CS 140 Project 2: User Programs

January 24, 2020

Today’s Topics

- **Overview**
- **Project 2 Requirements**
    - Process Termination Messages
    - Argument Passing
    - System Calls
    - Denying Writes to Executables
- **Getting Started**

Project Overview

- **Allow user programs to run on top of Pintos**
  - Interact with OS via system calls
  - More than one process can run at a time
  - Each process has one thread (no multi-threaded processes)
- **Protect kernel from user programs**
- **Test your solution by running user programs**
  - Free to modify kernel code however you like

Project Overview

Reference Implementation:

```
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(-)
```

- Most changes in `userprog/process.c` and `userprog/syscall.c`.
- Need to get familiar with a few other files (covered later).

Default File System in Pintos

- **Simple file system implementation provided to help you**
    - No need to modify (that's Project 4)
    - Get familiar with functions defined in `filesys.h` and `file.h`
- **Be careful about the limitations!**
    - E.g., the file system is not thread-safe
    - Details in Section 3.1.2

Virtual Memory Layout

- **Virtual memory divided into two regions**
  - User virtual memory: **`[0, PHYS_BASE)`**
  - Kernel virtual memory: **`[PHYS_BASE, 4GB)`**
- **User virtual memory is per-process**
  - Switch virtual address space during context switch
- **Kernel virtual memory is global**
  - Always mapped to contiguous memory starting from physical address 0

```
        4GB +----------------------------------+
            |                                  |
            |      kernel virtual memory       |
            |                                  |
  PHYS_BASE +----------------------------------+
            |            user stack            |
            |                 |                |
            |                 |                |
            |                 V                |
            |          grows downward          |
            |                                  |
            |                                  |
            |                                  |
            |           grows upward           |
            |                 ^                |
            |                 |                |
            |                 |                |
            +----------------------------------+
            | uninitialized data segment (BSS) |
            +----------------------------------+
            |     initialized data segment     |
            +----------------------------------+
            |           code segment           |
 0x08048000 +----------------------------------+
            |                                  |
            |                                  |
          0 +----------------------------------+
```

Accessing User Memory

- **Kernel must validate pointers provided by a user program**
    - E.g., null pointers, pointers to unmapped/kernel virtual memory
    - Terminate the offending process and free its resources
- **Two approaches to implement**
    - Approach 1: check `is_user_vaddr()` and mapped (hint: `userprog/pagedir.h`)
    - Approach 2: check `is_user_vaddr()`; dereference and handle page fault
    - Details in Section 3.1.5

80x86 Calling Convention

- **How to make a normal function call? (Details omitted)**
  - Caller pushes arguments on the stack one by one, from right to left
  - Caller pushes the return address and jumps to the first line of the callee
  - Callee executes and takes arguments above the stack pointer
  - Details in Section 3.5 and Lecture 2 slides
- **Also applicable to scenarios beyond normal function calls**
  - Program startup
  - System call

Today’s Topics

- Overview
- **Project 2 Requirements**
    - Process Termination Messages
    - Argument Passing
    - System Calls
    - Denying Writes to Executables
- Getting Started

Process Termination Messages

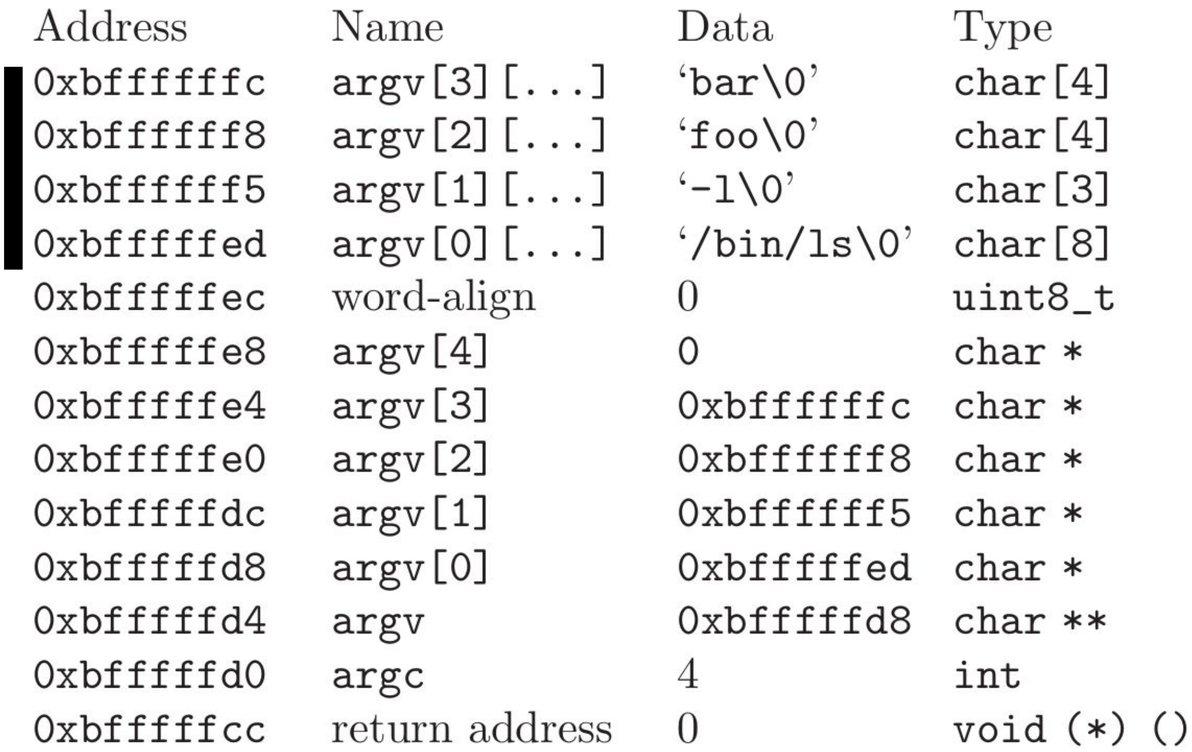
- **`printf(“%s: exit(%d)\n”, process_name, exit_code)`**
    - Print the message whenever a user process terminates
    - Do not print command-line arguments
    - Do not print when a kernel thread terminates
    - Do not print when the `halt` system call is invoked

