

Lab5 实验报告

一、实验思路、相关代码截图、结果截图

PartA: Uthread: switching between threads

这个实验的主要目的是手写实现线程之间的转换。

首先根据提示，在 struct thread 中补充一个存储寄存器的结构（由于寄存器的数量很多，可以使用一个结构体用以存储）。

```
struct context {
    /* register */
    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;
};
```

然后仿照 kernel/switch.S，完成 notxv6/uthread_switch.S

```
9  thread_switch:
10     /* YOUR CODE HERE */
11     sd ra, 0(a0)
12     sd sp, 8(a0)
13     sd s0, 16(a0)
14     sd s1, 24(a0)
15     sd s2, 32(a0)
16     sd s3, 40(a0)
17     sd s4, 48(a0)
18     sd s5, 56(a0)
19     sd s6, 64(a0)
20     sd s7, 72(a0)
21     sd s8, 80(a0)
22     sd s9, 88(a0)
23     sd s10, 96(a0)
24     sd s11, 104(a0)
25
26     ld ra, 0(a1)
27     ld sp, 8(a1)
28     ld s0, 16(a1)
29     ld s1, 24(a1)
30     ld s2, 32(a1)
31     ld s3, 40(a1)
32     ld s4, 48(a1)
33     ld s5, 56(a1)
34     ld s6, 64(a1)
35     ld s7, 72(a1)
36     ld s8, 80(a1)
37     ld s9, 88(a1)
38     ld s10, 96(a1)
39     ld s11, 104(a1)
40
41     ret /* return to ra */
```

根据提示对相关寄存器进行初始化：

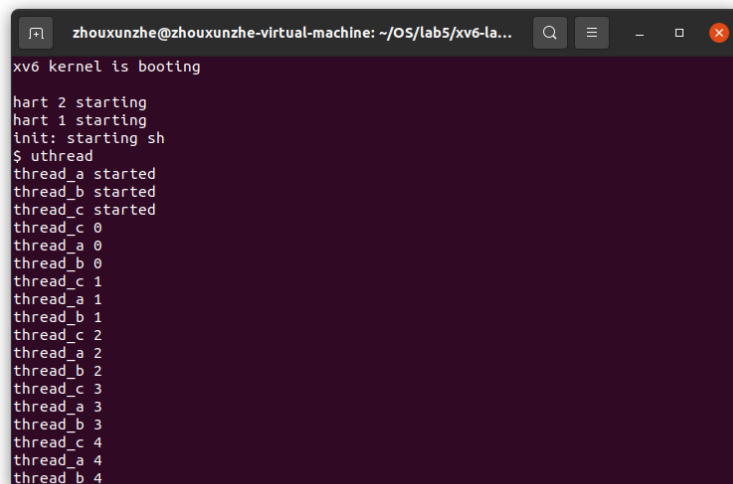
ra 寄存器指向线程要运行的函数，switch 结束后会返回到 ra 处开始运行；
sp 指向线程自己的栈。要注意：压栈是减小栈指针，所以一开始在最高处。

```
91 void
92 thread_create(void (*func)())
93 {
94     struct thread *t;
95
96     for (t = all_thread; t < all_thread + MAX_THREAD; t++) {
97         if (t->state == FREE) break;
98     }
99     t->state = RUNNABLE;
100     // YOUR CODE HERE
101
102     t->context.ra = (uint64)func;
103     t->context.sp = (uint64)&t->stack[STACK_SIZE-1];
104 }
```

