

# COMP 322: Fundamentals of Parallel Programming

## Lecture 18: Abstract vs Real Performance - An “under the hood” look at HJlib

Mack Joyner  
mjoyner@rice.edu

<http://comp322.rice.edu>



# Worksheet #17:

## Critical Path Length for Computation with Signal Statement

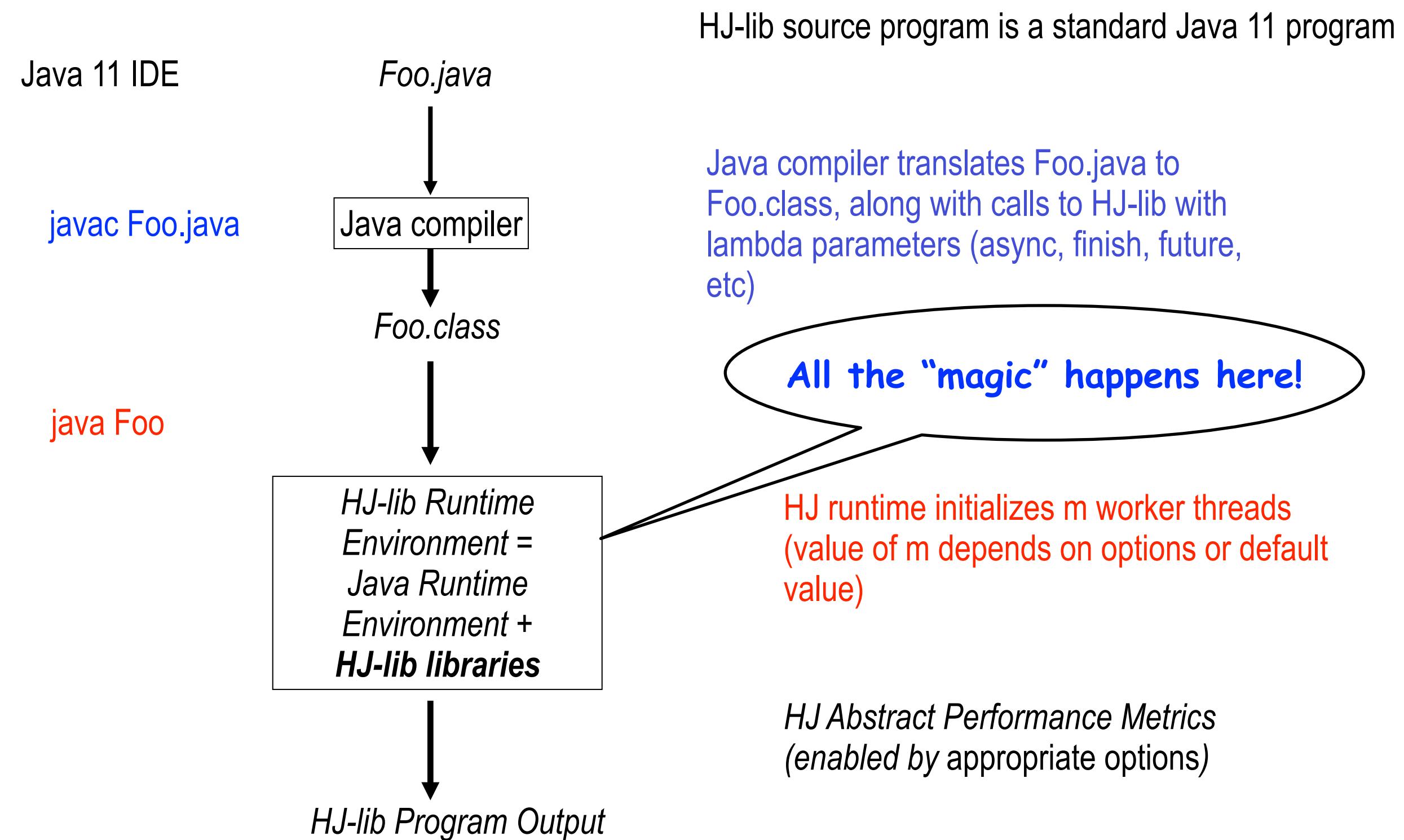
Compute the WORK and CPL values for the program shown below. How would they be different if the `signal()` statement was removed? (Hint: draw a computation graph as in slide 11)

**WORK = 204, CPL = 102. If the `signal()` is removed, CPL = 202.**

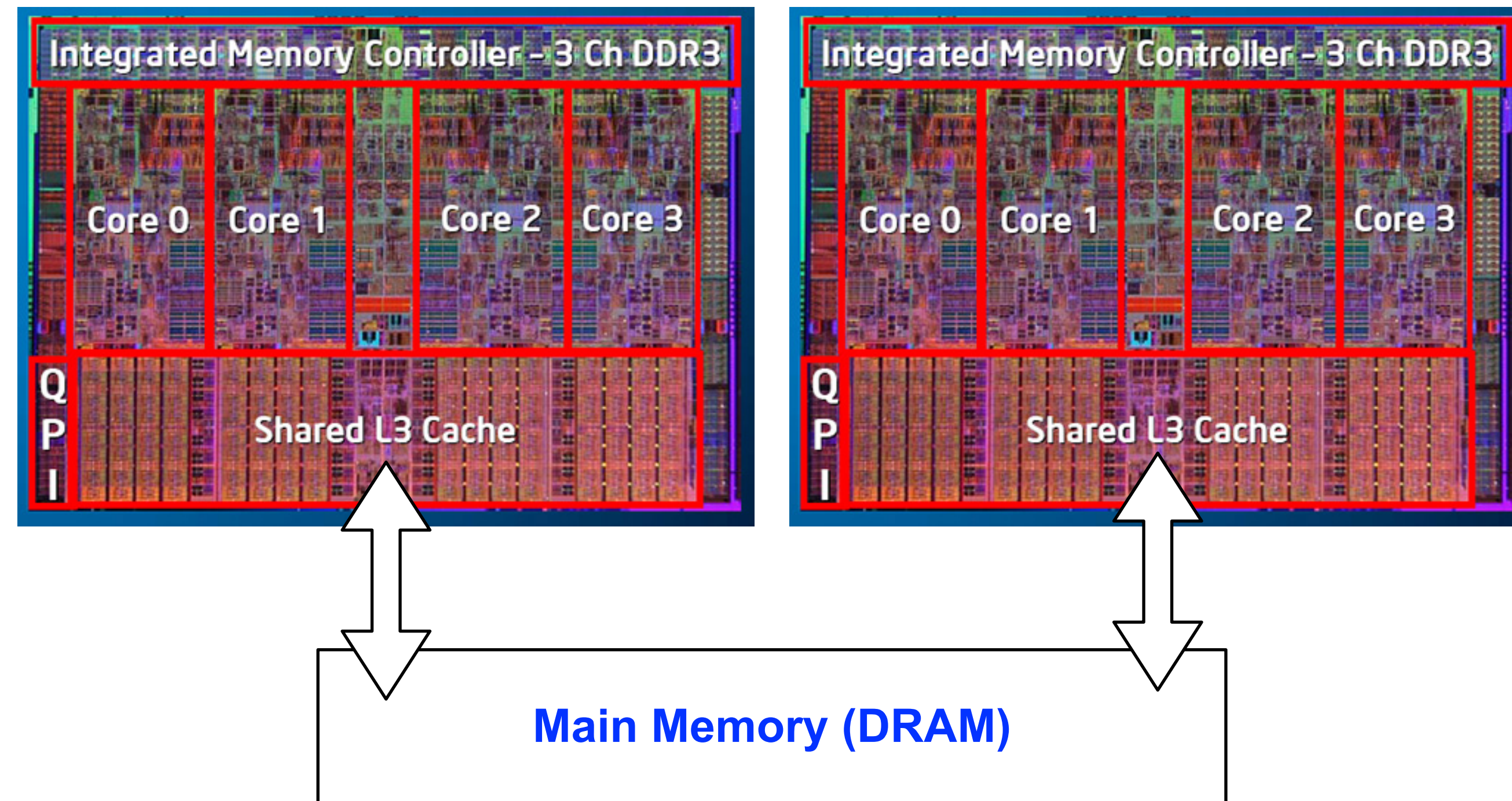
```
1. finish(() -> {
2.   final HjPhaser ph = newPhaser(SIG_WAIT);
3.   asyncPhased(ph.inMode(SIG_WAIT), () -> { // Task T1
4.     A(0); doWork(1);    // Shared work in phase 0
5.     signal();
6.     B(0); doWork(100); // Local work in phase 0
7.     next(); // Wait for T2 to complete shared work in phase 0
8.     C(0); doWork(1);
9.   });
10.  asyncPhased(ph.inMode(SIG_WAIT), () -> { // Task T2
11.    A(1); doWork(1);    // Shared work in phase 0
12.    next(); // Wait for T1 to complete shared work in phase 0
13.    C(1); doWork(1);
14.    D(1); doWork(100); // Local work in phase 0
15.  });
16.}); // finish
```



