第三十章

Concurrency

Condition Variables

Liu yufeng

Fx_yfliu@163.com

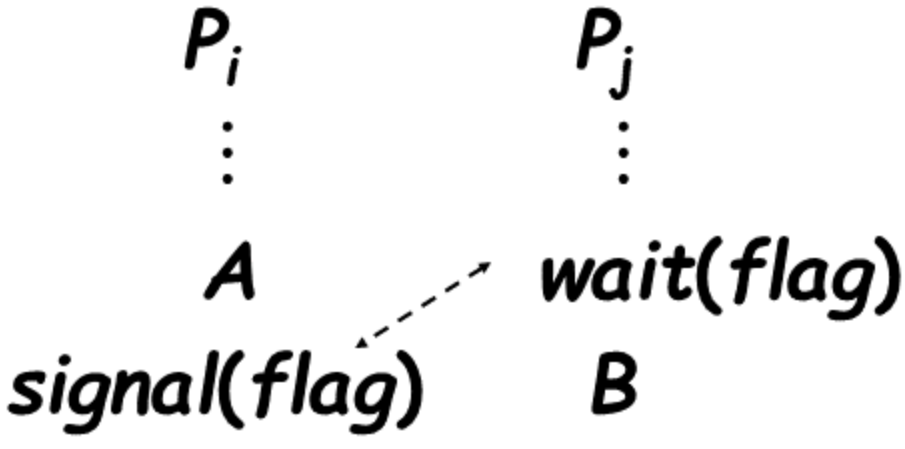
Hunan University

Condition Variables

- Locks are not the only primitives that are needed to build concurrent programs.
- In particular, there are many cases where a thread wishes to check whether a **condition** is true before continuing its execution.
- For example, a parent thread might wish to check whether a child thread has completed before continuing (this is often called a `join()`); how should such a wait be implemented?

Synchronization Tool

- Execute *B* in P_j only after *A* executed in P_i
  - Use semaphore *flag* initialized to 0
    - Code:



```
1   void *child(void *arg) {
2       printf("child\n");
3       // XXX how to indicate we are done?
4       return NULL;
5   }
6
7   int main(int argc, char *argv[]) {
8       printf("parent: begin\n");
9       pthread_t c;
10      Pthread_create(&c, NULL, child, NULL); // create child
11      // XXX how to wait for child?
12      printf("parent: end\n");
13      return 0;
14  }
```

A Parent Waiting For Its Child

What we would like to see here is the following output:

```
parent: begin
child
parent: end
```

How should we implement the wait?

```
1   volatile int done = 0;
2
3   void *child(void *arg) {
4       printf("child\n");
5       done = 1;
6       return NULL;
7   }
8
9   int main(int argc, char *argv[]) {
10      printf("parent: begin\n");
11      pthread_t c;
12      Pthread_create(&c, NULL, child, NULL); // create child
13      while (done == 0)
14          ; // spin
15      printf("parent: end\n");
16      return 0;
17  }
```

Parent Waiting For Child: Spin-based Approach

What is the problem with this solution?

- This solution will generally work, but it is hugely inefficient as the parent spins and wastes CPU time.
- What we would like here instead is some way to put the parent to **sleep** until the **condition** we are waiting for (e.g., the child is done executing) comes true.

关键问题：如何等待一个条件?

多线程程序中，一个线程等待某些条件是很常见的。简单的方案是自旋直到条件满足，这是极其低效的，某些情况下甚至是错误的。那么，线程应该如何等待一个条件?

Definition and Routines

- To wait for a condition to become true, a thread can make use of what is known as a **condition variable**.
- A condition variable is **an explicit queue** that threads can put themselves on when some state of execution is not as desired (by **waiting on the condition**).
- Some other thread, when it changes the state, can then wake one (or more) of waiting threads and allow them to continue (by **signaling on the condition**).

Origin

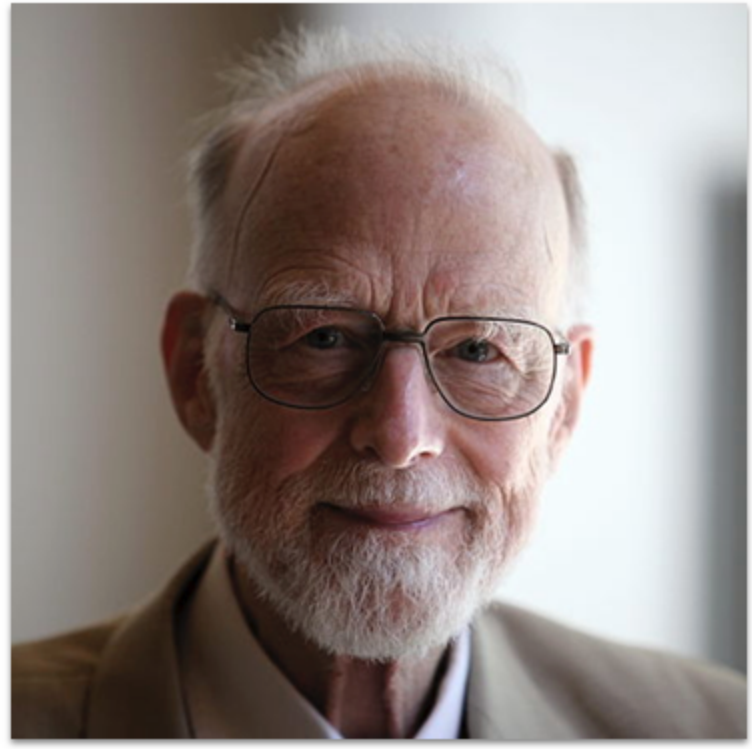
- The idea goes back to **Dijkstra's** use of "**private semaphores**"; a similar idea was later named a "**condition variable**" by **Hoare** in his work on monitors [H74].

[H74] "Monitors: An Operating System Structuring Concept"
C.A.R. Hoare
Communications of the ACM, 17:10, pages 549–557, October 1974
Hoare did a fair amount of theoretical work in concurrency. However, he is still probably most known for his work on Quicksort, the coolest sorting algorithm in the world, at least according to these authors.

