Lab - Traps 目录

1. RISC-V assembly (easy)

1.1 实验目的

- 了解RISC-V汇编知识
- 掌握RISC-V汇编中函数参数传递、函数调用和内存访问的方法

1.2 实验步骤

1. 执行`make fs.img`编译`user/call.c`，阅读生成的`user/call.asm`文件，了解函数`g()`、`f()`和`main()`的汇编代码，主要内容如下：

```
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

000000000000001c <main>:

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:   7b850513            addi    a0,a0,1976 # 7e0 <malloc+0xe6>
  30:   00000097            auipc   ra,0x0
  34:   612080e7            jalr    1554(ra) # 642 <printf>
  exit(0);
  38:   4501                    li  a0,0
  3a:   00000097            auipc   ra,0x0
  3e:   28e080e7            jalr    654(ra) # 2c8 <exit>
```

```
0000000000000042 <_main>:
```

2. 回答给定的问题，包括函数参数传递的寄存器、函数调用的位置、printf函数的地址和jalr指令执行后寄存器ra中的值。
   - Q：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？

     A：`a0`,`a1`,`a2`,`a3`用于保存函数的参数，其中`a3`保留了13
   - Q：`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)

     A：`f()`、`g()`被编译器优化为了内联函数，故并没有调用`f()`、`g()`，而是在编译期就完成了计算。
   - Q：`printf`函数位于哪个地址？

     A：位于ra+1554的位置，即0x642
   - Q：在`main`中`printf`的`jalr`之后的寄存器`ra`中有什么值？

     A：ra的值是`printf()`返回到`main()`的地址0x38
   - Q：运行以下代码，指出程序的输出，输出取决于RISC-V小端存储的事实。如果RISC-V是大端存储，为了得到相同的输出，你会把`i`设置成什么？是否需要将`57616`更改为其他值？

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

     A：小端的输出结果为`He110 World`。当RISC-V为大端时为获得相同输出需要将i改为0x726c64。其中57616的值不需要改变。
   - Q：在下面的代码中，“`y=`”之后将打印什么（注：答案不是一个特定的值）？为什么会发生这种情况？

     ```
     printf("x=%d y=%d", 3);
     ```

     A：`printf()`会从`a2`寄存器中读取第三个参数作为`y`的值，因此打印出的是寄存器`a2`中存储的值，但该值是无法预估的。

1.3 实验中遇到的问题和解决方法

- **问题**：汇编代码以及研究输出结果的原因难以理解

  解决方法：查阅RISC-V的参考手册，了解字节序的知识。

1.4 实验心得

在本实验中我深入了解了RISC-V汇编中函数参数传递、函数调用和内存访问的知识。阅读xv6仓库中的`user/call.c`和`user/call.asm`文件，让我更好地理解了汇编代码是如何与C代码相互作用的。在回答问题时，我查阅了RISC-V的参考手册，了解了`jalr`指令等，了解了RISC-V的字节序是小端的，因此输出结果受到字节序的影响。通过本实验，我对RISC-V汇编有了更深入的了解，并掌握了一些调试和输出结果的技巧。这对我的学习和理解操作系统和底层硬件非常有帮助。

