**Lab 5**

**Part A:** **Uthread: switching between threads**

本实验实现线程间的环境切换。首先在thread结构体中，添加变量heap数组，用以存放寄存器数据：

struct **thread** {

**uint64** heap[32];

char stack[**STACK\_SIZE**]; */\* the thread's stack \*/*

int state; */\* FREE, RUNNING, RUNNABLE \*/*

};

其中，heap中数值先后对应ra，sp，s0-s11寄存器中保存内容。由于caller-saved register，或volatile register会在线程切换之前将存储内容转移至堆栈中，因此不需要保存，只需保存callee-saved register （non-volatile register）的内容即可。之后，在thread\_create，亦即创建线程的阶段，向创建的thread结构体中heap的第0、1个元素（分别对应寄存器ra、sp）传入返回地址以及堆栈指针……

void

**thread\_create**(void (\*func)())

{

struct **thread** \*t;

for (t = all\_thread; t < all\_thread + **MAX\_THREAD**; t++) {

if (t->state == **FREE**) break;

}

t->state = **RUNNABLE**;

*// YOUR CODE HERE*

t->heap[0] = (**uint64**)func;

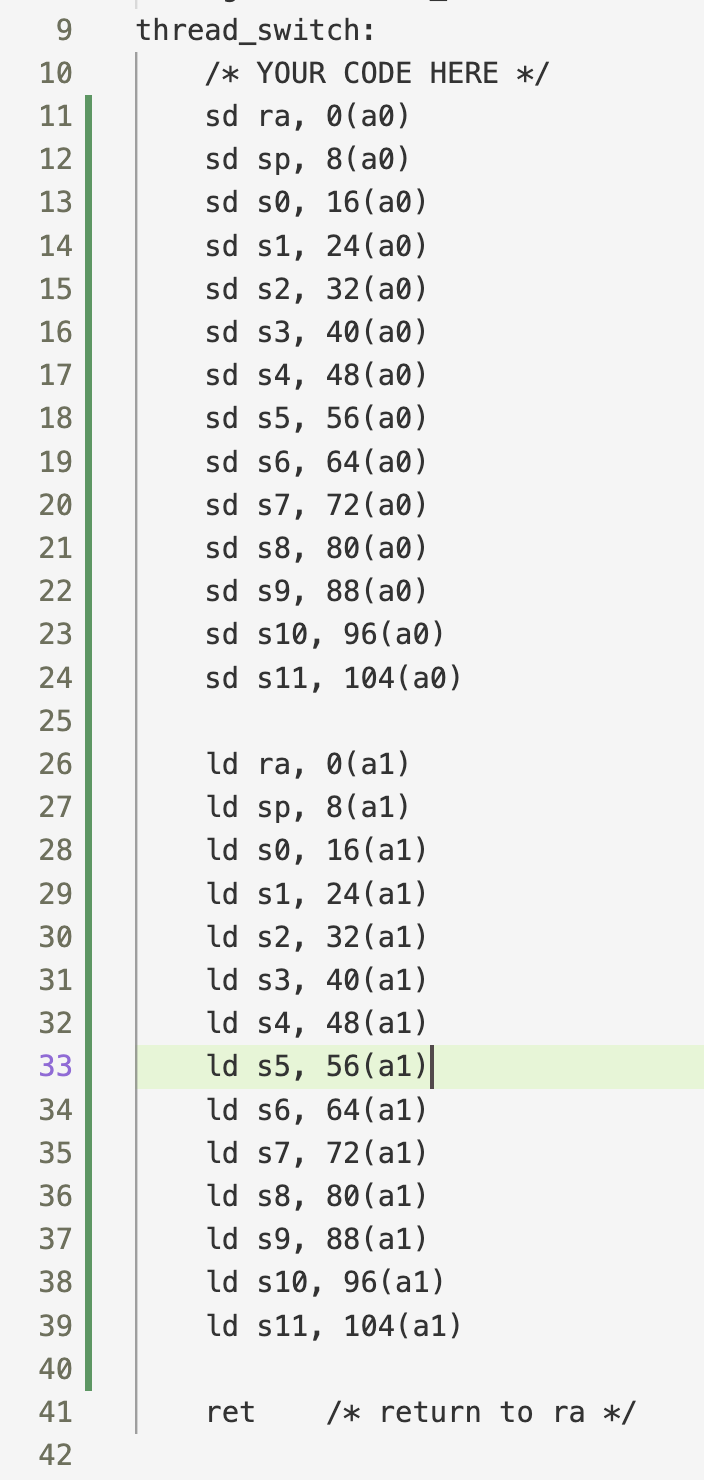
t->heap[1] = (**uint64**)&t->stack[**STACK\_SIZE**-1];

