Chapter Three

Introduction; Threads Vs process, Multi-threads Concepts

Nearly every operating system supports the concept of processes -- independently running programs that are isolated from each other to some degree.

Threading is a facility to allow multiple activities to coexist within a single process. Most modern operating systems support threads, and the concept of threads has been around in various forms for many years. Java is the first mainstream programming language to explicitly include threading within the language itself, rather than treating threading as a facility of the underlying operating system.

Threads are sometimes referred to as *lightweight processes*. Like processes, threads are independent, concurrent paths of execution through a program, and each thread has its own stack, its own program counter, and its own local variables. However, threads within a process are less insulated from each other than separate processes are. They share memory, file handles, and other per-process state.

A process can support multiple threads, which appear to execute simultaneously and asynchronously to each other. Multiple threads within a process share the same memory address space, which means they have access to the same variables and objects, and they allocate objects from the same heap. While this makes it easy for threads to share information with each other, you must take care to ensure that they do not interfere with other threads in the same process.

The Java thread facility and API is deceptively simple. However, writing complex programs that use threading effectively is not quite as simple. Because multiple threads coexist in the same memory space and share the same variables, you must take care to ensure that your threads don't interfere with each other.

What is Process?

An instance of a computer program that is being executed consist of:

- An image of the executable machine code
- Memory
    - Executable code

- Process-specific data
- Call stack
- Heap to hold intermediate computation data

- Operating system descriptors of resources (file descriptors, data sources and sinks)
- Processor state or Context (register contents, physical memory addressing,..)

What is Thread?

- ✓ A thread is a single sequential flow of control within a program.
- ✓ Thread does not have its own address space but uses the memory and other resources of the process in which it executes.
- ✓ There may be several threads in one process
- ✓ The Java Virtual Machine (JVM) manages these and schedules them for execution.
- ✓ Code, data and files are shared by multiple threads and unique stack and registers
- ✓ The smallest sequence of programmed instructions that can be managed independently by a scheduler
  - Thread of execution
  - Lighter than process
  - Share resources

What is Multithreading?

Multithreading is a conceptual programming paradigm where a program (process) is divided into two or more subprograms (process), which can be implemented at the same time in parallel.
→ For example, one subprogram can display an animation on the screen while another may build the next animation to be displayed.
→ A program that contains multiple flow of control is known as multithreaded program. Since threads in java are subprograms of a main application program and share the same memory space, they are known as lightweight threads or lightweight processes.
→ 'Threads running in parallel' does not really mean that they actually run at the same time.
→Since all the threads are running on a single processor, the flow of execution is shared between the threads. The java interpreter handles the switching of control between the threads in such a way that it appears they are running concurrently.

Multithreading enables programmers to do multiple things at one time.
→ For example, we can send tasks such as printing into the background and continue to perform

some other task in the foreground.

→ Threads are extensively used in java-enabled browser such as HotJava. These browsers can download a file to the local computer, display a web page in the window, and output another web page to a printer and so on.

→ The ability of a language to support multithreads is referred to as concurrency.

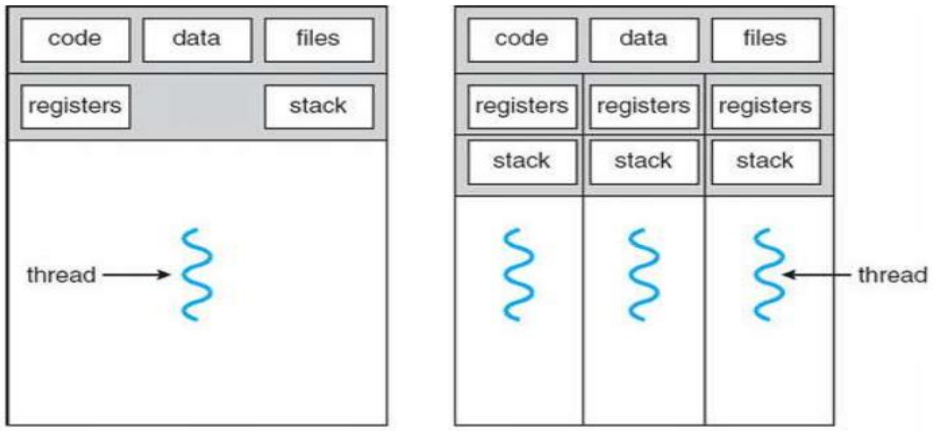


Fig above shows Single threaded and multi-threaded resp.

