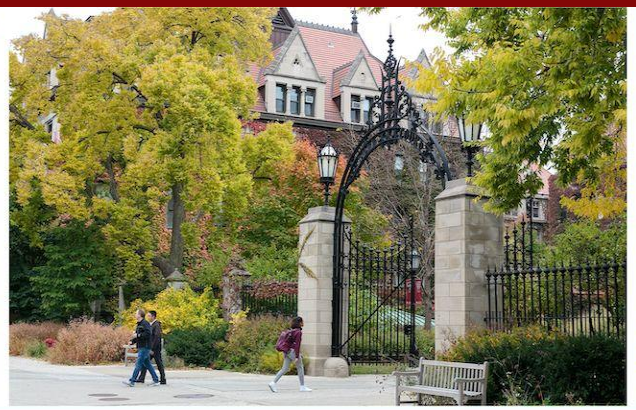
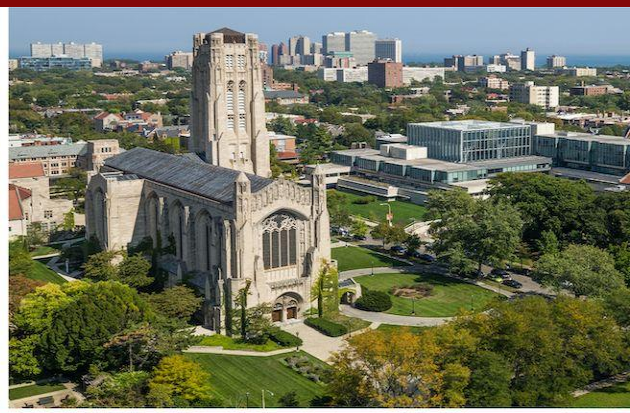
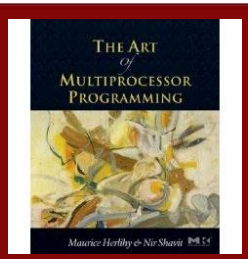
MPCS 52060 - Parallel Programming
M6: Advanced Parallel Patterns and Techniques (Part 1)

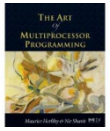






Original slides from "The Art of Multiprocessor Programming" by Maurice Herlihy & Nir Shavit with modifications by Lamont Samuels

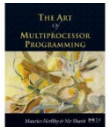
How to write Parallel Apps?

- Split a program into parallel parts
- In an effective way
- Thread management
- We've seen a few ways so far to do this
  - Fork-Join (Static, Map)
  - Producer/Consumer model with global queue
  - Pipeline (M6)



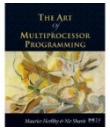
Matrix Multiplication

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



Matrix Multiplication

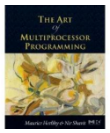
```
class Worker extends Thread {
  int row, col;
  Worker(int row, int col) {
    this.row = row; this.col = col;
  }
  public void run() {
    double dotProduct = 0.0;
    for (int i = 0; i < n; i++)
      dotProduct += a[row][i] * b[i][col];
    c[row][col] = dotProduct;
  }}}
```



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;
  }}}
```

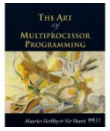
a thread



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;
  }}}
```

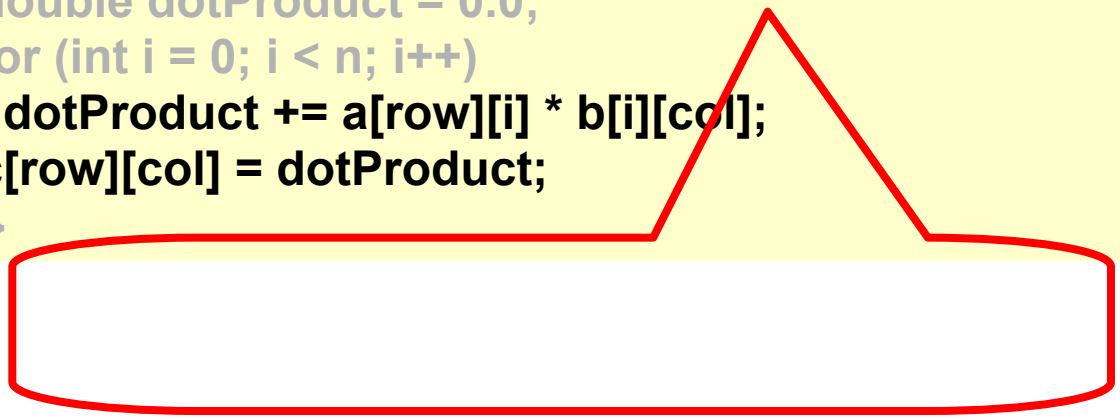
Which matrix entry to compute

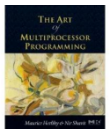


Matrix Multiplication

```
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;
  }}}
```

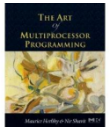
Actual computation





Matrix Multiplication

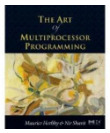
```
void multiply() {
  Worker[][] worker = new Worker[n][n];
  for (int row …)
    for (int col …)
      worker[row][col] = new Worker(row,col);
  for (int row …)
    for (int col …)
      worker[row][col].start();
  for (int row …)
    for (int col …)
      worker[row][col].join();
}
```



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();
}
```

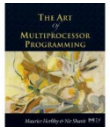
Create n x n threads



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

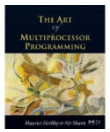


Matrix Multiplication

```
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



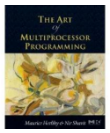
Matrix Multiplication

```
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

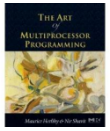
What's wrong with this picture?

Wait for them to finish



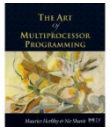
Thread Overhead

- Threads Require resources
  - Memory for stacks
  - Setup, teardown
  - Scheduler overhead
- Short-lived threads
  - Ratio of work versus overhead bad



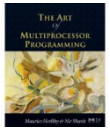
Thread Pools

- More sensible to keep a pool of
  - long-lived threads
- Threads assigned short-lived tasks
  - Run the task
  - Rejoin pool
  - Wait for next assignment



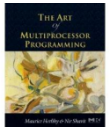
Thread Pool = Abstraction

- Insulate programmer from platform
  - Big machine, big pool
  - Small machine, small pool
- Portable code
  - Works across platforms
  - Worry about algorithm, not platform



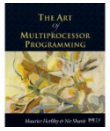
ExecutorService Interface

- 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>

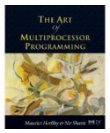
```
Callable<T> task = …;
…
Future<T> future = executor.submit(task);
…
T value = future.get();
```



Future<T>

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

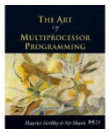
Submitting a **Callable<T>** task returns a **Future<T>** object



Future<T>

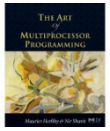
```
Callable<T> task = …;
…
Future<T> future = executor.submit(task);
…
T value = future.get();
```

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



Future<?>

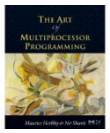
```
Runnable task = …;
…
Future<?> future = executor.submit(task);
…
future.get();
```



Future<?>

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

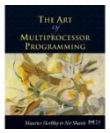
Submitting a **Runnable** task returns a **Future<?>** object



Future<?>

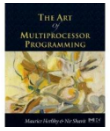
```
Runnable task = …;
…
Future<?> future = executor.submit(task);
…
future.get();
```

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



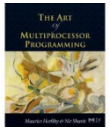
Note

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



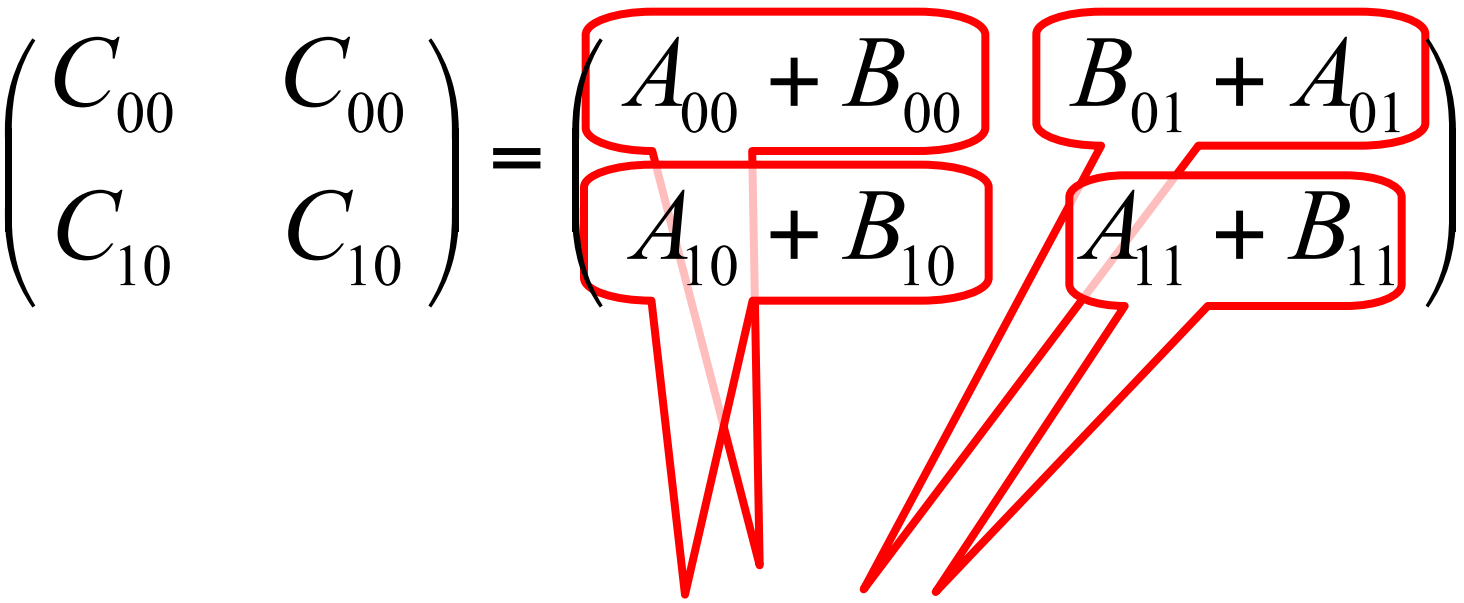
Matrix Addition

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

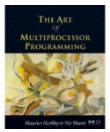


Matrix Addition

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

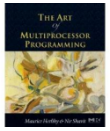


4 parallel additions



Matrix Addition Task

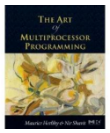
```
class AddTask implements Runnable {
  Matrix a, b; // add this!
  public void run() {
    if (a.dim == 1) {
     c[0][0] = a[0][0] + b[0][0]; // base case
    } else {
     (partition a, b into half-size matrices a_ij and b_ij)
     Future<?> f_00 = exec.submit(add(a_00,b_00));
     …
     Future<?> f_11 = exec.submit(add(a_11,b_11));
     f_00.get(); …; f_11.get();
     …
    }}
```



Matrix Addition Task

```
class AddTask implements Runnable {
  Matrix a, b; // add 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();
     …
    }}
```

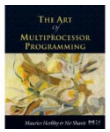
Constant-time operation



Matrix Addition Task

```
class AddTask implements Runnable {
  Matrix a, b; // add 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();
     …
    }}
```

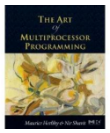
Submit 4 tasks



Matrix Addition Task

