CS 423
Operating System Design: Interrupts

Professor Adam Bates
Spring 2017
Goals for Today

• **Learning Objective:**
  • Work through a guided example of how interrupts interact (conflict?) with the network stack.

• **Announcements, etc:**
  • Poll for Special Topics is closed
  • HW1 due on 2/1 23:59 UTC-11 (Or 2/2 4:00am local)
  • MP Group sign-ups on Piazza. Due by Friday’s lecture.

**Reminder:** Please put away devices at the start of class
1) The Windows OS (vs. Linux OS)
2) System Start-up (Bootloaders, init, etc.)
3) Differences between Desktop and Mobile OS’
4) OS Support for Distributed Systems*

* We will pick a distributed topic that does not conflict with the rest of the curriculum

4-Credit Students: Info for sign-ups will come out shortly.
How does interrupt handling change the instruction cycle?

Fetch Stage

Execute Stage

Interrupt Stage

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

interrupts disabled
Interrupts Recap

- 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
Interrupts Recap

- 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
Interrupts Recap

- **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
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
Interrupt Context
Where does an interrupt’s stack come from?
Interrupt Context

- Where does an interrupt’s stack come from?
  - Option 1: Use the stack of the interrupted process
  - Option 2: Share a common single kernel stack
  - Option 3: Use own stack
Interrupt Handlers

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
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”
- 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);
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
Pop Quiz!
Quiz!

Please remember to write your name and NetID:

- **Q1)** Which stack can 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 INTs

- 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: Network INTs
Could we use interrupts for denial of service attacks?
Could we 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

TCP offers a reliable stream abstraction:

Write() – in append mode

Connection = Pipe

Read()
TCP offers a reliable stream abstraction:

TCP/IP Operation:
Transparent to the user

Connection = Pipe

Write() – in append mode

Read()
TCP Background

Routine for 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
TCP Server Connection

1. socket()
2. bind(80)
3. listen()
TCP Server Connection

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

Client

connect()

OS

Request from (IP, port)

Listen queue

Server

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

Server

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

OS

Connected socket

accept()

Listen queue

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