Barrier Synchronization

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit
Simple Video Game

• Prepare frame for display
  – By graphics coprocessor

• “soft real-time” application
  – Need at least 35 frames/second
  – OK to mess up rarely
Simple Video Game

while (true) {
    frame.prepare();
    frame.display();
}
Simple Video Game

```java
while (true) {
    frame.prepare();
    frame.display();
}
```

• What about overlapping work?
  – 1\textsuperscript{st} thread displays frame
  – 2\textsuperscript{nd} prepares next frame
Two-Phase Rendering

```java
while (true) {
    if (phase) {
        frame[0].display();
    } else {
        frame[1].display();
    }
    phase = !phase;
}
```

```java
while (true) {
    if (phase) {
        frame[1].prepare();
    } else {
        frame[0].prepare();
    }
    phase = !phase;
}
```
Two-Phase Rendering

while (true) {
    if (phase) {
        frame[0].display();
    } else {
        frame[1].display();
    }
    phase = !phase;
}

while (true) {
    if (phase) {
        frame[1].prepare();
    } else {
        frame[0].prepare();
    }
    phase = !phase;
}

even phases
Two-Phase Rendering

```java
while (true) {
    if (phase) {
        frame[0].display();
    } else {
        frame[1].display();
    }
    phase = !phase;
}
while (true) {
    if (phase) {
        frame[1].prepare();
    } else {
        frame[0].prepare();
    }
    phase = !phase;
}
```

odd phases
Synchronization Problems

• How do threads stay in phase?
• Too early?
  – “we render no frame before its time”
• Too late?
  – Recycle memory before frame is displayed
Ideal Parallel Computation
Ideal Parallel Computation
Real-Life Parallel Computation
Real-Life Parallel Computation

Uh, oh
Barrier Synchronization
Barrier Synchronization
Barrier Synchronization

Until every thread has left here

No thread enters here
Why Do We Care?

• Mostly of interest to
  – Scientific & numeric computation

• Elsewhere
  – Garbage collection
  – Less common in systems programming
  – Still important topic
Duality

• Dual to mutual exclusion
  – Include others, not exclude them

• Same implementation issues
  – Interaction with caches …
    • Invalidation?
    • Local spinning?
Example: Parallel Prefix

```
  a  b  c  d
  a  a+b  a+b+c  a+b+c+d
```

before

after
Parallel Prefix

One thread
Per entry
Parallel Prefix: Phase 1

\[ \begin{array}{cccc}
  \text{a} & \text{b} & \text{c} & \text{d} \\
  \text{a} & \text{a+b} & \text{b+c} & \text{c+d} \\
\end{array} \]
Parallel Prefix: Phase 2

\[
\begin{align*}
& a, b, c, d \\
& a, a+b, a+b+c, a+b+c+d
\end{align*}
\]
Parallel Prefix

• $N$ threads can compute
  – Parallel prefix
  – Of $N$ entries
  – In $\log_2 N$ rounds

• What if system is asynchronous?
  – Why we need barriers
Prefix

class Prefix extends Thread {
    int[] a;
    int i;
    Barrier b;
    void Prefix(int[] a,
                      Barrier b, int i) {
        a = a;
        b = b;
        i = i;
    }
}
class Prefix extends Thread {
  int[] a;
  int i;
  Barrier b;
  void Prefix(int[] a,
               Barrier b, int i) {
    a = a;
    b = b;
    i = i;
  }
}

Array of input values
class Prefix extends Thread {
    int[] a;
    int i;
    Barrier b;
    
    void Prefix(int[] a, Barrier b, int i) {
        a = a;
        b = b;
        i = i;
    }
}

Thread index
class Prefix extends Thread {
    int[] a;
    int i;
    Barrier b;
    void Prefix(int[] a, Barrier b, int i) {
        a = a;
        b = b;
        i = i;
    }
}

Art of Multiprocessor Programming
class Prefix extends Thread {
    int[] a;
    int i;
    Barrier b;
    void Prefix(int[] a, Barrier b, int i) {
        a = a;
        b = b;
        i = i;
    }
}

