6SENG006W Concurrent Programming

Week 8

Java Thread Scheduling & Thread Groups

Java Thread Scheduling & Thread Groups

The aim of this lecture is to describe two aspects of a Java thread.

Thread Scheduling:

- how the JVM *schedules* Java threads to be executed.

Thread Groups:

- a way of *grouping* collections of Java threads together.

PART I

Java Thread Scheduling

Thread Scheduling

The aim of the first part of the lecture is to explain how the JVM schedules Java threads:

- Thread *priority* & thread *priority queues*.
- `Thread` class's *priority* methods, e.g.
  - `setPriority( int new_priority )`
  - `getPriority()`
- The JVM *scheduling algorithm* & the *"scheduling rule"*.
- Relationship between the *scheduling algorithm* & *thread states*.
  - **`RUNNABLE`** & the **"non-runnable"** states.
- Comparing *time-slicing* vs. *non-time-slicing* systems.
- Dealing with *CPU intensive* threads via the `yield()` method.
- Introduce *"daemon"* threads.

Thread Scheduling

Threads should be written as if they are to be *run concurrently*, but in practice this is not usually the case.

Since in many computers there is either only one CPU or the JVM is running on just one CPU.

Therefore, threads must share the CPU & actually run one at a time in such a way as to *simulate concurrency*.

The execution of multiple threads on a single CPU, in some order, is called *scheduling*.

The Java run time supports a very simple, *deterministic* scheduling algorithm known as *"fixed priority scheduling"*.

This algorithm *schedules threads*:

- based on their *priority* relative to other *runnable* threads.

The scheduling algorithm is **ONLY CONCERNED** with threads in the **`RUNNABLE`** state.

It is **NOT CONCERNED** with threads in any of the *other thread states*.

Thread Priority

Each Java thread is given a *numeric priority* when it is created.

Thread *priority* ranges between the value of two constants defined in the `Thread` class (see “Field” summary):

- `MIN_PRIORITY` = 1,
- `MAX_PRIORITY` = 10.

By *default* a thread’s priority is set to `NORM_PRIORITY` (= 5).

A thread *inherits* its priority from the *thread that created it*.

A thread’s priority can be *modified* at any time after its creation using the `Thread` class method:

```
public final void setPriority( int new_priority )
```

To *get* a thread’s current priority use the `Thread` class method:

```
public final int getPriority()
```

Java Scheduling Algorithm

- When there are threads in the **`RUNNABLE`** state ready to be executed, the run time system will have to pick one of these threads to run next.
- The system will choose a thread that has the **highest priority** of those threads currently in the **`RUNNABLE`** state.
- If the current highest priority is *hp*, then the chosen thread is either:
  - the only thread with priority *hp*, or
  - is one of a group of threads that all have priority *hp*.
- If there is a group of threads with the same priority waiting to execute, the scheduler chooses them in a *"round-robin"* fashion.
- Only when that thread:
  - *terminates*,
  - *yields*,
  - or becomes *not runnable* for some reason,

  will a *lower priority thread* start executing.
- The Java run time system's thread scheduling algorithm is also *"preemptive"*.

“Preemptive” Scheduling

“Preemptive” scheduling means that:

> *If at any time a thread with a higher priority than the currently executing thread enters or re-enters the* **`RUNNABLE`** *state & thus becomes runnable, then the JVM scheduler:*
>
> 1. ***Stops*** *the currently executing lower priority thread.*
> 2. *Moves it into the pool of “ready to run” threads.*
>   *(Note it is still in the* **`RUNNABLE`** *state);*
> 3. *Starts executing the (newly runnable) higher priority thread in its place.*

The *new higher priority thread* is said to **“preempt”** the *lower priority thread* that was executing.

Thread Priority Queues

The Java scheduler maintains a *collection of priority queues* of the **RUNNABLE** threads, that are *"ready to run"*, as follows:

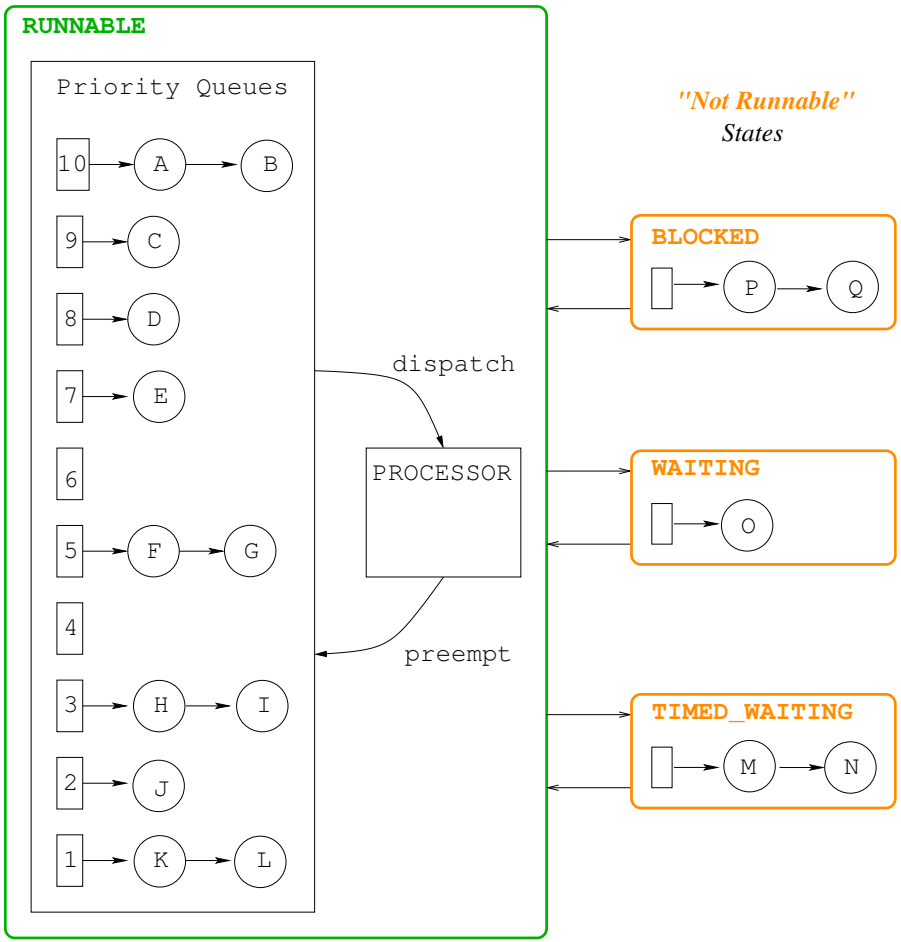

Figure : 8.1 JVM's Thread Priority Queues

Java Scheduling Rule

Java run time system thread scheduling scheme is given by the rule:

> **Definition:** *Scheduling Rule*
>
> At any given time, the *highest priority runnable* thread is running.