Passing Arguments to New Process

- **Extend `process_execute()` to parse command arguments**
  - `process_execute(“grep foo bar”)` should run `grep` with two args
  - Helper functions in `lib/string.h`
- **Set up the stack for the program entry function `_start()`**
  - Signature: `void _start(int argc, char* argv[])`
  - Push `C` strings referenced by the elements of `argv`
  - Push `argv[i]` in reverse order (`argv[0]` last)
  - Push `argv` (the address of `argv[0]`) and then `argc`
  - Push a fake “return address” (required by 80x86 calling convention)
  - Details in Section 3.5.1 [Program Startup Details]

Example: “/bin/ls -l foo bar”

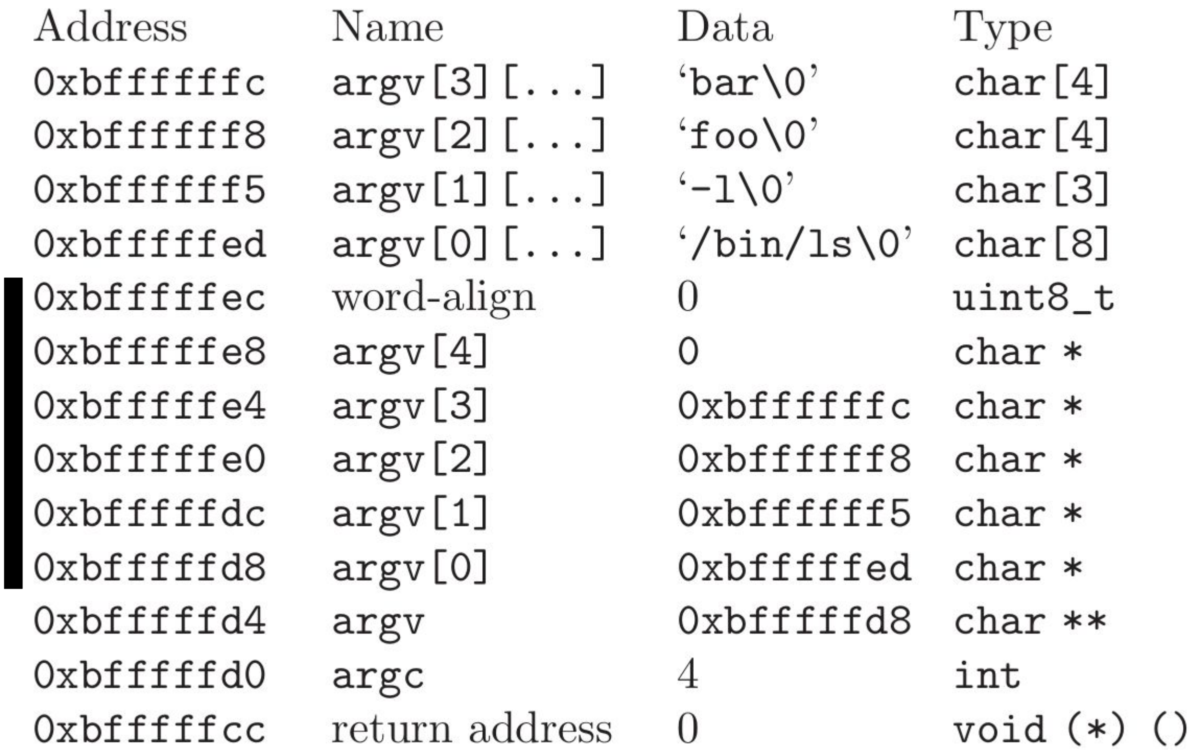
PHYS_BASE = 0xc0000000



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 (*) ()

Example: “/bin/ls -l foo bar”

