Practical Concurrent and Parallel Programming VI

Performance and Scalability

Jørgen Staunstrup

Agenda

- **Executors and Future (continued from Week 5)**
- Scalability, speed-up and loss (of scalablity) classification
  Example: QuickSort
- Lock striping
  - A case study with Hash maps

From week05 splitting tasks

Body of: countPrimesTask

```
@Override
  public void run() {
  if ((high-low) < threshold) {
      for (int i=low; i<=high; i++)  if (isPrime(i)) lc.increment();
  } else {
      int mid= low+(high-low)/2;
      pool.submit(new countPrimesTask(lc, low, mid, pool, threshold) );
      pool.submit(new countPrimesTask(lc, mid+1, high, pool, threshold) );
  }
}
```

Shortcomings:

1. How to stop?
2. ~~Will create too many "small" tasks~~
3. Returning result (# primes)

Splitting tasks


```
public void run() {
  if ((high-low) < threshold) {      ...
    } else {
      int mid= low+(high-low)/2;
      Future<?> f1= pool.submit( new countPrimesTask(lc, low, mid,
                                 pool, threshold) );
      Future<?> f2= pool.submit( new countPrimesTask(lc, mid+1, high,
                                 pool, threshold) );
      ...

    }
  }
```

Combining tasks


```
public void run() {
  if ((high-low) < threshold) {      ...
    } else {
      int mid= low+(high-low)/2;
      Future<?> f1= pool.submit( new countPrimesTask(lc, low, mid,
                                 pool, threshold) );
      Future<?> f2= pool.submit( new countPrimesTask(lc, mid+1, high,
                                 pool, threshold) );
      try {  f1.get();f2.get(); }
      catch (InterruptedException | ExecutionException e) {   }
    }
  }
```

Does the order of f1.get and f2.get matter?

Combining tasks


```
public void run() {
  if ((high-low) < threshold) {      ...
    } else {
      int mid= low+(high-low)/2;
      Future<?> f1= pool.submit( new countPrimesTask(lc, low, mid,
                                        pool, threshold) );
      Future<?> f2= pool.submit( new countPrimesTask(lc, mid+1, high,
                                        pool, threshold) );
      try {  f1.get();f2.get(); }
      catch (InterruptedException | ExecutionException e) {   }
    }
  }
```

Shortcomings:

1. ~~How to stop?~~
2. ~~Will create too many "small" tasks~~
3. Returning result (# primes)

How do we get the result?

Java Executors - Tasks

- **Tasks** are a central concept for executors
- When designing a program using executors, first think about the tasks to be executed (e.g. countPrimesTask)
  - Like for threads, tasks can be conveniently defined in their own class

countPrimesTask


```
public class countPrimesTask implements Runnable {
  private final int low;
    ...

  private static boolean isPrime(int n) {
    ...
  }

  public countPrimesTask(PrimeCounter lc, int low, int high,
   ExecutorService pool, int threshold) {
    this.lc        = lc;
    ...
  }

  @Override
  public void run() {
    ...
  }
}
```

Java Executors - Tasks

- **Tasks** are a central concept for executors
- When designing a program using executors, first think about the tasks to be executed (e.g. countPrimesTask)
  - Like for threads, tasks can be conveniently defined in their own class
- Ideally, tasks should be independent

Why?

countPrimesTask class


```
public class countPrimesTask implements Runnable {
  private final int low;  private final int high;  priva
  private final ExecutorService pool;
  private final PrimeCounter lc;

  private static boolean isPrime(int n) {...)
  }

  public countPrimesTask(PrimeCounter lc... ) {this.lc= lc;...  }

  @Override
  public void run() {
    if ((high-low) < threshold) {
      for (int i=low; i<=high; i++) if (isPrime(i)) lc.increment();
   } else {
      int mid= low+(high-low)/2;
      Future<?> f1= pool.submit( new countPrimesTask(lc, ... ) );
      Future<?> f2= pool.submit( new countPrimesTask(lc, ... ) );
      try {  f1.get();f2.get();      }
      catch (InterruptedException | ExecutionException e) {   e.printStackTrace();
}
     }
  }
}
```

Are countPrimesTasks independent?

Code in `countPrimesTask.java`

PrimeCounter


```
class PrimeCounter {

  private int count= 0;
  public synchronized void increment() {
    count= count + 1;
  }
  public synchronized int get() {
    return count;
  }
  public synchronized void setZero() {
    count= 0;
  }
}
```

