Semaphores — OS Support for Mutual Exclusion (Review)

Even with semaphores, some synchronization errors can occur:

<table>
<thead>
<tr>
<th>Honest Mistake</th>
<th>Careless Mistake</th>
</tr>
</thead>
<tbody>
<tr>
<td>milk-&gt;V();</td>
<td>milk-&gt;P();</td>
</tr>
<tr>
<td>if (noMilk)</td>
<td>if (noMilk)</td>
</tr>
<tr>
<td>buy milk;</td>
<td>buy milk;</td>
</tr>
<tr>
<td>milk-&gt;P();</td>
<td>milk-&gt;P();</td>
</tr>
</tbody>
</table>

- Other variations possible

Solution — new language constructs

- (Conditional) Critical region
  - region v when B do S;
  - Variable v is a shared variable that can only be accessed inside the critical region
  - Boolean expression B governs access
  - Statement S (critical region) is executed only if B is true; otherwise it blocks until B does become true

- Monitor

From Semaphores to Locks and Condition Variables

- A semaphore serves two purposes:
  - Mutual exclusion — protect shared data
    - mutex in Coke machine
    - milk in Too Much Milk
    - Always a binary semaphore
  - Synchronization — temporally coordinate events (one thread waits for something, other thread signals when it’s available)
    - fullSlot and emptySlot in Coke machine
    - Either a binary or counting semaphore

- Idea — two separate constructs:
  - Locks — provide mutually exclusion
  - Condition variables — provide synchronization
  - Like semaphores, locks and condition variables are language-independent, and are available in many programming environments

Locks

**Locks** provide mutually exclusive access to shared data:

- A lock can be “locked” or “unlocked” (sometimes called “busy” and “free”)

Operations on locks (Nachos syntax):

- Lock(*name) — create a new (initially unlocked) Lock with the specified name
- Lock::Acquire() — wait (block) until the lock is unlocked; then lock it
- Lock::Release() — unlock the lock; then wake up (signal) any threads waiting on it in Lock::Acquire()

Can be implemented:

- Trivially by binary semaphores (create a private lock semaphore, use P and V)
- By lower-level constructs, much like semaphores are implemented

Example of using locks for mutual exclusion (here, “milk” is a lock):

<table>
<thead>
<tr>
<th>Thread A</th>
<th>Thread B</th>
</tr>
</thead>
<tbody>
<tr>
<td>milk-&gt;Acquire();</td>
<td>milk-&gt;Acquire();</td>
</tr>
<tr>
<td>if (noMilk)</td>
<td>if (noMilk)</td>
</tr>
<tr>
<td>buy milk;</td>
<td>buy milk;</td>
</tr>
<tr>
<td>milk-&gt;Release();</td>
<td>milk-&gt;Release();</td>
</tr>
</tbody>
</table>

- The test in threads/threadtest.cc should work exactly the same if locks are used instead of semaphores

Locks (cont.)

Conventions:

- Before accessing shared data, call Lock::Acquire() on a specific lock
  - Complain (via ASSERT) if a thread tries to Acquire a lock it already has
- After accessing shared data, call Lock::Release() on that same lock
  - Complain if a thread besides the one that Acquired a lock tries to Release it
Locks vs. Condition Variables

- Consider the following code:

```cpp
Queue::Add() {
    lock->Acquire();
    add item
    lock->Release();
} return item;
```

- Consider the following code:

```cpp
Queue::Remove() {
    lock->Acquire();
    if item on queue
    remove item
    lock->Release();
} return item;
```

- Queue::Remove will only return an item if there’s already one in the queue.

- If the queue is empty, it might be more desirable for Queue::Remove to wait until there is something to remove.

- Can’t just go to sleep — if it sleeps while holding the lock, no other thread can access the shared queue, add an item to it, and wake up the sleeping thread.

- Solution: condition variables will let a thread sleep inside a critical section, by releasing the lock while the thread sleeps.

Condition Variables

- Condition variables coordinate events

- Operations on condition variables (Nachos syntax):
  - Condition(*name) — create a new instance of class Condition (a condition variable) with the specified name
    - After creating a new condition, the programmer must call Lock::Lock() to create a lock that will be associated with that condition variable
  - Condition::Wait(conditionLock) — release the lock and wait (sleep); when the thread wakes up, immediately try to re-acquire the lock; return when it has the lock
  - Condition::Signal(conditionLock) — if threads are waiting on the lock, wake up one of those threads and put it on the ready list; otherwise do nothing

Condition Variables (cont.)

- Operations (cont.):
  - Condition::Broadcast(conditionLock) — if threads are waiting on the lock, wake up all of those threads and put them on the ready list; otherwise do nothing

- Important: a thread must hold the lock before calling Wait, Signal, or Broadcast

- Can be implemented:
  - Carefully by higher-level constructs (create and queue threads, sleep and wake up threads as appropriate)
  - Carefully by binary semaphores (create and queue semaphores as appropriate, use P and V to synchronize)
    - This sounds possible, but actually it does not work — details on why next time
  - Carefully by lower-level constructs, much like semaphores are implemented

Using Locks and Condition Variables

- Associated with a data structure is both a lock and a condition variable
  - Before the program performs an operation on the data structure, it acquires the lock
  - If it needs to wait until another operation puts the data structure into an appropriate state, it uses the condition variable to wait

- Unbounded-buffer producer-consumer:

```cpp
Lock *lk; int avail = 0;
Condition *c;
/* consumer */
while (1) {
    lk->Acquire( );
    lk->Acquire( );
    if (avail==0)
        lk->Release( );
    c->Signal(lk);
    avail--;
    lk->Release( );
}
/* producer */
while (1) {
    lk->Acquire( );
    lk->Acquire( );
    if (avail==0)
        c->Wait(lk);
    produce next item
    avail++;
    c->Signal(lk);
    lk->Release( );
    lk->Release( );
}
```