

# Section 5: Condition Variables and Spin Locks

Jason Jia, Roger Chen

February 19, 2015

## Contents

<b>1</b>	<b>Warmup</b>	<b>2</b>
1.1	Hello World . . . . .	2
<b>2</b>	<b>Vocabulary</b>	<b>2</b>
<b>3</b>	<b>Problems</b>	<b>3</b>
3.1	Hello Word Continued . . . . .	3
3.2	Spot the Problem . . . . .	3
3.3	Baking with Condition Variables . . . . .	4
3.4	test_and_set . . . . .	6
3.5	test_and_test_and_set? . . . . .	7

# 1 Warmup

## 1.1 Hello World

Will this code compile/run?

Why or why not?

```
pthread_mutex_t lock;
pthread_cond_t cv;
int hello = 0;

void print_hello() {
    hello += 1;
    printf("First line (hello=%d)\n", hello);
    pthread_cond_signal(&cv);
    pthread_exit(0);
}

void main() {
    pthread_t thread;
    pthread_create(&thread, NULL, (void *) &print_hello, NULL);
    while (hello < 1) {
        pthread_cond_wait(&cv, &lock);
    }
    printf("Second line (hello=%d)\n", hello);
}
```

## 2 Vocabulary

- **Condition Variable** - A synchronization variable that provides serialization (ensuring that events occur in a certain order). A condition variable is associated with:
  - a lock (a condition variable + its lock are known together as a **monitor**)
  - some boolean condition (e.g. `hello < 1`)
  - a queue of threads waiting for the condition to be true

In order to access any CV functions **OR** to change the truthfulness of the condition, a thread must/should hold the lock. Condition variables offer the following methods:

- **cv\_wait(cv, lock)** - Atomically unlocks the lock, adds the current thread to **cv**'s thread queue, and puts this thread to sleep.
- **cv\_notify(cv)** - Removes one thread from **cv**'s queue, and puts it in the ready state.
- **cv\_broadcast(cv)** - Removes all threads from **cv**'s queue, and puts them all in the ready state.

When a `wait()`ing thread is notified and put back in the ready state, it also re-acquires the lock before the `wait()` function returns.

When a thread runs code that may potentially make the condition true, it should acquire the lock, modify the condition however it needs to, call `notify()` or `broadcast()` on the condition's CV, so waiting threads can be notified, and finally release the lock.

Why do we need a lock anyway? Well, consider a race condition where thread 1 evaluates the condition  $C$  as false, then thread 2 makes condition  $C$  true and calls `cv.notify`, then 1 calls `cv.wait` and goes to sleep. Thread 1 might never wake up, since it went to sleep too late.

- **Hoare Semantics** - (In terms of condition variable) Wake a blocked thread when the condition is true and transfer control of the CPU and ownership of the lock to that thread immediately. This is difficult to implement in practice and generally not used despite being conceptually easier to deal with.
- **Mesa Semantics** - (In terms of condition variable) Wake a blocked thread when the condition is true, with no guarantee that the thread will execute immediately. The newly woken thread simply gets put on the ready queue and is subject to the same scheduling mechanisms as any other thread. The implication of this is that **you must check the condition with a while loop instead of an if statement because it is possible for the condition to change to false between the time the thread was unblocked and the time it takes over the CPU.**
- **Spin Locks** - A type of lock where the implementation of `lock.acquire()` is to simply check if the lock is available in a loop ("spin"). Since the thread remains active but is not performing a useful task, the use of such a lock is a kind of busy waiting.

## 3 Problems

### 3.1 Hello Word Continued

Add in the necessary code to the warmup to make it work correctly.

### 3.2 Spot the Problem

What is wrong with this code?

```
pthread_mutex_t lock;
pthread_cond_t cv;
int n = 3;
void counter() {
    pthread_mutex_lock(&lock);
    for (n = 3; n > 0; n--)
        printf("%d\n", n);
    pthread_cond_signal(&cv);
    pthread_mutex_unlock(&lock);
}
void announcer() {
    while (n != 0) {
        pthread_mutex_lock(&lock);
        pthread_cond_wait(&cv, &lock);
        pthread_mutex_unlock(&lock);
    }
    printf("BLAST OFF!\n");
}
```

### 3.3 Baking with Condition Variables

A number of people are trying to bake cakes. Unfortunately, they each know only one skill, so they need to all work together to bake cakes. Use independent threads (one person is one thread) which communicate through condition variables to solve the problem. A skeleton has been provided, fill in the blanks to make the implementation work.

