Lab4

实验报告

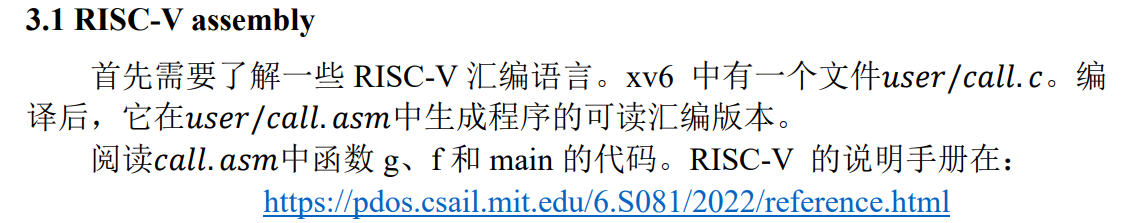
姓名 李晓畅

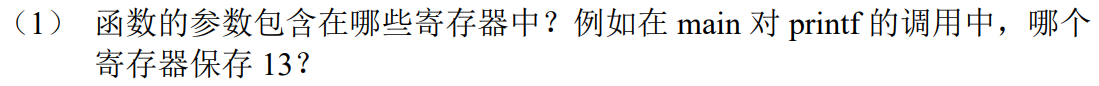
学号 20307130261

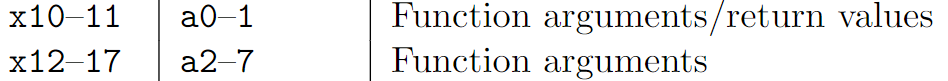
班级 计算机科学技术

# 实现思路

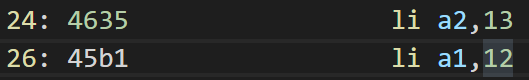
## 3.1 RISC-V assembly

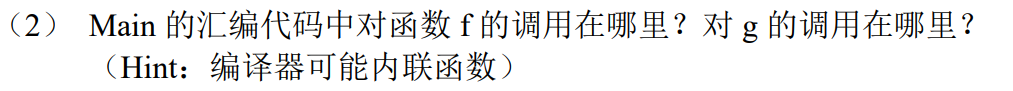


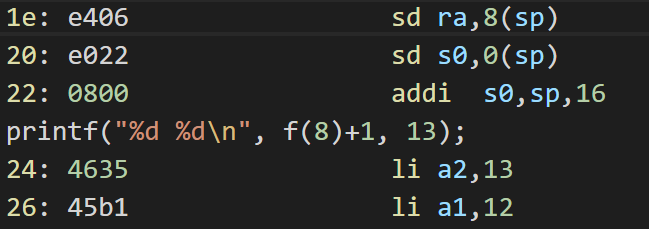
* 



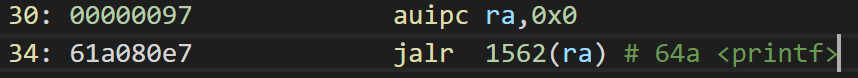
RISC-V为我们预留了a0到a7作为函数参数寄存器，这里涉及的函数只使用到了a0到a2。

具体在本例的main函数，不难看出13保存在a2寄存器中。

* 

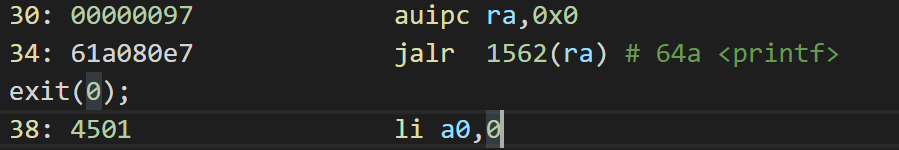
可以看出本例中并没有显式地调用f()和g(),而是直接算出了调用结果再加一的12。但是可以看出，为了准备函数调用，特意保存了返回地址寄存器的值到栈中。虽然最终没有什么用处。

