Multithreading

Differences between multi threading and multitasking

Multi-Tasking

- Two kinds of multi-tasking:
  1) process-based multi-tasking
  2) thread-based multi-tasking
- Process-based multi-tasking is about allowing several programs to execute concurrently, e.g. Java compiler and a text editor.
- Processes are heavyweight tasks:
  1) that require their own address space
  2) inter-process communication is expensive and limited
  3) context-switching from one process to another is expensive and limited

Thread-Based Multi-Tasking

- Thread-based multi-tasking is about a single program executing concurrently
- several tasks e.g. a text editor printing and spell-checking text.
- Threads are lightweight tasks:
  1) they share the same address space
  2) they cooperatively share the same process
  3) inter-thread communication is inexpensive
  4) context-switching from one thread to another is low-cost
- Java multi-tasking is thread-based.

Reasons for Multi-Threading

- Multi-threading enables to write efficient programs that make the maximum use of the CPU, keeping the idle time to a minimum.
- There is plenty of idle time for interactive, networked applications:

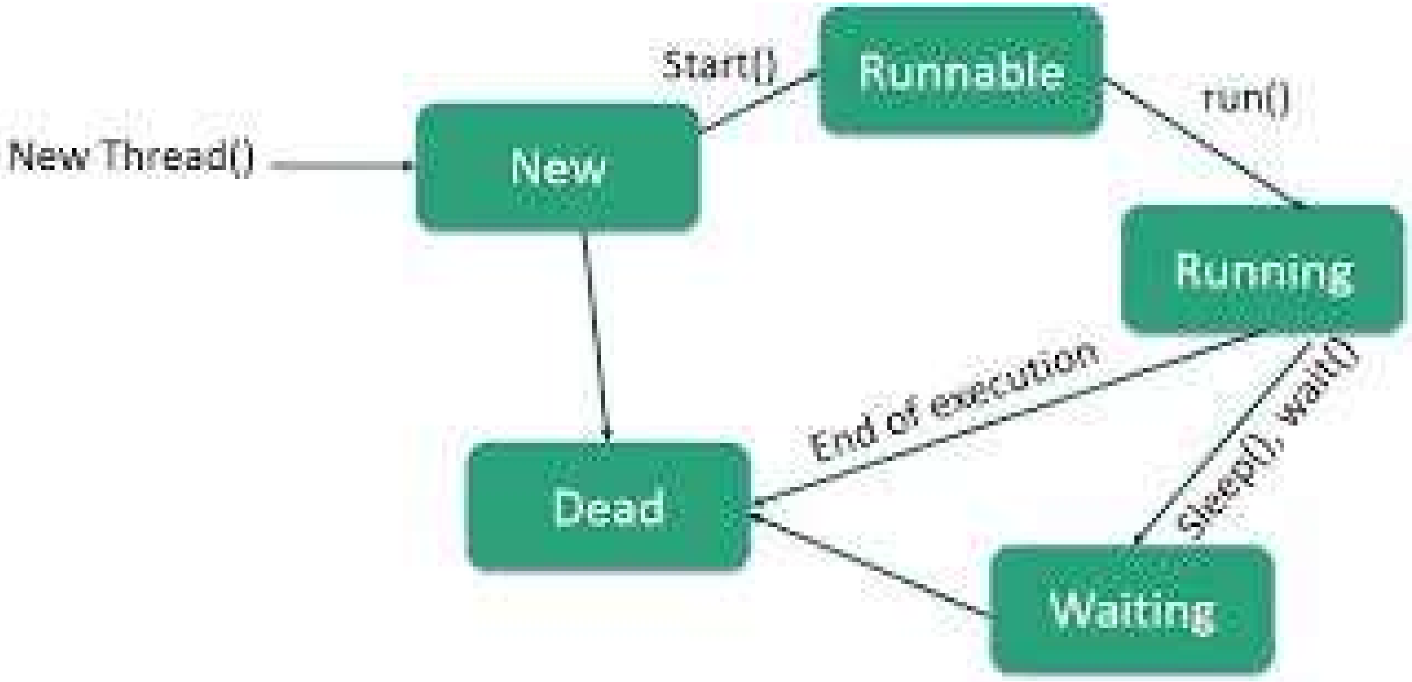
  1) the transmission rate of data over a network is much slower than the rate at which the computer can process it

