

Lab4: Traps

2351882 王小萌

Tongji University, 2025 Summer

代码仓库链接: <https://github.com/wmxmw/Tongji-University-xv6-labs-2021.git>

1. RISC-V assembly

1.1 实验目的

1.2 实验步骤

1.3 实验中遇到的问题和解决办法

1.4 实验心得

2. Backtrace

2.1 实验目的

2.2 实验步骤

2.3 实验中遇到的问题和解决办法

2.4 实验心得

3 Alarm

3.1 实验目的

3.2 实验步骤

3.3 实验中遇到的问题和解决办法

3.4 实验心得

4 实验检验得分

本实验探索系统调用如何通过陷阱 (trap) 来实现。首先, 将进行一个利用栈的热身练习, 然后实现一个用户级陷阱处理 (user-level trap handling) 的示例。

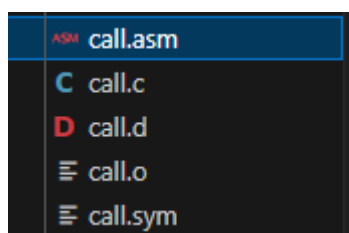
1 RISC-V assembly

1.1 实验目的

了解一些 RISC-V 汇编很重要。在 xv6 repo 中有一个文件 user/call.c。make fs.img 会对其进行编译, 并生成 user/call.asm 中程序的可读汇编版本。

1.2 实验步骤

1. 在 xv6 的命令行中输入运行 make fs.img, 编译 user/call.c 程序, 得到可读性比较强的 user/call.asm 文件。



代码如下:

```
user > C call.c
1  #include "kernel/param.h"
2  #include "kernel/types.h"
3  #include "kernel/stat.h"
4  #include "user/user.h"
5
6  int g(int x) {
7      return x+3;
8  }
9
10 int f(int x) {
11     return g(x);
12 }
13
14 void main(void) {
15     printf("%d %d\n", f(8)+1, 13);
16     exit(0);
17 }
18
```

```
13  int g(int x) {
14      0: 1141                addi sp,sp,-16
15      2: e422                sd s0,8(sp)
16      4: 0800                addi s0,sp,16
17      return x+3;
18  }
19      6: 250d                addiw a0,a0,3
20      8: 6422                ld s0,8(sp)
21      a: 0141                addi sp,sp,16
22      c: 8082                ret
23
24  000000000000000e <f>:
25
26  int f(int x) {
27      e: 1141                addi sp,sp,-16
28      10: e422                sd s0,8(sp)
29      12: 0800                addi s0,sp,16
30      return g(x);
31  }
32      14: 250d                addiw a0,a0,3
33      16: 6422                ld s0,8(sp)
34      18: 0141                addi sp,sp,16
35      1a: 8082                ret
36
37  000000000000001c <main>:
```

2. 阅读 call.asm 中的 g ,f 和 main 函数。回答下列问题:

Q1

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

哪些寄存器保存函数的参数? 例如, 在 main 对 printf 的调用中哪个寄存器保存 13?

```
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
```

第三个参数 (13) 在 a2。 因此, 13 保存在 a2 寄存器中。

Q2

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.)

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 (提示: 编译器可能会将函数内联)

```

25
26 int f(int x) {
27     e: 1141          addi sp,sp,-16
28     10: e422          sd s0,8(sp)
29     12: 0800          addi s0,sp,16
30     return g(x);
31 }
32     14: 250d          addiw a0,a0,3
33     16: 6422          ld s0,8(sp)
34     18: 0141          addi sp,sp,16
35     1a: 8082          ret
36
37 000000000000001c <main>:

```

```

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: 7b050513      addi a0,a0,1968 # 7d8 <malloc+0xea>

```

在 main 函数中没有直接的函数调用指令，而是内联了 f 和 g 的计算结果。

Q3

At what address is the function printf located?

printf 函数位于哪个地址？

```

48     2c: 7b050513      addi a0,a0,1968 # 7d8 <malloc+0xea>
49     30: 00000097      auipc ra,0x0
50     34: 600080e7      jalr 1536(ra) # 630 <printf>
51     exit(0);

```

630 <printf> 表示 printf 函数的起始地址是 0x630。

```

1092 0000000000000630 <printf>:
1093
1094 void
1095 printf(const char *fmt, ...)
1096 {
1097     630: 711d ..... addi sp,sp,-96
1098     632: ec06          sd ra,24(sp)
1099     634: e822          sd s0,16(sp)
1100     636: 1000          addi s0,sp,32
1101     638: e40c          sd a1,8(s0)
1102     63a: e810          sd a2,16(s0)
1103     63c: ec14          sd a3,24(s0)
1104     63e: f018          sd a4,32(s0)
1105     640: f41c          sd a5,40(s0)
1106     642: 03043823      sd a6,48(s0)
1107     646: 03143c23      sd a7,56(s0)
1108     va_list ap;

