Parallel Software

- Why is it so hard?
  - Conscious mind is inherently sequential
  - (sub-conscious mind is extremely parallel)
- Identifying parallelism in the problem
- Expressing parallelism to the hardware
- Effectively utilizing parallel hardware
  - Balancing work
  - Coordinating work
- Debugging parallel algorithms

Finding Parallelism

1. Functional parallelism
   - Car: {engine, brakes, entertain, nav, …}
   - Game: {physics, logic, UI, render, …}
   - Signal processing: {transform, filter, scaling, …}
2. Automatic extraction
   - Decompose serial programs
3. Data parallelism
   - Vector, matrix, db table, pixels, …
4. Request parallelism
   - Web, shared database, telephony, …

Expressing Parallelism

- SIMD – introduced by Cray-1 vector supercomputer
  - MMX, SSE/SSE2/SSE3/SSE4, AVX at small scale
- SPMD or SIMT – GPGPU model
  - All processors execute same program on disjoint data
  - Loose synchronization vs. rigid lockstep of SIMD
- MIMD – most general
  - Each processor executes its own program
- Expressed through standard interfaces
  - API, ABI, ISA

Programming Models

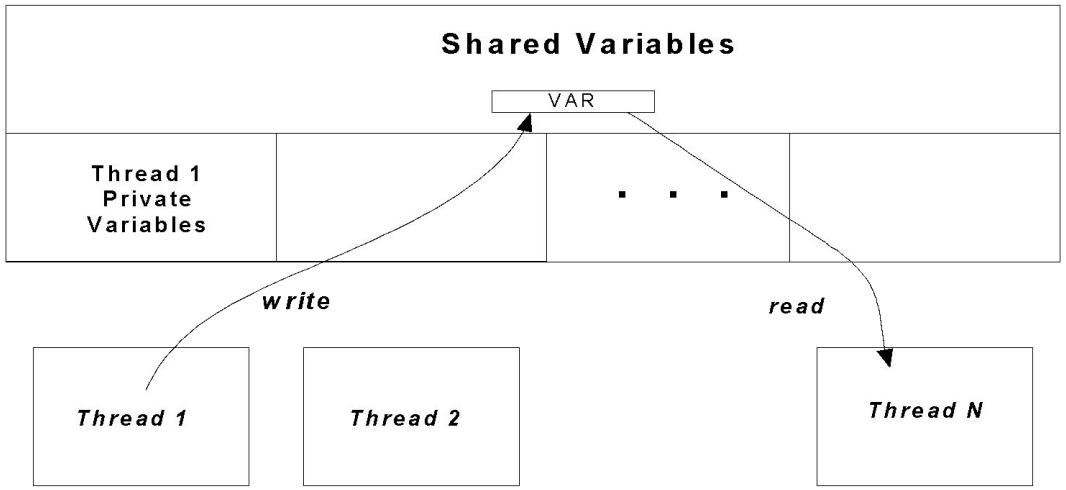
- High level paradigm for expressing an algorithm
  - Examples:
    - Functional
    - Sequential, procedural
    - Shared memory
    - Message Passing
- Embodied in high level languages that support concurrent execution
  - Incorporated into HLL constructs
  - Incorporated as libraries added to existing sequential language
- Top level features:
  - For conventional models – shared memory, message passing
  - Multiple threads are conceptually visible to programmer
  - Communication/synchronization are visible to programmer

