IE411: Operating Systems

Condition Variables

Review: Concurrency Objectives

- 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

Review: Concurrency Objectives

- 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
  - This is often called a join

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()`?

How to implement thr_join()?

- Parent thread must wait until child terminates
- Option 1: spin until that happens
  - Waste of CPU time

How to implement thr_join()?

- Parent thread must 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
  - 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

Generalizing Option 2

- Condition Variable: queue of waiting threads with two basic operations
- Thread B waits for a signal on cv before running
  - cond_wait(cv, . . .)

Generalizing Option 2

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

Thread join: Attempt 1 (broken)

Parent

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

Child

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

Thread join: Attempt 1 (broken)

Parent

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

Child

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

- What's the issue here?

Thread join: Attempt 1 (broken)

Parent

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

Child

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

- What's the issue here?
- Just before it calls `cond_wait()` to go to sleep, the parent is interrupted and the child runs
- The child calls `cond_signal()`
  - But no thread is waiting and thus no thread is woken (condition variables have no history)
  - When the parent runs again, it sleeps forever

Thread join: Attempt 2 (broken)

Parent

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

Child

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

Thread join: Attempt 2 (broken)

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

Thread join: Attempt 2 (broken)

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
- Before calling cond_wait() in parent thread, check if the child thread has already called cond_signal()

Thread join: Attempt 2 (broken)

Parent

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

Child

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

- Is there a problem here?

Thread join: Attempt 2 (broken)

Parent

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

Child

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

- Is there a problem here?
- Again, parent may sleep indefinitely

  Parent: a b
  Child: x y

Thread join: Attempt 2 (broken)

Parent

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

Child

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

- Is there a problem here?
- Again, parent may sleep indefinitely

  Parent: a b
  Child: x y

- Solution?

Thread join: Attempt 3

Parent:

```
void thread_join() {
        Mutex_lock(&m);              // w
        if (done == 0)               // x
                Cond_wait(&c, &m); // y
        Mutex_unlock(&m);            // z
}
```

Child:

```
void thread_exit() {
        Mutex_lock(&m);              // a
        done = 1;                    // b
        Cond_signal(&c);             // c
        Mutex_unlock(&m);            // d
}
```

- We use a lock to ensure that checking condition (parent thread) and modifying it (child thread) remain mutually exclusive

Thread join: Attempt 3

Parent:

```
void thread_join() {
        Mutex_lock(&m);              // w
        if (done == 0)               // x
                Cond_wait(&c, &m); // y
        Mutex_unlock(&m);            // z
}
```

Child:

```
void thread_exit() {
        Mutex_lock(&m);              // a
        done = 1;                    // b
        Cond_signal(&c);             // c
        Mutex_unlock(&m);            // d
}
```

- We use a lock to ensure that checking condition (parent thread) and modifying it (child thread) remain mutually exclusive
- Additionally

Thread join: Attempt 3

Parent:

```
void thread_join() {
        Mutex_lock(&m);              // w
        if (done == 0)               // x
                Cond_wait(&c, &m); // y
        Mutex_unlock(&m);            // z
}
```

Child:

```
void thread_exit() {
        Mutex_lock(&m);              // a
        done = 1;                    // b
        Cond_signal(&c);             // c
        Mutex_unlock(&m);            // d
}
```

- We use a lock to ensure that checking condition (parent thread) and modifying it (child thread) remain mutually exclusive
- Additionally
  - checking condition and putting thread to sleep should remain atomic (in parent)

Thread join: Attempt 3

Parent:

```
void thread_join() {
        Mutex_lock(&m);              // w
        if (done == 0)               // x
                Cond_wait(&c, &m); // y
        Mutex_unlock(&m);            // z
}
```

Child:

```
void thread_exit() {
        Mutex_lock(&m);              // a
        done = 1;                    // b
        Cond_signal(&c);             // c
        Mutex_unlock(&m);            // d
}
```

- We use a lock to ensure that checking condition (parent thread) and modifying it (child thread) remain mutually exclusive
- Additionally
  - checking condition and putting thread to sleep should remain atomic (in parent)
  - modifying condition and signaling parent should remain atomic (in child)

Thread join: Attempt 3

Parent:

```
void thread_join() {
        Mutex_lock(&m);              // w
        if (done == 0)               // x
                Cond_wait(&c, &m); // y
        Mutex_unlock(&m);            // z
}
```

Child:

```
void thread_exit() {
        Mutex_lock(&m);              // a
        done = 1;                    // b
        Cond_signal(&c);             // c
        Mutex_unlock(&m);            // d
}
```

- The wait() call takes a lock as a parameter
- wait(cond_t *cv, mutex_t *lock)
  - it is assumed that the thread is holding the lock is held when wait() is called
  - puts caller to sleep + releases the lock (atomically)
  - when awoken, reacquires lock before returning

Thread join: Attempt 3

Parent:

```
void thread_join() {
        Mutex_lock(&m);              // w
        if (done == 0)               // x
                Cond_wait(&c, &m); // y
        Mutex_unlock(&m);            // z
}
```

Child:

```
void thread_exit() {
        Mutex_lock(&m);              // a
        done = 1;                    // b
        Cond_signal(&c);             // c
        Mutex_unlock(&m);            // d
}
```

- wait() puts caller to sleep + release the lock (atomically)
  - If lock is not released, child thread cannot make progres
  - If release w/ going to sleep is not atomic, we get a race condition. Can you identify it?

Exercise

- Implement `cond_wait` and `cond_signal`
- Hint: can use `park()`, `unpark()` and `setpark()`
  - as we did for sleeping lock