Condition Variables

Why Condition ?

- cases where a thread wishes to check whether a **condition** is true before continuing its execution

```
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 }
```

output

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

Why Condition ?

- cases where a thread wishes to check whether a **condition** is true before continuing its execution

```
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 }
```

inefficient as the parent spins and wastes CPU time

We need a way to put the parent to sleep until the condition we are waiting for comes true.

Condition Variable

- A **condition variable** is an explicit queue [Hoare, 1974]
  - wait(): put a thread on the queue (sleep) when some state of execution (i.e., condition) is not as desired
  - signal(): wake one (or more) of waiting threads when a thread has changed the state
- POSIX
  - pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
  - pthread_cond_signal(pthread_cond_t *c);

Charles Antony Richard Hoare
Turing Award (1980)

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 }
```

wait() releases the mutex lock and put the calling thread to sleep (atomically). When the thread wakes up (by signal()), it must re-acquire the lock before returning to the caller.

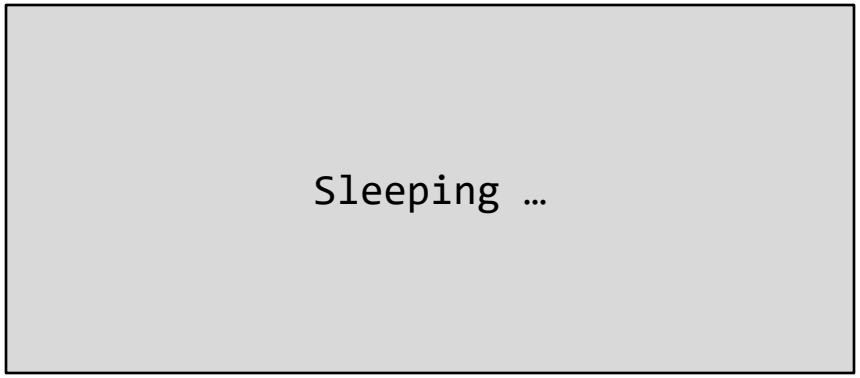
```
17
18 void thr_join() {
19    Pthread_mutex_lock(&m);
20    while (done == 0)
21          Pthread_cond_wait(&c, &m);
22    Pthread_mutex_unlock(&m);
23 }
24
25 int main(int argc, char *argv[]) {
26    printf("parent: begin\n");
27    pthread_t p;
28    Pthread_create(&p, NULL, child, NULL);
29    thr_join();
30    printf("parent: end\n");
31    return 0;
32 }
```

Q1: Why acquire mutex_lock before wait/signal?
To prevent race conditions
Hold the lock when calling signal or wait!
Q2: Why do we need the "done" variable?
Permanent sleep if signal() is called before wait()?
Q3: Why does join() check "done" repeatedly?
Multiple threads can wait.

Scenario 1

Parent

```
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();

18 void thr_join() {
19    Pthread_mutex_lock(&m);
20    while (done == 0)
21          Pthread_cond_wait(&c, &m);
```




```
20    while (done == 0)
22    Pthread_mutex_unlock(&m);
23 }
```

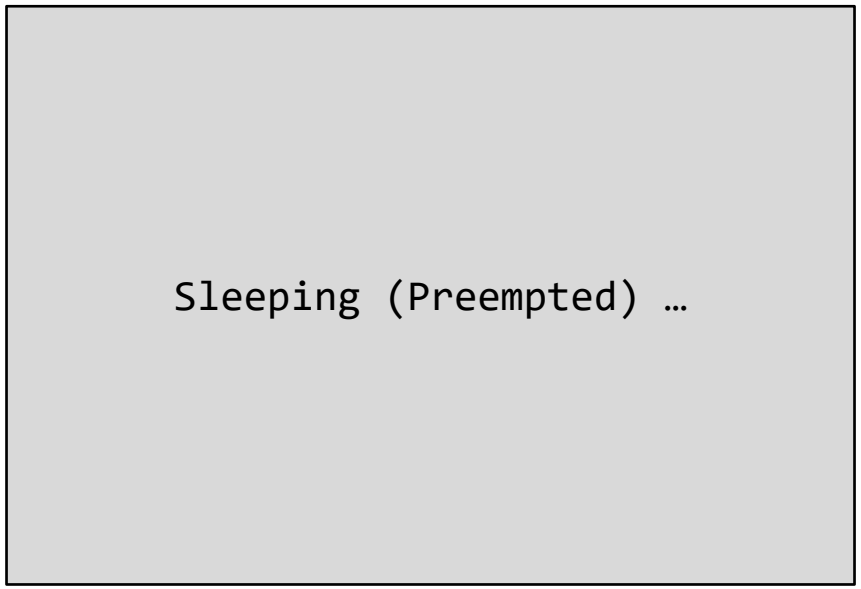
Child

