Synchronization: Monitors, Barriers

Chris Rossbach
Today

• Questions?
• Administrivia
  • Start looking at Lab 2!
• Material for the day
  • 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/31EcIPmMniL.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?
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);
```

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

```c
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);
```

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

```
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);
- ...
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()`
Does this code work?

- Uses “if” to check invariants.
- Why doesn’t if work?
- How could we MAKE it work?
Hoare-style Monitors
(aka blocking condition variables)

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

**Enter:**
- if(locked):
  - e.push_back(thread)
- else
  - lock

**Wait C:**
- C.q.push_back(thread)
- schedule // block this thread

**Schedule:**
- if s.any()
  - t ← s.pop_first()
  - t.run
- else if e.any()
  - t ← e.pop_first()
  - t.run
- else
  - unlock // monitor unoccupied

**Leave:**
- schedule

**Signaler must wait, but gets priority over threads on entrance queue**

- Lock only released by
  - Schedule (if no waiters)
  - Application

**Pros/Cons?**

- Must run signaled thread immediately
- Options for signaler:
  - Switch out (go on s queue)
  - Exit (Hansen monitors)
  - Continue executing?
Mesa-style monitors
(aka non-blocking condition variables)

<table>
<thead>
<tr>
<th>enter:</th>
<th>schedule:</th>
</tr>
</thead>
<tbody>
<tr>
<td>if locked:</td>
<td>if e.any()</td>
</tr>
<tr>
<td>e.push_back(thread)</td>
<td>t ← e.pop_front</td>
</tr>
<tr>
<td>block</td>
<td>t. run</td>
</tr>
<tr>
<td>else</td>
<td>else</td>
</tr>
<tr>
<td>lock</td>
<td>unlock</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>notify C:</th>
</tr>
</thead>
<tbody>
<tr>
<td>if C.q.any()</td>
</tr>
<tr>
<td>t ← C.q.pop_front()</td>
</tr>
<tr>
<td>// t is &quot;notified &quot;</td>
</tr>
<tr>
<td>e.push_back(t)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>wait C:</th>
</tr>
</thead>
<tbody>
<tr>
<td>C.q.push_back(thread)</td>
</tr>
<tr>
<td>schedule block</td>
</tr>
</tbody>
</table>

- Leave still calls schedule
- No signal queue
- Extendable with more queues for priority
- What are the differences/pros/cons?
Mesa, Hansen, Hoare
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

begin

\[
\begin{align*}
a & \quad b & \quad c & \quad d & \quad e & \quad f \\
\end{align*}
\]

delimiter

delimiter

delimiter

delimiter

delimiter

end

\[
\begin{align*}
a & \quad a+b & \quad a+b+c & \quad a+b+c+d & \quad a+b+c+d+e & \quad a+b+c+d+e+f \\
\end{align*}
\]
**Prefix Sum**

<table>
<thead>
<tr>
<th>begin</th>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
<th>e</th>
<th>f</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>a+b</td>
<td>c</td>
<td>d</td>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>a+b</td>
<td>a+b+c</td>
<td>d</td>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>a+b</td>
<td>a+b+c</td>
<td>a+b+c+d</td>
<td>e</td>
<td>f</td>
</tr>
<tr>
<td></td>
<td>a</td>
<td>a+b</td>
<td>a+b+c</td>
<td>a+b+c+d</td>
<td>a+b+c+d+e</td>
<td>f</td>
</tr>
</tbody>
</table>

| end            | a        | a+b  | a+b+c| a+b+c+d| a+b+c+d+e| a+b+c+d+e+f |

```plaintext
BEGIN

    a
    a+b
    a+b+c
    a+b+c+d
    a+b+c+d+e

    a+b+c+d+e+f

END
```

![Prefix Sum diagram](image-url)
Parallel Prefix Sum

begin

```
a
b+c
da+e
e+f
```

end

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

Chapter 5
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

\begin{tabular}{cccccc}
\hline
a & b & c & d & e & f \\
\hline
\end{tabular}

barrier

\begin{tabular}{cccccc}
\hline
a & a+b & b+c & c+d & d+e & e+f \\
\hline
\end{tabular}

barrier

\begin{tabular}{cccccc}
\hline
a & a+b & a+b+c & a+b+c+d & b+c+d+e & c+d+e+f \\
\hline
\end{tabular}

dose
dose

\begin{tabular}{cccccc}
\hline
a & a+b & a+b+c & a+b+c+d & a+b+c+d+e & a+b+c+d+e+f \\
\hline
\end{tabular}

Chapter 5

time
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

Desirable 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: same amount of work for all processes
• Algorithm simplicity
• Simple basic primitive
• Minimal propagation time
• Reusability of the barrier (must!)
Barrier Building Blocks

• Conditions
• 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)
     else
       10. sem_post(arrival)

First N-1 threads post on arrival, wait on departure
Nth thread posts 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
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

**Await**
- For brevity, we use the `await` macro
- Not an operation of an object
- This is also called: “spinning”

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

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

**shared**
- counter: fetch and increment reg. – \{0,...n\}, initially = 0
- go: atomic bit, initial value is immaterial

**local**
- local.go: a bit, initial value is immaterial
- 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)
```
Simple Barrier Using an Atomic Counter
Run for n=2 Threads

<table>
<thead>
<tr>
<th></th>
<th>counter</th>
<th>go</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>local.go</td>
<td>local.counter</td>
</tr>
<tr>
<td>P2</td>
<td>local.go</td>
<td>local.counter</td>
</tr>
</tbody>
</table>

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

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

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
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])

Pros/Cons?
Does this actually reduce contention?
## Comparison of counter-based 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><strong>Cons:</strong></td>
<td><strong>Cons:</strong></td>
</tr>
</tbody>
</table>
## Comparison of counter-based 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 go
• The signal propagates down the tree until all the threads get the message
Assume $n = i2^k - 1$

A Tree-based Barrier: indexing

Indexing starts from 2
Root $\rightarrow$ 1, doesn’t need wait objects

Step 1: label numerically with depth-first traversal
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
- Wait for parent go signal
- Tell children to go

Child:
- arrive
- Wait for parent go signal

Root:
- Wait for arriving children
- Tell children to go
A Tree-based Barrier
Example Run for n=7 threads

```
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
  if arrive[2i] = 1 then
    await(go[i] := 1)
    go[i] := 0
  fi
fi
```

```
arrive = 0 0 0 0 0 0
go = 1 1 1 1 1 1

At this point all non-root threads in some await(go) case
```
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
    • $\# \text{(remote memory ref.)} = O(1)$ per process

• **Cons:**
  
  • Shared memory space complexity – $O(n)$
  • Asymmetric – all the processes don't do the same amount of work
Butterfly Barrier

• When would this be preferable?
Hardware Supported Barriers
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?