Microkernel Construction

Threads and Synchronization
SS2011

Torsten
Frenzel

TU Dresden
Operating
Systems Group

Outline

- Thread creation and destruction
- Synchronization
  - Synchronization primitives
    - Lock-based synchronization
    - Lock-free synchronization
      - Synchronization of linked lists
      - Synchronization of bitfiels
    - Wait-free synchronization
      - Helping-lock
      - Thread-lock
- Synchronization and thread destruction

Torsten Frenzel

TU Dresden
Operating Systems Group

Threads (Revisited)

- Threads are kernel objects
  - TCB (Thread Control Block)
    - Holds all information about a thread
    - Located in the kernel
    - Needs to be backed with memory
  - Has to be constructed and destructed
  - Active entities that execute system calls
  - Can manipulate data structures and even other threads
  - Subject and object of synchronization
    - Can hold locks
    - Can be locked

Torsten Frenzel

TU Dresden Operating Systems Group

Thread Creation (1)

1. Create new TCB
   - Allocate memory for TCB
   - Initialize TCB
2. Enqueue thread into ready-list
3. Two cases to switch to new thread
   - New thread is selected by the scheduler
   - New thread has higher priority than creating thread
     - Cooperative switch immediately after creation
4. Started thread immediately switches to user mode
   - Kernel has special exit path:
     - Initialize and clean all user registers
     - Return to user-level

Torsten Frenzel

TU Dresden Operating Systems Group

Thread Creation (2)

- **Implicit creation**
  - Creation occurs on demand
  - On access to TCB:
    - On read access: map shared read-only zero page
    - On write-access: allocate and map new page
- **Explicit creation**
  - Creation occurs on request using system calls
  - ➔ Avoid unwanted allocation due to program errors
  - ➔ Avoid ambigious ownership of allocated memory to enforce quotas

Torsten Frenzel

TU Dresden Operating Systems Group

Thread Destruction (1)

- Cleanup thread state
  1. Release all locks owned by thread
  2. Set state to *dead* (never run this thread again!)
  3. Remove hanging references to this thread:
     1. Detach from interrupts
     2. Remove from timeout queues
     3. Remove from present queue
     4. Remove from ready queue
  4. Dequeue waiting senders
  5. Stop possible ongoing IPC operation!
  6. If no pending references to this thread exist anymore:
     - Free TCB memory

Torsten Frenzel

TU Dresden Operating Systems Group

Thread Destruction (2)

- Releasing locks
  - Lock target thread
  - If *lock counter* of target thread is greater zero
    - ➔ Switch to target thread to release locks
  - Scheduler:
    - Run target thread despite beeing locked
    - Target thread should execute code to release locks!
      - Prevent exit to user mode
      - Low complexity of kernel operations assures fast progress
  - On last owned lock release operation:
    - Stop target thread and switch to someone else (schedule)

Torsten Frenzel

TU Dresden Operating Systems Group

Why Synchronization?

- Control concurrent access to data structures inside the kernel
  - Threads execute kernel code in parallel
  - Access concurrent to resources and data structures
  - Avoid inconsistencies and data corruption
  - Provide mutual exclusion for critical sections
- Object of the lock
  - Defines the data to be protected
- Subject of the lock
  - Defines the *lock holder* or *lock owner*
  - In the context of microkernels this is a thread

Torsten Frenzel

TU Dresden
Operating Systems Group

Synchronization Properties

- Granularity
  - Fine-grained synchronization allows high concurrency
  - Coarse-grained synchronization reduces overhead
- Overhead
  - Depends on the used synchronization primitive
  - Depends on the number critical sections
- Fairness
  - Avoid starvation of threads that want to enter a critical section
- Lock holder preemption
  - The lock holder should always run to leave the critical section
- Multi-processor synchronization
  - Parallel execution of multiple threads inside the kernel
  - Requires higher synchronization effort compared to single-processor synchronization

Torsten Frenzel

TU Dresden Operating Systems Group

Preemptibility and Concurrency

- No preemptibility
  - Only one thread executes code inside the kernel
  - Kernel is protected with one big lock
- Restricted preemptibility
  - Allowing preemption at *preemption points*
  - Preemption point defines consistent state
  - Requires *safe* resume of preempted operation
  - Suitable for multi-processor synchronization
- High preemptibility
  - Thread can be preempted at any point in time
  - State has to be consistent for all possible preemptions that can occur
  - Only applicable for special data structures

Torsten Frenzel

TU Dresden
Operating Systems Group

Real-Time Requirements

- Low latency (interrupt and timeout response is critical)
  - **Fast synchronization primitives**
  - **Thread and task switching must be fast**
- High preemptibility
  - Full preemptibility (hard to implement correctly)
  - **Short non-preemptible critical sections with preemption points**
- Avoid priority inversion when using locks by using special protocols
  - Priority ceiling
  - **Priority inheritance**

Torsten Frenzel

TU Dresden Operating Systems Group

Which data to protect?

- Thread control blocks (TCB)
  - state field
  - pager, preempter, ...
  - kernel stack
- UTCB
- Present list and ready list
- Mapping database (not discussed in this lecture)
- Kernel memory allocators
- Memory space (page tables)
- Capability space (capability tables)

Torsten Frenzel

TU Dresden Operating Systems Group

Synchronization Primitives

- Disabling Interrupts
- Blocking synchronization
  - Semaphore
  - Spin locks
  - Ticket locks
  - MCS locks
- Non-blocking synchronization
  - Lock-free synchronization
    - Uses atomic update operations
    - For simple data structures
  - Wait-free synchronization
    - Uses locks that implement priority inheritance
    - For complex data structures

Torsten Frenzel

TU Dresden Operating Systems Group

Disabling Interrupts

- Easy to implement but not multi-processor safe:
  - For CPU-local data structures (stack)
  - For kernel entry/exit code
- Granularity
  - Disable all interrupts: has lower costs
  - Disable priority levels: allows
- Pessimistic
  - Always acquire the lock for entering critical section
  - → Costs are equal in preemption and non-preemption case
- Optimistic
  - Do not acquire lock before entering critical section
  - If preemption occurs acquire lock and resume critical section
  - → Low costs in non-preemption case and high costs in preemption case
  - Not implemented in real-world systems

Torsten Frenzel

TU Dresden Operating Systems Group

Blocking Synchronization

- Semaphore (sleeping locks)
  - Put the waiting thread that wants to enter the critical section to sleep and schedule another thread
  - Wake up waiting thread when critical section becomes free again
  - → High synchronization overhead because of scheduling overhead and wait queue maintenance
  - → Suitable for long critical sections
- Spin Locks (spinning locks)
  - Waiting thread that wants to enter the critical section spins in a tight loop until critical section becomes free again
  - → Suitable for short critcal sections
  - → Burns processing time by idling

Overview: Spin Locks

- Test and set lock
  - Try to aquire lock, if it fails retry (spin)
- Test and test and set lock
  - Try to aquire the lock only if it is free else spin
  - → Avoid cache-line bouncing and bus traffic
  - → Reduces overhead
- Ticket locks
  - Ticket is increased by every thread that wants to acquire lock
  - Counter is increased when thread releases the lock
  - → Guarantees fairness
- MCS locks
  - Threads spin inside queue on local variable
  - → Minimizes overhead and guarantees fairness
- → See lecture: *Distributed Operating Systems* for details

Torsten Frenzel

TU Dresden Operating Systems Group

Reader-Writer Locks

- Idea: Concurrent access of readers to critical section is possible without interference
  - Many readers can enter the critical section
  - Only one writer can enter exclusively the critical section
- Fairness between readers and writers is hard to achieve
  - Preference of readers
    - Writer can only enter critical section, if no reader wants to enter
  - Preference is writers:
    - Reader can only enter critical section, if no writer wants to
    - enter
  - Complete fairness requires tickets or sorted queue

Non-blocking Synchronization

- Avoids blocking
- Avoids unlimited priority inversion

- Lock-free synchronization
  - No locks
  - Atomic memory update
    - Disabling interrupts
    - Atomic instructions
  - Multiprocessor-safe
  - Deadlock free
- Wait-free synchronization
  - Special locks with *helping* semantic
    - Switch to preempted lock holder (which is a thread)

Torsten Frenzel

TU Dresden Operating Systems Group

Lock-Free Synchronization (1)

- Principle:
  1. Prepare data out of line
  2. Atomically try to exhange old data for new data
  3. If it fails, retry
- Properties
  - Requires atomic memory operations
  - No lock
    - → No deadlock
    - → Crashed threads hold no locks
    - → Trivially multiprocessor-safe
  - Bounded number of retries [Anderson]
  - Hard to implement correctly for complex data structures

Lock-Free Synchronization (2)

- Requires atomic memory-update operation
  - Single compare-and-swap operation
    (x86: CMPXCHG, CMPXCHG8B)
    - Counter
    - Bit field
    - LIFO stack
    - FIFO queue
    - Single-linked list with insertion/deletion in the middle
  - Double compare-and-swap operation
    (Motorola 68K: DCAS)
    - Double-linked list
    - Tree

Atomic Memory Operations

- Syntax: `CAS(addr, old, new)`
  - `addr`: Address of memory location that should be changed
  - `old`: old value of the memory location
  - `new`: new value of the memory location
  - returns failure or success
- Semantic:

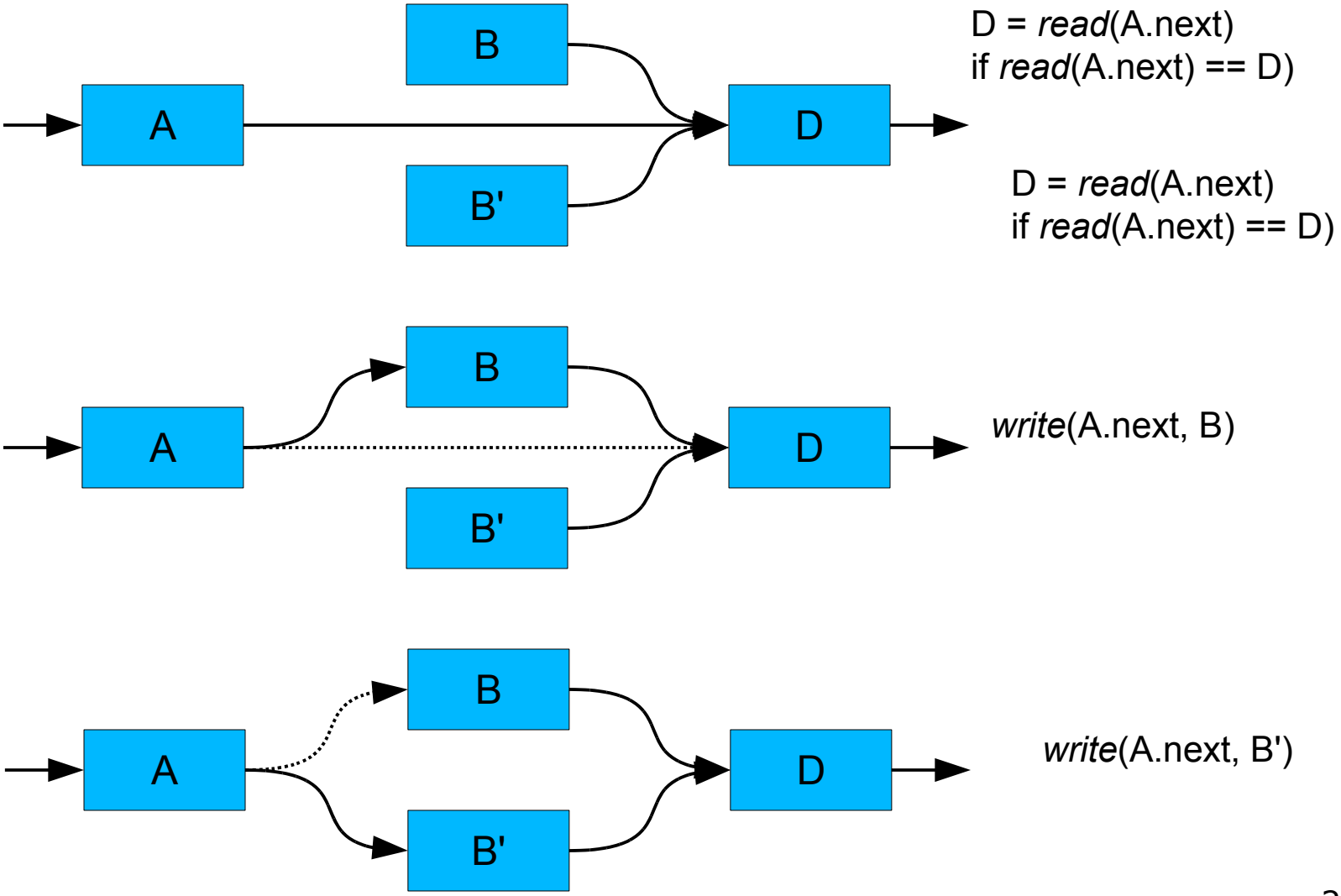
```
if (read(addr) == old)
   write(addr, new)
   return success
else
   return failure
```

Torsten Frenzel

TU Dresden Operating Systems Group

Single-Linked List

- Concurrent insert without atomic operations:

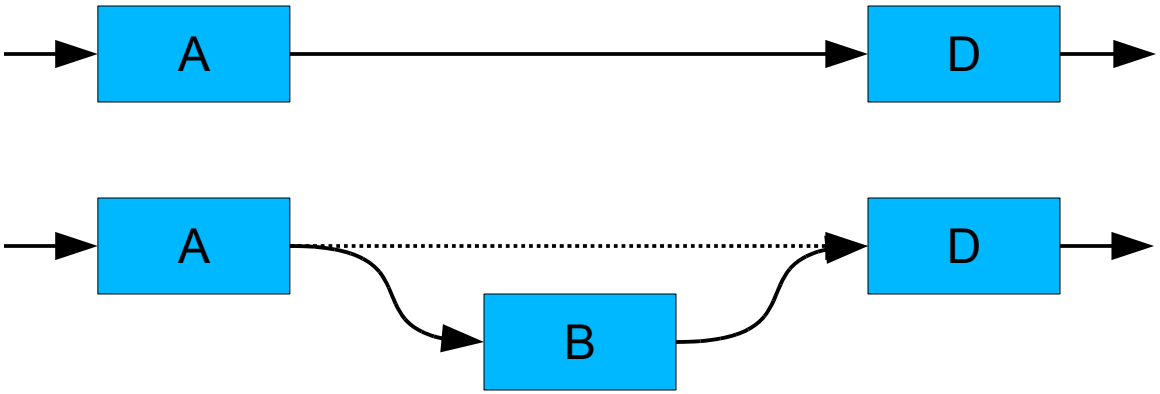


Torsten
Frenzel

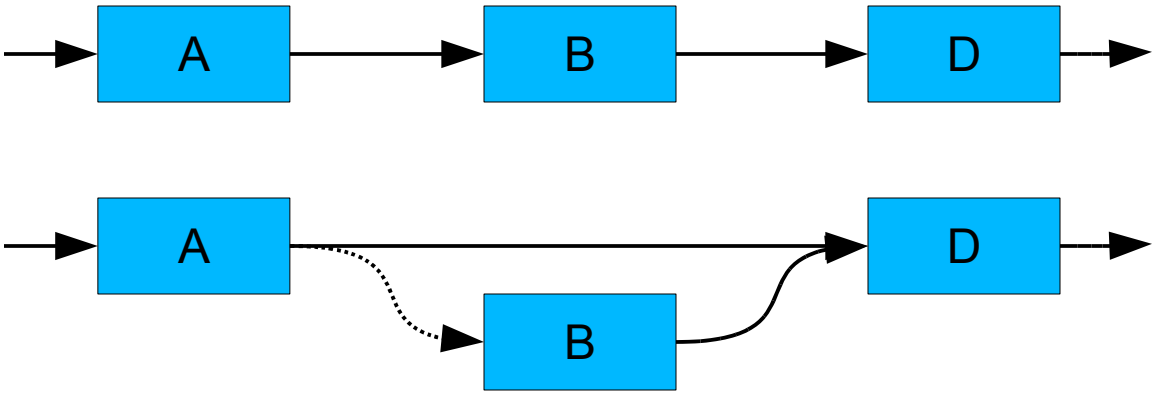
TU Dresden
Operating
Systems Group

Example: Single-Linked List

- Insert with atomic operation:



- Delete with atomic operation:

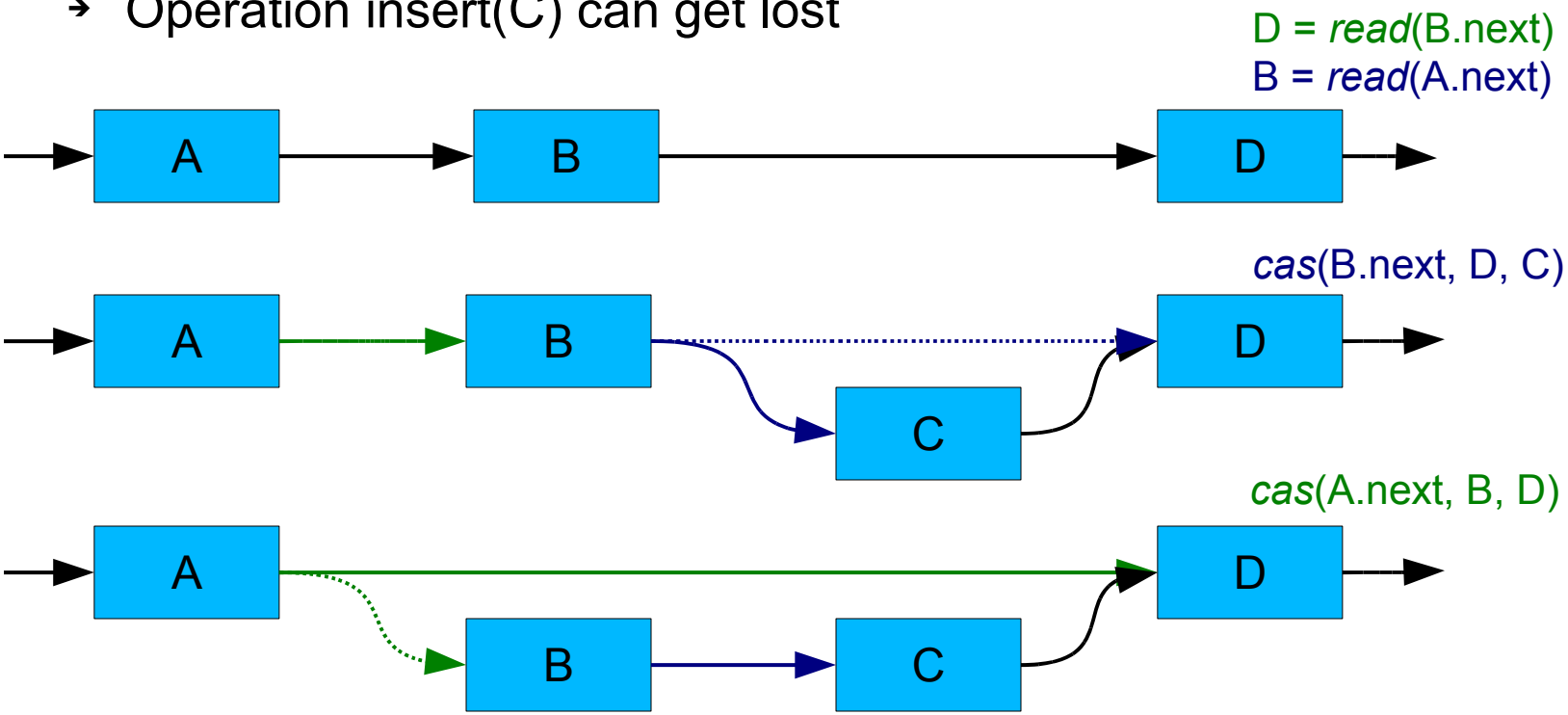


Torsten Frenzel

TU Dresden Operating Systems Group

Example: Single-Linked List

- Concurrent delete and insert operations:
  - Delete of element B and insert of element C
    - Operation delete(B) changes pointer A.next from B to D
    - Operation insert(C) changes pointer B.next from D to C
  - ➔ Operation insert(C) can get lost

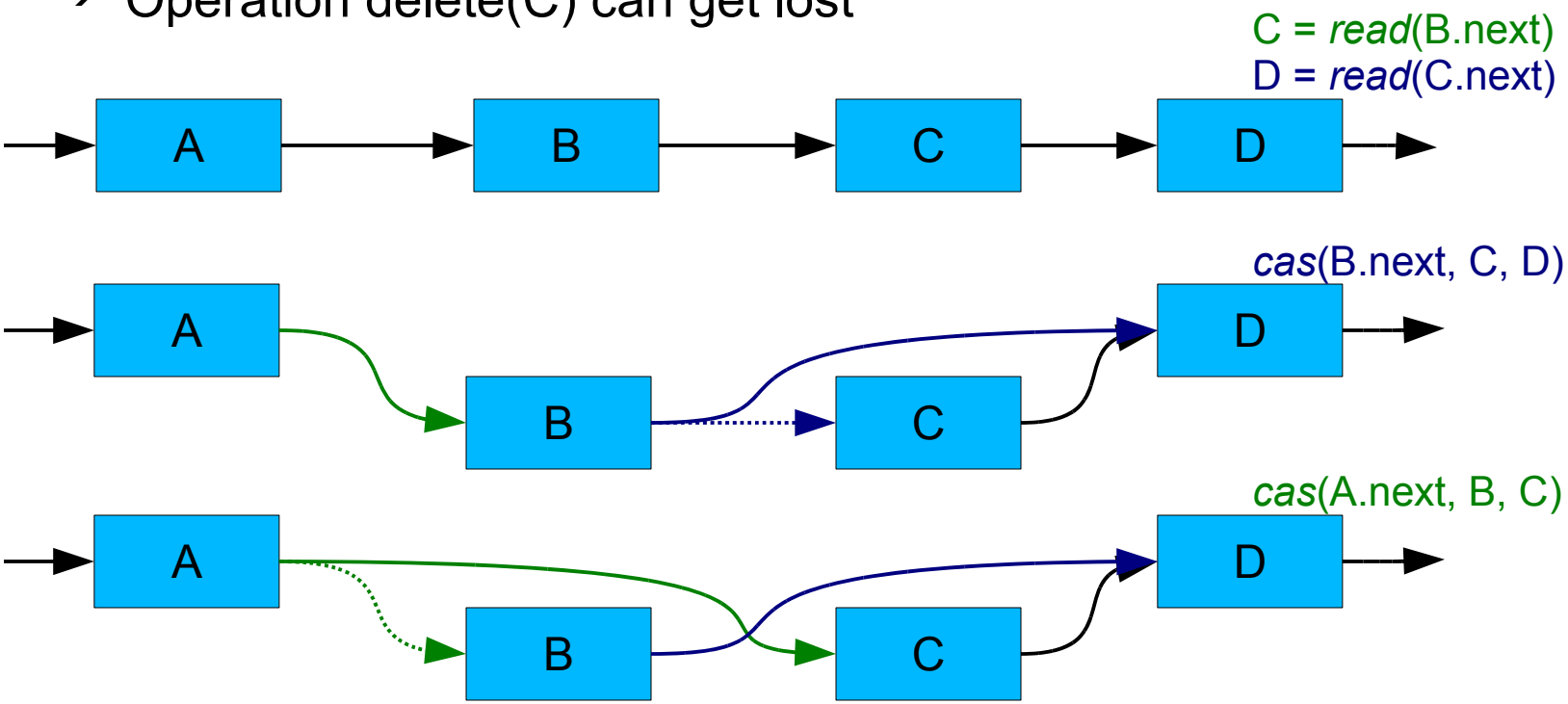


Torsten Frenzel

TU Dresden Operating Systems Group

Example: Single-Linked List

- Concurrent delete operations:
  - Deletion of elements B and C
    - Operation delete(B) changes pointer A.next from B to C
    - Operation delete(C) changes pointer B.next from C to D
  - ➔ Operation delete(C) can get lost

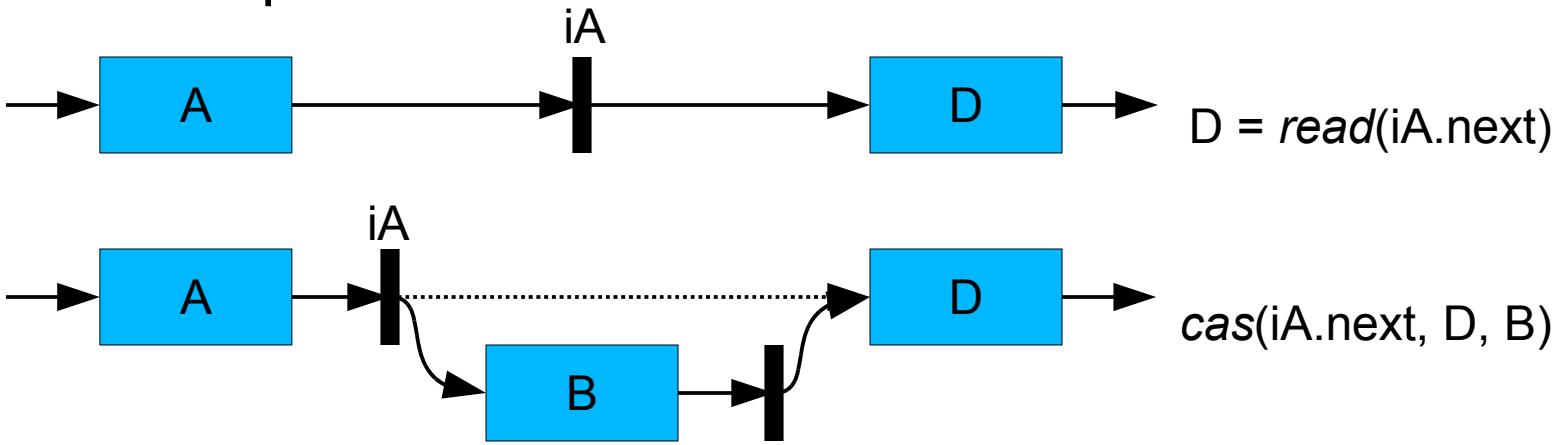


Torsten Frenzel

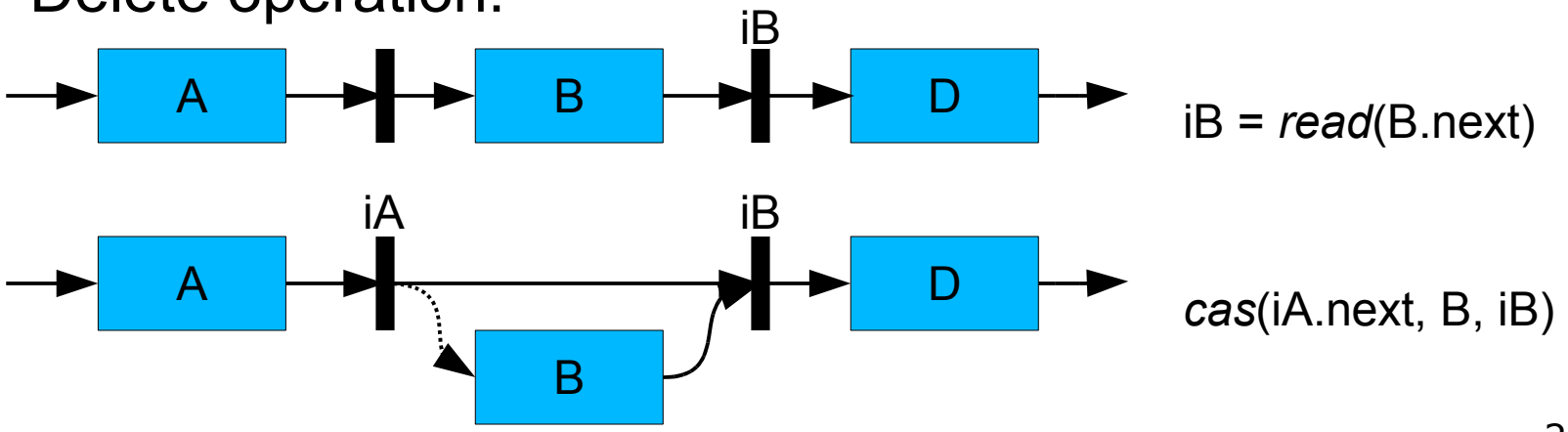
TU Dresden Operating Systems Group

Example: Single-Linked List

- Idea: Introduce intermediate nodes
- Insert operation:



- Delete operation:

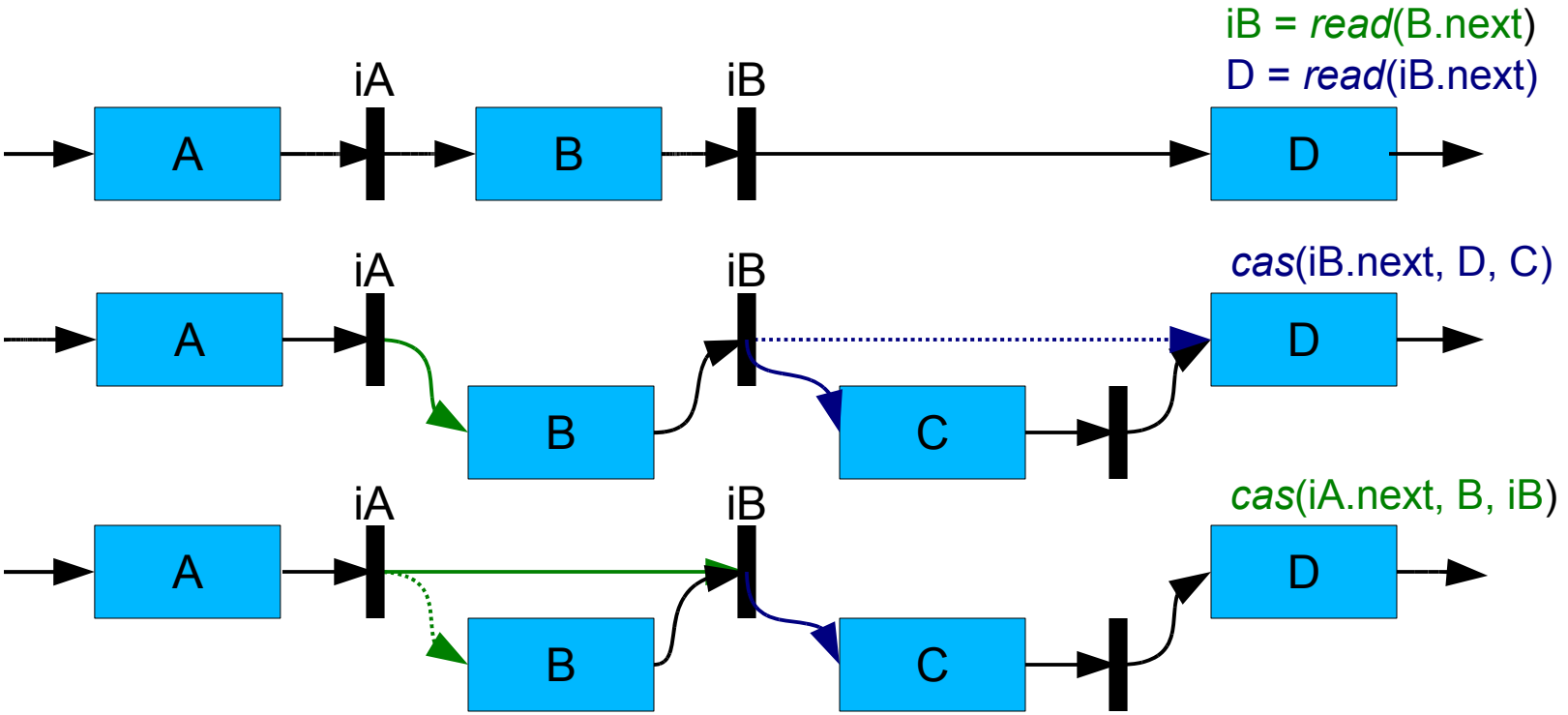


Torsten Frenzel

TU Dresden Operating Systems Group

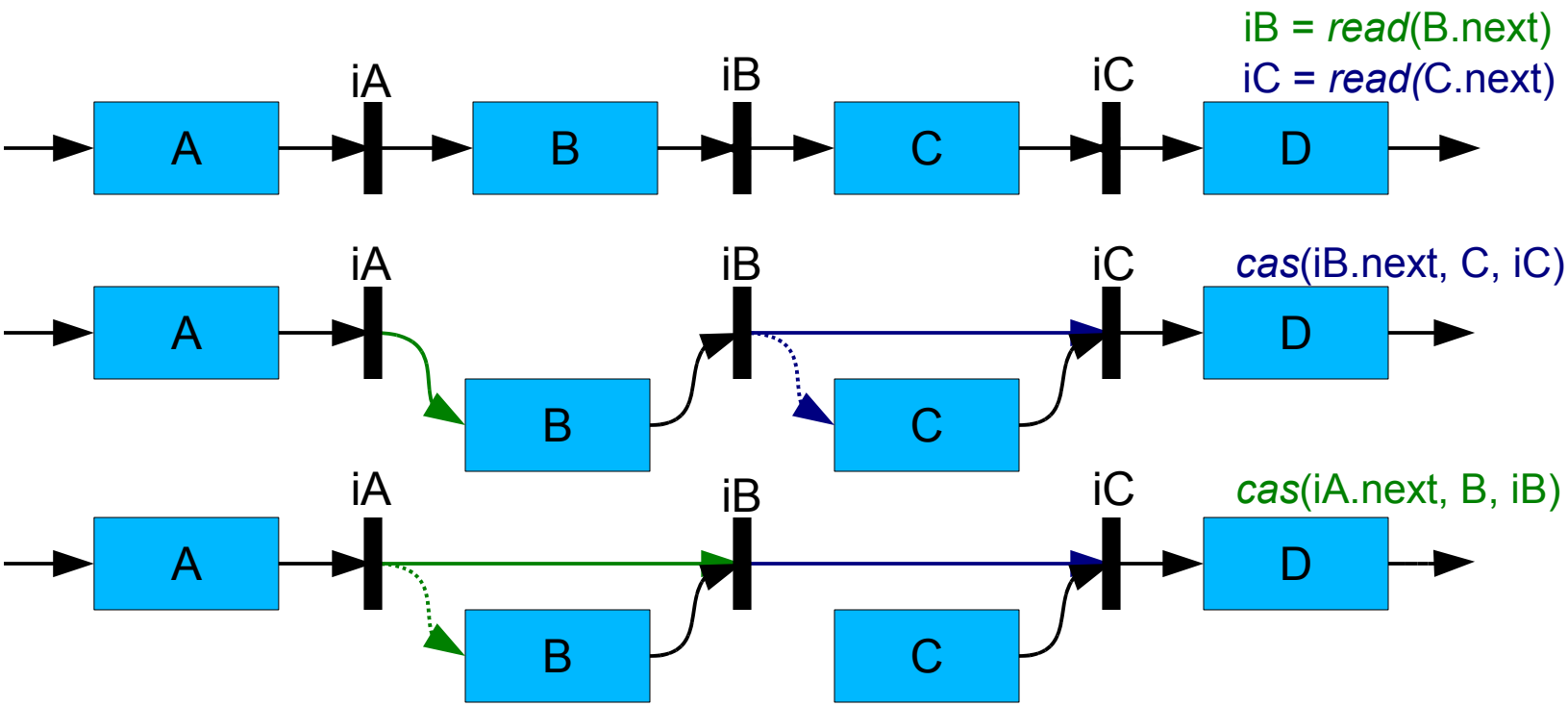
Example: Single-Linked List

- Concurrent delete and insert operations:
  - Delete of element B and insert of element C
    - Operation delete(B) changes pointer iA.next from B to iB
    - Operation insert(C) changes pointer iB.next from D to C



Example: Single-Linked List

- Concurrent delete operations:
  - Deletion of elements B and C
    - Operation delete(B) updates pointer i(A)->B to i(A)->i(B)
    - Operation delete(C) updates pointer i(B)->C to i(B)->i(C)



ABA-Problem (1)

- Problem:
  - Reading a value twice with a preemption point in between cannot detect changes that leave this value unmodified
- Example:
  - Thread T1 reads value A
  - Thread T1 is preempted and Thread T2 runs
  - Thread T2 modifies the value A to value B and back to A before preemption
  - Thread T1 begins execution again, sees that the value has not changed and continues

Torsten Frenzel

TU Dresden Operating Systems Group

ABA-Problem (2)

- Possible solutions:
  - Double-CAS:
    - Add an additional tag value that is incremented on every update of the pointer
    - Double-CAS is used to update both values atomically
  - Load-locked/store-conditional:
    - Load-locked operation detects changes of the pointer after the read operation
    - Store-conditional only succeeds if the pointer has not changed since the load operation
  - Transactional memory:
    - Defines transactions on arbitrary memory access sequences

Example: Bitfield (1)

- Changing a bits in three different ways:
  - `change1(clear_cond, set_cond, set)`
    - `clear_cond`: flip a number of bits from 1 to 0
    - `set_cond`: flip a number of bits from 0 to 1
    - `set`: set a number of bits (unconditionally)
  - *Condition 1*: bits in set `clear_cond` must be 1 in old state
  - *Condition 2*: bits in set `set_cond` must be 0 in old state
  - *Condition 3*: all three sets must be disjunct
- Algorithm:

```
1: old_state := state
   if (not condition1 or not condition2 or not condition3)
     goto 1
    new_state := old_state & ~clear_cond | set_cond | set
   if failed cas(state, old_state, new_state)
     goto 1
```

Torsten Frenzel

TU Dresden Operating Systems Group

Example: Bitfield (2)

- New version with only two parameters
  - `change2(c_mask, s_mask)`
    - `mask_c`: clear bits not set in this mask
    - `set_c`: set bits set in this mask
  - *Condition 1*: Bits cleared in `c_mask` must be set in old state
  - *Condition 2*: Bits set in `s_mask` must be clear in old state or clear in `c_mask`
- Algorithm:

```
1: old_state := state
if (not condition1 or not condition2)
  goto 1
new_state := old_state & c_mask | s_mask
if failed cas(state, old_state, new_state)
  goto 1
```

- Transformation of parameters from `change1`:
  - `c_mask := ~clear_cond & ~(set & old_state)`
  - `s_mask := set_cond | set`

Example: Bitfield (3)

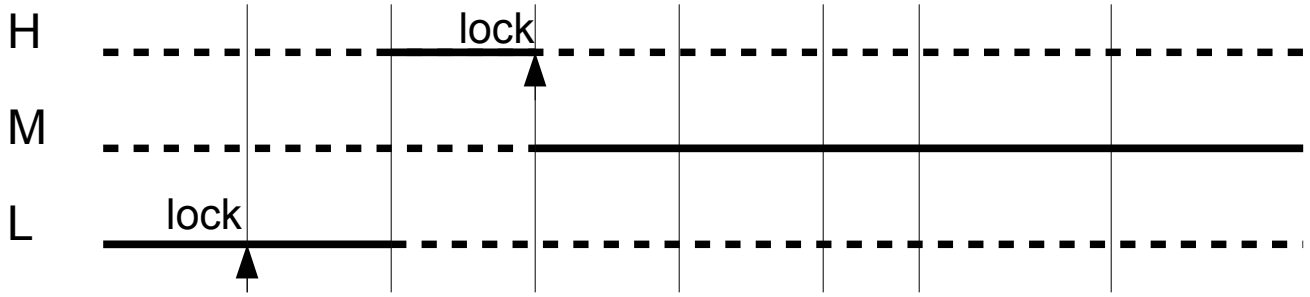
- Example:
  - `old_state := 1100`
  - Parameters for function `change1`:
    - `clear_cond := 1000`
    - `set_cond := 0001`
    - `set := 0110`
  - All three condition hold!
  - Using function `change1` returns:
    - `new_state := 1100 & ~(1000) | 0001 | 0110 = 0111`
  - Transformation of parameters for function `change2`:
    - `c_mask := ~(1000) & ~(0110 & 1100) = 0011`
    - `s_mask := 0001 | 0110 = 0111`
  - Both conditions must hold!
  - Using function `change2` returns:
    - `new_state := 1100 & 0011 | 0111 = 0111`

Torsten Frenzel

TU Dresden Operating Systems Group

Wait-Free Synchronization (1)

