Operating Systems

Lecture 6

Condition Variables

Condition Variables

There are many cases where a thread wishes to <u>check</u> 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

```
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

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 it 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 <u>mutex</u> 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

KOREA UNIVERSITY

```
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.

The importance of the state variable `done`

```
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*.

- 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.

Producer

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

Consumer

- Get data items out of the buffer to consume them in some way

Example: Multi-threaded web server

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

The Put and Get Routines (Version 1)

```c
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;
}
```

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 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`

```
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

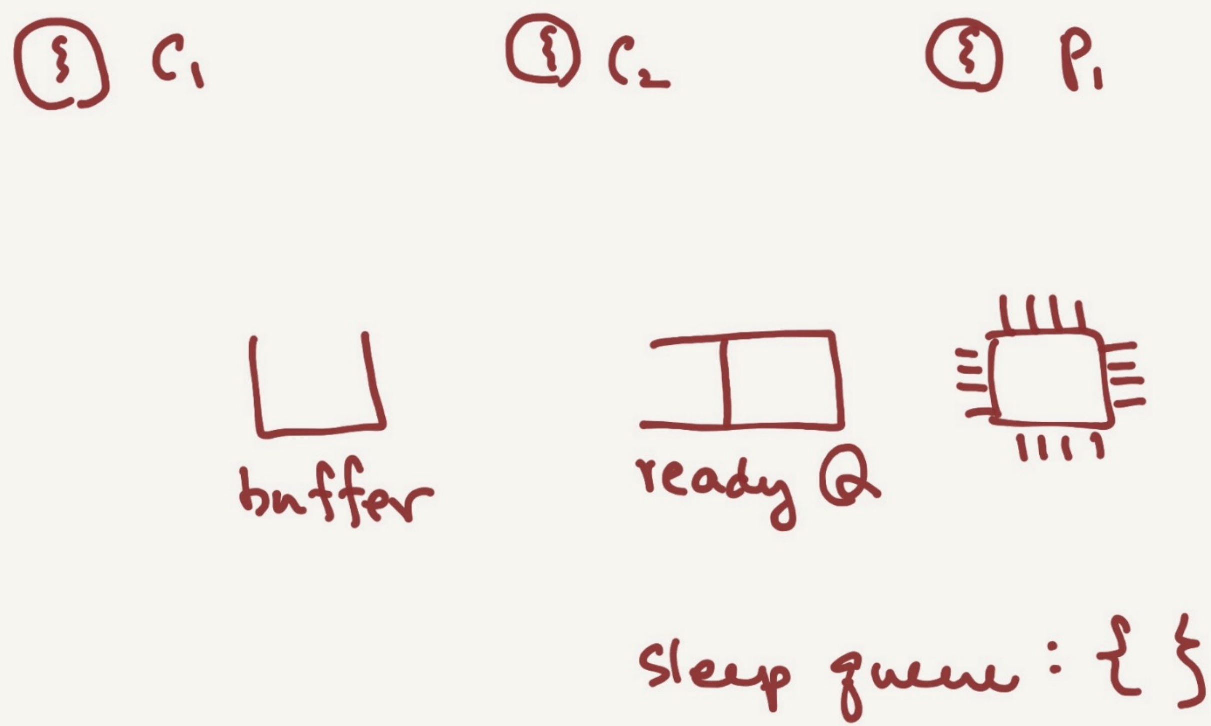
```
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);                    // c1
