EC440 Pintos Project Lab2 Overview

William Wang (xwill@bu.edu)

Administrivia

- **Lab 2 deadline: Friday 11/07 11:59 pm**
  - Estimated time: 50~60 hours per group
- **Submission**
  - create a “lab2-handin” branch

Outline

- **User Programs In Pintos**
- **An Overview of Project 2 Requirements**
- **Getting start**
- **Tips**

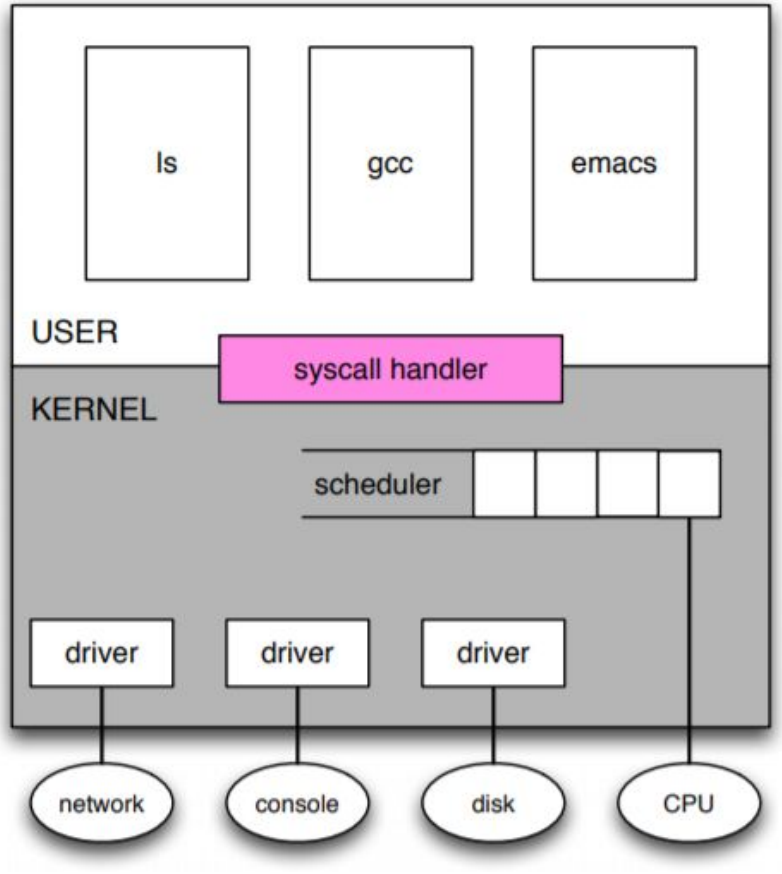
Overview

- **Project 2: Userprog**
  - Allow user programs to run on top of pintos

- **Lab 2 requires good understanding of**
  - How user programs run in general
  - Distinctions between user and kernel virtual memory
  - System call infrastructure
  - File system interface

User Process & Syscalls

- **Syscalls provide the interface between user process and OS**



How to Run A User Program In Linux

- **What happens when a user runs (in the shell)**
  - cp -r ~/foo ~/foo1
- **Shell parses user input**
  - - argc = 4, argv = "cp", "-r", "~/foo", "~/foo1"
- **shell calls fork() and execve("cp", argv, env)**
- **cp uses file system interface to copy files**
- **cp may print messages to stdout**
- **cp exits**

Load user program

User program calls syscall

User Programs In Pintos

- **Pintos implements a basic program loader**
  - Parse and load ELF executables
  - Start executables as a user process with one thread

- **But this system has problems (fixed by you in this lab)**
  - User processes crash immediately :(
  - System calls only print “system call!”

How Does Pintos Start A User Program?

