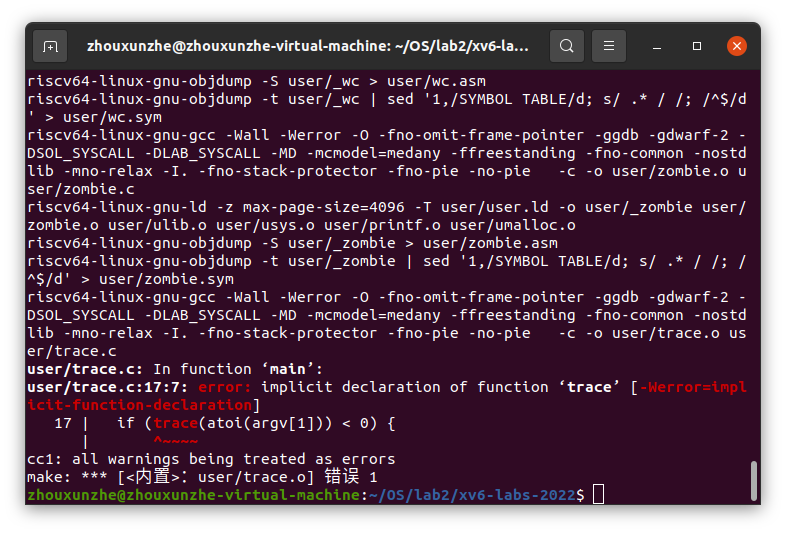
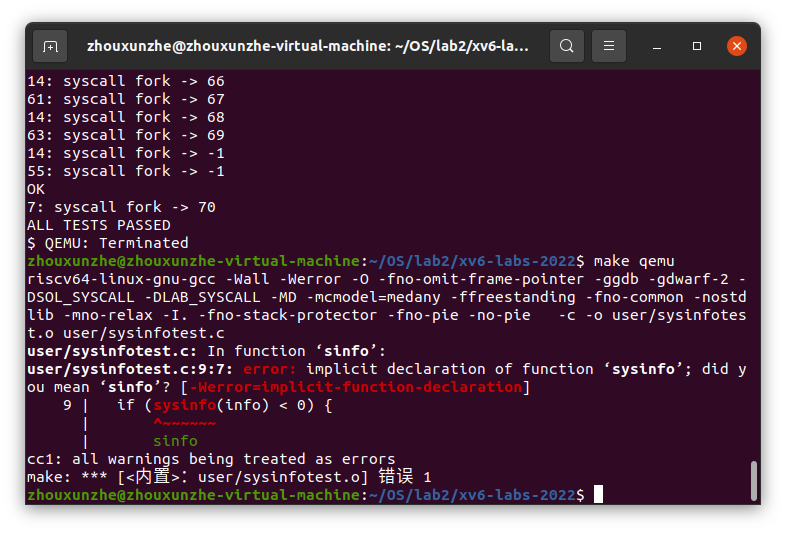
**Lab2实验报告**

一、实验思路、相关代码截图、结果截图

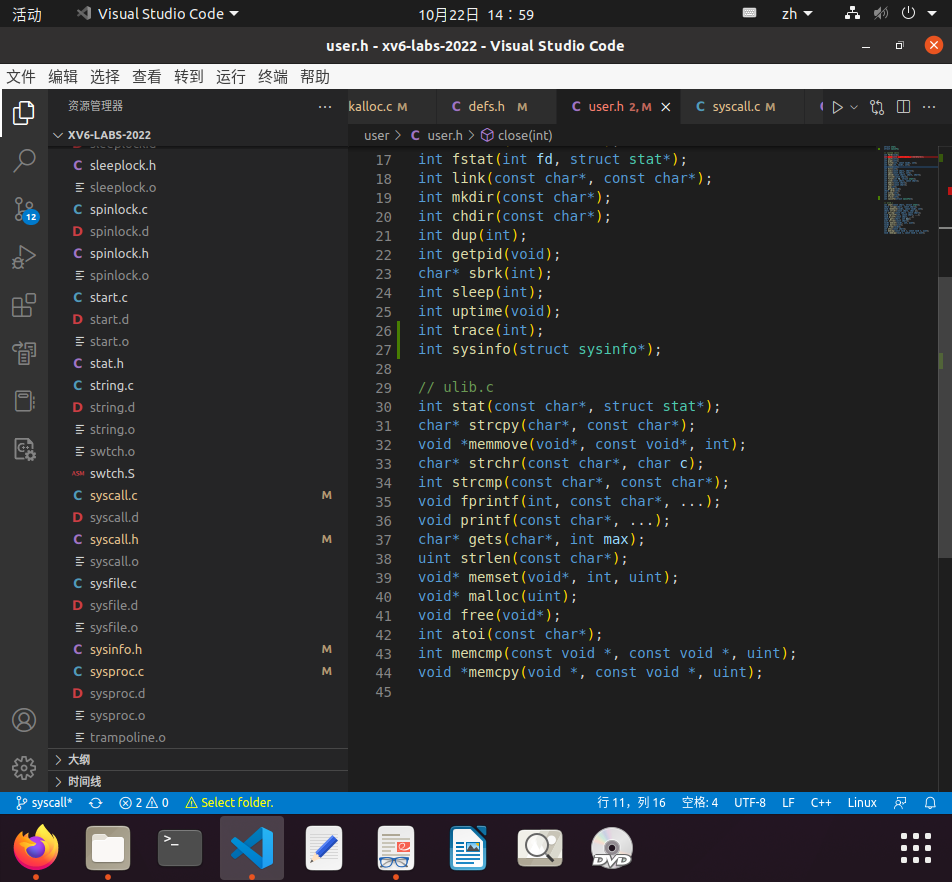
1. 根据提示，在makefile文件中添加$U/\_trace\和$U/\_sysinfotest\到UPROGS目录下。

