Synchronization:
Semaphores, Monitors, Barriers

Chris Rossbach
CS378H
Today

• Questions?
• Administrivia
  • Start looking at Lab 2!
• 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/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?
Lab 1: Baseline

```c
void compute_sequential_prefix_sum_baseline(int * vals, int nvals) {
    for (int i = 0; i < nvals; ++i) {
        osum = sum;
        sum += vals[i];
        vals[i] = osum;
    }
}
```
Lab 1: Algorithm in Sequential Context

```java
vals[nvals - 1] = 0;
```
Lab 1: Parallel

```
vals[nvals - 1] = 0;
```

Upsweep

Downsweep
struct prefix_sum_args_t {
    int* input_vals;
    int* output_vals;
    int* vals_padded;
    bool spin;
    bool compute;
    bool profile_compute;
    bool profile_barrier;
    bool no_barrier;
    bool sequential_sweep;
    bool prefetch;
    bool affinity;
    bool syncwake;
    pthread_barrier_t* barrier;
    pthread_barrier_t* wakebarrier;
    pthread_spinlock_t* spinlock;
    spin_barrier_t* s_barrier;
    int n_vals;
    int n_vals_padded;
    int n_blocks;
    int n_threads;
    int n_chunk_size;
    int t_id;
    std::vector<int> upops;
    std::vector<int> downops;
    std::vector<std::chrono::time_point<std::chrono::high_resolution_clock>> upstarts;
    std::vector<std::chrono::time_point<std::chrono::high_resolution_clock>> upends;
    std::vector<std::chrono::time_point<std::chrono::high_resolution_clock>> downstarts;
    std::vector<std::chrono::time_point<std::chrono::high_resolution_clock>> downends;
    std::vector<std::chrono::time_point<std::chrono::high_resolution_clock>> barrierin;
    std::vector<std::chrono::time_point<std::chrono::high_resolution_clock>> barrierout;

    prefix_sum_args_t() {
        compute = true;
        spin = false;
        no_barrier = false;
        profile_compute = false;
        profile_barrier = false;
        sequential_sweep = false;
        prefetch = false;
        affinity = false;
        syncwake = true;

        upops.reserve(2000);
        downops.reserve(2000);
    }
}
Instrumentation

```cpp
void up_sweep(prefix_sum_args_t* args, 
        int* pstride) {
  // ... <snip> ...
  for (i = args->n_vals >> 1; i > 0; i >>= 1) {
    pfxsum_barrier_wait(args);
    if (args->compute) {
      ts = stride;
      if (args->profile_compute)
        back(std::chrono::high_resolution_clock::now());
    }
    blocks[tidbase + tid} {
      1) - 1;
      2) - 1;
      FETCH_DEPTH; p++) {
        if ((2 * ptid + 1) - 1;
        src[pfxdix] = (2 * ptid + 2) - 1;
        src[pfidy] = fetch(pfxaddr, 0, 0);
        src[pfxaddr, 0, 0];
        src[10y] += src[10x];
        ops++;
      }
    if (args->profile_compute)
      args->upends.push_back(std::chrono::high_resolution_clock::now());
  }
  stride *= 2;
  *pstride = stride;
  if (args->profile_compute)
    args->upops.push_back(ops);
  }
```
Discussion

Could you make it 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, %lf, %lf, %lf, %lf, %lf, %lf, %lf, %lf, %lf, %lf\n",
               _options->sync_typestr().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

```bash
#!/bin/bash
# run-step4.sh
# step 4 of lab 0 includes
# 1. different read-write ratios for spinlocks
# 2. different read-write ratios for atomics

MAX_COUNTER=1000000
ITERS=1
#TIMEFORMAT=%3B
echo "synctype"
echo "synctype"
for sync in sp
  for aff in 1
  for barrier in 1
    for ld in 1
      for wr in 1
        for wref in 1
          for co in 1
            for co in 1
              for co in 1
                for co in 1
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
                  done
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()

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

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?

// initialize to $X$
sem_init(s, 0, X)
sem_wait(s);
// critical section
sem_post(s);

// thread 0
... // 1\textsuperscript{st} half of computation
sem_post(s);

// thread 1
sem_wait(s);
... // 2\textsuperscript{nd} half of computation
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`

```c
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);

- ??

- ???
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

- **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
```

**schedule:**

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

**signal C :**

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

**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:**

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

**schedule:**

```cpp
if e.any()
    t ← e.pop_front
    t. run
else
    unlock
```

**notify C:**

```cpp
if C.q.any()
    t ← C.q.pop_front() // t is "notified"
    e.push_back(t)
```

**wait C:**

```cpp
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?

Storage Allocator: 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.

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

begin  a   b   c   d   e   f

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

time
Prefix Sum

begin

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

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

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

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

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

d ... end

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

begin

\begin{align*}
  &a \\
  &b+c \\
  &c+d \\
  &d+e \\
  &e+f \\
\end{align*}

end

\begin{align*}
  &a \\
  &a+b \\
  &a+b+c \\
  &a+b+c+d \\
  &b+c+d+e \\
  &c+d+e+f \\
\end{align*}
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  b  c  d  e  f

barrier

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

barrier

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

end

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

Chapter 5
Synchronization Algorithms and Concurrent Programming
Gadi Taubenfeld © 2014
What is a Barrier?

- Coordination mechanism (algorithm)
- processes/threads to wait until all reached specified point.
- Once all reach barrier, all can pass.
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
• 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)

Phase I
1  sem_wait(arrival);
2  if(++counter < N)
3      sem_post(arrival);
4  else
5      sem_post(departure);

Phase II
6
7
8
9
10

First N-1 threads post on 
arrival, wait on

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

\( N = 3 \)

```c
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)
```

```
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)
```

```
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

```
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”

```
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                                   |

```plaintext
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

```go
local.go := go
local.counter := fetch-and-increment (counter)
if local.counter + 1 = n then
    counter := 0
    go := 1 - go
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

```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

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

<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 awakened</td>
<td>• spinning is done on local memory</td>
</tr>
<tr>
<td>• Contentsion on the counter register (*)</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>• Contentsion 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 awakened</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 $\rightarrow$ tells children they can go
- The signal propagates down the tree until all the threads get the message
Assume $n = 2^k - 1$

A Tree-based Barrier: indexing

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

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
```

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
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[6]

Waiting for
go[8]

At this point all non-root threads in some await(go) case
Tree Barrier Tradeoffs

• Pros:

• Cons:
Butterfly Barrier

• When would this be preferable?
Hardware Supported Barriers

CPU

GPU
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?