IT308: Operating Systems

Condition Variables

Review: Concurrency Objectives

- Mutual Exclusion – if one thread is in critical section then no other thread is
  - Solved using locks
- Ordering – B runs after A does something
  - Solved using condition variables

Example 1: Thread Join

```
pthread_t p1, p2;

// create child threads
pthread_create(&p1, NULL, mythread, "A");
pthread_create(&p2, NULL, mythread, "B");

…

// join waits for the child threads to finish
thr_join(p1, NULL);
thr_join(p2, NULL);

return 0;
```

how to implement `thr_join()`?

Waiting for an Event

- Parent thread has to wait until child terminates
- Option 1: spin until that happens
  - Waste of CPU time

Waiting for an Event

- Parent thread has to wait until child terminates
- Option 1: spin until that happens
  - Waste of CPU time
- Option 2: wait (sleep) in a queue until that happens
  - Better use of CPU time
  - Child thread will signal the parent to wake up before its termination

Generalizing Option 2

- Condition Variable: queue of waiting threads with two basic operations

Generalizing Option 2

- Condition Variable: queue of waiting threads with two basic operations
- Thread B waits for a signal on cv before running
  - cond_wait(cv, . . .)

Generalizing Option 2

- Condition Variable: queue of waiting threads with two basic operations
- Thread B waits for a signal on cv before running
  - cond_wait(cv, ...)
- Thread A sends signal to cv to wake-up one waiting thread
  - cond_signal(cv, ...)

Thread join: Attempt 1

Parent

```
void thr_join() {
  cond_wait(&c);
}
```

Child

```
void thr_exit() {
  cond_signal(&c);
}
```

- Does this work? If not, what's the problem?

Thread join: Attempt 1

Parent

```
void thr_join() {
  cond_wait(&c);
}
```

Child

```
void thr_exit() {
  cond_signal(&c);
}
```

- Does this work? If not, what's the problem?
- Child may run and call cond_signal() before parent called cond_wait()

Thread join: Attempt 1

Parent

```
void thr_join() {
  cond_wait(&c);
}
```

Child

```
void thr_exit() {
  cond_signal(&c);
}
```

- Does this work? If not, what's the problem?
- Child may run and call cond_signal() before parent called cond_wait()
  - cond_signal() signals nobody (parent is not there yet!)

Thread join: Attempt 1

Parent

```
void thr_join() {
  cond_wait(&c);
}
```

Child

```
void thr_exit() {
  cond_signal(&c);
}
```

- Does this work? If not, what's the problem?
- Child may run and call cond_signal() before parent called cond_wait()
  - cond_signal() signals nobody (parent is not there yet!)
  - parent goes to sleep indefinitely

Thread join: Attempt 2

Parent

```
void thr_join() {
  if (done == 0) {
    cond_wait(&c);
  }
}
```

Child

```
void thr_exit() {
  done = 1;
  cond_signal(&c);
}
```

- Let's keep some state then

Thread join: Attempt 2

Parent

```
void thr_join() {
  if (done == 0) {
    cond_wait(&c);
  }
}
```

Child

```
void thr_exit() {
  done = 1;
  cond_signal(&c);
}
```

- Let's keep some state then
- Is there a problem here?

Thread join: Attempt 2

Parent

```
void thr_join() {
  if (done == 0) {          //a
    cond_wait(&c);          //b
  }
}
```

Child

```
void thr_exit() {
  done = 1;                 //x
  cond_signal(&c);          //y
}
```

- Trace

  Parent: a b
  Child: x y

Thread join: Attempt 2

Parent

```
void thr_join() {
  if (done == 0) {          //a
    cond_wait(&c);          //b
  }
}
```

Child

```
void thr_exit() {
  done = 1;                 //x
  cond_signal(&c);          //y
}
```

- Trace

  Parent: a b
  Child: x y

- Again, parent may sleep indefinitely
- Solution?