PrimeCountExecutor class

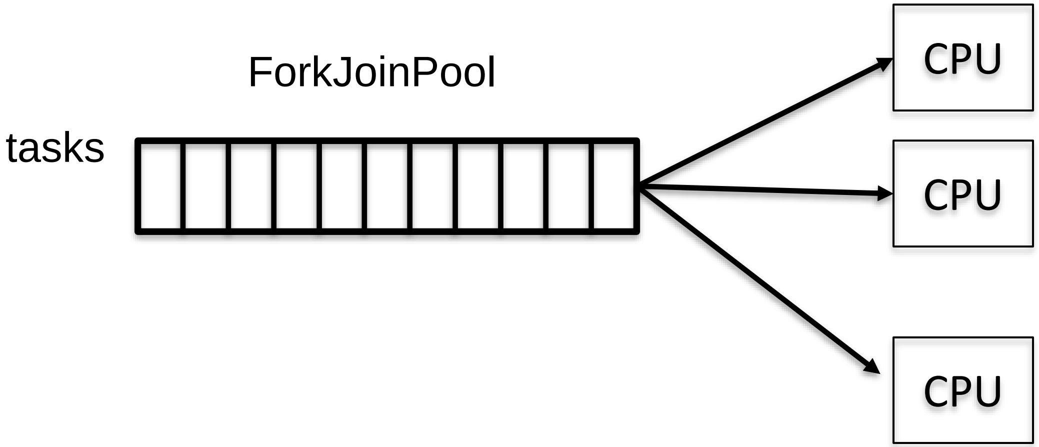
- Kick-off class for the program
- It initializes the Executor service

```
class PrimeCountExecutor {
   private ExecutorService pool;
     ...
   public PrimeCountExecutor () {
     pool= new ForkJoinPool();
     Future<?> done= pool.submit(new countPrimesTask( ... ));

     try {  done.get();  }

    }
}
```

https://docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html

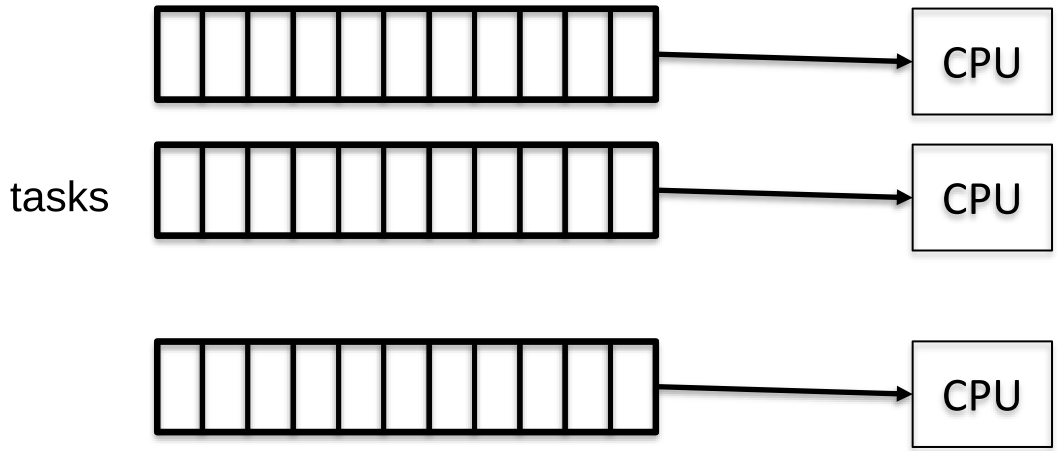
Thread pools


```
class PrimeCountExecutor {
   private ExecutorService pool;
     ...
   public PrimeCountExecutor () {
     pool= new ForkJoinPool(1);
     ...
}
```



Thread pools


```
class PrimeCountExecutor {
   private ExecutorService pool;
     ...
   public PrimeCountExecutor () {
     pool= newFixedThreadPool(n);

     ...
}
```

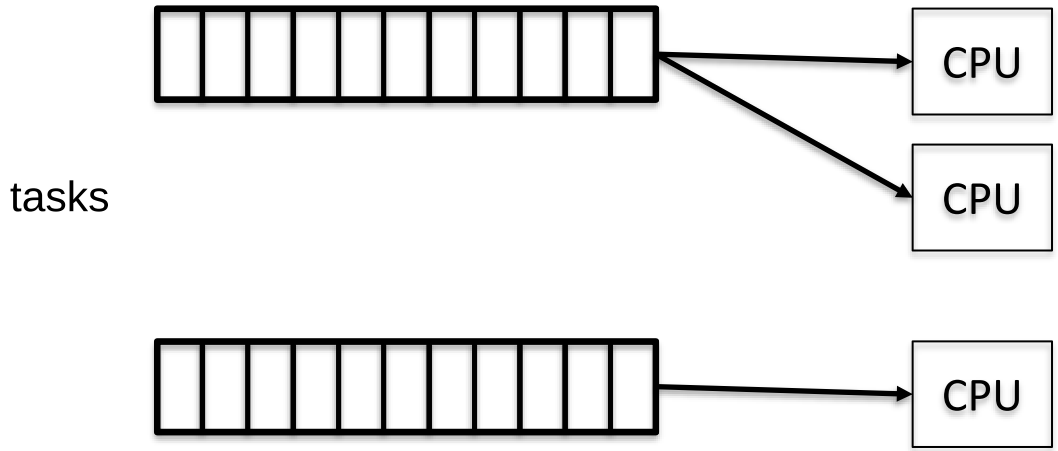


https://docs.oracle.com/javase/tutorial/essential/concurrency/pools.html

Thread pools


```
class PrimeCountExecutor {
   private ExecutorService pool;
     ...
   public PrimeCountExecutor () {
     pool= newWorkStealingPool(x);

     ...
}
```

Quite a few more types of ExecutorService



ForkJoinPool


```
import java.util.concurrent.ForkJoinPool;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Future;

class PrimeCountExecutor {
   private ExecutorService pool;
     ...
   public PrimeCountExecutor () {
     pool= new ForkJoinPool( n );
     ...

    }
}
```

n specifies the number of cores that will be used

default value: Runtime.availableProcessors()

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ForkJoinPool.html

Agenda

- Executors and Future
- **Scalability, speed-up and loss (of scalablity) classification**
  Example: QuickSort
- Lock striping
  - A case study with Hash maps

Quicksort

1 2 43 78 19 54 33 21 64 52 17 53

1 2 43 78 19 54 33 21 64 52 17 53

1 2 43 78 19 54 33 21 64 52 17 53

1 2 17 78 19 54 33 21 64 52 43 53

1 2 17 78 19 54 33 21 64 52 43 53

...

