Lab5_Multithreading

20307130350 信息安全 陈丹纯

Uthread: switching between threads

实验内容

设计用户级线程系统上下文切换机制并实现。

实验思路

通过xv6 book对进程调度及其上下文的机制的学习，该实验可以仿照进程的上下文切换，给struct thread也增加一个用户级线程的上下文`context`，然后在`uthread_switch.s`中实现store old and restore new.

实验代码

1. 在`user/uthread.c`中创建一个`struct context`并在thread中添加一个context字段。(直接copy`kernel/proc.h`的context结构)

```c
//add for lab
struct context {
  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;
};


struct thread {
  char       stack[STACK_SIZE]; /* the thread's stack */
  int        state;             /* FREE, RUNNING, RUNNABLE */
  struct     context context;
};
```

2. 在`uthread_switch.S`中保存和恢复上下嗯（仿照`kernel/swtch.S`）

```
        .globl thread_switch
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)
```

3. 在`thread_create`中初始化ra和sp。

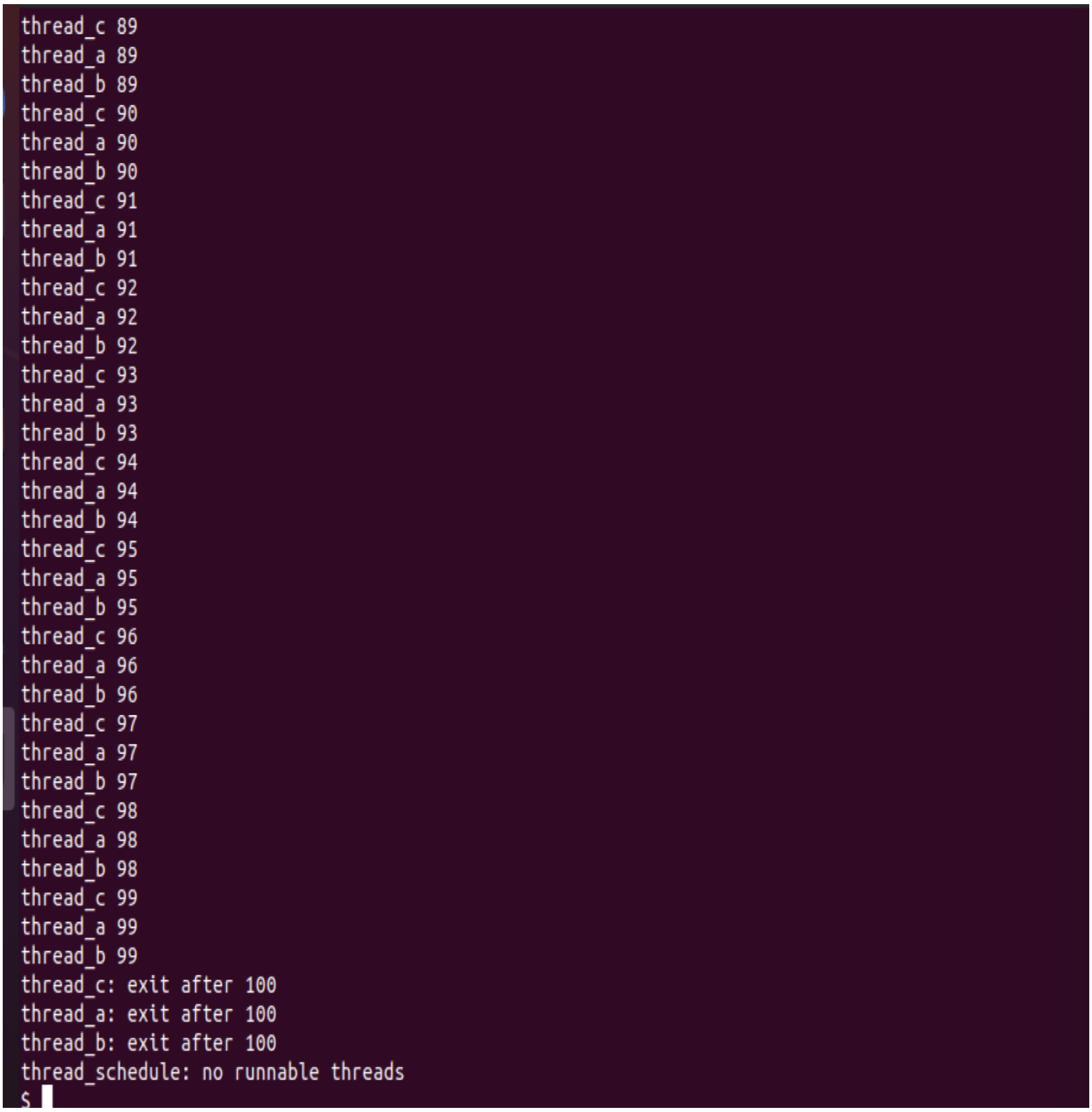
```c
void
thread_create(void (*func)())
{
  struct thread *t;

  for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
    if (t->state == FREE) break;
  }
  t->state = RUNNABLE;
  // YOUR CODE HERE
  t->context.ra = (uint64)func;
  t->context.sp = (uint64)&t->stack[STACK_SIZE-1];
}

