Threads and Multithreading

Prof. Anita Agrawal
BITS Pilani k.K.Birla Goa campus

Multithreading

Difference between Multiprogramming, Multiprocessing, multitasking and multithreading:

- Multiprogramming: One of the most important aspects of the operating system is to multiprogram. More than one program is being executed simultaneously. The CPU is allocated to one job at a time, until it is interrupted by some external factor or it goes for an I/O task.
  - Example: word, internet browser, excel, paint running simultaneously.
- Multiprocessing: Multiprocessing is the use of two or more CPUs (processors) within a single Computer system. Multiprocessing refers to the hardware (i.e., the CPU units) rather than the software (i.e., running processes).

- Multitasking: is a logical extension of multi programming. It is based on time sharing concept.

- Each process is assigned some specific quantum of time for which a process is meant to execute.

Program, process, thread

- Program is a code or a set of instructions to perform a particular task. It is a passive entity.
- A program in execution is a process. Process is an active entity
- A process code can be split up into small code segments
- Each code segment of the same process is called as a thread
- A thread is a light-weight process.
- If a process has multiple threads of control, it can perform more than one task at a time

Multithreading

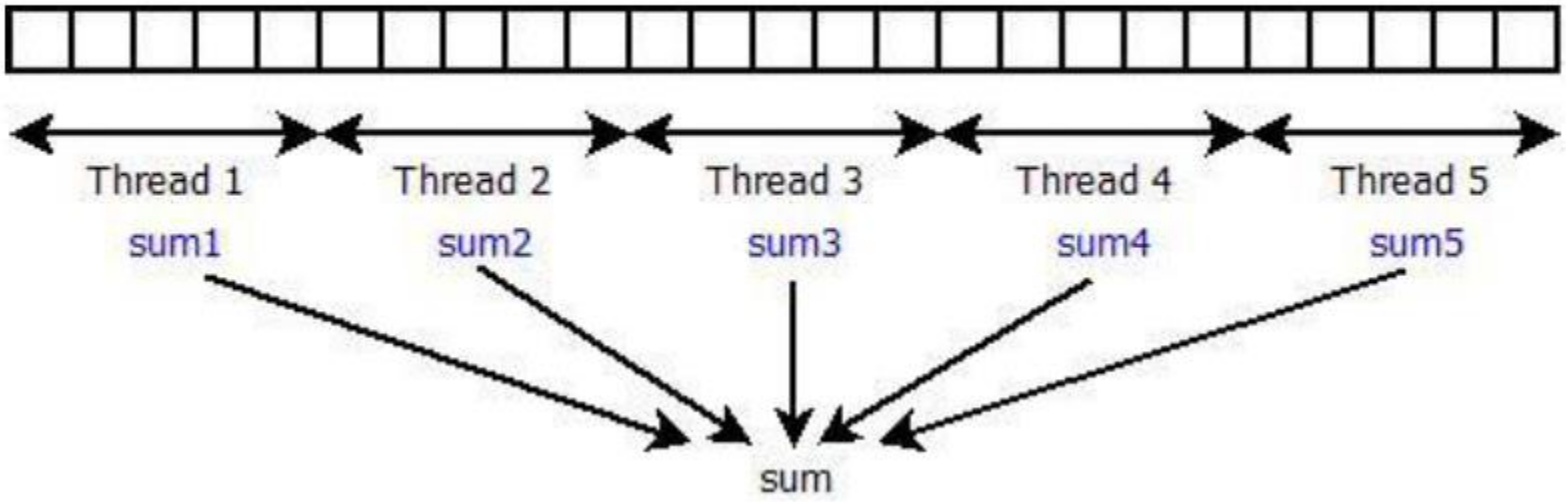
- Multithreading is a feature that allows concurrent execution of two or more threads of a process for maximum utilization of CPU.
- So, threads are light-weight processes within a process.
- A thread shares with other threads belonging to the same process the
  - Code section
  - Data section
  - Operating-system resources, such as open files

Example 1: A media player

- A media player, where
  - one thread is used for opening the media player,
  - one thread for playing a particular song and
  - one thread for adding new songs to the playlist.