2. 运行make qemu后确实无法编译。

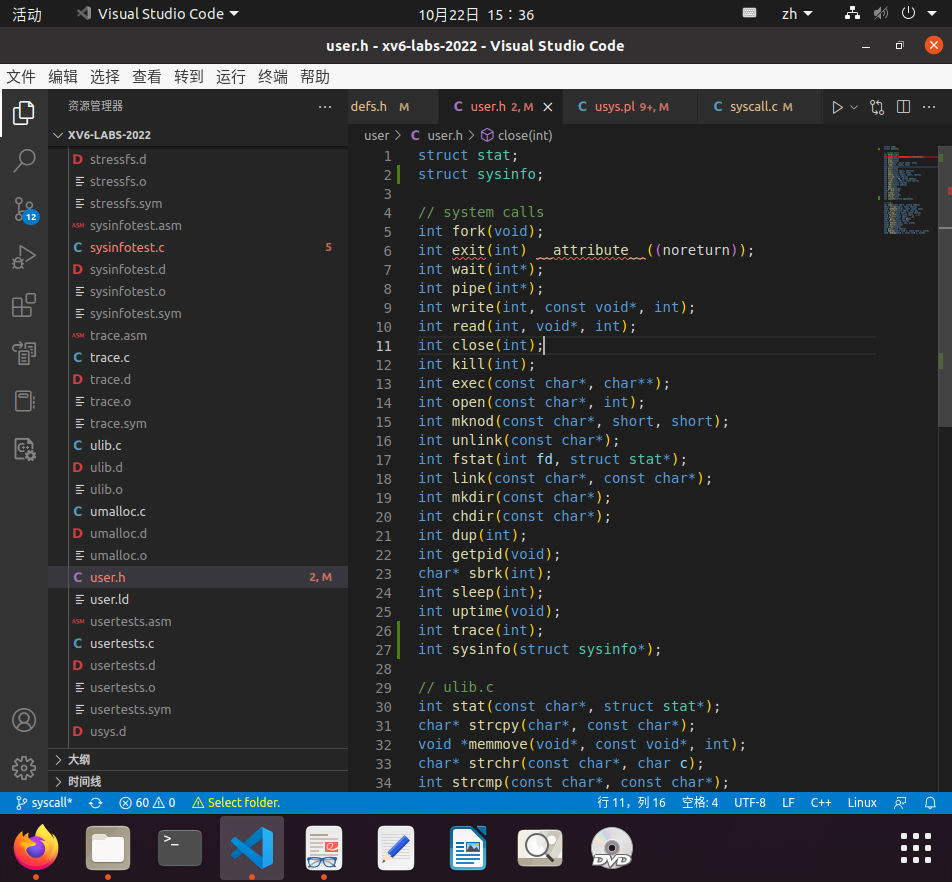




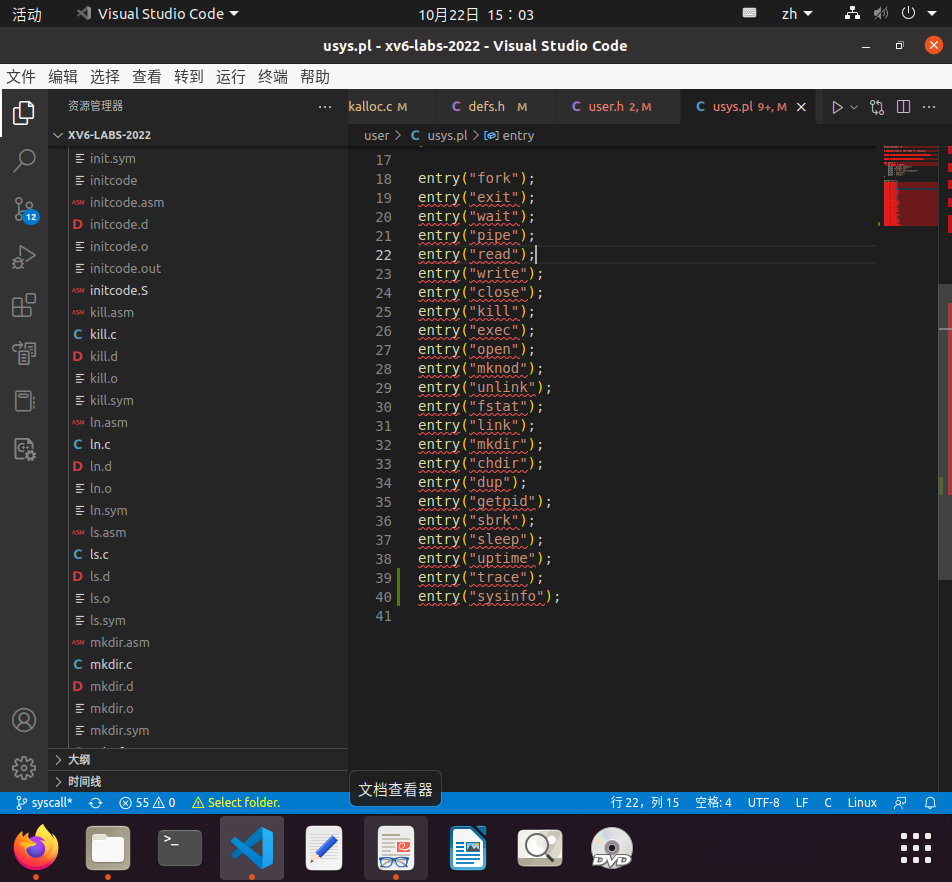
根据提示，在user.h中加入声明



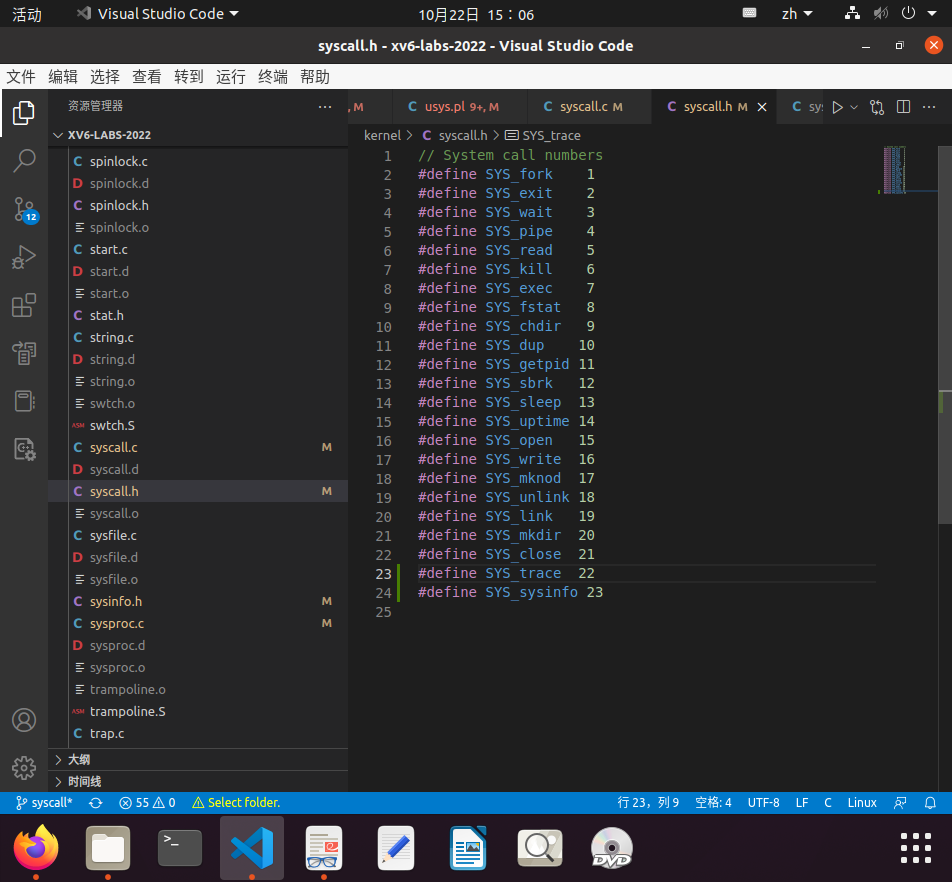
除此以外由于sysinfo（）需要使用结构体struct sysinfo，所以还需要增加声明