That is, a thread in the `RUNNABLE` state with the *highest priority*.

Warning: Unfortunately, this rule can **not** be guaranteed.

The scheduler may choose to run a *lower priority thread* to avoid *starvation*.

Starvation could happen because either:

- *low priority threads* never get a chance to execute because there are always higher priority threads that are runnable; or
- when the low priority threads do get a chance to execute they are always *preempted* before they are able to make any progress.

Java Scheduling **Health Warning**

Because of the above issue of the Java run-time system **not** strictly applying its general scheduling rule, it is necessary to issue the following **"health warning"**:

- Only use **priority** to *affect the scheduling policy for efficiency*.
- **DO NOT** use it to try to achieve *program correctness*.

YOU HAVE BEEN WARNED!

Example of Thread Priorities

`RaceApplet` is an Applet that implements an *animated race* between two `Runner` threads.

`Runner` is:

- a *thread*,
- its `run()` method simply increments `tick` variable from 1 to 400,000.
- `tick` is `public`.

  So the `RaceApplet` can determine how far a `Runner` has progressed.

  Used to draw a line proportional to value of `tick`.
- `RaceApplet` uses the **`terminate`** variable to *terminate* `Runner` when the web page is left.

Example of Thread Priorities: `RaceApplet`

`RaceApplet` applet:

- Implements `Runnable`.
- Creates & runs 2 `Runner` threads, either:
  - **"fair"** race – equal priorities, or
  - **"unfair"** race – different priorities.
- Uses a *utility thread* `updateThread` to redraw screen every 10 ms.

  It has a *higher priority* than the 2 `Runner` threads.
- Draws a line proportional to progress of 2 `Runner` threads, i.e. value of `tick`.
- *terminates* `updateThread` & 2 `Runner` threads when left web page.

Diagrammatic View of `RaceApplet`: "Unfair" race

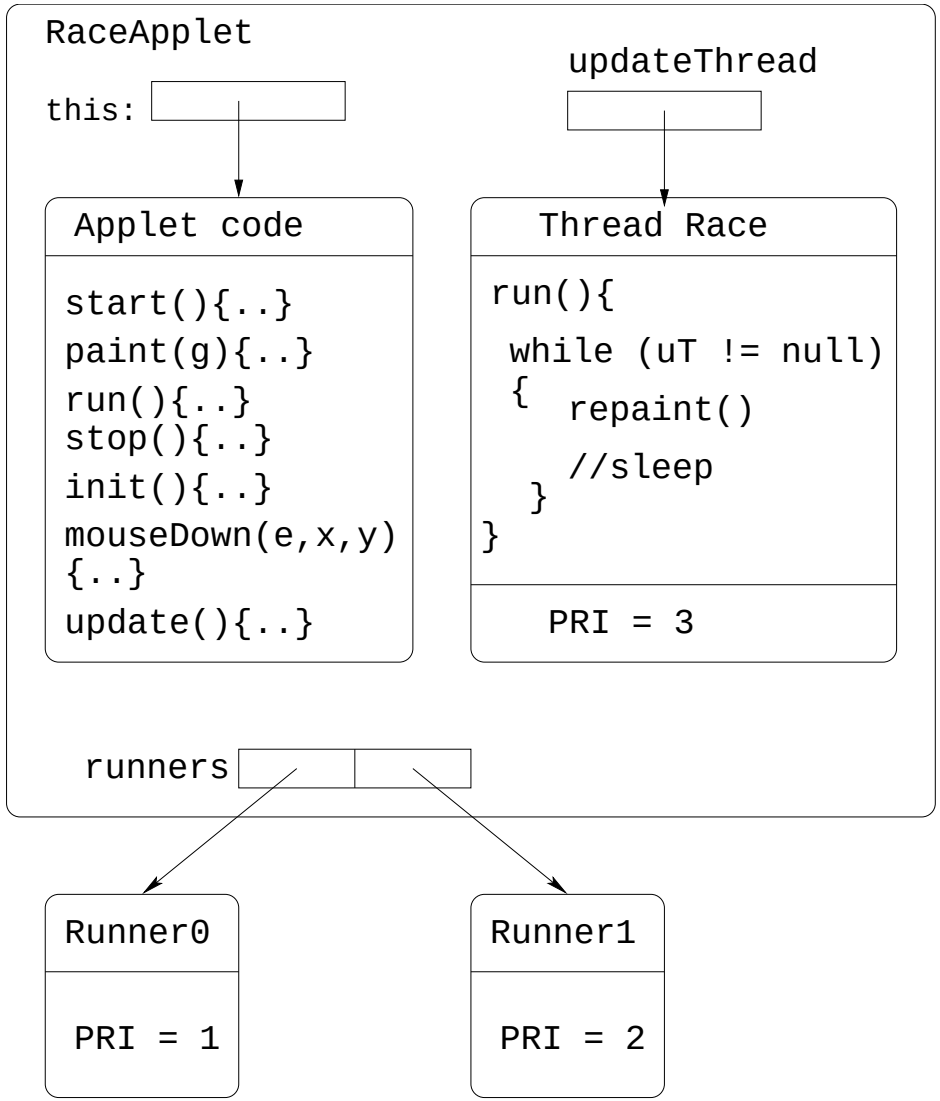

Figure : 8.2 `RaceApplet`, with "Unfair" Race.

