Queued Spinlocks in the Linux Kernel

Motivation, Design and Implementation

Gautham R. Shenoy
<gautham.shenoy@amd.com>

AMD
together we advance_

Spin Locks

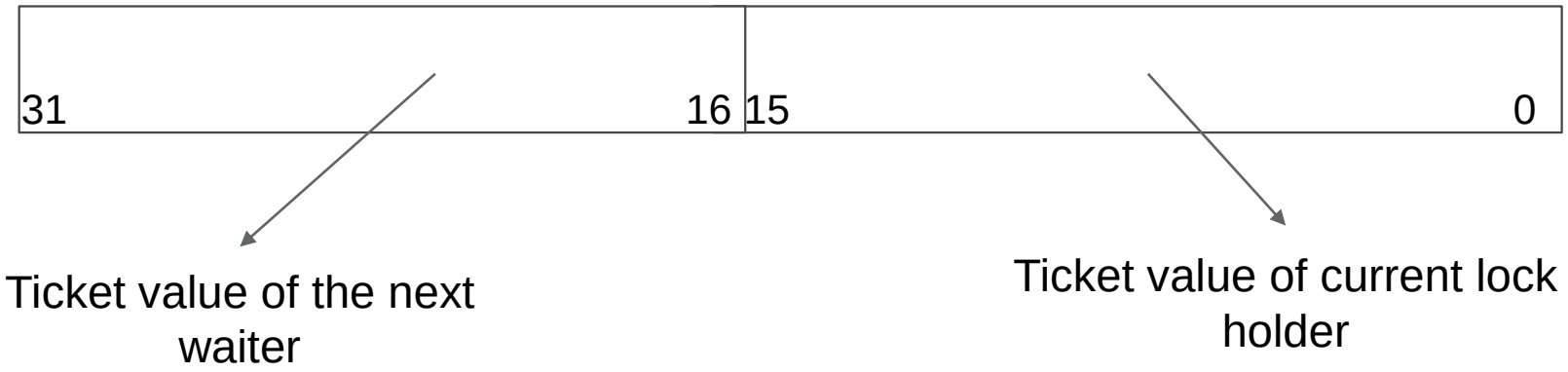
- Used to serialize access to shared-data in short critical sections.
- As the name suggests, when the lock is held by some other thread, the waiting threads “spin” in a tight-loop until the lock is released.
- Three implementations
  - Classic Spin Locks
  - Ticket Spin Locks
  - Queued Spin Locks

Overview: Classic Spin Lock Implementation

- A 32-bit word modelling the lock variable.
- Value == 0 implies the lock is in unlocked state.
- spin_lock(): Tries to atomically compare-and-exchange the lock variable from 0 to 1.
  - If compare-exchange is successful, then the lock has been successfully acquired.
  - Else, it spins until the value becomes 0 before trying the atomic compare-exchange again.
- spin_unlock():
  - Resets the value of the lock variable to 0.

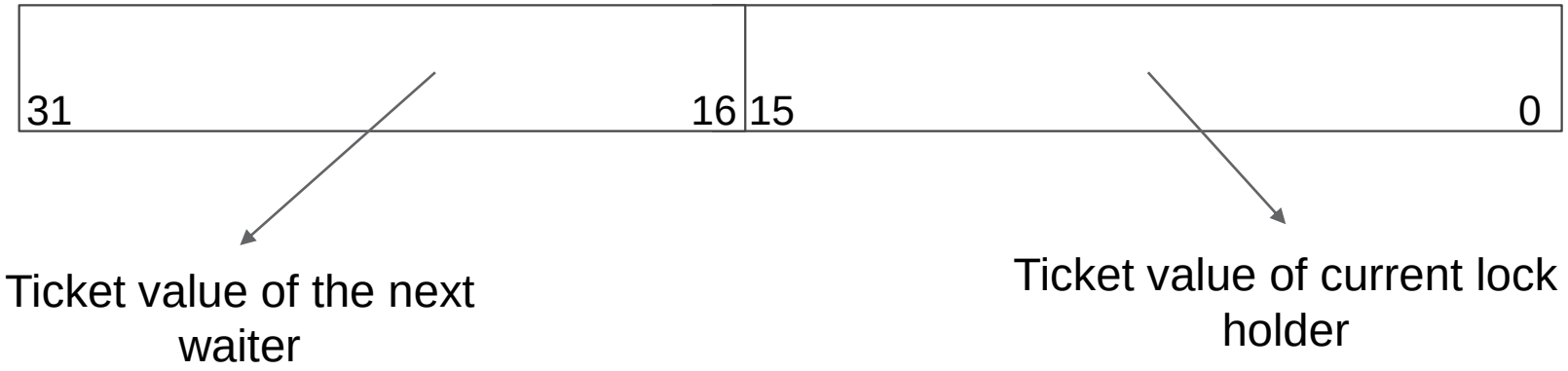
Overview: Ticket Spin Lock implementation

- A 32-bit word modelling the lock variable. Contains 2 parts



Overview: Ticket Spin Lock implementation

- A 32-bit word modelling the lock variable. Contains 2 parts

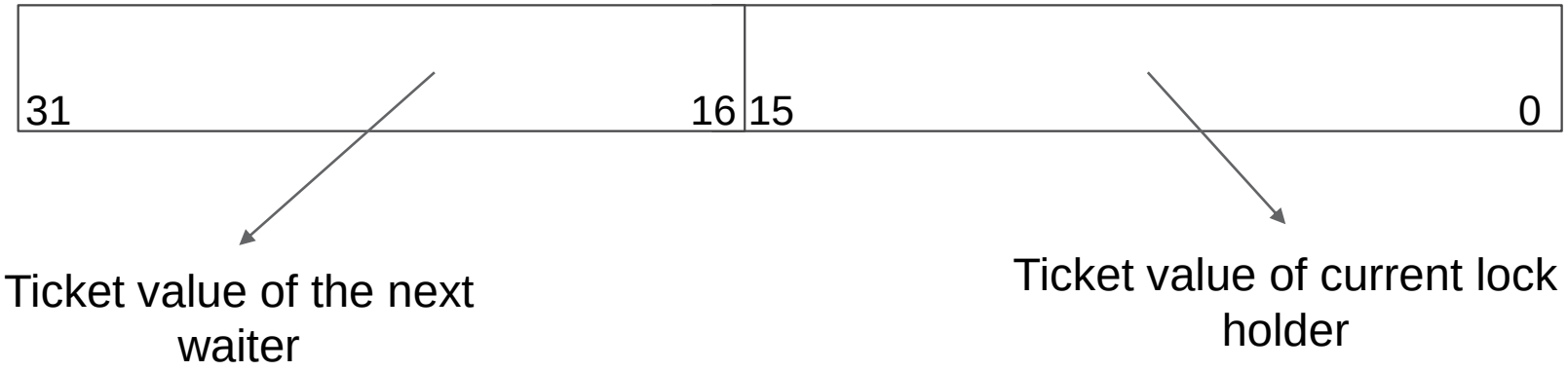


spin_lock():

- old_val = atomic_fetch_add(1 << 16, lock)
- my_ticket = old_val >> 16
- current_ticket = (u16) old_val;
- If (my_ticket == current_ticket) Yay! I got the lock!
- Else spin until till lock[15:0] == my_ticket.

Overview: Ticket Spin Lock implementation

- A 32-bit word modelling the lock variable. Contains 2 parts



spin_unlock ():

- lock[15:0] = lock[15:0] + 1

Queued Spin Locks

Problem with classic spinlocks implementation:

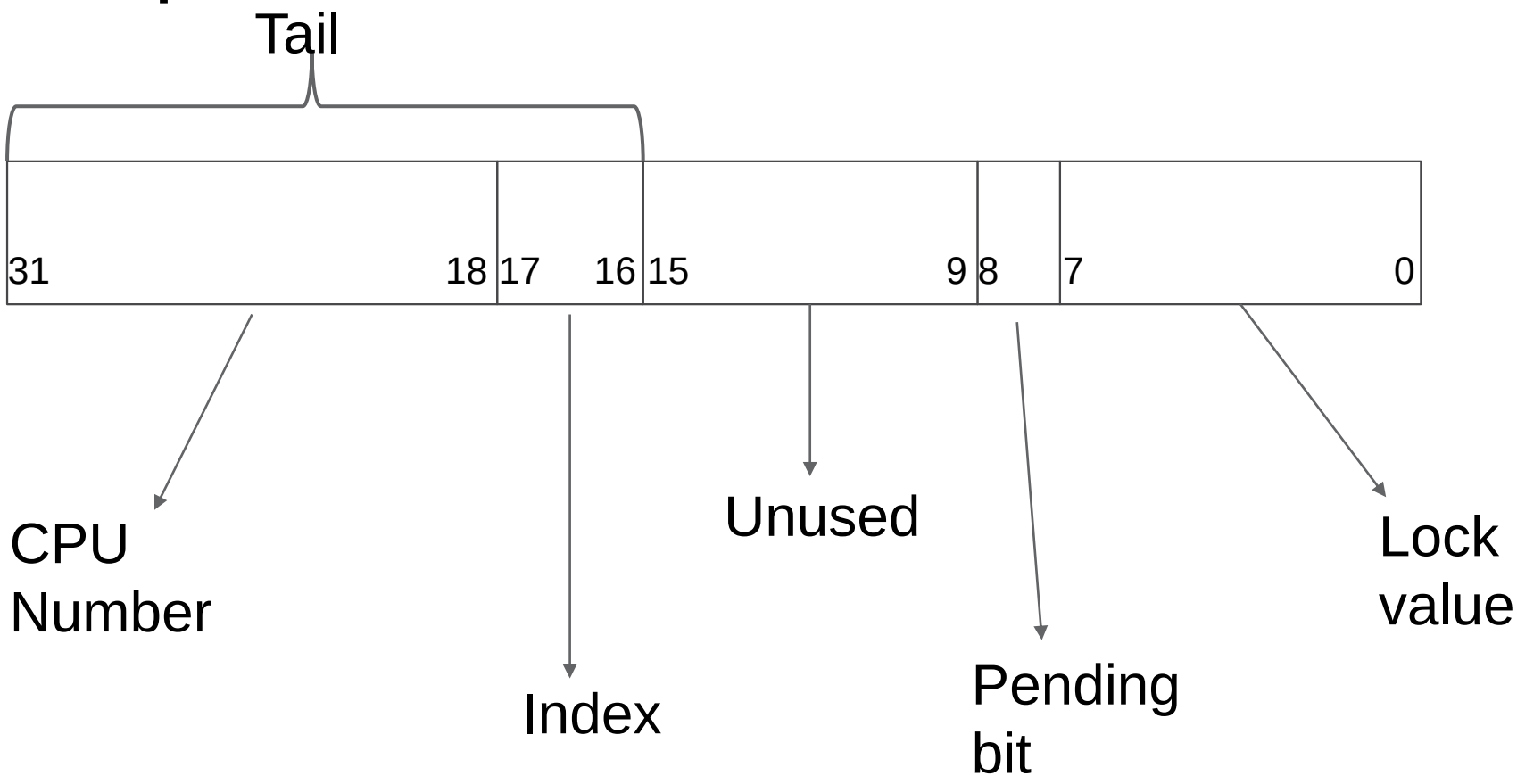
- **It is unfair:** A CPU that came in first and started spinning waiting on the lock may not necessarily get the lock if there is some other CPU in contention. It really depends on who sees the LOCK → UNLOCK transition first.

- **Cacheline Bouncing:** It causes cacheline bouncing of the line containing the lock variable when all the contending CPUs spin on the lock. Problem exacerbates on systems with large number of CPUs which try to contend on the lock.

Ticketing spinlock implementation can address the unfairness issue, but it won't address the cacheline bouncing issue.

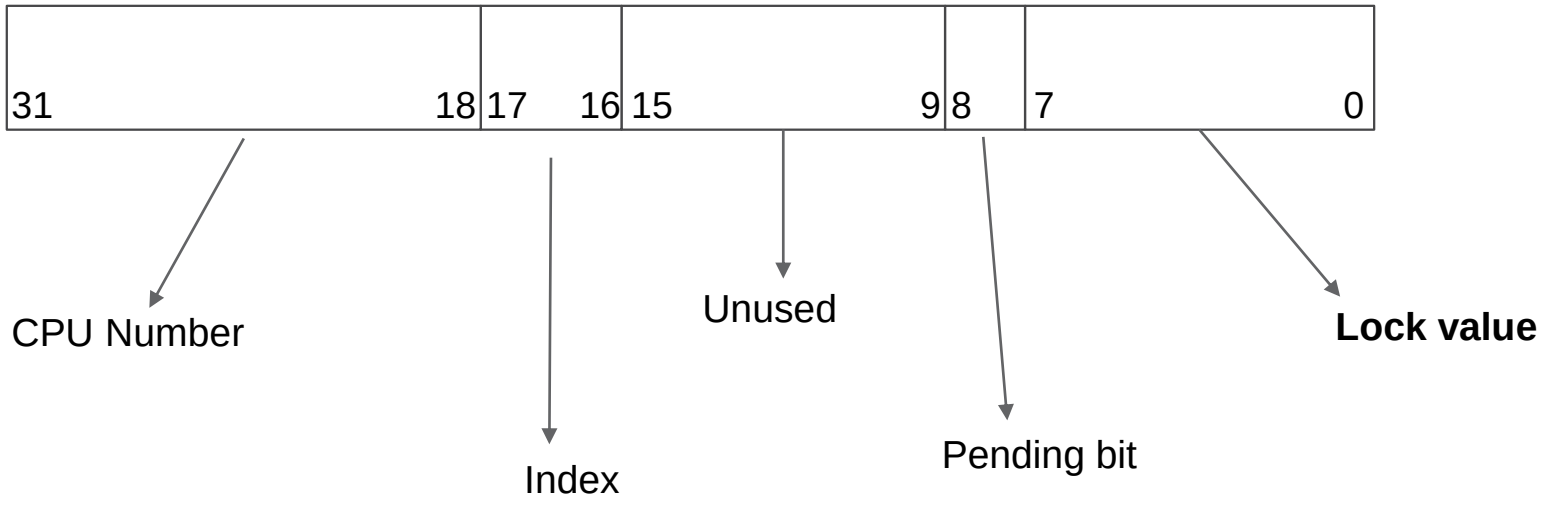
Hence Queued Spin Locks!

Queued Spin Locks



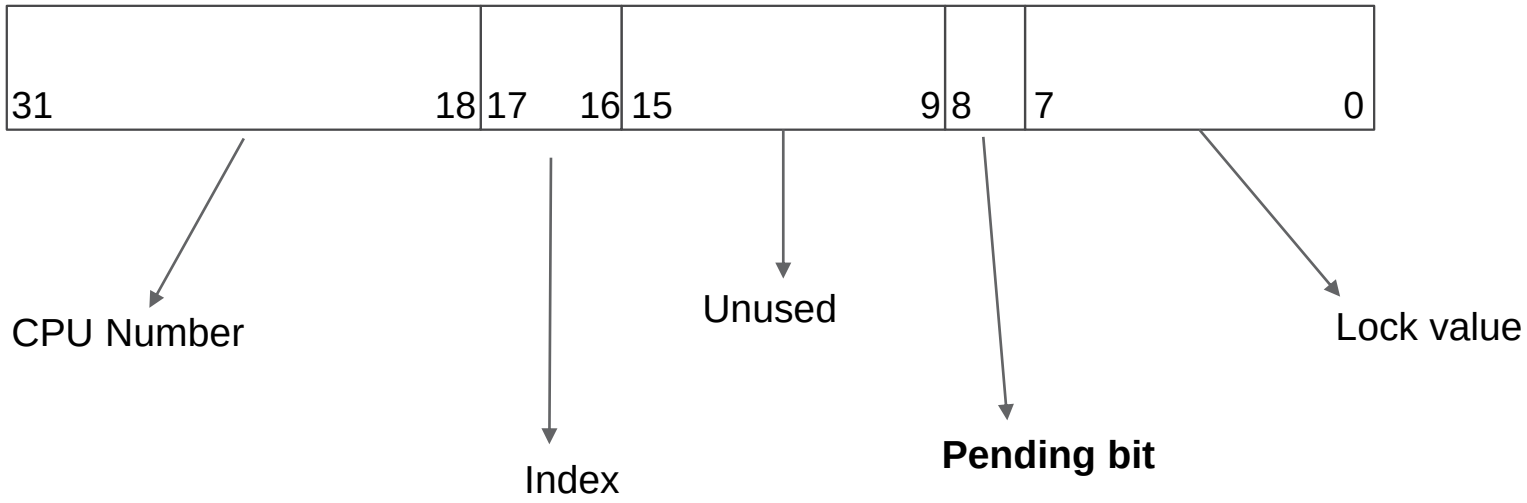
qspinlock variable. 32 bits

Queued Spin Locks



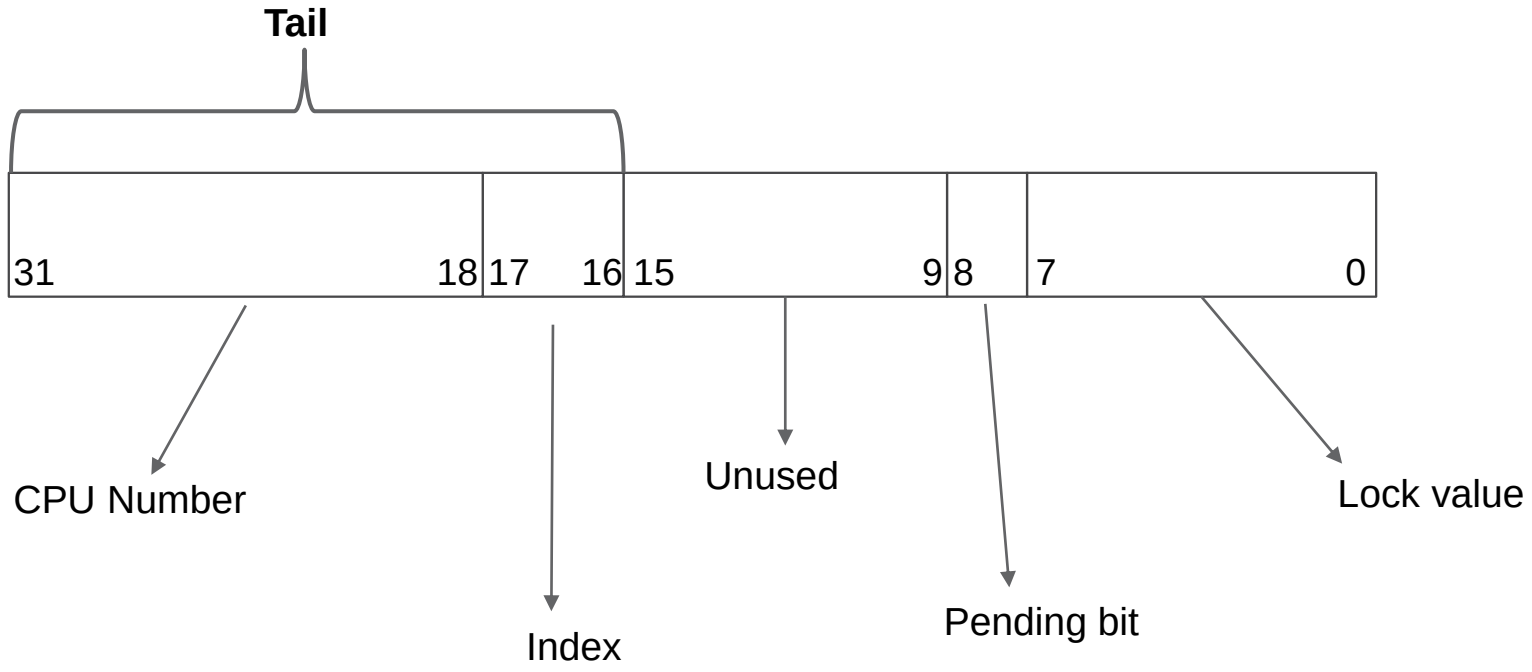
Lock Value: Indicates the lock is held by someone or not. This is a single byte, but only Bit 0 is set while locking.

Queued Spin Locks



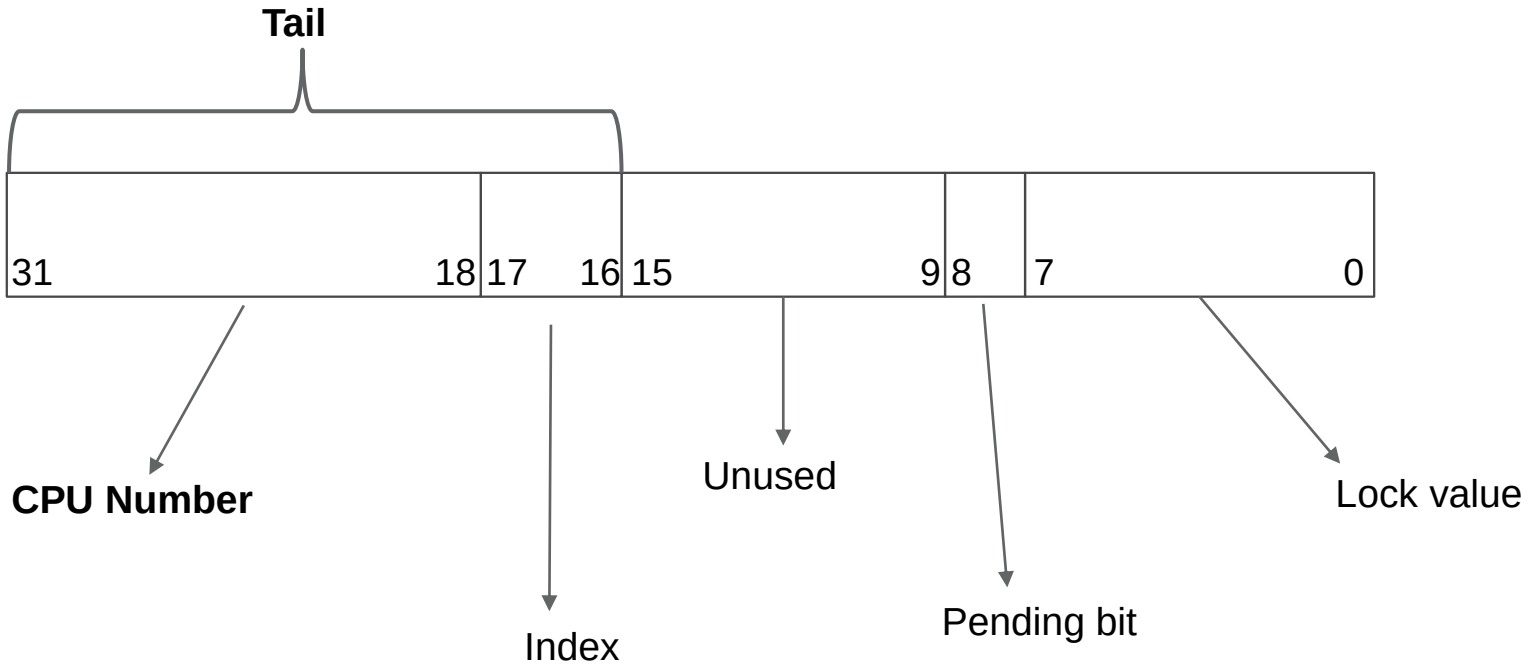
Pending bit: Used to indicate that there is one contender. If the lock is held and only one other waiter exists, only “Lock Value” and “Pending bit” suffices

Queued Spin Locks



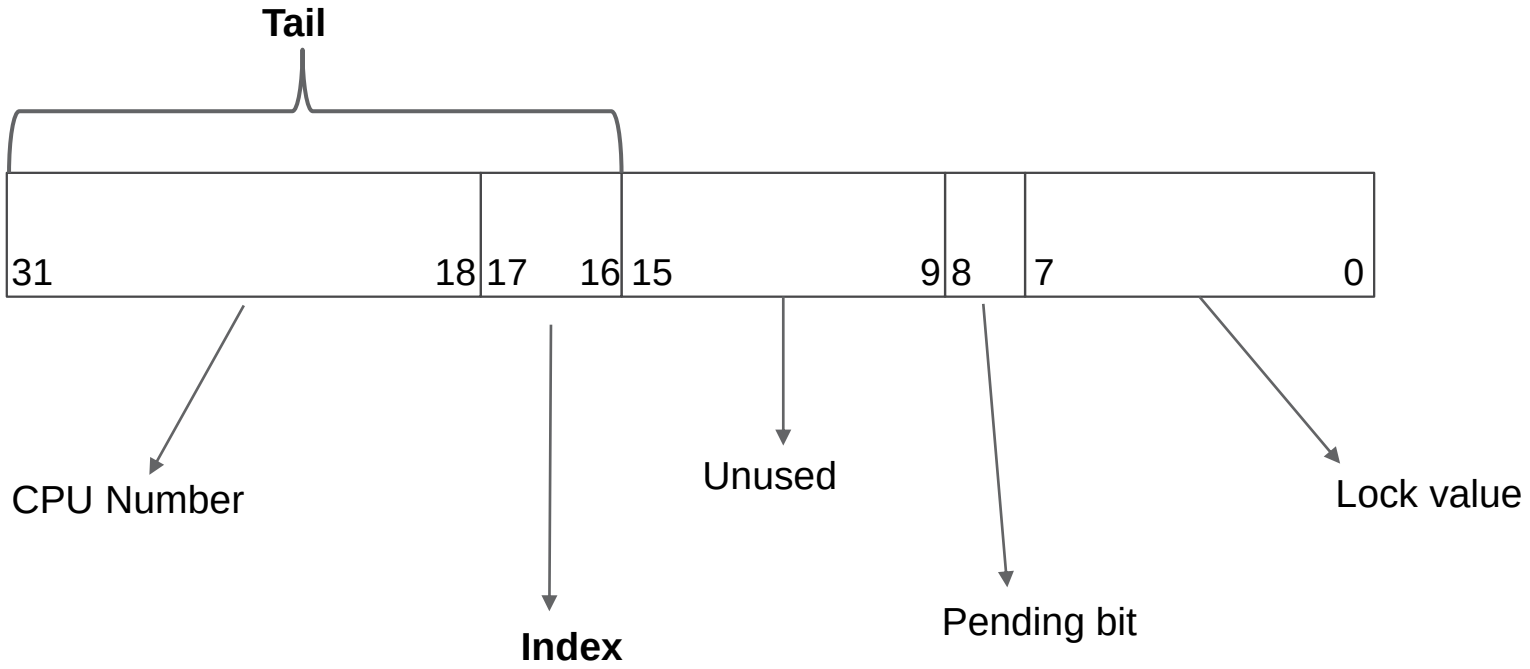
Tail: The combination of CPU Number and Index is used to indicate the tail of the queue of waiters of this lock. The tail bits are set when there are more than 1 waiters.

Queued Spin Locks



CPU Number: Indicates the CPU number of the tail waiter.

Queued Spin Locks

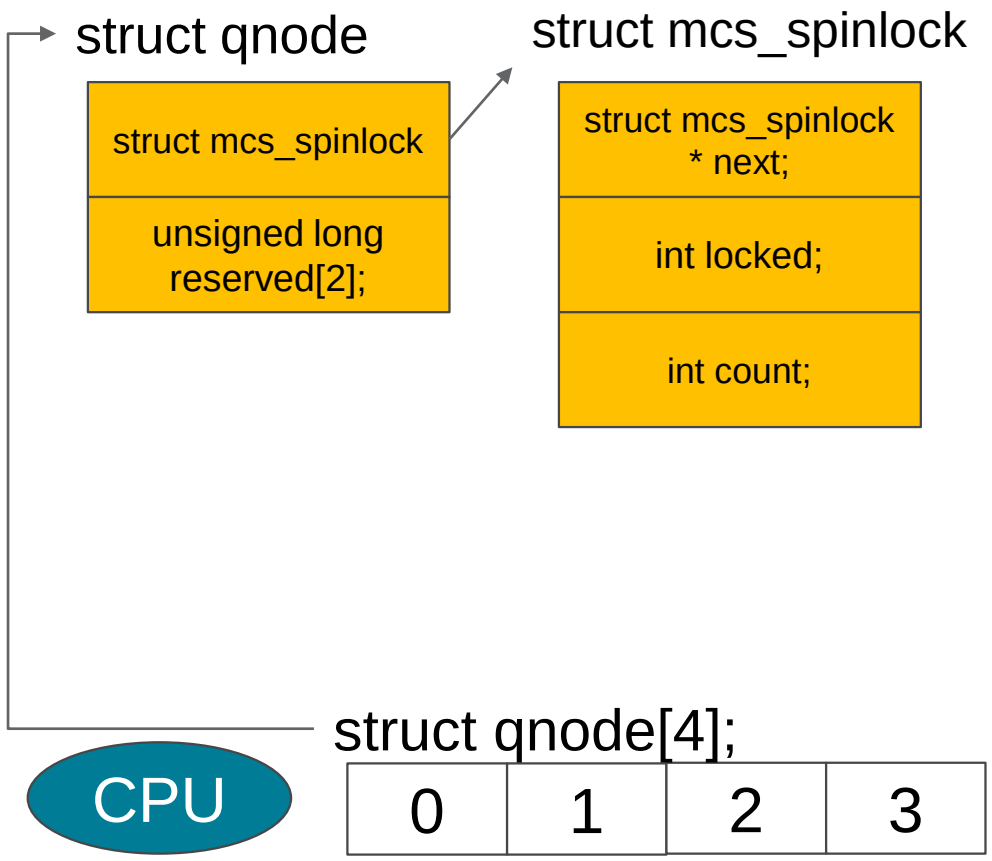


Index: Indicates the “context” of the tail waiter. Note that each CPU can contend for *some* spin-lock in nested contexts of depth at most 4: Task, softirq, hardirq, NMI.

Hence 2 bits (Bits 16:17) are good enough to encode the index

NOTE: The spin-locks attempted to be taken in each of these contexts are different locks else there will be deadlocks.

Queued Spin Locks



Every CPU has a per-cpu variable, which is an array of 4 qnodes.

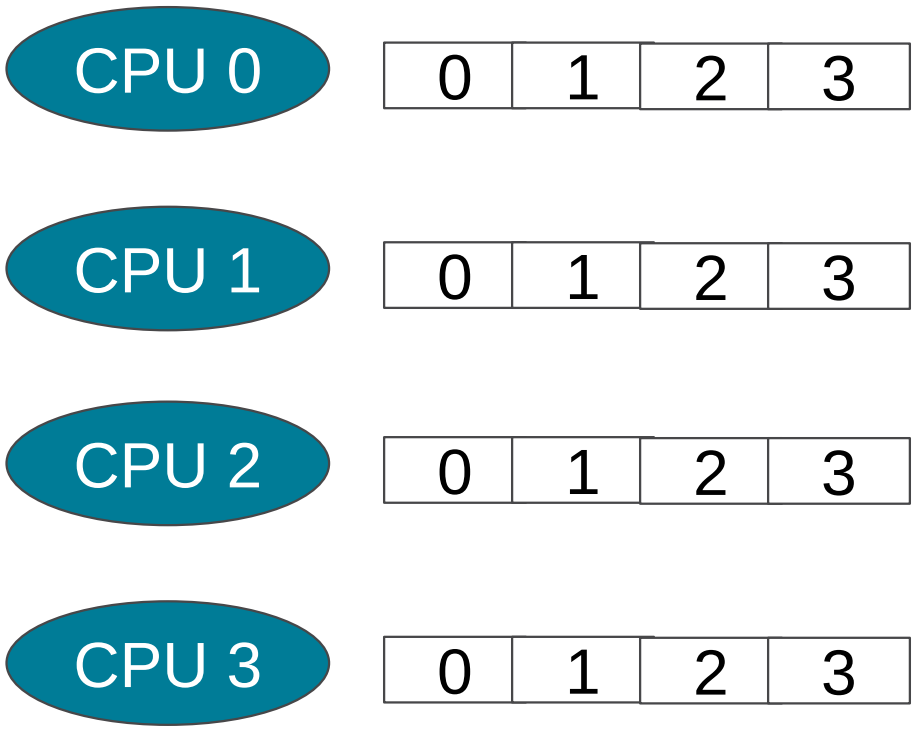
Each qnode contains a mcs_spinlock object and a couple of reserved 8 bytes which are used in paravirt-spinlock case (not covered today)

Each mcs_spinlock object contains a pointer to next element in the queue, a locked 4-byte element and a count 4-byte element.

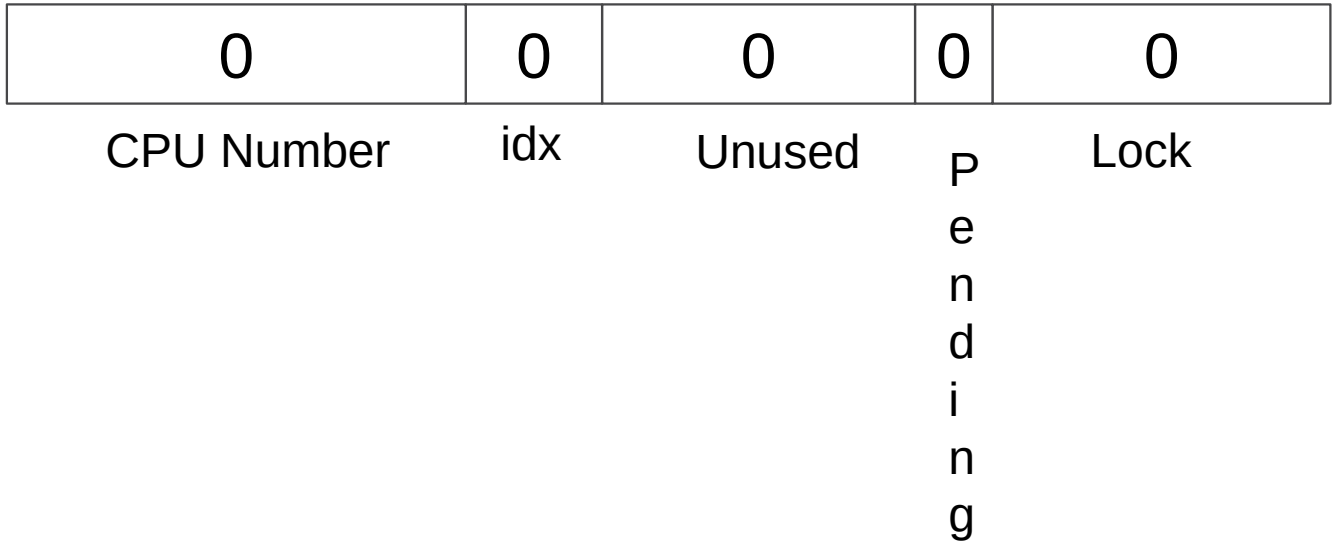
locked: Each waiter which is not in the head of the queue spins on the locked variable of its own mcs_spinlock.

count: Indicates the nesting depth. Only updated for qnode[0].mcs_spinlock.count

Queued Spin Locks

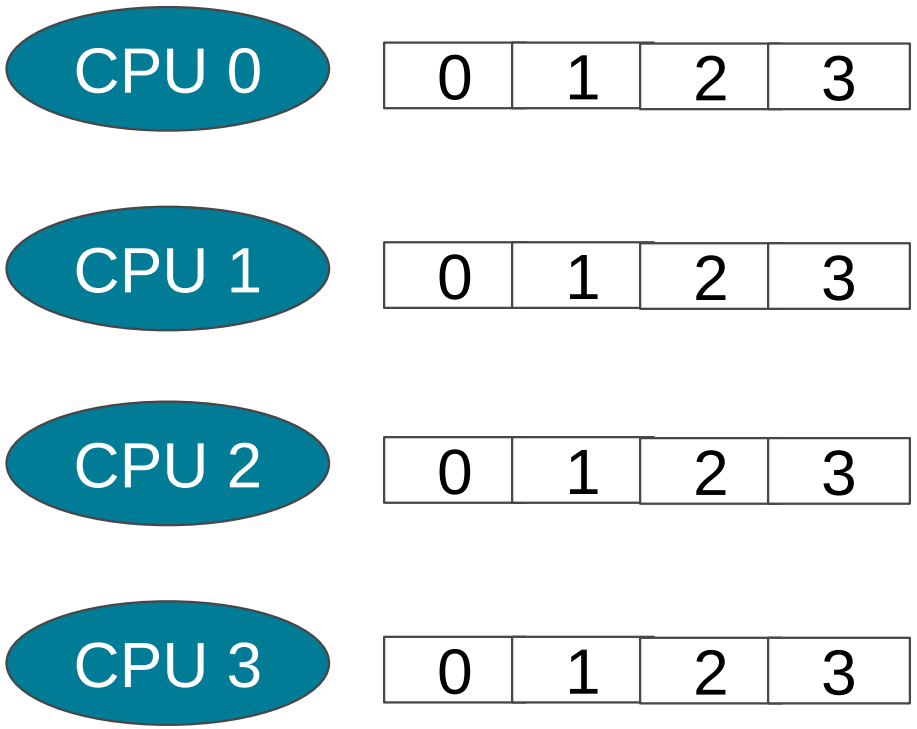


Initially



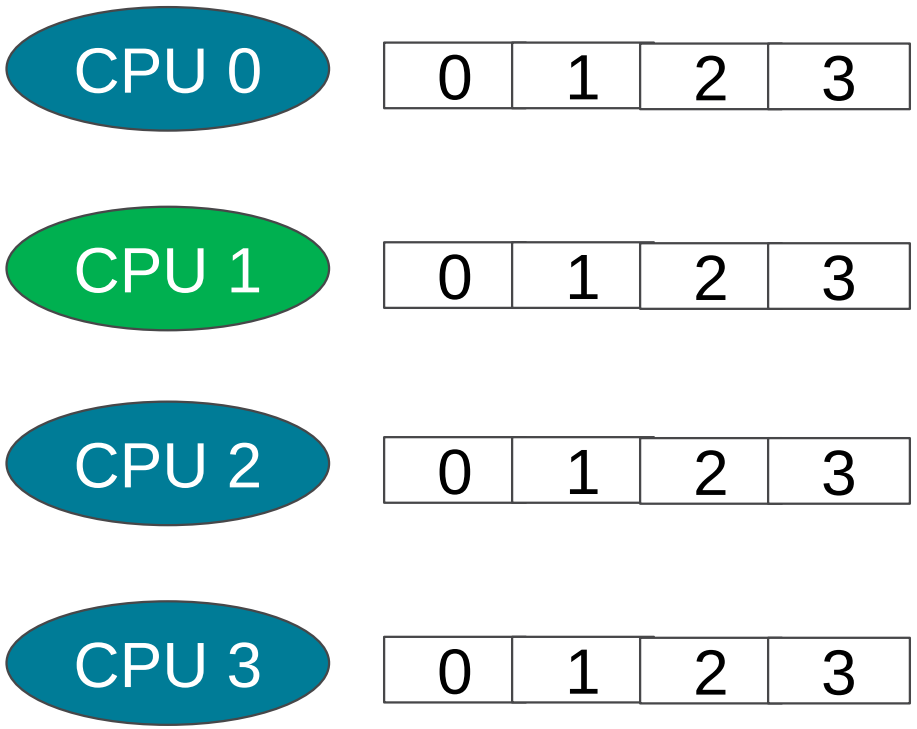
Only one contender

Queued Spin Locks

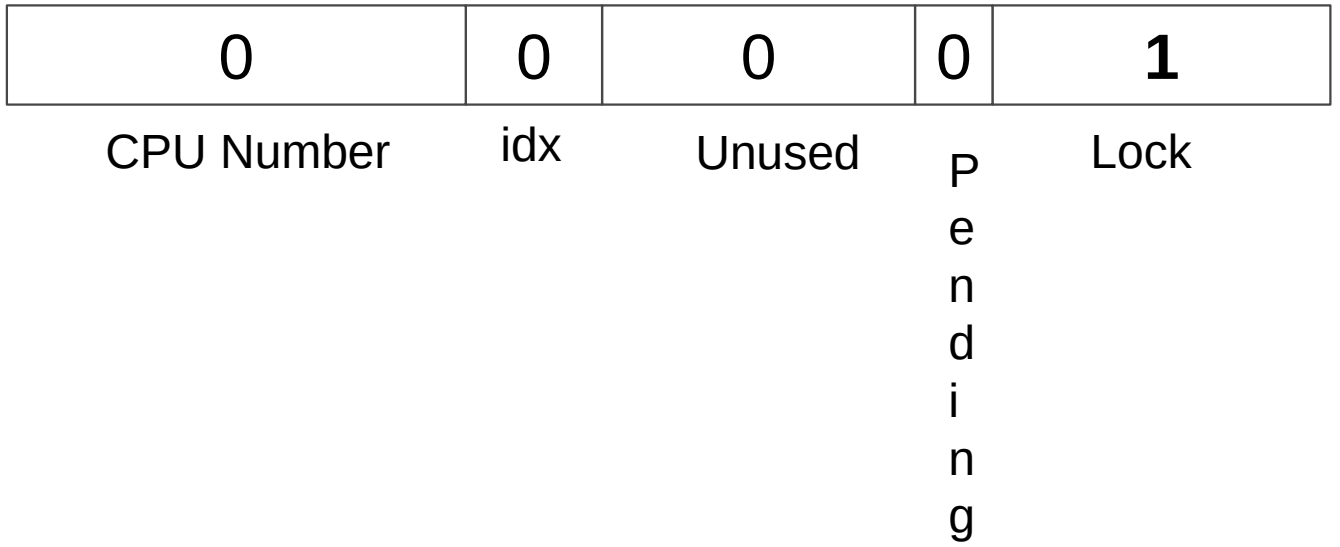


CPUs 1 tries to get the lock by compare exchanging 0 with Q_LOCKED (value = 1) for the qspinlock 32-bit variable

