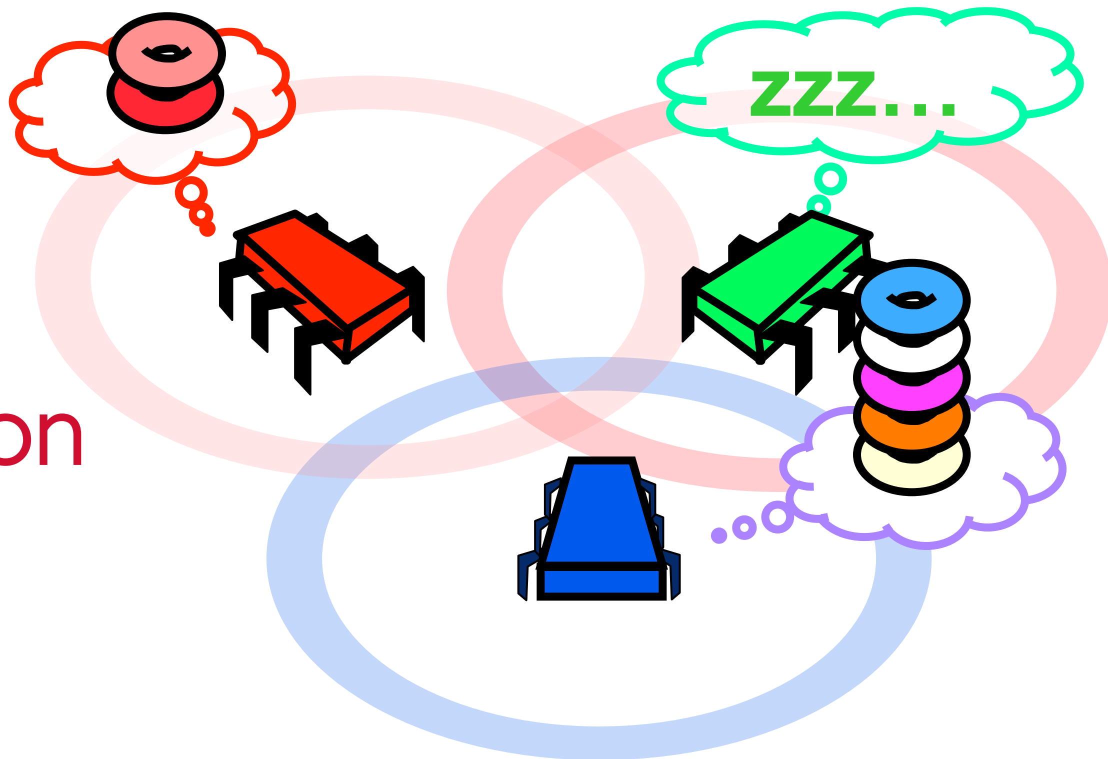
CS-206 Concurrency

Lecture 10

Scheduling & Work Distribution



Spring 2015
Prof. Babak Falsafi
parsa.epfl.ch/courses/cs206/

Adapted from slides originally developed by Maurice Herlihy and Nir Shavit from the Art of Multiprocessor Programming, and Babak Falsafi

EPFL Copyright 2015

Where are We?

M	T	W (Lecture & Lab)	T	F
16-Feb	17-Feb	18-Feb	19-Feb	20-Feb
23-Feb	24-Feb	25-Feb	26-Feb	27-Feb
2-Mar	3-Mar	4-Mar	5-Mar	6-Mar
9-Mar	10-Mar	11-Mar	12-Mar	13-Mar
16-Mar	17-Mar	18-Mar	19-Mar	20-Mar
23-Mar	24-Mar	25-Mar	26-Mar	27-Mar
30-Mar	31-Mar	1-Apr	2-Apr	3-Apr
6-Apr	7-Apr	8-Apr	9-Apr	10-Apr
13-Apr	14-Apr	15-Apr	16-Apr	17-Apr
20-Apr	21-Apr	22-Apr	23-Apr	24-Apr
27-Apr	28-Apr	29-Apr	30-Apr	1-May
4-May		6-May	7-May	8-May
11-May	12-May	13-May	14-May	15-May
18-May	19-May	20-May	21-May	22-May
25-May	26-May	27-May	28-May	29-May

- Work scheduling
  - Threads vs. Futures
  - Dependencies
- Work distribution
  - Work stealing
  - Work balancing
- Next week
  - GPUs

Evaluations: Sample feedback

- Faire faire aux étudiants des devoirs notés …et ne pas les avoir rendu alors que la fin du semestre approche est pénible et démotivant.
- I find (it) strange to forbid (the) use of laptops in a CS course in the 21st century...
- ….The midterm was only about learning the slides and did not require any reflection, it's not what we expect of an engineer.
- More explanations on slides would be a plus...Some notes would be useful for the exam
- The class is alarmingly average. But that's not necessarily a bad thing.
- The course is dynamic and interesting, well structured. The lab sessions are adapted. I enjoy going for class.
- Good course, Babak is cool, but the slides are poor.
- The slides are hard to read and understand…What do Bob & Alice have to do with concurrency????
- Le professeur ne se donne pas beaucoup de peine pour faire un cours pédagogique. Bon diaporama (fr.wikipedia.org/wiki/Diaporama).
- Excellent professor. Really good assignments and enough time to do it

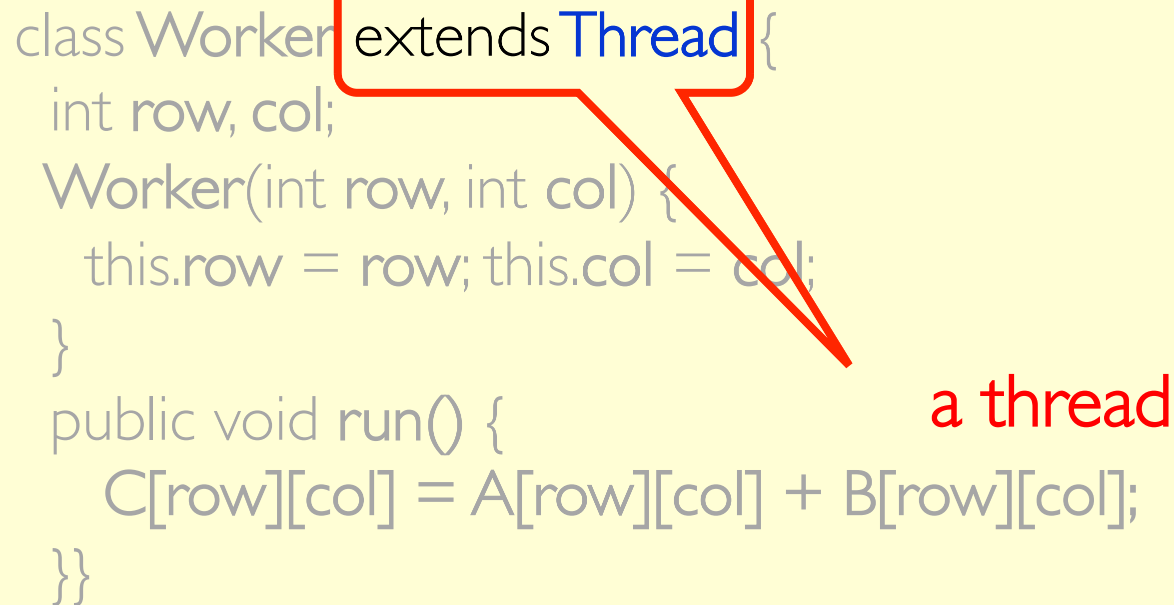
Matrix Add

$$\left(C\right)=\left(A\right)+\left(B\right)$$

Matrix Add

```
class Worker extends Thread {
  int row, col;
  Worker(int row, int col) {
    this.row = row; this.col = col;
  }
  public void run() {
    C[row][col] = A[row][col] + B[row][col];
  }}
```

Matrix Add



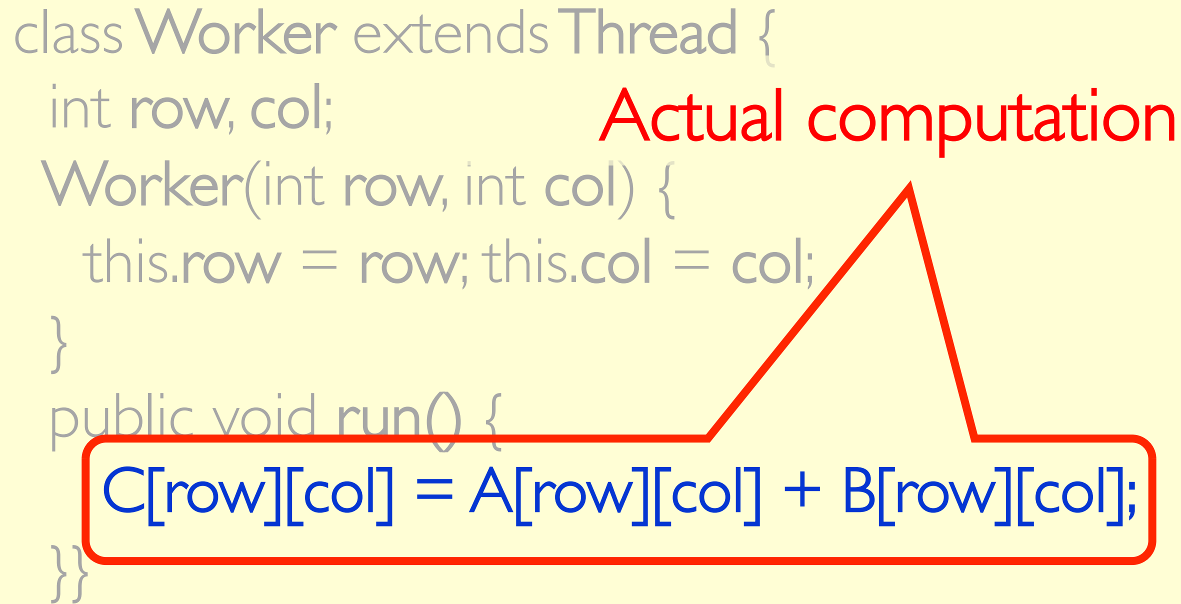

```
class Worker extends Thread {
  int row, col;
  Worker(int row, int col) {
    this.row = row; this.col = col;
  }
  public void run() {
    C[row][col] = A[row][col] + B[row][col];
  }}
```

Matrix Add

```
class Worker extends Thread {
  int row, col;
  Worker(int row, int col) {
    this.row = row; this.col = col;
  }
  public void run() {
     C[row][col] = A[row][col] + B[row][col];
  }}
```

Which matrix entry to compute

Matrix Add



Matrix Add

```
void add() {
  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();
}
```

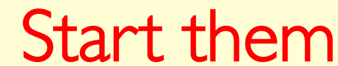
Matrix Add

```
void add() {
 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 Add

```
void add() {
 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 Add

```
void add() {
  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 Add

```
void add() {
  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

- 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();
```

Example Overhead: Threads vs. Futures

- Creating threads: ~200**µ**s
- Submitting tasks to futures: ~10**µ**s

- Each task will do a certain amount of work

- What fraction of overall time is in task submission?
  - For tasks with 1**µ**s of work?
  - For tasks with 1**m**s of work?

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

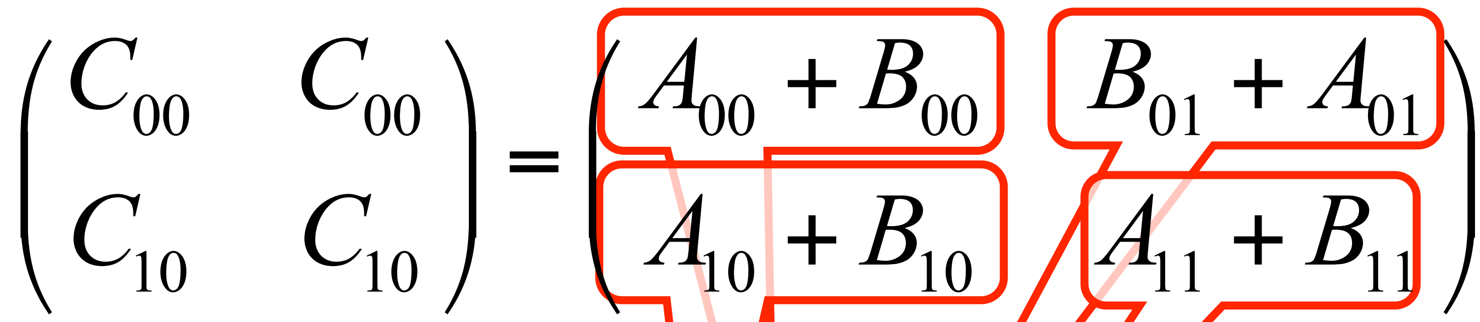
Note

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

Matrix Addition (4 elements)

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

Matrix Addition (4 elements)



Matrix Addition (4 elements)

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

- How do we add larger matrices?
  - Hint: for now assume dimensions are power of 2

Matrix Addition Task

```
public class Matrix {
int dim;
double[][] data;
int rowDisplace, colDisplace;
  ...
  public double get(int row, int col) {
    return data[row+rowDisplace][col+colDisplace];
  }

  public void set(int row, int col, double value) {
    data[row+rowDisplace][col+colDisplace] = value;
  }

  Matrix[][] split();
…
}
```

Matrix Addition Task

```
public class Matrix {
int dim;
double[][] data;
int rowDisplace, colDisplace;
   …
   public double get(int row, int col) {
      return data[row+rowDisplace][col+colDisplace];
   }

   public void set(int row, int col, double value) {
      data[row+rowDisplace][col+colDisplace] = value;
   }

   Matrix[][] split();
…
}
```

Data matrix reference

Matrix Addition Task

```
public class Matrix {
int dim;
double[][] data;
int rowDisplace, colDisplace;
   …
   public double get(int row, int col) {
      return data[row+rowDisplace][col+colDisplace];
   }

   public void set(int row, int col, double value) {
      data[row+rowDisplace][col+colDisplace] = value;
   }

   Matrix[][] split();
…
}
```

Access data through displacement variables…

Matrix Addition Task

```
public class Matrix {
int dim;
double[][] data;
int rowDisplace, colDisplace;
   …
   public double get(int row, int col) {
      return data[row+rowDisplace][col+colDisplace];
   }

   public void set(int row, int col, double value) {
      data[row+rowDisplace][col+colDisplace] = value;
   }

   Matrix[][] split();
…
}
```

..so we can split the matrix in constant time

Matrix Addition Task

```
Matrix[][] split() {
   Matrix[][] result = new Matrix[2][2];
   int newDim = dim / 2;
   result[0][0] =
         new Matrix(data, rowDisplace, colDisplace, newDim);
   result[0][1] =
         new Matrix(data, rowDisplace,
                        colDisplace + newDim, newDim);
   result[1][0] =
          new Matrix(data, rowDisplace + newDim,
                         colDisplace, newDim);
   result[1][1] =
         new Matrix(data, rowDisplace + newDim,
                        colDisplace + newDim, newDim);
   return result;
 }
```

Matrix Addition Task

```
Matrix[][] split() {
  Matrix[][] result = new Matrix[2][2];
  int newDim = dim / 2;
  result[0][0] =
      new Matrix(data, rowDisplace, colDisplace, newDim);
  result[0][1] =
      new Matrix(data, rowDisplace,
                       colDisplace + newDim, newDim);
  result[1][0] =
       new Matrix(data, rowDisplace + newDim,
                        colDisplace, newDim);
  result[1][1] =
      new Matrix(data, rowDisplace + newDim,
                       colDisplace + newDim, newDim);
  return result;
}
```

Split into 4 matrices..

Matrix Addition Task

```
Matrix[][] split() {
    Matrix[][] result = new Matrix[2][2];
    int newDim = dim / 2;
    result[0][0] =
        new Matrix(data, rowDisplace, colDisplace, newDim);
    result[0][1] =
        new Matrix(data, rowDisplace,
                   colDisplace + newDim, newDim);
    result[1][0] =
        new Matrix(data, rowDisplace + newDim,
                   colDisplace, newDim);
    result[1][1] =
        new Matrix(data, rowDisplace + newDim,
                   colDisplace + newDim, newDim);
    return result;
}
```

…with half dimentions

Matrix Addition Task

```
Matrix[][] split() {
    Matrix[][] result = new Matrix[2][2];
    int newDim = dim / 2;
    result[0][0] =
        new Matrix(data, rowDisplace, colDisplace, newDim);
    result[0][1] =
        new Matrix(data, rowDisplace,
                        colDisplace + newDim, newDim);
    result[1][0] =
        new Matrix(data, rowDisplace + newDim,
                        colDisplace, newDim);
    result[1][1] =
        new Matrix(data, rowDisplace + newDim,
                        colDisplace + newDim, newDim);
    return result;
}
```

Reassign displacement variables, no data copy

Matrix Addition Task

```
class AddTask implements Runnable {
  Matrix a, b, c; // add this!
  public void run() {
    if (a.dim == 1) {
      c.set(0,0,a.get(0,0) + b.get(0,0)); // base case
    } else {
     Matrix[][] splitA = a.split(); Matrix[][] splitB = b.split();
     Matrix[][] splitC = c.split();
     Future<?> f[][] = new Future<?>[2][2];
     Future<?> f[0][0] =
          exec.submit(new AddTask(splitA[0][0],splitB[0][0],splitC[0][0]));
     …
     Future<?> f[1][1] =
          exec.submit(new AddTask(splitA[1][1],splitB[1][1], splitC[1][1]));
     f[0][0].get(); …; f[1][1].get();
     …
    }}
```

Matrix Addition Task

```
class AddTask implements Runnable {
  Matrix a, b, c; // add this!
  public void run() {
    if (a.dim == 1) {
      c.set(0,0,a.get(0,0) + b.get(0,0)); // base case
    } else {
     Matrix[][] splitA = a.split(); Matrix[][] splitB = b.split();
     Matrix[][] splitC = c.split();
     Future<?> f[][] = new Future<?>[2][2];
     Future<?> f[0][0] =
          exec.submit(new AddTask(splitA[0][0],splitB[0][0],splitC[0][0]));
     …
     Future<?> f[1][1] =
          exec.submit(new AddTask(splitA[1][1],splitB[1][1], splitC[1][1]));
     f[0][0].get(); …; f[1][1].get();
     …
    }}
```

Constant-time operation

Matrix Addition Task

```
class AddTask implements Runnable {
  Matrix a, b, c; // add this!
  public void run() {
   if (a.dim == 1) {
     c.set(0,0,a.get(0,0) + b.get(0,0)); // base case
   } else {
    Matrix[][] splitA = a.split(); Matrix[][] splitB = b.split();
    Matrix[][] splitC = c.split();
    Future<?> f[][] = new Future<?>[2][2];
    Future<?> f[0][0] =
          exec.submit(new AddTask(splitA[0][0],splitB[0][0],splitC[0][0]));
    …
    Future<?> f[1][1] =
          exec.submit(new AddTask(splitA[1][1],splitB[1][1], splitC[1][1]));
    f[0][0].get(); …; f[1][1].get();
    …
   }}
```

Matrix Addition Task

