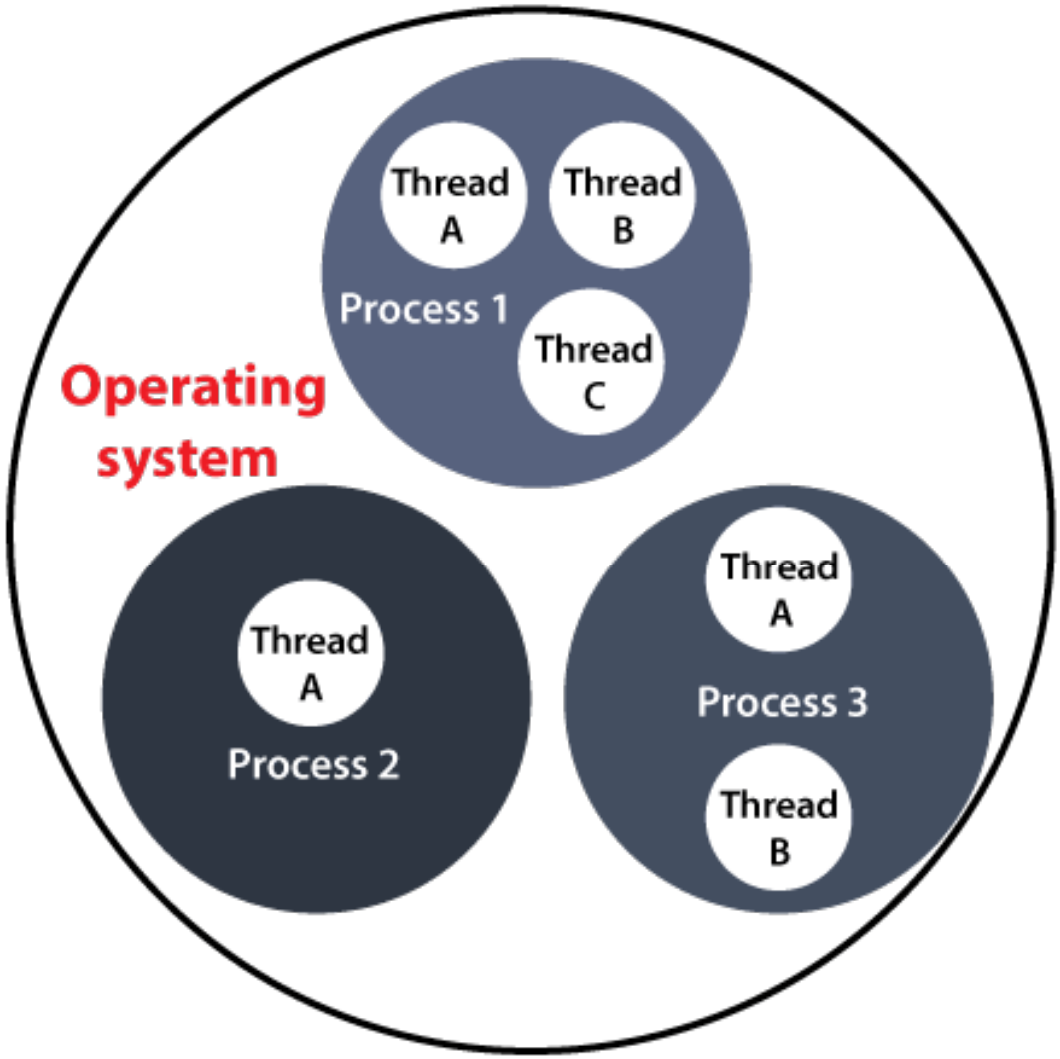
Java - MultiThreading

- A **Thread** is a very light-weighted process, or we can say the smallest part of the process that allows a program to operate more efficiently by running multiple tasks simultaneously.
- In order to perform complicated tasks in the background, we used the **Thread concept in Java**. All the tasks are executed without affecting the main program. In a program or process, all the threads have their own separate path for execution, so each thread of a process is independent.

