
CS2030 Lecture 12

Fork/Join Framework

Henry Chia (hchia@comp.nus.edu.sg)

Semester 2 2018 / 2019

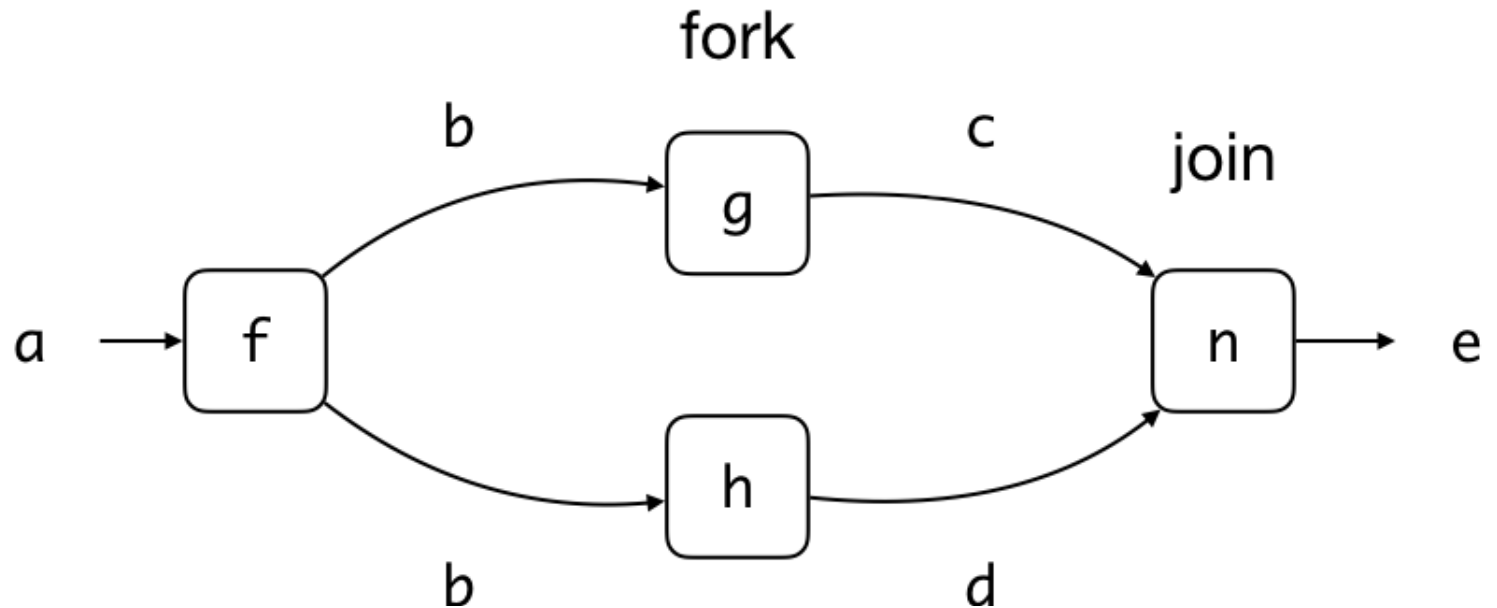
Lecture Outline

- Fork and join tasks
- Java's Fork/join framework
 - Sub-classing a `RecursiveTask`
- Thread pools
 - Global queue
 - Local deque (double-ended queue)
- Work stealing
- Order of fork and join
- Overhead of fork and join

Fork and Join

- Given the following program fragment and *computation* graph

```
b = f(a);  
c = g(b);  
d = h(b);  
e = n(c,d);
```



- $f(a)$ invoked before $g(b)$ and $h(b)$; $n(c,d)$ invoked after
- How about the order of $g(b)$ and $h(b)$?
 - If g and h does not produce side effects, then parallelize
 - **Fork** task g to execute at the same time as h , and **join** back task g later

