

# Microkernel Construction

---

## Synchronization

SS2014

# 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

# 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

# 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

# Disabling Interrupts

- Easy to implement but not multi-processor safe
  - For CPU-local data structures (stack)
  - For kernel entry/exit code
- 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

# Blocking / Waiting 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 critical sections
  - ▶ Burns processing time by idling

# Spinlocks: Overview

- Test and set lock
  - Try to acquire lock, if it fails spin (writing)
- Test and test and set lock
  - Try to acquire the lock only if it is free else spin (reading)
  - 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 enqueue in a list and spin on a local variable
  - Minimizes overhead and guarantees fairness

See lecture: *Distributed Operating Systems* for details

# 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)



# Lock-Free Synchronization

- Principle:
  - Prepare data out of line
  - Atomically try to exchange old data with new one
  - If it fails, retry
- Read Copy Update (RCU)
  - Reader/writer separation, readers can concurrently access data
  - Writers modify data offline and atomically update
- Properties
  - Requires atomic memory operations (xchg, cmpxchg)
  - No lock → no deadlock
    - Crashed threads hold no locks, don't have to release them
    - Trivially multiprocessor-safe
  - Bounded number of retries [Anderson]
  - Hard to implement correctly for complex data structures

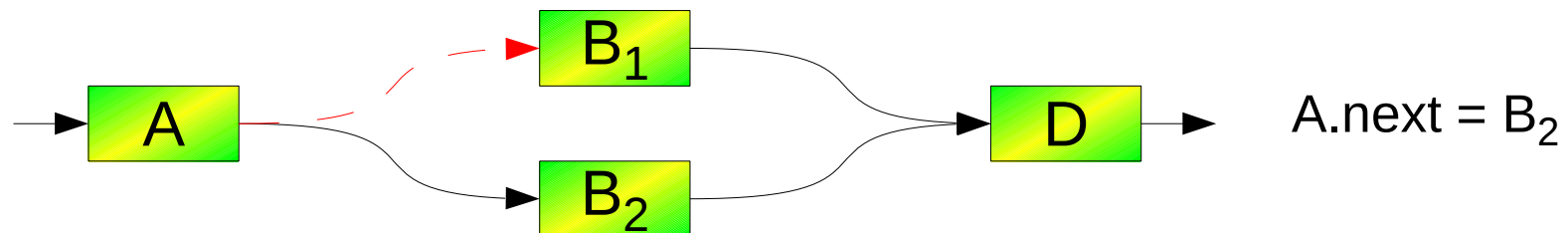
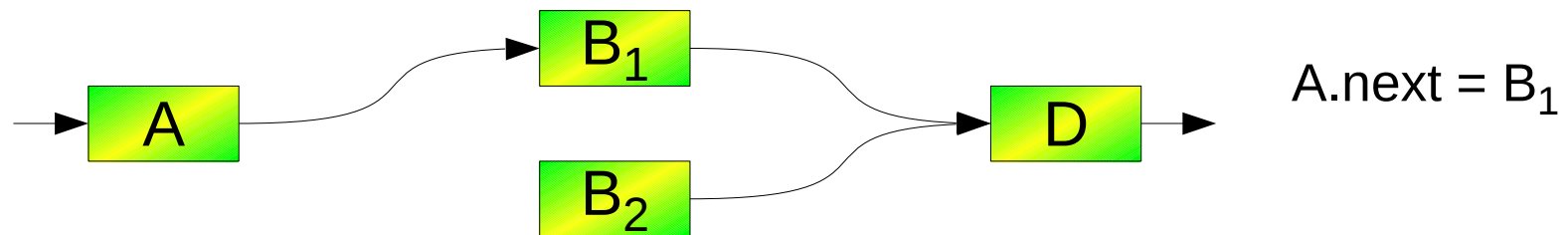
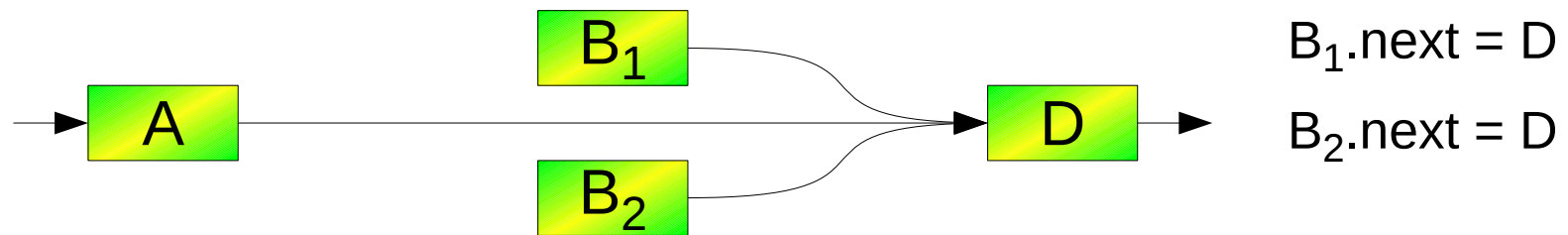
# Atomic Memory Operations

- 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
```

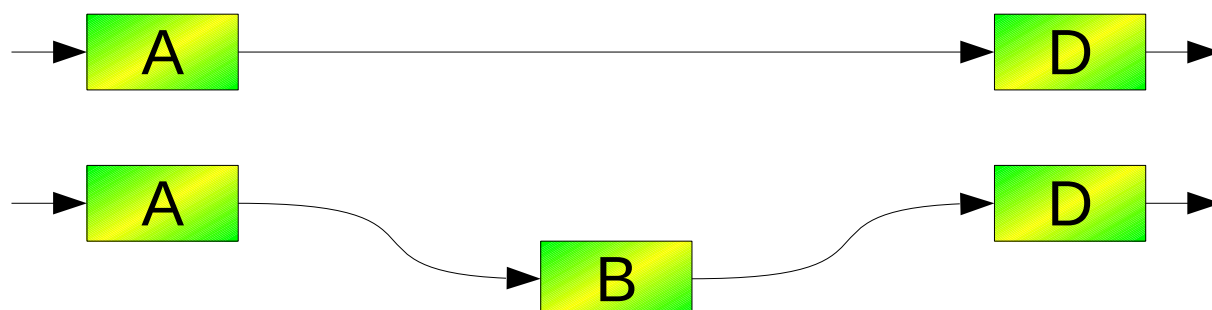
# Single Linked List

- Concurrent insert without atomic operations



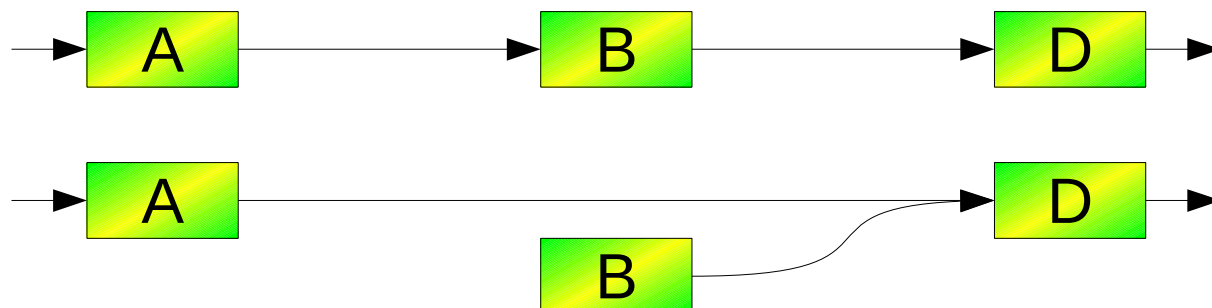
# Single Linked List

## ■ Insert with atomic operations



```
B.next = D;
tmp = A.next;
cas (A.next, tmp, B);
```

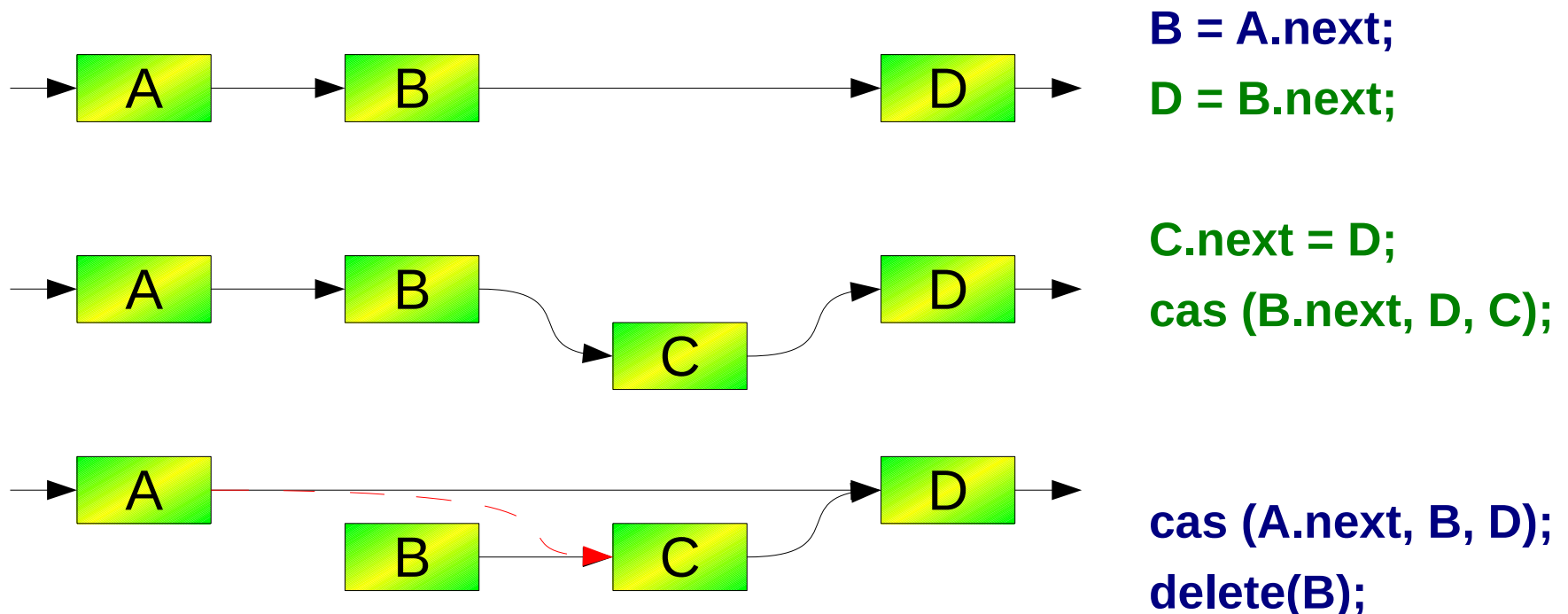
## ■ Delete with atomic operations