1 2 17 21 19 33 54 78 64 52 43 53

1 2 17 21 19 33 54 78 64 52 43 53

Two parts can be sorted independently

Distributing work to threads

1 2 17 21 19 33 54 78 64 52 43 53

…

No further splitting when the sorting problem is smaller than a threshold (similarly to what we did for prime counting)

These tasks may differ in size !!

Quicksort executor (pseudo code)


```
class QuicksortTask implements Runnable {
  Task p;  // low and high boundaries
  ExecutorService pool;

  @Override  public void run() { qsort(p, pool, ... ); }

  public static void qsort(Task p, ExecutorService pool, ... ) {
     //split task in two: Low and High

    if (Low.size>= threshold) pool.submit( new QuicksortTask( pLow, pool, ... ))
     else Quicksort(pLow); //sequential sort

    if (High.size>= threshold) pool.submit( new QuicksortTask( pHigh, pool, ... ))
     else Quicksort(pHigh);

  }
}
```

See complete code in: code-lecture/ …/QuickSortTask.java

Sorting results

`new ForkJoinPool(4)`

Executor	
1	8.5 s
2	4.8 s
4	2.6 s
8	2.2 s
16	2.2 s

1/8: 3.9

Threads	
1	11.2 s
2	6.4 s
4	3.8 s
8	3.2 s
16	3.5 s

1/8: 3.9

Sorting 100_000 number

Count prime results

Executor		Threads	
1	120.6 s	1	126.7 s
2	68.0 s	2	82.4 s
4	37.7 s	4	47.7 s
8	32.2 s	8	38.2 s
16	32.4 s	16	37.2 s

Executor 1/8: 3.7

Threads 1/8: 3.9

range 2.. 1_000_000

Quicksort

1 2 43 78 19 54 33 21 64 52 17 53

1 2 43 78 19 54 33 21 64 52 17 53

1 2 43 78 19 54 33 21 64 52 17 53

1 2 17 78 19 54 33 21 64 52 43 53

1 2 17 78 19 54 33 21 64 52 43 53

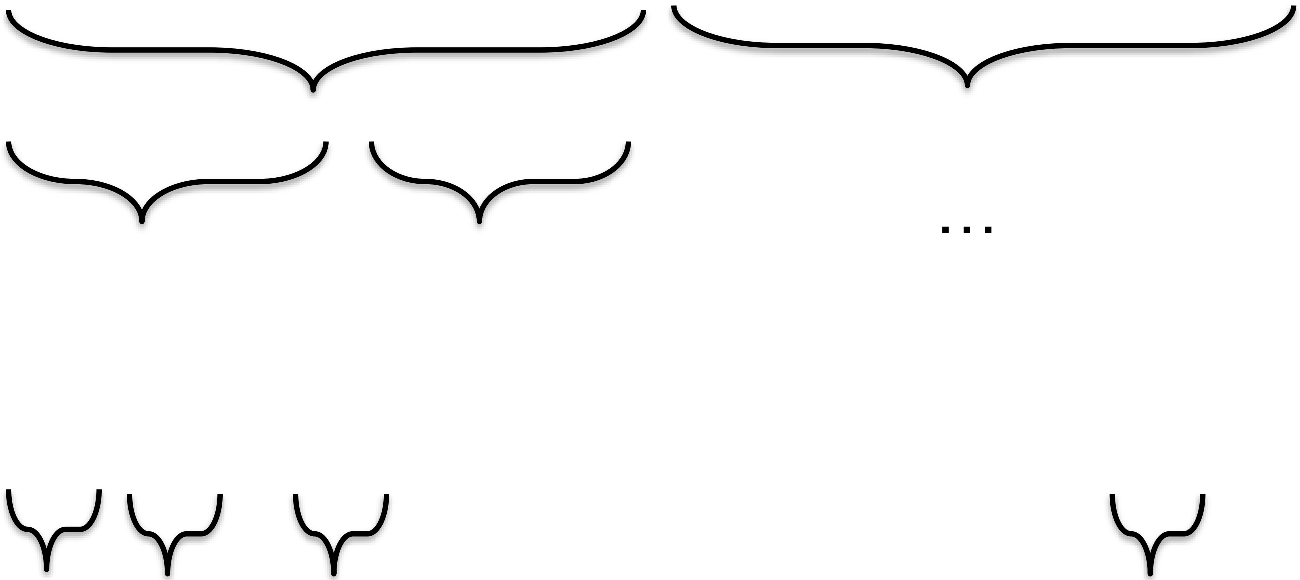
...

