

# Intro to Parallel and Concurrent Programming

Threads and ExecutorServices



[WashU CSE 2301](#)  
Prof. [Dennis Cosgrove](#)  
#26: Tue, Dec 02, 2025

# Course Evals

<https://one.washu.edu/task/all/course-evals>

*S&Q: Can threads be run in parallel?*

yes. that is what they are for.

*S&Q: Still confused on how runnables work.*

# Contrast: Phaser internalAwaitAdvance

```
private int internalAwaitAdvance(int phase, QNode node) {
    // assert root == this;
    releaseWaiters(phase); // ensure old queue clean
    boolean queued = false; // true when node is enqueued
    int lastUnarrived = 0; // to increase spins upon change
    int spins = SPINS_PER_ARRIVAL;
    long s;
    int p;
    while ((p = (int)((s = state) >>> PHASE_SHIFT)) == phase) {
        if (node == null) { // spinning in noninterruptible mode
            int unarrived = (int)s & UNARRIVED_MASK;
            if (unarrived != lastUnarrived &&
                lastUnarrived - unarrived < N_CPU)
                spins += SPINS_PER_ARRIVAL;
            boolean interrupted = Thread.interrupted();
            if (interrupted || --spins < 0) { // need node to record intr
                node = new QNode(this, phase, false, false, 0L);
                node.wasInterrupted = interrupted;
            }
            else
                Thread.onSpinWait();
        }
        else if (node.isReleasable()) // done or aborted
            break;
        else if (!queued) { // push onto queue
            AtomicReference<QNode> head = (phase & 1) == 0 ? evenQ : oddQ;
            QNode q = node.next = head.get();
            if ((q == null || q.phase == phase) &&
                (int)(state >>> PHASE_SHIFT) == phase) // avoid stale enq
                queued = head.compareAndSet(q, node);
        }
        else {
            try {
                ForkJoinPool.managedBlock(node);
            } catch (InterruptedException cantHappen) {
                node.wasInterrupted = true;
            }
        }
    }
    if (node != null) {
        if (node.thread != null)
            node.thread = null; // avoid need for unpark()
        if (node.wasInterrupted && !node.isInterruptible)
            Thread.currentThread().interrupt();
        if (p == phase && p == (int)(state >>> PHASE_SHIFT)) == phase
            return abortWait(phase); // possibly clean up on abort
    }
    releaseWaiters(phase);
    return p;
}
```

*S&Q: Still confused on how runnables work.*

```
@FunctionalInterface
public interface Runnable {
    /**
     * Runs this operation.
     */
    void run();
}
```

*S&Q: I'm still confused by how callable works.*

```
@FunctionalInterface
public interface Callable<V> {
    /**
     * Computes a result, or throws an exception if unable to do so.
     *
     * @return computed result
     * @throws Exception if unable to compute a result
     */
    V call() throws Exception;
}
```

*S&Q: What is the difference between a callable and a runnable?*

- Callable<V>: returns value and throws checked exception
- Runnable: “returns” void

*What if java is running on a processor that does not support threading/tasks?*

*S&Q: I'm not sure I understand the value of ThreadFactory. Could you elaborate on when/why you would use it?*

Sometimes you want finer control:

Executors.[newCachedThreadPool\(threadFactory\)](#)



*S&Q: What does using the threads directly allow us to do that the fork/join functions don't already?*

- run on another processor

ForkJoin uses Executors which use Threads

If we didn't ask the OS for a Thread, we would be stuck on the main Thread on a single processor.

*S&Q: What is the performance difference between executors and threads?*

*S&Q: Why would you ever use threads over fork/join?*

*S&Q: What is the difference between a fork and a thread?*

*S&Q: How do thread and executor services differ in terms of performance and desired use cases?*

*S&Q: Are these things more efficient the 231's  
join\_fork wrappers? Less overhead?*

*S&Q: What are the main differences between this and your own fork\_join and other stuff like that? How come you made your own for this class. Are you abstracting more complicated things that would be hard to do with this java library? such as something like join\_void\_fork\_loop*

