COMP 322: Fundamentals of Parallel Programming

Lecture 15: Abstract vs. Real Performance — an "under the hood" look at HJlib

Vivek Sarkar, Eric Allen
Department of Computer Science, Rice University

Contact email: vsarkar@rice.edu

https://wiki.rice.edu/confluence/display/PARPROG/COMP322

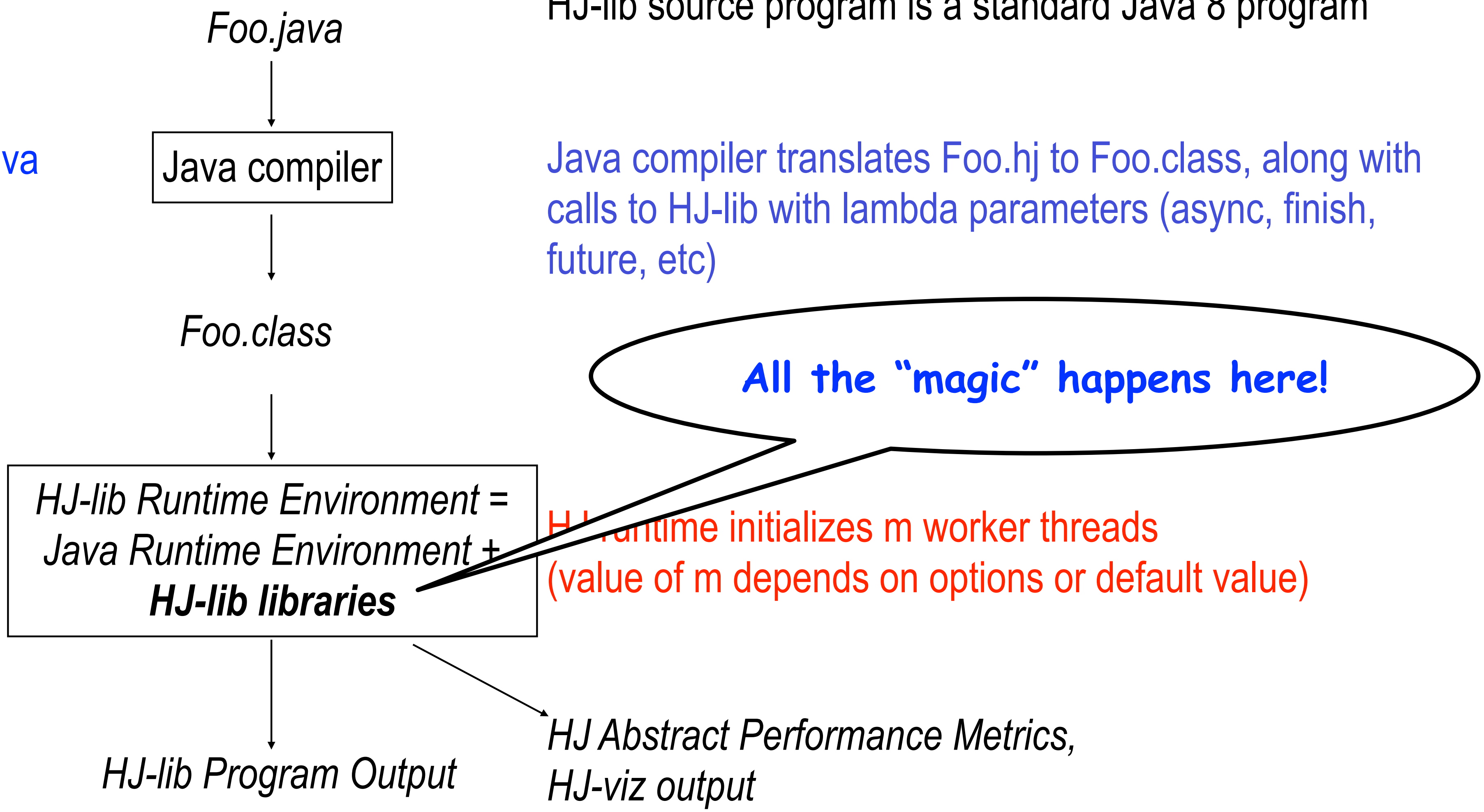
Worksheet #14 solution: Data-Driven Tasks

For the example below, will reordering the five async statements change the meaning of the program (assuming that the semantics of the reader/writer methods depends only on their parameters)? If so, show two orderings that exhibit different behaviors. If not, explain why not. (You can use the space below this slide for your answer.)

```
DataDrivenFuture left = new DataDrivenFuture();
DataDrivenFuture right = new DataDrivenFuture();
finish {
  async await(left) leftReader(left); // Task3
  async await(right) rightReader(right); // Task5
  async await(left,right)
        bothReader(left,right); // Task4
  async left.put(leftWriter()); // Task1
  async right.put(rightWriter());// Task2
}
```

No, reordering consecutive async's will never change the meaning of the program, whether or not the async's have await clauses.

HJ-lib Compilation and Execution Environment



Java 8 IDE

Foo.java

HJ-lib source program is a standard Java 8 program

javac Foo.java

Java compiler

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

Foo.class

All the "magic" happens here!

