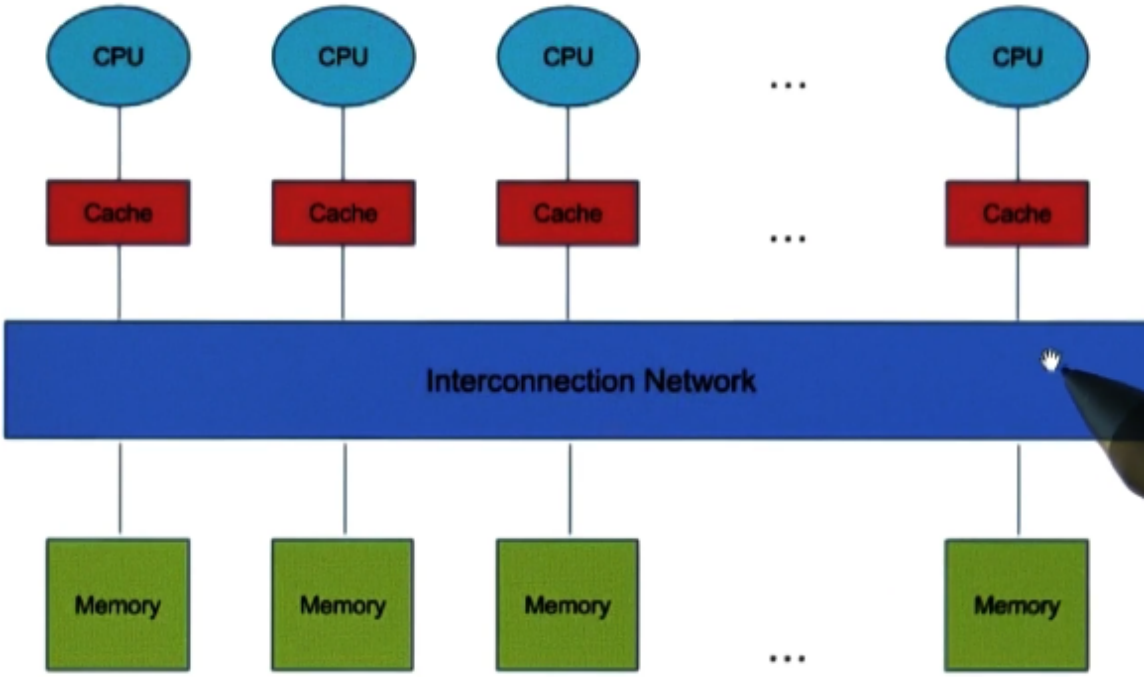
Parallel Systems

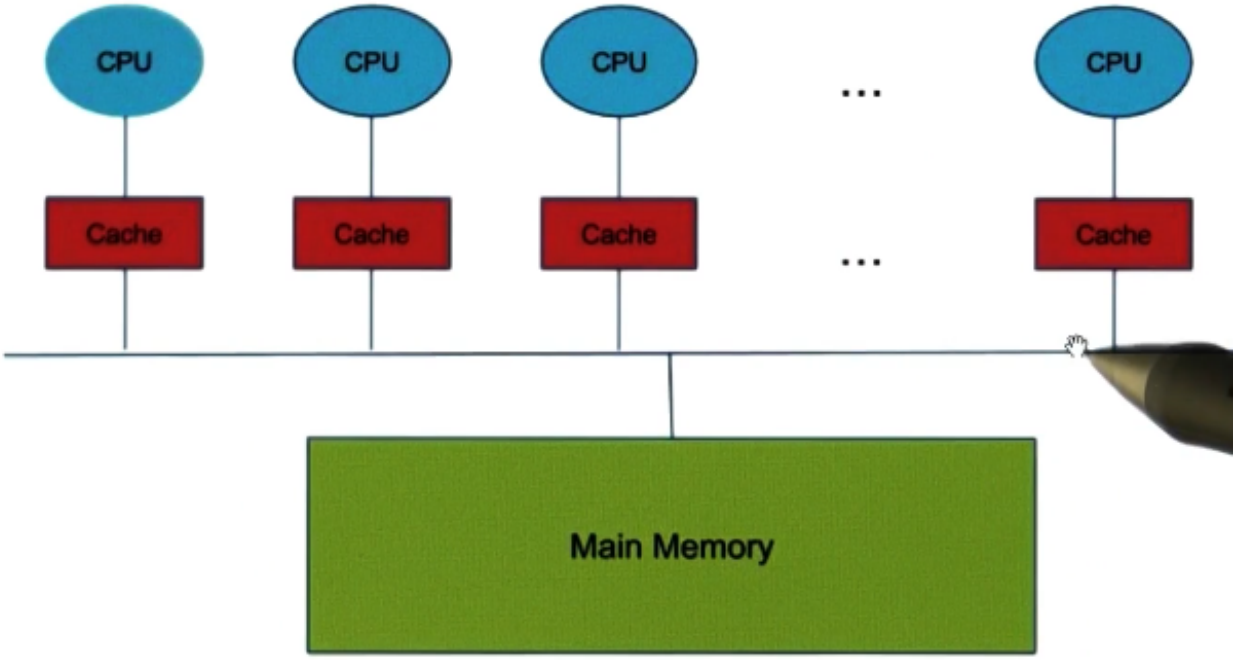
Shared Memory Machines

```
* Shared Memory Machine Models
    1. Dance hall architecture: CPUs on one side, memory on other side of
    interconnection network
```



