Futures, Scheduling, and Work Distribution

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

How to write Parallel Apps?

- How to
  - split a program into parallel parts
  - In an effective way
  - Thread management

Matrix Multiplication

$$(C) = (A) \bullet (B)$$

Matrix Multiplication

$$c_{ij} = \sum_{k=0}^{N-1} a_{ik} * b_{kj}$$

Matrix Multiplication

```
class Worker extends Thread {
  int row, col;
  Worker(int row, int col) {
    this.row = row;
    this.col = col;
  }
  public void run() {
    double dotProduct = 0.0;
    for (int i = 0; i < n; i++)
      dotProduct += a[row][i] * b[i][col];
    c[row][col] = dotProduct;
  }
}
```

Matrix Multiplication

```
class Worker extends Thread {
  int row, col;
  Worker(int row, int col) {
    this.row = row; this.col = col;
  }
  public void run() {
    double dotProduct = 0.0;
    for (int i = 0; i < n; i++)
      dotProduct += a[row][i] * b[i][col];
    c[row][col] = dotProduct;
  }}}
```

a thread

Matrix Multiplication

```
class Worker extends Thread {
  int row, col;
  Worker(int row, int col) {
    this.row = row; this.col = col;
  }
  public void run() {
    double dotProduct = 0.0;
    for (int i = 0; i < n; i++)
      dotProduct += a[row][i] * b[i][col];
    c[row][col] = dotProduct;
  }}}
```

Which matrix entry to compute

Matrix Multiplication

```
class Worker extends Thread {
  int row, col;
  Worker(int row, int col) {
    this.row = row; this.col = col;
  }
  public void run() {
    double dotProduct = 0.0;
    for (int i = 0; i < n; i++)
      dotProduct += a[row][i] * b[i][col];
    c[row][col] = dotProduct;
  }}}
```

Actual computation

Matrix Multiplication

```
void multiply() {
  Worker[][] worker = new Worker[n][n];
  for (int row ...)
    for (int col ...)
      worker[row][col] = new Worker(row,col);
  for (int row ...)
    for (int col ...)
      worker[row][col].start();
  for (int row ...)
    for (int col ...)
      worker[row][col].join();
}
```

Matrix Multiplication

```
void multiply() {
  Worker[][] worker = new Worker[n][n];
  for (int row ...)
    for (int col ...)
      worker[row][col] = new Worker(row,col);
  for (int row ...)
    for (int col ...)
      worker[row][col].start();
  for (int row ...)
    for (int col ...)
      worker[row][col].join();
}
```

Create nxn threads

Matrix Multiplication

```
void multiply() {
  Worker[][] worker = new Worker[n][n];
  for (int row ...)
    for (int col ...)
      worker[row][col] = new Worker(row,col);
  for (int row ...)
    for (int col ...)
      worker[row][col].start();
  for (int row ...)
    for (int col ...)
      worker[row][col].join();
}
```

Start them

Matrix Multiplication

```
void multiply() {
  Worker[][] worker = new Worker[n][n];
  for (int row ...)
    for (int col ...)
      worker[row][col] = new Worker(row,col);
  for (int row ...)
    for (int col ...)
      worker[row][col].start();
  for (int row ...)
    for (int col ...)
      worker[row][col].join();
}
```

Start them

Wait for them to finish

Matrix Addition

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00}+B_{00} & B_{01}+A_{01} \\ A_{10}+B_{10} & A_{11}+B_{11} \end{pmatrix}$$

Matrix Addition

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00}+B_{00} & B_{01}+A_{01} \\ A_{10}+B_{10} & A_{11}+B_{11} \end{pmatrix}$$

4 parallel additions

Matrix Addition Task

```
class AddTask implements Runnable {
  Matrix a, b; // multiply this!
  public void run() {
    if (a.dim == 1) {
      c[0][0] = a[0][0] + b[0][0]; // base case
    } else {
      (partition a, b into half-size matrices a_ij and b_ij)
      Future<?> f00 = exec.submit(add(a00,b00));
      ...
      Future<?> f11 = exec.submit(add(a11,b11));
      f00.get(); ...; f11.get();
      ...
    }}
```

This is not real Java Code

Matrix Addition Task

```
class AddTask implements Runnable {
  Matrix a, b; // multiply this!
  public void run() {
    if (a.dim == 1) {
      c[0][0] = a[0][0] + b[0][0]; // base case
    } else {
      (partition a, b into half-size matrices a_ij and b_ij)
      Future<?> f00 = exec.submit(add(a00,b00));
      ...
      Future<?> f11 = exec.submit(add(a11,b11));
      f00.get(); ...; f11.get();
      ...
    }}
```

Base case: add directly

Matrix Addition Task

```
class AddTask implements Runnable {
  Matrix a, b; // multiply this!
  public void run() {
    if (a.dim == 1) {
      c[0][0] = a[0][0] + b[0][0]; // base case
    } else {
      (partition a, b into half-size matrices a_ij and b_ij)
      Future<?> f00 = exec.submit(add(a00,b00));
      ...
      Future<?> f11 = exec.submit(add(a11,b11));
      f00.get(); ...; f11.get();
      ...
    }}
```

Constant-time operation

Matrix Addition Task

```
class AddTask implements Runnable {
  Matrix a, b; // multiply this!
  public void run() {
    if (a.dim == 1) {
      c[0][0] = a[0][0] + b[0][0]; // base case
    } else {
      (partition a, b into half-size matrices a_ij and b_ij)
      Future<?> f00 = exec.submit(add(a00,b00));
      ...
      Future<?> f11 = exec.submit(add(a11,b11));
      f00.get(); ...; f11.get();
      ...
    }}
```

Submit 4 tasks

Matrix Addition Task

```
class AddTask implements Runnable {
  Matrix a, b; // multiply this!
  public void run() {
    if (a.dim == 1) {
      c[0][0] = a[0][0] + b[0][0]; // base case
    } else {
      (partition a, b into half-size matrices a_ij and b_ij)
      Future<?> f00 = exec.submit(add(a00,b00));
      ...
      Future<?> f11 = exec.submit(add(a11,b11));
      f00.get(); ...; f11.get();
      ...
    }}
```

Let them finish

Dependencies

- Matrix example is not typical
- Tasks are independent
  - Don't need results of one task …
  - To complete another
- Often tasks are not independent

Fibonacci

- Note
  - potential parallelism
  - Dependencies

$$F(n) \begin{cases} 1 \text{ if } n = 0 \text{ or } 1 \\ F(n-1) + F(n-2) \text{ otherwise} \end{cases}$$

Multithreaded Fibonacci

```
class FibTask implements Callable<Integer> {
  static ExecutorService exec = Executors.newCachedThreadPool();
  int arg;
  public FibTask(int n) {
    arg = n;
  }
  public Integer call() {
    if (arg > 2) {
      Future<Integer> left  = exec.submit(new FibTask(arg-1));
      Future<Integer> right = exec.submit(new FibTask(arg-2));
      return left.get() + right.get();
    } else {
      return 1;
    }
  }
}
```

