Practical Concurrent and Parallel Programming VI

Performance and Scalability

Raúl Pardo

Assignment 1 grading is completed

- Make sure you have a grade on assignment 1
    - **<u>If not, contact us!</u>**

- At the end of the semester, we will simply check that you got a 100 in at least 5 assignments

Oral feedback

- Remember that during oral feedback sessions you will need to explain your solutions
  - So please revisit your solutions before the oral feedback session so that you remember what you did

Exam answers

- For the exam, scarce answers may be considered insufficient
- Answers should be self-contained
  - Make sure that you include the context and all your assumptions are clearly stated

Agenda

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

Previously on PCPP…

Week 5

Speedup for quicksort:	3.6 using 8 threads
	2.9 using 4 threads
Speedup for counting primes:	3.9 using 8 threads
	2.3 using 4 threads

Performance versus scalability

Performance and scalability

Performance (of software)

- **Latency:** time till first result (response time)
- **Throughput**: results per second

Scalability (one way to improve performance)
Improve throughput/latency by adding more resources

One may sacrifice performance for scalability
Maybe OK to be slower on 1 core, if faster on 2 or 4 or ...

Goetz chapter 11

What limits performance?

Suggestions?

What limits performance?

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

Input/output-bound
- Eg. reading from network
- To speed up, use more tasks (*inherent*)

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

(*Much of this lecture*)

Scalability, speed-up and loss classification
- Example: QuickSort

Quicksort

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

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
```

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
```

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
    ↑
```

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
      ↑                               ↑
```

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
      ↑                               ↑
```

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
         ↑           ↑
```

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
         ↑           ↑

               ...
```

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
```

Quicksort

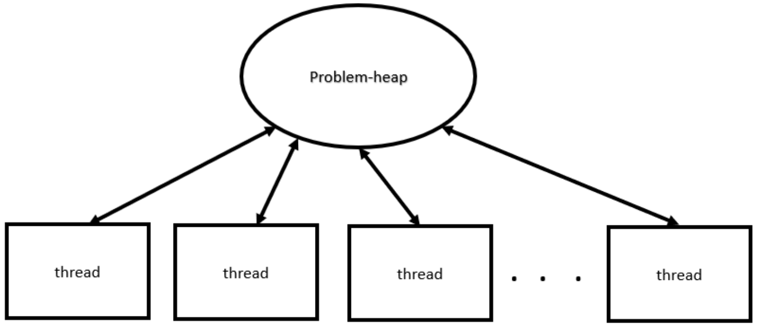
```
1  2  43  78  19  54  33  21  64  52  17  53

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
```

Quicksort

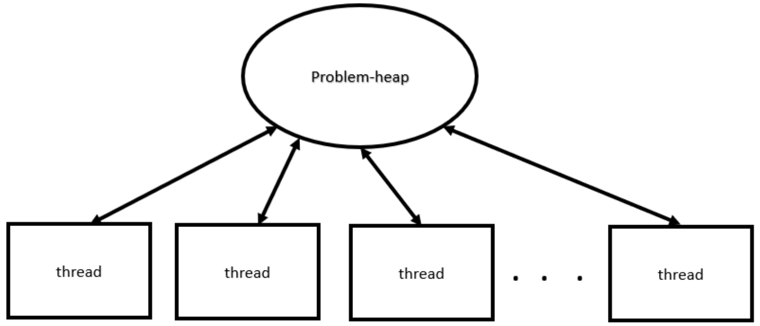
```
1  2  43  78  19  54  33  21  64  52  17  53
1  2  43  78  19  54  33  21  64  52  17  53
1  2  43  78  19  54  33  21  64  52  17  53
      ↑                               ↑
1  2  17  78  19  54  33  21  64  52  43  53
      ↑                               ↑
1  2  17  78  19  54  33  21  64  52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

Multi-threaded version of Quicksort

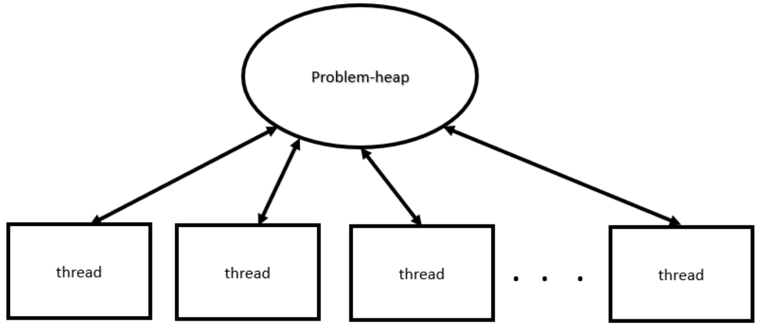


Multi-threaded version of Quicksort



```
class Problem {
      public int[] arr;
      public int low, high;
      ...
      }
}

class ProblemHeap {
      list<Problem> heap= new List<Problem>;
      ...
}
```

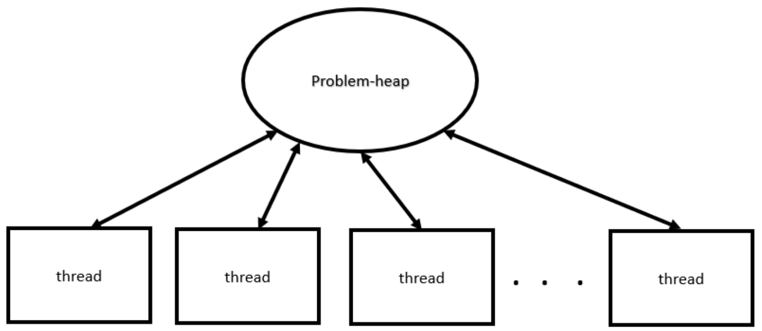
Multi-threaded version of Quicksort



```
class Problem {
      public int[] arr;
      public int low, high;
      ...
      }
}

class ProblemHeap {
      list<Problem> heap= new List<Problem>;
      ...
}
```

```
private static void qsort(Problem problem, ProblemHeap heap) {
  int[] arr= problem.arr;
  int a= problem.low;
  int b= problem.high;




}
```

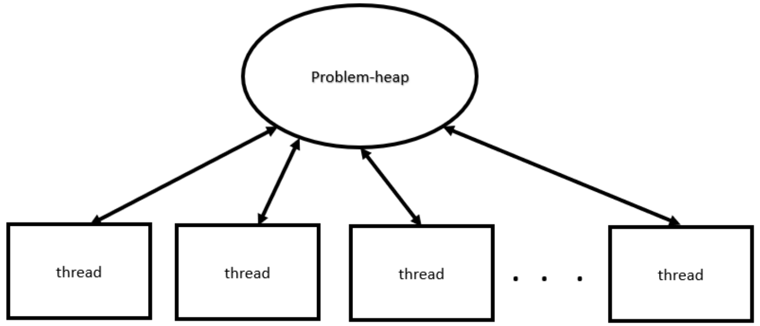
Multi-threaded version of Quicksort



```
class Problem {
        public int[] arr;
        public int low, high;
        ...
        }
}

class ProblemHeap {
        list<Problem> heap= new List<Problem>;
        ...
}
```

```
private static void qsort(Problem problem, ProblemHeap heap) {
  int[] arr= problem.arr;
  int a= problem.low;
  int b= problem.high;
  ... // solve a subtask of quicksort


}
```

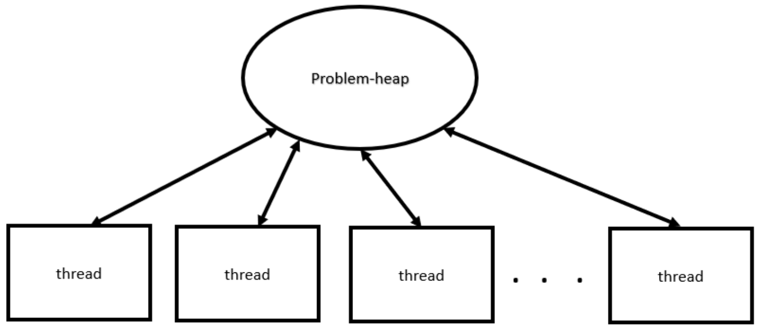
Multi-threaded version of Quicksort



```
class Problem {
      public int[] arr;
      public int low, high;
      ...
      }
}

class ProblemHeap {
      list<Problem> heap= new List<Problem>;
      ...
}
```

```
private static void qsort(Problem problem, ProblemHeap heap) {
   int[] arr= problem.arr;
   int a= problem.low;
   int b= problem.high;
   ... // solve a subtask of quicksort
   heap.add(new Problem(arr, a, j); //qsort(arr, a, j);

}
```

Multi-threaded version of Quicksort



```
class Problem {
        public int[] arr;
        public int low, high;
        ...
        }
}

class ProblemHeap {
        list<Problem> heap= new List<Problem>;
        ...
}
```

```
private static void qsort(Problem problem, ProblemHeap heap) {
    int[] arr= problem.arr;
    int a= problem.low;
    int b= problem.high;
    ... // solve a subtask of quicksort
    heap.add(new Problem(arr, a, j); //qsort(arr, a, j);
    heap.add(new Problem(arr, i, b));//qsort(arr, i, b);
 }
```

Java code for Quicksort (1)

```
public static void problemHeapStart(int threadCount, int pSize, int[] intArray) {
 ProblemHeap heap= new ProblemHeap(threadCount);
 heap.add(new Problem(intArray, 0, pSize-1));

 for (int t=0; t<threadCount; t++) {









 }
}
```

Java code for Quicksort (1)

```
public static void problemHeapStart(int threadCount, int pSize, int[] intArray) {
 ProblemHeap heap= new ProblemHeap(threadCount);
 heap.add(new Problem(intArray, 0, pSize-1));

 for (int t=0; t<threadCount; t++) {


      while (newProblem != null) {  // when newProblem == null alg stops
        qsort(newProblem, heap);
        newProblem= heap.getProblem();
      }


 }
}
```

Java code for Quicksort (1)

```
public static void problemHeapStart(int threadCount, int pSize, int[] intArray) {
 ProblemHeap heap= new ProblemHeap(threadCount);
 heap.add(new Problem(intArray, 0, pSize-1));

 for (int t=0; t<threadCount; t++) {
   threads[t]= new Thread( () -> { try {
       Problem newProblem= heap.getProblem();
       while (newProblem != null) {  // when newProblem == null alg stops
         qsort(newProblem, heap);
         newProblem= heap.getProblem();
       }