}

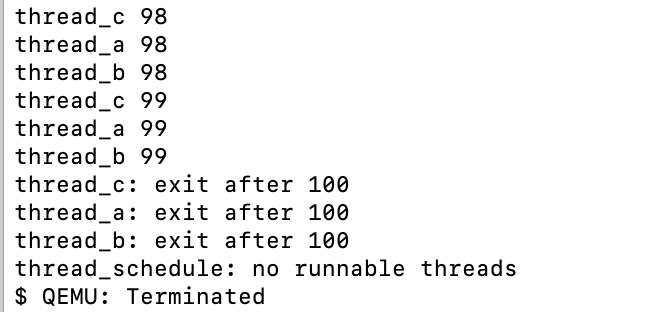
……并在thread\_schedule的相应部分调用线程切换函数。其中，函数的两个变量为thread结构体对应堆数组的首地址，因此在thread\_switch中我们可以通过事先约定好的寄存器顺序直接访存相关数据。

**thread\_switch**((**uint64**)t->heap, (**uint64**)next\_thread->heap);

thread\_switch代码如下。

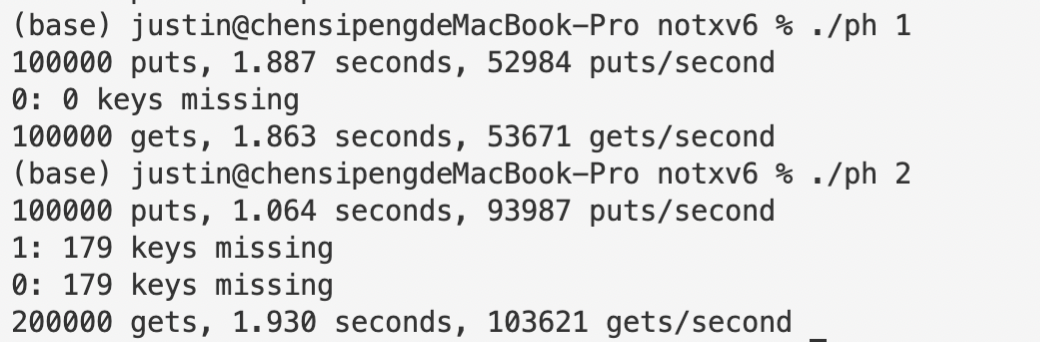


运行结果如下。



**Part B:** **Using threads**

在多线程的情况下，假设线程A与线程B交替运行，设想以下情况：线程A在执行put操作时，在工作指针已经指向数组尾部，但尚未插入新key的情况下，系统转而执行线程B。此时，线程B视野下的数组尾和A是一致的，因此，如果此时线程B先完成了插入表元的操作，系统进而继续执行A，那么A将会在B插入表元的位置再插入新表元，B插入的表元因而被覆盖，从而造成了key丢失的现象。



在不考虑效率的情况下，解决key丢失的方法可以非常直接：只需要让put函数成为一个完整的原子操作即可。

static

void **put**(int key, int value)

{

**pthread\_mutex\_lock**(&lock);

int i = key % **NBUCKET**;

*// is the key already present?*

struct **entry** \*e = 0;

for (e = table[i]; e != 0; e = e->next) {

if (e->key == key)

break;

}

if(e){

*// update the existing key.*

e->value = value;

} else {

*// the new is new.*

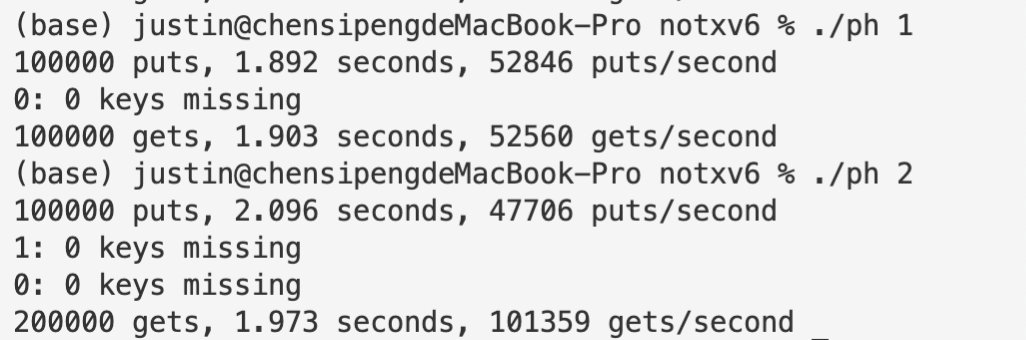
**insert**(key, value, &table[i], table[i]);

}

**pthread\_mutex\_unlock**(&lock);

}

值得一提的是，get函数即使不是原子操作也不会造成key丢失，因为get只涉及对数组元素的读操作，而不涉及写操作。完成以上修改后，可在如下的输出结果中看到key丢失的问题已经解决。



现在，尝试对锁进行优化，从而提升多线程的工作效率。首先，尽可能缩小上锁部分代码：由于对数组信息读取的部分不必是原子操作，因此仅对写部分上锁即可。另外，由于table中不同bucket的读写操作相互独立，因此应当允许对不同bucket的操作可以交替并行执行，亦即，对每个bucket单独分配一个互斥锁，而不应该简单地为整个表分配一个单独的锁。

