

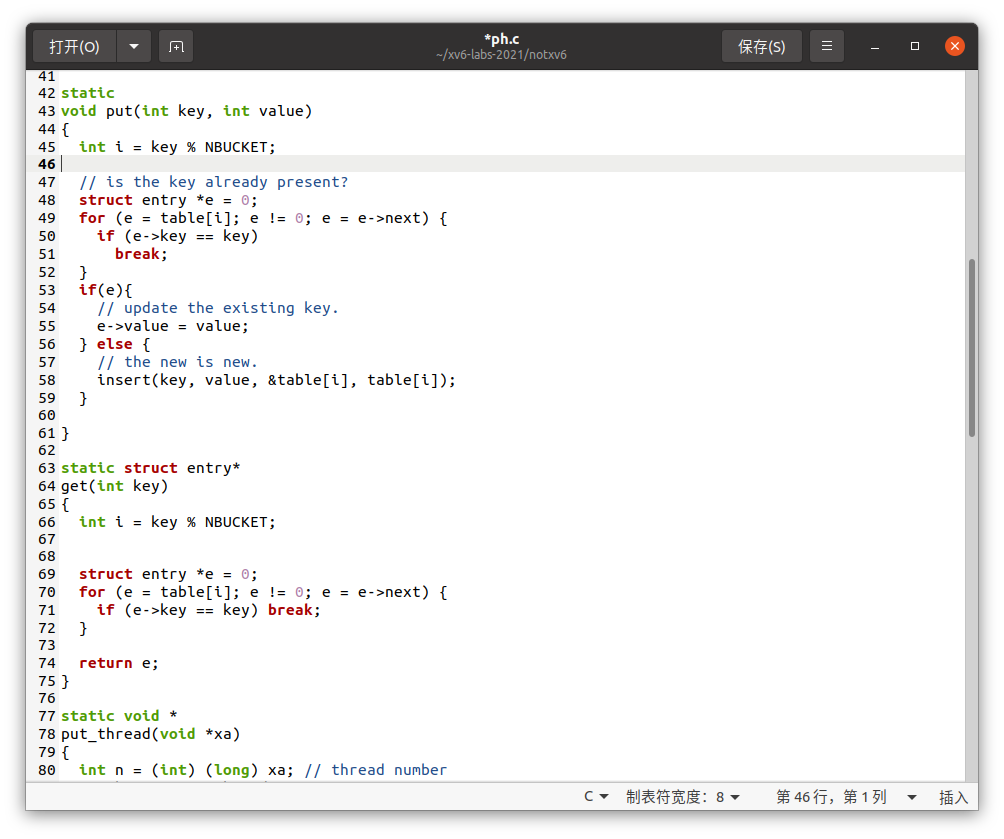
如图，./ph 1表示只有一个线程运行。./ph 2表示有两个线程同时运行。

Put表示将keys放入hash表中，个体表示从hash表中取keys。

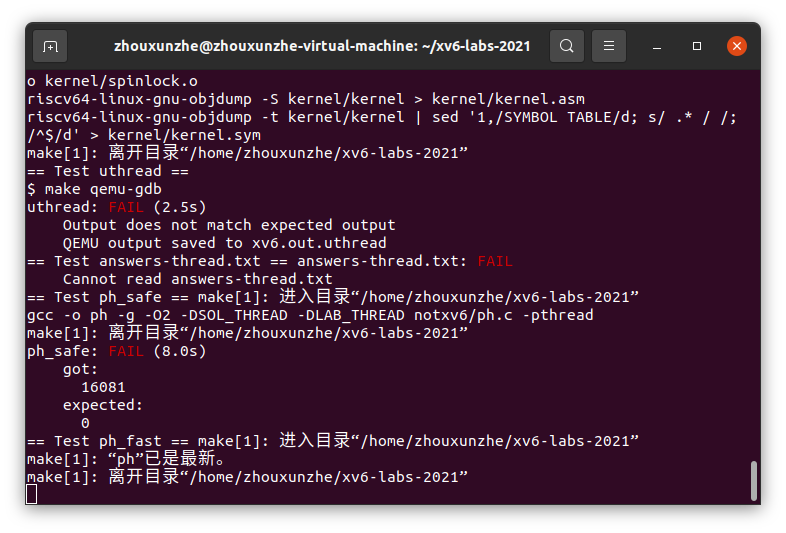