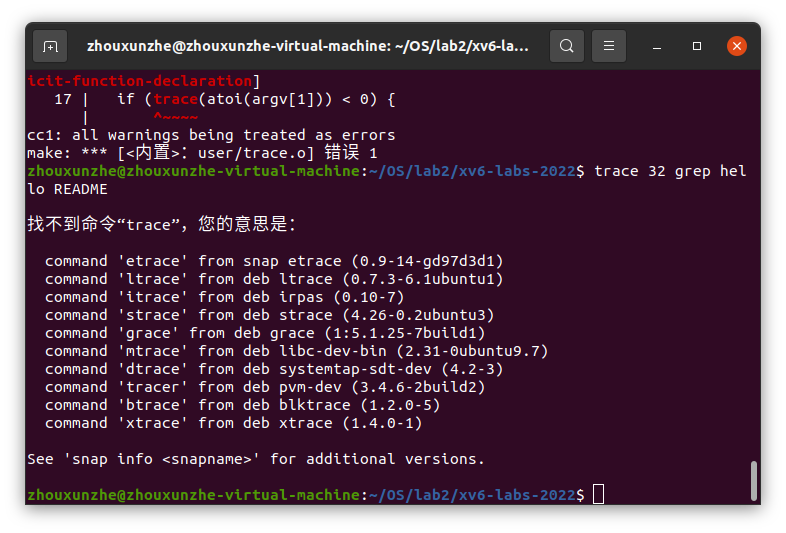
在usys中加入调用，使得用户层和kernel层能够相互协调

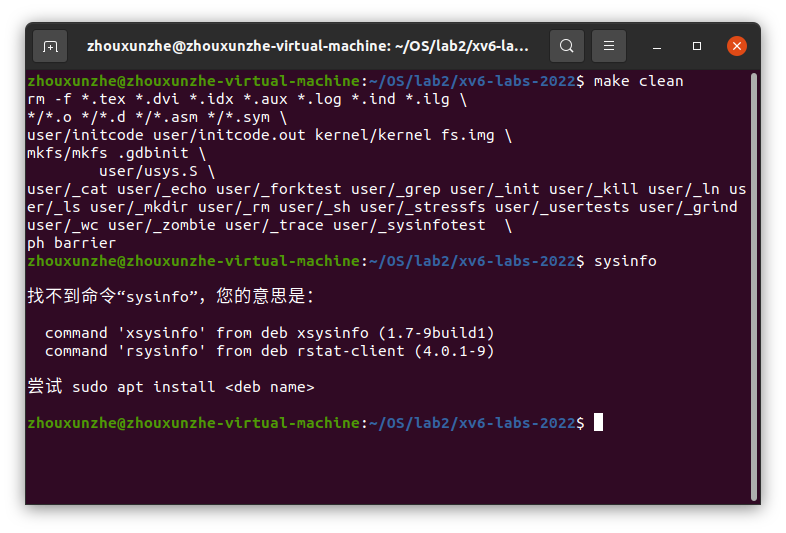


在syscall.h中加入一个syscall number，给系统调用一个编号

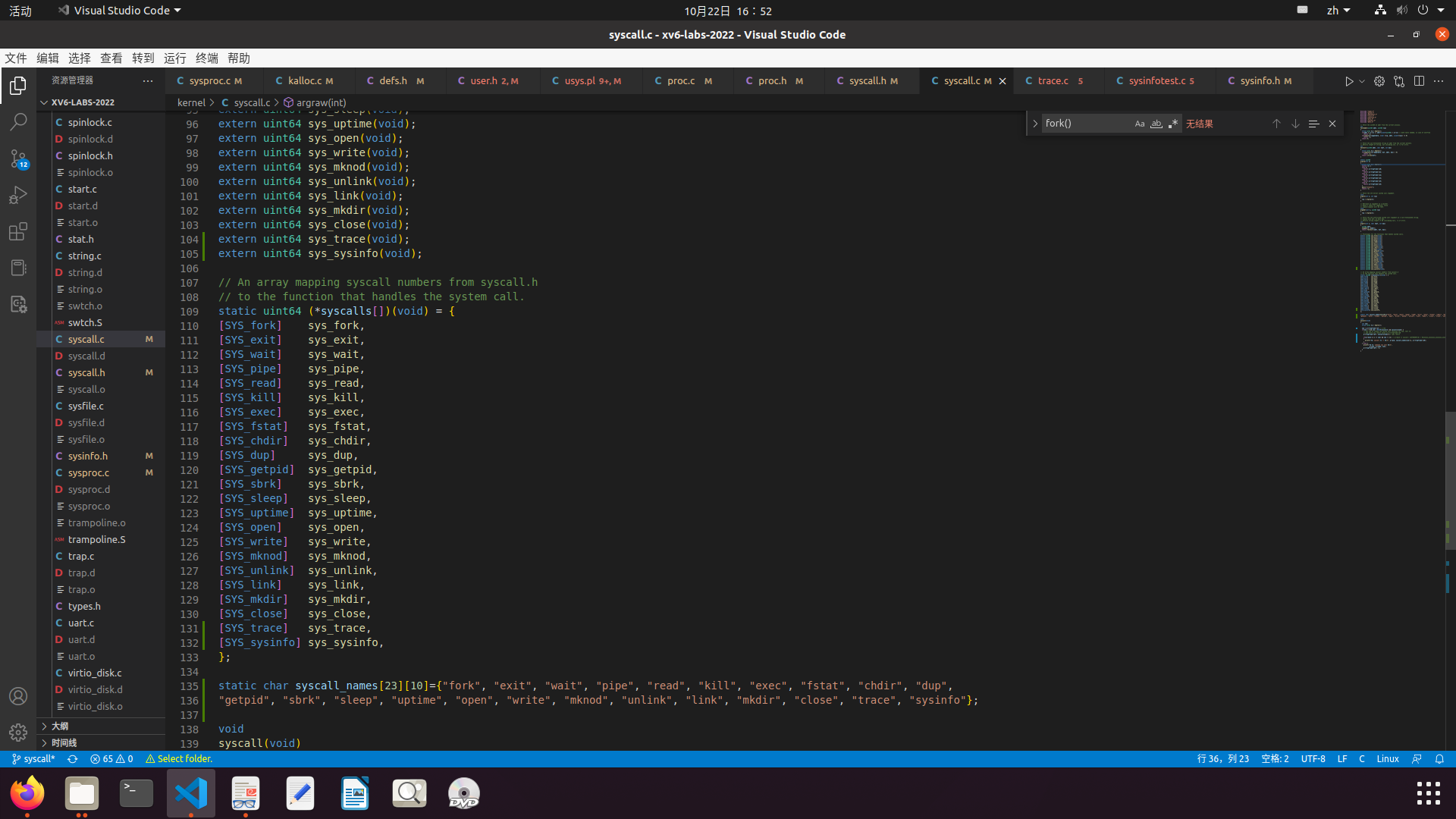


若此时尝试运行trace、sysinfo指令，会发现运行不了，因为没有编写trace、sysinfo的具体操作。



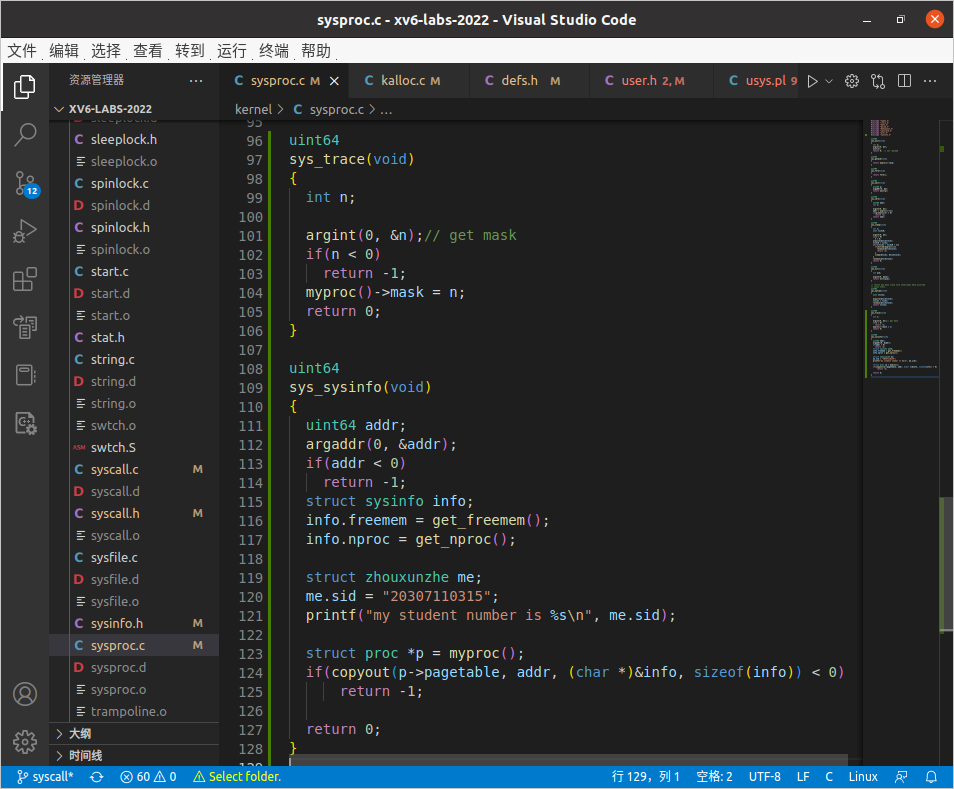


除此以外，由于实现是用户层调用函数，其中函数使用系统调用的指令，所以在syscall.c中还需要声明系统调用的函数。



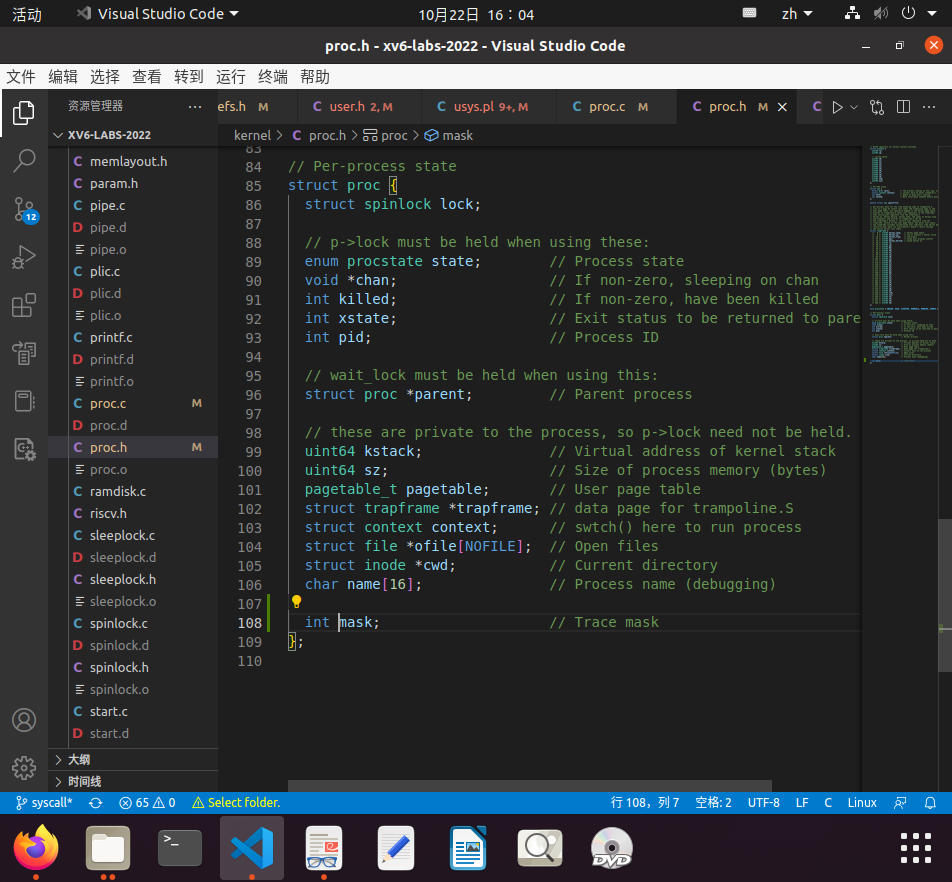
3. 根据提示，实现trace具体函数

（1）根据提示，在sysproc.c中加入sys­\_trace()函数，这个文件的目的就是将用户层的指令中的系统调用转换为kernel层的系统调用。

