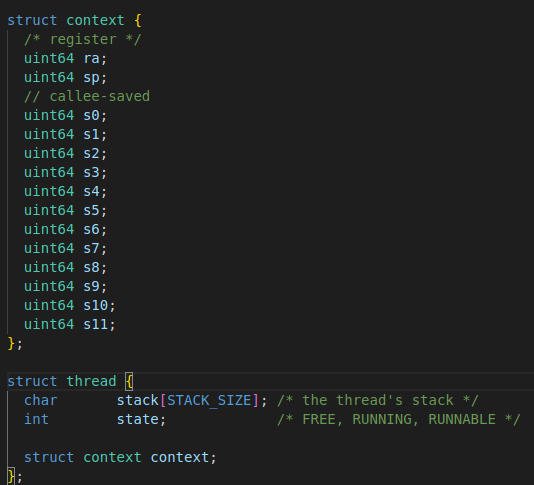
**Lab5实验报告**

一、实验思路、相关代码截图、结果截图

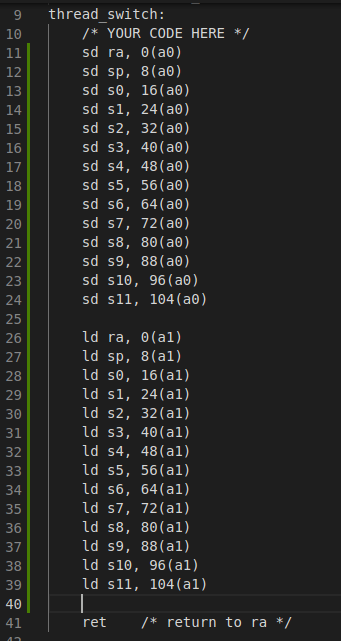
PartA: Uthread: switching between threads

这个实验的主要目的是手写实现线程之间的转换。

首先根据提示，在struct thread中补充一个存储寄存器的结构（由于寄存器的数量很多，可以使用一个结构体用以存储）。



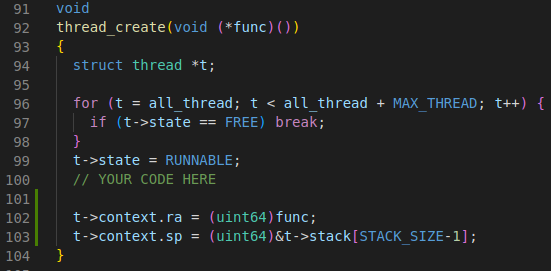
然后仿照kernel/switch.S ，完成notxv6/uthread\_switch.S



根据提示对相关寄存器进行初始化：

ra 寄存器指向线程要运行的函数，switch 结束后会返回到 ra 处开始运行；

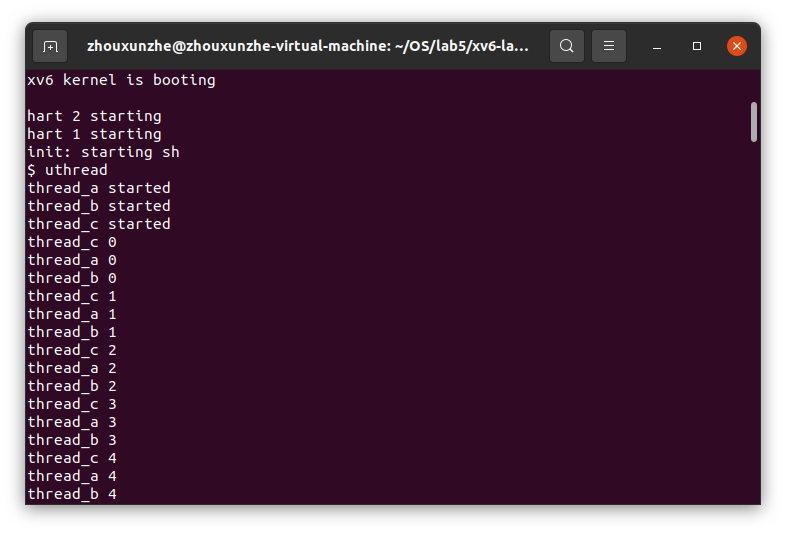
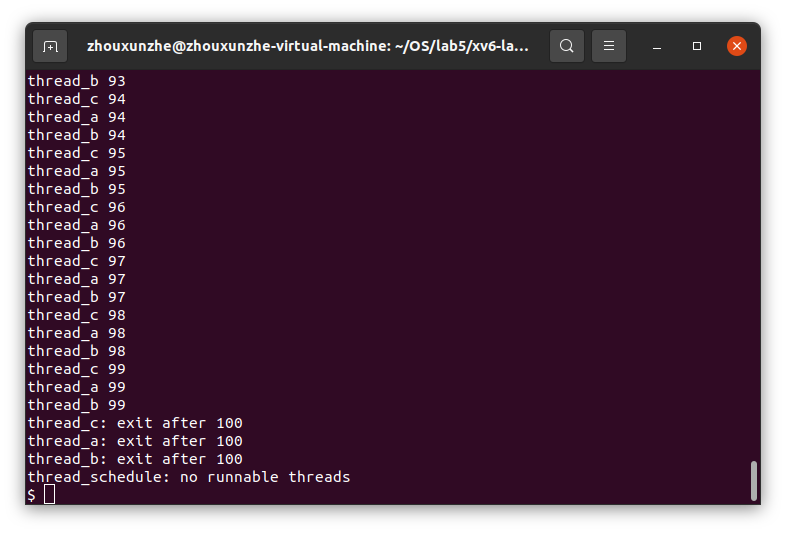
sp 指向线程自己的栈。要注意：压栈是减小栈指针，所以一开始在最高处。