Initialize fields
public void run() {
    int d = 1, sum = 0;
    while (d < N) {
        if (i >= d)
            sum = a[i-d];
        if (i >= d)
            a[i] += sum;
        d = d * 2;
    }
}
Where Do the Barriers Go?

```java
public void run() {
    int d = 1, sum = 0;
    while (d < N) {
        if (i >= d)
            sum = a[i-d];
        b.await();
        if (i >= d)
            a[i] += sum;
        d = d * 2;
    }
}
```

Make sure everyone reads before anyone writes
Where Do the Barriers Go?

```java
public void run() {
    int d = 1, sum = 0;
    while (d < N) {
        if (i >= d)
            sum = a[i-d];
        b.await();
        if (i >= d)
            a[i] += sum;
        b.await();
        d = d * 2;
    }
}
```

Make sure everyone reads before anyone writes

Make sure everyone writes before anyone reads
Barrier Implementations

• Cache coherence
  – Spin on locally-cached locations?
  – Spin on statically-defined locations?

• Latency
  – How many steps?

• Symmetry
  – Do all threads do the same thing?
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n){
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement()==1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement()==1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}

Barriers

Number of threads participating
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}

If I’m last, reset fields for next time
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n){
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement()==1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}

Otherwise, wait for everyone else
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0); // What’s wrong with this protocol?
        }
    }
}}
Barrier b = new Barrier(n);
while ( mumble() ) {
    work();
    b.await();
}

public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}

Waiting for Phase 1 to finish
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = new AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
Um-Oh

public class Barrier {
  AtomicInteger count;
  int size;
  public Barrier(int n) {
    count = new AtomicInteger(n);
    size = n;
  }
  public void await() {
    if (count.getAndDecrement() == 1) {
      count.set(size);
    } else {
      while (count.get() != 0); // Waiting for Phase 1 to finish
    }
  }
}

// Waiting for Phase 2 to finish

Art of Multiprocessor Programming
Basic Problem

• One thread “wraps around” to start phase 2
• While another thread is still waiting for phase 1
• One solution:
  – Always use two barriers
public class Barrier {
    AtomicInteger count;
    int size;
    volatile boolean sense = false;
    threadSense = new ThreadLocal<boolean>…

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = mySense
        } else {
            while (sense != mySense) {}
        }
        threadSense.set(!mySense)}
}
public class Barrier {
    AtomicInteger count;
    int size;
    volatile boolean sense = false;
    threadSense = new ThreadLocal<boolean>…

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = mySense
        } else {
            while (sense != mySense) {}
        }
        threadSense.set(!mySense)}
    }
public class Barrier {
    AtomicInteger count;
    int size;
    volatile boolean sense = false;
    threadSense = new ThreadLocal<boolean>…

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = mySense
        } else {
            while (sense != mySense) {}
        }
        threadSense.set(!mySense)
    }
}
public class Barrier {
    AtomicInteger count;
    int size;
    volatile boolean sense = false;
    threadSense = new ThreadLocal<boolean>…

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = mySense
        } else {
            while (sense != mySense) {}
        }
        threadSense.set(!mySense)}

Get new sense determined by last phase
public class Barrier {
  AtomicInteger count;
  int size;
  volatile boolean sense = false;
  threadSense = new ThreadLocal<boolean>…

  public void await {
    boolean mySense = threadSense.get();
    if (count.getAndDecrement() == 1) {
      count.set(size); sense = mySense
    } else {
      while (sense != mySense) {}
    }
    threadSense.set(!mySense)}

If I’m last, reverse sense for next time
public class Barrier {
    AtomicInteger count;
    int size;
    volatile boolean sense = false;
    ThreadLocal<boolean> threadSense = new ThreadLocal<boolean>... 