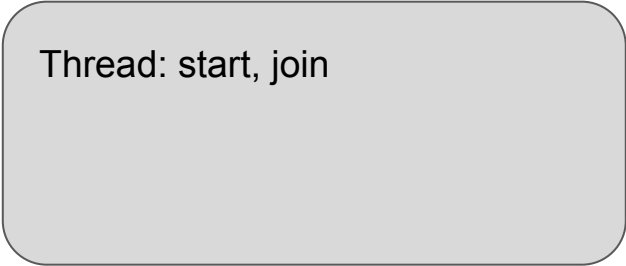
i would not describe ExecutorService as more complicated, just more awkward and slightly more annoying

*S&Q: Why does Java have `ExecutorService` when there are so many other options for parallel?*

*S&Q: Are both executor service and threading included in Java's implementation? Is one method more optimal in certain scenarios (why are there two parallel implementations)?*


- flippant answer: because Threads came first
- bonus answer: reasonable division of labor

# Thread: heavyweight, OS level



Thread: start, join

# ExecutorService: reuse, spread the overhead cost



ExecutorService: submit, invokeAll, future.get

The diagram consists of a light gray rounded rectangle containing the text 'ExecutorService: submit, invokeAll, future.get'. Inside this rectangle is a smaller, darker gray rounded rectangle containing the text 'Thread: start, join'.

Thread: start, join

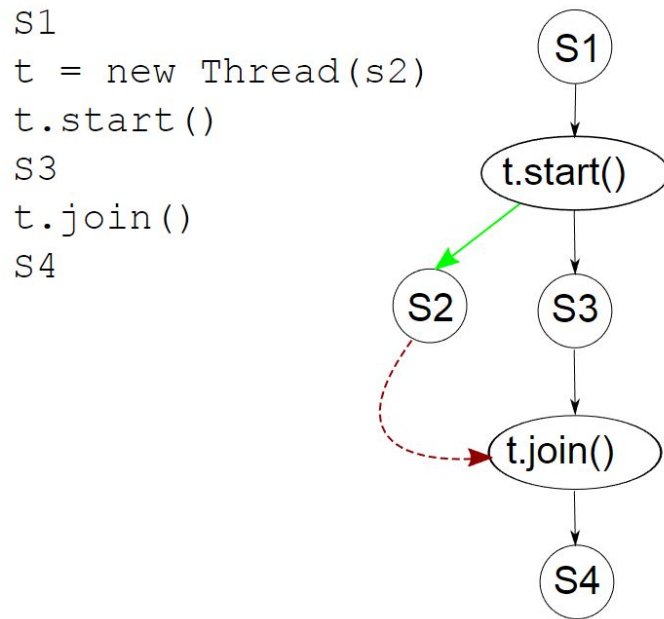
# CSE 231's ForkJoin: less cumbersome



# Computation Graph?

*What would the computation graph for a series involving threads and executors look like?*

- exactly the same  
applicable across languages, paradigms



# Thread

public Thread(Runnable target)

public void start()

public void join()

bonus:

public void run()



# ExecutorService



public <T> Future<T> submit(Callable<T> task)

public Future<?> submit(Runnable task)

public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)

## notes

- I wish they did not provide the Runnable version of submit (bonus: why not return Future<Void>?).
- Often using Callable<Void> and returning null is the right choice.

# Threads lower level. Faster?

*Is it more efficient to use threads than to use forks and joins?*

- not if you did it per task

**HUGE** amount of overhead per Thread (OS is involved)

ExecutorService and ForkJoin amortize the cost by reusing Threads

## *S&Q: When do you use thread versus executorservice?*

I suppose if you knew you were only going to do 1 thing in parallel in your application, you could create, start, and join a Thread instead of using an ExecutorService.

Little reason not to use ExecutorService, anyway, though.