   });
 }
}
```

Java code for Quicksort (1)

```
public static void problemHeapStart(int threadCount, int pSize, int[] intArray) {
 ProblemHeap heap= new ProblemHeap(threadCount);
 heap.add(new Problem(intArray, 0, pSize-1));

 for (int t=0; t<threadCount; t++) {
   threads[t]= new Thread( () -> { try {
       Problem newProblem= heap.getProblem();
       while (newProblem != null) {  // when newProblem == null alg stops
         qsort(newProblem, heap);
         newProblem= heap.getProblem();
       }
       }catch (InterruptedException exn) { }  //needed because getProblem may wait
   });
 }
}
```

Java code for Quicksort (2)

“Measure, don’t guess”

Goetz p. 224

We use Mark8Setup to measure runtime

```
Benchmark.Mark8Setup("Problem heap quicksort",
        String.format("%2d", threadCount),
        new Benchmarkable() {








        }
    );
```

Java code for Quicksort (2)

“Measure, don’t guess”

Goetz p. 224

We use Mark8Setup to measure runtime

```
 Benchmark.Mark8Setup("Problem heap quicksort",
          String.format("%2d", threadCount),
          new Benchmarkable() {
            public void setup() {
               shuffle(intArray);
               problemHeapStart(threadCount, pSize, intArray);
            }



          }
    );
```

Java code for Quicksort (2)

“Measure, don’t guess”

Goetz p. 224

We use Mark8Setup to measure runtime

```
Benchmark.Mark8Setup("Problem heap quicksort",
         String.format("%2d", threadCount),
         new Benchmarkable() {
           public void setup() {
              shuffle(intArray);
              problemHeapStart(threadCount, pSize, intArray);
           }
           public double applyAsDouble(int i) {
              problemHeapFinish(threadCount, intArray); return 0.0;
           }
         }
    );
```

Java code for Quicksort (2)

“Measure, don’t guess”
Goetz p. 224

We use Mark8Setup to measure runtime

```
Benchmark.Mark8Setup("Problem heap quicksort",
         String.format("%2d", threadCount),
         new Benchmarkable() {
           public void setup() {
              shuffle(intArray);
              problemHeapStart(threadCount, pSize, intArray);
           }
           public double applyAsDouble(int i) {
              problemHeapFinish(threadCount, intArray); return 0.0;
           }
         }
   );
```

Code in `ProblemHeapSortingBenchmarkable.java`

Java Executors (1)

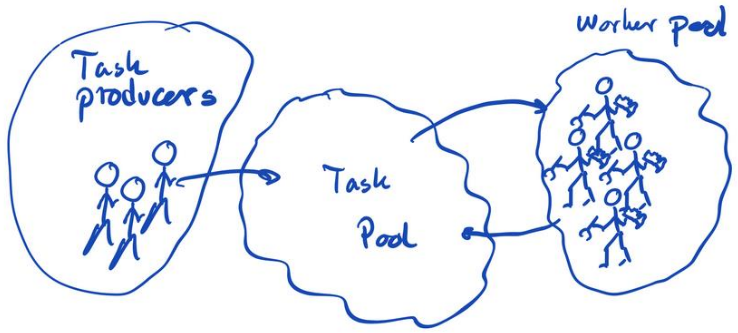
Motivation

- Threads are expensive to start - executors reuse threads
- Problem heap - breaking a problem down to smaller problems (tasks)

Java Executors (1)

Motivation

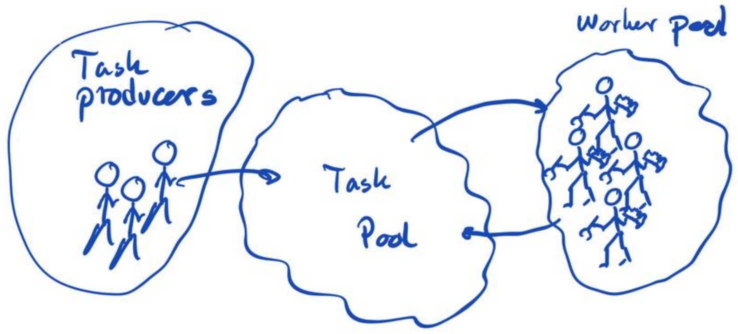
- Threads are expensive to start - executors reuse threads
- Problem heap - breaking a problem down to smaller problems (tasks)



Java Executors (1)

Motivation

- Threads are expensive to start - executors reuse threads
- Problem heap - breaking a problem down to smaller problems (tasks)

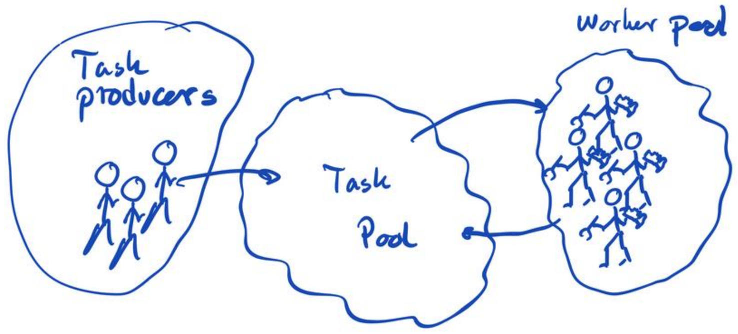


Task producers, and *Workers* are threads

Java Executors (1)

Motivation

- Threads are expensive to start - executors reuse threads
- Problem heap - breaking a problem down to smaller problems (tasks)

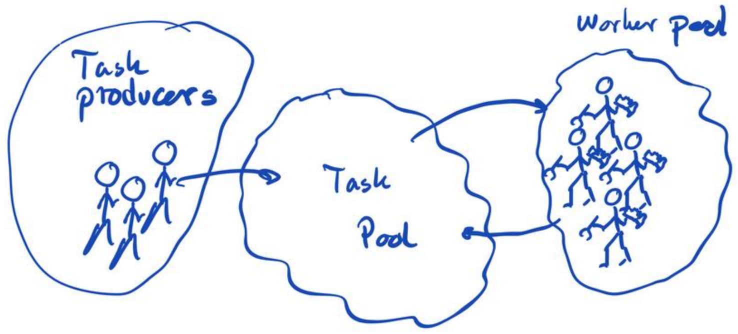


Task producers, and *Workers* are threads
Workers may themselves produce new tasks

Java Executors (1)

Motivation

- Threads are expensive to start - executors reuse threads
- Problem heap - breaking a problem down to smaller problems (tasks)



Task producers, and *Workers* are threads
Workers may themselves produce new tasks

The Task pool and Worker pool together is called an *Executor service*

Java Executors (2)

Goetz 6.2

```
new Thread(runnable1).start();
...
new Thread(runnable2).start();
...
new Thread(runnable3).start();
```

Threads are expensive !

```
ExecutorService pool;

pool.execute(runnable1);
...
pool.execute(runnable2);
...
pool.execute(runnable2);
```

Reuse of threads

https://howtodoinjava.com/java/multi-threading/java-fixed-size-thread-pool-executor-example/

Java Executors (3) - Tasks

- **Tasks** are a central concept for executors

- When designing a program using executors, first think about the tasks to be executed
  - Like for threads, tasks can be conveniently defined in their own class

- Ideally, tasks should be independent

Java Executors (3) - Tasks

- **Tasks** are a central concept for executors

- When designing a program using executors, first think about the tasks to be executed
  - Like for threads, tasks can be conveniently defined in their own class

- Ideally, tasks should be independent

Why?

Quicksort Task class

Quicksort Task class

```
class QuicksortTask implements Runnable {


















}
```

Quicksort Task class

```
class QuicksortTask implements Runnable {
 private Problem p;














}
```

Quicksort Task class

```
class QuicksortTask implements Runnable {
 private Problem p;
 private ExecutorService pool;










}
```

Quicksort Task class

```
class QuicksortTask implements Runnable {
 private Problem p;
 private ExecutorService pool;
 private static int threshold;


















}
```

Quicksort Task class

```
class QuicksortTask implements Runnable {
 private Problem p;
 private ExecutorService pool;
 private static int threshold;
 ...









}
```

Quicksort Task class

```
class QuicksortTask implements Runnable {
 private Problem p;
 private ExecutorService pool;
 private static int threshold;
 ...

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









}
```

Quicksort Task class

```
class QuicksortTask implements Runnable {
 private Problem p;
 private ExecutorService pool;
 private static int threshold;
 ...

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

 public static void qsort(ExecutorService pool, ... ) {










 }
}
```

Quicksort Task class

```
class QuicksortTask implements Runnable {
 private Problem p;
 private ExecutorService pool;
 private static int threshold;
 ...

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

 public static void qsort(ExecutorService pool, ... ) {
  if (a < b) {










 }
}
```

Quicksort Task class

```
class QuicksortTask implements Runnable {
 private Problem p;
 private ExecutorService pool;
 private static int threshold;
 ...

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

 public static void qsort(ExecutorService pool, ... ) {
  if (a < b) {

   ... // quicksort on Problem p





  }
  ... // subarray already ordered
 }
}
```

Quicksort Task class

```
class QuicksortTask implements Runnable {
 private Problem p;
 private ExecutorService pool;
 private static int threshold;
 ...

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

 public static void qsort(ExecutorService pool, ... ) {
  if (a < b) {

   ... // quicksort on Problem p

   if ((j-a)>=threshold) pool.execute(new QuicksortTask(new Problem(...), pool, ...));




  }
  ... // subarray already ordered
 }
}
```

Quicksort Task class

```
class QuicksortTask implements Runnable {
 private Problem p;
 private ExecutorService pool;
 private static int threshold;
 ...

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

 public static void qsort(ExecutorService pool, ... ) {
  if (a < b) {

    ... // quicksort on Problem p

    if ((j-a)>=threshold) pool.execute(new QuicksortTask(new Problem(...), pool, ...));
    else qsort( new Problem(...), ...);



  }
  ... // subarray already ordered
 }
}
```

Quicksort Task class

```
class QuicksortTask implements Runnable {
 private Problem p;
 private ExecutorService pool;
 private static int threshold;
 ...

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

 public static void qsort(ExecutorService pool, ... ) {
  if (a < b) {

    ... // quicksort on Problem p

    if ((j-a)>=threshold) pool.execute(new QuicksortTask(new Problem(...), pool, ...));
    else qsort( new Problem(...), ...);

    if ((b-i)>= threshold) pool.execute(new QuicksortTask(new Problem(...), pool, ...));
    else qsort(new Problem(...), ...);
   }
   ... // subarray already ordered
  }
}
```

Quicksort Task class

```
class QuicksortTask implements Runnable {
 private Problem p;
 private ExecutorService pool;
 private static int threshold;
 ...

