Synchronization: Implementing Barriers Promises + Futures

Chris Rossbach
CS378H
Today

• Questions?

• Administrivia
  • Lab 2 due sooner than you’d like

• Material for the day
  • Barrier implementation
  • Promises & Futures

• Acknowledgements
  • Thanks to Gadi Taubenfield: I borrowed from some of his slides on barriers
Faux Quiz  (answer any N, 5 min)

• How are promises and futures related? Since there is disagreement on the nomenclature, don’t worry about which is which—just describe what the different objects are and how they function.
Barriers
Barriers
Review: Barrier Basics
Review: Barrier Basics

- Coordination mechanism
- participants wait until all reach same point.
- Once all reach it, all can pass.
Review: Barrier Basics

- Coordination mechanism
- participants wait until all reach same point.
- Once all reach it, all can pass.
Review: Barrier Basics

- Coordination mechanism
- Participants wait until all reach same point.
- Once all reach it, all can pass.
- Workhorse of BSP programming models
Review: Barrier Basics

- Coordination mechanism
- Participants wait until all reach same point.
- Once all reach it, all can pass.
- Workhorse of BSP programming models
Review: Barrier Basics

- Coordination mechanism
- Participants wait until all reach same point.
- Once all reach it, all can pass.
- Workhorse of BSP programming models

Fundamental primitive in many parallel models
Can you make a lock with a barrier?

**Coordination mechanism**
- Participants wait until all reach the same point.
- Once all reach it, all can pass.
- Workhorse of BSP programming models

**Fundamental primitive in many parallel models**
Barriers based on counters
Counter Barrier Ingredients

Fetch-and-Increment register

- A shared register that supports a F&I operation:
- Input: register $r$
- Atomic operation:
  - $r$ is incremented by 1
  - the old value of $r$ is returned

```plaintext
function fetch-and-increment (r : register)
    orig_r := r;
    r := r + 1;
    return (orig_r);
end-function
```

Await

- For brevity, we use the `await` macro
- Not an operation of an object
- This is just “spinning”

```plaintext
macro await (condition : boolean condition)
    repeat
        cond = eval(condition);
    until (cond)
end-macro
```
Simple Barrier Using an Atomic Counter

<table>
<thead>
<tr>
<th>shared</th>
<th>counter: fetch and increment reg. – {0,..n}, initially = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>go: atomic bit, initial value does not matter</td>
</tr>
<tr>
<td>local</td>
<td>local.go: a bit, initial value does not matter</td>
</tr>
<tr>
<td></td>
<td>local.counter: register</td>
</tr>
</tbody>
</table>
Simple Barrier Using an Atomic Counter

<table>
<thead>
<tr>
<th>shared</th>
<th>counter: fetch and increment reg. – ({0, \ldots, n}), initially = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>go: atomic bit, initial value does not matter</td>
</tr>
<tr>
<td>local</td>
<td>local.go: a bit, initial value does not matter</td>
</tr>
<tr>
<td></td>
<td>local.counter: register</td>
</tr>
</tbody>
</table>

1. `local.go := go`
2. `local.counter := fetch-and-increment (counter)`
3. `if local.counter + 1 = n then`
4. `counter := 0`
5. `go := 1 - go`
6. `else await(local.go ≠ go)`
### Simple Barrier Using an Atomic Counter

<table>
<thead>
<tr>
<th>shared</th>
<th>counter: fetch and increment reg. – {0,..n}, initially = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>go: atomic bit, initial value does not matter</td>
</tr>
<tr>
<td>local</td>
<td>local.go: a bit, initial value does not matter</td>
</tr>
<tr>
<td></td>
<td>local.counter: register</td>
</tr>
</tbody>
</table>

```
1  local.go := go
2  local.counter := fetch-and-increment (counter)
3  if local.counter + 1 = n then
4    counter := 0
5    go := 1 - go
6  else await(local.go ≠ go)
```

**Pros/Cons?**
Simple Barrier Using an Atomic Counter

```
shared  counter: fetch and increment reg. – {0,...n}, initially = 0
go: atomic bit, initial value does not matter
local  local.go: a bit, initial value does not matter
      local.counter: register

1  local.go := go
2  local.counter := fetch-and-increment (counter)
3  if local.counter + 1 = n then
4      counter := 0
5      go := 1 - go
6  else await(local.go ≠ go)
```

Pros/Cons?
Simple Barrier Using an Atomic Counter

<table>
<thead>
<tr>
<th>shared</th>
<th>counter: fetch and increment reg. – {0,..n}, initially = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>go: atomic bit, initial value does not matter</td>
</tr>
<tr>
<td>local</td>
<td>local.go: a bit, initial value does not matter</td>
</tr>
<tr>
<td></td>
<td>local.counter: register</td>
</tr>
</tbody>
</table>

1. local.go := go
2. local.counter := fetch-and-increment (counter)
3. if local.counter + 1 = n then
   4. counter := 0
   5. go := 1 - go
   6. else await(local.go ≠ go)

Pros/Cons?

- There is high memory contention on go bit
Simple Barrier Using an Atomic Counter

shared
counter: fetch and increment reg. – {0,..n}, initially = 0
go: atomic bit, initial value does not matter

local
local.go: a bit, initial value does not matter
local.counter: register

1 local.go := go
2 local.counter := fetch-and-increment (counter)
3 if local.counter + 1 = n then
4    counter := 0
5    go := 1 - go
6 else await(local.go ≠ go)

Pros/Cons?

- There is high memory contention on go bit
- Reducing the contention:
  - Replace the go bit with n bits: go[1],...,go[n]
  - Process \( p_i \) may spin only on the bit go[\( i \)]
A Local Spinning Counter Barrier
Program of a Thread i

<table>
<thead>
<tr>
<th>shared</th>
<th>counter: fetch and increment reg. – {0,..n}, initially = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>go[1..n]: array of atomic bits, initial values are immaterial</td>
</tr>
<tr>
<td>local</td>
<td>local.go: a bit, initial value is immaterial</td>
</tr>
<tr>
<td></td>
<td>local.counter: register</td>
</tr>
</tbody>
</table>
A Local Spinning Counter Barrier
Program of a Thread $i$

**Shared**
- `counter`: fetch and increment reg. – {0,..n}, initially = 0
- `go[1..n]`: array of atomic bits, initial values are immaterial

**Local**
- `local.go`: a bit, initial value is immaterial
- `local.counter`: register

1. `local.go := go[i]`
2. `local.counter := fetch-and-increment (counter)`
3. **if** `local.counter + 1 = n` **then**
   4. `counter := 0`
   5. **for** $j=1$ to $n$ { `go[j] := 1 - go[j]` }
6. **else** `await(local.go ≠ go[i])`
A Local Spinning Counter Barrier
Example Run for n=3 Threads

1. local.go := go[i]
2. local.counter := fetch-and-increment (counter)
3. if local.counter + 1 = n then
   counter := 0
4. for j=1 to n { go[j] := 1 – go[j] }
5. else await(local.go ≠ go[i])
A Local Spinning Counter Barrier
Example Run for n=3 Threads

1. local.go := go[i]
2. local.counter := fetch-and-increment (counter)
3. if local.counter + 1 = n then
   counter := 0
4. for j=1 to n { go[j] := 1 – go[j] }
5. else await(local.go ≠ go[i])
A Local Spinning Counter Barrier
Example Run for n=3 Threads

1. local.go := go[i]
2. local.counter := fetch-and-increment (counter)
3. if local.counter + 1 = n then
   counter := 0
4. for j=1 to n { go[j] := 1 – go[j] }
5. else await(local.go ≠ go[i])
A Local Spinning Counter Barrier
Example Run for n=3 Threads

```
1. local.go := go[i]
2. local.counter := fetch-and-increment (counter)
3. if local.counter + 1 = n then
   counter := 0
4. for j=1 to n { go[j] := 1 – go[j] }
5. else await(local.go ≠ go[i])
```
A Local Spinning Counter Barrier
Example Run for n=3 Threads

1. local.go := go[i]
2. local.counter := fetch-and-increment (counter)
3. if local.counter + 1 = n then
   counter := 0
4. for j=1 to n { go[j] := 1 – go[j] }
5. else await(local.go ≠ go[i])
A Local Spinning Counter Barrier
Example Run for n=3 Threads

1. local.go := go[i]
2. local.counter := fetch-and-increment (counter)
3. if local.counter + 1 = n then
4.    counter := 0
5.    for j=1 to n { go[j] := 1 – go[j] }
6. else await(local.go ≠ go[i])
A Local Spinning Counter Barrier
Example Run for n=3 Threads

```plaintext
1  local.go := go[i]
2  local.counter := fetch-and-increment (counter)
3  if local.counter + 1 = n then
4    counter := 0
5  for j=1 to n { go[j] := 1 – go[j] }
6  else await(local.go ≠ go[i])
```
A Local Spinning Counter Barrier
Example Run for n=3 Threads

1. local.go := go[i]
2. local.counter := fetch-and-increment (counter)
3. if local.counter + 1 = n then
4.   counter := 0
5. for j=1 to n { go[j] := 1 – go[j] }
6. else await(local.go ≠ go[i])

<table>
<thead>
<tr>
<th>counter</th>
<th>go</th>
<th>SM</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>loc.go</th>
<th>loc.counter</th>
</tr>
</thead>
</table>
| 0      | 0           | P1

<table>
<thead>
<tr>
<th>loc.go</th>
<th>loc.counter</th>
</tr>
</thead>
</table>
| ?      | ?           | P2

<table>
<thead>
<tr>
<th>loc.go</th>
<th>loc.counter</th>
</tr>
</thead>
</table>
| ?      | ?           | P3
A Local Spinning Counter Barrier
Example Run for n=3 Threads

1. local.go := go[i]
2. local.counter := fetch-and-increment
3. if local.counter + 1 = n then
4.     counter := 0
5.     for j=1 to n { go[j] := 1 – go[j] }
6. else await(local.go ≠ go[i])

counter 1

sm
go 0 0 0

loc.go 0
counter 0

loc.go ?
counter ?

loc.go ?
counter ?
A Local Spinning Counter Barrier
Example Run for n=3 Threads

```plaintext
1 local.go := go[i]
2 local.counter := fetch-and-increment (counter)
3 if local.counter + 1 = n then
4   counter := 0
5 for j=1 to n { go[j] := 1 - go[j] }
6 else await(local.go ≠ go[i])
```
A Local Spinning Counter Barrier
Example Run for n=3 Threads

1. local.go := go[i]
2. local.counter := fetch-and-increment (counter)
3. if local.counter + 1 = n then
   4. counter := 0
5. for j=1 to n { go[j] := 1 – go[j] }
6. else await(local.go ≠ go[i])

P1 Busy wait
A Local Spinning Counter Barrier
Example Run for n=3 Threads

counter = 1

loc.go = 0
loc.counter = 0

P1

loc.go = ?
loc.counter = ?

