# Lab4 traps

# 1. RISC-V assembly (easy)

## 1）实验目的

> It will be important to understand a bit of RISC-V assembly, which you were exposed to in 6.004. There is a file `user/call.c` in your xv6 repo. `make fs.img` compiles it and also produces a readable assembly version of the program in `user/call.asm`.
> Read the code in call.asm for the functions g, f, and main. The instruction manual for RISC-V is on the reference page. Here are some questions that you should answer (store the answers in a file `answers-traps.txt`):
> 本实验探讨如何使用陷阱实现系统调用。 您将首先对堆栈进行热身练习，然后实现用户级陷阱处理的示例。了解一些RISC-V程序集非常重要，这在6.004中已经介绍过。xv6存储库中有一个文件 `user/call.c` 。 `make fs.img` 对其进行编译，并在 `user/call.asm` 中生成该程序的可读汇编版本。
> 阅读 `call.asm` 中有关g，f和main函数的代码。 RISC-V的说明手册在参考页上。 以下是您应回答的一些问题（将回答存储在文件 `answers-traps.txt` 中）：

# 2）实验步骤

## 哪些寄存器包含函数的参数？ 例如，哪个寄存器在main对printf的调用中保留13？

a0 a1 a2 ... a7等寄存器存储函数参数
通过实验目的中的提示进行操作，查看 `user/call.asm` 文件

```
void main(void) {
  ...
  22:   0800                    addi    s0,sp,16
  printf("%d %d\n", f(8)+1, 13);
  24:   4635                    li  a2,13
  ...
}
```

可以看出a2寄存器保留了13

## 在main的汇编代码中对函数f的调用在哪里？对函数g的调用在哪里？（提示：编译器可以内联函数。）

并没有对 `f` 和 `g` 函数的调用，g被内联f函数，f函数被内联到main函数

## 函数printf位于哪个地址?

```
    34:   600080e7            jalr    1536(ra) # 630 <printf>
```

查看 `user/call.asm` 文件，看到 `printf` 的入口为 `0x630`

## main中，jalr跳转到printf之后，ra的值是多少?

`ra = pc + 4 = 0x34 + 4 = 0x38` 即返回地址

## 运行下面的代码,回答问题

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

### 输出是什么

输出是 `He110 World` 字符串

### 如果RISC-V是大端序的，要实现同样的效果，需要将 i 设置为什么？需要将 57616 修改为别的值吗?

需要将i设置为0x726c6400；57616不修改

`%x` 表示以十六进制数形式输出， `57616` 的16进制表示就是 `e110` ，与大小端序无关， `%s` 是输出字符串，以整数 `i` 所在的开始地址读取，直到读取到 `'\0'` 为止。当是小端序表示的时候，内存中存放的数是： `72 6c 64 00` ，对应rld

**在下面的代码中，“y=”之后将打印什么？（注意：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);
```

输出为

```
x=3 y=1
```

y输出的值取决于执行这条语句前寄存器a2的值

# 3）实验中遇到的问题和解决方法

## 阅读代码

第一次看到以 `asm` 结尾的文件还不知道是什么，有点像计算机系统结构中的 `MIPS` 指令，但也有不懂得地方
对于实验要求的地方，我们上网查询xv6系统对应指令集中每个指令对应的含义

## 大小端序的理解

题目提到大小端序的时候有点不解，一开始猜测是地址从高位开始还是从低位开始，但不确定还是上网搜索一下资料

# 4）实验心得

## 汇编语言

第一次接触到汇编语言这样这么底层的语言，看着比较难懂，但是还是通过网络的力量一点点将其看得大致明白
其中对于 `ra` 寄存器好像在本实验中用得比较多。我们去更加了解以下它的作用：在RISC-V架构中，ra寄存器是一个特殊的寄存器，全称为返回地址寄存器（Return Address Register）。它用于保存**函数调用的返回地址**，即函数执行完毕后继续执行的下一条指令的地址。当程序执行函数调用时，当前指令的地址会被保存到ra寄存器中。然后，函数执行完成后，通过从ra寄存器中恢复保存的返回地址，程序可以顺利返回到函数调用的位置，继续执行后续的指令。在汇编代码中，通常使用jal和jalr指令进行函数调用和返回。在函数调用时，jal或jalr指令将返回地址保存到ra寄存器中。而在函数返回时，使用jr、ret或jalr指令将保存在ra寄存器中的返回地址加载到程序计数器（PC）中，以实现跳转到函数调用的下一条指令

### 大端序和小端序

是描述多字节数据在内存中存储顺序的两种不同方式，看来大致是猜对了！
在大端序中，高位字节（Most Significant Byte，MSB）存储在低地址处，而低位字节（Least Significant Byte，LSB）存储在高地址处。这意味着数据的最高有效字节排在最前面
在大端序中，高位字节（Most Significant Byte，MSB）存储在低地址处，而低位字节（Least Significant Byte，LSB）存储在高地址处。这意味着数据的最高有效字节排在最前面
例如：对于一个 `int` 类型的整数 `0x12345678` ，在大端序中，从低地址到高地址顺序为 `12 34 56 78` ；但在小端序中，从低地址到高地址顺序为 `78 56 34 12`
同时这里也涉及到16进制的相关转换，之前的计算机组成原理和计算机系统结构的知识在这里用上了

# 2. Backtrace (moderate)

## 1）实验目的

> For debugging it is often useful to have a backtrace: a list of the function calls on the stack above the point at which the error occurred.
> Implement a `backtrace()` function in `kernel/printf.c` . Insert a call to this function in `sys_sleep` , and then run `bttest` , which calls `sys_sleep` . Your output should be as follows:

