Chapter 14 -- Exception Handling

```
EXCEPTION HANDLERS
------------------

The trouble with programmed I/O is that it both wastes CPU
resources and it has potential for "incorrect" operation.

What we really want:
  (Since most I/O devices are slow), have I/O devices signal
  the CPU when they have a change in status.

  The I/O devices tell the CPU that they are "ready."

In order to do this we need:
  Hardware (wires) from devices to the CPU.
  A way for special software to be invoked when the a device
  signals on the wire.


The modern solution bundles the software to deal with
these signals (interrupts) and other situations  into
an EXCEPTION HANDLER.  (Effectively part of the OS.)


EXCEPTIONS
----------
 1.  interrupts
     --initiated outside the instruction stream
     --arrive asynchronously (at no specific time)

     examples:
       I/O device status change
       I/O device error condition
       thermal override shutdown
       internal error detection

     when should the interrupt be dealt with?
       as soon as possible

 2.  traps
     --occur due to something in instruction stream
     --arrive synchronously (while instruction is executing)
         good test:  if program was re-run, the trap would
         occur in precisely the same place in the code.

     examples:
       bad opcode
       arithmetic overflow
       I/O functionality, like put_ch
       attempt to access privileged or unavailable memory

     when should the trap be dealt with?
       right now!  The user program cannot continue until
       whatever caused the trap is dealt with.



exception handling
------------------

the mechanism for dealing with exceptions is simple; its
implementation can get complex.  The implementation varies
```

among computers (manufactures).

  situation: a user program is running (executing), and
              a device generates an interrupt request.
  mechanism to respond:
              the hardware temporarily "suspends" the user
              program, and instead runs code called
              an EXCEPTION HANDLER. After the handler
              is finished doing whatever it needs to,
              the hardware returns control to the user program.

  limitations of exception handler:
     since it is being invoked (potentially) in the middle
     of a user program, the handler must take extra care
     not to change the state of the user program.
       -- it can't change register values
       -- it can't change the stack
     So, how can it do anything at all?
       The key to this answer is that any portion of the
       state that it wants to change, it must save the
       state and also restore it before returning to the
       user program.

       The handler often uses a stack to temporarily
       store register values.

WHEN to handle an interrupt -- 2 possiblilities:
  1. right now! Note that this could be in the middle of
      an instruction. In order to do this, the hardware
      must be able to know where the instruction is in
      its execution and be able to "take up where it left off"

      This is very difficult to do.
      But, it has been done in simpler forms on a few machines.
      Example: IBM 360 arbitrary memory to memory copy

  2. wait until the currently executing instruction finishes,
      then handle. THIS IS THE METHOD OF CHOICE.

      The instruction fetch/execute cycle must be expanded to

       1. handle pending interrupts
       2. instruction fetch
       3. PC update
       4. decode
       5. get operand(s)
       6. operation
       7. store result

some terms
----

interrupt request -- the activation of hardware somewhere that
                     signals the initial request for an interrupt.
pending interrupt -- an interrupt that hasn't been handled yet,
                      but needs to be
kernel-- the exception handler
         In most minds, when people think of a kernel, they think
         of critical portions of an operating system. The exception
         handler IS a critical portion of an operating system!
handler -- the code of the exception handler.

Pentium exception handling mechanism

---

hardware does the following:

1. signals an interrupt
   on the external pins of the chip, comes a signal that
   means an interrupt request has come in. Or, placed on
   the pins by the circuitry from within the chip,
   comes a signal that means an trap has come.

   With that signal comes a vector.
   A vector is an encoded value that categorizes the type
   of exception.

   examples: vector
   0 divide by zero (a trap, really)
   2 non-maskable interrupt
   4 overflow (a trap, really)
   6 invalid opcode (how could we get one of these?)
   7 device not available
   there can be up to 256 unique vectors

2. Uses the vector to get at the code for the exception handler.

   The vector is used as an array index. The array element
   desired is a Descriptor for the exception handler code.

   a Descriptor: (Intel calls this a "gate".)

   segment selector offset bits 15..0

   offset bits 31..16 P DBL 0111 I/T 000 reserved

   P -- whether the exception handler is in memory or not

   I/T -- distinguishes between trap and interrupt
   (1 for trap, 0 for interrupt)

   DBL --

   segment selector -- an index into yet another array. This
   array element will contain a base address within memory
   to the segment where the exception handler code is.

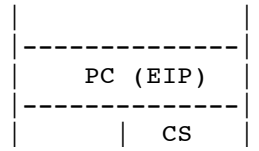
   offset bits 31..0 -- offset from base address to location
   of the exception handler code.

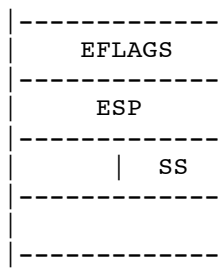
3. Save current program's state.

   This stuff logically belongs on a stack. Intel (like many
   other machines) has several stacks. There is one set
   aside just for use when an exception occurs.

   Yet another table keeps the values of the stack pointers
   within the stacks.

   The stack for exceptions will have (after saving state)





Note that we now have saved away the return address for
when the exception handling is finished.

Then, the code within the exception handler is run.
It does whatever it needed to do.

When the exception has been handled, it is time to restore
the state of the previously running code, and then go back
to executing that code.