Multithreaded Fibonacci

```
class FibTask implements Callable<Integer> {
  static ExecutorService exec =
Executors.newCachedThreadPool();
  int arg;
  public FibTask(int n) {
    arg = n;
  }
  public Integer call() {
    if (arg > 2) {
      Future<Integer> left  = exec.submit(new FibTask(arg-1));
      Future<Integer> right = exec.submit(new FibTask(arg-2));
      return left.get() + right.get();
    } else {
      return 1;
    }}}
```

Parallel calls

Multithreaded Fibonacci

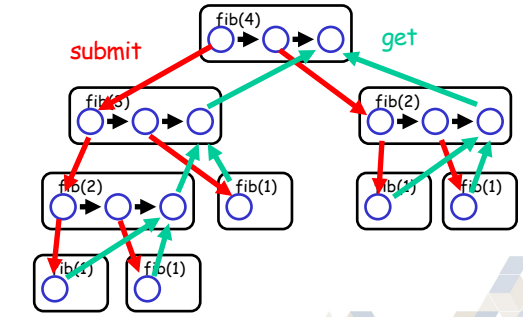
```
class FibTask implements Callable<Integer> {
  static ExecutorService exec =
Executors.newCachedThreadPool();
  int arg;
  public FibTask(int n) {
    arg = n;
  }
  public Integer call() {
    if (arg > 2) {
      Future<Integer> left  = exec.submit(new FibTask(arg-1));
      Future<Integer> right = exec.submit(new FibTask(arg-2));
      return left.get() + right.get();
    } else {
      return 1;
    }}}
```

Pick up & combine results

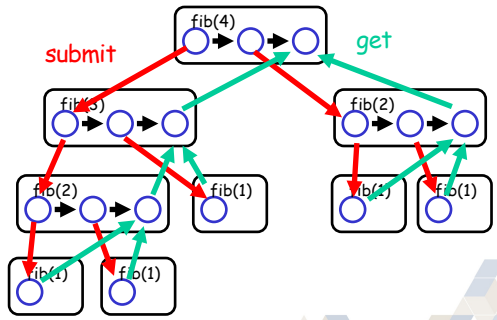
Dynamic Behavior

- Multithreaded program is
  - A directed acyclic graph (DAG)
  - That unfolds dynamically
- Each node is
  - A single unit of work

Fib DAG



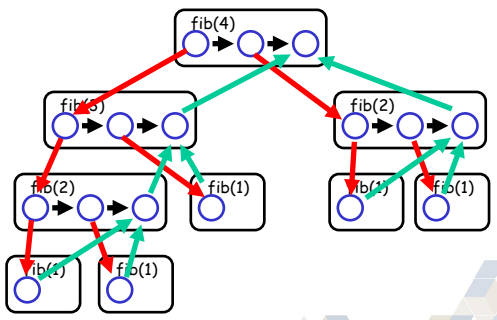
Arrows Reflect Dependencies



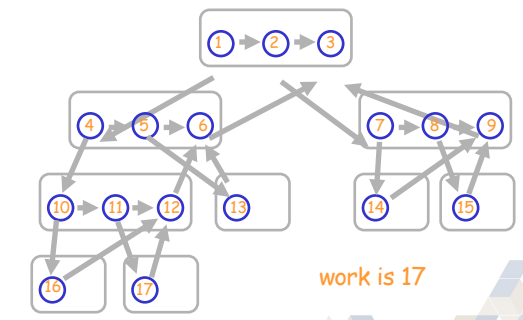
How Parallel is That?

- Define work:
  - Total time on one processor
- Define critical-path length:
  - Longest dependency path
  - Can't beat that!

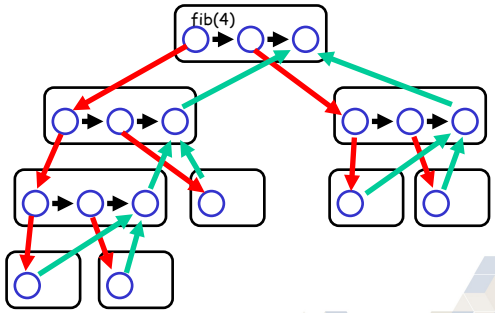
Fib Work



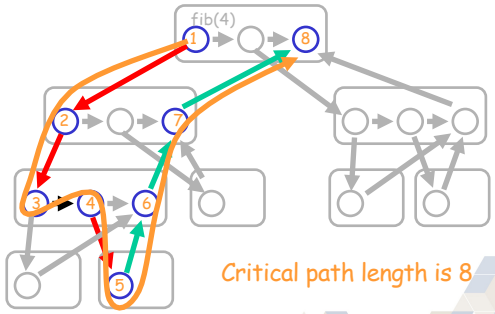
Fib Work



Fib Critical Path



Fib Critical Path



Notation Watch

- T_P = time on P processors
- T_1 = work (time on 1 processor)
- T_∞ = critical path length (time on ∞ processors)

Simple Bounds

- $T_P \geq T_\infty \geq T_1 / P$
  - In one step, can't do more than P work
  - Can't beat inherent sequentialism

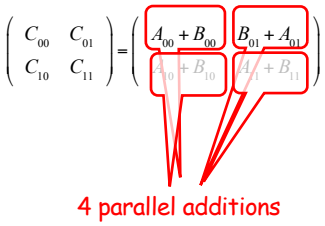
More Notation Watch

- Speedup on P processors
  - Ratio T_1 / T_P How much faster with P processors
- Linear speedup
  - $T_1 / T_P = \Theta(P)$
- Max speedup (average parallelism)
  - T_1 / T_∞

Matrix Addition

$$\begin{pmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{pmatrix} = \begin{pmatrix} A_{00} + B_{00} & B_{01} + A_{01} \\ A_{10} + B_{10} & A_{11} + B_{11} \end{pmatrix}$$

Matrix Addition



Addition

- Let $A_P(n)$ be running time
  - For n x n matrix
  - on P processors
- For example
  - $A_1(n)$ is work
  - $A_\infty(n)$ is critical path length

Addition

- Work is
  - $A_1(n) = 4\ A_1(n/2) + \Theta(1)$

Partition, synch, etc

4 spawned additions

Addition

- Work is
  - $A_1(n) = 4\ A_1(n/2) + \Theta(1)$
  - $= \Theta(n^2)$

Same as double-loop summation

Addition

- Critical Path length is
  - $A_\infty(n) = A_\infty(n/2) + \Theta(1)$

spawned additions in parallel

Partition, synch, etc

Addition

- Critical Path length is
  - $A_\infty(n) = A_\infty(n/2) + \Theta(1)$
  - $= \Theta(\log n)$