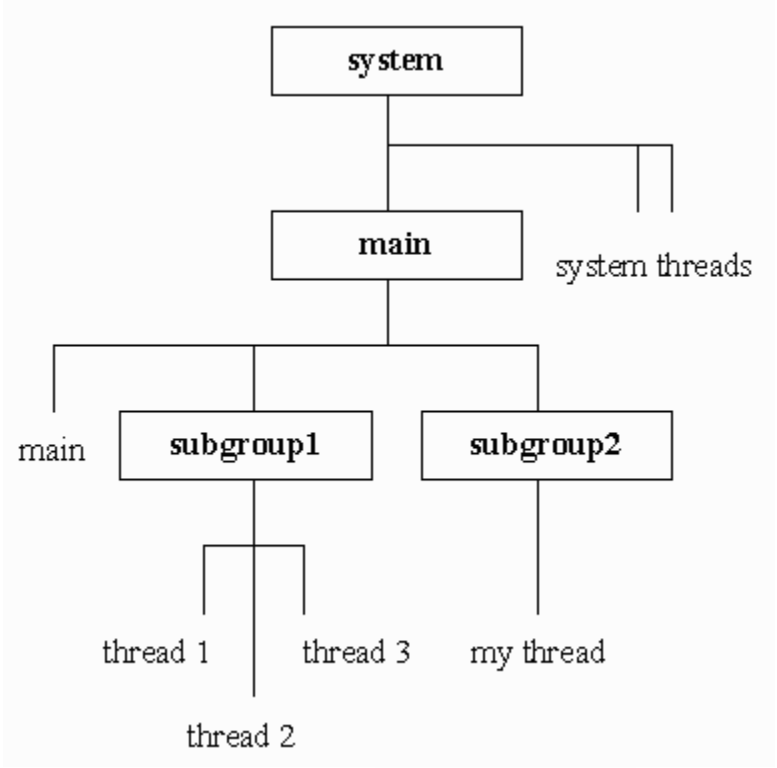
Example 2: Array sum

We can think of threads as child processes that share the parent process resources but execute independently.



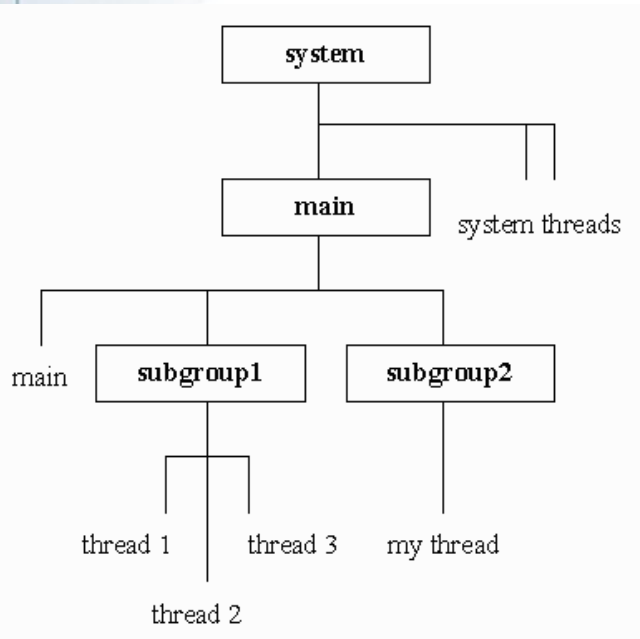
The Main Thread

- When a Java program starts up, one thread begins running immediately
- This is usually called the *mainthread* of your program
- The main thread is created automatically when your program is started
- The main thread is important for two reasons
  - It is the thread from which other “child” threads will be spawned
  - Often, it must be the last thread to finish execution because it performs various shutdown actions



An application's hierarchical thread-group structure begins with a main thread group just below the system thread group