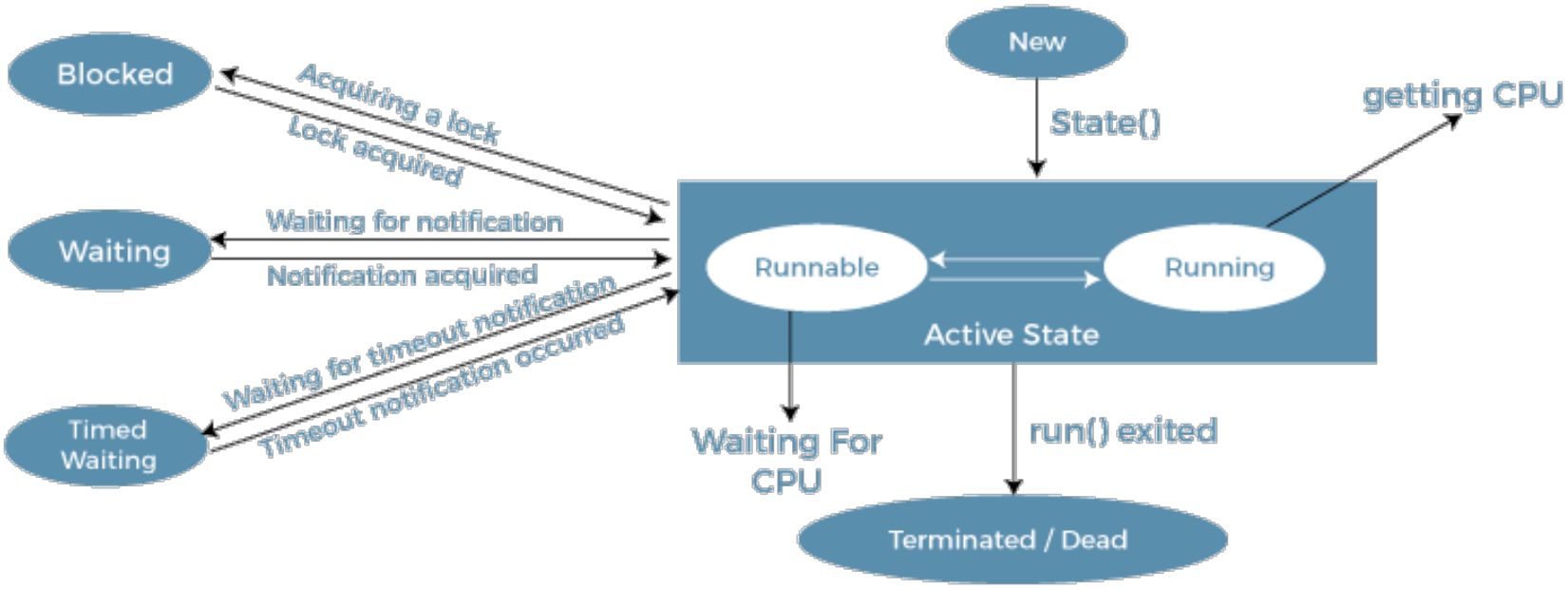
- Another benefit of using **thread** is that if a thread gets an exception or an error at the time of its execution, it doesn't affect the execution of the other threads. All the threads share a common memory and have their own stack, local variables and program counter. When multiple threads are executed in parallel at the same time, this process is known as **Multithreading**.


Thread
A
Thread
B
Process 1
Thread
C
Operating
system
Thread
A
Process 2
Thread
A
Process 3
Thread
B

Life cycle of a Thread (Thread States)

- The life cycle of the thread in java is controlled by JVM. The java thread states are as follows:
- New
- Active
- Blocked / Waiting
- Timed Waiting
- Terminated

Life cycle of a Thread (Thread States)