- Sir Charles Antony Richard Hoare, commonly known as Tony Hoare or C. A. R. Hoare, is a British computer scientist.

- He developed the sorting algorithm **quicksort** in 1960. He also developed **Hoare logic** for **verifying program correctness**, and the formal language Communicating Sequential Processes (CSP) to specify the interactions of concurrent processes (including the dining philosophers problem).

- ACM **Turing Award** for "**fundamental contributions to the definition and design of programming languages**".



```
pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
pthread_cond_signal(pthread_cond_t *c);
```

- To declare a condition variable, one simply writes something like this: `pthread_cond_t c;`, which declares `c` as a condition variable.
- A condition variable has two operations associated with it: `wait()` and `signal()`.
- The `wait()` call is executed when a thread wishes to put itself to sleep.
- The `signal()` call is executed when a thread has changed something in the program and thus wants to wake a sleeping thread waiting on this condition.

条件变量

- 条件变量：条件变量是用来等待而不是用来上锁的。条件变量用来自动阻塞一个线程，直到某特殊情况发生为止。通常条件变量和互斥锁同时使用。
- 因此，当一个线程互斥的访问某个变量时，它可能发现在其它线程改变状态之前，它什么也做不了。例如一个线程访问队列时，发现队列为空时，他只能等待，直到其它线程将一个节点添加到队列中。这种情况就可以用到条件变量。

Condition Variables

- **Condition variables** are useful when some kind of signaling must take place between threads.

```
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
```

- `pthread_cond_wait`:
  - Put the calling thread to sleep.
  - Wait for some other thread to signal it.
- `pthread_cond_signal`:
  - Unblock at least one of the threads that are blocked on the condition variable

条件变量典型代码

- wait等待条件变量代码：

pthread_mutex_lock(&mutex);

while (条件为假)

pthread_cond_wait(cond, mutex);

修改条件

pthread_mutex_unlock(&mutex);

- signal给条件信号发送信号代码：
- pthread_mutex_lock(&mutex);

  修改条件为真
- pthread_cond_signal(cond);

  pthread_mutex_unlock(&mutex);

- 细节问题pthread_cond_wait(&cond, &mutex);内部完成了三件事：
- 1. 对mutex进行解锁；
- 2，睡眠等待条件，直到有线程向它发起通知；
- 3，重新对mutex进行加锁。
- pthread_cond_signal(&cond);向等待条件的线程发起通知，如果没有任何一个线程处于等待条件的状态，这个通知将被忽略。为什么检测条件用while而不是if？因为pthread_cond_wait会产生信号，pthread_cond_wait可能会被虚假唤醒，因此还需要重新判断。
- pthread_cond_signal一般不会有“惊群现象”产生，他最多只给一个线程发信号。假如有多个线程正在阻塞等待着这个条件变量的话，那么是根据各等待线程优先级的高低确定哪个线程接收到信号开始继续执行。如果各线程优先级相同，则根据等待时间的长短来确定哪个线程获得信号。
- pthread_cond_signal 在多处理器上可能同时唤醒多个线程

Condition Variables (Cont.)

- A thread calling wait routine:

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t init = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (initialized == 0)
    pthread_cond_wait(&init, &lock);
pthread_mutex_unlock(&lock);
```

  - The wait call releases the lock when putting said caller to sleep.
  - Before returning after being woken, the wait call re-acquire the lock.

- A thread calling signal routine:

```
pthread_mutex_lock(&lock);
initialized = 1;
pthread_cond_signal(&init);
pthread_mutex_unlock(&lock);
```

Condition Variables (Cont.)

- The waiting thread **re-checks** the condition in a while loop, instead of a simple if statement.

```
pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t init = PTHREAD_COND_INITIALIZER;

pthread_mutex_lock(&lock);
while (initialized == 0)
    pthread_cond_wait(&init, &lock);
pthread_mutex_unlock(&lock);
```

- Without rechecking, the waiting thread will continue thinking that the condition has changed *even though it has not*.

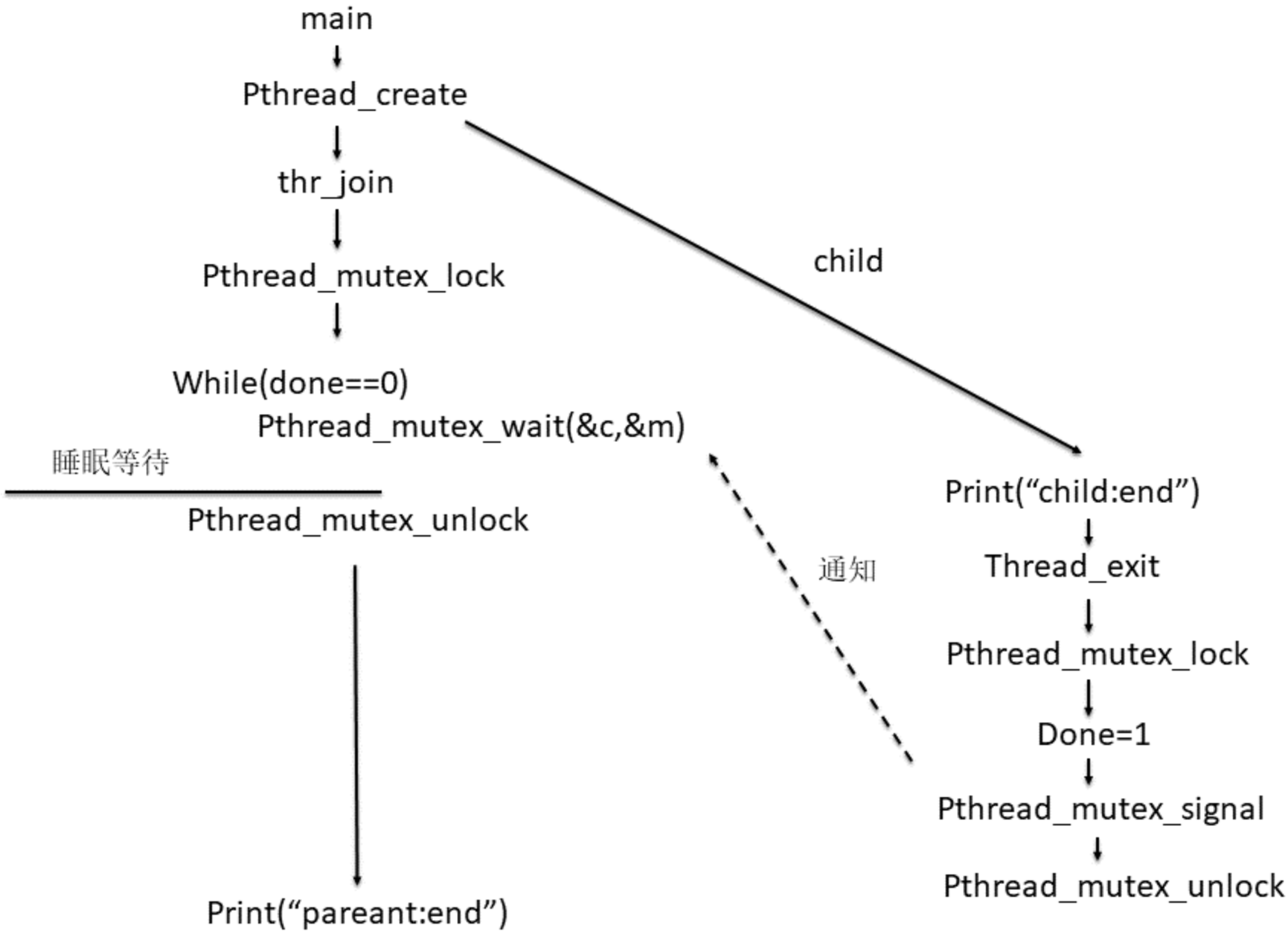
在多核处理器下或者某些竞态条件下，pthread_cond_signal可能会激活多于一个线程（阻塞在条件变量上的线程）。结果就是，当一个线程调用pthread_cond_signal()后，多个调用pthread_cond_wait()或pthread_cond_timedwait()的线程返回。这种效应就称为“虚假唤醒”。因为等待在条件变量上的线程被唤醒有可能不是因为条件满足而是由于虚假唤醒。所以，我们需要对条件变量的状态进行不断检查直到其满足条件