java Foo

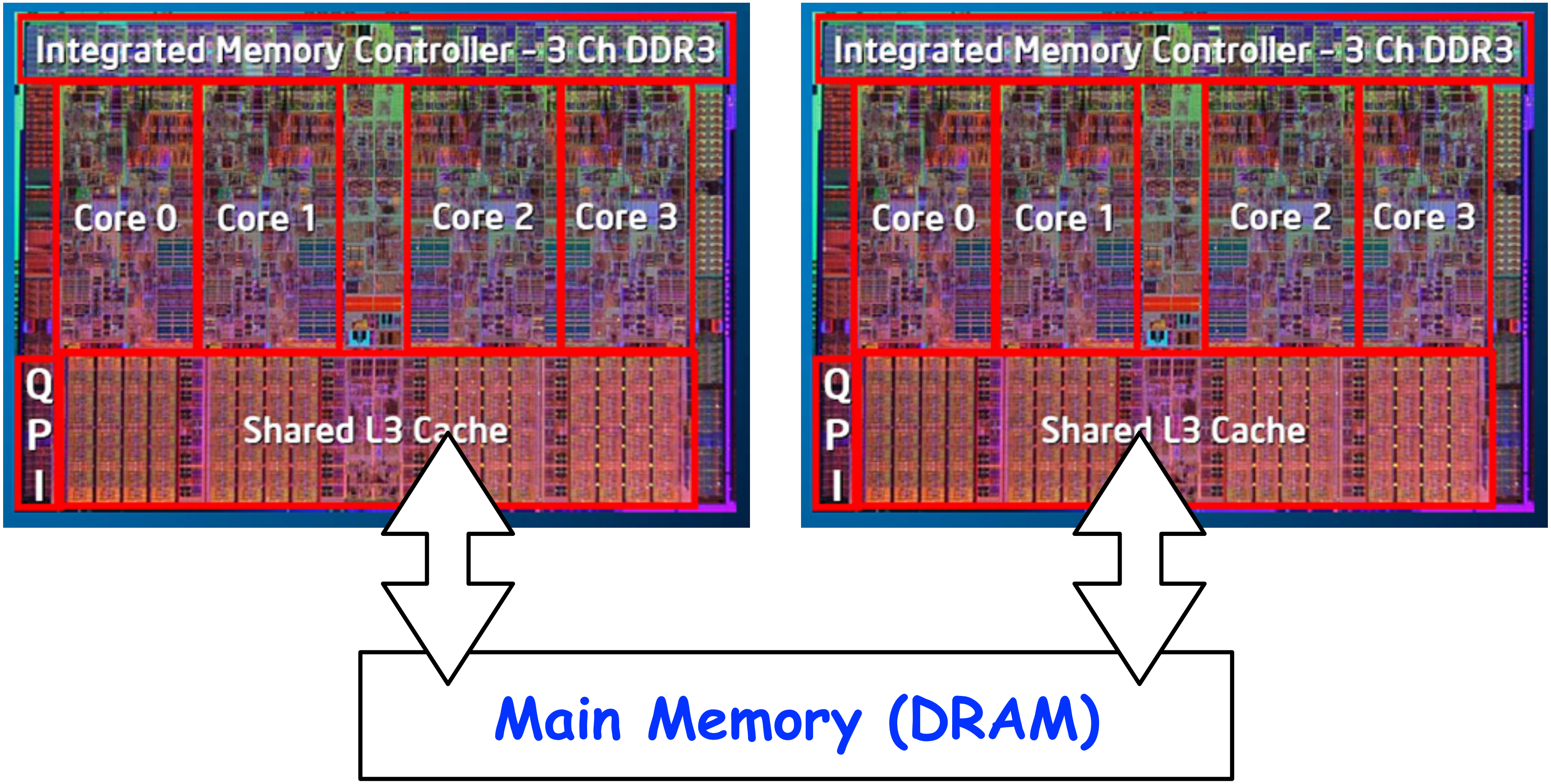
HJ-lib Runtime Environment =
Java Runtime Environment +
HJ-lib libraries

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

HJ-lib Program Output

HJ Abstract Performance Metrics,
HJ-viz output
(all enabled by appropriate options)

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

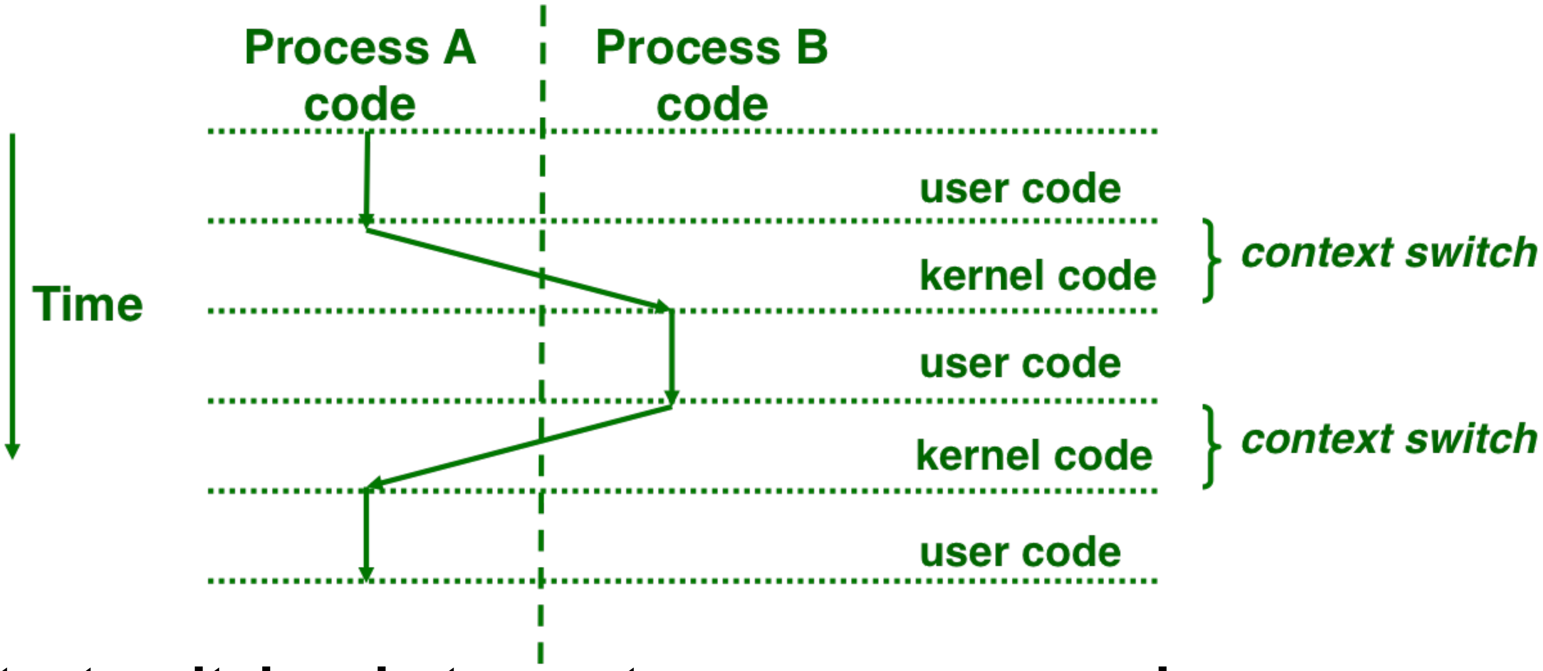


Main Memory (DRAM)

A single STIC compute node contains two quad-core 2.4GHz Intel Xeon (Nahalem) CPUs, for a total of 8 cores/node

Next, 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, Scott Rixner)

