

Lab4 Traps

Lab4 Traps

前置知识

实验内容

RISC-V assembly (easy)

任务

生成汇编代码

阅读 `call.asm`

运行指定代码

Backtrace (moderate)

任务

添加 `backtrace()` 函数原型

在 `kernel/printf.c` 中添加函数

在 `kernel/sysproc.c` 中的 `sys_sleep` 函数中调用

编译并运行 `bttest`

使用 `addr2line` 命令查看代码位置

Alarm (hard)

任务

修改 `struct proc`

实现两个函数

修改 `usertrap()` 函数

修改 `Makefile` 和添加系统调用

`Makefile`

`user/user.h`

更新 `user/usys.pl`

更新 `kernel/syscall.h` 和 `kernel/syscall.c`

测试成功

涉及到的文件

实验得分

前置知识

实验内容

RISC-V assembly (easy)

任务

- 理解RISC-V汇编代码。
- 分析函数调用中的寄存器使用情况。
- 理解little-endian和big-endian的差异及其在编程中的影响。

生成汇编代码

在你的xv6仓库中，`user/call.c` 文件已经存在。运行以下命令：

```
1 make fs.img
```

这个命令会编译代码并生成程序的可读汇编版本，文件位于 `user/call.asm`。

阅读`call.asm`

```
1 void main(void) {
2     1c:  1141                addi    sp,sp,-16
3     1e:  e406                sd     ra,8(sp)
4     20:  e022                sd     s0,0(sp)
5     22:  0800                addi    s0,sp,16
6     printf("%d %d\n", f(8)+1, 13);
7     24:  4635                li     a2,13
8     26:  45b1                li     a1,12
9     28:  00000517            auipc   a0,0x0
10    2c:  7b850513            addi    a0,a0,1976 # 7e0
        <malloc+0xe6>
11    30:  00000097            auipc   ra,0x0
```

```

12 34: 612080e7      jalr    1554(ra) # 642 <printf>
13  exit(0);
14 38: 4501          li     a0,0
15 3a: 00000097      auipc   ra,0x0
16 3e: 28e080e7      jalr    654(ra) # 2c8 <exit>

```

- Q1: Which registers contain arguments to functions? For example, which register holds 13 in main's call to `printf`?
- A: 通过阅读 `call.asm` 文件中的 `main` 函数可知，调用 `printf` 函数时，13 被寄存器 `a2` 保存。 `a1`, `a2`, `a3` 等通用寄存器；13 被寄存器 `a2` 保存。

```

1  int g(int x) {
2      0: 1141          addi    sp,sp,-16
3      2: e422          sd     s0,8(sp)
4      4: 0800          addi    s0,sp,16
5      return x+3;
6  }
7      6: 250d          addiw   a0,a0,3
8      8: 6422          ld     s0,8(sp)
9      a: 0141          addi    sp,sp,16
10     c: 8082          ret
11
12 0000000000000000e <f>:
13
14  int f(int x) {
15     e: 1141          addi    sp,sp,-16
16    10: e422          sd     s0,8(sp)
17    12: 0800          addi    s0,sp,16
18    return g(x);
19  }
20    14: 250d          addiw   a0,a0,3
21    16: 6422          ld     s0,8(sp)
22    18: 0141          addi    sp,sp,16
23    1a: 8082          ret

```

- Q2: Where is the call to function `f` in the assembly code for main? Where is the call to `g`? (Hint: the compiler may inline functions.)

- 通过阅读函数 `f` 和 `g` 得知：函数 `f` 调用函数 `g`；函数 `g` 使传入的参数加 3 后返回。
- 所以总结来说，函数 `f` 就是使传入的参数加 3 后返回。考虑到编译器会进行内联优化，这就意味着一些显而易见的，编译时可以计算的数据会在编译时得出结果，而不是进行函数调用。

查看 `main` 函数可以发现，`printf` 中包含了一个对 `f` 的调用。

```
1 printf("%d %d\n", f(8)+1, 13);
2 24: 4635                li a2,13
3 26: 45b1                li a1,12
```

