Synchronization:
Semaphores, Monitors, Barriers

Chris Rossbach
CS378H Fall 2018
9/17/18
Today

• Questions?
• Administrivia
  • Lab 2 due next week
• Material for the day
  • Lab 1 discussion
  • Semaphores
  • Monitors
  • Barriers

• Acknowledgements
  • Thanks to Gadi Taubenfield: I borrowed and modified some of his slides on barriers

• Image credits
  • https://images-na.ssl-images-amazon.com/images/I/31EcIPmMnlL.jpg
Faux Quiz (answer any 2, 5 min)

• What is the difference between Mesa and Hoare monitors?
• Why recheck the condition on wakeup from a monitor wait?
• How can you build a barrier with spinlocks?
• How can you build a barrier with monitors?
• How can you build a barrier without spinlocks or monitors?
• What is the difference between mutex and semaphores?
• How are monitors and semaphores related?
• Why does pthread_cond_init accept a pthread_mutex_t parameter? Could it use a pthread_spinlock_t? Why [not]?
• Why do modern CPUs have both coherence and HW-supported RMW instructions? Why not just one or the other?
• What is priority inheritance?
Each cache line has a state (M, E, S, I)

- Processors “snoop” bus
- Initially $\rightarrow$ ‘I’ $\rightarrow$ Invalid
- Read one $\rightarrow$ ‘E’ $\rightarrow$ exclusive
- Reads $\rightarrow$ ‘S’ $\rightarrow$ multiple copies
- Write $\rightarrow$ ‘M’ $\rightarrow$ single copy
Random Followups: Cache Coherence

Real implementations are
- Asymmetric:
  - Bus-driven requests vs Processor-driven requests
- Asynchronous: request response protocol
  - Have transient states
- Complicated
- Optimized to the Nth degree

MOESI correction:
- Owned:
  - Line is one of many valid lines, but...
  - This core is responsible for writing it back
- Enables sharing dirty data
- When is that good?
HW Support for Read-Modify-Write (RMW)

Techniques:
- Bus locking
- Single Instruction ISA extensions
- Multi-instruction ISA extensions:

Why do we need both coherence and HW RMW?

IDEA: hardware implements something like:

```c
bool rmw(addr, value) {
    atomic {
        tmp = *addr;
        newval = modify(tmp);
        *addr = newval;
    }
}
```
Lab 1: Step 1

- Unsynchronized
- What did you expect for load imbalance
- What did you expect for scaling?
Lab 1: Step 2

- Spinlocks, mutex, atomic
- What did you expect for load imbalance?
- What did you expect for scaling?

Load imbalance

Runtime (seconds)

Threads

Threads

Load imbalance

Scaling
Step 3

- Use/abuse affinity
- What did you expect for load imbalance
- What did you expect for scaling?

![Diagram showing runtime (seconds) against threads for different synctype options: atomic, atomic-aff, atomic-aff-nobalance, mutex, mutex-aff, mutex-aff-nobalance, spinlock, spinlock-aff, spinlock-aff-nobalance.](image)
Step 4

- Read-write ratios
- What did you expect for scaling?
Reader-Writer Locks

• Requirements:
  • State to track number of readers and writers
  • Synchronization on that state
  • Synchronization on entry/exit based on that state

• Nice to have:
  • Performance/low-overhead/lack of contention
  • Fairness
Version 0.1

- Pros and cons?

```cpp
read_lock() {
  lock();
}

write_lock() {
  lock();
}

upgrade() {
}
```
Version 0.2

- State: readers -> positive integer, writers negative
- plock/punlock $\rightarrow$ lock/unlock private state variable

```c
read_lock() {
    plock();
    while(state < 0) {
        punlock();
        plock();
    }
    state++;
    punlock();
}
```

```c
write_lock() {
    plock();
    while(state) {
        punlock();
        plock();
    }
    state = -1;
    punlock();
}
```

```c
upgrade() {
    unlock();
    write_lock();
}
```

- Pros/cons?
- Can we improve upgrade?
- Other ideas?
- Well-researched problem, cf.:
  - RCU: Read-copy-update
  - RLU: Read-log-update
Discussion

