Multiprocessor Synchronization
CSCI 176

Futures, Scheduling, and Work Distribution

Lecture 21
22 November 2022

Maurice Herlihy
Brown University
How to write Parallel Apps?

- Split a program into parallel parts
- In an effective way
- Thread management
Matrix Multiplication

\[
(C) = (A) \cdot (B)
\]
Matrix Multiplication

\[ c_{ij} = \sum_{k=0}^{n-1} a_{ki} \cdot b_{jk} \]

No synchronization!

Massive parallelism!
Matrix Multiplication

class Worker extends Thread {
    int row, col;
    Worker(int row, int col) {
        row = row; 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) {
        row = row; 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) {
        row = row; 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) {
        row = row; 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

```java
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

```java
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

```java
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();
        }
    }
}
```

Art of Multiprocessor Programming
Thread Overhead

Threads require *resources*

- Memory for stacks
- Setup, teardown
- Scheduler overhead

- Short-lived threads
- Bad ratio of work versus overhead
Thread Pools

More sensible to keep a *pool* of long-lived threads

Thread:

- Assigned a short-lived *task*

  *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

Package `java.util.concurrent`

No result value expected?

Task = Runnable object

Call `void run()`.

Type T result expected?

Task = Callable<T> object

Call `T call()`.
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.
Callable\(<<T>\)\ task = \ldots; \\
\ldots \\
Future\(<<T>\)\ future = executor.submit(task); \\
\ldots \\
T\ value = future.get(); \\
The Future’s get() method blocks until the value is available
Runnable task = …;

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

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

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

The Future's `get()` method blocks until the computation is complete (no return value)
Warning

Executor Service requests …

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

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

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 aij and bij)
            Future<?> f00 = exec.submit(addTask(a00,b00));
            ...
            Future<?> f11 = exec.submit(addTask(a11,b11));
            f00.get(); ...; f11.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 aij and bij)
            Future<? extends ? extends ? > f00 = exec.submit(addTask(a00,b00));
            ...
            Future<? extends ? extends ? > f11 = exec.submit(addTask(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 aij and bij
            Future<?> f00 = exec.submit(addTask(a00,b00));
            ...
            Future<?> f11 = exec.submit(addTask(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 aij and bij)
            Future<?> f00 = exec.submit(addTask(a00,b00));
            ...
            Future<?> f11 = exec.submit(addTask(a11,b11));
            f00.get(); ...; f11.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 aij and bij)
            Future<?> f00 = exec.submit(addTask(a00,b00));
            ... 
            Future<?> f11 = exec.submit(addTask(a11,b11));
            f00.get(); ...; f11.get();
            ...
        }
    }
}

Let them finish
Dependencies

Matrix example is not typical!

Tasks are *independent*

Don’t need results of one task …

To complete another

Often tasks are *not independent*
Fibonacci

\[ F(n) \left\{ \begin{array}{ll} 1 & \text{if } n = 0 \text{ or } 1 \\ F(n-1) + F(n-2) & \text{otherwise} \end{array} \right. \]

**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

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;
        }
    }
}
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

\[ \text{fib(4)} \]
Fibonacci DAG

fib(3) → fib(4)
Fibonacci DAG

fib(4) → fib(3) → fib(2) → fib(2)
Fibonacci DAG

fib(4) → fib(3) → fib(2) → fib(1)

fib(1) → fib(1) → fib(1)

fib(1) → fib(2) → fib(3) → fib(4) → fib(5)
Fibonacci DAG

- fib(4)
- fib(3)
- fib(2)
- fib(1)

Call and get connections between nodes.
How Parallel is That?

Define *work*: Total time on one processor

Define *span* (critical-path length):
Longest dependency path

Never faster than that!
Unfolded DAG
Parallelism?

Serial fraction $= \frac{3}{18} = \frac{1}{6}$ …

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?

$T_\infty$: time needed on as many processors as you like
Critical Path?

$T_\infty$: time needed on as many processors as you like

Longest path ....

