Interrupts

Interrupt: (verb) the interruption of a CPU's normal processing . . .

. . . using a mechanism provided for this purpose.

Interrupt: (noun) . . .

. . . (1) an event that causes an interrupt . . .

. . . (2) the interface and CPU hardware implementing a particular interrupt *level*.

An interrupt can be any of the following:

- Interruption by an external device.
  In class, this is what is meant by *interrupt*.
- Interruption by attempted illegal instruction or memory access.
  Also called *exceptions*.
- Interruption by timer within computer.
- "Interruption" by execution of a special instruction.
  Also called *traps*. (Used for system calls.)

In this set, external-device interrupt will be covered.

External-Device Interrupts

Hardware & Software Involved

- External Event

  Possible events

  Temperature exceeding limit.

  Person pressing a button.

  Disk drive signaling that data is ready.

- Sensor, Conditioning Circuit, Etc.

  Converts event to a logic level.
  This was covered earlier in the semester.

- Computer Input, Called *Interrupt Request (IRQ)*

  Usually several IRQs available.
  A single IRQ can be shared.

- Kernel Code Called *Service Routine*

  Attends to routine matters.

- Kernel Code Called *Handler*

  Called by service routine.
  Attends to cause of interrupt.

Interrupt Steps, Overview

Overview of Interrupt Activities

- Event occurs.

  Detected by sensor.

- An IRQ asserted by conditioning-circuit output.

- *When* CPU *allows* interruption . . .
  . . . it finishes in-progress instructions . . .
  . . . prevents (masks) other interrupts . . .
  . . . and jumps to service routine.

- Service routine . . .
  . . . saves context . . .
  . . . determines source of interrupt . . .
  . . . and calls *handler* for interrupt source.

- Handler . . .
  . . . stops interrupt . . .
  . . . and carries out interrupt-specific activities.

- After the handler returns . . .
  . . . the service routine restores registers . . .
  . . . and any interrupted task resumed.

Software Interrupts

Exceptions

These are caused by illegal instructions, operands, and memory accesses.

The service routine can usually determine the reason for the exception by examining a register.

The OS may stop the task or run a task-provided handler.

These will not be discussed further.

Traps

These are special instructions which work something like interrupts.

They are used for system calls, the type of system call is placed in a register before executing the trap.

After the trap is executed, the register's contents will be examined by the service routine.

These will not be discussed further.

Interrupt Masking

Please Do Not Disturb

Untimely interruptions cause errors, etc.

Therefore, interrupt requests sometimes temporarily ignored.

Ignored by *masking* the interrupts.

Mask Register

Interrupts masked using a *mask register*.

Mask register typically has one bit per IRQ line.

When bit is set, corresponding interrupt ignored.
(Ignored interrupts usually persist until unmasked.)

Frequently, all interrupts masked.

Reasons For Ignoring Interrupts

- Already Handling Event
- Manipulating Shared Data

Cannot stop in middle. . .
. . .without confusing next reader of shared data.

This is important, but not covered here.

- Responding to Higher-Priority Event

Done for performance reasons.

The Non-Maskable Interrupt (NMI)

Interrupt that cannot be masked.

Used for events that, if ignored, will damage system.

NMI Usage

Use of an NMI could also damage system,. . .
. . . but hopefully less than ignoring NMI.

NMIs also used to get control of “hung” system.

DOS/Windows 3.X and Macintosh users make frequent use of NMIs.

(Strong) Interrupt Priority

IRQ Choice

Several IRQs can be simultaneously asserted.

Hardware chooses using *strong priority*. . .

. . .a priority policy implemented by CPU interrupt hardware.

Priorities Levels

Usually based on labels of IRQ inputs.

E.g., IRQ3 before IRQ2.

About 10 levels typically available.

Interrupt Vector Table

The *interrupt vector table* (IVT) is. . .

. . .used by the hardware. . .

. . .to find an interrupt’s service routine.

IVT Structure

Table of memory addresses. . .

. . .kept in special place in memory.

One entry for each IRQ.

Table entry points to IRQ’s service routine.

IVT Use

Suppose IRQi is asserted while unmasked:

. . .CPU will finish current instruction. . .

. . .will read address in entry i of IVT. . .