```

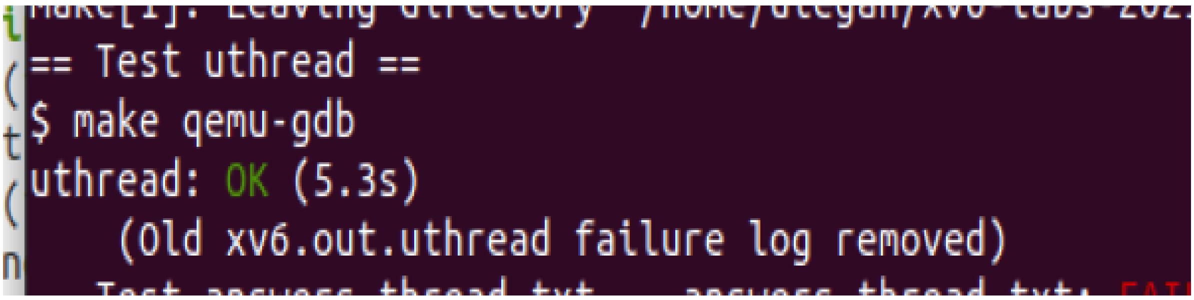
4. 在`thread_shedule`中切换上下文。

```c
  }

  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(??, ??);
     */
     thread_switch((uint64)(&t->context), (uint64)(&current_thread->context));
  } else
    next_thread = 0;
}
```

实验结果

```
qemu-system-riscv64 -machine virt -bios none -kernel kernel/kernel -m 128M -smp 3 -nographic -drive fil
=fs.img,if=none,format=raw,id=x0 -device virtio-blk-device,drive=x0,bus=virtio-mmio-bus.0

xv6 kernel is booting

hart 2 starting
hart 1 starting
init: starting sh
$ uthread
thread_a started
thread_b started
thread_c started
thread_c 0
thread_a 0
thread_b 0
thread_c 1
thread_a 1
thread_b 1
thread_c 2
thread_a 2
thread_b 2
thread_c 3
thread_a 3
thread_b 3
thread_c 4
thread_a 4
thread_b 4
thread_c 5
thread_a 5
thread_b 5
thread_c 6
thread_a 6
thread_b 6
thread_c 7
```





实验问题

thread_switch只需要保存/恢复被调用者保存的寄存器。 为什么?