# HJ-lib Compilation and Execution Environment



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

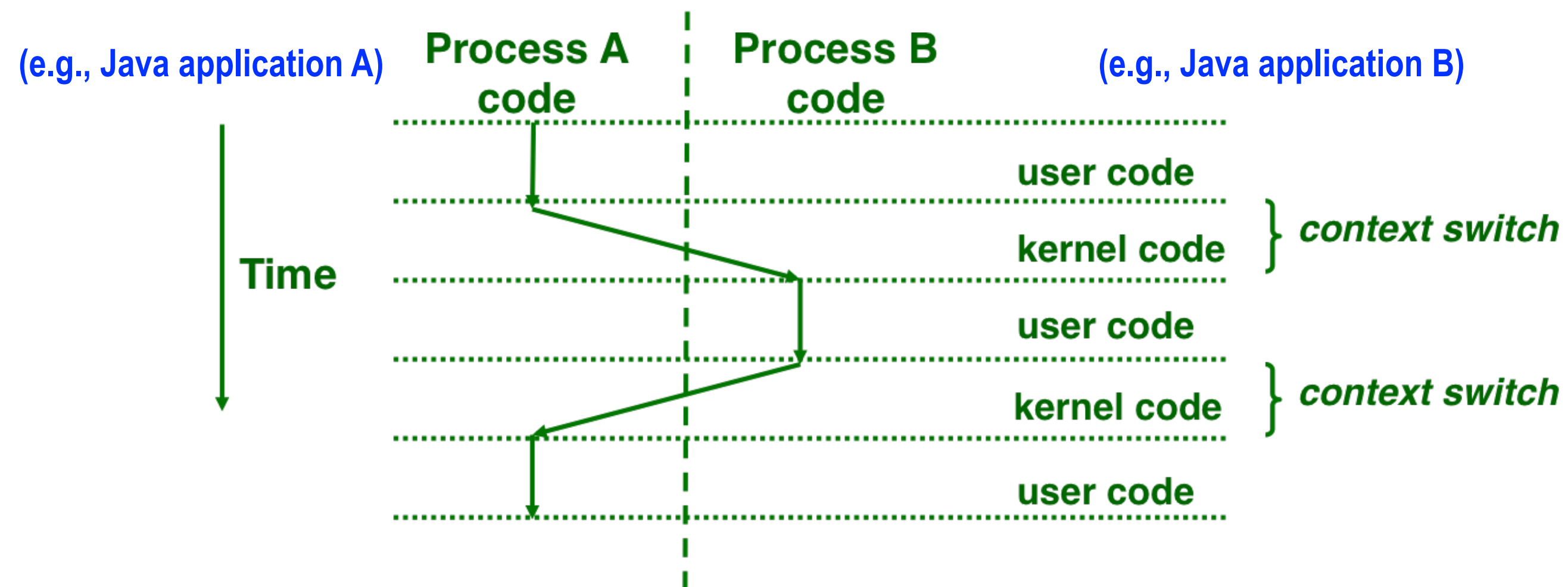


# How does a process run on a single core?

## 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



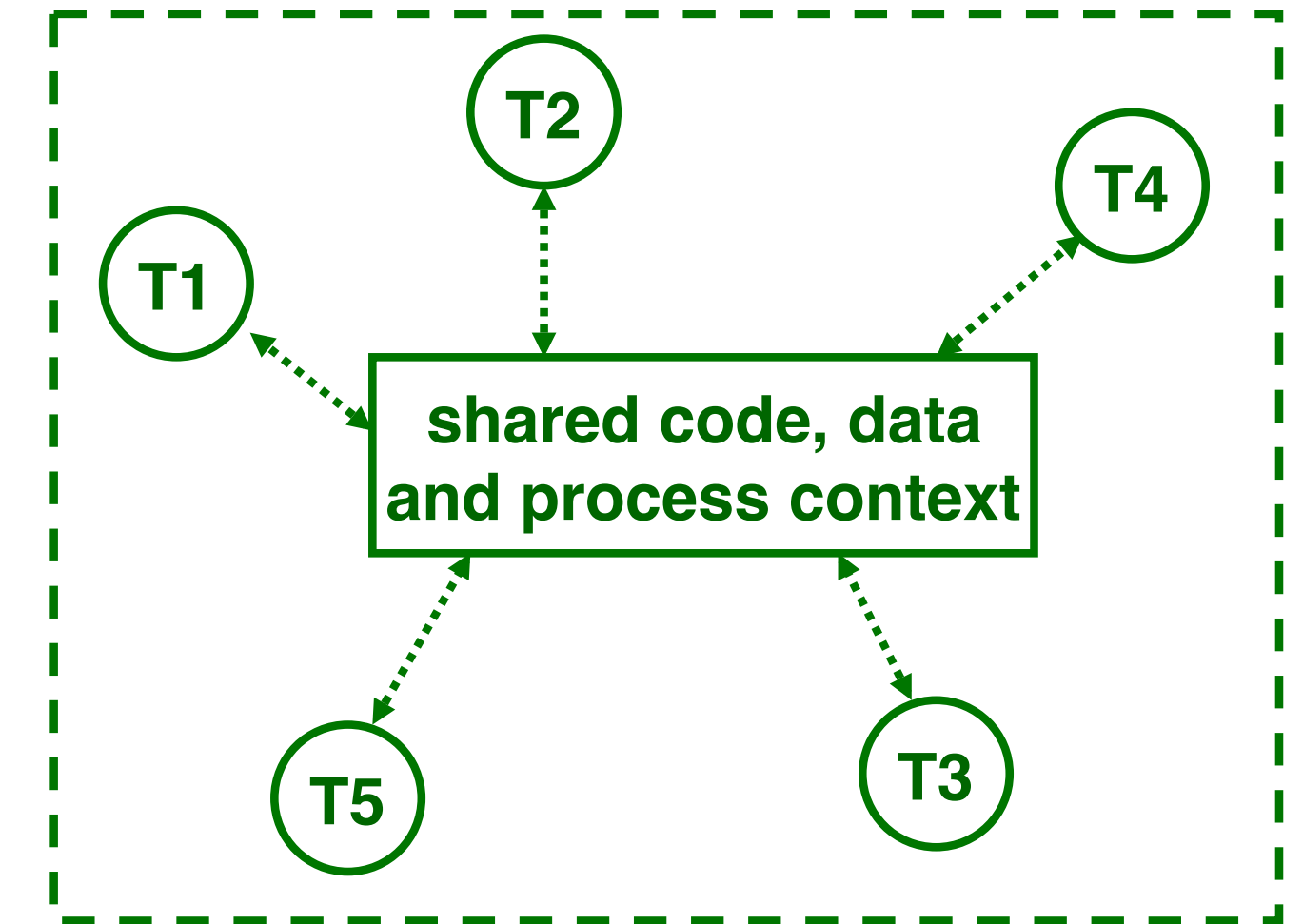
Context switches between two processes can be very expensive!