Matrix Multiplication Redux

$$(C) = (A) \bullet (B)$$

Matrix Multiplication Redux

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \bullet \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

First Phase …

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

8 multiplications

Second Phase …

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

4 additions

Multiplication

- Work is
  - $M_1(n) = 8\ M_1(n/2) + 4A_1(n/2)$
    $= 8\ M_1(n/2) + A_1(n)$

8 parallel multiplications

Final addition

Multiplication

- Work is
  - $M_1(n) = 8\ M_1(n/2) + \Theta(n^2)$
  - $= \Theta(n^3)$

Same as serial triple-nested loop

Multiplication

- Critical path length is

Final addition

$M_\infty(n) = M_\infty(n/2) + A_\infty(n)$

Half-size parallel multiplications

Multiplication

- Critical path length is
  - $M_\infty(n) = M_\infty(n/2) + A_\infty(n)$
  - $= M_\infty(n/2) + \Theta(\log n)$
  - $= \Theta(\log^2 n)$

Parallelism

- $M_1(n)/ M_\infty(n) = \Theta(n^3/\log^2 n)$
- To multiply two 1000 x 1000 matrices
  - $1000^3/10^2=10^7$
- Much more than number of processors on any real machine

Shared-Memory Multiprocessors

- Parallel applications
  - Do not have direct access to HW processors
- Mix of other jobs
  - All run together
  - Come & go dynamically

Matrix Multiplication

```
void multiply() {
  Worker[][] worker = new Worker[n][n];
  for (int row ...)
    for (int col ...)
      worker[row][col] = new Worker(row,col);
  for (int row ...)
    for (int col ...)
      worker[row][col].join();
}
```

Start them

What's wrong with this picture?

Wait for them to finish

Thread Overhead

- Threads Require resources
  - Memory for stacks
  - Setup, teardown
- Scheduler overhead
- Worse for short-lived threads

Thread Pools

- More sensible to keep a pool of long-lived threads
- Threads assigned short-lived tasks
  - Runs the task
  - Rejoins pool
  - Waits for next assignment

Thread Pool = Abstraction

- Insulate programmer from platform
  - Big machine, big pool
  - And vice-versa
- Portable code
  - Runs well on any platform
  - No need to mix algorithm/platform concerns

ExecutorService Interface

- In java.util.concurrent
  - Task = Runnable object
    - If no result value expected
    - Calls run() method.
  - Task = Callable<T> object
    - If result value of type T expected
    - Calls T call() method.

Future<T>

```
Callable<T> task = ...;
...
Future<T> future = executor.submit(task);
...
T value = future.get();
```

Future<T>

```
Callable<T> task = ...;
...
Future<T> future = executor.submit(task);
...
T value = future.get();
```

Submitting a **Callable<T>** task returns a **Future<T>** object

Future<T>

```
Callable<T> task = ...;
...
Future<T> future = executor.submit(task);
...
T value = future.get();
```

The Future's **get()** method blocks until the value is available

Future<?>

```
Runnable task = ...;
...
Future<?> future = executor.submit(task);
...
future.get();
```

Future<?>

```
Runnable task = ...;
...
Future<?> future = executor.submit(task);
...
future.get();
```

Submitting a **Runnable** task returns a **Future<?>** object

Future<?>

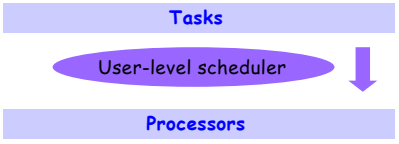
```
Runnable task = ...;
...
Future<?> future = executor.submit(task);
...
future.get();
```

The Future's **get()** method blocks until the computation is complete

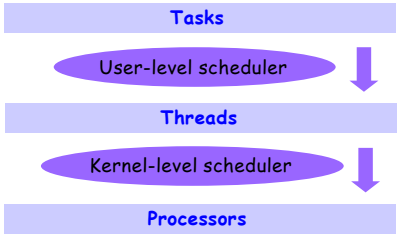
Note

- Executor Service submissions
  - Are purely advisory in nature
- The executor
  - Is free to ignore any such advice
  - And could execute tasks sequentially …

Ideal Scheduling Hierarchy



Realistic Scheduling Hierarchy



For Example

- Initially,
  - All P processors available for application
- Serial computation
  - Takes over one processor
  - Leaving P-1 for us
  - Waits for I/O
  - We get that processor back ….

Speedup

- Map threads onto P processes
- Cannot get P-fold speedup
  - What if the kernel doesn't cooperate?
- Can try for speedup proportional to
  - time-averaged number of processors
  - the kernel gives us

Scheduling Hierarchy

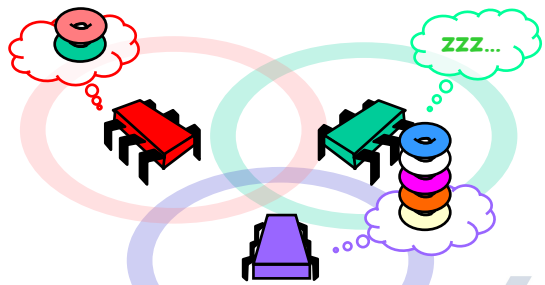
- User-level scheduler
  - Tells kernel which threads are ready
- Kernel-level scheduler
  - Synchronous (for analysis, not correctness!)
  - Picks pi threads to schedule at step i
  - Processor average
  - over T steps is:

$$P_A = \frac{1}{T}\sum_{i=1}^{T} p_i$$

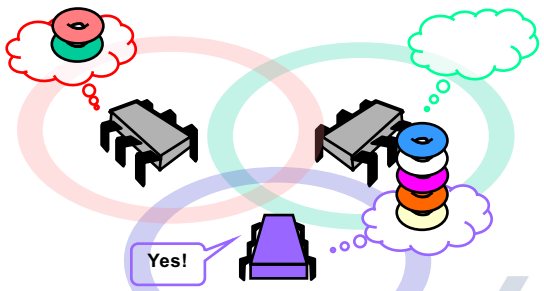
Greed is Good

- Greedy scheduler
  - Schedules as much as it can
  - At each time step
- Optimal schedule is greedy
- But not every greedy schedule is optimal

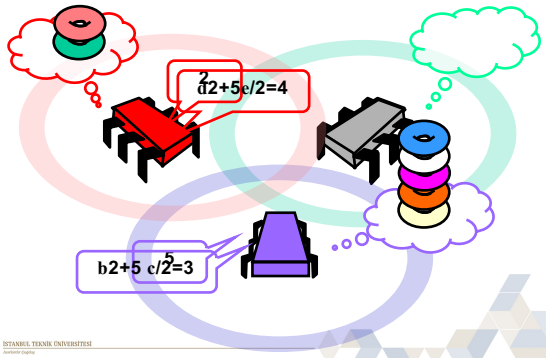
Work Distribution



Work Dealing



Work Balancing



Work-Balancing Thread

```
public void run() {
 int me = ThreadID.get();
 while (true) {
  Runnable task = queue[me].deq();
  if (task != null) task.run();
  int size = queue[me].size();
  if (random.nextInt(size+1) == size) {
   int victim = random.nextInt(queue.length);
   int min = …, max = …;
    synchronized (queue[min]) {
     synchronized (queue[max]) {
        balance(queue[min], queue[max]);
}}}}}
```

