static void syscall_handler (struct intr_frame *f )
{
    ...
  case SYS_READ:
    {
      int fd, return_code;
      void *buffer;
      unsigned size;
      memread_user(f->esp + 4, &fd, sizeof(fd));
      memread_user(f->esp + 8, &buffer, sizeof(buffer));
      memread_user(f->esp + 12, &size, sizeof(size));
      return_code = sys_read(fd, buffer, size);
      f->eax = (uint32_t) return_code;
      break;
    }
    ...
}
int
sys_read(int fd, void *buffer, unsigned size)
{
   /* memory validation : [buffer+0, buffer+size)
   should be all valid. */
   check_user((const uint8_t*) buffer);
   check_user((const uint8_t*) buffer + size - 1);
```

```
    lock_acquire (&filesys_lock);
    int ret;

  if(fd == 0)
    { /**< stdin */
      unsigned i;
      for(i = 0; i < size; ++i)
        {
          if(! put_user(buffer + i, input_getc()) )
            {
              lock_release (&filesys_lock);
              sys_exit(-1); /**< segfault */
            }
        }
      ret = size;
    }
  else
    {
      /* read from file */
      struct file_desc* file_d =
      find_file_desc(thread_current(), fd);

      if(file_d && file_d->file)
        {
          ret = file_read(file_d->file, buffer, size);
        }
      else /**< no such file or can't open */
        ret = -1;
    }

  lock_release (&filesys_lock);
  return ret;
}
```

sys_write

```
// syscall.c
static void syscall_handler (struct intr_frame *f )
{
    ...
  case SYS_WRITE:
  {
    int fd, return_code;
    const void *buffer;
    unsigned size;
    memread_user(f->esp + 4, &fd, sizeof(fd));
    memread_user(f->esp + 8, &buffer, sizeof(buffer));
    memread_user(f->esp + 12, &size, sizeof(size));
    return_code = sys_write(fd, buffer, size);
    f->eax = (uint32_t) return_code;
    break;
  }
    ...
}
```

```
int
sys_write(int fd, const void *buffer, unsigned size) {
  /* memory validation : [buffer+0, buffer+size)
  should be all valid */
  check_user((const uint8_t*) buffer);
  check_user((const uint8_t*) buffer + size - 1);
  lock_acquire (&filesys_lock);
  int ret;
  if(fd == 1)
    { /**< write to stdout */
      putbuf(buffer, size);
      ret = size;
    }
  else
    {
      /* write into file */
      struct file_desc* file_d =
      find_file_desc(thread_current(), fd);

      if(file_d && file_d->file)
        {
          ret = file_write(file_d->file, buffer, size);
        }
      else /**< no such file or can't open */
        ret = -1;
    }

  lock_release (&filesys_lock);
  return ret;
}
```

sys_seek

```
1   // syscall.c
2   static void syscall_handler (struct intr_frame *f UNUSED)
3   {
4       ...
5       case SYS_SEEK:
6       {
7         int fd;
8         unsigned position;
9         memread_user(f->esp + 4, &fd, sizeof(fd));
10        memread_user(f->esp + 8, &position, sizeof(position));
11        sys_seek(fd, position);
12        break;
13      }
14      ...
15  }
16  void sys_seek(int fd, unsigned position)
17  {
18    lock_acquire (&filesys_lock);
19    struct file_desc* file_d = find_file_desc(thread_current(), fd);
20
21    if(file_d && file_d->file)
22      {
23        file_seek(file_d->file, position);
24      }
25    else
26      return; // TODO need sys_exit?
27    lock_release (&filesys_lock);
28  }
```

sys_tell

```
1   // syscall.c
2   static void syscall_handler (struct intr_frame *f UNUSED)
3   {
4       ...
5    case SYS_TELL:
6       {
7         int fd;
8         unsigned return_code;
9         memread_user(f->esp + 4, &fd, sizeof(fd));
10        return_code = sys_tell(fd);
11        f->eax = (uint32_t) return_code;
12        break;
13      }
14      ...
15  }
16  unsigned
17  sys_tell(int fd)
18  {
19    lock_acquire (&filesys_lock);
20    struct file_desc* file_d = find_file_desc(thread_current(), fd);
21    unsigned ret;
22    if(file_d && file_d->file)
23      {
24        ret = file_tell(file_d->file);
25      }
26    else
27      ret = -1; // TODO need sys_exit?
28
29    lock_release (&filesys_lock);
30     return ret;
```

sys_close

