Lecture 18: Abstract vs. Real Performance

"Everything You Ever Wanted to Know About HJLib but Were Too Afraid to Ask"

Zoran Budimlić and Mack Joyner
{zoran, mjoyner}@rice.edu

http://comp322.rice.edu

Functional Approach to Parallelism

- "Functional": futures, future tasks, streams, data-driven tasks and futures
- "Not-so functional": async tasks and finish scopes, tasks that modify shared memory
- Advantages to functional approach
  - Easier to reason about
  - Don't have to worry about data races
  - Leads to compact, elegant, easy to read code
  - Easy to scale to massively parallel (because you don't need to worry about data races)
- Disadvantages
  - May be hard to express exactly the computation graph you need (i.e. a finish scope with millions of tasks)
  - May be more expensive to execute (blocking future.get() vs. simply reading a shared memory location)
  - May need additional data structures (futures, data-driven futures) to express the computation
  - May need copying of data structures to avoid data races and mutation
  - Hard to scale to massively parallel (because of overheads)

Abstract vs. Real Performance

- Abstract performance
  - Focus on operation counts for WORK and CPL, regardless of actual execution time
  - Ignore the nitty-gritty of task creation and execution overhead
  - Same “performance” regardless of the machine
- Real performance
  - Lots of things happening “under the hood”
  - Operating system, runtime and hardware all have an impact
  - Process creation/execution vs. thread creation/execution vs. task creation/execution
  - Tasks could be blocked, waiting on some event
  - Complex matter, but important to at least have a general idea of the costs

Lab 4: Recursive Task Parallelism

```java
private static double recursiveMaxParallel(final double[] inX, final int start, final int end)
        throws SuspendableException
{
    if (end - start == 2) {
        doWork(1);
        return 1/inX[end - 1] + 1/inX[start];
    } else {
        var bottom = future(() -> recursiveMaxParallel(inX, start, (end + start) / 2));
        var top = future(() -> recursiveMaxParallel(inX, (end+start) / 2, end));
        var bVal = bottom.get();
        var tVal = top.get();
        doWork(1);
        return bVal + tVal;
    }
}
```

Lab 4: Recursive Task Parallelism

```java
private static double recursiveMaxParallel(final double[] inX, final int start, final int end)
        throws SuspendableException
{
    if (end - start == 2) {
        doWork(1);
        return 1/inX[end - 1] + 1/inX[start];
    } else {
        var bottom = future(() -> recursiveMaxParallel(inX, start, (end + start) / 2));
        var top = future(() -> recursiveMaxParallel(inX, (end+start) / 2, end));
        var bVal = bottom.get();
        var tVal = top.get();
        doWork(1);
        return bVal + tVal;
    }
}
```

Test Results	4 sec 459 ms
edu.rice.comp322.Lab4CorrectnessTest	4 sec 459 ms
testReciprocalParallelism2Futures	241 ms
testReciprocalParallelism4Futures	58 ms
testReciprocalParallelism8Futures	58 ms
testReciprocalMaxParallelism	4 sec 102 ms

Cutoff Strategy for Recursive Task Parallelism

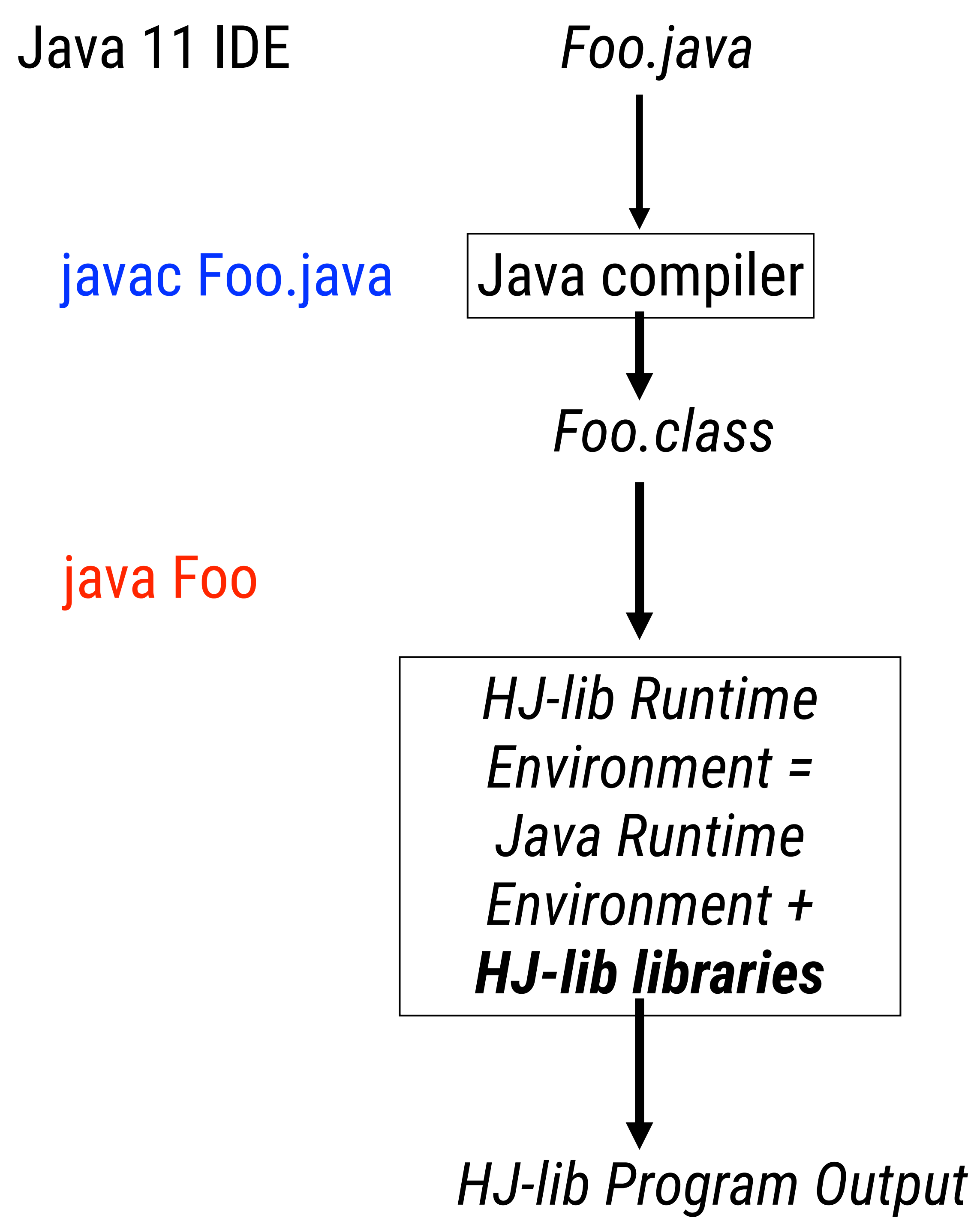
```
private static double recursiveMaxParallelCutoff(final double[] inX, final int start, final int end,
                                                  final int threshold) throws SuspendableException {
    if (end - start ≤ threshold) {
        double sum = 0.0;
        for(int i = start; i < end; i++) {
            doWork(1);
            sum = sum + 1 / inX[i];
        }
        return sum;
    } else {
        var bottom = future(() → recursiveMaxParallelCutoff(inX, start, (end + start) / 2, threshold));
        var top = future(() → recursiveMaxParallelCutoff(inX, (end+start) / 2, end, threshold));
        var bVal = bottom.get();
        var tVal = top.get();
        doWork(1);
        return bVal + tVal;
    }
}
```

Cutoff Strategy for Recursive Task Parallelism

```
private static double recursiveMaxParallelCutoff(final double[] inX, final int start, final int end,
                                                  final int threshold) throws SuspendableException {
    if (end - start ≤ threshold) {
        double sum = 0.0;
        for(int i = start; i < end; i++) {
            doWork(1);
            sum = sum + 1 / inX[i];
        }
        return sum;
    } else {
        var bottom = future(() → recursiveMaxParallelCutoff(inX, start, (end + start) / 2, threshold));
        var top = future(() → recursiveMaxParallelCutoff(inX, (end+start) / 2, end, threshold));
        var bVal = bottom.get();
        var tVal = top.get();
        doWork(1);
        return bVal + tVal;
    }
}
```

```
Execution with threshold 64000 took 56 milliseconds.
Execution with threshold 128000 took 54 milliseconds.
Execution with threshold 256000 took 4 milliseconds.
Execution with threshold 512000 took 3 milliseconds.
Execution with threshold 1024000 took 6 milliseconds.
Execution with threshold 2048000 took 10 milliseconds.
Execution with threshold 4096000 took 11 milliseconds.
```

