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

$$\left(\mathbf{C}\right) = \left(\mathbf{A}\right) \bullet \left(\mathbf{B}\right)$$

Matrix Multiplication

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

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 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

What's wrong with this picture?

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
  - Like New England traffic signs
  - Are purely advisory in nature
- The executor
  - Like the New England driver
  - Is free to ignore any such advice
  - And could execute tasks sequentially …

Matrix Addition

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

Matrix Addition

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

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<?> $f_{00}$ = exec.submit(add($a_{00}$,$b_{00}$));
     …
     Future<?> $f_{11}$ = exec.submit(add($a_{11}$,$b_{11}$));
     $f_{00}$.get(); …; $f_{11}$.get();
     …
    }}
```

This is not real Java code (see notes)

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 aij and bij)
    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 aij and bij)
     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<?> $f_{00}$ = exec.submit(add($a_{00}$,$b_{00}$));
      …
      Future<?> $f_{11}$ = exec.submit(add($a_{11}$,$b_{11}$));
      $f_{00}$.get(); …; $f_{11}$.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 aij and bij)
     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

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

- Note
  - potential parallelism
  - Dependencies

Disclaimer

- This Fibonacci implementation is
  - Egregiously inefficient
    - So don't deploy it!
  - But illustrates our point
    - How to deal with dependencies
- Exercise:
  - Make this implementation efficient!

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

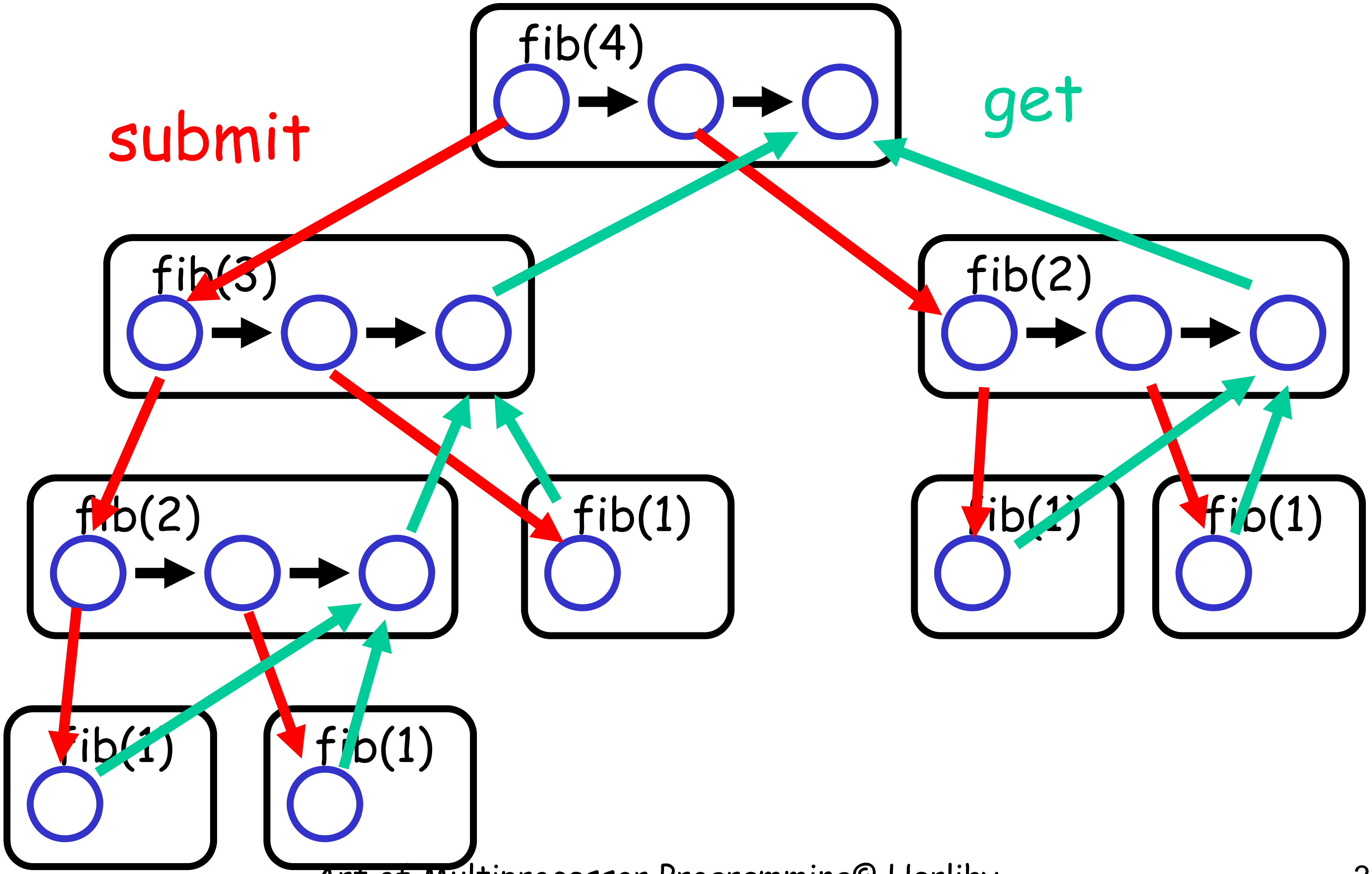
```
class FibTask implements Callable<Integer> {
  static ExecutorService exec =
Executors.newCachedThreadPool();
  int arg;
  public FibTask(int n) {
    arg = n;
  }
  public Integer call() {
    if (arg > 2) {
      Future<Integer> left  = exec.submit(new FibTask(arg-1));
      Future<Integer> right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

Pick up & combine results

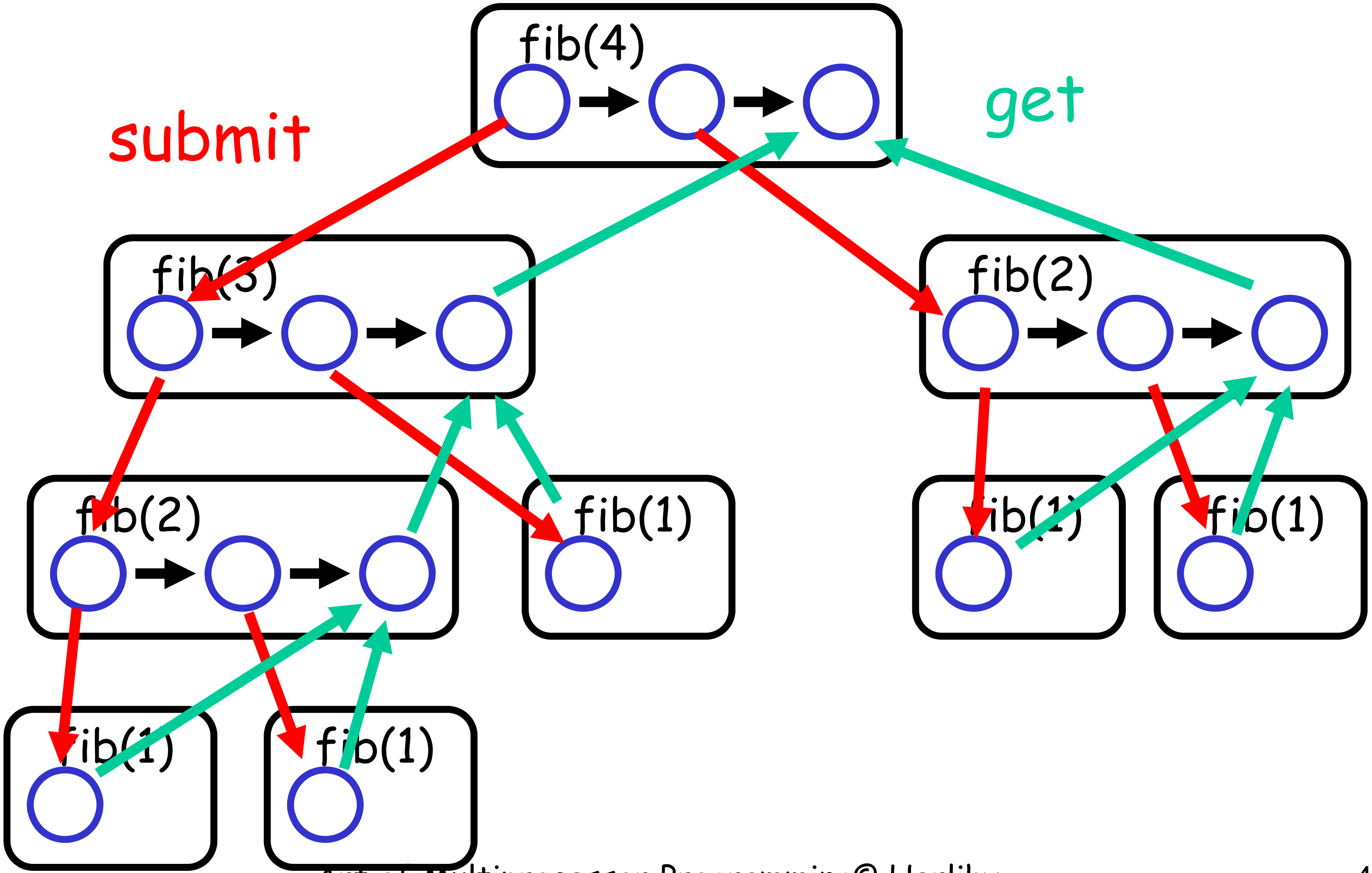
Dynamic Behavior

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

Fib DAG



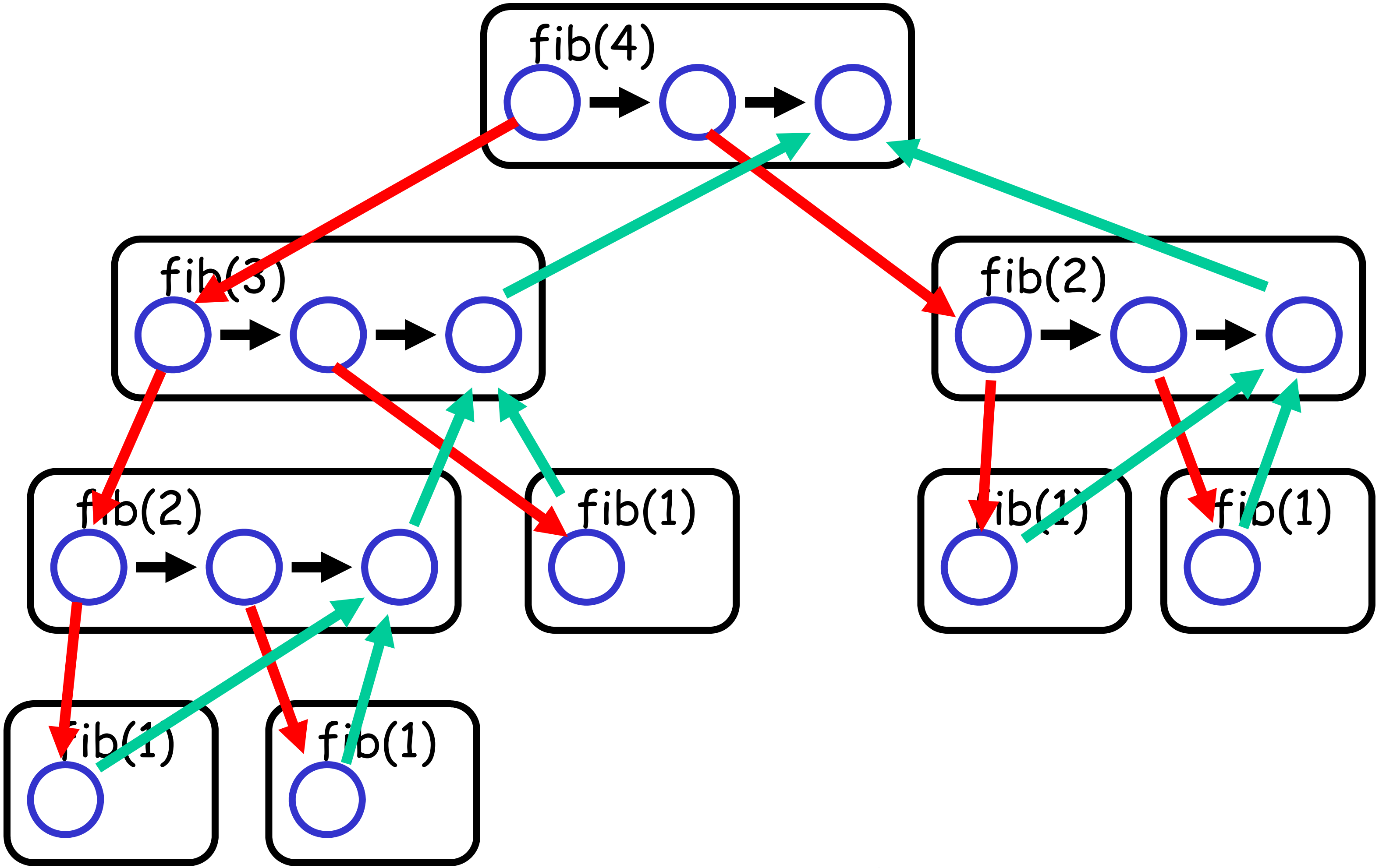
Arrows Reflect Dependencies



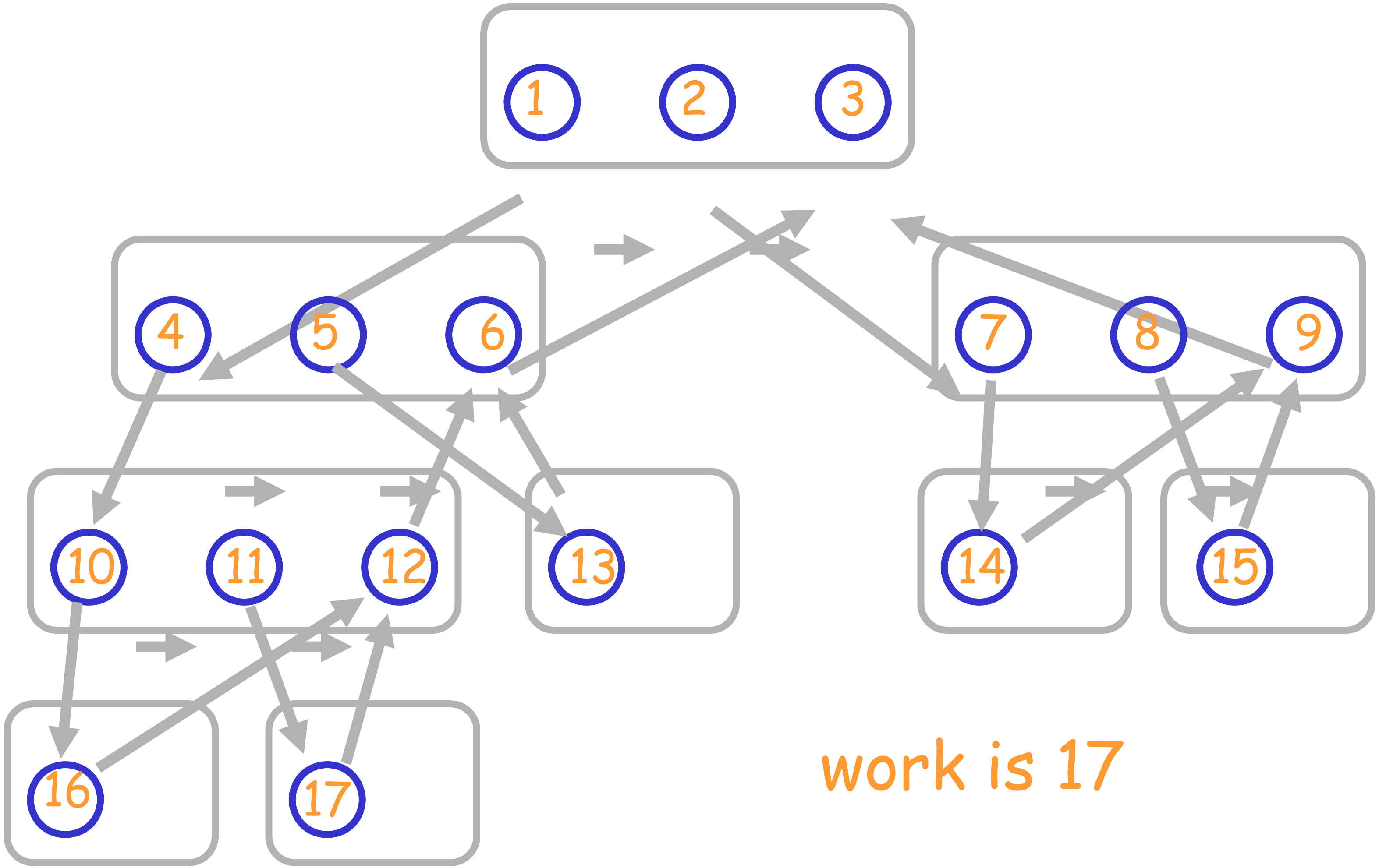
How Parallel is That?

