

Practical Concerns for Scalable Synchronization

Josh Triplett

May 10, 2006

The basic problem

- Operating systems need concurrency
- Operating systems need shared data structures

The basic problem

- Operating systems need concurrency
- Operating systems need shared data structures

Mutual exclusion?

- Readers and writers acquire a lock
- Doesn't scale
- High contention
- Priority inversion
- Deadlock

Mutual exclusion?

- Readers and writers acquire a lock
- Doesn't scale
- High contention
- Priority inversion
- Deadlock

Mutual exclusion?

- Readers and writers acquire a lock
- Doesn't scale
- High contention
- Priority inversion
- Deadlock

Mutual exclusion?

- Readers and writers acquire a lock
- Doesn't scale
- High contention
- Priority inversion
- Deadlock

Mutual exclusion?

- Readers and writers acquire a lock
- Doesn't scale
- High contention
- Priority inversion
- **Deadlock**

Speed up contended case

- Better spin locks
- Queuing locks
- How much does the high-contention case matter?

Speed up contended case

- Better spin locks
- **Queuing locks**
- How much does the high-contention case matter?

Speed up contended case

- Better spin locks
- Queuing locks
- How much does the high-contention case matter?

Reduce contention

- Contention-reducing data structures (two-lock queue)
- Reader-writer locking
- Localizing data to avoid sharing or false sharing

Reduce contention

- Contention-reducing data structures (two-lock queue)
- Reader-writer locking
- Localizing data to avoid sharing or false sharing

Reduce contention

- Contention-reducing data structures (two-lock queue)
- Reader-writer locking
- Localizing data to avoid sharing or false sharing

Contention less relevant

- Atomic instructions expensive
 - Memory barriers expensive
 - 3 orders of magnitude worse than regular instruction
 - For locking, lock maintenance time dominates
 - For non-blocking synchronization, CAS or LL/SC time dominates

Contention less relevant

- Atomic instructions expensive
- **Memory barriers expensive**
 - 3 orders of magnitude worse than regular instruction
 - For locking, lock maintenance time dominates
 - For non-blocking synchronization, CAS or LL/SC time dominates

Contention less relevant

- Atomic instructions expensive
- Memory barriers expensive
- 3 orders of magnitude worse than regular instruction
- For locking, lock maintenance time dominates
- For non-blocking synchronization, CAS or LL/SC time dominates

Contention less relevant

- Atomic instructions expensive
- Memory barriers expensive
- 3 orders of magnitude worse than regular instruction
- For locking, lock maintenance time dominates
- For non-blocking synchronization, CAS or LL/SC time dominates

Contention less relevant

- Atomic instructions expensive
- Memory barriers expensive
- 3 orders of magnitude worse than regular instruction
- For locking, lock maintenance time dominates
- For non-blocking synchronization, CAS or LL/SC time dominates

Avoiding expensive instructions

- **Writer needs locking or non-blocking synchronization**
- What about the reader?
- Need to ensure that the reader won't crash
- Crashes caused by following a bad pointer
- Assignment to an aligned pointer happens atomically
- Insert and remove items atomically
- Need some memory barriers: prevent insertion before initialization

Avoiding expensive instructions

- Writer needs locking or non-blocking synchronization
- **What about the reader?**
 - Need to ensure that the reader won't crash
 - Crashes caused by following a bad pointer
 - Assignment to an aligned pointer happens atomically
 - Insert and remove items atomically
 - Need some memory barriers: prevent insertion before initialization

Avoiding expensive instructions

- Writer needs locking or non-blocking synchronization
- What about the reader?
- **Need to ensure that the reader won't crash**
 - Crashes caused by following a bad pointer
 - Assignment to an aligned pointer happens atomically
 - Insert and remove items atomically
 - Need some memory barriers: prevent insertion before initialization

Avoiding expensive instructions

- Writer needs locking or non-blocking synchronization
- What about the reader?
- Need to ensure that the reader won't crash
- Crashes caused by following a bad pointer
 - Assignment to an aligned pointer happens atomically
 - Insert and remove items atomically
 - Need some memory barriers: prevent insertion before initialization

