

# NUMA-Aware Locking Routines -Design and Implementation in Linux

# NUMA-Aware Locks Motivation

- Simple Test-and-Set spinlocks perform poorly under high contention, when the memory latency between intra-node and inter-node accesses, vary a lot
- Simple spinlocks can also suffer from unfairness and starvation especially in NUMA systems
  - CPUs on the same node have an unfair advantage over the other CPUs in acquiring the lock

# NUMA-aware Locking Routines in Linux

- A simple locking routine that overcomes this problem is J-Locks
  - have a global lock bitmask
  - CPU that wants the lock sets the bit in the global bitmask corresponding to its CPU number and if no other CPU had bid for it acquires it, else spins on its wakeup pointer
  - when a CPU releases a lock, searches the lock's global bitmask circularly from the releasing CPU's bit. If some CPU has its bit set, then it sets the other CPU's pointer to NULL and clears its lock bit

# NUMA-aware Locking Routines in Linux

- When a CPU is interrupted, it checks its state to see if it was spinning for a lock. If so, releases its bid for the lock and renews its bid when it returns from interrupt

# NUMA-aware Locking Routines Implementation in Linux

- Implementation of J-Locks in Linux-  
Data Structures required:
  - spinlock\_t 's lock acts as the lock's global bitmask
  - struct numa\_spins {
    - spinlock\_t \* wakeup; // Pointer in which CPU spins for the lock
    - spinlock\_t \* irq[3]; // Pointers to save the spin status when a CPU is interrupted
  - }; // An array of numa\_spins is maintained for every CPU

# NUMA-aware Locking Routines Implementation in Linux

## ■ Important Routines :

- `numa_lock (spinlock_t *)`; // to obtain a lock
- `numa_unlock (spinlock_t *)`; //to release a lock
- `numa_wakeup (spinlock_t *, mask)`; // called by `numa_unlock` to find the next candidate to get the lock
- `numa_unspin( )`; // called in the beginning of `do_IRQ( )`
- `numa_respin( )`; // called at the end of `do_IRQ( )`

# NUMA-aware Locking Routines Implementation in Linux

- `numa_lock( spinlock_t *)`
  - called for obtaining a lock, with interrupts enabled
  - sets its bit in lock's bitmask and checks the lock's bitmask to see if someone else holds it, if not acquires else sets its wakeup pointer to the lock address and spins on it with interrupts enabled

# NUMA-aware Locking Routines Implementation in Linux

- `numa_unlock(spinlock_t *)`
  - called for releasing a lock, with interrupts enabled
  - checks lock's bitmask to see if other CPUs want it, if so calls `numa_wakeup()` to find the next candidate
  - clears its bit in the lock's bitmask and returns
- `numa_wakeup(spinlock_t *, ulong)` - finds the next candidate wanting the lock by doing a circular search on lock's bitmask

# NUMA-aware Locking Routines Implementation in Linux

## ■ numa\_unspin

- called upon entry of the do\_IRQ routine
- checks to see if were interrupted in the middle of spinning
- if so saves state in irq pointers and clears its bit in the lock's bitmask

## ■ numa\_respin

- called upon exit of the do\_IRQ routine
- checks to see if lock was given up in numa\_unspin if so again bids for lock by setting its bit in the lock's bitmask and the wakeup pointer

# NUMA-aware Locking Routines Implementation in Linux

- In addition to these functions, the following locking interfaces are also provided
  - `numa_lock_irqdisable(spinlock_t * ); // disables interrupts and calls a variant of numa_lock which doesn't worry about interrupts`
  - `numa_lock_irqsave(spinlock_t *, unsigned long flags ); // saves flags and gets the lock by calling a variant of numa_lock`
  - `numa_unlock_irqenable(spinlock_t * ); // unlocks by calling a variant of numa_unlock and then enables interrupts`
  - `numa_unlock_irqrestore(spinlock_t *, unsigned long flags ); // unlocks by calling a variant of numa_unlock and then restores based on flags`