**Lab4实验报告**

一、实验思路、相关代码截图、结果截图

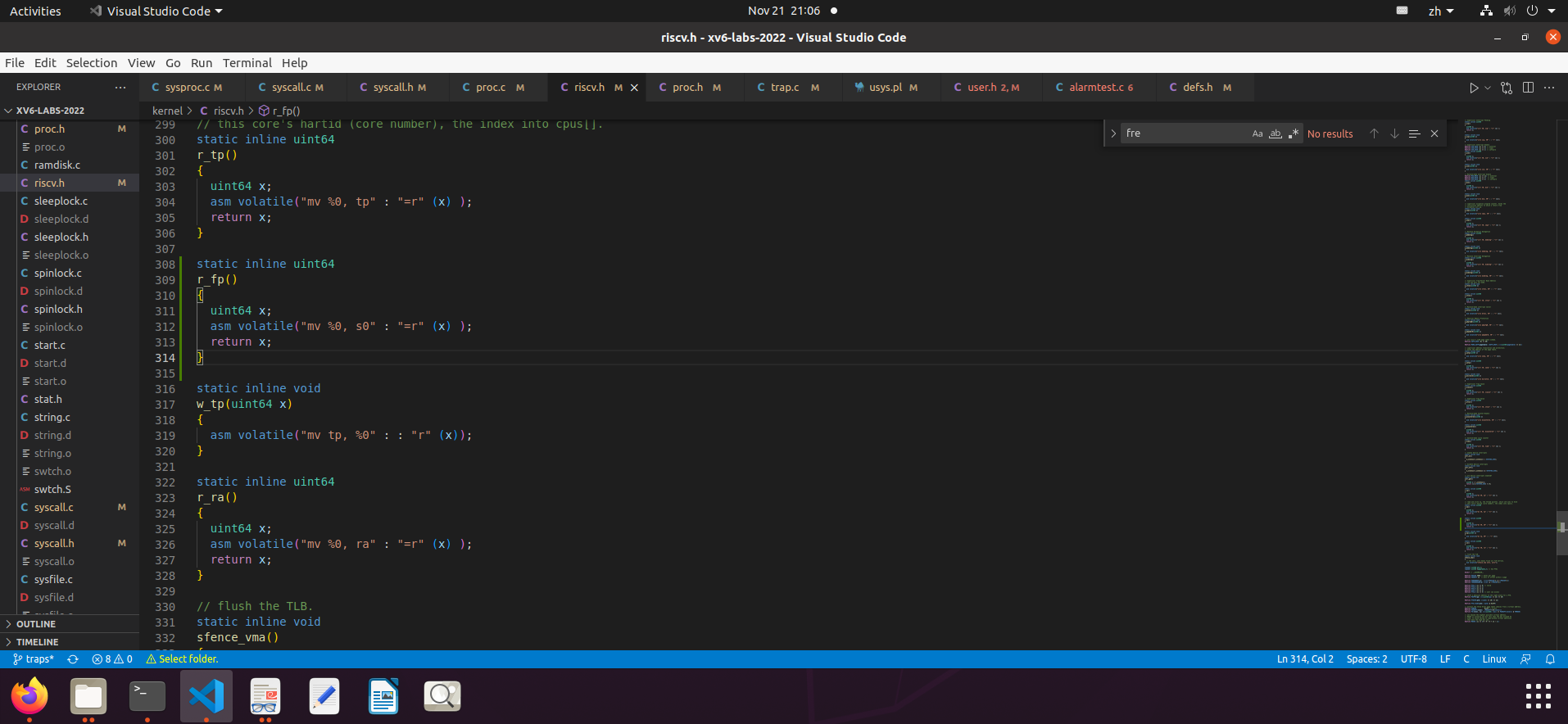
Backtrace

**实验目的：** 实现 backtrace 功能，用于打印出栈上调用链的所有返回地址

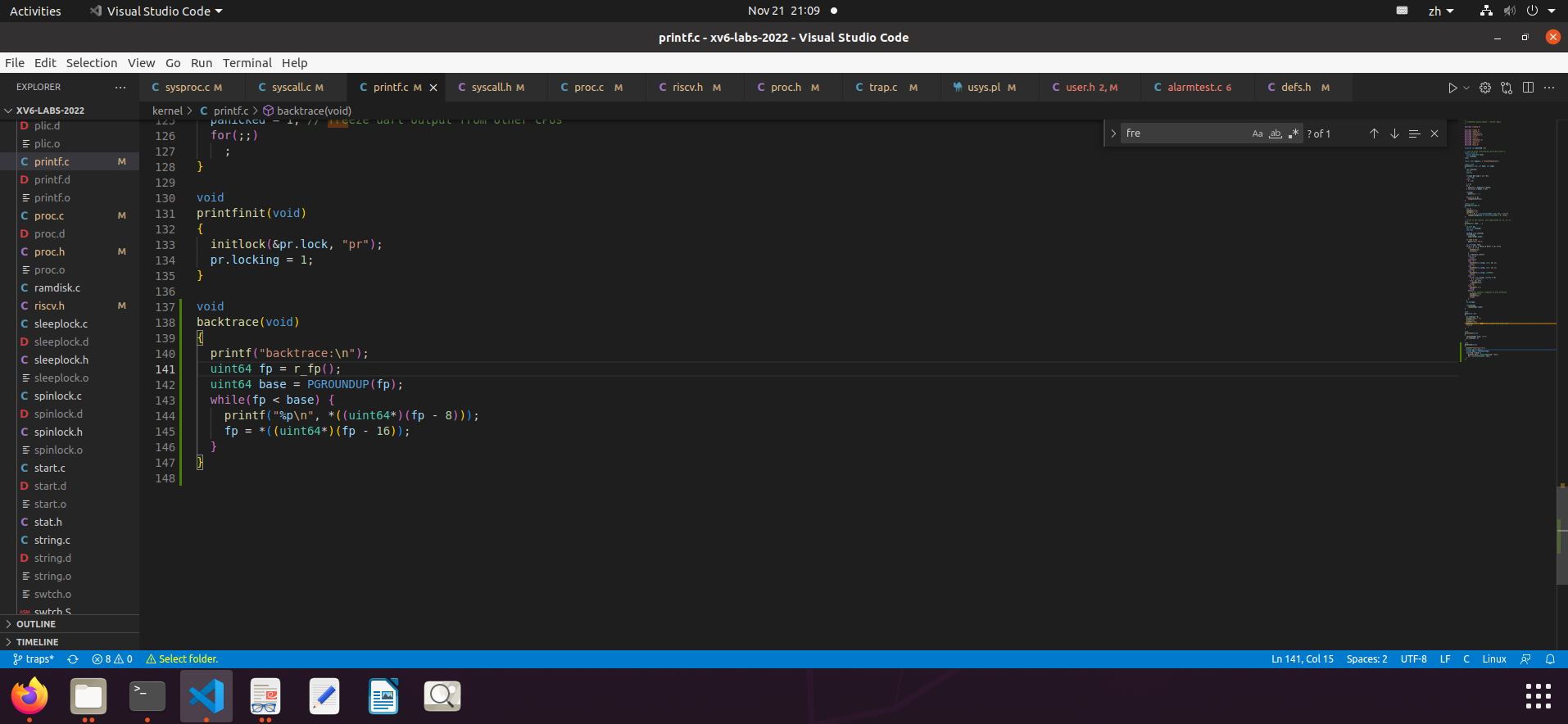
