实验报告

Lab 4

1. RISC-V Assembly

首先看`user/call.c`的代码:

```
#include "kernel/param.h"
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int g(int x) {
  return x+3;
}

int f(int x) {
  return g(x);
}

void main(void) {
  printf("%d %d\n", f(8)+1, 13);
  exit(0);
}
```

`make qemu`后，生成相应的`call.asm`文件，`g`、`f`、`main`函数的汇编代码如下:

```
0000000000000000 <g>:
#include "kernel/param.h"
#include "kernel/types.h"
#include "kernel/stat.h"
#include "user/user.h"

int g(int x) {
   0: 1141                addi  sp,sp,-16
   2: e422                sd s0,8(sp)
   4: 0800                addi  s0,sp,16
  return x+3;
}
   6: 250d                addiw a0,a0,3
   8: 6422                ld s0,8(sp)
   a: 0141                addi  sp,sp,16
   c: 8082                ret

000000000000000e <f>:

int f(int x) {
   e: 1141                addi  sp,sp,-16
  10: e422                sd s0,8(sp)
  12: 0800                addi  s0,sp,16
  return g(x);
}
  14: 250d                addiw a0,a0,3
  16: 6422                ld s0,8(sp)
  18: 0141                addi  sp,sp,16
  1a: 8082                ret

000000000000001c <main>:

void main(void) {
  1c: 1141                addi  sp,sp,-16
  1e: e406                sd ra,8(sp)
  20: e022                sd s0,0(sp)
  22: 0800                addi  s0,sp,16
  printf("%d %d\n", f(8)+1, 13);
  24: 4635                li a2,13
  26: 45b1                li a1,12
  28: 00000517            auipc a0,0x0
  2c: 7c850513            addi  a0,a0,1992 # 7f0 <malloc+0xe8>
  30: 00000097            auipc ra,0x0
  34: 61a080e7            jalr  1562(ra) # 64a <printf>
  exit(0);
  38: 4501                li a0,0
  3a: 00000097            auipc ra,0x0
  3e: 298080e7            jalr  664(ra) # 2d2 <exit>
```

问题回答

（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?

根据RISC-V手册，函数的参数应包含在a0-a7寄存器中，如图:

Register	ABI Name	Description	Saver
x0	zero	Hard-wired zero	—
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3	gp	Global pointer	—
x4	tp	Thread pointer	—
x5	t0	Temporary/alternate link register	Caller
x6–7	t1–2	Temporaries	Caller
x8	s0/fp	Saved register/frame pointer	Callee
x9	s1	Saved register	Callee
x10–11	a0–1	Function arguments/return values	Caller
x12–17	a2–7	Function arguments	Caller
x18–27	s2–11	Saved registers	Callee
x28–31	t3–6	Temporaries	Caller
f0–7	ft0–7	FP temporaries	Caller
f8–9	fs0–1	FP saved registers	Callee
f10–11	fa0–1	FP arguments/return values	Caller
f12–17	fa2–7	FP arguments	Caller
f18–27	fs2–11	FP saved registers	Callee
f28–31	ft8–11	FP temporaries	Caller

在`main`函数对`printf`的调用中，由地址0x24的指令`li a2,13`可知，li指令将立即数13加载到a2寄存器，所以a2寄存器保存13。

（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）

由地址0x26的指令`li a1,12`可知，`main`直接得到了`f(8)+1`的结果12并把其保存在a1寄存器，说明编译器内联展开了函数并进行了优化。这点从`f`函数的汇编代码和`g`函数一模一样也能看出。

（3）函数`printf`位于哪个地址？

由`main`函数地址34的指令的注释可知，printf函数位于地址0x64a。如图:

```
void
printf(const char *fmt, ...)
{
 64a: 711d                addi  sp,sp,-96
 64c: ec06                sd ra,24(sp)
 64e: e822                sd s0,16(sp)
 650: 1000                addi  s0,sp,32
 652: e40c                sd a1,8(s0)
 654: e810                sd a2,16(s0)
 656: ec14                sd a3,24(s0)
 658: f018                sd a4,32(s0)
 65a: f41c                sd a5,40(s0)
 65c: 03043823            sd a6,48(s0)
 660: 03143c23            sd a7,56(s0)
  va_list ap;
```

可以通过计算验证。因为地址0x30的指令为`auipc ra 0x0`，就是把pc加上0赋值给ra，所以此时ra=0x30，由0x34地址的指令`jalr 1562(ra)`可知跳转到0x30+1562=0x64a。

（4）在jalr到`main`中的`printf`之后，寄存器ra中存储的值是?

应当返回下一地址，即0x38。从jalr指令的功能也可以分析出来，`jalr 1562(ra)`相当于跳转到ra+1562地址的指令，并增加ra使其指向下一指令地址作为返回地址。如图:

jalr rs	jalr x1, 0(rs)	Jump and link register

其中x1寄存器就是ra寄存器。

（5）运行以下代码：

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

输出是什么？注：https://www.asciitable.com/是一个将字节映射到字符的 ASCII 表。
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57616为不同的值?
参考链接：http://www.webopedia.com/TERM/b/big_endian.html；https://www.rfc-editor.org/ien/ien137.txt

对于前半部分，因为57616=0xE110，而`%x`表示按16进制输出，所以前半部分应该是`HE110`。

对于后半部分，将变量i的地址作为参数传入`printf`，并要求以%s，即字符串格式输出。很显然，传入i的地址就是把其作为指针，即字符串看待。i的内容是0x00646c72，可以分割为三个十六进制

数:0x64、0x6c、0x72，查ASCII表，分别对应d、l、r。由于RISC-V是小端的，所以相当于要倒着读，即对应的字符串是`rld`。所以后半部分应该是`world`。

综合即得输出结果是`HE110 World`，下图是执行结果：

```
$ call
HE110 World$
```

如果RISC-V是大端的，那么这几个字符就按顺序存放，i的内容就应该是0x726c6400，从而产生相同的输出。并不需要去修改57616为不同的值，因为57616是我们看到的值，内存中的存放方式对我们来讲是透明的，所以无需考虑。

（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);
```

结果如图:

```
$ call
x=3 y=8229$
```

可以看到x如期打印出了3，但y打印出了一个奇怪的值。查看汇编代码可以看到，`main`函数只把3传递到了a1寄存器，但`printf`函数需要的参数按照顺序依次是a1,a2...a7寄存器（见`printf`的汇编代码)，这里需要两个参数所以是a1和a2寄存器，所以最后打印出的是a1和a2寄存器的值，但`main`函数没有传递值给a2寄存器，打印的a2寄存器的值就是a2寄存器上次的（最初的）值，这是未知的。

2. Backtrace

2.1 实验要求

根据要求，我们需要类似命令`backtrace`，在xv6中实现自己的`backtrace()`函数，使得其能够追踪打印函数的调用栈。

2.2 实验过程

2.2.1 准备工作

在`kernel/printf.c`中定义`backtrace()函数`