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

Fib Work

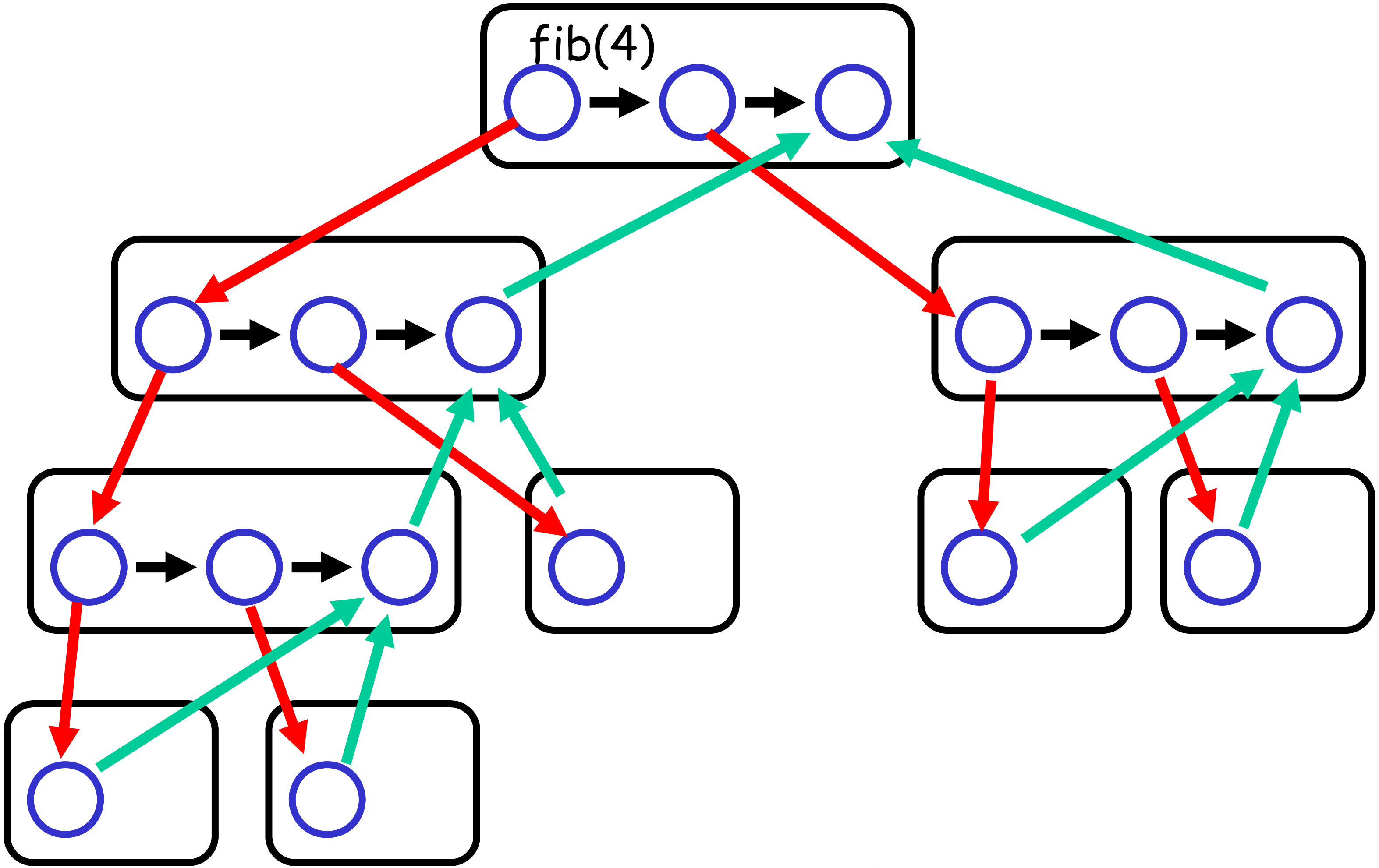


Fib Work

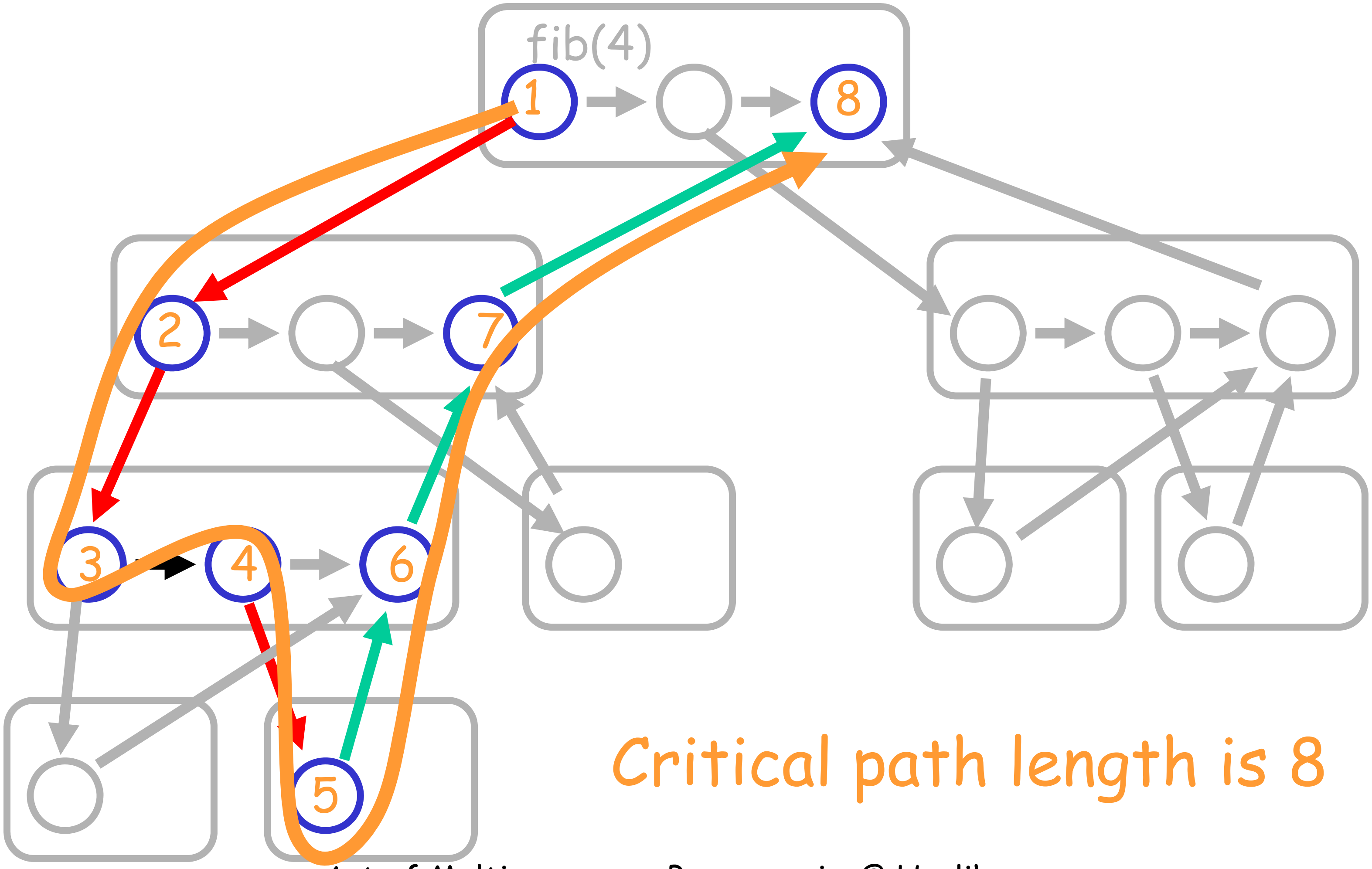


work is 17

Fib Critical Path



Fib Critical Path



Critical path length is 8

Notation Watch

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

Simple Bounds

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

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

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

Matrix Addition

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

4 parallel additions

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

Partition, synch, etc

$$A_1(n) = \boxed{4\ A_1(n/2)} + \boxed{\Theta(1)}$$

4 spawned additions

Addition

- Work is

$$\begin{aligned} A_1(n) &= 4\ A_1(n/2) + \Theta(1) \\ &= \Theta(n^2) \end{aligned}$$

Same as double-loop summation

Addition

- Critical Path length is

$$A_\infty(n) = \boxed{A_\infty(n/2)} + \boxed{\Theta(1)}$$

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 …

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

8 multiplications

Second Phase …

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

4 additions

Multiplication

- Work is

Final addition

$$M_1(n) = \boxed{8\ M_1(n/2)} + \boxed{A_1(n)}$$

8 parallel multiplications

Multiplication

- Work is

$$
\begin{aligned}
M_1(n) &= 8\, M_1(n/2) + \Theta(n^2) \\
&= \Theta(n^3)
\end{aligned}
$$

Same as serial triple-nested loop

Multiplication

- Critical path length is

Final addition

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

Half-size parallel multiplications

Multiplication

- Critical path length is

$$
\begin{aligned}
M_\infty(n) &= M_\infty(n/2) + A_\infty(n) \\
&= M_\infty(n/2) + \Theta(\log n) \\
&= \Theta(\log^2 n)
\end{aligned}
$$

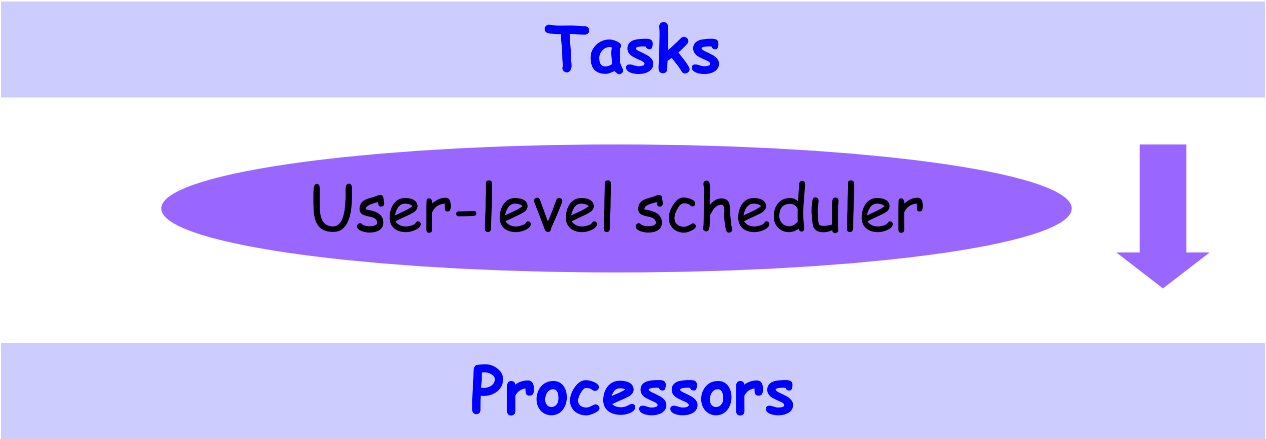
Parallelism

- $M_1(n)/ M_\infty(n) = \Theta(n^3/\log^2 n)$
- 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

Ideal Scheduling Hierarchy

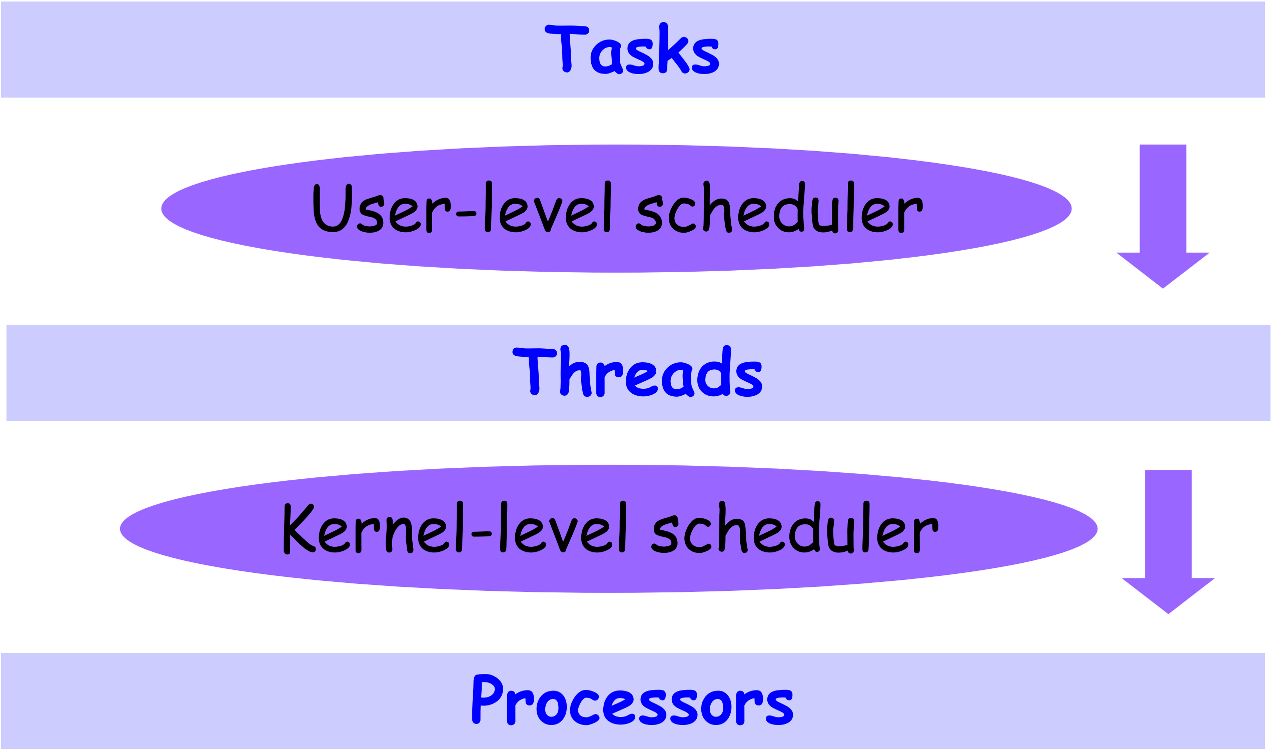


Tasks

User-level scheduler

Processors

Realistic Scheduling Hierarchy



Tasks

User-level scheduler

Threads

Kernel-level scheduler

Processors

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 $\mathbf{p_i}$ threads to schedule at step $\mathbf{i}$
  - 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 (why?)
- But not every greedy schedule is optimal

Theorem

- Greedy scheduler ensures that

$$T \leq T_1/P_A + T_\infty(P-1)/P_A$$

Deconstructing

$$T \leq T_1/P_A + T_\infty(P-1)/P_A$$

Deconstructing

$$\boxed{T} \le T_1/P_A + T_\infty(P-1)/P_A$$

Actual time

Deconstructing

$$T \leq \boxed{T_1/P_A} + T_\infty(P-1)/P_A$$

Work divided by processor average

Deconstructing

$$T \le T_1/P_A + \boxed{T_\infty}(P-1)/P_A$$

Cannot do better than critical path length

Deconstructing

$$T \le T_1/P_A + T_\infty \boxed{(P-1)/P_A}$$

The higher the average
the better it is …

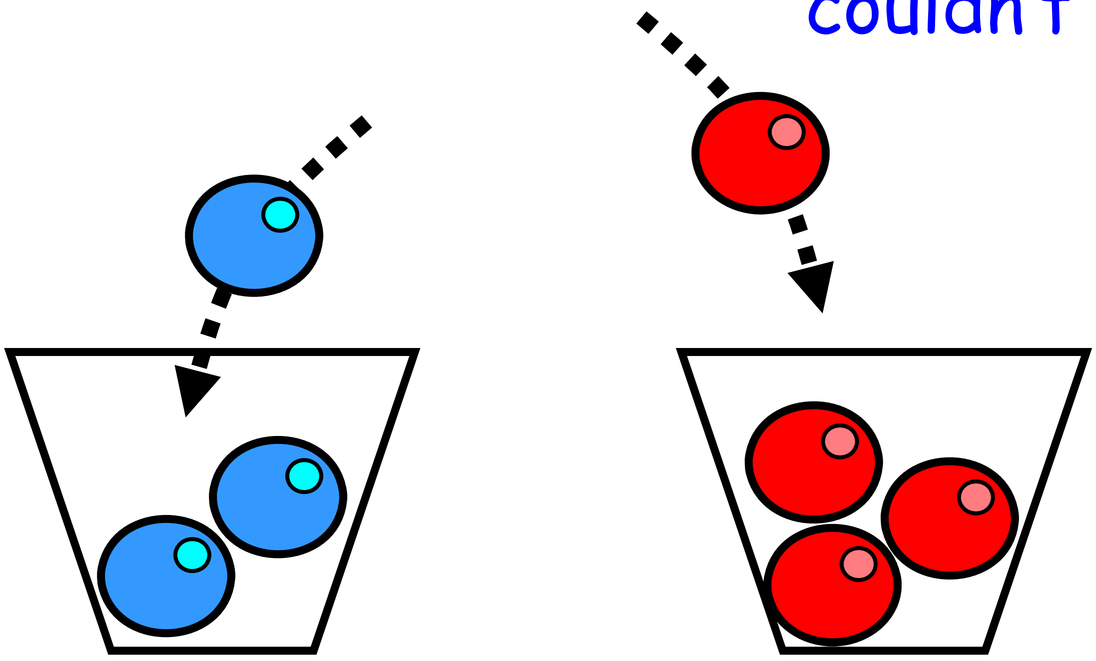
Proof Strategy

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

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

Bound this!