P2

loc.go = ?
loc.counter = ?

P3

g0 = 0
g1 = 0
g2 = 0

SM

1. local.go := go[i]
2. local.counter := fetch-and-increment (counter)
3. if local.counter + 1 = n then
4.    counter := 0
5. for j=1 to n { go[j] := 1 – go[j] }
6. else await(local.go ≠ go[i])
A Local Spinning Counter Barrier
Example Run for n=3 Threads

counter = 1

go = 0 0 0

loc.go | loc.counter
--- | ---
P1 | 0 0
P2 | 0 ?
P3 | ? ?

1 local.go := go[i]
2 local.counter := fetch-and-increment (counter)
3 if local.counter + 1 = n then
4 counter := 0
5 for j=1 to n { go[j] := 1 - go[j] }
6 else await(local.go ≠ go[i])

P1 Busy wait
A Local Spinning Counter Barrier
Example Run for n=3 Threads

```plaintext
counter 1
go 0 0 0

loc.go 0 loc.go 0
loc.counter 0 ?
loc.counter ?

1 local.go := go[i]
2 local.counter := fetch-and-increment (counter)
3 if local.counter + 1 = n then
4 counter := 0
5 for j=1 to n { go[j] := 1 - go[j] }
6 else await(local.go ≠ go[i])

P1 Busy wait

P1
P2
P3
```

- **Counter**: The counter variable is used to keep track of the progress of the barrier synchronization. Initially set to 1, it is decremented after each thread completes its task.
- **Go**: Represents the status of each thread. Initially all threads are in the `go` state.
- **Loc.go**: Indicates the local go state of each thread.
- **Loc.counter**: Represents the local counter value for each thread.

The diagram illustrates the progression of the example run, with each thread executing the provided code snippet. The green region highlights where P1 is busy waiting due to the asynchronous nature of the barrier synchronization.
A Local Spinning Counter Barrier
Example Run for n=3 Threads

```
1 local.go := go[i]
2 local.counter := fetch-and-increment (counter)
3 if local.counter + 1 = n then
4    counter := 0
5 for j=1 to n { go[j] := 1 – go[j] }
6 else await(local.go ≠ go[i])
```
A Local Spinning Counter Barrier
Example Run for n=3 Threads

```
1  local.go := go[i]
2  local.counter := fetch-and-increment (counter)
3  if local.counter + 1 = n then
4      counter := 0
5  for j=1 to n { go[j] := 1 – go[j] }
6  else await(local.go ≠ go[i])
```

P1 Busy wait
A Local Spinning Counter Barrier
Example Run for n=3 Threads

1. `local.go := go[i]`
2. `local.counter := fetch-and-increment (counter)`
3. `if local.counter + 1 = n then`
4. `counter := 0`
5. `for j=1 to n { go[j] := 1 – go[j] }`
6. `else await(local.go ≠ go[i])`

P1 Busy wait
A Local Spinning Counter Barrier
Example Run for n=3 Threads

1. local.go := go[i]
2. local.counter := fetch-and-increment()
3. if local.counter + 1 = n then
   counter := 0
4. for j=1 to n { go[j] := 1 - go[j] }
5. else await(local.go ≠ go[i])

P1 Busy wait
A Local Spinning Counter Barrier
Example Run for n=3 Threads

1. `local.go := go[i]`
2. `local.counter := fetch-and-increment (counter)`
3. `if local.counter + 1 = n then`
4. `counter := 0`
5. `for j=1 to n { go[j] := 1 - go[j] }`
6. `else await(local.go ≠ go[i])`

P1, P2 Busy wait
A Local Spinning Counter Barrier
Example Run for n=3 Threads

1. local.go := go[i]
2. local.counter := fetch-and-increment (counter)
3. if local.counter + 1 = n then
4.   counter := 0
5. for j=1 to n { go[j] := 1 – go[j] }
6. else await(local.go ≠ go[i])

P1, P2 Busy wait

P1
P2
P3
A Local Spinning Counter Barrier
Example Run for n=3 Threads

```plaintext
1. local.go := go[i]
2. local.counter := fetch-and-increment (counter)
3. if local.counter + 1 = n then
   counter := 0
4. for j=1 to n { go[j] := 1 – go[j] }
5. else await (local.go ≠ go[i])
6. P1,P2 Busy wait
```
A Local Spinning Counter Barrier

Example Run for n=3 Threads

1. `local.go := go[i]`
2. `local.counter := fetch-and-increment (counter)`
3. `if local.counter + 1 = n then`
4. `counter := 0`
5. `for j=1 to n { go[j] := 1 – go[j] }`
6. `else await(local.go ≠ go[i])`

P1,P2 Busy wait
A Local Spinning Counter Barrier
Example Run for n=3 Threads

1. local.go := go[i]
2. local.counter := fetch-and-increment (counter)
3. if local.counter + 1 = n then
4.     counter := 0
5.     for j=1 to n { go[j] := 1 − go[j] }
6. else await (local.go ≠ go[i])

P1, P2 Busy wait
A Local Spinning Counter Barrier
Example Run for n=3 Threads

1. local.go := go[i]
2. local.counter := fetch-and-increment (counter)
3. if local.counter + 1 = n then
4.     counter := 0
5.     for j=1 to n { go[j] := 1 – go[j] }
6. else await(local.go ≠ go[i])
A Local Spinning Counter Barrier
Example Run for n=3 Threads

```
1 local.go := go[i]
2 local.counter := fetch-and-increment (counter)
3 if local.counter + 1 = n then
4     counter := 0
5   for j=1 to n { go[j] := 1 - go[j] }
6 else await(local.go ≠ go[i])
```

P1, P2 Busy wait
A Local Spinning Counter Barrier
Example Run for n=3 Threads

1. `local.go := go[i]`
2. `local.counter := fetch-and-increment(counter)`
3. **if** `local.counter + 1 = n` **then**
4. `counter := 0`
5. **for** `j=1 to n` **do** `{ `go[j] := 1 – go[j] `}`
6. **else** `await(local.go ≠ go[i])`

- **P1**
  - `local.go = 0`
  - `local.counter = 0`

- **P2**
  - `local.go = 0`
  - `local.counter = 1`

- **P3**
  - `local.go = 0`
  - `local.counter = 2`

- **SM**
  - `counter = 3`
  - `go = 0 0 0`

**P1, P2 Busy wait**
A Local Spinning Counter Barrier
Example Run for n=3 Threads

1. local.go := go[i]
2. local.counter := fetch-and-increment (counter)
3. if local.counter + 1 = n then
   4. counter := 0
   5. for j=1 to n { go[j] := 1 - go[j] }
5. else await(local.go ≠ go[i])
6. P1,P2 Busy wait
A Local Spinning Counter Barrier
Example Run for n=3 Threads

1. local.go := go[i]
2. local.counter := fetch-and-increment (counter)
3. if local.counter + 1 = n then
4.   counter := 0
5.   for j=1 to n { go[j] := 1 - go[j] }
6. else await( local.go ≠ go[i] )

P1, P2 Busy wait
A Local Spinning Counter Barrier
Example Run for n=3 Threads

```
1 local.go := go[i]
2 local.counter := fetch-and-increment (counter)
3 if local.counter + 1 = n then
4    counter := 0
5  for j=1 to n { go[j] := 1 – go[j] }
6 else await(local.go ≠ go[i])
```

P1,P2 Busy wait
A Local Spinning Counter Barrier
Example Run for n=3 Threads

1. `local.go := go[i]`
2. `local.counter := fetch-and-increment (counter)`
3. `if local.counter + 1 = n then`
4. `counter := 0`
5. `for j=1 to n { go[j] := 1 – go[j] }`
6. `else await(local.go ≠ go[i])`

P1, P2 Busy wait
A Local Spinning Counter Barrier
Example Run for n=3 Threads

1. local.go := go[i]
2. local.counter := fetch-and-increment (counter)
3. if local.counter + 1 = n then
4.     counter := 0
5. for j=1 to n { go[j] := 1 – go[j] }
6. else await(local.go ≠ go[i])

P1, P2 Busy wait

P1

P2

P3
A Local Spinning Counter Barrier
Example Run for n=3 Threads

1. `local.go := go[i]`
2. `local.counter := fetch-and-increment (counter)`
3. `if local.counter + 1 = n then`
4. `counter := 0`
5. `for j=1 to n { go[j] := 1 – go[j] }`
6. `else await(local.go ≠ go[i])`

```
local.go 0 0 0
local.counter 0 1 2
```

```
counter 0
```

```
go 1 1 1
```

P1, P2 Busy wait
A Local Spinning Counter Barrier
Example Run for n=3 Threads

1. local.go := go[i]
2. local.counter := fetch-and-increment (counter)
3. if local.counter + 1 = n then
4.     counter := 0
5. for j=1 to n { go[j] := 1 - go[j] }
6. else await(local.go ≠ go[i])
A Local Spinning Counter Barrier
Example Run for n=3 Threads

Pros/Cons?
Does this actually reduce contention?
Comparison of counter-based Barriers

Simple Barrier

• Pros:

• Cons:

Simple Barrier with go array

• Pros:

• Cons:
### Simple Barrier

#### Pros:
- Very Simple
- Shared memory: $O(\log n)$ bits
- Takes $O(1)$ until last waiting $p$ is awaken

#### Cons:
- High contention on the go bit
- Contention on the counter register (*)

### Simple Barrier with go array

#### Pros:
- Low contention on the go array
- In some models:
  - spinning is done on local memory
  - remote mem. ref.: $O(1)$

#### Cons:
- Shared memory: $O(n)$
- Still contention on the counter register (*)
- Takes $O(n)$ until last waiting $p$ is awaken
Tree Barriers
A Tree-based Barrier
A Tree-based Barrier

- Threads are organized in a binary tree
- Each node is owned by a predetermined thread
A Tree-based Barrier

• Threads are organized in a binary tree
• Each node is owned by a predetermined thread
• Each thread waits until its 2 children arrive
  • combines results
  • passes them on to its parent
A Tree-based Barrier