- **In threads/init.c**
  - pintos_init()->run_actions() --> run_task(argv)
  - run_task() --> process_wait(process_execute(task))

```c
/* 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);
}
```

Execute user program & wait

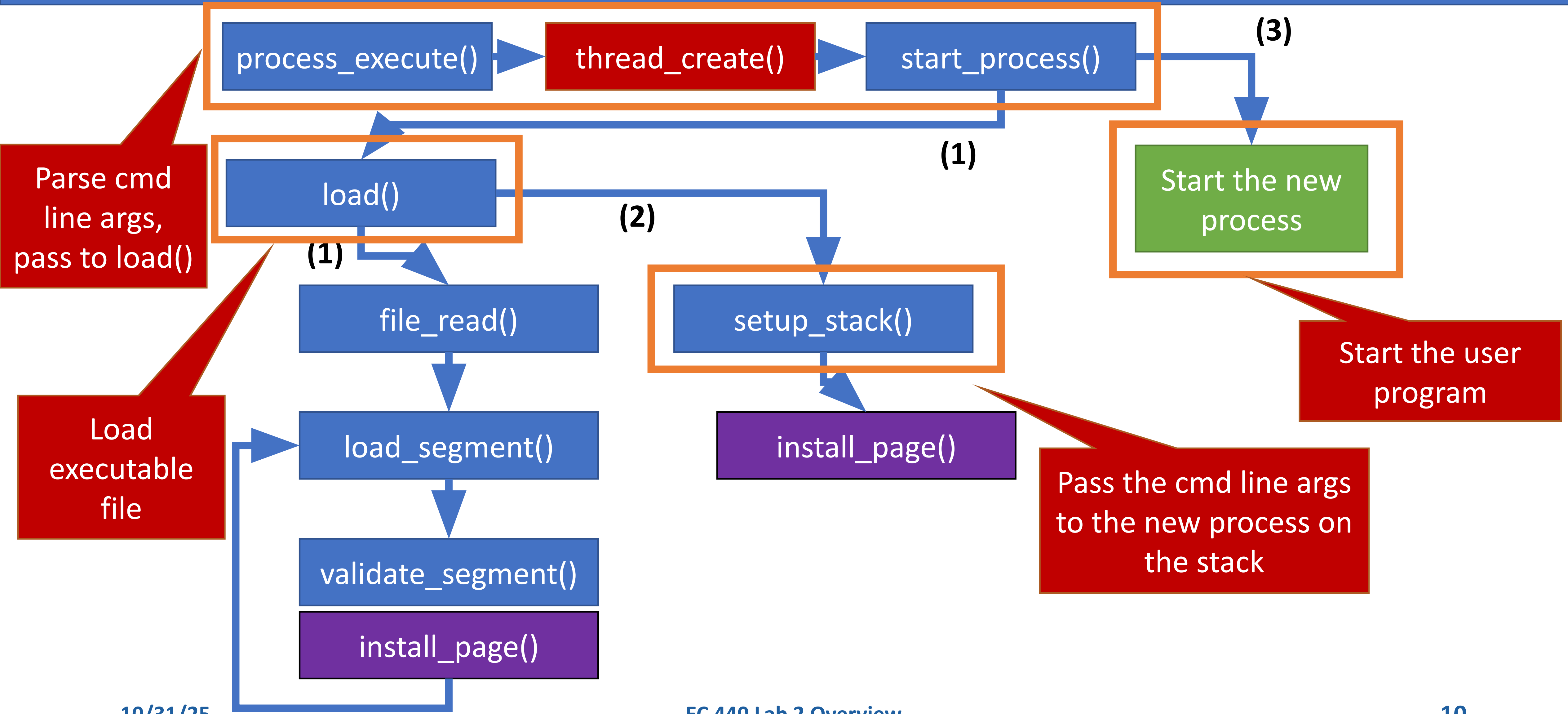
Important Process Functions In Pintos

- **The process_execute() start a process**:
  - creates thread running start_process()
  - start_process() thread loads executable file
  - sets up user virtual memory (stack, data, code)
  - starts executing user process (jump to the start)

