Locks and Condition Variables

Lecture Notes for CS 140
Spring 2020
John Ousterhout

- Readings for this topic from *Operating Systems: Principles and Practice*: Sections 5.2-5.4.

- Needed: higher-level synchronization mechanism that provides
  - Mutual exclusion: easy to create critical sections
  - Blocking: delay a thread until some desired event occurs

Locks

- *Lock*: an object that can only be owned by a single thread at any given time. Basic operations on a lock:
  - `acquire`: mark the lock as owned by the current thread; if some other thread already owns the lock then first wait until the lock is free. Lock typically includes a queue to keep track of multiple waiting threads.
  - `release`: mark the lock as free (it must currently be owned by the calling thread).

- Too much milk solution with locks (using Pintos APIs):

```
struct lock l;
...
lock_acquire(&l);
if (milk == 0) {
  buy_milk();
}
lock_release(&l);
```

- A more complex example: producer/consumer.
  - Producers add characters to a buffer
  - Consumers remove characters from the buffer
  - Characters will be removed in the same order added
  - Version 1:

```
char buffer[SIZE];
int count = 0, putIndex = 0, getIndex = 0;
struct lock l;
lock_init(&l);

void put(char c) {
    lock_acquire(&l);
    count++;
    buffer[putIndex] = c;
    putIndex++;
    if (putIndex == SIZE) {
        putIndex = 0;
    }
    lock_release(&l);
}

char get() {
    char c;
    lock_acquire(&l);
    count--;
    c = buffer[getIndex];
    getIndex++;
    if (getIndex == SIZE) {
        getIndex = 0;
    }
    lock_release(&l);
    return c;
}
```

- Version 2 (handle empty/full cases):

```
char buffer[SIZE];
int count = 0, putIndex = 0, getIndex = 0;
struct lock l;
lock_init(&l);

void put(char c) {
    lock_acquire(&l);
    while (count == SIZE) {
        lock_release(&l);
        lock_acquire(&l);
    }
    count++;
    buffer[putIndex] = c;
    putIndex++;
    if (putIndex == SIZE) {
        putIndex = 0;
    }
    lock_release(&l);
}

char get() {
    char c;
    lock_acquire(&l);
    while (count == 0) {
        lock_release(&l);
        lock_acquire(&l);
    }
    count--;
    c = buffer[getIndex];
    getIndex++;
    if (getIndex == SIZE) {
        getIndex = 0;
    }
    lock_release(&l);
    return c;
}
```

Condition Variables

- Synchronization mechanisms need more than just mutual exclusion; also need a way to wait for another thread to do something (e.g., wait for a character to be added to the buffer)

- *Condition variables*: used to wait for a particular state to be reached (e.g. characters in buffer).
    - `wait(condition, lock)`: atomically release lock, put thread to sleep until `condition` is signaled; when thread wakes up again, re-acquire lock before returning.
    - `signal(condition, lock)`: if any threads are waiting on `condition`, wake up one of them. Caller must hold `lock`, which must be the same as the `lock` used in the `wait` call.
    - `broadcast(condition, lock)`: same as `signal`, except wake up all waiting threads.
    - Note: after `signal`, signaling thread keeps lock, waking thread goes on the queue waiting for the lock.

  - Warning: when a thread wakes up after `cond_wait` there is no guarantee that the desired condition still exists: another thread might have snuck in.

- Producer/Consumer, version 3 (with condition variables):

```
char buffer[SIZE];
int count = 0, putIndex = 0, getIndex = 0;
struct lock l;
struct condition charAdded;
struct condition charRemoved;

lock_init(&l);
cond_init(&charAdded);
cond_init(&charRemoved);

void put(char c) {
    lock_acquire(&l);
    while (count == SIZE) {
        cond_wait(&charRemoved, &l);
    }
    count++;
    buffer[putIndex] = c;
    putIndex++;
    if (putIndex == SIZE) {
        putIndex = 0;
    }
    cond_signal(&charAdded, &l);
    lock_release(&l);
}

char get() {
    char c;
    lock_acquire(&l);
    while (count == 0) {
        cond_wait(&charAdded, &l);
    }
    count--;
    c = buffer[getIndex];
    getIndex++;
    if (getIndex == SIZE) {
        getIndex = 0;
    }
    cond_signal(&charRemoved, &l);
    lock_release(&l);
    return c;
}
```

Monitors

- When locks and condition variables are used together like this, the result is called a *monitor*:
  - A shared data structure
  - A collection of procedures
  - One lock that must be held whenever accessing the shared data (typically each procedure acquires the lock at the very beginning and releases the lock before returning).

    - One or more condition variables used for waiting.

- There are other synchronization mechanisms besides locks and condition variables. Be sure to read about semaphores in the book or in the Pintos documentation.