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

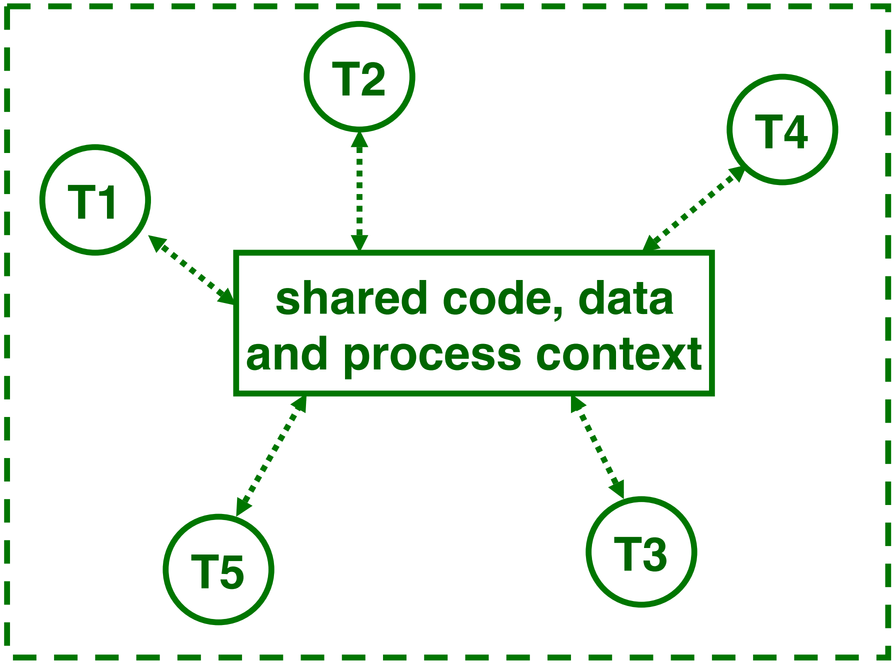
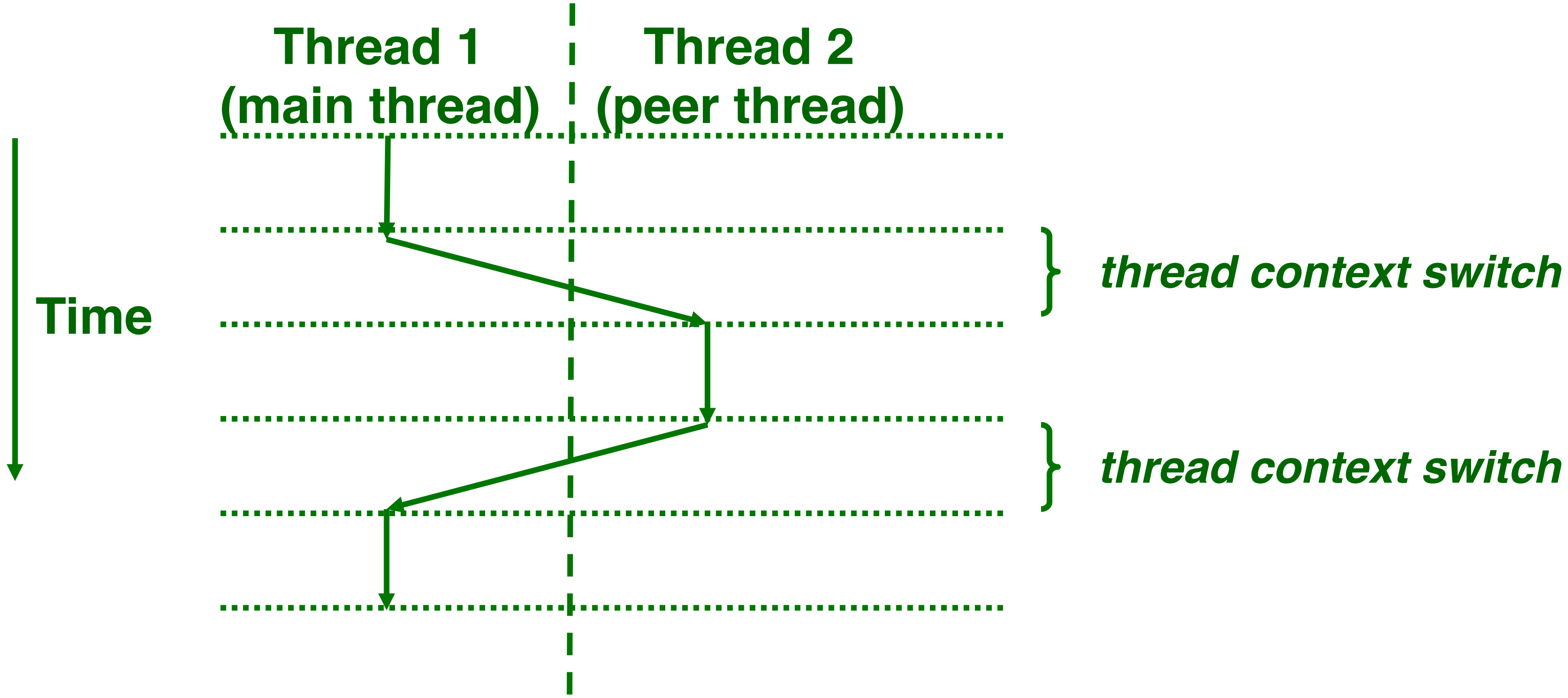


Figure source: COMP 321 lecture on Concurrency (Alan Cox, Scott Rixner)

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, Scott Rixner)

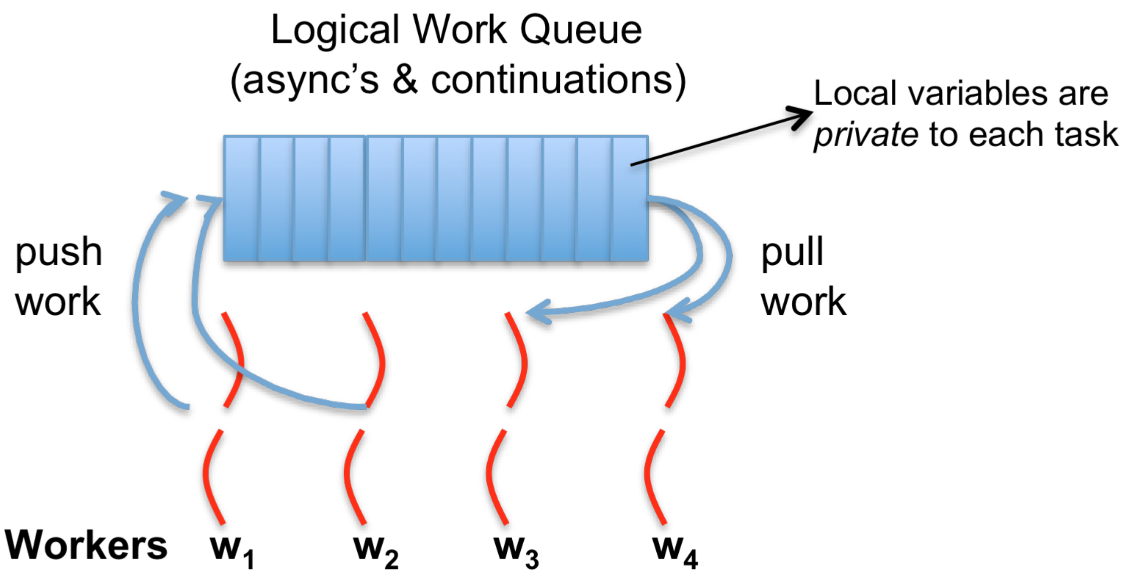
Now, what happens in a task-parallel Java program (e.g., HJ-lib, Java ForkJoin, etc)