Programming Model Elements

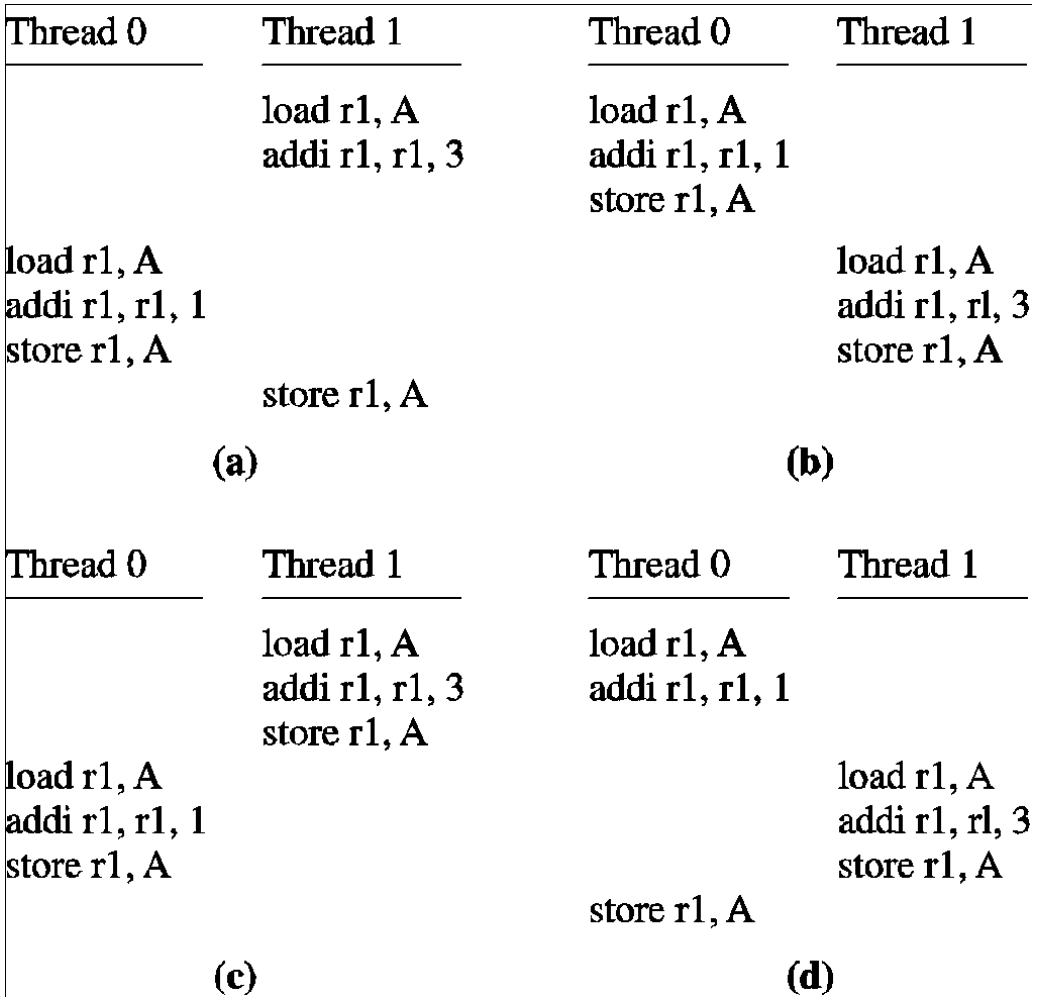
- For both Shared Memory and Message Passing
- Processes and threads
  - ***Process:*** A shared address space and one or more threads of control
  - ***Thread:*** A program sequencer and private address space
  - ***Task***: Less formal term – part of an overall job
  - Created, terminated, scheduled, etc.
- Communication
  - Passing of data
- Synchronization
  - Communicating control information
  - To assure reliable, deterministic communication

Shared Memory

- Flat shared memory or object heap
  - Synchronization via memory variables enables reliable sharing
- Single process
- Multiple threads per process
  - Private memory per thread
- Typically built on shared memory hardware system



Shared Memory Communication



Thread 0	Thread 1
	load r1, A
	addi r1, r1, 3
load r1, A	
addi r1, r1, 1	
store r1, A	
	store r1, A

(a)

Thread 0	Thread 1
load r1, A	
addi r1, r1, 1	
store r1, A	
	load r1, A
	addi r1, rl, 3
	store r1, A

(b)

Thread 0	Thread 1
	load r1, A
	addi r1, r1, 3
	store r1, A
load r1, A	
addi r1, r1, 1	
store r1, A	

(c)

Thread 0	Thread 1
load r1, A	
addi r1, r1, 1	
	load r1, A
	addi r1, rl, 3
	store r1, A
store r1, A	

(d)

- Reads and writes to shared variables via normal language (assignment) statements (e.g. assembly load/store)