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

 public static void qsort(ExecutorService pool, ... ) {
  if (a < b) {

   ... // quicksort on Problem p

   if ((j-a)>=threshold) pool.execute(new QuicksortTask(new Problem(...), pool, ...));
   else qsort( new Problem(...), ...);

   if ((b-i)>= threshold) pool.execute(new QuicksortTask(new Problem(...), pool, ...));
   else qsort(new Problem(...), ...);
  }
  ... // subarray already ordered
 }
}
```

Are Quicksort tasks independent of each other?

Quicksort Task class

```
class QuicksortTask implements Runnable {
 private Problem p;
 private ExecutorService pool;
 private static int threshold;
 ...

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

 public static void qsort(ExecutorService pool, ... ) {
  if (a < b) {

   ... // quicksort on Problem p

   if ((j-a)>=threshold) pool.execute(new QuicksortTask(new Problem(...), pool, ...));
   else qsort( new Problem(...), ...);

   if ((b-i)>= threshold) pool.execute(new QuicksortTask(new Problem(...), pool, ...));
   else qsort(new Problem(...), ...);
  }
  ... // subarray already ordered
 }
}
```

Are Quicksort tasks independent of each other?

Code in `QuicksortTask.java`

Quicksort Executor class

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

```
class QuicksortExecutor {
     private ExecutorService pool;
     ...
    private static void setUpQS(int threadCount, int[] intArray, int threshold,...) {
     pool = Executors.newFixedThreadPool(threadCount);
     pool.execute( new QuicksortTask( new Problem(intArray, 0, intArray.length-1),
                          pool, threshold, ...) );
    }
}
```

Quicksort Executor class

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

```
class QuicksortExecutor {
     private ExecutorService pool;
     ...
    private static void setUpQS(int threadCount, int[] intArray, int threshold,...) {
     pool = Executors.newFixedThreadPool(threadCount);
     pool.execute( new QuicksortTask( new Problem(intArray, 0, intArray.length-1),
                          pool, threshold, ...) );
    }
}
```

- There are several type of thread pools:
  - newFixedThreadPool
  - newCachedThreadPool (useful for testing)
  - newSingleThreadExecutor
  - …

Work stealing queues

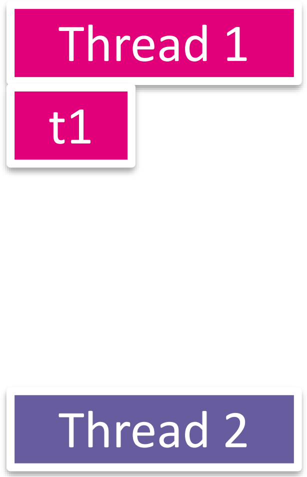
- Instead of having a single task pool, each worker has its own task pool
    - This tasks pools are implemented as *queues*
- Workers add tasks to their own queues
    - Minimizes thread contention in a single thread resource
- If worker can work but its queue is empty, it *steals* a task from another worker task pool
    - From the end of the queue
    - To further minimize contention

Work stealing queues

- Instead of having a single task pool, each worker has its own task pool
    - This tasks pools are implemented as *queues*
- Workers add tasks to their own queues
    - Minimizes thread contention in a single thread resource
- If worker can work but its queue is empty, it *steals* a task from another worker task pool
    - From the end of the queue
    - To further minimize contention

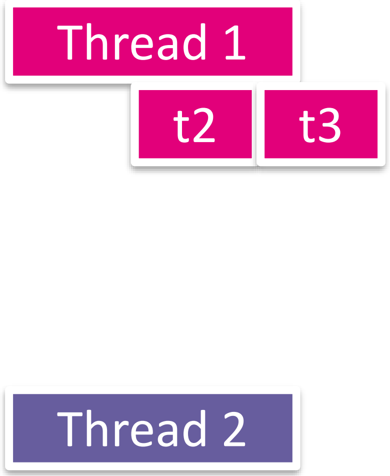
Work stealing queues

- Instead of having a single task pool, each worker has its own task pool
  - This tasks pools are implemented as *queues*
- Workers add tasks to their own queues
  - Minimizes thread contention in a single thread resource
- If worker can work but its queue is empty, it *steals* a task from another worker task pool
  - From the end of the queue
  - To further minimize contention



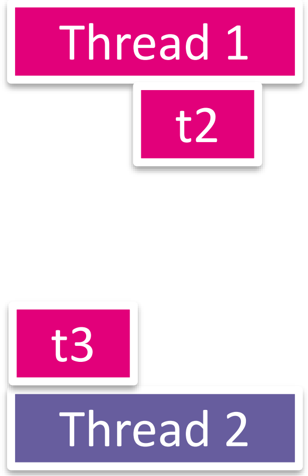
Work stealing queues

- Instead of having a single task pool, each worker has its own task pool
  - This tasks pools are implemented as *queues*
- Workers add tasks to their own queues
  - Minimizes thread contention in a single thread resource
- If worker can work but its queue is empty, it *steals* a task from another worker task pool
  - From the end of the queue
  - To further minimize contention



Work stealing queues

- Instead of having a single task pool, each worker has its own task pool
  - This tasks pools are implemented as *queues*
- Workers add tasks to their own queues
  - Minimizes thread contention in a single thread resource
- If worker can work but its queue is empty, it *steals* a task from another worker task pool
  - From the end of the queue
  - To further minimize contention



Work stealing queues

- Instead of having a single task pool, each worker has its own task pool
  - This tasks pools are implemented as *queues*
- Workers add tasks to their own queues
  - Minimizes thread contention in a single thread resource
- If worker can work but its queue is empty, it *steals* a task from another worker task pool
  - From the end of the queue
  - To further minimize contention

Work stealing queues

- Instead of having a single task pool, each worker has its own task pool
  - This tasks pools are implemented as *queues*
- Workers add tasks to their own queues
  - Minimizes thread contention in a single thread resource
- If worker can work but its queue is empty, it *steals* a task from another worker task pool
  - From the end of the queue
  - To further minimize contention

Thread 1

Thread 2

In what type of algorithms may work stealing queues be useful?

Performance of Executor Quicksort

```
Executor quicksort  1     98003405.0 ns
Executor quicksort  2     53568593.9 ns
Executor quicksort  4     36397241.3 ns
Executor quicksort  8     21714103.7 ns
Executor quicksort 16     22237307.4 ns
Executor quicksort 32     22510681.9 ns
```

Speedup = 4.5

A bit better speed-up than using native Threads (slide 3)

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

Two parts can be sorted independently

What limits scalability?

What limits scalability?

Example: growing a crop

- 4 months growth + 1 month harvest if done by 1 person
- Growth (sequential) cannot be speeded up
- Using 30 people to harvest, takes 1/30 month = 1 day
- Speed-up using many harvesters: 5/(4+1/30) = 1.24 times faster

What limits scalability?

Example: growing a crop

- 4 months growth + 1 month harvest if done by 1 person
- Growth (sequential) cannot be speeded up
- Using 30 people to harvest, takes 1/30 month = 1 day
- Speed-up using many harvesters: 5/(4+1/30) = 1.24 times faster

Amdahl's law (Goetz 11.2)

F = sequential fraction of problem = 4/5 = 0.8

N = number of threads (people) = 30

$$speed\ up \leq \frac{1}{F + \frac{(1-F)}{N}} = \frac{1}{0.8 + 0.2/30} = 1.24$$

Other types of "loss" of scalability

- Starvation loss
  - Minimize the time that the problem heap is empty

Other types of "loss" of scalability

- Starvation loss
  - Minimize the time that the problem heap is empty
- Separation loss (best threshold)
  - Find a good threshold to distribute workload evenly

Other types of "loss" of scalability

- Starvation loss
  - Minimize the time that the problem heap 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

Other types of "loss" of scalability

- Starvation loss
  - Minimize the time that the problem heap 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 threads as soon as the problem is solved

Other types of "loss" of scalability

- Starvation loss
  - Minimize the time that the problem heap 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 threads 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

Executors and Future

- Example: Dot Product

Termination in Executors

```
class QuicksortTask implements Runnable {

  public void run() { qsort(p, ...); }
}
```

Termination in Executors

```
class QuicksortTask implements Runnable {

  public void run() { qsort(p, ...); }
}
```

```
class QuicksortTask implements Runnable {

  public void run() { qsort(p, ...); }
}
```

Termination in Executors

```
class QuicksortTask implements Runnable {

  public void run() { qsort(p, ...); }
}
```

```
class QuicksortTask implements Runnable {

  public void run() { qsort(p, ...); }
}
```

. . .

```
class QuicksortTask implements Runnable {

  public void run() { qsort(p, ...); }
}
```

Termination in Executors

```
class QuicksortTask implements Runnable {

  public void run() { qsort(p, ...); }
}
```

```
class QuicksortTask implements Runnable {

  public void run() { qsort(p, ...); }
}
```

. . .

```
class QuicksortTask implements Runnable {

  public void run() { qsort(p, ...); }
}
```

```
public class QuicksortExecutor {
  //main (thread)
  setUpQS( … ) // Initializes pool, executes initial task
  finishQS( … ) //wait for all tasks to be completed
}
```

Termination in Executors

```
class QuicksortTask implements Runnable {

  public void run() { qsort(p, ...); }
}
```

```
class QuicksortTask implements Runnable {

  public void run() { qsort(p, ...); }
}
```

. . .

```
class QuicksortTask implements Runnable {

  public void run() { qsort(p, ...); }
}
```

```
public class QuicksortExecutor {
  //main (thread)
  setUpQS( … ) // Initializes pool, executes initial task
  finishQS( … ) //wait for all tasks to be completed
}
```

How do we known when all tasks are finished?

Termination in Executors

```
class QuicksortTask implements Runnable {

  public void run() { qsort(p, ...); }
}
```

```
class QuicksortTask implements Runnable {

  public void run() { qsort(p, ...); }
}
```

. . .

```
class QuicksortTask implements Runnable {

  public void run() { qsort(p, ...); }
}
```

```
public class QuicksortExecutor {
  //main (thread)
  setUpQS( … ) // Initializes pool, executes initial task
  finishQS( … ) //wait for all tasks to be completed
}
```

How do we known when all tasks are finished?

Need for a signal from the workers to the main thread!

Shut down

The ExecutorService has methods to shut down the pool.

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

pool.shutdown();
```

Shut down

The ExecutorService has methods to shut down the pool.