* 

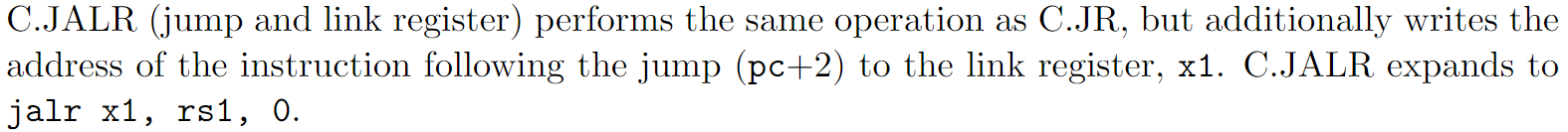


在0x30+0x0+1564=0x64a处

* 

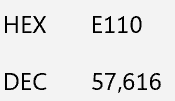


传入时ra=0x30。



因为调用的是jalr,跳转并链接，所以ra=ra+2\*4=0x38，也就是返回到紧接着的下一条指令地址。

* 

输出16进制整数应该不会受到大小端影响，毕竟只要理解一致进制转化应该是一样的，都是e110，输出He110。

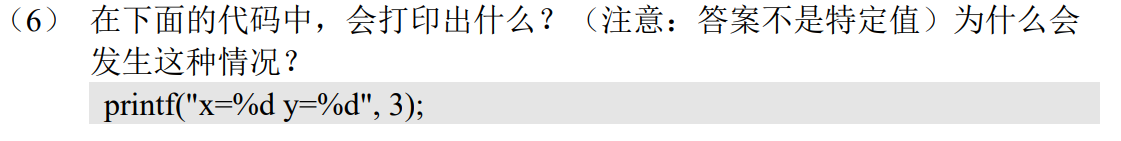
对于输出字符串，应该会受到大小端影响。

如果是小段模式，那么倒着翻译就是”led\0”，正好是一个字符串,打印输出Wold。

如果是大段，那么就是”\0del”,打印输出Wo。

结果就是这样的呢。

所以说如果是大端模式，i就要倒过来，变成：i=0x726c6400。

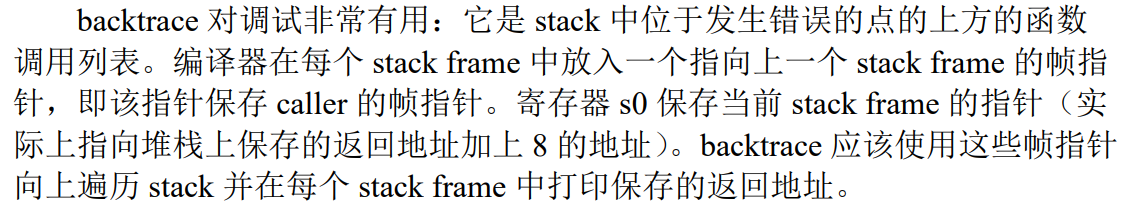
* 

说真的，编译就给报错。。除非屏蔽报错。

