Java Threads

Basics: Thread. Runnable

Synchronization Issues

Concurrent collections: java.util.concurrent.*

Task Parallelism/Task Pools:

Executor Framework, ForkJoin Framework

Bibliography

- Parallel Programming (in Java) @ ETH Zurich https://spcl.ethz.ch/Teaching/2020-pp/
- Lesson: Concurrency (The Java™ Tutorials > Essential Java Classes)

Threads and Parallelism in Java

- Java supports multiple APIs/frameworks for concurrency/parallelism
  - different levels of abstractions, different concepts

1. Raw Threads: (similar Posix threads)
   - Explicitly create threads assigning them work to do
   - Synchronization primitives for mutual exclusion and signaling
   - Class Thread, Interface Runnable
2. Task pools: (similar omp task)
   - Executor framework
   - Fork-Join framework
3. Parallel streams – not discussed in this class

Java Threads

- Thread:
  - A set of instructions to be executed one at a time, in a specified order
  - Threads exist within a process — every process has at least one thread. Threads share the process's resources, including memory and open files.
  - A special Thread class is part of the core language java.lang.Thread
- Creating Java Threads: 2 options:
  1. Instantiating a subclass of Thread
  2. Implementing a Runnable

Thread Objects

- Each thread is associated with an instance of the class Thread.
- `Class java.lang.Thread`
- `Interface java.lang.Runnable`
- `Class Thread implements Runnable`

- There are different strategies for using Thread objects to create a concurrent application:
  - Low-level concurrency API: applications directly control thread creation and management by instantiating Thread each time the application needs
  - Higher-level concurrency frameworks: they help abstract thread management, applications do not explicitly create thread objects

Defining and starting a Thread

- An application that creates an instance of Thread must provide the code that will run in that thread. There are two ways to do this:

1. Implement Runnable interface
    - The Runnable interface defines a single method, run, meant to contain the code executed in the thread. The Runnable object is passed to the Thread constructor
2. Subclass Thread
    - The Thread class itself implements Runnable, though its run method does nothing. An application can subclass Thread, providing its own implementation of run

Create Java Thread: Option1

- Implement `java.lang.Runnable`
- Single method: public void run()
- Write a Class that implements `Runnable`
- Create a Thread object with this `Runnable`

```
public class GreetingRunnable implements Runnable {
   public void run() {
   // code here executed by thread
   }
}
GreetingRunnable greetingRunnable = new GreetingRunnable();
Thread t = new Thread(greetingRunnable);
t.start();
```

Create Java Thread: Option2

- Instantiate a subclass of `java.lang.Thread`
- Override run method (must be overridden) to say what thread will be doing
- Call start() method to create a new thread. The start() method invokes run()
- Calling run() directly does not create a new thread !

```java
public class GreetingThread extends Thread {
    public void run() {
    // code here executed by thread
    }
}

GreetingThread greetingThread = new GreetingThread();

greetingThread.start();
```

Creating thread object
does not start the thread!
Must call start() to actually
create and start the thread

Create Thread: Option1 vs Option 2

- Which of these idioms should you use?
- Option1 which employs a Runnable object
  - is more general, because the Runnable object can subclass a class other than Thread.
- Option2 which subclasses Thread:
  - is easier to use in simple applications, but is limited by the fact that your task class must be a descendant of Thread.
- Recommended: Option1 which separates the Runnable task from the Thread object that executes the task.
  - more flexible
  - applicable to the higher-level thread management APIs

Joining Threads

- Common scenario: Main thread starts several *worker threads*, then needs to wait for the worker's results to be available
- The join method allows one thread to wait for the completion of another. If t is a Thread object whose thread is currently executing,
- t.join();
- causes the current thread to pause execution until t's thread terminates. Overloads of join allow the programmer to specify a waiting period.

First Example: Greetings

```java
class Hello implements Runnable {
    private final int externalID;

    public Hello(int externalID) {
        this.externalID = externalID;
    }

    @Override
    public void run() {
        System.out.printf("Thread %s  with externalID=%d says Hello!\n",
                                                        Thread.currentThread().getName(), externalID);
    }
```

First Example: Greetings (contd)

```
  Thread[] threads = new Thread[NTHREADS];
  for (int i = 0; i < NTHREADS; ++i) {
    if (i % 2 == 0) threads[i] = new Thread(new Ciao(i));
    else threads[i] = new Thread(new Hello(i));
  }
  System.out.println("Thread objects have been created but not yet started");

  for (int i = 0; i < NTHREADS; ++i) {
    threads[i].start();
  }

  System.out.println("Wait to join threads ...");
  for (int i = 0; i < NTHREADS; ++i) {
    try {
      threads[i].join();
    } catch (InterruptedException e) {
      throw new RuntimeException(e);
    }
  }
  System.out.println("All threads joined!");
}
```

Source Code

- https://staff.cs.upt.ro/~ioana/apd/java/Greetings.java

Synchronization Issues

synchronized

wait, notify, notifyAll

Shared Memory Between Threads

- Example: Shared Counter: Two threads increment the same counter:

```
class Counter {
    private int value;
    public Counter(int value){
        this.value = value;
    }
    public void increment(){
        value++;
    }
    public void decrement(){
        value--;
    }
    public int getValue(){
        return value;
    }
}
```

```
class Incrementer implements Runnable {

    private Counter aCounter;
    private int repeats;
    public Incrementer(Counter aCounter, int repeats) {
        this.aCounter = aCounter;
        this.repeats = repeats;
    }

    @Override
    public void run() {
        for (int i=0; i<repeats; i++) {
            aCounter.increment();
        }
    }
}
```

```java
public class WrongSharedCounter {

    private static int REPEATS=1000000;
    public static void main(String[] args) {

        Counter theCounter = new Counter(0);

        Thread t1 = new Thread(new Incrementer(theCounter, REPEATS));
        Thread t2 = new Thread(new Incrementer(theCounter, REPEATS));

        t1.start();
        t2.start();

        try {
            t1.join();
            t2.join();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }

        System.out.println("Counter value is "+theCounter.getValue());
    }
}
```

```
Counter value is 1459995

Counter value is 1444810

Counter value is 1478871
```

Shared Memory Between Threads

- Example: Shared C ame counter:

Wrong! Needs Mutual Exclusion!

```
class Counter {
    private int value;
    public Counter(int value){
        this.value = value;
    }
    public void increment(){
        value++;
    }
    public void decrement(){
        value--;
    }
    public int getValue(){
        return value;
    }
}
```

```
nnable {

    private Counter aCounter;
    private int repeats;
    public Incrementer(Counter aCounter, int repeats) {
        this.aCounter = aCounter;
        this.repeats = repeats;
    }

    @Override
    public void run() {
        for (int i=0; i<repeats; i++) {
            aCounter.increment();
        }
    }
}
```

Critical sections - Mutual exclusion

- When two or more threads read/write the same data (shared objects), the programmer is responsible for avoiding bad interleaving by explicit synchronization – mutual exclusion!
- We know that mutual exclusion can be realized by locks
- ***In Java, all objects have an internal lock, called intrinsic lock or monitor lock: every object can be used as a lock!***
- ***Keyword: synchronized***

```
 synchronized (object) {
   statement(s); // critical sections
}
```

Using locks to protect shared value

```
class Counter {
   private int value;
   public Counter(int value){
      this.value = value;
   }
   public  void increment(){
      synchronized (this) {
         value++;
      }
  }
   public  void  decrement(){
      synchronized (this) {
         value--;
      }
  }
   public  int getValue(){
      synchronized (this) {
         return value;
      }
   }
}
```

The current object this is used as a lock.
Two different threads cannot access concurrently the value attribute of the same object

Synchronized methods

- ***In Java, methods can be declared as synchronized:***
  - By default uses the ***this*** object as a lock
  - A synchronized method aquires the object lock at the start, runs to completion, then releases the lock
  - This is useful for methods whose *entire* bodies are critical sections

Using synchronized methods

```
class SynchronizedCounter {
    private int value;
    public SynchronizedCounter(int value){
        this.value = value;
    }
    public synchronized void increment(){
        value++;
    }
    public synchronized void  decrement(){
        value--;
    }
    public synchronized int getValue(){
        return value;
    }
}
```

Synchronized methods
are equivalent with a
lock on the current
object.
Easier for this example.

When to use locks on objects instead on synchronized methods?

```
class Counter {
   private int value;
   public Counter(int value){
      this.value = value;
   }
   public  void increment(){
      synchronized (this) {
         value++;
      }
      System.out.println("did increment");
   }
}
}
```

Use locks instead of synchronized method if the method also contains **some non-critical code**!

Also synchronized(object) can be used if the object lock is not necessarily the **this** object!

Source Code

- staff.cs.upt.ro/~ioana/apd/java/WrongSharedCounter.java
- staff.cs.upt.ro/~ioana/apd/java/CorrectSharedCounter1.java
- staff.cs.upt.ro/~ioana/apd/java/CorrectSharedCounter2.java

Coordinating actions of multiple threads

- Guarded blocks: such blocks keep a check for a particular condition before resuming the execution.
- Concept similar to Condition Variables in POSIX threads

- Object.wait() to suspend a thread
- Object.notify() to wake a thread up

wait(), notify(), notifyAll()

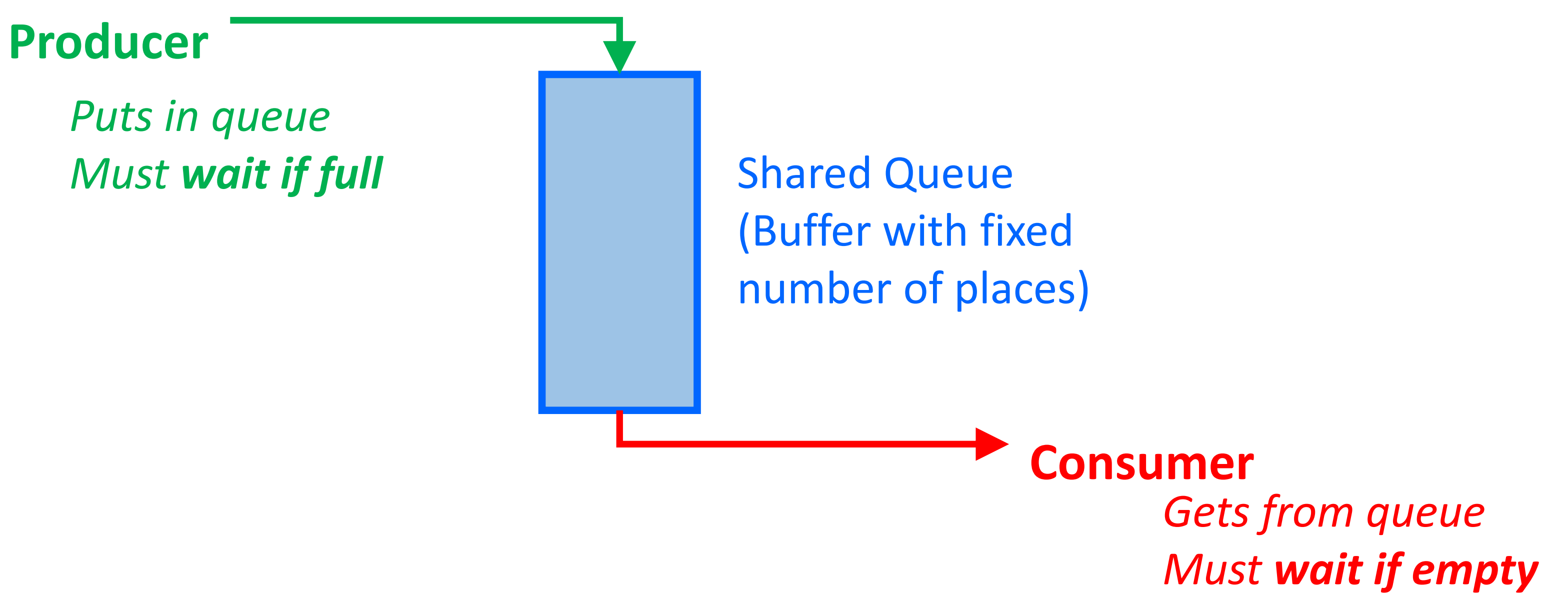
- Inter-thread communication allows threads to coordinate and synchronize: wait, notify, and notifyAll methods
- May only be called when object is locked (e.g. inside `synchronized`)
- `wait()` forces the current thread to enter a waiting state until another thread calls notify() or notifyAll() on the same object. To make this happen, the current thread must own the lock of that object. During wait, lock is released.
- `notify()` wakes the highest-priority thread closest to front of object's internal queue
- `notifyAll()` wakes up all waiting threads. Threads non-deterministically compete for access to object.