Avoiding expensive instructions

- Writer needs locking or non-blocking synchronization
- What about the reader?
- Need to ensure that the reader won't crash
- Crashes caused by following a bad pointer
- **Assignment to an aligned pointer happens atomically**
- Insert and remove items atomically
- Need some memory barriers: prevent insertion before initialization

Avoiding expensive instructions

- Writer needs locking or non-blocking synchronization
- What about the reader?
- Need to ensure that the reader won't crash
- Crashes caused by following a bad pointer
- Assignment to an aligned pointer happens atomically
- **Insert and remove items atomically**
- Need some memory barriers: prevent insertion before initialization

Avoiding expensive instructions

- Writer needs locking or non-blocking synchronization
- What about the reader?
- Need to ensure that the reader won't crash
- Crashes caused by following a bad pointer
- Assignment to an aligned pointer happens atomically
- Insert and remove items atomically
- Need some memory barriers: prevent insertion before initialization

Reclamation

- **What about reclamation?**
- Can remove item atomically: reader sees structure with or without
- Can't free item immediately:
- What if memory reused, reader interprets new data as pointer to item?
- Segmentation fault: core dumped
- (Best case scenario)

Reclamation

- What about reclamation?
- Can remove item atomically: reader sees structure with or without
- Can't free item immediately:
- What if memory reused, reader interprets new data as pointer to item?
- Segmentation fault: core dumped
- (Best case scenario)

Reclamation

- What about reclamation?
- Can remove item atomically: reader sees structure with or without
- **Can't free item immediately:**
 - What if memory reused, reader interprets new data as pointer to item?
 - Segmentation fault: core dumped
 - (Best case scenario)

Reclamation

- What about reclamation?
- Can remove item atomically: reader sees structure with or without
- Can't free item immediately:
- What if memory reused, reader interprets new data as pointer to item?
- Segmentation fault: core dumped
- (Best case scenario)

Reclamation

- What about reclamation?
- Can remove item atomically: reader sees structure with or without
- Can't free item immediately:
- What if memory reused, reader interprets new data as pointer to item?
- **Segmentation fault: core dumped**
- (Best case scenario)

Reclamation

- What about reclamation?
- Can remove item atomically: reader sees structure with or without
- Can't free item immediately:
- What if memory reused, reader interprets new data as pointer to item?
- Segmentation fault: core dumped
- (Best case scenario)

Deferred reclamation

- **Insertion fine anytime**
- Removal fine anytime, but. . .
- Can't reclaim an item out from under a reader
- Removal prevents new readers
- How to know current readers stopped using it?

Deferred reclamation

- Insertion fine anytime
- Removal fine anytime, but. . .
- Can't reclaim an item out from under a reader
- Removal prevents new readers
- How to know current readers stopped using it?

Deferred reclamation

- Insertion fine anytime
- Removal fine anytime, but. . .
- **Can't reclaim an item out from under a reader**
- Removal prevents new readers
- How to know current readers stopped using it?

Deferred reclamation

- Insertion fine anytime
- Removal fine anytime, but. . .
- Can't reclaim an item out from under a reader
- **Removal prevents new readers**
- How to know current readers stopped using it?

Deferred reclamation

- Insertion fine anytime
- Removal fine anytime, but. . .
- Can't reclaim an item out from under a reader
- Removal prevents new readers
- How to know current readers stopped using it?