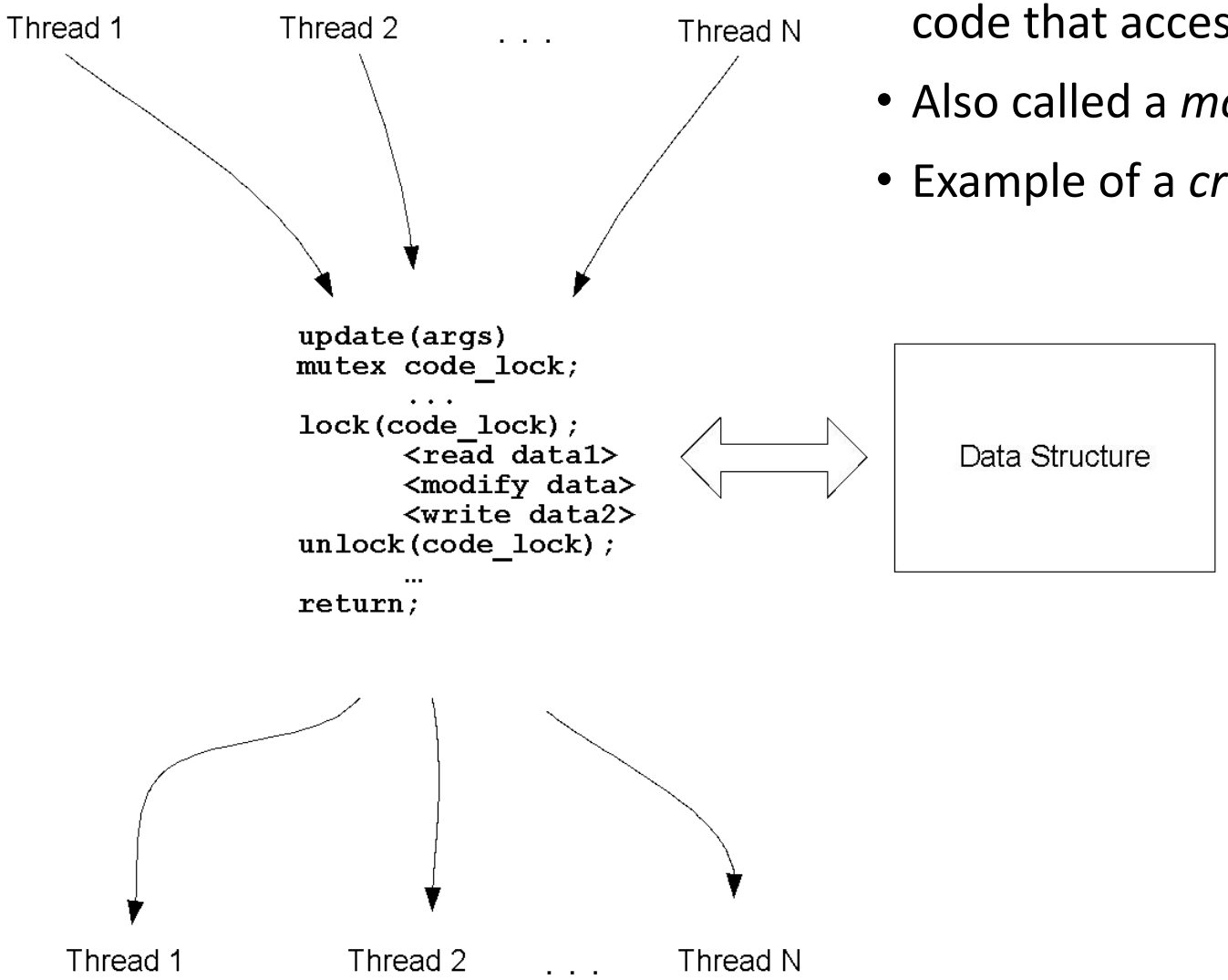
Shared Memory Synchronization

- What really gives shared memory programming its structure
- Usually explicit in shared memory model
  - Through language constructs or API
- Three major classes of synchronization
  - Mutual exclusion (mutex)
  - Point-to-point synchronization
  - Rendezvous
- Employed by *application design patterns*
  - *A general description or template for the solution to a commonly recurring software design problem.*

Mutual Exclusion (mutex)

- Assures that only one thread at a time can access a code or data region
- Usually done via *locks*
  - One thread acquires the lock
  - All other threads excluded until lock is released
- Examples
  - pthread_mutex_lock
  - pthread_mutex_unlock
- Two main application programming patterns
  - Code locking
  - Data locking

