中断和异常

概述

目标

1. 了解 CPU 的中断机制
2. 了解 RISC-v 架构是如何支持 CPU 中断的
3. 掌握与软件相关的中断处理
4. 掌握时钟中断管理
5. 掌握异常处理方法

简述

在操作系统中，中断和异常机制扮演着非常重要的角色。中断和异常都是指导操作系统如何响应外部事件或者内部错误的机制。

操作系统中的中断是由外部设备发出的请求，通知操作系统需要进行相应的处理。例如，在操作系统中，键盘输入的每个字符都会通过中断机制来通知操作系统进行输入处理。当一个中断被触发时，CPU会暂停当前进程的执行并交给操作系统来处理中断事件。在操作系统中，中断可以被用于实现异步I/O、处理网络数据包、定时器等各种系统级别的功能。

异常则是指在程序执行过程中出现了不正常的情况，例如访问非法的内存地址或者除以0等错误。当一个异常被触发时，CPU会停止当前进程的执行，并将控制权转交给操作系统。操作系统可以决定如何处理这个异常，例如强制终止该进程或者尝试修复错误。在操作系统中，异常通常是由程序自身引发的错误，但也可能由硬件引发，例如页面错误（Page Fault）。

操作系统中的中断和异常机制使得操作系统能够有效地管理和响应外部事件和内部错误，从而确保系统的稳定性和可靠性。

在 RISC-V 中，处理器提供了一些特殊寄存器来支持中断和异常处理。以下是与中断和异常相关的一些寄存器：

- mstatus 寄存器：mstatus 寄存器包含了处理器的状态信息，例如当前进程的运行模式、处理器是否处于中断或异常状态等。当发生中断或异常时，处理器会将 mstatus 寄存器中的某些位修改为相应的状态值来保存当前的处理器状态。
- mip 寄存器：mip 寄存器用于保存中断请求信息。当一个中断请求被触发时，中断控制器会设置 mip 寄存器中对应的位，通知处理器需要进行中断处理。
- mie 寄存器：mie 寄存器用于控制中断响应。当 mie 寄存器中的某个中断位被打开时，处理器会响应该中断类型的中断请求。当 mie 寄存器中的某个中断位被关闭时，则不会响应该中断类型的中断请求。
- mtvec 寄存器：mtvec 寄存器用于指定中断向量表的起始地址。当一个中断被触发时，处理器会根据 mtvec 寄存器中的模式选择跳转到指定地址执行中断处理程序。
- mcause 寄存器：mcause 寄存器用于保存异常或中断的原因代码。当处理器发生异常或中断时，它会将相应的原因代码存储到 mcause 寄存器中，以便异常处理程序能够识别和处理该异常类型。
- mepc 寄存器：mepc 寄存器用于保存处理器在触发异常或中断时的执行地址。当一个异常被触发时，处理器会将当前指令的地址存储到 mepc 寄存器中，以便异常处理程序能够恢复正常的执行流程。

总之，在 RISC-V 处理器中，这些特殊寄存器与中断和异常相关，用于管理和响应中断和异常事件，从而使处理器能够实现可靠的系统级别功能。

本部分将会介绍 RISC-V 处理中断/异常的硬件机制，以及 XV6 中对这部分进行处理的代码。在实验部分，将会安排若干实验，以对中断/异常进行自定义处理。

讲解

中断

定义

首先，先来理清几个概念。

在 RISC-V architecture 上

- 中断（Interrupts）
  - 分为全局中断（Global interrupts）和本地中断（Local interrupts）
  - Global interrupts 存在仲裁，根据优先级决定谁先中断，谁去处理中断，由 PLIC （Platform-Level Interrupt Controller）管理。由外部设备产生的均为此类。
  - Local interrupts 没有仲裁，一有中断马上响应。
    - 软件中断和计时器中断是由 CLINT（Core Local Interruptor）生成的本地中断。
- 异常（Exceptions）

    - 运行时发生的，由 RISC-V 指令引起，可以说是同步的。比如：Illegal instruction，Syscall，Page fault 等
    - An unusual condition occurring at run time associated with an instruction in the current RISC-V hart
- 陷阱（Traps）
    - 转移控制流的一种机制，当某些异常/中断发生时，可以跳转到特定位置进行处理。但不是所有异常/中断都会引发 Trap。

在 RISC-V 上，异常（Exceptions）特定于体系结构的（比如 RV32I），对所有该体系结构的设备都适用。而中断是特定于实现的，不同的设备可能有自己的约定。

简单概括一下，中断

- 异步（asynchronous）。当硬件生成中断时，中断处理程序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。
- 并行（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。

