CS 112/212 Project 2
User Programs

Spring 2026

Project Overview

Goals

- Allow user programs to run on top of Pintos
  - Interact with OS via system calls
  - Support multiple concurrent processes
  - All processes single-threaded
- Protect/insulate kernel from user programs
  - Nothing a user program does should ever crash/panic/malfunction the OS
- Test your solution by running user programs
  - Check out src/examples
  - More freedom to modify kernel code

General Considerations

- **Project requires a good understanding of:**
  - Steps for running a user program
  - Distinctions between user and kernel virtual memory
  - System call interface and handling
  - Kernel file system interface

Code TODOs

- **1. Argument Passing**
  - Tokenize program arguments and pass them to the new process
- **2. System Calls**
  - Majority of the code you'll write
  - Ensure safe memory access
- **3. Process Termination Messages**
- **4. Denying Writes to Executables**
  - Can be dangerous to change code while it's being run
- **5. Design Doc ⚠️ (50% of your grade!)**

How Much Code?

```
threads/thread.c    |  13
threads/thread.h    |  26 +
userprog/exception.c|   8
userprog/process.c  | 247 +++++++++++++++--
userprog/syscall.c  | 468 ++++++++++++++++++++++++++++++++++++++++++++++++++-
userprog/syscall.h  |   1
_____________________________________________________________________________
6 files changed, 725 insertions(+), 38 deletions(-)
```

**One possible solution — you may modify different files*

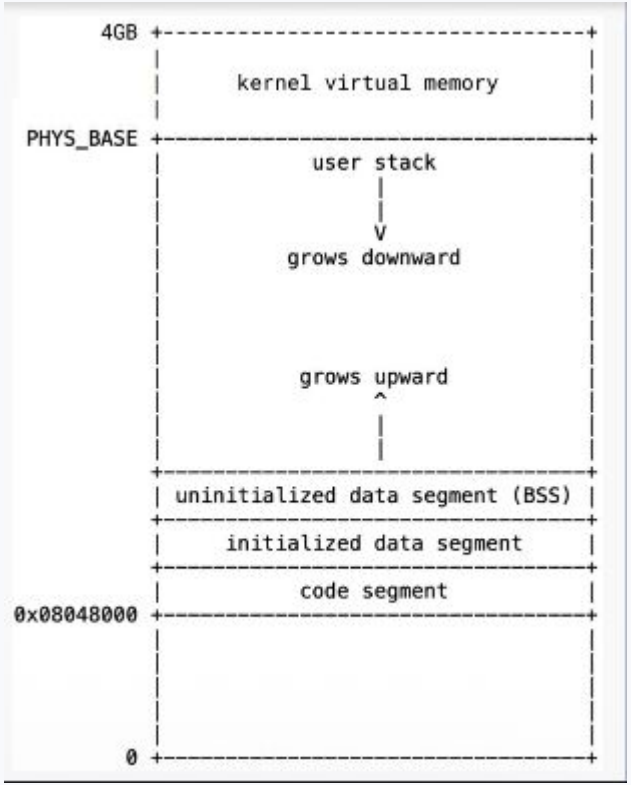
Project Background

File System

- Need to interact with the (bare-bones) file system to load user programs and implement syscalls
  - More functionality coming in Project 4…
- Limitations: fixed-size files, not thread-safe, no subdirectories, etc.
- Need to create a simulated "disk" + file system partition to copy files to/from
  - Including user program executables!
- See Section 3.1.2 for details
- Interfaces in filesys/filesys.h, filesys/file.h

Virtual Memory

- PHYS_BASE is the divider between kernel & user VM
- **Kernel virtual memory is global**
  - Always mapped to contiguous physical memory, starting at PHYS_BASE ↔ physical address 0x0
- **User virtual memory is per-process**
  - A user program can only access its own user virtual memory
  - Kernel switches address spaces on process switch
    - Pointer to page table in struct thread
- See sections 3.1.4, 3.1.4.1



Safe User Memory Access

- ⚠️ **The kernel will often access memory through user-provided pointers!**
- **Beware:**
  - Null pointers
  - Pointers to unmapped virtual addresses
  - Pointers to kernel addresses
- For any of the above: must kill the process
  - Free its resources (e.g. locks, memory)
- **Carefully validate buffers, strings, and all pointers passed from user → kernel**

Safe User Memory Access — Two Approaches

- **Approach 1: Verify every pointer before dereference (simpler)**
  - Ensure pointer is in user's address space, i.e. below PHYS_BASE (see threads/vaddr.h)
  - Ensure it is mapped: pagedir.c:pagedir_get_page()
  - For buffers, must validate the entire buffer

Safe User Memory Access — Two Approaches

- **Approach 1: Verify every pointer before dereference (simpler)**
  - Ensure pointer is in user's address space, i.e. below PHYS_BASE (see threads/vaddr.h)
  - Ensure it is mapped: pagedir.c:pagedir_get_page()
  - For buffers, must validate the entire buffer
- **Approach 2: Modify fault handler in userprog/exception.c (better performance)**
  - Only ensure pointer/buffer is below PHYS_BASE, then dereference
  - Invalid pointers will trigger page faults
  - Trickier; see section 3.1.5 for full details

x86 Calling Convention TL;DR

- Relevant for: normal function calls, user process creation, system calls
  - **1. Caller pushes each argument on the stack, right-to-left order**
  - **2. Caller pushes return address and jumps to callee**
  - **3. Callee executes; return value stored in register eax**
  - **4. Callee returns by popping the return address and jumping to it**
  - **5. Caller pops arguments off the stack**
- **ONLY A SUMMARY**; **Read section 3.5** (including 3.5.1 and 3.5.2) for full details!