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = mySense
        } else {
            while (sense != mySense) {}
        }
        threadSense.set(!mySense)}}}
Sense-Reversing Barriers

public class Barrier {
    AtomicInteger count;
    int size;
    volatile boolean sense = false;
    threadSense = new ThreadLocal<boolean>…

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = mySense
        } else {
            while (sense != mySense) {}
        }
    }

    threadSense.set(!mySense)}}}
Combining Tree Barriers

2-barrier

2-barrier

2-barrier
Combining Tree Barriers

![Diagram of combining tree barrier structure]
Combining Tree Barrier

```java
public class Node{
    AtomicInteger count; int size;
    Node parent; volatile boolean sense;

    public void await() {...
        if (count.getAndDecrement()==1) {
            if (parent != null)
                parent.await();
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {} 
        }
    }...}}
```
Combining Tree Barrier

```java
public class Node{
    AtomicInteger count, int size;
    Node parent;
    volatile boolean sense;

    public void await() {
        if (count.getAndDecrement()==1) {
            if (parent != null)
                parent.await();
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {}
        }
    }
}
```
public class Node{
    AtomicInteger count; int size;
    Node parent; volatile boolean sense;

    public void await() {
        if (count.getAndDecrement() == 1) {
            if (parent != null)
                parent.await();
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {}
        }
    }
}
public class Node{
    AtomicInteger count; int size;
    Node parent; volatile boolean sense;

    public void await() {
        if (count.getAndDecrement()==1) {
            if (parent != null)
                parent.await();
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {}
        }
    }
}
Combining Tree Barrier

```java
public class Node{
    AtomicInteger count; int size;
    Node parent; volatile boolean sense;

    public void await() {...
        if (count.getAndDecrement()==1) {
            if (parent != null)
                parent.await()
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {} 
        }
    }...
}
```
public class Node {
    AtomicInteger count; int size;
    Node parent; volatile boolean sense;

    public void await() {
        if (count.getAndDecrement() == 1) {
            if (parent != null)
                parent.await();
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {}
        }
    }
}
public class Node{
    AtomicInteger count; int size; Node parent; volatile boolean sense;

    public void await() {
        if (count.getAndDecrement()==1) {
            if (parent != null) {
                parent.await();
                count.set(size);
                sense = mySense;
            } else {
                while (sense != mySense) {}
            }
        }
    }
}
Combining Tree Barrier

• No sequential bottleneck
  – Parallel getAndDecrement() calls

• Low memory contention
  – Same reason

• Cache behavior
  – Local spinning on bus-based architecture
  – Not so good for NUMA
Remarks

• Everyone spins on sense field
  – Local spinning on bus-based (good)
  – Network hot-spot on distributed architecture (bad)
• Not really scalable
Tournament Tree Barrier

• If tree nodes have fan-in 2
  – Don’t need to call `getAndDecrement()`
  – Winner chosen statically
• At level $i$
  – If $i$-th bit of id is 0, move up
  – Otherwise keep back
Tournament Tree Barriers

root

winner

loser

winner

loser

winner

loser

winner

loser
Tournament Tree Barriers

All flags blue
Tournament Tree Barriers

Loser thread sets winner’s flag
Tournament Tree Barriers

Loser spins on own flag
Tournament Tree Barriers

Winner spins on own flag
Tournament Tree Barriers

Winner sees own flag, moves up, spins
Tournament Tree Barriers

Bingo!
Tournament Tree Barriers

Sense-reversing:
next time use blue flags
Tournament Barrier

class TBARRIER {
    volatile boolean flag;
    TBARRIER partner;
    TBARRIER parent;
    boolean top;
    ...
}

Art of Multiprocessor Programming
Tournament Barrier

class TBarrier {
    volatile boolean flag;
    TBarrier partner;
    TBarrier parent;
    boolean top;
    ...
}

Notifications delivered here
Tournament Barrier

class TBARRIER {
    volatile boolean flag;
    TBARRIER partner;
    TBARRIER parent;
    boolean top;
    ...
}

Other thread at same level
Tournament Barrier

class TBarrier {
    volatile boolean flag;
    TBarrier partner;
    TBarrier parent;
    boolean top;
    ...
}

Parent (winner) or null (loser)
Tournament Barrier

class TBarrier {
    volatile boolean flag;
    TBarrier partner;
    TBarrier parent;
    boolean top;
    ...
}

Am I the root?
void await(boolean mySense) {
    if (top) {
        return;
    } else if (parent != null) {
        while (flag != mySense) {
        parent.await(mySense);
        partner.flag = mySense;
    } else {
        partner.flag = mySense;
        while (flag != mySense) {
    }
}
Tournament Barrier

```java
void await(boolean mySense) {
    if (top) {
        return;
    } else if (parent != null) {
        while (flag != mySense) {};
        parent.await(mySense);
        partner.flag = mySense;
    } else {
        partner.flag = mySense;
        while (flag != mySense) {};
    }
}
```

