Lab4: traps 实验报告

- 有三种事件会导致中央处理器搁置普通指令的执行，并强制将控制权转移到处理该事件的特殊代码上：
  1. 系统调用，当用户程序执行 `ecall` 指令要求内核为其做些什么时
  2. 异常：（用户或内核）指令做了一些非法的事情，例如除以零或使用无效的虚拟地址
  3. 设备中断，一个设备，例如当磁盘硬件完成读或写请求时，向系统表明它需要被关注
  - xv6实验使用陷阱（trap）作为这些情况的通用术语
  - 陷阱是透明的——也就是说，陷阱发生时正在执行的任何代码都需要稍后恢复，并且不需要意识到发生了任何特殊的事情
- 本实验探索如何使用陷阱实现系统调用。首先使用栈做一个热身练习，然后实现一个用户级陷阱处理的示例。
  1. RISC-V 汇编，编译user/call.c文件，在user/call.asm中生成程序的可读汇编版本。阅读call.asm中函数g、f和main的代码并回答问题。
  2. 添加backtrace功能，打印出调用栈，用于调试。
  3. 在xv6中添加alarm功能，使得进程在使用CPU时定期向其发出警报。

---

1. 首先切换分支：

```
$ git fetch
$ git checkout traps
$ make clean
```

RISC-V assembly (easy)

执行`make fs.img`编译项目，并阅读*call.asm*中函数`g`、`f`和`main`的代码，回答以下问题（将答案存储在*answers-traps.txt*文件中）：

1. 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？

   a0-a7 存储了函数调用的参数; 13 被存在了a2寄存器中

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(提示：编译器可能会将函数内联)

   没有这样的代码。 g 被内联到 f 中，然后 f 又被进一步内联到 main() 中。所以看到的不是函数跳转，而是优化后的内联函数

3. `printf`函数位于哪个地址？

   ```
   30: 00000097            auipc ra,0x0
   34: 5e6080e7            jalr  1510(ra) # 616 <printf>
   ```

   位于`0X0000000000000616`地址中：ra=pc=0x30=48，
   1510(ra)=1510+48=1558=0x616

4. 在`main`中`printf`的`jalr`之后的寄存器`ra`中有什么值？

   0x38(ra=pc+4)

5. 运行以下代码。

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

   程序的输出是什么？

`%x` 是按 16 进制输出，`57616=0xe110`，所以输出的前半段是 `He110`；ASCII 码中 0x64 对应 `d`，0x6c 对应 `l`，0x72 对应 `r`， RISC-V 为小端存储（低地址存 0x72），且 `%s` 从低地址开始读取数据输出，所以会输出 `rld`，于是输出的后半段就是 `World`。

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?

> - 小端：较高的有效字节存放在较高的的存储器地址，较低的有效字节存放在较低的存储器地址。
> - 大端：较高的有效字节存放在较低的存储器地址，较低的有效字节存放在较高的存储器地址。

如果是大端存储，将 `i` 设置为 `0x726c6400`； `57616` 不需要更改

6. 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?

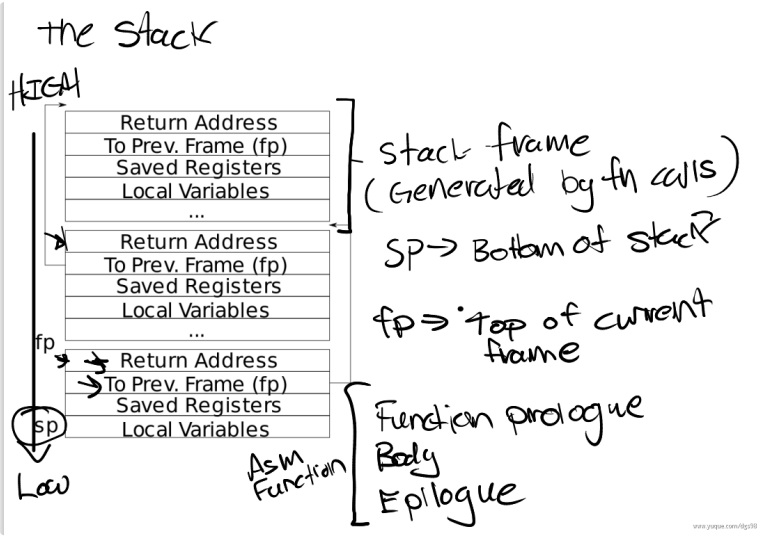
```
printf("x=%d y=%d", 3);
```

此时，printf尝试读的参数数量比提供的参数数量多：printf的format字符串储存在寄存器a0中，第二个参数 3 储存在寄存器 a1 中，想要读取的第三个参数储存在a2寄存器中，所以 y 的输出依赖于a2 寄存器的值。