```
class AddTask implements Runnable {
  Matrix a, b, c; // add this!
  public void run() {
   if (a.dim == 1) {
     c.set(0,0,a.get(0,0) + b.get(0,0)); // base case
   } else {
    Matrix[][] splitA = a.split(); Matrix[][] splitB = b.split();
    Matrix[][] splitC = c.split();
    Future<?> f[][] = new Future<?>[2][2];
    Future<?> f[0][0] =
         exec.submit(new AddTask(splitA[0][0],splitB[0][0],splitC[0][0]));
    …
    Future<?> f[1][1] =
         exec.submit(new AddTask(splitA[1][1],splitB[1][1], splitC[1][1]));
    f[0][0].get(); …; f[1][1].get();
    …
   }}
```

Base case: add directly

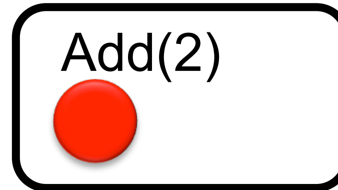
Matrix Addition Task

```
class AddTask implements Runnable {
  Matrix a, b, c; // add this!
  public void run() {
   if (a.dim == 1) {
     c.set(0,0,a.get(0,0) + b.get(0,0)); // base case
   } else {
    Matrix[][] splitA = a.split(); Matrix[][] splitB = b.split();
    Matrix[][] splitC = c.split();
    Future<?> f[][] = new Future<?>[2][2];
    Future<?> f[0][0] =
          exec.submit(new AddTask(splitA[0][0],splitB[0][0],splitC[0][0]));
    …
    Future<?> f[1][1] =
          exec.submit(new AddTask(splitA[1][1],splitB[1][1], splitC[1][1]));
    f[0][0].get(); …; f[1][1].get();
    …
   }}
```

Let them finish

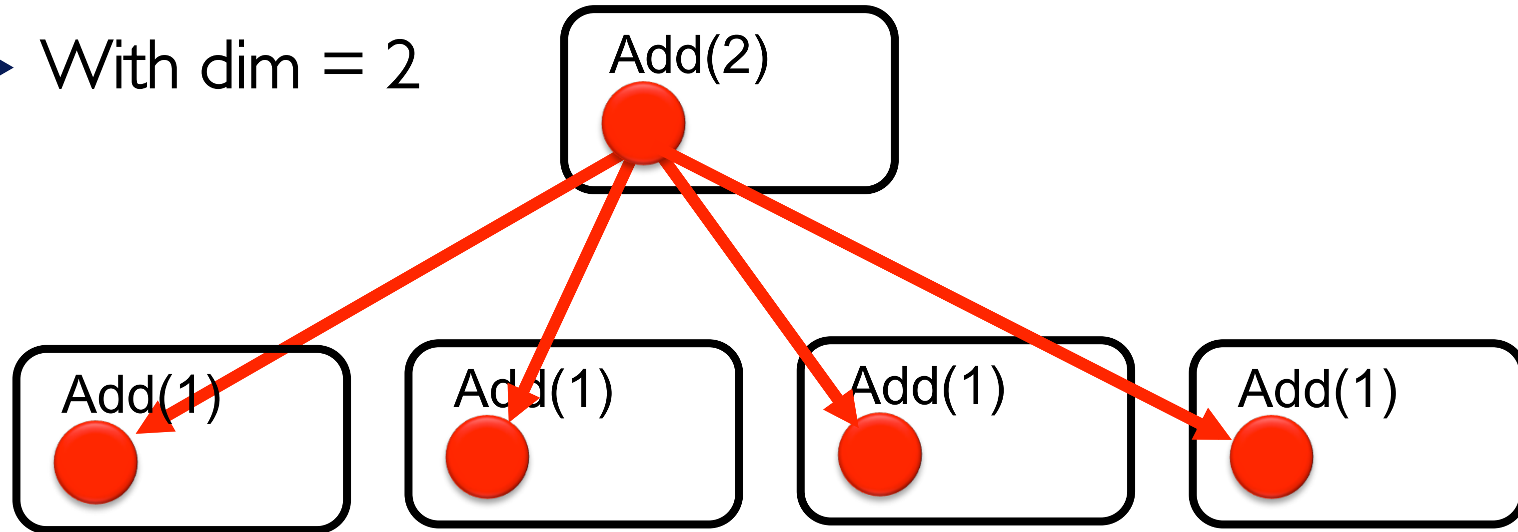
Matrix Addition DAG

- With dim = 2



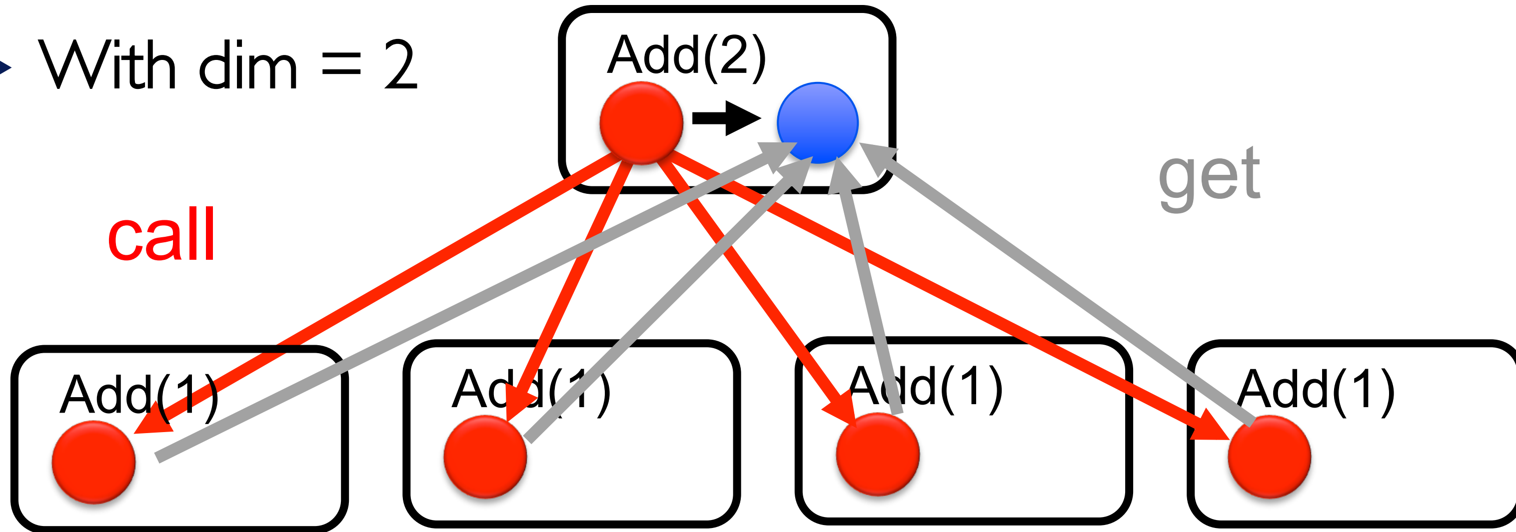
Matrix Addition DAG

- With dim = 2



Matrix Addition DAG

- With dim = 2



Dependencies

- Matrix example is not typical
- Individual tasks are independent
  - Don't need results of one task …
  - Only need to split recursively
- Often task results 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 try this at home or job!
  - But illustrates our point
    - How to deal with dependencies

- Exercise:
  - Make this implementation efficient!

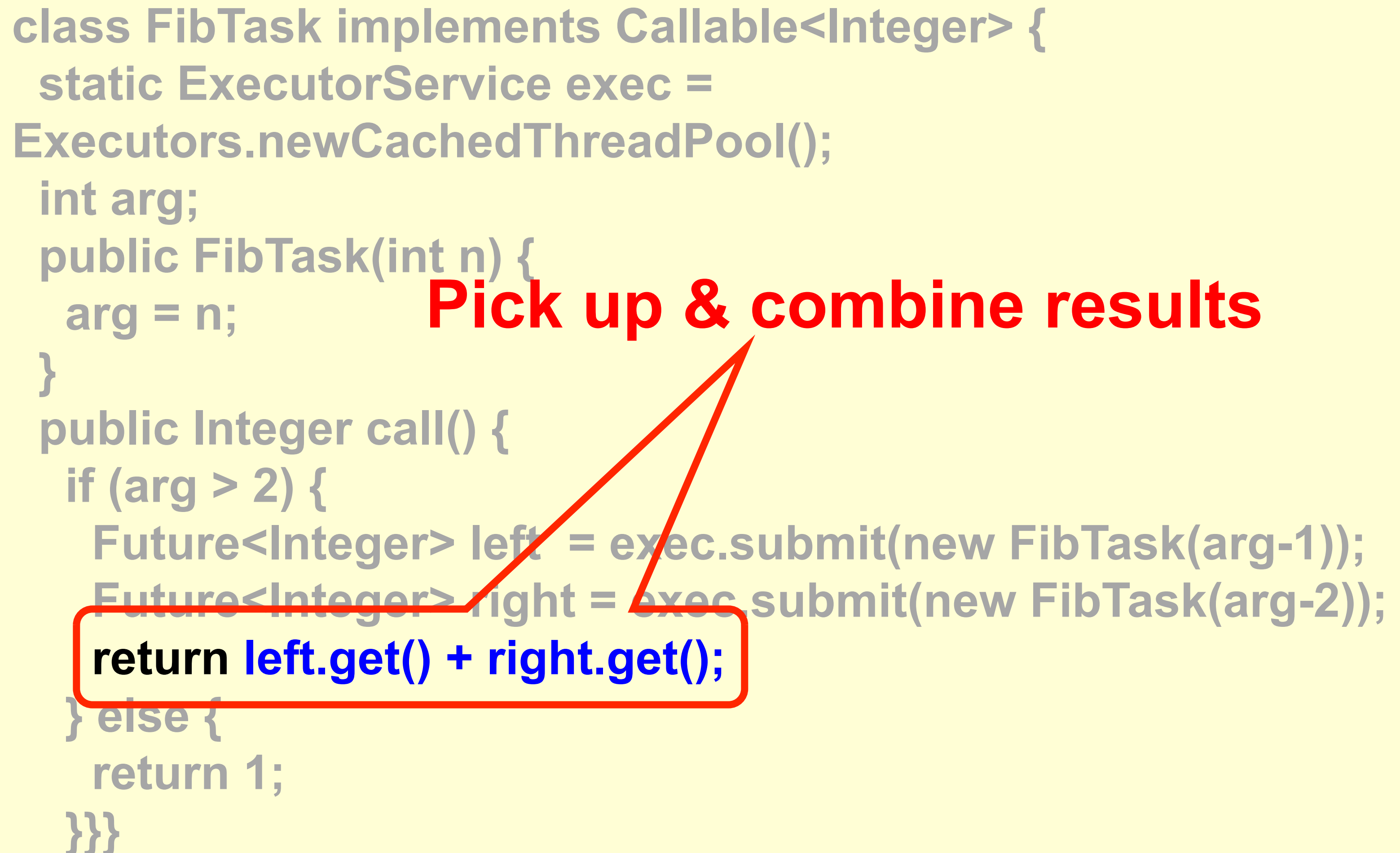
Multithreaded Fibonacci