Le root, c’est moi

Current sense

Current sense
Tournament Barrier

void await(boolean mySense) {
  if (top) {
    return;
  } else if (parent != null) {
    while (flag != mySense) {};
    parent.await(mySense);
    partner.flag = mySense;
  } else {
    partner.flag = mySense;
    while (flag != mySense) {};
  }
}

I am already a winner
Tournament Barrier

```java
void await(boolean mySense) {
    if (top) {
        return;
    } else if (parent != null) {
        while (flag != mySense) {};
        parent.await(mySense);
        partner.flag = mySense;
    } else {
        partner.flag = mySense;
        while (flag != mySense) {};
    }
}
```

Wait for partner
Tournament Barrier

void await(boolean mySense) {
    if (top) {
        return;
    } else if (parent != null) {
        while (flag != mySense) {};
        parent.await(mySense);
        partner.flag = mySense;
    } else {
        partner.flag = mySense;
        while (flag != mySense) {};
    }
}
void await(boolean mySense) {
    if (top) {
        return;
    } else if (parent != null) {
        while (flag != mySense) {};
        parent.await(mySense);
        partner.flag = mySense;
    } else {
        partner.flag = mySense;
        while (flag != mySense) {};
    }
}
Tournament Barrier

```java
void await(boolean mySense) {
    if (top) {
        return;
    } else if (parent != null) {
        while (flag != mySense) {}
        parent.await(mySense);
        partner.flag = mySense;
    } else {
        partner.flag = mySense;
        while (flag != mySense) {}
    }
}
```

Inform partner

Order is important (why?)
void await(boolean mySense) {
    if (top) {
        return;
    } else if (parent != null) {
        while (flag != mySense) {};
        parent.await(mySense);
        partner.flag = mySense;
    } else {
        partner.flag = mySense;
        while (flag != mySense) {};
    }
}
Tournament Barrier

```java
void await(boolean mySense) {
    if (top) {
        return;
    } else if (parent != null) {
        while (flag != mySense) {};
        parent.await(mySense);
        partner.flag = mySense;
    } else {
        partner.flag = mySense;
        while (flag != mySense) {};
    }
}
```

Tell partner I’m here
void await(boolean mySense) {
    if (top) {
        return;
    } else if (parent != null) {
        while (flag != mySense) {};
        parent.await(mySense);
        partner.flag = mySense;
    } else {
        partner.flag = mySense;
        while (flag != mySense) {};
    }
}
Remarks

• No need for read-modify-write calls
• Each thread spins on fixed location
  – Good for bus-based architectures
  – Good for NUMA architectures
Dissemination Barrier

• At round $i$
  – Thread $A$ notifies thread $A + 2^i \pmod{n}$

• Requires $\log n$ rounds
Dissemination Barrier

+1  +2  +4
Remarks

• Elegant
• Good source of homework problems
• Not cache-friendly
Ideas So Far

• Sense-reversing
  – Reuse without reinitializing

• Combining tree
  – Like counters, locks …

• Tournament tree
  – Optimized combining tree

• Dissemination barrier
  – Intellectually Pleasing (matter of taste)
Which is best for Multicore?

• On a cache coherent multicore chip: perhaps none of the above…
• Here is another (arguably) better algorithm …
Static Tree Barrier

One node per thread, statically assigned
Static Tree Barrier

Sense-reversing flag
Static Tree Barrier

Node has count of missing children
Static Tree Barrier

Spin until zero …
Static Tree Barrier

My counter is zero, decrement parent
Static Tree Barrier

Spin on done flag

Art of Multiprocessor Programming
Static Tree Barrier
Static Tree Barrier
Static Tree Barrier
Static Tree Barrier
Static Tree Barrier
Static Tree Barrier

yowzah!
Static Tree Barrier

yowzah!
Static Tree Barrier

yowzah!
Static Tree Barrier

yes!

yes!

yes!

yes!
Static Tree Barrier

Art of Multiprocessor Programming
Remarks

• Very little cache traffic
• Minimal space overhead
• On message-passing architecture
  – Send notification & sense down tree
The Nature of Progress*

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit
Concurrent Programming

• Many real-word data structures
  – blocking (lock-based) implementations &
  – non-blocking (no locks) implementations

• For example:
  – linked lists, queues, stacks, hash maps,…

Does this make sense?
Concurrent Programming

• Many data structures combine blocking & non-blocking methods
• Java™ concurrency package
  – skiplists, hash tables, excl.
  – on 10 million desktops

Can seemingly contradictory conditions co-exist in same alg?…
Progress Conditions

• **Deadlock-free:**
  – Some thread eventually acquires lock.

