Operating Systems
ECE344

Lecture 2: Architectural Support for Operating Systems
Ding Yuan

Announcements & reminders

- Lab schedule is out
  - Form your group of 2 by this Friday (18th), 5PM

- Grading policy:
  - Final exam: 50%
  - Midterm exam: 25%
  - Lab assignment: 25%

- Piazza Q/A
  - Please prefix your post with: [Lab0],[Lab1],[Lab2], [Lab3],[Other]
Announcements & reminders

- TA information
  - Inaz Alaei-novin (eyenaz.17@gmail.com)
  - Wei Huang (whuang.nju@gmail.com)
  - Akshay Kumar (jit.akshay@gmail.com)
  - Jun Li (junl.li@mail.utoronto.ca)
  - Ali Shariat (shariat@gmail.com)
  - Xin Tong (xtong@eecg.toronto.edu)

- TAs will be at the lab sessions (3 TAs on Thursday and 3 TAs on Friday)

Content of this lecture

- Review of introduction
- Hardware overview
- A peek at Unix
- Hardware (architecture) support
- Summary
Review

- What are the two main responsibilities of OS?
  - Manage hardware resources
  - Provide a clean set of interface to programs

- Managing resources:
  - Allocation
  - Protection
  - Reclamation
  - Virtualization

- Questions?

Why Start With Hardware?

- Operating system functionality fundamentally depends upon hardware
  - Key goal of an OS is to manage hardware
  - protection and resource sharing
  - If done well, applications can be oblivious to HW details

- Hardware support can greatly simplify – or complicate – OS tasks
  - Early PC operating systems (DOS, MacOS) lacked virtual memory in part because the hardware did not support it
So what is inside a computer

- An abstract overview
  - http://www.youtube.com/watch?v=Q2hmuqS8bwM&feature=related
- An introduction with a real computer
  - http://www.youtube.com/watch?v=VWzX4MEYOBk

A Typical Computer from a Hardware Point of View
Memory-storage Hierarchy

<table>
<thead>
<tr>
<th>Access Time</th>
<th>Typical Capacity</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.3 ns</td>
<td>1 – 16 KB</td>
</tr>
<tr>
<td>0.5 ns</td>
<td>2 – 64 MB</td>
</tr>
<tr>
<td>100 ns</td>
<td>4 – 64 GB</td>
</tr>
<tr>
<td>10,000,000 ns</td>
<td>64 – 4 TB</td>
</tr>
</tbody>
</table>

1 nanosecond = $10^{-9}$ second

A peek into Unix structure

- Application
- Libraries
- Portable OS Layer
- Machine-dependent layer

Written by programmer
Compiled by programmer
Uses library calls (e.g., printf)
A peek into Unix structure

Application

Libraries

Portable OS Layer

Machine-dependent layer

Example: stdio.h
Written by elves
Uses system calls
Defined in headers
Input to linker (compiler)
Invoked like functions
May be “resolved” when program is loaded.

A peek into Unix structure

Application

Libraries

Portable OS Layer

Machine-dependent layer

System calls (read, open..)
All “high-level” code
A peek into Unix structure

Application
Libraries
Portable OS Layer
Machine-dependent layer

Bootstrap
System initialization
Interrupt and exception
I/O device driver
Memory management
Kernel/user mode switching
Processor management

Some systems do not have clear user-kernel boundary
User/kernel mode is supported by hardware

Cannot execute “protected_instruction”, e.g., directly access I/O device

Kernel mode
• Some systems do not have clear user-kernel boundary
• User/kernel mode is supported by hardware
Why hardware has to support User/Kernel mode?

**Imaginary OS code (software-only solution)**

```c
if ([PC] != protected_instruction)
    execute(PC);
else
    switch_to_kernel_mode();
```

Does it work?

Application's code:

```
lw      $t0, 4($gp)
mult    $t0, $t0, $t0
lw      $t1, 4($gp)
or      $t2, $zero, 3
mul     $t1, $t1, $t2
add     $t2, $t0, $t1
sw      $t2, 0($gp)
```

OS: check if next instruction is protected instruction.
Why hardware has to support User/Kernel mode?

