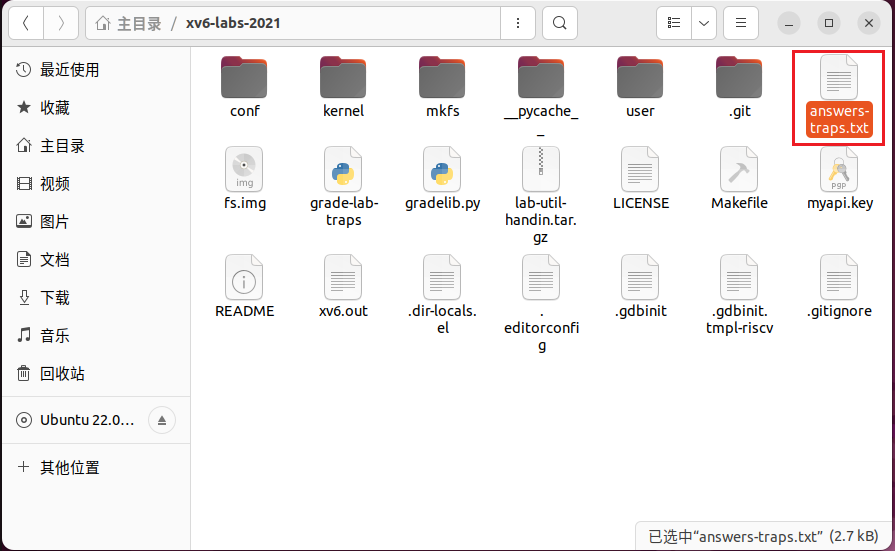
# 1 RISC-V assembly

## 1.1实验内容

在本次实验中，我们回答几个问题，答案保存在answers-traps.txt文件中。

## 1.2代码位置与截图





1.Which registers contain arguments to functions? For example, which register holds 13 in main's call to printf?

（哪些寄存器存储了函数调用的参数？举个例子，main 调用 printf 的时候，13 被存在了哪个寄存器中？）

答：a0到a7保存了函数调用的参数，其中a0和a1还能保存函数的返回值。

通给查看call.asm文件，在第45行可知a2寄存器保存了13这个参数。

2.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.)

（main 中调用函数 f 对应的汇编代码在哪？对 g 的调用呢？ (提示：编译器有可能会内链(inline)一些函数)）

答：没有调用f函数的代码，因为f函数与g函数都比较简单，因此g函数被内联到了f函数里，f函数又进一步内联到了main函数里。

3.At what address is the function printf located?

（printf 函数所在的地址是？）

答：0x0000000000000628, main 中使用 pc 相对寻址来计算得到这个地址。

4.What value is in the register ra just after the jalr to printf in main?

（在 main 中 jalr 跳转到 printf 之后，ra 的值是什么？）

答：ra是return address，是执行完函数后下一条指令的地址，在这里也就是jalr指令的下一条汇编指令的地址，

这里是0x0000000000000038。

5.Run the following code.

unsigned int i = 0x00646c72;

printf("H%x Wo%s", 57616, &i);

What is the output? Here's an ASCII table that maps bytes to characters.

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 to in order to yield the same output? Would you need to change to a different value?i57616

（运行下面的代码

unsigned int i = 0x00646c72;

printf("H%x Wo%s", 57616, &i);

输出是什么？

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？）

答：会输出"He110 World";

如果RISC-V是大端序的，要实现同样的效果，需要将 i 设置为0x726c6400；

不需要将57616修改为别的值。57616的十六进制是110，这无关端序，都会输出110。

6.In the following code, what is going to be printed after ? (note: the answer is not a specific value.) Why does this happen? 'y='

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

(在下面的代码中，'y=' 之后会打印什么？ (note: 答案不是一个具体的值) 为什么?)

答：输出的是一个受调用前的代码影响的“随机”的值。因为printf尝试读的参数数量比提供的参数数量多。

第二个参数`3`在寄存器a1里，所以'y='之后会打印寄存器a2的值。

# 2 Backtrace

## 2.1实验内容

本次实验中，需要我们在kernel/printf.c文件里编写一个函数backtrace函数， backtrace函数要实现输出若干地址，这些地址其实是调用过程中的各个函数的返回地址。

我们需要了解函数调用栈、栈帧等概念。函数调用栈深度最大为5，在使用gdb调试时可以输入i f 0(数字可以从0到4)，查看栈中不同深度的栈帧（i f是info frame的缩写）。栈帧会包含ra寄存器的信息，也就是执行完该函数后，下一条指令的地址。那么backtrace的任务，就是输出每一层次栈帧的ra的内容。

