Lecture 16
Synchronized Structures Part 3

Christopher Mitchell, Ph.D.
cmitchell@cs.nyu.edu || http://z80.me
Work Distribution
Work Dealing
The Problem with Work Dealing
Work Stealing

Core
Core
Core
Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core

Core
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 blocks or terminates

```
popBottom()
```
New Work

- Unblock node
- Spawn node

pushBottom()
No More Work

popBottom()
Steal Work from Others
Steal this Task!

[Diagram showing a processor with cores and tasks]
Task DEQueue

Methods

• pushBottom()
• popBottom()
• popTop()
Task DEQueue

Methods

• pushBottom()
• popBottom()
• popTop()

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

- Wait-Free
- Linearizable
- Constant time
Compromise

- Method `popTop()` may fail if
  - Concurrent `popTop()` succeeds, or a
  - Concurrent `popBottom()` takes last task
DEQueue ABA Problem

CAS(&top, old_top, new_top)
DEQueue ABA Problem
DEQueue ABA Problem
DEQueue ABA Problem
DEQueue ABA Problem
DEQueue ABA Problem
DEQueue ABA Problem
DEQueue ABA Problem

```c
CAS(&top, old_top, new_top)
```
DEQueue ABA Fix

code:

```
top : counter

CAS(&top, old_top, new_top)
```
Bounded DEQueue

class BDEQueue {
  PointerWithCounter top;
  Pointer bottom;
  Task[] tasks;
  //...
}

“Thread Scheduling for Multiprogrammed Multiprocessors” by Arora, Blumofe, and Plaxton
class BDEQueue {
    // ...
    void pushBottom(Task r) {
        tasks[bottom] = r;
        bottom++;
    }
    // ...
}
pushBottom()

class BDEQueue {
    // ...
    void pushBottom(Task r) {
        tasks[bottom] = r;
        bottom++;
    }
    // ...
}

top : counter
bottom
pushBottom()

class BDEQueue {
    // ...
    void pushBottom(Task r) {
        tasks[bottom] = r;
        bottom++;
    }
    // ...
}

top : counter

bottom
Task* popTop() {
  PointerWithCounter old_top = top;
  PointerWithCounter new_top = old_top;
  new_top.pointer++;
  new_top.counter++;
  if (bottom <= old_top.pointer) {
    return nullptr;
  }
  Task* rval = tasks[old_top.pointer];
  if (CAS(&top, old_top, new_top)) {
    return rval;
  } else {
    return nullptr;
  }
}
Task* popTop() {
    PointerWithCounter old_top = top;
    PointerWithCounter new_top = old_top;
    new_top.pointer++;
    new_top.counter++;
    if (bottom <= old_top.pointer) {
        return nullptr;
    }
    Task* rval = tasks[old_top.pointer];
    if (CAS(&top, old_top, new_top)) {
        return rval;
    } else {
        return nullptr;
    }
}
Steal Work

```
Task* popTop() {
  PointerWithCounter old_top = top;
  PointerWithCounter new_top = old_top;
  new_top.pointer++;
  new_top.counter++;
  if (bottom <= old_top.pointer) {
    return nullptr;
  }
  Task* rval = tasks[old_top.pointer];
  if (CAS(&top, old_top, new_top)) {
    return rval;
  } else {
    return nullptr;
  }
}
```
Task* popBottom() {
    if (bottom == nullptr) { return nullptr; }
    bottom--; 
    Task rval = tasks[bottom];
    PointerWithCounter old_top = top;
    PointerWithCounter new_top = old_top;
    new_top.pointer = nullptr;
    new_top.counter++;
    if (bottom > old_top.pointer) { return rval; } 
    if (bottom == old_top.pointer) {
        bottom = nullptr;
        if (CAS(&top, old_top, new_top)) { return rval; } 
    }
    bottom = nullptr;
    top = new_top;
    return nullptr;
}
Take Work

