

# Lab4 实验报告

## 一、实验思路、相关代码截图、结果截图

### Backtrace

**实验目的：** 实现 backtrace 功能，用于打印出栈上调用链的所有返回地址

### 实验思路：

首先根据提示在 riscv.h 中添加函数 r\_fp ()：

已知当前函数的帧指针存在寄存器 s0 中，在 kernel/riscv.h 中添加获得该值的函数。

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

根据提示，实现 backtrace 函数，首先输出一个 backtrace 的字段，然后获取当前的函数，并且使用 PGUPGROUND() 获得当前栈的顶部地址；由 RISC-V 的栈结构可知，返回地址存放在帧指针的 -8 偏移量处，保存的帧指针存放在帧指针的 -16 偏移量处。

```
void
backtrace(void)
{
    printf("backtrace:\n");
    uint64 fp = r_fp();
    uint64 base = PGROUNDUP(fp);
    while(fp < base) {
        printf("%p\n", *((uint64*)(fp - 8)));
        fp = *((uint64*)(fp - 16));
    }
}
```

最后在 sys\_sleep () 中调用 backtrace 函数即可。

```
uint64
sys_sleep(void)
{
    int n;
    uint ticks0;

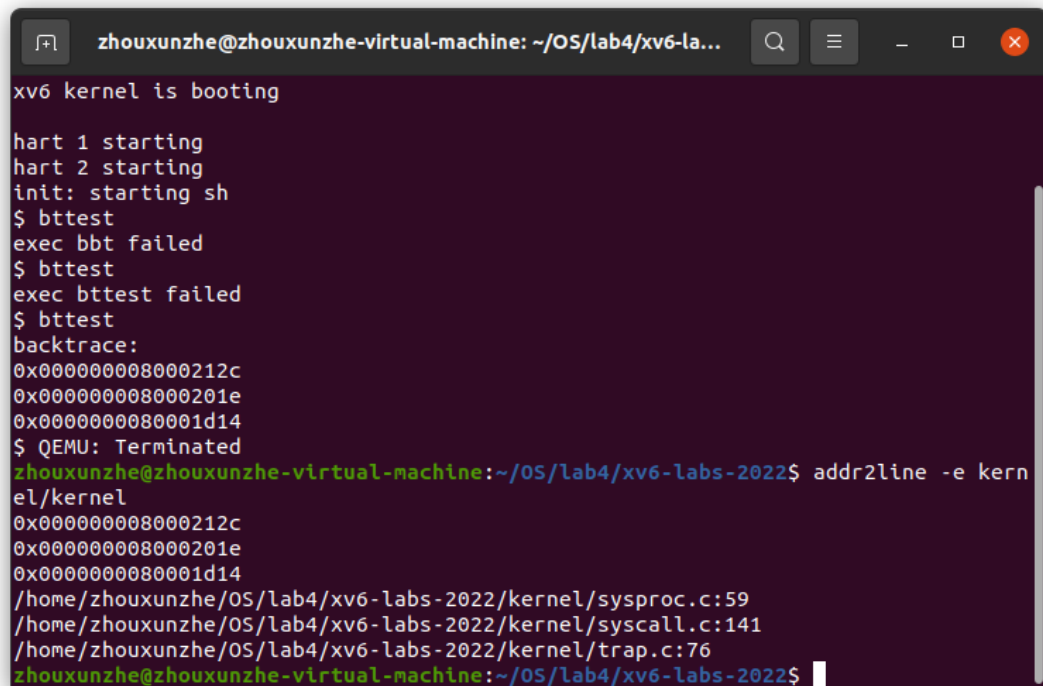
    backtrace();
}
```

执行 bttest:

```
xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ backtrace
exec backtrace failed
$ bttest
backtrace:
0x000000008000212c
0x000000008000201e
0x0000000080001d14
$
```

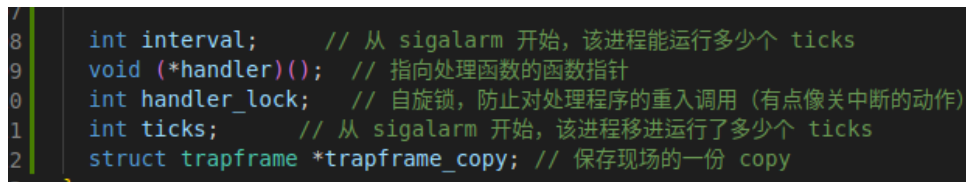
可以看到输出三个地址，然后执行 `addr2line -e kernel/kernel`，查找地址对应的代码：