HJ-Lib Tasks & Continuations

Worker threads

Operating System

Hardware cores

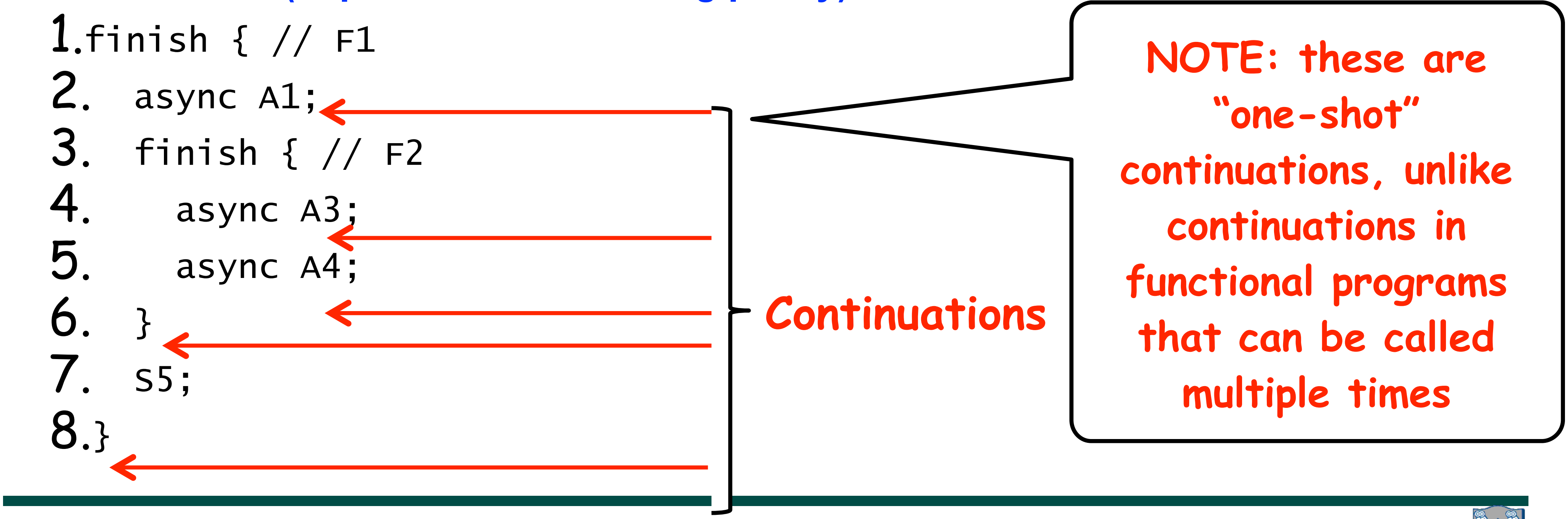

Logical Work Queue
(async's & continuations)
Local variables are *private* to each task
push work
pull work
Workers
w_1
w_2
w_3
w_4

