



# Lab4

## 实验报告

姓名 李晓畅

学号 20307130261

班级 计算机科学技术



# 实现思路

## 3.1 RISC-V ASSEMBLY

### 3.1 RISC-V assembly

首先需要了解一些 RISC-V 汇编语言。xv6 中有一个文件 *user/call.c*。编译后，它在 *user/call.asm* 中生成程序的可读汇编版本。

阅读 *call.asm* 中函数 g、f 和 main 的代码。RISC-V 的说明手册在：

<https://pdos.csail.mit.edu/6.S081/2022/reference.html>

- (1) 函数的参数包含在哪些寄存器中？例如在 main 对 printf 的调用中，哪个寄存器保存 13？

x10-11		a0-1	Function arguments/return values
x12-17		a2-7	Function arguments

RISC-V 为我们预留了 a0 到 a7 作为函数参数寄存器，这里涉及的函数只使用到了 a0 到 a2。

24: 4635	li a2,13
26: 45b1	li a1,12

具体在本例的 main 函数，不难看出 13 保存在 a2 寄存器中。

- (2) Main 的汇编代码中对函数 f 的调用在哪里？对 g 的调用在哪里？  
(Hint: 编译器可能内联函数)

1e: e406	sd ra,8(sp)
20: e022	sd s0,0(sp)
22: 0800	addi s0,sp,16
printf("%d %d\n", f(8)+1, 13);	
24: 4635	li a2,13
26: 45b1	li a1,12

可以看出本例中并没有显式地调用 f() 和 g()，而是直接算出了调用结果再加一的 12。但是可以看出，为了

准备函数调用，特意保存了返回地址寄存器的值到栈中。虽然最终没有什么用处。

- (3) 函数 printf 位于哪个地址？

30: 00000097	auipc ra,0x0
34: 61a080e7	jalr 1562(ra) # 64a <printf>

在  $0x30 + 0x0 + 1564 = 0x64a$  处

- (4) 在 jalr 到 main 中的 printf 之后，寄存器 ra 中存储的值是？

```
30: 00000097          auipc  ra,0x0
34: 61a080e7          jalr   1562(ra) # 64a <printf>
exit(0);
38: 4501              li     a0,0
```

传入时  $ra=0x30$ 。

C.JALR (jump and link register) performs the same operation as C.JR, but additionally writes the address of the instruction following the jump ( $pc+2$ ) to the link register,  $x1$ . C.JALR expands to  $jalr x1, rs1, 0$ .

因为调用的是  $jalr$ , 跳转并链接, 所以  $ra=ra+2*4=0x38$ , 也就是返回到紧接着的下一条指令地址。

- (5) 运行以下代码:

```
unsigned int i = 0x00646c72;
printf("H%ox Wo%os", 57616, &i);
```

输出是什么？注：<https://www.asciiitable.com/> 是一个将字节映射到字符的 ASCII 表。

输出取决于 RISC-V 是 little-endian 的。如果 RISC-V 是 big-endian, 怎样设置来产生相同的输出？是否需要更改  $i57616$  为不同的值？

参考链接：[http://www.webopedia.com/TERM/b/big\\_endian.html](http://www.webopedia.com/TERM/b/big_endian.html);

<https://www.rfc-editor.org/ien/ien137.txt>

HEX E110 输出 16 进制整数应该不会受到大小端影响，毕竟只要理解

DEC 57,616 一致进制转化应该是一样的，都是 e110，输出 He110。

对于输出字符串，应该会受到大小端影响。

如果是小段模式，那么倒着翻译就是“led\0”，正好是一个字符串,打印输出 Wold。

如果是大段，那么就是“\0del”，打印输出 Wo。

结果就是这样的呢。  
He110 World[Inferior 1 (process 134008) exited normally]