Example: Summing an Array... *Recursively*

```
class Summer {  
    static int threshold;  
  
    static int sumLeftRight(int[] array, int low, int high) {  
        if (high - low < threshold) {  
            int sum = 0;  
            for (int i = low; i <= high; i++) {  
                sum += array[i];  
            }  
            return sum;  
        } else {  
            int middle = (low + high) / 2;  
            int leftSum = sumLeftRight(array, low, middle);  
            int rightSum = sumLeftRight(array, middle + 1, high);  
            return leftSum + rightSum;  
        }  
    }  
}  
  
int[] array = IntStream.rangeClosed(1, 10).toArray();  
Summer.threshold = Integer.parseInt(args[0]);  
int sum = sumLeftRight(array, 0, array.length - 1);
```

Transform to a Recursive Class

```
class Summer {
    int[] array;
    int low;
    int high;
    static int threshold;

    Summer(int[] array, int low, int high) {
        this.array = array;
        this.low = low;
        this.high = high;
    }

    int compute() {
        if (high - low < threshold) {
            int sum = 0;
            for (int i = low; i <= high; i++) {
                sum += array[i];
            }
            return sum;
        } else {
            int middle = (low + high) / 2;
            Summer left = new Summer(array, low, middle);
            Summer right = new Summer(array, middle + 1, high);
            return left.compute() + right.compute();
        }
    }
}
```

recursive new Task object

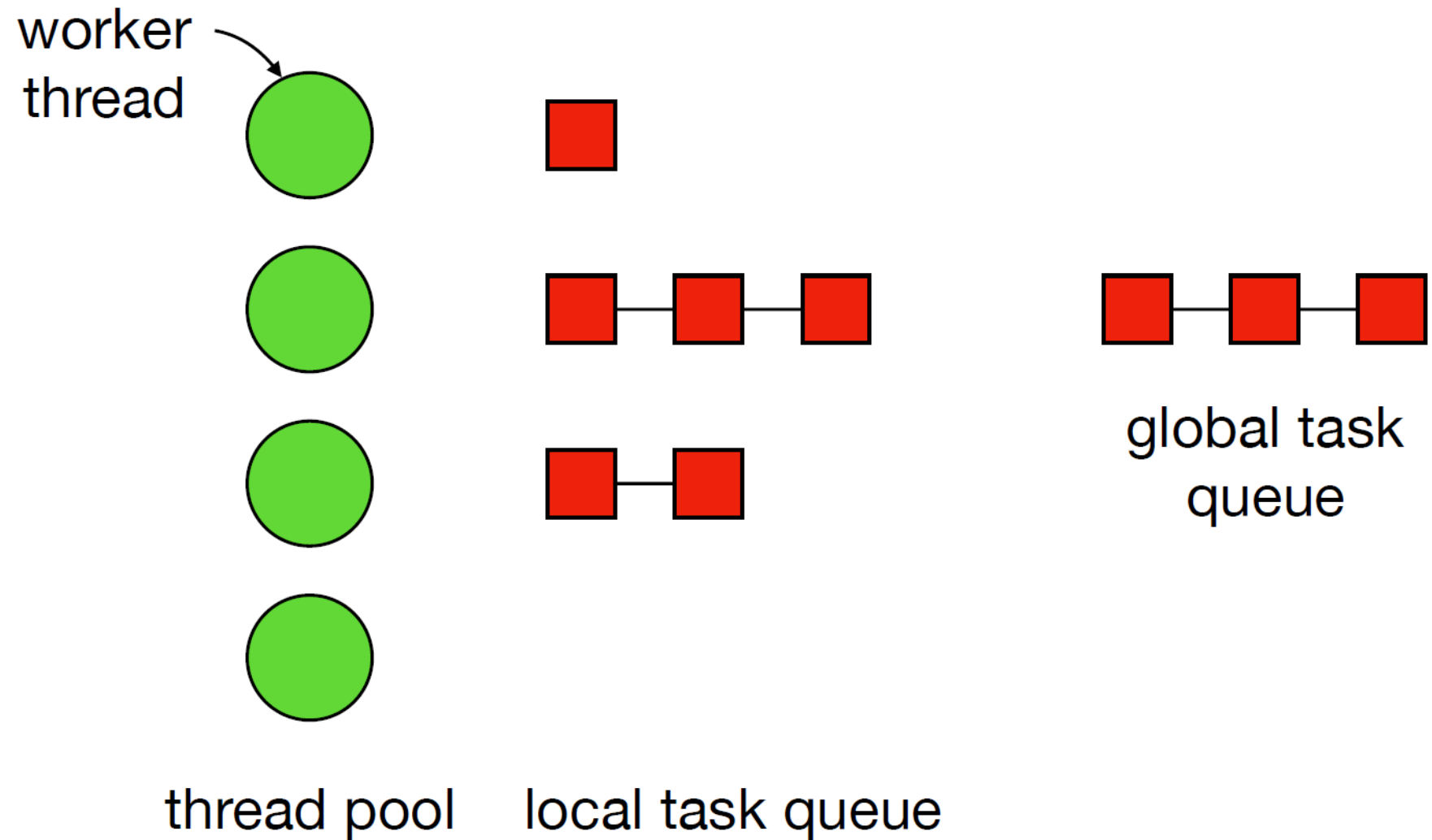
Subclassing RecursiveTask<T> for Fork/Join

```
import java.util.concurrent.RecursiveTask;

class Summer extends RecursiveTask<Integer> {
    ...
    @Override
    protected Integer compute() {
        if (high - low < threshold) {
            int sum = 0;
            for (int i = low; i < high; i++) {
                sum += array[i];
            }
            return sum;
        } else {
            System.out.println(thread.currentThread().getName() )
            int middle = (low + high) / 2;
            Summer left = new Summer(low, middle, array);
            Summer right = new Summer(middle, high, array);

            left.fork();
            return right.compute() + left.join();
        }
    }
}
```