The Producer-Consumer Problem

See also the previous discussion of this problem with POSIX threads:
https://staff.cs.upt.ro/~ioana/apd/Synchronization.pdf

Producer-Consumer with Bounded Buffer

- A number of **Producers** put items into a Shared Queue (a Buffer)
- A number of **Consumers** get items out of the Shared Queue
- All Producers and Consumers work concurrently
- The size of the Queue is **fixed (there are a limited number of places in the Queue = a Bounded Buffer)**

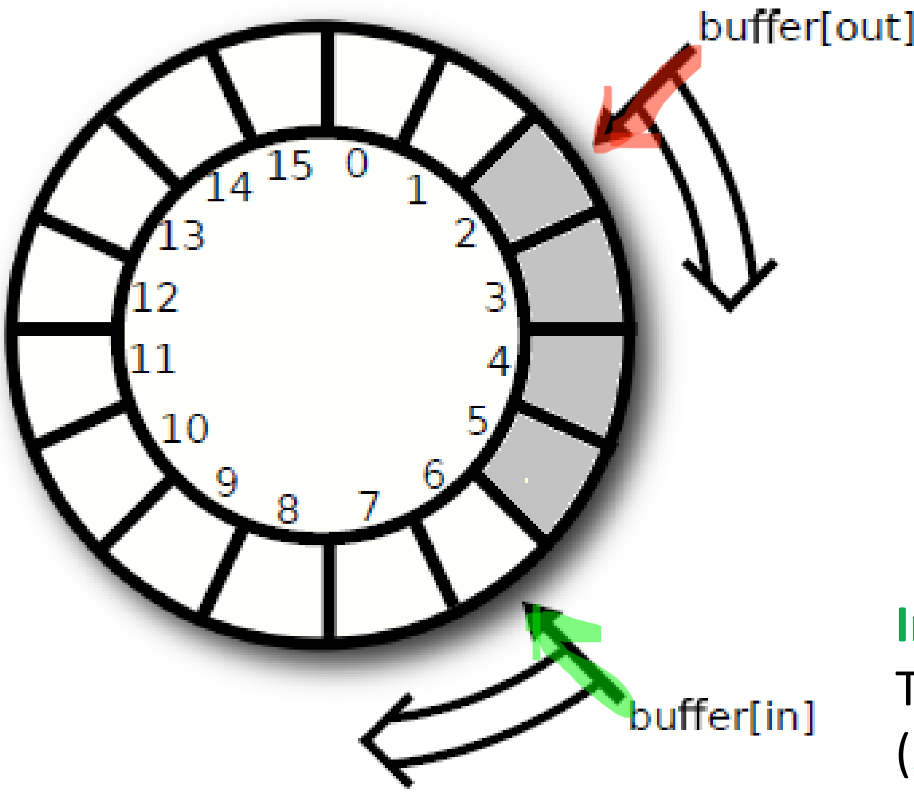


Producer-Consumer with Bounded Buffer

- Problems:
  - If the Buffer is **empty**, *Consumers must block* until some item appear in queue
  - If the Buffer is **full**, *Producers must block* until some item is removed from queue
  - Several Producers or Consumers must not attempt to put or get items from the queue at the same time (the **classical mutual exclusion** – cannot have 2 threads increment the same head or tail index at the same time)

Bunded Buffer Queue Implementation

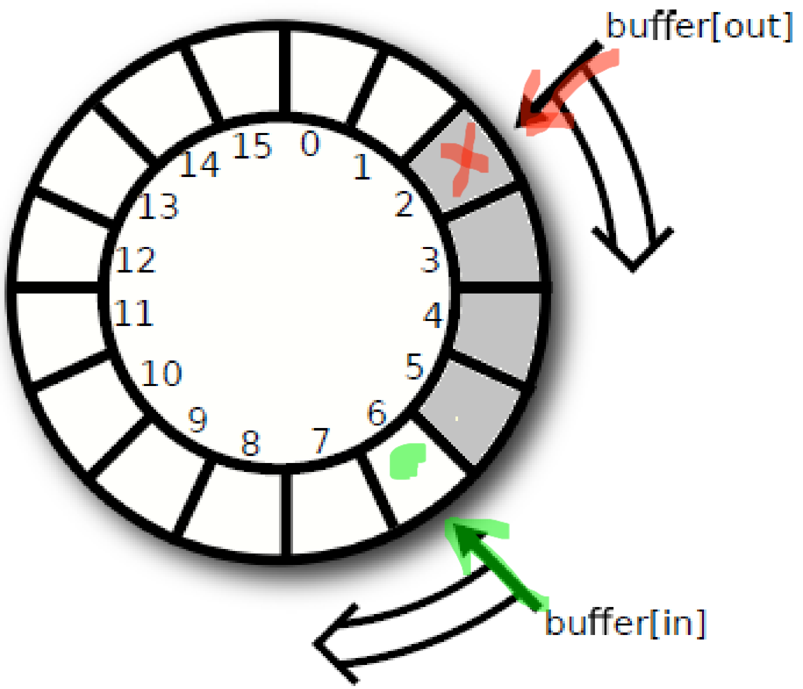
- **Circular Array** (**Ring Buffer**): an array exploited in a circular way: after the last index, we consider that the next element follows at the first index



out: the index of the Head of the Queue (*Consumers get this element – dequeue*)

In: the index of the Tail of the Queue (*Producers put here - enqueue*)

Bunded Buffer Queue Implementation



- Initially: Buffer empty, in=out=0
- Enqueue: Put in buffer:
  - b[in] = value;
  - in = (in + 1) % BUFFER_SIZE;
- Test Buffer is Full:
  - If ((in + 1) % BUFFER_SIZE == out)
- Dequeue: Get from buffer:
  - value = b[out];
  - out = (out + 1) % BUFFER_SIZE;
- Test Buffer is Empty:
  - If (out == in)

