KERNEL THREAD IMPLEMENTATION DETAILS

CS124 – Operating Systems

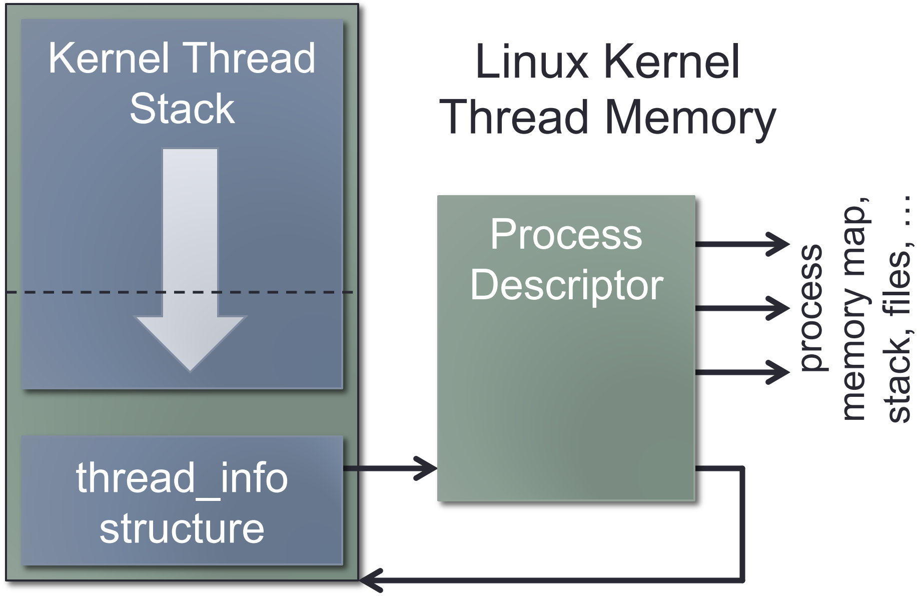
Spring 2024, Lecture 8

Last Time: Kernel Threads

- OS kernel must provide a multitasking implementation
- **Kernel threads** are the minimal form of schedulable task inside the kernel
- Tend to be very lightweight in nature, with few resources
  - CPU state: registers, flags, stack pointer (and that's it)
- May *correspond to* a user-mode process or thread, but it doesn't actually hold user-space resources
  - These are in the Process Control Block (PCB) for the process
  - Specifically, the user process has its own area for data and stack
- Want to remain in kernel for as short a time as possible…
- Kernel threads tend to have very limited stacks to use

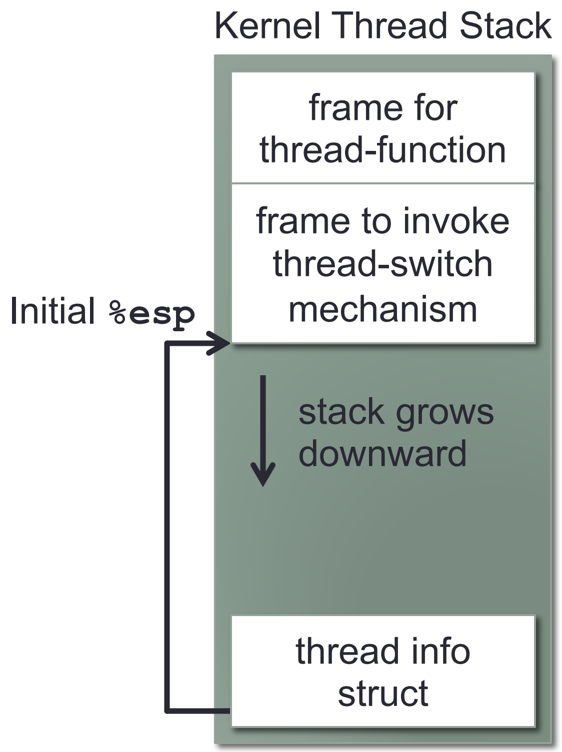
Kernel-Thread Stacks: Examples

- Linux has only 8KB for kernel-thread stacks
  - (on IA32, kernel-thread stacks are usually multiples of 4KB, due to virtual memory paging)
  - Kernel thread info and stack are stored in same 8KB area (~8140 bytes for stack)
- Windows has varying sizes for kernel-thread stacks
  - On IA32, 12KB kernel stacks
  - On x86-64, 24KB kernel stacks
- Pintos has 4KB kernel-thread stacks



Kernel-Thread Mechanics

- When a kernel thread is created, its stack is initialized to work with the kernel's thread-switch machinery
- Initial kernel-thread state:
  - A stack-frame for the kernel-thread's initial function, plus any function argument
  - A stack-frame for the kernel's thread-switch operation, containing the thread's initial machine context
  - Other details, including where the "top of stack" is
- Mechanism is obviously hardware-specific
- Pintos:
  - See thread_create() in thread.c for how this is set up
  - Types for initial frames are defined in switch.h
  - Thread-switch mechanism must be IA32 assembly; see switch_threads() function in switch.S