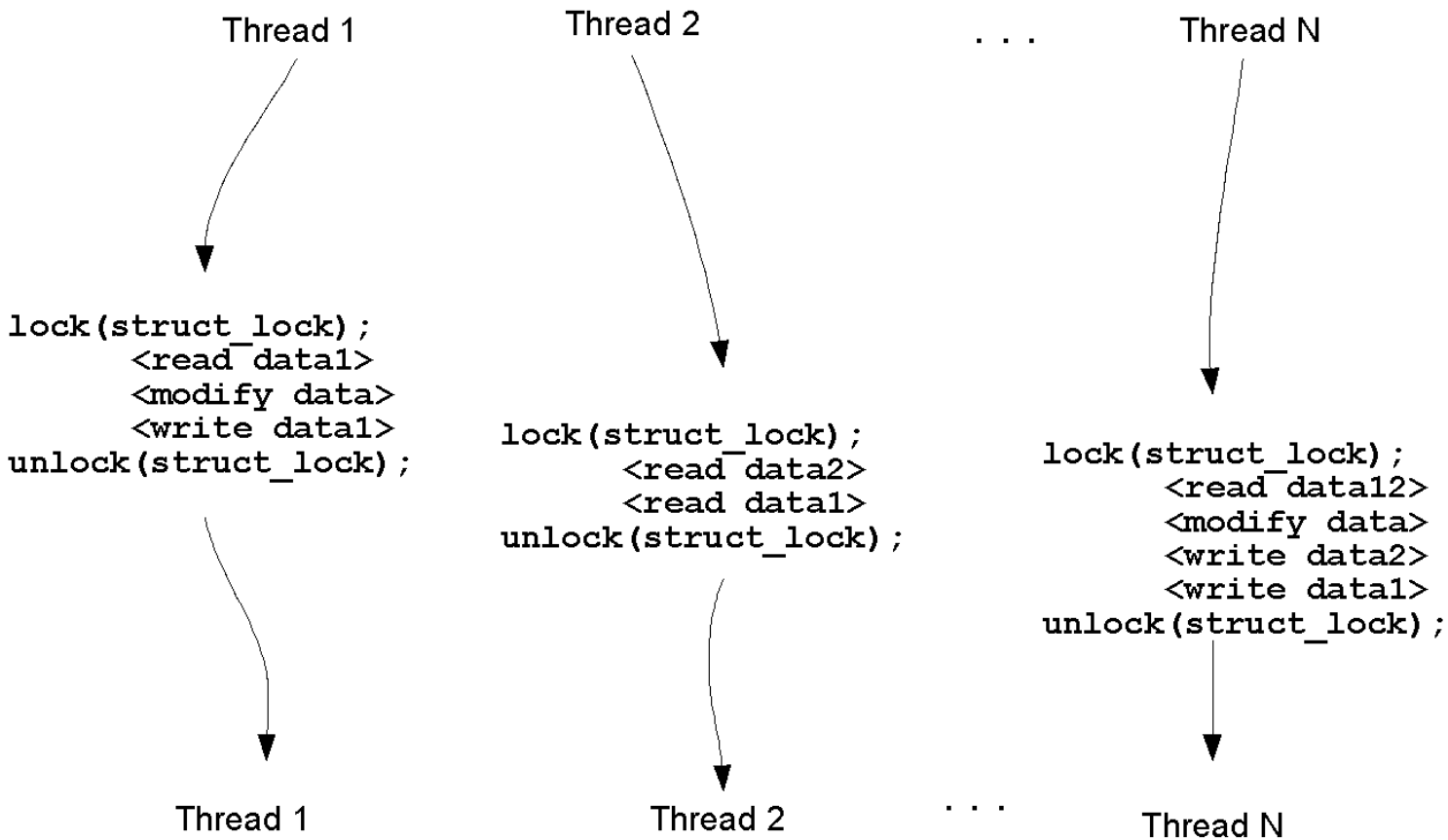
Code Locking



- Protect shared data by locking the code that accesses it
- Also called a *monitor* pattern
- Example of a *critical section*

Data Locking

- Protect shared data by locking data structure



Data Locking

- Preferred when data structures are read/written in combinations
- Example:

```
<thread 0>
Lock(mutex_struct1)
Lock(mutex_struct2)
    <access struct1>
    <access struct2>
Unlock(mutex_data1)
Unlock(mutex_data2)
```

```
<thread 1>
Lock(mutex_struct1)
Lock(mutex_struct3)
    <access struct1>
    <access struct3>
Unlock(mutex_data1)
Unlock(mutex_data3)
```

```
<thread 2>
Lock(mutex_struct2)
Lock(mutex_struct3)
    <access struct2>
    <access struct3>
Unlock(mutex_data2)
Unlock(mutex_data3)
```

Deadlock

- Data locking is prone to deadlock
  - If locks are acquired in an unsafe order
- Example:

<thread 0>

```
Lock(mutex_data1)
Lock(mutex_data2)
    <access data1>
    <access data2>
Unlock(mutex_data1)
Unlock(mutex_data2)
```

<thread 1>

```
Lock(mutex_data2)
Lock(mutex_data1)
    <access data1>
    <access data2
Unlock(mutex_data1)
Unlock (mutex_data2)
```

- Complexity
  - Disciplined locking order must be maintained, else deadlock
  - Also, composability problems
    - Locking structures in a nest of called procedures

Efficiency

- Lock contention
  - Causes threads to wait
- Function of lock *granularity*
  - Size of data structure or code that is being locked
- Extreme Case:
  - "One big lock" model for multithreaded OSes
  - Easy to implement, but very inefficient
- Finer granularity
  + Less contention
  - More locks, more locking code
  - perhaps more deadlock opportunities
- Coarser granularity
  - opposite +/- of above