```
backtrace:
0x0000000080002cda
0x0000000080002bb6
0x0000000080002898
```

> After `bttest` exit qemu. In your terminal: the addresses may be slightly different but if you run `addr2line -e kernel/kernel` (or `riscv64-unknown-elf-addr2line -e kernel/kernel` ) and cut-and-paste the above addresses as follows:

```
$ addr2line -e kernel/kernel
0x0000000080002de2
0x0000000080002f4a
0x0000000080002bfc
Ctrl-D
```

> You should see something like this:

```
kernel/sysproc.c:74
kernel/syscall.c:224
kernel/trap.c:85
```

> The compiler puts in each stack frame a frame pointer that holds the address of the caller's frame pointer. Your backtrace should use these frame pointers to walk up the stack and print the saved return address in each stack frame.
> 对于调试来说，有一个 backtrace 通常是有用的：error发生点之前，栈上的一系列函数调用。 kernel/printf.c 中实现 backtrace() 函数

在 sys_sleep() 中插入对这个函数的调用，然后运行 bttest ， 它调用 sys_sleep 。编写函数 backtrace()，遍历读取栈帧(frame pointer)并输出函数返回地址

# 2）实验步骤

## 编写代码

在 kernel/defs.h 声明 backtrace() ， 使得 sys_sleep 可以调用这个函数

```
void backtrace();
```

在 kernel/riscb.h 中添加内联函数 r_fp() ， 从而能够读取帧指针(frame pointer)的值
GCC编译器存储当前执行函数的帧指针在寄存器 s0 ， 因此我们从 s0 中读取

```
static inline uint64 r_fp() {
    uint64 x;
    asm volatile("mv %0, s0" : "=r" (x) );
    return x;
}
```

在 kernel/printf.c 中编写 backtrace() 的实现，使其输出所有的栈帧

```
void backtrace() {
  // 当前栈帧
  uint64 fp = r_fp();
  // 用户栈最高地址
  uint64 top = PGROUNDUP(fp);
  // 用户栈最低地址
  uint64 bottom = PGROUNDDOWN(fp);
  // 输出当前栈中返回地址
  for (; fp >= bottom && fp < top; fp = *((uint64 *) (fp - 16))) {
      printf("%p\n", *((uint64 *) (fp - 8)));
  }
}
```

该函数通过调用上述的 r_fp() 函数读取寄存器 s0 中的当前函数栈帧 fp
参考RISC-V的栈结构, 我们可以知道 fp-8 存放返回地址，fp-16 存放原栈帧。

从而可以通过原栈帧得到上一级栈结构, 直到获取到最初的栈结构

接着在 `kernel/sysproc.c` 的 `sys_sleep()` 函数中调用 `backtrace()`

```
uint64
sys_sleep(void)
{
  ...
  release(&tickslock);
  // lab4-2
  backtrace();
  return 0;
}
```

然后在 `kernel/printf.c` 的函数 `panic()` 中调用 `backtrace()`