caller-saved寄存器由调用的C代码保存在堆栈上，switch保存了ra寄存器和sp，可以从新的上下文中恢复寄存器，新的上下文中保存着前一次switch所保存的寄存器值。当switch返回时，它返回到被恢复的ra寄存器所指向的指令，也就是新线程之前调用switch的指令。此外，sp会指向新线程的栈。

实验反思

通过本实验，深入了解了context的切换机制。

Using threads

实验内容

分析

1. There is something wrong in `notxv6/ph.c`, thus making the code do not function as expected when working with multi-threads.
2. Check out the `ph.c` to find out the reasons, especially in `put()` and `insert()`. Write them in the `answers-thread.txt`.
3. Modify the `ph.c` especially `put()` and `get()` to lock and unlock the **critical sections**. Make sure the modified can pass the `ph_safe` by running `make grade` in terminal.
4. Try more times to optimize the performances to pass the `ph_fast`.

原文

more details in [Lab: Multithreading (mit.edu)](#).

实验分析

实验代码和answers-thread.txt已上传git。

多线程出错的原因

因为table是global variable，所以当两个及以上多个线程正好在同一个table中put()时，以两个线程为例：

1. **error**: `insert()` 中的插入链表操作可能会导致覆盖。eg.当线程0运行完`e->next=n;`之后发生context switch，线程1也执行put`e->next=n`之后并继续执行`*p=e`；再切换为线程0执行`*p=e`,先执行`*p=e`的线程会丢失新添加的key。
2. **warning**: put()中执行`e->value=value`来更新存在的key时会导致覆盖，导致不同的线程丢失率不同。比如两个线程都要修改同一个 entry, 但由于调度先后执行`e->value=value`，则先执行该语句的会丢失更新的key，另一线程得以保留。因此两个线程对key的丢失不同。
3. **error**: 两个线程一个在执行`put()`时恰好执行到`insert(key,value,&table,table[i])`时，切换到另一个线程执行`insert()`中的`*p=e*`，如果针对同一个table，再切换回来会导致`put()`中的`insert(args)`中的`table`变化，导致key丢失。

```
static void *
put_thread(void *xa)
{
  int n = (int) (long) xa; // thread number
  int b = NKEYS/nthread;

  for (int i = 0; i < b; i++) {
    put(keys[b*n + i], n);
  }

  return NULL;
}
```

proc.h × swtch.S × ph.c ×

```c
insert(int key, int value, struct entry **p, struct entry *n)
{
  struct entry *e = malloc(sizeof(struct entry));
  e->key = key;
  e->value = value;
  e->next = n;
  *p = e;
}

static
void put(int key, int value)
{

  int i = key % NBUCKET;

  pthread_mutex_lock(lock+i);
  // 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+i);

}

```

Lock方案

根据上面多线程出错原因分析，是对全局变量`table`操作时需要互斥。要保证0 missing，只需解决`原因1`和`原因3`即`insert()`中对entry节点的覆盖对其他线程`insert()`或者`put()中insert()`。因此有多种加锁方案。

1. 对hash表的每个table的`insert`（smallest critical section）都加互斥锁

1. 因为要对于每个table都加锁，定义**全局变量**`lock[NBUCKET]`；

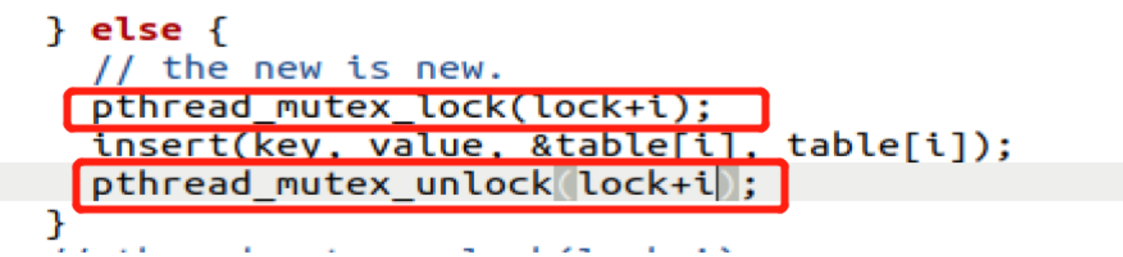
```
pthread_mutex_t lock[NBUCKET];
```

2. 在`main()`中初始化锁。

```
for(int i=0;i<NBUCKET;i++){
    pthread_mutex_init(lock+i,NULL);
}
```