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



# What happens when executing 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

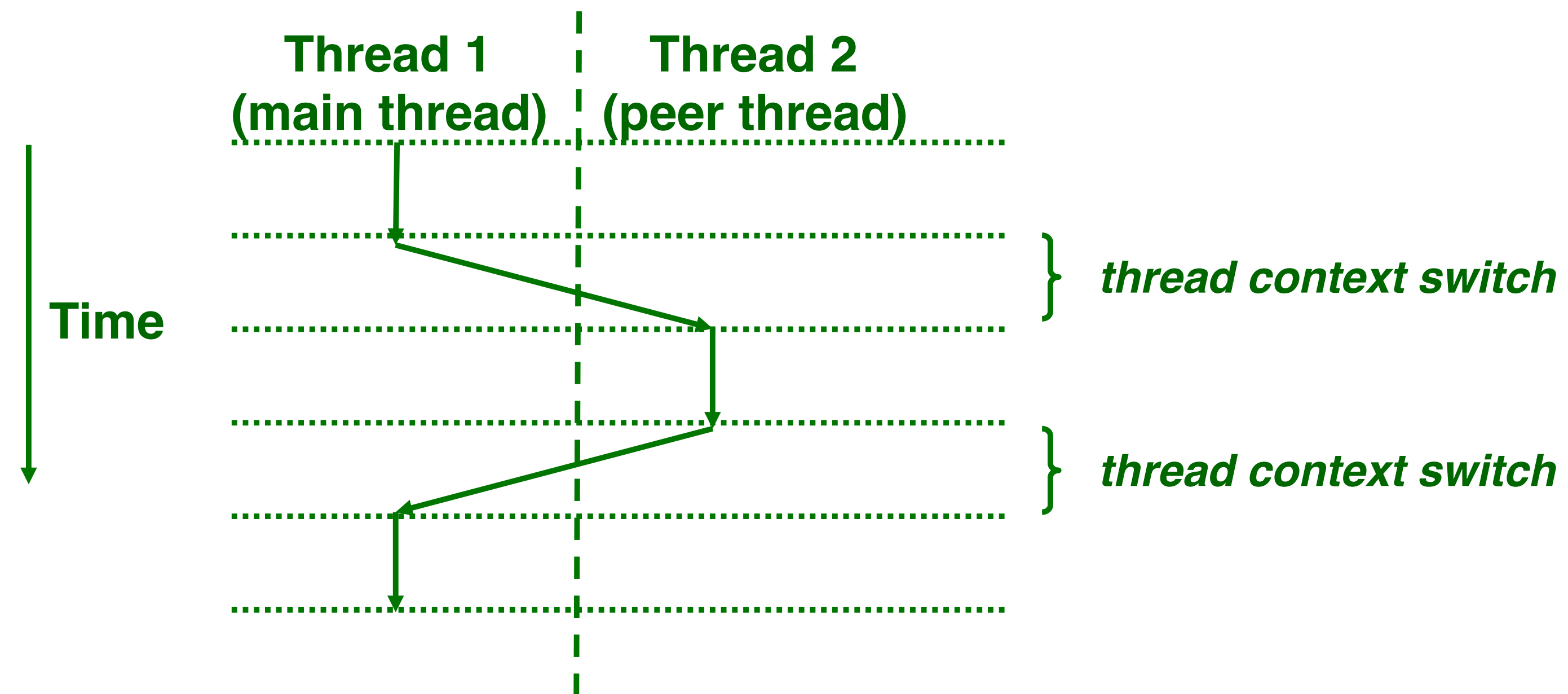


Java application with five threads —  
T1, T2, T3, T4, T5 — all of which can  
access a common set of shared objects