```
tmp = B.next;
cas (A.next, B, tmp);
delete (B);
```

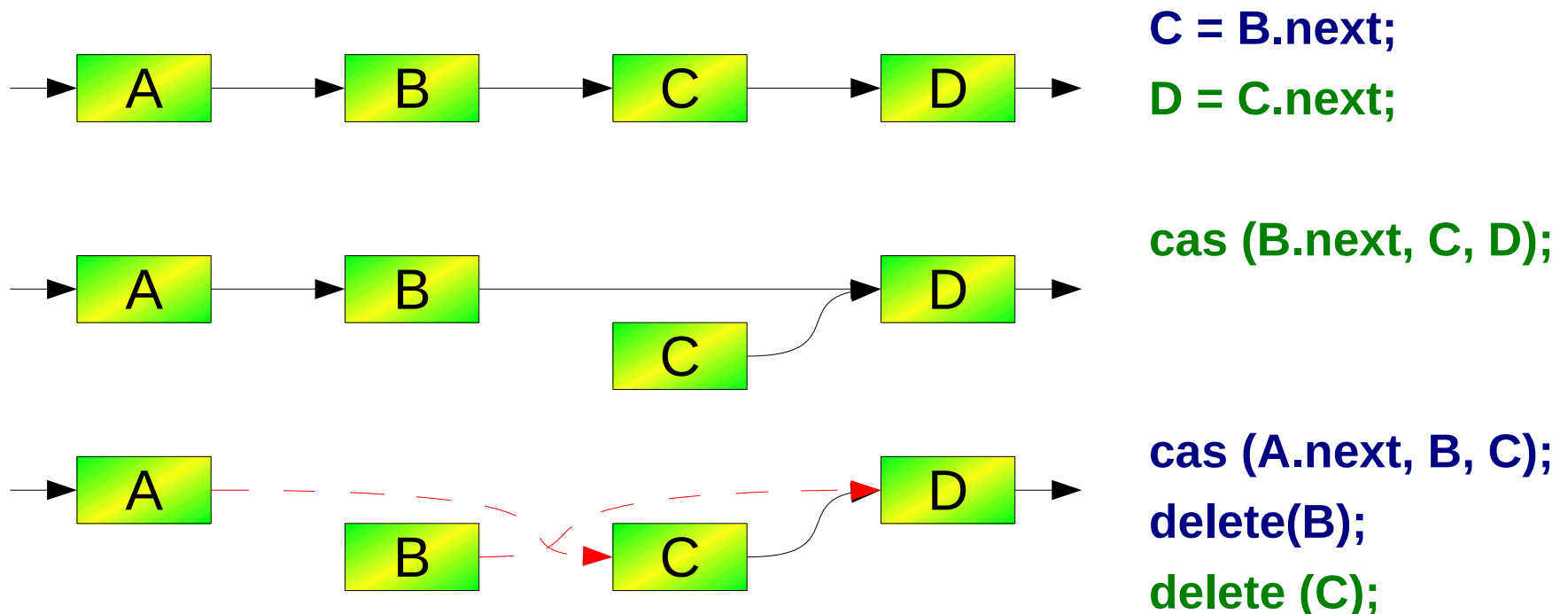
# Single Linked List

- Concurrent insert and delete operations
  - Delete element B and insert element C
  - Operation **delete(B)** changes pointer A.next from B to D
  - Operation **insert(C)** changes pointer B.next from D to C
  - Inconsistency: insert C might get lost, memory leak



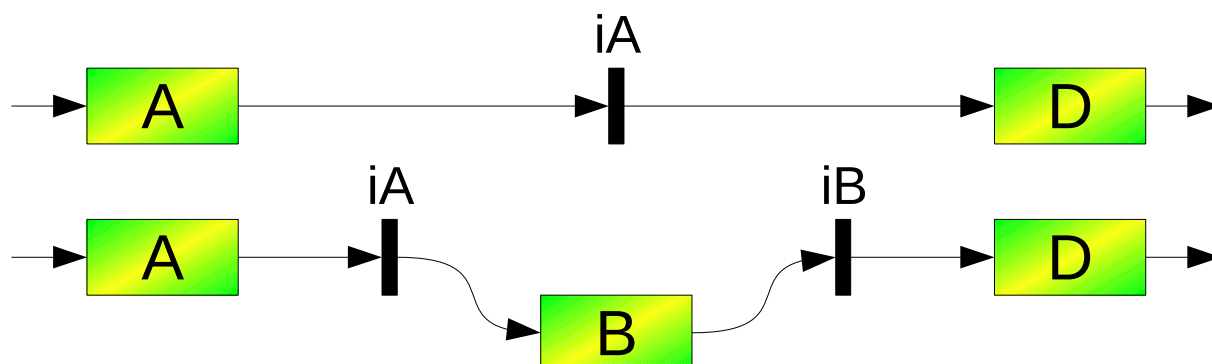
# Single Linked List

- Concurrent delete operations
  - Delete element 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
  - Broken list, dangling pointer (A.next)



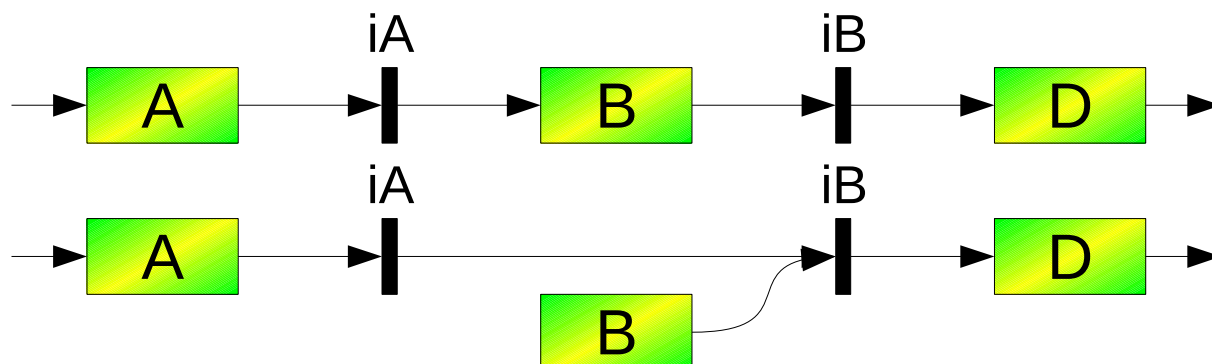
# Single Linked List

- Introduce intermediate nodes
- Insert operation:



```
B.next = iB;
tmp = iA.next;
cas (iA.next, tmp, B);
```

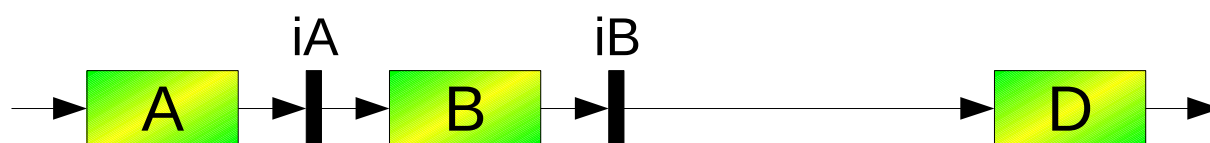
- Delete operation:



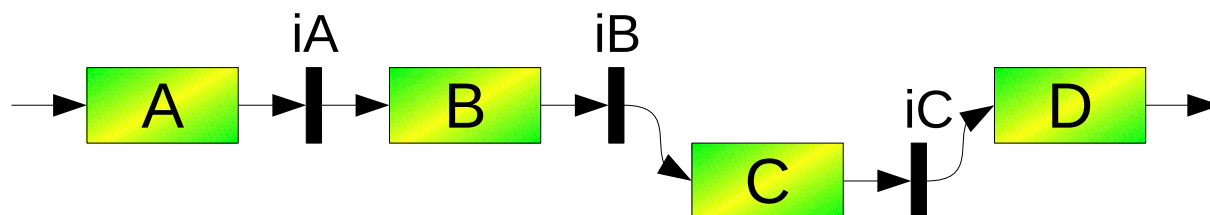
```
tmp = B.next;
cas (Ai.next, B, tmp);
delete (B);
```

# Single Linked List

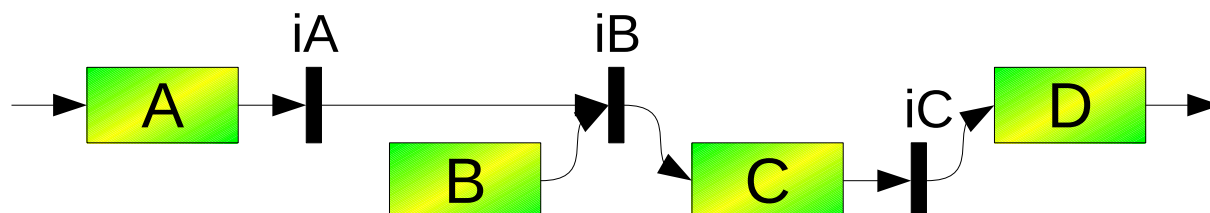
- Concurrent insert and delete operations
  - Delete element B and insert element C
  - Operation **delete(B)** changes pointer `iA.next` from B to `iB`
  - Operation **insert(C)** changes pointer `iB.next` from D to C



**`iB = B.next;`**  
**`D = iB.next;`**



**`C.next = iC;`**  
**`cas (iB.next, D, C);`**

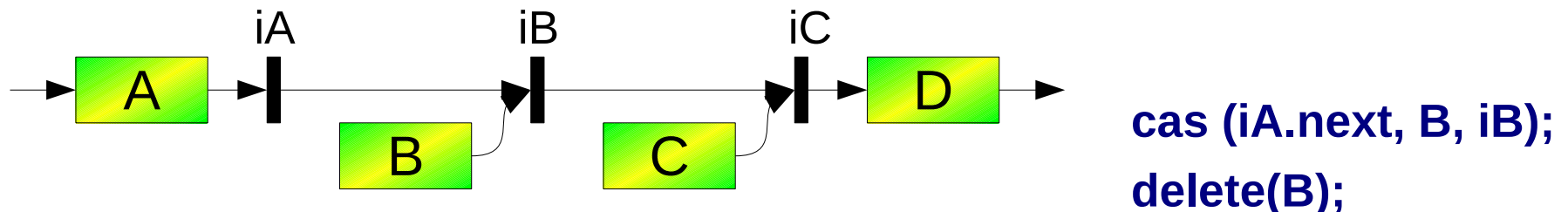
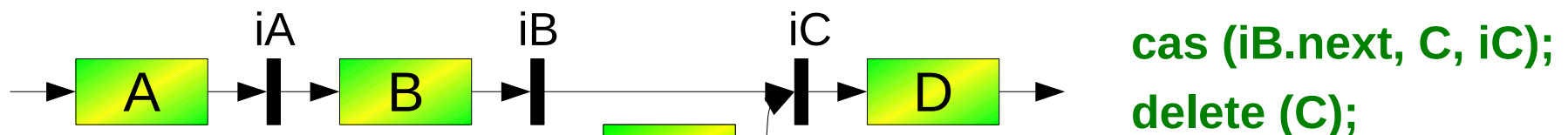
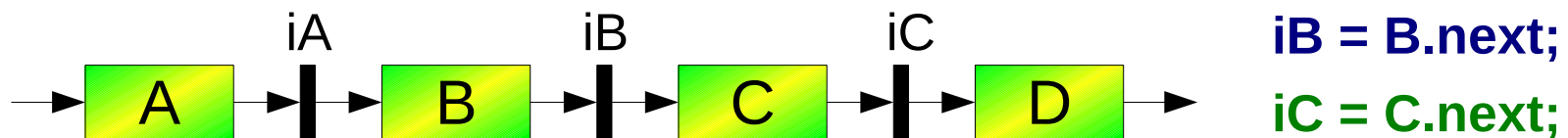


**`cas (iA.next, B, iB);`**  
**`delete(B);`**



# Single Linked List

- Concurrent delete operations
  - Delete element B and C
  - Operation **delete(B)** changes pointer `iA.next` from B to `iB`
  - Operation **delete(C)** changes pointer `iB.next` from C to `iC`



# ABA Problem

Thread 1	Thread 2
<code>tmp = read (mem)</code>	
	<code>write (mem, B);</code>
	<code>write (mem, A);</code>
<code>if (tmp == read (mem)) ...</code>	

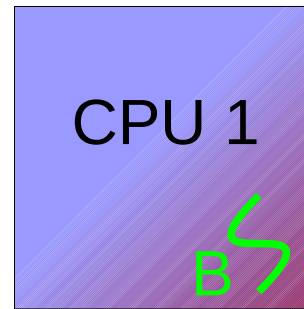
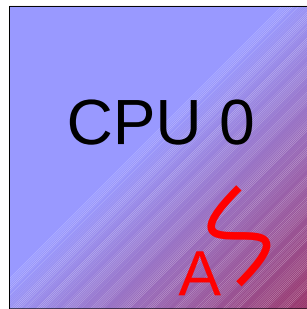
- Problem:
  - Reading a value twice being preemptable in between cannot detect changes that restore the old value
- Example:
  - Thread T1 reads value A
  - Thread T1 is preempted and Thread T2 runs
  - Thread T2 modifies the value A to value B and back to A
  - Thread T1 begins execution again, sees that the value has not changed and continues

# ABA Problem: Solutions

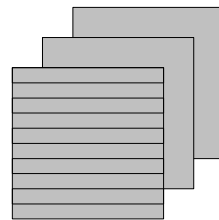
- Double-CAS (DCAS)
  - Add an additional tag value that is incremented on every update of the pointer
  - Double-CAS is used to update both values atomically
- Load-Locked/Store-Conditional (LL/SC)
  - Load-Locked operation detects changes of the pointer after the read operation
  - Store-Conditional only succeeds if the pointer has not changed since the load operation
- Transactional memory
  - Defines transactions on arbitrary memory access sequences

# Synchronizing Page Table Updates

- Address space comprising multiple CPUs
- Share page tables between them
- Modifying page tables requires synchronization

