L04b. Synchronization

Basic Synchronization Primitives:

- Lock:
  - A primitive to protect shared data structure in multi-threaded operations.
  - Types of locks:
    1. Mutual exclusion: Can be used by one thread at a time (e.g. protect a data structure while writing to it).
    2. Shared lock: Allows multiple threads to access the data at the same time (e.g. Multiple threads reading the same data at the same time, while preventing the modification of this data).
- Barrier:
  - A primitive to ensure that multiple threads reached a particular point in terms of computations.
  - It blocks threads that reach this point until the other threads (those that use the same barrier) reach the same point, and then all threads can continue execution.
  - The access time from any CPU to the memory is the same.

Atomic Operations:

- If we have a lock (e.g. global variable) that is shared between multiple threads, each thread will need to execute read/write operations to acquire/release this lock.
- Read/write operations are atomic, but the whole process of checking the lock, blocking on it if not available and then acquiring it is not atomic. This can cause problems while acquiring/releasing the lock.
- We need a new semantic atomic instruction to use with the lock.
- Atomic RMW (Read – Modify – Write) instructions:
  - Test-and-set: Takes a specific memory location, returns its current value and sets it to one.
  - Fetch-and- increment: Takes a specific memory location, returns its current value and increments it by one, or any given value.
  - These instructions called in general “Fetch-and-ϕ”, where ϕ is any given operation.

Scalability Issues with Synchronization:

- Latency: The overhead of locking/unlocking the lock.
- Waiting time: How long to wait till a thread can acquire the lock.
- Contention: If there’re multiple threads spinning on a specific lock, we expect an overhead on the system bus. This overhead increases time consumed from the point a thread releases that lock to the point it’s acquired by another thread.

Spinlocks:

- Naïve Spinlock (spin on test-and-set): A thread that is waiting for a lock to be released will spin on the lock (infinitely execute test-and-set) till it's released. This type has three issues:
  - Too much contention.
  - Doesn't exploit caches.
  - Disrupts useful work.
- Caching Spinlock (spin on read): When we have a system where the architecture ensures cache coherence, the threads that are waiting for a lock to be released can spin on the cached value of the lock. This removes the contention from the communication network. The problem with this type is once the lock is released, all the spinning threads will try to execute test-and-set to acquire the lock, which produces a huge overhead on the bus.
- Spinlock with delay: Whenever a lock is released, each pending processor will wait for a specified delay time before trying to acquire the lock.
  - Each processor will have its own delay.
  - In some situations, the lock isn't blocking a lot of threads and can be acquired immediately without causing much contention. To avoid unnecessary delays in this case, we can use dynamic delays. When the lock is released, the processor waits for a certain amount of time, and then tries to acquire the lock. If the lock was acquired during this delay, this means that is has high contention, then the processor will double its delay time.
  - This algorithm is compatible with non-cache coherent systems because it doesn't use the cache.
- Ticket lock: This lock structure maintains data fields to determine which thread has the lock now (now_serving), and which one should acquire the lock next (next_ticket).
  - If a thread wants to acquire the look, it executes (fetch-and-increment) on the next_ticket field. Then waits till the now_serving value of the lock is equal to the ticket value of the thread.
  - This structure ensures fairness between threads. Each thread gets a chance to acquire the lock.
  - It adds overhead to the network for maintaining the lock information.

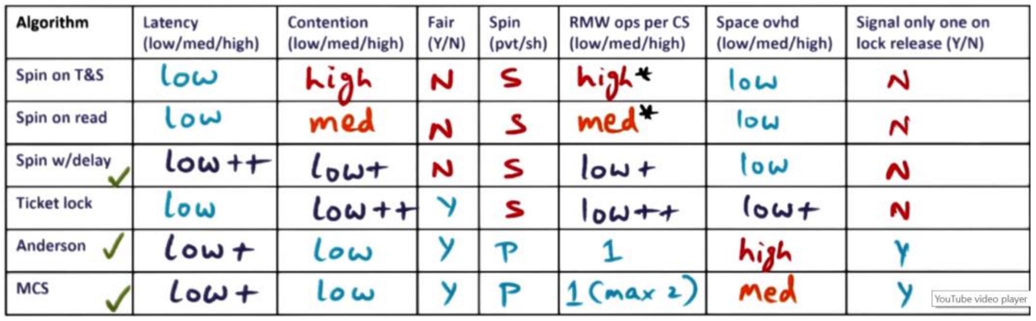
Array-based Queuing Lock:

- The thread that is releasing the lock will signal another pending thread to acquire the lock.
- The lock maintains two data structures:
  1. A circular queue of flags, where each processor either has the lock (has_lock) or waiting for it to be released (must_wait).
  2. A "queue_last" variable to specify the next free spot in the queue.
- The pending thread will spin on its position in the queue till it can jump to (has_lock) and acquire the lock.
- When a thread is releasing the lock, it will signal the next thread in the queue, by setting the next position in the flags array to (has_lock).

- Advantages:
  - There’s only one atomic operation a thread has to execute to acquire the lock.
  - The lock is fair. It maintains the order of requests.
  - The spin position of each processor (must_wait) ensures independence between different threads.
  - Whenever a lock is released, exactly one thread is signaled to acquire the lock. This decreases contention.
- Disadvantage: The problem with this algorithm is that the size of the queue will depend on the number of processors. Since the lock maintains a static data structure, it must create a structure that can accommodate for the worst-case scenario (all processors are waiting for the lock). If we have a large-scale multi-processing system, this can be a problem (high space complexty).

Linked List-based Queuing Lock:

- To avoid the space complexity of the Array-based Queuing Lock, a linked list can be used as a representation for the queue, so the size of the queue will be exactly equal to the dynamic sharing of the lock.
  - This means that the size of the queue will be equal to the number of processors that are actually using the queue right now, not all the processors in the system.
- The lock data structure is a dummy node and will be the initial node of the linked list.
- Each thread gets a node with two fields:
  - got_it: To indicate that the thread acquired the lock.
  - next: To point to the next thread in the queue.
- Whenever a new thread wants to join the queue:
  - It has to change the “next” pointer in the last requester’s node to point to itself.
  - Set the “next” point of its node to null.
  - Spin on “got_it”.
- Joining the queue is an atomic operation (fetch-and-store).
- When a thread releases the lock, it should remove itself from the list and signal the next thread. If there’s no thread waiting for the lock, the releasing thread should set the “next” pointer to null.
- A problem can arise if a new thread tries to acquire the lock at the same time the releasing thread sets the pointer to null. A race condition will happen.
- To avoid this issue, the releasing thread should do a comp-and-swap atomic operation. It checks if the pointer is still pointing at itself and if yes it sets the pointer the null.



Algorithm	Latency (low/med/high)	Contention (low/med/high)	Fair (Y/N)	Spin (pvt/sh)	RMW ops per CS (low/med/high)	Space ovhd (low/med/high)	Signal only one on lock release (Y/N)
Spin on T&S	low	high	N	S	high*	low	N
Spin on read	low	med	N	S	med*	low	N
Spin w/delay ✓	low++	low+	N	S	low+	low	N
Ticket lock	low	low++	Y	S	low++	low+	N
Anderson ✓	low+	low	Y	P	1	high	Y
MCS ✓	low+	low	Y	P	1 (max 2)	med	Y

YouTube video player