lab traps

lab 实现

exercise 0

xv6的trap分为两种，从用户陷入和从内核陷入，在`kernel/trap.c`中分别有对应的`usertrap`函数和`kerneltrap`函数。这里主要讲一下从用户函数陷入内核处理trap的流程。当用户程序需要进行一个trap，RISC-V硬件首先会执行一系列操作，主要是将pc复制到sepc，设置sstatus和scause，将stvec复制到pc，然后程序会从`uservec`开始执行。`uservec`以及`userret`都存储在进程的trampoline page中，每一个进程的trampoline都映射在地址TRAMPOLINE，为了使在`uservec`和`userret`中即使页表切换了程序也能继续执行，在内核页中也同样映射了trampoline page在相同的地址。`uservec`保存了用户程序的寄存器在进程的trapframe中，从中读出一些内核所需的寄存器值，并跳转到`usertrap`，在`usertrap`根据sstatus和scause中的值执行对应的trap处理函数，然后执行`usertrapret`，`usertrapret`会重新恢复一些寄存器的值，在trapframe中记录内核的一些值便于用户程序再次陷入trap。最后跳转到`userret`恢复用户程序上下文，完成一次trap。xv6在内核中进行trap会简单许多，不需要保存和恢复上下文，并且对于非设备中断直接panic。

RISC-V assembly

1. a0-a7存储函数参数，对于`main()`中的13，由a2存储。

```
printf("%d %d\n", f(8)+1, 13);
24:   4635                    li  a2,13
26:   45b1                    li  a1,12
28:   00000517                auipc   a0,0x0
2c:   7c050513                addi    a0,a0,1984 # 7e8 <malloc+0xea>
```

2. `main()`没有直接调用`f()`，直接将f(8)+1的值12写入a1，`f()`中也没有直接调用`g()`，直接将a0的值+3。
3. `printf()`的地址在0x0000000000000640

```
0000000000000640 <printf>:
```

4. 调用`printf()`时ra的值为0x30，而在从`printf()`返回后的第一条指令ra的值为0x38，即为从`printf()`发返回后下一条指令的地址?

```
Breakpoint 2, 0x0000000000000034 in main () at user/call.c:15
15          printf("%d %d\n", f(8)+1, 13);
(gdb) p $ra
$1 = (void (*)()) 0x30 <main+20>
(gdb) b *0x38
Breakpoint 3 at 0x38: file user/call.c, line 17.
(gdb) c
Continuing.

Breakpoint 3, main () at user/call.c:17
17          exit(0);
(gdb) p $ra
$2 = (void (*)()) 0x38 <main+28>
```

5. 输出为HE110 World

   由于是大端机器i应该为0x726c6400。57676不用改变，不论是在大端机器还是在小段机器都是用十六进制的格式去读十进制的57676

6. 当指定了y=%d的值，在调用`printf()`之前会将x=%d和y=%d参数依次写入a1, a2。

```
printf("x=%d y=%d", 3, 4);
24:   4611                    li  a2,4
26:   458d                    li  a1,3
28:   00000517                auipc   a0,0x0
2c:   7c050513                addi    a0,a0,1984 # 7e8 <malloc+0xea>
30:   00000097                auipc   ra,0x0
34:   610080e7                jalr    1552(ra) # 640 <printf>
```