Thread Pools



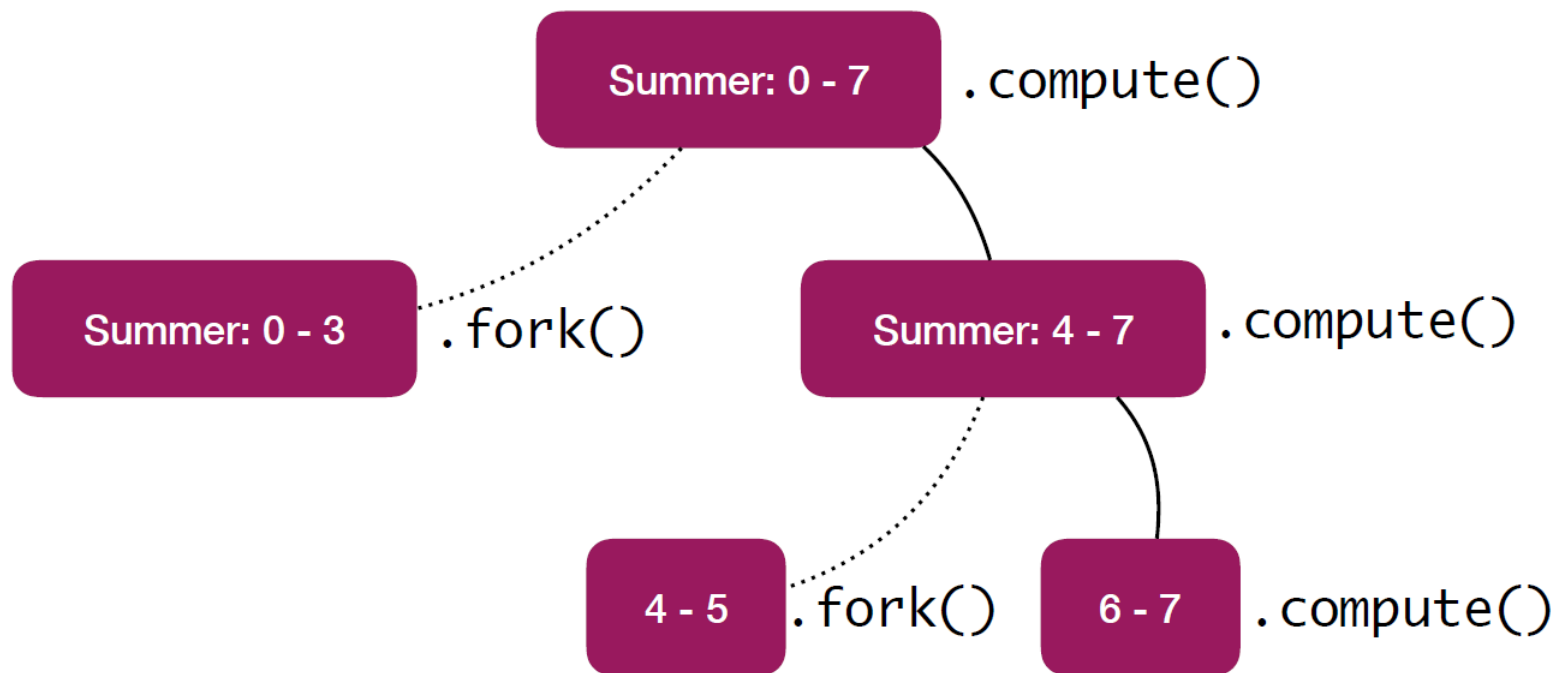
Thread Pools

- Java maintains a pool of *worker threads*
 - Each thread is an abstraction of a running task
 - Task submitted to the pool for execution, and joins the global queue or worker queue
 - Worker thread picks a task from the ^{global} queue to execute
- `ForkJoinPool` is the class that implements the thread pool for `RecursiveTask` (a sub-class of `ForkJoinTask`)
- To submit task to the thread pool for execution, either
 - `task.compute()` that invokes task immediately; may result in stack overflow if too many recursive tasks
 - `invoke(task)` that gets the task to join the queue, waiting to be carried out by a worker (recommended)

Example: Summing an Array

- Summing array of eight elements with threshold set to 2

0	1	2	3	4	5	6	7
3	3	1	3	6	8	4	2



Example: Summing an Array

```
@Override
protected Integer compute() {
    System.out.println(low + "," + high + ":" + Thread.currentThread().getName());
