

Operating Systems

CSCI 3150

Lecture 6: Synchronization (II)

-- Condition Variables

Hong Xu

<https://github.com/henryhxu/CSCI3150>

Condition Variables

- ▣ There are many cases where a thread wishes to check whether a **condition** is true before continuing its execution.
- ▣ Example:
 - ◆ A parent thread might wish to check whether a child thread has *completed*.
 - ◆ This is often called a `join()`.

Condition Variables

A Parent Waiting For Its Child

```
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     }
```

What we would like to see here is:

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

Parent waiting for child: Spin-based Approach

```
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     }
```

- ◆ This is hugely inefficient as the parent spins and **wastes CPU time**.

How to wait for a condition

▣ Condition variable

- ◆ Queue of threads
- ◆ **Waiting** on the condition
 - An explicit queue that threads can put themselves on when some state of execution is not as desired.
- ◆ **Signaling** on the condition
 - Some other thread, *when it changes its state*, can wake one of those waiting threads and allow them to continue.

▣ Three in a package

- ◆ condition variable `c`
- ◆ state variable `m`
- ◆ lock `L`; `//` to protect state variable

Definition and Routines

▣ Declare condition variable

```
pthread_cond_t c;
```

- ◆ Proper initialization is required.

▣ Operation (the POSIX calls)

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

- ◆ The wait() call takes a mutex as a parameter.
 - The wait() call release the lock and put the calling thread to sleep.
 - When the thread wakes up, it must re-acquire the lock.

Parent waiting for Child: Use a condition variable

```
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
```

Parent waiting for Child: Use a condition variable

```
(cont.)
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

▣ Parent:

- ◆ Creates the child thread and continues running itself.
- ◆ Calls into `thr_join()` to wait for the child thread to complete.
 - Acquires the lock.
 - Checks if the child is done.
 - Puts itself to sleep by calling `wait()`.
 - Releases the lock.

▣ Child:

- ◆ Prints the message "child".
- ◆ Calls `thr_exit()` to wake up the parent thread.
 - Grabs the lock.
 - Sets the state variable `done`.
 - Signals the parent thus waking it.

The importance of the state variable `done`

```
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     }
```

`thr_exit()` and `thr_join()` without variable `done`

- ◆ Imagine the case where the *child runs immediately*.
 - The child will signal, but there is no thread asleep on the condition.
 - When the parent runs, it will call wait and be stuck.
 - No thread will ever wake it.

Another poor implementation

```
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      }
```

- ◆ The issue here is a subtle **race condition**.
 - The parent calls `thr_join()`.
 - The parent checks the value of `done`.
 - It will see that it is 0 and try to go to sleep.
 - *Just before* it calls `wait` to go to sleep, the parent is interrupted and the child runs.
 - The child changes the state variable `done` to 1 and signals.
 - But no thread is waiting and thus no thread is woken.
 - When the parent runs again, it sleeps forever.

The Producer / Consumer (Bound Buffer) Problem

□ **Producer**

- ◆ **Produce** data items
- ◆ Wish to place data items in a buffer

□ **Consumer**

- ◆ Grab data items out of the buffer **consume** them in some way

□ **Example: Multi-threaded web server**

- ◆ *A producer* puts HTTP requests in to a work queue
- ◆ *Consumer threads* take requests out of this queue and process them

Bounded buffer

- 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.
 - ◆ Bounded buffer is Shared resource → **Synchronized access** is required.

The Put and Get Routines (Version 1)

```
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.
 - i.e., when the buffer is *empty*.
- ◆ Only get data from the buffer when `count` is one.
 - i.e., when the buffer is *full*.

Producer/Consumer Threads (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** puts an integer into the shared buffer loops number of times.
- ◆ **Consumer** gets the data out of that shared buffer.

Producer/Consumer: Single CV and If Statement

- A single condition variable `cond` and associated lock `mutex`

```
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
```


Producer/Consumer: Single CV and If Statement

```
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     }
```


- ♦ p1-p3: A producer waits for the buffer to be empty.
- ♦ c1-c3: A consumer waits for the buffer to be full.
- ♦ With just *a single producer* and *a single consumer*, the code works.


