Lab7 Multithreading

1. Uthread: switching between threads (moderate)

1）实验目的

> In this exercise you will design the context switch mechanism for a user-level threading system, and then implement it. To get you started, your xv6 has two files `user/uthread.c` and `user/uthread_switch.S`, and a rule in the Makefile to build a uthread program. uthread.c contains most of a user-level threading package, and code for three simple test threads. The threading package is missing some of the code to create a thread and to switch between threads.
> Your job is to come up with a plan to create threads and save/restore registers to switch between threads, and implement that plan. When you're done, make grade should say that your solution passes the uthread test.
> 在本练习中，您将为用户级线程系统设计上下文切换机制，然后实现它。首先，您的xv6有两个文件 `user/uthread.c` 和 `user/uthread_switch.S` ，以及Makefile中用于构建uthread程序的规则。c包含大部分用户级线程包，以及三个简单测试线程的代码。线程程序包缺少一些用于创建线程和在线程之间切换的代码。
> 您的工作是制定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。

2）实验步骤

编写代码

参考 `kernel/proc.h` 文件中的 `struct context` 进程上下文结构体，在 `user/uthread.h` 文件较为靠前的位置中声明线程上下文 `struct thread_context` 结构体

```
// 线程上下文lab7
struct thread_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;
};
```

在 `struct thread` 线程结构体中添加数据成员记录上下文

```
struct thread {
  char       stack[STACK_SIZE]; /* the thread's stack */
  int        state;             /* FREE, RUNNING, RUNNABLE */
  // 上下文lab7
  struct thread_context thrdctx;
};
```

在 `thread_create` 线程创建函数初始化上下文数据成员
该函数遍历所有线程，将未初始化的线程进行初始化

```
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
  // 初始化上下数据成员 lab7
  t->thrdctx.ra = (uint64)func;
  t->thrdctx.sp = (uint64)t->stack + STACK_SIZE;
}
```

修改 thread_schedule() 函数，在其中调用 thread_switch() 线程切换函数

```
void
thread_schedule(void)
{
  ...
  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->thrdctx, (uint64)&current_thread->thrdctx);
  } else
    next_thread = 0;
}
```

参照 kernel/swtch.S 中的 swtch() 函数，在 user/uthread_switch.S 文件中编写线程的上下文切换函数 thread_switch

```
        .text
        /*
         * save the old thread's registers,
         * restore the new thread's registers.
         */
        .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)
        ret    /* return to ra */
```

