CSCI3150 Introduction to Operating Systems

Lecture 6: Synchronization 2: 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

```c
void *child(void *arg) {
    printf("child\n");
    // XXX how to indicate we are done?
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    pthread_create(&c, NULL, child, NULL); // create child
    // XXX how to wait for child?
    printf("parent: end\n");
    return 0;
}
```

What we would like to see here is:

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

Parent waiting for child: Spin-based Approach

```c
volatile int done = 0;

void *child(void *arg) {
    printf("child\n");
    done = 1;
    return NULL;
}

int main(int argc, char *argv[]) {
    printf("parent: begin\n");
    pthread_t c;
    pthread_create(&c, NULL, child, NULL); // create child
    while (done == 0)
        ; // spin
    printf("parent: end\n");
    return 0;
}
```

- This is hugely <u>inefficient</u> as the parent spins and **wastes CPU time**.

How to wait for a condition

- Condition variable
  - Queue of threads
  - **Waiting** on the condition
    - <u>An explicit queue</u> 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 it state*, can wake AT LEAST one of those waiting threads and allow them to continue.
    - The wakeup order is unspecified, though FIFO may be more commonly used

How to wait for a condition

- Three in a package
  - A condition variable
  - A state variable
  - A lock to protect the shared 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 releases the lock and puts the calling thread to sleep.
    - When the thread wakes up, it must re-acquire the lock before returning to the caller.

Parent waiting for child: Use a condition variable

```
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);
}
```

Parent waiting for Child: Use a condition variable

(cont.)

```
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;
}
```

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.

Why do we need the state variable?

```
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);
}
```

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

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

Another poor implementation

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

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

- 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 <u>interrupted</u> 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 (bounded buffer) problem

- **Producer**
  - Produces data items
  - Wishes to place data items in a buffer
- **Consumer**
  - Takes data items out of the buffer; consumes them
- Example: Multi-threaded web server
  - A producer puts HTTP requests into 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 (word count).
    - Between them is an in-kernel bounded buffer.
  - Bounded buffer is a shared resource → **Synchronized access** is required.

The Put and Get routines (Version 1)

```
int buffer;
int count = 0;    // initially, empty

void put(int value) {
        assert(count == 0); // assumes buffer is empty
        count = 1;
        buffer = value;
}

int get() {
        assert(count == 1);
        count = 0;
        return buffer;
}
```

- 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)

```
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);
        }
}
```

- **Producer** puts an integer into the shared buffer a 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`

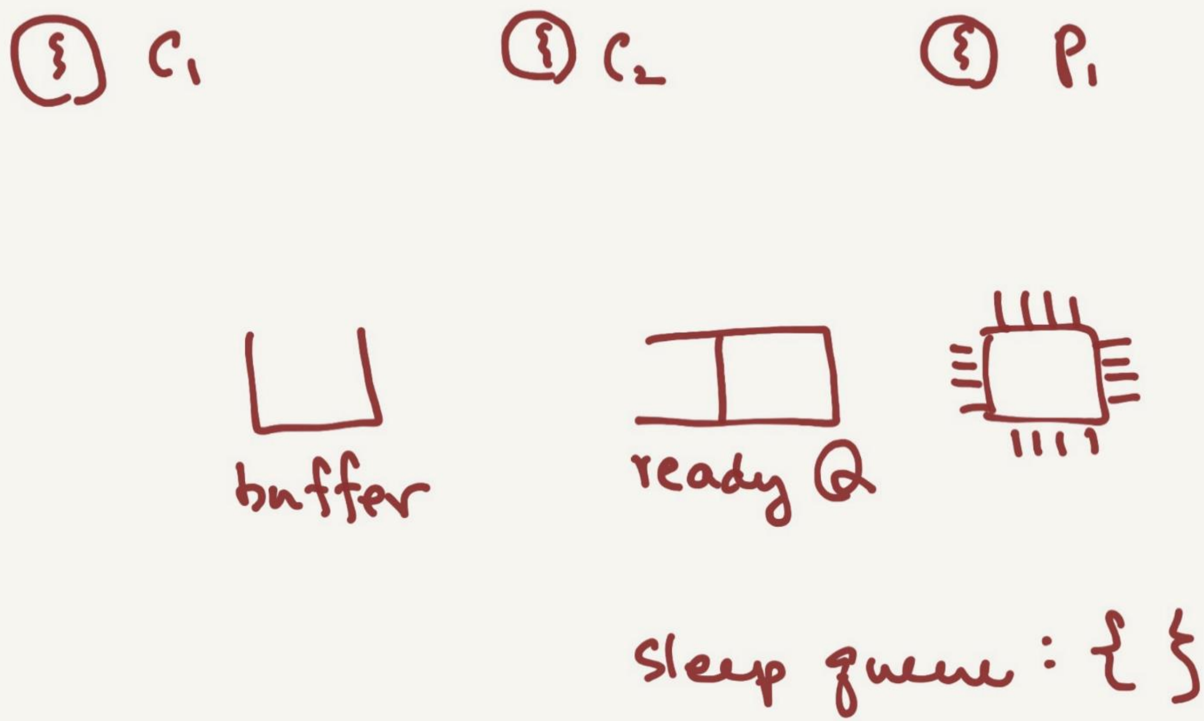
```
cond_t cond;
mutex_t mutex;

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
    }
}
```