    if (high - low < threshold) {
        int sum = 0;
        for (int i = low; i < high; i++) {
            sum += array[i];
        }
        return sum;
    } else {
        int middle = (low + high) / 2;
        Summer left = new Summer(array, low, middle);
        Summer right = new Summer(array, middle + 1, high);
        left.fork();
        return right.compute() + left.join();
    }
}
```

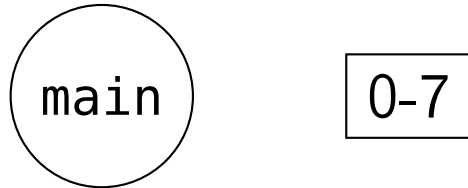
□ Running with ForkJoinPool.common.parallelism=2

```
0,7:main
4,7:main
6,7:main
0,3:ForkJoinPool.commonPool-worker-1
4,5:ForkJoinPool.commonPool-worker-2
2,3:ForkJoinPool.commonPool-worker-1
0,1:ForkJoinPool.commonPool-worker-2
```

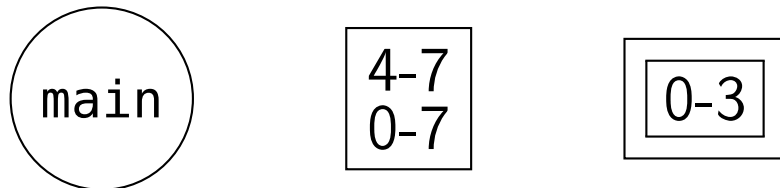
```
0,7:main
4,7:main
6,7:main
4,5:main
0,3:ForkJoinPool.commonPool-worker-2
0,1:ForkJoinPool.commonPool-worker-1
2,3:ForkJoinPool.commonPool-worker-2
```

Queuing of Forked Tasks

- main thread performs compute on task [0,7]