Application's code:

```assemble
lw  $t0, 4($gp)
mult $t0, $t0, $t0
lw  $t1, 4($gp)
ori $t2, $zero, 3
mult $t1, $t1, $t2
add $t2, $t0, $t1
sw  $t2, 0($gp)
```

OS: check if next instruction is protected instruction.

- Performance overhead is too big: OS needs to check every instruction of the application!
  - Simulators

Why hardware has to support User/Kernel mode?

Application's code:

```assemble
lw  $t0, 4($gp)
mult $t0, $t0, $t0
lw  $t1, 4($gp)
ori $t2, $zero, 3
mult $t1, $t1, $t2
add $t2, $t0, $t1
sw  $t2, 0($gp)
```

OS: set-up the environment; load the application

- Instead, what we really want is to give the CPU entirely to the application
  - Bare-metal execution

Any problems?

- How can OS check if application executes protected instruction?
  - How can OS know it will ever run again?

Return to OS after termination; OS: schedule next application to execute..
Why hardware has to support User/Kernel mode?

- Give the CPU to the user application
  - Why: Performance and efficiency
  - OS will not be executing

- Without hardware’s help, OS loses control of the machine!
  - Analogy: give the car key to someone, how do you know if he will return the car?

- This is the most fundamental reason why OS will need hardware support --- not only for user/kernel mode

Questions?

Hardware Features for OS

- Features that directly support the OS include
  - Protection (kernel/user mode)
  - Protected instructions
  - Memory protection
  - System calls
  - Interrupts and exceptions
  - Timer (clock)
  - I/O control and operation
  - Synchronization
Types of Hardware Support

- Manipulating privileged machine state
  - Protected instructions
  - Manipulate device registers, TLB entries, etc.

- Generating and handling “events”
  - Interrupts, exceptions, system calls, etc.
  - Respond to external events
  - CPU requires software intervention to handle fault or trap

- Mechanisms to handle concurrency
  - Interrupts, atomic instructions

Protected Instructions

- A subset of instructions of every CPU is restricted to use only by the OS
  - Known as protected (privileged) instructions

- Only the operating system can
  - Directly access I/O devices (disks, printers, etc.)
    - Security, fairness (why?)
  - Manipulate memory management state
    - Page table pointers, page protection, TLB management, etc.
  - Manipulate protected control registers
    - Kernel mode, interrupt level
    - Halt instruction (why?)
OS Protection

- Hardware must support (at least) two modes of operation: kernel mode and user mode
  - Mode is indicated by a status bit in a protected control register
  - User programs execute in user mode
  - OS executes in kernel mode (OS == "kernel")
- Protected instructions only execute in kernel mode
  - CPU checks mode bit when protected instruction executes
  - Setting mode bit must be a protected instruction
  - Attempts to execute in user mode are detected and prevented
    - x86: General Protection Fault

Memory Protection

- OS must be able to protect programs from each other
- OS must protect itself from user programs
- We need hardware support
  - Again: once OS gives the CPU to the user programs, OS loses control
Memory Protection

- Memory management hardware provides memory protection mechanisms
  - Base and limit registers
  - Page table pointers, page protection, TLB
  - Virtual memory
  - Segmentation
- Manipulating memory management hardware uses protected (privileged) operations

Hardware Features for OS

- Features that directly support the OS include
  - Protection (kernel/user mode)
  - Protected instructions
  - Memory protection
  - System calls
  - Interrupts and exceptions
  - Timer (clock)
  - I/O control and operation
  - Synchronization

Questions?
OS Control Flow

- When the processor receives an event of a given type, it
  - transfers control to handler within the OS
  - handler saves program state (PC, registers, etc.)
  - handler functionality is invoked
  - handler restores program state, returns to program

Events

- After the OS has booted, all entry to the kernel happens as the result of an event
  - event immediately stops current execution
  - changes mode to kernel mode, event handler is called
- An event is an “unnatural” change in control flow
  - Events immediately stop current execution
  - Changes mode, context (machine state), or both
- The kernel defines a handler for each event type
  - Event handlers always execute in kernel mode
  - The specific types of events are defined by the machine
- In effect, the operating system is one big event handler
Categorizing Events

- Two kinds of events, interrupts and exceptions
- Exceptions are caused by executing instructions
  - CPU requires software intervention to handle a fault or trap
- Interrupts are caused by an external event
  - Device finishes I/O, timer expires, etc.
- Two reasons for events, unexpected and deliberate
- Unexpected events are, well, unexpected
  - What is an example?
- Deliberate events are scheduled by OS or application
  - Why would this be useful?

