SE350: Operating Systems

Lecture 2: The Kernel Abstraction
Main Points

• Process concept
  • A process is the OS abstraction for executing a program with limited privileges

• Dual-mode operation: user vs. kernel
  • Kernel-mode: execute with complete privileges
  • User-mode: execute with fewer privileges

• Safe control transfer
  • How do we switch from one mode to the other?
Booting

- Physical Memory
  - BIOS
    - Bootloader instructions and data
  - OS kernel
    - Bootloader copies OS kernel
  - Login app
    - OS kernel copies login application

- Disk
  - Bootloader
  - OS kernel
  - Login app

1. BIOS copies bootloader
2. Bootloader copies OS kernel
3. OS kernel copies login application
Challenge: Protection

• 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
Physical Memory

Edits → Source Code → Compiler → Executable Image: Instructions and Data → Operating System Copy

Physical Memory

Process

Operating System Kernel

Machine Instructions
Data
Heap
Stack
Machine Instructions
Data
Heap
Stack
Process Abstraction

• Process: an instance of a program, running with limited rights
  • Analogous to objects and classes in OO programming

• 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
Thought Experiment

• 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 Javascript and other interpreted languages

• How do we go faster?
  • Run the unprivileged code directly on the CPU!
Dual-Mode Operation

• Kernel mode
  • Execution with the full privileges of the hardware
  • Read/write to any memory, access any I/O device, read/write any disk sector, send/read any packet

• User mode
  • Limited privileges
  • Only those granted by the operating system kernel

• On the x86, mode stored in EFLAGS register
• On the MIPS, mode in the status register
A Model of a CPU

Branch Address

Select PC

New PC

Program Counter

CPU Instructions Fetch and Execute

opcode

New PC + 4
A CPU with Dual-Mode Operation

Handler PC → Select PC → New PC → Program Counter → CPU Instructions Fetch and Execute

New Mode → Select Mode → Mode

Branch Address

Opcode
Hardware Support for Dual-Mode Operation

- Privileged instructions
  - Available to kernel and not available to user code

- Limits on memory accesses
  - To prevent user code from overwriting the kernel

- Timer
  - To regain control from a user program in a loop
Privileged Instructions

• Examples?

• What should happen if a user program attempts to execute a privileged instruction?
For a “Hello world” program, the kernel must copy the string from the user program memory into the screen memory. Why not allow the application to write directly to the screen’s buffer memory?

- Hint: buffer overflow!
Simple Memory Protection

Processor

Virtual Address

Base

Bound

Physical Address

Raise Exception

Physical Memory

Base

Base + Bound

Implementation
Towards Virtual Addresses

• Problems with base and bounds?
Virtual Addresses

- Translation done in hardware, using a table
- Table set up by operating system kernel
Virtual Address Example

```c
int staticVar = 0;       // a static variable
int main() {
    staticVar += 1;
    usleep(5000000);       // sleep for 5 seconds
    printf("static address: %x, value: %d\n", &staticVar, staticVar);
}
```

- What happens if we run two instances of this program at the same time?
- What if we took the address of a procedure local variable in two copies of the same program running at the same time?
Virtual Address vs Physical Address

• The same virtual address in two different processes can refer to different physical addresses. Why?
• The same virtual address in two different processes can refer to the same physical address. Why?
• Different virtual addresses can refer to the same physical address. Why?
Question

• Suppose you have a type-safe object-oriented language. If the OS only ran programs written in that language, would it still need hardware memory address protection?
  • Hint: who do you trust?
Hardware Timer

- HW device that periodically interrupts the processor
  - Returns control to the kernel handler
  - Interrupt frequency set by the kernel
    - Not by user code!
  - Interrupts can be temporarily deferred
    - Not by user code!
    - Interrupt deferral crucial for implementing mutual exclusion
User to Kernel Mode Switch (Trap)

- Interrupts
  - Triggered by timer and I/O devices

- Exceptions
  - Triggered by unexpected program behavior
  - Or malicious behavior!

