EECS 570
Lecture 5
Synchronization

Winter 2019
Prof. Thomas Wenisch

http://www.eecs.umich.edu/courses/eecs570/

Slides developed in part by Profs. Adve, Falsafi, Hill, Lebeck, Martin, Narayanasamy, Nowatzyk, Reinhardt, Roth, Smith, Singh, and Wenisch. Some slides derived from Herlihy & Shavit “The Art of Multiprocessor Programming” used under http://creativecommons.org/licenses/by-sa/3.0/
Announcements

Project Proposals due - 1/30

Programming Assignment 1 due Friday 2/8 11:59pm
• Upload zip in Canvas
Readings

For Today:

- Michael Scott. *Shared-Memory Synchronization*. Morgan & Claypool Synthesis Lectures on Computer Architecture (Ch. 1, 4.0-4.3.3, 5.0-5.2.5).


For Wednesday:


GPU Architectures

- NVIDIA G80 – extreme SIMD parallelism in shader units
Throughput Computing: Hardware Basics

Justin Hensley
Advanced Micro Devices, Inc
Graphics Product Group

adapted from Kayvon Fatahalian’s SIGGRAPH’08 talk
What does a modern graphics API do?

1. **Vertex Assembly**
2. **Vertex Shader**
3. **Geometry Assembly**
4. **Geometry Shader**
5. **Scan Conversion**
6. **Pixel Shader**
7. **Blend**
8. **Display**
A Simple Program - Diffuse Shader

```c
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;
float4 diffuseShader(float3 norm, float2 uv)
{
    float3 kd;
    kd = myTex.Sample(mySamp, uv);
    kd *= clamp( dot(lightDir, norm), 0.0, 1.0);
    return float4(kd, 1.0);
}
```

Each invocation is independent, but no explicitly exposed parallelism.
Shader is compiled

1 Unshaded fragment in

```plaintext
sampler mySamp;
Texture2D<float3> myTex;
float3 lightDir;
float4 diffuseShader(float3 norm, float2 uv)
{
    float3 kd;
    kd = myTex.Sample(mySamp, uv);
    kd *= clamp(dot(lightDir, norm), 0.0, 1.0);
    return float4(kd, 1.0);
}
```

1 Shaded fragment out

```plaintext
<diffuseShader>:
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, 1(0.0), 1(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, l(1.0a
```
Exploit data parallelism! - add two cores

Each invocation is independent!

adapted from Kayvon Fatahalian’s SIGGRAPH’08 talk
Add even more cores - four cores
How about even more cores - 16 cores
128 cores?

How do you feed all these cores?

Think data parallel! - Graphics requires hardware process *lots* of “items” that share the same shader
Back to the simple core...

- How do you feed all these cores?
- Share cost of fetch / decode across many ALUs
- **SIMD** Processing
Back to the simple core...

- How do you feed all these cores?

- Share cost of fetch / decode across many ALUs
- **SIMD** Processing
  - Single
  - Instruction
  - Multiple
  - Data

adapted from Kayvon Fatahalian’s SIGGRAPH'08 talk
Back to the simple core...

- How do you feed all these cores?
  - Share cost of fetch/decode across many ALUs
  - **SIMD Processing**
    - **Single**

**SIMD Processing does not imply SIMD instructions!**

adapted from Kayvon Fatahalian’s SIGGRAPH’08 talk
Back to a single core...

```plaintext
<diffuseShader>:
sample r0, v4, t0, s0
mul r3, v0, cb0[0]
madd r3, v1, cb0[1], r3
madd r3, v2, cb0[2], r3
clmp r3, r3, 1(0.0), 1(1.0)
mul o0, r0, r3
mul o1, r1, r3
mul o2, r2, r3
mov o3, 1(1.0)
```
128-Fragments in parallel

16 cores → 128 ALUs (16 cores * 8 ALUs) → 16 independent instruction streams
128-things in parallel

• X cores can work on primitives (triangles)
  – “geometry shader”
• Y cores can work on vertices
  – “vertex shader”
• Z cores can work on fragments
  – “pixel shader”
• N cores can work on data/work/etc
  – “compute kernels”/“compute shaders”
• Which cores working on what data changes over time
What about branching?

```cpp
<unconditional shader code>
if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
<resume unconditional shader code>
```
What about branching?

```cpp
<unconditional shader code>

if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional shader code>
```
What about branching?