Runner Thread

```
class Runner extends Thread
{
  // Used to TERMINATE the thread
  public volatile boolean terminate = false ;

  public int tick = 1 ;

  public Runner( int id )
  {
     super( "Runner" + id ) ;
  }


  public void run()
  {
     while ( tick < 400000 && !terminate )
     {
         tick++ ;
     }
  }
}
```

(1) RaceApplet: Declarations & run()

```
import java.awt.* ;
import java.applet.Applet ;

public class RaceApplet extends Applet implements Runnable
{
  final static int NUMRUNNERS = 2  ;
  final static int SPACING    = 20 ;

  Thread updateThread == null ;

  // An "ARRAY of THREADS"
  Runner runners[] = new Runner[NUMRUNNERS] ;


  // Provides body for "updateThread"
  public void run()
  {
      while ( updateThread != null )
      {
          repaint() ;
          try {
                updateThread.sleep(10) ;
          } catch (InterruptedException e) { }
      }
  }
```

(2) RaceApplet: init()

```
public void init()
{
   // Get argument value of HTML tag "type"
   String raceType = getParameter("type") ;

   for ( int i = 0 ; i < NUMRUNNERS ; i++ )
   {
      runners[i] = new Runner(i) ;

      if ( raceType.compareTo("unfair") == 0 )
            runners[i].setPriority( i + 1 ) ; // unfair
      else
            runners[i].setPriority( 2 ) ;     // fair
   }

   if ( updateThread == null )
   {
      // "this" is the runnable target
      updateThread = new Thread( this, "Thread Race" ) ;

      updateThread.setPriority( NUMRUNNERS + 1 ) ;
   }
}
```

(3) RaceApplet: mouseDown()

Start the Runner threads racing by a mouse click.

```
public boolean mouseDown( java.awt.Event evt, int x, int y )
{
    if ( !updateThread.isAlive() )
    {
        updateThread.start() ;
    }

    for ( int i = 0 ; i < NUMRUNNERS ; i++ )
    {
        if ( !runners[i].isAlive() )  // in NEW state
        {
           runners[i].start() ;       // in RUNNABLE state
        }
    }
    return true ;
}
```

(4) RaceApplet: paint() & update()

```
public void paint( Graphics g )
{
  g.setColor(Color.lightGray) ;
  g.fillRect(0, 0, size().width, size().height) ;
  g.setColor(Color.black) ;

  for ( int i = 0 ; i < NUMRUNNERS ; i++ )
  {
    int pri = runners[i].getPriority() ;

    g.drawString( new Integer(pri).toString(), 0, (i+1)*SPACING ) ;
  }
  update(g) ;
}

public void update(Graphics g)
{
  for (int i = 0 ; i < NUMRUNNERS ; i++)
  {
    g.drawLine( SPACING, (i+1)*SPACING,
                SPACING + ( runners[i].tick ) / 1000,
                (i+1) * SPACING ) ;
  }
}
```

(5) RaceApplet: run() & stop()

```
    // Applet's stop method
    public void stop()
    {
        for (int i = 0 ; i < NUMRUNNERS ; i++)
        {
            // Terminate Runner thread
            if ( runners[i].isAlive() )
            {
                runners[i].terminate = true ;
                runners[i] = null ;
            }
        }

        if ( updateThread.isAlive() )
        {
            updateThread = null ;     // Terminate updateThread
        }
    }
}
```

"Unfair" Race – How it is Executed

1. Clicking the mouse over the applet starts the two runners & the line drawing thread.
2. `Runner0` given priority 1 (the lowest possible priority);
   `Runner1` given priority 2;
   `updateThread` given priority 3.
3. `updateThread`'s `run()` method is an infinite loop, .
4. During each iteration it draws a line for each runner thread (based on its `tick` variable) & then sleeps for 10 milliseconds.
5. Whenever the drawing thread wakes up after 10 milliseconds, it becomes the *highest priority thread*, **preempting** whichever `Runner` thread is currently executing & draws the lines.
6. This is **not a fair race** because the `Runner`s have *different priorities*.
7. Each time the drawing thread relinquishes the CPU by going to sleep, the scheduler chooses the *highest priority runnable thread* to run.

   This is *always* `Runner1`.

(1) Scheduling the `RaceApplet`:"Unfair" Race

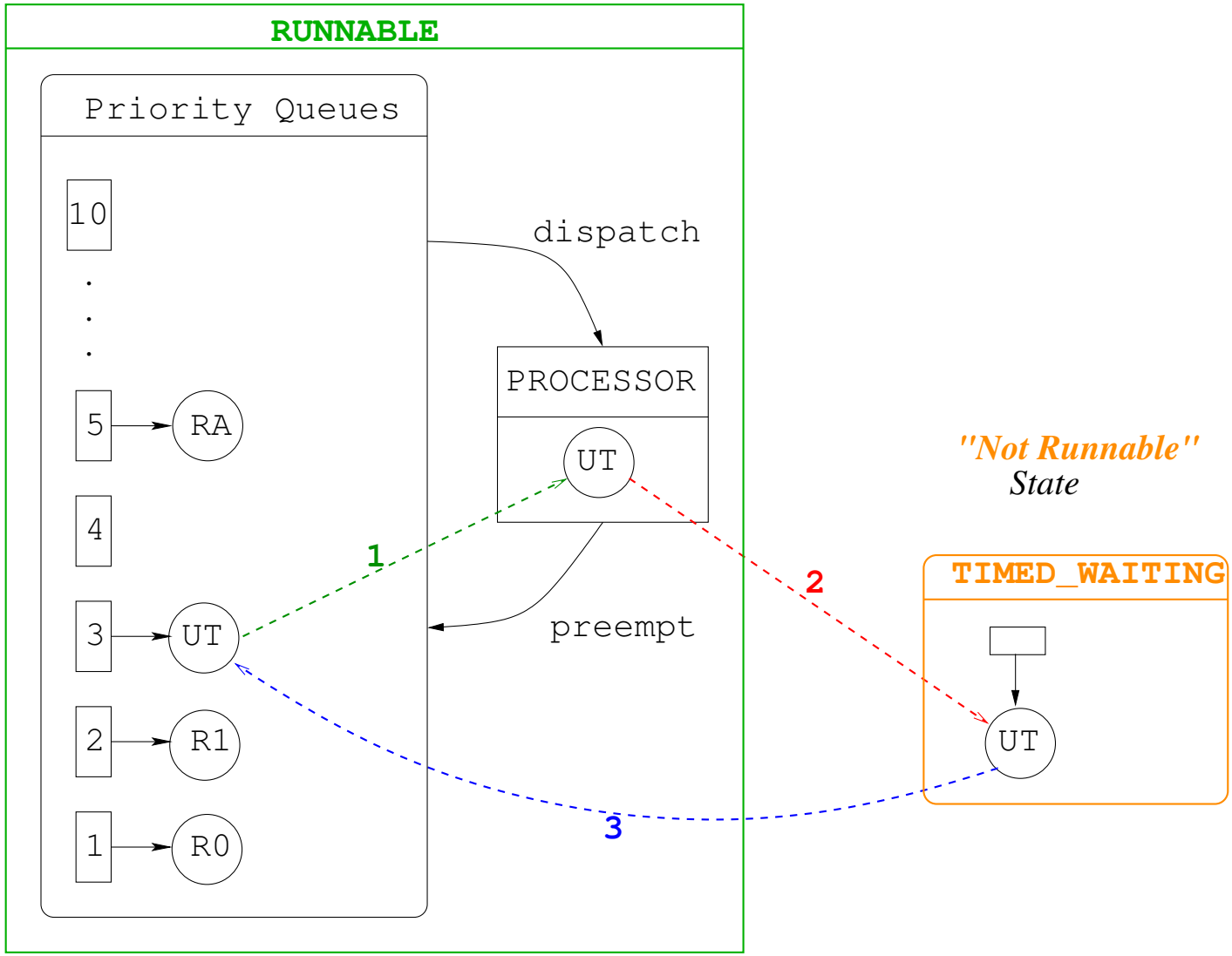

Figure : 8.3 (a) Scheduling "Unfair" `RaceApplet`

(1) `updateThread` Actions

Here `updateThread` (`UT`) has priority 3, higher than both `Runner` threads.