Work-Balancing Thread

```
public void run() {
 int me = ThreadID.get();
 while (true) {
  Runnable task = queue[me].deq();
  if (task != null) task.run();
  int size = queue[me].size();
  if (random.nextInt(size+1) == size) {
   int victim = random.nextInt(queue.length);
   int min = …, max = …;
    synchronized (queue[min]) {
     synchronized (queue[max]) {
        balance(queue[min], queue[max]);
}}}}}
```

Keep running tasks

Work-Balancing Thread

```
public void run() {
 int me = ThreadID.get();
 while (true) {
  Runnable task = queue[me].deq();
  if (task != null) task.run();
  int size = queue[me].size();
  if (random.nextInt(size+1) == size) {
   int victim = random.nextInt(queue.length);
   int min = …, max = …;
    synchronized (queue[min]) {
     synchronized (queue[max]) {
        balance(queue[min], queue[max]);
}}}}}
```

With probability 1/|queue|

Work-Balancing Thread

```
public void run() {
 int me = ThreadID.get();
 while (true) {
  Runnable task = queue[me].deq();
  if (task != null) task.run();
  int size = queue[me].size();
  if (random.nextInt(size+1) == size) {
   int victim = random.nextInt(queue.length);
   int min = …, max = …;
    synchronized (queue[min]) {
     synchronized (queue[max]) {
        balance(queue[min], queue[max]);
}}}}}
```

Choose random victim

Work-Balancing Thread

```
public void run() {
 int me = ThreadID.get();
 while (true) {
  Runnable task = queue[me].deq();
  if (task != null) task.run();
  int size = queue[me].size();
  if (random.nextInt(size+1) == size) {
   int victim = random.nextInt(queue.length);
   int min = …, max = …;
    synchronized (queue[min]) {
     synchronized (queue[max]) {
        balance(queue[min], queue[max]);
}}}}}
```

Lock queues in canonical order

Work-Balancing Thread

```
public void run() {
 int me = ThreadID.get();
 while (true) {
  Runnable task = queue[me].deq();
  if (task != null) task.run();
  int size = queue[me].size();
  if (random.nextInt(size+1) == size) {
   int victim = random.nextInt(queue.length);
   int min = …, max = …;
    synchronized (queue[min]) {
     synchronized (queue[max]) {
      balance(queue[min], queue[max]);
}}}}}
```

Rebalance queues

Starting Threads

- Executor framework

```
public interface Executor {
        void execute(Runnable command);
}
```

Executor Example

```
class MyTask implements Runnable {
  public void run() {…}
  public static void main(…) {
   Runnable task1 = new MyTask();
   Executor exec = Executors.newCachedThreadPool();
   exec.execute(task1);
  }
}
```

Provided Thread Pool Executors

- `newFixedThreadPool`
  - Bounded size
  - Replace if thread dies
- `newCachedThreadPool`
  - Demand driven variable size
- `newSingleThreadExecutor`
  - Just one thread
  - Replace if thread dies
- `newScheduledThreadPool`
  - Delayed and periodic task execution
  - Good replacement for class `Timer`

Executor Shut-down

- Executor is a serice provider
- Executor abstracts thread management
- To maintain the abstraction, the service should generally provide methods for shutting down the service
- JVM cannot shut down until threads do

ExecutorService

```
public interface ExecutorService extends Executor {
  void shutdown();
  List<Runnable> shutdownNow();
  boolean isShutdown();
  boolean isTerminated();
  boolean awaitTermination(long timeout, TimeUnit unit)
                  throws InterruptedException;
}
```

ExecutorService Notes

- Implies three states:
  - Running
  - Shutting down
  - Terminated
- `shutdown()` runs tasks in queue
- `shutdownNow()` returns unstarted tasks
- `awaitTermination()` blocks until the service is in the terminated state

ExecutorService implementation

- ExecutorService is an interface
- How do you implement shut down and cancellation?

Cancellation in Java

- *Cooperative*, not mandated
- Traditional method: *interruption*

```
public class Thread {
  public void interrupt();
  public void isInterrupted();
  public static boolean interrupted();
  …
}
```

Interruption

- Just a *request*!
- Sets a flag that must be explicitly cleared!
- Some methods clear the flag & throw *InterruptedException*
- Others *ignore* it, leaving other code to handle it

IMPORTANT!

- Your code should follow protocol when interrupted
- If *interrupted()*,
  - Throw exception
  - Reset the flag for other code
- If *InterruptedException*
  - Pass it along
  - Reset the flag for other code
- Don't swallow, unless your code handles the interruption policy

Restoring Interrupted Status

```
catch(InterruptedException e) {
  Thread.currentThread().interrupt();
}

// checks (AND CLEARS!) current thread
if (Thread.interrupted()) {
  Thread.currentThread().interrupt();
}
```

Handling Interruption

- Long computations "break" to check interrupted status
- If interrupted, stop computation, throw *InterruptedException* or restore the interrupted status

Interrupting Non-Blocking Operations

- Socket read/writes do not support interruption!
  - If you detect interruption, close the socket causing read/write to throw an exception
- Locks via synchronized can not be interrupted

Future

- Interface representing the *lifecycle* of a task

```
public interface Future<V> {
  boolean cancel(boolean mayInterruptIfRunning);
  boolean isCancelled();
  boolean isDone();
  V get() throws InterruptedException,
    ExecutionException, CancellationException;
  V get() throws InterruptedException,
    ExecutionException, CancellationException,
    TimeoutException;
}
```

CompletionService

- Combines `Executor` with `BlockingQueue`
- `ExecutorCompletionService`

```
public interface CompletionService<V> {
  Future<V> poll();
  Future<V> poll(long timeout, TimeUnit unit);
  Future<V> submit(Callable<V> task);
  Future<V> submit(Runnable task);
  Future<V> take();
}
```