        if (count == 0)                                // c2
           Pthread_cond_wait(&cond, &mutex);           // c3
        int tmp = get();                               // 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 C_2 P_1

C_1 executes first.

Sleep queue : { }

C_1 C_2 P_1

C_1 executes first.

$\{C_1\}$

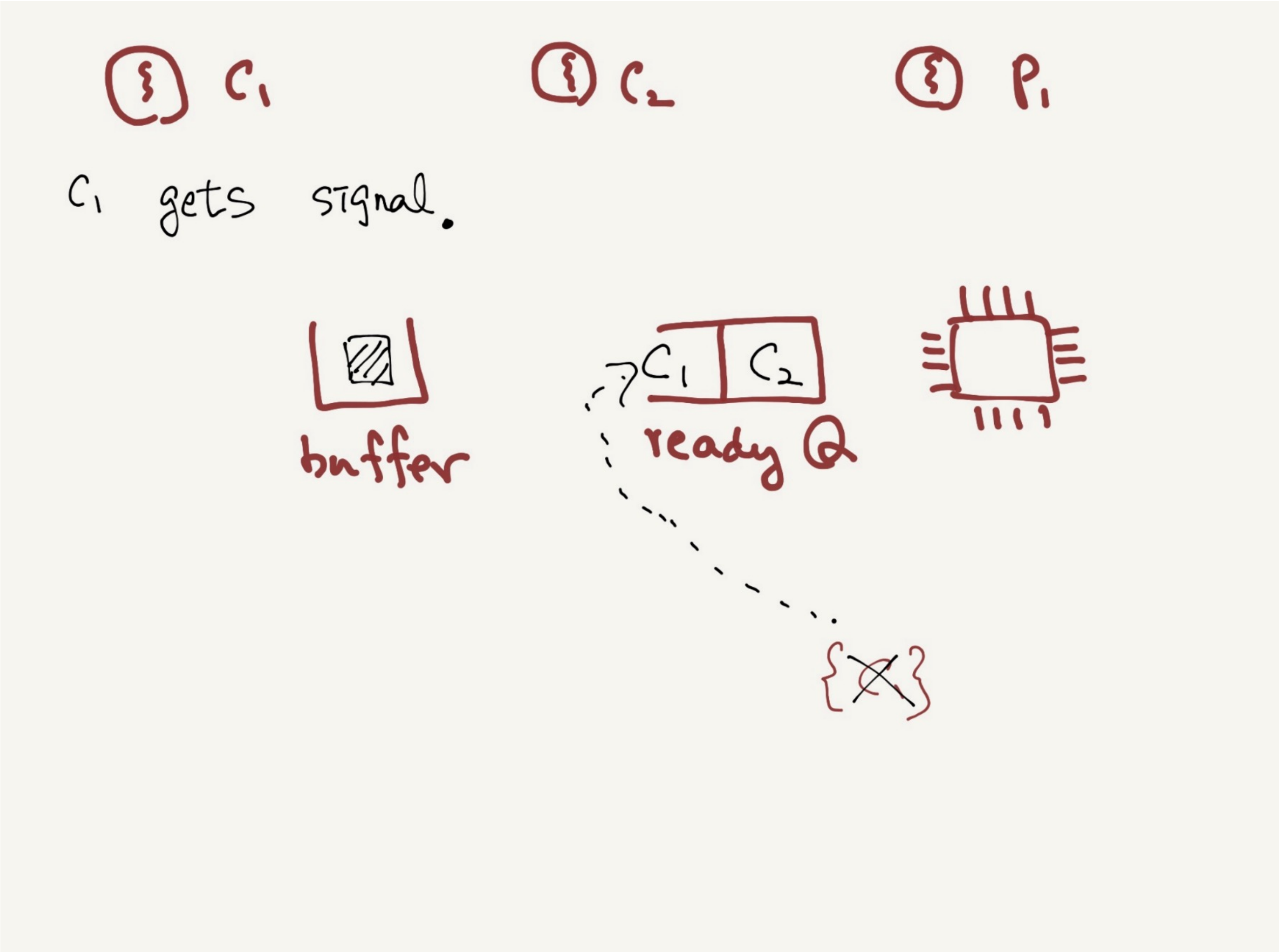
P_1 executes and send signal.

$\{C_1\}$

C_1 C_2 P_1

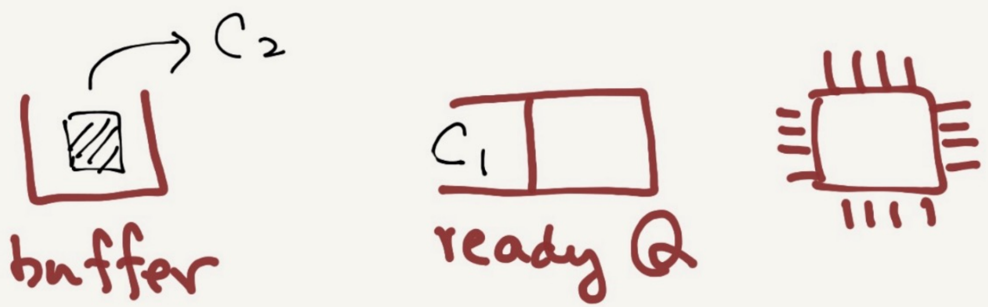