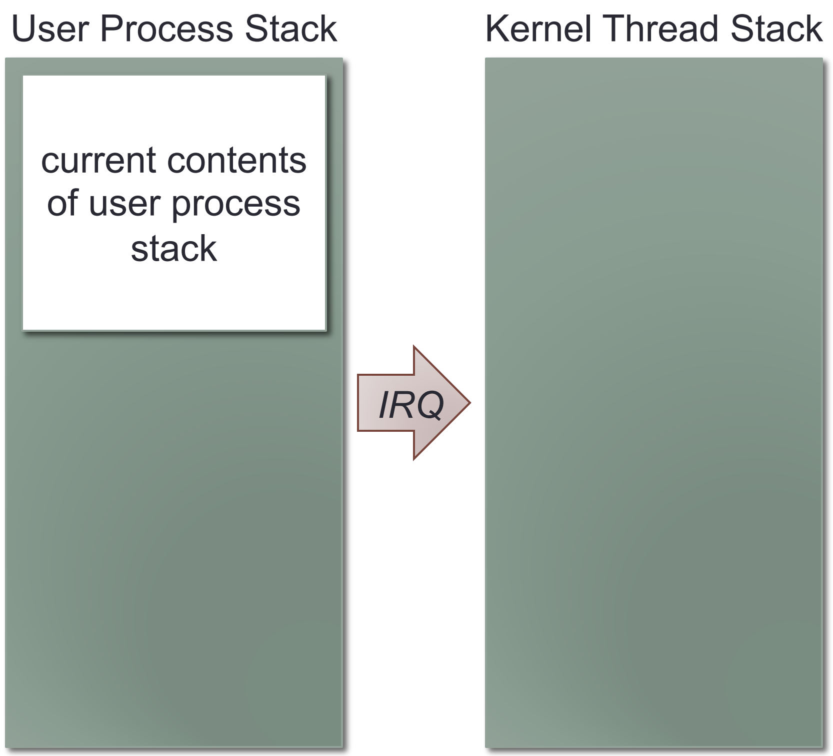


Kernel Control Paths

- A **kernel control path** is the series of instructions executed in the kernel to handle a trap, fault, interrupt, …
- A kernel is **reentrant** if kernel control paths are allowed to execute concurrently (i.e. overlap in time)
  - Non-reentrant kernels cannot have overlapping kernel control paths
  - All modern operating systems have reentrant kernels
- An extremely common occurrence:
  - Program makes a system call, passing pointers to memory areas
  - Inside the kernel trap handler, a page fault occurs
  - In this case, this kernel control path suspends while disk controller performs disk accesses to resolve the page fault, allowing another suspended kernel control path to resume
- Also extremely common to have other hardware interrupts fire during a software trap handler

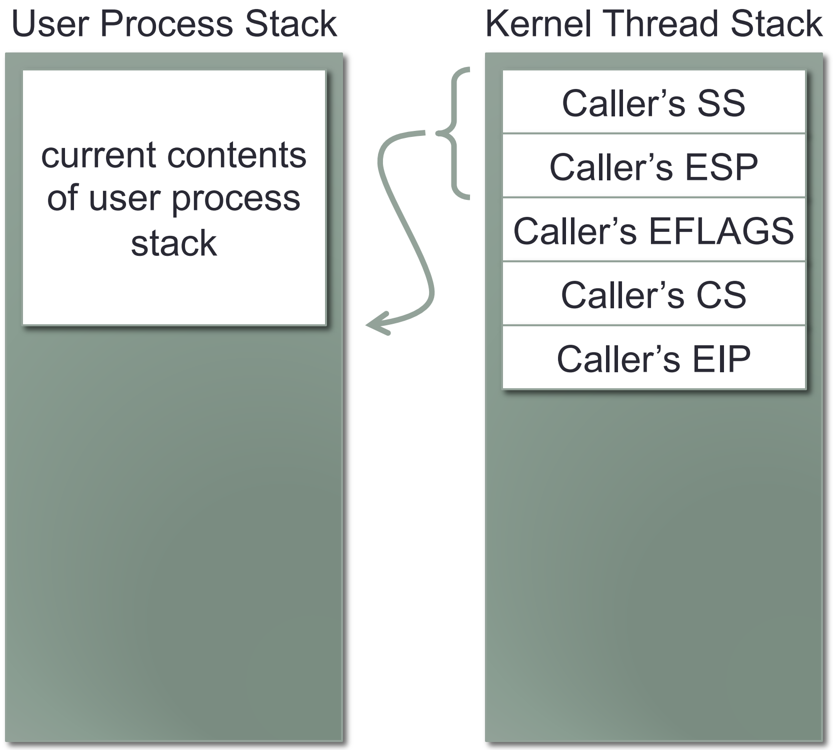
Interrupt Mechanics

- A kernel control path starts with a software trap or a hardware interrupt that causes entry (or reentry) into the kernel
- Again, this mechanism is very hardware-specific
- Example: IA32 has a stack per protection level…
  - User process has its own stack
  - The trap or interrupt causes the CPU to switch to the kernel-mode stack associated with the process
- Each user process (or thread) is typically associated with a kernel thread
  - Each user process has its own separate kernel thread context, including its own kernel stack



