引言

驱动程序需要做的事情：

- 配置硬件
- 让硬件进行操作
- 处理中断
- 和等待I/O的进程交互

驱动程序一般在如下两种情况下执行：

- 被 system call 调用，在内核进程中运行，通知设备开始某个 operation
- 在处理中断的时候被调用。通常是设备完成了某个操作后发出一个中断，此时驱动程序的中断处理部分开始运作。

Console input

值得注意的是 Console 也是一种 “设备”，需要与驱动程序交互。在 xv6 系统中，Console 的驱动程序通过 RISC-V 提供的 UART 串行端口来接收输入的数据。同时，一些用户进程用 `read` 等 system call 来从 console 中得到输入。

!!! note UART 通用异步收发传输器（Universal Asynchronous Receiver/Transmitter），通常称作UART。

软件可以通过内存映射的方式访问 UART 的控制寄存器。内存映射的其实地址定义在文件 `kernel/memlayout.h` 中

```c
// qemu puts UART registers here in physical memory.
#define UART0 0x10000000L
#define UART0_IRQ 10
```

寄存器的偏移定义在文件 `uart.c` 中

```
// 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_RX_ENABLE (1<<0)
#define IER_TX_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
```

在 `main.c` 中会调用 `consoleinit` 来初始化 UART 设备，`consoleinit` 调用 `uartinit`

```
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;
}
```

这里配置了 UART 设备使得收到一个 byte 就会生成一个中断。（通过写寄存器的方式）

xv6 的 shell 从 console 中通过一个文件描述符读取数据，这个 fd 是在 `init.c` 中打开的。

```
if(open("console", O_RDWR) < 0){
  mknod("console", CONSOLE, 0);
  open("console", O_RDWR);
}
```

当调用 system call `read` 的时候，会通过 kernel 调用 `consoleread`，`consoleread` 通过中断等待输入。

当用户在键盘上敲一个字符的时候，UART 设备通过 RISC-V 发出一个中断，中断会激活 xv6 的 trap handler，调用 `kernel/trap.c` 中的 `devintr`，这个函数再通过 RISC-V 的一个寄存器去找是哪一个外部设备触发了本次中断。如果触发中断的设备是 UART，`devintr` 就会调用 `uartintr`，处理本次中断。

`uartintr` 从 UART 硬件中读入字符，把他们扔给 `consoleintr` 处理。`consoleintr` 会把字符存在 buf 中（对 backspace 等特殊字符处理一下），当一行输入结束后，`consoleintr` 会唤醒阻塞的 `consoleread`，`consoleread` 把 buf 扔到 user space 中，并通过 system call 机制返回给 user space。

Console output

当`write` 这个 system call 被用来在一个连到 console 的 fd 上操作时，代码会执行到 `uart.c` 中的 `uartputc` 中。UART 的驱动程序会维护一个叫做 `uart_tx_buf` 的buf，`uartputc` 直接把字符扔到 buf 中，调用 `uartstart` 开始到设备的数据传输并返回。

```
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;
    }

    // 每次传一个字符
    int c = uart_tx_buf[uart_tx_r % UART_TX_BUF_SIZE];
    uart_tx_r += 1;

    // maybe uartputc() is waiting for space in the buffer.
    wakeup(&uart_tx_r);

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

当 UART 设备输送完一个 byte 之后，会产生一个中断，`trap.c` 中的 `devintr` 调用 `uartintr`，`uartintr` 调用 `uartstart`，检查是否还有别的字符要发送。

```
// trap.c
    // 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);
    }
```

Concurrency in drivers

仍然以 Console 为例，在 `consoleintr` 中，首先有这样一行代码

```
acquire(cons.lock)
```

这里获得一个锁的目的是为了保护 console 的数据结构（`cons`），防止其被其它并行的进程访问或修改。这里有三种并发的问题，这些现象可能会导致竞争或者死锁。

- 不同 CPU 上的两个进程同时调用 `consoleread`
- 在 CPU 执行 `consoleread` 的时候收到一个 `consoleintr`
- 在一个 CPU 执行 `consoleread` 的时候另一个 CPU 收到 `consoleintr`

另一个需要处理的并发问题：当一个进程等待输入的时候，表示输入的中断可能在另一个进程被执行的时候到来。因此，中断处理程序不允许依赖于正在执行的进程或是代码。

Timer interrupts

Xv6 的计数器的中断机制和其它的中断机制有所不同。

RISC-V 要求 timer interrupts 在 machine mode，而不是 supervisor mode 中被处理。在 RISC-V 中，一般的 kernel mode 代码不能直接在 machine mode 中运行，因此，xv6 用一个和 trap 机制分离的做法来实现 timer interrupt

这部分的 machine mode 代码在 `start.c` 中，大概有三部分

- 对 CLINT 硬件编程，在一定的 delay 之后生成一个中断
- 设置一个 scratch area，帮助 timer interrupt 处理程序存储寄存器和 CLINT 的寄存器的地址
- 把处理 machine mode trap 的 mtvec 设成 timervec

Timer interrupt 可能发生在程序的任何位置，无论是 user 还是 kernel 的代码执行时都可能收到 timer interrupt，即使 kernel 执行一些非常重要的操作时也不例外。于是这个中断处理程序必须要保证它不会影响到 kernel 的代码。最基本的策略是中断处理程序让 RISC-V 生成一个 “software interrupt” 并立即返回，于是 RISC-V 会把 software interrupt 用通常的 trap 机制发送到 kernel 中，这样 kernel 就可以把这个中断干掉。