```
/ my edit
void
backtrace(void)
{

}
// edit ends
```

在`kernel/sysproc.c/sys_sleep()`中插入对该函数的调用

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

在`kernel/defs.h`中声明`backtrace()`函数

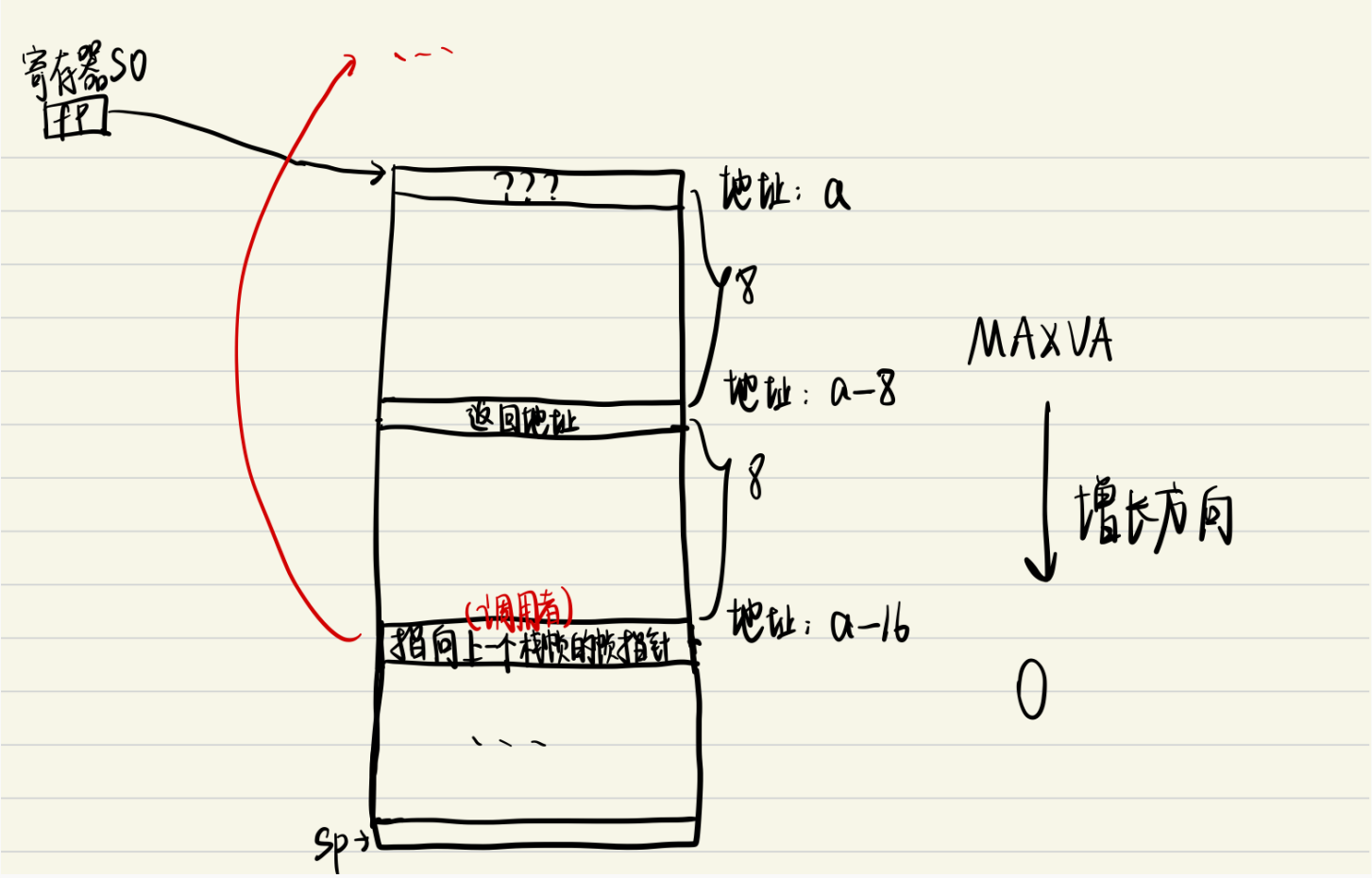
```
// printf.c
void            printf(char*, ...);
void            panic(char*) __attribute__((noreturn));
void            printfinit(void);
void            backtrace(void);
```

将以下函数添加到`kernel/riscv.h`

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

2.2.2 实验原理

对于每个栈帧，其结构大致如图：



每个函数对应一个栈帧，每次函数调用创建一个新的栈帧。栈增长的方向从高地址到低地址，对于每个栈帧，栈指针(sp)指向栈顶，帧指针(fp)指向栈底，用于确定栈帧。函数完成返回时，移动栈指针和帧指针。

寄存器s0保存了当前栈帧的栈**指针**，假设其指向地址a，那么地址a-8保存的**内容**为当前栈对应函数的**返回地址**，地址a-16保存的**内容**为指向上一个栈帧（**调用当前函数的那个栈帧**）的帧指针。

我们需要依次打印出函数的调用栈，即每次打印出函数的返回地址。那么，分为以下几个步骤：

1. 用`r_fp()`函数获取s0寄存器的值，即当前栈帧的帧指针。
2. 通过当前栈帧的帧指针，获取返回地址（将帧指针-8作为地址**的内容**）。
3. 通过当前栈帧的帧指针，获取上一个栈帧的帧指针(帧指针-16作为地址**的内容**)。
4. 用上一个栈帧的帧指针，重复上述步骤。

`backtrace()`函数是一个递归的过程，但递归需要有一个出口。xv6为每个内核栈在按页地址对齐的情况下分配一页大小的内存，所以我们可以使用`PGROUNDDOWN`和`PGROUNDUP`计算堆栈页面的顶部和底部地址（这两个函数在lab3中已经见过，作用是将地址按页向下/向上对齐）。

对于`backtrace()`函数，由于栈增长的方向为从高地址到低地址，所以这样一个回溯的过程是从低地址到高地址的，从而整个过程中，帧指针**不能超过最高的栈底**，即`PGROUNDUP(fp)`。

2.2.3 函数实现

```
// my edit
void
backtrace(void)
{
  printf("backtrace:\n");
  uint64 fp;
  fp = r_fp();

  uint64 top = PGROUNDUP(fp);

  while(fp<top)
  {
    printf("%p\n",*(uint64*)(fp-8));
    fp=*(uint64*)(fp-16);
  }
}
// edit ends
```

函数的实现本身比较简单。主要注意的一点在于while内部这一部分。在2.2中多次强调了**内容**和**地址**的区别（可以看上面的图），因为这里需要用到。我们获得的`fp`是一个指针(地址)，那么要获得这个地址-8对应的**内容**，需要先将`uint64`类型的变量作为指针类型（即作为地址）`uint64*`，再用`*`运算符获取其内容。获取调用栈帧的帧指针也是同理的。

2.2.4 实验结果

`make qemu`后执行`bttest`，结果如图(**注：这个结果随代码文件的内容不同而不同，但通过`addr2line`追踪到的代码行的内容应该是相同的**):

```
$ bttest
backtrace:
0x00000000800021ac
0x000000008000201e
0x0000000080001d14
$
```

退出qemu后执行`addr2line -e kernel/kernel`，并输入上面`backtrace`的结果，如图:

```
breezer@ubuntu:~/xv6-labs4-2022$ addr2line -e kernel/kernel
0x00000000800021ac
/home/breezer/xv6-labs4-2022/kernel/sysproc.c:71
0x000000008000201e
/home/breezer/xv6-labs4-2022/kernel/syscall.c:141
0x0000000080001d14
/home/breezer/xv6-labs4-2022/kernel/trap.c:76
```

我们可以验证一下结果的正确性。在`kernel/sysproc.c`的第71行：

```
67      sleep(&ticks, &tickslock);
68    }
69    release(&tickslock);
70    backtrace();        //here
71    return 0;
72  }
```

第70行调用`backtrace()`后返回地址应该是下一条指令，所以确实是第71行。

在`kernel/syscall.c`的第141行：

```
137   num = p->trapframe->a7;
138   if(num > 0 && num < NELEM(syscalls) && syscalls[num]) {
139     // Use num to lookup the system call function for num, call it,
140     // and store its return value in p->trapframe->a0
141     p->trapframe->a0 = syscalls[num]();
142   } else {
143     printf("%d %s: unknown sys call %d\n",
144             p->pid, p->name, num);
145     p->trapframe->a0 = -1;
```

第138行这里调用了`syscalls[num]`，会进入到这个函数指针数组调用对应的syscall(这里是`sys_sleep`)，返回地址是下一条指令，即第141行。

在`kernel/trap.c`的第76行：

```
53  if(r_scause() == 8){
54    // system call
55
56    if(killed(p))
57      exit(-1);
58
59    // sepc points to the ecall instruction,
60    // but we want to return to the next instruction.
61    p->trapframe->epc += 4;
62
63    // an interrupt will change sepc, scause, and sstatus,
64    // so enable only now that we're done with those registers.
65    intr_on();
66
67    syscall();
68  } else if((which_dev = devintr()) != 0){
69    // ok
70  } else {
71    printf("usertrap(): unexpected scause %p pid=%d\n", r_scause(), p->pid);
72    printf("            sepc=%p stval=%p\n", r_sepc(), r_stval());
73    setkilled(p);
74  }
75
76  if(killed(p))
77    exit(-1);
78
```

第67行调用了`syscall()`函数，返回地址是下一条指令。注意下面的两个else都不进入，所以下一条指令是第76行。

通过验证，说明了我们得到的结果是正确的。

3. Alarm

3.1 实验要求

我们需要在xv6中添加一个功能alarm，该功能会在进程使用CPU一定时钟周期后进行提醒，相当于一种user-level的interrupt/fault handler。

具体地，实现分为两个系统调用`sigalarm()`和`sigreturn()`。对于函数`sigalarm(interval,handler)`，要求其在`interval`个"ticks"的CPU时间后执行`handler`函数。当`handler`函数返回时，需要执行`sigreturn()`函数，使得应用程序从原本中断的地方继续。

3.2 实验过程

3.2.1 准备工作

在`Makefile`的`UPROGS`中添加alarmtest的内容。

```
UPROGS=\
    $U/_cat\
    $U/_echo\
    $U/_forktest\
    $U/_grep\
    $U/_init\
    $U/_kill\
    $U/_ln\
    $U/_ls\
    $U/_mkdir\
    $U/_rm\
    $U/_sh\
    $U/_stressfs\
    $U/_usertests\
    $U/_grind\
    $U/_wc\
    $U/_zombie\
    $U/_alarmtest\
```

就像lab2添加系统调用一样，添加两个系统调用`sigalarm()`和`sigreturn()`

```
// 1. user/user.h
// my edit
int sigalarm(int interval,void(*handler)(void));
int sigreturn(void);
// edit ends
```

注意这里`sigalarm`的第二个参数是一个函数指针`void(*)(void)`类型。

```
# 2. user/usys.pl
# my edit
entry("sigalarm");
entry("sigreturn");
# edit ends
```

```
// 3. kernel/syscall.h
// my edit
#define SYS_sigalarm  22
#define SYS_sigreturn 23
// edit endss
```

```
// 4. kernel/syscall.c
// Prototypes for the functions that handle system calls.
...
// my edit
extern uint64 sys_sigalarm(void);
extern uint64 sys_sigreturn(void);
// edit ends

// An array mapping syscall numbers from syscall.h
// to the function that handles the system call.
static uint64 (*syscalls[])(void) = {
...
[SYS_sigalarm]  sys_sigalarm,
[SYS_sigreturn] sys_sigreturn,
};
```

```
//5. kernel/sysproc.c
// my edit
uint64
sys_sigalarm(void)
{

}

uint64
sys_sigreturn(void)
{

}
// edit ends
```

3.2.2 Test0:invoke handler

按照提示，此时`sys_sigreturn()`函数只需要返回0。

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

`user/alarmtest.c`中，`test0()`函数定义如下：

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

// tests whether the kernel calls
// the alarm handler even a single time.
void
test0()
{
  int i;
  printf("test0 start\n");
  count = 0;
  sigalarm(2, periodic);
  for(i = 0; i < 1000*500000; i++){
    if((i % 1000000) == 0)
      write(2, ".", 1);
    if(count > 0)
      break;
  }
  sigalarm(0, 0);
  if(count > 0){
    printf("test0 passed\n");
  } else {
    printf("\ntest0 failed: the kernel never called the alarm handler\n");
  }
}
```

可见，`test0()`函数首先调用`sigalarm()`函数，这是一个系统调用（不是时钟中断），然后进行循环以触发时钟中断，并检测到达alarm时间时是否执行了设定的handler函数。只要能够触发一次handler函数，就通过test0。

按照提示，`sys_sigalarm()`应该将interval和handler存储在proc结构的新字段中，且还要一个新字段用作ticks的计数。先在`kernel/proc.h`中定义这三个新字段：

```
// my edit
int nticks;
int interval;
uint64 handler;
// edit ends
```

并在`kernel/proc.c allocproc()`中初始化nticks。

```
...
// my edit
p->nticks = 0;
// edit ends
...
```

首先，我们应该从用户态获取参数。这可以使用`argint()`和`argaddr()`完成。然后把内容存到`myproc()`上。

```
// my edit
uint64
sys_sigalarm(void)
{
  int n;
  uint64 fn;
  argint(0,&n);
  argaddr(1,&fn);

  struct proc* p = myproc();
  p->interval = n;
  p->handler = fn;

  return 0;
}
```

具体的中断处理在`kernel/trap.c usertrap()`函数中。当`which_dev==2`时，说明当前trap是一次时钟中断。我们每次时钟中断都让计数器`nticks`自增，当`nticks == interval`时说明已经达到设定的警报时间，需要返回到用户态，并让用户执行给定的`handler()`函数。可以写出如下代码，注意对于不执行handler的所有情况都执行原本的`yield()`函数，保证CPU利用率。

```
// my edit
// this is a timer interrupt.
if(which_dev == 2)
{
  // if it is 0,then yield()
  if(p->interval !=0)
  {
    ++p->nticks;
    // it's time to call handler
    if(p->nticks ==p->interval)
    {
      ...
    }
    else
      yield();
  }
  else
    yield();
}
// edit ends
```

现在需要思考的问题是返回用户态时，怎么让下一个执行的函数是指定的`handler()`。根据xv6-book:

> sepc: When a trap occurs, RISC-V saves the program counter here (since the pc is then overwritten with the value in stvec). The sret (return from trap) instruction copies sepc to the pc. The kernel can write sepc to control where sret goes.

