**实验报告**

**Lab 4**

姓名：施君豪

班级：20 人工智能

学号：20307140008

**实验报告填写要求**

1.请在每个exercise之后简要叙述实验原理，详细描述实验过程。可以使用中文进行描述，不对语言做要求。

2.请将你认为的关键步骤附上必要的截图。

3.有需要写代码的实验，必须配有代码、注释以及对代码功能的说明。

4.你还可以列举包括但不局限于以下方面:实验过程中碰到的问题你是如何解决的、实验之后你还留有哪些疑问和感想。

5.如果实验附有练习，请在每个练习之后作答，这是实验报告评分的重要部分。

6.Challenge为加分选作题。每个lab可能有多个challenge,我们会根据完成情况以及难度适当加分，具体情况会在课上说明。这部分的实验过程描述应该比exercise更加详细。

7.切勿抄袭亦或是去互联网复制粘贴答案。

【练习题模板】

1. Question
2. Code
3. Screenshot
4. Difficulties and solutions

实验练习一：**RISC-V assembly**

**It will be important to understand a bit of RISC-V assembly, which you were exposed to in 6.1910 (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.**

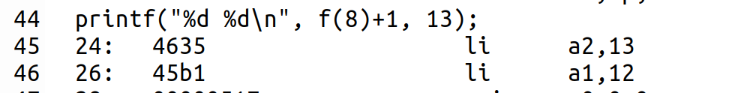
本题无代码，仅问题回答

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？

保存在寄存器a0 - a7中, a2保存13

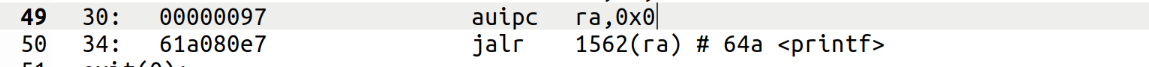
1. Main 的汇编代码中对函数 f的调用在哪里？对 g 的调用在哪里？

第46行，编译器进行了函数内联, 直接将 f(8)+1的值 12 计算出来。



1. 函数 printf位于哪个地址？

由第 49 和 50 行可以看出, jalr 跳转的地址为 0x30+1562=0x64a, 即函数 printf 的地址为 0x64a



1. 在 jalr 到 main 中的 printf之后，寄存器 ra 中存储的值是？

值为0x38，为下一条指令的地址。

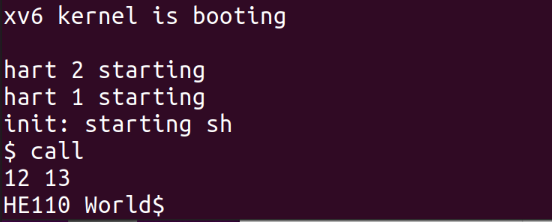
1. 运行以下代码：

unsigned int i = 0x00646c72;

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

输出是什么？

HE110，World



补：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 i，57616 为不同的值？

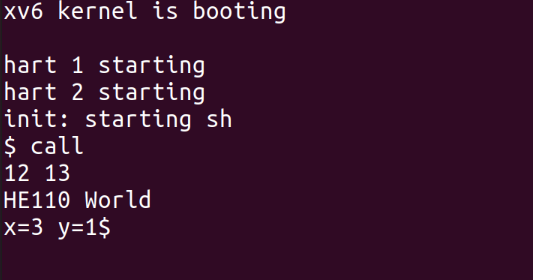
需要将i的值改为：0x726c6400，不需要更改57616，因为57616是一个值，二进制读取时会进行更改。（00/64/62/72——72/6c/64/00）

1. 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);

会打印出x = 3，y = 1。

原因为根据函数的传参规则, y= 后跟的值应该为寄存器 a2 的值。虽然c文件中没有提供足够的参数，但在进行编译后函数执行时仍然从原本参数应该加载的寄存器取值。按照 RISC-V 的传参规则, 第二个不定参数应该被存于寄存器 a2, 因此在实际输出时也是将 a2 寄存器的值进行输出。