```
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;
    }}}
```

Multithreaded Fibonacci



The Blumofe-Leiserson DAG Model

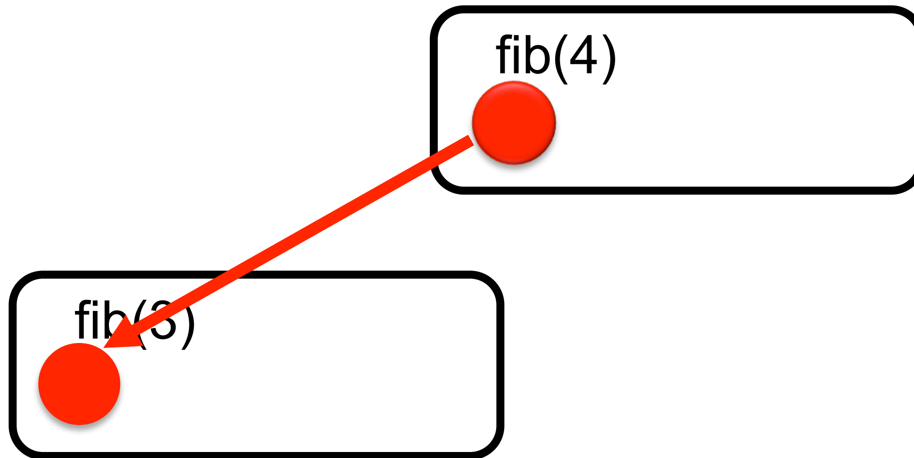
- Multithreaded program is
  - A directed acyclic graph (DAG)
  - That unfolds dynamically

- Each node is
  - A single unit of work

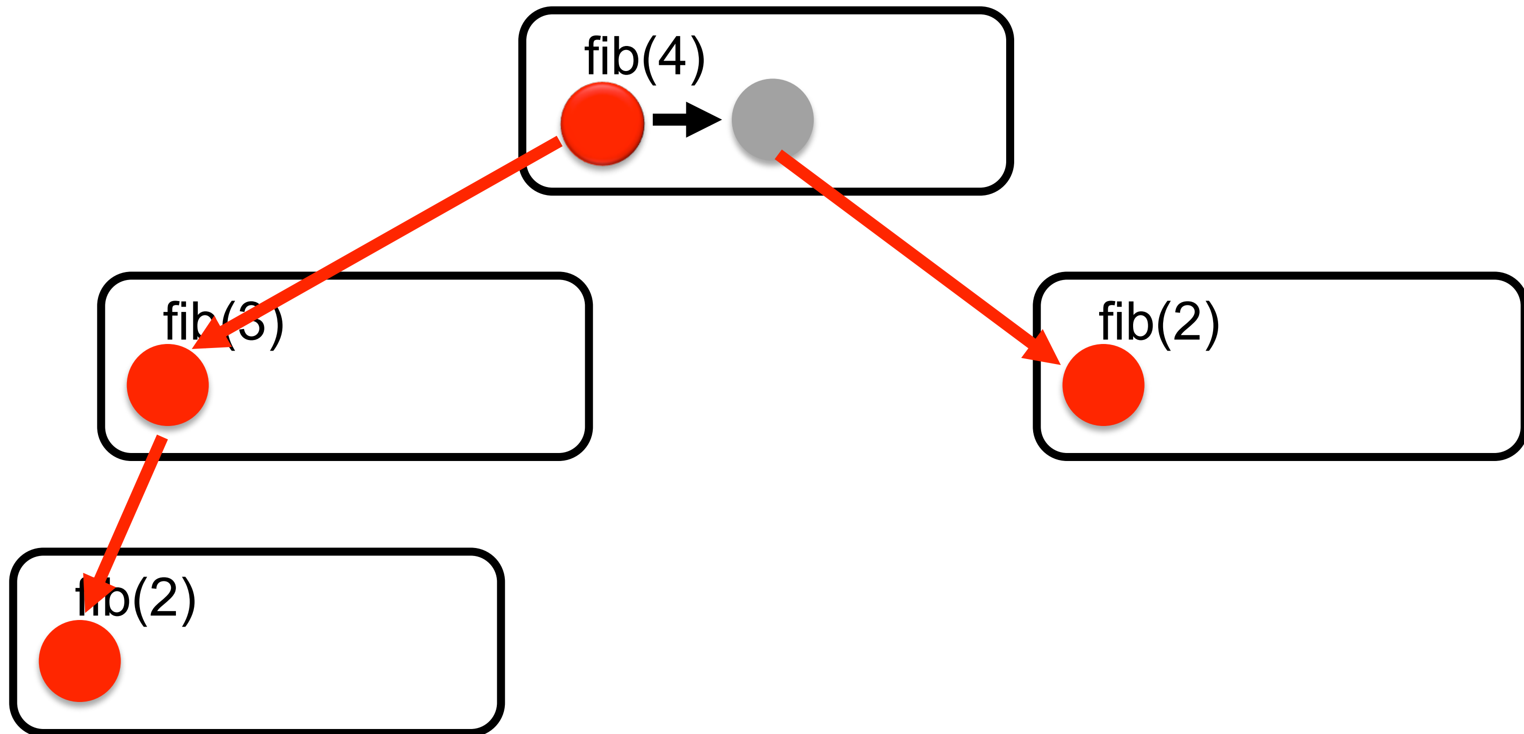
Fibonacci DAG

fib(4)

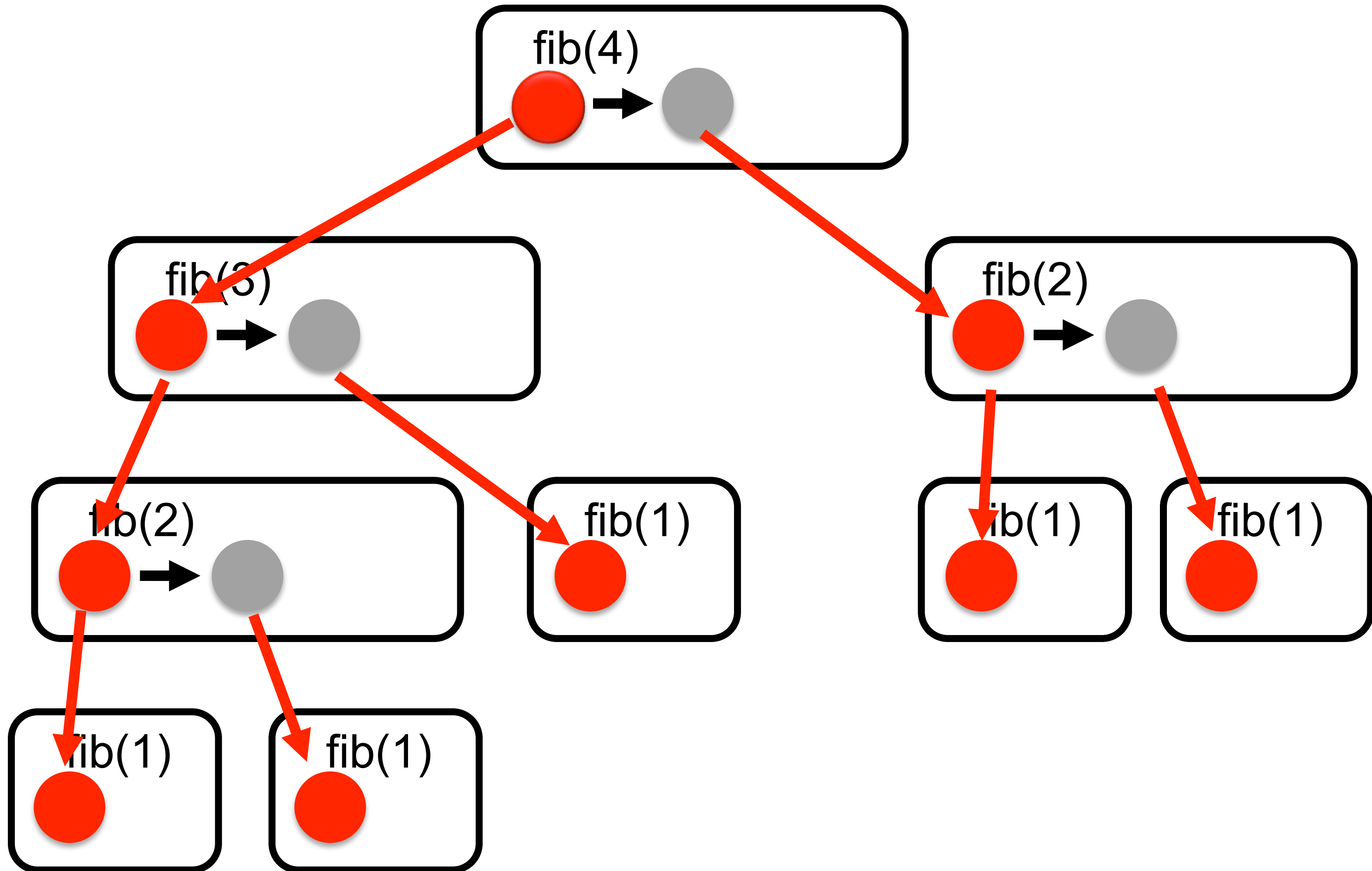
Fibonacci DAG



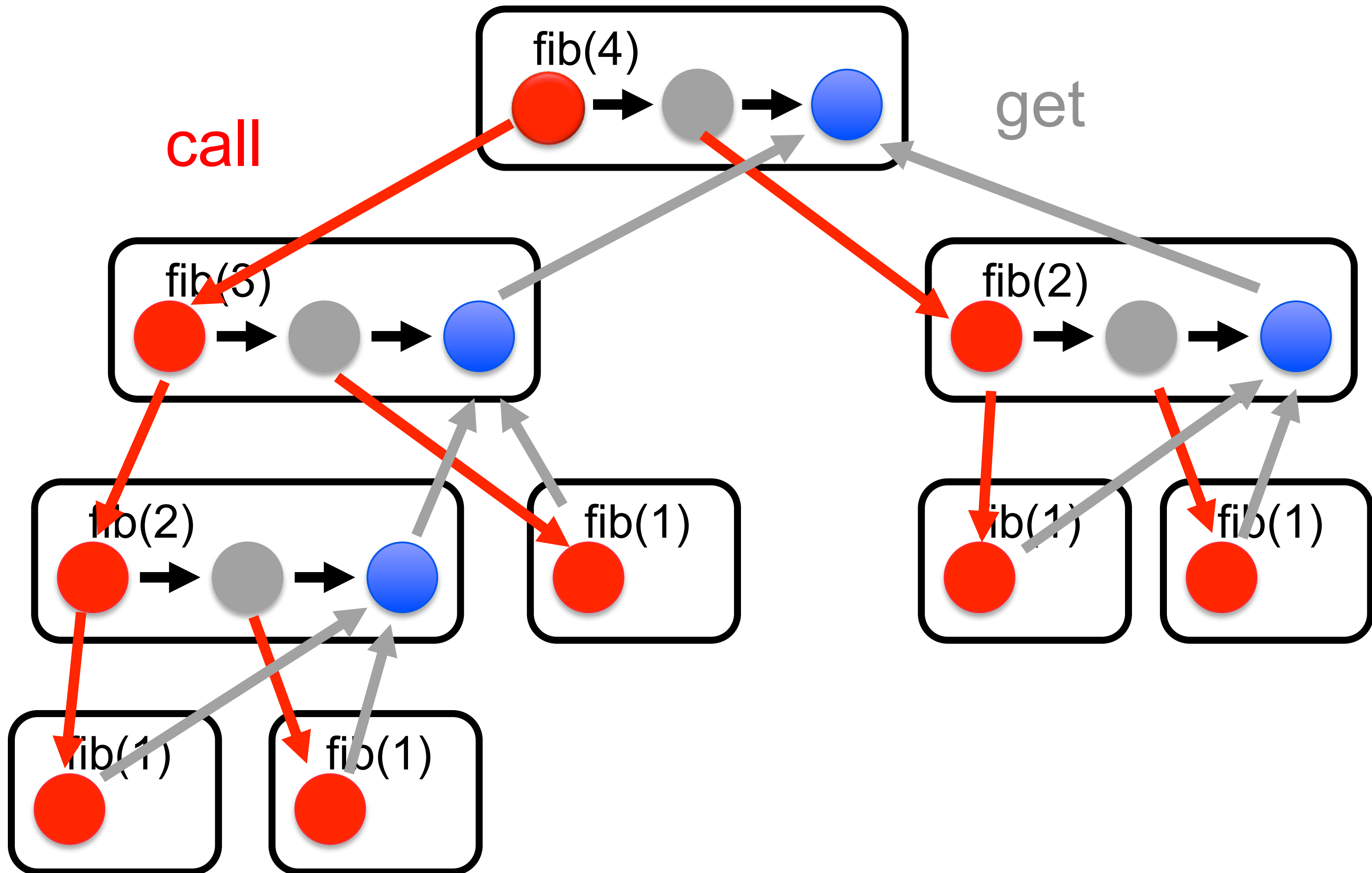
Fibonacci DAG



Fibonacci DAG



Fibonacci DAG



How Parallel is That?

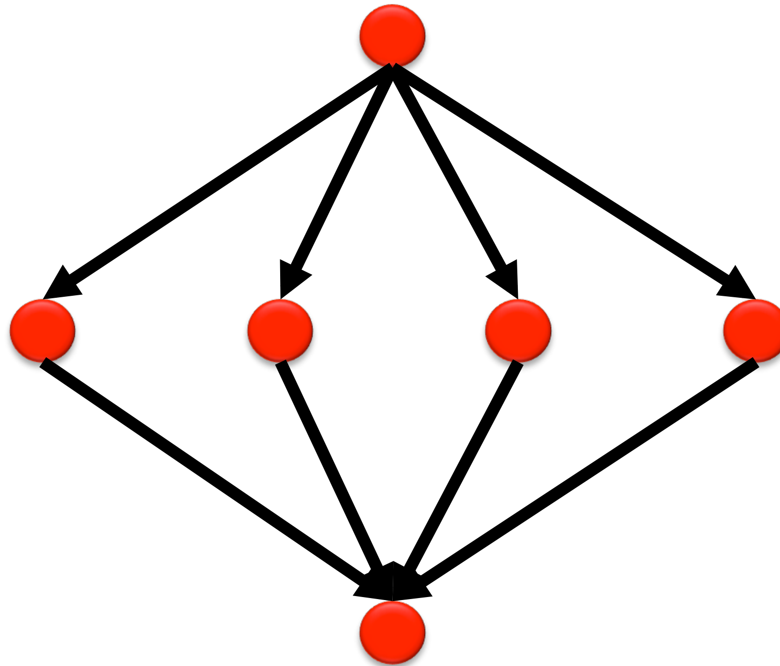
- Define work:
  - Total time on one processor
- Define critical-path length:
  - Longest dependency path
  - Can't beat that!

Notation Watch

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

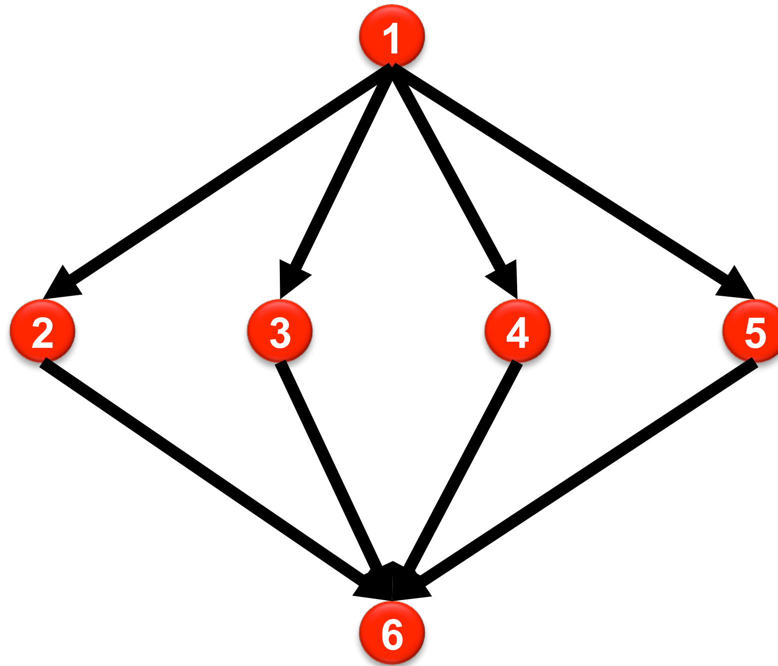
Unfolded DAG

- With dim = 2



Work?

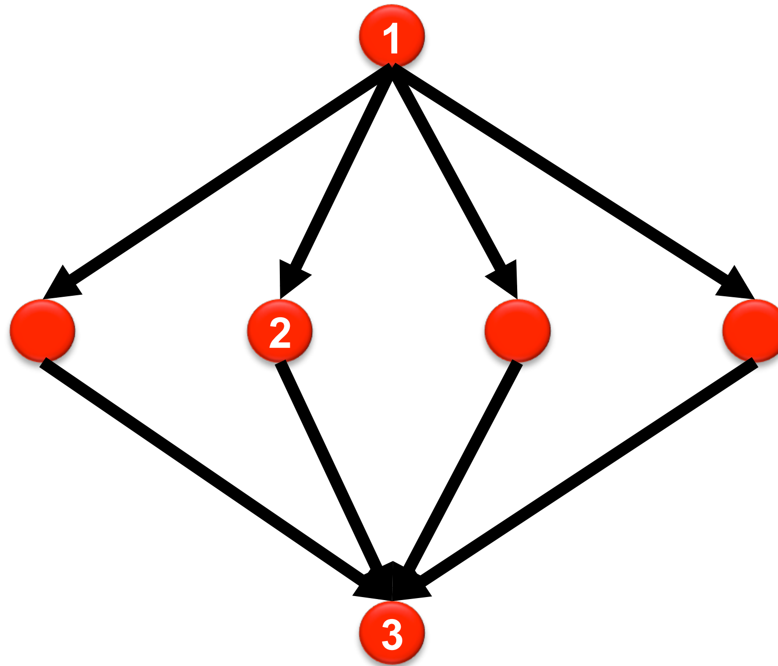
- With dim = 2



$T_1=6$

Critical path?

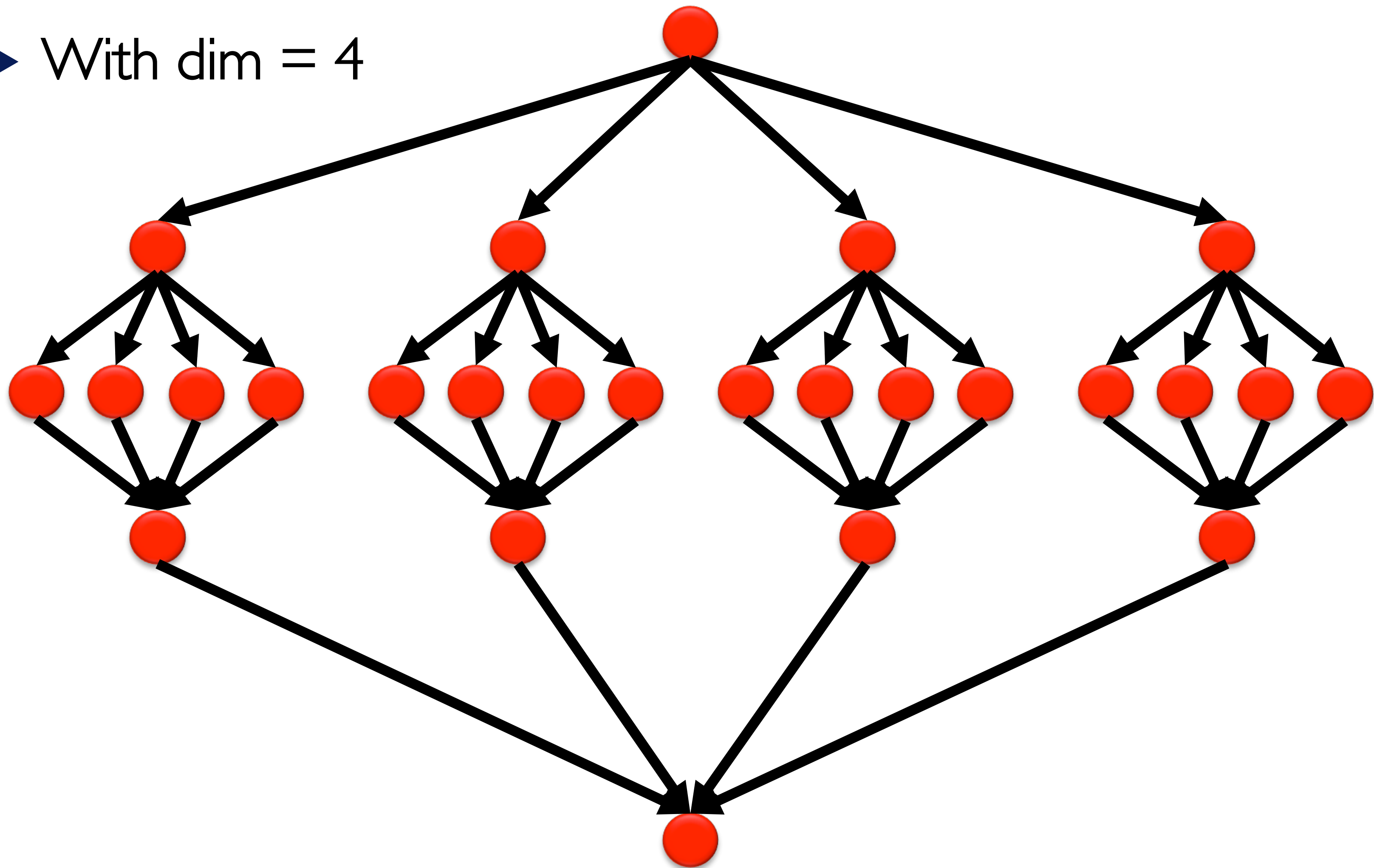
- With dim = 2



$T_\infty = 3$

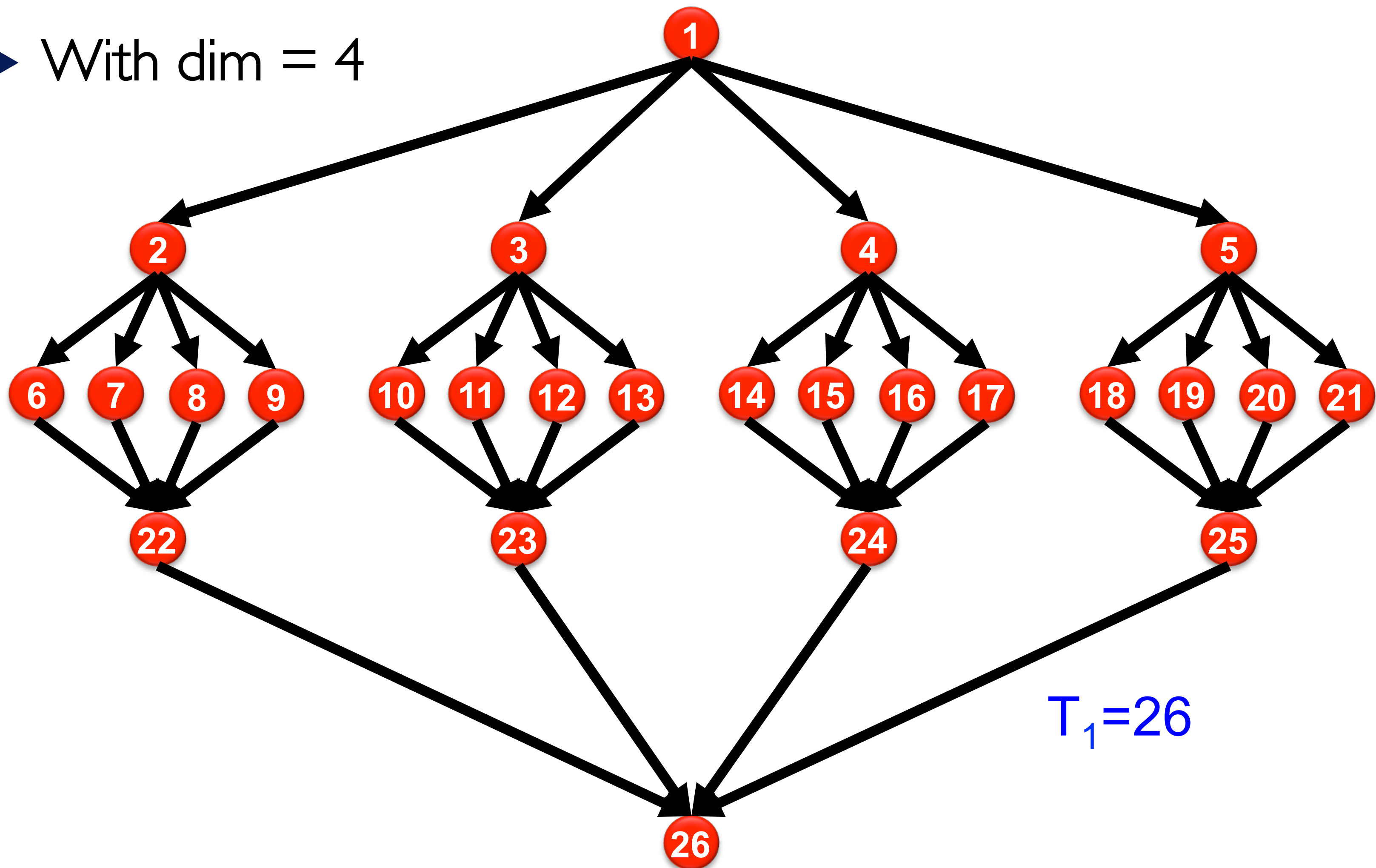
Unfolded DAG

- With dim = 4



Work?

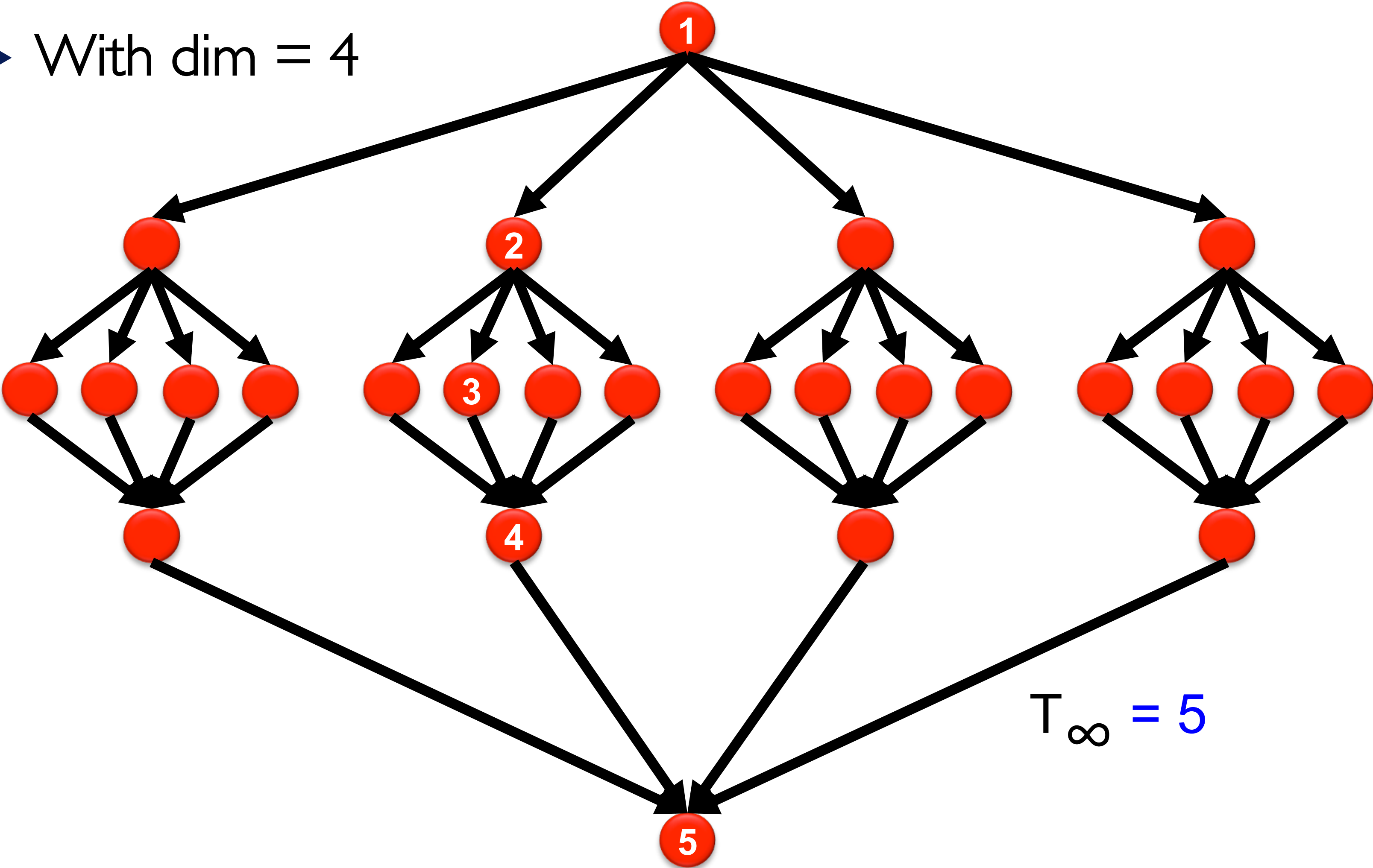
- With dim = 4



$T_1 = 26$

Critical path?

- With dim = 4



Question

- Assume dimension is a power of 2
- For dim=2^n, what are $\mathbf{T_1}$ and $\mathbf{T_\infty}$?