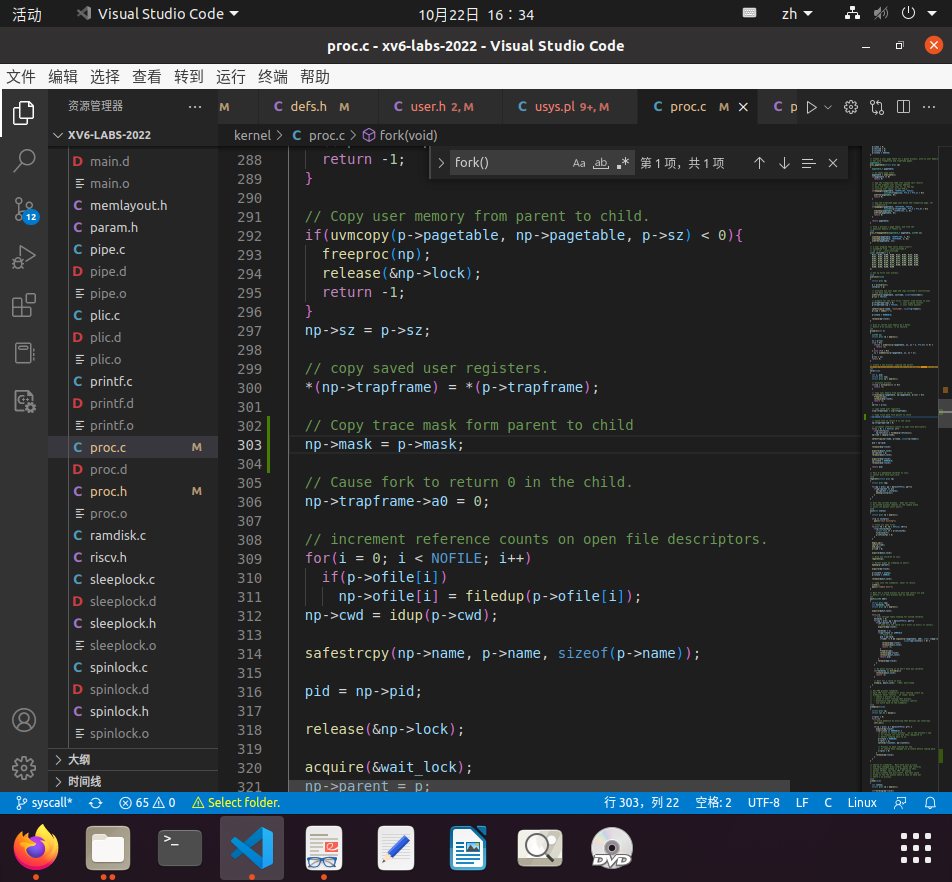


对于sys\_trace()来说，trace的操作就是追踪一个指令中包含的指定的某个系统调用，比如我想要追踪grep hello README指令中的read系统调用，其中，read操作的编号为5（syscall.h），那么就需要设置mask（trace.c），即trace指令中的第二个参数为2^5=32=100000（b），这样就可以只追踪read的系统调用了。

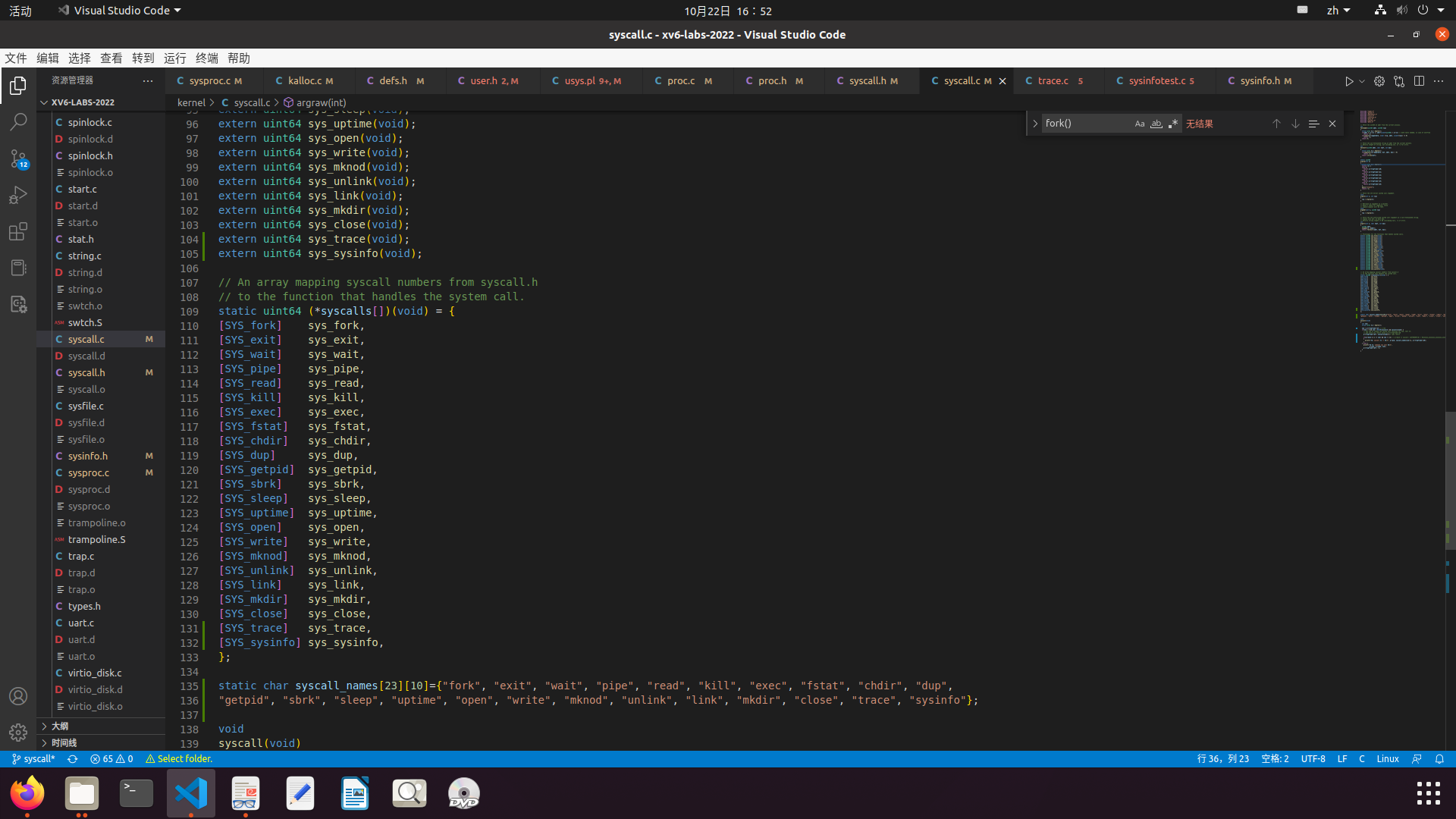
其中，argint（int num，int\* n）函数的操作就是将寄存器a<num>中的值存到n中，由于指数入两个参数，第一个为指令，第二个为实际参数，所以trace输入的参数会存到a0寄存器中，只需要设置num=0即可获取输入的值，由于追踪的指令编号大于零，所以mask如果小于0，则报错返回。然后将输入的mask存到结构体proc中即可。