2. Backtrace (moderate)

2.1 实验目的

- 实现`backtrace()`函数，用于调试回溯

- 了解栈帧的结构

2.2 实验步骤

1. 在 `kernel/defs.h` 中添加 `backtrace()` 的原型，并在 `sys_sleep` 中插入对此函数的调用。

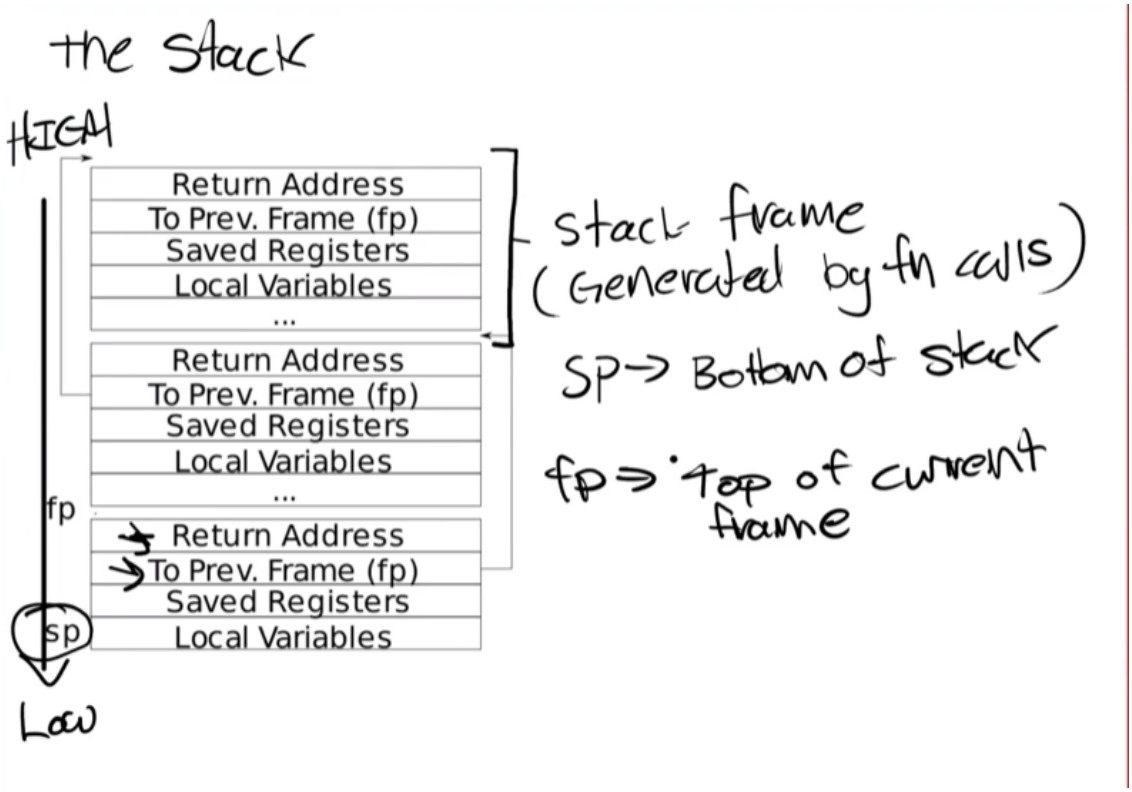
```
// printf.c
void printf(char *, ...);
void panic(char *) __attribute__((noreturn));
void printfinit(void);
void backtrace(void);
```

```
uint64 sys_sleep(void)
{
    int n;
    uint ticks0;
    backtrace();
    argint(0, &n);
    if (n < 0)
        n = 0;
    acquire(&tickslock);
    ticks0 = ticks;
    while (ticks - ticks0 < n) {
        if (killed(myproc())) {
            release(&tickslock);
            return -1;
        }
        sleep(&ticks, &tickslock);
    }
    release(&tickslock);
    return 0;
}
```

2. 将下面的函数添加到 `kernel/riscv.h`，并在 `backtrace()` 中调用此函数来读取当前的帧指针。这个函数使用内联汇编来读取 `s0`：

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

3. 明确在栈帧中，若 `FP` 寄存器的值是地址值，那么 `FP-8` 就是返回地址的值，`FP-16` 就是上一个栈指针的地址值。结构如图所示：



据此在`kernel/printf.c`中实现名为`backtrace()`的函数：

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

4. 运行`bttest`，结果如下：

```
init: starting sh
$ bttest
backtrace:
0x000000008000212e
0x0000000080002020
0x0000000080001d16
```

5. 运行`grade`测试程序，结果如下：

```
zheng-linux@zhenglinux-desktop:~/xv6-labs-2022/xv6-lab-traps$ ./grade-lab-traps backtrace
make: “kernel/kernel”已是最新。
== Test backtrace test == backtrace test: OK (1.4s)
```

6. 运行`addr2line -e kernel/kernel`，并输入刚刚回溯得到的地址，得到输出：

```
zheng-linux@zhenglinux-desktop:~/xv6-labs-2022/xv6-lab-traps$ addr2line -e kernel/kernel
0x000000008000212e
0x0000000080002020
0x0000000080001d16
/home/zheng-linux/xv6-labs-2022/xv6-lab-traps/kernel/sysproc.c:51
/home/zheng-linux/xv6-labs-2022/xv6-lab-traps/kernel/syscall.c:117
/home/zheng-linux/xv6-labs-2022/xv6-lab-traps/kernel/trap.c:76
```

7. 在`kernel/printf.c`的`panic()`中同样进行调用，在`panic()`发生时即可对内核进行回溯。

2.3 实验中遇到的问题和解决方法

- **问题**：指针越界，输出错误结果

  解决方法： 在遍历调用链时，需要识别出最后一个栈帧并停止遍历，利用页面对齐的特性，用`PGROUNDDOWN`宏对帧指针进行判断。

2.4 实验心得

在此次实验中我成功实现了`backtrace()`函数，可以获取函数调用链上每个栈帧的返回地址，并输出返回地址的列表。在其实现过程中，需要充分利用RISC-V的寄存器和内联汇编，获取当前帧指针并根据帧指针访问返回地址。通过查阅相关资料和参考手册，我对RISC-V的指令和寄存器有了更深入的了解。随后，我使用`addr2line`工具，输入返回的地址列表，成功获取了每个返回地址对应的源代码位置。这有助于调试和理解内核代码，可以更方便地定位和解决问题。同时巩固了对RISC-V汇编的理解。