以下为main函数中的互斥锁初始化部分：

for (int i = 0; i < **NBUCKET**; i++){

**pthread\_mutex\_init**(&lock[i], **NULL**);

}

修改后的put函数为如下所示：

static

void **put**(int key, int value)

{

int i = key % **NBUCKET**;

*// is the key already present?*

struct **entry** \*e = 0;

for (e = table[i]; e != 0; e = e->next) {

if (e->key == key)

break;

}

**pthread\_mutex\_lock**(&lock[i]);

if(e){

*// update the existing key.*

e->value = value;

} else {

*// the new is new.*

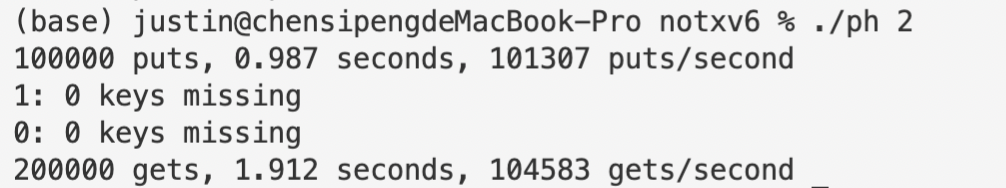
**insert**(key, value, &table[i], table[i]);

}

**pthread\_mutex\_unlock**(&lock[i]);

}

完成以上所有修改后重新运行ph脚本。



**Part C:** **Barrier**

调用barrier功能时，首先获得互斥锁，使得对nthread的增加成为原子操作，确保函数能够正确执行。随后进行条件判断：若当前状态下nthread变量尚未达到线程总数，则通过cond\_wait功能解开互斥锁并等待其他线程；若已经达到线程总数，则通过cond\_broadcast唤醒所有正在等待的线程从中断处继续执行，并将nthread值置为零（nthread表示当前到达barrier的线程总数），round值增加1。

static void

**barrier**()

{

*// YOUR CODE HERE*

*//*

*// Block until all threads have called barrier() and*

*// then increment bstate.round.*

*//*

**pthread\_mutex\_lock**(&bstate.barrier\_mutex);

bstate.nthread++;

if (bstate.nthread == nthread){

**pthread\_cond\_broadcast**(&bstate.barrier\_cond);

bstate.round++;

bstate.nthread = 0;

}

else{

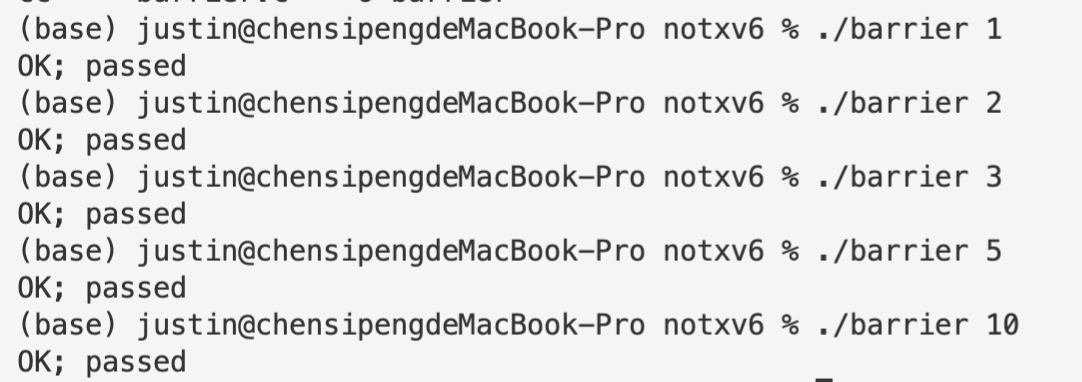
**pthread\_cond\_wait**(&bstate.barrier\_cond, &bstate.barrier\_mutex);

}

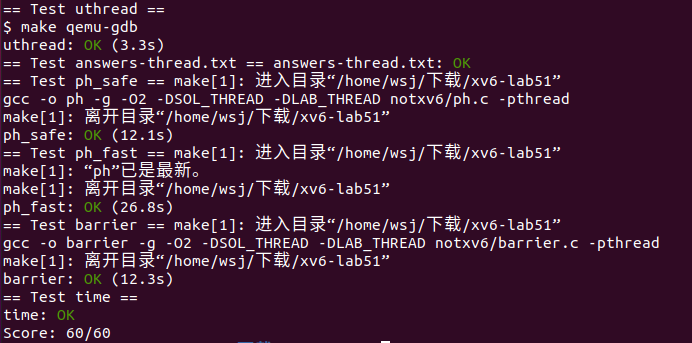
**pthread\_mutex\_unlock**(&bstate.barrier\_mutex);

}

测试结果如下。



由于个人目前配置的lab环境为amazon aws单核云服务器，无法正确进行ph实验，因此借用了同学电脑运行make grade。



**Thoughts**

本实验以多线程为主题，实现了进程切换时寄存器数据的存储和barrier，并通过多线程并行读写的情境探索了读写操作的原子性及锁对进程执行效率的影响，加深了对线程、锁即条件变量的理解。