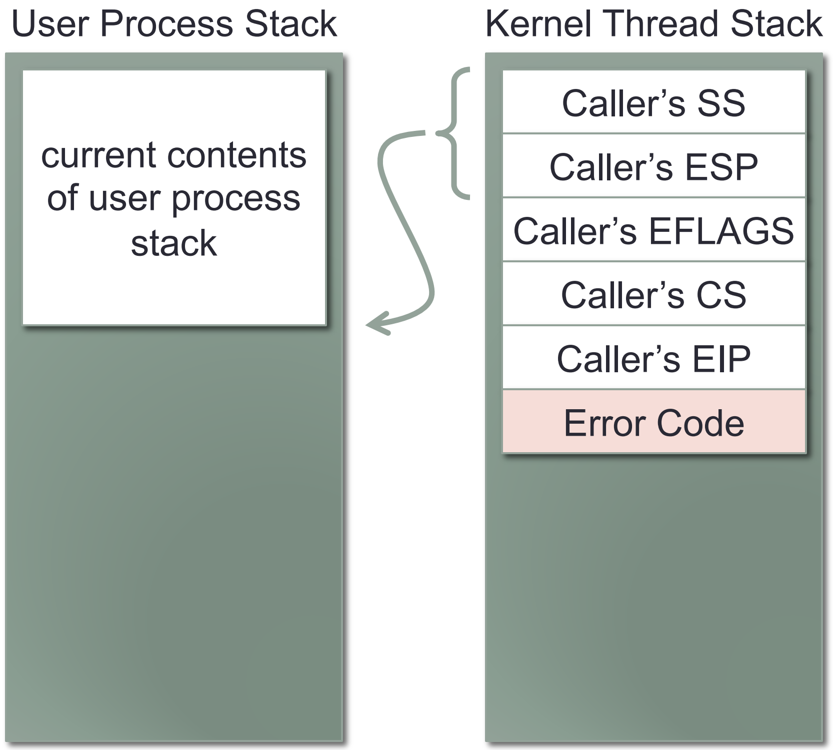
Interrupt Mechanics (2)

- When execution changes from user mode to kernel mode, IA32 saves pointer to previous stack on new stack
  - Segment and logical address of the user process' stack is stored on the kernel thread stack
  - Note: nothing is written onto the user process stack when the trap or interrupt is invoked
  - Note: caller's **`ss:esp`** isn't saved onto next stack when protection level doesn't change
- Next, the CPU saves some minimal execution state: **`cs`**, **`eip`** and **`eflags`**



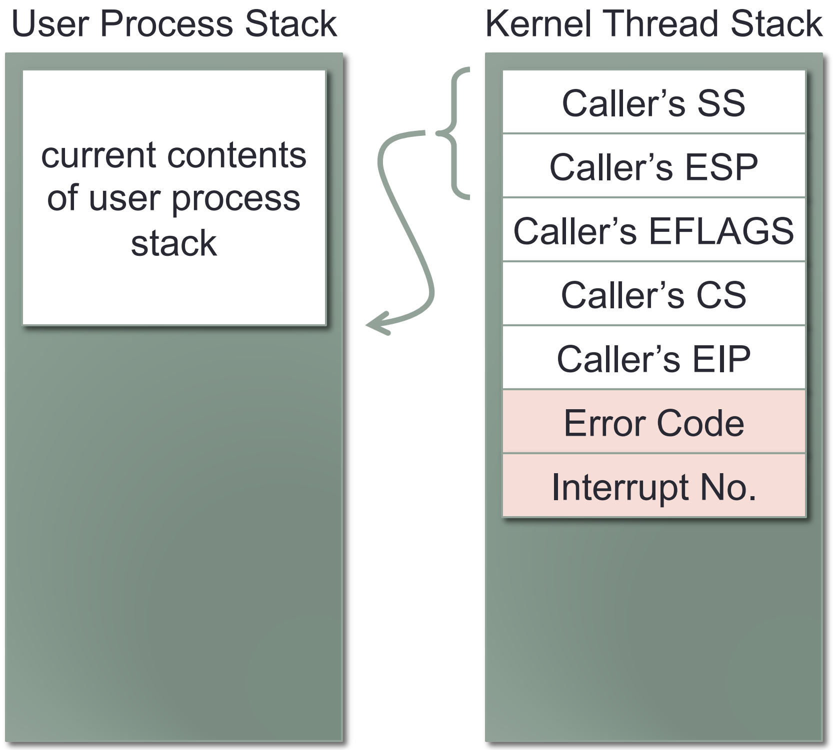
Interrupt Mechanics (3)

