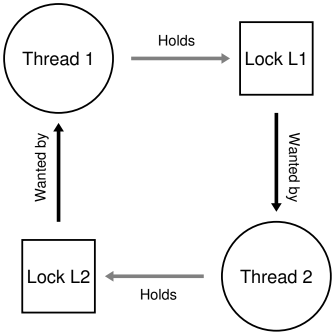


CSL 301

OPERATING SYSTEMS

Lecture 23

Condition Variables
Concurrency Problems

Instructor
Dr. Dhiman Saha

A Parent Waiting For Its Child

What if a parent thread wants to wait for a child thread to complete?

```
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); // child
        // XXX how to wait for child?
        printf("parent:␣end\n");
        return 0;
}
```

Desired output:

```
parent: begin
child
parent: end
```

The Spin-based Approach (and its problems)

We could use a shared variable.

```
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); // child
        while (done == 0)
        ; // spin
        printf("parent:␣end\n");
        return 0;
}
```

This is hugely inefficient! The parent thread spins, wasting CPU time.

How to Wait for a Condition?

The Crux of the Problem

In multi-threaded programs, it is often useful for a thread to wait for some condition to become true before proceeding. The simple approach of just spinning until the condition becomes true, is grossly inefficient and wastes CPU cycles, and in some cases, can be incorrect. Thus, how should a thread wait for a condition?

Condition Variables

A **condition variable** is an explicit queue that threads can put themselves on when some state of execution is not as desired.

- A thread can **wait** on a condition, putting it to sleep.
- Another thread can **signal** on the condition, waking up a waiting thread.

POSIX Condition Variables:

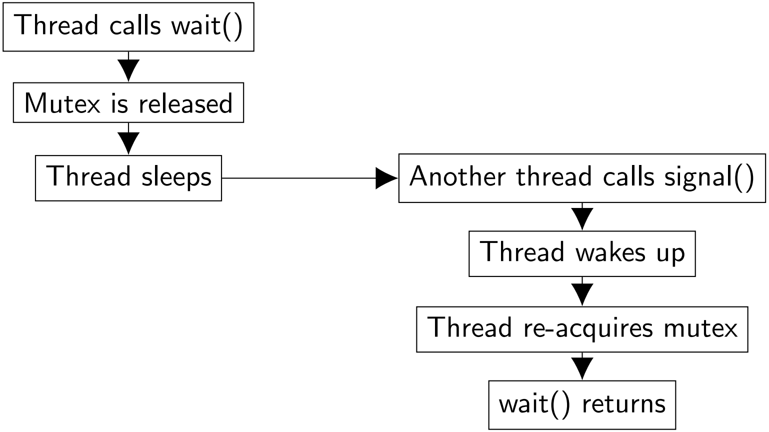
- `pthread_cond_t c;`
- `pthread_cond_wait(pthread_cond_t *c, pthread_mutex_t *m);`
- `pthread_cond_signal(pthread_cond_t *c);`

The `wait()` and `signal()` operations

The `wait()` call is executed when a thread wishes to put itself to sleep; the `signal()` call is executed when a thread has changed something in the program and thus wants to wake a sleeping thread waiting on this condition.

Important: The `wait()` call takes a mutex as a parameter.

- The mutex is assumed to be **locked** when `wait()` is called.
- `wait()` **releases the lock** and puts the calling thread to sleep (atomically).
- When the thread wakes up, it **re-acquires the lock** before returning.


Thread calls wait()
Mutex is released
Thread sleeps
Another thread calls signal()
Thread wakes up
Thread re-acquires mutex
wait() returns

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;
}
```

Parent Waiting For Child: Using a Condition Variable

```
void thr_join() {
        pthread_mutex_lock(&m);
        while (done == 0)
        pthread_cond_wait(&c, &m);
        pthread_mutex_unlock(&m);
}

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: Use A Condition Variable

```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);
}

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;
}
```

Point-to-ponder

- Why while loop?
- Would a `if` condition have sufficed?

Point-to-ponder

- Note the mutex! Do we need it?