- Threads are organized in a binary tree
- Each node is owned by a predetermined thread
- Each thread waits until its 2 children arrive
  - combines results
  - passes them on to its parent
- Root learns that its 2 children have arrived → tells children they can go
- The signal propagates down the tree until all the threads get the message
A Tree-based Barrier: indexing
A Tree-based Barrier: indexing

Step 1: label numerically with depth-first traversal
A Tree-based Barrier: indexing

Step 1: label numerically with depth-first traversal
A Tree-based Barrier: indexing

Assume $n = 2^k - 1$

Step 1: label numerically with depth-first traversal
A Tree-based Barrier: indexing

Step 1: label numerically with depth-first traversal
A Tree-based Barrier: indexing

Step 1: label numerically with depth-first traversal

- $i$
- $2i$
- $2i + 1$

```
arrive

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

go

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
```
A Tree-based Barrier: indexing

Step 1: label numerically with depth-first traversal

Indexing starts from 2
Root $\rightarrow$ 1, doesn’t need wait objects
A Tree-based Barrier
program of thread i

| shared | arrive[2..n]: array of atomic bits, initial values = 0 |
|        | go[2..n]: array of atomic bits, initial values = 0 |

1  if i=1 then \hspace{1cm} // root
3    await(arrive[3] = 1); arrive[3] := 0
5  else if i ≤ (n-1)/2 then \hspace{1cm} // internal node
6    await(arrive[2i] = 1); arrive[2i] := 0
7    await(arrive[2i+1] = 1); arrive[2i+1] := 0
8    arrive[i] := 1
9    await(go[i] = 1); go[i] := 0
10   go[2i] = 1; go[2i+1] := 1
11  else \hspace{1cm} // leaf
12     arrive[i] := 1
13    await(go[i] = 1); go[i] := 0 fi
14  fi
A Tree-based Barrier program of thread $i$

shared arrivals[2..n]: array of atomic bits, initial values = 0
shared go[2..n]: array of atomic bits, initial values = 0

1  if $i=1$ then // root
5  else if $i \leq (n-1)/2$ then // internal node
6      await(arrivals[$2i$] = 1); arrivals[$2i$] := 0
7      await(arrivals[$2i+1$] = 1); arrivals[$2i+1$] := 0
8      arrivals[$i$] := 1
9      await(go[$i$] = 1); go[$i$] := 0
10     go[$2i$] = 1; go[$2i+1$] := 1
11    else // leaf
12       arrivals[$i$] := 1
13       await(go[$i$] = 1); go[$i$] := 0 fi
14    fi
A Tree-based Barrier
program of thread $i$

shared
- arrive[2..n]: array of atomic bits, initial values = 0
- go[2..n]: array of atomic bits, initial values = 0

1    if $i=1$ then
     // root
3       await(arrive[3] = 1); arrive[3] := 0
5    else if $i \leq (n-1)/2$ then
     // internal node
6       await(arrive[2i] = 1); arrive[2i] := 0
7       await(arrive[2i+1] = 1); arrive[2i+1] := 0
8       arrive[i] := 1
9       await(go[i] = 1); go[i] := 0
10      go[2i] = 1; go[2i+1] := 1
11    else
     // leaf
12       arrive[i] := 1
13       await(go[i] = 1); go[i] := 0 fi
14    fi
A Tree-based Barrier program of thread i

**shared**
- arrive[2..n]: array of atomic bits, initial values = 0
- go[2..n]: array of atomic bits, initial values = 0

```
1   if i=1 then // root
3     await(arrive[3] = 1); arrive[3] := 0
5   else if i ≤ (n-1)/2 then // internal node
6     await(arrive[2i] = 1); arrive[2i] := 0
7     await(arrive[2i+1] = 1); arrive[2i+1] := 0
8     arrive[i] := 1
9     await(go[i] = 1); go[i] := 0
10    go[2i] = 1; go[2i+1] := 1
11   else // leaf
12      arrive[i] := 1
13      await(go[i] = 1); go[i] := 0 fi
14  fi
```

**Root:**
- Wait for arriving children
- Tell children to go

**Internal:**
- Wait for arriving children
- Tell parent about it
- Wait for parent go signal
- Tell children to go

**Leaf:**
- Wait for arriving children
- Tell children to go
A Tree-based Barrier
program of thread i

| shared | arrive[2..n]: array of atomic bits, initial values = 0 |
|        | go[2..n]: array of atomic bits, initial values = 0 |

1. if i=1 then

5. else if i ≤ (n-1)/2 then  // internal node
   6. await(arrive[2i] = 1); arrive[2i] := 0
   7. await(arrive[2i+1] = 1); arrive[2i+1] := 0
   8. arrive[i] := 1
   9. await(go[i] = 1); go[i] := 0
   10. go[2i] = 1; go[2i+1] := 1

11. else  // leaf
    12. arrive[i] := 1
    13. await(go[i] = 1); go[i] := 0 fi

14. fi

Root:
- Wait for arriving children
- Tell children to go

Internal:
- Wait for arriving children
- Tell parent about it
- Wait for parent go signal
- Tell children to go

Child:
- arrive
- Wait for parent go signal
A Tree-based Barrier
Example Run for n=7 threads

```
shared    arrive[2..n]: array of atomic bits, initial values = 0
          go[2..n]: array of atomic bits, initial values = 0

1
   if i=1 then  // root
   await(arrive[3] = 1); arrive[3] := 0
   else if i ≤ (n-1)/2 then  // internal node
   await(arrive[2*i] = 1); arrive[2*i] := 0
   await(arrive[2*i+1] = 1); arrive[2*i+1] := 0
   arrive[i] := 1
   await(go[i] = 1); go[i] := 0
   go[2*i] := 1; go[2*i+1] := 1
   else  // leaf
   else
   await(arrive[i] = 1)
   await(go[i] = 1); go[i] := 0
   fi
fi
```
A Tree-based Barrier
Example Run for n=7 threads

```
1

2 3

4 5

6 7

shared
arrive[2..n]: array of atomic bits, initial values = 0

if i=1 then // root

else if i = (n-1)/2 then // internal node
  await[arrival[2i] = 1]; arrive[2i] := 0
  await[arrival[2i+1] = 1]; arrive[2i+1] := 0
  arrive[i] := 1
  await[go[i] = 1]; go[i] := 0
  go[2i] := 1; go[2i+1] := 1

else // leaf
  arrive[i] := 1
  await[go[i] = 1]; go[i] := 0
fi
```
A Tree-based Barrier
Example Run for n=7 threads

arrive[2]=1?
A Tree-based Barrier
Example Run for n=7 threads

```java
shared
arrive[2..n]: array of atomic bits, initial values = 0

if i=1 then // root
    await[arrive[2]] = 1; arrive[2] := 0
    await[arrive[3]] = 1; arrive[3] := 0
else if i = (n-1)/2 then // internal node
    await[arrive[2i]] = 1; arrive[2i] := 0
    await[arrive[2i+1]] = 1; arrive[2i+1] := 0
    arrive[i] := 1
    await[go[i]] = 1; go[i] := 0
    go[2i] := 1; go[2i+1] := 1
else // leaf
    arrive[i] := 1
    await[go[i]] = 1; go[i] := 0
fi
```

```
arrive
2 3 4 5 6 7

go
2 3 4 5 6 7
```
A Tree-based Barrier
Example Run for n=7 threads

Waiting for p₄ to arrive

```
shared
arrive[2..n]: array of atomic bits, initial values = 0

arrive[2..n]: array of atomic bits, initial values = 0

if i=1 then // root
else if i ≤ (n-1)/2 then // internal node
    await[arrive[2i] = 1]; arrive[2i] := 0
    await[arrive[2i+1] = 1]; arrive[2i+1] := 0
    arrive[i] := 1
    await[go[i] = 1]; go[i] := 0
    go[2i] := 1; go[2i+1] := 1
else if i < n then // leaf
    arrive[i] := 1
    await[go[i] = 1]; go[i] := 0
fi
```
A Tree-based Barrier
Example Run for n=7 threads

```plaintext
shared
arrive[2..n]: array of atomic bits, initial values = 0
go[2..n]: array of atomic bits, initial values = 0

1 if i=1 then // root
  await(arrive[3] = 1); arrive[3] := 0
5 else if i = (n-1)/2 then // internal node
  await(arrive[2i] = 1); arrive[2i] := 0
  await(arrive[2i+1] = 1); arrive[2i+1] := 0
  arrive[i] := 1
  await(go[2i] = 1); go[2i] := 0
  go[2i] := 1; go[2i+1] := 1
11 else // leaf
  arrive[i] := 1
13 await(go[2i] = 1); go[2i] := 0 fi
14 fi
```

arrive

```
2 3 4 5 6 7
```

go

```
2 3 4 5 6 7
```
A Tree-based Barrier
Example Run for n=7 threads

```
shared arrive[2..n]: array of atomic bits, initial values = 0
go[2..n]: array of atomic bits, initial values = 0

1 if i=1 then // root
5 else if i <= (n-1)/2 then // internal node
    await[arrival[2i] = 1]; arrive[2i] := 0
    await[arrival[2i+1] = 1]; arrive[2i+1] := 0
    arrive[i] := 1
    await[go[i] = 1]; go[i] := 0
10    go[2i] := 1; go[2i+1] := 1
11 else // leaf
12    await[i] := 1
13    await[go[i] = 1]; go[i] := 0 fi
14 fi
```

arrive

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>1</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

go

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
A Tree-based Barrier
Example Run for n=7 threads

Waiting for go[5]
A Tree-based Barrier
Example Run for n=7 threads

```
shared

if i=1 then // root
    await[arrive[2]] = 1; arrive[2] := 0
    await[arrive[3]] = 1; arrive[3] := 0
else if i = (n-1)/2 then // internal node
    if i = 0 then
        await[arrive[2i]] = 1; arrive[2i] := 0
        await[arrive[2i+1]] = 1; arrive[2i+1] := 0
        arrive[i] := 1
        await[go[i]] = 1; go[i] := 0
        go[2i] := 1; go[2i+1] := 1
    else
        arrive[i] := 1
        await[go[i]] = 1; go[i] := 0
    fi
fi
```