BoundedBuffer Producer-Consumer

- Producer and consumer run indefinitely
- Producer adds items into a shared buffer, consumer removes them out
- Buffer is bounded (has a limited capacity)
- Producer can add only if buffer is not full. If buffer is full, Producer must wait until place is freed in buffer (a Consumer eventually takes something out)
- Consumer can remove only if buffer is not empty. If buffer is empty, Consumer must wait until some elements appear in the buffer (a Producer eventually puts something in)

```java
class BoundedBuffer {
    private final Long[] elements;
    private final int capacity;
    private int in, out;
    private int count;

    public BoundedBuffer(int capacity) {
        this.capacity = capacity;
        this.elements = new Long[capacity];
        in = 0;
        out = 0;
        count = 0;
    }
    public boolean isEmpty() {  return (count == 0);   }
    public boolean isFull() {return (count == capacity);  }

    public void add(long value) {
        elements[in] = value;
        in = (in + 1) % capacity;  count++;
    }

    public long remove() {
        long v = elements[out];
        out = (out + 1) % capacity; count--;
        return v;
    }
}
```

A simple, bounded buffer that is not thread-safe - The buffer does not perform its own synchronization

It works as a circular buffer

```java
class Producer extends Thread {
    private final BoundedBuffer buffer;

    public Producer(BoundedBuffer buffer) {
        this.buffer = buffer;
    }

    @Override
    public void run() {
        long number = 0;

        while (true) {
            number = … // … produce a value
            synchronized (buffer) {
                while (buffer.isFull()) { //needs while, not if!!
                    try {
                        buffer.wait(); //producer is blocked
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
                buffer.add(number);
                buffer.notifyAll(); // producer notifies blocked consumers
            }
        }
    }
```

```java
class Consumer extends Thread {
    private final int id;
    private final BoundedBuffer buffer;

    public Consumer(int id, BoundedBuffer buffer) {
        this.id = id;
        this.buffer = buffer;
    }

    @Override
    public void run() {
        long number;
        while (true) {
            synchronized (buffer) {
                while (buffer.isEmpty()) { // need a while loop, not if !!!
                    try {
                        buffer.wait(); // consumer is blocked
                    } catch (InterruptedException e) {
                        throw new RuntimeException(e);
                    }
                }
                number = buffer.remove();
                buffer.notifyAll();  // consumer notifies blocked producers
            }
            performLongRunningComputation(number);
        }
    }
```

Why do we need loops with wait/notify

- ***Recall condition variables with POSIX threads: the same reason: spurious wake-ups!***
- Suppose that Consumer uses an if:
- ```
  if (buffer.isEmpty())
          buffer.wait();
  ```
- The problem is that the consumer can return from a wait() call for reasons other than being notified (e.g. due to a thread interrupt), or because different consumers have different conditions
- If we do not recheck the isEmpty() condition upon return from wait, we do not know why the thread returned from wait()
  ```

Source Code

- staff.cs.upt.ro/~ioana/apd/java/ProducerConsumer.java

ThreadSafe Datastructures. Concurent Collections.

ThreadSafe Blocking BoundedBuffer

- We can **make the BoundedBuffer threadsafe** and blocking, by moving the synchronization issues from Producer and Consumer to the methods add() and remove() of the buffer
- In this case, the Producer and Consumer working with a BlockingBoundedBuffer do not need to address any synchronization issues!

```java
class BlockingBoundedBuffer {

// … same attributes …

    public synchronized boolean isEmpty() {  return (count == 0); }

    public synchronized boolean isFull() {  return (count == capacity); }

    private void _add(long value) {
        elements[in] = value;
        in = (in + 1) % capacity;
        count++;
    }

    public synchronized void add(long value) {
            while (this.isFull()) {
                try {
                    this.wait();
                } catch (InterruptedException e) {
                    throw new RuntimeException(e);
                }
            }
            _add(value);
            this.notifyAll();
    }
```

```java
class Producer extends Thread {
    private final BlockingBoundedBuffer buffer;

    public Producer(BlockingBoundedBuffer buffer) {
        this.buffer = buffer;
    }