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

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

Producer-Consumer with 2 CVs

```
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 == MAX)
            Pthread_cond_wait(&empty, &mutex);
        put(i);
        Pthread_cond_signal(&fill);
        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(&fill, &mutex);
        int tmp = get();
        Pthread_cond_signal(&empty);
        Pthread_mutex_unlock(&mutex);
        printf("%d\n", tmp);
    }
```

Concurrency Problems

Bugs

- Are bugs in concurrent programs deterministic?
- Bug types: Deadlock, non-deadlock
- Non-deadlock bugs:
    - Due to atomicity: solution: locks
    - Serialization bugs : solution: cv

Deadlock bugs

- Classic example

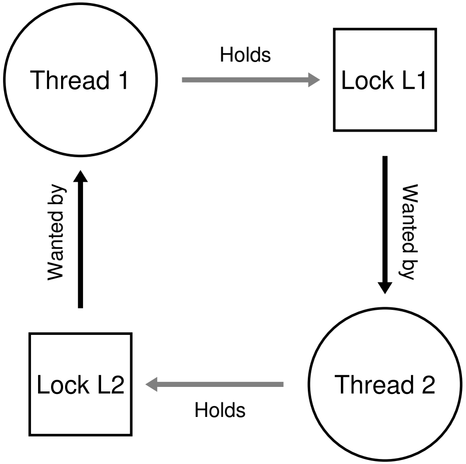
Thread 1:

```
␣␣␣pthread_mutex_lock(L1);
␣␣␣pthread_mutex_lock(L2);
␣␣␣
```

Thread 2:

```
␣␣␣pthread_mutex_lock(L2);
␣␣␣pthread_mutex_lock(L1);
␣␣␣
```

The Deadlock Dependency Graph



Cycle $\Longrightarrow$ Deadlock

Deadlock Conditions

- **Mutual exclusion:** Threads claim exclusive control of resources that they require (e.g., a thread grabs a lock).

Deadlock Conditions

- **Mutual exclusion:** Threads claim exclusive control of resources that they require (e.g., a thread grabs a lock).
- **Hold-and-wait:** Threads hold resources allocated to them while waiting for additional re-sources.

Deadlock Conditions

- **Mutual exclusion:** Threads claim exclusive control of resources that they require (e.g., a thread grabs a lock).
- **Hold-and-wait:** Threads hold resources allocated to them while waiting for additional re-sources.
- **No preemption:** Resources (e.g., locks) cannot be forcibly removed from threads that are holding them.

Deadlock Conditions

- **Mutual exclusion:** Threads claim exclusive control of resources that they require (e.g., a thread grabs a lock).
- **Hold-and-wait:** Threads hold resources allocated to them while waiting for additional re-sources.
- **No preemption:** Resources (e.g., locks) cannot be forcibly removed from threads that are holding them.
- **Circular wait:** Cycle in dependency graph

Deadlock Conditions

- **Mutual exclusion:** Threads claim exclusive control of resources that they require (e.g., a thread grabs a lock).
- **Hold-and-wait:** Threads hold resources allocated to them while waiting for additional re-sources.
- **No preemption:** Resources (e.g., locks) cannot be forcibly removed from threads that are holding them.
- **Circular wait:** Cycle in dependency graph

ALL four of the above conditions must hold for adeadlock to occur

Preventing Deadlocks

Preventing Circular Wait

- Acquire locks in order (e,g. address of lock variable)

```
if (m1 > m2) {  // grab locks in high-to-low address order
    pthread_mutex_lock(m1);
    pthread_mutex_lock(m2);
    }
else {
    pthread_mutex_lock(m2);
    pthread_mutex_lock(m1);
    }
    // Code assumes that m1 != m2 (it is not the same lock
```

Preventing Deadlocks

Preventing Hold and Wait

- Acquire all locks at once, **atomically**.

```
pthread_mutex_lock(prevention);   // begin lock acquisitio
pthread_mutex_lock(L1);
pthread_mutex_lock(L2);
...
pthread_mutex_unlock(prevention); // end
```

Preventing Deadlocks

Preventing No Preemption

```
top:
    pthread_mutex_lock(L1);
    if (pthread_mutex_trylock(L2) != 0) {
        pthread_mutex_unlock(L1);
    goto top;
    }
```

Preventing Mutual Exclusion

```
int ComNSwap(int*address, int expected, int new) {
    if (*address == expected) {
         *address = new;
         return 1; // success
         }
    return 0; // failure
    }

void AtomicIncrement(int*value, int amount) {
    do {
        int old =*value;
        } while (ComNSwap(value, old, old + amount) == 0);
        }
```

Other Solutions

Deadlock avoidance

- OS Support
- Use Scheduling
- Banker's Algorithm : Impractical
  - Assumes a priori knowledge about resource requirements

	T1	T2	T3	T4
L1	yes	yes	no	no
L2	yes	yes	yes	no

	T1	T2	T3	T4
L1	yes	yes	yes	no
L2	yes	yes	yes	no

Deadlock Recovery

- Kill deadlocked processes/ Shutdown-reboot