```
void
panic(char *s)
{
  pr.locking = 0;
  printf("panic: ");
  printf(s);
  printf("\n");
  // lab4-2
  backtrace();
  panicked = 1; // freeze uart output from other CPUs
  for(;;)
    ;
}
```

## 测试程序

在存放 `Makefile` 文件的目录下执行如下命令

```
$ make qemu
```

启动xv6系统 QEMU模拟器
键入指令 `bttest` 进行测试，成功则会输出 3 个栈帧的返回地址

```
$ bttest
0x0000000080002d4c
0x0000000080002bae
0x0000000080002898
```

键入 `Ctrl+a` ，松开，然后键入 `x` ，退出xv6系统
退出xv6进行单元测试

```
root@LAPTOP-UER420HO:~/traps_lab4# ./grade-lab-traps  backtrace
make: 'kernel/kernel' is up to date.
== Test backtrace test == backtrace test: OK (2.0s)
```

## 3）实验中遇到的问题和解决方法

本实验的步骤较之前来说相对简单，主要在于概念的理解，了解xv6系统中的函数调用栈帧结构的相关知识

## 4）实验心得

### fp-16和fp-8

我们在获得帧指针(Frame Pointer)的时候，这个寄存器指向的是上一级栈帧的最后一个位置，因此fp-8存放的就是上一级函数的返回地址ra，而fp-16即为上一级的帧指针，地址按照层层递减存放的，实际上形成一个数组，在数组的最后RISC-V明确规定了最后一个指针为0，因此我们不仅可以通过上述代码中设置的 `top` 和 `bottom` 检测循环是否结束，还可以通过地址判断。同时，在xv6系统中这个栈只有一页的大小，我们也可以利用 `PGSIZE` 进行检测

# 3. Alarm (hard)

## 1）实验目的

> In this exercise you'll add a feature to xv6 that periodically alerts a process as it uses CPU time. This might be useful for compute-bound processes that want to limit how much CPU time they chew up, or for processes that want to compute but also want to take some periodic action. More generally, you'll be implementing a primitive form of user-level interrupt/fault handlers; you could use something similar to handle page faults in the application, for example. Your solution is correct if it passes alarmtest and usertests.
> 在本练习中，您将向xv6添加一个功能，该功能在进程使用CPU时间时定期提醒进程。这对于想要限制占用CPU时间的计算绑定进程，或者对于想要计算但也想要采取一些周期性操作的进程来说可能很有用。更一般地说，您将实现用户级中断/故障处理程序的原始形式；例如，您可以使用类似的方法来处理应用程序中的页面错误。如果您的解决方案通过了警报测试和用户测试，那么它就是正确的
> 您应该添加一个新的 `sigalarm(interval, handler)` 系统调用。 如果应用程序调用 `sigalarm(n, fn)` ，则在程序每消耗n个“ tick” **CPU时间之后，内核应导致调用应用程序函数fn。**

> 当fn返回时，应用程序应从中断处恢复。 tick是xv6中相当随意的时间单位，由硬件计时器产生中断的频率决定。 如果应用程序调用**sigalarm(0，0)**，则内核应停止生成定期警报调用

> alarmtest在test0中调用 `sigalarm(2, periodic)` ，以要求内核每2个滴答强制一次对 `periodic()` 的调用，然后旋转一段时间。 您可以在 `user/alarmtest.asm` 中看到 `alarmtest` 的汇编代码，这对于调试很方便。 当 `alarmtest` 产生这样的输出并且 `usertests` 也正确运行时，您的解决方案是正确的：

