

# Lab2: User Programs

## TA Session

Prepare1 : Open up Pintos in your IDE/Text Editor

Prepare2 : Launch Pintos container



TA : zhongyinmin  
Email : [zhongyinmin@pku.edu.cn](mailto:zhongyinmin@pku.edu.cn)  
Github : PKUFlyingPig

# Some announcements:

- Lab 2 Code will due two weeks later
- No grace day
- Start early, Start early, Start early
- You can complete Lab2 from a clean codebase




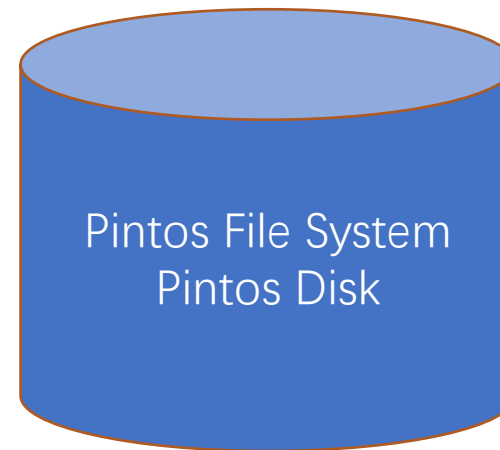
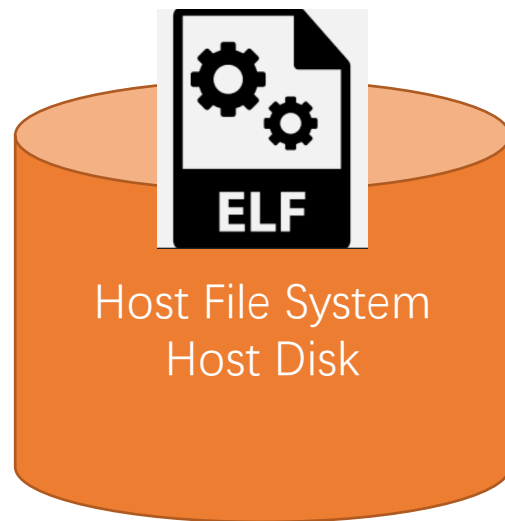
# Contents

- Pintos Disk and File System
- System Call
- Interrupt Handling
- Lab2 tasks and suggestions



# Where are the User Programs?

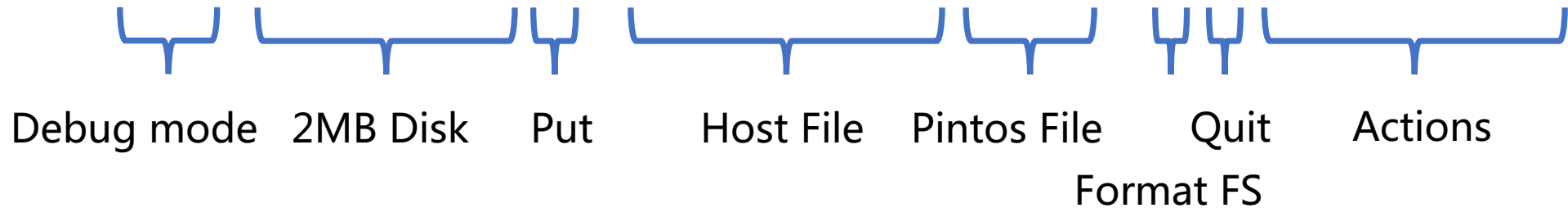
- Source files are under `/src/examples/` directory
- Run ``make`` under `/src/examples/`  ELF files



# Debug example:

- Run `make && cd build` under `/src/userprog/` directory

- `pintos --gdb --fileysys-size=2 -p ../../examples/echo -a echo -- -f -q run 'echo iloveos'`



