

Monitors

ICS432 - Fall 2008 Concurrent and High-Performance Programming

Henri Casanova (henric@hawaii.edu)

What Was Bad with Semaphores

- There are three acknowledged problems with Semaphores
- Issue #1: low-level and error prone
 - Forget to call V() and get a deadlock
 - Forget to call P() and get race conditions
- Issue #2: confused conceptually distinct operations
 - Mutual exclusion
 - Signaling
 - This can be considered a good thing at first glance
- Issue #3: unstructured and not modular
 - Synchronization code is dispersed all over the place
 - One must constantly go back and forth between the code of different threads to make sure everything works, which is difficult for large programs

Monitors to the Rescue

- In the late 70s, Brinch-Hansen proposed the concept of a **Monitor**.
 - Popularized by Hoare (1974)
- A monitor is really an abstract data type representing a shared "resource"
 - e.g., a class/object
- It is a construct of a programming language
 - Or can be emulated
- A Semaphore **controls access** to a shared resource
- A Monitor **encapsulates** a shared resource
 - Implements a shared data structure
 - Implements the coarse-grained and atomic operations to manipulate it (e.g., methods)
- Java provides Monitors as its basic mechanism for thread synchronization

Monitors

- There is nothing magical here, we still need the two basic functionalities:
 - mutual exclusion
 - condition variables for signaling
- But monitors constrain several things
 - Condition variables are not visible outside the monitor
 - They are hidden/encapsulated
 - One interacts with them via special monitor operations
 - Mutual exclusion is implicit
 - Monitor operations execute by definition in mutual exclusion
- We will see that these apparently innocuous properties make writing concurrent code MUCH less error-prone

Mutual Exclusion with Monitors

- By definition: at most *one* instance of *one* monitor operation can be active at a time
- The monitor maintains an **entry queue**
 - If a thread is executing a monitor method and another thread calls a (possibly different) monitor method, the second thread blocks and is placed in the entry queue
 - When the first thread is done, the first thread of the entry queue is allowed to execute the monitor's method
 - The entry queue is typically FIFO

Easy Critical Sections

- Critical Sections can be basically implicit
 - Keep all shared variables and data structures private inside the monitor
 - Implement all inter-thread data sharing and communication in monitor methods
 - And voila: All accesses to shared state are now guaranteed to be in mutual exclusion

Monitors and Cond. Variables

- A monitor encapsulates condition variables
 - A cond. variable maintains a queue of blocked threads
- The expected operations are defined:
 - wait(cond): block and wait
 - signal(cond): wake up the first thread in cond's queue of blocked threads
 - If the queue is empty, nothing happens
 - Different from a semaphore's V() operation
 - which always increments the semaphore
 - signal_all(cond): wake up all thread(s) in cond's queue of blocked threads
 - If the queue is empty, nothing happens
 - Different from a semaphore's V() operation
 - which always increments the semaphore

Monitor Declaration

- In pseudo-code, one typically writes monitors as follows

```
monitor MyMonitor {
    <declaration of variables>; // state and
                                // private cond. variables

    <initialization code>;
    <methods>; // operations done
                // in mutual exclusion
}
```

- We will see how it's done in Java in the next lecture

Example: 1-slot Prod/Cons

```
monitor ProdCons {
    cond_var full, notfull;
    int buffer;
    bool beginning = true;

    void produce(int x) {
        if (!beginning) {
            wait(notfull);
            beginning = false;
        }
        buffer = x;
        signal(full);
    }
    int consume() {
        int content;
        wait(full);
        content = buffer;
        signal(notfull);
        return content;
    }
}
```

```
ProdCons m; // the monitor

void producer() {
    while(true) {
        m.produce(generate_content());
    }
}

void consumer() {
    while(true) {
        process(m.consume());
    }
}
```

- Once you have defined the monitor "class", the producer/consumer code is completely trivial

So What???

- What we gain with monitors: **a higher level of abstraction**
 - Nothing fundamental, just easier to write correct code with
- If you have semaphores you can easily emulate monitors
 - One binary semaphore for mutual exclusion of all calls to all methods
 - Condition variables emulated by signaling semaphores
- If you have monitors you can easily emulate semaphores
 - Upcoming (very easy) programming assignment exercise
- In fact, the "power" of semaphores, monitors, cond. variable + locks are the same
 - There is nothing you can do with one that you cannot do with the others
- It's just a matter of what you like to program with
- Monitors are most likely the best/easiest way, and that's why "hand-holding" Java provides them as sole mean of thread synchronization
- But with Pthreads you see a lot of programmers use locks, cond. variables, and semaphores