Futures Again

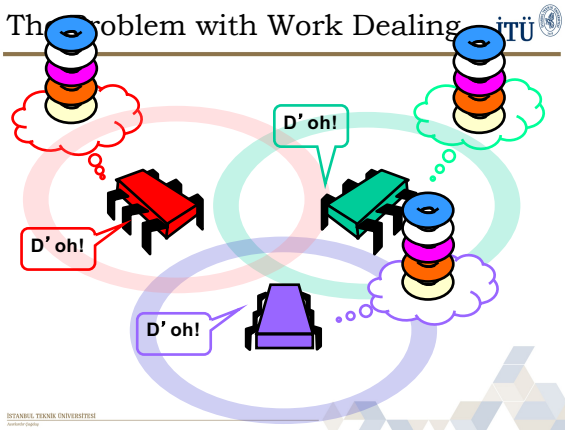
- *ExecutorService* interface also provides

```
public interface ExecutorService {
  …
  Future<T> submit(Callable<T> task);
  Future<?> submit(Runnable task);
  Future<T> submit(Runnable task, T result);
}
```

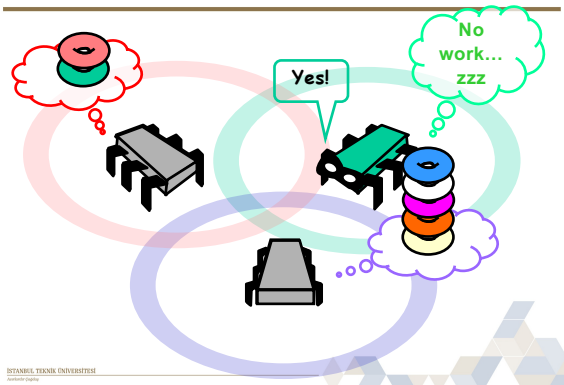
Executor Example

```
class WebServer {
Executor pool = Executors.newFixedThreadPool(7);
public static void main(String[] args) {
   ServerSocket soc = new ServerSocket(80);
   while (true) {
      Socket conn = soc.accept();
      Runnable r = new Runnable() {
            public void run() {
                  handleRequest(conn);
            }
      };
      pool.execute(r);
   }
}}
```

The problem with Work Dealing



Work Stealing

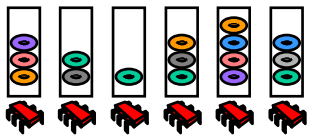


Lock-Free Work Stealing

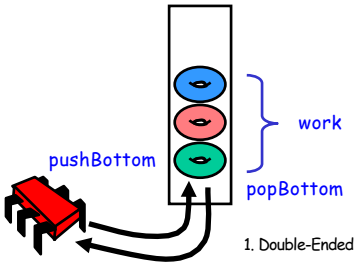
- Each thread has a pool of ready work
- Remove work without synchronizing
- If you run out of work, steal someone else's
- Choose victim at random

Local Work Pools

Each work pool is a Double-Ended Queue



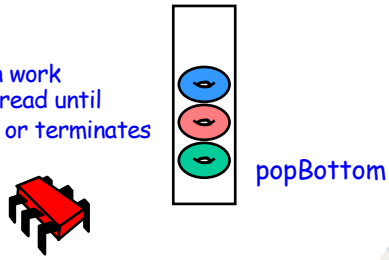
Work DEQueue1



1. Double-Ended Queue

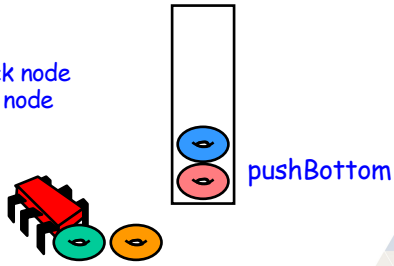
Obtain Work

- Obtain work
- Run thread until
- Blocks or terminates

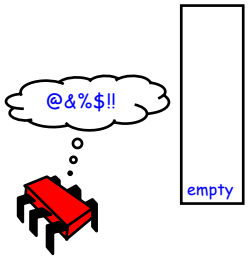


New Work

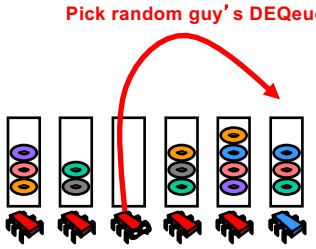
- Unblock node
- Spawn node



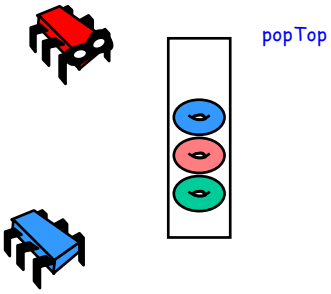
Whatcha Gonna do When the Well Runs Dry?



Steal Work from Others



Steal this Thread!



Thread DEQueue

- Methods
  - pushBottom
  - popBottom
  - popTop

Never happen concurrently

Thread DEQueue

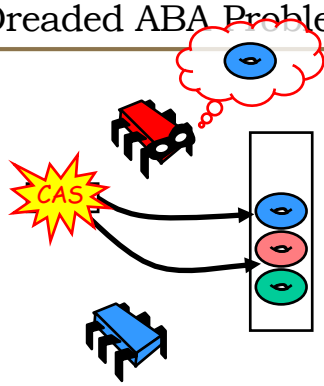
- Methods
  - pushBottom
  - popBottom
  - popTop

These most common - make them fast

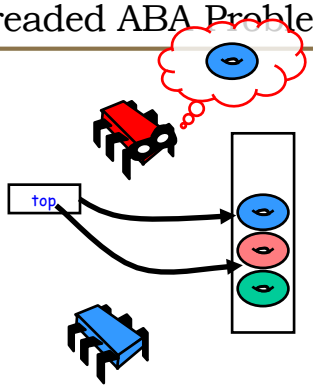
Compromise

- Method popTop may fail if
  - Concurrent popTop succeeds, or a
  - Concurrent popBottom takes last work