测试程序

在存放 `Makefile` 文件的目录下执行如下命令

```
$ make qemu
```

启动xv6系统 QEMU模拟器
键入指令 `uthread` 进行测试

```
$ 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
thread_a 7
thread_b 7
thread_c 8
thread_a 8
thread_b 8
thread_c 9
thread_a 9
thread_b 9
thread_c 10
thread_a 10
thread_b 10
thread_c 11
thread_a 11
thread_b 11
thread_c 12
thread_a 12
thread_b 12
thread_c 13
thread_a 13
thread_b 13
thread_c 14
thread_a 14
thread_b 14
thread_c 15
```

```
thread_a 15
thread_b 15
thread_c 16
thread_a 16
thread_b 16
thread_c 17
thread_a 17
thread_b 17
thread_c 18
thread_a 18
thread_b 18
thread_c 19
thread_a 19
thread_b 19
thread_c 20
thread_a 20
thread_b 20
thread_c 21
thread_a 21
thread_b 21
thread_c 22
thread_a 22
thread_b 22
thread_c 23
thread_a 23
thread_b 23
thread_c 24
thread_a 24
thread_b 24
thread_c 25
thread_a 25
thread_b 25
thread_c 26
thread_a 26
thread_b 26
thread_c 27
thread_a 27
thread_b 27
thread_c 28
thread_a 28
thread_b 28
thread_c 29
thread_a 29
thread_b 29
thread_c 30
thread_a 30
thread_b 30
thread_c 31
thread_a 31
thread_b 31
thread_c 32
```

```
thread_a 32
thread_b 32
thread_c 33
thread_a 33
thread_b 33
thread_c 34
thread_a 34
thread_b 34
thread_c 35
thread_a 35
thread_b 35
thread_c 36
thread_a 36
thread_b 36
thread_c 37
thread_a 37
thread_b 37
thread_c 38
thread_a 38
thread_b 38
thread_c 39
thread_a 39
thread_b 39
thread_c 40
thread_a 40
thread_b 40
thread_c 41
thread_a 41
thread_b 41
thread_c 42
thread_a 42
thread_b 42
thread_c 43
thread_a 43
thread_b 43
thread_c 44
thread_a 44
thread_b 44
thread_c 45
thread_a 45
thread_b 45
thread_c 46
thread_a 46
thread_b 46
thread_c 47
thread_a 47
thread_b 47
thread_c 48
thread_a 48
thread_b 48
thread_c 49
```

```
thread_a 49
thread_b 49
thread_c 50
thread_a 50
thread_b 50
thread_c 51
thread_a 51
thread_b 51
thread_c 52
thread_a 52
thread_b 52
thread_c 53
thread_a 53
thread_b 53
thread_c 54
thread_a 54
thread_b 54
thread_c 55
thread_a 55
thread_b 55
thread_c 56
thread_a 56
thread_b 56
thread_c 57
thread_a 57
thread_b 57
thread_c 58
thread_a 58
thread_b 58
thread_c 59
thread_a 59
thread_b 59
thread_c 60
thread_a 60
thread_b 60
thread_c 61
thread_a 61
thread_b 61
thread_c 62
thread_a 62
thread_b 62
thread_c 63
thread_a 63
thread_b 63
thread_c 64
thread_a 64
thread_b 64
thread_c 65
thread_a 65
thread_b 65
thread_c 66
```

```
thread_a 66
thread_b 66
thread_c 67
thread_a 67
thread_b 67
thread_c 68
thread_a 68
thread_b 68
thread_c 69
thread_a 69
thread_b 69
thread_c 70
thread_a 70
thread_b 70
thread_c 71
thread_a 71
thread_b 71
thread_c 72
thread_a 72
thread_b 72
thread_c 73
thread_a 73
thread_b 73
thread_c 74
thread_a 74
thread_b 74
thread_c 75
thread_a 75
thread_b 75
thread_c 76
thread_a 76
thread_b 76
thread_c 77
thread_a 77
thread_b 77
thread_c 78
thread_a 78
thread_b 78
thread_c 79
thread_a 79
thread_b 79
thread_c 80
thread_a 80
thread_b 80
thread_c 81
thread_a 81
thread_b 81
thread_c 82
thread_a 82
thread_b 82
thread_c 83
```

```
thread_a 83
thread_b 83
thread_c 84
thread_a 84
thread_b 84
thread_c 85
thread_a 85
thread_b 85
thread_c 86
thread_a 86
thread_b 86
thread_c 87
thread_a 87
thread_b 87
thread_c 88
thread_a 88
thread_b 88
thread_c 89
thread_a 89
thread_b 89
thread_c 90
thread_a 90
thread_b 90
thread_c 91
thread_a 91
thread_b 91
thread_c 92
thread_a 92
thread_b 92
thread_c 93
thread_a 93
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
```

结果符合预期

键入 Ctrl+a ， 松开，然后键入 x ， 退出xv6系统

退出xv6进行单元测试

```
root@LAPTOP-UER420HO:~/xv6-labs-2020# ./grade-lab-thread uthread
make: 'kernel/kernel' is up to date.
== Test uthread == uthread: OK (1.6s)
```

3）实验中遇到的问题和解决方法

```
user/uthread.c:86:19: error: passing argument 1 of ‘thread_switch’ makes integer from pointer wi
   86 |     thread_switch(&t->thrdctx, &current_thread->thrdctx);
      |                   ^~~~~~~~~~~
      |                   |
      |                   struct thread_context *
user/uthread.c:41:27: note: expected ‘uint64’ {aka ‘long unsigned int’} but argument is of type
   41 | extern void thread_switch(uint64, uint64);
      |                           ^~~~~~
user/uthread.c:86:32: error: passing argument 2 of ‘thread_switch’ makes integer from pointer wi
   86 |     thread_switch(&t->thrdctx, &current_thread->thrdctx);
      |                                ^~~~~~~~~~~~~~~~~~~~~~~~
      |                                |
      |                                struct thread_context *
user/uthread.c:41:35: note: expected ‘uint64’ {aka ‘long unsigned int’} but argument is of type
   41 | extern void thread_switch(uint64, uint64);
      |                                   ^~~~~~
cc1: all warnings being treated as errors
make: *** [<builtin>: user/uthread.o] Error 1
```

需要进行显式的类型转换

```
    thread_switch((uint64)&t->thrdctx, (uint64)&current_thread->thrdctx);
```