```
$ 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
$ usertests
...
ALL TESTS PASSED
$
```

> 完成后，您的解决方案将只有几行代码，但是正确实现可能有些棘手。 我们将使用原始存储库中的 `alerttest.c` 版本测试您的代码。 您可以修改 `alarmtest.c` 以帮助调试，但请确保原始的alarmtest表示所有测试均通过

## test0: invoke handler

- 通过修改内核以跳转到用户空间中的警报处理程序开始，这将导致test0打印“ alarm！”。 别担心，“警报”之后会发生什么！ 输出; 如果您的程序在打印“警报！”后崩溃，现在可以。 这里有一些提示：
- 您需要修改 `Makefile` 才能将 `alarmtest.c` 编译为xv6用户程序。
- 放置在 `user/user.h` 中的正确声明是：

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

- 更新 `user/usys.pl` （生成 `user/usys.S` ）、 `kernel/syscall.h` 和 `kernel/syscall.c` ，以允许 `alarmtest` 调用 `sigalarm` 和 `sigreturn` 系统调用
- 现在，您的 `sys_sigreturn` 应该只返回零
- 您的 `sys_sigalarm()` 应该在 `strcut proc` （在 `kernel/proc.h` 中）的新字段中存储警报间隔和指向处理函数的指针
- 您需要跟踪自上次调用（或留到下一次调用）到流程的警报处理程序以来经过了多少滴答； 您也需要为此在 `struct proc` 中添加一个新字段。 您可以在proc.c中的 `allocproc()` 中初始化proc字段
- 每次滴答，硬件时钟都会强制产生一个中断，该中断在 `kernel/trap.c` 中的 `usertrap()` 中处理
- 您只想在有计时器中断的情况下处理进程的警报滴答声； 你想要类似 `if (which_dev == 2) ...`
- 仅当进程的计时器溢出时才调用警报功能。 请注意，用户警报功能的地址可能为0（例如，在user / alarmtest.asm中，周期位于地址0）
- 您需要修改usertrap（），以便在进程的警报间隔到期时，用户进程执行处理程序函数。 当RISC-V上的陷阱返回到用户空间时，由什么决定用户空间代码恢复执行的指令地址？如果告诉qemu仅使用一个CPU，则使用gdb查看陷阱更容易。 通过运行

```
make CPUS=1 qemu-gdb
```

- 执行操作，如果alarmtest打印“ alarm！”，则表示成功。

### test1/test2(): resume interrupted code

- 可能是在test0或test1打印出“ alarm！”后，alarmtest崩溃，或者（最终）alarmtest打印出“ test1 failure”，或者退出了alerttest而没有打印“ test1 pass”。 要解决此问题，必须确保在完成警报处理程序后，控制返回到最初由计时器中断中断用户程序的指令。 您必须确保将寄存器的内容恢复到中断时所保存的值，以便用户程序可以在发生警报后不受干扰地继续运行。 最后，您应该在每次警报计数器关闭后对其进行“重新布防”，以便定期调用该处理程序
- 首先，我们为您做出了设计决策：用户警报处理程序需要在完成后调用 `sigreturn` 系统调用。 请查看 `alarmtest.c` 中的 `periodic` 作为示例。 这意味着您可以将代码添加到 `usertrap` 和 `sys_sigreturn` 中，这些代码可以协作以使用户进程在处理警报后正常恢复

## 2）实验步骤

### test0 invoke handler

#### 编写代码

在 `user/user.h` 文件中声明需要添加的系统调用 `sigalarm` 和 `sigreturn`

```
// lab4-3
int sigalarm(int ticks, void (*handler)());
int sigreturn(void);
```

在 user/usys.pl 文件中添加上述两个系统调用的入口

```
# lab4-3
entry("sigalarm");
entry("sigreturn");
```

在 kernel/syscall.h 添加定义

```
// lab4-3
#define SYS_sigalarm 22
#define SYS_sigreturn 23
```

在 kernel/syscall.c 添加函数声明

```
// lab4-3
extern uint64 sys_sigalarm(void);
extern uint64 sys_sigreturn(void);
...
static uint64 (*syscalls[])(void) = {
...
// lab4-3
[SYS_sigalarm]  sys_sigalarm,
[SYS_sigreturn] sys_sigreturn,
};
```

在 kernel/proc.h 文件中给 proc 结构体增加数据成员

```
struct proc {
  ...
  // lab4-3
  // 间隔
  int interval;
  // 调用函数地址
  uint64 handler;
  // 经过的时钟周期数
  int passtick;
  ...
};
```

在 kernel/proc.c 文件的 allocproc() 函数中对新的数据成员初始化赋值

```
static struct proc*
allocproc(void)
{
  ...
found:
  ...
  // lab4-3
  p->interval = 0;
  p->handler = 0;
  p->passtick = 0;

  return p;
}
```

回到 kernel/sysproc.c 并中添加 sys_sigalarm() 函数的实现，为 proc 中的相关数据成员赋值

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

针对test0，在 kernel/sysproc.c 中添加 sys_sigreturn() 函数的实现

