CS310 Operating Systems

Lecture 10: User Mode ⟺ Kernel Mode Transfers

Ravi Mittal

IIT Goa

Acknowledgements !

- Contents of this class presentation has been taken from various sources. Thanks are due to the original content creators:
  - Class presentation: University of California, Berkeley: David Culler, Anthony D. Joseph, John Kubiatowicz, AJ Shankar, George Necula, Alex Aiken, Eric Brewer, Ras Bodik, Ion Stoica, Doug Tygar, and David Wagner
  - Book: The Operating System Concepts, third edition: Silberschatz, Peter Galvin, Greg Gagne,

Read the following:

- Book: Operating Systems: Principles and Practice (2nd Edition) Anderson and Dahlin
  - Volume 1, Kernel and Processes
    - Chapter 2.2: Dual Mode of Operation

We will study..

- Revision of the last class
- Safe Mode Transfer

Previous Classes..

Four Fundamental OS Concepts

- **Thread: Execution Context**
  - Fully describes program state

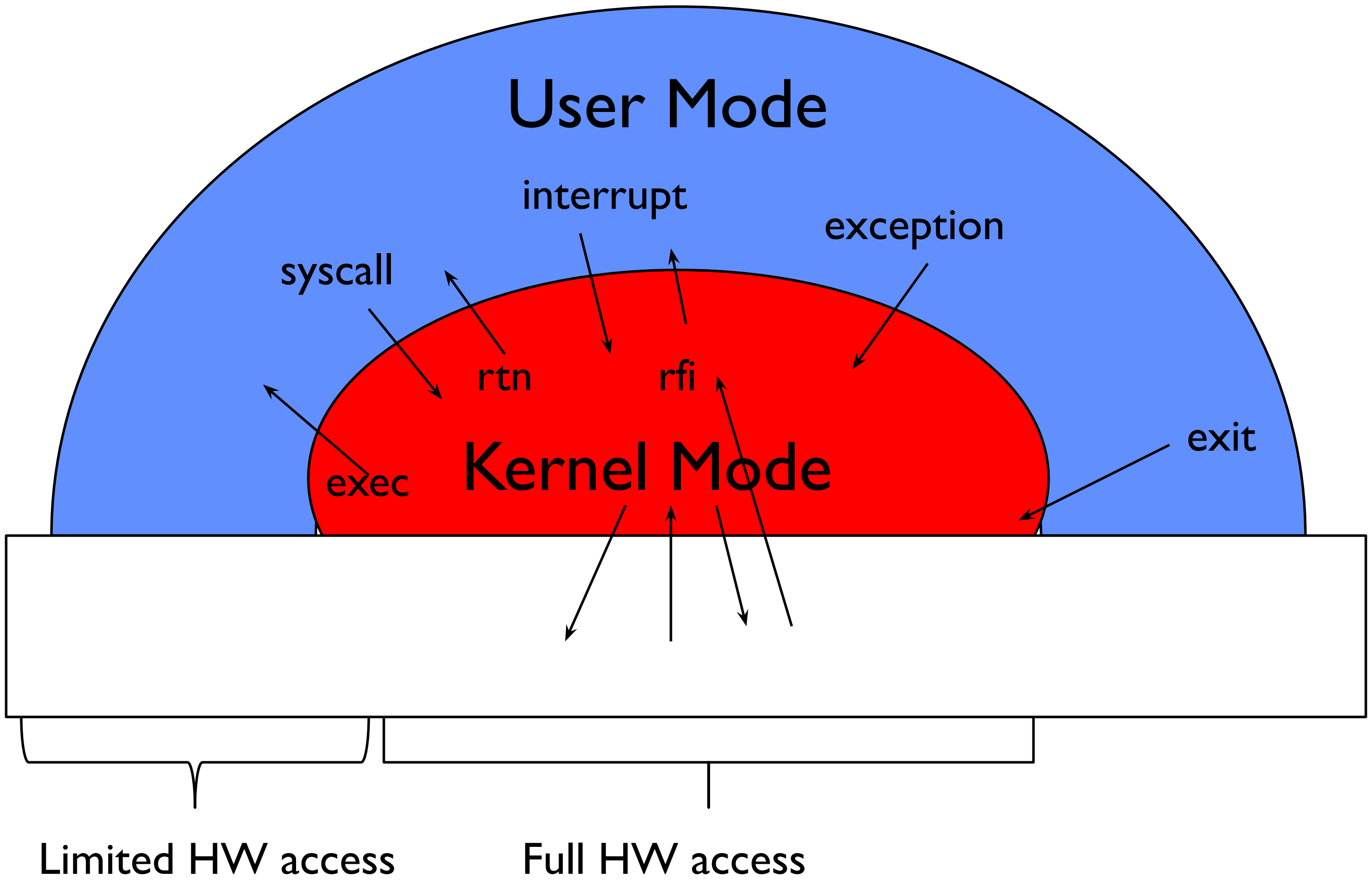
Done

- **Address space**
  - Set of memory addresses accessible to program (for read or write)
- **Process: an instance of a running program**
  - Protected Address Space + One or more Threads
- **Dual mode operation / Protection**
  - Only the "system" has the ability to access certain resources

Fourth OS Concept: Dual Mode Operation

- Hardware provides at least two modes:
  - Kernel mode (or “supervisor” or “protected”)
  - User mode: Normal programs executed
- 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

User/Kernel (Privileged) Mode



rfi : return from Interrupt

Hardware Support: Dual-Mode Operation

- Privileged instructions
  - Available to kernel
  - Not available to user code
- Limits on memory accesses
  - In user mode, all memory accesses outside of a process's valid memory region must be prohibited
  - Prevent user code from overwriting the kernel
- Timer Interrupts
  - Processor must have a way to regain control from a user program in a loop

User Kernel Mode Transfer

- System Call (syscalls)
  - User Process requests a system service
    - Open or delete a file, read/write data into files, create a new user process, establish a connection to web server etc
- Interrupt