4）实验心得

本实验让我了解到了用户线程切换上下文的基本步骤

基本上大部分都可以参照xv6系统内部进程调度的代码进行编写，比较简单。以下两点是我在写代码的时候比较疑惑的地方，这里展开分析

其中的 ra 寄存器的作用之前的实验已经提到过也使用过。它的全称为 Return Address ， 用来保存线程

切换的返回地址。当一个线程被抢占或者暂停执行的时候，当前线程的执行状态需要保存，在切换回该线程的时候，程序才可以通过 `ra` 寄存器中保存的返回地址恢复到之前被恢复的执行点
而对于 `sp` 寄存器，全称为 `Stack Pointer` ，指示了当前线程的栈顶位置。在线程切换之前，操作系统会保存当前线程的上下文信息，包括程序计数器、通用寄存器等，以便稍后恢复该线程的执行状态。其中也包括保存当前线程的栈指针值。操作系统决定切换到另一个线程时，它会加载下一个线程的上下文信息，包括栈指针值。通过将 `sp` 寄存器设置为下一个线程的栈指针值，操作系统可以确保下一个线程从正确的栈位置开始执行。操作系统决定切换到另一个线程时，它会加载下一个线程的上下文信息，包括栈指针值。通过将SP寄存器设置为下一个线程的栈指针值，操作系统可以确保下一个线程从正确的栈位置开始执行

2. Using threads (moderate)

1）实验目的

> Modify your code so that some `put` operations run in parallel while maintaining correctness. You're done when `make grade` says your code passes both the `ph_safe` and `ph_fast` tests. The `ph_fast` test requires that two threads yield at least 1.25 times as many puts/second as one thread.
>
> > 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
> > To avoid this sequence of events, insert lock and unlock statements in `put` and `get` in `notxv6/ph.c` so that the number of keys missing is always 0 with two threads. The relevant pthread calls are:

```
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
```

> Modify your code so that some `put` operations run in parallel while maintaining correctness. You're done when `make grade` says your code passes both the `ph_safe` and `ph_fast` tests. The `ph_fast` test requires that two threads yield at least 1.25 times as many puts/second as one thread.
> You're done when `make grade` says that your code passes the `ph_safe` test, which requires zero missing keys with two threads. It's OK at this point to fail the `ph_fast` test.
> 修改代码，使某些 `put` 操作在保持正确性的同时并行运行。当 `make grade` 表示您的代码通过了 `ph_safe` 和 `ph_fast` 测试时，您就完成了。 `ph_fast` 测试要求两个线程每秒的输出量至少是一个

> 线程的1.25倍。
> 为什么两个线程都丢失了键，而不是一个线程？确定可能导致键丢失的具有2个线程的事件序列。
> 在 `answers-thread.txt` 中提交您的序列和简短解释。
> 首先使用1个线程测试您的代码，然后使用2个线程测试它。 是否正确（即您是否消除了丢失的钥匙）？
> 相较于单线程版本，双线程版本是否可以实现并行加速（即，单位时间内更多的总工作量）？
> 在某些情况下，并发 `put()` 在哈希表中读取或写入的内存中没有重叠， 因此不需要锁来保护彼此。 您可以更改 `ph.c` 来利用这种情况以获得某些 `put()` 的并行加速吗？
> 每个哈希存储桶有锁吗？修改您的代码，以便在保持正确性的同时并行运行某些 `put` 操作

2）实验步骤

未改进测试

构建包含不安全线程的哈希表的 `ph` 程序
执行 `make ph` 命令

```
root@LAPTOP-UER420HO:~/xv6-labs-2020# make ph
gcc -o ph -g -O2 notxv6/ph.c -pthread
```

运行 `ph 1` 程序，即使用单线程运行该哈希表，查看输出
执行 `./ph 1` 命令

```
root@LAPTOP-UER420HO:~/xv6-labs-2020# ./ph 1
100000 puts, 5.660 seconds, 17666 puts/second
0: 0 keys missing
100000 gets, 6.968 seconds, 14351 gets/second
```

没有键丢失

运行 `ph 2` 程序，即使用两个线程运行该哈希表，查看输出
执行 `./ph 2` 命令

```
root@LAPTOP-UER420HO:~/xv6-labs-2020# ./ph 2
100000 puts, 2.988 seconds, 33463 puts/second
1: 16579 keys missing
0: 16579 keys missing
200000 gets, 7.906 seconds, 25297 gets/second
```

