

实验报告

Lab 4

traps

姓名：张明瑞

班级：2020 级信息安全

学号：20307130247

一、实验

1.RISC-V assembly

```
39 void main(void) {
40 1c: 1141                addi    sp,sp,-16
41 1e: e406                sd     ra,8(sp)
42 20: e022                sd     s0,0(sp)
43 22: 0800                addi    s0,sp,16
44 printf("%d %d\n", f(8)+1, 13);
45 24: 4635                li     a2,13
46 26: 45b1                li     a1,12
47 28: 00000517            auipc   a0,0x0
48 2c: 7c850513            addi    a0,a0,1992 # 7f0 <malloc+0xe8>
49 30: 00000097            auipc   ra,0x0
50 34: 61a080e7            jalr    1562(ra) # 64a <printf>
51 exit(0);
52 38: 4501                li     a0,0
53 3a: 00000097            auipc   ra,0x0
54 3e: 298080e7            jalr    664(ra) # 2d2 <exit>
55
```

(1)

函数的参数保存在寄存器 a0-a7 里面。

在 main 对 printf 的调用中，寄存器 a2 保存 13。

(2)

main 中没有直接调用 f 的代码，编译器将 g 内联至 f 中，又将 f 内联到 main 中，产生函数调用的结果。

(3)

```
1111 000000000000064a <printf>:
1112
1113 void
1114 printf(const char *fmt, ...)
1115 {
1116 64a: 711d                addi    sp,sp,-96
1117 64c: ec06                sd     ra,24(sp)
1118 64e: e822                sd     s0,16(sp)
1119 650: 1000                addi    s0,sp,32
1120 652: e40c                sd     a1,8(s0)
1121 654: e810                sd     a2,16(s0)
1122 656: ec14                sd     a3,24(s0)
1123 658: f018                sd     a4,32(s0)
1124 65a: f41c                sd     a5,40(s0)
1125 65c: 03043823            sd     a6,48(s0)
1126 660: 03143c23            sd     a7,56(s0)
```

通过寻找可知函数 printf()的地址为 0x64a。

(4)

jalr 会将 pc+4 即下条指令的地址的值存储到当前寄存器，所以在 jalr 到 main 中的 printf 之后，寄存器 ra 中存储的值是 0x38。

(5)

%x 表示按十六进制输出，%s 表示从指定地址向后输出字符直到遇见\0。

57616 的十六进制表示为 e110，而 0x00646c72 按照小端排序其内容为 72 6c 64 00，即 rld\0。

①输出为 He110 World

②如果 RISC-V 是大端排序，可以将 i 的值更改为 0x726c6400，不需要更改 57616。

(6)

X 输出的值为 3，y 输出的值是寄存器内当前的值，并不确定。

2.Backtrace

Part I 实验步骤

(1) 在 kernel/defs.h 中添加函数声明：

```

79 // printf.c
80 void      printf(char*, ...);
81 void      panic(char*) __attribute__((noreturn));
82 void      printfinit(void);
83 void      backtrace(void);

```

(2) 在 kernel/riscv.h 中添加实验给出的函数 r_fp():

```

330 static inline uint64
331 r_fp()
332 {
333     uint64 x;
334     asm volatile("mv %0, s0" : "=r" (x) );
335     return x;
336 }

```

(3) 在 kernel/printf.c 中实现函数 backtrace():

```

138 void
139 backtrace()
140 {
141     printf("backtrace:\n");
142     uint64 fp = r_fp();
143     uint64 Bottom = PGROUNDDOWN(fp);
144     for(; fp > Bottom; fp = *((uint64*)(fp - 16)))
145         printf("%p\n", *((uint64*)(fp-8)));
146 }

```

首先调用 r_fp() 获取当前帧指针并将其保存在变量 fp 中，然后使用函数 PGROUNDDOWN() 获取堆栈页面的底部地址，以偏移量 16 为单位不断将 fp 指针指向的区域向前移动，并输出栈帧所保存的返回地址，直到遍历整个页表。