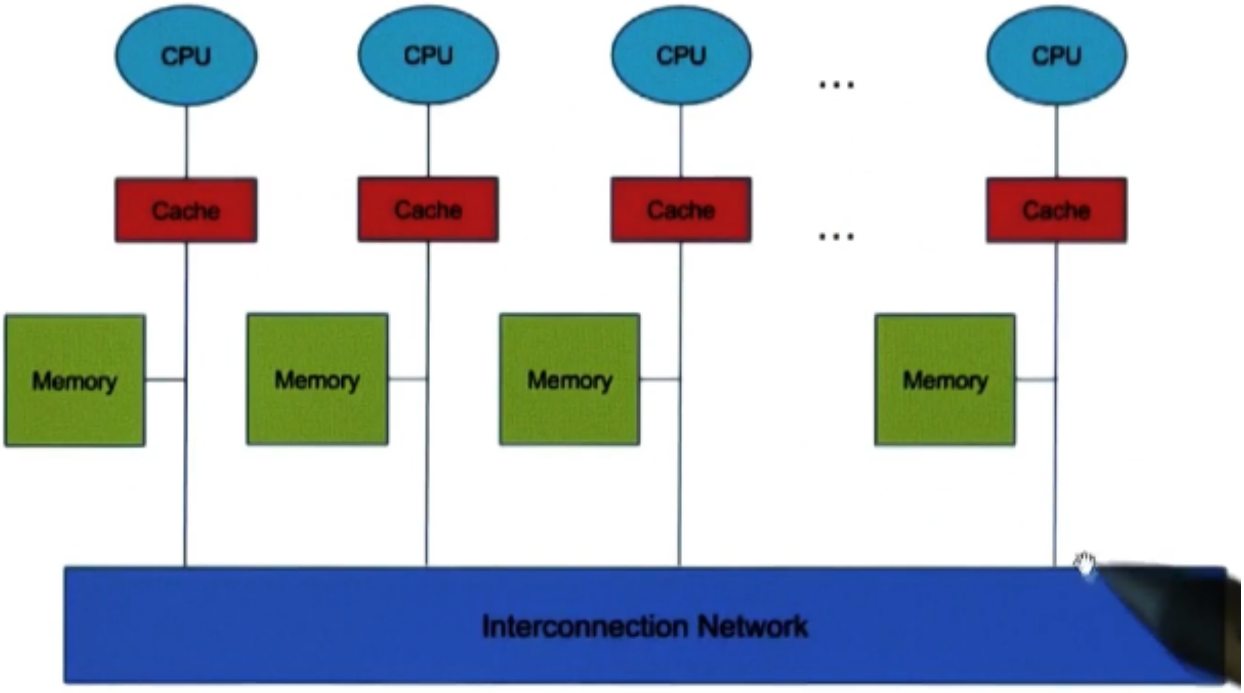
Dance Hall Architecture

```
    2. Symmetric Multiprocessor: Bus connects CPUs to memory
```



Symmetric Multiprocessor Architecture

3. Distributed Shared Memory: Each CPU has nearby memory, but all memory is accessible by all CPUs



Distributed Shared Memory Architecture

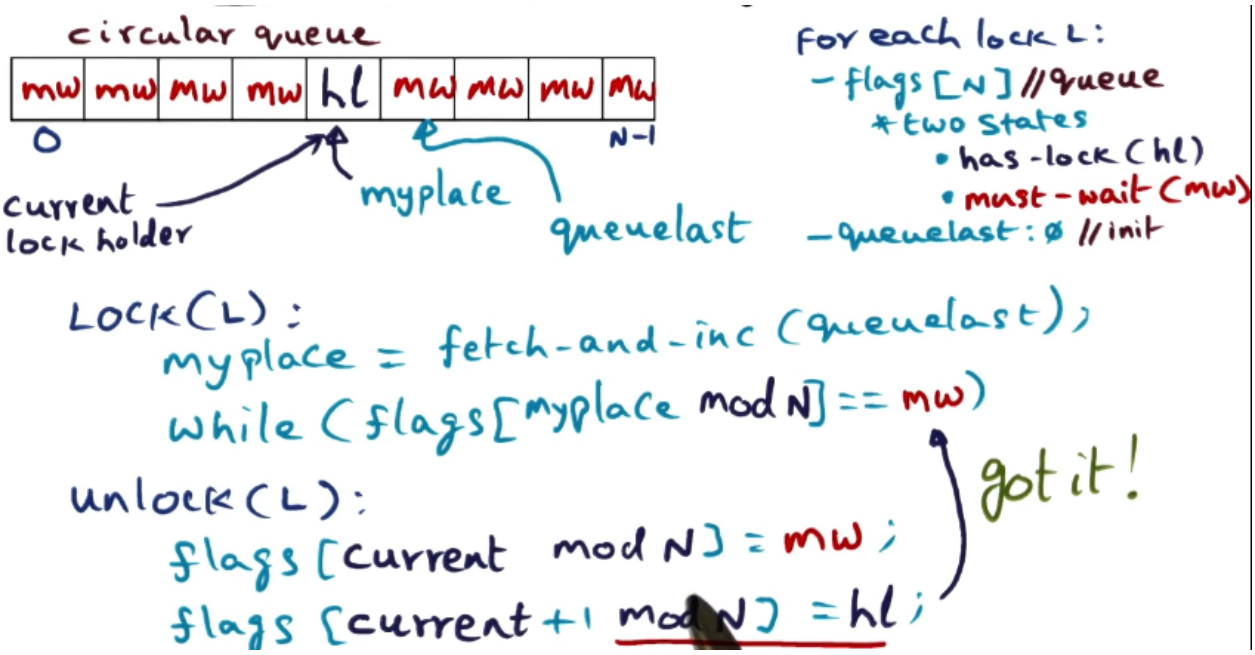
* Shared Memory and Caches
    - Accessing cache: ~2 cycles
    - Accessing memory: 100+ cycles
    - Issue: Memory shared across processors, caches are private to CPU
        + When one CPU writes to its cache, all of the others must be updated to stay in sync
        + Need to maintain consistency
* Memory Consistency Models
    - Sequential consistency - Leslie Lamport, 1977
        + Program order (order in which a process generates memory accesses) is maintained
        + Arbitrary interleaving between processes
* Memory Consistency and Cache Coherence
    - Process P1: a = a + 1; b = b + 1;
    - Process P2: d = b; c = a;
    - What are possible values for d and c?
        + c = d = 0 (Yes)
        + c = d = 1 (Yes)
        + c = 1, d = 0 (Yes)
        + c = 0, d = 1 (Not possible with sequential consistency)
    - Memory consistency: What is the model presented to the programmer?
    - Cache coherence: How is the system implementing the model in the presence of private caches?
    - Non-Cache Coherent Shared Multiprocessor: System software is required to maintain consistency across caches, hardware only provides shared address space (NCC)
    - Cache Coherent Shared Multiprocessor: Hardware does everything (CC)
* Hardware Cache Coherence

