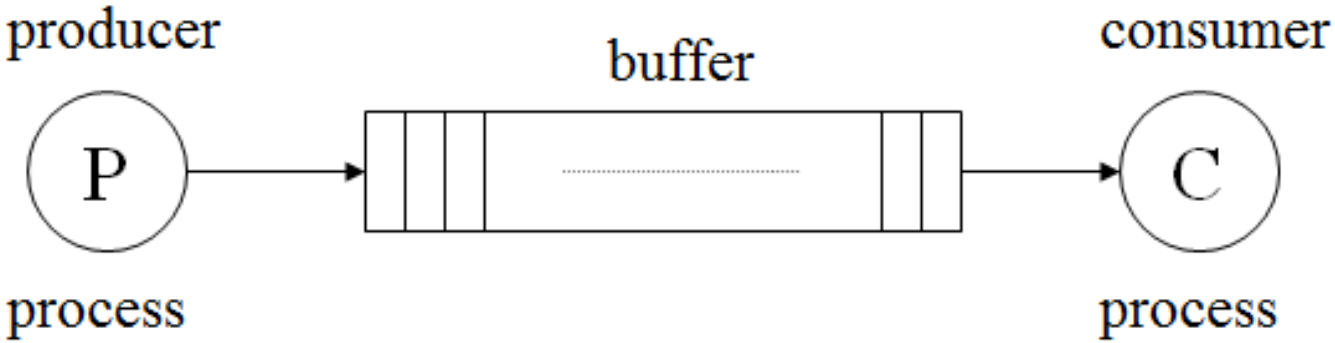
IE411: Operating Systems

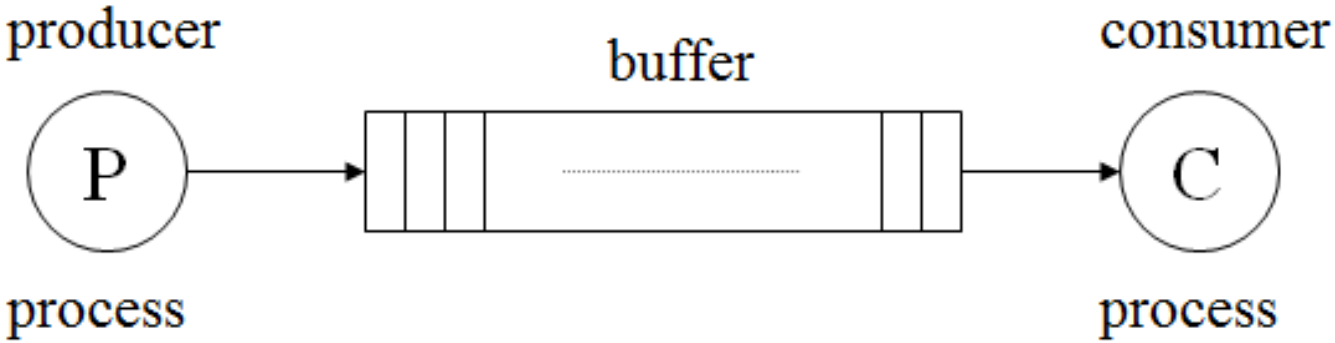
Producer-Consumer Problem

Producer/Consumer (bounded buffer) Problem



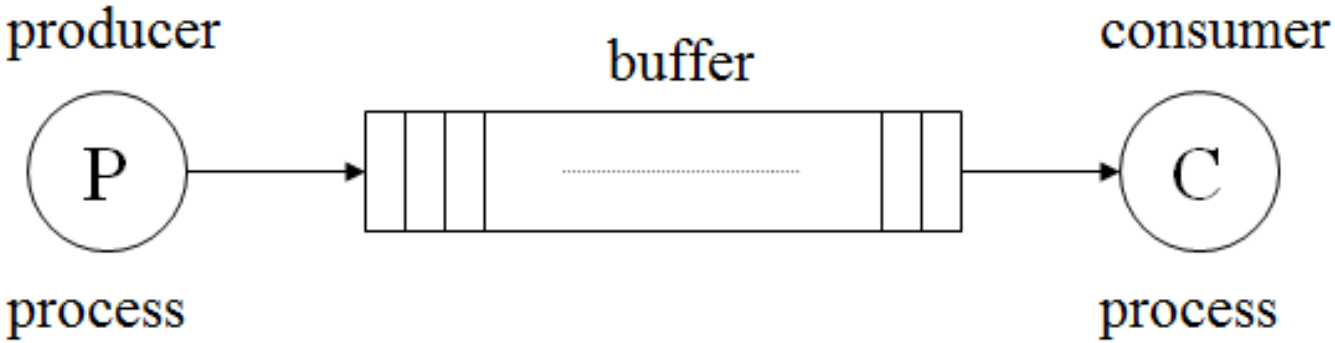
- Assume shared, finite size buffer
- from time to time
  - the producer adds items to buffer
  - the consumer removes items from buffer

