Java Thread Synchronization

Terminology

- Sequential (serial) computing
    - One thread, instructions executed one at a time
- Concurrent computing
    - Multiple processes (or threads)
    - Instruction execution overlaps but is not simultaneous
    - Improves throughput (from system perspective) over sequential computing
- Parallel computing
    - Simultaneous execution
    - Ex threads on multi-core node or processes on multiple nodes

Synchronization

- Processes (and threads)
  - Coordinating activities to achieve some goal
- Data
  - Avoiding unintended modification (corruption) of data

No Problems

x = 0

Thread A

Read x
x = x - 3
Write x

Thread B

Read x
x = x + 2
Write x

Time

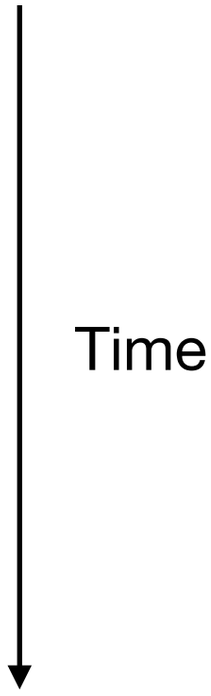
x = -1

These threads are synchronized or read/modify/write is atomic.

Lost Update Problem

x = 0

Thread A	**Thread B**
Read x	
	Read x
	x = x + 2
	Write x
x = x - 3	
Write x	
	x = -3



The x=x+2 update is lost.
Inter-leaving threads can leads to race conditions (the behavior (output) depends on a sequence or timing of events).

Critical Section

- A part of a program where concurrent access may result in unexpected behavior or non-deterministic results
    - Ex. A section of code that accesses a data structure or a hardware device
- Need to take steps to enforce mutually exclusive access
    - Avoid race conditions
- Programmers should design critical sections to be as short as possible.

Dining Philosophers Problem

- Classic problem illustrates synchronization issue presented by Edsger Dijkstra (1965)

- 5 silent philosophers sit around a table with 5 forks between them (1 fork between each pair of philosophers)

- Philosophers think or eat. Eating requires two forks.

- Must put down both forks after eating.

- Find an algorithm so that no philosopher starves.

Types of Locks

- A lock is a synchronization mechanism that enforces mutual exclusion for a critical section

- Semaphore
    - Invented by Edsger Dijkstra (early 1960s)
    - A variable or object that keeps track of how many of a resource are in use
- Binary Semaphore
    - A semaphore initialized to 1
- Monitor

Java Support for Mutual Exclusion

- synchronized statement
- synchronized method modifier
- java.util.Collections (some methods)
- java.util.concurrent.Semaphore class

synchronized statement

- Syntax: synchronized (*expression*) *statement*
  - *expression* must resolve to an object or an array
  - *statement* is the code of the critical section
- JVM does not execute critical section until it obtains an exclusive lock on *expression*
- JVM maintains exclusive lock until critical section is completed

synchronized example

```
public static void SortIntArray(int[] a) {
    synchronized(a) {
        // do the array sort here
    }
}
```

- This is synchronized so that some other thread can’t change elements of the array during the sort.

- At least not other threads that protect their changes to the array while synchronized.

Thread-safe Example

```
public class MyIntList {
    private int[] list;

    public MyIntList(int size) {
        list = new int[size];
    }

    public void sort() {
        synchronized(list) {
            // do the sort
        }
    }

    public void set(int index, int val) {
        synchronized(list) {
            list[index] = val;
        }
    }

    public int get(int index) {
        synchronized(list) {
            return list[index];
        }
    }

    public int size() {
        return list.length;
    }
}
```

synchronized method modifier

- The entire method is a critical section
- JVM obtains an exclusive lock
- synchronized **static** methods obtain lock on **class**
- synchronized **instance** methods obtain lock on **object**
- No other thread can execute any synchronized method
- Any other thread can execute non-synchronized methods

Still Thread-safe

```
public class MyIntList {
    private int[] list;

    public MyIntList(int size) {
        list = new int[size];
    }

    public synchronized void sort() {
        // do the sort
    }

    public synchronized void set(int index, int value) {
        list[index] = value;
    }

    public synchronized int get(int index) {
        return list[index];
    }

    public int size() {
        return list.length;
    }
}
```

java.util.concurrent.Semaphore

- A counting semaphore. For controlling access to a limited pool of resources

- Example. Given a pool of 5 resources and 10 threads, a counting semaphore ensures that no more than 5 resources are allocated at a time. If the pool is fully allocated, the next thread to request a resource is blocked until another thread returns its resource.

- Semaphore.acquire() — if count > 0, count = count - 1 else block
- Semaphore.release() — count++