Deferred reclamation procedure

- Remove item from structure, making it inaccessible to new readers
- Wait for all old readers to finish
- Free the old item
- Note: only synchronizes between readers and reclaimers, not writers
- Complements other synchronization

Deferred reclamation procedure

- Remove item from structure, making it inaccessible to new readers
- Wait for all old readers to finish
- Free the old item
- Note: only synchronizes between readers and reclaimers, not writers
- Complements other synchronization

Deferred reclamation procedure

- Remove item from structure, making it inaccessible to new readers
- Wait for all old readers to finish
- **Free the old item**
- Note: only synchronizes between readers and reclaimers, not writers
- Complements other synchronization

Deferred reclamation procedure

- Remove item from structure, making it inaccessible to new readers
- Wait for all old readers to finish
- Free the old item
- **Note: only synchronizes between readers and reclaimers, not writers**
- Complements other synchronization

Deferred reclamation procedure

- Remove item from structure, making it inaccessible to new readers
- Wait for all old readers to finish
- Free the old item
- Note: only synchronizes between readers and reclaimers, not writers
- **Complements other synchronization**

Epochs

- **Maintain per-thread and global epochs**
- Reads and writes associated with an epoch
- When all threads have passed an epoch, free items removed in previous epochs
- Reader needs atomic instructions, memory barriers

Epochs

- Maintain per-thread and global epochs
- Reads and writes associated with an epoch
- When all threads have passed an epoch, free items removed in previous epochs
- Reader needs atomic instructions, memory barriers

Epochs

- Maintain per-thread and global epochs
- Reads and writes associated with an epoch
- When all threads have passed an epoch, free items removed in previous epochs
- Reader needs atomic instructions, memory barriers

Epochs

- Maintain per-thread and global epochs
- Reads and writes associated with an epoch
- When all threads have passed an epoch, free items removed in previous epochs
- Reader needs atomic instructions, memory barriers

Hazard pointers

- Readers mark items in use with hazard pointers
- Writers check for removed items in all hazard pointers before freeing.
- Reader still needs atomic instructions, memory barriers

Hazard pointers

- Readers mark items in use with hazard pointers
- Writers check for removed items in all hazard pointers before freeing.
- Reader still needs atomic instructions, memory barriers

Hazard pointers

- Readers mark items in use with hazard pointers
- Writers check for removed items in all hazard pointers before freeing.
- Reader still needs atomic instructions, memory barriers

Problem: reader efficiency

- Epochs and hazard pointers have expensive read sides
- Readers must also write
- Readers must use atomic instructions
- Readers must use memory barriers
- Can we know readers have finished as an external observer?

Problem: reader efficiency

- Epochs and hazard pointers have expensive read sides
- Readers must also write
- Readers must use atomic instructions
- Readers must use memory barriers
- Can we know readers have finished as an external observer?

Problem: reader efficiency

- Epochs and hazard pointers have expensive read sides
- Readers must also write
- Readers must use atomic instructions
- Readers must use memory barriers
- Can we know readers have finished as an external observer?

Problem: reader efficiency

- Epochs and hazard pointers have expensive read sides
- Readers must also write
- Readers must use atomic instructions
- Readers must use memory barriers
- Can we know readers have finished as an external observer?

Problem: reader efficiency

- Epochs and hazard pointers have expensive read sides
- Readers must also write
- Readers must use atomic instructions
- Readers must use memory barriers
- Can we know readers have finished as an external observer?

Quiescent-state-based reclamation

- Define quiescent states for threads
 - Threads cannot hold item references in a quiescent state
 - Let “grace periods” contain a quiescent state for every thread
 - Wait for one grace period; every thread passes through a quiescent state
 - No readers could hold old references, new references can't see removed item

Quiescent-state-based reclamation

- Define quiescent states for threads
- **Threads cannot hold item references in a quiescent state**
- Let “grace periods” contain a quiescent state for every thread
- Wait for one grace period; every thread passes through a quiescent state
- No readers could hold old references, new references can't see removed item

Quiescent-state-based reclamation

- Define quiescent states for threads
- Threads cannot hold item references in a quiescent state
- Let “grace periods” contain a quiescent state for every thread
- Wait for one grace period; every thread passes through a quiescent state
- No readers could hold old references, new references can't see removed item

Quiescent-state-based reclamation

- Define quiescent states for threads
- Threads cannot hold item references in a quiescent state
- Let “grace periods” contain a quiescent state for every thread
- Wait for one grace period; every thread passes through a quiescent state
- No readers could hold old references, new references can't see removed item