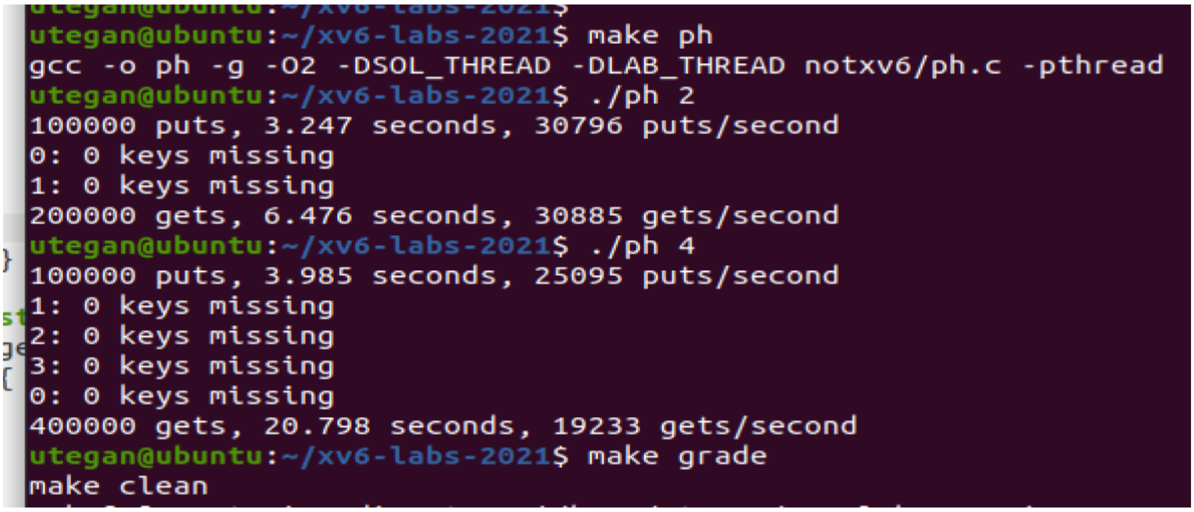
3. `the smallest critical section`为：

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

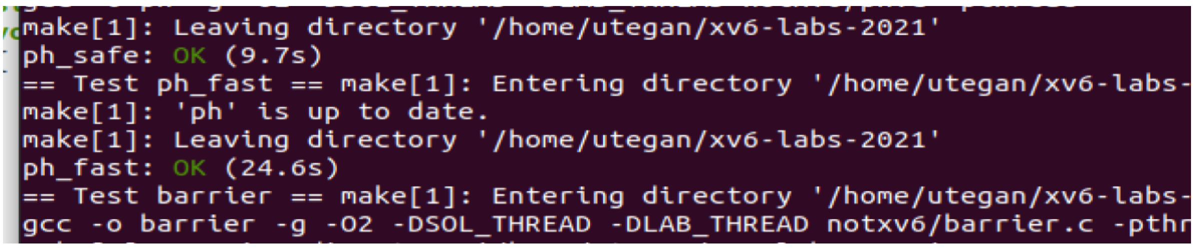


```
} else {
  // the new is new.
  pthread_mutex_lock(lock+i);
  insert(key, value, &table[i], table[i]);
  pthread_mutex_unlock(lock+i);
}
```