- So far, no difference between how IA32 handles traps, faults or interrupts
- However: Some faults also push an error code onto the kernel stack
  - e.g. page-fault interrupt (int 14 or 0x0e) pushes details of memory access causing the fault
- For interrupts without an error code, OS may push a dummy error-code value in the interrupt handler
  - Allows OS interrupt service routines to be more uniform
  - (Pintos: see threads/intr-stubs.S)
- Traps don't have an error code
  - A dummy value of 0 is pushed



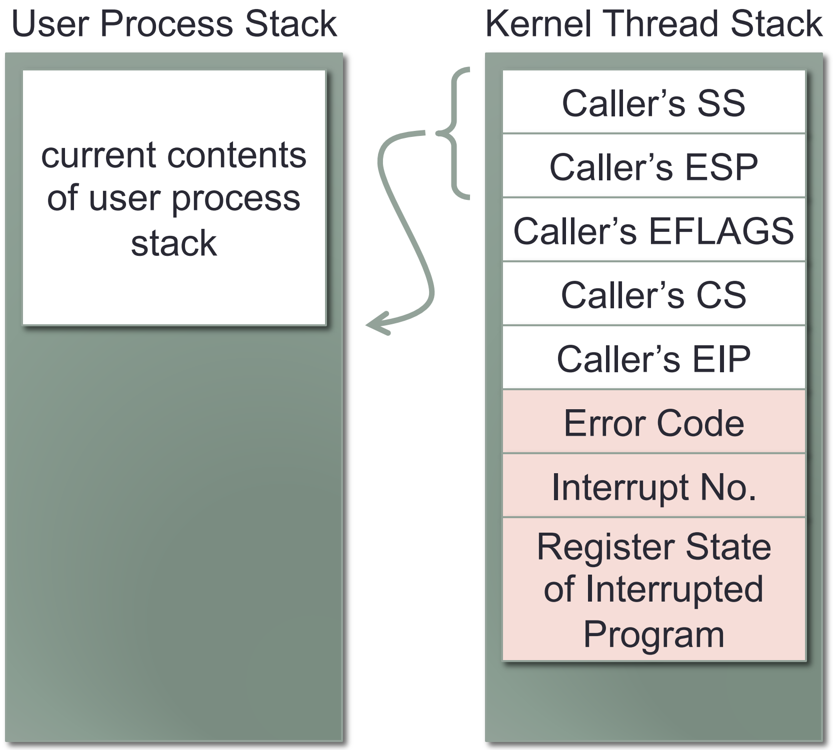
Interrupt Mechanics (4)

- OSes frequently use short stubs of assembly code to invoke their interrupt service routines (ISRs)
  - We want to write the ISR in C, not in assembly language!!!
  - Still need some assembly code to set up for the ISR
- A stub is generated for every possible interrupt on the CPU
  - Don’t want an unexpected random interrupt to kill the OS kernel
  - If interrupt doesn’t push an error code, stub pushes 0 dummy-value onto stack
  - Next, stub pushes the number of the interrupt onto stack



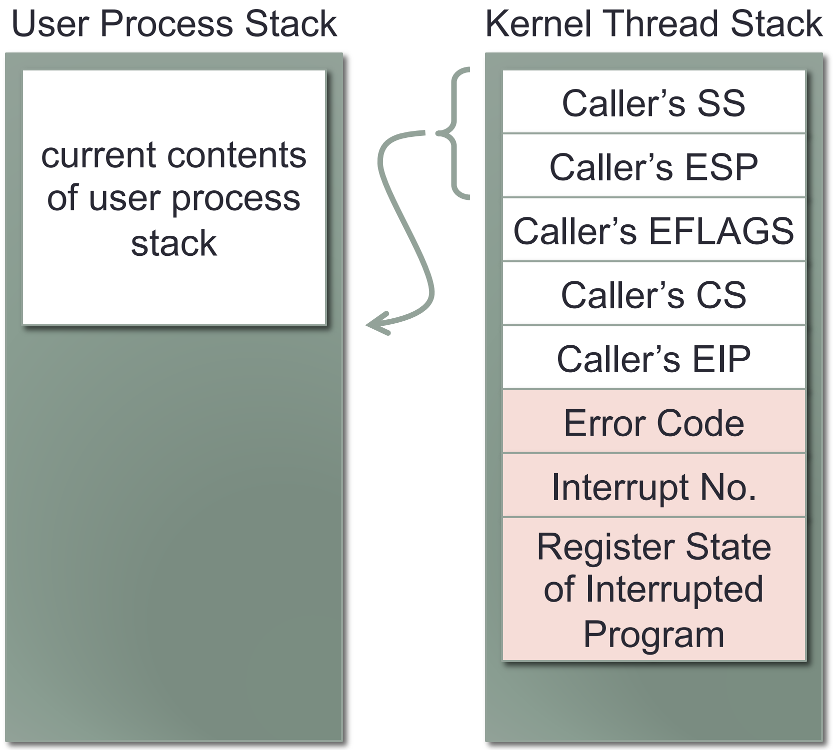
Interrupt Mechanics (5)