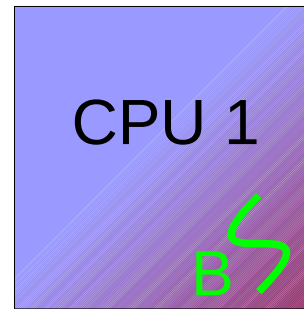
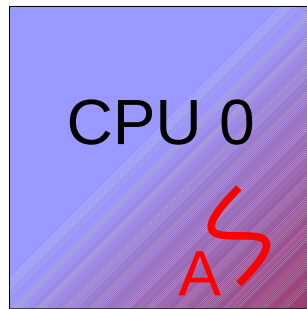


- 2 threads on 2 CPUs in one address space

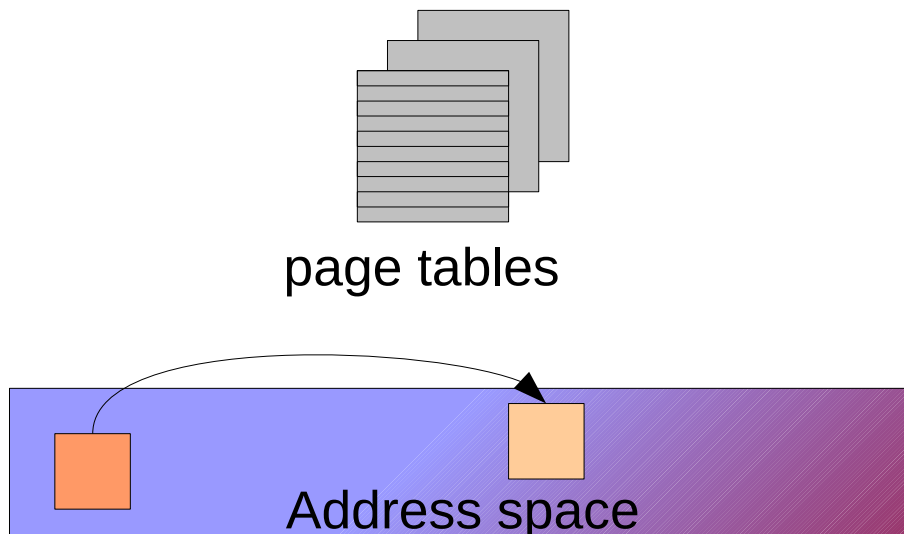


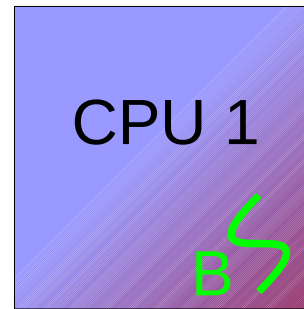
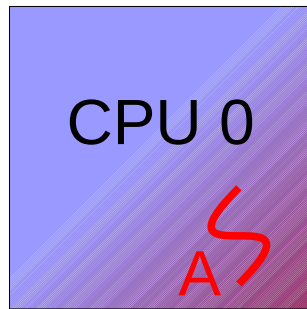
page tables



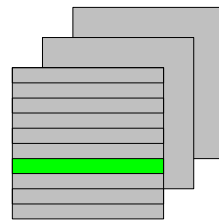


- 2 threads on 2 CPUs in one address space
- Thread A maps a page within its address space

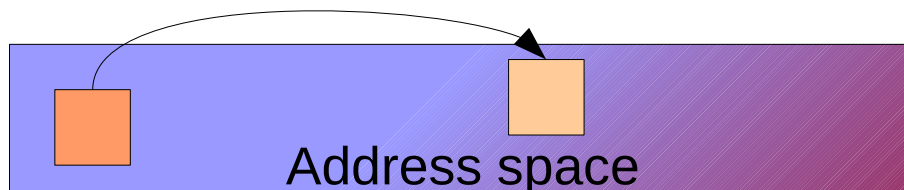


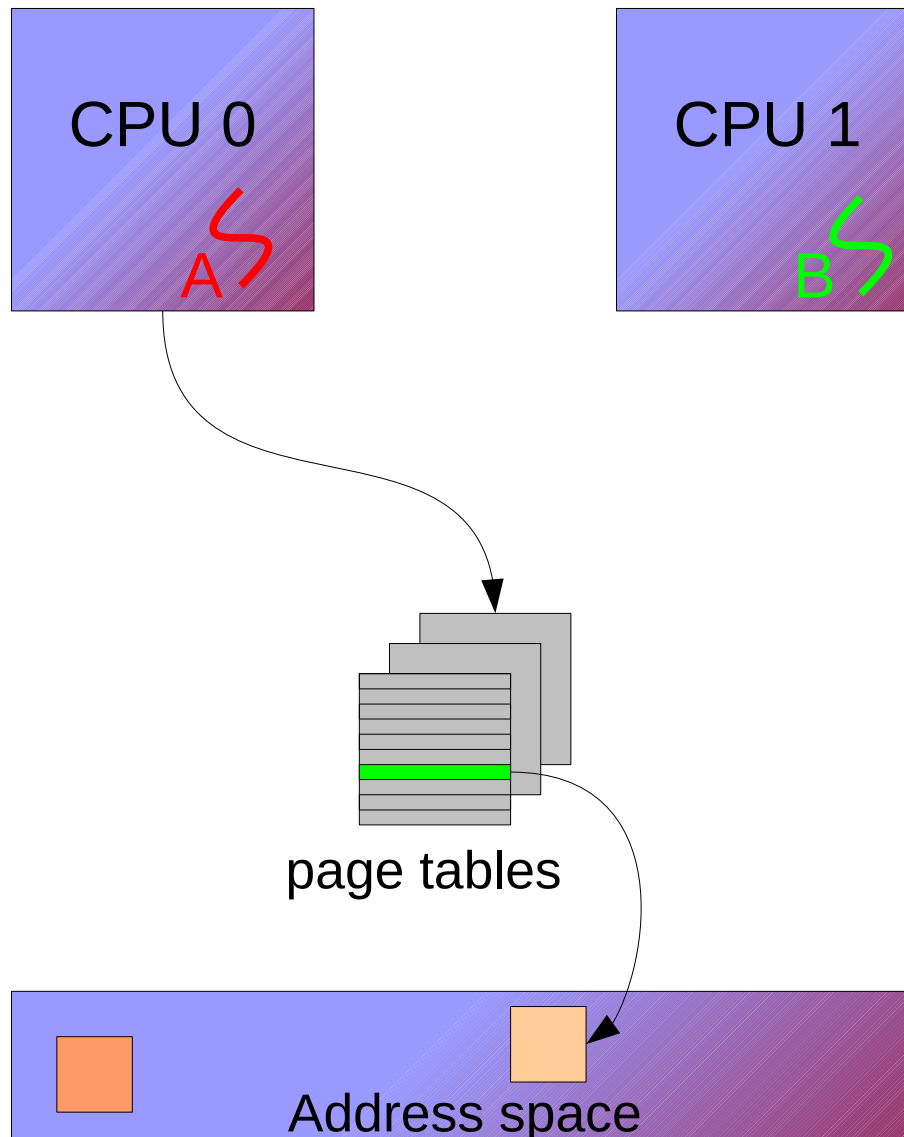


- 2 threads on 2 CPUs in one address space
- Thread A maps a page within its address space
- Thus modifies the page table to add new page



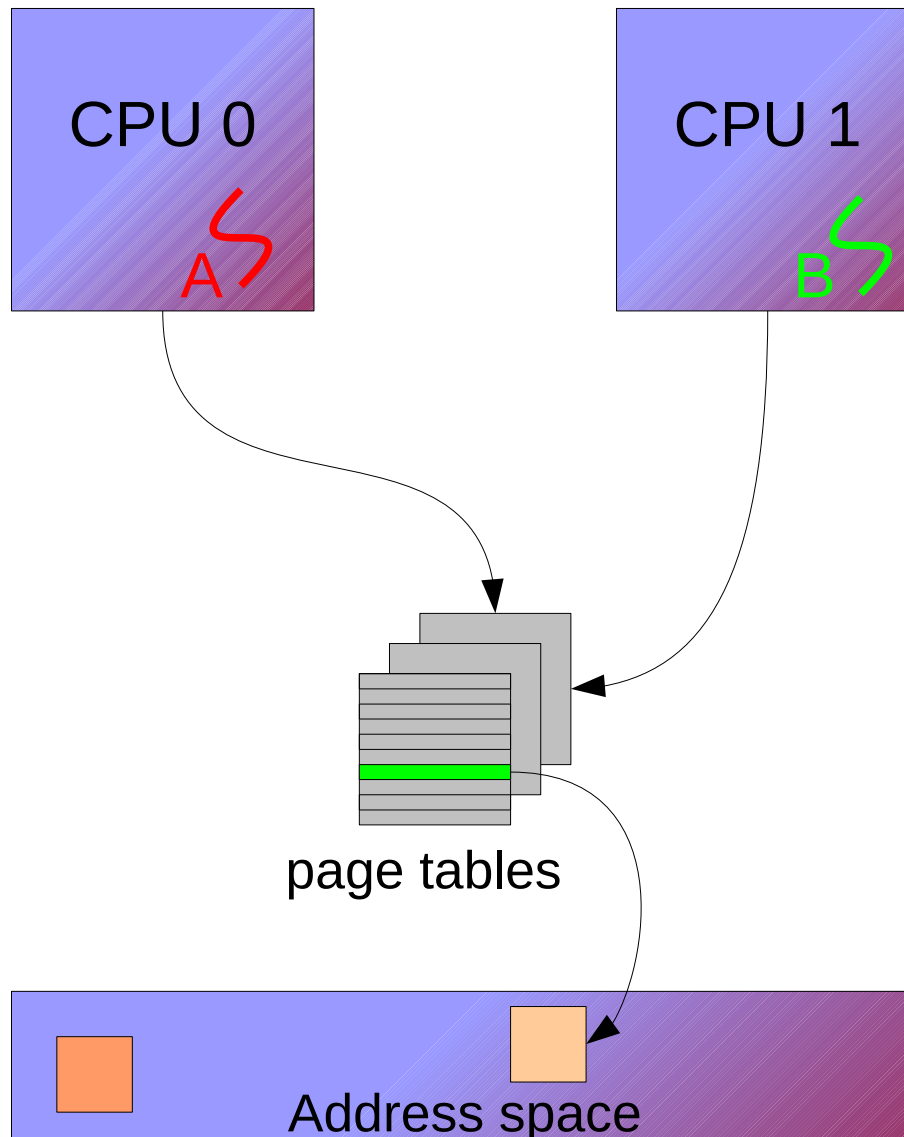
page tables



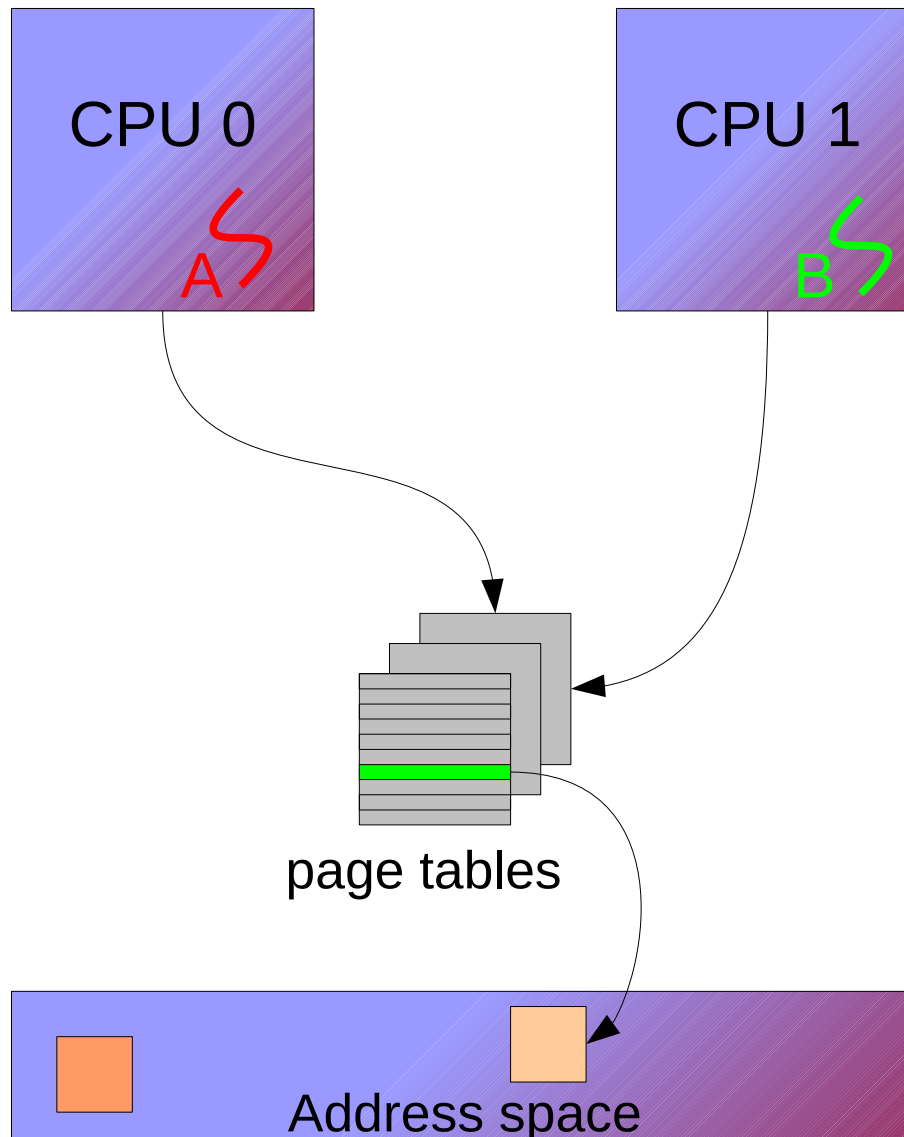


- 2 threads on 2 CPUs in one address space
- Thread A maps a page within its address space
- Thus modifies the page table to add new page
- Thread A can access newly mapped page



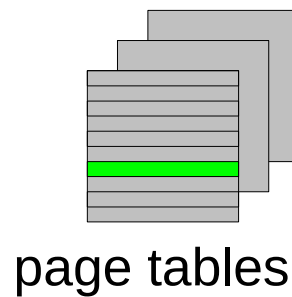
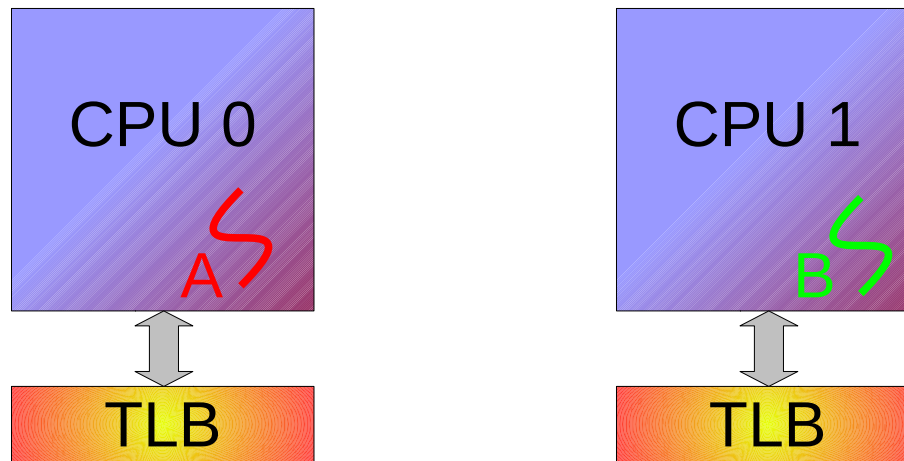


- 2 threads on 2 CPUs in one address space
- Thread A maps a page within its address space
- Thus modifies the page table to add new page
- Thread A can access newly mapped page
- Thread A and B share same address space, so B can access page too

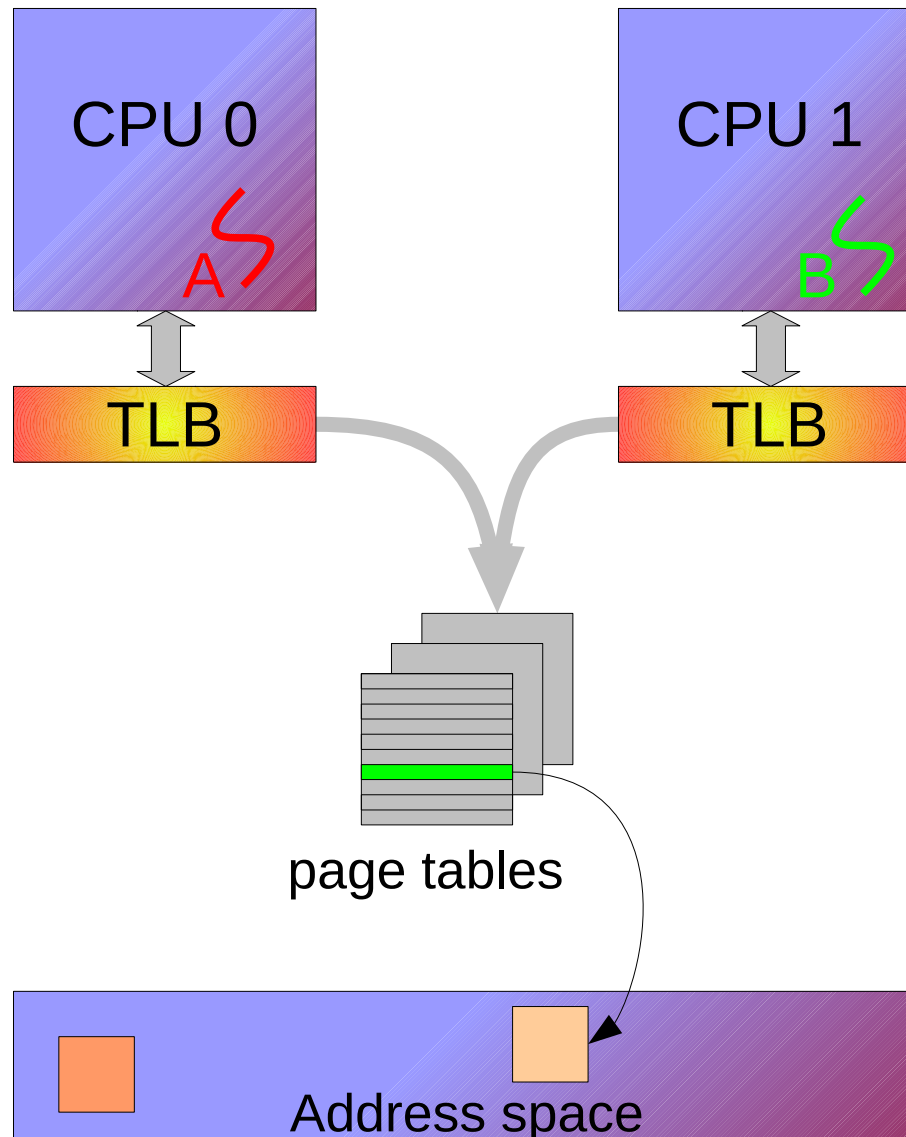


- 2 threads on 2 CPUs in one address space
- Thread A maps a page within its address space
- Thus modifies the page table to add new page
- Thread A can access newly mapped page
- Thread A and B share same address space, so B can access page too

Right ???

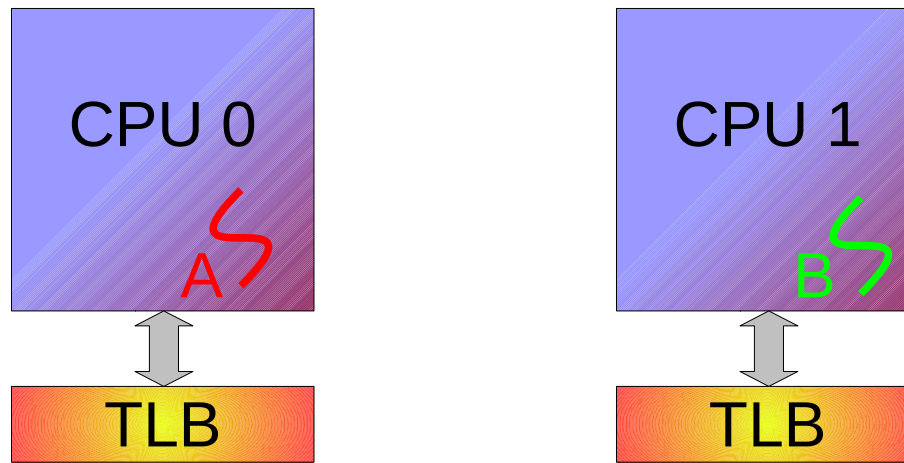


- Page tables are just memory → slow access
- Translation Lookaside Buffer caches recent address resolutions
- Very fast, but limited capacity (1<sup>st</sup>, 2<sup>nd</sup> level)

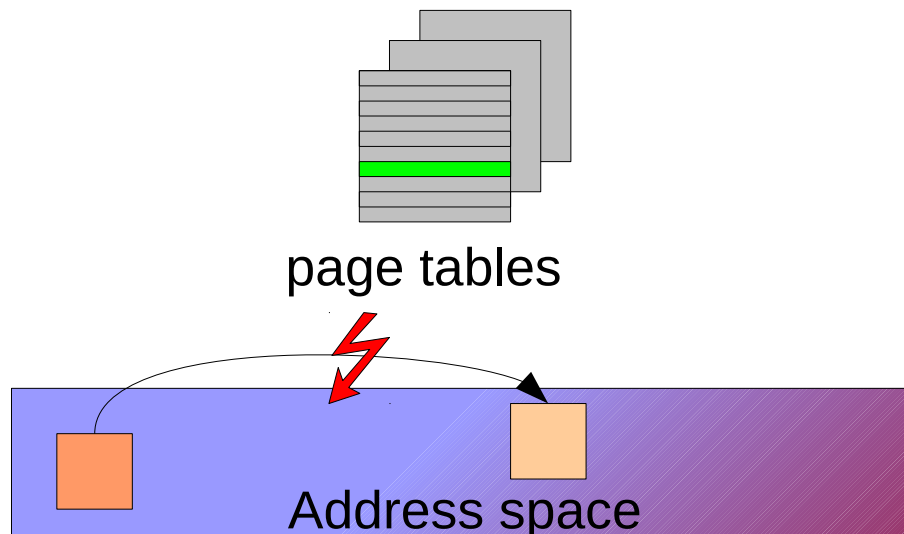


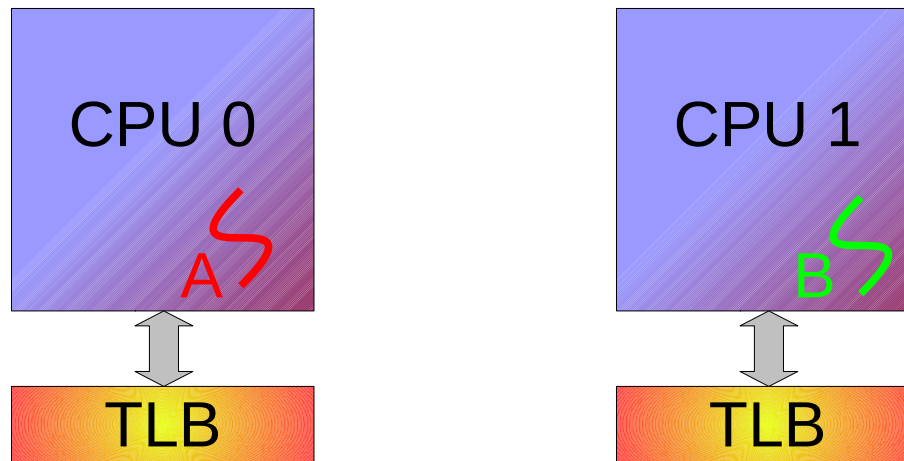
- Page tables are just memory → slow access
- Translation Lookaside Buffer caches recent address resolutions
- Very fast, but limited capacity (1<sup>st</sup>, 2<sup>nd</sup> level)
- TLB miss → page table walk (in hard/software)
- Thread A and B share same address space, so B can access page too

**Right !**

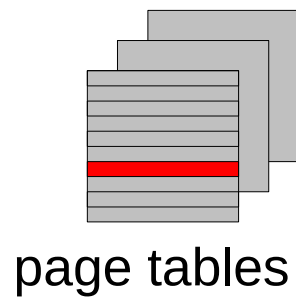


- Thread A wants to unmap the page ...