```
12 void *child(void *arg) {
13    printf("child\n");
14    thr_exit();

5 void thr_exit() {
6     Pthread_mutex_lock(&m);
7     done = 1;
8     Pthread_cond_signal(&c);
9     Pthread_mutex_unlock(&m);
10 }
```

Scenario 2

Parent

```
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();

18 void thr_join() {
19    Pthread_mutex_lock(&m);
20    while (done == 0)  // now done is 1
22    Pthread_mutex_unlock(&m);
23 }
```

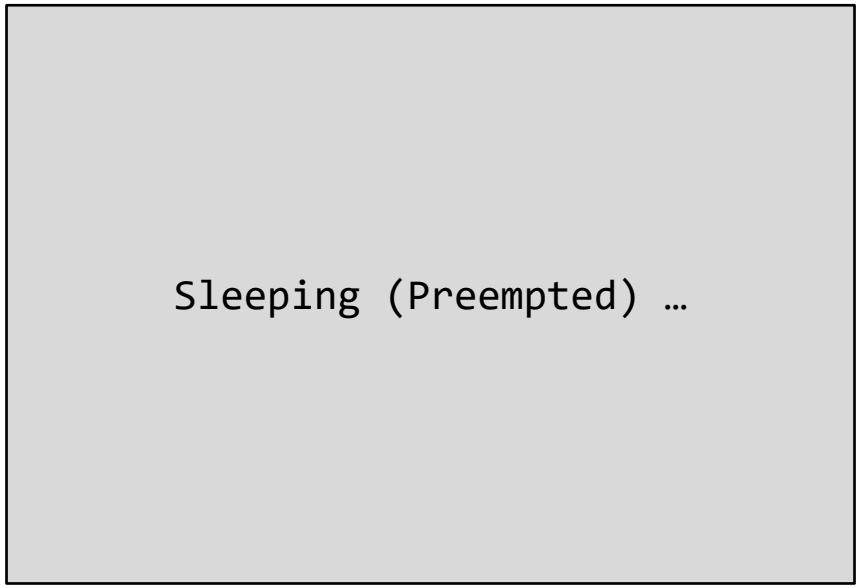
Child

```
12 void *child(void *arg) {
13    printf("child\n");
14    thr_exit();

5 void thr_exit() {
6     Pthread_mutex_lock(&m);
7     done = 1;
8     Pthread_cond_signal(&c); // none is
waiting
9     Pthread_mutex_unlock(&m);
10 }

15    return NULL;
16 }
```

Why we need the state variable done?

Parent

```
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();

18 void thr_join() {
19    Pthread_mutex_lock(&m);
20    while (done == 0)   // now done is 1
21          Pthread_cond_wait(&c, &m);
```

Child

```
12 void *child(void *arg) {
13    printf("child\n");
14    thr_exit();

5 void thr_exit() {
6     Pthread_mutex_lock(&m);
7     done = 1;
8     Pthread_cond_signal(&c); // none is
waiting
9     Pthread_mutex_unlock(&m);
10 }

15    return NULL;
16 }
```

no thread will wake it

Why one need to hold a lock to signal/wait?

Parent

```
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();

18 void thr_join() {
19    Pthread_mutex_lock(&m);
20    while (done == 0)
```

Preemtpted