**实验思路：**

首先根据提示在riscv.h中添加函数r\_fp（）：

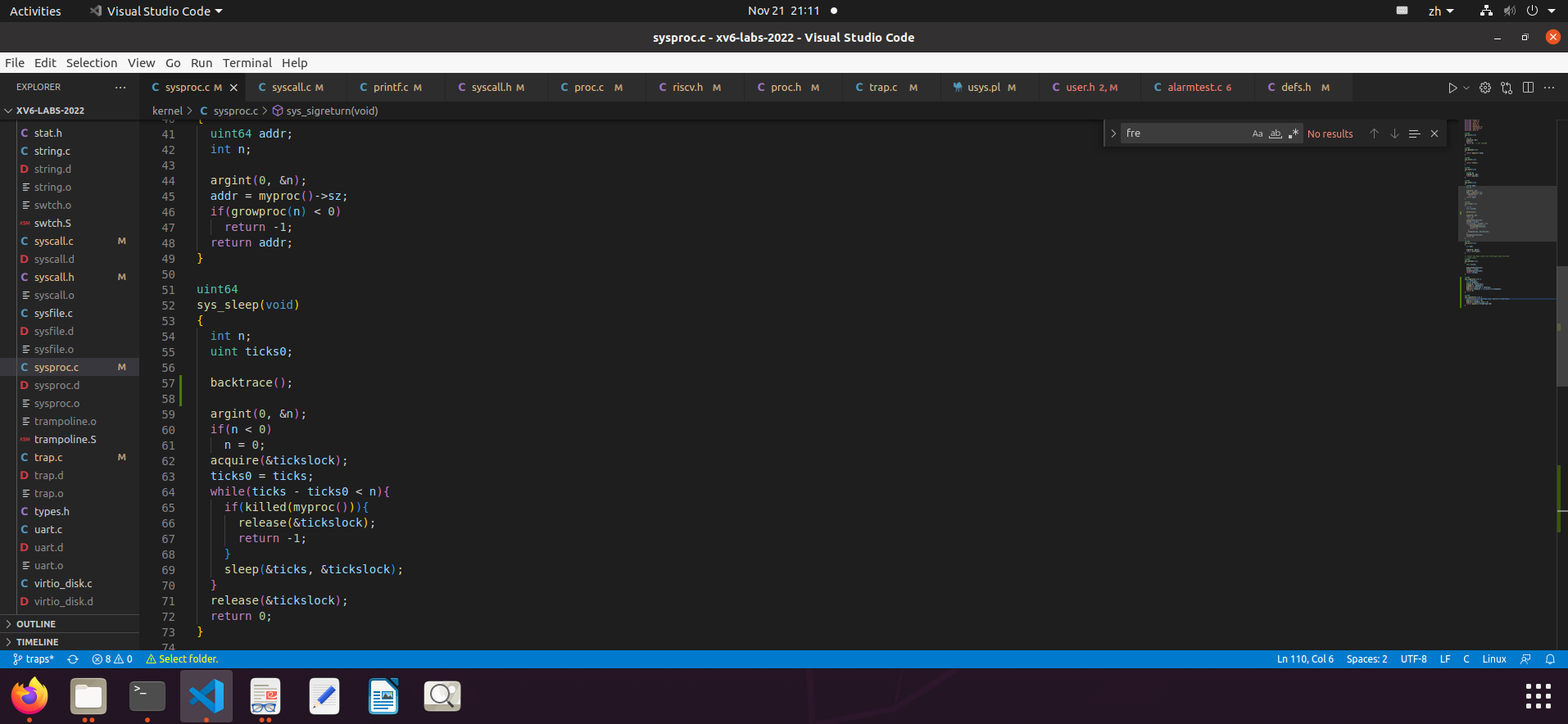
已知当前函数的帧指针存在寄存器 s0 中，在 kernel/riscv.h 中添加获得该值的函数。



