

OSLab4 Traps

邹怡21307130422

RISC-V

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

- a0~a7都可保存函数参数。a2寄存器保存13.

```
void main(void) {
1c: 1141          addi sp,sp,-16
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
28: 00000517      auipc a0,0x0
2c: 7c850513      addi a0,a0,1992 # 7f0 <malloc+0xe8>
30: 00000097      auipc ra,0x0
34: 61a080e7      jalr 1562(ra) # 64a <printf>
exit(0);
```

2. 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.)

- 在main函数中并没有显式调用f，f中也没有显式调用g，编译器使用内联函数，main函数中直接计算出f(8)+1的地址，省略显式调用f和g的过程 26: 45b1 li a1,12，在f中通过

```
14: 250d          addiw a0,a0,3
16: 6422          ld s0,8(sp)
```

计算g的地址

3. At what address is the function printf located?

- 0x64a 0x30+0x0+1562(0x61a)

4. What value is in the register ra just after the jalr to printf in main?

- ra的值是0x38，也就是返回地址

5. 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?

- 输出为HE110 World,如果是大端法要将i设置为0x726c6400,不需要改变57616的值, 因为E110是57616对应的十六进制数, 不会因为大端小端改变

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

- 取决于调用printf之前a2寄存器的值, 因为根据之前的汇编代码可以知道printf第二个参数是存储在a2寄存器中的, 经过gdb调试本次结果是1

```
42: e822      sd s0,16(sp)
43: 1000      addi s0,sp,32
44:          unsigned int i = 0x00646c72;
45: 006477b7   lui a5,0x647
46: c727879b   addiw a5,a5,-910
47: 2c: fef42623 sw a5,-20(s0)
48:          printf("H%x Wo%s", 57616, &i);
49: 30: fec40613 addi a2,s0,-20
50: 34: 65b9     lui a1,0xe
51: 36: 11058593 addi a1,a1,272 # e110 <base+0xd100>
52: 3a: 00000517 auipc a0,0x0
53: 3e: 7f650513 addi a0,a0,2038 # 830 <malloc+0xf0>
54: 42: 00000097 auipc ra,0x0
55: 46: 640080e7 jalr 1600(ra) # 682 <printf>
56:          printf("x=%d y=%d", 3);
57: 4a: 458d     li a1,3
58: 4c: 00000517 auipc a0,0x0
59: 50: 7f450513 addi a0,a0,2036 # 840 <malloc+0x100>
60: 54: 00000097 auipc ra,0x
61: 58: 62e080e7 jalr 1582(ra) # 682 <printf>
```

```
o- mmio-bus.0 -S -gdb tcp::26000
qemu-system-riscv64: QEMU: Terminated via GDB stub
indifference@LAPTOP-3TB8KCCP: ~/Desktop/xv6-l
abs-2022$ make qemu-gdb CPUs=1
*** Now run 'gdb' in another window.
qemu-system-riscv64 -machine virt -bios none
-kernel kernel/kernel -m 128M -smp 3 -nogra
phic -global virtio-mmio.force-legacy=false
-drive file=fs.img,if=none,format=raw,id=x0
-device virtio-blk-device,drive=x0,bus=virt
o-mmio-bus.0 -S -gdb tcp::26000

xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ call
HE110 World x=3 y=1
```

```
0x6a4 <printf+34> mv a1,a0
0x6a6 <printf+36> li a0,1
0x6a8 <printf+38> auipc ra,0x
0x6ac <printf+42> jalr -562(
> 0x6b0 <printf+46> ld ra,24
0x6b2 <printf+48> ld s0,16
0x6b4 <printf+50> addi sp,sp
0x6b6 <printf+52> ret
0x6b8 <free> addi sp,sp
0x6ba <free+2> sd s0,8(
0x6bc <free+4> addi s0,sp

In: printf L112 PC: 0x6b0
at user/printf.c:53
-- Type <RET> for more, q to quit, c to cont
inue without paging-- c
0x000000000000006b0 in printf (fmt=fmt@entry
=0x840 "x=%d y=%d") at user/printf.c:112
(gdb)
```

Backtrace

实验内容

- 编写函数backtrace()函数，递归读取栈帧，并输出函数返回的地址值
- 参考下面栈帧的结构，发现旧的fp与当前fp相差16，即previous fp=fp-16，返回地址则是当前fp-8，故输出时可以考虑通过迭代，只要fp没有超过当前页，就让其等于previous fp，通过不断迭代回溯函数并打印返回值