- For T_1 upper half

  $1 + 4 + \ldots + 2^{2n}$

  + lower half

  $$1 + 4 + \ldots + 2^{2(n-1)} = 2\sum_{i=0}^{n-1} 2^{2i} + 2^{2n}$$

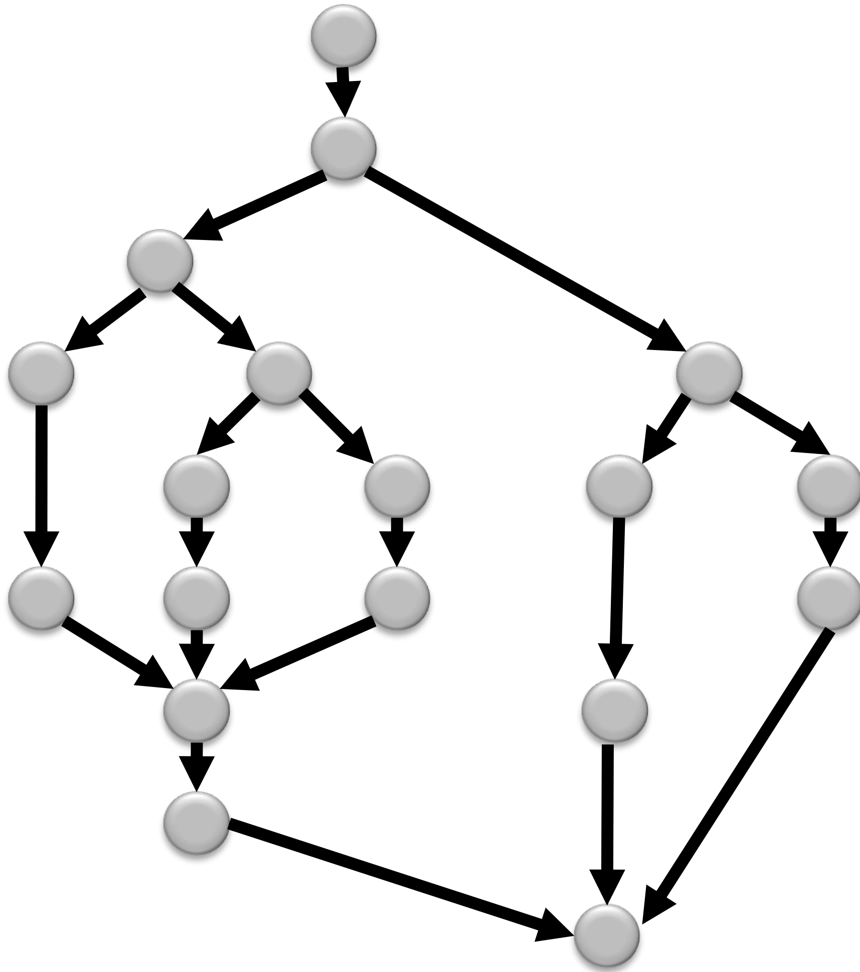
- $\mathbf{T_\infty} = 2n+1$

Question

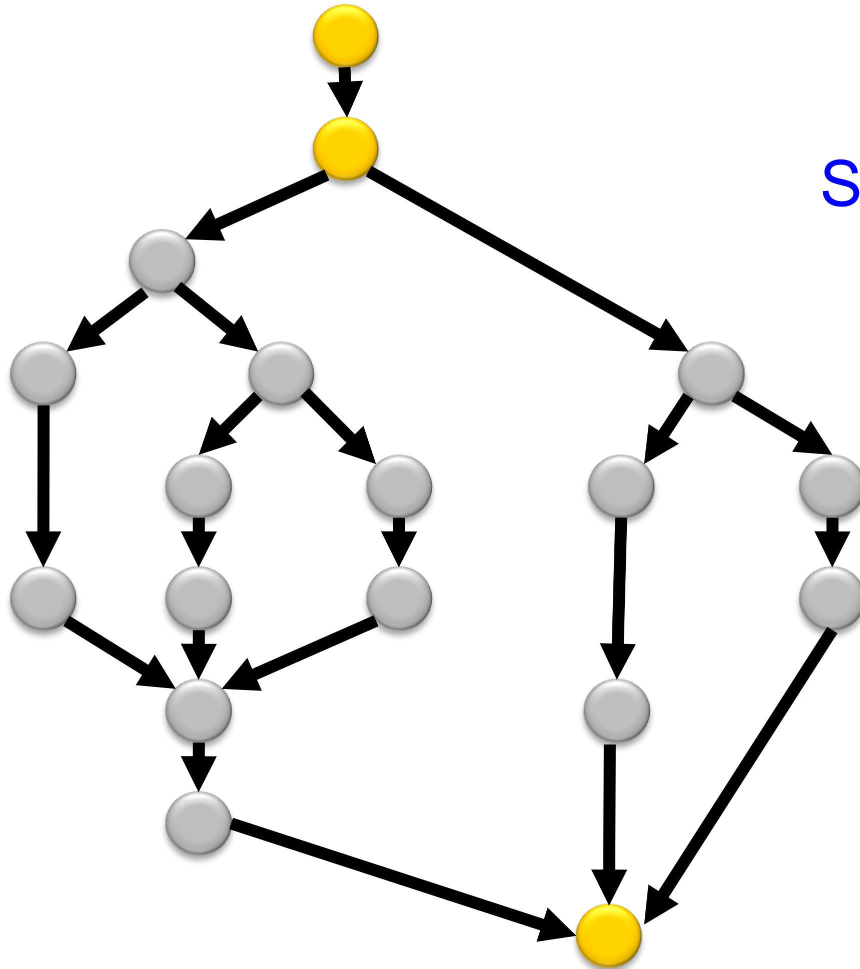
- Assume dimension is a power of 2
- For dim=2^n, what are $\mathbf{T_1}$ and $\mathbf{T_\infty}$?

- For T_∞ upper half and lower half are the logs + 1
  = n+1 + n = 2n + 1

Unfolded DAG



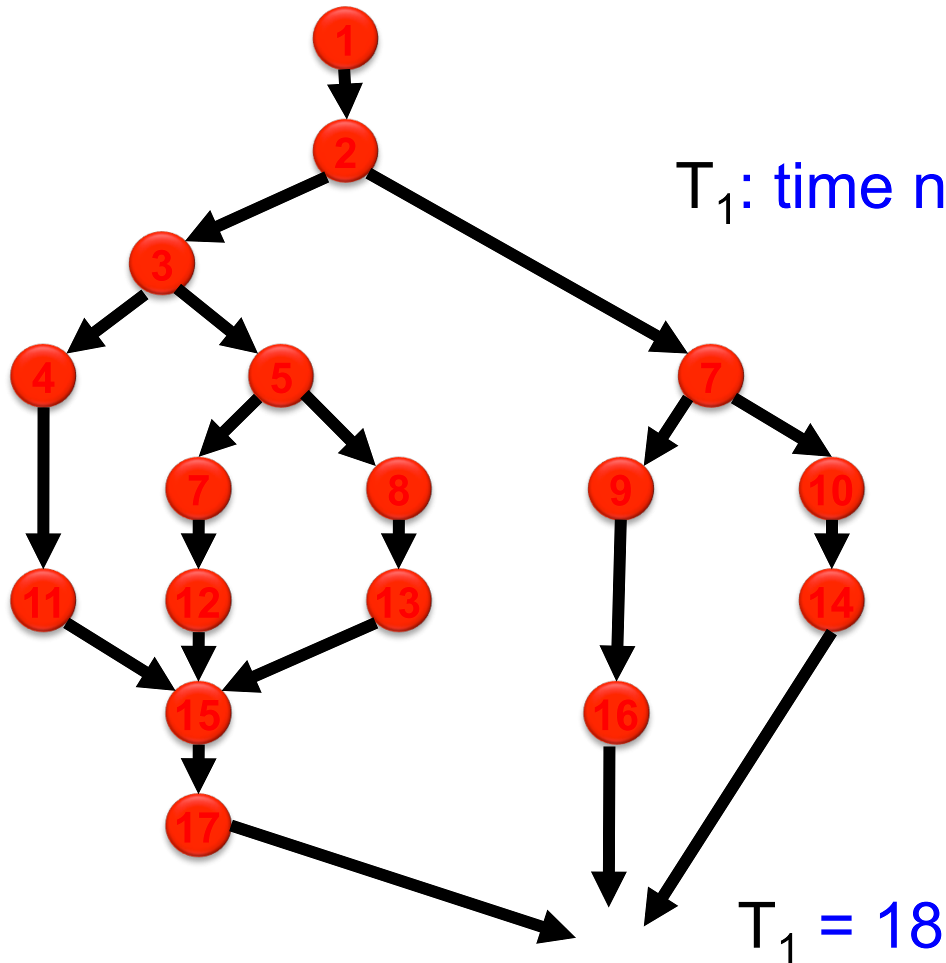
Parallelism?



Serial fraction = 3/18 = 1/6 …

Amdahl's Law says
speedup cannot exceed 6.

Work?

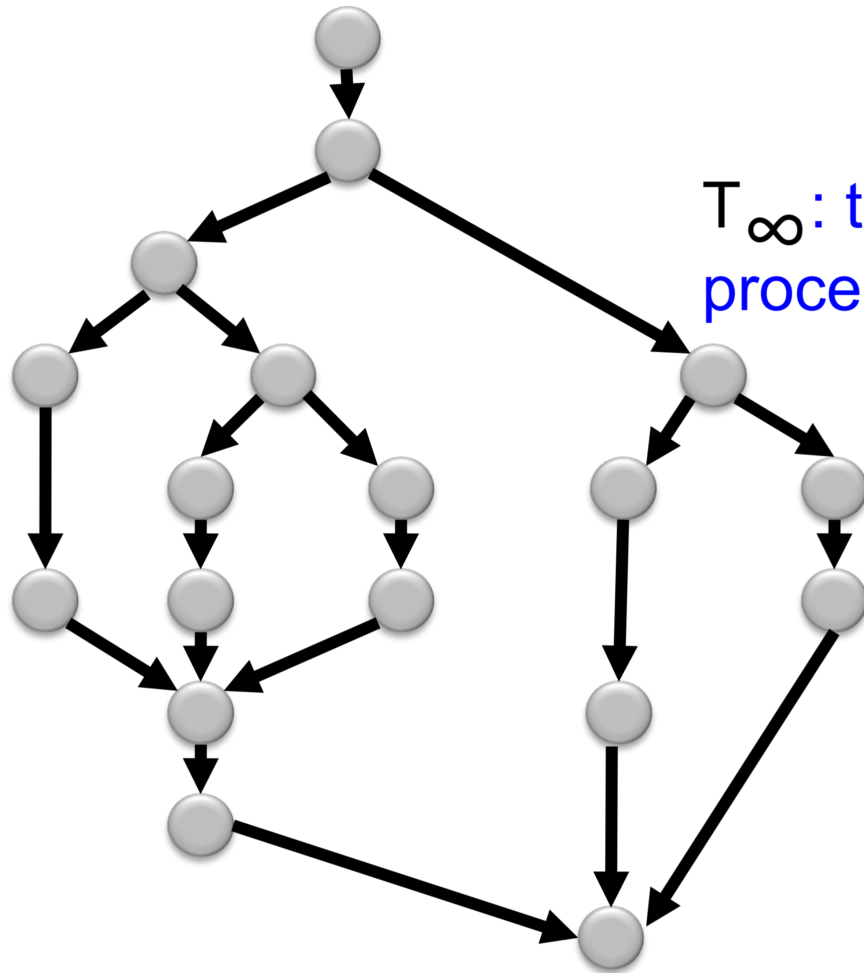


T_1: time needed on one processor

Just count the nodes ….

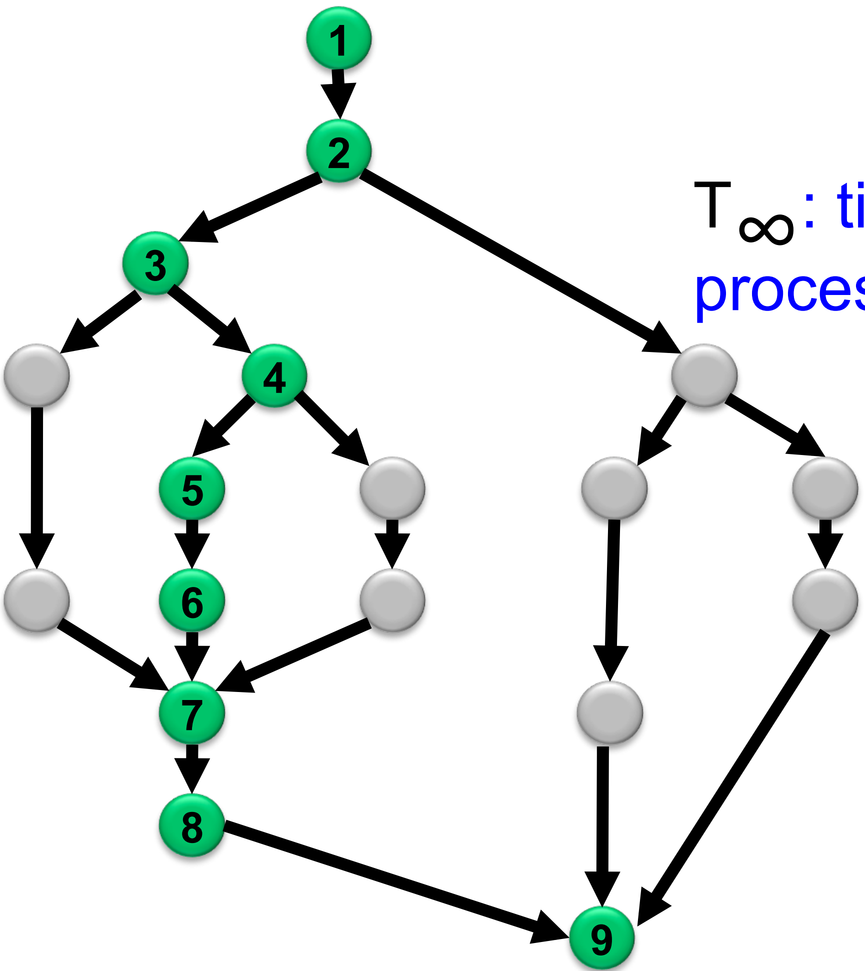
$T_1 = 18$

Critical Path?



T_∞: time needed on as many processors as you like

Critical Path?



T_∞: time needed on as many processors as you like

Longest path ….

$T_\infty = 9$

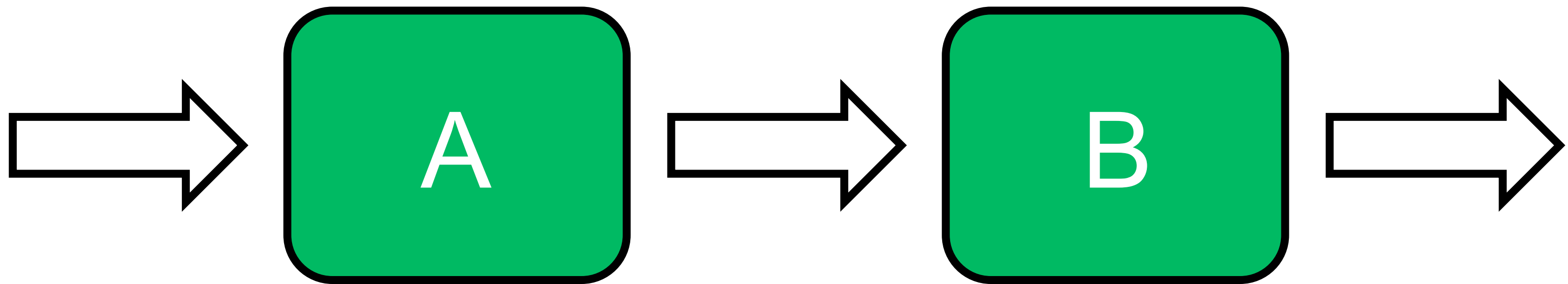
Simple Laws

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

Performance Measures

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

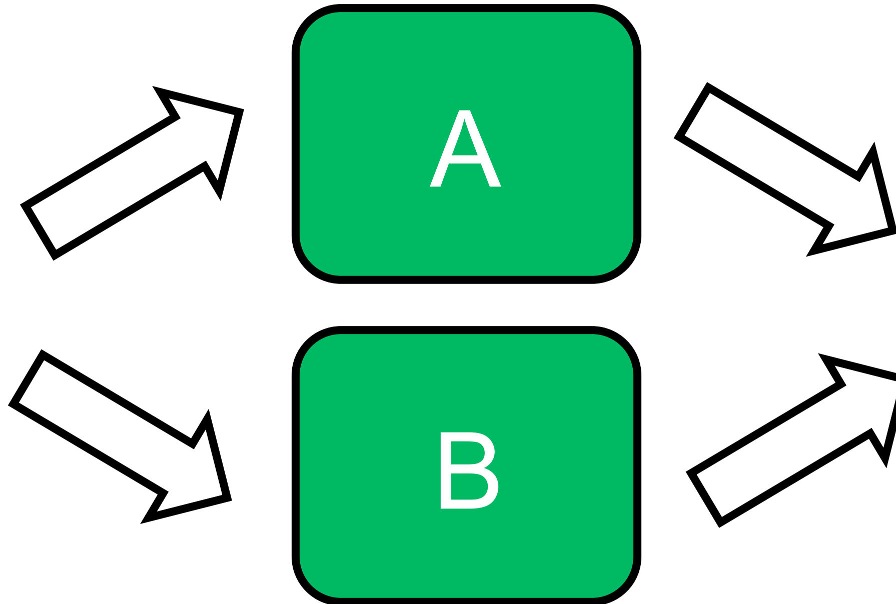
Sequential Composition



Work: $T_1(A) + T_1(B)$

Critical Path: $T_\infty(A) + T_\infty(B)$

Parallel Composition



