Distributed OS

Torsten Frenzel

Multiprocessor Synchronization

using Read-Copy Update

TECHNISCHE
UNIVERSITÄT
DRESDEN

Outline

- Basics
  - Introduction
  - Examples
- Design
  - Grace periods and quiescent states
  - Grace period measurement
- Implementation in Linux
  - Data structures and functions
  - Examples
- Evaluation
  - Scalability
  - Performance
- Conclusion

Introduction

- Multiprocessor OSs need to synchronize access to shared data structures

➔ Fast synchronization primitives are crucial for performance and scalability

- Two important facts about OSs
  - Small critical sections
  - Data structures with many reads and few writes/updates
- Goals
  - Reducing synchronization overhead
  - Reducing lock contention
  - Avoiding deadlocks

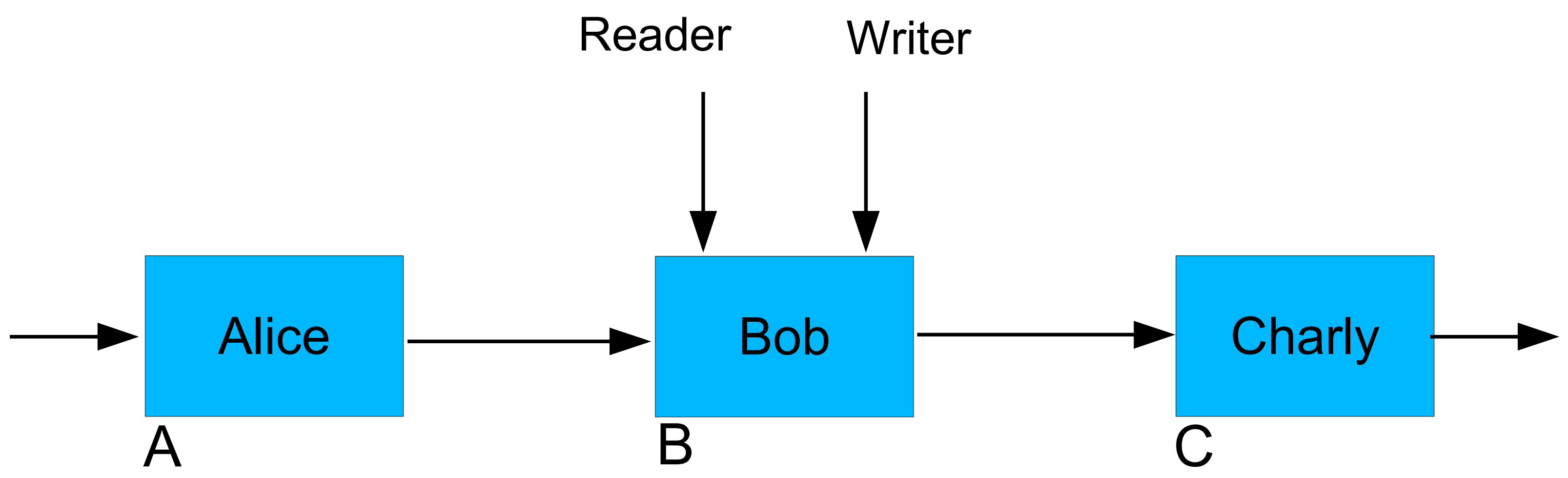
Synchronization Primitives

- Locks
  - Spinlock
  - MCS-lock
  - Reader-writer lock
- Granularity
  - Coarse: One large critical section
  - Fine: Many small critical sections
- Lock-free synchronization
  - ➔ Fine grained
  - Uses atomic operations to update data structures
  - Avoids disadvantages of locks
  - Hard (to do right) for complex data structures