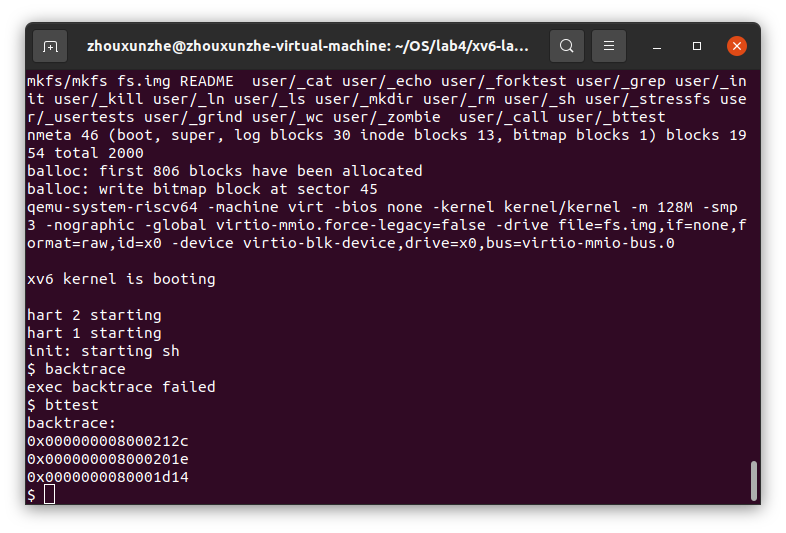
根据提示，实现backtrace函数，首先输出一个backtrace的字段，然后获取当前的函数，并且使用 PGUPGROUND() 获得当前栈的顶部地址；由 RISC-V 的栈结构可知，返回地址存放在帧指针的 -8 偏移量处，保存的帧指针存放在帧指针的 -16 偏移量处。