当trap发生时，RISC-V将pc保存到sepc寄存器，这样之后从trap中返回时通过将sepc的内容复制到pc，就能从原来产生trap的地方继续执行。注意最后一句中提到**内核能通过写sepc寄存器来控制从trap返回后从哪里开始执行。**

因此，我们只需要将`myproc()`中保存的`handler`(的地址)复制到sepc寄存器，并将`nticks`清零即可（这个是有必要的，不然后面的几个test不能进行）。

由xv6-book我们可知，用户态的寄存器被保存在相应进程的trapframe中，而trapframe中有一个`epc`变量，这就是我们需要的sepc寄存器。

```
struct trapframe {
  /*   0 */ uint64 kernel_satp;   // kernel page table
  /*   8 */ uint64 kernel_sp;     // top of process's kernel stack
  /*  16 */ uint64 kernel_trap;   // usertrap()
  /*  24 */ uint64 epc;           // saved user program counter
...
}
```

因此在`usertrap()`中的修改如下：

```
  // my edit
  // this is a timer interrupt.
  if(which_dev == 2)
  {
    // if it is 0,then yield()
    if(p->interval!=0)
    {
      ++p->nticks;
      // it's time to call handler
      if(p->nticks == p->interval)
      {
        p->trapframe->epc = p->handler;
        p->nticks = 0;
      }
      else
        yield();
    }
    else
      yield();
  }
  // edit ends
```

测试test0，实验结果如下:

```
init: starting sh
$ alarmtest
test0 start
.....................alarm!
test0 passed
```

3.2.3 Test1/2/3:resume interrupted code

test0只是完成了需要alarm时调用`handler()`的测试，而寄存器的内容并没有正确的恢复。

我们先看一下test1/2/3的内容。test1测试两方面内容。一是能够正确触发多次handler，二是能正确恢复用户寄存器；test2测试内核不允许重入调用handler；test3测试是否修改了a0寄存器。

现在，当alarm调用`handler()`后，我们需要完善`sigreturn()`函数以便恢复用户程序中断处的寄存器状态。

我们有必要梳理一下系统调用所经过的trap过程：

首先，我们知道系统调用会将系统调用号写到a7寄存器，并调用`ecall`指令开始trap。

`ecall`指令完成以下三件事：

1. 切换到内核模式。
2. 将pc值保存到sepc寄存器。
3. 将stvec复制到pc，开始执行新指令。

现在CPU还没有切换到内核页表和内核栈，也没有保存除了pc以外的其他用户寄存器，这些由软件完成。

stevec寄存器指向的是`uservec`的地址，同时也是trampoline的地址。由于trampoline被固定映射到每个进程（包括内核）页表的最顶部，从而在trampoline上完成用户态到内核态的切换时，指令能够继续执行。

对于`uservec`函数，它需要将页表切换到内核页表，栈指针指向内核栈，并保存用户寄存器。xv6提供了trapframe完成这件事。trapframe被固定映射到每个进程页表中trampoline的下面一页。在每次由内核态返回到用户态时，内核设置sscratch寄存器指向每个进程的trapframe，而trapframe有空间保存相应的内容，如用户寄存器等。

`uservec`首先交换a0寄存器和sscratch的内容，使得a0保存trapframe的地址，sscratch保存原本a0的内容。然后用trapframe的内容（已经预先保存好了内核栈、内核页表等的内容，见`usertrapret`）设置栈指针sp指向内核栈、satp指向内核页表等，并将用户寄存器保存到trapframe上（**包括把用户的a0寄存器保存到trapframe的a0寄存器，见trampoline.S**）。最后跳转到内核态的`usertrap`函数。

`usertrap`函数首先设置stevc指向`kernelvec`，使得内核发生trap时使用该函数处理。然后需要保存用户pc，虽然用户pc已经被保存在了sepc寄存器中，但`usertrap`有可能切换到另一个进程并进入到那个进程的用户空间，如果在该用户空间内再调用系统调用，就会导致sepc寄存器的内容被覆盖，从而最初的用户pc丢失而无法回到原本的位置继续执行。因此，我们需要将sepc保存到进程对应的trapframe上的epc寄存器保证不丢失。

调用相应的系统调用函数后，返回值应该放在trapframe的a0寄存器中（**注：trapframe的a0寄存器和用户的a0寄存器不同。后者现在还是保存trapframe的地址，而trapframe的a0寄存器通过查看`proc.h`中trapframe的定义就能看到，它实际的地址位于(a0+112)，现在保存的是用户传入的a0寄存器的值**）。接下来需要返回用户空间，执行`usertrapret`函数。

`usertrapret`函数设置stvec指向`uservec`，便于下一次用户态trap。然后设置trapframe的一些控制寄存器（satp、sp等，指向内核态内容），同样便于下一次用户trap。再之后，设置sepc为trapframe保存

的epc寄存器，即恢复最初系统调用用户的pc，并计算出用户页表的地址（在`userret`段使用，因为转换页表的工作只能在trampoline段完成）。最后跳转到`userret`段。

`userret`段先将satp指向用户页表，再恢复所有用户寄存器(除了a0)。最后，将trapframe的a0寄存器恢复到用户的a0寄存器（作为返回值），并执行`sret`返回。

回顾整个过程，可以发现，保存、恢复用户寄存器的关键在于trapframe。如果按照test0的写法，`sigreturn()`函数直接返回0，那么，由于返回用户空间后先执行了`perodic()`函数，这个函数用到了用户的寄存器，从而导致用户空间trap处保存的那些用户寄存器已经被修改了（**也就是说，虽然最初保存的用户寄存器没有丢失且成功返回到用户空间了，但已经被修改过了**）。

考虑到这一点，我们应该**将trapframe复制一份在进程结构体中**。这样，当执行`sigreturn()`时，就把复制的内容复制回trapframe；执行`sigalarm()`时，把现在的trapframe复制一份，从而完成状态的保存。（实际上应该用不到那么多，但这样实现比较简单）

因此，可以在`kernel/proc.h`的proc结构体中添加新成员`copy_trapframe`。这里还添加了一个`flag`变量，用于标识是否已经开始执行handler（防止重入调用）

```
// my edit
int nticks;
int interval;
uint64 handler;

struct trapframe* copy_trapframe;
int flag;
// edit ends
```

在`kernel/proc.c allocproc()`中初始化`copy_trapframe`(和`trapframe`的初始化一样)

```
// my edit
// Allocate a copy trapframe page.
if((p->copy_trapframe = (struct trapframe *)kalloc()) == 0){
  freeproc(p);
  release(&p->lock);
  return 0;
}
// edit ends
```

在`sigalarm()`函数中设置`flag`的值

```
// my edit
uint64
sys_sigalarm(void)
{
  int n;
  uint64 fn;
  argint(0,&n);
  argaddr(1,&fn);

  struct proc* p = myproc();
  p->interval = n;
  p->handler = fn;
  p->flag = 1;

  return 0;
}
```