- **process_wait() waits for executable to finish**

- **process_exit() frees resources of program**

Pintos Program Loading Flowchart



User Program Startup

- **After Pintos loads executables, it jumps to user process**

```
static void start_process(void *file_name_) {
    struct intr_frame if_ = /* initialize */;
    asm volatile ("movl %0, %%esp; jmp intr_exit" : : "g" (&if_) : "memory");
    NOT_REACHED ();
}
```

- **_start() in lib/user/entry.c is entry point of user programs**

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

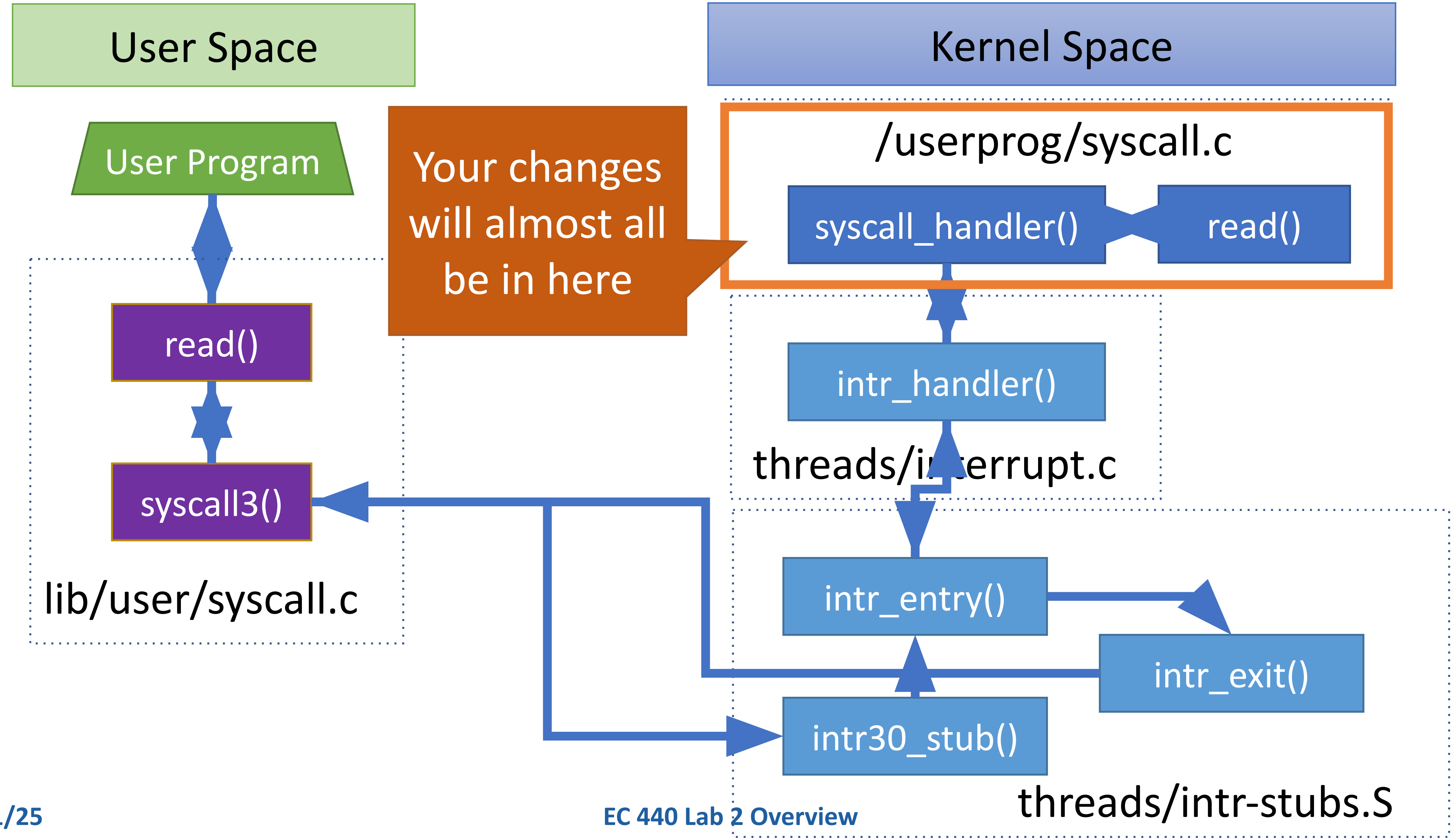
- **Kernel must pass process start arguments on user stack**

How Does Pintos Handle Syscall?