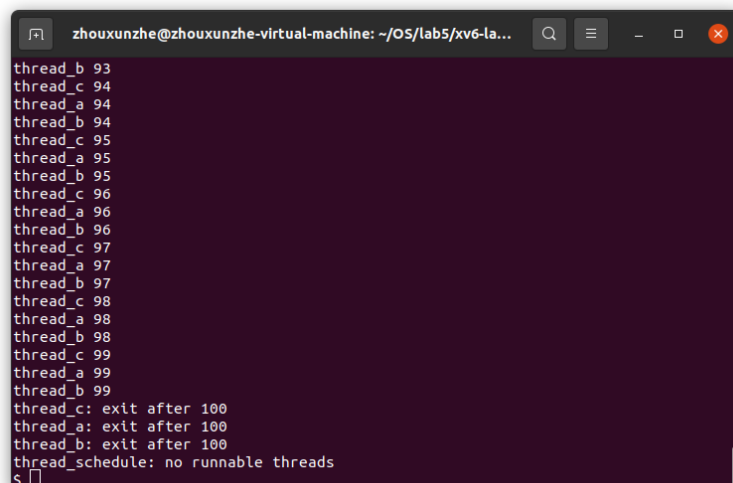
最后一步，在 thread_schedule() 里调用刚才汇编写的 thread_switch：

```
86 thread_switch((uint64)&t->context, (uint64)&next_thread->context);
```

在 qemu 中尝试调用 uthread：



```
zhouxunzhe@zhouxunzhe-virtual-machine: ~/OS/lab5/xv6-la...
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
```



```
zhouxunzhe@zhouxunzhe-virtual-machine: ~/OS/lab5/xv6-la...
thread_b 93
thread_c 94
thread_a 94
thread_b 94
thread_c 95
thread_a 95
thread_b 95
thread_c 96
thread_a 96
thread_b 96
thread_c 97
thread_a 97
thread_b 97
thread_c 98
thread_a 98
thread_b 98
thread_c 99
thread_a 99
thread_b 99
thread_c: exit after 100
thread_a: exit after 100
thread_b: exit after 100
thread_schedule: no runnable threads
$
```

可以看到，程序成功运行，实现创建线程并保存/恢复寄存器以在线程之间切换的任务。

PartB: Using Threads

```
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./ph 1
100000 puts, 5.353 seconds, 18680 puts/second
0: 0 keys missing
100000 gets, 5.415 seconds, 18466 gets/second
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./ph 2
100000 puts, 4.341 seconds, 23034 puts/second
0: 15991 keys missing
1: 15991 keys missing
200000 gets, 7.178 seconds, 27863 gets/second
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$
```

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)

```
41
42 static
43 void put(int key, int value)
44 {
45     int i = key % NBUCKET;
46
47     // is the key already present?
48     struct entry *e = 0;
49     for (e = table[i]; e != 0; e = e->next) {
50         if (e->key == key)
51             break;
52     }
53     if(e){
54         // update the existing key.
55         e->value = value;
56     } else {
57         // the new is new.
58         insert(key, value, &table[i], table[i]);
59     }
60 }
61
```

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。

```
ph_safe: FAIL (8.0s)
got:
16081
expected:
0
== Test ph_fast == make[1]: 进入目录"/home/zhouxunzhe/xv6-labs-2021"
make[1]: "ph"已是最新。
make[1]: 离开目录"/home/zhouxunzhe/xv6-labs-2021"
```

如图为实验测试结果。(ph_safe 未通过)

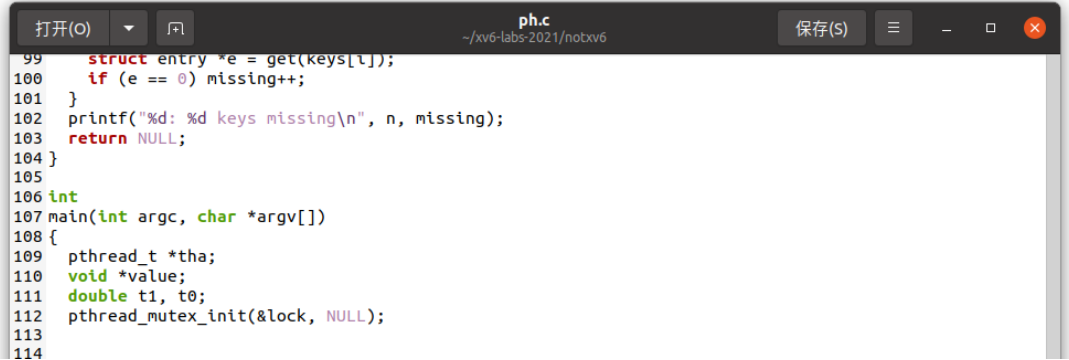
针对以上原理，该实验的主要目的就是给 put 过程加锁，以保证 put 过程的完整性，从而使 hash 表中内容符合预期。