Project Tasks

Argument Passing

Argument Passing

- Before starting a user process, the kernel must set up the stack by pushing command-line arguments
- Extend process_execute() to parse arguments
  - See Section 3.3.3; helper functions in lib/string.h
- Set up the stack for the program entry function _start()
  - Full signature: void _start(int argc, char* argv[])
  - Push argc and elements of argv with associated pointers
  - Must follow the x86 calling convention (Section 3.5.1)

Argument Passing — Stack Layout Example

Example: "/bin/ls -l foo bar"

PHYS_BASE = 0xc0000000

C strings referenced by the elements of argv

argv[i] in reverse order (argv[0] last)

argv (the address of argv[0]) and then argc

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 addr	0	void(*)()

hex_dump() from <stdio.h> can be helpful for debugging

System Calls

System Calls — Overview

- Implement syscall dispatcher in syscall.c:syscall_handler()
  - Read syscall number and args; dispatch to specific handler
  - Validate everything the user provides (syscall number, args, pointers)
  - See Section 3.5.2 for syscall mechanics and calling convention
- Implement 13 system call handlers
  - Syscall numbers defined in lib/syscall-nr.h
  - See Section 3.3.4 for the complete list

System Calls — Details

- **Filesystem-related syscalls**
  - Familiarize yourself with filesys/filesys.h and filesys/file.h
- **Process-related syscalls**
  - exec(), wait(), exit()
  - Likely the most work to design and implement
- **Synchronization**
  - Any number of user processes can make syscalls simultaneously
  - The provided file system is NOT thread-safe; use a coarse lock to protect it

Other Tasks

Process Termination Messages

- See Section 3.3.2
- `printf("%s: exit(%d)\n", thread_current()->name, exit_code);`
- DO print when a user process terminates
- Don't print command-line arguments
- Don't print when a kernel thread (non-user-process) terminates
- Don't print when the halt syscall is invoked

Denying Writes to Executables

- See Section 3.3.5
- Executables are files that happen to contain code to run
- User processes should not be able to modify currently-running code
  - file_deny_write() — prevents writes to an open file
  - file_allow_write() — re-enables writes
  - Closing a file will re-enable writes automatically
- Keep the executable open for as long as the process runs, then close on termination

Implementation Order & Tips

Suggested Order of Implementation (Section 3.2)

- Create the simulated disk to put user programs on
- Implement argument passing
  - Temporarily bypass: set *esp = PHYS_BASE - 12 to avoid early page faults
- Implement safe user memory access/validation
  - All syscalls need to read user memory safely
- Implement syscall infrastructure
  - Read syscall number from f->esp and dispatch to handler
- Implement exit() — used by every user program that finishes normally
- Implement write() to fd 1 (STDOUT_FILENO) for printing
- Change process_wait() to an infinite loop initially, then implement properly
  - Otherwise Pintos powers off before processes get to run

Suggested Project 2 Split– Group of 3

Person 1 — Argument Passing & Memory Safety

- Parse args in `process_execute()`, set up stack (argv, argc, x86 convention)
- User memory validation (`get_user`/`put_user` or pagedir checks)
- *Finish first — everyone else is blocked on this*

Person 2 — File System Calls

- `create`, `remove`, `open`, `close`, `read`, `write`, `seek`, `tell`, `filesize`
- File descriptor table per process
- Global filesystem lock
- `halt`, `exit`, deny writes to executables

Person 3 — Process Lifecycle

- `exec` with parent/child load synchronization
- `wait` — hardest part; shared state that survives both processes dying
- Process termination messages, `process_wait()`

Suggested Project 2 Split– Group of 3

Person 1 — Argument Passing & Memory Safety

- Parse args in `process_execute()`, set up stack (argv, argc, x86 convention)
- User memory validation (`get_user`/`put_user` or pagedir checks)
- *Finish first — everyone else is blocked on this*

Person 2 — File System Calls

- `create`, `remove`, `open`, `close`, `read`, `write`, `seek`, `tell`, `filesize`
- File descriptor table per process
- Global filesystem lock
- `halt`, `exit`, deny writes to executables

Person 3 — Process Lifecycle

- `exec` with parent/child load synchronization
- `wait` — hardest part; shared state that survives both processes dying
- Process termination messages, `process_wait()`

Key dependencies:

- Person 1 unblocks everyone → finish first
- Persons 2 & 3 coordinate on `exit` (Person 3 owns it)
- Person 3 has hardest conceptual work; Person 2 has most lines of code

General Tips

- Read the guide 2× before starting (including FAQs!)
- Read the tests so you know how syscalls are invoked
- **Read through the design doc before starting — it's 50% of your grade!**
    - Come up with preliminary answers to the design doc before writing any code
- Try the simplest thing first
- Use the GDB macro: loadusersymbols (see Appendix E.5.2)
- **Don't write code until you're confident you understand the requirements**