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

pool.shutdown();
```

The challenge is *when to shut down*

Shut down

The ExecutorService has methods to shut down the pool.

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

pool.shutdown();
```

The challenge is *when to shut down*

In the Quicksort example, a counter is used to track the number of pending tasks. When there are no pending tasks we can shut down.

Solution 1: shared counter + barrier

```
// main thread
class public class QuicksortExecutor {
     ...
     final AtomicInteger count = new AtomicInteger(0);
     final CyclicBarrier done = new CyclicBarrier(2);

     private static void setUpQS(int threadCount, int[] intArray, int threshold,
                                CyclicBarrier done, AtomicInteger count) {
          pool = Executors.newFixedThreadPool(threadCount);
          count.set(1); // Initial task count
          pool.execute( new QuicksortTask(…, done, count) );
     }
     ...

}
```

Counts the number of pending tasks

It will be used to signal when to shutdown

Using the counter + barrier

```
public static void qsort(.., CyclicBarrier done, AtomicInteger count) {
if (a < b) {
  ...

  if ((j-a)>= threshold) count.incrementAndGet();
  if ((b-i)>= threshold) count.incrementAndGet();


  if ((j-a)>= threshold) {
    pool.execute(new QuicksortTask(new Problem(arr, a, j), pool, c) );
  } else qsort(...) // sequentially
  if ((b-i)>= threshold) {
    pool.execute(new solveProblem(new Problem(arr, i, b), pool, c) );
  } else qsort(...) // sequentially

  ...

  if (count.decrementAndGet() == 0) { done.await(); pool.shutdown(); }
}
```

Using the counter + barrier

```
public static void qsort(.., CyclicBarrier done, AtomicInteger count) {
if (a < b) {
   ...

   if ((j-a)>= threshold) count.incrementAndGet();
   if ((b-i)>= threshold) count.incrementAndGet();


   if ((j-a)>= threshold) {
      pool.execute(new QuicksortTask(new Problem(arr, a, j), pool, c) );
   } else qsort(...) // sequentially
   if ((b-i)>= threshold) {
      pool.execute(new solveProblem(new Problem(arr, i, b), pool, c) );
   } else qsort(...) // sequentially

   ...

   if (count.decrementAndGet() == 0) { done.await(); pool.shutdown(); }
}
```

Why not incrementing in the branch creating the task?

Termination in Executors

```
class QuicksortTask implements Runnable {

  public void run() { qsort(p, ...); }
}
```

```
class QuicksortTask implements Runnable {

  public void run() { qsort(p, ...); }
}
```

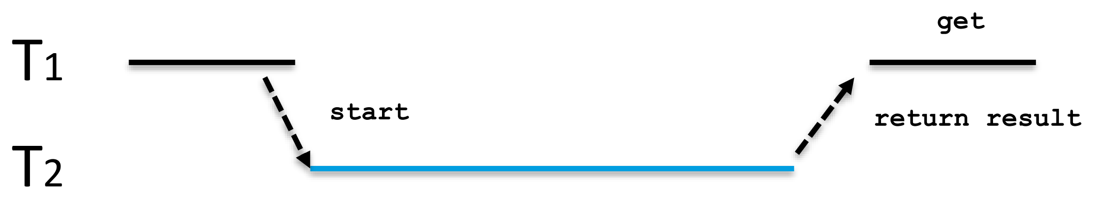
. . .

```
class QuicksortTask implements Runnable {

  public void run() { qsort(p, ...); }
}
```

```
public class QuicksortExecutor {
  //main (thread)
  setUpQS( … ) // Initializes pool, executes initial task
  finishQS( … ) //wait for all tasks to be completed
}
```

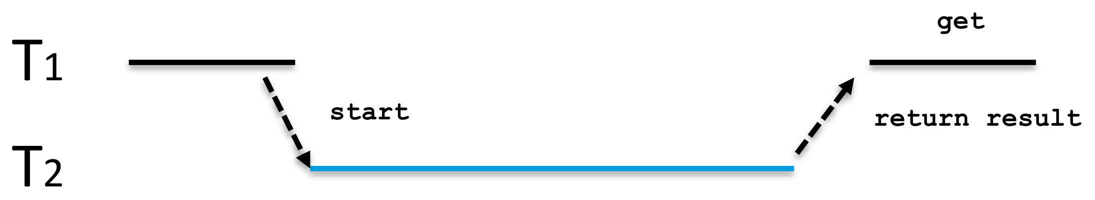
```
private static void finishQS(CyclicBarrier done) {
   try { done.await(); }
   catch (InterruptedException | BrokenBarrierException e) { ... ); }
}
```

Solution 2: Future



get() is a blocking call, so *watch out for deadlocks!*

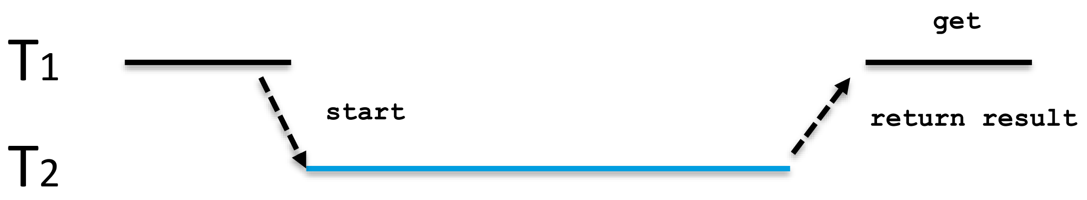
Solution 2: Future



```
T1:
  Future<Integer> future = T2.calculate(  );  // start
      ...
  future.get();
```

get() is a blocking call, so ***watch out for deadlocks!***

Solution 2: Future



```
T2:
  public Future<Integer> calculate(Integer input) {
    return executor.submit(() -> {
      ... // compute result
      return result;
    });
  }
}

T1:
  Future<Integer> future = T2.calculate(  );  // start
     ...
  future.get();
```

get() is a blocking call, so *watch out for deadlocks!*

Complete example

https://www.baeldung.com/java-future

```
private ExecutorService executor
      = Executors.newSingleThreadExecutor();

   public Future<Integer> calculate(Integer input) {
       return executor.submit(  //Callable
          () -> {...      return ... ;   }
      );
   }
```

Code in `futureExample.java`

Runnable vs. Callable

Both are used to specify the code of a thread.

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()

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()

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

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()

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

Futures are an example of message passing (coming weeks)

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 example, Runnables may use shared data (e.g., to deliver a result)

Futures are an example of message passing (coming weeks)

dotProduct – Task class

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

dotProduct – Task class

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

We use futures to multiple each vector position

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 ...









...
```

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));









...
```

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);






...
```

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



...
```

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
...
```

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`

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

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

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, …

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 synchronized/concurrent implementations

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

Example: synchronized ArrayList

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











  public syncCollectionExample() {






  }
}
```

Goetz p. 80

Example: synchronized ArrayList

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









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



  }
}
```

Goetz p. 80

Example: synchronized ArrayList

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









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

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

Goetz p. 80

Example: synchronized ArrayList

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

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

Goetz p. 80

Example: synchronized ArrayList

```
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);
    ...
  }
}
```

Goetz p. 80

Thread-safety (from week 3)

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

p only accesses thread-safe <u>classes</u>

$\not\Rightarrow$

p is a thread-safe <u>program</u>

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?

What if the data structure is huge?

and used by many threads?

What if the data structure is huge?

and used by many threads?

for example:

a bank
Facebook updates
…

What if the data structure is huge?

and used by many threads?

for example:

a bank
Facebook updates
…

Would not work if everything I "synchronized"

What if the data structure is huge?

and used by many threads?

for example:

a bank
Facebook updates
…

Would not work if everything I "synchronized"

What can we do?

What if the data structure is huge?

and used by many threads?

for example:

a bank
Facebook updates
…

Would not work if everything I "synchronized"

What can we do?

Reduce locking !!

Example: A huge HashMap

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

Example: A huge HashMap

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

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

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) { ... }
  ...
}
```