```
1   // syscall.c
2   static void syscall_handler (struct intr_frame *f UNUSED)
3   {
4       ...
5       case SYS_CLOSE:
6       {
7         int fd;
8         memread_user(f->esp + 4, &fd, sizeof(fd));
9         sys_close(fd);
10        break;
11      }
12      ...
13  }
14  void
15  sys_close(int fd)
16  {
17    lock_acquire (&filesys_lock);
18    struct file_desc* file_d = find_file_desc(thread_current(), fd);
19
20    if(file_d && file_d->file)
21      {
22        file_close(file_d->file);
23        list_remove(&(file_d->elem));
24        palloc_free_page(file_d);
25      }
26    lock_release (&filesys_lock);
27  }
```

Finish touchs

```
1   // syscall.c
2   static void syscall_handler (struct intr_frame *f UNUSED)
3   {
4       ...
5        /* unhandled case */
6         default:
7           printf("[ERROR] system call %d is unimplemented!\n", syscall_number);
8           sys_exit(-1);
9           break;
10      }
11  }
```

Task 5: Denying Writes to Executables

Exercise 5.1

Add code to deny writes to files in use as executables.

Many OSes do this because of the unpredictable results if a process tried to run code that was in the midst of being changed on disk.

This is especially important once virtual memory is implemented in project 3, but it can't hurt even now. You can use file_deny_write() to prevent writes to an open file.

Calling file_allow_write() on the file will re-enable them (unless the file is denied writes by another opener).

Closing a file will also re-enable writes. Thus, to deny writes to a process's executable, you must keep it open as long as the process is still running.

Exercise 5.1

```
1   // sprocess.c
2   bool
3   load (const char *file_name, void (**eip) (void), void **esp)
4   {
5    /* Deny writes to executables. */
6     file_deny_write (file);
7     thread_current()->executing_file = file;
8     success = true;
9   done:
10    /* We arrive here whether the load is successful or not. */
11
12    /* do not close file here, postpone until it terminates. */
13    return success;
14  }
15  void
16  process_exit (void)
17  {
18    ...
19    /* Release file for the executable */
20    if(cur->executing_file)
21      {
22        file_allow_write(cur->executing_file);
23        file_close(cur->executing_file);
24      }
25      ...
26  }
```

Finishing touchs

Some terrible problems

```
FAIL tests/userprog/exec-bound-3
run: TIMEOUT after 61 seconds of wall-clock time - load average: 0.77, 0.29, 0.10
pintos -v -k -T 60 --qemu  --filesys-size=2 -p tests/userprog/exec-multiple -a exec-multip
le -p tests/userprog/child-simple -a child-simple -- -q  -f run exec-multiple < /dev/null
2> tests/userprog/exec-multiple.errors > tests/userprog/exec-multiple.output
perl -I../.. ../../tests/userprog/exec-multiple.ck tests/userprog/exec-multiple tests/user
prog/exec-multiple.result
```

```
FAIL tests/userprog/no-vm/multi-oom
run: after run 1/10, expected depth 52, actual depth 39.: FAILED
pintos -v -k -T 60 --qemu  --filesys-size=2 -p tests/filesys/base/lg-create -a lg-create -
- -q  -f run lg-create < /dev/null 2> tests/filesys/base/lg-create.errors > tests/filesys/
base/lg-create.output
perl -I../.. ../../tests/filesys/base/lg-create.ck tests/filesys/base/lg-create tests/file
sys/base/lg-create.result
```

Their fixes are **validate_user_string ()** and **timely release memory** to prevent memory leaks.

results

```
pass tests/filesys/base/sm-random
pass tests/filesys/base/sm-seq-block
pass tests/filesys/base/sm-seq-random
pass tests/filesys/base/syn-read
pass tests/filesys/base/syn-remove
pass tests/filesys/base/syn-write
All 80 tests passed.
```

```
azureuser@jp:~/pintos_lab/pintos$ git diff --stat 2389331af4eea8753b938b4092ef4227e093d76
 b3bbf2988f18d3782b7e0f6d94cfa269ae6ec890
 src/threads/synch.c        |   5 +-
 src/threads/thread.c       |  19 ++
 src/threads/thread.h       |   6 +-
 src/userprog/exception.c   |  13 +-
 src/userprog/process.c     | 335 ++++++++++++++++++++++++++++++-----
 src/userprog/process.h     |  33 +++-
 src/userprog/syscall.c     | 537 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++-
 src/userprog/syscall.h     |  18 ++
 8 files changed, 914 insertions(+), 52 deletions(-)
```

Thank you!
Questions?