```
1   int done  = 0;
2   pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3   pthread_cond_t c  = PTHREAD_COND_INITIALIZER;
4
5   void thr_exit() {
6       Pthread_mutex_lock(&m);
7       done = 1;
8       Pthread_cond_signal(&c);
9       Pthread_mutex_unlock(&m);
10  }
11
12  void *child(void *arg) {
13      printf("child\n");
14      thr_exit();
15      return NULL;
16  }
17
18  void thr_join() {
19      Pthread_mutex_lock(&m);
20      while (done == 0)
21          Pthread_cond_wait(&c, &m);
22      Pthread_mutex_unlock(&m);
23  }
24
25  int main(int argc, char *argv[]) {
26      printf("parent: begin\n");
27      pthread_t p;
28      Pthread_create(&p, NULL, child, NULL);
29      thr_join();
30      printf("parent: end\n");
31      return 0;
32  }
```

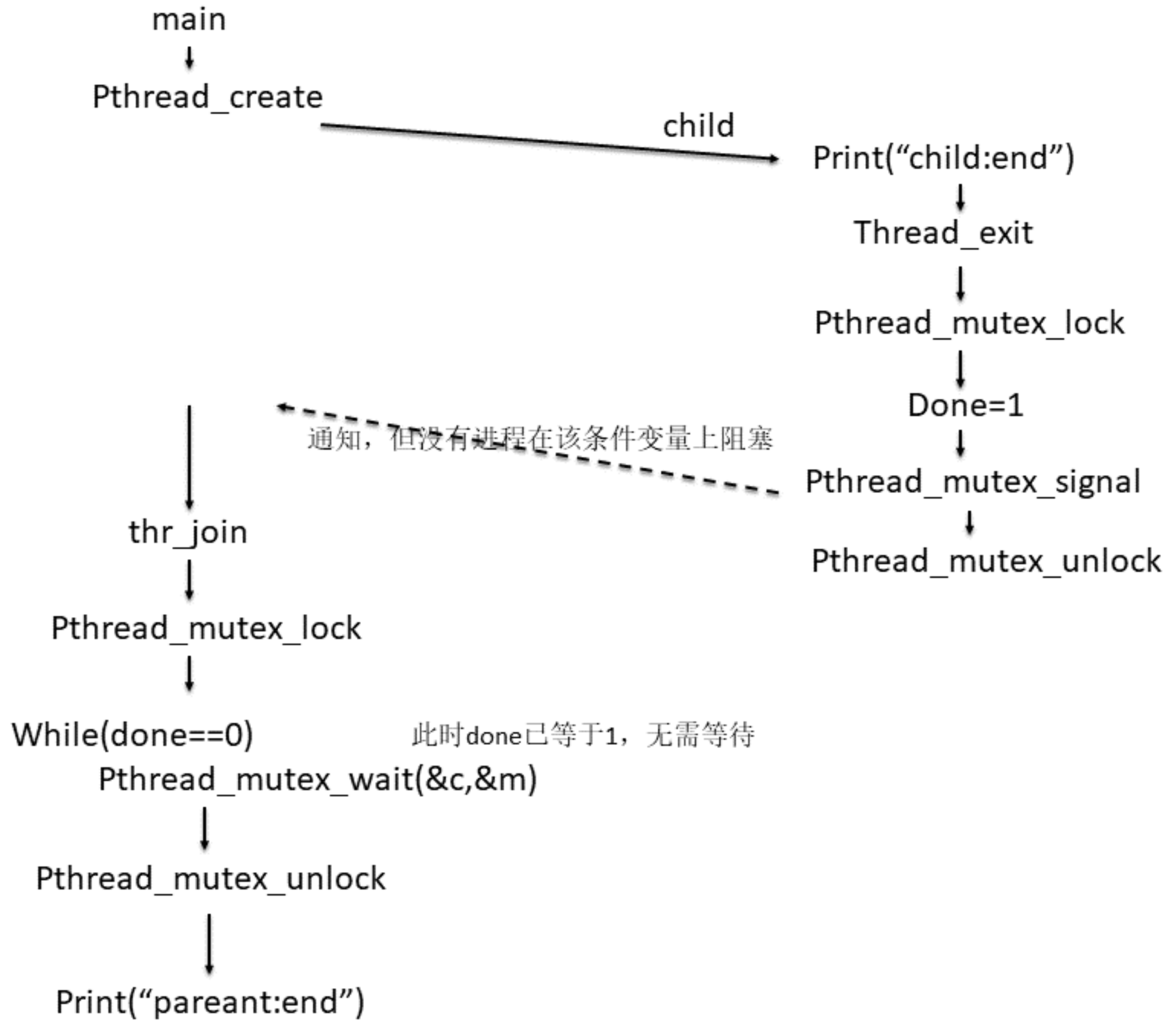
Parent Waiting For Child: Use A Condition Variable

- The **`wait()` takes a mutex as a parameter**; it assumes that this mutex is locked when **`wait()`** is called.
