Condition Variables

CMPU 334 – Operating Systems
Jason Waterman
Condition Variables

• There are many cases where we wish to have coordination between threads

• 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
  • As we have seen, this is a join()
A Parent Waiting For Its Child

```c
void *child(void *arg) {
    printf("child\n");
    // TODO: 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
    // TODO: 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

This is hugely inefficient as the parent spins and wastes CPU time.

How should a thread wait for a condition?

```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;
}
```
How to wait for a condition

• **Condition variable** – an object used to wait for some condition to be true
  • **Waiting** on the condition variable
    • An explicit queue that threads can put themselves on when some state of execution is not as desired
    • The thread is no longer running, freeing up the CPU to run another thread
  • **Signaling** on the condition variable
    • Some other thread, *when it changes said state*, can wake one of those waiting threads and allow them to continue
Pthread Condition Variables

• Declare condition variable

```c
pthread_cond_t c;
```

• Proper initialization is required

```c
pthread_cond_t c = PTHREAD_COND_INITIALIZER; // Declaration and initialization
or
pthread_cond_init(pthread_cond_t *c, pthread_attr_t *attr) // Initialization with attributes
```

• Operation

```c
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 thread that calls `wait()` is assumed to be holding the mutex lock
  • 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 `wait()` returns

• `signal()` will wake up a thread that is waiting on the condition variable
Parent waiting for Child: Use a condition variable

```c
1 volatile int done = 0;
2 pthread_mutex_t m = PTHREAD_MUTEX_INITIALIZER;
3 pthread_cond_t c = PTHREAD_COND_INITIALIZER;
4
5 void thr_exit() {
6     Pthread_mutex_lock(&m);
7     done = 1;
8     Pthread_cond_signal(&c);
9     Pthread_mutex_unlock(&m);
10 }
11
12 void *child(void *arg) {
13     printf("child\n");
14     thr_exit();
15     return NULL;
16 }
17
18 void thr_join() {
19     Pthread_mutex_lock(&m);
20     while (done == 0)
21         Pthread_cond_wait(&c, &m);
22     Pthread_mutex_unlock(&m);
23 }
```

```c
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```
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

don't

• Can you think of a scenario where we could run into problems?
• Imagine the case where the child runs immediately
  • The child will signal, but there is no thread sleeping on the condition
  • When the parent runs, it will call wait and be stuck
  • No thread will ever wake it, sad panda!

```c
1 void thr_exit() {
2     Pthread_mutex_lock(&m);
3     Pthread_cond_signal(&c);
4     Pthread_mutex_unlock(&m);
5 }
6
7 void thr_join() {
8     Pthread_mutex_lock(&m);
9     Pthread_cond_wait(&c, &m);
10    Pthread_mutex_unlock(&m);
11}
25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```

thr_exit() and thr_join() without variable done
Importance of locks

- Can you find the bug? (assume you don’t need a lock to use signal and wait)
- The issue here is a 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 `pthread_cond_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, sad!

```c
1 volatile int done = 0;
2
3 void thr_exit() {
4     done = 1;
5     Pthread_cond_signal(&c);
6 }

7 void thr_join() {
8     if (done == 0)
9         Pthread_cond_wait(&c);
10 }

25 int main(int argc, char *argv[]) {
26     printf("parent: begin\n");
27     pthread_t p;
28     Pthread_create(&p, NULL, child, NULL);
29     thr_join();
30     printf("parent: end\n");
31     return 0;
32 }
```
The Producer / Consumer (Bounded Buffer) Problem

• **Producer**
  • Produces data items
  • Wishes to place data items in a buffer

• **Consumer**
  • Grabs data items out of the buffer to **consume** them in some way

• **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
Producer/Consumer (non-working)

- Put -- Only put data into the buffer when count is zero (i.e., when the buffer is empty)
- Get -- Only get data from the buffer when count is one (i.e., when the buffer is full)
- **Producer** -- puts an integer into the shared buffer loops number of times
- **Consumer** -- gets the data out of that shared buffer
- Need synchronization between the producer and consumer

```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;
}
```

```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);
    }
}
```
A single condition variable `cond` and associated lock `mutex`
  • Works if there is one producer and one consumer

What happens if that is not the case (e.g., 2 consumers, 1 producer)?
  • C1 runs and waits, P1 puts an item in and signals C1
  • Before C1 gets to run, C2 sneaks in and consumes the item, setting count to 0
  • When C1 runs, no more items left, sad!