Not all ALUs do useful work! Worst case: 1/8 performance

```cpp
if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}

<resume unconditional shader code>
```
What about branching?

```cpp
<unconditional shader code>
if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
<resume unconditional shader code>
```
How to handle stalls?

• Memory access latency = 100’s to 1000’s of cycles
  – Stalls occur when a core cannot run the next instruction

• GPUs don’t have the large / fancy caches and logic that helps avoid stall because of a dependency on a previous operation.

• But we have **LOTS** of independent fragments.
  – Interleave processing of many fragments on a single core to avoid stalls caused by high latency operations.
Hiding Memory Stalls

Time (clocks)

Frag 1 ... 8

Fetch/Decode

ALU

ALU

ALU

ALU

ALU

ALU

Ctx

Ctx

Ctx

Ctx

Ctx

Ctx

Shared Ctx Data

adapted from Kayvon Fatahalian’s SIGGRAPH’08 talk
Hiding Memory Stalls

Time (clocks)

Frag 1 ... 8

Frag 9 ... 16

Frag 17 ... 24

Frag 25 ... 32

Fetch/Decode

ALU

ALU

ALU

ALU

ALU

ALU

ALU

ALU

31  adapted from Kayvon Fatahalian’s SIGGRAPH’08 talk
Hiding Memory Stalls

Time (clocks)

Frag 1 ... 8

Frag 9 ... 16

Frag 17 ... 24

Frag 25 ... 32

Stall

Runnable

adapted from Kayvon Fatahalian’s SIGGRAPH’08 talk
Hiding Memory Stalls

Time (clocks)

Frag 1 ... 8

Frag 9 ... 16

Frag 17 ... 24

Frag 25 ... 32

Stall

Runnable

adapted from Kayvon Fatahalian’s SIGGRAPH’08 talk
Hiding Memory Stalls

- Frag 1 ... 8
- Frag 9 ... 16
- Frag 17 ... 24
- Frag 25 ... 32

Time (clocks)

1. Stall
2. Stall
3. Stall
4. Stall

Runnable
Runnable
Runnable
Runnable

adapted from Kayvon Fatahalian’s SIGGRAPH’08 talk
Throughput computing

Increase run time of one group
To maximum throughput of many groups

adapted from Kayvon Fatahalian’s SIGGRAPH’08 talk
Latency Hiding with “Thread Warps”

- Warp: A set of threads that execute the same instruction (on different data elements)

- Fine-grained multithreading
  - One instruction per thread in pipeline at a time (No branch prediction)
  - Interleave warp execution to hide latencies

- Register values of all threads stay in register file

- No OS context switching

- Memory latency hiding
  - Graphics has millions of pixels
Warp-based SIMD vs. Traditional SIMD

• Traditional SIMD contains a single thread
  - Lock step
  - Programming model is SIMD (no threads) → SW needs to know vector length
  - ISA contains vector/SIMD instructions

• Warp-based SIMD consists of multiple scalar threads executing in a SIMD manner (i.e., same instruction executed by all threads)
  - Does not have to be lock step
  - Each thread can be treated individually (i.e., placed in a different warp) → programming model not SIMD
    - SW does not need to know vector length
    - Enables memory and branch latency tolerance
  - ISA is scalar → vector instructions formed dynamically
GPU Microarchitecture - Key ideas

[Aamodt et al]

- SIMT Stack – tracks thread divergence/reconvergence
  - SIMT deadlock? How to avoid?
- Scoreboard – allows overlap of instructions from same warp
- Operand collector – resolves RF conflicts
- Cache hierarchy – sub-blocking for BW and partial writes
Two-level scheduling & large warps

[Narasiman et al]