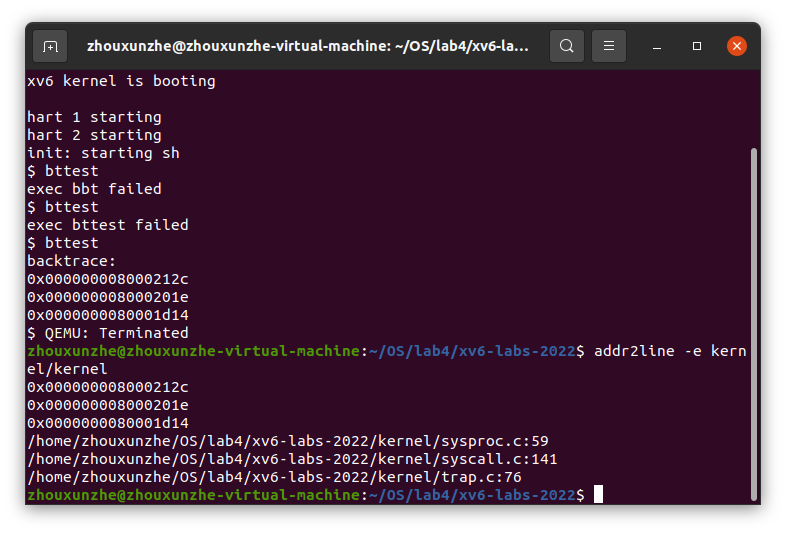
最后在sys\_sleep（）中调用backtrace函数即可。



执行bttest：

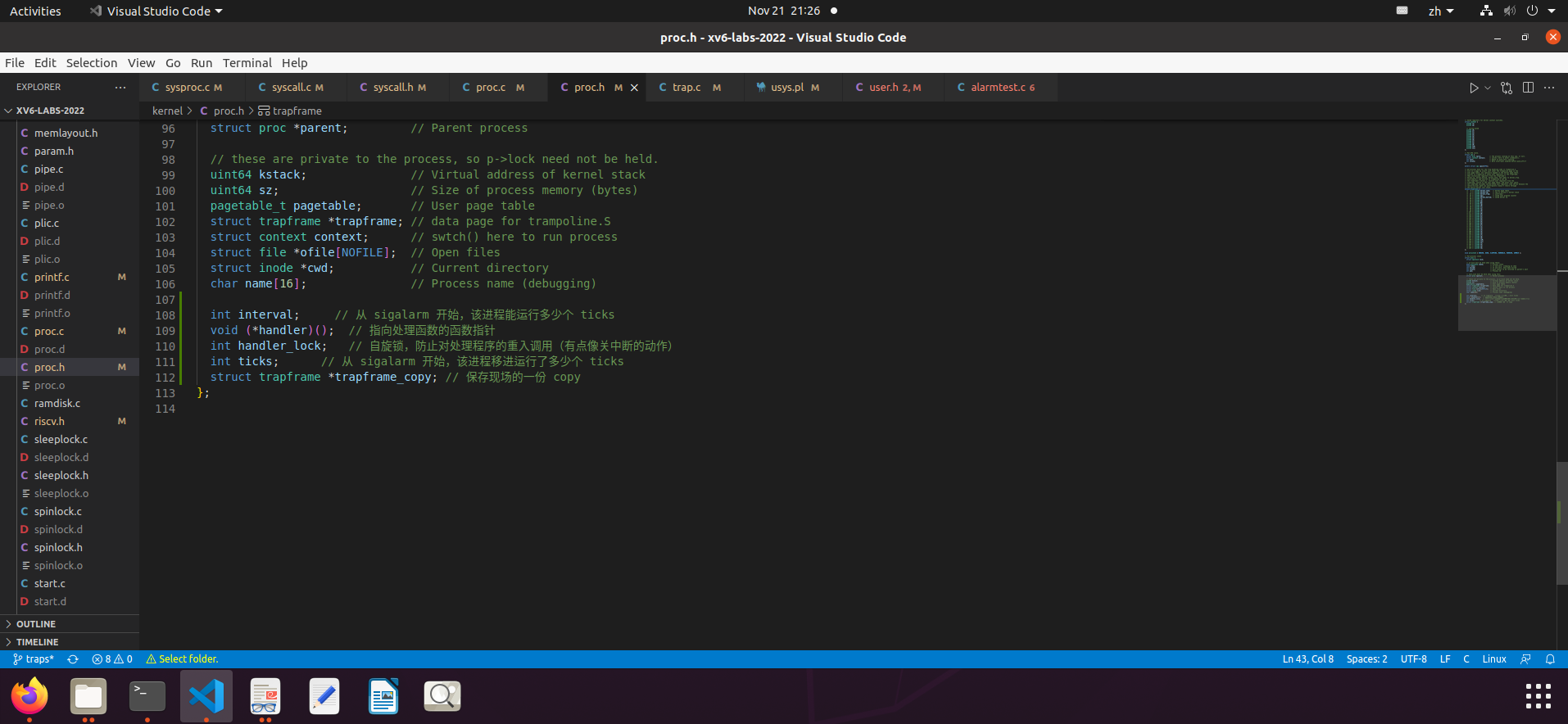