- Finally, the interrupt service routine (ISR) is interrupting another control flow…
  - Maybe a kernel control flow, maybe a user process
- Interrupted control flow had its own register state, etc.
- The stub records the state of all registers onto the kernel stack
- Now the ISR can run without disrupting the interrupted code



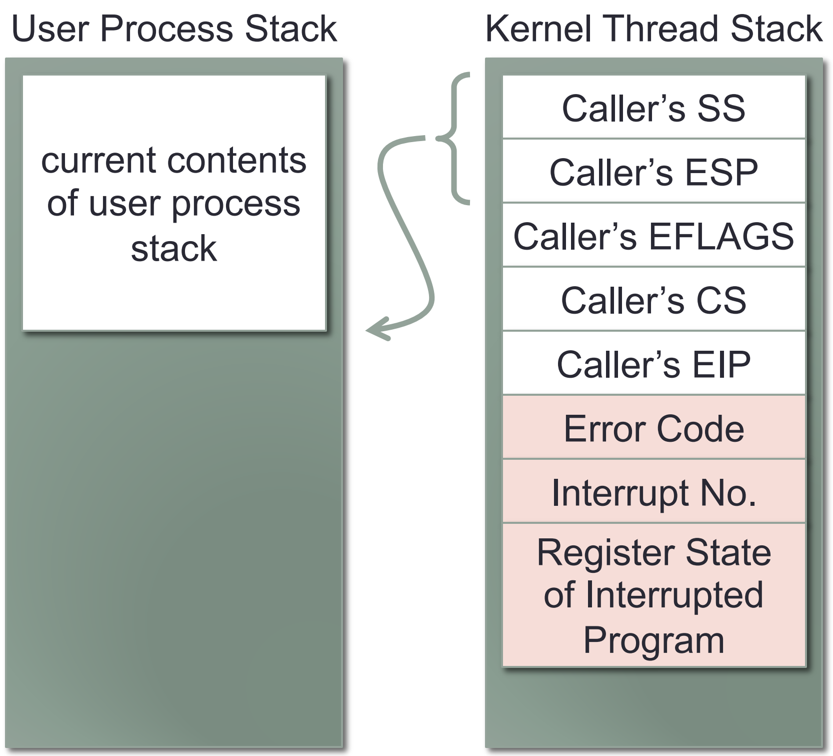
Interrupt Mechanics (6)

- The OS exposes the interrupted program's CPU and register state as arguments to the ISR
  - Typically exposed to ISR as a struct with a field for each register
  - Pintos: see intr_frame struct in threads/interrupt.c
- ISR does whatever it needs to
  - If it needs to see the interrupted program's register state, it's easily accessible
  - e.g. trap handlers may need arguments from the user program, passed in registers and/or on the user stack



Interrupt Mechanics (7)

- If the current kernel control flow is interrupted by another hardware interrupt, this process is simply repeated
  - Execution state of interrupted control flow is saved onto kernel stack
- Only difference: already in kernel mode
  - No stack switch, so caller's `ss:esp` isn't saved again
- Kernel-thread stack size constrains max nesting level of interrupts and function calls within kernel
  - Typically, interrupts don't nest very deeply
  - Kernel code must take care to not overflow the kernel stack



Kernel Control Paths (2)

- Since kernel control paths can overlap, must ensure that overlapping control paths don't corrupt shared state
  - Not possible to make all kernel control paths reentrant
  - Often, control paths must manipulate shared kernel state
- To properly synchronize kernel components, must know what kernel paths can overlap, when, and where
  - i.e. "what can interrupt what?"
  - (will revisit this question momentarily)
- Also must understand how to achieve synchronization in various scenarios

Kernel Control Paths (3)

- Example: What kernel control paths are allowed to block?
  - Blocking a kernel control path means that it can be suspended, and the kernel can switch to another control path
- Hardware interrupt / fault handlers cannot passively wait!
  - Cannot suspend an interrupt handler and resume something else…
  - Must run to completion without blocking on locks, resources, etc.
- Generally cannot use locks to guard shared state inside interrupt handlers
  - Some very specific exceptions for very specific circumstances – e.g. multicore + spinlocks
- Sometimes, the only way to prevent access to shared state from interrupt handlers is to turn interrupt-handling off
  - Want to do this as little as possible, for as short as possible…

Kernel Control Paths: Contexts

- Kernel control paths execute in specific contexts
- If the kernel code was invoked (directly or indirectly) by a user process, it is running in **process context**
    - The kernel is executing code on behalf of a specific process
    - A kernel control path in process context is allowed to block: the corresponding process is also blocked
    - Trap handlers run in process context
    - Fault handlers also (typically) run in process context (!!!!!)
- If the kernel code was invoked by a hardware interrupt handler, it is running in **interrupt context**
    - The kernel is <u>not</u> executing code on behalf of a specific process; it is servicing hardware
    - Kernel control paths in interrupt context are <u>never</u> allowed to block
    - Example: Hardware interrupt handlers really should never page-fault; if they do, it is usually a bug in the operating system