- The responsibility of **`wait()`** is to release the lock and put the calling thread to sleep (**atomically**); when the thread wakes up, it must re-acquire the lock before returning to the caller.
- This complexity stems from the desire to **prevent certain race conditions** from occurring when a thread is trying to sleep.

情况一


main
Pthread_create
thr_join
Pthread_mutex_lock
While(done==0)
Pthread_mutex_wait(&c,&m)
睡眠等待
Pthread_mutex_unlock
Print(“pareant:end”)
child
Print(“child:end”)
Thread_exit
Pthread_mutex_lock
Done=1
Pthread_mutex_signal
Pthread_mutex_unlock
通知

情况二


main
Pthread_create
child
Print(“child:end”)
Thread_exit
Pthread_mutex_lock
Done=1
通知，但没有进程在该条件变量上阻塞
Pthread_mutex_signal
Pthread_mutex_unlock
thr_join
Pthread_mutex_lock
While(done==0)
此时done已等于1，无需等待
Pthread_mutex_wait(&c,&m)
Pthread_mutex_unlock
Print(“pareant:end”)

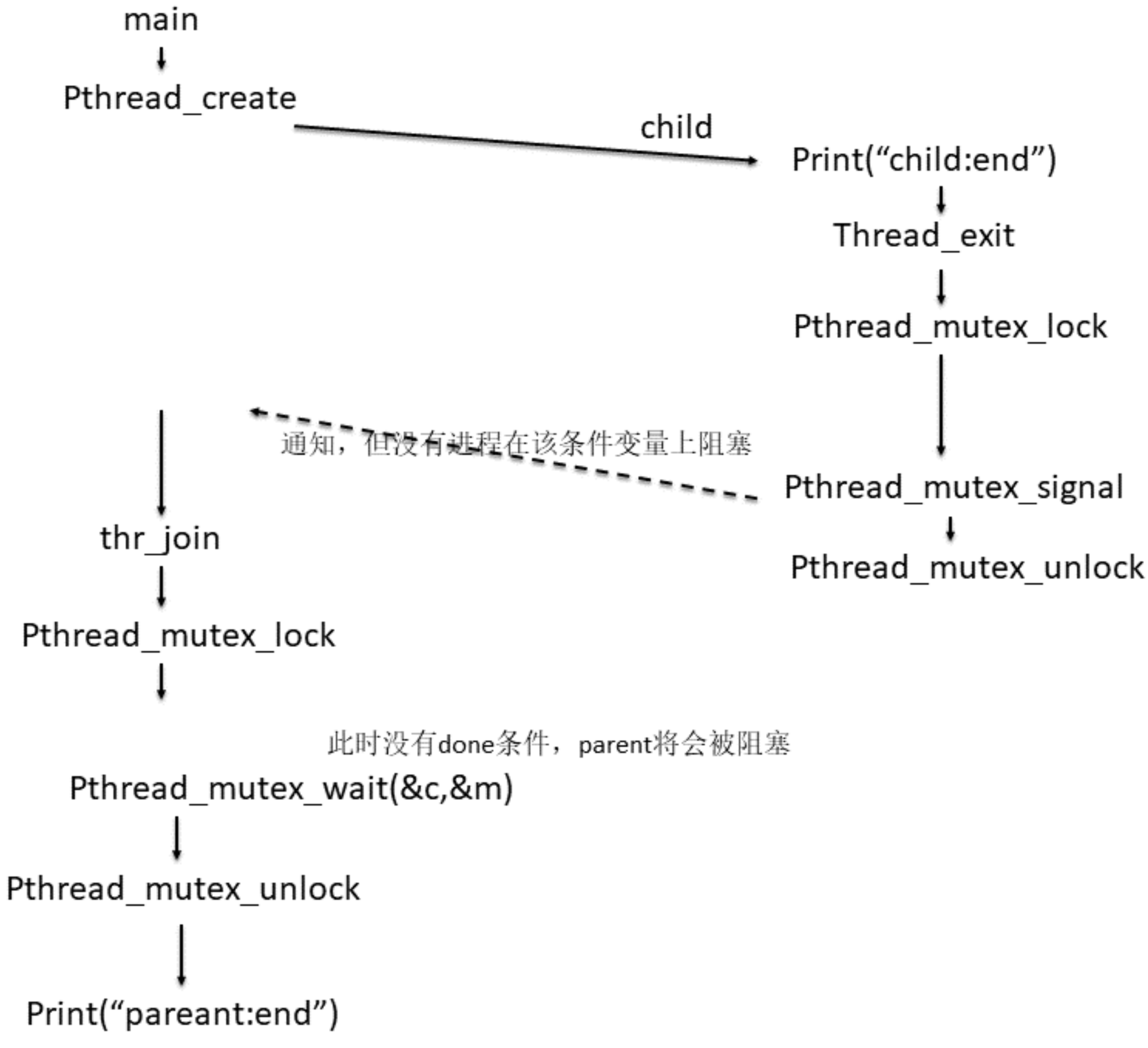
A Few Alternate Implementations

- First, you might be wondering if we need the state variable **done**. What if the code looked like the example below? Would this work?