HJ-lib Compilation and Execution Environment



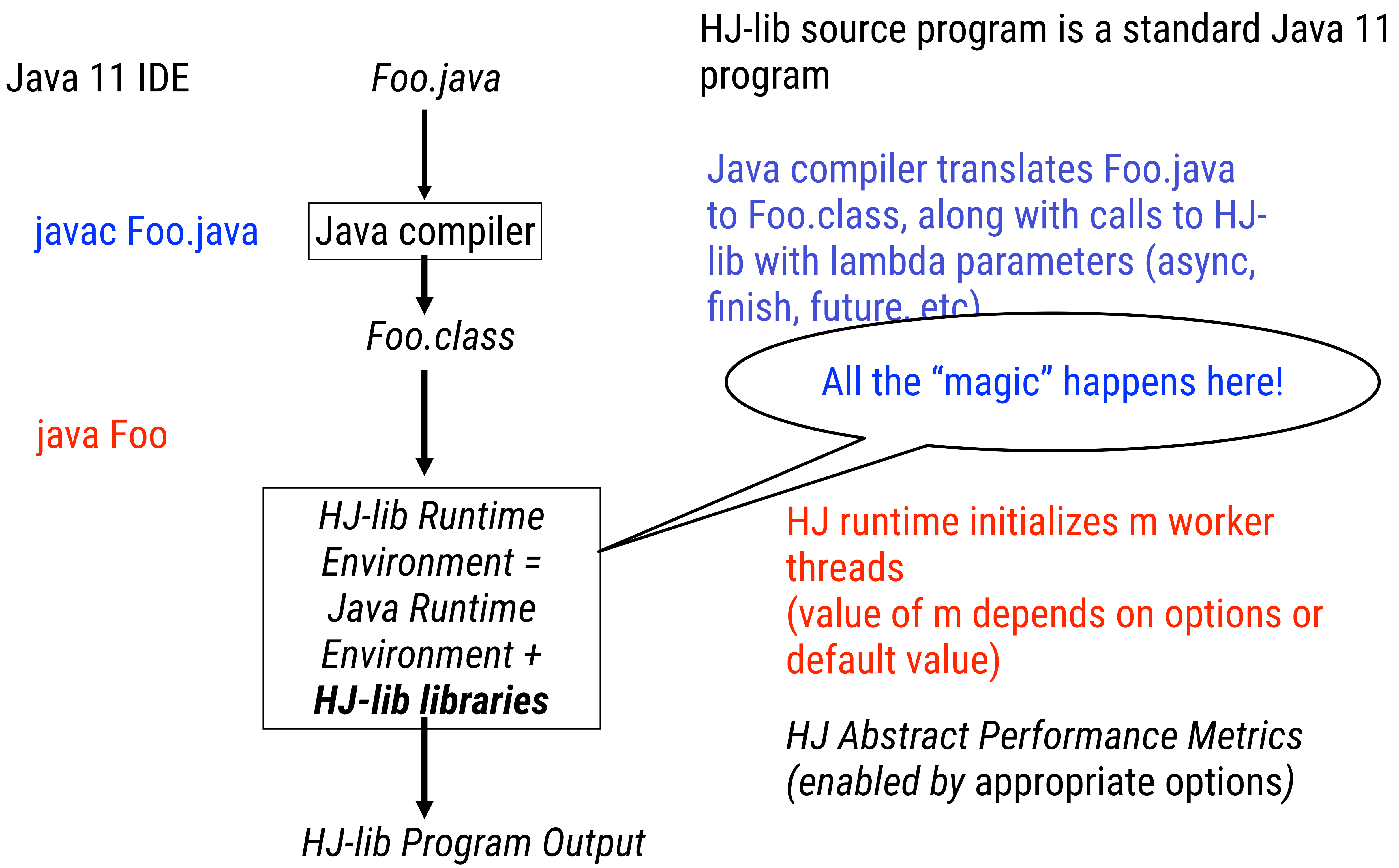
HJ-lib source program is a standard Java 11 program

Java compiler translates Foo.java to Foo.class, along with calls to HJ-lib with lambda parameters (async, finish, future, etc)

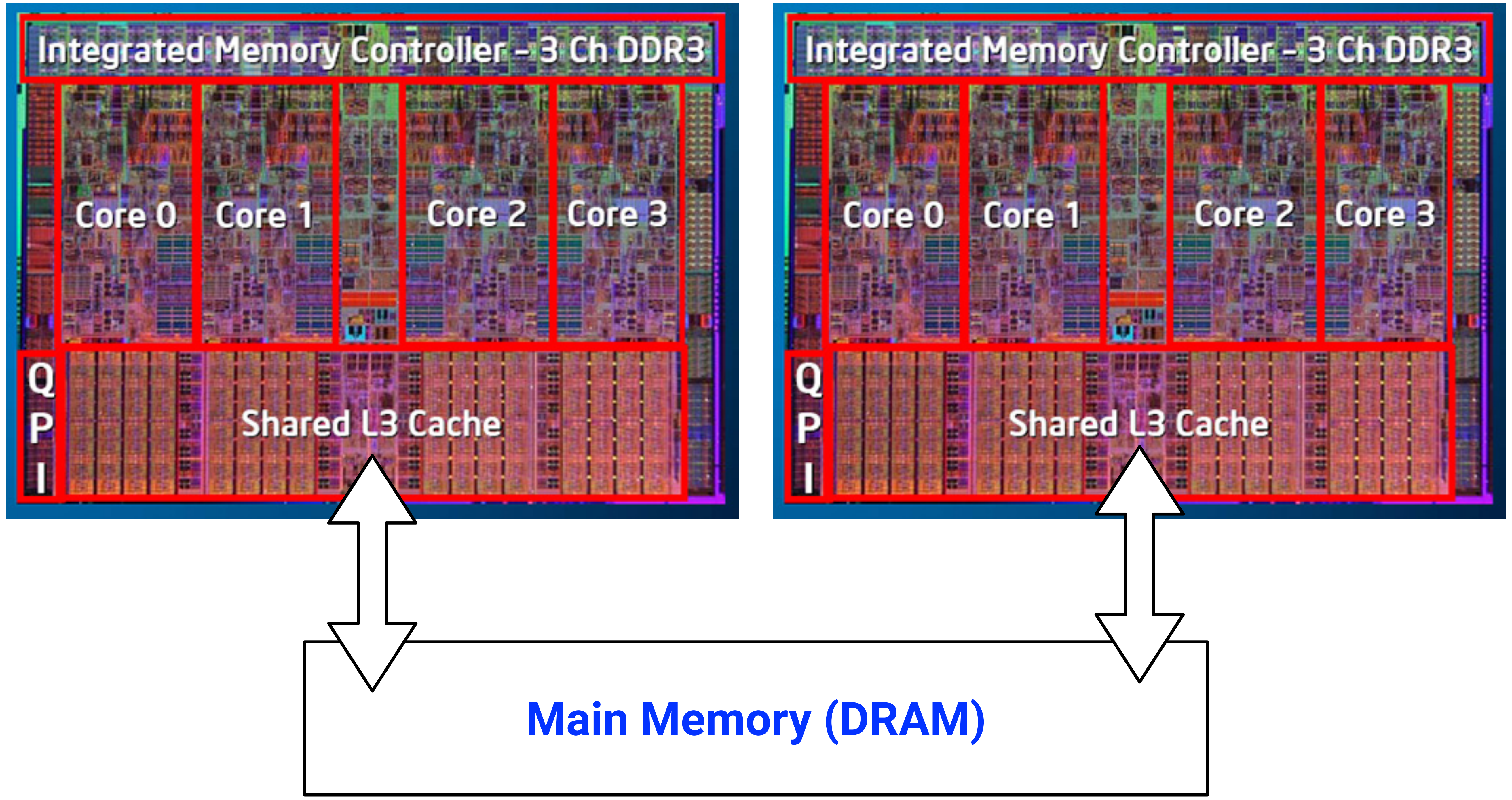
HJ runtime initializes m worker threads
(value of m depends on options or default value)