1 2 17 21 19 33 54 78 64 52 43 53

1 2 17 21 19 33 54 78 64 52 43 53

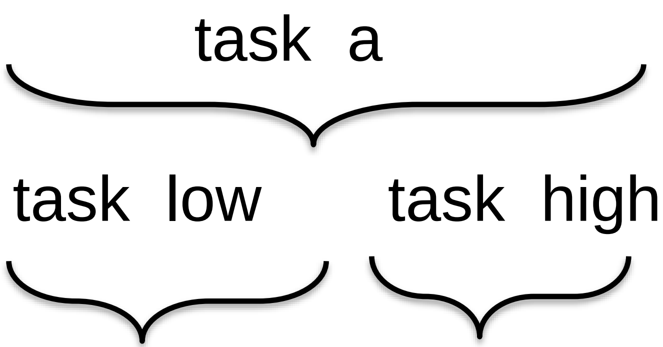
Two parts can be sorted independently

Termination



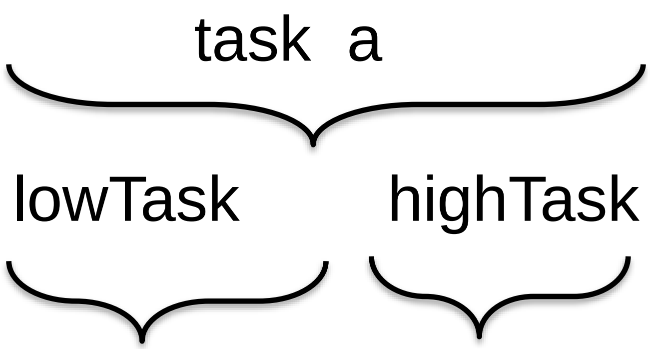
How do we know when all task are done?

Termination



When can task a finish?

Termination



`lowTask.get()` returns when lowTask has finished

`highTask.get()` returns when highTask has finished

```
@Override
 public void run() { // modified Quicksort using Executor tasks
      ...

   //Waiting for longest running subtask to finish
   try {
     lowTask.get();
     highTask.get();
   } catch (InterruptedException | ExecutionException e) { e.printStackTrace(); }
 }
```

Quicksort Executor

Code in QuickSortTask.java

```
@Override
 public void run() { // modified Quicksort using Executor tasks
   int a= low;  int b= high;
   Future<?> lowTask= null;    Future<?> highTask= null;

   if (a < b) { … // split array in two independent part

     if ((j-a)>= threshold)
       lowTask= pool.submit(new QuickSortTask(arr, a, j, pool, threshold));
     else // all remaining work done without starting more tasks
       SearchAndSort.qsort(arr, a, j);

     if ((b-i)>= threshold)
       highTask= pool.submit(new QuickSortTask(arr, i, b, pool, threshold));
     else // all remaining work done without starting more tasks
       SearchAndSort.qsort(arr, i, b);
   }

   //Waiting for longest running subtask to finish
   try {
     if (lowTask != null ) lowTaskF.get();
     if (highTask != null) highTaskF.get();
   } catch (InterruptedException | ExecutionException e) { e.printStackTrace(); }
 }
```

Java code for Quicksort (2)

“Measure, don’t guess”
Goetz p. 224

We use Mark8Setup to measure runtime

```
private static void runSize(ExecutorService pool, int pSize, int threshold, int n) {
  final int[] intArray= fillIntArray(pSize);
  Benchmark.Mark8Setup("Quicksort Executor", String.format("%2d", n),
    new Benchmarkable() {

      public void setup() {
        shuffle(intArray);
      }

      public double applyAsDouble(int i) {
        Future<?> done= pool.submit(new QuickSortTask(intArray, 0, pSize-1, pool, threshold));
        PoolFinish(done);
        //testSorted(intArray); //only needed while testing
        return 0.0;
      }
    }
  );
}
```

Code in `PoolSortingBenchmarkable.java`

Other types of "loss" of scalability

- Starvation loss
  - Minimize the time that the task pool is empty
- Separation loss (best threshold)
  - Find a good threshold to distribute workload evenly
- Saturation loss (locking common data structure)
  - Minimize high thread contention in the problem
- Braking loss
  - Stop all tasks as soon as the problem is solved

Møller-Nielsen, P and Staunstrup, J, Problem-heap. A paradigm for multiprocessor algorithms. *Parallel Computing*, 4:63-74, 1987

Shut down

The ExecutorService can be shut down.

```
 // Executor body
...
...

pool.shutdown();
```

The challenge is often *when to shut down*

After shutdown the pool cannot be reused,
but you may assign it a new value (of type **`ExecutorService`**)

Runnable vs. Callable

Both are used to specify the code of a thread.

- Runnable cannot return a result
  - Overrides run()
- Callable returns a result (via a Future)
  - Overrides call()

Could Callables use shared data as well?

