Chapter5: Interrupts and Device Drivers
Code: Console input
Code: Console Output
Concurrency in drivers
Timer interrputs
Real World
Lec09 Interrupts
Interrupt硬件部分
设备驱动概述
XV6中中断的设置
Interrupt相关的并发

Chapter5: Interrupts and Device Drivers

设备驱动程序Device driver是一段驻留在操作系统内的**代码**，用于管理特定的硬件设备。驱动程序负责设置好这些硬件设备，告诉设备要执行什么动作，处理设备产生的中断，并与正在等待该设备的上层用户进程进行交互等。编写设备驱动程序是一件相当棘手的工作，因为驱动程序和它所管理的设备通常是**并发**执行的，而且编写驱动程序的程序员要对底层硬件设备接口相当熟悉了解，这些资料往往很复杂也很稀少。

如果操作系统需要关注一些硬件设备，我们可以配置该设备，使其能够产生**中断Interrupt**，中断是引起**trap**的其中一种方式。内核的trap handler会注意到有设备发出了中断，接着内核就为该设备调用特定的**中断处理程序Interrupt handler**。在xv6中，所有设备的中断，都首先经过中断处理程序**devintr**（kernel/trap.c），再由devintr跳转到具体设备的中断处理程序（如uartintr、virtio_disk_intr等）。

许多设备驱动程序在两种上下文中执行代码：顶部 top half 在进程的内核线程中运行，底部 bottom half 在中断时执行。

top half 通过read和write这样的需要设备进行IO的系统调用被调用，会让设备开始执行一些具体的操作（例如从磁盘上读取一个块），然后代码会开始等待操作结束。当操作结束的时候，设备就会产生中断，驱动程序的bottom half开始执行，它会查看设备完成的是什么工作，在适当的时候唤醒等待该工作的进程，同时让设备开始做新的工作。

一个设备驱动程序的top half和bottom half，可以**并发**地运行在不同的CPU上。

Code: Console input

控制台驱动程序(console.c)是一个驱动结构的简单描述。

console driver接受用户的**键盘**输入，通过**UART串口**硬件连接到RISC-V。控制台驱动程序每次累积一行输入，处理特殊的输入字符，如backspace和ctrl。用户进程(如shell)使用read系统调用从控制台获取输入行。

QEMU仿真的UART是16550系列的芯片。在真实的计算机上，UART可能还会负责管理经典的RS232串行连接。在QEMU中键盘输入实际上就是由QEMU仿真的UART硬件传输到xv6内核的。

UART硬件对软件来说是一组**内存映射的控制寄存器**。RISC-V硬件将UART设备连接到事先约定好的物理地址上，对这些固定物理地址的读或写指令，相当于直接于硬件设备进行交互，而不是与RAM交互。

UART经内存映射到从物理地址0x10000000开始的部分上，有一些UART控制寄存器，每个寄存器的宽度为一个字节，如下代码：

```
// the UART control registers are memory-mapped
// at address UART0. this macro returns the
// address of one of the registers.
#define Reg(reg) ((volatile unsigned char *)(UART0 + reg))

// the UART control registers.
// some have different meanings for
// read vs write.
// see http://byterunner.com/16550.html
#define RHR 0                 // receive holding register (for input bytes)
#define THR 0                 // transmit holding register (for output bytes)
#define IER 1                 // interrupt enable register
#define IER_TX_ENABLE (1<<0)
#define IER_RX_ENABLE (1<<1)
#define FCR 2                 // FIFO control register
#define FCR_FIFO_ENABLE (1<<0)
#define FCR_FIFO_CLEAR (3<<1) // clear the content of the two FIFOs
#define ISR 2                 // interrupt status register
#define LCR 3                 // line control register
#define LCR_EIGHT_BITS (3<<0)
#define LCR_BAUD_LATCH (1<<7) // special mode to set baud rate
#define LSR 5                 // line status register
#define LSR_RX_READY (1<<0)   // input is waiting to be read from RHR
#define LSR_TX_IDLE (1<<5)    // THR can accept another character to send
```

其中有一些非常重要的控制寄存器。在**RHR**中保持着UART接收的输入，等待内核将其内容取走；**THR**保持着内核的输入，等待UART将其发送。对于内核的read或write指令，将访问对应的RHR或THR控制寄存器。

LSR包含一些位，如LSR_RX_READY表示输入字符是否等待被软件读取（这些字符从RHR寄存器中读取），每次内核读取一个字符时，UART硬件都会从其内部的FIFO缓冲区中删除它，并在FIFO为空时清除LSR中的“ready”位。以上是UART的接收部分硬件，发送部分硬件很大程度上与其相互独立。如果内核往THR中写入了一个字节，那么UART就发送该字节。LSR_TX_IDLE位就表示THR是否是空，是否可以接受要发送的字符。

xv6在main进行了控制台的初始化，调用`consoleinit`来完成，然后`consoleinit`又调用`uartinit`初始化UART（kernel/uart.c）,然后将read和write系统调用绑定为`consoleread`和`consolewrite`（对于write系统调用，会调用`filewrite`函数，会根据fd，发现要写的是一个设备，然后根据设备是console调用`consolewrite`；对于read系统调用，会调用`fileread`函数，根据fd发现要写的是一个设备，根据设备是console调用`consoleread`）。

```
void
consoleinit(void)
{
  initlock(&cons.lock, "cons");

  uartinit();

  // connect read and write system calls
  // to consoleread and consolewrite.
  devsw[CONSOLE].read = consoleread;
  devsw[CONSOLE].write = consolewrite;
}
```