HJ Abstract Performance Metrics (enabled by appropriate options*)*

HJ-lib Compilation and Execution Environment



Looking under the hood - let's start with the hardware

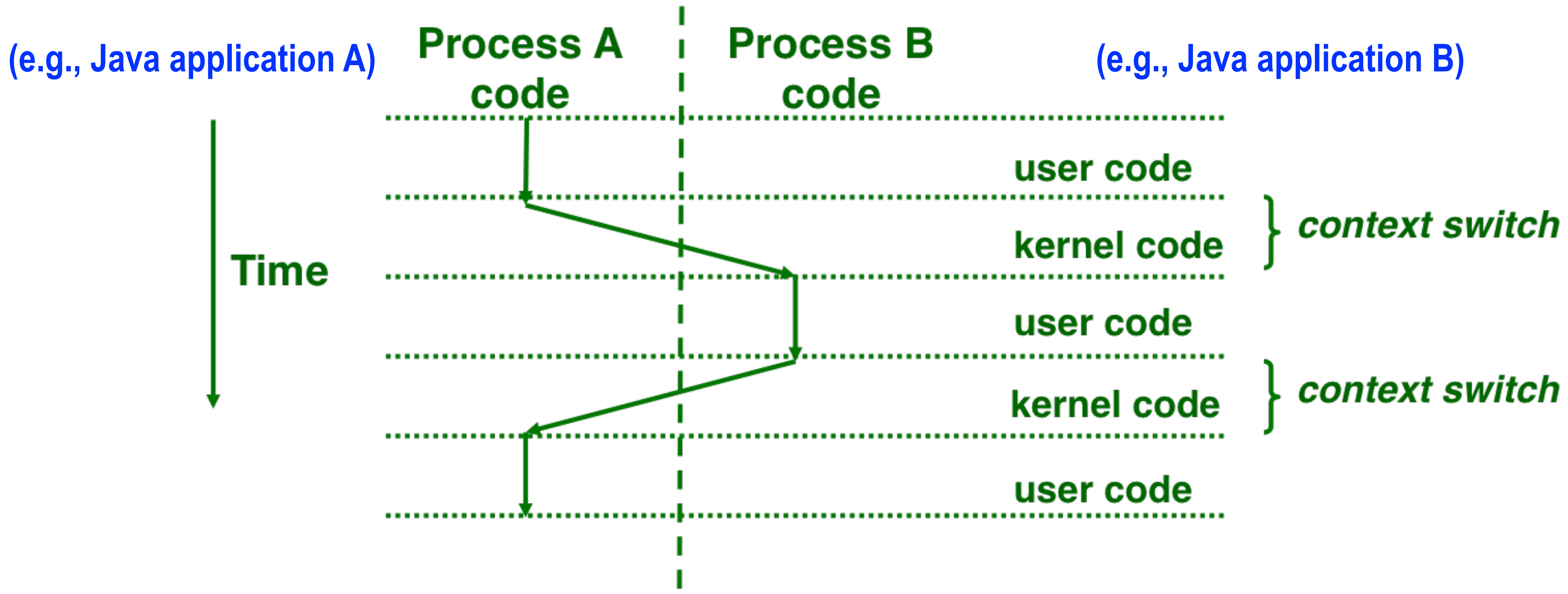


How does a process run on a single core?

Processes are managed by OS kernel

- **Important: the kernel is not a separate process, but rather runs as part of some user process**

Control flow passes from one process to another via a context switch

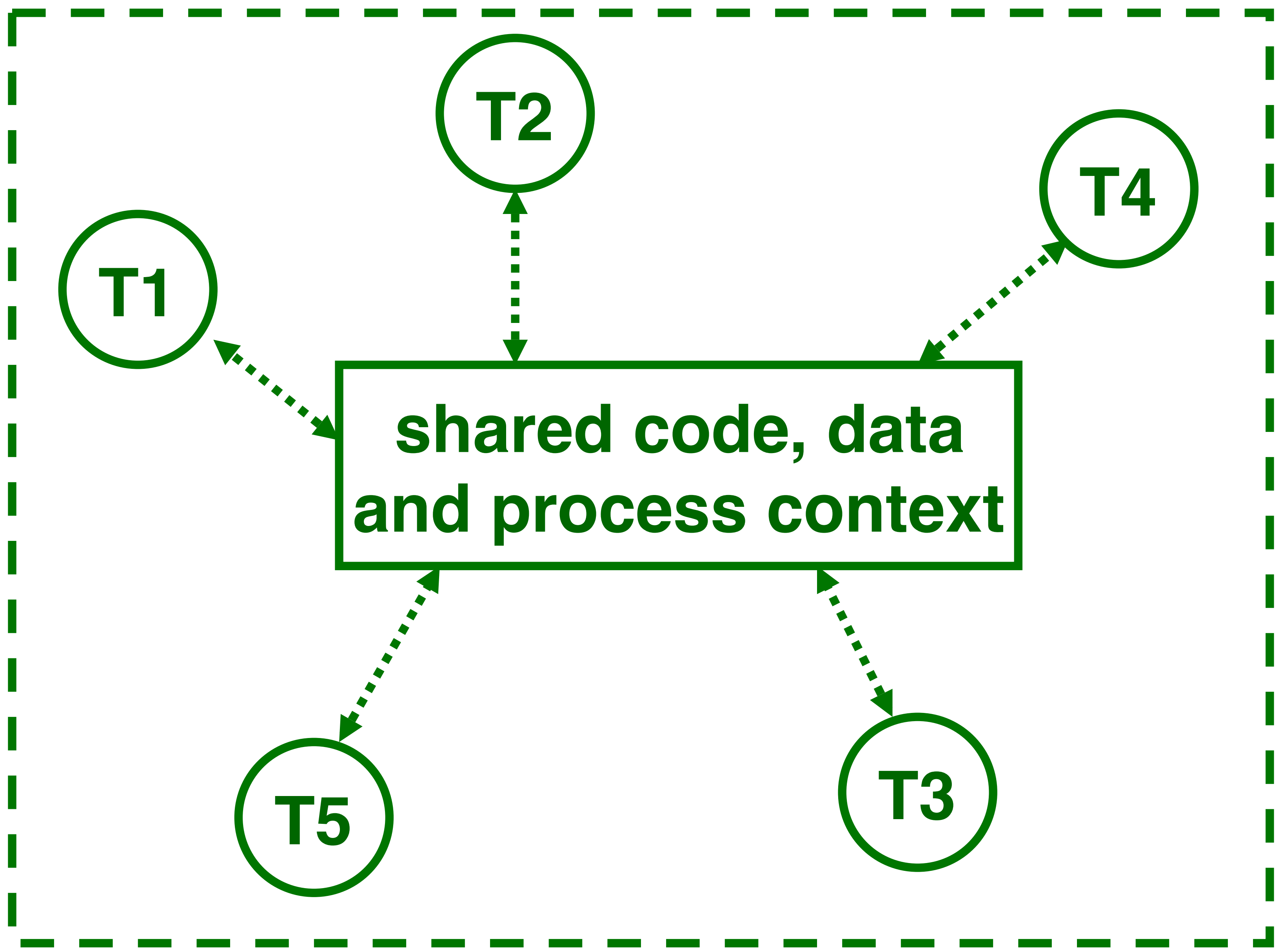


Context switches between two processes can be very expensive!

Source: COMP 321 lecture on Exceptional Control Flow (Alan Cox)