Large warp width = SIMD width = N

<table>
<thead>
<tr>
<th>Row 0</th>
<th>Th₀</th>
<th>Th₁</th>
<th>⋮</th>
<th>Thₙ₋₁</th>
</tr>
</thead>
<tbody>
<tr>
<td>Row 1</td>
<td>Thₙ</td>
<td>Thₙ₊₁</td>
<td>⋮</td>
<td>Th₂ₙ₋₁</td>
</tr>
<tr>
<td>Row 2</td>
<td>Th₂ₙ</td>
<td>Th₂ₙ₊₁</td>
<td>⋮</td>
<td>Th₃ₙ₋₁</td>
</tr>
<tr>
<td>Row K−1</td>
<td>Thₙ(K−1)</td>
<td>Thₙ(K−1)+1</td>
<td>⋮</td>
<td>ThₙK−₁</td>
</tr>
</tbody>
</table>

Figure 4: Large warp active mask

Two-level scheduling & large warps

Figure 7: Baseline round-robin vs two-level round-robin scheduling

EECS 570 Lecture 5
CUDA In One Slide

Thread

per-thread local memory

Local barrier

Block

per-block shared memory

Global barrier

Kernel $\text{foo}()$

Kernel $\text{bar}()$

per-device global memory

© NVIDIA Corporation 2009
CUDA Devices and Threads

- A compute device
  - Is a coprocessor to the CPU or host
  - Has its own DRAM (device memory)
  - Runs many threads in parallel
  - Is typically a GPU but can also be another type of parallel processing device

- Data-parallel portions of an application are expressed as device kernels which run on many threads

- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few
Thread Batching: Grids and Blocks

- A kernel is executed as a grid of thread blocks
  - All threads share data memory space
- A thread block is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
  - For hazard-free shared memory accesses
  - Efficiently sharing data through a low latency shared memory
- Two threads from two different blocks cannot cooperate
Execution Model

• Each thread block is executed by a single multiprocessor
  - Synchronized using shared memory

• Many thread blocks are assigned to a single multiprocessor
  - Executed concurrently in a time-sharing fashion
  - Keep GPU as busy as possible

• Running many threads in parallel can hide DRAM memory latency
  - Global memory access : 2~300 cycles
CUDA Device Memory Space Overview

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

- The host can R/W global, constant, and texture memories
Example: Vector Addition Kernel

// Pair-wise addition of vector elements
// One thread per addition

__global__ void vectorAdd(float* iA, float* iB, float* oC)
{
    int idx = threadIdx.x + blockDim.x * blockIdx.x;
    oC[idx] = iA[idx] + iB[idx];
}

Courtesy NVIDIA
Example: Vector Addition Host Code

```c
float* h_A = (float*) malloc(N * sizeof(float));
float* h_B = (float*) malloc(N * sizeof(float));
// ... initialize h_A and h_B

// allocate device memory
float* d_A, d_B, d_C;
cudaMalloc( (void**) &d_A, N * sizeof(float) );
cudaMalloc( (void**) &d_B, N * sizeof(float) );
cudaMalloc( (void**) &d_C, N * sizeof(float) );

// copy host memory to device
cudaMemcpy( d_A, h_A, N * sizeof(float), cudaMemcpyHostToDevice );
cudaMemcpy( d_B, h_B, N * sizeof(float), cudaMemcpyHostToDevice );

// execute the kernel on N/256 blocks of 256 threads each
vectorAdd<<< N/256, 256>>>( d_A, d_B, d_C);
```

Courtesy NVIDIA
CUDA-Strengths

- Easy to program (small learning curve)
- Success with several complex applications
  - At least 7X faster than CPU stand-alone implementations
- Allows us to read and write data at any location in the device memory
- More fast memory close to the processors (registers + shared memory)
CUDA-Limitations

- Some hardwired graphic components are hidden
- Better tools are needed
  - Profiling
  - Memory blocking and layout
  - Binary Translation
- Difficult to find optimal values for CUDA execution parameters
  - Number of thread per block
  - Dimension and orientation of blocks and grid
  - Use of on-chip memory resources including registers and shared memory
Synchronization
Synchronization objectives

- Low overhead
  - Synchronization can limit scalability (E.g., single-lock OS kernels)