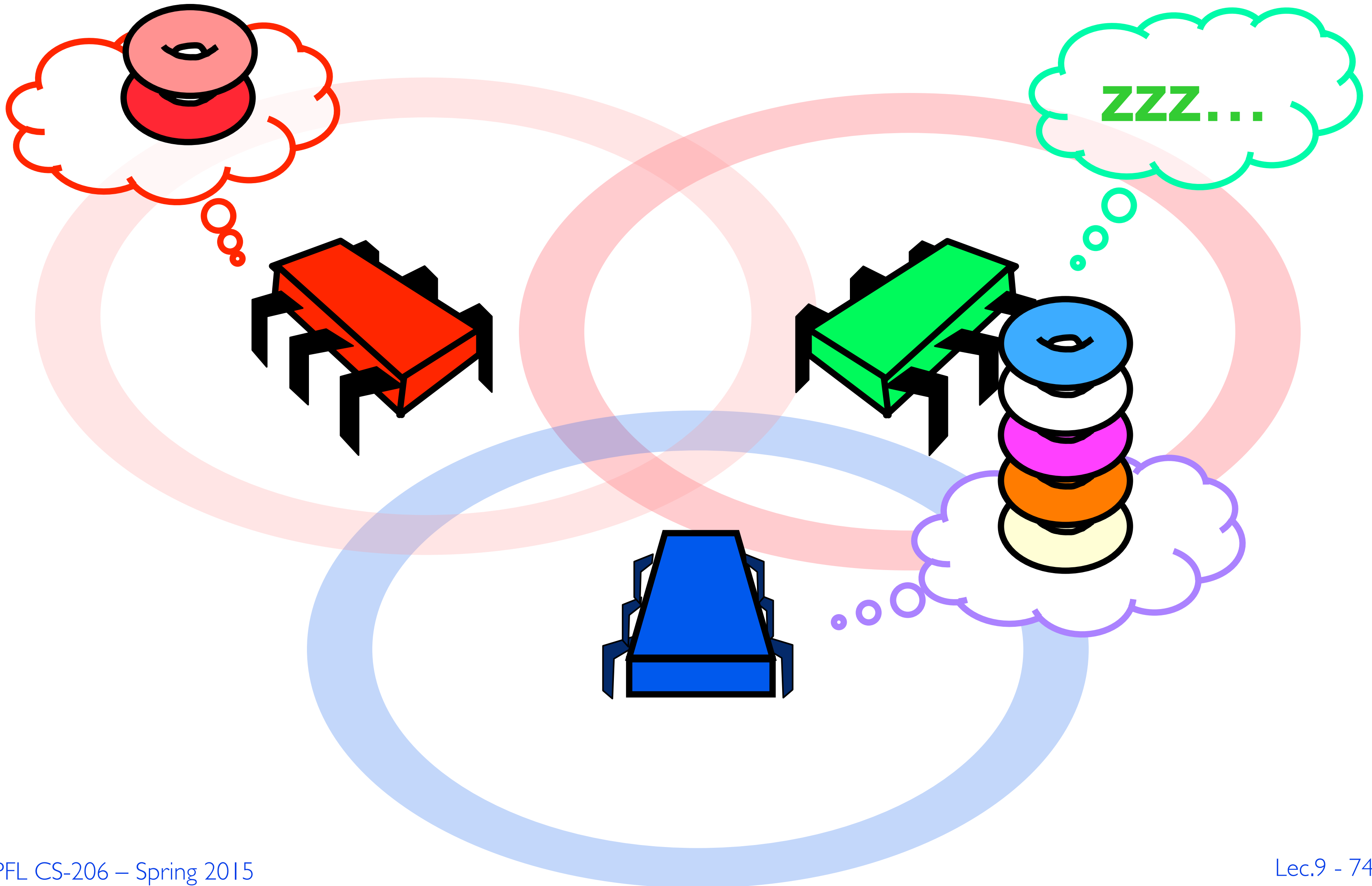
Work: $T_1(A) + T_1(B)$

Critical Path: $\max\{T_\infty(A), T_\infty(B)\}$

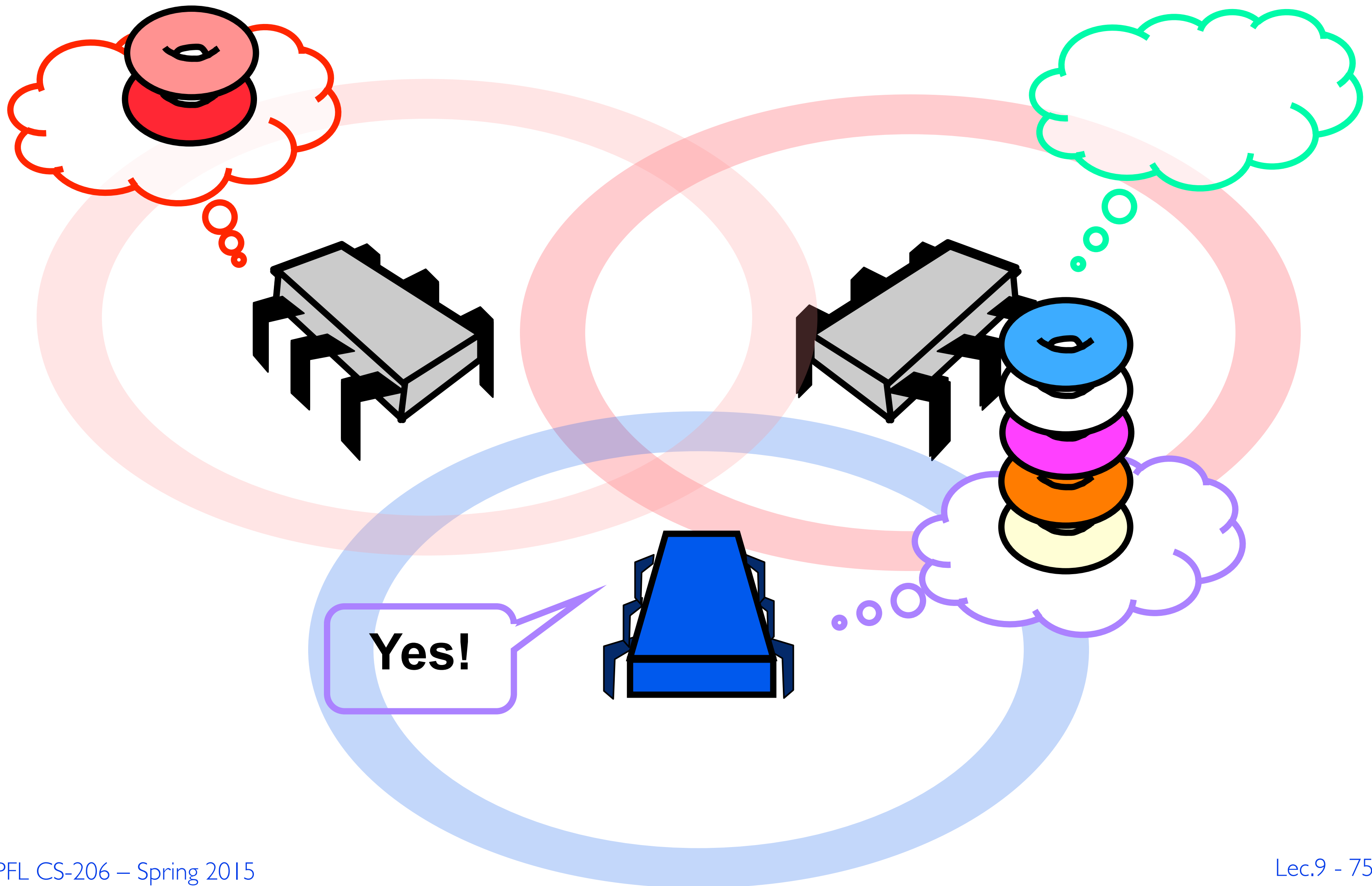
Back to Matrix Addition

- How much work?
- What is the critical path?

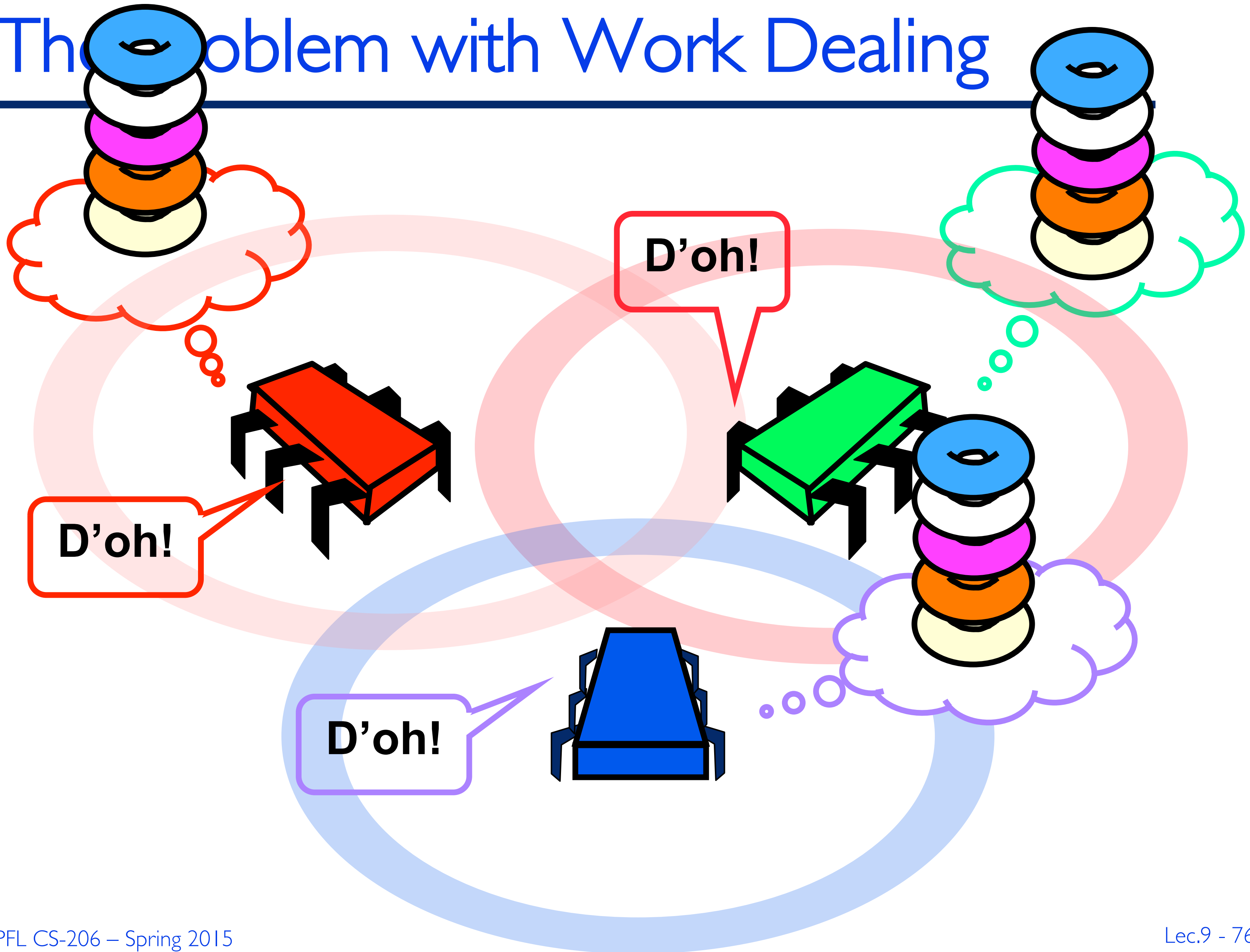
Work Distribution



Work Dealing



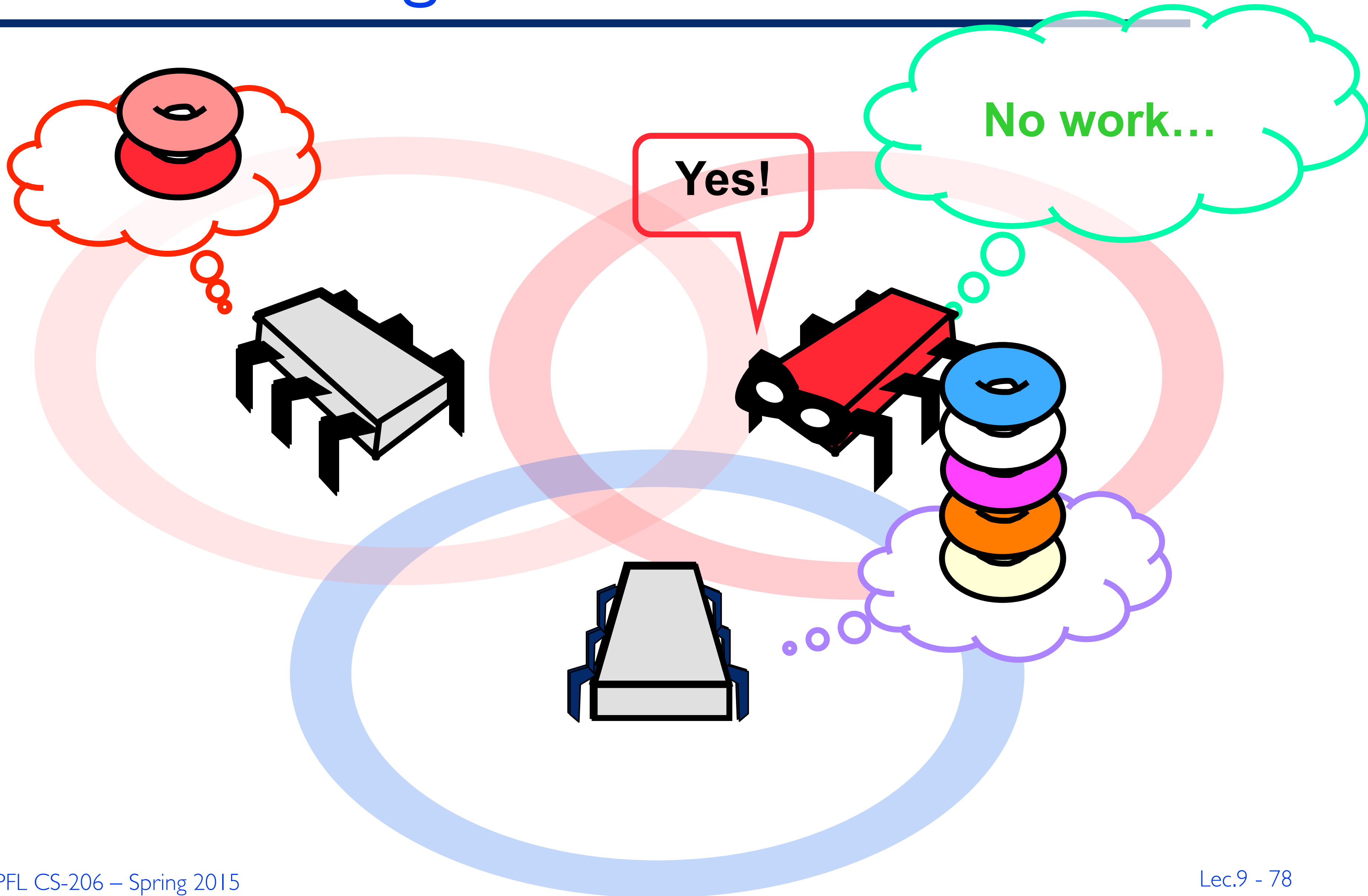
The Problem with Work Dealing



Work Dealing is fundamentally flawed!

- Work dealing
  - Everyone passes extra work to others
  - Spend time dealing rather than actually working
  - There is contention in work distribution

Work Stealing

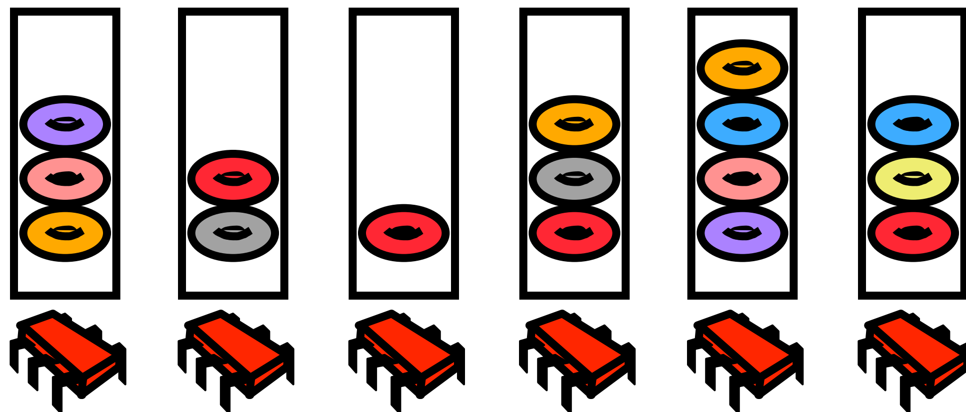


Lock-Free Work Stealing

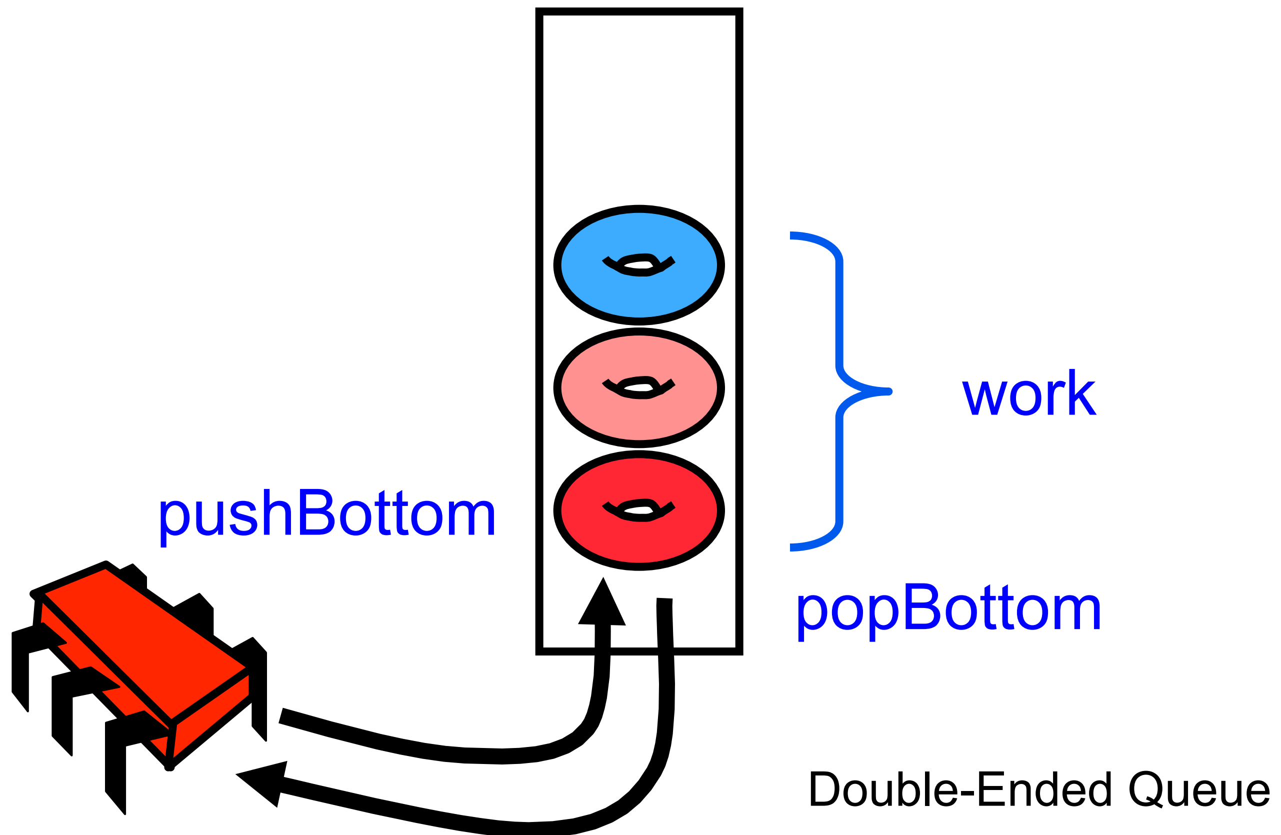
- Each thread has a pool of ready work
- Remove work without synchronizing
- If you run out of work, steal someone else's
- Choose victim at random

Local Work Pools

Each work pool is a Double-Ended Queue

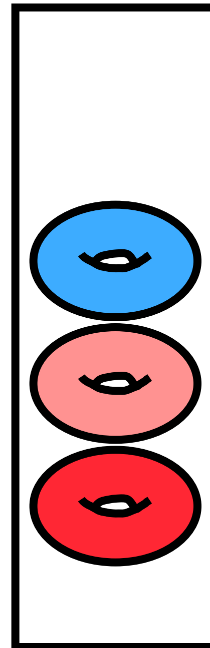
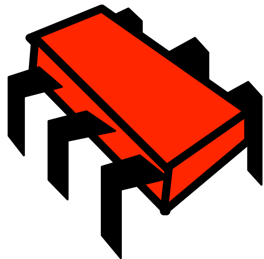


Work DEQueue[1]



Obtain Work

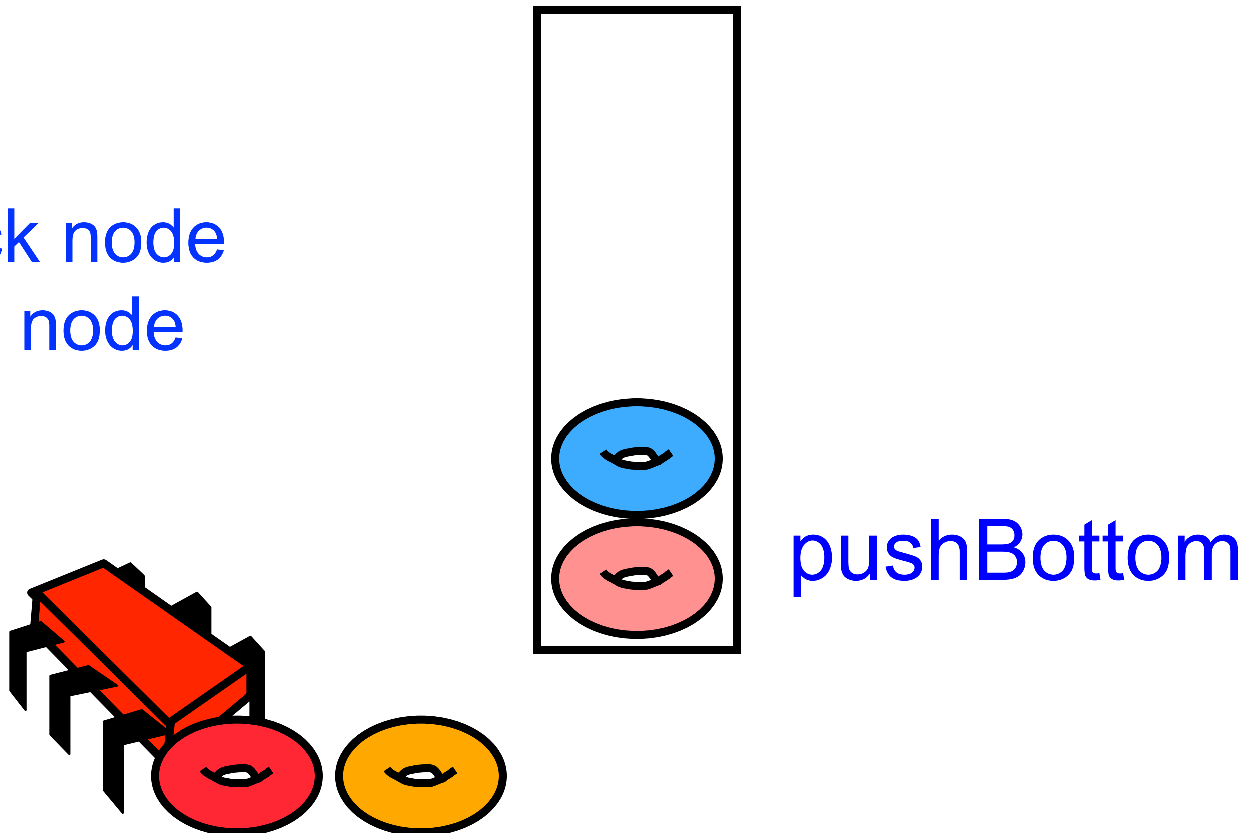
- Obtain work
- Run task until…
- …blocks or terminates

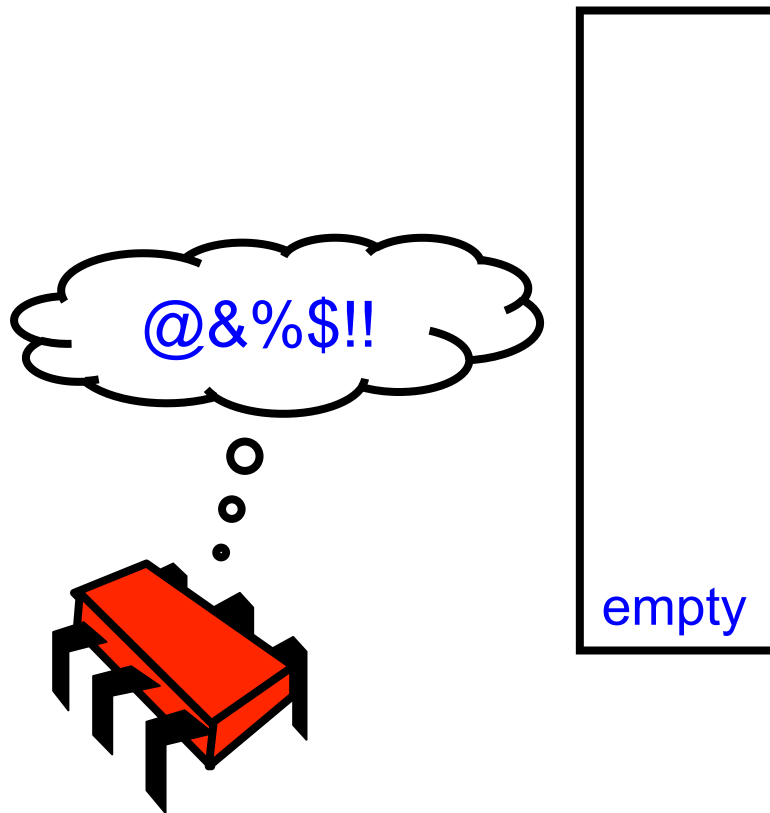
popBottom

New Work

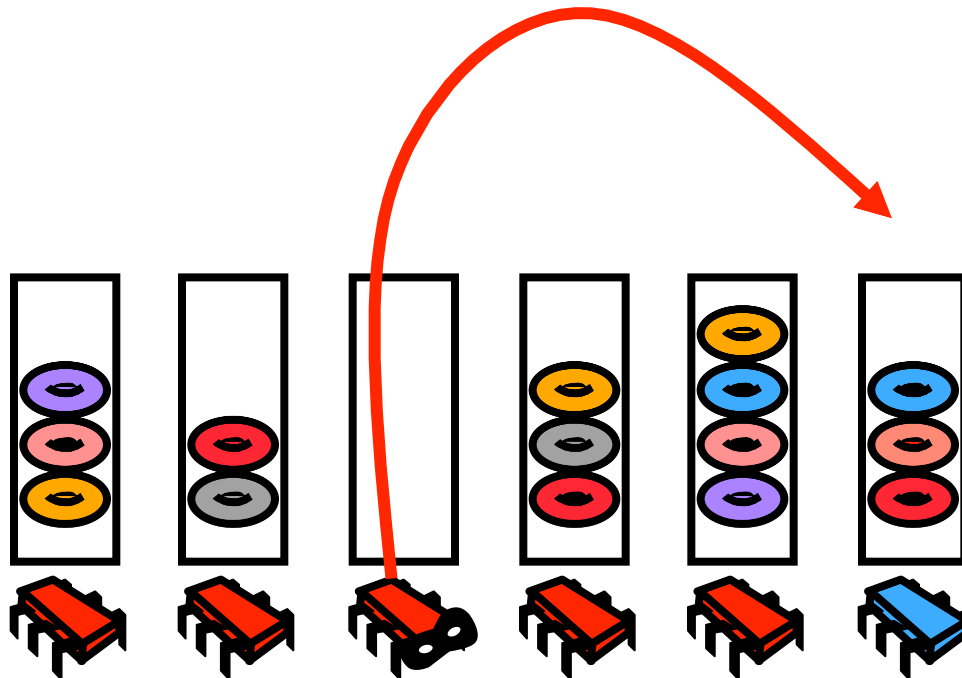
- Unblock node
- Spawn node



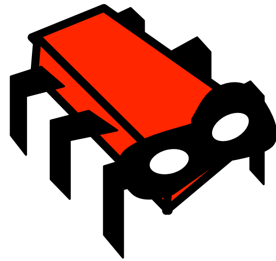