What happens when we execute a Java program

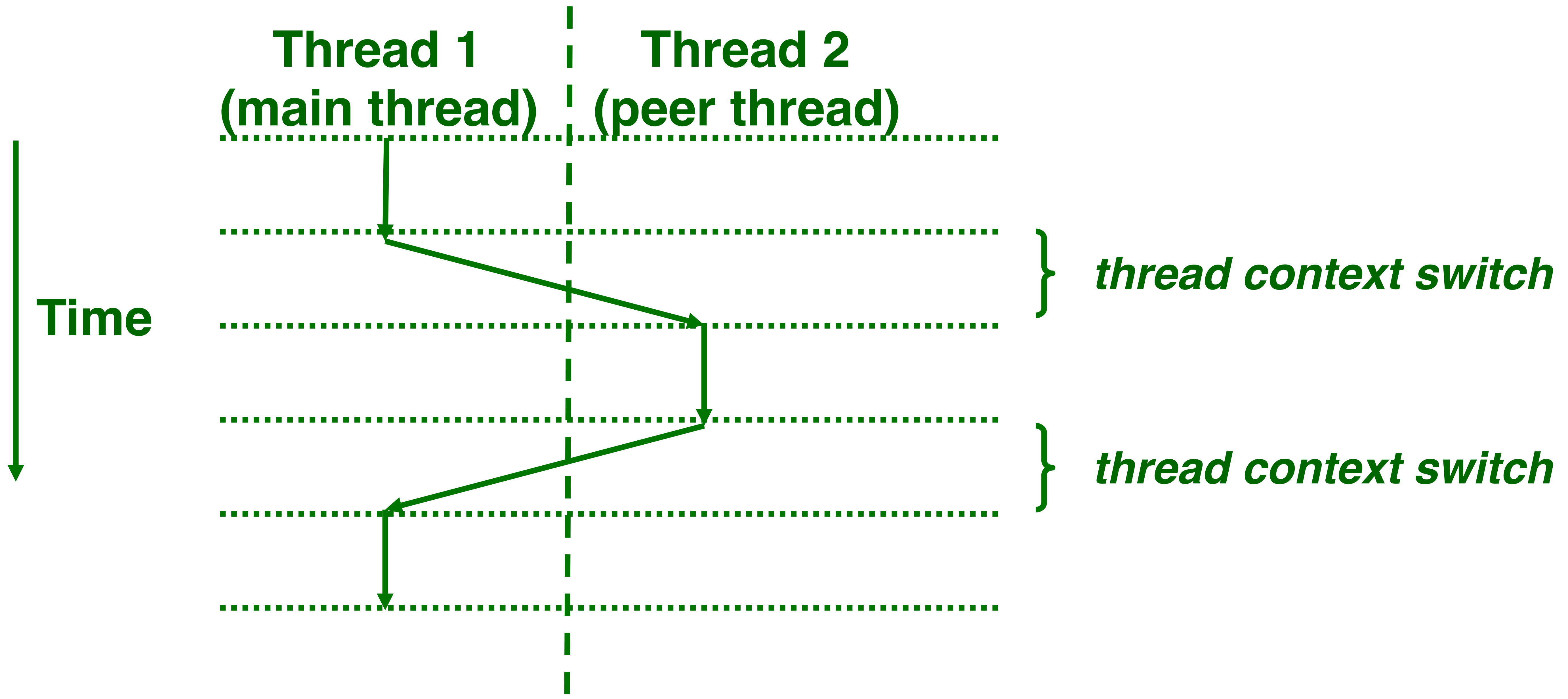
- A Java program executes in a single Java Virtual Machine (JVM) process with multiple threads
- Threads associated with a single process can share the same data
- Java main program starts with a single thread (T1), but can create additional threads (T2, T3, T4, T5) via library calls
- Java threads may execute concurrently on different cores, or may be context-switched on the same core



Java application with five threads —- T1, T2, T3, T4, T5 — all of which can access a common set of shared objects

Figure source: COMP 321 lecture on Concurrency (Alan Cox)

Thread-level Context Switching on the same processor core

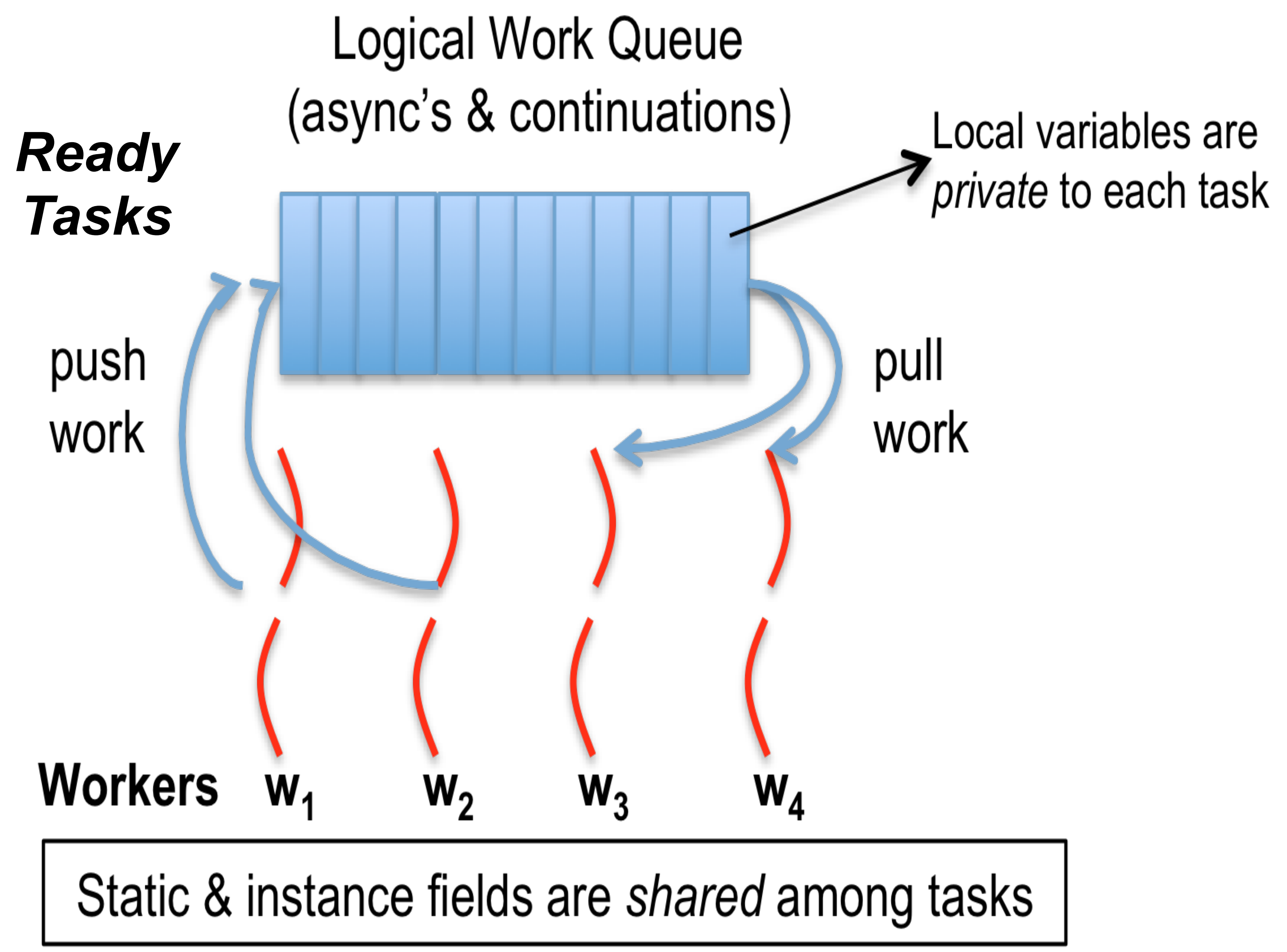


- Thread context switch is cheaper than a process context switch, but is still expensive (just not "very" expensive!)
- It would be ideal to just execute one thread per core (or hardware thread context) to avoid context switches

Figure source: COMP 321 lecture on Concurrency (Alan Cox)

Now, what happens is a task-parallel Java program (e.g., HJ-lib, Java Fork/Join, etc.)

HJ-Lib Tasks & Continuations
Worker threads
Operating System
Hardware cores



- HJ-lib runtime creates a *small number of worker threads*, typically one per core
- Workers push new tasks and "continuations" into a logical work queue
- Workers pull task/continuation work items from logical work queue when they are idle (remember greedy scheduling?)

