Lab 4

Part A: RISC-V assembly

1. **函数的参数包含在哪些寄存器中?例如在 main 对 printf 的调用中，哪个寄存器保存 13?**

```
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: 7d850513            addi  a0,a0,2008 # 800 <malloc+0xe8>
49     30: 00000097            auipc ra,0x0
50     34: 62a080e7            jalr  1578(ra) # 65a <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
```

参数存储在寄存器 a0-a7 中。在上述实验中，13 被存储在寄存器 a2 中。

2. **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? (Hint:编译器可能内联函数)**

如上图所示，Main 函数不对 f、g 进行调用，直接引用 f(8)+1 的结果 12。

3. **函数 printf 位于哪个地址?**

1578(ra) = 0x30 + 0x65a = 0x68a。

4. **在 jalr 到 main 中的 printf 之后，寄存器 ra 中存储的值是?**

ra 存储 return address，在以上实验中即存储 main 函数中断后代码的地址，为 0x38。

5. **运行以下代码:**

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

输出是什么?输出取决于 RISC-V 是 little-endian 的。如果 RISC-V 是 big-endian，怎样设置来产生相同的输出?是否需要更改 i，57616 为不同的值?

输出结果如下图所示，为 H(e110) Wo(rld)。其中，e110 为 57616 的十六进制表示，rld 为 i 按照 little-endian 对应的 ascii 序列：little-endian 中最低位字节存储在地址最低位处，因此与通常的阅读习惯相反，打印出的对应顺序为(72)(6c)(64)，即 rld。

```
(base) justin@chensipengdeMacBook-Pro Desktop % gcc oslab4.c
oslab4.c:5:31: warning: format specifies type 'char *' but the argument has type 'unsigned int *' [-Wformat]
    printf("H%x Wo%s", 57616, &i);
                  ~~          ^~
1 warning generated.
(base) justin@chensipengdeMacBook-Pro Desktop % ./a.out
He110 World%
```

若采用 big-endian，57616 不需要修改，而 i 应当被改为 0x726c6400。

6. **在下面的代码中，会打印出什么?(注意:答案不是特定值)为什么会发生这种情况?**

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

输出结果如下。

```
(base) justin@chensipengdeMacBook-Pro Desktop % gcc oslab4.c
oslab4.c:5:31: warning: format specifies type 'char *' but the argument has type 'unsigned int *' [-Wformat]
    printf("H%x Wo%s", 57616, &i);
                  ~~          ^~
oslab4.c:8:22: warning: more '%' conversions than data arguments [-Wformat-insufficient-args]
    printf("x=%d, y=%d", 3);
                   ~^
2 warnings generated.
(base) justin@chensipengdeMacBook-Pro Desktop % ./a.out
He110 World

x=3, y=0%
```

编译器将直接从寄存器中读取先前存储的值。

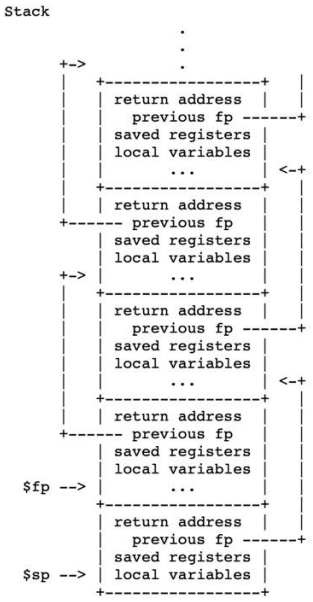
Part B: Backtrace

按照题目提示在 riscv.h 中加入 r_fp()函数，backtrace()函数通过该函数获得当前帧指针后向上回溯能够分别获得返回地址以及先前保存的帧指针。根据题目要求通过标准输出打印出返回地址，并用保存的帧指针回溯到调用函数的页帧处，之后循环上述过程，直到遍历整个堆栈。代码如下：

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

其中，通过 PGROUNDUP 和 PGROUNDDOWN 计算得到当前页的上/下限。另外，需要注意通过 fp 计算得到的返回地址与上一个帧指针均为指向地址的地址，为二级指针，因此在打印/回溯的过程中需要两次引用指针内容。完成 backtrace 的编写后，在 sys_sleep 中加入命令 backtrace()。

