第五部分：中断与设备驱动

- 第五部分：中断与设备驱动
  - 设备中断
    - devintr
    - uartintr
    - consoleintr
    - consoleputc
    - uartputc
  - 时钟中断
    - timervec
    - kerneltrap
  - 练习
    - 练习 1
    - 练习 2
  - 阅后心得
  - 参考

驱动（driver）是操作系统软件的一部分，负责提供与某个设备的连接。

一般来说，（IO）设备产生、输出数据的速度要远远地低于 CPU 处理数据的速度，所以进程并不会以轮询的方式去等待 IO 设备的输入，这样的效率很低。进程如果读不到数据就进入阻塞状态，等待设备获取、处理完读入后释放一个中断信号。

硬件接收到中断后，会通过设置 pc 为某个特定的寄存器的值，跳转到中断处理程序处。[1]

我们希望能够将操作系统的设备部分尽量解除耦合，我们主要就是用中断和缓冲技术实现。

操作系统和设备都拥有自己的缓冲区，设备驱动程序负责衔接操作系统的缓冲区和设备的缓冲区。

操作系统的进程等等只需要调用设备驱动的向上接口，把数据写入缓冲区；同时询问设备驱动程序完成下面的任务。

设备则会在需要读取和可以写入的时候发出中断，操作系统调用设备驱动程序完成数据的向上（向操作系统进程）和向下（向设备）传递。

本部分对应于 xv6 book 的 chapter 5。

设备中断

外接设备中断也是 Traps 的一部分，可能是用户态的中断，也可能是内核态的中断。无论是何种中断（除了下文提及的machine mode 中断），RISC-V 的硬件在接收到中断信号后，都会跳转到 stvec 寄存器的地址处继续运行程序。

在用户态中，stvec 会被设置为 `uservec` 的位置，位于 `kernel/trampoline.S:16`；而在内核态中，stvec 会被设置为 `kernel/kernelvec.S:10` 的位置。[2]

无论是在用户态被中断还是在内核态被中断，进入的是 usertrap 还是 kerneltrap 函数，都会调用 devintr 判断当前中断到底是什么类型。devintr 函数则会返回当前中断的类型。

QEMU 也模拟了 RISC-V 的硬件中断处理器 PLIC （这个硬件会被映射到物理地址空间的 `0x0c000000` 及之后的部分地址），其中储存了硬件中断的类型等参数。

下面先解读 devintr 函数。

devintr

```c
// 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;
  }
}
```

这段代码 dispatch 中断给各自的处理程序。

如果是外部的中断，则会从 PLIC 中读取中断的相关信息，如来源等等；如果从 PLIC 中读取中断来自UART设备 ，就继续进入 uartintr 函数处理。

uartintr

```
// 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);
}
```

UART 是字节流的输入输出设备，我们的中断有两种可能性：

1. 我们收到了很多的字符需要处理
2. 我们往缓冲区里面放置了字符，需要输出。

对于情况一：

我们调用 uartgetc 函数，读取一个字符。如果读到了 -1，证明没有多余的字符可供读取；读取字符之后调用 consoleintr 函数传递给 console 设备去处理。

对于情况二：

我们调用 uartstart 函数输出字符。

```
// 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_BUF_SIZE];
```

```
    uart_tx_r += 1;

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

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

同样，也是借助寄存器判断是否能够输出，在（缓存区已满）不能输出时 sleep（设备如果缓存区非空则会触发中断）；在缓存区非满能够输出时则向寄存器写入要输出的值，并唤醒可能因为缓存区已满而 sleep 的 uartputc 函数。

consoleintr

```
//
// 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);
    }
```

console 接收到 uartintr 传输来的字符之后，根据具体情况做处理:

1. 如果收到是特殊的字符（`C(x)` 就是 `ctrl+x` ），就做特殊的处理（例如打印所有的进程，例如删除最后一个字符）。
2. 如果收到是回车字符，就唤醒上面可能正在等待读入而 sleep 的 consoleread 函数。consoleread 函数会返回到 userlevel 的 read 函数上
3. 如果收到是正常的字符，就不做任何处理，只放到缓冲区。

可见的字符，都要在 console 上重新输出出来（不然按一个 c，为什么电脑屏幕上就会显示一个 c 呢？），这里调用的是 consoleputc 函数。

consoleputc

读完了字符，也要向屏幕输出字符啊。

```
//
// send one character to the uart.
// called by printf, and to echo input characters,
// but not from write().
//
void
consputc(int c)
{
  if(c == BACKSPACE){
    // if the user typed backspace, overwrite with a space.
    uartputc_sync('\b'); uartputc_sync(' '); uartputc_sync('\b');
  } else {
    uartputc_sync(c);
  }
}
```

这里调用了 uartputc 。

uartputc

