

Today

- Debugging Process
- Operating System: Protection
- System Calls

Sept 25, 2015

Sprenkle - CSC330

1

Review

- How many predefined streams are there?
 - What are they called? What do they do?
- How do we redirect I/O on the command-line?
- How do we represent streams in C?

Sept 25, 2015

Sprenkle - CSC330

2

What is your debugging process?

DEBUGGING

Sept 25, 2015

Sprenkle - CSC330

3

Debugging as Engineering

- Much of your time in this course will be spent debugging
 - In industry, **50%** of software dev is debugging
 - Even more for kernel development
- How do you reduce time spent debugging?
 - Produce working code with smallest effort
- Optimize a process involving you, code, computer

Sept 25, 2015

Sprenkle - CSC330

4

Debugging as Science

- Understanding -> design -> code
 - not the opposite
- Form a hypothesis that explains the bug
 - Which tests work, which don't? Why?
 - Add tests to narrow possible outcomes – what's the minimal input required to fail the test & reproduce the bug?
- Use best practices
 - Always walk through your code line by line
 - Unit tests – narrow scope of where problem is
 - Develop code in stages, with dummy stubs for later functionality

Sept 25, 2015

Sprenkle - CSC330

5

OPERATING SYSTEMS: PROTECTION

Sept 25, 2015

Sprenkle - CSC330

6

OS Challenges

- Reliability
 - Does the system do what it was designed to do?
- Availability
 - What portion of the time is the system working?
 - Mean Time To Failure (MTTF), Mean Time to Repair
- Security
 - Can the system be compromised by an attacker?
- Privacy
 - Data is accessible only to authorized users

Sept 25, 2015

Sprengle - CSC330

7

OS Challenges

- Portability
 - For programs:
 - Application programming interface (API)
 - Abstract virtual machine (AVM)
 - For the operating system
 - Hardware abstraction layer

Sept 25, 2015

Sprengle - CSC330

8

OS Challenges

- Performance
 - Latency/response time
 - How long does an operation take to complete?
 - Throughput
 - How many operations can be done per unit of time?
 - Overhead
 - How much extra work is done by the OS?
 - Fairness
 - How equal is the performance received by different users?
 - Predictability
 - How consistent is the performance over time?

Sept 25, 2015

Sprengle - CSC330

9

Key OS Challenge: Protection

- Isolate misbehaving applications and users
 - From other applications, OS
- How do we execute code with restricted privileges?
 - Either because the code is buggy or if it might be malicious
- Some examples:
 - A script running in a web browser
 - A program you just downloaded off the Internet
 - A program you just wrote that you haven't tested yet

Sept 25, 2015

Sprengle - CSC330

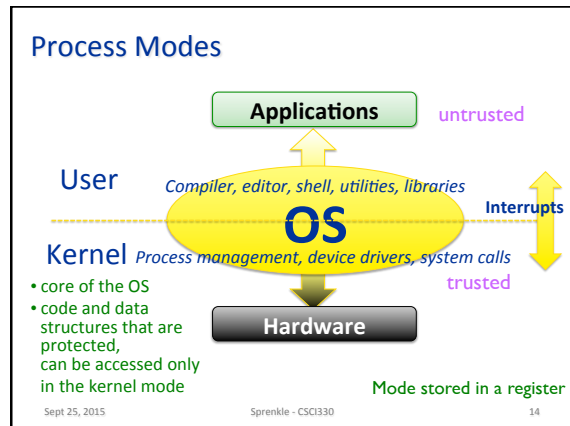
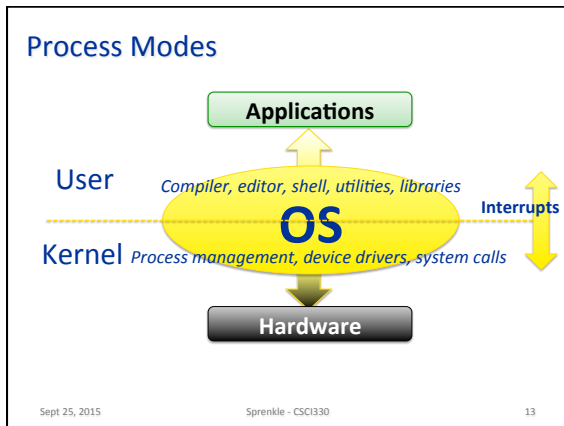
10

Tradeoffs in Protection

- How can we implement execution with limited privilege?
 - Execute each program instruction in a simulator
 - If the instruction is permitted, do the instruction
 - Otherwise, stop the process
 - Basic model in JVM and other interpreted languages
- How do we go faster?
 - Run the unprivileged code directly on the CPU!

Process Abstraction

- OS abstraction for executing a program with limited privileges
- Process: an *instance* of a program, running with limited rights
 - Thread: a sequence of instructions within a process
 - Potentially many threads per process (for now 1:1)
 - Address space: set of rights of a process
 - Memory that the process can access
 - Other permissions the process has (e.g., which system calls it can make, what files it can access)



- ### Processor Modes
- Modern processors typically can operate in 2 modes: user mode and kernel mode
 - User mode
 - processor executes normal instructions in the user's program.
 - Kernel mode
 - processor executes both normal and privileged instructions
 - Processor can access additional registers and memory address space that are accessible only in kernel mode
- How do we switch from one mode to the other safely?
- Sept 25, 2015 Sprenkle - CSC330 15

- ### Common Functions of Interrupts
- An operating system is **interrupt-driven**
 - Sits, waiting for something to happen
 - A **trap** or **exception** is a software-generated interrupt caused by an error or a user request
 - "Hey! Look at me! I'm ready to do something!"
 - "Oopsies! I divided by 0!"
-
- Sept 25, 2015 Sprenkle - CSC330 16

Exceptions: trap, fault, interrupt

	intentional happens every time	unintentional contributing factors
synchronous caused by an instruction	trap: system call open, close, read, write, fork, exec, exit, wait, kill, etc.	fault invalid or protected address or opcode, page fault, overflow, etc.
asynchronous caused by some other event	"software interrupt" software requests an interrupt to be delivered at a later time	interrupt caused by an external event: I/O op completed, clock tick, power fail, etc.

