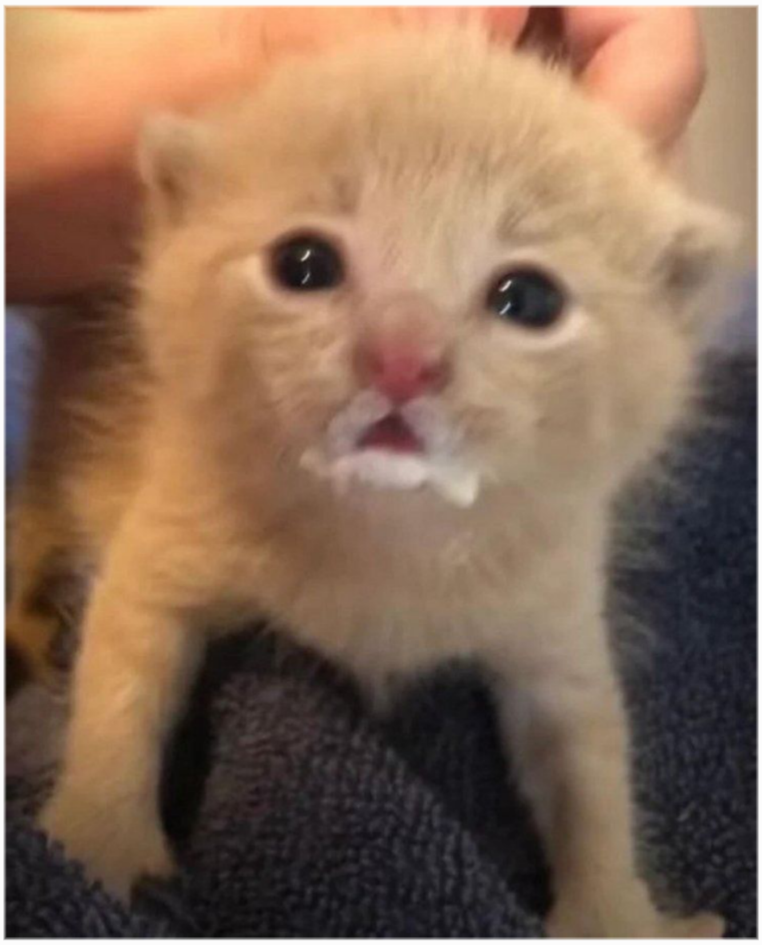
24. Condition Variables



locks are not the only primitives that are needed to build concurrent programs

thread wishes to check whether a condition is true before continuing its execution (ex. parent check child completion / join())

