**实验七 Multithreading**

**7.1 实验目的**

1. 理解多线程的概念

2. 通过实验完成如何实现线程的切换

3. 学习如何完成多线程之间的同步

**7.2 实验内容**

7.2.1 Uthread:switiching between threads（moderate）

您的工作是提出一个创建线程和保存/恢复寄存器以在线程之间切换的计划，并实现该计划。完成后，make grade应该表明您的解决方案通过了uthread测试。

7.2.2 Using threads（moderate）

您的第一项工作是修改内核，以便每个进程在内核中执行时都使用自己的内核页表副本。修改struct proc以维护每个进程的内核页表，并修改调度器以在切换进程时切换内核页表。对于这个步骤，每个进程的内核页表应该与现有的全局内核页表相同。如果usertests运行正常，则通过这部分实验。

7.2.3 Barrier（moderate）

为了避免这种事件序列，请在notxv6/ph.c中的put和get中插入lock和unlock语句，以便在两个线程中丢失的密钥数始终为0。相关的pthread调用包括：

pthread\_mutex\_t lock；//申报锁

pthread\_mutex\_init（&lock，NULL）；//初始化锁

pthread\_mutex\_lock（&lock）；//获取锁

pthread\_mutex\_unlock（&lock）；//释放锁

当make grade说您的代码通过ph\_安全测试时，您就完成了，该测试需要两个线程的零缺失键。在这一点上，PHU快速测试失败是可以的。

7.2.4 submit

提交实验。

**7.3 实验过程**

7.3.1 uthread: switching between threads

（1）在thread结构体里添加寄存器字段。

struct thread {

  /\*stored registers\*/

  uint64 ra;

  uint64 sp;

  // callee-saved

  uint64 s0;

  uint64 s1;

  uint64 s2;

  uint64 s3;

  uint64 s4;

  uint64 s5;

  uint64 s6;

  uint64 s7;

  uint64 s8;

  uint64 s9;

  uint64 s10;

  uint64 s11;

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

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

};

（2）参考kernel/swtch.S，在uthread\_switch.S中保存当前线程的寄存器，恢复即将要切换到线程的寄存器。

thread\_switch:

/\* YOUR CODE HERE \*/

sd ra, 0(a0)

sd sp, 8(a0)

sd s0, 16(a0)

sd s1, 24(a0)

sd s2, 32(a0)

sd s3, 40(a0)

sd s4, 48(a0)

sd s5, 56(a0)

sd s6, 64(a0)

sd s7, 72(a0)

sd s8, 80(a0)

sd s9, 88(a0)

sd s10, 96(a0)

sd s11, 104(a0)

ld ra, 0(a1)

ld sp, 8(a1)

ld s0, 16(a1)

ld s1, 24(a1)

ld s2, 32(a1)

ld s3, 40(a1)

ld s4, 48(a1)

ld s5, 56(a1)

ld s6, 64(a1)

ld s7, 72(a1)

ld s8, 80(a1)

ld s9, 88(a1)

ld s10, 96(a1)

ld s11, 104(a1)

ret /\* return to ra \*/

（3）在uthread.c中的thread\_create()函数中添加保存新线程返回地址和栈指针

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->ra=(uint64)func;

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

}

（4）根据提示在uthread.c中的thread\_schedule()添加调用，调用thread\_switch()

void

thread\_schedule(void)

{

  struct thread \*t, \*next\_thread;

  /\* Find another runnable thread. \*/

  next\_thread = 0;

  t = current\_thread + 1;

  for(int i = 0; i < MAX\_THREAD; i++){

    if(t >= all\_thread + MAX\_THREAD)

      t = all\_thread;

    if(t->state == RUNNABLE) {

      next\_thread = t;

      break;

    }

    t = t + 1;

  }

  if (next\_thread == 0) {

    printf("thread\_schedule: no runnable threads\n");

    exit(-1);

  }

  if (current\_thread != next\_thread) {         /\* switch threads?  \*/

    next\_thread->state = RUNNING;

    t = current\_thread;

    current\_thread = next\_thread;