What could you do to make scale?
Lab Tricks: Output CSV

```c
if (_options->bCSV) {
    /*
        headers:
        sync-type, w-prob, threads, norm-lost, avg-reads, normminreads, normmaxreads,
        avg-writes, normminwrites, normmaxwrites, exec-sec
    */
    /* R doesn't like to group by numerical categories,
       and some of the experiments really want to be grouped that
       way (e.g. by thread count, or by RW percent. This is a
       hack, but with this flag on, output will prepend those values
       with some character data so R interprets them as strings.
       Useful for step 4. */
    printf("%s, %d, %.3f, %.3f, %.3f, %.3f, %d, %.3f, %.3f, %.3f, %.3f, %.3f\n",
           _options->synctypestr().c_str(),
           std::to_string((int)(_options->dWriteProb*100.0d)).c_str(),
           _num_threads,
           norm_lost_updates,
           norm_avg_reads,
           norm_min_reads,
           norm_max_reads,
           norm_avg_writes,
           norm_min_writes,
           norm_max_writes,
           ticks/1000000.0
       );
```
Lab Tricks: scripting your experiments
Producer-Consumer (Bounded-Buffer) Problem

- Bounded buffer: size ‘N’
  - Access entry 0… N-1, then “wrap around” to 0 again
- Producer process writes data to buffer
  - Must not write more than ‘N’ items more than consumer “consumes”
- Consumer process reads data from buffer
  - Should not try to consume if there is no data
OK, let’s write some code for this (using locks only)

object array[N]
void enqueue(object x);
object dequeue();
Semaphore Motivation

• Problem with locks: mutual exclusion, but *no ordering*
• Inefficient for producer-consumer (and lots of other things)
  • **Producer**: creates a resource
  • **Consumer**: uses a resource
  • **bounded buffer** between them
  • You need synchronization for correctness, *and*...
  • Scheduling order:
    • *producer waits if buffer full, consumer waits if buffer empty*
Semaphores

• Synchronization variable
  • Integer value
    • Can’t access value directly
    • **Must** initialize to some value
      • `sem_init(sem_t *s, int pshared, unsigned int value)`

• Two operations
  • `sem_wait`, or `down(), P()`
  • `sem_post`, or `up(), V()`

```c
int sem_wait(sem_t *s) {  
    wait until value of semaphore s  
    is greater than 0  
    decrement the value of  
    semaphore s by 1  
}
```

```c
int sem_post(sem_t *s) {  
    increment the value of  
    semaphore s by 1  
    if there are 1 or more  
    threads waiting, wake 1  
}
```
Semaphore Uses

• Mutual exclusion
  • Semaphore as mutex
  • What should initial value be?
    • Binary semaphore: X=1
    • (Counting semaphore: X>1)

• Scheduling order
  • One thread waits for another
  • What should initial value be?

  // thread 0
  ... // 1st half of computation
  sem_post(s);

  // thread 1
  sem_wait(s);
  ... // 2nd half of computation

// initialize to X
sem_init(s, 0, X)

sem_wait(s);
// critical section
sem_post(s);
Producer-Consumer with semaphores

• Two semaphores
  • `sem_t full; // # of filled slots`
  • `sem_t empty; // # of empty slots`

• Problem: mutual exclusion?

```c
sem_init(&full, 0, 0);
sem_init(&empty, 0, N);

producer() {
    sem_wait(empty);
    ... // fill a slot
    sem_post(full);
}

consumer() {
    sem_wait(full);
    ... // empty a slot
    sem_post(empty);
}
```
Producer-Consumer with semaphores

- Three semaphores
  - `sem_t full; // # of filled slots`
  - `sem_t empty; // # of empty slots`
  - `sem_t mutex; // mutual exclusion`

```
sem_init(&full, 0, 0);
sem_init(&empty, 0, N);
sem_init(&mutex, 0, 1);
```

```c
producer() {
    sem_wait(empty);
    sem_wait(&mutex);
    ...
    // fill a slot
    sem_post(&mutex);
    sem_post(full);
}
```

```c
consumer() {
    sem_wait(full);
    sem_wait(&mutex);
    ...
    // empty a slot
    sem_post(&mutex);
    sem_post(empty);
}
```
Pthreads and Semaphores

- No `pthread_semaphore_t`!
  - Type: `pthread_semaphore_t`
  - `int pthread_semaphore_init(pthread_spinlock_t *lock);`
  - `int pthread_semaphore_destroy(pthread_spinlock_t *lock);`
  - ???

```c
int sem_wait(sem_t *sem)
```
What is a monitor?