    - Write invalidate: If a particular memory location is in multiple caches and one CPU updates it, it will send an "invalidate" signal on the bus to all other CPUs (CPUs know to fetch the new value)
    - Write update: If a particular memory location is in multiple caches and one CPU updates it, the hardware will send an "update" message to change the values in the other CPUs
    - Want to minimize overhead, but grows with number of CPUs
* Scalability
    - Expectation is that performance should increase with more processors
    - However, synchronization overhead increases with number of processors as well
    - "Shared memory machines scale well when you don't share memory" - Chuck Thacker

Synchronization

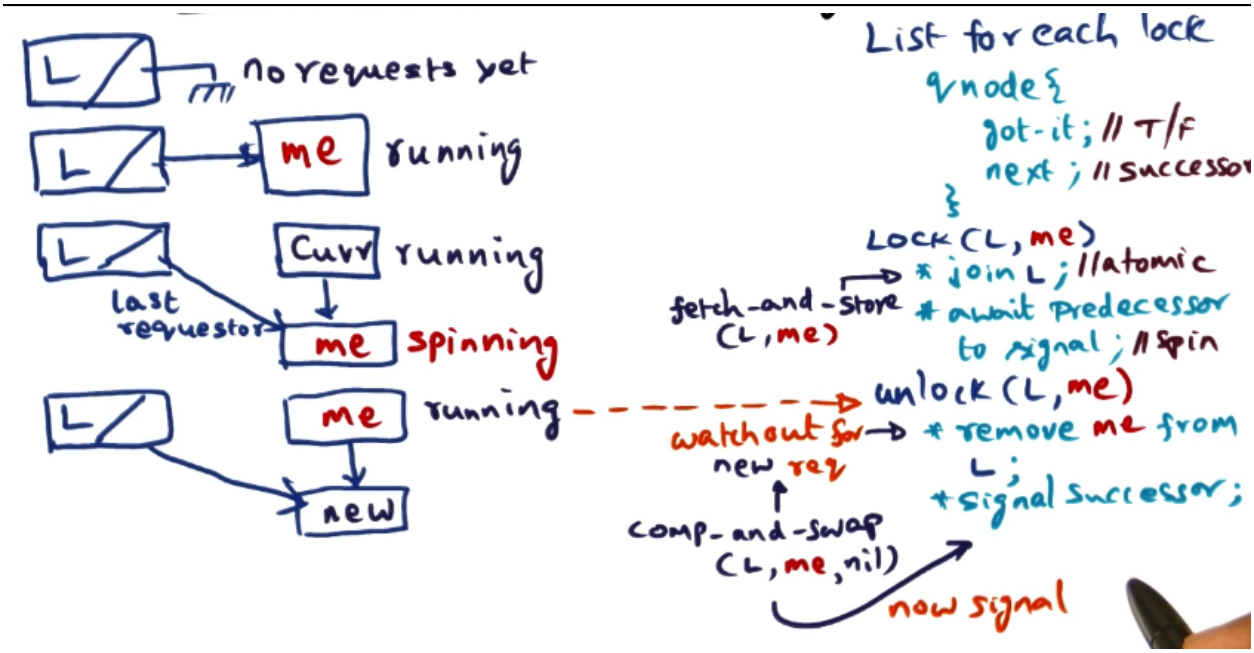
* Synchronization Primitives
    - Exclusive lock: One thread can hold a lock at a time
    - Shared lock: Multiple threads hold lock to read
    - Barrier: Provide guarantee that all threads have reached a particular point
* Atomic Operations
    - Simple atomic read/write primitives are insufficient for implementing a lock
        + Groups of reads/writes are not atomic and could be interleaved
        + Requires "read-modify-write" instruction (RMW)
    - Atomic RMW Instructions
        + test_and_set(L): return current value in L, set L to 1
        + fetch_and_increment(L): return current value in L, increment L
* Scalability Issues with Synchronization
    - Latency: How long does it take to acquire a lock?
    - Contention: When multiple threads are attempting to acquire the lock, how long does it take for one to win?
    - Waiting time: How long will a thread wait to acquire a lock?
        + Application-dependent
* Native Spinlock (spin on test_and_set)
    - lock(L):
        + while(test_and_set(L) == locked);
    - unlock(L):
        + L = unlocked;
    - Problems with Native Spinlock
        + Too much contention (all threads are spinning on the memory location)
        + Does not exploit caches (can't use cached value because multiple threads are sharing the same atomic memory location)
        + Disrupts useful work (contention impedes other threads from doing useful work)
* Caching Spinlock (spin on read)
    - Spin on value in local cache, then try test_and_set when unlocked
    - lock(L):
        + while(L == locked);
        + if(test_and_set(L) == locked) go back;
    - Problems with Caching Spinlock
        + All CPUs will see the update at the same time and try to acquire the lock at the same time (doesn't solve contention issue)

```
        + This results in heavy bus usage; only solves the caching issue
* Spinlock with Delay
    - Delay after lock release
    - while( (L == locked) || (test_and_set(L) == locked) )
        + while( L == locked );
        + delay( d[process_id] );
    - Delay with exponential backoff
    - while( test_and_set(L) == locked )
        + delay(d);
        + d = d * 2;
    - Static delay: Delay based on a predetermined amount of time for each
    process
    - Dynamic delay: Delay increases with successive failures to acquire
    - Dynamic delay means that when contention is low, delay will be short
* Ticket Lock
    - Fairness: Determining which process gets the lock when it becomes
    available; should be the one waiting the longest
    - struct lock{ int next_ticket; int now_serving; };
    - acquire_lock(L):
        + int my_ticket = fetch_and_increment(L->next_ticket);
        + pause( my_ticket - L->now_serving );
        + if( L->now_serving == my_ticket ) { return; }
        + goto( pause );
    - release_lock(L):
        + L->now_serving++;
* Spinlock Summary
    1. read and test_and_set (no fairness)
    2. test_and_set with delay (no fairness)
    3. ticket_lock (fair but noisy)
* Array-based Queueing Lock
    - For each lock L: flags[N] (queue where N = number of processors)
        + Two states (has-lock (HL) and must-wait (MW))
        + Queue array is a circular buffer
    - For each lock L: queuelast = 0; (init)
        + Indicates the starting point of the queue
        + When you want to add yourself to the queue, you add yourself at
        position queuelast
    - lock(L):
        + myplace = fetch_and_increment(queuelast);
        + while( flags[myplace%N] == MW )
    - unlock(L):
        + flags[current%N] = MW;
        + flags[(current+1)%N] = HL;
    - Only one atomic operation (fetch_and_increment)
    - Fair: Next process in queue is guaranteed to get the lock
    - Unaffected by other processes acquiring/releasing lock
    - Cons:
        + Size of queue is as big as number of processors (N)
        + Still allocates this data structure Even if not all processors
        will contend for lock (statically sized)
```