Static & instance fields are *shared* among tasks

- **Task-parallel 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?)**

Continuations

- A continuation is one of two kinds of program points
  - The point in the parent task immediately following an async
  - The point immediately following a *blocking* operation, such as an end-finish, future get(), or barrier
- 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;
}
```

NOTE: these are "one-shot" continuations, unlike continuations in functional programs that can be called multiple times

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

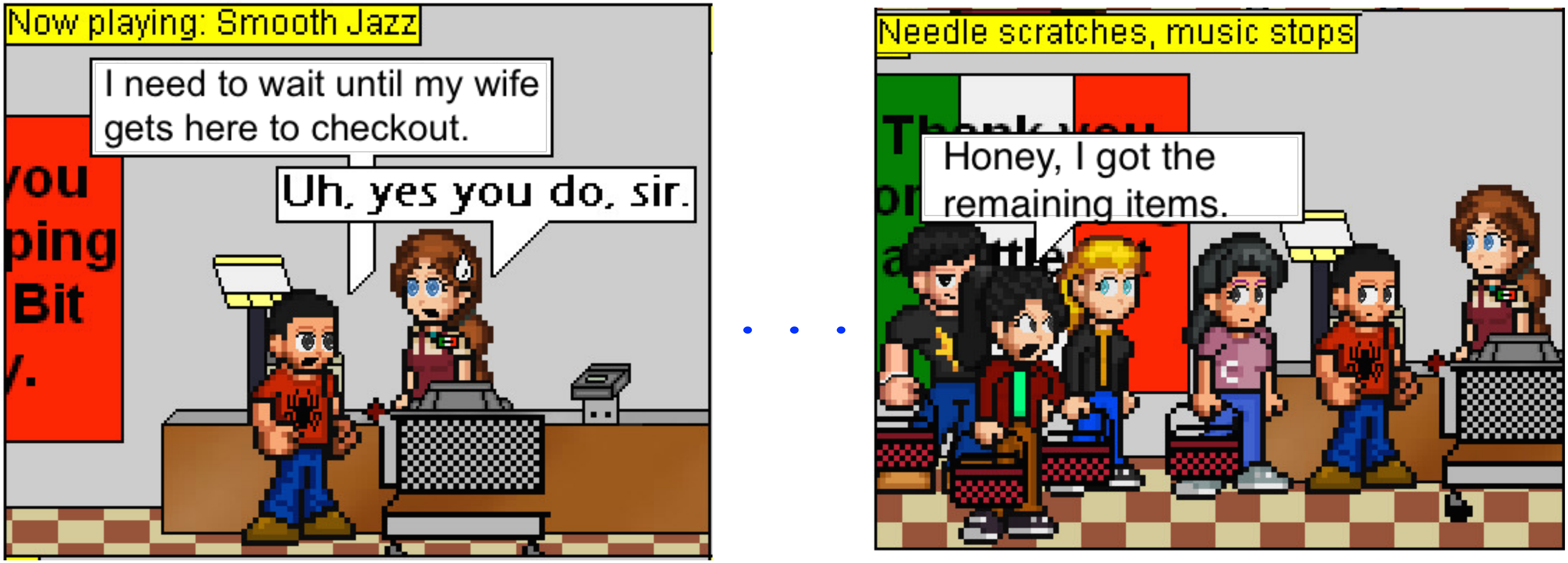
Task-Parallel Model: Checkout Counter Analogy

Last Tuesday
1
2
This is going to be a great day.
3
30 seconds later
1
Wanna bet?
3

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

source: http://www.deviantart.com/art/Randomness-20-178737664

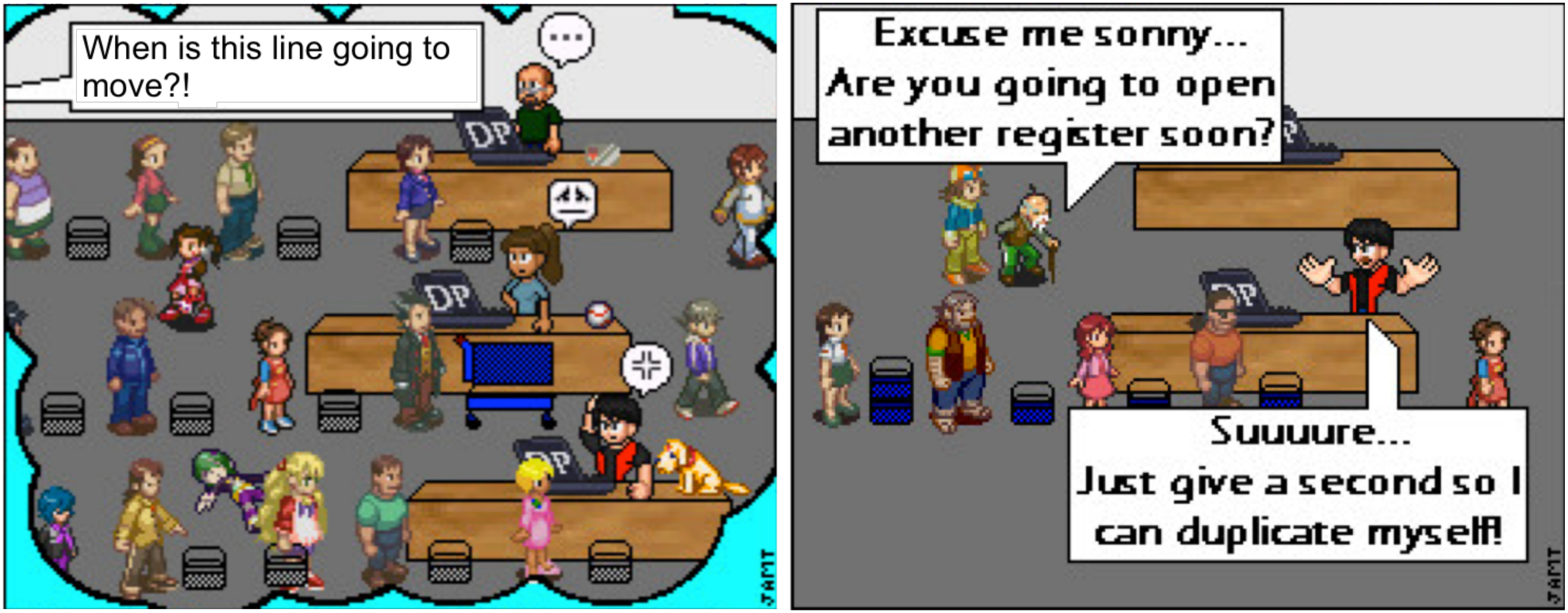
All is well until a task blocks …


Now playing: Smooth Jazz
I need to wait until my wife gets here to checkout.
Uh, yes you do, sir.
you
ping
Bit
y.
. . .
Needle scratches, music stops
Honey, I got the remaining items.

- 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)


When is this line going to move?!
...
DP
JAMT
Excuse me sonny... Are you going to open another register soon?
Suuuure... Just give a second so I can duplicate myself!
JAMT

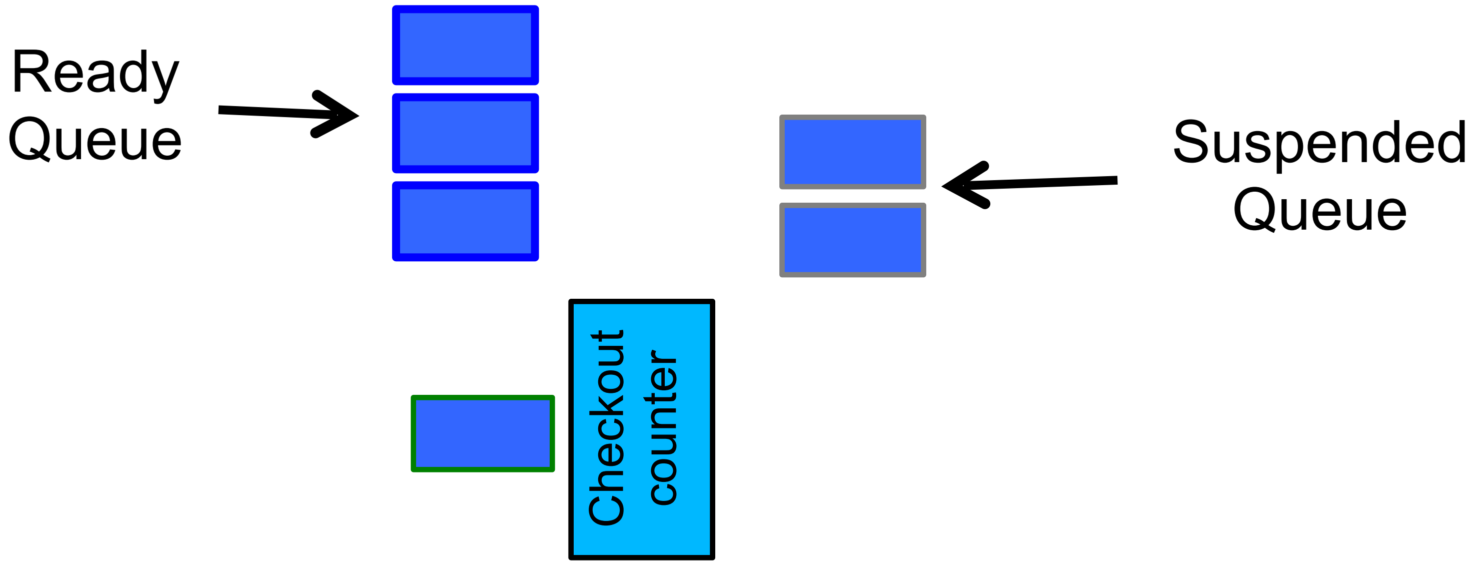
- Creating too many worker threads can exhaust system resources (OutOfMemoryError), and also leads to context-switch overheads when blocked worker threads get unblocked