- Monitor: one big lock for set of operations/methods
- Language-level implementation of mutex
  - Entry procedure: called from outside
  - Internal procedure: called within monitor
  - Wait within monitor releases lock

Many variants...
Pthreads and conditions/monitors

- Type `pthread_cond_t`

```c
int pthread_cond_init(pthread_cond_t *cond,
                    const pthread_condattr_t *attr);
int pthread_cond_destroy(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Java: synchronized keyword
wait() / notify() / notifyAll()

C#: Monitor class
Enter() / Exit() / Pulse() / PulseAll()
Hoare-style Monitors
(aka blocking condition variables)

Given entrance queue ‘e’, signal queue ‘s’, condition var ‘c’

**enter:**

```python
if (locked):
    e.push_back(thread)
else
    lock
```

**wait C:**

```python
C.q.push_back(thread)
schedule // block this thread
```

**signal C:**

```python
if (C.q.any())
    t = C.q.pop_front() // t → "the signaled thread"
    s.push_back(t)
schedule // or t.run
    // block this thread
```

**schedule:**

```python
if s.any()
    t ← s.pop_first()
    t.run
else if e.any()
    t ← e.pop_first()
    t.run
else
    unlock // monitor unoccupied
```

- Leave calls schedule
- Signaler must wait, but gets priority over threads on entrance queue
- How is this different from Mesa monitors?
- Is s queue necessary?
Mesa-style monitors
(aka non-blocking condition variables)

**enter:**
   if locked:
   e.push_back(thread)
   block
   else
   lock

**notify C:**
   if C.q.any()
   t ← C.q.pop_front() // t is "notified"
   e.push_back(t)

**wait C:**
   C.q.push_back(thread)
   schedule
   block

**schedule:**
   if e.any()
   t ← e.pop_front
   t.run
   else
   unlock

- (Leave calls schedule)
- Can be extended with extra queues for priority
- What are the differences?
Example: anyone see a bug?

StorageAllocator: MONITOR = BEGIN
    availableStorage: INTEGER:
    moreAvailable: CONDITION:
END;

Allocate: ENTRY PROCEDURE [size: INTEGER
RETURNS [p: POINTER] = BEGIN
    UNTIL availableStorage ≥ size
        DO WAIT moreAvailable ENDLOOP;
    p ← <remove chunk of size words & update availableStorage>
END;

    <put back chunk of size words & update availableStorage>;
    NOTIFY moreAvailable END;

    pNew ← Allocate[size];
    <copy contents from old block to new block>;
    Free[pOld] END;
END.
Barriers
Prefix Sum

\[
\text{begin} \quad a \quad b \quad c \quad d \quad e \quad f
\]

\[
\text{end} \quad a \quad a+b \quad a+b+c \quad a+b+c+d \quad a+b+c+d+e \quad a+b+c+d+e+f
\]
Prefix Sum

```
begin
  a  b  c  d  e  f
  a  a+b  c  d  e  f
  a  a+b  a+b+c  d  e  f
  a  a+b  a+b+c  a+b+c+d  e  f
  a  a+b  a+b+c  a+b+c+d  a+b+c+d+e  f
```

time

```
end
  a  a+b  a+b+c  a+b+c+d  a+b+c+d+e  a+b+c+d+e+f
```
Parallel Prefix Sum

begin
a
b
\text{a+b}
c
\text{b+c}
d
\text{c+d}
e
\text{d+e}
f
\text{e+f}

end
a
a+b
a+b+c
a+b+c+d
\text{b+c+d+e}
c+d+e+f

Chapter 5
Synchronization Algorithms and Concurrent Programming
Gadi Taubenfeld © 2014
Pthreads Parallel Prefix Sum

```c
int g_values[N] = { a, b, c, d, e, f }; 

void prefix_sum_thread(void * param) {
    int i;
    int id = *((int*)param);
    int stride = 0;

    for(stride=1; stride<=N/2; stride<<=1) {
        g_values[id+stride] += g_values[id];
    }
}
```

Will this work?
Pthreads Parallel Prefix Sum