对应的实验结果：



```
utegan@ubuntu:~/xv6-labs-2021$ make ph
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
utegan@ubuntu:~/xv6-labs-2021$ ./ph 2
100000 puts, 3.247 seconds, 30796 puts/second
0: 0 keys missing
1: 0 keys missing
200000 gets, 6.476 seconds, 30885 gets/second
utegan@ubuntu:~/xv6-labs-2021$ ./ph 4
100000 puts, 3.985 seconds, 25095 puts/second
1: 0 keys missing
2: 0 keys missing
3: 0 keys missing
0: 0 keys missing
400000 gets, 20.798 seconds, 19233 gets/second
utegan@ubuntu:~/xv6-labs-2021$ make grade
make clean
```



```
make[1]: Leaving directory '/home/utegan/xv6-labs-2021'
ph_safe: OK (9.7s)
== Test ph_fast == make[1]: Entering directory '/home/utegan/xv6-labs-
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/utegan/xv6-labs-2021'
ph_fast: OK (24.6s)
== Test barrier == make[1]: Entering directory '/home/utegan/xv6-labs-
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthr
```

2. 直接对所有table的smallest critical section加锁。

因为无需对每个table分别加锁，所有的table都加同样的锁,只需定义并初始化一个lock。

```
  // the new is new.
  pthread_mutex_lock(lock);
  insert(key, value, &table[i], table[i]);
  pthread_mutex_unlock(lock);
}
```

结果：

```
make[1]: Leaving directory '/home/utegan/xv6-labs-2021'
ph_safe: OK (13.2s)
== Test ph_fast == make[1]: Entering directory '/home/utegan/xv6-labs-2021'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/utegan/xv6-labs-2021'
ph_fast: OK (26.7s)
== Test barrier == make[1]: Entering directory '/home/utegan/xv6-labs-2021'
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: Leaving directory '/home/utegan/xv6-labs-2021'
```

3. 直接对put()加锁。

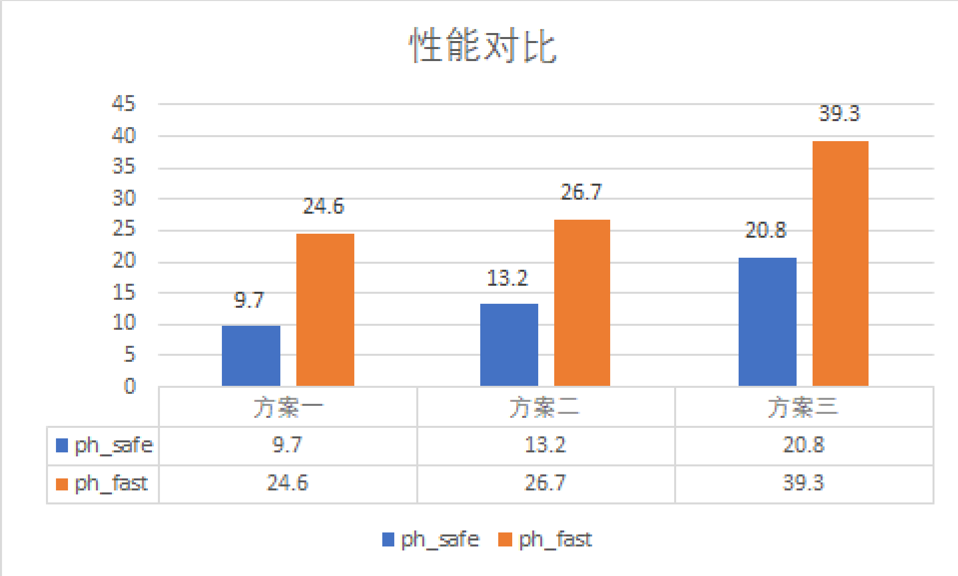
```
for (int i = 0; i < b; i++) {
    pthread_mutex_lock(lock);
    put(keys[b*n + i], n);
    pthread_mutex_unlock(lock);
}
```