Array-based Queueing Lock

* Linked List-based Queueing Lock
    - Mello-Crummy and Scott (MCS) queueing lock
    - One linked list for each lock (guarantees fairness)
    - struct q_node { bool got_it; q_node* next; };
    - lock(L,me):
        + join L; // atomic, point last_requester in head node to me, point "next" at end of queue to me
        + await precessor to signal; // spin
    - fetch_and_store(L,me): returns what was contained in L and stores me in L; use this to implement the join functionality
    - unlock(L,me):
        + remove me from L;
        + signal successor;
        + If there is no successor, set head node to null; this can cause a race condition if a new request is forming
    - To solve this race condition, use an atomic compare_and_swap when removing "me" from the list. comp_and_swap(L,me,null)
        + if(L == me) { swap(me, null); }
        + Returns true if swapped, false otherwise
        + If it returns false during unlock, wait until my next pointer is not null before finishing unlock
    - Pros:
        + Fair
        + Solves space complexity issue of array-based queueing lock (dynamic instead of static)
    - Cons:
        + Linked list maintainence introduces some overhead that isn't present in the array-based version
        + Performance can decrease if the architecture doesn't implement the fancier atomic operations used



Linked List Queueing Lock

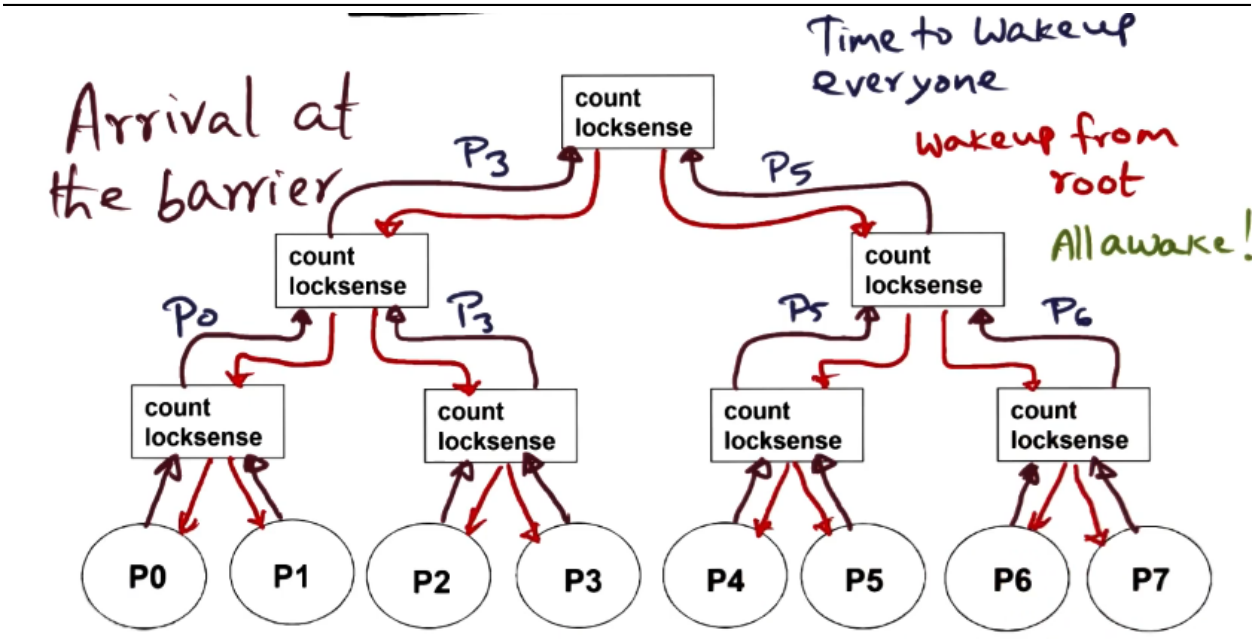
* Algorithm Grading
  - If processor provides a fetch_and_free operation, queueing locks are a good choice. If the processor only provides test_and_set, expontential backoff is a better choice.

Algorithm	Latency	Contention	Fair	Spin	RMW Ops/CS	Space	Signal one lock
Spin on T&S	Low	High	No	Shm	High	Low	No
Spin on read	Low	Med	No	Shm	Med	Low	No
Spin with delay	Low	Low	No	Shm	Low	Low	No
Ticket lock	Low	Low	Yes	Shm	Low	Low	No
Array Queue Lock	Low	Low	Yes	Pvt	1	High	Yes
LL Queue Lock	Low	Low	Yes	Pvt	1 (max 2)	Med	Yes