- 但是对应的汇编代码却是直接将 `f(8)+1` 替换为 `12`。这就说明编译器对这个函数调用进行了优化，所以对于 `main` 函数的汇编代码来说，其并没有调用函数 `f` 和 `g`，而是在运行之前由编译器对其进行了计算。
- A: `main` 的汇编代码没有调用 `f` 和 `g` 函数。编译器对其进行了优化。

```
1 void
2 printf(const char *fmt, ...)
3 {
4 642: 711d                addi    sp,sp,-96
5 644: ec06                sd     ra,24(sp)
6 646: e822                sd     s0,16(sp)
7 648: 1000                addi    s0,sp,32
8 64a: e40c                sd     a1,8(s0)
9 64c: e810                sd     a2,16(s0)
10 64e: ec14                sd     a3,24(s0)
11 650: f018                sd     a4,32(s0)
12 652: f41c                sd     a5,40(s0)
13 654: 03043823            sd     a6,48(s0)
14 658: 03143c23            sd     a7,56(s0)
15     va_list ap;
16
17     va_start(ap, fmt);
18 65c: 00840613            addi    a2,s0,8
19 660: fec43423            sd     a2,-24(s0)
20     vprintf(1, fmt, ap);
21 664: 85aa                mv     a1,a0
22 666: 4505                li     a0,1
23 668: 00000097            auipc   ra,0x0
24 66c: dce080e7            jalr    -562(ra) # 436 <vprintf>
```

- Q3: At what address is the function `printf` located?
- A: `0x642`
- `auipc` 和 `jalr` 的配合，可以跳转到任意 32 位的地址。

```
1 30: 00000097      auipc    ra,0x0
2 34: 612080e7      jalr     1554(ra) # 642 <printf>
```

- 第 49 行，使用 `auipc ra,0x0` 将当前程序计数器 `pc` 的值存入 `ra` 中。
- 第 50 行，`jalr 1554(ra)` 跳转到偏移地址 `printf` 处，也就是 `0x642` 的位置。
- 根据 [reference1](#) 中的信息，在执行完这句命令之后，寄存器 `ra` 的值设置为 `pc + 4`，也就是 `return address` 返回地址 `0x38`。
- Q4: What value is in the register `ra` just after the `jalr` to `printf` in `main`?
- A: `0x38`

运行指定代码

```
1 #include <stdio.h>
2
3 int main() {
4     unsigned int i = 0x00646c72;
5     printf("H%x Wo%s\n", 57616, (char*)&i);
6     return 0;
7 }
8
```

- 观察输出:

```
chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023
Merge made by the 'ort' strategy.
 docs/Lab4 Traps.md | 143 +++++
 1 file changed, 143 insertions(+)
 create mode 100644 docs/Lab4 Traps.md
chengyu@chengyu-virtual-machine:~/os-labs/xv6-labs-2023$ git add -A
chengyu@chengyu-virtual-machine:~/os-labs/xv6-labs-2023$ git commit -m 'add images'
[traps 1fc9f8c] add images
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 docs/img/test.png
chengyu@chengyu-virtual-machine:~/os-labs/xv6-labs-2023$ git push origin
枚举对象中: 13, 完成.
对象计数中: 100% (13/13), 完成.
使用 4 个线程进行压缩
压缩对象中: 100% (9/9), 完成.
写入对象中: 100% (10/10), 38.26 KiB | 2.55 MiB/s, 完成.
总共 10 (差异 3), 复用 0 (差异 0), 包复用 0
remote: Resolving deltas: 100% (3/3), completed with 1 local object.
To https://github.com/kiy-00/xv6-labs-2023.git
 0225fea..1fc9f8c traps -> traps
chengyu@chengyu-virtual-machine:~/os-labs/xv6-labs-2023$ gcc -o test test.c
chengyu@chengyu-virtual-machine:~/os-labs/xv6-labs-2023$ ./test
He110 World
chengyu@chengyu-virtual-machine:~/os-labs/xv6-labs-2023$
```

1 He110 world

- 首先，57616 转换为 16 进制为 e110，所以格式化描述符 %x 打印出了它的 16 进制值。

其次，如果在小端（little-endian）处理器中，数据 0x00646c72 的高字节存储在内存的高位，那么从内存低位，也就是低字节开始读取，对应的 ASCII 字符为 rld。

如果在 大端（big-endian）处理器中，数据 0x00646c72 的高字节存储在内存的低位，那么从内存低位，也就是高字节开始读取其 ASCII 码为 dlr。

所以如果大端序和小端序输出相同的内容 i，那么在其为大端序的时候，i 的值应该为 0x726c64，这样才能保证从内存低位读取时的输出为 rld。

无论 57616 在大端序还是小端序，它的二进制值都为 e110。大端序和小端序只是改变了多字节数据在内存中的存放方式，并不改变其真正的值的大小，所以 57616 始终打印为二进制 e110。

- Q5: The output depends on that fact that the RISC-V is little-endian. If the RISC-V were instead big-endian what would you set i to in order to yield the same output? Would you need to change 57616 to a different value?

