CS5460: Operating Systems

Lecture 3: OS Organization

(Chapters 2-3)

Last Time

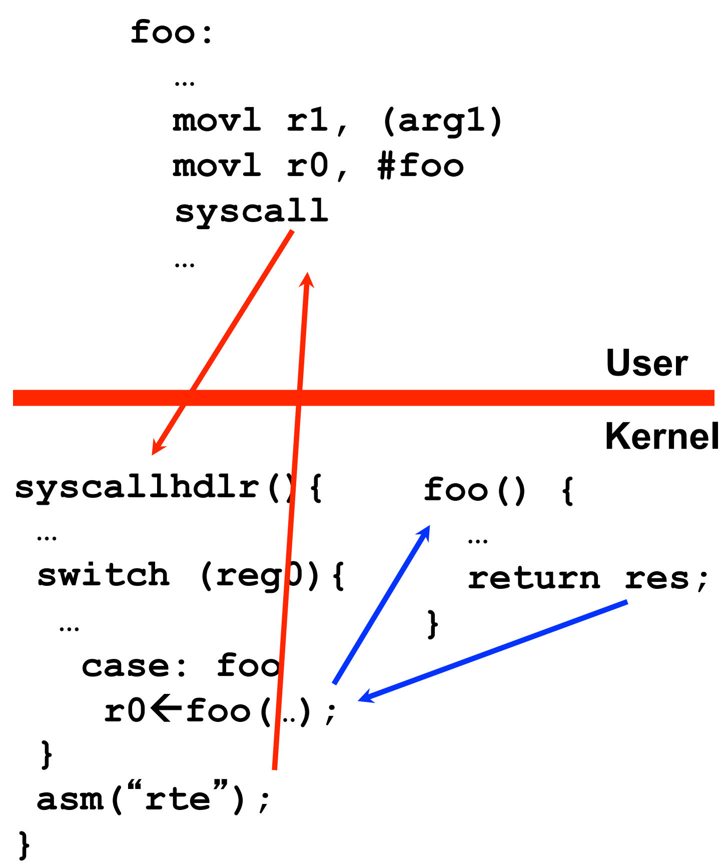
- Generic computer architecture
  - Lots of pieces
  - OS has to manage all of them
- Processor modes
  - OS executes with the CPU in kernel mode
  - User programs execute with the CPU in user mode
  - Kernel mode code is trusted – if it is bad, the whole OS is bad
- Main OS Goal: Provide the "process model"
  - Dynamically created virtual address spaces + virtual CPUs
  - Processes are isolated by default
  - System call mechanism pokes a hole in the firewall

Last Time Continued

- **What are the 4 basic ways in which the processor can start executing in kernel mode?**
  - **Boot**
  - **System call**
  - **Other trap**
  - **Interrupt**

Anatomy of a System Call

- **User apps make system calls to execute privileged instructions**
- **Anatomy of a system call:**
  - **Program puts syscall params in registers**
  - **Program executes a trap:**
    - » **Minimal processor state (PC, PSW) pushed on stack**
    - » **CPU switches mode to KERNEL**
    - » **CPU vectors to registered trap handler in the OS kernel**
  - **Trap handler uses param to jump to desired handler (e.g., fork, exec, open, …)**
  - **When complete, reverse operation**
    - » **Place return code in register**
    - » **Return from exception**



Last Time Continued

- **System call**
  - Arguments passed in registers
  - Return code passed in register
  - Usually, there are side effects in kernel state or process's memory
  - Should operate correctly for all possible inputs
    - » Why?
    - » How would you test this?
  - Should be semantically simple

Today

- More traps
- Device I/O
- Interrupts
- Introduction to processes
  – What are they?
  – Where do they come from?
  – How do they relate to I/O?
- The process abstraction and how it is built is one of the main topics of this class
  – Every user program runs inside a process
  – Every system call comes from some process
  – The kernel is not a process

Traps

- Architecture detects special events:
  - trap request – `syscall, int`
  - read or write invalid memory access
  - divide by zero
  - privileged instruction by user mode code
  - …
- When processor detects condition:
  - Save minimal CPU state (PC, sp, …) – done by hardware
  - Switches to KERNEL mode
  - Transfers control to trap handler
    - » Indexes trap table w/ trap number
    - » Jumps to address in trap table (*)
    - » Handler saves more state and may disable interrupts
  - `RTE`/`IRET` instruction reverses operation

TRAP VECTOR:

0x0082404	Illegal address
0x0084d08	Mem Violation
0x008211c	Illegal instruction
0x0082000	System call
…	

Here, 0x82404 is address of `handle_illegal_addr()`.

