Project 2

Thierry Sans

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

Scope of the work

```
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`

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 4.1.2

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 4.1.2

Compiling and running

0. Compile the examples

```
$ cd /pintos/src/examples
$ make
```

1. Compile the code in `src/userprog` (no need to compile `src/threads` anymore)

```
$ cd /pintos/src/userprog
$ make
```

2. Run pintos with the `userprog` kernel and filesystem

```
$ cd /pintos/src/userprog/build
$ pintos
--filesys-size=2
-p /pintos/src/examples/echo -a echo
-- -f -q run 'echo x'
```

Desirable timeline

- Week 1 : safe memory access and system call setup
- Week 2 & 3 : argument passing and more system calls
- End of week 3 : denying writes to executables

Tips

- Use GDB for user programs
  GDB Macro : `loadusersymbols` program

✓ Details in Appendix E.5.2

- Read the design doc early
  design, then write code

- Read the specification carefully
  lots of pieces in this assignment

week 1

safe memory access & system call setup

Virtual Memory Layout

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

Virtual memory divided into two regions

- **User virtual memory** is per-process `[0, PHYS_BASE)`
  switch virtual address space during context switch

- **Kernel virtual memory** is global `[PHYS_BASE, 4GB)`
  always mapped to contiguous memory starting from physical address 0

Safe Memory Access

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 4.1.5

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

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 4.5

➡ Also applicable to scenarios beyond normal function calls

- System call (this week)
- Program startup (next week)

So let us ignore passing argument to **process** for now

1. Add a bypass argument passing
    - in setup_stack(),
      change `*esp = PHYS_BASE;`
      to `*esp = PHYS_BASE - 12;`
    - and run test programs with no command-line arguments
2. Enforce safe user memory access
   all system calls need to access user memory
3. Setup the system call infrastructure
   read syscall numbers and args, dispatch to the correct handler

System Calls

Implement system call dispatcher i.e. `syscall_handler()`

- Read system call number and args; dispatch to specific handler
- Validate everything user provides (e.g. syscall numbers, arguments, pointers)

✓ Details in Section 4.5.2

Synchronization

- Any number of user processes can make system calls at once
- The provided file system is not thread-safe

Start implementing your first system calls

- 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 to not let Pintos power off before any processes actually get to run

➡ Simple user programs should start to work

week 2 & 3

argument passing & more system calls

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`
- Do not forget to remove the argument passing bypass from last week

Set up the stack for the program entry function `void _start(int argc, char* argv[])`

1. Push C strings referenced by the elements of `argv`
2. Push `argv[i]` in reverse order (`argv[0]` last)
3. Push `argv` (the address of `argv[0]`) and then `argc`
4. Push a fake "return address" (required by 80x86 calling convention)

✓ Details in section 4.5.1

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 (*) ()

Then finish implementing all system calls

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)

end of week 3

denying writes to executables

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)

Acknowledgments

Some of the course materials and projects are from

- Ryan Huang - teaching CS 318 at *John Hopkins University*
- David Mazière - teaching CS 140 at *Stanford*