

Mit_Lab4: traps

20307130350 陈丹纯 信息安全 2022/11/21

RISC-V assembly

得到的main函数

```
int g(int x) {
    0: 1141          addi    sp, sp, -16
    2: e422          sd      s0, 8(sp)
    4: 0800          addi    s0, sp, 16
    return x+3;
}
    6: 250d          addiw   a0, a0, 3
    8: 6422          ld       s0, 8(sp)
    a: 0141          addi    sp, sp, 16
    c: 8082          ret

000000000000000e <f>:

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

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);
   38: 4501          li       a0, 0
   3a: 00000097      auipc   ra, 0x0
   3e: 298080e7      jalr    664(ra) # 2d2 <exit>
```

1. a0-a7 contains arguments to functions, and 13 is saved in a2.

```
24:    4635                li        a2, 13
```

2. main没有直接调用function f和g, 而是通过将f内联进main函数中, 而g被内联在f函数中。可见编译器作了优化。

3. printf is located in 64a.

```
34:    61a080e7            jalr     1562(ra) # 64a <printf>
```

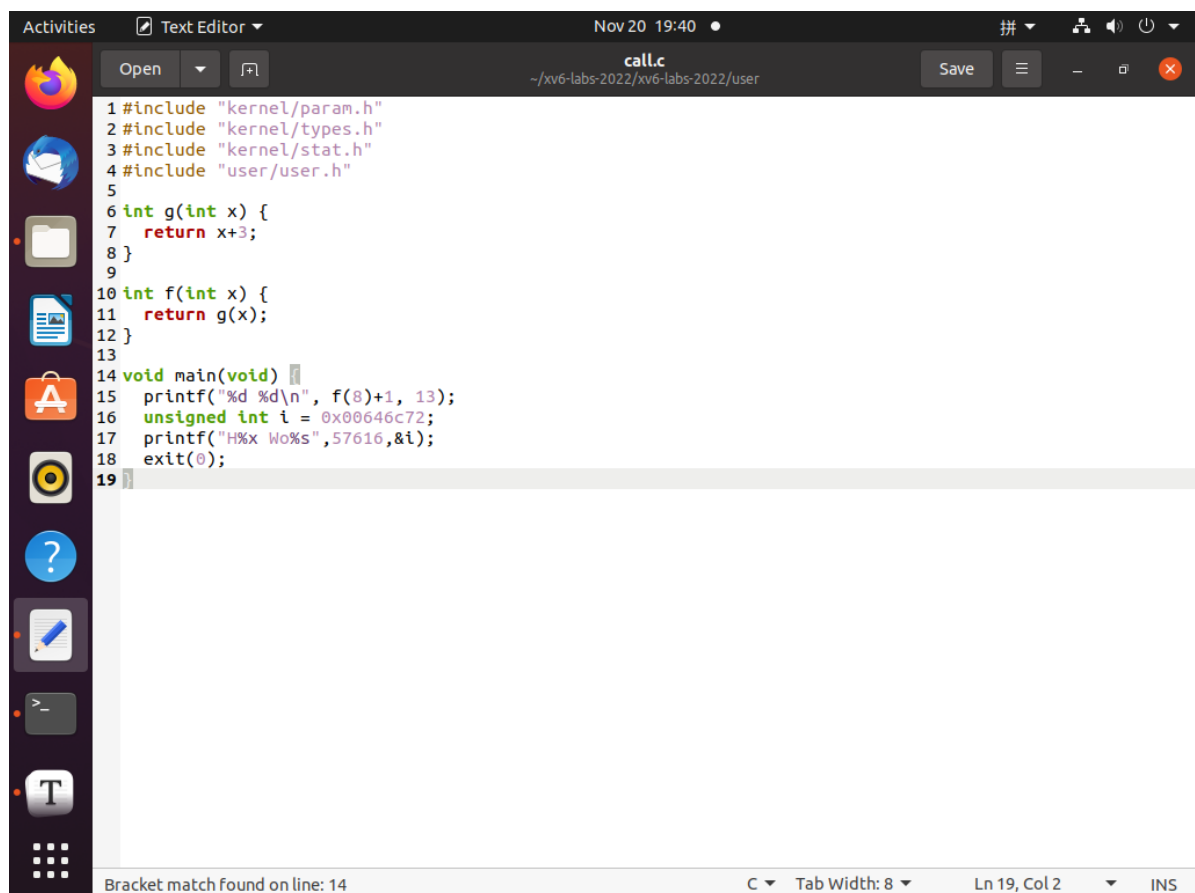
```
=0000000000000064a <printf>:
```

```
void
printf(const char *fmt, ...){ .... }
```

4. ra 的数值为 0x38.