  - External asynchronous event, independent of the process
  - e.g., Timer, I/O device
- Processor Exception (trap)
  - Hardware event caused by user program behavior that causes context switch
  - E.g., Divide by zero, bad memory access (segmentation fault)

Implementation of Safe Mode Transfer

Implementing Safe Kernel Mode Transfers (1)

- Mode transfer must be carefully done
  - Buggy or malicious user program should not corrupt the kernel
  - It is done at the runtime
- Context switch must be carefully crafted
- All Operating Systems have a common sequence of instructions for mode transfer
  - Limited Entry into the Kernel
  - Atomic changes to processor state
  - Transparent, re-startable execution
- Hardware support: Interrupt Vector Table and Interrupt Stack

Implementing Safe Kernel Mode Transfer (2)

- Limited Entry to the Kernel
  - Only limited places in OS are entry points
- Atomic Changes to processor State
  - Atomic (at the same time) changes to
    - Program counter
    - Stack pointer
    - Memory protection
    - Kernel/user mode bit

Implementing Safe Mode Transfer (4)

- Transparent, Restartable Execution
  - System must be able to restore the state of the program before the interrupt occurred
  - To a user process an interrupt is invisible
    - Except that the running program temporarily slows down

- On an interrupt
  - Processor saves it’s current state to memory
  - Temporarily defers further evens (eg another interrupt)
  - Changes to Kernel Mode
  - Jumps to Interrupt or exception handler
  - When handler finishes, the processor state is restored from its saved location, and restarts execution from where it was interrupted

Structure for Mode Transfer

Interrupt Vector

Interrupt Vector Table

- Interrupt Vector Table is required to handle
  - Interrupts
  - Processor exception
  - System Call
- Interrupt Vector Table (also called Exception Table)
  - Lists Kernel routines to handle various interrupts, exceptions, and system calls
  - An array of pointers
    - Each pointer points to the first instruction of a different handler procedure in the kernel
- Interrupt Handler
  - The procedure called by the kernel on an interrupt