- A: 如果在大端序，`i` 的值应该为 `0x726c64` 才能保证与小端序输出的内容相同。不用该变 `57616` 的值。
- Q6: In the following code, what is going to be printed after 'y='? (note: the answer is not a specific value.) Why does this happen?

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

- 通过之前的章节可知，函数的参数是通过寄存器 `a1`, `a2` 等来传递。如果 `printf` 少传递一个参数，那么其仍会从一个确定的寄存器中读取其想要的参数值，但是我们并没有给出这个确定的参数并将其存储在寄存器中，所以函数将从此寄存器中获取到一个随机的不确定的值作为其参数。故而此例中，`y=` 后面的值我们不能够确定，它是一个垃圾值。
- A: `y=` 之后的值为一个不确定的垃圾值。

Backtrace (moderate)

任务

- 实现一个 `backtrace()` 函数，该函数用于调试时输出函数调用栈的回溯信息。这有助于理解程序在出现错误时的执行路径。我们将通过遍历栈帧并打印每个栈帧中的保存的返回地址来实现这个功能。

添加 `backtrace()` 函数原型

- 首先，在 `kernel/defs.h` 文件中添加 `backtrace()` 函数的原型声明，以便在其他地方调用它：

```
1 // printf.c
2 void          printf(char*, ...);
3 void          panic(char*) __attribute__((noreturn));
4 void          printfinit(void);
5 void          backtrace(void);
```

- GCC 编译器将当前正在执行的函数的帧指针（frame pointer）存储到寄存器 `s0` 中。在 `kernel/riscv.h` 中添加以下代码：


```

1 static inline uint64
2 r_fp()
3 {
4     uint64 x;
5     asm volatile("mv %0, s0" : "=r" (x) );
6     return x;
7 }

```

- 在 `backtrace` 中调用此函数，将会读取当前帧指针。`r_fp()` 使用内联汇编读取 `s0`。

在 `kernel/printf.c` 中添加函数

```

1 void
2 backtrace(void)
3 {
4     uint64 fp_address = r_fp();
5     while(fp_address != PGROUNDDOWN(fp_address)) {
6         printf("%p\n", *(uint64*)(fp_address-8));
7         fp_address = *(uint64*)(fp_address - 16);
8     }
9 }

```

- `PGROUNDDOWN(fp)` 总是表示 `fp` 所在的这一页的起始位置。

在 `kernel/sysproc.c` 中的 `sys_sleep` 函数中调用

```

1 void sys_sleep(void){
2     ...
3     backtrace();
4     ...
5 }

```

编译并运行 `bttest`

```
chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023

riscv64-linux-gnu-objdump -S user/_grind > user/grind.asm
riscv64-linux-gnu-objdump -t user/_grind | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > user/grind.sym
mkfs/mkfs fs.img README user/_cat user/_echo user/_forktest user/_grep user/_in
it user/_kill user/_ln user/_ls user/_mkdir user/_rm user/_sh user/_stressfs use
r/_usertests user/_grind user/_wc user/_zombie user/_call user/_bttest
nmeta 46 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 1) blocks 19
54 total 2000
ballocc: first 811 blocks have been allocated
ballocc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp
3 -nographic -global virtio-mmio.force-legacy=false -drive file=fs.img,if=none,f
ormat=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
$ bttest
0x0000000008000220a
0x0000000008000207c
0x00000000080001d72
$
```

