CS 423
Operating System Design: Interrupts

Professor Adam Bates
Spring 2016
Goals for Today

- **Learning Objective:**
  - Understand how and under what conditions a processor receives/handles an interrupt

- **Announcements, etc:**
  - HW1 due on 2/1 23:59 UTC-11 (Or 2/2 4:00am local)
  - Note: 3 Office Hours between now and that due date.
  - End of class — special topics discussion

**Reminder:** Please put away devices at the start of class
Are kernel threads just for multithreading the kernel’s process?

What is Intel’s HyperThreading Technology?

How are threads scheduled (vs. processes)?

Static noise in lecture recordings!! >_<

Would you mind clarifying “Things suitable for threading?”

What is Intel’s HyperThreading Technology?
Would you mind clarifying "Things suitable for threading?"

Are kernel threads just for multithreading the kernel’s processes?

Static noise in lecture recordings!!

Would you mind clarifying “Things suitable for threading?”

Are kernel threads just for multithreading the kernel’s processes?

Figure 1. Multithreaded Execution with Increasing Levels of TLP Hardware Support
Are kernel threads just for multithreading the kernel’s process?

What is Intel’s HyperThreading Technology?

How are threads scheduled (vs. processes)?

Static noise in lecture recordings!! >_<

Would you mind clarifying “Things suitable for threading?”

What is Intel’s HyperThreading Technology?
Why should an application use multiple threads?

Things suitable for threading

- Block for potentially long waits
- Use many CPU cycles
- Respond to asynchronous events
- Execute functions of different importance
- Execute parallel code
Last class, we discussed how context switches plus scheduling allowed a “real” hardware CPU…
Where We Are:

To expose many “Virtual” CPUs.

Context Switching + Scheduling

The Hardware (CPU)
To expose many “Virtual” CPUs.

How does the processor know when to context switch?

Context Switching + Scheduling

The Hardware (CPU)
Where We Are:

To expose many “Virtual” CPUs.

*What’s missing from this picture?*

```
“Virtual” CPU
“Virtual” CPU
“Virtual” CPU
```

Context Switching + Scheduling

The Hardware (CPU)
Where We Are:

Interrupts to drive scheduling decisions!

Interrupt handlers are also tasks that share the CPU.

“Virtual” CPU  “Virtual” CPU  …  “Virtual” CPU

Context Switching + Scheduling

The Hardware (CPU)

Interrupt Handler

External Devices
Hardware generated:

- Different I/O devices are connected to different physical lines (pins) of an “Interrupt controller”
- Device hardware signals the corresponding line
- Interrupt controller signals the CPU (by signaling the Interrupt pin and passing an interrupt number)
- CPU saves return address after next instruction and jumps to corresponding interrupt handler
Why Hardware INTs?

- Hardware devices may need asynchronous and immediate service. For example:
  - Timer interrupt: Timers and time-dependent activities need to be updated with the passage of time at precise intervals
  - Network interrupt: The network card interrupts the CPU when data arrives from the network
  - I/O device interrupt: I/O devices (such as mouse and keyboard) issue hardware interrupts when they have input (e.g., a new character or mouse click)
Ex: Itanium 2 Pinout
LINTx — lines/pins for hardware interrupts.

In this case...

LINT0 — line for unmaskable interrupts

LINT1 — line for maskable interrupts
How does interrupt handling change the instruction cycle?
Instruction Cycle w/ INTs

How does interrupt handling change the instruction cycle?

START ➔ Fetch next instruction ➔ Execute Instruction ➔ Check for INT, init INT handler ➔ HALT

Fetch Stage ➔ Execute Stage ➔ Interrupt Stage

interrupts disabled
Device controller or other hardware issues an interrupt.

Processor finishes execution of current instruction.

Processor signals acknowledgment of interrupt.

Processor pushes PSW and PC onto control stack.

Processor loads new PC value based on interrupt.

Save remainder of state information.

Process interrupt.