`uartinit`主要的工作是写入相关的控制寄存器，配置好传输的波特率，重置FIFO缓冲区，最后开启**接收中断receive interrupt**和**发送完成中断transmit complete interrupt**。之后，每当UART收到一个字节的输入时，就会产生receive interrupt；每当UART完成一个字节的发送时，就会产生transmit complete interrupt。

```
void
uartinit(void)
{
  // disable interrupts.
  WriteReg(IER, 0x00);

  // special mode to set baud rate.
  WriteReg(LCR, LCR_BAUD_LATCH);

  // LSB for baud rate of 38.4K.
  WriteReg(0, 0x03);

  // MSB for baud rate of 38.4K.
  WriteReg(1, 0x00);

  // leave set-baud mode,
  // and set word length to 8 bits, no parity.
  WriteReg(LCR, LCR_EIGHT_BITS);

  // reset and enable FIFOs.
  WriteReg(FCR, FCR_FIFO_ENABLE | FCR_FIFO_CLEAR);

  // enable transmit and receive interrupts.
  WriteReg(IER, IER_TX_ENABLE | IER_RX_ENABLE);

  initlock(&uart_tx_lock, "uart");
}
```

xv6 shell通过init.c (user/init.c)打开的文件描述符从控制台读取。对read系统调用的调用通过内核到达**驱动程序的top half**，然后执行`consoleread`(kernel/console.c)。`consoleread`等待输入到达(通过中断处理程序)并且缓存在`cons.buf`中，然后将输入拷贝到用户空间中，当整行输入完成后，返回用户进程。如果用户还没输入完一整行，但是`cons.buf`中没有字符可以读了，所有read的进程将通过sleep系统调用进行等待。

```
struct {
  struct spinlock lock;

  // input
#define INPUT_BUF 128
  char buf[INPUT_BUF];
  uint r;  // Read index
  uint w;  // Write index
  uint e;  // Edit index
} cons;

// user read()s from the console go here.
// copy (up to) a whole input line to dst.
// user_dist indicates whether dst is a user
// or kernel address.
//
int
consoleread(int user_dst, uint64 dst, int n)
{
  uint target;
  int c;
  char cbuf;

  target = n;
  acquire(&cons.lock);
  while(n > 0){
    // wait until interrupt handler has put some
    // input into cons.buffer.
    while(cons.r == cons.w){
      if(myproc()->killed){
        release(&cons.lock);
        return -1;
      }
      sleep(&cons.r, &cons.lock);
    }

    c = cons.buf[cons.r++ % INPUT_BUF];

    if(c == C('D')){  // end-of-file
      if(n < target){
        // Save ^D for next time, to make sure
        // caller gets a 0-byte result.
        cons.r--;
      }
      break;
    }

    // copy the input byte to the user-space buffer.
    cbuf = c;
    if(either_copyout(user_dst, dst, &cbuf, 1) == -1)
      break;

    dst++;
    --n;
```

```
    if(c == '\n'){
      // a whole line has arrived, return to
      // the user-level read().
      break;
    }
  }
  release(&cons.lock);

  return target - n;
}
```

当用户输入字符时，UART硬件向CPU发出中断，从而激活xv6的trap handler。trap处理程序调用`devintr`(kernel/trap.c)，它查看RISC-V `SCAUSE`寄存器，发现中断来自外部设备。然后它请求一个叫做PLIC的硬件单元告诉它哪个设备中断了(kernel/trap.c)。如果是UART设备的中断，则`devintr`调用`uartintr`。

```
// check if it's an external interrupt or software interrupt,
// and handle it.
// returns 2 if timer interrupt,
// 1 if other device,
// 0 if not recognized.
int
devintr()
{
  uint64 scause = r_scause();

  if((scause & 0x8000000000000000L) &&
     (scause & 0xff) == 9){
    // this is a supervisor external interrupt, via PLIC.

    // irq indicates which device interrupted.
    int irq = plic_claim();

    if(irq == UART0_IRQ){
      uartintr();
    } else if(irq == VIRTIO0_IRQ){
      virtio_disk_intr();
    } else if(irq){
      printf("unexpected interrupt irq=%d\n", irq);
    }

    // the PLIC allows each device to raise at most one
    // interrupt at a time; tell the PLIC the device is
    // now allowed to interrupt again.
    if(irq)
      plic_complete(irq);

    return 1;
  } else if(scause == 0x8000000000000001L){
    // software interrupt from a machine-mode timer interrupt,
    // forwarded by timervec in kernelvec.S.
```

```
    if(cpuid() == 0){
      clockintr();
    }

    // acknowledge the software interrupt by clearing
    // the SSIP bit in sip.
    w_sip(r_sip() & ~2);

    return 2;
  } else {
    return 0;
  }
}
```

