Project 2: User Programs

Abdelmounaam Rezgui

Acknowledgment: The content of some slides is partly taken from Josh Wiseman's presentation

Overview

- Objective: Enable user programs to run and interact with the OS via system calls
- Directories:
  - userprog/
  - threads/
  - examples/

Using the File System -1-

- NOT the focus of this project
- User programs are loaded from the FS
- Many system calls deal with the FS (e.g., open(), read(), write())
- Simple file system provided in the filesys directory
  - Look at: filesys.h and file.h
  - Limit is 16 files
  - No subdirectories

Using the File System -2-

- Create a (simulated) disk:
  - pintos-mkdisk <img-file> <size>
    - img-file = filename for disk (usually "fs.dsk")
    - size = disk capacity, MB
- Format disk
  - pintos -f -q
    - -f = format, -q = quit after boot
- Copy to disk:
  - Pintos -p <source> -a <dest> -- -q
    - -p = source file, -a = filename on disk
- Run program:
  - Pintos run <executable>
- $ make check: will build a disk for you

Sample user programs

- In examples/
  - cat, echo, halt, hex-dump, ls, shell
- You should be able to write and run your own programs
- Create a test disk
- Copy programs to the test disk

Requirements

1) Argument passing
2) System calls
3) Process termination
4) Deny writes to executables
5) DESIGNDOC

Getting Started

- In the default version, programs will crash
- To get simple programs to run:
  *esp = PHYS_BASE – 12;
- This will NOT make argument passing work

Argument Passing -1-

- Motivation:
  - kernel creates first process
  - one process creates another
- $ pintos run 'pgm alpha beta'
- You may:
  - use strtok_r() to parse the command line
  - assume cmd line length < 4KB

pgm.c

```
main(int argc,
     char *argv[]) {
     ...
  }
```

$ pintos run 'pgm alpha beta'

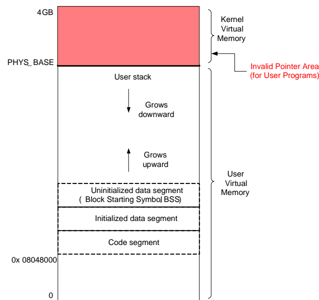
argc = 3

argv[0] = "pgm"

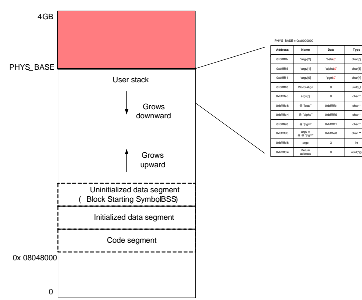
argv[1] = "alpha"

argv[2] = "beta"

Argument Passing -2-



Argument Passing -3-



Argument Passing -4-

Argument Passing -5-

bffffd0 00 00 00 00 03 00 00 00 e0 ff ff bf ....

bfffffe0 f1 ff ff bf f5 ff ff bf fb ff ff bf 00 00 00 00 ....

bffffff0 00 70 67 6d 00 61 6c 70 68 61 00 62 65 74 61 00 .pgm.alpha .beta.

System Calls -1-

- User programs make system calls
  - E.g., open(), close(), exit(), halt(), ...
- Implement system calls:
  - Process control:
    - exit(), exec(), and wait()
  - Filesystem calls:
    - create(),open(),close(),read(),write(),seek(), tell(),filesize(),remove()
  - halt()

13

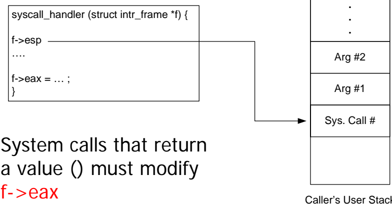
System Calls -2-

- The OS deals with software exceptions (called "internal interrupts" in the x86)
- SEs are events that occur in program code. They may be:
  - Errors: e.g., page faults, division by 0
  - System Calls
- SEs don't execute in interrupt context
  - i.e., intr_context() == false
- In the 80x86 arch, the 'int' instruction is used to make system calls
- In Pintos, user programs invoke 'int $0x30' to make system calls