```

也可以查阅得到其地址在 0x630 。

Q4

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

在 main 中 printf 的 jalr 之后的寄存器 ra 中有什么值？

```

49 30: 00000097      auipc ra,0x0
50 34: 600080e7      jalr 1536(ra) # 630 <printf>
51 exit(0);
52 38: 4501          li a0,0
53 3a: 00000097      auipc ra,0x0
54 3e: 27e080e7      jalr 638(ra) # 2b8 <exit>
55

```

在执行 jalr 指令时，ra 寄存器会保存返回地址，也就是 jalr 指令的下一条指令的地址。因此执行 jalr 指令后，ra 寄存器中保存的值是 0x38，即 main 函数中 printf 调用之后的返回地址。

Q5

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?

Here's a description of little- and big-endian and a more whimsical description.

```
user > C call.c
10  int f(int x) {
11      ...
12  }
13
14  void main(void) {
15      unsigned int i = 0x00646c72;
16      printf("H%x Wo%s", 57616, &i);
17      |
18      exit(0);
19  }
20
```

输出:

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

输出为 HE110 World。若为大端对齐, i 需要设置为 0x726c6400, 不需要改变 57616 的值 (因为他是按照二进制数字读取的而非单个字符)。

Q6

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);
```

```
14  void main(void) {
15      |printf("x=%d y=%d", 3);
16
17      exit(0);
18  }
19
```

输出:

```
xv6 kernel is booting
hart 2 starting
hart 1 starting
init: starting sh
$ call
x=3 y=5221$
```

在这段代码中，printf 函数的格式字符串要求两个整数参数，但实际只提供了一个整数参数 3。由于 printf 期望两个参数，而只提供了一个，这会导致未定义行为。具体来说，“y=”之后将打印什么取决于栈中紧接着的内容，这些内容可能是任何值。

1.3 实验中遇到的问题和解决办法

问题： 注意 make fs.img。

解决办法： 要在终端执行。

1.4 实验心得

了解系统调用所发挥的重要作用，更清楚的认识到了系统调用的作用。

2 Backtrace

2.1 实验目的

实现一个回溯（backtrace）功能，用于在操作系统内核发生错误时，输出调用堆栈上的函数调用列表。这有助于调试和定位错误发生的位置。

2.2 实验步骤

1. 在 kernel/defs.h 中添加 backtrace 函数的原型 void backtrace(void); 以便在 sys_sleep 中调用该函数。

```
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 中添加以下内联汇编函数

```

334 static inline uint64
335 r_fp()
336 {
337     uint64 x;
338     asm volatile("mv %0, se" : "=r" (x));
339     return x;
340 }
341

```

3. 在 kernel/printf.c 中实现一个名为 backtrace 的函数。这个函数的目标是通过遍历调用堆栈 中的帧指针来输出保存在每个栈帧中的返回地址。

```

136 void backtrace(void)
137 {
138     printf("backtrace:\n");
139     uint64 fp = r_fp();
140     while(fp < PGROUNDUP(fp))
141     {
142         printf("%p\n", *(uint64*)(fp-8));
143         fp = *(uint64*)(fp-16);
144     }
145 }

```

4. 在 sysproc.c 中的 sys_sleep 函数中调用 backtrace 函数。

```

69     }
70     sleep(&ticks, &tickslock);
71 }
72 release(&tickslock);
73 backtrace();
74 return 0;
75 }
76

```

5. 在 kernel/printf.c 的 panic() 函数中添加对 backtrace() 的调用

```

117 void
118 panic(char *s)
119 {
120     pr.locking = 0;
121     printf("panic: ");
122     printf(s);
123     printf("\n");
124     panicked = 1; // freeze uart output from other CPUs
125     backtrace();
126     for(;;)
127     ;
128 }
129

```

6. 终端执行 make qemu 编译运行 xv6。在命令行中输入 bttest 命令结果：

```
xv6 kernel is booting
hart 1 starting
hart 2 starting
init: starting sh
$ bttest
backtrace:
0x0000000080002144
0x0000000080001fa6
0x0000000080001c90
$ _
```

```
0x000 000 00 80002144
```

```
0x0000000080001fa6
```

```
0x0000000080001c90
```

退出 xv6 后运行 `addr2line -e kernel/kernel` 将以上输出作为输入，输出对应的调用栈函数。

```
vale@Puppyyoo: /xv6-labs-2021$ addr2line -e kernel/kernel
0x0000000080002144
/home/vale/xv6-labs-2021/kernel/sysproc.c:74
0x0000000080001fa6
/home/vale/xv6-labs-2021/kernel/syscall.c:140
0x0000000080001c90
/home/vale/xv6-labs-2021/kernel/trap.c:76
```