- Correctness (and ease of programmability)
  - Synchronization failures are extremely difficult to debug

- Coordination of HW and SW
  - SW semantics must be tightly specified to prove correctness
  - HW can often improve efficiency
Synchronization Forms

• Mutual exclusion (critical sections)
  - Lock & Unlock

• Event Notification
  - Point-to-point (producer-consumer, flags)
  - I/O, interrupts, exceptions

• Barrier Synchronization

• Higher-level constructs
  - Queues, software pipelines, (virtual) time, counters

• Next lecture: optimistic concurrency control
  - Transactional Memory
Anatomy of a Synchronization Op

• Acquire Method
  □ Way to obtain the lock or proceed past the barrier

• Waiting Algorithm
  □ Spin (aka busy wait)
    ○ Waiting process repeatedly tests a location until it changes
    ○ Releasing process sets the location
    ○ Lower overhead, but wastes CPU resources
    ○ Can cause interconnect traffic
  □ Block (aka suspend)
    ○ Waiting process is descheduled
    ○ High overhead, but frees CPU to do other things
  □ Hybrids (e.g., spin, then block)

• Release Method
  □ Way to allow other processes to proceed
HW/SW Implementation Trade-offs

• User wants high-level (ease of programming)
  - LOCK(lock_variable); UNLOCK(lock_variable)
  - BARRIER(barrier_variable, numprocs)

• SW advantages: flexibility, portability

• HW advantages: speed

• Design objectives:
  - Low latency
  - Low traffic
  - Low storage
  - Scalability (“wait-free”-ness)
  - Fairness
Challenges

• Same sync may have different behavior at different times
  ▪ Lock accessed with low or high contention
  ▪ Different performance needs: low latency vs. high throughput
  ▪ Different algorithms best for each, need different primitives

• Multiprogramming can change sync behavior
  ▪ Process scheduling or other resource interactions
  ▪ May need algorithms that are worse in dedicated case

• Rich area of SW/HW interactions
  ▪ Which primitives are available?
  ▪ What communication patterns cost more/less?
Locks
Lock-based Mutual Exclusion

Synchronization period

- Acquire starts
- Acquire done
- Release starts
- Release done

No contention:
- Want low latency

Contention:
- Want low period
- Low traffic
- Fairness
How Not to Implement Locks

• **LOCK**
  ```c
  while (lock_variable == 1);
  lock_variable = 1;
  ```

• **UNLOCK**
  ```c
  lock_variable = 0;
  ```
Solution: Atomic Read-Modify-Write

- Test&Set($r,x$)
  
  \begin{verbatim}
  {r=m[x]; m[x]=1;}
  \end{verbatim}

- Fetch&Op($r1,r2,x,op$)
  
  \begin{verbatim}
  {r1=m[x]; m[x]=op(r1,r2);}
  \end{verbatim}

- Swap($r,x$)
  
  \begin{verbatim}
  {temp=m[x]; m[x]=r; r=temp;}
  \end{verbatim}

- Compare&Swap($r1,r2,x$)
  
  \begin{verbatim}
  {temp=r2; r2=m[x]; if r1==r2 then m[x]=temp;}
  \end{verbatim}
Implementing RMWs

• Bus-based systems:
  - Hold bus and issue load/store operations without any intervening accesses by other processors

• Scalable systems
  - Acquire exclusive ownership via cache coherence
  - Perform load/store operations without allowing external coherence requests
Load-Locked Store-Conditional

- **Load-locked**
  - Issues a normal load...
  - ...and sets a flag and address field
- **Store-conditional**
  - Checks that flag is set and address matches...
  - ...only then performs store
- **Flag is cleared by**
  - Invalidation
  - Cache eviction
  - Context switch

```c
lock: while (1) {
    load-locked r1, lock_variable
    if (r1 == 0) {
        mov r2 = 1
        if (SC r2, lock) break;
    }
}

unlock: st lock_variable, #0
```
Test-and-Set Spin Lock (T&S)

• Lock is “acquire”, Unlock is “release”

• acquire(lock_ptr):
  
  while (true):
  
    // Perform “test-and-set”
    old = compare_and_swap(lock_ptr, UNLOCKED, LOCKED)
    if (old == UNLOCKED):
      break    // lock acquired!
  
