Lab6：Multithreading

该实验由Uthread: switching between threads、Using threads、Barrier三部分组成。

1. Uthread: switching between threads

实验目的

设计并实现一个用户级线程系统的上下文切换机制。补充完成一个用户级线程的创建和切换上下文的代码。需要创建线程、保存/恢复寄存器以在线程之间切换，并且确保解决方案通过测试。

实验步骤

1. 参考’kernel/proc.h’中的’struct context’结构体，在’kernel/uthread.c’中设计’struct ctx’结构体用于保存寄存器

struct ctx {

    uint64 ra;

    uint64 sp;

    uint64 s0;

    uint64 s1;

    uint64 s2;

    uint64 s3;

    uint64 s4;

    uint64 s5;

    uint64 s6;

    uint64 s7;

    uint64 s8;

    uint64 s9;

    uint64 s10;

    uint64 s11;

};

1. 为线程结构体添加上下文字段

struct thread {

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

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

  struct thread\_context context;       // 线程上下文

};

1. 修改thread\_create()函数，使其能在线程数组中遍历，发现一个未初始化的线程就执行初始化。

  // YOUR CODE HERE

  t->context.ra = (uint64)func;

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

1. 修改thread\_schedule()函数，它负责负责用户多线程间的调度。它从当前线程的位置开始，在线程数组中寻找一个 `RUNNABLE` 状态的线程进行运行，在找到线程后调用函数 `thread\_switch()`进行线程的切换。

    thread\_switch((uint64)&t->context, (uint64)&current\_thread->context);

5.参考 `swtch` 代码在user/uthread\_switch.S中实现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)

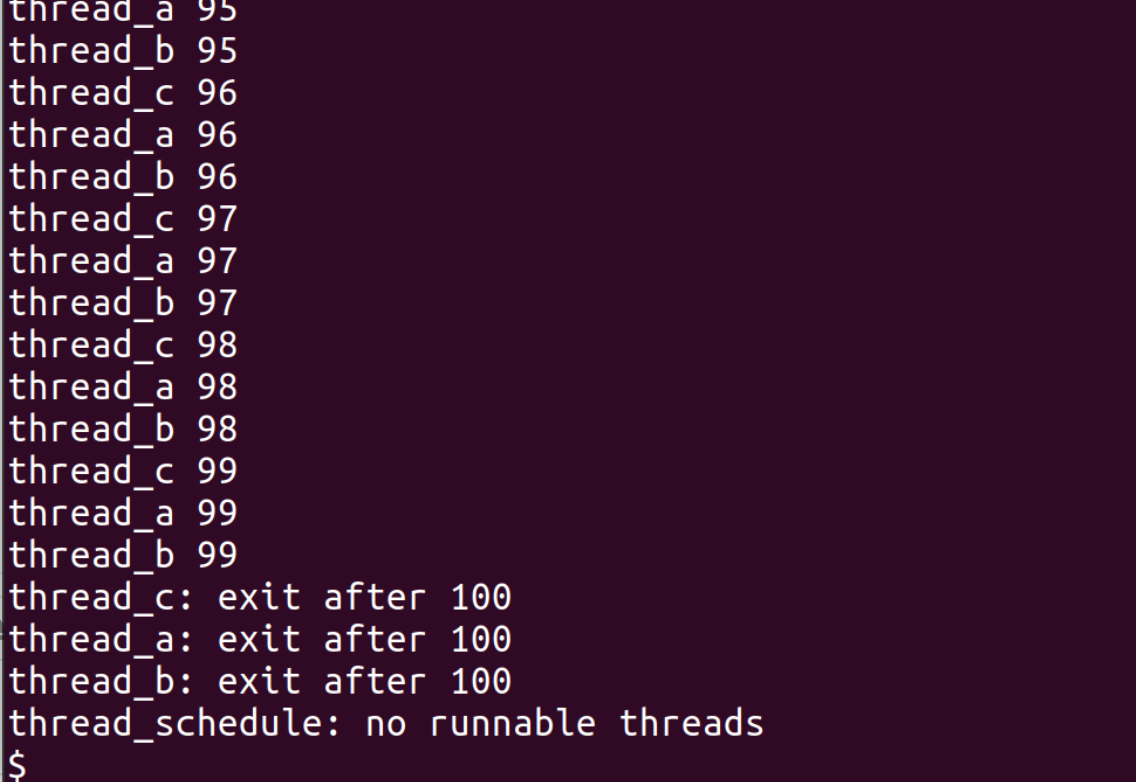
ld s8, 80(a1)