当执行jalr跳转到printf之后, ra指向jalr指令的下一条汇编指令的地址。

5. 直接在call中运行。



```
Activities Text Editor Nov 20 19:40
call.c
~/xv6-labs-2022/xv6-labs-2022/user
Save

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     unsigned int i = 0x00646c72;
17     printf("H%x Wo%s", 57616, &i);
18     exit(0);
19 }
```

Bracket match found on line: 14 C Tab Width: 8 Ln 19, Col 2 INS

得到结果

```
$ call
12 13
HE110 World$
```

因为是小端序列, 根据ASCII tale, 可以验证如下:

```

unsigned int i = 0x00646c72;
int c1=0x64;
int c2=0x6c;
printf("%c%c\n", c1, c2);
printf("H%x Wo%s", 57616, &i);

```

得到结果：

```

init: starting sh
$ call
12 13
dl
HE110 World$

```

确实是小端序，低位字节在低地址。00 代表字符串的结束。

如果是大端序，则需要令 `i = 0x726c6400`；此外，57616是从十进制转到十六进制，并不需要改变。

6. 输出y=1；经过gdb调试，a2=1；

```

a2          0x1      1

```

Backtrace

实验思路

每个stack占一页，栈指针sp指向栈顶。每个栈中含有多个栈帧stack frame，每个stack frame中存放一个frame pointer（fp）。

根据提示：

```
ra = fp - 8;
```

```
previous fp = fp - 16;
```

根据当前fp所在page来判断是否递归完毕——use `PGROUNDOWN(fp)` (see `kernel/riscv.h`) to identify the page that a frame pointer refers to.

实验代码

1. Add the prototype for your `backtrace()` to `kernel/defs.h` so that you can invoke `backtrace` in `sys_sleep`.

```
void backtrace(void);
```

2. Add the `r_fp` function to `kernel/riscv.h`:

```

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

3. add `backtrace()` in `sys_sleep` in `kernel/sysproc.c`.

```
52 sys_sleep(void)
53 {
54     int n;
55     uint ticks0;
56     argint(0, &n);
57     if(n < 0)
58         n = 0;
59     acquire(&tickslock);
60     ticks0 = ticks;
61     while(ticks - ticks0 < n){
62         if(killed(myproc())){
63             release(&tickslock);
64             return -1;
65         }
66         sleep(&ticks, &tickslock);
67     }
68     release(&tickslock);
69     backtrace();
70     return 0;
71 }
72
```

4. Implement a `backtrace()` function in `kernel/printf.c`.

```
38 void
39 backtrace(void){
40     // pr.locking = 0;
41     printf("backtrace:\n");
42     uint64 fp=r_fp();
43     // printf("%p\n",fp);
44     //uint64 *frame = (uint64*) fp;
45     uint64 page = PGROUNDDOWN(fp);
46     // uint64 pageup = PGROUNDUP(fp);
47     // printf("%p\n",page);
48     // printf("%p\n",pageup);
49     uint64 ra;
50     while(PGROUNDDOWN(fp)==page){
51         ra = *(uint64*)(fp-8);
52         fp = *(uint64*)(fp-16);
53         printf("%p\n",ra);
54     }
55     // printf("%p\n",frame);
56 }
```

5. add `backtrace()` in `panic()` in `kernel/printf.c` to see the kernel's bt when it panics.

测试结果

```
hart 1 starting
hart 2 starting
init: starting sh
$ bttest
backtrace:
0x00000000800021ac
0x000000008000201e
0x0000000080001d14
```