What has been done by the hardware (saving state, setting
new values for registers), must be undone by the hardware!
On this architecture, this is accomplished by a single
instruction called iret.
   iret pops stuff off the stack (the one for exception handlers),
   and puts the stuff back in the right place.
     The PC is restored.
     The EFLAGS register is restored.
     The previous value of ESP is restored.

some advanced topics

PRIVILEDGE

The operating system (OS) needs to be able to control access
to ALL computer system resources.

  Some resources:
    the processor (for executing code!)
    main memory
    all I/O devices
    programs (both applications and the OS code)

As a simplification, there are 2 important ways that
most computer systems (architecture's really) use to acheive
access control of resources:
  1. memory access restriction.
  Each program is allocated a portion of memory. This portion
  will contain program code and data, and space for whatever
  is needed by the program.

  At EACH memory access, the address is checked (by hardware)
  to see if the address falls within the boundaries set for the
  program. If the address is OK, the memory access continues.
  If the address is not within the program's allowable memory,
  then the hardware generates an exception (an addressing error
  exception).

  2. privileged instructions.

Specific instructions within the instruction set can only be executed by the OS code. To make this work, there must be one or more bits in a register somewhere that identify the privilege level of the currently running program. This bit is checked each time an instruction is decoded. Each instruction has its own required privilege level.

If the current privilege level isn't enough to execute a decoded instruction, then the hardware must generate an exception (trap).

Intel's implementation of this:

There are 4 levels of privilege
- 0 The most privileged, allowed to do everything/anything. The OS will be given this level of privilege.
- 1
- 2
- 3 The lowest privilege, restricted. Most applications (user programs) will have this level.

Associated with each program is a descriptor. In that descriptor is the Current Priviledge Level (CPL).

Associated with each segment of memory is a descriptor. In that descriptor is the Descriptor Priviledge Level (DPL). The DPL is the privilege level needed to access memory within the boundaries of the segment.

Associated with every instruction in the instruction set is a required privilege level. (Intel determines this.) An example of an instruction that requires privilege level 0 for execution is the instruction that loads the IDTR.

At each instruction decoding:
The CPL is compared with the instruction's required privilege level. If CPL is not privileged enough (when CPL > required privilege level), an exception is generated.

At each memory access:
The CPL is compared with the DPL. If CPL is not privileged enough (when CPL > DPL), an exception is generated. Note that these are memory accesses for EITHER instruction fetches OR data.

PRIORITIES

problem: Multiple interrupt requests can arrive simultaneously. Which one should get handled first?

general solutions:

FCFS -- the first one to arrive gets handled first.

difficulty 1) This might allow a malicious/recalcitrant device or program to gain control of the processor.

difficulty 2) There must be hardware that maintains an ordering of pending exceptions.

prioritize all exceptions -- the one with the highest priority

gets handled first. This is a common method for solving the problem.

Priorities for various exceptions are assigned either by the manufacturer, or by a system manager through software. The priorities are normally set when a machine is booted (the OS is started up).

difficulty 1) Exceptions with the same priority must still be handled in some order. Example of same priority exceptions might be all keyboard interrupts. Consider a machine with many terminals hooked up.

The instruction fetch/execute cycle becomes:

1. any interrupts with a higher priority than whatever is currently running pending?
2. fetch
3. PC update
4. decode
5. operands
6. operation
7. result

NOTE: This implies that there is some hardware notion of the priority for whatever is running (user program, keyboard interrupts, clock interrupt, etc.)

Intel's solution:
Hardware is placed in a chip separate from the processor. This separate chip is called the Programmable Interrupt Controller (PIC), and it makes the decisions about what interrupts are given to the processor, and which come first.

In general, what should get given the highest priority?
clock? power failure? thermal shutdown? arithmetic overflow? keyboard? I/O device ready?

priorities are a matter of which is most urgent, and therefore cannot wait, and how long it takes to process the interrupt.

-- clock is urgent, and takes little processing, maybe only a variable increment.
-- power failure is very urgent, but takes a lot or processing, because the machine will be stopped.
-- overflow is urgent to the program which caused it, because it cannot continue.
-- keyboard is urgent because we don't want to lose a second key press before the first is handled.

REENTRANT EXCEPTION HANDLERS

Can an exception handler itself be interrupted?
If the answer is NO, then we have what is called a nonreentrant exception handler.
Why would we want to do this?

There are many details to get right to make this possible. The instruction fetch/execute cycle remains the same. At the beginning of EVERY instruction (even those within the exception handler), a check is made if there are pending interrupts.

The instruction fetch/execute cycle must be expanded to

1. Handle a pending interrupt that is of a higher priority
than the currently executing code.
2. instruction fetch
3. PC update
4. decode
5. get operand(s)
6. operation
7. store result

The exception handler must be modified so that it can
be interrupted. Its own state must be saved (safely).
Nothing can interrupt while this state is being saved.

The Intel implementation of this allocates a bit within the
EFLAGS register, called the Interrupt Enable Flag (bit #9).
When this bit is 0, interrupts (maskable ones) are disabled.
When this bit is 1, interrupts are enabled.
On the way to executing the code of an exception handler,
IF is cleared.
The iret instruction restores interrupts to their enabled
state.

With the IF bit, we automatically have nonreentrant exception
handlers.

To allow reentrant handlers (ones that can be interrupted),
the exception handler must reenable interrupts.