Quiescent-state-based reclamation

- Define quiescent states for threads
- Threads cannot hold item references in a quiescent state
- Let “grace periods” contain a quiescent state for every thread
- Wait for one grace period; every thread passes through a quiescent state
- No readers could hold old references, new references can't see removed item

Read Copy Update (RCU)

- **Read-side critical sections**
 - Items don't disappear inside critical section
 - Quiescent states outside critical section
 - Writers must guarantee reader correctness at every point
 - In theory: copy entire data structure, replace pointer
 - In practice: insert or remove items atomically
 - Writers defer reclamation by waiting for read-side critical sections
 - Writers may block and reclaim, or register a reclamation callback

Read Copy Update (RCU)

- Read-side critical sections
- **Items don't disappear inside critical section**
- Quiescent states outside critical section
- Writers must guarantee reader correctness at every point
- In theory: copy entire data structure, replace pointer
- In practice: insert or remove items atomically
- Writers defer reclamation by waiting for read-side critical sections
- Writers may block and reclaim, or register a reclamation callback

Read Copy Update (RCU)

- Read-side critical sections
- Items don't disappear inside critical section
- **Quiescent states outside critical section**
- Writers must guarantee reader correctness at every point
- In theory: copy entire data structure, replace pointer
- In practice: insert or remove items atomically
- Writers defer reclamation by waiting for read-side critical sections
- Writers may block and reclaim, or register a reclamation callback

Read Copy Update (RCU)

- Read-side critical sections
- Items don't disappear inside critical section
- Quiescent states outside critical section
- **Writers must guarantee reader correctness at every point**
- In theory: copy entire data structure, replace pointer
- In practice: insert or remove items atomically
- Writers defer reclamation by waiting for read-side critical sections
- Writers may block and reclaim, or register a reclamation callback

Read Copy Update (RCU)

- Read-side critical sections
- Items don't disappear inside critical section
- Quiescent states outside critical section
- Writers must guarantee reader correctness at every point
- **In theory: copy entire data structure, replace pointer**
- In practice: insert or remove items atomically
- Writers defer reclamation by waiting for read-side critical sections
- Writers may block and reclaim, or register a reclamation callback

Read Copy Update (RCU)

- Read-side critical sections
- Items don't disappear inside critical section
- Quiescent states outside critical section
- Writers must guarantee reader correctness at every point
- In theory: copy entire data structure, replace pointer
- **In practice: insert or remove items atomically**
- Writers defer reclamation by waiting for read-side critical sections
- Writers may block and reclaim, or register a reclamation callback

Read Copy Update (RCU)

- Read-side critical sections
- Items don't disappear inside critical section
- Quiescent states outside critical section
- Writers must guarantee reader correctness at every point
- In theory: copy entire data structure, replace pointer
- In practice: insert or remove items atomically
- **Writers defer reclamation by waiting for read-side critical sections**
- Writers may block and reclaim, or register a reclamation callback

Read Copy Update (RCU)

- Read-side critical sections
- Items don't disappear inside critical section
- Quiescent states outside critical section
- Writers must guarantee reader correctness at every point
- In theory: copy entire data structure, replace pointer
- In practice: insert or remove items atomically
- Writers defer reclamation by waiting for read-side critical sections
- **Writers may block and reclaim, or register a reclamation callback**

Classic RCU

- Read lock: disable preemption
- Read unlock: enable preemption
- Quiescent state: context switch
- Scheduler flags quiescent states
- Readers perform no expensive operations

Classic RCU

- Read lock: disable preemption
- Read unlock: enable preemption
- Quiescent state: context switch
- Scheduler flags quiescent states
- Readers perform no expensive operations

Classic RCU

- Read lock: disable preemption
- Read unlock: enable preemption
- **Quiescent state: context switch**
- Scheduler flags quiescent states
- Readers perform no expensive operations