`uartintr`(kernel/UART.c)从UART硬件尝试从其控制寄存器RHR中共读出一个字符，并将它们交给console驱动程序的bottom half 即`consoleintr`(kernel/console.c);如果RHR中没有字符可以读，`uartintr`**并不会阻塞并等待，因为未来的输入将引发新的中断。**处理完RHR中的字符后，会调用`uartstart`，在该函数中，就检查UART的缓冲区中是否有需要UART发送的数据，如果有，并且THR为空，就将该字符写入THR中，UART就会发送该字节。

```
// read one input character from the UART.
// return -1 if none is waiting.
int
uartgetc(void)
{
  if(ReadReg(LSR) & 0x01){
    // input data is ready.
    return ReadReg(RHR);
  } else {
    return -1;
  }
}

// handle a uart interrupt, raised because input has
// arrived, or the uart is ready for more output, or
// both. called from trap.c.
void
uartintr(void)
{
  // read and process incoming characters.
  while(1){
    int c = uartgetc();
    if(c == -1)
      break;
    consoleintr(c);
  }

  // send buffered characters.
  acquire(&uart_tx_lock);
  uartstart();
  release(&uart_tx_lock);
}
```

`consoleintr`的工作是在`cons.buf`中累积输入字符，直到出现一整行输入。`consoleintr`会专门处理退格和其他一些字符。当换行符到达时，`consoleintr`**唤醒**一个用户进程的`consoleread`(如果有的话)。一旦被唤醒，`consoleread`将在`cons.buf`中看到一整行，将其复制到用户空间，并(通过系统调用机制)返回到用户进程。

```
// the console input interrupt handler.
// uartintr() calls this for input character.
// do erase/kill processing, append to cons.buf,
// wake up consoleread() if a whole line has arrived.
//
void
consoleintr(int c)
{
  acquire(&cons.lock);

  switch(c){
  case C('P'):  // Print process list.
    procdump();
    break;
  case C('U'):  // Kill line.
    while(cons.e != cons.w &&
          cons.buf[(cons.e-1) % INPUT_BUF] != '\n'){
      cons.e--;
      consputc(BACKSPACE);
    }
    break;
  case C('H'): // Backspace
  case '\x7f':
    if(cons.e != cons.w){
      cons.e--;
      consputc(BACKSPACE);
    }
    break;
  default:
    if(c != 0 && cons.e-cons.r < INPUT_BUF){
      c = (c == '\r') ? '\n' : c;

      // echo back to the user.
      consputc(c);

      // store for consumption by consoleread().
      cons.buf[cons.e++ % INPUT_BUF] = c;

      if(c == '\n' || c == C('D') || cons.e == cons.r+INPUT_BUF){
        // wake up consoleread() if a whole line (or end-of-file)
        // has arrived.
        cons.w = cons.e;
        wakeup(&cons.r);
      }
    }
    break;
  }

  release(&cons.lock);
```

```
}
```

总结：从用户键盘键入，到shell程序中read系统调用接收到输入的整个过程，

- shell先通过read系统调用，调用`consoleread`函数，然后进入sleep，等待用户在控制台输入字符
- 用户键入一个字符
- UART在RHR中接受到该字符，发出中断
- 接到中断，陷入trap，trap handler发现是外部设备中断，设备是UART，于是调用`uartintr`
- 发现RHR中有字符可读，调用`consoleintr`
- `consoleintr`将字符缓冲到cons.buf中，如果读到'\n'或者'ctrl+D'，就唤醒`consoleread`
- `consoleread`读出一整行的用户输入，拷贝到用户空间中

对于console input，驱动中断程序的top half是`consoleread`，bottom half是`consoleintr`。

Code: Console Output

之后看console的输出，总体流程：用户进程需要产生一些输出到控制台上，以便通过屏幕显示给用户。所以，用户进程，例如shell，通过系统调用**write**，往某个发送缓冲区里写入一些字符；我们通过UART来传输这些字符，所以我们应该往UART的**发送缓冲区**里写入它们，然后用户进程就可以返回；UART会在适当的时候，将一个字符写入**控制寄存器THR**；最后，UART将字符成功地发送到了控制台，控制台呈现输出给用户。

UART的发送缓冲区定义如下，该**发送缓冲区uart_tx_buf**由UART的驱动程序维护。

```
// the transmit output buffer.
struct spinlock uart_tx_lock;
#define UART_TX_BUF_SIZE 32
char uart_tx_buf[UART_TX_BUF_SIZE];
int uart_tx_w; // write next to uart_tx_buf[uart_tx_w++]
int uart_tx_r; // read next from uart_tx_buf[uar_tx_r++]
```

用户进程请求的write系统调用，最终将导向到UART驱动程序的top half，并执行`uartputc`（kernel/uart.c），如下所示。对于用户进程来说，只需要通过`uartputc`向发送缓冲区中写入一个字符，并且调用`uartstart`，而`uartstart`无论如何都会很快就返回；如果发送缓冲区已满，驱动程序的会暂时将该进程挂起，让其睡眠，什么时候唤醒呢？同样也是在`uartstart`中。

