Synchronization (2015-11-24)

Critical Section
A critical section is a piece of code that:
- Accesses a shared resource AND
- Must not be concurrently accessed by multiple threads

If we consider the up/down example we have looked at in the past:

```
void *up(void *ptr) {
  int i;
  for (i = 0; i < X; i++)
    ct++;  /*< This line is the critical section! */
}
```

A critical section is said to be correct if it meets three conditions:
(1): **Mutual Exclusion**: Only one thread may be in the critical section at any one time
(2): **Progress**: If a thread requests access to the critical section AND no thread is currently in the critical section, the a thread must be able to enter the critical section.
(3): **Bounded Wait**: Every thread must be able to get access to the critical section within a bounded (or finite) amount of time.

Software Solution #1: Single Lock Variable Solution

```
int lock = 0;

/* Running by two threads: T1 and T2 */
void *up(void *ptr) {
  int i;
  for (i = 0; i < X; i++) {
    while (lock) { /* do nothing and wait for the lock */ }
    lock = 1;
    ct++;  /* Critical Section */
    lock = 0;
  }
}
```

This solution is not correct because it it possible for two threads to access the critical section at one time. We say that this violates Mutual Exclusion.

Software Solution #2: Turns with Strict Alternation

```
int turn = some_id;

/* Running by two threads: T1 and T2 */
void *up(void *ptr) {
  int i;
  for (i = 0; i < X; i++) {
    while (turn == other_id) { }
    ct++;  /* Critical Section */
    turn = other_id;
  }
}
```

This solution is not correct because it it possible for a thread to request access to the critical section while no thread is in the critical section, but it is not able to enter into the critical section. We say that this violates Progress.

Software Solution #3: Other Flag

```
int owner[2] = { false, false };

/* Running by two threads: T1 and T2 */
void *up(void *ptr) {
  int i;
  for (i = 0; i < X; i++) {
    while ( owner[other_id] ) { }
    owner[my_id] = true;
    ct++;  /* Critical Section */
    owner[my_id] = false;
  }
}
```

This solution is not correct because it it possible for two threads to access the critical section at one time. We say that this violates Mutual Exclusion.

Software Solution #4: Two Flag

```
int owner[2] = { false, false };

/* Running by two threads: T1 and T2 */
void *up(void *ptr) {
  int i;
  for (i = 0; i < X; i++) {
    owner[my_id] = true;
    while ( owner[other_id] ) { }
    ct++;  /* Critical Section */
    owner[my_id] = false;
  }
}
```

This solution is not correct because it it possible for a thread to request access to the critical section while no thread is in the critical section, but it is not able to enter into the critical

section. We say that this violates Progress.

Software Solution #5: Two Flag and Turns / Peterson's Solution

```
int owner[2] = { false, false };
int turn = some_id;

/* Running by two threads: T1 and T2 */
void *up(void *ptr) {
  int i;
  for (i = 0; i < X; i++) {
    owner[my_id] = true;
    turn = other_id;
    while ( owner[other_id] && turn == other_id ) { }
    ct++;  /* Critical Section */
    owner[my_id] = false;
  }
}
```

This solution works!

Peterson's Solution
The previous solution (Solution #5) is a famous software-only solution to synchronization called Peterson's Solution. This solution does not require any hardware primitives to exist or any operating system support. However, it suffers from **busy waiting**
- **Busy waiting** is when a thread or process repeatedly checks to see if a condition is true, thereby wasting CPU cycles

Synchronization Primitives
As an alternative to using Peterson's Solution, the OS provides us with three synchronization primitives:
(1): Mutex, a "locK"
(2): Conditional Variables, a "monitor"
(3): Semaphore, a "counting lock" or "counting mutex"

mutex
A mutex is an atomic lock. It it initially in the **unlocked** state when it is initialized.