  2) local file system resources can be read and written at a much slower rate than can be processed by the CPU

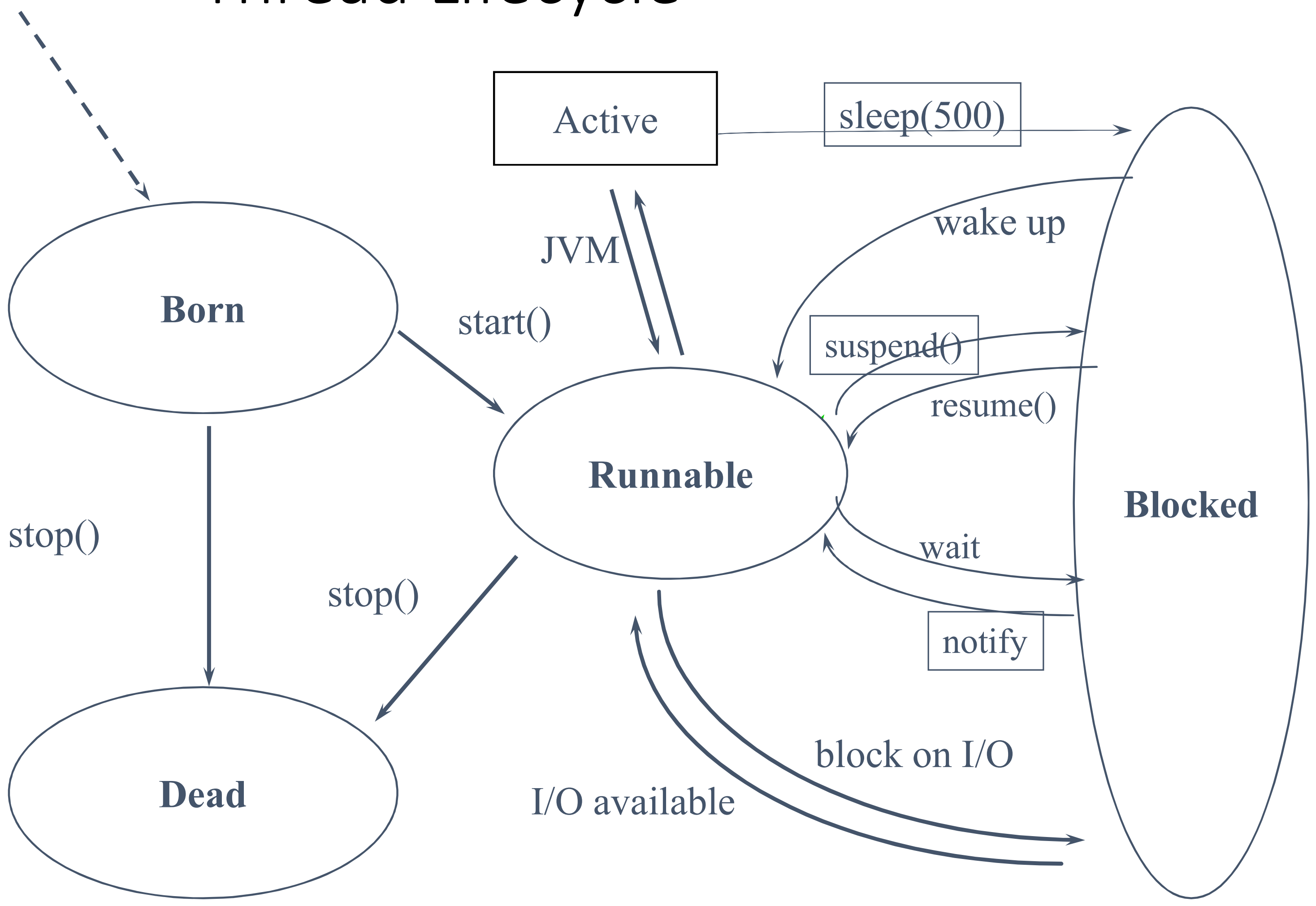
  3) of course, user input is much slower than the computer