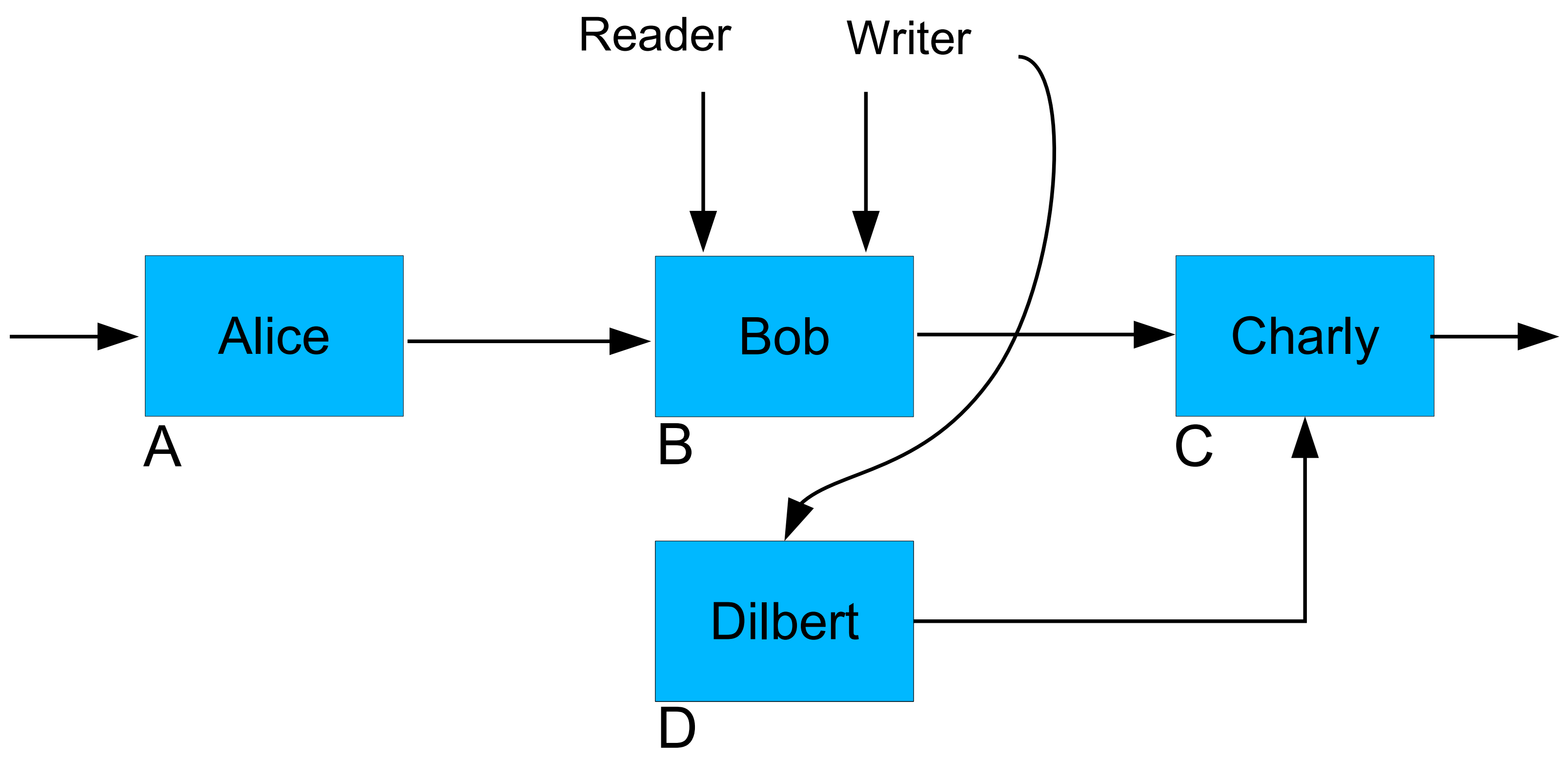
'Lockless' Synchronization

- Idea
  - No locks on reader side
  - Locks only on writer side
  - **Two-phase update** protocol
- Prerequisites
  - Many readers and few writers on data structure
  - Short critical sections
  - Data structures support atomic updates
  - Stale data tolerance for readers