因为`uartputc`会导致阻塞，所以不会通过中断调用。

```
// add a character to the output buffer and tell the
// UART to start sending if it isn't already.
// blocks if the output buffer is full.
// because it may block, it can't be called
// from interrupts; it's only suitable for use
// by write().
void
uartputc(int c)
{
```

```
  acquire(&uart_tx_lock);

  if(panicked){
    for(;;)
      ;
  }

  while(1){
    if(((uart_tx_w + 1) % UART_TX_BUF_SIZE) == uart_tx_r){
      // buffer is full.
      // wait for uartstart() to open up space in the buffer.
      sleep(&uart_tx_r, &uart_tx_lock);
    } else {
      uart_tx_buf[uart_tx_w] = c;
      uart_tx_w = (uart_tx_w + 1) % UART_TX_BUF_SIZE;
      uartstart();
      release(&uart_tx_lock);
      return;
    }
  }
}
```

`uartstart`。`userstart`的主要工作是，尝试发送一个位于发送缓冲区中的字符（按**FIFO**的方式），如果**发送缓冲区为空**，或者控制寄存器**THR还持有着字符**（这代表着对于上一个字符，UART的发送**已经就绪**，只是还没发送出去），那么`uartstart`将会**直接返回**。如果条件满足可以发送，`uartstart`就会按FIFO的方式，将缓冲区中的一个字符写入寄存器**THR**中，之后UART就会发送THR中的字符；同时**唤醒**一个可能在睡眠的`uartputc`进程，表明现在发送缓冲区空出了位置，该进程可以继续往缓冲区中写入字符。

```
// if the UART is idle, and a character is waiting
// in the transmit buffer, send it.
// caller must hold uart_tx_lock.
// called from both the top- and bottom-half.
void
uartstart()
{
  while(1){
    if(uart_tx_w == uart_tx_r){
      // transmit buffer is empty.
      return;
    }

    if((ReadReg(LSR) & LSR_TX_IDLE) == 0){
      // the UART transmit holding register is full,
      // so we cannot give it another byte.
      // it will interrupt when it's ready for a new byte.
      return;
    }

    int c = uart_tx_buf[uart_tx_r];
    uart_tx_r = (uart_tx_r + 1) % UART_TX_BUF_SIZE;

    // maybe uartputc() is waiting for space in the buffer.
```

```
    wakeup(&uart_tx_r);

    WriteReg(THR, c);
  }
}
```

还有一个地方也会调用`uartstart`，就是前面提到的`uartintr`。前面讨论的情况是receive interrupt，这里相关的是transmit complete interrupt，即**每当UART发送完一个字符之后，就产生transmit complete interrupt**。同样地，通过中断引发了trap，内核检查后发现是设备中断，在`devintr`中发现是UART中断，最后又跳转到了`uartintr`。如前面所示，`uartintr`的后半部分会调用`userstart`。

你可能注意到了这其中的延续性：UART发送完一个字符，引发了中断，因此调用`userstart`继续发送新的字符。一般来说，如果一个用户进程写入了多个字符，第一个字符由`uartputc`调用`uartstart`发送，剩下则的通过transmit complete interrupt调用`uartintr`再调用`uartstart`发送。

我们可以看到，无论该字符能不能马上被UART发送出去，`uartstart`都会**很快就返回**，因此我们可以认为`uartstart`几乎是**无阻塞**的。再返回到上一层，在`uartputc`中，我们要么会调用`uartstart`，要么让当前进程挂起并睡眠，因此`uartputc`也是很快就返回的。所以，UART暴露给用户程序或内核的`uartputc`接口是**异步**的，用户进程的write可以使用这种接口，因为用户进程可以很快地返回或被挂起，从而很快地进行后续工作或者让出CPU资源。

`uartputc`也提供了**同步**，或者说**阻塞**的版本，`uartputc_sync`。该版本的接口，用于满足那些需要**马上响应**的需求，因此CPU就阻塞在某处，直到THR中的字符被发送，然后就把需要发送的新字符写入THR。你也可以看到，该字符不会写入发送缓冲区中。事实上，内核的**printf**就使用这个同步的版本，因为内核打印的消息比较重要，我们希望能尽快地显示给用户。

```
// alternate version of uartputc() that doesn't
// use interrupts, for use by kernel printf() and
// to echo characters. it spins waiting for the uart's
// output register to be empty.
void
uartputc_sync(int c)
{
  push_off();

  if(panicked){
    for(;;)
      ;
  }

  // wait for Transmit Holding Empty to be set in LSR.
  while((ReadReg(LSR) & LSR_TX_IDLE) == 0)
    ;
  WriteReg(THR, c);

  pop_off();
}
```

通过控制台输入和输出的两个案例，我们可以认识到，**设备行为**和**用户进程行为**的**解耦**是一个很好的模式，这种模式可以利用**缓冲**和**中断**两种机制来实现。控制台驱动程序可以处理用户输入，尽管当前没有用户进程想要读取输入，但即使是以后才发生的读取，也能看到这些输入；类似的，用户进程可以简单地写入数据就马上离开，而不需要等待设备驱动程序的处理。这些行为都表现出良好的效率，因为设备和用户进程的解耦允许它们**并发**地执行，这会带来很多好处，尤其是用户进程比设备快得多的时候，这种思想称为I/O并发性（**I/O Concurrency**）。