It goes through the following *scheduling cycle*:

1. `updateThread` is in the **`RUNNABLE`** state, & runs on the processor, since it has the highest priority.

2. It executes `sleep` & is moved to the **`TIMED_WAITING`** state.

3. The time expires & is moved back to the **`RUNNABLE`** state.

4. The JVM scheduler then *preempts*, which ever `Runner` thread is running with the `updateThread`, i.e. goes to 1.

(2) Scheduling the `RaceApplet`

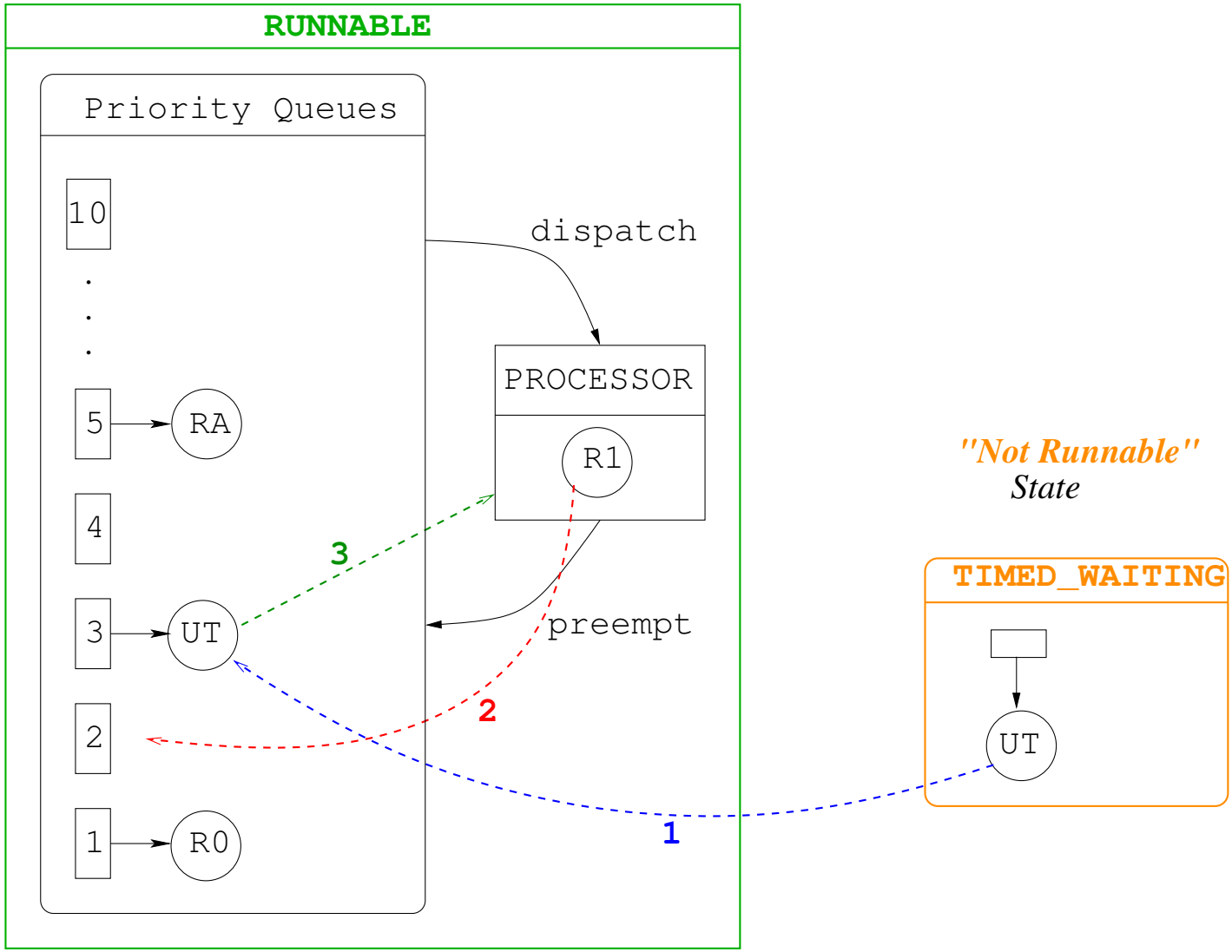

Figure : 8.4 (b) Scheduling **"Unfair"** `RaceApplet`

(2) `Runner1` Actions

Here `Runner1` (`R1`) has priority 2 & is running on the processor.

`updateThread` (`UT`) has priority 3 & is in the **`TIMED_WAITING`** state.

Then:

1. The `sleep` time for `updateThread` expires & it is moved back to the **`RUNNABLE`** state.

2. `Runner1` has a lower priority than `updateThread` , so it is *preempted*.

   `Runner1` is added to the end of the priority 2 queue.

3. `updateThread` is dispatched to the processor, since it has the highest priority.

“Fair” Runners Race

The applet implements a **“fair race”** if the parameter `type` is not equal to “`unfair`”, e.g. “`fair`”.

Then both `Runner`s are assigned the *same priority* (= 2) in the applet’s `init()` method.

Consequently, they have an *equal chance* of being chosen by the scheduler to run.

In this race, each time the drawing thread relinquishes the CPU by going to sleep, there are *two runnable threads of equal priority* – the `Runner`s – waiting for the CPU.

The scheduler must choose one of the threads to run.

In this situation, the scheduler chooses the next thread to run in a *round-robin* fashion.