If we have more than one of producer and consumer?

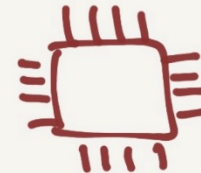
③ C_1

③ C_2

③ P_1


buffer


ready Q



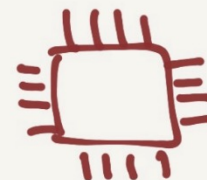
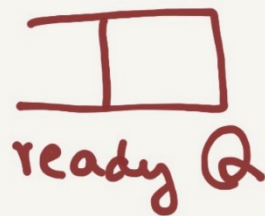
sleep queue: { }

③ C_1

③ C_2

③ P_1

C_1 executes first.



sleep queue: { }

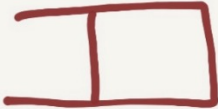
③ C_1

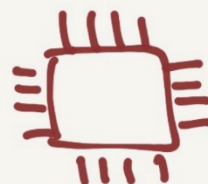
③ C_2

③ P_1

C_1 executes first.


buffer


ready Q



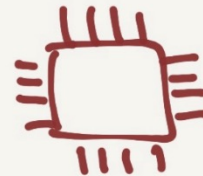
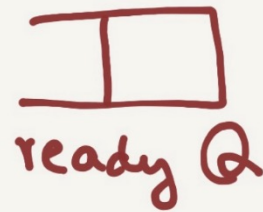
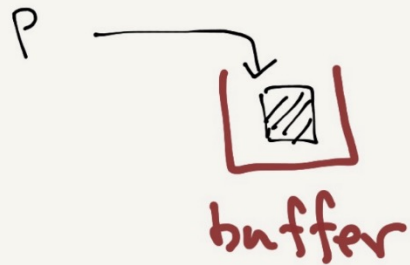
$\{C_1\}$

③ C_1

③ C_2

③ P_1

P_1 executes. and send signal.



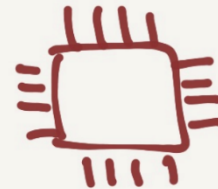
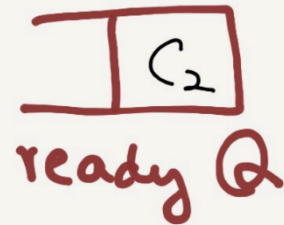
$\{C_1\}$

③ C_1

③ C_2

③ P_1

C_2 arrives.



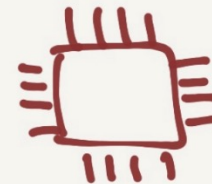
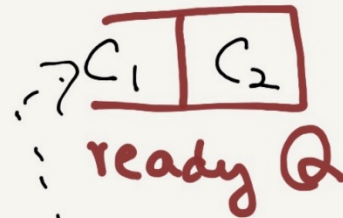
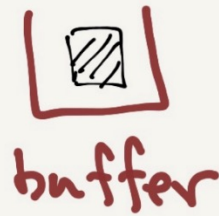
$\{C_1\}$

③ C₁

③ C₂

③ P₁

C₁ gets signal.



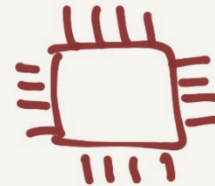
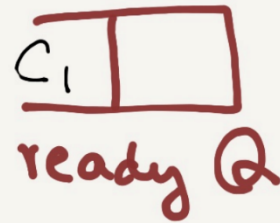
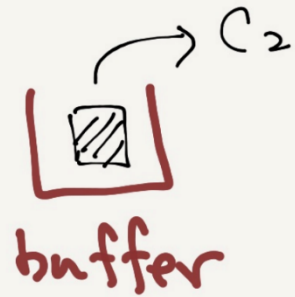
~~{X}~~

③ C₁

③ C₂

③ P₁

C₂ consumes buffer.