Oct 7, 2015 Sprenkle - CSC330 17

- ### How do we take interrupts safely?
- Interrupt vector
 - Limited number of entry points into kernel
 - Atomic transfer of control
 - Single instruction to change:
 - Program counter
 - Stack pointer
 - Memory protection
 - Kernel/user mode
 - Transparent restartable execution
 - User program does not know interrupt occurred
- Sept 25, 2015 Sprenkle - CSC330 18

SYSTEM CALLS & LIBRARIES

Sept 25, 2015

Sprenkle - CSC330

19

System Calls

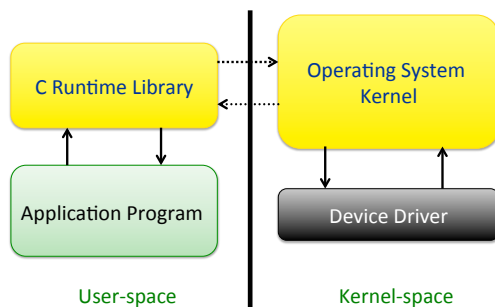
- User programs are not allowed to access system resources directly
 - must ask OS to do that on their behalf
- System calls: set of functions for user programs to request for OS services
 - Run in kernel mode
 - Invoked by special instruction (trap/interrupt) causing the kernel to switch from user mode
 - When the system call finishes, processor returns to the user program and runs in user mode.

Sept 25, 2015

Sprenkle - CSC330

20

How system calls work



Sept 25, 2015

Sprenkle - CSC330

21

How Process Works

1. Interrupt transfers control to the **interrupt service routine (ISR)**
 - ISR is part of BIOS or OS
 - Generally, transferred through the **interrupt vector**, which contains the addresses of all the service routines
2. Interrupt architecture must save the address of the interrupted instruction
3. Figure out which system call made
4. Verify parameters
5. Execute Request
6. Back to the calling instruction.

Sept 25, 2015

Sprenkle - CSC330

22

Vectored Interrupts

- Each device is assigned an **interrupt request number (IRQ)**.
- The device's IRQ is used as an index into the **interrupt vector**
 - The value at each index is the address of the ISR associated with the interrupt.
- The value from the interrupt vector is loaded into the PC

Sept 25, 2015

Sprenkle - CSC330

23

Libraries & System Calls

- Many standard C library functions use system calls
- Example: the `malloc()` C library function uses system calls
 - `brk()`: grab some more heap space by moving upper bound of heap
 - `mmap()`: find new large chunk of addressable memory space

Sept 25, 2015

Sprenkle - CSC330

24

System Calls: Files & I/O

Commands	Purpose
open, close	open and close a file
create, unlink	create and remove a file
read, write	read and write a file
lseek	move to a specified byte in a file
chmod	change access permission mode
mkdir, rmdir	make and remove a directory
stat	get file status
ioctl	control device

Sept 25, 2015

Sprenkle - CSC330

25

System Calls: Process Management

- fork create a new process
- exec execute a file
- exit terminate process
- wait wait for a child process to terminate
- sbrk change the size of the space allocated in the heap data segment of the process (used by memory allocation functions, e.g. malloc)

Sept 25, 2015

Sprenkle - CSC330

26

System Calls: Interprocess Communication

- kill, sigsend send a signal to a process
- pause suspend process until signal
- sigaction set signal handler

Sept 25, 2015

Sprenkle - CSC330

27

Library functions vs System Calls

- C has predefined library functions with the same names as the system calls – mostly wrapper functions
- System calls run in kernel-mode but library functions run in user-mode and may call system calls
- Relatively small number of system calls -- about 250 in Linux, see /usr/include/asm/unistd.h

Sept 25, 2015

Sprenkle - CSC330

28

Why use libraries?

- **Convenience:** hide underlying system details & easier to repeatedly call function
- **Difficulty:** Avoid coding difficult or optimized algorithms that are well-established (sin, cos)
- **Portability:** different systems may use slightly different system calls to implement the same functionality – this is all taken care of by the hardware-specific C libraries

Sept 25, 2015

Sprenkle - CSC330

29

System Call Instructions

- A process makes a system call by executing a special machine/assembly language instruction:
➤ e.g., SYSCALL TRAP SC INT
- Usually you do not see the system call instruction because it is wrapped inside a language library (Java / C, C++ / etc.)
- Conceptually, a system call is like a function call to a function that is part of the operating system.
➤ The mechanism is just a little different

Sept 25, 2015

Sprenkle - CSC330

30

System Call Parameters

- System Call parameters can be passed to the OS in three ways:
 - On the system stack
 - In registers
 - In a block of memory
- Different techniques are used for different system calls and even for individual parameters of the same system call.
 - E.g. Writing to a file.
 - The file to write is usually indicated by an integer passed in a register.
 - The data to be written is passed using a pointer to block of memory (the pointer can be passed in a register).

Sept 25, 2015

Sprenkle - CSC330

31

Looking Ahead

- Assign0b
 - 1-Pointers.txt
 - Practice pointers
 - More prepared for the code I gave you
 - Due Monday
- Moving toward Project 1
 - Booting!!

```
char** argv  
*argv  
*argv[i]
```

Sept 25, 2015

Sprenkle - CSC330

32