## *S&Q: Are there instances where we would prefer to use Threads without an ExecutorService?*

- when you wanted to build an ExecutorService
- we will build a scheduler later in this course

note: CSE 231s is not about parallel specific to Java, but it would seem crazy to not have everyone construct a Thread, start it, and join it.

imagines interview:

interviewer: *“I see you’ve taken a Parallel course. Say you have a thread...”*

interviewee: *“What’s a thread?”*

interviewer: *“Thank you for your time.”*

*S&Q: What difference if any is there between fork and join, Start and Join, Submit and future get?*

[https://classes.engineering.wustl.edu/cse231/core/index.php?title=Fork\\_Join\\_Rosetta\\_Stone](https://classes.engineering.wustl.edu/cse231/core/index.php?title=Fork_Join_Rosetta_Stone)

- less overhead: ExecutorService & ForkJoin
- absolutely necessary: Threads

cleaner: ForkJoin

*S&Q: What is the advantage of using threads over forks if they're so similar? The same for executor service over fork?*

- less overhead
- cleaner
- less likely to forget to join

*S&Q: In the first video, there was no error or warning when we did not join the threads. Is there any way we can print something on the terminal to let us know we have not joined the threads.*

Thread.State state = thread.getState()

note: tricky to confirm in testing b/c cannot override join. could leverage AspectJ.

*S&Q: What would happen if one or more executor is assigned in total more threads than available CPU cores? Do the threads just execute things in turns?*

- computing has used multiple threads since before I was born

\*\*\* why would you create multiple threads when you only have 1 processor?

# Worksheet

- <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html#get-->
- <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#invokeAll-java.util.Collection->
- <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#submit-java.util.concurrent.Callable->

```
f = let-it-go ( () -> x * x )  
y2 = y * y  
x2 = hold-it-back (f)  
distance = Math.sqrt(x2 + y2)
```



# doItYourselfInvokeAll

```
public static <T> List<Future<T>> doItYourselfInvokeAll
    (ExecutorService executor,
     Collection<? extends Callable<T>> tasks) {
    futures = make List of Futures
    for task in tasks
        future = exec.submit(task)
        futures.add(future)
    for future in futures
        future.get()
    return futures
}
```

```
counts = join_fork_loop(ranges, range ->  
    return count(range)
```

```
for count in counts
```

# doItYourselfInvokeAll \*\*\* what is wrong?

```
public static <T> List<Future<T>> doItYourselfInvokeAll
    (ExecutorService executor,
     Collection<? extends Callable<T>> tasks) {
    futures = make List of Futures
    for task in tasks
        future = executor.submit(task)
        futures.add(future)
    return futures
}
```

# doItYourselfInvokeAll

```
public static <T> List<Future<T>> doItYourselfInvokeAll
    (ExecutorService executor,
     Collection<? extends Callable<T>> tasks) {
    futures = make List of Futures
    for task in tasks
        future = executor.submit(task)
        futures.add(future)
    for future in futures
        future.get()
    return futures
}
```

AAA

*S&Q: I'm confused on the implementation of the invoke all method. Do you have to make a list of "callables"?*

- yes

note: not the most onerous chore, but therein lies the motivation for join\_fork\_loop

*S&Q: What makes the Java version of parallel programming more difficult to manage/use than the one we have been using for the rest of the class?*

*S&Q: will invokeAll launch all tasks in parallel and finish all before exiting? Or will it finish when we call future.get()?*

# Worksheet

- <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/Future.html#get-->
- <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#invokeAll-java.util.Collection->
- <https://docs.oracle.com/javase/8/docs/api/java/util/concurrent/ExecutorService.html#submit-java.util.concurrent.Callable->

# doItYourselfInvokeAll

```
public static <T> List<Future<T>> doItYourselfInvokeAll(ExecutorService executor, Collection<?
extends Callable<T>> tasks) throws InterruptedException, ExecutionException {
```

```
    futs = {}
```

```
    for task in tasks
```

```
        fut = es.submit(task)
```

```
        futs.add(fut)
```

```
    for fut in futs:
```

```
        fut.get()
```