RISC-V 硬件对中断的处理

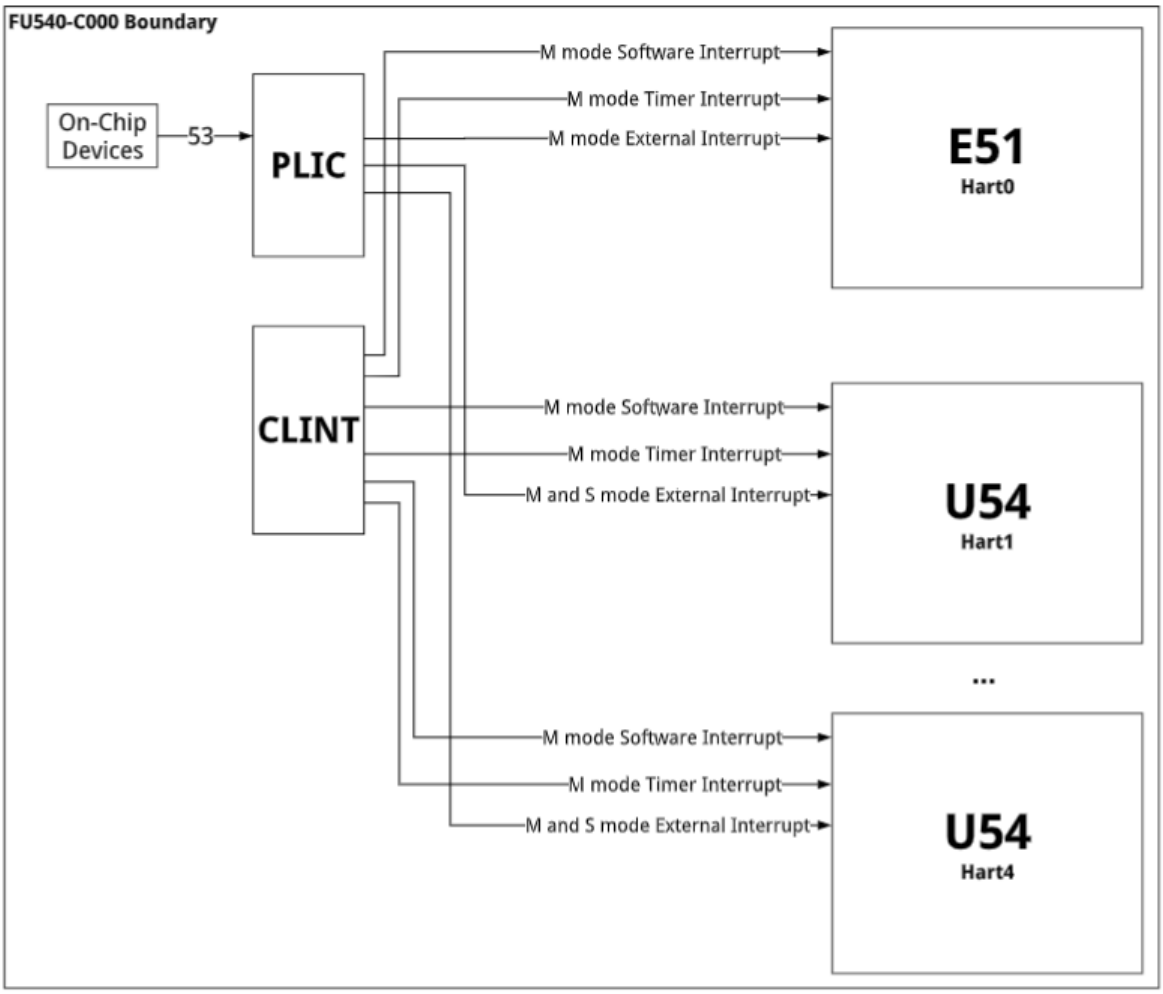


Figure 3: FU540-C000 Interrupt Architecture Block Diagram.

上图是对 RISC-V 的中断处理体系的一个示意图。

从左上角可以看出，我们有 53 个不同的来自于设备的外部中断。这些中断到达 PLIC 之后，PLIC 会路由这些中断。图的右侧是 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。

这里具体的流程是：

- 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
    - xv6 代码中的 `plic_claim()`
- CPU 核处理完中断之后，CPU 会通知 PLIC
    - xv6 代码中的 `plic_complete()`
- PLIC 将不再保存中断的信息

- 一次中断处理结束

PLIC 的机制十分灵活，可以通过控制寄存器，对 PLIC 进行编程来，告诉它中断应该分发到哪，以及决定中断的优先级。

值得注意的是，默认情况下，所有的中断对会在 M-mode 下处理，后文会介绍中断委托，使得中断可以在 S-mode 下处理。

> 问：当 UART（串口通信） 触发中断的时候，所有的 CPU 核都能收到中断吗?
> 答：取决于你如何对 PLIC 进行编程。对于 XV6 来说，所有的 CPU 都能收到中断，但是只有一个 CPU 会 Claim 相应的中断。

XV6 初始化中断寄存器

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。
- 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。

接下来我们看看 XV6 是如何对其他寄存器进行编程，使得 CPU 处于一个能接受中断的状态。

接下来看看代码，首先是位于 start. c 的 start 函数。

```
void start() {
  // set M Previous Privilege mode to Supervisor, for mret.
  unsigned long x = r_mstatus();
  x &= ~MSTATUS_MPP_MASK; // "xPP holds the previous privilege mode (x=M,S or
                          // U). The xPP fields can only hold privilege modes up
                          // to x, so MPP is two bits wide, SPP is one bit wide,
                          // and UPP is implicitly zero."
  x |= MSTATUS_MPP_S;     // set mpp to supervisor mode(0b01)
  w_mstatus(x);

  // set M Exception Program Counter to main, for mret.
  // requires gcc -mcmodel=medany
  w_mepc(x: (uint64)main);

  // disable paging for now.
  w_satp(x: 0); // 想要禁用页表地址翻译，需要将mode字段赋值0. 全部赋0更方便.

  // delegate all interrupts and exceptions to supervisor mode.
  // 默认情况下，任何特权级别的所有陷阱都在机器模式下处理
  // 为了提高性能，可以在 medeleg 和 mideleg
  // 对单个位的写入来指定特定的异常和中断应该由 RISC-V
  // 那种特权模式来处理。（通常就是 M 模式委托给 S 模式）
  w_medeleg(x: 0xffff);
  w_mideleg(x: 0xffff);

  w_sie(x: r_sie() | SIE_SEIE | SIE_STIE | SIE_SSIE);      You, 6天前 • first comm
```

默认情况下，中断，异常都是在 M-mode 下进行处理。这里我们设置了 `medeleg` 和 `mideleg` 两个 CSR ，来把所有中断和异常委托给 S-mode。

> XV6 的所有中断，异常都是在 S-mode 下进行处理的，只有一个例外，就是定时器中断（在 M-mode 处理）。关于这一点，请之后的小节 **定时器中断**。

这里将所有的中断都设置在 Supervisor mode，然后设置 `SIE` 寄存器来接收外部，软件和定时器中断。

我们第一个外部设备是 console，在 `main` 函数中，调用 `consoleinit()` 对其进行初始化。

```
void consoleinit(void) {      You, 6天前 •
  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。

```
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");
}      You, 6天前 • first commit
```

这里的流程是先关闭 UART 设备的中断，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。

以上就是 `uartinit` 函数，运行完这个函数之后，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。

```
void plicinit(void) {
  // set desired IRQ priorities non-zero (otherwise disabled).
  *(uint32 *)(PLIC + UART0_IRQ * 4) = 1;
  *(uint32 *)(PLIC + VIRTIO0_IRQ * 4) = 1;
}      You, 6天前 • first commit
```

PLIC 与外设一样，也占用了一个 I/O 地址（0xC0000000）。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。

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` 函数表明该 hart 对于哪些外设中断感兴趣。

```
void plicinithart(void) {      You, 6天前 • first commit
  int hart = cpuid();