Thread Group

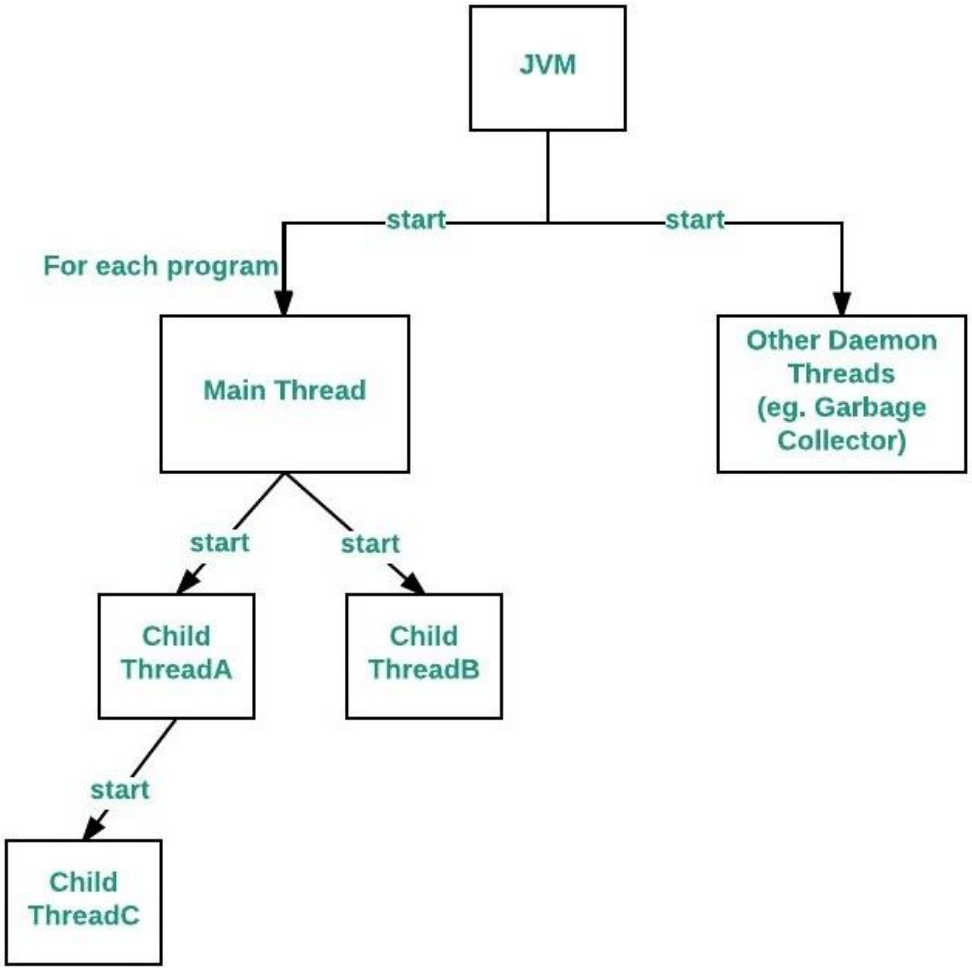


- Java requires every thread and every thread group—save the root thread group, system—to join some other thread group

- **System thread group:** The JVM-created system group organizes JVM threads that deal with object finalization and other system tasks, and serves as the root thread group.

- **Main Thread group**: JVM-created main thread group, which is system's subthread group (subgroup, for short).
  - main contains at least one thread that executes byte-code instructions in the main() method.
- Subgroup 1 and Subgroup 2 subgroups: Application-created subgroups

- Thread 1, Thread 2, and Thread 3: Subgroup 1's three application-created threads
- my thread: subgroup 2 group's one application-created thread


JVM
start
start
For each program
Main Thread
Other Daemon Threads (eg. Garbage Collector)
start
start
Child ThreadA
Child ThreadB
start
Child ThreadC

- You can control the main thread through a **Thread** object
- To do so, you must obtain a reference to it by calling the method **currentThread ( )**
  which is public static member of **Thread** class
- This method returns a reference to the thread in which it is called.
- Once you have a reference to the main thread, you can control it just like any other thread.

Main Thread Example

- A reference to the current thread (the main thread, in this case) is obtained by calling **currentThread( )**, and this reference is stored in the local variable **t**.
- Next, the program displays information about the thread.
- The program calls **setName( )** to change the internal name of the thread
- Information about the thread is then redisplayed
- Next, the loop counts from 0 with a pause of 1s in between two consecutive counts
- The thread execution can be paused by the **sleep( )** method

- The argument to **sleep( )** specifies the delay period in milliseconds
- We use **try/catch** block around the loop
- The **sleep( )** method in **Thread** might throw an **Interrupted Exception**
- This would happen if some other thread wanted to interrupt this sleeping one

MultiThreading in Java

- Multithreading in Java can be accomplished by two mechanisms:
  - **Implementing the Runnable interface**
  - **Extending the Thread class**

Implementing Runnable Interface

- Create a class that implements **Runnable**
- Instantiate an object of type **Thread**
- **Thread** defines several constructors.
- After the new thread is created, it will not start running until you call its **start( )** method

start ( ) executes a call to **run( )**

Creating a Thread: Implementing Runnable interface

- A class need only implement a single method called **run()**

  public void run()

- Inside **run( )**, you will define the code that constitutes the new thread

- **run( )** establishes the entry point for another thread of execution within your program

- **run( )** can call other methods, use other classes, and declare variables, just like the main thread can

In short…

- The Runnable interface should be implemented by any class whose instances are intended to be executed by a thread. Runnable interface have only one method named run().
- **public void run():** is used to perform action for a thread.
- **start() method** of Thread class is used to start a newly created thread.
- The following tasks are performed:

- A new thread starts (with new callstack).
- The thread moves from New state to the Runnable state.
- When the thread gets a chance to execute, its target run() method will run.

Thread: Constructor Overloading

- Thread():
- Thread(Runnable target):
- Thread(Runnable target, String name):
- Thread(String name):
- Thread(ThreadGroup group, Runnable target):
- Thread(ThreadGroup group, Runnable target, String name):
- Thread(ThreadGroup group, Runnable target, String name, long stackSize):
- Thread(ThreadGroup group, String name):