    // keep spinning, back to top of while loop

• release(lock_ptr):
  
  store[lock_ptr] <- UNLOCKED

• Performance problem

  - CAS is both a read and write; spinning causes lots of invalidations
Test-and-Test-and-Set Spin Lock (TTS)

- **acquire**(lock_ptr):
  
  ```
  while (true):
      // Perform “test”
      load [lock_ptr] -> original_value
      if (original_value == UNLOCKED):
          // Perform “test-and-set”
          old = compare_and_swap(lock_ptr, UNLOCKED, LOCKED)
          if (old == UNLOCKED):
              break  // lock acquired!

      // keep spinning, back to top of while loop
  ```

- **release**(lock_ptr):
  
  ```
  store[lock_ptr] <- UNLOCKED
  ```

- **Now “spinning” is read-only, on local cached copy**
TTS Lock Performance Issues

- Performance issues remain
  - Every time the lock is released...
  - All the processors load it, and likely try to CAS the block
  - Causes a storm of coherence traffic, clogs things up badly

- One solution: backoff
  - Instead of spinning constantly, check less frequently
  - Exponential backoff works well in practice

- Another problem with spinning
  - Processors can spin really fast, starve threads on the same core!
  - Solution: x86 adds a “PAUSE” instruction
    - Tells processor to suspend the thread for a short time

- (Un)fairness
Ticket Locks

- To ensure fairness and reduce coherence storms

- Locks have two counters: `next_ticket`, `now_serving`
  - Deli counter

- `acquire(lock_ptr)`: 
  - `my_ticket = fetch_and_increment(lock_ptr->next_ticket)`
  - `while(lock_ptr->now_serving != my_ticket); // spin`

- `release(lock_ptr)`: 
  - `lock_ptr->now_serving = lock_ptr->now_serving + 1`
    - (Just a normal store, not an atomic operation, why?)

- Summary of operation
  - To “get in line” to acquire the lock, CAS on `next_ticket`
  - Spin on `now_serving`
Ticket Locks

• Properties
  - Less of a “thundering herd” coherence storm problem
    - To acquire, only need to read new value of now_serving
  - No CAS on critical path of lock handoff
    - Just a non-atomic store
  - FIFO order (fair)
    - Good, but only if the O.S. hasn’t swapped out any threads!

• Padding
  - Allocate now_serving and next_ticket on different cache blocks
    - struct { int now_serving; char pad[60]; int next_ticket; } ...
  - Two locations reduces interference

• Proportional backoff
  - Estimate of wait time: (my_ticket - now_serving) * average hold time
Array-Based Queue Locks

• Why not give each waiter its own location to spin on?
  ☐ Avoid coherence storms altogether!

• Idea: “slot” array of size N: “go ahead” or “must wait”
  ☐ Initialize first slot to “go ahead”, all others to “must wait”
  ☐ Padded one slot per cache block,
  ☐ Keep a “next slot” counter (similar to “next_ticket” counter)

• Acquire: “get in line”
  ☐ my_slot = (atomic increment of “next slot” counter) mod N
  ☐ Spin while slots[my_slot] contains “must_wait”
  ☐ Reset slots[my_slot] to “must wait”

• Release: “unblock next in line”
  ☐ Set slots[my_slot+1 mod N] to “go ahead”
Array-Based Queue Locks

• Variants: Anderson 1990, Graunke and Thakkar 1990

• Desirable properties
  □ Threads spin on dedicated location
    ○ Just two coherence misses per handoff
    ○ Traffic independent of number of waiters
  □ FIFO & fair (same as ticket lock)

• Undesirable properties
  □ Higher uncontended overhead than a TTS lock
  □ Storage O(N) for each lock
    ○ 128 threads at 64B padding: 8KBs per lock!
    ○ What if N isn’t known at start?

