

Lab4

20307130340 杨孟颖 计算机

一、回答问题

- (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？
函数的参数保存在寄存器 `a0-a7` 中。根据汇编代码，
24: 4635 `li a2,13`
13 存在 `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）
没有对 `f` 和 `g` 的调用。因为 `f` 是 `main` 的内联函数而 `g` 是 `f` 的内联函数。
- (3) 函数 `printf` 位于哪个地址？
根据汇编代码：000000000000064a `printf`, `printf` 储存在 000000000000064a 中。
- (4) 在 `jalr` 到 `main` 中的 `printf` 之后，寄存器 `ra` 中存储的值是？寄存器类型：
`scause`: 发生 `trap` 的类型。
根据汇编代码：
34: 61a080e7 `jalr 1562(ra) # 64a <printf>`
`exit(0);`
38: 4501 `li a0,0`
存储的内容是 `0x38`，即执行完 `jalr` 之后执行的下一行代码。
- (5) 运行代码输出：

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

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 `i57616` 为不同的值？

不需要更改。`57616` 的十六进制表示是 `110`，十六进制没有端序要求。

(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);`

输出值是受之前运行代码影响的随机值。由于 `printf` 需要输出的数据数目比实际赋值的数目少，因此在执行命令时只将 `a1` 赋值为 3，`a2` 没有进行赋值，还保存着上一次使用 `a2`。

`Scratch`: `traplime` 的地址

`Sepc`: 当前 `pc` 的值。从用户切换为 `kernel` 会改变 `pc`

`Stvec`: `trap handler` 的地址。

`Csrrw`: 交换两个寄存器中的值

二、Backtrace

`fp` 指向当前栈帧的开始地址，`sp` 指向当前栈帧的结束地址。（栈从高地址往低地址生

长, 所以 fp 虽然是帧开始地址, 但是地址比 sp 高) 栈帧中从高到低第一个 8 字节 fp-8 是 return address, 也就是当前调用层应该返回到的地址。栈帧中从高到低第二个 8 字节 fp-16 是 previous address, 指向上一层栈帧的 fp 开始地址。剩下的为保存的寄存器、局部变量等。一个栈帧的大小不固定, 但是至少 16 字节。

在 xv6 中, 使用一个页来存储栈, 如果 fp 已经到达栈页的上界, 则说明已经到达栈底。

首先进行尝试, 根据提示得知用 r_fp() 函数获得当前执行的函数的帧指针, 想要向上遍历 stack 并在每个 stack frame 中打印保存的返回地址按照提示的要求需要偏移 8 个地址和 16 个地址, 于是通过 printf("%p\n", frame[-1]); printf("%p\n", frame[-2]); 打印相邻地址。

```
void backtrace(){
    uint64 fp = r_fp();
    uint64 * frame = (uint64 *)fp;
    printf("%p\n", fp);
    printf("%p\n", frame[-1]);
    printf("%p\n", frame[-2]);
}
```

经测试可以成功打印保存的返回地址。下一步需要检验是否满足取值范围在合法范围内。

```
void backtrace(){
    uint64 fp = r_fp();
    uint64 * frame = (uint64 *)fp;
    // printf("%p\n", fp);
    // printf("%p\n", frame[-1]);
    // printf("%p\n", frame[-2]);

    uint64 up = PGROUNDUP(fp);
    uint64 down = PGROUNDDOWN(fp);
    while(fp < up && fp > down){
        printf("%p\n", frame[-1]);
        fp = frame[-2];
        frame = (uint64 *)fp;
    }
}
```

经过检验可以通过这种方式实现。

```
$ bttest
0x000000008000219c
0x000000008000201e
0x0000000080001d14
```

```
yangmy@ubuntu:~/xv6-labs-2022$ addr2line -e kernel/kernel
0x000000008000219c
0x000000008000201e
0x0000000080001d14
/home/yangmy/xv6-labs-2022/kernel/sysproc.c:70
/home/yangmy/xv6-labs-2022/kernel/syscall.c:141
/home/yangmy/xv6-labs-2022/kernel/trap.c:76
```

```

void
panic(char *s)
{
    pr.locking = 0;
    printf("panic: ");
    printf(s);
    printf("\n");
    panicked = 1; // freeze uart output from other CPUs
    backtrace();
    for(;;)
        ;
}