```
// lab4-3-test0
uint64 sys_sigreturn(void) {
  return 0;
}
```

在 kernel/trap.c 文件的 usertrap() 函数中添加时钟中断添加相应的处理代码

```
void
usertrap(void)
{
  ...
  if(p->killed)
    exit(-1);
  // lab4-3
  // 时钟中断
  if (which_dev == 2) {
    // 经过了规定的时间间隔重置 执行handler
    if (p->interval != 0 && ++p->passtick == p->interval) {
      p->passedticks = 0;
      p->trapframe->epc = p->handler;
    }
  }
  ...
  usertrapret();
}
```

在 `Makefile` 文件中添加相关指令，使得可以对 `alarmtest.c` 编译

```
    ...
        $U/_zombie\
        $U/_alarmtest\
    ...
```

**测试程序**

在存放 `Makefile` 文件的目录下执行如下命令

```
$ make qemu
```

启动xv6系统 QEMU模拟器
键入指令 `alarmtest` 进行测试

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

看到 `test0 passed` 则说明test0测试成功

## test1/test2(): resume interrupted code

### 编写代码

继续在 kernel/proc.h 文件中给 proc 结构体增加数据成员，增加 struct trapframe* 类型的 sigframe 字段，以及 int 类型的 siglfag 字段

```
struct proc {
  ...
  struct trapframe* sigframe;
  int sigflag;
  ...
};
```

在 kernel/trap.c 文件中的 usertrap() 函数中执行 handler 之前对 sigframe 进行赋值

```
void
usertrap(void)
{
  ...
  if(p->killed)
    exit(-1);
  // lab4-3
  // 时钟中断
  if (which_dev == 2) {
    // 经过了规定的时间间隔重置 执行handler
    if (p->interval != 0 && ++p->passtick == p->interval && p->sigflag == 0) {

      p->sigflag = 1;

      p->passedticks = 0;

      // 使用trapframe后的一部分内存，trapframe大小为288B
      // 因此只要在trapframe地址后288以上地址都可
      // 此处512只是为了取整数幂
      p->sigframe = p->trapframe + 512;
      memmove(p->sigframe, p->trapframe, sizeof(struct trapframe));

      p->trapframe->epc = p->handler;
    }
  }
  ...
  usertrapret();
}
```

在 kernel/sysproc.c 文件中实现 sys_sigreturn() 函数，它将frame复制回来

```
// lab4-3
uint64 sys_sigreturn(void) {
  struct proc* p = myproc();
  // 判断地址是否正确
  if (p->sigframe != p->trapframe + 512)
    return -1;
  memmove(p->trapframe, p->sigframe, sizeof(struct trapframe));
  // 重置
  p->passtick = 0;
  p->sigframe = 0;
  p->sigflag = 0;
  return 0;
}
```

在 kernel/proc.c j文件中的 allocproc() 中, 初始化 p->sigframe

```
static struct proc*
allocproc(void)
{
  ...
found:
  ...
  // lab4-3
  p->interval = 0;
  p->handler = 0;
  p->passtick = 0;
  p->sigframe = 0;
  p->sigflag = 0;

  return p;
}
```

**测试程序**

在存放 Makefile 文件的目录下执行如下命令

```
$ make qemu
```

启动xv6系统 QEMU模拟器
键入指令 alarmtest 进行测试

```
$ 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
```

看到test0、test1和test2均测试成功

键入 `usertests`

```
$ usertests
usertests starting
test execout: OK
test copyin: OK
test copyout: OK
test copyinstr1: OK
test copyinstr2: OK
test copyinstr3: OK
test truncate1: OK
test truncate2: OK
test truncate3: OK
test reparent2: OK
test pgbug: OK
test sbrkbugs: usertrap(): unexpected scause 0x000000000000000c pid=3236
            sepc=0x0000000000005406 stval=0x0000000000005406
usertrap(): unexpected scause 0x000000000000000c pid=3237
            sepc=0x0000000000005406 stval=0x0000000000005406
OK
test badarg: OK
test reparent: OK
test twochildren: OK
test forkfork: OK
test forkforkfork: 0x0000000080002da0
0x0000000080002c02
0x00000000800028b0
0x0000000080002da0
0x0000000080002c02
0x00000000800028b0
OK
test argptest: OK
test createdelete: OK
test linkunlink: OK
test linktest: OK
test unlinkread: OK
test concreate: OK
test subdir: OK
test fourfiles: OK
test sharedfd: OK
test exectest: OK
test bigargtest: OK
test bigwrite: OK
test bsstest: OK
test sbrkbasic: OK
test sbrkmuch: OK
test kernmem: usertrap(): unexpected scause 0x000000000000000d pid=6216
            sepc=0x000000000000201a stval=0x0000000080000000
usertrap(): unexpected scause 0x000000000000000d pid=6217
            sepc=0x000000000000201a stval=0x000000008000c350
usertrap(): unexpected scause 0x000000000000000d pid=6218
            sepc=0x000000000000201a stval=0x00000000800186a0
```