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



# Thread-level Context Switching on the same processor core



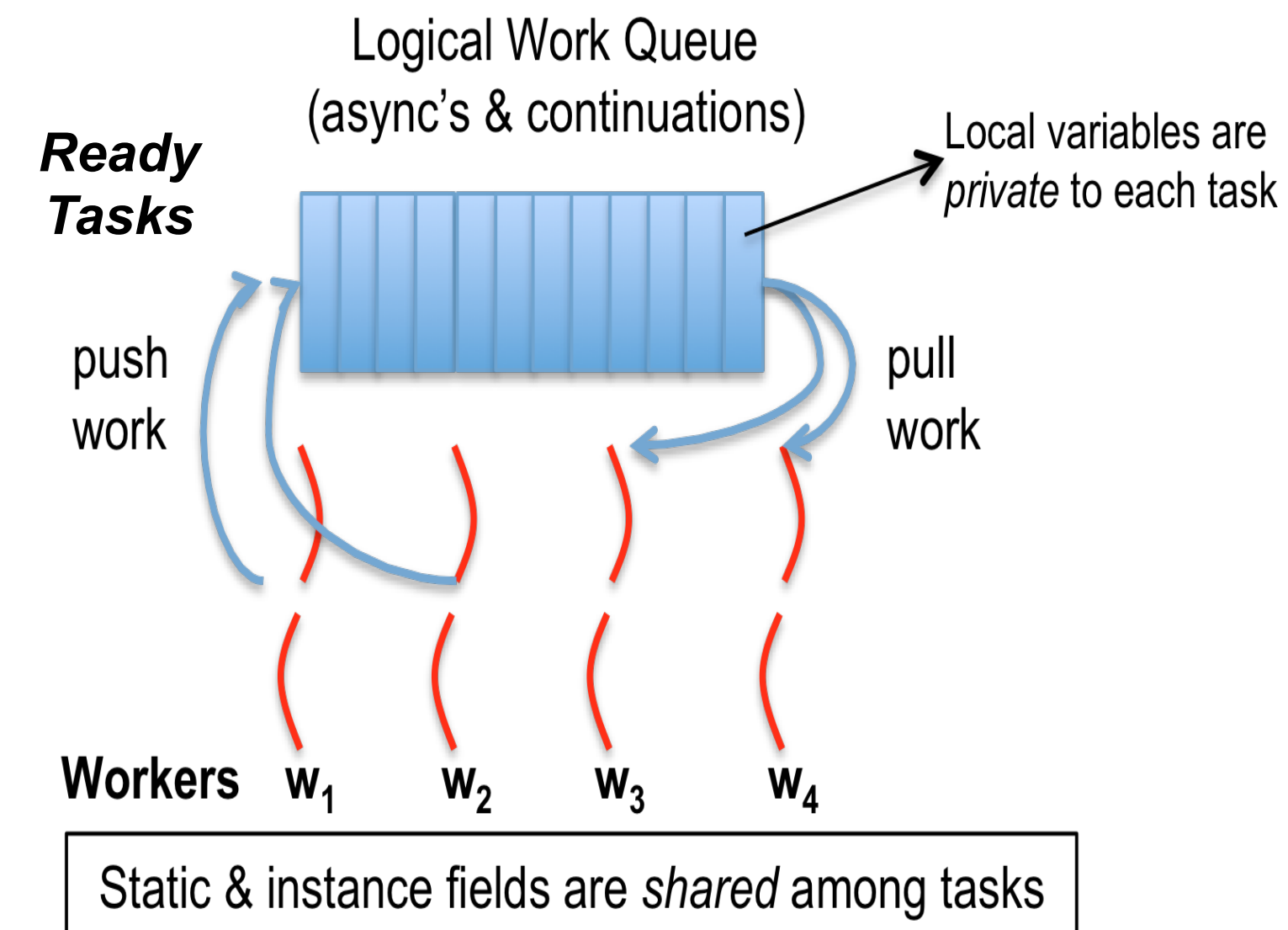
- Thread context switch is cheaper than a process context switch, but is still expensive (just not “very” expensive!)
- It would be ideal to just execute one thread per core (or hardware thread context) to avoid context switches

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



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

<b>HJ-Lib Tasks &amp; Continuations</b>
<b>Worker threads</b>
<b>Operating System</b>
<b>Hardware cores</b>



- 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>



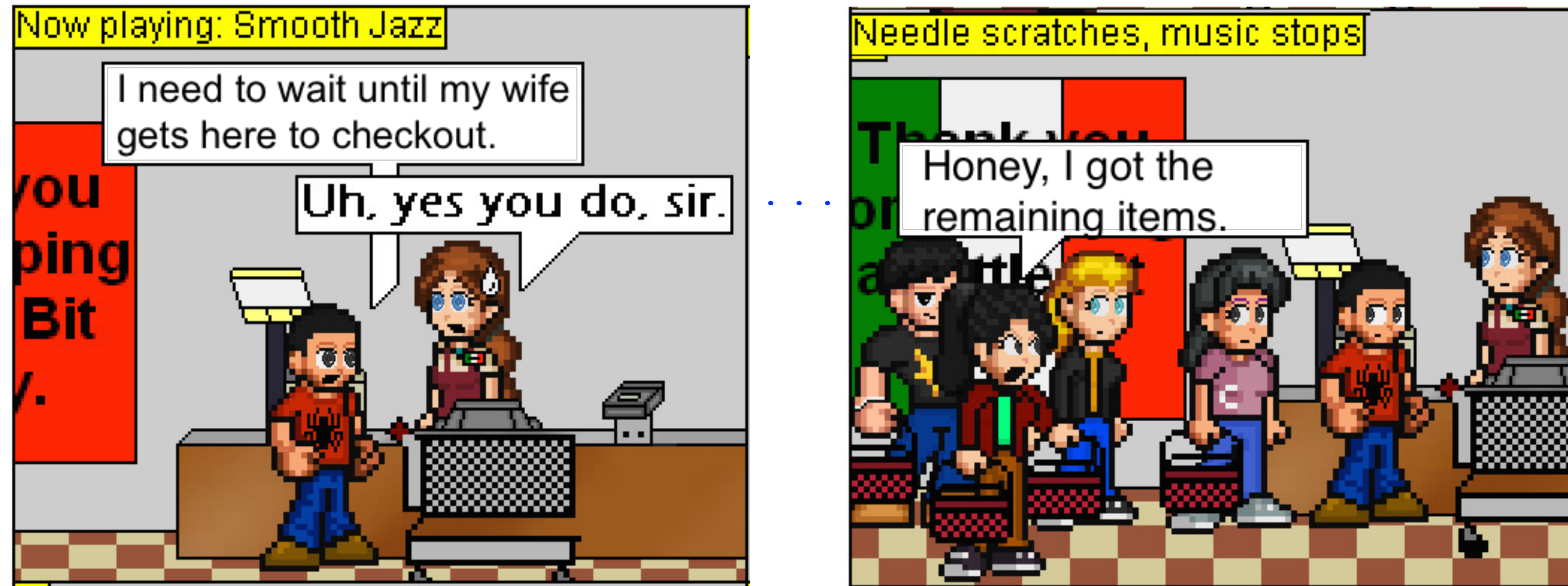
# Task-Parallel Model: Checkout Counter Analogy



- 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>

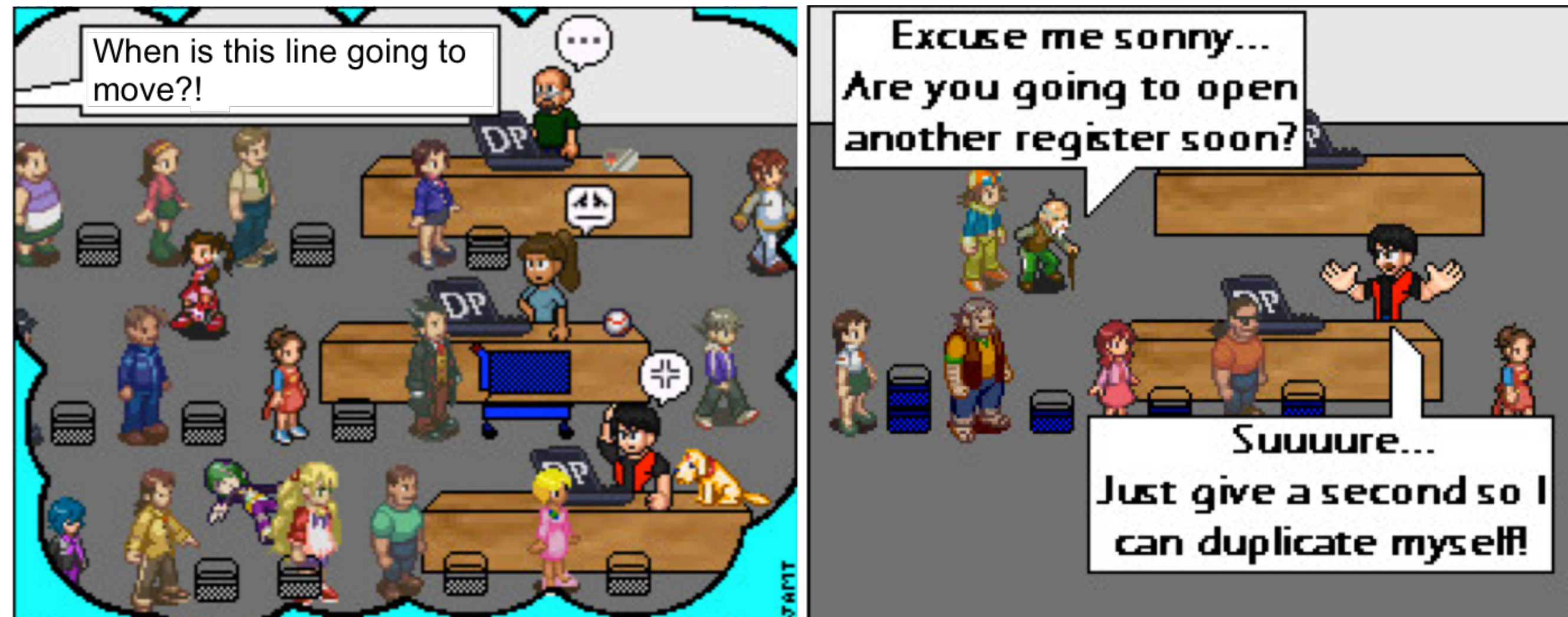
# All is well until a task blocks ...