  - Context-switching in checkout counters stretches the analogy — maybe assume that there are 8 keys to be shared by all active checkout counters?

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

Blocking Runtime (contd)

- Examples of blocking operations
  - End of finish
  - Future get
  - Barrier next
- Blocks underlying worker thread, and launches an additional worker thread
- Too many blocking constructs can result in lack of performance and exceptions
  - `java.lang.IllegalStateException: Error in executing blocked code! [89 blocked threads]`
  - Maximum number of worker threads can be configured if needed
  - `System.setProperty(HjSystemProperty.maxThreads.propertyKey(), "100");`

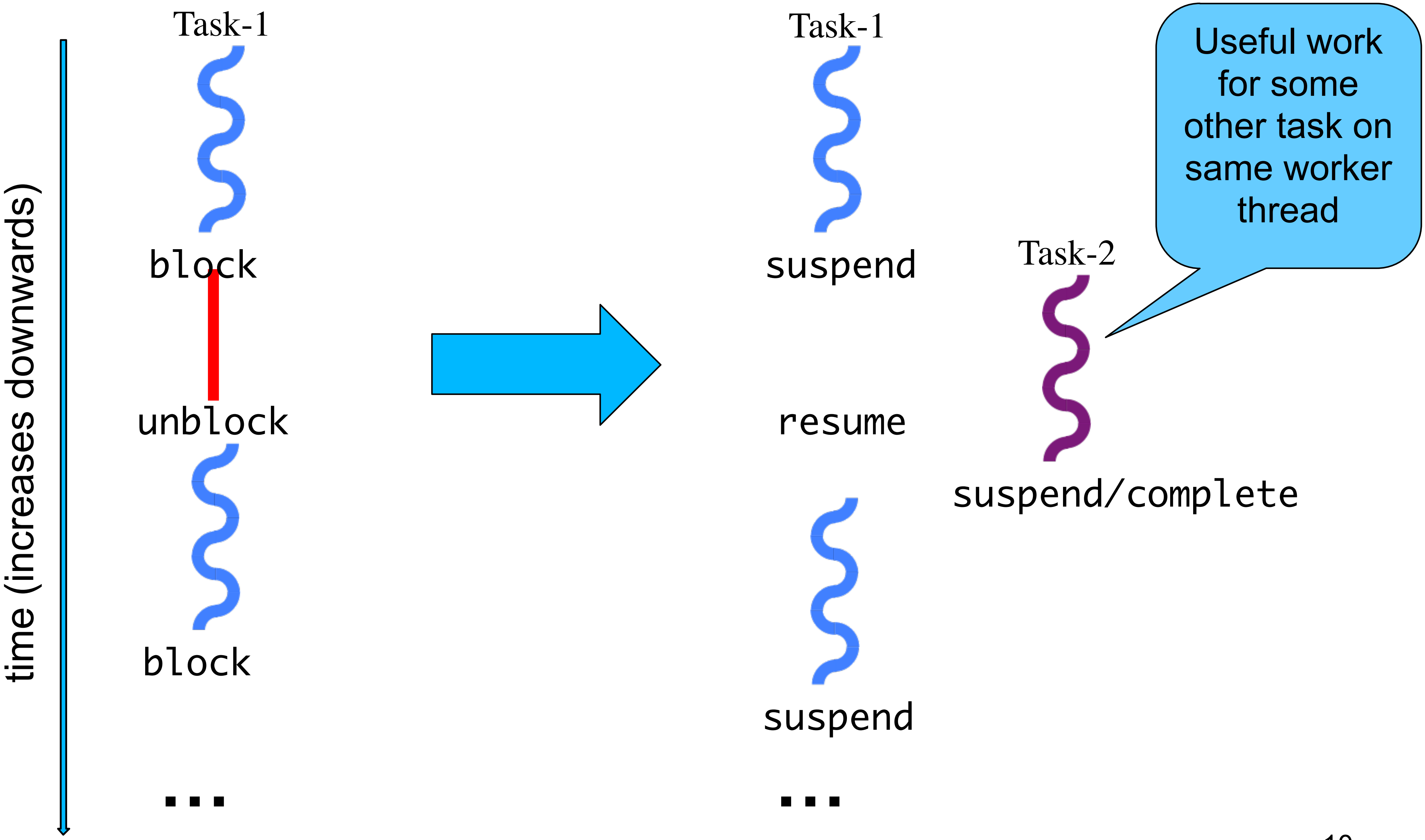
Approach 2: Suspend task continuations at blocking points (as in HJ-Lib's Cooperative Runtime)



- Task actively 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: Complexity and overhead of creating continuations

Cooperative Scheduling: http://en.wikipedia.org/wiki/Computer_multitasking#Cooperative_multitasking

Cooperative Scheduling (view from a single worker)



time (increases downwards)

Task-1

block

unblock

block

...

Task-1

suspend

resume

suspend

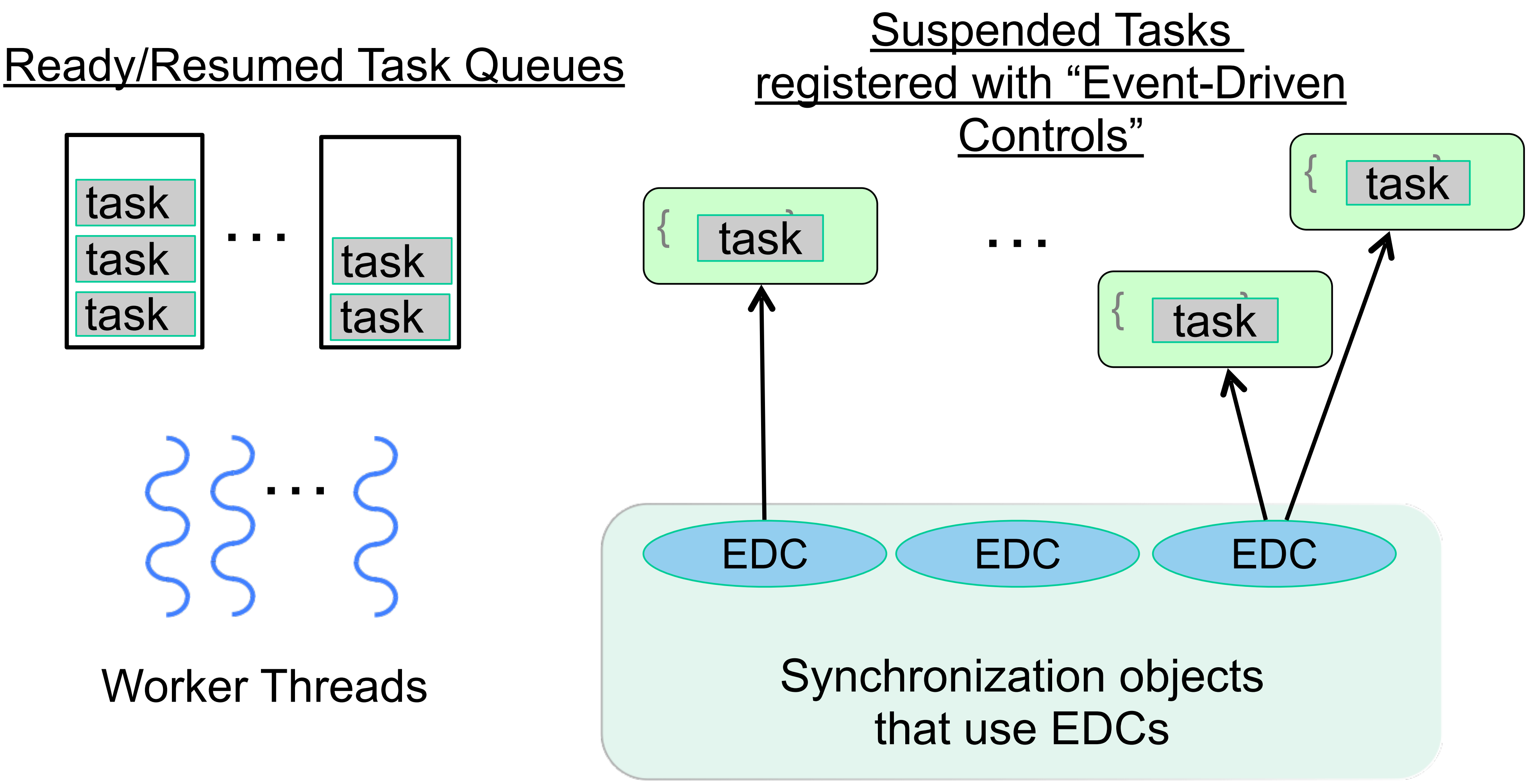
...

Task-2

suspend/complete

Useful work for some other task on same worker thread

HJ-lib’s Cooperative Runtime



Ready/Resumed Task Queues

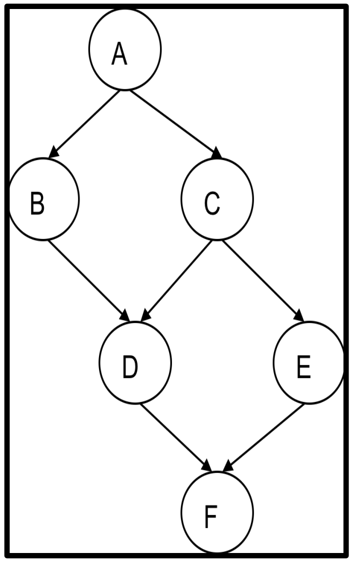
Suspended Tasks registered with “Event-Driven Controls”

Worker Threads

Synchronization objects that use EDCs

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, …

Recap of Data-Driven Tasks



