Java Threads in a Nutshell

Chris Kauffman

Last Updated:
Wed Mar 30 10:55:17 AM CDT 2022

Logistics

Schedule

11/15	Mon	OpenMP Wrap
		Java Threads
11/17	Wed	CUDA / GPUs
		Mini-Exam 3
11/22	Mon	CUDA / GPUs
11/24	Wed	CUDA / GPUs
	Thu	**Thanksgiving**
11/29	Mon	CUDA / GPUs
		Applications
12/01	Wed	Guest Lecture

Threads Reading

Java Concurrency Tutorial

GPU Reading

GPU parallel program development using CUDA by Tolga Soyata

- Ch 6 start GPU Coverage
- UMN Library Link

Logistics

A2

- Release Later this week
- 30% of grade
- 3 Problems: 2 MPI + 1 OpemMP
- Makeup Credit Available
- Due at End of Semester

A3

- 10% CUDA Programming, Short
- Release after Thanksgiving
- Due at End of Semester

Mini-Exam 3

- Focus on Threads, OpenMP, Java Threads
- Contrast approaches in each
- Short code optimizations similar to lecture examples

Later Exams

- Mini-Exam 4: CUDA + Applications
- Final Exam: Cumulative
- Separately worth 10%
- Merge to 20% last-day-of-class exam?

Threads in Java

- Java was built with concurrency in mind
- `java.lang.Thread` is a core part of the language
- Represents a runnable unit
- `java.lang.Runnable` does so similarly as an interface

Typical Parallel Setup

- Create a class which `extends Thread`
- Override the `run()` method to do real work
- In a `main()` method, instantiate the thread class and invoke the `thread.start()`, thread begins execution asynchronously
- Eventually call `thread.join()` to wait for thread to finish

Picalc example (again)

Three variants

1. Reduction version: no thread synchronization required
2. Synchronized methods: only one thread executes a method at a time
3. Synchronized statements: any java object can be a lock

Highlights of PicalcReduction.java

- Create a nested subclass

  ```
  static class CalcThread extends Thread
  ```

- Fields and constructor allows initialization information to be communicated

  ```
  public CalcThread(int threadNum, int nPoints){
    this.threadNum = threadNum;
    this.nPoints = nPoints;
  }
  ```

- Override the `public void run()` to initialize a `Random` number generator, perform hit computations, update `this.hits`
- Accessor `public int getHits()` allows retrieval of hits computed
- `main()` method creates an array of `CalcThreads`, starts them running
- `join()` each thread to wait for it to finish, sum up hits with

  ```
  threads[i].join();
  totalHits += threads[i].getHits();
  ```

- Must be aware of an irritating `InterruptedException`

Highlights of `PicalcSynchMethod.java`

- Class field to track total hits

  ```
  static int totalHits;
  ```

- Class method to control updates: **synchronized**

  ```
  static synchronized void incrTotal(){
    totalHits++;
  }
  ```

- Only one thread in the method at a time
- Nested class `CalcThread` calls `incrTotal()` to update `totalHits`
- `main()` spins up threads and waits to join
- No need to perform any reductions

Highlights of `PicalcSynchStatement.java`

- Class field to track total hits

  ```
  static int totalHits;
  ```

- Class field to serve as a lock to control access

  ```
  static Object lock = new Object();
  ```

- Nested class `CalcThread` directly updates by acquiring/releasing lock

  ```
  synchronized(lock){
    totalHits++;
  }
  ```

- Only one thread in the critical section at time
- `main()` spins up threads and waits to join
- No need to perform any reductions

Timings of Java Variants

```
>> cd 13-java-threads-code
>> time java PicalcReduction
npoints: 75000000
threads: 4
hits:    58909237
pi_est:  3.141826

real    0m0.982s
user    0m3.401s
sys     0m0.071s
```

samples	75M	75M	75M
threads	1	2	4
JAVA			
PicalcReduction	3.378	1.803	0.988
PicalcSynchMethod	4.272	7.755	10.623
PicalcSynchStatement	3.366	7.753	10.866
PTHEADS			
`mutex_fast`	1.032	0.521	0.268

Java

Keep in mind the slower times might be improved with tweaks to compilation + runtime invocation, perhaps with ahead of time compilation via `jaotc`. The JVM does a LOT of stuff favoring flexibility over performance

Exercise: Java Collisions

- `Collisions.java` gives a serial collision detector
- Discuss with a neighbor how to parallelize this in Java
- Will require a `Thread` subclass
- Discuss using reductions or `synchronized` methods/statements
- Be fairly specific with your design: sketch subclasses, fields, methods

Note on Synchronized Sections

- Synchronized methods are synced on the associated object
- Only one thread is in ANY method at a time
- Maintain consistency of object state
- `static` methods sync on class, can only be in one at a time

```
class C {
  int total;
  public C(){ this.total = 0;  }

  synchronized void incrTotal(){
    total++;
  }
  synchronized void decrTotal(){
    total++;
  }
}
```

```
class D {
  static int total;


  synchronized static void incrTotal(){
    total++;
  }
  synchronized static void decrTotal(){
    total++;
  }
}
```

Contrast

- Unlike the new collection implementations, `Vector` is synchronized.
- `ArrayList`: Note that this implementation is not synchronized.