Try 1. parents spin =>wastes CPU times = inefficient

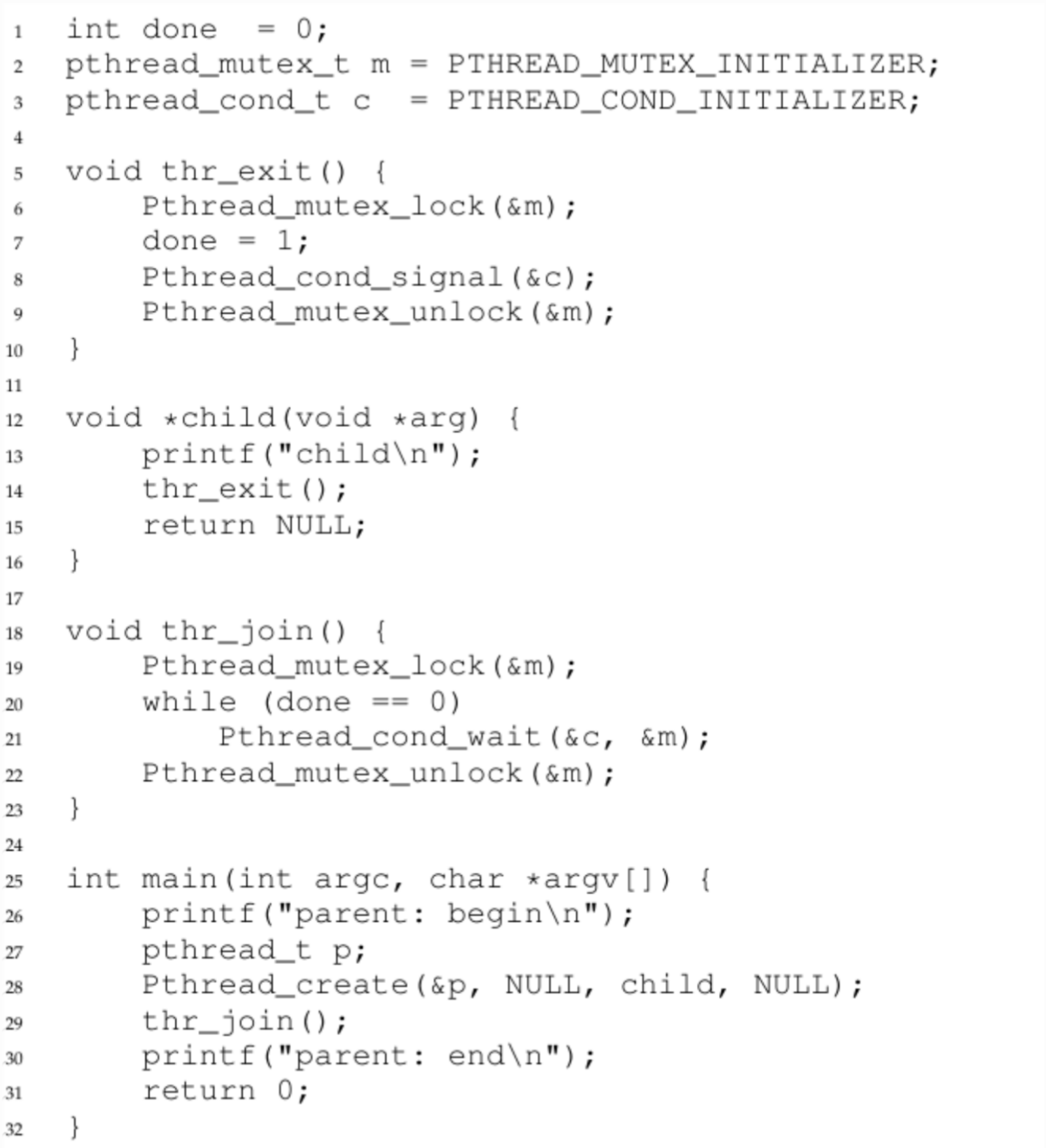
How To Wait for a Condition?

Condition Variable

explicit queue that threads can put themselves on
when some state of execution (= some condition) is not as desired (by waiting on the condition)

some other thread, when it changes said state, can then wake one (or more) of those waiting threads and thus allow them to continue (by signaling on the condition)

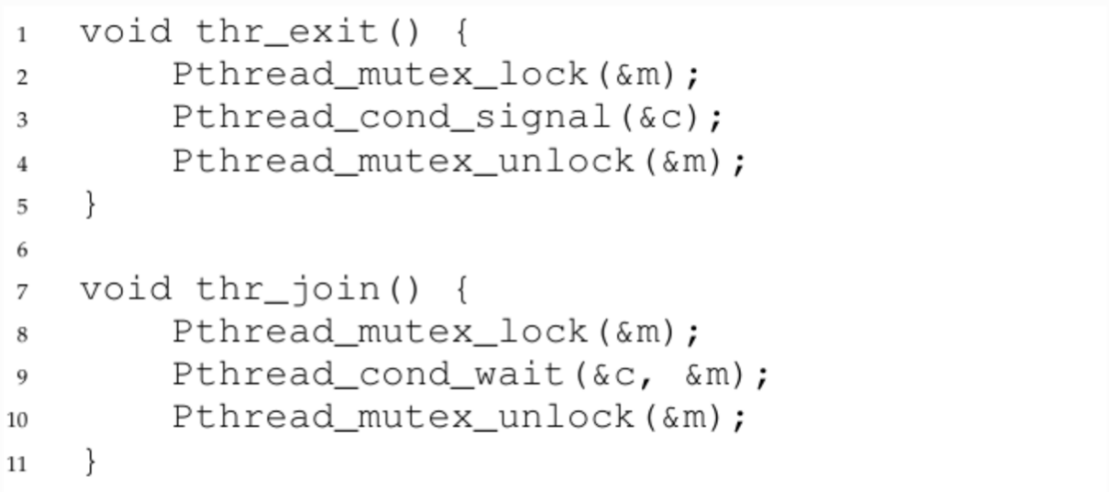
- pthread_cond_t c;
- pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);
- pthread_cond_signal (pthread_cond_t *c);



```
int done   = 0;
pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t c  = PTHREAD_COND_INITIALIZER;

void thr_exit() {
    Pthread_mutex_lock(&m);
    done = 1;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void *child(void *arg) {
    printf("child\n");
    thr_exit();
    return NULL;
}

void thr_join() {
    Pthread_mutex_lock(&m);
    while (done == 0)
        Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t p;
    Pthread_create(&p, NULL, child, NULL);
    thr_join();
    printf("parent: end\n");
    return 0;
}
```