- System calls (aka protected procedure call)
  - Request by program for kernel to do some operation on its behalf
  - Only limited # of very carefully coded entry points
Question

- Examples of exceptions
- Examples of system calls
Kernel to User Mode Switch

- New process/new thread start
  - Jump to first instruction in program/thread
- Return from interrupt, exception, system call
  - Resume suspended execution
- Process/thread context switch
  - Resume some other process
- User-level upcall (UNIX signal)
  - Asynchronous notification to user program
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
Interrupt Vector

- Table set up by OS kernel; pointers to code to run on different events

```c
// Function to handle timer interrupts
handleTimerInterrupt() {
    ...
}

// Function to handle divide-by-zero error
handleDivideByZero() {
    ...
}

// Function to handle system call
handleSystemCall() {
    ...
}
```
Interrupt Stack

• Per-processor, located in kernel (not user) memory
  • Usually a process/thread has both: kernel and user stack
  • Hardware and interrupt handler together push interrupted process’s registers onto the interrupt stack

• Why can’t the interrupt handler run on the stack of the interrupted user process?
  • Hint: reliability and security!
Interrupt Stack (cont.)

User Stack

- Running: 
  - ...
  - Proc2
  - Proc1
  - Main

- Ready to Run: 
  - ...
  - Proc2
  - Proc1
  - Main

- Waiting for I/O: 
  - Syscall
  - Proc2
  - Proc1
  - Main

Kernel Stack

- User CPU State

- I/O Driver
- Top Half
- Syscall Handler
- User CPU State
Interrupt Masking

- Interrupt handler runs with interrupts off
  - Re-enabled when interrupt completes
- OS kernel can also turn interrupts off
  - E.g., when determining the next process/thread to run
  - On x86
    - CLI: disable interrupts
    - STI: enable interrupts
    - Only applies to the current CPU (on a multicore)
- We will need this to implement synchronization
Interrupt Handlers: Top and Bottom Halves

- Hardware I/O devices have limited buffers
  - E.g., keyboard drops keystrokes if its buffer is full
  - Masking interrupts could cause dropped events
- Interrupt handlers have two halves
  - Bottom half is non-blocking, runs to completion
    - Waits only for limited duration
    - Once done, re-enables interrupt
  - Top half runs as a kernel thread
Case Study: x86 Interrupt (Hardware Support)

- Hardware saves current stack pointer (SP)
- Saves current program counter (PC)
- Saves current processor status word (PSW)
- Switches to kernel stack
- Puts SP, PC, PSW on stack
- Switches to kernel mode
- Vectors through interrupt table
- Interrupt handler saves registers it might clobber
Before Interrupt

User-level Process

foo () {
    while(...) {
        x = x+1;
        y = y-2;
    }
}

User Stack

Registers

SS: ESP
CS: EIP
EFLAGS
Other Registers: EAX, EBX, ...

Kernel

handler() {
    pushad
    ...
}

Interrupt Stack
During Interrupt

User-level Process

```plaintext
foo () {
    while(...) {
        x = x+1;
        y = y-2;
    }
}
```

User Stack

Registers

- SS: ESP
- CS: EIP
- EFLAGS
- other registers: EAX, EBX, ...

Kernel

```plaintext
handler() {
pushad...
}
```

Interrupt Stack

- Error
- EIP
- CS
- EFLAGS
- ESP
- SS
After Interrupt

User-level Process

foo () {  
while(...) {  
x = x+1;  
y = y-2;  
}  
}

Registers

Kernel

Stack

Interrupt Stack

EFLAGS
other registers: EAX, EBX, ...

SS: ESP
CS: EIP
EBX
EAX
ESP
SS
Error
EIP
CS
EFLAGS
ESP
SS
Question

• Why is the stack pointer saved twice on the interrupt stack?
  • Hint: is it the same stack pointer?
At the End of Handler

• Handler restores saved registers
• **Atomically** return to interrupted process/thread
  • Restore program counter
  • Restore program stack
  • Restore processor status word
  • Switch to user mode
Question

• Suppose the OS over-writes a value in the trap frame. What happens when the handler returns?
  • Hint: trap frame saves state of interrupted process

• Why might the OS want to do this?
  • Hint: return address and return value
Question

• The trap frame is stored on the interrupt stack; where is it stored after a context switch to a different process?
Secure System Calls

User Program

```c
main () {
    file_open(arg1, arg2);
}
```

(1) \arrow{up}

(6)

User Stub

```c
file_open(arg1, arg2) {
    push SYSCALL_OPEN
    trap
    return
}
```

