6.S081: Interrupts

Adam Belay

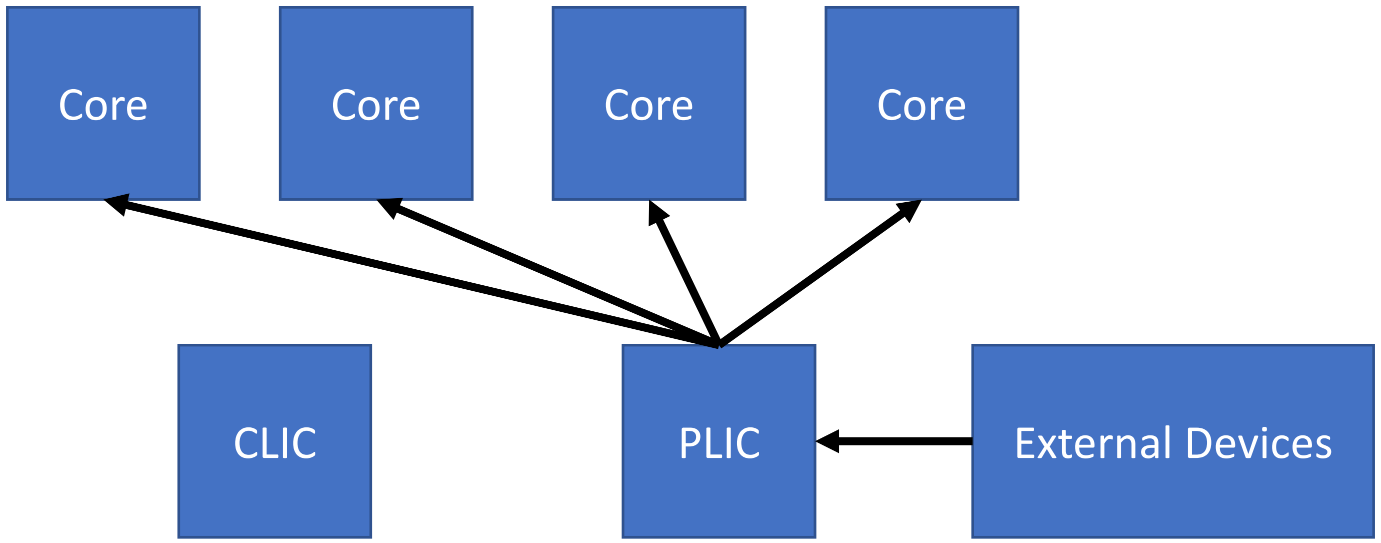
abelay@mit.edu

Interrupts

- Previous lectures: System calls + page faults
- What if hardware wants attention?
  - E.g., a packet arrived, or a clock tick
- Software must set aside current work and respond
  - For RISC-V, the mechanism is the same as for syscalls
- New complications:
  - Interrupts are asynchronous
  - Happens during a running process
  - Concurrency: device runs at same time
  - Devices are difficult to program

Where do interrupts come from?

- PLIC: Platform-level interrupt controller
  - Fields interrupts from external devices
- CLIC: Core-local interrupt controller
  - Interrupts for each core and between cores (e.g., timers)



Device interrupts

- E.g. UART – Qemu uses to display text to console
  - universal asynchronous receiver/transmitter
  - A type of “serial port”
- An interrupt tells the kernel the device needs attention
- The driver in the kernel gets the interrupt, knows how to tell the device what to do next

Case study: Console + keyboard

- $ls
  - How does ‘$’ show up on the console?
    - Printing to a simulated console (think like a line printer)
    - Driver puts characters into UART’s send FIFO
    - Interrupts when character sent
    - Driver uses interrupts to find out when it can send more
  - How are ‘l’ and ‘s’ read from the keyboard?
    - Simulated keyboard plugged into UART’s input
    - When key hit, triggers UART interrupt
    - Driver reads character from UART’s receive FIFO
    - These inputs are then printed to console (see above)

Which device caused interrupt?