Using Locks to Achieve Atomicity

- wait(cond_t *cv, mutex_t *lock)
  - assumes the lock is held when wait() is called
  - puts caller to sleep + releases the lock (atomically)
  - when awoken, reacquires lock before returning
- signal(cond_t *cv)
  - wake a single waiting thread (if >= 1 thread is waiting)
  - if there is no waiting thread, just return, doing nothing

Thread join: Attempt 3

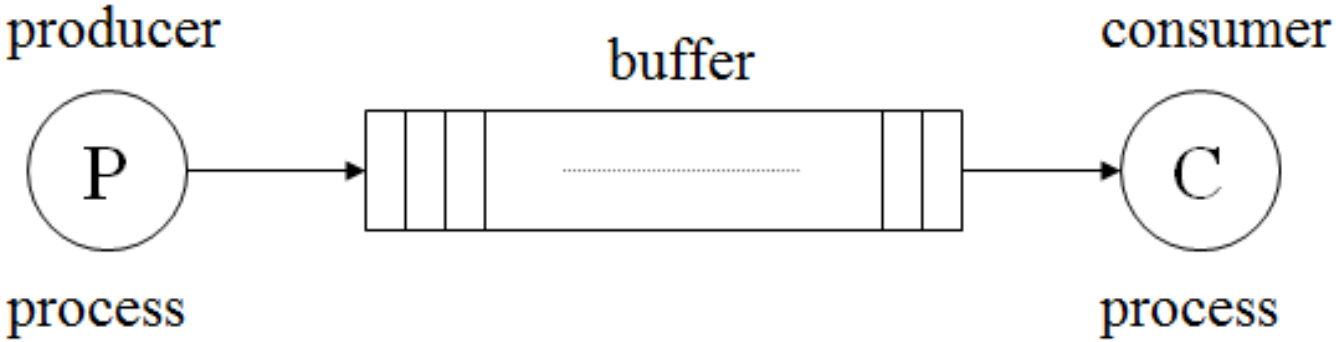
Parent:

```
void thread_join() {
        Mutex_lock(&m);              // w
        if (done == 0)               // x
                Cond_wait(&c, &m); // y
        Mutex_unlock(&m);            // z
}
```

Child:

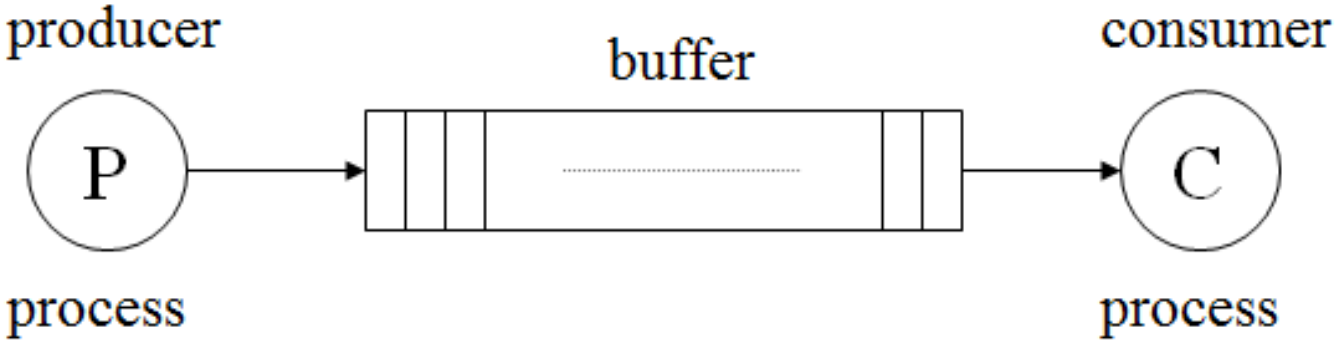
```
void thread_exit() {
        Mutex_lock(&m);              // a
        done = 1;                    // b
        Cond_signal(&c);             // c
        Mutex_unlock(&m);            // d
}
```

Producer/Consumer (bounded buffer) Problem



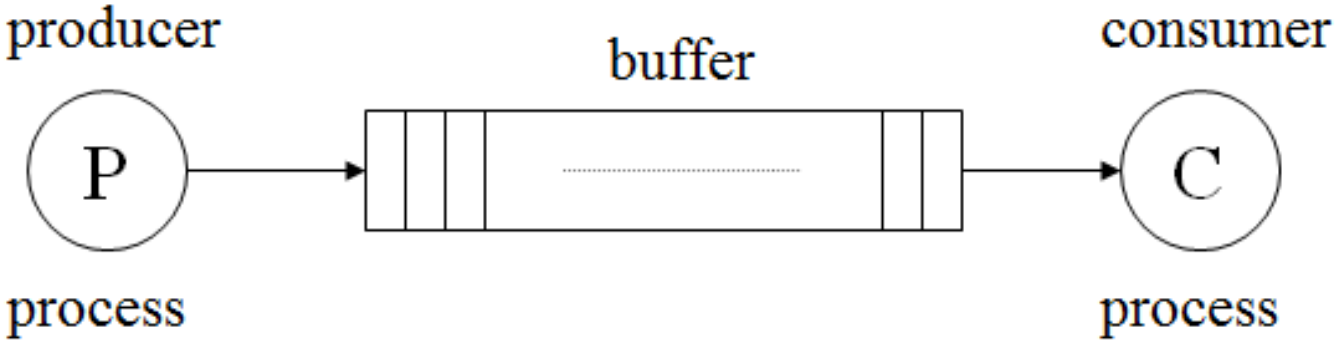
- Assume shared, finite size buffer
- from time to time, the producer adds items to buffer
- the consumer removes items from buffer