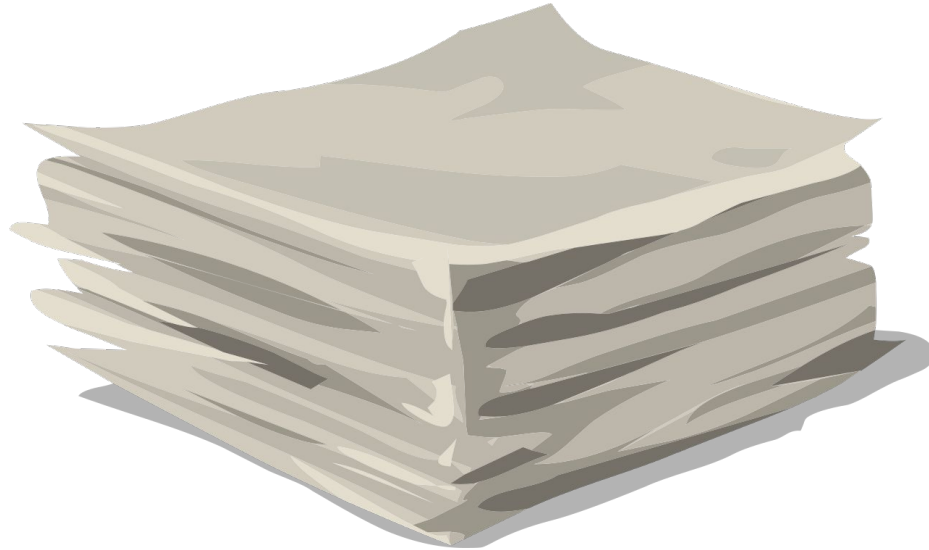
```
    return futs
```

AAA

# Reflect, Synthesize, Ask, and Turn In Worksheets

Note: you are responsible for getting the worksheets to the us.

Do not leave a pile on the end of your table and hope.



*S&Q: I understand that normally when trying to solve the problem of blocking, synchronization, and locking are commonly used. However, when working with threads and executors in Java, is there a specific API to perform similar tasks as Lock and Synchronize? If not, I guess I will still use concurrent hashmap and the materials we've learned previously. Just to be curious.*

we have been using the standard synchronization mechanism in Java



*S&Q: When we create a list of futures and get them one by one, is possible to better parallelize this? I feel like we are creating tasks and getting them one by one doesn't feel good enough.*

- you could divide and conquer get() them

iterable.[splititerator\(\)](#);

*S&Q: Why are any of these frameworks preferable to a regular fork and join?*

They ship with Java.

*S&Q: I've heard Thread is mostly used in companies. Why is Thread powerful compared to other parallel executable classes?*

In practice, Threads are NOT necessarily used more directly.

Thread is a universal term.

People tend to use libraries which amortize the cost of Threads by reusing them.

# Parallel in Interview, Coding Assessment Setting?

*Do you suggest I use parallelization in online coding assessment? In dpf or bfs it could be a good strategy, but I don't want to waste my time during the coding interview to implement something that might not be supported. I just want to know if the industry has a standard answer or preference?*

- at whiteboard, I would mention what could be parallelized
- coding assessment, I would use ExecutorService

note: one of your TAs did this recently and passed over a bar

*S&Q: Can we mix and match these methods with what we have already learned (join and fork), or will that cause errors in our code?*

- in practice, yes, you can mix and match

note: for exercise, the tests enforce that you do not use fork, join (to help catch potential problems earlier).

# Thread and ExecutorService Exercise

- Thread
  - ThreadHalfAndHalfNucleobaseCounter
- ExecutorService
  - ExecutorHalfAndHalfNucleobaseCounter
  - ExecutorCoarseningNucleobaseCounter

# ThreadHalfAndHalfNucleobaseCounter

```
public class ThreadHalfAndHalfNucleobaseCounter implements NucleobaseCounter {  
    @Override  
    public int count(byte[] chromosome, Nucleobase targetNucleobase) {  
        throw new NotImplementedException();  
    }  
}
```

# ExecutorHalfAndHalfNucleobaseCounter

```
public class ExecutorHalfAndHalfNucleobaseCounter implements NucleobaseCounter {  
    public ExecutorHalfAndHalfNucleobaseCounter(ExecutorService executorService) {  
        throw new NotImplementedException();  
    }  
  
    public ExecutorService executorService() {  
        throw new NotImplementedException();  
    }  
  
    @Override  
    public int count(byte[] chromosome, Nucleobase targetNucleobase) {  
        throw new NotImplementedException();  
    }  
}
```

# ExecutorCoarseningNucleobaseCounter

```
public class ExecutorCoarseningNucleobaseCounter implements NucleobaseCounter {
    public ExecutorCoarseningNucleobaseCounter(ExecutorService executorService,
                                                int numRanges) {
        throw new NotImplementedException();
    }

    public ExecutorService executorService() {
        throw new NotImplementedException();
    }

    public int numRanges() {
        throw new NotImplementedException();
    }

    @Override
    public int count(byte[] chromosome, Nucleobase targetNucleobase) {
        throw new NotImplementedException();
    }
}
```

## TA Signup

Talk to the TAs about it, if interested.