- Details on Lab Document

# Action: run 'echo iloveos'

pintos\_init():

```
/* Break command line into arguments and parse options. */  
argv = read_command_line ();  
argv = parse_options (argv);
```

```
printf ("Boot complete.\n");  
  
if (*argv != NULL) {  
    /* Run actions specified on kernel command line. */  
    run_actions (argv);  
} else {  
    // TODO: no command line passed to kernel. Run interactively  
}
```

```
(gdb) x/6s *argv  
0xc0007d3e:  "-f"  
0xc0007d41:  "-q"  
0xc0007d44:  "extract"  
0xc0007d4c:  "run"  
0xc0007d50:  "echo iloveos"  
0xc0007d5d:  ""
```

```
(gdb) x/4s *argv  
0xc0007d44:  "extract"  
0xc0007d4c:  "run"  
0xc0007d50:  "echo iloveos"  
0xc0007d5d:  ""
```



Lab2 User Program



Lab0 Shell

# Action: extract run 'echo iloveos'

run\_actions():

```
/* An action. */
struct action
{
    char *name;
    int argc;
    void (*function) (char **argv);
};
```

```
/* Table of supported actions. */
static const struct action actions[] =
{
    {"run", 2, run_task},
#ifdef FILESYS
    {"ls", 1, fsutil_ls},
    {"cat", 2, fsutil_cat},
    {"rm", 2, fsutil_rm},
    {"extract", 1, fsutil_extract},
    {"append", 2, fsutil_append},
#endif
    {NULL, 0, NULL},
};
```

# Action: extract run 'echo iloveos'

run\_task():

```
/* 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);
}
```

Q1: What is ARGV[0] ?

Q2: What does run\_test() do ?

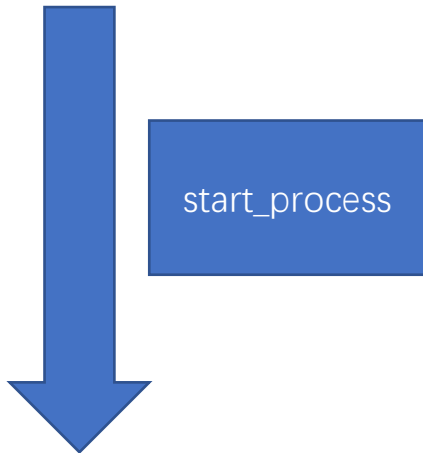


# Action: run 'echo iloveos'

process\_execute():

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

Main Thread:



Ready List:



# Main Thread

run\_task():

```
/* 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);
}
```

Return Immediately !!



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

# start\_process:

```
/* 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;
success = load (file_name, &if_.eip, &if_.esp);
```

- Load the ELF file from Disk into memory
- We are still in the kernel !!
- Initialize interrupt frame (eip, esp, segment registers, eflags)
- Start the user process by simulating a return from an interrupt