```
usertrap(): unexpected scause 0x000000000000000d pid=6219
            sepc=0x000000000000201a stval=0x00000000800249f0
usertrap(): unexpected scause 0x000000000000000d pid=6220
            sepc=0x000000000000201a stval=0x0000000080030d40
usertrap(): unexpected scause 0x000000000000000d pid=6221
            sepc=0x000000000000201a stval=0x000000008003d090
usertrap(): unexpected scause 0x000000000000000d pid=6222
            sepc=0x000000000000201a stval=0x00000000800493e0
usertrap(): unexpected scause 0x000000000000000d pid=6223
            sepc=0x000000000000201a stval=0x0000000080055730
usertrap(): unexpected scause 0x000000000000000d pid=6224
            sepc=0x000000000000201a stval=0x0000000080061a80
usertrap(): unexpected scause 0x000000000000000d pid=6225
            sepc=0x000000000000201a stval=0x000000008006ddd0
usertrap(): unexpected scause 0x000000000000000d pid=6226
            sepc=0x000000000000201a stval=0x000000008007a120
usertrap(): unexpected scause 0x000000000000000d pid=6227
            sepc=0x000000000000201a stval=0x0000000080086470
usertrap(): unexpected scause 0x000000000000000d pid=6228
            sepc=0x000000000000201a stval=0x00000000800927c0
usertrap(): unexpected scause 0x000000000000000d pid=6229
            sepc=0x000000000000201a stval=0x000000008009eb10
usertrap(): unexpected scause 0x000000000000000d pid=6230
            sepc=0x000000000000201a stval=0x00000000800aae60
usertrap(): unexpected scause 0x000000000000000d pid=6231
            sepc=0x000000000000201a stval=0x00000000800b71b0
usertrap(): unexpected scause 0x000000000000000d pid=6232
            sepc=0x000000000000201a stval=0x00000000800c3500
usertrap(): unexpected scause 0x000000000000000d pid=6233
            sepc=0x000000000000201a stval=0x00000000800cf850
usertrap(): unexpected scause 0x000000000000000d pid=6234
            sepc=0x000000000000201a stval=0x00000000800dbba0
usertrap(): unexpected scause 0x000000000000000d pid=6235
            sepc=0x000000000000201a stval=0x00000000800e7ef0
usertrap(): unexpected scause 0x000000000000000d pid=6236
            sepc=0x000000000000201a stval=0x00000000800f4240
usertrap(): unexpected scause 0x000000000000000d pid=6237
            sepc=0x000000000000201a stval=0x0000000080100590
usertrap(): unexpected scause 0x000000000000000d pid=6238
            sepc=0x000000000000201a stval=0x000000008010c8e0
usertrap(): unexpected scause 0x000000000000000d pid=6239
            sepc=0x000000000000201a stval=0x0000000080118c30
usertrap(): unexpected scause 0x000000000000000d pid=6240
            sepc=0x000000000000201a stval=0x0000000080124f80
usertrap(): unexpected scause 0x000000000000000d pid=6241
            sepc=0x000000000000201a stval=0x0000000080131 2d0
usertrap(): unexpected scause 0x000000000000000d pid=6242
            sepc=0x000000000000201a stval=0x000000008013d620
usertrap(): unexpected scause 0x000000000000000d pid=6243
            sepc=0x000000000000201a stval=0x0000000080149970
usertrap(): unexpected scause 0x000000000000000d pid=6244
```