Example: A huge HashMap

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

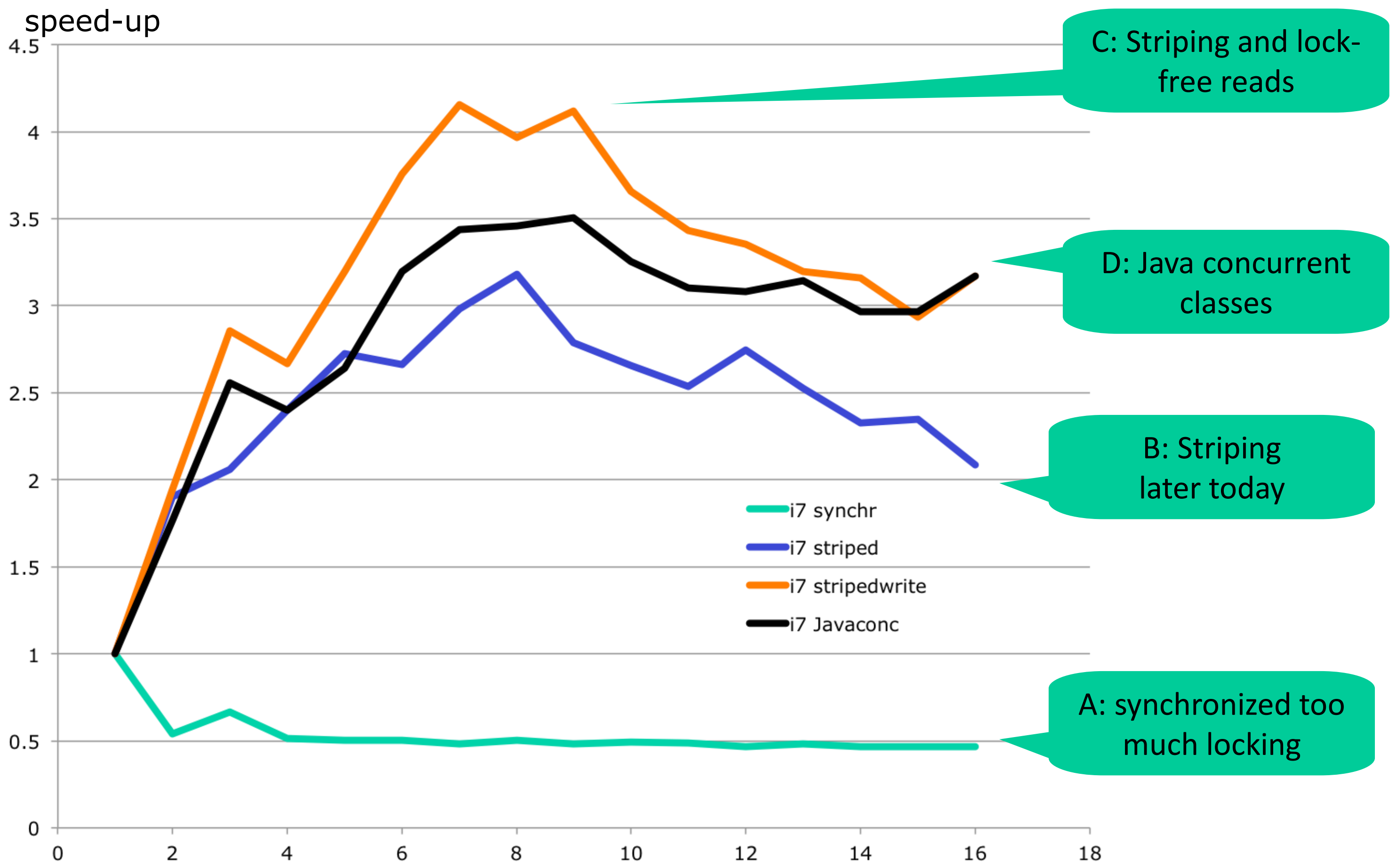
Key value pairs: <k1, v1>, <k2, v2>, …

```
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?

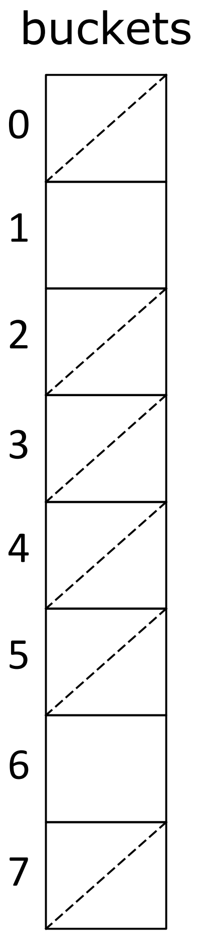
Scaling a HashMap

133

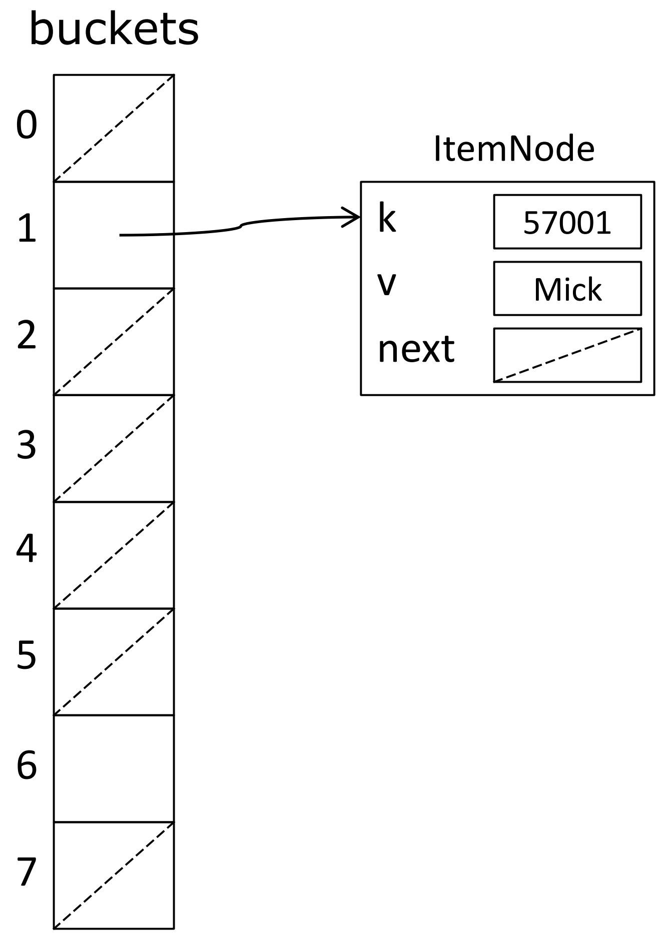


# threads

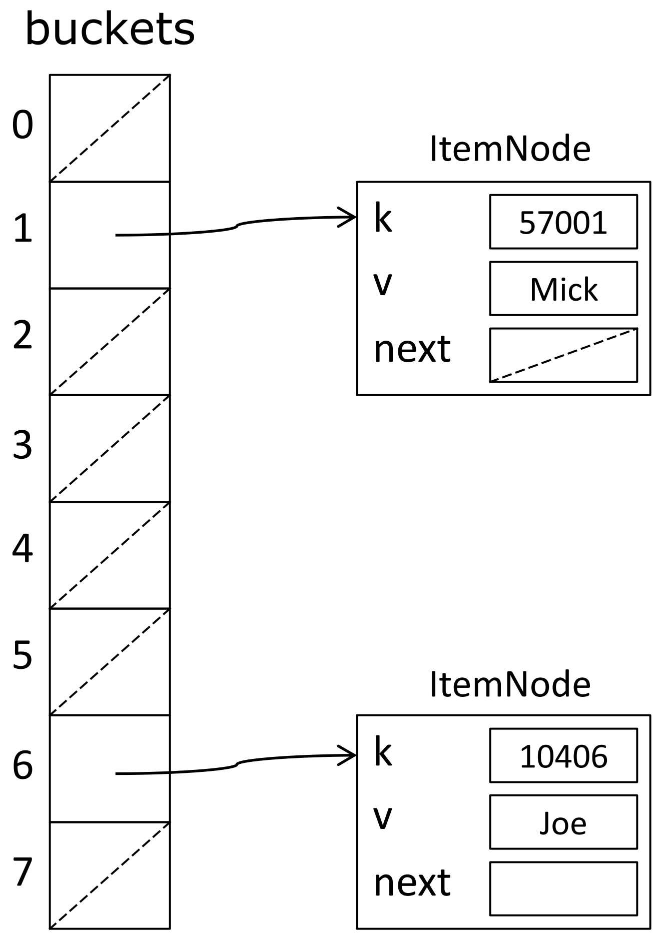