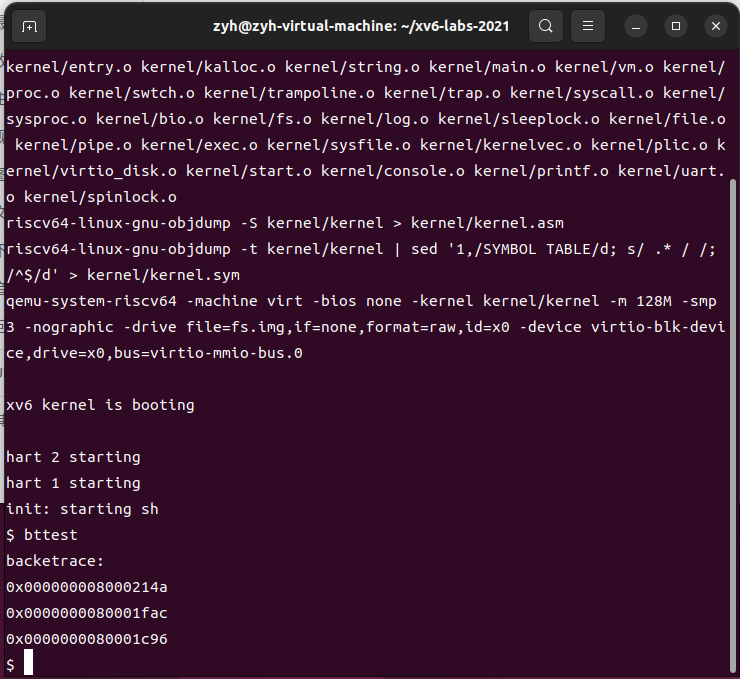
在层层嵌套的函数调用中，嵌套越深的函数，i f查询的数字就越大。

我们需要在sleep系统调用中加入backtrace函数，并通过user/bttest.c来测试我们编写的函数。bttest实际上就调用了sleep系统调用。

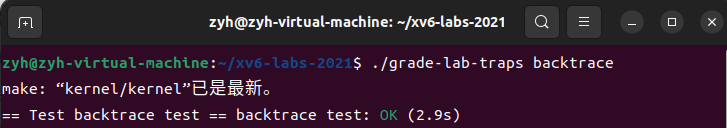
## 2.2代码位置与截图



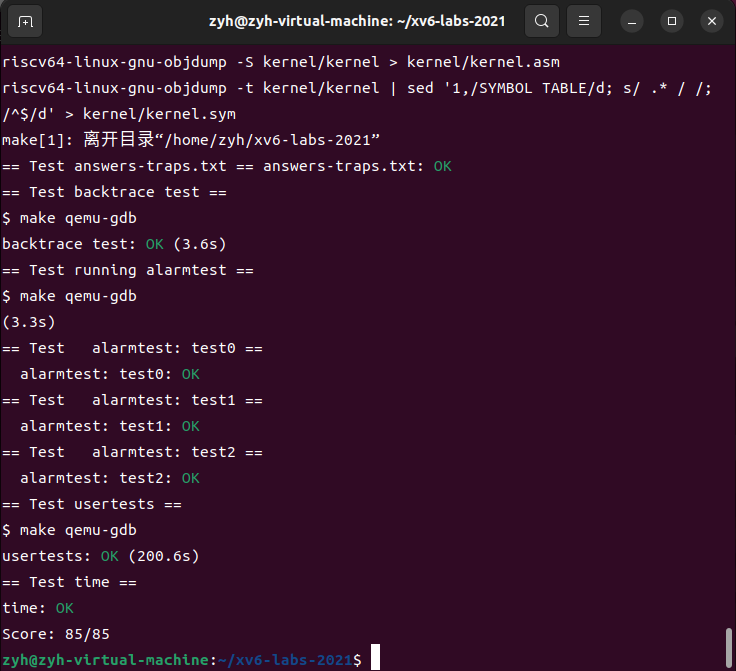
## 2.3实验结果



./grade-lab-traps backtrace



make grade



## 2.4代码分析&结果分析

### 2.4.1代码实现

根据提示，我们要怎么获取到当前正在执行的函数的栈帧指针呢？它其实就在s0寄存器里。然后，我们需要在kernel/riscv.h文件里添加一个函数：

static inline uint64

r\_fp()

{

uint64 x;

asm volatile("mv %0, s0" : "=r" (x) );

return x;

}

该函数的功能就是返回当前执行的函数的栈帧指针。这样，我们就能对栈里的栈帧进行遍历。

在我们拿到栈帧指针后，根据提示，栈帧指针偏移8个字节，就能获得函数的返回地址；偏移16个字节。就能获取更深层的栈帧的地址。注意，第一个栈帧占据着高地址，深层栈帧占据低地址。