步骤

1. kernel/riscv.h：添加下面函数可获取s0寄存器保存的fp的值

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

2. kernel/defs.h中添加backtrace()的函数声明

```
// printf.c
void      printf(char*, ...);
void      panic(char*) __attribute__((noreturn));
void      printfinit(void);
void      backtrace(void); //new
```

3. kernel/sysproc.c: 调用backtrace()函数

```
51      uint64
52      sys_sleep(void)
53      {
54          int n;
55          uint ticks0;
56
57          backtrace(); //new
58
```

4. kernel/printf.c:

- 先通过r_fp()函数获取当前栈的fp值
- 在fp未到达页面顶之前，根据栈帧结构，打印ra，并将fp回退到上一个fp的地址，并更新pre_fp,其中PGROUNDDOWN(fp)表示当前fp所在页的最低地址，PGROUNDUP(fp)表示当前fp所在页的最高地址，如果fp在这两个地址之间，说明没有越界，故作为循环判断的条件

```
void
backtrace(void)
{
    printf("backtrace:\n");
    uint64 fp;
    fp=r_fp();
    uint64 pre_fp=((uint64*)(fp-16));
    uint64 top=PGROUNDUP(fp),bottom=PGROUNDDOWN(fp);
    while(fp>bottom&&fp<top)
    {
        uint64 ra=((uint64*)(fp-8));
        printf("%p\n",ra);
        fp=pre_fp;
        pre_fp=((uint64*)(fp-16));
    }
}
```

测试结果

运行xv6，调用bttest程序，那么就会打印栈帧中ra的值，这都是一些地址值。退出qemu，并使用addr2line程序，对地址进行转换就可以获得返回地址对应文件的行 发现是 backtrace() 函数的所有调用栈的返回地址(函数调用完后的下一代码)。

The screenshot shows a development environment with a code editor and a terminal. The code editor displays the `sys_sleep` function in `sysproc.c`, which calls `backtrace()`. The terminal shows the output of running `bttest` in the `xv6` kernel, followed by using `addr2line` to convert the printed addresses back to source code locations.

```

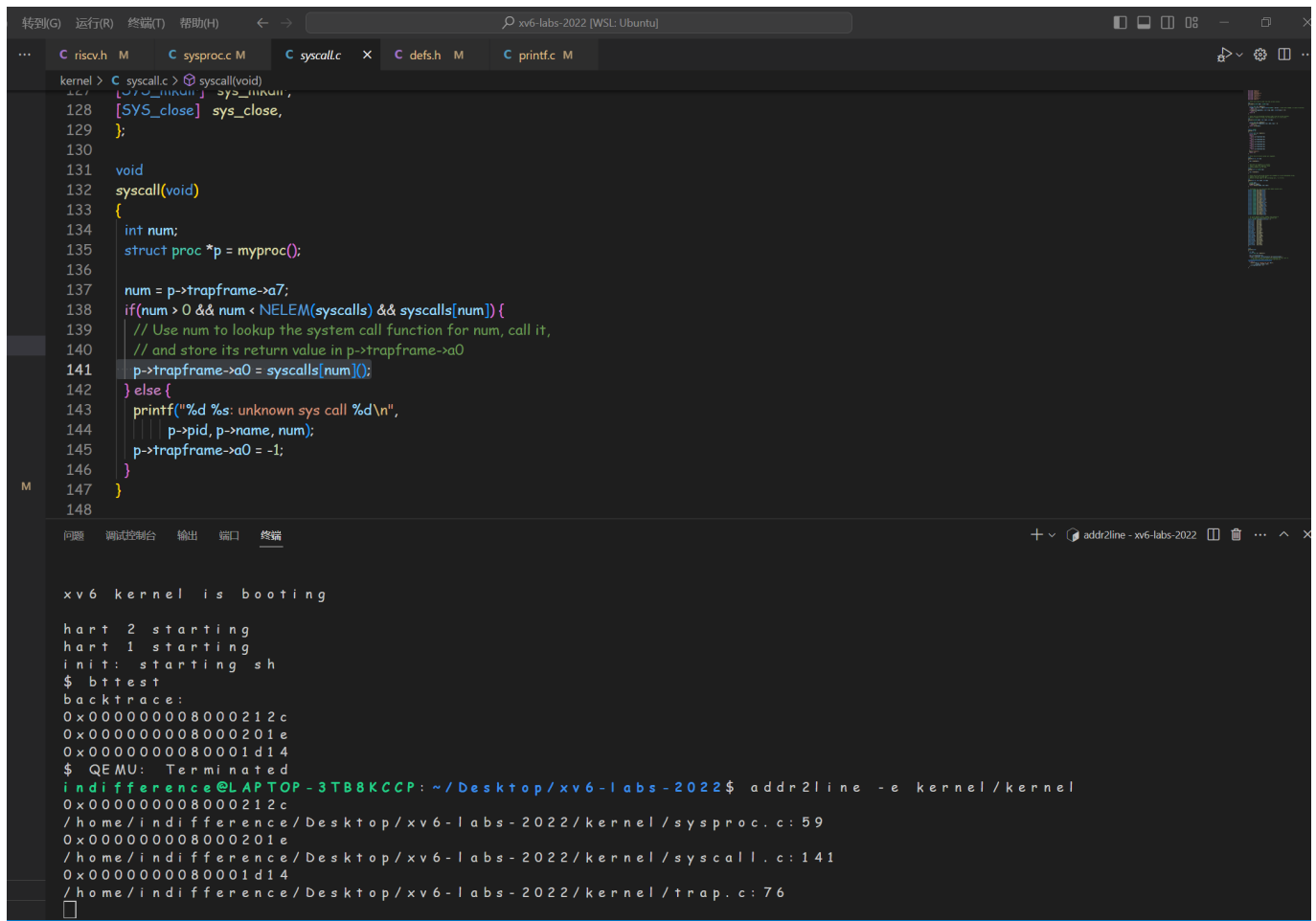
C riscv.h M  C sysproc.c M X  C defs.h M  C printf.c M
kernel > C sysproc.c > sys_sleep(void)
47     return -1;
48     return addr;
49 }
50
51 uint64
52 sys_sleep(void)
53 {
54     int n;
55     uint ticks0;
56
57     backtrace();//new
58
59     argint(0, &n);
60     if(n < 0)
61         n = 0;
62     acquire(&tickslock);
63     ticks0 = ticks;
64     while(ticks - ticks0 < n){
65         if(killed(myproc())){
66             release(&tickslock);
67             return -1;
68         }
69     }
70     return 0;
71 }

问题  调试控制台  输出  端口  终端
+ v  addr2li

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ bttest
backtrace:
0x000000008000212c
0x000000008000201e
0x0000000080001d14
$ QEMU: Terminated
indifference@LAPTOP-3TB8KCCP: ~/Desktop/xv6-labs-2022$ addr2line -e kernel/kernel
0x000000008000212c
/home/indifference/Desktop/xv6-labs-2022/kernel/sysproc.c:59
0x000000008000201e
/home/indifference/Desktop/xv6-labs-2022/kernel/syscall.c:141
0x0000000080001d14
/home/indifference/Desktop/xv6-labs-2022/kernel/trap.c:76