(3) Scheduling the `RaceApplet` "Fair" Race

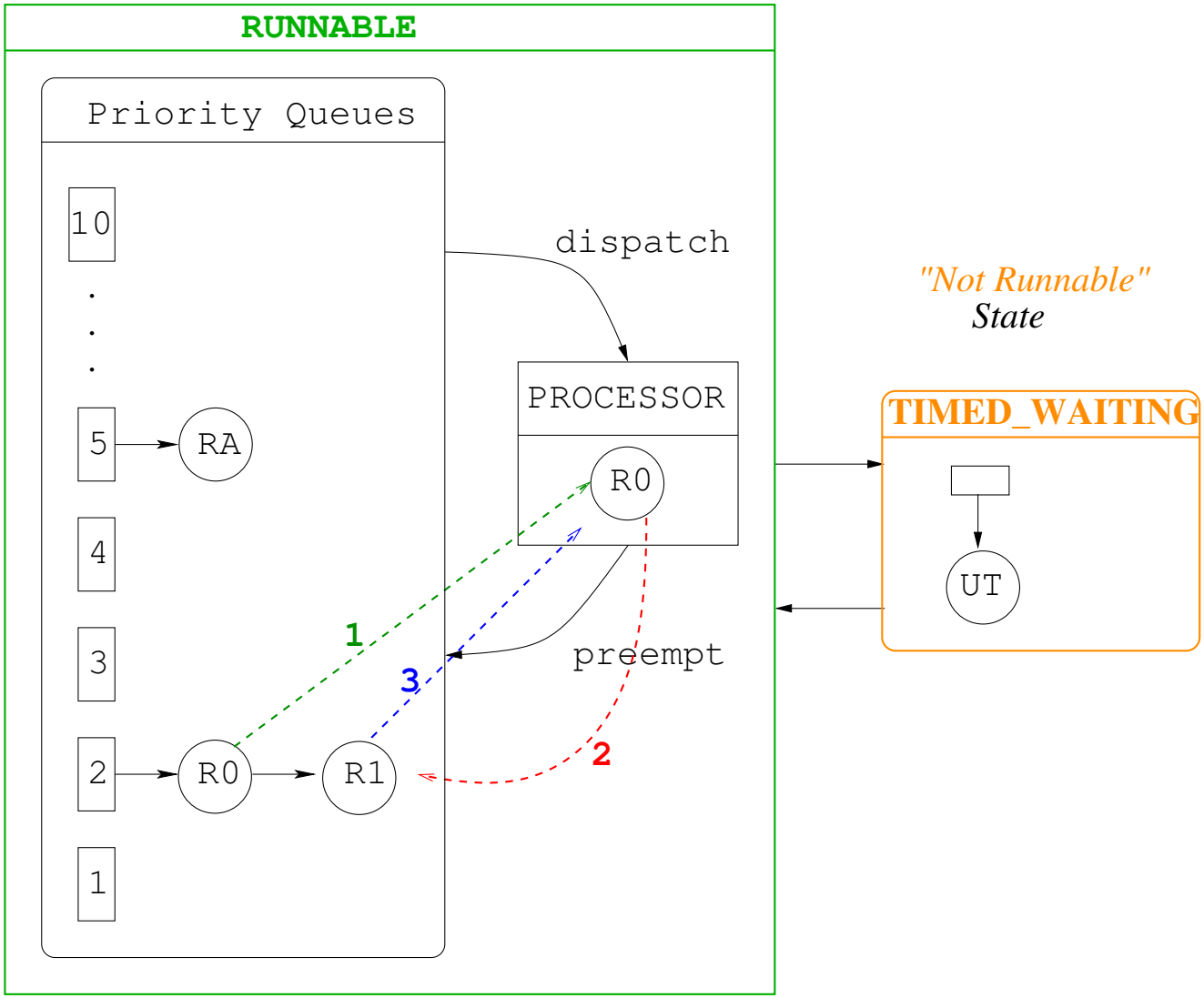

Figure : 8.5 Scheduling "Fair" `RaceApplet`

(3) Scheduling the `RaceApplet`

`updateThread` (`UT`) is in the **`TIMED_WAITING`** state via `sleep`, then:

1. `Runner0` (`R0`) is dispatched to the processor, since it is at the front of the highest priority queue with runnable threads.

2. When `updateThread`'s `sleep` expires `Runner0` (`R0`) is preempted & placed at the end of the priority 2 queue.

3. The next time `updateThread` sleeps `Runner1` (`R1`) is dispatched to the processor, since it will be at the *front* of the queue.

Selfish Threads

The `Runner` class used in the races above actually implements **"socially-impaired"** thread behaviour.

Recall the `run()` method from the `Runner` class used in the races above.

The `while` loop in the `run()` method is in a *"tight"* loop.

This means that once the scheduler chooses a thread with this thread body for execution, the thread **never voluntarily relinquishes** control of the CPU.

The thread continues to run until the `while` loop terminates naturally or until the thread is preempted by a higher priority thread.

In some situations, having *"selfish"* threads does not cause any problems because a higher priority thread preempts the selfish one.

But threads with **"CPU greedy"** `run()` methods, can take over the CPU & cause other threads to have to wait for a long time before getting a chance to run.

Clearly, this can cause problems in some systems.

Time-Slicing

Some systems fight selfish thread behaviour with a strategy known as *time-slicing*.

Time-slicing comes into play when there are:

- multiple *runnable* threads of equal priority;
- and these threads are the highest priority threads competing for the CPU.

Time-Slicing Example: `SelfishRunner`

Consider the following **selfish** thread class `SelfishRunner`.

```
class SelfishRunner extends Thread
{
  public int tick = 1 ;

  SelfishRunner( int id ){  super( "SelfishRunner-" + id ) ;  }

  public void run()
  {
    while ( tick < 400000 )
    {
      tick++ ;

      if ( ( tick % 50000 ) == 0 )
      {
        System.out.println( getName() + ": tick = " + tick ) ;
      }
    }
  }
}
```

`run()` contains a *tight loop* that increments `tick` & every 50,000 ticks prints out the thread's name & its `tick` count.

The main class `RaceTest`

`RaceTest` *creates two equal priority* **selfish** threads `SelfishRunner`.

```
class RaceTest
{
  final static int NumRunners = 2 ;

  public static void main( String args[] )
  {
    SelfishRunner runners[] = new SelfishRunner[ NumRunners ] ;

    for ( int i = 0 ; i < NumRunners ; i++ )
    {
        runners[i] = new SelfishRunner( i ) ;

        runners[i].setPriority( 2 ) ;
    }

    for ( int i = 0 ; i < NumRunners ; i++ )
    {
        runners[i].start() ;
    }
  }
}
```

Execution on a Time-Sliced System

The messages from both threads would be intermingled with one another when run on a *time-sliced* system.

```
SelfishRunner-1: tick = 50000
SelfishRunner-0: tick = 50000
SelfishRunner-0: tick = 100000
SelfishRunner-1: tick = 100000
          .
          .
SelfishRunner-0: tick = 400000
SelfishRunner-1: tick = 400000
```

Because a time-sliced system *divides the CPU into time slots* & *iteratively* gives each of the *equal-&-highest priority* threads a time slot in which to run.