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
```

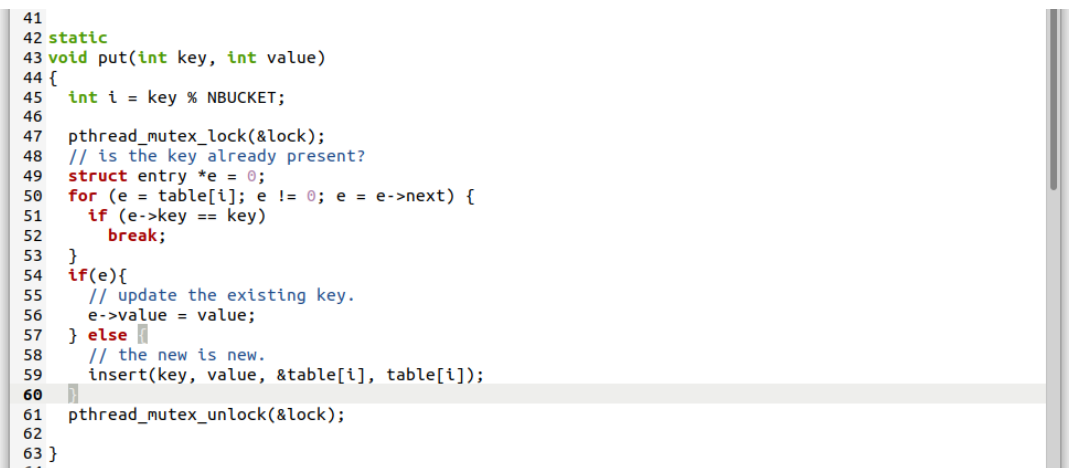
```
37 }
38
39 pthread_mutex_t lock;
40
41
42 static
43 void put(int key, int value)
44 {
45     int i = key % NBUCKET;
```

首先定义一个锁为全局变量，这样所有的函数都可以调用。



```
99 struct entry *e = get(keys[i]);
100 if (e == 0) missing++;
101 }
102 printf("%d: %d keys missing\n", n, missing);
103 return NULL;
104 }
105
106 int
107 main(int argc, char *argv[])
108 {
109     pthread_t *tha;
110     void *value;
111     double t1, t0;
112     pthread_mutex_init(&lock, NULL);
113
114 }
```

然后要调用锁，首先在主函数中将锁初始化。



```
41
42 static
43 void put(int key, int value)
44 {
45     int i = key % NBUCKET;
46
47     pthread_mutex_lock(&lock);
48     // is the key already present?
49     struct entry *e = 0;
50     for (e = table[i]; e != 0; e = e->next) {
51         if (e->key == key)
52             break;
53     }
54     if(e){
55         // update the existing key.
56         e->value = value;
57     } else {
58         // the new is new.
59         insert(key, value, &table[i], table[i]);
60     }
61     pthread_mutex_unlock(&lock);
62
63 }
```

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。

```

zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./ph 1
100000 puts, 5.106 seconds, 19586 puts/second
0: 0 keys missing
100000 gets, 5.073 seconds, 19713 gets/second
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./ph 2
100000 puts, 7.509 seconds, 13318 puts/second
0: 0 keys missing
1: 0 keys missing
200000 gets, 5.456 seconds, 36655 gets/second
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$

```

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。