  // set enable bits for this hart's S-mode
  // for the uart and virtio disk.
  *(uint32 *)PLIC_SENABLE(hart) = (1 << UART0_IRQ) | (1 << VIRTIO0_IRQ);

  // set this hart's S-mode priority threshold to 0.
  *(uint32 *)PLIC_SPRIORITY(hart) = 0;
}
```

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，目前仍处于初始化的阶段，我们还不希望被打断，因此处于关中断的状态，之后会调用 `intr_on` 来开中断，使得 CPU 能接收到中断。

外部中断与驱动程序

XV6 通过串口 UART 与外界通信，也就是说你和 XV6 的交互，输入命令到终端，XV6 输出运行结果让你看到，都是通过 UART 完成的。这里以 UART 为例介绍操作系统如何处理外部中断，为其编写驱动程序。

通常来说，管理设备的代码称为驱动，所有的驱动都在内核中。UART 设备的驱动代码在 `uart.c` 文件中。如果我们查看代码的结构，我们可以发现大部分驱动都分为两个部分（Linux kernel 也是如此），top / bottom 两部分。

- top 部分是用户进程或者内核的其他部分调用的接口，其运行在某进程的上下文中，从内核或用户空间中获取数据和命令，并将它们置于与下半部分共享的数据结构中。在 top 部分，中断被禁用。这部分代码仅包含关键代码。此部分代码的执行时间应尽可能短。
- bottom 部分通常是中断处理程序，其不运行在任何特定进程的上下文中，它只是处理中断。bottom 部分来完成中断事件的绝大多数任务。bottom 部分是一种中断处理程序，因此其**不应该阻塞**，也**不应该访问任何用户空间的数据**。

top 部分

首先，要说明几个概念：

- Shell
  - 命令行解释器，是一个用户程序，可以用来启动其他程序，或者运行一些脚本。
- Console
  - 从历史上看，Console 是一个单一的键盘和监视器，插入计算机上的专用串行控制台端口，用于与操作系统进行通信。
  - 在 XV6 上，Console 指你和 XV6 交互的那个命令行窗口。
- UART
  - UART 通常用在与其他通信接口的连接上。具体实物表现为独立的模块化芯片，或是微处理器中的内部周边设备。一般和 RS-232 规格的芯片进行搭配，作为连接外部设备的接口。
  - XV6 使用都是 RS232 协议，UART 是一个真实的芯片，有它自己的控制寄存器等。

如何从 Shell 程序输出提示符“$ ”到 Console。首先我们看 `user/init.c` ，这是系统启动后运行的第一个用户进程。

```c
int main(void) {
  int pid, wpid;

  if (open("console", O_RDWR) < 0) {
    mknod("console", CONSOLE, 0);      You, 6天前 • first
    open("console", O_RDWR);
  }
  dup(0); // stdout
  dup(0); // stderr

  for (;;) {
    printf("init: starting sh\n");
    pid = fork();
    if (pid < 0) {
      printf("init: fork failed\n");
      exit(1);
    }
    if (pid == 0) {
      exec("sh", argv);
      printf("init: exec sh failed\n");
      exit(1);
    }

    for (;;) {
      // this call to wait() returns if the shell exits,
      // or if a parentless process exits.
      wpid = wait((int *)0);
      if (wpid == pid) {
        // the shell exited; restart it.
        break;
      } else if (wpid < 0) {
        printf("init: wait returned an error\n");
        exit(1);
      } else {
        // it was a parentless process; do nothing.
      }
    }
```

首先这个进程的 main 函数创建了一个代表 Console 的设备。这里通过 mknod 操作创建了 console 设备。

因为这是第一个打开的文件，所以这里的文件描述符 fd 为 0。之后通过 dup 创建 stdout 和 stderr。这里实际上通过复制 fd 0，得到了另外两个文件描述符 1，2。

最终，fd 0，1，2 都用来代表 Console。

在 Unix 系统上，默认情况：

- fd 0，代表 stdin （标准输入）
- fd 1，代表 stdout （标准输出）
- fd 2，代表 stderr （标准错误输出）

在 XV6 上，这三者都连接到 Console。

向 Conlose 写字符，即调用 `write` 系统调用向 fd 1 写入内容。

在 `write` 系统调用中会判断文件描述符的类型。`mknod` 生成的文件描述符属于设备（FD_DEVICE），而对于设备类型的文件描述符，我们会为这个特定的设备执行设备相应的 write 函数。因为我们现在的设备是 Console，所以我们知道这里会调用 `console.c` 中的 `consolewrite` 函数。

```c
int consolewrite(int user_src, uint64 src, int n) {      You, 6天前 •
  int i;

  for (i = 0; i < n; i++) {
    char c;
    if (either_copyin(dst: &c, user_src, src: src + i, len: 1) == -1)
      break;
    uartputc(c);
  }

  return i;
}
```

这里先通过 `either_copyin` 将字符从用户空间拷入到内核空间，之后调用 `uartputc` 函数。`uartputc` 函数将字符写入给 UART 设备，所以你可以认为 `consolewrite` 是一个 UART 驱动的 top 部分。

```c
// 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) {      You, 6天前 • first commit …
  acquire(&uart_tx_lock);

  if (panicked) {
    for (;;)
      ;
  }
  while (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);
  }
  uart_tx_buf[uart_tx_w % UART_TX_BUF_SIZE] = c;
  uart_tx_w += 1;
  uartstart();
  release(&uart_tx_lock);
}
```

在 UART 的内部会有一个 buffer 用来发送数据，buffer 的大小是 32 个字符。同时还有一个为 consumer 提供的读指针和为 producer 提供的写指针，来构建一个环形队列。

在我们的例子中，Shell 是 producer，所以需要调用 `uartputc` 函数。

在函数中第一件事情是判断环形 buffer 是否已经满了。如果读写指针相同，那么 buffer 是空的，如果写指针加 1 等于读指针，那么 buffer 满了。当 buffer 是满的时候，向其写入数据是没有意义的，所以这里会 sleep 一段时间，将 CPU 出让给其他进程。

最终，字符会被送到 buffer 中，更新写指针，之后再调用 `uartstart` 函数。