实验练习二：**Backtrace**

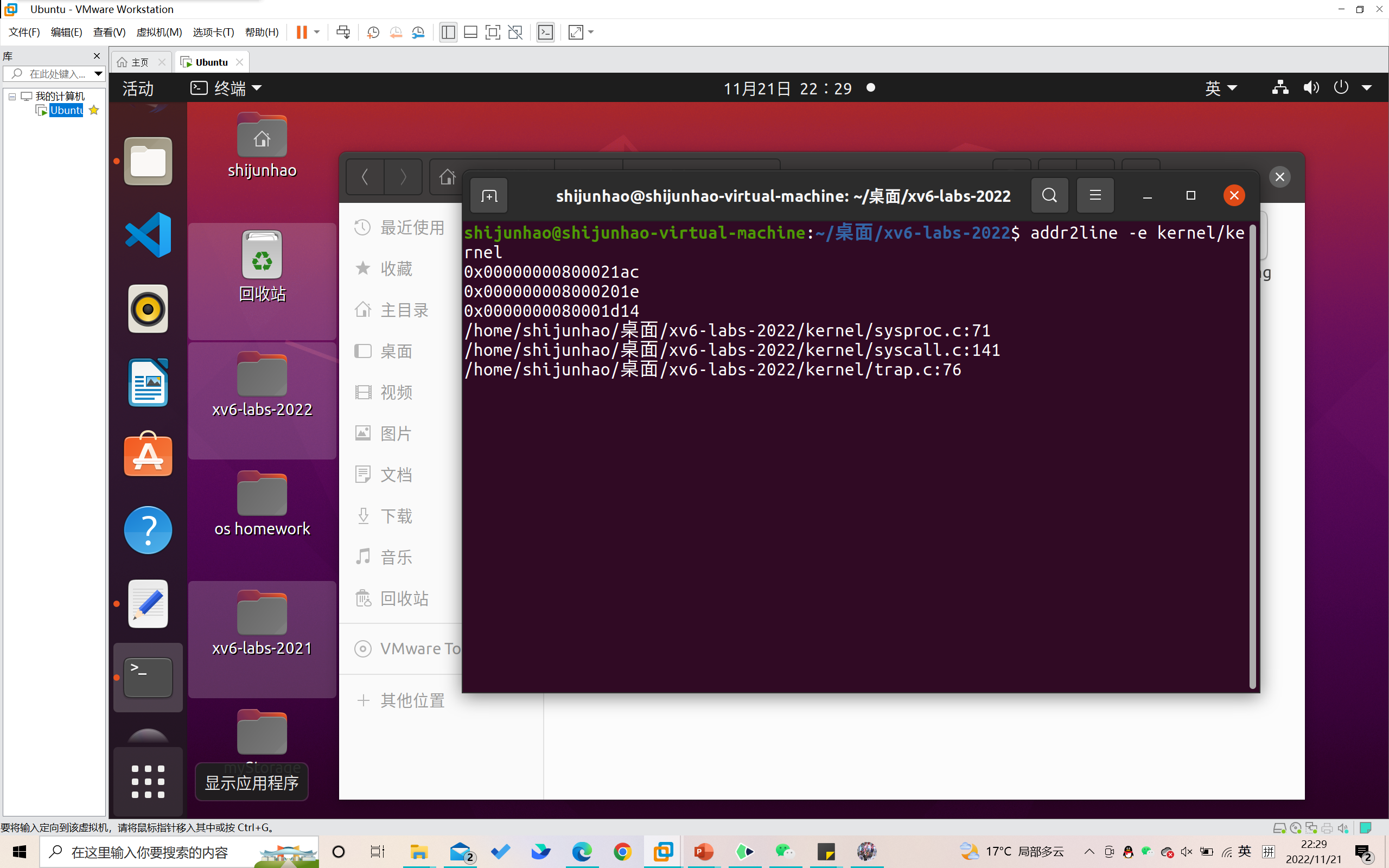
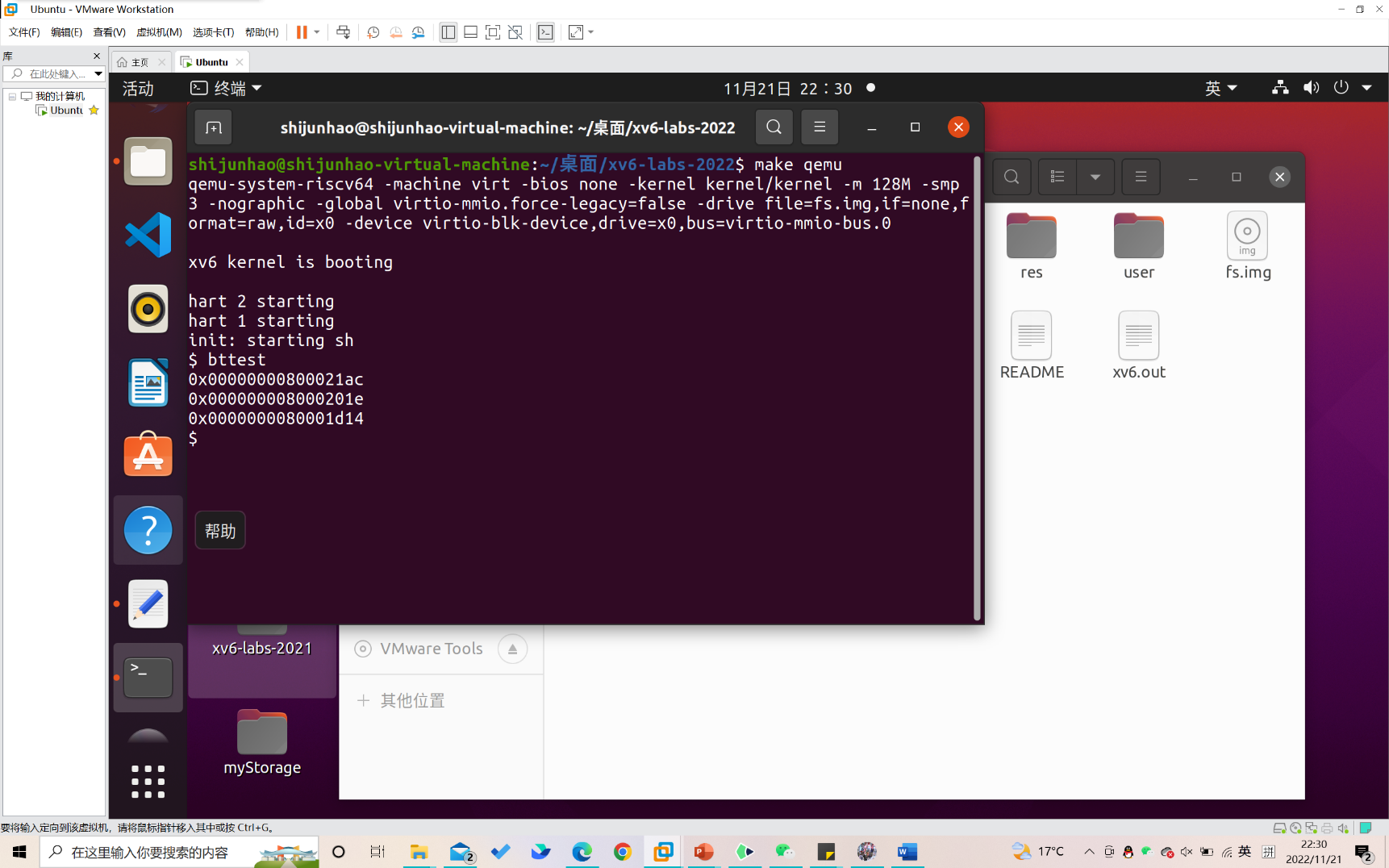
**backtrace 对调试非常有用：它是 stack 中位于发生错误的点的上方的函数 调用列表。编译器在每个 stack frame 中放入一个指向上一个 stack frame 的帧指 针，即该指针保存 caller 的帧指针。寄存器 s0 保存当前 stack frame 的指针（实 际上指向堆栈上保存的返回地址加上 8 的地址）。backtrace 应该使用这些帧指针。向上遍历 stack 并在每个 stack frame 中打印保存的返回地址。**