Restore process state information.

Restore old PSW and PC.
Other Interrupts

- **Software Interrupts:**
  - Interrupts caused by the execution of a software instruction:
    - INT <interrupt_number>
  - Used by the system call `interrupt()`
- Initiated by the running (user level) process
- Cause current processing to be interrupted and transfers control to the corresponding interrupt handler in the kernel
Other Interrupts

- Exceptions
  - Initiated by processor hardware itself
  - Example: divide by zero
- Like a software interrupt, they cause a transfer of control to the kernel to handle the exception
Designing an Interrupt handler (top half):

- request_irq (irq, handler, flags, name, dev)
- free_irq (irq, dev)

Notes:
- handler is a pointer to the interrupt handler function
- Interrupt handlers need not be re-entrant (same irq is masked until handler exits)
- IRQ lines can be shared by multiple devices. The parameter dev is a unique “cookie” to be supplied by the given device and checked by the handler
- The kernel sequentially invokes all handlers registered for a given irq
INTs, Priorities, & Blocking

- Interrupts (as the name suggests) have the highest priority (compared to user and kernel threads) and therefore run first
  - What are the implications on regular program execution?
    - Must keep interrupt code short in order not to keep other processing stopped for a long time
    - Cannot block (regular processing does not resume until interrupt returns, so if the interrupt blocks in the middle the system “hangs”)

Can an interrupt handler use malloc()?

Can an interrupt handler write data to disk?

Can an interrupt handler use busy wait?
  - E.G. — while (!event) loop;
Interrupt Context

- Interrupt handlers execute in the kernel. They are historically “trusted” (at least in Linux)
- Problem?
  - A bad handler (e.g., a buggy device driver that does not correctly handle interrupts from its I/O device) can crash the kernel or cause inefficiencies and poor performance
Interrupt Context

- Interrupt stack is a configuration option
  - Option 1: Use the stack of the interrupted process
  - Option 2: Share the common single kernel stack
  - Option 3 (v2.6 and higher): Use own stack
A Note on Multicore

How are interrupts handled on multicore machines?

- On x86 systems each CPU gets its own local Advanced Programmable Interrupt Controller (APIC). They are wired in a way that allows routing device interrupts to any selected local APIC.
- The OS can program the APICs to determine which interrupts get routed to which CPUs.
- The default (unless OS states otherwise) is to route all interrupts to processor 0.
Designing an Interrupt Handler (Bottom Half):

- Since the interrupt handler must be minimal, all other processing related to the event that caused the interrupt must be deferred
  - Example:
    - Network interrupt causes packet to be copied from network card
    - Other processing on the packet should be deferred until its time comes
  - The deferred portion of interrupt processing is called the "Bottom Half"
soft_irq’s

- 32 handlers that must be statically defined in the Linux kernel.
- A hardware interrupt (before returning) uses raise_softirq() to mark that a given soft_irq must execute the bottom half.
- At a later time, when scheduling permits, the marked soft_irq handler is executed:
  - When a hardware interrupt is finished
  - When a process makes a system call
  - When a new process is scheduled
soft_irq types

- HI_SOFTIRQ
- TIMER_SOFTIRQ
- NET_TX_SOFTIRQ
- NET_RX_SOFTIRQ
- BLOCK_SOFTIRQ
- TASKLET_SOFTIRQ
- SCHED_SOFTIRQ
- ...

CS 423: Operating Systems Design 30
soft_irq types

- HI_SOFTIRQ
- TIMER_SOFTIRQ
- NET_TX_SOFTIRQ
- NET_RX_SOFTIRQ
- BLOCK_SOFTIRQ
- TASKLET_SOFTIRQ
- SCHED_SOFTIRQ
- ...