What to do When the Well Runs Dry?

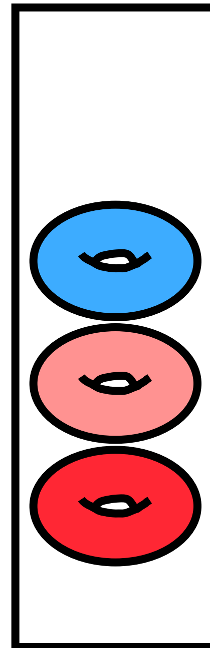


Steal Work from Others

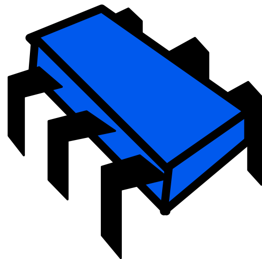


Steal this Task!





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 double-ended queue

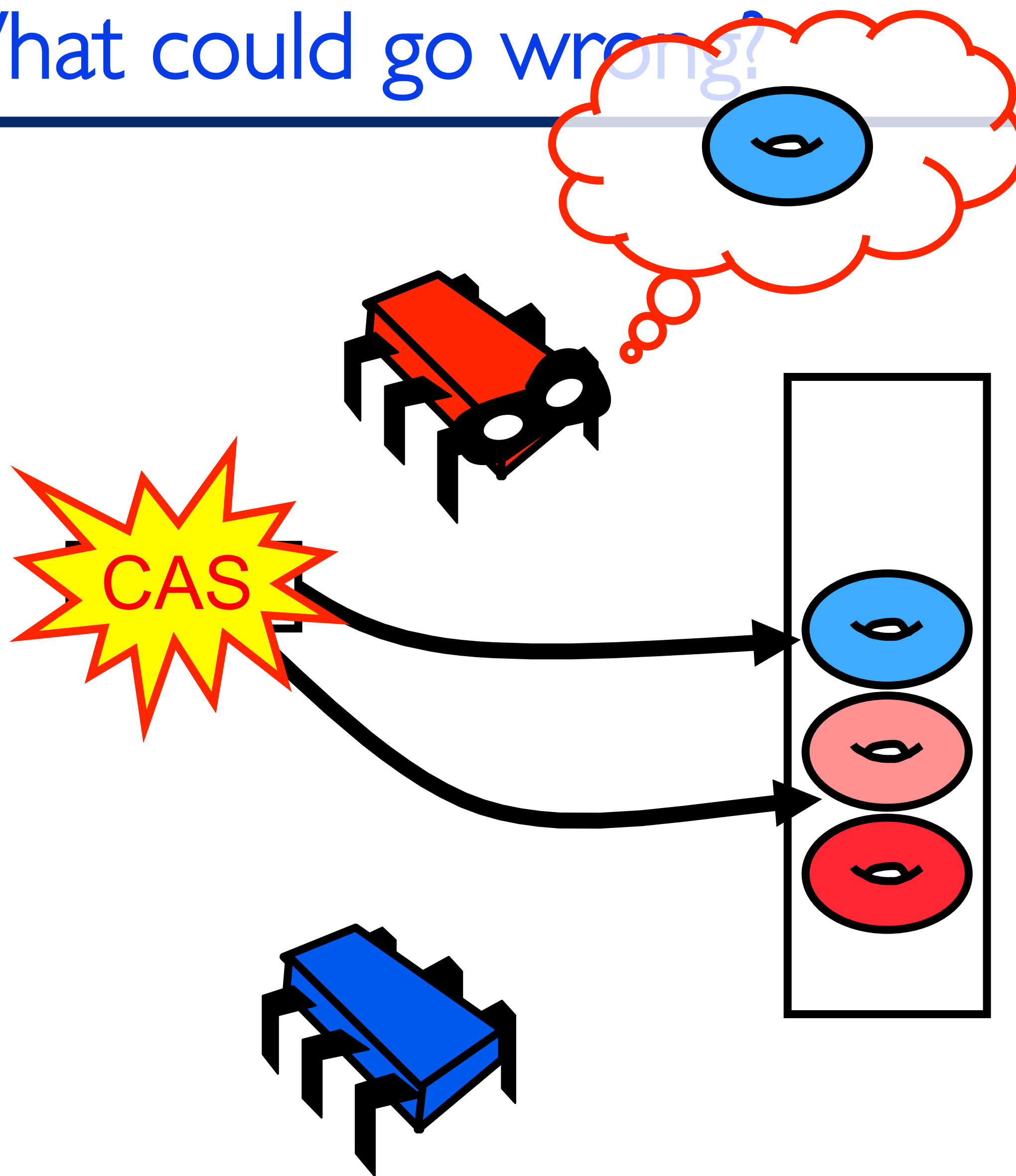
- Wait-Free
  - No locking or starvation
- Linearizable
  - All methods appear to execute instantaneously
  - Method execution can follow a total order
  - Can reason about correctness
- Constant time

But, can not always have the cake and eat it too!

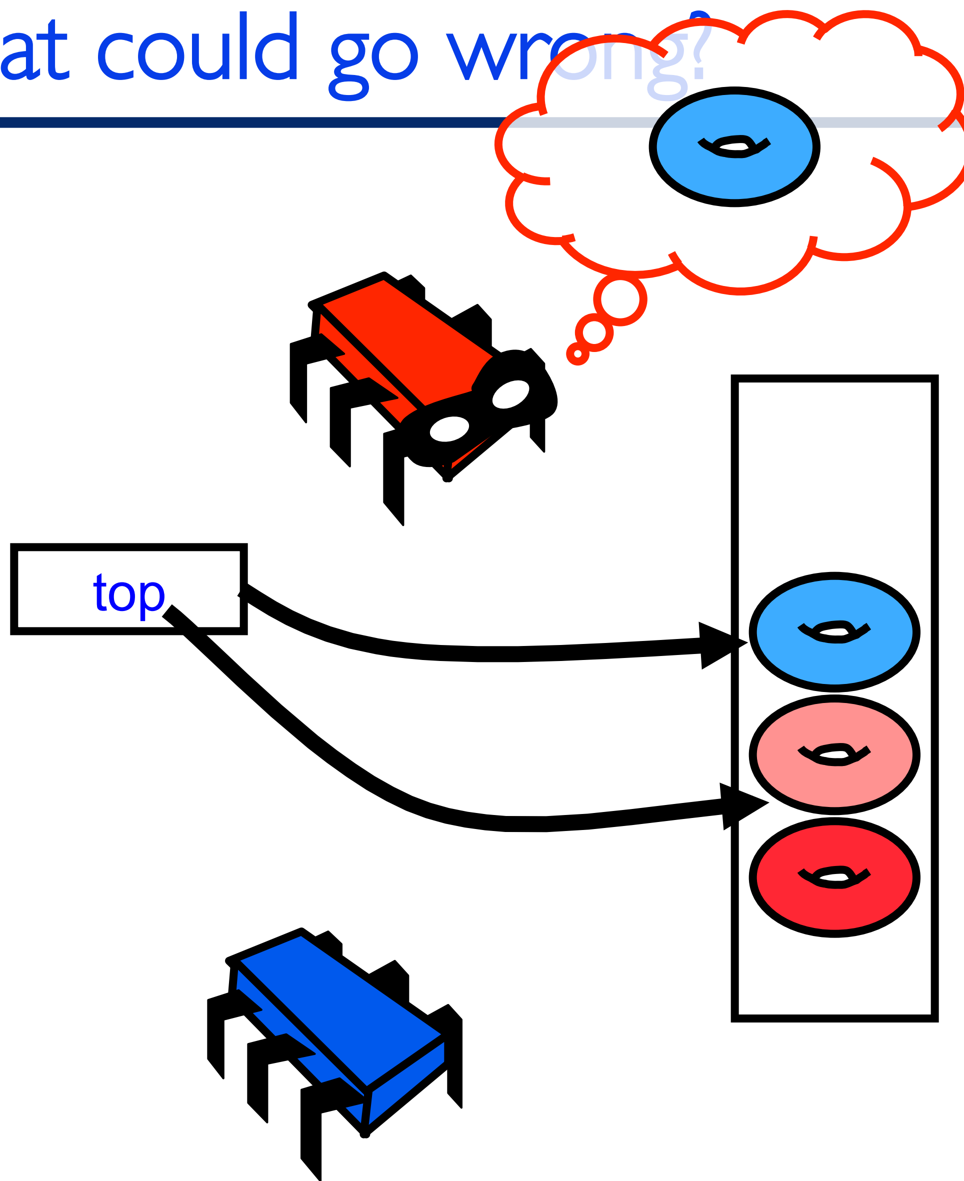
Compromise

- Method popTop may fail if
  - *popTop* succeeds, or a
  - *popBottom* (concurrent with *popTop*) takes last task

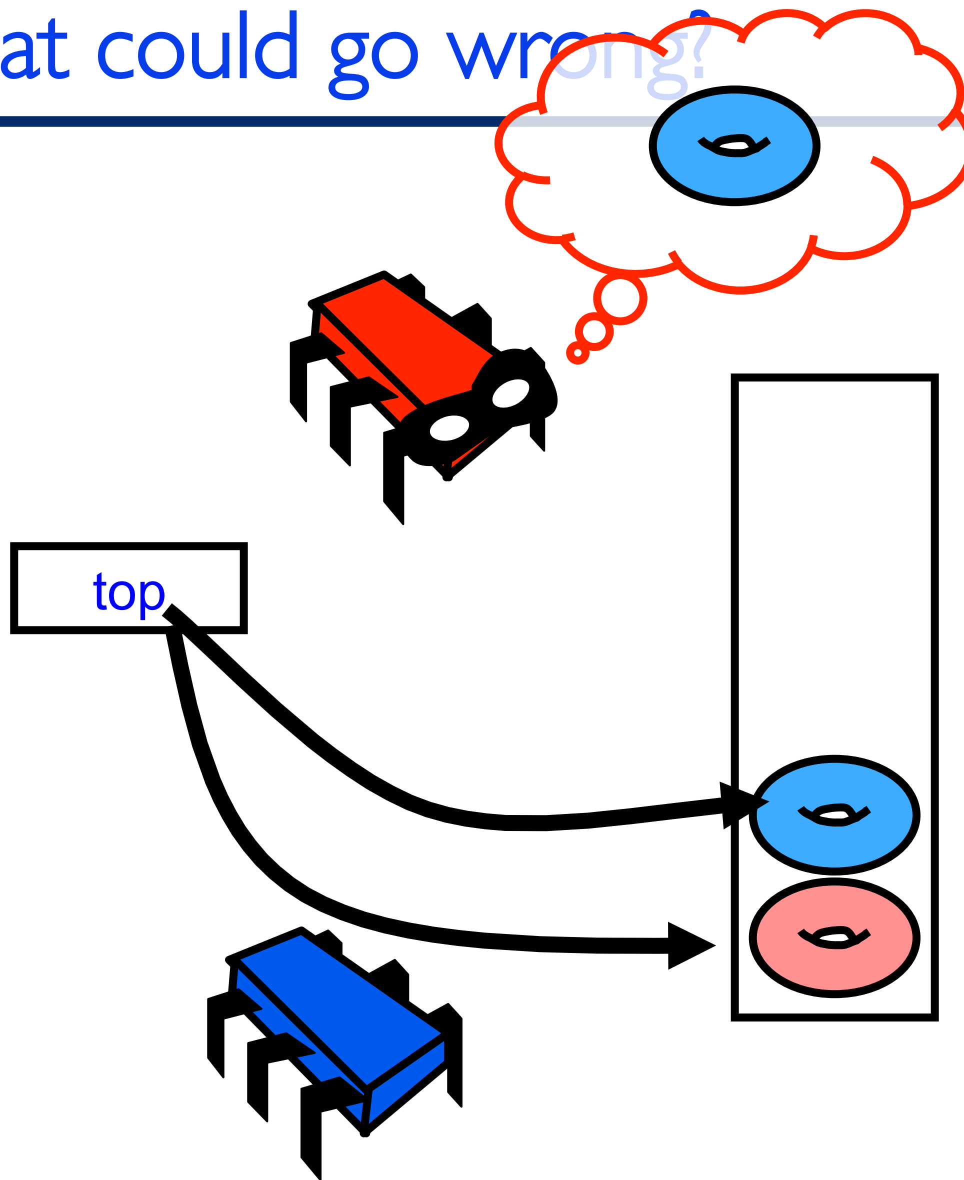
What could go wrong?



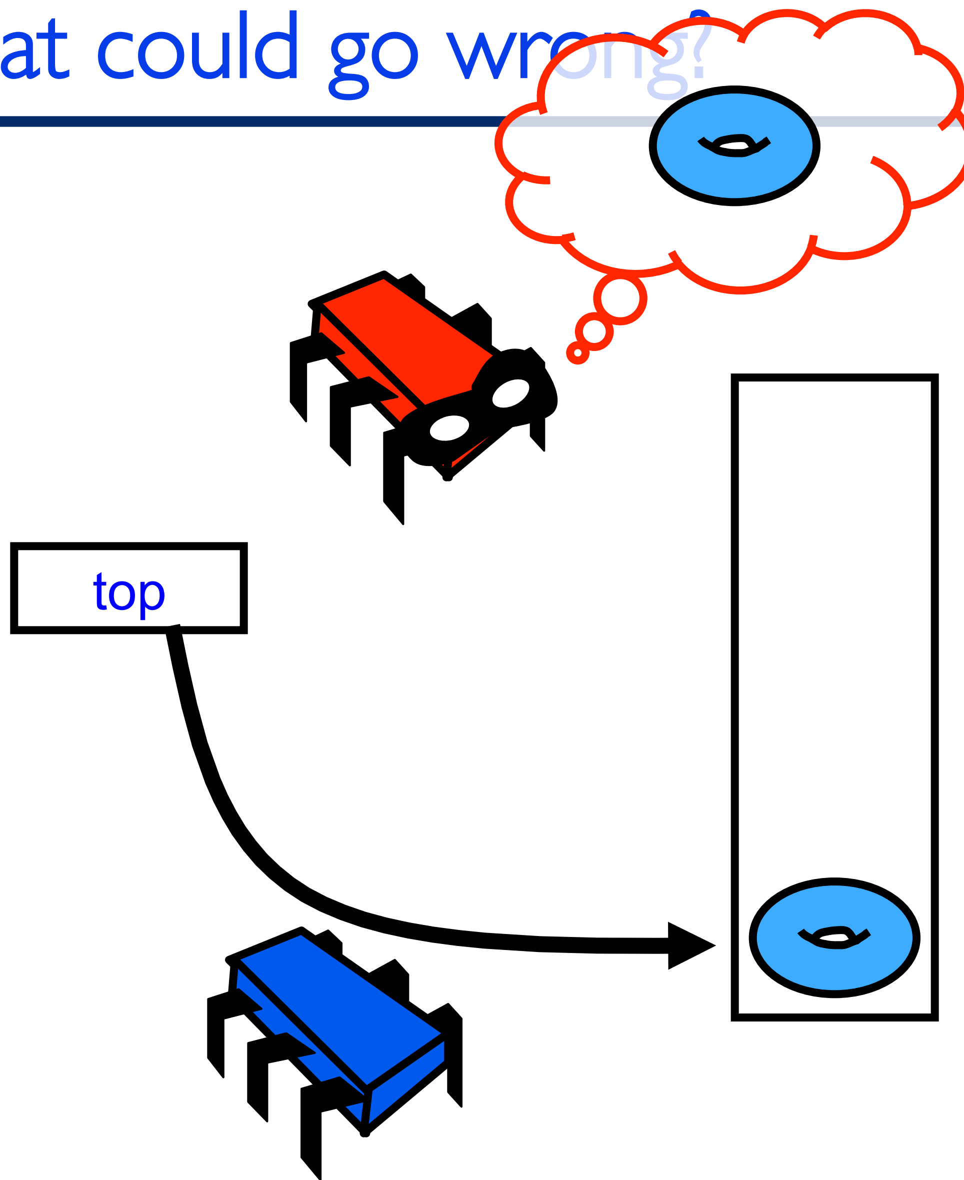
What could go wrong?



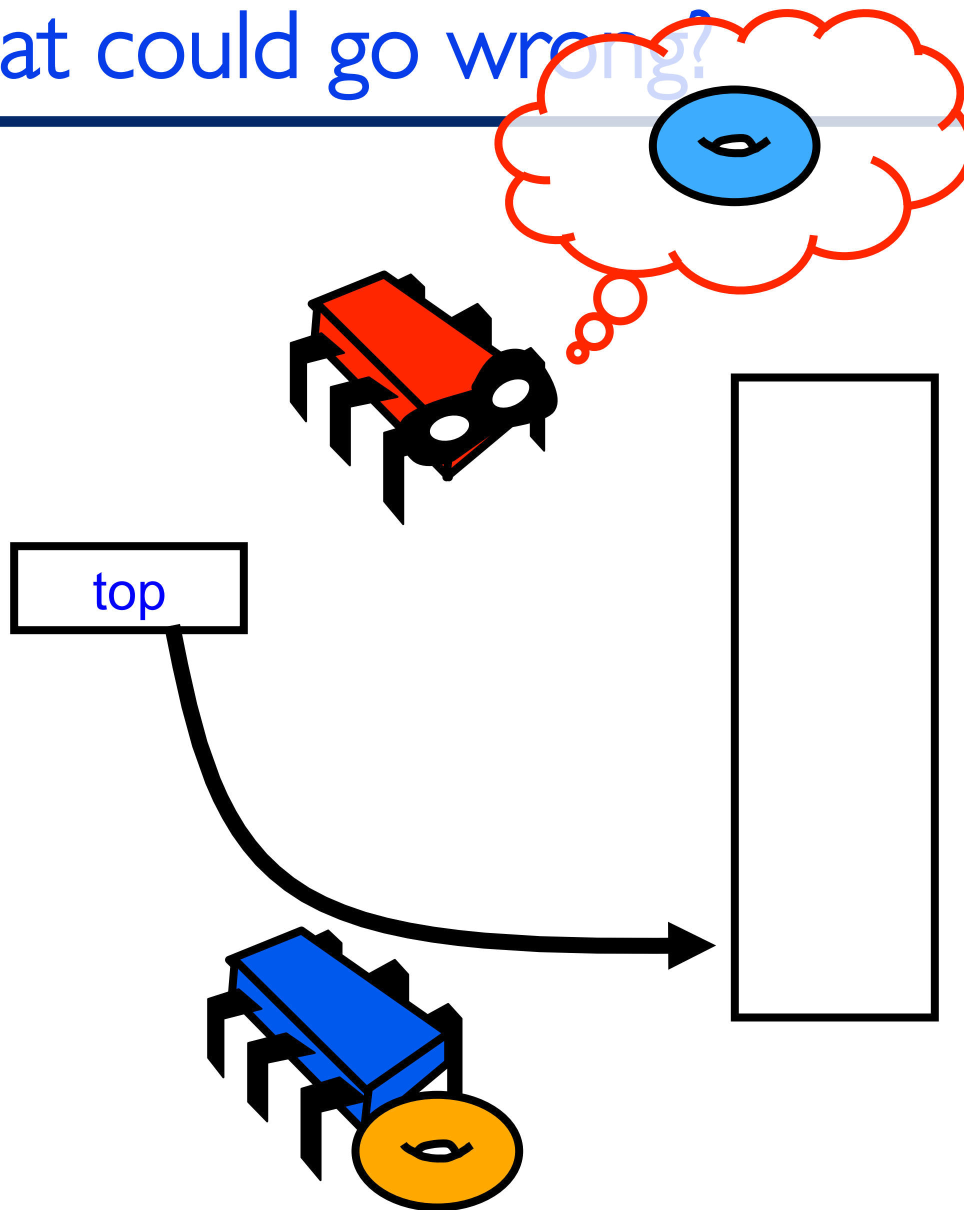
What could go wrong?



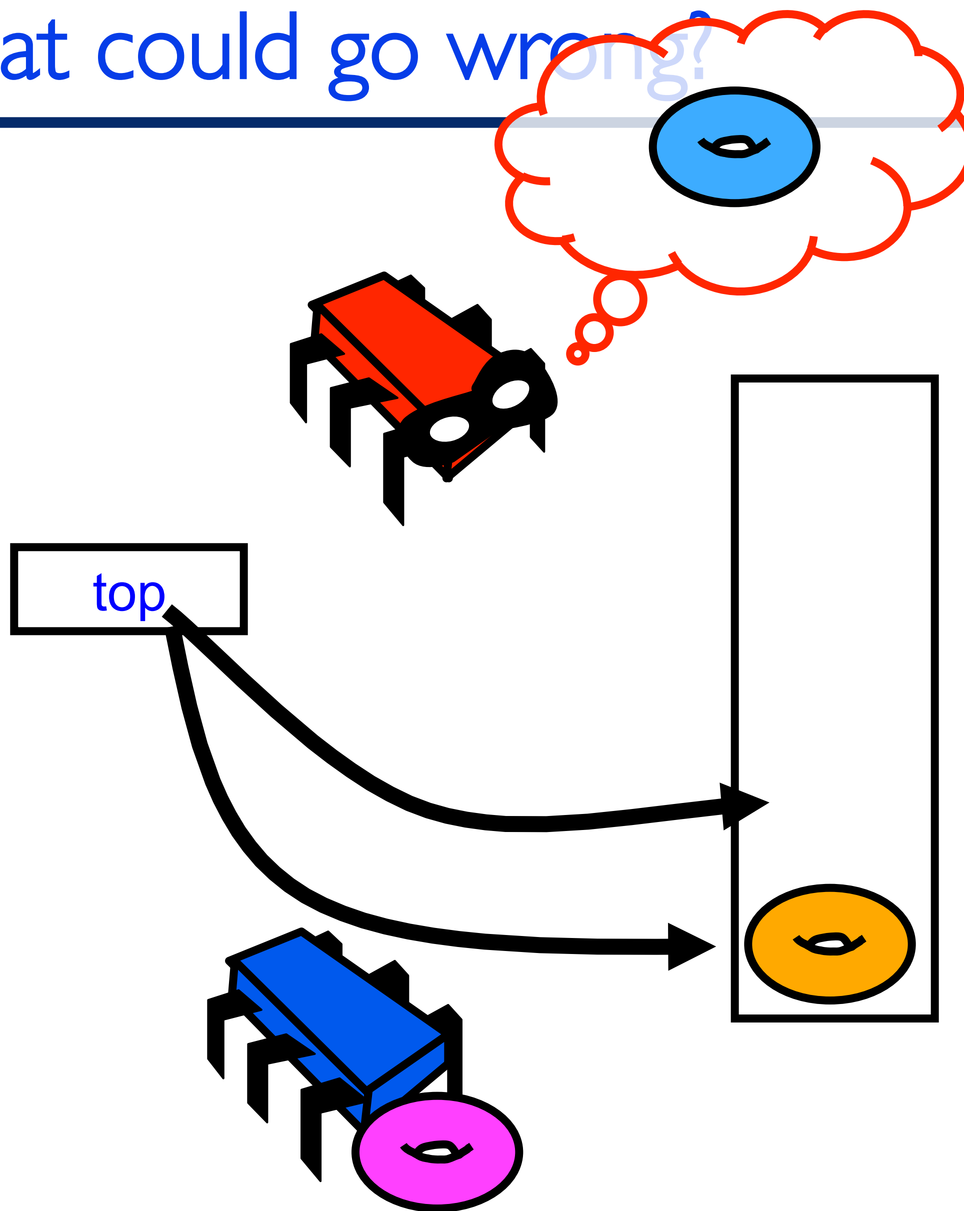
What could go wrong?



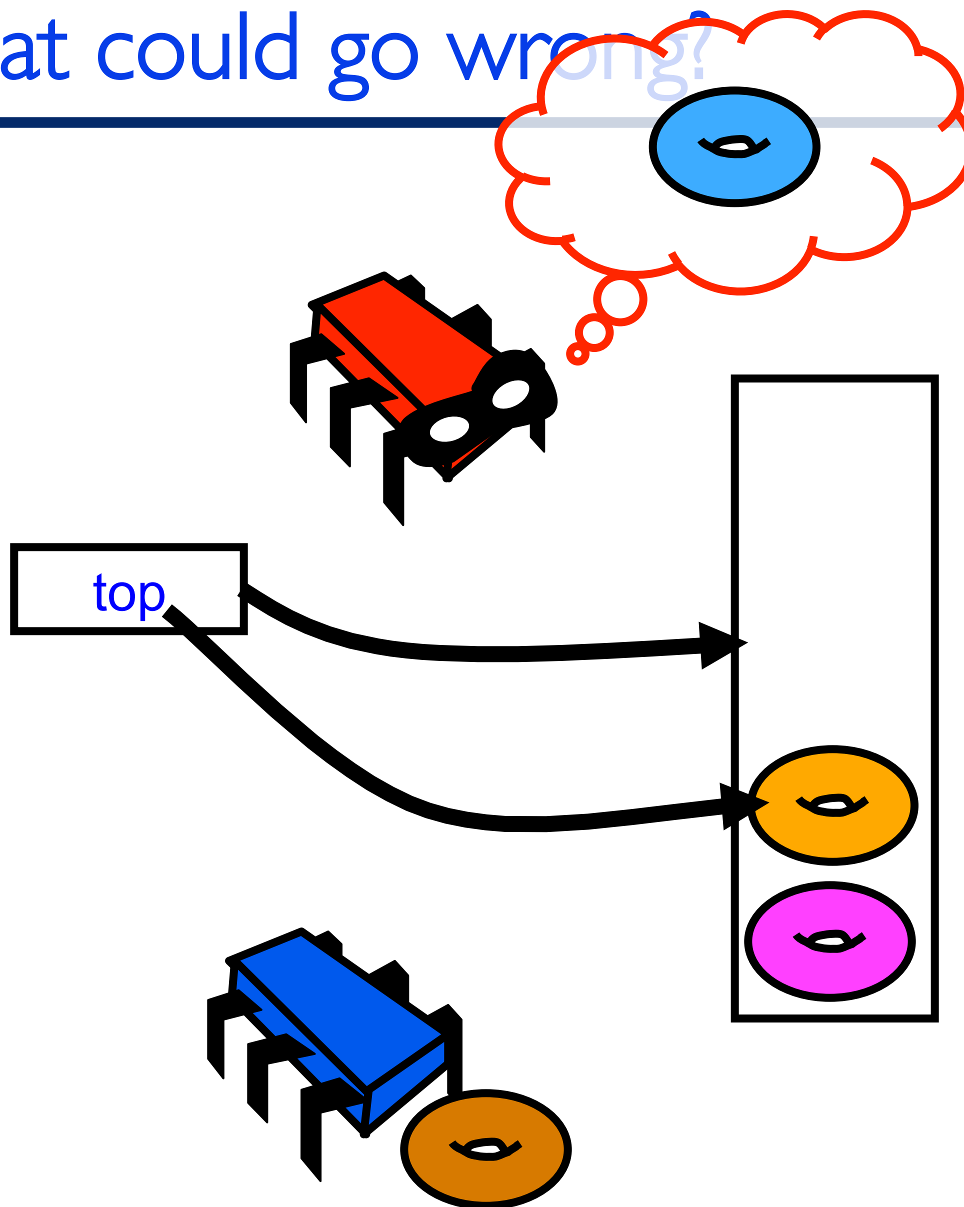
What could go wrong?



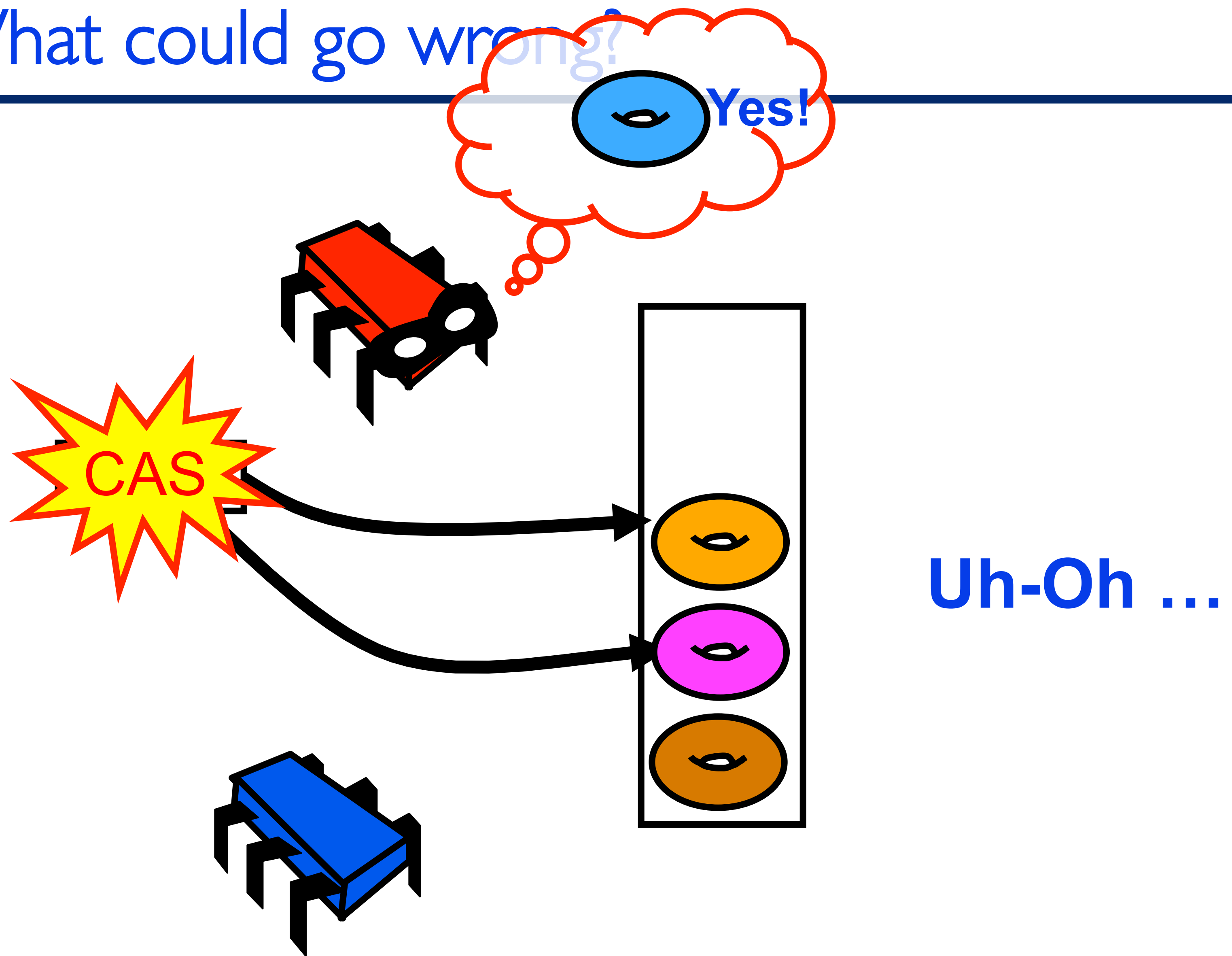
What could go wrong?



What could go wrong?

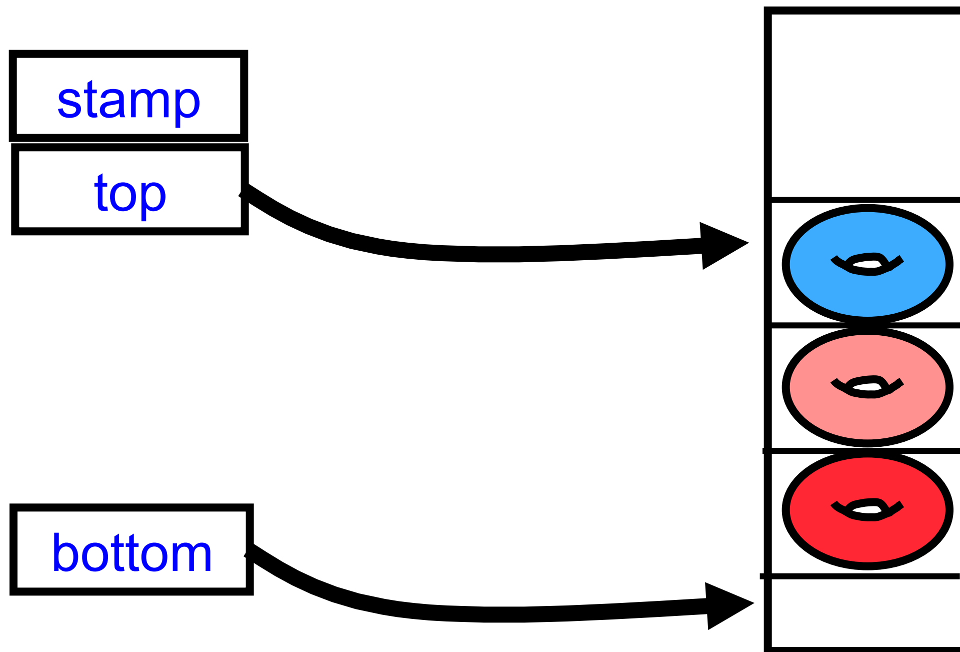


What could go wrong?



Uh-Oh …

Fix



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[] tasks;
 …
}
```