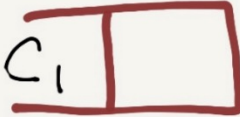
{ }

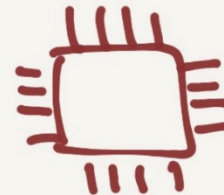
③ C_1

③ C_2

③ P_1


buffer


ready Q



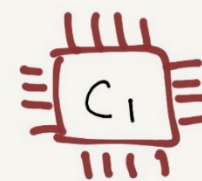
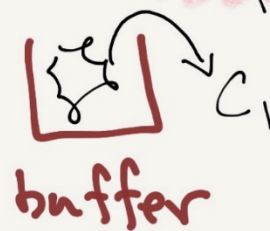
{ P_1 }

③ C_1

③ C_2

③ P_1

C_1 reads the ^{empty} buffer!!!



{ P_1 }



Reason: C_1 blindly reads the buffer after it gets woken up.

Thread Trace: Broken Solution (Version 1)

	State		State		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	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	sneaks in ...
	Ready	c2	Running		Sleep	1	
	Ready	c4	Running		Sleep	0	... and grabs data
	Ready	c5	Running		Ready	0	awoken
	Ready	c6	Running		Ready	0	
c4	Running		Ready		Ready	0	Oh oh! No data

Thread Trace: Broken Solution (Version 1)

- ▣ The problem arises for a simple reason:
 - ◆ 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 → 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: Single CV and While

- 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;
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: Single CV and While


```
(Cont.)
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     }
```


- ◆ A simple rule to remember with condition variables is to **always use while loops**.
- ◆ However, this code still has a bug (*next page*).

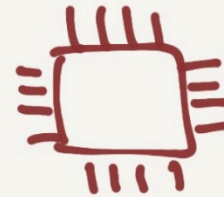
③ C_1

③ C_2

③ P_1


buffer


ready Q



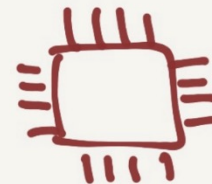
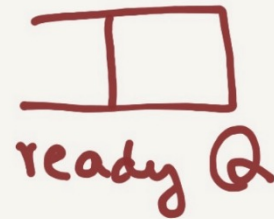
sleep queue: { }

③ C_1

③ C_2

③ P_1

C_1 executes first.



sleep queue: { }


③ C_1

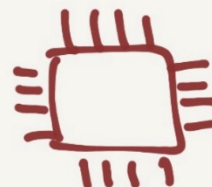
③ C_2

③ P_1

C_1 executes first.


buffer


ready Q



$\{C_1\}$

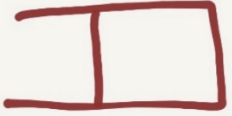
③ C_1

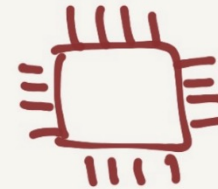
③ C_2

③ P_1

C_2 executes next.


buffer


ready Q



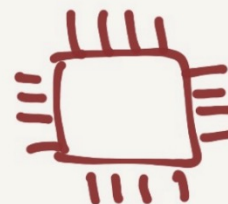
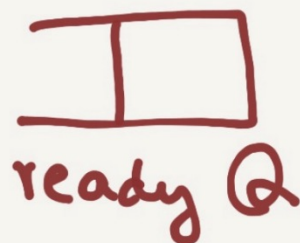
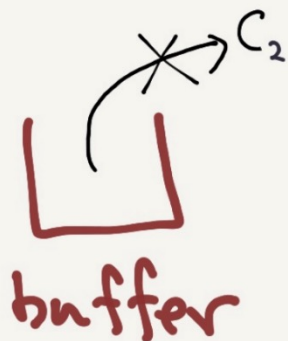
$\{C_1\}$

③ C_1

③ C_2

③ P_1

C_2 executes next



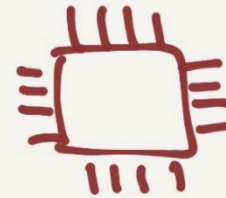
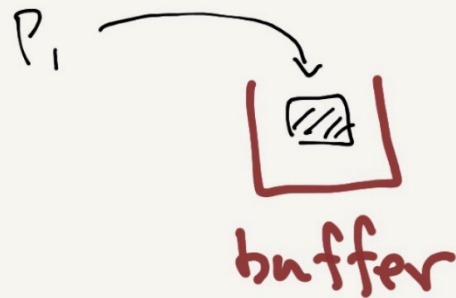
sleep queue : $\{C_1, C_2\}$

③ C_1

③ C_2

③ P_1

P_1 executes. and send signal.