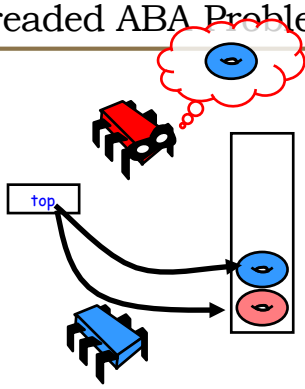
Dreaded ABA Problem



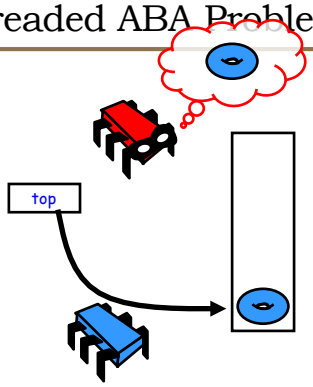
Dreaded ABA Problem



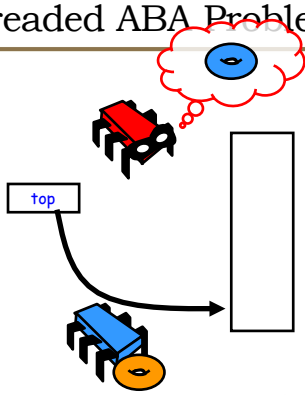
Dreaded ABA Problem



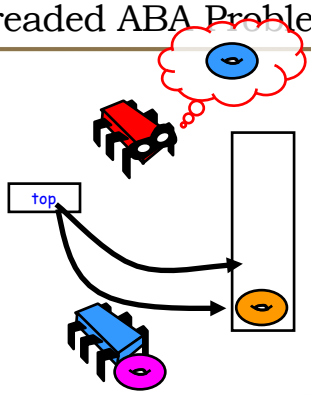
Dreaded ABA Problem



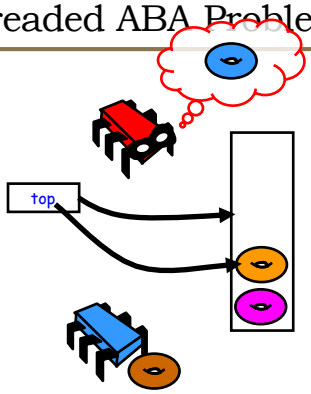
Dreaded ABA Problem



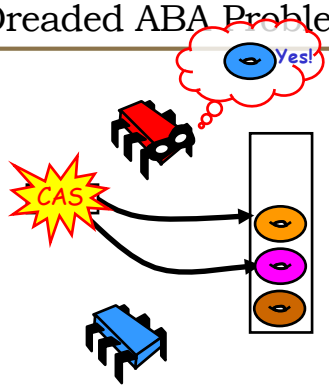
Dreaded ABA Problem



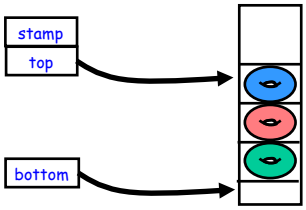
Dreaded ABA Problem



Dreaded ABA Problem



Fix for Dreaded ABA



Bounded DEQueue

```
public class BDEQueue {
 AtomicStampedReference<Integer> top;
 volatile int bottom;
 Runnable[] tasks;
 …
}
```

Bounded DQueue

```
public class BDEQueue {
 AtomicStampedReference<Integer> top;
 volatile int bottom;
 Runnable[] tasks;
 …
}
```

Index & Stamp
(synchronized)

Bounded DEQueue

```
public class BDEQueue {
 AtomicStampedReference<Integer> top;
 volatile int bottom;
 Runnable[] deq;
 …
}
```

Index of bottom thread (no need to synchronize The effect of a write must be seen - so we need a memory barrier)

Bounded DEQueue

```
public class BDEQueue {
 AtomicStampedReference<Integer> top;
 volatile int bottom;
 Runnable[] tasks;
 …
}
```

Array holding tasks

pushBottom()

```
public class BDEQueue {
 …
 void pushBottom(Runnable r){
  tasks[bottom] = r;
  bottom++;
 }
 …
}
```

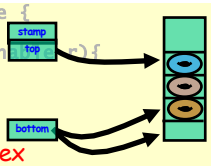
pushBottom()

```
public class BDEQueue {
 …
 void pushBottom(Runnable r){
  tasks[bottom] = r;
  bottom++;
 }
 …
}
```

Bottom is the index to store the new task in the array

pushBottom()

```
public class BDEQueue {
 …
 void pushBottom(Runnable r){
  tasks[bottom] = r;
  bottom++;
 }
 …
}
```



Adjust the bottom index

Steal Work

```
public Runnable popTop() {
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = oldTop + 1;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom <= oldTop)
   return null;
 Runnable r = tasks[oldTop];
 if (top.CAS(oldTop, newTop, oldStamp, newStamp))
  return r;
 return null;
 }
```

Steal Work

```
public Runnable popTop() {
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = oldTop + 1;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom <= oldTop)
   return null;
 Runnable r = tasks[oldTop];
 if (top.CAS(oldTop, newTop, oldStamp, newStamp))
  return r;
 return null;
 }
```

Read top (value & stamp)

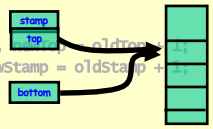
Steal Work

```
public Runnable popTop() {
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = oldTop + 1;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom <= oldTop)
   return null;
 Runnable r = tasks[oldTop];
 if (top.CAS(oldTop, newTop, oldStamp, newStamp))
  return r;
 return null;
 }
```

Compute new value & stamp

Steal Work

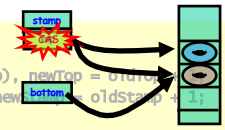
```
public Runnable popTop() {
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = oldTop + 1;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom <= oldTop)
   return null;
 Runnable r = tasks[oldTop];
 if (top.CAS(oldTop, newTop, oldStamp, newStamp))
  return r;
 return null;
 }
```



Quit if queue is empty

Steal Work

```
public Runnable popTop() {
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = oldTop + 1;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom <= oldTop)
   return null;
 Runnable r = tasks[oldTop];
 if (top.CAS(oldTop, newTop, oldStamp, newStamp))
  return r;
 return null;
 }
```



Try to steal the task

Steal Work

```
public Runnable popTop() {
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = oldTop + 1;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom <= oldTop)
   return null;
 Runnable r = tasks[oldTop];
 if (top.CAS(oldTop, newTop, oldStamp, newStamp))
  return r;
 return null;
 }
```

Give up if conflict occurs

Take Work

```
Runnable popBottom() {
 if (bottom == 0) return null;
 bottom--;
 Runnable r = tasks[bottom];
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = 0;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom > oldTop) return r;
 if (bottom == oldTop){
   bottom = 0;
   if (top.CAS(oldTop, newTop, oldStamp, newStamp))
     return r;
   }
 top.set(newTop,newStamp); return null;
}
```

Take Work

```
Runnable popBottom() {
 if (bottom == 0) return null;
 bottom--;
 Runnable r = tasks[bottom];
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = 0;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom > oldTop) return r;
 if (bottom == oldTop){
   bottom = 0;
   if (top.CAS(oldTop, newTop, oldStamp, newStamp))
     return r;
   }
 top.set(newTop,newStamp); return null;
}
```

Make sure queue is non-empty

Take Work

```
Runnable popBottom() {
 if (bottom == 0) return null;
 bottom--;
 Runnable r = tasks[bottom];
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = 0;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom > oldTop) return r;
 if (bottom == oldTop){
   bottom = 0;
   if (top.CAS(oldTop, newTop, oldStamp, newStamp))
     return r;
   }
 top.set(newTop,newStamp); return null;
}
```

Prepare to grab bottom task

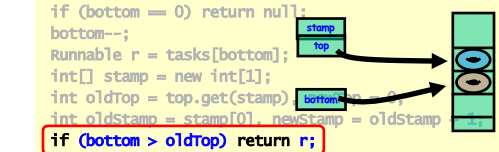
Take Work

```
Runnable popBottom() {
 if (bottom == 0) return null;
 bottom--;
 Runnable r = tasks[bottom];
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = 0;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom > oldTop) return r;
 if (bottom == oldTop){
   bottom = 0;
   if (top.CAS(oldTop, newTop, oldStamp, newStamp))
     return r;
   }
 top.set(newTop,newStamp); return null;
}
```

