Week 6
System Call Programming

11 February 2019

CS 35L Lab 4

Jeremy Rotman

Announcements

- Assignment #5 is due Saturday by 11:55pm
- For Assignment #10
  - **Email me to tell me what story you are choosing**
  - Here is the link to see what stories people have signed up for already
    - Choose a story at least one week before you present
- The Final is on Sunday March 17 3-6pm, and is a common final
  - Let me know if this creates a schedule conflict ASAP

Outline

- ➔ Dual-Mode Operation
- ➔ System Calls
- ➔ Buffered vs. Unbuffered I/O
- ➔ Lab 5

Questions?

Processor Modes

- The CPU (in Linux) has two distinct modes of operation
  - Kernel mode
    - Unrestricted access
    - Can execute any instruction, and reference any memory address
    - Assumes it is running trusted software
  - User mode
    - Non-privileged access
    - Cannot directly access hardware
    - Must use a system call to perform privileged instructions

Why Dual-Mode Operation?

Why Dual-Mode Operation?

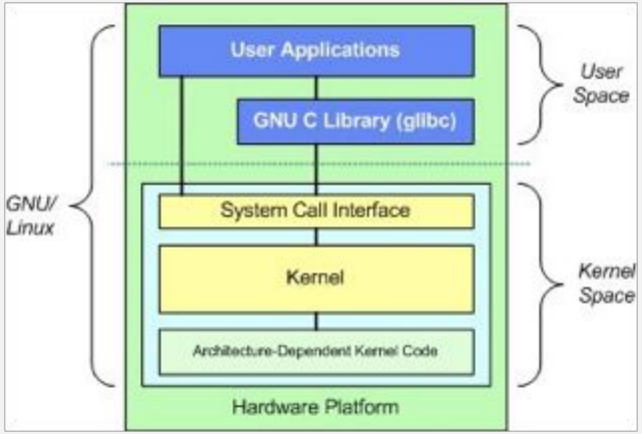
- I/O protection
  - Input/output cannot be directly controlled by user-created code
- Memory protection
  - User mode only has access to a set partition of memory
  - The user is not allowed to access memory addresses that define these bounds, or other addresses outside their partition
- CPU Protection
  - User mode is not allowed to change things related to the OS's scheduler or timer

Trusted Software

- ➔ What is the trusted software that can be run in kernel mode?

Trusted Software

- What is the trusted software that can be run in kernel mode?
  - Software in the kernel space
    - Cannot be changed from the outside
    - Implements protection mechanisms
- System call interface bridges the gap between User Mode and Kernel Mode
  - User processes can execute privileged operations through the interface

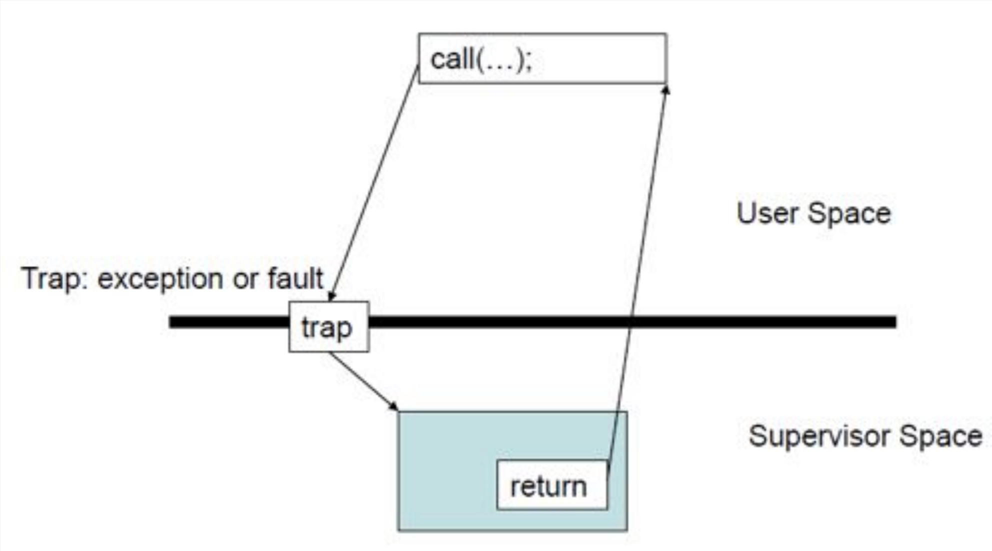


System Calls

- ➔ Used by user-level processes to request service from kernel
- ➔ Changes CPU's mode from User to Kernel
- ➔ Part of the Kernel of the OS
- ➔ Verifies User should be allowed to do operation, then handles operation
- ➔ The only way that a user program can perform privileged operations