ld s9, 88(a1)

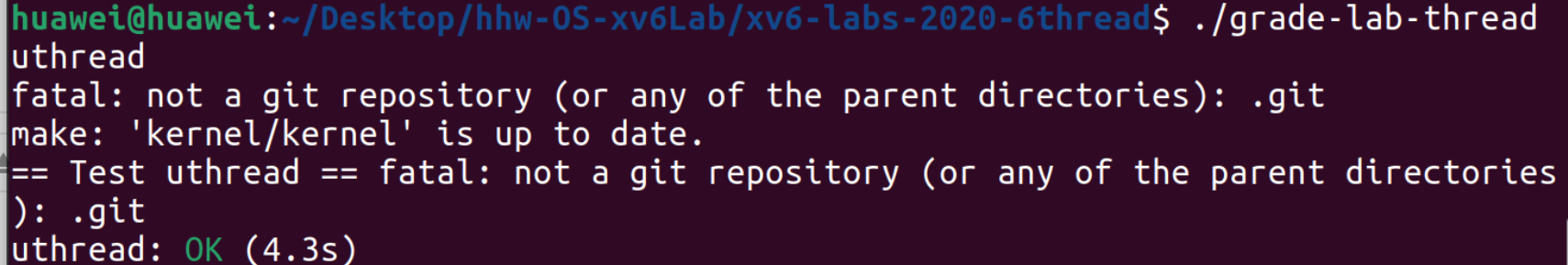
ld s10, 96(a1)

ld s11, 104(a1)

6.进入xv6，输入uthread后得到预期结果：



7. 在终端里运行 ./grade-lab-thread uthread：



遇到的问题与心得

遇到的问题：

在修改thread\_schedule()时要从t切换到next\_thread，不是从current\_thread切换到next\_thread

实验心得

本次实验使我熟悉了多线程编程，理解了进程切换方式和执行顺序。

2. Using threads

实验目的

探索使用哈希表的线程和锁的并行编程，使用 UNIX pthread 线程库实现一个线程安全的哈希表。

实验步骤

1. 在ph.c中声明多线程的锁，并且在main函数里初始化

pthread\_mutex\_t lock[NBUCKET];

  // 初始化锁

  for (int i = 0; i < NBUCKET; i++){

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

  }

1. 在put函数里上锁

线程的安全问题是对桶中的链表进行操作而导致的，需要在链表操作的前后加锁，在`put`函数读写bucket之前加锁，在函数结束时释放锁。

  if(e){

    // update the existing key.

    e->value = value;