最后一步，在 thread\_schedule() 里调用刚才汇编写的 thread\_switch：

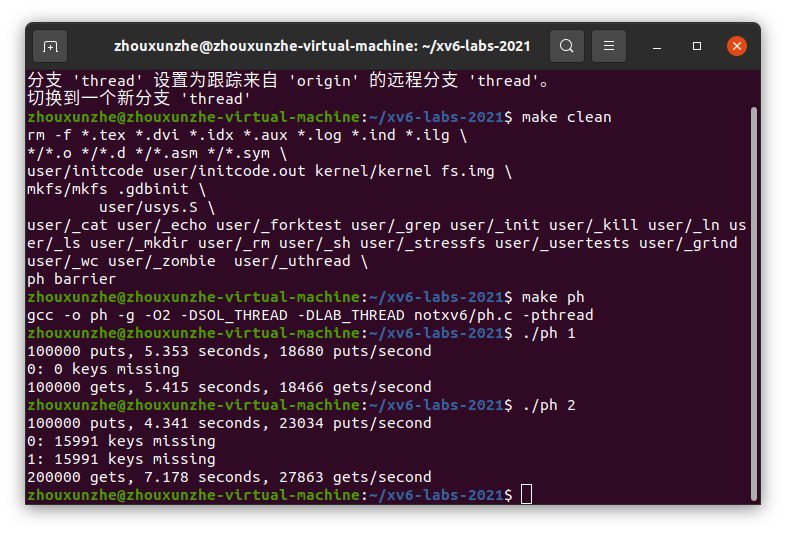


在qemu中尝试调用uthread：

可以看到，程序成功运行，实现创建线程并保存/恢复寄存器以在线程之间切换的任务。

PartB：Using Threads



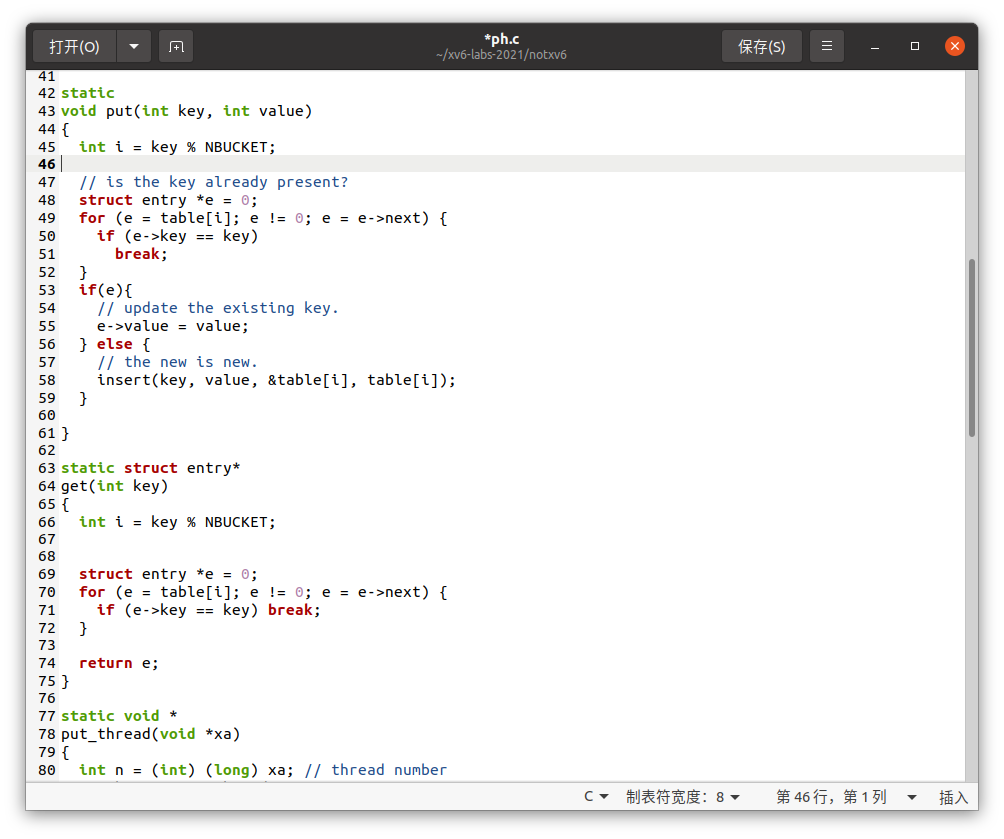
如图，./ph 1表示只有一个线程运行。./ph 2表示有两个线程同时运行。

Put表示将keys放入hash表中，个体表示从hash表中取keys。

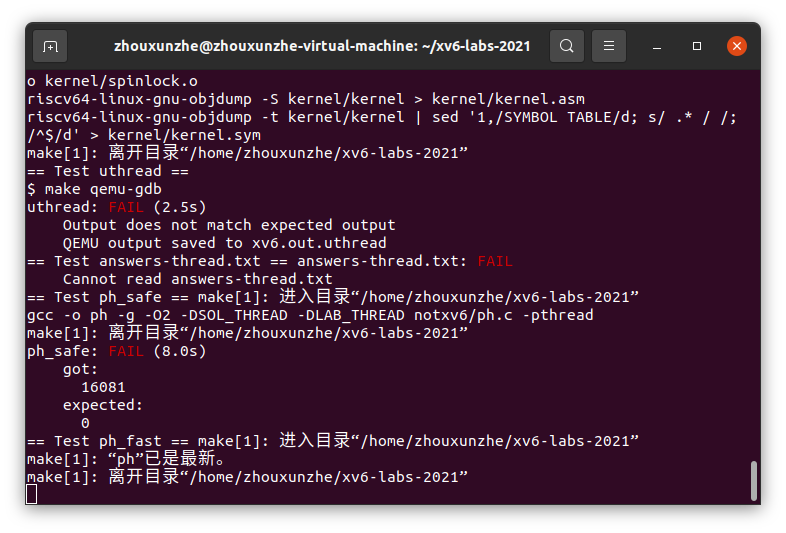
记录keys被put以后但是没有被get的数量。

同时记录put和get的时间，用put和get的数量除以时间就可以得到put和get的效率了。

（put可以看作写操作，get可以看作读操作）



由图中代码可以看出，在未加锁前，对于1个线程来说，不存在missing的情况，因为不存在同时put的情况。而对于2个或多个线程，由于可能同时put的情况，于是可能存在同时修改hash表的情况出现，当两个线程同时对一个key处理，如果value不相同，则可能出现赋值的先后顺序不符合预期，或者key对应空与否的判断在并发情况下不符合预期等等异常情况，都有可能使get出现missing。