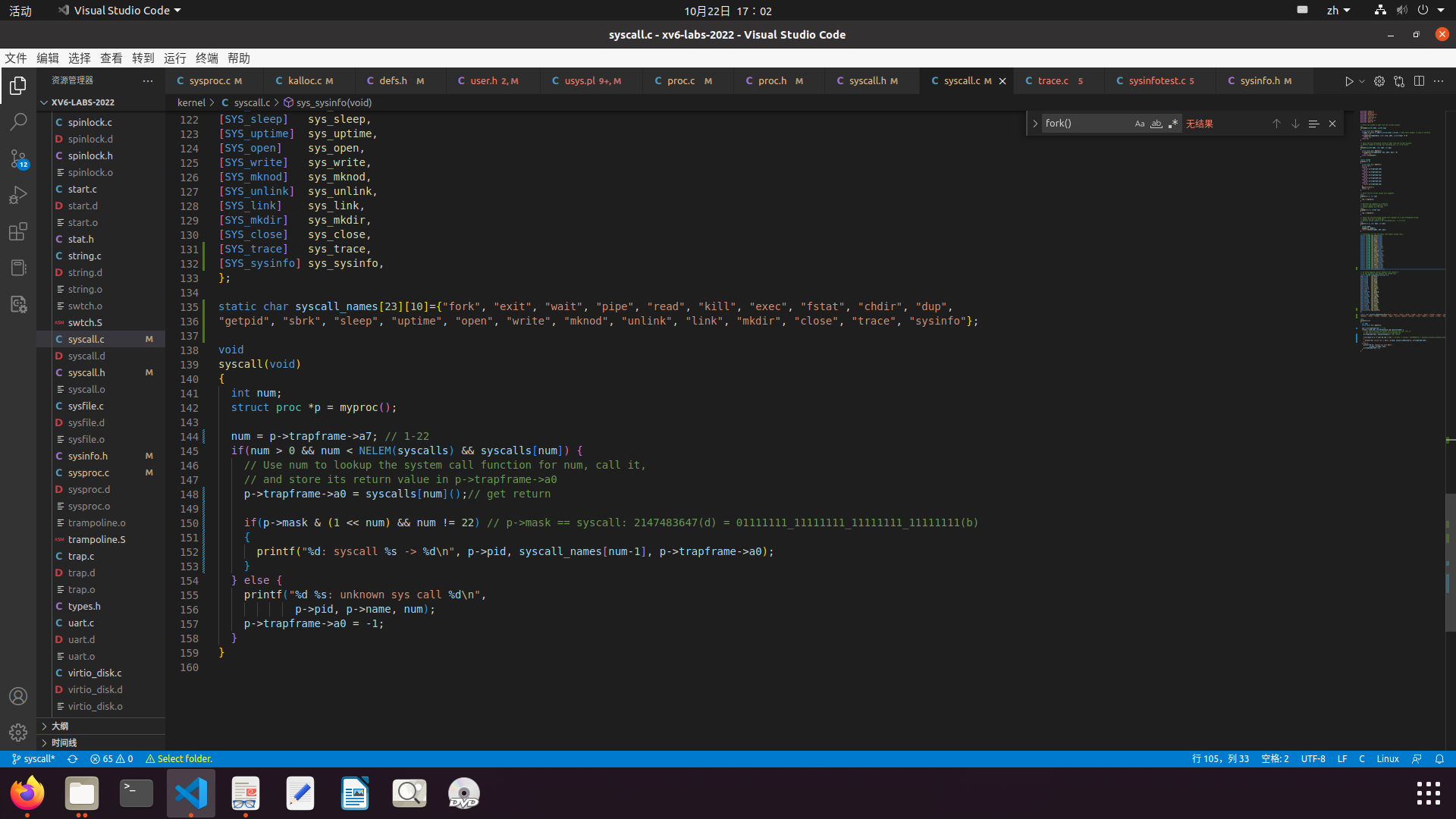


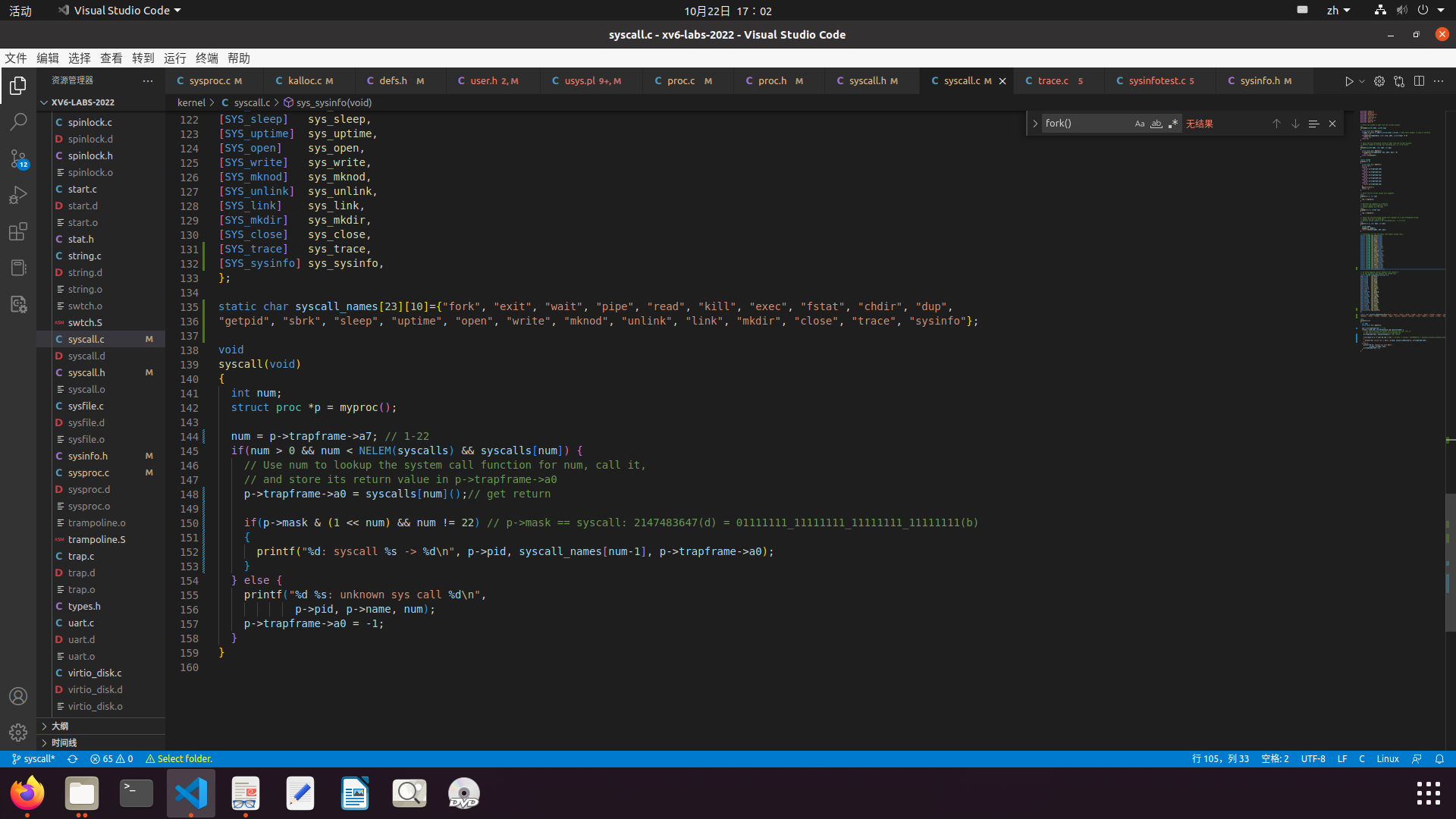
（2）修改proc.c中的fork（）函数，使得子进程能够继承父进程中的mask变量。



（3）修改syscall.c中的syscall（）函数，使得能够输出需要追踪的系统调用的进程号，系统调用名称以及返回值。其中，由于需要输出调用的系统调用的名称，所以还需要增加一个字符串数组用来存储协调调用的名称。除此以外，还有一个匹配协调调用函数的数组，其中有两个map参数，一个是协调调用的编号，另一个是对应协调调用的函数。