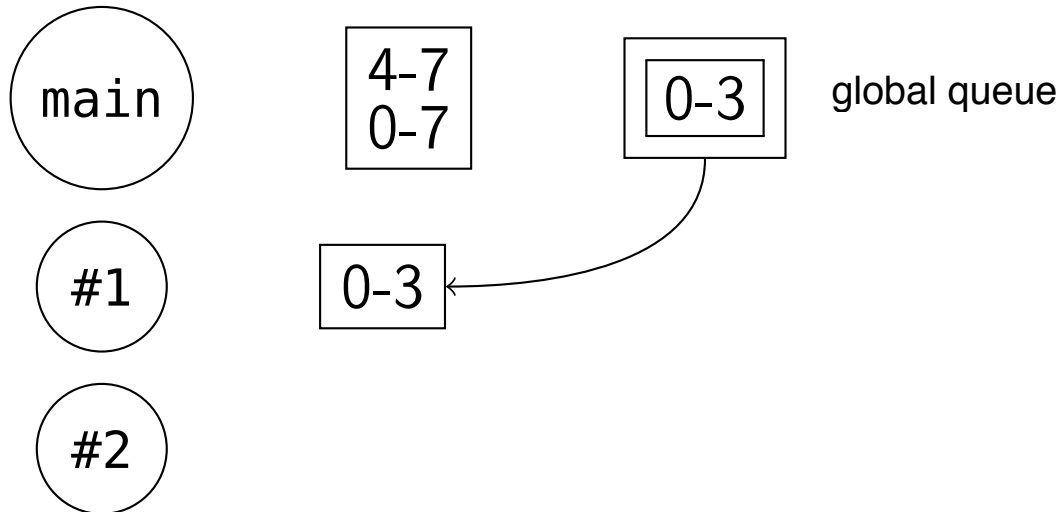


- main forks task [0,3] to global queue, then computes [4,7]