Overlapping Interrupt Handlers

- Another question: Are interrupt handlers allowed to overlap?
- Software-generated exceptions (e.g. traps for syscalls) are generally allowed to be interrupted by hardware interrupts
  - (i.e. the kernel is reentrant)
- By default, hardware interrupts aren't allowed to overlap other hardware interrupts
  - On IA32, two mechanisms are used to ensure this
- <u>1st mechanism</u>: All maskable interrupts are disabled when a hardware interrupt occurs
  - When IA32 dispatches to handler, the CPU Interrupt Flag is cleared
  - (When returning from interrupt handler, the Interrupt Flag is automatically reenabled)
  - **As long as this approach is followed, hardware interrupts are not allowed to overlap at all.**
  - (Hardware interrupt handlers can also reenable interrupts, if they want to…)

Interrupt Handlers (2)

- 2nd mechanism: IA32 APIC requires acknowledgment of hardware interrupts
  - When APIC signals an interrupt to the CPU, the interrupt handler must acknowledge the signal before APIC will report that interrupt again
  - As long as interrupt is acknowledged at end of interrupt handler, multiple occurrences of a specific interrupt will never overlap
- Pintos relies on both mechanisms
  - In Pintos, hardware interrupts can never overlap
  - Software interrupts can be interrupted by both software and hardware interrupts
    - To prevent this, handlers must turn off maskable interrupts when necessary
- More advanced operating systems selectively allow maskable hardware interrupts to overlap
  - A given kind of interrupt will still usually never overlap itself

Example: Linux Interrupt Handlers

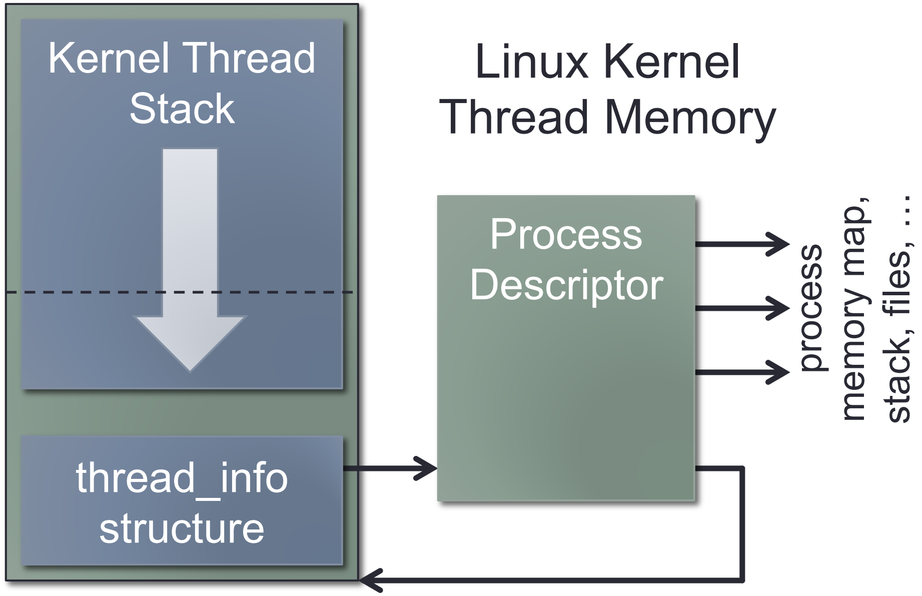
- Linux groups hardware interrupts into three categories:
- <u>Critical</u> interrupts must be performed as soon as possible
  - Handler is executed with maskable interrupts disabled
- <u>Noncritical</u> interrupts are short and can be finished quickly
  - Handler is executed with maskable interrupts enabled
  - Can be interrupted by other hardware interrupts
- <u>Noncritical deferrable</u> interrupts can be delayed for a long time without adversely affecting kernel behavior
  - May involve a longer task that shouldn't be performed within the interrupt handler routine
  - However, don't want to delay acknowledgment of the interrupt or we might miss subsequent interrupts of that type
  - Also don't want to sit in the interrupt handler for a long time: this will prevent other parts of the system from progressing

Example: Linux Interrupt Handlers (2)

- In Linux, noncritical deferrable interrupts are handled via **softirq** mechanism:
  - The interrupt handler initializes and saves a **deferrable function** to be executed later
  - Then the interrupt handler returns
- Deferrable functions are eventually executed as a batch in one of several ways, e.g.
  - At end of processing an I/O interrupt, pending softirqs are executed
  - A special kernel thread called **`ksoftirqd`** periodically checks for and executes any pending softirqs
- Other commercial OSes use similar approaches
  - Handlers that execute immediately are called "first level interrupt handlers" (older Linux versions called them "upper half" handlers)
  - Handlers whose processing is deferred are called "second level interrupt handlers" (older Linuxen: "lower" / "bottom half" handlers)