```

ph_safe: OK (11.2s)
== Test ph_fast == make[1]: 进入目录"/home/zhouxunzhe/xv6-labs-2021"
make[1]: "ph"已是最新。
make[1]: 离开目录"/home/zhouxunzhe/xv6-labs-2021"
ph_fast: FAIL (20.8s)
Parallel put() speedup is less than 1.25x
== Test barrier == make[1]: 进入目录"/home/zhouxunzhe/xv6-labs-2021"
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: 离开目录"/home/zhouxunzhe/xv6-labs-2021"

```

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。

```

38
39 pthread_mutex_t lock[NBUCKET];
40
41 static
42 void put(int key, int value)
43 {
44     int i = key % NBUCKET;
45
46     pthread_mutex_lock(&lock[i]);
47     // is the key already present?
48     struct entry *e = 0;
49     for (e = table[i]; e != 0; e = e->next) {
50         if (e->key == key)
51             break;
52     }
53     if(e){
54         // update the existing key.
55         e->value = value;
56     } else {
57         // the new is new.
58         insert(key, value, &table[i], table[i]);
59     }
60     pthread_mutex_unlock(&lock[i]);
61 }
62 }
63

```

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。

```

105 int
106 main(int argc, char *argv[])
107 {
108     pthread_t *tha;
109     void *value;
110     double t1, t0;
111
112     for(int i=0; i<NBUCKET; i++){
113         pthread_mutex_init(&lock[i], NULL);
114     }
115 }

```

初始化过程即将 NBUCKET 个锁一一初始化即可。

```

zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ make ph
make: "ph"已是最新。
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./ph 1
100000 puts, 4.810 seconds, 20789 puts/second
0: 0 keys missing
100000 gets, 5.070 seconds, 19725 gets/second
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./ph 2
100000 puts, 3.732 seconds, 26798 puts/second
1: 0 keys missing
0: 0 keys missing
200000 gets, 5.166 seconds, 38712 gets/second
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$

```

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。

```

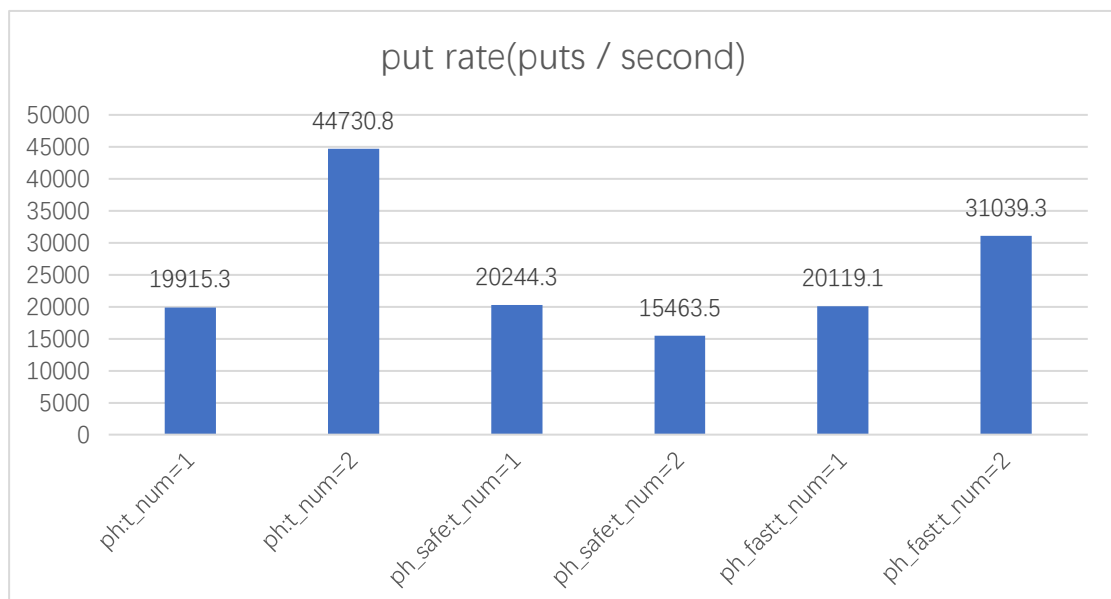
ph_safe: OK (8.3s)
== Test ph_fast == make[1]: 进入目录"/home/zhouxunzhe/xv6-labs-2021"
make[1]: "ph"已是最新。
make[1]: 离开目录"/home/zhouxunzhe/xv6-labs-2021"
ph_fast: OK (17.3s)
== Test barrier == make[1]: 进入目录"/home/zhouxunzhe/xv6-labs-2021"
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: 离开目录"/home/zhouxunzhe/xv6-labs-2021"