Recheck state (in a while loop) upon returning from wait!
Thread Trace: Broken Solution

• 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 up thread runs, the state will still be as desired $\rightarrow$ Mesa semantics
  • Virtually every system ever built employs Mesa semantics

• Hoare semantics provides a stronger guarantee that the woken up thread will run immediately upon being woken
This fixes our previous problem, however, this code still has a bug

- Assume two consumers and one producer
- C1 runs, finds the buffer empty and waits, C2 runs, finds the buffer empty and waits
- P1 runs, produces an item, signals, and waits because buffer is full
- C1 wakes (from P1 signal) and consumes the buffer, signals, and then waits
  - Who gets the signal, P1 or C2?
- C2 wakes, finds the buffer empty and waits – everyone is sleeping, sad!

```c
void *producer(void *arg) {
    int i;
    int loops = (int) arg;
    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);
    }
}

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
", tmp);
    }
}
```
The single Buffer Producer/Consumer Solution

• Use two condition variables and while loops
  • **Producer** threads wait on the condition `empty`, and signals `fill`
  • **Consumer** threads wait on `fill` and signal `empty`

```c
1  cond_t empty, fill;
2  mutex_t mutex;
3
4  void *producer(void *arg) {
5    int i;
6    int loops = (int) arg;
7    for (i = 0; i < loops; i++) {
8      Pthread_mutex_lock(&mutex);
9      while (count == 1)
10         Pthread_cond_wait(&empty, &mutex);
11      put(i);
12      Pthread_cond_signal(&fill);
13      Pthread_mutex_unlock(&mutex);
14    }
15 }
```

```c
16  void *consumer(void *arg) {
17    int i;
18    for (i = 0; i < loops; i++) {
19      Pthread_mutex_lock(&mutex);
20      while (count == 0)
21         Pthread_cond_wait(&fill, &mutex);
22      int tmp = get();
23      Pthread_cond_signal(&empty);
24      Pthread_mutex_unlock(&mutex);
25      printf("%d\n", tmp);
26    }
27 }
```
The 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;
}
```

```c
1 cond_t empty, fill;
2 mutex_t mutex;
3
4 void *producer(void *arg) {
5     for (int i = 0; i < loops; i++) {
6         int loops = (int) arg;
7         Pthread_mutex_lock(&mutex);
8         while (count == MAX)
9             Pthread_cond_wait(&empty, &mutex);
10         put(i);
11         Pthread_cond_signal(&fill);
12         Pthread_mutex_unlock(&mutex);
13     }
14 }
15
16 void *consumer(void *arg) {
17     for (int i = 0; i < loops; i++) {
18         int loops = (int) arg;
19         Pthread_mutex_lock(&mutex);
20         while (count == 0)
21             Pthread_cond_wait(&fill, &mutex);
22         int tmp = get();
23         Pthread_cond_signal(&empty);
24         Pthread_mutex_unlock(&mutex);
25         printf("%d\n", tmp);
26     }
27 }
```
Covering Conditions

• Assume we have implemented a multi-threaded memory allocator
• Also, assume there are zero bytes are currently free
  • Thread $T_a$ calls $\text{allocate}(100)$
  • Thread $T_b$ calls $\text{allocate}(10)$
  • Both $T_a$ and $T_b$ wait on the condition and go to sleep
  • Thread $T_c$ calls $\text{free}(50)$
    • Which waiting thread should be woken up?
Covering Conditions Example Code

```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); // who do we signal??
    Pthread_mutex_unlock(&m);
}
```
Covering Conditions Solution

• Solution:
  • Replace `pthread_cond_signal()` with `pthread_cond_broadcast()`

• `pthread_cond_broadcast()`
  • Wake up all waiting threads
  • Cost: too many threads might be woken up
  • Threads that shouldn’t be woken up will simply wake up, re-check the condition, and then go back to sleep
Covering Conditions Solution Code

```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_broadcast(&c);  // wake up all the threads waiting
    Pthread_mutex_unlock(&m);
}
```
Condition Variable Summary

- We have a new *synchronization* primitive beyond locks
  - Condition variables

- Allows for a thread to sleep when some program state is not as desired
  - Once sleeping, another thread must wake up the thread by signal/broadcast

- Condition variables are used in conjunction with a lock
  - When waiting on the CV, the lock is (temporarily) given up
  - While returning from the wait, the thread re-acquires the lock

- When a thread is signaled, it may not wake up right away
  - The state of the world may have changed
  - Recheck your state (in a while loop) upon returning from wait if there is any chance the state may have changed