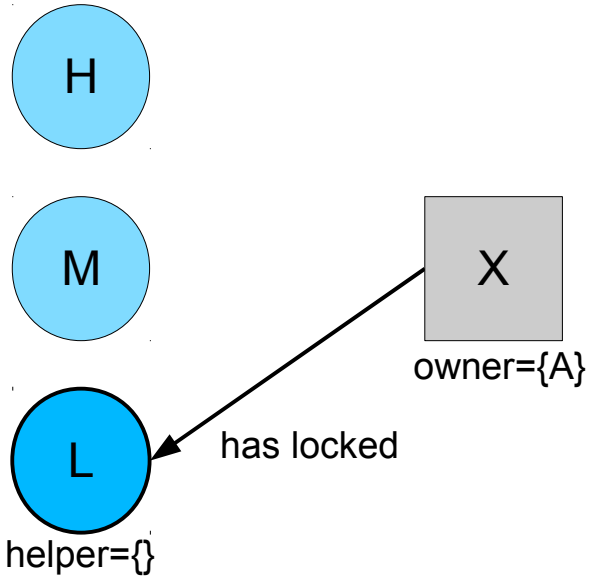
- Problem of priority inversion:

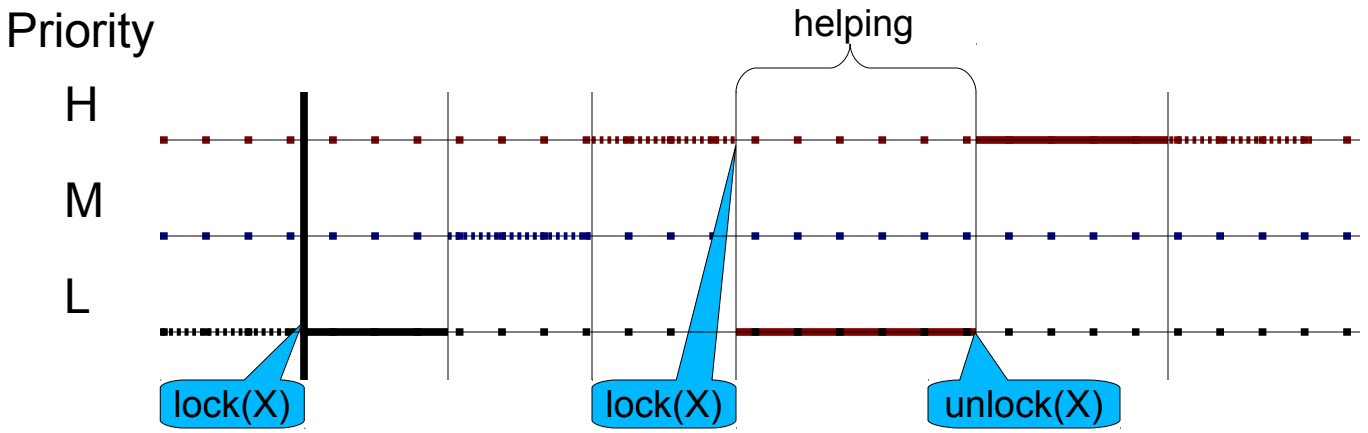


- Principle of helping:
  1. Thread L is in critical section
  2. Thread H wants to enter
  3. Thread H *helps* thread L to finish critical section
  4. Thread L switches to thread H after critical section
- Properties
  - No unlimited priority inversion (implements priority inheritance)
  - No starvation and a bounded number of retries
  - Not easy to implement for multi-processor systems

Basic Helping (1)

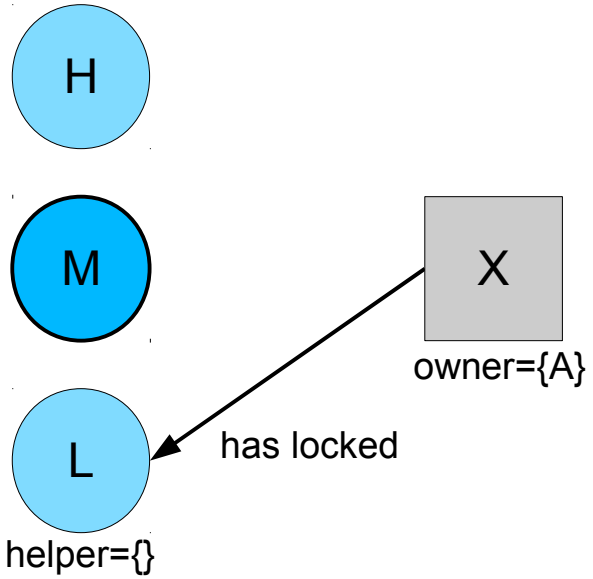
1. **L acquires lock X**
2. M preempts L
3. H preempts M
4. H wants lock X and helps lock owner L
5. L releases lock X and switches to helper H which acquires lock X
6. H releases lock X

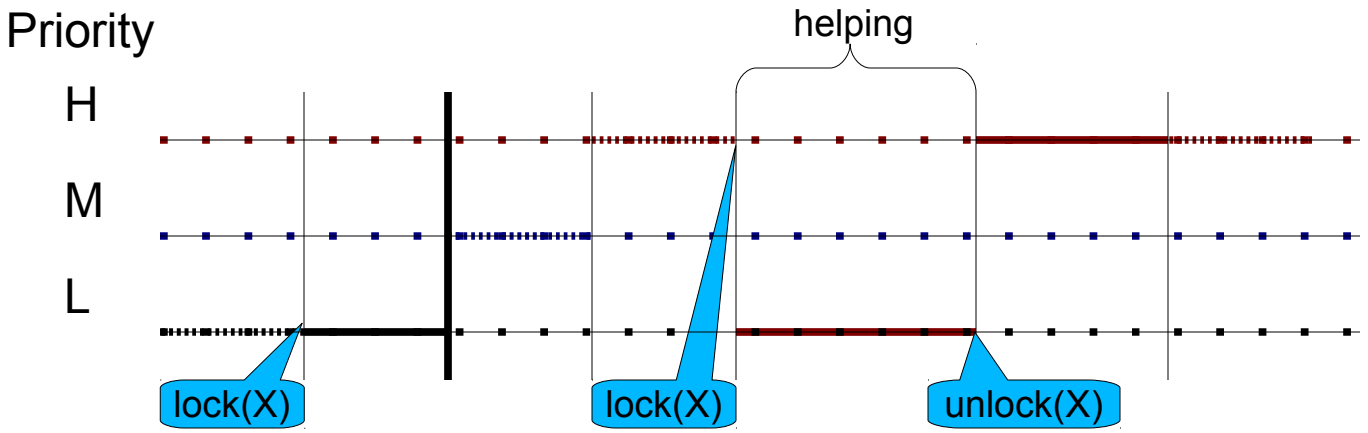




Basic Helping (2)

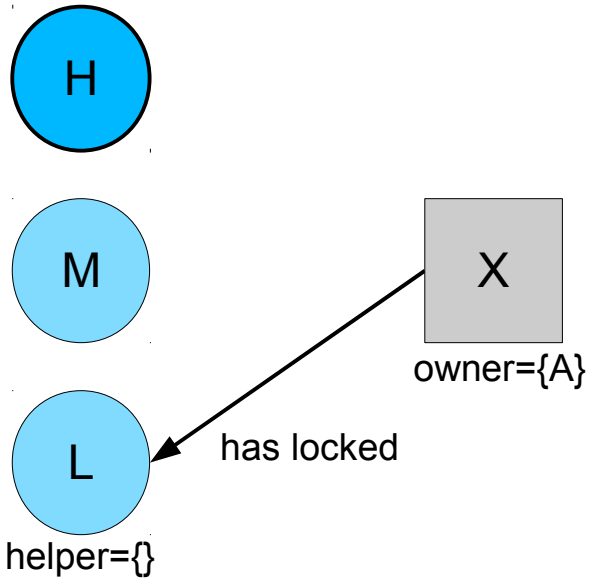
1. L acquires lock X
2. **M preempts L**
3. H preempts M
4. H wants lock X and helps lock owner L
5. L releases lock X and switches to helper H which acquires lock X
6. H releases lock X

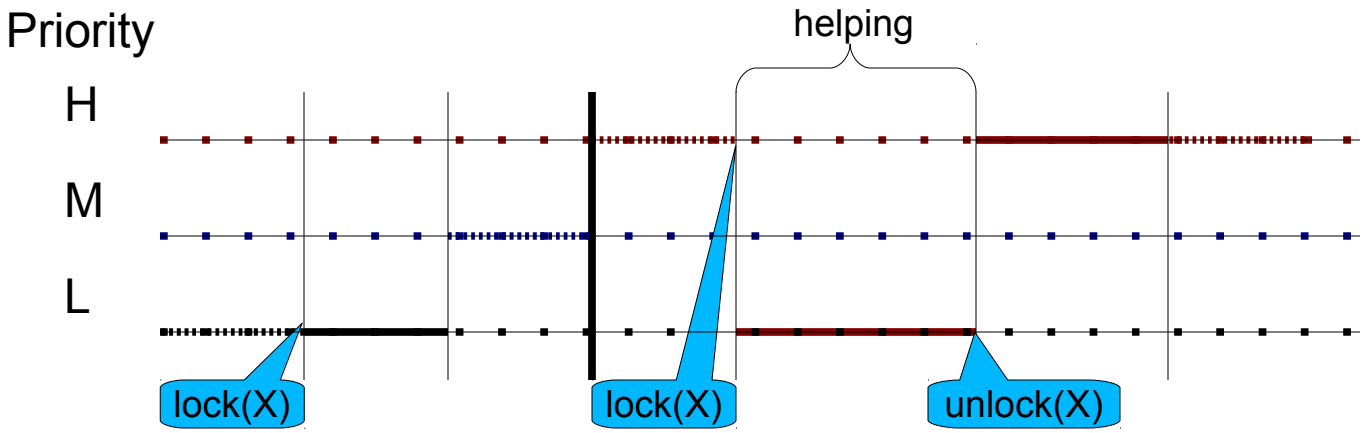




Basic Helping (3)

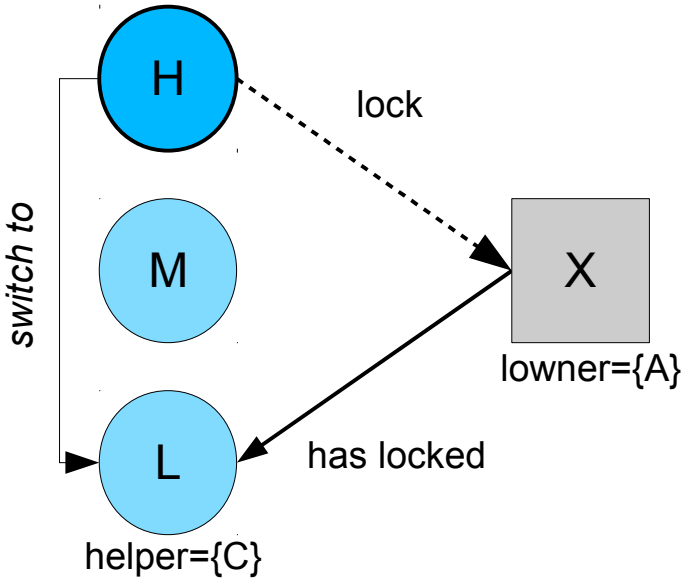
1. L acquires lock X
2. M preempts L
3. **H preempts M**
4. H wants lock X and helps lock owner L
5. L releases lock X and switches to helper H which acquires lock X
6. H releases lock X

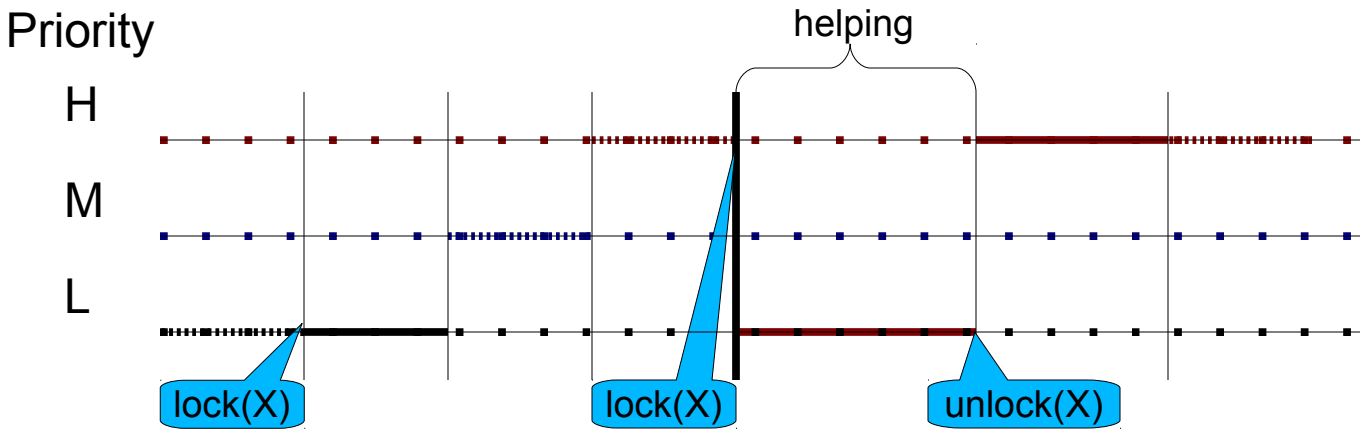




Basic Helping (4)