所以说如果是大端模式， $i$  就要倒过来，变成： $i=0x726c6400$ 。

- (6) 在下面的代码中，会打印出什么？（注意：答案不是特定值）为什么会发生这种情况？

```
• printf("x=%d y=%d", 3);
```

说真的，编译就给报错。`warning: format ‘%d’ expects a matching ‘int’ argument`。除非屏蔽报错。

在实际运行中第二个输出 y 的值应该会同执行这条语句时 a2 的值有关。

因为 `printf` 还是会取 a2 作为第二个参数，但因为没有为它赋值，所以到底是什么取决于它之前是什么样子的。

## 3.2 BACKTRACE

`backtrace` 对调试非常有用：它是 stack 中位于发生错误的点的上方的函数调用列表。编译器在每个 stack frame 中放入一个指向上一个 stack frame 的帧指针，即该指针保存 caller 的帧指针。寄存器 s0 保存当前 stack frame 的指针（实际上指向堆栈上保存的返回地址加上 8 的地址）。`backtrace` 应该使用这些帧指针向上遍历 stack 并在每个 stack frame 中打印保存的返回地址。

按照提示进行：

- 将 `backtrace` 的函数原型添加到 `kernel/defs.h` 以便可以在 `sys_sleep` 中
- 调用 `backtrace`

```
• void backtrace(void); 照例添加。
```

然后在 `sleep` 中也要添加。`backtrace();`

- GCC 编译器将当前执行的函数的帧指针存储在寄存器 s0 中。将以下函数添加到 `kernel/riscv.h`：

```
static inline uint64  
r_fp()  
{  
    uint64 x;  
    asm volatile("mv %0, s0" : "=r" (x));  
    return x;  
}
```

并在 `backtrace` 中调用此函数以读取当前帧指针。此函数使用内联汇编来读取 s0。

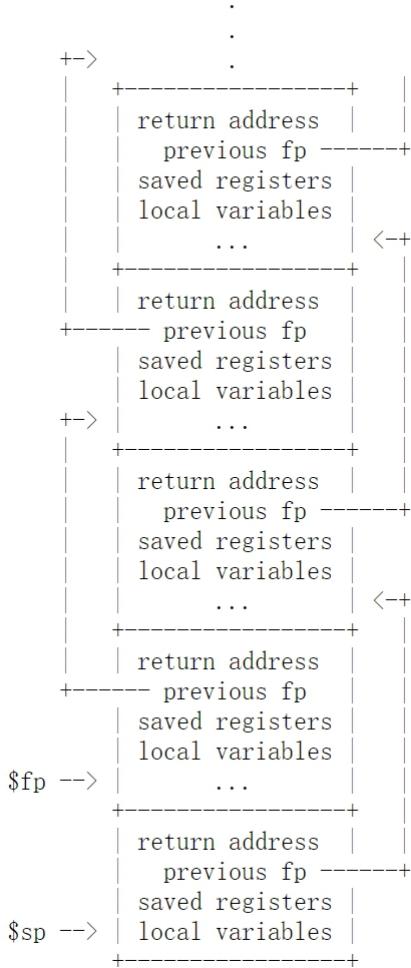
添加函数。

```
366 static inline uint64
367 r_fp()
368 {
369     uint64 x;
370     asm volatile("mv %0, $0" : "=r" (x));
371     return x;
372 }
373
```

- <https://pdos.csail.mit.edu/6.1810/2022/lec/l-riscv.txt> 有一张 stack frame 布局的图片。请注意，返回地址位于距当前 **stack frame** 的帧指针的固定偏移量 (-8) 处，而保存的帧指针位于距当前 **stack frame** 的帧指针的固定偏移量 (-16) 处。

他指的是这张指示栈结构的图：

Stack



另外还有一个提示：

- xv6 为 xv6 内核中的每个 stack 在 PAGE 地址对齐处分配一个页面。可以使用 PGROUNDDOWN(fp)和 PGROUNDDUP(fp)计算堆栈页面的顶部和底部地址。请参阅*kernel/riscv.h*。这些数字有助于 backtrace 终止其遍历。

那要怎么实现就很容易了。

测试结果截图：

```
$ bttest
backtrace:
0x00000000800021ac 0x000000008000201e
0x0000000080001d14 /home/wojtek/Desktop/xv6-lab4-2022/kernel/sysproc.c:71
0x000000008000201e /home/wojtek/Desktop/xv6-lab4-2022/kernel/syscall.c:141
0x0000000080001d14 /home/wojtek/Desktop/xv6-lab4-2022/kernel/trap.c:76
```

运行成功

### 3.3 ALARM

#### 3.3 Alarm

本练习中将向 xv6 中添加一个功能，该功能会在进程使用 CPU 时定期提醒它。这对于想要限制它们占用 CPU 时间的计算密集型进程，或者对于想要计算但又想要采取一些定期操作的进程可能很有用。更一般地说，你将实现一种原始形式的 user-level interrupt/fault handler；例如，你可以使用类似的东西来处理应用程序中的 page fault。