结果：

```
ph_safe: OK (20.8s)
== Test ph_fast == make[1]: Entering directory '/home/utegan/xv6-labs-2021'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/utegan/xv6-labs-2021'
ph_fast: FAIL (39.3s)
    Parallel put() speedup is less than 1.25x
== Test barrier == make[1]: Entering directory '/home/utegan/xv6-labs-2021'
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
```

实验结果分析

不同方案性能分析



原因分析

1. 显然方案三不是最小critical section，其`ph_fast`也没有达标。
2. 下面重点分析方案一和方案二：
    1. 两者都是对smallest critical section加锁。
    2. 但是方案一的性能比方案二更高的原因在于方案一**分别对每个table设置了一个特定的锁**，因此当且仅当当不同线程对同一个`table`（即`i`相同时）执行`insert`时，才会互斥。
    3. 方案二是对所有的`table`**不加区分**地设置了**同一个锁**，因此当不同线程执行`insert`的时候都会发生互斥，和`i`无关，频率更高，对性能的影响自然更大。

Barrier

实验内容

概述

1. read `notxv6/barrier.c`.
2. implement barrier() to achieve the desired barrier which is that each thread blocks in `barrier()` until all `nthreads` of them have called `barrier()`.
3. useful primitives

    ```
    pthread_cond_wait(&cond, &mutex);
    pthread_cond_broadcast(&cond);
    ```

4. skip the assert triggers and pass make grade's barrier test.

原文

more details in [Lab: Multithreading (mit.edu)](#).

实验分析

实验代码上传git。

代码报错原因

Each thread executes a loop. In each loop iteration a thread calls `barrier()` and then sleeps for a random number of microseconds. **The assert triggers, because one thread leaves the barrier before the other thread has reached the barrier.**

```
static void *
thread(void *xa)
{
  long n = (long) xa;
  long delay;
  int i;

  for (i = 0; i < 20000; i++) {
    int t = bstate.round;
    assert (i == t);
    barrier();
    usleep(random() % 100);
  }

  return 0;
}
```

Solution

therefore, to avoid the `assert(i==t)` triggering, the `barrier()` is supposed to make each thread with the same `bstate.round` when running `thread()'s block` until all threads get the same `bstate.round`, wake up all threads to go on the next loop.

Algorithm

To realize this goal, in the `barrier()`, increase the `nthread` to acount the number of threads in the current round. When `bstate.nthread` is less than the number of all created threads, call `pthread_mutex_wait()`; When equal, increase the `bstate.round` by one and `bstate.nthread` to 0, call `pthread_mutex_broadcast()`;

Note: Don't forget the lock and unlock.

Implement(code)

```c
static void
barrier()
{
  // YOUR CODE HERE
  //
  pthread_mutex_lock(&bstate.barrier_mutex);
  bstate.nthread++;
  if(bstate.nthread==nthread){
        bstate.nthread=0;
        bstate.round++;
        pthread_cond_broadcast(&bstate.barrier_cond);
  }else{
        pthread_cond_wait(&bstate.barrier_cond,&bstate.barrier_mutex);
  }
  pthread_mutex_unlock(&bstate.barrier_mutex);
  // Block until all threads have called barrier() and
  // then increment bstate.round.
  //

}
```

实验结果

```
== Test barrier == make[1]: Entering directory '/home/utegan/xv6-labs-2021'
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: Leaving directory '/home/utegan/xv6-labs-2021'
barrier: OK (11.8s)
== Test time ==
time: FAIL
    Cannot read time.txt
Score: 24/60
make: *** [Makefile:336: grade] Error 1
utegan@ubuntu:~/xv6-labs-2021$ ./barrier 1
OK; passed
utegan@ubuntu:~/xv6-labs-2021$ ./barrier 2
OK; passed
utegan@ubuntu:~/xv6-labs-2021$ ./barrier 4
OK; passed
utegan@ubuntu:~/xv6-labs-2021$
```