Concurrency in drivers

在`consoleread`和`consoleintr`中，我们可能已经注意到，对于驱动程序的某些数据结构（如console的缓冲），多个进程会**并发地访问**它们，因此我们需要**锁**来保护这些数据结构不被并发访问，并通过`acquire`来获取锁。

如果不使用锁的话，可能会有以下的并发问题：

- 两个不同CPU上的进程同时调用`consoleread`。
- 即使CPU已经在执行`consoleread`，但硬件要求该CPU响应一个控制台（UART）的中断。
- 当`consoleread`执行时，硬件可能会在不同的CPU上响应一个控制台（UART）中断。

在下一章我们将讨论锁，以用于解决这些并发问题。

还有一种需要注意的情况是，一个进程可能正在等待设备工作的完成，但是当该设备的工作完成并产生中断时，正在运行的是另一个进程。因为这种原因，中断处理程序不应该认为当前运行的进程就是它所要交付工作的进程。例如，中断处理程序简单地使用当前进程的页表来调用`copyout`是不安全的。因此，更好的方式是，中断处理程序只做很小一部分工作，例如将数据拷贝到缓冲区中，然后在top half中，唤醒特定的进程来完成剩下的工作。

Timer interrputs

xv6使用了**计时器中断**，维护计时器的运行，并且能因此引入基于时间片的进程切换机制（Round-Robin）。计时器中断产生时，首先在**机器模式**下产生一个软件中断，提醒内核及时处理计时器到时这一事件；在后续内核进行user trap或kernel trap的过程中，如果中断开放，就能够注意到这一软件中断，从而执行计时器到时的相应逻辑事件（如调度）。我们在第四章中已经介绍过trap的相关部分，最终将由`usertrap`或`kerneltrap`负责响应计时器发出的软件中断，执行`yield`来完成进程切换。有关进程调度的内容我们将在第七章中说明。

计时器中断来自于硬件的计时器芯片，每个CPU都有一片，xv6负责周期性地对这些计时器芯片重新编程，以重置并开始新一轮计时。

正如在上一章中所提及，需要注意的是，计时器中断在**机器模式**下处理，而不是监管者模式。在RISC-V的机器模式下执行，**页表将被禁用**，而且有专门的一套寄存器（机器模式下的寄存器以m开头命名，监管者模式下的以s开头，监管者模式下的寄存器前面已经介绍了很多）。因此，**xv6内核是不能运行在机器模式下的**，所以xv6对待计时器中断的方式，和用户空间下的trap，内核空间下的trap不同。

早在xv6的启动阶段，在`main`之前，还是机器模式下时，就进行了计时器的初始化（kernel/start.c），如下所示。主要的工作有，对CLINT（core-local interruptor）进行编程，使其在一定时延后产生中断；然后设置一个类似`trapframe`的容器`scratch`，用于在机器模式下保存寄存器和CLINT的地址；最后，设置`mtvec`，使得机器模式的trap跳转到`timervec`，并且开放计时器中断。

```
// set up to receive timer interrupts in machine mode,
// which arrive at timervec in kernelvec.S,
// which turns them into software interrupts for
// devintr() in trap.c.
void
timerinit()
{
  // each CPU has a separate source of timer interrupts.
  int id = r_mhartid();

  // ask the CLINT for a timer interrupt.
  int interval = 1000000; // cycles; about 1/10th second in qemu.
  *(uint64*)CLINT_MTIMECMP(id) = *(uint64*)CLINT_MTIME + interval;

  // prepare information in scratch[] for timervec.
  // scratch[0..3] : space for timervec to save registers.
  // scratch[4] : address of CLINT MTIMECMP register.
  // scratch[5] : desired interval (in cycles) between timer interrupts.
  uint64 *scratch = &mscratch0[32 * id];
  scratch[4] = CLINT_MTIMECMP(id);
  scratch[5] = interval;
  w_mscratch((uint64)scratch);

  // set the machine-mode trap handler.
  w_mtvec((uint64)timervec);

  // enable machine-mode interrupts.
  w_mstatus(r_mstatus() | MSTATUS_MIE);

  // enable machine-mode timer interrupts.
  w_mie(r_mie() | MIE_MTIE);
}
```

计时器中断可以在任何时刻发生，不管是在执行用户或内核代码。即使内核正在**临界区**中执行，也**不应该关闭计时器中断**。计时器的中断处理程序应该在一个，不会干扰被中断的内核代码的地方运行，因此**计时器中断处理要从机器模式下开始**。

计时器到时，xv6会陷入机器模式，陷入我们预先在寄存器`mtvec`中设置好的中断向量`timervec`中，`timervec`主要重置计时器，并且发出**软中断software interrupt**，然后立刻返回。通过发出软中断，CPU将处理计时器中断的任务交付给了内核，现在内核可以按照与普通中断相同的trap机制来处理该中断，显然内核也可以屏蔽该软件中断（例如当执行类似acquire的原子操作时，要关闭中断）。处理该软件中断的流程在`devintr`中。

