CS 455: Introduction To Distributed Systems [Threads]

Shrideep Pallickara
Computer Science
Colorado State University

Frequently asked questions from the previous class survey

- How does the receiver inform the sender that its buffer is no longer full?
- Maximum segment size: Max data that can be received in a single segment
- Retransmission of a lost segment: Do you discard subsequent segments that were received? No.
  - Can the receiver buffer fill up before the "next-in-order" segment arrives?
- How do threads communicate with each other?
- Permanent generation

Topics covered in this lecture

- Creation and Management
- Thread lifecycle
  - Creating and starting threads
- Stopping and interrupting threads
- Approaches to writing threads
  - Subclassing `Threads` vs Implementing `Runnable`

Thread Abstraction

- A **thread** is a *single execution sequence* that represents a separately schedulable task
  - **Single execution sequence**
    - Each thread executes sequence of instructions – assignments, conditionals, loops, procedures, etc. – just as the sequential programming model
  - **Separately schedulable task**
    - The OS can run, suspend, or resume a thread at any time

Threads and ... Heaps And Stacks

Threads and heaps

- For performance reasons, heaps may **internally subdivide** their space into per-thread regions
  - Threads can allocate objects at the same time *without interfering* with each other
  - By allocating objects used by the same thread from the same memory region?
    - Cache hit rates may improve
- Each subdivision of the heap has thread-local variables
  - Track parts of thread-local heap in use, those that are free, etc.
- New memory allocations (`malloc()` and `new()`) can take memory from **shared heap**, only if local heap is used up

How big a stack?

- The size of the stack must be large enough to accommodate the **deepest nesting level** needed during the thread's *lifetime*
- Kernel threads
  - Kernel stacks are allocated in physical memory
  - The nesting depth for kernels tends to be small
  - E.g. 8KB default in Linux on an Intel x86
  - Buffers and data structures are allocated on the heap and never as procedure local variables

How big a stack?

- User-level stacks are allocated in virtual memory
- To catch program errors
  - Most OS will trigger **error** if the program stack grows too large too quickly
    - Indication of an unbounded recursion
  - Google's GO will automatically grow the stack as needed … this is very uncommon
  - POSIX for e.g. allows default stack size to be library dependent (e.g. larger on a desktop, smaller on a phone)
    - "Exceeding default stack limit is very easy to do, with the usual results"
      - Program termination

THREAD CREATION

Thread creation

- Using the `Thread` class
- Using the `Runnable` interface

The Thread class

```
package java.lang;

public class Thread implements Runnable {
   public Thread();
   public Thread(Runnable target);
   public Thread(ThreadGroup group, Runnable target);
   public Thread(String name);
   public Thread(ThreadGroup group, String name);
   public Thread(Runnable target, String name);
   public Thread(ThreadGroup group, Runnable target,
                 String name);
   public Thread(ThreadGroup group, Runnable target,
                 String name, long stackSize);

   public void start();
   public void run();

}
```


Threads require 4 pieces of information

- Thread **name**
  - Default is Thread-N; N is a unique number
- **Runnable target**
  - *List of instructions* that the thread executes
  - Default: `run()` method of the thread itself
- Thread **group**
  - A thread is assigned to the thread group of the thread that calls the constructor
- **Stack size**
  - Store temporary variables during method execution
  - Platform-dependent: range of legal values, optimal value, etc.

A simple thread

```
public class RandomGen extends Thread {
   private Random random;
   private int nextNumber;
   public RandomGen() {random = new Random();}

   public void run() {
     for (;;) {
       nextNumber = random.nextInt();
       try {

       } catch (InterruptedException ie) {
           ... return;
       }
     }
   }
}
```


About the code snippet

- Extends the `Thread` class
- Actual instructions we want to execute is in the `run()` method
  - Standard method of the `Thread` class
    - Place where `Thread` begins execution

Contrasting the `run()` and `main()` methods

- `main()` method
  - This is where the *first thread starts executing*
  - The **main thread**
- The `run()` method
  - *Subsequent threads* start executing with this method