Interrupting Interrupt Handlers

- If interrupt handlers can be interrupted, must keep track of how deeply nested the interrupt handlers are
  - Only the outermost interrupt handler returns back to user mode
  - Nested interrupt handlers return to other interrupt handlers, i.e. other kernel control paths that got preempted
- Example: Linux kernel threads keep a preempt_count in their thread_info structs
  - (Interrupts execute in a kernel thread's context, and use the kernel thread's stack.)
  - ISRs increment this field on entry, decrement it on exit
  - Kernel can easily tell if a kernel thread is in the middle of handling interrupts



Non-Preemptive Kernels

- Already discussed reeentrant kernels…
  - Kernel control paths are allowed to overlap
  - (e.g. traps may be interrupted by faults, hardware interrupts, etc.)
- Most process context-switches occur at very specific points
  - e.g. when the process yields, or makes a long-running system call
  - e.g. the timer interrupt fires because the process' time-slice is up
- In these cases, the context-switch is performed at the end of kernel processing, immediately before returning to user mode
  - i.e. the kernel never leaves its work unfinished before switching back to the user-mode program
  - Called a **scheduled process-switch**
- Kernels that only allow context-switches at these points are called **non-preemptive** kernels
  - i.e. the kernel is not allowed to preempt itself
  - (this term is not related to "preemptive multitasking")

Preemptive Kernels

- **Preemptive kernels** can do **forced process-switches**
  - A process may be forced off the CPU, even when in the middle of kernel-mode code execution
- Example:
  - Process A is in the middle of a system call, copying data from a kernel buffer into the process' address space
  - An I/O interrupt fires, allowing higher-priority Process B to proceed
  - The I/O interrupt handler *forces* a context-switch from Process A to Process B, even though Process A's kernel task is incomplete
  - Kernel won't complete this task on behalf of Process A until A is again rescheduled onto the CPU
- In other words, preemptive kernels can replace the currently executing process with another process <u>at any time</u>
- Purpose: reduce **dispatch latency** in the kernel – the time between a process becoming runnable, and the process actually entering the "running" state

Preemptive Kernels (2)

- Kernel preemption primarily produces a benefit in real-time applications (both soft and hard real-time settings)
- Somewhat increases the complexity of coordinating operations within the kernel
  - Allows a kernel control path to be interrupted in more situations than a non-preemptive kernel would allow

Kernel Preemption

- Example: a kernel with preemptive multitasking
  - A timer interrupt forces the current process off of the CPU when its time-slice is completed
  - However, the process might be in the middle of a system call when the timer interrupt fires

- Approach in a non-preemptive kernel:
  - Timer ISR cannot perform a process context-switch in the middle of a system call…
  - If not in a system call, can invoke the kernel scheduler directly
  - Otherwise, the ISR can set a flag that the time-slice is completed, then return from the interrupt handler
  - The system-call handler must check this flag at the end of its processing, and perform the process context-switch if necessary

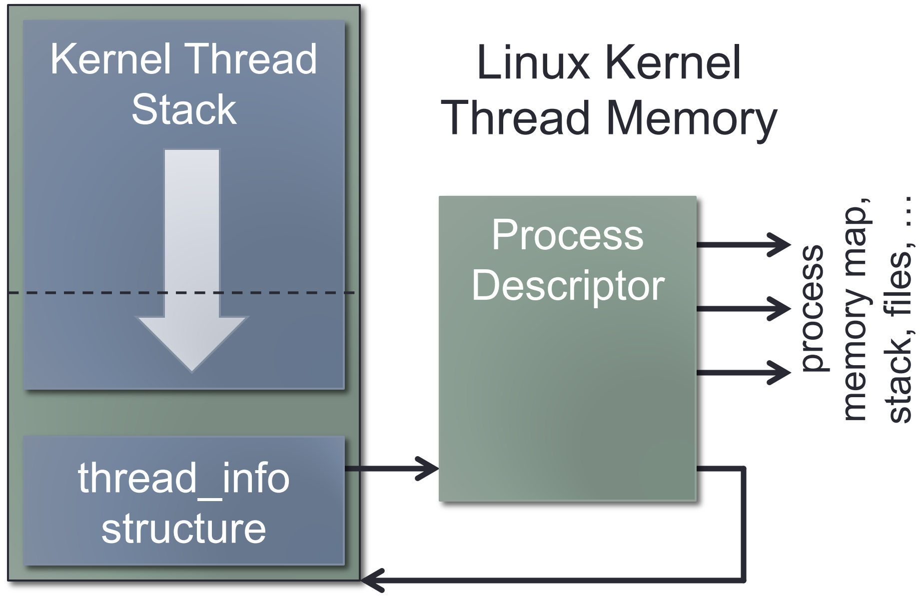
Kernel Preemption (2)

- Example: a kernel with preemptive multitasking
  - A timer interrupt forces the current process off of the CPU when its time-slice is completed
  - However, the process might be in the middle of a system call when the timer interrupt fires