    /\* YOUR CODE HERE

     \* Invoke thread\_switch to switch from t to next\_thread:\*/

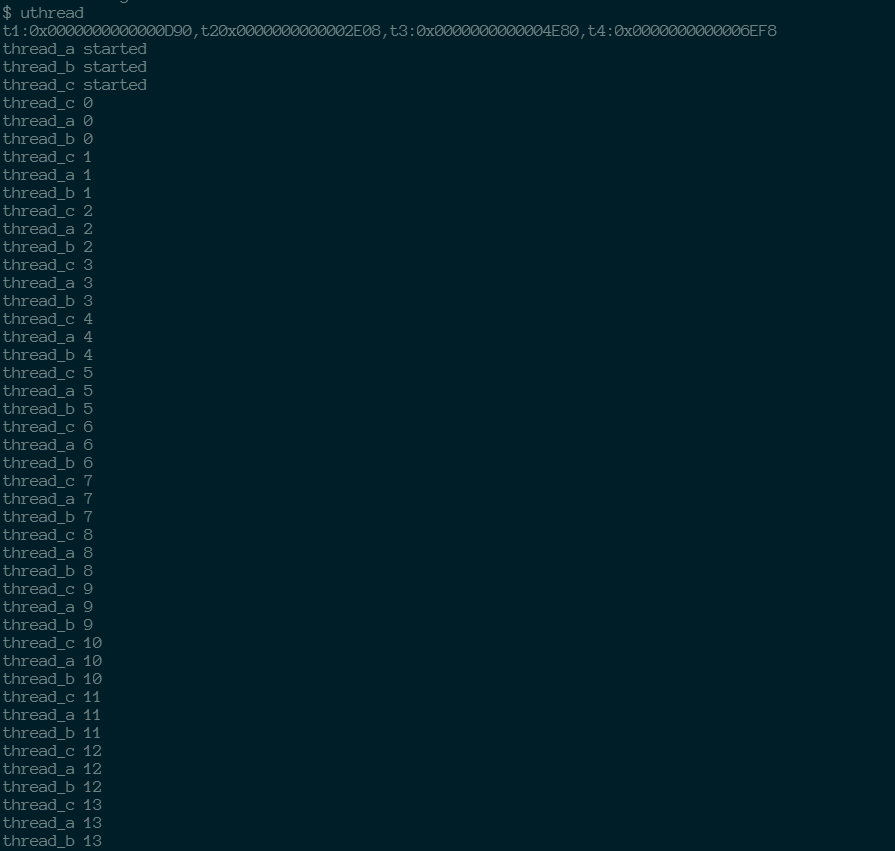
    thread\_switch((uint64)t,(uint64)next\_thread);

  } else

    next\_thread = 0;

}

（5）运行uthread





7.3.2 using threads

（1）根据提示运行make ph，./ph 1以及./ph 2，其中./ph 1无丢失的键，./ph 2由丢失的键 。

（2）Why are there missing keys with 2 threads, but not with 1 thread? Identify a sequence of events with 2 threads that can lead to a key being missing. Submit your sequence with a short explanation in answers-thread.txt

答：因为由多个线程同时工作，所以当其中一个进程给哈希表填入一个键的时候其他进程也可能在添加键，当两个哈希值不相等时插入成功，但两个哈希值相同时会导致后写的覆盖另一个，导致键的丢失。

（3）为提高多线程的效率设计锁数组。

pthread\_mutex\_t locks[NBUCKET];//declare a lock

（4）根据代码可知数组table的元素table[i]为链表，其中存放元素entry，并行访问不同的table不会导致数据的丢失，所以修改put函数。

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(&locks[i]);

  if(e){

    // update the existing key.

    e->value = value;

  } else {

    // the new is new.

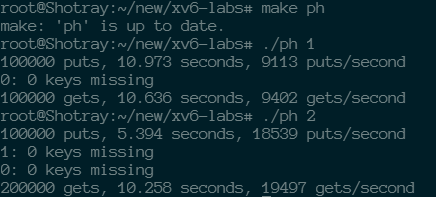
    insert(key, value, &table[i], table[i]);

  }

  pthread\_mutex\_unlock(&locks[i]);

}

（5）运行./ph 2



7.3.3 barrier

（1）通过round调用barrier的线程进行技术，并用pthread\_cond\_wait和pthread\_cond\_broadcast进行线程在cond上的等待和唤醒。一个线程只有在其他线程调用pthread\_cond\_wait进入等待之后才能获得mutex。

（2）barrier实现如下

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+=1;

  if(bstate.nthread<nthread)

  {

     pthread\_cond\_wait(&bstate.barrier\_cond,&bstate.barrier\_mutex);//go to sleep on cond,releasing lock mutes acquirint upon wake up

  }

  else

  {

      bstate.round+=1;//# of barrier calls

      bstate.nthread=0;

      pthread\_cond\_broadcast(&bstate.barrier\_cond); // wake up every thread sleeping on cond

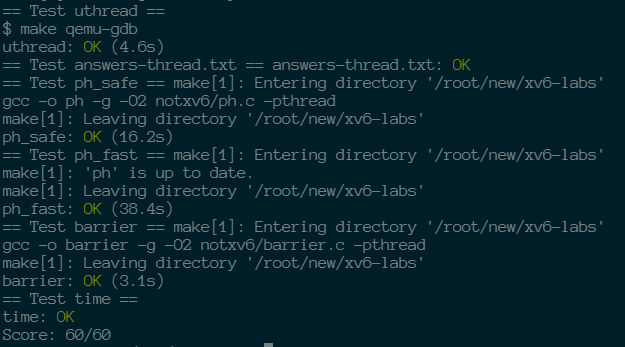
  }

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

}

**7.4 实验结果**

通过make grade对所有实验内容进行测试，结果如下图所示。



**7.5 实验小结**

本次实验是有关多线程的编写以及锁的运用。我们在本次实验中实现了线程的切换，来保存当前线程以及恢复将要切换到的线程，并且掌握如何通过锁和条件信号量来完成多线程之间的同步，对线程这一概念掌握的更加深刻。