Point-to-Point Synchronization

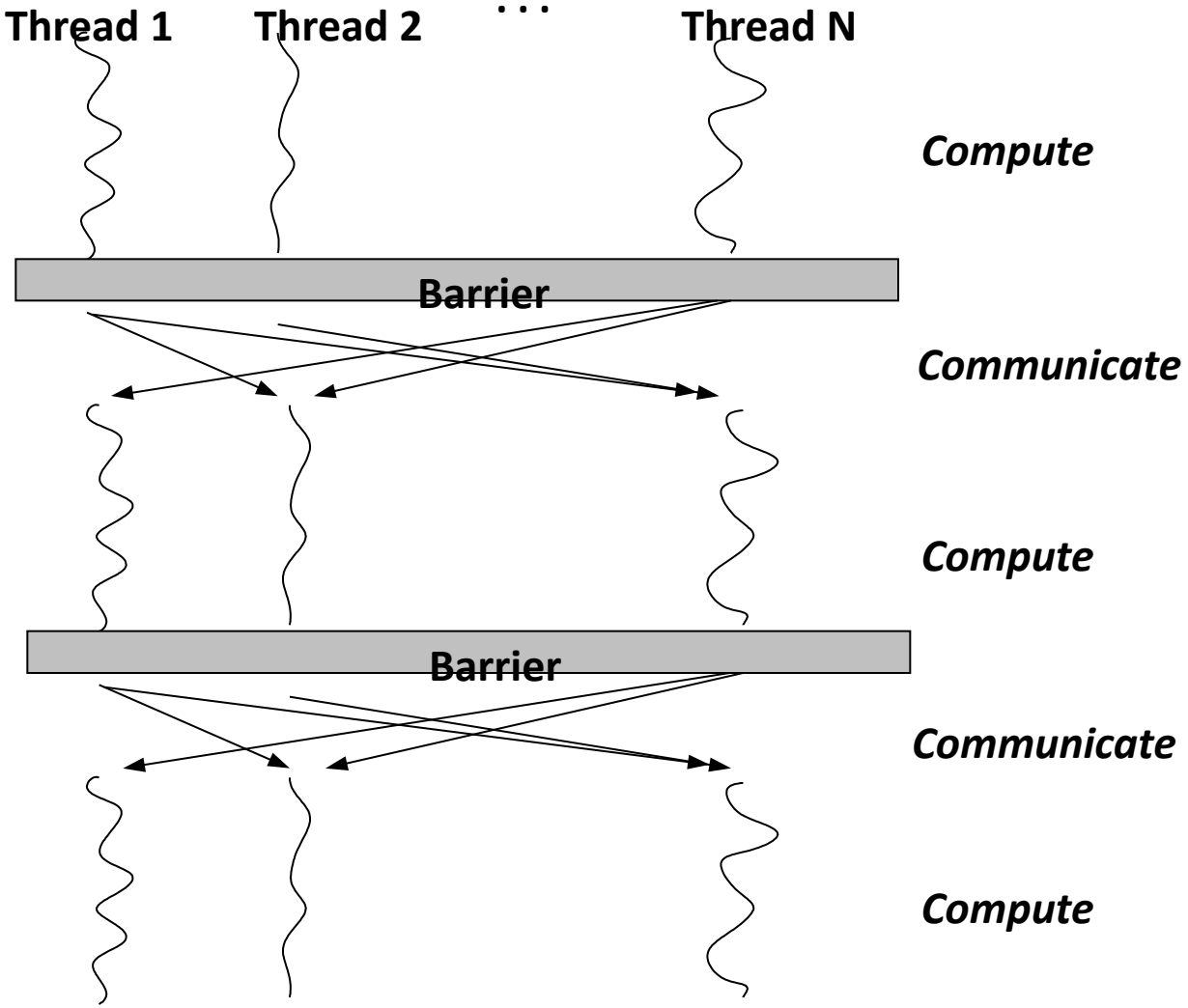
- One thread signals another that a condition holds
  - Can be done via API routines
  - Can be done via normal load/stores
- Examples
  - pthread_cond_signal
  - pthread_cond_wait
    - suspends thread if condition not true
- Application program pattern
  - Producer/Consumer

```
<Producer>                    <Consumer>
while (full ==1){};           wait   while (full == 0){}; wait
buffer = value;                         b = buffer;
full = 1;                     full = 0;
```

Rendezvous

- Two or more cooperating threads must reach a program point before proceeding
- Examples
  - wait for another thread at a join point before proceeding
    - example: pthread_join
  - barrier synchronization
    - many (or all) threads wait at a given point
- Application program pattern
  - Bulk synchronous programming pattern

Bulk Synchronous Program Pattern



Summary: Synchronization and Patterns

- mutex (mutual exclusion)
  - code locking (monitors)
  - data locking
- point to point
  - producer/consumer
- rendezvous
  - bulk synchronous