Classic RCU

- Read lock: disable preemption
- Read unlock: enable preemption
- Quiescent state: context switch
- Scheduler flags quiescent states
- Readers perform no expensive operations

Classic RCU

- Read lock: disable preemption
- Read unlock: enable preemption
- Quiescent state: context switch
- Scheduler flags quiescent states
- Readers perform no expensive operations

Realtime RCU

- Quiescent states tracked by per-CPU counters
- read lock, read unlock: manipulate counters
- Readers perform no expensive operations
- Allows preemption in critical sections
- Less efficient than classic RCU

Realtime RCU

- Quiescent states tracked by per-CPU counters
- read lock, read unlock: manipulate counters
- Readers perform no expensive operations
- Allows preemption in critical sections
- Less efficient than classic RCU

Realtime RCU

- Quiescent states tracked by per-CPU counters
- read lock, read unlock: manipulate counters
- Readers perform no expensive operations
- Allows preemption in critical sections
- Less efficient than classic RCU

Realtime RCU

- Quiescent states tracked by per-CPU counters
- read lock, read unlock: manipulate counters
- Readers perform no expensive operations
- **Allows preemption in critical sections**
- Less efficient than classic RCU

Realtime RCU

- Quiescent states tracked by per-CPU counters
- read lock, read unlock: manipulate counters
- Readers perform no expensive operations
- Allows preemption in critical sections
- Less efficient than classic RCU

Read-mostly structures

- RCU ideal for read-mostly structures
 - Permissions
 - Hardware configuration data
 - Routing tables and firewall rules

Read-mostly structures

- RCU ideal for read-mostly structures
- **Permissions**
- Hardware configuration data
- Routing tables and firewall rules

Read-mostly structures

- RCU ideal for read-mostly structures
- Permissions
- **Hardware configuration data**
- Routing tables and firewall rules

Read-mostly structures

- RCU ideal for read-mostly structures
- Permissions
- Hardware configuration data
- Routing tables and firewall rules

Synchronizing between updates

- RCU doesn't solve this
 - Need separate synchronization to coordinate updates
 - Can build on non-blocking synchronization or locking
 - Many non-blocking algorithms don't account for reclamation at all
 - Can add RCU to avoid memory leaks
 - Reclamation strategy mostly orthogonal from update strategy

Synchronizing between updates

- RCU doesn't solve this
- **Need separate synchronization to coordinate updates**
- Can build on non-blocking synchronization or locking
- Many non-blocking algorithms don't account for reclamation at all
- Can add RCU to avoid memory leaks
- Reclamation strategy mostly orthogonal from update strategy

Synchronizing between updates

- RCU doesn't solve this
- Need separate synchronization to coordinate updates
- Can build on non-blocking synchronization or locking
- Many non-blocking algorithms don't account for reclamation at all
- Can add RCU to avoid memory leaks
- Reclamation strategy mostly orthogonal from update strategy

Synchronizing between updates

- RCU doesn't solve this
- Need separate synchronization to coordinate updates
- Can build on non-blocking synchronization or locking
- Many non-blocking algorithms don't account for reclamation at all
- Can add RCU to avoid memory leaks
- Reclamation strategy mostly orthogonal from update strategy

Synchronizing between updates

- RCU doesn't solve this
- Need separate synchronization to coordinate updates
- Can build on non-blocking synchronization or locking
- Many non-blocking algorithms don't account for reclamation at all
- Can add RCU to avoid memory leaks
- Reclamation strategy mostly orthogonal from update strategy

Synchronizing between updates

- RCU doesn't solve this
- Need separate synchronization to coordinate updates
- Can build on non-blocking synchronization or locking
- Many non-blocking algorithms don't account for reclamation at all
- Can add RCU to avoid memory leaks
- Reclamation strategy mostly orthogonal from update strategy

Memory consistency model

- **Handles non-sequentially-consistent memory**
- Minimal memory barriers
- Does not provide sequential consistency
- Provides weaker consistency model
- Readers may see writes in any order
- Readers cannot see an inconsistent intermediate state
- Does not provide linearizability
- Many algorithms do not require these guarantees