C_2 arrives.

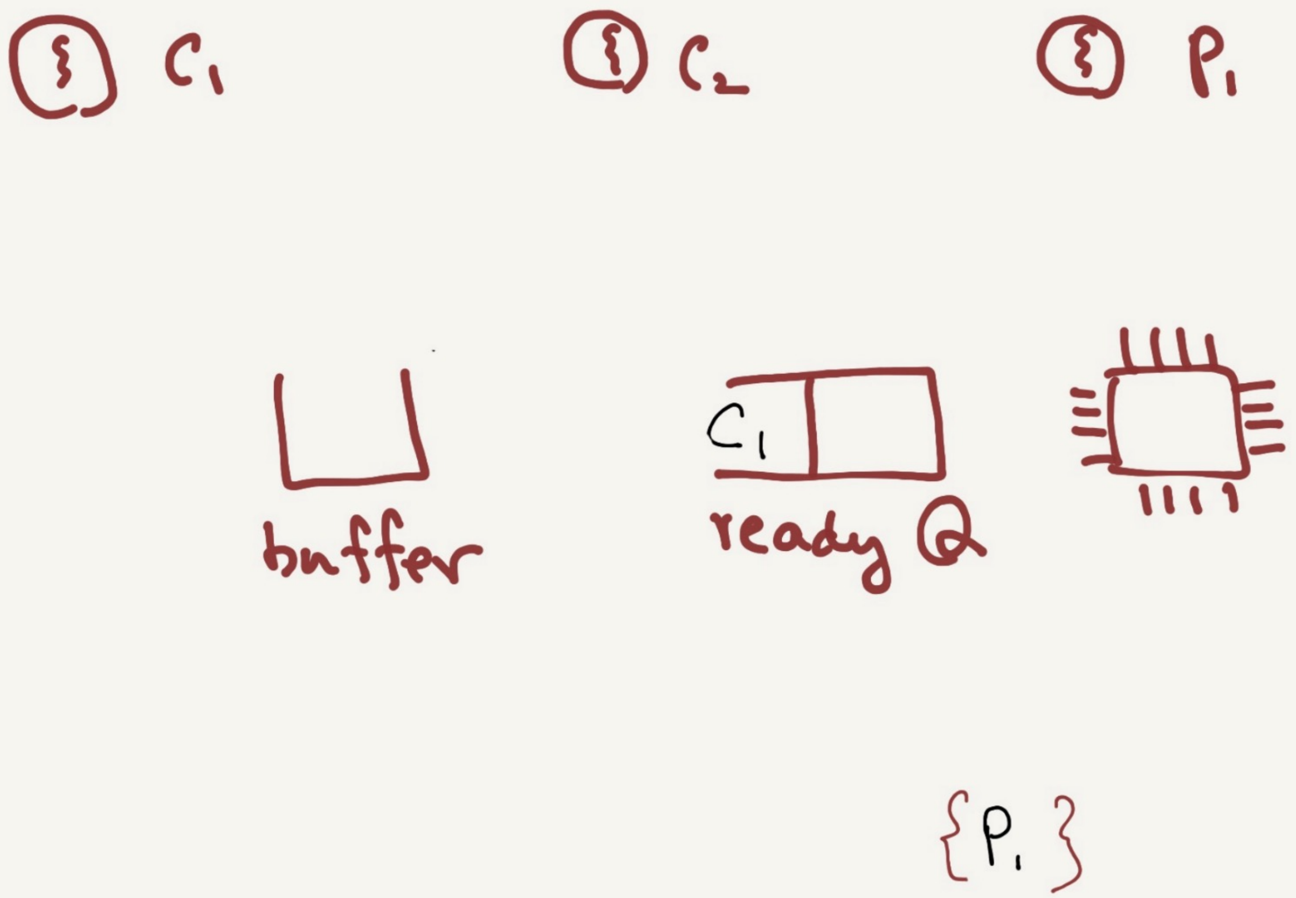
$\{C_1\}$

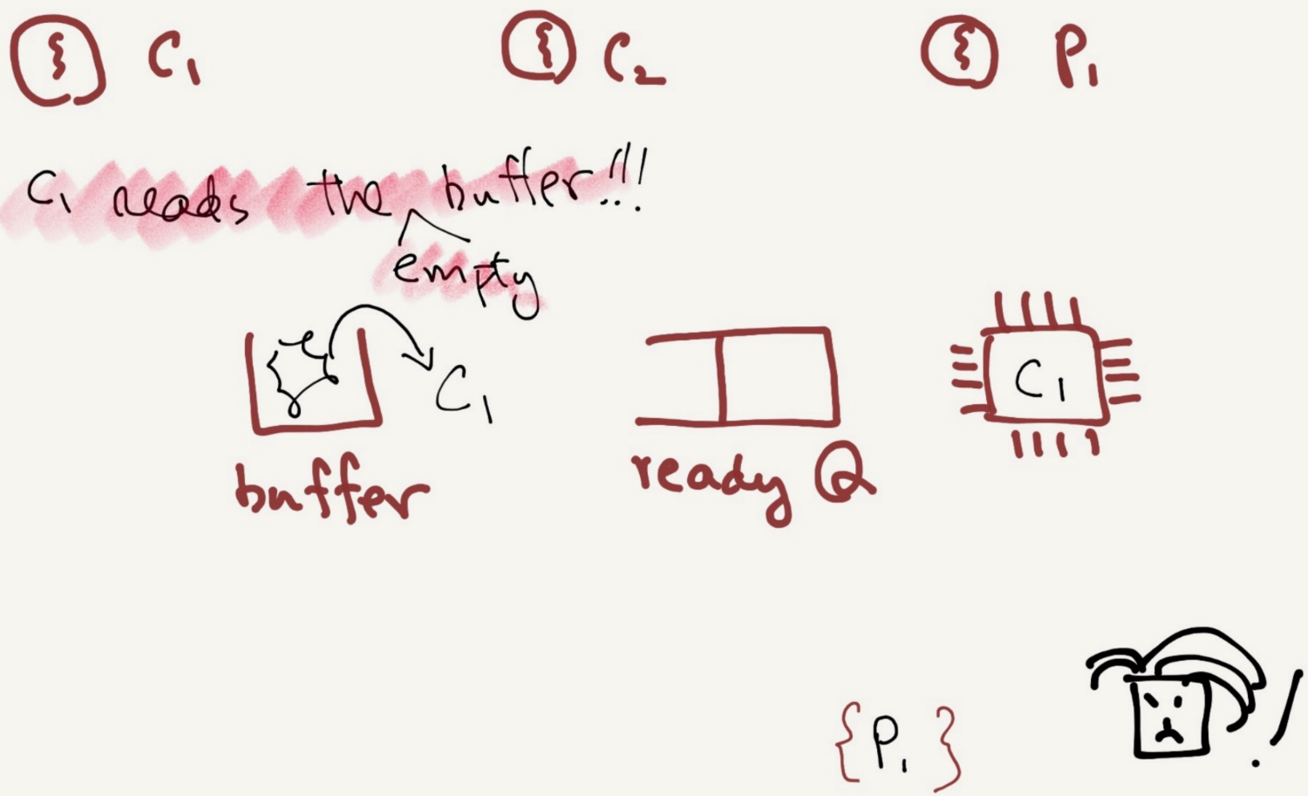

C1
C2
P1
C1 gets signal.
buffer
C1
C2
ready Q

C_1 C_2 P_1

C_2 consumes buffer.




C_1
C_2
P_1
C_1
buffer
ready Q
$\{P_1\}$



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

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.

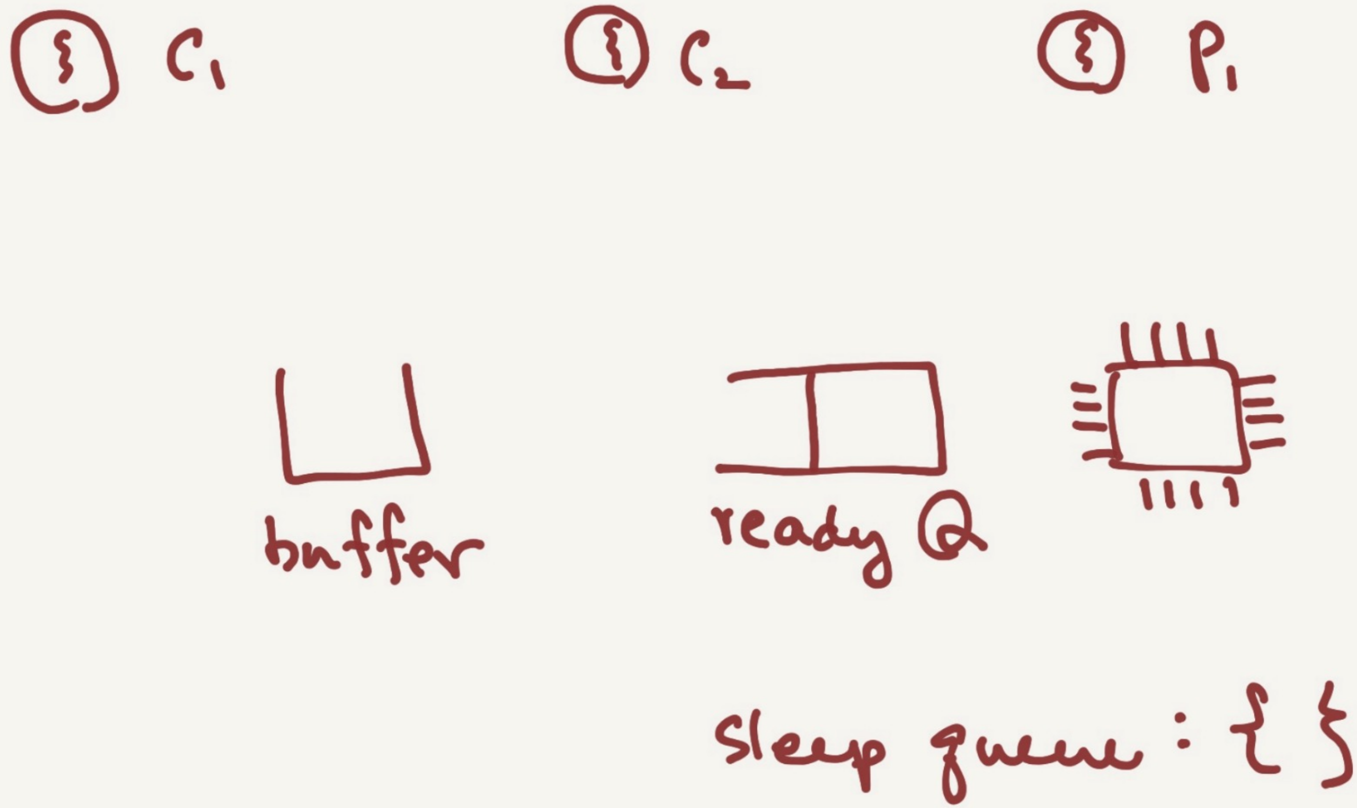
```
cond_t cond;
mutex_t mutex;