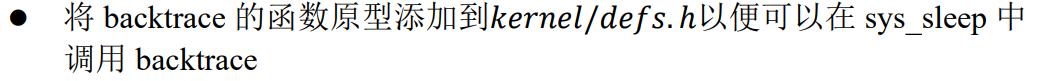
在实际运行中第二个输出y的值应该会同执行这条语句时a2的值有关。

因为***printf***还是会取a2作为第二个参数，但因为没有为它赋值，所以到底是什么取决于它之前是什么样子的。

## 3.2 Backtrace

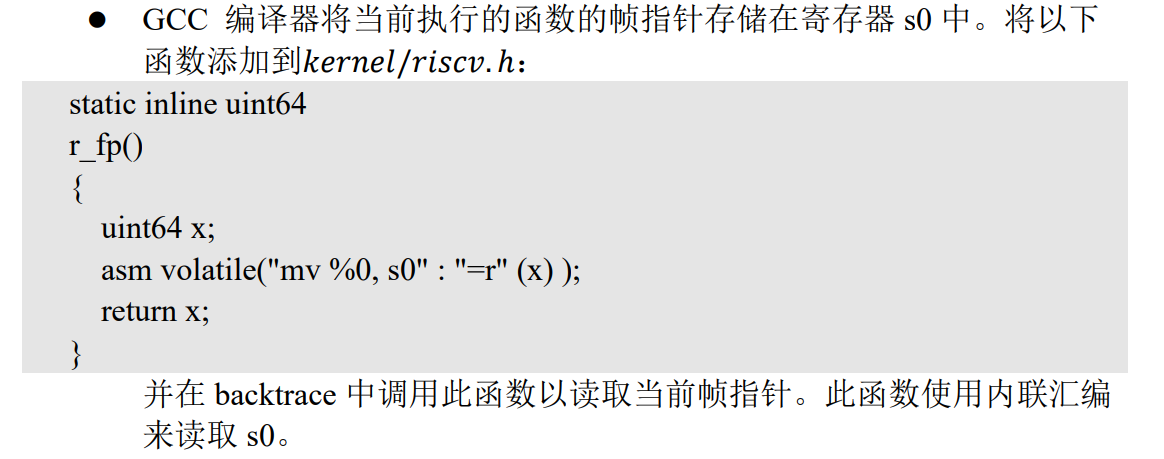


按照提示进行：

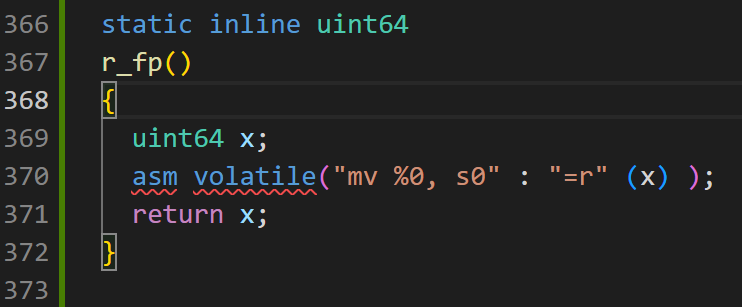
* 

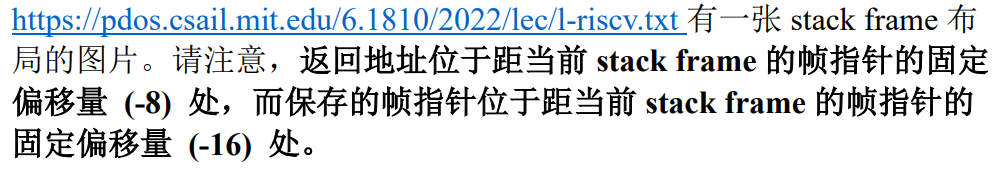
照例添加。

然后在sleep中也要添加。

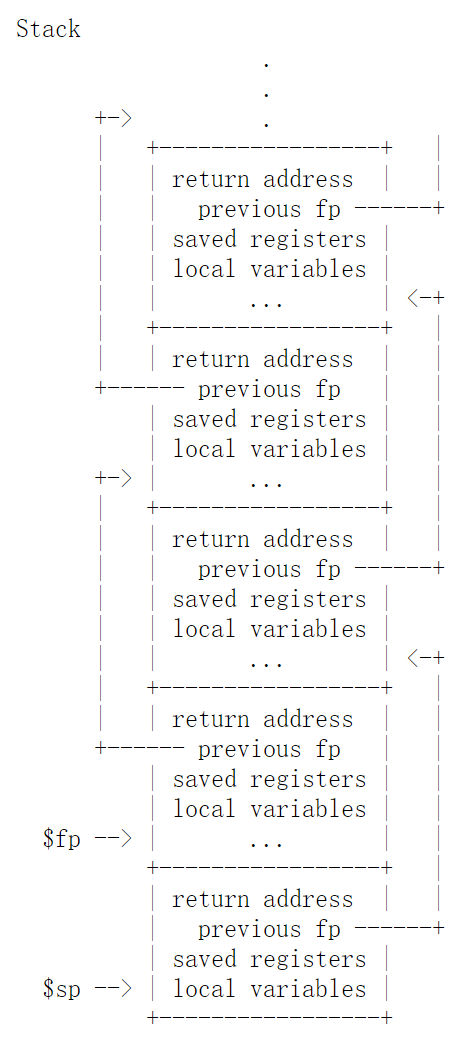
* 

添加函数。

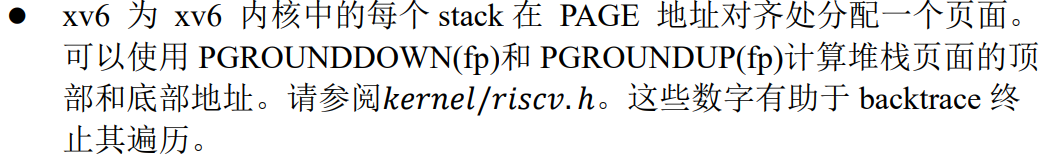


* 

他指的是这张指示栈结构的图：

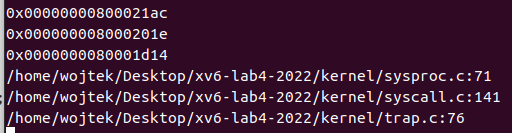
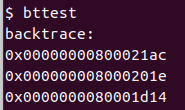


