Java Threads

Multitasking

- Perform multiple actions/tasks simultaneously on the machine
- **1. Process-Based Multitasking (Multiprocessing)**
  - processes are heavyweight and each process was allocated by a separate memory area.
  - Process is heavyweight , the cost of communication between processes is high and it takes a long time for switching between processes as it involves actions such as loading, saving in registers, updating maps, lists, etc
  - can not be controlled by JRE

Multitasking

2. Thread-Based Multitasking

- Thread are light weight and share same address space
- A thread is a single sequential flow of control within a program
- Switching between thread is quicker than then processes
- Threads make Utilization of multiprocessor architecture
- Thread synchronization functions could be used to improve inter-process communication.

Three ways to create Thread

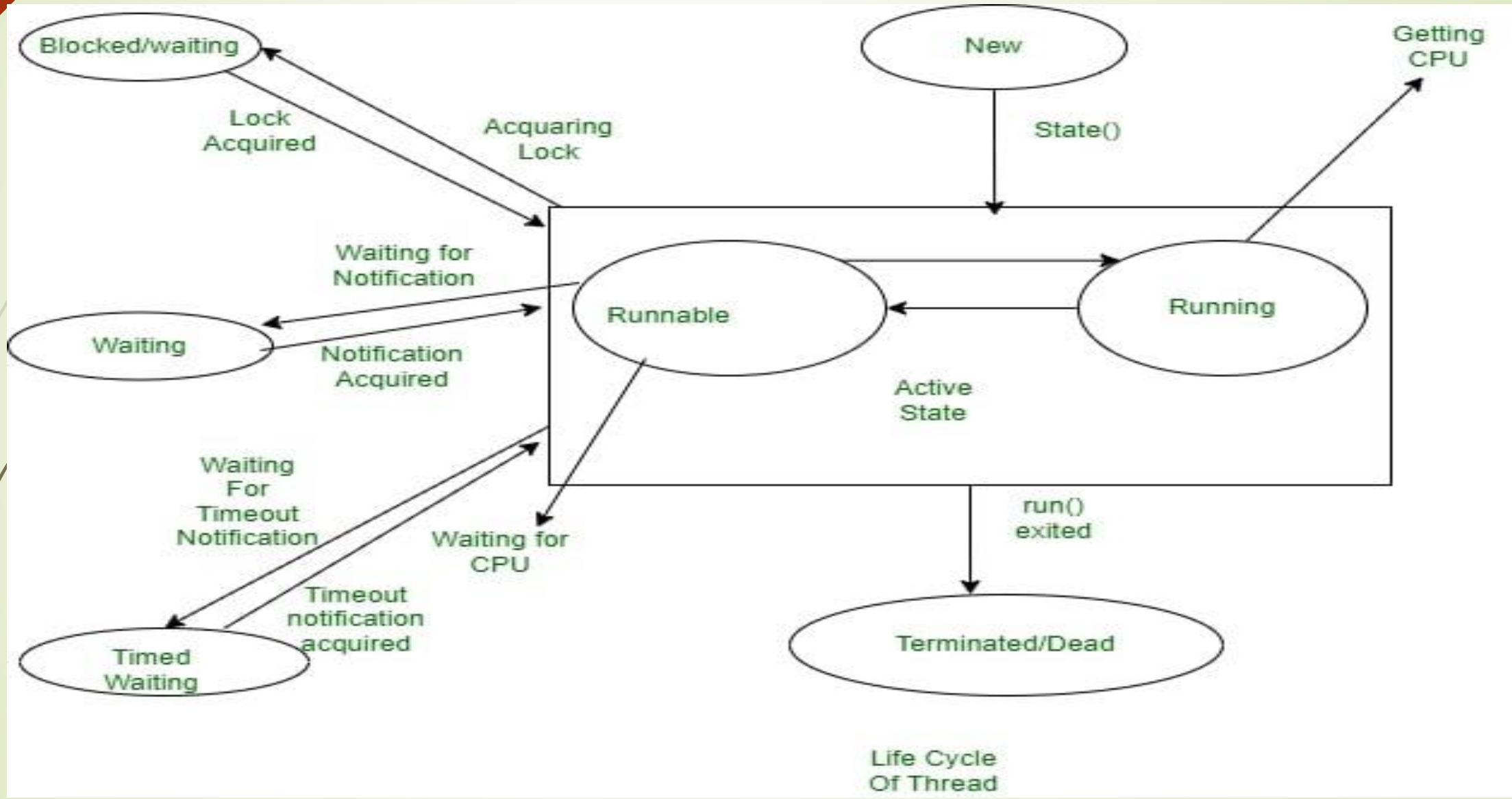
- **Threads can be created in Java using below 3 ways**