As illustrated by the Quicksort and countPrimes examples, Runnables may use shared data (e.g., to deliver a result)

Futures are an example of message passing (coming weeks)

Submit vs Execute

Both are used to spawn a task

- **`pool.execute`** cannot return a result
  may complicate determining when to finish

- **`pool.submit`** returns a result (via a Future)

https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Executor.html

dotProduct – Task class

Given two vectors *x, y* of equal size, their dot product equals $\sum_i x_i y_i$

```
public class DotProductTask implements Callable<Integer> {
     final int pos;
     final int[] x, y;

     public DotProductTask(int[] x, int[] y, int pos) {
          this.x   = x;
          this.y   = y;
          this.pos = pos;
     }

     @Override
     public Integer call() {
          return x[pos] * y[pos];
     }
}
```

We use futures to multiple each vector position

dotProduct – Future execution

Given two vectors x, y of equal size, their dot product equals $\sum_i x_i y_i$

```
...
List<DotProductTask> tasks = new ArrayList<DotProductTask>();

// Randomly initialize arrays ...

// Create the list of tasks (Futures) to execute
for (int i = 0; i < N; i++)
    tasks.add(new DotProductTask(x,y,i));

...

// Add all futures to the execution pool at once
List<Future<Integer>> futures = pool.invokeAll(tasks);
for(Future<Integer> f : futures) {
     result += f.get(); // Wait for each future to be executed
                        // and add partial result

pool.shutdown(); // We are sure to be done, so we shut down the pool
...
```

Code in `FuturesDotProduct.java`

What limits performance?

CPU-bound

- Eg. counting prime numbers
- To speed up, add more CPUs (cores) (*exploitation*)

Input/output-bound

- Eg. I/O or reading from network
- To speed up, use more threads/tasks (*inherent*)

Synchronization-bound (Saturation loss)

- Eg. Algorithm using shared data structure
- **To speed up, improve shared data structure**

  (*Rest of this lecture*)

Agenda

- Executors and Future
- Scalability, speed-up and loss (of scalablity) classification
  Example: QuickSort
- **Lock striping**
  - **A case study with Hash maps**

Scalability of Java Collections

A *collection* is simply an object that groups multiple elements into a single unit

Package: java.util

Examples: ArrayList, HashMap, TreeSet, …

https://docs.oracle.com/javase/tutorial/collections/intro/index.html

Methods: add, remove, size, contains, …

Many of the classes have thread-safe/concurrent implementations

https://www.baeldung.com/java-synchronized-collections

Example: synchronizedCollection


```
import java.util.*;
public class syncCollectionExample {
  public static void main(String[] args) {  new syncCollectionExample(); }

  public String getLast(ArrayList<String> l) {
    int last= l.size()-1;
    return l.get(last);
  }

  public static void delete(ArrayList<String> l) {
    int last= l.size()-1;
    l.remove(last);
  }

  public syncCollectionExample() {
    ArrayList<String> a= new ArrayList<String>();
    a.add("A");  ...

    Collection<String> synColl = Collections.synchronizedCollection(a);
    ...
  }
}
```

Thread-safe?

Goetz p. 80

Thread-safety (from week 3)

It is very important to note that for a program *p*:

p only accesses thread-safe classes

$\not\Rightarrow$

p is a thread-safe program

Making the synchronized ArrayList thread safe

```
import java.util.*;
public class syncCollectionExample {
  public static void main(String[] args) {  new syncCollectionExample(); }

  public String getLast(ArrayList<String> l) {
    synchronized(l) {
      int last= l.size()-1;
      return l.get(last);
    }
  }

  public static void delete(ArrayList<String> l) {
    synchronized(l) {
      int last= l.size()-1;
      l.remove(last);
    }
  }

  public syncCollectionExample() {
   ...
  }
}
```

Goetz p. 80

What if the data structure is huge?

and used by many threads?

for example:

a bank
Facebook updates
…

Would not work if everything is "synchronized"

What can we do?

Reduce locking !!