机器模式下的计时器中断vector是`timervec`。它保存一些在xv6启动阶段就设置好的机器模式相关寄存器到`mscratch`中，设置CLINT以便产生下一次计时器中断，并让CPU发出软件中断，恢复寄存器，然后返回。在计时器中断处理程序中没有C代码。

```
.globl timervec
.align 4
timervec:
        # start.c has set up the memory that mscratch points to:
```

```
        # scratch[0,8,16] : register save area.
        # scratch[32] : address of CLINT's MTIMECMP register.
        # scratch[40] : desired interval between interrupts.

        csrrw a0, mscratch, a0
        sd a1, 0(a0)
        sd a2, 8(a0)
        sd a3, 16(a0)

        # schedule the next timer interrupt
        # by adding interval to mtimecmp.
        ld a1, 32(a0) # CLINT_MTIMECMP(hart)
        ld a2, 40(a0) # interval
        ld a3, 0(a1)
        add a3, a3, a2
        sd a3, 0(a1)

        # raise a supervisor software interrupt.
        li a1, 2
        csrw sip, a1

        ld a3, 16(a0)
        ld a2, 8(a0)
        ld a1, 0(a0)
        csrrw a0, mscratch, a0

        mret
```

总结：

计时器中断有两个地方的处理，一个在`timervec`里面，一个在`devintr`里面。
计时器到时后首先会自动陷入机器模式，而在xv6初始化的时候已经将`mtvec`设为`timervec`了，所以会去`timervec`里面做一些简单的工作（重新开始新一轮计时），但是在`timervec`里面我们只是重新设置了计时器，还没有处理计时器到时应该处理的事件（例如进程应该调度），而此时内核可能又在做一些重要的工作，中断可能是关闭的，所以`timervec`通过设置寄存器发出了一个软件中断（可以被屏蔽），目的就是提醒内核有外部计时器到时这个事件发生了，内核需要按trap的流程来处理它。
之后就是何时相应计时器到时，处理相应事件的问题，`usertrap`和`kerneltrap`里面都会调用`devintr`来检查是不是定时器产生了中断。只要在用户空间或者内核空间没有特别要紧的事情（有一些例外，比如在执行原子操作的时候，中断要关闭，比如锁的acquire；又或者是在`usertrap`的一开始，把`stvec`设置好为`kernelvec`之后才能重新开放中断），中断开放，那么这个软件中断就能够被响应，从而到`usertrap`或`kerneltrap`里面的`devintr`去处理，再进一步决定执行调度之类的事情。

Real World

xv6的设备中断和计时器中断，可以发生在内核空间或用户空间下。即使是在内核空间下，计时器中断也会造成线程的切换（通过yield）。这种公平地时分利用CPU的方式非常有效，特别是内核线程有时会耗费大量的时间，却没有返回到用户空间下，这会降低用户的体验。这种方便的进程/线程切换机制，却给xv6的代码带来了很大的复杂性，因为线程现在要被挂起，然后又可能在一个不同的CPU上被唤醒。如果我们设计，设备中断和计时器中断只能在用户空间下产生，那么我们的内核代码将简单得多。

使一台计算机支持所有的设备是一件很庞大的工作，因为设备类型很多，设备的特性也很多，设备和设备驱动程序之间的协议比较复杂，而且关于以上这些信息也只有较少的文档记载。事实上，现在的操作系统中的设备驱动程序代码量，要远大于内核的代码量。

UART驱动程序每次从控制寄存器RHR中读出一个字节大小的字符，这种模式称为编程I/O（**Programmed I/O**），因为我们通过软件来驱动数据传输。编程I/O的方式比较简单，但是很难用于高速率传输的场合。一种更为人熟知的方式是直接存储器访问**DMA**（**Direct Memory Access**），DMA硬件会直接从RAM中读出数据，或将数据直接写入RAM中。现代磁盘和网络设备基本都使用DMA，因为它们对传输速率的要求比较高。DMA中的驱动程序，会在RAM中准备好相应的数据后，通过写入控制寄存器来通知相应设备处理这些数据。

设备在无法预知的时间点完成工作，并不时地让CPU的注意到它，我们使用了中断这种机制。但中断有较高的CPU开销，因此高速设备如网络、磁盘控制器等要设法减少中断的次数。一种方式是，对于一大串的输入输出请求，我们执行一次中断；另一种方式是，禁用中断，而让设备驱动程序**周期性**地检查设备工作是否完成，这种技术称为**轮询Polling**。如果设备的工作很快，轮询是高效的，但如果设备经常处于闲置状态，轮询又会浪费CPU资源。一些驱动程序采用中断和轮询相交替的方式，取决于当前设备的工作负载。

UART驱动程序将数据先拷贝到内核的缓冲区中，再拷贝到用户空间下。如果数据传输速率低，这么做是可以的，但是两次的复制显然不太高效。因此，一些操作系统可以直接在用户缓冲区和设备之间拷贝数据，这通常结合DMA来实现。