```
// 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() {      You, 上周 • first commit
  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);
  }
}
```

`uartstart` 就是通知设备执行操作。qemu 模拟的是 16550 UART 芯片，其 spec 见以下链接，
http://byterunner.com/16550.html

> 简单叙述一下 16550 的工作工程：16550 有一个指示当前状态的寄存器 LSR（Line Status Register），通过读取这个寄存器，可以知道 16550 目前的状态：准备好发送数据 CPU 把数据送来吧（transmit holding empty），或者是已经接收到了数据 CPU 可以取走了（receive data ready）等等。
> 当处在 transmit holding empty 状态时，向 THR（Transmit Holding Register） 写一个字符，16550 就会发送出去。
> 当处在 receive data ready 状态时，就可以从 RHR（Receive Holding Register）读取接收到的字符。
> 值得注意的是，THR 和 RHR 是其实同一个寄存器，因状态而区分用途。

首先是检查当前设备是否空闲，如果空闲的话，我们会从 buffer 中读出数据，然后将数据写入到 THR（Transmission Holding Register）寄存器。这里相当于告诉设备，我这里有一个字节需要你来发送。

一旦数据送到了设备的寄存器，系统调用会返回，用户应用程序 Shell 就可以继续执行。

与此同时，UART 设备会将数据送出。在某个时间点，我们会收到中断，因为我们之前设置了要处理 UART 设备中断。

接下来我们看一下，当发生中断时，实际会发生什么。

bottom 部分

Bottom 部分即中断处理部分。

在我们向 Console 输出字符时，如果发生了中断，RISC-V 会做什么操作？我们之前已经在 SSTATUS 寄存器中打开了中断，所以处理器会被中断。假设键盘生成了一个中断并且发向了 PLIC，PLIC 会将中断路由给一个特定的 CPU 核，并且如果这个 CPU 核设置了 SIE 寄存器的 E bit（注，针对外部中断的 bit 位），那么会发生以下事情：

- 首先，会清除 SIE 寄存器相应的 bit，这样可以阻止 CPU 核被其他中断打扰，该 CPU 核可以专心处理当前中断。处理完成之后，可以再次恢复 SIE 寄存器相应的 bit。
- 之后，会设置 SEPC 寄存器为当前的程序计数器。我们假设 Shell 正在用户空间运行，突然来了一个中断，那么当前 Shell 的程序计数器会被保存。
- 之后，要保存当前的 mode。在我们的例子里面，因为当前运行的是 Shell 程序，所以会记录 user mode。
- 再将 mode 设置为 Supervisor mode。

- 最后将程序计数器的值设置成 STVEC 的值。（STVEC 用来保存 trap 处理程序的地址）在 XV6 中，STVEC 保存的要么是 uservec 或者 kernelvec 函数的地址，具体取决于发生中断时程序运行是在用户空间还是内核空间。
  - 在我们的例子中，Shell 运行在用户空间，所以 STVEC 保存的是 uservec 函数的地址。而从之前的课程我们可以知道 uservec 函数会调用 usertrap 函数。所以最终，我们在 usertrap 函数中。

在 `usertrap` 中会调用 `devintr` 函数来判断中断来源

```
    syscall();
  } else if ((which_dev = devintr()) != 0) {
    // ok      You, 上周 • first commit
  } else {
    printfk("usertrap(): unexpected scause %p pid=%d
    printfk("            sepc=%p stval=%p\n", r_sepc
    setkilled(p);
  }
```

在 `trap.c` 的 `devintr` 函数中，首先会通过 `SCAUSE` 寄存器判断当前中断是否是来自于外设的中断。如果是的话，再调用 `plic_claim` 函数来获取中断。

```
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) {
      printfk("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) {
```

`plic_claim` 函数位于 `plic.c` 文件中。在这个函数中，当前 CPU 核会告知 PLIC，自己要处理中断，`PLIC_SCLAIM` 会将中断号返回，对于 UART 来说，返回的中断号是 10。

```
// ask the PLIC what interrupt we should serve.
int plic_claim(void) {      You, 上周 • first com
  int hart = cpuid();
  int irq = *(uint32 *)PLIC_SCLAIM(hart);
  return irq;
}
```

从 `devintr` 函数可以看出，如果是 UART 中断，那么会调用位于 `uart.c` 文件的 `uartintr` 函数。

```c
// handle a uart interrupt, raised because input has
// arrived, or the uart is ready for more output, or
// both. called from devintr().
void uartintr(void) {      You, 上周 • first commit …
  // 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);
}
```

该函数首先会调用 `uartgetc` 从 UART 的接受寄存器中读取数据，从 UART 读数据就和写数据类似，先读控制寄存器看 UART 芯片是否收到了数据，可以让 CPU 读走，若是，CPU 读 RHR 寄存器把数据取走。

```c
// read one input character from the UART.
// return -1 if none is waiting.
int uartgetc(void) {      You, 上周 • first
  if (ReadReg(LSR) & LSR_RX_READY) {
    // input data is ready.
    return ReadReg(RHR);
  } else {
    return -1;
  }
}
```

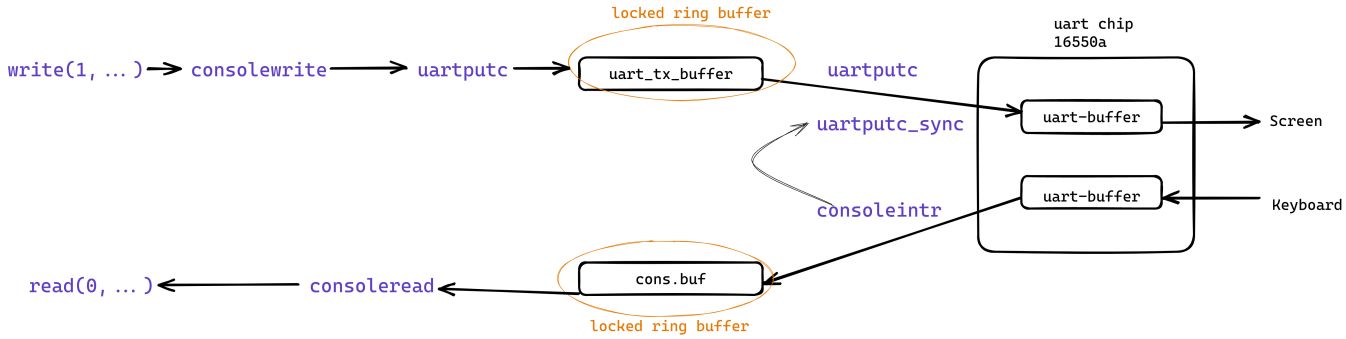
如果读到了数据，便将获取到的数据传递给 `consoleintr` 函数。