Put Tokens in Buckets



Processor found work

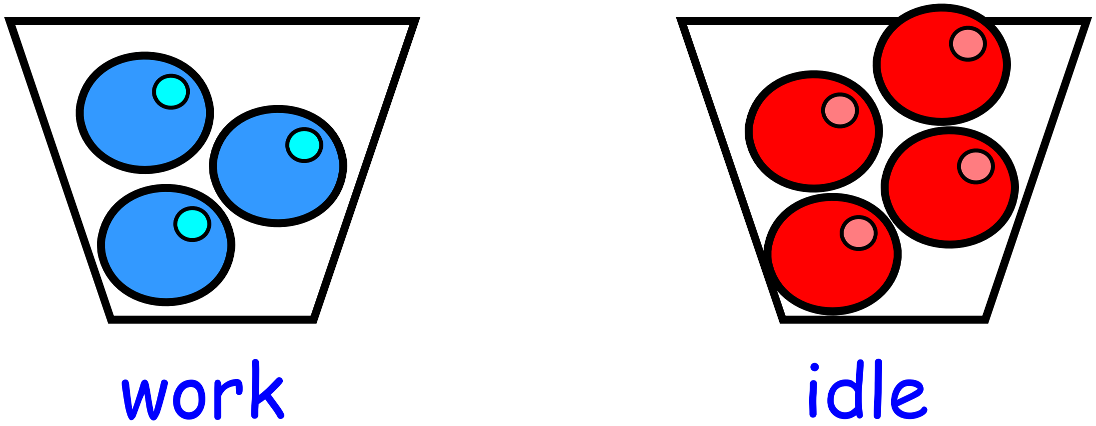
Processor available but couldn't find work

work

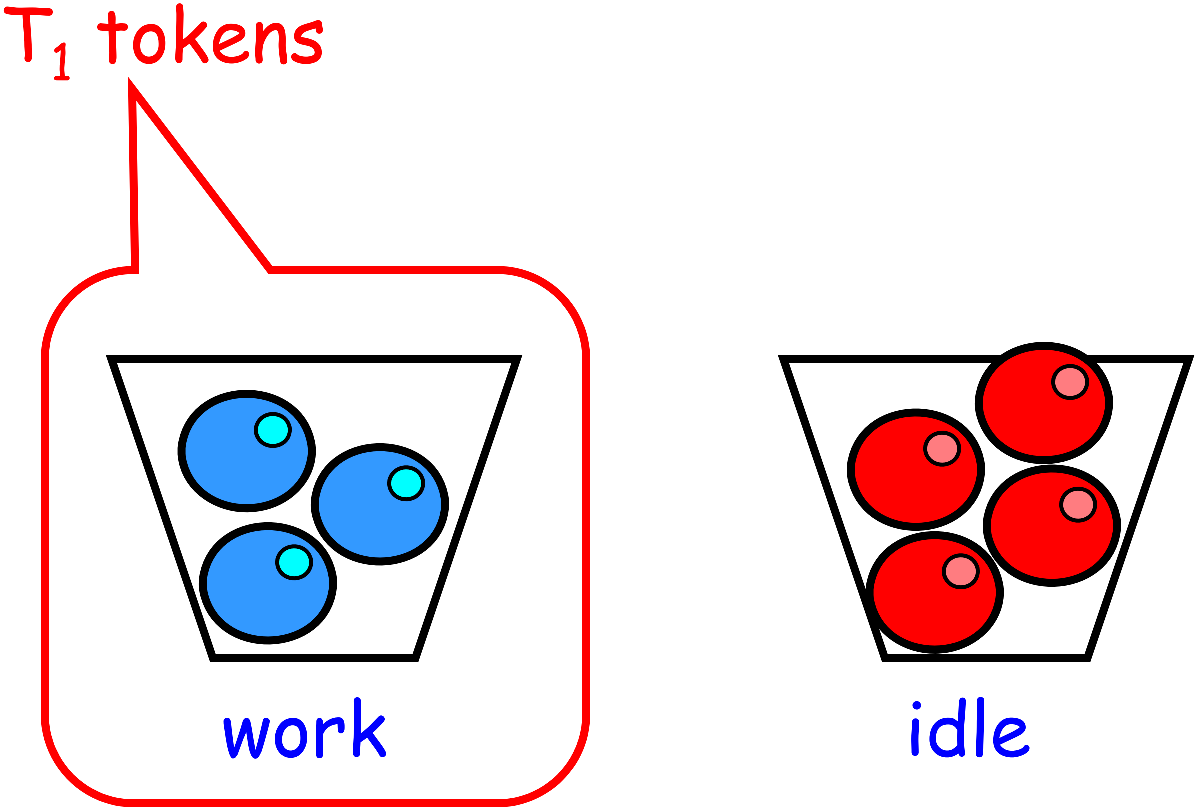
idle

At the end ….

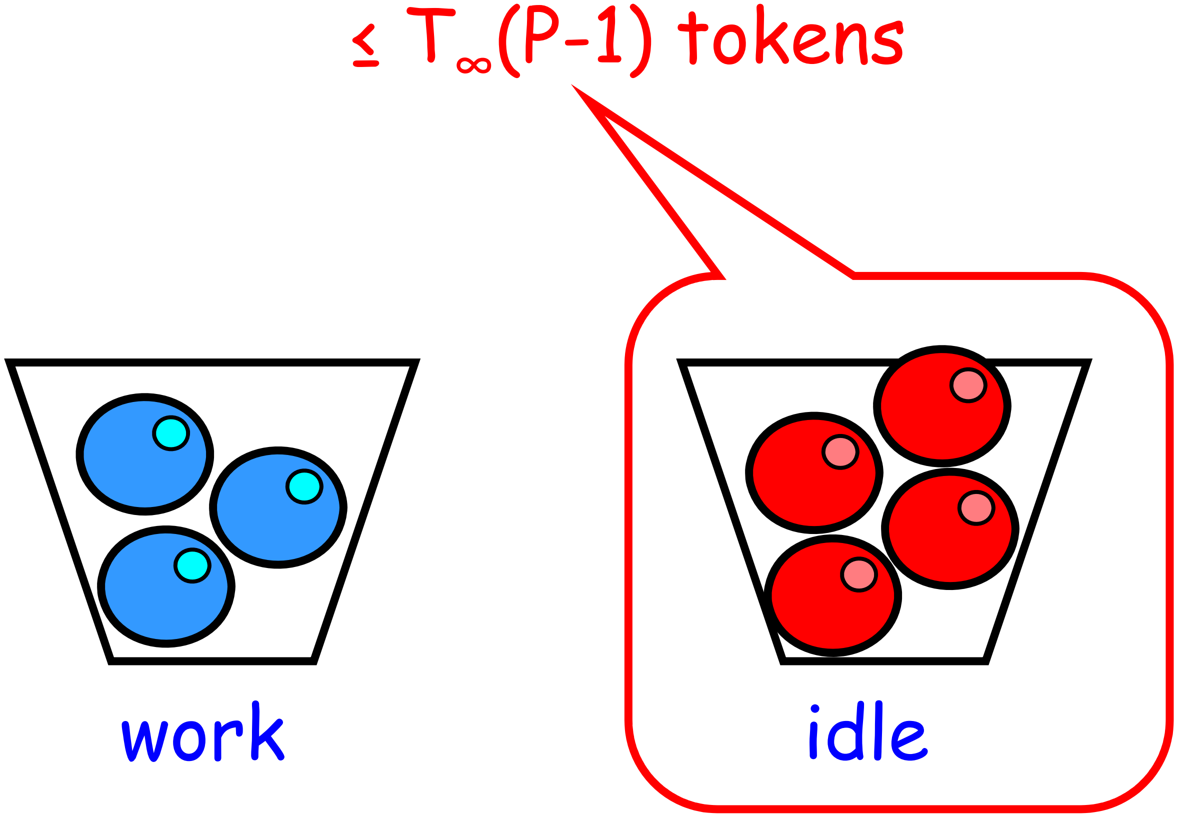
Total #tokens = $\sum_{i=1}^{T} \mathbf{p}_i$



At the end ….



Must Show



$\leq T_\infty(P-1)$ tokens

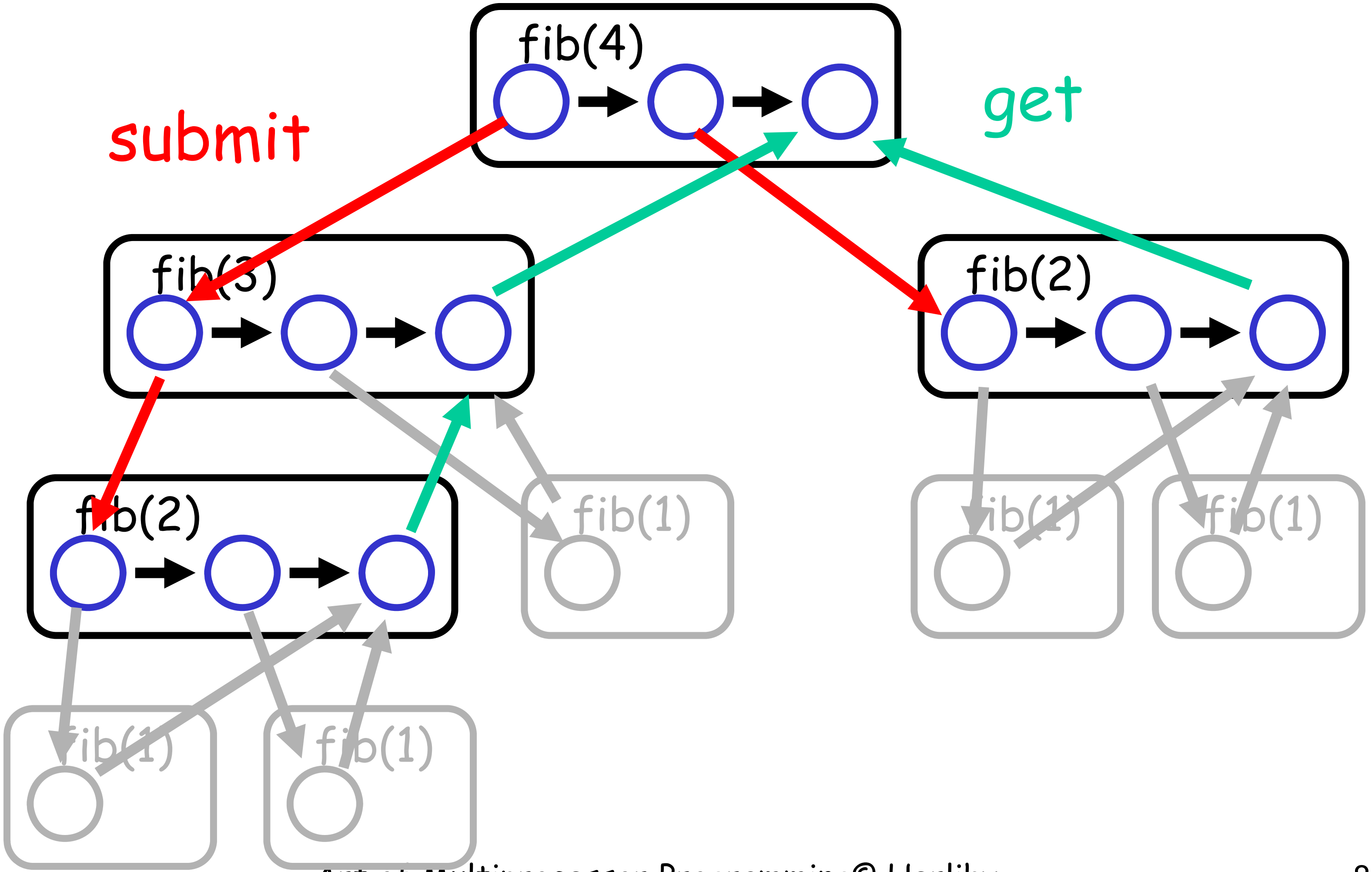
Idle Steps

- An idle step is one where there is at least one idle processor
- Only time idle tokens are generated
- Focus on idle steps

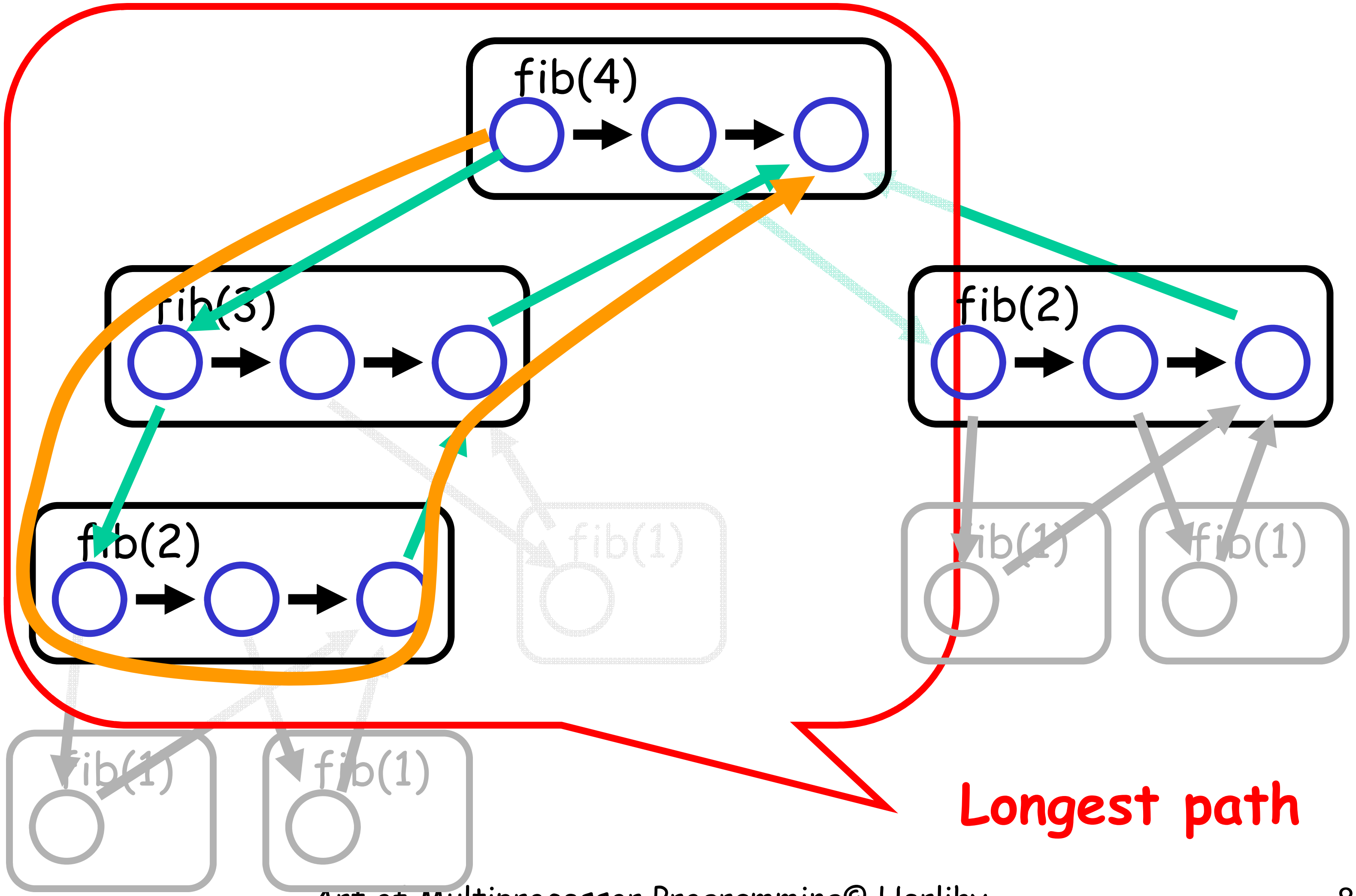
Every Move You Make …

- Scheduler is greedy
- At least one node ready
- Number of idle threads in one idle step
    - At most $p_i-1 \le P-1$
- How many idle steps?