另外还有一个提示：



那要怎么实现就很容易了。

测试结果截图：



运行成功

## 3.3 ALARM

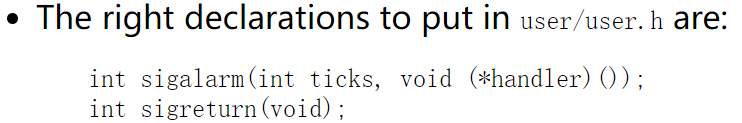


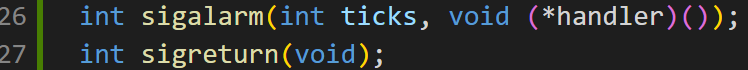
### Test0: invoke handler

按照提示进行：

* 

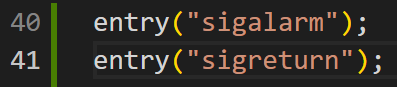
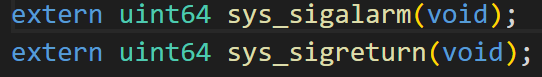
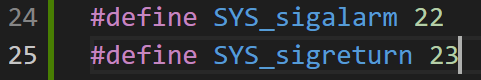
对应添加

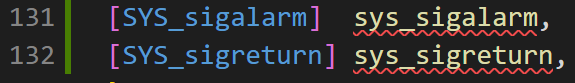
* 

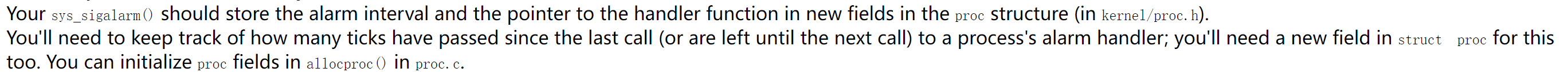


* 

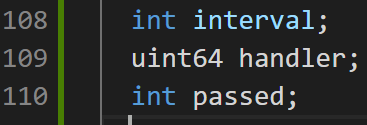
对应添加即可



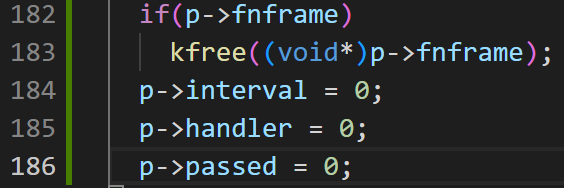
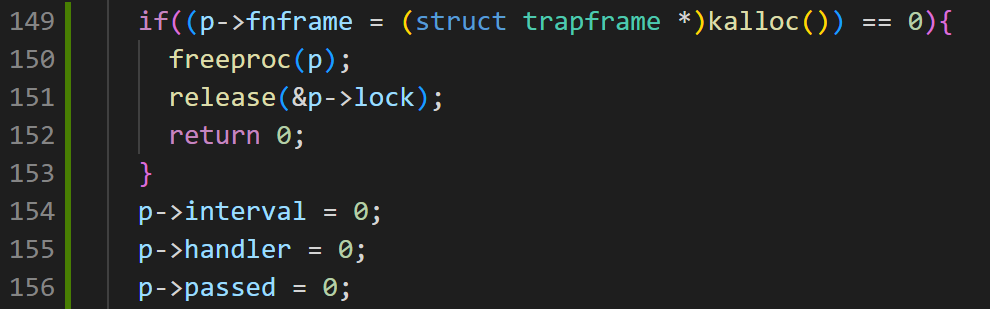
* 