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

<https://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html>

# Kernel Mappings

# load:

## struct thread:

```

#ifdef USERPROG
    /* Owned by userprog/process.c. */
    uint32_t *pagedir; /* Page directory. */
#endif

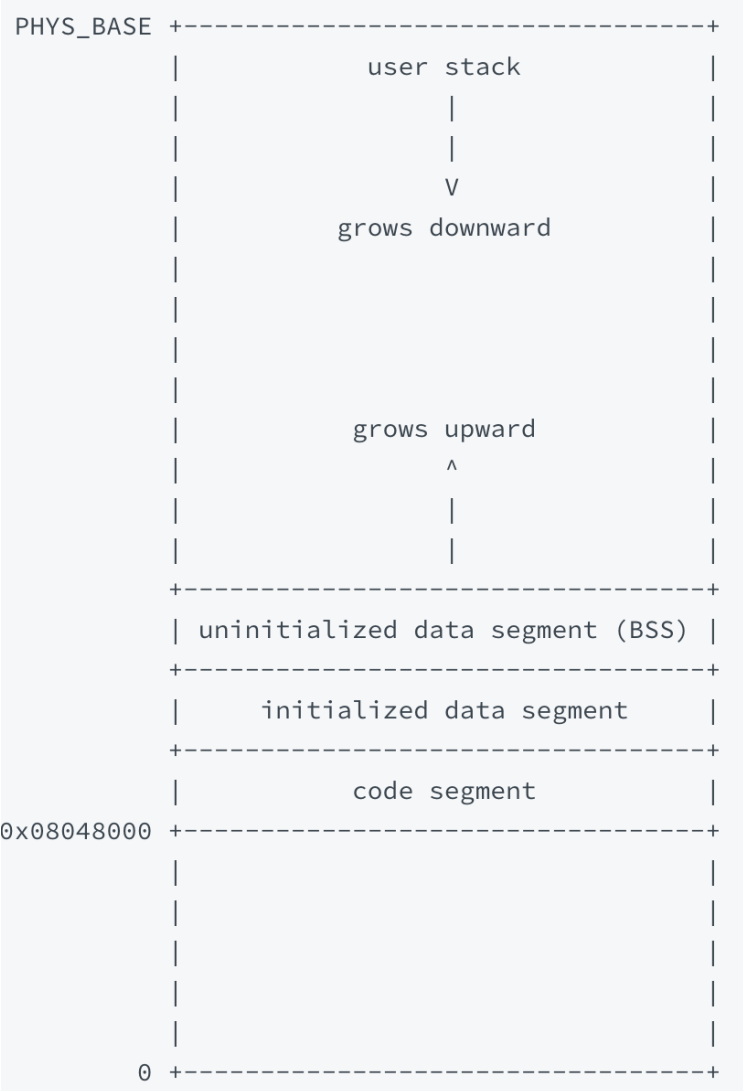
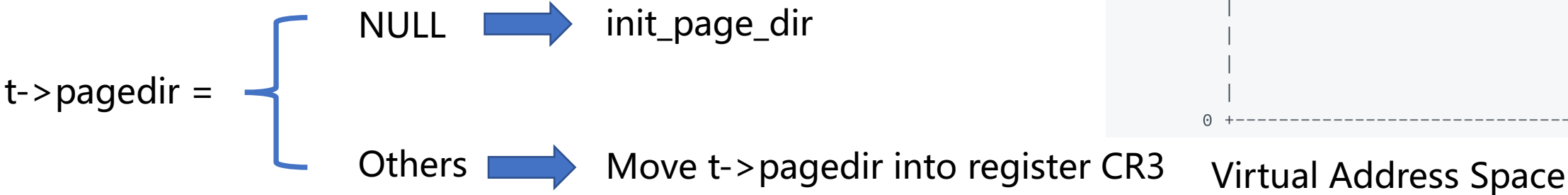
```

## Create a new page directory:

```

/* Allocate and activate page directory. */
t->pagedir = pagedir_create ();
if (t->pagedir == NULL)
    goto done;
process_activate ();