- **Pintos uses int 0x30 for system calls**
- **Pintos has code for dispatching syscalls from user programs**
  - i.e. user processes will push 1) syscall number and 2) arguments onto the stack and execute int 0x30
- **In the kernel, calling syscall_handler() in userprog/syscall.c**

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

Syscall Flowchart (exit)



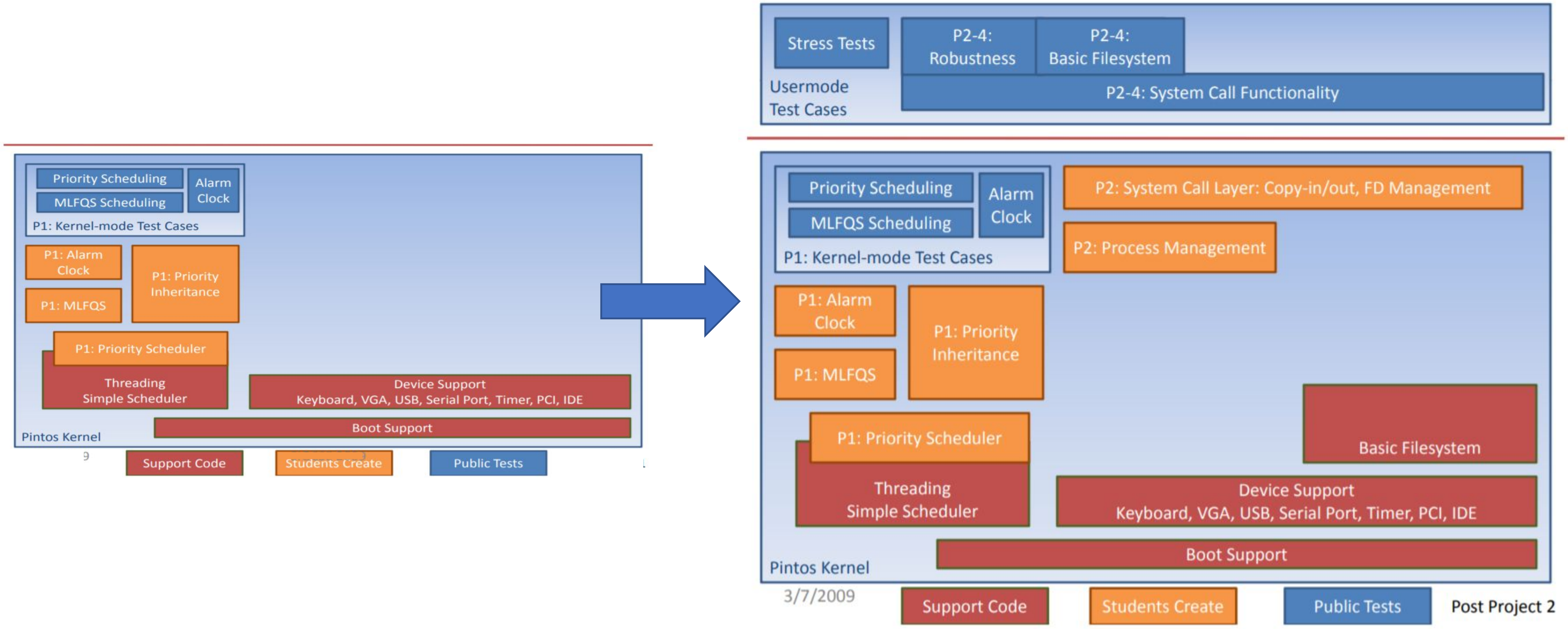
Outline

- User Programs In Pintos
- An Overview of Project 2
- Getting Start
- Tips

Project 2 Requirements

- **In Project 2, you need to implement:**
  - Process exit messages
  - Argument Passing
  - System calls (Major)
  - Safe memory access
  - Denying write to in-use executable files (*Extra Credit*)