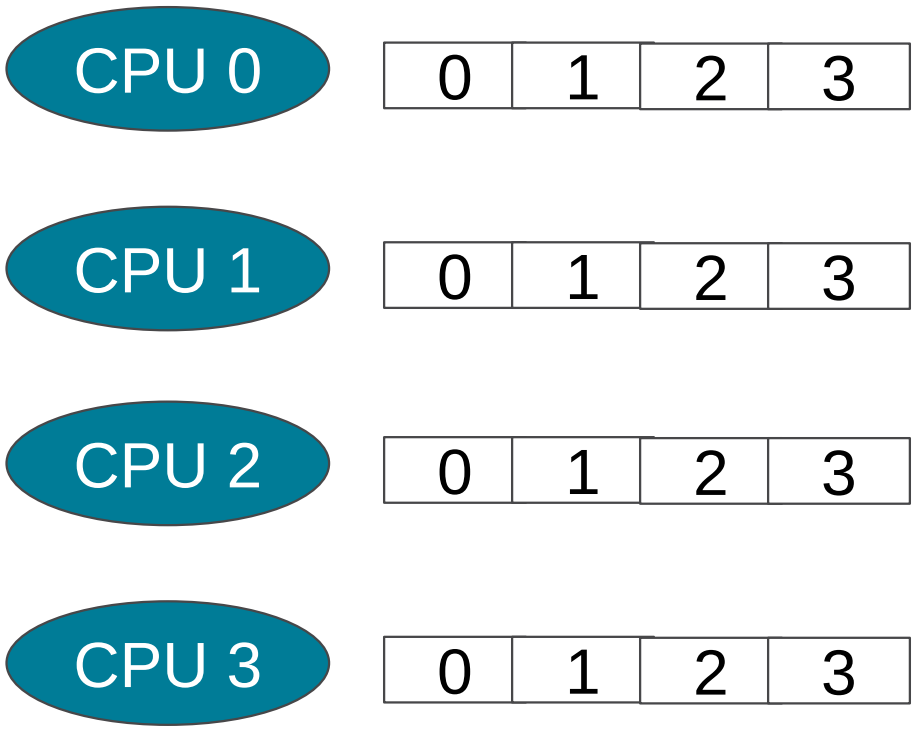
Queued Spin Locks



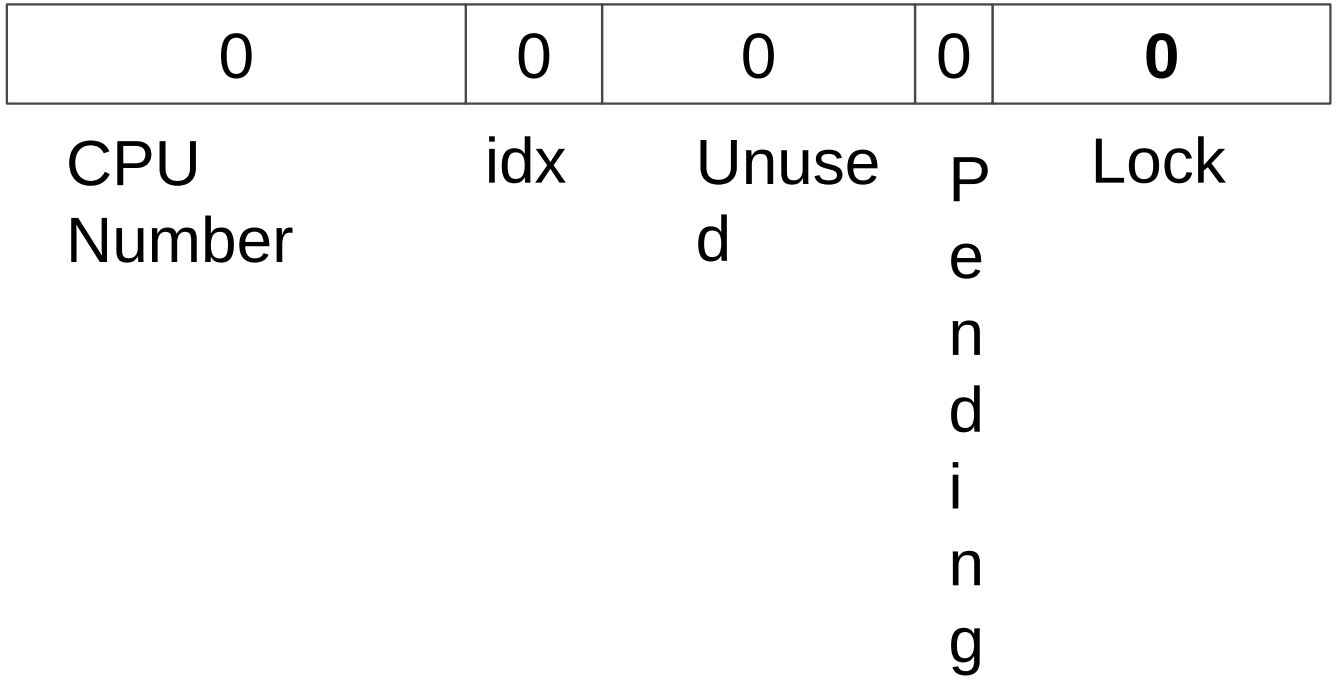
CPUs 1 succeeds and has the lock



Queued Spin Locks

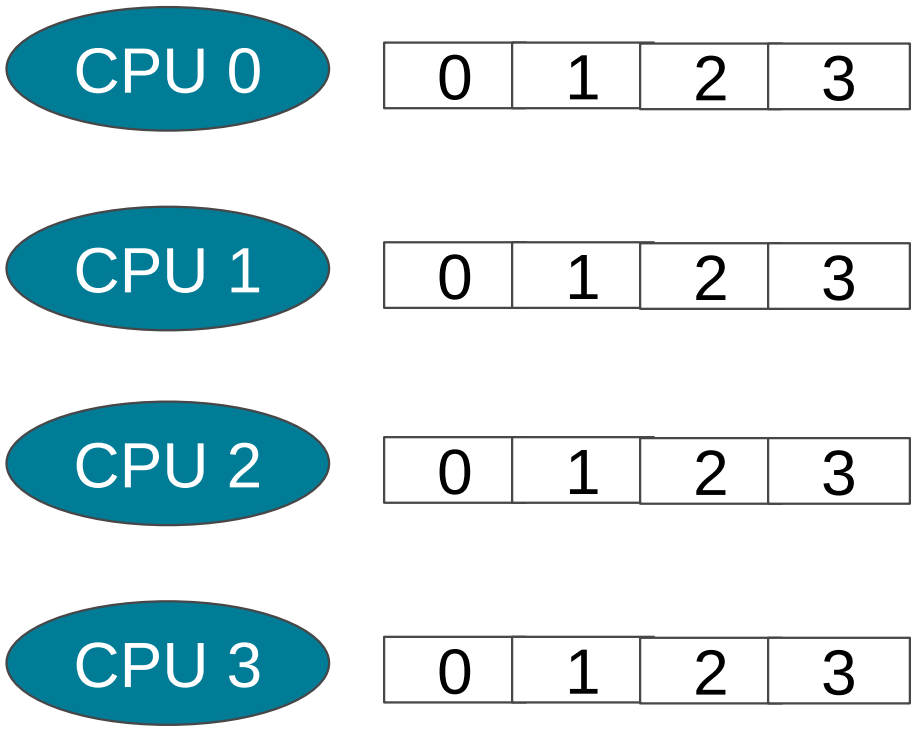


Once CPUs 1 is done, it sets the Lock byte to 0, thus releasing the lock

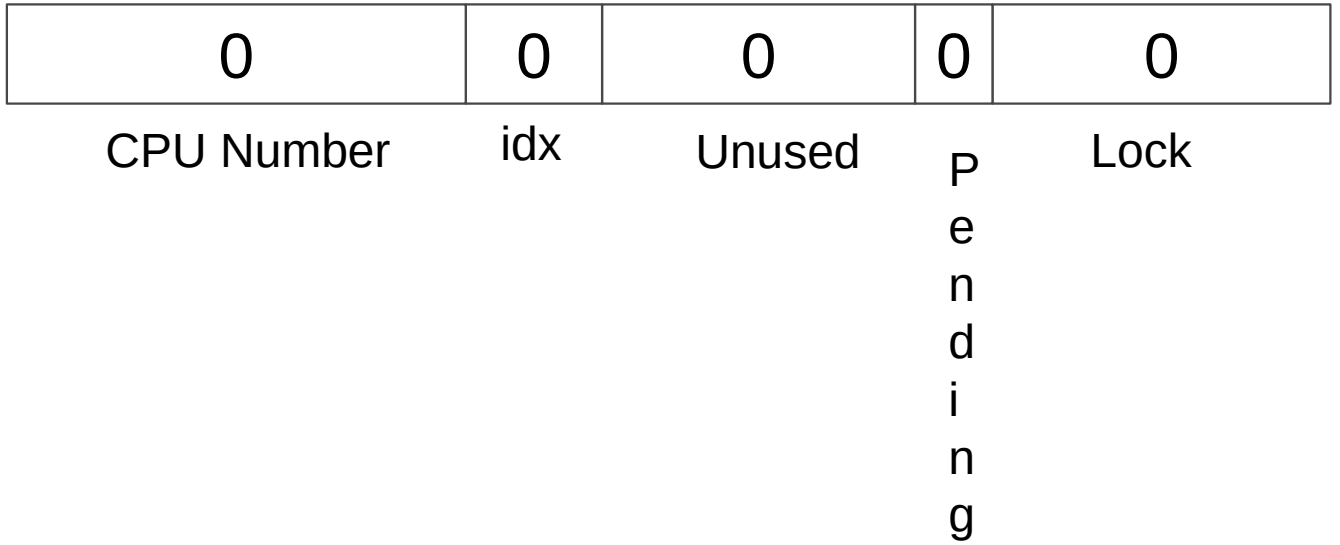


Two contenders

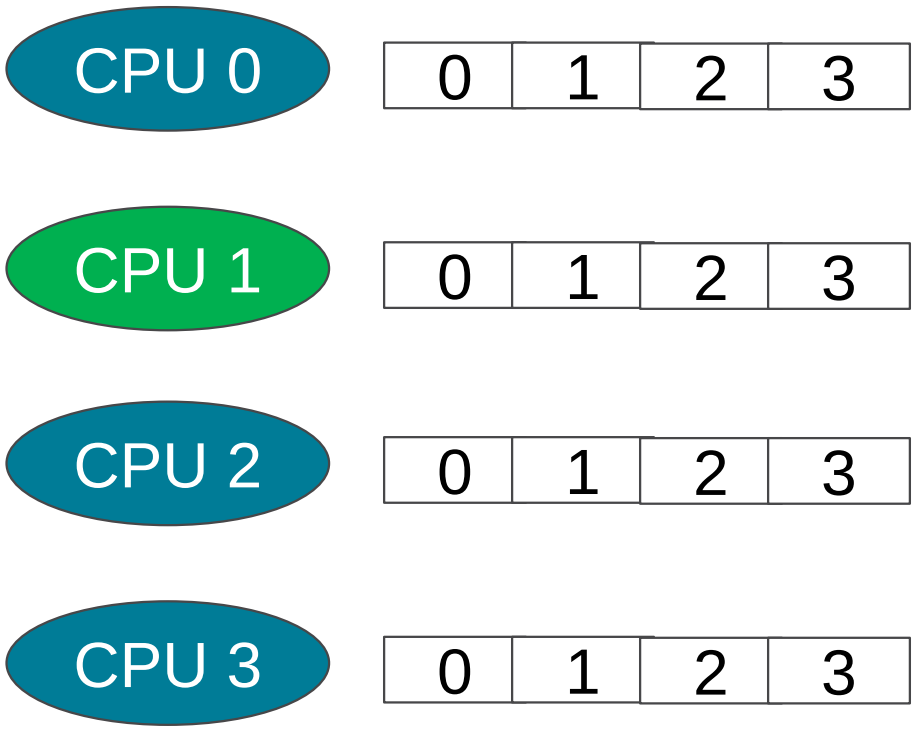
Queued Spin Locks



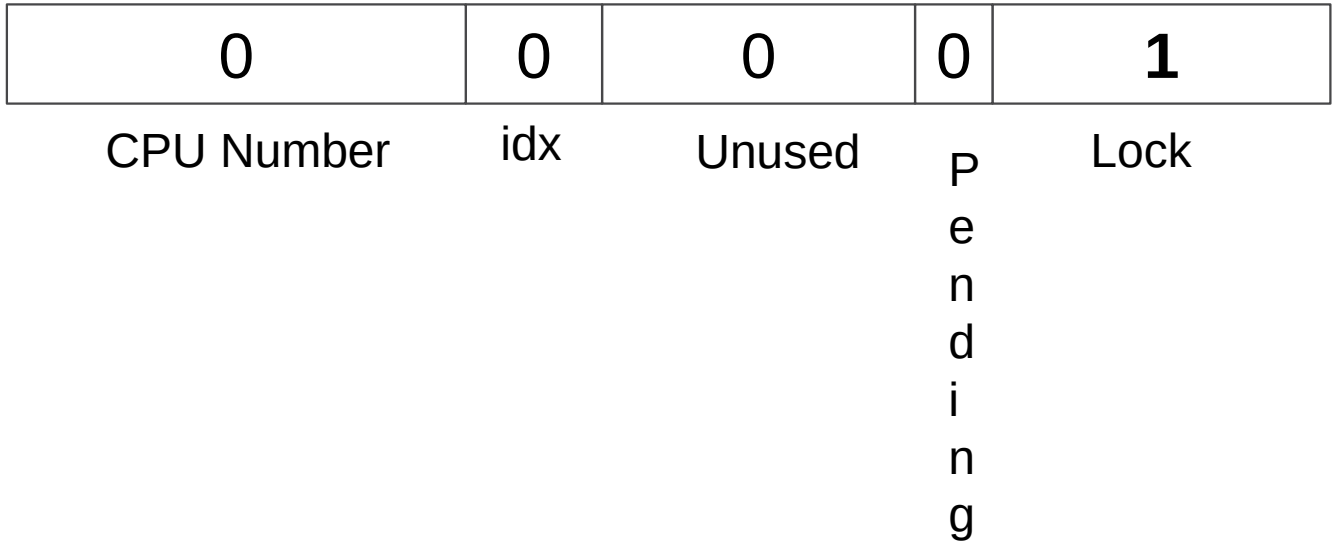
CPUs 1, 2 : Concurrently try to get the lock by compare exchanging 0 with Q_LOCKED on the 32-bit qspinlock variable



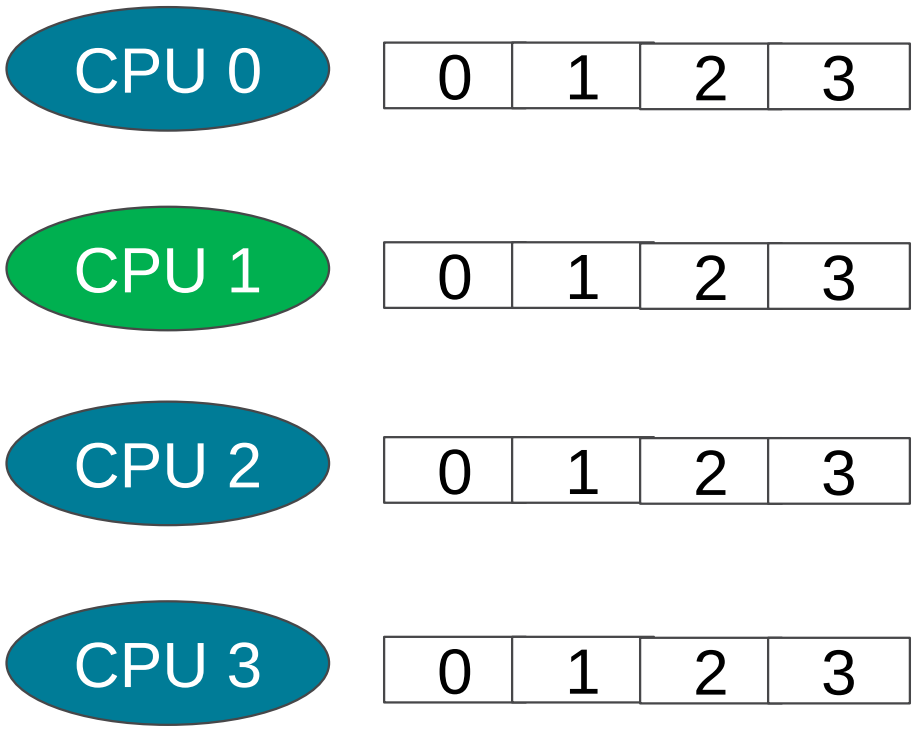
Queued Spin Locks



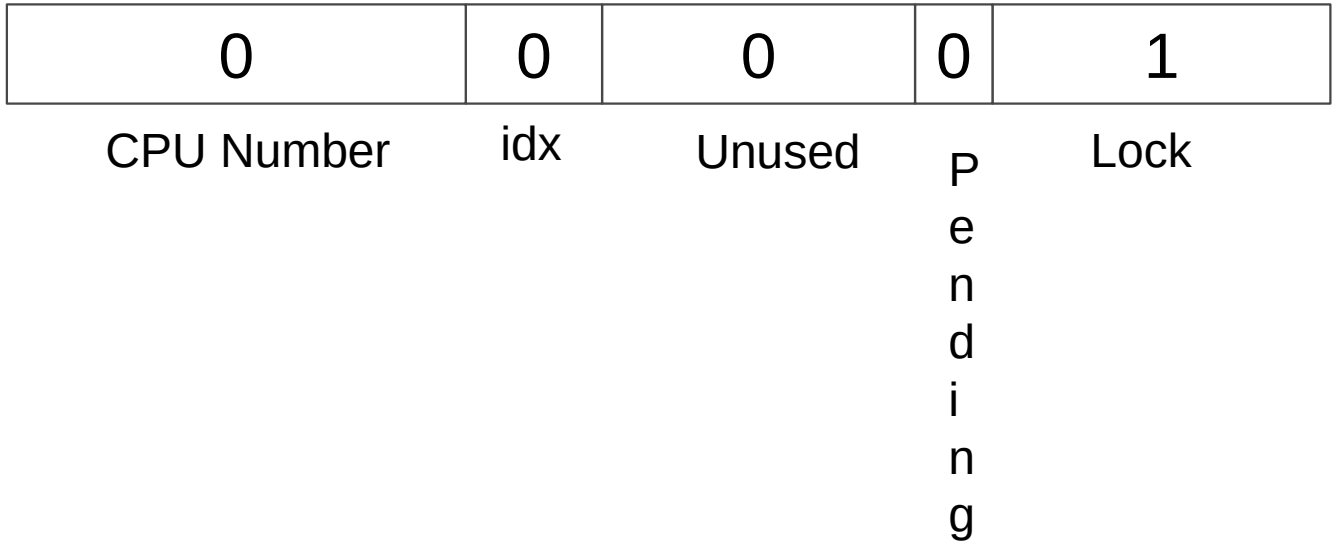
WLOG, let us assume that CPU 1 wins the race and is able to set the qspinlock value to 1. CPU 1 now has the lock.



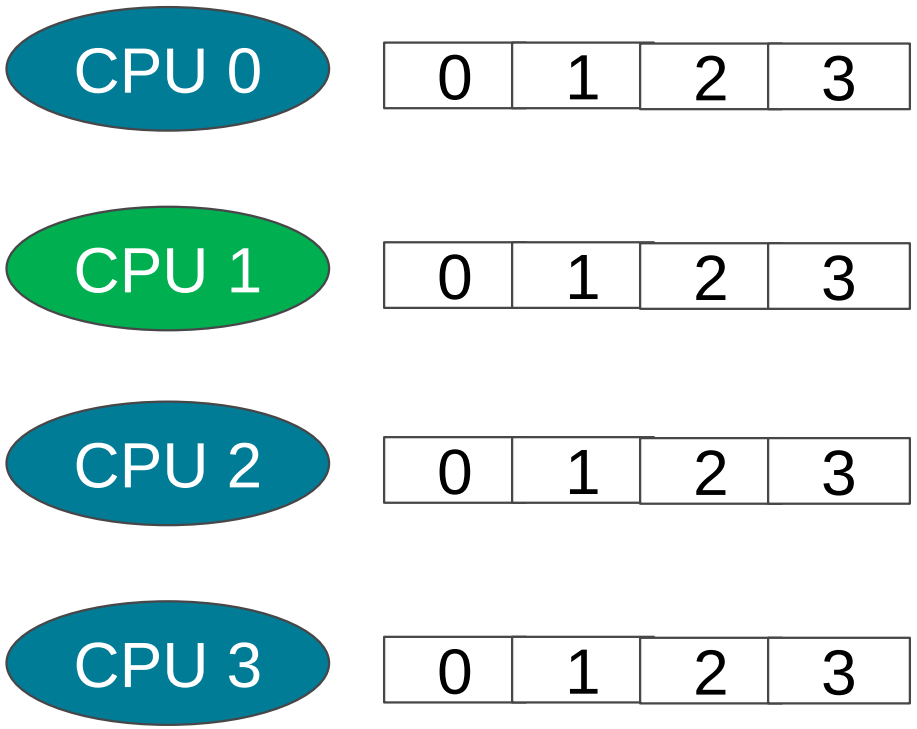
Queued Spin Locks



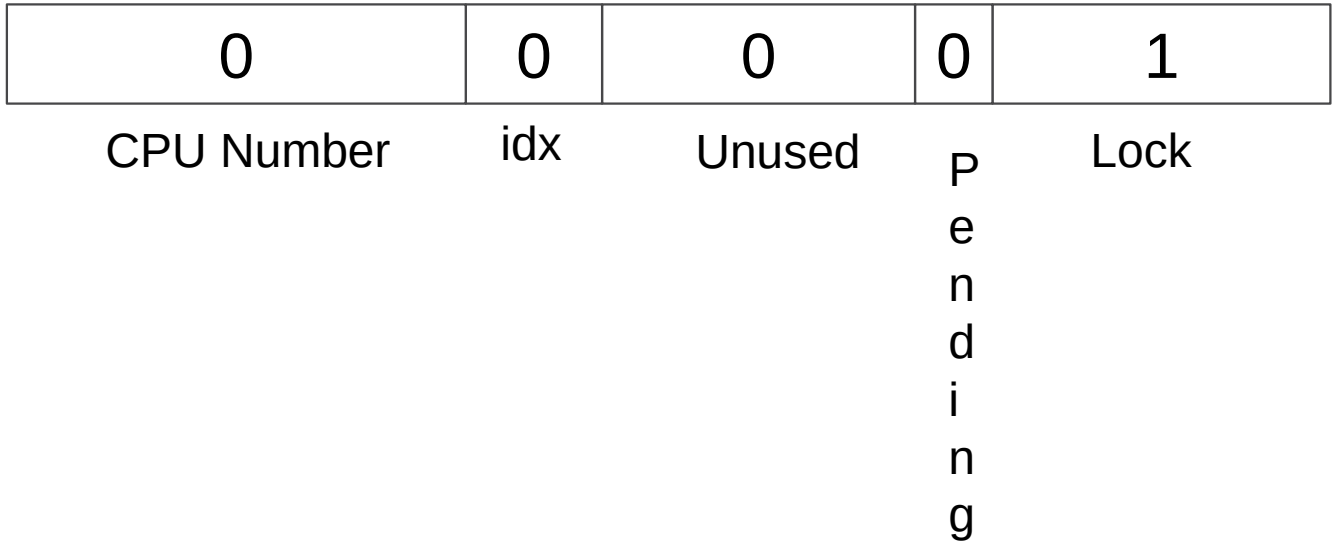
CPU 2 loses the race as it sees the old value to be non-zero. It sees if apart from the lock byte any other bytes are set. That would indicate other waiters. In this case there are none.



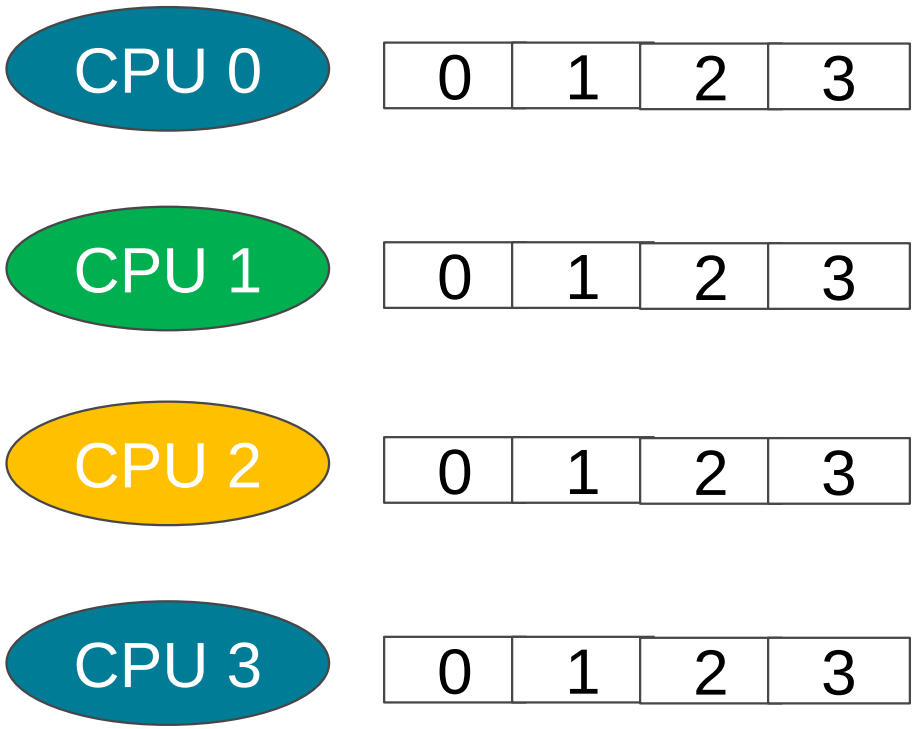
Queued Spin Locks



Since there are no waiters yet, it tries to atomically set the pending bit and get the old value (using atomic_fetch_or).



Queued Spin Locks

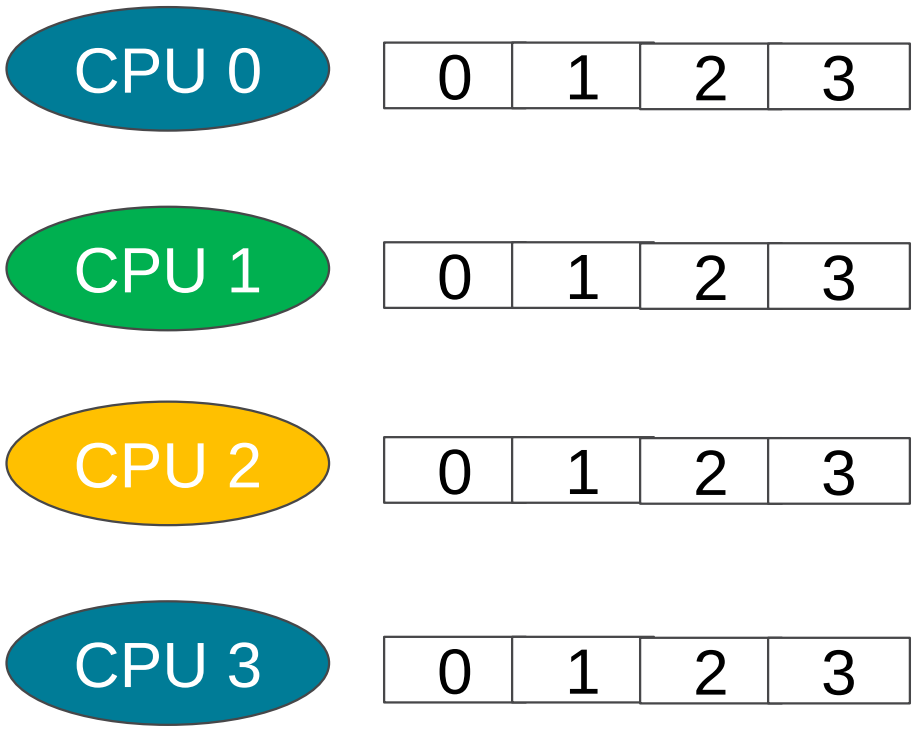


Since there are no waiters yet, it tries to atomically set the pending bit and get the old value (using atomic_fetch_or).

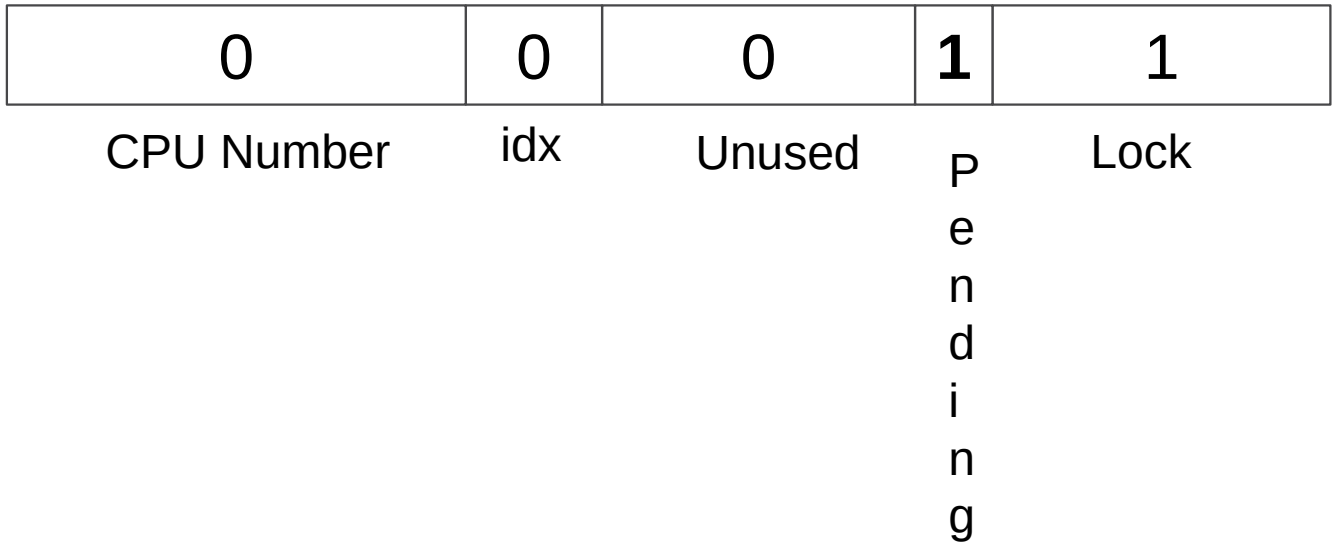
0	0	0	**1**	1
CPU Number	idx	Unused	Pending	Lock

CPU 2 will read the old value (0, 0, 0, 0, 1) having updated the pending bit.

Queued Spin Locks

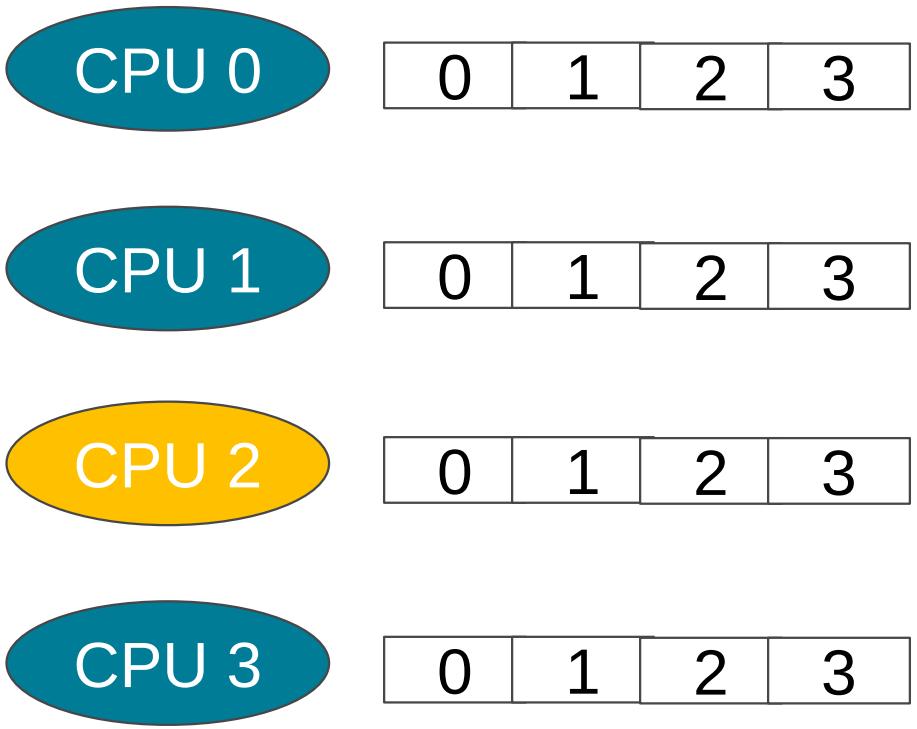


From the old value, CPU 2 knows that it was the first to update the pending bit. So, it just spins on the lock byte to become 0



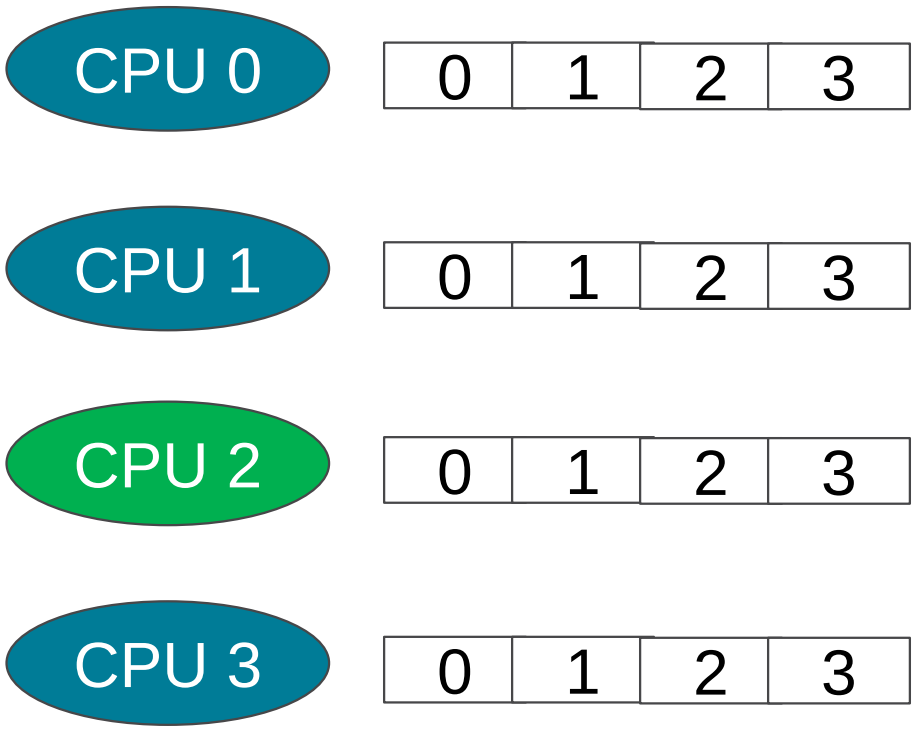
CPU 2 will read the old value (0, 0, 0, 0, 1) having updated the pending bit.