Producer/Consumer (bounded buffer) Problem



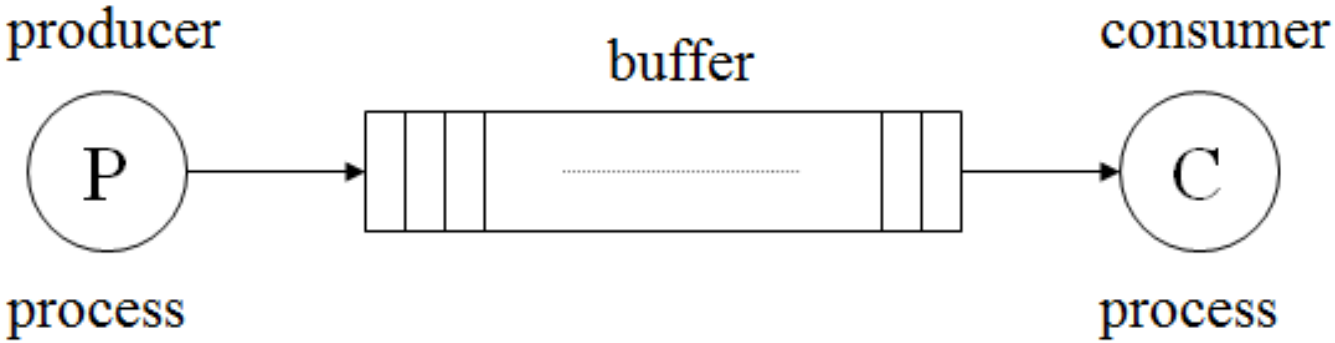
- Assume shared, finite size buffer
- from time to time
  - the producer adds items to buffer
  - the consumer removes items from buffer
- buffer is shared resource $\rightarrow$ synchronized access is required

Producer/Consumer (bounded buffer) Problem



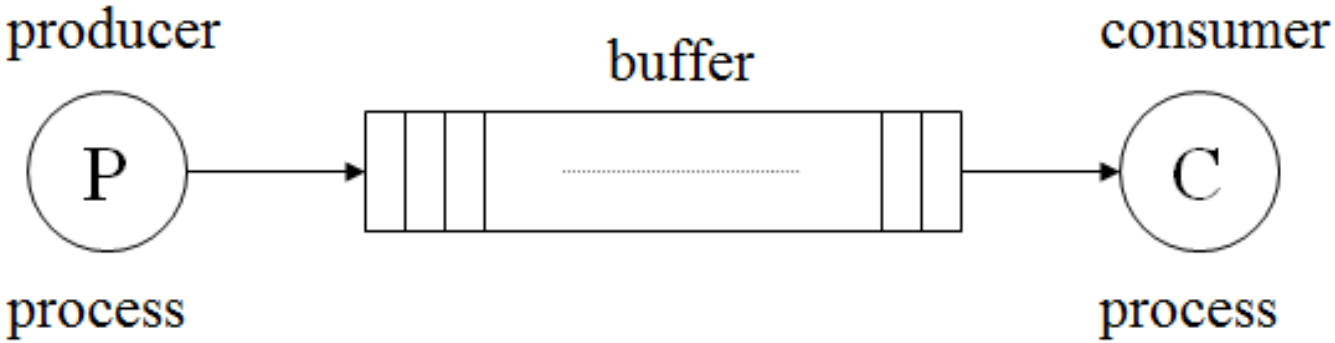
- Assume shared, finite size buffer
- from time to time
  - the producer adds items to buffer
  - the consumer removes items from buffer
- buffer is shared resource $\rightarrow$ synchronized access is required
  - the consumer must wait when the buffer is empty