Lab 2 Structure



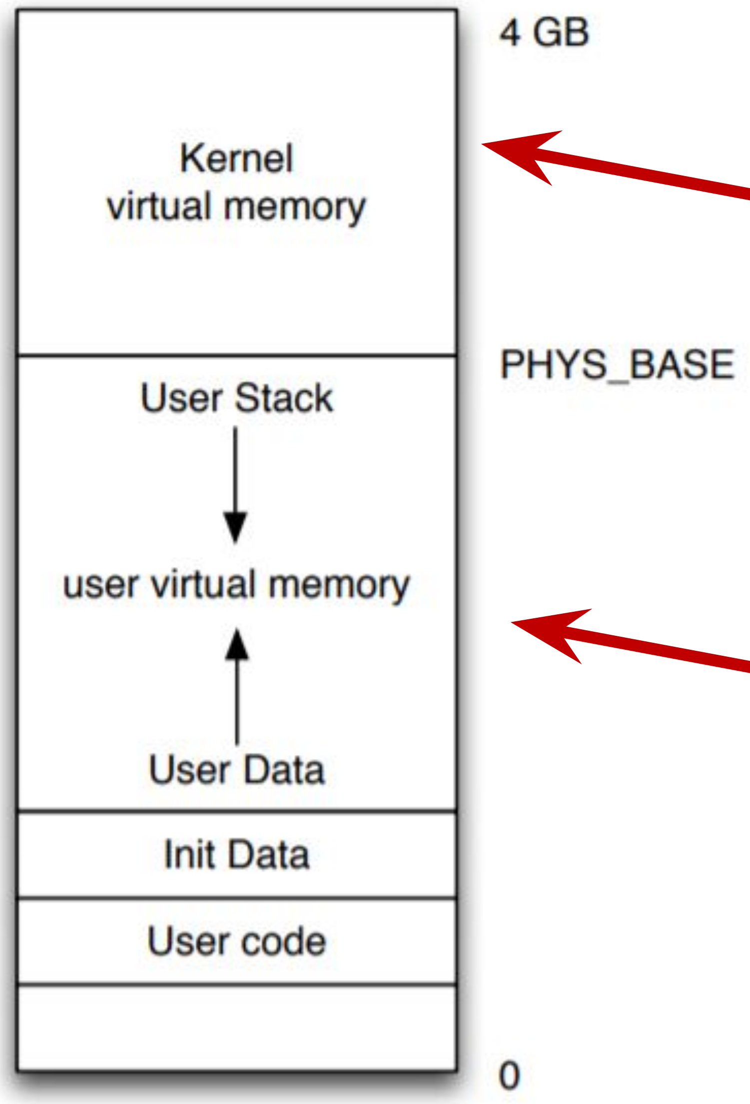
Argument Passing

- **A user process starts at "int main(int argc, char** argv)"**
- **In preparation to start a user process, the kernel must**
  - parse the command

    "/bin/ls -l foo bar" => "/bin/ls" , "-l" , "foo" , "bar"
  - push function's arguments onto the stack

- **Implement the string parsing however you like in load()**
  - strtok_r(...) in lib/string.c is helpful