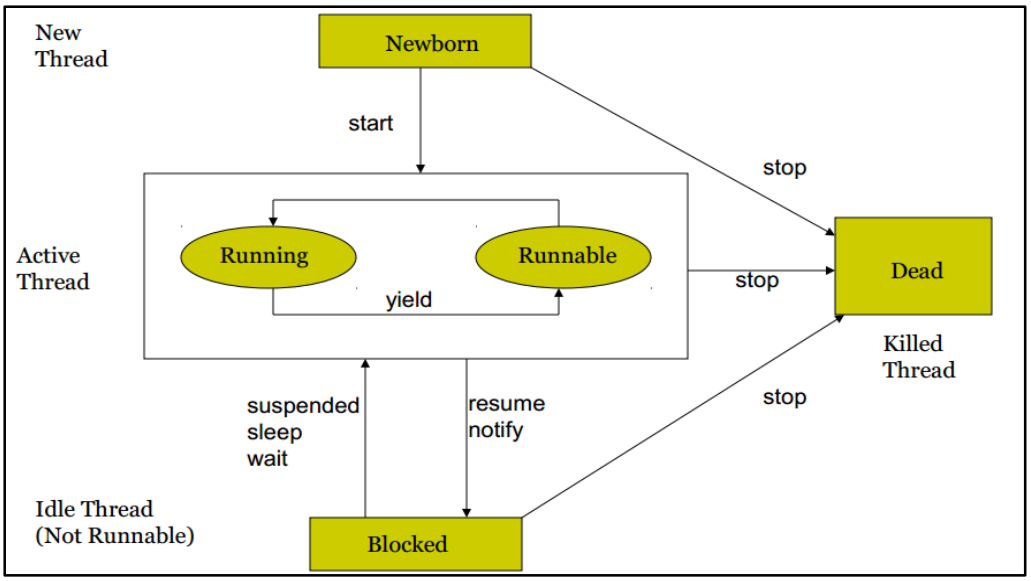
Why use threads?

- ➔ Make the User Interfaces more responsive
- ➔ Take advantage of multiprocessor systems
- ➔ Simplify modeling
- ➔ Perform asynchronous or background processing

Life Cycle of Thread

During the life time of a thread, there are many states it can enter. They include:

- Newborn state
- Runnable State
- Running State
- Blocked State
- Dead State



1. Newborn State

When we create a thread object, the thread is born and is said to be in newborn state. The thread is not yet scheduled for running. At this state, we can do only one of the following with it: Schedule it for running using start() method. Kill it using stop() method

➔ If scheduled, it moves to the runnable state. If we attempt to use any other method at this stage, an exception will be thrown.

2. Runnable state (start())

➔The runnable state means that the thread is ready for execution and is waiting for the availability of the processor.

➔ That is, the thread has joined the queue of threads that are waiting for execution.

➔ If all threads have equal priority, then they are given time slots for execution in round robin fashion. i.e. first-come, first-serve manner.

→The thread that relinquishes control joins the queue at the end and again waits for its turn. This process of assigning time to threads is known as *timeslicing.*

→ If we want a thread to relinquish control to another thread of equal priority before its turn comes, we can do so by using the yield() method.

3. Running State

➔Running means that the processor has given its time to the thread for its execution.

➔ The thread runs until it relinquishes control on its own or it is preempted by a higher priority thread.

➔ A running thread may relinquish its control in one of the following situations:

Suspend() and resume() Methods:-

This approach is useful when we want to suspend a thread for some time due to certain reason, but do not want to kill it.

Sleep() Method :-
This means that the thread is out of the queue during this time period. The thread re-enter the runnable state as soon as this time period is elapsed.
Wait() and notify() methods :- blocked
until certain condition occurs

4. Blocked State

- ⇨ A thread is said to be blocked when it is prevented from entering into the runnable state and subsequently the running state.
- ⇨ This happens when the thread is suspended, sleeping, or waiting in order to satisfy certain requirements.
- ⇨ A blocked thread is considered "not runnable" but not dead and therefore fully qualified to run again.