如图为实验测试结果。（ph\_safe未通过）

针对以上原理，该实验的主要目的就是对put过程加锁，以保证put过程的完整性，从而使得hash表中内容符合预期。

对加锁操作进行研究，发现要对一个操作加锁，至少进行四个操作：定义一个锁，锁的初始化，获取锁，解锁。

pthread\_mutex\_t lock; // declare a lock

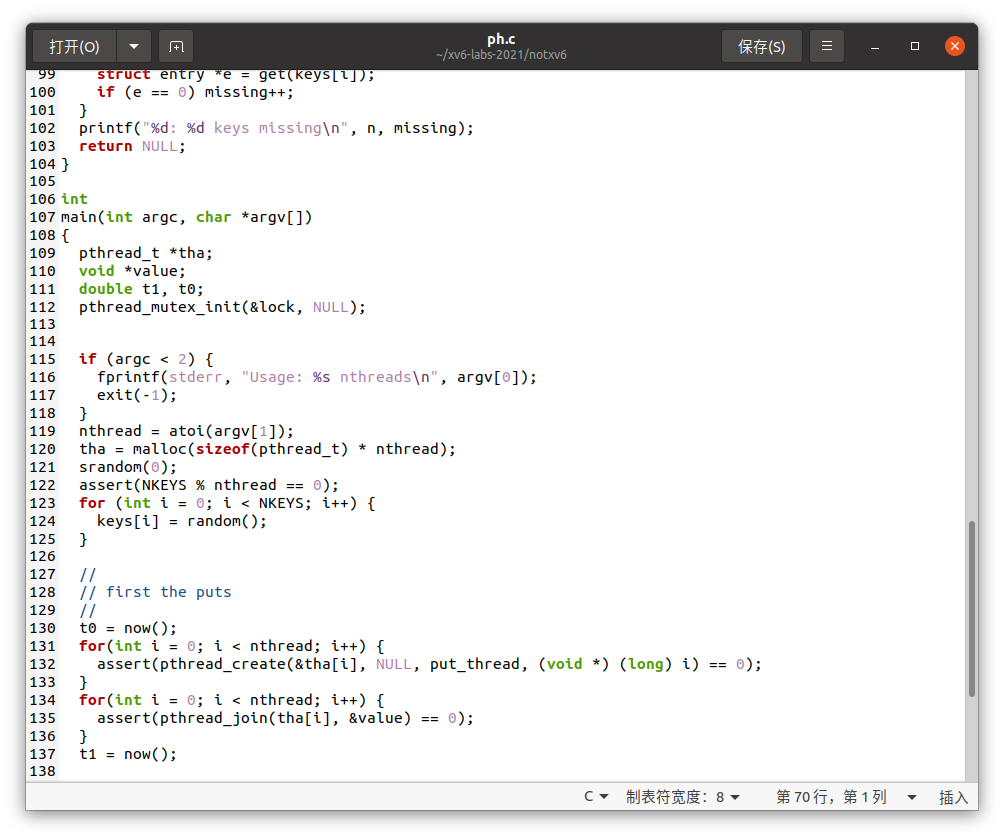
pthread\_mutex\_init(&lock, NULL); // initialize the lock

pthread\_mutex\_lock(&lock); // acquire lock

pthread\_mutex\_unlock(&lock); // release lock



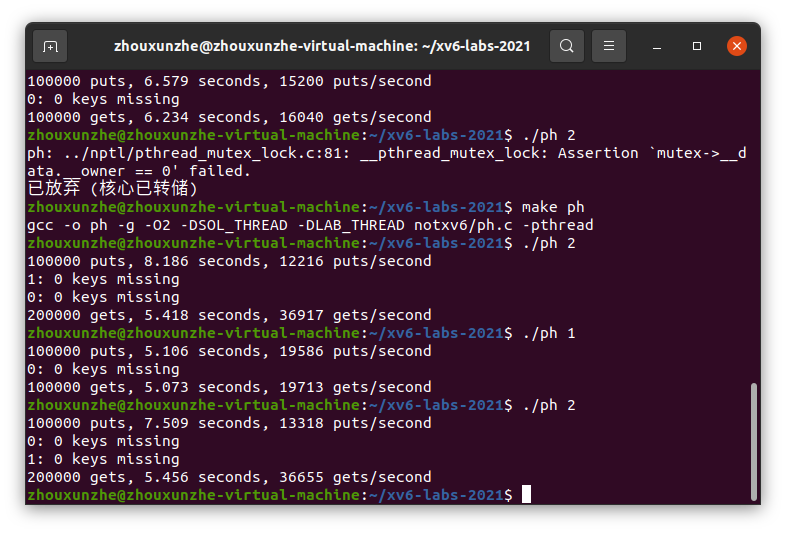
首先定义一个锁为全局变量，这样所有的函数都可以调用。



然后要调用锁，首先在主函数中将锁初始化。

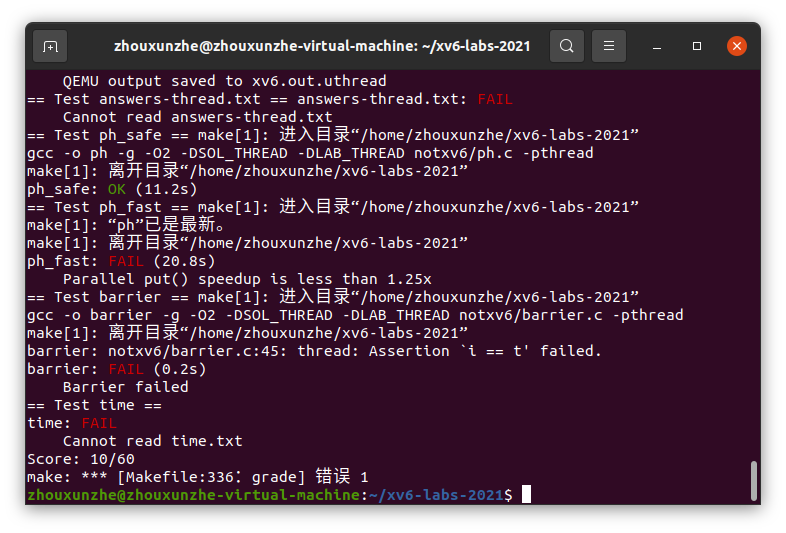


最后如图，在put函数的开始和结束分别加入语句pthread\_mutex\_lock(&lock);和pthread\_mutex\_unlock(&lock);实现获取锁和解锁的操作。