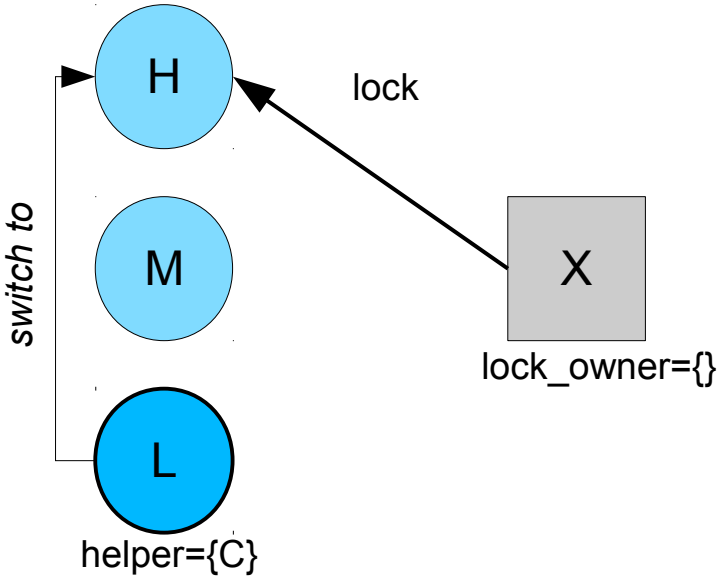
1. L acquires lock X
2. M preempts L
3. H preempts M
4. **H wants lock X and helps lock owner L**
5. L releases lock X and switches to helper H which acquires lock X
6. H releases lock X

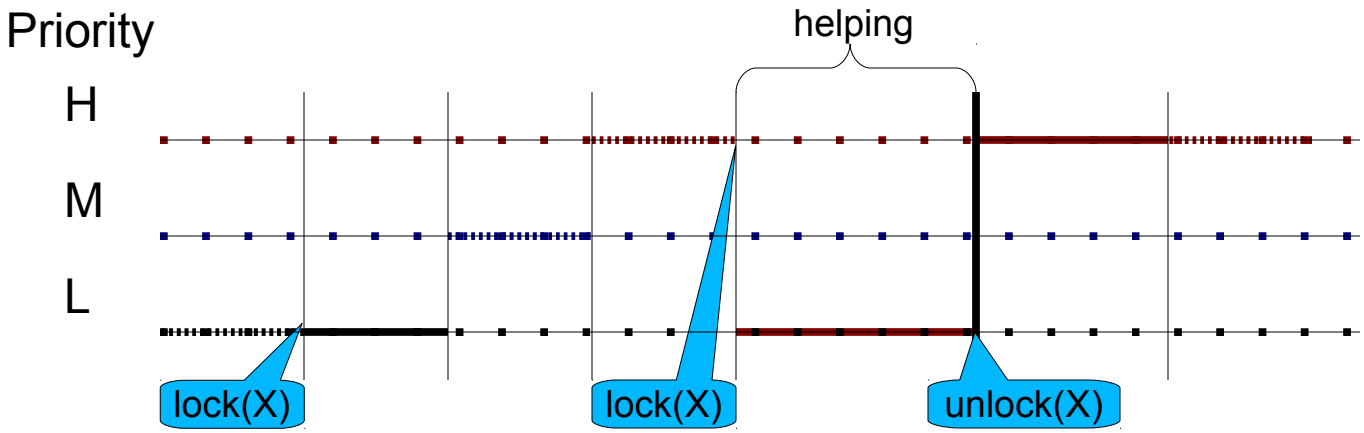




Basic Helping (5)

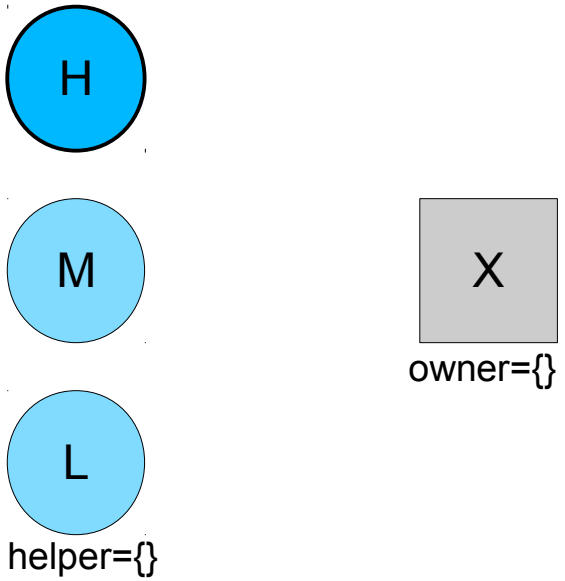
1. L acquires lock X
2. M preempts L
3. H preempts M
4. H wants lock X and helps lock owner L
5. **L releases lock X and switches to helper H which acquires lock X**
6. H releases lock X

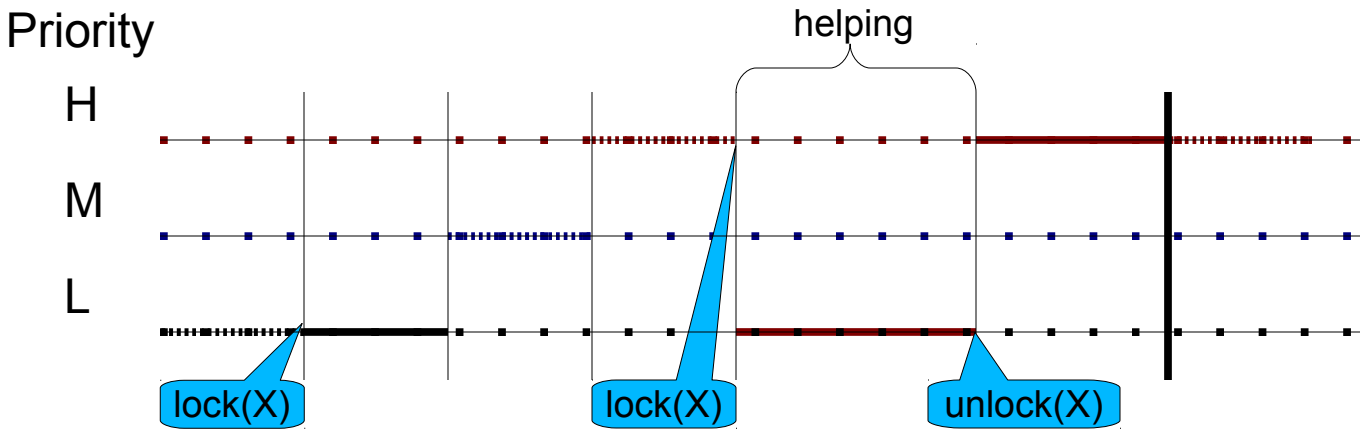




Basic Helping (6)

1. L acquires lock X
2. M preempts L
3. H preempts M
4. H wants lock X and helps lock owner L
5. L releases lock X and switches to helper H which acquires lock X
6. **H releases lock X**



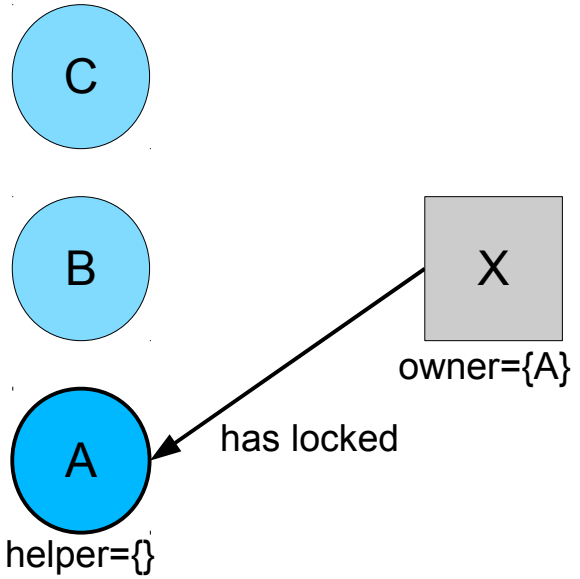


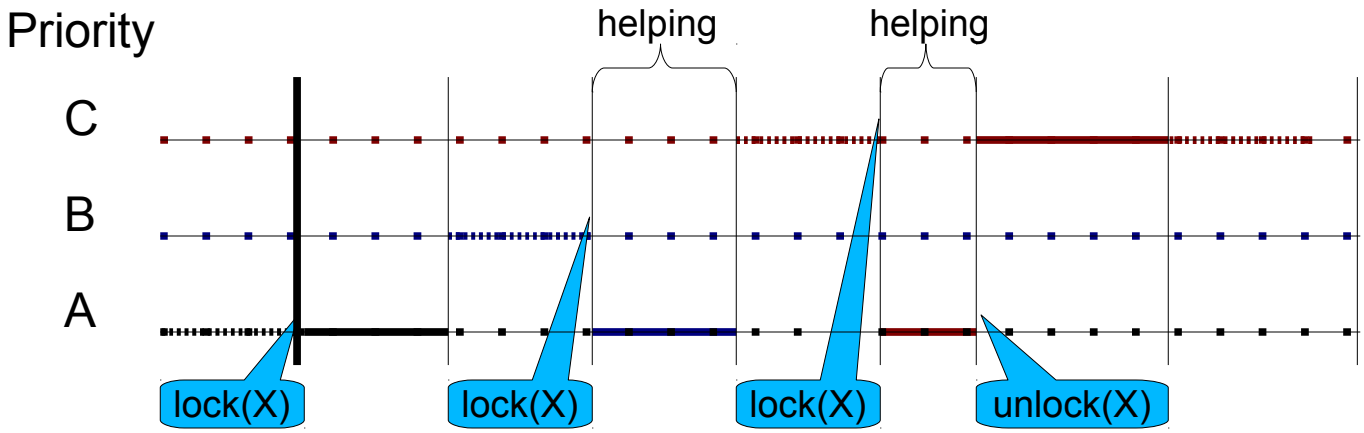
Torsten Frenzel

TU Dresden Operating Systems Group

Concurrent Helping (1)

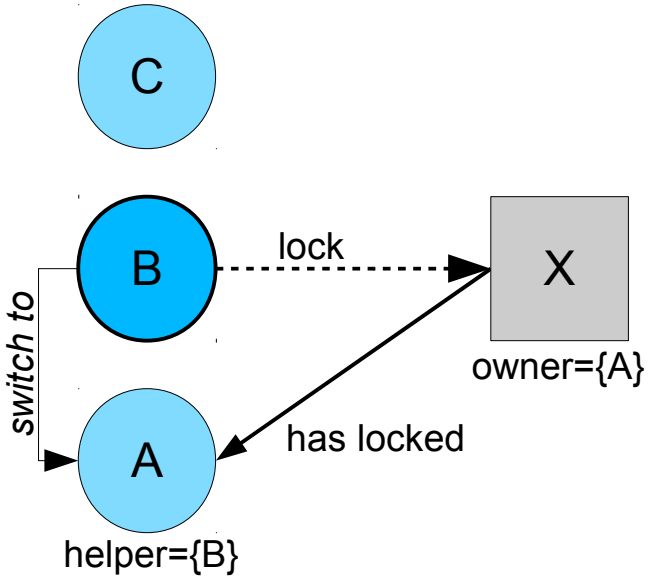
1. **A acquires lock X**
2. B wants lock X and helps A
3. C wants lock X and helps A
4. A releases lock X and switches to helper C
5. C releases lock X
6. Finally B can acquire lock X

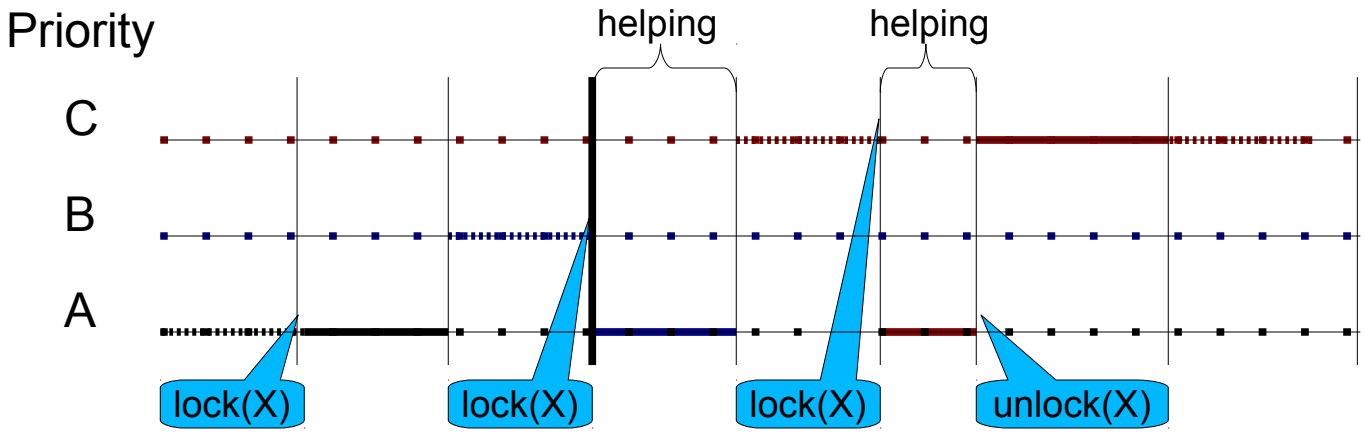




Concurrent Helping (2)

1. A acquires lock X
2. **B wants lock X and helps A**
3. C wants lock X and helps A
4. A releases lock X and switches to helper C
5. C releases lock X
6. Finally B can acquire lock X