```
class AddTask implements Runnable {
  Matrix a, b; // add 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();
    …
   }}
```

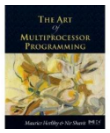
Base case: add directly



Matrix Addition Task

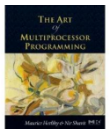
```
class AddTask implements Runnable {
  Matrix a, b; // multiply this!
  public void run() {
    if (a.dim == 1) {
     c[0][0] = a[0][0] + b[0][0]; // base case
    } else {
     (partition a, b into half-size matrices a_ij and b_ij)
     Future<?> f_00 = exec.submit(add(a_00,b_00));
     …
     Future<?> f_11 = exec.submit(add(a_11,b_11));
     f_00.get(); …; f_11.get();
     …
    }}
```

Let them finish

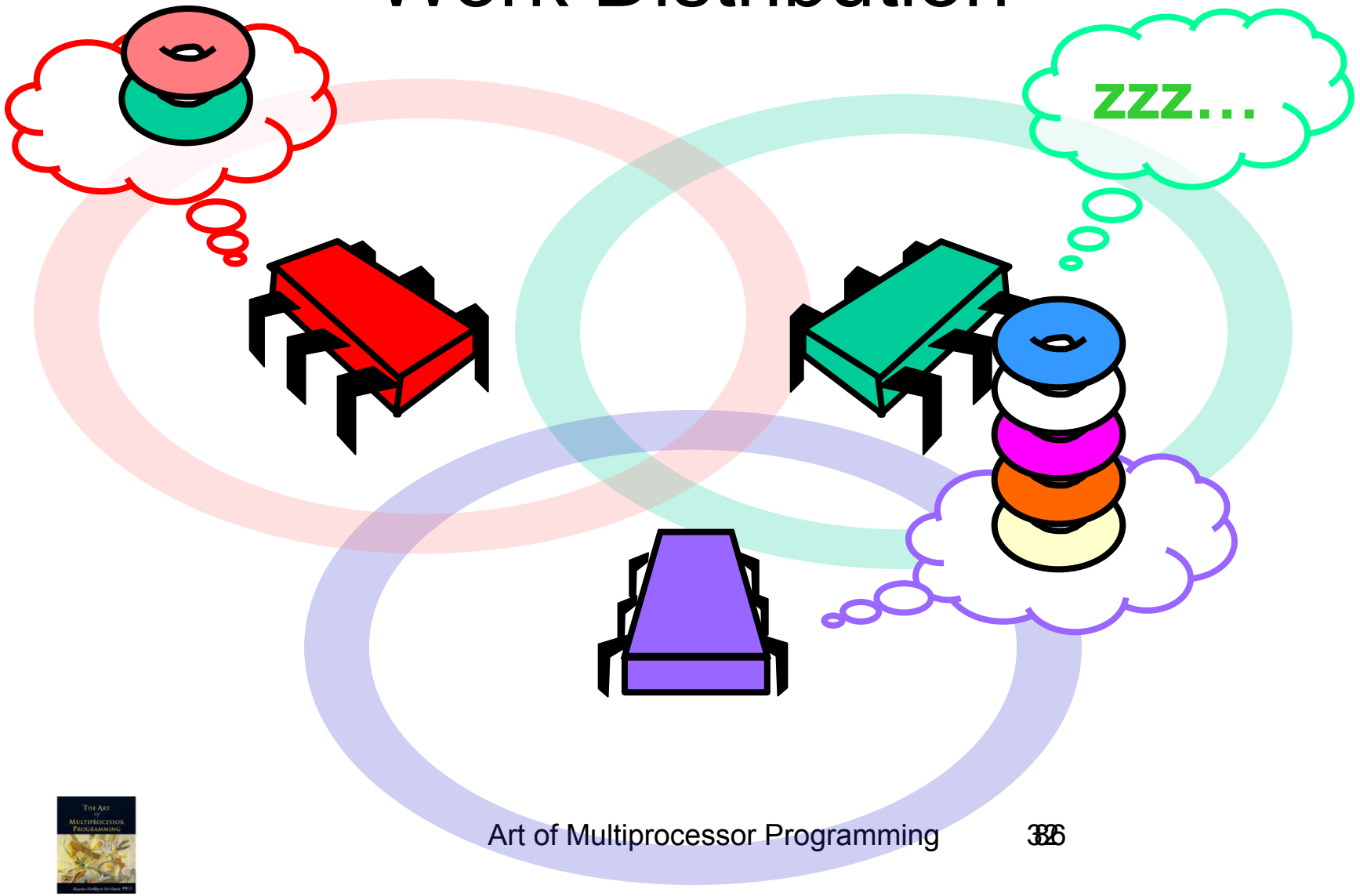


Dependencies

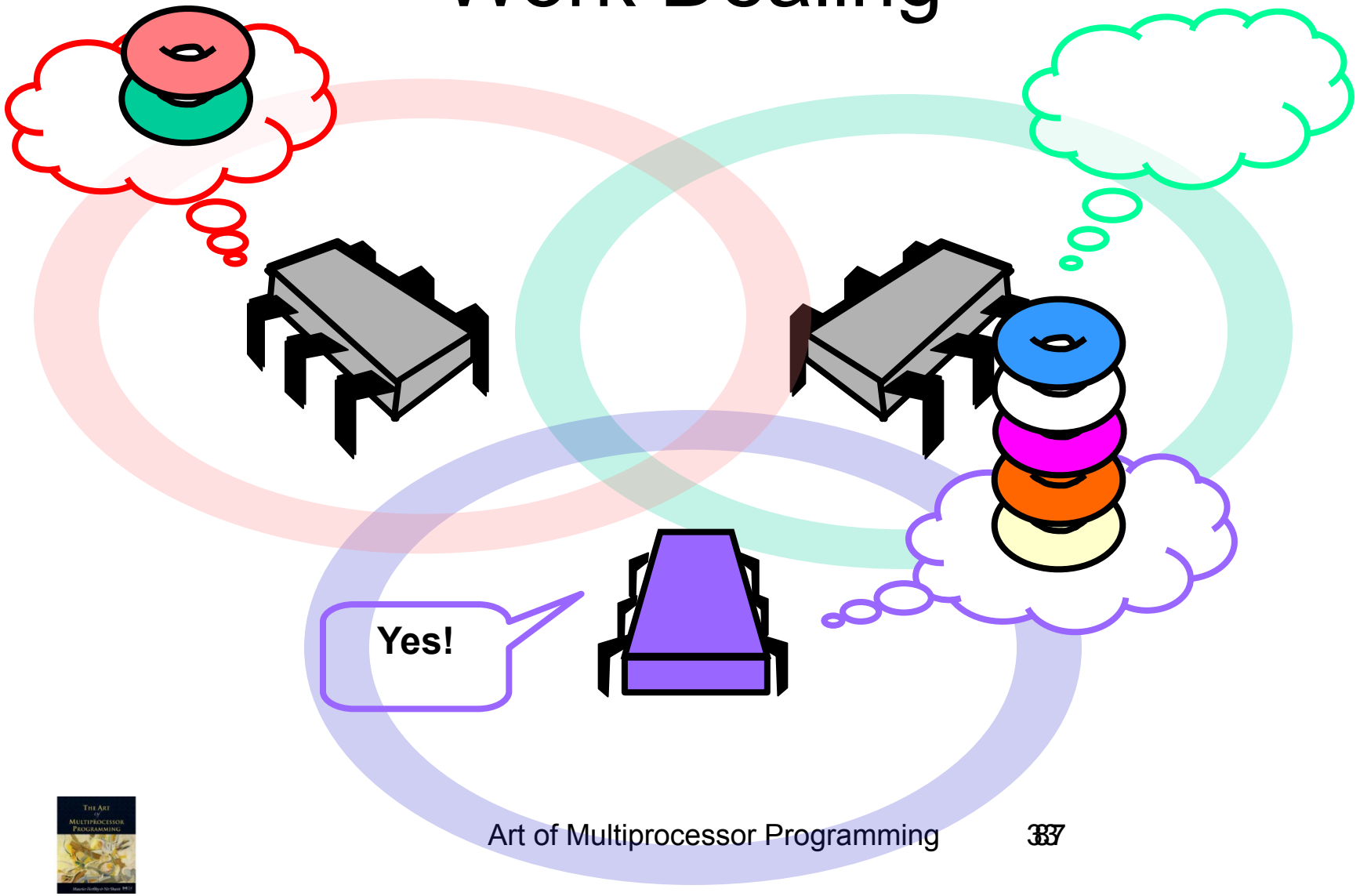
- Matrix example is not typical
- Tasks are independent
  - Don't need results of one task …
  - To complete another
- Often tasks are not independent
  - We can use other ways to handle these types of tasks
  - For example - pipelining, barriers, etc.



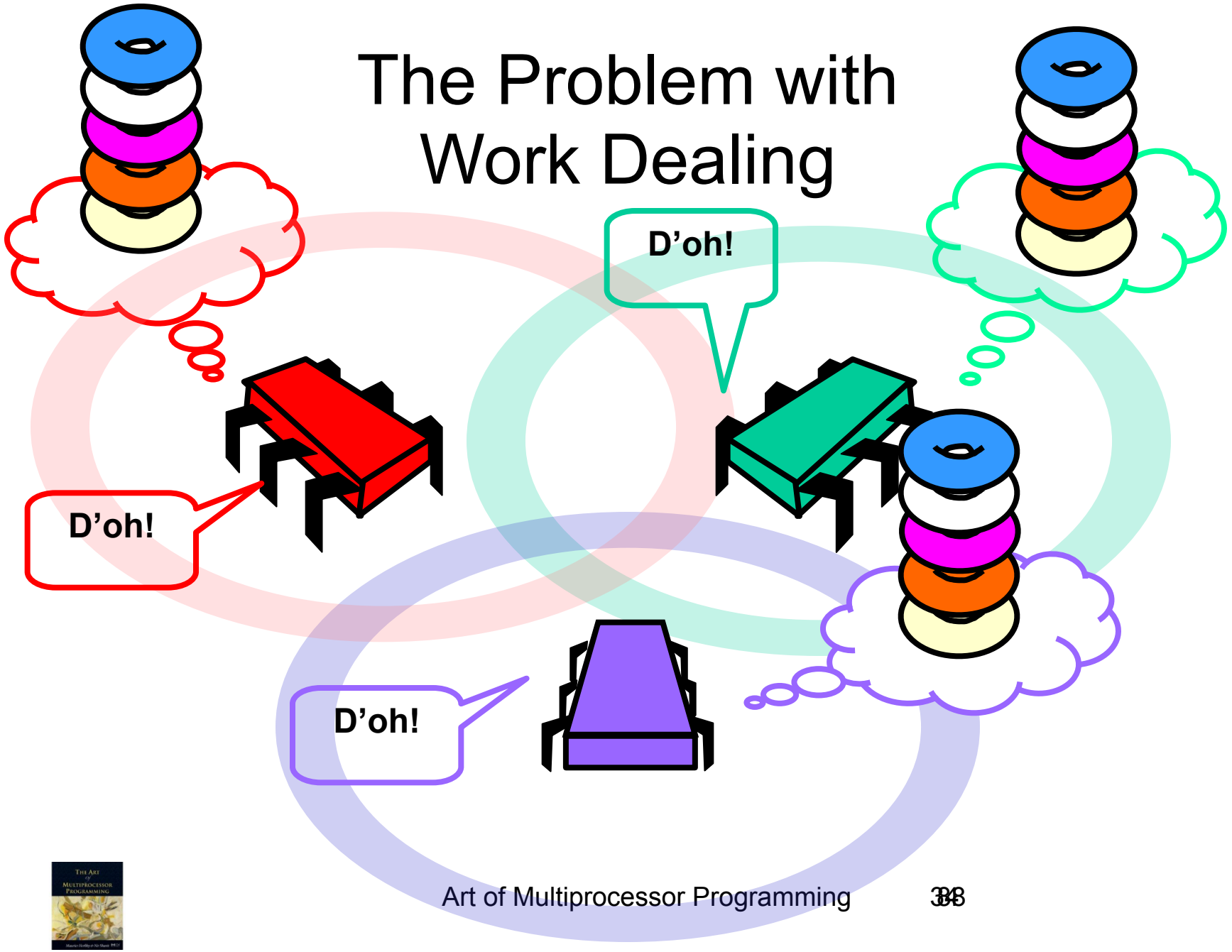
Work Distribution



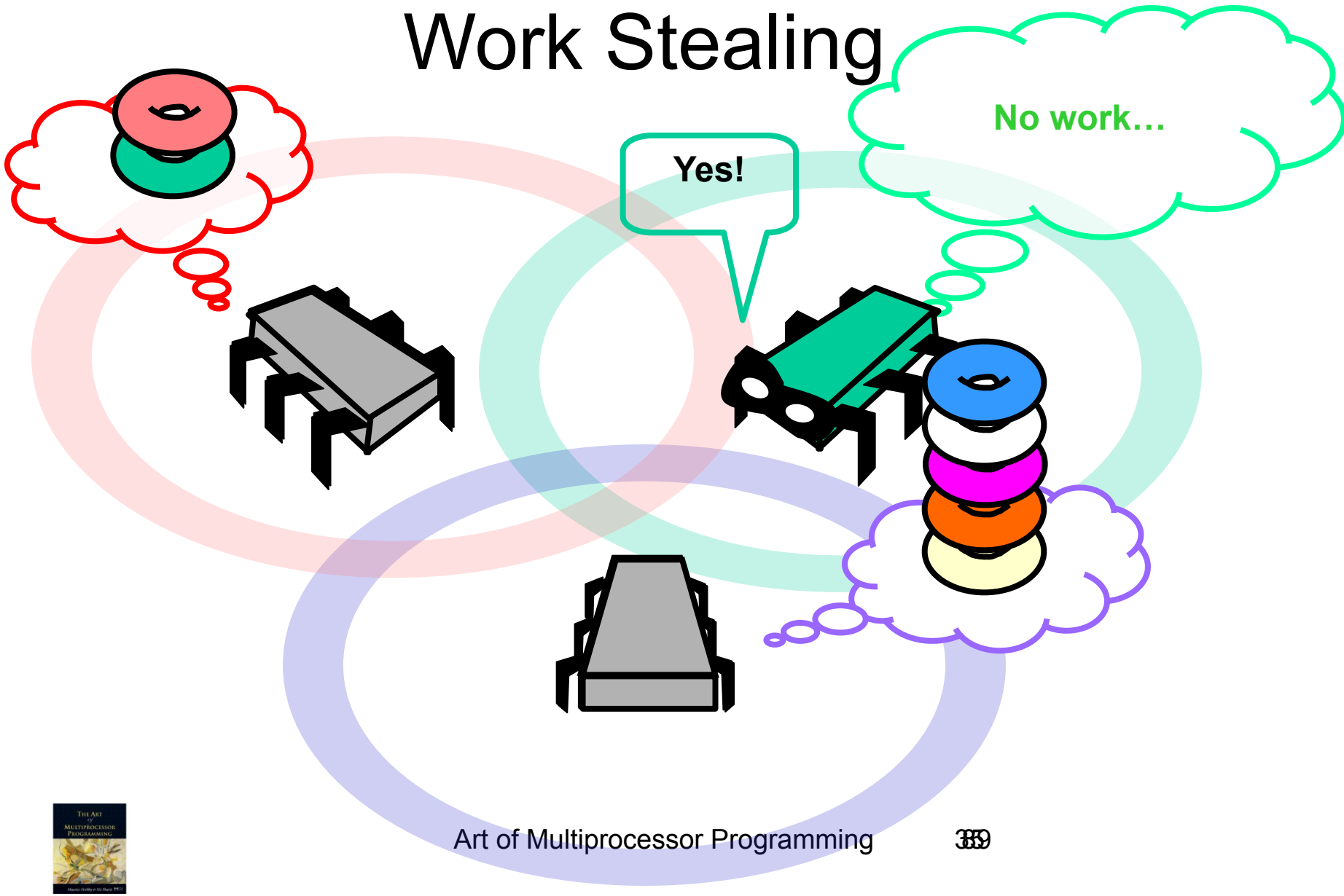
Work Dealing



The Problem with Work Dealing

