Synchronization: Implementing Monitors + Barriers

Chris Rossbach & Calvin Lin
CS380P
Today

• Material for the day
  • Monitor implementation
  • Barrier implementation

• Acknowledgements
  • Thanks to Gadi Taubenfield: we borrowed from some of his slides on barriers
What is a monitor?

- Same as a condition variable?
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...

Monitor != condition variable
- Encapsulates shared data behind API
- Compiler support usually involved
- May be built on conditions
Pthreads and conditions

- **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()

- Why a mutex_t parameter for pthread_cond_wait?
- Why not in p_cond_init?
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

**signal C:**
if (C.q.any())
    t = C.q.pop_front() /// t \rightarrow \text{"the signaled thread"}
s.push_back(thread)
t.run

**schedule:**
if s.any()
    t ñ s.pop_first()
t.run
else if e.any()
    t ñ 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)

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

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

**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
```

- Leave still calls schedule
- No signal queue
- Extendable with more queues for priority
- What are the differences/pros/cons?
Example: anyone see a bug?

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

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.

Solutions?
• Timeouts
• notifyAll
• Can Hoare monitors support notifyAll?
Barriers
Prefix Sum

begin

\[ \begin{array}{cccccc}
  a & b & c & d & e & f \\
\end{array} \]

end

\[ \begin{array}{cccccc}
  a & a + b & a + b + c & a + b + c + d & a + b + c + d + e & a + b + c + d + e + f \\
\end{array} \]
Prefix Sum

begin

\[
\begin{array}{cccccc}
  a & b & c & d & e & f \\
  a & a+b & c & d & e & f \\
  a & a+b & a+b+c & d & e & f \\
  a & a+b & a+b+c & a+b+c+d & e & f \\
  a & a+b & a+b+c & a+b+c+d & a+b+c+d+e & f \\
\end{array}
\]

end

\[
\begin{array}{cccccc}
  a & a+b & a+b+c & a+b+c+d & a+b+c+d+e & f \\
\end{array}
\]

cs380p: Monitors and Barriers
Parallel Prefix Sum

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

cs380p: Monitors and Barriers
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

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

barrier

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

barrier

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

dend

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

- Coordination mechanism (algorithm)
- threads wait until all reach specified point.
- Once all reach barrier, all can pass.

Diagram:

- Four threads 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

Desirable barrier properties:

• Low shared memory space complexity
• Low contention on shared objects
• Few shared memory references per thread/process
• No need for shared memory initialization
• Symmetric: same amount of work for all processes
• Algorithm simplicity
• Minimal propagation time
• Reusability (a 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

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

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)
```

Phase I
- 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?)

Phase II
- First N-1 threads post on departure, last posts arrival
Semaphore Barrier Action Zone

N == 3

shared

sem_t arrival =
sem_t departure =
atomic int counter =

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?

1

CPU 0

CPU 1

CPU 2
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

```c
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”

```c
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 doesn’t matter</td>
</tr>
<tr>
<td>local</td>
<td>local.go: a bit, initial value doesn’t 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
Run for n=2 Threads

counter \[?\] go \[?\] SM

local.go \[?\] P1
local.counter \[?\]

local.go \[?\] P2
local.counter \[?\]

code:

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
4. else await(local.go ≠ go)
Simple Barrier Using an Atomic Counter
Run for n=2 Threads

```plaintext
1 local.go := go
2 local.counter := fetch-and-increment
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>

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

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

**Simple Barrier**

- Pros:
- Cons:

**Simple Barrier with go array**

- Pros:
- Cons:
### Comparison of counter-based Barriers

- **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

- 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 \( \rightarrow \) tells children they can go
- The signal propagates down the tree until all the threads get the message
Assume \( n = 2^k - 1 \)

Arrive

Go

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

Internal:

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

Waiting for p₃ to arrive

Waiting for p₄ to arrive

Waiting for go[3]

Waiting for go[5]

Waiting for go[6]

Waiting for go[7]

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
    - \# (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
Butterfly Barrier

- When would this be preferable?
Hardware Supported Barriers

CPU
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?