```
void consoleintr(int c) {       You, 上周 • first commit
  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_SIZE] != '\n') {
      cons.e--;
      consputc(c: BACKSPACE);
    }
    break;
  case C('H'): // Backspace
  case '\x7f': // Delete key
    if (cons.e != cons.w) {
      cons.e--;
      consputc(c: BACKSPACE);
    }
    break;
  default:
    if (c != 0 && cons.e - cons.r < INPUT_BUF_SIZE) {
      c = (c == '\r') ? '\n' : c;

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

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

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

这个函数很长，但主要做了一下几件事：

- 判读是否为控制字符，比如 `ctrl+p` 等等，从而做出特殊操作
- 判读是否为特殊字符，比如退格（删除）
  - 当你按下删除键的时候，屏幕上少了一个一个字符，它实际做了很多事
  -
```
void consputc(int c) {
  if (c == BACKSPACE) {
    // if the user typed b
    uartputc_sync('\b');
    uartputc_sync(' ');
    uartputc_sync('\b');
```
  - 假如把屏幕比作一个打字机，它让喷头：向后倒退了一格，打印一个空格，然后再倒退一格。实际上的删除是用一个空格盖住了已有的字符，然后光标回到空格前的位置。
- 将读到的字符加入到 console 的 buffer （同样是一个环形队列，同样存在一个生产者-消费者模型），然后将字符回显（echo）
  - 你向终端输入一个字符，若是没有回显，你的屏幕上不会出现你刚输入的字符，你并不知道你是否输入成功了。
  - 通常的逻辑是：回显你输入的任何字符，然后把回车符作为运行命令的一个标志。

读字符的部分完成，开始写字符的部分，`uartintr` 会运行到 `uartstart` 函数，这个函数会将存储在 buffer 中的字符送出，和上文介绍的一致。



生产者消费者问题

> https://en.wikipedia.org/wiki/Producer%E2%80%93consumer_problem

UART 的读写过程存在一个十分典型的生产者消费者问题（Producer-consumer problem）。

生产者消费者问题，也称有限缓冲问题（Bounded-buffer problem），是一个多进程同步问题的经典案例。该问题描述了共享固定大小缓冲区的两个进程——即所谓的“生产者”和“消费者”——在实际运行时会发生的问题。生产者的主要作用是生成一定量的数据放到缓冲区中，然后重复此过程。与此同时，消费者也在缓冲区消耗这些数据。该问题的关键就是要保证生产者不会在缓冲区满时加入数据，消费者也不会在缓冲区中空时消耗数据。

解决这一问题通常使用队列（XV6 中用的是环形队列）。
其带来了两个优点：

- 平衡高速设备与低速设备。
  - 如果生产者生产数据的速度很快，而消费者消费数据的速度很慢，那么生产者就必须等待消费者消费完了数据才能够继续生产数据；同理如果消费者的速度大于生产者那么消费者就会经常处理等待状态。
  - 在上面的驱动程序中，`write` 调用写字符，前面只是在内存里拷来拷去相对很快，而后边用 UART 串口输出相对较慢。
- 多个消费者共用一个生产者，或多个生产者共用一个消费者。
  - 在上面的例子里，`uart_tx_buffer` 就是多生产者单消费者，因为所有进程都能向 UART 输出，而 UART 芯片只有一个。`cons.buf` 是多消费者单生产者，同理。

中断与轮询（Polling）与 DMA

当 Unix 刚被开发出来的时候，Interrupt 处理还是很快的。这使得硬件可以很简单，当外设有数据需要处理时，硬件可以中断 CPU 的执行，并让 CPU 处理硬件的数据。

如果你有一个高性能的设备，例如你有一个网卡，这个网卡收到了大量的小包，网卡每秒可以生成 1.5Mpps，这意味着每一个微秒，CPU 都需要处理一个中断，这就超过了 CPU 的处理能力。那么当网卡收到大量包，并且处理器不能处理这么多中断的时候该怎么办呢？

这里的解决方法就是使用 polling。除了依赖 Interrupt，CPU 可以一直读取外设的控制寄存器，来检查是否有数据。对于 UART 来说，我们可以一直读取 RHR 寄存器，来检查是否有数据。现在，CPU 不停的在轮询设备，直到设备有了数据。

这种方法浪费了 CPU cycles，当我们在使用 CPU 不停的检查寄存器的内容时，我们并没有用 CPU 来运行任何程序。

所以，对于一个慢设备，你肯定不想一直轮询它来得到数据。我们想要在没有数据的时候切换出来运行一些其他程序。但是如果是一个快设备，那么 Interrupt 的 overhead 也会很高，那么我们在 polling 设备的时候，是经常能拿到数据的，这样可以节省进出中断的代价。

所以对于一个高性能的网卡，如果有大量的包要传入，那么应该用 polling。对于一些精心设计的驱动，它们会在 polling 和 Interrupt 之间动态切换（注，也就是网卡的 NAPI）。

Polling 的问题是，CPU 需要一直读取外设的控制寄存器，这会占用 CPU 时间。CPU 无时不刻的在处理着大量的事务，但有些事情却没有那么重要，比方说数据的复制和存储数据，如果我们把这部分的 CPU 资源拿出来，让 CPU 去处理其他的复杂计算事务，是不是能够更好的利用 CPU 的资源呢？

DMA 就是基于以上设想设计的，它的作用就是解决大量数据转移过度消耗 CPU 资源的问题。有了 DMA 使 CPU 更专注于更加实用的操作、计算、控制等。

假设程序想拷贝 1GB 的数据到磁盘，相对于 CPU，磁盘很慢，如果我们想让程序以循环的方式一点一点把数据拷贝到磁盘，那么 CPU 的时间开销会非常巨大。

如果系统当中有另外一个比较 Tiny 的 CPU，它只负责执行 memory copy（从内存读一个字节到这个 Tiny CPU，再从这个 Tiny CPU 把刚刚读到的字节传入磁盘，反过来也需要做到，从内存到内存的数据拷贝也需要做到）。当计算机系统的某个 CPU 正在执行一个比较耗时且简单的操作的时候，可以把这个操作交给刚刚说的 Tiny CPU 来做，等到 Tiny CPU 把任务做完了再给最开始执行任务的 CPU 的发中断。