Producer/Consumer: Single CV and If Statement

```
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);
    }
}
```

- 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?


C1
C2
P1
buffer
ready Q
Sleep queue : { }

C_1 executes first.

sleep queue : { }

C_1 C_2 P_1

C_1 executes first.

$\{C_1\}$

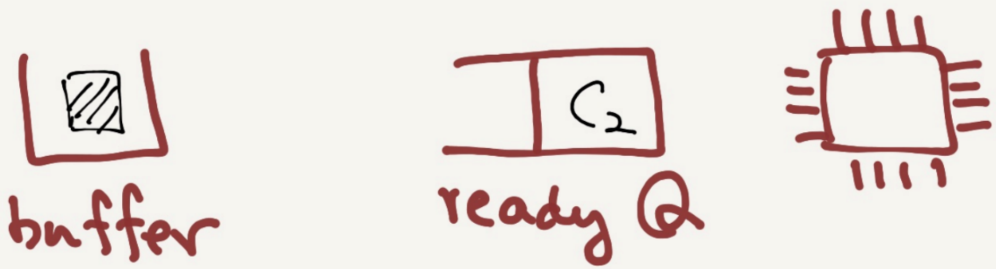
3 C_1 3 C_2 3 P_1

P_1 executes. and send signal.

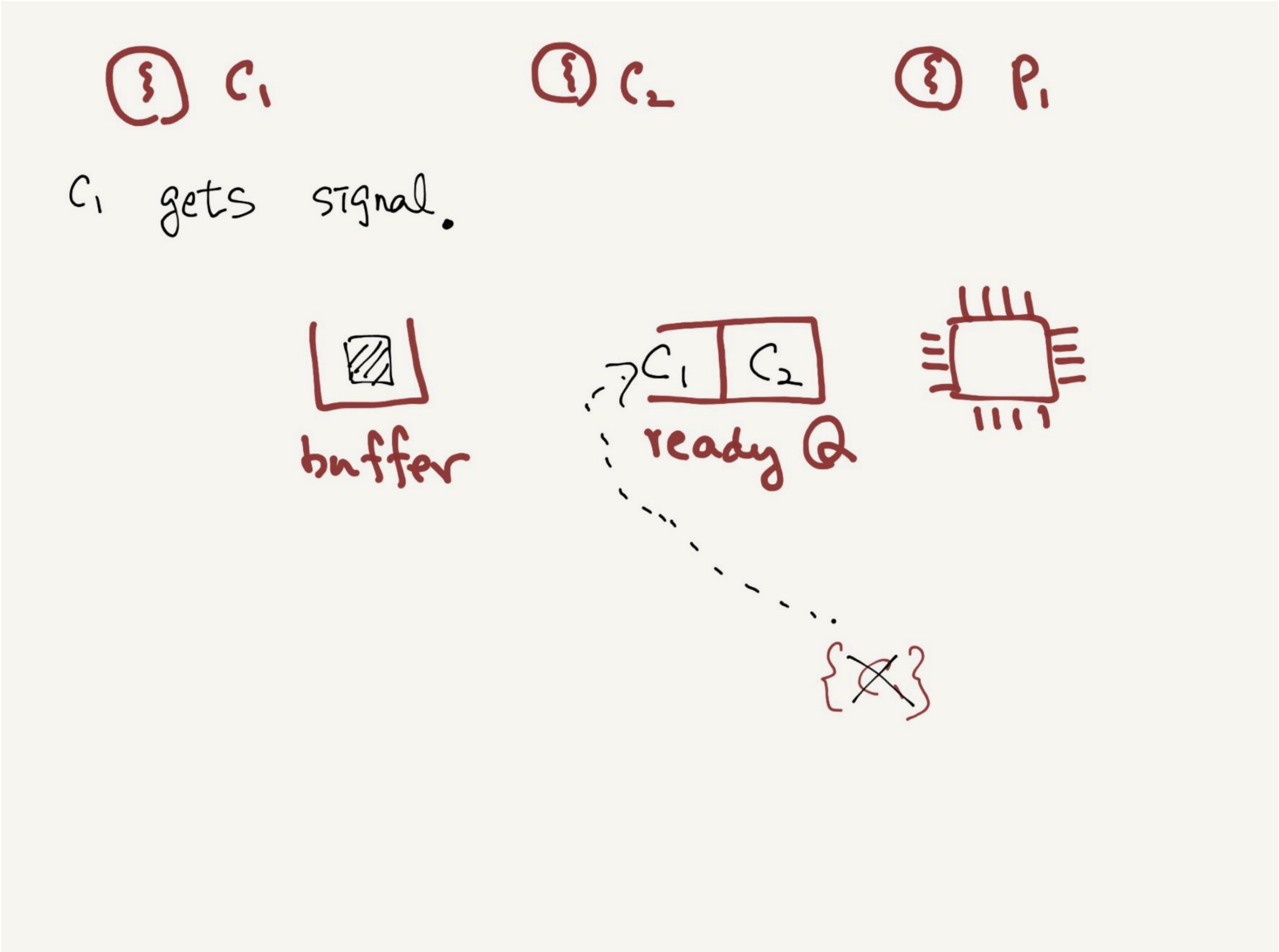
$\{C_1\}$

C_1 C_2 P_1

C_2 arrives.



