```
6.S081 2020 Lecture 9: Interrupts

Interrupts
  hardware wants attention now!
    e.g., pkt arrived, clock interrupt
  software must set aside current work and respond
    on RISC-V use same trap mechanism as for syscalls and exceptions
  new issues/complications:
    asynchronous
      interrupts running process
      interrupt handler may not run in context of process who caused interrupt
    concurrency
      devices and process run in parallel
    programming devices
      device can be difficult to program

Where do device interrupts come from?
  diagram: Fig 1 of SiFive board in FU540-C000-v1.0.pdf
    CPUs, CLINT, PLIC, devices
      Fig 3 has more detail for interrupts
      Section 8 and 9 of FU540-C000-v1.0.pdf
    UART (universal asynchrnonous receiver/transmitter)
      See Section 13 of FU540-C000-v1.0.pdf
      Although the QEMU version is slightly different
        Follows http://byterunner.com/16550.html
  the interrupt tells the kernel the device hardware wants attention
  the driver (in the kernel) knows how to tell the device to do things
  often the interrupt handler calls the relevant driver
    but other arrangements are possible (schedule a thread; poll)
  [diagram: top-half/bottom-half]

Case study: console output and keyboard input
  $ ls
  how does $ show up on console?
    printing to simulated console:
      driver puts characters into UART's send FIFO
      interrupt when character has been sent
        informs driver that it can send more
  how are the characters l and s read from keyboard (and echo to console)?
    reading from simulated keyboard
      user hits key, which returns in UART interrupt
      driver gets character from UART's receive FIFO

how does kernel know which device interrupted?
  each device has a unique source/IRQ (interrupt request) number
    defined by hardware platform
    UART0 is 10 on qemu (see kernel/memlayout.h)
    different on SiFive board

RISC-V interrupt-related registers
  sie --- supervisor interrupt enabled register
    one bit per software interrupt, external interrupt, timer interrupt
  sstatus --- supervisor status register
    one bit to enable interrupts
  sip --- supervisor interrupt pending register
  scause --- supervisor cause register
  stvec --- supervisor trap vector register
  mdeleg --- machine delegate register

Let's look at how xv6 sets up the interrupt machinery
  start():
    w_sie(r_sie() | SIE_SEIE | SIE_STIE | SIE_SSIE);
  main():
    consoleinit();
```

```
      uartinit()
    plicinit();
    scheduler();
      intr_on();
         w_sstatus(r_sstatus() | SSTATUS_SIE);

Printing "$"
  shell is started with fd 0, 1, 2 for "console"
    setup by init
    Unix presents console device as a file!

  printf()
    putc()
      write(2, "$", 1)

  sys_write()
    filewrite()
      consolewrite()
        uartputc()
          add "$" to buffer
          uartstart()   // kick off sending character
            put the character in UART fifo
            return to user space ...
            while at the same time UART is sending character to console

  shell calls sys_read() to wait for input (see below)

  UART completes sending character and raises interrupt
    PLIC passes interrupt on to a CPU
    the CPU performs the following steps on interrupt:
    - If the trap is a device interrupt, and the SIE bit
      is clear, don't do any of the following.
    - Disable interrupts by clearing SIE.
      Prevents interrupts being interrupted
    - Copy the pc to sepc
    - Save the current mode (user or supervisor) in the SPP bit in sstatus.
    - Set scause to reflect the interrupt's cause.
    - Set the mode to supervisor.
    - Copy stvec to the pc.
      stvec contains kernelvec or usertrap, depending if interrupt
      happened in user space or kernel space
    - Start executing at the new pc.
    same mechanism we have seen before systems calls, pgfaults, etc.

    both kernelvec/usertrap call devintr() to check for interrupts
      external interrupt for UART
        plic_claim()
        uartintr()
          if there are more characters send them
          may send multiple characters
        plic_complete()
      return from kernelvec/usertrap
        resumes interrupted computation

What if several interrupts arrive?