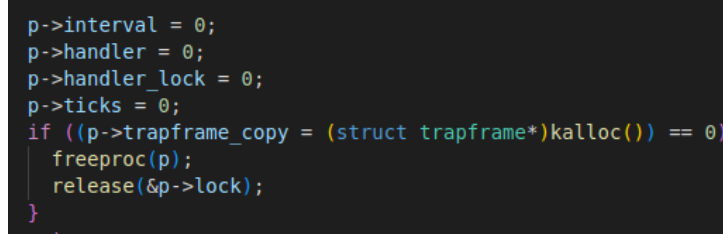
```
zhexunzhe@zhexunzhe-virtual-machine: ~/OS/lab4/xv6-la...
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ bttest
exec bbt failed
$ bttest
exec bttest failed
$ bttest
backtrace:
0x000000008000212c
0x000000008000201e
0x0000000080001d14
$ QEMU: Terminated
zhexunzhe@zhexunzhe-virtual-machine:~/OS/lab4/xv6-labs-2022$ addr2line -e kern
el/kernel
0x000000008000212c
0x000000008000201e
0x0000000080001d14
/home/zhexunzhe/OS/lab4/xv6-labs-2022/kernel/sysproc.c:59
/home/zhexunzhe/OS/lab4/xv6-labs-2022/kernel/syscall.c:141
/home/zhexunzhe/OS/lab4/xv6-labs-2022/kernel/trap.c:76
zhexunzhe@zhexunzhe-virtual-machine:~/OS/lab4/xv6-labs-2022$
```

## Alarm

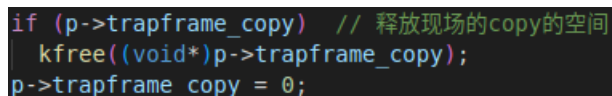
首先在 `struct proc` 中添加相关变量，并添加构造和析构动作，用于辅助 `sigalarm()` 和 `sigreturn()` 系统调用实现，由于需要实现寄存器的值在调用 `sys_sigreturn()` 后仍然保持不变，所以需要使用页来存储。



```
7
8     int interval;      // 从 sigalarm 开始，该进程能运行多少个 ticks
9     void (*handler)(); // 指向处理函数的函数指针
10    int handler_lock;   // 自旋锁，防止对处理程序的重入调用（有点像关中断的动作）
11    int ticks;         // 从 sigalarm 开始，该进程移进运行了多少个 ticks
12    struct trapframe *trapframe_copy; // 保存现场的一份 copy
```



```
p->interval = 0;
p->handler = 0;
p->handler_lock = 0;
p->ticks = 0;
if ((p->trapframe_copy = (struct trapframe*)kalloc()) == 0)
    freeproc(p);
    release(&p->lock);
}
```



```
if (p->trapframe_copy) // 释放现场的copy的空间
    kfree((void*)p->trapframe_copy);
p->trapframe_copy = 0;
```

根据提示，实现一个用来存储和导出寄存器的值的函数。

```

31
32 void store(struct trapframe* tf_from, struct trapframe* tf_to) {
33     tf_to->kernel_satp = tf_from->kernel_satp;
34     tf_to->kernel_sp = tf_from->kernel_sp;
35     tf_to->kernel_trap = tf_from->kernel_trap;
36     tf_to->epc = tf_from->epc;
37     tf_to->kernel_hartid = tf_from->kernel_hartid;
38     tf_to->ra = tf_from->ra;
39     tf_to->sp = tf_from->sp;
40     tf_to->gp = tf_from->gp;
41     tf_to->tp = tf_from->tp;
42     tf_to->t0 = tf_from->t0;
43     tf_to->t1 = tf_from->t1;
44     tf_to->t2 = tf_from->t2;
45     tf_to->s0 = tf_from->s0;
46     tf_to->s1 = tf_from->s1;
47     tf_to->a0 = tf_from->a0;
48     tf_to->a1 = tf_from->a1;
49     tf_to->a2 = tf_from->a2;
50     tf_to->a3 = tf_from->a3;
51     tf_to->a4 = tf_from->a4;
52     tf_to->a5 = tf_from->a5;
53     tf_to->a6 = tf_from->a6;
54     tf_to->a7 = tf_from->a7;
55     tf_to->s2 = tf_from->s2;
56     tf_to->s3 = tf_from->s3;
57     tf_to->s4 = tf_from->s4;
58     tf_to->s5 = tf_from->s5;
59     tf_to->s6 = tf_from->s6;
60     tf_to->s7 = tf_from->s7;
61     tf_to->s8 = tf_from->s8;
62     tf_to->s9 = tf_from->s9;
63     tf_to->s10 = tf_from->s10;