Producer/Consumer (bounded buffer) Problem



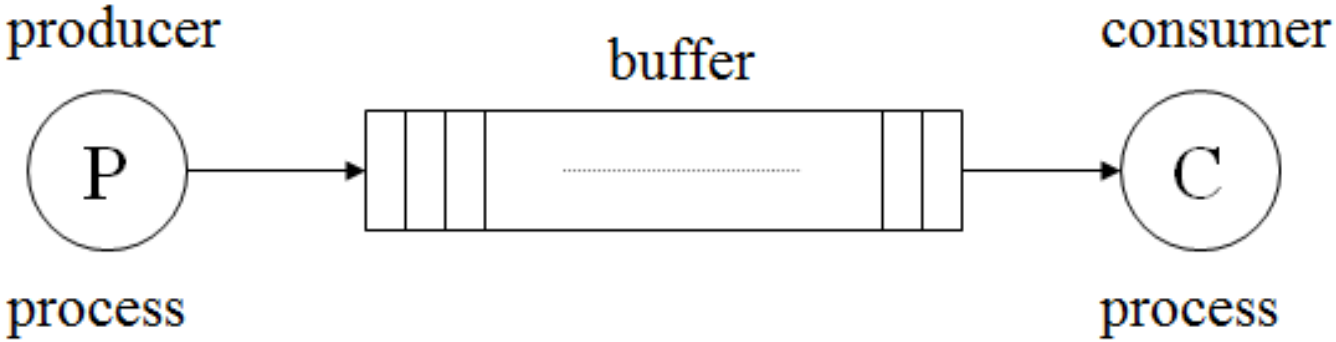
- Assume shared, finite size buffer
- from time to time, the producer adds items to buffer
- the consumer removes items from buffer
- careful synchronization required

Producer/Consumer (bounded buffer) Problem



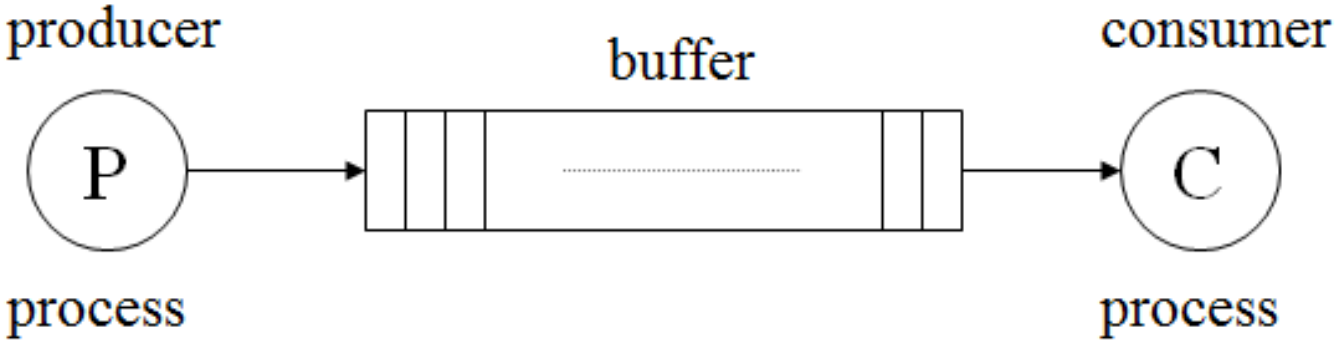
- Assume shared, finite size buffer
- from time to time, the producer adds items to buffer
- the consumer removes items from buffer
- careful synchronization required
  - the consumer must wait when the buffer is empty