使用 `addr2line` 命令查看代码位置

```
chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023

riscv64-linux-gnu-objdump -t kernel/kernel | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > kernel/kernel.sym
riscv64-linux-gnu-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_TRAPS -DLAB_TRAPS -MD -mmodel=medany -ffreestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -n
o-pie -c -o user/usertests.o user/usertests.c
riscv64-linux-gnu-ld -z max-page-size=4096 -T user/user.ld -o user/usertests user/usertests.o user/ulib.o user/usys.o user/printf.o user/umalloc.o
riscv64-linux-gnu-objdump -S user/usertests > user/usertests.asm
riscv64-linux-gnu-objdump -t user/_grind | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > user/grind.sym
riscv64-linux-gnu-gcc -Wall -Werror -O -fno-omit-frame-pointer -ggdb -gdwarf-2 -DSOL_TRAPS -DLAB_TRAPS -MD -mmodel=medany -ffreestanding -fno-common -nostdlib -mno-relax -I. -fno-stack-protector -fno-pie -n
o-pie -c -o user/grind.o user/grind.c
riscv64-linux-gnu-ld -z max-page-size=4096 -T user/user.ld -o user/_grind user/grind.o user/ulib.o user/usys.o user/printf.o user/umalloc.o
riscv64-linux-gnu-objdump -S user/_grind > user/grind.asm
riscv64-linux-gnu-objdump -t user/_grind | sed '1,/SYMBOL TABLE/d; s/ .* / /; /^$/d' > user/grind.sym
mkfs/mkfs fs.img README user/_cat user/_echo user/_forktest user/_grep user/_init user/_kill user/_ln user/_ls user/_mkdir user/_rm user/_sh user/_stressfs user/_usertests user/_grind user/_wc user/_zombie
user/_call user/_bttest
nmeta 46 (boot, super, log blocks 30 inode blocks 13, bitmap blocks 1) blocks 1954 total 2000
ballocc: first 811 blocks have been allocated
ballocc: write bitmap block at sector 45
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -global virtio-mmio.force-legacy=false -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
$ bttest
0x0000000008000220a
0x0000000008000207c
0x00000000080001d72
$ QEMU: Terminated
chengyu@chengyu-virtual-machine:~/os-labs/xv6-labs-2023$ addr2line -e kernel/kernel
0x00000000080002de2
0x00000000080002f4a
0x00000000080002bfc/home/chengyu/os-labs/xv6-labs-2023/kernel/fs.c:462
/home/chengyu/os-labs/xv6-labs-2023/kernel/fs.c:522
/home/chengyu/os-labs/xv6-labs-2023/kernel/fs.c:315
chengyu@chengyu-virtual-machine:~/os-labs/xv6-labs-2023$ addr2line -e kernel/kernel
0x00000000080002de2
0x00000000080002f4a
0x00000000080002bfc/home/chengyu/os-labs/xv6-labs-2023/kernel/fs.c:462
/home/chengyu/os-labs/xv6-labs-2023/kernel/fs.c:522
/home/chengyu/os-labs/xv6-labs-2023/kernel/fs.c:315
chengyu@chengyu-virtual-machine:~/os-labs/xv6-labs-2023$
```

Alarm (hard)

任务

- 在这个实验中，你将为 `xv6` 添加一个功能，定期向进程发出 CPU 时间警告。这个功能可以帮助计算密集型进程限制其使用的 CPU 时间，或者让进程在计算时也能执行一些周期性操作。本质上，你将实现一种原始的用户级中断/故障处理程序。实验的目的是让 `alarmtest` 通过，并且 `usertests -q` 成功运行。

修改 `struct proc`

- 首先在 `kernel/proc.h` 中的 `proc` 结构体中添加需要的内容。

```
1 struct proc {
2     ...
3     // the virtual address of alarm handler function in user page
4     uint64 handler_va;
5     int alarm_interval;
6     int passed_ticks;
7     // save registers so that we can re-store it when return to
    interrupted code.
8     struct trapframe saved_trapframe;
9     // the bool value which show that is or not we have returned
    from alarm handler.
10    int have_return;
11    ...
12 }
```

实现两个函数

- 在 `kernel/sysproc.c` 中实现 `sys_sigalarm` 和 `sys_sigreturn` :

```
1 uint64
2 sys_sigreturn(void)
3 {
4     struct proc* proc = myproc();
5     // re-store trapframe so that it can return to the interrupt
    code before.
6     *proc->trapframe = proc->savd_trapframe;
```

```

7   proc->have_return = 1; // true
8   return proc->trapframe->a0;
9 }
10
11 uint64
12 sys_sigalarm(void)
13 {
14     int ticks;
15     uint64 handler_va;
16
17     argint(0, &ticks);
18     argaddr(1, &handler_va);
19     struct proc* proc = myproc();
20     proc->alarm_interval = ticks;
21     proc->handler_va = handler_va;
22     proc->have_return = 1; // true
23     return 0;
24 }