```
21        Pthread_cond_wait(&c, &m);
22    Pthread_mutex_unlock(&m);
23 }
```

Child

```
12 void *child(void *arg) {
13    printf("child\n");
14    thr_exit();

5 void thr_exit() {
6     Pthread_mutex_lock(&m);
7     done = 1;
8     Pthread_cond_signal(&c); // none is
waiting
9     Pthread_mutex_unlock(&m);
10 }
```

sleep forever

Producer/Consumer (Bounded Buffer) Problem

- Producers generate data items and place them in a bounded buffer
- Consumers grab items from the buffer and consume them
- For example, in a multi-threaded web server,
  - producer puts HTTP requests into a work queue
  - consumer threads take requests out of this queue and process them.

```
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;
}
```

```
void *producer(void *arg) {
  int i;
  int loops = (int) arg;
  for (i = 0; i < loops; i++) {
    put(i);
  }
}
void *consumer(void *arg) {
  int i;
  while (1) {
    int tmp = get();
    printf("%d\n", tmp);
  }
}
```

The bounded buffer is a shared resource
Need synchronized access to it, lest a race condition arise

Solution 1: Single CV & If Statement

```
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 }
```

a single condition variable cond
and associated lock mutex

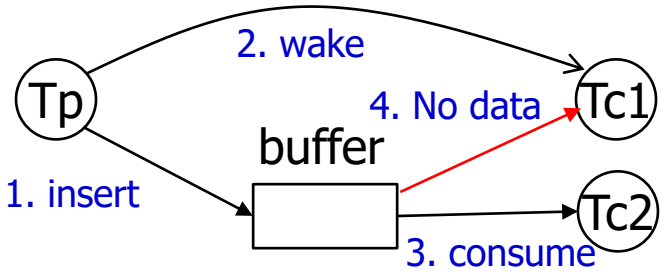
With just a single producer and a single consumer, the code works.

However, if we have more than one of these threads?

Thread Trace: Broken Solution (Version 1)

two consumers (Tc1 and Tc2) and one producer (Tp)

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



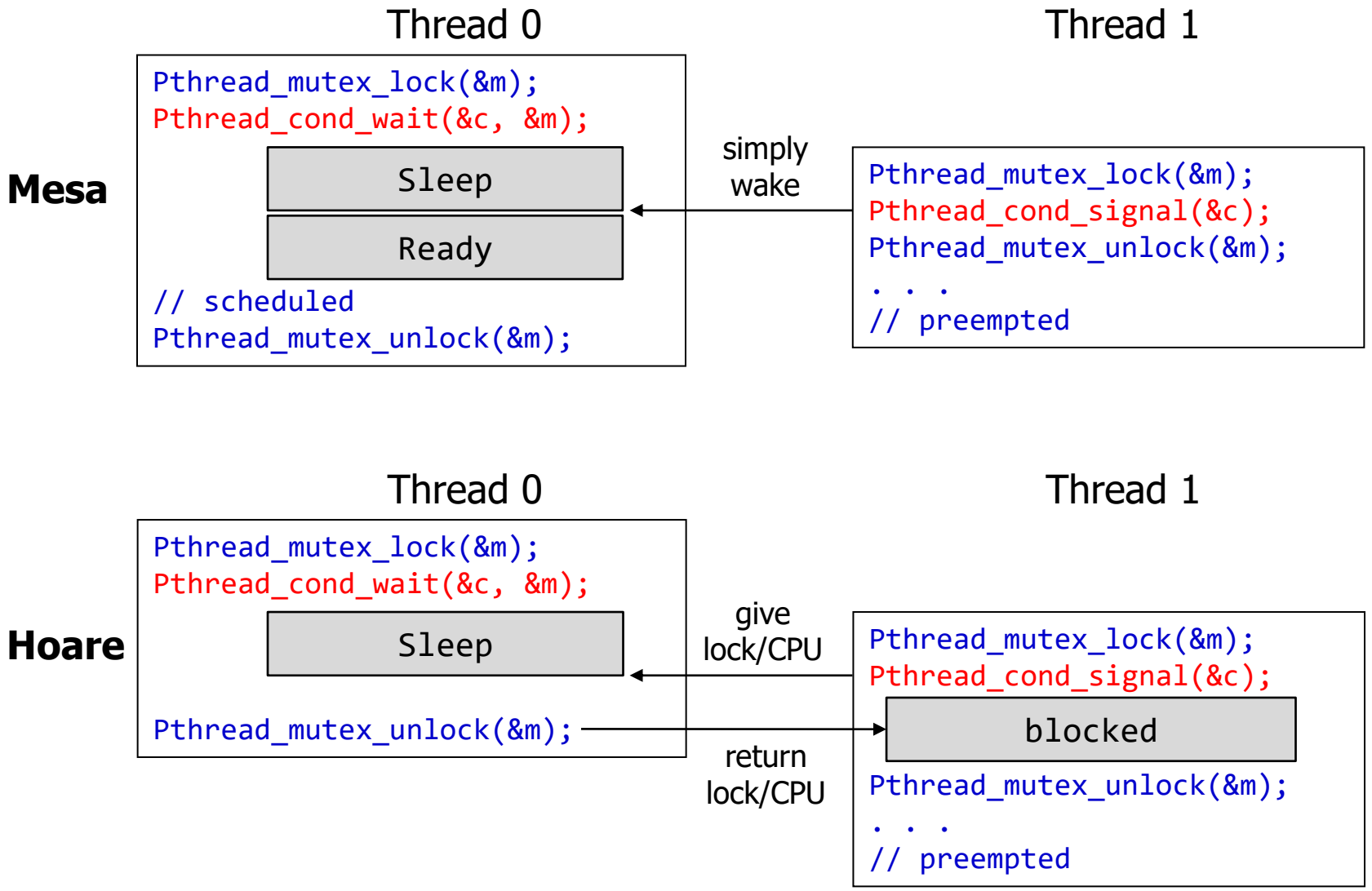
Problem: state change before the woken thread runs
➔ mesa-style