```
// 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();
}

// 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 == uart_tx_r + UART_TX_BUF_SIZE){
      // 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 % UART_TX_BUF_SIZE] = c;
      uart_tx_w += 1;
      uartstart();
      release(&uart_tx_lock);
      return;
    }
  }
}
```

这里就是采取轮询和阻塞中断方式分别向 uart 设备的寄存器写入字符。

时钟中断

RISC-V 的硬件接收到时钟中断（是 machine mode 模式的中断）信号后，会跳转到 mtvec 寄存器的地址处继续运行程序。

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

timervec

这里的 timervec 是定义于 kernelvec.S 中的一个"函数":

xv6 处理时钟中断的方法是，尽快把其转化成 supervisor 中的软中断，然后统一在 kernel trap 中处理。

```
timervec:
        # start.c has set up the memory that mscratch points to:
        # scratch[0,8,16] : register save area.
        # scratch[24] : address of CLINT's MTIMECMP register.
        # scratch[32] : 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, 24(a0) # CLINT_MTIMECMP(hart)
        ld a2, 32(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
```

这一段代码完成了以上的事情:

1. 从 mscratch 寄存器读取需要的数据，包括 CLINT 硬件（设置下次时钟中断的时间）等。
2. 设置下一次时钟重点的时间。
3. 通过设置 sip 寄存器使中断立刻发生。

这个时候取决于操作系统处于用户态还是内核态，接受到中断信号后，操作系统会跳转进入 usertrap 函数或者 kerneltrap 函数。

两者相差不大。

kerneltrap

```
// interrupts and exceptions from kernel code go here via kernelvec,
// on whatever the current kernel stack is.
void
kerneltrap()
{
  int which_dev = 0;
  uint64 sepc = r_sepc();
```

```c
  uint64 sstatus = r_sstatus();
  uint64 scause = r_scause();

  if((sstatus & SSTATUS_SPP) == 0)
    panic("kerneltrap: not from supervisor mode");
  if(intr_get() != 0)
    panic("kerneltrap: interrupts enabled");

  if((which_dev = devintr()) == 0){
    printf("scause %p\n", scause);
    printf("sepc=%p stval=%p\n", r_sepc(), r_stval());
    panic("kerneltrap");
  }

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2 && myproc() != 0 && myproc()->state == RUNNING)
    yield();

  // the yield() may have caused some traps to occur,
  // so restore trap registers for use by kernelvec.S's sepc instruction.
  w_sepc(sepc);
  w_sstatus(sstatus);
}
```

```c
//
// handle an interrupt, exception, or system call from user space.
// called from trampoline.S
//
void
usertrap(void)
{
  int which_dev = 0;

  if((r_sstatus() & SSTATUS_SPP) != 0)
    panic("usertrap: not from user mode");

  // send interrupts and exceptions to kerneltrap(),
  // since we're now in the kernel.
  w_stvec((uint64)kernelvec);

  struct proc *p = myproc();

  // save user program counter.
  p->trapframe->epc = r_sepc();

  if(r_scause() == 8){
    // system call

    if(p->killed)
      exit(-1);

    // sepc points to the ecall instruction,
    // but we want to return to the next instruction.
    p->trapframe->epc += 4;

    // an interrupt will change sstatus &c registers,
    // so don't enable until done with those registers.
```

```
    intr_on();

    syscall();
  } else if((which_dev = devintr()) != 0){
    // ok
  } else {
    printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
    printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
    p->killed = 1;
  }

  if(p->killed)
    exit(-1);

  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2)
    yield();

  usertrapret();
}
```

在两个函数，中断处理程序均调用了 `devintr` 函数获知中断的类型。在 `devintr` 的实现中，返回值为 2 代表为时钟中断。

如果判断的确是时钟中断，无论是内核态中断处理还是用户态的中断处理，都会令当前 CPU 上正在运行的用户进程让出 CPU，由 scheduler 挑选下一个运行的进程。

练习

练习 1

https://github.com/ChenQiqian/xv6-code-report/compare/riscv...ex5

这里修改了 uart.c console.c ，将所有的中断驱动均改为轮询，从而保持 consoleread 语义不变的情况下，改用

```
diff --git a/kernel/console.c b/kernel/console.c
index 23a2d35..d1d88cc 100644
--- a/kernel/console.c
+++ b/kernel/console.c
@@ -41,17 +41,6 @@ consputc(int c)
   }
 }

-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 write()s to the console go here.
```