一、思路及代码：

通过调用hint中已经给出的 r\_fp() 函数来读取寄存器 s0 中的当前函数栈帧 fp。 根据 RISC-V 的栈结构, fp-8 存放返回地址, fp-16 存放原栈帧.。因此可以进而通过原栈帧得到上一级的栈结构, 如此往复直到获取到最初的栈结构。具体的停止条件为初始从寄存器 s0 读取到的栈帧，读到该帧说明已经返回最初，结束递归。

具体代码详见附件。

1. 屏幕截图



1. 困难与解决办法

由于数据类型的不同，该问中涉及到了uint类型的强制转换。一开始没意识到从而导致得不出理想的结果，和同学探讨之后得以解决。

实验练习三：**Alarm**

**本练习中将向 xv6 中添加一个功能，该功能会在进程使用 CPU时定期提 醒它。这对于想要限制它们占用CPU 时间的计算密集型进程，或者对于想要 计算但又想要采取一些定期操作的进程可能很有用。更一般地说，你将实现一 种原始形式的 user-level interrupt/fault handler；例如，你可以使用类似的东西来 处理应用程序中的 page fault。 如果你的测试通过了 alarmtest 和‘usertests -q’，那么结果就是正确的。 你应该添加一个新的 sigalarm(interval,handler)系统调用。如果一个应用程序调用了 sigalarm(n,fn)，那么在它消耗了 n 个CPU "ticks"之后（在 n 个时钟 中断后），将执行 handler 函数。当 handler 通过调用 sigreturn()返回时，应用程 序应该从中断的地方继续。Tick 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)，则停止产生周期性调用。 xv6 存储库中有一个文件。将其添加到Makefile 中。只有添加了 sigalarm和 sigreturn 系统调用才能正确编译。alarmtest 在 test0 中调用 sigalarm(2,periodic)以要求内核在 2 个时钟中断后调用一次 periodic()。可以在中看到 alarmtest 的汇编代码。**

一、思路及代码：

Test0：

首先参照lab1的内容流程为sigalarm和sigreturn添加系统调用。 随后在struct proc 结构体中添加记录时间间隔, 调用函数地址, 以及经过时钟数的字段，并在allocproc， freeproc的函数中进行赋0。