- 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)

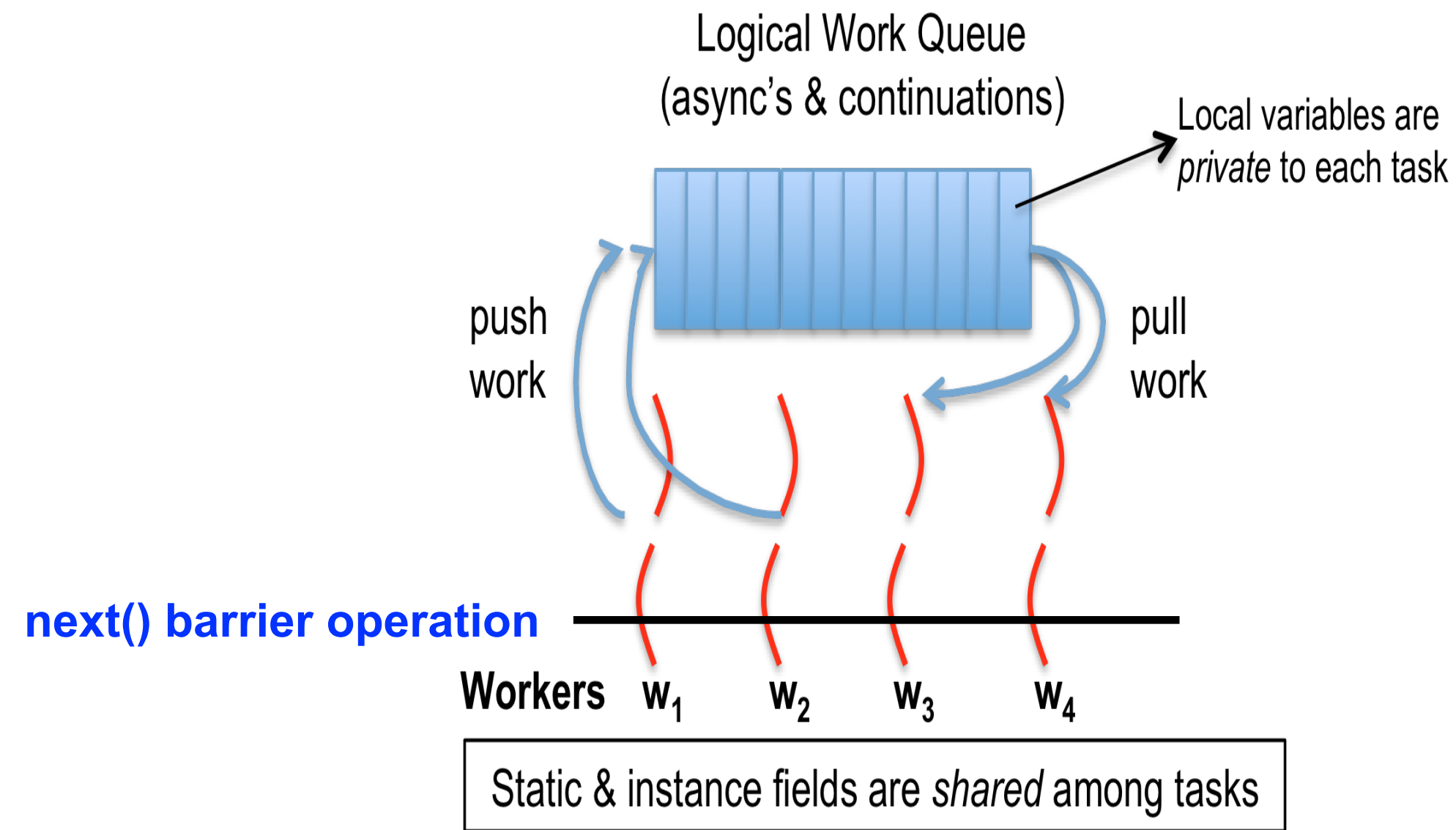


- Creating too many worker threads can exhaust system resources (OutOfMemoryError)
- Leads to context-switch overheads when blocked worker threads get unblocked

