**IX. Operation System**

1. **Concurrency**

* **Context Switch**
* process of storing and restoring the [state](https://en.wikipedia.org/wiki/State_(computer_science)) of a [process](https://en.wikipedia.org/wiki/Process_(computing)) or [thread](https://en.wikipedia.org/wiki/Thread_(computing)) so that execution can be resumed from the same point at a later time.
* enables multiple processes to share a single [CPU](https://en.wikipedia.org/wiki/Central_processing_unit)
* essential feature of a multitasking OS
* **Multithreading**
* A Java feature that allows concurrent execution of two or more parts of a program for maximum utilization of CPU.
* multiple threads try to access the same resources and may lead to unforeseen results
* **Synchronization**
* So we need to restrict access to shared resources.
* Synchronized method: public synchronized void foo(){}
* Synchronized blocks: synchronized(this) {}
* Parameter: an object whose lock associates with the monitor
* All synchronized blocks synchronized on the same object can only have one thread executing inside them at a time
* Why use synchronized block rather than method?
* Because sometimes it is preferable to synchronize only part of a method
* **Inner thread communication**
* Polling problem: waste CPU cycles and inefficient
* wait(): It tells the calling thread to give up the lock and go to sleep until some other thread enters the same monitor and calls notify().
* notify(): It wakes up one single thread that called wait() on the same object.
* notifyAll()-It wakes up all
* **Process VS Thread**
* Each process provides the resources needed to execute a program. All threads of a process share its memory space and system resources.
* Both processes and threads are independent sequences of execution. The typical difference is that threads (of the same process) run in a shared memory space, while processes run in separate memory spaces
* **Creates an instance of Thread**
* By implement the java.lang.Runnable interface
* Create a Runnable object. Override run() method.
* Create object of Thread by passing Runnable object as argument to Thread constructor.
* Call start() to invoke the thread object
* By extending the java.lang.Thread class
* Create a subclass extending Thread class. Override run() method.
* Call start() on the instance of the class itself
* **Extending Thread Class VS Implementing Runnable Interface**
* Implementing Runnable is preferred. Java does not support multiple inheritance.
* A class might only be interested in being runnable, so inheriting Thread class will cause overhead
* **join() VS sleep()**
* try {Thread.sleep(4000);

} catch (InterruptedException e) {return;

}

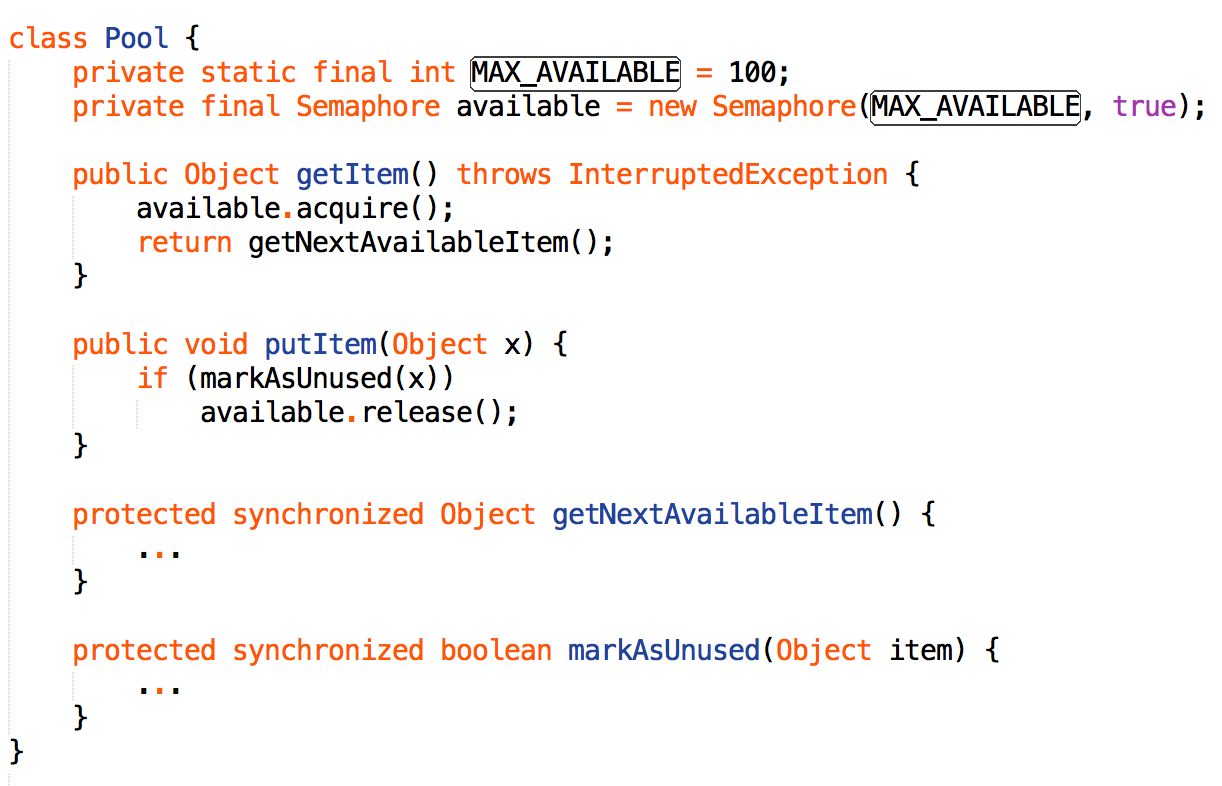
* Thread.sleep causes the current thread to suspend execution for a specified period. This is an efficient way of making processor time available to the other threads of an application.
* The join method allows one thread to wait for the completion of another
* join() will wait until the timeout expires or the thread finishes.
* sleep() will just wait for the specified amount of time unless interrupted
* **wait() VS sleep()**
* The fundamental difference is wait() is from Object and sleep() is static method of Thread.
* wait (and notify) must happen in a block synchronized on the monitor object