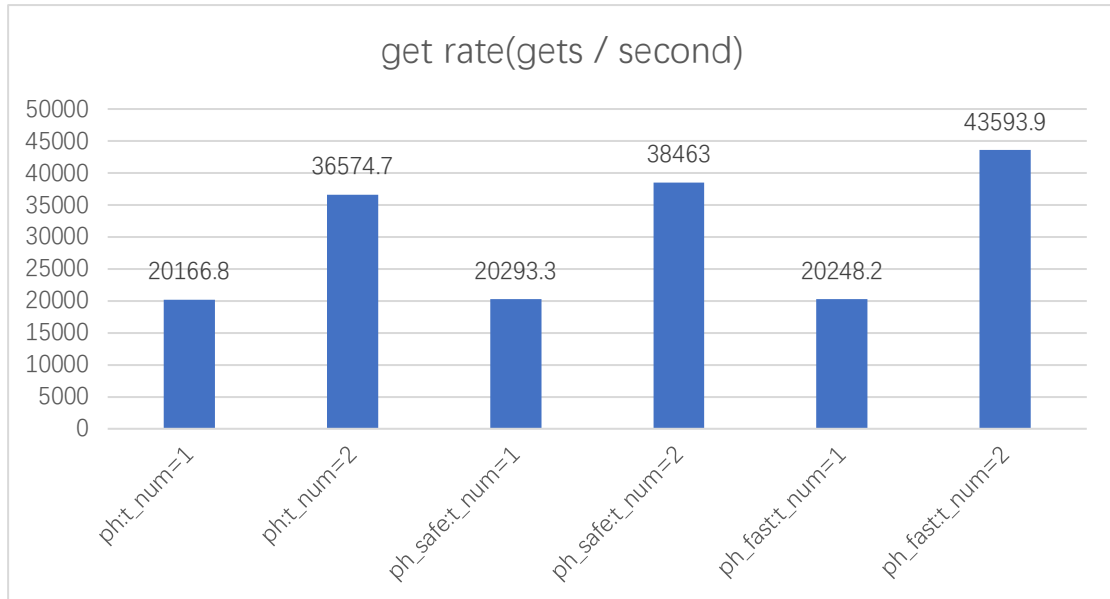
```

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。

PartC: Barrier

首先执行 make barrier 和 ./barrier 2 指令，由于 barrier 函数不完整，所以断言 assert (i == t);会报错。仔细研究，发现这个断言目的是为了判断此时的 round 是否正确。

Barrier 函数的目的就是记录多个线程的多个循环。在命令行的参数的含义是需要运行的线程数量。每当一个线程调用 barrier () 函数，如果在本次循环有线程还未调用，则会进入 wait 状态等待所有线程调用，当所有线程都调用 barrier 以后，则会执行 broadcast 操作唤醒所有的线程。

```

35
36 static void
37 barrier()
38 {
39     // YOUR CODE HERE
40     //
41     // Block until all threads have called barrier() and
42     // then increment bstate.round.
43     //
44     bstate.nthread++;
45     if(bstate.nthread==nthread){ // all threads have reached the barrier
46         bstate.nthread=0;
47         bstate.round++;
48         pthread_cond_broadcast(&bstate.barrier_cond); // wake up every thread sleeping on cond
49     }
50     else
51         pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex); // go to sleep on cond, releasing lock mutex, acquiring upon wake up
52 }
53
54

```

具体实现后运行，发现运行结果会出现死锁的情况。当测试更改循环次数后，如果将次数降为 2, 20, 100 次后能正常运行并输出结果。但是当循环次数为 200 次及以上（只测试 2×10^n 次和 100 次），则会出现死循环，没有运行结果，会一直卡在进程中。