Queued Spin Locks



CPU 1, once it is done will unlock by setting the lock byte to 0.

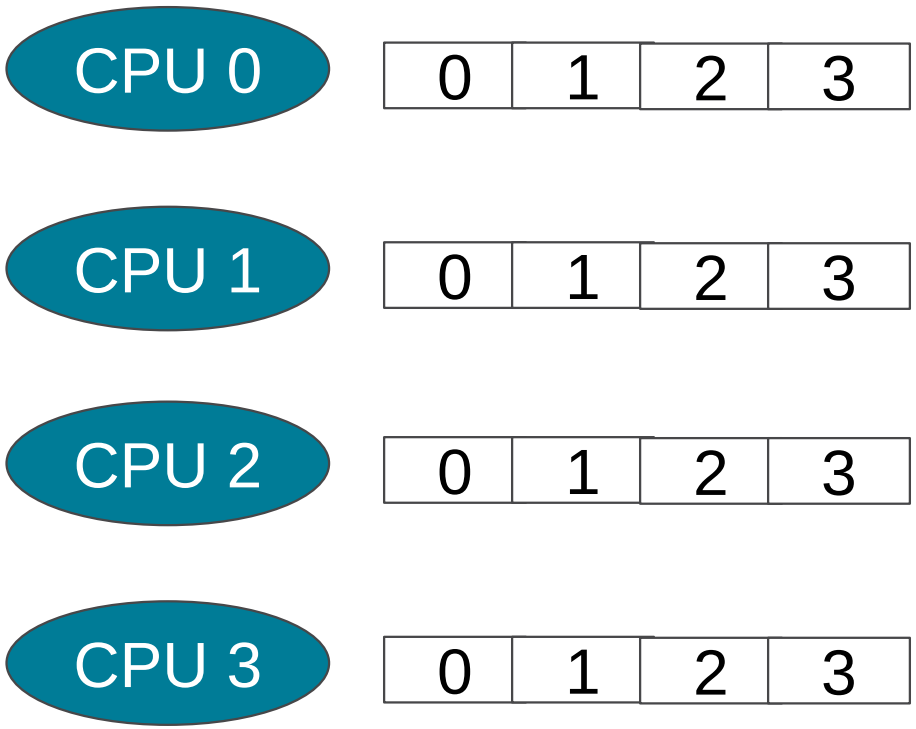
Queued Spin Locks



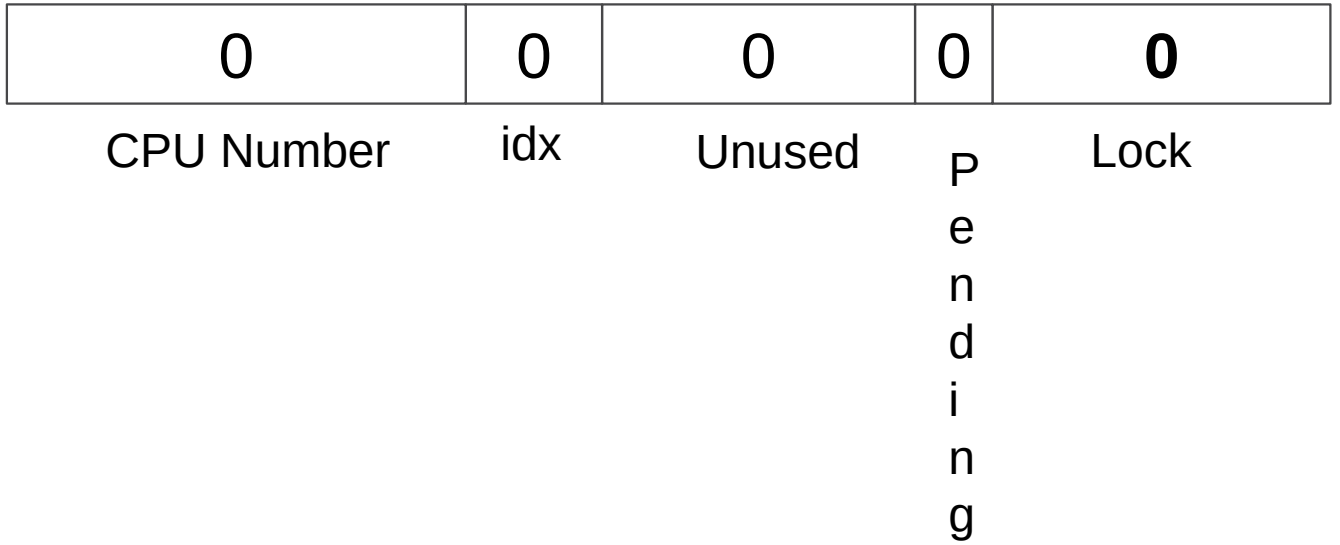
CPU 2, notice that the “Lock” byte is 0. It will atomically clear the pending bit and set the lock byte and acquire the lock.

0	0	0	**0**	**1**
CPU Number	idx	Unused	Pending	Lock

Queued Spin Locks

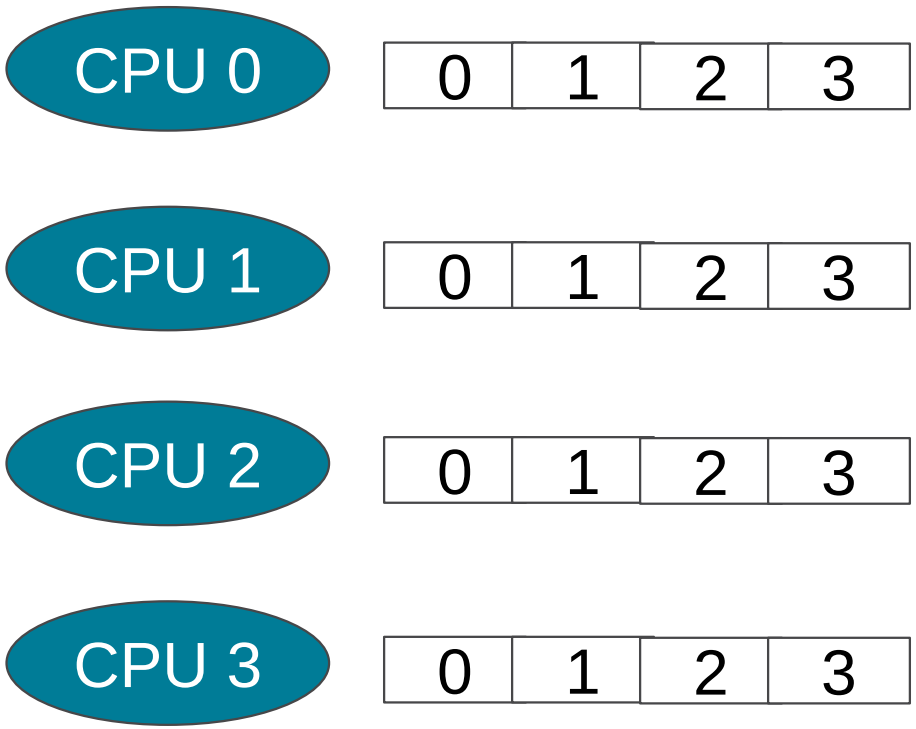


Once it is done, CPU 2 releases the lock by clearing the Lock byte.

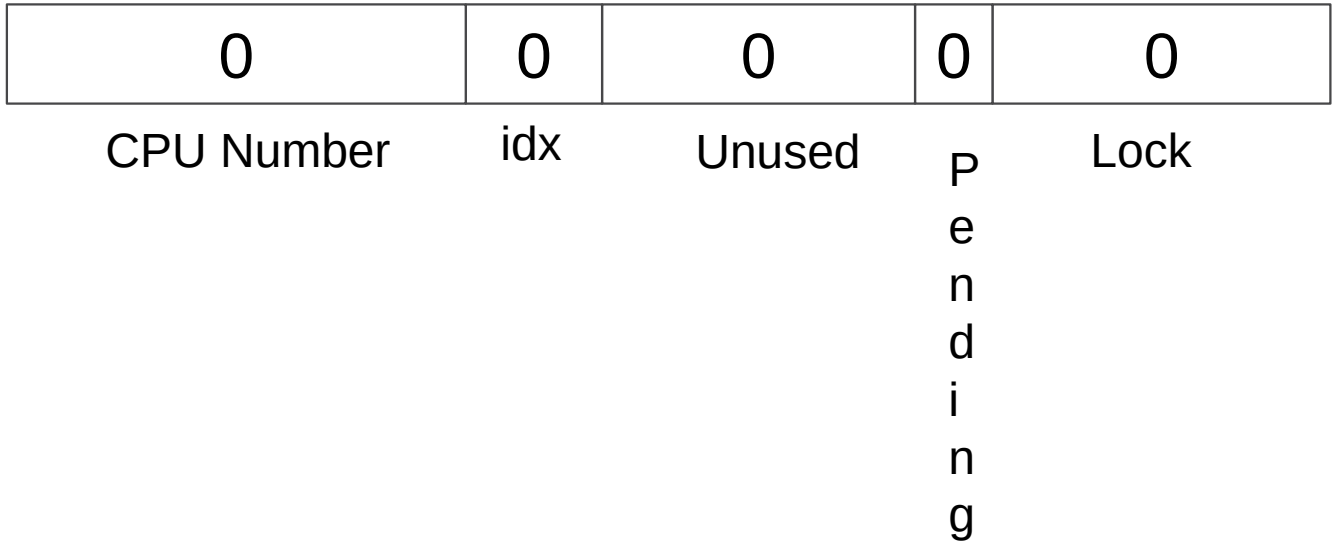


Three contenders

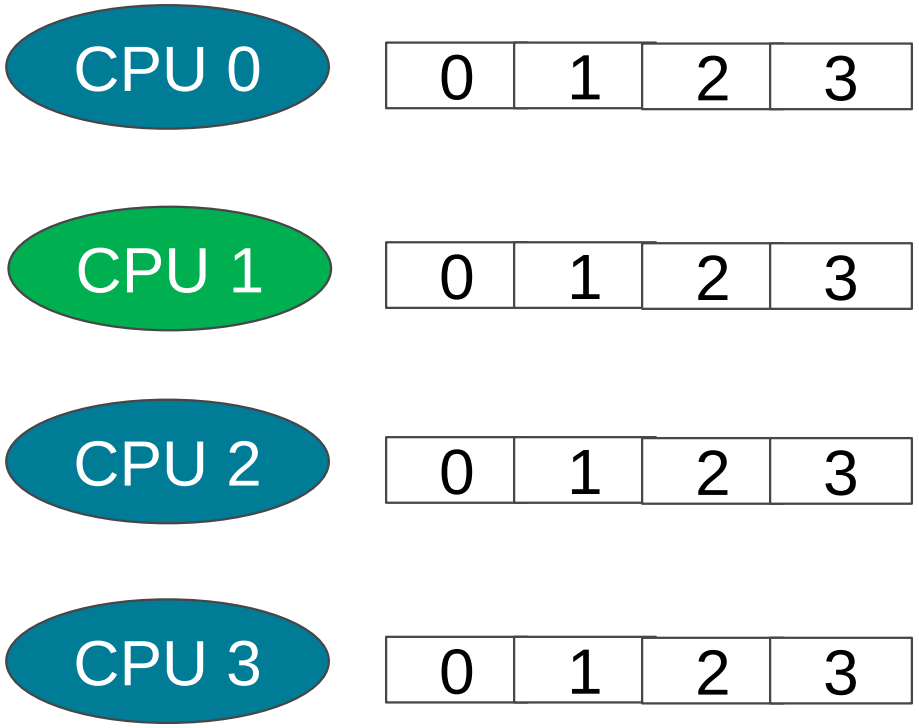
Queued Spin Locks



CPUs 0, 1, 2: All concurrently try to get the lock by compare exchanging 0 with Q_LOCKED (value = 1).



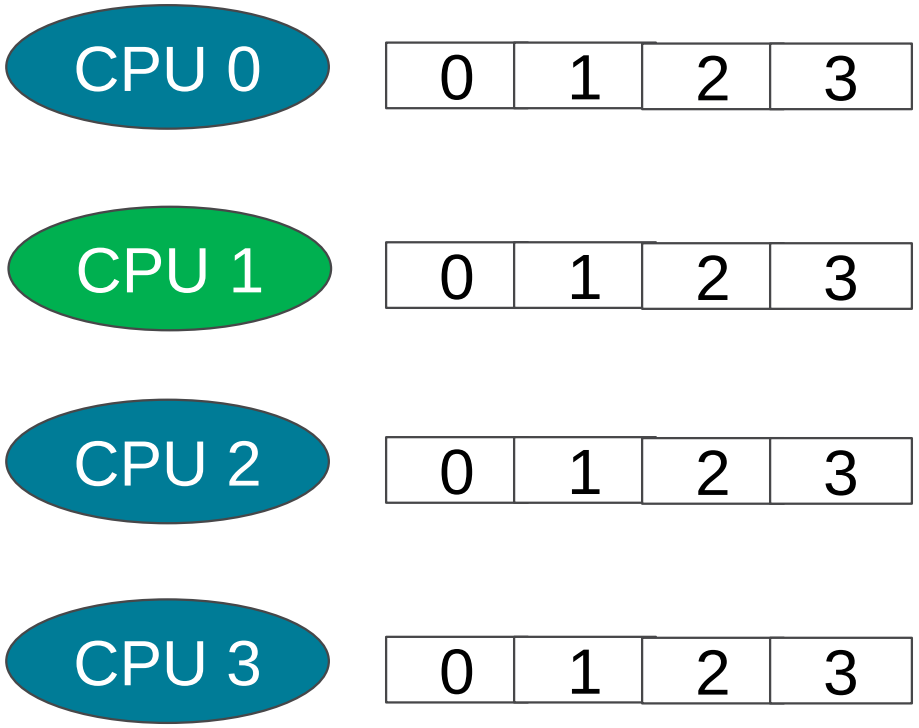
Queued Spin Locks



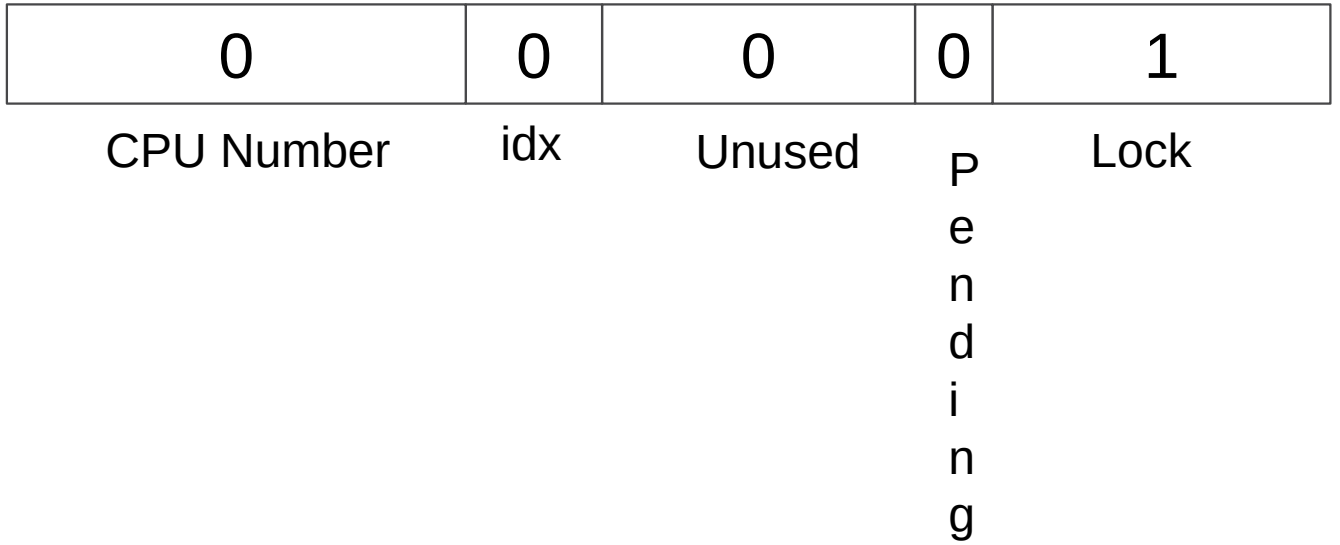
WLOG, let us assume that CPU 1 wins the race and is able to set the Lock bit to 1.

0	0	0	0	1
CPU Number	idx	Unused	Pending	Lock

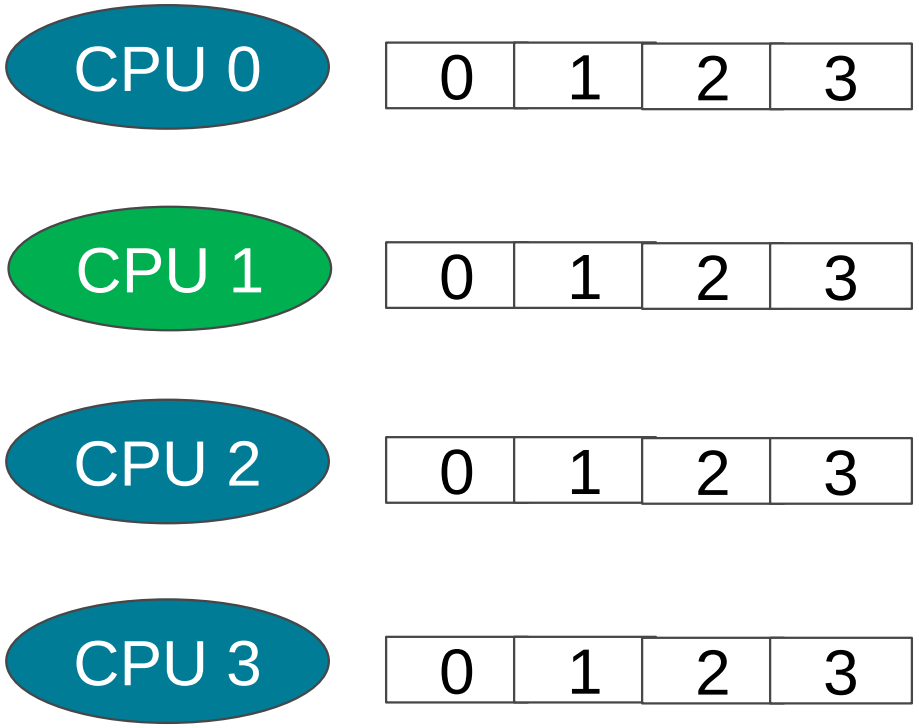
Queued Spin Locks



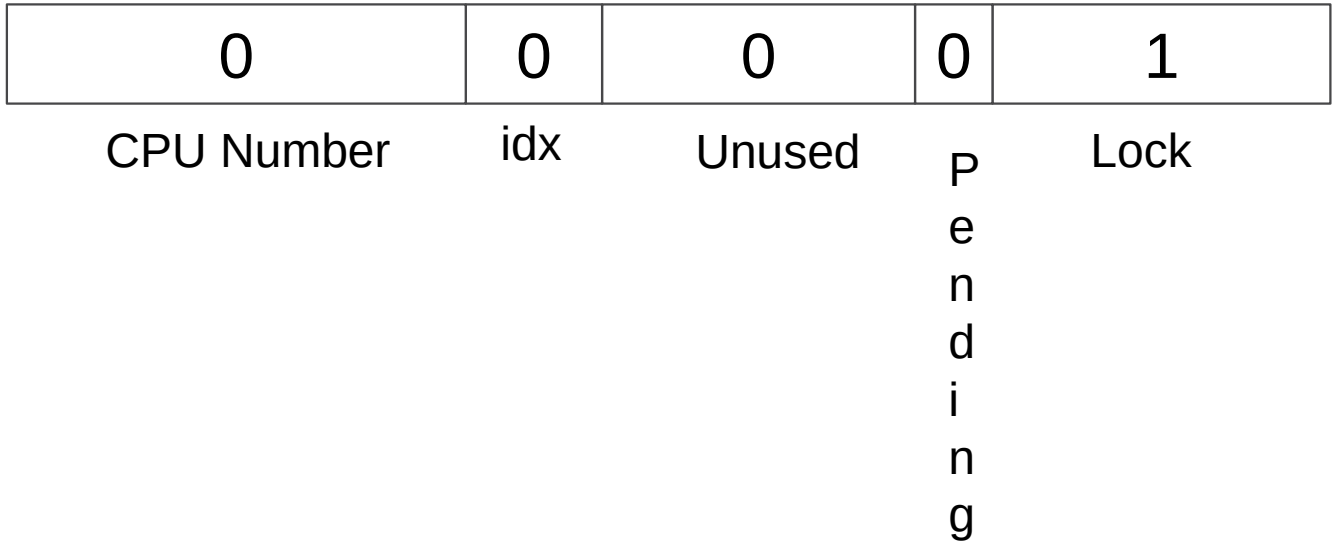
CPUs 0 and 2 check any of the bits other than Lock byte is set. If they find it so, they go to the queuing phase.



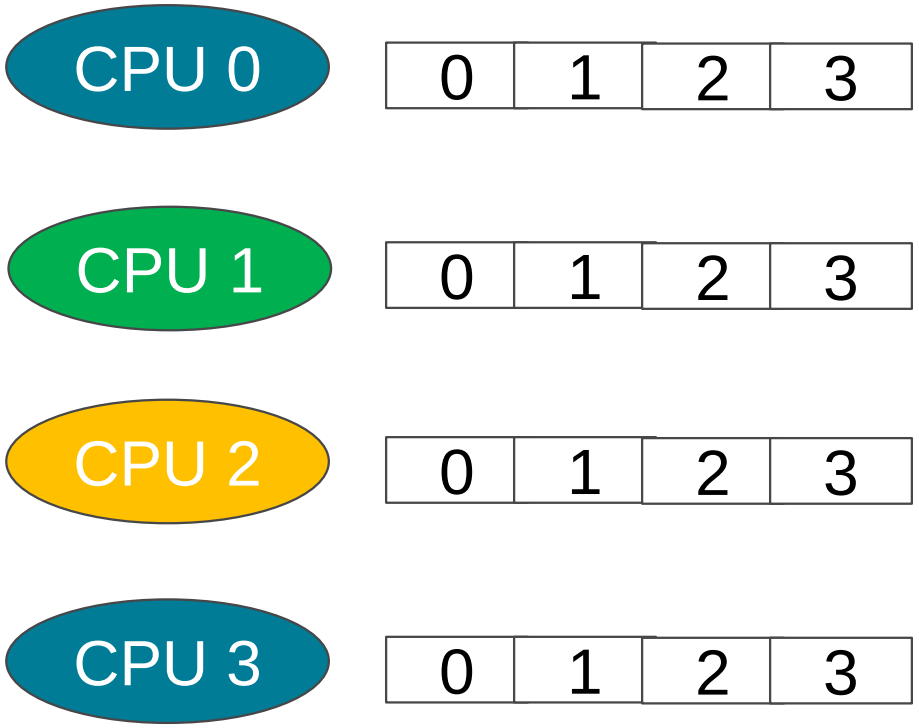
Queued Spin Locks



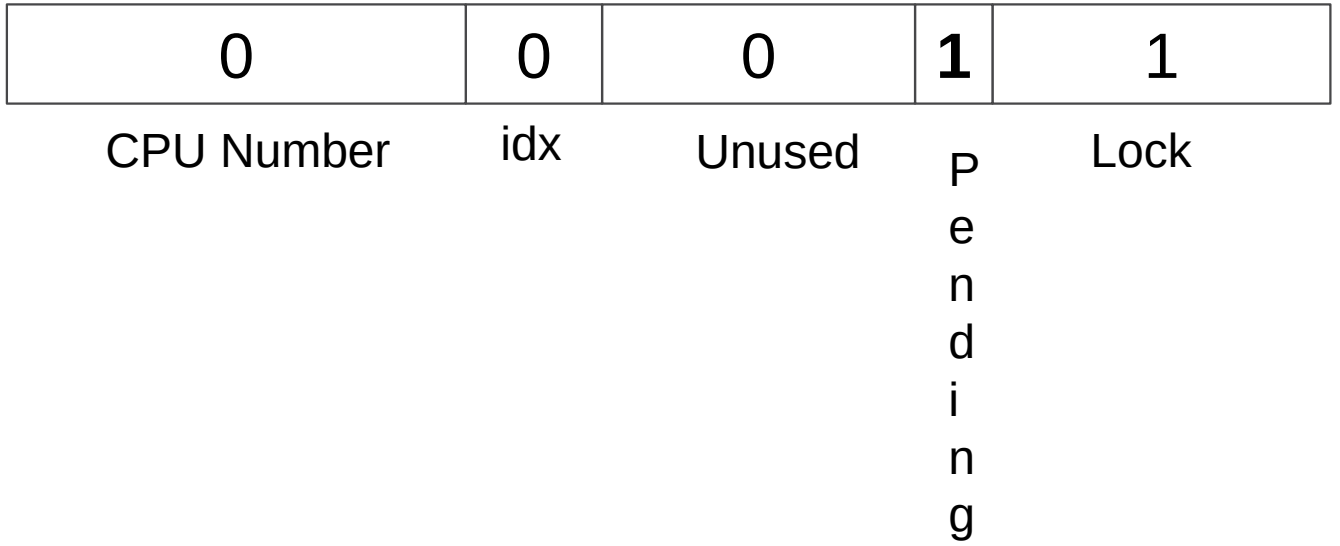
In this case, since CPUs 0 and 2 don't see any bits set, they try to atomically set the pending bit and get the old value (using atomic_fetch_or).



Queued Spin Locks

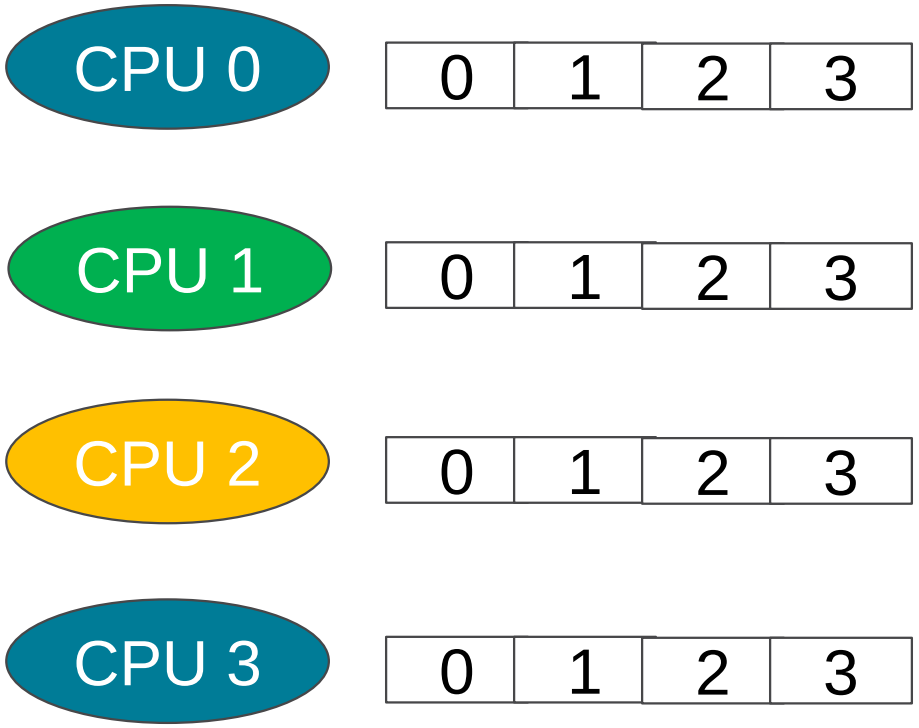


In this case, since CPUs 0 and 2 don't see any bits set, they try to atomically set the pending bit and get the old value (using atomic_fetch_or).

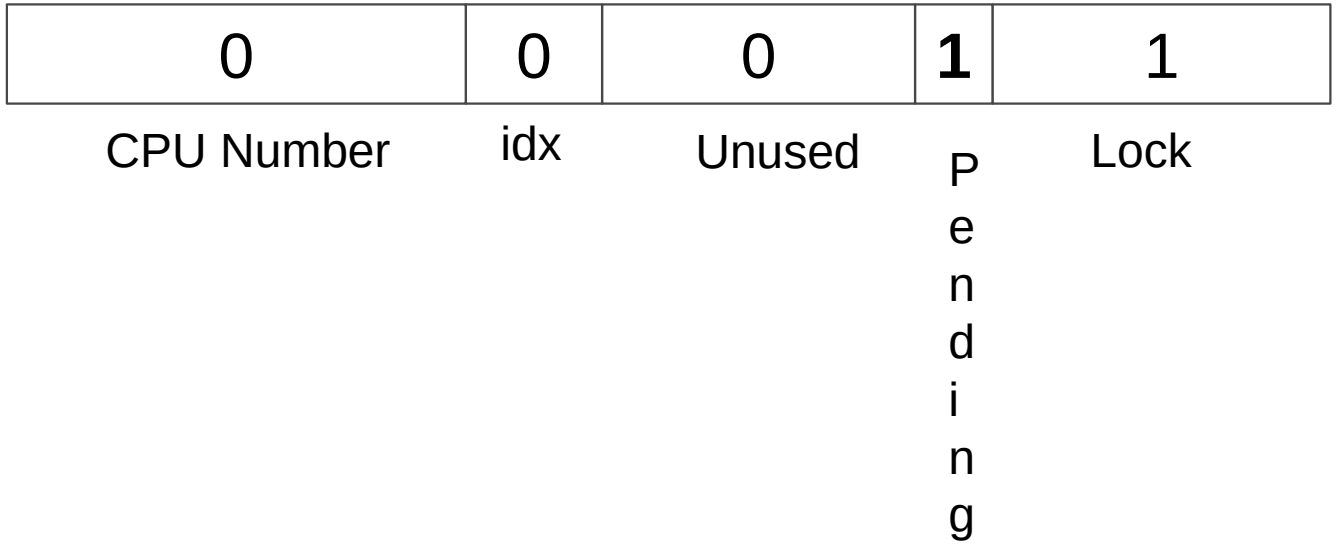


Assume that CPU 2 wins the race. It will read the old value (0, 0, 0, 0, 1) having updated the pending bit.
CPU 0 will read the old value (0, 0, 0, 1, 1) having updated the pending bit.

Queued Spin Locks

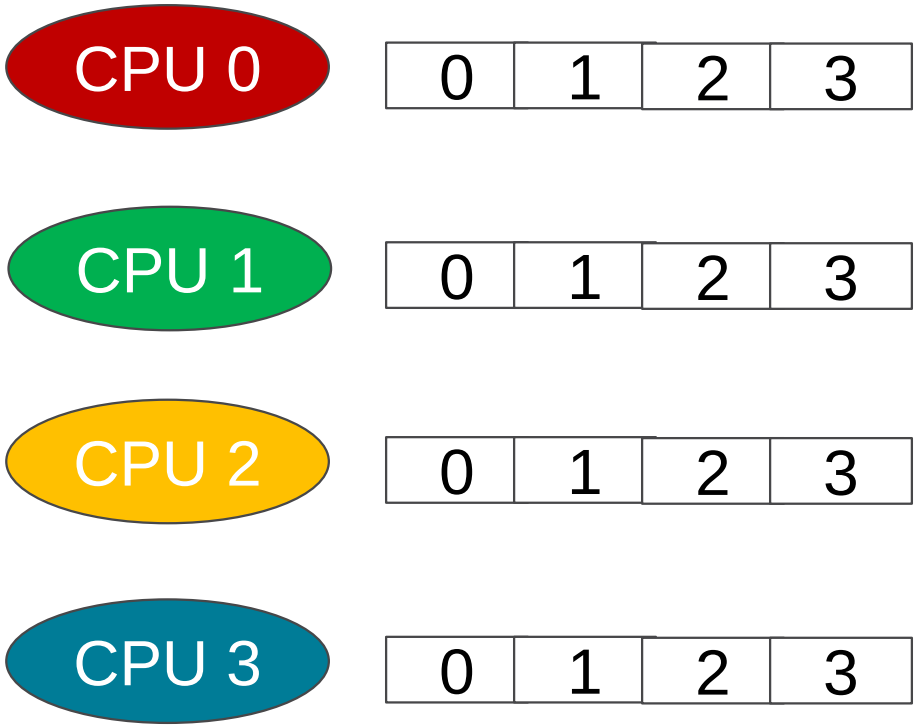


CPU 2 knows that it has successfully set the pending bit and no other bits are set. Hence it is next in line. It just spins until the Lock byte becomes 0.

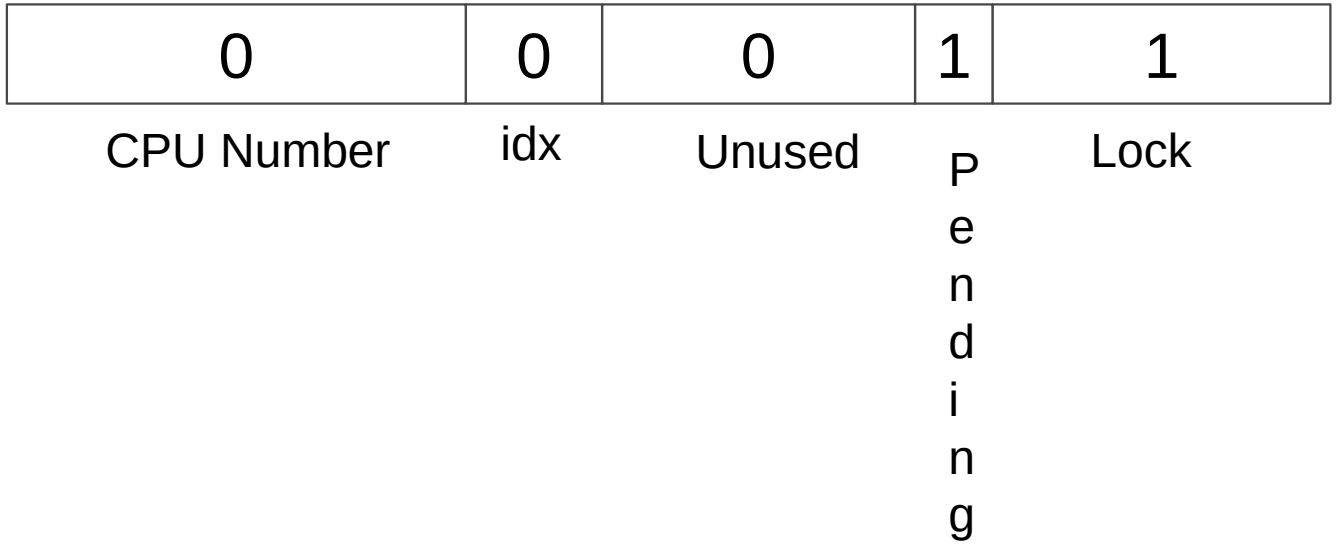


Assume that CPU 2 wins the race. It will read the old value (0, 0, 0, 0, 1) having updated the pending bit.
CPU 0 will read the old value (0, 0, 0, 1, 1) having updated the pending bit.

Queued Spin Locks

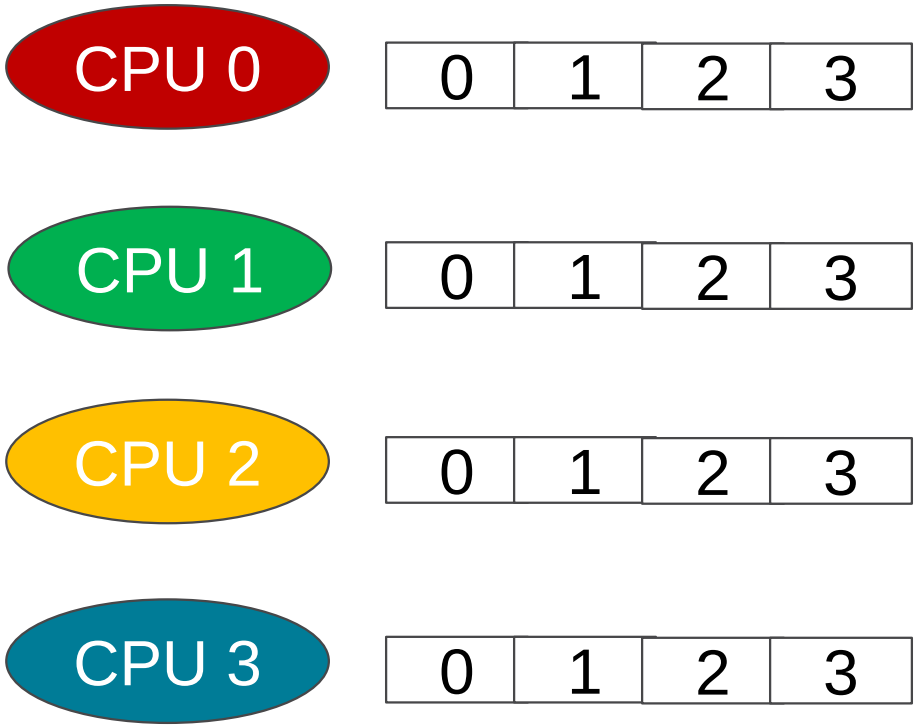


CPU 0 knows that it wasn't the first to set the pending bit. So, it has to queue.



Assume that CPU 2 wins the race. It will read the old value (0, 0, 0, 0, 1) having updated the pending bit.
CPU 0 will read the old value (0, 0, 0, 1, 1) having updated the pending bit.

