Project 2 Overview

CS162, 7/13-14/2015

Deadlines

- **Design document:** 7/16/15 *before class*
- **Design review:** 7/17/15, sign up by 7/15/15
- **Checkpoint 1:** 7/20/15
- **Project deadline:** 7/27/15

After project 1, what does pintos do/not do?

- You can:
  - Launch user programs
  - Use different scheduling policies
- Programs cannot:
  - Make I/O calls
  - Interact with the user *or* the OS

What are *you* building?

System call handlers!

What does this require *you* to build?

- General:
    - Infrastructure for parsing the arguments of a system call
    - Infrastructure for directing the arguments to the code to run the system call
- Specific:
    - 3 simple syscalls
    - 4 Process syscalls
    - 9 File System syscalls

Syscalls in Pintos

How do syscalls work in pintos?

- To make a syscall, the syscall number and syscall arguments are pushed onto the stack as if the syscall is a normal function
- The user program then calls `int $0x30`
- This invokes the `syscall_handler` interrupt handler inside of the pintos kernel, which then dispatches the syscall
- The current handler terminates if the exit syscall is run.

Review: passing arguments

- Arguments are passed to a function by pushing the arguments in *reverse order* onto the stack.
    - After we push the arguments, we push the return address.
- E.g., for myFn(0, 2, 1), we push:

Review: passing arguments

- Arguments are passed to a function by pushing the arguments in *reverse order* onto the stack.
    - After we push the arguments, we push the return address.
- E.g., for myFn(0, 2, 1), we push:

```
stack ptr ->  0xbffffe7c: 0x1
```

Review: passing arguments

- Arguments are passed to a function by pushing the arguments in *reverse order* onto the stack.
    - After we push the arguments, we push the return address.
- E.g., for myFn(0, 2, 1), we push:

```
                  0xbffffe7c: 0x1
stack ptr ->      0xbffffe78: 0x2
```

Review: passing arguments

- Arguments are passed to a function by pushing the arguments in *reverse order* onto the stack.
    - After we push the arguments, we push the return address.
- E.g., for myFn(0, 2, 1), we push:

```
                    0xbffffe7c: 0x1
                    0xbffffe78: 0x2
stack ptr ->        0xbffffe74: 0x0
```

Review: passing arguments

- Arguments are passed to a function by pushing the arguments in *reverse order* onto the stack.
    - After we push the arguments, we push the return address.
- E.g., for myFn(0, 2, 1), we push:

```
                    0xbffffe7c: 0x1
                    0xbffffe78: 0x2
                    0xbffffe74: 0x0
stack ptr ->        0xbffffe70: RET
```

Review: passing arguments

- Arguments are passed to a function by pushing the arguments in *reverse order* onto the stack.
    - After we push the arguments, we push the return address.
- E.g., for myFn(0, 2, 1), we push:

```
                    0xbffffe7c: 0x1
                    0xbffffe78: 0x2
                    0xbffffe74: 0x0
stack ptr —>        0xbffffe70: RET
```

- Return values are written to the EAX register.

How do we access the stack from the interrupt handler?

- There will be a `struct intr_frame` that is accessible from inside of the interrupt handler.

- Useful members:

    - `esp`: Stack pointer

    - `eax`: Value that is written to `eax` register on exit from interrupt handler.

- Our expectation is that you won't write repetitive code to handle arguments. Hint: you can exploit the constant word size of pointers/integers on the stack.

How do we make *different* syscalls?

- All syscalls go through the same handler
- From a code cleanliness perspective, *please do not implement all of your syscall functionality in one big if-else statement!!!*
- Use a table or case statement to call to the correct call.

Checkpoint 1

Due 7/20/15

What do you need to do?

- Make argument parsing work
- Implement the three simple syscalls:
    - int null(int i)
    - int write(int fd, const void * buffer, unsigned size)
    - void exit(int status)

All other syscalls are needed for the final checkpoint

Nit: Accessing user memory

- Syscalls will frequently need to access data stored in the user memory space (0x0–0xC0000000)
- However, we can't always guarantee that a pointer provided by a user will be valid.
    - Could be a null pointer.
    - Could be a pointer that the process shouldn't have permission to access (e.g., write to read only memory).
- If we don't check the validity of a user pointer before accessing it and our access fails, what happens?

Handling these accesses

- Two general strategies. There may be more approaches…:

1. Check if pointer is valid (not null, is accessible, etc).

2. Check if pointer is in user space and dereference. If an invalid pointer, page fault will occur. Handle page fault at runtime.

- Make sure not to leak resources! This is more difficult with strategy 2, but strategy 2 is faster.

Simple File System in Pintos

Syscalls & the FS

- Syscalls allow user programs to access the file system, and also depend on the file system for *loading* user programs.

- We've provided a complete file system, albeit with some limitations:

    - Single root directory (implemented as a file) with no subdirectories

    - File names <14 characters

    - No repair tool —> crashes may not be recoverable

More subtle (?) limitations

- File system is *not* internally synchronized. What happens if two processes access a single file? How can we prevent file system synchrony bugs *without* modifying the file system?

- File size is fixed at creation time, and files are required to be a single extent —> contiguous range of sectors. What can happen over time if many files are created?

If you are modifying code in the file system, you are likely doing something unnecessary.

What will you need to add *around* the file system?

- Synchronization!
- Tracking of file handles:
  - Do you want to do this globally, per process, etc?
  - Does your solution require synchronization?
  - Does your solution incur high memory overhead?

Waiting

How to wait

- We expect the wait syscall to be the hardest system call to implement
- Why?
  - A parent can wait for child processes in any order
  - A parent can wait for children who have already completed (and these waits must succeed…)
  - Wait calls can fail for many reasons
  - Wait requires modifying exit
  - A parent cannot wait on a grandchild

How to wait (con't)

- Your approach will vary widely depending on what data structures you choose
- General guidance:
  - Implement `process_wait` correctly, and use this to implement the `wait` syscall
  - Wait should not require global synchronization, disabling interrupts, or calling thread_block
  - Wait should not require a global search of all processes.