7. 实验单项评分

```
vale@Puppyyoo: /xv6-labs-2021$ ./grade-lab-traps backtrace
make: 'kernel/kernel' is up to date.
== Test backtrace test == backtrace test: OK (1.4s)
```

2.3 实验中遇到的问题和解决办法

问题：在 `backtrace` 函数中如何判断循环终止条件。

解决办法：使用 `PGROUNDOWN` 和 `PGROUNDUP` 宏可以帮助计算栈页的顶部和底部地址，从而确定循环终止的条件。

2.4 实验心得

在本次实验中，通过编写 `backtrace()` 函数并成功实现栈帧信息的输出，我深刻体会到了对底层栈结构的理解和对系统调用栈的掌握的重要性。特别是在解决获取上一级栈帧的终止条件和栈帧指针有效性检查的问题时，进一步加深了我对 RISC-V 架构和操作系统内部机制的认识。

3 Alarm

3.1 实验目的

本次实验将向 xv6 内核添加一个新的功能，即周期性地为进程设置定时提醒。这个功能类似于用户级的中断/异常处理程序，能够让进程在消耗一定的 CPU 时间后执行指定的函数，然后恢复执行。通过实现这个功能，我们可以为计算密集型进程限制 CPU 时间，或者为需要周期性执行某些操作的进程提供支持。

3.2 实验步骤

1. 在 Makefile 中添加 `$U/_alarmtest\`，以便将 `alarmtest.c` 作为 xv6 用户程序编译。

```
188     $U/_grind\  
189     $U/_wc\  
190     $U/_zombie\  
191     $U/_alarmtest\  
192
```

2. 在 `user/user.h` 中设置正确的声明，两个系统调用的入口，分别用于设置定时器和从定时器中断处理过程中返回：

```
25     int uptime(void);  
26     int sigalarm(int ticks, void (*handler)());  
27     int sigreturn(void);  
28
```

3. 更新 `user/usys.pl`（用于生成 `user/usys.S`）：在 `usys.pl` 中添加相应的用户态库函数入口。

```
38     entry("uptime");  
39     entry("sigalarm");  
40     entry("sigreturn");  
41  
42
```

4. 在 `kernel/syscall.h` 中声明 `sigalarm` 和 `sigreturn` 的用户态库函数：

```
21     #define SYS_mkdir 20  
22     #define SYS_close 21  
23     #define SYS_sigalarm 22  
24     #define SYS_sigreturn 23  
25
```

5. 在 `syscall.c` 中添加对应的系统调用处理函数：在

```
131     [SYS_close]    sys_close,  
132     [SYS_sigalarm] sys_sigalarm,  
133     [SYS_sigreturn] sys_sigreturn,  
134     };
```

```

105  extern uint64 sys_write(void);
106  extern uint64 sys_uptime(void);
107  extern uint64 sys_sigalarm(void);
108  extern uint64 sys_sigreturn(void);

```

6. .kernel/proc.h 在 sys_sigalarm 中，将警报间隔和处理函数的指针存储在 proc 结构体的新字段中； sys_sigreturn 只返回零：

```

108
109      int alarm_interval;           //alarm_interval
110      void(*alarm_handler)();      //alarm_handler
111      int ticks;                   //timer ticks
112      struct trapframe *saved_trapframe; //register for save and return
113      int alarm_handling;
114  };

```

7. 在 proc.c 中的 allocproc() 函数中初始化 proc 的这些字段。

```

144
145      p->alarm_handler = 0;
146      p->alarm_interval=0;
147      p->ticks=0;
148      p->alarm_handling=0;
149      if((p->savetrapframe = (struct trapframe*)kalloc())==0){
150          freeproc(p);
151          release(&p->lock);
152          return 0;
153      }
154      return p;
155  }
156

```

8. 在 sysproc.c 中实现 sys_sigalarm 和 sys_sigreturn 的内核处理逻辑。

```

116      uint64
117      sys_sigreturn(void)
118      {
119          struct proc* p=myproc();
120          copytrapframe(p->trapframe,p->savetrapframe);
121          p->alarm_handling = 0;
122          return 0;
123      }
124

```

```

100     uint64
101     sys_sigalarm(void)
102     {
103         int interval;
104         uint64 handler;
105         if(argint(0,&interval)<0)
106             return -1;
107         if(argaddr(1,&handler)<0)
108             return -1;
109         struct proc* proc =myproc();
110         proc->alarm_interval=interval;
111         proc->alarm_handler = (void(*)(void))handler;
112
113         return 0;
114     }
115