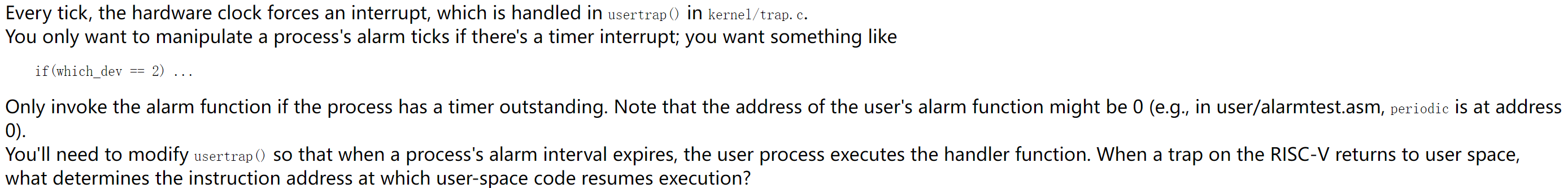
在proc中添加：



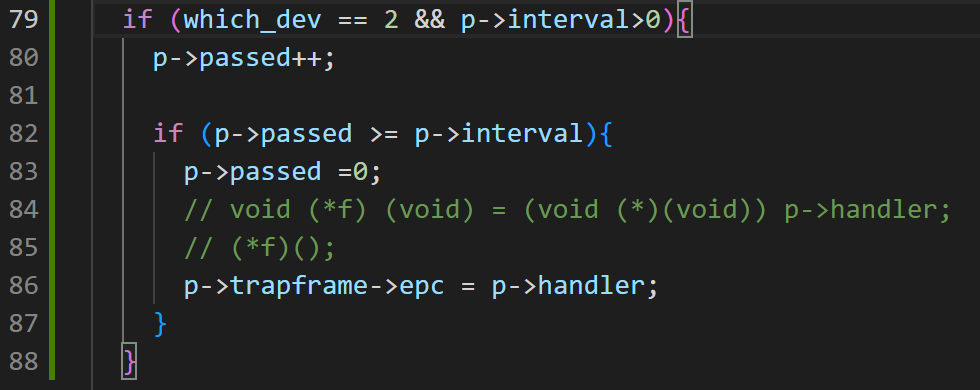
对应给出就行。

对应要分配和释放：



* 

大概可以这样写：



时钟tick一下就检查，如果要提醒用户的话就增加经过的tick数。

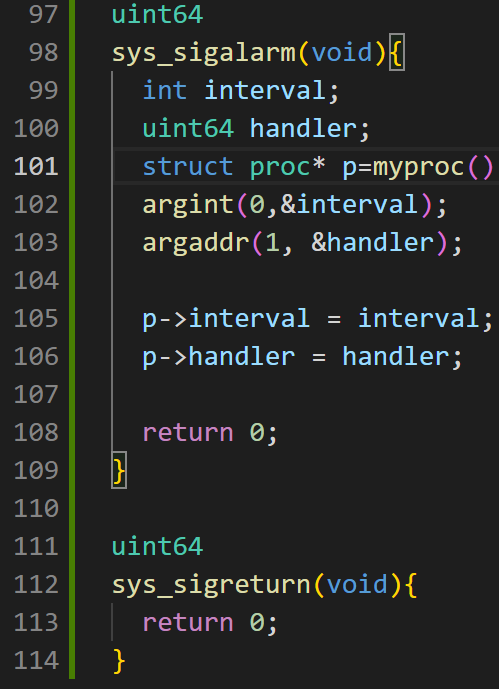
修改***trapframe***保存的程序计数器的值为handler函数地址，这样结束后会接着执行***handler***。