The time-sliced system will continue to iterate through the equal-and-highest priority threads allowing each one a time slot to run until one or more of them finishes or *until a higher priority thread preempts them*.

Note: time-slicing makes **no guarantees** as to *how often* or *in what order* threads are scheduled to run.

Execution on a Non-Time-Sliced System

On a *non-time-sliced* system, the messages from one thread finish printing before the other thread starts.

```
SelfishRunner-0: tick = 50000
SelfishRunner-0: tick = 100000
SelfishRunner-0: tick = 150000
SelfishRunner-0: tick = 200000
SelfishRunner-0: tick = 250000
SelfishRunner-0: tick = 300000
SelfishRunner-0: tick = 350000
SelfishRunner-0: tick = 400000
SelfishRunner-1: tick = 50000
          .
          .
SelfishRunner-1: tick = 400000
```

This is because a non-time-sliced system *chooses one of the equal-and-highest priority threads* to run.

And allows that thread to *run until it relinquishes the CPU* (by sleeping, yielding, finishing) or *until a higher priority preempts it*.

CPU Intensive Threads

CPU intensive code can have **negative repercussions** on other threads running in the same process.

In general, you should try to write *"well-behaved"* threads that *voluntarily relinquish* the CPU periodically & give other threads an opportunity to run.

This can be done by using the `yield()` method.

In particular, you should **never** write Java code that *relies on time-sharing*.

This will almost certainly guarantee that your program will give different results on a different computer system.

Social Responsibility: the `yield()` Method

A thread can *voluntarily* give up its right to execute by *"yielding"* the CPU by calling the `Thread` class method `yield()`:

```
public static void yield()
```

`yield()` gives other threads **of the same priority** a chance to run.

Note: a thread can only yield the CPU to another thread with the *same priority*; an attempt to yield to a *lower* priority thread is **ignored**.

So, if there are no equal priority threads in the **`RUNNABLE`** state, then the `yield()` is ignored.

If the currently running thread yields the CPU, using `yield()`, then the scheduler implements a simple:

– *non-preemptive round-robin scheduling order*.

Daemon Threads

Any Java thread can be a *daemon* thread.

Daemon threads are *service providers* for other threads or objects running in the same process as the daemon thread.

For example, a web browser could create a daemon thread that reads images from the file system or the network.

Daemon threads are typically *independent threads* within an application that *provide services for other objects* within that same application.

Daemon thread `run()` method is usually an infinite loop that waits for requests.

When the only remaining threads in a process are daemon threads, the *interpreter exits*, i.e. the program terminates.

Since there are no threads left to request a service.

To specify that a thread is a daemon thread use: `setDaemon(true)`.

To determine if a thread is a daemon thread use: `isDaemon()`.

PART II

Thread Groups

Thread Groups

The aim of this section of the lecture is to describe Java *thread groups*.

- Concept of thread *groups*
  - *default* & *current* thread groups.
- Creating a *thread group*, using the `ThreadGroup` class constructor.
- Creating a *thread group hierarchy*, using the `Thread` & `ThreadGroup` class constructors.
- Describing thread group *attributes*.
- Describing the `ThreadGroup` class *methods* that deal with the following aspects of thread groups:
  - Collection Management,
  - Operations on a Group,
  - Operations on all the threads within a Group,
  - Access Restrictions.
- Asking what is missing from the `ThreadGroup` class?

Thread Groups

Thread groups provide a mechanism for *collecting multiple threads into a single object* & manipulating them as a group.

For example, list all the threads within a group with a single method call.

Thread groups are implemented by `ThreadGroup` class in `java.lang.package`, see its API at:

https://docs.oracle.com/en/java/javase/15/docs/api/java.base/java/lang/ThreadGroup.html

Every Java thread is put in a thread group when it is created.

There are *two possibilities* when you *create a new thread*, you can either:

- let the run time system put it in some *default group*,
- explicitly put it in some *specific thread group*.

A thread is a **permanent member** of whatever thread group it is placed in, i.e. it **CANNOT BE MOVED** to a different group.

The `ThreadGroup` Class

The `ThreadGroup` class manages groups of threads for Java applications.

*Thread groups can contain both **threads** & other **thread groups***.

The *top-most thread group* in a Java application is the **"`main`"** thread group.

You can create *"thread group hierarchies"* with "`main`" as the *"root"*, with threads & thread groups in subgroups of `main` & so on.

A thread group can contain any number of threads, usually related in some way, e.g. who created them or what function they perform.

There are two `ThreadGroup` constructors:

```
ThreadGroup( String groupName )
```

```
ThreadGroup( ThreadGroup groupParent, String groupName )
```

In the first, the *parent of this new group* is the thread group of the thread that called it.

In the second, the *parent* is the specified thread group – **`groupParent`**.

Threads in the “Default” Thread Group

`Thread` class constructors that do not take a `ThreadGroup` argument:

```
public Thread( )

public Thread( String name )

public Thread( Runnable target )

public Thread( Runnable target, String name )
```

Creating a thread using one of these, results in the thread being put in the *same group as the thread that created it*, i.e. the *“default”* thread group.

When a Java application first starts up, the Java run time system creates a `ThreadGroup` called **“`main`”**.

So, *unless specified otherwise*, **all new threads are put in the “`main`” thread group**.

In this case **“`main`”** is the *“default”* thread group.

This is not always the case for a thread created within an applet.

Explicitly Putting a Thread in a Thread Group

A thread **cannot be moved** to a new group after it has been created.

So, if you want to put a thread in a thread group other than the default, you must *specify the thread group explicitly when you create the thread*.

To do this, use one of the following `Thread` class constructors:

```
public Thread( ThreadGroup group, String name )

public Thread( ThreadGroup group, Runnable target )

public Thread( ThreadGroup group, Runnable target, String name )
```

Each of these constructors:

- *creates* a new thread,
- *initialises it* based on the `Runnable` & `String` parameters,
- *puts* the new thread into the *specified thread group* – **`group`**.