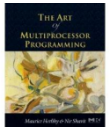


Work Stealing



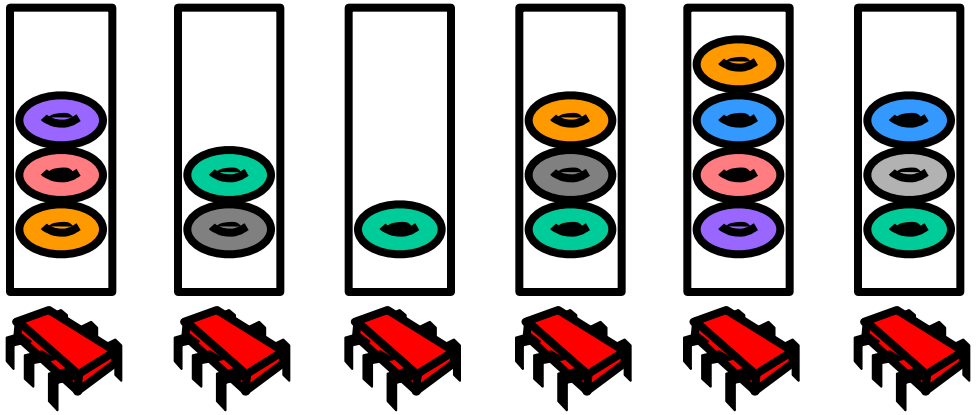
Lock-Free Work Stealing

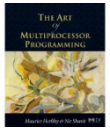
- Each thread has a pool of ready work
- Remove work without synchronizing
- If you run out of work, steal someone else's
- Choose victim at random



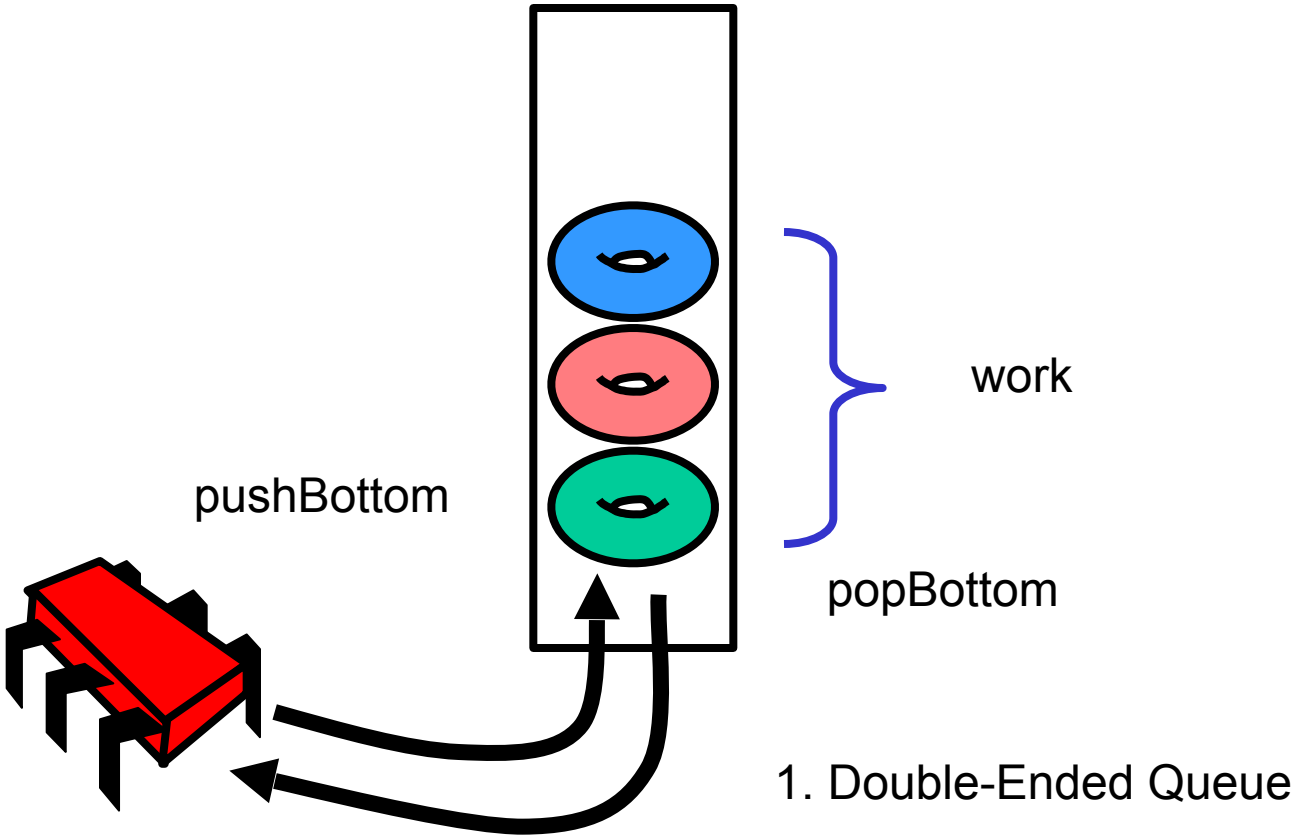
Local Work Pools

Each work pool is a Double-Ended Queue

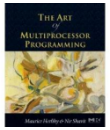




Work DEQueue[1]

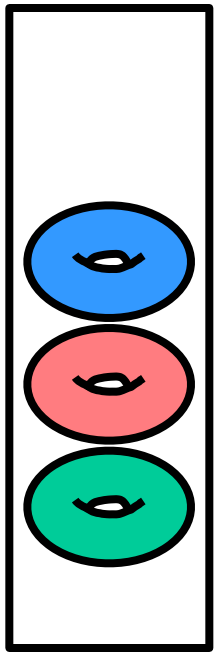


1. Double-Ended Queue

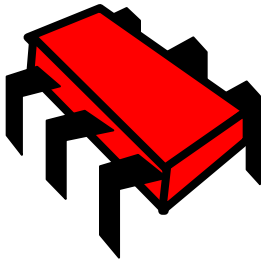


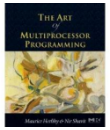
Obtain Work

- Obtain work
- Run task until
- Blocks or terminates



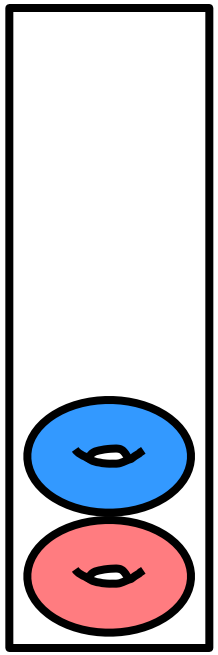
popBottom



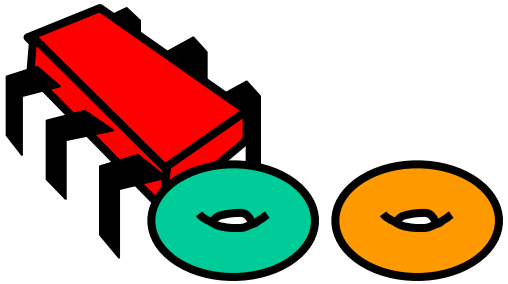


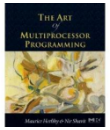
New Work

- Unblock node
- Spawn node

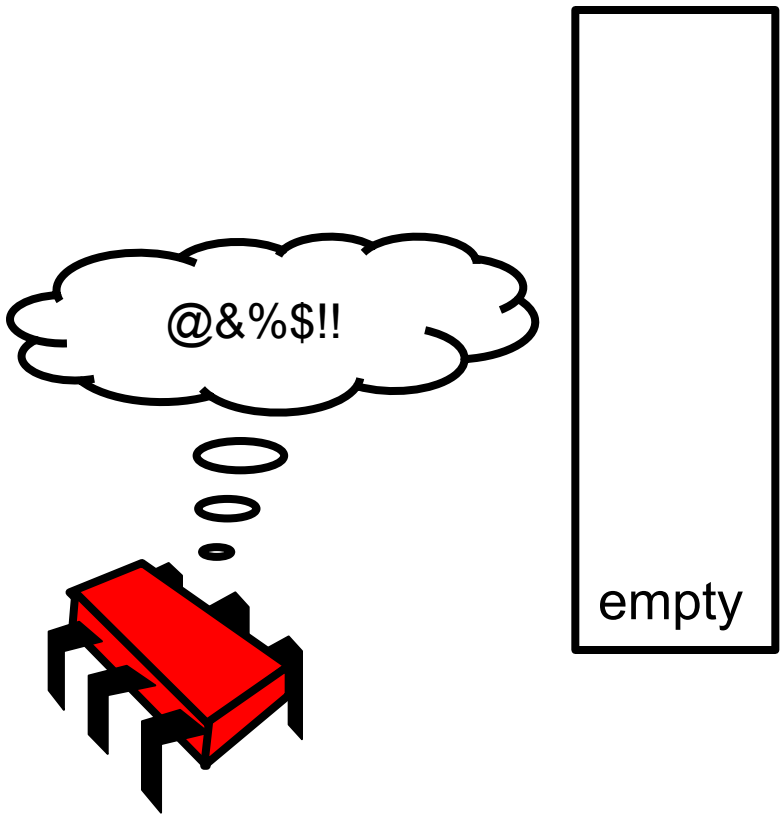


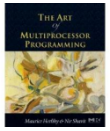
pushBottom





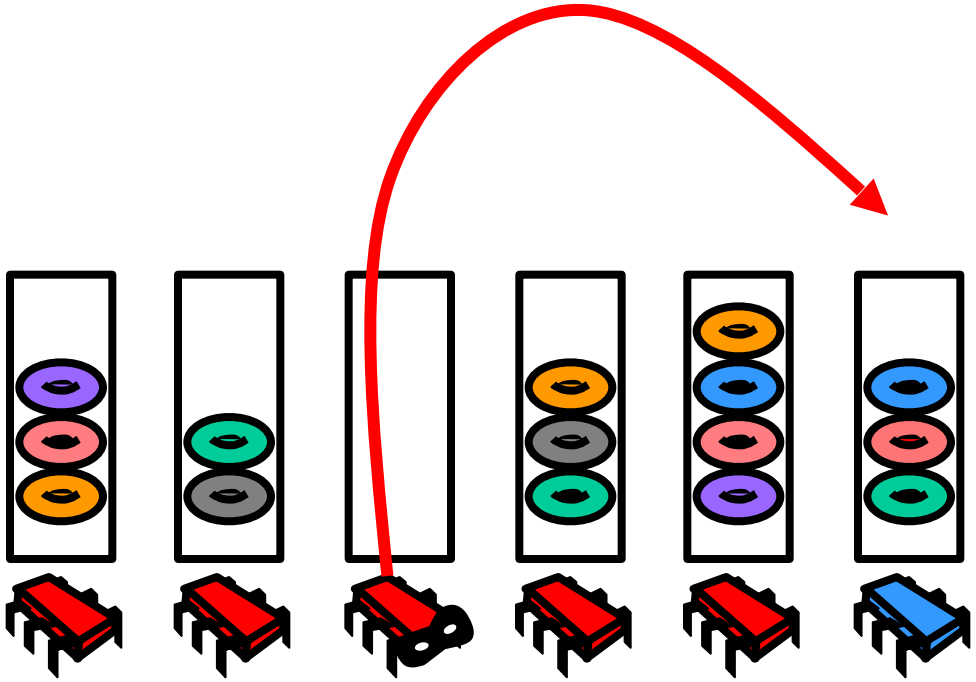
Whatcha Gonna do When the Well Runs Dry?

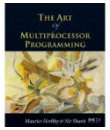




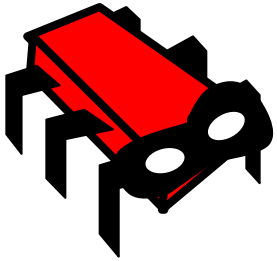
Steal Work from Others