```
1   void thr_exit() {
2       Pthread_mutex_lock(&m);
3       Pthread_cond_signal(&c);
4       Pthread_mutex_unlock(&m);
5   }
6
7   void thr_join() {
8       Pthread_mutex_lock(&m);
9       Pthread_cond_wait(&c, &m);
10      Pthread_mutex_unlock(&m);
11  }
```

通过这个例子，应该认识到变量 **done 的重要性**，它记录了线程有兴趣知道的值。睡眠、唤醒和锁都离不开它。

如果没有done


main
Pthread_create
child
Print(“child:end”)
Thread_exit
Pthread_mutex_lock
通知，但没有进程在该条件变量上阻塞
Pthread_mutex_signal
Pthread_mutex_unlock
thr_join
Pthread_mutex_lock
此时没有done条件，parent将会被阻塞
Pthread_mutex_wait(&c,&m)
Pthread_mutex_unlock
Print(“pareant:end”)

Another Poor Implementation

- In this example, we imagine that one **does not need to hold a lock** in order to signal and wait. What **problem** could occur here?

```
1   void thr_exit() {
2       done = 1;
3       Pthread_cond_signal(&c);
4   }
5
6   void thr_join() {
7       if (done == 0)
8           Pthread_cond_wait(&c);
9   }
```

如果父进程调用 thr_join()，然后检查完 done 的值为 0，然后试图睡眠。但在调用 **wait** 进入睡眠之前，父进程被中断。子线程修改变量 done 为 1，发出信号，同样没有等待线程。父线程再次运行时，就会长眠不醒。

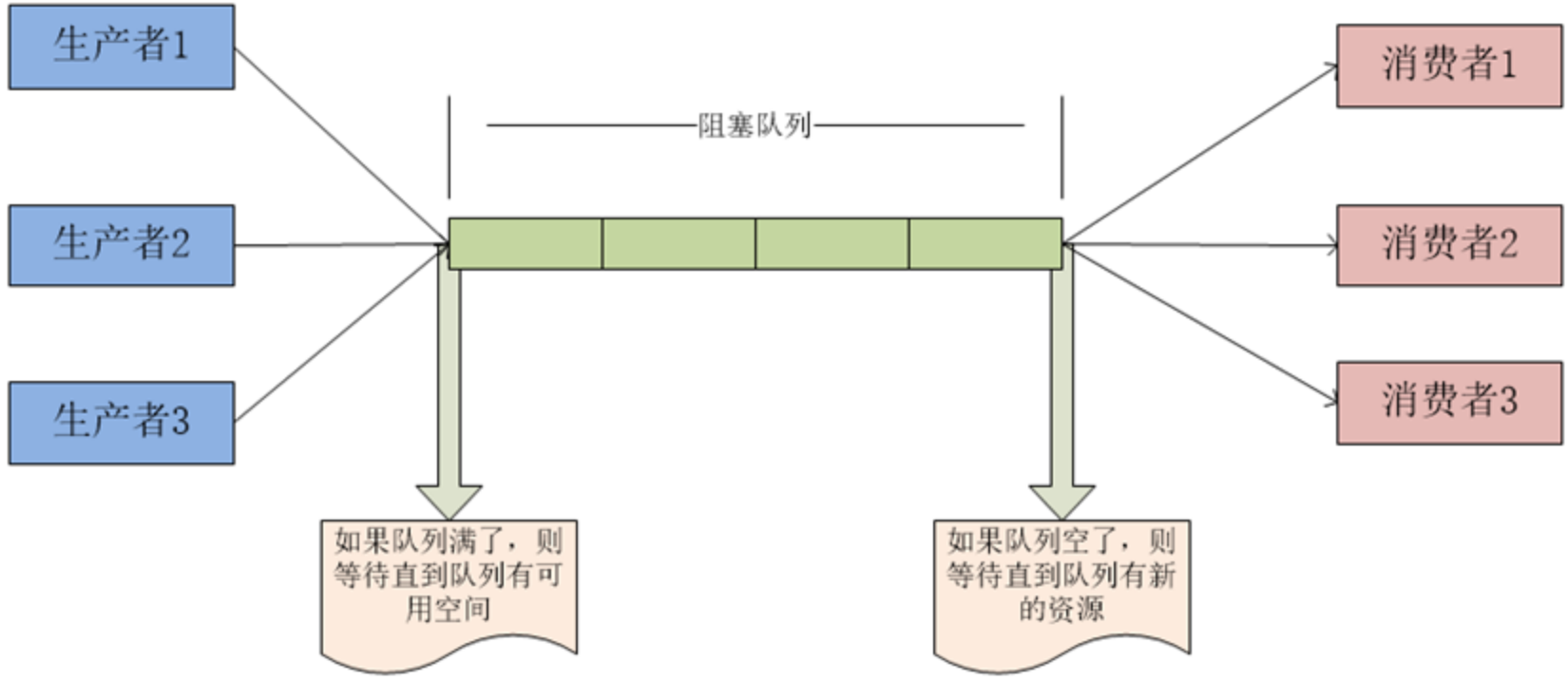
换个角度理解，加锁保证了锁之间代码的原子性

提示：发信号时总是持有锁

尽管并不是所有情况下都严格需要，但有效且简单的做法，还是在使用条件变量发送信号时持有锁。虽然上面的例子是必须加锁的情况，但也有一些情况可以不加锁，而这可能是你应该避免的。因此，为了简单，请在调用 signal 时持有锁（hold the lock when calling signal）。

这个提示的反面，即调用 wait 时持有锁，不只是建议，而是 wait 的语义强制要求的。因为 wait 调用总是假设你调用它时已经持有锁、调用者睡眠之前会释放锁以及返回前重新持有锁。因此，这个提示的一般化形式是正确的：调用 signal 和 wait 时要持有锁（hold the lock when calling signal or wait），你会保持身心健康的。

The Producer/Consumer (Bound Buffer) Problem



The Producer/Consumer (Bound Buffer) Problem

- First posed by Dijkstra.
- Imagine one or more producer threads and one or more consumer threads. Producers generate data items and place them in a buffer; consumers grab said items from the buffer and consume them.
- Example: `grep foo file.txt | wc -l`.

