Synchronized Methods of Old Versions of Java

Older versions of Java (pre-1.5) do not have lock objects. Instead, every object has a lock that behaves like a ReentrantLock. If the lock is available, it is acquired when a synchronized method is called. A synchronized method is declared as

```java
public synchronized void push(Object item)
{
    // code for the method goes here
}
```

and is synchronized on the lock of its implicit argument (this)
Synchronized Methods

Synchronized instance methods allow at most one thread to run any of the object’s synchronized methods at any time.

Synchronized methods are simpler but less flexible.

The Account class would be defined as follows if we use synchronized methods...

```java
public class SyncAccount {
    private double balance;

    public synchronized void deposit(double amount) {
        double tempBalance = balance;
        System.gc(); // run an expensive process
        balance = tempBalance + amount;
    }

    public double getBalance() {
        return balance;
    }
}
```
Advanced Synchronization

Suppose we have a producer that produces items for a consumer to consume.

The producer and consumer will run on different threads.

The producer will place an item into a buffer where it will be retrieved by the consumer. The buffer will store only a single item:

- producer produces an item and places it in the buffer; the producer must wait if the buffer is full (buffer stores only one item)
- consumer removes the item from the buffer; the consumer must wait if the buffer is empty (there's nothing to consume)

```
Buffer Class: First attempt

public class BadBuffer {
    private int currentItem;
    private boolean full = false;
    private Lock lock = new ReentrantLock();

    public void add(int item) {
        lock.lock();
        try {
            // wait if buffer is full
            while (full);
            currentItem = item;
            full = true;
        }
        finally {
            lock.unlock();
        }
    }
}
```
Buffer Class: First attempt (cont)

```java
public int remove() {

    lock.lock();
    try {
        // wait if buffer is empty
        while (!full) ;
        full = false;
        return currentItem;
    } finally {
        lock.unlock();
    }
}
```

Deadlock

There are many pitfalls that one must be aware of when programming with threads.

**Deadlock** is one of them.

Deadlock occurs when two or more threads are unable to make progress because they are each waiting for each other to do something.
BadBuffer Problems

BadBuffer provides mutual exclusion:
add and remove cannot be executed at the same time

But, there is a possibility for deadlock
two or more threads are waiting for each other to release some locks; none can make any progress

Suppose buffer is empty and consumer executes remove(). What will happen?

Synchronization Using Conditions

To resolve this problem we should use condition objects

A condition object allows a thread to release a lock temporarily, so another thread can get that lock and run

Each condition object belongs to a lock object and is created as follows:

```java
Condition myCondition = lock.newCondition();
```
Synchronization Using Conditions

A condition object implements the `Condition` interface that includes:

- `await()`: the current thread releases the associated lock
  the current thread moves to the `waitblocked` state until another thread calls `signal()` or `signalAll()` on this condition

- `signal()` or `signalAll()`
  causes one or all of the threads that are blocked waiting on the condition to move to the `runnable` state
  these threads will compete to get the lock again
  one of them will get the lock and continue to run

Buffer Class: Using Conditions

```java
public class GoodBuffer {
    private int currentItem;
    private boolean full = false;
    private Lock lock = new ReentrantLock();
    private Condition bufferEmpty = lock.newCondition();
    private Condition bufferFull = lock.newCondition();

    public void add(int item) {
        lock.lock();
        try {
            while (full) { // wait for buffer to be empty
                bufferEmpty.await();
                currentItem = item;
                full = true;
                bufferFull.signalAll(); // notify consumers
            }
        } catch (InterruptedException e) {
        }
        finally {
            lock.unlock();
        }
    }
}
```
Buffer Class (cont’d)

```java
public int remove() {
    int returnValue = 0;

    lock.lock();
    try {
        // wait for buffer to be full
        while (!full)
            bufferFull.await();
        full = false;
        returnValue = currentItem;
        bufferEmpty.signalAll(); // notify producers
    }
    catch (InterruptedException e) {
    }
    finally {
        lock.unlock();
    }

    return returnValue;
}
```

Dining Philosophers

A “Classical” Synchronization Problem devised by Edsger Dijkstra in 1965 (modified for the masses by Tony Hoare)

You have

5 philosophers sitting around a table
Spaghetti on the table
5 forks around the table
Each philosopher needs 2 forks to eat
Dining philosophers (cont)

Each philosopher goes in a cycle
   Think for a while
   Get 2 forks
   Eat for a while
   Put down the forks
   Repeat

Your task is to devise a scheme for the “Get 2 forks” step
Conclusion

We’ve discussed how to build programs with multiple threads.

To synchronize threads we can use Java’s primitives:
- lock objects
- condition objects

In CPSC 213 you will learn how to create independent processes on different processors and make them communicate/synchronize over a network.

Exercises

2nd Edition: Chapter 23, page 901
Exercises P23.1, P23.2, P23.7

3rd Edition: Chapter 20, page 883
Exercises P20.1, P20.2, P20.5

4th Chapter 20, page 836
Exercises P20.1, P20.2, P20.5
Appendix: Main Methods of class Thread

public Thread() --- Allocates a new Thread object
public Thread(Runnable target)

public final boolean isAlive() - Tests if this thread is alive
public static Thread currentThread() - Get reference to currently executing thread

public final String getName()
public final void setName(String name)
public final int getPriority()
public final void setPriority(int newPriority)

public void start() --- Causes thread to be scheduled; JVM calls its run() method
public void run() --- If thread was constructed using a separate Runnable object, then that Runnable object's run method is called; otherwise, this method does nothing.
public void interrupt() --- Interrupts this thread.
public final void join() --- waits until the thread to which it is applied has died
public static void sleep(long millis) --- puts currently executing thread to sleep
public static void yield() --- currently executing thread is temporarily paused and allow other threads to execute