Pick random thread's DEQueue

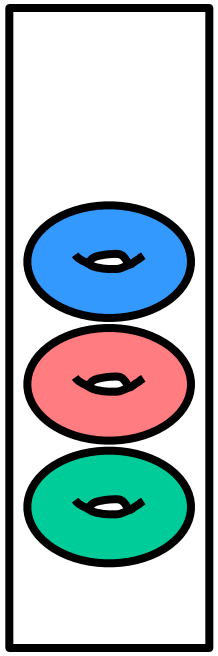


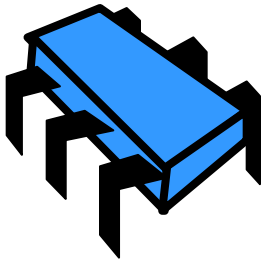


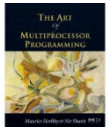
Steal this Task!



popTop



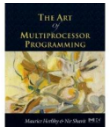




Task DEQueue

- Methods
  - pushBottom
  - popBottom
  - popTop

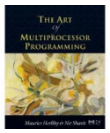
} Never happen concurrently



Task DEQueue

- Methods
  - pushBottom
  - popBottom
  - popTop

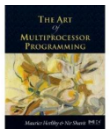
Most common – make them fast (minimize use of CAS)



Ideal

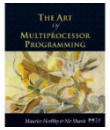
- Wait-Free
- Linearizable
- Constant time

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

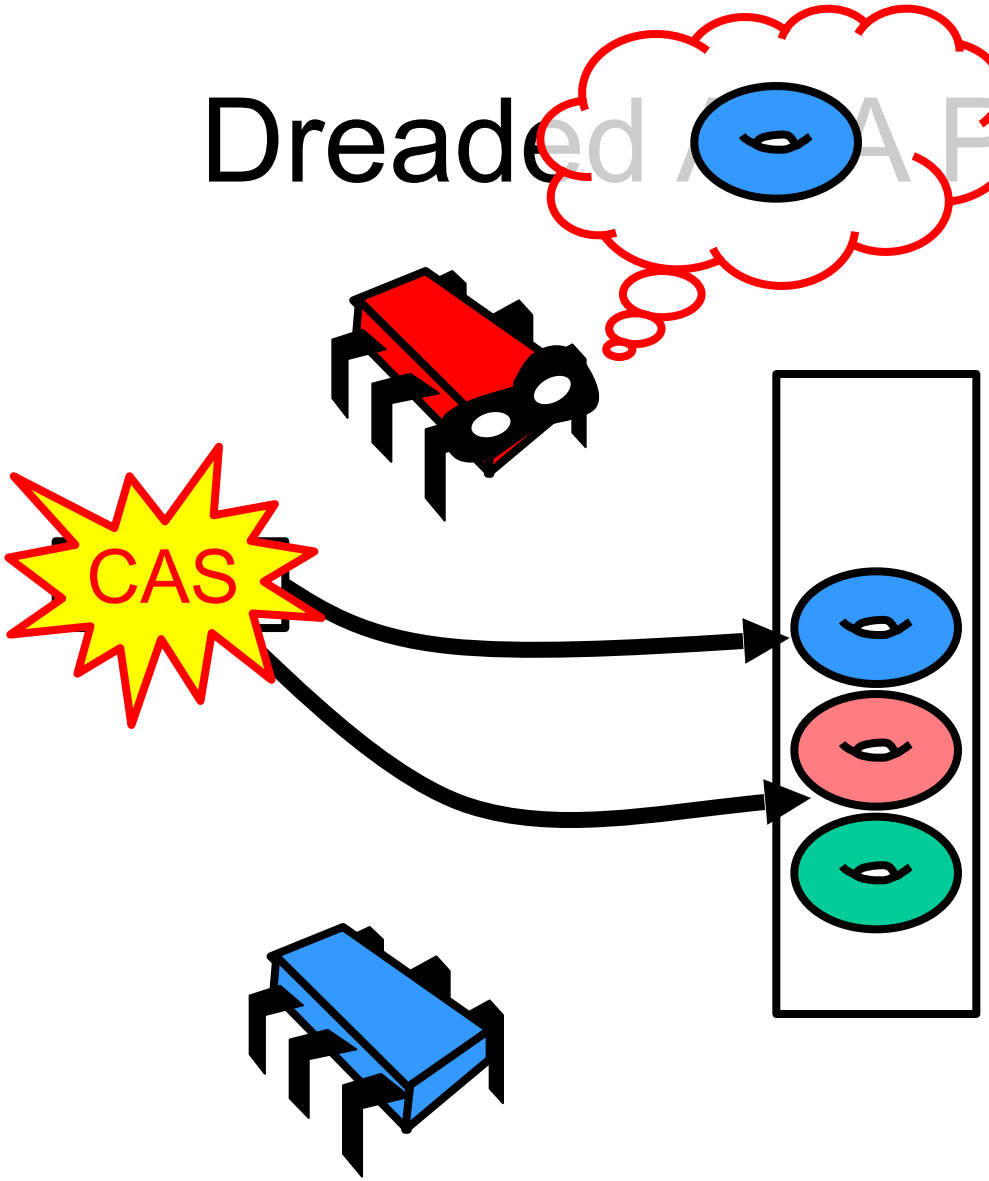


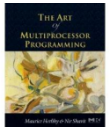
Compromise

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

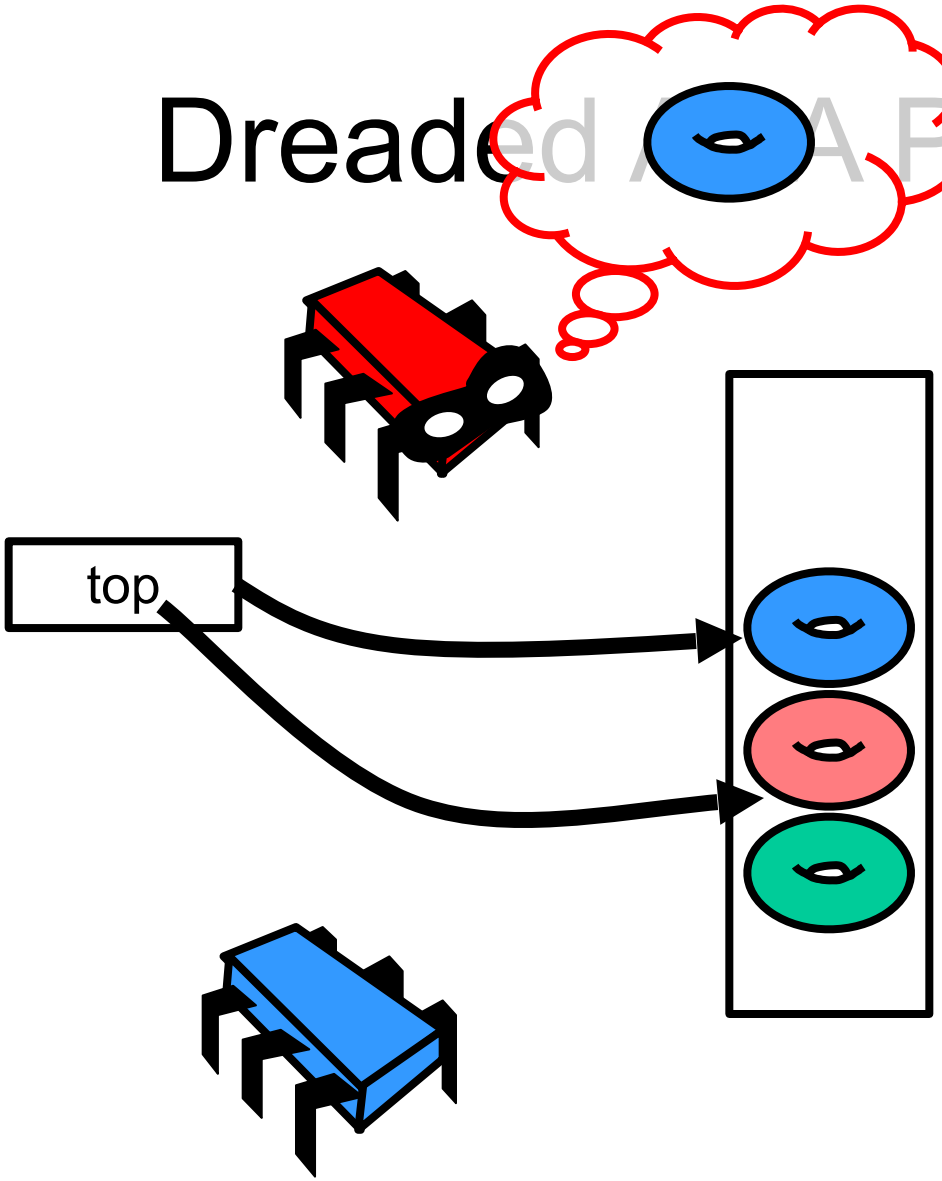


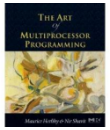
Dreaded ABA Problem



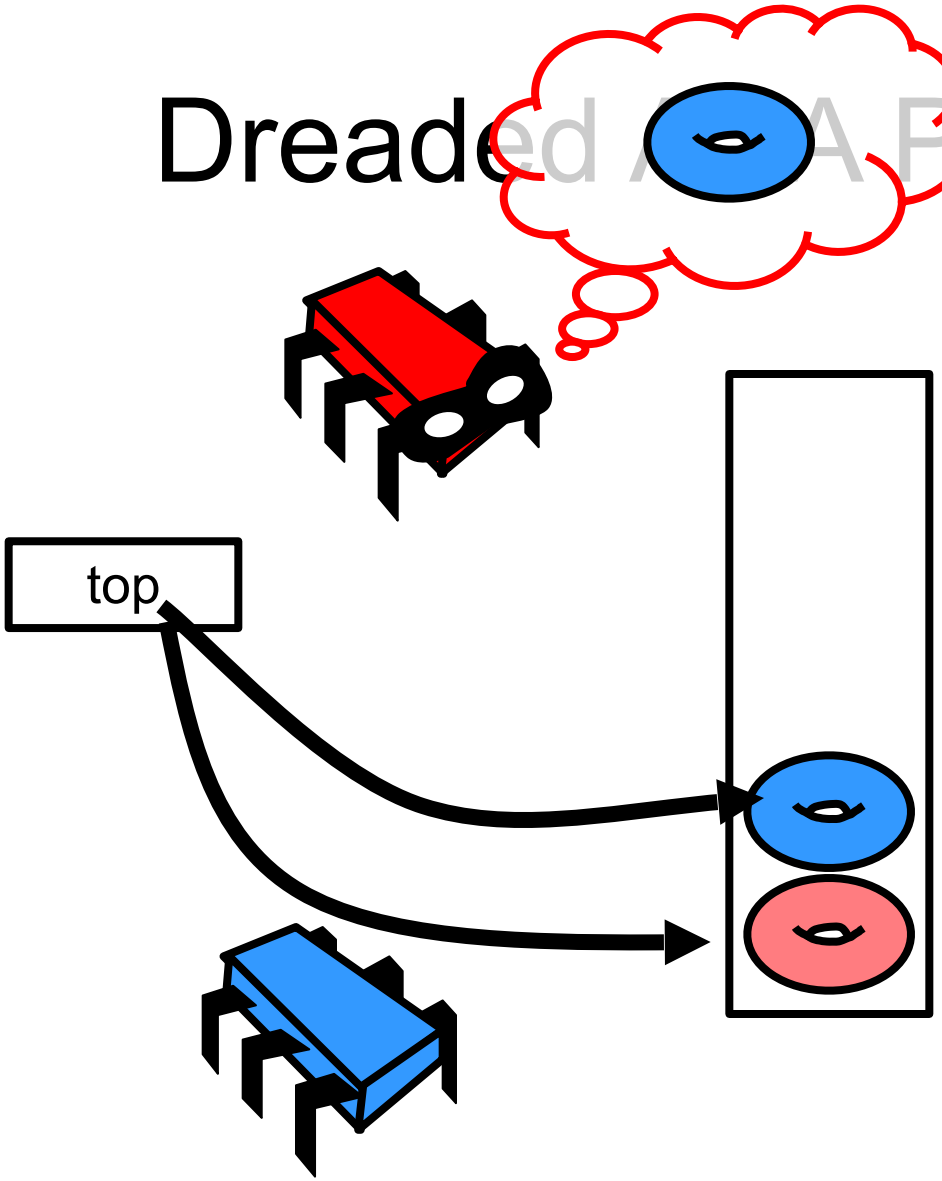


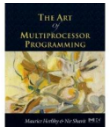
Dreaded ABA Problem



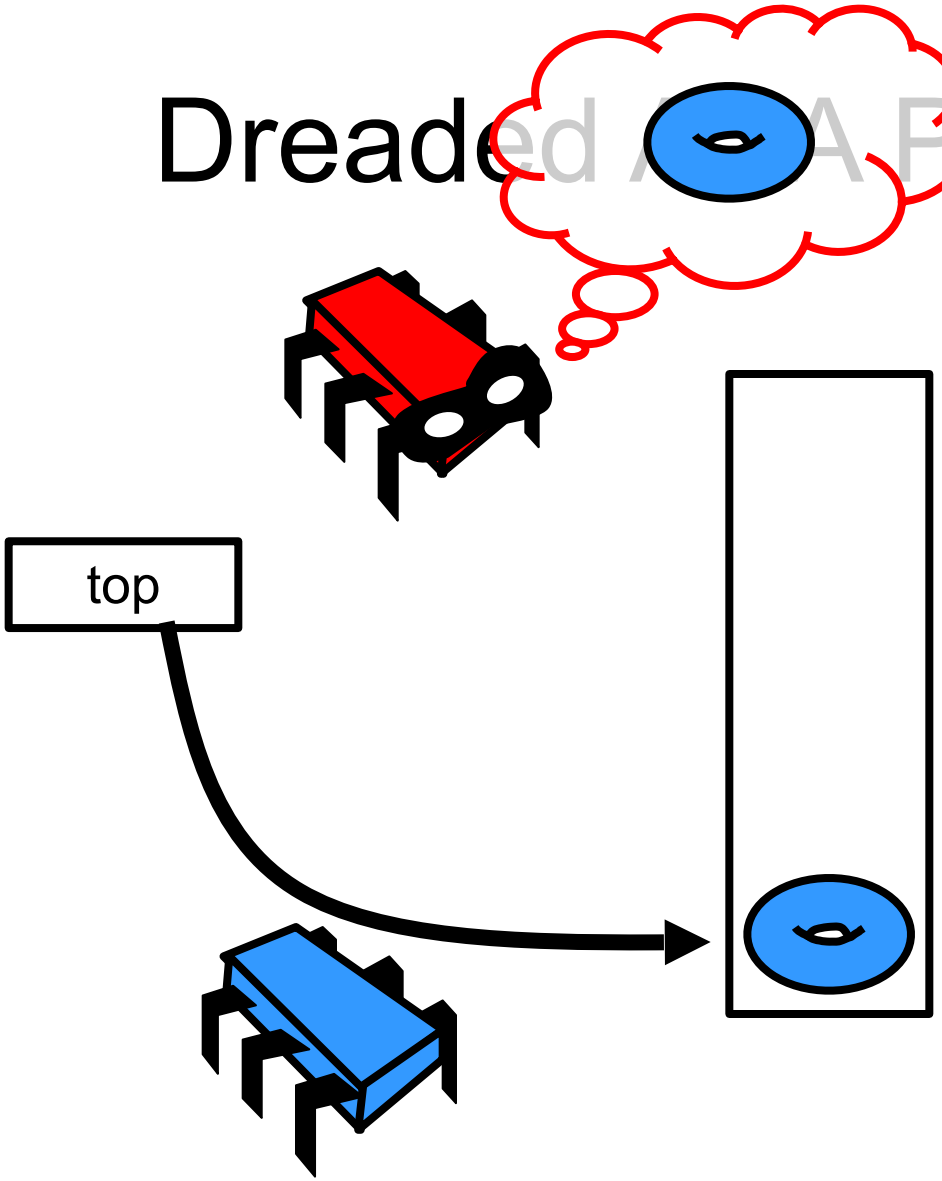


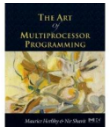
Dreaded ABA Problem



