Condition Variables
Questions

• What is a condition variable for?

• Using a condition variable, how is a thread waken up?
Basic Idea

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;
}

• A parent thread waits for the completion of a child thread.
  ▪ Completion of child thread is a condition.
Basic Idea

```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;
}
```

- Use a shared variable
  - Spin-waiting wastes CPU time
- Put parent to sleep until condition becomes true.
Definitions and Routines

• Condition variable
  – A queue for threads waiting for some condition
  – Another thread changes the condition (signals on the condition) and allows the waiting thread to continue.

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

To prevent race condition, parent locks a mutex before calling `wait()`. `wait()` puts parent thread to sleep and releases the lock. Child runs and calls `thr_exit()` to wake the parent. `wait()` returns and parent continues.
Definitions and Routines

```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);
}
```
Definitions and Routines

First case
- Parent calls `thr_join()` for putting itself to sleep (`wait()` for child)
- Child runs and calls `thr_exit()` to wake the parent.
- `wait()` returns and parent continues.

```c
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;
}
```
Definitions and Routines

Second case

• Child runs immediately upon creation.
• Calls signal() but it returns immediately.
• Parent calls join() but does not wait.
Importance of done Variable

- Child runs upon creation and calls thr_exit().
  - No sleeping thread to wake. Child is done.
- Parent calls wait() but gets stuck (there’s no thread to wake the parent).
Another Poor Implementation

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

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

- Parent calls `thr_join()`.
  - Attempts to sleep
- Before parent sleeps, an interrupt occurs.
  - Child runs and calls `thr_exit()` but there’s no sleep thread to wake.
- Parent runs again but gets stuck after calling `wait()`.
The Producer/Consumer Problem

• Producer and consumer threads
  – Producers generate data and place them in a buffer.
  – Consumers grab data and consume them.

• Pipe the output of one program into another
  – E.g., grep foo file.txt | wc
  – grep process is producer and wc process is consumer.
  – Bounder buffer is maintained by kernel.
The Producer/Consumer Problem

• Buffer is a shared resource.
  – Synchronize access to it

• Assume that the shared buffer is an integer

• Two routines to call to put/get a value into/from the shared buffer
The Producer/Consumer Problem

```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;
}
```

- Conditions
  - Put data only if buffer is empty
  - Get data only if buffer is full
The Producer/Consumer Problem

```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);
    }
}
```

- Two types of threads
  - Producer threads
  - Consumer threads
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);
    }
}
A Broken Solution

• Problem
  – If-statement
  – Two consumers (TC₁ and TC₂) and one producer (TP)
  – TC₁ runs first. TP and TC₂ run subsequently.
  – After TP woke TC₁ and before TC₁ ever ran, the state of the bounded buffer was changed by TC₂.

• When a woken thread runs, there is no guarantee that the state of the buffer is as desired.
While, Not If

```c
void *consumer(void *arg) {
    int i;
    for (i = 0; i < loops; i++) {
        Pthread_mutex_lock(&mutex);
        while (count == 0) {
            Pthread_cond_wait(&cond, &mutex);
        }
        int tmp = get();
        Pthread_cond_signal(&cond);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
}
```
While, Not If

```c
cond_t cond;
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(&cond, &mutex);
        put(i);
        Pthread_cond_signal(&cond);
        Pthread_mutex_unlock(&mutex);
    }
}
```
Directed Signaling

- \(T_{C1}\) and \(T_{C2}\) run. After that, \(T_p\) runs.
- A consumer needs to wake a producer, not another consumer.

- Use two condition variables
  - Specify a type of thread that should wakeup when the state of the buffer changes.
  - E.g., producer waits for the empty condition and signals on the fill condition.
The Single Buffer Producer/Consumer Solution

```c
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

```c
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 Correct Producer/Consumer Solution

```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--;  // Count decremented here
    return tmp;
}
```
The Correct Producer/Consumer Solution

```c
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 Correct Producer/Consumer Solution

```c
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);
    }
}
```
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 = ...;
    bytesLeft -= size;
    Pthread_mutex_unlock(&m);
    return ptr;
}

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

bytesLeft = 0
T_a: allocate(100) → wait
T_b: allocate(10) → wait
T_c: free(50) → signal

`pthread_cond_broadcast()`

*Performance overhead*