```

三、Alarm

```

int ticks;
uint64 handler;
int ticks_epc;
int ticks_cnt;

```

在 `proc.c` 文件的 `static struct proc* allocproc(void)` 函数中对被初始化的 `proc` 进行赋值:

```

p->ticks = 0;

```

在 `trap.c` 文件中: `void usertrap(void)` 函数中:

```

if(which_dev == 2){
    if(p->ticks > 0){
        p->ticks_cnt++;
        if(p->ticks_cnt < p->ticks){
            p->ticks_cnt = 0;
            p->trapframe->epc = p->handler;
        }
    }
    yield();
}

```

进行完这一步的更改, 可以实现 `test0`.

在进行 `test1` 时, 发现由于在 `sigalarm` 函数中改变了结构体的 `handler` 值, 所以应当在下一步进行之前将其回复。根据 `alarmtest` 中的函数:

```

periodic()
{
    count = count + 1;
    printf("alarm!\n");
    sigreturn();
}

```

在 `sigreturn()` 将原先的值回复。如果仅进行: `p->trapframe->epc = p->ticks_epc;` 则会出错。原因是在进行完 `p->trapframe->epc = p->handler;` 改变了 `p->trapframe->epc` 的值之后, 系统有执行了很多其他函数, 所以其他寄存器的值都有变化。只还原一个寄存器中的就会导致结果错误。因此采取使用函数的方法储存和恢复寄存器中的值。

在 `trap.c` 中添加:

```

void store(){

```

```

struct proc *p = myproc();
p->ra = p->trapframe->ra;
p->sp = p->trapframe->sp;
p->gp = p->trapframe->gp;
p->tp = p->trapframe->tp;
p->t0 = p->trapframe->t0;
p->t1 = p->trapframe->t1;
p->t2 = p->trapframe->t2;
p->s0 = p->trapframe->s0;
p->s1 = p->trapframe->s1;
p->a0 = p->trapframe->a0;
p->a1 = p->trapframe->a1;
p->a2 = p->trapframe->a2;
p->a3 = p->trapframe->a3;
p->a4 = p->trapframe->a4;
p->a5 = p->trapframe->a5;
p->a6 = p->trapframe->a6;
p->a7 = p->trapframe->a7;
p->s2 = p->trapframe->s2;
p->s3 = p->trapframe->s3;
p->s4 = p->trapframe->s4;
p->s5 = p->trapframe->s5;
p->s6 = p->trapframe->s6;
p->s7 = p->trapframe->s7;
p->s8 = p->trapframe->s8;
p->s9 = p->trapframe->s9;
p->s10 = p->trapframe->s10;
p->s11 = p->trapframe->s11;
p->t3 = p->trapframe->t3;
p->t4 = p->trapframe->t4;
p->t5 = p->trapframe->t5;
p->t6 = p->trapframe->t6;
}

```

在 sysproc.c 中添加:

```

void restore(){
    struct proc *p = myproc();
    p->trapframe->ra = p->ra;
    p->trapframe->sp = p->sp;
    p->trapframe->gp = p->gp;
    p->trapframe->tp = p->tp;
    p->trapframe->t0 = p->t0;
    p->trapframe->t1 = p->t1;
    p->trapframe->t2 = p->t2;
    p->trapframe->s0 = p->s0;