Thread Lifecycle

- Thread exist in several states:
  1) ready to run
  2) running
  3) a running thread can be suspended
  4) a suspended thread can be resumed
  5) a thread can be blocked when waiting for a resource
  6) a thread can be terminated
- Once terminated, a thread cannot be resumed.


New Thread()
New
Start()
Runnable
run()
Running
End of execution
Sleep(), wait()
Dead
Waiting

Thread Lifecycle



- **New state –** After the creations of Thread instance the thread is in this state but before the start() method invocation. At this point, the thread is considered not alive.
- **Runnable (Ready-to-run) state –** A thread start its life from Runnable state. A thread first enters runnable state after the invoking of start() method but a thread can return to this state after either running, waiting, sleeping or coming back from blocked state also. On this state a thread is waiting for a turn on the processor.
- **Running state –** A thread is in running state that means the thread is currently executing. There are several ways to enter in Runnable state but there is only one way to enter in Running state: the scheduler select a thread from runnable pool.
- **Dead state –** A thread can be considered dead when its run() method completes. If any thread comes on this state that means it cannot ever run again.
- **Blocked -** A thread can enter in this state because of waiting the resources that are hold by another thread.

Creating Threads

- To create a new thread a program will:

  1) extend the Thread class, or

  2) implement the Runnable interface
- Thread class encapsulates a thread of execution.
- The whole Java multithreading environment is based on the Thread class.

Thread Methods

- Start: a thread by calling start its run method
- Sleep: suspend a thread for a period of time
- Run: entry-point for a thread
- Join: wait for a thread to terminate
- isAlive: determine if a thread is still running
- getPriority: obtain a thread’s priority
- getName: obtain a thread’s name

New Thread: Runnable

- To create a new thread by implementing the Runnable interface:

  1) create a class that implements the run method (inside this method, we define the code that constitutes the new thread):

  ```
  public void run()
  ```

  2) instantiate a Thread object within that class, a possible constructor is:

  ```
  Thread(Runnable threadOb, String threadName)
  ```

  3) call the start method on this object (start calls run):

  ```
  void start()
  ```

Example

- A class NewThread that implements Runnable:

```
class NewThread implements Runnable {
Thread t;
//Creating and starting a new thread. Passing this to the Thread
  //constructor – the new thread will call ‘this’ object’s run method:
NewThread() {
t = new Thread(this, "Demo Thread");
System.out.println("Child thread: " + t);
t.start();
}
//This is the entry point for the newly created thread – a five-iterations loop
//with a half-second pause between the iterations all within try/catch:
public void run() {
try {
for (int i = 5; i > 0; i--) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
} catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}
```

```
class ThreadDemo {
public static void main(String args[]) {
//A new thread is created as an object of
// NewThread:
new NewThread();
//After calling the NewThread start
 method,
// control returns here.
//Both threads (new and main) continue concurrently.
//Here is the loop for the main thread:
try {
for (int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}
```

New Thread: Extend ‘Thread’

- The second way to create a new thread:
  1) create a new class that extends Thread
  2) create an instance of that class
- Thread provides both run and start methods:
  1) the extending class must override run
  2) it must also call the start method

Example

- The new thread class extends Thread:

```
class NewThread extends Thread {
//Create a new thread by calling the Thread’s
// constructor and start method:
NewThread() {
super("Demo Thread");
System.out.println("Child thread: " + this);
this.start();
}
```