时间加快但存在键丢失的问题

针对问题的回答

多个线程同时调用 `put()` 时，有一定几率对同一个桶的数据进行操作，前一个操作还未完成，后一个就将前一个操作的数据覆盖

我们可以设置**素数**个数的桶，在取模的同时降低映射到同一个桶的可能性

我们为每一个桶加锁，虽然保证了同一个时间一个桶的数据仅能被一个线程访问并修改，但可能会降低并发性

编写代码

我们在此利用互斥锁解决线程不安全问题。对于哈希表，当多个线程同时对一个桶操作的时候，有可能造成数据丢失，因此我们给每个桶配置一个互斥锁

在 `notxv6/ph.c` 文件中声明互斥锁数组

```
// 一个桶一个的互斥锁数组 lab7-2
pthread_mutex_t lock[NBUCKET];
```

在 `notxv6/ph.c` 文件中的 `main()` 函数初始化互斥锁

```
int
main(int argc, char *argv[])
{
  ...
  // 初始化互斥锁 lab7
  for (int i = 0; i < NBUCKET; ++i) {
    pthread_mutex_init(&lock[i], NULL);
  }
}
```

在 `notxv6/ph.c` 文件中的 `put()` 函数中的 `insert()` 操作的前后加上互斥锁的相关操作，保证操作的互斥性

```
static
void put(int key, int value)
{
  ...{
    // update the existing key.
    e->value = value;
  } else {
    // 加锁
    pthread_mutex_lock(&lock[i]);
    // the new is new.
    insert(key, value, &table[i], table[i]);
    // 解锁
    pthread_mutex_unlock(&lock[i]);
  }
}
```

测试程序

执行 `make ph` 编译文件
执行 `./ph 2` 进行测试

```
root@LAPTOP-UER420HO:~/xv6-labs-2020# ./ph 2
100000 puts, 2.774 seconds, 36043 puts/second
0: 0 keys missing
1: 0 keys missing
200000 gets, 7.275 seconds, 27492 gets/second
```

执行 `./grade-lab-thread ph_fast` 进行单元测试

```
root@LAPTOP-UER420HO:~/xv6-labs-2020# ./grade-lab-thread ph_fast
make: 'kernel/kernel' is up to date.
== Test ph_fast == make: 'ph' is up to date.
ph_fast: OK (23.8s)
```

执行 `./grade-lab-thread ph_ph_safe` 进行单元测试

```
root@LAPTOP-UER420HO:~/xv6-labs-2020# ./grade-lab-thread ph_safe
make: 'kernel/kernel' is up to date.
== Test ph_safe == gcc -o ph -g -O2 notxv6/ph.c -pthread
ph_safe: OK (10.0s)
```

3）实验中遇到的问题和解决方法

修改代码后执行 `./ph 2` 命令仍然丢失键

应该是没有编译的问题，重新编译就好了

4）实验心得

本次实验相当于是对锁的应用，之前的实验中也有使用过，因此没什么问题，比较基础，使用互斥锁保证数据不会同时被多个线程访问和修改造成丢失的情况

当多个线程同时访问共享资源时，可能会导致数据不一致或者其他问题。为了避免这种情况，可以使用线程互斥锁来实现线程间的互斥访问

线程互斥锁是一种同步机制，用于保护共享资源，确保在任意时刻只有一个线程可以访问该资源

使用线程互斥锁可以有效地避免多个线程同时访问共享资源而引发的竞态条件和数据不一致问题。然而，过度使用互斥锁可能会导致性能下降，因为它会引入线程间的竞争和等待。因此，在设计多线程程序时，需要权衡使用互斥锁的粒度和性能之间的关系

3. Barrier (moderate)

1）实验目的

> Your goal is to achieve the desired barrier behavior. In addition to the lock primitives that you have seen in the `ph` assignment, you will need the following new pthread primitives; look here and here for details.

```
pthread_cond_wait(&cond, &mutex);  // go to sleep on cond, releasing lock mutex, acquiring upon
pthread_cond_broadcast(&cond);     // wake up every thread sleeping on cond
```

> Make sure your solution passes `make grade`'s `barrier` test.
> 在本作业中，您将实现一个屏障 `Barrier`： 应用程序中的一个点，所有参与的线程在此点上必须等待，直到所有其他参与线程也达到该点。您将使用 `pthread` 条件变量，这是一种序列协调技术，类似于xv6的 `sleep` 和 `wakeup` 。

2）实验步骤

未改进测试

编译 `barrier` 程序，它可以使得多个线程执行至同一位置后在继续执行
执行 `make barrier`

```
root@LAPTOP-UER420HO:~/xv6-labs-2020# make barrier
gcc -o barrier -g -O2 notxv6/barrier.c -pthread
```

执行 `./barrier 2` 命令，使两个线程运行

```
root@LAPTOP-UER420HO:~/xv6-labs-2020# ./barrier 2
barrier: notxv6/barrier.c:45: thread: Assertion `i == t' failed.
barrier: notxv6/barrier.c:45: thread: Assertion `i == t' failed.
Aborted (core dumped)
```