```



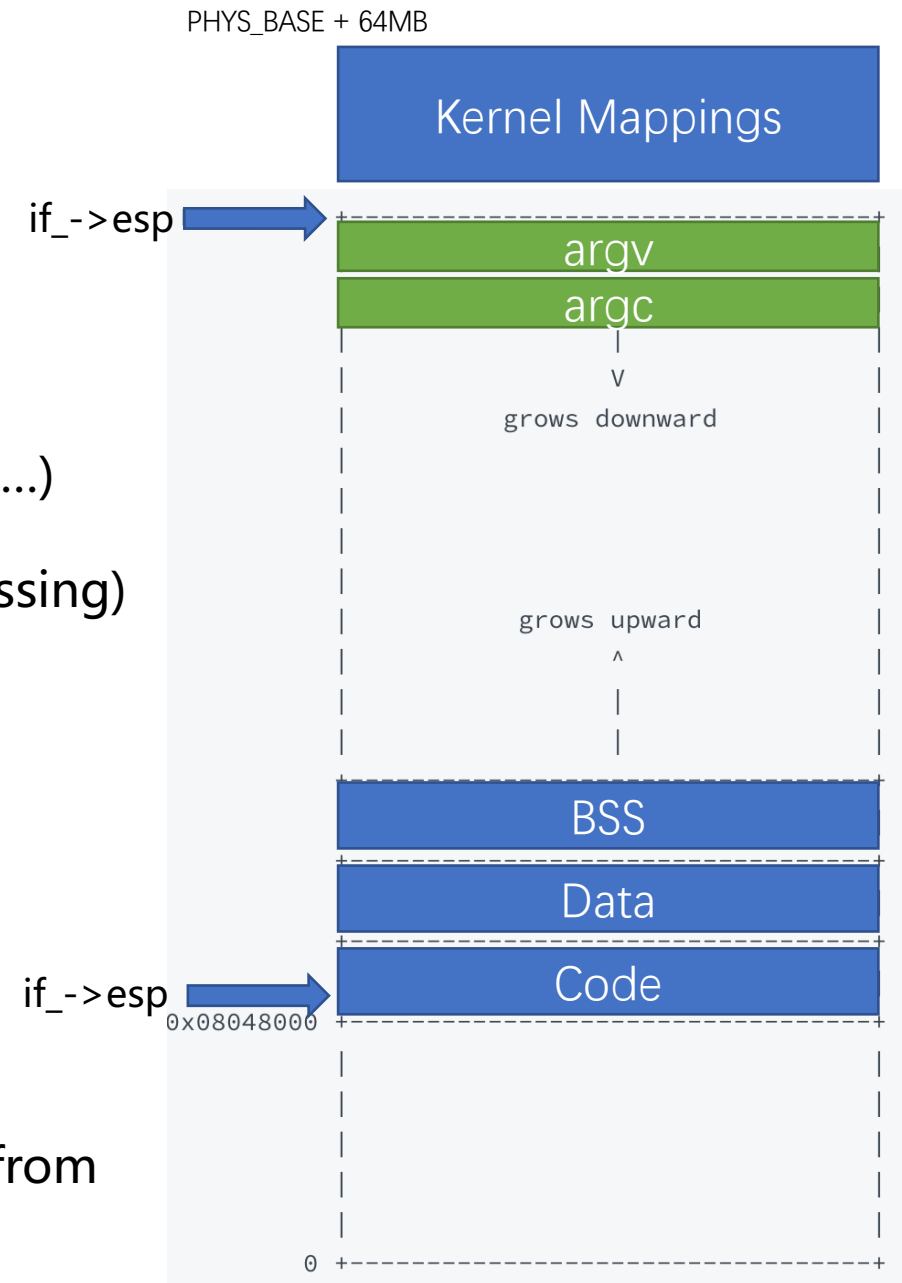
# load:

- Read and verify ELF executable header
- Read ELF program header and load segments (code, data .....
- Set up `if_>stack` (You will fix this in Exercise2: Argument Passing)
- Set up `if_>eip` with the entry point in executable header

`/src/lib/user/entry.c:`

```
void
_start (int argc, char *argv[])
{
    exit (main (argc, argv));
}
```

- After loading, start the user process by simulating a return from an interrupt with interrupt frame `if_`



Virtual Address Space

# Wow, your process is running in User Space!

- But, we want system call support !!

/src/lib/user/syscall.h:

```
void
halt (void)
{
    syscall0 (SYS_HALT);
    NOT_REACHED ();
}
```

/src/lib/user/syscall.c:

```
/* Invokes syscall NUMBER, passing no arguments, and returns the
   return value as an `int'. */
#define syscall0(NUMBER)
({
    int retval;
    asm volatile
        ("pushl %[number]; int $0x30; addl $4, %%esp"
         : "=a" (retval)
         : [number] "i" (NUMBER)
         : "memory");
    retval;
})
```

# Wow, your process is running in User Space!

- But, we want system call support !!

80x86 Calling Convention !!