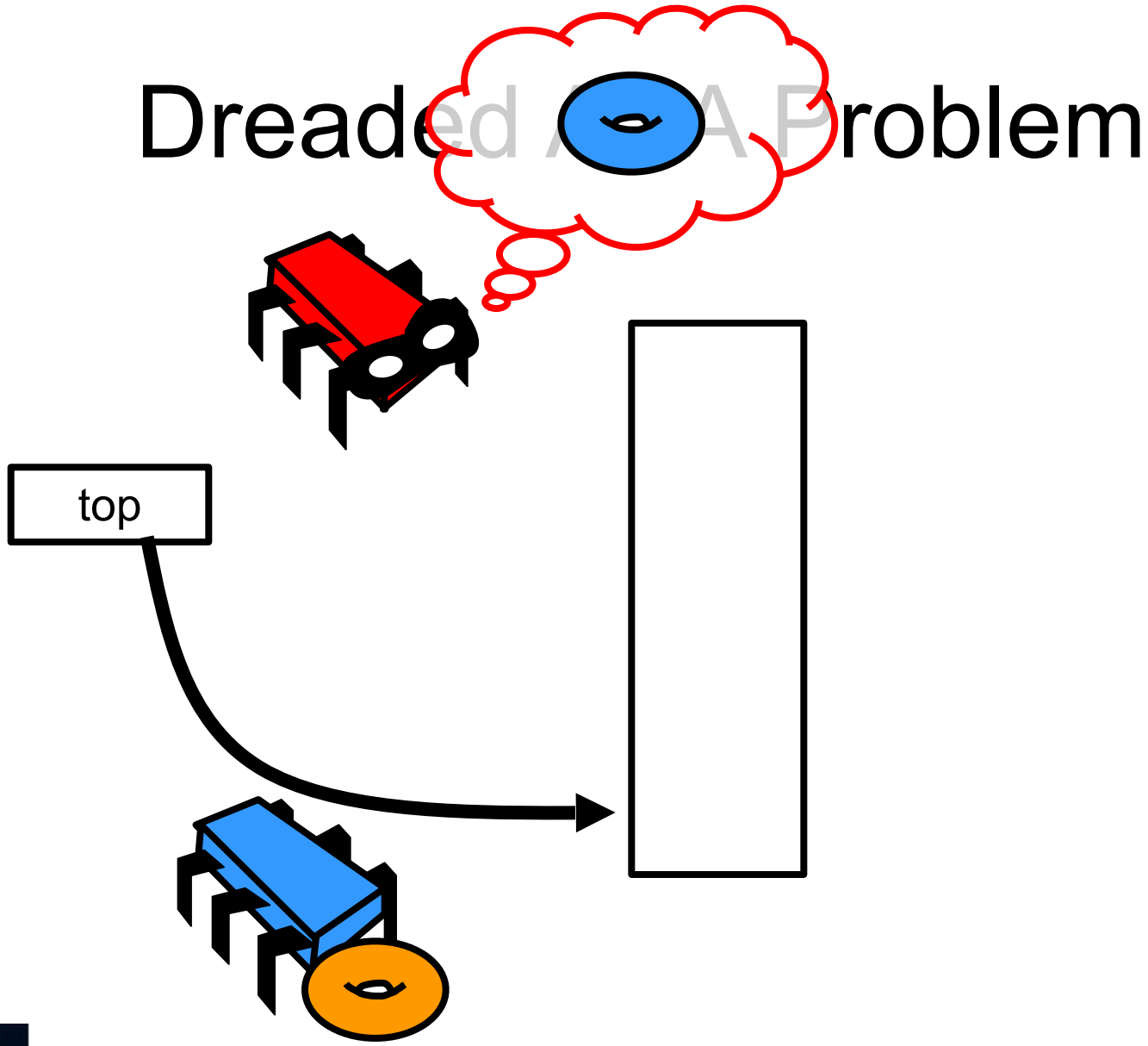


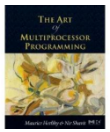
Dreaded ABA Problem



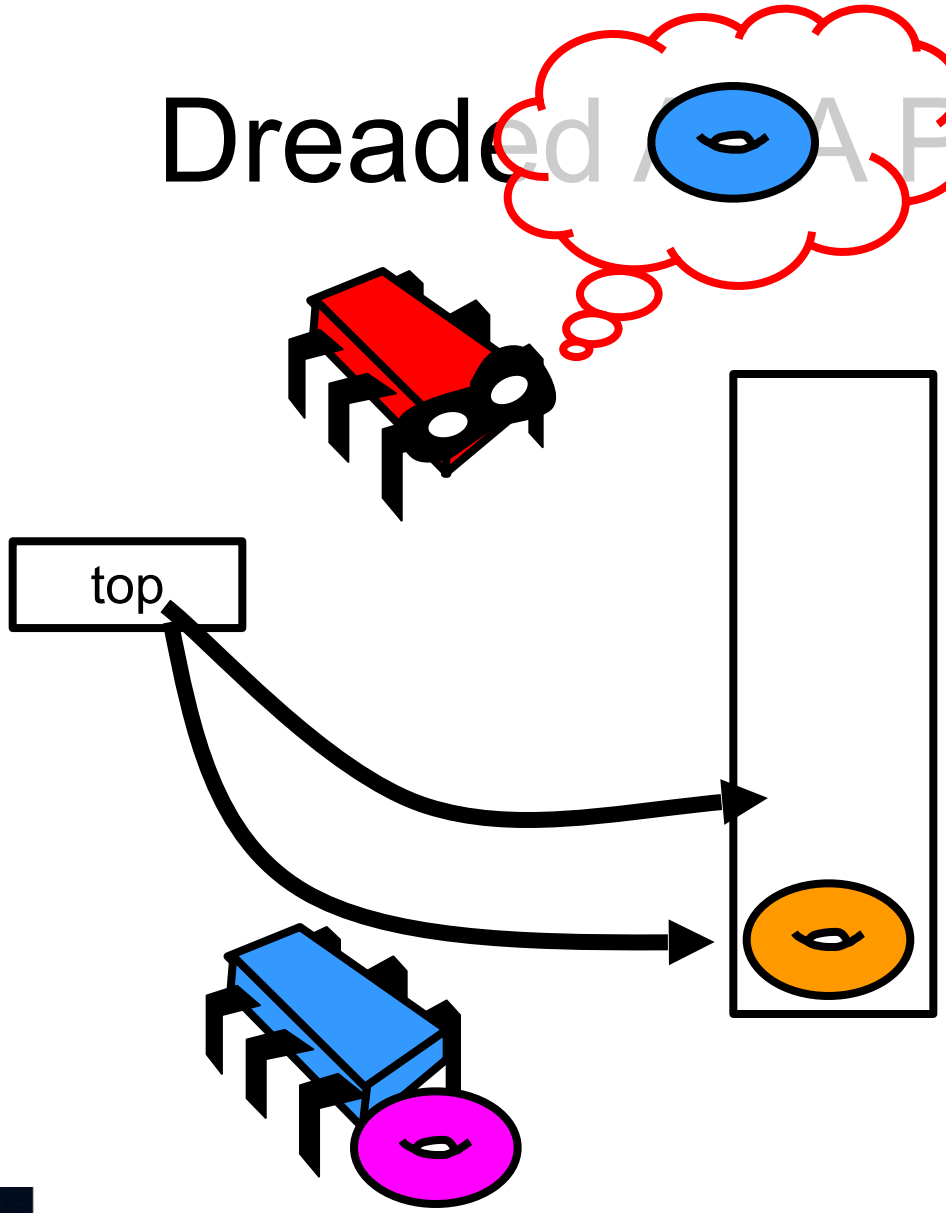


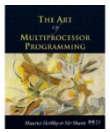
Dreaded ABA Problem



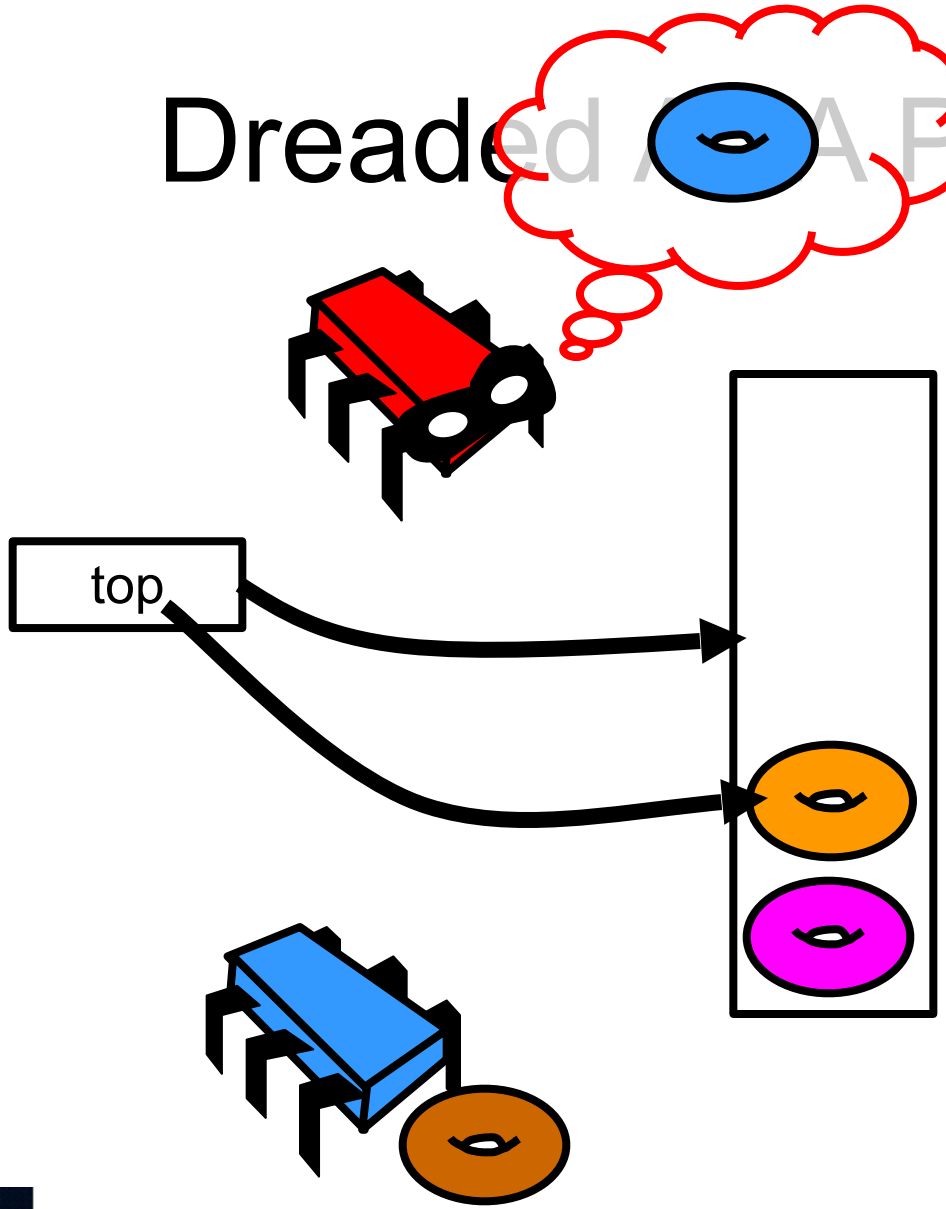


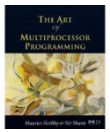
Dreaded ABA Problem



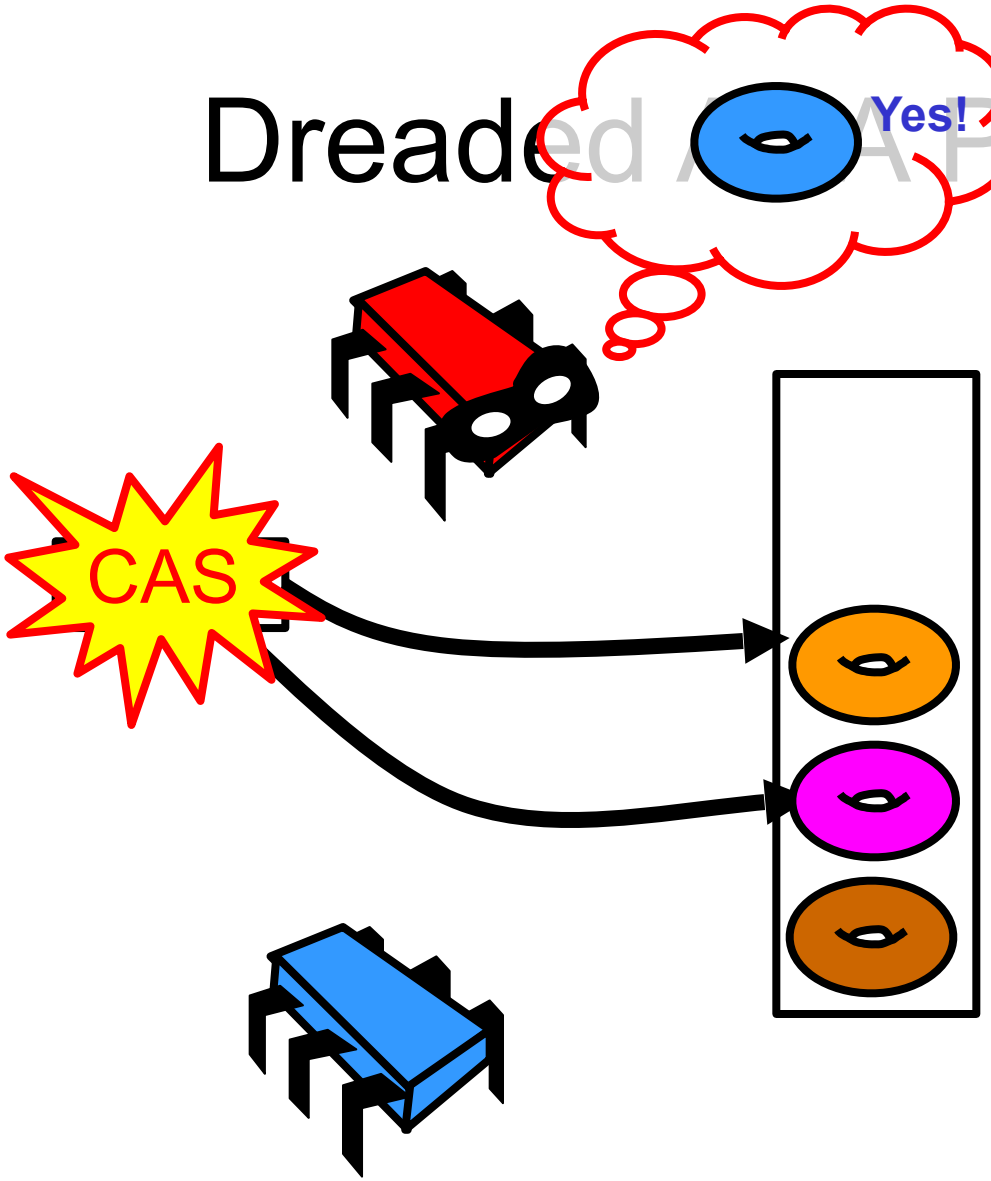


Dreaded ABA Problem

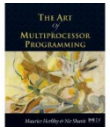




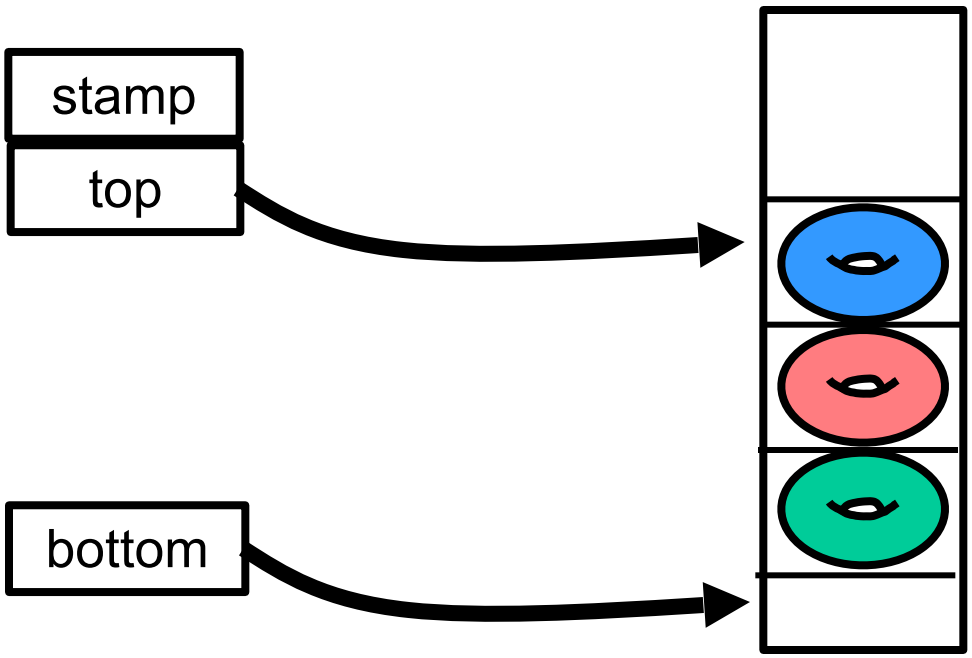
Dreaded ABA Problem

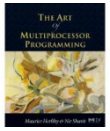


Uh-Oh …



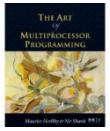
Fix for Dreaded ABA





Bounded DEQueue

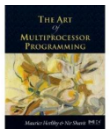
```
public class BDEQueue {
 AtomicStampedReference<Integer> top;
 volatile int bottom;
 Runnable[] tasks;
 …
}
```



Bounded DEQueue

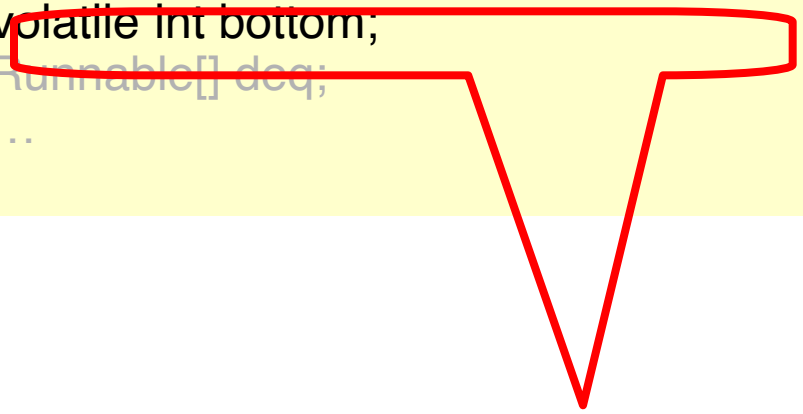
```
public class BDEQueue {
 AtomicStampedReference<Integer> top;
 volatile int bottom;
 Runnable[] tasks;
 …
}
```

Index & Stamp (synchronized)

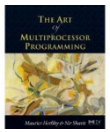


Bounded DEQueue

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

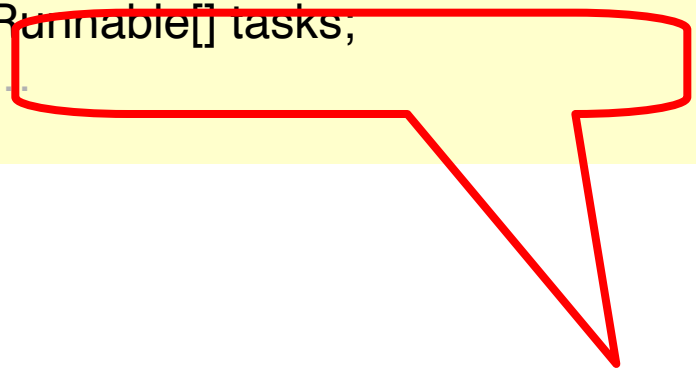


index of bottom task