Task-Parallel Model: Checkout Counter Analogy

- Think of each checkout counter as a processor core

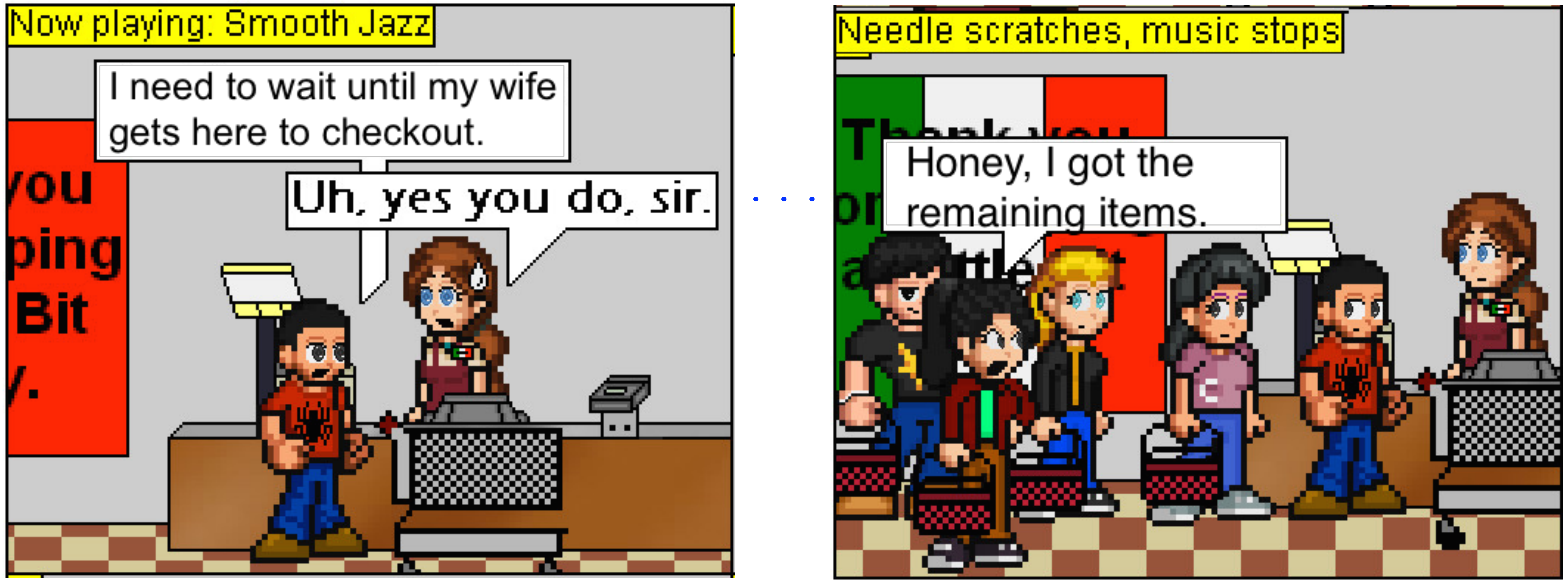
Image sources: http://www.deviantart.com/art/Randomness-20-178737664, http://www.wholefoodsmarket.com/blog/whole-story/new-haight-ashbury-store

Task-Parallel Model: Checkout Counter Analogy

- Think of each checkout counter as a processor core
- And of customers as tasks

Image sources: http://www.deviantart.com/art/Randomness-20-178737664, http://www.wholefoodsmarket.com/blog/whole-story/new-haight-ashbury-store

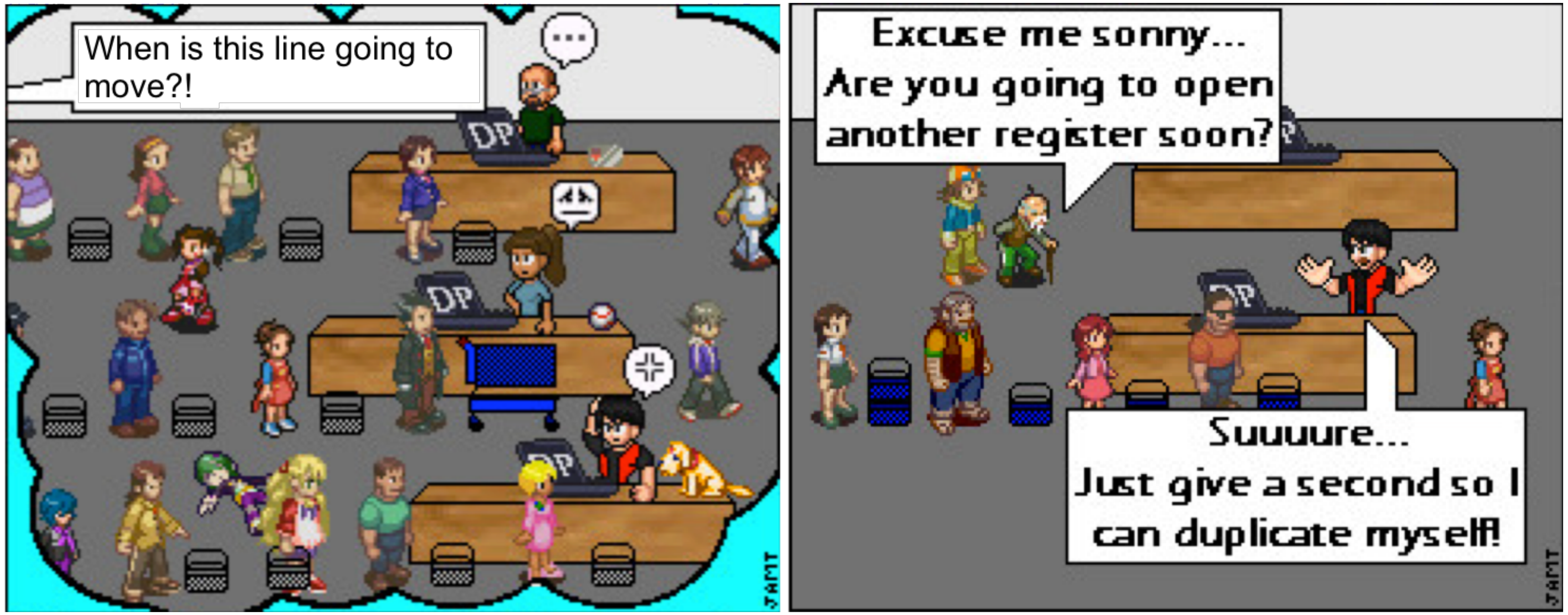
All is well until a task blocks …



- A blocked task/customer can hold up the entire line
- What happens if each checkout counter has a blocked customer?

source: http://viper-x27.deviantart.com/art/Checkout-Lane-Guest-Comic-161795346

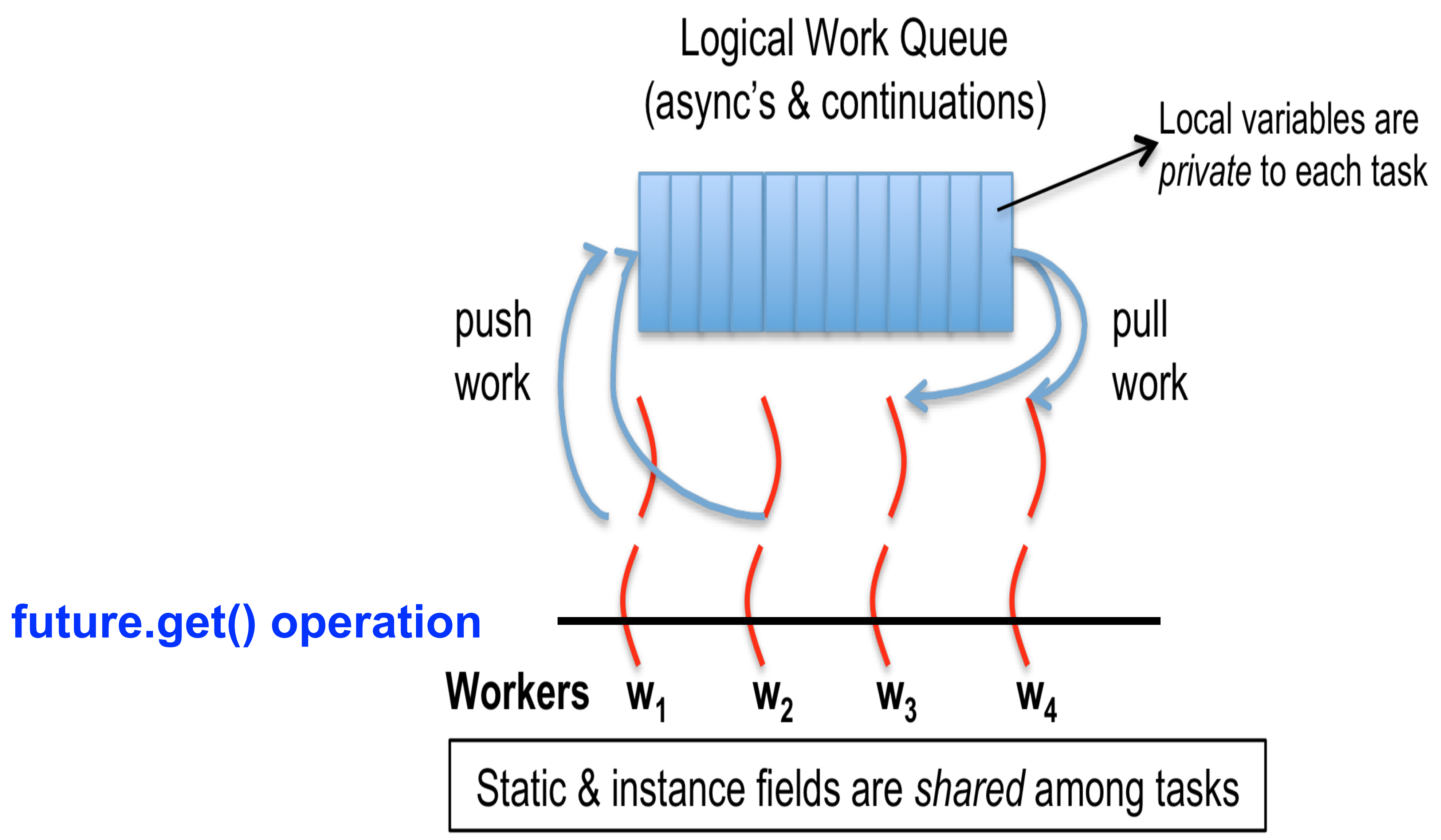
Approach 1: Create more worker threads (as in HJ-Lib's Blocking Runtime)