```



The image shows a code editor with several tabs: `riscv.h`, `sysproc.c`, `syscall.c`, `defs.h`, and `printf.c`. The `syscall.c` tab is active, displaying the following code:

```
kernel > C syscall.c > syscall(void)
127 [SYS_num] sys_num,
128 [SYS_close] sys_close,
129 };
130
131 void
132 syscall(void)
133 {
134     int num;
135     struct proc *p = myproc();
136
137     num = p->trapframe->a7;
138     if(num > 0 && num < NELEM(syscalls) && syscalls[num]){
139         // Use num to lookup the system call function for num, call it,
140         // and store its return value in p->trapframe->a0
141         p->trapframe->a0 = syscalls[num](0);
142     } else {
143         printf("%d %s: unknown sys call %d\n",
144             p->pid, p->name, num);
145         p->trapframe->a0 = -1;
146     }
147 }
148
```

Below the code editor is a terminal window with the following output:

```
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ btttest
backtrace:
0x000000008000212c
0x000000008000201e
0x0000000080001d14
$ QEMU: Terminated
indifference@LAPTOP-3TB8KCCP: ~/Desktop/xv6-labs-2022$ addr2line -e kernel/kernel
0x000000008000212c
/home/indifference/Desktop/xv6-labs-2022/kernel/sysproc.c:59
0x000000008000201e
/home/indifference/Desktop/xv6-labs-2022/kernel/syscall.c:141
0x0000000080001d14
/home/indifference/Desktop/xv6-labs-2022/kernel/trap.c:76
```

```

C riscv.h M  C sysproc.c M  C trap.c X  C defs.h M  C printf.c M
kernel > C trap.c > usertrap(void)
61  p->trapframe->epc += 4;
62
63  // an interrupt will change sepc, scause, and sstatus,
64  // so enable only now that we're done with those registers.
65  intr_on();
66
67  syscall();
68  } else if((which_dev = devintr()) != 0){
69  // ok
70  } else {
71  printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
72  printf("      sepc=%p stval=%p\n", r_sepc(), r_stval());
73  setkilled(p);
74  }
75
76  if(killed(p))
77  exit(-1);
78
79  // give up the CPU if this is a timer interrupt.
80  if(which_dev == 2)
81  yield();