Figure 30.3: **Parent Waiting For Child: Use A Condition Variable**

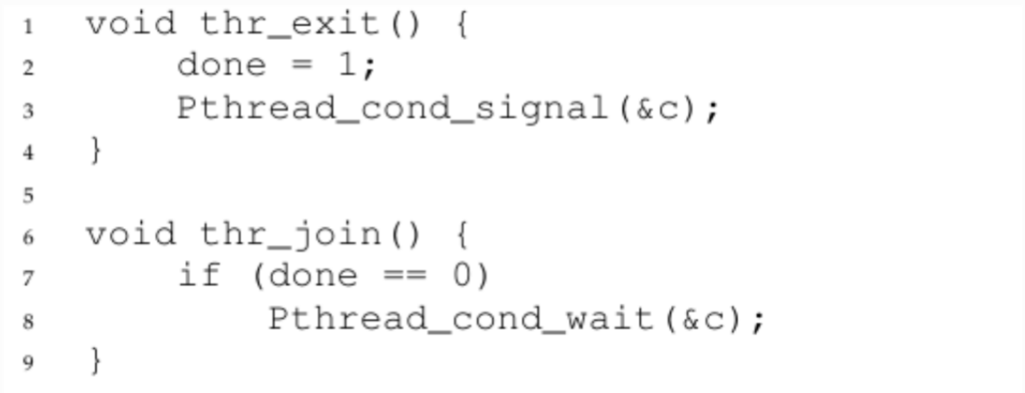
the responsibility of wait()
: release the lock and put the calling thread to sleep (atomically).
when the thread wakes up (after some other thread has signaled it),
it must re-acquire the lock before returning to the caller.



```
void thr_exit() {
    Pthread_mutex_lock(&m);
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}

void thr_join() {
    Pthread_mutex_lock(&m);
    Pthread_cond_wait(&c, &m);
    Pthread_mutex_unlock(&m);
}
```

Figure 30.4: **Parent Waiting: No State Variable**

Problem :
child runs first -> parents run -> call wait -> stuck forever (no one wakes)
=> **state value** needed.
records the value the threads are interested in knowing
the sleeping, waking, and locking all are built around it