Read top, & prepare new values

Take Work

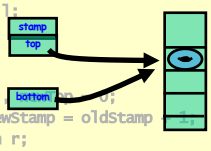
```
Runnable popBottom() {
 if (bottom == 0) return null;
 bottom--;
 Runnable r = tasks[bottom];
 int[] stamp = new int[1];
 int oldTop = top.get(stamp);
 if (bottom > oldTop) return r;
 if (bottom == oldTop){
   bottom = 0;
   if (top.CAS(oldTop, newTop, oldStamp, newStamp))
     return r;
   }
 return null;
}
```



If top & bottom 1 or more apart, no conflict

Take Work

```
Runnable popBottom() {
 if (bottom == 0) return null;
 bottom--;
 Runnable r = tasks[bottom];
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = 0;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom > oldTop) return r;
 if (bottom == oldTop){
   bottom = 0;
   if (top.CAS(oldTop, newTop, oldStamp, newStamp))
     return r;
   }
 top.set(newTop,newStamp); return null;
}
```



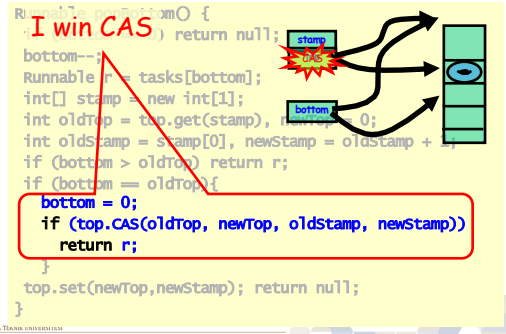
At most one item left

Take Work

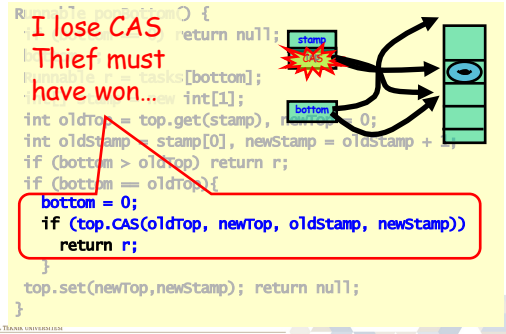
Try to steal last item.
In any case reset Bottom
because the DEQueue will be empty
even if unsucessful (why?)

```
if (bottom > oldTop) return r;
if (bottom == oldTop){
  bottom = 0;
  if (top.CAS(oldTop, newTop, oldStamp, newStamp))
    return r;
  }
top.set(newTop,newStamp); return null;
}
```

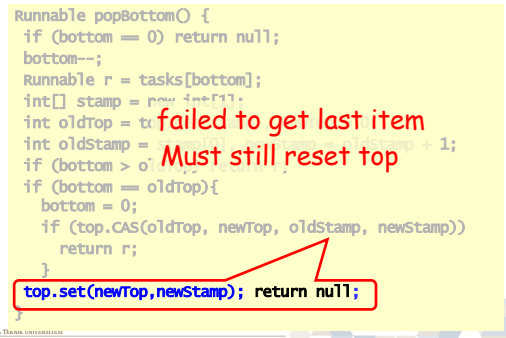
Take Work



Take Work



Take Work



Work Stealing & Balancing

- Clean separation between app & scheduling layer
- Works well when number of processors fluctuates.
- Works on “black-box” operating systems

Fork/Join Framework

- The fork/join framework was presented in Java 7.
- It provides **a divide and conquer approach**.

Fork/Join Framework

- In practice, this means that **the framework first “forks”**, recursively breaking the task into smaller independent subtasks until they are simple enough to be executed asynchronously.
- After that, **the “join” part begins**, in which results of all subtasks are recursively joined into a single result, or in the case of a task which returns void, the program simply waits until every subtask is executed.

Fork/Join Framework

- To provide effective parallel execution, the fork/join framework uses a pool of threads called the *ForkJoinPool*, which manages worker threads of type *ForkJoinWorkerThread*.

ForkJoinPool

- The ForkJoinPool is the heart of the framework. It is an implementation of the ExecutorService that manages worker threads and provides us with tools to get information about the thread pool state and performance.
- This architecture is vital for balancing the thread's workload with the help of the **work-stealing algorithm.**

ForkJoinPool

```
ForkJoinPool commonPool = ForkJoinPool.commonPool();

public static ForkJoinPool forkJoinPool
                              = new ForkJoinPool(2);

ForkJoinPool forkJoinPool = PoolUtil.forkJoinPool;
```

ForkJoinTask<V>

- ForkJoinTask is the base type for tasks executed inside ForkJoinPool. In practice, one of its two subclasses should be extended:
  - the RecursiveAction for void tasks and
  - the RecursiveTask<V> for tasks that return a value.
- They both have an abstract method compute() in which the task's logic is defined.

RecursiveAction

```
public class CustomRecursiveAction extends RecursiveAction {

    private String workload = "";
    private static final int THRESHOLD = 4;

    private static Logger logger =
      Logger.getAnonymousLogger();

    public CustomRecursiveAction(String workload) {
        this.workload = workload;
    }

    @Override
    protected void compute() {
        if (workload.length() > THRESHOLD) {
            ForkJoinTask.invokeAll(createSubtasks());
        } else {
            processing(workload);
        }
    }
```

RecursiveAction

```
    private List<CustomRecursiveAction> createSubtasks() {
        List<CustomRecursiveAction> subtasks = new ArrayList<>();

        String partOne = workload.substring(0, workload.length() / 2);
        String partTwo = workload.substring(workload.length() / 2, workload.length());

        subtasks.add(new CustomRecursiveAction(partOne));
        subtasks.add(new CustomRecursiveAction(partTwo));

        return subtasks;
    }

    private void processing(String work) {
        String result = work.toUpperCase();
        logger.info("This result - (" + result + ") - was processed by "
          + Thread.currentThread().getName());
    }
}
```

RecursiveTask<V>

```
public class MaxNumberCalculator extends RecursiveTask {

    public static final int THRESHOLD = 5;

    private int[] numbers;
    private int start;
    private int end;

    public MaxNumberCalculator(int[] numbers) {
        this(numbers, 0, numbers.length);
    }

    public MaxNumberCalculator(int[] numbers, int start, int end) {
        this.numbers = numbers;
        this.start = start;
        this.end = end;
    }
```