Controlling I/O Devices

- Hardware is controlled using device registers
  - CPU can read/write device registers
  - Device drivers read/write registers to control device
    - » Memory-mapped I/O: registers mapped to special addresses
    - » Programmed I/O: special instructions to read/write registers
  - Registers may look like memory but they don't act like it!
- DMA: Direct Memory Access
  - Modern I/O devices can directly read/write system memory
  - OS manages "DMA channels" to control memory device can access
- Device signaling: Polling vs Interrupts
  - Polling: OS "polls" devices to see if they need attention
  - Interrupts: Devices signal OS when they need attention

Polling and Interrupts

- **I/O is concurrent with main processor**
  - CPU initiates I/O with I/O register writes
  - CPU detects I/O completion/signal via:
    - » Interrupt (async hardware signal)
    - » Polling (loop reading I/O register)
  - Question: Polling vs interrupts - when?
- **Interrupt raises signal on CPU pin**
  - Each device configured to use a particular interrupt number
  - Usually, CPU "traps" to the appropriate interrupt handler next cycle
  - Can selectively mask interrupts (not traps!)
- **Interrupts can cost performance**
  - Flush CPU pipeline + cache/TLB misses
  - Handlers often need to disable interrupt

INTERUPT VECTOR:

0x008c408	Clock
0x0088044	Disk
0x008317c	Mouse
0x0089f0c	Keyboard
…	

Issues with Interrupts

- **Interrupt overload**
  - Some devices can generate interrupts faster than CPU can handle
  - Example: “receiver livelock” in high speed networks
  - Solution: buffering, adapt between polling and interrupts
- **Interrupts on PCs go through external interrupt controller**
  - Can be many sources of interrupts
  - Interrupt may be shared between devices
    - » Question: How can this be done?!?
  - Embedded CPUs often have much nicer interrupt subsystems than PCs do

Issues with Interrupts

- **What stack do interrupts use?**
- **What process is running when an interrupt arrives?**
- **Good manners for interrupt handlers:**
  - **When invoked, perform all work associated with device**
  - **Do not disable interrupts very long (e.g., up to 100usec)**
- **Interrupts can be very hard to get right**
  - **Concurrency is hard**
  - **Standard debugging techniques may not work**

Initializing Traps/Interrupts

- Vectors pinned at known physical addresses
  - Location specified by CPU vendor or configurable w/ register

- Initialized (carefully!) during boot process

```
// interrupts disabled on boot
…
intr_vector[0] = (void *) handle_clock_int;
intr_vector[1] = (void *) handle_disk_int;
…
enable_interrupts();

void handle_clock_int() { … };
```

Traps vs Interrupts

- **Traps are synchronous**
  - Generated inside the processor due to instruction being executed
  - Instructions may
    - » Always trap – example?
    - » Sometimes trap – example?
    - » Never trap – example?
  - Cannot be masked
  - System calls are one kind of trap
- **Interrupts are asynchronous**
  - Generated outside the processor
  - Can be masked

Quick Review

- **System calls:**
  – Arguments places in well-known registers
  – Perform trap instruction → vector to system call handler
    » Low level code carefully saves cpu state
    » Processor switches to protected/kernel mode
    » Syscall handler checks param and jumps to desired handler
  – Return from system call
    » Result placed in register and low level code restores state
    » Perform "rte" instruction: switches to user mode and returns to location where "trap" was called
- **OS manages trap/interrupt tables**
  – Controls the "entry points" in the kernel → secure
  – Traps are synchronous; interrupts are asynchronous

Intro to Processes

- **How are OS and I/O protected from user processes?**
- **Spatial protection**
  - **Memory protection: OS and I/O registers mapped into protect memory (supervisor-only)**
  - **Privileged instructions: User processes may not be allowed to perform I/O ops**
- **Temporal protection**
  - **Timer interrupts keep user processes from hogging CPU**
- **Traps allow user processes to break through protection barrier:**
  - **BUT OS controls entry points into kernel**
  - **Most system calls vectored to a single system call handler**
    - » **Parameter to system call specifies what operation desired**

What's a Process?

- **Process: execution context of running program**
- **A <u>process</u> does not equal a <u>program</u>!**
  - **Process is an *instance* of a program**
  - **Many copies of same program can be running at same time**
- **OS manages a variety of activities**
  - **User programs**
  - **Batch jobs and scripts**
  - **System programs (e.g., print spool, file server, network daemons, …)**
- **Each of these activities is encapsulated in a process**
- **Everything that happens is either in the OS or in a process**
  - **Again, the OS is not a process**