Task* popBottom() {
    if (bottom == nullptr) { return nullptr; }
    bottom--;
    Task* rval = tasks[bottom];
    PointerWithCounter old_top = top;
    PointerWithCounter new_top = old_top;
    new_top.pointer = nullptr;
    new_top.counter++;
    if (bottom > old_top.pointer) { return rval; }
    if (bottom == old_top.pointer) {
        bottom = nullptr;
        if (CAS(&top, old_top, new_top)) { return rval; }
    }
    bottom = nullptr;
    top = new_top;
    return nullptr;
}
Task* popBottom() {
    if (bottom == nullptr) { return nullptr; }
    bottom--;
    Task rval = tasks[bottom];
    PointerWithCounter old_top = top;
    PointerWithCounter new_top = old_top;
    new_top.pointer = nullptr;
    new_top.counter++;
    if (bottom > old_top.pointer) { return rval; }
    if (bottom == old_top.pointer) {
        bottom = nullptr;
        if (CAS(&top, old_top, new_top)) { return rval; }
    }
    bottom = nullptr;
    top = new_top;
    return nullptr;
}
Task* popBottom() {
    if (bottom == nullptr) { return nullptr; }
    bottom--;
    Task rval = tasks[bottom];
    PointerWithCounter old_top = top;
    PointerWithCounter new_top = old_top;
    new_top.pointer = nullptr;
    new_top.counter++;
    if (bottom > old_top.pointer) { return rval; }
    if (bottom == old_top.pointer) {
        bottom = nullptr;
        if (CAS(&top, old_top, new_top)) { return rval; }
    }
    top = new_top;
    bottom = nullptr;
    return nullptr;
}
Take Work

```c
Task* popBottom() {
    if (bottom == nullptr) { return nullptr; }
    bottom--;
    Task rval = tasks[bottom];
    PointerWithCounter old_top = top;
    PointerWithCounter new_top = old_top;
    new_top.pointer = nullptr;
    new_top.counter++;
    if (bottom > old_top.pointer) { return rval; }
    if (bottom == old_top.pointer) {
        bottom = nullptr;
        if (CAS(&top, old_top, new_top)) { return rval; }
    }
    top = new_top;
    bottom = nullptr;
    return nullptr;
}
```
Task* popBottom() {
    if (bottom == nullptr) { return nullptr; }
    bottom--;
    Task rval = tasks[bottom];
    PointerWithCounter old_top = top;
    PointerWithCounter new_top = old_top;
    new_top.pointer = nullptr;
    new_top.counter++;
    if (bottom > old_top.pointer) { return rval; }
    if (bottom == old_top.pointer) {
        bottom = nullptr;
        if (CAS(&top, old_top, new_top)) { return rval; }
    }
    top = new_top;
    bottom = nullptr;
    return nullptr;
}
Take Work

```c
Task* popBottom() {
    if (bottom == nullptr) { return nullptr; }
    bottom--;
    Task rval = tasks[bottom];
    PointerWithCounter old_top = top;
    PointerWithCounter new_top = old_top;
    new_top.pointer = nullptr;
    new_top.counter++;
    if (bottom > old_top.pointer) { return rval; }
    if (bottom == old_top.pointer) {
        bottom = nullptr;
        if (CAS(&top, old_top, new_top)) { return rval; }
    }
    top = new_top;
    bottom = nullptr;
    return nullptr;
}
```
```c++
Task* popBottom() {
    if (bottom == nullptr) { return nullptr; }
    bottom--;
    Task rval = tasks[bottom];
    PointerWithCounter old_top = top;
    PointerWithCounter new_top = old_top;
    new_top.pointer = nullptr;
    new_top.counter++;
    if (bottom > old_top.pointer) { return rval; }
    if (bottom == old_top.pointer) {
        bottom = nullptr;
        if (CAS(&top, old_top, new_top)) { return rval; }
    }
    top = new_top;
    bottom = nullptr;
    return nullptr;
}
```
Variations

• Stealing is expensive
  • Pay CAS
  • Only one task taken

• What if
  • Move more than one task
  • Randomly balance loads?
  • Use balancing thread
Work Stealing & Balancing

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

• Practical systems can’t predict workload costs a priori
• Need to distribute work reactively or proactively
• Work stealing, dealing, and balancing