如果你的测试通过了 **alarmtest** 和‘**usertests -q**’，那么结果就是正确的。

你应该添加一个新的 **sigalarm(interval,handler)** 系统调用。如果一个应用程序调用了 **sigalarm(n,fn)**，那么在它消耗了 n 个 CPU "ticks" 之后（在 n 个时钟中断后），将执行 **handler** 函数。当 **handler** 通过调用 **sigreturn()** 返回时，应用程序应该从中断的地方继续。Tick 由硬件计时器产生中断的频率决定。如果应用程序调用 **sigalarm(0,0)**，则停止产生周期性调用。

xv6 存储库中有一个 **user/alarmtest.c** 文件。将其添加到 Makefile 中。只有添加了 **sigalarm** 和 **sigreturn** 系统调用才能正确编译。

**alarmtest** 在 **test0** 中调用 **sigalarm(2,periodic)** 以要求内核在 2 个时钟中断后调用一次 **periodic()**。可以在 **user/alarmtest.asm** 中看到 **alarmtest** 的汇编代码。

## TEST0: INVOKE HANDLER

按照提示进行：

- You'll need to modify the Makefile to cause `alarmtest.c` to be compiled as an xv6 user program.

对应添加  
191 | | \$U/\_alarmtest\|

- The right declarations to put in `user/user.h` are:

- int sigalarm(int ticks, void (\*handler)());  
int sigreturn(void);

26 | int sigalarm(int ticks, void (\*handler)());  
27 | int sigreturn(void);

- Update `user/usys.pl` (which generates `user/usys.S`, `kernel/syscall.h`, and `kernel/syscall.c` to allow `alarmtest` to invoke the `sigalarm` and `sigreturn` system calls.

对应添加即可

```
40 | entry("sigalarm");  
41 | entry("sigreturn");  
24 | #define SYS_sigalarm 22  
25 | #define SYS_sigreturn 23  
extern uint64 sys_sigalarm(void);  
extern uint64 sys_sigreturn(void);  
  
131 | [SYS_sigalarm] sys_sigalarm,  
132 | [SYS_sigreturn] sys_sigreturn,
```

Your `sys_sigalarm()` should store the alarm interval and the pointer to the handler function in new fields in the `proc` structure (in `kernel/proc.h`).  
You'll need to keep track of how many ticks have passed since the last call (or are left until the next call) to a process's alarm handler; you'll need a new field in `struct proc` for this too. You can initialize `proc` fields in `allocproc()` in `proc.c`.

在 `proc` 中添加：

```
108 | int interval;  
109 | uint64 handler;  
110 | int passed;
```

对应给出就行。

对应要分配和释放：

```
154 | p->interval = 0; 184 | p->interval = 0;  
155 | p->handler = 0; 185 | p->handler = 0;  
156 | p->passed = 0; 186 | p->passed = 0;
```

```
Every tick, the hardware clock forces an interrupt, which is handled in usertrap() in kernel/trap.c.  
You only want to manipulate a process's alarm ticks if there's a timer interrupt; you want something like  
if(which_dev == 2) ...  
Only invoke the alarm function if the process has a timer outstanding. Note that the address of the user's alarm function might be 0 (e.g., in user/alarmtest.asm, periodic is at address 0).  
You'll need to modify usertrap() so that when a process's alarm interval expires, the user process executes the handler function. When a trap on the RISC-V returns to user space,  
● what determines the instruction address at which user-space code resumes execution?
```

大概可以这样写：

```
79  if (which_dev == 2 && p->interval>0){  
80      p->passed++;  
81  
82      if (p->passed >= p->interval){  
83          p->passed =0;  
84          // void (*f) (void) = (void (*)(void)) p->handler;  
85          // (*f)();  
86          p->trapframe->epc = p->handler;  
87      }  
88  }
```

时钟 tick 一下就检查，如果要提醒用户的话就增加经过的 tick 数。

修改 **trapframe** 保存的程序计数器的值为 **handler** 函数地址，这样结束后会接着执行 **handler**。

执行完 **handler** 我们也不考虑还原，可能有一些问题。再说吧。

注释掉的部分有一些问题，是不对的。之后在问题环节细说。

- For now, your sys\_sigreturn should just return zero.