<table>
<thead>
<tr>
<th>arrive</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>go</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A Tree-based Barrier
Example Run for n=7 threads

```java
shared
arrive[2..n]: array of atomic bits, initial values = 0
go[2..n]: array of atomic bits, initial values = 0

1

if i+1 then
  // root
  await(arrive[i] = 1); arrive[i] := 0
  await(arrive[i+1] = 1); arrive[i+1] := 0
  go[i+1] := 1; go[i] := 1
else if i ≤ (n-1)/2 then
  // internal node
  await(arrive[i] = 1); arrive[i] := 0
  await(arrive[i+1] = 1); arrive[i+1] := 0
  arrive[i+1] := 1
  await(go[i] = 1); go[i] := 0
  go[i+1] := 1; go[i+1] := 1
else
  // leaf
  arrive[i] := 1
  await(go[i] = 1); go[i] := 0 end
}fi
```

<table>
<thead>
<tr>
<th>arrive</th>
<th></th>
<th></th>
<th>1</th>
<th>1</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>go</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
A Tree-based Barrier
Example Run for n=7 threads

shared

arrive[2..n]: array of atomic bits, initial values = 0
go[2..n]: array of atomic bits, initial values = 0

1 if i=1 then // root
5 else if i ≤ (n-1)/2 then // internal node
6 await [arrive[2i] = 1]; arrive[2i] := 0
7 await [arrive[2i+1] = 1]; arrive[2i+1] := 0
8 arrive[i] := 1
9 await [go[0] = 1]; go[0] := 0
10 go[2i] := 1; go[2i+1] := 1
11 else // leaf
12 if i = 1
13 await [go[0] = 1]; go[0] := 0
14 fi

arrive

| 2 | 3 | 4 | 5 | 6 | 7 |

go

| 2 | 3 | 4 | 5 | 6 | 7 |
A Tree-based Barrier
Example Run for n=7 threads

shared
arrive[2..n]: array of atomic bits, initial values := 0

go[2..n]: array of atomic bits, initial values := 0

1 if i=1 then // root
5 else if i < (n-1)/2 then // internal node
6 await(arrive[2i] := 1; arrive[2i] := 0
7 await(arrive[2i+1] := 1; arrive[2i+1] := 0
8 arrive[i] := 1
9 await(go[i] := 1; go[i] := 0
10 go[2i] := 1; go[2i+1] := 1
11 else // leaf
12 arrive[i] := 1
13 await(go[i] := 1; go[i] := 0)
14 fi

arrive

<table>
<thead>
<tr>
<th></th>
<th></th>
<th>1</th>
<th>1</th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>

go

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td></td>
</tr>
</tbody>
</table>
A Tree-based Barrier
Example Run for n=7 threads

P2 zeros
arrive[4,5]

shared
arrive[2..n]: array of atomic bits, initial values = 0
go[2..n]: array of atomic bits, initial values = 0

if i=1 then // root
else if i ≤ (n-1)/2 then // internal node
  await(arrive[2i] := 1; arrive[2i] := 0
  await(arrive[2i+1] := 1; arrive[2i+1] := 0
  arrive[i] := 1
  await(go[i] := 1; go[i] := 0
  go[2i] := 1; go[2i+1] := 1
else // leaf
  arrive[i] := 1
  await(go[i] := 1; go[i] := 0
fi

arrive

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

go

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A Tree-based Barrier
Example Run for n=7 threads

```
shared arrive[2..n]: array of atomic bits, initial values = 0
go[2..n]: array of atomic bits, initial values = 0

if i=1 then // root
  await(arrive[3] = 1); arrive[3] := 0
else if i = (n-1)/2 then // internal node
  await(arrive[2i] = 1); arrive[2i] := 0
  await(arrive[2i+1] = 1); arrive[2i+1] := 0
  arrive[i] := 1
  await(go[i] = 1); go[i] := 0
  go[2i] := 1; go[2i+1] := 1
else // leaf
  arrive[i] := 1
  await(go[i] = 1); go[i] := 0 fi
fi
```
A Tree-based Barrier
Example Run for n=7 threads

```
if i=1 then // root
    atomical variable
    go[i] := 1; go[i] := 0
else if i = (n-1)/2 then // internal node
    atomical variable
    go[i] := 1; go[i] := 0
    atomical variable
    atomical variable
else // leaf
    atomical variable
    atomical variable
fi
```
A Tree-based Barrier
Example Run for $n=7$ threads

```
shared arr[2..n]: array of atomic bits, initial values = 0
arr[2..n]: array of atomic bits, initial values = 0

1 if i=1 then // root
2     wait[arr[2]] = 1; arr[2] := 0
3     wait[arr[i]]= 1; arr[i] := 0
5 else if i = (n-1)/2 then // internal node
6     wait[arr[2]] = 1; arr[2] := 0
7     wait[arr[i+1]]= 1; arr[i+1] := 0
8     arr[i] := 1
9     wait[go[i]] = 1; go[i] := 0
10    go[2] = 1; go[2+1] := 1
11 else // leaf
12    arr[i] := 1
13    wait[go[i]] = 1; go[i] := 0
14 fi
```

Waiting for $go[2]$
A Tree-based Barrier
Example Run for n=7 threads

```c
shared
```

```
arrive[2:n]; array of atomic bits, initial values = 0
go[2:n]; array of atomic bits, initial values = 0
```

```c
1 if i=1 then // root
3    await(arrive[3] = 1); arrive[3] := 0
5  else if i = (n-1)/2 then // internal node
6      await(arrive[2i] = 1); arrive[2i] := 0
7      await(arrive[2i+1] = 1); arrive[2i+1] := 0
8      arrive[i] := 1
9      await(go[i] = 1); go[i] := 0
10     go[2i] := 1; go[2i+1] := 1
11  else // leaf
12    arrive[i] := 1
13    await(go[i] = 1); go[i] := 0 fi
14  fi
```
A Tree-based Barrier
Example Run for n=7 threads

shared
arrive[2..n]: array of atomic bits, initial values = 0
go[2..n]: array of atomic bits, initial values = 0

1
if i=1 then // root
2
3
4
5
else if i = (n-1)/2 then // internal node
6
await(arrive[2i] = 1; arrive[2i] := 0
7
await(arrive[2i+1] = 1; arrive[2i+1] := 0
8
arrive[i] := 1
9
await(go[i] = 1; go[i] := 0
10
go[2i] := 1; go[2i+1] := 1
11
else // leaf
12
arrive[i] := 1
13
await(go[i] = 1; go[i] := 0 fi
14
fi

arrive

| 1 | 0 | 0 | 1 |
---|---|---|---|

go

| 2 | 3 | 4 | 5 | 6 | 7 |
A Tree-based Barrier
Example Run for n=7 threads

```
shared  arrive[2..n]: array of atomic bits, initial values = 0
        go[2..n]: array of atomic bits, initial values = 0

if i=1 then // root
  await(arrive[3] = 1); arrive[3] := 0
else if i = (n-1)/2 then // internal node
  await(arrive[2*i] = 1); arrive[2*i] := 0
  await(arrive[2*i+1] = 1); arrive[2*i+1] := 0
  arrive[i] := 1
  await(go[i] = 1); go[i] := 0
  go[2*i] := 1; go[2*i+1] := 1
else // leaf
  if i = 1
    await(arrive[i] = 1)
  else
    await(go[i] = 1); go[i] := 0 fi
fi
```

**Arrive**

```
|   | 1 | 0 | 0 | 1 |
```

**Go**

```
|   | 2 | 3 | 4 | 5 | 6 | 7 |
```
A Tree-based Barrier
Example Run for n=7 threads

```
shared
arrive[2..n]: array of atomic bits, initial values = 0
go[2..n]: array of atomic bits, initial values = 0

if i=1 then // root
  await(arrive[3] = 1); arrive[3] := 0
else if i = (n-1)/2 then // internal node
  await(arrive[2i] = 1); arrive[2i] := 0
  await(arrive[2i+1] = 1); arrive[2i+1] := 0
  arrive[i] := 1
  await(go[2i] = 1); go[2i] := 0
  go[2i] := 1; go[2i+1] := 1
else // leaf
  arrive[i] := 1
  await(go[2i] = 1); go[2i] := 0
fi
```
A Tree-based Barrier
Example Run for n=7 threads

```plaintext
shared arr[2..n]; array of atomic bits, initial values := 0
    go[2..n]; array of atomic bits, initial values := 0

if i=1 then // root
else if i = (n-1)/2 then // internal node
    wait(arr[2*i] := 1; arr[2*i] := 0
    wait(arr[2*i+1] := 1; arr[2*i+1] := 0
    arr[i] := 1
    wait(go[i] := 1; go[i] := 0
    go[2*i] := 1; go[2*i+1] := 1
else // leaf
    arr[i] := 1
    wait(go[i] := 1; go[i] := 0
fi
```

<table>
<thead>
<tr>
<th>Arrive</th>
<th>Go</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

```

2 3 4 5 6 7
```

```
A Tree-based Barrier
Example Run for n=7 threads

```
if i=1 then // root
  await[arrival[i]] = 1; arrival[i] := 0
3 await[arrival[i]] = 1; arrival[i] := 0
4 go[i] = 1; go[i] := 1
else if i ≺ (n-1)/2 then // internal node
  await[arrival[i]] = 1; arrival[i] := 0
7 await[arrival[i+1]] = 1; arrival[i+1] := 0
8 arrival[i] := 1
9 await[go[i]] = 1; go[i] := 0
10 go[i+1] = 1; go[i+1] := 1
else // leaf
  arrival[i] := 1
13 await[go[i]] = 1; go[i] := 0 fi
14 fi
```

<table>
<thead>
<tr>
<th>arrive</th>
<th>1</th>
<th>0</th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>go</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td>7</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A Tree-based Barrier
Example Run for \( n = 7 \) threads

```c
if \( i = 1 \) then // root
    await(arrive[2]; arrive[2] := 0
    await(arrive[3]; arrive[3] := 0
else if \( i < (n-1)/2 \) then // internal node
    await(arrive[2]; arrive[2] := 0
    await(arrive[2i+1]; arrive[2i+1] := 0
    arrive[i] := 1
    await(go[i]; go[i] := 0
    go[2i] := 1; go[2i+1] := 1
else // leaf
    arrive[i] := 1
    await(go[i]; go[i] := 0
fi
```
A Tree-based Barrier
Example Run for n=7 threads