```
1   int buffer;
2   int count = 0; // initially, empty
3
4   void put(int value) {
5       assert(count == 0);
6       count = 1;
7       buffer = value;
8   }
9
10  int get() {
11      assert(count == 1);
12      count = 0;
13      return buffer;
14  }
```

The Put and Get Routines (Version 1)

```
1   void *producer(void *arg) {
2       int i;
3       int loops = (int) arg;
4       for (i = 0; i < loops; i++) {
5           put(i);
6       }
7   }
8
9   void *consumer(void *arg) {
10      int i;
11      while (1) {
12          int tmp = get();
13          printf("%d\n", tmp);
14      }
15  }
```

Producer/Consumer Threads (Version 1)

- Use **a single integer** for simplicity, and the two inner routines to put a value into the shared buffer, and get a value from the buffer.

- The **`put()`** assumes the buffer is empty, and puts a value into the shared buffer and marks it full by setting **`count`** to **1**. The **`get()`** sets the buffer to empty (set **`count`** to **0**) and returning the value.

A Broken Solution

```
1   int buffer;
2   int count = 0; // initially, empty
3
4   void put(int value) {
5       assert(count == 0);
6       count = 1;
7       buffer = value;
8   }
9
10  int get() {
11      assert(count == 1);
12      count = 0;
13      return buffer;
14  }
```

```
1   cond_t  cond;
2   mutex_t mutex;
3
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           Pthread_mutex_lock(&mutex);            // p1
8           if (count == 1)                        // p2
9               Pthread_cond_wait(&cond, &mutex); // p3
10          put(i);                                // p4
11          Pthread_cond_signal(&cond);            // p5
12          Pthread_mutex_unlock(&mutex);          // p6
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          Pthread_mutex_lock(&mutex);            // c1
20          if (count == 0)                        // c2
21              Pthread_cond_wait(&cond, &mutex); // c3
22          int tmp = get();                       // c4
23          Pthread_cond_signal(&cond);            // c5
24          Pthread_mutex_unlock(&mutex);          // c6
25          printf("%d\n", tmp);
26      }
27  }
```

- When a producer wants to fill the buffer, it waits for it to be empty (**p1–p3**). The consumer has the exact same logic, but waits for a different condition: fullness (**c1–c3**).
- With just **a single producer and a single consumer**, the code works.
- However, if we have **more than one** of these threads (two consumers), the solution has **two critical problems**. What are they?

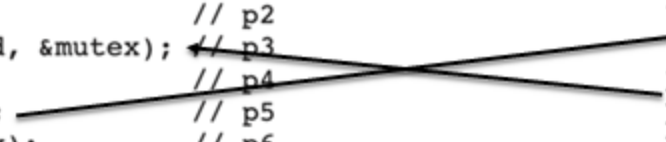
```
int buffer;
int count = 0; // initially, empty

void put(int value) {
    assert(count == 0);
    count = 1;
    buffer = value;
}

int get() {
    assert(count == 1);
    count = 0;
    return buffer;
}
cond_t  cond;
mutex_t mutex;
```

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // p1
        if (count == 1)                        // p2
            Pthread_cond_wait(&cond, &mutex);  // p3
        put(i);                                // p4
        Pthread_cond_signal(&cond);            // p5
        Pthread_mutex_unlock(&mutex);          // p6
    }
}
```

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // c1
        if (count == 0)                        // c2
            Pthread_cond_wait(&cond, &mutex);  // c3
        int tmp = get();                       // c4
        Pthread_cond_signal(&cond);            // c5
        Pthread_mutex_unlock(&mutex);          // c6
        printf("%d\n", tmp);
    }
```



Mutex加锁，实现互斥，Wait实现同步

T_{c1}	State	T_{c2}	State	T_p	State	Count	*Comment*
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep		Ready	p1	Running	0	
	Sleep		Ready	p2	Running	0	
	Sleep		Ready	p4	Running	1	Buffer now full
	Ready		Ready	p5	Running	1	T_{c1} awoken
	Ready		Ready	p6	Running	1	
	Ready		Ready	p1	Running	1	
	Ready		Ready	p2	Running	1	
	Ready		Ready	p3	Sleep	1	Buffer full; sleep
	Ready	c1	Running		Sleep	1	T_{c2} sneaks in ...
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	... and grabs data
	Ready	c5	Running		Ready	0	T_p awoken
	Ready	c6	Running		Ready	0	
c4	Running		Ready		Ready	0	Oh oh! No data

假设有两个消费者 (Tc1 和 Tc2)，一个生产者(Tp)。

Thread Trace: Broken Solution (Version 1)

```
void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // p1
        if (count == 1)                        // p2
            Pthread_cond_wait(&cond, &mutex);  // p3
        put(i);                                // p4
        Pthread_cond_signal(&cond);            // p5
        Pthread_mutex_unlock(&mutex);          // p6
    }
}
```

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // c1
        if (count == 0)                        // c2
            Pthread_cond_wait(&cond, &mutex);  // c3
        int tmp = get();                       // c4
        Pthread_cond_signal(&cond);            // c5
        Pthread_mutex_unlock(&mutex);          // c6
        printf("%d\n", tmp);
    }
```

Tc1被生产者唤醒之后，还没来得及消费数据，也就是在Tc1执行get（）之前，另一个消费者（Tc2）抢先执行，消费了缓冲区中的值（c1,c2,c4,c5,c6，跳过了 c3 的等待，因为缓冲区是满的）。现在假设 Tc1 运行，由于唤醒后不再检查条件（count==0）,所以调用 get()，但缓冲区已无数据消费

- The **problem arises for a simple reason**: after the producer woke $\mathbf{T}_{c1}$, but ***before*** $\mathbf{T}_{c1}$ ever ran, the state of the bounded buffer changed (due to $\mathbf{T}_{c2}$).