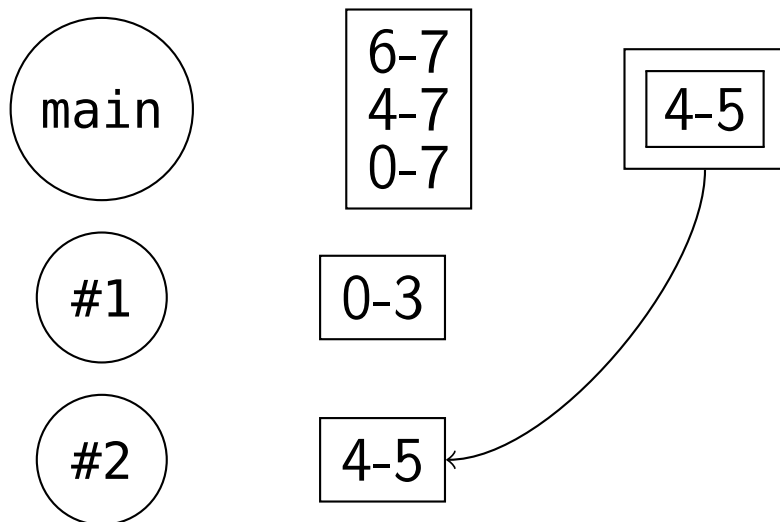


Work Fetching from Global Queue

- Worker #1 fetches task [0-3] from global queue

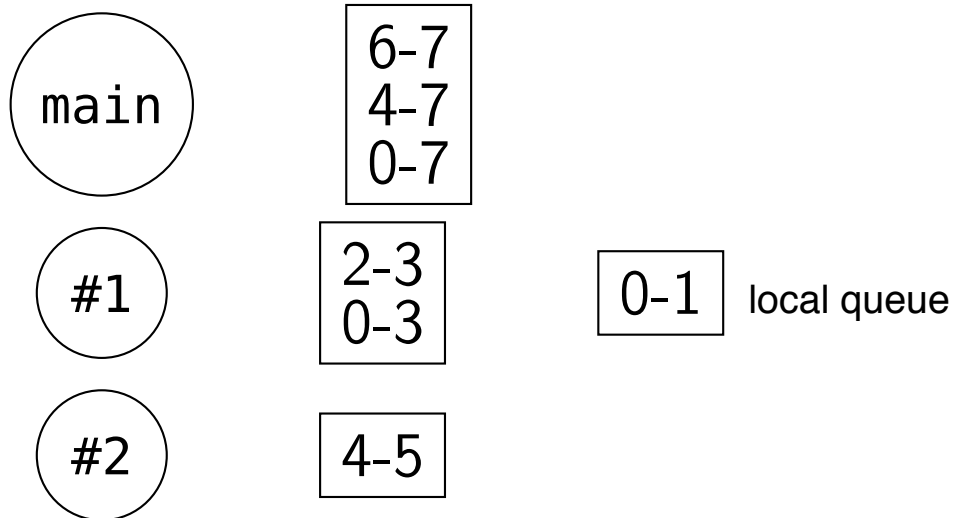


- main forks [4-5] & computes [6-7]; worker #2 fetches [4-5]

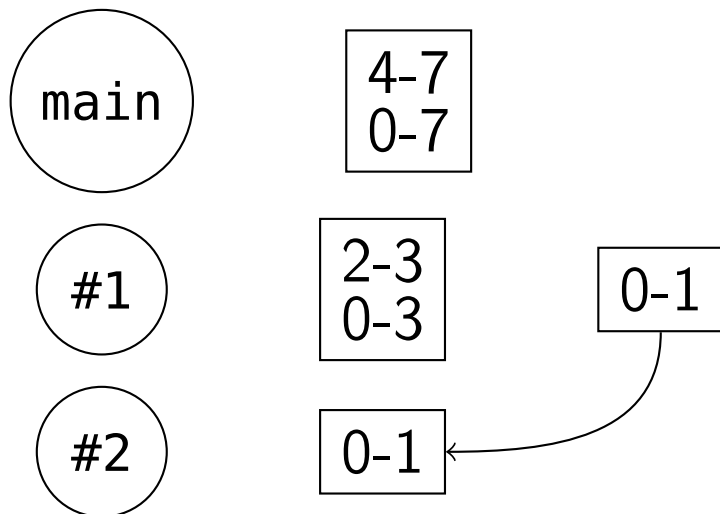


Work Stealing

- Worker #1 forks [0-1] and computes [2-3]

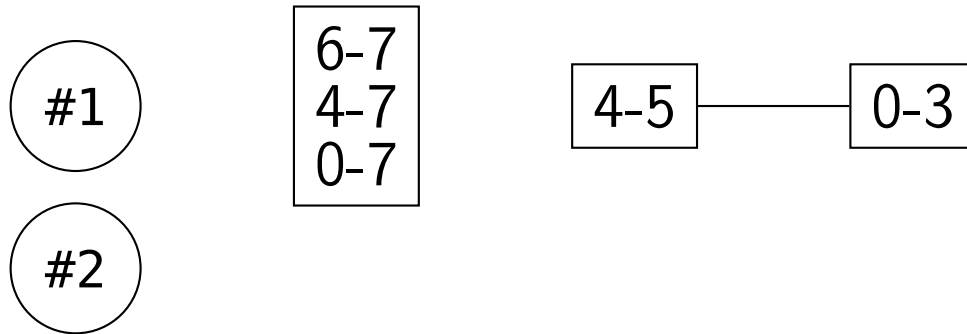


- Worker #2 completes [4-5] returns, and steals [0-1]



Local Deque (Double-Ended Queue)