```
zhouxunzhe@zhouxunzhe-virtual-machine: ~/xv6-labs-2021
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ make barrier
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./barrier 2
n_round: 2; OK; passed
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ make barrier
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./barrier 2
n_round: 20; OK; passed
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ make barrier
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./barrier 2
^Z
[1]+ 已停止                  ./barrier 2
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ make barrier
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./barrier 2
n_round: 100; OK; passed
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$
```

如图所示，当使用两个进程时，测试 200 次时会出现死循环，没有运行结果。

仔细研究，发现是可能循环次数增加以后，可能出现一个线程对 `nthread` 自增的情况，另一个线程设置 `nthread=0`，可能导致本应全部进程唤醒，但是最后一个进程却执行了等待操作，最后没有进程可以执行的情况，导致程序不能继续执行，所有进程都在等待被唤醒，从而没有输出结果。

解决办法就是加入锁操作使得 `barrier` 执行的过程中不会出现其他进程干扰的情况。

```
35
36 static void
37 barrier()
38 {
39     // YOUR CODE HERE
40     //
41     // Block until all threads have called barrier() and
42     // then increment bstate.round.
43     //
44     pthread_mutex_lock(&bstate.barrier_mutex); // avoid one thread races
45     bstate.nthread++;
46     if(bstate.nthread==nthread){ // all threads have reached the barrier
47         bstate.nthread=0;
48         bstate.round++;
49         pthread_cond_broadcast(&bstate.barrier_cond); // wake up every thr
50     }
51     else
52         pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex); //
53     pthread_mutex_unlock(&bstate.barrier_mutex);
54
55 }
56
```

加入了锁操作以后，`barrier` 可以正常执行。


```
zhouxunzhe@zhouxunzhe-virtual-machine: ~/xv6-labs-2021
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./barrier 2
n_round: 200; OK; passed
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ make barrier
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./barrier 2
n_round: 2000; OK; passed
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ make barrier
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./barrier 2
n_round: 20000; OK; passed
```

根据提示，测试 1、2 以及多个线程的情况。

```
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ make barrier
make: "barrier"已是最新。
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./barrier 1
OK; passed
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./barrier 2
OK; passed
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./barrier 3
OK; passed
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./barrier 4
OK; passed
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$
```

由于加了锁，各个线程之间互不干扰，所以结果也是符合预期的。

```
zhouxunzhe@zhouxunzhe-virtual-machine: ~/xv6-labs-2021
$ make qemu-gdb
uthread: FAIL (2.8s)
  Output does not match expected output
  QEMU output saved to xv6.out.uthread
== Test answers-thread.txt == answers-thread.txt: FAIL
  Cannot read answers-thread.txt
== Test ph_safe == make[1]: 进入目录"/home/zhouxunzhe/xv6-labs-2021"
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: 离开目录"/home/zhouxunzhe/xv6-labs-2021"
ph_safe: OK (9.0s)
== Test ph_fast == make[1]: 进入目录"/home/zhouxunzhe/xv6-labs-2021"
make[1]: "ph"已是最新。
make[1]: 离开目录"/home/zhouxunzhe/xv6-labs-2021"
ph_fast: OK (20.8s)
== Test barrier == make[1]: 进入目录"/home/zhouxunzhe/xv6-labs-2021"
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: 离开目录"/home/zhouxunzhe/xv6-labs-2021"
barrier: OK (12.5s)
== Test time ==
time: FAIL
  Cannot read time.txt
Score: 34/60
make: *** [Makefile:336: grade] 错误 1
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$
```

最后对 barrier 进行测试，结果也是输出 OK。

由于在测试多个线程时，主观感受不同数量的线程，运行时间不尽相同，于是准备测试不同线程之间的运行时间的差异。学习 ph 中测试运行时间的方法，得到线程数为 1、2、20、200 时的运行时间。