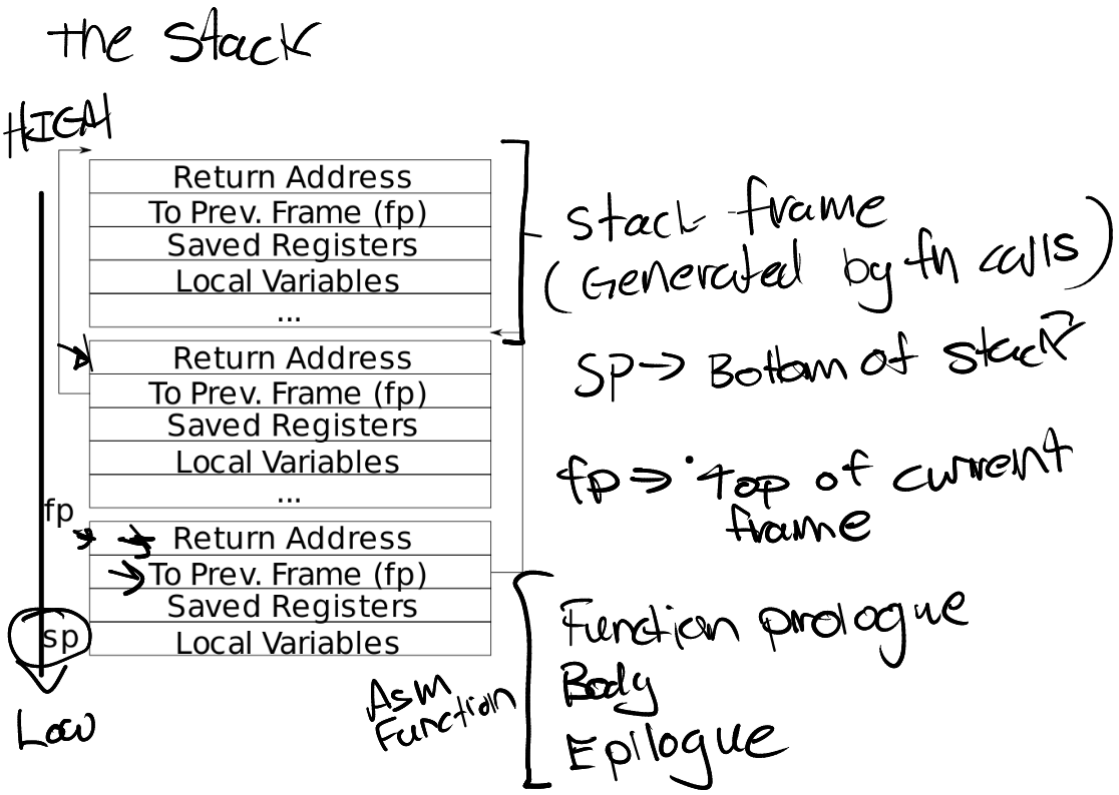
由此猜想当没有指定y=%d的值时会输出a2的值，经过验证确实如此。

```
(gdb) b *0x32
Breakpoint 2 at 0x32: file user/call.c, line 15.
(gdb) p $a2
$1 = 5237
```

```
init: starting sh
$ call
x=3 y=5237
```

exercise backtrace

通过栈帧逐个打印函数调用的返回地址，首先要弄清楚内存中栈帧的layout，在给出的一个lecture note中有直观的展示。



根据layout对于每一个存储在s0寄存器中的栈帧地址fp，fp-8为返回地址的地址，fp-16为前一个栈帧的地址，一层层往回找即可。由于给进程分配的栈为一个PGSIZE，所以利用`PGROUNDUP(fp)`定位到栈的最高位地址为边界。

```
void backtrace(void){
  uint64 fp = r_fp();
  uint64 stack_top = PGROUNDUP(fp);

  printf("backtrace:\n");
  while(fp < stack_top){
    printf("%p\n", *(uint64*)(fp-8)); // return address
    fp = *(uint64*)(fp-16);
  }
}
```

代码比较简单，利用了`r_fp()`中的inline-assembly去读取s0中的值并返回。

exercise alarm

test 0

在第一步中我们需要设置好`sigalarm`和`sigreturn`的大体框架，并使通过trap进程的handler能在固定ticks后运行。按照提示的步骤

1. 在`makefile`中添加`alarmtest.c`使之能进行编译。
2. 在`user/user.h`中添加

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

   使用户态程序能调用操作系统提供的`sigalarm`和`sigreturn`接口
3. 完善以上两个函数的系统调用部分，和**lab syscall**一样在`user/usys.pl`增加entry函数，在`kernel/syscall.h`中添加编号，在`kernel/syscall.c`中加入函数表。
4. 在`kernel/proc.h`的`struct proc`中添加新的部分并在`kernel/proc.c`相应部分进行初始化。

   ```
     int tick_num;
     int per_tick_num;
     uint64 handler_adr;
   ```

5. 在`kernel/sysproc.c`中添加具体的系统调用函数。`sys_sigreturn`暂时只返回0。

   ```
   uint64 sys_sigalarm(void){
     int per_tick_num;
     uint64 handler_adr;

     if(argint(0, &per_tick_num) < 0){
       return -1;
     }
     if(argaddr(1, &handler_adr) < 0){
       return -1;
     }

     struct proc* p = myproc();
     p->per_tick_num = per_tick_num;
     p->handler_adr = handler_adr;
     p->handler_on = 1;
   ```

```
  return 0;

}

uint64 sys_sigreturn(void){
  return 0;
}
```