Lec09 Interrupts

Interrupt硬件部分

中断与系统调用的区别：

1. asynchronous。当硬件生成中断时，Interrupt handler与当前运行的进程在CPU上没有任何关联。但如果是系统调用的话，系统调用发生在运行进程的context下。
2. concurrency。对于中断来说，CPU和生成中断的设备是并行的在运行。网卡自己独立的处理来自网络的packet，然后在某个时间点产生中断，但是同时，CPU也在运行。所以我们在CPU和设备之间是真正的并行的，我们必须管理这里的并行。
3. program device。我们这节课主要关注外部设备，例如网卡，UART，而这些设备需要被编程。每个设备都有一个编程手册，就像RISC-V有一个包含了指令和寄存器的手册一样。设备的编程手册包含了它有什么样的寄存器，它能执行什么样的操作，在读写控制寄存器的时候，设备会如何响应。不过通常来说，设备的手册不如RISC-V的手册清晰，这会使得对于设备的编程会更加复杂。

所有的设备都连接到处理器上，处理器上是通过Platform Level Interrupt Control，简称PLIC来处理设备中断。PLIC会管理来自于外设的中断。再进一步深入的查看PLIC的结构图，

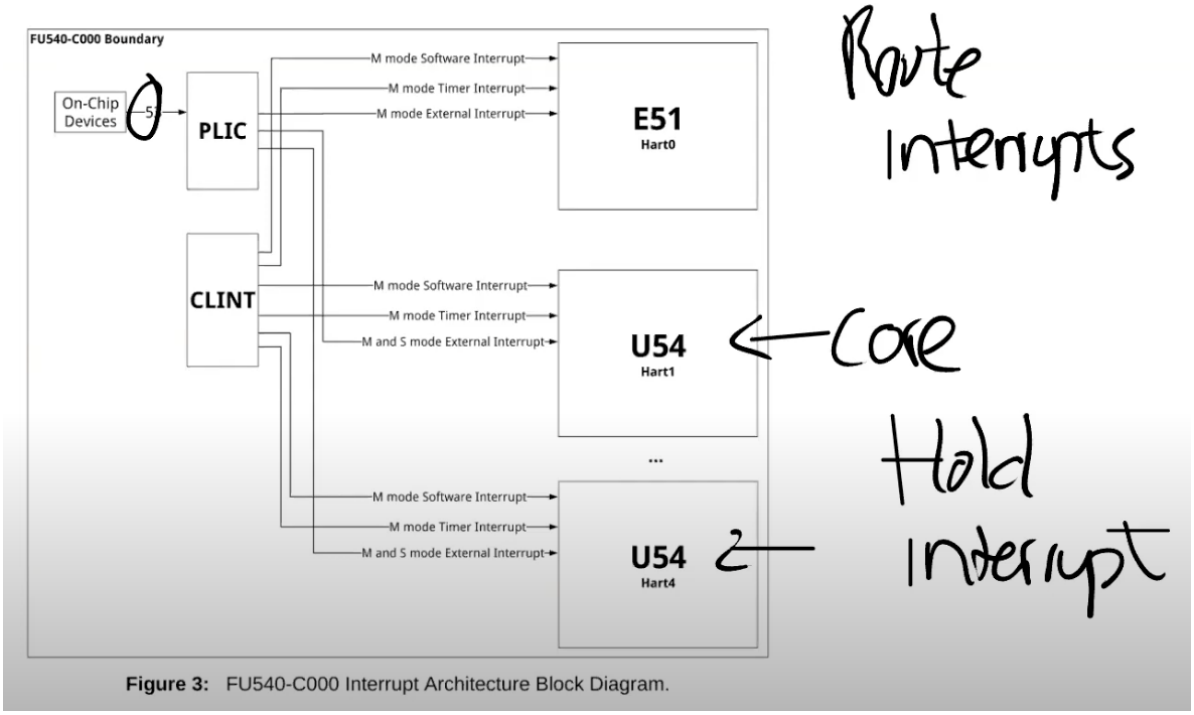

Figure 3: FU540-C000 Interrupt Architecture Block Diagram.

从左上角可以看出，有53个不同的来自于设备的中断。这些中断到达PLIC之后，PLIC会路由这些中断。图的右下角是CPU的核，PLIC会将中断路由到某一个CPU的核。如果所有的CPU核都正在处理中断，PLIC会保留中断直到有

一个CPU核可以用来处理中断。所以PLIC需要保存一些内部数据来跟踪中断的状态。

具体流程是：

- PLIC会通知当前有一个待处理的中断
- 其中一个CPU核会Claim接收中断，这样PLIC就不会把中断发给其他的CPU处理
- CPU核处理完中断之后，CPU会通知PLIC
- PLIC将不再保存中断的信息

设备驱动概述

管理设备的代码称为驱动，所有的驱动都在内核中。

大部分驱动都分为两个部分，bottom/top。bottom部分通常是Interrupt handler。当一个中断送到了CPU，并且CPU设置接收这个中断，CPU会调用相应的Interrupt handler。Interrupt handler并不运行在任何特定进程的context中，它只是处理中断。top部分，是用户进程，或者内核的其他部分调用的接口。对于UART来说，这里有read/write接口，这些接口可以被更高层级的代码调用。