Synchronization

- Implement locks and rendezvous (barriers)
- Use loads and stores to implement lock:

```
        <thread 0>                  <thread 1>
            .                           .                       .
LAB1:   Load R1, Lock           LAB2:   Load R1, Lock
        Branch LAB1 if R1==1            Branch LAB2 if R1==1
        Ldi R1, 1                   Ldi R1,1
        Store Lock, R1                  Store Lock, R1
        .                           .
        <critical section>          <critical section>
        .                           .
        Ldi R1, 0                   Ldi R1, 0
        Store  Lock, R1             Store Lock, R1
```

Lock Implementation

- *Does not work*
- Violates mutual exclusion if both threads attempt to lock at the same time
  - In practice, may work *most* of the time…
  - Leading to an unexplainable system hang every few days

```
      <thread 0>                <thread 1>
          .                         .                         .
LAB1:   Load R1, Lock       LAB2:   Load R1, Lock
        Branch LAB1 if R1==1        Branch LAB2 if R1==1
        Ldi R1, 1                   Ldi R1,1
        Store Lock, R1              Store Lock, R1
```

Lock Implementation

- Reliable locking can be done with *atomic* read-modify-write instruction
- Example: test&set
  - read lock and write a one
  - some ISAs also set CCs (test)

```
    <thread 1>                    <thread 2>
        .                             .
LAB1:   Test&Set R1, Lock     LAB2:   Test&Set R1, Lock
        Branch LAB1 if R1==1          Branch LAB2 if R1==1
        .                             .
        <critical section>            <critical section>
        .                             .
        Reset Lock                    Reset Lock
```

Atomic Read-Modify-Write

- Many such instructions have been used in ISAs

```
Test&Set(reg,lock)      Fetch&Add(reg,value,sum)    Swap(reg,opnd)
reg ←mem(lock);         reg ← mem(sum);             temp←mem(opnd);
mem(lock) ← 1;          mem(sum)←mem(sum)+value;    mem(opnd)← reg;
                                reg ← temp
```

- More-or-less equivalent
  - One can be used to implement the others
  - Implement Fetch&Add with Test&Set:

```
try:   Test&Set(lock);
       if lock == 1 go to try;
          reg ←mem(sum);
          mem(sum) ← reg+value;
          reset (lock);
```

Lock Efficiency

- Spin Locks
  - tight loop until lock is acquired

```
LAB1:  Test&Set R1, Lock
       Branch LAB1 if R1==1
```

- Inefficiencies:
  - Memory/Interconnect resources, spinning on read/writes
  - With a cache-based systems,
    writes ⇒ lots of coherence traffic
  - Processor resource
    - not executing useful instructions

Efficient Lock Implementations

- Test&Test&Set
  - spin on check for unlock only, then try to lock
  - with cache systems, all reads can be local
    - no bus or external memory resources used

```
test_it:  load      reg, mem(lock)
          branch    test_it if reg==1
lock_it:  test&set  reg, mem(lock)
          branch    test_it if reg==1
```

- Test&Set with Backoff
  - Insert delay between test&set operations (not too long)
  - Each failed attempt ⇒ longer delay
    (Like ethernet collision avoidance)

Efficient Lock Implementations

- Solutions just given save memory/interconnect resource
  - Still waste processor resource
- Use runtime to suspend waiting process
  - Detect lock
  - Place on wait queue
  - Schedule another thread from run queue
  - When lock is released move from wait queue to run queue

Sub-Atomic Locks

- Use two instructions:

  Load linked + Store conditional

  - Load linked
    - reads memory value
    - sets special flag
    - writes address to special global address register
  - Flag cleared on
    - operations that may violate atomicity
      - (implementation-dependent)
      - e.g., write to address by another processor
      - can use cache coherence mechanisms (later)
    - context switch
  - Store conditional
    - writes value if flag is set
    - no-op if flag is clear
    - sets CC indicating success or failure

Load-Linked Store-Conditional

- Example: atomic swap (r4,mem(r1))

```
try: mov r3,r4      ;move exchange value
     ll  r2,0(r1)   ;load locked
     sc  r3,0(r1)   ;store conditional
     beqz   r3,try     ;if store fails
     mov r4,r2      ;load value to r4
```

- RISC- style implementation
  - Like many early RISC ideas, it seemed like a good idea at the time…

    register windows, delayed branches, special divide regs, etc.