    private long produceNumber() {
        // produce and return a value
    }

    @Override
    public void run() {
        long number = 0;
        while (true) {
            number = produceNumber();
            //synchronization is handled by the buffer
            buffer.add(number);
        }
    }
}
```

Source Code

- https://staff.cs.upt.ro/~ioana/apd/java/BlockingArrayProdCons.java

Java Synchronized Collections

- Two ways:
  1. Thread-safe collection wrappers via static methods in the Collection class
  2. **Package java.util.concurrent** contains collections that are optimized for being used as threadsafe by multiple threads (more efficient than synchronized wrapper on standard collection)

Concurrent Collections

- Interface BlockingQueue: a ready-to-use synchronized Buffer
  - Class ArrayBlockingQueue
  - Class LinkedBlockingQueue
  - Class PriorityBlockingQueue
- Class ConcurrentHashMap: **get** and **put** operations block access to a specific element but do **NOT** block access to the entire data structure. In other words, two or more threads can access it simultaneously.

AtomicInteger

- java.util.concurrent.atomic.AtomicInteger
- ensures atomic operations on integer variables
- Example: updating a shared counter: a compound operation which involves 2 steps:
  - Read the existing value from memory
  - Update the new value to memory
- With AtomicInteger the update operation is performed in a single atomic operation
  - AtomicInteger is implemented using the C-A-S (CompareAndSwap or CompareAndSet) operation which is a primitive CPU level operation available at the hardware level

Task pools

Executor Framework

ForkJoin Framework

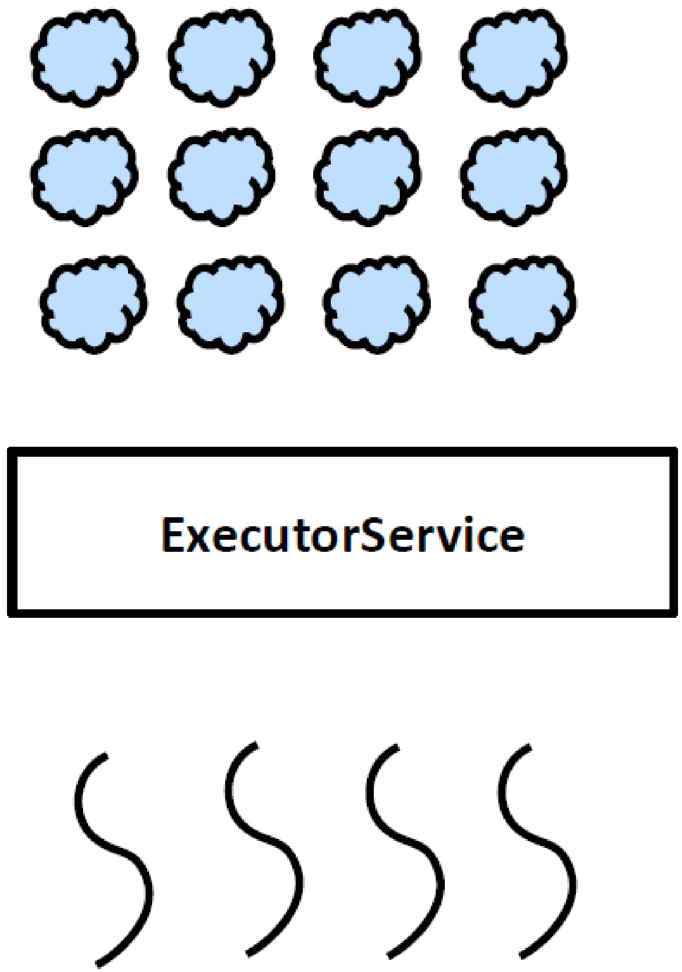
Threads vs Tasks

- Cassical approach: one thread per task. When the programmer creates a parallel task, the programmer also explicitly creates a new thread
  - Thread.start(), Thread.join()
- Task-parallel (Task pool) approach: the programmer specifies parallel tasks (tasks that could be executed in parallel) and these tasks are executed (scheduled) on a number of threads that are available.
  - ***Similar with OpenMP Tasking!***

The Executor Framework

- The Executor Framework: an abstraction layer for managing the execution of tasks asynchronously in a multithreaded environment
- It decouples task submission from task execution!
- Programmers specify what can be executed concurrently rather than how it should be executed
- Allows control of Thread resources: a fixed number of threads are in a thread pool and only these threads are used
- Threads submit Tasks to a ThreadPoolExecutor, which choses Tasks and schedules them for execution to Threads

ExecutorService Manages Tasks



Using ExecutorService

- User submits tasks to ExecutorService
- Tasks can be:
  - Runnable: task does not return a result
    - Task executes method `void run()` of `Runnable`
  - Callable<T>: task returns a result of type T
    - Task executes method `T call()` of `Callable<T>`
- The submit method returns a `Future<?>` or `Future<T>` object, which can be used to retrieve the Callable return value and to manage the status of both Callable and Runnable tasks

QuickSort - Serial version

```
class SerialQuicksort {
    public static void sort(int[] a) {
        sort(a, 0, a.length - 1);
    }
    public static void sort(int[] a, int left, int right) {

        if (left < right) {
            int pivotIndex = QuicksortUtils.partition(a, left, right);
            sort(a, left, pivotIndex - 1);
            sort(a, pivotIndex + 1, right);
        }
    }
}
```

QuickSort – Parallelization idea

```
class SerialQuicksort {
    public static void sort(int[] a) {
        sort(a, 0, a.length - 1);
    }
    public static void sort(int[] a, int left, int right) {

        if (left < right) {
            int pivotIndex = QuicksortUtils.partition(a, left, right);
            sort(a, left, pivotIndex - 1);
            sort(a, pivotIndex + 1, right);
        }
    }
}
```

Main idea: do the two recursive calls in parallel!

If we create a thread for every recursive call, system gets flooded. Create a TASK instead

QuickSort – Parallel with Executor

```
Int[] a = ArrayUtils.generateArray(arraySize);

ExecutorService tpe = Executors.newFixedThreadPool(threadNum);

AtomicInteger inQueue = new AtomicInteger(0);

inQueue.incrementAndGet();

tpe.submit(new QuicksortRunnable(a, tpe,  inQueue));
```

inQueue: keeps track of number of tasks in pool, in order to shut down when finished!

Create a Pool and put the first task (corresponding to the first call of the recursive function) into the pool.

```java
class QuicksortRunnable implements Runnable {
    int[] a;
    int low;
    int high;
    final int SIZELIMIT = 1000;
    ExecutorService tpe;
    AtomicInteger inQueue;

    public QuicksortRunnable(int[] a, ExecutorService tpe, AtomicInteger inQueue) {
        this(a, 0, a.length - 1, tpe, inQueue);
    }

    public QuicksortRunnable(int[] a, int low, int high, ExecutorService tpe,
                                                                AtomicInteger inQueue) {
        this.a = a;
        this.low = low;
        this.high = high;
        this.tpe = tpe;
        this.inQueue=inQueue;
    }
```

```java
@Override
public void run() {
    if (low < high) {
        if (high - low < SIZELIMIT) {
            SerialQuicksort.sort(a, low, high);
        } else {

            int pivotIndex = QuicksortUtils.partition(a, low, high);

            QuicksortRunnable t1 = new QuicksortRunnable(a, low, pivotIndex - 1, tpe, inQueue);
            inQueue.incrementAndGet();
            tpe.submit(t1);

            QuicksortRunnable t2 = new QuicksortRunnable(a, pivotIndex + 1, high, tpe, inQueue);
            inQueue.incrementAndGet();
            tpe.submit(t2);
        }
    }
    int left = inQueue.decrementAndGet();
    if (left == 0) {
        tpe.shutdown();
    }
}
```

Must explicitly keep track of number of tasks in pool and shut down when finished, otherwise program will not terminate!

Example: Recursive Fibonacci

```
class SerialFibonacci {

   public static long fib(int n){
      if (n < 2)
         return n;
      long x1 = fib(n-1);
      long x2 = fib(n-2);
      return x1 + x2;
   }

   public static void main(String[] args) {
      int n=10;
      System.out.println("Fibo "+n+" "+fib(n));
   }
}
```

Fibonacci Parallelization with Tasks