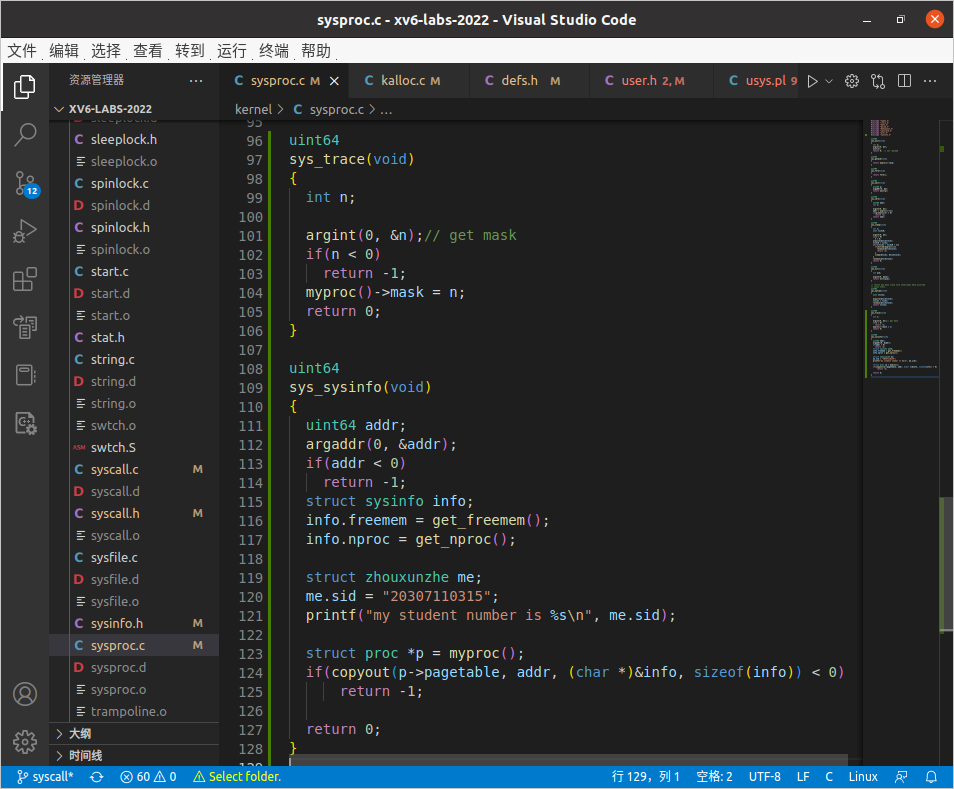




其中，在syscall（）函数中，num参数用来获取系统调用的编号，然后用a0寄存器存储syscall[num]()的返回值，最后只要输出mask中指定二进制数为1的系统调用的值。（用p->mask & (1 << num)）作为if的判别式。

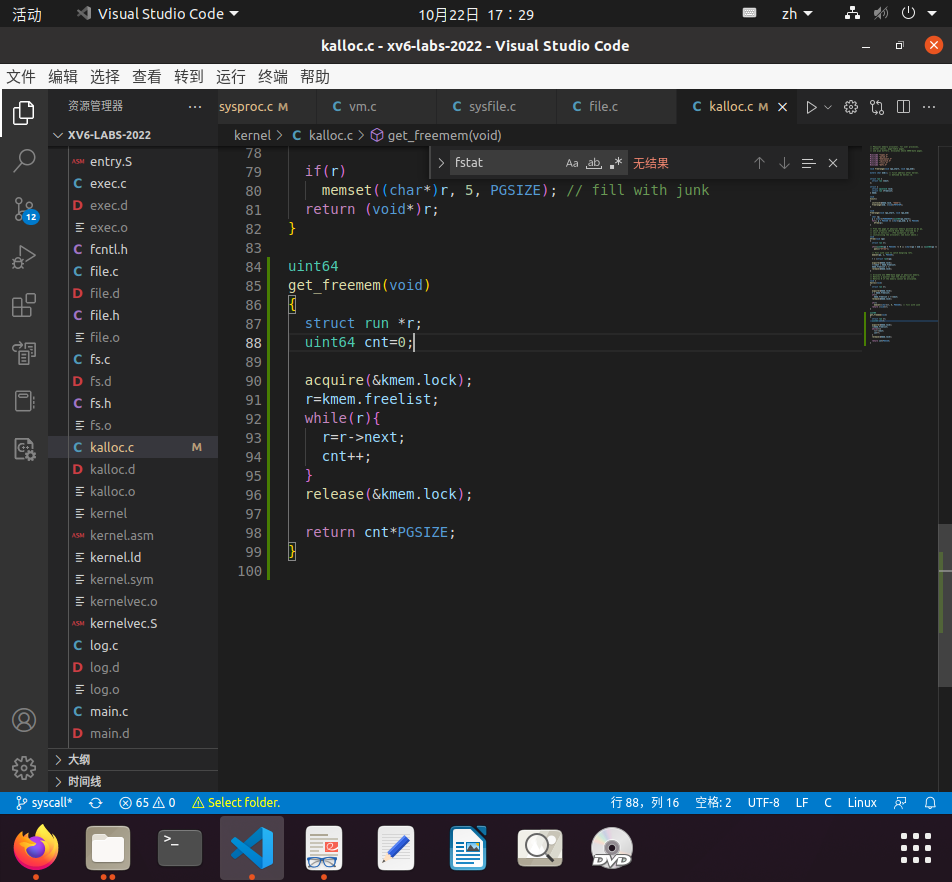
4. 根据提示，实现trace具体函数

（1）对于sys\_sysinfo()来说，sysinfo要实现的功能就是获取系统信息，如空闲寄存器的数量、调用的进程的数量等。



在sysinfo（）函数中，首先应该获取系统信息中空闲的寄存器以及调用的进程的数量。需要使用到两个函数，分别在kalloc.c和pro.c中获取信息。

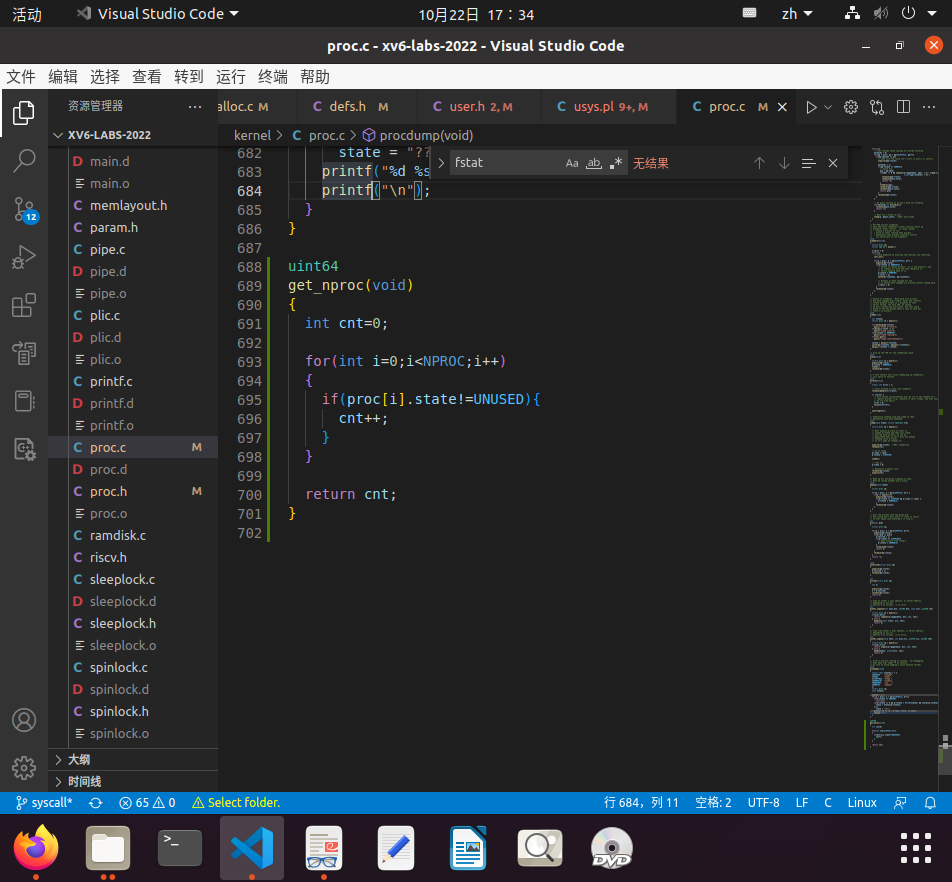
（2）get\_freemem()



参照其他操作，由于可能在统计过程中会发生空闲寄存器数量的变化，所以需要在计数前加锁，然后根据kmem的结构是链表结构顺序获得空闲寄存器的数量。

最后，由于题目需要获取的是空闲寄存器的byte数，所以返回结果应该是kmem的数量乘以PGSIZE。

（3）get\_nproc()