RecursiveTask<V>

```
    @Override
    public Integer compute() {
        int length = end - start;
        int max = 0;
        if (length < THRESHOLD) {
            for (int x = start; x < end; x++) {
                max = numbers[x];
            }
            return max;
        } else {
            int split = length / 2;
            MaxNumberCalculator left = new MaxNumberCalculator(numbers, start,
                start + split);
            left.fork();
            MaxNumberCalculator right = new MaxNumberCalculator(numbers, start
                + split, end);
            return Math.max(right.compute(), left.join());
        }
    }
}
```


ForkJoinPool

- To submit tasks to the thread pool, use the ***submit()*** or ***execute()*** method.

```
forkJoinPool.execute(customRecursiveTask);int result =
customRecursiveTask.join();
```

- The ***invoke() and invokeAll()*** method forks the task and waits for the result, and doesn't need any manual joining

```
int result = forkJoinPool.invoke(customRecursiveTask);
```

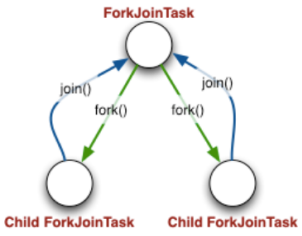
- Alternatively, you can use separate ***fork()*** **and** ***join()*** methods. The *fork()* method submits a task to a pool, but it doesn't trigger its execution. The *join()* method is be used for this purpose.

```
customRecursiveTaskFirst.fork();
result = customRecursiveTaskLast.join();
```


ForkJoinPool

```
    public static void main(String[] args) {

        final int[] numbers = new int[SIZE];
        int maxNum = 0;

        // Start sequential calculation
        long st = System.currentTimeMillis();

        for (int i = 0; i < SIZE; i++) {
            numbers[i] = (int) (Math.random() * 10000);
            if (numbers[i] > maxNum) {
                maxNum = numbers[i];
            }
        }

        System.out.println("Calculated maximum number (sequential execution): "
                + maxNum + " -- Total time: "
                + (System.currentTimeMillis() - st));

        // Start parallel calculation
        long pt = System.currentTimeMillis();

        ForkJoinPool pool = new ForkJoinPool(4);
        MaxNumberCalculator fbn = new MaxNumberCalculator(numbers);
        System.out.println("Calculated maximum number (parallel execution): "
                + pool.invoke(fbn) + " -- Total time: "
                + (System.currentTimeMillis() - pt));
    }
```


ForkJoinPool



Example: WordCount

```
class Document {
    private final List<String> lines;

    Document(List<String> lines) {
        this.lines = lines;
    }

    List<String> getLines() {
        return this.lines;
    }

    static Document fromFile(File file) throws IOException {
        List<String> lines = new LinkedList<>();
        try(BufferedReader reader = new BufferedReader(new FileReader(file))) {
            String line = reader.readLine();
            while (line != null) {
                lines.add(line);
                line = reader.readLine();
            }
        }
        return new Document(lines);
    }
}
```

Example: WordCount

```
class Folder {
    private final List<Folder> subFolders;
    private final List<Document> documents;

    Folder(List<Folder> subFolders, List<Document> documents) {
        this.subFolders = subFolders;
        this.documents = documents;
    }

    List<Folder> getSubFolders() {
        return this.subFolders;
    }

    List<Document> getDocuments() {
        return this.documents;
    }

    static Folder fromDirectory(File dir) throws IOException {
        List<Document> documents = new LinkedList<>();
        List<Folder> subFolders = new LinkedList<>();
        for (File entry : dir.listFiles()) {
            if (entry.isDirectory()) {
                subFolders.add(Folder.fromDirectory(entry));
            } else {
                documents.add(Document.fromFile(entry));
            }
        }
        return new Folder(subFolders, documents);
    }
}
```

Example: WordCount

Sequential Recursive Version

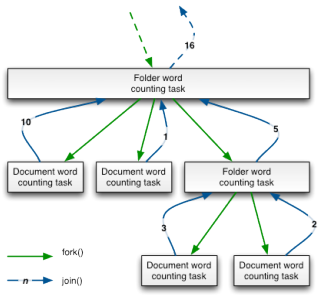
```
Long countOccurrencesOnSingleThread(Folder folder, String searchedWord) {
    long count = 0;
    for (Folder subFolder : folder.getSubFolders()) {
        count = count + countOccurrencesOnSingleThread(subFolder, searchedWord);
    }
    for (Document document : folder.getDocuments()) {
        count = count + occurrencesCount(document, searchedWord);
    }
    return count;
}
```

Example: WordCount

```
public class WordCounter {

    String[] wordsIn(String line) {
        return line.trim().split("(\\s|\\p{Punct})+");
    }

    Long occurrencesCount(Document document, String searchedWord) {
        long count = 0;
        for (String line : document.getLines()) {
            for (String word : wordsIn(line)) {
                if (searchedWord.equals(word)) {
                    count = count + 1;
                }
            }
        }
        return count;
    }
}
```

```
class DocumentSearchTask extends RecursiveTask<Long> {
    private final Document document;
    private final String searchedWord;

    DocumentSearchTask(Document document, String searchedWord) {
        super();
        this.document = document;
        this.searchedWord = searchedWord;
    }

    @Override
    protected Long compute() {
        return occurrencesCount(document, searchedWord);
    }
}
```

Example: WordCount



Example: WordCount

```
class FolderSearchTask extends RecursiveTask<Long> {
    private final Folder folder;
    private final String searchedWord;

    FolderSearchTask(Folder folder, String searchedWord) {
        super();
        this.folder = folder;
        this.searchedWord = searchedWord;
    }

    @Override
    protected Long compute() {
        long count = 0L;
        List<RecursiveTask<Long>> forks = new LinkedList<>();
        for (Folder subFolder : folder.getSubFolders()) {
            FolderSearchTask task = new FolderSearchTask(subFolder, searchedWord);
            forks.add(task);
            task.fork();
        }
        for (Document document : folder.getDocuments()) {
            DocumentSearchTask task = new DocumentSearchTask(document, searchedWord);
            forks.add(task);
            task.fork();
        }
        for (RecursiveTask<Long> task : forks) {
            count = count + task.join();
        }
        return count;
    }
}
```

Example: WordCount

```
private final ForkJoinPool forkJoinPool = new ForkJoinPool();

Long countOccurrencesInParallel(Folder folder, String searchedWord) {
    return forkJoinPool.invoke(new FolderSearchTask(folder, searchedWord));
}
```

```
public static void main(String[] args) throws IOException {
    WordCounter wordCounter = new WordCounter();
    Folder folder = Folder.fromDirectory(new File(args[0]));
    System.out.println(wordCounter.countOccurrencesOnSingleThread(folder, args[1]));
}
```

Example: WordCount

Number of Cores	Single-Thread Execution Time (ms)	Fork/Join Execution Time (ms)	Speedup
2	18798	11026	1.704879376
4	19473	8329	2.337975747
8	18911	4208	4.494058935
12	19410	2876	6.748956885