When you call `pthread_mutex_lock(pthread_mutex_t *)`:
- if the mutex is locked, the thread will block until the mutex is unblocked.
- If the mutex is unlocked, the mutex will become locked but the calling thread will continue.

When you call `pthread_mutex_unlock(pthread_mutex_t *)`:
- Unlocks the mutex.
- If any threads are blocked on `_mutex_lock()`, one thread will wake up, "acquire" the lock by locking it, and continue.

Implementing a mutex: test_and_set()
Modern hardware provides a CPU operation to implement a test_and_set() function.

```
int test_and_set(int *atomic)
```

...atomically sets the value in atomic to 1 and returns the previous value in atomic.

However, we need additional operating system support from the scheduler! The following is still busy waiting:

```
while ( test_and_set(&atomic) == 0 ) { }
```

The actual implementation of a mutex looks is similar to:

```
int pthread_mutex_lock(pthread_mutex_t *mutex)
{
  if ( test_and_set(mutex->lock) == 1 )
    add_to_blocked_queue_on(mutex);
  else
    return 0;
}
```

Example of using a mutex

```
int ct = 0;
int X = 10000000;
pthread_mutex_t mutex;

void *up(void *ptr) {
  int i;
  for (i = 0; i < X; i++) {
    pthread_mutex_lock(&mutex);
    ct++;
    pthread_mutex_unlock(&mutex);
  }
}

void main() {
  pthread_mutex_init(&mutex, NULL);
   /* ... */
  pthread_mutex_destroy(&mutex);
}
```

Conditional Variable
A conditional variable is the synchronization needed to implement a monitor.

On `pthread_cond_wait(pthread_cond_t, pthread_mutex_t)`:
- The calling thread will first give up its lock to the mutex and then move to a blocked state.
- Only once signaled, the waiting mutex will wake up, require the lock, and continue.

On `pthread_cond_signal(pthread_cond_t)`:
- Signals a waiting thread (in a pthread_cond_wait()) to wake up
- If no threads are waiting, this call does nothing. A signal is not "saved".

Creating a monitor with a conditional variable

```
pthread_mutex_lock(&mutex);

while ( /* predicate */ )
  pthread_cond_wait( &cond, &mutex );

/* critical section */

pthread_mutex_unlock(&mutex);
```

...in addition to this, every time anything that is included in the predicate changes, you must call a `pthread_cond_signal()` to ensure you re-check the predicate.

Monitor Example: Blocking Bounded Queue
We'll look at a data structure that is:
- "Blocking": Blocks until it is able to complete an operation is able to be successfully completed.
- "Bounded": Fixed size
- "Queue"

This means that we need to block if the queue is full and a user is trying to add to the queue.

The solution:

```
void blocking_queue_push(queue_t *q, void *data) {
  pthread_mutex_lock(&mutex);

  while ( queue_is_full(q) )   /**< Prevent against buffer overflow */
    pthread_cond_wait(&cond, &mutex);

  /* queue_push() adds the element to the queue; queue_push() is not thread-safe */
  queue_push(q, data);

  pthread_cond_signal(&cond);  /**< Signals that the size of the queue has changed */

  pthread_mutex_unlock(&mutex);
}

void *blocking_queue_pop(queue_t *q) {
  pthread_mutex_lock(&mutex);

  while ( queue_is_empty(q) )   /**< Prevent against buffer underflow */
    pthread_cond_wait(&cond, &mutex);

  /* queue_pop() pops the top element; queue_pop() is not thread-safe */
  void *data = queue_pop(q);

  pthread_cond_signal(&cond);  /**< Signals that the size of the queue has changed */

  pthread_mutex_unlock(&mutex);
}
```

_signal() vs. _broadcast()
There are two ways to wake up threads in a `_cond_wait()`:
(1): `pthread_cond_signal()`: Signals **one** thread that is waiting on the conditional variable.
(2): `pthread_cond_broadcast()`: Signals **all** threads waiting on the conditional variable.