如图为加入锁之后的实现图。可以看到相比加锁前，对于单线程而言，其运行速度也变快了。对于双线程而言，put过程要慢，是因为加了锁操作以后，若获取锁成功，则其他进程无法运行，自旋等待，会消耗一定时间，所以效率降低。而双线程的get比put效率高，是因为get操作不需要等待锁。同时最为关键的是，双线程的missing率为0，说明锁的作用就是防止多个线程同时操作导致写的过程出现并发错误，从而避免missing。

相较而言，由于双线程中是同时get，所以其效率会远超单线程的get效率。



如图为实验测试结果。（ph\_safe通过, ph\_fast未通过）

对于ph\_fast而言，由于需要同时更快的put，因此需要将对整个put过程的锁，转换为对某一个hash bucket的锁（即将hash表看作对多个keys的写入，而非一整个的写入过程），这样可以保证同时可以进行put并且不影响hash表的写入。

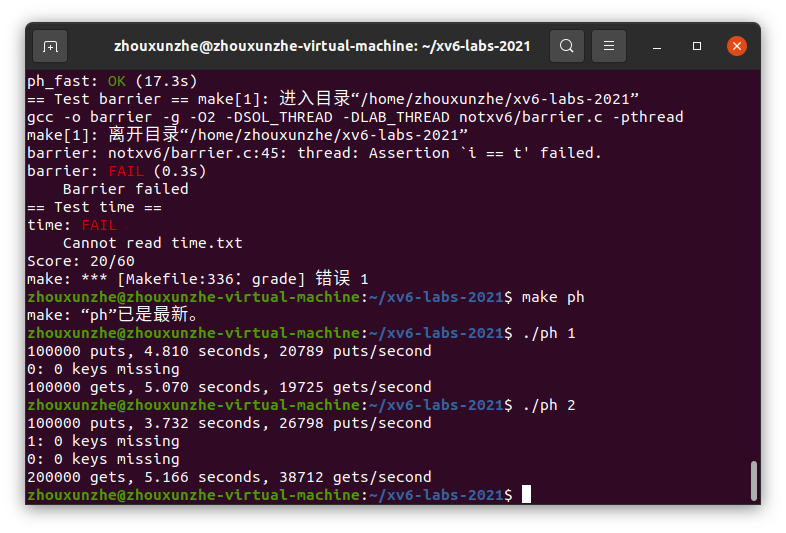
具体实现的思路即是，将put的一个锁转换为hash\_table.size()大小的锁，即lock[NBUCKET]，每次对其中一个lock[i]获取锁以及解锁即可。



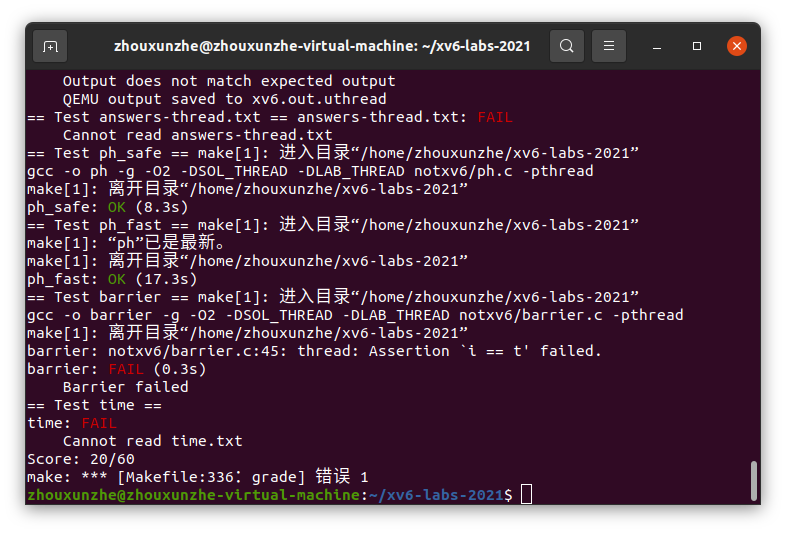
如图，只需要修改lock为一个大小为NBUCKET的容器即可，并且每次获取锁和解锁对象为需要写入的bucket。



初始化过程即将NBUCKET个锁一一初始化即可。



运行结果如图，可见相比普通锁，其双线程的put过程的效率提高了近一倍，说明对bucket加锁可以有效避免自旋等待造成的效率低下。并且可以看出，双线程的写过程，由于可以同时执行多次put，其效率是单线程的put约1.289倍。