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 executes first.

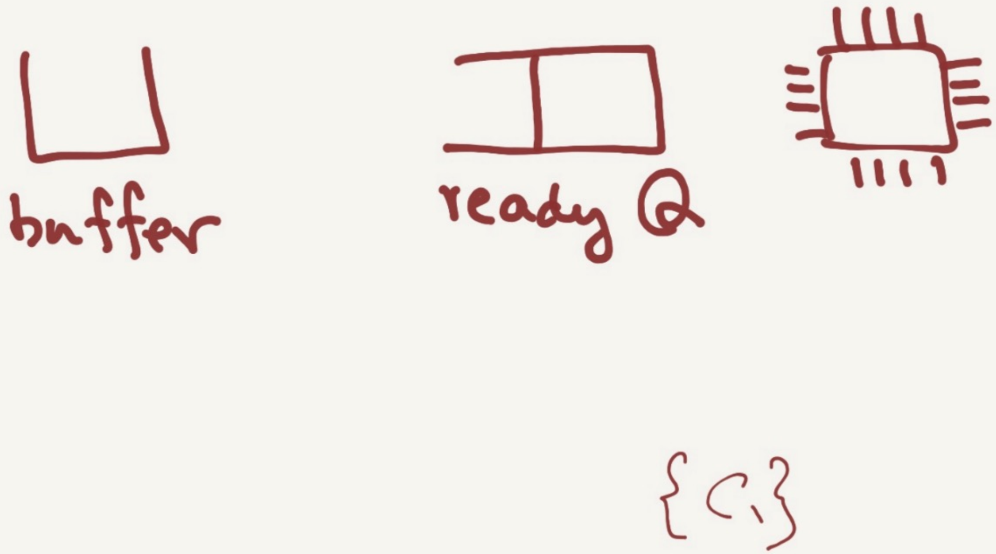
Sleep queue : { }

C_1 C_2 P_1

C_1 executes first.

$\{C_1\}$

C_2 executes next.



C_1 C_2 P_1

C_2 executes next

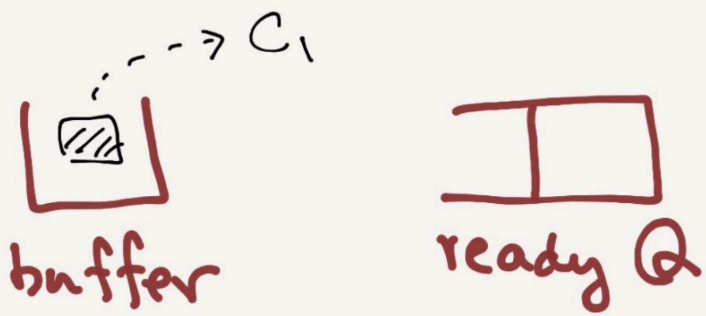
sleep queue : $\{C_1, C_2\}$

ⓢ C_1 ⓢ C_2 ⓢ P_1

P_1 executes. and send signal.

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

C_1 gets signal, executes and consumes buffer.