```
finish(() -> {
  HjDataDrivenFuture<Void> ddfA = newDataDrivenFuture();
  HjDataDrivenFuture<Void> ddfB = newDataDrivenFuture();
  HjDataDrivenFuture<Void> ddfC = newDataDrivenFuture();
  HjDataDrivenFuture<Void> ddfD = newDataDrivenFuture();
  HjDataDrivenFuture<Void> ddfE = newDataDrivenFuture();
  async(() -> { ... ; ddfA.put(null); }); // Task A
  asyncAwait(ddfA, () -> { ... ;  ddfB.put(null); }); // Task B
  asyncAwait(ddfA, () -> { ... ;  ddfC.put(null); }); // Task C
  asyncAwait(ddfB, ddfC, ()->{ ... ; ddfD.put(null); }); // Task D
  asyncAwait(ddfC, () -> { ... ;  ddfE.put(null); }); // Task E
  asyncAwait(ddfD, ddfE, () -> { ... }); // Task F
}); // finish
```

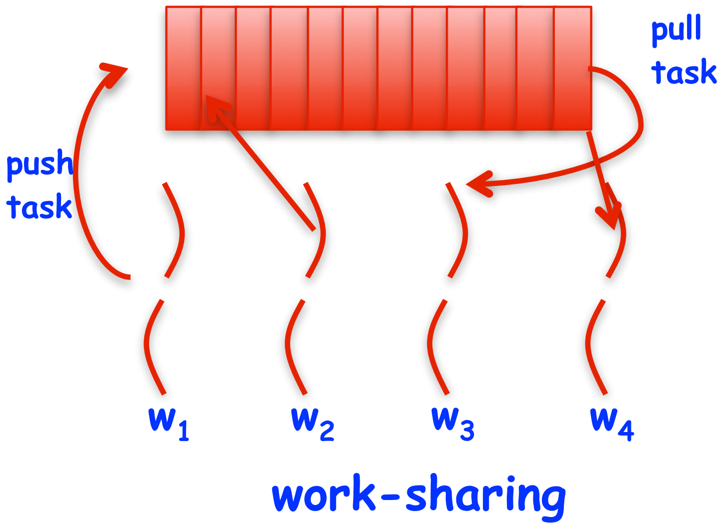
Why are Data-Driven Tasks (DDTs) more efficient than Futures?

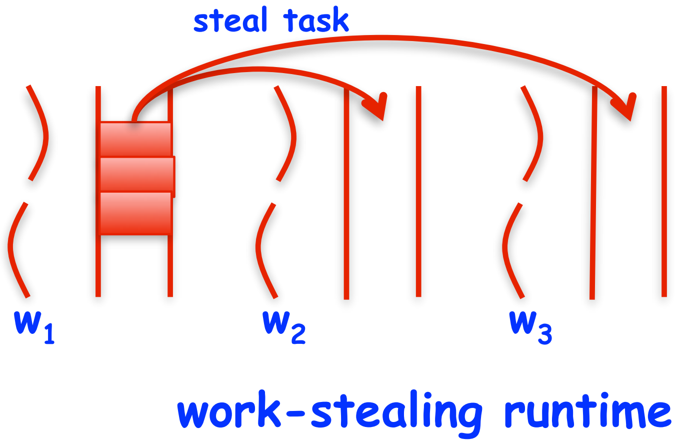
- Consumer task blocks on get() for each future that it reads, whereas async-await does not start execution till all Data-Driven Futures (DDFs) are available
  - An “asyncAwait” statement does not block the worker, unlike a future.get()
  - No need to create a continuation for asyncAwait; a data-driven 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

Work-Sharing vs. Work-Stealing Scheduling Paradigms

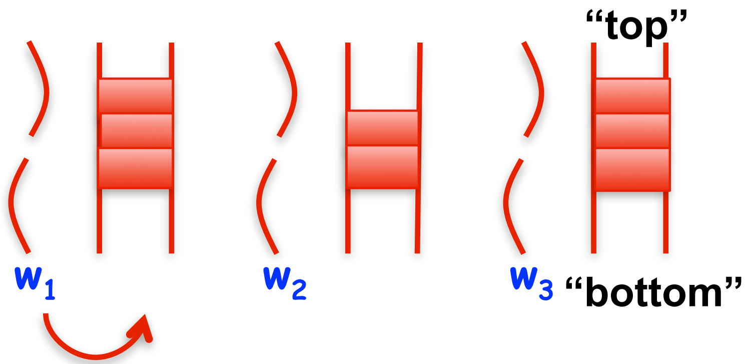
- <u>Work-Sharing</u>
  - Busy worker eagerly distributes new work
  - Easy implementation with global task pool
  - Access to the global pool needs to be synchronized: scalability bottleneck
- <u>Work-Stealing</u>
  - Each worker has its own double-ended queue (deque)
  - Idle worker “steals” the tasks from busy worker’s deque
  - When task T_a spawns T_b, the worker can
    - stay on T_a, making T_b available for execution by another processor (<u>help-first</u> policy), or
    - start working on T_b first (<u>work-first</u> policy)





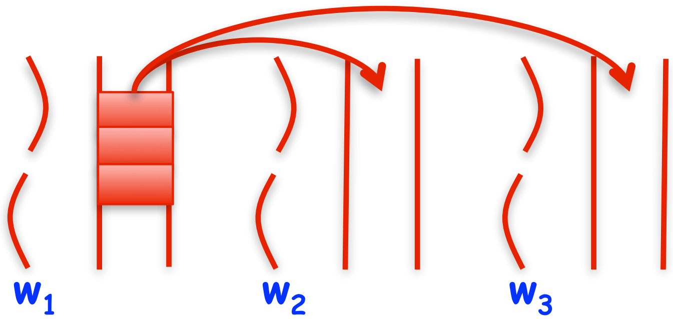
Work-first vs. Help-first work-stealing policies

- When encountering an async
  - Help-first policy
    - Push async on “bottom” of local queue, and execute next statement
  - Work-first policy
    - Push continuation (remainder of task starting with next statement) on “bottom” of local queue, and execute async
- When encountering the end of a finish scope
  - Help-first policy & Work-first policy
    - Store continuation for end-finish
      - Will be resumed by last async to complete in finish scope
    - Pop most recent item from “bottom” of local queue
    - If local queue is empty, steal from “top” of another worker’s queue
- Current HJ-lib runtime only supports help-first policy



Local push/pop by w_1

Stealing by w_2 and w_3



Work-first vs. Help-first work-stealing policies on 2 processors