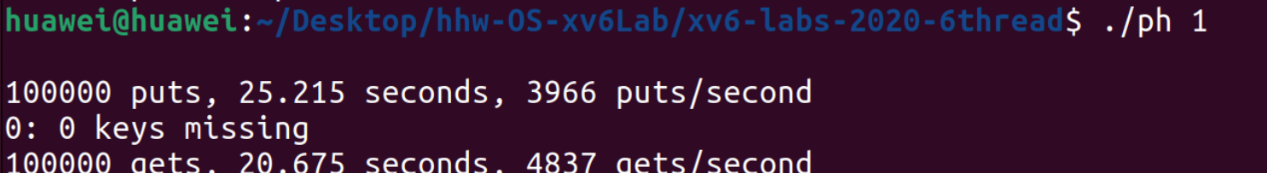
  } else {

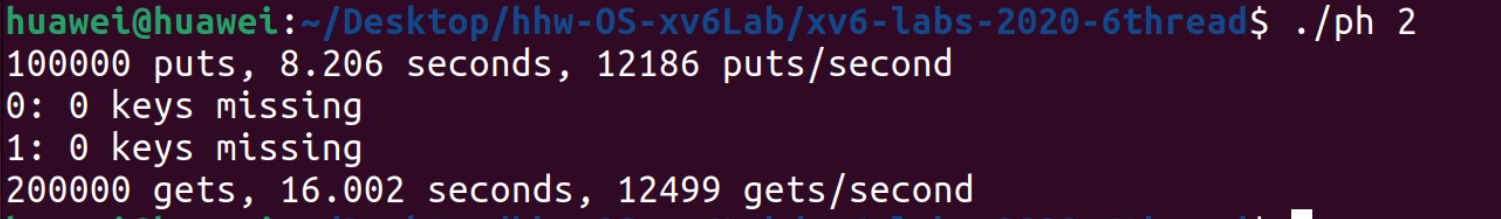
    pthread\_mutex\_lock(&lock[i]);

    // the new is new.

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

3.再使用make ph 编译notxv6/ph.c ，运行 ./ph 1、./ph 2：



****

遇到的问题与心得

遇到的问题：

当不同线程对同一个bucket进行put操作时，可能会覆盖前一个put的结果，造成错误。

当整个哈希表被加锁后整体的性能甚至不如之前，原因是每一时刻只能有一个线程进行操作，相当于单线程了，优化思路是降低锁的粒度，从对于整个哈希表加锁到对于每个bucket加锁。

实验心得

在实验中，实现了进程切换、运用锁来完成线程间的互斥，理解了锁的含义，能够判断何时、何处应该加锁。

3. Barrier

实验目的

实现 barrier 函数。其作用是：当进程到达 barrier 函数调用时，会开始等待其他进程，当所有进程都到达 barrier 时，才停止等待。

实验步骤

1. 实验一个屏障’barrier()’函数，它首先会加锁然后判断到达屏障点的线程数量，如果所有线程都到达了就唤醒其他线程，否则就wait.

static void

barrier()

{

  // YOUR CODE HERE

  //

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

  // then increment bstate.round.

  //

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

  // 所有进程都要调用一次barrier

  if (++bstate.nthread == nthread) {

    ++bstate.round; // 轮数++

    bstate.nthread = 0; // 重置nthread

    pthread\_cond\_broadcast(&bstate.barrier\_cond); // 唤醒所有的睡眠线程

  }

  else {

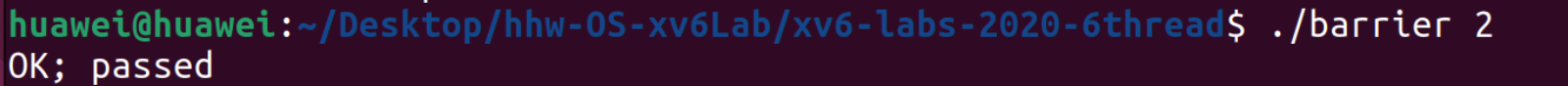
    pthread\_cond\_wait(&bstate.barrier\_cond, &bstate.barrier\_mutex); // 等待