6. 在`kernel/proc.h`的`struct proc`中添加

```
  int tick_num;
  int per_tick_num;
  uint64 handler_adr;
```

7. 由于每一个时钟周期都会进行时钟中断，所以只要在`kernel/trap.c`的`trap.c`中增加对于时钟中断的处理即可。考虑到中断返回时会将`epc`中的地址写入`pc`实现中断指令的恢复，所以在进程的tick达到周期数时将handler的地址写入`p->trapframe`即可。

```
  if(which_dev == 2){
    p->tick_num++;
    if(p->tick_num == p->per_tick_num){
        p->trapframe->epc = p->handler_adr;
    }

    yield();
  }
```

由此即可通过alarmtest的test0，但是test1会fail.，其原因为当`p->handler_adr`写入了`p->trapframe->epc`原有的epc值丢失，在执行完handler会进程无法继续执行中断后的指令，这就需要接下来的步骤捏。

```
$ alarmtest
test0 start
.....................alarm!
test0 passed
test1 start
.alarm!
.alarm!
.alarm!
..alarm!
.alarm!
..alarm!
.alarm!
.alarm!
.alarm!
.alarm!

test1 failed: foo() executed fewer times than it was called

test1 failed: foo() executed fewer times than it was called
```

test1/test2

由于需要在执行完handler后恢复进程在执行handler之前的运行状态，所以在usertrap覆盖`p->trapframe->epc`之前需要保存`p->trapframe`中用户态相关的寄存器（内核态是相对于整个进程而言的，就算在handler中陷入内核对内核栈等进行了修改也不用将状态恢复到调用handler之前，所以不用保存），然后再`sigreturn`后恢复保存的寄存器即可。

`usertrap`中修改后如下

```
  if(which_dev == 2){
    p->tick_num++;
    // printf("tick: %d\n", p->tick_num);
    if(p->tick_num == p->per_tick_num){
        if(!p->in_handler){

          p->handler_frame->epc = p->trapframe->epc;
          // printf("ra: %p\n", p->trapframe->ra);

          uint64 start = (uint64)p->handler_frame + 8, end = start + 256,
                 tf = (uint64)p->trapframe + 40;
          // printf("store!\n");
          while(start < end){
            *(uint64*)start = *(uint64*)tf;
            start += 8;
            tf += 8;
          }
          // printf("ticks!\n");
          // printf("ra: %p\n", p->handler_frame->ra);
          p->trapframe->epc = p->handler_adr;
          p->in_handler = 1;
        }
        p->tick_num = 0;
    }

    yield();
  }
```

这里还需要考虑一个问题，即在时钟周期时可能handler仍在执行，所以在proc中增加了`in_handler`变量标识此时钟周期时是否该进程是否有handler在运行，在没有运行时才修改epc保存寄存器。

对`sigreturn`的修改如下，主要是恢复保存的寄存器和将`in_handler`重新设置为0。

```
uint64 sys_sigreturn(void){
  struct proc* p = myproc();
  p->trapframe->epc = p->handler_frame->epc;

  uint64 start = (uint64)p->handler_frame + 8, end = start + 256,
         tf = (uint64)p->trapframe + 40;
  while(start < end){
    *(uint64*)tf = *(uint64*)start;
    tf += 8;
    start += 8;
  }
  p->in_handler = 0;

  // printf("return\n");
  // printf("ra: %p\n", p->trapframe->ra);
  return 0;
}
```

为了通过`usertest`测试我们还需要在`kernel/proc.c`中的`allocproc`和`freeproc`中对新分配的保存寄存器的页进行分配和回收内存的操作。这样就可以通过所有测试了捏。

```
== Test answers-traps.txt == answers-traps.txt: OK
== Test backtrace test ==
$ make qemu-gdb
backtrace test: OK (5.3s)
== Test running alarmtest ==
$ make qemu-gdb
(4.8s)
== Test   alarmtest: test0 ==
  alarmtest: test0: OK
== Test   alarmtest: test1 ==
  alarmtest: test1: OK
== Test   alarmtest: test2 ==
  alarmtest: test2: OK
== Test usertests ==
$ make qemu-gdb
usertests: OK (129.6s)
== Test time ==
time: OK
Score: 85/85
```