Unexecuted sub-DAG

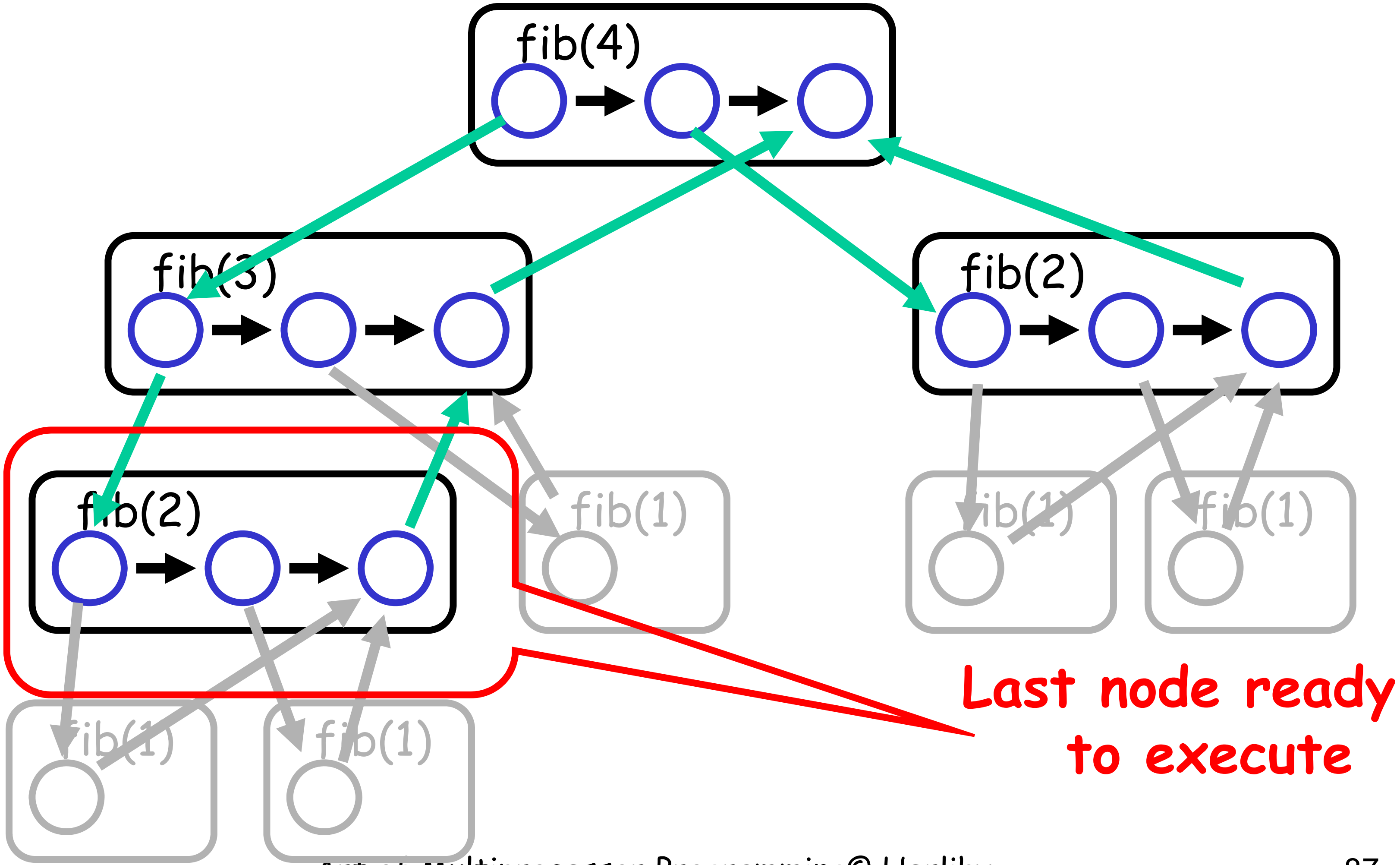


Unexecuted sub-DAG



Longest path

Unexecuted sub-DAG



Last node ready to execute

Every Step You Take …

- Consider longest path in unexecuted sub-DAG at step i
- At least one node in path ready
- Length of path shrinks by at least one at each step
- Initially, path is T_∞
- So there are at most T_∞ idle steps

Counting Tokens

- At most P-1 idle threads per step
- At most T_∞ steps
- So idle bucket contains at most
  - $T_\infty(P-1)$ tokens
- Both buckets contain
  - $T_1 + T_\infty(P-1)$ tokens

Recapitulating

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

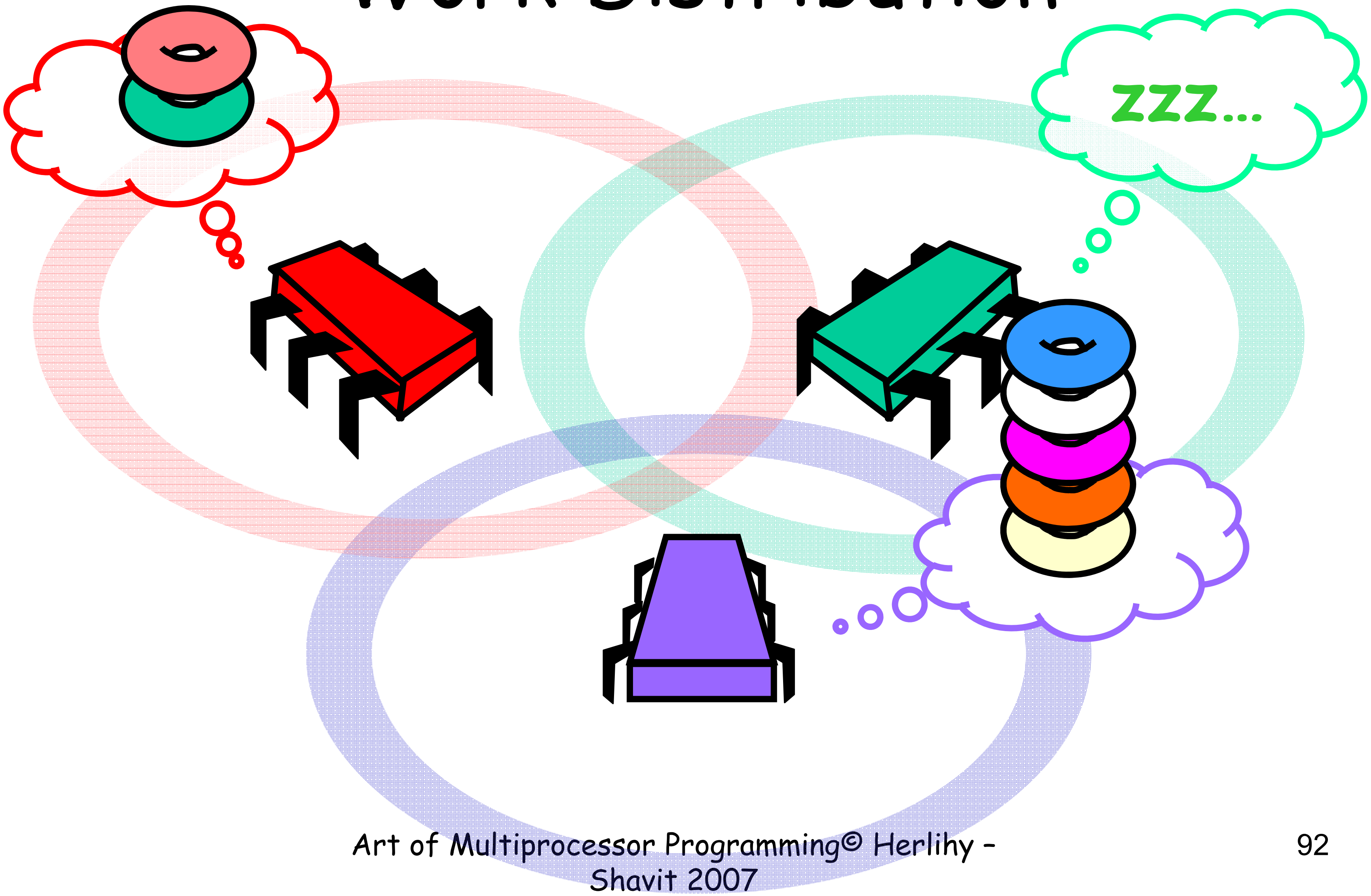
$$\sum_{i=1}^{T} p_i \le T_1 + T_\infty(P-1)$$