Backtrace(moderate)

在*kernel/printf.c*中实现名为 `backtrace()` 的函数，用于在出错时输出这之前栈中的函数调用。编译器会在每个栈帧中存入一个帧指针，指向调用者的帧指针。`backtrace()` 应该用这些帧指针来遍历栈并输出每个栈帧的保存的返回地址。

- 这个 课堂笔记中有张栈帧布局图。注意返回地址位于栈帧帧指针的固定偏移(-8)位置，并且保存的帧指针位于帧指针的固定偏移(-16)位置



1. 在 *kernel/defs.h* 中添加 backtrace 的声明：

```
79  // printf.c
80  void            printf(char*, ...);
81  void            panic(char*) __attribute__((noreturn));
82  void            printfinit(void);
83  void            backtrace(void);
```

2. 在 *kernel/riscv.h* 中添加读取帧指针 fp 的方法 r_fp，GCC编译器将当前正在执行的函数的帧指针保存在 `s0` 寄存器

```
322 // read the current frame pointer
323 static inline uint64
324 r_fp()
325 {
326   uint64 x;
327   asm volatile("mv %0, s0" : "=r" (x) );
328   return x;
329 }
```

3. 在 *kernel/printf.c* 中添加 backtrace 函数，调用r_fp函数来读取当前的帧指针，循环打印每个栈帧中保存的返回地址

> fp 指向当前栈帧的开始地址，sp为栈指针。fp-8 存放返回地址，fp-16 存放原栈帧（调用函数的fp）

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

4. 在 *kernel/sysproc.c* 的 sys_sleep 函数和*kernel/printf.c* 的 panic 函数中添加对 backtrace 函数的调用

```
uint64
sys_sleep(void)
{
  int n;
  uint ticks0;

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

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

5. 执行测试命令：`bttest`

```
$ bttest
backtrace:
0x0000000080002dd2
0x0000000080002c2e
0x0000000080002918
```

运行 `addr2line -e kernel/kernel`，并将上面的地址剪切粘贴，得到输出

```
220110512@comp4:~/xv6-labs-2020$ addr2line -e kernel/kernel
0x0000000080002dd2
/home/students/220110512/xv6-labs-2020/kernel/sysproc.c:74
0x0000000080002c2e
/home/students/220110512/xv6-labs-2020/kernel/syscall.c:140 (discriminator 1)
0x0000000080002918
/home/students/220110512/xv6-labs-2020/kernel/trap.c:76
```

通过

Alarm(Hard)

给 xv6 加一个功能——**在进程使用CPU时间时定期发出警告**。这对于限制 CPU 密集型（计算密集型）进程的占用时间，或对于在计算过程中有其他定期动作的进程可能很有用。更普遍的来说，你将实现用户级中断/故障处理程序的一种初级形式。

我们需要添加一个 `sigalarm(interval, handler)` 系统调用。如果一个应用调用了 `sigalarm(n, fn)`，则该应用每耗时 `n` 个 ticks，内核应该使之调用 `fn`，`fn` 返回后应用应当在它离开的地方恢复执行。如果一个应用调用 `sigalarm(0, 0)`，内核应该停止产生 alarm calls。

test0: invoke handler(调用处理程序)

1. 在*makeflie*添加alarmtest用户程序

```
161 UPROGS=\
162     $U/_cat\
163     $U/_echo\
164     $U/_forktest\
165     $U/_grep\
166     $U/_init\
167     $U/_kill\
168     $U/_ln\
169     $U/_ls\
170     $U/_mkdir\
171     $U/_rm\
172     $U/_sh\
173     $U/_stressfs\
174     $U/_usertests\
175     $U/_grind\
176     $U/_wc\
177     $U/_zombie\
178     $U/_alarmtest\
```

2. 在 *user/user.h* 中添加 `sigalarm` 和 `sigreturn` 的声明

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

3. 在 *user/usys.pl* 中添加 `sigalarm` 和 `sigreturn` 的 entry