  }

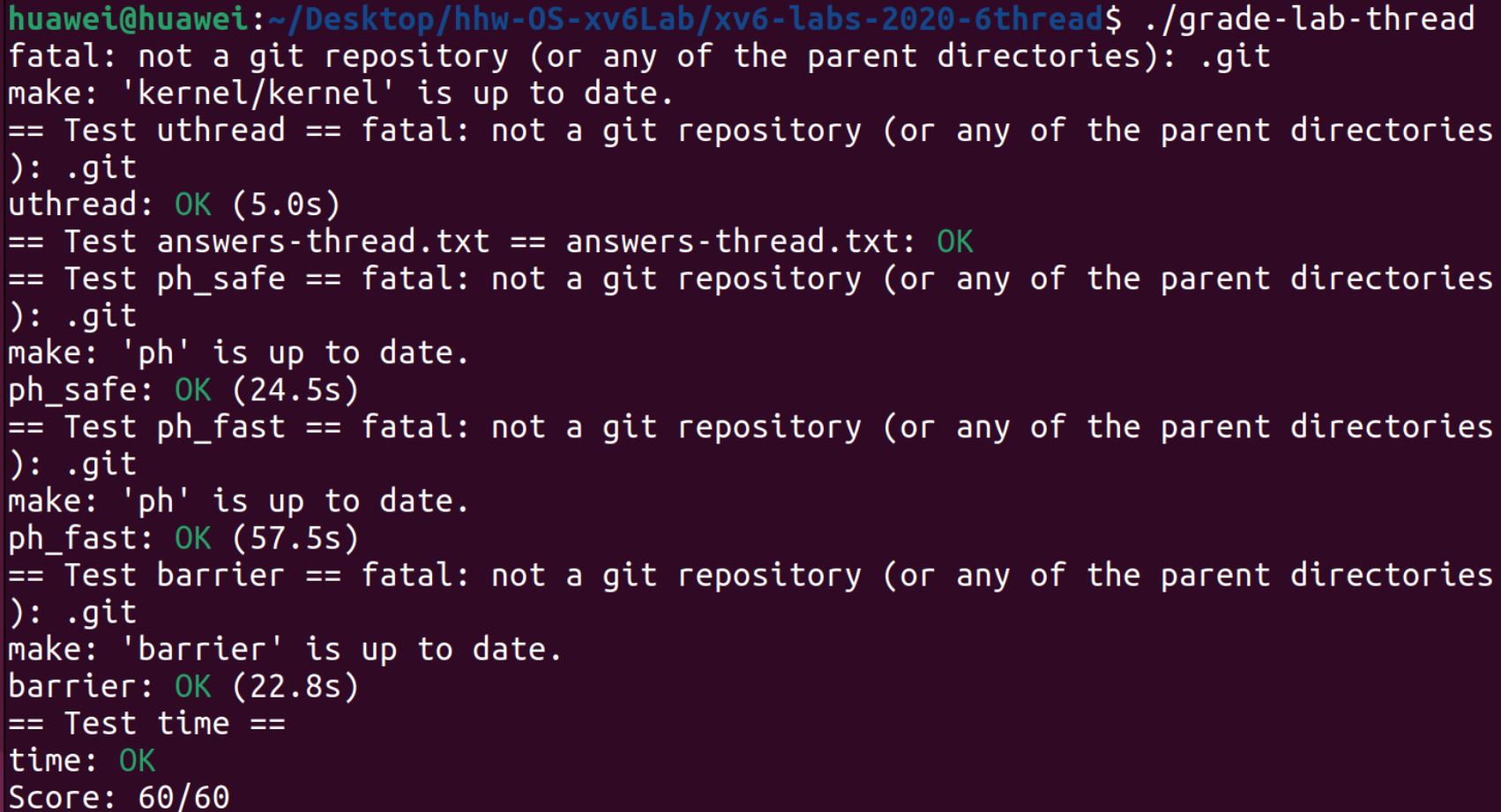
  pthread\_mutex\_unlock(&bstate.barrier\_mutex);

}

2.保存后在终端里执行make barrier 编译notxv6/barrier.c ，运行./barrier 2：



3.退出xv6，在终端运行./grade-lab-thread:



遇到的问题与心得

遇到的问题：

某个线程在等待屏障时被意外唤醒，导致线程同步错误。

解决办法：在等待屏障时，始终使用循环检查条件，以避免线程在不满足条件时被错误唤醒。

实验心得

运用锁来提高效率是一种很重要的方法，其中选取上锁的粒度和上锁的时机很重要，关乎着能否提高效率，如果粒度选取不好，甚至会加大时间和空间开销。