14

System Calls -3-

- A system call has:
  - System call number
  - (possibly) arguments
- System call numbers are in: lib/syscall-nr.h
- When syscall_handler() gets control:



- System calls that return a value () must modify f->eax

15

System Calls -4-

- Filesystem calls
  - You must decide how to implement file descriptors
    - O(n) data structures for file descriptors are OK
  - For this project, access granularity is the entire file syst. Use ONE lock for the entire file system
- write(1, ...) writes to the console. Use putbuf().
- read(0, ...) reads from the stdin (keyboard). Use kbd_getc().

16

System Calls -5-

- exec(const char *cmd_line)
  - Runs 'cmd_line', passes args, returns pid
- exit() retains error codes for wait()

```
main() {
pid_t p;
p = exec("cp file1 file2");
...
}
```

17

System Calls -6-

- wait(pid_t pid)
  - Waits for process pid to die and returns the status that pid returned to exit()
  - Returns -1 if
    - pid was killed by the kernel
    - pid is not a child
    - Wait has already been successfully called

18

System Calls -7-

- Parent may or may not wait for its child
- Parent may call wait() after child terminates!
- Implement process_wait() in process.c
- Then, implement wait() in terms of process_wait()
- Conditions and semaphores will help
  - Think about what semaphores may be used for and how they must be initialized

```
main() {
int i; pid_t p;
p = exec("pgm a b");
i = wait (p);
... /* i must be 5 */
}
```

```
main() {
int status;
... status = 5;
exit(status);
}
```

pgm.c

19

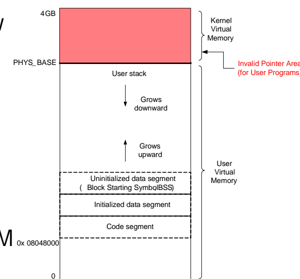
Process Termination

- Record the argument passed to the exit() syscall
- When a user process exits:
  printf("%s: exit(%d)\n",...)
- ALL programs call exit() unless, of course, if they're terminated
  - Returning from main implicitly calls exit()
  _start() is { exit(main(...)); }

20

Virtual Memory Layout -1-

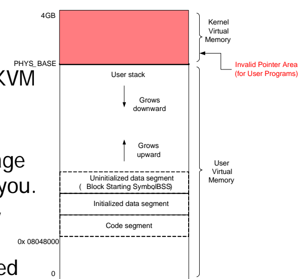
- Stack does NOT grow until Project 3
- Heap never grows
- Uninitialized means "zero-initialized"
- UVM is per-process
- A user program accesses only its UVM



21

Virtual Memory Layout -2-

- If it attempts to access KVM → Page fault
- Kernel threads access KVM and the UVM of the running user process
- In this project, this image is ALREADY set up for you.
- You only have to query the page table to see which pages are mapped



22

Memory Access -1-

- Kernel needs to access memory through pointers given by a user program
- user-provided pointers may be invalid
  - point to unmapped VM
  - point to KVM address space
- How to handle this ?

23

Memory Access -2-

- Two options:
  - verify the validity of a user-provided pointer, then dereference it –
    STRONGLY RECOMMENDED!
  - dereference and handle during page fault
    - Check only that user pointer points below PHYS_BASE
    - Dereference it
    - If it causes a page fault, handle it
    - You need to modify the code for page_fault()

24

Memory Access -3-

- In BOTH cases:
  - Graceful termination
    - Misbehaving processes must be killed
    - Orderly shutdown
  - No resource leaks
    - E.g., release locks, free allocated memory pages, etc.
- Data may span page boundaries

Denying Writes to Executables

- You may use:
  - file_deny_write() to prevent writes to an open file
- file_allow_write() re-enables them (if no other process has already denied them)
- If a file is closed, writes are re-enabled.

Misc

- Read Section 4.2 (page 51) (Suggested Order of Implementation) in Pintos documentation
- Do not forget the Design Document
- Good Luck !