```

- 注意到一点，`sys_sigreturn(void)` 的返回值不是 0，而是 `proc->trapframe->a0`。这是因为我们想要完整的恢复所有寄存器的值，包括 `a0`。但是一个系统调用返回的时候，它会将其返回值存到 `a0` 寄存器中，那这样就改变了之前 `a0` 的值。所以，我们干脆让其返回之前想要恢复的 `a0` 的值，那这样在其返回之后 `a0` 的值仍没有改变。

修改 `usertrap()` 函数

- 然后修改 `kernel/trap.c` 中的 `usertrap` 函数。

```

1 void
2 usertrap(void) {
3     ...
4     // give up the CPU if this is a timer interrupt.
5     if(which_dev == 2) {
6         struct proc *proc = myproc();
7         // if proc->alarm_interval is not zero
8         // and alarm handler is returned.
9         if (proc->alarm_interval && proc->have_return) {
10             if (++proc->passed_ticks == 2) {
11                 proc->saved_trapframe = *p->trapframe;

```

```

12         // it will make cpu jmp to the handler function
13         proc->trapframe->epc = proc->handler_va;
14         // reset it
15         proc->passed_ticks = 0;
16         // Prevent re-entrant calls to the handler
17         proc->have_return = 0;
18     }
19 }
20 yield();
21 }
22 ...
23 }

```

- 从内核跳转到用户空间中的 **alarm handler** 函数的关键一点就是：修改 **epc** 的值，使 **trap** 在返回的时候将 **pc** 值修改为该 **alarm handler** 函数的地址。这样，我们就完成了从内核调转到用户空间中的 **alarm handler** 函数。但是同时，我们也需要保存之前寄存器栈帧，因为后来 **alarm handler** 调用系统调用 **sys_sigreturn** 时会破坏之前保存的寄存器栈帧(**p->trapframe**)。

修改 **Makefile** 和添加系统调用

Makefile

```

1  UPROGS=\
2      $U/_cat\
3      $U/_echo\
4      $U/_forktest\
5      $U/_grep\
6      $U/_init\
7      $U/_kill\
8      $U/_ln\
9      $U/_ls\
10     $U/_mkdir\
11     $U/_rm\
12     $U/_sh\
13     $U/_stressfs\
14     $U/_usertests\
15     $U/_grind\
16     $U/_wc\
17     $U/_zombie\

```

user/user.h

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

更新 user/usys.pl

- 在 `usys.pl` 文件中，添加 `sigalarm` 和 `sigreturn` 的系统调用入口，以便生成 `usys.S`。

```
1 entry("sigalarm");
2 entry("sigreturn");
```