no need to synchronize
memory barrier needed

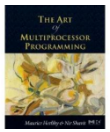


Bounded DEQueue

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

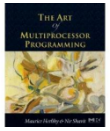


Array holding tasks



pushBottom()

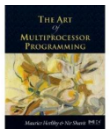
```
public class BDEQueue {
 …
 void pushBottom(Runnable r){
  tasks[bottom] = r;
  bottom++;
 }
 …
}
```



pushBottom()

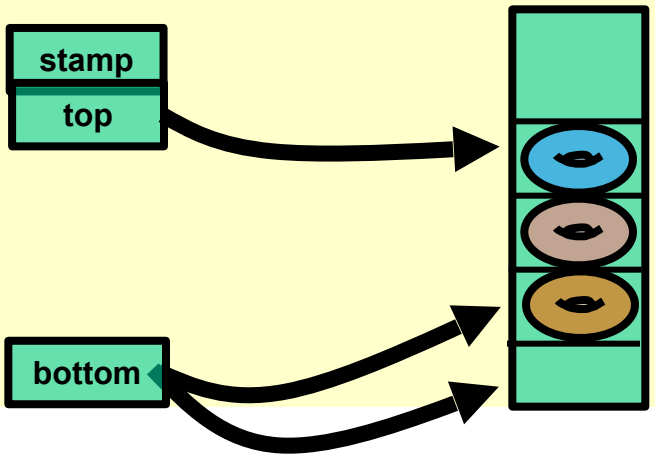
```
public class BDEQueue {
 …
 void pushBottom(Runnable r){
  tasks[bottom] = r;
  bottom++;
 }
 …
}
```

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

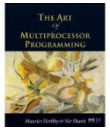


pushBottom()

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

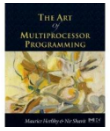


Adjust the bottom index



Steal Work

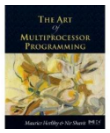
```
public Runnable popTop() {
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = oldTop + 1;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom <= oldTop)
   return null;
 Runnable r = tasks[oldTop];
 if (top.CAS(oldTop, newTop, oldStamp, newStamp)) return r;
 return null;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