This gives us a convenient table:

<table>
<thead>
<tr>
<th></th>
<th>Unexpected</th>
<th>Deliberate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exceptions (sync)</td>
<td>fault</td>
<td>syscall trap</td>
</tr>
<tr>
<td>Interrupts (async)</td>
<td>interrupt</td>
<td>software interrupt</td>
</tr>
</tbody>
</table>

- Terms may be used slightly differently by various OSes, CPU architectures...
- No need to “memorize” all the terms
- Software interrupt – a.k.a. async system trap (AST), async or deferred procedure call (APC or DPC)
- Will cover faults, system calls, and interrupts next
Faults

- Hardware detects and reports “exceptional” conditions
  - Page fault, unaligned access, divide by zero
- Upon exception, hardware “faults” (verb)
  - Must save state (PC, registers, mode, etc.) so that the faulting process can be restarted
- Fault exceptions are a performance optimization
  - Could detect faults by inserting extra instructions into code (at a significant performance penalty)
Handling Faults

- Some faults are handled by “fixing” the exceptional condition and returning to the faulting context
  - Page faults cause the OS to place the missing page into memory
  - Fault handler resets PC of faulting context to re-execute instruction that caused the page fault

- Some faults are handled by notifying the process
  - Fault handler changes the saved context to transfer control to a user-mode handler on return from fault
  - Handler must be registered with OS
  - Unix signals
    - SIGSEGV, SIGALRM, SIGTERM, etc.

Handling Faults

- The kernel may handle unrecoverable faults by killing the user process
  - Program faults with no registered handler
  - Halt process, write process state to file, destroy process
  - In Unix, the default action for many signals (e.g., SIGSEGV)

- What about faults in the kernel?
  - Dereference NULL, divide by zero, undefined instruction
  - These faults considered fatal, operating system crashes
  - Unix panic, Windows “Blue screen of death”
    - Kernel is halted, state dumped to a core file, machine locked up
System Calls

• For a user program to do something “privileged” (e.g., I/O) it must call an OS procedure
  • Known as crossing the protection boundary, or a protected procedure call

• Hardware provides a system call instruction that:
  • Causes an exception, which vectors to a kernel handler
  • Passes a parameter determining the system routine to call
  • Saves caller state (PC, registers, etc.) so it can be restored
  • Returning from system call restores this state

• Requires hardware support to:
  • Restore saved state, reset mode, resume execution

System Call Functions

• Process control
  • Create process, allocate memory

• File management
  • Create, read, delete file

• Device management
  • Open device, read/write device, mount device

• Information maintenance
  • Get time

• Programmers generally do not use system calls directly
  • They use runtime libraries (e.g., stdio.h)
  • Why?
Function call

main () {
    foo (10);
}

Compile

main:   push $10
    call foo
    ...
    foo:   ...
    ret

System call

open (path, flags, mode);

open: ; Linux convention:
    ; parameters via registers.
    mov eax, 5 ; syscall number for open
    mov ebx, path ; ebx: first parameter
    mov ecx, flags ; ecx: 2nd parameter
    mov edx, mode ; edx: 3rd parameter
    int 80h

open: ; FreeBSD convention:
    ; parameters via stacks.
    push dword mode
    push dword flags
    push dword path
    mov eax, 5
    push dword eax ; syscall number
    int 80h
    add esp, byte 16

More information:
http://www.int80h.org
Directly using system call?

- Write assembly code
  - Hard
- Poor portability
  - write different version for different architecture
  - write different version for different OSes
- Application programmers use library
  - Libraries written by elves

**System Call**

Firefox: open()

Trap to kernel mode, save state

Trap handler
  - open read handler in vector table
  - open() kernel routine

Restore state, return to user level, resume execution
Steps in making a syscall

Example:
read (fd, buffer, nbytes)

System Call Issues

- What would happen if the kernel did not save state?
- Why must the kernel verify arguments?
- Why is a table of system calls in the kernel necessary?
Interrupts

- Interrupts signal asynchronous events
  - I/O hardware interrupts
  - Software and hardware timers

- Two flavors of interrupts
  - Precise: CPU transfers control only on instruction boundaries
  - Imprecise: CPU transfers control in the middle of instruction execution
    - What the heck does that mean?
  - OS designers like precise interrupts, CPU designers like imprecise interrupts
    - Why?