```

9. 修改 kernel/trap.c 中的 usertrap() 函数，实现周期性执行函数，判断是否为定时器中断的 if 语句块修改定时器中断发生时的行为，为每个有 sigalarm 的进程更新其已经消耗的 ticks 数；判断该进程已经使用的 ticks 数是否已经足以触发 alarm：

```

107     } else if((which_dev = devintr()) != 0){
108         // ok
109         if(which_dev ==2){
110             //interrupt
111             p->ticks++;
112             if(p->alarm_interval!=0&& p->ticks%p->alarm_interval==0){
113                 if(p->alarm_handling==0){
114                     p->alarm_handling =1;
115                     cpytrapframe(p->saved_trapframe,p->trapframe);
116                     p->trapframe->epc = (uint64)p->alarm_handler;
117                 }
118             }
119         }
120     } else {

```

10. 在 sysproc.c 中增加 cpytrapframe 函数声明：

```

9
10 void cpytrapframe(struct trapframe* ,struct trapframe *);
11

```

11. 编译运行。执行 alarmtest ， 结果如下：

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

执行 `usertests` 结果如下:

```
OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
```

12. 单项评分:

```

vale@Puppyyoo: /xv6-labs-2021$ ./grade-lab-traps alarmtest
make: 'kernel/kernel' is up to date.
= Test running alarmtest = (3.6s)
= Test  alarmtest: test0 =
  alarmtest: test0: OK
= Test  alarmtest: test1 =
  alarmtest: test1: OK
= Test  alarmtest: test2 =
  alarmtest: test2: OK

```

3.3 实验中遇到的问题和解决办法

问题：运行测试时发现 test1/test2() 无法通过。

解决办法：是因为在执行定时函数 handler 后，进程无法正确恢复到中断前的状态，导致程序继续执行时出现错误。为了解决 这个问题，我们需要在进入定时函数前保存进程的 trapframe 状态，在定时函数执行完成后 恢复该状态。为此，我们在进程结构体 struct proc 中添加了一个新的字段 trapframecopy 来存储 trapframe 的副本。

3.4 实验心得

这次实验使我深入理解了操作系统的信号处理机制，通过实现定时提醒功能，我学会了如何在内核中添加系统调用、管理进程状态和处理中断，提升了系统编程和调试能力。


力。此外，这次 实验也涉及到用户态和管理态的转换，我再次巩固了如何设置声明和入口使得二者连接。实验中遇到的挑战，如正确保存和恢复 trapframe 以及防止函数重入，使我认识到细致的状态管理和全面的 测试对于系统开发的重要性。

4 实验检验得分

1. 在实验目录下创建 time.txt，填写完成实验时间数。

 time.txt	2025/7/10 0:01	文本文档	1 KB
--------------------------------------------------------------------------------------------	----------------	------	------

2. 创建 answers-traps.txt 文件,将程序运行结果填入。

 answers-traps.txt	2025/7/10 17:33	文本文档	10 KB
-----------------------------------------------------------------------------------------------------	-----------------	------	-------

3. 在终端中执行 make grade,得到 lab4 总分:

```
riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel/kernel.sym
make[1]: Leaving directory '/home/vale/xv6-labs-2021'
== Test answers-traps.txt == answers-traps.txt: OK
== Test backtrace test ==
$ make qemu-gdb
backtrace test: OK (1.6s)
== Test running alarmtest ==
$ make qemu-gdb
(3.5s)
== Test alarmtest: test0 ==
alarmtest: test0: OK
== Test alarmtest: test1 ==
alarmtest: test1: OK
== Test alarmtest: test2 ==
alarmtest: test2: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (66.2s)
(Old xv6.out.usertests failure log removed)
== Test time ==
time: OK
Score: 85/85
```

4. Lab4 代码提交

保存并提交到本地分支:

```
vale@Puppyyoo: /xv6-labs-2021$ git add .
vale@Puppyyoo: /xv6-labs-2021$ git commit -m "traps finished"
[traps 8b52af0] traps finished
28 files changed, 379 insertions(+), 3 deletions(-)
create mode 100644 Makefile:Zone.Identifier
create mode 100644 answers-traps.txt
create mode 100644 answers-traps.txt:Zone.Identifier
create mode 100644 kernel/Makefile:Zone.Identifier
create mode 100644 kernel/defs.h:Zone.Identifier
create mode 100644 kernel/printf.c:Zone.Identifier
create mode 100644 kernel/proc.c:Zone.Identifier
create mode 100644 kernel/proc.h:Zone.Identifier
create mode 100644 kernel/riscv.h:Zone.Identifier
create mode 100644 kernel/syscall.c:Zone.Identifier
create mode 100644 kernel/syscall.h:Zone.Identifier
create mode 100644 kernel/sysproc.c:Zone.Identifier
create mode 100644 kernel/trap.c:Zone.Identifier
create mode 100644 time.txt
create mode 100644 user/user.h:Zone.Identifier
create mode 100644 user/usys.pl:Zone.Identifier
vale@Puppyyoo: /xv6-labs-2021$
```