如图为实验测试结果。（ph\_safe通过, ph\_fast通过）

最后，对三种运行思路进行效率测试。

为了避免系统运行的随机性，现对三种加锁方法进行重复10次实验，获取平均值进行比较。

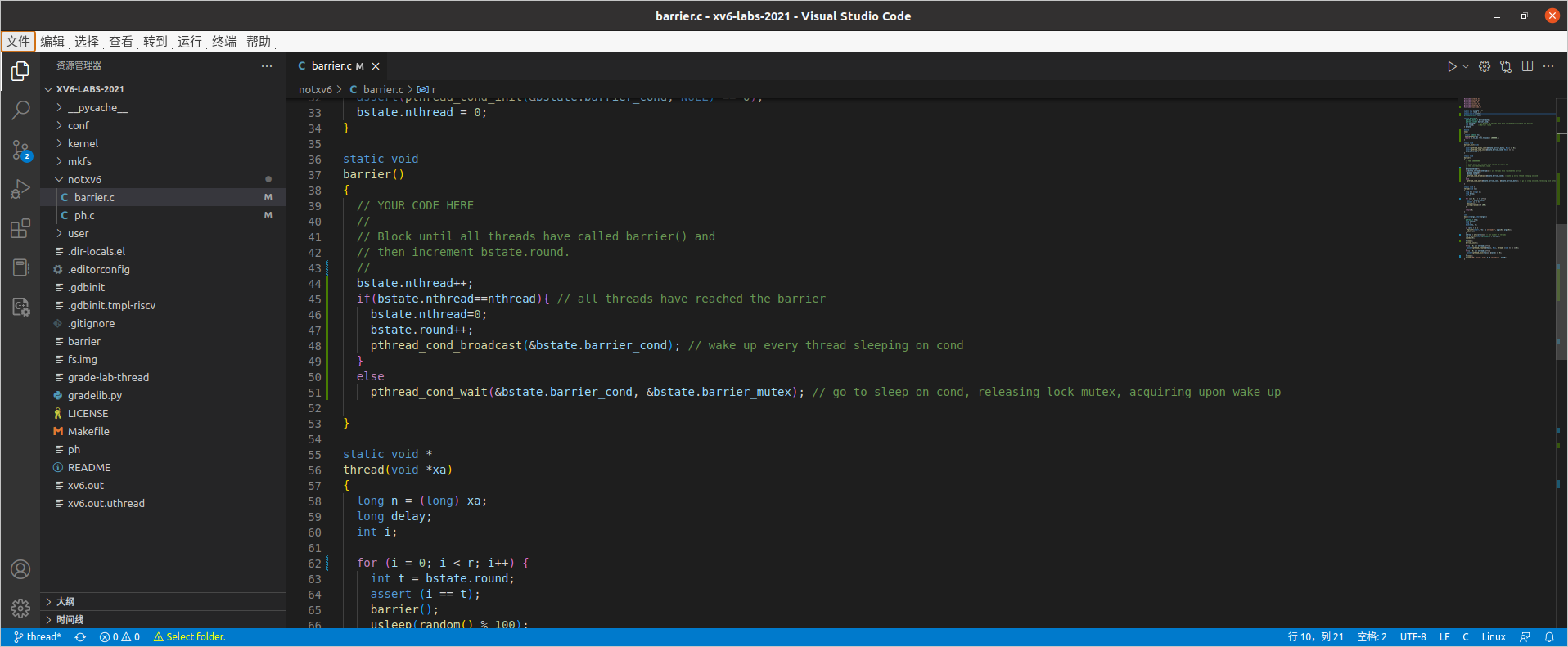
Avg\_missing (ph:t\_num=2) = 16435.8

有统计图可以看出：对于按bucket加锁的算法，put在双线程的效率约为单线程的1.54倍，切其效率约为按put过程加锁算法中双线程效率的2.00倍。

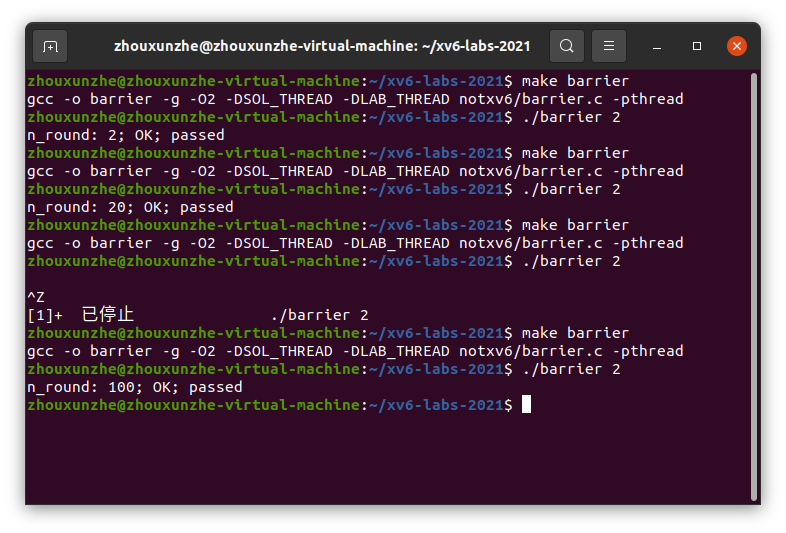
PartC：Barrier

首先执行make barrier和./barrier 2指令，由于barrier函数不完整，所以断言assert (i == t);会报错。仔细研究，发现这个断言目的是为了判断此时的round是否正确。

Barrier函数的目的就是记录多个线程的多个循环。在命令行的参数的含义是需要运行的线程数量。每当一个线程调用barrier（）函数，如果在本次循环有线程还未调用，则会进入wait状态等待所有线程调用，当所有线程都调用barrier以后，则会执行broadcast操作唤醒所有的线程。