Mesa-style vs. Hoare-style

- **Mesa-style (Nachos, most real OS)**
  - Signaler keeps lock/processor (nonblocking condition variable)
  - Waiter simply put on ready queue, with no special priority
  - Waiter may have to wait for lock again
- **Hoare-style (most theory, textbook)**
  - Signaler passes lock/CPU to waiter; waiter runs immediately
  - Waiter gives lock/processor back to signaler when it exits critical section or if it waits again

- **For Mesa-semantics, the woken thread must re-check the condition (use "while").**
- **For Hoare-semantics you can change it to "if"**

Mesa-style vs. Hoare-style

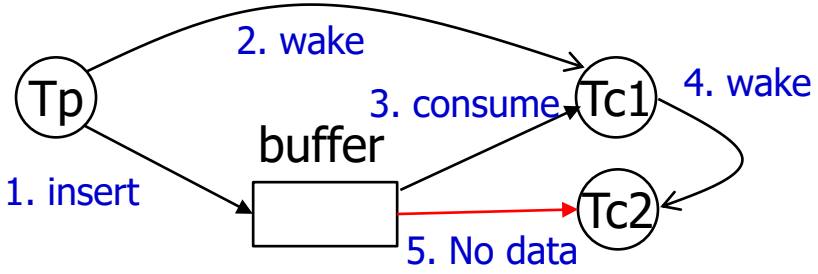


Solution 2: Single CV & While Statement

```
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 }
```

Thread Trace: Broken Solution (Version 2)

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...



Problem: single cond variable

Solution 3: Two CV & While Statement

```
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 == 1)                     // 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 }
```

More Concurrency and Efficiency

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

```
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 }
```

Covering Conditions

Scenario
1.No free bytes
2.Ta calls allocate(100)
3.Tb calls allocate(10)
4.Tc calls free(50)
- calls signal to wake
- Which thread?

•Pthread_cond_broadcast
- wakes up *all* waiting threads
- negative performance impact **(thundering herd)**

Pthread_cond_broadcast()

```
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 }
```

Homework

- Homework in Chap 30 (Condition Variables)