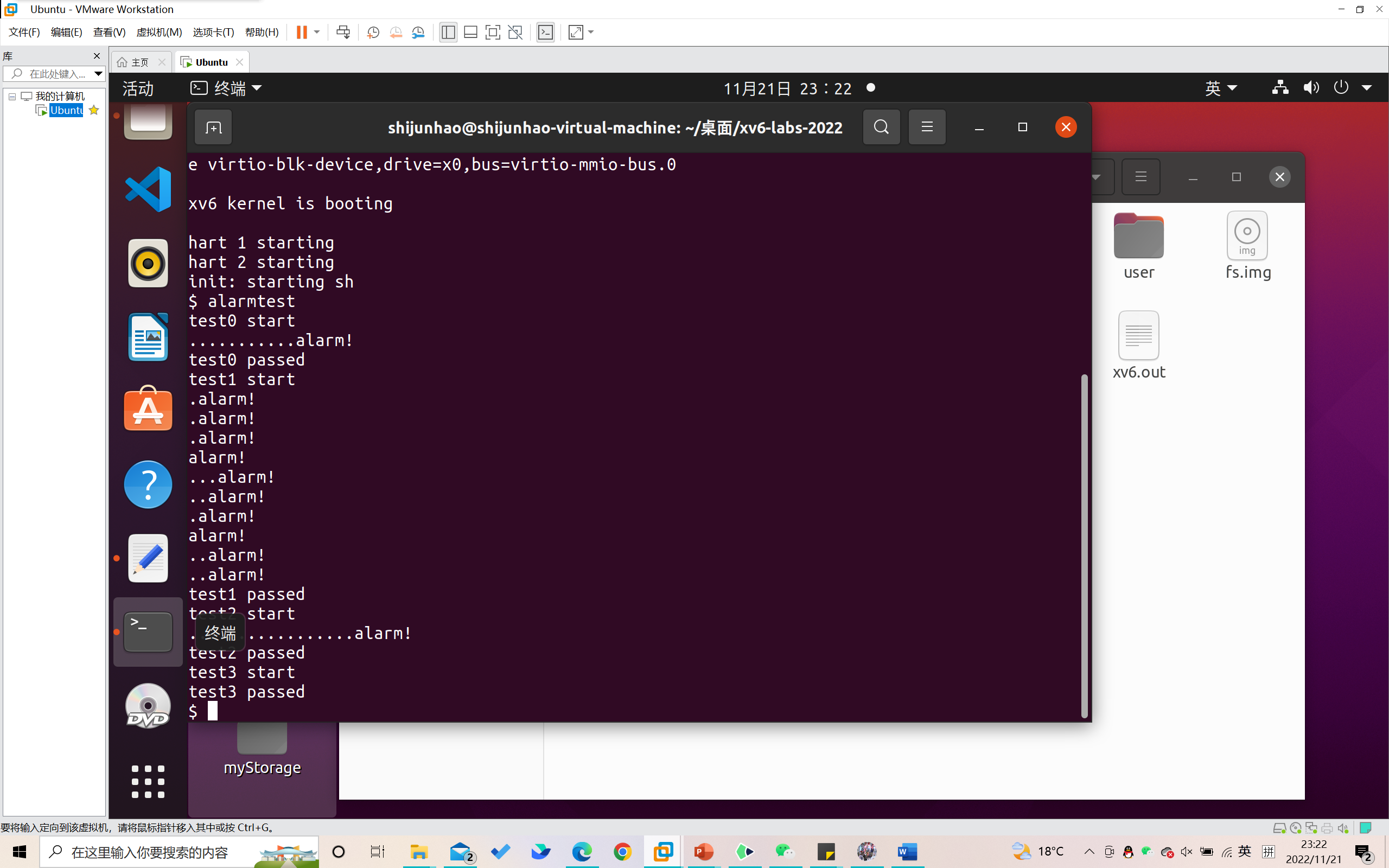
在这之后编写 sys\_sigalarm() 函数, 将 interval 和 handler 的值存到当前进程的 struct proc 结构体的相应字段中。在trap.c中，将触发alarm时的p->trapfram->epc 置为 p->handler, 这样在返回到用户空间时, 程序计数器为 handler 定时函数的地址, 便达到了执行定时函数的目的.