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;
 }
```

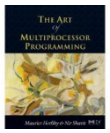
Read top (value & stamp)



Steal Work

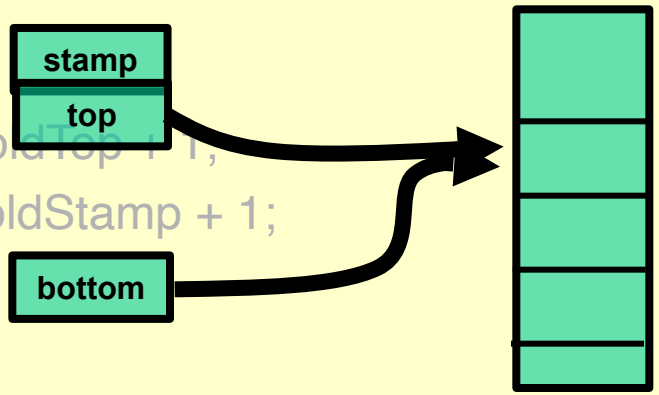
```
public Runnable popTop() {
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = oldTop + 1;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom <= oldTop)
   return null;
 Runnable r = tasks[oldTop];
 if (top.CAS(oldTop, newTop, oldStamp, newStamp)) return r;
 return null;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

Compute new value & stamp

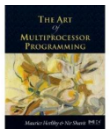


Steal Work

```
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;
}
```

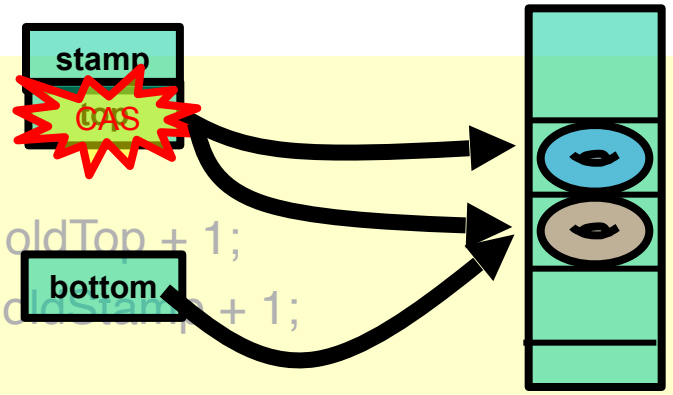


Quit if queue is empty

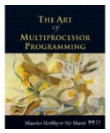


Steal Work

```
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;
}
```



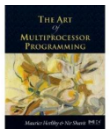
Try to steal the task



Steal Work

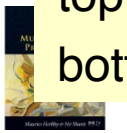
```
public Runnable popTop() {
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = oldTop + 1;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom <= oldTop)
   return null;
 Runnable r = tasks[oldTop];
 if (top.CAS(oldTop, newTop, oldStamp, newStamp)) return r;
 return null;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

Give up if conflict occurs



Take Work

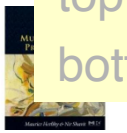
```
Runnable popBottom() {
 if (bottom == 0) return null;
 bottom--;
 Runnable r = tasks[bottom];
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = 0;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom > oldTop) return r;
 if (bottom == oldTop){
   bottom = 0;
   if (top.CAS(oldTop,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