然后就是遍历栈帧。提示告诉我们，可以使用PGROUNDDOWN(fp)和PGROUNDUP(fp)获取当前函数栈的地址范围。查阅手册可知，一个函数栈的大小与一个页表的大小一致，都是4KB。当我们知道了当前函数栈的地址范围，就能利用循环打印出所有栈帧的返回地址了。

对于输出的地址，可以在命令行输入addr2line -e kernel/kernel，然后输入地址，获取该地址对应的具体文件和具体行数。

下面是代码：

void backtrace()

{

uint64 fp = r\_fp(); // 栈帧指针

uint64 \*frame = (uint64 \*)fp; // 为方便地址偏移而定义，这样frame[-1]就表示从fp往低地址偏移8个字节

uint64 up = PGROUNDUP(fp);

uint64 down = PGROUNDDOWN(fp);

printf("backetrace:\n");

while (fp < up && fp > down)

{

printf("%p\n", frame[-1]);

fp = frame[-2];

frame = (uint64 \*)fp;

}

}

# 3 Alarm

## 3.1实验内容

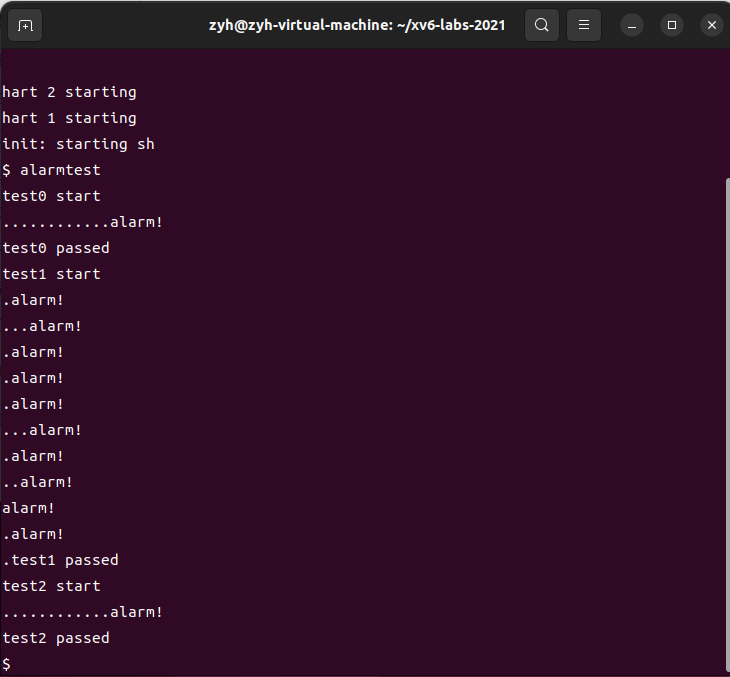
本次实验中，我们需要给xv6添加一个新特性，该特性是周期性的，每当一个进程执行一段时间后，都会执行一个函数。我们在实现这个特性时，需要借助中断机制。具体来说，我们需要实现一个叫sigalarm的系统调用，它接收两个入参，一个表示执行的时间周期，一个表示要周期执行的函数的函数指针。如果两个入参都是0，那么将不进行函数调用。实验提供了alarmtest和usertests来验证我们写的系统调用。

我们需要了解xv6的陷入机制。陷入机制是指，每当程序执行系统调用，或者程序出现了类似page fault、运算时除以0的错误，或者一个设备触发了中断使得当前程序运行需要响应内核设备驱动，都会导致CPU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。触发陷入机制的特殊情况称之为陷阱。

