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

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

Physical Memory

<table>
<thead>
<tr>
<th>BIOS</th>
<th>Bootloader instructions and data</th>
</tr>
</thead>
<tbody>
<tr>
<td>OS kernel</td>
<td>OS kernel instructions and data</td>
</tr>
<tr>
<td>Login app</td>
<td>Login app instructions and data</td>
</tr>
</tbody>
</table>
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
  - Machine Instructions
  - Data
  - Heap
  - Stack

- Process
  - Machine Instructions
  - Data
  - Heap
  - Stack

- Operating System Kernel
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

+4

opcode
A CPU with Dual-Mode Operation

Handler PC

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

Select Mode → New 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?
Question

• 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?
Simple Memory Protection

Implementation

Processor

Virtual Address

Base

Bound

Raise Exception

Physical Address

Physical Memory

Base

Base + Bound
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
main() {
    staticVar += 1;
    sleep(10);          // sleep for x 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

- Hardware 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

- From user mode to kernel mode
  - 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
  - Trap: user to kernel mode switch
• Examples of exceptions

• Examples of system calls
Kernel to User Mode Switch

- From kernel mode to user mode
  - 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
Restoring User State

• We need to be able to interrupt and transparently resume the execution of a user program for several reasons:
  • I/O device signals I/O completion
  • Periodic hardware timer to check if app is hung
  • Multiplexing multiple apps on a single CPU
  • App unaware it has been interrupted!
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
Interrupt Stack

• Per-processor, located in kernel (not user) memory
  • Usually a process/thread has both: kernel and user stack
• Why can’t the interrupt handler run on the stack of the interrupted user process?
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  
- 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

• Non-blocking, run to completion
  • Minimum necessary to allow device to take next interrupt
  • Any waiting must be limited duration
  • Wake up other threads to do any real work
    • Linux: semaphore

• Rest of device driver runs as a kernel thread
Case Study: x86 Interrupt (Hardware Support)

- Hardware saves current stack pointer
- Saves current program counter
- Saves current CPU status word (condition codes)
- 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

```c
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

```
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

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

User-level Process

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

Stack

Interrupt Stack

Kernel

Registers

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

All Registers

- EIP
- CS
- EFLAGS
- ESP
- SS
- Error
- EBX
- EAX
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/condition codes
  • Switch to user mode
Question

• Suppose the OS overwrites a value in the trapframe. What happens when the handler returns?

• Why might the OS want to do this?
The trapframe is stored on the interrupt stack; where is it stored after a context switch to a different process?
Upcall: User-level Event Delivery

- Notify user process of some event that needs to be handled right away
  - Time expiration
    - Real-time user interface
    - Time-slice for user-level thread manager
  - Interrupt delivery for VM player
  - Asynchronous I/O completion (async/await)
- AKA UNIX signal
Upcalls vs Interrupts

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

```c
x = y + z;

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

... x = y + z; ...

Program Counter

Stack

Stack Pointer

Signal Stack

Saved Registers

SP

PC

signal_handler() {
...
}

Secure System Calls

User Program

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

Kernel

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

User Stub

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

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
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
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?

• Can we run legacy applications on a new operating system 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
Host User Mode

Host User Mode
Guest Kernel Mode

Host Kernel Mode

Guest Process

Guest PC
Guest SP
Guest Flags

Guest Kernel

Guest file system
and other kernel services

Guest Process
... trap ...

Guest Interrupt Table

Syscall Handler

Host User Mode

Guest User Mode

Guest Kernel Mode

Guest Kernel

Timer Handler

Guest Flags

Guest Exception Stack

Interrupt Table

Syscall Handler

Host Kernel

Host Process

Host PC
Host SP
Host Flags

Host Kernel

Virtual Disk

Timer Handler

Host Interrupt Table

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
Acknowledgment

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