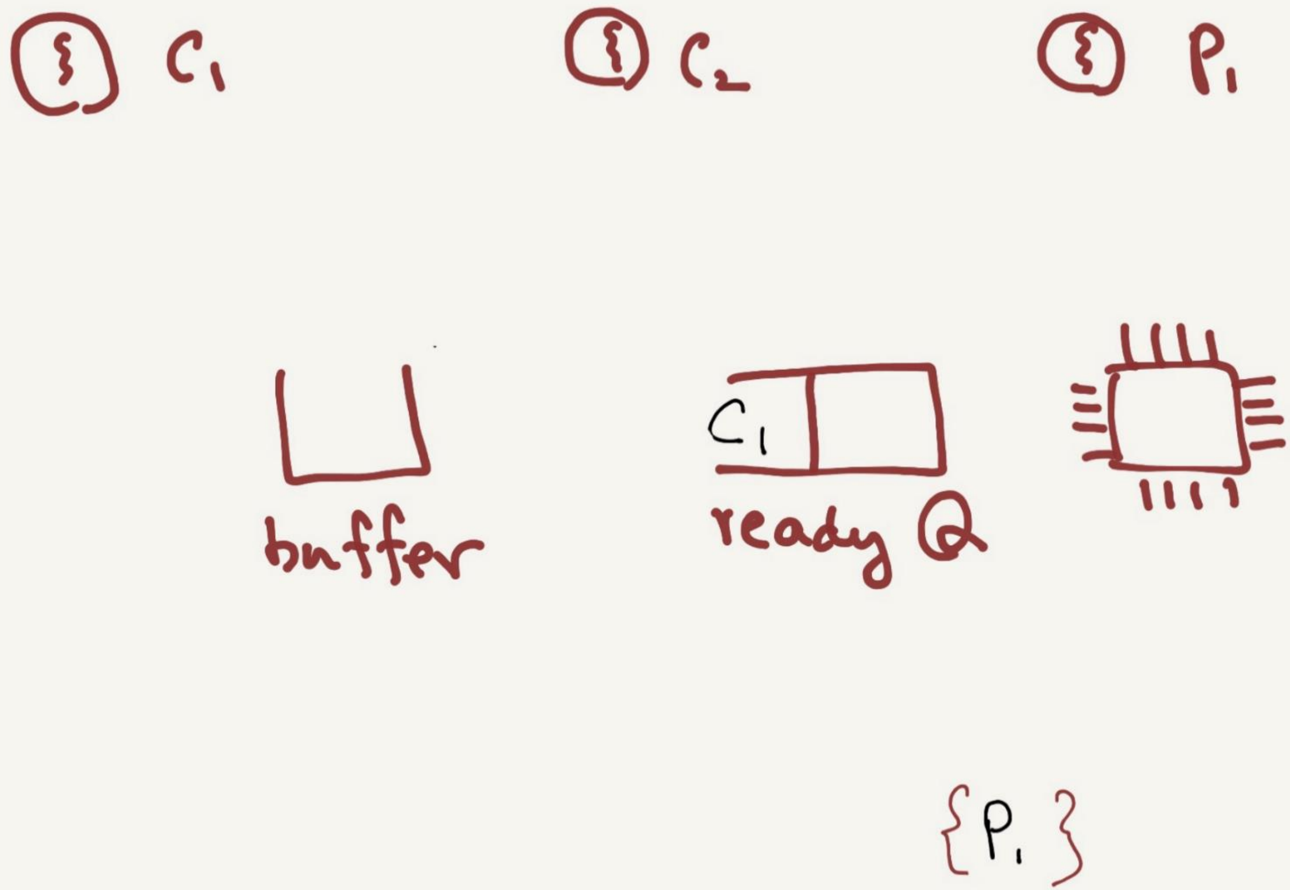
$\{C_1\}$

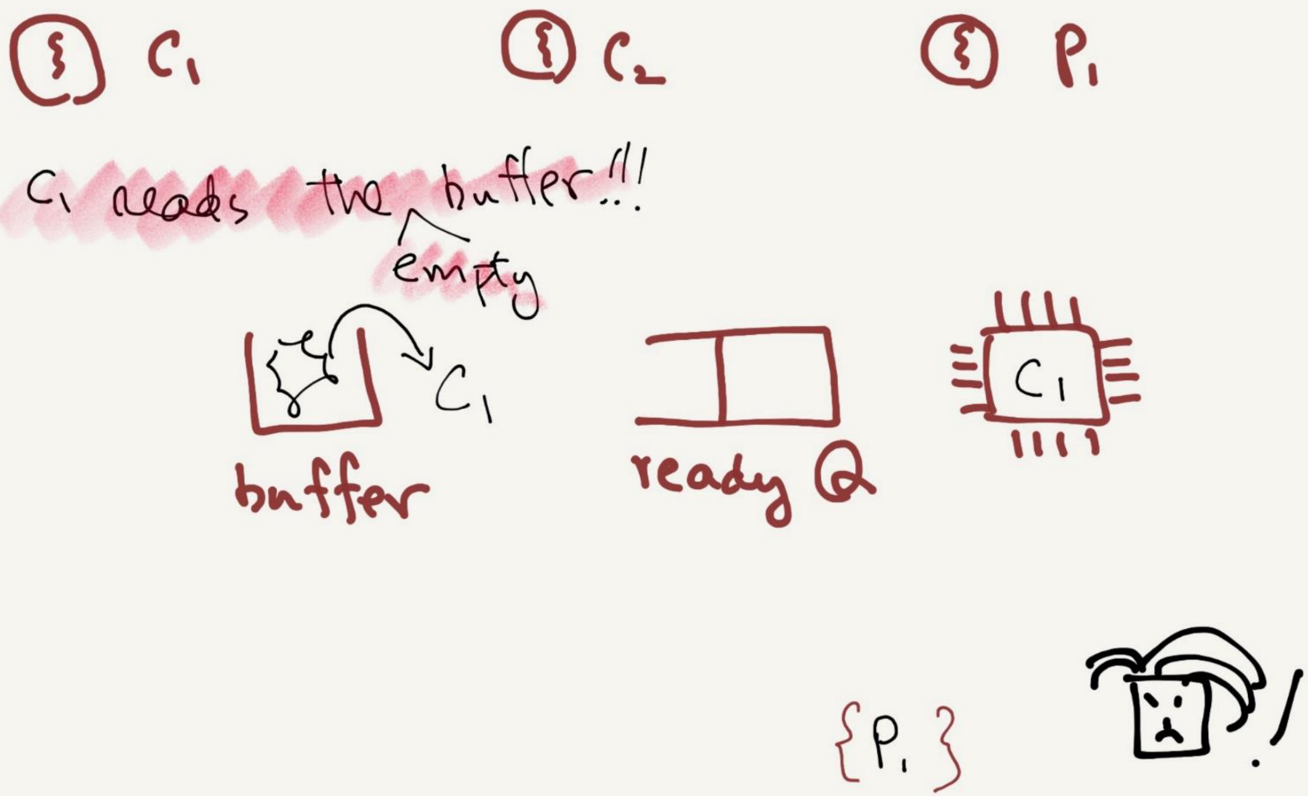


C_1
C_2
P_1
C_1 gets signal.
buffer
C_1
C_2
ready Q

C2 consumes buffer.

{ }