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



# Blocking Runtime (contd)

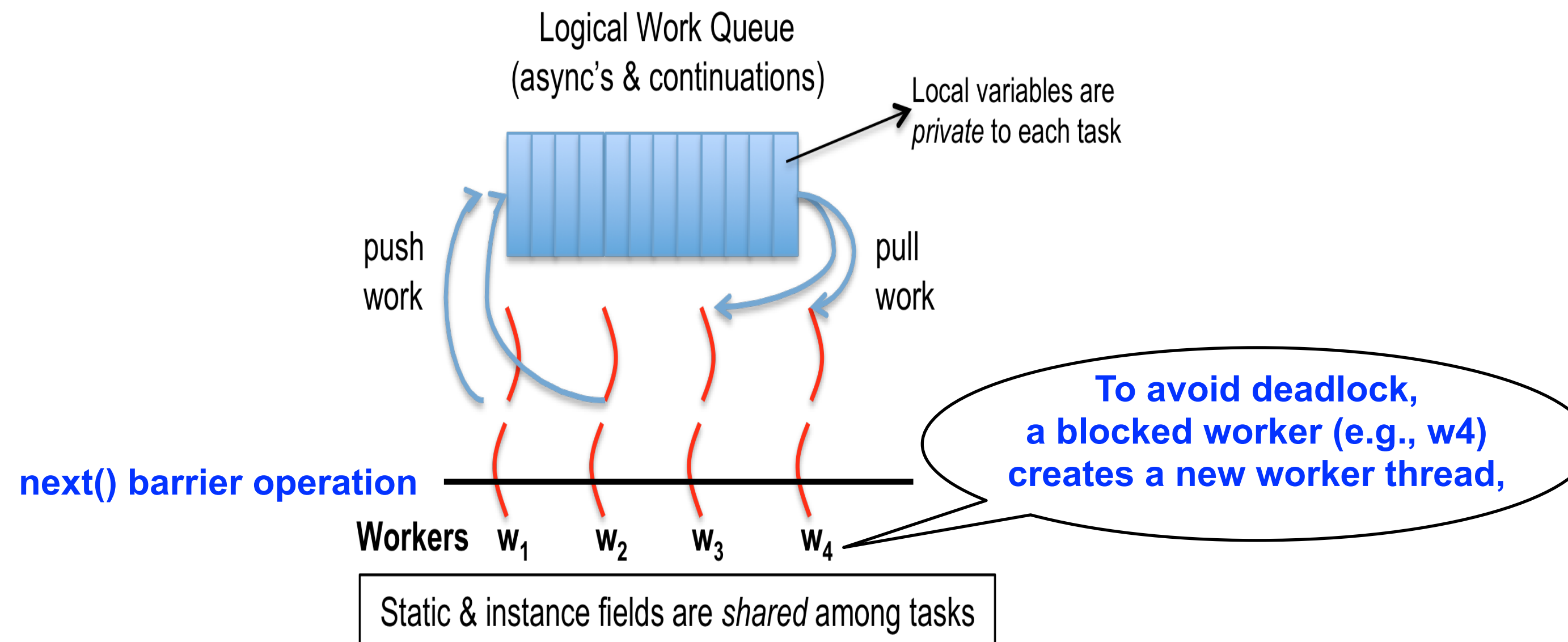


- Assume that five tasks (A1 ... A5) are registered on a barrier
- Q: What happens if four tasks (say, A1 ... A4) executing on workers w1 ... w4 all block at the same barrier?

•



# Blocking Runtime (contd)



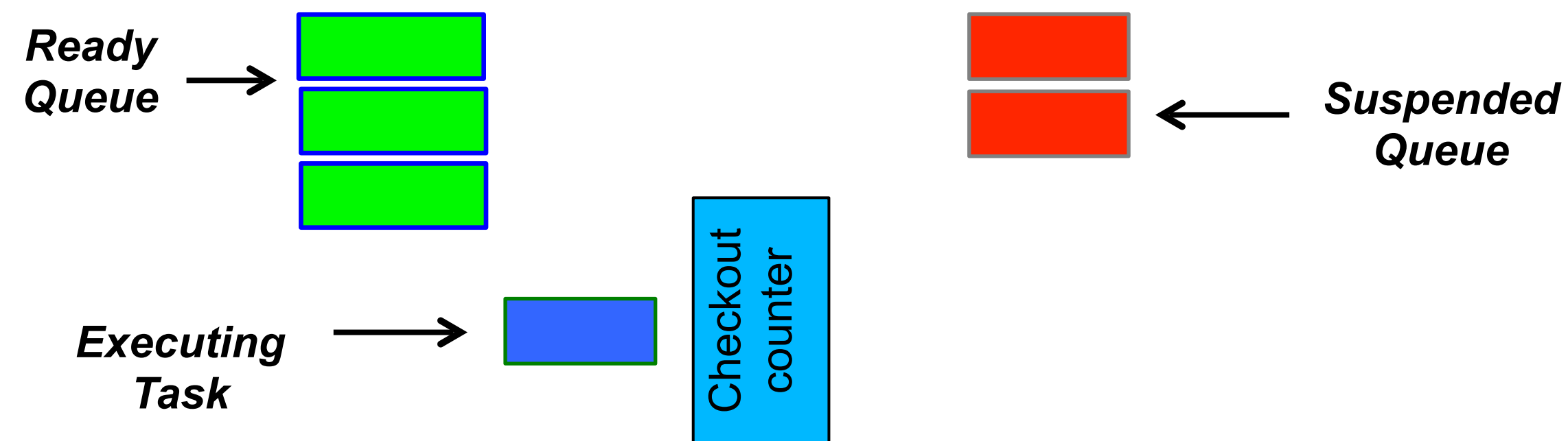
- Assume that five tasks (A1 ... A5) are registered on a barrier
- Q: What happens if four tasks (say, A1 ... A4) executing on workers  $w_1$  ...  $w_4$  all block at the same barrier?
- A: Deadlock! (All four tasks will wait for task A5 to enter the barrier.)
- Blocking Runtime's solution to avoid 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)