当触发alarm，且`flag==1`时，说明之前还没有执行handler，于是将`flag`设为0，表示不允许重入；然后用`memmove`函数(可以在`kernel/defs.h`中找可能有用的函数)将`trapframe`的内容复制到`copy_trapframe`，最后设置epc。

```
// my edit
// this is a timer interrupt.
if(which_dev == 2)
{
  // if it is 0,then yield()
  if(p->interval!=0)
  {
    ++p->nticks;
    // it's time to call handler
    if(p->nticks == p->interval)
    {
      p->nticks = 0;
      if(p->flag)
      {
        p->flag = 0;
        memmove(p->copy_trapframe,p->trapframe,sizeof(struct trapframe));
        p->trapframe->epc = p->handler;
      }
      else
        yield();
    }
    else
      yield();
  }
  else
    yield();
}
// edit ends
```

当执行`sigreturn()`函数时，将`flag`设为1表示handler已经返回，并将保存的`copy_trapframe`复制给`trapframe`。注意返回值是`copy_trapframe`的a0寄存器，这是为了满足test3的要求。(详见4.4 关于test3的疑惑)

```
uint64
sys_sigreturn(void)
{
  struct proc* p = myproc();

  p->flag = 1;
  memmove(p->trapframe,p->copy_trapframe,sizeof(struct trapframe));

  return p->copy_trapframe->a0;
}
// edit ends
```

3.2.4 实验结果

`make qemu`后执行`alarmtest`，结果如图:

```
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
test3 start
test3 passed
```

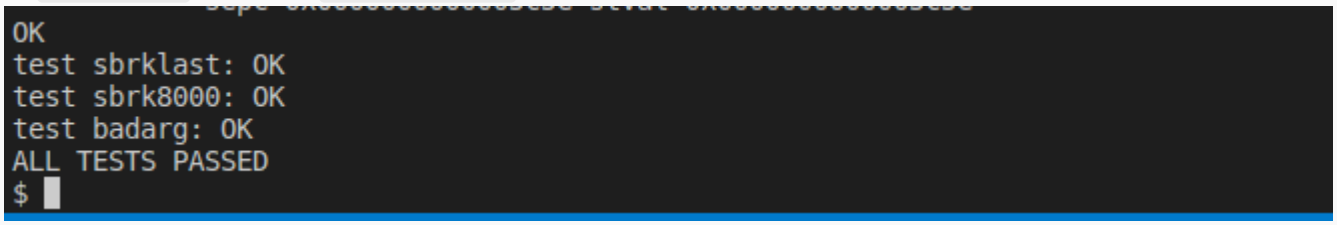
执行`usertests -q`，结果:

```
            sepc=0x0000000000005c5e stval=0x0000000000005c5e
OK
test sbrklast: OK
test sbrk8000: OK
test badarg: OK
FAILED -- lost some free pages 25899 (out of 32454)
```

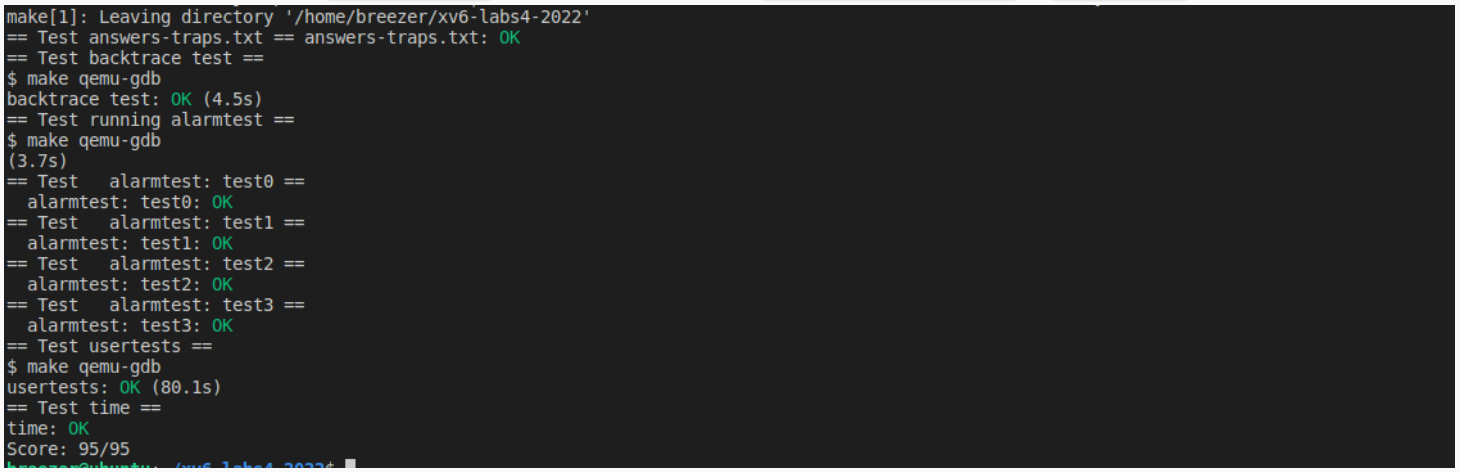
可以看到它提示丢失了一些free pages。这是由于我们没有释放掉`copy_trapframe`申请的空间。修改`kernel/proc.c freeproc()`函数:

```
// free a proc structure and the data hanging from it,
// including user pages.
// p->lock must be held.
static void
freeproc(struct proc *p)
{
  if(p->trapframe)
    kfree((void*)p->trapframe);
  p->trapframe = 0;
  // my edit
  if(p->copy_trapframe)
    kfree((void*)p->copy_trapframe);
  p->copy_trapframe = 0;
  // edit ends
  ...
}
```

`make qemu`后再执行`usertests -q`：



最后，在目录下执行`make grade`检查所有实验(需要`answer-traps.txt`和`time.txt`)



4. 实验问题

4.1 对RISC-V指令集不熟悉

因为只学过MIPS指令集，RISC-V指令集显得比较陌生。所幸本次实验中涉及的汇编代码不是很多，参考手册也能做的下去。

4.2 地址和内容的转换

对于backtrace实验，这个实验思路很容易，但刚开始实际实现起来总是有问题，主要还是在于指针(地址)和内容的区分吧。我们用`argaddr`函数获得的地址以`uint64`形式保存，所以如果我们要访问这个地址（或者这个地址附近）的内容的话，需要先强制类型转换为`uint64*`类型，然后用`*`运算符解引用。

4.3 对sepc寄存器的理解

最开始测试alarm实验的test0的时候，在`kernel/trap.c usertrap()`函数中，关于如何将handler的地址写到sepc寄存器，我是这样写的:

```
w_sepc(p->handler);
```

这个函数定义在`kernel/riscv.h`中，因为第一部分的实验中用了一个函数`r_fp()`来读取fp寄存器的值，所以我觉得应该可以类似地用`w_sepc()`函数将handler的地址写道sepc寄存器。

但这样执行的结果是test0无法通过。最开始我始终没理解为什么，在梳理了一遍trap过程后就很明白了：因为在`usertrapret`的时候，要将trapframe的epc寄存器写回到sepc寄存器，所以直接用`w_sepc()`函数写的handler地址相当于被覆盖掉了。

```
  // kernel/trap.c usertrapret()
  // set S Exception Program Counter to the saved user pc.
  w_sepc(p->trapframe->epc);
```