```
1.  finish {
2.    // Start of Task T0 (main program)
3.    sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
4.    async { // Task T1 computes sum of upper half of array
5.      for(int i=X.length/2; i < X.length; i++)
6.        sum2 += X[i];
7.    }
8.    // T0 computes sum of lower half of array
9.    for(int i=0; i < X.length/2; i++) sum1 += X[i];
10.   }
11.   // Task T0 waits for Task T1 (join)
12.   return sum1 + sum2;
13. } // finish
```

- **Help-first policy: Worker 0 executes lines 1, 2, 3 in T0, pushes out async on line 4, and then executes lines 8, 9 in Task T0. Worker 1 steals async on line 4 and executes task T1.**
- **Work-first policy: Worker 0 executes lines 1, 2, 3 in T0, pushes out continuation on line 8, and then executes async in task T0. Worker 1 steals continuation at line 8 in T0.**

Work-first vs. Help-first work-stealing policies on 2 processors (contd)

```
finish {
  // Start of Task T0 (main program)
  sum1 = 0; sum2 = 0; // sum1 & sum2 are static fields
  async { // Task T1 computes sum of upper half of array
    for(int i=X.length/2; i < X.length; i++)
      sum2 += X[i];
  }
  // T0 computes sum of lower half of array
  for(int i=0; i < X.length/2; i++) sum1 += X[i];
  }
  // Task T0 waits for Task T1 (join)
  return sum1 + sum2;
} // finish
```

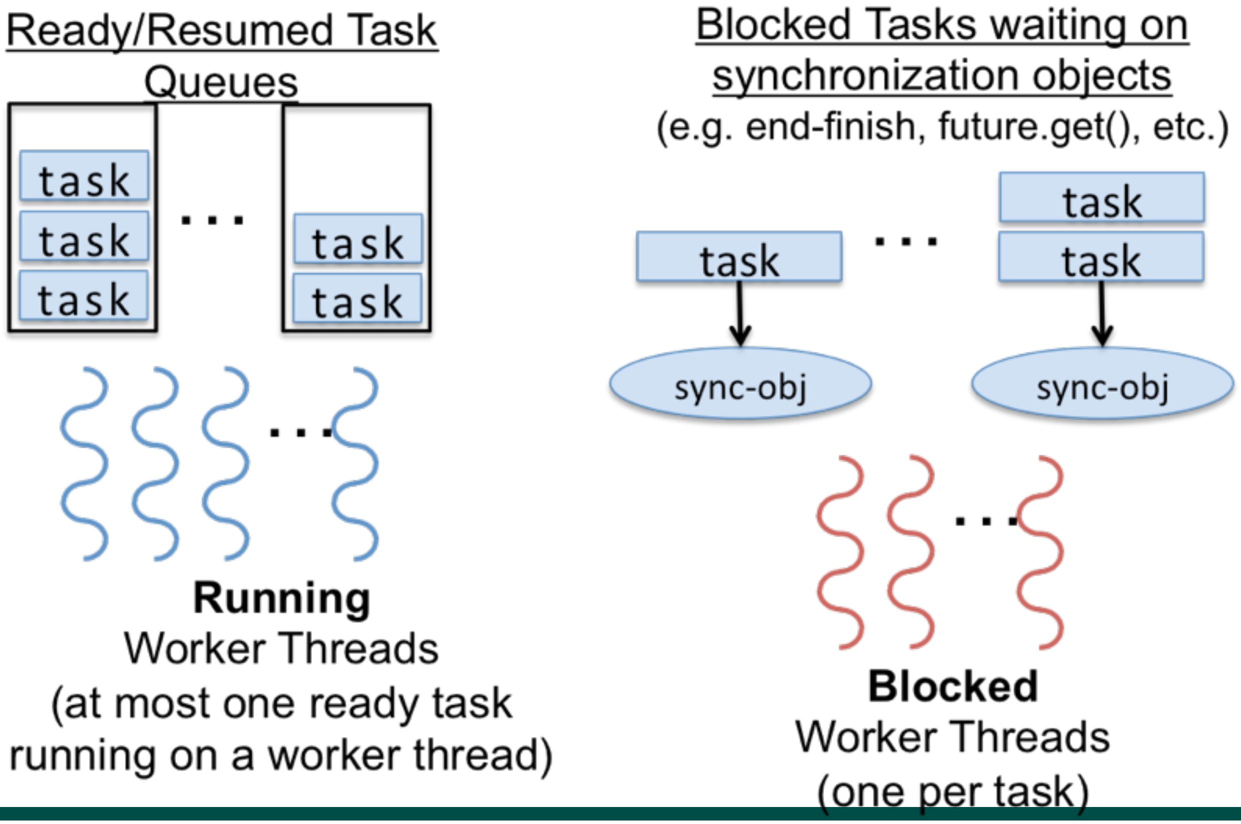
Help-First worker does not switch tasks
Work-first worker will switch tasks

Continuations

Help-First worker can switch tasks
Work-first worker can switch tasks

Summary: Abstract vs. Real Performance in HJlib

- **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 (option-controlled)


Ready/Resumed Task Queues
task
task
task
. . .
task
task
. . .
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.)
task
. . .
task
task
sync-obj
sync-obj
. . .
Blocked Worker Threads
(one per task)

We'll study ForkJoinPool and other Java libraries in detail later in the course --- they manage parallelism at a lower level than HJ