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
- buffer is shared resource $\rightarrow$ synchronized access is 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
- buffer is shared resource $\rightarrow$ synchronized access is required
  - the consumer must wait when the buffer is empty
  - the producer must wait when the buffer is full

Bounded buffer example

- A bounded buffer is used when you pipe the output of one program into another
- Example: grep foo file.txt | wc -l
  - the grep process is the producer
  - The wc process is the consumer
  - Between them is an in-kernel bounded buffer

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      }
```

- 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      }
```

- typical solution would use a shared variable count
- 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      }
```

- typical solution would use a shared variable count
- 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: producer waits for the buffer to be empty
- c1–c3: 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: producer waits for the buffer to be empty
- c1–c3: consumer waits for the buffer to be full
- This code works for 1P and 1C. How about 1P and 2C?

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

- T_{c1} finds the buffer empty so it is waiting (line c3)

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

- T_{c1} finds the buffer empty so it is waiting (line c3)
- T_{c2} finds the buffer empty so it is waiting (line c3)

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)

- Both T_{c1} and T_{c2} initially find buffer empty so they are waiting (line c3)
- T_p adds an item to buffer (line p4), signals cond (line p5), waking up T_{c1}, waits on cond until signaled (line p3)

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

- T_{c1} removes item to buffer (line c4), signals cond (line c5), waking up T_{c2}, finds buffer empty, goes to sleep waiting on cond until signaled (line c3)

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...

- T_{c2}, being woken up by T_{c1}, finds buffer empty, goes to sleep waiting on cond (line c3)
- Everyone is sleeping $\rightarrow$ P can't produce $\rightarrow$ no forward progress

Approach 1: Wake up everyone

- When not sure if next waiting thread is the right one to wake up, just wake up all

Approach 1: Wake up everyone

- When not sure if next waiting thread is the right one to wake up, just wake up all
- Not the most elegant solution
  - Probably bad for performance: all awoken threads will compete for mutex again
  - But a good fallback mechanism to ensure correctness

Approach 1: Wake up everyone

- When not sure if next waiting thread is the right one to wake up, just wake up all
- Not the most elegant solution
  - Probably bad for performance: all awoken threads will compete for mutex again
  - But a good fallback mechanism to ensure correctness
- Need a new API: cond_broadcast(cv)
  - Semantics: wakes up all the threads waiting on cv

Approach 2: Use multiple CVs

- Two different conditions in bounded buffer problem
  - waiting for buffer to become empty (parent)
  - waiting for buffer to become full (child)

Approach 2: Use multiple CVs

- Two different conditions in bounded buffer problem
  - waiting for buffer to become empty (parent)
  - waiting for buffer to become full (child)
- Use a separate CV for each condition using cond_wait() and cond_signal()
- More elegant, better-performing solution than using cond_broadcast()

Single buffer solution: 2 CVs

- Producer threads wait on the condition `empty` and signal `fill`
- Consumer threads wait on `fill` and signal `empty`

```
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
```

Single buffer solution: 2 CVs

```
(Cont.)
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      }
```

Multiple buffer 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

Multiple buffer solution

```
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
```

Multiple buffer solution

```
(Cont.)
23                  Pthread_cond_signal(&empty);                    // c5
24                  Pthread_mutex_unlock(&mutex);                   // c6
25                  printf("%d\n", tmp);
26              }
27          }
```

The Final Working Solution (Cont.)

- p2: a producer sleeps only if all buffers are currently filled
- c2: a consumer sleeps only if all buffers are currently empty

One more example

- Assume there are 0 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)`
- Q: Which waiting thread should be woken up?

One more example

```
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);        // whom to signal??
23      Pthread_mutex_unlock(&m);
24  }
```

Solution

- Wake up all waiting threads!
- Replace `pthread_cond_signal()` with `pthread_cond_broadcast()`