- Creating too many worker threads can exhaust system resources (OutOfMemoryError)
- Leads to context-switch overheads when blocked worker threads get unblocked

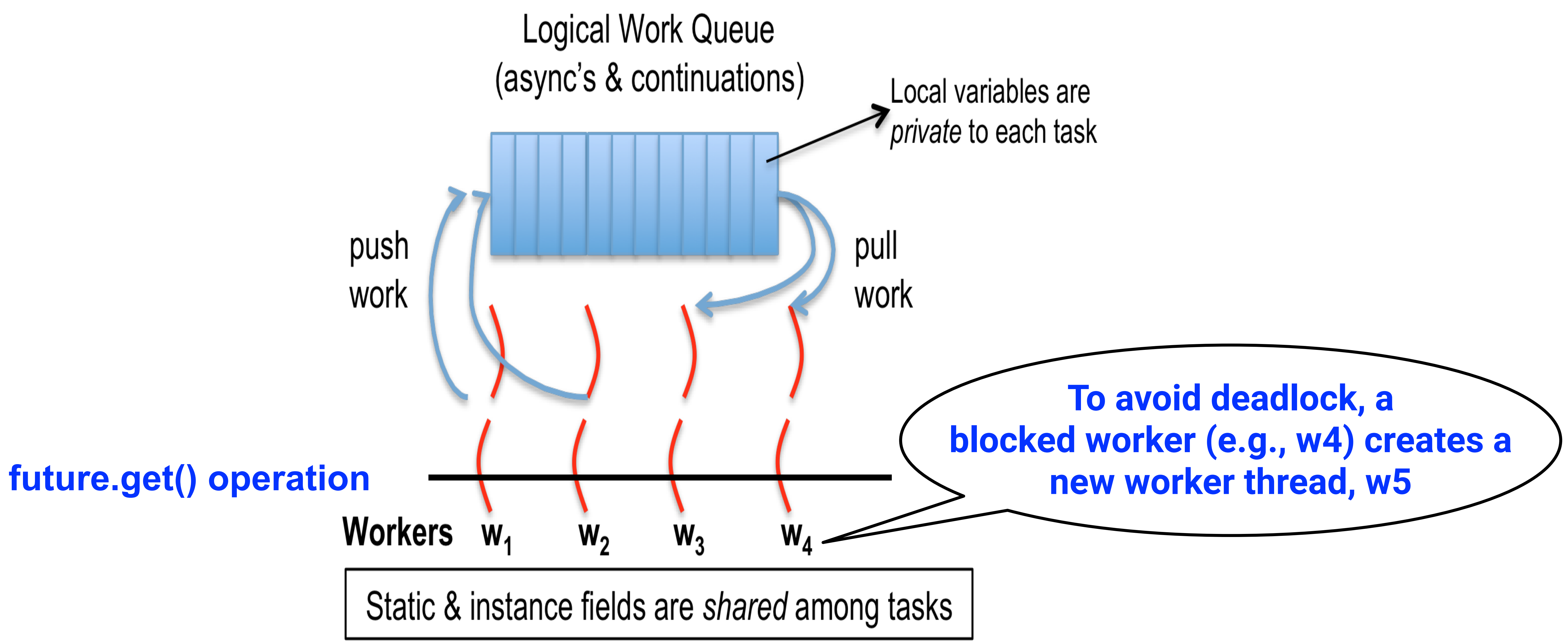
source: http://www.deviantart.com/art/Randomness-5-90424754

Blocking Runtime (contd)



- Assume that there are five tasks (A1 … A5)
- Q: What happens if four tasks (say, A1 … A4) executing on workers w1 … w4 all wait on the same future that's computed by A5?

Blocking Runtime (contd)

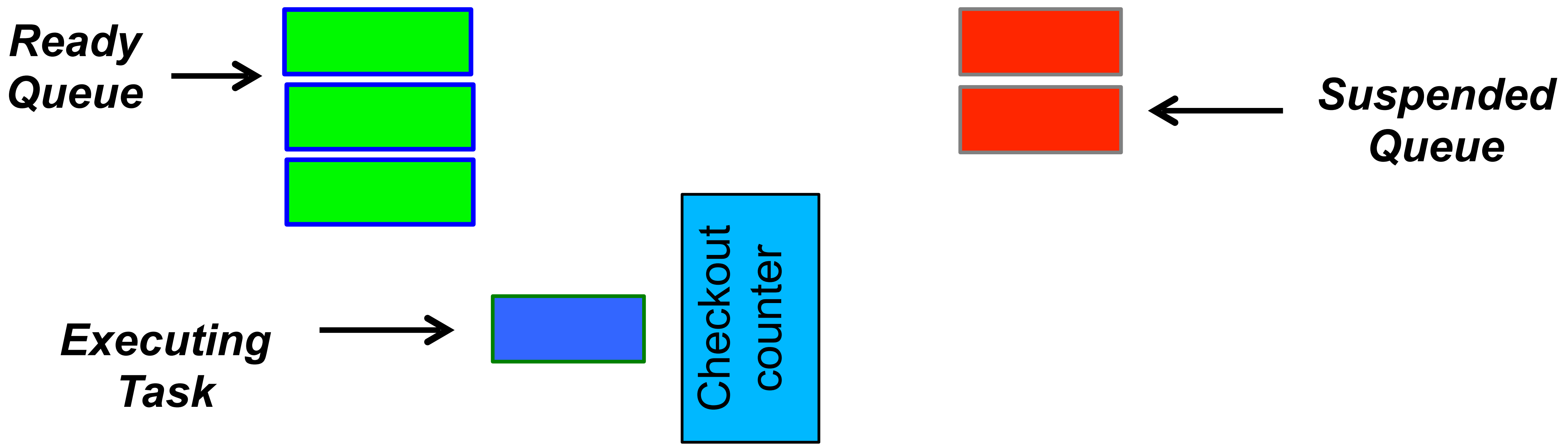


- Assume that there are five tasks (A1 … A5)
- Q: What happens if four tasks (say, A1 … A4) executing on workers w1 … w4 all wait on the same future that's computed by A5?
- A: Deadlock! (All four tasks will wait for task A5 to compute the future.)
- Blocking Runtime's solution to avoid deadlock: keep task blocked on worker thread, and create a new worker thread when task blocks

Blocking Runtime (contd)

- Examples of blocking operations
  - End of finish
  - Future get
  - Barrier next
- Approach: Block underlying worker thread when task performs a blocking operation, and launch an additional worker thread
- Too many blocking operations can result in exceptions and/or poor performance, e.g.,
  - `java.lang.IllegalStateException: Error in executing blocked code! [89 blocked threads]`
- Maximum number of worker threads can be configured if needed
  - `HjSystemProperty.maxThreads.set(100);`

Approach 2: Suspend task continuations at blocking points (as in HJ-Lib's Cooperative Runtime)



- Upon a blocking operation, the currently executing tasks suspends itself and yields control back to the worker
- Task's continuation is stored in the suspended queue and added back into the ready queue when it is unblocked
- Pro: No overhead of creating additional worker threads
- Con: Need to create continuations (enabled by -javaagent option)