Point-to-Point Synchronization

- *Can* use normal variables as flags

```
while (full ==1){}  ;spin  while (full == 0){} ;spin
a = value;                 b = value;
full = 1;                full = 0;
```

- Assumes sequential consistency (later)
  - Using normal variables may cause problems with relaxed consistency models
- May be better to use special opcodes for flag set/clear

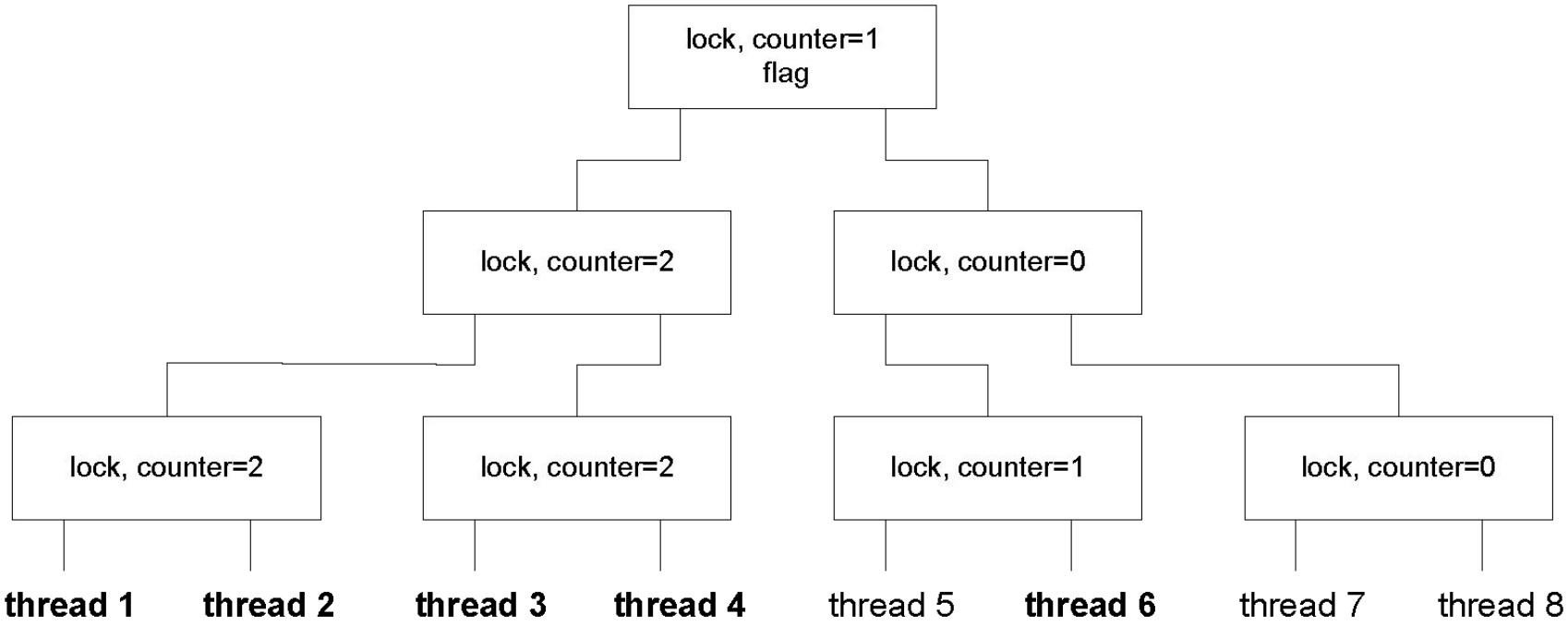
Barrier Synchronization

- Uses a lock, a counter, and a flag
  - lock for updating counter
  - flag indicates all threads have incremented counter

```
Barrier (bar_name, n) {
    Lock (bar_name.lock);
    if (bar_name.counter = 0)  bar_name.flag = 0;
    mycount = bar_name.counter++;
    Unlock (bar_name.lock);
    if (mycount == n) {
        bar_name.counter = 0;
        bar_name.flag = 1;
    }
    else  while(bar_name.flag = 0) {};   /* busy wait */
}
```

Scalable Barrier Synchronization

- Single counter can be point of contention
- Solution: use tree of locks
- Example:
  - threads 1,2,3,4,6 have completed



Memory Ordering

- Program Order
  - Processor executes instructions in architected (PC) sequence

    or at least appears to
- Loads and stores from a single processor execute in *program order*
  - Program order *must* be satisfied
  - It is part of the ISA
- What about ordering of loads and stores from *different* processors

Memory Ordering

- Producer/Consumer example:

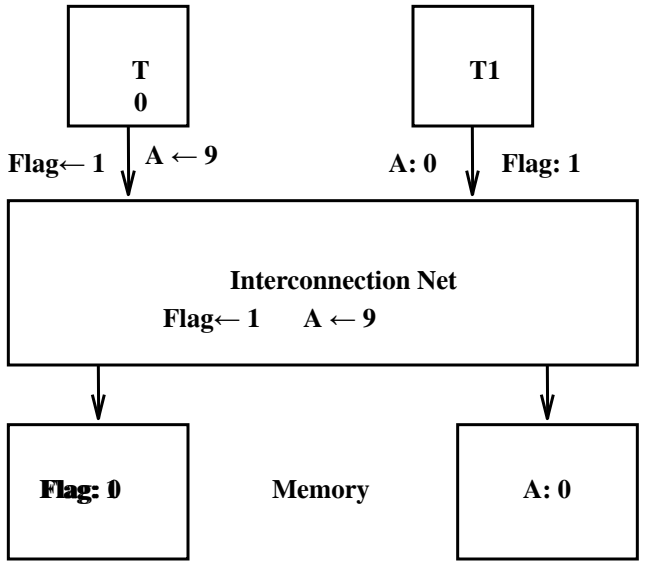
```
T0:   A=0;          T1:
      Flag = 0;
      ....
      A=9;              While (Flag==0){};
      Flag = 1;         L2: if (A==0)...
```

- Intuitively it is *impossible* for A to be 0 at L2
  - *But* it can happen if the updates to memory are reordered by the memory system
- In an MP system, memory ordering rules must be carefully defined and maintained

Practical Implementation

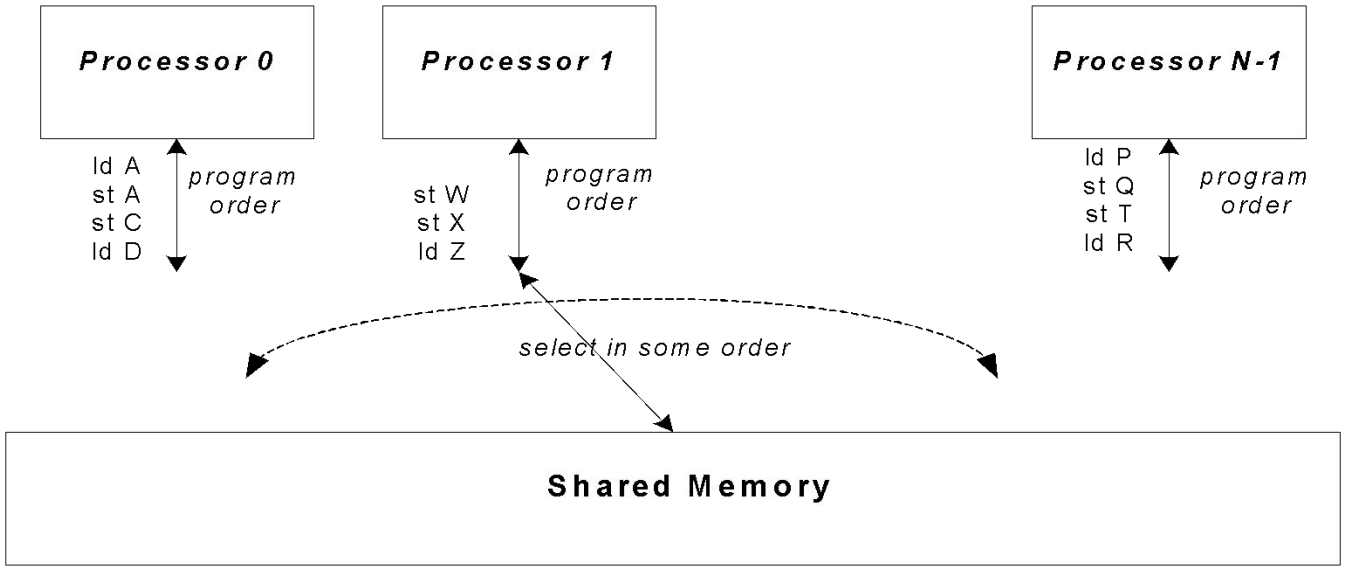
- Interconnection network with contention and buffering

```
T0:    A=0;      T1:
       Flag = 0;
       ....
       A=9;           While (Flag==0){};
       Flag = 1;           if (A==0)...
```



Sequential Consistency

"A system is sequentially consistent if the result of any execution is the same as if the operations of all processors were executed in some sequential order and the operations of each individual processor appears in this sequence in the order specified by its program" -- Leslie Lamport



Memory Coherence

- WRT individual memory locations, consistency is always maintained
  - In producer/consumer examples, coherence is always maintained
- Practically, memory coherence often reduces to cache coherence
  - Cache coherence implementations to be discussed later
- Summary
  - *Coherence* is for a single memory location
  - *Consistency* applies to apparent ordering of all memory locations
  - Memory coherence and consistency are ISA concepts

```
Thread0:                Thread1:
Store A←0
Store A←9

                        Load A=9
```

(a)

```
Thread0:                Thread1:
Store Flag←0

                        Load Flag=0
                        Load Flag=0
Store Flag←1

                        Load Flag=1
```

(b)