记录keys被put以后但是没有被get的数量。

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

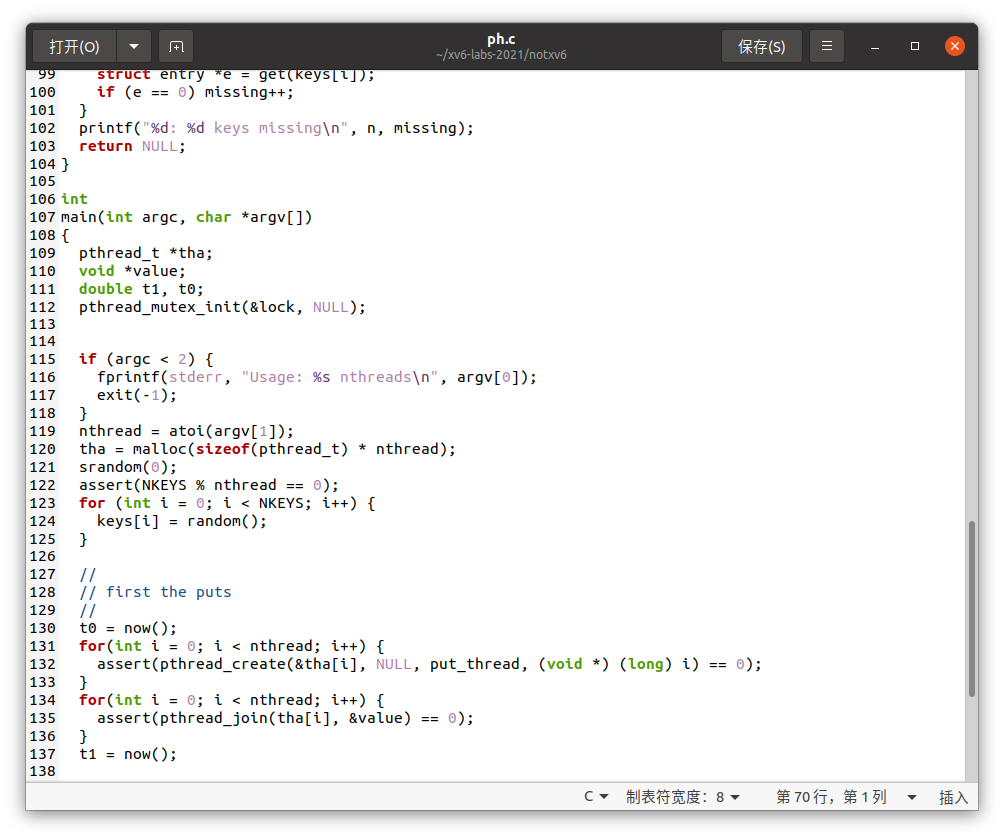
pthread\_mutex\_init(&lock, NULL); // initialize the lock