Example: A huge HashMap

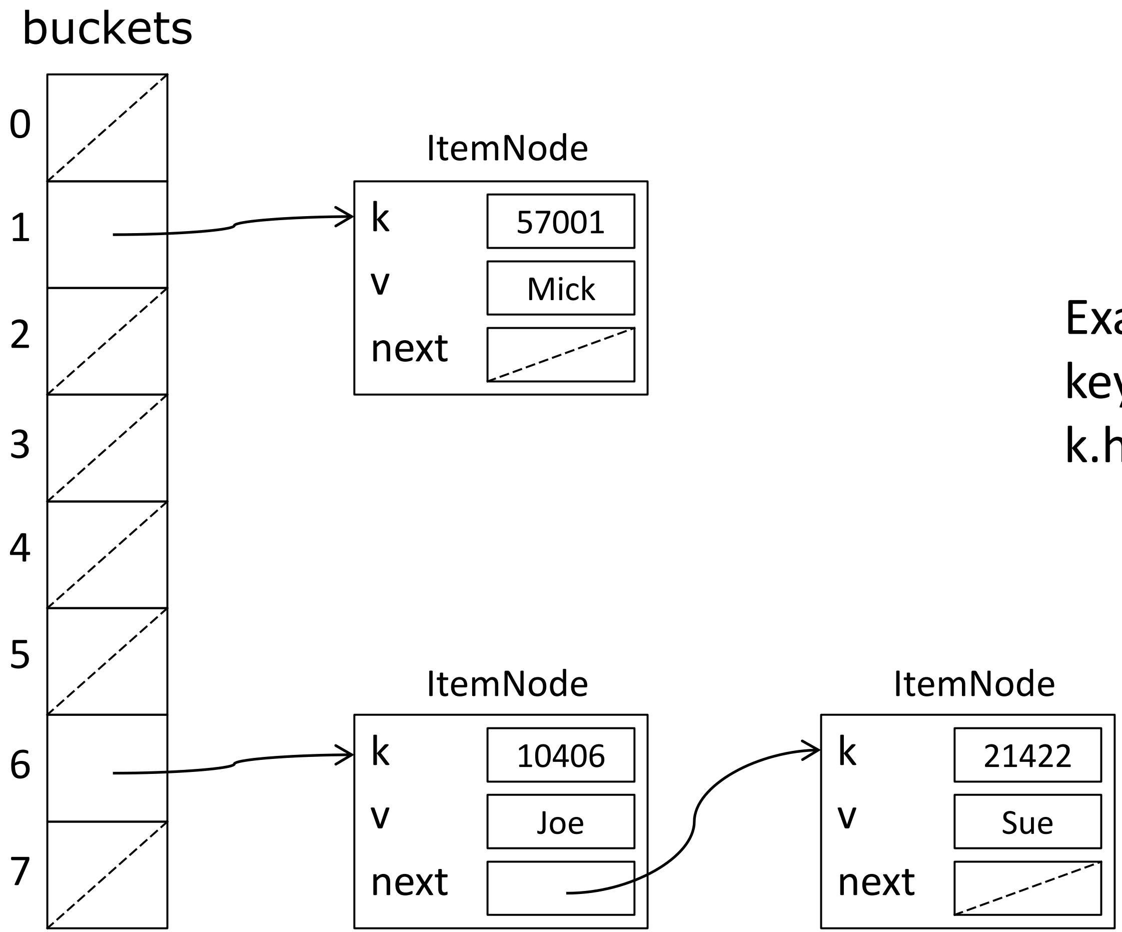
Key value pairs: <k1, v1>, <k2, v2>, …

Key	Value
Peter	20487612
Anna	51251218
Lena	34458318
Holger	89545010
Lisa	94959500

```
class HashMap<K,V> {
  ...  // data structure
  public V get(K k) { ... }
  public V put(K k, V v) { ... }
  public boolean containsKey(K k) { ... }
  public int size() { return cachedSize; }
  public V remove(K k) { ... }
  ...
}
```

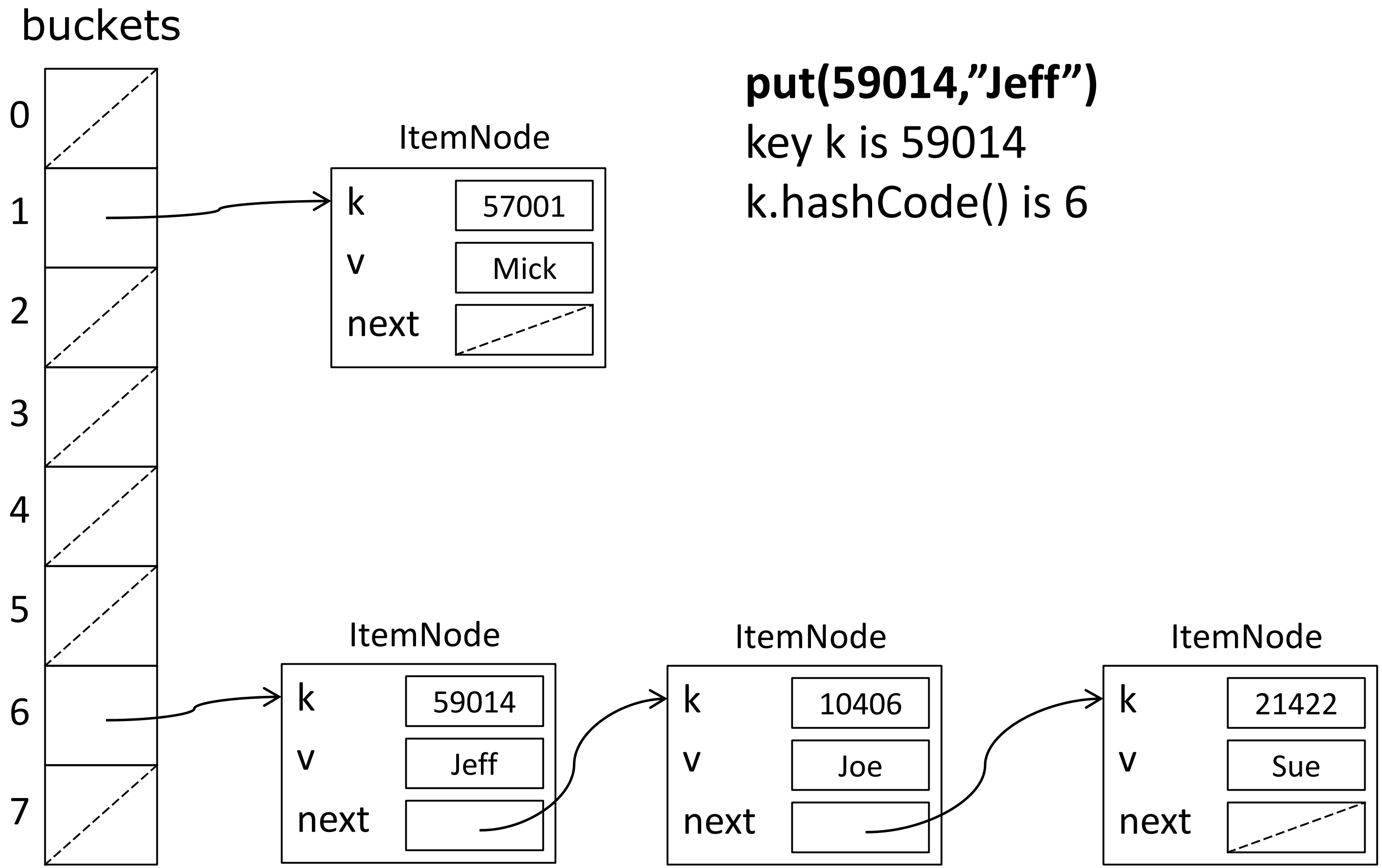
How to make it thread-safe?
(without making all the methods synchronized)