```
zhouxunzhe@zhouxunzhe-virtual-machine: ~/xv6-labs-2021
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ make barrier
make: "barrier"已是最新。
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./barrier 1
OK; passed; time: 14.776 seconds
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./barrier 2
OK; passed; time: 11.999 seconds
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./barrier 1
OK; passed; time: 14.825 seconds
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./barrier 20
OK; passed; time: 13.458 seconds
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./barrier 200
OK; passed; time: 60.750 seconds
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$
```

发现随着线程数增大，其运行时间也会增多。其原理就是各个线程都要进行 20000 次循环，其复杂度为 $O(m \cdot n)$ ， m 是线程数， n 是循环次数。但是当线程数增多以后，由于多个线程可以同时调用，则会提高效率，所以当线程数差别不大时，多个线程运行时间会比一个线程少。

最后使用 `make grade` 进行检查，所有测试都通过了。

```
zhouxunzhe@zhouxunzhe-virtual-machine: ~/OS/lab5/xv6-lab...
make[1]: Leaving directory '/home/zhouxunzhe/OS/lab5/xv6-labs-2022'
== Test uthread ==
$ make qemu-gdb
uthread: OK (5.6s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make[1]: Entering directory '/home/zhouxunzhe/OS/lab5/xv6-lab
s-2022'
gcc -o ph -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/ph.c -pthread
make[1]: Leaving directory '/home/zhouxunzhe/OS/lab5/xv6-labs-2022'
ph_safe: OK (10.1s)
== Test ph_fast == make[1]: Entering directory '/home/zhouxunzhe/OS/lab5/xv6-lab
s-2022'
make[1]: 'ph' is up to date.
make[1]: Leaving directory '/home/zhouxunzhe/OS/lab5/xv6-labs-2022'
ph_fast: OK (18.2s)
== Test barrier == make[1]: Entering directory '/home/zhouxunzhe/OS/lab5/xv6-lab
s-2022'
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
make[1]: Leaving directory '/home/zhouxunzhe/OS/lab5/xv6-labs-2022'
barrier: OK (11.9s)
== Test time ==
time: OK
Score: 60/60
zhouxunzhe@zhouxunzhe-virtual-machine:~/OS/lab5/xv6-labs-2022$
```

二、实验中碰到的问题。

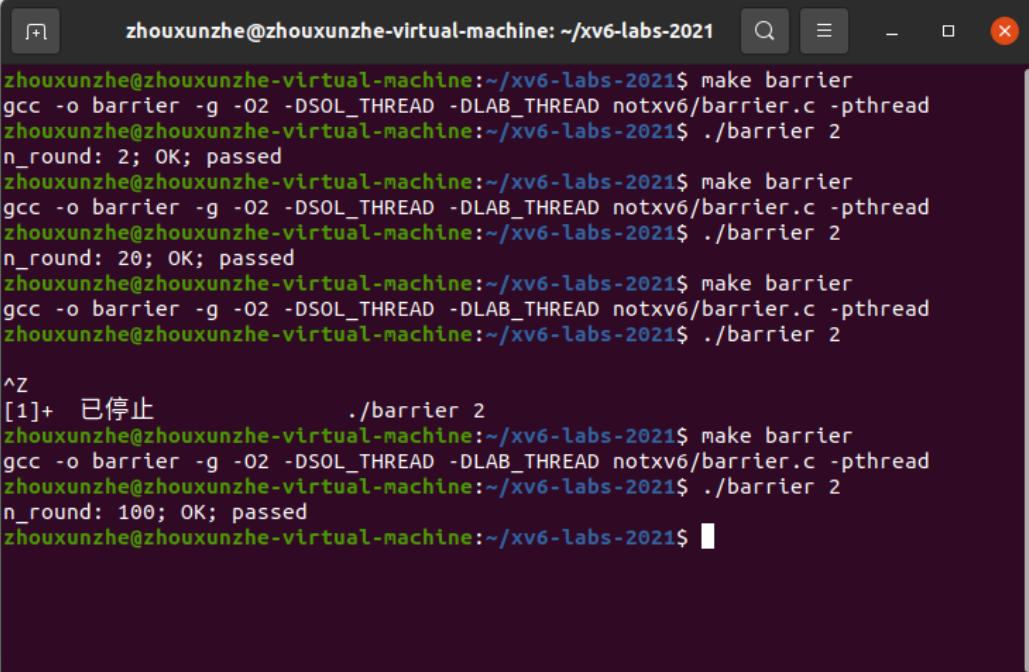
在 barrier 实验中：

首先执行 make barrier 和 ./barrier 2 指令，由于 barrier 函数不完整，所以断言 assert (i == t); 会报错。仔细研究，发现这个断言目的是为了判断此时的 round 是否正确。

Barrier 函数的目的就是记录多个线程的多个循环。在命令行的参数的含义是需要运行的线程数量。每当一个线程调用 barrier () 函数，如果在本次循环有线程还未调用，则会进入 wait 状态等待所有线程调用，当所有线程都调用 barrier 以后，则会执行 broadcast 操作唤醒所有的线程。