(4) 在函数 kernel/printf.c 中的函数 panic() 和 kernel/sysproc.c 中的函数 sys_sleep() 中添加对函数 backtrace() 的调用：

```

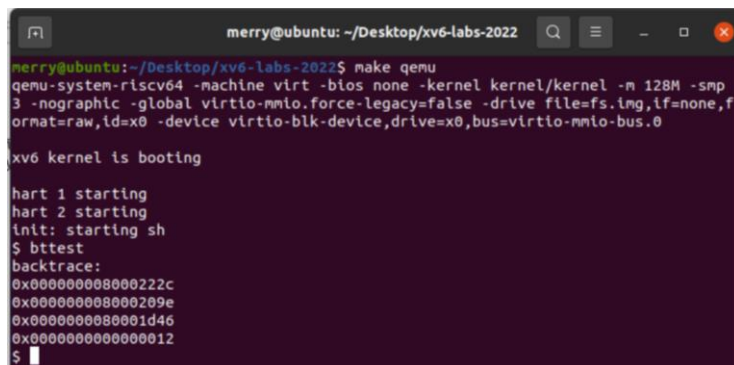
124     printf("\n");
125     backtrace();
126     panicked = 1; // freeze uart output from other CPUs
127     for(;;)
128         ;

67     sleep(&ticks, &tickslock);
68 }
69 release(&tickslock);
70 backtrace();
71 return 0;

```

Part II 测试结果

在编译后输入 bttest 指令：



```

merry@ubuntu: ~/Desktop/xv6-labs-2022
merry@ubuntu:~/Desktop/xv6-labs-2022$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
3 -nographic -global virtio-mmio.force-legacy=false -drive file=fs.img,if=none,f
ormat=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ bttest
backtrace:
0x000000000000222c
0x000000000000209e
0x0000000000001d46
0x0000000000000012
$

```

可以发现命令行中成功打印出四个地址。

3.Alarm

Part I 实验步骤

Test 0:

(1) 在 user/user.h 中添加函数 sigalarm()和 sigreturn()的声明:

```
25 int sigalarm(int ticks, void (*handler)());
26 int sigreturn(void);
```

(2) 在 user/usys.pl 中添加用以生成汇编代码的 entry():

```
39 entry("sigalarm");
40 entry("sigreturn");
```

(3) 在 kernel/syscall.h 中添加函数 sigalarm()和 sigreturn()的系统调用号:

```
23 #define SYS_sigalarm 22
24 #define SYS_sigreturn 23
```

(4) 在 kernel/syscall.c 中添加与函数 sigalarm()和 sigreturn()相对应的函数名数组内容:

```
104 extern uint64 sys_sigalarm(void);
105 extern uint64 sys_sigreturn(void);

131 [SYS_sigalarm] sys_sigalarm,
132 [SYS_sigreturn] sys_sigreturn,
133 };
```

(5) 在 kernel/proc.h 的结构 proc 中添加新变量:

```
107 int interval;
108 int ticks;
109 uint64 handler;
```

其中 interval 表示时钟间隔, ticks 表示已经经过的时钟数, handler 用以记录调用的函数信息。

(6) 在 kernel/proc.c 中对新增变量进行初始化和释放:

```
134 // An empty user page table.
135 p->pagetable = proc_pagetable(p);
136 if(p->pagetable == 0){
137     freeproc(p);
138     release(&p->lock);
139     return 0;
140 }
141
142 // Set up new context to start executing at forkret
143 // which returns to user space.
144 memset(&p->context, 0, sizeof(p->context));
145 p->context.ra = (uint64)forkret;
146 p->context.sp = p->kstack + PGSIZE;
147
148 p->interval = 0;
149 p->handler = 0;
150 p->ticks = 0;
151 p->pretf = (struct trapframe*) kalloc();
152 return p;
153 }

158 static void
159 freeproc(struct proc *p)
160 {
161     if(p->trapframe)
162         kfree((void*)p->trapframe);
163     p->trapframe = 0;
164     if(p->pagetable)
165         proc_freepagetable(p->pagetable, p->sz);
166     p->pagetable = 0;
167     p->sz = 0;
168     p->pfd = 0;
169     p->parent = 0;
170     p->name[0] = 0;
171     p->chan = 0;
172     p->killed = 0;
173     p->xstate = 0;
174     p->interval = 0;
175     p->handler = 0;
176     p->ticks = 0;
177     if(p->pretf)
178         kfree((void *)p->pretf);
```