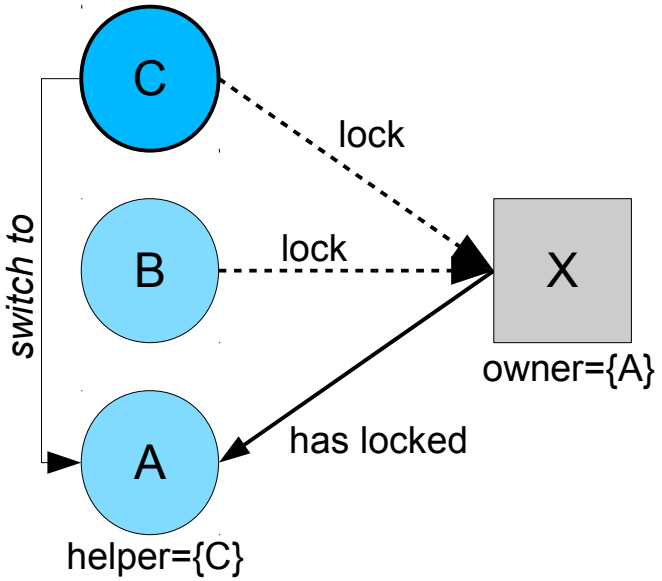


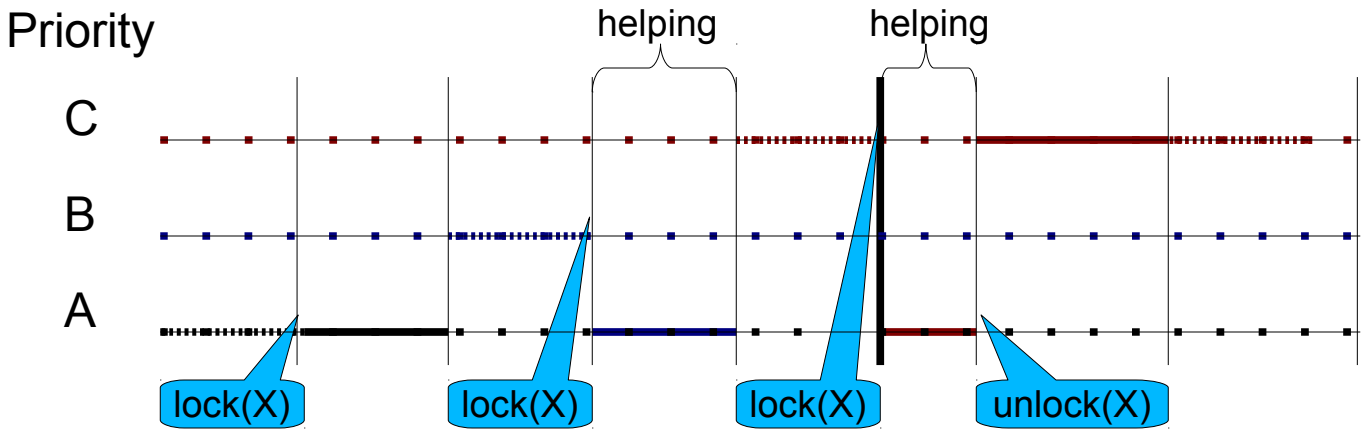
Torsten Frenzel

TU Dresden Operating Systems Group

Concurrent Helping (3)

1. A acquires lock X
2. B wants lock X and helps A
3. **C wants lock X and helps A**
4. A releases lock X and switches to helper C
5. C releases lock X
6. Finally B can acquire lock X



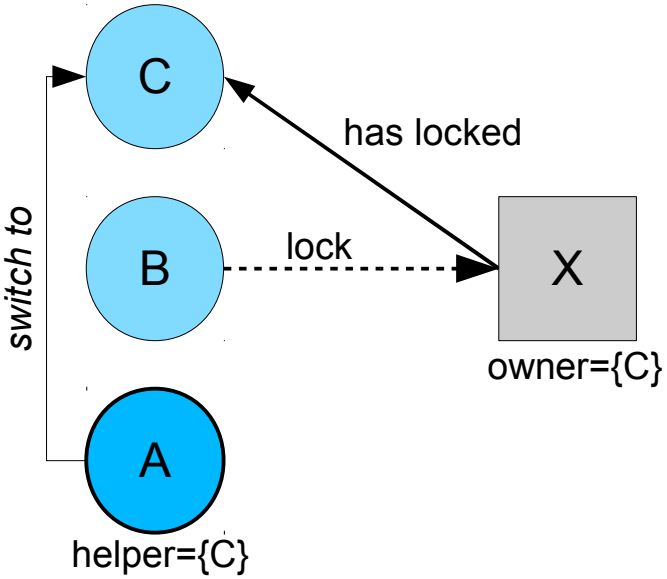


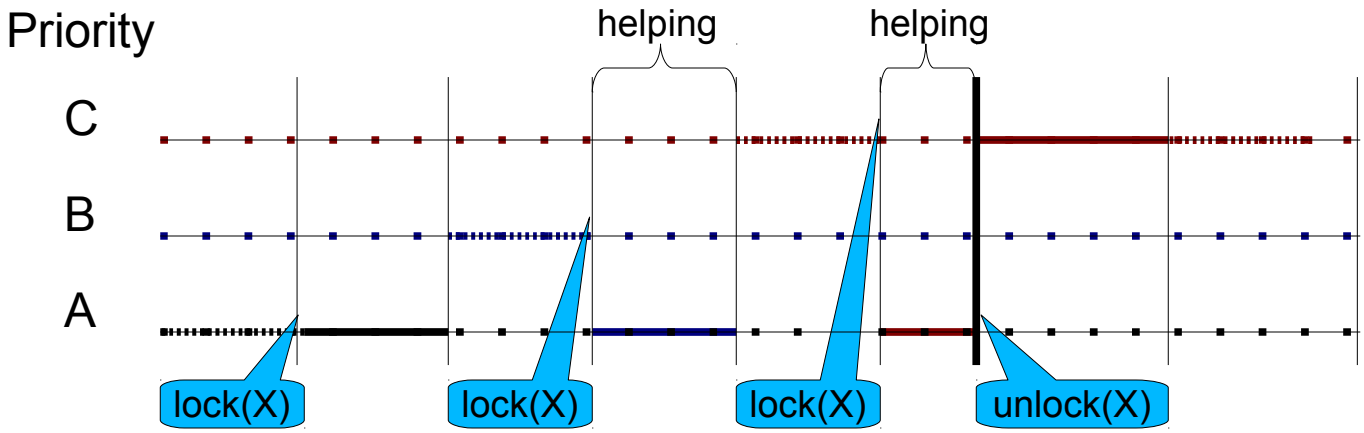
Torsten Frenzel

TU Dresden Operating Systems Group

Concurrent Helping (4)

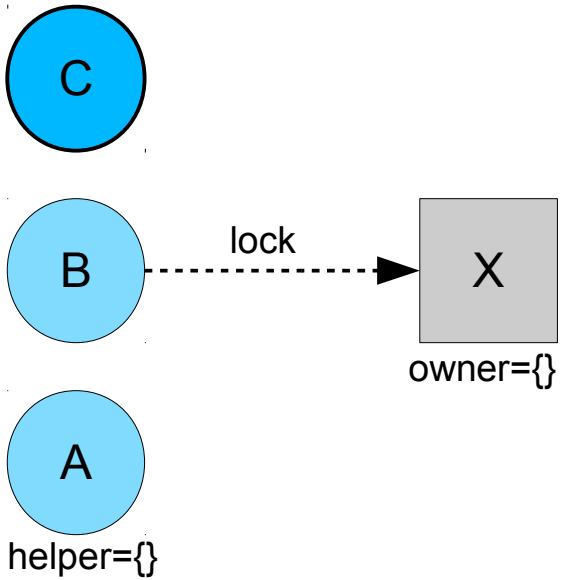
1. A acquires lock X
2. B wants lock X and helps A
3. C wants lock X and helps A
4. **A releases lock X and switches to helper C**
5. C releases lock X
6. Finally B can acquire lock X

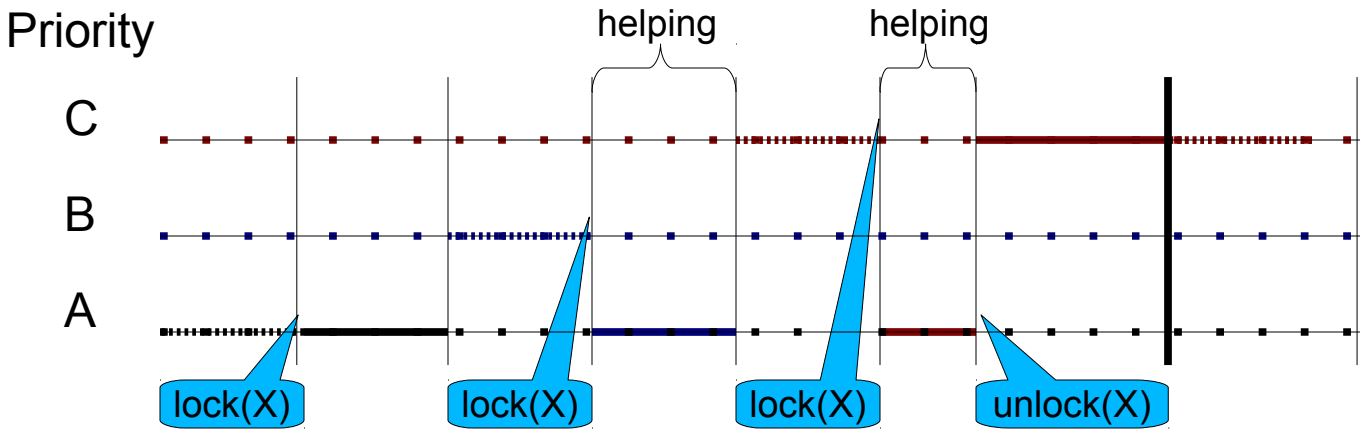




Concurrent Helping (5)

1. A acquires lock X
2. B wants lock X and helps A
3. C wants lock X and helps A
4. A releases lock X and switches to helper C
5. **C releases lock X**
6. Finally B can acquire lock X



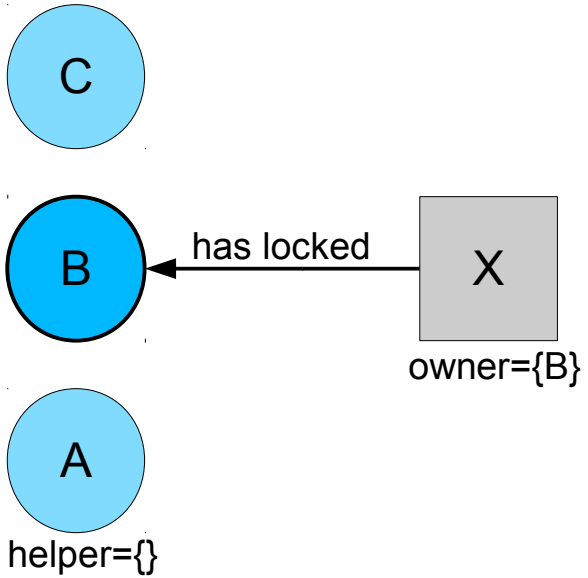


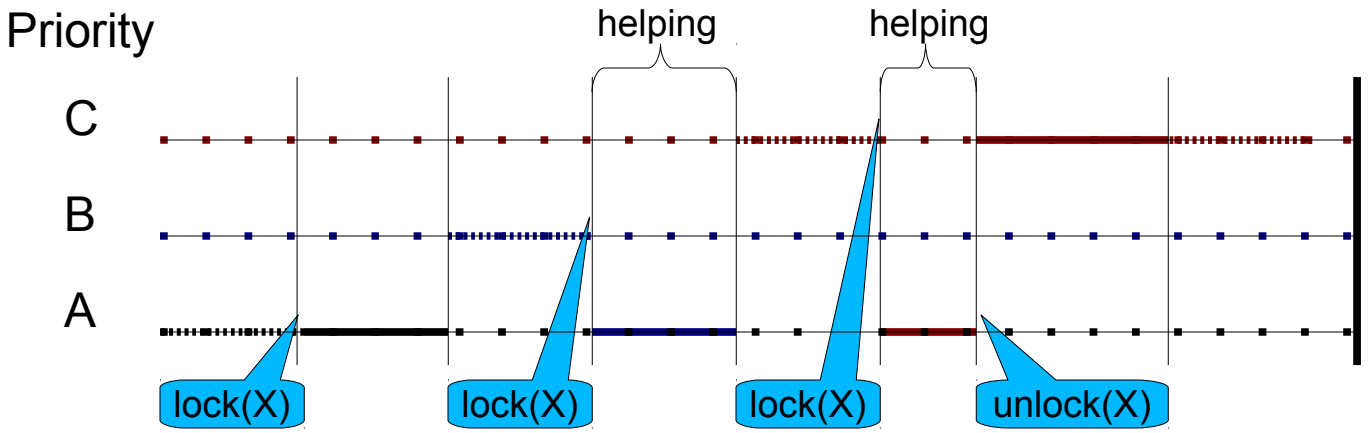
Torsten Frenzel

TU Dresden Operating Systems Group

Concurrent Helping (6)

1. A acquires lock X
2. B wants lock X and helps A
3. C wants lock X and helps A
4. A releases lock X and switches to helper C
5. C releases lock X
6. **Finally B can acquire lock X**