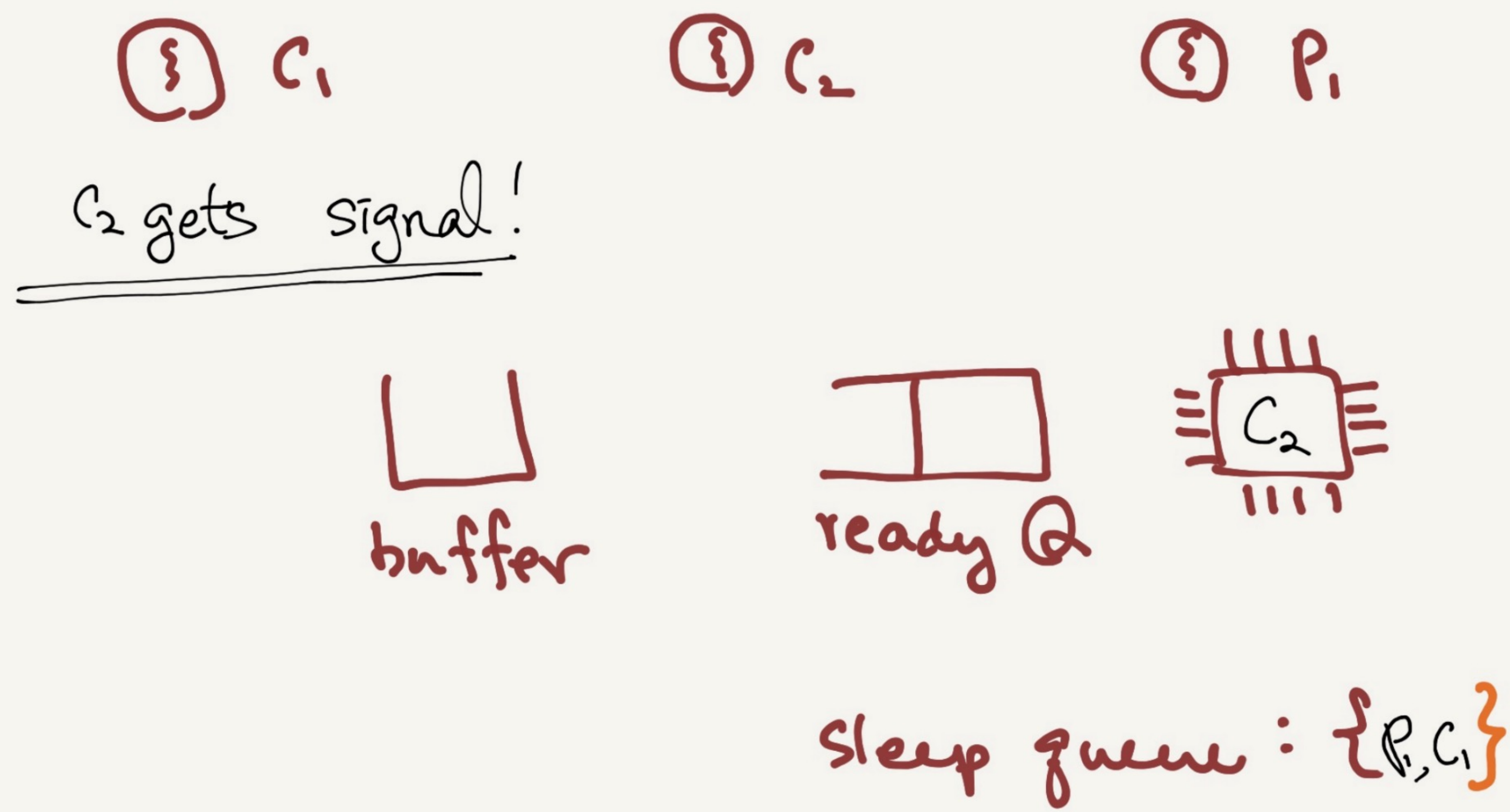


sleep queue : $\{P_1, C_2\}$

C_1 C_2 P_1

C_1 sends signal.

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


C1
C2
P1
C2 gets signal!
buffer
ready Q
C2
Sleep queue : {P1, C1}

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!

The single Buffer Producer/Consumer Solution

KOREA UNIVERSITY

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);
    }
}
```

More **concurrency** and **efficiency** → Add more buffer slots.

Allow concurrent production or consuming to take place.

```
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

```
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

```
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.

The END