Conditional 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 (Cont.)

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
  - **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 the state*, can wake one of those waiting threads and allow them to continue

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: Using 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: Using 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: Using a Condition Variable

- **Parent:**
  - Create the child thread and continues running itself
  - Call into `thr_join()` to wait for the child thread to complete
    - Acquire the lock
    - Check if the child is done
    - Put itself to sleep by calling `wait()`
    - Release the lock
- **Child:**
  - Print the message “child”
  - Call `thr_exit()` to wake the parent thread
    - Grab the lock
    - Set the state variable `done`
    - Signal 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 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

```
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 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

Put and Get Routines (Version 1)

```
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)

```c
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`

```c
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
```

Producer/Consumer: Single CV and If Statement

```
            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?

Thread Trace: Broken Solution (Version 1)

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**

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*)

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}**

Thread Trace: Broken Solution (Version 2)

T_{c1}	State	T_{c2}	State	T_p	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

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);
    }
}
```

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);
    }
}
```

Final Producer/Consumer Solution

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

```c
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

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
    }
}

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
```

Final Producer/Consumer Solution

```
(Cont.)
                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

Covering Conditions

- Assume there are zero bytes free
  - Thread T_a calls `allocate(100)`
  - Thread T_b calls `allocate(10)`
  - Both T_a and T_b wait on the condition and go to sleep
  - Thread T_c calls `free(50)`

Which waiting thread should be woken up?

Covering Conditions

```c
// how many bytes of the heap are free?
int bytesLeft = MAX_HEAP_SIZE;

// need lock and condition too
cond_t c;
mutex_t m;

void *
allocate(int size) {
    Pthread_mutex_lock(&m);
    while (bytesLeft < size)
        Pthread_cond_wait(&c, &m);
    void *ptr = ...;                // get mem from heap
    bytesLeft -= size;
    Pthread_mutex_unlock(&m);
    return ptr;
}

void free(void *ptr, int size) {
    Pthread_mutex_lock(&m);
    bytesLeft += size;
    Pthread_cond_signal(&c);        // whom to signal??
    Pthread_mutex_unlock(&m);
}
```

Covering Conditions

- Solution (Suggested by Lampson and Redell)
  - Replace `pthread_cond_signal()` with `pthread_cond_broadcast()`
  - `pthread_cond_broadcast()`
    - Wake up **all waiting threads**
    - Cost: too many threads might be woken
    - Threads that shouldn't be awake will simply wake up, re-check the condition, and then go back to sleep