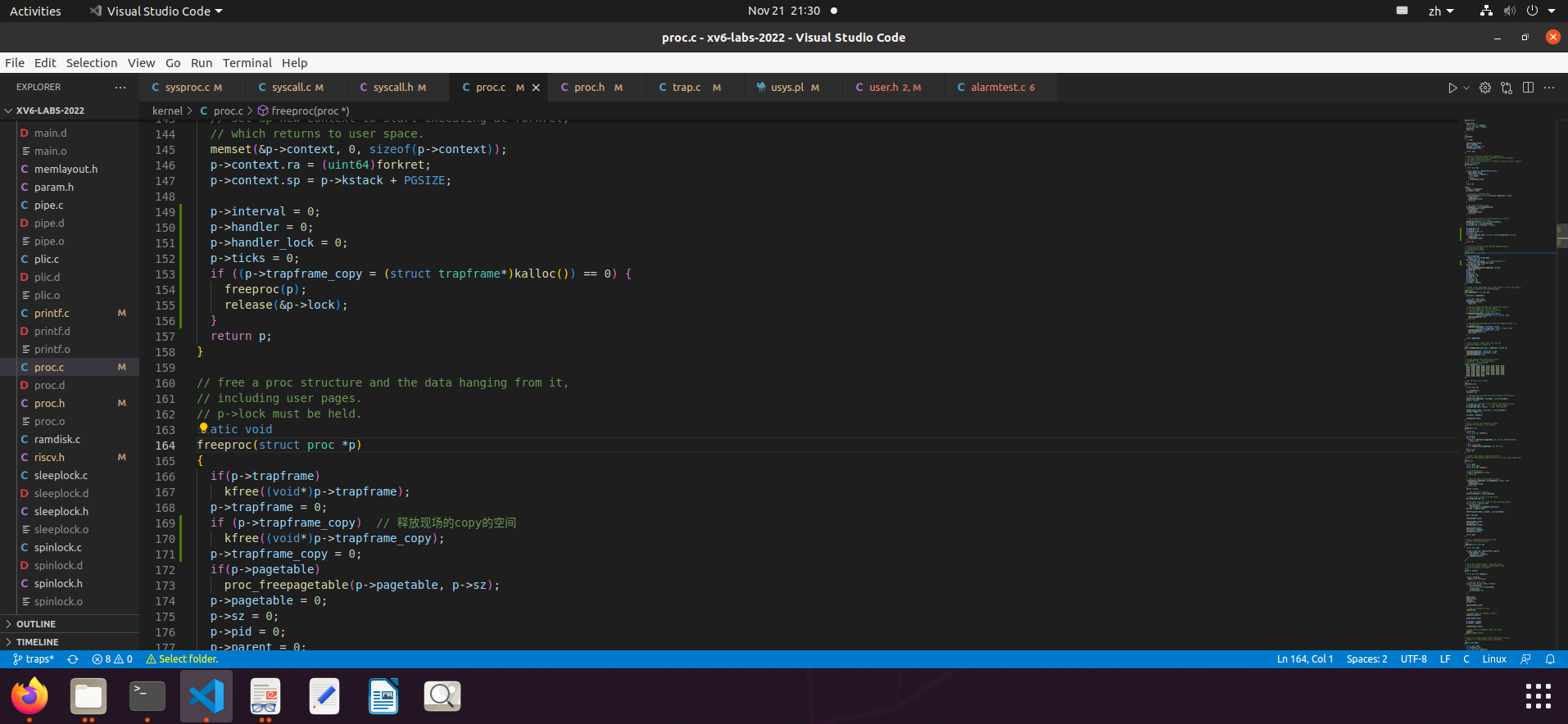
可以看到输出三个地址，然后执行 addr2line -e kernel/kernel ，查找地址对应的代码：