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

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

Figure 30.5: **Parent Waiting: No Lock**

Problem : got interrupted right after "if(done == 0)" => sleep forever

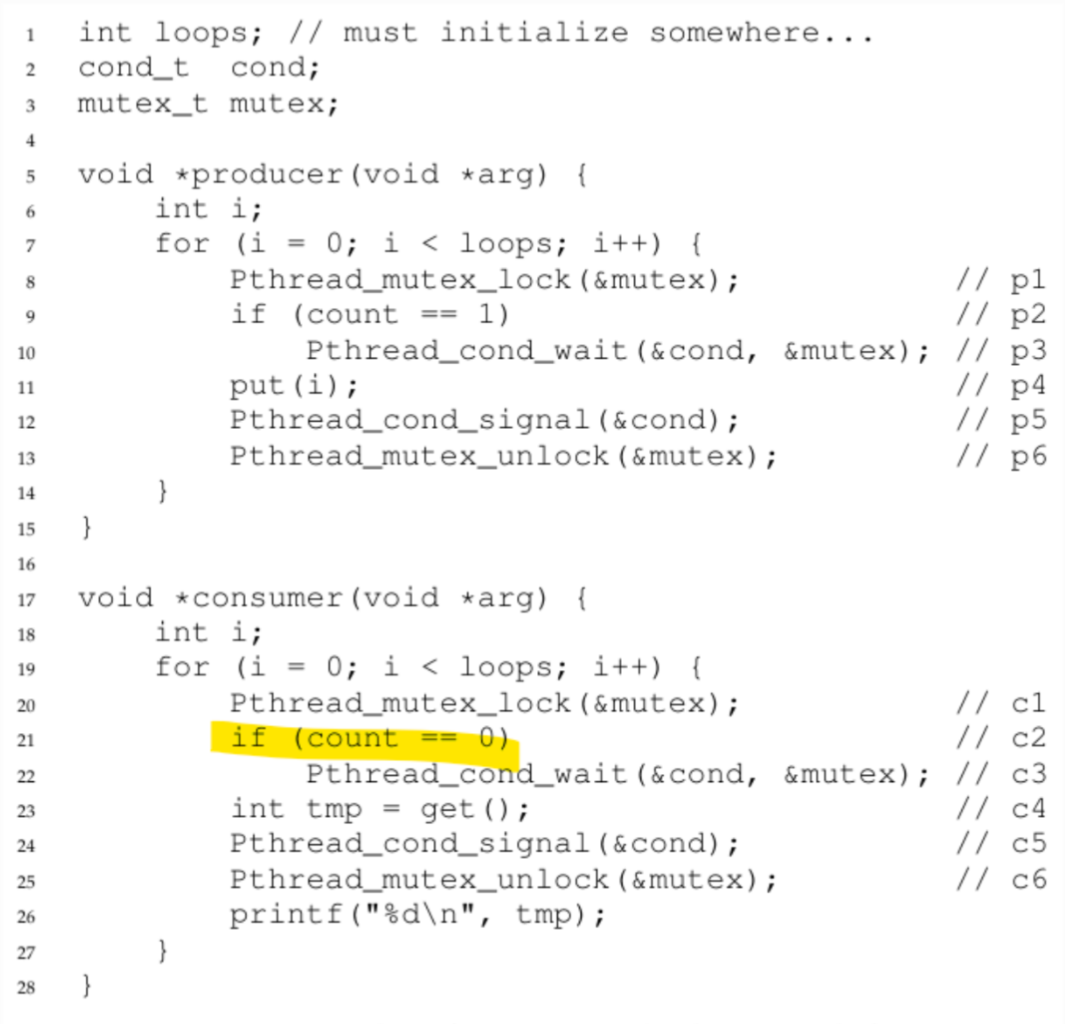
Always hold the lock while calling signal or wait.

Producer/Consumer (bounded-buffer) problem

- producers : generate data items & place them in a buffer
- consumers : grab said items from the buffer & consume them

bounded buffer (in RAM) is a shared resource,
we must require synchronized access to it, avoid a race condition arise.