NewThread overrides the Thread’s run method:

```
public void run() {
try {
for (int i = 5; i > 0; i--) {
System.out.println("Child Thread: " + i);
Thread.sleep(500);
}
} catch (InterruptedException e) {
System.out.println("Child interrupted.");
}
System.out.println("Exiting child thread.");
}
}
```

```
class ExtendThread {
public static void main(String args[]) {
//After a new thread is created:
new NewThread();
//the new and main threads continue
//concurrently…
//This is the loop of the main thread:
try {
for (int i = 5; i > 0; i--) {
System.out.println("Main Thread: " + i);
Thread.sleep(1000);
}
} catch (InterruptedException e) {
System.out.println("Main thread interrupted.");
}
System.out.println("Main thread exiting.");
}
}
```

Threads: Synchronization

- Multi-threading introduces asynchronous behavior to a program.
- How to ensure synchronous behavior when we need it?
- For instance, how to prevent two threads from simultaneously writing and reading the same object?
- Java implementation of monitors:

  1) classes can define so-called synchronized methods

  2) each object has its own implicit monitor that is automatically entered when one of the object's synchronized methods is called

  3) once a thread is inside a synchronized method, no other thread can call any other synchronized method on the same object

Thread Synchronization

- Language keyword: `synchronized`
- Takes out a monitor lock on an object
    - Exclusive lock for that thread
- If lock is currently unavailable, thread will block

Thread Synchronization

- Protects access to code, not to data
  - Make data members private
  - Synchronize accessor methods
- Puts a “force field” around the locked object so no other threads can enter
    - Actually, it only blocks access to other synchronizing threads

Daemon Threads

- Any Java thread can be a *daemon* thread.
- Daemon threads are service providers for other threads running in the same process as the daemon thread.
- The run() method for a daemon thread is typically an infinite loop that waits for a service request. When the only remaining threads in a process are daemon threads, the interpreter exits. This makes sense because when only daemon threads remain, there is no other thread for which a daemon thread can provide a service.
- To specify that a thread is a daemon thread, call the setDaemon method with the argument true. To determine if a thread is a daemon thread, use the accessor method isDaemon.

Thread Groups

- Every Java thread is a member of a *thread group*.
- Thread groups provide a mechanism for collecting multiple threads into a single object and manipulating those threads all at once, rather than individually.
- For example, you can start or suspend all the threads within a group with a single method call.
- Java thread groups are implemented by the “ThreadGroup” class in the java.lang package.
- The runtime system puts a thread into a thread group during thread construction.
- When you create a thread, you can either allow the runtime system to put the new thread in some reasonable default group or you can explicitly set the new thread's group.
- The thread is a permanent member of whatever thread group it joins upon its creation--you cannot move a thread to a new group after the thread has been created

The ThreadGroup Class

- The “ThreadGroup” class manages groups of threads for Java applications.
- A ThreadGroup can contain any number of threads.
- The threads in a group are generally related in some way, such as who created them, what function they perform, or when they should be started and stopped.
- ThreadGroups can contain not only threads but also other ThreadGroups.
- The top-most thread group in a Java application is the thread group named main.
- You can create threads and thread groups in the main group.
- You can also create threads and thread groups in subgroups of main.

Creating a Thread Explicitly in a Group

- A thread is a permanent member of whatever thread group it joins when its created--you cannot move a thread to a new group after the thread has been created. Thus, if you wish to put your new thread in a thread group other than the default, you must specify the thread group explicitly when you create the thread.
- The Thread class has three constructors that let you set a new thread's group:

  public Thread(ThreadGroup *group*, Runnable *target*) public Thread(ThreadGroup *group*, String *name*) public Thread(ThreadGroup *group*, Runnable *target*, String *name*)
- Each of these constructors creates a new thread, initializes it based on the Runnable and String parameters, and makes the new thread a member of the specified group.

  For example:

  ThreadGroup myThreadGroup = new ThreadGroup("My Group of Threads");

  Thread myThread = new Thread(myThreadGroup, "a thread for my group");