HashMap implementation



Example **get(10406)**
key k is 10406
k.hashCode() is 6

HaspMap put



put(59014,”Jeff”)
key k is 59014
k.hashCode() is 6

Synchronized implementation

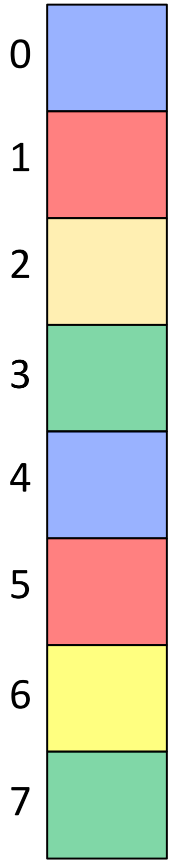

```
static class ItemNode<K,V> {
  private final K k;
  private V v;
  private ItemNode<K,V> next;
  public ItemNode(K k, V v, ItemNode<K,V> next) { ... }
}
```

```
class SynchronizedMap<K,V> {
  private ItemNode<K,V>[] buckets;  // guarded by this
  private int cachedSize;           // guarded by this
  public synchronized V get(K k) { ... }
  public synchronized boolean containsKey(K k) { ... }
  public synchronized int size() { return cachedSize; }
  public synchronized V put(K k, V v) { ... }
  public synchronized V remove(K k) { ... }
}
```