Thread Lifecycle

Lifecycle of a thread

- Creation
- Starting
- Terminating
- Pausing, suspending, and resuming

Thread: Methods that impact the thread's lifecycle

```
public class Thread implements Runnable {
   public void start();
   public void run();
   public void stop();
   public void resume();
   public void suspend();
   public static void sleep(long millis);
   public boolean isAlive();
   public void interrupt();
   public boolean isInterrupted();
   public static boolean interrupted();
   public void join();
}
```

Deprecated, do not use

Thread creation

- Threads are represented by instances of the `Thread` class
- When you extend the `Thread` class?
  - Your instances are also `Threads`
- We looked at the 4 constructor arguments in the `Thread` class

Starting a thread (1/2)

- Thread exists once it's been constructed
  - But it is *not executing* … it's in a **waiting** state
- In the waiting state, other threads can *interact* with the existing **thread object**
  - Object state may be changed by other threads
    - Via method invocations

Starting a thread (2/2)

- When we're ready for a thread to begin executing code
  - Call the **`start()`** method
  - `start()` performs internal house-keeping and *then calls* the **`run()`** method
- When the `start()` method returns?
  - **Two threads** are executing in parallel
    1. The original thread which just returned from calling **`start()`**
    2. The newly started thread that is executing its **`run()`** method

After a thread's `start()` method is called

- The new thread is said to be **alive**
- The `isAlive()` method tells you about the state
  - `true`: Thread has been started and *is executing* its `run()` method
  - `false`: Thread may *not be started* yet or may be *terminated*

Terminating a thread

- Once started, a thread executes only one method: `run()`
- This `run()` may be complicated
  - May *execute forever*
  - Call *several other methods*
- Once the `run()` *finishes* executing, the thread has **completed** its execution

Like all Java methods, `run()` finishes when it …

1. Executes a `return` statement
2. Executes the last statement in its method body
3. When it *throws an exception*
   - Or fails to catch an exception thrown *to it*

The only way to terminate a thread?

- Arrange for its `run()` method to **complete**
- But the documentation for the `Thread` class lists a `stop()` method?
  - This has a *race condition* (unsafe), and has been deprecated

Some more about the `run()` method

- Cannot throw a ***checked*** exception
- But it can throw an ***unchecked*** exception
  - Exception that extends the `RuntimeException`
- A thread can be **stopped** by:
  1. *Throwing* an unchecked exception in `run()`
  2. *Failing to catch* an unchecked exception thrown by something that `run()` has called

Pausing, suspending and resuming threads

- Some thread models support the concept of **thread suspension**
  - Thread is told to *pause* execution and then told to *resume* its execution
- Thread contains `suspend()` and `resume()`
  - Suffers from vulnerability to *race conditions*: **deprecated**
- Thread can *suspend its own execution* for a specified period
  - By calling the `sleep()` method

But sleeping is not the same thing as thread suspension

- With true thread suspension
  - One thread can suspend (and later resume) *another thread*
- `sleep()` affects only the thread that executes it
  - Not possible to tell another thread to go to sleep

But you can achieve the functionality of suspension and resumption

- Use `wait` and `notify` mechanisms
- Threads **must be coded** to use this technique
  - This is not a generic suspend/resume that is imposed by another thread

Thread cleanup

- As long as some other active object holds a reference to the terminated thread object
  - Other threads can execute methods on the terminated thread … retrieve information
- If the object representing the terminated thread goes *out of scope*?
  - The thread object is **garbage collected**

Holding onto a thread reference allows us to determine if work was completed

- Done using the `join()` method
- The `join()` method
  - **Blocks** until the thread has completed
  - *Returns immediately* if
    - The thread has already completed its `run()` method
      - You can call `join()` any number of times
- Don't use `join()` to poll if the thread is still running
  - Use `isAlive()`

STOPPING A THREAD

Two approaches to stopping a thread

- Setting a flag
- Interrupting a thread

Stopping a Thread: Setting a flag

- **Set some internal flag** to signal that the thread should stop
- Thread periodically **queries the flag** to determine if it should exit

Stopping a Thread: Setting a flag

```
public class RandomGen extends Thread {
   private volatile boolean done = false;

   public void run() {
     while (!done) {
        ...
     }
   }

   public void setDone() {
      done = true;
   }
}
```

`run()` method investigates the state of the done variable on every loop. Returns when the done flag has been set.