/src/lib/user/syscall.h:

```
void
exit (int status)
{
    syscall1 (SYS_EXIT, status);
    NOT_REACHED ();
}
```

/src/lib/user/syscall.c:

```
/* Invokes syscall NUMBER, passing argument ARG0, and returns the
   return value as an `int'. */
#define syscall1(NUMBER, ARG0)
({
    int retval;
    asm volatile
        ("pushl %[arg0]; pushl %[number]; int $0x30; addl $8, %%esp" \
         : "=a" (retval) \
         : [number] "i" (NUMBER), \
         [arg0] "g" (ARG0) \
         : "memory");
    retval;
})
```

# System Call Numbers:

/src/lib/syscall-nr.h

```
/* System call numbers. */
enum
{
    /* Projects 2 and later. */
    SYS_HALT,          /* Halt the operating system. */
    SYS_EXIT,         /* Terminate this process. */
    SYS_EXEC,         /* Start another process. */
    SYS_WAIT,         /* Wait for a child process to die. */
    SYS_CREATE,       /* Create a file. */
    SYS_REMOVE,       /* Delete a file. */
    SYS_OPEN,         /* Open a file. */
    SYS_FILESIZE,     /* Obtain a file's size. */
    SYS_READ,         /* Read from a file. */
    SYS_WRITE,        /* Write to a file. */
    SYS_SEEK,         /* Change position in a file. */
    SYS_TELL,         /* Report current position in a file. */
    SYS_CLOSE,        /* Close a file. */

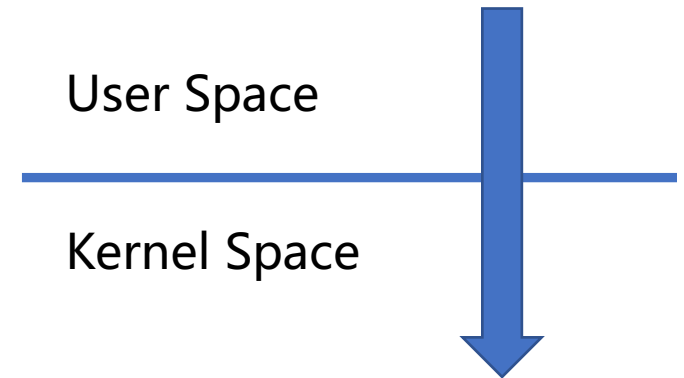
    /* Project 3 and optionally project 4. */
    SYS_MMAP,         /* Map a file into memory. */
    SYS_MUNMAP,       /* Remove a memory mapping. */

    /* Project 4 only. */
    SYS_CHDIR,        /* Change the current directory. */
    SYS_MKDIR,        /* Create a directory. */
    SYS_READDIR,      /* Reads a directory entry. */
    SYS_ISDIR,        /* Tests if a fd represents a directory. */
    SYS_INUMBER       /* Returns the inode number for a fd. */
};
```



Now, all the magic is behind `int 0x30`

```
/* Invokes syscall NUMBER, passing no arguments, and returns the
   return value as an `int'. */
#define syscall0(NUMBER)
    ({
        int retval;
        asm volatile
            ("pushl %[number]; int $0x30; addl $4, %%esp"
             : "=a" (retval)
             : [number] "i" (NUMBER)
             : "memory");
        retval;
    })
```



# Interrupt Handler

- save the context of the interrupted

/src/threads/interrupt.c:

```
void  
intr_handler (struct intr_frame *frame)
```

```
/* Invoke the interrupt's handler. */  
handler = intr_handlers[frame->vec_no];  
if (handler != NULL)  
    handler (frame);
```

/src/user

```
void  
syscall_init  
{  
    intr_regist  
}
```

```
static void  
syscall_hand  
{  
    printf ("s  
    thread_exit  
}
```