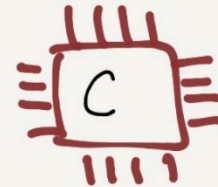
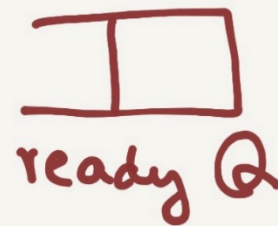
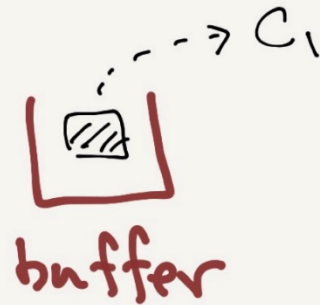
Sleep queue: $\{C_1, C_2, P_1\}$

③ C_1

③ C_2

③ P_1

C_1 gets signal, executes and consumes buffer.



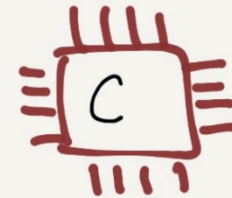
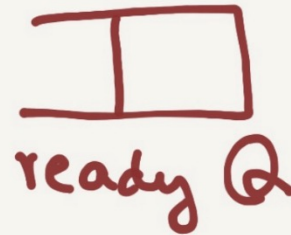
sleep queue: $\{P_1, C_2\}$

③ C_1

③ C_2

③ P_1

C_1 sends signal.




sleep queue : $\{P, C\}$

③ C_1

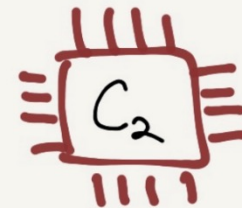
③ C_2

③ P_1

C_2 gets signal!


buffer


ready Q



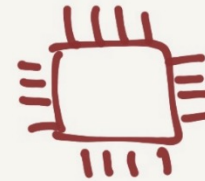
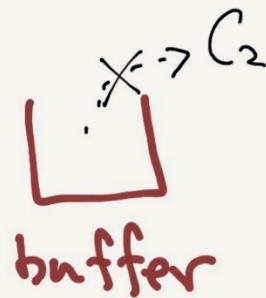
sleep queue: $\{P_1, C_1\}$

③ C_1

③ C_2

③ P_1

C_2 gets executed. \rightarrow finds buffer empty. \rightarrow sleep.



sleep queue: $\{P_1, C_1, C_2\}$

Reason: C_2 gets signal.

P_1 should have gotten it!



Thread Trace: Broken Solution (Version 2)

	State		State		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	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	grabs data
c5	Running		Ready		Sleep	0	Oops! Woke

Thread Trace: Broken Solution (Version 2) (Cont.)

	State		State		State	Count	Comment
...	(<i>cont.</i>)
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 ...

- ◆ A consumer should not wake other consumers, only producers, and vice-versa.

The single Buffer Producer/Consumer Solution

- ▣ Use **two** condition variables and while
 - ◆ **Producer** threads wait on the condition `empty`, and signals `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
```

The single Buffer Producer/Consumer Solution

(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     }
```

The Final Producer/Consumer Solution

- ▣ More **concurrency** and **efficiency** → Add more buffer slots.
 - ◆ Allow concurrent production or consuming to take place.
 - ◆ Reduce context switches.

```
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

The Final Producer/Consumer Solution (Cont.)

```
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
```

The Final Producer/Consumer Solution (Cont.)

```
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 (Cont.)

- ♦ p2: **A producer** only sleeps if all buffers are currently filled.
- ♦ c2: **A consumer** only sleeps if all buffers are currently empty.