index of bottom task
no need to synchronize
but must volatile (why?)

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++;
  }
  …
}
```

stamp

top

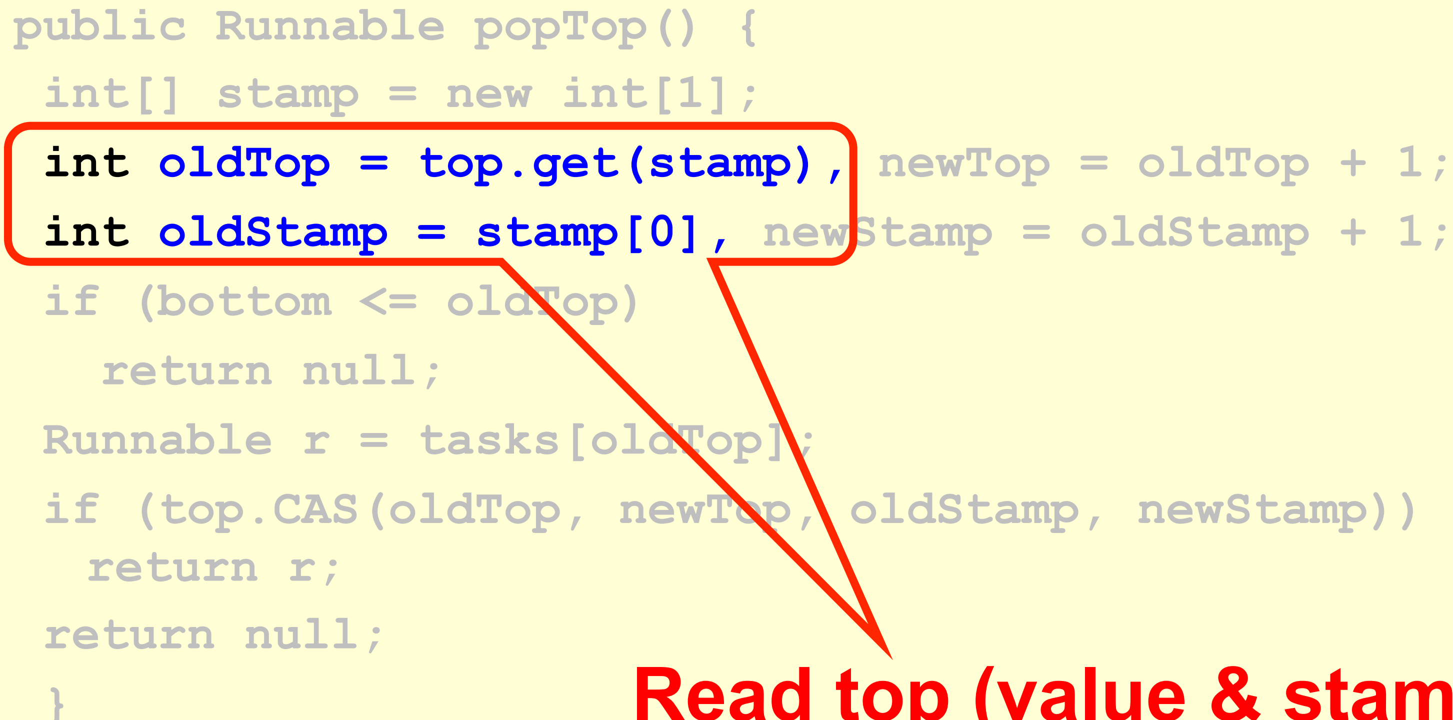
bottom

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



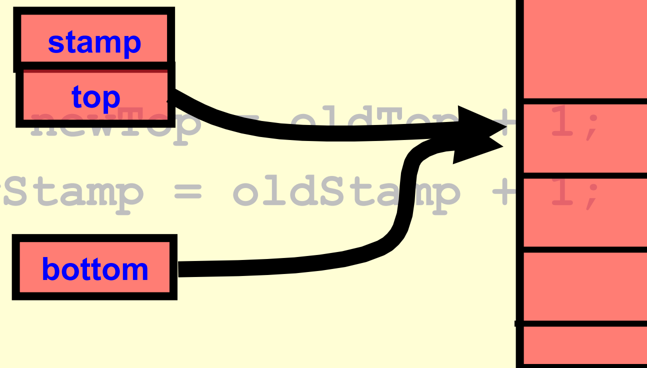
Steal Work

