JAVA CONCURRENCY

Java Concurrent Programming

- Introduction to Concurrent Programming
- Java Multi-Threading Overview
- Java Concurrency API Overview

Concurrency

- Concurrency is the ability to run several parts of a program or several programs in parallel. If time consuming tasks can be performed asynchronously or in parallel, this improves the throughput and interactivity of your program.

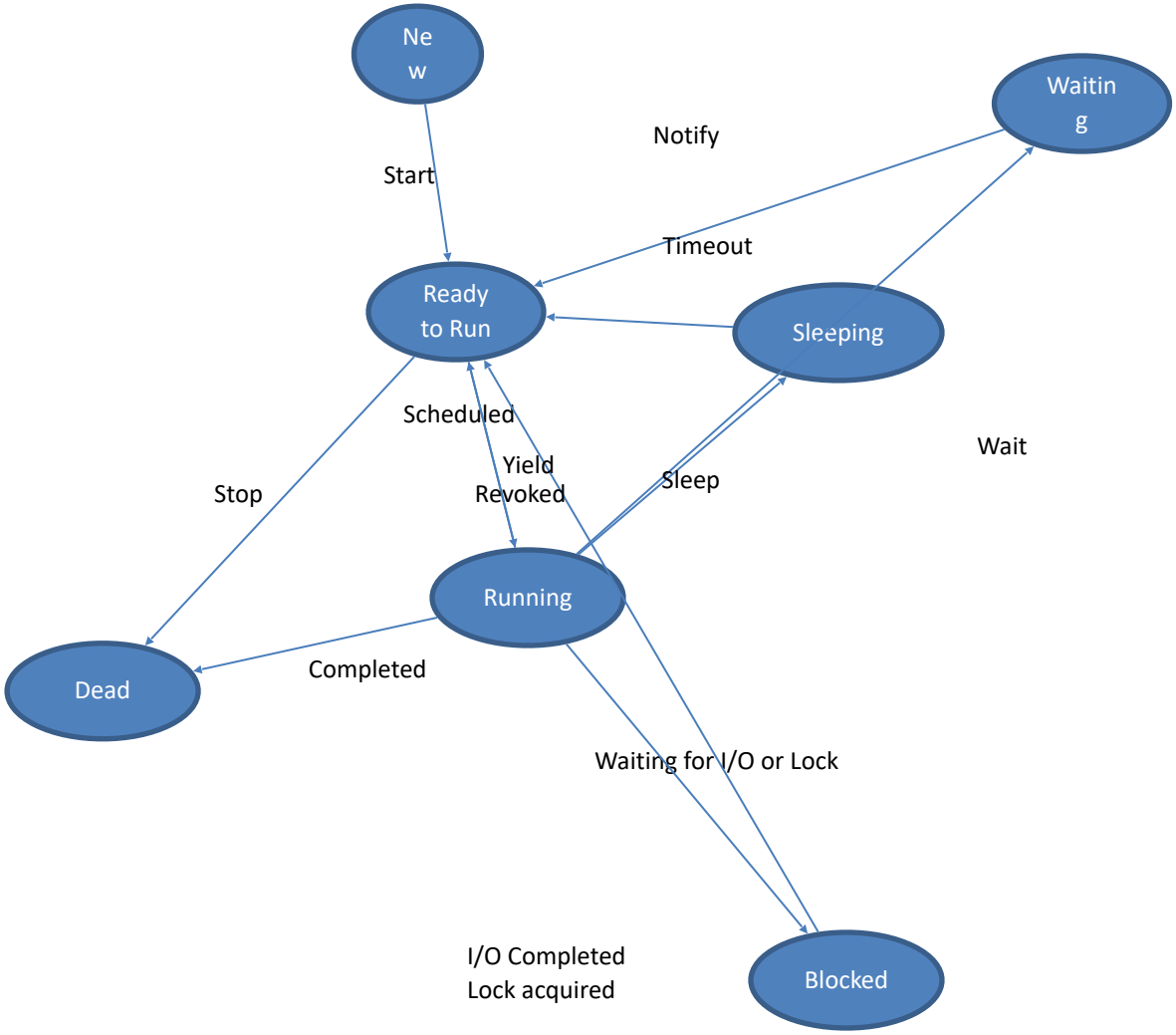
- A modern computer has several CPU's or several cores within one CPU. The ability to leverage these multi-cores can be the key for a successful high-volume application

Process vs. Threads

- **Process:** runs independently and isolated from other processes. It cannot directly access shared data in other processes. The resources of the process are allocated to it via the operating system, e.g. memory and CPU time.

- **Threads:** so called lightweight processes which have their own call stack but can access shared data. Every thread has its own memory cache or registers

Thread Lifecycle



Three ways to create threads

- Extend Thread
- Implement Runnable
- Implement Callable
  - Able to return an object
  - Supports typed exceptions

Constructs

- Wait
- Notify
- NotifyAll
- Interrupt
- Sleep
- Join
- Synchronized
- Volatile
- Yield

Deadlock

- Situation in which one thread is blocked by another thread and the second thread is blocked by the first thread, effectively blocking each other from doing any work.
- It can happen with more than two thread too (in a cyclic manner)

Thread Priority

- MIN_PRIORITY, MAX_PRIORITY, NORM_PRIORITY
- Default is the priority of the thread that is creating the new thread
- setPriority() can change the priority. It is set to the minimum of the passed value or the max priority of the group
- A thread with higher priority is run in preference to a thread with lower priority (platform dependent)

Daemon Threads

- Used for performing background work
- Has very low priority
- The JVM doesn’t wait for daemon threads to finish before exiting
- Finally blocks are not executed for daemon threads in case of JVM exit

Thread Groups

- Allows threads to be maintained as a group
- Can control the priority of threads
- Can interrupt a group of threads

Synchronized

- Each “Java object” has an associated lock
- Use synchronized(obj) { … } to acquire lock for duration of block
  - Locks automatically released
  - Locks are recursive
    - A thread can acquire the lock on an object multiple times
- Provides mutually exclusive access to code/data protected using the same lock

Three Aspects of Synchronization

- Atomicity
  - Prevention of interference through locking and mutual exclusion
- Visibility
  - Everything in one synchronized block occurs before and is visible to everything in a later block
- Ordering
  - Ensuring that you aren't surprised by the order in which statements are executed

Pitfalls

- Hold the lock only as long as absolutely necessary
- Obtaining a lock on an object doesn't prevent other threads from modifying it (only the thread that are trying to acquire the same lock are stopped)
- The locked object and the object being modified can be different (this is where locks come in)
- If not all locks are available then release the locks (avoids deadlock)
- Avoid holding locks while doing I/O, sleeping, calling external code
- Avoid unnecessary synchronization of methods
- Synchronize only the block that requires thread control

Volatile

- Alternative for synchronization
- Use in case of one-writer/many-reader
- Use in case of flags

Pitfalls

- Don’t use if there are multiple writers
- Don’t use in case where the current value depends on previous value (like incrementing)

New Constructs

- Thread Executor Framework
- Fork Join Framework
- Callable
- Futures
- Completable Futures
- Executors
- Thread Pool
- Concurrent Collections
- Locks
- Condition
- Atomic
- ThreadLocal
- Semaphores

Thank You!