# Thread and ExecutorService Exercise

- Thread
  - ThreadHalfAndHalfNucleobaseCounter
- ExecutorService
  - ExecutorHalfAndHalfNucleobaseCounter
  - ExecutorCoarseningNucleobaseCounter

# Thread

public Thread(Runnable target)

public void start()

public void join()

# Runnable

- Interface with single method
  - `public void run()`
- You need to implement run

# Runnable Example

```
public static void main(String[] args) throws InterruptedException {  
    Thread myThread = new Thread(new MyRunnable());  
    myThread.start();  
    doSomethingElse(); // can run in parallel with doSomething  
    myThread.join();  
}
```

```
static class MyRunnable implements Runnable {  
    MyRunnable() {  
  
    @Override  
    public void run() {  
        doSomething(); // can run in parallel with doSomethingElse  
    }  
}
```

# Same thing with anonymous classes

```
Thread myThread = new Thread(new Runnable() {  
    @Override  
    public void run() {  
        doSomething(); // can run in parallel with doSomethingElse  
    }  
});  
myThread.start();  
doSomethingElse(); // can run in parallel with doSomething  
myThread.join();
```

# Same thing with lambdas (best)

```
Runnable myRunnable = () -> {  
    // overrides run() method  
    doSomething();  
};  
Thread myThread  
    = new Thread(myRunnable);  
myThread.start();  
doSomethingElse();  
myThread.join();
```

```
Thread myThread = new Thread(() -> {  
    doSomething();  
});  
myThread.start();  
doSomethingElse();  
myThread.join();
```

# Executors

- Can “submit” tasks to be executed
  - Internally creates and runs on threads
- Motivation: one thread can be used for multiple tasks
  - Thread creation is expensive
- Added convenience methods
- Futures
- Several implementations with several configurations
  - Some use a single thread (single-threaded executor)
  - Some have a fixed number of threads (fixed thread pool)
  - Some create new threads for every new task, but reuse old threads (cached thread pool)

# Executors

public static ExecutorService newSingleThreadExecutor()

public static ExecutorService newFixedThreadPool(int nThreads)

public static ExecutorService newCachedThreadPool()

# ExecutorService

public <T> Future<T> submit(Callable<T> task)

public Future<?> submit(Runnable task)

public <T> List<Future<T>> invokeAll(Collection<? extends Callable<T>> tasks)

# Callable<V>

- Single method interface
  - `public V call()`
- Like Runnable, but for returning a value
  - Useful for futures

Example:

```
Random random = new Random();  
Callable<Integer> dieRoll = () -> random.nextInt(6) + 1;
```

Or, equivalently:

```
Callable<Integer> dieRoll = () -> {  
    return random.nextInt(6) + 1;  
};
```

Future<V>

public V get()

# Example: Array Sum

```
ExecutorService executor = Executors.newFixedThreadPool(2);
int[] array = generateArray();
int mid = array.length / 2;
Future<Integer> lowerTask = executor.submit(() -> {
    int lowerSum = 0;
    for (int i = 0; i < mid; i++)
        lowerSum += array[i];
    return lowerSum;
});
Future<Integer> upperTask = executor.submit(() -> {
    int upperSum = 0;
    for (int i = mid; i < array.length; i++)
        upperSum += array[i];
    return upperSum;
});
int sum = lowerTask.get() + upperTask.get();
executor.shutdown();
System.out.println(sum);
```