User vs Kernel Virtual Memory



mapped one-to-one to physical memory

per-process, struct thread contains a pointer to a process's page table

Setting Up User Stack

- **userprog/process.c**

```
/* Create a minimal stack by mapping a zeroed page at the top of user virtual memory.
*/
static bool setup_stack (void **esp) {
    uint8_t *kpage;
    bool success = false;

    kpage = palloc_get_page (PAL_USER | PAL_ZERO);
    if (kpage != NULL) {
        success = install_page (((uint8_t *) PHYS_BASE) - PGSIZE, kpage, true);
        if (success) *esp = PHYS_BASE;
        else palloc_free_page (kpage);
    }
    return success;
}
```

Get one page from page pool, return kernel virtual address

Map the user virtual address to the page

Set the stack point

You need to place argc and *argv on the initial stack, since they are parameters to main()

Argument Passing (Stack)

- Push the words onto the stack
- Word-align
- Push a null pointer sentinel
- Push the address of each word in right-to-left order
- Push argv and argc
- Push 0 as a fake return address

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 (*) ()

Design Tips For Argument Passing

- **Implement user stack push function for argument passing**
  - lib/string.c is helpful
- **Distinguish user virtual address and kernel virtual address when you are coding**
- **hex_dump() function is useful for seeing the layout of stack**
  - void hex_dump (uintptr_t ofs, const void *buf_, size_t size, …)
  - Dumps the SIZE bytes in BUF to the console
  - ofs is the starting address of buf

Safe Memory Access

- **Kernel may access memory through user-provided pointers**
  - E.g. read(), write()
- **This is dangerous!**
  - null pointers
  - pointers to unmapped virtual addresses
  - pointers to kernel addresses
- **In lab 2, you need to support reading from and writing to user memory for system calls that only access valid address**

Two Approaches To Solving Memory Access

- **Approach #1 (simplest): verify every user pointer before dereferencing.**
  - Check in user address space (< PHYS_BASE)
  - Check mapped (using pagedir_get_page() in userprog/pagedir.c)

- **Approach #2: Modify page fault handler in exception.c**
  - Check in user address space (< PHYS_BASE)
  - Dereference. Invalid pointers will trigger page faults
  - More convenient for lab 3

Two Approaches To Solving Memory Access

```c
/* Reads a byte at user virtual address UADDR.
   UADDR must be below PHYS_BASE.
   Returns the byte value if successful, -1 if a segfault
   occurred. */
static int
get_user (const uint8_t *uaddr)
{
  int result;
  asm ("movl $1f, %0; movzbl %1, %0; 1:"
       : "=&a" (result) : "m" (*uaddr));
  return result;
}
```

```c
/* Writes BYTE to user address UDST.
   UDST must be below PHYS_BASE.
   Returns true if successful, false if a segfault occurred. */
static bool
put_user (uint8_t *udst, uint8_t byte)
{
  int error_code;
  asm ("movl $1f, %0; movb %b2, %1; 1:"
       : "=&a" (error_code), "=m" (*udst) : "q" (byte));
  return error_code != -1;
}
```

https://stackoverflow.com/questions/14922022/need-to-figure-out-the-meaning-of-following-inline-assembly-code

System Calls: how do they work?

- **Execute internal interrupt (int instruction)**
  - syscall handler (struct intr_frame *f)

- **Stack pointer: f->esp**
- **Program pointer: f->eip**
- **Return value just like functions (f->eax)**

- **Calling handlers**
  - Pass args to handler
  - Return value to user process

System Calls: Implementation

- **Read syscall number at stack pointer**
- **Dispatch a particular function to handle syscall**
- **Read (validate!) arguments (above the stack pointer)**
  - Above the stack pointer
  - Validate pointers and buffers!
- **Syscall numbers defined in lib/syscall-nr.h**

Syscalls To Implement

- read
- write
- seek
- tell
- close
- create
- remove
- open
- filesize

File syscall

halt

exec

exit

wait

Process syscall

System Call: File System

- **Many syscalls involve file system functionality**
- **Simple filesys implement is provided**: *filesys.h*, *file.h*
  - No need to modify it, but familiarize yourself
- **File system is not thread-safe**!
  - Use a coarse-grained lock to protect it
- **Syscalls take file descriptors as args**
  - Pintos represents files with struct file*
  - You must design the mapping

