Operating Systems
CMPSCI 377
Spring 2017

Mark Corner
University of Massachusetts Amherst
What is a Monitor?

- Ties data and the synchronization operations together
- Monitors guarantee mutual exclusion, i.e., only one thread may execute a given monitor method at a time.
Monitors: A Formal Definition

- A Monitor defines a **lock** and zero or more **condition variables** for managing concurrent access to shared data.
  - The monitor uses the lock to insure that only a single thread is active in the monitor at any instance.
  - The lock also provides mutual exclusion for shared data.
  - Condition variables enable threads to go to sleep inside of critical sections, by releasing their lock at the same time it puts the thread to sleep.
Monitor Operations

- Encapsulates the shared data you want to protect.
- Acquires the mutex at the start.
- Operates on the shared data.
- Temporarily releases the mutex if it can't complete.
- Reacquires the mutex when it can continue.
- Releases the mutex at the end.
Monitors in Java

- It is simple to turn a Java class into a monitor:
  - Make all the data private
  - Make all methods synchronized (or at least the non-private ones)

```java
class Queue {
    private ...; // queue data
    public void synchronized add( Object item ) {
        put item on queue;
    }
    public Object synchronized remove() {
        if queue not empty {
            remove item;
            return item;
        }
    }
}
```
Condition Variables

• How can we change \texttt{remove()} to wait until something is on the queue?
  
  – Logically, we want to go to sleep inside of the critical section

  – But if we hold on to the lock and sleep, then other threads cannot access the shared queue, add an item to it, and wake up the sleeping thread

• \textbf{Solution}: use condition variables

  – Condition variables enable a thread to sleep inside a critical section

  – Any lock held by the thread is atomically released when the thread is put to sleep
Operations on CVs

- **Condition variable**: is a queue of threads waiting for something inside a critical section.

- Condition variables support three operations:
  1. **Wait(Lock lock)**: atomic (release lock, go to sleep), when the process wakes up it re-acquires lock.
  2. **Signal()**: wake up waiting thread, if one exists. Otherwise, it does nothing.
  3. **Broadcast()**: wake up all waiting threads

- Rule: thread must hold the lock when doing condition variable operations.
Condition Variables in Java

- Use `wait()` to give up the lock.
- Use `notify()` to signal that the condition a thread is waiting on is satisfied.
- Use `notifyAll()` to wake up all waiting threads.
- Effectively one condition variable per object.

```java
class Queue {
    private ...; // queue data

    public void synchronized Add( Object item ) {
        put item on queue;
        notify();
    }

    public Object synchronized Remove() {
        while queue is empty
            wait(); // give up lock and go to sleep
        remove and return item;
    }
}
```
Implicit Locks

public Object synchronized Remove() {
  (lock)
  while queue is empty
    (unlock)
    wait();
    (lock)
  remove and return item;
  (unlock)
}


Clicker Question #1

If you don’t know the ordering of how threads get run, what could happen with an `if` instead of a `while`?

(A) Always works fine

(B) Remove may sleep forever

(C) Items get duplicated

(D) “remove and return” may get run on an empty queue

```java
class QueueExample {
    private final Object emptyMarker = new Object();
    private final Object lock = new Object();

    public void synchronized Add( Object item ) {
        put item on queue;
        notify ();
    }

    public Object synchronized Remove() {
        if queue is empty
            wait (); // give up lock and go to sleep
        remove and return item;
    }
}
```
Answer on Next Slide
Mesa versus Hoare Monitors

What should happen when signal() is called?

- No waiting threads => the signaler continues and the signal is effectively lost (unlike what happens with semaphores).

- If there is a waiting thread, one of the threads starts executing, others must wait

  • **Mesa-style:** (Nachos, Java, and most real operating systems)
    - The thread that signals keeps the lock (and thus the processor).
    - The waiting thread waits for the lock.

  • **Hoare-style:** (most textbooks)
    - The thread that signals gives up the lock and the waiting thread gets the lock.
    - When the thread that was waiting and is now executing exits or waits again, it releases the lock back to the signaling thread.
Clicker Question #2

What if we used Hoare monitors instead?

(A) Always works fine

(B) Remove may sleep forever

(C) Items get duplicated

(D) “remove and return” may get run on an empty queue

```java
public void synchronized Add( Object item ) {
    put item on queue;
    notify ();
}
```

```java
public Object synchronized Remove() {
    if queue is empty
        wait (); // give up lock and go to sleep
    remove and return item;
}
```
Answer on Next Slide
In Class Exercise

- Build semaphores using monitors (Java)
public synchronized sem_signal(){
    value++
    notify();
}

public synchronized sem_wait(){
    while (value == 0)
        wait();
    value -= 1;
}

You can also implement monitors w/semaphores, but it is tricky…
Monitors in C++

- Monitors in C++ are more complicated.
- No synchronization keyword

  => The class must explicitly provide the lock, acquire and release it correctly

- Most portable library for threads is “pthreads”

  - C++11 also provides
    std:thread+mutex+condition_variable (quickly catching up to pthreads)
Monitors in C++: Example

class Queue {
   public:
   Add();
   Remove();

   private:
   pthread_cond_t cv;
   pthread_mutex_t mutex;
   // cv and mutex must be initialized
   // queue data();
};

Queue::Add() {
   pthread_mutex_lock(&mutex);
   put item on queue;  // ok to access shared data
   pthread_cond_signal(&cv);
   pthread_mutex_unlock(&mutex);
}

Queue::Remove() {
   pthread_mutex_lock(&mutex);
   while queue is empty
      pthread_cond_wait(&cv,&mutex)
   remove item from queue;
   pthread_mutex_unlock(&mutex);
   return item;
}
What about broadcast?

• We have only shown the use of signal thus far

• What happens if you replace signal with broadcast in the previous examples?

• There may be good reasons to use broadcast, consider the case of a memory allocator
Covering Conditions

Assume bytesLeft = 0. Then one thread allocate(100), another allocate(10). Then another thread free(50), which should it signal?

```c
// how many bytes of the heap are free?
int bytesLeft = MAX_HEAP_SIZE;

// need lock and condition too
cond_t c;
mutex_t m;

void * allocate(int size) {
    Pthread_mutex_lock(&m);
    while (bytesLeft < size)
        Pthread_cond_wait(&c, &m);
    void *ptr = ...
    bytesLeft -= size;
    Pthread_mutex_unlock(&m);
    return ptr;
}

void free(void *ptr, int size) {
    Pthread_mutex_lock(&m);
    bytesLeft += size;
    Pthread_cond_signal(&c);
    Pthread_mutex_unlock(&m);
}
```
Summary

• Monitor wraps operations with a mutex

• Condition variables release mutex temporarily

• Java has monitors built into the language

• C++ does not provide a monitor construct, but monitors can be implemented by following the monitor rules for acquiring and releasing locks

• It is possible to implement semaphores w/monitors and vice versa…