Communication

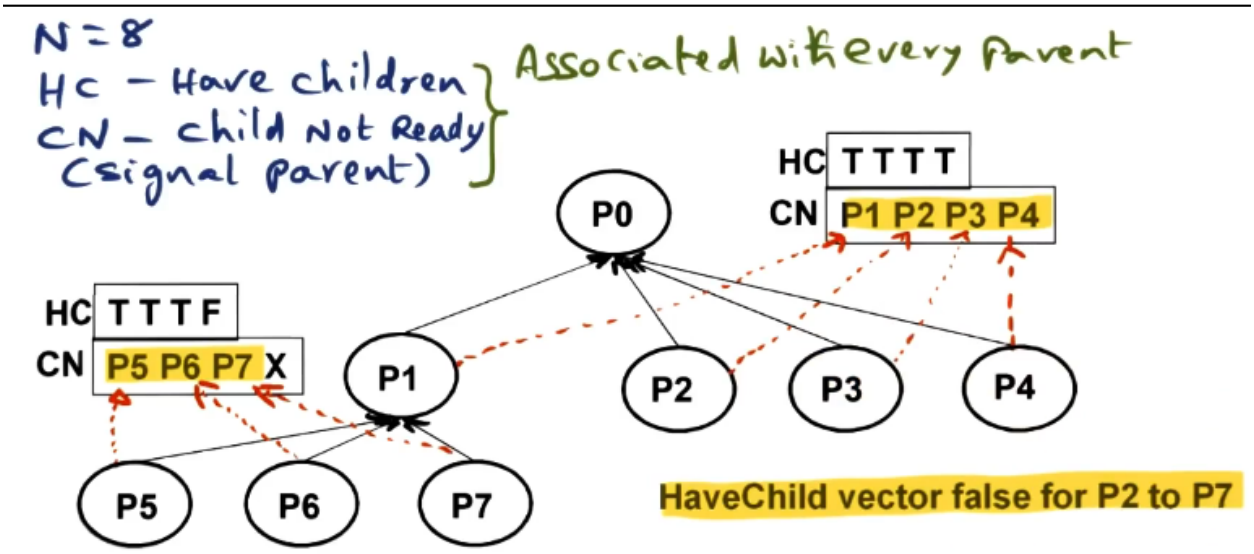
* Centralized (Counting) Barrier
  - Threads wait for all others to arrive at one point before proceeding
  - Count is initialized to N
  - When a thread arrives at the barrier, it atomically decrements and waits for the count to become 0
  - count = N; // init
  - atomic_decrement(count);
  - if(count == 0)
    + count = N;
  - else
    while(count > 0);
    while(count != N); // this fixes the issue cited below
  - Issue: Before the last processor sets count to N, other processors may race to the next barrier and proceed incorrectly
    + Need to wait for count to reset before proceeding

```
* Sense Reversing Barrier
    - Goal is to eliminate the while(count > 0) spinning loop
    - Use a shared sense variable to indicate which side of the barrier
    we're on
    - True indicates barrier hasn't happened yet, false indicates it has
    - count = N; // init
    - sense  true;
    - atomic_decrement(count);
    - if(count == 0)
        + count = N;
        + sense = false
    - else
        while(sense);
    - Issues: High contention on network for single shared "sense" variable
* Tree Barrier
    - Build a tree of sense variables such that fewer processes are sharing
    the same variable
    - Once one barrier is met, move up the tree to the next barrier
    - When a processor arrives at a variable, it decrements the count
        + If 0, must recurse (repeat at the next level)
        + If 1, spin on the local locksense
    - When the last process reaches the root, it will flip the locksense flag
        + Then, recurse back down the tree, flipping locksense at each level
    - Issues
        1. locksense variable is dynamically determined by arrival pattern
        2. Lots of contention if number of processes is large
        3. If not cache coherent, the spin occurs on remote memory
    - Non-Uniform Memory Architecture (NUMA): Accessing local memory is
    faster than accessing remote memory
        + Same as distributed shared memory architecture
```



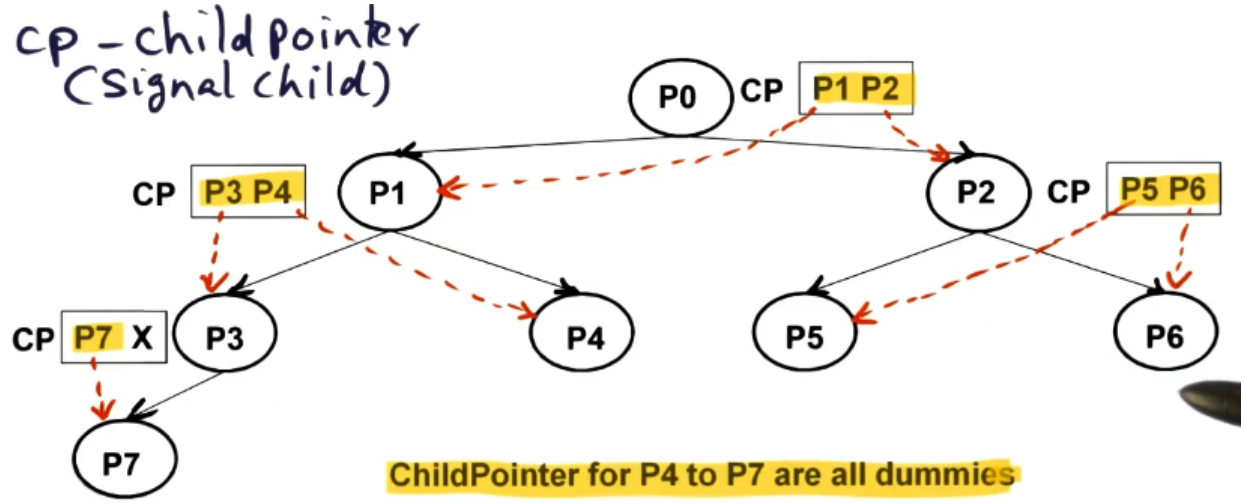
Tree Barrier

* MCS Tree Barrier (4-ary Arrival Tree)
    - Two data structures for each node
        + HC: Have children
        + CN: Child not ready (signal parent)
    - Chose 4 children for performance reasons (tested)
    - In a cache coherent multiprocessor, parent only has to spin on one word (memory location)