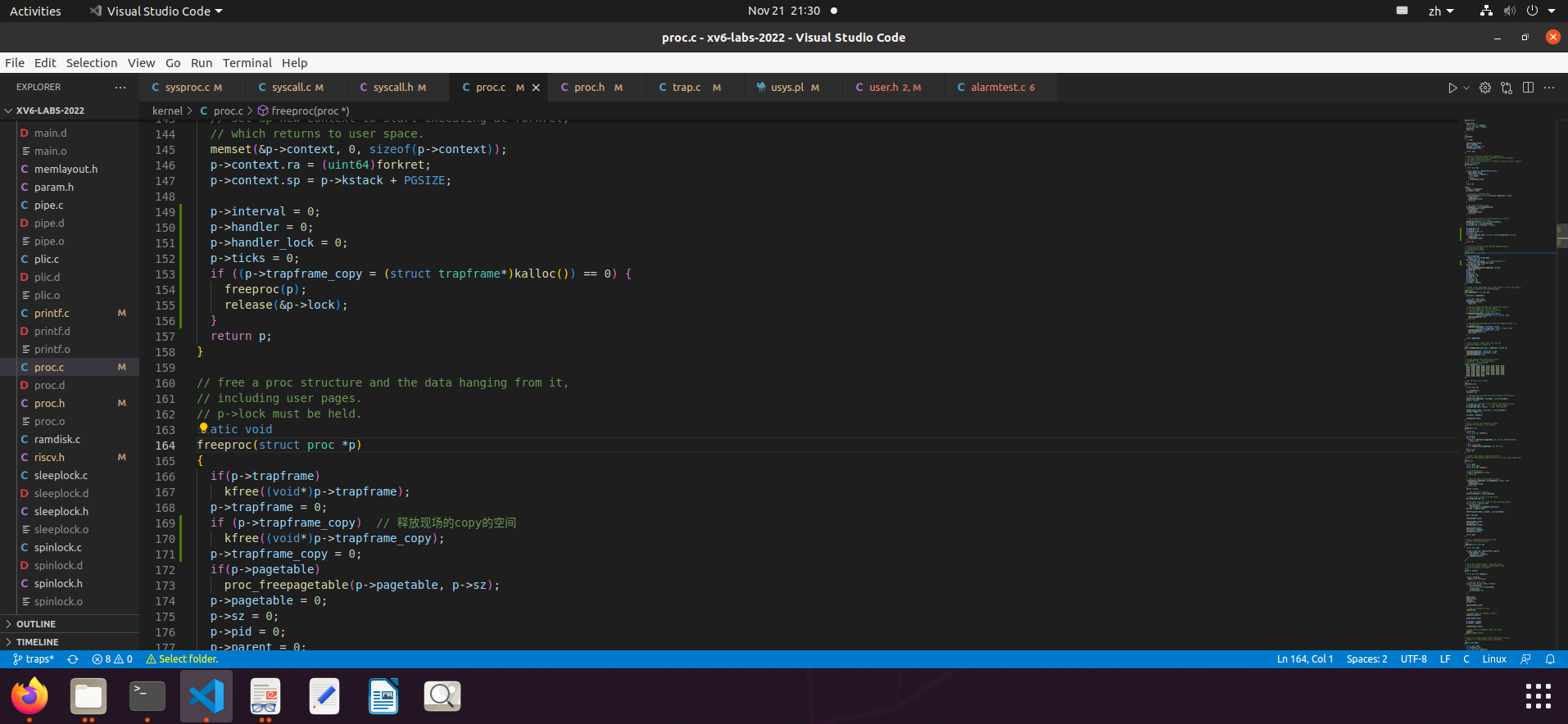


Alarm

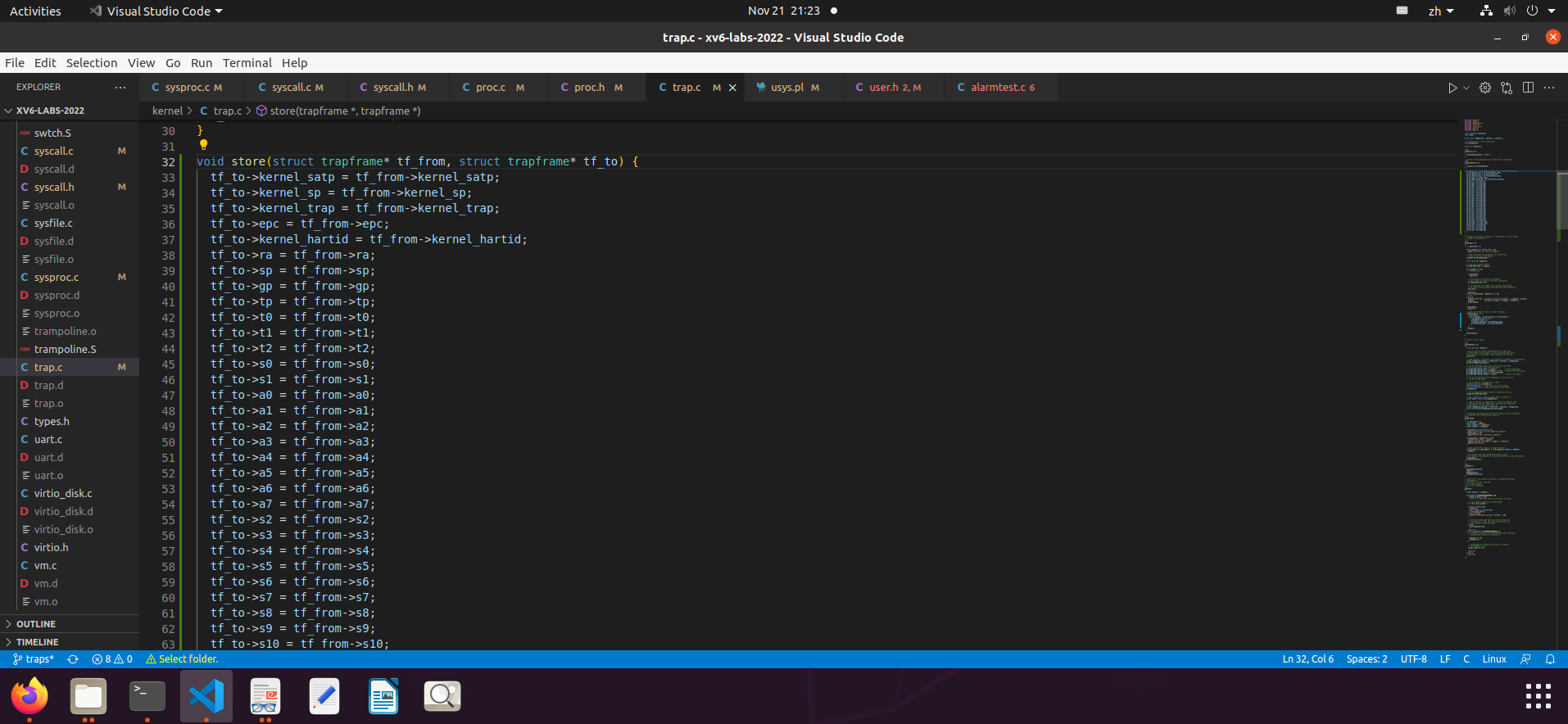
首先在 struct proc 中添加相关变量，并添加构造和析构动作，用于辅助 sigalarm() 和 sigreturn() 系统调用实现，由于需要实现寄存器的值在调用sys\_sigreturn()后仍然保持不变，所以需要使用页来存储。