System Calls

- When a system call is made
  - The program being executed is interrupted
  - Control is passed to the kernel
  - If the operation is valid
    - Kernel performs it
  - Else
    - Throw an exception



System Calls

- ➔ Overhead

System Calls

- Overhead
  - System calls include a lot of overhead
  - Many things that must be done
    - Process interrupted and computer saves its state
    - OS takes control of CPU and verifies validity of operation
    - OS performs requested action
    - OS restores saved context, switches to user mode
    - OS gives control of the CPU back to user process

Example System Calls

- ssize_t read(int *fildes*, void* *buf*, size_t *nbyte*)
  - *fildes*: file descriptor
  - *buf*: address of buffer to read into
  - *nbyte*: maximum number of bytes to read
- ssize_t write(int *fildes*, void* *buf*, size_t *nbyte*)
  - *fildes*: file descriptor
  - *buf*: address of buffer to read into
  - *nbyte*: maximum number of bytes to write
- int open(const char* *pathname*, int *flags*, mode_t *mode*)
- int close(int *fildes*)
- fd: 0 = stdin; 1 = stdout; 2 = stderr;

Example System Calls

- void exit(int *status*)
  - Terminates process with *status*
- pid_t fork(void)
  - Creates child process
- pid_t getpid(void)
  - Returns the process ID of the calling process
- int dup(int *fildes*)
  - Duplicates a file descriptor, *fildes*
- int fstat(int *fildes*, struct stat* *buf*)
  - Return information about the file with file descriptor *fildes*, into *buf*

C Library Functions

- ➔ There are functions that are part of the standard C library that do the same thing
  - getchar & putchar are similar to read & write (standard I/O)
  - fopen & fclose are similar to open & close (file I/O)
- ➔ These C functions then make system calls of their own
- ➔ What is the advantage?

C Library Functions

- There are functions that are part of the standard C library that do the same thing
  - getchar & putchar are similar to read & write (standard I/O)
  - fopen & fclose are similar to open & close (file I/O)
- These C functions then make system calls of their own
- What is the advantage?
  - Library functions make fewer system calls
  - This reduces the amount of overhead
  - Efficiency :D

Unbuffered vs. Buffered I/O

- Unbuffered
  - Every byte is read/written by the kernel through system call
- Buffered
  - Collect as many bytes as possible (in a buffer) and read/write them all in one system call
- Buffered I/O reduces overhead because you don't have to switch mode for every byte
  - Thought exercise: when might you want to use unbuffered I/O?
  - Also, you must still always be prepared for buffer overflow attacks

Lab 5

- Write C code to transliterate bytes
  - 2 arguments:
    - *from*: the bytes to transliterate
    - *to*: the bytes to transliterate them into
  - Works basically the same way as tr
    - tr ‘abcd’ ‘nopq’ < input_file
      - Replace ‘a’ with ‘n’, ‘b’ with ‘o’, etc.
- You will write 2 programs (one buffered, one unbuffered)
  - tr2b
    - Uses getchar and putchar to read from stdin and write to stdout
  - tr2u
    - Uses read and write with the nbyte argument as 1

Lab 5

- The programs will need to be tested on a very large file
  - At least 5,000,000 bytes
  - head --bytes=*num* /dev/urandom > output.txt
- **time** [options] *command* [arguments...]
  - Output:
    - real 0m0.145s
      - Elapsed real time (in [hours:]minutes:seconds)
    - user 0m0.001s
      - Total number of CPU-seconds that process spent in user mode
    - sys 0m0.003s
      - Total number of CPU-seconds that process spent in kernel mode

Lab 5

- ➔ **strace** -o *strace_output command* [arguments…]
  - Intercepts and prints out system calls to stderr or to an output file
  - Every line will include
    - system call name
    - arguments in parentheses
    - return value

Questions?