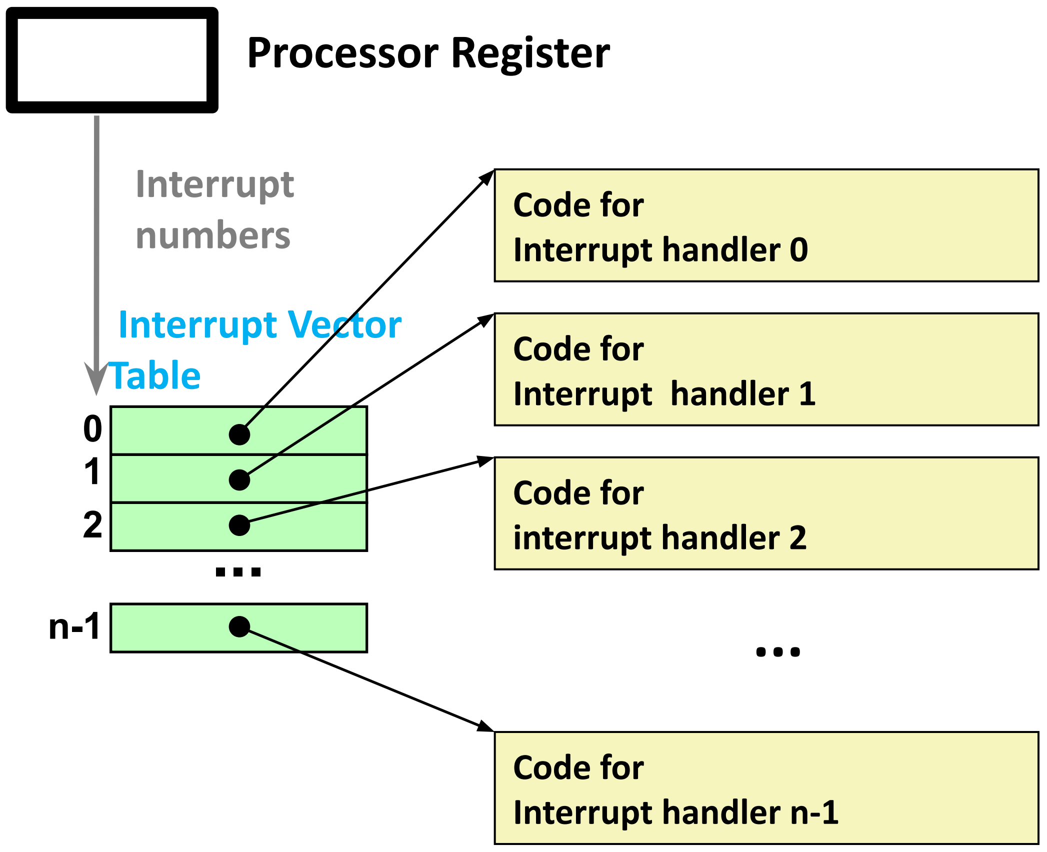
Interrupt (or exception) Vector

- Interrupt occurred
  - Question is what to execute next ?
- There are many types of exceptions/traps – each need to be handled in different way
- Each interrupt/exception provides a number
- Number used to index into
  - Interrupt Vector Table (also called Exception Table)
- Each Interrupt Vector Table entry points to specific interrupt handler
- Kernel sets up Interrupt vector table at the boot time

  - Why is Interrupt vector stored in Kernel instead of user memory ?

Interrupt Vector Table

- Table set up by OS kernel; pointers to code to run on different events; Each interrupt/exception has a specific number


Processor Register
Interrupt numbers
Interrupt Vector Table
0
1
2
...
n-1
Code for Interrupt handler 0
Code for Interrupt handler 1
Code for interrupt handler 2
...
Code for Interrupt handler n-1

- Each type of event has a unique Interrupt/ exception number k
- k = index into Interrupt Vector Table
- Handler k is called each time Interrupt/exception k occurs

Handles – Interrupts, Exceptions, Syscalls

Interrupt Vector Table

- Processor has a special register that points Interrupt Vector Table
  - Stored in an area of kernel memory
  - Format of Interrupt Vector Table is processor-specific
- On x86, Interrupt Vector table- entries:
  - 0 – 31: Different types of processor exceptions – eg divide-by zero
  - 32 – 255: Diff types of interrupts – timer, keyboard, etc
  - 64 : System Call trap handler
- Hardware determines what has caused interrupt
  - Which Device, or trap/exception, or sys call
- Based on the above, the hardware selects the right entry from the Interrupt Vector Table and invoke the appropriate handler

Structure for Mode Transfer

Syscall Handling

Kernel System Call Handler Functions

- Vector through well-defined syscall entry points!
  - Table maps system a call number to respective handler
- Locate arguments
  - In registers or on user stack
- 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

Mode Transfer: Kernel to User

Kernel to User Mode transition

- To start a new process
  - Kernel copies the program to memory
  - Sets PC to the first instruction
  - Sets the stack pointer to the base of the user stack
  - Switches to user Mode
- Resume after an interrupt, processor exception or system call
- Switch to a different process

Return from Syscall

- When OS is done handling syscall or interrupt, it calls a special instruction `return-from-trap`
  - Restore context of CPU registers from kernel stack/PCB
  - Change CPU privilege from kernel mode to user mode
  - Restore PC and jump to user code after trap
- User process unaware that it was suspended, resumes execution as always
- Must always return to the same user process from kernel mode? No
- Before returning to user mode, OS checks if it must switch to another process

Switching between Processes

Context Switching .. By OS ?