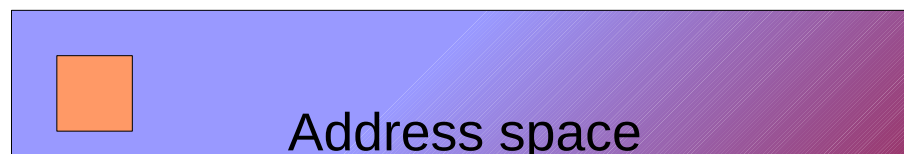


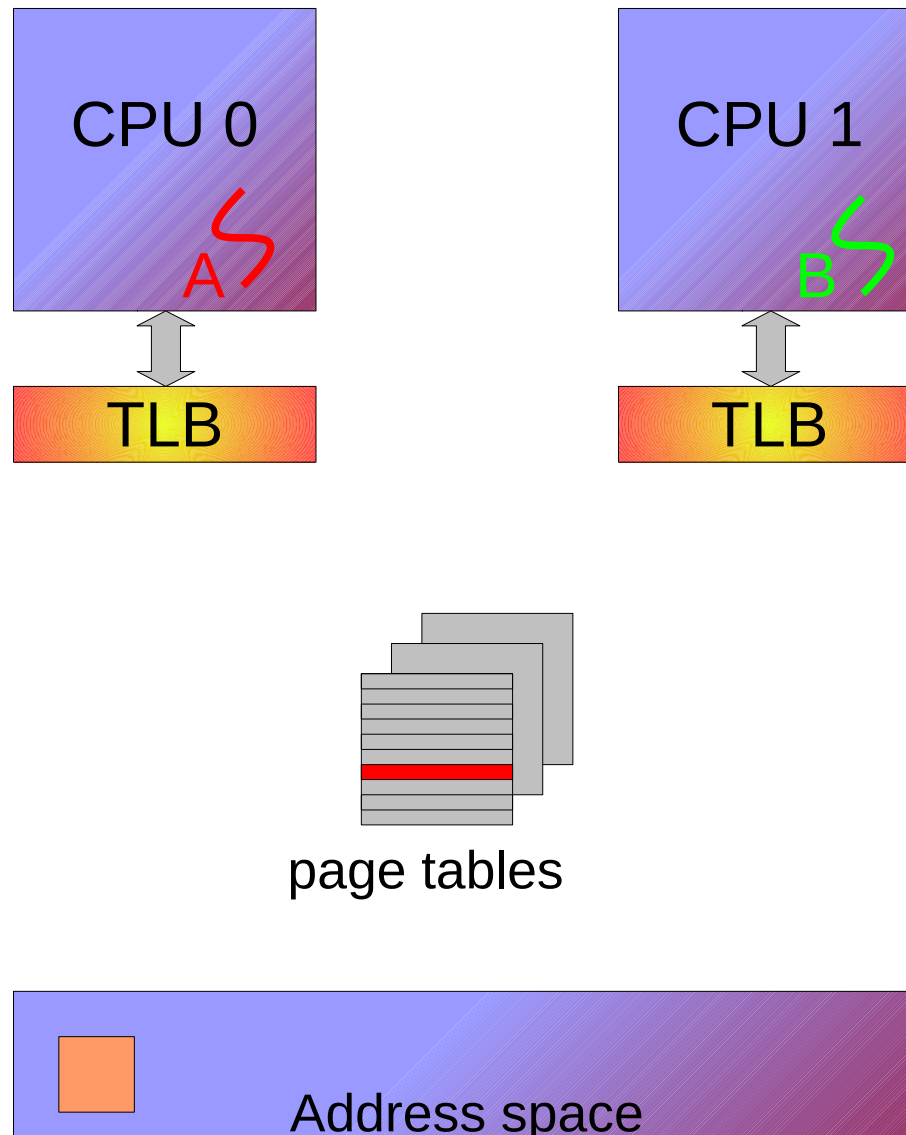


- Thread A wants to unmap the page ...
- Updates the page table, but still has an TLB entry
- Need to invalidate TLB



page tables





- Thread A wants to unmap the page ...
- Updates the page table, but still has an TLB entry
- Need to invalidate TLB
- But: CPU 1 also has a valid TLB entry, thus can access the page
- Need to invalidate TLB of CPU 1,2,3,4 ...

Page tables and TLBs have to be kept synchronized

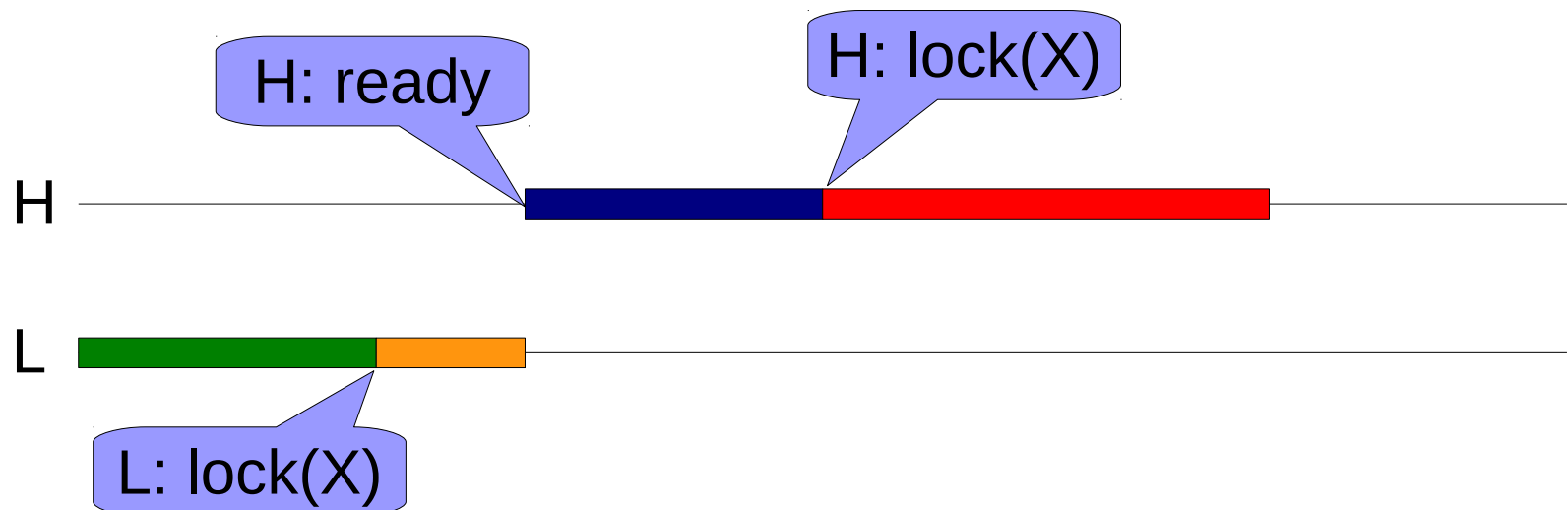
# TLB Shootdown

- *Adding* access rights to page table entries won't hurt
  - Will cause a TLB miss, followed by a page table walk and a update of the corresponding TLB entry
- *Removing* access right from page tables
  - Have to invalidate at least the affected entries in local TLB
  - Plus remote TLB invalidation of all CPUs that might have that entry cached → Inter-Processor-Interrupt to force TLB shutdown
- Local CPU has to wait for all remote CPUs to confirm their TLB invalidation
- Unmap is recursive, thus quite costly and potentially very long running



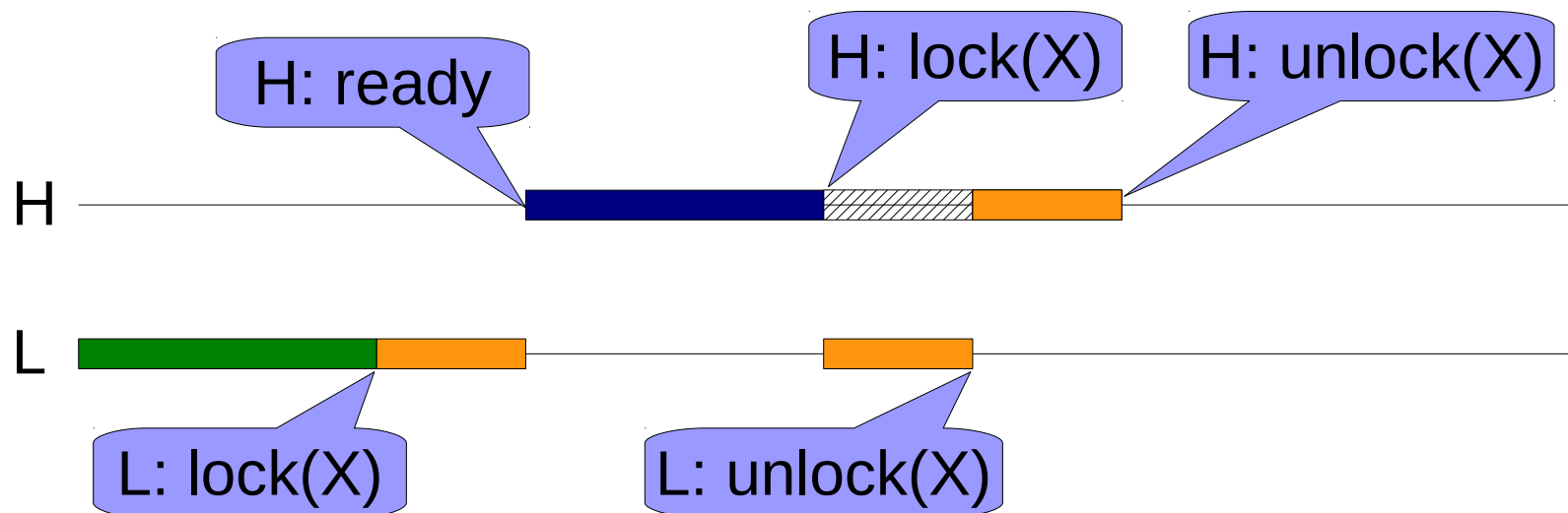
# Priority Inversion

- Low priority thread L and high priority thread H share critical section, protected by a **spinlock**
- L runs first, acquires lock and enters **critical section**
- H becomes ready, **preempts L** and executes
- H tries to grab lock, **spins ... forever**



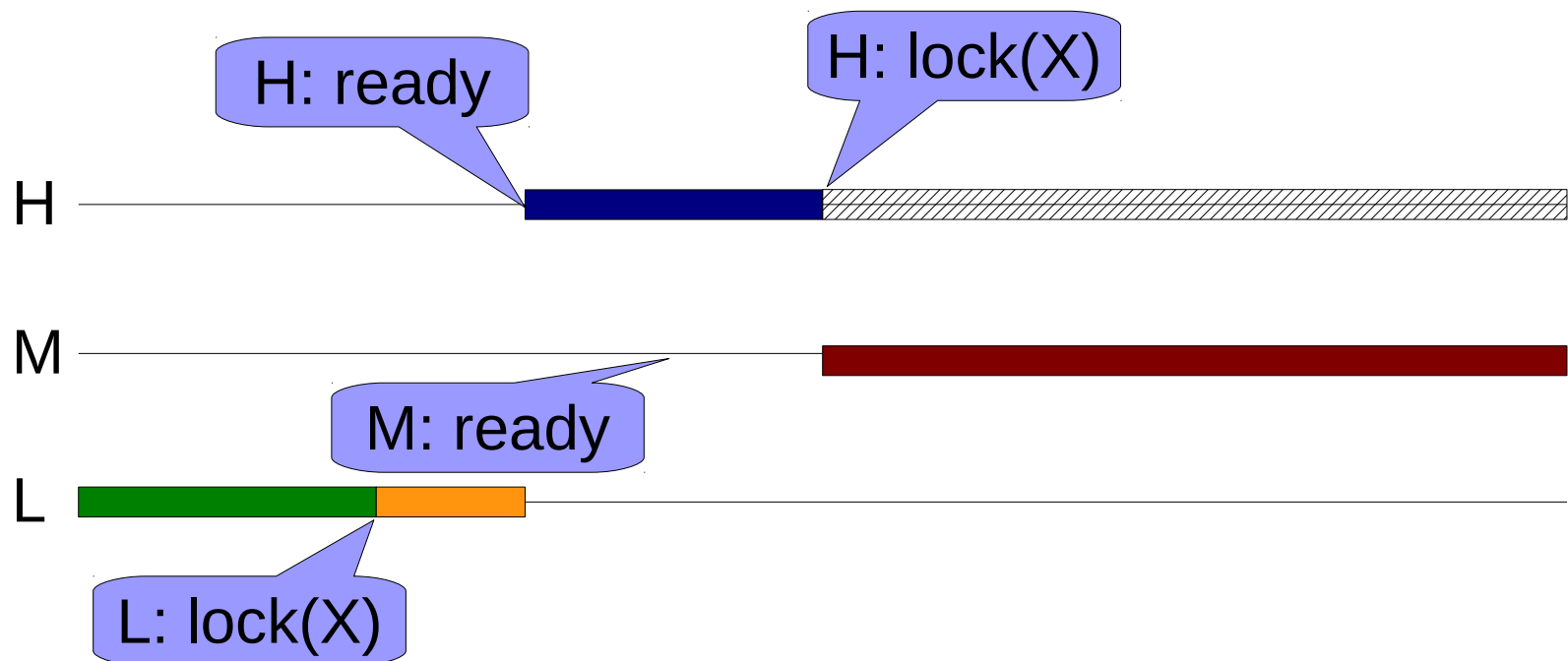
# Priority Inversion

- Low priority thread L and high priority thread H share critical section, protected by a **semaphore**
- L runs first, acquires lock and enters **critical section**
- H becomes ready, **preempts L** and executes
- H tries to grab lock and **blocks**
- L continues, completes critical section
- H enters **critical section** and completes it



# Priority Inversion

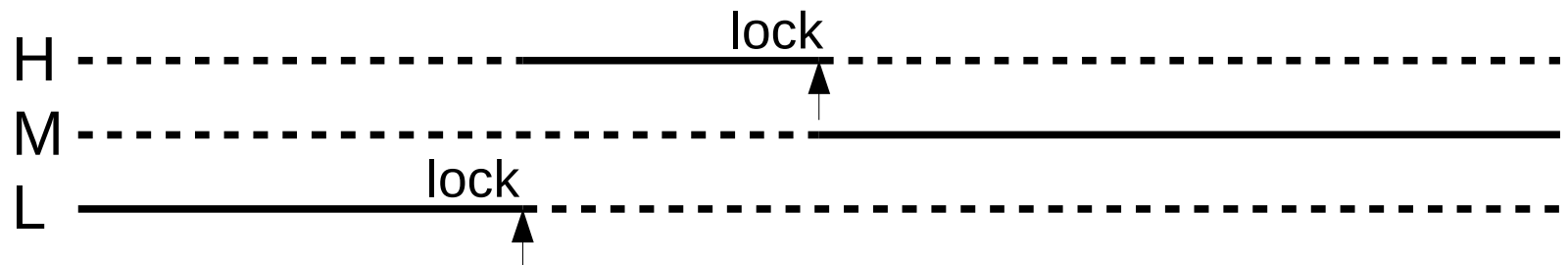
- Low priority thread L and high priority thread H share critical section, protected by a **semaphore**
- L runs first, grabs lock, gets preempted by H
- H runs, meanwhile M becomes ready
- H tries to grab lock, blocks, now M can run ... forever



- Spinlock: Deadlock, since L can never run to release lock