Producer/Consumer (bounded buffer) Problem



- Assume shared, finite size buffer
- from time to time, the producer adds items to buffer
- the consumer removes items from buffer
- careful synchronization required
  - the consumer must wait when the buffer is empty
  - the producer must wait when the buffer is full

Producer/Consumer (bounded buffer) Problem



- Assume shared, finite size buffer
- from time to time, the producer adds items to buffer
- the consumer removes items from buffer
- careful synchronization required
  - the consumer must wait when the buffer is empty
  - the producer must wait when the buffer is full
- typical solution would use a shared variable count

The Put and Get Routines (Version 1)

- Assume buffer can hold only one item

```
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)

- Assume buffer can hold only one item

```
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      }
```

- Only put data into the buffer when count is zero (buffer is empty)

The Put and Get Routines (Version 1)

- Assume buffer can hold only one item

```
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      }
```

- Only put data into the buffer when count is zero (buffer is empty)
- Only get data from the buffer when count is one (buffer is full)

Producer/Consumer Threads (Version 1)

- Using a condition variable (and mutex) to synchronise producer and consumer

```
1   cond_t cond;
2   mutex_t mutex;
3
4   void *producer(void *arg) {
5       int i,loops = (int) arg;
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,loops = (int) arg;
18      for (i = 0; i < loops; i++) {
19          Pthread_mutex_lock(&mutex);              // c1
```

Producer/Consumer Threads (Version 1)

```
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      }
```

- p1–p3: A producer waits for the buffer to be empty
- c1–c3: A consumer waits for the buffer to be full

Producer/Consumer Threads (Version 1)

```
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      }
```

- p1–p3: A producer waits for the buffer to be empty
- c1–c3: A consumer waits for the buffer to be full
- The above code works for 1P and 1C. Can you find a problematic timeline with 2 consumers (still 1 producer)?

Thread Trace: Broken Solution (Version 1)

```
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);
    }
}
```

```
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
    }
}
```

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

Thread Trace: Broken Solution (Version 1)

```
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);
    }
}
```

```
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
    }
}
```

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	

Thread Trace: Broken Solution (Version 1)

```
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);
    }
}
```

```
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
    }
}
```

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	

Thread Trace: Broken Solution (Version 1)

```
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);
    }
}
```

```
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
    }
}
```

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

Thread Trace: Broken Solution (Version 1)

- After the producer woke T_{c1}, but before T_{c1} ever ran, the state of the bounded buffer changed by T_{c2}

Thread Trace: Broken Solution (Version 1)

- After the producer woke T_{c1}, but before T_{c1} ever ran, the state of the bounded buffer changed by T_{c2}
- There is no guarantee that when the woken thread runs, the state will still be as desired $\rightarrow$ Mesa semantics
  - Virtually every system ever built employs Mesa semantics

Thread Trace: Broken Solution (Version 1)

- After the producer woke T_{c1}, but before T_{c1} ever ran, the state of the bounded buffer changed by T_{c2}
- There is no guarantee that when the woken thread runs, the state will still be as desired $\rightarrow$ Mesa semantics
  - Virtually every system ever built employs Mesa semantics
- Hoare semantics provides a stronger guarantee that the woken thread will run immediately upon being woken

Producer/Consumer (Version 2)

- Consumer T_{c1} wakes up and re-checks the state of the shared variable
- If the buffer is empty, the consumer simply goes back to sleep

```
1       cond_t cond;
2       mutex_t mutex;
3
4       void *producer(void *arg) {
5           int i,loops = (int) arg;
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
```

Producer/Consumer (Version 2)

```
(Cont.)
16      void *consumer(void *arg) {
17          int i,loops = (int) arg;
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      }
```

- A simple rule to remember with condition variables is to always use while loops
- However, this code still has a bug!

Thread Trace: Broken Solution (Version 2)

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // c1
        while (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);
    }
}
```

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

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

Thread Trace: Broken Solution (Version 2)

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // c1
        while (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);
    }
}
```

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

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

Thread Trace: Broken Solution (Version 2)

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // c1
        while (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);
    }
}
```

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

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)

Thread Trace: Broken Solution (Version 2)

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // c1
        while (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);
    }
}
```

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

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

Thread Trace: Broken Solution (Version 2)

```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);            // c1
        while (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);
    }
}
```

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

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...

Producer/Consumer (Version 3)

- A consumer should not wake other consumers, only producers, and vice-versa
- Can't use the same cond var for two things (signaling the buffer is empty and the buffer is full)