- OS decides to stop one process and start another
- If a process is running on the system then the OS is not running
  - Then how does the OS comes into the picture?
- How can the operating system regain control of the CPU so that it can switch between processes?
- Two approaches
  - Co-operative Approach
  - Non-cooperative Approach

Co-operative Approach

- Old Mac system (M11) and Xerox Alto system used this approach
- the OS *trusts* the processes of the system to behave reasonably
- Processes often do system calls (eg to read a file or to send a message or to create a new process etc)
- Processes also use `Yield` system call to give control to OS

- How can the OS gain control of the CPU even if processes are not being cooperative? What can the OS do to ensure a rogue process does not take over the machine?

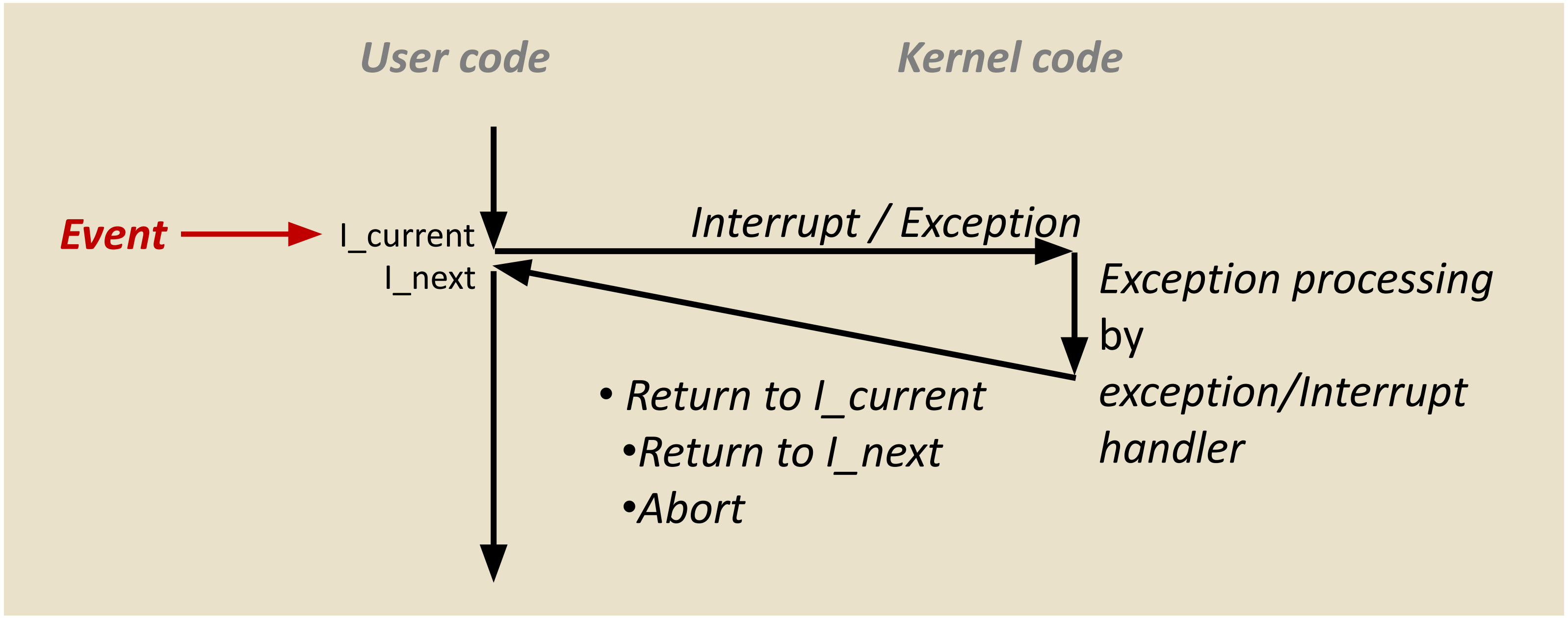
Non Co-operative Approach

- Use a timer interrupt
- A timer device can be programmed to raise an interrupt every so many milliseconds
- The OS has regained control of the CPU, and thus can do what it pleases
  - the OS must inform the hardware of which code to run when the timer interrupt occurs
  - During the boot sequence, the OS must start the timer
- After gaining control OS must make a decision: whether to continue running the currently-running process, or switch to a different one
  - Scheduler is used

Interrupt during Interrupt Processing?