System Calls: Processes(1)

- **Generally, these syscalls require the most design and implementation time**

- **pid_t exec(const char *cmd line)**
  - Similar to UNIX fork() + execve()
  - Creates a child process
  - Returns after the new process has been created
  - Creation is successful if child has successfully loaded its executable and there is a thread ready to run

System Calls: Processes(2)

- **int wait (pid_t pid)**
  - parent must block until child process pid exits
  - returns exit status of the child
  - must work if child has ALREADY exited
  - must fail if it has already been called on child before
  - you may need to consider many race conditions
- **void exit (int status)**
  - exit with status and free resources
  - process termination message
  - parent must be able to retrieve status via wait

System Calls: Security

- **How does system recover from null pointer segfault in user program?**
  - kill user process, life goes on

- **What about in kernel space?**
  - Verify all user-passed memory references (pointers, buffers, strings)
  - Kill user program if passed illegal addresses

Denying Writes To Executables(Extra Credit)

- **Executables are files like any other**

- **Pintos should not allow code that is currently running to be modified**
  - Use file_deny_write() to prevent writes to an open file
  - Closing a file will re-enable writes
  - Keep executable open as long as the process is running

Outline

- User Programs In Pintos
- An Overview of Project 2
- Getting Start
- Tips

Getting Started

- **Lab 2 does not depend on Lab 1**
  - You can either build on your lab1 submission or start from beginning
- **Lab 3 and lab 4 are built on top of lab2**
  - Any design defects in lab 2 might affect lab3 and lab4

Getting Started: File System Setup

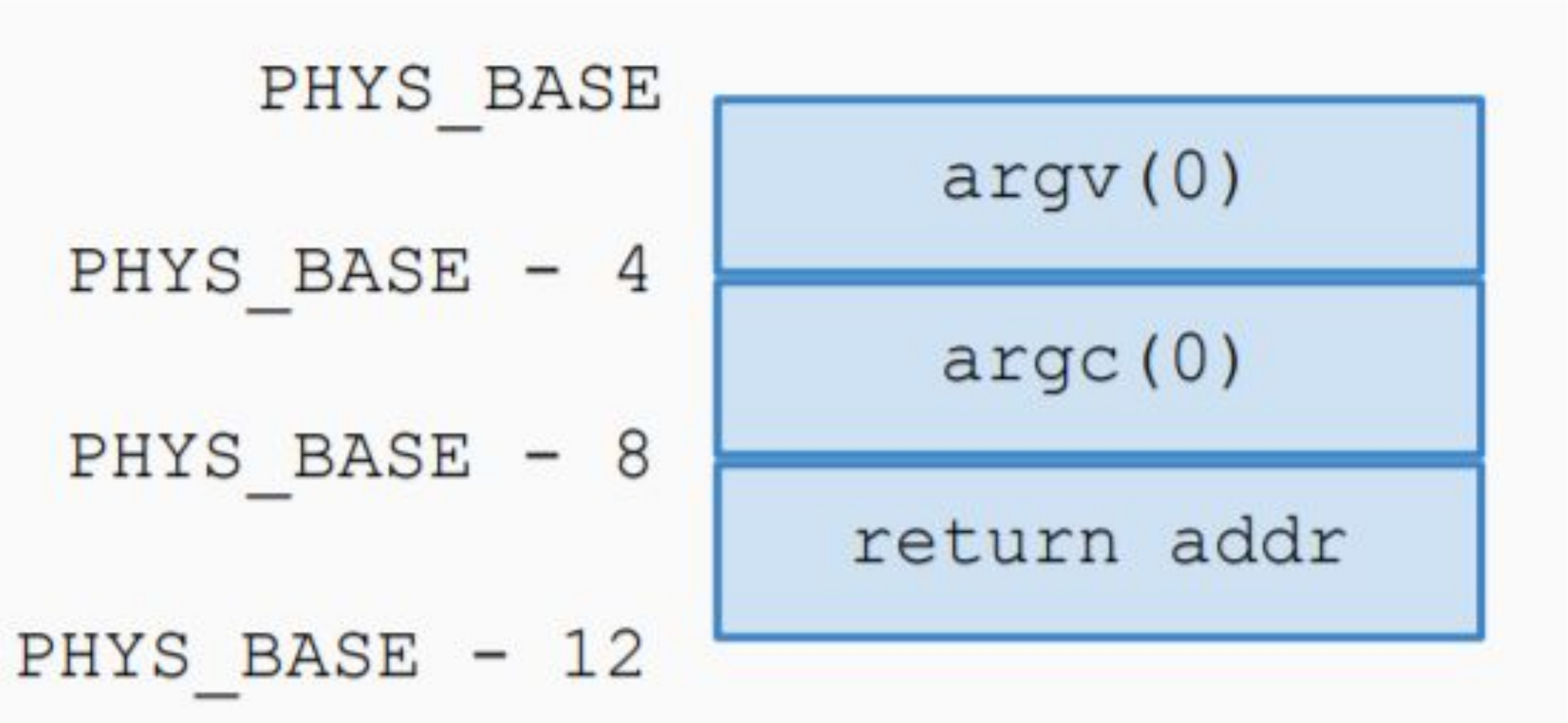
- **You need to format a file system to store user programs**

