**实验四 traps**

**4.1 实验目的**

1. 探索如何使用陷阱

2. 了解一定RISC-V汇编知识

3. 理解xv6中的堆栈

4. 进行用户级陷阱处理的编写

**4.2 实验内容**

4.2.1 RISC-V assembly（easy）

了解RISC-V组件，阅读call.asm中函数g、f和main的代码并回答相应问题。

4.2.2 Backtrace（moderate）

在kernel/printf.c中实现backtrace（）函数。在sys\_sleep中插入对该函数的调用，然后运行bttest，它调用sys\_sleep。您的输出应如下所示：

backtrace:

0x0000000080002cda

0x0000000080002bb6

0x0000000080002898

测试结束后退出qemu。在终端中：地址可能略有不同，但如果运行addr2line-e kernel/kernel（或riscv64-unknown-elf-addr2line-e kernel/kernel）并按如下方式剪切和粘贴上述地址：

$ **addr2line -e kernel/kernel**

0x0000000080002de2

0x0000000080002f4a

0x0000000080002bfc

Ctrl-D

您应该看到如下内容：

kernel/sysproc.c:74

kernel/syscall.c:224

kernel/trap.c:85

4.2.3 Alarm（hard）

在本练习中，您将向xv6添加一个特性，该特性在进程使用CPU时间时定期向其发出警报。这对于希望限制占用多少CPU时间的受计算限制的进程，或者对于希望进行计算但又希望执行某些周期性操作的进程，可能很有用。更一般地说，您将实现用户级中断/故障处理程序的原始形式；例如，您可以使用类似的方法来处理应用程序中的页面错误。如果您的解决方案通过alarmtest和usertests，那么它就是正确的。

4.2.4 submit

提交实验。

**4.3 实验过程**

4.3.1 RISC-V assembly

（1）首先执行

$ **make fs.img**



用于生成call.asm文件。

（2）Which registers contain arguments to functions? For example, which register holds 13 in main's call to printf?

答：

printf("%d %d\n", f(8)+1, 13);

24: 4635 li a2,13

26: 45b1 li a1,12

28: 00000517 auipc a0,0x0

2c: 7b050513 addi a0,a0,1968 # 7d8

根据call.asm中的代码可以看出存储13的为a2寄存器。

（3）Where is the call to function f in the assembly code for main? Where is the call to g? (Hint: the compiler may inline functions.)

答：

printf("%d %d\n", f(8)+1, 13);

24: 4635 li a2,13

26: 45b1 li a1,12

28: 00000517 auipc a0,0x0

2c: 7b050513 addi a0,a0,1968 # 7d8

同样从这段代码中可以看出printf()函数对于f函数的调用直接获得了12这一结果，所以在这里对f函数进行了内联优化处理。

int f(int x) {

e: 1141 addi sp,sp,-16

10: e422 sd s0,8(sp)

12: 0800 addi s0,sp,16

return g(x);

}

14: 250d addiw a0,a0,3

16: 6422 ld s0,8(sp)

18: 0141 addi sp,sp,16

1a: 8082 ret

在函数f中调用了函数g，同样可以看出这里的g函数也进行了内联优化处理。

（4）At what address is the function printf located?

答：

<malloc+0xea>

30: 00000097 auipc ra,0x0

34: 600080e7 jalr 1536(ra) # 630

从这里可以看出跳转至了ra+1536的位置，也就是print f位于0x630的位置。

（5）What value is in the register ra just after the jalr to printf in main ?

答：ra存储的是printf函数的返回地址，0x38

（6）Run the following code.

unsigned int i = 0x00646c72;

printf("H%x Wo%s", 57616, &i);

What is the output? Here's an ASCII table that maps bytes to characters.

The output depends on that fact that the RISC-V is little-endian. If the RISC-V were instead big-endian what would you set i to in order to yield the same output? Would you need to change 57616 to a different value?

