Learning Objectives in this Part of the Lesson

- Understand how the Java fork-join framework processes tasks in parallel
- Recognize the structure & functionality of the fork-join framework
- Know how the fork-join framework is implemented internally
Java Fork-Join Pool Framework Internals
Java Fork-Join Pool Framework Internals

- Each worker thread in a fork-join pool runs a loop that scans for (sub-)tasks to execute.
Each worker thread in a fork-join pool runs a loop that scans for (sub-)tasks to execute.

The goal is to keep the worker threads as busy as possible!
Each worker thread in a fork-join pool runs a loop that scans for (sub-)tasks to execute

- The goal is to keep the worker threads as busy as possible!
- A worker thread only blocks waiting for work if no (sub-)tasks are available to run
Java Fork-Join Pool Framework Internals

- Each worker thread in a fork-join pool runs a loop that scans for (sub-)tasks to execute
  - The goal is to keep the worker threads as busy as possible!
  - A worker thread only blocks waiting for work if no (sub-)tasks are available to run

Blocking a working thread is very costly on modern processors
Each worker thread in a fork-join pool runs a loop that scans for (sub-)tasks to execute

- The goal is to keep the worker threads as busy as possible!
- A worker thread only blocks waiting for work if no (sub-)tasks are available to run
- Each worker thread therefore checks multiple input sources for (sub-)tasks to execute
A worker thread has a “double-ended queue” (aka “deque”) that serves as its main input source.

See en.wikipedia.org/wiki/Double-ended_queue
A worker thread has a “double-ended queue” (aka “deque”) that serves as its main input source.

- Implemented by WorkQueue

See java8/util/concurrent/ForkJoinPool.java
If a task run by a worker thread calls `fork()` the new task is pushed on the head of the worker’s deque.

<table>
<thead>
<tr>
<th>Sub-Task1.1</th>
<th>WorkQueue</th>
<th>WorkQueue</th>
<th>WorkQueue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sub-Task1.2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sub-Task1.3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sub-Task1.4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sub-Task2.4</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

See gee.cs.oswego.edu/dl/papers/fj.pdf
Java Fork-Join Pool Framework Internals

- If a task run by a worker thread calls fork() the new task is pushed on the head of the worker’s deque.
- A worker thread processes its deque in LIFO order.

See en.wikipedia.org/wiki/Stack_(abstract_data_type)
### Java Fork-Join Pool Framework Internals

- If a task run by a worker thread calls fork() the new task is pushed on the head of the worker’s deque.
- A worker thread processes its deque in LIFO order, i.e.
- It pops (sub-)tasks from the head of its deque & runs them to completion.

#### WorkQueue

<table>
<thead>
<tr>
<th>Sub-Task1.1</th>
<th>Sub-Task1.2</th>
<th>Sub-Task1.3</th>
<th>Sub-Task1.4</th>
</tr>
</thead>
</table>

#### WorkQueue

<table>
<thead>
<tr>
<th>Sub-Task3.3</th>
<th>Sub-Task3.4</th>
</tr>
</thead>
</table>

Java Fork-Join Pool Framework Internals

- If a task run by a worker thread calls fork() the new task is pushed on the head of the worker’s deque
- A worker thread processes its deque in LIFO order
- LIFO order improves locality of reference & cache performance

If a task run by a worker thread calls `join()` it pitches in” to pop & execute (sub-)tasks.
Java Fork-Join Pool Framework Internals

- If a task run by a worker thread calls join() it pitches in” to pop & execute (sub-)tasks

"Collaborative Jiffy Lube” model of processing!

See en.wikipedia.org/wiki/Jiffy_Lube
To maximize core utilization, idle worker threads “steal” work from the tail of busy threads’ deques.

See [docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html](docs.oracle.com/javase/tutorial/essential/concurrency/forkjoin.html)
To maximize core utilization, idle worker threads “steal” work from the tail of busy threads’ deques.

The worker thread deque to steal from is selected randomly to lower contention.
Java Fork-Join Pool Framework Internals

- Tasks are stolen in FIFO order

See [en.wikipedia.org/wiki/FIFO_(computing_and_electronics)](en.wikipedia.org/wiki/FIFO_(computing_and_electronics))
Java Fork-Join Pool Framework Internals

• Tasks are stolen in FIFO order, e.g. WorkQueue
• Minimizes contention with thread owning the deque

Java Fork-Join Pool Framework Internals

- Tasks are stolen in FIFO order, e.g. WorkQueue
  - Minimizes contention with thread owning the deque
  - An older stolen task may provide a larger unit of work

- WorkQueue
  - Sub-Task_{1,2}
  - Sub-Task_{1,3}
  - Sub-Task_{1,4}

- poll()
Java Fork-Join Pool Framework Internals

- Tasks are stolen in FIFO order, e.g.
  - Minimizes contention with thread owning the deque
  - An older stolen task may provide a larger unit of work
  - Enables further recursive decompositions by the stealing thread
Java Fork-Join Pool Framework Internals

- The WorkQueue deque that implements work-stealing minimizes locking contention

See [www.dre.vanderbilt.edu/~schmidt/PDF/work-stealing-deque.pdf](http://www.dre.vanderbilt.edu/~schmidt/PDF/work-stealing-deque.pdf)
Java Fork-Join Pool Framework Internals

- The WorkQueue deque that implements work-stealing minimizes locking contention
- `push()` & `pop()` are only called by the owning worker thread
Java Fork-Join Pool Framework Internals

- The WorkQueue deque that implements work-stealing minimizes locking contention
  - push() & pop() are only called by the owning worker thread
  - These operations use wait-free “compare-and-swap” (CAS) operations

See en.wikipedia.org/wiki/Compare-and-swap
The WorkQueue deque that implements work-stealing minimizes locking contention
- `push()` & `pop()` are only called by the owning worker thread
- `poll()` may be called from another worker thread to “steal” a (sub-)task
Java Fork-Join Pool Framework Internals

- The WorkQueue deque that implements work-stealing minimizes locking contention
  - `push()` & `pop()` are only called by the owning worker thread
  - `poll()` may be called from another worker thread to "steal" a (sub-)task
  - May not always be wait-free

See gee.cs.oswego.edu/dl/papers/fj.pdf
Java Fork-Join Pool Framework Internals

• The WorkQueue deque that implements work-stealing minimizes locking contention
  • push() & pop() are only called by the owning worker thread
  • poll() may be called from another worker thread to “steal” a (sub-)task
  • May not always be wait-free
    • See “Implementation Overview” comments in the ForkJoinPool source code for details..

See java8/util/concurrent/ForkJoinPool.java
End of the Java Fork-Join Pool Framework (Part 3)