Lab4

姓名: 陈锐林, 学号:21307130148

2023 年 11 月 8 日

实验 1:RISCV-实验

Question-1:

(1) 根据参考资料可知，RISCV 中，函数参数保存在寄存器 a0-a7 中。(2) 如这里 main 中的参数 13，应该是保存在 a2 中；因为有这么一句”li a2,13”。

Question-2:

可以看到，因为系统内联了函数，所以在 main 的汇编代码中并没有明确调用。这里要计算 f(8)+3，能看到，这里是直接”li a1,12”，进行了赋值。

```
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
```

Question-3:

主要是下面两行代码。auipc 这行就是把 pc+(常数 »12) 给 ra；下一行就是跳转到地址为 1554+ra 的地方，即 printf。所以这里 printf 的地址就是这个结果 0x0000000000000642。

```
30: 00000097          auipc ra,0x0
34: 612080e7          jalr  1554(ra) # 642 <printf>
```

Question-4:

ra 应该保存函数返回地址，即 38。

Question-5:

这里 57616 即 0xe110，%x 即直接以 16 进制输出。(1) 按小端执行，i 的内部储存为”0x726c6400”，% 输出字符串，再根据 ASCII 码表，每两位一输出；最后结果是”He110,World\0”。(2) 如果改成大端法表示，57616 无需修改，因为还是 16 进制输出；而需要对 i 进行翻转，unsigned int i = 0x726c6400。

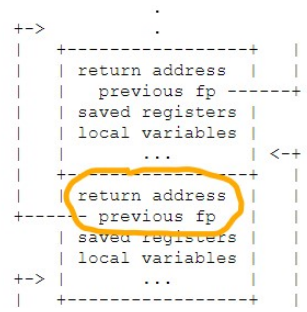
Question-6:

多次运行，会看到输出的值不同。联系 1 中内容可以知道，这时候 printf 需要两个参数，第一个参数给出为 3，但是第二个未知；这时候应该直接输出 a2 中的值。

实验 2:Backtrace 实验

1. 思路如下：

由参考资料的这个图能看出来，函数的调用栈是这么个结构 (可递归自底向上)。假设我们知道最底部的 fp，那么首先可以根据 fp-8 得到这个函数的返回地址 ra；还能知道上一个函数栈的 fp 储存在地址为 fp-16 的地方。于是可以递归打印。



2. 具体实现：

(1) 首先是前期准备工作：包括在 kernel/risv.h 中添加函数 r_fp(); 在 kernel/defs.h 中需添加 backtrace() 的函数声明; 在 panic 和 sleep 中添加函数 backtrace() 的调用。

(2) 重点实现：在 kernel/printf.c 中实现 backtrace()。根据思路对 fp 进行递归，*(fp-8) 是 ra，*(fp-16) 是下一个 fp；循环结束条件用到了提示中给出的 PGROUNDUP(fp)，因为栈是往上递归的。最后代码如下：

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

3. 测试截图：

```
$ bttest
backtrace:
0x0000000800021f4
0x0000000080002066
0x0000000080001d22
$ QEMU: Terminated
chenrl959@SK-20210701MSSI:~/xv6-labs-2022$ addr2line -e kernel/kernel
0x0000000800021f4
/home/chenrl959/xv6-labs-2022/kernel/sysproc.c:71
0x0000000080002066
/home/chenrl959/xv6-labs-2022/kernel/syscall.c:145
0x0000000080001d22
/home/chenrl959/xv6-labs-2022/kernel/trap.c:76
```

实验 3-1:Alarm 实验初步实现

1. 思路如下：

第一个部分里，sigreturn 先直接返回 0；重点在于 sigalarm。修改后的 proc 应该能储存时间间隔、调用警报函数、已过去多少 tick。这里重点在于对 trap 中 usertrap 的修改和 sigalarm 的实现。根据提示，在 usertrap 中，若 tick==interval，就清空 tick 并且跳转到目的函数，方法是修改 trapframe->epc。