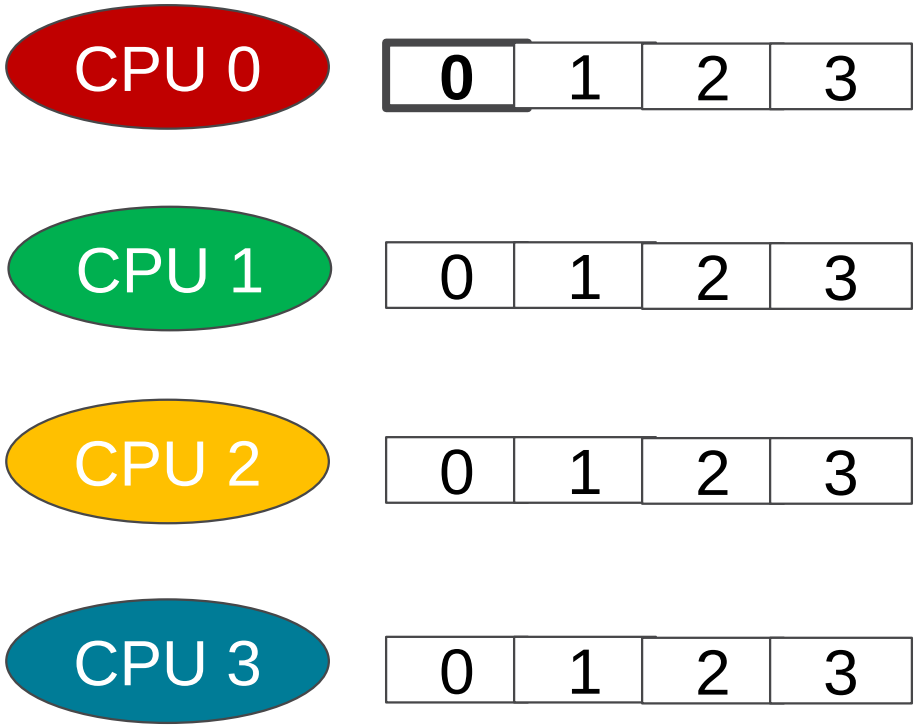
Queued Spin Locks



CPU 0 grabs the first available qnode. It does so by checking and incrementing qnode[0].mcs_spinlock.count.

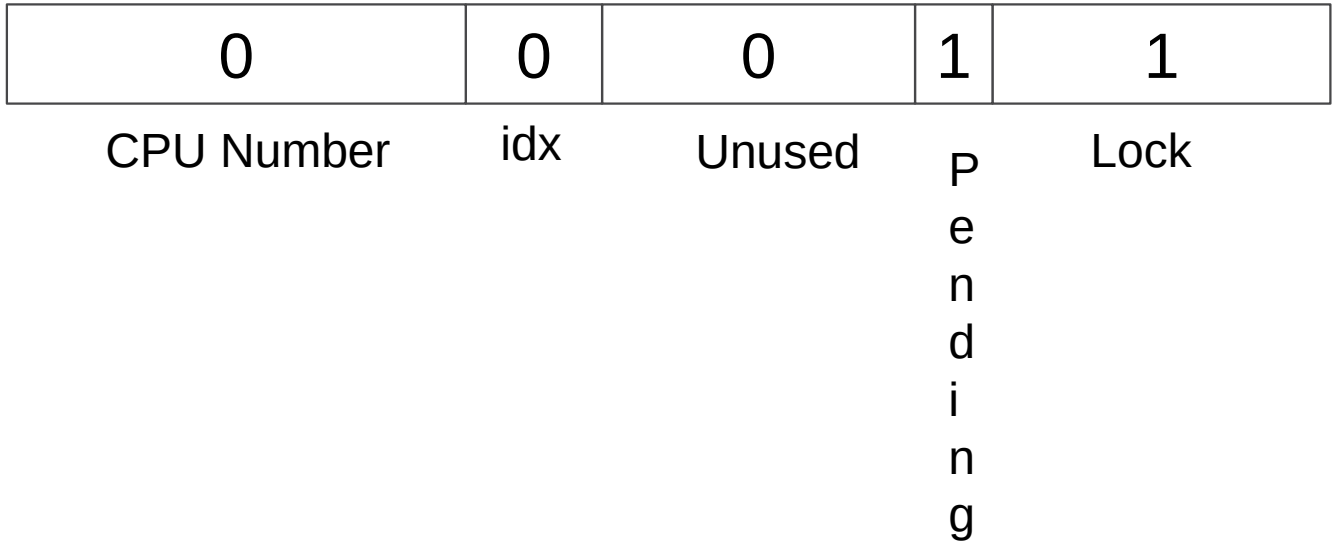
0	0	0	1	1
CPU Number	idx	Unused	Pending	Lock

Queued Spin Locks

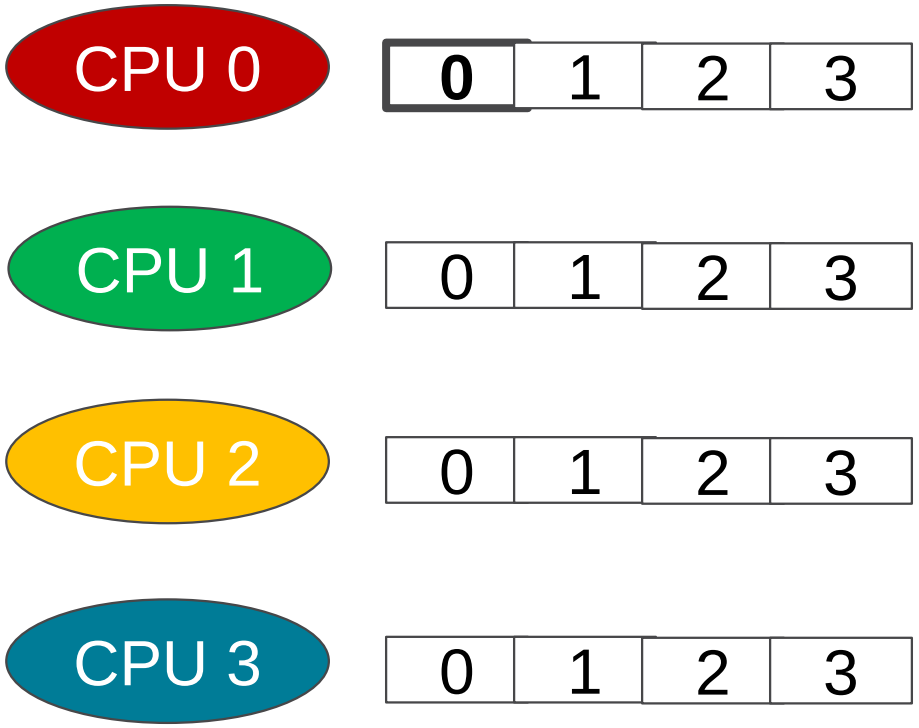


WLOG assume CPU 0 has the idx 0 available.

It then generates the tail-encoding as (CPU-Number +1 , idx) = (1, 0).



Queued Spin Locks

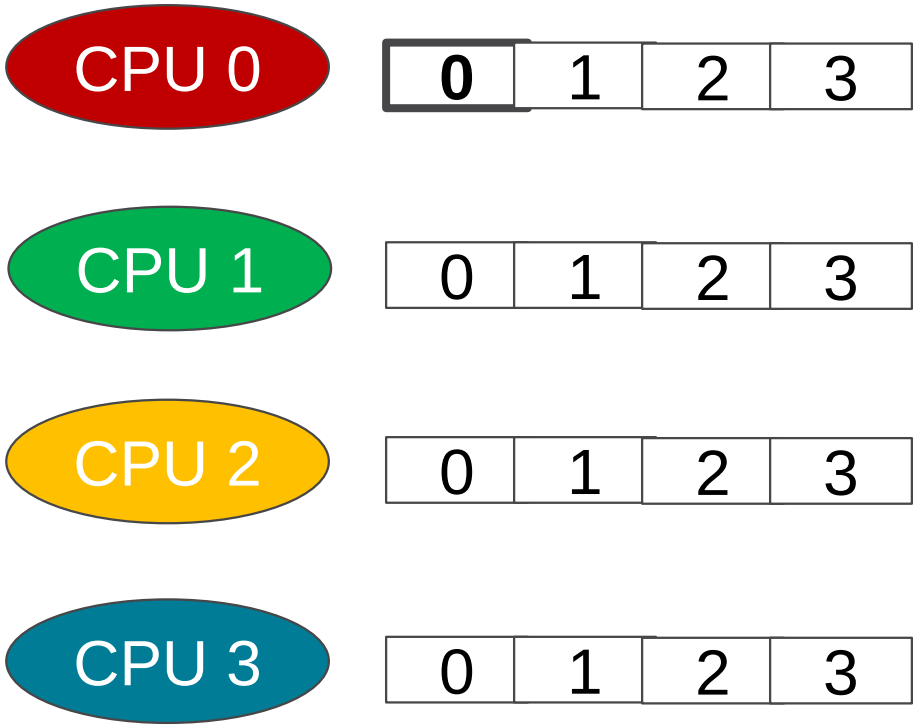


CPU 0 then atomically exchanges the tail of the qspinlock with its tail encoding (1, 0).

The old tail is (0, 0).

1	**0**	0	1	1
CPU Number	idx	Unused	Pending	Lock

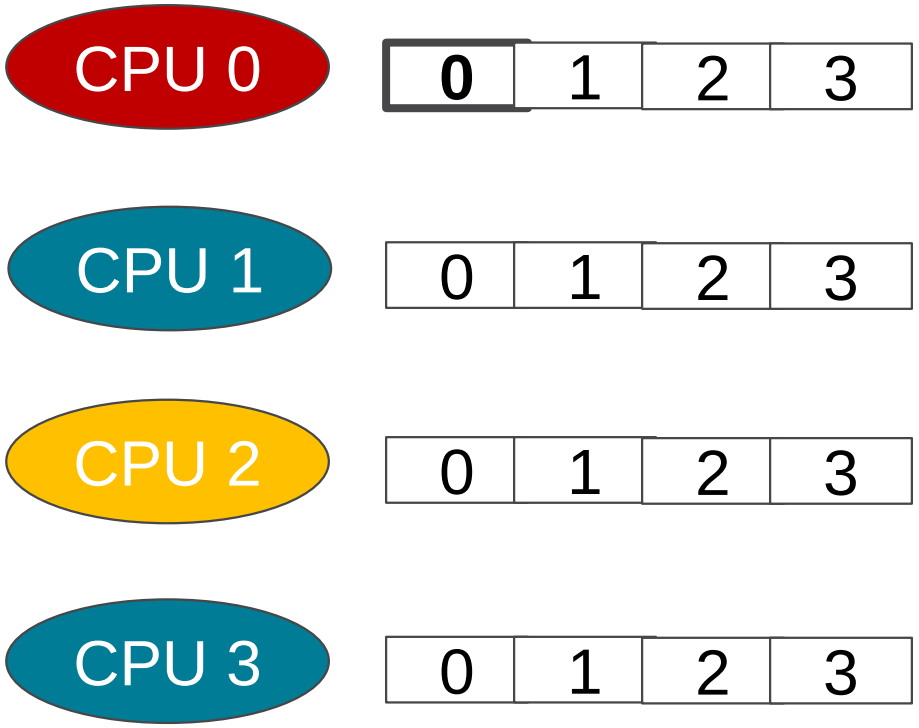
Queued Spin Locks



Since the old tail is (0, 0), CPU0 knows that it is at the head of the waiters. It just spins until (Pending, Lock) becomes (0, 0).

1	**0**	0	1	1
CPU Number	idx	Unused	Pending	Lock

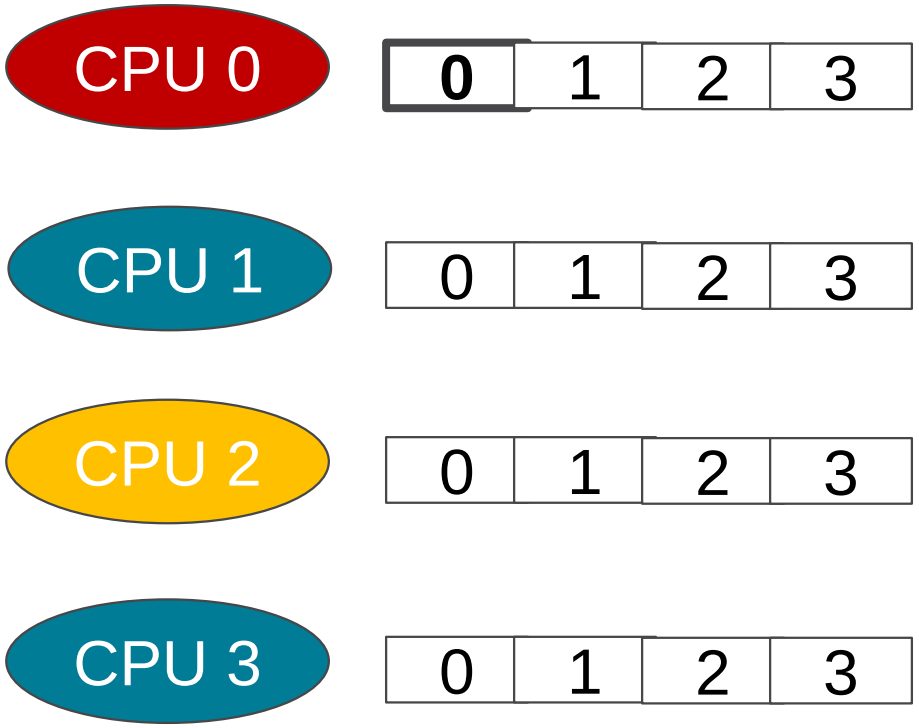
Queued Spin Locks



CPU 1 is done with the lock. It releases it by setting the Lock byte to 0.

1	0	0	1	**0**
CPU Number	idx	Unused	Pending	Lock

Queued Spin Locks

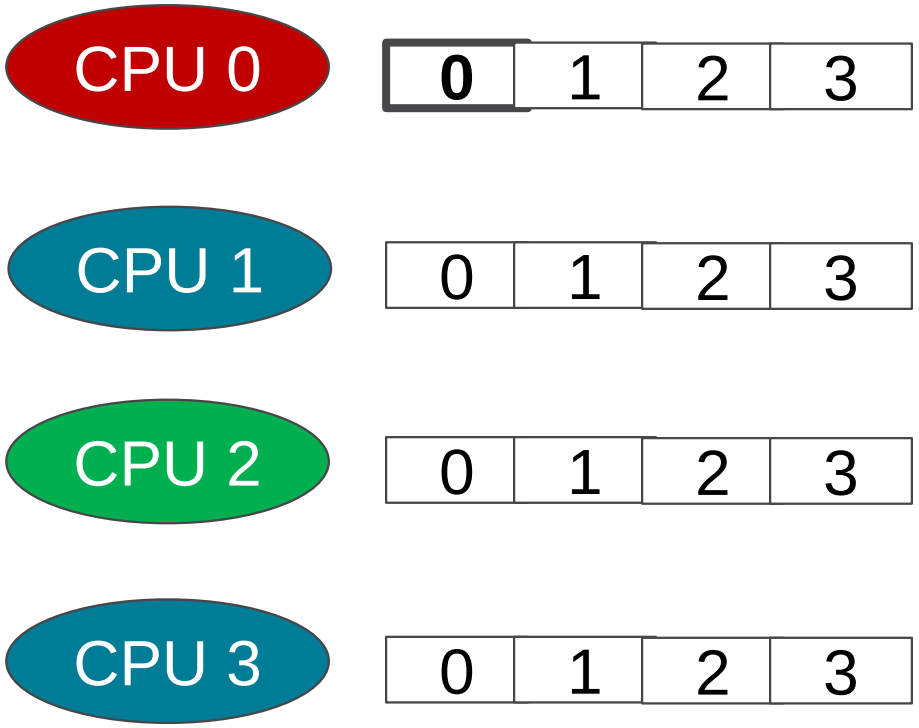


As CPU 0 is waiting for (Pending, Lock) to be (0, 0) and since pending bit is set, it still has to wait.

However, CPU 2 is waiting for the Lock byte to be 0 which it is.

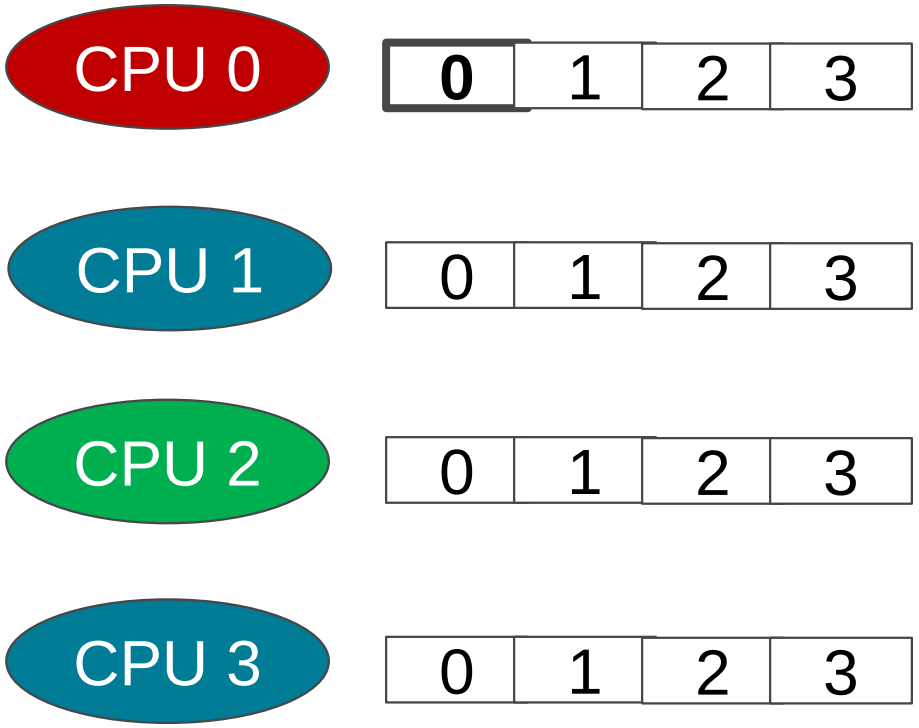
1	0	0	1	**0**
CPU Number	idx	Unused	Pending	Lock

Queued Spin Locks

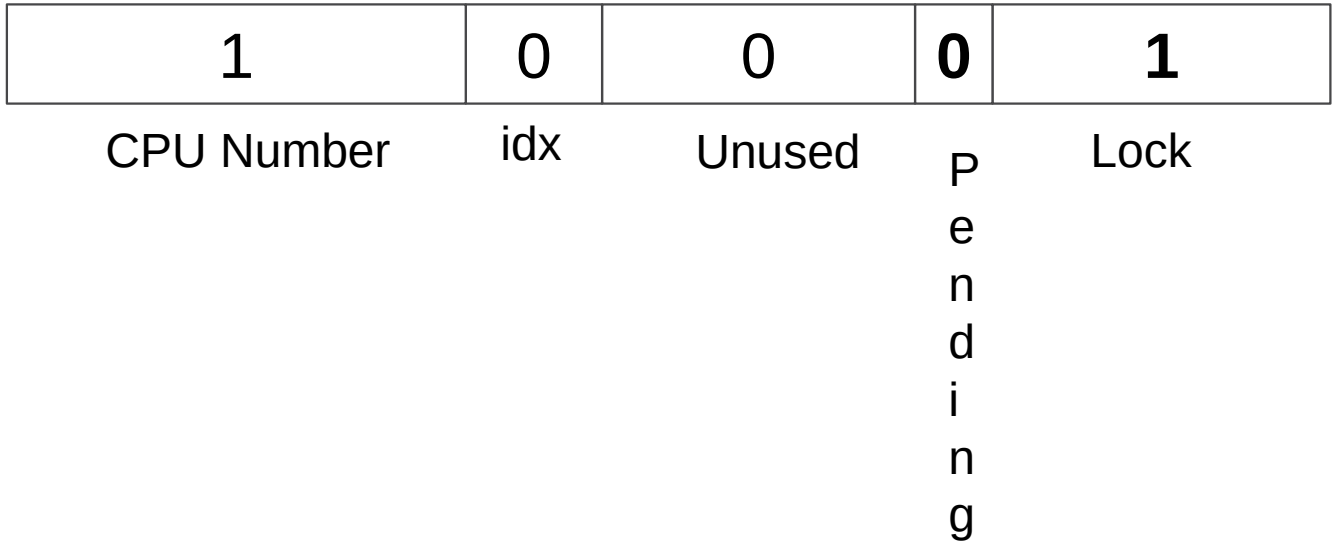


CPU 2 atomically sets (Pending, Lock) to (0, 1)

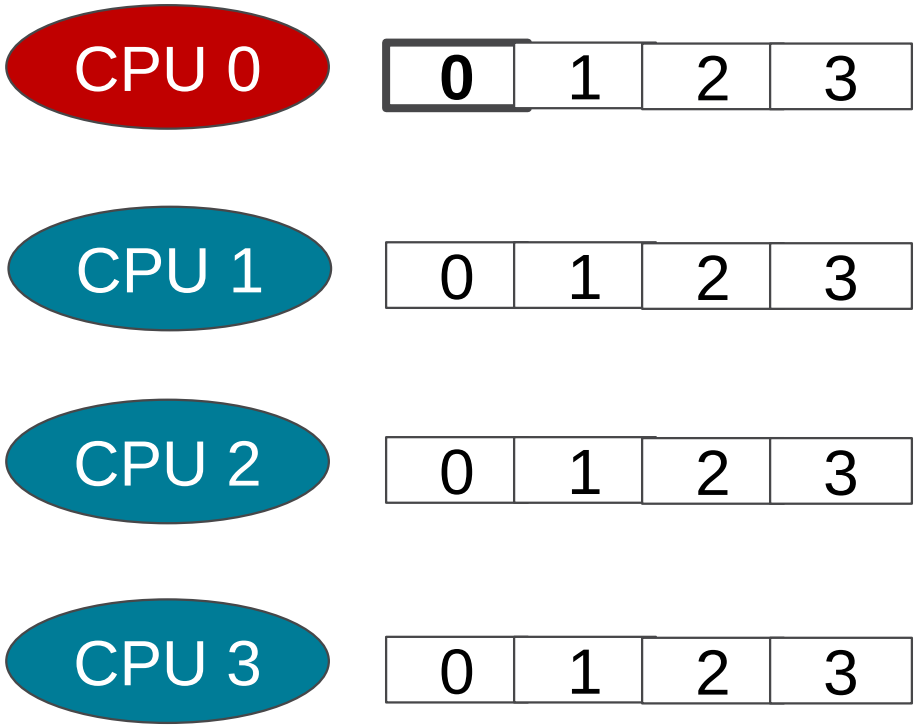
Queued Spin Locks



CPU 0 still has to wait because it needs to have (Pending, Lock) to be (0, 0).

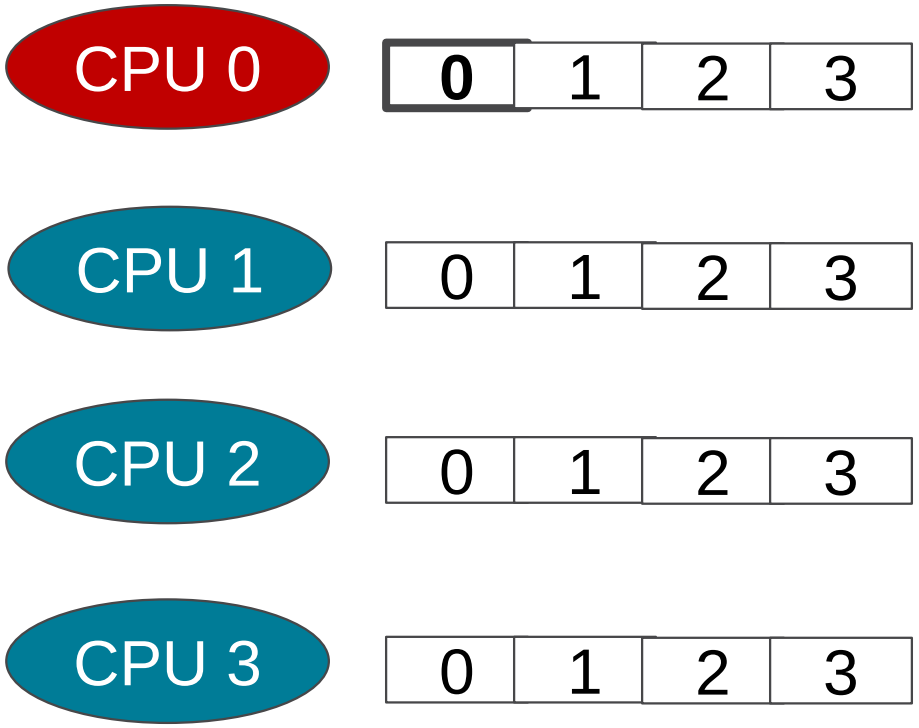


Queued Spin Locks

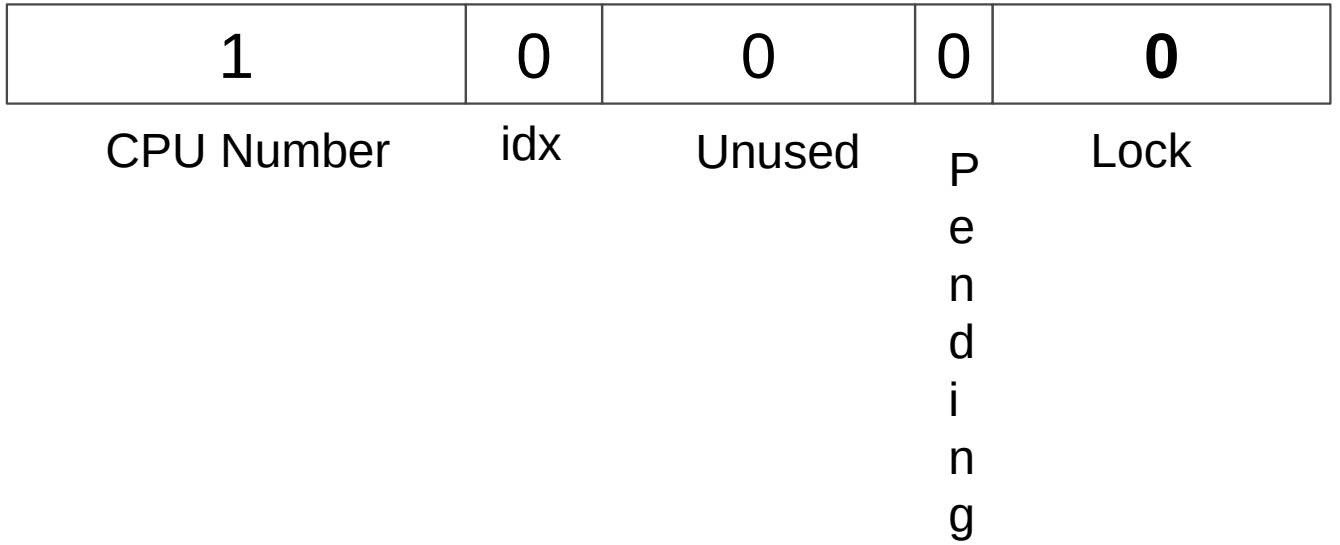


CPU 2 is done. It will set Lock byte to 0 and release the lock

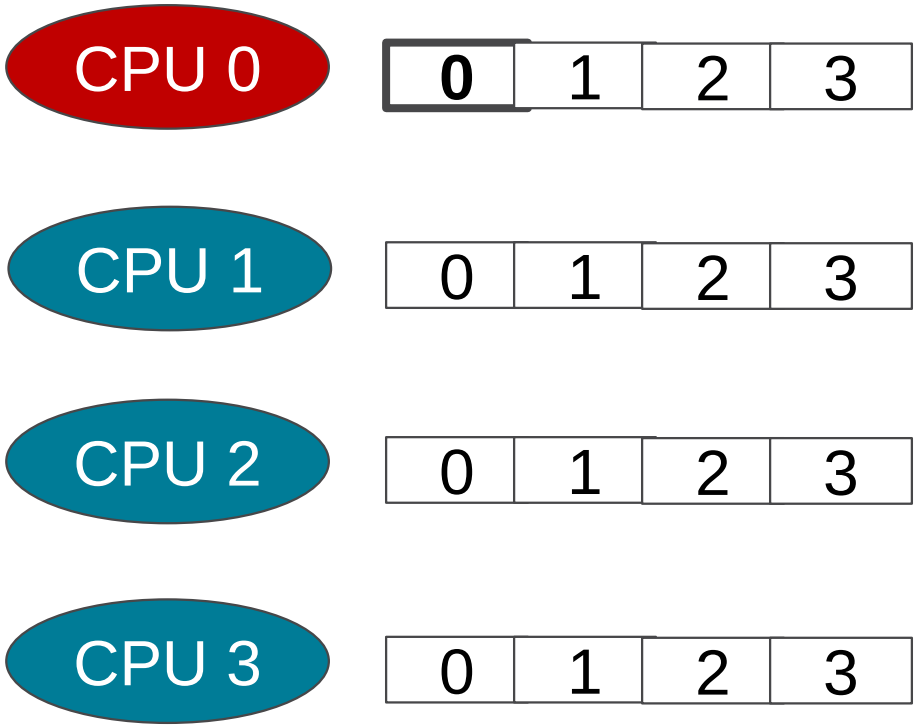
Queued Spin Locks



CPU 0 notices (Pending, Lock) = (0, 0).
It checks if it is the tail, i.e.,
qspinlock.tail == (CPU0 + 1, CPU0. idx).

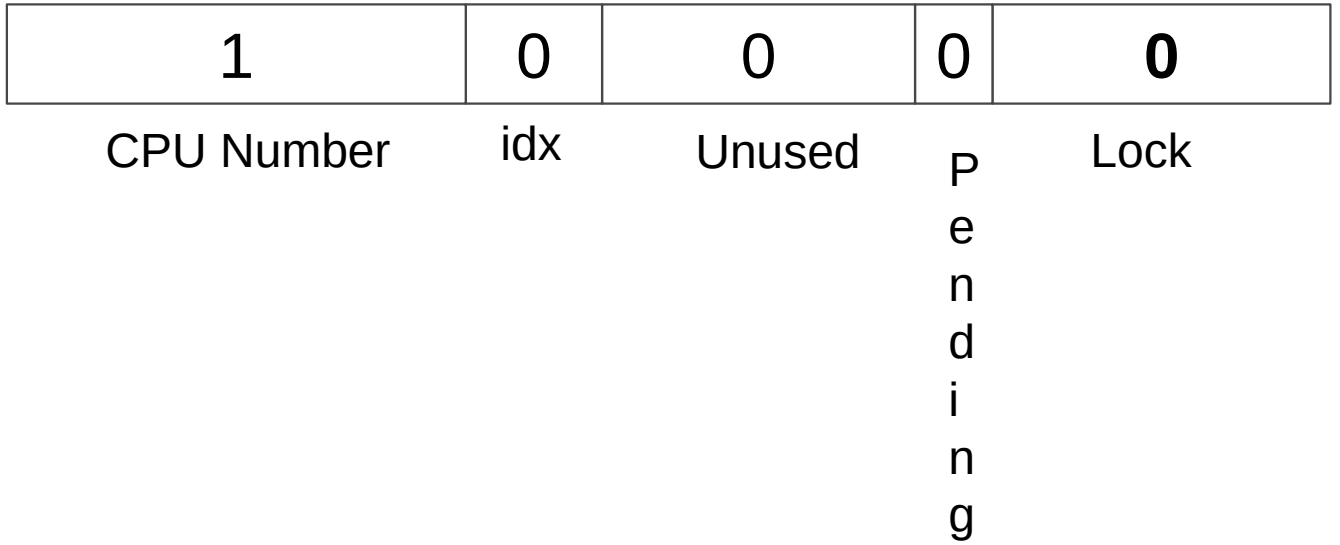


Queued Spin Locks

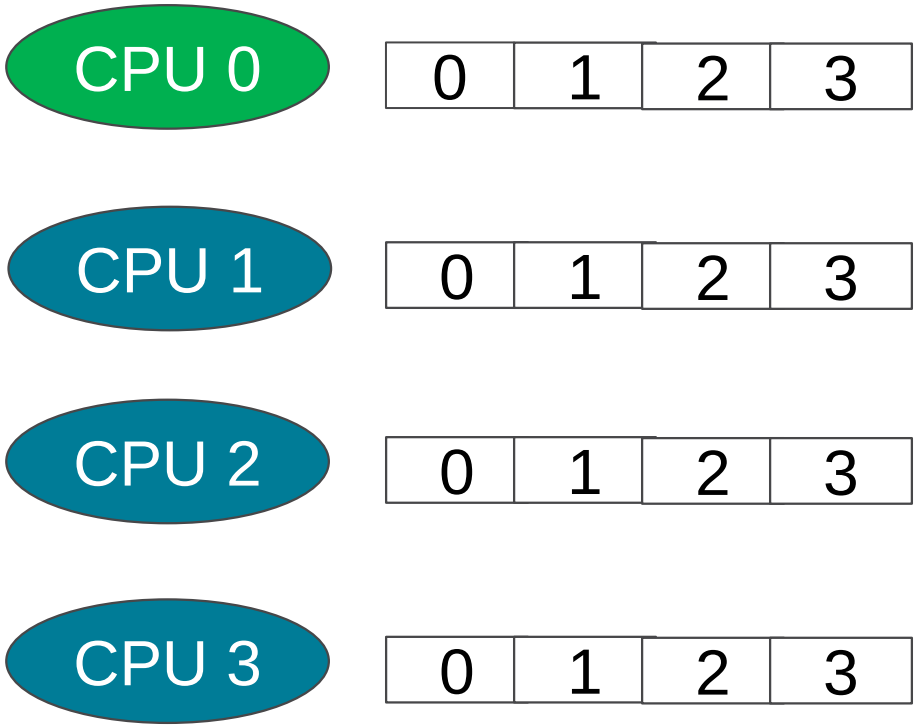


CPU 0 notices (Pending, Lock) = (0, 0).
It checks if it is the tail, i.e.,
qspinlock.tail == (CPU0 + 1, CPU0. idx).

As it is true, it atomically tries to compare exchange the 32-bit qspinlock dword from (1, 0, 0, 0, 0) to (0, 0, 0, 0, 1) thus clearing CPU number, idx and setting the Lock byte.



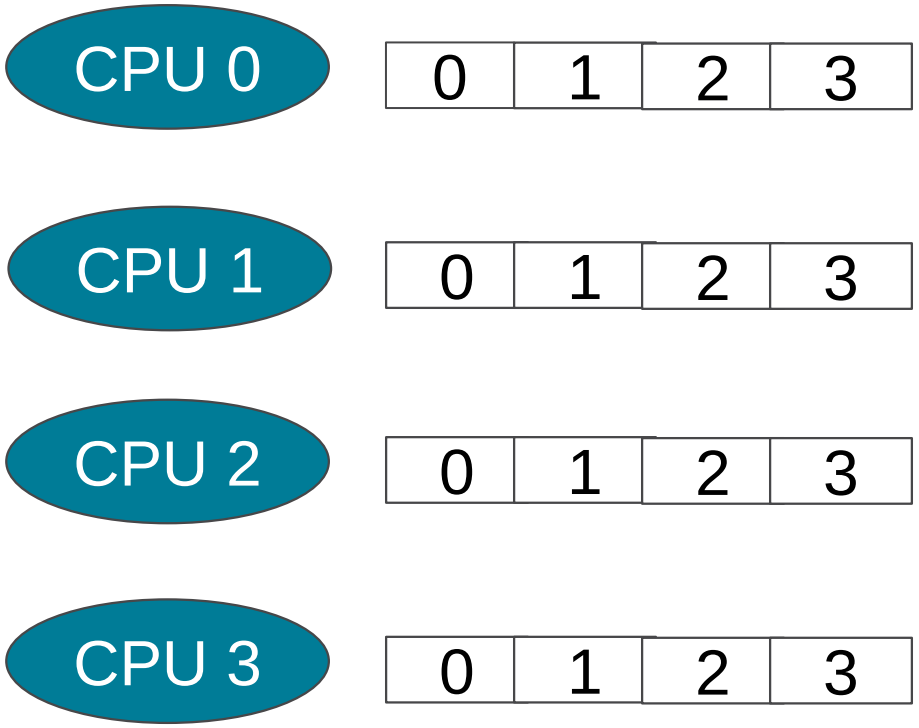
Queued Spin Locks



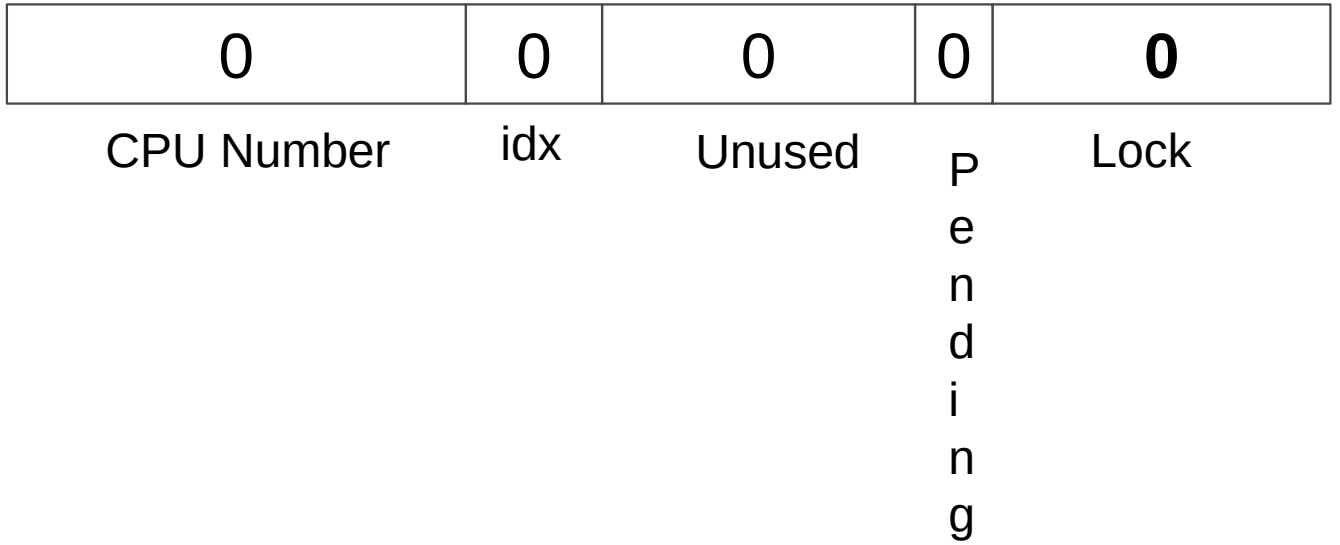
CPU 0 succeeds in doing the atomic compare exchange and acquires the lock. It also decrements CPU0.qnode[0].mcs_spinlock.count to release the qnode.