发现程序报错

编写代码

完成 `barrier()` 函数的实现，通过记录到达屏障点的线程数量，进行线程的唤醒或等待，从而实现该函数的功能

```
static void
barrier()
{
  // YOUR CODE HERE
  //
  // Block until all threads have called barrier() and
  // then increment bstate.round.
  //
  // 加锁
  pthread_mutex_lock(&bstate.barrier_mutex);
  // 计数增加
  ++bstate.nthread;
  // 到达数量
  if (bstate.nthread == nthread) {
    ++bstate.round;
    // 重置
    bstate.nthread = 0;
    // 广播唤醒
    pthread_cond_broadcast(&bstate.barrier_cond);
  }
  else {
    // 没到达数量继续等待
    pthread_cond_wait(&bstate.barrier_cond, &bstate.barrier_mutex);
  }
  // 解锁
  pthread_mutex_unlock(&bstate.barrier_mutex);
}
```

测试程序

执行 `make barrier` 重新编译

```
root@LAPTOP-UER420HO:~/xv6-labs-2020# make barrier
gcc -o barrier -g -O2 notxv6/barrier.c -pthread
```

依次执行 `./barrier 1` ， `./barrier 2` ， `./barrier 5` 测试

```
root@LAPTOP-UER420HO:~/xv6-labs-2020# ./barrier 1
OK; passed
root@LAPTOP-UER420HO:~/xv6-labs-2020# ./barrier 2
OK; passed
root@LAPTOP-UER420HO:~/xv6-labs-2020# ./barrier 5
OK; passed
```

执行 `./grade-lab-thread barrier` 命令进行单元测试

```
root@LAPTOP-UER420HO:~/xv6-labs-2020# ./grade-lab-thread barrier
make: 'kernel/kernel' is up to date.
== Test barrier == make: 'barrier' is up to date.
barrier: OK (10.4s)
```

执行 `./grade-lab-thread` 命令进行单元测试

```
root@LAPTOP-UER420HO:~/xv6-labs-2020# ./grade-lab-thread
make: 'kernel/kernel' is up to date.
== Test uthread == uthread: OK (1.4s)
== Test answers-thread.txt == answers-thread.txt: OK
== Test ph_safe == make: 'ph' is up to date.
ph_safe: OK (10.1s)
== Test ph_fast == make: 'ph' is up to date.
ph_fast: OK (22.3s)
== Test barrier == make: 'barrier' is up to date.
barrier: OK (10.4s)
== Test time ==
time: OK
Score: 60/60
```

3）实验中遇到的问题和解决方法

在逻辑上没什么问题，主要就是利用所给的函数进行操作，搞清楚函数如何使用就行了

4）实验心得

本实验是对锁的另一个使用，从而实现线程同步功能。让我深刻地理解到，原来同步不只有之前操作系统理论课上提到的信号量帮助实现。

下面介绍实现的两个比较重要的函数

pthread_cond_wait() 函数用于在多线程编程中实现条件变量的等待操作。条件变量是一种线程间同步的机制，它允许一个或多个线程等待某个特定条件的发生

pthread_cond_wait 函数用于将当前线程阻塞，并等待条件变量的信号。当其他线程调用 p thread_cond_signal 或 pthread_cond_broadcast 发送信号时，被阻塞的线程将被唤醒并继续执行

本实验中就是等待到了一定数量的线程到达某点后就执行 pthread_cond_broadcast 发送信号唤醒线程

pthread_cond_wait 函数需要与互斥锁一起使用，以确保线程在等待条件变量时不会出现竞态条件。在调用 pthread_cond_wait 之前，通常需要先获取互斥锁，然后在等待期间释放互斥锁，以允许其他线程修改共享数据。当线程被唤醒后，它会重新获取互斥锁，并检查条件是否满足

pthread_cond_broadcast() 函数用于向所有等待在特定条件变量上的线程发送信号，它会唤醒所有等待在该条件变量上的线程，这些线程之后可以竞争获取锁并继续执行。这个函数广播一个信号给所有等待线程，相当于通知它们某个条件已经满足，可以继续执行了。同时，他也需要配合互斥锁一起使用。一般的使用方式是，在修改共享数据之前，先获取互斥锁，然后检查条件是否满足，如果不满足，则调用 pthread_cond_wait 等待条件满足。当其他线程修改了共享数据，并调用 pthread_cond_broadcast 发送信号后，等待的线程会被唤醒，再次检查条件是否满足，如果满足则继续执行，否则继续等待