- Semaphore: H is delayed by *unrelated, lower priority* thread M, thus effectively M have a higher priority than H
- Reason: Although H is blocked on L, the priority of L is unchanged, therefore M can execute
- Two solutions: Priority Inheritance and Priority Ceiling
  - Priority Ceiling: For a given resource all potential resource requester are known a priori, ceiling priority of this set is assigned to resource
  - Priority Inheritance: Whenever a higher priority thread blocks on a resource (lock), the resource owner (lock holder) gets its priority elevated to that one it blocks

# Wait-Free Synchronization

- 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

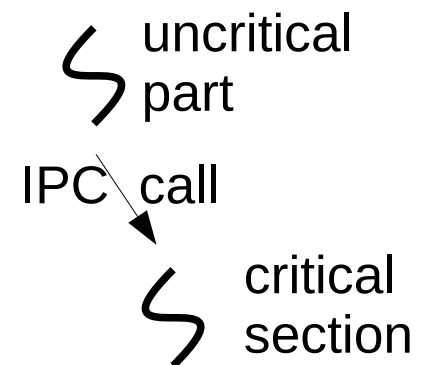
# Helping: Basics

- Resources / critical sections are modelled as service threads
- Entering a critical section = acquire lock is sending an IPC to the service provider
- Leaving = unlocking is done by replying

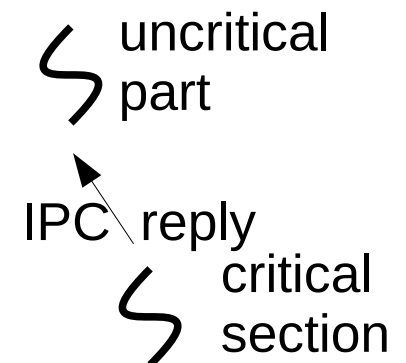
## ■ Uniprocessor

- Thread L (low) wants to access resource, thus sends to thread R (resource arbiter)
- Thread H (high) becomes ready, also wants R, tries to send to R
  - R free → normal IPC as usual
  - R occupied → donates its own time to R, thus *helping* current lock holder (L)
  - R replies to L, accepts message from H
  - H is delayed by length of critical section

Lock()

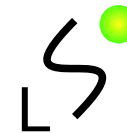


Unlock()



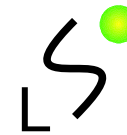
# Helping: Example

- Free-standing threads have a time quantum associated ●



## Helping: Example

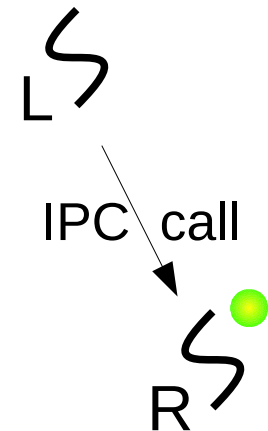
- Free-standing threads have a time quantum associated
- Service threads don't have time on their own, need time to provide service







# Helping: Example



- Free-standing threads have a time quantum associated
- Service threads don't have time on their own, need time to provide service
- Calling a service implies donating time



## Helping: Example

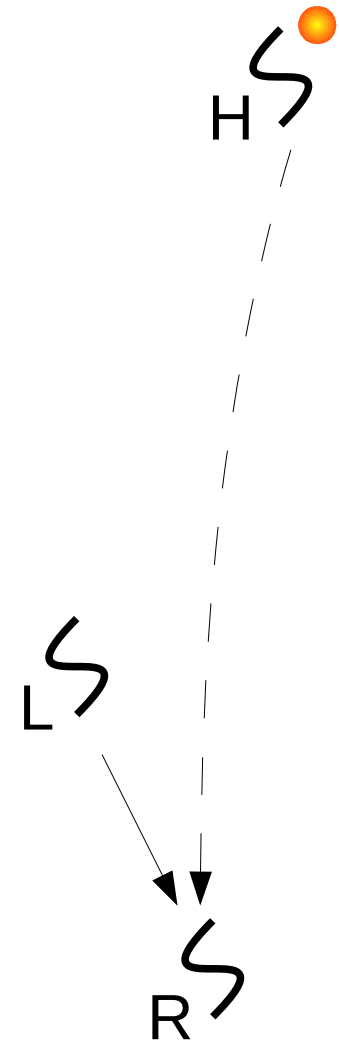
- Free-standing threads have a time quantum associated
- Service threads don't have time on their own, need time to provide service
- Calling a service implies donating time
- A higher priority thread becomes ready, i.e. its time quantum  gets selected
- Currently active time quantum  is deselected, thus thread R is preempted