C1
C2
P1
C1
buffer
ready Q
{P1}



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

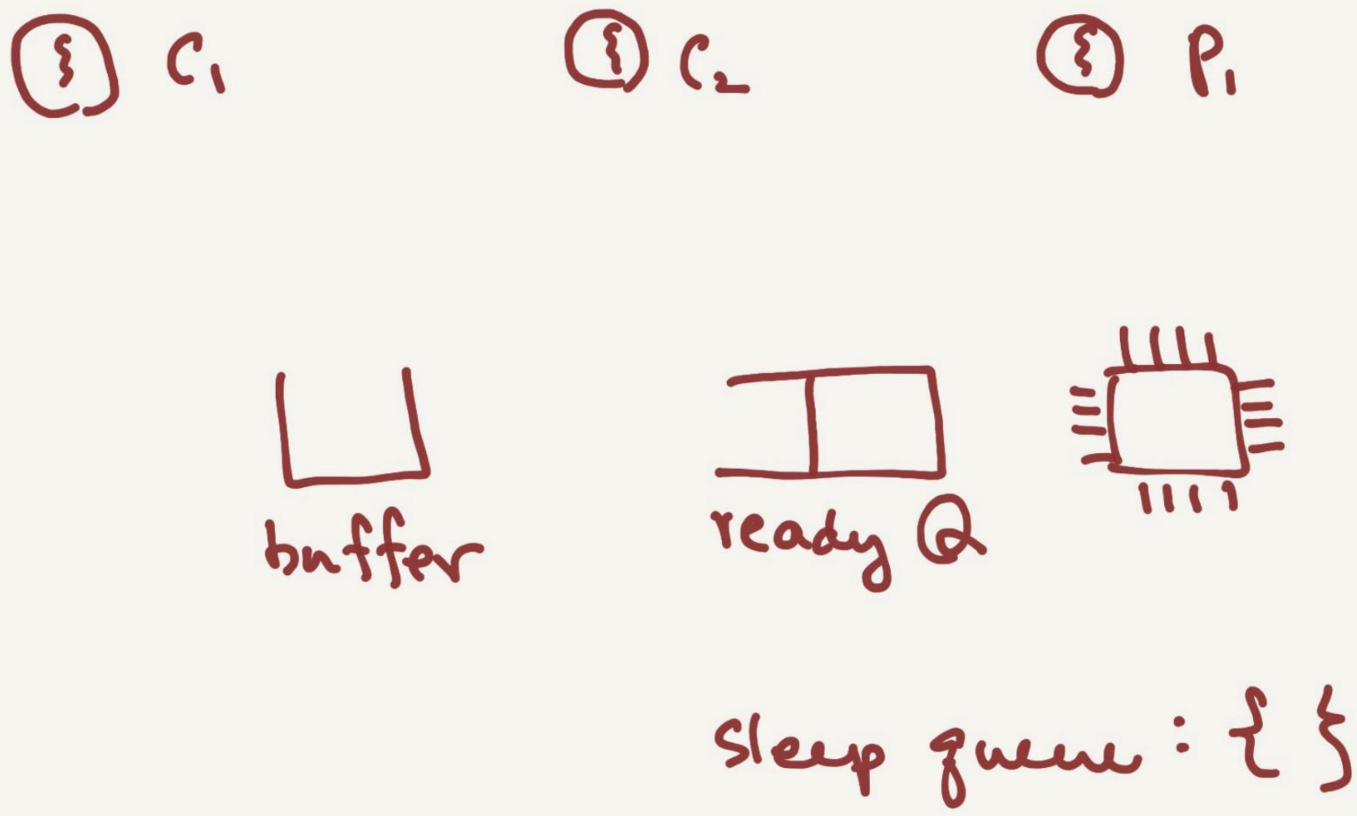
- Consumer T_{c1} wakes up and re-checks the state of the shared variable.
  - If the buffer is empty, the consumer simply goes back to sleep.

```
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
    }
}