```
36 static void
37 barrier()
38 {
39     // YOUR CODE HERE
40     //
41     // Block until all threads have called barrier() and
42     // then increment bstate.round.
43     //
44     bstate.nthread++;
45     if(bstate.nthread==nthread){ // all threads have reached the barrier
46         bstate.nthread=0;
47         bstate.round++;
48         pthread_cond_broadcast(&bstate.barrier_cond); // wake up every thread sleeping on cond
49     }
50     else
51         pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex); // go to sleep on cond, releasing lock mutex, acquiring upon wake up
52 }
53
54
```

具体实现后运行，发现运行结果会出现死锁的情况。当测试更改循环次数后，如果将次数降为 2, 20, 100 次后能正常运行并输出结果。但是当循环次数为 200 次及以上（只测试 2×10^n 次和 100 次），则会出现死循环，没有运行结果，会一直卡在进程中。



```
zhouxunzhe@zhouxunzhe-virtual-machine: ~/xv6-labs-2021
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ make barrier
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./barrier 2
n_round: 2; OK; passed
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ make barrier
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./barrier 20
n_round: 20; OK; passed
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ make barrier
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./barrier 200
^Z
[1]+ 已停止                  ./barrier 2
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ make barrier
gcc -o barrier -g -O2 -DSOL_THREAD -DLAB_THREAD notxv6/barrier.c -pthread
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$ ./barrier 100
n_round: 100; OK; passed
zhouxunzhe@zhouxunzhe-virtual-machine:~/xv6-labs-2021$
```

如图所示，当使用两个进程时，测试 200 次时会出现死循环，没有运行结果。

仔细研究，发现是可能循环次数增加以后，可能出现一个线程对 nthread 自增的情况，另一个线程设置 nthread=0，可能导致本应全部进程唤醒，但是最后一个进程却执行了等待操作，最后没有进程可以执行的情况，导致程序不能继续执行，所有进程都在等待被唤醒，从而没有输出结果。

最后的解决办法就是加入锁操作使得 barrier 执行的过程中不会出现其他进程干扰的情况。

三、实验总结

在本次实验中，我学习到了进程之间切换以及具体调用锁的原理以及代码的具体实现。通过对实验的调试，发现锁在进程切换中是一个不可或缺的事物，同时应该如何避免死锁也是一个需要考虑的问题。

通过本次实验，我更加理解了操作系统中的进程切换，为以后的学习打下坚实基础。