5. Dead State

➔A running thread ends its life when it has completed executing its run() method. It is a natural death.
➔ However, we can kill it by sending the stop message to it at any state thus causing a premature death to it.
➔A thread can be killed as soon it is born, or while it is running, or even when it is in "not runnable" (blocked) condition.

Creating a thread

Java defines two ways by which a thread can be created:

○ By extending the **Thread** class

○ By implementing the **Runnable** interface.

1. **Create Thread by Extending Thread Class:**

```
class ThreadDemo extends Thread {
private Thread t;
private String threadName;
ThreadDemo( String name){
threadName = name;
System.out.println("Creating " + threadName );
}public void run() {
```

```
System.out.println("Running " + threadName );
try {
for(int i = 4; i > 0; i--) {
System.out.println("Thread: " + threadName + ", " + i);
// Let the thread sleep for a while.
Thread.sleep(50);
}
} catch (InterruptedException e) {
System.out.println("Thread " + threadName + " interrupted.");
}
System.out.println("Thread " + threadName + " exiting.");
}
 public void start ()
{
System.out.println("Starting " + threadName );
if (t == null)
{
t = new Thread (this, threadName);
t.start ();
}
}
}
 public class TestThread {
public static void main(String args[]) {
ThreadDemo T1 = new ThreadDemo( "Thread-1");
T1.start();
ThreadDemo T2 = new ThreadDemo( "Thread-2");
T2.start();
}
}
```

2. **Create Thread by Implementing Runnable Interface:**

```
class RunnableDemo implements Runnable {
private Thread t;
private String threadName;
RunnableDemo( String name){
threadName = name;
System.out.println("Creating " + threadName );
}
public void run() {
System.out.println("Running " + threadName );
try {
for(int i = 4; i > 0; i--) {
System.out.println("Thread: " + threadName + ", " + i);
// Let the thread sleep for a while.
Thread.sleep(50);
}
} catch (InterruptedException e) {
System.out.println("Thread " + threadName + " interrupted.");
}S
ystem.out.println("Thread " + threadName + " exiting.");
}
public void start ()
{
System.out.println("Starting " + threadName );
if (t == null)
{
t = new Thread (this, threadName);
t.start ();
}
}
}
```

```
public class TestThread {
public static void main(String args[]) {
RunnableDemo R1 = new RunnableDemo( "Thread-1");
R1.start();
RunnableDemo R2 = new RunnableDemo( "Thread-2");
R2.start();
}
}
```

Thread Priorities:

Every Java thread has a priority that helps the operating system determine the order in which threads are scheduled.

Java thread priorities are in the range between MIN_PRIORITY a constant of 1 and MAX_PRIORITY
a constant of 10. By default, every thread is given priority NORM_PRIORITY a constant of 5. Threads with higher priority are more important to a program and should be allocated processor time before lower-priority threads. However, thread priorities cannot guarantee the order in which threads execute and very much platform dependent.

Synchronization

➔ When two or more threads need access to a shared resource, they need some way to ensure that the resource will be used by only one thread at a time. The process by which this is achieved is called synchronization.

➔ This can be done in two ways. First, a method can be synchronized by using the synchronized keyword as a modifier in the method declaration. When a thread begins executing a synchronized instance method, it automatically acquires a lock on that object.

➔ The lock is automatically relinquished when the method completes.

➔Only one thread has this lock at any time.

➔ Therefore, only one thread may execute any of the synchronized instance methods for that same object at a particular time.

Another way to synchronize access to common data is via a synchronized statement block. This has the following syntax:

```
synchronized(obj)
{
//statement block
}
```

Here, the **obj** is a reference to an object whose lock associates with the monitor that the synchronized statement represents.

Why use Synchronization

→ The synchronization is mainly used to

→ To prevent thread interference: **Interference** occurs when two operations, running in different threads, but acting on the same data

→ To prevent consistency problem: **Consistency Errors** occurs when different threads have inconsistent views of the shared data

*** The **Thread** class defines several methods that help to manage threads. Several of those used are:

Method	Meaning
getName()	Obtain a thread's name.
getPriority()	Obtain a thread's priority.
isAlive()	Determine if a thread is still running.
join()	Wait for a thread to terminate.
run()	Entry point for the thread.
sleep()	Suspend a thread for a period of time.
start()	Start a thread by calling its run method.