$$T \le \frac{1}{P_A}\left(T_1 + T_\infty(P-1)\right)$$

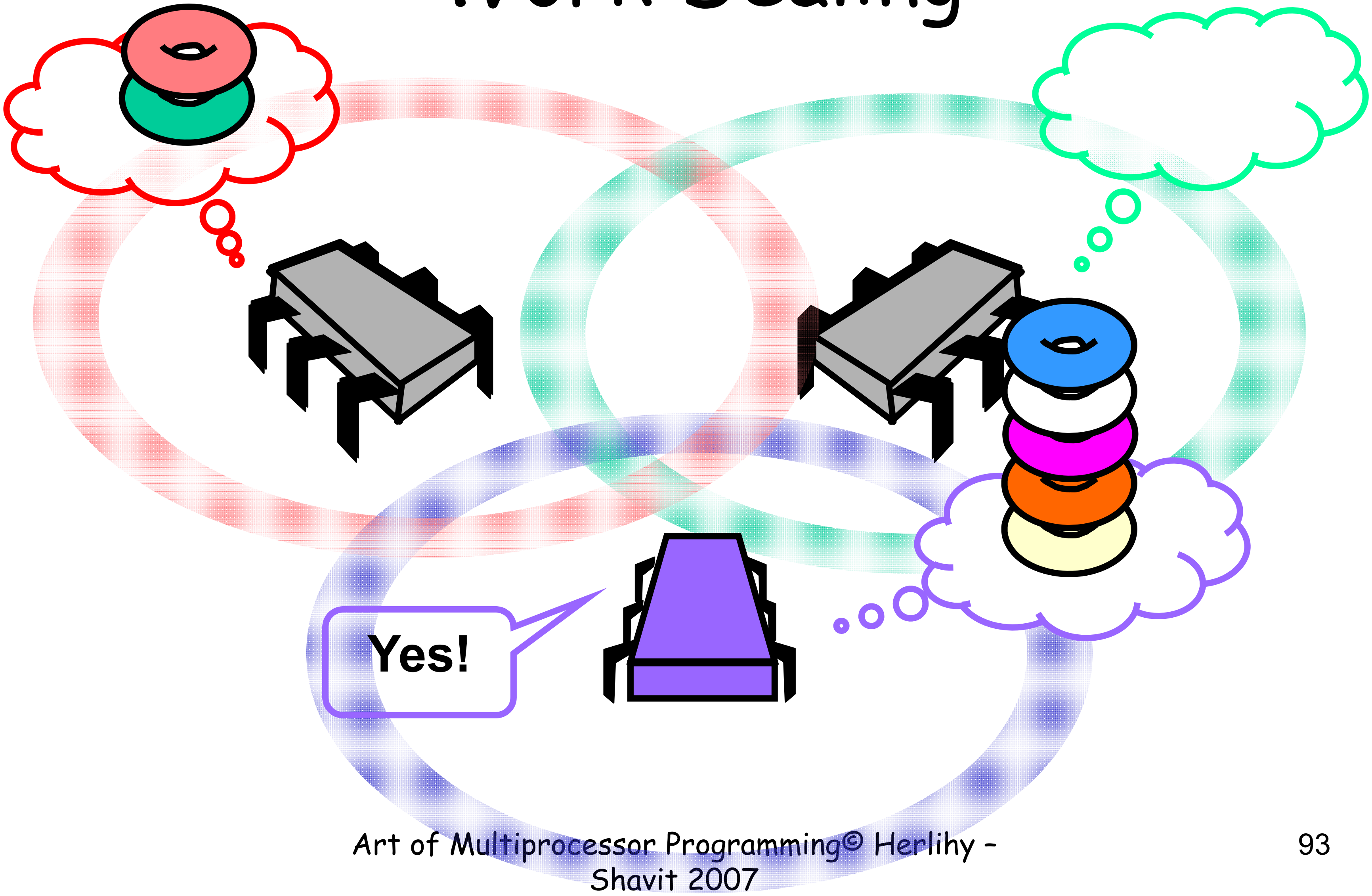
Turns Out

- This bound is within a factor of 2 of optimal
- Actual optimal is NP-complete

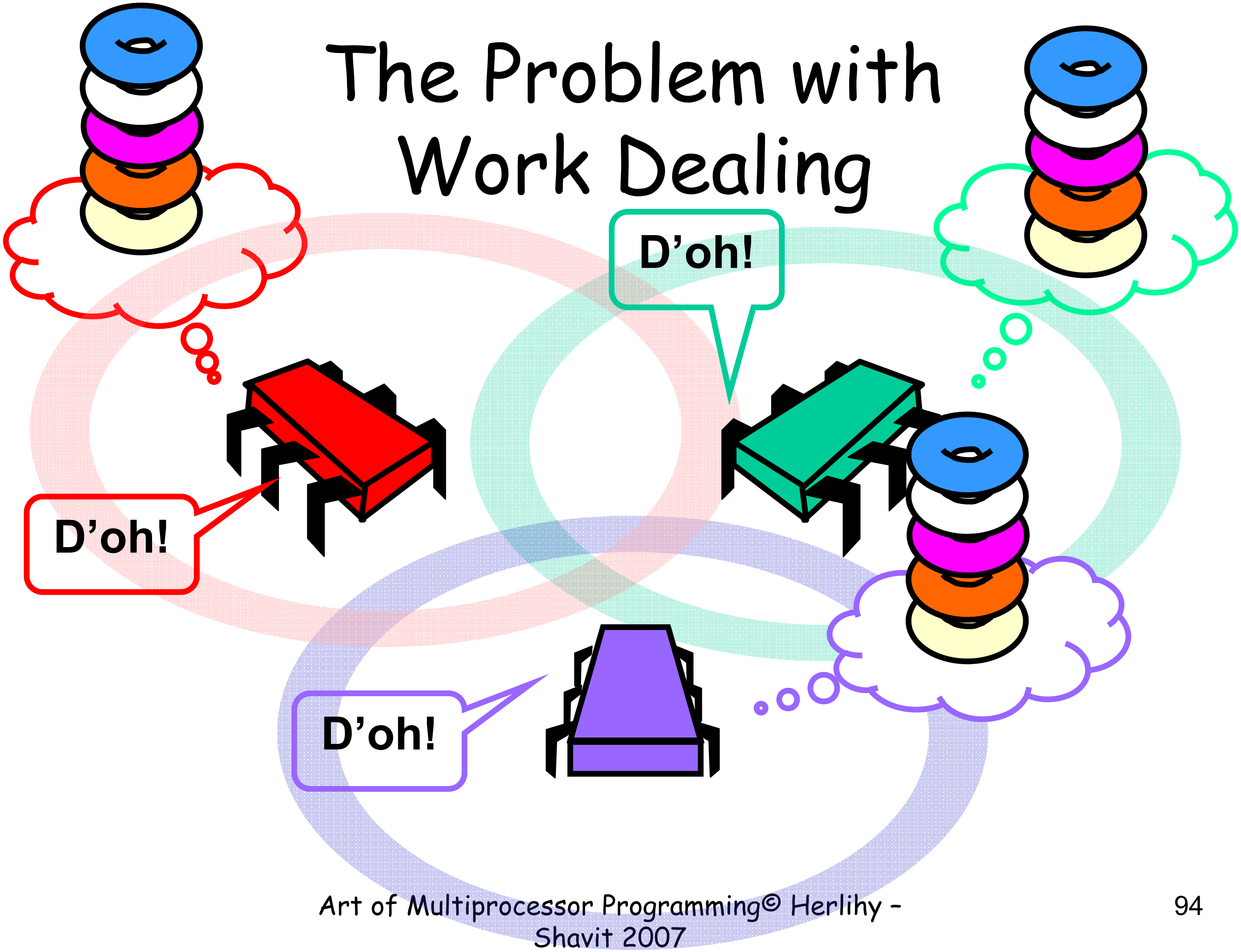
Work Distribution



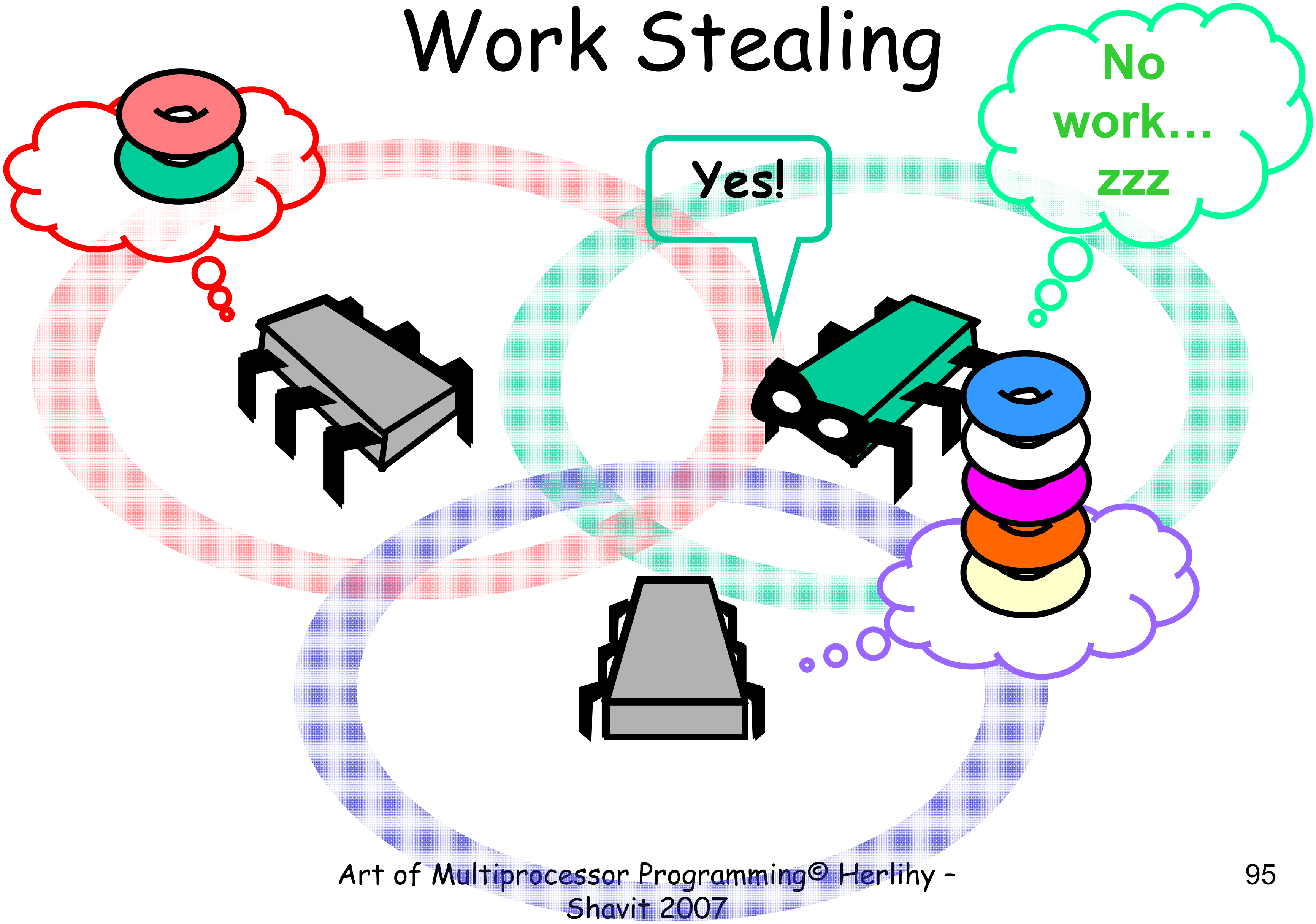
Work Dealing



The Problem with Work Dealing



Work Stealing

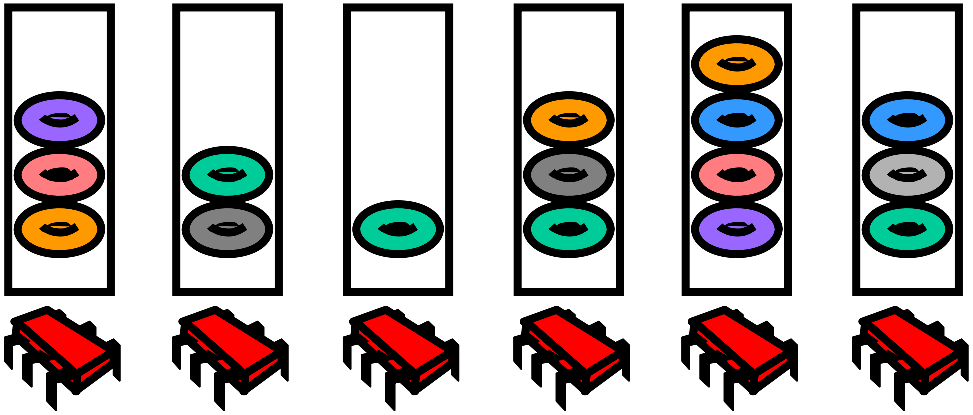


Lock-Free Work Stealing

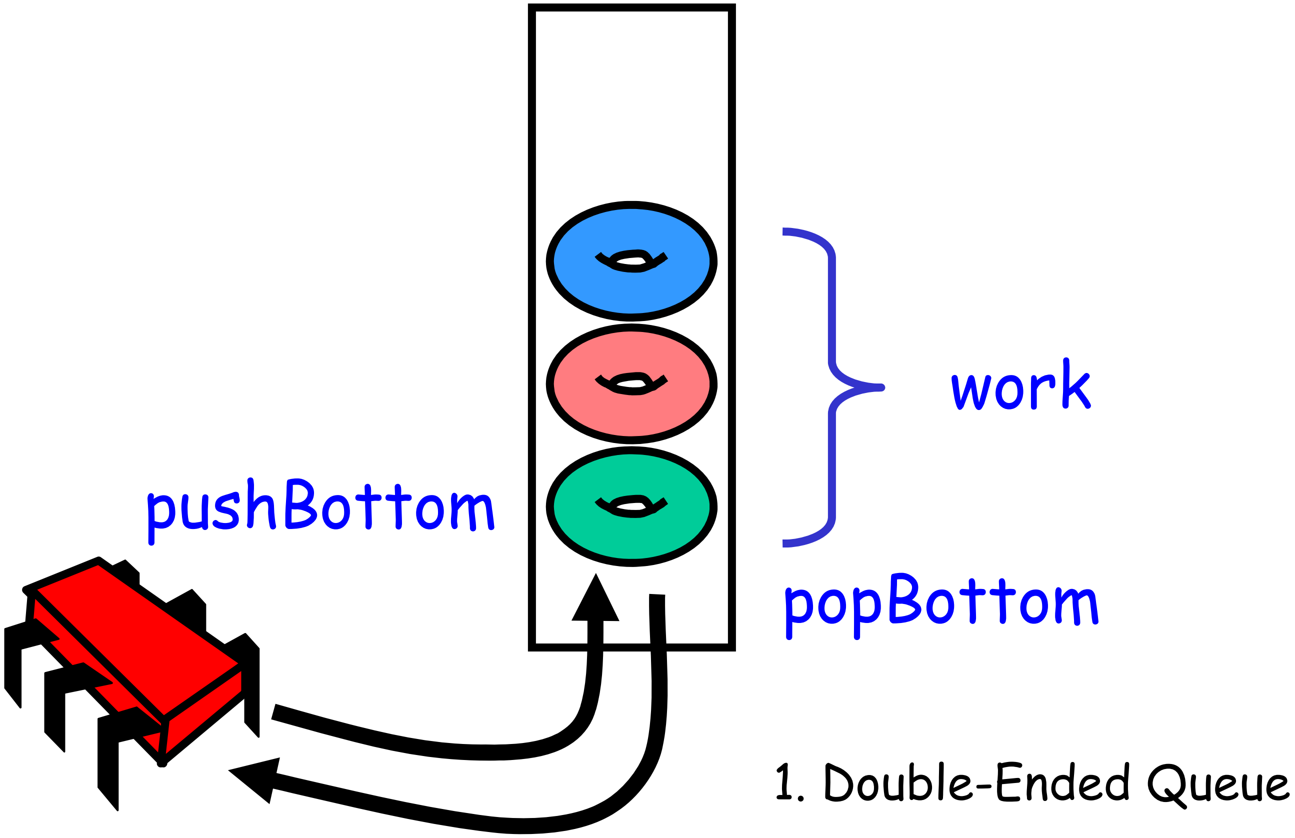
- Each thread has a pool of ready work
- Remove work without synchronizing
- If you run out of work, steal someone else's
- Choose victim at random

Local Work Pools

Each work pool is a Double-Ended Queue



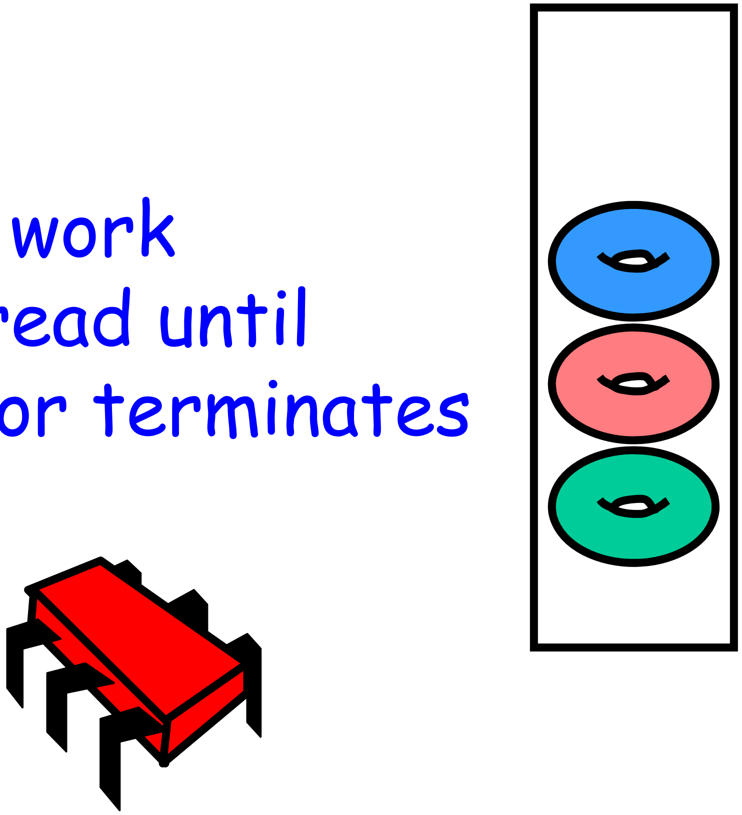
Work DEQueue[1]



[1]. Double-Ended Queue

Obtain Work

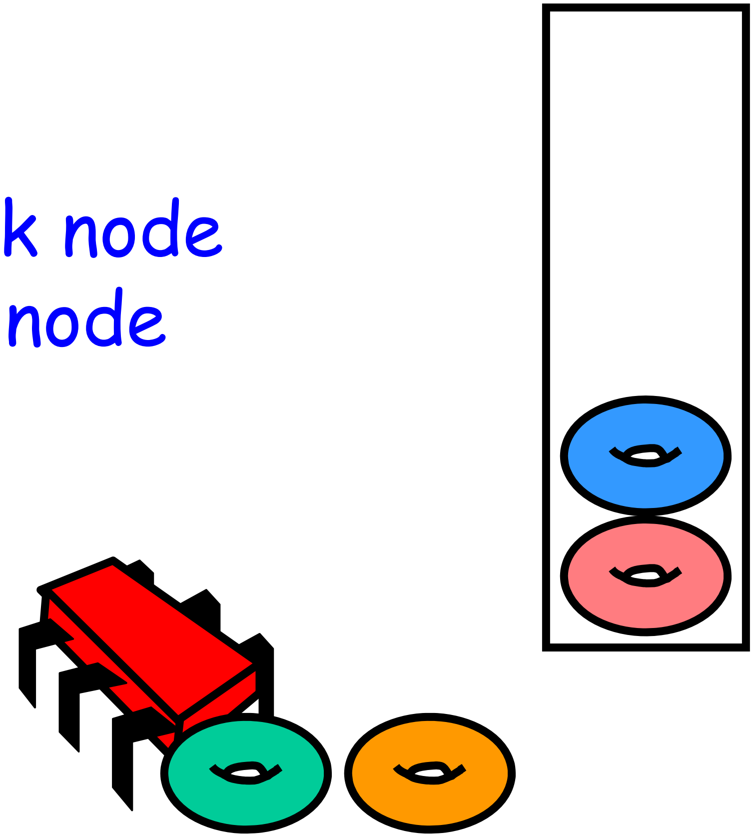
- Obtain work
- Run thread until
- Blocks or terminates



popBottom

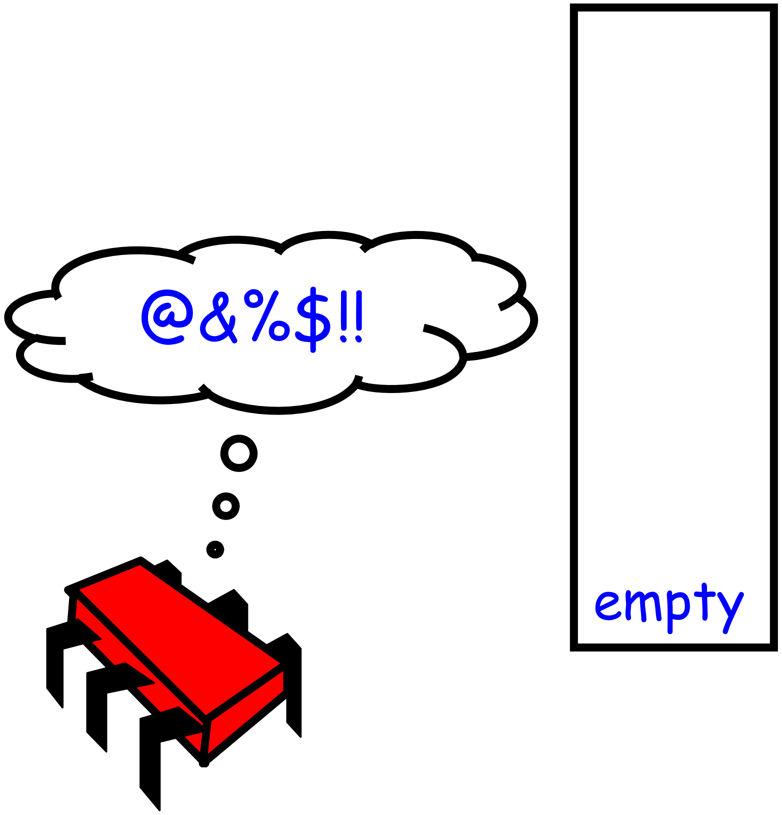
New Work

- Unblock node
- Spawn node

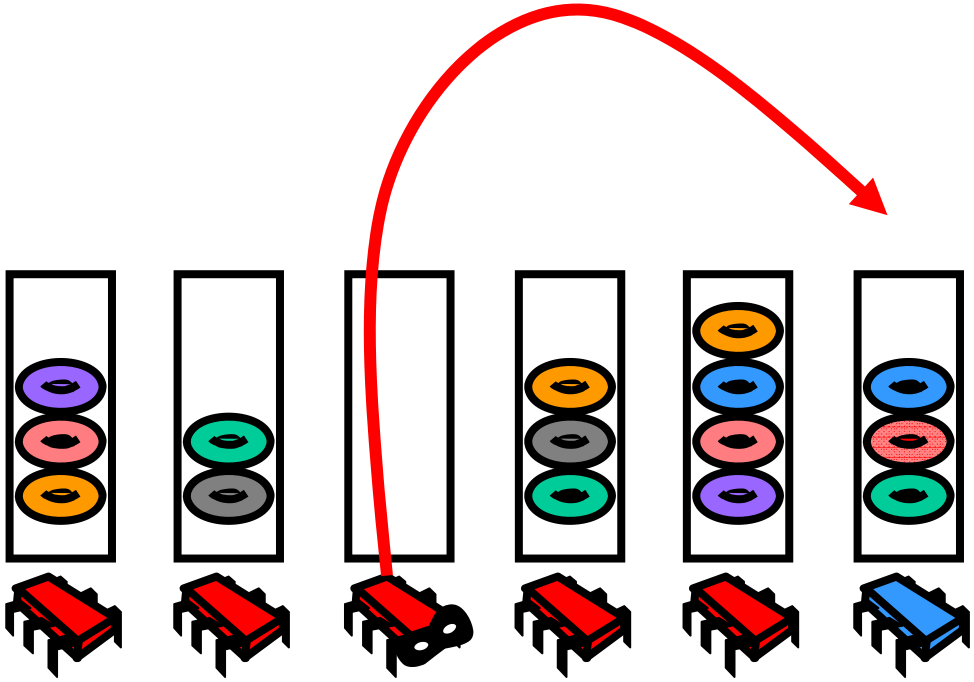


pushBottom

Whatcha Gonna do When the Well Runs Dry?