```
utegan@ubuntu:~/xv6-labs-2022/xv6-labs-2022$ addr2line -e kernel/kernel
0x00000000800021ac
0x000000008000201e
0x0000000080001d14
/home/utegan/xv6-labs-2022/xv6-labs-2022/kernel/sysproc.c:70
/home/utegan/xv6-labs-2022/xv6-labs-2022/kernel/syscall.c:141
/home/utegan/xv6-labs-2022/xv6-labs-2022/kernel/trap.c:76
```

Alarm

实验思路

主要分成两个功能：

1. 实现handler的周期性调用。

test0：只需要将当前进程的trapframe的epc改成handler即可。

test1：在test0的基础上，因为考虑到调用次数，实际是题目要求的，在调用sigalarm之后周期性执行handler，不再调用sigalarm，只是周期性执行。test1测试是否有周期性执行。需要注意切换到handler时的条件。

2. trap时保存上下文。

test1/test2：陷入trap前store原来的trapframe，在sigreturn的时候restore即可。只需要在proc.h中定义一个中间trapframe变量即可。

test3：在前面的基础上，因为a0既保存了函数调用的第一个参数，也保存了 sigreturn的返回值。因此sigreturn需要返回a0，以通过test3；

实验代码

syscall的调用

1. modify the Makefile to cause `alarmtest.c` to be compiled as an xv6 user program.

```
UPROGS=\
....
$U/_alarmtest\
```

2. The right declarations to put in `user/user.h` are:

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

3. Update `user/usys.pl` (which generates `user/usys.S`), `kernel/syscall.h`, and `kernel/syscall.c` to allow `alarmtest` to invoke the `sigalarm` and `sigreturn` system calls.

`user/usys.pl`

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

`user/usys.S`

```

.global sigalarm
sigalarm:
    li a7, SYS_sigalarm
    ecall
    ret
.global sigreturn
sigreturn:
    li a7, SYS_sigreturn
    ecall
    ret

```

kernel/syscall.h

```

#define SYS_sigalarm 22
#define SYS_sigreturn 23

```

kernel/syscall.c

```

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

1 [SYS_sigalarm] sys_sigalarm,
2 [SYS_sigreturn] sys_sigreturn,
3 };

```

4. add the two syscall in `kernel/sysproc.c`. (会在后面具体test中详细修改，此时只是保证可以通过编译。)

```

uint64
sys_sigalarm(void)
{
    int ticks;
    uint64 handler;
    argint(0, &ticks);
    argaddr(1, &handler);
    return 0;
}

uint64
sys_sigreturn(void)
{
    return 0;
}

```

test0

1. Your `sys_sigalarm()` should store the alarm interval and the pointer to the handler function in new fields in the `proc` structure (in `kernel/proc.h`).
2. You'll need to keep track of how many ticks have passed since the last call (or are left until the next call) to a process's alarm handler; you'll need a new field `nticks` in `struct proc` for this too.

```
int ticks;
int nticks;
void (*handler)();
```

3. Initialize `proc` fields in `allocproc()` in `proc.c`.

```
p->ticks = 0;
p->nticks = 0;
```

4. manipulate a process's alarm ticks if there's a timer interrupt. add the situation in `kernel/trap.c`.

```
if((which_dev = devintr()) != 0){
    // ok
    if(which_dev==2){
        if(p->ticks==0 && p->handler==0){
            p->nticks = 0;
        }else{
            p->nticks++;
            if((p->nticks%(p->ticks))==0){
                p->trapframe->epc = (uint64)p->handler;
            }
        }
    }
}
```

5. pass the args to the proc in `sysproc.c`.

```
uint64
sys_sigalarm(void)
{
    struct proc *p=myproc();
    int ticks;
    uint64 handler;
    argint(0,&ticks);
    argaddr(1,&handler);
    p->handler = (void(*)()) handler;
    p->ticks = ticks;
    p->nticks=0;
    // printf("alarm:a0=%d\thandler=%d\n",p->trapframe->a0,handler);

    return 0;
}
```

test1/test2/test3

1. 在struct proc中定义一个临时变量store_trapframe来保存和恢复上下文。定义handle_existing 来作为是否正在切换中的标志来Prevent re-entrant calls to the handler----if a handler hasn't returned yet, the kernel shouldn't call it again. test2 tests this.

```
struct trapframe store_trapframe;
int handle_existing;
```

2. 在sys_sigalarm中初始化handle_existing为0。

```
p->handle_existing = 0;
```

3. 在trap.c中保存上下文并修改标识。

```
if((which_dev = devintr()) != 0){
    // ok
    if(which_dev==2){
        if(p->ticks==0 && p->handler==0){
            p->nticks = 0;
        }else{
            p->nticks++;
            if(p->handle_existing==0 && (p->nticks)%(p->ticks)==0){
                p->handle_existing=1;
                p->store_trapframe = *p->trapframe;
                p->trapframe->epc = (uint64)p->handler;
            }
        }
    }
}
```

4. 在sigreturn中恢复上下文，并返回a0 (test3) .