函数大概可以这样实现。**Sigalaem** 就初始化参数就可以了。**Sigreturn** 先直接返回 0。

```

97  uint64
98  sys_sigalarm(void){
99      int interval;
100     uint64 handler;
101     struct proc* p=myproc()
102         argint(0,&interval);
103         argaddr(1, &handler);
104
105     p->interval = interval;
106     p->handler = handler;
107
108     return 0;
109 }
110
111 uint64
112 sys_sigreturn(void){
113     return 0;
114 }
```

尝试运行：

```

$ alarmtest
test0 start
.....alarm!
test0 passed
test1 start
.alarm!
```

至少 test0 和我们预想的一样。其他当然是过不了的。因为没有恢复到之前状态。

### TEST1/TEST20/TEST30: RESUME INTERRUPTED CODE

按照提示进行：

当计时器到期时，让 usertrap 在 struct proc 中保存足够的状态，以便 sigreturn 可以正确返回到被中断的用户代码。

- 防止对处理程序的重入调用——如果处理程序尚未返回，内核不应再次调用它。

这要求我们在 *proc* 里面添加几个量。

```

112 |     int handling;
113 |     struct trapframe *fnframe;
```

一个正在处理，防止重复执行。Xv6 没给 bool 型，用一个整型。

Your solution will require you to save and restore registers---what registers do you need to save and restore to resume the interrupted code correctly? (Hint: it will be many).

一个临时 *trapframe*, 临时保存 *trapframe*, 以备之后恢复。虽然我觉得我只修改了 *epc*, 但是我们要考虑到或许 *handler* 函数会用到很多预料之外的东西, 所以全部保存应该是更好的。所以没必要单独保存寄存器。

然后对应的创建删除也要修改。

```
if((p->fnframe = (struct trapframe *)kalloc()) == 0){  
    freeproc(p);  
    release(&p->lock);  
    return 0;  
}  
  
if(p->fnframe)  
    kfree((void*)p->fnframe);
```

调用时要保存, 返回时要恢复。

调用要检查是否在处理, 需要处理要防止重复处理。因为这就一循环, 不会并发, 没必要考虑用更高级的东西。

```
if (which_dev == 2 && p->interval>0){  
    p->passed++;  
  
    if (p->handling == 0 && p->passed >= p->interval){  
        p->handling = 1;  
        p->passed =0;  
        // void (*f) (void) = (void (*)(void)) p->handler;  
        // (*f)();  
        *p->fnframe = *p->trapframe;  
        p->trapframe->epc = p->handler;  
    }  
}
```

- Make sure to restore a0. `sigreturn` is a system call, and its return value is stored in a0.

```

97 |     uint64
98 |     sys_sigalarm(void){
99 |         int interval;
100 |         uint64 handler;
101 |         struct proc* p=myproc();
102 |         argint(0,&interval);
103 |         argaddr(1, &handler);
104 |
105 |         p->interval = interval;
106 |         p->handler = handler;
107 |         p->handling = 0;
108 |
109 |         return 0;
110 |     }
111 |
112 |     uint64
113 |     sys_sigreturn(void){
114 |         struct proc* p= myproc();
115 |         *p->trapframe = *p-> fnframe;
116 |         p->handling = 0;
117 |         return p->trapframe->a0;
118 |     }

```

最后修改一下 *sigalarm* 和 *sigreturn*.

返回 *a0* 寄存器的值应该就能恢复 *a0* 了。

测试截图：

<pre> \$ alarmtest test0 start ...alarm! test0 passed test1 start .alarm! test1 passed test2 start ....alarm! test2 passed test3 start test3 passed </pre>	<pre> OK test sbrkarg: OK test validateitest: OK test bsstest: OK test bigargtest: OK test argptest: OK test stacktest: usertrap()                      sepc=0x0000000 OK test textwrite: usertrap()                      sepc=0x0000000 OK test pbug: OK test sbrkbugs: usertrap():                      sepc=0x0000000 usertrap(): unexpected sca                      sepc=0x0000000 OK test sbrklast: OK test sbrk8000: OK test badarg: OK ALL TESTS PASSED </pre>
--	--