```
shared arrive[2..n]; array of atomic bits, initial values = 0
go[2..n]; array of atomic bits, initial values = 0

if i=1 then // root
  await(arrive[3] = 1); arrive[3] := 0
else if i = (n+1)/2 then // internal node
  await(arrive[2i] = 1); arrive[2i] := 0
  await(arrive[2i+1] = 1); arrive[2i+1] := 0
  arrive[i] := 1
  await(go[i] = 1); go[i] := 0
  go[2i] := 1; go[2i+1] := 1
else // leaf
  arrive[i] := 1
  await(go[i] = 1); go[i] := 0 fi
fi
```
A Tree-based Barrier
Example Run for n=7 threads

```
shared    arrive[2..n]: array of atomic bits, initial values := 0
          go[2..n]: array of atomic bits, initial values := 0

1
   if i=1 then // root
   await[|i|]; arrive[i] := 0
   2
   await[|i|+1]; arrive[i+1] := 0
   3
   go[i] := 1; go[i+1] := 1
   4
   else if i < (n-1)/2 then // internal node
   await[|2i|+1]; arrive[2i+1] := 0
   5
   await[|2i+1|+1]; arrive[2i+1+1] := 0
   6
   arrive[i] := 1
   7
   await[|0i|]; go[i] := 0
   8
   go[2i] := 1; go[2i+1] := 1
   9
   else // leaf
   arrive[i] := 1
   10
   await[|0i|]; go[i] := 0 fi
   11
fi
```

```
A Tree-based Barrier
Example Run for n=7 threads

Waiting for p₃ to arrive

```
shared  arrive[2:n]; array of atomic bits, initial values = 0
        go[2:n]; array of atomic bits, initial values = 0

1 if i=1 then // root
2 else if i = (n-1)/2 then // internal node
  await[ arrive[2*i] = 1]; arrive[2*i] := 0
  await[ arrive[2*i+1] = 1]; arrive[2*i+1] := 0
  arrive[i] := 1
3   await[ go[0] = 1]; go[0] := 0
4   go[2*i] := 1; go[2*i+1] := 1
5 else if i = (n-1)/2+1 then // leaf
  await[ arrive[i] = 1]
6   await[ go[0] = 1]; go[0] := 0 fi
7 fi
```

```
arrive
0 0 0 1

go
2 3 4 5 6 7
```
A Tree-based Barrier
Example Run for n=7 threads

```
shared arr[2..n]; array of atomic bits, initial values = 0
go[2..n]; array of atomic bits, initial values = 0

1
   if i=1 then // root
   await(arr[i] = 1); arr[i] := 0
   await(arr[i] = 1); arr[i] := 0
   go[i] := 1; go[i] := 1
   else if i < (n-1)/2 then // internal node
   await(arr[2i] = 1); arr[2i] := 0
   await(arr[2i+1] = 1); arr[2i+1] := 0
   arr[i] := 1
   await(go[i] = 1); go[i] := 0
   go[2i] := 1; go[2i+1] := 1
   else // leaf
   arr[i] := 1
   await(go[i] = 1); go[i] := 0

arrive: 0 0 0 1

1
   if i=1 then // root
   await(arr[i] = 1); arr[i] := 0
   await(arr[i] = 1); arr[i] := 0
   go[i] := 1; go[i] := 1
   else if i < (n-1)/2 then // internal node
   await(arr[2i] = 1); arr[2i] := 0
   await(arr[2i+1] = 1); arr[2i+1] := 0
   arr[i] := 1
   await(go[i] = 1); go[i] := 0
   go[2i] := 1; go[2i+1] := 1
   else // leaf
   arr[i] := 1
   await(go[i] = 1); go[i] := 0

arrive: 0 0 0 1
```

```
A Tree-based Barrier
Example Run for n=7 threads

```plaintext
shared   arrive[2..n]; array of atomic bits, initial values := 0
go[2..n]; array of atomic bits, initial values := 0

1
   if i=1 then // root
      await[2]; arrive[2] := 0
      await[3]; arrive[3] := 0
   else if i = (n-1)/2 then // internal node
      await[2]; arrive[2] := 0
      await[2i+1] := 1; arrive[2i+1] := 0
      arrive[i] := 1
      await[2i]; go[2i] := 0
      go[2i] := 1; go[2i+1] := 1
   else if i < (n-1)/2 then // leaf
      arrive[i] := 1
      await[2i]; go[2i] := 0
      wait[2i+1]; go[2i+1] := 0
      fi
   fi
```

<table>
<thead>
<tr>
<th>arrive</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>go</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
A Tree-based Barrier
Example Run for n=7 threads

shared
arrive[2..n]: array of atomic bits, initial values = 0

if i=1 then // root
else if i ≤ (n-1)/2 then // internal node
   await(arrive[2i] = 1; arrive[2i] := 0
   await(arrive[2i+1] = 1; arrive[2i+1] := 0
   arrive[i] := 1
   await(go[i] = 1); go[i] := 0
   go[2i] := 1; go[2i+1] := 1
else
   /leaf
   arrive[i] := 1
   await(go[i] = 1); go[i] := 0 fi
fi

arrive
0 0 0 1 1

go
2 3 4 5 6 7

Waiting for go[7]
A Tree-based Barrier
Example Run for n=7 threads

shared arrive[2..n]: array of atomic bits, initial values = 0
go[2..n]: array of atomic bits, initial values = 0

1

if i=1 then
   // root
   await(arrive[3] = 1); arrive[3] := 0
else if i ≤ (n-1)/2 then
   // internal node
   await(arrive[2i] = 1); arrive[2i] := 0
   await(arrive[2i+1] = 1); arrive[2i+1] := 0
   arrive[i] := 1
   await(go[i] = 1); go[i] := 0
   go[2i] := 1; go[2i+1] := 1
else
   // leaf
   arrive[i] := 1
   await(go[i] = 1); go[i] := 0
fi

arrive
0 0 0 1 1

go
2 3 4 5 6 7
A Tree-based Barrier
Example Run for n=7 threads

shared

arrive[2..n]: array of atomic bits, initial values = 0
go[2..n]: array of atomic bits, initial values = 0

if i=1 then // root
  await(arrive[2]) := 0
  await(arrive[3]) := 0
else if i = (n-1)/2 then // internal node
  await(arrive[2i]) := 0
  await(arrive[2i+1]) := 0
  arrive[i] := 1
  await(go[i]) := 0
  go[2i] := 1; go[2i+1] := 1
else // leaf
  arrive[i] := 1
  await(go[i]) := 0 fi
fi

arrive

0 0 0 1 1

go

2 3 4 5 6 7
A Tree-based Barrier
Example Run for n=7 threads

shared

array[2..n]: array of atomic bits, initial values := 0
go[2..n]: array of atomic bits, initial values := 0

if i=1 then  // root
  await[arrival[2]] := 0
  await[arrival[3]] := 0
else if i ≤ (n-1)/2 then  // internal node
  await[arrival[2i]] := 0
  await[arrival[2i+1]] := 0
  await[i] := 1
  await[go[i]] := 0
  go[2i] := 1; go[2i+1] := 1
else  // leaf
  await[i] := 1
  await[go[i]] := 0
fi

arrive[6..7]

P_3 zeros

0 0 0 0 0 0

2 3 4 5 6 7
A Tree-based Barrier
Example Run for n=7 threads

```
shared
    arrive[2..n]: array of atomic bits, initial values = 0
    go[2..n]: array of atomic bits, initial values = 0

1
   if i=1 then // root
       await(arrive[3] = 1); arrive[3] := 0
   else if i ≤ (n-1)/2 then // internal node
       await(arrive[2i] = 1); arrive[2i] := 0
       await(arrive[2i+1] = 1); arrive[2i+1] := 0
       arrive[i] := 1
       await(go[i] = 1); go[i] := 0
       go[2i] := 1; go[2i+1] := 1
   else // leaf
       arrive[i] := 1
       await(go[i] = 1); go[i] := 0
fi
```

```
arrive
0 0 0 0 0 0

go
2 3 4 5 6 7
```
A Tree-based Barrier
Example Run for n=7 threads

shared
arrive[2..n]: array of atomic bits, initial values = 0
go[2..n]: array of atomic bits, initial values = 0

if i=1 then  // root
    arrive[i] := 1; arrive[i] := 0
    go[i] := 1; go[i] := 1
else if i ≤ (n-1)/2 then  // internal node
    arrive[2i] := 1; arrive[2i] := 0
    arrive[2i+1] := 1; arrive[2i+1] := 0
    arrive[i] := 1
    go[0i] := 1; go[0i] := 0
    go[2i] := 1; go[2i+1] := 1
else  // leaf
    arrive[i] := 1
    go[0i] := 1; go[0i] := 0
fi

0 1 0 0 0 0

2 3 4 5 6 7
A Tree-based Barrier
Example Run for n=7 threads

```
shared
arrive[2..n]: array of atomic bits, initial values = 0

go[2..n]: array of atomic bits, initial values = 0

1 if i=1 then // root
3    await(arrive[3] = 1); arrive[3] := 0
5  else if i <= (n-1)/2 then // internal node
6    await(arrive[2i] = 1); arrive[2i] := 0
7    await(arrive[2i+1] = 1); arrive[2i+1] := 0
8    arrive[i] := 1
9    await(go[i] = 1); go[i] := 0
10   go[2i] := 1; go[2i+1] := 1
11 else // leaf
12    await(all[i] = 1)
13    await(go[i] = 1); go[i] := 0 fi
14 fi
```

arrive 0 1 0 0 0 0

go

<p>| | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
A Tree-based Barrier
Example Run for n=7 threads

```plaintext
shared
arrive[2..n]: array of atomic bits, initial values = 0
go[2..n]: array of atomic bits, initial values = 0

if i=1 then // root
  await[arrive[2]] = 1; arrive[2] := 0
  await[arrive[3]] = 1; arrive[3] := 0
else if i ≤ (n-1)/2 then // internal node
  await[arrive[2i]] = 1; arrive[2i] := 0
  await[arrive[2i+1]] = 1; arrive[2i+1] := 0
  arrive[i] := 1
  await[go[i]] = 1; go[i] := 0
  go[2i] := 1; go[2i+1] := 1
else // leaf
  arrive[i] := 1
  await[go[i]] = 1; go[i] := 0
fi
```
A Tree-based Barrier
Example Run for n=7 threads

shared

arrive[2..n]: array of atomic bits, initial values = 0

go[2..n]: array of atomic bits, initial values = 0

if i=1 then /* root */
    await(arrive[3] = 1); arrive[3] := 0
else if i = (n-1)/2 then /* internal node */
    await(arrive[2i] = 1); arrive[2i] := 0
    await(arrive[2i+1] = 1); arrive[2i+1] := 0
    arrive[i] := 1
    await(go[i] = 1); go[i] := 0
    go[2i] := 1; go[2i+1] := 1
else /* leaf */
    arrive[i] := 1
    await(go[i] = 1); go[i] := 0
fi

arrive

0 0 0 0 0 0 0

go

2 3 4 5 6 7
A Tree-based Barrier
Example Run for n=7 threads

