Operating Systems
Fall 2014

Operating-System Operations

Myungjin Lee
myungjin.lee@ed.ac.uk
Lower-level architecture affects the OS even more dramatically

• The operating system supports sharing and protection
  – multiple applications can run concurrently, sharing resources
  – a buggy or malicious application can’t nail other applications or the system

• There are many approaches to achieving this

• The architecture determines which approaches are viable (reasonably efficient, or even possible)
  – includes instruction set (synchronization, I/O, …)
  – also hardware components like MMU or DMA controllers
• Architectural support can vastly simplify (or complicate!) OS tasks
  – e.g.: early PC operating systems (DOS, MacOS) lacked support for virtual memory, in part because at that time PCs lacked necessary hardware support
    • Apollo workstation used two CPUs as a bandaid for non-restartable instructions!
  – Until very recently, Intel-based PCs still lacked support for 64-bit addressing (which has been available for a decade on other platforms: MIPS, Alpha, IBM, etc…)
    • Changed driven by AMD’s 64-bit architecture
Architectural features affecting OS’s

- These features were built primarily to support OS’s:
  - timer (clock) operation
  - synchronization instructions (e.g., atomic test-and-set)
  - memory protection
  - I/O control operations
  - interrupts and exceptions
  - protected modes of execution (kernel vs. user)
  - privileged instructions
  - system calls (and software interrupts)
  - virtualization architectures
    - Intel: http://www.intel.com/technology/itj/2006/v10i3/1-hardware/7-architecture-usage.htm
Privileged instructions

• Some instructions are restricted to the OS
  – known as privileged instructions
• e.g., only the OS can:
  – directly access I/O devices (disks, network cards)
    • why?
  – manipulate memory state management
    • page table pointers, TLB loads, etc.
    • why?
  – manipulate special ‘mode bits’
    • interrupt priority level
    • why?
OS protection

• So how does the processor know if a privileged instruction should be executed?
  – the architecture must support at least two modes of operation: kernel mode and user mode
    • VAX, x86 support 4 protection modes
  – mode is set by status bit in a protected processor register
    • user programs execute in user mode
    • OS executes in kernel (privileged) mode (OS == kernel)

• Privileged instructions can only be executed in kernel (privileged) mode
  – what happens if code running in user mode attempts to execute a privileged instruction?
Crossing protection boundaries

• So how do user programs do something privileged?
  – e.g., how can you write to a disk if you can’t execute an I/O instructions?

• User programs must call an OS procedure – that is, get the OS to do it for them
  – OS defines a set of system calls
  – User-mode program executes system call instruction

• Syscall instruction
  – Like a protected procedure call
• The syscall instruction atomically:
  – Saves the current PC
  – Sets the execution mode to privileged
  – Sets the PC to a handler address

• With that, it’s a lot like a local procedure call
  – Caller puts arguments in a place callee expects (registers or stack)
    • One of the args is a syscall number, indicating which OS function to invoke
  – Callee (OS) saves caller’s state (registers, other control state) so it can use the CPU
  – OS function code runs
    • OS must verify caller’s arguments (e.g., pointers)
  – OS returns using a special instruction
    • Automatically sets PC to return address and sets execution mode to user
A kernel crossing illustrated

Firefox: `read(int fileDescriptor, void *buffer, int numBytes)`

- User mode
  - Save user PC
  - PC = trap handler address
  - Enter kernel mode

- Kernel mode
  - Trap handler
    - Save app state
    - Verify syscall number
    - Find `sys_read()` handler in vector table

- `sys_read()` kernel routine
  - Verify args
  - Initiate read
  - Choose next process to run
  - Setup return values
  - Restore app state

- ERET instruction
  - `PC = saved PC`
  - Enter user mode

System call issues

• What would be wrong if a syscall worked like a regular subroutine call, with the caller specifying the next PC?
• What would happen if kernel didn’t save state?
• Why must the kernel verify arguments?
• How can you reference kernel objects as arguments to or results from system calls?
Exception Handling and Protection

• *All* entries to the OS occur via the mechanism just shown
  – Acquiring privileged mode and branching to the trap handler are inseparable

• Terminology:
  – **Interrupt**: asynchronous; caused by an external device
  – **Exception**: synchronous; unexpected problem with instruction
  – **Trap**: synchronous; intended transition to OS due to an instruction

• Privileged instructions and resources are the basis for most everything: memory protection, protected I/O, limiting user resource consumption, …
Memory protection

- OS must protect user programs from each other
  - maliciousness, ineptitude
- OS must also protect itself from user programs
  - integrity and security
  - what about protecting user programs from OS?
- Simplest scheme: base and limit registers
  - are these protected?

<table>
<thead>
<tr>
<th>Prog A</th>
<th>base reg</th>
<th>base and limit registers are loaded by OS before starting program</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prog B</td>
<td>limit reg</td>
<td></td>
</tr>
<tr>
<td>Prog C</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
More sophisticated memory protection

- coming later in the course
- paging, segmentation, virtual memory
  - page tables, page table pointers
  - translation lookaside buffers (TLBs)
  - page fault handling
I/O control

• Issues:
  – how does the OS start an I/O?
    • special I/O instructions
    • memory-mapped I/O
  – how does the OS notice an I/O has finished?
    • polling
    • Interrupts
  – how does the OS exchange data with an I/O device?
    • Programmed I/O (PIO)
    • Direct Memory Access (DMA)
Asynchronous I/O

• Interrupts are the basis for asynchronous I/O
  – device performs an operation asynchronously to CPU
  – device sends an interrupt signal on bus when done
  – in memory, a vector table contains list of addresses of kernel routines to handle various interrupt types
    • who populates the vector table, and when?
  – CPU switches to address indicated by vector index specified by interrupt signal

• What’s the advantage of asynchronous I/O?
• How can the OS prevent runaway user programs from hogging the CPU (infinite loops?)
  – use a hardware timer that generates a periodic interrupt
  – before it transfers to a user program, the OS loads the timer with a time to interrupt
    • “quantum” – how big should it be set?
  – when timer fires, an interrupt transfers control back to OS
    • at which point OS must decide which program to schedule next
    • very interesting policy question: we’ll dedicate a class to it
• Should access to the timer be privileged?
  – for reading or for writing?
Synchronization

- Interrupts cause a wrinkle:
  - may occur any time, causing code to execute that interferes with code that was interrupted
  - OS must be able to synchronize concurrent processes
- Synchronization:
  - guarantee that short instruction sequences (e.g., read-modify-write) execute atomically
  - one method: turn off interrupts before the sequence, execute it, then re-enable interrupts
    - architecture must support disabling interrupts
      - Privileged???
  - another method: have special complex atomic instructions
    - read-modify-write
    - test-and-set
    - load-linked store-conditional
“Concurrent programming”

• Management of concurrency and asynchronous events is biggest difference between “systems programming” and “traditional application programming”
  – modern “event-oriented” application programming is a middle ground
  – And in a multi-core world, more and more apps have internal concurrency

• Arises from the architecture
  – Can be sugar-coated, but cannot be totally abstracted away

• Huge intellectual challenge
  – Unlike vulnerabilities due to buffer overruns, which are just sloppy programming
Some questions

• Why wouldn’t you want a user program to be able to access an I/O device (e.g., the disk) directly?
• OK, so what keeps this from happening? What prevents user programs from directly accessing the disk?
• So, how does a user program cause disk I/O to occur?
• What prevents a user program from scribbling on the memory of another user program?
• What prevents a user program from scribbling on the memory of the operating system?
• What prevents a user program from running away with the CPU?