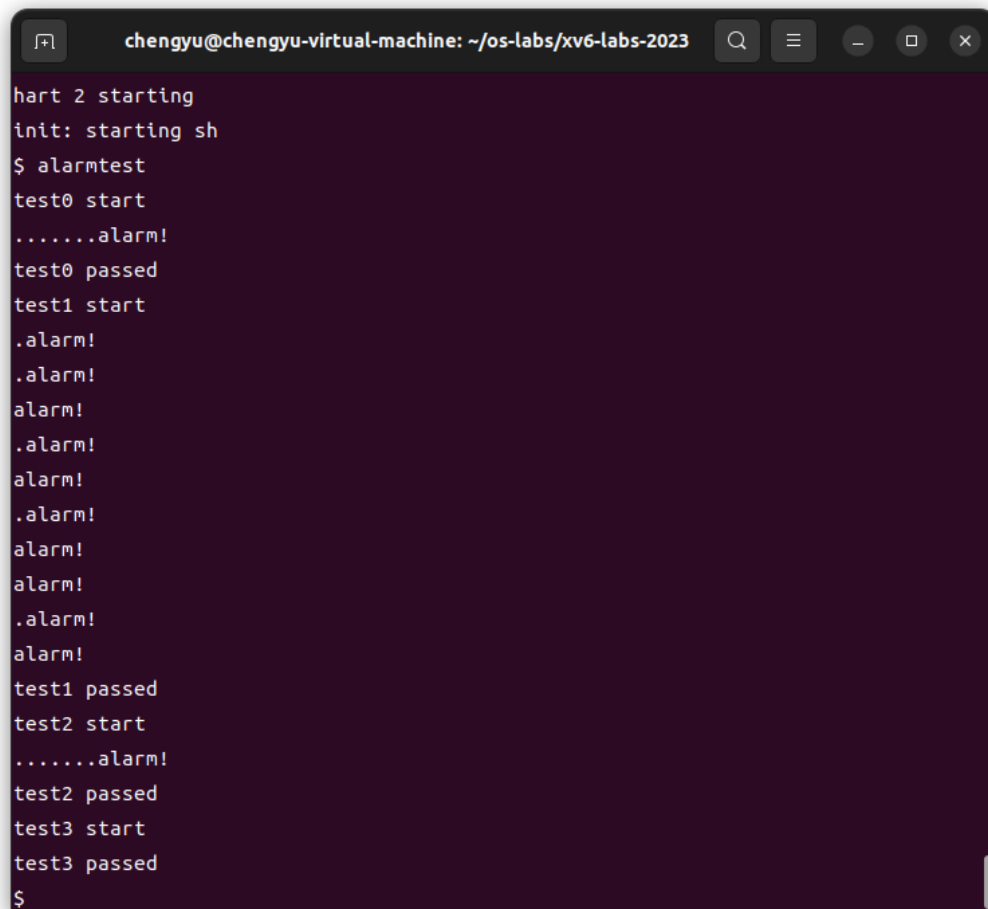
更新 kernel/syscall.h 和 kernel/syscall.c

- 在 `syscall.h` 中添加 `SYS_sigalarm` 和 `SYS_sigreturn` 的定义。
- 在 `syscall.c` 中，将 `sys_sigalarm` 和 `sys_sigreturn` 添加到 `syscalls` 数组中。

```
1 #define SYS_sigalarm 22
2 #define SYS_sigreturn 23
```

```
1 extern uint64 sys_sigalarm(void);
2 extern uint64 sys_sigreturn(void);
3
4 [SYS_sigalarm] sys_sigalarm,
5 [SYS_sigreturn] sys_sigreturn
```

测试成功

A terminal window with a dark background and light text. The title bar at the top shows the user 'chengyu' on a 'chengyu-virtual-machine' at the path '~/os-labs/xv6-labs-2023'. The terminal output shows a sequence of test results for 'hart 2', including starting, initialization, and three test cases (test0, test1, test2, test3) each with an alarm and a passed status. The prompt '\$' is visible at the bottom.

```
chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023
hart 2 starting
init: starting sh
$ alarmtest
test0 start
.....alarm!
test0 passed
test1 start
.alarm!
.alarm!
alarm!
.alarm!
alarm!
.alarm!
alarm!
.alarm!
alarm!
test1 passed
test2 start
.....alarm!
test2 passed
test3 start
test3 passed
$
```

涉及到的文件

```
chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023
test3 passed
$ QEMU: Terminated
chengyu@chengyu-virtual-machine:~/os-labs/xv6-labs-2023$ git status
位于分支 traps
您的分支与上游分支 'origin/traps' 一致。

尚未暂存以备提交的变更:
  (使用 "git add <文件>..." 更新要提交的内容)
  (使用 "git restore <文件>..." 丢弃工作区的改动)
    修改:      Makefile
    修改:      kernel/proc.h
    修改:      kernel/syscall.c
    修改:      kernel/syscall.h
    修改:      kernel/sysproc.c
    修改:      kernel/trap.c
    修改:      user/user.h
    修改:      user/usys.pl

未跟踪的文件:
  (使用 "git add <文件>..." 以包含要提交的内容)
    docs/img/alarmtest.png

修改尚未加入提交 (使用 "git add" 和/或 "git commit -a")
chengyu@chengyu-virtual-machine:~/os-labs/xv6-labs-2023$
```


实验得分

```
chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023
make[1]: 离开目录“/home/chengyu/os-labs/xv6-labs-2023”
== Test answers-traps.txt ==
answers-traps.txt: OK
== Test backtrace test ==
$ make qemu-gdb
backtrace test: OK (13.4s)
== Test running alarmtest ==
$ make qemu-gdb
(8.8s)
== 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 (294.5s)
== Test time ==
time: OK
Score: 95/95
chengyu@chengyu-virtual-machine: ~/os-labs/xv6-labs-2023$
```