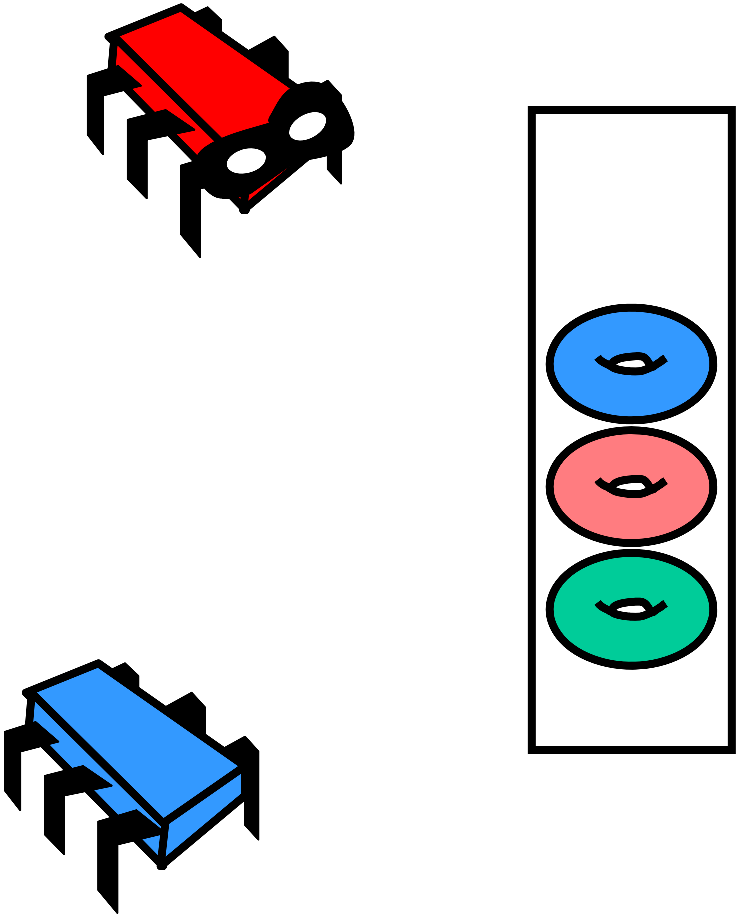


Steal Work from Others



Pick random guy's DEQeueue

Steal this Thread!



popTop

Thread DEQueue

- Methods
  - pushBottom
  - popBottom
  - popTop

Never happen concurrently

Thread DEQueue

- Methods
  - pushBottom
  - popBottom
  - popTop

These most common – make them fast (minimize use of CAS)

Ideal

- Wait-Free
- Linearizable
- Constant time

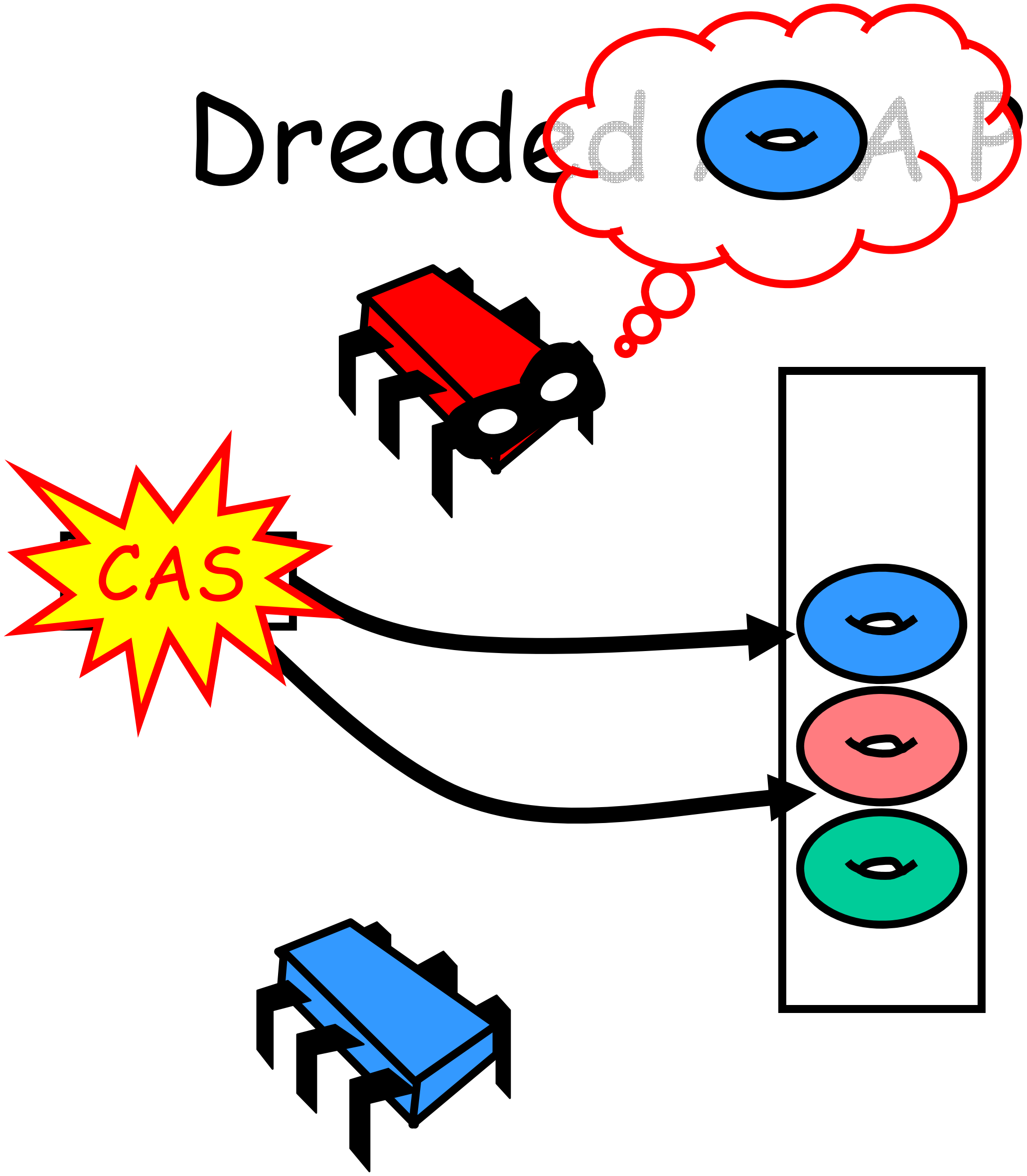
Fortune Cookie: “It is better to be young, rich and beautiful, than old, poor, and ugly”

Compromise

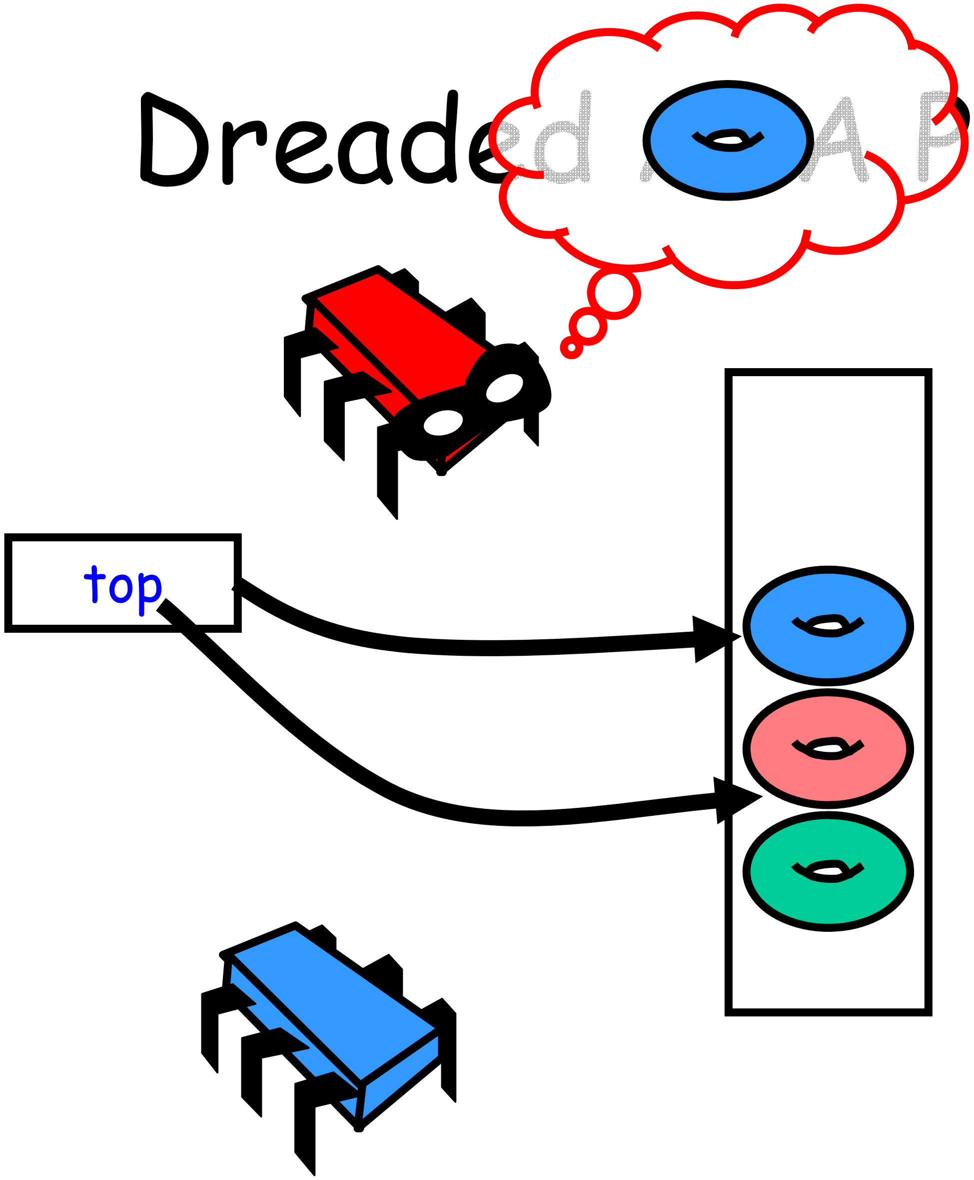
- Method `popTop` may fail if
  - Concurrent `popTop` succeeds, or a
  - Concurrent `popBottom` takes last work

Blame the victim!