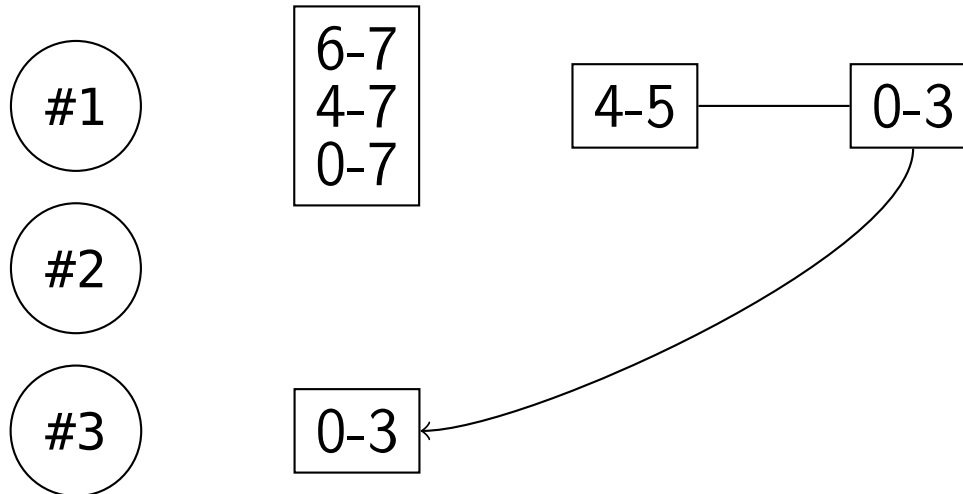
- Suppose main is busy
- Worker #1 processes 0-7: forks [0-3] and computes [4-7]
- Worker #1 processes 4-7: forks [4-5] and computes [6-7]



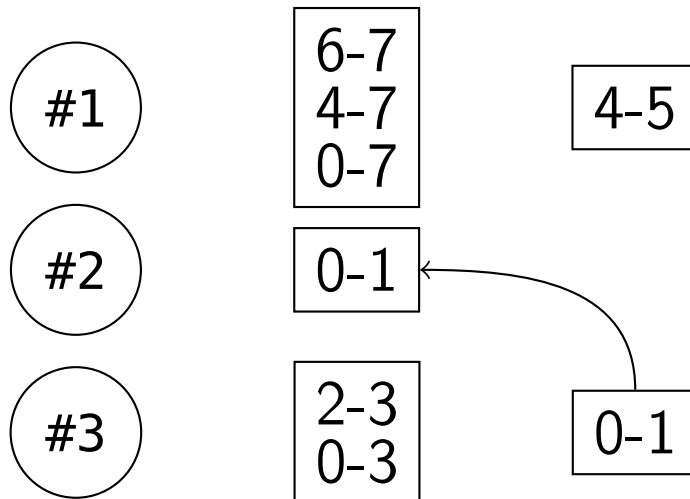
- When worker #1 completes [6-7], it makes more sense for him to work on [4-5], so as to complete 4-7]
- Local task queue is a double ended queue
 - Forked tasks added to the **head** of the queue
 - Steal tasks from the **end of the queue**
 - Rational: bigger tasks are stolen; smaller ones self-served

Work Stealing from Back of Dequeue

- Worker #3 steals task [0-3]

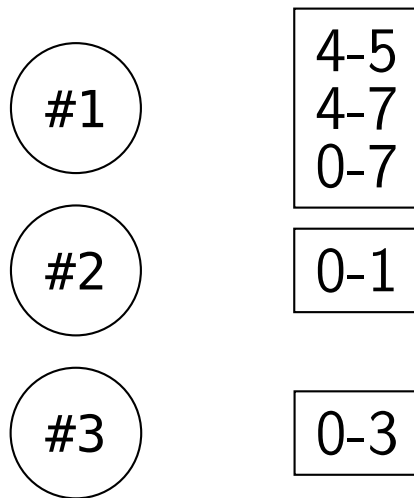


- Worker #3 forks [0-1] (worker #2 steals) and computes [2-3]



Another Possible Scenario...

- Worker #1 completes [6-7], but worker #2 has not
- Worker #3 completes [2-3], but is now blocked waiting for worker #2 to return with a value
- Worker #1 can then service [4-5] from the head of its queue



Order of Fork/Join

```
@Override
protected Integer compute() {
    System.out.println(low + ", " + high + ":" + Thread.currentThread().getName());
    if (high - low < threshold) {
        int sum = 0;
        for (int i = low; i < high; i++) {
            sum += array[i];
        }
        return sum;
    } else {
        JOIN INNER OUT!!!
        int middle = (low + high) / 2;
        Summer left = new Summer(array, low, middle);
        Summer right = new Summer(array, middle + 1, high);

        left.fork();
        right.fork();

        return right.compute() + left.compute();
    }
}
```

- ☐ How about using only forks and joins
- ☐ Does the ordering matter?