. . .and jump to this address while switching to privileged mode.

Service Routine

Service Routine

First code executed after interrupt.

Prepares system for handler.

Service Routine Actions

- Context information saved.
- Some interrupts may be unmasked.

  IRQ that caused interrupt is not unmasked.

  (If it were the handler might never start.)
- Find source of interrupt. (Poll interrupts.)
- Start Handler

After handler finishes,

- Returns mask to its previous value.
- Return to interrupted task.

Finding Interrupt Source

Called: *interrupt polling*.

Reason: an IRQ can be shared by interrupt sources.

Side Effect: A second round of priority, *weak priority*.

Interrupt Polling

Polling: checking external devices to determine interrupt source.

Procedure

Start with list of possible sources.

Use I/O port to check each source.

Note which sources are requesting interrupt.

First Come, Maybe First Served

Interrupts can happen any time.

Suppose interrupt X is asserted on IRQ1.

Moments later interrupt Y, also on IRQ1, is asserted.

Suppose the service routine checks Y before X.

Then Y—not X—serviced first.

(X serviced after Y).

Polling Sequence and *Weak* Priority

Order of checking is called *polling sequence*.

Possible orders: round robin (with each interrupt source a class) or priority.

Priority implemented by polling sequence called *weak priority*.

Interrupt Source Choice

Polling creates something like a ready list.

Many different scheduling policies could be used, but ...
... since interrupt *latency* should be small ...
... only fast methods are used.

E.g., start handler for first active source found.

The Interrupt Hander

Interrupt handler: code written to attend to interrupt.

Interrupt handler must stop the interrupt ...
... and attend to event that caused the interrupt.

An interrupt handler should finish quickly ...
... because while it's running other interrupts may be blocked.

Blocked interrupts may miss deadlines ...
... or result in unacceptable performance.

Options for interrupts requiring lengthy service.

Handler would attend to any time-critical parts ...
... while remainder handled by either ...
... a *second-level* handler ...
... or a dæmon (or other type of) task.

Second-Level Handler

Definition: Code implementing second part of handler.

Can run with fewer interrupts masked.

Advantage: does not block higher-priority interrupts.

Two-Level Interrupts

Definition: interrupt using a second-level handler.

End of Interruption

Suppose the handler has finished, and no other same-level interrupts are pending.

Then the interrupt mask is restored to its previous value.

In task-preemptive systems, the scheduler might be called before the task returns.

Otherwise, the interrupted task will resume.

Keyboard Example

How a pressed key on a keyboard results in a character stored in a user task's memory.

This does not describe any particular system. [1]

The Hardware

Keyboard consists of a grid of switches. Pressing a key closes a switch.

Keyboard hardware generates two outputs:

An interrupt request. This is asserted when any key changes state. Suppose this is connected to IRQ3.

A scan code. This is read through an I/O port.

The Software

The IRQ3 service routine.

The handler. This reads the scan code.

The server (on X-Window systems). This converts the code into an event, and sends a message, including the event, to the appropriate task.

The task. The code for which the key is intended.

[1] I made up some details.

Sequence of Events

1: A key is pressed.

2: IRQ3 line is asserted.

3: Interrupt starts if/when level 3 unmasked.

4: At start of interrupt all interrupts are masked.

5: Jump to address stored at entry 3 in IVT, starting the service routine.

6: Context saved and some interrupts are unmasked.

7: Poll devices connected to IRQ 3.

8: Poll results: keyboard requested an interrupt, so keyboard handler started.

9: Handler reads scan code from I/O port.

10: Handler takes whatever action is necessary to stop the interrupt.

11: Scan code translated into device-independent form, called a *key code*.

12: Key code written into an area of memory accessible to server.

13: Finally, the handler signals the server that a new key is available.

14: Handler returns to the service routine.

15: Service routine returns the mask to its state before the interrupt.

16: Service routine returns to the running task.

Later.

17: Server task moves to **Run** state.

18: Server task reads key code and dispatches a message to relevant task.

Later.

19: Relevant task moves to **Run** state.

20: Finds message containing key value.

The server's work in processing keyboard input could have been done by the handler.

However, that might result in poor performance because of the handler taking too long to run.