*synchronized (monitor) { synchronized (this) {*

*monitor.wait(); wait();*

*} }*

* The major difference is that wait() releases the lock, while sleep() cause currently executing thread to sleep for specified time.
* A wait can be "woken up" by another thread calling notify() on the monitor
* **start() VS run()**
* The idea behind the thread is to create new stack every time new thread starts running.
* Invoking the run() method from main thread, the run() method goes onto the current call stack rather than at the beginning of a new call stack
* **Starvation**
* Starvation describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress
* **Deadlocks & Prevention**
* Deadlock describes a situation where two or more threads are blocked forever, waiting for each other.
* **livelock** is similar to a deadlock, except that the states of the processes constantly change with regard to one another, with no progressing
* Avoid by removing any of the four:
* Mutual Exclusion: Limited access
* Hold and wait: Process hold and require src without release own resource
* No Preemption: A process cannot force another to release resource
* Circular Wait: each process is waiting on another resource in a circular chain
* **Thread synchronization doesn’t come for free**
* It takes time to acquire and release locks whenever a shared resource is accessed.
* This is why some libraries include both thread-safe and non-thread-safe classes, for instance StringBuffer and StringBuilder in Java
* **Thread Class**
* getId(): return identifier of Thread
* setName()/getName(): return thread’s name
* setPriority()/getPriority()
* interrupt(): if thread is blocked (wait, join, sleep), throw InterruptedException; o/w interrupt status will be set
* boolean interrupted()
* isAlive()
* join()/(long millis): wait at most milliseconds for this thread to die
* run(): the Runnalbe object(Constructor parameter)’s run method is called
* start(): cause thread to begin execution, JVM call run() of the thread
* static void sleep(long millis): won’t lose ownership of any monitors.
* toString(): return a string representation of thread’s name, priority, group
* **Monitors & Semaphores**
* A semaphore restricts the number of lock of a shared resource up to a maximum number.
* Two fundamental thread synchronization constructs are monitors and semaphores.
* Both are designed to restrict accessed from multiple threads. But monitors are simpler to use because they handle all details of lock [acquire()](https://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html#acquire()) and release().
* When using semaphores, each thread must be careful to release every lock it acquires, including under conditions in which it terminates unexpectedly
* **Semaphore Class**
* Constructor: Semaphore(int permits, Boolean fair);
* acquire()/(int): acquire one/more permit from semaphore
* Note that no synchronization lock is held when acquire() is called
* int availablePermits(): return number of permit available
* reducePermits(int): shrink the number of permits
* release()/(int): release one/more permits



* **Tutorial**
* <http://quiz.geeksforgeeks.org/synchronized-in-java/>
* <http://www.geeksforgeeks.org/inter-thread-communication-java/>
* <http://www.geeksforgeeks.org/producer-consumer-solution-using-threads-java/>
* **Producer-Consumer Problem**