```

```

p->trapframe->s1 = p->s1;
p->trapframe->a0 = p->a0;
p->trapframe->a1 = p->a1;
p->trapframe->a2 = p->a2;
p->trapframe->a3 = p->a3;
p->trapframe->a4 = p->a4;
p->trapframe->a5 = p->a5;
p->trapframe->a6 = p->a6;
p->trapframe->a7 = p->a7;
p->trapframe->s2 = p->s2;
p->trapframe->s3 = p->s3;
p->trapframe->s4 = p->s4;
p->trapframe->s5 = p->s5;
p->trapframe->s6 = p->s6;
p->trapframe->s7 = p->s7;
p->trapframe->s8 = p->s8;
p->trapframe->s9 = p->s9;
p->trapframe->s10 = p->s10;
p->trapframe->s11 = p->s11;
p->trapframe->t3 = p->t3;
p->trapframe->t4 = p->t4;
p->trapframe->t5 = p->t5;
p->trapframe->t6 = p->t6;
}

```

```

uint64 sys_sigalarm(void){
    int ticks;//0
    uint64 handler;//1

    argint(0, &ticks);

    argaddr(1, &handler);

    struct proc * p = myproc();
    p->storea0 = p->trapframe->a0;
    p->ticks = ticks;
    p->handler = handler;
    return 0;
}

```

```

uint64 sys_sigreturn(void){
    struct proc *p = myproc();
    p->trapframe->epc = p->ticks_epc;
    restore();
}

```

```

    p->sreturn = 0;
    //p->trapframe->a0 = 0;
    return 0;
}

```

以上步骤操作完成之后可以成功通过 `test2`。

下面进行 `test3`。根据提示，`test3` 需要保证 `sigreturn` 不改变 `a0` 寄存器的值。刚开始的想法是寻找 `sigreturn` 执行之后紧接着执行的函数，在紧接着执行的函数里用 `restore` 将寄存器中的参数恢复。但是 `sigreturn` 执行之后的函数不确定，因此查找有关函数返回修改寄存器值的方式。发现是函数通过返回值修改寄存器值，于是将 `sigreturn` 的返回值定为原先的 `a0` 寄存器的值 `p->trapframe->a0`，从而实现不改变寄存器 `a0` 的值。

```

uint64 sys_sigreturn(void){
    struct proc *p = myproc();
    p->trapframe->epc = p->ticks_epc;
    restore();
    p->sreturn = 0;
    //p->trapframe->a0 = 0;
    return p->trapframe->a0;
}

```

修改返回之后的系统可以成功通过 `test3` 和 `usertests -q`。

```

$ 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

```

[illegible]

```

usertrap(): unexpected scause 0x000000000000000f pid=6541
          sepc=0x000000000000229e stval=0x1000000000000000
usertrap(): unexpected scause 0x000000000000000f pid=6542
          sepc=0x000000000000229e stval=0x2000000000000000
usertrap(): unexpected scause 0x000000000000000f pid=6543
          sepc=0x000000000000229e stval=0x4000000000000000
usertrap(): unexpected scause 0x000000000000000f pid=6544
          sepc=0x000000000000229e stval=0x8000000000000000
OK
test sbrkfail: usertrap(): unexpected scause 0x000000000000000d pid=6556
              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=6564
                sepc=0x0000000000002410 stval=0x0000000000010eb0
OK
test textwrite: usertrap(): unexpected scause 0x000000000000000f pid=6566
                sepc=0x0000000000002490 stval=0x0000000000000000
OK
test pgbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=6569
               sepc=0x0000000000005c5e stval=0x0000000000005c5e
usertrap(): unexpected scause 0x000000000000000c pid=6570
               sepc=0x0000000000005c5e stval=0x0000000000005c5e
OK
test sbrklast: OK
test sbrk8000: OK
test badarg: OK
ALL TESTS PASSED

```

四、实验中遇到的问题及解决方法

在一开始实现 `sig_return()` 中的恢复寄存器值时，只恢复了 `p->trapframe->epc = p->ticks_epc`；但是这样实验结果是错误的，原因是在改变 `p->trapframe->epc` 的值之后又进行了别的操作，寄存器的其他值也相应改变，只改回 `p->trapframe->epc` 的值会导致与其他寄存器的值不匹配。