Example of Easy Creation of a Synchronized Instance

From `ArrayList` Java Docs

Note that this implementation is not synchronized. If multiple threads access an ArrayList instance concurrently, and at least one of the threads modifies the list structurally, it must be synchronized externally. (A structural modification is any operation that adds or deletes one or more elements, or explicitly resizes the backing array; merely setting the value of an element is not a structural modification.) This is typically accomplished by synchronizing on some object that naturally encapsulates the list. If no such object exists, the list should be "wrapped" using the Collections.synchronizedList method. This is best done at creation time, to prevent accidental unsynchronized access to the list:

```
List list = Collections.synchronizedList(new ArrayList(...));
```

Question

What does the code for `synchronizedList(..)` look like?

Iterators are Inherently Serial

Manual synchronization on iterators is still required.

```
synchronized (list) {
  Iterator i = list.iterator();
  while (i.hasNext())
    foo(i.next());
}
```

- Required if another thread is performing `list.add(x)`
- Prevents `ConcurrentModificationException`

Wait, Notify, Volatility

```
class C {
  public volatile boolean joy = false;
  public void guardedJoy() {
    while(!joy) {}                    // Busy polling
    System.out.println("Joy has been achieved!");
  }

  public synchronized void guardedJoy() {
    while(!joy) {
      try {
        this.wait();                  // Blocking wait
      } catch (InterruptedException e) {}
    }
    System.out.println("Joy and efficiency have been achieved!");
  }
  public synchronized notifyJoy() {
    this.joy = true;
    this.notifyAll();
  }
}
```

See: `WaitNotify.java` for timings

Java's Memory Model

At the bottom of this issue lies the need for aggressive optimization in the face of concurrency: any mechanism which ensures memory coherency between threads is expensive, and much (most) of the data is not shared between threads. Therefore the data not explicitly marked volatile, or protected by locks, is treated as thread-local by default (without strict guarantees, of course).
– *Marko Topolnik, Stack Overflow*

Exercise: False Sharing in Java

- A subtle performance issue may occur in Java
- How might **false sharing** happen in the nearby Picalc computation?
  Hint: where are the `CalcThread` objects allocated?
- How can such false sharing be avoided?

```
public class PicalcReduction {
  static class CalcThread extends Thread{
    int hits; // hits in this thread

    public void run(){
      ...;
      if (x*x + y*y <= 1.0){
        this.hits++;
      }
    }
  }

  public static void main(String args[]) {
    ...;
    for(int i=0; i<nThreads; i++){
      threads[i] = new CalcThread(i, ...);
      threads[i].start();
    }
    ...;
  }
}
```

Answers: False Sharing in Java 1 / 2 (Problem)

- **Heap-allocated data** may be on the same cache line
- `CalcThread` objects are heap allocated and their fields `this.hits` are incremented
- If the objects cross cache lines, may result in false sharing