1. Extending Thread class
2. Implementing Runnable(Interface) and overriding **public void run()** method
3. Implementing Callable(Interface) and overriding call **public Object call() throws Exception**
4. Using **java.util.concurrent.Executor** for auto creation of threads

Main Thread

- Main Method in each and every Java Program, which acts as an entry point for the code to get executed by JVM
- In Multithreading Concept, Each Program has one Main Thread which was provided by default by JVM, hence whenever a program is being created in java, JVM provides the Main Thread for its Execution

Thread States and Life Cycle



Thread states

- **New State**
  - By default, a Thread will be in a new state, in this state, code has not yet been run.
- **Active State**
  - A Thread that is a new state by default gets transferred to Active state when it invokes the start() method, his Active state contains two sub-states:
  - **Runnable State:** The Thread is ready to run at any given time and it's the job of the Thread Scheduler to provide the thread time for the runnable state preserved threads. A program that has obtained Multithreading shares slices of time intervals which are shared between threads hence, these threads run for some short span of time and wait in the runnable state to get their schedules slice of a time interval.
  - **Running State:** When The Thread Receives CPU allocated by Thread Scheduler, it transfers from the "Runnable" state to the "Running" state. and after the expiry of its given time slice session, it again moves back to the "Runnable" state and waits for its next time slice.

Thread states

- **Waiting/Blocked State**
  - If a Thread is inactive but on a temporary time, then either it is at waiting or blocked state, for example, if there are two threads, T1 and T2 where T1 need to communicate to the camera and other thread T2 already using a camera to scan then T1 waits until T2 Thread completes its work, at this state T1 is parked in waiting for the state, and in another scenario, the user called two Threads T2 and T3 with the same functionality and both had same time slice given by Thread Scheduler then both Threads T1, T2 is in a blocked state. When there are multiple threads parked in Blocked/Waiting state Thread Scheduler clears Queue by rejecting unwanted Threads and allocating CPU on a priority basis.
- **Timed Waiting State**
  - Sometimes the longer duration of waiting for threads causes starvation, if we take an example like there are two threads T1, T2 waiting for CPU and T1 is undergoing Critical Coding operation and if it does not exit CPU until its operation gets executed then T2 will be exposed to longer waiting with undetermined certainty, In order to avoid this starvation situation, we had Timed Waiting for the state to avoid that kind of scenario as in Timed Waiting, each thread has a time period for which sleep() method is invoked and after the time expires the Threads starts executing its task.

Thread states

- **5. Terminated State**
  - A thread will be in Terminated State, due to the below reasons:
  - Termination is achieved by a Thread when it finishes its task Normally.
  - Sometimes Threads may be terminated due to unusual events like segmentation faults, exceptions…etc. and such kind of Termination can be called Abnormal Termination.
  - A terminated Thread means it is dead and no longer available
  - Dead or terminated thread can not be re-started. **IllegalThreadStateException** on starting dead thread

Thread blocked state

- Sleeping
- Join (waiting for threads to finish)
- Blocked to achieve lock (monitor)
- Blocked on I/O operation (System.out.read())
- Blocked on wait

Race Condition

- The situation where two or more threads compete for the same resource access.
- The sequence of threads in resource is accessed is significant, This is called race condition.
- A code section that create race conditions is called **critical section**.
- Critical section should be executed by only one thread at a time. Once one thread finishes execution then give chance to other.

Thread Synchronization

- Synchronization guarantee thread safety.
- Thread synchronization is needed if many thread use **SHARED RESOURCE**
- Synchronized method or synchronized block can be used to guard critical section.
- Synchronized keyword puts lock on Object (i.e shared resource or object)
- Synchronized block should be preferred over synchronized methods

Static and instance method Synchronization

- Threads calling instance synchronized two methods in the same class will y block each other if they are invoked using the same instance.
- Threads calling two static synchronized methods in the same class will block each other as lock in on Class.
- A static synchronized method and a instance synchronized method will not block each other
- The static method locks on a Class instance while the instance method locks on the **this** instance

Inter-thread communication

- Inter-thread communication in Java is a mechanism in which a thread is paused running in its critical section and another thread is allowed to enter (or lock) in the same critical section to be executed.
- **Methods of Object class (Inter thread communication)**

1. wait () method
   - Forces the executing thread to release lock or monitor & wait outside.
   - It throws for InterruptedException and IllegalMonitorStateException
   - Some other thread sends interrupt signal to the waiting thread.

Inter-thread communication

2. notify () method

- Un-blocks only 1 thread which is waiting on same object's monitor.
- May throw IllegalMonitorStateException , if the current thread is not the owner of a lock.

3. notifyAll () method

- Un-blocks **All** waiting threads on the same object's monitor.