H  

L   
↓  
R 

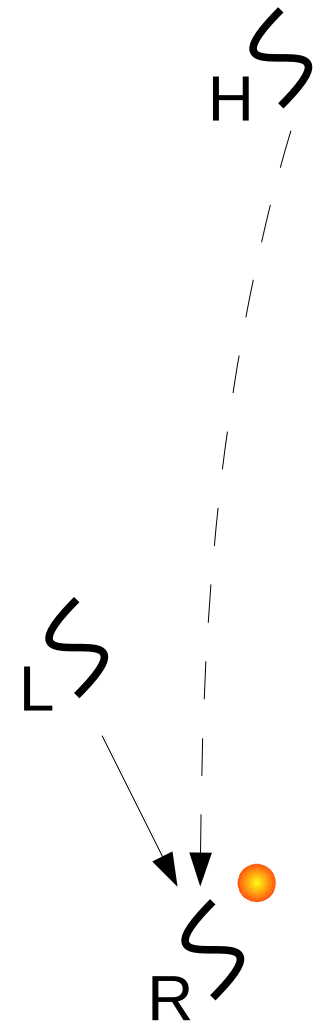
## Helping: Example

- Free-standing threads have a time quantum associated
- Service threads don't have time on their own, need time to provide service
- Calling a service implies donating time
- A higher priority thread becomes ready, i.e. its time quantum gets selected
- Currently active time quantum is deselected, thus thread R is preempted
- H tries to lock R = send IPC to R, but R is not ready to receive = already locked



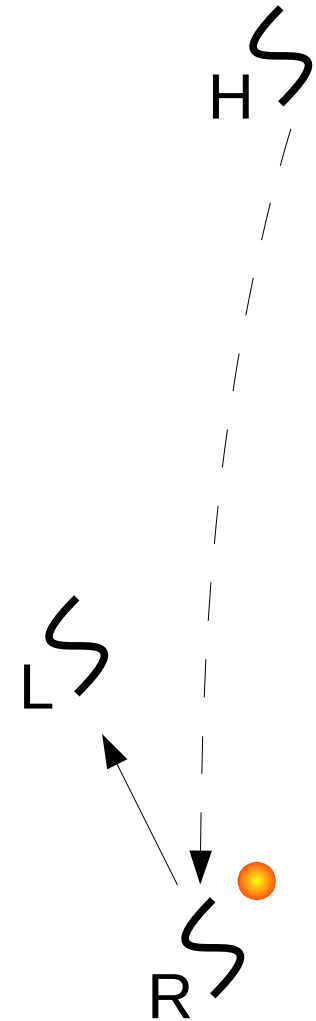
## Helping: Example

- Free-standing threads have a time quantum associated
- Service threads don't have time on their own, need time to provide service
- Calling a service implies donating time
- A higher priority thread becomes ready, i.e. its time quantum gets selected
- Currently active time quantum is deselected, thus thread R is preempted
- H tries to lock R = send IPC to R, but R is not ready to receive = already locked
- H donates its time to R, *helping* L to complete its critical section



# Helping: Example

- R replies to L, but immediately switches back to H



## Helping: Example

- R replies to L, but immediately switches back to H
- L has left its critical section = released lock, H is activated again and R is ready to receive

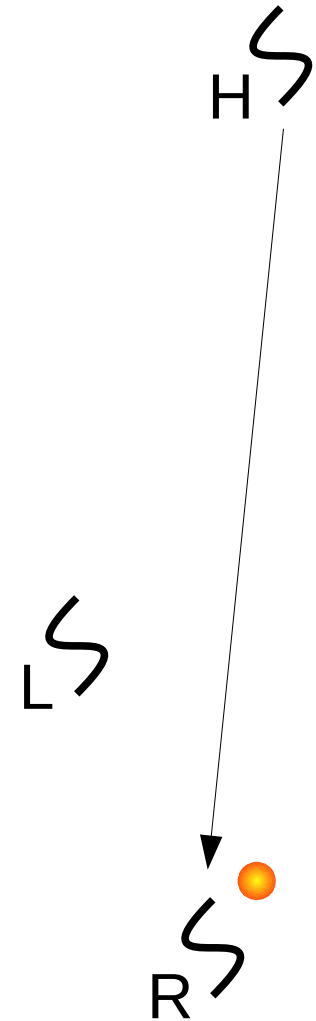
H 

L 

R 

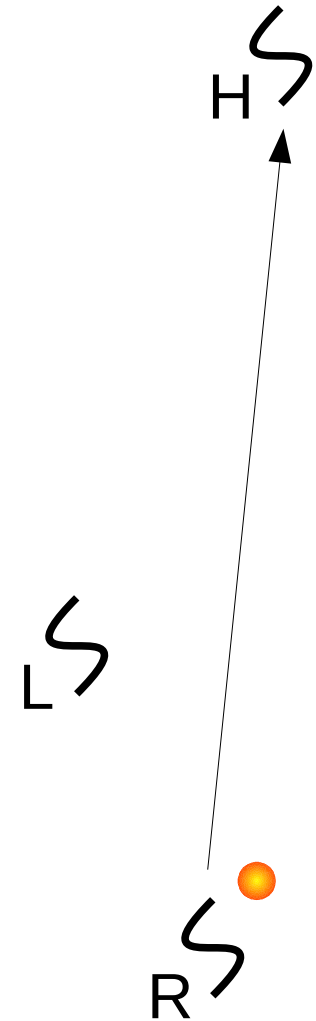
## Helping: Example

- R replies to L, but immediately switches back to H
- L has left its critical section = released lock, H is activated again and R is ready to receive
- H retries to acquire lock = send an IPC call to thread R, this time successfully



## Helping: Example

- R replies to L, but immediately switches back to H
- L has left its critical section = released lock, H is activated again and R is ready to receive
- H retries to acquire lock = send an IPC call to thread R, this time successfully
- R replies, giving received time quantum back to H, which has left its critical section





# Helping: Example

- R replies to L, but immediately switches back to H
- L has left its critical section = released lock, H is activated again and R is ready to receive
- H retries to acquire lock = send an IPC call to thread R, this time successfully