3. Alarm (hard)

3.1 实验目的

- 实现`sigalarm()`系统调用
-

3.2 实验步骤

1. 明确函数功能，在进程使用CPU的时间内定期向进程发出警报，`sigalarm(interval, handler)`接收两个参数，`interval`为时间间隔，即间隔`interval`个`tick`发出警告，调用函数指针`handler`指向的函数。如果一个程序调用了`sigalarm(0, 0)`，系统应当停止生成周期性的报警调用。
2. 将`user/alarmtest.c`作为用户调用添加到`Makefile`
3. 在`user/user.h`中添加用户调用的接口声明：

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

4. 更新`user/usys.pl`、`kernel/syscall.h`和`kernel/syscall.c`以允许`alarmtest`调用`sigalarm`和`sigreturn`系统调用：

   ```
   // usys.pl
   entry("sigalarm");
   entry("sigreturn");
   ```

   ```
   // 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,
};
```

5. 在`kernel/proc.h`中的`proc`结构体中添加变量：

```
struct proc {

    ...

    int interval;                       // 间隔
    int ticks;                          // Tick数
    uint64 handler;                     // 处理函数
    struct trapframe *trapframe_saved; // 用来保存和还原寄存器状态
};
```

6. 在kernel/proc.c中的allocproc()中进行初始化：

```
static struct proc *allocproc(void)
{
    ...
found:
    ...

    p->interval = 0;
    p->ticks = 0;
    p->handler = 0;

    return p;
}
```

7. 在`kernel/sysproc.c`中添加`sys_sigalarm()`系统调用，完成参数传递：

```
uint64 sys_sigalarm(void)
{
    int interval;
    uint64 handler;
    struct proc *p = myproc();
    argint(0, &interval);
    argaddr(1, &handler);

    p->interval = interval;
    p->handler = handler;
    return 0;
}
```

8. 在`kernel/trap.c`中的`usertrap()`添加中断处理，每次中断将导致`tick`值+1，当进程的报警间隔期满时，用户进程执行处理程序函数，同时将`tick`复位，若是首次（`trapframe_saved`指向了`NULL`）进入，则要为`trapframe_saved`创建实例用于保存寄存器现场：

```
void usertrap(void)
{
    int which_dev = 0;

    ...

    // give up the CPU if this is a timer interrupt.
    if (which_dev == 2) {
        if (p->interval != 0) {
            p->ticks += 1;
            if (p->ticks == p->interval) {
                p->ticks = 0;
                if (p->trapframe_saved == 0) {
                    p->trapframe_saved = (struct trapframe *)kalloc();
                    memmove(p->trapframe_saved, p->trapframe, sizeof(*p-
>trapframe_saved));
                    p->trapframe->epc = p->handler;
                }
            }
        }
        yield();
    }

    usertrapret();
}
```

9. 在`kernel/sysproc.c`中添加`sys_sigreturn()`系统调用，用于恢复寄存器状态并释放空间。同时要注意返回寄存器`a0`的值。

```
uint64 sys_sigreturn(void)
{
    struct proc *p = myproc();
    if (p->trapframe_saved) {
        memmove(p->trapframe, p->trapframe_saved, sizeof(*p-
>trapframe_saved));
        kfree((void *)p->trapframe_saved);
        p->trapframe_saved = 0;
    }
    p->ticks = 0;
    return p->trapframe->a0;
}
```

10. 进行`alarmtest`测试，得到结果如下：

```
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
```

11. 进行`grade`测试，结果如下：

```
== Test running alarmtest == (5.1s)
== Test   alarmtest: test0 ==
  alarmtest: test0: OK
== Test   alarmtest: test1 ==
  alarmtest: test1: OK
== Test   alarmtest: test2 ==
  alarmtest: test2: OK
== Test   alarmtest: test3 ==
  alarmtest: test3: OK
```

```
zheng-linux@zhenglinux-desktop:~/xv6-labs-2022/x
make: “kernel/kernel”已是最新。
== Test usertests == usertests: OK (69.6s)
```

3.3 实验中遇到的问题和解决方法

- **问题：**无法通过`test3`，错误信息如下：

```
test3 start
test3 failed: register a0 changed
```

解决方法：题目要求`sys_sigreturn()`要返回寄存器`a0`中存储的内容，并以此来完成测试判断，因此需要添加`return p->trapframe->a0;`

3.4 实验心得。

本次实验实现了xv6内核中的`sigalarm()`和`sys_sigreturn()`系统调用，从而实现周期性的CPU时间警报功能。在实验过程中，我加深了对时钟中断、用户进程状态的保存和恢复等方面的理解。通过调试和测试`alarmtest`程序，不断验证程序正确性，并确保其输出符合预期，而在处理如时钟中断和正确保存用户进程状态的过程中，对操作系统内核的理解不断加深。