Project 2: User Programs

1/21/2022

Today

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

Overview

- Enable user programs. In other words, facilitate processes on top of the OS
  - *Multiple* processes
  - Interact with OS via **system calls**
- Protect the kernel from user programs
- Test your solution by running actual user programs
  - See *src/examples* directory

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(-)
```

- See the spec for other files to understand: pagedir.h, gdt.h, tss.h

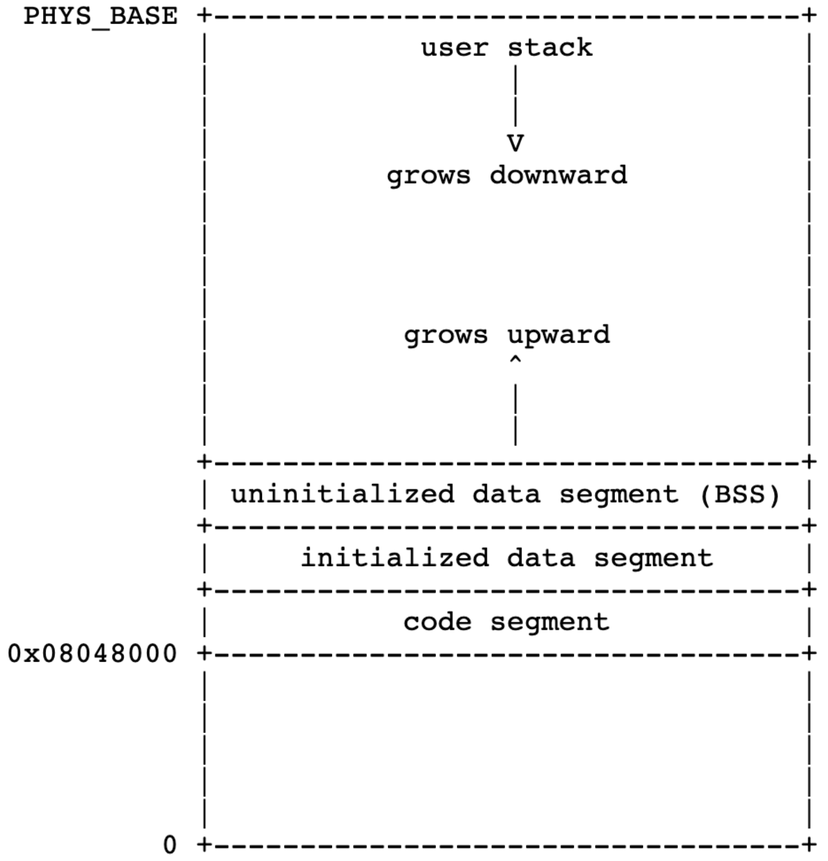
Filesystem

- You will need to interface with the Pintos file system.
- Naive:
  - No internal synchronization
  - Fixed file sizes
  - Files stored in contiguous segments
  - File names <= 14 characters
- Create a simulated, partitioned disk:

```
pintos-mkdisk filesys.dsk --filesys-size=2
pintos -f -q
pintos -p ../../examples/echo -a echo -- -q
pintos -q run 'echo x'
```

Virtual Memory

- Virtual memory divided into two regions
  - User virtual memory: [0, PHYS_BASE)
  - Kernel virtual memory: [PHYS_BASE, 4GB)
- User virtual memory is per-process
  - Check out the pointer to the page table within the thread struct
- Kernel virtual memory is mapped to contiguous physical memory starting from address 0
- Page faults



Avoiding page faults in kernel mode

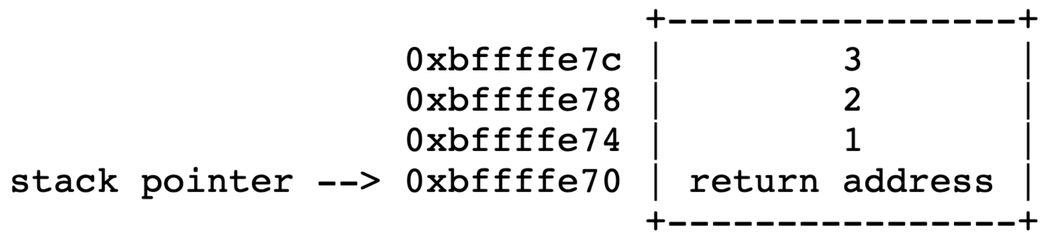
- Kernel must validate pointers provided by a user program
- Why?…

Accessing user memory in kernel mode

- Kernel must validate pointers provided by a user program
- Why: null pointers, unmapped virtual address, a pointer to a kernel VA
- How:
  - (Simpler) Validate a user-provided pointer before dereferencing
  - Allow the page fault

80x86 Calling Convention

- f(1, 2, 3);



- Caller pushes arguments onto the stack, from right to left.
- Caller pushes the return address and jumps to the first line of the callee

Requirements

Process Termination Message

- printf ("%s: exit(%d)\n", thread_current()->name, exit_code);
- Don’t print when a kernel thread terminates
- Don’t print upon halt

Passing Arguments to a New Process

- Start by tracing through a call to process_execute();
- Main idea: process_execute(“grep foo bar”) should run grep with its two arguments
- You need to prepare the stack for the program entry function, _start().

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

Passing Arguments to a New Process

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() will be your friend when implementing this!!

System Calls

- Implement system call dispatcher (i.e., syscall_handler())

intr_register_int (0x30, 3, INTR_ON, syscall_handler, "syscall");

- Read system call number and args

f->esp

- Implement 13 system call handlers
  - Syscall numbers are defined in lib/syscall-nr.h
  - For filesystem-related syscalls, be familiar with what the filesys does for you. (see filesys.h & file.h)
- Synchronization
  - Any number of user processes can make syscalls at once
  - The provided file system is not thread-safe

Denying Writes to Executables

- A user process shouldn’t be able to modify in-use executables
- *file_deny_write()* will prevent writes to an open file
- Closing a file will re-enable writes

Getting Started

Where to start (see 3.2)

- Temporarily bypass argument passing
  - **esp = PHYS_BASE - 12;*
- Basic system call infrastructure
  - Read the syscall numbers and surface arguments from the stack
- User memory access
- Exit()
- Implement write() to write to fd 1, the system console
- Modify process_wait() to infinite loop

Tips

- Read the spec 2x before starting
- Read the tests so you know how the syscalls are invoked
- Read through the design doc before starting
- Don’t write any code until you feel confident that you understand the requirements
- Try the simplest thing first