```
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

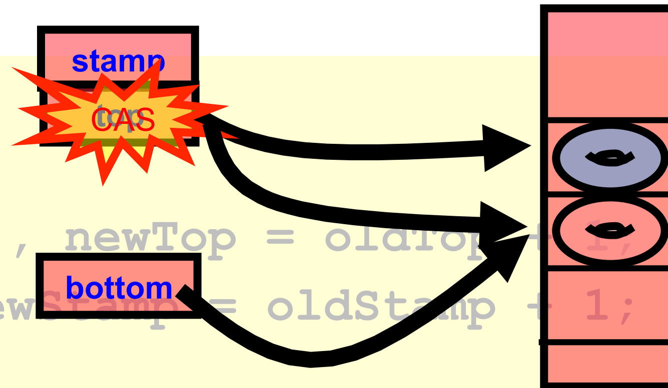
Steal Work

```
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

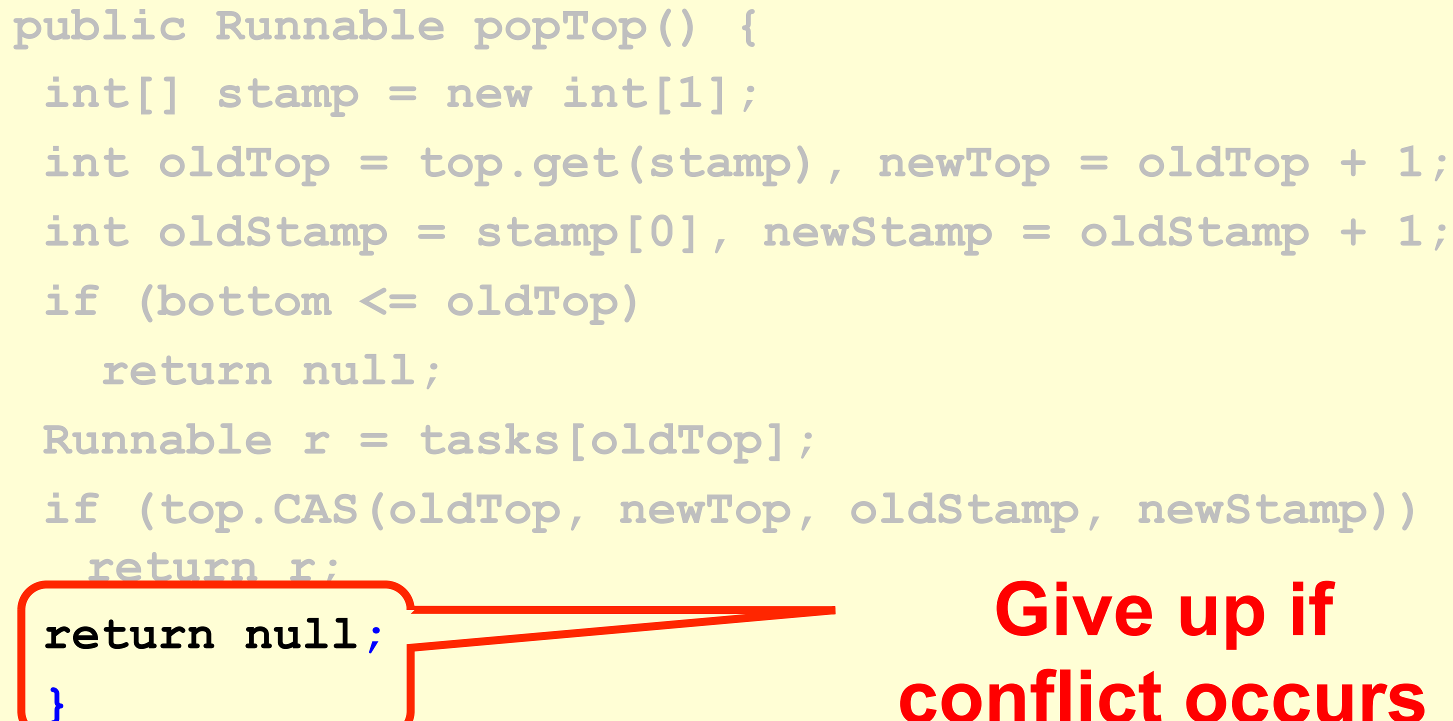
Steal Work




```
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



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() {
 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; }
```

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;}
```

stamp

top

bottom

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;}
```

stamp
top
bottom

At most one item left

Take Work

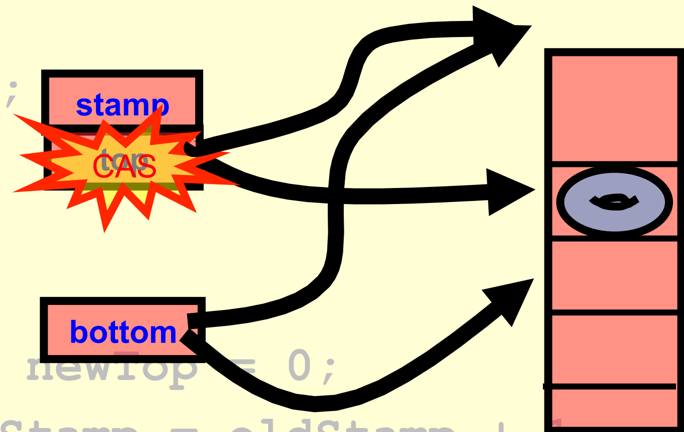
Try to steal last task.
Reset `bottom` because the DEQueue will be empty even if unsuccessful (why?)

```
Runnable popBottom() {
 if (bottom == 0) return null;
 b
 Runnable r = tasks[bottom];
 int[] stamp = new int[1];
 i
 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;}
```

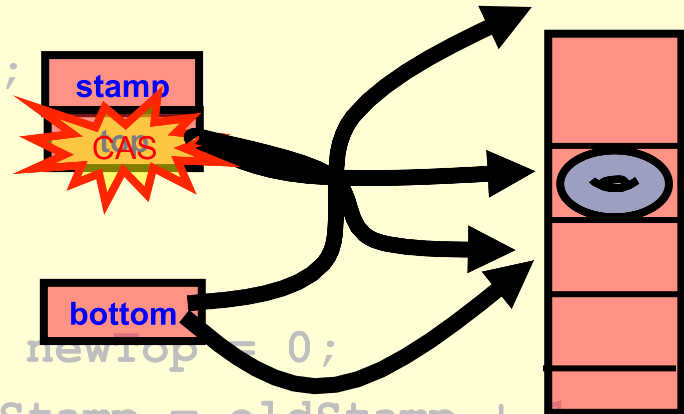
I win CAS



Take Work

```
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 I lose CAS, thief must have won…



Take Work

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

Must still reset top and bottom since deque is empty

```
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

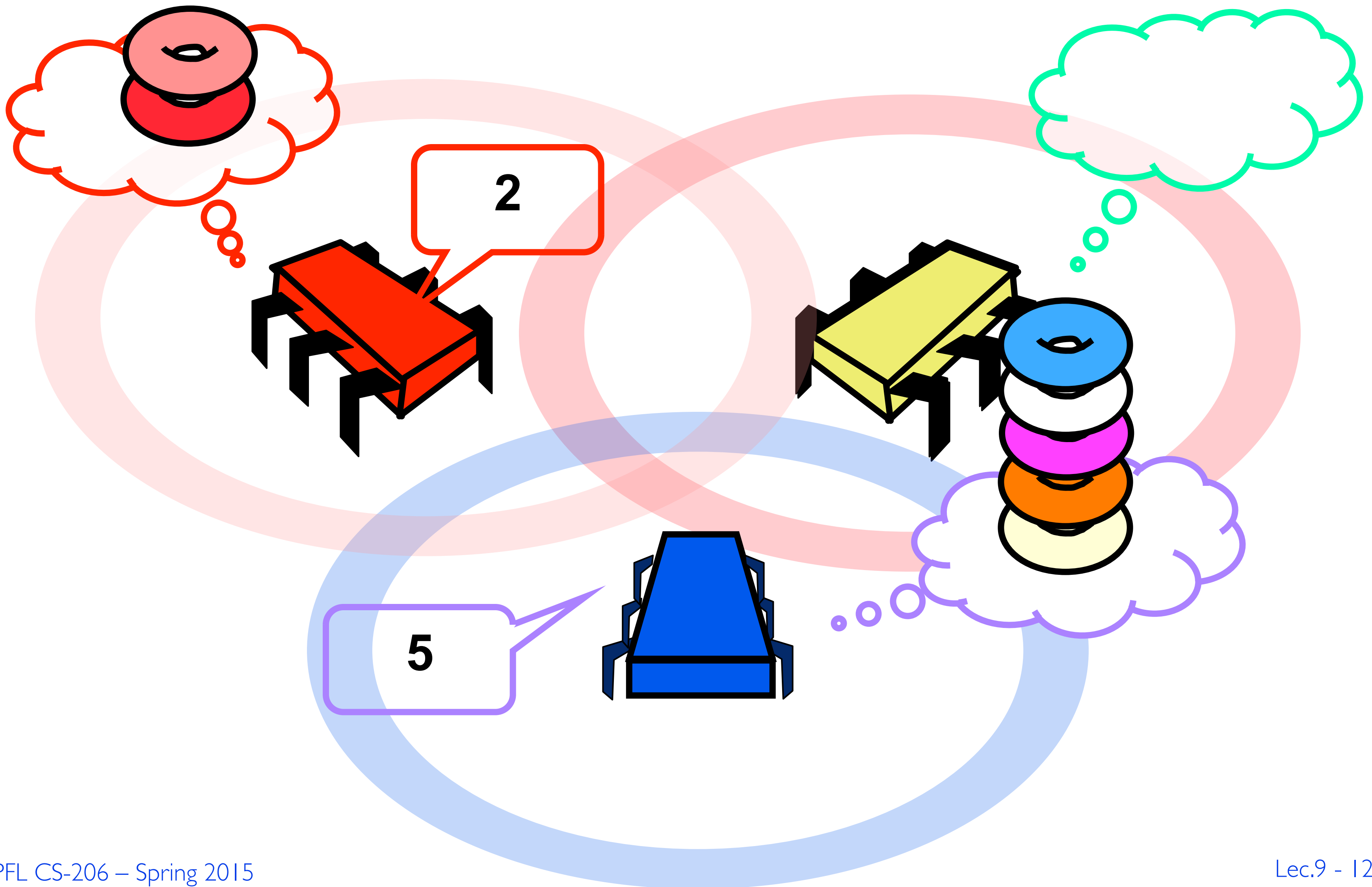
- "May as well be hanged for stealing a sheep as a goat"
- From which we conclude:
  - Stealing was punished severely
  - Sheep were worth more than goats

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]);
}}}}}
```

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]);
}}}}}
```

Summary

- Futures are lighter weight
- Worry about the critical path, speedup
- Work distribution
- Work dealing is flawed
- Work stealing can work but expensive
- Work balancing