```

```

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ btttest
backtrace:
0x000000008000212c
0x000000008000201e
0x0000000080001d14
$ QEMU: Terminated
indifference@LAPTOP-3TB8KCCP: ~/Desktop/xv6-labs-2022$ addr2line -e kernel/kernel
0x000000008000212c
/home/indifference/Desktop/xv6-labs-2022/kernel/sysproc.c:59
0x000000008000201e
/home/indifference/Desktop/xv6-labs-2022/kernel/syscall.c:141
0x0000000080001d14
/home/indifference/Desktop/xv6-labs-2022/kernel/trap.c:76

```

Alarm

本次实验主要是实现两个系统调用 `sigalarm(int, void(*handler)())`、`sigreturn()`。通过这两个系统实现 alarm 功能，监测用户程序使用 CPU 的时长，从而对用户程序发出提醒。

- `sigalarm`：第一个参数是设置时钟中断个数当达到个数时调用指定函数，第二个参数时设置 alarm 的处理函数。
- `sigreturn`：在 alarm 处理函数中执行完所以代码后调用，主要功能是恢复中断前的状态。

监测时间主要是以时钟中断为单位(进程调度的基础)，如果时钟中断为 50ms，那么每 50ms 进行一次中断，那么通过统计用户进程进行了几次中断，就可以获知用户进程到达了警报的 CPU 使用时长，从而给出提醒。

实验准备

1. 在 makefile 中添加 alarmtest 用户程序

```

UPROGS=\
...
$U/_alarmtest\

```

2. kernel/syscall.h和kernel/syscall.c中添加函数调用声明:

```

----syscall.h
#define SYS_sigalarm 22
#define SYS_sigreturn 23

----syscall.c
extern uint64 sys_sigalarm(void);
extern uint64 sys_sigreturn(void);

static uint64 (*syscalls[])(void) = {
...
[SYS_sigalarm] sys_sigalarm,
[SYS_sigreturn] sys_sigreturn,
}

```

3. user/user.h中添加系统调用定义, user/usys.pl添加系统调用入口

```

----user.h
int sigalarm(int, void (*)(void));
int sigreturn(void);
----usys.pl
entry("sigalarm");
entry("sigreturn");

```

test0

实现sigalarm(n,fn)系统调用此函数会以n个单位时间为间隔,调用fn函数当调用sigalarm(0,0)时,表示停止sigalarm的间隔调用过程,在test0循环中,不断调用write系统调用,内核产生时钟中断,当时钟中断的数量到达触发alarm的数量时,将设置frame中epc寄存器值为处理函数入口地址,(epc储存了发生中断时用户程序的指令地址)

实现步骤

1. 在kernel/proc.h中proc结构体添加字段:

- interval: 触发alarm所需的时钟中断数量
- ticks: 统计时钟中断的数量
- handler: 记录处理alarm的函数的地址

```

struct proc{
...
int interval;
int ticks;
uint64 handler;
...
}

```


2. 在kernel/sysproc.c中添加sys_sigalarm()和sys_sigreturn()函数(sys_sigreturn()在test0中返回0即可)
- 在sys_sigalarm()中读入传进的参数，根据alarmtest.c中可知第一个参数是触发alarm的时钟中断数量，第二个参数是处理alarm的函数的地址。初始化p->interval, p->handler, p->ticks

```
7 //sys_sigalarm
8 uint64
9 sys_sigalarm(void)
10 {
11     int interval;
12     uint64 addr;
13     struct proc *p=myproc();
14
15     argint(0,&interval);
16     argaddr(1,&addr);
17
18     p->interval=interval;
19     p->handler=addr;
20     p->ticks=0;
21     return 0;
22 }
23
24 //sys_sigreturn
25 uint64
26 sys_sigreturn(void)
27 {
28     return 0;
29 }
```