• List-based locks address the O(N) storage problem
  □ Several variants of list-based locks: MCS 1991, CLH 1993/1994
List-Based Queue Lock (MCS)

- A “lock” is a pointer to a linked list node
  - next node pointer
  - boolean must_wait
  - Each thread has its own local pointer to a node “I”

- acquire(lock):
  
  I->next = null;
  predecessor = fetch_and_store(lock, I)
  if predecessor != nil
    I->must_wait = true
    predecessor->next = I
  repeat while I->must_wait
  //spin till lock is free

- release(lock):
  
  if (I->next == null)
    if compare_and_swap(lock, I, nil)
      return
  repeat while I->next = nil
  //spin to learn successor
  I->next->must_wait = false
  //wake successor
MCS Lock Example: Time 0

- **acquire(lock):**
  - I->next = null;
  - pred = FAS(lock,I)
  - if pred != nil
    - I->must_wait = true
    - pred->next = I
  - repeat while I->must_wait

- **release(lock):**
  - if (I->next == null)
    - if CAS(lock,I,nil)
      - return
    - repeat while I->next == nil
  - I->next->must_wait = false
MCS Lock Example: Time 1

- $t_1$: Acquire(L)

- acquire(lock):
  - $I->next = null$;
  - $pred = FAS(lock,I)$
  - if $pred != nil$
    - $I->must_wait = true$
    - $pred->next = I$
    - repeat while $I->must_wait$

- release(lock):
  - if $(I->next == null)$
    - if CAS(lock,I,nil)
      - return
    - repeat while $I->next == nil$
    - $I->next->must_wait = false$
MCS Lock Example: Time 2

- $t_1$: Acquire(L)
- $t_2$: Acquire(L)

```
• acquire(lock):
  I->next = null;
  pred = FAS(lock,I)
  if pred != nil
    I->must_wait = true
    pred->next = I
  repeat while I->must_wait

• release(lock):
  if (I->next == null)
    if CAS(lock,I,nil)
      return
  repeat while I->next == nil
  I->next->must_wait = false
```
MCS Lock Example: Time 3

- \( t_1 \): Acquire(L)
- \( t_2 \): Acquire(L)
- \( t_3 \): Acquire(L)

```
acquire(lock):
  I->next = null;
  pred = FAS(lock,I)
  if pred != nil
    I->must_wait = true
    pred->next = I
  repeat while I->must_wait

release(lock):
  if (I->next == null)
    if CAS(lock,I,nil)
      return
    repeat while I->next == nil
  I->next->must_wait = false
```
MCS Lock Example: Time 4

- $t_1$: Acquire(L)
- $t_2$: Acquire(L)
- $t_3$: Acquire(L)
- $t_4$: Release(L)

**acquire(lock):**
- $I->next = null$
- $pred = FAS(lock,I)$
- if $pred != nil$
  - $I->must_wait = true$
  - $pred->next = I$
  - repeat while $I->must_wait$

**release(lock):**
- if $(I->next == null)$
  - if CAS(lock,I,nil)
    - return
  - repeat while $I->next == nil$
  - $I->next->must_wait = false$
MCS Lock Example: Time 5

- $t_1$: Acquire($L$)
- $t_2$: Acquire($L$)
- $t_3$: Acquire($L$)
- $t_1$: Release($L$)
- $t_2$: Release($L$)

acquire($lock$):

- $I=>$next = null;
- pred = FAS($lock$,I)
- if pred != nil
  - $I=>$must_wait = true
  - pred->next = I
  - repeat while $I=>$must_wait

release($lock$):

- if ($I=>$next == null)
  - if CAS($lock$,I,nil)
    - return
  - repeat while $I=>$next == nil
- $I=>$next->must_wait = false
MCS Lock Example: Time 6

- \( t_1 \): Acquire(L)
- \( t_2 \): Acquire(L)
- \( t_3 \): Acquire(L)
- \( t_1 \): Release(L)
- \( t_2 \): Release(L)
- \( t_3 \): Release(L)

- acquire(lock):
  \[
  I->next = null; \\
  \text{pred} = \text{FAS(lock, I)} \\
  \text{if pred} \neq \text{nil} \\
  \quad I->\text{must\_wait} = \text{true} \\
  \quad \text{pred->next} = I \\
  \quad \text{repeat while I->\text{must\_wait}}
  \]