0	**0**	0	0	**1**
CPU Number	idx	Unused	Pending	Lock

Queued Spin Locks

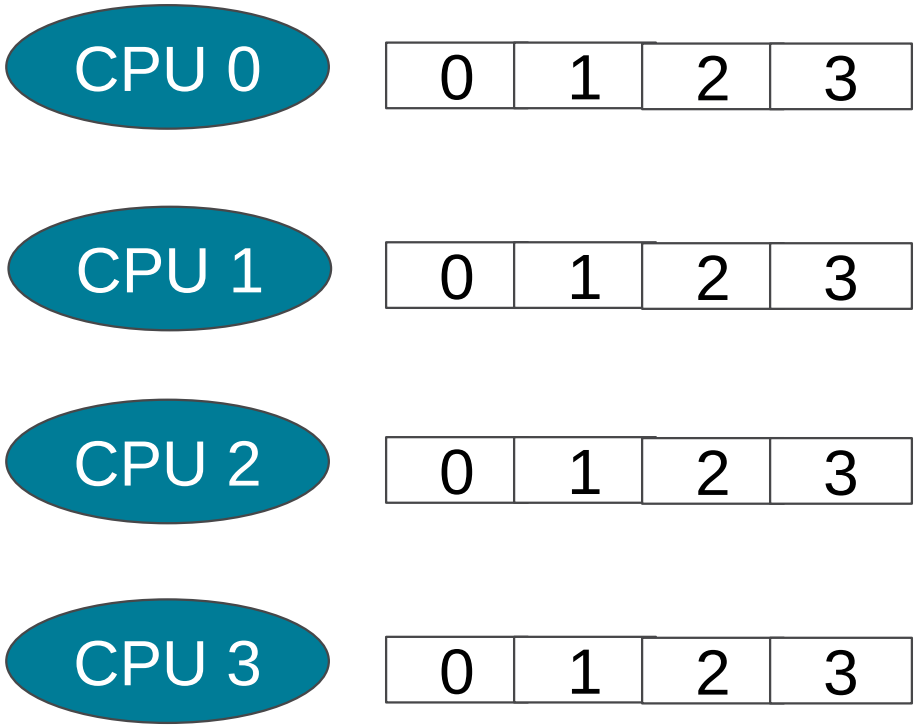


Once CPU 0 is done, it will release the lock by clearing the Lock byte.

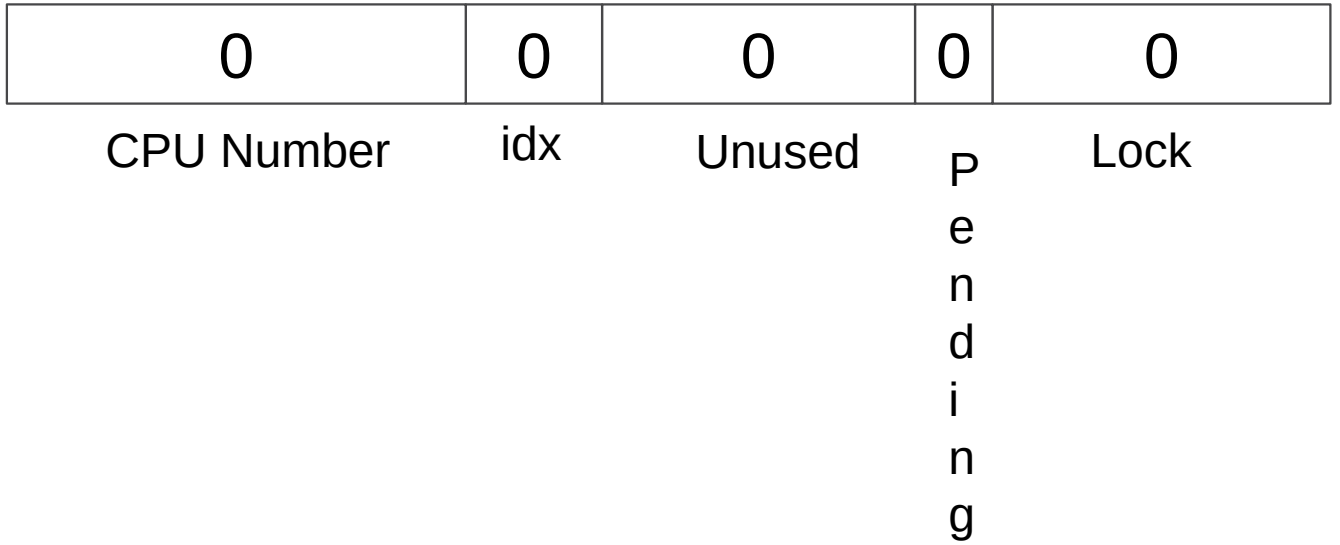


Four (or more) contenders

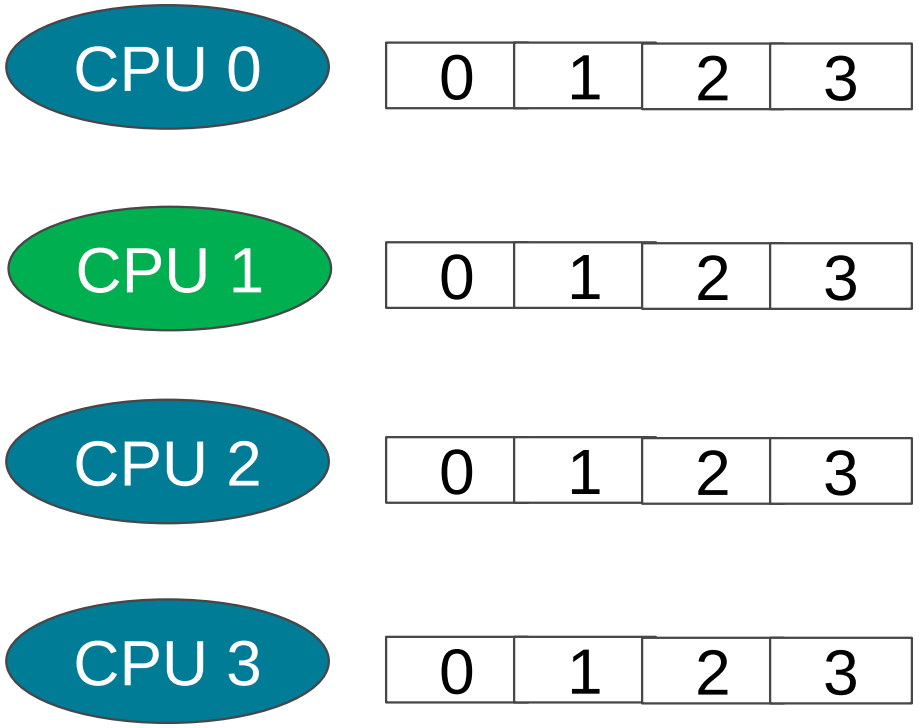
Queued Spin Locks



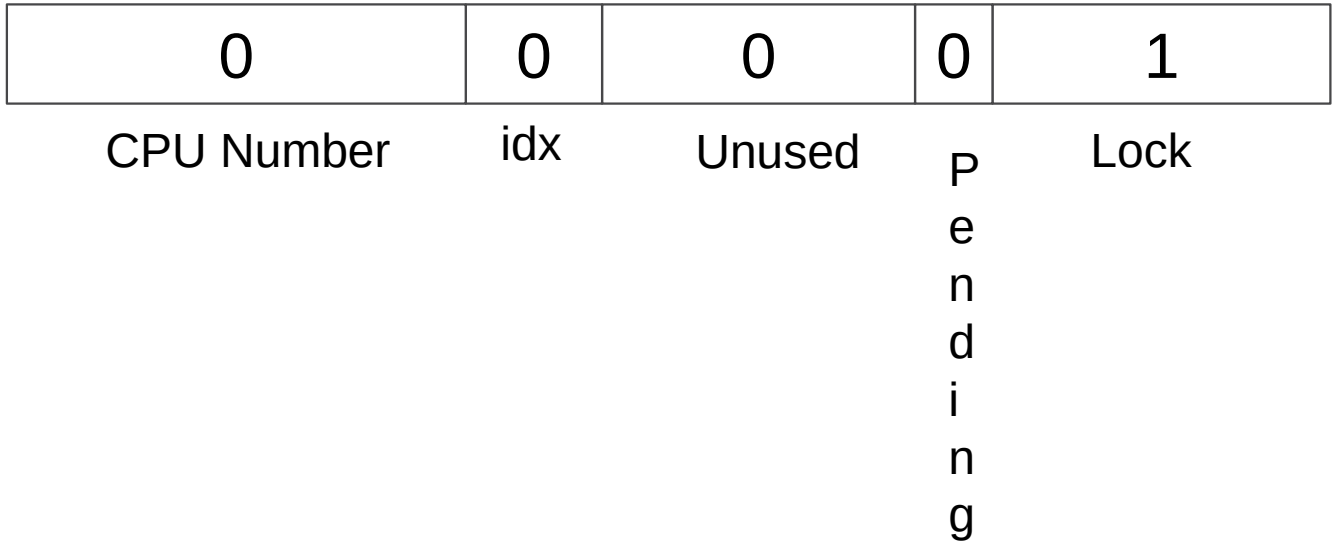
CPUs 0, 1, 2: All concurrently try to get the lock by compare exchanging 0 with Q_LOCKED (value = 1).



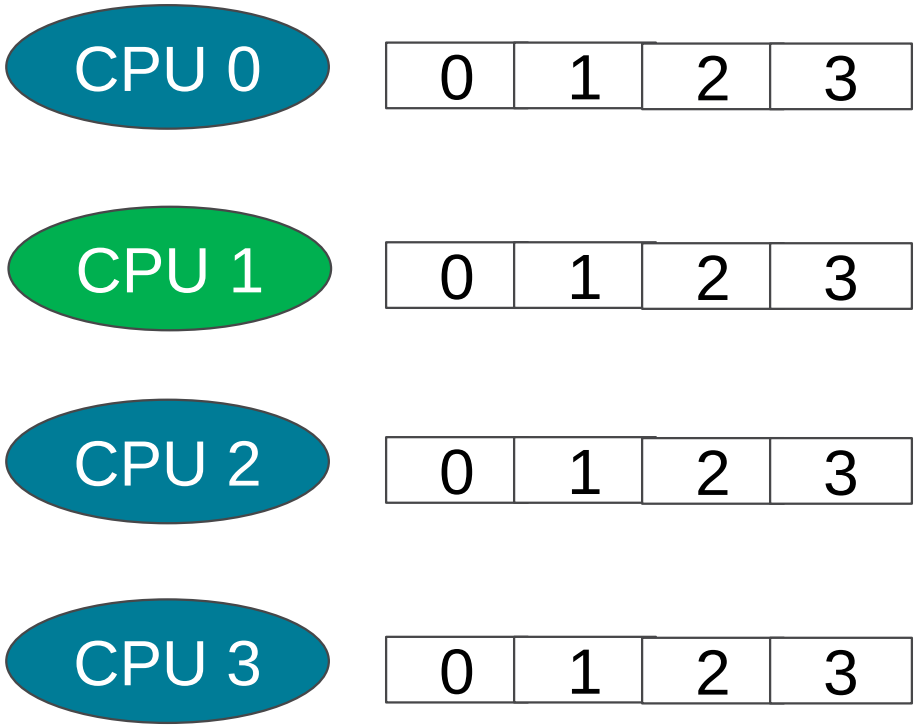
Queued Spin Locks



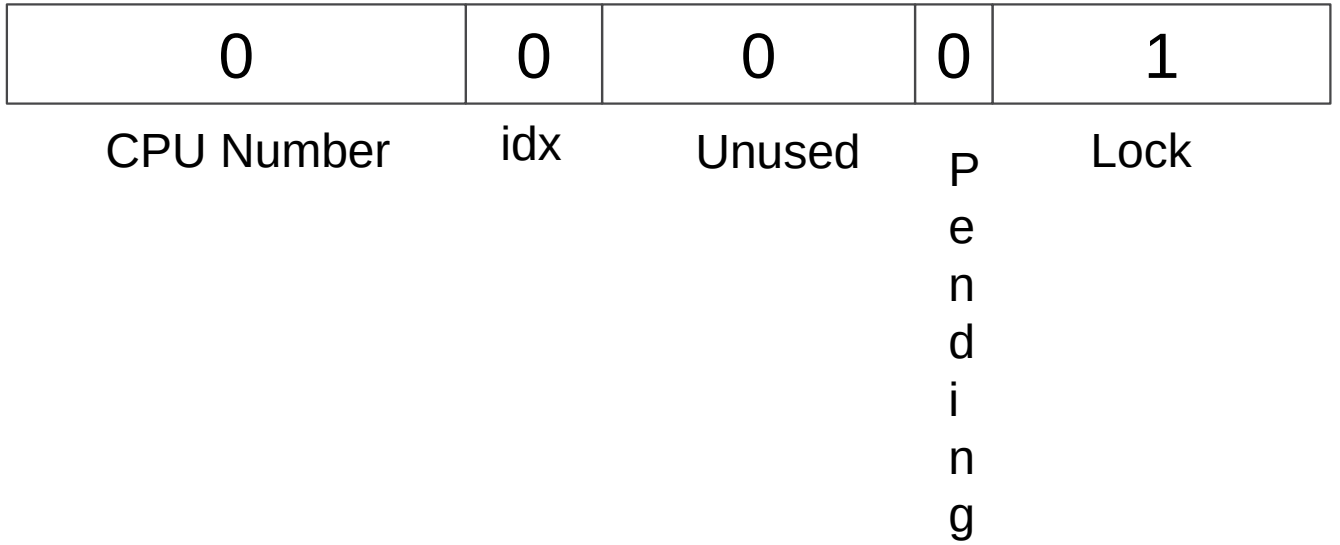
WLOG, let us assume that CPU 1 wins the race and is able to set the Lock bit to 1.



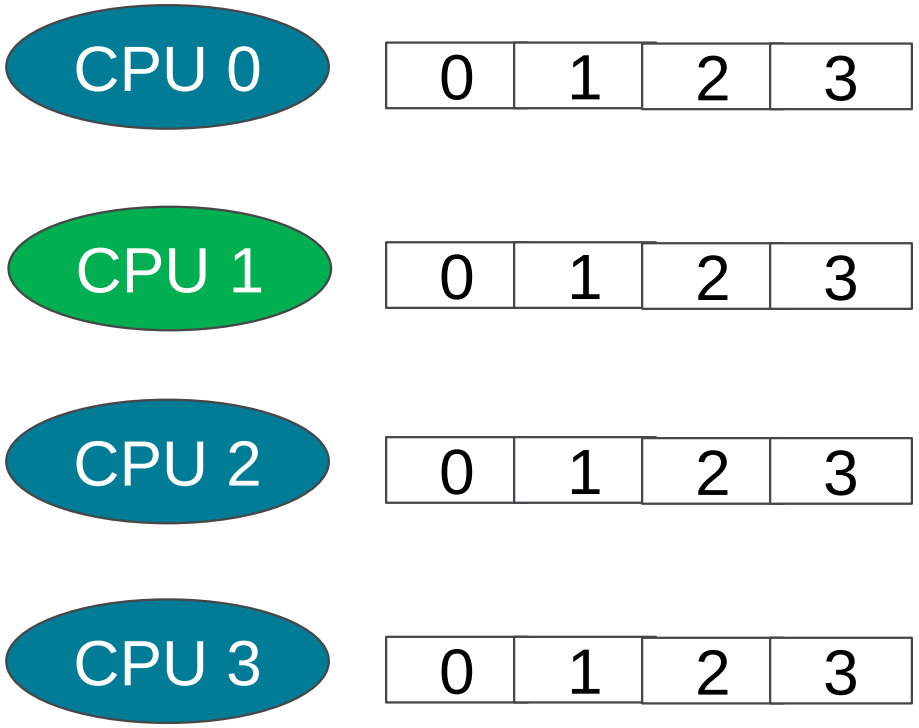
Queued Spin Locks



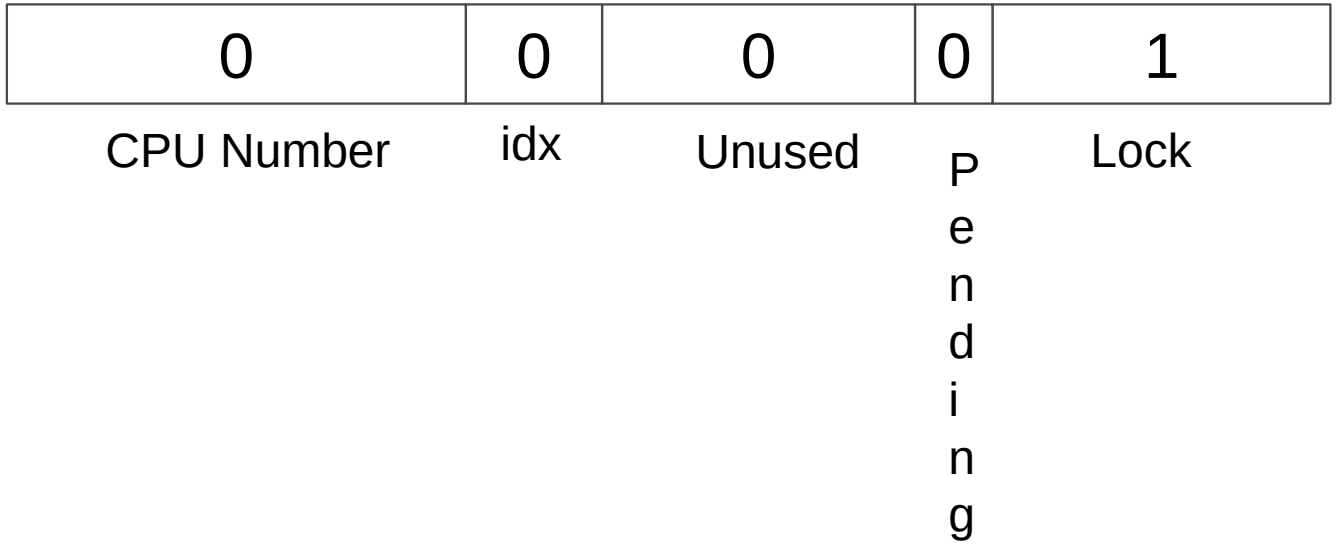
CPUs 0 and 2 check any of the bits other than Lock byte is set. If they find it so, they go to the queuing phase.



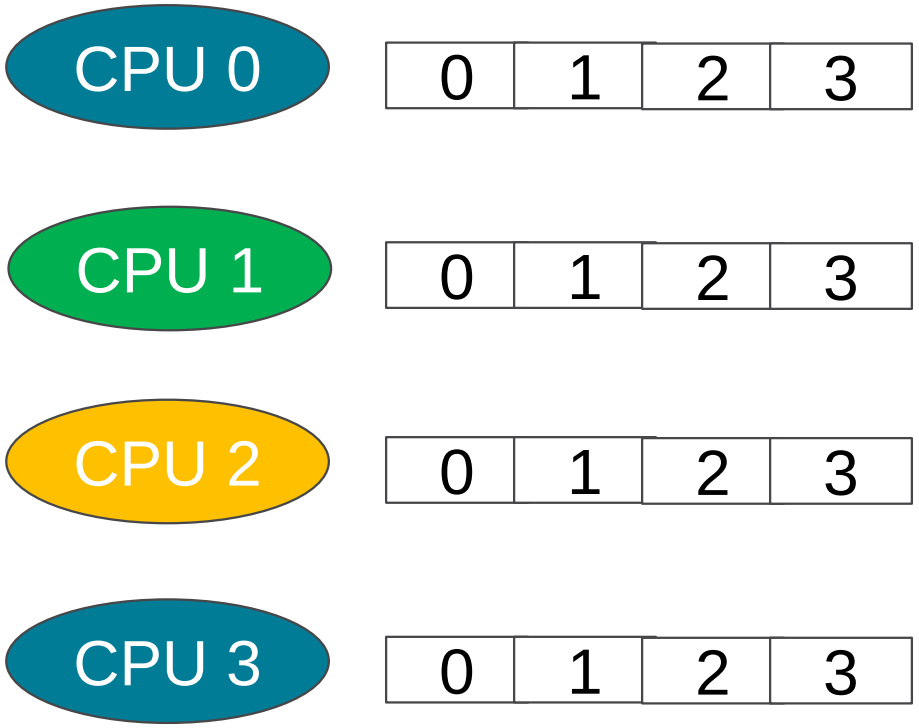
Queued Spin Locks



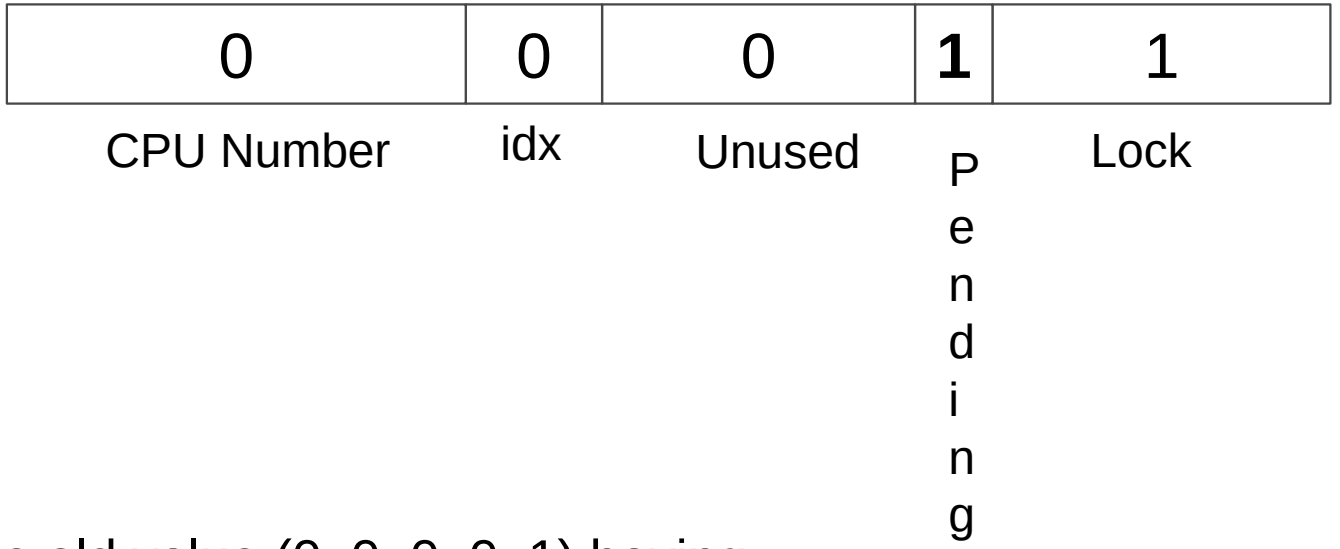
In this case, since CPUs 0 and 2 don't see any bits set, they try to atomically set the pending bit and get the old value (using atomic_fetch_or)



Queued Spin Locks

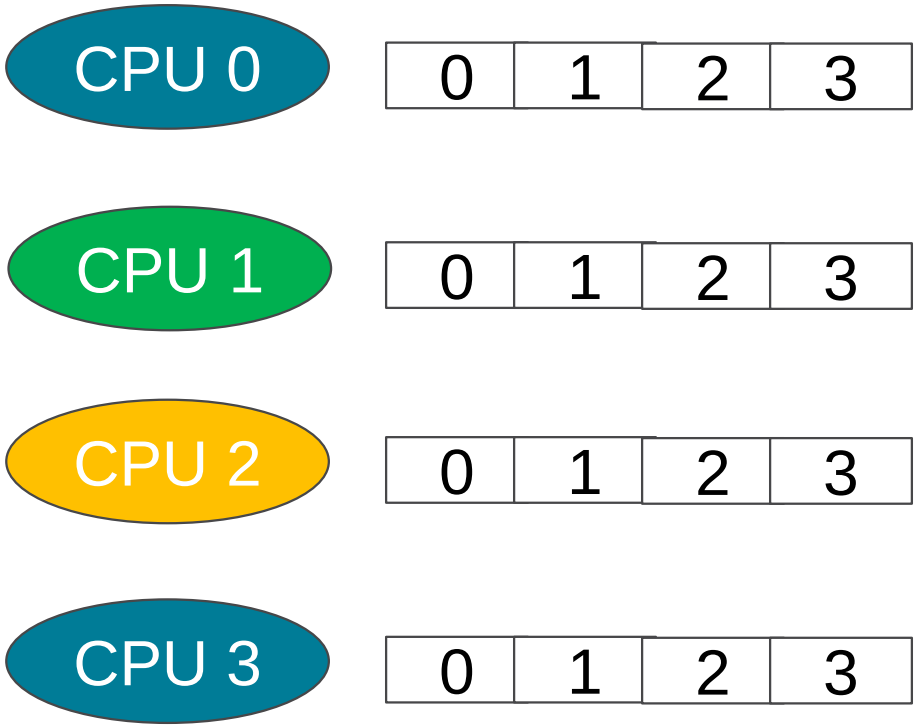


In this case, since CPUs 0 and 2 don't see any bits set, they try to atomically set the pending bit and get the old value (using atomic_fetch_or)

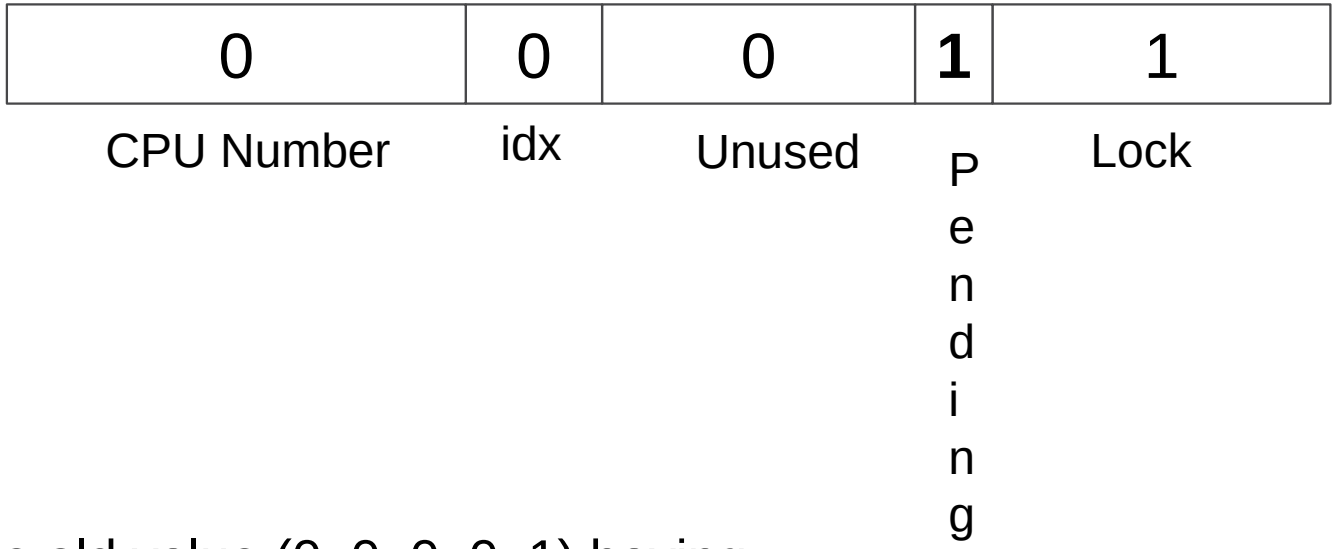


Assume that CPU 2 wins the race. It will read the old value (0, 0, 0, 0, 1) having updated the pending bit.
CPU 0 will read the old value (0, 0, 0, 1, 1) having updated the pending bit.

Queued Spin Locks

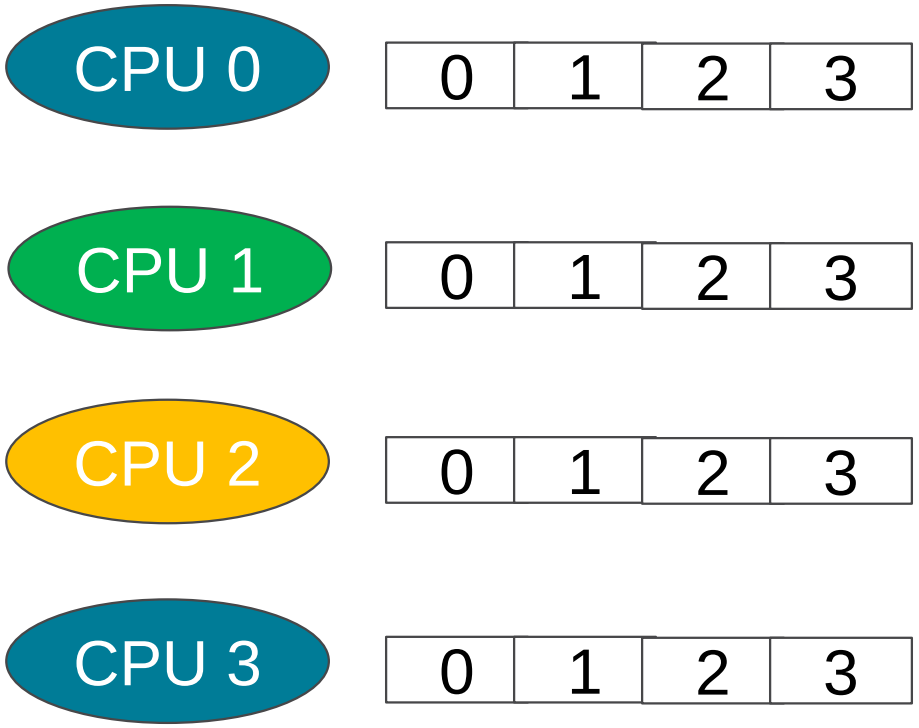


CPU 2 knows that it has successfully set the pending bit and no other bits are set. Hence it is next in line. It just spins until the Lock byte becomes 0.

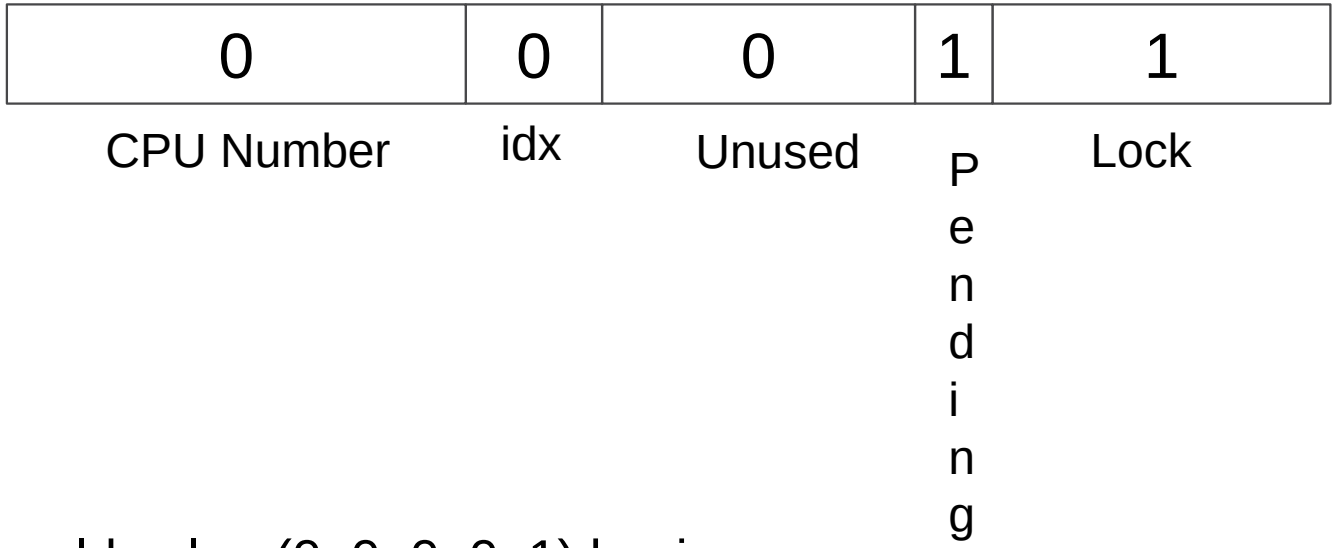


Assume that CPU 2 wins the race. It will read the old value (0, 0, 0, 0, 1) having updated the pending bit.
CPU 0 will read the old value (0, 0, 0, 1, 1) having updated the pending bit.

Queued Spin Locks

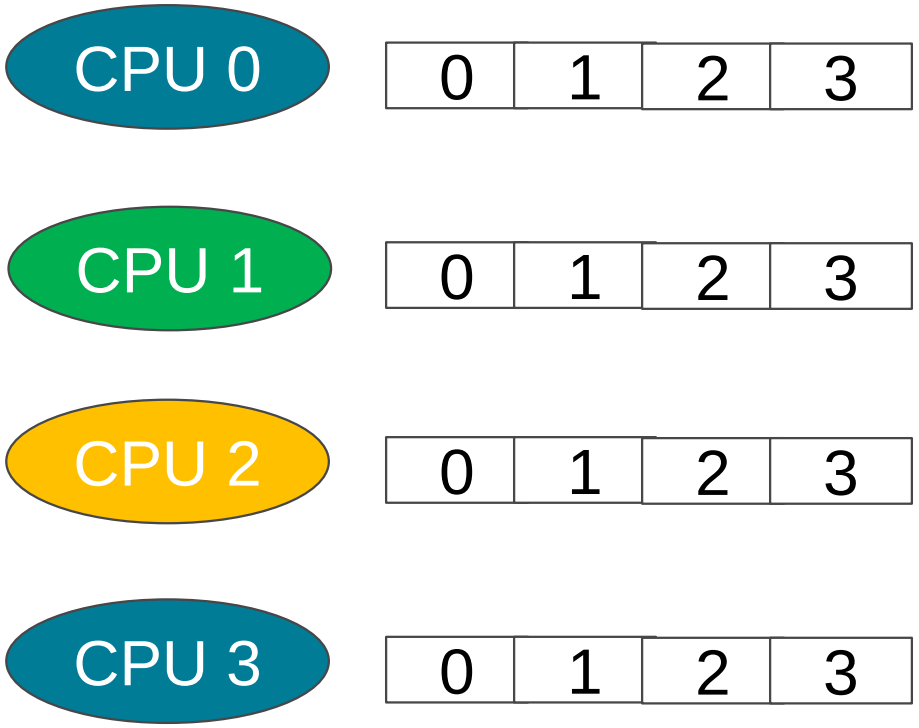


CPU 0 knows that it wasn't the first to set the pending bit. So, it has to queue.

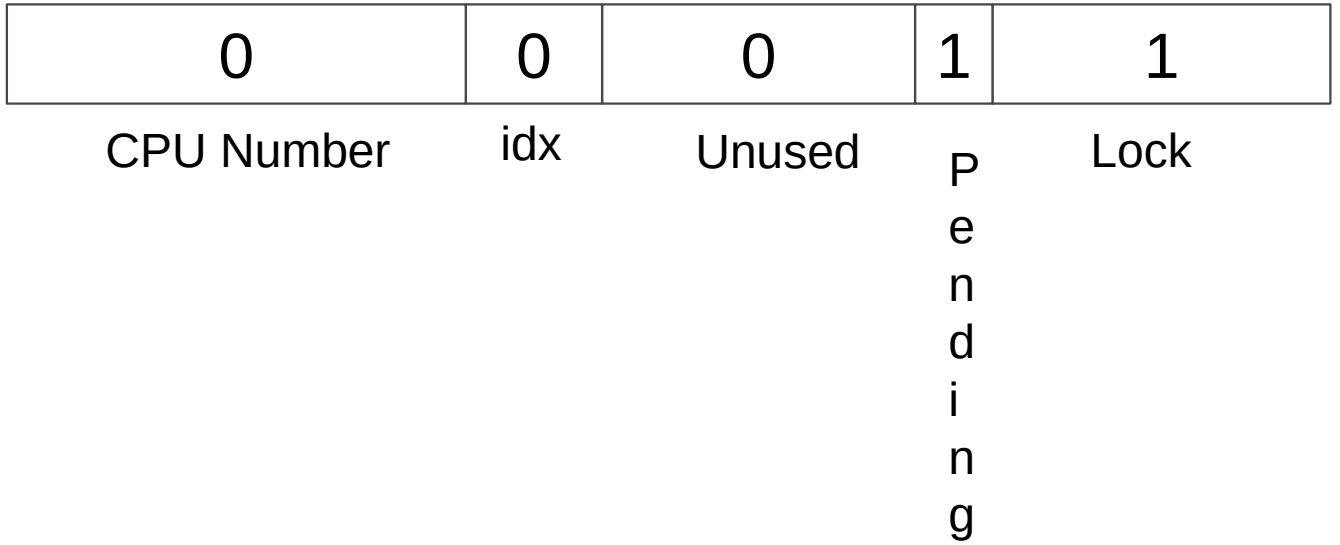


Assume that CPU 2 wins the race. It will read the old value (0, 0, 0, 0, 1) having updated the pending bit.
CPU 0 will read the old value (0, 0, 0, 1, 1) having updated the pending bit.

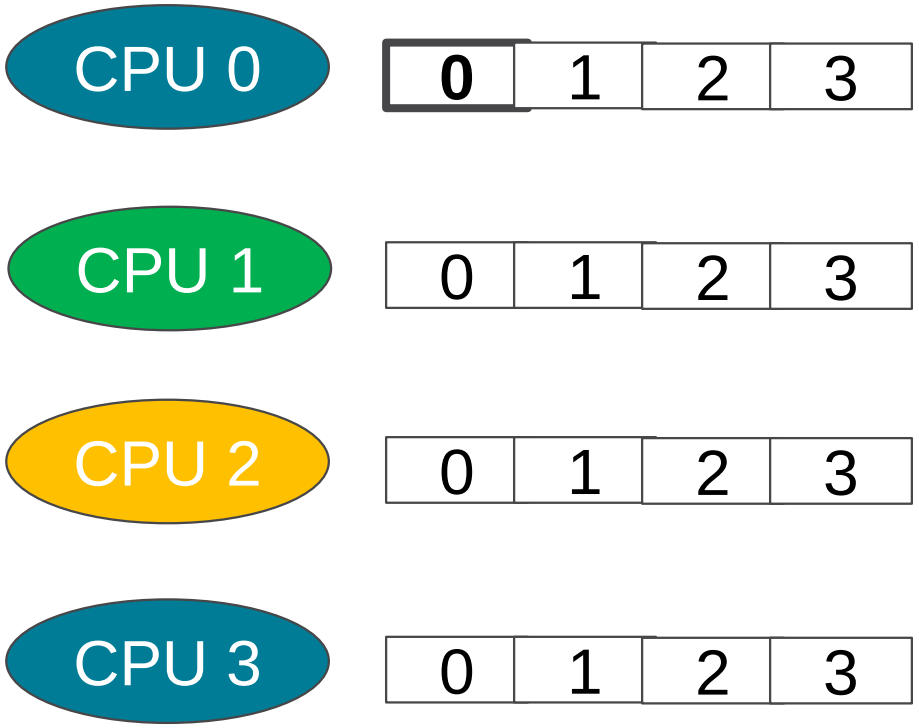
Queued Spin Locks



CPU 0 grabs the first available qnode. It does so by checking and incrementing qnode.mcs_spinlock.count.