$T_\infty = 9$
Notation Watch

\[ T_P = \text{time on } P \text{ processors} \]

\[ T_1 = \text{work (time on 1 processor)} \]

\[ T_\infty = \text{critical path length (time on } \infty \text{ processors)} \]
Simple Laws

**Work Law:** $T_P \geq \frac{T_1}{P}$

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**

$T_1/T_P = \Theta(P)$

**Max speedup (average parallelism)**

$T_1/T_\infty$
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: $\max\{T_\infty(A), T_\infty(B)\}$
Matrix Addition Review

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

4 parallel additions
Addition

Let $A_P(n)$ be running time

For $n \times n$ matrix on $P$ processors

For example

$A_1(n)$ is work

$A_\infty(n)$ is critical path length
Addition Work

\[ A_1(n) = 4A_1(n/2) + \Theta(1) \]

4 spawned additions

Partition, synch, etc
Addition Work

\[ A_1(n) = 4 \ A_1\left(\frac{n}{2}\right) + \Theta(1) \]
\[ = \Theta(n^2) \]

Same as double-loop summation
Addition Span

\[ A_\infty(n) = A_\infty\left(\frac{n}{2}\right) + \Theta(1) \]

Partition, synch, etc

spawned additions in parallel
Addition Span

\[ A_\infty(n) = A_\infty\left(\frac{n}{2}\right) + \Theta(1) = \Theta(\log n) \]
Matrix Multiplication

\[(C) = (A) \cdot (B)\]
Matrix Multiplication Redux

\[
\begin{pmatrix}
C_{00} & C_{01} \\
C_{10} & C_{11}
\end{pmatrix}
= 
\begin{pmatrix}
A_{00} & A_{01} \\
A_{10} & A_{11}
\end{pmatrix}
\cdot 
\begin{pmatrix}
B_{00} & B_{01} \\
B_{10} & B_{11}
\end{pmatrix}
\]
Matrix Multiplication Redux

\[
\begin{pmatrix}
C_{00} & C_{01} \\
C_{10} & C_{11}
\end{pmatrix} =
\begin{pmatrix}
A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\
A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{10}
\end{pmatrix}
\]
Matrix Multiplication Redux

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

Phase 1: 8 multiplications
Matrix Multiplication Redux

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

Phase 2: 4 additions
Multiplication Work

\[ M_1(n) = 8 M_1\left(\frac{n}{2}\right) + A_1(n) \]

8 parallel multiplications

Final addition
Multiplication Work

\[ M_1(n) = 8 \ M_1\left(\frac{n}{2}\right) + A_1(n) = \Theta(n^3) \]

Same as serial triple-nested loop
Multiplication Span

\[ M_\infty(n) = M_\infty\left(\frac{n}{2}\right) + A_\infty(n) \]

Final addition

Half-size parallel multiplications
Multiplication Span

\[ M_\infty(n) = M_\infty\left(\frac{n}{2}\right) + A_\infty(n) \]
\[ = M_\infty\left(\frac{n}{2}\right) + \Theta(\log n) \]
\[ = \Theta(\log^2 n) \]
Parallelism

\[ M_1(n) / M_\infty(n) = \Theta(n^3 / \log^2 n) \]

To multiply two 1000 x 1000 matrices

\[ 1000^3 / 10^2 = 10^7 \]

Much more than number of processors on any real machine
Shared-Memory Multicores

- Parallel applications
- No direct access to HW processors
- Mix of other jobs
- All run together
- Come & go dynamically
Ideal Scheduling Hierarchy

- Tasks
- User-level scheduler
- Processors
Realistic Scheduling Hierarchy

- Tasks
  - User-level scheduler
- Threads
  - Kernel-level scheduler
- Processors
For Example

Initially,

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

Map threads onto $P$ processes

All $P$ Cannot get $P$-fold speedup

What if the kernel doesn’t cooperate?

Can try for speedup proportional to $P$
Scheduling Hierarchy

User-level scheduler
- Tells kernel which threads are ready

Kernel-level scheduler
- Synchronous (for analysis, not correctness!)
- Picks $p_i$ threads to schedule at step $i$
Greedy Scheduling

A node is *ready* if predecessors are done

*Greedy*: schedule as many ready nodes as possible

*Optimal* scheduler is greedy (why?)

But not every greedy scheduler is optimal
Greedy Scheduling

There are $P$ processors

**Complete Step:**
If $> P$ nodes ready, run any $P$

**Incomplete Step:**
If $\leq P$ nodes ready, run them all
Theorem

For any greedy scheduler,

\[ T_P \leq \frac{T_1}{P} + T_\infty \]

Actual time
Theorem

For any greedy scheduler,

\[ T_P \leq \frac{T_1}{P} + T_\infty \]

No better than work divided among processors
Theorem

For any greedy scheduler,

\[ T_P \leq T_{1/P} + T_\infty \]

No better than critical path length
Proof.