```c
pthread_mutex_t g_locks[N] = { MUTEX_INITIALIZER, ...};
int g_values[N] = { a, b, c, d, e, f };

void prefix_sum_thread(void * param) {
    int i;
    int id = *((int*)param);
    int stride = 0;

    for(stride=1; stride<=N/2; stride<<=1) {
        pthread_mutex_lock(&g_locks[id]);
        pthread_mutex_lock(&g_locks[id+stride]);
        g_values[id+stride] += g_values[id];
        pthread_mutex_unlock(&g_locks[id]);
        pthread_mutex_unlock(&g_locks[id+stride]);
    }
}
```
Parallel Prefix Sum

begin

\[ a \]
\[ a + b \]
\[ a + b + c \]
\[ a + b + c + d \]
\[ a + b + c + d + e \]
\[ a + b + c + d + e + f \]

\text{barrier}

\[ a \]
\[ a + b \]
\[ a + b + c \]
\[ a + b + c + d \]
\[ a + b + c + d + e \]
\[ a + b + c + d + e + f \]

\text{barrier}

\[ a \]
\[ a + b \]
\[ a + b + c \]
\[ a + b + c + d \]
\[ a + b + c + d + e \]
\[ a + b + c + d + e + f \]

\text{end}

\[ a \]
\[ a + b \]
\[ a + b + c \]
\[ a + b + c + d \]
\[ a + b + c + d + e \]
\[ a + b + c + d + e + f \]
What is a Barrier?

- Coordination mechanism (algorithm)
- forces processes/threads to wait until each one of them has reached a certain point.
- Once all the processes/threads reach barrier, they all can pass the barrier.

Diagram:

- Four processes approach the barrier.
- All except P4 arrive.
- Once all arrive, they continue.
Pthreads and barriers

- **Type** `pthread_barrier_t`

  ```c
  int pthread_barrier_init(pthread_barrier_t *barrier,
                          const pthread_barrierattr_t *attr,
                          unsigned count);
  int pthread_barrier_destroy(pthread_barrier_t *barrier);
  int pthread_barrier_wait(pthread_barrier_t *barrier);
  ```
Pthreads Parallel Prefix Sum

```c
pthread Barrier_t g_barrier;
pthread_mutex_t g_locks[N];
int g_values[N] = { a, b, c, d, e, f };

void init_stuff() {
    ... 
    pthread_barrier_init(&g_barrier, NULL, N-1);
}

void prefix_sum_thread(void * param) {

    int i;
    int id = (*((int*)param));
    int stride = 0;

    for(stride=1; stride<=N/2; stride<<1) {
        pthread_mutex_lock(&g_locks[id]);
        pthread_mutex_lock(&g_locks[id+stride]);
        g_values[id+stride] += g_values[id];
        pthread_mutex_unlock(&g_locks[id]);
        pthread_mutex_unlock(&g_locks[id+stride]);
        pthread_barrier_wait(&g_barrier);
    }
}
Barrier Goals

Ideal barrier properties:

- Low shared memory space complexity
- Low contention on shared objects
- Low shared memory references per process
- No need for shared memory initialization
- Symmetric-ness (same amount of work for all processes)
- Algorithm simplicity
- Simple basic primitive
- Minimal propagation time
- Reusability of the barrier (must!)
Barrier Building Blocks

- Semaphores
- Atomic Bit
- Atomic Register
- Fetch-and-increment register
- Test and set bits
- Read-Modify-Write register
Barrier with Semaphores
Barrier using Semaphores
Algorithm for N threads

shared  sem_t arrival = 1;  // sem_init(&arrival, NULL, 1)
        sem_t departure = 0;  // sem_init(&departure, NULL, 0)
atomic int counter = 0;  // (gcc intrinsics are verbose)

1  sem_wait(arrival);
2  if(++counter < N)
3     sem_post(arrival);
4  else
5     sem_post(departure);
6  sem_wait(departure);
7  if(--counter > 0)
8     sem_post(departure)
9  else
10    sem_post(arrival)

First N-1 threads post on arrival, wait on departure
Nth thread post on departure, releasing threads into phase II
(what is value of arrival?)
First N-1 threads post on departure, last posts arrival
Semaphore Barrier Action Zone

N == 3

shared
sem_t arrival = 1
sem_t departure = 0
atomic int counter = 0

sem_wait(arrival);
if(++counter < N)
    sem_post(arrival);
else
    sem_post(departure);
sem_wait(departure);
if(--counter > 0)
    sem_post(departure)
else
    sem_post(arrival)