Interrupting a thread

- In the previous slide, there may be a *delay* in the `setDone()` being invoked & thread terminating
  - Some statements are executed after `setDone()` and before the value of `done` is checked
  - In the worst case, `setDone()` is called right after the the `done` variable was checked
- **Delays** while waiting for a thread to terminate are *inevitable*
  - But it would be good if they could be minimized

Interrupting a thread

- When we arrange for thread to terminate, we:
  - Want it to *complete its blocking method* immediately
  - Don't wish to wait for the data (or …) because the thread will exit
- Use `interrupt()` method of the `Thread` class to **interrupt** any *blocking method*

Effects of the interrupt method

- Causes blocked method to **throw** an **`InterruptedException`**
  - `sleep()`, `wait()`, `join()`, and methods to read I/O
- Sets a **flag** inside the thread object to indicate that the thread has been interrupted
  - Queried using `isInterrupted()`
    - Returns `true` if it was interrupted, even though it was not blocked

Stopping a thread: Using interrupts

```
public class RandomGen extends Thread {

   public void run() {
     while (!isInterrupted()) {
        ...
     }
   }
}
```

`radomGeneratorThread.interrupt()`

The `Runnable` interface

- Allows **separation** of the *implementation* of the task *from the thread* used to run task

```
public interface Runnable {

   public void run();

}
```


Creation of a thread using the `Runnable` interface

- Construct the thread
  - Pass runnable object to the thread's constructor
- Start the thread
  - Instead of starting the runnable object

Creation of a thread using the `Runnable` interface

```
public class RandomGenerator implements Runnable {

   public void run() { ... }

}

...
     generator = new RandomGenerator();
     Thread createdThread = new Thread(generator);
     createdThread.start();
```

When to use `Runnable` and `Thread`

- If you would like your class to inherit behavior from the `Thread` class
  - **Extend** `Thread`
- If your class needs to inherit from other classes
  - **Implement** `Runnable`

If you extend the `Thread` class?

- You **inherit** *behavior* and *methods* of the Thread class
  - The `interrupt()` method is part of the `Thread` class
  - You can `interrupt()` *if you extend*

Advantages of using the `Runnable` interface

- Java provides several classes that handle threading *for* you
  - Implement pooling, scheduling, or timing
  - These require the **`Runnable`** interface

But what if I still can't decide?

- Do a UML model of your application
- The object hierarchy tells you what you need:
  - If your task needs to subclass another class?
    - Use `Runnable`
  - If you need to use methods of `Thread` within your class?
    - Use `Thread`

Threads and Objects

- Instance of the `Thread` class is just an **object**
  - Can be passed to other methods
  - If a thread has a reference to another thread
    - It can invoke *any method* of that thread's object
- The `Thread` object is not the thread itself
  - It is the set of methods and data that *encapsulate* information about the thread

But what does this mean?

- You cannot look at the object source and know *which thread is*:
  - Executing its methods or examining its data
- You may wonder about which thread is running the code, but …
  - There may be many possibilities

Determining the current thread

- Code within a thread object might want to see that code is being executed either:
  - By thread represented by the object or
  - By a completely different thread
- Retrieve reference to current thread
  - `Thread.currentThread()`
  - Static method

Checking which thread is executing the code

```
public class MyThread extends Thread {

   public void run() {
      if (Thread.currentThread() != this) {
         throw new IllegalStateException
            ("Run method called by incorrect thread …);
      } /* end if */

      ... Main logic
   }

}
```


Allowing a `Runnable` object to see if it has been interrupted

```
public class MyRunnable implements Runnable {

   public void run() {
      if (!Thread.currentThread().isInterrupted() ) {
            ... Main logic
      }
   }

}
```


The contents of this slide-set are based on the following references

- *Java Threads. Scott Oaks and Henry Wong. . 3rd Edition. O'Reilly Press. ISBN: 0-596-00782-5/978-0-596-00782-9.* [Chapters 3, 4]
- Operating Systems Principles and Practice. Thomas Anderson and Michael Dahlin. 2nd Edition. ISBN: 978-0985673529. [Chapter 4]