```
39 entry("sigalarm");
40 entry("sigreturn");
```

4. 在 *kernel/syscall.h* 中添加 `sigalarm` 和 `sigreturn` 的系统调用号

```
23 #define SYS_sigalarm  22
24 #define SYS_sigreturn 23
```

5. 在 *kernel/syscall.c* 中添加系统调用的声明和系统调用号到对应系统调用的映射

```
107 extern uint64 sys_sigalarm(void);
108 extern uint64 sys_sigreturn(void);
```

```
132 [SYS_sigalarm]  sys_sigalarm,
133 [SYS_sigreturn] sys_sigreturn,
```

6. 在 *kernel/proc.h* 的 proc 结构体中增加警报相关属性

```
  int interval;                 // 时间间隔
  uint64 handler;               // 调用的函数
  int ticks;                    // 经过的时钟数
};
```

7. 在 *kernel/sysproc.c* 中添加系统调用函数的实现

   - sys_sigaram：将报警间隔和指向处理程序函数的指针存储在`struct proc`的新字段中
   - sys_sigreturn：此时只返回0

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

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

8. 在 *kernel/proc.c* 中的 allocproc 函数中初始化

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

9. 在 *kernel/proc.c* 中的 freeproc 函数中，在进程结束后释放

```
156     p->interval = 0;
157     p->handler = 0;
158     p->ticks = 0;
159   }
```

10. 在 *kernel/trap.c* 中的 usertrap 函数中，添加时钟中断时相应的处理代码：

> 函数在返回时，调用 ret 指令，使用 trapframe 内事先保存的寄存器的值进行恢复，epc决定了用户空间代码恢复执行的指令地址，将其修改为处理函数的入口便能够在trap返回时，直接执行alarm处理函数

```
79    // give up the CPU if this is a timer interrupt.
80    if(which_dev == 2) {
81      if(p->interval) {
82        if(p->ticks == p->interval) {
83          // 该进程已经用完了它的时间片，需要重新设置它的时间片
84          p->ticks = 0;  // 待会儿需要删掉这一行
85          p->trapframe->epc = p->handler;
86        }
87        p->ticks++;
88      }
89      // 此时调用yield函数，将CPU的控制权交给其他进程。
90      yield();
91    }
92    usertrapret();
93  }
```

11. 执行测试命令： `alarmtest`

```
220110512@comp4:~/xv6-labs-2020$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive file=fs.img,if=none,format
=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ alarmtest
test0 start
.............alarm!
test0 passed
test1 start
...alarm!
.alarm!
```

test0通过

test1/test2(): resume interrupted code(恢复被中断的代码)

test0 的代码里存在一个问题：`p->trapframe->epc` 被覆盖后无法恢复。改正方法也很简单：事先复制一份，在 sigreturn 系统调用时恢复它。除了epc 以外，由于处理程序还有可能更改寄存器，因此我们将整个 `trapframe` 复制下来用于寄存器的保存。

1. 修改 *kernel/proc.h* 的 proc 结构体：新增 `st_trapframe` 字段储存中断时的 trapframe，用于中断处理结束后恢复原程序

```
  int interval;                // 时间间隔
  uint64 handler;              // 调用的函数
  int ticks;                   // 经过的时钟数
  struct trapframe *st_trapframe; // 保存的寄存器，用于中断后恢复原程序
};
```

2. 在 *kernel/sysproc.c* 中 sys_sigreturn 函数中还原 trapframe。

```
uint64
sys_sigreturn(void)
{
  struct proc *p = myproc();
  acquire(&p->lock);
  *p->trapframe = *p->st_trapframe;
  p->ticks = 0;
  release(&p->lock);
  return 0;
}
```

3. 修改 *kernel/trap.c* 中的 usertrap 函数：在中断时保存寄存器状态至 `st_trapframe` 中

