CS 140 Project 2: User Programs

Fall 2015
Important Dates

• Project 1: DUE TODAY (Friday, October 9\textsuperscript{th}) at 12pm
• Project 2 will be due Friday, October 23\textsuperscript{rd} at 12pm
  – 2 weeks
  – Start early 😊
Overview

• Project 2: User programs
  – Build support for running user processes

• Requires good understanding of:
  – Steps for running a user program
  – Distinctions between user and kernel virtual memory
  – System call interface and handling
  – Kernel file system interface
User Programs

• Suppose a user types in the command
  
  \% cp –r fromDir toDir

• What happens?
% cp -r fromDir toDir

• Step 1: shell parses input
  argc = 4; argv = {“cp”, “-r”, “fromDir”, “toDir”}

• Step 2: shell calls fork() and execve(argv[0], argv, env)

• Step 3: cp uses system call interface to read/write files.

• Step 4: cp may print messages to stdout

• Step 5: cp exits
User Programs in Pintos

- **threads/init.c**
  - `run_actions()` → `run_task(argv)`
  - `run_task()` → `process_wait(process_execute(task))`

- **userprog/process.c:process_execute()**
  - creates thread running `start_process()`
  - thread loads executable file
  - sets up user virtual memory (stack, data, code)
    - User programs have NO heap/malloc
  - starts executing user process
User vs Kernel Virtual Memory
Starting User Processes

- lib/user/entry.c
  
  ```c
  void _start (int argc, char *argv[]) {
    exit(main(argc, argv));
  }
  ```

- Pass process start arguments

![Diagram showing stack pointer (esp) and function arguments](image)
Project 2 Requirements

• You will be implementing:
  – Argument passing
  – Safe memory access
  – System calls
  – Process exit messages
  – Deny write to in-use executable files
Argument Passing

• In preparation of starting a user process, the kernel must push the command line arguments onto the stack
• Break command line input into individual tokens
  – input: “cp -r fromDir toDir”
  – output: {“cp”, “-r”, “fromDir”, “toDir”}
• `strtok_r(…)` in `lib/string.c` will be helpful
Argument Passing (stack)

- push tokens (c-strings)
- push null sentinel
- push address of each token (right to left)
- push argv, push argc
- push return address (0)
Safe Memory Access

- Kernel virtual memory
- User Stack
  - user virtual memory
  - User Data
  - Init Data
  - User code

4 GB

PHYS_BASE

0
Safe Memory Access (cont.)

• The kernel will often access memory through user-provided pointers
• THIS IS DANGEROUS
  – null pointers
  – pointers to unmapped virtual addresses
  – pointers to kernel addresses
• Kill the process (free its resources, e.g. locks, memory)
• Be careful with buffers, strings, any other pointers
Safe Memory Access (cont.)

There are 2 approaches to solving this problem:

• Simpler Approach: Verify every user pointer before dereferencing
  – Ensure that it is in user address space (i.e. below PHYS_BASE)
  – Ensure it is mapped (look at userproglmage/page dir.c:pagedir_get_page())
  – For buffer, ensure for the entire buffer

• Modify fault handler in userproglmage/exception.c
  – Ensure pointer or buffer is below PHYS_BASE
  – Invalid pointers should trigger page faults
  – See 3.1.5 – Accessing User Memory for more details
System Calls: how do they work?!

- Works like normal function calls (args in stack)
- Execute internal interrupt (int instruction)
  - syscall_handler(struct intr_frame *f)
- Stack pointer (f->esp) at syscall number
- Calling thread data available
  - to pass args to handler
  - to return value to user process
- Return value just like functions (f->eax)
System Call: Implementation

• userprog/syscall.c:syscall_handler()
• Read syscall number at stack pointer
• Dispatch a particular function to handle syscall
• Read (validate!) arguments (above the stack pointer)
  – above the stack pointer
  – validate pointer and buffers!
• syscall numbers defined in lib/syscall-nr.h
• See 3.3.4 – System Calls for Project 2’s required calls
System Call: File System

