Condition Variables & Semaphores

Nima Honarmand
(Based on slides by Prof. Andrea Arpaci-Dusseau)

Review: Concurrency Objectives

- **Mutual Exclusion** — A & B don't run at the same time
  - Solved using ***locks***
- **Ordering** — B runs after A does something
  - Solved using ***condition variables***

Example 1: Thread Join

```
pthread_t p1, p2;

// create child threads
pthread_create(&p1, NULL, mythread, "A");
pthread_create(&p2, NULL, mythread, "B");

…

// join waits for the child threads to finish
thr_join(p1, NULL);
thr_join(p2, NULL);

return 0;
```

how to implement `thr_join()`?

Waiting for an Event

- Parent thread has to wait until child terminates
- Option 1: spin until that happens
  - Waste of CPU time
- Option 2: wait (sleep) in a queue until that happens
  - Better use of CPU time
  - Similar to the idea in queue-based lock of previous lecture
  - Child thread will signal the parent to wake up before its termination

Generalizing Option 2

- **Condition Variable**: queue of waiting threads with two basic operations
- *B* waits for a signal on `cv` before running
  - `cond_wait(cv, …)`
- *A* sends signal to `cv` to wake-up one waiting thread
  - `cond_signal(cv, …)`

Thread Join: Attempt 1

Parent

```
void thr_join() {
  cond_wait(&c);
}
```

Child

```
void thr_exit() {
  cond_signal(&c);
}
```

- Does this work? If not, what's the problem?
- Child may run and call `cond_signal()` before parent called `cond_wait()`

  → Parent will sleep indefinitely

Thread Join: Attempt 2

Parent

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

Child

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

- Let's keep some state then
- Is there a problem here?

Thread Join: Attempt 2

Parent

```
void thr_join() {
  if (done == 0) {        //a
    cond_wait(&c);        //b
  }
}
```

Child

```
void thr_exit() {
  done = 1;               //x
  cond_signal(&c);        //y
}
```

- Let's keep some state then

  Parent: a b
  Child: x y

- Again, parent may sleep indefinitely
- Solution?

Using Locks to Achieve Atomicity

Waiting Thread

```
mutex_lock(&m);
if (! check_cond())
    cond_wait(&c, &m);
…
mutex_unlock(&m);
```

Waking Thread

```
mutex_lock(&m);
set_cond();
cond_signal(&c);
…
mutex_unlock(&m);
```

- Need a lock (called `mutex` in pthreads) to ensure two things
  1) Checking condition (waiting thread) & modifying it (waking thread) remain mutually exclusive
  2) Checking condition & putting thread to sleep (waiting thread) remain atomic

- `cond_wait()` should unlock mutex atomically w/ going to sleep
  - If mutex not released, waking thread cannot make progress
  - If release is not atomic, we get a race condition. Can you identify it?

Using Locks to Achieve Atomicity

- `cond_wait()` releases the mutex atomically with going to sleep
- `cond_wait()` re-acquires the mutex immediately after being awoken (before returning)
- To be safe, should always be holding mutex when calling `cond_signal()`

Spurious Wakeups

- In most systems, a sleeping thread might be awoken *spuriously*
  - In addition to being awoken when signaled
- So, no guarantee that condition you've been waiting for is true when you are awoken
- Need to check the condition again before continuing
  - How?

Waiting Thread

```
mutex_lock(&m);
while (! check_cond())
    cond_wait(&c, &m);
…
mutex_unlock(&m);
```

Thread Join: Correct Solution

Parent

```
void thr_join() {
  mutex_lock(&m);
  while (done == 0)
    cond_wait(&c, &m);
  mutex_unlock(&m);
}
```

Child

```
void thr_exit() {
  mutex_lock(&m);
  done = 1;
  cond_signal(&c);
  mutex_unlock(&m);
}
```

- This code works for one parent and one child
- Does it work for one parent and multiple children?
  - Yes
- What if there were multiple parents each with multiple children?
  - It won't work; we'll revisit that case later

Exercise

- Implement cond_wait and cond_signal
- Hine: can use `park()`, `unpark()` and `setpark()`
  - As we did for the queue lock

Recap: CV Rules of Thumb (Take 1)

- Shared state determines if condition is true or not
- Check the state before waiting on cv
  - In a while loop
- Use a mutex to protect
  1) the shared state on which condition is based, as well as,
  2) operations on the cv
- Remember to acquire the mutex before calling `cond_signal()`