- release(lock):
  \[
  \text{if (I->next == null)} \\
  \quad \text{if CAS(lock, I, nil)} \\
  \quad \text{return} \\
  \quad \text{repeat while I->next == nil} \\
  \quad I->\text{next->must\_wait} = \text{false}
  \]

release() w/o CAS is more complex; see paper
Queue-based locks in HW: QOLB

• Queue On Lock Bit
  - HW maintains doubly-linked list between requesters
    - This is a key idea of “Scalable Coherence Interface”, see Unit 3
  - Augment cache with “locked” bit
    - Waiting caches spin on local “locked” cache line
  - Upon release, lock holder sends line to 1st requester
    - Only requires one message on interconnect
## Fundamental Mechanisms to Reduce Overheads

[Kägi, Burger, Goodman ASPLOS 97]

- **Basic mechanisms**
  - Local Spinning
  - Queue-based locking
  - Collocation
  - Synchronous Prefetch

<table>
<thead>
<tr>
<th></th>
<th>Local Spin</th>
<th>Queue</th>
<th>Collocation</th>
<th>Prefetch</th>
</tr>
</thead>
<tbody>
<tr>
<td>T&amp;S</td>
<td>No</td>
<td>No</td>
<td>Optional</td>
<td>No</td>
</tr>
<tr>
<td>T&amp;T&amp;S</td>
<td>Yes</td>
<td>No</td>
<td>Optional</td>
<td>No</td>
</tr>
<tr>
<td>MCS</td>
<td>Yes</td>
<td>Yes</td>
<td>Partial</td>
<td>No</td>
</tr>
<tr>
<td>QOLB</td>
<td>yes</td>
<td>Yes</td>
<td>Optional</td>
<td>Yes</td>
</tr>
</tbody>
</table>
Microbenchmark Analysis

![Graph showing relative sync period vs. number of CPUs for different algorithms: T&S, T&T&S, MCS, QOLB. The graph is sourced from Kägi 97.](image-url)
Performance of Locks

• Contention vs. No Contention
  □ Test-and-Set best when no contention
  □ Queue-based is best with medium contention
  □ Idea: switch implementation based on lock behavior
    ♢ Reactive Synchronization – Lim & Agarwal 1994
    ♢ SmartLocks – Eastep et al 2009

• High-contention indicates poorly written program
  □ Need better algorithm or data structures
Point-to-Point Event Synchronization

• Can use normal variables as flags
  
  \[
  a = f(x); \quad \text{while} \ (\text{flag} == 0); \\
  \text{flag} = 1; \quad \text{b} = g(a);
  \]

• If we know initial conditions
  
  \[
  a = f(x); \quad \text{while} \ (a == 0); \\
  \text{b} = g(a);
  \]

• Assumes Sequential Consistency!

• Full/Empty Bits
  - Set on write
  - Cleared on read
  - Can’t write if set, can’t read if clear
Barriers
Barriers

• Physics simulation computation
  - Divide up each timestep computation into N independent pieces
  - Each timestep: compute independently, synchronize

• Example: each thread executes:

```plaintext
segment_size = total_particles / number_of_threads
my_start_particle = thread_id * segment_size
my_end_particle = my_start_particle + segment_size - 1
for (timestep = 0; timestep += delta; timestep < stop_time):
    calculate_forces(t, my_start_particle, my_end_particle)
    barrier()
    update_locations(t, my_start_particle, my_end_particle)
    barrier()
```

• Barrier? All threads wait until all threads have reached it
Example: Barrier-Based Merge Sort

Step 1

Step 2

Step 3
Global Synchronization Barrier

• At a barrier
  - All threads wait until all other threads have reached it

• Strawman implementation (**wrong**!)

```plaintext
global (shared) count : integer := P

procedure central_barrier
    if fetch_and_decrement(&count) == 1
        count := P
    else
        repeat until count == P
```

• What is wrong with the above code?
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    threadSense = new ThreadLocal<boolean>…