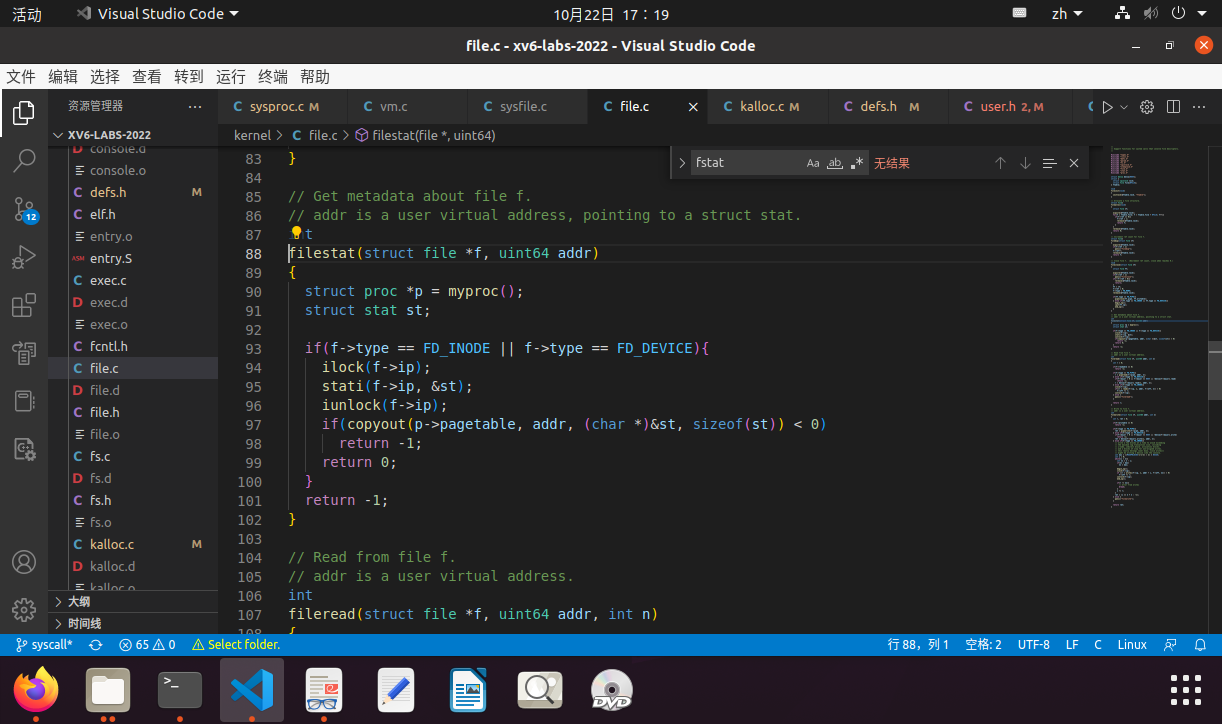


相比于寄存器需要上锁，获取进程数，由于是对进程进行统计，则上锁也没用。

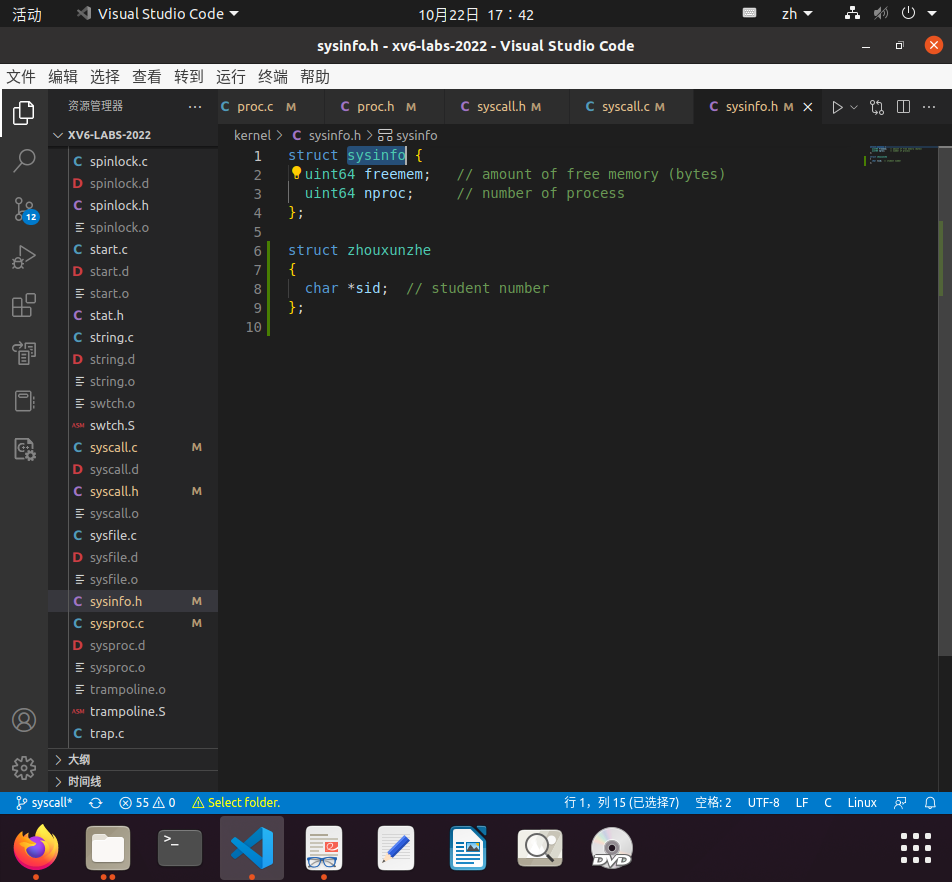
所以只需要简单对所有进程进行遍历，统计其中UNUSED的锁的数量即可。

（4）copyout()

首先需要获取系统调用的首地址，然后才能对地址为addr的sysinfo进行copyout（）操作，复制一个struct sysinfo到用户层。（需要参考file.c中的filestat（）函数）。

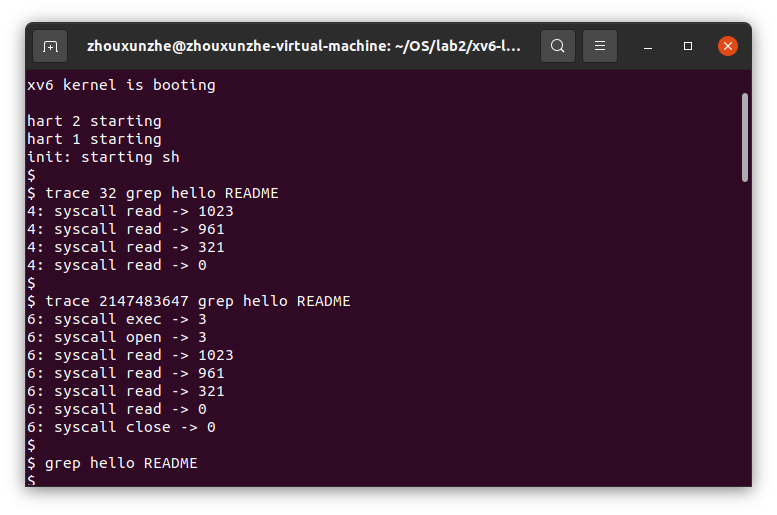


（5）根据题目要求，每次调用sysinfo时，都输出一次学号，以便可视化sysinfo被调用。



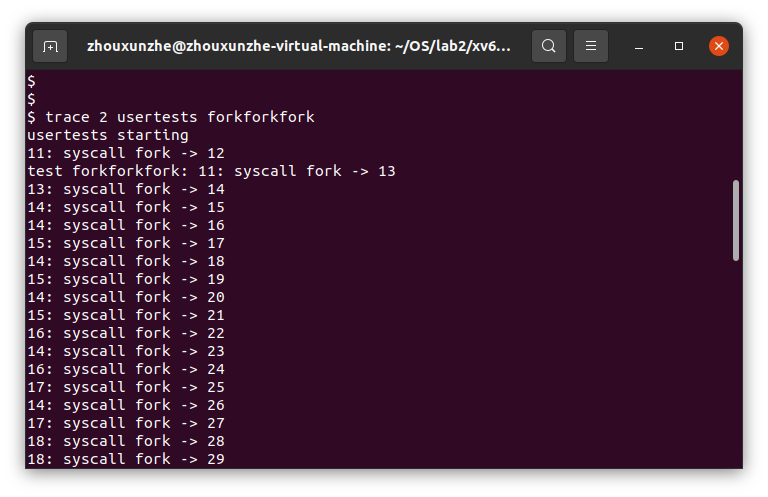
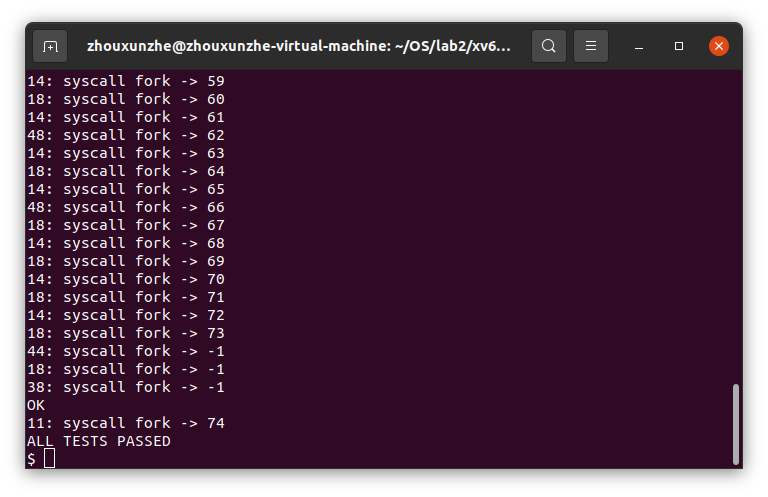
5. 最后，分别运行题目中要求的指令，结果如下：