Do we need two phases?
Still correct if counter is not atomic?
Barrier using Semaphores

Properties

• **Pros:**
  • Very Simple
  • Space complexity $O(1)$
  • Symmetric

• **Cons:**
  • Required a strong object
    • Requires some central manager
    • High contention on the semaphores
  • Propagation delay $O(n)$
Barriers based on counters
Preliminaries and Primitives

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 also called: “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 is 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>

1. \text{local.go} := \text{go}
2. \text{local.counter} := \text{fetch-and-increment (counter)}
3. \textbf{if} local.counter + 1 = n \textbf{then}
4. \hspace{1em} \text{counter} := 0
5. \hspace{1em} \text{go} := 1 - \text{go}
6. \textbf{else} \text{await}(local.go \neq \text{go})
Simple Barrier Using an Atomic Counter
Run for n=2 Processes

1. \text{local.go} := \text{go}
2. \text{local.counter} := \text{fetch-and-increment (counter)}
3. \textbf{if} \text{local.counter} + 1 = n \textbf{then}
   \hspace{1cm} \text{counter} := 0
4. \hspace{1cm} \text{go} := 1 - \text{go}
5. \hspace{1cm} \textbf{else} \textbf{await} (\text{local.go} \neq \text{go})
Simple Barrier Using an Atomic Counter
Run for n=2 Processes

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

Pros/Cons?
Simple Barrier Using an Atomic Counter

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

<table>
<thead>
<tr>
<th>shared</th>
<th>counter: fetch and increment reg. ( \mathbb{Z} ), initially ( 0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>( \text{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>

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

1. local.go := go[i]
2. local.counter := fetch-and-increment
3. if local.counter + 1 = n then
6. else await(local.go ≠ go[i])

P1, P2 Busy wait

P3, P2 Busy wait

2+1=3

counter 0

local.go 0
local.counter 0 P1

local.go 0
local.counter 1 P2

local.go 0
local.counter 2 P3
Comparison of counter-based Barriers

<table>
<thead>
<tr>
<th>Simple Barrier</th>
<th>Simple Barrier with go array</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Pros:</td>
<td>• Pros:</td>
</tr>
<tr>
<td></td>
<td>• Cons:</td>
</tr>
<tr>
<td></td>
<td>• Cons:</td>
</tr>
</tbody>
</table>
### Comparison of fetch-and-increment Barriers

<table>
<thead>
<tr>
<th>Simple Barrier</th>
<th>Simple Barrier with go array</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Pros:</strong></td>
<td><strong>Pros:</strong></td>
</tr>
<tr>
<td>• Very Simple</td>
<td>• Low contention on the go array</td>
</tr>
<tr>
<td>• Shared memory: $O(\log n)$ <strong>bits</strong></td>
<td>• In some models:</td>
</tr>
<tr>
<td>• Takes $O(1)$ until last waiting $p$ is awaken</td>
<td>• spinning is done on local memory</td>
</tr>
<tr>
<td></td>
<td>• remote mem. ref.: $O(1)$</td>
</tr>
<tr>
<td><strong>Cons:</strong></td>
<td><strong>Cons:</strong></td>
</tr>
<tr>
<td>• High contention on the go bit</td>
<td>• Shared memory: $O(n)$</td>
</tr>
<tr>
<td>• Contention on the counter register (*)</td>
<td>• Still contention on the counter register (*)</td>
</tr>
<tr>
<td></td>
<td>• Takes $O(n)$ until last waiting $p$ is awaken</td>
</tr>
</tbody>
</table>
Tree Barriers
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 move on
- The signal propagates down the tree until all the threads get the message
Assume $n = i2^k - 1$

A Tree-based Barrier
A Tree-based Barrier
program of process i

<table>
<thead>
<tr>
<th>shared</th>
<th>arrive[2..n]: array of atomic bits, initial values = 0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>go[2..n]: array of atomic bits, initial values = 0</td>
</tr>
</tbody>
</table>

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 Processes
A Tree-based Barrier

• **Pros:**
  • Low shared memory contention
    • No bit is shared by more than 2 processes
    • Good for larger n
  • Fast (in comparison to local spinning)
    – information from the root propagates after \( \log(n) \) steps
  • Uses only atomic bits (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 – not all the processes do the same amount of work
(*)
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
Questions?