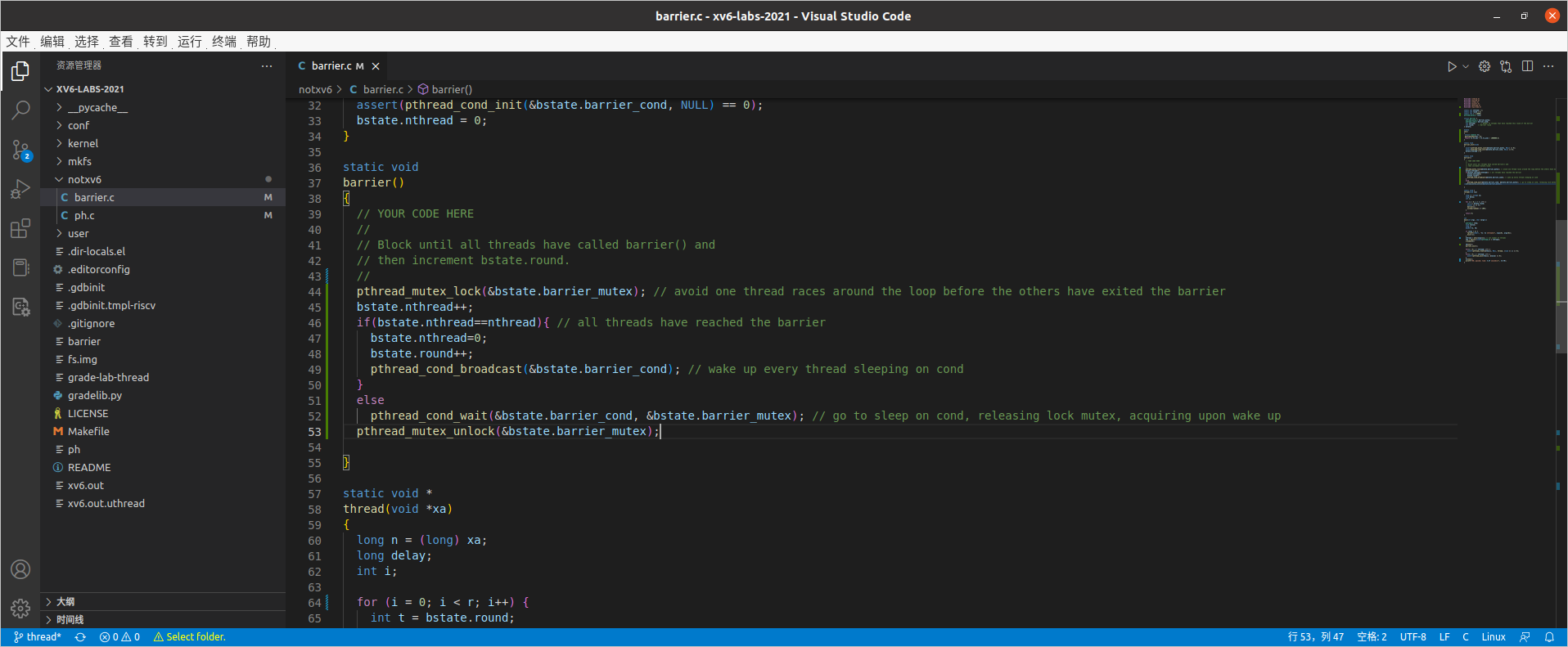
具体实现后运行，发现运行结果会出现死锁的情况。当测试更改循环次数后，如果将次数降为2，20，100次后能正常运行并输出结果。但是当循环次数为200次及以上（只测试2\*10^n次和100次），则会出现死循环，没有运行结果，会一直卡在进程中。



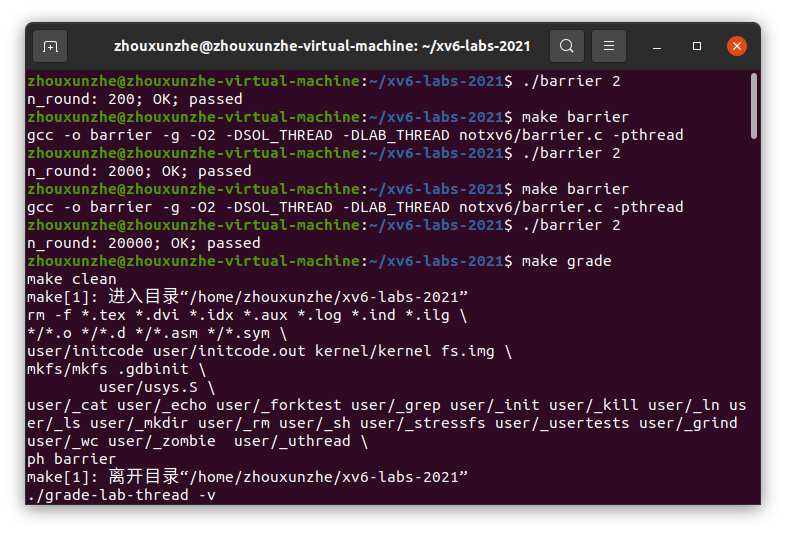
如图所示，当使用两个进程时，测试200次时会出现死循环，没有运行结果。

仔细研究，发现是可能循环次数增加以后，可能出现一个线程对nthread自增的情况，另一个线程设置nthread=0，可能导致本应全部进程唤醒，但是最后一个进程却执行了等待操作，最后没有进程可以执行的情况，导致程序不能继续执行，所有进程都在等待被唤醒，从而没有输出结果。

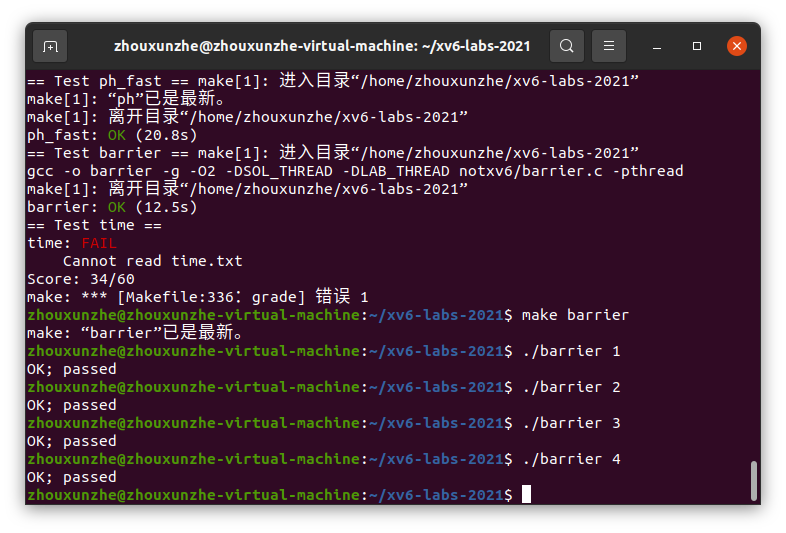
解决办法就是加入锁操作使得barrier执行的过程中不会出现其他进程干扰的情况。