Interrupt Illustrated

- Device
  - Raise Interrupt

- User process
  - Suspend user process
  - Execute OS's interrupt handler
  - Save user process's state
  - Execute device driver

- Kernel Mode
  - Mode bit = 0
  - Return state
  - Clear interrupt

- User Mode
  - Mode bit = 1
  - Resume process
How to find interrupt handler?

- Hardware maps interrupt type to interrupt number
- OS sets up Interrupt Descriptor Table (IDT) at boot
  - Also called interrupt vector
  - IDT is in memory
  - Each entry is an interrupt handler
  - OS lets hardware know IDT base
- Hardware finds handler using interrupt number as index into IDT
  - handler = IDT[intr_number]

Timer

- The timer is critical for an operating system
- It is the fallback mechanism by which the OS reclaims control over the machine
  - Timer is set to generate an interrupt after a period of time
    - Setting timer is a privileged instruction
    - When timer expires, generates an interrupt
    - Handled by kernel, which controls resumption context
      - Basis for OS scheduler (more later…)
- Prevents infinite loops
  - OS can always regain control from erroneous or malicious programs that try to hog CPU
- Also used for time-based functions (e.g., sleep())
I/O Control

- I/O issues
  - Initiating an I/O
  - Completing an I/O
- Initiating an I/O
  - Special instructions
  - Memory-mapped I/O
    - Device registers mapped into address space
    - Writing to address sends data to I/O device

I/O Completion

- Interrupts are the basis for asynchronous I/O
  - OS initiates I/O
  - Device operates independently of rest of machine
  - Device sends an interrupt signal to CPU when done
  - OS maintains a vector table containing a list of addresses of kernel routines to handle various events
  - CPU looks up kernel address indexed by interrupt number, context switches to routine
I/O Example

1. Ethernet receives packet, writes packet into memory
2. Ethernet signals an interrupt
3. CPU stops current operation, switches to kernel mode, saves machine state (PC, mode, etc.) on kernel stack
4. CPU reads address from vector table indexed by interrupt number, branches to address (Ethernet device driver)
5. Ethernet device driver processes packet (reads device registers to find packet in memory)
6. Upon completion, restores saved state from stack

Interrupt Questions

• Interrupts halt the execution of a process and transfer control (execution) to the operating system
• Can the OS be interrupted? (Consider why there might be different IRQ levels)
• Interrupts are used by devices to have the OS do stuff
  • What is an alternative approach to using interrupts?
  • What are the drawbacks of that approach?
Alternative approach

• Polling
  ```c
  while (Ethernet_card_queue_is_empty)
  // Ethernet card received packets.
  handle_packets();
  ```
• Problems?
• Analogy:
  • Polling: keeps checking the email every 30 seconds
  • Interrupt: when email arrives, give me a ring

Summary

• Protection
  • User/kernel modes
  • Protected instructions

• System calls
  • Used by user-level processes to access OS functions
  • Access what is “in” the OS

• Exceptions
  • Unexpected event during execution (e.g., divide by zero)

• Interrupts
  • Timer, I/O
Summary (2)

- After the OS has booted, all entry to the kernel happens as the result of an event
  - event immediately stops current execution
  - changes mode to kernel mode, event handler is called

- When the processor receives an event of a given type, it
  - transfers control to handler within the OS
  - handler saves program state (PC, registers, etc.)
  - handler functionality is invoked
  - handler restores program state, returns to program

Architecture Trends Impact
OS Design

- Processor
  - Single core to multi-core
    - OS must better handle concurrency

- Network
  - Isolation to dial-up to LAN to WAN
    - OS must devote more efforts to communications
  -Disconnected to wired to wireless
    - OS must manage connectivity more
  - Isolated to shared to attacked
    - OS must provide more security/protection

- Mobile/battery-operated
  - OS must pay attention to energy consumption
### May you live in Interesting Times

- Multicores
- Smart phones
- Tapes $\rightarrow$ disks $\rightarrow$ flash memory $\rightarrow$ ..
- 3G, 4G..

- Cloud
- Wearable computers
- Virtual reality
- Motion capturing device
- ..