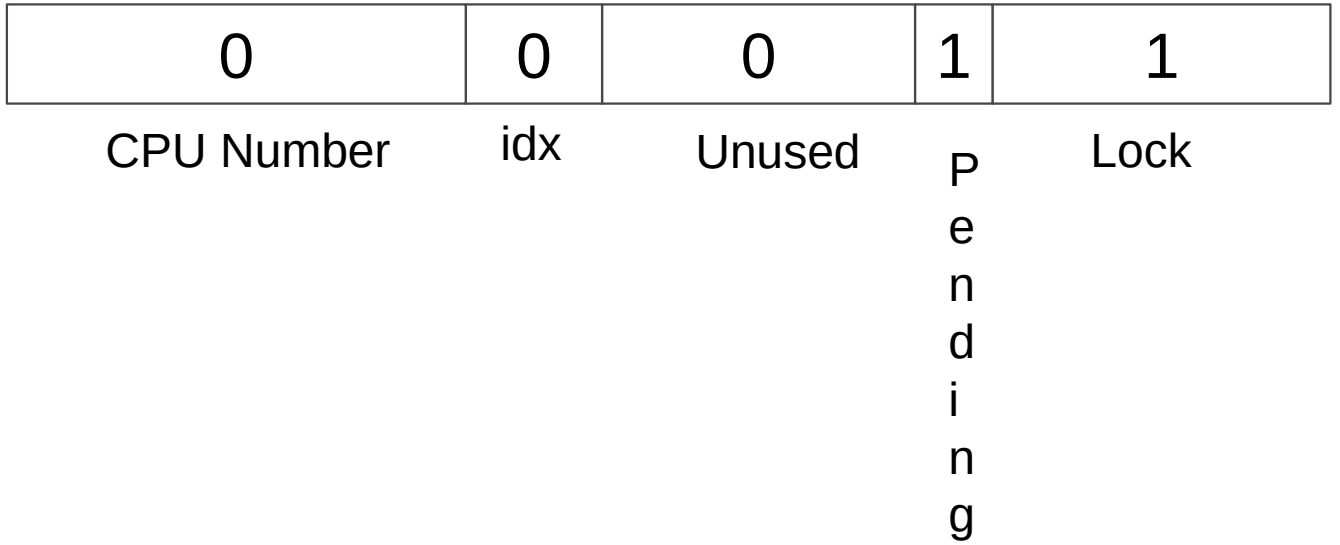


Queued Spin Locks

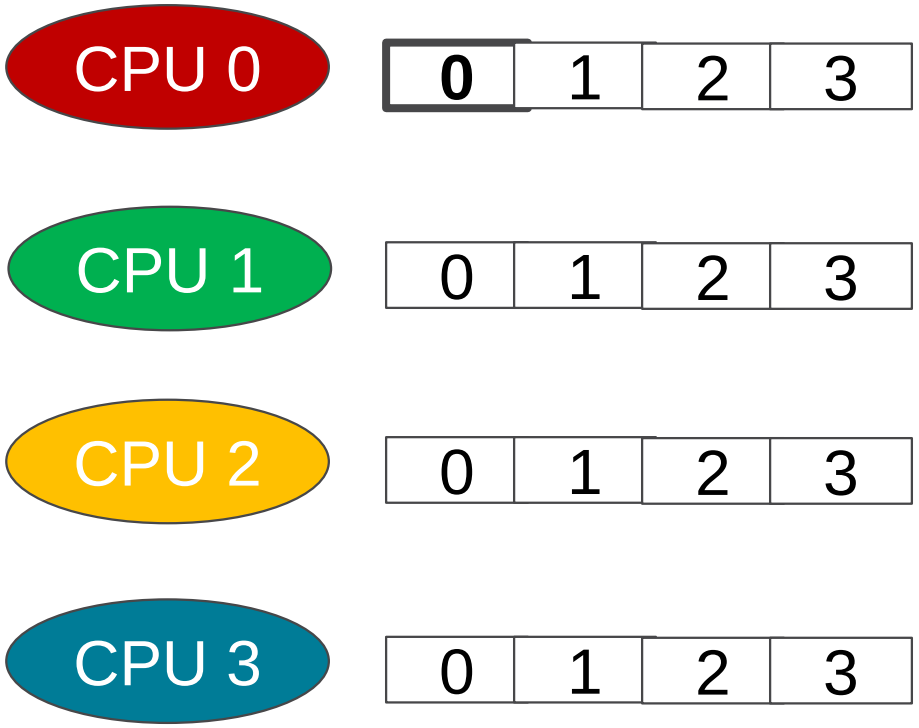


WLOG assume CPU 0 has the idx 0 available.

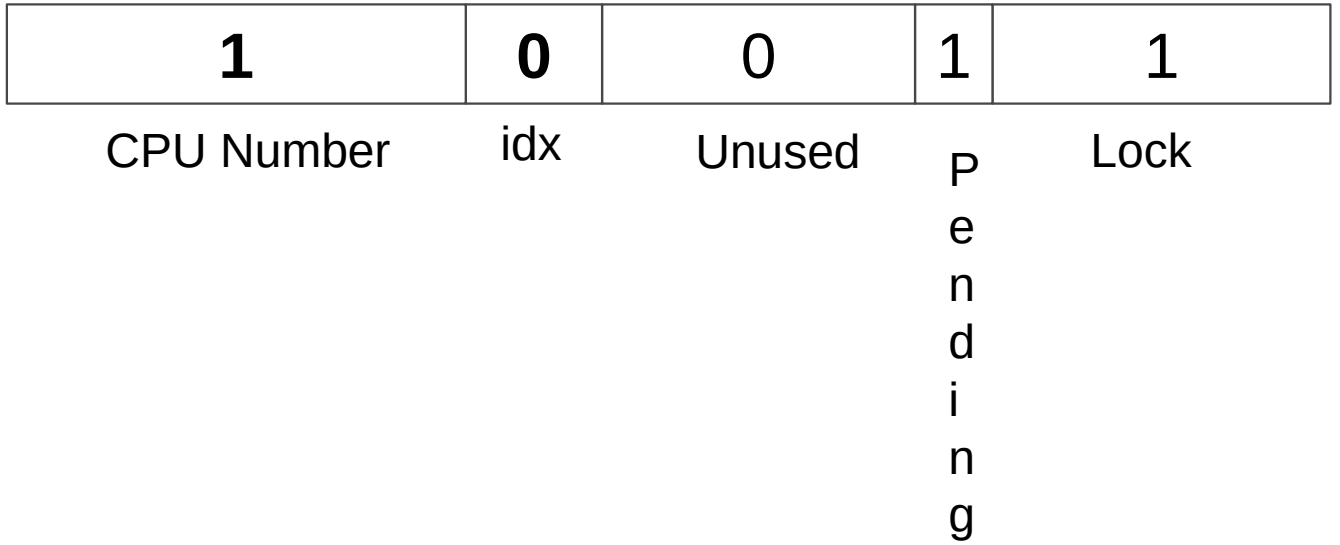
It then generates the tail-encoding as (CPU Number +1 , idx) = (1, 0).



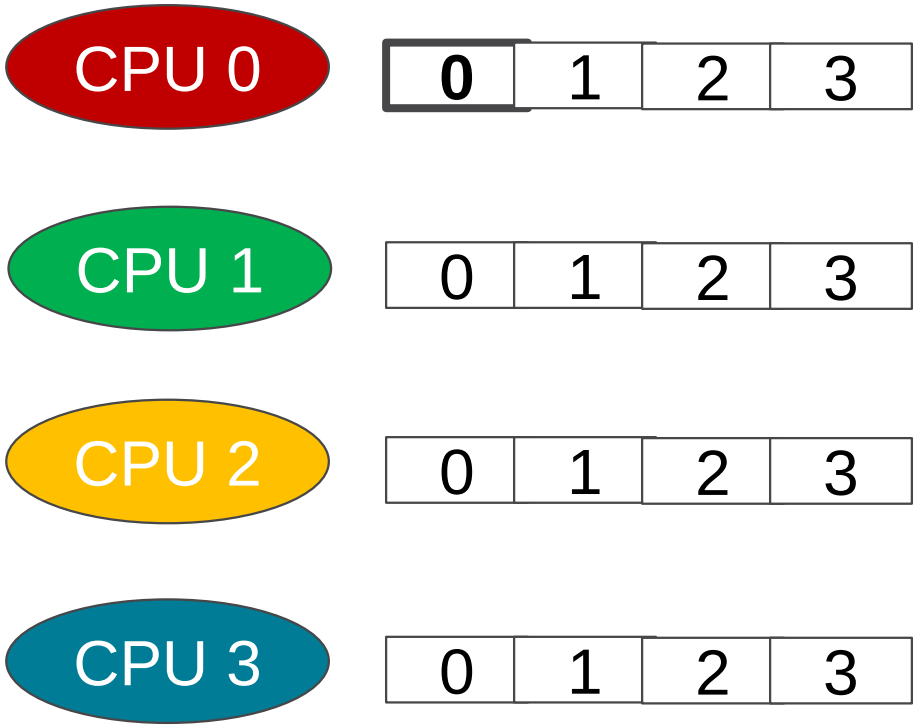
Queued Spin Locks



CPU 0 then atomically exchanges the tail of the qspinlock with its tail encoding (1, 0).
The old tail is (0, 0).



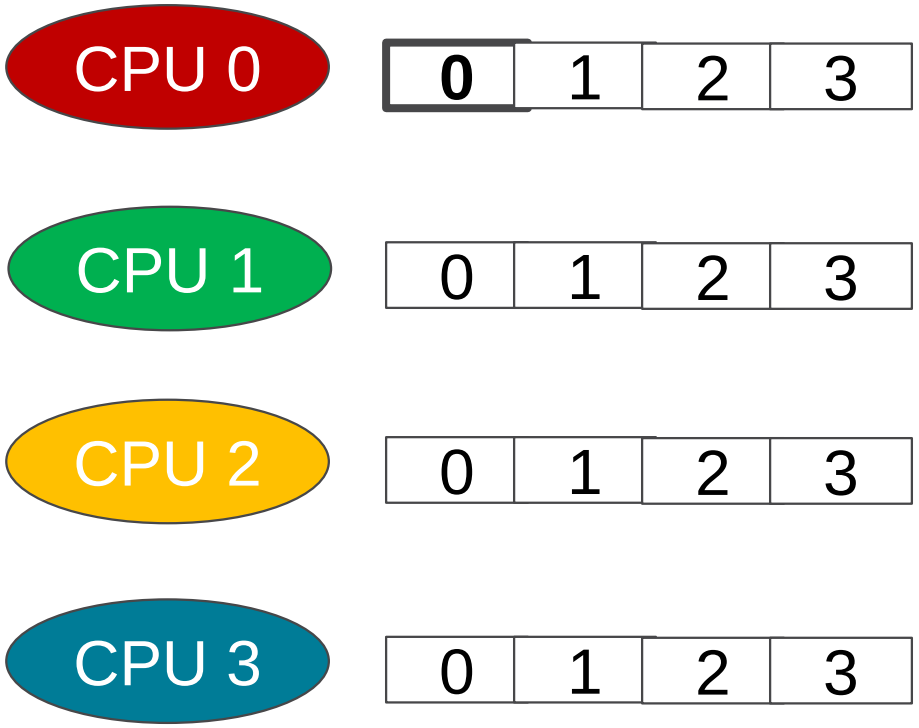
Queued Spin Locks



Since the old tail is (0, 0), CPU0 knows that it is at the head of the waiters. It just spins until (Pending, Lock) becomes (0, 0)

1	**0**	0	1	1
CPU Number	idx	Unused	Pending	Lock

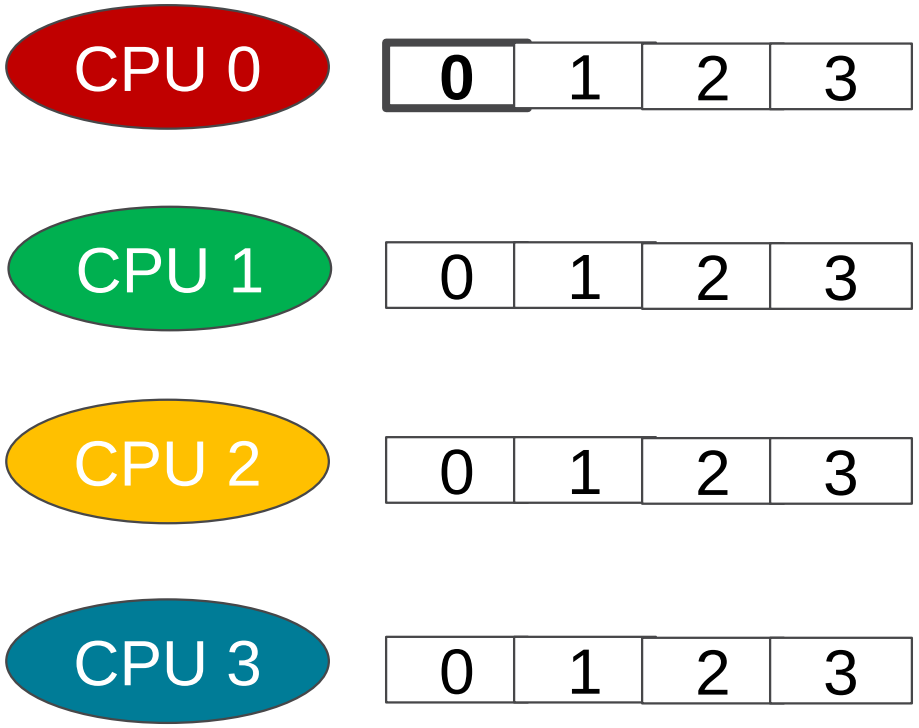
Queued Spin Locks



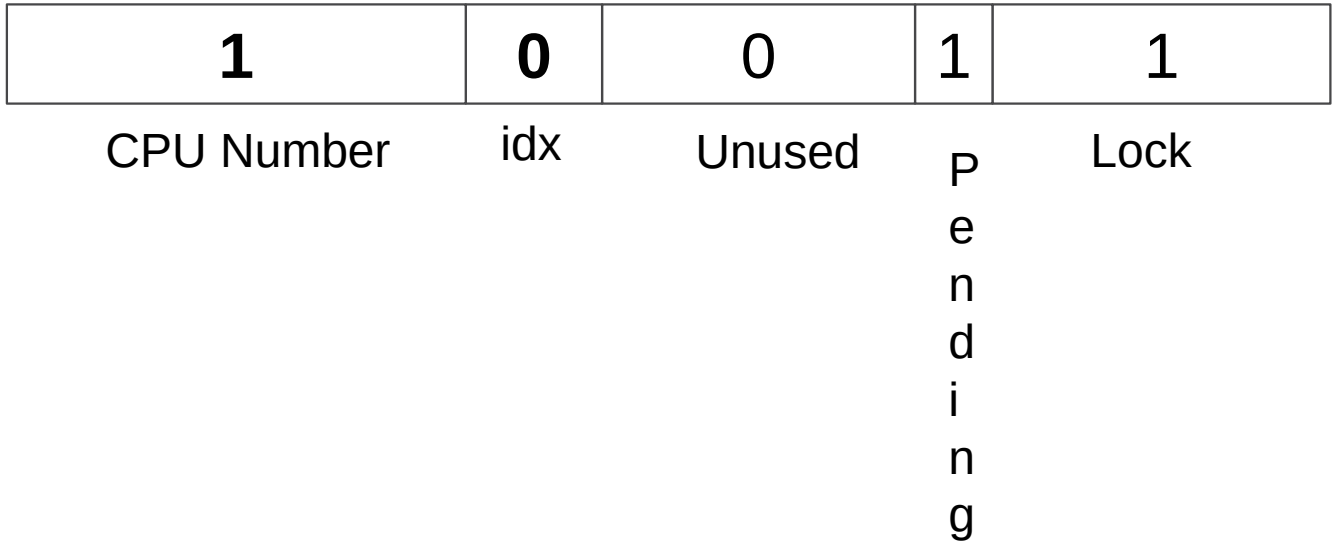
Now CPU3 attempts to get the lock. The attempt to compare exchange 32-bit qspinlock variable from 0 to 1 fails.

1	**0**	0	1	1
CPU Number	idx	Unused	Pending	Lock

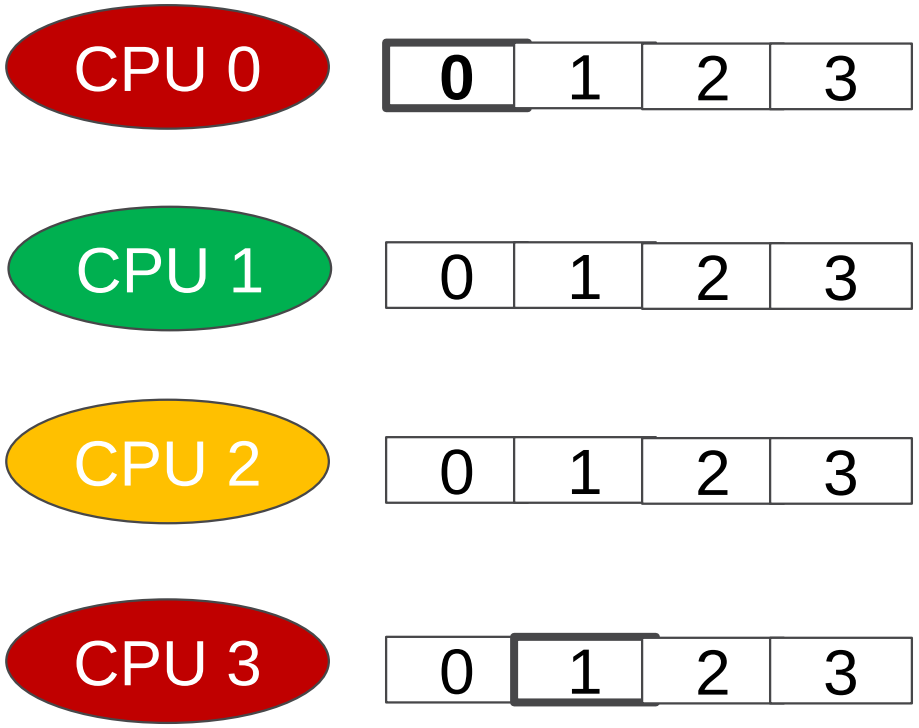
Queued Spin Locks



CPU3 checks if apart from the Lock byte any other bits are set. They are. So, CPU 3 has to queue.



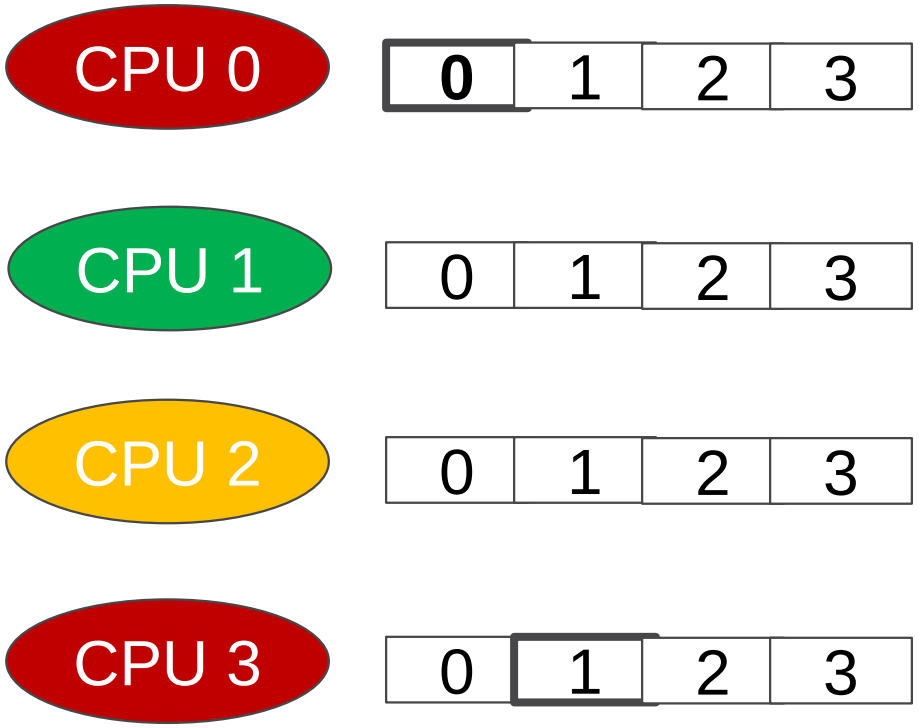
Queued Spin Locks



CPU 3 checks qnodes[0].mcs_spinlock.count. WLOG assume it is 1. Which means that qnode[0] is already taken in another context. So, it grabs qnode[1] after incrementing qnodes[0].mcs_spinlock.count to 2.

1	**0**	0	1	1
CPU Number	idx	Unused	Pending	Lock

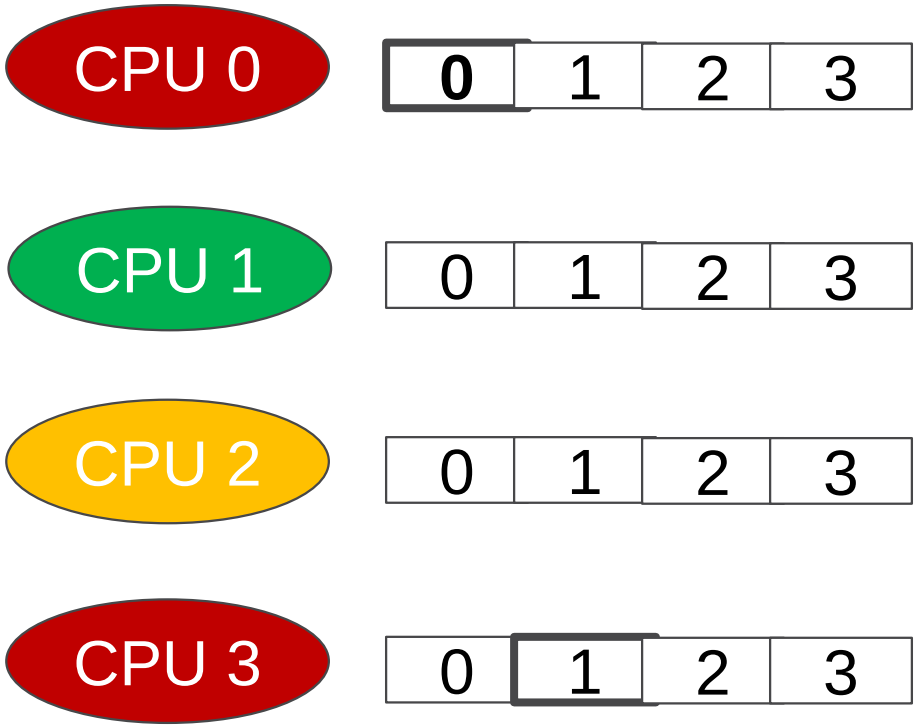
Queued Spin Locks



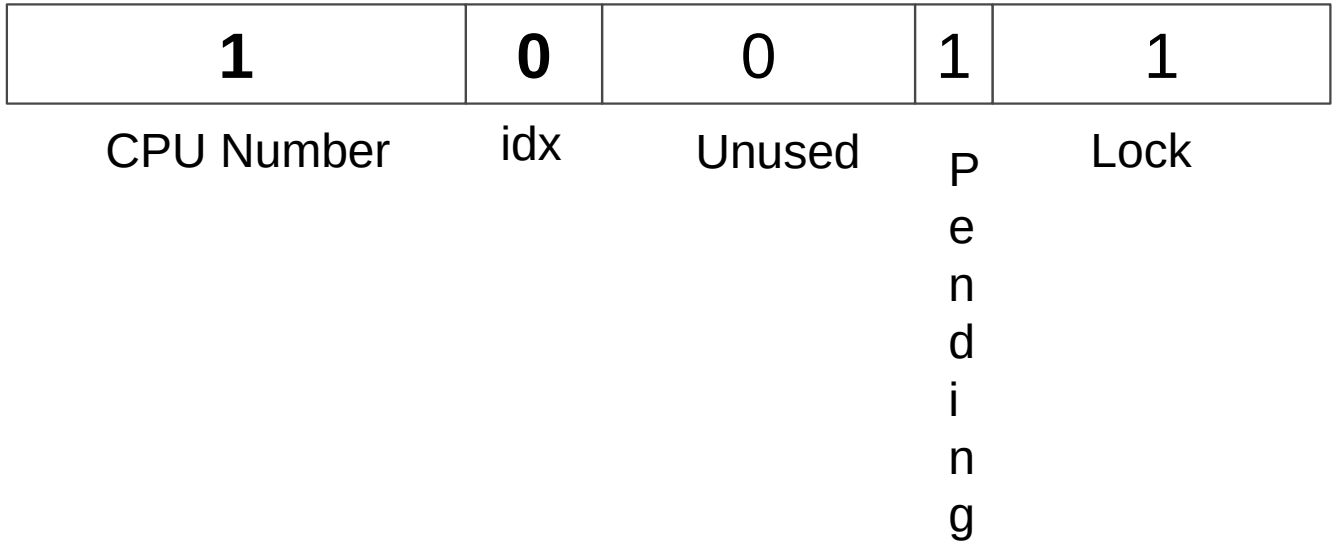
CPU 3 then sets
qnode[1].mcs_spinlock.locked = 0.
qnode[1].mcs_spinlock.next = NULL.

1	**0**	0	1	1
CPU Number	idx	Unused	Pending	Lock

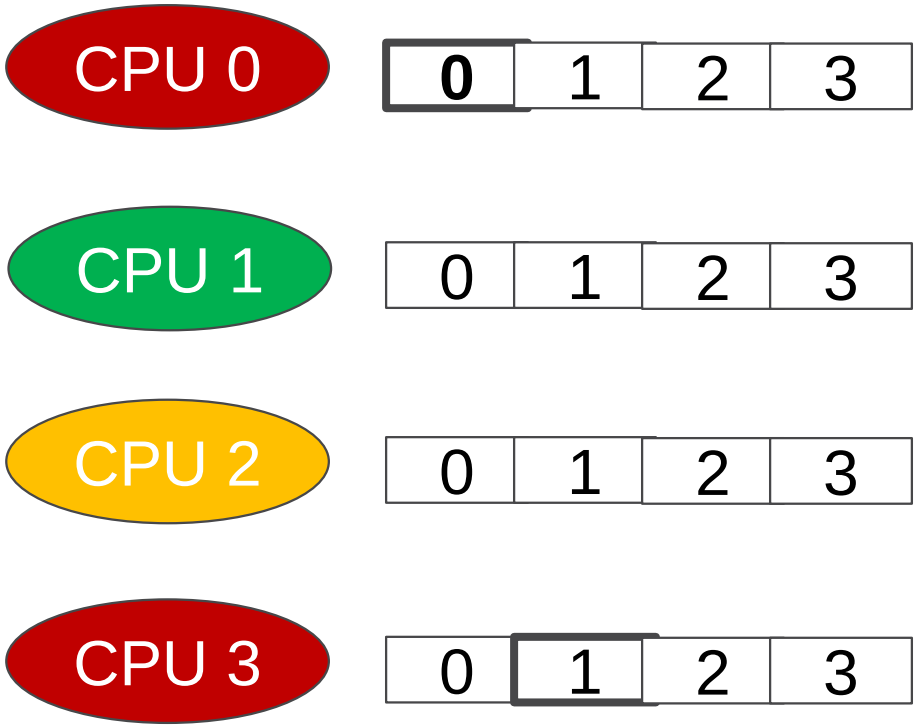
Queued Spin Locks



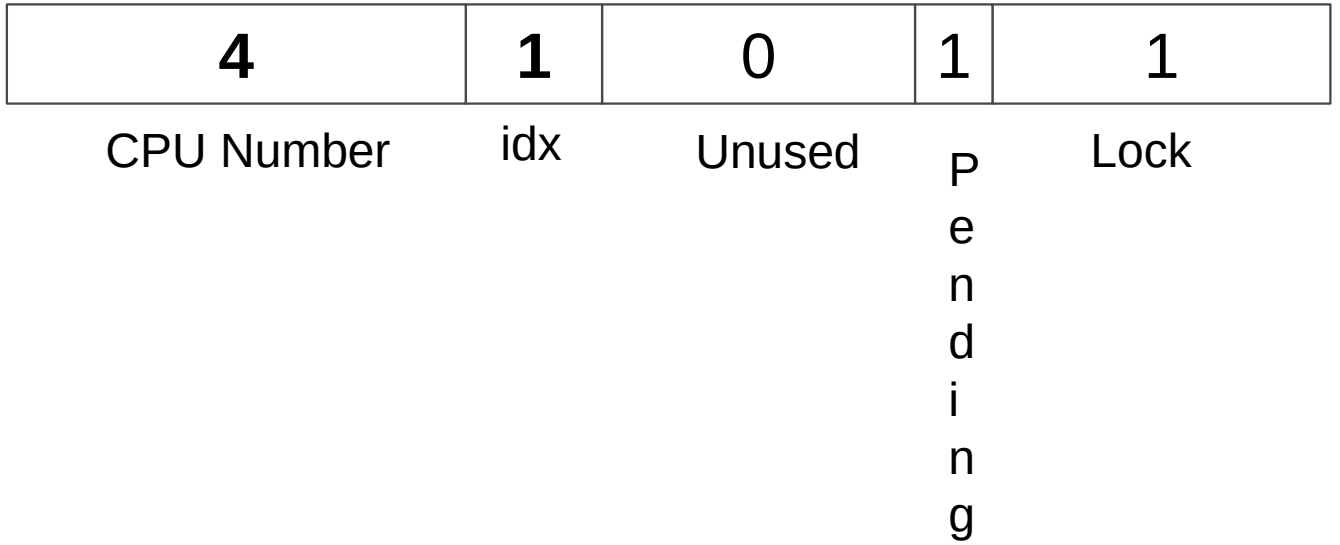
CPU 3 then generates the tail encoding as (CPU ID + 1, idx) = (4, 1).



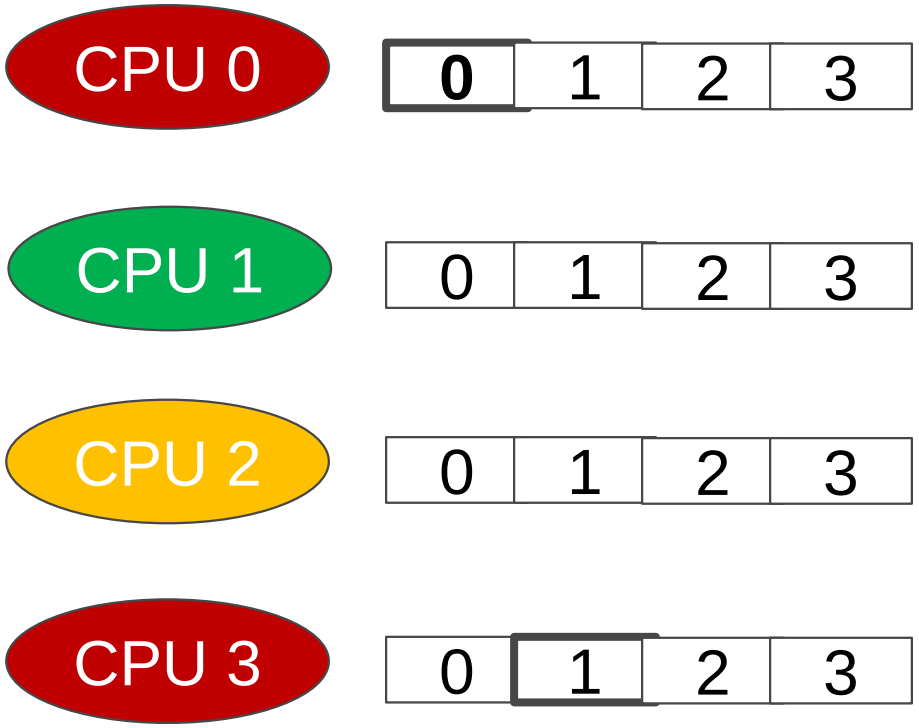
Queued Spin Locks



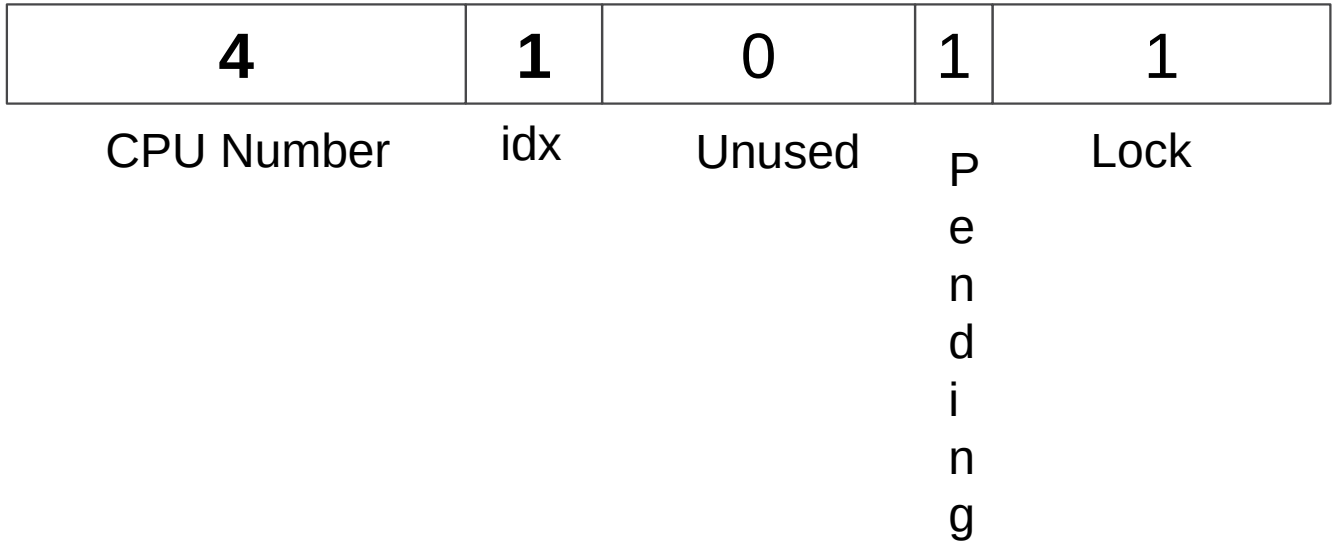
CPU 3 atomically exchanges the tail of the qspinlock (1, 0) with the new tail encoding (4, 1)



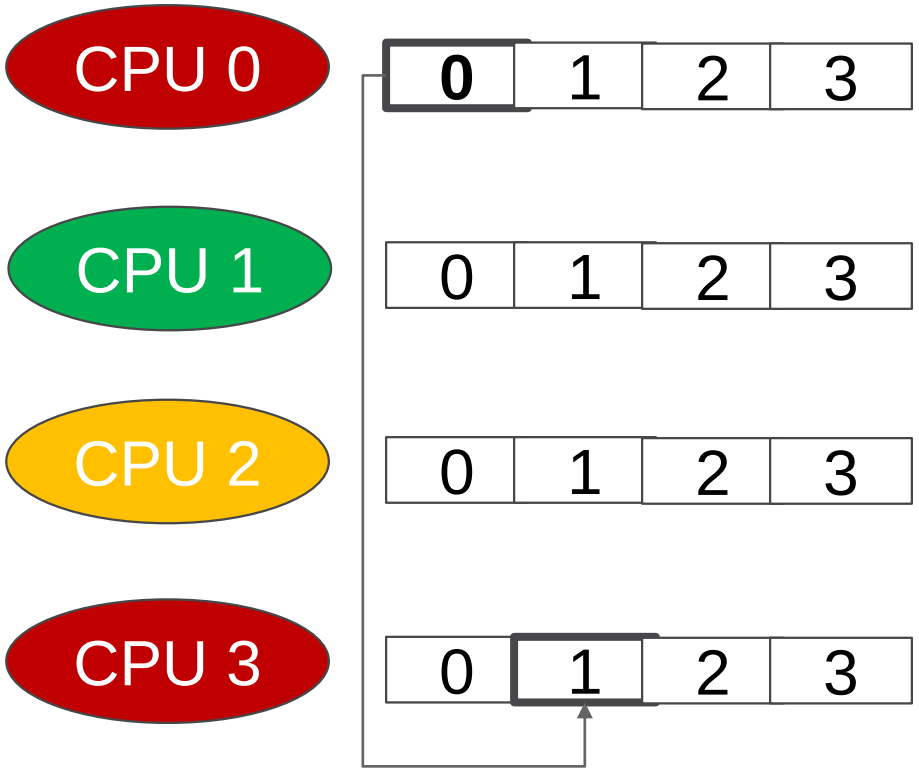
Queued Spin Locks



From the old tail value (1, 0) CPU 3 decodes that the previous node is CPU 0 idx 0.



