CSE451 Architectural Supports for Operating Systems
Spring 2001

Gary Kimura

Lecture #2

March 28, 2001

Today

- What hardware support does the OS use?
- But before we start, some clarifications
  – The book "Understanding the Linux Kernel" is NOT required
  – If I'm going too fast or my terminology is foreign to you then please STOP me

OS and Architectures

- What an OS can do is dictated, at least in part, by the architecture.
- Architecture support can greatly simplify (or complicate) OS tasks
- Example: Early PC operating systems have been primitive, in part because PCs lacked hardware support (e.g., for VM)

Architectural Features for OS

- Features that directly support OS needs include:
  - Timer (clock) operation
  - Synchronization (atomic instructions)
  - Memory protection
  - I/O control and operation
  - Interrupts and exceptions
  - OS protection (kernel/user mode)
  - Protected instructions

Protected Instructions

- Some instructions are typically restricted to the OS
  - Users cannot be allowed direct access to I/O (disks, printers, etc) [can be done through either privileged instructions or through memory mapping]
  - Must control instructions that manipulate memory management state (page table pointers, TLB load, etc)
  - Setting of special mode bits (kernel mode)
  - Halt instruction

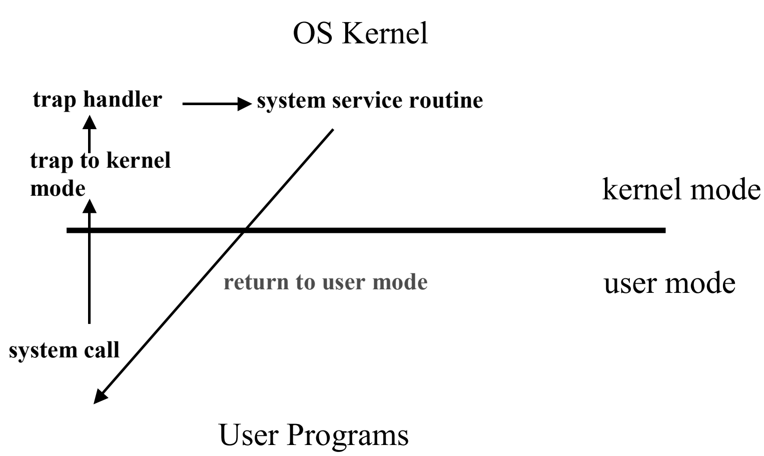
OS Protection

- How do we know if we can execute a protected instruction?
  - Architecture must support (at least) two modes of operation: *kernel* mode and *user* mode
  - Mode is indicated by a status bit in a protected processor register
  - User programs execute in user mode; the OS executes in kernel mode
- Protected instructions can only be executed in kernel mode.

Crossing Protection Boundaries

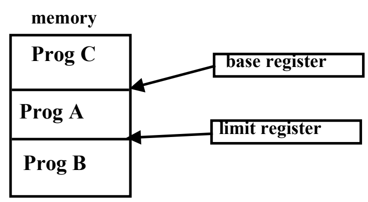
- For a user to do something “privileged” (e.g., I/O) it must call an OS procedure.
- How does a user-mode program call a kernel-mode service?
- There must be a system call instruction that:
  - Causes an exception, which vectors to a kernel handler
  - Passes a parameter, saying which system routine to call
  - Saves caller’s state (PC, mode bit) so it can be restored
  - Architecture must permit OS to verify caller’s parameters
  - Must provide a way to return to user mode when done

Protection Modes and Crossing



Memory Protection

•Must be able to protect user programs from each other
•Must protect OS from user programs
•May or may not protect user programs from OS
•Simplest scheme is base and limit registers:



base and limit registers are loaded by the OS before starting a program

virtual memory and segmentation are similar

Exceptions

- Hardware must detect special conditions: page fault, write to a read-only page, overflow, trace trap, odd address trap, privileged instruction trap, syscall...
- Must transfer control to handler within the OS
- Hardware must save state on fault (PC, etc) so that the faulting process can be restarted afterwards
- Modern operating systems use VM traps for many functions: debugging, distributed VM, garbage collection, copy-on-write...
- Exceptions are a performance optimization, i.e., conditions could be detected by inserting extra instructions in the code (at high cost)

I/O Control

- I/O issues:
  - How to start an I/O (special instructions or memory-mapped I/O
  - I/O completion (interrupts)
- Interrupts are the basis for asynchronous I/O
  - Device controller performs an operation asynch to CPU
  - Device sends an interrupt signal on bus when done
  - In memory is a *vector table* containing a list of addresses of kernel routines to handle various events
  - CPU switches to address indicated by vector specified by the interrupt signal

I/O Control (continued)

device interrupts

↓

CPU stops current operation, switches to kernel mode, and saves current PC and other state on kernel stack

↓

CPU fetches proper vector from vector table and branches to that address (to routine to handle interrupt)

↓

interrupt routine examines device database and performs action required by interrupt

↓

handler completes operation, restores saved (interrupted state) and returns to user mode (or calls scheduler to switch to another program)

Timer Operation

- How does the OS prevent against runaway user programs (infinite loops)?
- A timer can be set to generate an interrupt in a given time
- Before it transfers to a user program, the OS loads the timer with a time to interrupt
- When the time arrives, the executing program is interrupted and the OS regains control
- This ensures that the OS can get the CPU back even if a user program erroneously or purposely continues to execute past some allotted time.
- The timer is privileged: only the OS can load it

Synchronization

- Interrupts cause potential problems because an interrupt can occur at any time -- causing code to execute that interferes with code that was interrupted
- OS must be able to synchronize concurrent processes.
- This involves guaranteeing that short instruction sequences (read-modify-write) execute atomically.
- One way to guarantee this is to turn off interrupts before the sequence, execute it, and re-enable interrupts; CPU must have a way to disable interrupts.
- Another is to have special instructions that can perform a read/modify/write in a single cycle, or can atomically test and conditionally set a bit, based on its previous value.

Next Time

- We now know what the hardware gives us to use, so
- How do we conceptually organize an OS to put it all together?