- **Create a simulated disk called filesys.dsk with a 2MB Pintos file system partition, and then copy programs and run them**
  - Make disk: pintos-mkdisk filesys.dsk --filesys-size=2
  - Format disk: pintos -- -f -q
  - Copy program: pintos -p ../../examples/echo -a echo -- -q
  - Run program: pintos -q run ‘echo x’

Getting Started: Implement this first! (1)

- **Argument passing: change *esp = PHYS_BASE; to *esp = PHYS_BASE - 12;**
  - Allows running programs with no arguments
  - Change again to correct implementation later
- **User memory access**
  - All system calls need to read user memory
- **System call infrastructure**
  - Read system call number from the user stack and dispatch to a handler

Why *esp = PHYS_BASE – 12?



Getting Started: Implement this first! (2)

- **Exit system call**
    - Write system call for STDOUT

- **Temporarily change process_wait to an infinite loop so pintos doesn't immediately power off**

- **Refine your implementation and pass the test**

Outline

- **User Programs In Pintos**
- **An Overview of Project 2**
- **Getting Start**
- **Tips**

General Tips

- **Key to implement lab2: understand the user program**
  - 80x86 Calling Convention
  - Program Startup Details
  - System Call Details
- **Read the design doc together, make sure every member in your group understand the user program**
- **Follow the suggested order of implementation!**
- **Be brave in modifying original definitions**

Debugging Tips

- **If you're confused about why a test is failing, read the source code in tests/userprog**
- **Read the system call APIs carefully, and make sure you validate all user memory addresses**

Common Errors(1)

- **My string is modified after being strtok_r()!**
  - strtok_r() modifies the string, so copy it first
  - be careful when allocating memory for copied buffer, allocating a large buffer = kernel PANIC!
- **hex_dump() prints nothing like it is supposed to be!**
  - check your user page layout and double check how it would be copied to kernel page, also did you specify the right address to print

Common Errors(2)

- **Process terminates before it prints anything!**
  - before you implement sys_wait(), use a while(1) loop to hang main thread so you can see output from user programs
- **Any program with arguments will fail!**
  - use *esp = PHYS_BASE - 12; for now
  - or you can implement arguments passing first (~1 hour)

Security Tips

- **Cast struct file * to int, and use it as the file descriptor? Use struct thread * as pid_t?**
  - info leak
- **write() can be used to dump kernel memory to a file**
  - Forget to check kernel memory boundary?
  - read() can be used to overwrite kernel memory
- **User program takes over kernel!**