- Signaling a thread only wakes them up; it is thus a **hint** that the state of the world has changed, but there is no guarantee that when the woken thread runs, the state will **still** be as desired.

Better, But Still Broken: While, Not If

- Change the **`if`** to a **`while`**. Think about why this works; now consumer $\mathbf{T_{c1}}$ wakes up and (**with the lock held**) immediately re-checks the state of the shared variable (**c2**). If the buffer is empty at that point, the consumer simply goes back to sleep (**c3**). The `if` is also changed to a `while` in the producer (**p2**).

- A simple rule to remember with condition variables is to **always use while loops**.

```
1   cond_t  cond;
2   mutex_t mutex;
3
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           Pthread_mutex_lock(&mutex);            // p1
8           while (count == 1)                     // p2
9               Pthread_cond_wait(&cond, &mutex); // p3
10          put(i);                                // p4
11          Pthread_cond_signal(&cond);            // p5
12          Pthread_mutex_unlock(&mutex);          // p6
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          Pthread_mutex_lock(&mutex);            // c1
20          while (count == 0)                     // c2
21              Pthread_cond_wait(&cond, &mutex); // c3
22          int tmp = get();                       // c4
23          Pthread_cond_signal(&cond);            // c5
24          Pthread_mutex_unlock(&mutex);          // c6
25          printf("%d\n", tmp);
26      }
27  }
```

Producer/Consumer: Single CV and While

```
1  cond_t  cond;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5      int i;
6      for (i = 0; i < loops; i++) {
7          Pthread_mutex_lock(&mutex);            // p1
8          while (count == 1)                     // p2
9              Pthread_cond_wait(&cond, &mutex);  // p3
10         put(i);                                // p4
11         Pthread_cond_signal(&cond);            // p5
12         Pthread_mutex_unlock(&mutex);          // p6
13     }
14 }
15
16 void *consumer(void *arg) {
17     int i;
18     for (i = 0; i < loops; i++) {
19         Pthread_mutex_lock(&mutex);            // c1
20         while (count == 0)                     // c2
21             Pthread_cond_wait(&cond, &mutex);  // c3
22         int tmp = get();                       // c4
23         Pthread_cond_signal(&cond);            // c5
24         Pthread_mutex_unlock(&mutex);          // c6
25         printf("%d\n", tmp);
26     }
27 }
```

T_{c1}	State	T_{c2}	State	T_p	State	Count	*Comment*
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Running		Ready	0	
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Running	0	
	Sleep		Sleep	p2	Running	0	
	Sleep		Sleep	p4	Running	1	Buffer now full
	Ready		Sleep	p5	Running	1	T_{c1} awoken
	Ready		Sleep	p6	Running	1	
	Ready		Sleep	p1	Running	1	
	Ready		Sleep	p2	Running	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Running		Sleep		Sleep	1	Recheck condition
c4	Running		Sleep		Sleep	0	T_{c1} grabs data
c5	Running		Ready		Sleep	0	Oops! Woke T_{c2}
c6	Running		Ready		Sleep	0	
c1	Running		Ready		Sleep	0	
c2	Running		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Running		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	Everyone asleep...

Thread Trace: Broken Solution (Version 2)

- However, this code **still has a bug**. It has something to do with the fact that there is **only one condition variable**.

可能出现3个线程都在睡眠

```
1   cond_t  cond;
2   mutex_t mutex;
3
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           Pthread_mutex_lock(&mutex);            // p1
8           while (count == 1)                     // p2
9               Pthread_cond_wait(&cond, &mutex);  // p3
10          put(i);                                // p4
11          Pthread_cond_signal(&cond);            // p5
12          Pthread_mutex_unlock(&mutex);          // p6
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          Pthread_mutex_lock(&mutex);            // c1
20          while (count == 0)                     // c2
21              Pthread_cond_wait(&cond, &mutex);  // c3
22          int tmp = get();                       // c4
23          Pthread_cond_signal(&cond);            // c5
24          Pthread_mutex_unlock(&mutex);          // c6
25          printf("%d\n", tmp);
26      }
27  }
```

假设两个消费者 (Tc1 和 Tc2) 先运行，都睡眠了 (c3)。生产者开始运行，在缓冲区放入一个值，唤醒了一个消费者 (假定是 Tc1)，并开始睡眠。现在是一个消费者马上要运行 (Tc1)，两个线程 (Tc2 和 Tp) 都等待在同一个条件变量上。问题马上就要出现了！

消费者 Tc1 醒过来并从 wait( ) 调用返回 (c3)，重新检查条件 (c2)，发现缓冲区是满的， 消费了这个值 (c4)。这个消费者然后在该条件上发信号 (c5)，唤醒一个在睡眠的线程。但是，应该唤醒哪个线程呢?

因为消费者已经清空了缓冲区，很显然，应该唤醒生产者。但是，如果**它唤醒了 Tc2** (这绝对是可能的，取决于等待队列是如何管理的)，问题就出现了。具体来说，消费者 Tc2 会醒过来，发现队列为空 (c2)，又继续回去睡眠 (c3)。生产者 Tp 刚才在缓冲区中放了一个值，现在在睡眠。另一个消费者线程 Tc1 也回去睡眠了。3 个线程都在睡眠，显然是一个缺陷。

T_{c1}	State	T_{c2}	State	T_p	State	Count	*Comment*
c1	Running		Ready		Ready	0	
c2	Running		Ready		Ready	0	
c3	Sleep		Ready		Ready	0	Nothing to get
	Sleep	c1	Running		Ready	0	
	Sleep	c2	Running		Ready	0	
	Sleep	c3	Sleep		Ready	0	Nothing to get
	Sleep		Sleep	p1	Running	0	
	Sleep		Sleep	p2	Running	0	
	Sleep		Sleep	p4	Running	1	Buffer now full
	Ready		Sleep	p5	Running	1	T_{c1} awoken
	Ready		Sleep	p6	Running	1	
	Ready		Sleep	p1	Running	1	
	Ready		Sleep	p2	Running	1	
	Ready		Sleep	p3	Sleep	1	Must sleep (full)
c2	Running		Sleep		Sleep	1	Recheck condition
c4	Running		Sleep		Sleep	0	T_{c1} grabs data
c5	Running		Ready		Sleep	0	Oops! Woke T_{c2}
c6	Running		Ready		Sleep	0	
c1	Running		Ready		Sleep	0	
c2	Running		Ready		Sleep	0	
c3	Sleep		Ready		Sleep	0	Nothing to get
	Sleep	c2	Running		Sleep	0	
	Sleep	c3	Sleep		Sleep	0	Everyone asleep...