- Each device has a unique source number (IRQ #)
  - Numbers defined by hardware platform
  - UART0 is 10 in QEMU, but different in RISC-V board
- Types of interrupts in RISC-V
  - External interrupts: devices like the UART
  - Software interrupts: Cores can send to each other
  - Timer interrupts: Interrupts sent after a timeout

RISC-V registers for interrupts

- **sie**: supervisor interrupt enabled register
  - One bit per software interrupt, external interrupt, and timer interrupt
- **sstatus**: supervisor status register
  - SIE bit enables or disables interrupts
  - Other bits record the mode when interrupt occurred
- **sip**: supervisor interrupt pending register
  - Which type of interrupts are waiting to be handled
- **scause**: the platform-specific cause of the interrupt
  - Also familiar exceptions like page faults are causes
- **stvec**: sets up the interrupt handler (addr in kernel)

How xv6 sets up interrupts

- kernel/start.c:
  - w_sie(r_sie() | SIE_SEIE | SIE_STIE | SIE_SSIE);
  - Enables external interrupts (SEIE), timer interrupts (STIE), and internal interrupts (SSIE)
- kernel/riscv.h:
  - intr_on(): w_sstatus(r_sstatus() | SSTATUS_SIE);
  - intr_off(): w_sstatus(r_sstatus() & ~SSTATUS_SIE);
  - SIE – supervisor interrupt enable
  - Why enable and disable interrupts?

Printing "$"

Userspace path:

1. shell is started with fd 0, 1, 2 for "console”
   - Setup by init()
   - UNIX presents console device as a file
2. printf()
   - putc()
   - write(2, "$", 1)
3. Now enter kernel via write system call

Printing "$"

Kernelspace path:

1. sys_write() – entry point from syscall table
2. filewrite() – figures out this is console file
3. uartputc() – places “$” in buffer
4. uartstart() – put the character in UART fifo
5. Return to userspace

While userspace is running, UART is concurrently delivering the character

Meanwhile back in userspace

- Shell calls sys_read(), blocks waiting for input characters
- UART still running while this happens

UART finishes sending ‘$’

- Time to raise an interrupt inside device
- PLIC passes interrupt to CPU
- CPU platform performs these steps to deliver
  1. If SIE bit is already cleared, stop here, can’t deliver yet
  2. Otherwise, clear SIE bit to disable interrupts now. This prevents interrupt handler from being interrupted.
  3. Copy the current PC to **sepc** register
  4. Save the current user or kernel mode (SPP bit in **sstatus**)
  5. Set **scause** to reflect the type of interrupt (i.e., UART)
  6. Switch to supervisor mode
  7. Jump the PC to the value in the **stvec** register (contains kerneltrap() or usertrap() depending on mode)
  8. Start executing at PC, same mechanism as system calls now

What if several interrupts arrive?

- The PLIC distributes interrupts across cores
  - In some cases, interrupts are handled in parallel
- While no core claims the interrupt, it will stay pending
- Eventually interrupt is delivered to a core

Interrupts expose concurrency

1. Between the device and the CPU
   - E.g., producer, consumer parallelism
2. Interrupts on the same core running the shell
   - Must disable interrupts when code must be atomic
3. Interrupts on a different core
   - Need locks (next topic)

Producer/consumer parallelism

- For printing to console:
  - Shell is producer
  - UART is consumer
- Need to decouple the two
  - Solution: maintain a ring buffer (like a pipe)
  - Top-half puts characters in the ring
    - Waits if there is no room
    - Runs in the context of the calling process
  - Bottom-half
    - Interrupt handler wakes up top-half
    - May not run in the shell’s context (e.g., a different core)

kernel devintr() +uart code

Example

1. addi sp,sp,-48
2. sd ra,40(sp)

- Q: Could an interrupt run between 1 and 2?

Example

1. addi sp,sp,-48
2. sd ra,40(sp)

- Q: Could an interrupt run between 1 and 2?

- Yes absolutely! (e.g., timer or UART interrupt)
- For user code this may be fine, kernel will resume code where it left off
- For kernel code, this could cause bugs
  - What if 1 and 2 access the same memory as the interrupt handler?

Interrupt race condition example

My code:

```
x = 0
if x == 0 then
  f()
```

Interrupt code:

```
x = 1
```

- F might be executed or not!
- Depends on interleaving between normal code and interrupts

Solution: Turn off interrupts

- In xv6, this can be achieved with intr_off()
- Makes the code block “atomic”
- Xv6 automatically turns off interrupts during a trap (interrupt/exception)
- Could you get an interrupt in the trampoline?

First glimpse of concurrency, will revisit next week.

Interrupt evolution

- Interrupts used to be fast relative to code, now they are slow
  - Old approach: every event triggers an interrupt (e.g., our UART code)
  - New approach: H/W completes a lot of work before interrupting
- Some devices can generate interrupts faster than once per microsecond
- But an interrupt takes about a microsecond to handle
  - Cache misses, saving register state, internal CPU delays

Alternative: Polling

- Processor spins until device wants attention
- Wastes cycles if device generates events infrequently
- But very efficient if device generates events frequently

Modern kernels use a combination of interrupts and polling