Creating a User Defined Thread Group

```
ThreadGroup myThreadGroup  = new ThreadGroup("ThreadGroup_A") ;
Thread myThread  = new Thread( myThreadGroup, "Thread_A_1") ;
```

Creates a `ThreadGroup` object named `ThreadGroup_A` and then creates a thread called `Thread_A_1` & places it in that group.

The `ThreadGroup` passed into a `Thread` constructor does not necessarily have to be a group that you create.

It can be a group created by the Java run time system, or by the program in which your thread is running.

To find out what group a (live) thread is in, you can call it's `getThreadGroup()` method.

```
ThreadGroup theGroup;
theGroup = myThread.getThreadGroup() ;
```

A thread's thread group can be used to query the group for information, such as what other threads are in the group.

You can also modify the threads in that group with a single method call.

A Thread Group Hierarchy

Fig. 8.6 illustrates a typical *thread group hierarchy*, it is a tree structure of *threads* & *thread groups*.

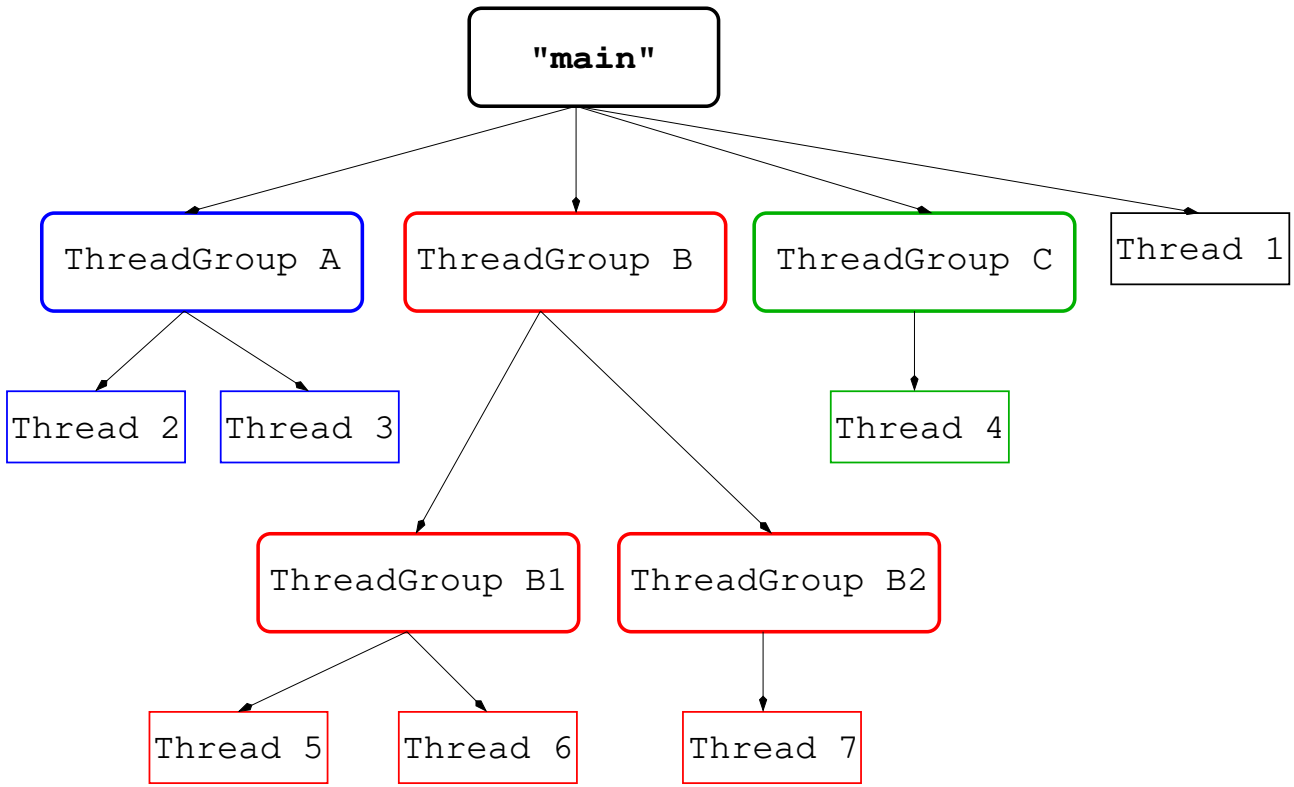

Figure : 8.6 A Thread Group Hierarchy.

We shall now look at the Java code fragments that would construct this thread group hierarchy.

Code for Thread Group Hierarchy: Thread Groups

The following Java code produces the thread group *hierarchy structure*:

```
final int NUM_GROUPS  = 6 ;
final int NUM_THREADS = 7 ;
final int main        = 0 ;

ThreadGroup groups[] ;
Thread      threads[] ;

groups  = new ThreadGroup[NUM_GROUPS] ;
threads = new Thread[NUM_THREADS] ;

// get ``main'' system group
groups[main] = Thread.currentThread().getThreadGroup() ;

// create the thread group hierarchy
groups[1] = new ThreadGroup(groups[main], "ThreadGroup A");
groups[2] = new ThreadGroup(groups[main], "ThreadGroup B");
groups[3] = new ThreadGroup(groups[main], "ThreadGroup C");

groups[4] = new ThreadGroup(groups[2],  "ThreadGroup B1");
groups[5] = new ThreadGroup(groups[2],  "ThreadGroup B2");
```

Thread Group Hierarchy: Threads

The following Java code populates the thread group hierarchy with the required *threads*:

```
// create the threads & place them in the hierarchy
threads[0] = new Thread( groups[main], "Thread 1" ) ;

threads[1] = new Thread( groups[1], "Thread 2" ) ;
threads[2] = new Thread( groups[1], "Thread 3" ) ;

threads[3] = new Thread( groups[3], "Thread 4" ) ;

threads[4] = new Thread( groups[4], "Thread 5" ) ;
threads[5] = new Thread( groups[4], "Thread 6" ) ;

threads[6] = new Thread( groups[5], "Thread 7" ) ;
```