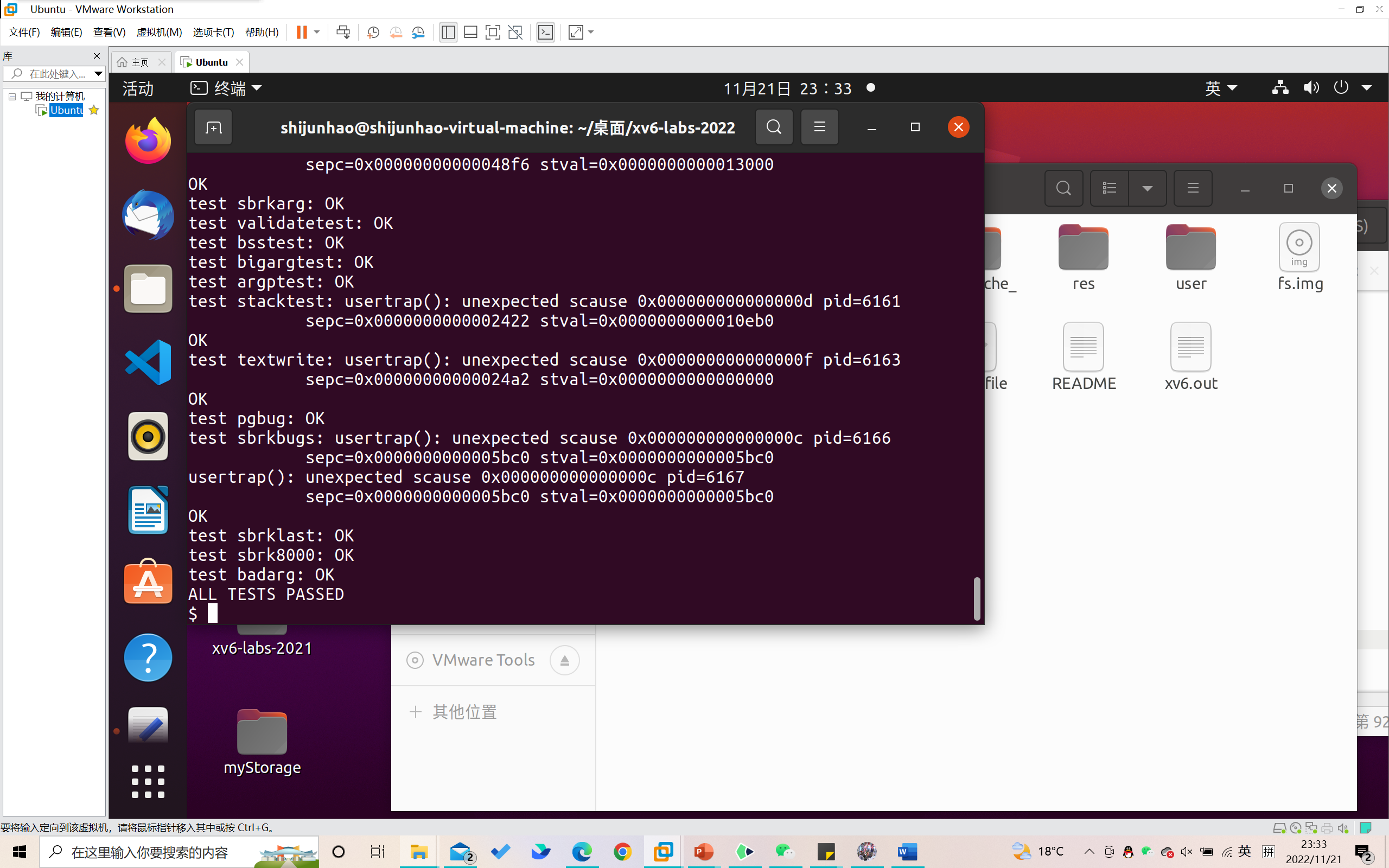
Test1、2、3：

为了能够在完成测试后恢复原来的寄存器，需要在 sigalarm() 函数中将寄存器值进行保存, 同时在 sigreturn() 函数中进行恢复。考虑到handler等参数会在测试中不断进行改变，所以在 struct proc 中保存一个 trapframe 的副本trapcopy, 在覆盖 epc 之前先保存原始副本, 然后在 sys\_sigreturn() 中将副本还原到 p->trapframe 中, 从而在 sigreturn 系统调用结束后恢复用户寄存器状态时能够将执行定时函数前的寄存器状态进行恢复，同时对trapcopy进行初始化赋0，还原后清0。

具体代码详见附件

1. 屏幕截图





1. 困难与解决办法

根据hint一步步实现其实都蛮有难度的，实际上慢慢来，用调试工具一个个试花了不少时间。

**Part 3：The Ending**

通过本实验我了解了xv6系统中对于traps的设置，同时对操作系统中内存的内容有了更深入的理解。在过程中也可以亲身体会到整个实验流程是循序渐进的，前面实验所学到的东西在后面的实验中会有所体现。