```
  // give up the CPU if this is a timer interrupt.
  if(which_dev == 2) {
    if(p->interval) {
      if(p->ticks == p->interval) {
        // 该进程已经用完了它的时间片，需要重新设置它的时间片
        // p->ticks = 0;
        *p->st_trapframe = *p->trapframe;
        p->trapframe->epc = p->handler;
      }
      p->ticks++;
    }
    // 此时调用yield函数，将CPU的控制权交给其他进程。
    yield();
  }
  usertrapret();
}
```

4. 在 *kernel/proc.c* 中的 allocproc 函数中初始化

```
  p->interval = 0;
  p->handler = 0;
  p->ticks = 0;
  if((p->st_trapframe = (struct trapframe *)kalloc()) == 0){
    release(&p->lock);
    return 0;
  }
  return p;
}
```

5. 在 *kernel/proc.c* 中的 freeproc 函数中，在进程结束后释放指针

```
  p->interval = 0;
  p->handler = 0;
  p->ticks = 0;
  if(p->st_trapframe)
    kfree((void*)p->st_trapframe);
}
```

> 需要注意的是，如果有一个handler函数正在执行，就不能让第二个handler函数继续执行。为此，可以再次添加一个字段，用于标记是否有 handler 在执行。但事实上有些多余，可以直接取消 trap.c 中的 ticks 置 0 操作，在 sys_sigreturn 函数中设置ticks为0，这样即使第一个 handler 还没执行完，由于 ticks 一直是递增的，第二个 handler 始终无法执行。

6. 执行测试命令：`alarmtest`

```
220110512@comp4:~/xv6-labs-2020$ make qemu
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nogra
phic -drive file=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=
virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 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
```

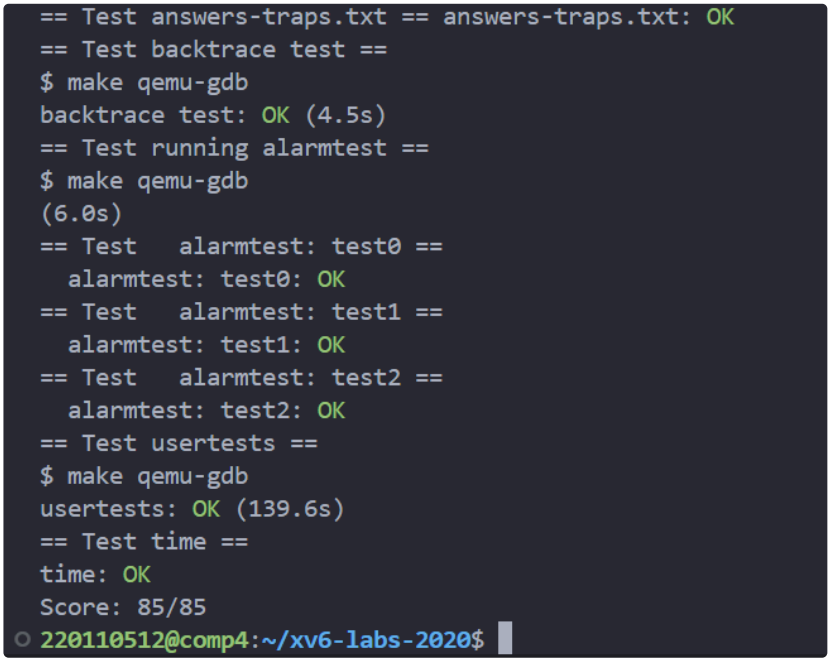
执行测试命令：`usertests`

```
test exitiput: OK
test iput: OK
test mem: OK
test pipe1: OK
test preempt: kill... wait... OK
test exitwait: OK
test rmdot: OK
test fourteen: OK
test bigfile: OK
test dirfile: OK
test iref: OK
test forktest: OK
test bigdir: OK
ALL TESTS PASSED
$
```

test1/2通过

结果截图

执行命令`make grade`


== Test answers-traps.txt == answers-traps.txt: OK
== Test backtrace test ==
$ make qemu-gdb
backtrace test: OK (4.5s)
== Test running alarmtest ==
$ make qemu-gdb
(6.0s)
== 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 (139.6s)
== Test time ==
time: OK
Score: 85/85
220110512@comp4:~/xv6-labs-2020$