Example 2: Bounded Buffer

- Classic producer/consumer problem
- Multiple producers and multiple consumers communicate using a shared, finite-size buffer
- Producers add items to buffer
  - If buffer is full, producer has to wait until there is free space
- Consumers remove items from buffer
  - If buffer is empty, consumer has to wait until one or more items are added
- Common examples:
  - Unix pipe: bounded buffer in kernel (multiple producers & consumers)
  - Work queue in a web server (one producer, multiple consumers)

Bounded Buffer: Attempt 1

Producer

```
for (int i=0; i<loops; i++) {
  mutex_lock(&m);
  while (numfull == MAX)
    cond_wait(&cond, &m);
  do_fill(i);
  cond_signal(&cond);
  mutex_unlock(&m);
}
```

Consumer

```
while(1) {
  mutex_lock(&m);
  while (numfull == 0)
    cond_wait(&cond, &m);
  int tmp = do_get();
  cond_signal(&cond);
  mutex_unlock(&m);
  printf("%d\n", tmp);
}
```

- Starting simple: assume one producer, one consumer
  - `numfull`: number of elements in the buffer
- Does this code work for 1P and 1C?
  - Yes ☺

Bounded Buffer: Attempt 1

Producer

```
for (int i=0; i<loops; i++) {
  mutex_lock(&m);
  while (numfull == MAX)
    cond_wait(&cond, &m); //a
  do_fill(i);             //b
  cond_signal(&cond);     //c
  mutex_unlock(&m);
}
```

Consumer

```
while(1) {
  mutex_lock(&m);
  while (numfull == 0)
    cond_wait(&cond, &m); //x
  int tmp = do_get();     //y
  cond_signal(&cond);     //z
  mutex_unlock(&m);
  printf("%d\n", tmp);
}
```

- How about 1P and 2C? Would it work?
  - No ☹ Why?

Bounded Buffer: Attempt 1

- Say queue size is one (i.e., it can hold only one item)
- C1 and C2 initially find queue empty so they are waiting (line x)

1) P adds an item to buffer (line b), signals `cond` (line c), waking up C1, waits on `cond` until signaled (line a)
2) C1 is awoken, removes item from buffer (line y), signals `cond` (line z), waking up C2, finds buffer empty, goes to sleep (line x)
3) C2, being woken up by C1, finds buffer empty, goes to sleep waiting on `cond` (line x)

- Everyone is sleeping → P can't produce → no forward progress

- **Crux**: C1's signal was meant to awaken P but it awoke C2

Solution 1: Wake up Everyone

- When not sure if next waiting thread is the right one to wake up, just wake up all
- Not the most elegant solution (that's Solution 2)
  - Probably bad for performance: all awoken threads will compete for mutex again
  - But a good fallback mechanism to ensure correctness
- Need a new API: `cond_broadcast(cv)`
  - Semantic: wakes up all the queues waiting on cv
- There are cases where there is no elegant solution and we have to use broadcast
  - See the memory allocator example in OSTEP, Section 30.3

Solution 2: Use Multiple CVs

- Identify different conditions that need waiting for
- Use a separate CV for each condition using `cond_wait()` and `cond_signal()`
- More elegant, better-performing solution than using `cond_broadcast()`
- Different conditions in bounded buffer problem?
  - Two
  - Waiting for queue to become non-full
  - Waiting for queue to become non-empty

Bounded Buffer: Correct & Elegant Solution

Producer

```
for (int i=0; i<loops; i++) {
  mutex_lock(&m);
  while (numfull == MAX)
    cond_wait(&non_full, &m);
  do_fill(i);
  cond_signal(&non_empty);
  mutex_unlock(&m);
}
```

Consumer

```
while(1) {
  mutex_lock(&m);
  while (numfull == 0)
    cond_wait(&non_empty, &m);
  int tmp = do_get();
  cond_signal(&non_full);
  mutex_unlock(&m);
  printf("%d\n", tmp);
}
```

- Would it be okay also to use two mutexes?
  - No
- Why?
  - Because mutex protects associated with the shared state (buffer, in this case)

Example 3: Join w/ Multiple Parents

Parent 1

```
pthread_t p1, p2;

// create child threads
pthread_create(&p1, NULL, mythread, "A");
pthread_create(&p2, NULL, mythread, "B");

// …

// join waits for the child threads to finish
thr_join(p1, NULL);
thr_join(p2, NULL);

return 0;
```

Parent 2

```
pthread_t p1, p2;

// create child threads
pthread_create(&p1, NULL, mythread, "C");
pthread_create(&p2, NULL, mythread, "D");

// …

// join waits for the child threads to finish
thr_join(p1, NULL);
thr_join(p2, NULL);

return 0;
```

- Consider multiple parents each with multiple children
  - However, each child only has one parent
- Assume a parent thread may only join its own children
- **NOTE**: This semantic is different from `pthread_join()`