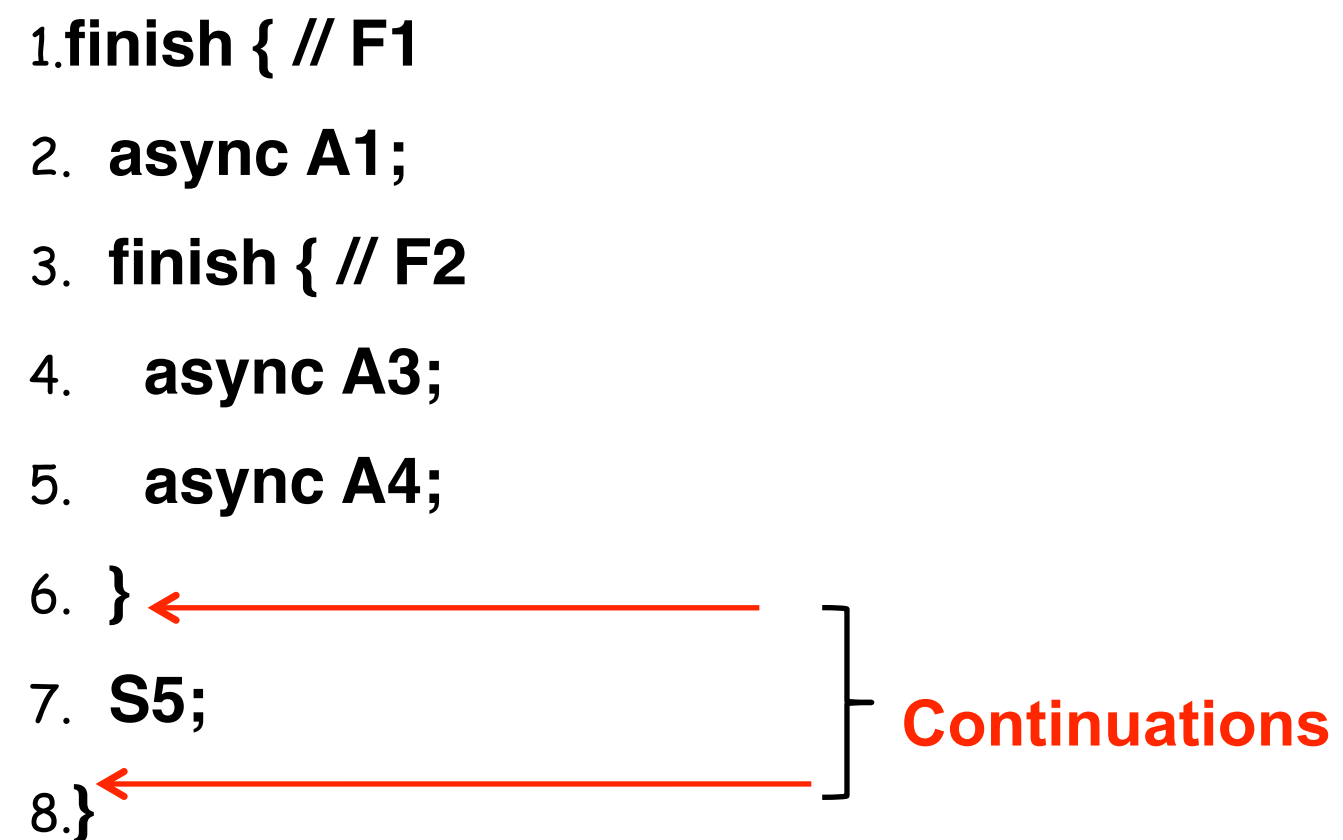


# Continuations

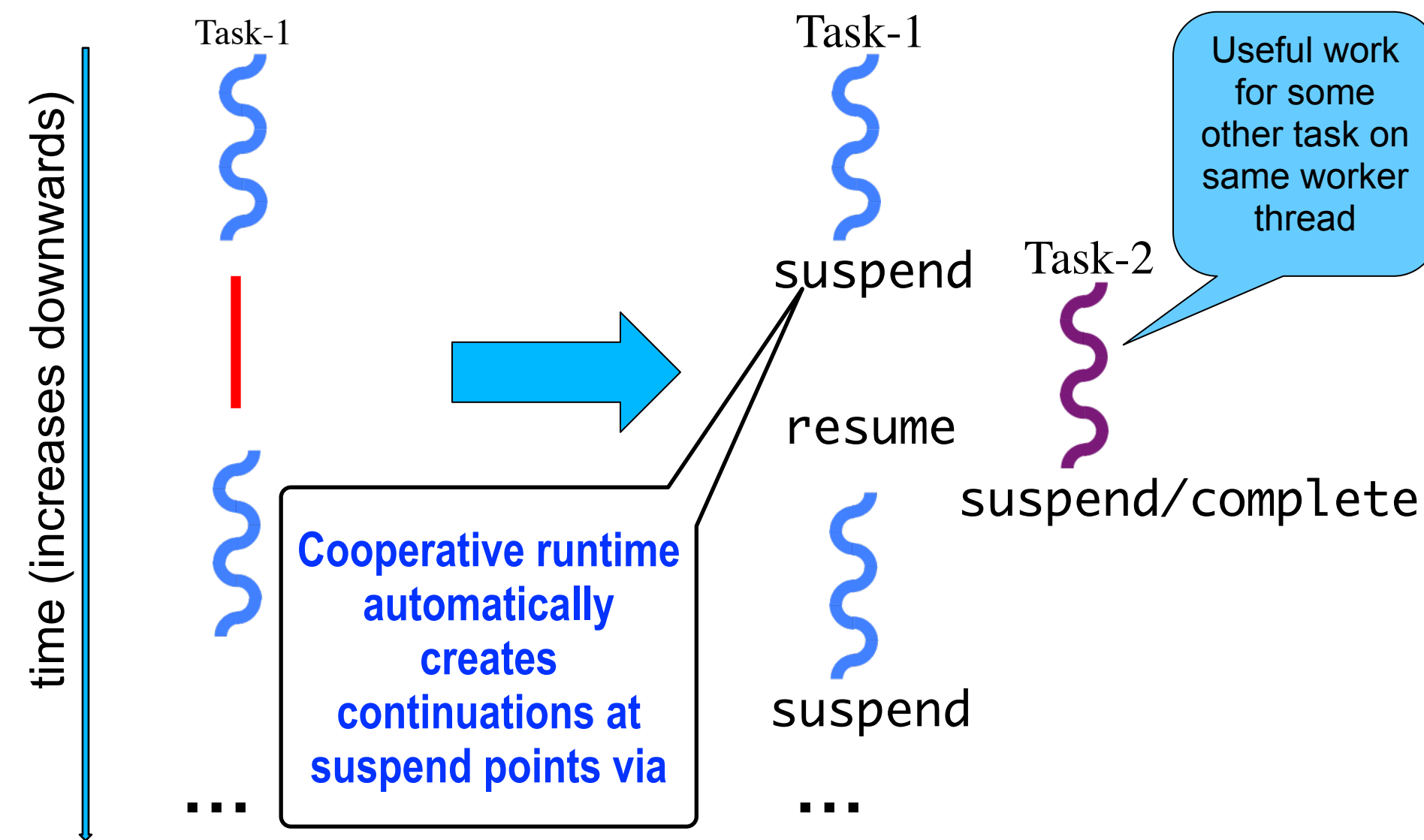
- A continuation can be a point immediately following a *blocking* operation, such as an `end-finish`, `future get()`, `barrier/phaser next()`, etc.
- Continuations are also referred to as task-switching points
  - Program points at which a worker may switch execution between different tasks (depends on scheduling policy)

```
1. finish { // F1
2.  async A1;
3.  finish { // F2
4.    async A3;
5.    async A4;
6.  } ←
7.  S5;
8.} ←
```