```
public static long fib(int n){
    if (n < 2)
        return n;
    spawn task  fib(n-1);
    spawn task  fib(n-2);
    wait for tasks to complete, get task results x1 x2
    return x1+x2;
}
```

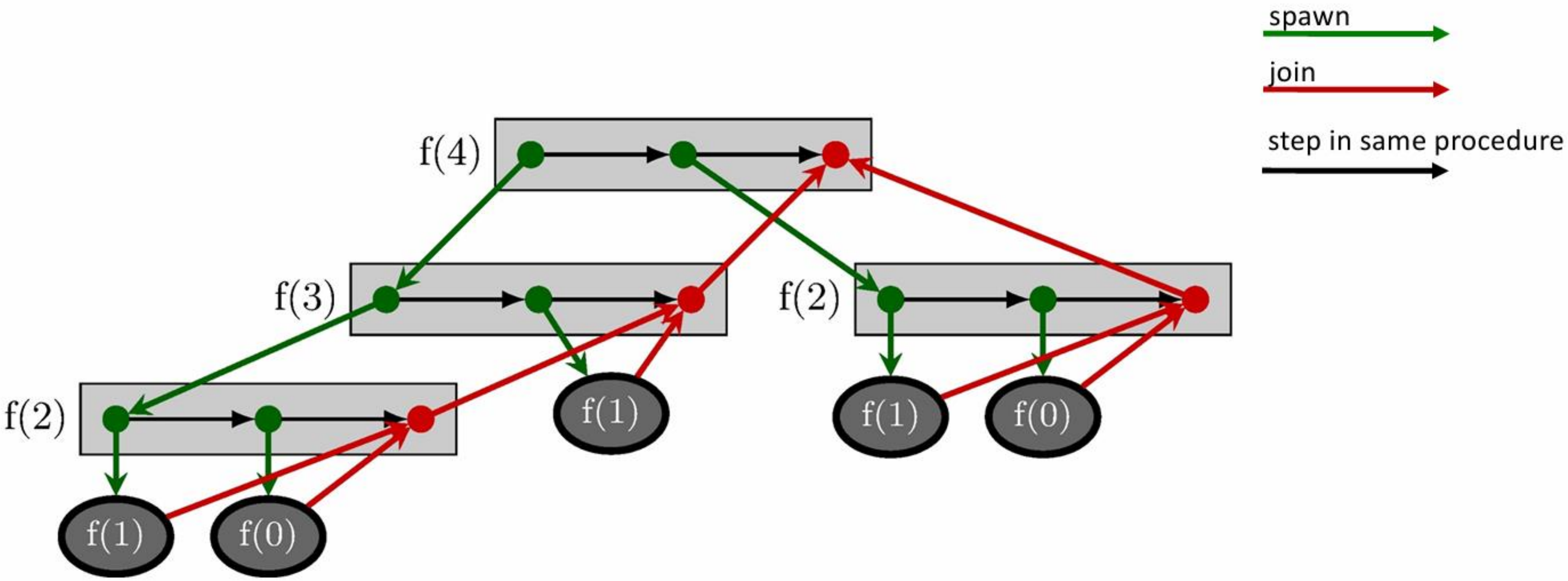
```java
class FibonacciTask implements Callable<Long> {
  private final int n;
  private final ExecutorService executor;

  public FibonacciTask(int n, ExecutorService executor) {
    this.n = n;
    this.executor = executor;
  }

  @Override
  public Long call() throws InterruptedException, ExecutionException {
    return fib(n, executor);  // Call the fib method recursively
  }

  private long fib(int n, ExecutorService executor) throws InterruptedException,
                                                              ExecutionException {
    if (n < 2) {
      return Long.valueOf(n);
    }
    FibonacciTask fib1 = new FibonacciTask(n - 1, executor);
    FibonacciTask fib2 = new FibonacciTask(n - 2, executor);
    Future<Long> result1 = executor.submit(fib1);
    Future<Long> result2 = executor.submit(fib2);
    return result1.get() + result2.get();
  }
}
```

```java
public static void main(String[] args) throws ExecutionException,
                                                InterruptedException {

    // Create a fixed thread pool with 5 threads
    ExecutorService executor = Executors.newFixedThreadPool(5);

    int n = 4; // call fib(4)

    Callable<Long> t = new FibonacciTask(n, executor);

    Future<Long> future = executor.submit(t);

    Long result = future.get() ;

    System.out.println("Fibonacci number #" + n + " is " + result);

    executor.shutdown();
}
```

```java
public static void main(String[] args) throws ExecutionException,
                                              InterruptedException {

    // Create a fixed thread pool with 4 threads
    ExecutorService executor = Executors.newFixedThreadPool(4);

    int n = 4; // call fib(4)

    Callable<Long> t = new FibonacciTask(n, executor);

    Future<Long> future = executor.submit(t);

    Long result = future.get() ;