堆栈结构如下图所示。



最终运行结果如下。

```
xv6 kernel is booting

hart 1 starting
hart 2 starting
init: starting sh
$ bttest
backtrace:
0x00000000800021ac
0x000000008000201e
0x0000000080001d14
$ QEMU: Terminated
ubuntu@ip-172-31-13-82:~/Desktop/xv6-lab4$ addr2line -e kernel/kernel
0x00000000800021ac
/home/ubuntu/Desktop/xv6-lab4/kernel/sysproc.c:71
0x000000008000201e
/home/ubuntu/Desktop/xv6-lab4/kernel/syscall.c:141
0x0000000080001d14
/home/ubuntu/Desktop/xv6-lab4/kernel/trap.c:76
```

Part C: Alarm

实现 alarm 的关键在于如何保存/重新获取中断前的信息，我们可以通过在 proc 结构体中定义新的变量以当前进程信息。在 proc.h 中添加以下定义信息：

```
int interval;
int n;
uint64 handler;
struct trapframe *alarmframe;
uint64 return_val;
```

其中，interval 记录从上一次中断起至今的时间间隔；n 表示用户输入的中断间隔时间；handler 为中断调用函数句柄；alarmframe 为指向结构体 trapframe 的指针，用来保存中断时正在运行的进程的相关寄存器信息；return_val 保存函数返回值。以上变量均在 proc.c 中 allocproc()函数内，亦即进程被分配的时候初始化。

之后在 sysproc.c 最后添加函数 sys_sigalarm，sys_sigreturn。sigalarm 负责将用户输入

的变量从寄存器中取出并存储到 proc 结构体的相关变量中；sigreturn 部分负责将进程切换回中断之前的状态，通过 memmove 函数将存储在 alarmframe 中的寄存器信息重新载入到进程的 trapframe 中，并将计时器 interval 置零。另外，为了防止 sigreturn 更改函数原先的返回值，sigreturn 函数本身的返回值应当被设为先前存储在进程中的 return_val 变量。

```
uint64
sys_sigalarm(void){
   argint(0, &myproc()->n);
   argaddr(1, &myproc()->handler);
   return 0;
}

uint64
sys_sigreturn(void){
   //extern char alarm_frame[512];
   struct proc *p = myproc();
   memmove(p->trapframe, p->alarmframe, 512);
   //kfree(p->alarmframe);
   //memset(p->alarmframe, 0, PGSIZE);
   p->interval = 0;
   return p->return_val;
}
```

随后，在 trap.c 中实现陷入过程的实现。首先增加计时器，随后对计时器的值做判断：若已经达到中断时长，则先将当前状态下的 trapframe、函数返回值（即 a0 寄存器中存储值）存储值 alarmframe、return_val 中，然后将当前进程的 epc，即用户态下的 PC 切换为中断函数的句柄。

```
if(which_dev == 2){
      if (p->n != 0){
         //acquire(&p->lock);
         p->interval++;
         if(p->interval == p->n){
            //p->alarmframe = (struct trapframe *)kalloc();
            memmove(p->alarmframe, p->trapframe, PGSIZE);
            p->return_val = p->trapframe->a0;
            p->trapframe->epc = p->handler;
         }
         //release(&p->lock);
      }
```

完成以上内容后，xv6 可以通过 alarmtest，但运行 usertests -q 报错提示 lost some free pages，经检查发现是由于释放进程时没有同时释放进程中存储的 alarmframe。在 proc.c 中 freeproc()部分加入以下代码后可以正常通过测试。

```
if (p->alarmframe)
      kfree((void*)p->alarmframe);
   p->alarmframe = 0;
```

最终运行结果如下。

```
xv6 kernel is booting

hart 1 starting
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
test sbrklast: OK
test sbrk8000: OK
test badarg: OK
ALL TESTS PASSED
```

Thoughts

本次试验以对陷入机制的探索为主题，实现了对函数调用的回溯以及以周期性报警为主题的进程切换，加深了对进程、帧指针等概念的理解。