CSx55: Distributed Systems [Threads]

Threads block when they can't get that lock

Wanna have your threads stall?
Go ahead, synchronize it all

The antidote to this liveness pitfall?
Keeping the lock scope small

Shrideep Pallickara
Computer Science
Colorado State University

Computer Science Department   Colorado State University

1

Frequently asked questions from the previous class survey

- Typical cache hit rates?
- What if a thread does not fully utilize its allocated local mini heap? Is that not inefficient?
- A thread T_1 can execute instructions that belong to some other Thread object T_2?
- Is liveness stall same as a deadlock?
- Threads create threads? Is that the only way?
- How does blocking occur with a blocking call?

Colorado State University   Professor: Shrideep Pallickara Computer Science Department   Threads   L4.2

2

Topics covered in this lecture

- Threads
  - Thread Lifecycle
- Data synchronization
- Synchronized blocks
- Lock scope

3

4

Two approaches to stopping a thread

- Setting a flag
- Interrupting a thread

5

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

6

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

17

But what if I still can't decide?

- Do a UML (Unified Modeling Language) 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`

18

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

21

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
   }

}
```


22

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
      }
   }

}
```


23

24

Heisenbugs

- Term coined by ACM Turing Award winner Jim Gray
  - Pun on the name of Werner Heisenberg
  - Act of observing a system, alters its state!
- Describes a particular class of bugs
  - Those that disappear or change behavior when you try to examine them
- Multithreaded programs are a common source of Heisenbugs

What about regular bugs?

- Sometimes referred to as Bohr bugs
  - Deterministic
  - Generally, much easier to diagnose

Two friends plan to meet at Starbucks
But there are two Starbucks on College Avenue

	@ the First Starbucks Store	**@ the Second Starbucks Store**
12:10	**A** is looking for friend **B**	**B** is looking for friend **A**
12:15	**A** leaves for the second store	**B** leaves for the first store
12:20	**B** arrives at store	**A** arrives at store
12:30	**B** is Looking for friend **A**	**A** is looking for friend **B**
12:40	**B** leaves for the second store	**A** leaves for the first store

Both friends are now frustrated and undercaffeinated!

DATA SYNCHRONIZATION

Why sharing data between threads is problematic

- **Race conditions**
  - Correct outcome depends on lucky timing of uncontrollable events
- Threads attempt to access data more or less *simultaneously*
  - A thread may change the value of data that some other thread is operating on

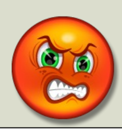
Example code with race condition

```
public class MyThread extends Thread {
   private byte[] values;
   private int position;

   public void
      modifyData(byte[] newValues, int newPosition) {
      ... Modify values and position
   }

   public void utilizeDataAndPerformFunction() {
      ... Use values and position
   }

   public void run() {
     ... Main logic
   }
}
```



In the previous snippet a race condition exists because …

- The thread that calls `modifyData()` is **accessing the same data** as the thread that calls `utilizeDataAndPerformFunction()`
- `utilizeDataAndPerformFunction()` and `modifyData()` **are not atomic**
  - It is possible that `values` and `position` are changed *while they are being used*

What is atomic?

- The code cannot be interrupted during its execution
  - Accomplished in hardware or *simulated* in software
- Code that cannot be found in an *intermediate state*

Eliminating the race condition using the `synchronized` keyword

- If we declared both `modifyData()` and `utilizeDataAndPerformFunction()` as **synchronized**?
  - Only one thread gets to call *either* method at a time
    - Only one thread accesses data at a time
  - When one thread calls one of these methods, while another is executing one of them?
    - The second thread must *wait*

Example code with no race conditions by using the synchronized keyword

```
public class MyThread extends Thread {
   private byte[] values;
   private int position;

   public void synchronized
      modifyData(byte[] newValues, int newPosition) {
      ... Modify values and position
   }

   public void synchronized
      utilizeDataAndPerformFunction() {
      ... Use values and position
   }

   public void run() {
     ... Main logic
   }
}
```

Revisiting the mutex lock

- **Mut**ually **ex**clusive lock
- If two threads try to grab a mutex?
  - Only one succeeds
- In Java, every object has an associated **lock**

When a method is declared `synchronized` ...

- The thread that wants to execute the method must **acquire** a lock
- Once the thread has acquired the lock?
  - It executes method and **releases** the lock
- When a method returns, the lock is released
  - Even if the return is because of an exception

Locks and objects

- There is only **one lock per object**
- If two threads call synchronized methods of the same object?
  - Only one can execute immediately
    - The other has to wait until the lock is released

37

Afraid of what the truth might bring
He locks his doors and never leaves
Desperately searching for signs
To terrify, to find a thing
He battens all the hatches down
And wonders why he hears no sound
Frantically searching his dreams
He wonders what it's all about

Telescope, Cage the Elephant

SYNCHRONIZATION PITFALLS

38

Another code snippet to look at …

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

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

   public void setDone(boolean isDone) {
     done = isDone;
   }
}
```

Can't we just synchronize the two methods as we did previously?

- If we synchronized both `run()` and `setDone()` ?
  - `setDone()` would never execute!
- The `run()` method does not exit until the `done` flag is set
  - But the `done` flag cannot be set because `setDone()` cannot execute till `run()` completes
- Uh oh …

The problem stems from the scope of the lock

- **Scope of a lock**
  - Period between grabbing and releasing a lock
- Scope of the `run()` method is too large!
  - Lock is grabbed and never released
- We will look at techniques to *shrink the scope* of the lock
- But let's look at another solution for now

Let's look at operations performed on the data item (`done`)

- The `setDone()` method stores a value into the flag
- The `run()` method reads the value

- In our previous example:
  - Threads were accessing *multiple* pieces of data
  - No way to update multiple data items *atomically* without the `synchronized` keyword

But Java specifies that the loading and storing of variables is atomic

- Except for `long` and `double` variables
- The `setDone()` should be atomic
  - The `run()` method has only one read operation of the data item
- The race condition should not exist
  - But why is it there?

Threads are allowed to hold values of variables in registers

- When one thread changes the value of the variable?
  - Another thread *may not see* the changed variable
- This is particularly true in loops controlled by a variable
  - Looping thread **loads value of variable in register** and *does not notice* when value is changed by another thread

Two approaches to solving this

- Providing setter and getter methods for variable and using the `synchronized` keyword
  - *When lock is acquired*, temporary values stored in registers are *flushed* to main memory
- The **`volatile`** keyword
  - Much cleaner solution

If a variable is marked as `volatile`

- Every time it is used?
  - Must be read from main memory
- Every time it is written?
  - Must be written to main memory
- Load and store operations are **atomic**
  - Even for `long` and `double` variables

Some more about volatile variables

- Prior to JDK 1.2 variables were always read from main memory
  - Using volatile variables was moot
- Subsequent versions introduced memory models and optimizations

Synchronization and the volatile keyword

- Can be used *only* when operations use a **single load and store**
  - Operations like `++`, `--`?
    - Load-change-store …
- The `volatile` keyword forces the JVM to not make temporary copies of a variable
- Declaring an array `volatile`?
  - The reference becomes volatile
  - The individual elements are not volatile

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]