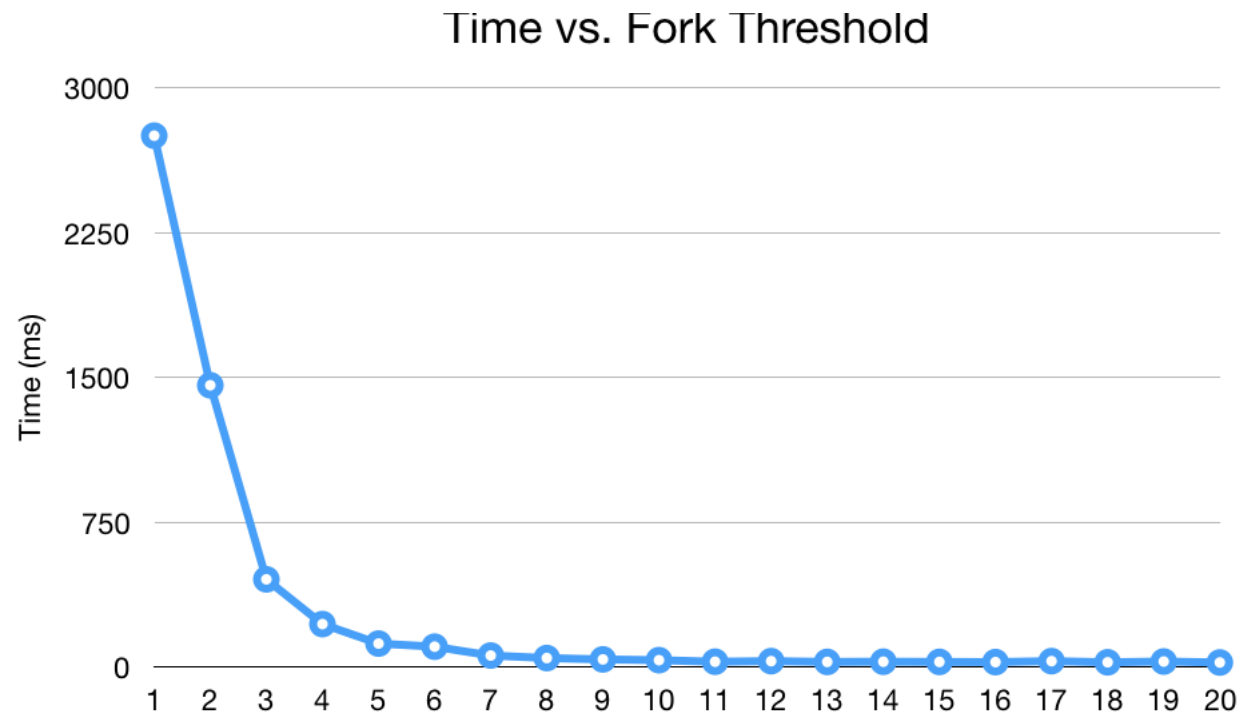
Order of Fork/Join

- Fork-join pair acts like a call (fork) and return (join) from a parallel recursive function.
 - Returns (joins) should be performed innermost-first
 - `a.fork(); b.fork(); b.join(); a.join();` is likely to be substantially more efficient than joining a before b.
- Work-stealing threadpools have a fixed number of threads
 - Any blocking operation in one of these threads will reduce overall performance
- Will all threads be blocked?
 - Synchronization primitive classes recognize when a thread is about to be blocked and create a *compensation thread* before the current thread blocks

EXTREMELY UNLIKELY

Overhead of Fork/Join

- Forking and joining creates additional overhead
 - wrap the computation in an object
 - submit object to a queue of tasks
 - workers go through the queue to execute tasks



Lecture Summary

- Appreciate the use of fork and join in parallel/concurrent programming
- Understand how tasks are forked in a local dequeue, and why the ordering of forks and joins matter
- Understand how work stealing distributes tasks among worker threads
- Appreciate the overhead involved in parallelizing using fork and join