XV6中中断的设置

RISC-V有许多与中断相关的寄存器：

- SIE（Supervisor Interrupt Enable）寄存器。这个寄存器中有一个bit（E）专门针对例如UART的外部设备的中断；有一个bit（S）专门针对软件中断，软件中断可能由一个CPU核触发给另一个CPU核；还有一个bit（T）专门针对定时器中断。我们这节课只关注外部设备的中断。
- SSTATUS（Supervisor Status）寄存器。这个寄存器中有一个bit来打开或者关闭中断。每一个CPU核都有独立的SIE和SSTATUS寄存器，除了通过SIE寄存器来单独控制特定的中断，还可以通过SSTATUS寄存器中的一个bit来控制所有的中断。
- SIP（Supervisor Interrupt Pending）寄存器。当发生中断时，处理器可以通过查看这个寄存器知道当前是什么类型的中断。
- SCAUSE寄存器，这个寄存器我们之前看过很多次。它会表明当前状态的原因是中断。

- STVEC寄存器，它会保存当trap，page fault或者中断发生时，CPU运行的用户程序的程序计数器，这样才能在稍后恢复程序的运行。

下面展示了XV6是如何在开始阶段对寄存器进行编程，使CPU处于能接受中断的状态。

首先是start函数：

这里将所有的中断都设置在Supervisor mode，然后设置SIE寄存器来接收External，软件和定时器中断，之后初始化定时器。

然后是main函数，看一下main函数是如何处理External中断：

第一个外设是console，这是我们print的输出位置。查看位于console.c的`consoleinit`函数。

首先初始化了锁，我们现在还不关心这个锁。然后调用了`uartinit`，`uartinit`函数位于uart.c文件。这个函数实际上就是配置好UART芯片使其可以被使用。

这里的流程是先关闭中断，之后设置波特率，设置字符长度为8bit，重置FIFO，最后再重新打开中断。

运行完这个`uartinit`函数之后，原则上UART就可以生成中断了。但是因为我们还没有对PLIC编程，所以中断不能被CPU感知。最终，在main函数中，需要调用`plicinit`函数。下图是`plicinit`函数。

PLIC与外设一样，也占用了一个I/O地址（0xC000_0000）。代码的第一行使能了UART的中断，这里实际上就是设置PLIC会接收哪些中断，进而将中断路由到CPU。类似的，代码的第二行设置PLIC接收来自IO磁盘的中断，我们这节课不会介绍这部分内容。

main函数中，`plicinit`之后就是`plicinithart`函数。`plicinit`是由0号CPU运行，之后，每个CPU的核都需要调用`plicinithart`函数表明对于哪些外设中断感兴趣。

所以在`plicinithart`函数中，每个CPU的核都表明自己对来自于UART和VIRTIO的中断感兴趣。因为我们忽略中断的优先级，所以我们将优先级设置为0。

到目前为止，我们有了生成中断的外部设备，我们有了**PLIC**可以传递中断到单个的CPU。但是CPU自己还没有设置好接收中断，因为我们还没有设置好`SSTATUS`寄存器。在main函数的最后，程序调用了`scheduler`函数，

`scheduler`函数主要是运行进程。但是在实际运行进程之前，会执行`intr_on`函数来使得CPU能接收中断。

```
// enable device interrupts
static inline void
intr_on()
{
  w_sstatus(r_sstatus() | SSTATUS_SIE);
}
```

`intr_on`函数只完成一件事情，就是设置`SSTATUS`寄存器，打开中断标志位。

在这个时间点，中断被完全打开了。如果PLIC正好有pending的中断，那么这个CPU核会收到中断。

Interrupt相关的并发

producer-consumer并发：设备与CPU是并行运行的。例如当UART向Console发送字符的时候，CPU会返回执行Shell，而Shell可能会再执行一次系统调用，向buffer中写入另一个字符，这些都是在并行的执行。

对于write来说。

Shell调用`uartputc`函数时，会将字符，例如提示符“$”，写入到写指针的位置，并将写指针加1。这就是producer对于buffer的操作。producer可以一直写入数据，直到写指针 + 1等于读指针，因为这时，buffer已经满了。当buffer满了的时候，producer必须停止运行。我们之前在`uartputc`函数中看过，如果buffer满了，代码会sleep，暂时搁置Shell并运行其他的进程。

Interrupt handler，也就是`uartintr`函数，在这个场景下是consumer，每当有一个中断，并且读指针落后于写指针，`uartintr`函数就会从读指针中读取一个字符再通过UART设备发送，并且将读指针加1。当读指针追上写指针，也就是两个指针相等的时候，buffer为空，这时就不用做任何操作。

> sleep会将当前在运行的进程存放于sleep数据中。它传入的参数是需要等待的**信号**，在这个例子中传入的是uart_tx_r的地址。在`uartstart`函数中，一旦buffer中有了空间，会调用与sleep对应的函数wakeup，传入的也是uart_tx_r的地址。任何等待在这个地址的进程都会被唤醒。有时候这种机制被称为conditional synchronization。

对于write来说，shell是producer，uart是consumer；对于read来说，shell是consumer，键盘是consumer。