```
shared  arrive[2..n]: array of atomic bits, initial values = 0
        go[2..n]: array of atomic bits, initial values = 0

1 if i=1 then // root
2    await[arrive[2]] := 0
3    await[arrive[3]] := 0
5  else if i ≤ (n-1)/2 then // internal node
6      await[arrive[2i]] := 0
7      await[arrive[2i+1]] := 0
8      arrive[i] := 1
9      await[go[i]] := 0
10     go[2i] := 1; go[2i+1] := 1
11  else // leaf
12    arrive[i] := 1
13    await[go[i]] := 0; go[i] := 0 fi
14 fi
```

arrive

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

go

|   | 2 | 3 | 4 | 5 | 6 | 7 |
A Tree-based Barrier
Example Run for n=7 threads

At this point all non-root threads in some await(go) case
A Tree-based Barrier
Example Run for n=7 threads

```
shared
arrive[2..n]: array of atomic bits, initial values = 0

if i=1 then
  // root
  await[arrive[2]] = 1; arrive[2] := 0
  await[arrive[3]] = 1; arrive[3] := 0
else if i <= (n-1)/2 then
  // internal node
  await[arrive[2i]] = 1; arrive[2i] := 0
  await[arrive[2i+1]] = 1; arrive[2i+1] := 0
  arrive[i] := 1
  await[go[i]] = 1; go[i] := 0
  go[2i] := 1; go[2i+1] := 1
else
  // leaf
  arrive[i] := 1
  await[go[i]] = 1; go[i] := 0
fi
```

```
arrive

0 0 0 0 0 0 0

go

2 3 4 5 6 7
```
A Tree-based Barrier
Example Run for n=7 threads

```plaintext
shared
arrive[2..n]: array of atomic bits, initial values = 0
go[2..n]: array of atomic bits, initial values = 0

if i=1 then // root
  await(arrive[3] = 1); arrive[3] := 0
else if i = (n-1)/2 then // internal node
  await(arrive[2*i] = 1); arrive[2*i] := 0
  await(arrive[2*i+1] = 1); arrive[2*i+1] := 0
  arrive[i] := 1
  await(go[i] = 1); go[i] := 0
  go[2*i] := 1; go[2*i+1] := 1
else // leaf
  await[i] := 1
  await(go[i] = 1); go[i] := 0 fi
fi
```

arrive

| 0 | 0 | 0 | 0 | 0 | 0 |

go

| 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
A Tree-based Barrier
Example Run for n=7 threads

```
shared
arrive[2..n]: array of atomic bits, initial values := 0
go[2..n]: array of atomic bits, initial values := 0

if i=1 then // root
else if i = (n-1)/2 then // internal node
    await(arrive[2i] := 1); arrive[2i] := 0
    await(arrive[2i+1] := 1); arrive[2i+1] := 0
    arrive[i] := 1
    await(go[2i] := 1); go[2i] := 0
    go[2i+1] := 1
else // leaf
    arrive[i] := 1
    await(go[0i] := 1); go[0i] := 0 fi
fi
```

<table>
<thead>
<tr>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
</table>

```
arrive

<table>
<thead>
<tr>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
</table>

| 2 | 3 | 4 | 5 | 6 | 7 |
```
A Tree-based Barrier
Example Run for n=7 threads

```plaintext
shared
arrive[n]: array of atomic bits, initial values = 0

if i=1 then // root
  await(arrive[i] = 1); arrive[i] := 0
  await(arrive[2i] = 1); arrive[2i] := 0
  go[2i] := 1; go[2i+1] := 1
else if i <= (n-1)/2 then // internal node
  await(arrive[2i] = 1); arrive[2i] := 0
  await(arrive[2i+1] = 1); arrive[2i+1] := 0
  arrive[i] := 1
  await(go[i] = 1); go[i] := 0
  go[2i] := 1; go[2i+1] := 1
else
  // leaf
  arrive[i] := 1
  await(go[i] = 1); go[i] := 0 fi
fi
```

arrive

| 0 | 0 | 0 | 0 | 0 | 0 |

go

| 1 | 1 | 1 | 1 | 1 | 1 |

2 3 4 5 6 7

Finished!!
Tree Barrier Tradeoffs

• Pros:

• Cons:
Tree Barrier Tradeoffs

• **Pros:**
  - Low shared memory contention
    • No wait object is shared by more than 2 processes
    • Good for larger n
  - Fast – information from the root propagates after log(n) steps
  - Can use only atomic primitives (no special objects)
  - On some models:
    • each process spins on a locally accessible bit
    • # (remote memory ref.) = O(1) per process

• **Cons:**
  - Shared memory space complexity – O(n)
  - Asymmetric – all the processes don’t the same amount of work
  - Corner cases for n ≠ 2^k-1
Butterfly Barrier
Butterfly Barrier
Butterfly Barrier

- When would this be preferable?
Hardware Supported Barriers
Hardware Supported Barriers

- When would this be useful?
Hardware Supported Barriers

• When would this be useful?
Barriers Summary

Seen:
• Semaphore-based barrier
• Simple barrier
  • Based on atomic fetch-and-increment counter
• Local spinning barrier
  • Based on atomic fetch-and-increment counter and go array
• Tree-based barrier

Not seen:
• Test-and-Set barriers
  • Based on test-and-test-and-set objects
  • One version without memory initialization
• See-Saw barrier
• Book has condition barriers
Asynchronous Programming
Events, Promises, and Futures
Programming Models for Concurrency
Programming Models for Concurrency

• Hardware execution model:
Programming Models for Concurrency

• Hardware execution model:
  • CPU(s) execute instructions sequentially
Programming Models for Concurrency

• Hardware execution model:
  • CPU(s) execute instructions sequentially

• Programming model dimensions:
  • How to specify computation
  • How to specify communication
  • How to specify coordination/control transfer
Programming Models for Concurrency

• Hardware execution model:
  • CPU(s) execute instructions sequentially

• Programming model dimensions:
  • How to specify computation
  • How to specify communication
  • How to specify coordination/control transfer

• Techniques/primitives
  • Message passing vs shared memory
  • Preemption vs Non-preemption
Programming Models for Concurrency

• Hardware execution model:
  • CPU(s) execute instructions sequentially

• Programming model dimensions:
  • How to specify computation
  • How to specify communication
  • How to specify coordination/control transfer

• Techniques/primitives
  • Message passing vs shared memory
  • Preemption vs Non-preemption

• Dimensions/techniques not always orthogonal
Programming Models for Concurrency

• Hardware execution model:
  • CPU(s) execute instructions sequentially

• Programming model dimensions:
  • How to specify computation
  • How to specify communication
  • How to specify coordination/control transfer

• Techniques/primitives
  • Message passing vs shared memory
  • Preemption vs Non-preemption

• Dimensions/techniques not always orthogonal

Futures & Promises touch all three dimension
Futures & Promises
Futures & Promises

• Values *that will eventually become available*
Futures & Promises

• Values *that will eventually become available*

• Time-dependent states:
  • **Completed/determined**
    • Computation complete, value concrete
  • **Incomplete/undetermined**
    • Computation not complete yet
Futures & Promises

• Values *that will eventually become available*

• Time-dependent states:
  • **Completed/determined**
    • Computation complete, value concrete
  • **Incomplete/undetermined**
    • Computation not complete yet

• Construct ( future X )
  • immediately returns value
  • concurrently executes X
```java
static void runAsyncExample() {
    CompletableFuture cf = CompletableFuture.runAsync(() -> {
        assertTrue(Thread.currentThread().isDaemon());
        randomSleep();
    });
    assertFalse(cf.isDone());
    sleepEnough();
    assertTrue(cf.isDone());
}
```
```java
static void runAsyncExample() {
    CompletableFuture cf = CompletableFuture.runAsync(() -> {
        assertTrue(Thread.currentThread().isDaemon);
        randomSleep();
    });
    assertFalse(cf.isDone());
    sleepEnough();
    assertTrue(cf.isDone());
}
```
Java Example

```java
static void runAsyncExample() {
    CompletableFuture cf = CompletableFuture.runAsync(() -> {
        assertTrue(Thread.currentThread().isDaemon());
        randomSleep();
    });
    assertFalse(cf.isDone());
sleepEnough();
    assertTrue(cf.isDone());
}
```

- CompletableFuture is a container for Future object type
Java Example

```java
static void runAsyncExample() {
    CompletableFuture cf = CompletableFuture.runAsync(() -> {
        assertTrue(Thread.currentThread().isDaemon());
        randomSleep();
    });
    assertFalse(cf.isDone());
    sleepEnough();
    assertTrue(cf.isDone());
}
```

- CompletableFuture is a container for Future object type
- cf is an instance
Java Example

```java
static void runAsyncExample() {
    CompletableFuture cf = CompletableFuture.runAsync(() -> {
        assertTrue(Thread.currentThread().isDaemon);
        randomSleep();
    });
    assertFalse(cf.isDone);
    sleepEnough();
    assertTrue(cf.isDone());
}
```

- CompletableFuture is a container for Future object type
- cf is an instance
- runAsync() accepts
  - Lambda expression
  - Anonymous function
  - Functor
Java Example

```java
static void runAsyncExample() {
    CompletableFuture cf = CompletableFuture.runAsync(() -> {
        assertTrue(Thread.currentThread().isDaemon);
        randomSleep();
    });
    assertFalse(cf.isDone);
    sleepEnough();
    assertTrue(cf.isDone());
}
```

- CompletableFuture is a container for Future object type
- cf is an instance
- runAsync() accepts
  - Lambda expression
  - Anonymous function
  - Functor
- runAsync() immediately returns a waitable object (cf)
Java Example

static void runAsyncExample() {
    CompletableFuture cf = CompletableFuture.runAsync(() -> {
        assertTrue(Thread.currentThread().isDaemon());
        randomSleep();
    });
    assertFalse(cf.isDone());
sleepEnough();
    assertTrue(cf.isDone());
}

- CompletableFuture is a container for Future object type
- cf is an instance
- runAsync() accepts
  - Lambda expression
  - Anonymous function
  - Functor
- runAsync() immediately returns a waitable object (cf)
- Where (on what thread) does the lambda expression run?
Futures and Promises: Why two kinds of objects?