- Approach in a preemptive kernel:
  - Timer interrupt handler can just invoke the process context-switch!
  - Interrupted process might be in the middle of a trap handler (i.e. a system call), but oh well! It's the new process' turn to run.
  - First process' trap handler will not complete until the kernel eventually reschedules the first process

Kernel Preemption (3)

- Even with kernel preemption, some kernel control paths are never preemptable
- Example: hardware interrupts must never be preempted; they must continually make progress towards completion
  - Even when hardware interrupt handlers are interrupted, the interruption is only for a finite, bounded period of time
  - (Cannot be interrupted by a task that can suspend)
- Linux avoids this with the preempt_count field
  - e.g. if preempt_count > 0, at least one hardware interrupt handler is running
  - Cannot preempt the kernel thread in that case



Kernel Synchronization

- Kernel control paths frequently manipulate shared state
- Clearly need synchronization primitives to guard state
- Different kernel control paths follow specific constraints
  - e.g. "Kernel control paths in an interrupt context can never block!"
  - e.g. "What kernel control paths may interrupt each other?"
  - These constraints often dictate when issues might occur, and what resolutions are valid
  - Can only solve problems effectively when you understand these constraints. Otherwise, you'll just be guessing…
- Example: Pintos is both reentrant and preemptive
  - Maskable hardware interrupts can never interrupt each other
  - Hardware interrupts can interrupt and preempt software exceptions (traps and faults), but any process-switch will occur at end of hardware interrupt handling
  - Software exceptions can interrupt software exceptions

Overlapping Kernel Control Paths

- Since kernels often have overlapping control paths, must synchronize access to shared state to avoid errors
- A **race condition** is a scenario where:
  - Two or more control paths [threads, processes, etc.] manipulate the same shared state
  - The outcome is dependent on the order that interactions take place (i.e. who wins the race)
- Manifestation of race conditions is dependent on timing
  - They don't always happen! ☹ Very difficult to reproduce and fix.
  - (May only appear when debugging/logging code is removed, etc.)

- Colloquially referred to as **Heisenbugs**

Race Conditions

- A simple example: two threads incrementing a counter
  - Threads T_1 and T_2, code is i = i + 1
  - Variable i must be read from memory, incremented, then stored
- Interleaved execution of T_1 and T_2:
  - T_1: load i into a register
  - <switch from T_1 to T_2>
  - T_2: load i into a register
  - T_2: register := register + 1
  - T_2: store register back into i
  - <switch from T_2 to T_1>
  - T_1: register := register + 1
  - T_1: store register back into i
- Final result: i has only been incremented by 1, not 2

Race Conditions (2)

- Race conditions cause much more spectacular failures
- Just one example: Northeast Blackout of 2003
  - A widespread power outage on Aug 14, 2003, affecting 45 million people in 8 US states, and 10 million people in Ontario, CA
  - A cascade failure of the electrical grids in this area of the continent
- A contributing factor was a race condition in an alarm reporting system in Ohio
  - Prevented error notifications for over an hour!
  - Operators completely unaware power grid was degrading rapidly
  - Alarm reporting system eventually crashed. Control switched over to a backup system, which also crashed nearly immediately.
- Ohio operators made decisions based on old information
  - When Ohio's power lines failed, began to bring down adjacent lines

Race Conditions (3)

- Culprit: GE Energy’s UNIX-based XA/21 alarm system
  - Approximately one million lines of code in the system
  - 8 weeks to track down the race condition in this system
- Mike Unum, manager of Commercial Solutions at GE Energy:
  - “It took us a considerable amount of time to go in and reconstruct the events.” In the end, they had to slow down the system, injecting deliberate delays in the code while feeding alarm inputs to the program.
  - The bug had a window of opportunity measured in milliseconds. “There was a couple of processes that were in contention for a common data structure, and through a software coding error in one of the application processes, they were both able to get write access to a data structure at the same time,” says Unum. “And that corruption led to the alarm event application getting into an infinite loop and spinning.”
- Ralph DiNicola, spokesman for GE Energy:
  - “This fault was so deeply embedded, it took them weeks of poring through millions of lines of code and data to find it.”

Tracking the Blackout Bug by Kevin Poulsen – https://www.theregister.com/2004/04/08/blackout_bug_report/

Critical Sections

- Race conditions can be avoided by preventing multiple control paths from accessing shared state concurrently
  - Threads, processes, etc.
- A **critical section** is a piece of code that must not be executed concurrently by multiple control paths
- **Mutual exclusion**: carefully control entry into the critical section to allow only one thread of execution at a time
- Many different tools to enforce mutual exclusion in critical sections (semaphores, mutexes, read-write locks, etc.)
  - Generally, these locks block threads (passive waiting) until they can enter the critical section
- OS kernels frequently require additional tools that are compatible with use in interrupt context (i.e. nonblocking)

Next Time

- Tools for synchronizing execution of kernel threads