```diff
 //
@@ -79,109 +68,51 @@ consolewrite(int user_src, uint64 src, int n)
 int
 consoleread(int user_dst, uint64 dst, int n)
 {
-  uint target;
+  uint nowread = 0;
   int c;
   char cbuf;
+  while(nowread < n){

-  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];
+    c = consolegetc();

-    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().
+    switch(c){
+    case C('P'):  // Print process list.
+      procdump();
       break;
+    default:
+      if(c != 0){
+        c = (c == '\r') ? '\n' : c;
+        // echo back to the user.
+        consputc(c);
+        cbuf = c;
+        if(either_copyout(user_dst, dst + nowread, &cbuf, 1) == -1)
```

```
+           break;
+         nowread++;
+         if(c == '\n' || c == C('D')){
+           printf("return! %d", nowread);
+           return nowread;
+         }
+       }
      }
    }
-   release(&cons.lock);
-
-   return target - n;
+   return nowread;
  }

-//
-// 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)
+// get a char from...
+int
+consolegetc()
 {
-  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){
```

```diff
-        // wake up consoleread() if a whole line (or end-of-file)
-        // has arrived.
-        cons.w = cons.e;
-        wakeup(&cons.r);
-      }
-    }
-    break;
+  int c = uartgetc();
+  while(c == -1){
+    c = uartgetc();
   }
-
-  release(&cons.lock);
+  return c;
+
 }

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

   uartinit();

diff --git a/kernel/defs.h b/kernel/defs.h
index 3564db4..0be0234 100644
--- a/kernel/defs.h
+++ b/kernel/defs.h
@@ -19,7 +19,7 @@ void            bunpin(struct buf*);

 // console.c
 void            consoleinit(void);
-void            consoleintr(int);
+int             consolegetc(void);
 void            consputc(int);

 // exec.c
diff --git a/kernel/uart.c b/kernel/uart.c
index f75fb3c..f3d7a8c 100644
--- a/kernel/uart.c
+++ b/kernel/uart.c
@@ -41,13 +41,12 @@
 // the transmit output buffer.
 struct spinlock uart_tx_lock;
 #define UART_TX_BUF_SIZE 32
-char uart_tx_buf[UART_TX_BUF_SIZE];
+// char uart_tx_buf[UART_TX_BUF_SIZE];
 uint64 uart_tx_w; // write next to uart_tx_buf[uart_tx_w % UART_TX_BUF_SIZE]
 uint64 uart_tx_r; // read next from uart_tx_buf[uart_tx_r % UART_TX_BUF_SIZE]

 extern volatile int panicked; // from printf.c

-void uartstart();

 void
 uartinit(void)
@@ -77,35 +76,11 @@ uartinit(void)
   initlock(&uart_tx_lock, "uart");
```

```
 }

-// 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().
+// use uartputc_sync
 void
 uartputc(int c)
 {
-  acquire(&uart_tx_lock);
-
-  if(panicked){
-    for(;;)
-      ;
-  }
-
-  while(1){
-    if(uart_tx_w == uart_tx_r + UART_TX_BUF_SIZE){
-      // 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 % UART_TX_BUF_SIZE] = c;
-      uart_tx_w += 1;
-      uartstart();
-      release(&uart_tx_lock);
-      return;
-    }
-  }
+  uartputc_sync(c);
 }

 // alternate version of uartputc() that doesn't
@@ -130,35 +105,6 @@ uartputc_sync(int c)
   pop_off();
 }

-// 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_BUF_SIZE];
-     uart_tx_r += 1;
-
-     // maybe uartputc() is waiting for space in the buffer.
-     wakeup(&uart_tx_r);
-
-     WriteReg(THR, c);
-   }
-}

 // read one input character from the UART.
 // return -1 if none is waiting.
@@ -179,16 +125,5 @@ uartgetc(void)
 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);
+   return; // ignore
 }
```

练习 2

写一个网卡，其实即为大作业。在这里就不做了。

阅后心得

作者写得很巧，对一个具体案例的解读篇幅合适，也能让读者清楚地掌握整个流程。我觉得我对大作业多了些自信。

参考

http://www.databusworld.cn/10468.html

---

1. 在 RISC-V 真正的处理中，“中断”和异常被统称为 trap ， 所以之后我们也会看到许多函数的名称中都有 “trap” 。 ↩

2. 内核态切换回用户态在 usertrapret 函数中完成，其中有 `w_stvec(TRAMPOLINE + (uservec - trampoline));` 和 `p->trapframe->kernel_trap = (uint64)usertrap;` 两则语句将 stvec 和 kernel_trap 两则设置到正确的地址；在 `trapinithart` 函数， `usertrap` 函数中。 ↩