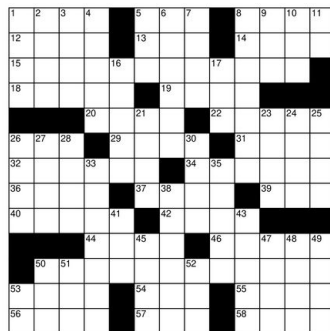
## public Future<?> submit(Runnable task)

```
ExecutorService executor = Executors.newWorkStealingPool()
Future<?> future = executor.submit(() -> {
    doSomething();
});
doSomethingElse();
future.get(); // returns null, no need to store return value
```

# Equivalent example with invokeAll

```
ExecutorService executor = Executors.newFixedThreadPool(2);
int[] array = generateArray();
int mid = array.length / 2;
List<Callable<Integer>> tasks = new LinkedList<>();
tasks.add(() -> {
    int lowerSum = 0;
    for (int i = 0; i < mid; i++)
        lowerSum += array[i];
    return lowerSum;
});
tasks.add(() -> {
    int upperSum = 0;
    for (int i = mid; i < array.length; i++)
        upperSum += array[i];
    return upperSum;
});
List<Future<Integer>> futures = executor.invokeAll(tasks);
executor.shutdown();
int sum = 0;
for (Future<Integer> future : futures)
    sum += future.get();
System.out.println(sum);
```

# Worksheet



**DOWN**  
 1 What flags do in the wind  
 2 The Hawkeye State  
 3 It's half the faun  
 4 End of a philosophers utencil  
 5 Pie \_\_\_ mode  
 6 Tranquillize  
 7 \_\_\_ Strauss jeans  
 8 Millennium Falcon pilot  
 9 Co. name completer  
 10 Adversary  
 11 \_\_\_-N-Out Burger  
 16 "We hold \_\_\_ truths..."  
 17 It may be framed

21 Sassy  
 23 Cream puff  
 24 Mideast bigwig  
 25 It may be reentrant  
 26 Cathedral recess  
 27 Lose, as skin  
 28 Put up, as a picture  
 30 Alliance since '49  
 33 How to give a reprimand  
 35 \_\_\_ eleven: how a soccer team is forced to proceed after a red card  
 38 "Carmen" and "Norma"  
 41 Anatomical pouch  
 43 Artisan alliance  
 45 Start of a philosophers utencil  
 47 Former Winter Palace resident: Var.  
 48 Disavow  
 49 "What are the \_\_\_?"  
 50 Compare and \_\_\_  
 51 Thumbs-up vote  
 52 Frequently, in poetry  
 53 Inclusion after the sig.

## ACROSS

1 Newton ingredients?  
 5 Digital communication?: Abbr.  
 8 Platter player  
 12 Ransack  
 13 Author Harper  
 14 Briefly unknown?  
 15 Arrive and \_\_\_  
 18 Quilt square  
 19 They're sometimes put on  
 20 Didn't part with  
 22 Sauna attire  
 26 Volcanic fallout  
 29 Spotted  
 31 Car with a bar  
 32 This class gives a shout out to Vijay Saraswat and Vivek Sarkar  
 34 "Get then put is not \_\_\_"  
 36 Shipped  
 37 Canvas bag  
 39 LASER alternative:  
 40 Noses (out)  
 42 Pioneering 70's video game  
 44 Condition preceder?  
 46 Perform better than  
 50 Implicit 25-Down keyword  
 53 Rind  
 54 Big galoot  
 55 Hit the runway  
 56 "Don't move!"  
 57 L.A. clock setting  
 58 Teetotalers

\*Inclusion

53. Down: Beatles song: \_\_\_\_, I love you.

# Final Prep (78 minute video!)



VALID SCHEDULES WITH TWO CORES/WORKERS

Worker 0  
Worker 1

```
graph TD; fib4["fib(4)"] --> fib3["fib(3)"]; fib4 --> fib2["fib(2)"]; fib3 --> fib2_0["fib(2)"]; fib3 --> fib1_0["fib(1)"]; fib2_0 --> fib1_0_0["fib(1)"]; fib2_0 --> fib0_0["fib(0)"]; fib2 --> fib1_1["fib(1)"]; fib2 --> fib0_1["fib(0)"];
```

PABLO HALPERN

Work Stealing

[www.CppCon.org](http://www.CppCon.org)

# Course Evals

<https://one.washu.edu/task/all/course-evals>