测试通过。

## 问题和解决

### 调试和检查

在 test1 修改后报了这样的错。

```
$ alarmtest
test0 start
...alarm!
.....usertrap(): unexpected scause 0x000000000000000c pid=3
    sepc=0x000000000014f50 stval=0x000000000014f50
```

利用添加输出语句、GDB 调试方法并加以分析，定位问题是 *trapframe* 恢复不正确。进一步发现是把拷贝结构体写成浅拷贝地址。

```
p->trapframe = p-> fnframe;
```

修改后即可。

```
*p->trapframe = *p-> fnframe;
```

其实不止可以赋值。调用 *memmove*, *memcpy* 这样的函数应该也可以实现对应的功能。

### 可以直接调用函数吗？

```
if (which_dev == 2 && p->interval>0){
    p->passed++;

    if (p->handling == 0 && p->passed >= p->interval){
        p->handling = 1;
        p->passed =0;
        // void (*f) (void) = (void (*)(void)) p->handler;
        // (*f)();
        *p->fnframe = *p->trapframe;
        p->trapframe->epc = p->handler;
    }
}
```

上面提到过，注释掉的代码有问题。

其原因在于，我们得到的一个函数指针是进程中的逻辑地址，但是在内核中地址上可不一定能找到对应的函数。

如果非得这么做的话，可以考虑给物理地址，像 *trampoline* 那样实现。

或者直接在 *sigreturn* 里面调用。

但是这里有一点问题在于即便这样做，也要准备好恢复寄存器。那还不如直接改 *epc* 就可以了。

## 实验总结

在本次实验中我们主要学习了 trap 的有关内容。

用户空间和内核空间的切换通常被称为 trap。当程序执行系统调用、出现例如 page fault 的故障、外设触发中断时，都会 trap 进内核中。trap 涉及了许多小心的设计和重要的细节，这些细节对于实现安全、隔离和高性能来说非常重要。

值得注意的一点是硬件状态的转换。用户态下可能遇到 trap，但只有在内核态下我们才能处理 trap。所以硬件状态的转化是很重要的。

实际要处理 trap 需要一系列操作。

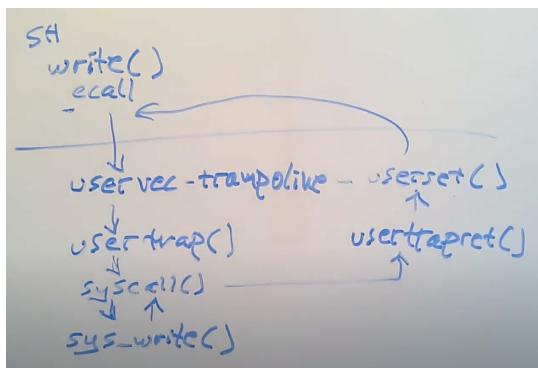
- 首先，我们需要保存 32 个用户寄存器。因为很显然我们需要恢复用户应用程序的执行，尤其是当用户程序随机的被设备中断所打断时。我们希望内核能够响应中断，之后在用户程序完全无感知的情况下再恢复用户代码的执行。所以这意味着 32 个用户寄存器不能被内核弄乱。但是这些寄存器又要被内核代码所使用，所以在 trap 之前，你必须先在某处保存这 32 个用户寄存器。
- 程序计数器也需要在某个地方保存，它几乎跟一个用户寄存器的地位是一样的，我们需要能够在用户程序运行中断的位置继续执行用户程序。
- 我们需要将 mode 改成 supervisor mode，因为我们想要使用内核中的各种各样的特权指令。
- SATP 寄存器现在正指向 user page table，而 user page table 只包含了用户程序所需要的内存映射和一两个其他的映射，它并没有包含整个内核

数据的内存映射。所以在运行内核代码之前，我们需要将 SATP 指向 kernel page table。

- 我们需要将堆栈寄存器指向位于内核的一个地址，因为我们需要一个堆栈来调用内核的 C 函数。
- 一旦我们设置好了，并且所有的硬件状态都适合在内核中使用，我们需要跳入内核的 C 代码。

而在内核中主要包括了 *uservec*, *usertrap*, *usertrapret* 和 *userret* 四个函数。

所以总的来说就是这样的流程：



(参考 [Lec06 Isolation & system call entry/exit \(Robert\) - MIT6.S081 \(gitbook.io\)](#))