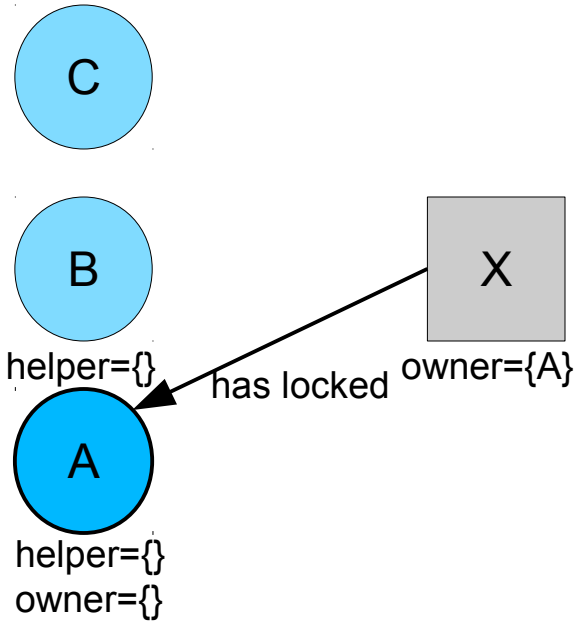


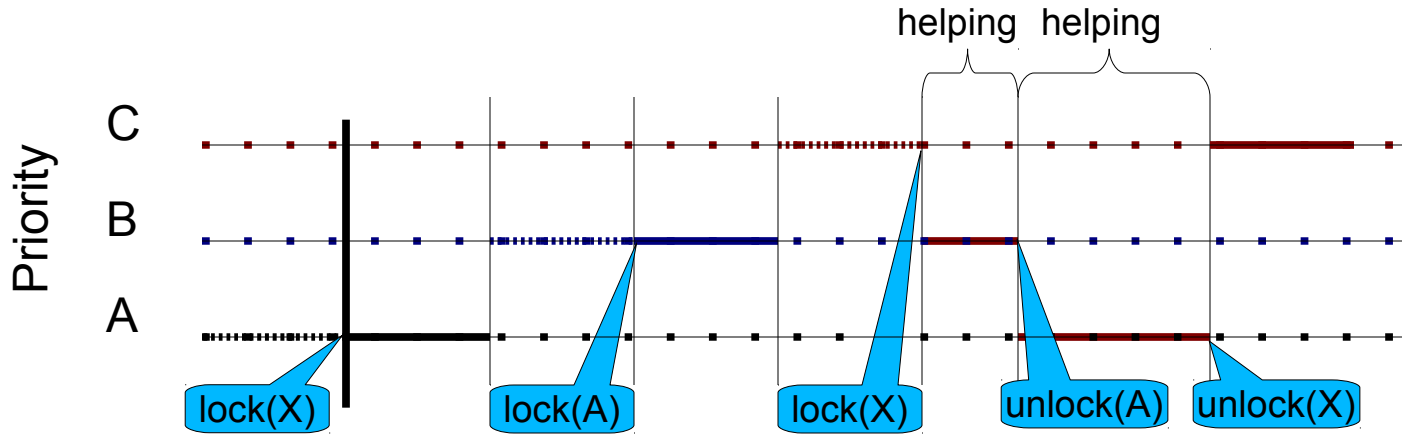
Torsten Frenzel

TU Dresden Operating Systems Group

Transitive Helping (1)

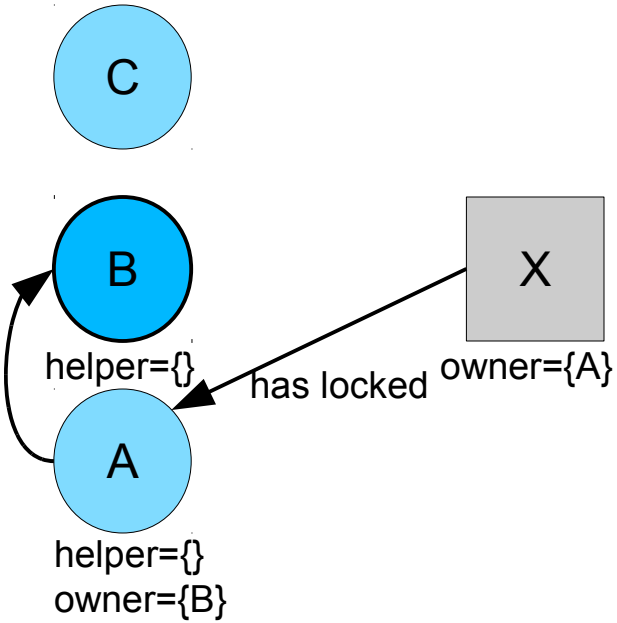
1. **A acquires lock X**
2. B acquires lock of A
3. C wants lock X and helps A
4. C helps B, because A is locked by B
5. B releases lock and switches to C
6. C again helps A
7. A releases lock and switches to C

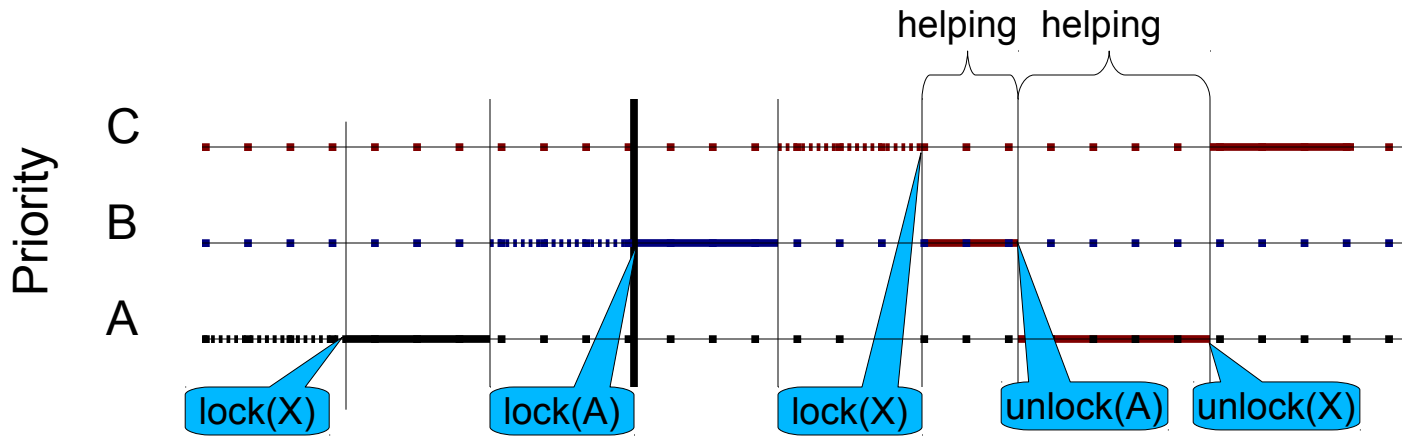




Transitive Helping (2)

1. A acquires lock X
2. **B acquires lock of A**
3. C wants lock X and helps A
4. C helps B , because A is locked by B
5. B releases lock and switches to C
6. C again helps A
7. A releases lock and switches to C



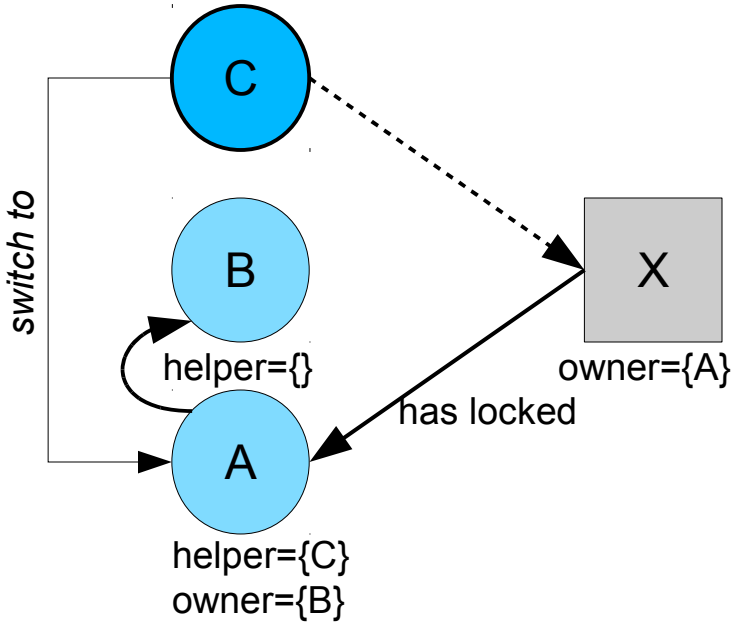


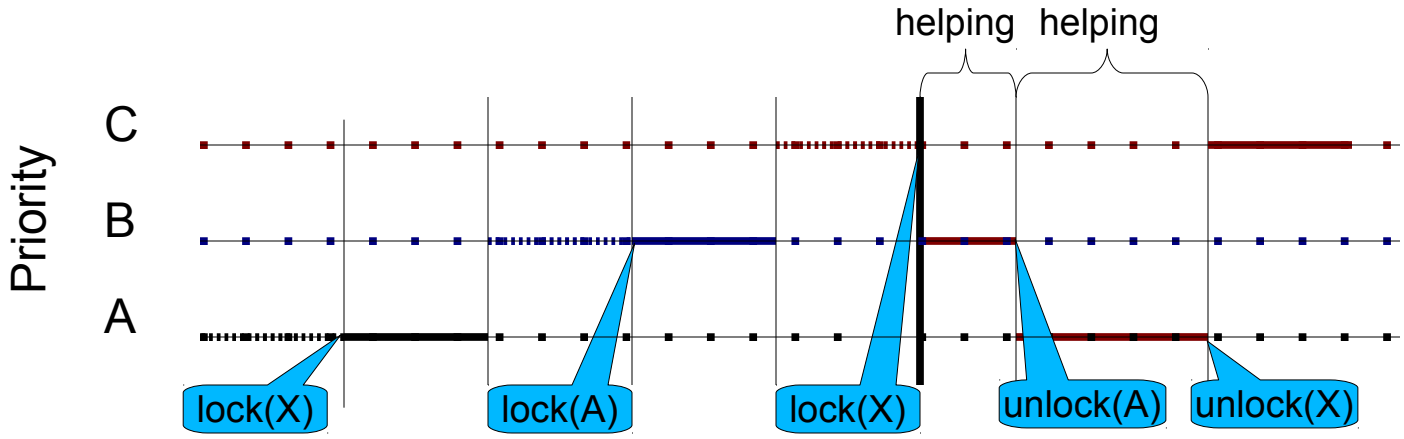
Torsten Frenzel

TU Dresden Operating Systems Group

Transitive Helping (3)

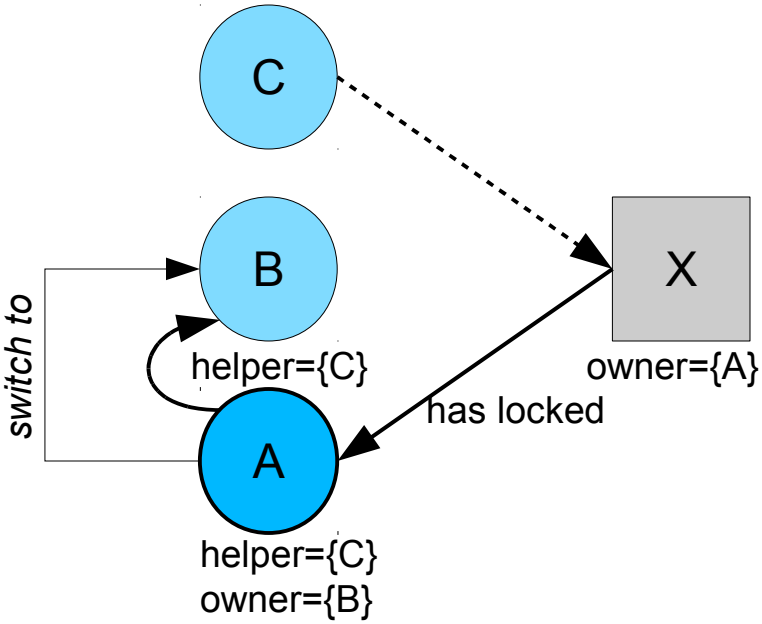
1. A acquires lock X
2. B acquires lock of A
3. **C wants lock X and helps A**
4. C helps B, because A is locked by B
5. B releases lock and switches to C
6. C again helps A
7. A releases lock and switches to C

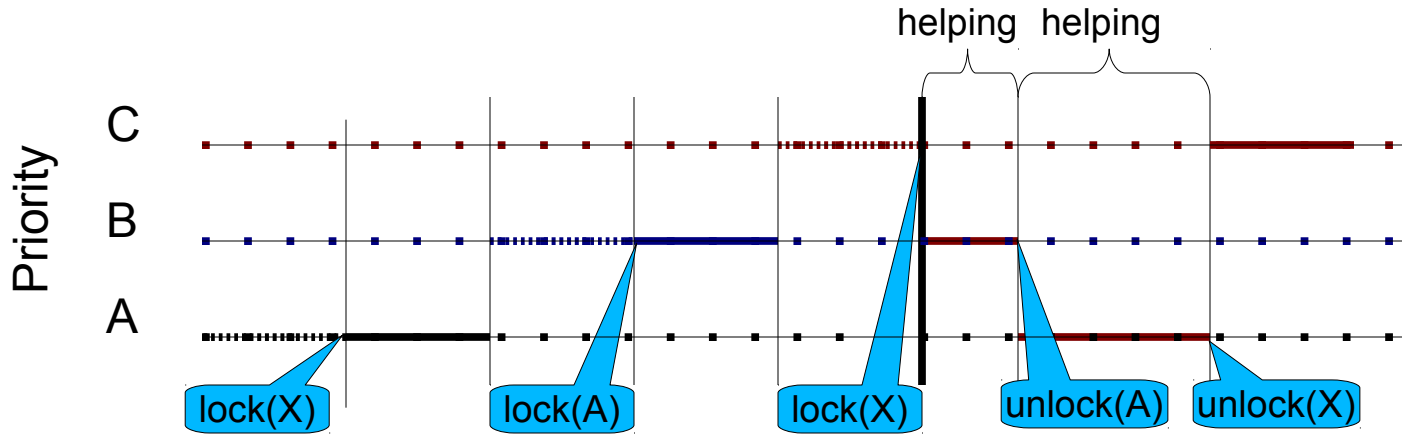




Transitive Helping (4)