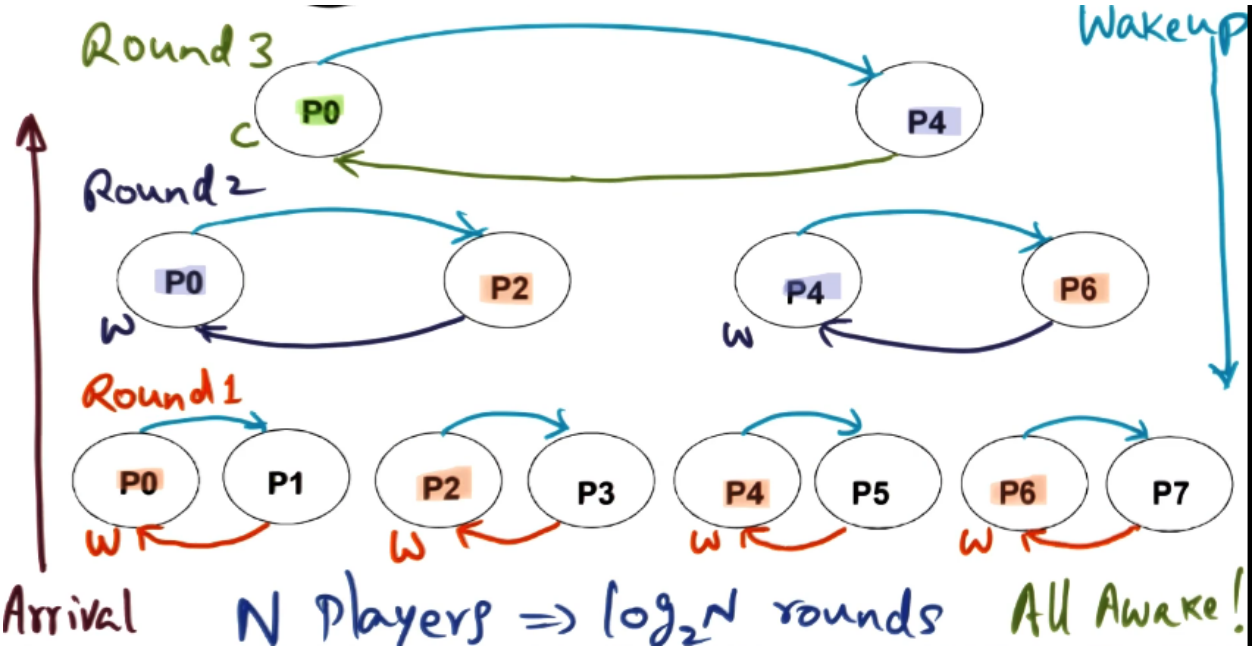
MCS Tree Barrier (4-ary Arrival)

* Binary Wakeup
    - One data structure
        + CP: Child pointer
    - Minimizes the path to furtherest child
    - Each parent is spinning on a statically determined location
    - Uses CP (child pointer) to signal child
    - Because everything is statically assigned, this solves the issue of accessing remote memory



MCS Tree Barrier (Binary Wakeup)

```
* Tournament Barrier
    - N players, log2(N) rounds
    - This means there are only two "contestants" in each round
    - The "winner" is predetermined so the memory locations to reference are
    static (match fixing)
        + This is especially important in non-cache coherent architectures
    - When the winner finishes, it waits for the loser to finish
        + If it is already finished, it proceeds
    - When the entire tree is finished, the winner tells the loser
        + This process recurses back through the tree
    - Tree barrier vs Tournament barrier
        + Spin locations are statically determined in tournament barrier
        + Tournament barrier can be implemented with only atomic read/write;
        tree barrier requires fetch_and_free
        + Total amount of communication is the same: O(logN)
        + Tournament barrier works even in the absence of physical shared
        memory (simply message passing)
    - MCS barrier vs Tournament barrier
        + Tournament can't exploit spatial locality; MCS takes advantage of
        children occupying the same cache line
        + Neither need fetch_and_free operation
        + Tournament barrier works even in the absence of physical shared
        memory (simply message passing)
```



Tournament Barrier

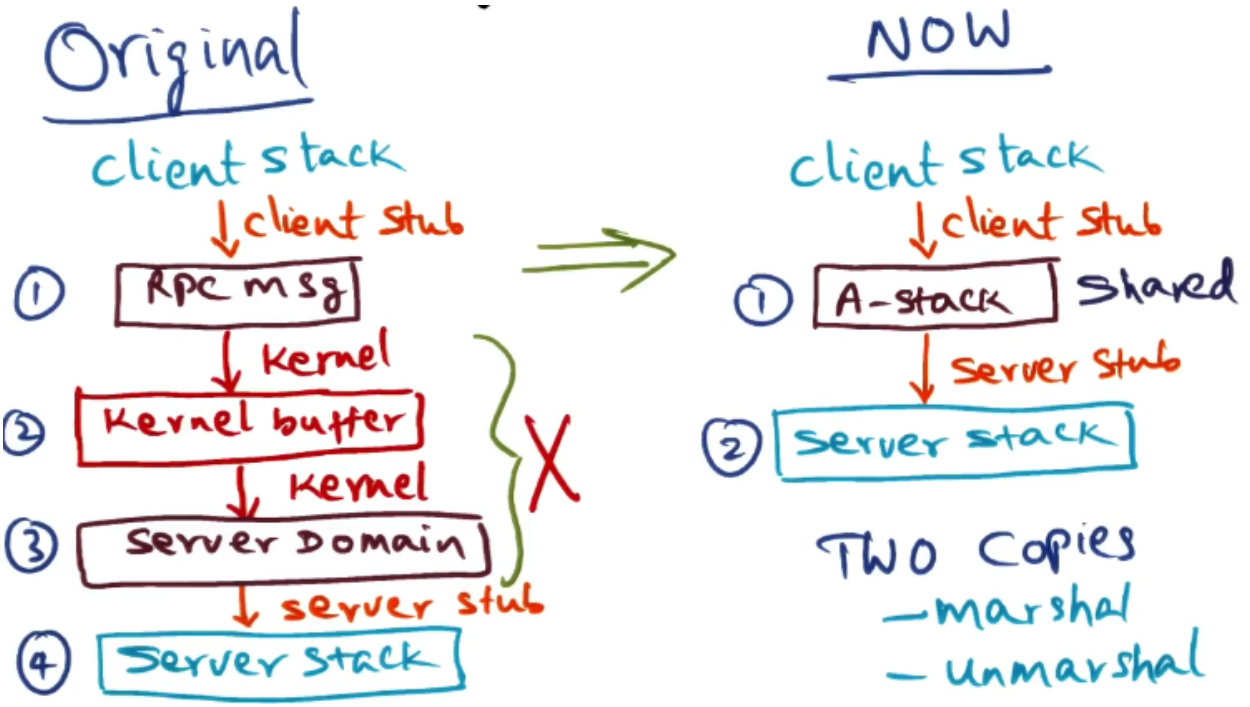
```
* Dissemination Barrier
    - Not pairwise communication, information diffusion
    - N doesn't need to be a power of 2
    - For each round(k) when N=5:
        + Pi sends a message to P(i+2^k)%N
        + For round 0: 0->1, 1->2, 2->3, 3->4, 4->0
        + For round 1: 0->2, 2->4, 4->1, 1->3, 3->0
        + For round 2: 0->4, 4->3, 3->2, 2->1, 1->0
        + For round 3: 0->3, 3->1, 1->4, 4->2, 2->0
        + O(N) communication events per round
    - ceil(log2(N)) rounds are required for barrier completion
        + Guarantees that every process has sent and received a message to
        every other process
    - No distinction between arrival and wakeup
    - Static determination of memory location
    - Pros:
        + No hierarchy
        + Works for NCC and clusters
        + Communication complexity is O(NlogN) versus O(logN) in MCS and
        tournament barriers
* Performance Evaluation
    - Spin Algorithms
        + Spin on test_and_set
        + Spin on read
        + Spin with delay
        + Ticket lock
        + Array queue lock
        + List queue lock
    - Barrier Algorithms
        + Counter
        + Tree
```