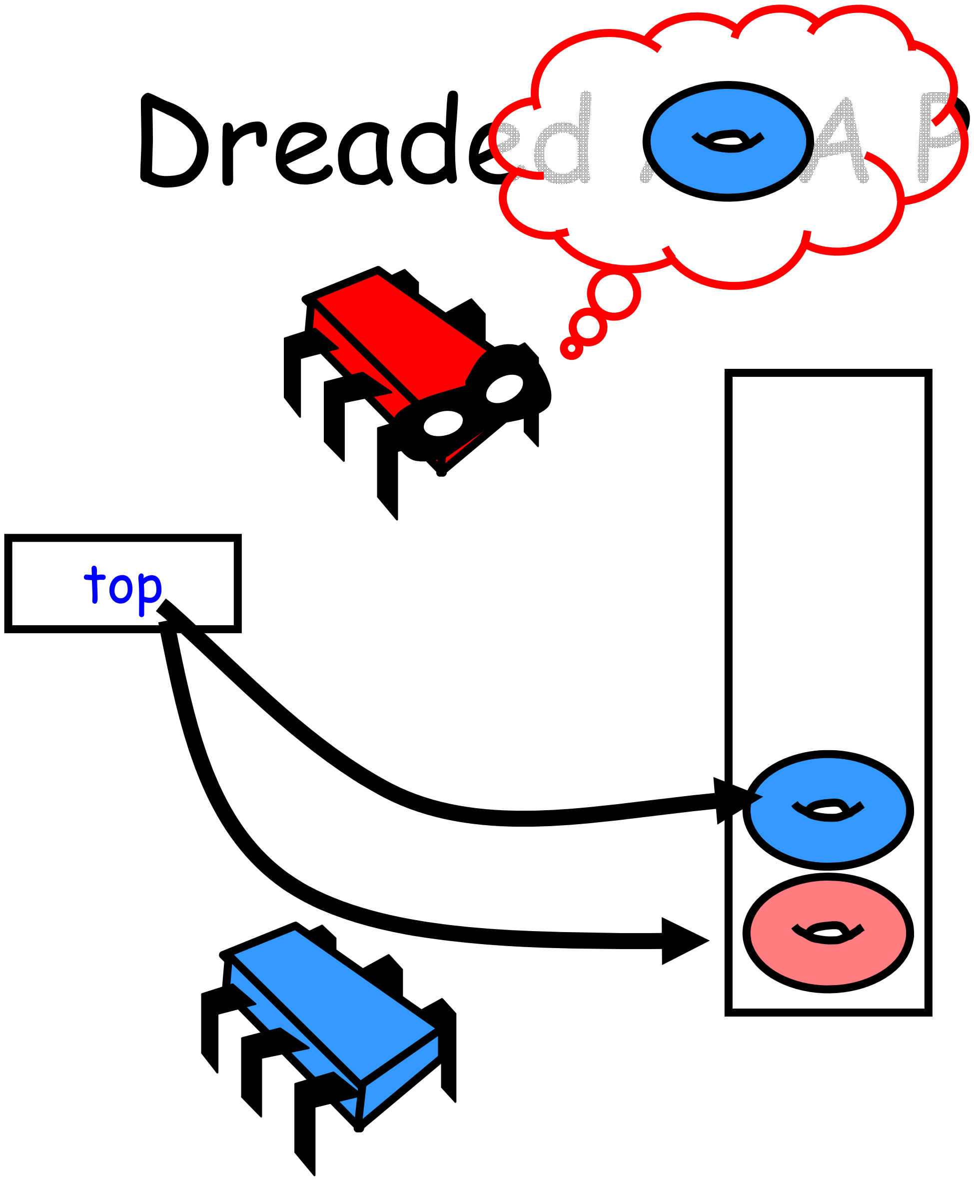
Dreaded A A Problem



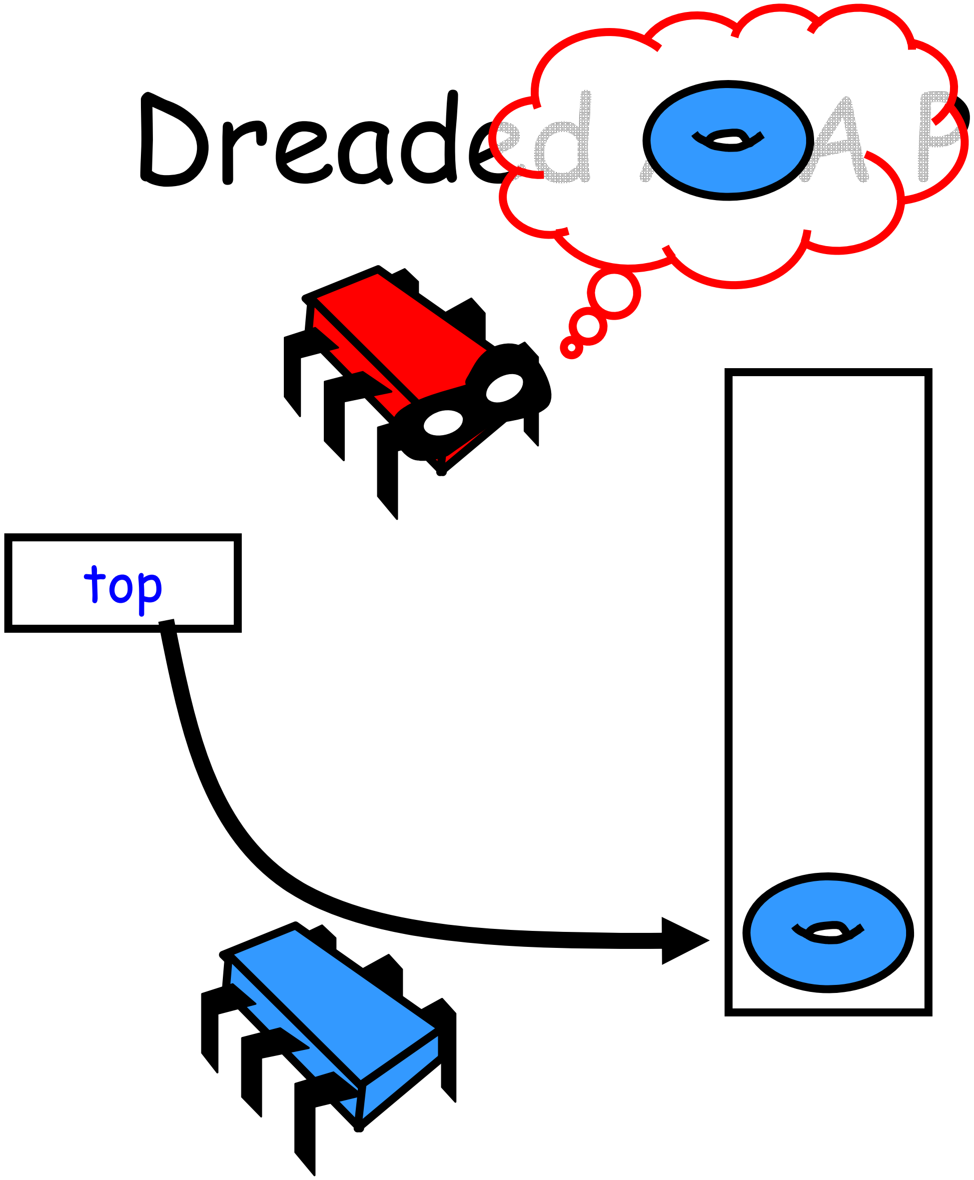
Dreaded ABA Problem



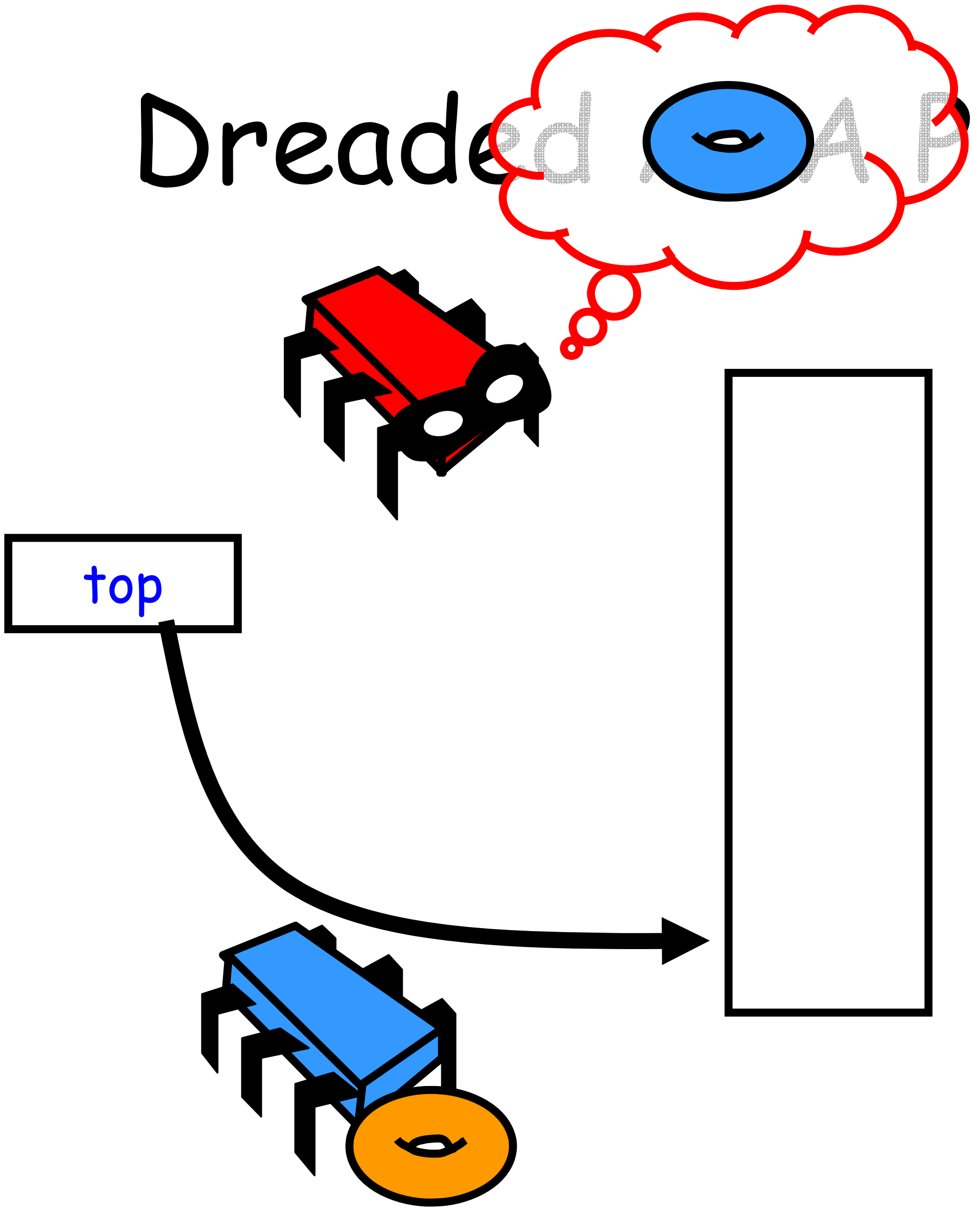
Dreaded ABA Problem



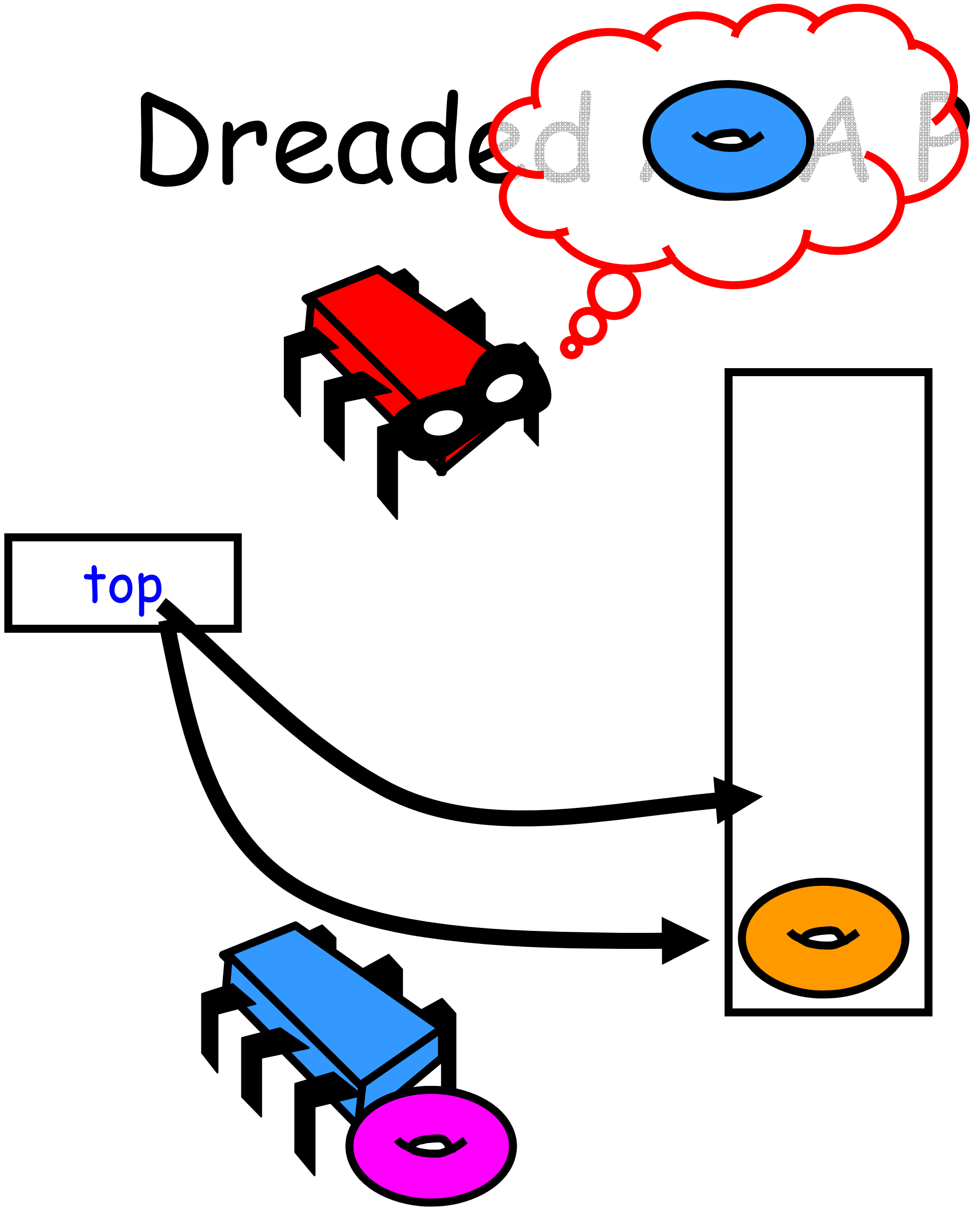
Dreaded ABA Problem



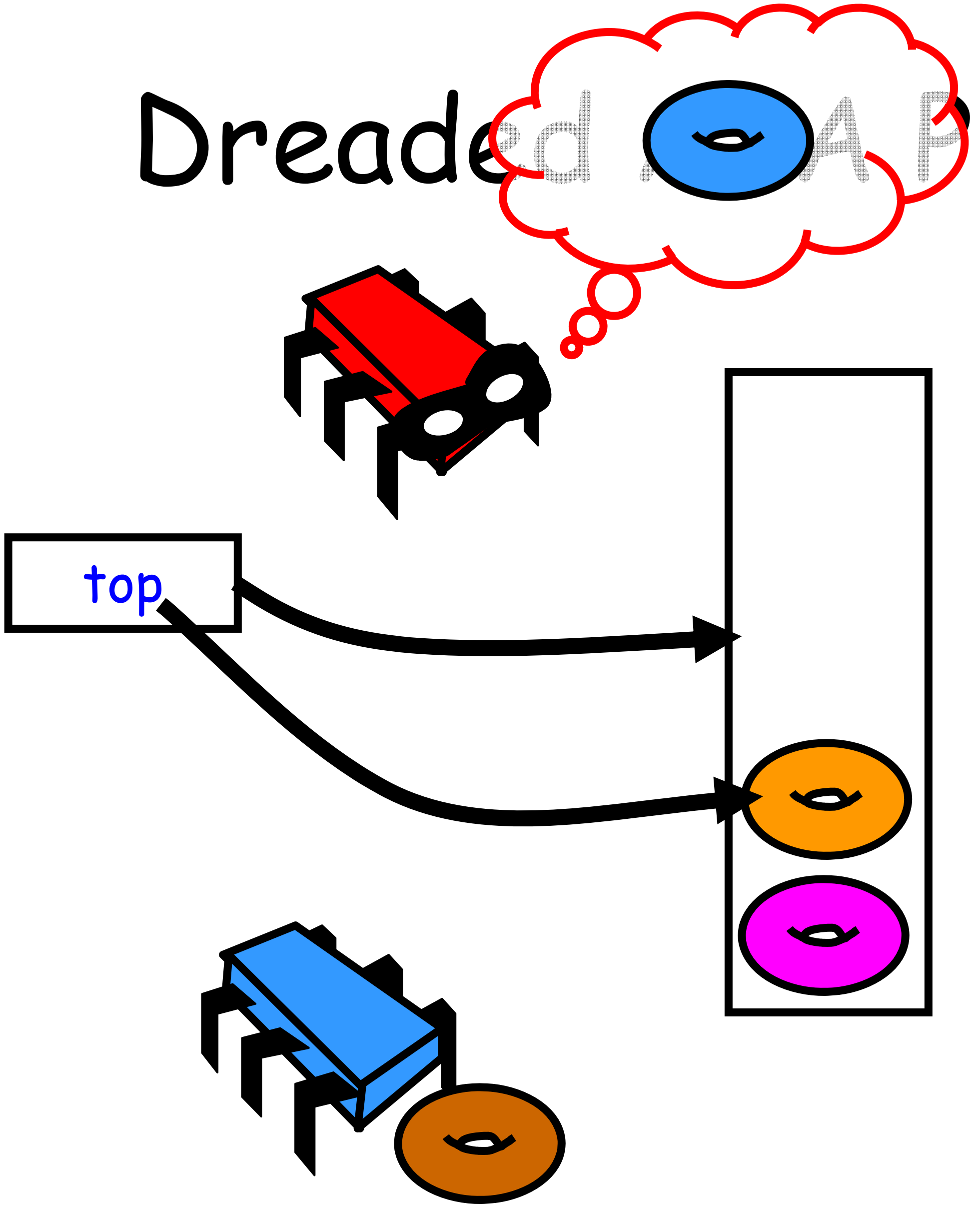
Dreaded ABA Problem



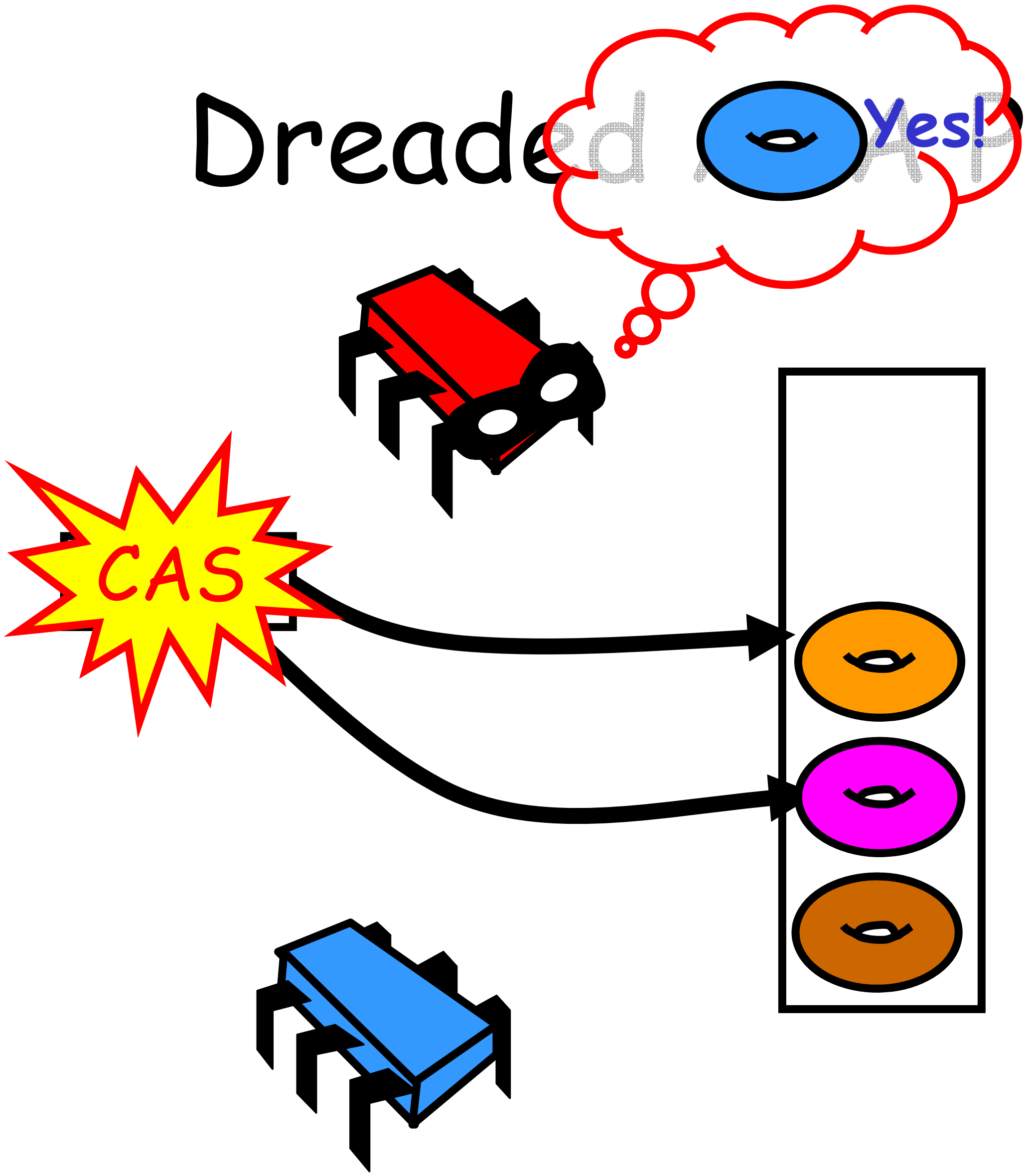
Dreaded ABA Problem



Dreaded A A Problem

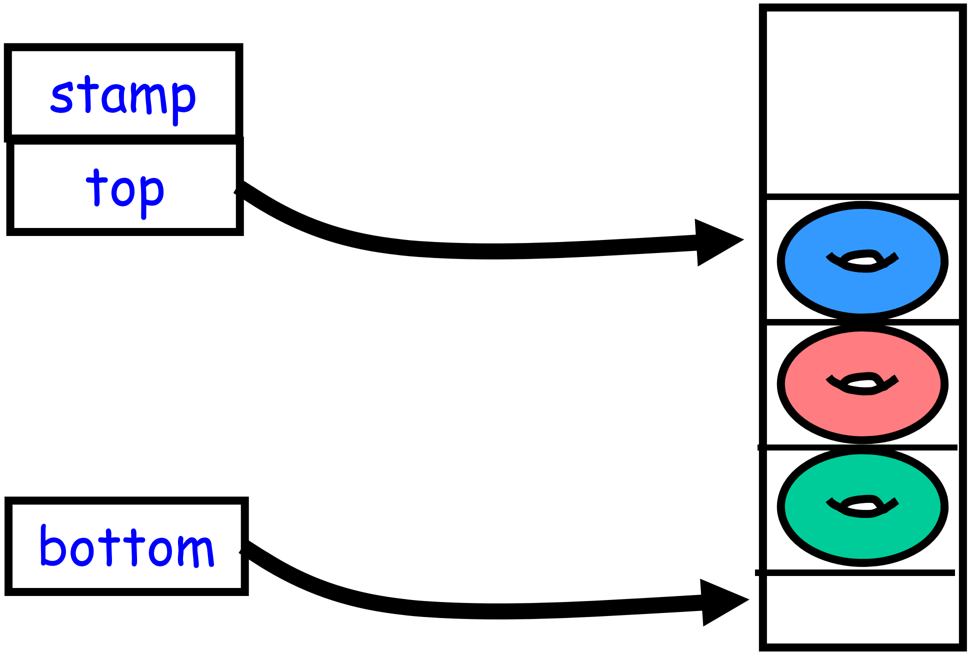


Dreaded ABA Problem



Uh-Oh …

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++;
 }
 …
}
```

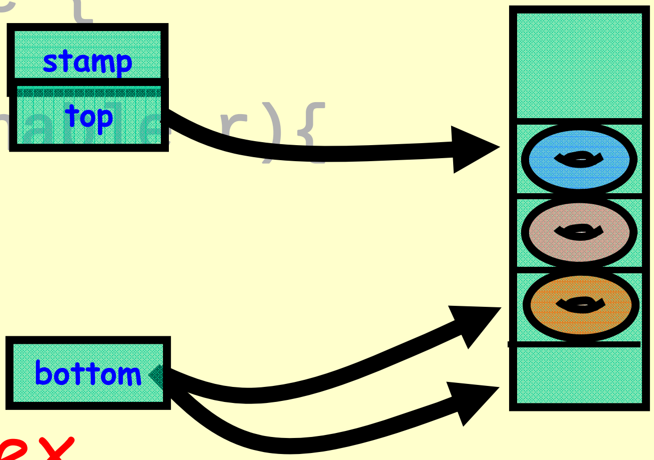
pushBottom()

```
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)

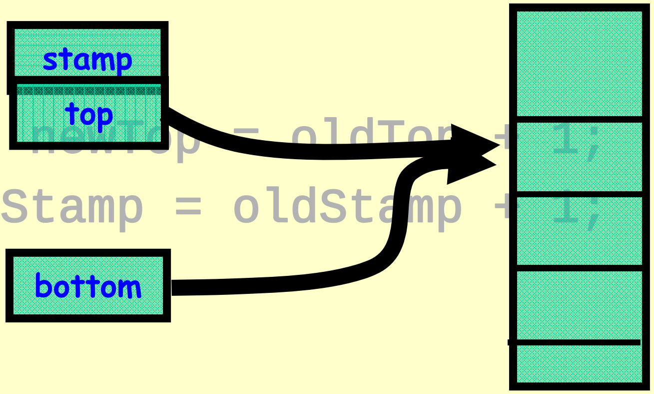
Steal Work

```
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

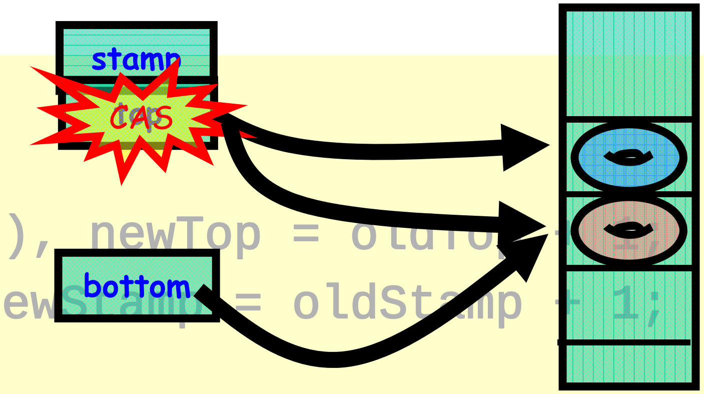
Steal Work

```
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

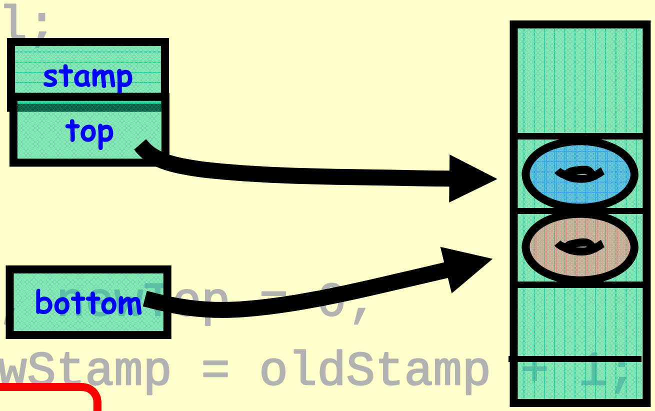
Take Work

```
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

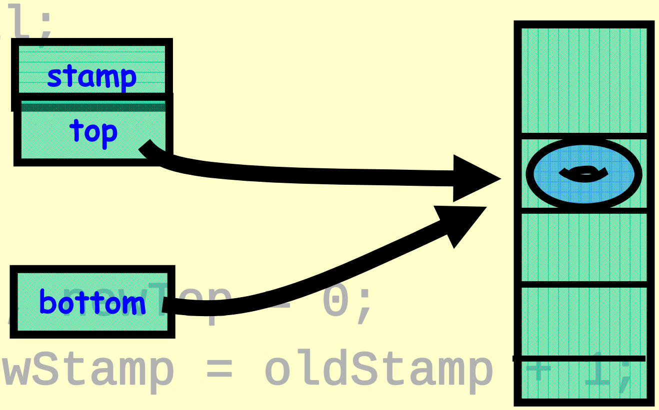
```
Runnable popBottom() {
 if (bottom == 0) return null;
 bottom--;
 Runnable r = tasks[bottom];
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = 0;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom > oldTop) return r;
 if (bottom == oldTop){
   bottom = 0;
   if (top.CAS(oldTop,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

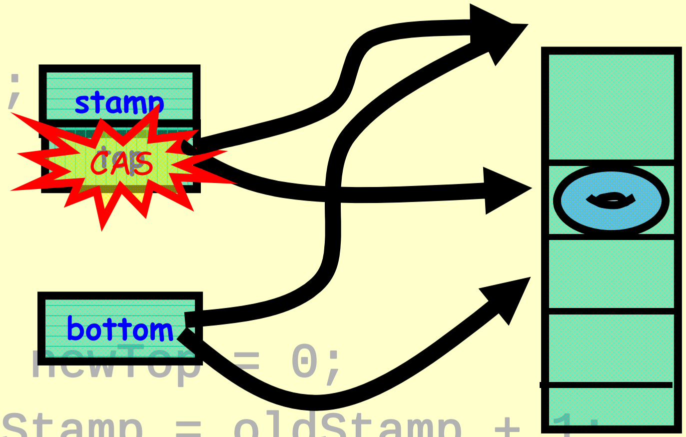
Try to steal last item.
In any case reset Bottom
because the DEQueue will be empty
even if unsucessful (why?)

```
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

I win CAS