```cpp
future<int> f1 = async(foo1);
...
int result = f1.get();
```
Futures and Promises:
Why two kinds of objects?

```java
future<int> f1 = async(foo1);
...
int result = f1.get();
```
Futures and Promises:
Why two kinds of objects?

Promise: “thing to be done”

Future: encapsulation
(something to give caller)

future<int> f1 = async(foo1);
...  
int result = f1.get();
Futures and Promises: Why two kinds of objects?

Promise: “thing to be done”

Future: encapsulation (something to give caller)

Promise to do something in the future
Futures vs Promises

- **Future**: read-only reference to uncompleted value
- **Promise**: single-assignment variable that the future refers to
- Promises *complete* the future with:
  - Result with success/failure
  - Exception
Futures vs Promises

- **Future:** read-only reference to uncompleted value
- **Promise:** single-assignment variable that the future refers to
- Promises *complete* the future with:
  - Result with success/failure
  - Exception

<table>
<thead>
<tr>
<th>Language</th>
<th>Promise</th>
<th>Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algol</td>
<td>Thunk</td>
<td>Address of async result</td>
</tr>
<tr>
<td>Java</td>
<td>Future&lt;T&gt;</td>
<td>CompletableFuture&lt;T&gt;</td>
</tr>
<tr>
<td>C#/.NET</td>
<td>TaskCompletionSource&lt;T&gt;</td>
<td>Task&lt;T&gt;</td>
</tr>
<tr>
<td>JavaScript</td>
<td>Deferred</td>
<td>Promise</td>
</tr>
<tr>
<td>C++</td>
<td>std::promise</td>
<td>std::future</td>
</tr>
</tbody>
</table>
Futures vs Promises

- **Future**: read-only reference to uncompleted value
- **Promise**: single-assignment variable that the future refers to
- Promises *complete* the future with:
  - Result with success/failure
  - Exception

<table>
<thead>
<tr>
<th>Language</th>
<th>Promise</th>
<th>Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algol</td>
<td>Thunk</td>
<td>Address of async result</td>
</tr>
<tr>
<td>Java</td>
<td>Future&lt;T&gt;</td>
<td>CompletableFuture&lt;T&gt;</td>
</tr>
<tr>
<td>C#/.NET</td>
<td>TaskCompletionSource&lt;T&gt;</td>
<td>Task&lt;T&gt;</td>
</tr>
<tr>
<td>JavaScript</td>
<td>Deferred</td>
<td>Promise</td>
</tr>
<tr>
<td>C++</td>
<td>std::promise</td>
<td>std::future</td>
</tr>
</tbody>
</table>
Futures vs Promises

- **Future**: read-only reference to uncompleted value
- **Promise**: single-assignment variable that the future refers to
- Promises *complete* the future with:
  - Result with success/failure
  - Exception

<table>
<thead>
<tr>
<th>Language</th>
<th>Promise</th>
<th>Future</th>
</tr>
</thead>
<tbody>
<tr>
<td>Algol</td>
<td>Thunk</td>
<td>Address of async result</td>
</tr>
<tr>
<td>Java</td>
<td>Future&lt;T&gt;</td>
<td>CompletableFuture&lt;T&gt;</td>
</tr>
<tr>
<td>C#/NET</td>
<td>TaskCompletionSource&lt;T&gt;</td>
<td>Task&lt;T&gt;</td>
</tr>
<tr>
<td>JavaScript</td>
<td>Deferred</td>
<td>Promise</td>
</tr>
<tr>
<td>C++</td>
<td>std::promise</td>
<td>std::future</td>
</tr>
</tbody>
</table>

Mnemonic: Promise to **do** something
Make a promise for the future
Putting Futures in Context
My unvarnished opinion
Putting Futures in Context
My unvarnished opinion

Futures:
Putting Futures in Context
My unvarnished opinion

Futures:

• *abstraction* for concurrent work supported by
  • Compiler: abstractions are *language-level objects*
  • Runtime: scheduler, task queues, thread-pools are *transparent*
Putting Futures in Context
My unvarnished opinion

Futures:

• *abstraction* for concurrent work supported by
  • Compiler: abstractions are *language-level objects*
  • Runtime: scheduler, task queues, thread-pools are *transparent*

• Programming remains **mostly** imperative/sequential
  • Threads of control peppered with asynchronous/concurrent tasks
Putting Futures in Context
My unvarnished opinion

Futures:

• *abstraction* for concurrent work supported by
  • Compiler: abstractions are *language-level objects*
  • Runtime: scheduler, task queues, thread-pools are *transparent*
• Programming remains *mostly* imperative/sequential
  • Threads of control peppered with asynchronous/concurrent tasks

```java
static void runAsyncExample() {
    CompletableFuture cf = CompletableFuture.runAsync(() -> {
        assertTrue(Thread.currentThread().isDaemon());
        randomSleep();
    });
    assertFalse(cf.isDone());
    sleepEnough();
    assertTrue(cf.isDone());
}
```
Putting Futures in Context
My unvarnished opinion

Futures:

• *abstraction* for concurrent work supported by
  • Compiler: abstractions are *language-level objects*
  • Runtime: scheduler, task queues, thread-pools are *transparent*

• Programming remains *mostly* imperative/sequential
  • Threads of control peppered with asynchronous/concurrent tasks
Putting Futures in Context
My unvarnished opinion

Futures:

• *abstraction* for concurrent work supported by
  • Compiler: abstractions are *language-level objects*
  • Runtime: scheduler, task queues, thread-pools are *transparent*

• Programming remains *mostly* imperative/sequential
  • Threads of control peppered with asynchronous/concurrent tasks

Compromise Programming Model between:
Putting Futures in Context
My unvarnished opinion

Futures:
• *abstraction* for concurrent work supported by
  • Compiler: abstractions are *language-level objects*
  • Runtime: scheduler, task queues, thread-pools are *transparent*

• Programming remains *mostly* imperative/sequential
  • Threads of control peppered with asynchronous/concurrent tasks

Compromise Programming Model between:
• Event-based programming
Putting Futures in Context
My unvarnished opinion

Futures:

• *abstraction* for concurrent work supported by
  • Compiler: abstractions are *language-level objects*
  • Runtime: scheduler, task queues, thread-pools are *transparent*

• Programming remains *mostly* imperative/sequential
  • Threads of control peppered with asynchronous/concurrent tasks

Compromise Programming Model between:

• Event-based programming
• Thread-based programming
Putting Futures in Context
My unvarnished opinion

Futures:

- *abstraction* for concurrent work supported by
  - Compiler: abstractions are *language-level objects*
  - Runtime: scheduler, task queues, thread-pools are *transparent*
- Programming remains *mostly* imperative/sequential
  - Threads of control peppered with asynchronous/concurrent tasks

Compromise Programming Model between:
- Event-based programming
- Thread-based programming
GUI Programming
GUI Programming

do {
    WaitForSomething();
    RespondToThing();
} until(forever);
GUI Programming

```c
int WINAPI WndProc(HWND hWnd, UINT msg, WPARAM wParam, LPARAM lParam)
{
    switch (msg)
    {
    case WM_DESTROY:
        PostMessage(hWnd, WM_QUIT, 0, 0);
        break;
    default:
        return DefWindowProc(hWnd, msg, wParam, lParam);
    }

    return 0;
}
```
GUI Programming

// Step 2: Creating the Window
hwnd = CreateWindowEx(
    WS_EX_CLIENTEDGE,
    g_szClassName,
    "The title of my window",
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, CW_USEDEFAULT, 240, 120,
    NULL, NULL, hInstance, NULL);

// Step 2: Creating the Window
hwnd = CreateWindowEx(
    WS_EX_CLIENTEDGE,
    g_szClassName,
    "The title of my window",
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, CW_USEDEFAULT, 240, 120,
    NULL, NULL, hInstance, NULL);

if(hwnd == NULL)
    MessageBox(NULL, "Window Creation Failed!", "Error!", MB_ICONEXCLAMATION | MB_OK);
    return 0;

ShowWindow(hwnd, nCmdShow);
UpdateWindow(hwnd);

// Step 3: The Message Loop
while(GetMessage(&msg, NULL, 0, 0) > 0)
    TranslateMessage(&msg);
    DispatchMessage(&msg);
    return msg.wParam;
GUI Programming

// Step 2: Creating the Window
hwnd = CreateWindowExW(
    WS_EX_CLIENTEDGE,
    g_szClassName,
    "The title of my window",
    WS_OVERLAPPEDWINDOW,
    CW_USEDEFAULT, CW_USEDEFAULT, 240, 120,
    NULL, NULL, hInstance, NULL);

// Step 3: The Message Loop
while(GetMessage(&Msg, NULL, 0, 0) > 0)
{
    TranslateMessage(&Msg);
    DispatchMessage(&Msg);
}
GUI Programming

```c
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                   LPTSTR lpCmdline, int nCmdShow)
{
    HWND hwnd;
    MSG Msg;

    // Step 1: Registering the Window Class
    WNDCLASS wc;
    wc.lpfnWndProc = WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = g_szClassName;
    wc.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

    if(RegisterClassEx(&wc))
    {
        MessageBox(NULL, "Window Registration Failed!", "Error!", MB_ICONEXCLAMATION | MB_OK);
        return 0;
    }

    // Step 2: Creating the Window
    hwnd = CreateWindowEx(WS_EX_CLIENTEDGE,
                           g_szClassName,
                           "The title of my window",
                           WS_OVERLAPPEDWINDOW,
                           CW_USEDEFAULT, CW_USEDEFAULT, 240, 120,
                           NULL, NULL, hInstance, NULL);

    if(hwnd == NULL)
    {
        MessageBox(NULL, "Window Creation Failed!", "Error!", MB_ICONEXCLAMATION | MB_OK);
        return 0;
    }

    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);

    // Step 3: The Message Loop
    while(GetMessage(&Msg, NULL, 0, 0) > 0)
    {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }
    return Msg.wParam;
}
```
GUI Programming

```c
int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,
                    LPSTR lpCmdLine, int nCmdShow)
{
    HWND hwnd;
    MSG Msg;