        + MCS Tree
        + Tournament
        + Dissemination
    - Parallel Architectures
        + CC SMP
        + CC NUMA
        + NCC NUMA
        + MP Cluster
    - The best performance depends on the architecture; the only way to know is to implement and test
        + Trends are more important than absolute numbers as hardware changes

Lightweight RPC

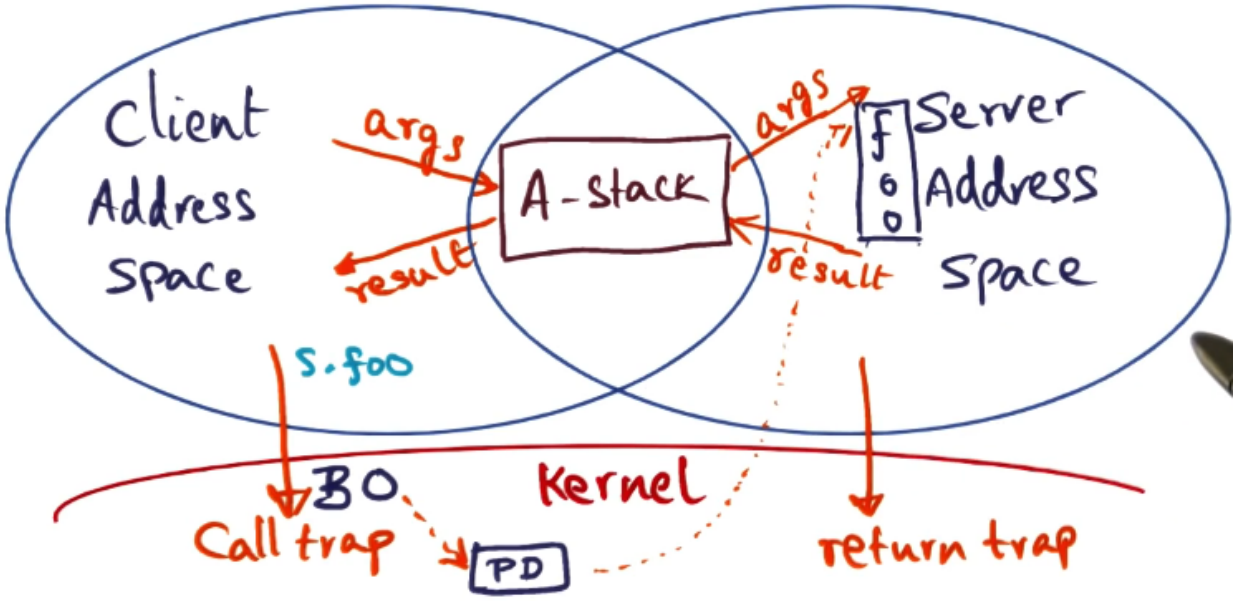
* Remote Procedure Calls
    - RPC allows for a client/server architecture across address spaces
    - This provides safety between processes (one crashing won't crash the other), but costs some performance, even on the same machine
* RPC vs Local Procedure Call
    - In a local call, everything happens at compile time
    - RPC:
        1. Caller traps into kernel (call trap)
        2. Kernel validates call
        3. Kernel copies arguments into kernel buffers from client address space
        4. Kernel locates server procedure
        5. Kernel copies arguments from kernel buffer to address space of the server
        6. Kernel schedules server to run the procedure
        7. When server is complete, it traps again (return trap)
        8. Kernel copies results from address space of server to kernel buffers
        9. Kernel copies results from kernel buffers to client address space
        10. Kernel schedules client to continue running
    - RPC results in two traps and two context switches, plus one execution
        + This results in four copies between user/kernel address space
    - Ideally, we'd like to get the kernel out of the way
* Copying Overhead
    - In each client/server or server/client switch, a total of four copies occur; two in the user address space and two in the kernel address space
* Reducing RPC Binding Overhead
    - Set up (binding) is a one time cost -> one time cost
    - Server publishes a "procedure descriptor" that contains:
        + Entry point in server address space
        + Argument stack size
        + # of simultaneous calls it can support (multithreading)
    - At binding, kernel allocates a buffer of shared memory and maps it into the address space of both the client and server
        + The size is specified by the server (argument stack size)
        + After this mapping is complete, the server can get out of the way
        + Client gets a "binding object" that it can present to the kernel to verify it is authenticated to make remote calls on the server
    - Kernel mediation (binding) only happens at the first call
* Reducing RPC Call Overhead