- R replies, giving received time quantum back to H, which has left its critical section
- H continues execution (L is also ready, but cannot run, its priority is too low)

H 

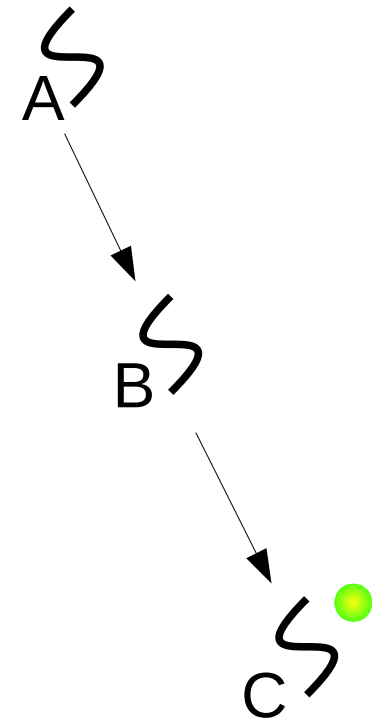
L 

R 

# Transitive Helping

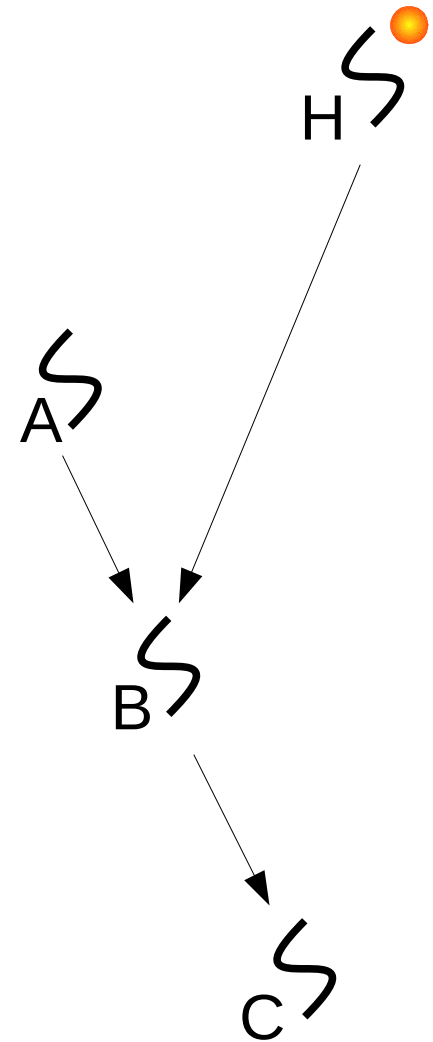
- Thread A calls Thread B, which in turn calls to C
- Time quantum of A is donated to C

H S



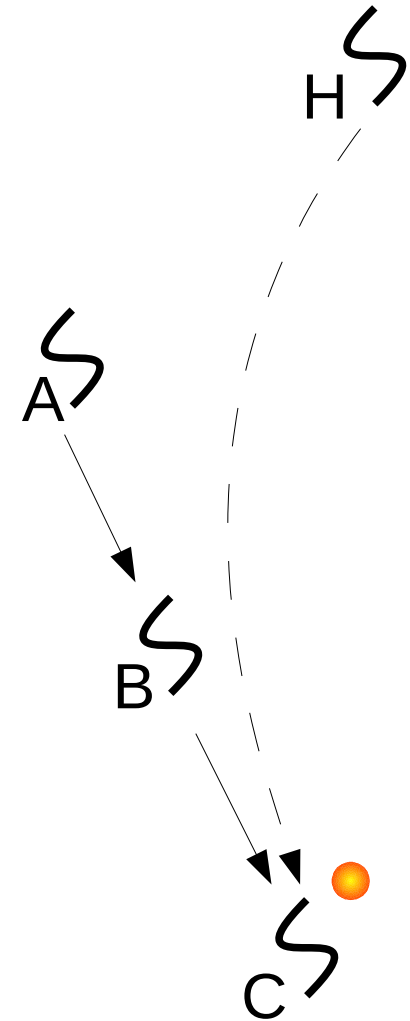
# Transitive Helping

- Thread A calls Thread B, which in turn calls to C
- Time quantum of A is donated to C
- H becomes ready, preempts C and sends IPC to B



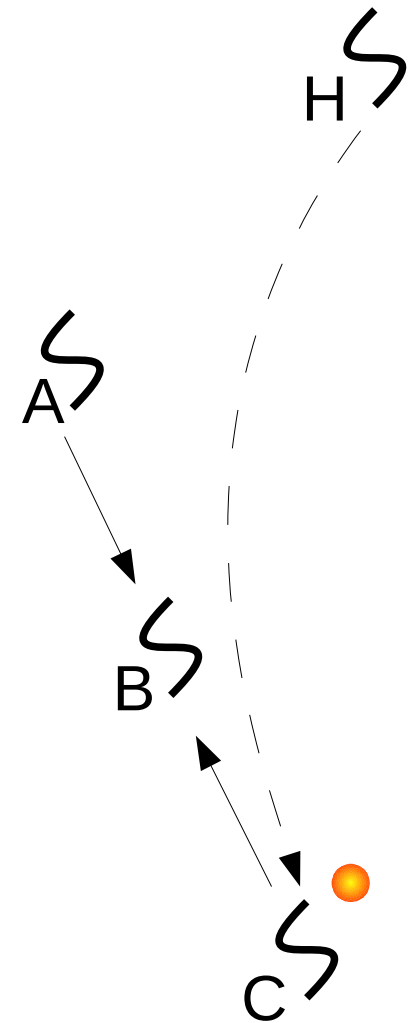
# Transitive Helping

- Thread A calls Thread B, which in turn calls to C
- Time quantum of A is donated to C
- H becomes ready, preempts C and sends IPC to B
- B is not receiving, thus H helps C through B transitively

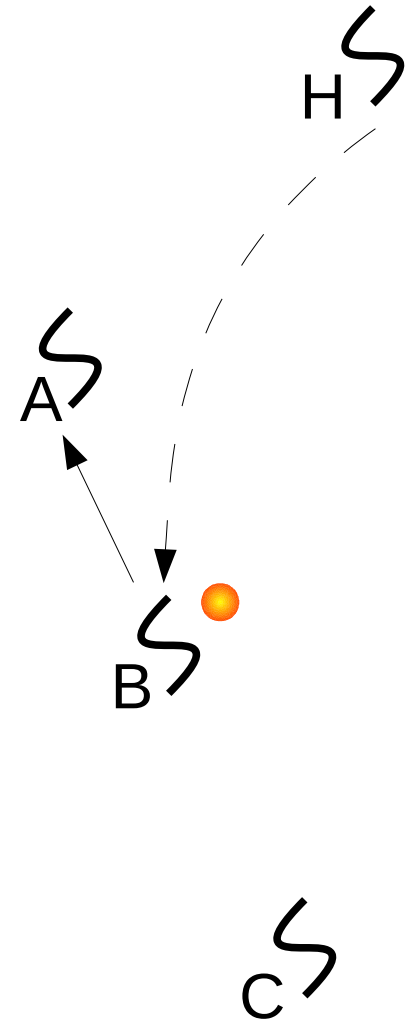


# Transitive Helping

- Thread A calls Thread B, which in turn calls to C
- Time quantum of A is donated to C
- H becomes ready, preempts C and sends IPC to B
- B is not receiving, thus H helps C through B transitively
- C, running on time from H finishes its service and replies to B

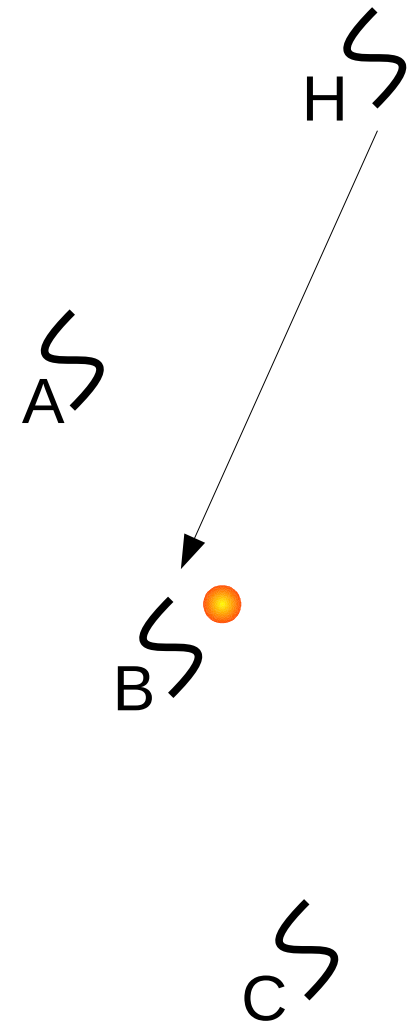


- Thread A calls Thread B, which in turn calls to C
- Time quantum of A is donated to C
- H becomes ready, preempts C and sends IPC to B
- B is not receiving, thus H helps C through B transitively
- C, running on time from H finishes its service and replies to B
- H helps B, B replies to A, B therefore becomes receiving



# Transitive Helping

- Thread A calls Thread B, which in turn calls to C
- Time quantum of A is donated to C
- H becomes ready, preempts C and sends IPC to B
- B is not receiving, thus H helps C through B transitively
- C, running on time from H finishes its service and replies to B
- H helps B, B replies to A, B therefore becomes receiving
- B receives IPC call from H, and might call C in turn as well



# Summary

---

- Disabling Interrupts
- Spinlocks (TS, TTS, Ticket, MCS) and Semaphores
- Lock-free and Wait-free synchronization
- Atomic updates (xchg) → example single linked list
- Page table modifications → TLB shutdown
- Priority Inversion
- Priority Inheritance → Helping