Queued Spin Locks

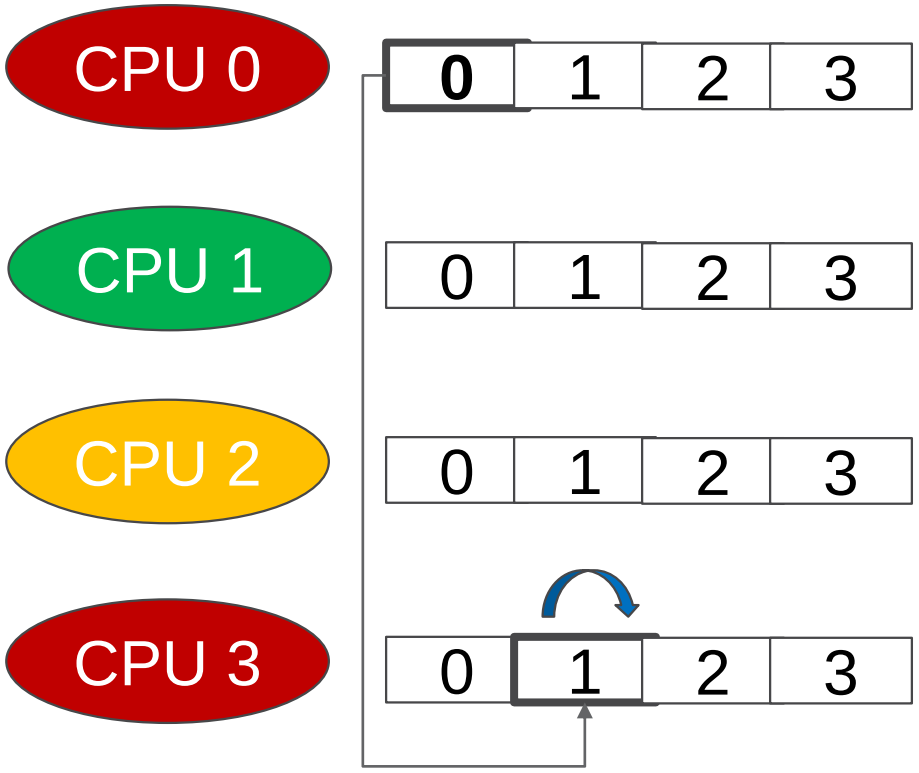


From the old tail value (1, 0) CPU 3 decodes that the previous node is CPU 0 idx 0.

So, it updates
CPU0.qnode[0].mcs_spinlock.next =
&CPU3.qnode[1].mcs_spinlock

4	**1**	0	1	1
CPU Number	idx	Unused	Pending	Lock

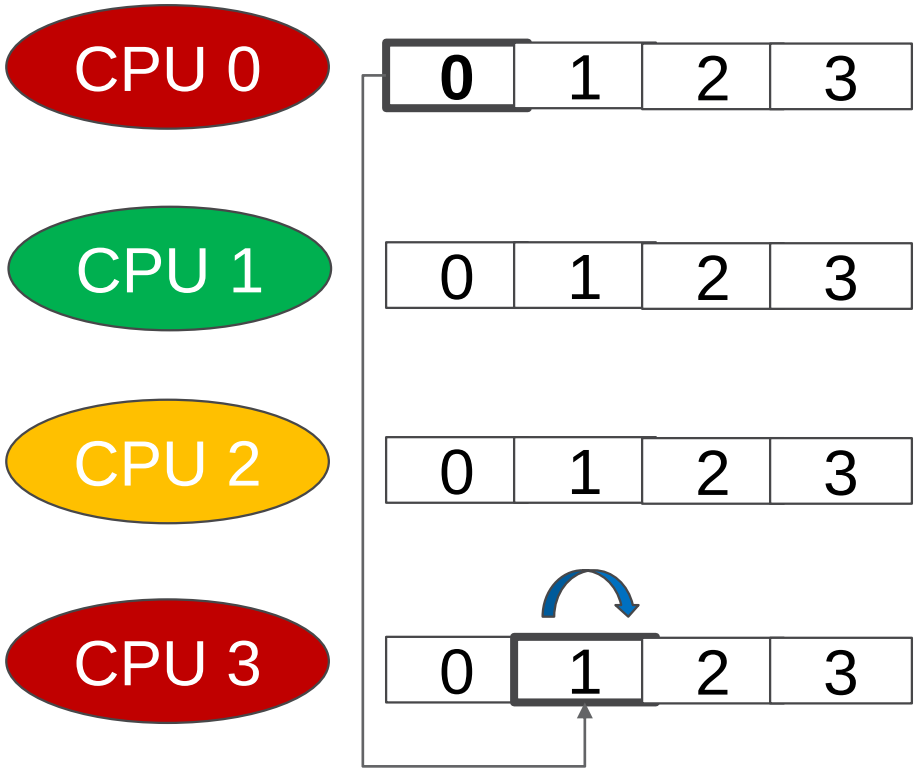
Queued Spin Locks



CPU 3 then spins until CPU3.qnode[1].mcs_spinlock.locked becomes non-zero.

4	**1**	0	1	1
CPU Number	idx	Unused	Pending	Lock

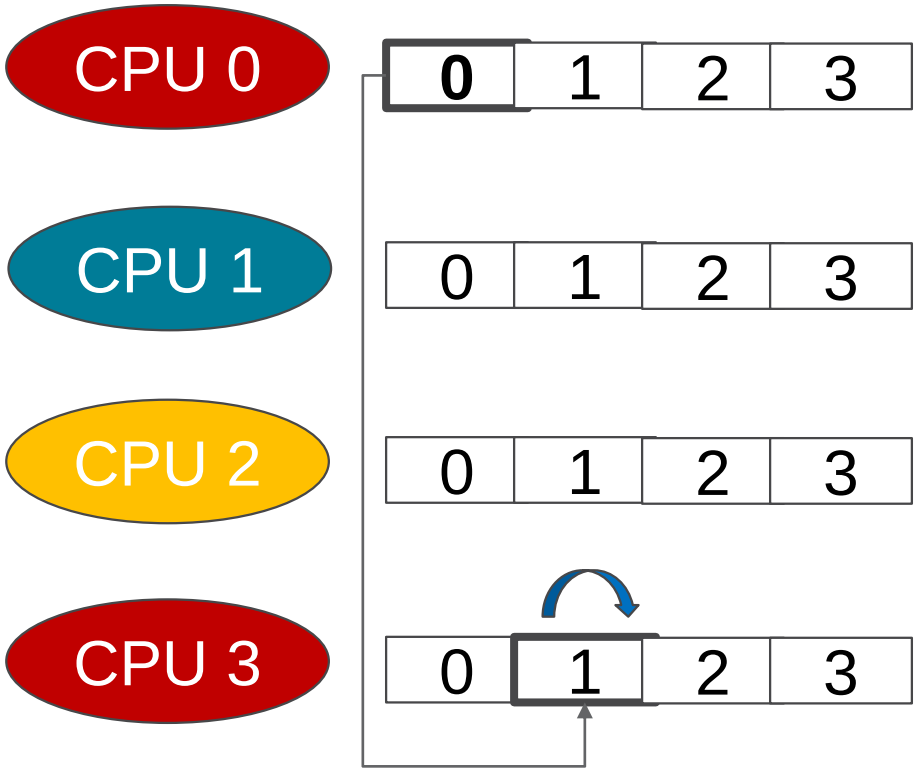
Queued Spin Locks



At this stage:
CPU 1 has the lock.
CPU 2 is spinning on Lock byte to become 0
CPU 0 is spinning on (Pending, Lock) to become (0, 0)
CPU 3 is spinning on CPU3.qn[0].mcs_sl.locked to be 1

4	**1**	0	1	1
CPU Number	idx	Unused	Pending	Lock

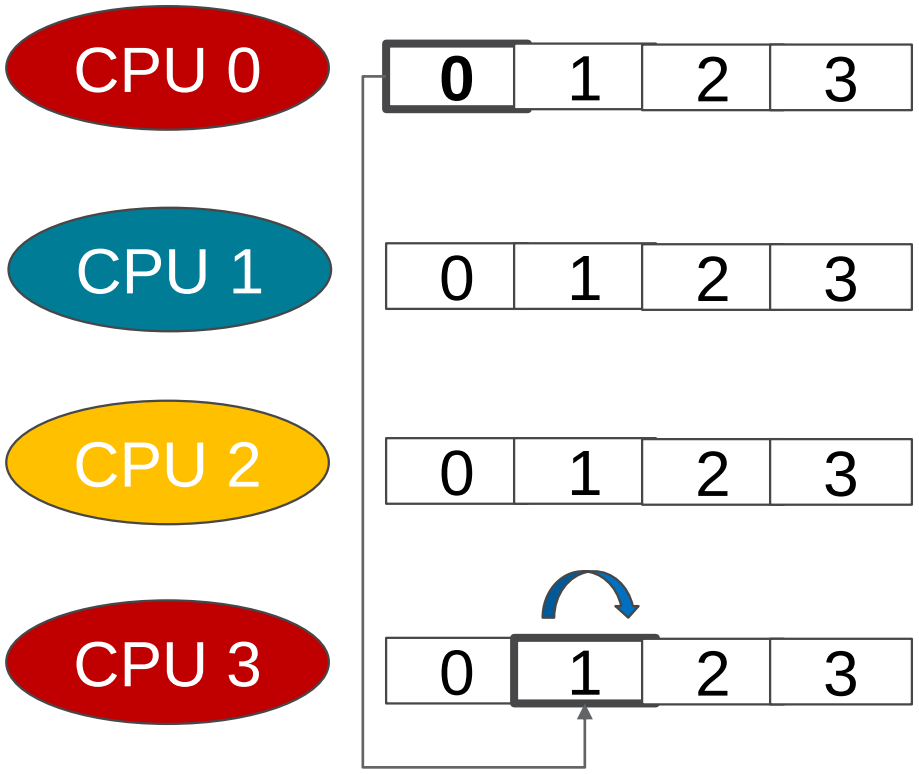
Queued Spin Locks



Now CPU 1 is done. It releases the lock by writing 0 to the Lock byte.

4	**1**	0	1	**0**
CPU Number	idx	Unused	Pending	Lock

Queued Spin Locks

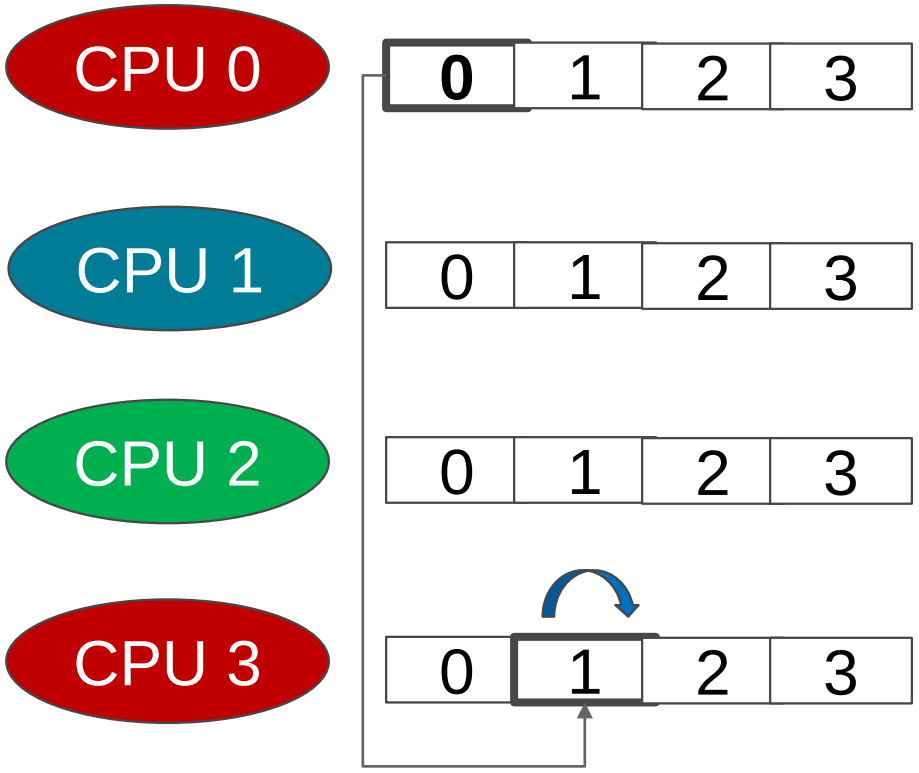


CPU 0 is spinning on (Pending, Lock) to become (0, 0)
CPU 3 is spinning on CPU3.qn[0].mcs_sl.locked to be 1
Neither is true.

CPU2 is waiting for Lock to be 0, which is now true.

4	**1**	0	1	**0**
CPU Number	idx	Unused	Pending	Lock

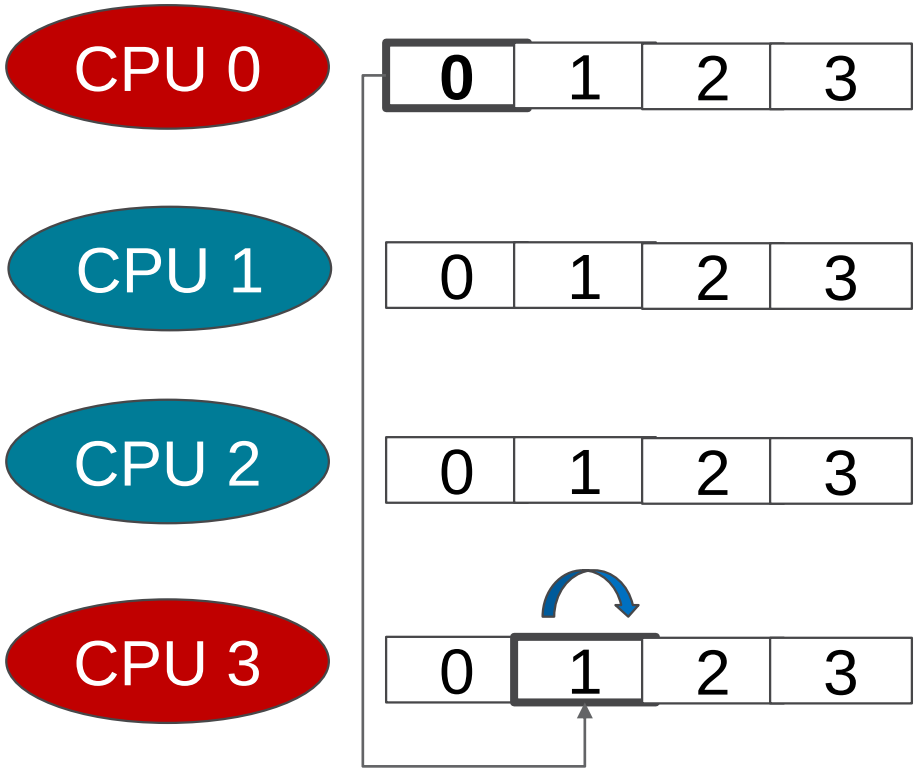
Queued Spin Locks



CPU2 atomically sets (pending, lock) to (0, 1), thus acquiring the lock.

4	**1**	0	**0**	**1**
CPU Number	idx	Unused	Pending	Lock

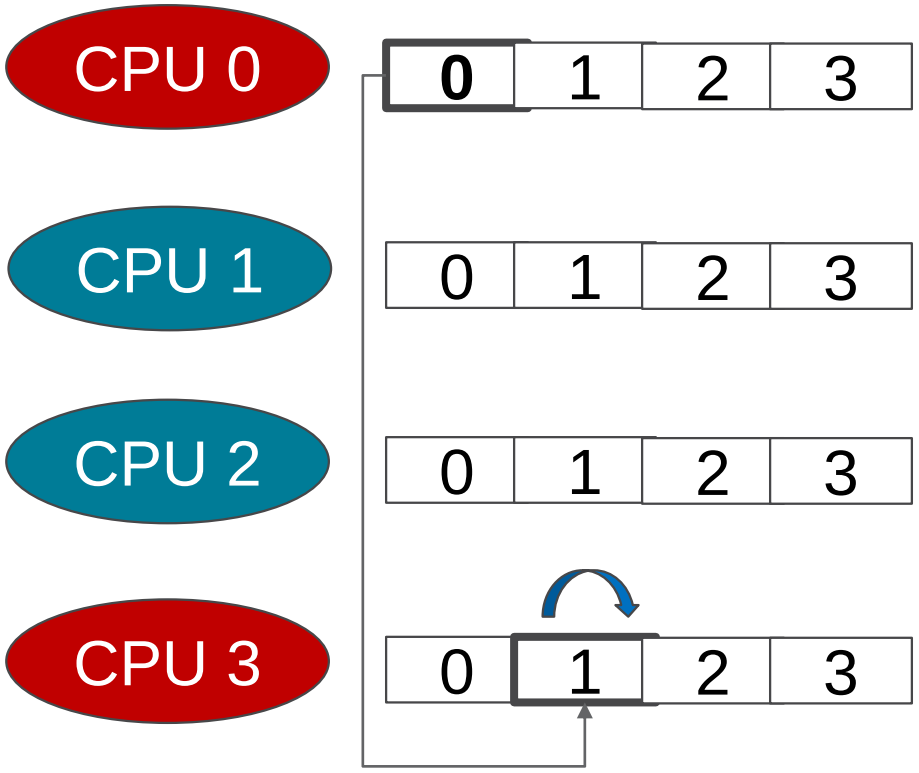
Queued Spin Locks



Once CPU2 is done, it will clear the Lock byte, thus releasing the lock.

4	1	0	0	**0**
CPU Number	idx	Unused	Pending	Lock

Queued Spin Locks

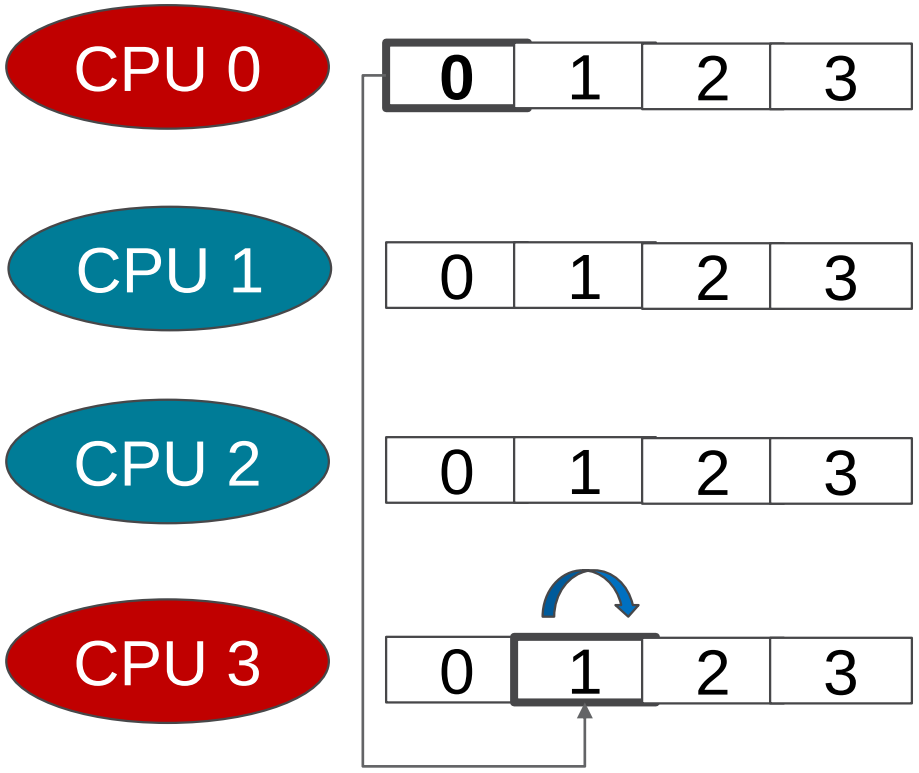


At this stage,
CPU 3 is waiting for CPU3.qn[1].mcs_sl.locked to be 1.

Which is not true.
CPU 0 is waiting for (pending, lock) to be (0, 0) which is true

4	1	0	0	**0**
CPU Number	idx	Unused	Pending	Lock

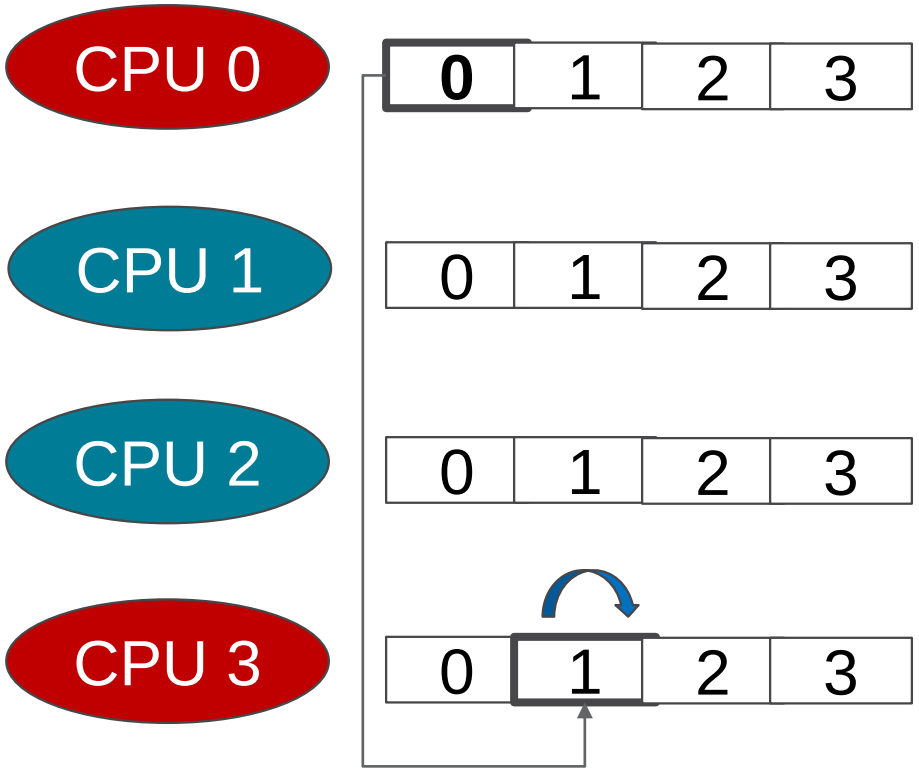
Queued Spin Locks



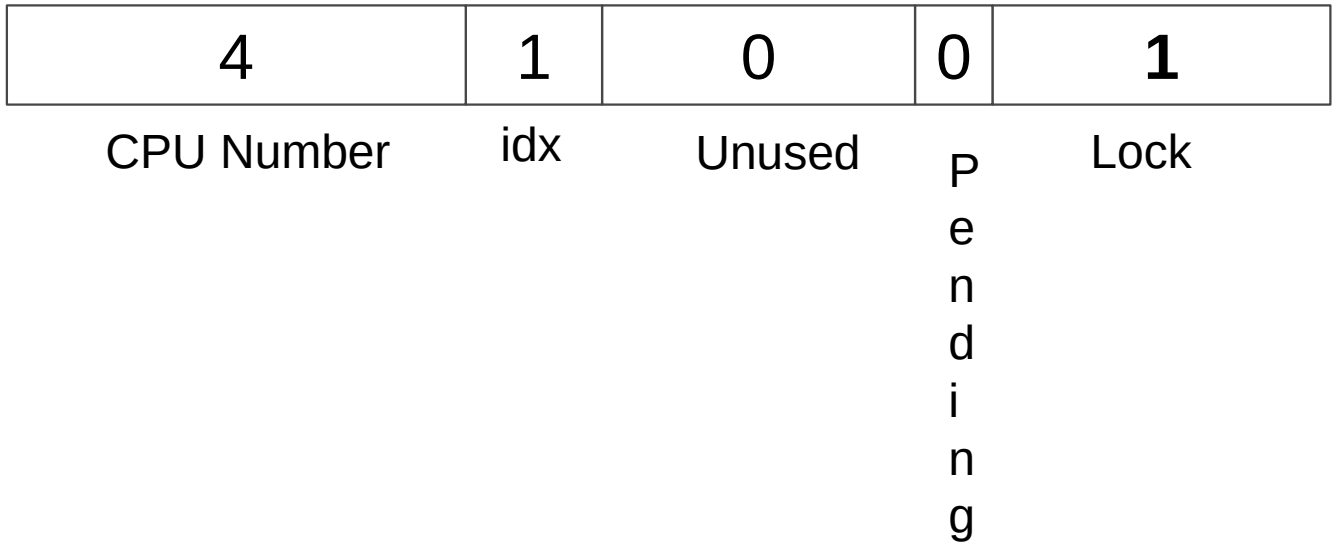
CPU 0 checks the qspinlock tail = (4, 1). Which is different from its encoding (1, 0).
So, there are waiters in the list.

4	1	0	0	**0**
CPU Number	idx	Unused	Pending	Lock

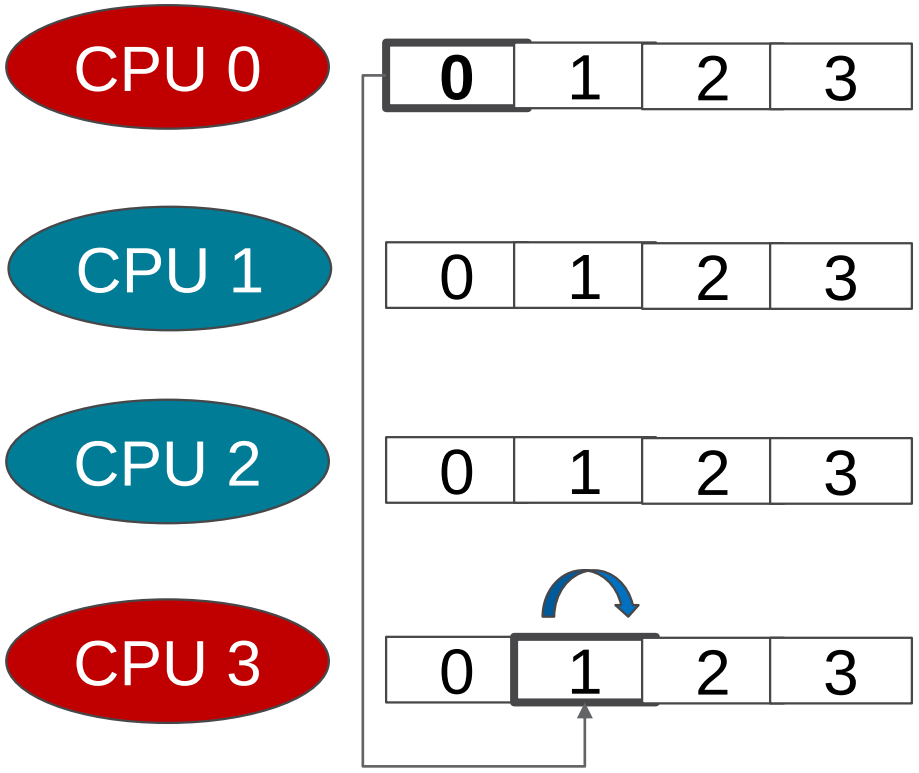
Queued Spin Locks



CPU 0 atomically sets the “Lock” byte to 1.

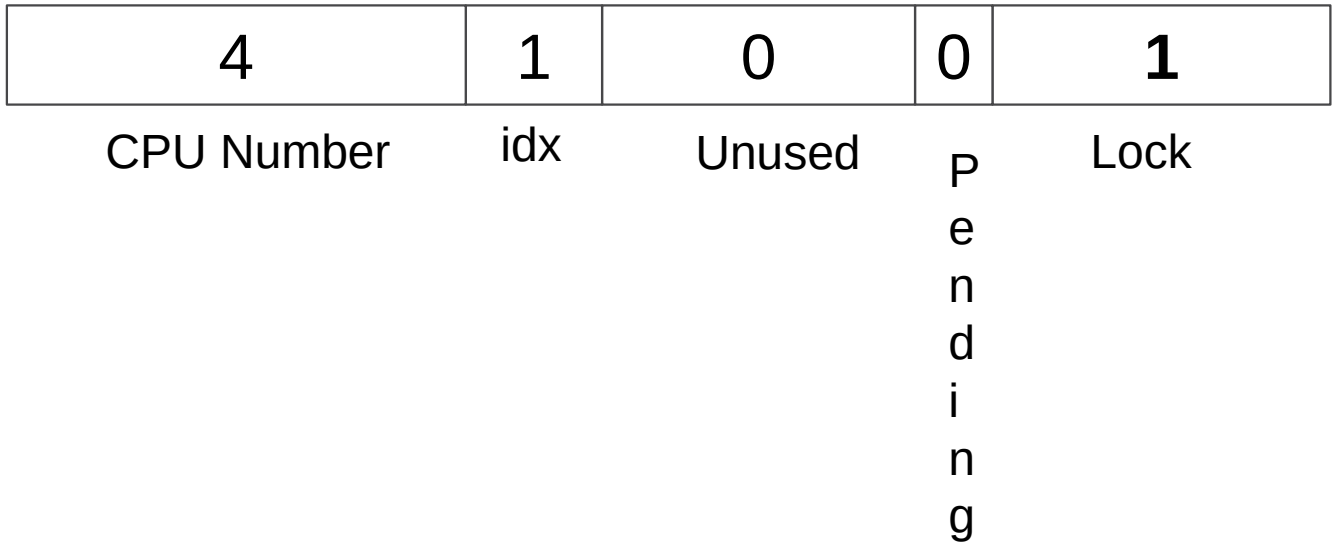


Queued Spin Locks

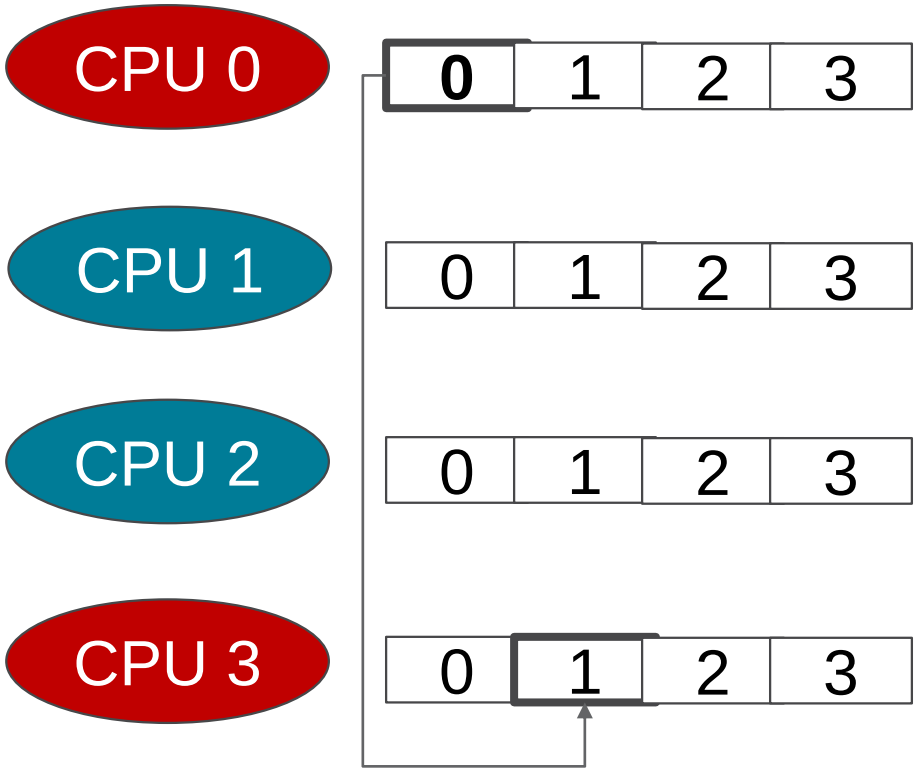


CPU 0 atomically sets the “Lock” byte to 1.

CPU 0 then sets CPU0.qn[0].mcs_sl.next->locked to 1.



Queued Spin Locks



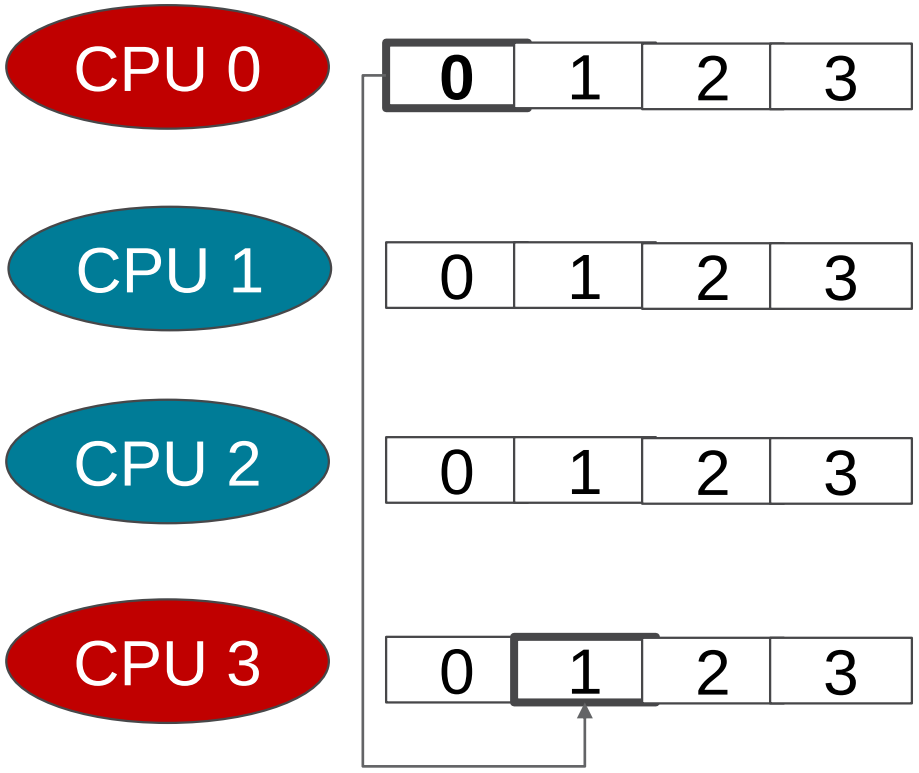
CPU 0 atomically sets the “Lock” byte to 1.

CPU 0 then sets CPU0.qn[0].mcs_sl.next->locked to 1.

CPU 3 stops spinning on CPU3.qn[1].mcs_sl.locked

4	1	0	0	**1**
CPU Number	idx	Unused	Pending	Lock

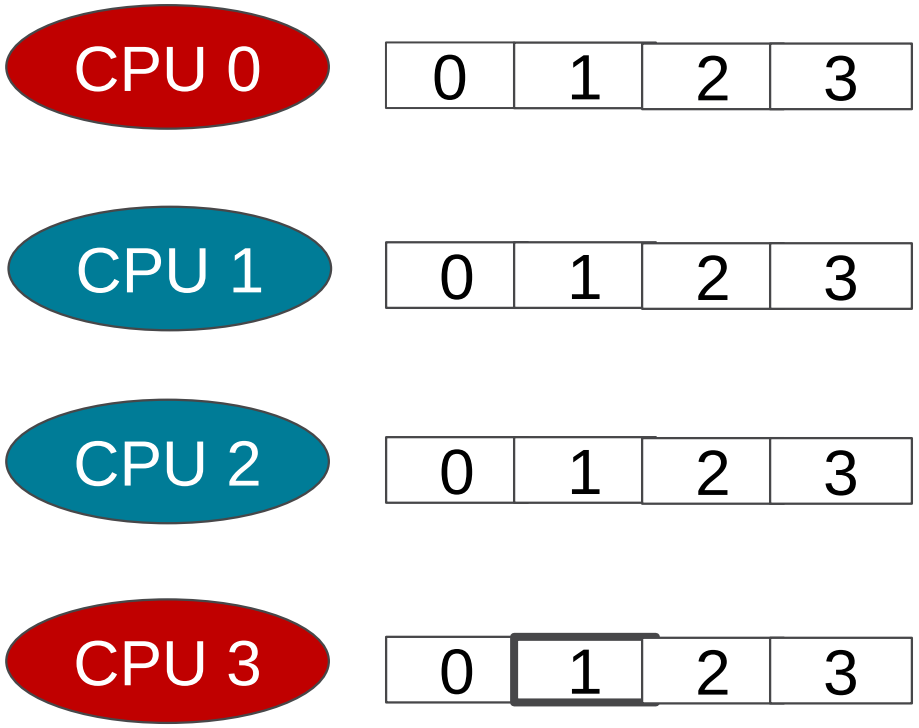
Queued Spin Locks



CPU 3 now spins until qspinlock (pending, lock) is (0, 0).

4	1	0	0	**1**
CPU Number	idx	Unused	Pending	Lock

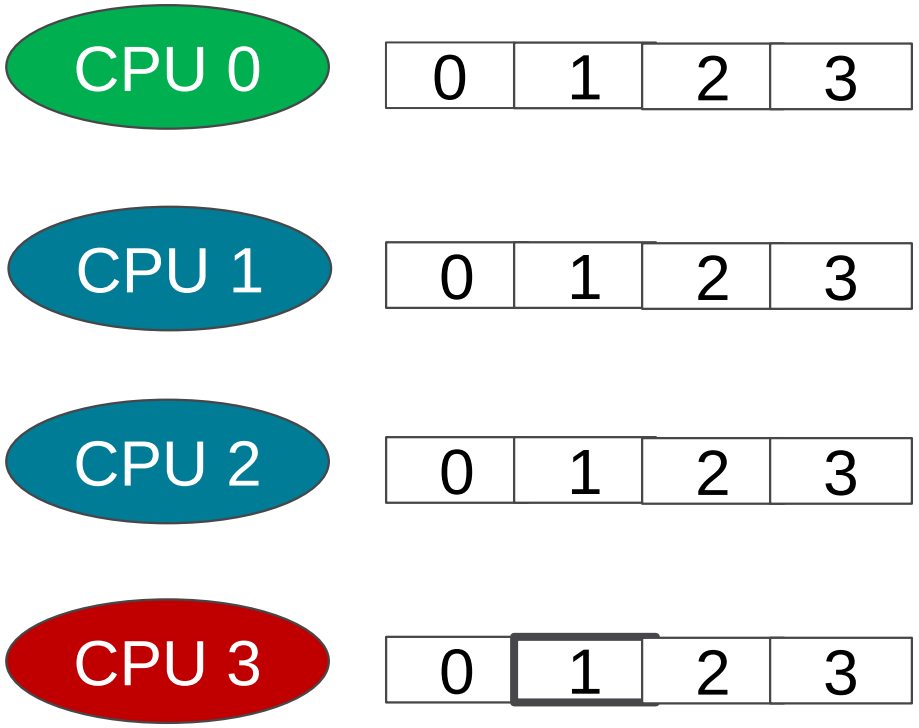
Queued Spin Locks



CPU 0 decrements CPU0.qn[0].mcs_sl.count and releases the qnode.