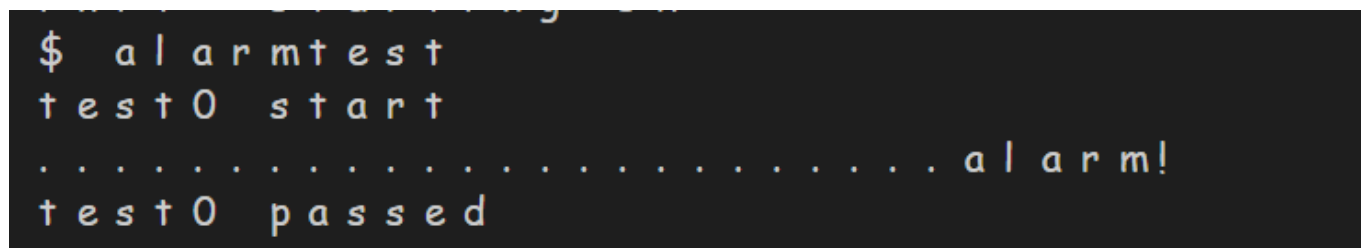
3. 在kernel/trap.c的usertrap函数中统计发生时钟中断次数，并在触发alarm时修改p->trapframe->epc。阅读代码可知which_dev=2时发生时钟中断。

```

void
usertrap(void)
{
    ...
    // give up the CPU if this is a timer interrupt.
    //需要修改, 当发生时钟中断时, 在进程的警报间隔到期时, 用户进程执行handler。
    if(which_dev == 2)
    {
        p->ticks++;
        if(p->interval>0&& p->ticks==p->interval)
        {
            p->trapframe->epc=p->handler;
        }
        else{
            yield();
        }
    }
    ...
}

```

实验结果:



```

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

```

test1,2,3

与test0对比, 需要保存中断前的函数调用栈的寄存器,在test0的基础上, 通过修改trapframe->epc的方式修改时钟中断的处理函数, 但是原本的epc被覆盖, 并且没有保存寄存器, 寄存器中的值也被覆盖。在系统调用时:

- 调用sys_sigalarm前, 寄存器信息保存在trapframe中, 执行sys_sigalarm()系统调用函数, 为ticks等字段赋值, 返回后通过trapframe的寄存器值恢复, 不用保存
- 当进行一定次数的时钟中断后, 将返回地址更改为了handler函数, 此时ret后开始执行handler函数, 此时当我们想再回到调用handler前的状态时, 会发现trapframe中存放的是sys_sigreturn执行前的trapframe, 无法返回到用户态的状态
- 从alarmtest中可以发现, 每次执行完handler会调用sigreturn函数, 用于恢复之前的状态。这要求在每次执行handler前要保存当时的寄存器信息, 保证能恢复用户态的状态。

代码实现

1. 在test0基础上在kernel/proc.h中添加新字段存储执行handler前栈的寄存器信息

```
struct trapframe *pre_trapframe; //保存被中断进程的寄存器值
```

在proc.c中进行初始化和释放空间 allocproc():

```
//Allocate a trapframe page to pre_trapframe(new)
if((p->pre_trapframe = (struct trapframe *)kalloc())==0){
    freeproc(p);
    release(&p->lock);
    return 0;
}
```

freeproc():

```
// p->lock must be held.
static void
freeproc(struct proc *p)
{
    if(p->trapframe)
        kfree((void*)p->trapframe);
    p->trapframe = 0;
    //释放空间
    if(p->pre_trapframe)
        kfree((void*)p->pre_trapframe);
    p->pre_trapframe=0;
}
```

2. 在出发alarm的时钟中断条件时，进入usertrap()函数后用pre_trapframe保存当前进程中trapframe的寄存器值

```
// give up the CPU if this is a timer interrupt.
//需要修改，当发生时钟中断时，在进程的警报间隔到期时，用户进程执行handl
if(which_dev == 2)
{
    p->ticks++;
    if(p->interval>0&& p->ticks==p->interval)
    {
        *p->pre_trapframe=*p->trapframe;
        p->trapframe->epc=p->handler;
    }
    else{
        yield();
    }
}
}
```

3. 在sys_sigreturn()中将trapframe恢复为执行handler前的栈帧值（即pre_trapframe），由于系统调用会将返回值储存于a0寄存器，故作为返回值

```
3
4 //sys_sigreturn
5 uint64
6 sys_sigreturn(void)
7 {
8     struct proc* p=myproc();
9     *p->trapframe=*p->pre_trapframe;
0     p->ticks=0;
1     return p->pre_trapframe->a0;
2 }
```

实验结果

执行alarmtest通过test1/test2/test3测试 执行usertests -q确保没有破坏其他函数功能

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ alarmtest
test0 start
..... alarm!
test0 passed
test1 start
... alarm!
... alarm!
... alarm!
.. alarm!
... alarm!
... alarm!
.. alarm!
... alarm!
... alarm!
.. alarm!
test1 passed
test2 start
..... alarm!
test2 passed
test3 start
test3 passed
```

```
test sbrklast: OK
test sbrk8000: OK
test badarg: OK
ALL TESTS PASSED
```