（1）trace



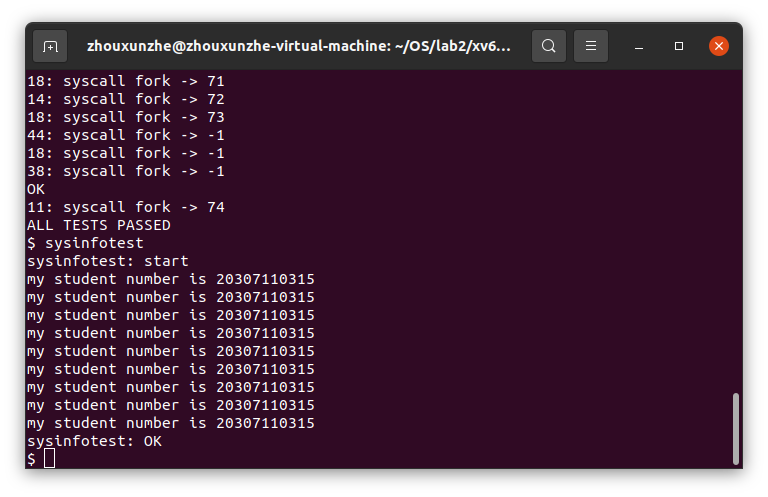
其中32为2^5表示只追踪read的系统调用

2147483647为01111111\_11111111\_11111111\_11111111表示对所有系统调用进行追踪，根据测试预期结果，应该是不用输出trace本身的系统调用，所以输出也符合预期。

对于usertests的test的追踪，其系统调用的名称和预期一样，但是线程号和返回值和参考文档中的结果有出入，但是结果都是全部测试成功，所以我觉得应该是不同电脑导致。所以结果也是基本符合预期。

（2）trace



输出的结果也是符合预期，表示空闲寄存器数量和进程数的统计都与实际相符，但是sys info的调用次数（学号的输出次数）与样例不符，猜测可能是对sys\_sysinfo()的书写有出入，所以导致调用次数的可视化有出入，可能实际调用次数是一样的。

二、问题回答

（1）System calls Part A部分，简述一下trace全流程.

Trace的主要目的就是追踪某个系统调用指令。首先用户层输入trace命令，在makefile中会进行编译进入系统调用中，一个是syscall中会实现用户层与核心层的转换，将用户层的系统调用与核心层的函数结合，一个是usys实现将系统调用转换为汇编代码。

在系统调用过程，syscall（）函数会实现相应系统调用的执行，trace系统调用则是增加一个mask参数用来追踪某个系统调用的相关值并输出。

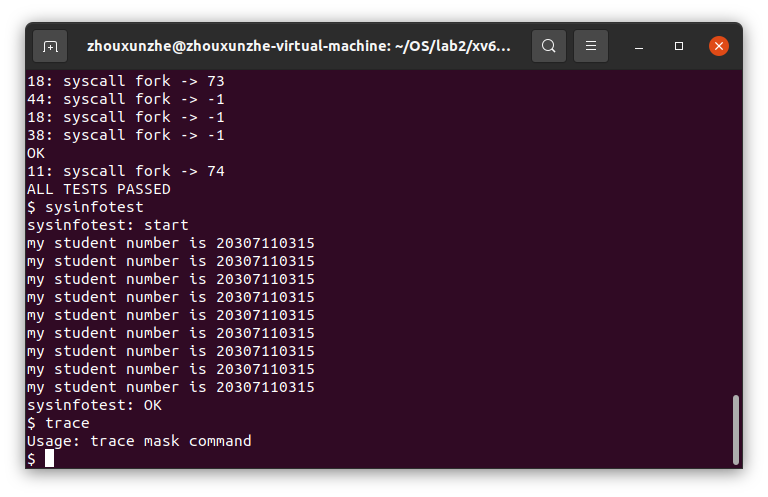
（2）kernel/syscall.h是干什么的，如何起作用的？

是syscall的头文件，其中包含了系统调用指令的对应编号，每一个编号对应一个系统调用，利用syscall.h可以通过编号直接寻找相对应的系统调用函数。

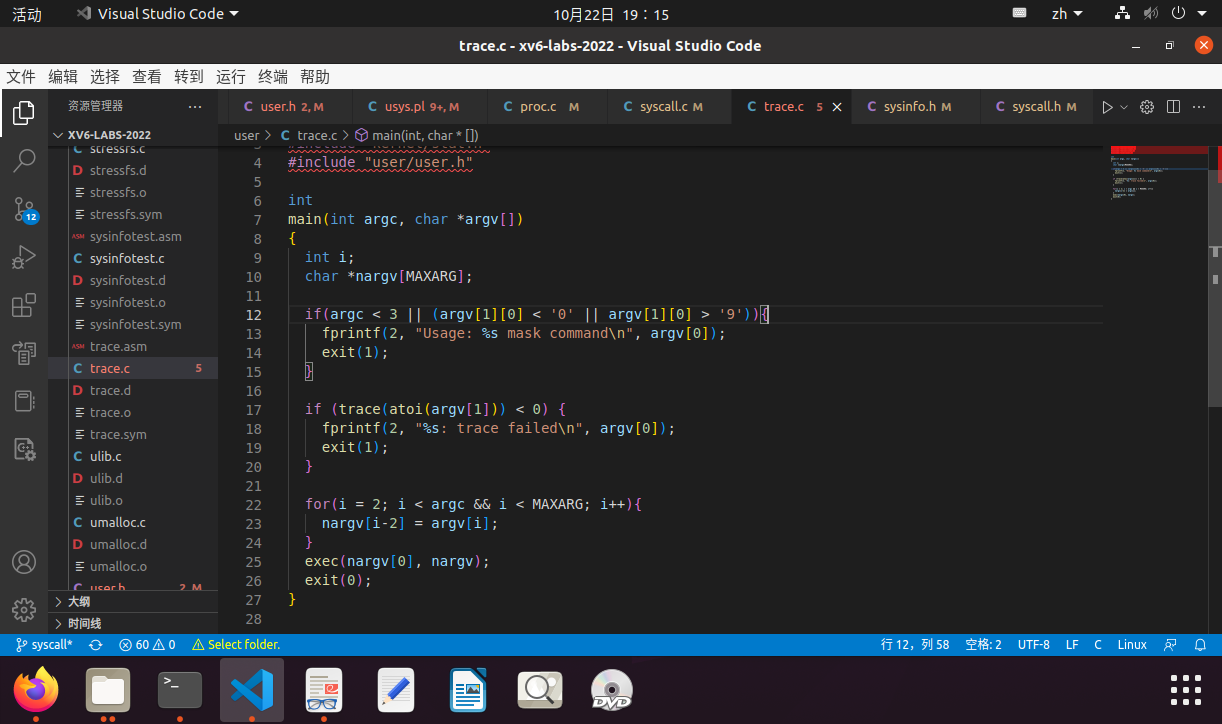
（3）命令 “trace 32 grep hello README”中的trace字段是用户态下的还是实现的系

统调用函数trace？

是用户态下的调用函数。可以测试输入一个错误的格式



结合user/trace.c，可以很明显的看出，其根本上还是调用的用户态下的函数。



只是在用户态下利用exec（）系统调用了sys\_trace（）以及syscall（）函数。

三、实验中碰到的问题。

1. get\_freemem()函数中，一开始没有加锁，导致在不同进程对freelist的数量互相干扰，得到的结果不准确。

后面参照其他函数，进行修改和加锁，输出结果符合预期。

2. get\_freemem()函数中，由于没有看到输出结果为byte的数量，输出只是kmem的数量，所以最后sysinfotest的时候输出结果出错。

仔细研究文档后修改测试结果正确。

四、实验感想

在本次实验中，我学会了如何实现系统调用，并且对系统调用的原理更加清楚了。其中usys.pl文件的功能相较其他.c或.h文件的功能十分新颖，我学会了如何书写汇编语言的方法。除此而外，我还了解了怎么追踪不同的系统调用函数的方法，以及他们自己本身是如何从用户层到核心层实现的。还有通过sysinfo实验，我了解了proc.c和kalloc.c文件的存储方式，以及怎么通过系统调用获取其中信息的方法。

本次实验主要就是对系统调用的深入了解，从中我学会了操作系统中用户层和内核层相关的很多知识，收获颇丰。