• **Starvation-free:**
  – Every thread eventually acquires lock.

• **Lock-free:**
  – Some method call returns.

• **Wait-free:**
  – Every method call returns.

• **Obstruction-free:**
  – Every method call returns if it executes in isolation.

We will show an example shortly.
List-Based Sets

• Unordered collection of elements
• No duplicates
• Methods
  – Add() a new element
  – Remove() an element
  – Contains() if element is present
Coarse Grained Locking

Lock is starvation-free: every attempt to acquire the lock eventually succeeds.
Fine Grained (Lock Coupling)

Overlapping locks detect overlapping operations

Deadlock-free: some thread eventually acquires lock.
Optimistic Fine Grained

add(), remove(), contains() lock destination nodes in order

Deadlock-free: some thread trying to acquire the locks eventually succeeds.
Obstruction-free contains()

Snapshot: if all nodes traversed twice are the same

Obstruction-free: the method returns if it executes in isolation for long enough.
The Simple Snapshot is Obstruction-Free

- Put increasing labels on each entry
- Collect twice
- If both agree,
  - We’re done
- Otherwise,
  - Try again
Obstruction-freedom

• **In the simple snapshot alg:**
  – The update method is wait-free
  – **But** scan is obstruction-free
    • Completes if it executes in isolation
    • (no concurrent updates).
Use mark bit + list ordering

1. Not marked $\rightarrow$ in the set
2. Marked or missing $\rightarrow$ not in the set
Lazy List-based Set Alg

Combine blocking and non-blocking: 
*deadlock-free* 
*add()* and *remove()* and *wait-free* 
*contains()*
Lock-free List-Based Set

Logical Removal =
Set Mark Bit

mark and reference
CASed together

CAS will fail

*Lock-free* add() and remove() and
*wait-free* contains()
So how can this make sense?

• Why have methods with different progress conditions?
• Let us try to understand this…
Progress Conditions

- **Deadlock-free:**
  - Some thread eventually acquires lock.
- **Starvation-free:**
  - Every thread eventually acquires lock.
- **Lock-free:**
  - Some method call returns.
- **Wait-free:**
  - Every method call returns.
- **Obstruction-free:**
  - Every method call returns if it executes in isolation.
A “Periodic Table” of Progress Conditions

<table>
<thead>
<tr>
<th>All make progress</th>
<th>Non-Blocking</th>
<th>Blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait-free</td>
<td>Obstruction-free</td>
<td>Starvation-free</td>
</tr>
<tr>
<td>Lock-free</td>
<td>Deadlock-free</td>
<td></td>
</tr>
</tbody>
</table>

All make progress
Some make progress
More Formally

- Standard notion of abstract object
- Progress conditions relate to method calls of an object
  - A thread is *active if it takes an infinite number of concrete (machine level) steps*
  - And is *suspended if not*. 
Maximal vs. Minimal

• Minimal progress
  – some call eventually completes
  – System matters, not individuals

• Maximal progress
  – every call eventually completes.
  – Individuals matter
The “Periodic Table” of Progress Conditions

<table>
<thead>
<tr>
<th></th>
<th>Non-Blocking</th>
<th></th>
<th>Blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximal progress</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Wait-free</td>
<td>Obstruction-free</td>
<td>Starvation-free</td>
</tr>
<tr>
<td>Minimal progress</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Lock-free</td>
<td></td>
<td>Deadlock-free</td>
</tr>
</tbody>
</table>
The Scheduler’s Role

Multiprocessor progress properties:
• Are not about the guarantees a method's implementation provides.
• Are about scheduling needed to provide minimal or maximal progress.
Fair Scheduling

• A history is *fair* if each thread takes an infinite number of steps

• A method implementation is *deadlock-free* if it guarantees minimal progress in every *fair history*. 
Starvation Freedom

- A method implementation is starvation-free if it guarantees maximal progress in every fair history.
Dependent Progress

• Dependent progress conditions
  – Do not guarantee minimal progress in every history

• Independent ones do.