Memory consistency model

- Handles non-sequentially-consistent memory
- **Minimal memory barriers**
- Does not provide sequential consistency
- Provides weaker consistency model
- Readers may see writes in any order
- Readers cannot see an inconsistent intermediate state
- Does not provide linearizability
- Many algorithms do not require these guarantees

Memory consistency model

- Handles non-sequentially-consistent memory
- Minimal memory barriers
- Does not provide sequential consistency
- Provides weaker consistency model
- Readers may see writes in any order
- Readers cannot see an inconsistent intermediate state
- Does not provide linearizability
- Many algorithms do not require these guarantees

Memory consistency model

- Handles non-sequentially-consistent memory
- Minimal memory barriers
- Does not provide sequential consistency
- Provides weaker consistency model
- Readers may see writes in any order
- Readers cannot see an inconsistent intermediate state
- Does not provide linearizability
- Many algorithms do not require these guarantees

Memory consistency model

- Handles non-sequentially-consistent memory
- Minimal memory barriers
- Does not provide sequential consistency
- Provides weaker consistency model
- Readers may see writes in any order
- Readers cannot see an inconsistent intermediate state
- Does not provide linearizability
- Many algorithms do not require these guarantees

Memory consistency model

- Handles non-sequentially-consistent memory
- Minimal memory barriers
- Does not provide sequential consistency
- Provides weaker consistency model
- Readers may see writes in any order
- Readers cannot see an inconsistent intermediate state
- Does not provide linearizability
- Many algorithms do not require these guarantees

Memory consistency model

- Handles non-sequentially-consistent memory
- Minimal memory barriers
- Does not provide sequential consistency
- Provides weaker consistency model
- Readers may see writes in any order
- Readers cannot see an inconsistent intermediate state
- Does not provide linearizability
- Many algorithms do not require these guarantees

Memory consistency model

- Handles non-sequentially-consistent memory
- Minimal memory barriers
- Does not provide sequential consistency
- Provides weaker consistency model
- Readers may see writes in any order
- Readers cannot see an inconsistent intermediate state
- Does not provide linearizability
- Many algorithms do not require these guarantees

Performance testing

- Tested RCU and hazard pointers, with locking or NBS
- All better than locking
- RCU variants: near-ideal performance
- Best performer for low write fractions
- Highly competitive for higher write fractions

Performance testing

- Tested RCU and hazard pointers, with locking or NBS
- All better than locking
- RCU variants: near-ideal performance
- Best performer for low write fractions
- Highly competitive for higher write fractions

Performance testing

- Tested RCU and hazard pointers, with locking or NBS
- All better than locking
- RCU variants: near-ideal performance
- Best performer for low write fractions
- Highly competitive for higher write fractions

Performance testing

- Tested RCU and hazard pointers, with locking or NBS
- All better than locking
- RCU variants: near-ideal performance
- **Best performer for low write fractions**
- Highly competitive for higher write fractions

Performance testing

- Tested RCU and hazard pointers, with locking or NBS
- All better than locking
- RCU variants: near-ideal performance
- Best performer for low write fractions
- Highly competitive for higher write fractions

Conclusion

- RCU implements quiescent-state-based deferred reclamation
 - No expensive overhead for readers
 - Minimally expensive overhead for writers
 - Ideal for read-mostly situations

Conclusion

- RCU implements quiescent-state-based deferred reclamation
- **No expensive overhead for readers**
- Minimally expensive overhead for writers
- Ideal for read-mostly situations

Conclusion

- RCU implements quiescent-state-based deferred reclamation
- No expensive overhead for readers
- **Minimally expensive overhead for writers**
- Ideal for read-mostly situations

Conclusion

- RCU implements quiescent-state-based deferred reclamation
- No expensive overhead for readers
- Minimally expensive overhead for writers
- **Ideal for read-mostly situations**