因此 DMA 的本质是通过在系统里增加一个简单的专用“CPU”来不断的 Polling ，不断的去搬运数据，等这一切的工作完成，向真正的 CPU 发一个中断，来告诉它，耗时的工作已经完成了，你来做一些简单的收尾工作。

这三者，彼此间并不是代替的关系，而是各有优缺点，互相配合，广泛的存在于现如今的计算机系统中。

定时器中断（时钟中断）

定时器中断（时钟中断）是一种计算机系统中的中断处理方式，通过设置一个定时器，当计时器计时结束时，就会发生中断。这种中断通常用于实现定期执行某些任务的功能，例如操作系统可以使用定时器中断来实现进程的调度。通过定期触发中断，在每个时间片内让各个任务交替运行，从而最大化地利用 CPU 资源，提高系统的性能和响应速度。

首先来看，XV6 对计时器中断的设置，其代码在 `start.c` 的 `timerinit()` 函数

```
void timerinit() {
  // each CPU has a separate source of timer interrupts.
  int id = r_mhartid();

  // ask the CLINT for a timer interrupt.
  // RISCV_ACLINT_DEFAULT_TIMEBASE_FREQ = 10000000
  // https://github.com/qemu/qemu/blob/e46e2628e9fcce39e7ae28ac8c24bcc643ac48eb/i
  int interval = 1000000; // cycles; about 1/10th second in qemu.
  // A timer interrupt becomes pending whenever `mtime` contains a value greater
  // than or equal to `mtimecmp`
  *(uint64 *)CLINT_MTIMECMP(id) = *(uint64 *)CLINT_MTIME + interval;

  // prepare information in scratch[] for timervec.
  // scratch[0..2] : space for timervec to save registers.
  // scratch[3] : address of CLINT MTIMECMP register.
  // scratch[4] : desired interval (in cycles) between timer interrupts.
  uint64 *scratch = &timer_scratch[id][0];
  scratch[3] = CLINT_MTIMECMP(id);
  scratch[4] = interval;
  w_mscratch(x: (uint64)scratch);

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

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

  // enable machine-mode timer interrupts.
  w_mie(x: r_mie() | MIE_MTIE);      You, 上周 • first commit
}
```

`mtime` 代表“machine time”，它是一个 64 位的计数器，记录了自系统启动以来经过的时间（单位为 CPU 时钟周期）。`mtime` 可以由软件读取，以便进行时间戳记录、性能分析和定时等操作。

`mtimecmp` 则代表“machine time compare”，它也是一个 64 位的寄存器，用于与 `mtime` 进行比较。当 `mtime` 的值大于或等于 `mtimecmp` 的值时，会产生中断，从而实现基于时间的调度和定时功能。`mtimecmp` 可以由软件编程设置，并且可以在硬件级别触发中断，使得处理器能够执行预定的任务。

根据我们使用的 qemu 的频率，我们把定时器的时间间隔设置为 0.1 秒。我们首先获取当前 `mtime` 的值，然后加上时间间隔，将结果写入到 `mtimecmp` 寄存器。当定时器中断发生时，跳转到 `timervec` 进行处理。

`timervec` 在 `kernelvec.s` 中，如下

```
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 # swap 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)  # a3 = *(CLINT_MTIMECMP(hart))
        add a3, a3, a2# a3 += interval
        sd a3, 0(a1)  # *(CLINT_MTIMECMP(hart)) = a3

        # arrange for a supervisor software interrupt
        # after this handler returns.
        li a1, 2
        csrw sip, a1

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

        mret
```

在这个函数里，大致做了如下几件事：

- 更新 `mtimecmp` 寄存器，以实现下一次的定时器中断
- 设置 `sip` 寄存器的 `SSIP` 位，实质上就是设置了一次 S-mode 的软中断

之后在 S-mode 下接收到中断或异常，会在 `devintr()` 中进行判断，处理如下

```
  } else if (scause == 0x8000000000000001L) {      You, 上周 •
    // 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(x: r_sip() & ~2);

    return 2;
```

通过判断 `scause` 寄存器，来判断是否为一次软中断。若是，说明这是一次定时器中断，一次时间片用完，要进行进程调度了。

> scause 寄存器的 spec 见链接， https://five-embeddev.com/riscv-isa-manual/latest/supervisor.html#sec:scause

一次定时器中断的处理流程大致如此，接下来简单总结下。

- 当在用户态时，时钟中断的过程
  - 当发生时钟中断时，先跳到 M-mode 执行 `timervec` （在 start. c 的 `timerinit` 中设置）
  - 在 `timervec` 中，触发一个软中断，然后从 M-mode 返回 U-mode 。
  - 在 U-mode 下，刚设置的软中断触发，跳到 `usertrap` 执行，然后就是正常的 U-mode 到 S-Mode 的中断处理了，此时的时钟中断就和系统调用等平级了。
  - U-mode => M-mode => U-mode => S-mode => U-mode，过程如此

为什么定时器中断要在 M-mode 下处理?

你可能会疑惑，为什么 XV6 把所有的中断/异常都委托给 S-mode 进行处理，而唯独定时器中断是个例外，要在 M-mode 下进行处理?