Life Cycle of a Thread

- **New:** Whenever a new thread is created, it is always in the new state. For a thread in the new state, the code has not been run yet and thus has not begun its execution.
- **Active:** When a thread invokes the start() method, it moves from the new state to the active state. The active state contains two states within it: one is **runnable**, and the other is **running**.
  - **Runnable:** A thread, that is ready to run is then moved to the runnable state. In the runnable state, the thread may be running or may be ready to run at any given instant of time. It is the duty of the thread scheduler to provide the thread time to run, i.e., moving the thread the running state.
    A program implementing multithreading acquires a fixed slice of time to each individual thread. Each and every thread runs for a short span of time and when that allocated time slice is over, the thread voluntarily gives up the CPU to the other thread, so that the other threads can also run for their slice of time. Whenever such a scenario occurs, all those threads that are willing to run, waiting for their turn to run, lie in the runnable state. In the runnable state, there is a queue where the threads lie.
  - **Running:** When the thread gets the CPU, it moves from the runnable to the running state. Generally, the most common change in the state of a thread is from runnable to running and again back to runnable.

- **Blocked or Waiting:** Whenever a thread is inactive for a span of time (not permanently) then, either the thread is in the blocked state or is in the waiting state.
- **Timed Waiting:** Sometimes, waiting for leads to starvation. To avoid such scenario, a timed waiting state is given to thread B. Thus, thread lies in the waiting state for a specific span of time, and not forever. A real example of timed waiting is when we invoke the sleep() method on a specific thread. The sleep() method puts the thread in the timed wait state. After the time runs out, the thread wakes up and start its execution from when it has left earlier.
- **Terminated:** A thread reaches the termination state because of the following reasons:
- When a thread has finished its job, then it exists or terminates normally.
- **Abnormal termination:** It occurs when some unusual events such as an unhandled exception or segmentation fault.

THE THREAD CLASS AND THE RUNNABLE INTERFACE

- To create a new thread, your program will either **extend Thread** or **implement the Runnable interface**.

- **Commonly used Constructors of Thread class:**
- Thread()
- Thread(String name)
- Thread(Runnable r)
- Thread(Runnable r, String name)

- **Commonly used methods of Thread class:**
- **public void run():** is used to perform action for a thread.
- **public void start():** starts the execution of the thread.JVM calls the run() method on the thread.
- **public void sleep(long milliseconds):** Causes the currently executing thread to sleep (temporarily cease execution) for the specified number of milliseconds.
- **public void join():** The *join*() *method* is used to hold the execution of currently running thread until the specified thread is dead(finished execution). The *join()* method allows one thread to wait for the completion of another.
- **public void join(long milliseconds):**used to wait for the thread on which it's called to be dead or wait for specified milliseconds.

- **public int getPriority():** returns the priority of the thread.
- **public int setPriority(int priority):** changes the priority of the thread.
- **public String getName():** returns the name of the thread.
- **public void setName(String name):** changes the name of the thread.
- **public Thread currentThread():** returns the reference of currently executing thread.

Creating thread by extending Thread Class

- **Extending the java.lang.Thread Class**
- Extend the Thread class in our class
- Then creating objects of class acts as threads
- Through the methods in this class, threads can be started, stopped, interrupted, named, prioritized, and queried regarding their current state

- **Overriding the run() Method**

After extending Thread, the next step is to override the run() method

public void run() { }

When a new thread is started, the entry point into the program is the run() method.

The first statement in run() will be the first statement executed by the new thread.

The new thread is considered to be alive from just before run() is called until just after run() returns, at which time the thread dies.

After a thread has died, it cannot be restarted.

- **Invoke the start() method**
- After the new thread is created, it will not start running until you call its start( ) method
- The start() method causes and not actually starts execution.
- It schedules the thread and when CPU scheduler picks this thread for execution then JVM calls the run() method to actually start execution.
- Thread can call start() method only once in a program.

```
class NewThread extends Thread{
public void run(){
for(int i=0;i<5;i++){
System.out.println("Value of i "+i);
}
}
}
class ExThread{
public static void main(String args[]){
NewThread t=new NewThread();
System.out.println(t.getName());
t.start();
}
}
```