A cake requires:

- 1 cake batter
- 2 eggs

Instructions:

1. Add ingredients to bowl
2. Heat bowl (it's oven-safe)
3. Eat the cake, clean out the bowl, and go back to step 1

Requirements:

- Don't start heating the cake in the oven unless there are exactly the right number of ingredients in the bowl.
- Don't add raw ingredients to a currently-baking cake or a finished cake.
- Don't eat the cake unless it's done baking.
- Given enough time, the code should bake an unbounded number of cakes, and should never stop.

```
int numBatterInBowl = 0;
int numEggInBowl = 0;
bool readyToEat = false;
pthread_mutex_t lock;
pthread_cond_t needIngredients;
pthread_cond_t readyToBake;
pthread_cond_t startEating;

void batterAdder()
{
    pthread_mutex_lock(&lock);
    while (1) {

        ----- {

        }
        addBatter(); // Sets numBatterInBowl += 1

        -----
    }
}
```

```

void eggBreaker()
{
    pthread_mutex_lock(&lock);
    while (1) {

        ----- {

            -----
        }
        addEgg(); // Sets numEggInBowl += 1

        -----
    }
}

void bowlHeater()
{
    pthread_mutex_lock(&lock);
    while (1) {

        ----- {

            -----
        }
        heatBowl(); // Sets readyToEat = true, numBatterInBowl = 0, numEggInBowl = 0

        -----
    }
}

void cakeEater()
{
    pthread_mutex_lock(&lock);
    while (1) {

        ----- {

            -----
        }
        eatCake(); // Sets readyToEat = false and cleans the bowl for another cake

        -----
    }
}

int main(int argc, char *argv[])
{
    // Initialize mutex and condition variables
    // Start threads: 1 batterAdder, 2 eggBreakers, 1 bowlHeater, and 1 cakeEater
    // main() sleeps forever

```

### 3.4 test\_and\_set

Assume that I use test\_and\_set to emulate the behavior of locks.

```
int value = 0;
int hello = 0;

void print_hello() {
    while (test_and_set(value));
    hello += 1;
    printf("Child thread: %d\n", hello);
    value = 0;
    pthread_exit(0);
}

void main() {
    pthread_t thread1;
    pthread_t thread2;
    pthread_create(&thread1, NULL, (void *) &print_hello, NULL);
    pthread_create(&thread2, NULL, (void *) &print_hello, NULL);
    while (test_and_set(value));
    printf("Parent thread: %d\n", hello);
    value = 0;
}
```

Assume the following sequence of events:

1. Main starts running and creates both threads and is then context switched right after
2. Thread2 is scheduled and run until after it increments hello and is context switched
3. Thread1 runs until it is context switched
4. The thread running main resumes and runs until it get context switched
5. Thread2 runs to completion
6. The thread running main runs to completion (but doesn't exit yet)
7. Thread1 runs to completion

Is this sequence of events possible? Why or why not?

And each step, if test\_and\_set is called, what value(s) will it return?

Given this sequence of events, what will C print?

Is this a good way to implement locks? Why or why not?

### 3.5 test\_and\_test\_and\_set?

To lower the overhead a more elaborate locking protocol test and test-and-set can be used. The main idea is not to spin in test-and-set but increase the likelihood of successful test-and-set by spinning until the lock seems like it is free.

Fill in the rest of the implementation for a test\_and\_test\_and\_set based lock:

```
int locked = 0;

void lock() {

    ----- // Spin until lock looks empty
    while (test_and_set(locked));
}

void unlock() {

    -----
}
```

Is this a better implementation of a lock than just using test\_and\_set? Why or why not?

Given the following implementations of lock and unlock, what are all the possible outputs that C might print out? Assume the pid for the main thread is 1 and the pid for created thread is 2. Also assume the print statements here are atomic.

```
int locked = 0;

void lock(pid_t pid) {
    while (locked == 1);
    printf("%s\n", "The lock is free!");
    while (test_and_set(&locked));
    printf("Lock acquired by: %d\n", pid);
}

void unlock(pid_t pid) {
    locked = 0;
    printf("Lock released by: %d\n", pid);
}

void do_nothing(void* arg) {
    pid_t pid = getpid();
    lock(pid);
    unlock(pid);
    pthread_exit(0);
}

void main() {
    pthread_t thread;
    pthread_create(&thread, NULL, (void *) &do_nothing, NULL);
    do_nothing();
}
```