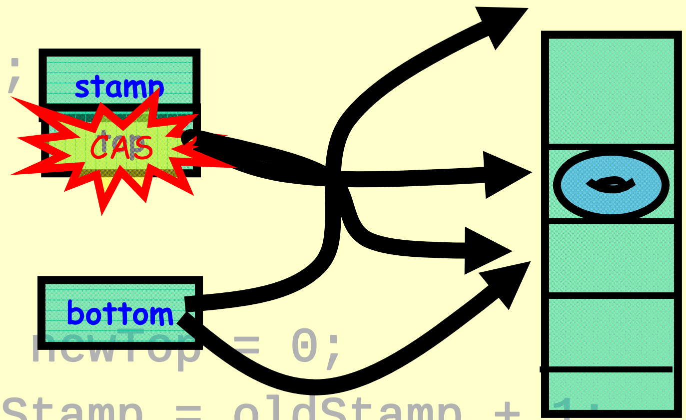
```
Runnable popBottom() {
 if (bottom == 0) return null;
 bottom--;
 Runnable r = tasks[bottom];
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = 0;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom > oldTop) return r;
 if (bottom == oldTop){
   bottom = 0;
   if (top.CAS(oldTop,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

I lose CAS
Thief must
have won…



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;
}
```

failed to get last item

Must still reset top

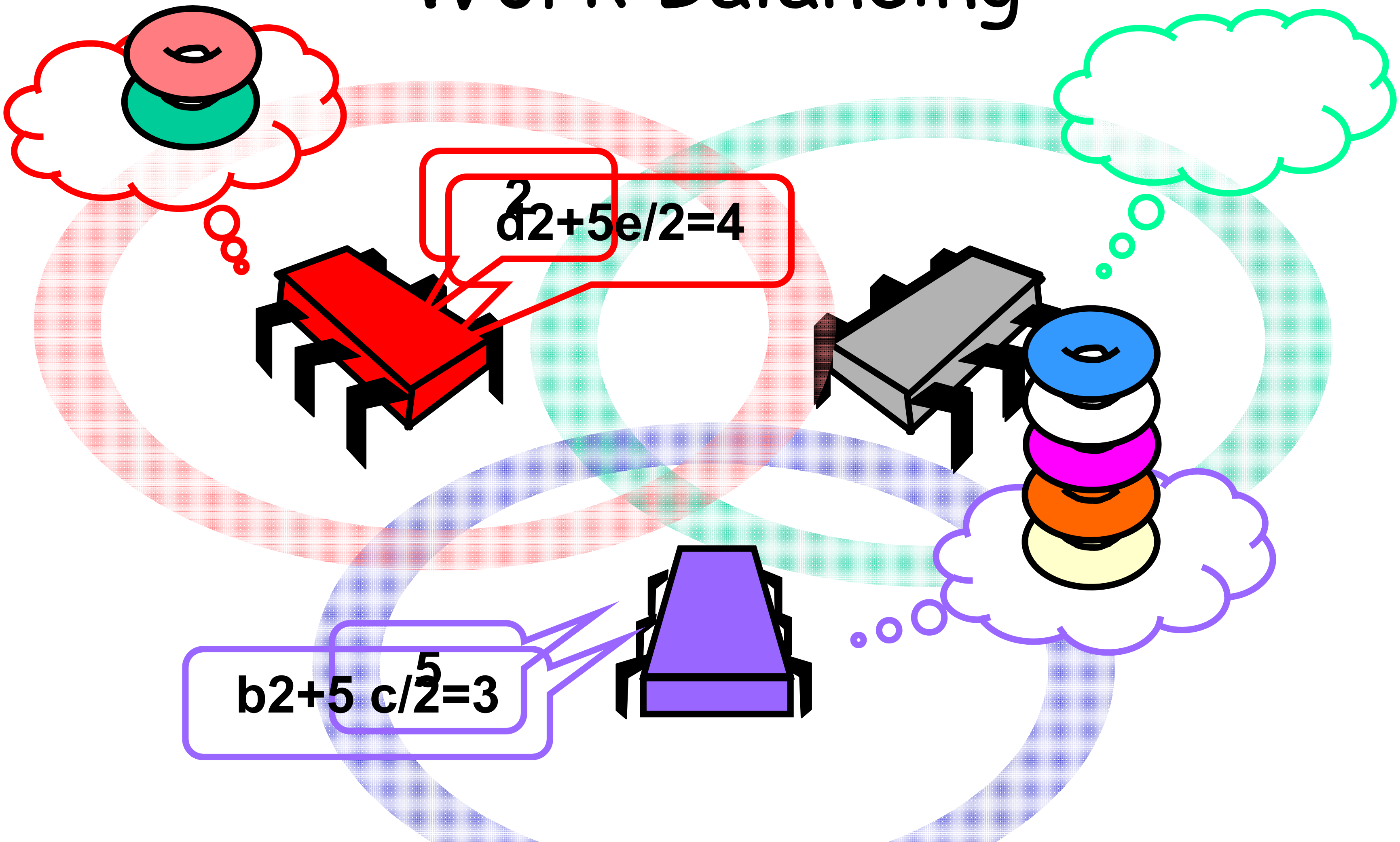
Old English Proverb

- "May as well be hanged for stealing a sheep as a goat"
- From which we conclude
  - Stealing was punished severely
  - Sheep were worth more than goats

Variations

- Stealing is expensive
  - Pay CAS
  - Only one thread taken
- What if
  - Randomly balance loads?

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

Work Stealing & Balancing

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

TOM

MARVOLO

RIDDLE

This work is licensed under a Creative Commons Attribution-ShareAlike 2.5 License.

- **You are free**:
  - **to Share** — to copy, distribute and transmit the work
  - **to Remix** — to adapt the work
- **Under the following conditions**:
  - **Attribution.** You must attribute the work to "The Art of Multiprocessor Programming" (but not in any way that suggests that the authors endorse you or your use of the work).
  - **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
  - http://creativecommons.org/licenses/by-sa/3.0/.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.