因此如果要改变返回时开始执行指令的地址，修改trapframe的epc寄存器即可。

4.4 关于test3的疑惑

最初我的`sys_sigreturn()`函数是这样写的:

```
uint64
sys_sigreturn(void)
{
  struct proc* p = myproc();

  if(p->flag == 0)
  {
    p->flag = 1;
    memmove(p->trapframe,p->copy_trapframe,sizeof(struct trapframe));
  }

  return 0;
}
// edit ends
```

神奇的是这样也是能直接通过test3(和整个alarmtest、make grade)的，即便直接返回了一个0。最初我觉得很理所当然，但后面我越想越觉得不对：

在alarmtest的4个测试中：

test0测试是否能调用handler。通过将handler地址复制到trapframe的epc寄存器完成；

test1测试是否能多次调用handler，以及恢复寄存器和状态。通过保存一份`copy_trapframe`完成；

test2测试是否会重入调用handler。通过设置flag并进行检测完成。

而test3描述的测试目标是"检测从`sys_sigreturn()`返回不会改变a0寄存器的值"。但按照前面分析的trap流程，作为用户寄存器的a0，首先与sscratch寄存器交换，于是a0寄存器保存trapframe的地址，然后sscratch寄存器保存到trapframe的a0寄存器(地址是**a0+112**)。完成系统调用后，其返回值会**写到trapframe的a0寄存器上**，这一步就**已经覆盖掉了用户寄存器a0的初始值**。最后返回时，也是将trapframe的所有用户寄存器复制回去，包括将trapframe->a0复制到a0寄存器。那么，test3所说的"`sys_sigreturn()`不修改a0寄存器的值"从何而来呢？这个test考察的究竟是什么，毕竟似乎什么也没做就通过了这个测试。

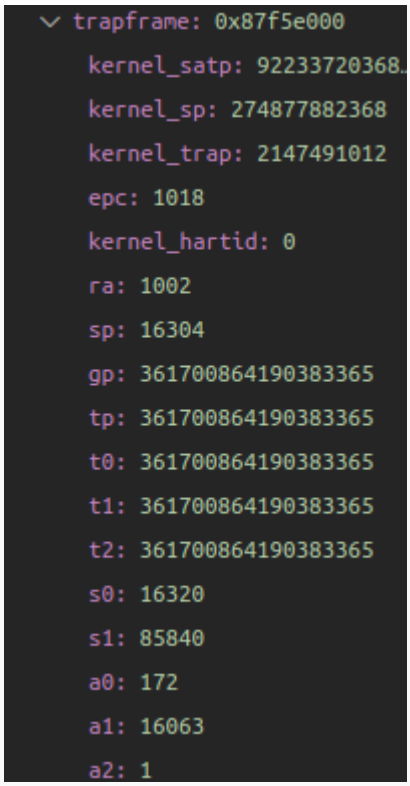
先看一下`alarmtest.c`中，`test3()`的代码:

```
//
// dummy alarm handler; after running immediately uninstall
// itself and finish signal handling
void
dummy_handler()
{
  sigalarm(0, 0);
  sigreturn();
}

//
// tests that the return from sys_sigreturn() does not
// modify the a0 register
void
test3()
{
  uint64 a0;

  sigalarm(1, dummy_handler);
  printf("test3 start\n");

  asm volatile("lui a5, 0");
  asm volatile("addi a0, a5, 0xac" : : : "a0");
  for(int i = 0; i < 500000000; i++)
    ;
  asm volatile("mv %0, a0" : "=r" (a0) );

  if(a0 != 0xac)
    printf("test3 failed: register a0 changed\n");
  else
    printf("test3 passed\n");
}
```

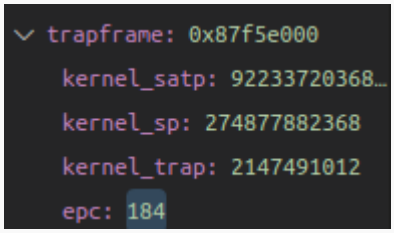
test3()执行了一串汇编代码，具体含义是，令a5寄存器=0,令a0=a5+0xac=0xac，然后开始循环。循环结束后，将a0寄存器的值复制到变量a0。确实是用来检测a0寄存器是否被修改的。注意这个`sigalarm(0,0)`表示销毁掉alarm，之后不再触发alarm。

我先在VScode上调试了一下，看了一下执行test3时，不同时间`trapframe`和`copy_trapframe`的值。(以下调试均在CPUS=1的情况下完成)

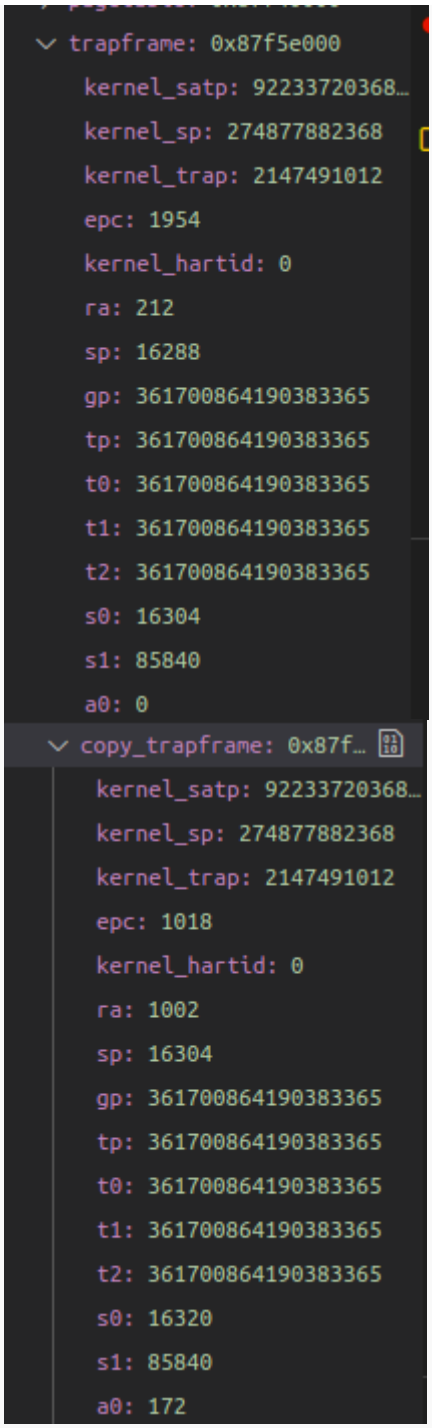
test3第一次触发时钟中断时，`trapframe`的值:



a0=172,epc=1018=0x3FA，在`alarmtest.asm`能找到是test3的for语句对应的汇编指令。

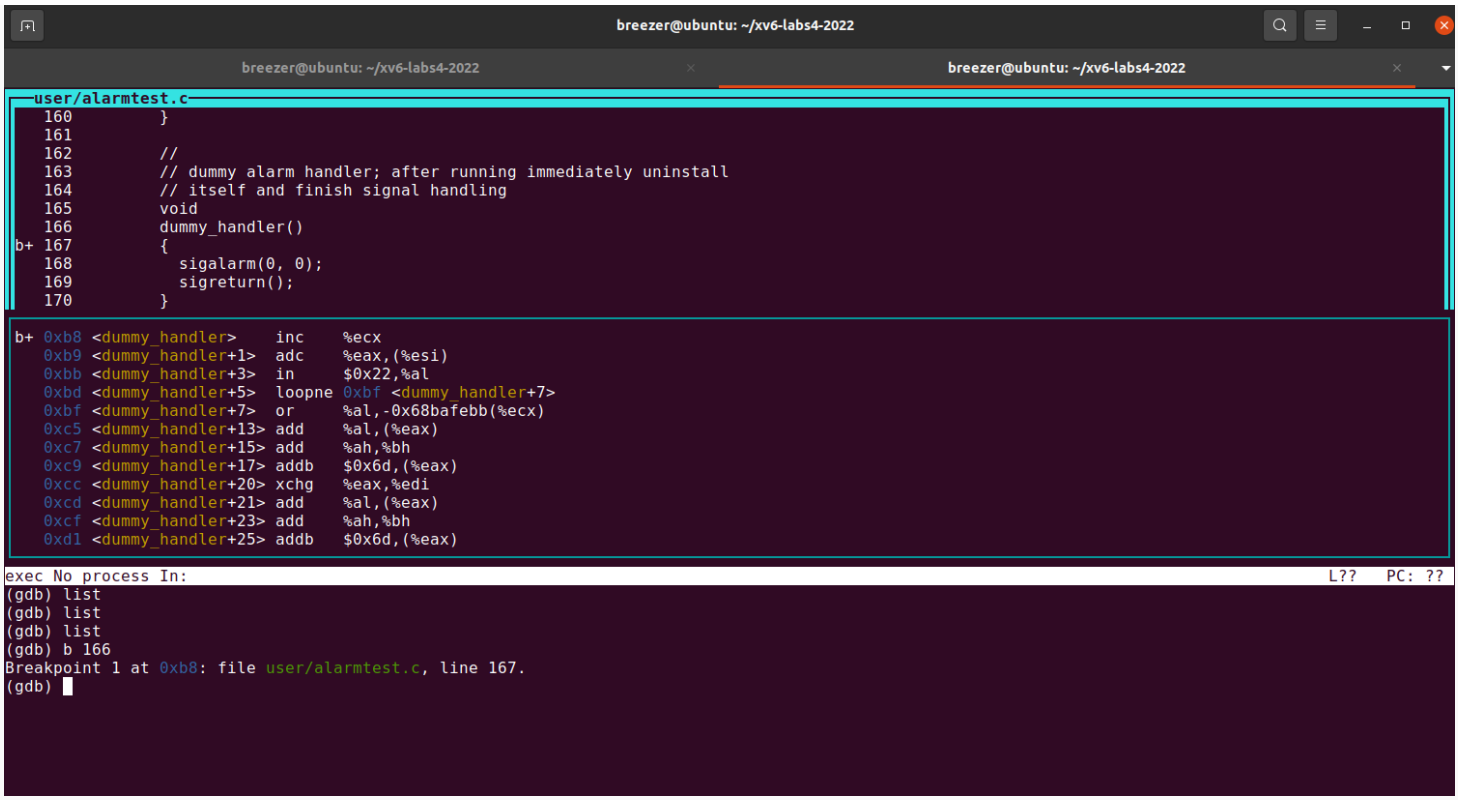


然后epc被改为handler的地址，184=0xB4是`dummy_handler()`的地址。
当执行`sys_sigreturn()`时，`trapframe`和`copy_trapframe`的值。

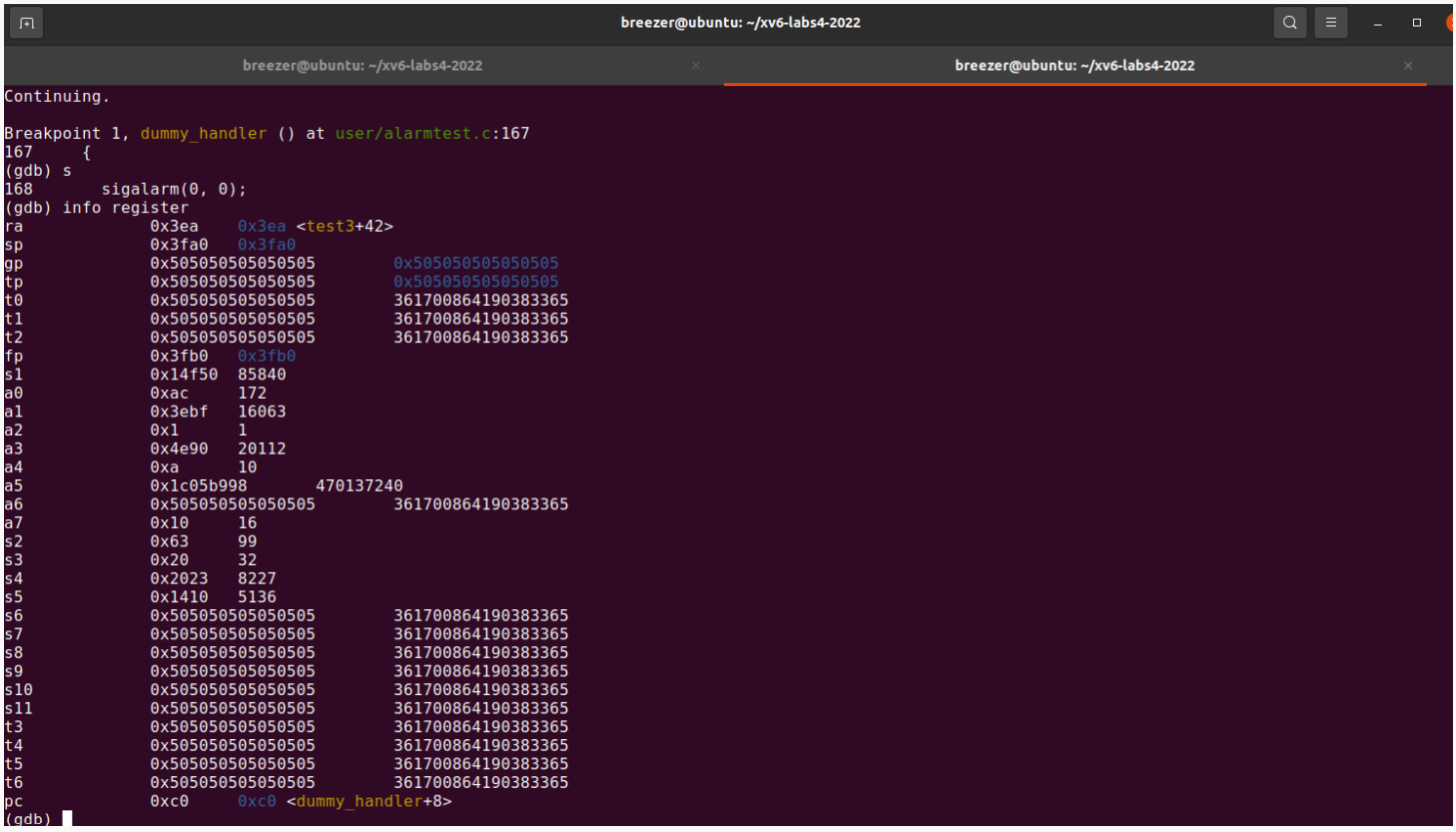


可见，`copy_trapframe`中a0的内容是传递进来的原始值172(0xac)，`trapframe`中a0的内容已经被写入了`sys_sigreturn()`的返回值0。这与上面的分析一致。这里`trapframe`的epc值1954=0x7A2对应`sigreturn()`的ret指令。