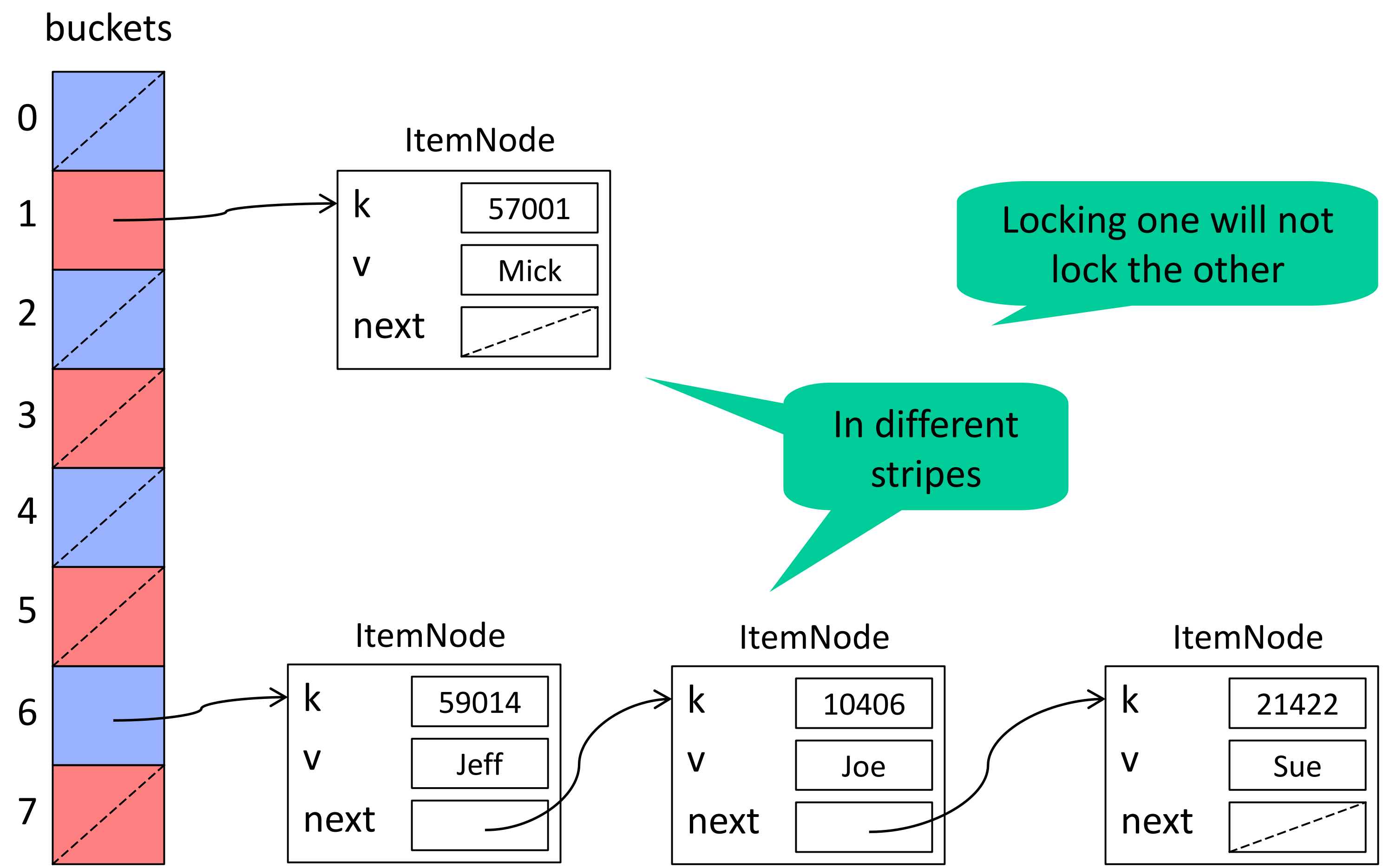
Improving scalability – Lock striping

- Guarding the table with a single lock works
  - ... but does not scale well (actually **very** badly)
- Idea: Each bucket could have its own lock
- In practice
  - use fewer, to illustrate we use 4, locks
  - guard every 4th bucket with the same lock
  - locks[0] guards bucket 0, 4, 8, ... (stripe 0)
  - locks[1] guards bucket 1, 5, 9, ... (stripe 1) et

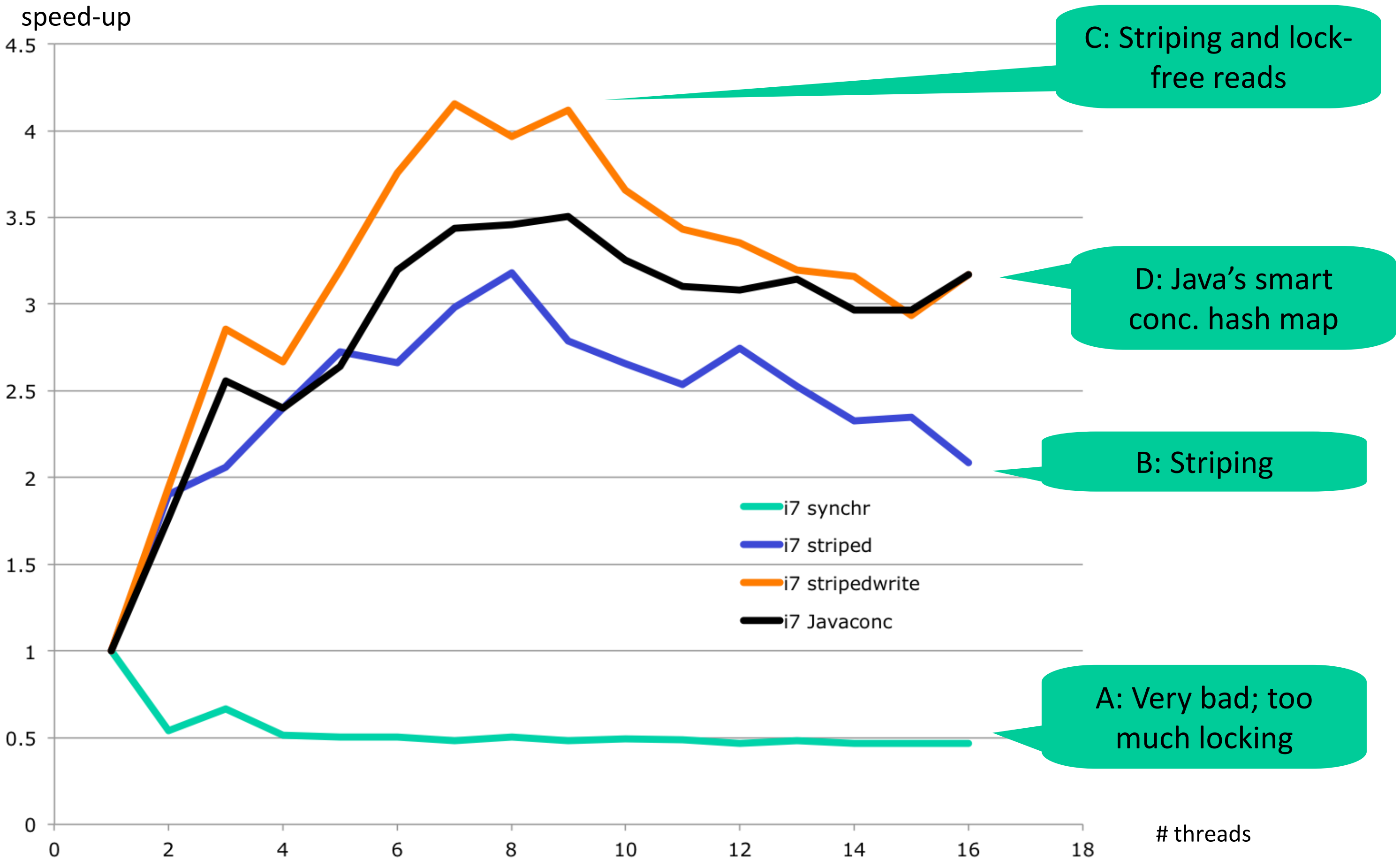
  - With high probability
  - two operations will work on different stripes
  - hence will take different locks
- Less lock contention, better scalability



Bucket idea


buckets
0
1
2
3
4
5
6
7
ItemNode
k
57001
v
Mick
next
Locking one will not lock the other
In different stripes
ItemNode
k
59014
v
Jeff
next
ItemNode
k
10406
v
Joe
next
ItemNode
k
21422
v
Sue
next

Reducing locking



Ultimate scalability

A web-shop, Facebook, …

We must give up thread safety,

but still maintain some sort of consistency

Week 13