• Blocking progress conditions
  – *deadlock*-freedom, *Starvation*-freedom
  – are dependent.
Non-blocking Independent Conditions

- A lock-free method guarantees
  - minimal progress
  - in every history.

- A wait-free method guarantees
  - maximal progress
  - in every history.
The “Periodic Table” of Progress Conditions

<table>
<thead>
<tr>
<th>Maximal progress</th>
<th>Minimal progress</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximal progress</td>
<td>Minimal progress</td>
</tr>
<tr>
<td>Non-Blocking</td>
<td>Blocking</td>
</tr>
<tr>
<td>Wait-free</td>
<td>Obstruction-free</td>
</tr>
<tr>
<td>Starvation-free</td>
<td>Deadlock-free</td>
</tr>
<tr>
<td>Independent</td>
<td>Dependent</td>
</tr>
</tbody>
</table>
Uniformly Isolating Schedules

- A history is uniformly isolating if any thread eventually runs by itself for "long enough"
- Modern systems do this with backoff, yield, etc.
A Non-blocking Dependent Condition

- A method implementation is obstruction-free if it guarantees
  - maximal progress
  - in every uniformly isolating history.
The “Periodic Table” of Progress Conditions

Non-Blocking

Maximal progress

Uniform iso

Blocking

Minimal progress

Starvation-free

Lock-free

Wait-free

Independent

Fair scheduler

Dependent

Obstruction-free

Lock-free
The “Periodic Table” of Progress Conditions

<table>
<thead>
<tr>
<th>Maximal progress</th>
<th>Non-Blocking</th>
<th>Blocking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Wait-free</td>
<td>Obstruction-free</td>
<td>Starvation-free</td>
</tr>
<tr>
<td>Lock-free</td>
<td>Clash-free</td>
<td>Deadlock-free</td>
</tr>
</tbody>
</table>

Independent

Dependent
Clash-Freedom: the “Einsteinium” of Progress

• A method implementation is clash-free if it guarantees
  – minimal progress
  – in every uniformly isolating history.

• Thm: clash-freedom strictly weaker than obstruction-freedom
Getting from Minimal to Maximal

<table>
<thead>
<tr>
<th>Maximal progress</th>
<th>Minimal progress</th>
</tr>
</thead>
<tbody>
<tr>
<td>Non-Blocking</td>
<td>Blocking</td>
</tr>
<tr>
<td>Wait-free</td>
<td>Obstruction-free</td>
</tr>
<tr>
<td>Lock-free</td>
<td>Starvation-free</td>
</tr>
<tr>
<td>Independent</td>
<td>Dependent</td>
</tr>
</tbody>
</table>

But helping is expensive
Maximal Progress Postulate

• Programmers want maximal progress.
• Methods’ progress conditions define
  – What we expect from the scheduler
  – For example
    • Don’t halt in critical section
    • Let me run in isolation long enough …
Why Lock-Free is OK

• We all want maximal progress
  – Wait-free
• Yet we often write lock-free or deadlock-free lock-based algorithms
• OK if we expect the scheduler to be benevolent
  – Often true (not always!)
Shared-Memory Computability

• What is (and is not) concurrently computable
• Wait-free Atomic Registers
• Lock-free/Wait-free Hierarchy
and Universal Constructions
Troubling Intellectual Question...

- Why use *non-blocking lock-free and wait-free* conditions when most code uses locks?
The Answer

• Not about being non-blocking…
• About being independent!
• Do not rely on the good behavior of the scheduler.
Shared-Memory Computability

- Independent progress: use *Lock-free* and *Wait-free* Memory Hierarchy and Universal Constructions
Programmers Expect the Best

- Programmers expect maximal progress.
- Progress conditions define scheduler requirements necessary to achieve it.
This work is licensed under a Creative Commons Attribution-ShareAlike 2.5 License.

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