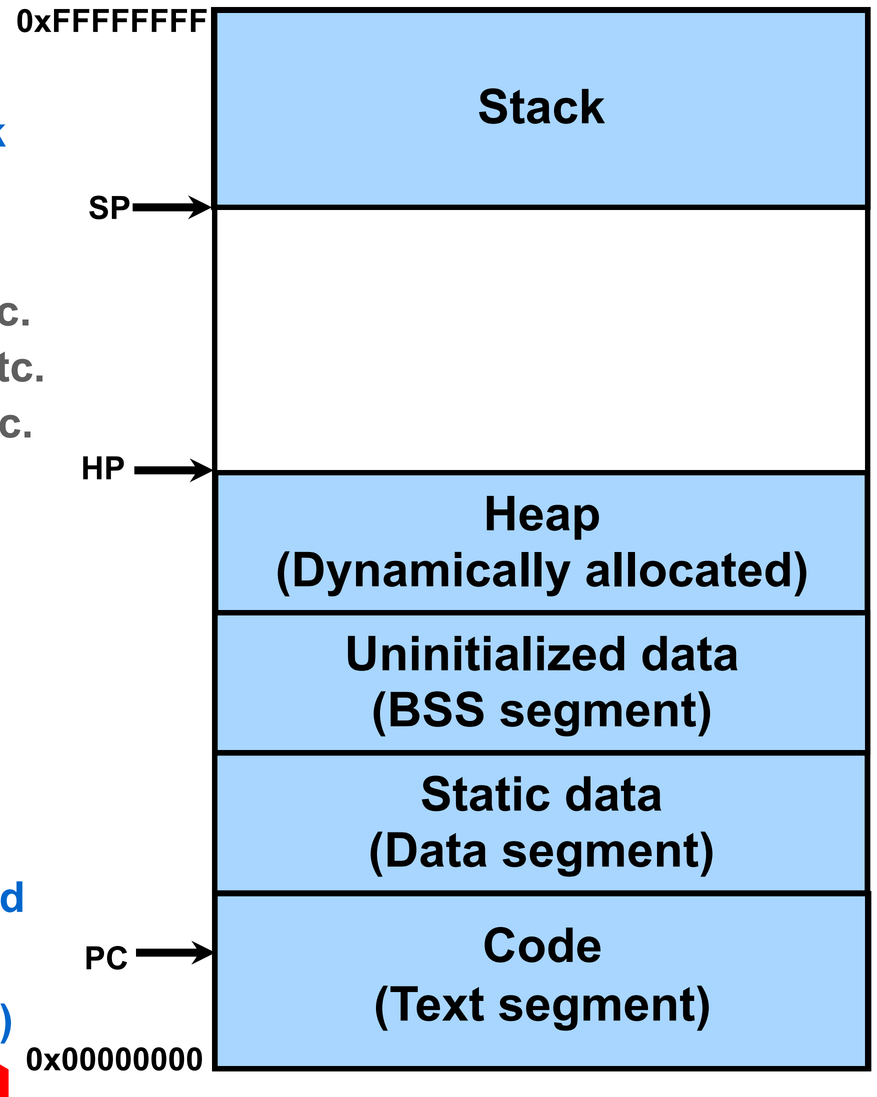
Process Management

- **OS manages processes:**
  - **OS creates, deletes, suspends, and resumes processes**
  - **OS schedules processes to manage CPU allocation ("scheduling")**
  - **OS manages inter-process communication and synchronization**
  - **OS allocates resources to processes (and takes them away)**

- **Processes use OS functionality to cooperate**
  - **Signals, sockets, pipes, files to communicate**
  - **Synchronization for resources modified by multiple processes**

What's in a Process?

- Process state consists of:
  - Memory state: code, data, heap, stack
  - Processor state: PC, registers, etc.
  - Kernel state:
    - » Process state: ready, running, etc.
    - » Resources: open files/sockets, etc.
    - » Scheduling: priority, cpu time, etc.
- Address space consists of:
  - Code
  - Static data (data and BSS)
  - Dynamic data (heap and stack)
  - See: Unix "`size`" command
- Special pointers:
  - PC: current instruction being executed
  - HP: top of heap (explicitly moved)
  - SP: bottom of stack (implicitly moved)



Practical Process Management

- On a Unix machine, try:
  - `ps -Af` : lots of info on all running processes
  - `kill -9 547` : immediately terminates process with PID 547
  - `top` : displays dynamic info on top running jobs
  - Write a program that calls:
    - » `getpid()`: returns current process's PID
    - » `fork()`: create a new process
    - » exec(): load a new program into the current process
    - » `sleep()`: puts current process to sleep for specified time
- Similar commands work on Mac OS X (BSD Unix!)
- On Windows → Task manager (CTL-ALT-DEL)

Processes and Interrupt-Driven IO

What you see is: 

What the OS sees is:

1. Cursor control process blocked on mouse device (e.g., /dev/cua0)
2. User moves mouse
3. Mouse hardware generates interrupt
4. If interrupts on, CPU saves state and jumps to `handle_mouse_int()`
5. `handle_mouse_int()` pushes some registers on stack, reads position delta from mouse controller
6. `handle_mouse_int()` checks to see if a process is waiting on device
7. `handle_mouse_int()` asks OS to wake up cursor-control process and returns
8. CPU scheduler evaluates priority of runnable processes:
   - Is cursor control process higher?
   - If so, switch to cursor control proc
9. Cursor control process computes new mouse position, sends cmd to video driver to update screen
10. Video driver updates screen
11. Cursor control process goes to sleep (assuming no new movement)

Important From Today

- Trap (synchronous)
- Interrupt (asynchronous)
- Interaction with devices through
  - Device registers
  - Interrupts
  - DMA
- Processes
  - This is a big part operating system's job
  - This is a big part of this class
  - Every system call comes from some process
- Boot sequence
- Flow of control when a process does I/O