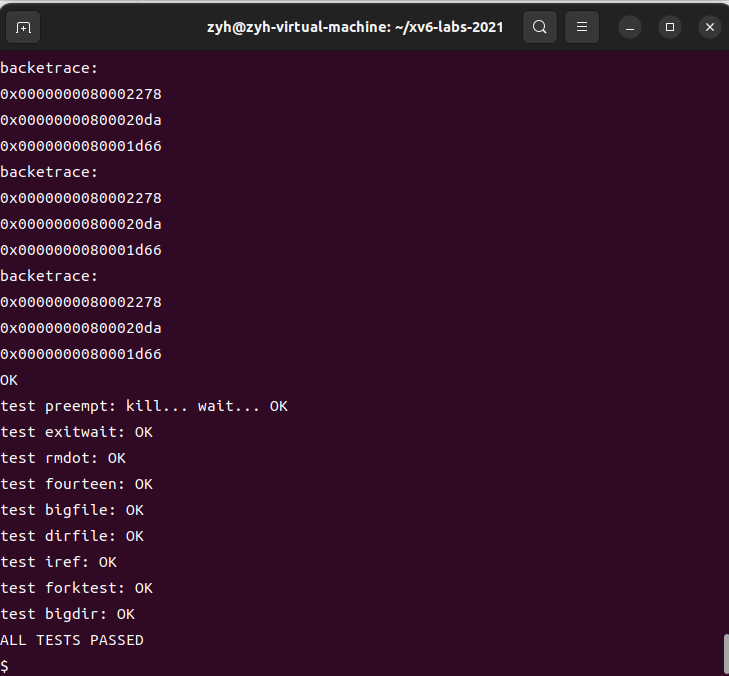
## 3.2代码位置与截图



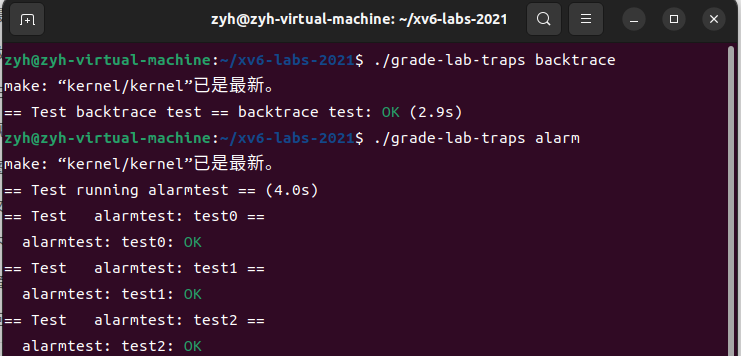
## 3.3实验结果



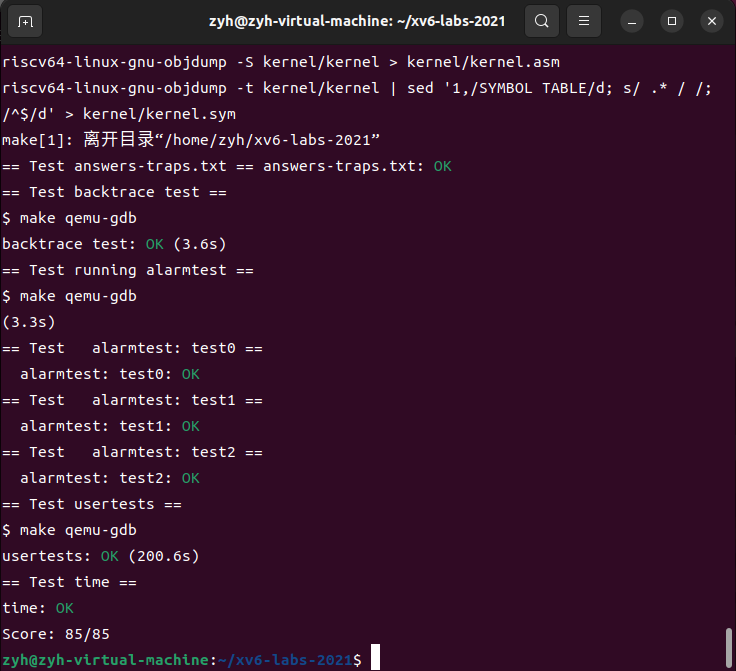
usertests的结果：



./grade-lab-traps alarm



make grade:



## 3.4代码分析&结果分析

### 3.4.1代码实现

根据提示，运行alarmtest我们需要通过test0、test1、test2三个小测试，分别解决它们我们就能完成这次实验。

test0需要我们实现搭建好基本代码框架，包括各种声明、函数的定义等等，同时需要我们实现定时执行指定的函数。sigalarm和sigreturn函数的定义在sysproc.c文件里实现，中断期间实现指定的函数，需要在trap.c文件里的usertrap函数的if(which\_dev == 2)里做修改。test0不需要我们恢复中断前的“现场”（包括各种寄存器的内容）。

usertrp函数是实现当出现trap时，从用户空间切换到内核空间的代码。which\_dev == 2用于判断trap的原因是不是时钟中断，包括轮转到期等等，我们的实现就要借助这个机制。

test1要求我们恢复中断前的“现场”。因为中断期间要执行的函数很可能会覆盖掉一些寄存器里的内容，所以为了中断结束后能正常执行原来的代码，我们需要保存“现场”。这可以通过在proc结构体里定义新的中间变量解决，这些变量保存各种寄存器原有的内容，然后在中断结束后又能通过它们还原寄存器。trapframe结构体包含了各种需要我们保存的信息，因此我们可以在proc结构体内再定义一个trapframe结构体，实现信息的保存。

test2要求我们在指定的函数执行完之前，不能再次中断并执行该函数。因为执行指定函数的时间有可能大于执行周期，因此我们需要防止在上一次执行完全前又尝试新的一次执行。这可以通给proc结构体添加一个标志位来实现。