执行完handler我们也不考虑还原，可能有一些问题。再说吧。

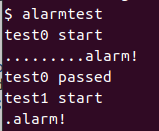
注释掉的部分有一些问题，是不对的。之后在问题环节细说。

* 

函数大概可以这样实现。***Sigalaem***就初始化参数就可以了。***Sigreturn***先直接返回0。



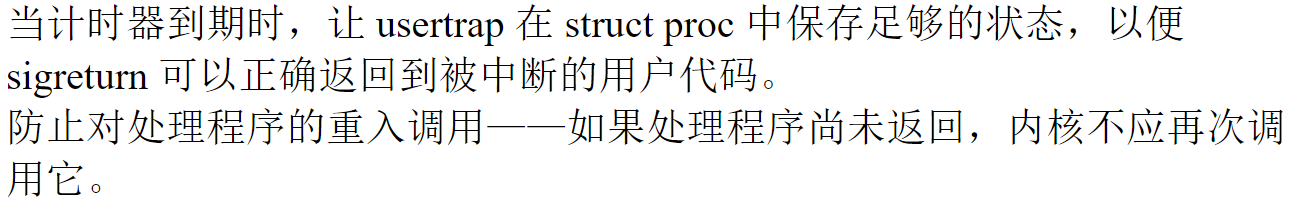
尝试运行：



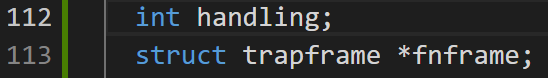
至少test0和我们预想的一样。其他当然是过不了的。因为没有恢复到之前状态。

### **test1/test2()/test3(): resume interrupted code**

按照提示进行：

* 

这要求我们在***proc***里面添加几个量。

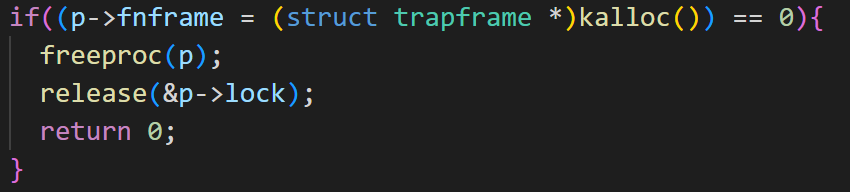
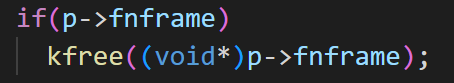


一个正在处理，防止重复执行。Xv6没给bool型，用一个整型。



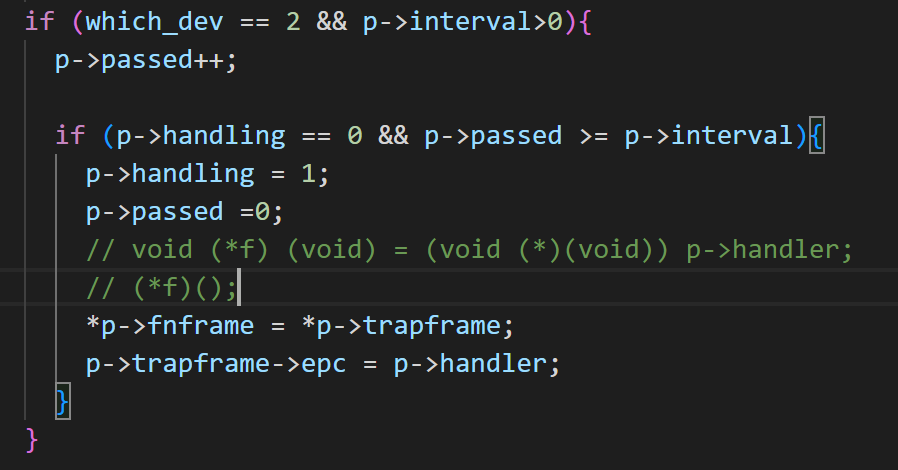
一个临时***trapframe***，临时保存***trapframe***，以备之后恢复。虽然我觉得我只修改了***epc***，但是我们要考虑到或许***handler***函数会用到很多预料之外的东西，所以全部保存应该是更好的。所以没必要单独保存寄存器。