Thread Trace: Broken Solution (Version 2)

The Single Buffer Producer/Consumer Solution

- The solution here is once again a small one: use **two** condition variables, **instead of one**, in order to properly signal which type of thread should wake up when the state of the system changes.

```
1   cond_t  empty, fill;
2   mutex_t mutex;
3
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           Pthread_mutex_lock(&mutex);
8           while (count == 1)
9               Pthread_cond_wait(&empty, &mutex);
10          put(i);
11          Pthread_cond_signal(&fill);
12          Pthread_mutex_unlock(&mutex);
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          Pthread_mutex_lock(&mutex);
20          while (count == 0)
21              Pthread_cond_wait(&fill, &mutex);
22          int tmp = get();
23          Pthread_cond_signal(&empty);
24          Pthread_mutex_unlock(&mutex);
25          printf("%d\n", tmp);
26      }
27  }
```

- Producer threads wait on the condition **empty**, and signals **fill**. **Conversely**, consumer threads wait on **fill** and signal **empty**. By doing so, **the second problem above is avoided by design**: a consumer can never accidentally wake a consumer, and a producer can never accidentally wake a producer.

The Final Producer/Consumer Solution

```
1   int buffer[MAX];
2   int fill  = 0;
3   int use   = 0;
4   int count = 0;
5
6   void put(int value) {
7       buffer[fill] = value;
8       fill = (fill + 1) % MAX;
9       count++;
10  }
11
12  int get() {
13      int tmp = buffer[use];
14      use = (use + 1) % MAX;
15      count--;
16      return tmp;
17  }
```

The Final Put and Get Routines

```
1   cond_t empty, fill;
2   mutex_t mutex;
3
4   void *producer(void *arg) {
5       int i;
6       for (i = 0; i < loops; i++) {
7           Pthread_mutex_lock(&mutex);             // p1
8           while (count == MAX)                    // p2
9               Pthread_cond_wait(&empty, &mutex); // p3
10          put(i);                                 // p4
11          Pthread_cond_signal(&fill);             // p5
12          Pthread_mutex_unlock(&mutex);           // p6
13      }
14  }
15
16  void *consumer(void *arg) {
17      int i;
18      for (i = 0; i < loops; i++) {
19          Pthread_mutex_lock(&mutex);             // c1
20          while (count == 0)                      // c2
21              Pthread_cond_wait(&fill, &mutex);  // c3
22          int tmp = get();                        // c4
23          Pthread_cond_signal(&empty);            // c5
24          Pthread_mutex_unlock(&mutex);           // c6
25          printf("%d\n", tmp);
26      }
27  }
```

The Final Working Solution

提示：对条件变量使用 while（不是 if）

多线程程序在检查条件变量时，使用 while 循环总是对的。if 语句可能会对，这取决于发信号的语义。因此，总是使用 while，代码就会符合预期。

对条件变量使用 while 循环，这也解决了假唤醒（spurious wakeup）的情况。某些线程库中，由于实现的细节，有可能出现一个信号唤醒两个线程的情况[L11]。再次检查线程的等待条件，假唤醒是另一个原因。

Covering Conditions

一个简单的多线程内存分配库

```
1       // how many bytes of the heap are free?
2       int bytesLeft = MAX_HEAP_SIZE;
3
4       // need lock and condition too
5       cond_t c;
6       mutex_t m;
7
8       void *
9       allocate(int size) {
10          Pthread_mutex_lock(&m);
11          while (bytesLeft < size)
12              Pthread_cond_wait(&c, &m);
13          void *ptr = ...;                // get mem from heap
14          bytesLeft -= size;
15          Pthread_mutex_unlock(&m);
16          return ptr;
17      }
18
19      void free(void *ptr, int size) {
20          Pthread_mutex_lock(&m);
21          bytesLeft += size;
22          Pthread_cond_signal(&c);        // 当线程调用进入内存分配代码时，它可能会因为内存不足而等待。相应的，线程释放内存时，会发信号说有更多内存空闲。但是，代码中有一个问题：应该唤醒哪个等待线程？
23          Pthread_mutex_unlock(&m);
24      }
```

Covering Conditions (Cont.)

- Assume there are zero bytes free
  - Thread T_a calls `allocate(100)`.
  - Thread T_b calls `allocate(10)`.
  - Both T_a and T_b wait on the condition and go to sleep.
  - Thread T_c calls `free(50)`.

Which waiting thread should be woken up?

Covering Conditions (Cont.)

- Solution (Suggested by Lampson and Redell)
  - 用 pthread_cond_broadcast()代替上述代码中的 pthread_cond_signal()，唤醒所有的等待线程
  - `pthread_cond_broadcast()`的作用
    - Wake up **all waiting threads**.
    - Cost: too many threads might be woken.
    - Threads that shouldn't be awake will simply wake up, re-check the condition, and then go back to sleep.

Lampson 和 Redell 把这种条件变量叫作覆盖条件（covering condition），因为它能覆盖 所有需要唤醒线程的场景（保守策略）

作业

- 1、使用Condition Variables编写生产者消费者问题（假设缓冲区大小为10, 系统中有5个生产者，10个消费者）。并回答以下问题:
- 在生产者和消费者线程中修改条件时为什么要加mutex?
- 消费者线程中判断条件为什么要放在while而不是if中?

- 2、4 个线程，线程 1 循环打印 A, 线程 2 循环打印 B, 线程 3 循环打印 C, 线程 4 循环打印 D. 完成下面两个问题:
  1) 输出 ABCDABCDABCD…
  2) 输出 DCBADCBADCBA…