Take Work

```
Runnable popBottom() {
f (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;
bottom = 0; }
```

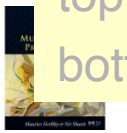
Make sure queue is non-empty



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;
 bottom = 0; }
```

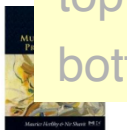
Prepare to grab bottom task



Take Work

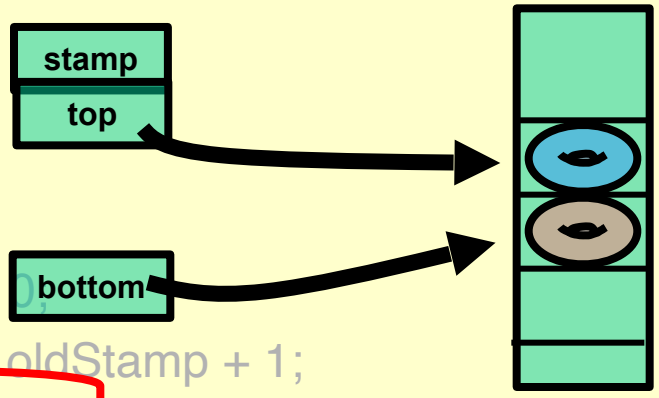
```
Runnable popBottom() {
 if (bottom == 0) return null;
 bottom--;
 Runnable r = tasks[bottom];
int[] stamp = new int[1];
int oldTop = top.get(stamp), newTop = 0;
int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom > oldTop) return r;
 if (bottom == oldTop){
   bottom = 0;
   if (top.CAS(oldTop,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

Read top, & prepare new values



Take Work

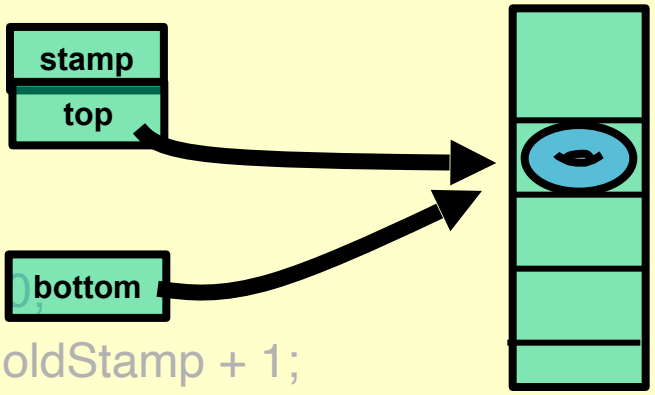
```
Runnable popBottom() {
 if (bottom == 0) return null;
 bottom--;
 Runnable r = tasks[bottom];
 int[] stamp = new int[1];
 int oldTop = top.get(stamp), newTop = 0;
 int oldStamp = stamp[0], newStamp = oldStamp + 1;
 if (bottom > oldTop) return r;
 if (bottom == oldTop){
   bottom = 0;
   if (top.CAS(oldTop,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



If top & bottom one 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;
 bottom = 0;}
```



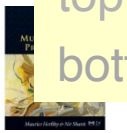
At most one item left

Take Work

Try to steal last task.

Reset bottom **because the DEQueue will be empty even if unsuccessful (why?)**

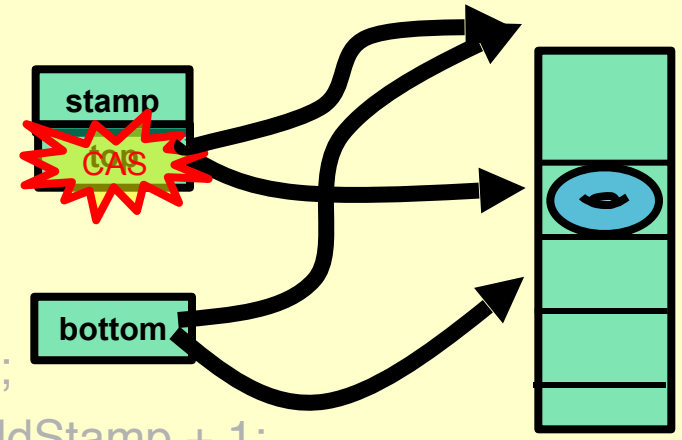
```
Runnable popBottom() {
 if (bottom == 0) return null;
 bottom--;
 Runnable r = tasks[bottom];
 int[] stamp = new int[1];
 int oldTop = top.get(stamp);
int oldStamp = stamp[0], newStamp = oldStamp + 1;
if (bottom > oldTop) return r;
if (bottom == oldTop){
 bottom = 0;
 if (top.CAS(oldTop, newTop,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

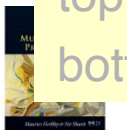


Take Work

```
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;
 bottom = 0;}
```

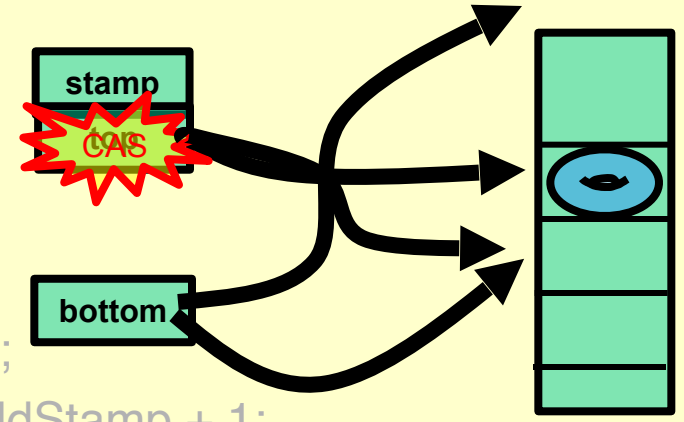
I win CAS





Take Work

If I lose CAS, thief must have won…




```
Runnable popBottom() {
if (bottom == 0) return null;
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;
bottom = 0;}
```

Take Work

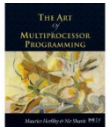
Failed to get last task (bottom could be less than top)

Must still reset top and bottom since deque is empty

```
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;
bottom = 0;}
```

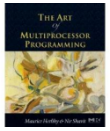
Old English Proverb

- "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

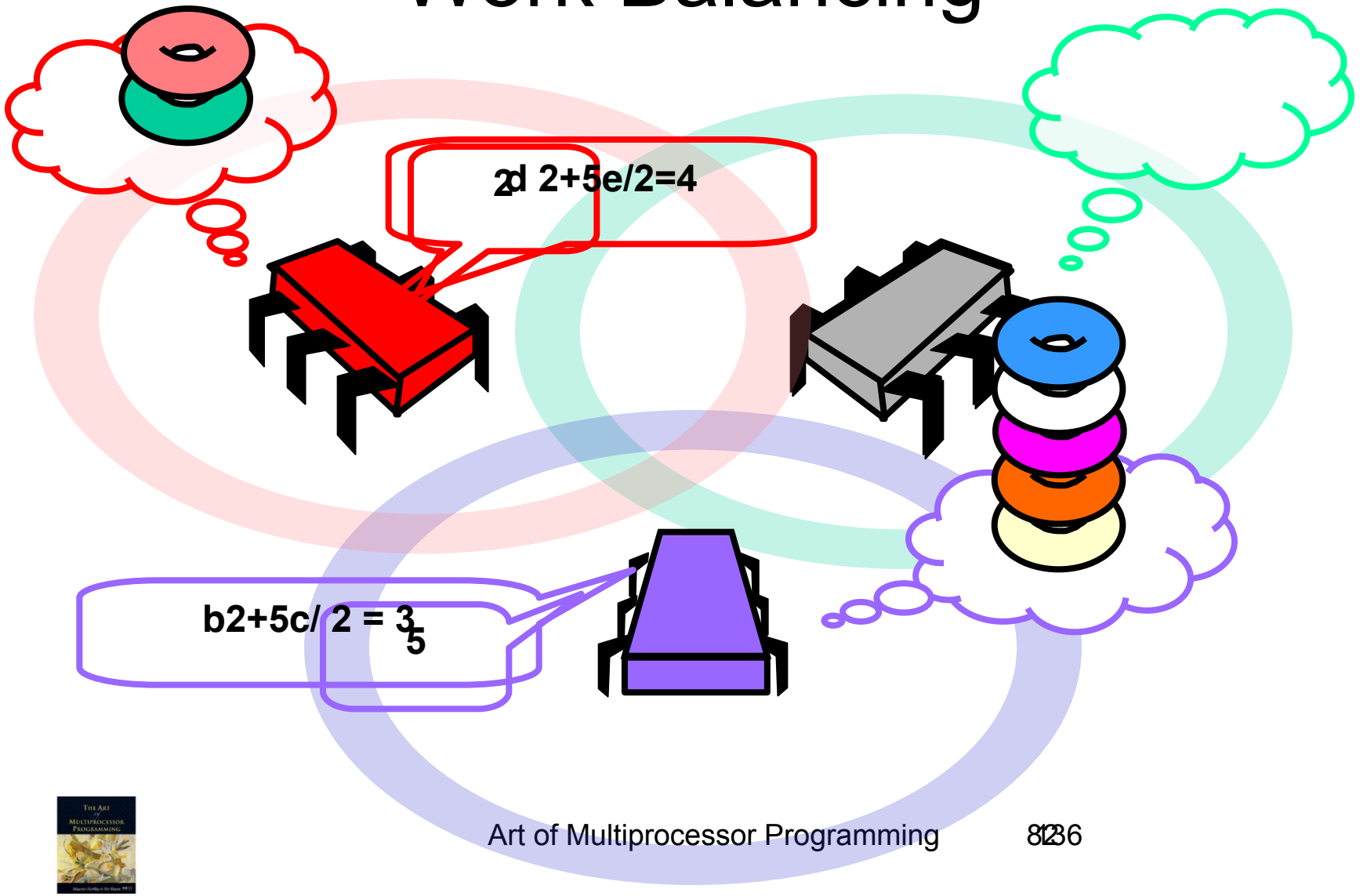


Variations

- Stealing is expensive
  - Pay CAS
  - Only one task taken
- What if
  - Move more than one task
  - Randomly balance loads?

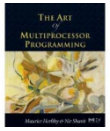


Work Balancing



Work-Balancing Thread

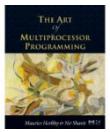
```
public void run() {
 int me = ThreadID.get();
 while (true) {
  Runnable task = queue[me].deq();
  if (task != null) task.run();
  int size = queue[me].size();
  if (random.nextInt(size+1) == size) {
   int victim = random.nextInt(queue.length);
   int min = …, max = …;
    synchronized (queue[min])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

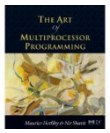
Keep running tasks



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]);
}}}}}
```

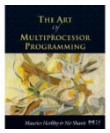
With probability 1/|queue|



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]);
}}}}}
```

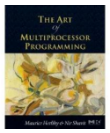
Choose random victim



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);
   nt min = …, max = …;
   synchronized (queue[min]) {
    synchronized (queue[max]) {
        balance(queue[min], queue[max]);
}}}}}
```

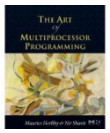
Lock queues in canonical order



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