...
Tasklet

- Bottom halves multiplexed on top of soft_irq’s
- Scheduled using
  - tasklet_schedule()
  - tasklet_hi_schedule()
- Same tasklet invocations are serialized
- Tasklets can be created or removed dynamically
- Cannot sleep (cannot save their context)
Work Queues

- Work deferred to its own thread
- Can be scheduled together with other threads according to priorities set by a scheduling policy
- Associated with its thread control block and hence can block (and save context)
  - DECLARE_WORK(name, void (*func)(void *), void *data);
  - INIT_WORK(struct work_struct *work, void (*func)(void *), void *data);
  - schedule_work(&work);
Work Queue Examples

- Allocate the work queue data structures:
  - struct workqueue_struct *create_workqueue(const char *name);
  - struct workqueue_struct *create_singlethread_workqueue(const char *name);

- Create new work:
  - INIT_WORK(struct work_struct *work, void (*function)(void *), void *data);

- Submit the work to the workqueue:
  - int queue_work(struct workqueue_struct *queue, struct work_struct *work);

For more info see
http://www.makelinux.net/ldd3/chp-7-sect-6
What is the size of the stack in the interrupt context??

- Answer: It is a configuration option.
- Typically 2 page sizes (shared across all interrupts)
  - 8 KB on 16 bit architectures
  - 16 KB on 32 bit architectures
- In Linux 2.6 and beyond
  - 4 KB per interrupt handler
Quiz

Please remember to clearly write your name and netID

- **Q1)** Which stack does an interrupt use? (hint: there are three options – enumerate all three)
  - a)
  - b)
  - c)

- **Q2)** Name two mechanisms for executing the bottom half of an interrupt handler
  - a)
  - b)
Example: Network Interrupts

- When packets are received from the network, packet reception triggers a hardware interrupt from the network card.
- The hardware interrupt copies a packet from the network card memory and raises a soft_irq.
- The soft_irq executes later and processes the packet in the kernel (enqueues it to the receiving application).
- The application is scheduled when priority permits and reads the packet.
Example: Using Interrupts for Denial of Service

- Can one use interrupts for denial of service attacks?
Example: Using Interrupts for Denial of Service

- Can one use interrupts for denial of service attacks?
  - Yes, since the network card interrupts you when it receives a packet from the network, external parties can use that to attack you
  - Example: The SYN attack in TCP
TCP Background: A Reliable Stream Abstraction

Packetization

TCP/IP Operation:
Transparent to the user
TCP Background: Connection Establishment

Sender

SYN (seq=x)

SYN ACK (seq=y, ack=x+1)

ACK (seq=y+1)

Receiver

Example:
SeqNo = y
AckNo = x+1
SYN bit is on
ACK bit is on
Connecting to a Server

Client

connect()

Server

1. socket()
2. bind(80)
3. listen()

OS

Request from (IP, port)

Listen queue

80
Client requests get queued-up in the listen queue First-come first-served.
Busy Server Operation

Client requests get queued-up in the listen queue First-come first-served

- Hardware Interrupts copy packets from network card
- soft_irq put packets in the right application queue

Client requests get queued-up in the listen queue First-come first-served
Denial of Service

Client requests get queued-up in the listen queue First-come first-served

OS

Server

80

Listen queue

Hardware Interrupts copy packets from network card

soft_irq put packets in the right application queue

Connection establishment (SYN) requests

Client requests get queued-up in the listen queue First-come first-served
Denial of Service

Client requests get queued-up in the listen queue First-come first-served

Connection establishment (SYN) requests

OS

Server

80

accept()

Connected socket

Listen queue

soft_irq put packets in the right application queue

Hardware Interrupts copy packets from network card
Denial of Service

Client requests get queued-up in the listen queue First-come first-served

OS

Connection establishment (SYN) requests

Server

80

Listen queue

Connected socket

 accept()

Has a lower priority than the OS kernel (hence, does not get to run)

soft_irq
put packets in the right application queue

Hardware Interrupts copy packets from network card

Client 1

Client 2

Client 2

Client 2

Client 2

Client 2

Client 2

Client 2

Client 3

Client 2