  The PLIC distributes interrupts among cores
    Interrupts can be handled in parallel
  If no CPU claims the interrupt, the interrupt stays pending
    Eventually each interrupt is delivered to some CPU

Interrupts expose several forms of concurrency
  1. Between device and CPU
    Producer/consumer parallelism
  2. Interrupt may interrupt the CPU that is returning to shell (still in kernel)
    Disable interrupts when code must be atomic
```

```
   3. Interrupt may run on different CPU in parallel with shell (or returning to shell)
     Locks; topic for Wed

 Producer/consumer parallelism
   For printing
     shell is producer
     device is consumer
   To decouple the two:
     a buffer in the driver
       top-half puts chars into buffer
         wait if there is no room
         runs in the context of the calling process
       bottom half remove chars from buffer
         interrupt handler wakes up producers
         may not run the context of the shell
   Note: bottom half and top half may run in parallel on different CPUs
     We will get to this in a later lecture

 Interrupts interrupt running code
   Interrupts run between my code
   For example, my code is
     1. addi sp,sp,-48
     2. sd ra,40(sp)
   Q: Might other code run between 1 and 2?
     Yes!
     Interrupt may happen between 1 and 2
       e.g., timer interrupt or uart interrupt
   For user code maybe not that bad
     Kernel will resume user code in in the same state
   For kernel code could be difficult
     Interrupt handler may update state that is observable by my code
     my code:               interrupt:
       x = 0
       if x = 0 then        x = 1
         f()
      f() may be executed or may not be executed!
     To make a block of code "atomic", turn off interrupts
       intr_off(): w_sstatus(r_sstatus() & ~SSTATUS_SIE);
   RISC-V turns of interrupt on a trap (interrupt/exception)
     Can kernel handle interrupt in trampoline.S?
   Our first glimps of "concurrency"
     We'll get back to this when discussing locking

 $
   shell is in read system call to get input from console
     usertrap() for system call
       w_stvec((uint64)kernelvec);
       consoleread()
         sleep()
           scheduler()
             intr_on()


 $ l
   user hits l, which causes UART interrupt

 kernelvec:
   save space on current stack;  which stack?
     save registers on the current stack
     in our example, the scheduler thread's stack
   kerneltrap()
     devintr()
       uartintr()
         c = uartgetc()
         consoleintr(c)
```

```
          handle ctrl characters
          echo character ('l') using uartput_sync()
          put c in buffer
          wakeup reader
    return from devintr()
  return from kerneltrap()
  load registers back
  sret
  Q: where does sret return too
    where ever the interrupt happened (in scheduler loop in this case)
  scheduler runs shell so that it can collect 'l'

Producer/consumer parallelism
  For reading from keyboard opposite from printing
    shell is consumer
    device is producer

Interrupt evolution
  Interrupts used to be relatively fast; now they are slow
    old approach: every event causes an interrupt, simple h/w, smart s/w
    new approach: h/w completes lots of work before interrupting
  Some devices generate events faster than one per microsecond
    e.g. gigabit ethernet can deliver 1.5 million small packets / second
  An interrupt takes on the order of a microsecond
    save/restore state
    cache misses
    what to do if interrupt comes in faster than 1 per microsecond?

Polling: another way of interacting with devices
  Processor spins until device wants attention
    Wastes processor cycles if device is slow
    One example in xv6: uartputc_sync()
  But inexpensive if device is fast
    No saving of registers etc.
  If events are always waiting, no need to keep alerting the software

Polling versus interrupts
  Polling rather than interrupting, for high-rate devices
  Interrupt for low-rate devices, e.g. keyboard
    constant polling would waste CPU
  Switch between polling and interrupting automatically
    interrupt when rate is low (and polling would waste CPU cycles)
    poll when rate is high  (and interrupting would waste CPU cycles)
  Faster forwarding of interrupts to user space
    for page faults and user-handled devices
    h/w delivers directly to user, w/o kernel intervention?
    faster forwarding path through kernel?
  We will be seeing many of these topics later in the course
```