    System.out.println("Fibonacci nu

    executor.shutdown();
}
```

This version will block because the tasks run out of available threads!

Task graph for fib(4)



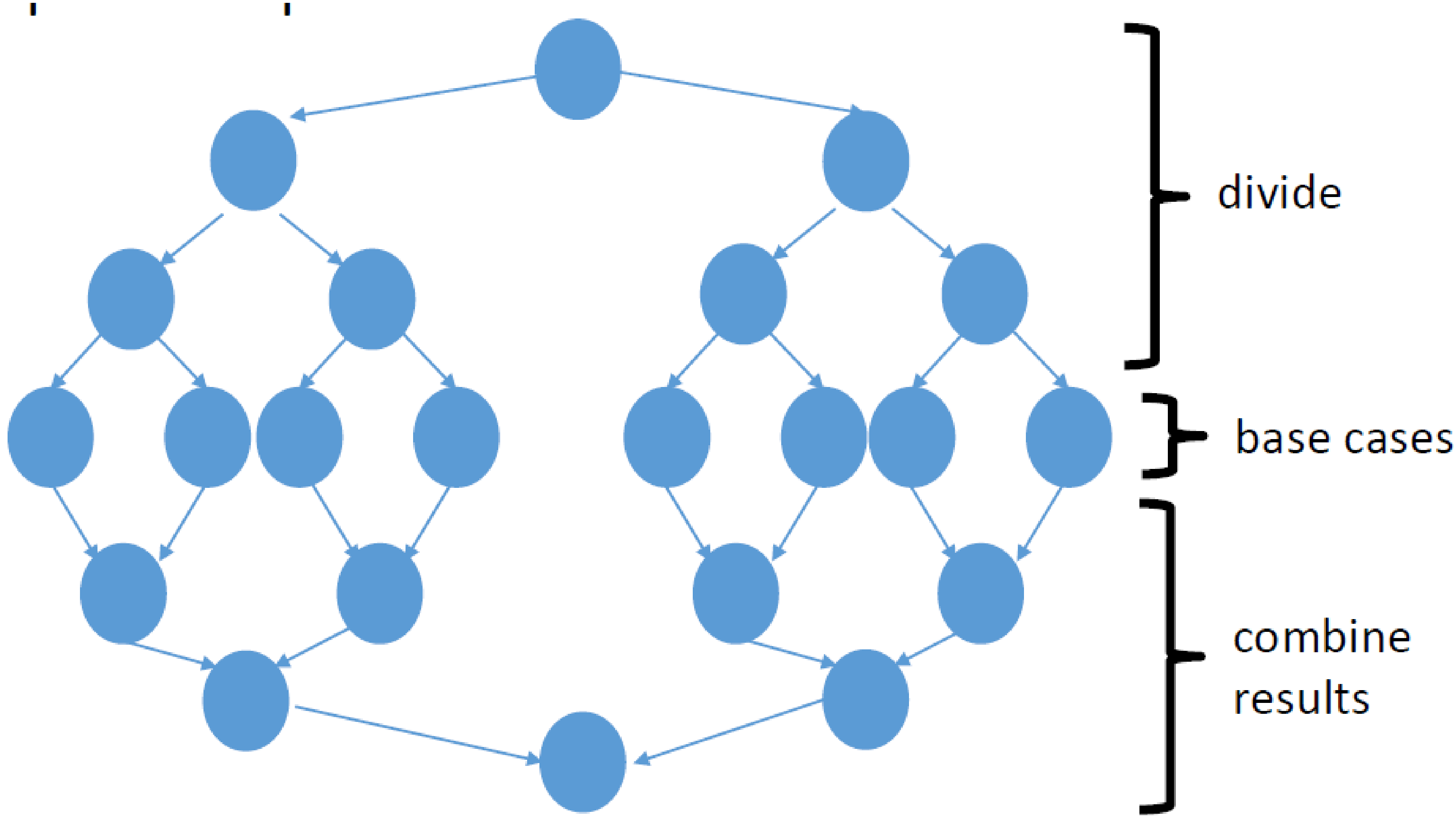
https://spcl.ethz.ch/Teaching/2020-pp/lectures/PP-L06-MODELS.pdf

```java
public static void main(String[] args) throws ExecutionException,
                                                InterruptedException {

    // Create a workstealing pool with 4 threads
    ExecutorService executor = Executors.newWorkStealingPool(4);

    int n = 20; // call fib(20)

    Callable<Long> t = new FibonacciTask(n, executo

    Future<Long> future = executor.sub

    Long result = future.get() ;

    System.out.println("Fibonac

    executor.shutdown();
}
```

No more blocking!
The WorkStealingPool allows its threads to create new tasks and suspend their current tasks when they wait for their child tasks to finish

ForkJoin Framework

- is designed to meet the needs of divide-and-conquer fork-join parallelism



Fork-Join Framework

- The fork/join framework is an implementation of the ExecutorService interface.
- It is designed for work that can be broken recursively into pieces
- As with any ExecutorService implementation, the fork/join framework distributes tasks to worker threads in a thread pool.
- The fork/join framework is distinct because it uses a work-stealing algorithm. Worker threads that run out of things to do can steal tasks from other threads that are still busy.
- The center of the fork/join framework is the ForkJoinPool class, an extension of the AbstractExecutorService class.
- ForkJoinPool can execute ForkJoinTask tasks
- pool.invoke(task) starts task, in current thread
- task.fork() – creates a new task and submits it to pool
- task.join() – waits for task to be finished
- ForkJoinTask subclasses:
  - RecursiveTask - can return a result
  - RecursiveAction

QuickSort - Serial version

```
class SerialQuicksort {
    public static void sort(int[] a) {
        sort(a, 0, a.length - 1);
    }
    public static void sort(int[] a, int left, int right) {

        if (left < right) {
            int pivotIndex = QuicksortUtils.partition(a, left, right);
            sort(a, left, pivotIndex - 1);
            sort(a, pivotIndex + 1, right);
        }
    }
}
```

QuickSort – Parallelization idea

```
class SerialQuicksort {
    public static void sort(int[] a) {
        sort(a, 0, a.length - 1);
    }
    public static void sort(int[] a, int left, int right) {

        if (left < right) {
            int pivotIndex = QuicksortUtils.partition(a, left, right);
            sort(a, left, pivotIndex - 1);
            sort(a, pivotIndex + 1, right);
        }
    }
}
```

Main idea: do the two recursive calls in parallel!

If we create a thread for every recursive call, system gets flooded. Create a TASK instead

QuickSort – parallel with ForkJoin

```
ForkJoinPool fjPool = new ForkJoinPool(threadNum);

ForkJoinQuicksortTask forkJoinQuicksortTask =
                                         new ForkJoinQuicksortTask(a, 0, a.length - 1);

fjPool.invoke(forkJoinQuicksortTask);
```

Create a Pool and put the first task (corresponding to the first call of the recursive function) into the pool

```
class ForkJoinQuicksortTask extends RecursiveAction {
    int[] a;
    int low, high;

    public ForkJoinQuicksortTask(int[] a) {
        this(a, 0, a.length - 1);
    }

    public ForkJoinQuicksortTask(int[] a, int low, int high) {
        this.a = a;     this.low = low;     this.high = high;
    }

    protected void compute() {
        if (low < high) {
               int pivotIndex = QuicksortUtils.partition(a, low, high);
                ForkJoinQuicksortTask t1 = new ForkJoinQuicksortTask(a, low, pivotIndex - 1);
                ForkJoinQuicksortTask t2 = new ForkJoinQuicksortTask(a, pivotIndex + 1, high
                t1.fork();
                t2.fork();
                t1.join();
                t2.join();
           }
        }
}
```

Basic solution
with tasks

Fork() does not
create a Thread,
but a Task!