Continuations

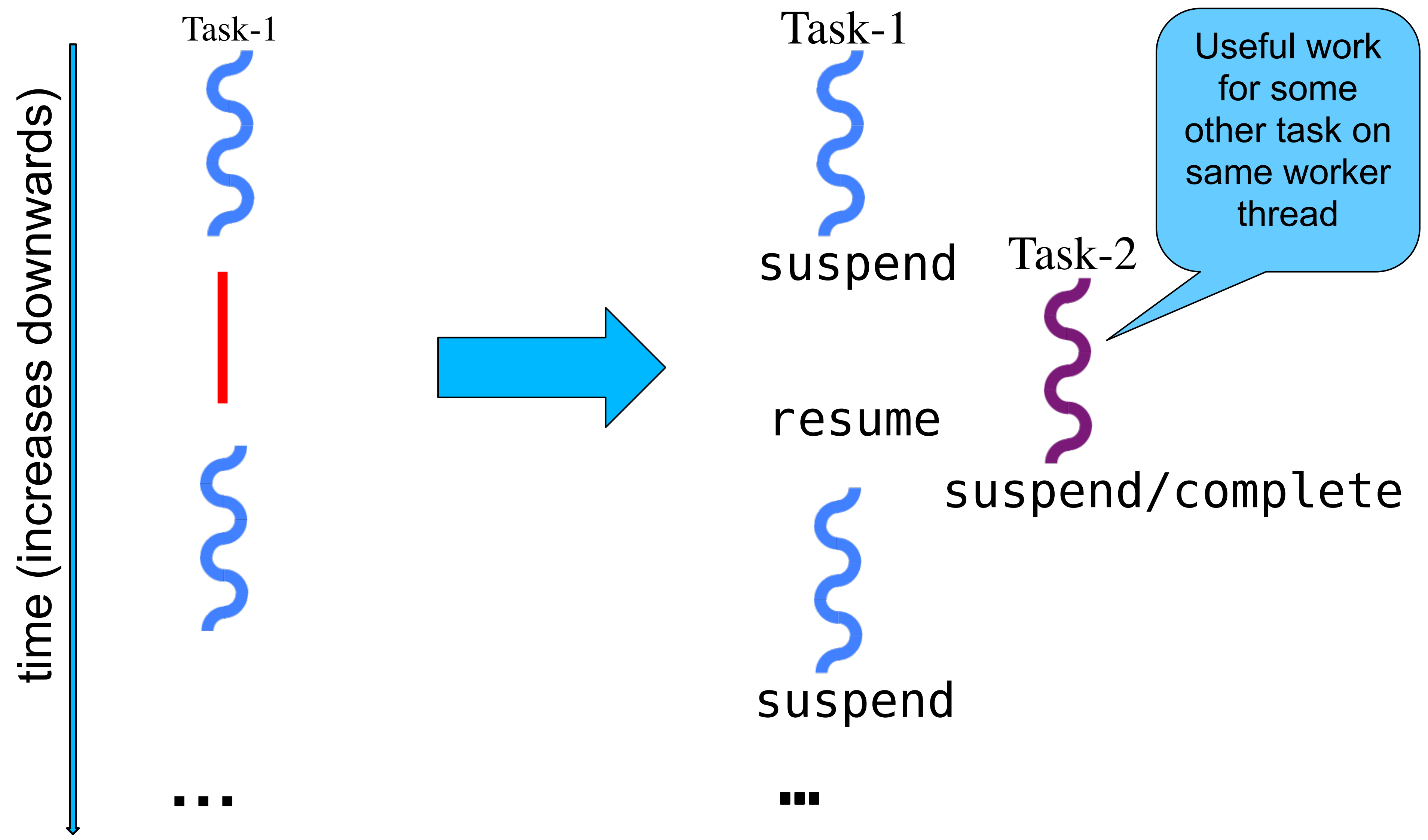
- A continuation can be a point immediately following a blocking operation, such as an end-finish, future get(), barrier/phaser next(), etc.

- Continuations are also referred to as task-switching points
  - Program points at which a worker may switch execution between different tasks (depends on scheduling policy)

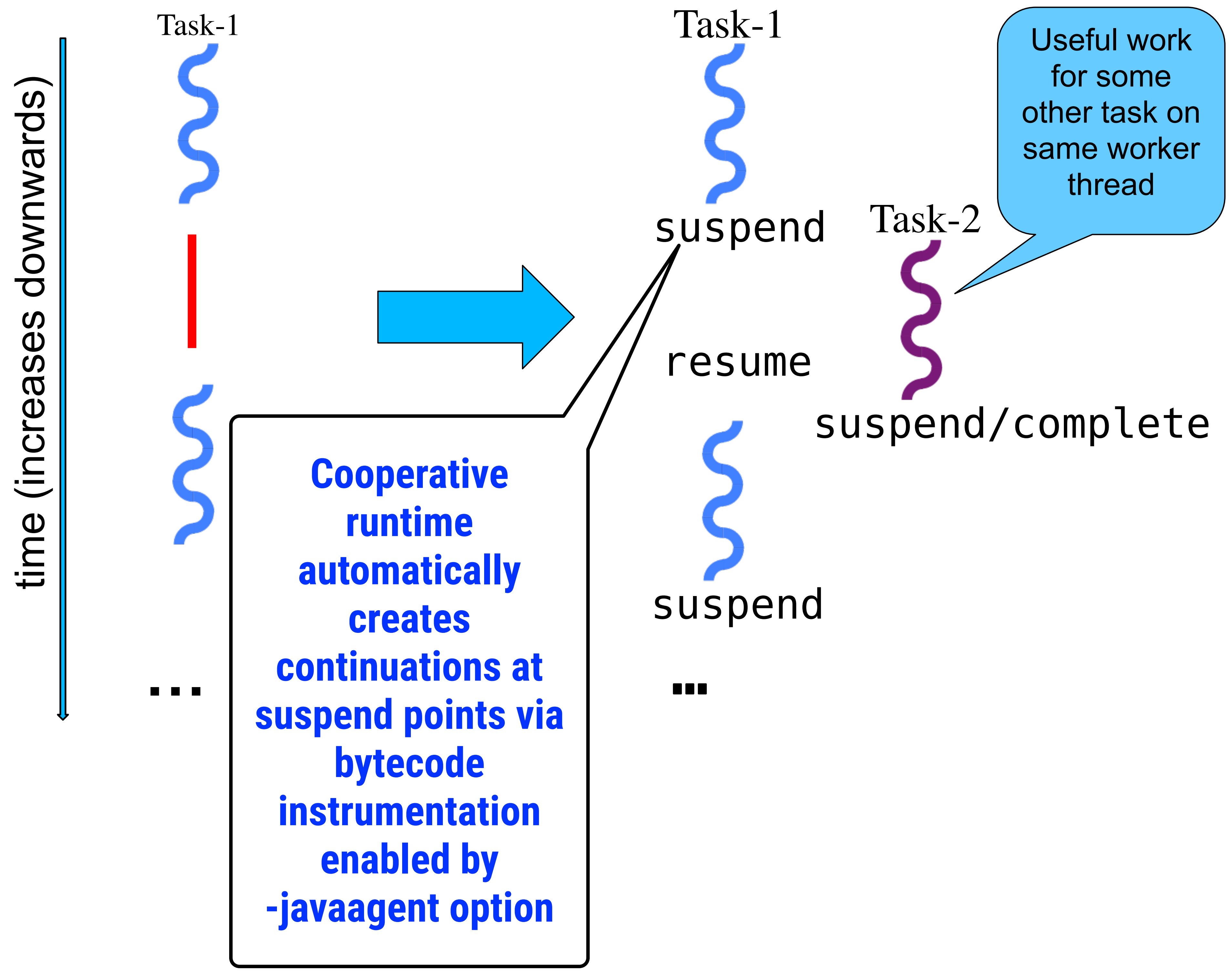
```
finish { // F1
  async A1;
  finish { // F2
    async A3;
    async A4;
  }
  S5;
}
```

Continuations (the points after line 6 `}` and line 8 `}`)