答：小端的输出结果为He110 World。当RISC-V为大端时为获得相同输出需要将i改为0x726c64。其中57616的值不需要改变。

（7）In the following code, what is going to be printed after 'y='? (note: the answer is not a specific value.) Why does this happen?

printf("x=%d y=%d", 3);

答：在printf函数中未写明y之后%d所对应的值，所以输出的答案不确定，取决于寄存器中存储的值。

4.3.2 backtrace

（1）在defs.h中添加函数声明以便之后调用。

void            backtrace(void);

（2）根据题目要求在riscv.h中添加r\_fp()函数。

static inline uint64

r\_fp()

{

  uint64 x;

  asm volatile("mv %0, s0" : "=r" (x) );

  return x;

}

（3）在printf.c中对backtrace函数进行编写。

void

backtrace(void)

{

  uint64 cur\_fp = r\_fp();

  while(cur\_fp != PGROUNDDOWN(cur\_fp))

  {

    printf("%p\n", \*(uint64 \*)(cur\_fp - 8));

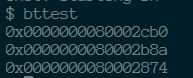
    cur\_fp = \*(uint64 \*)(cur\_fp - 16);

  }

}

即遍历一遍栈帧并打印返回地址，在xv6中只会给每个用户进程分配一个页（4KB）作为用户栈，所以可以通过检查fp是否到达页的开头来判断是否到达栈头。

（4）运行backtrace





4.3.3 alarm

（1）在user.h、usys.pl、syscall.h以及syscall.c中添加相应内容来调用sigalarm以及sigreturn

（2）首先在struct proc中添加相应的变量。

  //add traps

  int duration;                // ticks after last alarm

  int alarm;                   // alarm every n ticks

  uint64 handler;              // alarm函数

  struct trapframe \*alarm\_trapframe; // register saved for alarm

（3）实现sigalarm这个系统调用，该函数用于初始化新加的proc属性。

uint64

sys\_sigalarm(void)

{

  int ticks;

  uint64 handler;

  if(argint(0, &ticks) < 0)

    return -1;

  if(argaddr(1, &handler) < 0)

    return -1;

  struct proc\* p = myproc();

  p->alarm = ticks;

  p->handler = handler;

  p->duration = 0;

  p->alarm\_trapframe = 0;

  return 0;

}

（4）实现sys\_sigreturn函数，用于还原寄存器组。

uint64

sys\_sigreturn(void)

{

  struct proc\* p = myproc();

  if(p->alarm\_trapframe != 0){

    memmove(p->trapframe, p->alarm\_trapframe, 512);

    kfree(p->alarm\_trapframe);

    p->alarm\_trapframe = 0;

  }

  return 0;

}

（5）由于题目中提示每次tick产生都会产生一次中断，并且可以在trap.c中的usertrap里对中断进行处理，所以对trap.c中的usertrap函数进行修改。

  if(which\_dev == 2)

  {

    if(p->alarm != 0){

      p->duration++;

      if(p->duration == p->alarm){

        p->duration = 0;

        if(p->alarm\_trapframe == 0){

          p->alarm\_trapframe = kalloc();

          memmove(p->alarm\_trapframe, p->trapframe, 512);

          p->trapframe->epc = p->handler;

        }else{

          yield();

        }

      }else{

        yield();

      }

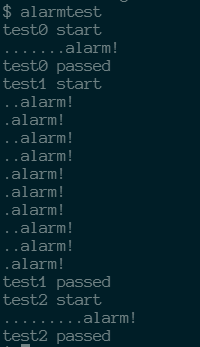
    }else{

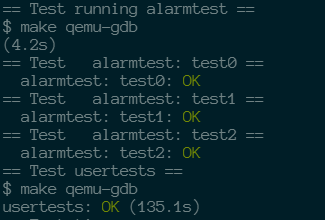
      yield();

    }

  }

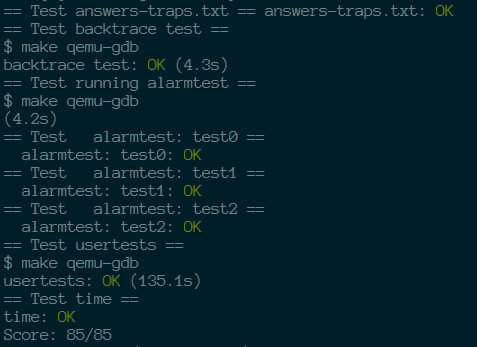
（6）运行alarm





**4.4 实验结果**

通过make grade对所有实验内容进行测试，结果如下图所示。



**4.5 实验小结**

本次实验引入了堆栈结构以及中断机制。

对于每一个进程的都有自己的堆栈对自身数据进行存储和使用，对于栈而言，存在两个重要的寄存器来确定栈的范围，一个是寄存栈底部指针的SP，另一个是寄存当前栈帧顶部的FP来确保不会产生读写错误。本次实现的backtrace就对进程的栈进行了遍历，让我更直观的看到栈是怎样的。

而本次实现的alarm就是中断现场的保存以及完成操作后现场的恢复，这让我对中断机制的执行过程更加了解了。