Always use While to check condition. not If



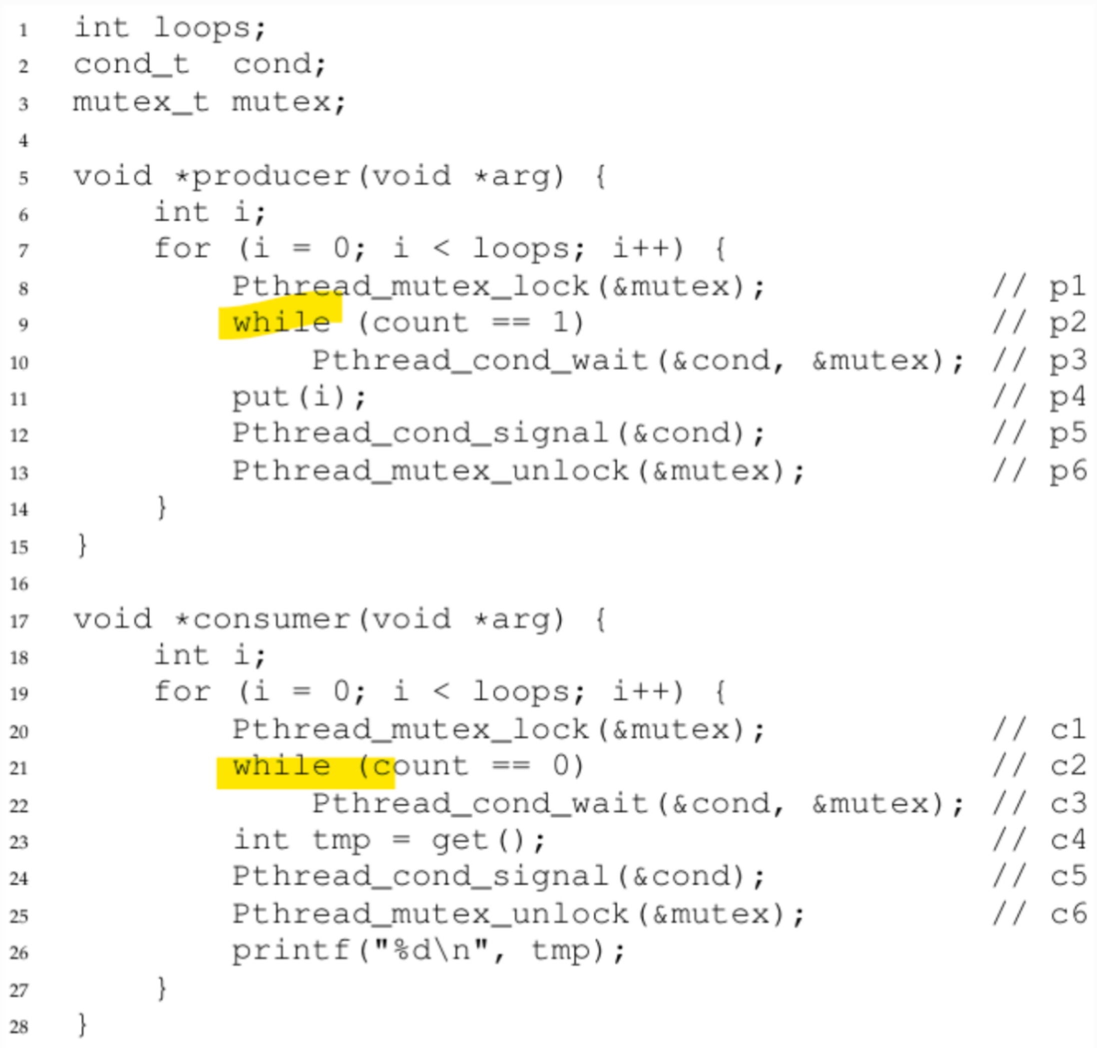
```
int loops; // must initialize somewhere...
cond_t  cond;
mutex_t mutex;

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

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

Figure 30.8: **Producer/Consumer: Single CV And If Statement**

wait if

1. consumerA get into if(count==0) and sleep.
2. producer write. wake consumerA
3. consumerB consume
4. consumerA try get() => but nothing to get.

use while => recheck the condition.



```c
int loops;
cond_t  cond;
mutex_t mutex;