```

由于需要存储 36 个寄存器以及类中变量的值，所以工作量会很大。  
然后为 sigalarm() 和 sigreturn() 系统调用添加相关声明

```

int sigalarm(int, void(*)(void));
int sigreturn(void);

entry("sigalarm"); #define SYS_sigreturn 22
entry("sigreturn"); #define SYS_sigalarm 23

extern uint64 sys_sigreturn(void);
extern uint64 sys_sigalarm(void);

[SYS_sigreturn] sys_sigreturn,
[SYS_sigalarm] sys_sigalarm,

```

根据提示，实现 sys\_sigalarm() 和 sys\_sigreturn() 系统调用

```

uint64
sys_sigalarm(void) {
    int interval;
    uint64 handler;
    argint(0, &interval);
    argaddr(1, &handler);
    myproc()->interval = interval;
    myproc()->handler = (void(*)(void))handler;
    return 0;
}

uint64
sys_sigreturn(void) {
    stdr(myproc()->trapframe_copy, myproc()->trapframe);
    myproc()->ticks = 0;
    myproc()->handler_lock = 0;
    return myproc()->trapframe->a0;
}

```

其中，sys\_sigalarm() 要从 a0 和 a1 获得用户调用系统调用时传入的 ticks 数和处理函

数，并且用这些值更新 struct proc 里的成员。sys\_sigreturn() 需要恢复现场、清除刚刚设置的成员、解除对处理函数的锁；由于中断处理对用户来说应当是透明的，当 sys\_sigreturn() 返回值时会更新 a0 寄存器的值，因此直接让其返回当前 a0 的值以保证不变。

进行测试：

```
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
$
```

然后使用 usertests -q 测试是否更改 kernel：

```
sepc=0x00000000000004994 stval=0x0000000000013000
OK
test sbrkarg: OK
test validate: OK
test bsstest: OK
test bigargtest: OK
test argptest: OK
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6563
sepc=0x00000000000002410 stval=0x0000000000010eb0
OK
test textwrite: usertrap(): unexpected scause 0x000000000000000f pid=6565
sepc=0x00000000000002490 stval=0x0000000000000000
OK
test pgbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=6568
sepc=0x00000000000005c5e stval=0x00000000000005c5e
usertrap(): unexpected scause 0x000000000000000c pid=6569
sepc=0x00000000000005c5e stval=0x00000000000005c5e
OK
test sbrklast: OK
test sbrk8000: OK
test badarg: OK
ALL TESTS PASSED
$
```

## 二、实验中遇到的问题，如何思考并解决

实验中首先遇到的问题就是由于一开始对于 RISC-V 栈结构不了解，所以在实现 backtrace 是，对于 PGUPGROUND()的使用并不是很理解，后续研究其栈结构后问题解决。

然后在实现 alarm 中，一开始，并没有使用页表结构进行系统调用，然后在测试 test3 时，会出现报错 register a0 changed，仔细思考，原因是，没有用页表结构进行存储。所以寄存器都是存在栈中，当系统调用后，其值就会改变。了解原因后，增加页表进行系统调用，问题解决。

## 三、实验总结

在本次实验中，我学会了 trap 的原理以及如何通过系统调用观察其流程。除此而外，还通过 gdb 调试了解了 riscv 的基本结构，通过 backtrace 实验，学会了如何观察系统中断，通过 alarm 实验，学会了如何利用寄存器进行系统调用输出特定函数系统中断所需要的 tick 数。总的来说，本次实验增强了我对 riscv 的结构的了解，也让我更加理解操作系统中系统中断的原理。