CS 112/212 Project 2: User Programs

Spring 2025

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 that a user program can do should ever cause OS to crash/panic/malfunction
- 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
   a. Tokenize program arguments and pass them to the new process
2. System Calls
   a. **Majority of the code you'll write**
   b. **Ensure safe memory access**
3. Process Termination Messages
4. Denying Writes to Executables
   a. Can be dangerous to change code while it's being run
5. **!!! Design Doc !!!**
   a. **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(-)
```

*see project guide for other files to read over (mainly for understanding filesystem and some VM-related functions)

Just one possible solution! Might not need to modify all of these, or might modify others.

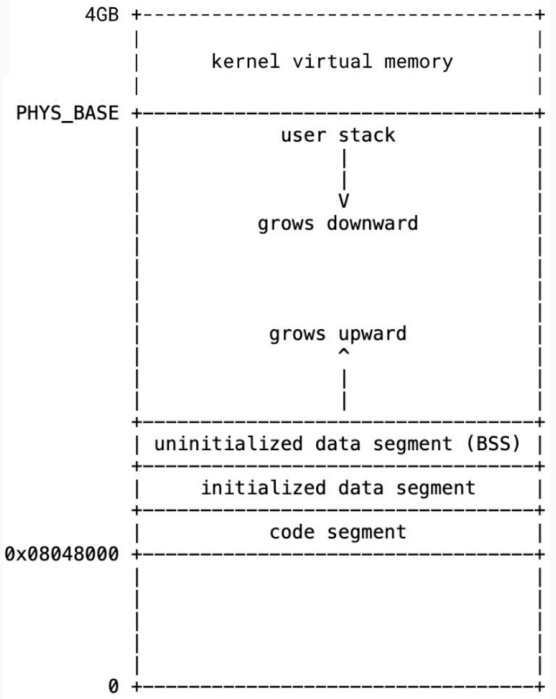
Project Background

File System

- Need to interact with the (for now, bare-bones) file system to load user programs and implement several system calls
  - More to come 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 (virtual address `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 section 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, any other pointers that are passed from user → kernel

Safe User Memory Access

Two approaches to solving this problem:

- Verify **every** user pointer before dereference (simpler)
  - Ensure it is in user's address space, i.e. below `PHYS_BASE` (see `threads/vaddr.h`)
  - Ensure it is mapped: look at `pagedir.c:pagedir_get_page()`
  - For buffers, must validate the entire buffer
- Modify **fault handler** in `userprog/exception.c` (better performance)
  - Only ensure pointer (or buffer) is below `PHYS_BASE`, then dereference
  - Invalid pointers will trigger page faults
  - Trickier than first approach; More nuances (and both approaches' descriptions) in section 3.1.5

x86 Calling Convention TL;DR

Relevant for all of: normal function calls, user process creation, system calls

- Caller pushes each of the function's arguments on the stack, in **right-to-left** order.
- Caller pushes the return address on the stack and jumps to the callee.
- Callee executes. If the callee has a return value, it is stored in register `eax`.
- Callee returns by popping the return address from the stack and jumping to it.
- The caller pops the arguments off the stack.

ONLY A SUMMARY; **please read section 3.5** (including 3.5.1 and 3.5.2) for important details

Project Tasks

Argument Passing

Argument Passing

- Before starting a user process, the kernel must set up the stack by pushing the command-line arguments
- Extend `process_execute()` to parse arguments
  - 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[])`
  - Need to properly push `argc`, and elements of `argv` with associated pointers, according to x86 calling convention
  - Section 3.5.1

Argument Passing

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

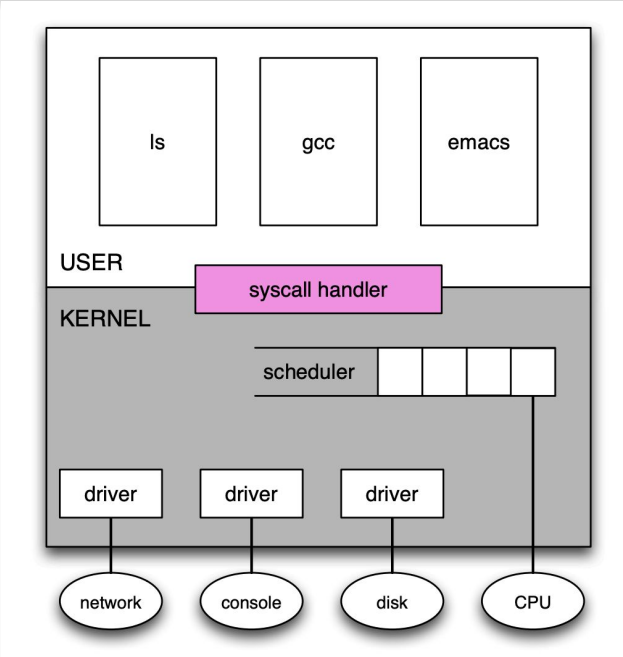
fake “return address”

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

System Calls

System Calls



System Calls

- Implement syscall dispatcher in `syscall.c:syscall_handler()`
  - Read syscall number and args; dispatch to specific function to handle each syscall
  - **Validate everything** that the user provides (e.g. syscall number, args, ptrs)
  - Section 3.5.2 for syscall mechanics and calling convention
- Implement 13 system call handlers
  - Syscall numbers are defined in `lib/syscall-nr.h`
  - Section 3.3.4 for the list of syscalls you need to implement

System Calls

- Filesystem-related syscalls
  - Be familiar with what the filesystem does for you (`filesys/filesys.h`, `filesys/file.h`)
- Process-related syscalls
  - `exec()`, `wait()`, `exit()`
  - Likely the most work to design & implement
- Synchronization
  - Any number of user processes can make syscalls at once
  - The provided file system is not thread-safe; must use coarse lock to protect

Other Tasks

Process Termination Messages

- Section 3.3.2
- `printf ("%s: exit(%d)\n", thread_current()->name, exit_code);`
- **Do** print this whenever a user process terminates
- **Don't** print command-line arguments
- **Don't** print when a kernel thread (that isn't a user process) terminates
- **Don't** print when `halt` syscall is invoked

Denying Writes to Executables

- Section 3.3.5
- Executables are files like any other, that happen to contain code to run
- User processes shouldn't be able to modify code that is currently running
  - `file_deny_write()` to prevent writes to an open file
  - `file_allow_write()` to enable writes to an open file
  - **Closing a file will re-enable writes**
  - Thus, keep executable open as long as the process is running, until the process terminates

Implementation Order + Tips

Suggested Order of Implementation (Section 3.2)

- Create the simulated disk to put user programs on
- Argument passing
  - To start, can temporarily bypass by setting `*esp = PHYS_BASE - 12` to avoid page faults
    - Can run programs with no arguments this way
- Safe user memory access/validation
  - All syscalls need to read user memory
- Syscall infrastructure
  - To start, read the system call number from `f->esp` and use it to dispatch to specific handler
- `exit()` (used by every user program that finishes normally)
- `write()` (writing to fd 1, `STDOUT_FILENO`, needed for printing)
- To start, change `process_wait()` to an infinite loop, then implement later
  - Otherwise, Pintos will power off before any processes actually get to run

General Tips

- Read the guide 2x before starting **(including FAQs)**
- Read the tests so you know how the syscalls are invoked
- Read through the **design doc (50% of your grade)** before starting
- Don’t write any code until you feel confident that you understand the requirements
  - Come up with preliminary answers to the **design doc (50% of your grade)** before coding
- Try the simplest thing first
- GDB macro: `loadusersymbols` (Appendix E.5.2)