- The client address space will copy arguments into the A-stack
    + Can only be done by value, not reference (local addresses will have no meaning to the server)
- Instead of copying all of the data, the kernel can borrow the client thread and doctor it to run on the server address space
    + Client is blocked after making the call
    + Execute the client thread in the address space of the server
    + Set program counter, address space descriptor, and stack
    + Kernel allocates an execution stack (E-stack) that the server can use to complete the RPC
    + Arguments/results are passed through the A-stack
- This eliminates the serialization and two of the copies
    + Now, only one copy to marshal the data into the A-stack and one to unmarshal the data into the destination address space



Copying Overhead

* Reducing RPC Overhead Summary
    - Explicit costs: Call trap, switching domain to server address space, return trap
    - Implicit costs: Loss of locality



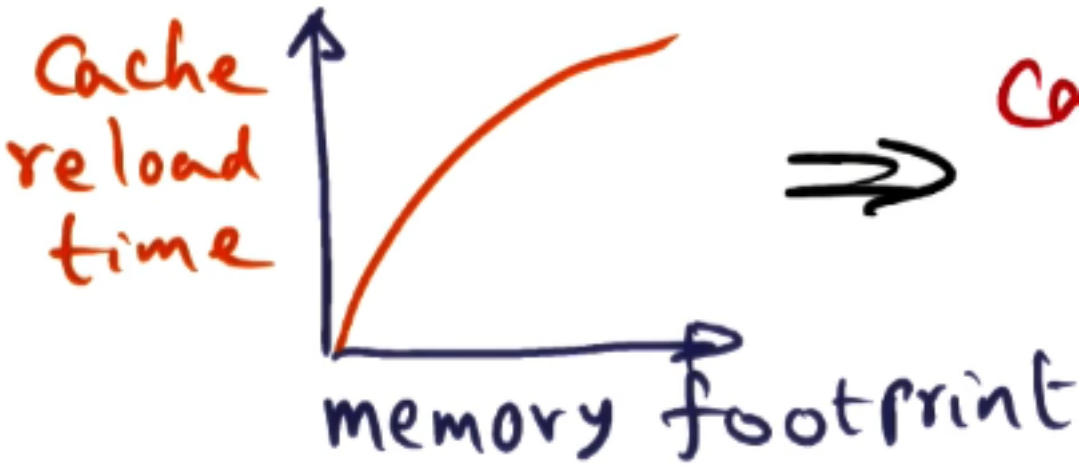
Making RPC Cheap

* RPC on SMP
    - Exploit multiple processors
        + Pre-load server domain in a particular processor (one CPU dedicated to preserving the server's address space)
        + Keep caches warm
    - Can map the server to multiple CPUs to service more requests
    - Taken mechanism typically used for distributed systems and made it efficient for providing services locally
        + Provides safety through putting each service in its own protection domain

Scheduling

* How should the scheduler choose the next thread to execute?
    - FCFS: Most fair
    - Highest static priority: Give priority to most important
    - Highest dynamic priority: Give priority to I/O bound tasks
    - Memory contents in cache: Will have the best performance
* Cache Affinity Scheduling
    - Two orders of magnitude between L1 cache and memory
    - Makes sense to schedule thread on the same processor after it's been descheduled because its contents might still be in cache
        + However, if another thread was scheduled in between, the original thread's contents won't be in cache
* Scheduling Policies
    - FCFS: Pick the thread that entered the runnable queue first
        + Ignores affinity for fairness
    - Fixed processor: Ti will always run on the same processor
        + Initial processor choice may depend on load balance
    - Last processor: Ti will always run on the same processor as it ran on previously

        + If there's no previous thread, it will still pick something
    - Minimum intervening: Pick the processor with highest affinity for Ti
        + Requires keeping track of the most data
* Minimum Intervening Policy
    - Calculate an affinity index for each thread for all processors
    - Track the affinity for thread Ti across all processors
    - Provides the best chance that its memory contents will be in cache
    - This can be prohibitively espensive for many threads/processors
        + Limited MI: Only keep affinity index for the top N processors
    - Smaller index == Higher affinity
* Minimum Intervening Plus Queue Policy
    - Keep a queue of threads for each processor
    - Then, if the queue is long, reschedule the thread elsewhere
        + If time spent in queue > time to reload data from memory
        + The threads ahead in the queue will also pollute the cache, so the
        affinity will be lower when Ti actually runs
* Summarizing Scheduling Policies
    - FCFS: Ignores affinity for fairness
    - Fixed processor: Ti always on Pfixed (focus on affinity)
    - Last processor: Ti on Plast (focus on affinity)
    - Minimum intervening: Ti -> Pj(min(I)) (focus on cache pollution)
    - MI plus queue: Ti -> Pj(min(I+Q)) (focus on cache pollution)
    - Fixed/last processor are thread centric
    - MI/MI plus queue are processor centric
* Implementation Issues
    - Queue-based
        + Global queue of threads to be run (makes logical sense for FCFS)
        + Affinity-based local queues (one per process, depends on specific
        policy)
        + In affinity-based, if one processors queue is empty, it might take
        work from another "work stealing"
    - Priority queue
        + Ti's priority = base priority + age + affinity
        + This determines the position in the queue
* Performance
    - Figures of Merit
        + Throughput: How many threads get executed per unit time
        + Response time: How long does a thread take to complete
        + Variance: How much does the completion time vary
        + Throughput is system-centric, response time and variance are user-
        centric metrics
    - Time to reload cache increases with memory footprint
        + Indicates that cache affinity is important
        + However, under heavy load, a fixed processor strategy might give
        better performance due to cache pollution
    - Procrastination: processor idles when its queue is empty before
    stealing work from another queue
        + If a thread enters the queue during the idle time, it will have
        better affinity than the thread stolen from another queue



Memory Footprint vs Cache Reload Time