Number of *incomplete steps* ≤ \( T_1/P \) …

… because each performs \( P \) work.

Number of *complete steps* ≤ \( T_\infty \) …

… because each shortens the unexecuted critical path by 1

QED
Near-Optimality

**Theorem:** any greedy scheduler is within a factor of 2 of optimal.

**Remark:** Optimality is NP-hard!
Proof of Near-Optimality

Let $T^*_P$ be the optimal time.

$T^*_P \geq \max\{T_1/P, T_\infty\}$

$T_P \leq T_1/P + T_\infty$

$T_P \leq 2 \max\{T_1/P, T_\infty\}$

$T_P \leq 2 T^*_P$

From work and critical path laws

Theorem

QED
Work Distribution
Work Dealing

Yes!
The Problem with Work Dealing

D’oh!
Work Stealing

Yes!

No work…
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. Double-Ended Queue
Obtain Work

Obtain work
Run task until it
Blocks or terminates

popBottom
New Work

- Unblock node
- Spawn node

pushBottom
Whatcha Gonna do When the Well Runs Dry?

empty

@&%$!!
Steal Work from Others

Pick random thread’s DEQeueue
Steal this Task!

popTop
Task DEQueue

- pushBottom
- popBottom
- popTop

Never happen concurrently
Task DEQueue

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

popTop() may fail if

Concurrent popTop() succeeds, or a

Concurrent popBottom() takes last task
Dreaded ABA Problem

CAS
Dreaded ABA Problem

top
Dreaded ABA Problem
Dreaded ABA Problem
Dreaded ABA Problem
Dreaded ABA Problem
Dreaded ABA Problem
Dreaded ABA Problem

CAS

Uh-Oh …
Fix for Dreaded ABA

- stamp
- top
- bottom
Bounded DEQueue

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

    ...
}
```
public class BDEQueue {

    AtomicStampedReference<Integer> top;
    volatile int bottom;
    Runnable[] tasks;

    ...

    }

Bounded DEQueue

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

Array of tasks
public class BDEQueue {
    ...
    void pushBottom(Runnable r) {
        tasks[bottom] = r;
        bottom++;
    }
    ...
}
public class BDEQueue {

... void pushBottom(Runnable r) {
    tasks[bottom] = r;
    bottom++;
}

...}

Where to store the new task in the array
public class BDEQueue {
    ...
    void pushBottom(Runnable r) {
        tasks[bottom] = r;
        bottom++;
    }
    ...
}

Adjust the bottom index
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;
}
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)
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;
}
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;
}
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;
}
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;
}
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;
}
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;
}
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;
}
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;
}
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 r;
}

If top & bottom one or more apart, no conflict
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;
}
Runnable popBottom() {
    int 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;
}
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;
}
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;
}
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;
}

Art of Multiprocessor Programming
“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 requires expensive CAS

But only one task stolen

What if we could

Steal more than one task

Randomly balance loads?
Work Balancing

\[
\left\lfloor \frac{2+5}{2} \right\rfloor = 3
\]

\[
\left\lfloor \frac{22+5}{2} \right\rfloor = 4
\]
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]);
                }
            }
        }
    }
}
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]);
                    balance(queue[min], queue[max]);
                    balance(queue[min], queue[max]);
                }
            }
        }
    }
}
Work-Balancing Thread

```java
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 $1/|\text{queue}|$
Work-Balancing Thread

```java
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
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]);
                }
            }
        }
    }
}
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 Stealing & Balancing

Clean separation between app & scheduling layer

Works well when number of processors fluctuates.

Works on “black-box” operating systems
Creative Commons Attribution-ShareAlike 2.5 License

- **You are free:**
  - to Share — to copy, distribute and transmit the work
  - to Remix — to adapt the work
- **Under the following conditions:**
  - Attribution. You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
  - Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
  - http://creativecommons.org/licenses/by-sa/3.0/.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.
\[ M_1(n) = 8 M_1(n/2) + A_1(n) \]

In an effective way

Thread management

Deal with underflow somehow …
Clean separation between app & scheduling layer

Works well when number of processors fluctuates.

Works on “black-box” operating systems.

Steal more than one task.

Randomly balance loads?

Waits for I/O.

We get that processor back …

Must tune fairness parameters.
TOM MARVOLO RIDDLE