4	1	0	0	**1**
CPU Number	idx	Unused	Pending	Lock

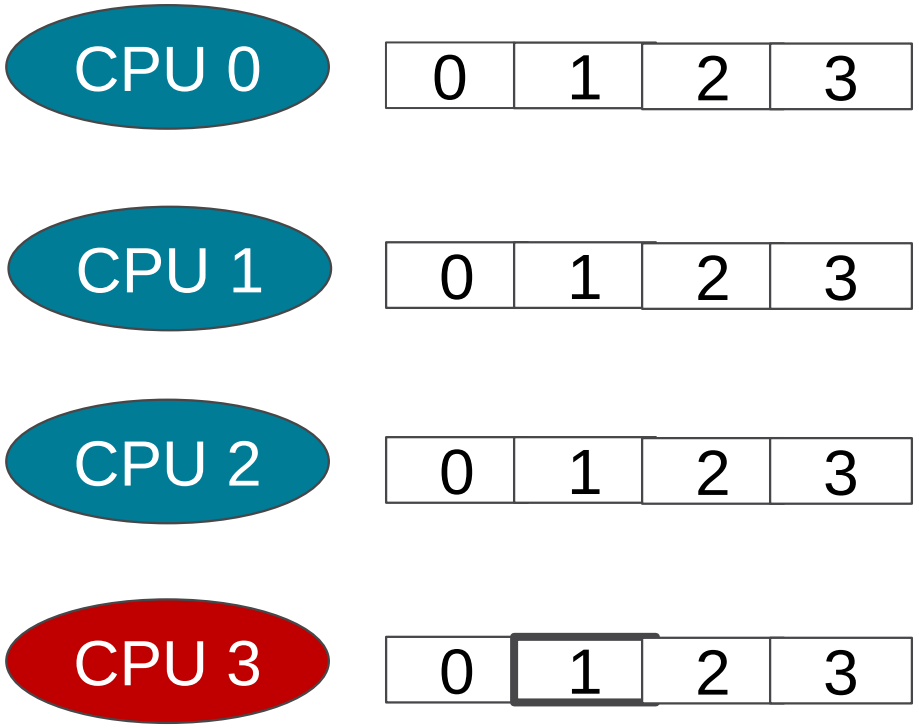
Queued Spin Locks



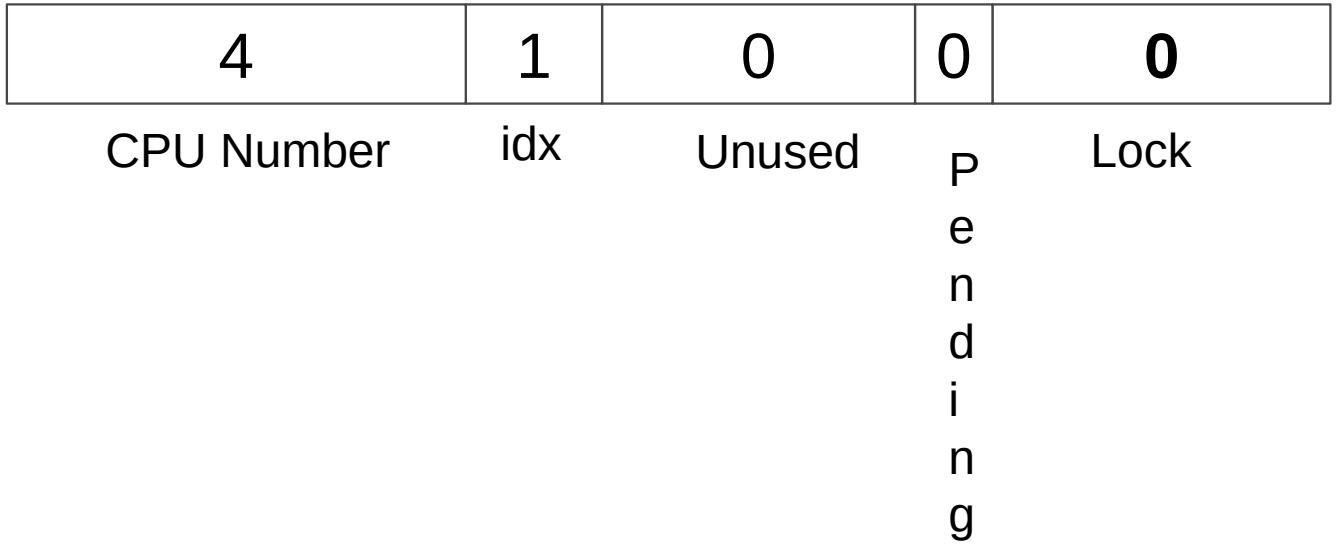
CPU 0 is now the lock owner.

4	1	0	0	**1**
CPU Number	idx	Unused	Pending	Lock

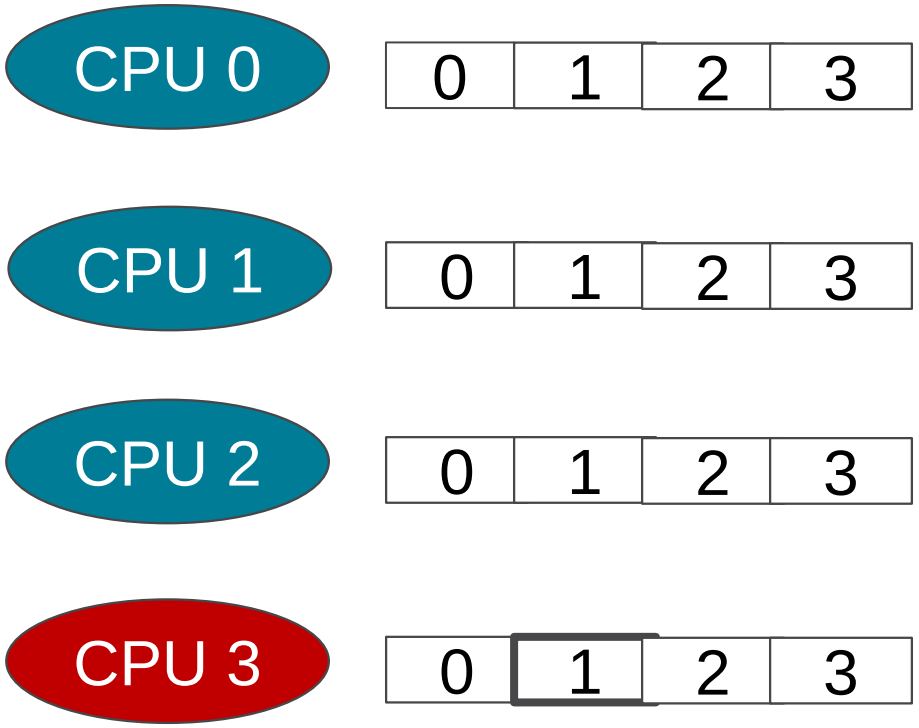
Queued Spin Locks



Once CPU 0 is done, it will release the lock by setting the qspinlock Lock byte to 0.



Queued Spin Locks

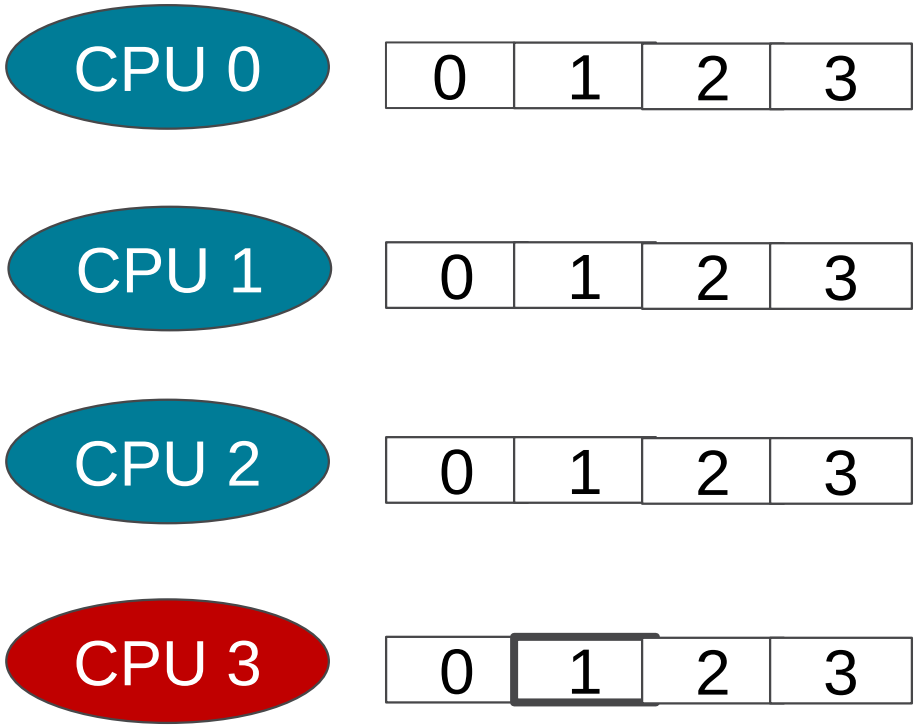


Since (pending, lock) is now (0, 0) CPU3 stops spinning.

CPU 3 checks qspinlock.tail = (4, 1) which matches its tail encoding.

4	1	0	0	**0**
CPU Number	idx	Unused	Pending	Lock

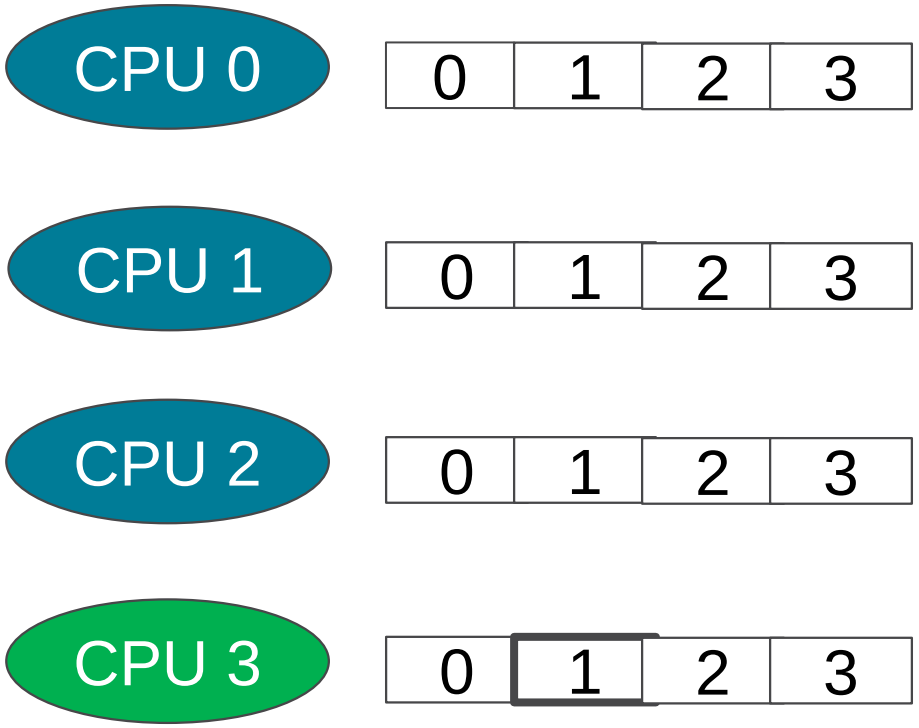
Queued Spin Locks



Since there is no other waiter at this point, CPU 3 atomically tries to change qspinlock from (4, 1, 0, 0, 0) to (0, 0, 0, 0, 1).

4	1	0	0	**0**
CPU Number	idx	Unused	Pending	Lock

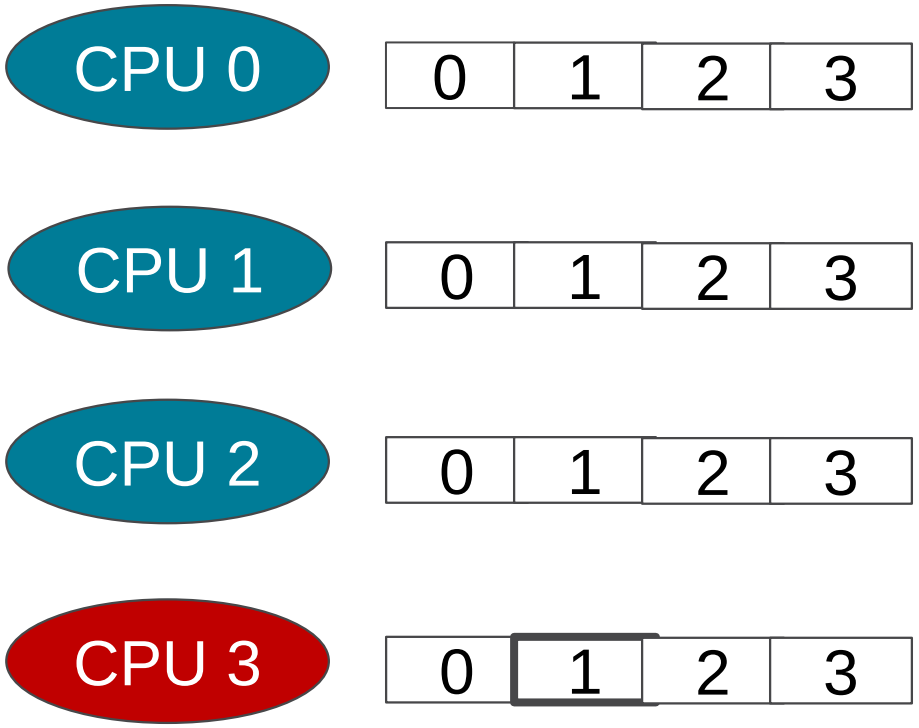
Queued Spin Locks



If CPU 3 succeeds, it gets the lock.

0	**0**	**0**	0	**1**
CPU Number	idx	Unused	Pending	Lock

Queued Spin Locks

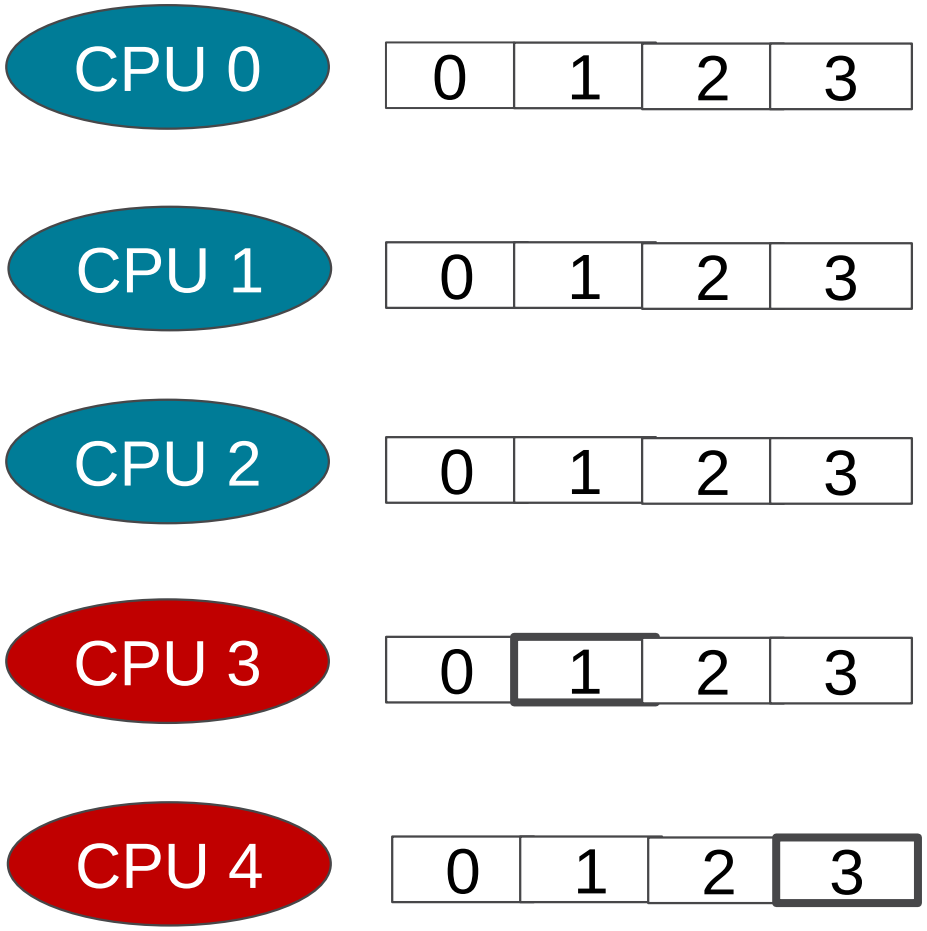


Since there is no other waiter at this point, CPU 3 atomically tries to change qspinlock from (4, 1, 0, 0, 0) to (0, 0, 0, 0, 1).

Suppose it fails!

4	1	0	0	**0**
CPU Number	idx	Unused	Pending	Lock

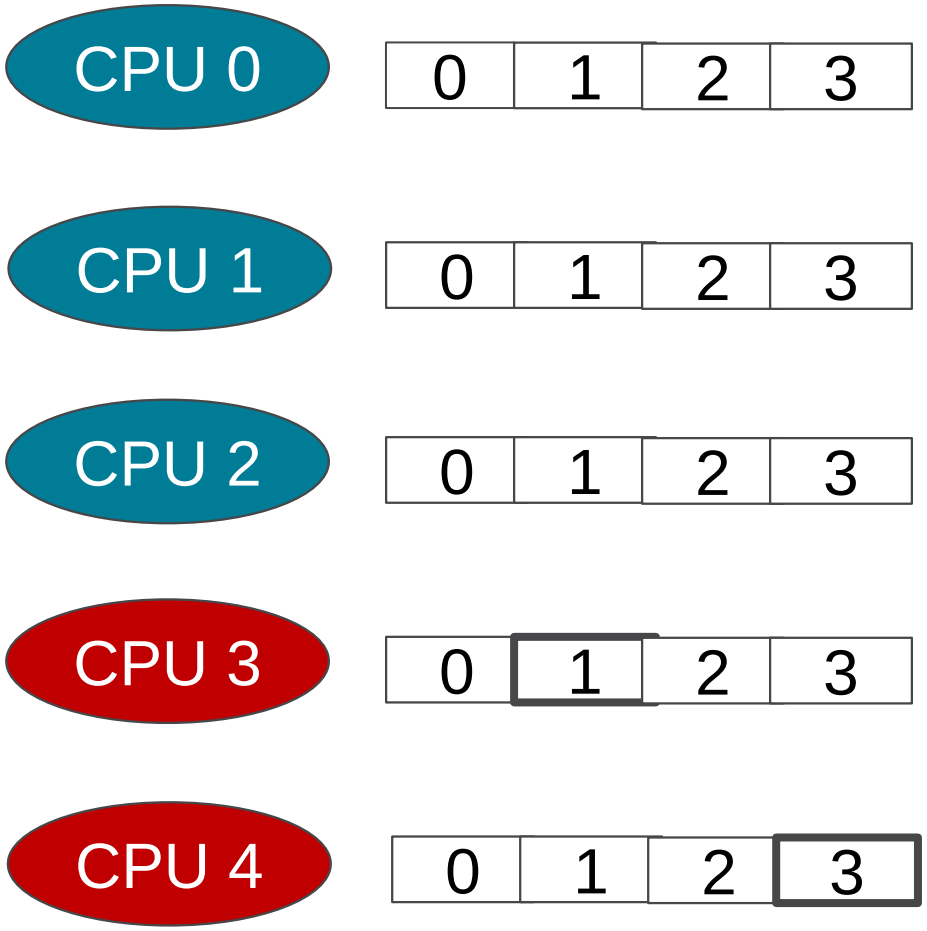
Queued Spin Locks



The failure can only happen if a new waiter has updated its identity in the tail. Say CPU 4 with index 3.

5	3	0	0	**0**
CPU Number	idx	Unused	Pending	Lock

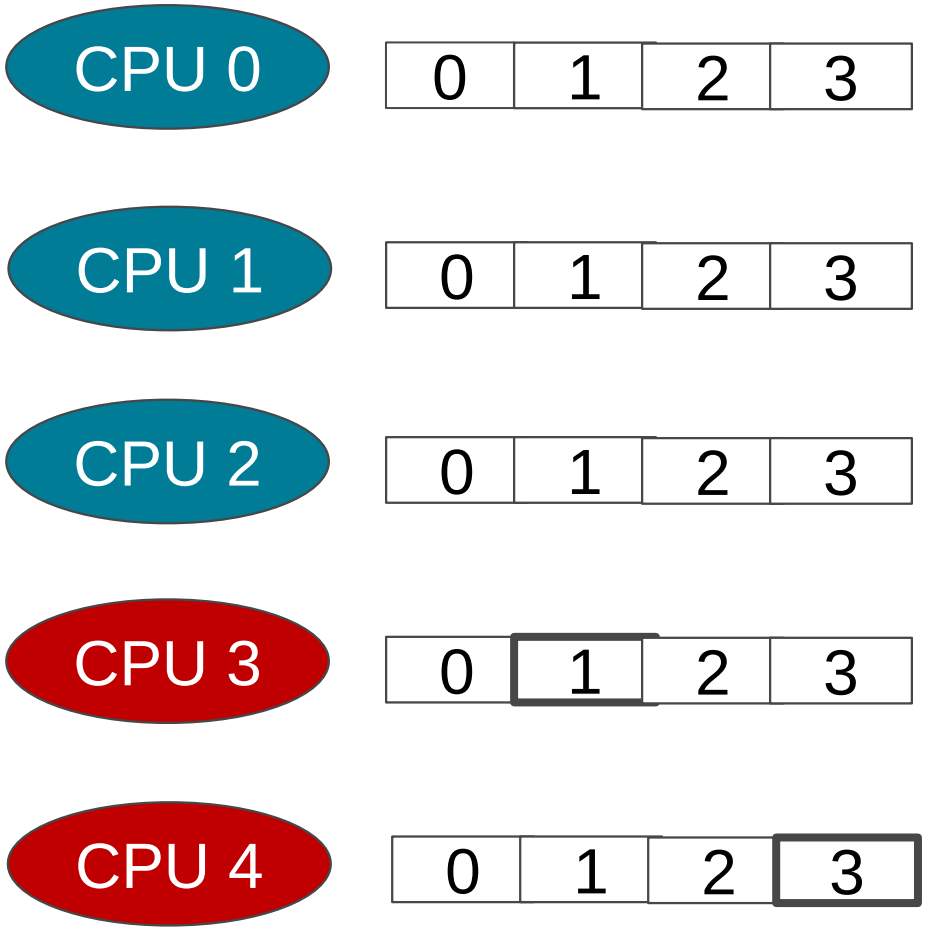
Queued Spin Locks



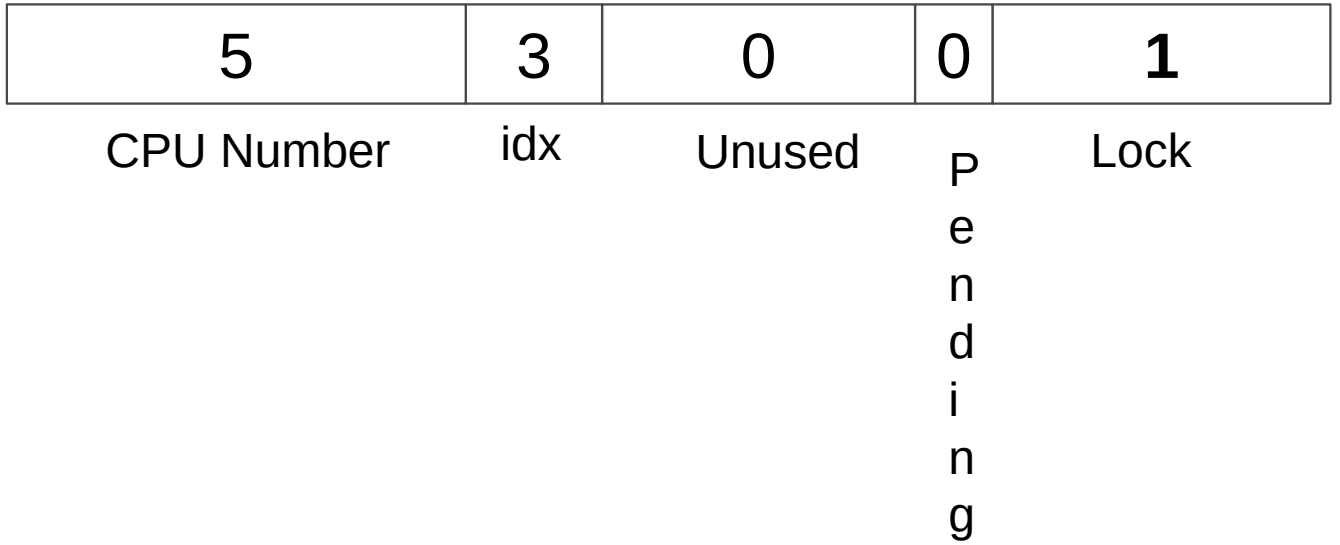
Then CPU 3 will atomically set the Lock byte.

5	3	0	0	1
CPU Number	idx	Unused	Pending	Lock

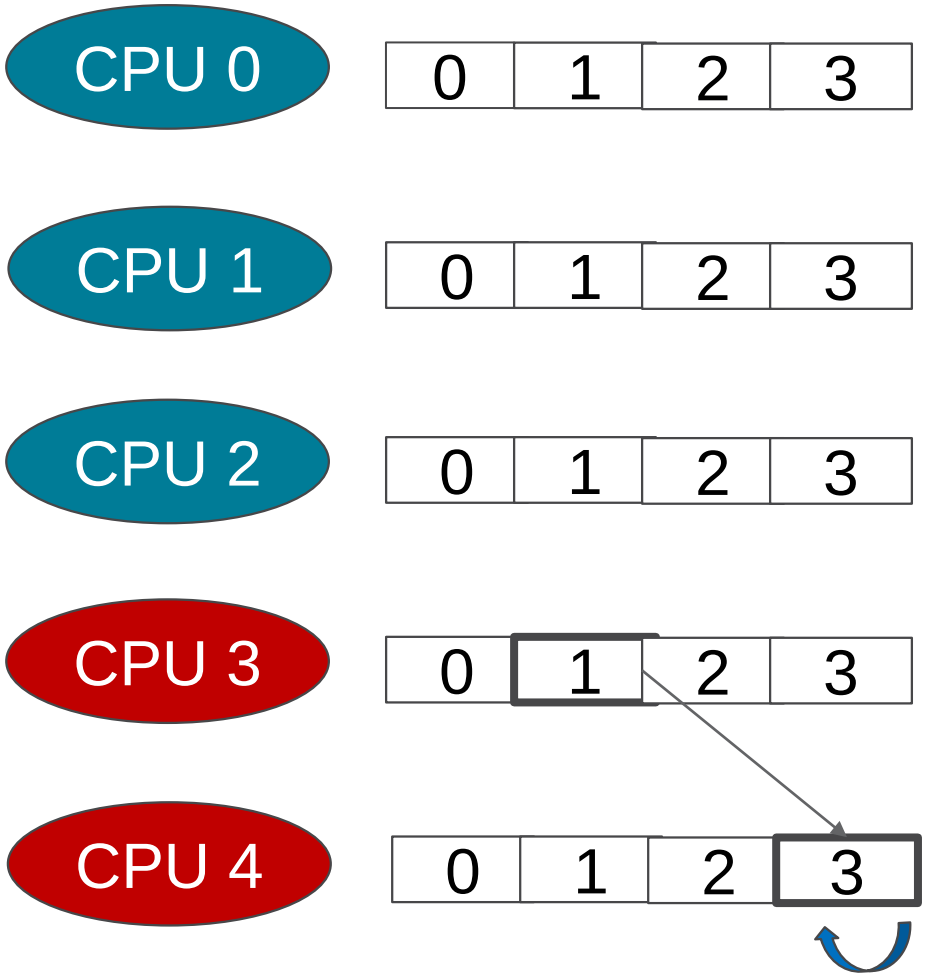
Queued Spin Locks



CPU 3 will then wait for CPU3.qn[1].mcs_sl.next to be non-NULL.



Queued Spin Locks

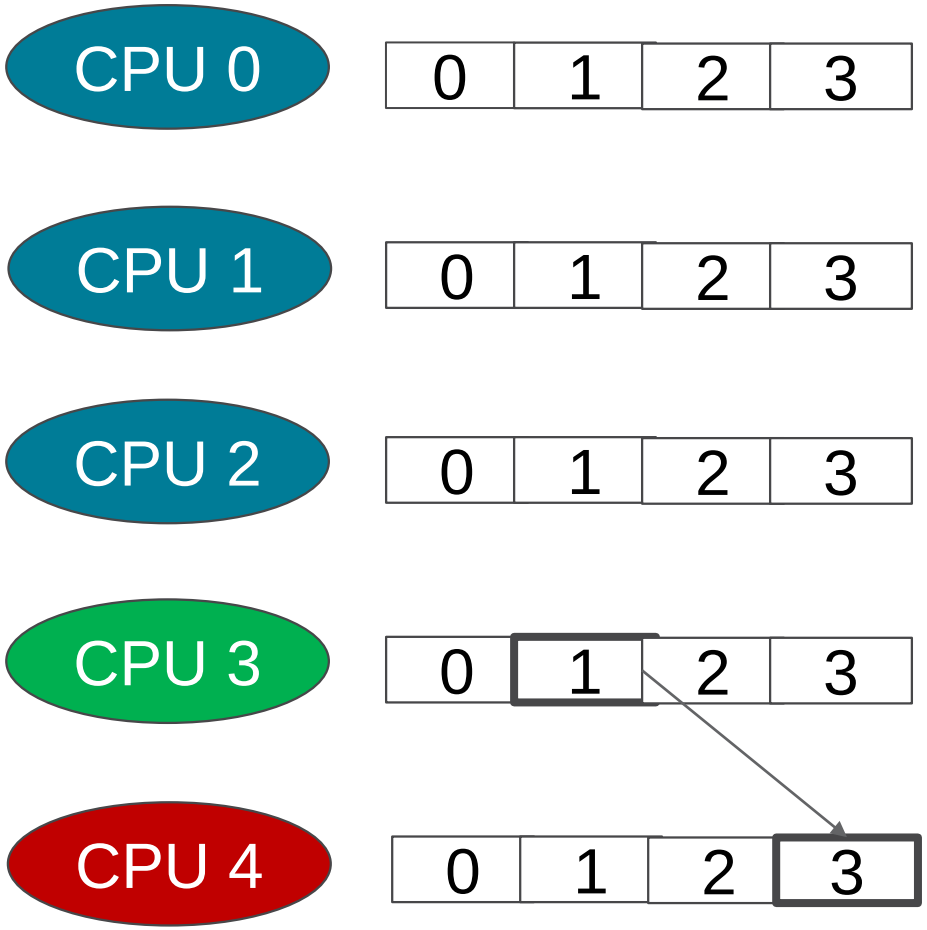


When CPU3.qn[1].mcs_sl.next is non-NULL, it will point to the next waiter.

Since CPU 3 is going to take the lock, it will update the next waiter's mcs_spinlock.locked variable to 1 to stop its spinning.

5	3	0	0	**1**
CPU Number	idx	Unused	Pending	Lock

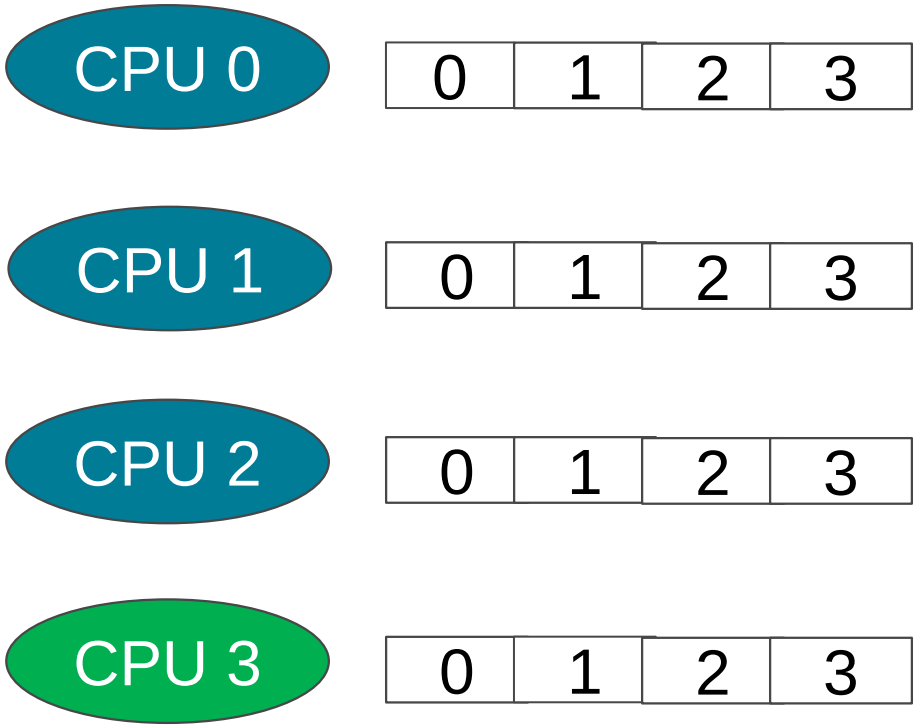
Queued Spin Locks



CPU3 now holds the lock.

5	3	0	0	1
CPU Number	idx	Unused	Pending	Lock

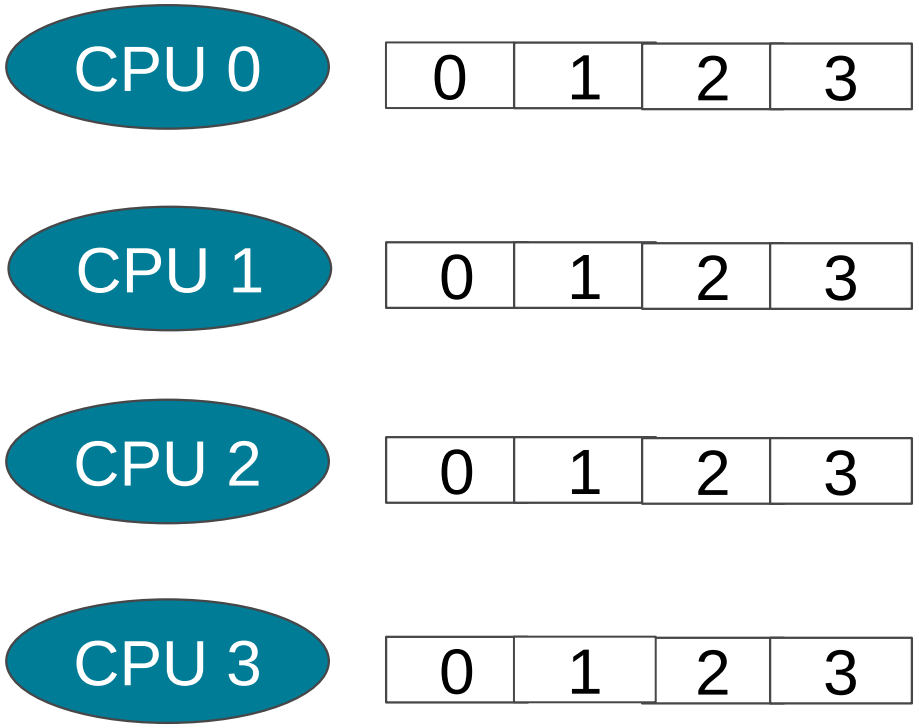
Queued Spin Locks



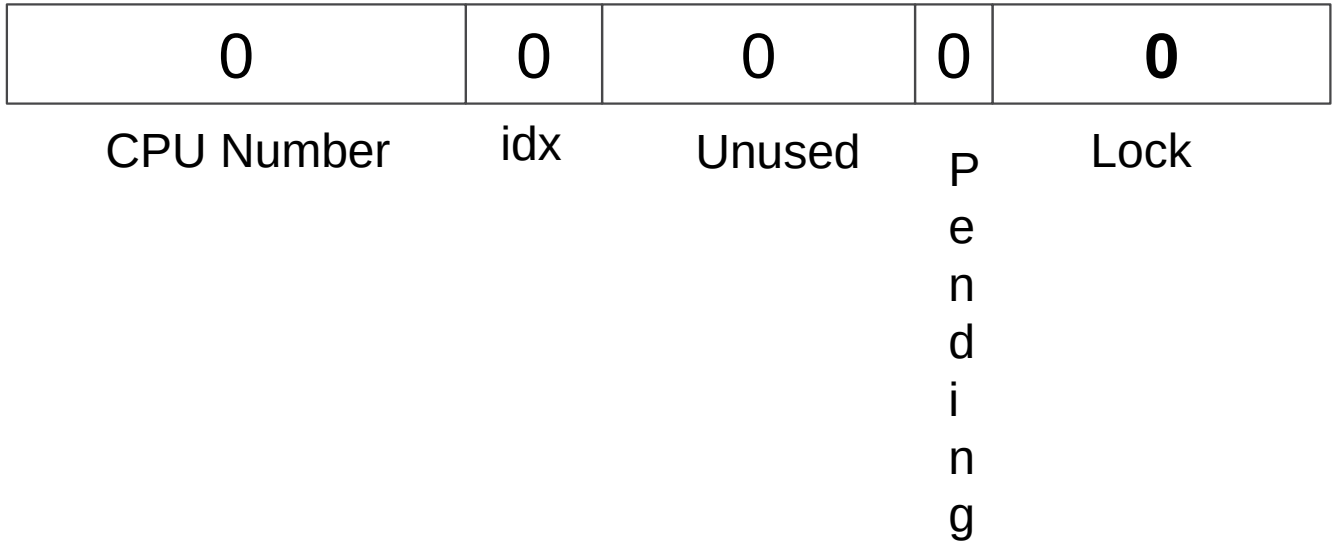
Let us go back to the case where CPU 3 was the last waiter and it succeeded in setting qspinlock to (0, 0, 0, 0, 1) and become the lock owner.

0	**0**	**0**	0	**1**
CPU Number	idx	Unused	Pending	Lock

Queued Spin Locks



When it is done, it will update the Lock byte to 0 thus releasing the lock.



Questions?

COPYRIGHT AND DISCLAIMER

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate releases, for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

THIS INFORMATION IS PROVIDED 'AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.


AMD