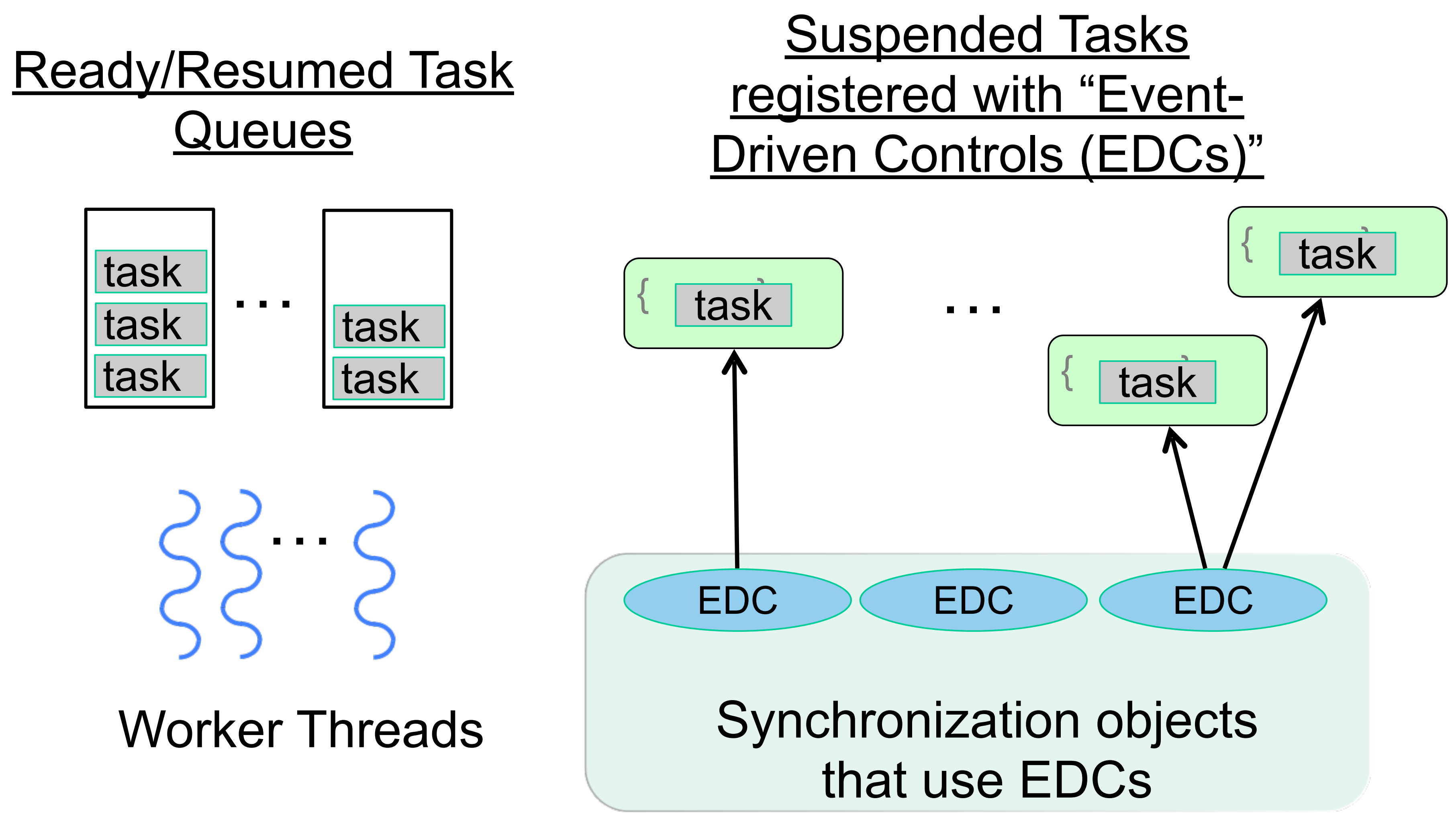
Cooperative Scheduling (view from a single worker)



Cooperative Scheduling (view from a single worker)



HJ-lib's Cooperative Runtime (contd)



Any operation that contributes to unblocking a task can be viewed as an event e.g., task termination in finish, return from a future, signal on barrier, put on a data-driven-future, …

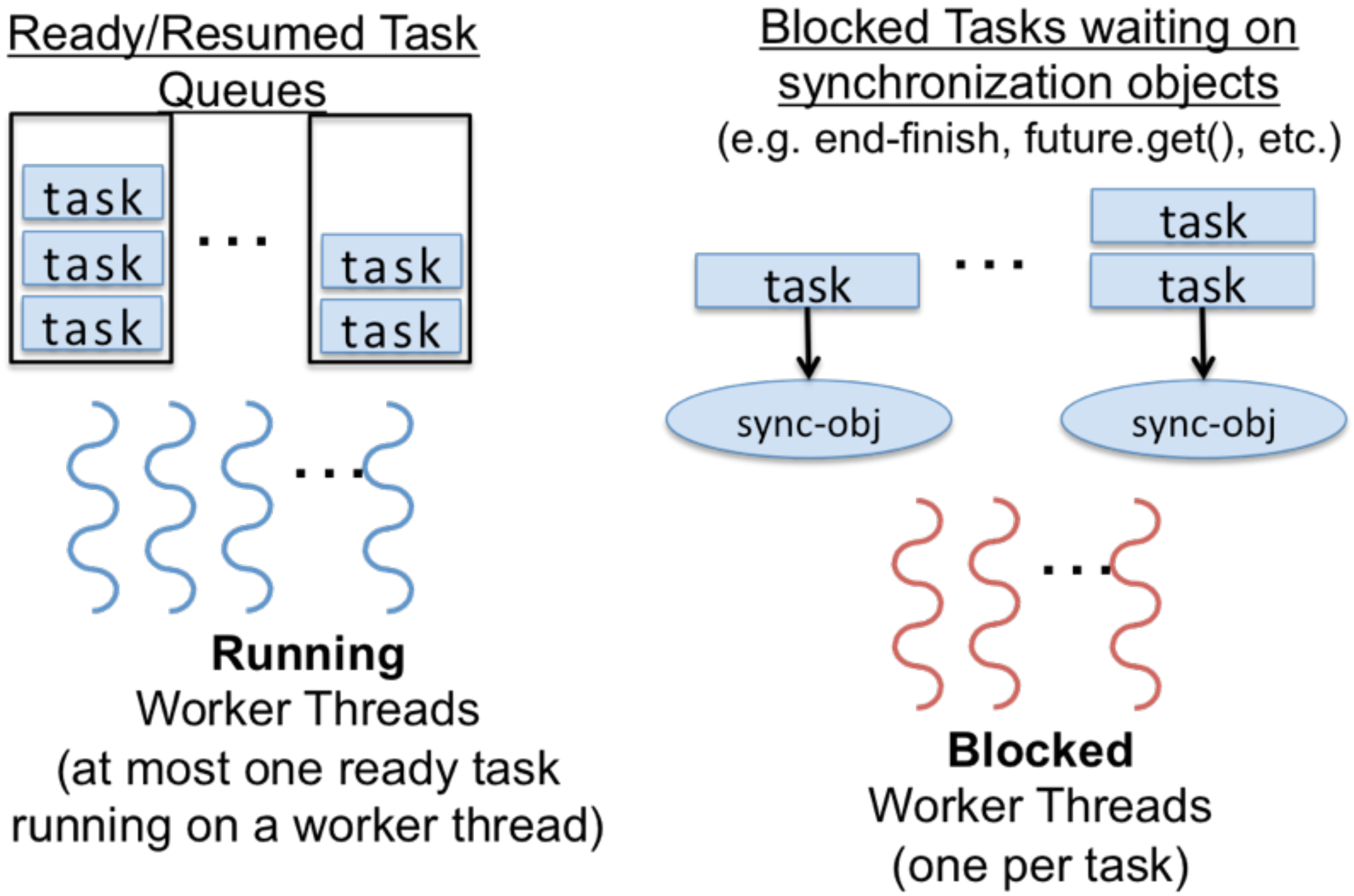
Why are Data-Driven Tasks (DDTs) more efficient than Futures?

- Consumer task blocks on get() for each future that it reads, whereas asyncAwait does not start execution until all Data-Driven Futures (DDFs) are available
  - An "asyncAwait" call does not block the worker, unlike a future.get()
  - No need to create a continuation for asyncAwait; a data-driven task is directly placed on the Suspended queue by default

- Therefore, DDTs can be executed on a Blocking Runtime without the need to create additional worker threads, or on a Cooperative Runtime without the need to create continuations

Abstract vs Real Performance in HJ-Lib

- Abstract Performance
  - Abstract metrics focus on operation counts for WORK and CPL, regardless of actual execution time
- Real Performance
  - HJlib uses ForkJoinPool implementation of Java Executor interface with Blocking or Cooperative Runtime (default)



Summary

- Functional approach is great, but sometimes can lead to performance issues
- Knowing what is happening “under the covers” can help you design better performing algorithms
- Cutoff strategy is a great way to balance parallelism and overhead for recursive task parallelism
- Depending on the runtime, your task parallel program may have some tasks that could block the whole CPU thread
- Processes are more expensive than threads, threads are more expensive than tasks
- In order to deliver performance, most runtimes assume they have a full control of OS threads
    - Don’t mix Java parallel Streams with HJLib constructs
    - Don’t mix Java threads with HJLib tasks and/or Java parallel Streams
    - An HJ runtime instance inside of its own Java thread is usually OK
    - A Java parallel Stream computation inside an HJ task is usually OK