Thread Group Hierarchy & Java Code

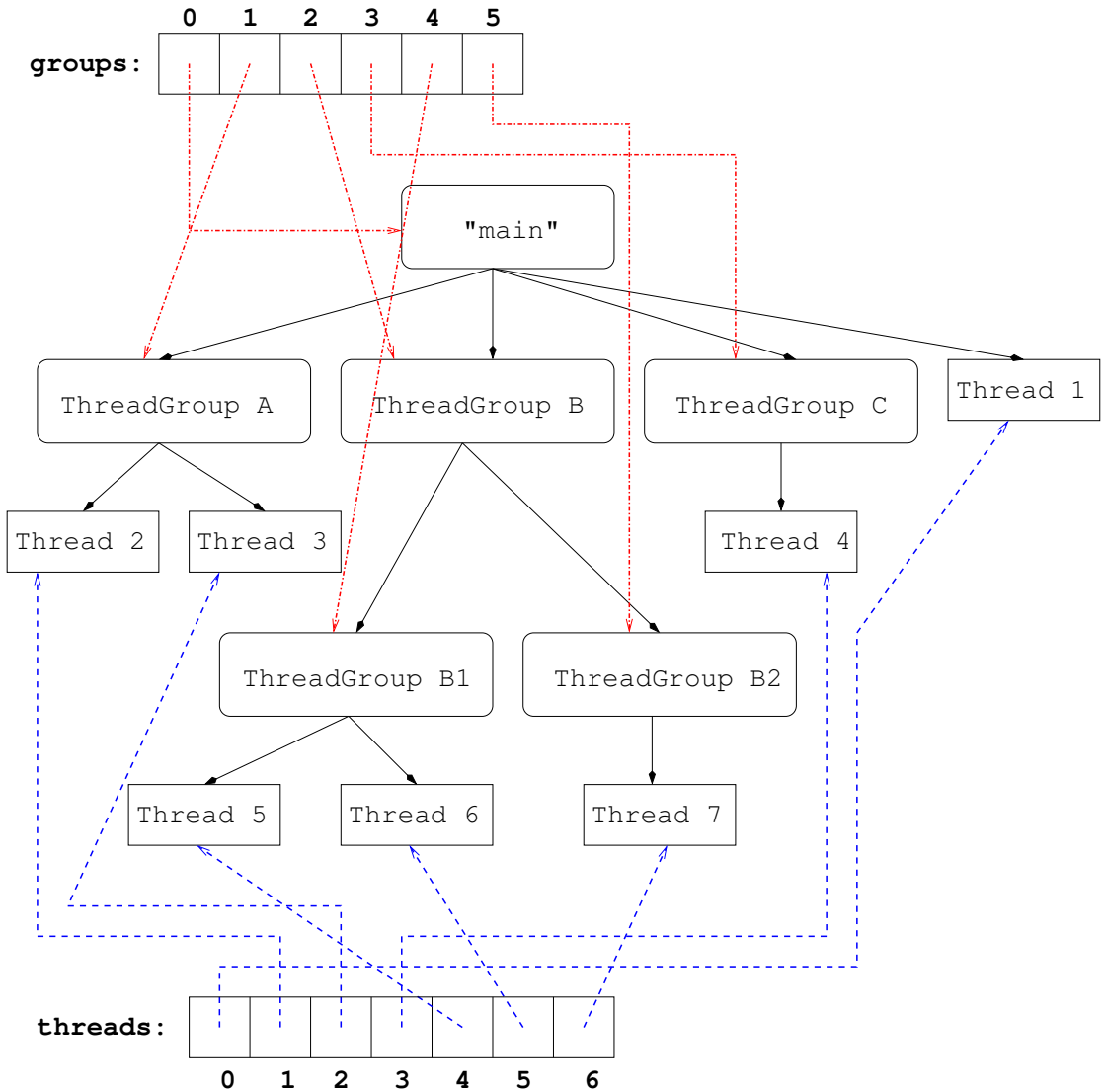

Figure : 8.7 Thread Group Hierarchy Data Structures.

`ThreadGroup` Class Methods

The `ThreadGroup` class has methods that can be categorised as follows:

Collection Management

Manage the collection of threads & subgroups contained in the thread group.

Operate on the Group

Set or get attributes of a `ThreadGroup` object.

Operate on All Threads within a Group

Perform some operation, such as interrupting, on all the threads & subgroups within a thread group.

Access Restrictions

Using the `SecurityManager` class to **restrict access to threads** based on *thread group membership*, via the `checkAccess()` method.

See the `ThreadGroup` class for full details.

Example of `ThreadGroup` Collection Management Methods

- *Manage* the threads & subgroups within the group,
- Allow *other objects to query* a thread group for *information*.

Two useful methods that that can be used together to get *an array of references* to all the *"active"* threads in a thread group & its subgroups are:

```
public int activeCount()
public int enumerate( Thread[] listOfThreads )
```

- `activeCount()` returns an **"estimate"** of the number of *"active"* threads currently in the *group & all of its subgroups*.
- `enumerate()` copies into the array **`listOfThreads`** every **"active"** thread currently in the *group & all of its subgroups*; & returns the number of threads copied into it.

If **`listOfThreads`** is **not big enough** to accommodate all of the active threads, they are **IGNORED**!

Use `activeCount()` to get an **estimate** of the size needed & **DOUBLE IT**!

Example: `listThreadsInGroup()` Method

`listThreadsInGroup()` fills the array **`listOfThreads`**, with all the *active threads* in the *thread group* passed as a parameter & *prints their names*.

```
void listThreadsInGroup( ThreadGroup threadGroup )
{
  int ac_count = threadGroup.activeCount() ;

  // allow a margin of error in the size of the array
  Thread listOfThreads[]  = new Thread[ ac_count * 2 ] ;

  int e_count = threadGroup.enumerate( listOfThreads ) ;

  System.out.println( "Threads in " + threadGroup.getName() ) ;

  for (int i = 0 ; i < e_count ; i++)
  {
     System.out.println( "Thread #" + i + " = " +
                         listOfThreads[i].getName() ) ;
  }
}
```

This can then be used as follows:

```
ThreadGroup currentGroup  = Thread.currentThread().getThreadGroup() ;
listThreadsInGroup( currentGroup ) ;
```

Methods that Operate on a Thread Group

The `ThreadGroup` class supports several *attributes* that are set & retrieved from the group as a whole.

The `ThreadGroup` *attributes* & methods that *set* & *get* them include:

- *maximum priority* that any thread within the group can have:
  `getMaxPriority(), setMaxPriority(int nmp)`
- whether the group is a *daemon group*:
  `isDaemon(), setDaemon()`
- *name* of the group:
  `getName()`
- *parent* of the group:
  `getParent(), parentOf()`
- String representation:
  `toString()`

Note that these methods *inspect* or *change* the *attribute of the* `ThreadGroup` *object*, **BUT DO NOT** affect any of the threads within the group.