```java
class ForkJoinQuicksortTask extends RecursiveAction {
    int[] a;
    int low, high;
    final int SIZELIMIT = 1000;

    public ForkJoinQuicksortTask(int[] a) {
        this(a, 0, a.length - 1);
    }

    public ForkJoinQuicksortTask(int[] a, int low, int high) {
        this.a = a;    this.low = low;    this.high = high;
    }

    protected void compute() {
        if (low < high) {
            if (high - low < SIZELIMIT) {
                SerialQuicksort.sort(a, low, high);
            } else {
                int pivotIndex = QuicksortUtils.partition(a, low, high);
                ForkJoinQuicksortTask t1 = new ForkJoinQuicksortTask(a, low, pivotIndex - 1);
                ForkJoinQuicksortTask t2 = new ForkJoinQuicksortTask(a, pivotIndex + 1, high);
                t1.fork();
                t2.compute();
                t1.join();
            }
        }
    }
}
```

Performance Improvements!

Quicksort Performance

8 threads/8 cores

	N=1M	N=2M	N=10M	N=20M	N=100M
Serial	Ts=84	Ts=172	Ts=892	Ts=1903	Ts=10011
Parallel with Executor	Tp=30 S=2.8	Tp=63 S=2.73	Tp=201 S=4.43	Tp=404 S=4.71	Tp=1988 S=5.03
Parallel with ForkJoin	Tp=21 S=4	Tp=39 S=4.41	Tp=178 S=5.01	Tp=360 S=5.28	Tp=2003 S=4.99

Source Code

- https://staff.cs.upt.ro/~ioana/apd/java/QuickSortVariants.java
- https://staff.cs.upt.ro/~ioana/apd/java/QuicksortUtils.java
- https://staff.cs.upt.ro/~ioana/apd/java/ArrayUtils.java

- See also: previous example of parallel Quicksort with OpenMP
- https://staff.cs.upt.ro/~ioana/apd/omp/omp_qsort.c

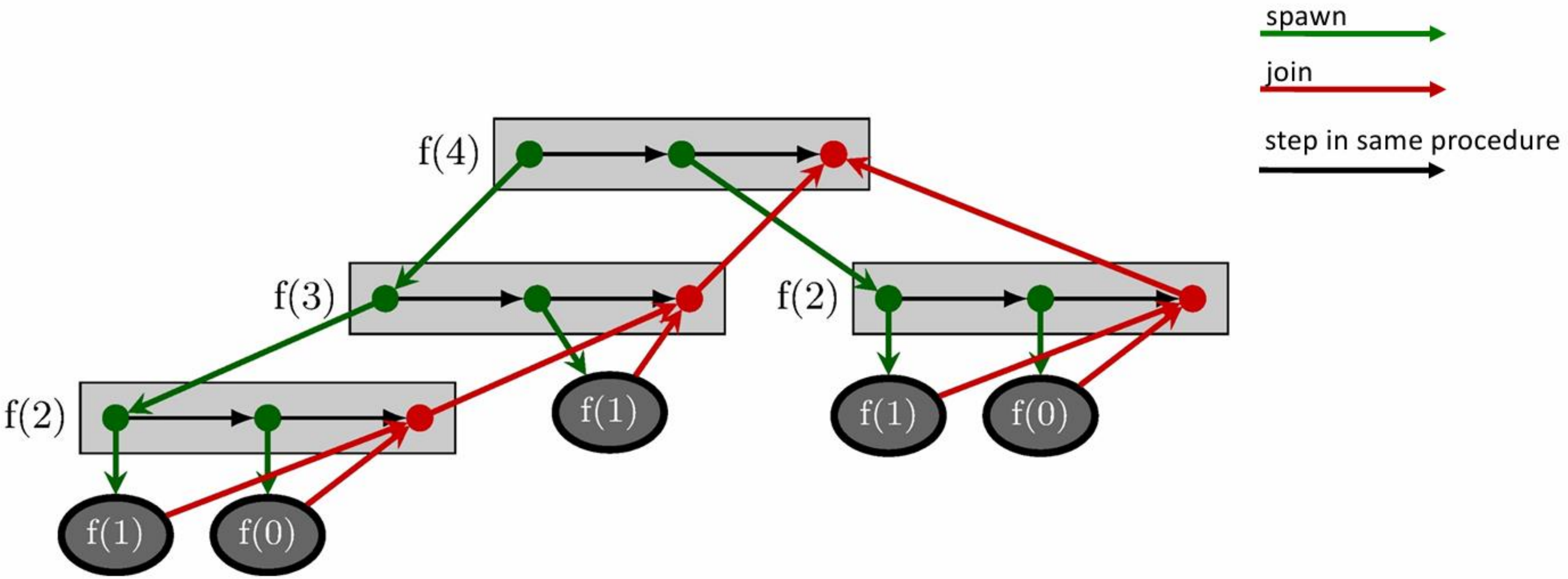
Example: Recursive Fibonacci

```
class SerialFibonacci {

   public static long fib(int n){
      if (n < 2)
         return n;
      long x1 = fib(n-1);
      long x2 = fib(n-2);
      return x1 + x2;
   }

   public static void main(String[] args) {
      int n=10;
      System.out.println("Fibo "+n+" "+fib(n));
   }
}
```

Fibonacci Parallelization with Tasks

```
public static long fib(int n){
   if (n < 2)
      return n;
   spawn task  fib(n-1);
   spawn task  fib(n-2);
   wait for tasks to complete, get task results x1 x2
   return x1+x2;
}
```

Task graph for fib(4)



https://spcl.ethz.ch/Teaching/2020-pp/lectures/PP-L06-MODELS.pdf

```java
public class FibonacciCalculator extends RecursiveTask<Long> {
    private final int n;

    public FibonacciCalculator(int n) {
        this.n = n;
    }

    @Override
    protected Long compute() {
        if (n <= 1) {
            return (long) n;
        } else {
            FibonacciCalculator fib1 = new FibonacciCalculator(n - 1);
            fib1.fork();

            FibonacciCalculator fib2 = new FibonacciCalculator(n - 2);
            fib2.fork();

            return fib2.join() + fib1.join();
        }
    }
```

```java
public static void main(String[] args) {
    ForkJoinPool forkJoinPool = new ForkJoinPool();
    FibonacciCalculator fibonacciCalculator = new FibonacciCalculator(30);
    long result = forkJoinPool.invoke(fibonacciCalculator);
    System.out.println("Result: " + result);
}
```