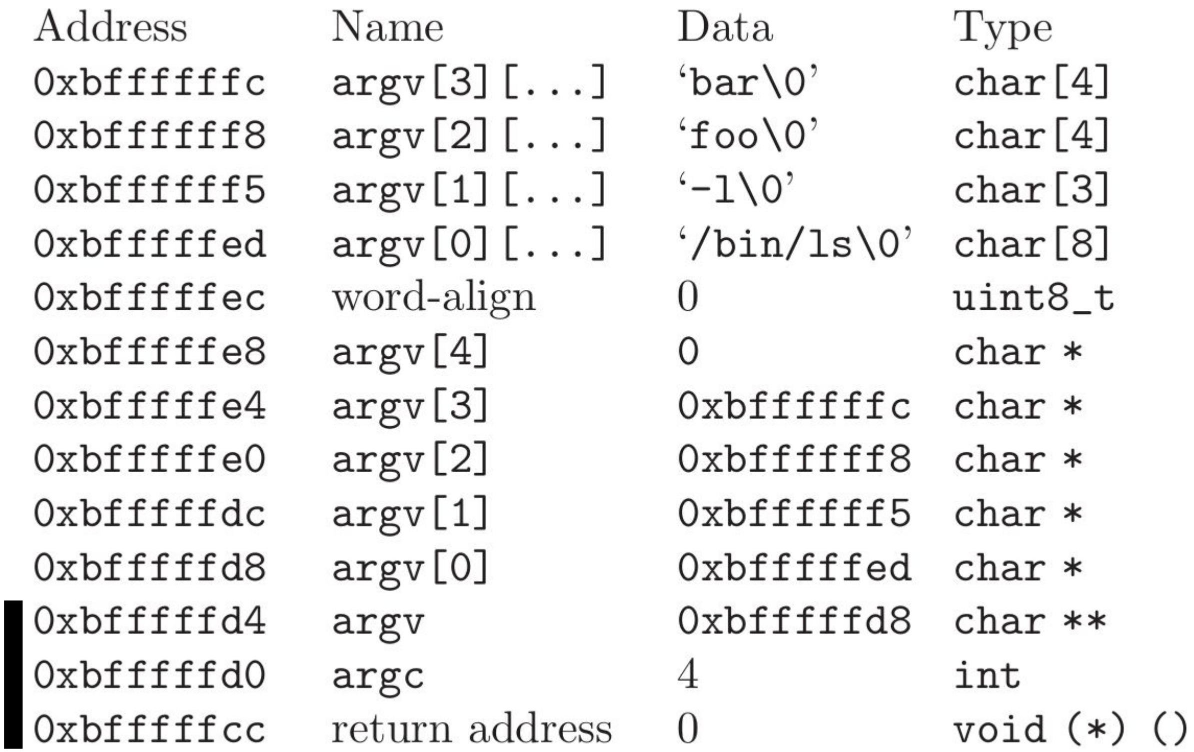
`PHYS_BASE = 0xc0000000`



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 (*) ()

Example: “/bin/ls -l foo bar”

PHYS_BASE = 0xc0000000



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 (*) ()

System Calls

- **Implement system call dispatcher (i.e., `syscall_handler()`)**
  - Read system call number and args; dispatch to specific handler
  - Details in Section 3.5.2
  - Validate everything user provides (e.g., syscall numbers, arguments, pointers)
- **Implement 13 system call handlers in `userprog/syscall.c`**
  - System call numbers defined in `lib/syscall-nr.h`
  - Some system call requires considerably more work than others (e.g. **`wait`**)
- **Synchronization**
  - Any number of user processes can make system calls at once
  - The provided file system is not thread-safe

Denying Writes to Executables

- **Deny writes to files in use as executable**
    - Unpredictable results to change and run code concurrently
    - Especially important once virtual memory is implemented in project 3
- **`file_deny/allow_write()`: disable/enable writes to open files**
    - Keep the executable file open until the process terminates

Today’s Topics

- **Overview**
- **Project 2 Requirements**
    - Process Termination Messages
    - Argument Passing
    - System Calls
    - Denying Writes to Executables
- **Getting Started**

Getting Started

- **You can build on top of Project 1 or start fresh**
  - No code from project 1 will be required
- **File system setup**
  - User programs must be loaded from this file system (not your host file system)
  - Create a simulated disk with a file system partition
  - Copy files into/from this file system
  - Details in Section 3.1.2

Suggested Order of Implementation

- **Bypass argument passing**
    - In `setup_stack()`, change `*esp = PHYS_BASE;` to `*esp = PHYS_BASE - 12;`
    - Run test programs with no command-line arguments
- **Safe user memory access**
    - All system calls need to access user memory
- **System call infrastructure**
    - Read syscall numbers and args, dispatch to the correct handler

Suggested Order of Implementation

- **The `exit` system call**
  - Every user program calls `exit` (sometimes implicitly)
- **The `write` system call to console**
  - User program can use `printf()` to write to screen
- **Change `process_wait()` to an infinite loop**
  - Don’t let Pintos power off before any processes actually get to run.

Simple user programs should start to work.

Tips

- **Use GDB for user programs**
    - GDB Macro: `loadusersymbols` *`program`*
    - Details in Appendix E.5.2
- **Use GDB Text User Interface (TUI)**
    - `tui enable`
- **Read the design doc early**
    - Design, then write code
- **Read the specification carefully**
    - Lots of pieces in this assignment

Questions?