More Synchronization; Concurrency in Java

CS 475, Spring 2018
Concurrent & Distributed Systems
Review: Semaphores

- Synchronization tool that provides more sophisticated ways (than Mutex locks) for process to synchronize their activities.
- Semaphore S – integer variable
- Can only be accessed via two indivisible (atomic) operations
  - `wait()`: consumes a resource (once available)
  - `signal()`: release a resource
Review: Semaphores

• Implementation of wait and signal. Starting value is # of permits

    wait(S):
    S.value--;
    if (S.value < 0) {
        enqueue(this,S.L);
        block();
    }

    signal(S):
    S.value++;
    if (S.value <= 0) {
        Thread toWake = pop(s.L);
        wakeup(toWake);
    }
Review: Bounded Buffer Example

Example: buffer can only hold 2 items

- **mutex = 0**: Makes sure buffer is viewed atomically
- **full = 2**: Represents number of full slots, tracks threads waiting to add more (if totally full)
- **empty = 0**: Represents number of empty slots, tracks threads waiting to read (if empty)
Review: Bounded Buffer Pseudocode

```plaintext
do {
    /* produce an item in next_produced */
    wait(empty);
    wait(mutex);
    /* add next produced to the buffer */
    signal(mutex);
    signal(full);
} while (true);

Producer

/* add next produced to the buffer */
signal(mutex);
signal(full);

Consumer

/* remove an item from buffer to next_consumed */
signal(mutex);
signal(empty);
/* consume the item in next consumed */

} while (true);
```

- **mutex** = 1: Makes sure buffer is viewed atomically
- **full** = 0: Represents number of full slots, tracks threads waiting to add more (if totally full)
- **empty** = 2: Represents number of empty slots, tracks threads waiting to read (if empty)
Review: Dining Philosophers

- Philosophers spend their lives alternating thinking and eating
- Don’t interact with their neighbors, occasionally try to pick up 2 chopsticks (one at a time) to eat from bowl
- Need both to eat, then release both when done
- In the case of 5 philosophers
- Solution: everybody acquires semaphores in the same order (even # seats pick up left first, right # seats pick up right first)
Announcements

• HW1 Discussion
  • Please complete brief poll on socrative.com (class CS475)
• Reminder: HW2 is out
Synchronization: The Sleeping Professor

- Professor Sleepy holds his office hours, and in between students, likes to take a nap
- Professor Sleepy has stressed to his students that if they see him sleeping - just wake him up, and he will be happy to help them
- Sometimes, there are a lot of students who want to see Professor Sleepy. If Professor Sleepy is busy when a student comes, that new student will go sit in the open area at the end of the hall
- When Professor Sleepy finishes with a student, he checks the waiting area, if there are no students, he takes a nap
The Sleeping Professor

Hallway

Students:
If Prof is busy, go to waiting area
If Prof is sleeping, wake Prof

Prof Sleepy
Talks to student
When done, goes to check waiting area
If no students: take nap

Waiting Area
Activity: The Sleeping Professor

- Go on socrative.com, class CS475
- Pair up in groups of 2-4
- There is a problem with this protocol. Discuss in your group and share on Socrative
High-level synchronization mechanisms

- Semaphores are a very powerful mechanism for process synchronization, but they are a low-level mechanism
- Several high-level mechanisms that are easier to use have been proposed
- Monitors (Condition variables and locks)
- Read/Write Locks
- Java and Pthreads provide both semaphores and high-level synchronization mechanisms
- NOTE: high-level mechanisms easier to use but equivalent to semaphores in power
Locking in Java

• Most locks are *reentrant*: if you hold it, and ask for it again, you don’t have to wait (because you already have it)

• Basic primitives:
  • synchronized{}
  • wait
  • notify

• Plus…
  • Lock API… lock.lock(), lock.unlock()
  • The *preferred* way
Synchronized methods in Java

public synchronized static void increment()
{
   i = i + 1;
}

Result: Before entering increment(), thread gets a lock on the Class object of increment()
wait and notify()

• Two mechanisms to enable coordination between multiple threads using the same monitor (target of synchronized)

• While holding a monitor on an object, a thread can wait on that monitor, which will temporarily release it, and put that thread to sleep

• Another thread can then acquire the monitor, and can notify a waiting thread to resume and re-acquire the monitor
wait and notify() example

```java
public class BlockingQueue<T> {

    private Queue<T> queue = new LinkedList<T>();
    private int capacity;

    public BlockingQueue(int capacity) {
        this.capacity = capacity;
    }

    public synchronized void put(T element) throws InterruptedException {
        while (queue.size() == capacity) {
            wait();
        }
        queue.add(element);
        notify(); // notifyAll() for multiple producer/consumer threads
    }

    public synchronized T take() throws InterruptedException {
        while (queue.isEmpty()) {
            wait();
        }
        T item = queue.remove();
        notify(); // notifyAll() for multiple producer/consumer threads
        return item;
    }
}
```

Only one thread can be in put or take of the same queue
Synchronized methods in Java

```java
public synchronized static void increment()
{
    i = i + 1;
}
```

Result: Before entering `increment()`, thread gets a lock on the Class object of `increment()`

```java
public synchronized static void incrementOther()
{
    j = j + 1;
}
```

Result: Before entering `incrementOther()`, thread gets a lock on the Class object of `incrementOther()`

Problem?
Synchronized blocks in Java

• Can also use *any* object as that monitor

```java
static Object someObject = new Object();
public static void increment()
{
    synchronized(someObject){
        i = i + 1;
    }
}

static Object someOtherObject = new Object();
public static void incrementOther()
{
    synchronized(someOtherObject){
        j = j + 1;
    }
}
```

Now, two different threads could call `increment()` and `incrementOther()` at the same time
Java Lock API

- **Synchronized** gets messy: what happens when you need to synchronize many operations? What if we want more complicated locking?
- **ReentrantLock**: same semantics as synchronized
- **ReadWriteLock**: allows many readers simultaneously, but writes are exclusive

```java
static ReentrantLock lock = new ReentrantLock();
public static void increment()
{
    lock.lock();
    try{
        i = i + 1;
    } finally{
        lock.unlock();
    }
}
```
Java Lock API

```java
static ReadWriteLock lock = new ReentrantReadWriteLock();
static int i;
public static void increment()
{
    lock.writeLock().lock();
    try{
        i = i + 1;
    } finally{
        lock.writeLock().unlock();
    }
}
public static int getI()
{
    lock.readLock().lock();
    try{
        return i;
    } finally{
        lock.readLock().unlock();
    }
}
```
Locking Granularity

• BIG design question in writing concurrent programs: how many locks should you have?
• Example: Distributed filesystem
  • It would be *correct* to block all clients from reading *any* file, when one client writes a file
  • However, this would not be performant at all!
  • It would be much better to instead lock on *individual files*
• *More locks -> more complicated semantics and tricky to avoid deadlocks, races*
Assignment 2 Discussion

Roadmap

• Next week: More concurrency patterns, then networks!