诚然，RISC-V 的 specification 是支持将定时器中断也委托给 S-mode 的，但厂商对 RISC-V 的具体实现并不一定如此。我们使用的是 qemu ，其实现是将定时器中断“硬连线”给 M-mode，因此定时器中断只能在 M-mode 下处理。也就是说，定时器中断只能在 M-mode 下处理是实现的限制，并不是 RISC-V 文档的限制。

> https://groups.google.com/a/groups.riscv.org/g/sw-dev/c/-KQnN5EPHF4?pli=1

至于厂商为什么如此实现，我没有找到具体的说明，不过可以进行一下猜想：

- 处理定时器中断，必然要读 `mtime` ，写 `mtimecmp` 寄存器，而这俩寄存器从名字就可以看出是 `machine-mode read-write register` ，也就是说只能在 M-mode 下才能进行读写。
- 即使支持了委托定时器中断给 S-mode ，但处理的过程还是避免不了要跳到 M-mode 。
- 由此，厂商从实现上就不支持委托定时器中断给 S-mode，必须在 M-mode 下处理。

异常

前文中说过，RISC-V 统一使用 Trap 这一术词，表达：暂停当前执行流，从低权限提高到高权限，跳到特定位置（Trap handler）进行执行（如从用户态到内核态）。

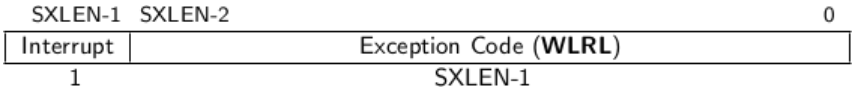
也就是说，不论是系统调用，亦或是中断，亦或是异常，统一在一位置进行分流，然后处理。
XV6 中在下图的 `usertrap` 对来自用户态的 trap 进行处理。

```c
void
usertrap(void)
{
  int which_dev = 0;

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

  struct proc *p = myproc();

  if(r_scause() == 8){      Robert Morris, 4年前 • fork/wait/exit work …
    // system call

    if(killed(p))
      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 sepc, scause, and sstatus,
    // so enable only now that we're 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());
    setkilled(p);
  }
```

图中的 `if .. else if ... else` 就是在对 Trap 进行分流，“分流”的依据就是 `scause` 寄存器。

> https://five-embeddev.com/riscv-isa-manual/latest/supervisor.html#sec:scause

RISC-V 对 `scause` 寄存器的定义如下：
`scause` 寄存器是一个 SXLEN 位读写寄存器，其格式如下图所示。当一个 Trap 被带入 S -mode 时，`scause` 被写入一个代码，表明引起该 Trap 的事件。



Interrupt	Exception Code	Description
1	0	*Reserved*
1	1	Supervisor software interrupt
1	2–4	*Reserved*
1	5	Supervisor timer interrupt
1	6–8	*Reserved*
1	9	Supervisor external interrupt
1	10–15	*Reserved*
1	≥16	*Designated for platform use*

Interrupt	Exception Code	Description
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned
0	7	Store/AMO access fault
0	8	**Environment call(ecall) from U-mode**(即系统调用)
0	9	Environment call(ecall) from S-mode
0	13	Load page fault

通过读取 `scause` 寄存器，就可以知道陷入 Trap 的原因：软中断，外部设备中断，指令异常，页表异常，系统调用等等...

Xv6 对异常的响应相当无趣: 如果用户空间中发生异常，内核将终止故障进程（上图中的 `setkilled(p)`）。

如果内核中发生异常，则内核会崩溃。

真正的操作系统通常以更有趣的方式做出反应。

例如，许多内核使用页面错误来实现写时拷贝版本的 `fork` —— *copy on write (COW) fork* 。

由页面错误驱动的 *COW fork* 可以使父级和子级安全地共享物理内存。
当 CPU 无法将虚拟地址转换为物理地址时，CPU 会生成页面错误异常。
RISC-V 有三种不同的页面错误:

- 加载页面错误 (当加载指令无法转换其虚拟地址时),
- 存储页面错误 (当存储指令无法转换其虚拟地址时)
- 指令页面错误 (当指令的地址无法转换时)

`scause` 寄存器中的值指示页面错误的类型，`stval` 寄存器包含无法翻译的地址。

COW fork 中的基本计划是让父子最初共享所有物理页面，但将它们映射为只读。因此，当子级或父级执行存储指令时，RISC-V CPU 引发页面错误异常。为了响应此异常，内核复制了包含错误地址的页面。它在子级的地址空间中映射一个权限为读/写的副本，在父级的地址空间中映射另一个权限为读/写的副本。更新页表后，内核会在导致故障的指令处恢复故障进程的执行。由于内核已经更新了相关的 PTE 以允许写入，所以错误指令现在将正确执行。

COW策略对`fork`很有效，因为通常子进程会在`fork`之后立即调用`exec`，用新的地址空间替换其地址空间。在这种常见情况下，子级只会触发很少的页面错误，内核可以避免拷贝父进程内存完整的副本。此外，*COW fork*是透明的: 无需对应用程序进行任何修改即可使其受益。

除 *COW fork* 以外，页表和页面错误的结合还开发出了广泛有趣的可能性。
另一个广泛使用的特性叫做惰性分配——*lazy allocation* 。它包括两部分内容：首先，当应用程序调用 `sbrk` 时，内核增加地址空间，但在页表中将新地址标记为无效。其次，对于包含于其中的地址的页面错误，内核分配物理内存并将其映射到页表中。由于应用程序通常要求比他们需要的更多的内存，惰性分配可以称得上一次胜利: 内核仅在应用程序实际使用它时才分配内存。
像 COW fork 一样，内核可以对应用程序透明地实现此功能。

利用页面故障的另一个广泛使用的功能是从磁盘分页。如果应用程序需要比可用物理 RAM 更多的内存，内核可以换出一些页面: 将它们写入存储设备 (如磁盘)，并将它们的 PTE 标记为无效。
如果应用程序读取或写入被换出的页面，则 CPU 将触发页面错误。然后内核可以检查故障地址。
如果该地址属于磁盘上的页面，则内核分配物理内存页面，将该页面从磁盘读取到该内存，将 PTE 更新为有效并引用该内存，然后恢复应用程序。
为了给页面腾出空间，内核可能需要换出另一个页面。
此功能不需要对应用程序进行更改，并且如果应用程序具有引用的地址 (即，它们在任何给定时间仅使用其内存的子集)，则该功能可以很好地工作。