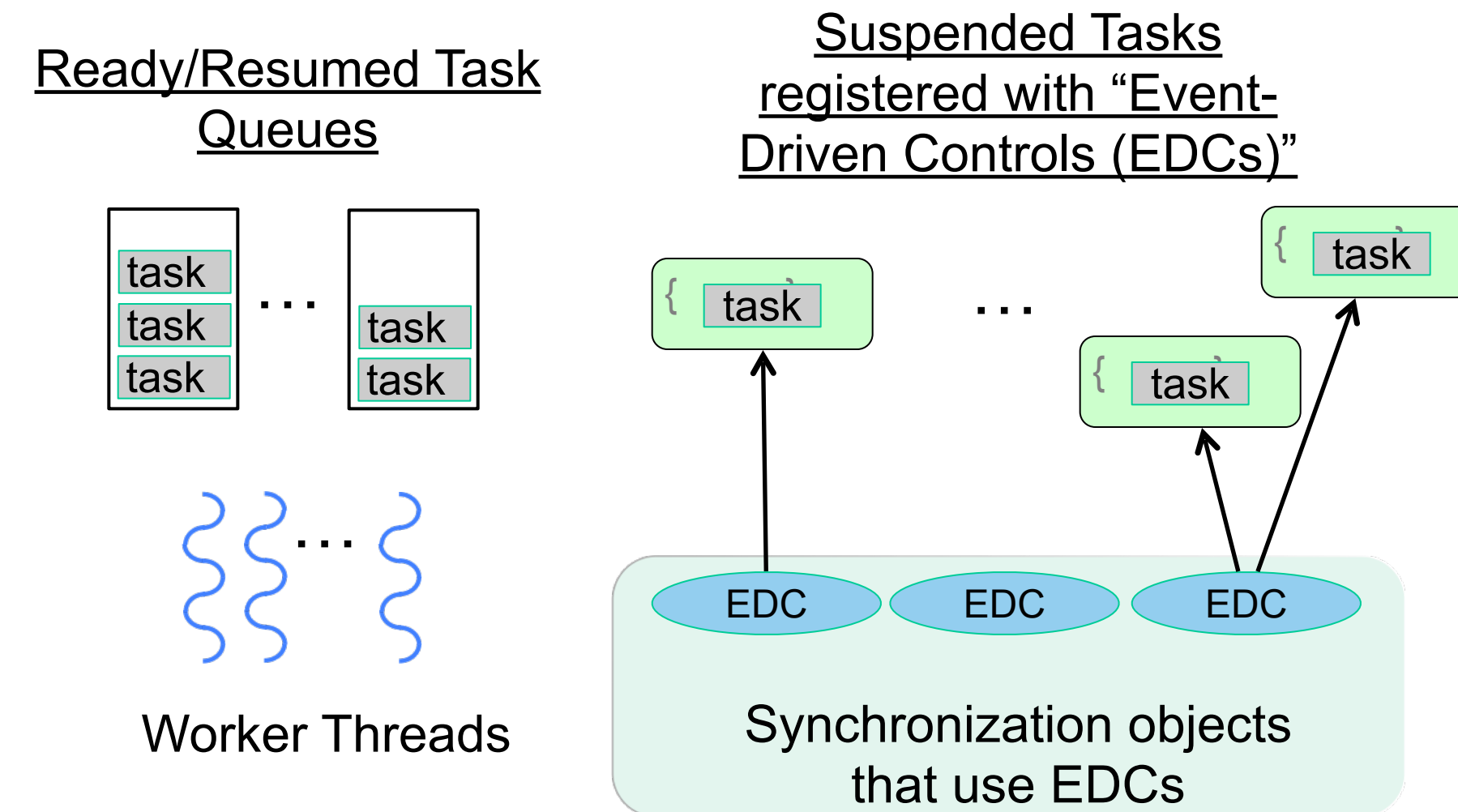
} Continuations



# 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 `async-  
await` does not start execution till all Data-Driven Futures (DDFs) are available
  - An “`asyncAwait`” statement 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



# Summary: Abstract vs Real Performance in HJ-Lib

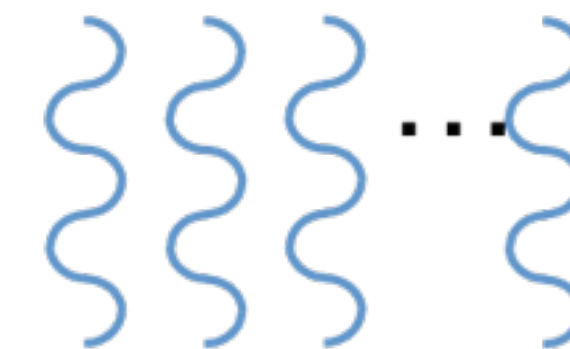
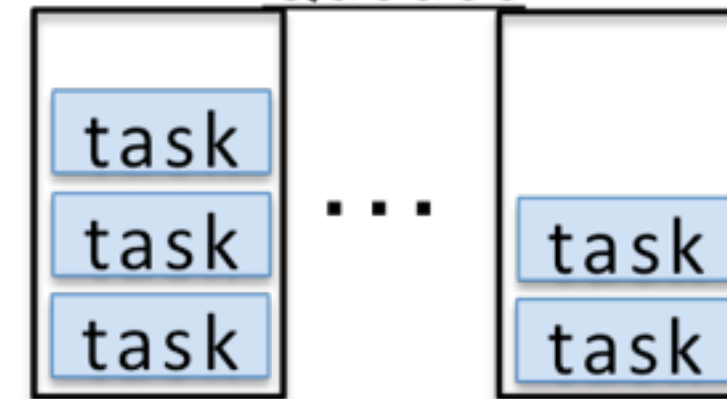
- Abstract Performance

- Abstract metrics focus on operation counts for WORK and CPL, regardless of actual execution time

- Real Performance

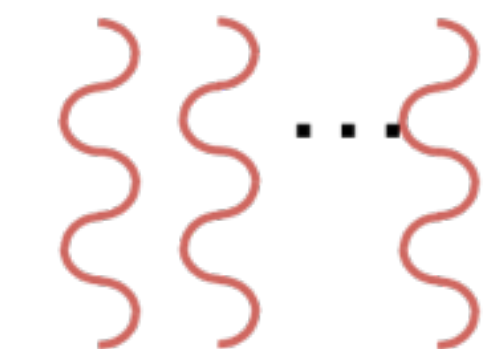
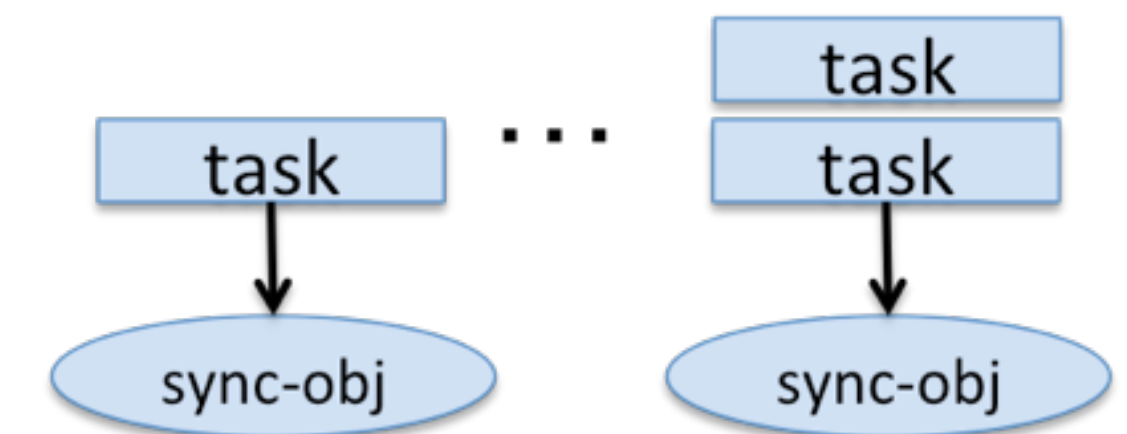
- HJlib uses ForkJoinPool implementation of Java Executor interface with Blocking or Cooperative Runtime (default)

Ready/Resumed Task Queues



**Running**  
Worker Threads  
(at most one ready task running on a worker thread)

Blocked Tasks waiting on synchronization objects  
(e.g. end-finish, future.get(), etc.)



**Blocked**  
Worker Threads  
(one per task)



# Announcements & Reminders

---

- Quiz #3 is due **today** at 11:59pm
- No lab this week
- Quiz #4 is now due Sunday, March 21st at 11:59pm
- HW #3 CP 1 is due Wednesday, Mar 24th at 11:59pm
- Watch the topic 5.1, 5.2, 5.6 videos for the next lecture



# Worksheet #18: Cooperative vs Blocking Runtime Schedulers

Assume that creating an async causes the task to be pushed into the work queue for execution by any available worker thread.

Fill in the following table for the program shown on the right by adding the appropriate number of threads required to execute the program. For the minimum or maximum numbers, your answer must represent a schedule where at some point during the execution all threads are busy executing a task or blocked on some synchronization constraint.

	Minimum number of threads	Maximum number of threads
Cooperative Runtime		
Blocking Runtime		

```

10. finish {
11.   async { S1; }
12.   finish {
13.     async {
14.       finish {
15.         async { S2; }
16.         S3;
17.       }
18.       S4;
19.     }
20.     async {
21.       async { S5; }
22.       S6;
23.     }
24.     S7;
25.   }
26.   S8;
27. }
    
```