HashMap implementation



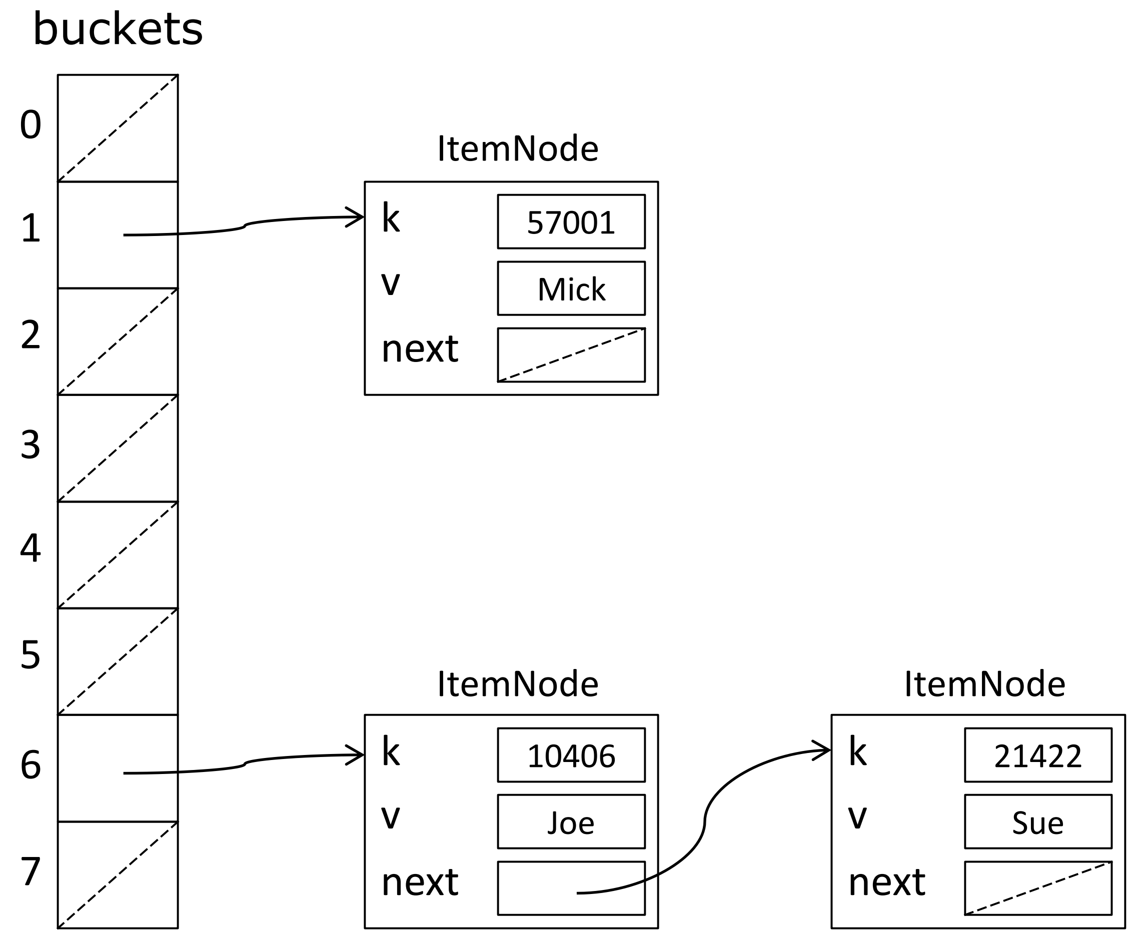
HashMap implementation



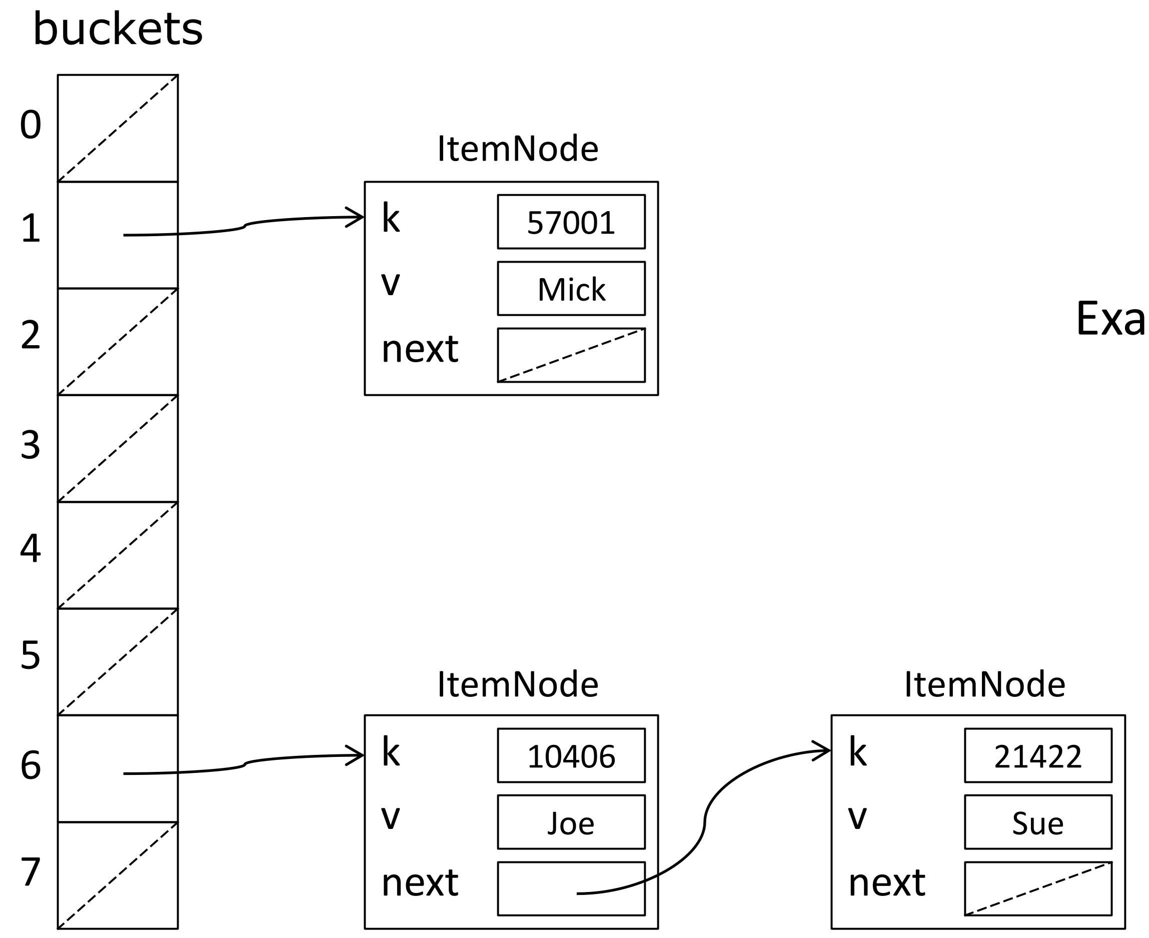
HashMap implementation



HashMap implementation

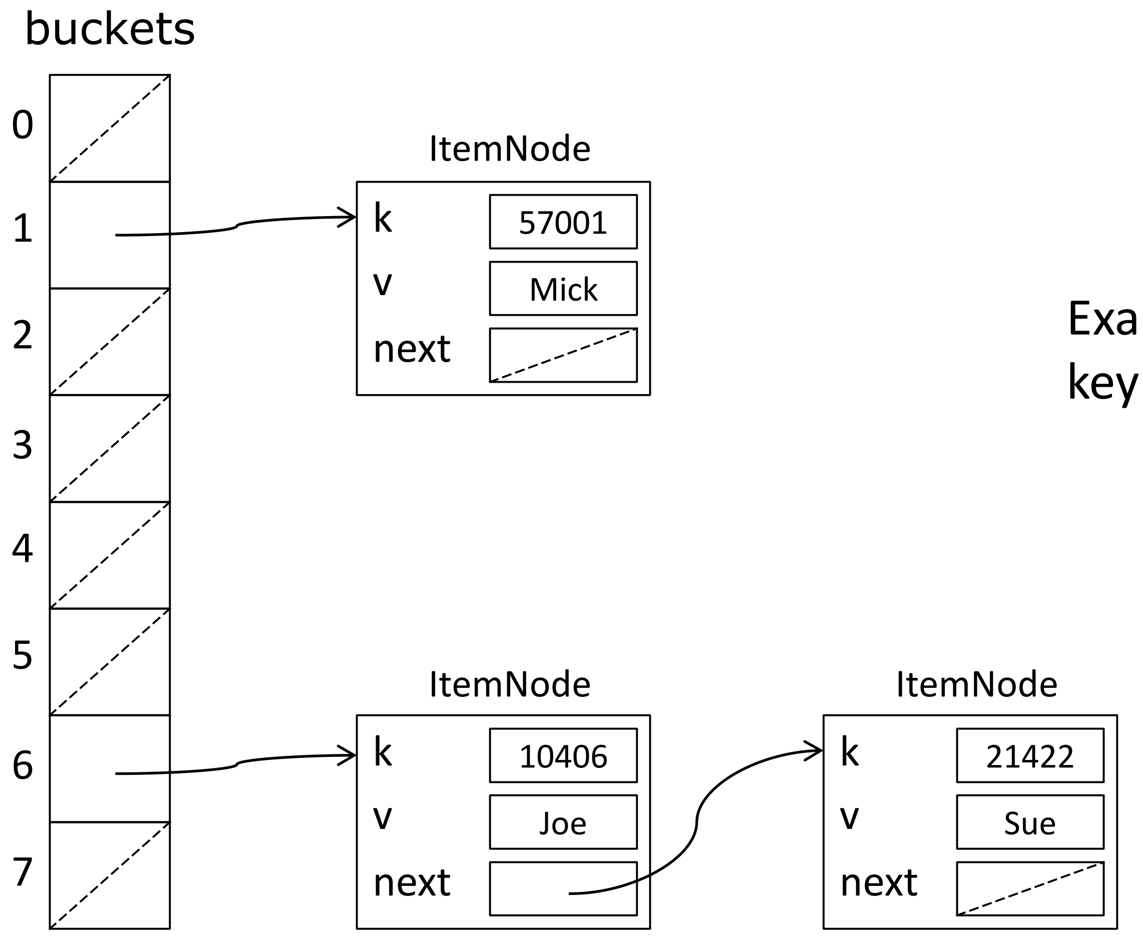


HashMap implementation



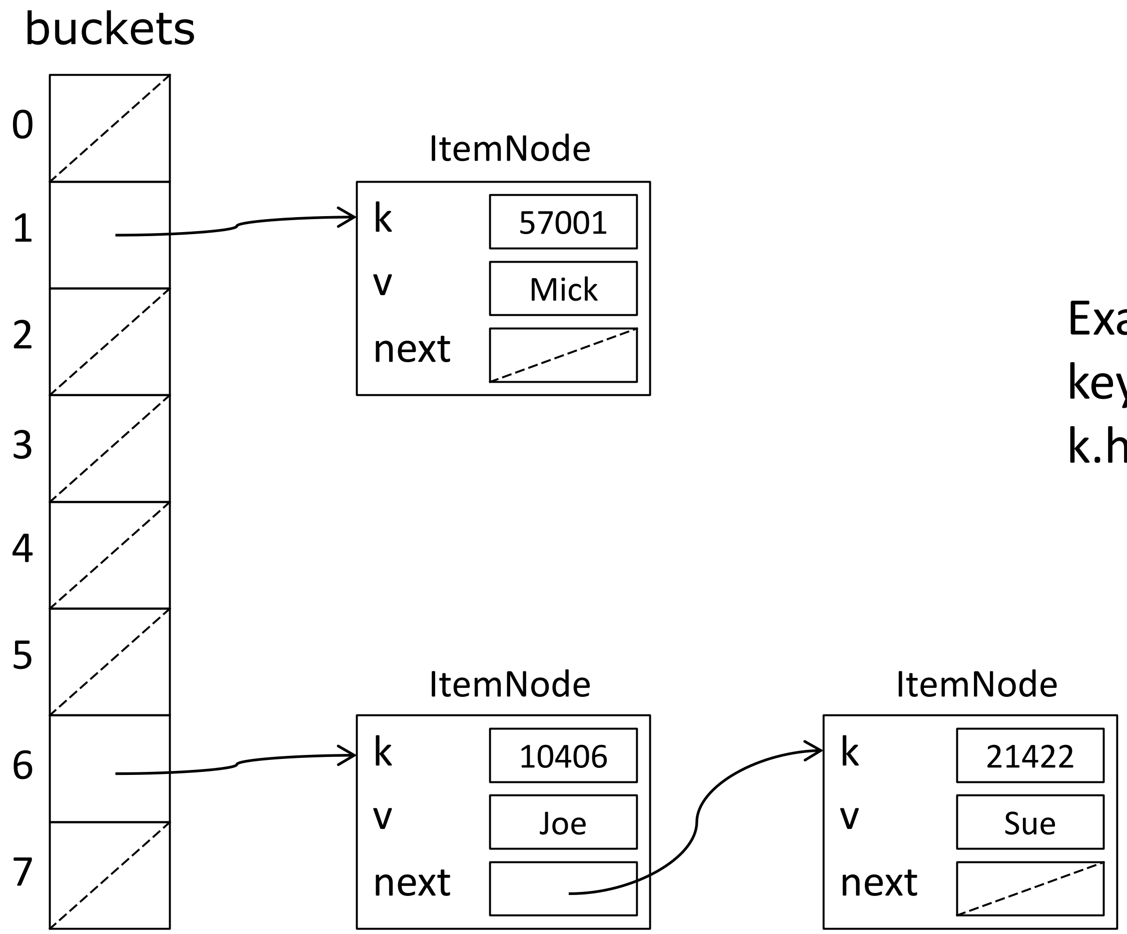
Example **get(10406)**

HashMap implementation



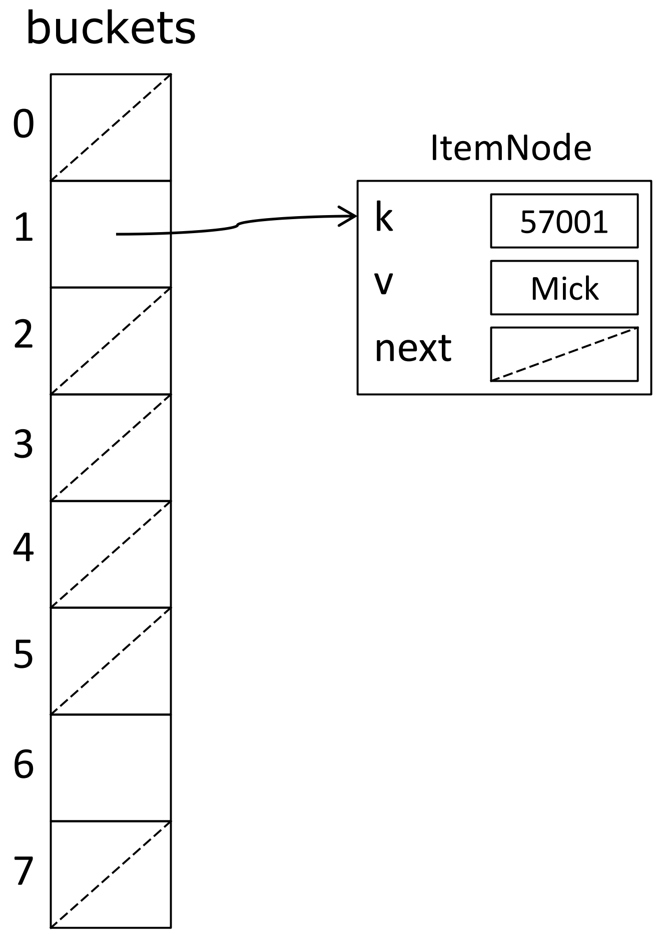