然后对应的创建删除也要修改。

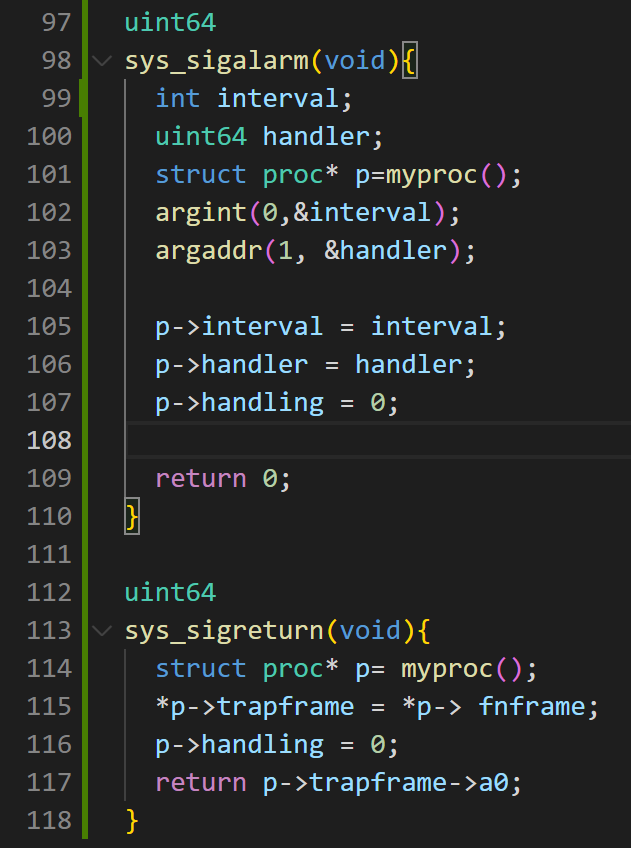
 

调用时要保存，返回时要恢复。

调用要检查是否在处理，需要处理要防止重复处理。因为这就一循环，不会并发，没必要考虑用更高级的东西。



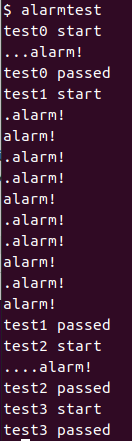
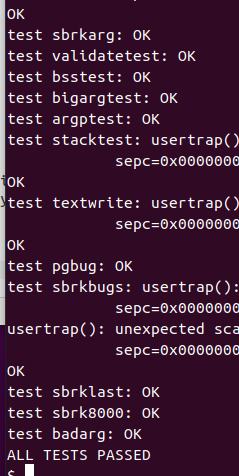
* 



最后修改一下***sigalarm***和***sigreturn***。

返回***a0***寄存器的值应该就能恢复***a0***了。

测试截图：

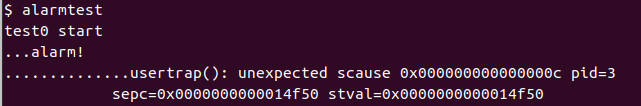
 

测试通过。

# 问题和解决

## 调试和检查

在test1修改后报了这样的错。



利用添加输出语句、GDB调试方法并加以分析，定位问题是***trapframe***恢复不正确。进一步发现是把拷贝结构体写成浅拷贝地址。

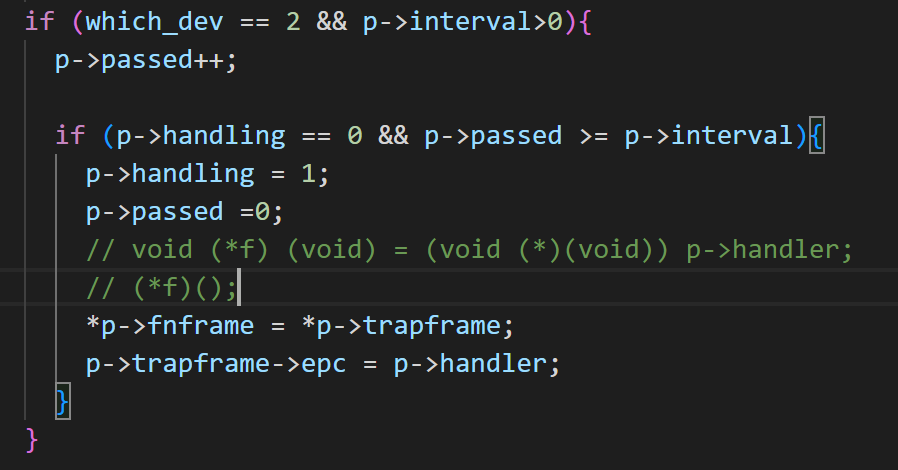


修改后即可。



其实不止可以赋值。调用***memmove, memcopy***这样的函数应该也可以实现对应的功能。

## 可以直接调用函数吗？



上面提到过，注释掉的代码有问题。

其原因在于，我们得到的一个函数指针是进程中的逻辑地址，但是在内核中地址上可不一定能找到对应的函数。

如果非得这么做的话，可以考虑给物理地址，像***trampoline***那样实现。

或者直接在***sigreturn***里面调用。

但是这里有一点问题在于即便这样做，也要准备好恢复寄存器。那还不如直接改***epc***就可以了。

# 实验总结

在本次实验中我们主要学习了trap的有关内容。

用户空间和内核空间的切换通常被称为trap。当程序执行系统调用、出现例如page fault的故障、外设触发中断时，都会trap进内核中。trap涉及了许多小心的设计和重要的细节，这些细节对于实现安全、隔离和高性能来说非常重要。