然后在gdb中调试`user/alarmtest.c`。考虑在`dummy_handler()`函数上打一个断点。



如下是触发时的寄存器情况，可以看到此时a0已经被写入172。



当`sigalarm(0,0)`结束时，a0就被写入返回值0了。

```
dummy_handler () at user/alarmtest.c:169
169       sigreturn();
(gdb) info reg
ra             0xcc     0xcc <dummy_handler+20>
sp             0x3fa0   0x3fa0
gp             0x505050505050505        0x505050505050505
tp             0x505050505050505        0x505050505050505
t0             0x505050505050505        361700864190383365
t1             0x505050505050505        361700864190383365
t2             0x505050505050505        361700864190383365
fp             0x3fb0   0x3fb0
s1             0x14f50  85840
a0             0x0      0
a1             0x0      0
a2             0x1      1
a3             0x4e90   20112
a4             0xa      10
a5             0x1c05b998       470137240
a6             0x505050505050505        361700864190383365
a7             0x16     22
s2             0x63     99
s3             0x20     32
s4             0x2023   8227
s5             0x1410   5136
s6             0x505050505050505        361700864190383365
s7             0x505050505050505        361700864190383365
s8             0x505050505050505        361700864190383365
s9             0x505050505050505        361700864190383365
s10            0x505050505050505        361700864190383365
s11            0x505050505050505        361700864190383365
t3             0x505050505050505        361700864190383365
t4             0x505050505050505        361700864190383365
t5             0x505050505050505        361700864190383365
t6             0x505050505050505        361700864190383365
pc             0xcc     0xcc <dummy_handler+20>
(gdb)
```

之后出现了令人匪夷所思的现象：`dummy_handler()`执行完`sigreturn()`后居然又回去执行前面的`asm volatile("lui a5,0")`指令了。因为时钟中断肯定是在for循环中触发的，且在上面的图里也能看到a5寄存器的值是470137240，如果看`alarmtest.asm`这一部分的汇编码就知道，循环的方式是令a5=500000000然后递减至0，说明触发`dummy_handler()`时确实在循环里面。

```
sigreturn () at user/usys.S:115
115       li a7, SYS_sigreturn
(gdb)
116       ecall
(gdb)
117       ret
(gdb)
test3 () at user/alarmtest.c:183
183        asm volatile("lui a5, 0");
(gdb) info reg
ra             0x3ea    0x3ea <test3+42>
sp             0x3fb0   0x3fb0
gp             0x505050505050505        0x505050505050505
tp             0x505050505050505        0x505050505050505
t0             0x505050505050505        361700864190383365
t1             0x505050505050505        361700864190383365
t2             0x505050505050505        361700864190383365
fp             0x3fc0   0x3fc0
s1             0x14f50  85840
a0             0x0      0
```

按照上面的分析，流程大概是这样的：触发时钟中断，将trapframe的epc设置为`dummy_handler()`的地址，并复制一份到trapframe；返回用户态后执行`dummy_handler()`，`dummy_handler()`首先执行`sigalarm(0,0)`终止alarm，然后`sigreturn()`。在`sys_sigreturn()`中，由于此时`flag`被重置为1，所以`copy_trapframe`的内容不会复制回`trapframe`，所以此时`sigreturn()`**不会返回for循环**，而是`dummy_handler()`。从而接下来是`dummy_handler()`返回上层函数。

但`dummy_handler()`是通过alarm调用的，并非直接意义上的函数调用，它会返回到哪里呢？这就需要看ra寄存器保存的地址了。我们看到，刚进入`dummy_handler()`的时候，ra是<test3+0x42>，这个地址

对应的正是`asm volatile("lui a5,0");`这条指令！因此，被设置为0的a0又被赋值为172，然后进入循环，这次没有alarm，所以最后结果就是正确的了。

这能通过test3显得多少有点太巧合了，完全不像题目要求的意思，因为上述流程的逻辑就是有问题的：调用`sigalarm(0,0)`就不返回中断处了？这显得很奇怪；其次，为什么不返回中断的地方就返回到了一个那么奇怪的地方，甚至都跑回去了，这是巧合还是什么？况且test3()的要求是"不修改a0"，这样一想，题目的意思似乎是想让`sys_sigreturn()`函数返回copy_trapframe上保存的a0值，这就能做到"不修改a0寄存器的值"。这样一想，逻辑就通顺多了。所以最终我的代码去掉了对flag的条件考察，并返回copy_trapframe的a0寄存器的值。

从上述的一系列思考引出一个问题：**通过alarm调用的函数，如果不返回中断的地方，它会返回哪里**？这个问题实际上在test0就应该出现了，因为test0的时候没有实现`sys_sigreturn()`函数。从原理上思考，刚调用函数时，增长栈指针，并把ra保存在sp+8的位置；需要返回时，再从sp+8的位置取出复制给ra。所以返回的ra实际就是**调用函数时ra寄存器的值**。调用`dummy_handler()`时在循环里，距离最近的ra改动在执行`printf()`函数时，如下图的汇编代码:

```
3d6: 3c2080e7             jalr  962(ra) # 794 <sigalarm>
 printf("test3 start\n");
3da: 00001517             auipc a0,0x1
3de: 9de50513             addi  a0,a0,-1570 # db8 <malloc+0x27e>
3e2: 00000097             auipc ra,0x0
3e6: 69a080e7             jalr  1690(ra) # a7c <printf>

 asm volatile("lui a5, 0");
3ea: 000007b7             lui   a5,0x0
 asm volatile("addi a0, a5, 0xac" : : : "a0");
3ee: 0ac78513             addi  a0,a5,172 # ac <slow_handler+0x72>
3f2: 1dcd67b7             lui   a5,0x1dcd6
3f6: 50078793             addi  a5,a5,1280 # 1dcd6500 <base+0x1dcd54f0>
 for(int i = 0; i < 500000000; i++)
3fa: 37fd                 addiw a5,a5,-1
3fc: fffd                 bnez  a5,3fa <test3+0x3a>
```

由jalr指令的功能就知道，此时的ra应该是下一条指令的地址，即0x3ea，这样就对上了!因此我们可以回答这个问题：**不返回中断的地方的话，它会返回上一次调用函数处的下一条指令。**

5. 实验感想

这次实验的第一部分主要是对汇编的理解，相对简单，可能需要注意一下编译器内联优化的情况；第二部分只要能理解栈指针、帧指针和返回地址的关系，就不难做出；第三部分比较难，需要对trap的整体流程有一个全面的把握，也需要关注分配内存、回收内存的工作。不过我大多数时间其实是用在探究test3这个问题上，自己动手调试还是很有收获的。