Implementing Runnable interface

- Create a class that implements the interface and override the run() method
- Create an object of the class
- Create Thread Object
- Start execution of the Thread using start()

```
class NewThread implements Runnable{
public void run(){
for(int i=0;i<5;i++){
System.out.println("Value of i "+i);
}
}
}
class ImplementThread{
public static void main(String args[]){
NewThread nt=new NewThread();
Thread t=new Thread(nt);
t.start();
}
}
```

Thread Class vs. Runnable Interface

- Implementing the Runnable Interface defines the unit of work that will be executed in a thread
- When extending the Thread class, the derived class cannot extend any other base classes because java only allows single inheritance
- By implementing Runnable Interface, the class can still extend other base classes if necessary.
- By extending Thread, each of your threads has a unique object associated with it, whereas implementing Runnable, many threads can share the same runnable instance or it shares the same object to multiple threads.

- **Thread Scheduler in Java**
- **Thread scheduler** in java is the part of the JVM that decides which thread should run.
- There is no guarantee that which runnable thread will be chosen to run by the thread scheduler.
- The thread scheduler mainly uses preemptive or time slicing scheduling to schedule the threads.

Sleep method in java

```
class NewThread extends Thread{
public void run(){
for(int i=0;i<5;i++){
try{Thread.sleep(500);}catch(Exception e){}
System.out.println("Value of i "+i);
}}}
class ExThread{
public static void main(String args[]){
NewThread t1=new NewThread();
NewThread t2=new NewThread();
t1.start();
t2.start();}}
```

Priority of a Thread (Thread Priority)

- Each thread have a priority.
- Priorities are represented by a number between 1 and 10.
- In most cases, thread scheduler schedules the threads according to their priority (known as pre-emptive scheduling). But it is not guaranteed because it depends on JVM specification that which scheduling it chooses.

- public static int MIN_PRIORITY
- public static int NORM_PRIORITY
- public static int MAX_PRIORITY

✓ Default priority of a thread is 5 (NORM_PRIORITY).

✓ The value of MIN_PRIORITY is 1

✓ The value of MAX_PRIORITY is 10.

```
class One extends Thread {
 public void run()   {
       System.out.println(getName()+" "+getPriority());
    } }
class MainThread {
 public static void main(String args[]) {
 One t1=new One();
 One t2=new One();
 One t3=new One();
t1.setPriority(Thread.NORM_PRIORITY);
t2.setPriority(Thread.MIN_PRIORITY);
t3.setPriority(Thread.MAX_PRIORITY);
  t1.start();
  t2.start();
  t3.start();
  } }
```

Synchronization in Java

- Synchronization in Java is the capability to control the access of multiple threads to any shared resource.
- Java Synchronization is better option where we want to allow *only one thread to access the shared resource*.

  Why use Synchronization?
- The synchronization is mainly used to

✓ To prevent thread interference.

✓ To prevent consistency problem.

- Thread Synchronization
- There are two types of thread synchronization mutual exclusive and inter-thread communication.
- Mutual Exclusive

✓ Synchronized method.

✓ Synchronized block.

✓ Static synchronization.

- Cooperation (Inter-thread communication in java)

- Concept of Lock in Java
- Synchronization is built around an internal entity known as the lock or monitor.
- Every object has a lock associated with it.
- By convention, a thread that needs consistent access to an object's fields has to acquire the object's lock before accessing them, and then release the lock when it's done with them.

- Java Synchronized Method
- If you declare any method as synchronized, it is known as synchronized method.
- Synchronized method is used to lock an object for any shared resource.
- When a thread invokes a synchronized method, it automatically acquires the lock for that object and releases it when the thread completes its task.

- Synchronized Block in Java
- Synchronized block can be used to perform synchronization on any specific resource of the method.
- Suppose we have 50 lines of code in our method, but we want to synchronize only 5 lines, in such cases, we can use synchronized block.
- If we put all the codes of the method in the synchronized block, it will work same as the synchronized method.

- Static Synchronization
- If you make any static method as synchronized, the lock will be on the class not on object.