```
uint64
sys_sigreturn(void)
{
    struct proc *p=myproc();
    *p->trapframe = p->store_trapframe;
    p->handle_existing = 0;
    return p->trapframe->a0;
}
```

test代码整合


```

95 uint64
96 sys_sigalarm(void)
97 {
98     struct proc *p=myproc();
99     // p->a0 = p->trapframe->a0;
100     int ticks;
101     uint64 handler;
102     argint(0,&ticks);
103     argaddr(1,&handler);
104     p->handler = (void(*)()) handler;
105     p->ticks = ticks;
106     p->nticks=0;
107     p->handle_existing = 0;
108     // printf("alarm:a0=%d\thandler=%d\n",p->trapframe->a0,handler);
109
110     return 0;
111 }
112
113 uint64
114 sys_sigreturn(void)
115 {
116     struct proc *p=myproc();
117     *p->trapframe = p->store_trapframe;
118     // p->trapframe->a0 = p->a0;
119     p->handle_existing = 0;
120     // printf("return:a0=%d\n",p->trapframe->a0);
121     return p->trapframe->a0;
122 }

```

```

    int ticks;
    int nticks;
    void (*handler)();
    struct trapframe store_trapframe;
    int handle_existing;

    syscall();
} else if((which_dev = devintr()) != 0){
    // ok
    if(which_dev==2){
        if(p->ticks==0 && p->handler==0){
            p->nticks = 0;
        }else{
            p->nticks++;
            if(p->handle_existing==0 && (p->nticks)%(p->ticks)==0){
                p->handle_existing=1;
                p->store_trapframe = *p->trapframe;
                p->trapframe->epc = (uint64)p->handler;
            }
        }
    }
} else {
    printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
    printf("          sepc=%p stval=%p\n", r_sepc(), r_stval());
    printf("          ...");
}

```

测试结果

```
hart 2 starting
hart 1 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
```

```
Activities Terminal Nov 21 05:16 拼 人 响 电
utegan@ubuntu: ~/xv6-labs-2022/xv6-labs-2022

usertrap(): unexpected scause 0x000000000000000f pid=6538
sepc=0x0000000000000229e stval=0x0400000000000000
usertrap(): unexpected scause 0x000000000000000f pid=6539
sepc=0x0000000000000229e stval=0x0800000000000000
usertrap(): unexpected scause 0x000000000000000f pid=6540
sepc=0x0000000000000229e stval=0x1000000000000000
usertrap(): unexpected scause 0x000000000000000f pid=6541
sepc=0x0000000000000229e stval=0x2000000000000000
usertrap(): unexpected scause 0x000000000000000f pid=6542
sepc=0x0000000000000229e stval=0x4000000000000000
usertrap(): unexpected scause 0x000000000000000f pid=6543
sepc=0x0000000000000229e stval=0x8000000000000000
OK
test sbrkfail: usertrap(): unexpected scause 0x000000000000000d pid=6555
sepc=0x0000000000004994 stval=0x0000000000013000
OK
test sbrkarg: OK
test validatetest: OK
test bsstest: OK
test bigargtest: OK
test argptest: OK
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6563
sepc=0x0000000000002410 stval=0x0000000000010eb0
OK
test textwrite: usertrap(): unexpected scause 0x000000000000000f pid=6565
sepc=0x0000000000002490 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
$
```

实验问题及解决

1. 对stack和stack frame不甚理解，通过实验二打印出其地址即可。
2. test3()时忽略了a0的提示，一直return 0，期间试图重新定义一个a02作为中间变量。但是最终trapframe->a0的结果都会被sigreturn的返回值替代。看了提示之后才修改return结果。

实验总结

1. 引起trap的三种原因：

1. syscall
2. 中断
3. 异常

2. trap调用流程：

userspace ——> syscall ——> `ecall` ——> `uservec()` (保存上下文，切换到kernel的页表和栈,调用 `usertrap()`) ——> `usertrap()` (真正执行trap) ——> `(syscall())` ——> `usertrapret()` ——> `userret()`

kernelvec 应该是在于内核态出问题的时候调用trap机制。

3. stack和stackframe

每个stack占一页，栈指针sp指向栈顶。每个栈中含有多个栈帧stack frame，每个stack frame中存放一个frame pointer (fp)。

4. 大端小端

小端：低字节在低地址

大端：高字节在高地址

5. 实验感想：

这次实验比较硬核，需要不断深入了解trap的源码，并结合gdb调试工具才能真正理解trap机制。