Example **get(10406)**
key k is 10406

HashMap implementation

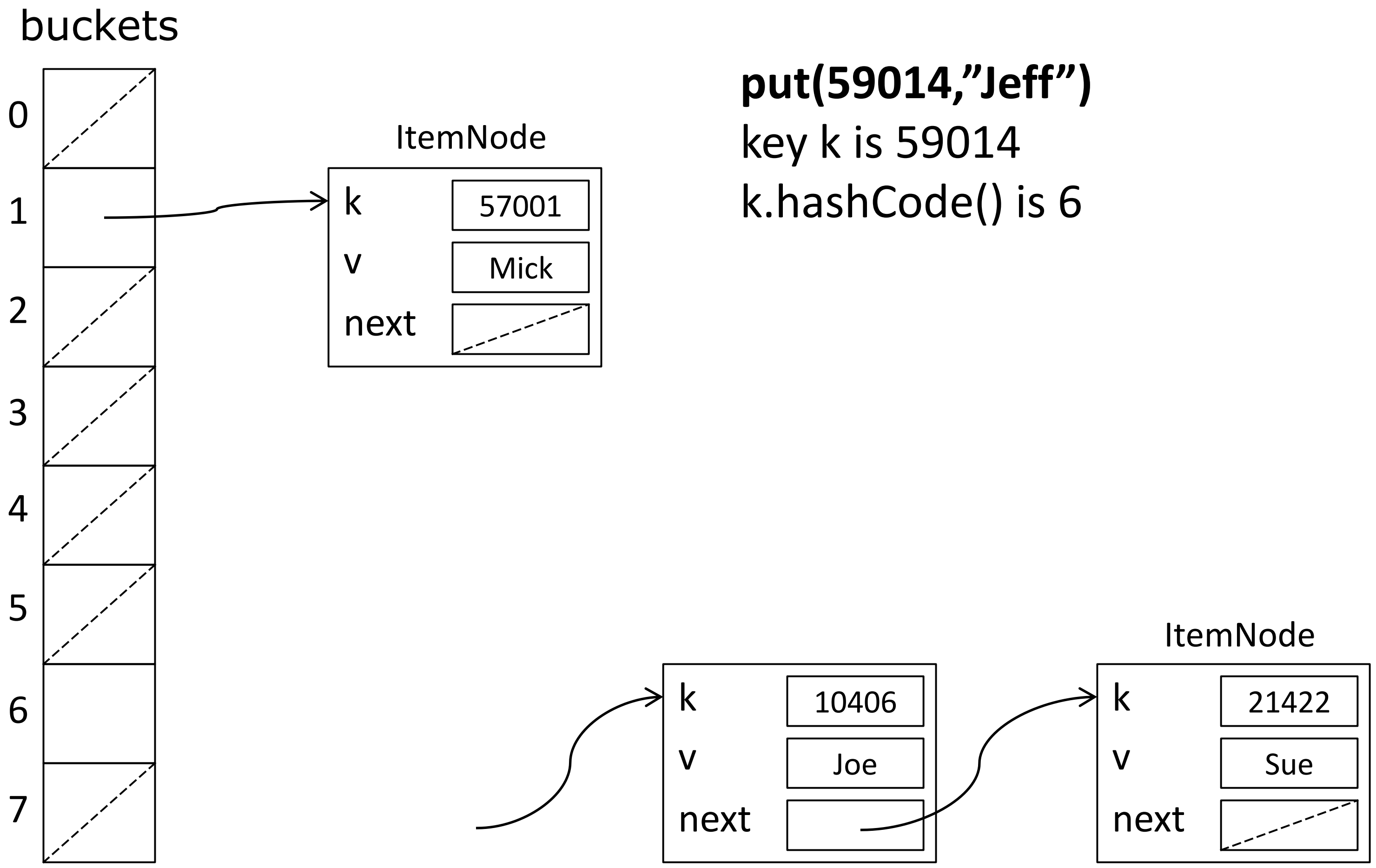


Example **get(10406)**
key k is 10406
k.hashCode() is 6

HaspMap put

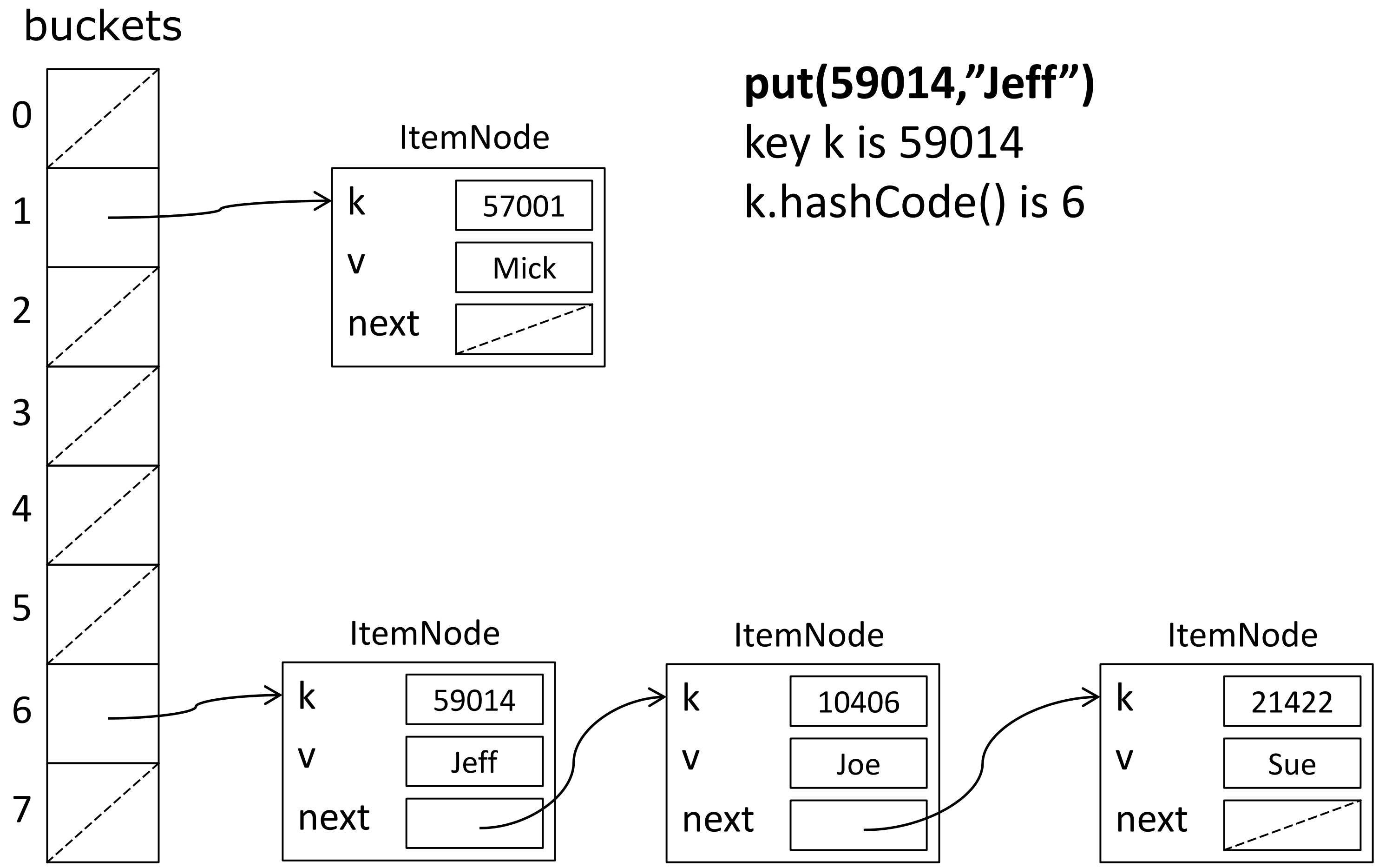


HaspMap put



put(59014,”Jeff”)
key k is 59014
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> {


  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) { ... }
}
```