(2) \arrow{right}

(3) \arrow{down}

(4)

Kernel

```c
file_open(arg1, arg2) {
    // do operation
}
```

(5)

Kernel Stub

```c
file_open_handler() {
    // copy arguments
    // from user memory
    // check arguments
    file_open(arg1, arg2);
    // copy return value
    // into user memory
    return;
}
```
Kernel System Call Handler

• Locate arguments
  • In registers or on user stack
  • Translate user addresses into kernel addresses

• Copy arguments
  • From user memory into kernel memory
  • Protect kernel from malicious code evading checks

• Validate arguments
  • Protect kernel from errors in user code

• Copy results back into user memory
  • Translate kernel addresses into user addresses
Upcall: User-level Event Delivery

- Notify user process of some event that needs to be handled right away
  - Preemptive user-level threads
  - Asynchronous I/O notification
  - Interprocess communication
  - User-level excepting handling
  - User-level resource allocation
Upcalls vs Interrupts

- Signal handlers = interrupt vector
- Signal stack = interrupt stack
- Automatic save/restore regs = transparent resume
- Signal masking: signals disabled while in signal handler
Upcall: Before

```c
x = y + z;
```

```
signal_handler() {
    ...
}
```
Upcall: During

... 

x = y + z; ←
...

signal_handler() {
...
}

... 

Program Counter

Stack

Stack Pointer

Signal Stack

Saved Registers

SP

PC
Question

• How many user-kernel transitions are needed for a static web server to read an incoming HTTP request and reply with the file data?
User-Kernel Transitions Example

1. Network Socket Read
2. Copy Arriving Packet (DMA)
3. Kernel Copy
4. Parse Request
5. File Read
6. Disk Request
7. Disk Data (DMA)
8. Kernel Copy
9. Format Reply
10. Write and Copy to Kernel Buffer
11. Format Outgoing Packet and DMA
12. Format Reply

Server

Kernel

Hardware

Network Interface

Disk Interface
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
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

• Use best practices
  • Always walk through your code line by line
  • Module tests – narrow scope of where problem is
  • Develop code in stages, with dummy replacements for later functionality
Virtual Machines

• How do we debug an operating system kernel?
  • Is the debugger an application? Part of the kernel?

• Solution: virtual machine
  • Run a “guest” operating system as a process
  • Run “guest” applications on the guest OS kernel
  • Examples: KVM, VMware, Xen, Citrix, QEMU, etc.
Questions

• Can we run a guest operating system directly on the CPU in user mode?
  • Emulate the behavior of privileged instructions executed by guest OS, as if executed on the hardware

• If hardware is virtualizable
  • Privileged instructions must cause trap when at user level, rather than fail silently
  • Or kernel must somehow re-write those instructions to cause a trap (VMware)

• Underlying kernel called the virtual machine monitor or host kernel
Mode Transfer inside Virtual Machine

- **Guest User Mode**
  - Guest Process
  - Guest Interrupt Stack
  - Guest Flags

- **Host User Mode**
  - Guest Process...trap...

- **Guest Kernel Mode**
  - Guest Kernel
    - Guest file system and other kernel services
  - Guest Interrupt Table
  - Timer Handler
  - Syscall Handler

- **Host Kernel Mode**
  - Host Process
  - Host Interrupt Stack
  - Host Flags
  - Host Kernel
    - Virtual Disk
    - Host Interrupt Table
    - Timer Handler
    - Syscall Handler

- **Hardware**
  - Physical Disk
User-Level Virtual Machine

• How does VM Player work?
  • Runs as a user-level application
  • How does it catch privileged instructions, interrupts, device I/O?

• Installs kernel driver, transparent to host kernel
  • Requires administrator privileges!
  • Modifies interrupt table to redirect to kernel VM code
  • If interrupt is for VM, upcall
  • If interrupt is for another process, reinstalls interrupt table and resumes kernel
Hardware Support for OS (Summary)

- Privilege levels
- Privileged instructions
- Memory translation
- Processor exceptions
- Timer interrupts
- Device interrupts
- Interprocessor interrupts
- Interrupt masking
- System calls
- Return from interrupt
- Boot ROM
Acknowledgment

- This lecture is a slightly modified version of the one prepared by Tom Anderson