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
    }
}

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);
    }
}
```

Figure 30.10: **Producer/Consumer: Single CV And While**

Mesa semantics

-> Signaling a thread only wakes them up. = just a hint that the state has changed/ no guarantee that when the woken thread runs

- virtually every system everr built employs Mesa semantics

Hoare semantics

provides a stronger guarantee that the woken thread will run immediately upon being woken

- harder to build

signal -> wake only one thread : which one should wake up?

what if consumerA wakes up consumerB

1. consumerA => sleep
2. consumerB waken => sleep
3. producer write and sleep

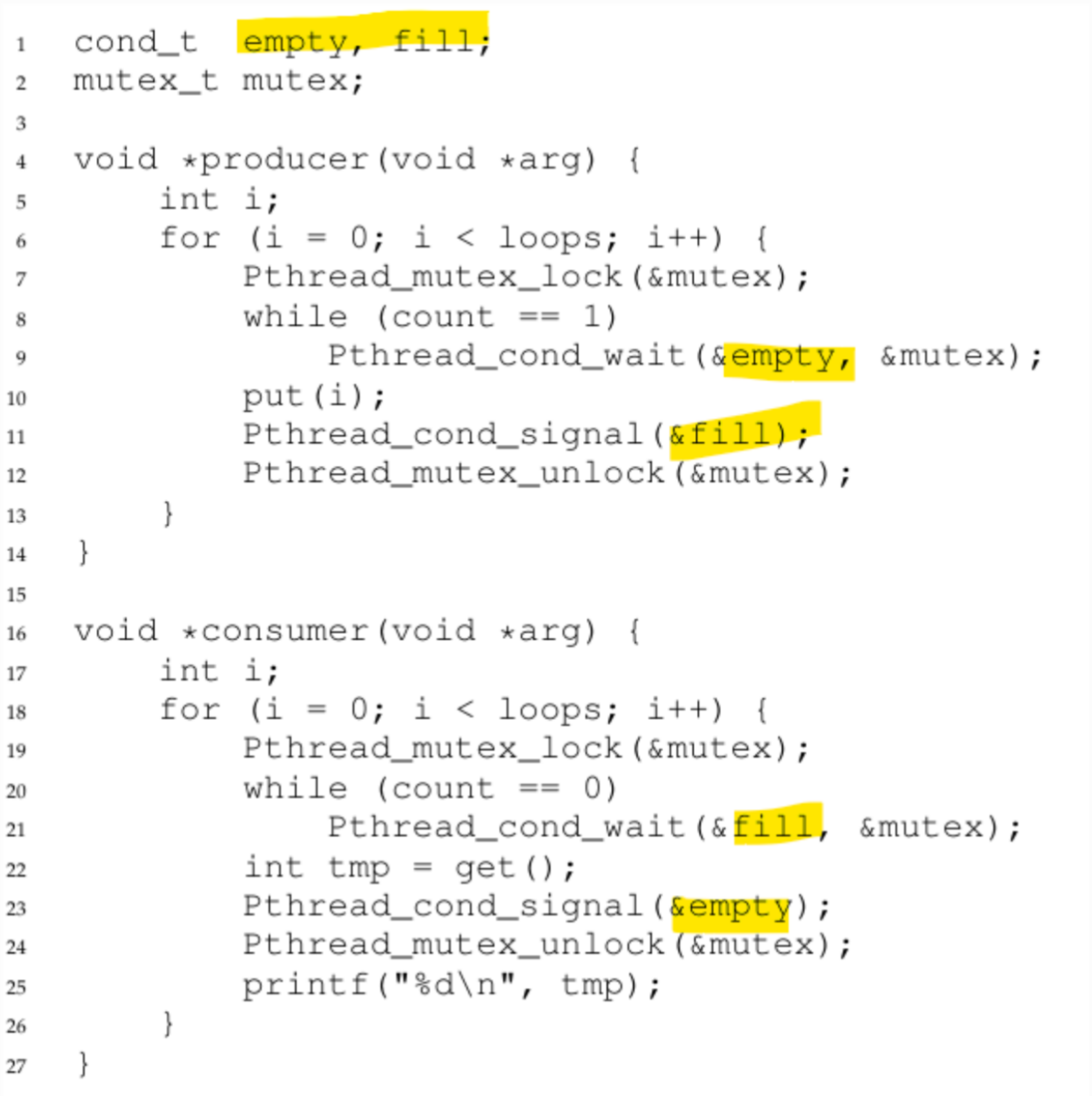
=> sleep forever altogether

=>> a consumer should not wake other consumers, vice-versa

The Single Buffer Producer/Consumer Solution

use 2 condition variables

in order to signal which type of thread should wake up when the state of the system changes.



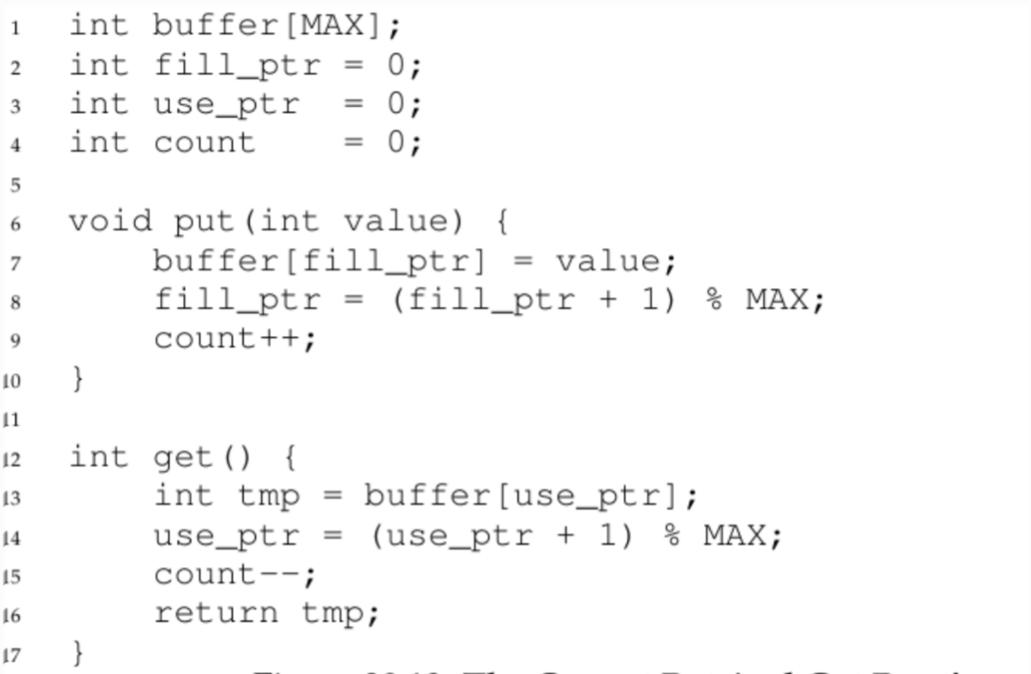
```
cond_t  empty, fill;
mutex_t mutex;