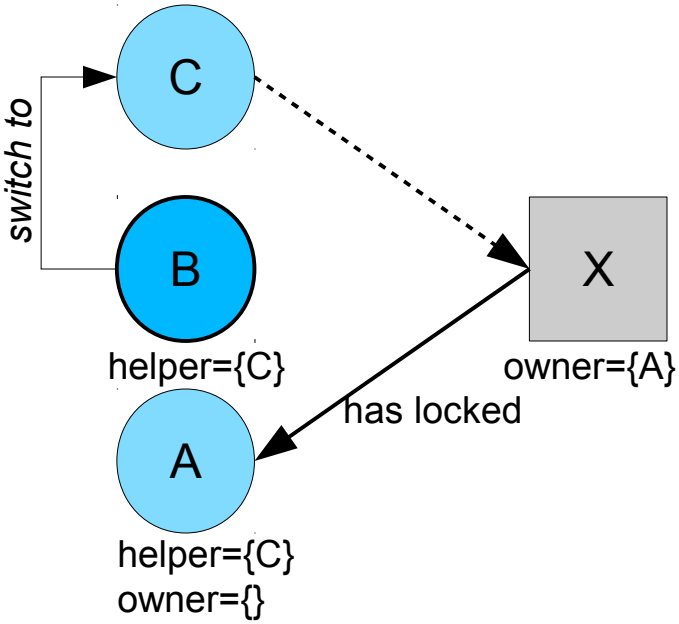
1. A acquires lock X
2. B acquires lock of A
3. C wants lock X and helps A
4. **C helps B, because A is locked by B**
5. B releases lock and switches to C
6. C again helps A
7. A releases lock and switches to C

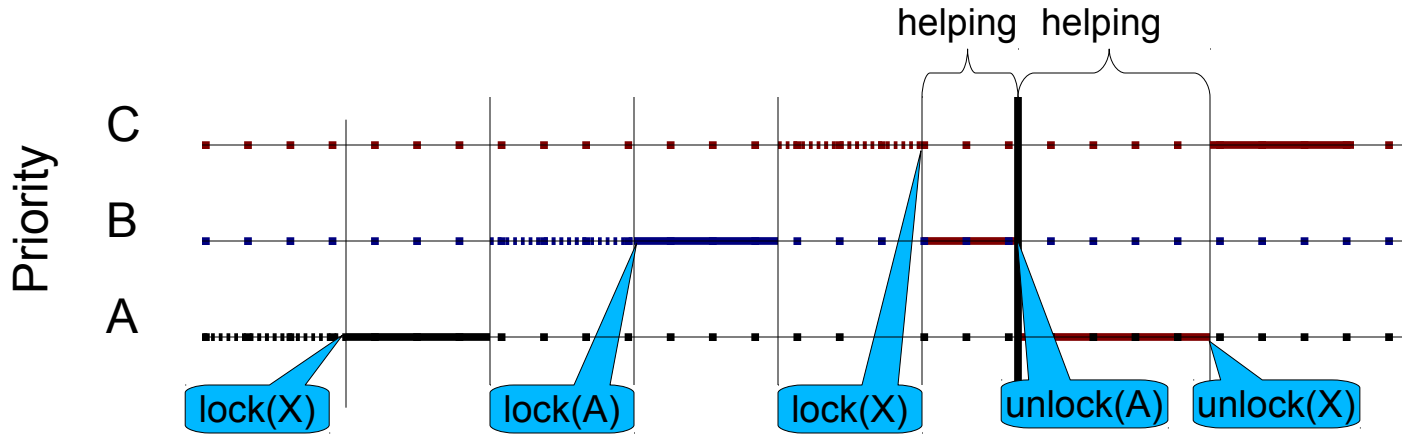




Transitive Helping (5)

1. A acquires lock X
2. B acquires lock of A
3. C wants lock X and helps A
4. C helps B, because A is locked by B
5. **B releases lock and switches to C**
6. C again helps A
7. A releases lock and switches to C



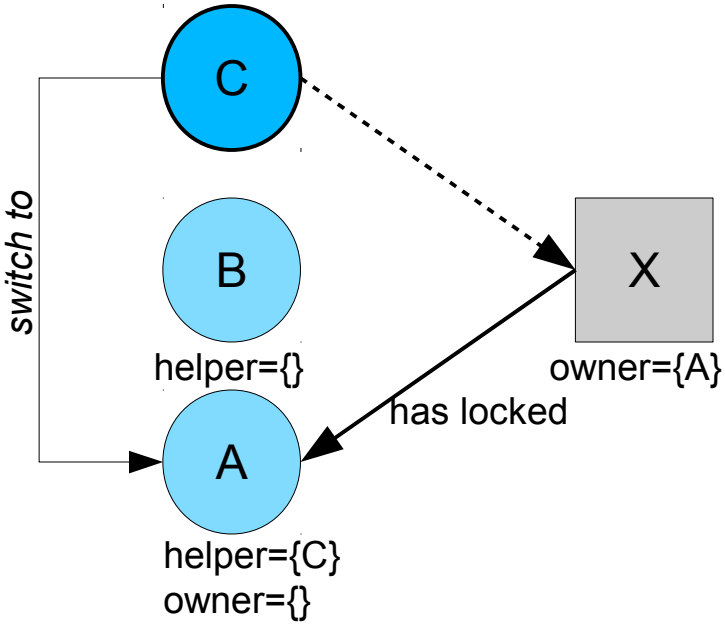


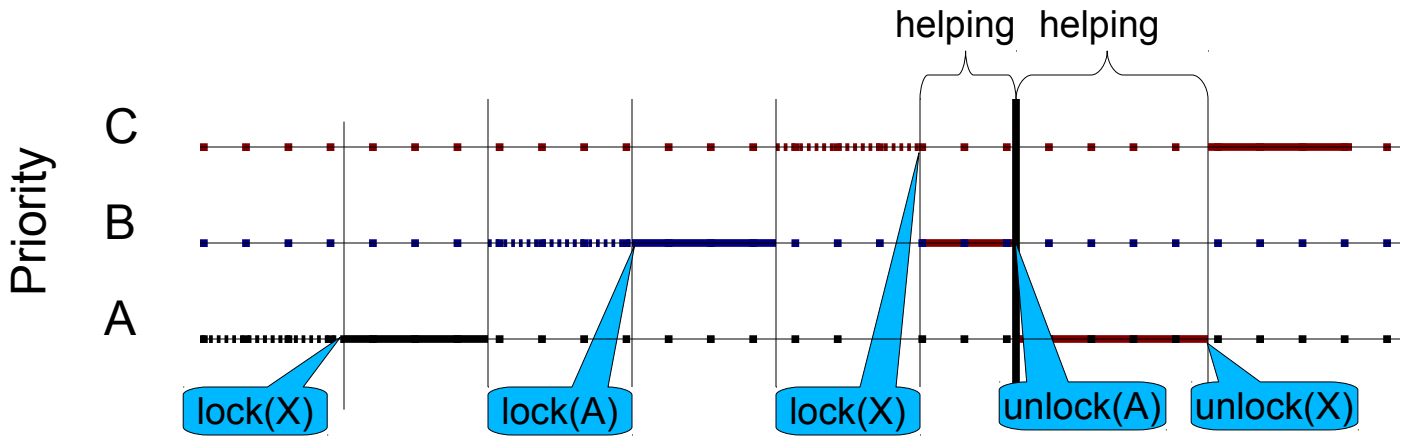
Torsten Frenzel

TU Dresden Operating Systems Group

Transitive Helping (6)

1. A acquires lock X
2. B acquires lock of A
3. C wants lock X and helps A
4. C helps B, because A is locked by B
5. B releases lock and switches to C
6. **C again helps A**
7. A releases lock and switches to C



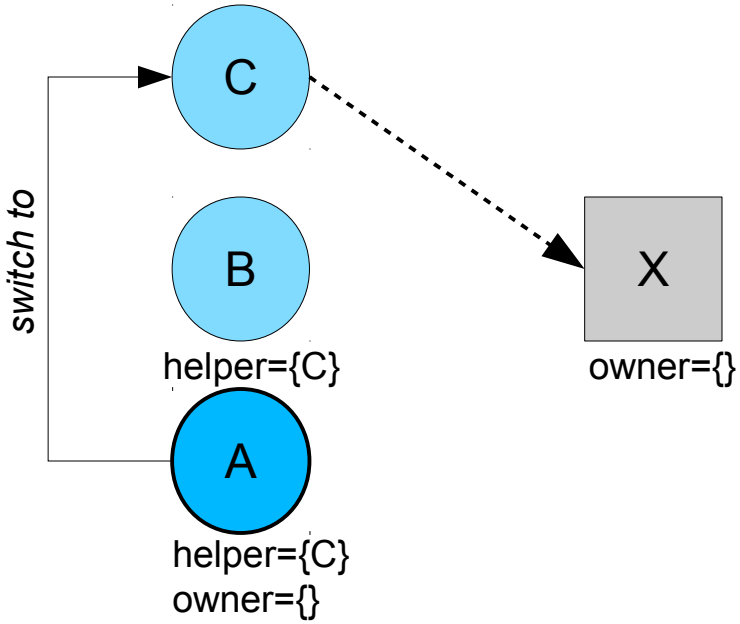


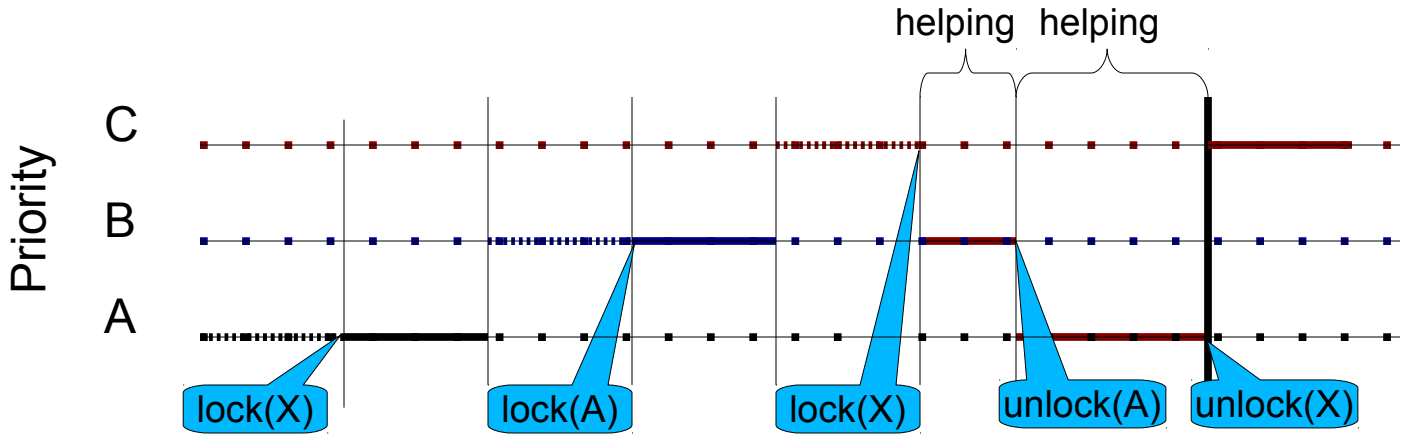
Torsten Frenzel

TU Dresden Operating Systems Group

Transitive Helping (7)

1. A acquires lock X
2. B acquires lock of A
3. C wants lock X and helps A
4. C helps B, because A is locked by B
5. B releases lock and switches to C
6. C again helps A
7. **A releases lock and switches to C**





Helping-Lock: Implementation

- Mechanism:
  - Helper lends time and priority to current lock owner
  - Switch to possible helper after lock release
- Lock acquire
  1. As long as lock not free:
     - Switch to current lock owner
  2. If lock is free become new lock owner
- Lock release
  1. Clear lock owner
  2. If previous lock owner (current thread) has helper switch to helper

Thread-Lock: Implementation

- Special implementation of helping lock
- Requires modification in the scheduling mechanism:
  - Locked threads have to be prevented from running
  - Scheduler switches to lock owner instead (Helping!)
- **Lock acquire**
  - Acquire switch lock
- **Lock release**
  - Release switch lock
  - If previous lock owner has helper switch to helper
  - Switch to the higher priority thread:
    - Previous locked thread
    - Previous lock owner

Torsten Frenzel

TU Dresden Operating Systems Group

Locking and Killing Threads

- Threads need to be locked before TCB is accessed
- Threads need to be locked for deletion
- Problem:
  1. Thread A acquires thread lock of target thread T
  2. Thread B requests also thread lock of target thread T
  3. Thread A deletes target thread T
  4. Thread lock of T becomes invalid (memory deallocated)
- Problems:
  - Thread B may block forever
  - Thread B may access invalid thread lock
- Solution:
  - Check validity of thread state before lock request
  - Wait until all references to TCB have vanished
    - *This is a non-trivial problem for another lesson!*

Torsten Frenzel

TU Dresden Operating Systems Group

Summary

- Thead creation
  - Implicitly on demand
  - Explicitly to allow resource accounting
- Thread destruction
  - Free all resources (release locks)
  - Invalidate all referencences (dequeue from lists, abort IPC operation)
  - Finally free memory of thread
- Synchronization
  - Lock-free synchronization
    - Complex even for simple data structures
  - Wait-free synchronization
    - Avoids priority inversion but not multi-processor safe