分别是在 allocproc()和 freeproc()中进行。

(7) 在 kernel/sysproc.c 中添加函数 sigalarm()和 sigreturn():

```

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

```

其中 Test 0 主要是要实现函数 sigalarm(), 其功能为读取当前时钟间隔和函数 handler()调用信息, 并将时钟数置零, 然后将其保存至进程 myproc()中。

Test 1:

- (1) 为了便于 sigreturn()恢复用户调用 handler 之前的状态, 在 kernel/proc.h 的结构体 proc 中新增变量 pretf, 用以保存指向前一个 trapframe 的指针:

```

110 struct trapframe *pretf;
111 };

```

- (2) 在 kernel/proc.c 中对其进行初始化和释放 (同上 Test 0 步骤(6))
- (3) kernel/trap.c, 在时钟中断处理中, 判断本次是否调用了 handler(), 如果对其进行了调用, 则使用 p->pretf 保存当前的 trapframe:

```

79 // give up the CPU if this is a timer interrupt.
80 if(which_dev == 2){
81     if(p->interval != 0){
82         if(p->ticks == p->interval){
83             *p->pretf = *p->trapframe;
84             p->trapframe->epc = p->handler;
85         }
86         p->ticks++;
87     }
88     yield();

```

- (4) 使用 sigreturn 恢复至原先的状态:

```

110 uint64
111 sys_sigreturn(void)
112 {
113     struct proc *p = myproc();
114     *p->trapframe = *p->pretf;
115     p->ticks = 0;
116     return p->trapframe->a0;
117 }

```

同时返回寄存器 a0 的值以进行还原。

Part II 测试结果:

在 Makefile 的 UPROS 中添加对 alarmtest.c 的调用:

```

174 UPROGS=
175 SU/_cat\
176 SU/_echo\
177 SU/_forktest\
178 SU/_grep\
179 SU/_init\
180 SU/_kill\
181 SU/_ln\
182 SU/_ls\
183 SU/_mkdir\
184 SU/_rm\
185 SU/_sh\
186 SU/_stressfs\
187 SU/_usertests\
188 SU/_grind\
189 SU/_wcl\
190 SU/_zombie\
191 SU/_alarmtest\

```

运行命令 `alarmtest` 和 `usertests -q`:

```

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

test sbrkarg: OK
test validate: OK
test bsstest: OK
test bigargtest: OK
test argptest: OK
test stacktest: usertrap
sepc=0x00000000
OK
test textwrite: usertrap
sepc=0x00000000
OK
test pgbug: OK
test sbrkbugs: usertrap
sepc=0x00000000
usertrap(): unexpected
sepc=0x00000000
OK
test sbrklast: OK
test sbrk8000: OK
test badarg: OK
ALL TESTS PASSED

```

(另外值得一提的是网站最后的测试图片中的指令有误，应将 `usertest -q` 更改为 `usertests -q`)

二、实验感想

本次实验我个人觉得比较有难度，我也因此去参考了不少博客，有关于 xv6 源码的，有关于 RISC-V 的，也有关于实验本身的，但任务越有挑战性就越能历练人，现在我应该能理解一些基本的系统调用和功能函数如何起作用了。

我时常因此感叹系统设计的精妙，统筹管理或许正是操作系统课程在计算机系列课程中的独特之处。

参考链接：

[1]Lab: traps

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

[2]MIT 6.S081 Lab4: traps

<https://blog.csdn.net/rocketeerLi/article/details/121665215>

[3] RISC-V 指令详解

<https://blog.csdn.net/ybhuangfugui/article/details/127330016>