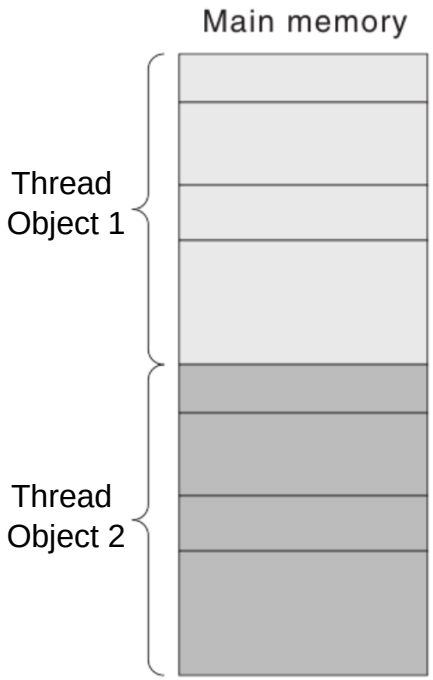

Figure 9.1 Per-thread variable memory layout

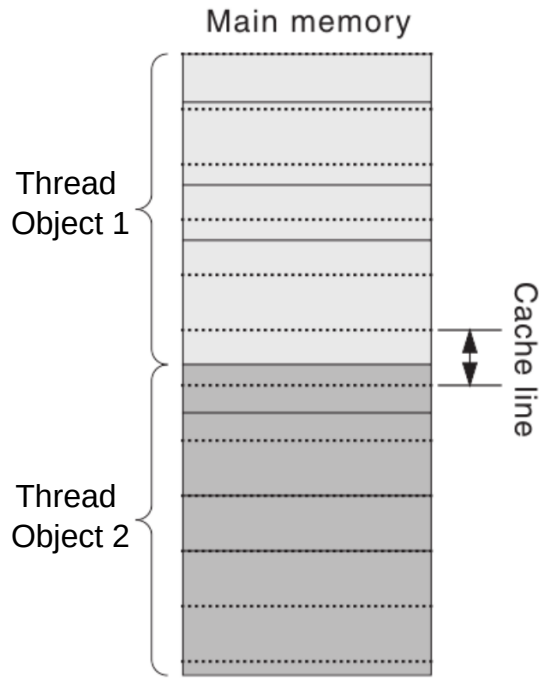

Figure 9.2 Memory layout showing cache line boundaries

Source: Building Parallel Programs, Kaminsky

Answers: False Sharing in Java 2 / 2 (Fixes)

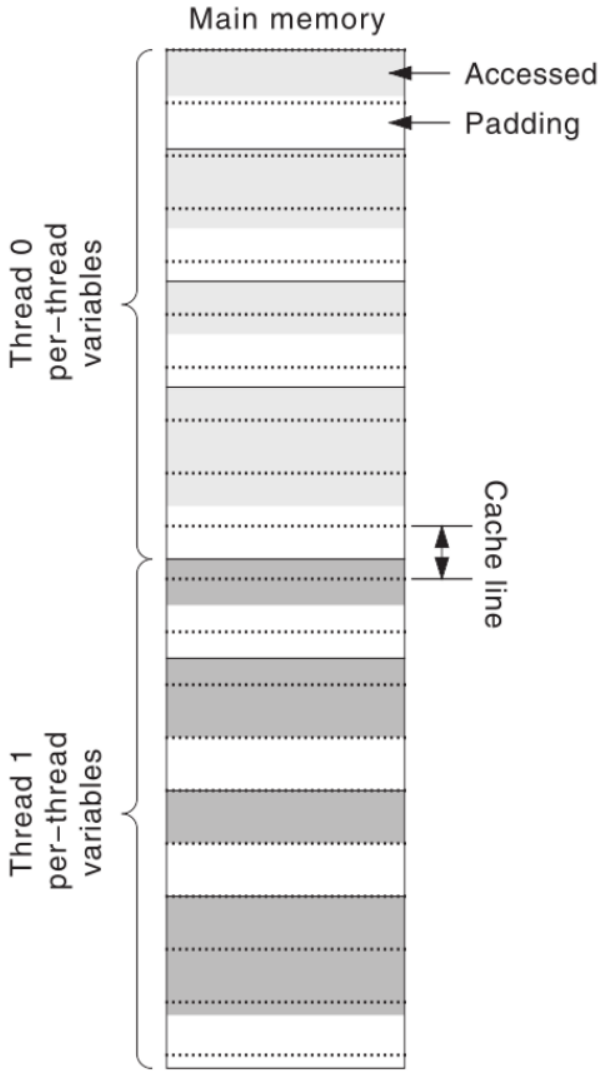

Figure 9.3 Memory layout with extra padding

```
public class PicalcReduction {
  static class CalcThreadPad extends Thread{
    int hits; // hits in this thread
    long pad0, pad1, pad2, pad3, // padding
         pad4, pad5, pad6, pad7;
    public void run(){
      ...;
      if (x*x + y*y <= 1.0){
        this.hits++;
      }
    }
  }
  static class CalcThreadLoc extends Thread{
    int hits; // hits in this thread
    public void run(){
      int myhits = 0; // stack local
      ...;
      if (x*x + y*y <= 1.0){
        myhits++;
      }
    }
    this.hits = myhits;
  }
}
```

Other Capabilities in Java

Generally concurrency is a prime part of Java and one of its strengths. Capabilities continue to evolve so one might explore any or all of the following for better concurrency and parallelism.

- Concurrent collections (`ConcurrentMap` rather than `HashMap` and `TreeMap`)
- `Runnable` interface - class provides a `run()` method

```
Runnable r = new Something();
Thread t = new Thread(r);
```

- `Executor` interface and associates for more complex scheduling
- Use of `ThreadPools` to farm out work
- New-ish `ForkJoinPool`