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {}
        }
    }
    threadSense.set(!mySense)}}}
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    threadSense = new ThreadLocal<boolean>…

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {}
        }
    threadSense.set(!mySense)]]>

Completed odd or even-numbered phase?
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;

    threadSense = new ThreadLocal<>

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {} 
        }
        threadSense.set(!mySense)}

Sense-Reversing Barriers

Store sense for next phase
Sense-Reversing Barriers

```java
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    ThreadLocal<boolean> threadSense = new ThreadLocal<boolean>…

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement() == 1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {}
        }
        threadSense.set(!mySense)}
```

Get new sense determined by last phase
Sense-Reversing Barriers

```java
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    ThreadLocal<boolean> threadSense = new ThreadLocal<boolean>…

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement() == 1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {}
        }
        threadSense.set(!mySense)}
```

If I’m last, reverse sense for next time
Sense-Reversing Barriers

public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    threadSense = new ThreadLocal<boolean>…

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement() == 1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {} // Otherwise, wait for sense to flip
        }
        threadSense.set(!mySense)}
}
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    threadSense = new ThreadLocal<boolean>...;

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement() == 1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {}
        }
        threadSense.set(!mySense)}}

Prepare sense for next phase
Other Barrier Implementations

• Problem with centralized barrier
  - All processors must increment each counter
  - Each read/modify/write is a serialized coherence action
    - Each one is a cache miss
  - O(n) if threads arrive simultaneously, slow for lots of processors

• Combining Tree Barrier
  - Build a \( \log_k(n) \) height tree of counters (one per cache block)
  - Each thread coordinates with \( k \) other threads (by thread id)
  - Last of the \( k \) processors, coordinates with next higher node in tree
  - As many coordination address are used, misses are not serialized
  - O(\( \log n \)) in best case

• Static and more dynamic variants
  - Tree-based arrival, tree-based or centralized release
Combining Tree Barrier

```java
public class Node{
    AtomicInteger count; int size;
    Node parent; Volatile boolean sense;

    public void await() {...
        if (count.getAndDecrement()==1) {
            if (parent != null) {
                parent.await()}
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {}{}
    }...}}
```
Combining Tree Barrier

```java
public class Node{
    AtomicInteger count; int size;
    Node parent;
    volatile boolean sense;

    public void await() {
        if (count.getAndDecrement() == 1) {
            if (parent != null) {
                parent.await()
            }
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {}
        }
    }
}
```

Parent barrier in tree
public class Node{
    AtomicInteger count; int size;
    Node parent; Volatile boolean sense;

    public void await() {
        if (count.getAndDecrement()==1) {
            if (parent != null) {
                parent.await()
            }
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {}
        }
    }
}
public class Node {
    AtomicInteger count; int size;
    Node parent; Volatile boolean sense;

    public void await() {
        if (count.getAndDecrement() == 1) {
            if (parent != null) {
                parent.await();
            }
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {}
        }
    }
}
public class Node{
    AtomicInteger count; int size;
    Node parent; Volatile boolean sense;

    public void await() {
        if (count.getAndDecrement() == 1) {
            if (parent != null) {
                parent.await();
            }
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {
                ...}
        }...}}}

Prepare for next phase
public class Node {
    AtomicInteger count; int size;
    Node parent; volatile boolean sense;

    public void await() {
        if (count.getAndDecrement() == 1) {
            if (parent != null) {
                parent.await();
            }
            count.set(size);
            sense = mySense;
        } else {
            while (sense != mySense) {}
        }
    }
}
Combining Tree Barrier

```java
public class Node{
    AtomicInteger count; int size;
    Node parent; Volatile boolean sense;

    public void await() {...
        if (count.getAndDecrement() == 1) {
            if (parent != null) {
                parent.await();
            }
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {} 
        }
    }
}
```

I’m not last, so wait for notification
Combining Tree Barrier

• No sequential bottleneck
  □ Parallel getAndDecrement() calls

• Low memory contention
  □ Same reason

• Cache behavior
  □ Local spinning on bus-based architecture
  □ Not so good for NUMA