```

Producer/Consumer: Single CV and While

```
(Cont.)
    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);
        }
    }
```

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


C1
C2
P1
buffer
ready Q
Sleep queue : { }

C_1 C_2 P_1

C_1 executes first.

Sleep queue: { }

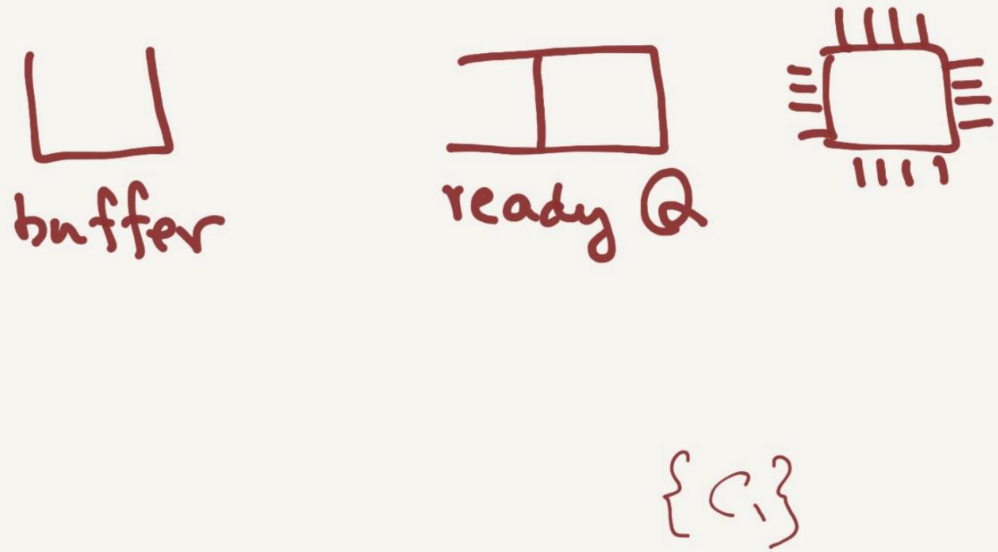
C_1 C_2 P_1

C_1 executes first.

$\{C_1\}$

C_1 C_2 P_1

C_2 executes next.



3 C_1 3 C_2 3 P_1

C_2 executes next

Sleep queue : { C_1, C_2 }

C_1 C_2 P_1

P_1 executes. and send signal.

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

C_1 C_2 P_1

C_1 gets signal, executes and consumes buffer.

sleep queue : {P_1, C_2}

C_1 C_2 P_1

C_1 sends signal.

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

C_1 C_2 P_1

C_2 gets signal!

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

C_1 C_2 P_1

C_2 gets executed. → finds buffer empty. → 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
…	…	…	…	…	…	…	(*cont.*)
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`.

```
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);
    }
}

```

The single Buffer Producer/Consumer Solution

```
(Cont.)
    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);
        }
    }
```

The Final Producer/Consumer Solution

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

```
int buffer[MAX];
int fill = 0;
int use = 0;
int count = 0;

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

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

The Final Put and Get Routines

The Final Producer/Consumer Solution (Cont.)

```
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
    }
}
```

The Final Producer/Consumer Solution (Cont.)

```
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);
    }
}
```

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.