2. 具体实现如下：

(1) 完成准备工作，如添加系统调用、添加声明、完成简单的 sigreturn 函数 (直接返回 0)。

(2) 完成对 proc 结构的修改，这里主要是间隔 interval，函数指针 handler，已过时间 tick；并对 proc.c 中的 allocproc 进行修改，都初始化为 0。

(3) 完成对 usertrap 的修改，判 interval 是否为 0→ 判 tick 是不是达到标准 → 修改 p->trapframe->epc 为 handler。下面贴出代码，注释掉的是后面的。

(4) 定义 sigalarm，主要功能是通过 argint argaddr 读入参数，然后修改当前进程的 interval/handler/tick。这之后已经能完成 test0 了，主要代码如下：

```c
if(which_dev == 2){
  if(p->interval)
  {
    if(p->tick == p->interval)
    {
      p->tick = 0;
      //p->tfcopy = p->trapframe + 512;
      //memmove(p->tfcopy,p->trapframe,sizeof
      p->trapframe->epc = (uint64)p->handler;
    }
    p->tick++;
  }
  yield();
}
```

(a) in usertrap

```c
sys_sigalarm(void)
{
  int interval;
  uint64 handler;
  struct proc * p;
  argint(0,&interval);
  argaddr(1,&handler);
  p = myproc();
  p->interval = interval;
  p->handler = handler;
  p->tick = 0;
  return 0;
}
```

(b) sigalarm

实验 3-2:Alarm 实验进一步实现

1. 思路如下：

这里主要解决俩个问题，一个是寄存器的恢复与保存，还要避免重复 alarm。第一个问题，我想到的解决方法是在改变 epc 前对信息进行备份，之后在 sigreturn 中拷贝回来即可，这需要 proc 中的补上新的字段，tfcopy。第二个问题，我想到的简易解决是从 tick 出发；处于前面的设置，当且仅当 tick==interval 时才会触发，那么在触发后就让 tick 不置 0，而是持续变大。

2. 具体实现：

(1) 完成 struct proc 和函数 allocproc 中的添加。

(2) 修改 usertrap，删去 tick=0，并且在 p->trapframe->epc 更新前，将 p->trapframe 的内容拷贝给 p->tfcopy(p->tfcopy 初始被置为 p->trapframe+512，(512? 凑整 + 大于 trapframe 的 288 大小))。

(3) 修改 sigreturn，对 p->tfcopy 做判断，如果它是 p->trapframe+512，那就进行恢复。最后要将 tick，tfcopy 置 0。最后为了完成 test3 中对 a0 不能更改的要求，要返回 p->trapframe->a0。代码如下：

```
if(which_dev == 2){
  if(p->interval)
  {
    if(p->tick == p->interval)
    {
      p->tfcopy = p->trapframe + 512;
      memmove(p->tfcopy,p->trapframe,
              sizeof(struct trapframe));
      p->trapframe->epc = (uint64)p->handler;
    }
    p->tick++;
  }
  yield();
}
```

(a) in usertrap

```
sys_sigreturn(void) {
  struct proc* p = myproc();
    if(p->tfcopy != p->trapframe + 512) {
        return -1;
    }
    memmove(p->trapframe, p->tfcopy,
            sizeof(struct trapframe));
    p->tick = 0;
    p->tfcopy = 0;
    return p->trapframe->a0;
}
```

(b) sigalarm

3. 测试截图：

```
== Test backtrace test ==
$ make qemu-gdb
backtrace test: OK (4.0s)
== Test running alarmtest ==
$ make qemu-gdb
(7.0s)
== Test   alarmtest: test0 ==
  alarmtest: test0: OK
== Test   alarmtest: test1 ==
  alarmtest: test1: OK
== Test   alarmtest: test2 ==
  alarmtest: test2: OK
== Test   alarmtest: test3 ==
  alarmtest: test3: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (90.0s)
```