```
            sepc=0x000000000000201a stval=0x0000000080155cc0
usertrap(): unexpected scause 0x000000000000000d pid=6245
            sepc=0x000000000000201a stval=0x0000000080162010
usertrap(): unexpected scause 0x000000000000000d pid=6246
            sepc=0x000000000000201a stval=0x000000008016e360
usertrap(): unexpected scause 0x000000000000000d pid=6247
            sepc=0x000000000000201a stval=0x000000008017a6b0
usertrap(): unexpected scause 0x000000000000000d pid=6248
            sepc=0x000000000000201a stval=0x0000000080186a00
usertrap(): unexpected scause 0x000000000000000d pid=6249
            sepc=0x000000000000201a stval=0x0000000080192d50
usertrap(): unexpected scause 0x000000000000000d pid=6250
            sepc=0x000000000000201a stval=0x000000008019f0a0
usertrap(): unexpected scause 0x000000000000000d pid=6251
            sepc=0x000000000000201a stval=0x00000000801ab3f0
usertrap(): unexpected scause 0x000000000000000d pid=6252
            sepc=0x000000000000201a stval=0x00000000801b7740
usertrap(): unexpected scause 0x000000000000000d pid=6253
            sepc=0x000000000000201a stval=0x00000000801c3a90
usertrap(): unexpected scause 0x000000000000000d pid=6254
            sepc=0x000000000000201a stval=0x00000000801cfde0
usertrap(): unexpected scause 0x000000000000000d pid=6255
            sepc=0x000000000000201a stval=0x00000000801dc130
OK
test sbrkfail: usertrap(): unexpected scause 0x000000000000000d pid=6267
            sepc=0x0000000000003e7a stval=0x0000000000012000
OK
test sbrkarg: OK
test validatetest: OK
test stacktest: usertrap(): unexpected scause 0x000000000000000d pid=6271
            sepc=0x0000000000002188 stval=0x000000000000fbc0
OK
test opentest: OK
test writetest: OK
test writebig: OK
test createtest: OK
test openiput: 0x0000000080002da0
0x0000000080002c02
0x00000000800028b0
OK
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
```

可以看到所有测试均通过

键入 Ctrl+a ， 松开，然后键入 x ， 退出xv6系统
退出xv6进行单元测试

键入 ./grade-lab-traps

```
root@LAPTOP-UER420HO:~/traps_lab4# ./grade-lab-traps
make: 'kernel/kernel' is up to date.
== Test answers-traps.txt == answers-traps.txt: OK
== Test backtrace test == backtrace test: OK (1.4s)
== Test running alarmtest == (3.5s)
== Test   alarmtest: test0 ==
  alarmtest: test0: OK
== Test   alarmtest: test1 ==
  alarmtest: test1: OK
== Test   alarmtest: test2 ==
  alarmtest: test2: OK
== Test usertests == usertests: OK (69.5s)
== Test time ==
time: OK
Score: 85/85
```

# 3）实验中遇到的问题和解决方法

## 多次调用alarm handler

```
test2 failed: alarm handler called more than once
$ QEMU: Terminated
```

设置 sigflag 记录是否已经调用过，若已经调用就不再相应其他的alarm

# 4）实验心得

之前在lab2的时候了解了如何进入内核态，这里再次深入了解，完成了对于进入然后返回的整个流程
其中的重点就是如何中断后进入要执行的函数以及执行完 alarm handler 执行的函数后，要返回用户态被中断的地方
第一个问题，我们可以设置在 kernle/trap.c 文件中的 usertrap() 函数中设置 p->trapframe->epc = p->handler; ，其中的 epc 即为一个特殊的寄存器，用于保存异常处理程序的入口地址，当系统发生异常或中断时，CPU 会自动跳转到 EPC 寄存器中保存的异常处理程序的地址

同样在 usertrap() 函数中为当前进程的记录时间的数据结构自增，到达一定程度后就可以执行 handler ，注意其中 epc 已经不是用户程序被打断的那条指令的地址，因此在执行完 uesrret 汇编程序返回用户空间后，无法正确回到原来被打断的位置，因此，我们需要多设置一个 trapframe 记录
这里采用的方法是直接在 trapframe 地址后512的地址，就不用在初始化进程的时候申请空间，在陷入内核态之前进行一次复制，从而保存原来的 trapframe 之中的相关内容；并且在退出的时候复制回来，就可以回到被打断的位置