Implemen

```
/* Interrupt stack frame. */  
struct intr_frame  
{  
    /* Pushed by intr_entry in intr-stubs.S.  
       These are the interrupted task's saved registers. */  
    uint32_t edi;           /* Saved EDI. */  
    uint32_t esi;           /* Saved ESI. */  
    uint32_t ebp;           /* Saved EBP. */  
    uint32_t esp_dummy;     /* Not used. */  
    uint32_t ebx;           /* Saved EBX. */  
    uint32_t edx;           /* Saved EDX. */  
    uint32_t ecx;           /* Saved ECX. */  
    uint32_t eax;           /* Saved EAX. */  
    uint16_t gs, :16;       /* Saved GS segment register. */  
    uint16_t fs, :16;       /* Saved FS segment register. */  
    uint16_t es, :16;       /* Saved ES segment register. */  
    uint16_t ds, :16;       /* Saved DS segment register. */  
  
    /* Pushed by intrNN_stub in intr-stubs.S. */  
    uint32_t vec_no;        /* Interrupt vector number. */  
  
    /* Sometimes pushed by the CPU,  
       otherwise for consistency pushed as 0 by intrNN_stub.  
       The CPU puts it just under `eip', but we move it here. */  
    uint32_t error_code;    /* Error code. */  
  
    /* Pushed by intrNN_stub in intr-stubs.S.  
       This frame pointer eases interpretation of backtraces. */  
    void *frame_pointer;    /* Saved EBP (frame pointer). */  
  
    /* Pushed by the CPU.  
       These are the interrupted task's saved registers. */  
    void (*eip) (void);     /* Next instruction to execute. */  
    uint16_t cs, :16;       /* Code segment for eip. */  
    uint32_t eflags;        /* Saved CPU flags. */  
    void *esp;              /* Saved stack pointer. */  
    uint16_t ss, :16;       /* Data segment for esp. */  
};
```

# Contents

- Pintos Disk and File System
- System Call
- Interrupt Handling
- Lab2 tasks and suggestions



# Some useful tips:

- Pintos exec == Unix fork + exec
- You can use malloc in kernel (`#include "threads/malloc.h"` )
- Useful GDB command: `loadusersymbols`
- Reference to [xv6 implementation](#)
- multi-oom testcase will take some time, be patient

# Suggested Order of Implementation:

## Step1: Argument Passing

/src/tests/main.c:

```
int
main (int argc UNUSED, char *argv[])
{
  test_name = argv[0];

  msg ("begin");
  random_init (0);
  test_main ();
  msg ("end");
  return 0;
}
```

Page Fault !!

## Why stack ?

- Set up the stack after loading
- Argument Passing details in Lab Doc
- Pass all the args-xxx tests

# Suggested Order of Implementation:

## Step2: Halt System Call

- Argument Passing
- System Call Infrastructure

# Suggested Order of Implementation:

## Step3: Some temporal workaround

- The exit system call (barely work is fine)
- The write system call for writing to fd 1, the system console
- change process\_wait() to an infinite loop (one that waits forever)

# Suggested Order of Implementation:

## Step4: Accessing User Memory

- User programs will pass arguments (char\*, int, unsigned) into kernel
- These arguments are on the user stack (esp is saved in intr\_frame)
- Ensure the address validity (in user page table)
- Avoid repeating code !!
- Two implementation suggestion (in Lab Doc)



# Suggested Order of Implementation:

## Step4: Process Control System Call

- exit, exec, wait
- Design all at first, they may share some data structures

## Step5: FS System Call

- No need to understand file system implementation
- Read the interfaces in `/src/filesys/file.c`, `/src/filesys/filesys.c`
- Pass all tests but rox-simple, rox-child, rox-multichild

# Suggested Order of Implementation:

## Step7: Denying Writes to Executables

- Why?
- Close a file will re-enable writes
- Keep the executable file open during execution

# Suggested Order of Implementation:

Step8: Cheers !!!

```
TOTAL TESTING SCORE: 100.0%  
ALL TESTED PASSED -- PERFECT SCORE
```

```
-----  
SUMMARY BY TEST SET
```

Test Set	Pts	Max	% Ttl	% Max
tests/userprog/Rubric.functionality	108	108	35.0%	35.0%
tests/userprog/Rubric.robustness	88	88	25.0%	25.0%
tests/userprog/no-vm/Rubric	1	1	10.0%	10.0%
tests/filesys/base/Rubric	30	30	30.0%	30.0%
Total			100.0%	100.0%



**AKUOS**

*Welcome to the World of Operating System*

Enjoy Your Pintos Journey ~ ~

Any Problem ?