根据提示，实现一个用来存储和导出寄存器的值的函数。



由于需要存储36个寄存器以及类中变量的值，所以工作量会很大。

然后为 sigalarm() 和 sigreturn() 系统调用添加相关声明

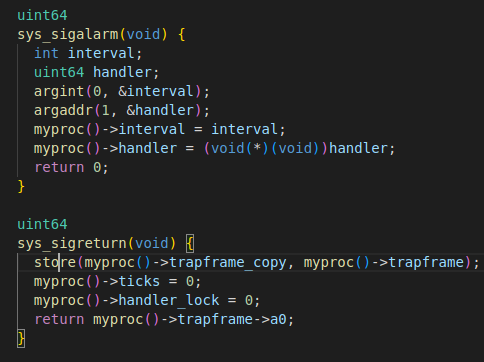




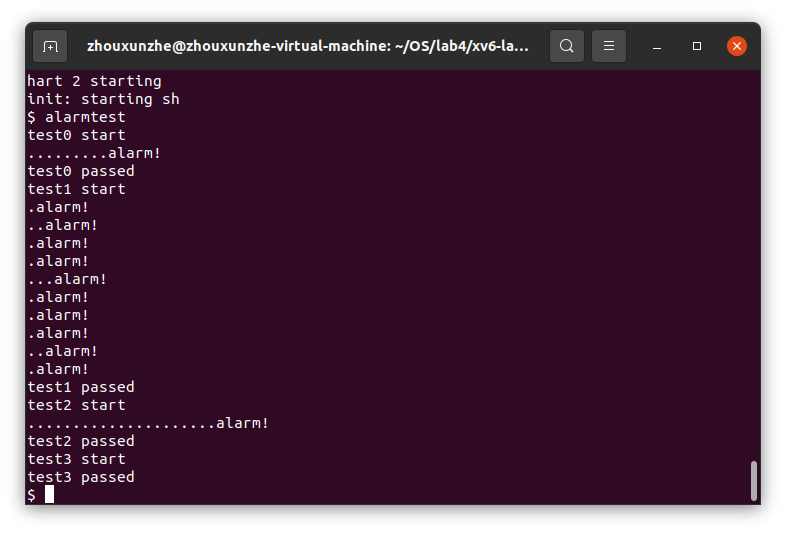


根据提示，实现 sys\_sigalarm() 和 sys\_sigreturn() 系统调用

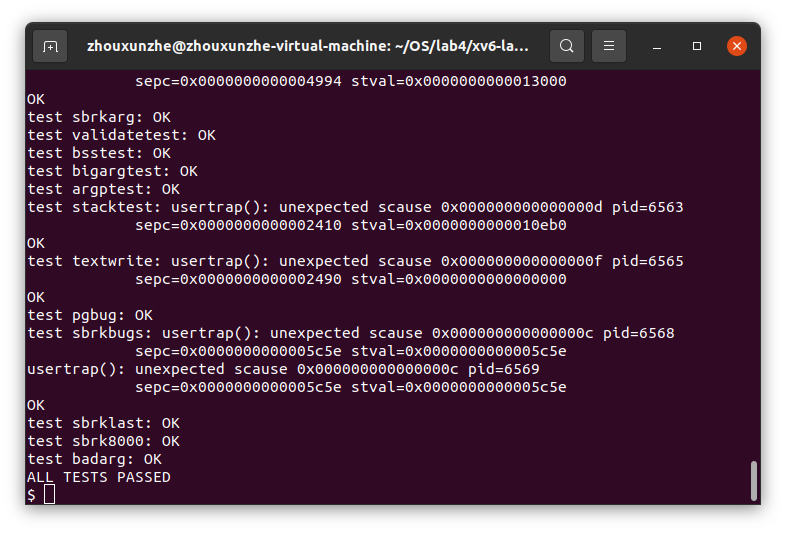


其中，sys\_sigalarm()要从 a0 和 a1 获得用户调用系统调用时传入的 ticks 数和处理函数，并且用这些值更新 struct proc 里的成员。sys\_sigreturn() 需要恢复现场、清除刚刚设置的成员、解除对处理函数的锁；由于中断处理对用户来说应当是透明的，当 sys\_sigreturn() 返回值时会更新 a0 寄存器的值，因此直接让其返回当前 a0 的值以保证不变。

进行测试：



然后使用usertests -q测试是否更改kernel：



二、实验中遇到的问题，如何思考并解决

实验中首先遇到的问题就是由于一开始对于RISCV栈结构不了解，所以在实现backtrace是，对于PGUPGROUND()的使用并不是很理解，后续研究其栈结构后问题解决。

然后在实现alarm中，一开始，并没有使用页表结构进行系统调用，然后在测试test3时，会出现报错register a0 changed，仔细思考，原因是，没有用页表结构进行存储。所以寄存器都是存在栈中，当系统调用后，其值就会改变。了解原因后，增加页表进行系统调用，问题解决。

三、实验总结

在本次实验中，我学会了trap的原理以及如何通过系统调用观察其流程。除此而外，还通过gdb调试了解了riscv的基本结构，通过backtrace实验，学会了如何观察系统中断，通过alarm实验，学会了如何利用寄存器进行系统调用输出特定函数系统中断所需要的tick数。总的来说，本次实验增强了我对riscv的结构的了解，也让我更加理解操作系统中系统中断的原理。