Example 3: Join w/ Multiple Parents

Parent

```
void thr_join(int i) {
  mutex_lock(&m);
  while (done[i] == 0)
    cond_wait(&c, &m);
  mutex_unlock(&m);
}
```

Child

```
void thr_exit() {
  mutex_lock(&m);
  done[my_id] = 1;
  cond_signal(&c);
  mutex_unlock(&m);
}
```

- Obviously we need an array of `done` flags, one per child
- Is this code correct?
  - No
  - When a child signals `c`, it is not guaranteed to awaken its own parent
- Solutions:
  1) Use `cond_broadcast()` to awaken all sleeping parents
  2) Use `cond_signal()` but use a separate CV for each parent
  3) Use `cond_signal()` but use a separate CV for each child

Example 3: Solution 1

Parent

```
void thr_join(int i) {
  mutex_lock(&m);
  while (done[i] == 0)
    cond_wait(&c, &m);
  mutex_unlock(&m);
}
```

Child

```
void thr_exit() {
  mutex_lock(&m);
  done[my_id] = 1;
  cond_broadcast(&c);
  mutex_unlock(&m);
}
```

- Obviously we need an array of `done` flags, one per child
- Is this code correct?
  - No
  - When a child signals `c`, it is not guaranteed to awaken its own parent
- Solutions:
  1) **Use `cond_broadcast()` to awaken all sleeping parents**
  2) Use `cond_signal()` but use a separate CV for each parent
  3) Use `cond_signal()` but use a separate CV for each child

Example 3: Solution 2

Parent

```
void thr_join(int i) {
  mutex_lock(&m);
  while (done[i] == 0)
    cond_wait(&c[my_id], &m);
  mutex_unlock(&m);
}
```

Child

```
void thr_exit() {
  mutex_lock(&m);
  done[my_id] = 1;
  cond_signal(&c[my_parent]);
  mutex_unlock(&m);
}
```

- Obviously we need an array of `done` flags, one per child
- Is this code correct?
  - No
  - When a child signals `c`, it is not guaranteed to awaken its own parent
- Solutions:
  1) Use `cond_broadcast()` to awaken all sleeping parents
  2) **Use `cond_signal()` but use a separate CV for each parent**
  3) Use `cond_signal()` but use a separate CV for each child

Example 3: Solution 3

Parent

```
void thr_join(int i) {
  mutex_lock(&m);
  while (done[i] == 0)
    cond_wait(&c[i], &m);
  mutex_unlock(&m);
}
```

Child

```
void thr_exit() {
  mutex_lock(&m);
  done[my_id] = 1;
  cond_signal(&c[my_id]);
  mutex_unlock(&m);
}
```

- Obviously we need an array of `done` flags, one per child
- Is this code correct?
  - No
  - When a child signals `c`, it is not guaranteed to awaken its own parent
- Solutions:
  1) Use `cond_broadcast()` to awaken all sleeping parents
  2) Use `cond_signal()` but use a separate CV for each parent
  3) **Use `cond_signal()` but use a separate CV for each child**

Recap: CV Rules of Thumb (Take 2)

- Shared state determines if condition is true or not
- Check the state before waiting on cv
  - In a while loop
- Use a mutex to protect
  1) the shared state on which condition is based, as well as,
  2) operations on the cv
- Remember to acquire the mutex before calling `cond_signal()` and `cond_broadcast()`
- Use different CVs for different conditions
- Sometimes, `cond_broadcast()` helps if you can't find an elegant solution using `cond_signal()`

Pthreads Condition Variable API

- Creation/destruction
  - `pthread_cond_init(cv, attr)`
  - `pthread_cond_destroy(cv)`
  - `pthread_condattr_init(attr)`
  - `pthread_condattr_destroy(attr)`
- Waiting and waking
  - `pthread_cond_wait(cv, mutex)`
  - `pthread_cond_timedwait(cv, mutex, time)`
  - `pthread_cond_signal(cv)`
  - `pthread_cond_broadcast(cv)`
- Required reading linked on the course schedule page

Semaphores

- A synchronization primitive that can work both as a lock, as well as a special case of condition variables
  - In particular, for Bounded Buffer problem
- Not easy to use as a general condition variable
- Not easy to use to build a general condition variable
  - Doable but quite difficult
  - See Microsoft Research's attempt at http://research.microsoft.com/pubs/64242/ImplementingCVs.pdf

Semaphores (2)

- Read more in OSTEP, Chapter 31
- More of an intellectual curiosity, IMHO
  - A nice one though, worth reading about
- Pthreads just have locks and condition variables, but no semaphores