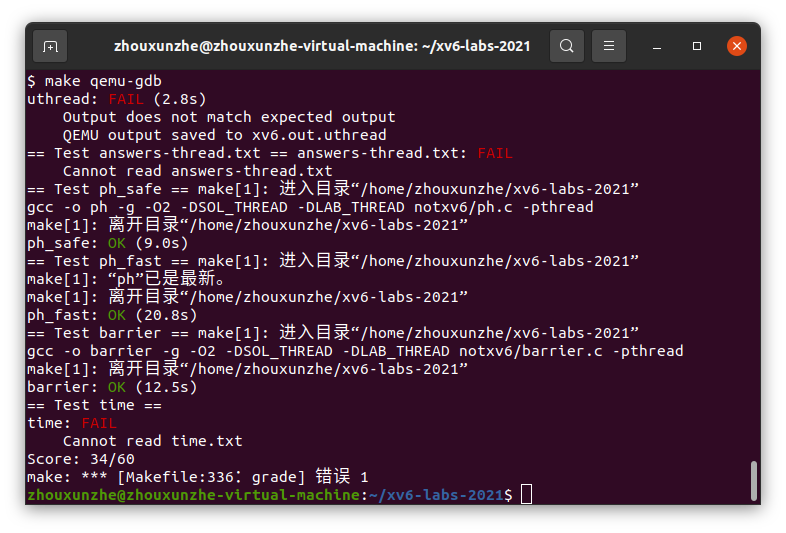
加入了锁操作以后，barrier可以正常执行。



根据提示，测试1、2以及多个线程的情况。

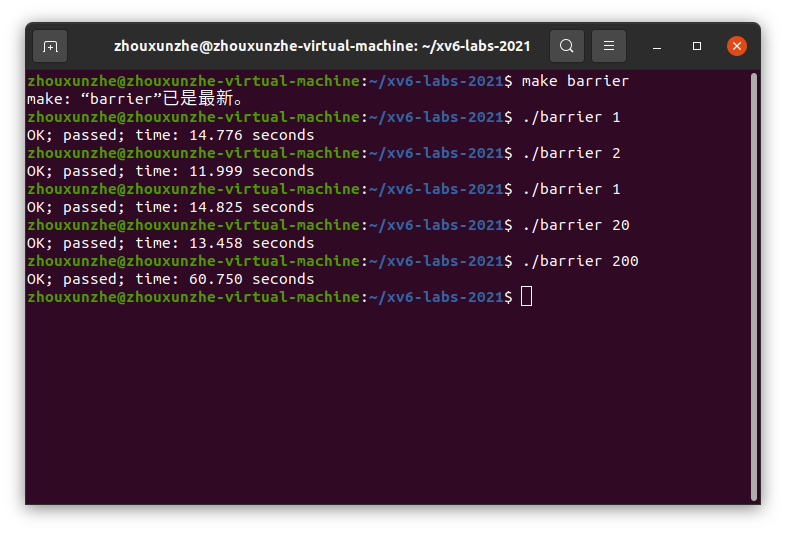


由于加了锁，各个线程之间互不干扰，所以结果也是符合预期的。



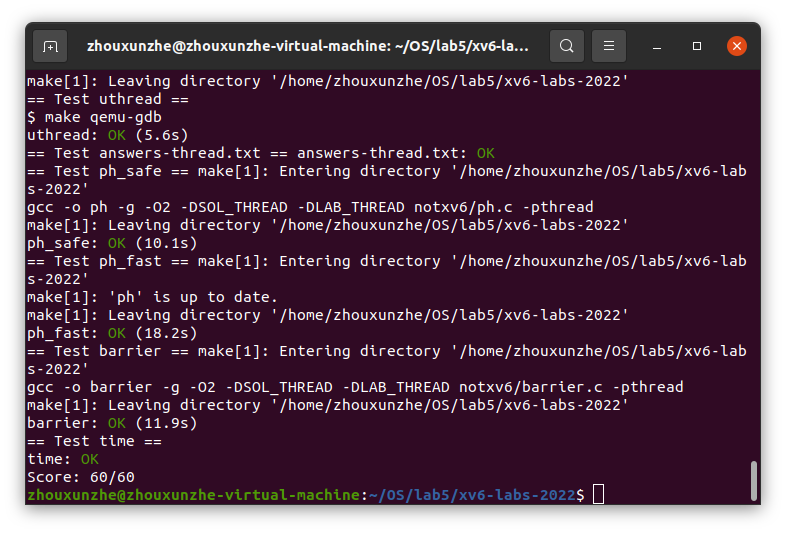
最后对barrier进行测试，结果也是输出OK。

由于在测试多个线程时，主观感受不同数量的线程，运行时间不尽相同，于是准备测试不同线程之间的运行时间的差异。学习ph中测试运行时间的方法，得到线程数为1、2、20、200时的运行时间。



发现随着线程数增大，其运行时间也会增多。其原理就是各个线程都要进行20000次循环，其复杂度为O（m\*n），m时线程数，n是循环次数。但是当线程数增多以后，由于多个线程可以同时调用，则会提高效率，所以当线程数差别不大时，多个线程运行时间会比一个线程少。