    //Step 1: Registering the Window Class
    wc.cbSize = sizeof(WNDCLASSEX);
    wc.style = 0;
    wc.lpfnWndProc = WndProc;
    wc.cbClsExtra = 0;
    wc.cbWndExtra = 0;
    wc.hInstance = hInstance;
    wc.hIcon = LoadIcon(NULL, IDI_APPLICATION);
    wc.hCursor = LoadCursor(NULL, IDC_ARROW);
    wc.hbrBackground = (HBRUSH)(COLOR_WINDOW+1);
    wc.lpszMenuName = NULL;
    wc.lpszClassName = g_szClassName;
    wc.hIconSm = LoadIcon(NULL, IDI_APPLICATION);

    if(RegisterClassEx(&wc))
    {
        MessageBox(NULL, "Window Registration Failed!", "Error!",
                    MB_ICONEXCLAMATION | MB_OK);
        return 0;
    }

    // Step 2: Creating the Window
    hwnd = CreateWindowEx(
        WS_EX_CLIENTEDGE,
        g_szClassName,
        "The title of my window",
        WS_OVERLAPPEDWINDOW,
        CW_USEDEFAULT, CW_USEDEFAULT, 240, 120,
        NULL, NULL, hInstance, NULL);

    if(hwnd == NULL)
    {
        MessageBox(NULL, "Window Creation Failed!", "Error!",
                    MB_ICONEXCLAMATION | MB_OK);
        return 0;
    }

    ShowWindow(hwnd, nCmdShow);
    UpdateWindow(hwnd);

    // Step 3: The Message Loop
    while(GetMessage(&Msg, NULL, 0, 0) > 0)
    {
        TranslateMessage(&Msg);
        DispatchMessage(&Msg);
    }
    return Msg.wParam;
}
```
GUI programming

```c
switch (message)
{
    //case WM_COMMAND:
    // handle menu selections etc.
    //break;
    case WM_PAINT:
    // draw our window - note: you must paint something here or not
    //break;
    case WM_DESTROY:
        PostQuitMessage(0);
    break;
    default:
    // We do not want to handle this message so pass back to Win32
    // to handle it in a default way
    return DefWindowProc(hWnd, message, wParam, lParam);
}
```
GUI programming

```c
switch (message) {
    case WM_COMMAND:
        // handle menu selections etc.
        break;
    case WM_PAINT:
        // draw our window - note: you must paint something here or not
        break;
    case WM_DESTROY:
        PostQuitMessage(0);
        break;
    default:
        // We do not want to handle this message so pass back to Win
        // to handle it in a default way
        return DefWindowProc(hWnd, message, wParam, lParam);
}
```
GUI programming

```c
void OnMove() { ... }
void OnSize() { ... }
void OnPaint() { ... }
```

```c
switch (message) {
    //case WM_COMMAND: // handle menu select
    //    break;
    case WM_PAINT: // draw our window -
    case WM_DESTROY:
        PostQuitMessage(0);
    break;
    default:
        // We do not want to handle this message so pass back to Win
        // to handle it in a default way
        return DefWindowProc(hWnd, message, wParam, lParam);
}
```
GUI Programming Distilled

```c
winmain(...) {
    while(true) {
        message = GetMessage();
        switch(message) {
        case WM_THIS: DoThis(); break;
        case WM_THAT: DoThat(); break;
        case WM_OtherThing: DoOtherThing(); break;
        case WM_DONE: return;
        }
    }
}
```
winmain(...) {
    while(true) {
        message = GetMessage();
        switch(message) {
        case WM_THIS: DoThis(); break;
        case WM_THAT: DoThat(); break;
        case WM_OTHERTHING: DoOtherThing(); break;
        case WM_DONE: return;
        }
    }
}
Pros

• Simple imperative programming
GUI Programming Distilled

```c
winmain(...) {
    while(true) {
        message = GetMessage();
        switch(message) {
        case WM_THIS: DoThis(); break;
        case WM_THAT: DoThat(); break;
        case WM_OTHERTHING: DoOtherThing(); break;
        case WM_DONE: return;
        }
    }
}
```

Pros

• Simple imperative programming

• Good fit for uni-processor
GUI Programming Distilled

Pros
• Simple imperative programming
• Good fit for uni-processor

Cons

```c
int winmain(...) {
    while(true) {
        message = GetMessage();
        switch(message) {
            case WM_THIS: DoThis(); break;
            case WM_THAT: DoThat(); break;
            case WM_OTHERTHING: DoOtherThing(); break;
            case WM_DONE: return;
        }
    }
}
```
GUI Programming Distilled

```c
void winmain(...) {
    while(true) {
        message = GetMessage();
        switch(message) {
            case WM_THIS: DoThis(); break;
            case WM_THAT: DoThat(); break;
            case WM_OTHERTHING: DoOtherThing(); break;
            case WM_DONE: return;
        }
    }
}
```

**Pros**
- Simple imperative programming
- Good fit for uni-processor

**Cons**
- Awkward/verbose
GUI Programming Distilled

```c
void winmain(...) {
    while(true) {
        message = GetMessage();
        switch(message) {
            case WM_THIS: DoThis(); break;
            case WM_THAT: DoThat(); break;
            case WM_OTHERTHING: DoOtherThing(); break;
            case WM_DONE: return;
        }
    }
}
```

**Pros**
- Simple imperative programming
- Good fit for uni-processor

**Cons**
- Awkward/verbose
- Obscures available parallelism
GUI Programming Distilled

```c
winmain(...) {
    while(true) {
        message = GetMessage();
        switch(message) {
            case WM_LONGRUNNING_CPU_HOG: HogCPU(); break;
            case WM_HIGH_LATENCY_IO: BlockForALongTime(); break;
            case WM_DO_QUICKIMPORTANT_THING: HopeForTheBest(); break;
        }
    }
}
```

**Pros**
- Simple imperative programming
- Good fit for uni-processor

**Cons**
- Awkward/verbose
- Obscures available parallelism
winmain(...) {
    while(true) {
        message = GetMessage();
        switch(message) {
        case WM_THIS: DoThis(); break;
        case WM_THAT: DoThat(); break;
        case WM_OTHERTHING: DoOtherThing(); break;
        case WM_DONE: return;
        }
    }
}
GUI Programming Distilled

```c
winmain(...) {
    while(true) {
        message = GetMessage();
        switch(message) {
        case WM_THIS: DoThis(); break;
        case WM_THAT: DoThat(); break;
        case WM_OHERTHING: DoOtherThing(); break;
        case WM_DONE: return;
        }
    }
}
```
```c
winmain(...) {  
    while(true) {  
        message = GetMessage();  
        switch(message) {  
        case WM_THIS: DoThis(); break;  
        case WM_THAT: DoThat(); break;  
        case WM_OTHERTHING: DoOtherThing(); break;  
        case WM_DONE: return;  
        }  
    }  
}
Parallel GUI Implementation

```c
winmain(...) {
    while(true) {
        message = GetMessage();
        switch(message) {
            case WM_THIS: DoThis(); break;
            case WM_THAT: DoThat(); break;
            case WM_OTHERTHING: DoOtherThing(); break;
            case WM_DONE: return;
        }
    }
}
```
winmain(...) {
    while(true) {
        message = GetMessage();
        switch(message) {
            case WM_THIS: DoThis(); break;
            case WM_THAT: DoThat(); break;
            case WM_OTHERTHING: DoOtherThing(); break;
            case WM_DONE: return;
        }
    }
}
```c
winmain(...) {
    while(true) {
        message = GetMessage();
        switch(message) {
        case WM_THIS: DoThis(); break;
        case WM_THAT: DoThat(); break;
        case WM_OTHERTHING: DoOtherThing(); break;
        case WM_DONE: return;
        }
    }
}
```
winmain() {
    pthread_create(&tids[i++], DoThisProc);
    pthread_create(&tids[i++], DoThatProc);
    pthread_create(&tids[i++], DoOtherThingProc);
    for(j=0; j<i; j++)
        pthread_join(&tids[j]);
}

DoThisProc() {
    while(true) {
        if(ThisHasHappened)
            DoThis();
    }
}
Parallel GUI Implementation 1

void winmain() {
    pthread_create(&tids[i++], DoThisProc);
    pthread_create(&tids[i++], DoThatProc);
    pthread_create(&tids[i++], DoOtherThingProc);
    for(j=0; j<i; j++)
        pthread_join(&tids[j]);
}

void DoThisProc() {
    while(true) {
        if(ThisHasHappened)
            DoThis();
    }
}
Parallel GUI Implementation 1

Pro:
• Encapsulates parallel work

Cons:
• Obliterates original code structure
• How to assign handlers→CPUs?
• Load balance?!?
• Utilization

```c
void winmain() {
    pthread_create(&tids[i++], &DoThisProc);
    pthread_create(&tids[i++], &DoThatProc);
    pthread_create(&tids[i++], &DoOtherProc);
    for (j=0; j<i; j++)
        pthread_join(&tids[j]);
}
```
Parallel GUI Implementation 2

```c
void winmain() {
    for(i=0; i<NUMPROCS; i++)
        pthread_create(&tids[i], HandlerProc);
    for(i=0; i<NUMPROCS; i++)
        pthread_join(&tids[i]);
}

void threadproc(...)
    while(true) {
        message = GetMessage();
        switch(message) {
            case WM_THIS: DoThis();
            case WM_THAT: DoThat();
        }
    }
```
Parallel GUI Implementation 2

```c
winmain() {
    for (i=0; i<NUMPROCS; i++)
        pthread_create(&tids[i], HandlerProc);
    for (i=0; i<NUMPROCS; i++)
        pthread_join(&tids[i]);
}

threadproc(...) {
    while (true) {
        message = GetMessage();
        switch (message) {
            case WM_THIS: DoThis();
            case WM_THAT: DoThat();
        }
    }
}
```
Parallel GUI Implementation 2

```c
void winmain() {
    for (i=0; i<NUMPROCS; i++)
        pthread_create(&tids[i], H);
    for (i=0; i<NUMPROCS; i++)
        pthread_join(&tids[i]);
}
```

```c
void threadproc(...) {
while (true) {
    message = GetMessage();
    switch (message) {
        case WM_THIS: DoThis();
        case WM_THAT: DoThat();
    }
}
}
```

Pros:
- Preserves programming model
- Can recover some parallelism

Cons:
- Workers still have same problem
- How to load balance?
- Shared mutable state a problem

Pros/cons?
Parallel GUI Implementation 2

Pros:
- Preserves programming model
- Can recover some parallelism

Cons:
- Workers still have same problem
- How to load balance?
- Shared mutable state a problem

Extremely difficult to solve without changing the whole programming model...so change it
Questions?