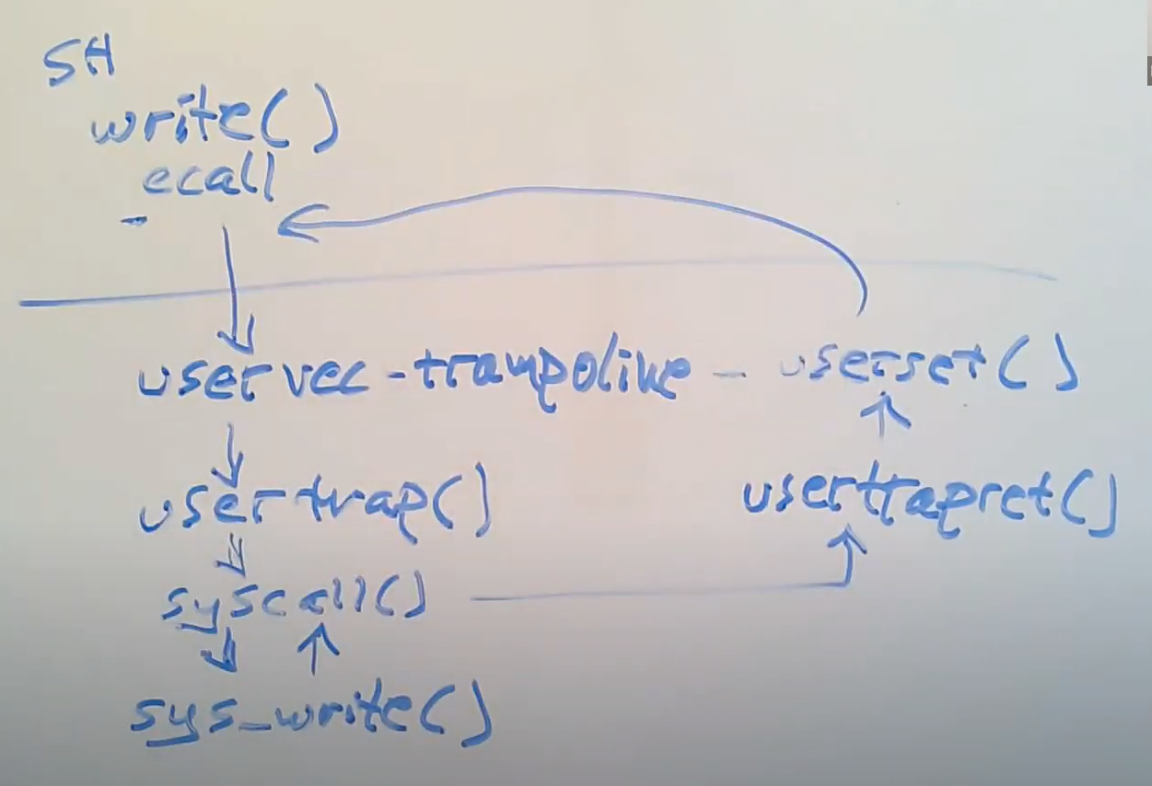
值得注意的一点是硬件状态的转换。用户态下可能遇到trap，但只有在内核态下我们才能处理trap。所以硬件状态的转化是很重要的。

实际要处理trap需要一系列操作。

* 首先，我们需要保存32个用户寄存器。因为很显然我们需要恢复用户应用程序的执行，尤其是当用户程序随机的被设备中断所打断时。我们希望内核能够响应中断，之后在用户程序完全无感知的情况下再恢复用户代码的执行。所以这意味着32个用户寄存器不能被内核弄乱。但是这些寄存器又要被内核代码所使用，所以在trap之前，你必须先在某处保存这32个用户寄存器。
* 程序计数器也需要在某个地方保存，它几乎跟一个用户寄存器的地位是一样的，我们需要能够在用户程序运行中断的位置继续执行用户程序。
* 我们需要将mode改成supervisor mode，因为我们想要使用内核中的各种各样的特权指令。
* SATP寄存器现在正指向user page table，而user page table只包含了用户程序所需要的内存映射和一两个其他的映射，它并没有包含整个内核数据的内存映射。所以在运行内核代码之前，我们需要将SATP指向kernel page table。
* 我们需要将堆栈寄存器指向位于内核的一个地址，因为我们需要一个堆栈来调用内核的C函数。
* 一旦我们设置好了，并且所有的硬件状态都适合在内核中使用， 我们需要跳入内核的C代码。

而在内核中主要包括了***uservec***, ***usertrap***, ***usertrapret***和***userret***四个函数。

所以总的来说就是这样的流程：



（参考[Lec06 Isolation & system call entry/exit (Robert) - MIT6.S081 (gitbook.io)](https://mit-public-courses-cn-translatio.gitbook.io/mit6-s081/lec06-isolation-and-system-call-entry-exit-robert)）