最后使用make grade进行检查，所有测试都通过了。

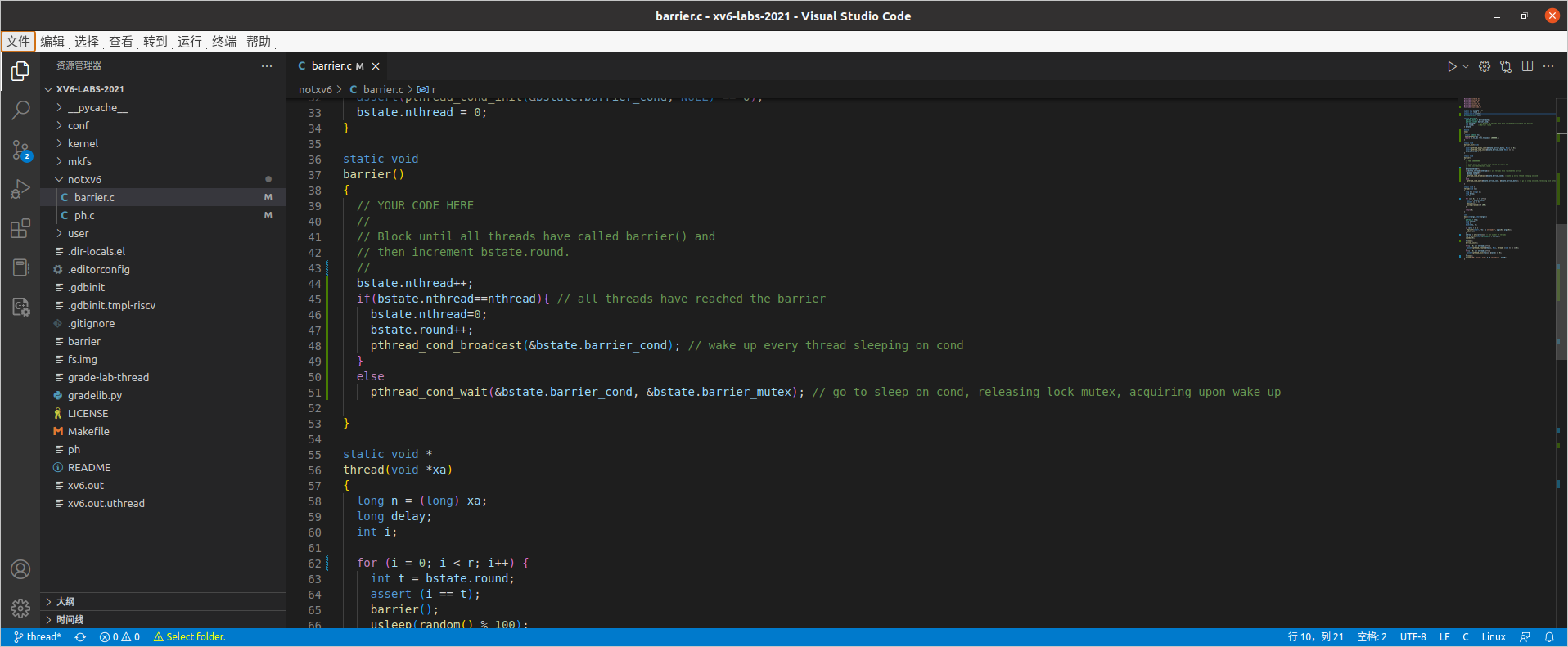


二、实验中碰到的问题。

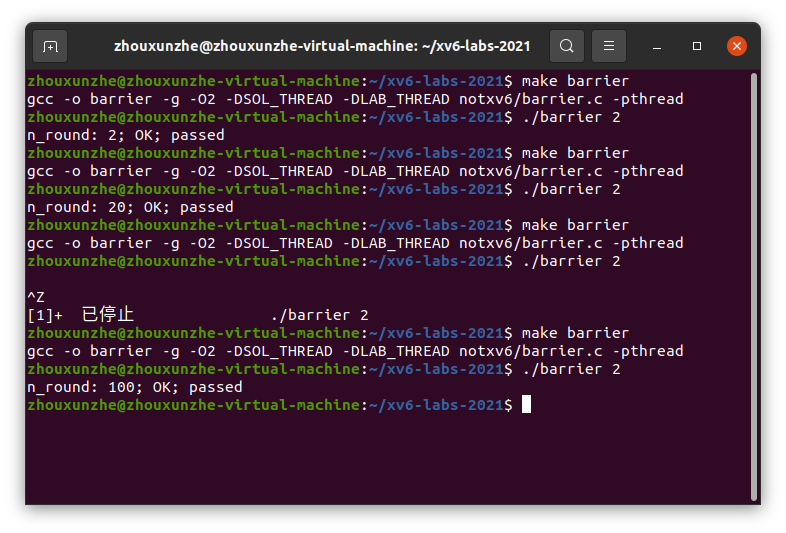
在barrier实验中：

首先执行make barrier和./barrier 2指令，由于barrier函数不完整，所以断言assert (i == t);会报错。仔细研究，发现这个断言目的是为了判断此时的round是否正确。

Barrier函数的目的就是记录多个线程的多个循环。在命令行的参数的含义是需要运行的线程数量。每当一个线程调用barrier（）函数，如果在本次循环有线程还未调用，则会进入wait状态等待所有线程调用，当所有线程都调用barrier以后，则会执行broadcast操作唤醒所有的线程。



具体实现后运行，发现运行结果会出现死锁的情况。当测试更改循环次数后，如果将次数降为2，20，100次后能正常运行并输出结果。但是当循环次数为200次及以上（只测试2\*10^n次和100次），则会出现死循环，没有运行结果，会一直卡在进程中。



如图所示，当使用两个进程时，测试200次时会出现死循环，没有运行结果。

仔细研究，发现是可能循环次数增加以后，可能出现一个线程对nthread自增的情况，另一个线程设置nthread=0，可能导致本应全部进程唤醒，但是最后一个进程却执行了等待操作，最后没有进程可以执行的情况，导致程序不能继续执行，所有进程都在等待被唤醒，从而没有输出结果。

最后的解决办法就是加入锁操作使得barrier执行的过程中不会出现其他进程干扰的情况。

三、实验总结

在本次实验中，我学习到了进程之间切换以及具体调用锁的原理以及代码的具体实现。通过对实验的调试，发现锁在进程切换中是一个不可或缺的事物，同时应该如何避免死锁也是一个需要考虑的问题。

通过本次实验，我更加理解了操作系统中的进程切换，为以后的学习打下坚实基础。