• Many syscalls involve file system functionality
• Simple filesys impl is provided: filesystem.h, file.h
  – No need to modify it, but familiarize yourself
• File system is not thread-safe! (Project 4)
  – Use a coarse lock to protect it
• Syscalls take file descriptors as args
  – Pintos represents file with struct file *
  – You must design the mapping
• Special cases: reading from keyboard and writing to console
  – write(STDOUT_FILENO, ...) use putbuf or putchar
  – read(STDIN_FILENO, ...) use input_getc
System Call: Processes

• Generally these syscalls require the most design and implementation time
• `pid_t exec(const char *cmd_line)`
  – like UNIX `fork() + execve()`
  – Creates a child process
  – Must not return until a new process has been created (or creation failed)
System Call: Processes

• `int wait (pid_t pid)`
  – Parent must block until child process `pid` exits
  – Returns exit status of the child
  – Must work if child has ALREADY exited
  – Must fail if it has already been called on child before

• `void exit (int status)`
  – Exit with status and free resources
  – Process termination message
  – Parent must be able to retrieve status via `wait`
System Calls: Security

- How does a system recover from null pointer segfault in user program?
- How does a system recover from null pointer segfault in kernel?
System Calls: Security

• How does a system recover from null pointer segfault in user program?
  – kill user process, life goes on
• How does a system recover from null pointer segfault in kernel?
  – it (basically) doesn’t
• Verify all user-passed memory references (pointers, buffers, strings)
• Kill user program if passed illegal addresses
Denying writes to executables

- Executables are files like any other
- Pintos should not allow code that is currently running to be modified
  - use file_deny_write() to prevent writes to an open file
  - closing a file will re-enable writes
  - keep executable open as long as the process is running
Utilities: Making disks

- User executables must be on virtual hard disk
  
  cd pintos/src/userprog
  
  make
  
  pintos-mkdisk fs.dsk --filesys-size=2 /* Create 2MB disk */
  
  pintos --disk=fs.dsk -- -f -q /* Format disk */
  
  pintos --disk=fs.dsk -p ../examples/echo -a echo -- -q /* copy to disk */
  
  pintos --disk=fs.dsk -- -q run "echo x" /* run program */

- User code examples in src/examples

- You can write your own code to test
  
  – But you don’t have to
Utilities: GDB

• You can use GDB to debug user code
• Start GDB as usual, then do:
  \( \text{(gdb) loadusersymbols } < \text{userprog.o} \)  
• You can set breakpoints and inspect data as usual
• User symbols will not override kernel symbols 
  - Work around duplicate symbols by inverting order 
  - Run gdb with 
    \( \text{pintos-gdb } < \text{userprog.o} \)  
  - Then load kernel symbols with 
    \( \text{(gdb) loadusersymbols kernel.o} \)
Getting Started

• Make a disk and add simple programs
  – run make in src/examples

• Temporarily setup stack to avoid page faulting
  – In userprog/process.c:setup_stack()
    – Change *esp = PHYS_BASE
    – to *esp = PHYS_BASE - 12
    – This will allow running programs with no arguments

• Implement safe user memory access
Getting Started (cont.)

• Setup syscall dispatch
• Implement exit
• Implement write to STDOUT_FILENO
  – No tests will pass until you can write to the console
• Change process_wait(…) to an infinite loop
  – Stub implementation exits immediately
  – Pintos will power off before any processes can run
• Start early!
• Good luck!
## Start early!

<table>
<thead>
<tr>
<th>File Name</th>
<th>Lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>threads/thread.c</td>
<td>13</td>
</tr>
<tr>
<td>threads/thread.h</td>
<td>26</td>
</tr>
<tr>
<td>userprog/exception.c</td>
<td>8</td>
</tr>
<tr>
<td>userprog/process.c</td>
<td>247</td>
</tr>
<tr>
<td>userprog/syscall.c</td>
<td>468</td>
</tr>
<tr>
<td>userprog/syscall.h</td>
<td>1</td>
</tr>
</tbody>
</table>

6 files changed, 725 insertions(+), 38 deletions(-)