结合分页和页面错误异常的其他功能包括自动扩展栈空间和内存映射文件。

实验

实验介绍

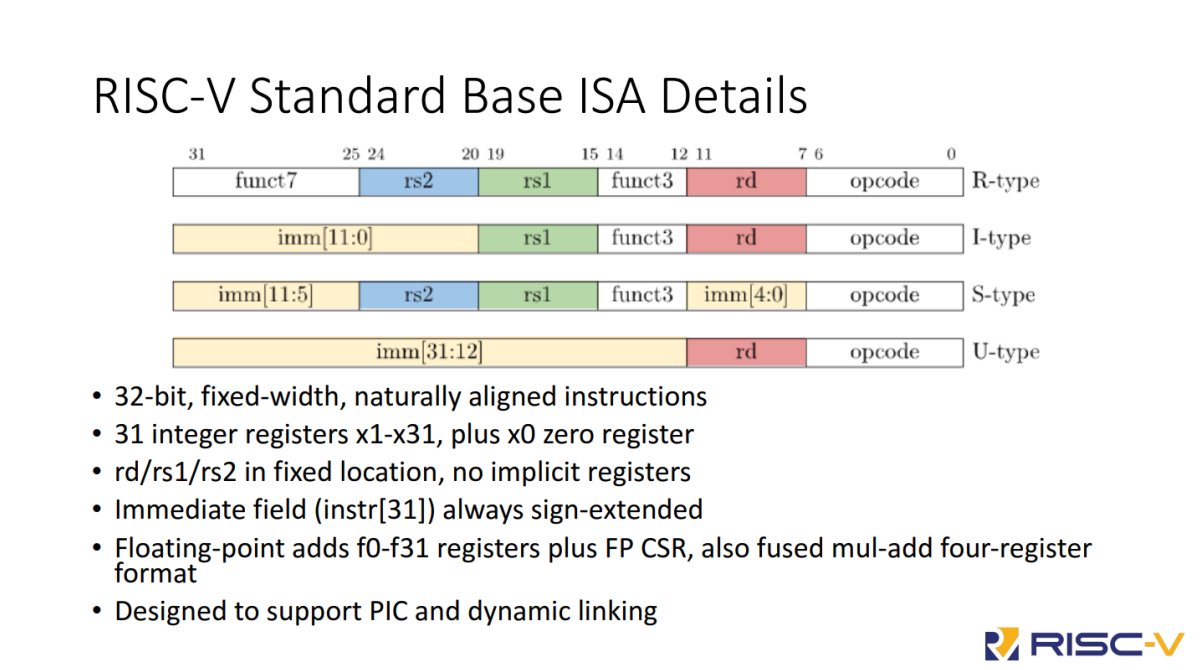
实验内容1 - 自定义异常处理

```shell
> git add .
> git commit -m 'message'    #暂存修改
> git switch lab5-1-exception    #切换到本实验的分支
```

- 任务：
  - 进一步完善 XV6 当前的异常处理
  - 当前 XV6 遇到进程异常就会将进场杀死，我们希望做的更有趣些：
    - 请修改 XV6，让其在遇到程序异常时，打印异常原因，然后跳到引发异常的下一条指令继续运行。

Tips：

- `epc` 寄存器保存着，从 trap 返回后继续运行的地址
- RISC-V 标准的基本指令集，指令长度定长为 32 bits

> https://www.linux-kvm.org/images/6/6a/02x04B-QEMU-Support_for_the_RISC-V_Instruction_Set_Architecture.pdf
>
> 
>

- `scause` 寄存器的说明 https://five-embeddev.com/riscv-isa-manual/latest/supervisor.html#sec:scause

预期输出：

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ test
PID 3 : Store/AMO page fault
PID 3 : Store/AMO page fault
PID 3 : Store/AMO page fault
PID 3 : Load page fault
PID 3 : Store/AMO page fault
PID 3 : Load page fault
PID 3 : Store/AMO page fault
PID 3 : Load page fault
PID 3 : Store/AMO page fault
$
```

实验内容2 - 极简版 GDB

SHELL

```
> git add .
> git commit -m 'message'   #暂存修改
> git switch lab5-2-minigdb   #切换到本实验的分支
```

要求：

- 请为 XV6 实现一个“极简版的 GDB”：
  - 用户程序会调用 `ebreak`，人工为程序添加端点
  - 请输出用户程序中 `ebreak` 前后的各 4 条指令的机器码

Tips：

- `ebreak` 添加一个端点，也是一个异常，可通过 `scause` 确定
- 使用 `copyin` 从用户地址空间，拷贝内存到内核空间

预期输出：

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ excep
PID 3 : Store/AMO page fault
PID 3 : Store/AMO page fault
PID 3 : Store/AMO page fault
PID 3 : Breakpoint at 0x0000000000000030
PID 3 : 0x0000000000000020 : f7 0 9 47 83 47 7 0 bb 87 f7 2 23 0 f7 0
PID 3 : 0x0000000000000030 : ebreak
PID 3 : 0x0000000000000034 : 83 47 7 0 bb 87 f7 2 23 0 f7 0 79 57 83 47
PID 3 : Store/AMO page fault
PID 3 : Load page fault
PID 3 : Store/AMO page fault
PID 3 : Load page fault
PID 3 : Store/AMO page fault
$
```

```
❯ objdump -s '/home/delta/workspace/xv6-labs-2022/user/excep.o'

/home/delta/workspace/xv6-labs-2022/user/excep.o:     文件格式 elf64-little

Contents of section .text:
 0000 411122e4 00080147 83470700 bb87f702  A."....G.G......
 0010 2300f700 05478347 0700bb87 f7022300  #....G.G......#.
 0020 f7000947 83470700 bb87f702 2300f700  ...G.G......#...
 0030 02907d57 83470700 bb87f702 2300f700  ..}W.G......#...
 0040 79578347 0700bb87 f7022300 f7007557  yW.G......#...uW
 0050 83470700 bb87f702 2300f700 01452264  .G......#....E"d
 0060 41018280                             A...
Contents of section .debug_info:
```

可以看到，XV6 正确的输出了程序 `ebreak` 前后几条指令的机器码。

> GDB 的实现就是向要调试的程序添加断点。当你为程序添加了断点后，GDB 就会将目标位置的指令保存起来，然后替换为 INT3 指令（x86 平台，类似于 RISC-V 平台的 `ebreak`）。当运行到端点位置时，就会触发异常，看起来程序就暂停了，你可以在 GDB 进行调试，若要继续运行，则先去保存起来的那条指令，然后再回到原程序运行。
> 操作系统一般会提供 `ptrace` 这种系统调用，以方便实现利用上述的工作。

拓展实验内容

> 上文 MinidGDB 仅仅打印了机器码，请对机器码进行反汇编，以打印成对应的汇编指令。

Tips:

- https://github.com/michaeljclark/riscv-disassembler
- https://jborza.com/emulation/2021/04/18/riscv-disassembler.html
- https://www.cs.sfu.ca/~ashriram/Courses/CS295/labs/Lab4/index.html