- During interrupt or trap handling, another interrupt may occur
- Many approaches to handle such events
  - Simple Approach: Disable Interrupts during interrupt processing

Interrupt /Exceptions/Syscalls – Way of Mode transfer (Recall)



Lecture Summary

- Hardware provides two modes of execution
  - User Mode
  - Kernel Mode
- User processes run in user mode – with limited privileges
  - It can’t access Kernel memory
  - It can use Syscalls for I/O, File manipulation etc
- Kernel Mode processes have full privileges of hardware
- Hardware Support for Dual Mode
  - Privilege instructions (available to Kernel)
  - Limits on Memory Access
  - Timer Interrups

Backup Slides

Mode Transfer: User to Kernel -

Synchronous Exception

Synchronous Exceptions

- Caused by events that occur as a result of executing an instruction:
  - ***Traps***
    - Intentional
    - Examples: ***system calls***, breakpoint traps, special instructions
    - Returns control to "next" instruction
  - ***Faults***
    - Unintentional but possibly recoverable
    - Examples: page faults (recoverable), protection faults (unrecoverable), floating point exceptions
    - Either re-executes faulting ("current") instruction or aborts
  - ***Aborts***
    - Unintentional and unrecoverable
    - Examples: illegal instruction, parity error, machine check
    - Aborts current program

System Calls in x86-64

- Each x86-64 system call has a unique ID number
- Examples:

Number	*Name*	*Description*
0	`read`	Read file
1	`write`	Write file
2	`open`	Open file
3	`close`	Close file
4	`stat`	Get info about file
57	`fork`	Create process
59	`execve`	Execute a program
60	`_exit`	Terminate process
62	`kill`	Send signal to process

Mode Transfer: User to Kernel -

Interrupt (asynchronous Exception)

Interrupts (Asynchronous Exception)

- An asynchronous signal to the processor
  - Some external event requires processor's attention
  - Indicated by setting the processor's *interrupt pin*
  - Processor stalls or completes existing instruction that is in progress; Saves current execution state; Starts execution of specially designated interrupt handler in the kernel
- Each different type of interrupt requires its own handler
- Example:
  - Timer Interrupt: Every few ms; an external timer chip triggers an interrupt ; Timer handler can switch execution to different process
  - I/O interrupt eg from mouse, keyboard, disk, Ethernet, WiFi, Flash drive, Inter-processor interrupts, DMA, arrival of a packet from a disk
  - Inter-processor Interrupt: For inter processor communication

Interrupt Masking

- Interrupt handler runs with interrupts off
    - Re-enabled when interrupt completes
- OS kernel can also turn interrupts off
    - Eg., when determining the next process/thread to run
    - On x86, privileged instructions: CLI, STI
        - CLI: disable interrupts
        - STI: enable interrupts
        - Only applies to the current CPU (on a multicore)

Processor Exceptions

- A hardware event caused by user program behavior
- Causes transfer of control to the Kernel
- Processor saves the current execution state and runs specially designated exception handler in the kernel
- Example:
  - User process attempts to execute a privileged instruction
  - User process tries to access memory out of it’s own memory region
  - Process divides an integer by zero
  - Process attempts to write to read-only memory
  - A benign event: setting up a breakpoint
- Processor exceptions are used effectively to emulate VMs

Exceptions cause Mode Transfer

- Syscall
  - Process requests a system service, e.g., exit
  - Like a function call, but “outside” the process
  - Does not have the address of the system function to call
  - Like a Remote Procedure Call (RPC) – for later
  - Marshall the syscall id and args in registers and exec syscall
- Trap or Exception
  - Internal synchronous event in process triggers context switch
  - e.g., Protection violation (segmentation fault), Divide by zero,
- Interrupt
  - External asynchronous event triggers context switch
  - e. g., Timer, I/O device
  - Independent of user process

- How does the OS kernel prevent a process from harming another process ?
- When there are multiple programs in Main Memory
  - What prevents a process from overwriting another process's data structures, or
  - Overwriting the OS image stored on disk?
- Recall RISC-V instructions
  - Most instructions such add, sub etc are perfectly safe
  - How can we allow them to execute directly on hardware?
- We implement as simple check in hardware called dual-mode operation
  - Represented by a single bit in the **processor status register** that signifies the current mode of the processor