void *producer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 1)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal(&fill);
        Pthread_mutex_unlock(&mutex);
    }
}

void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0)
            Pthread_cond_wait(&fill, &mutex);
        int tmp = get();
        Pthread_cond_signal(&empty);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```

Figure 30.12: **Producer/Consumer: Two CVs And While**

Add more concurrency & efficiency : add more buffer slots

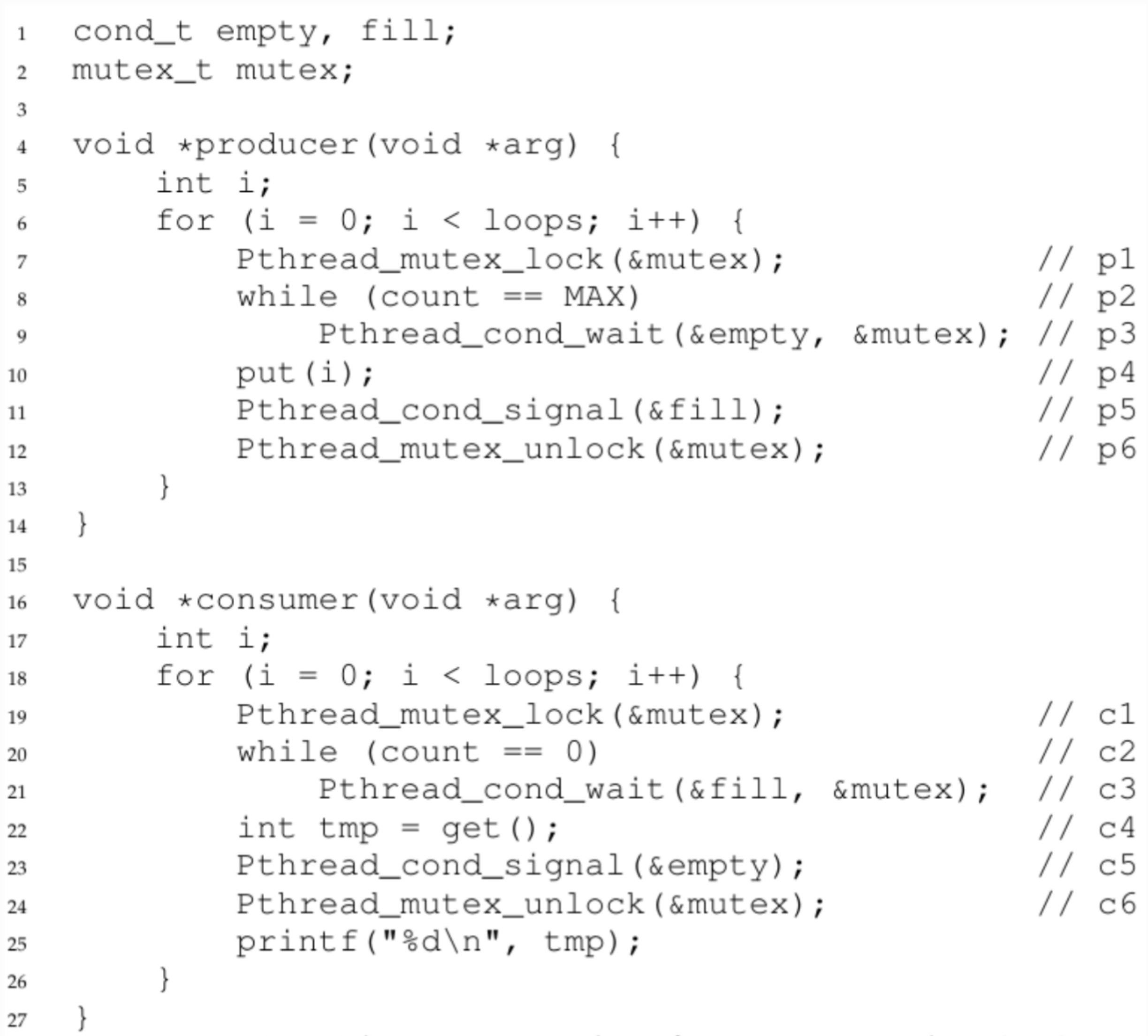
multiple values can be produced & consumed before sleeping

- reduces context switches / increase concurrency



```
int buffer[MAX];
int fill_ptr = 0;
int use_ptr  = 0;
int count    = 0;

void put(int value) {
    buffer[fill_ptr] = value;
    fill_ptr = (fill_ptr + 1) % MAX;
    count++;
}

int get() {
    int tmp = buffer[use_ptr];
    use_ptr = (use_ptr + 1) % MAX;
    count--;
    return tmp;
}
```

Figure 30.13: **The Correct Put And Get Routines**



```
cond_t empty, fill;
mutex_t mutex;

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

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

Figure 30.14: **The Correct Producer/Consumer Synchronization**

Covering Conditions

- Thread 1 : allocate(100)
- Thread 2 : allocate(10)
- Thread 3 : free (50)

which thread the thread 3 should wakes up? what if thread 1?
thread 1: sleep forever.

Solution
replace the pthread_cond_signal()
-> pthread_cond_broadcast() : wakes up all waiting threads

=> guarantee that any threads that should be woken are.
/ downside : can be a negative performance impact, needlessly wake up many other waiting threads that shouldn't be awake.
=> those threads will simply wake up, re-check the condition, and then go immediately back to sleep.

=> **Covering condition**
covers all the cases where a thread needs to wake up (conservatively)
the cost, too many threads might be woken

Why don't we use this approach in the producer/consumer problem with only a single condition variable?
: a better solution was available (use 2 condition variables)

If you find your program only works when you change your signals to broadcasts (but you don't think it should need to), you probably have a bug. fix it!

But in the memory allocator, broadcast may be the most straightforward solution available.
WHY? whether possible to categorize

- in provider/consumer situation => possible to categorization (provider vs. consumer). only 2 option.
- in memory allocator, even same allocate inst, too big one is not available / freeing something might change the situation