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

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



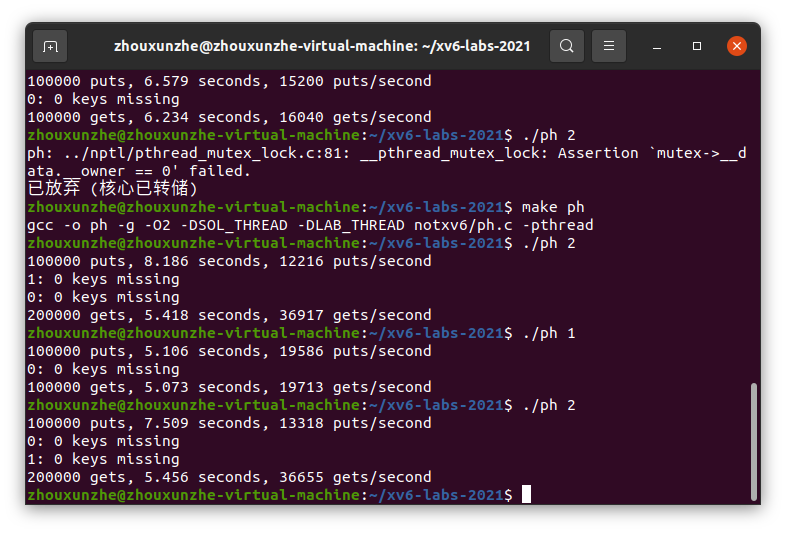
首先定义一个锁为全局变量，这样所有的函数都可以调用。



然后要调用锁，首先在主函数中将锁初始化。

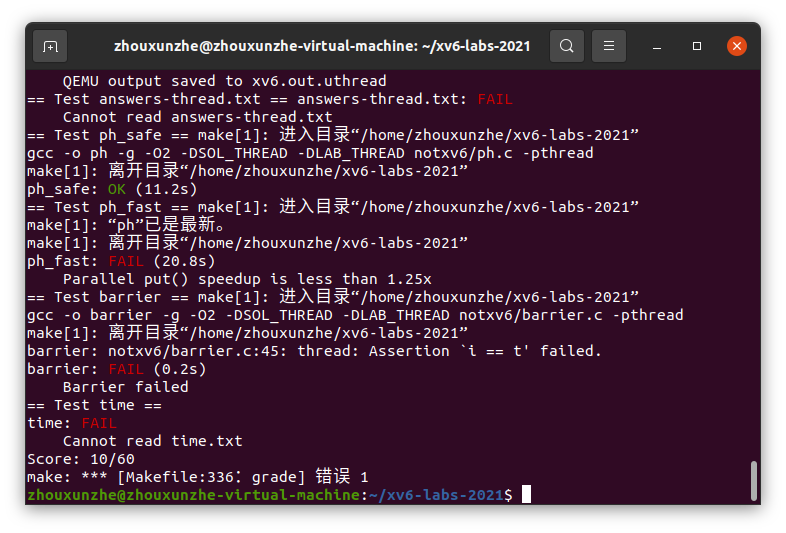


最后如图，在put函数的开始和结束分别加入语句pthread\_mutex\_lock(&lock);和pthread\_mutex\_unlock(&lock);实现获取锁和解锁的操作。



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。

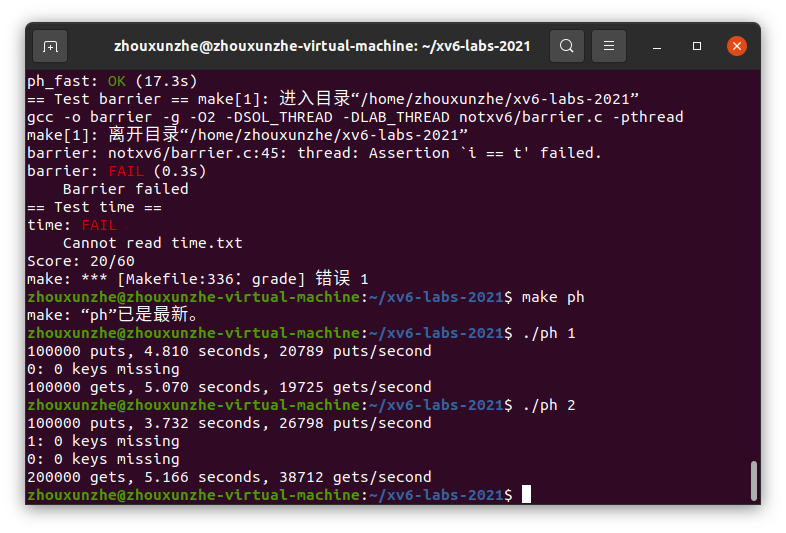
具体实现的思路即是，将put的一个锁转换为hash\_table.size()大小的锁，即lock[NBUCKET]，每次对其中一个lock[i]获取锁以及解锁即可。



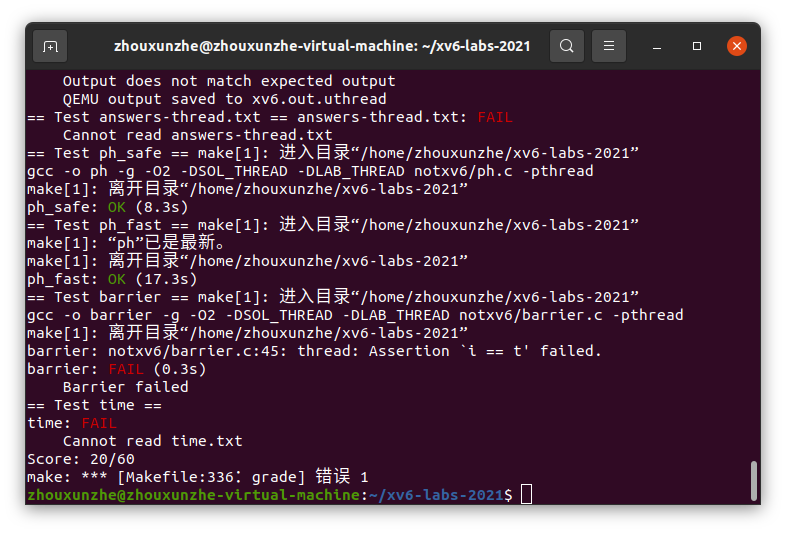
如图，只需要修改lock为一个大小为NBUCKET的容器即可，并且每次获取锁和解锁对象为需要写入的bucket。



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。