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)

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

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

Improving scalability – Lock striping

- Guarding the table with a single lock works

–... but does not scale well (actually **very** badly)

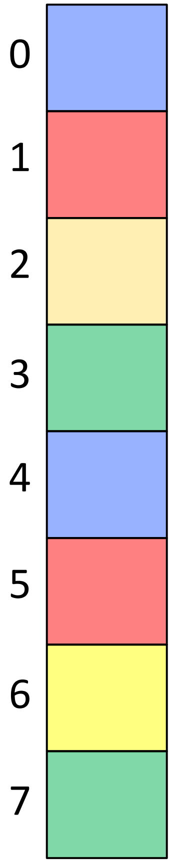
- Idea: Each bucket could have its own lock
- In practice

–use fewer, to illustrate we use 4, locks

–guard every 4th bucket with the same lock

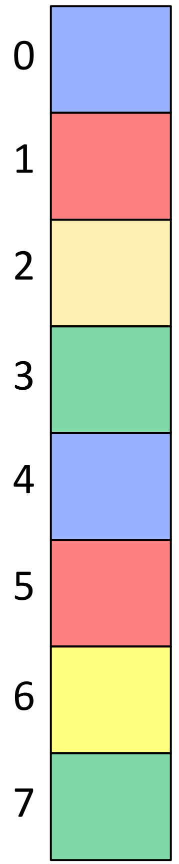
Improving scalability – Lock striping

- 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



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)



Improving scalability – Lock striping

- Guarding the table with a single lock works

–... but does not scale well (actually **very** badly)

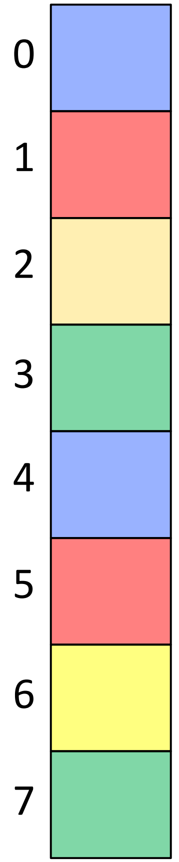
- Idea: Each bucket could have its own lock
- In practice

–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)

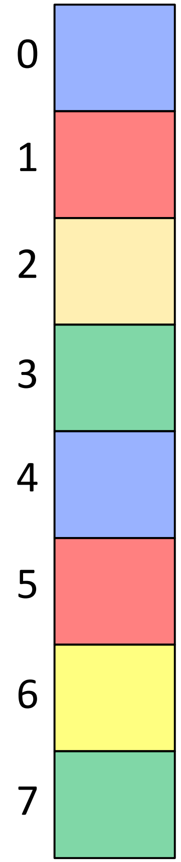
–locks[1] guards bucket 1, 5, 9, ... (stripe 1) et



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



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)

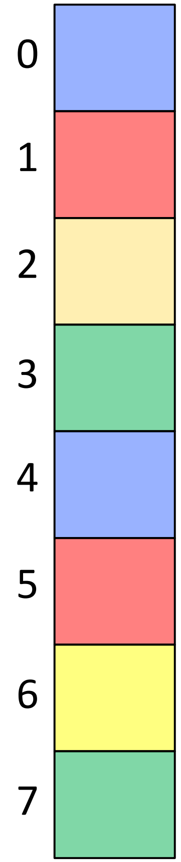
–locks[1] guards bucket 1, 5, 9, ... (stripe 1) et

–With high probability

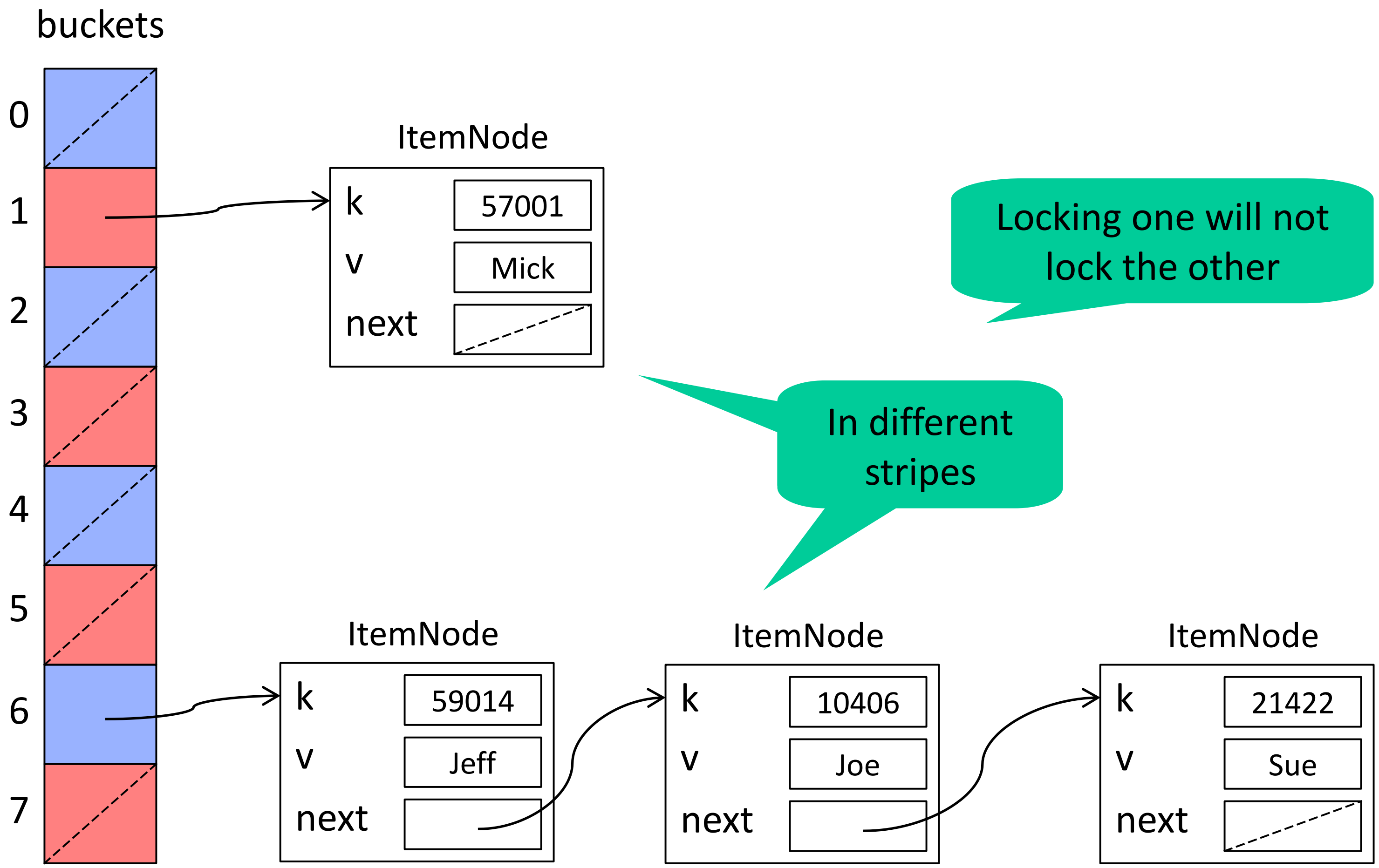
–two operations will work on different stripes

–hence will take different locks

- Less lock contention, better scalability



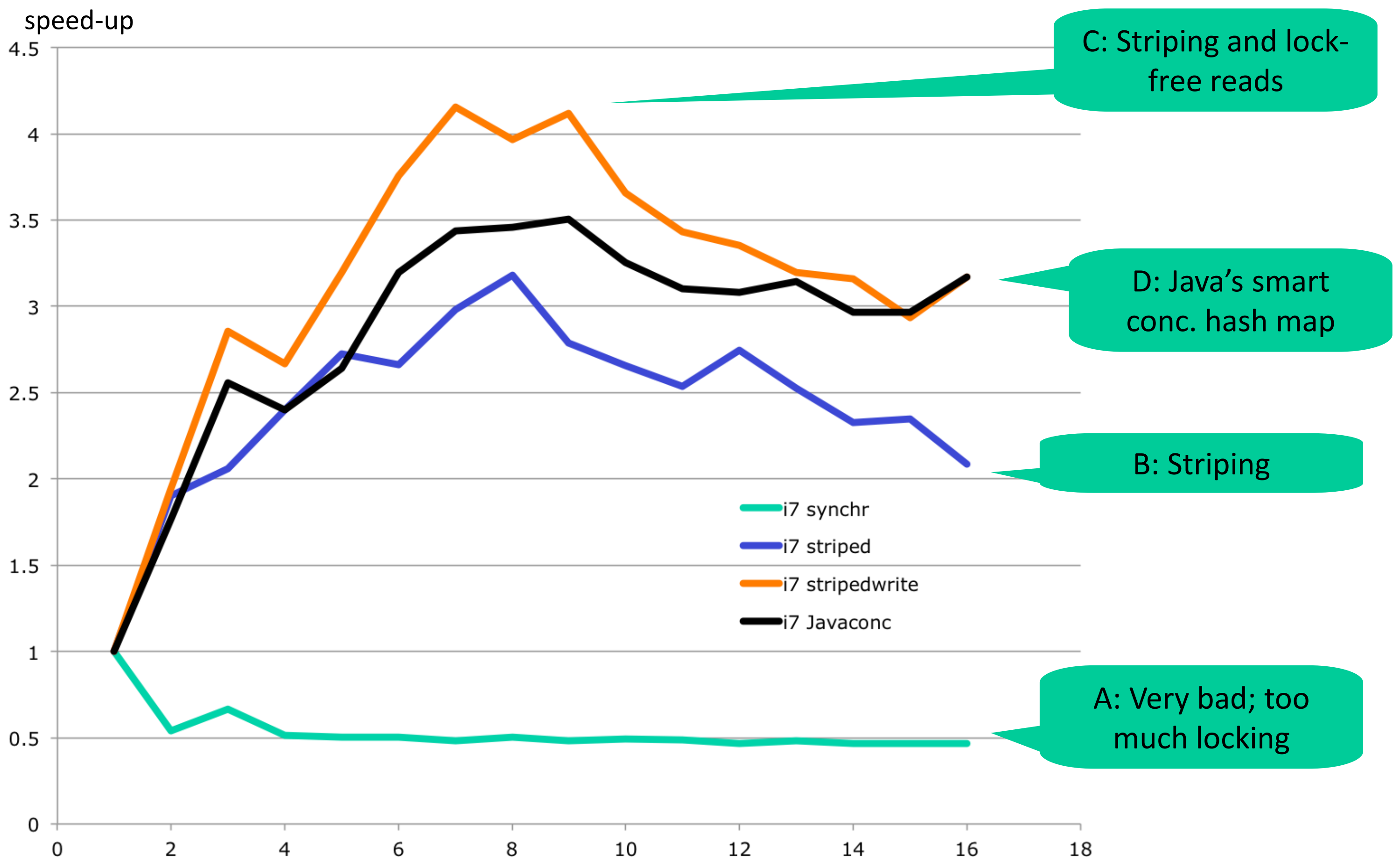
Bucket idea



Locking one will not lock the other

In different stripes

Reducing locking



Ultimate scalability

A web-shop, Facebook, …

We must give up thread safety,

but still maintain some sort of consistency

Ultimate scalability

A web-shop, Facebook, …

We must give up thread safety,

but still maintain some sort of consistency

Week 13