Example 1: List

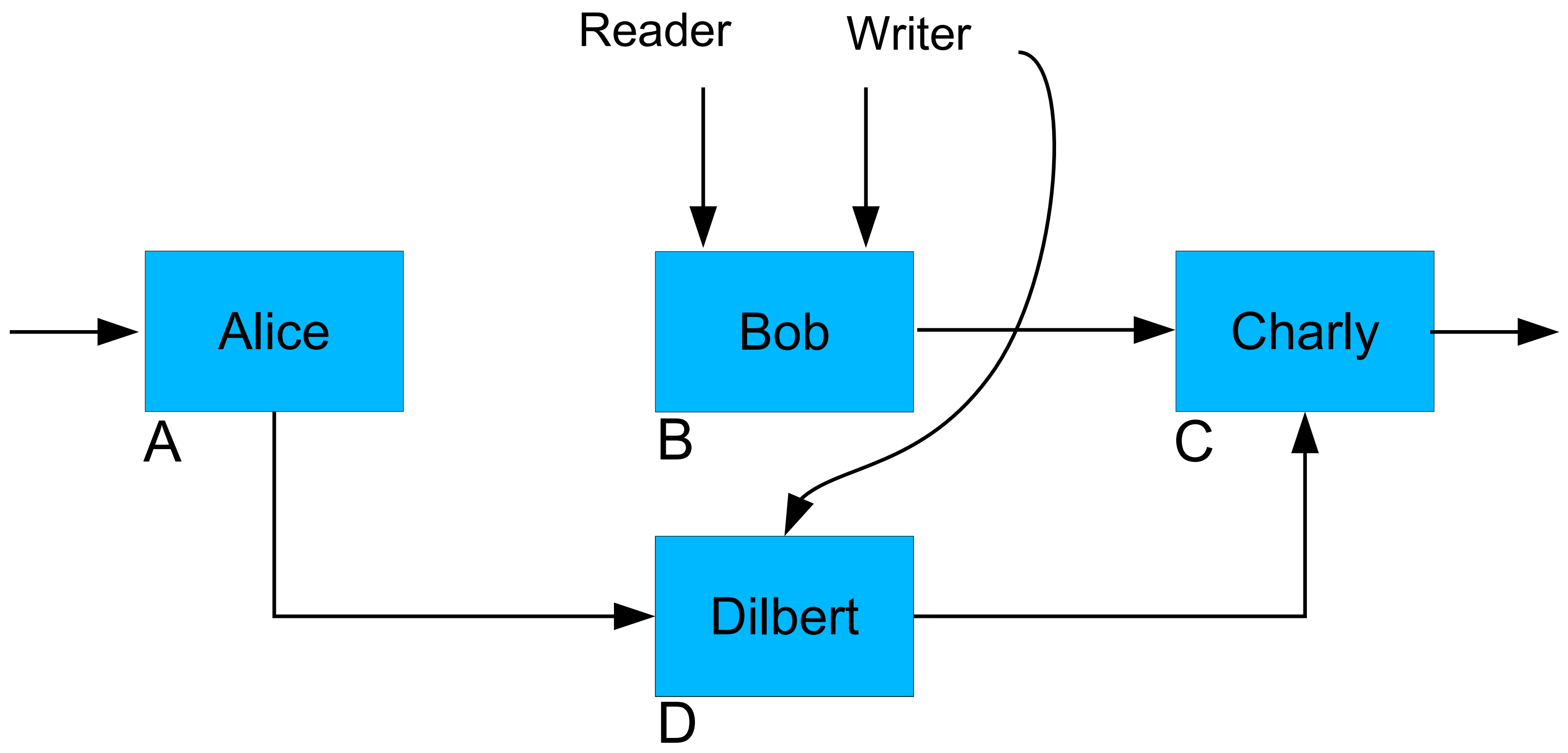


Example 1: List



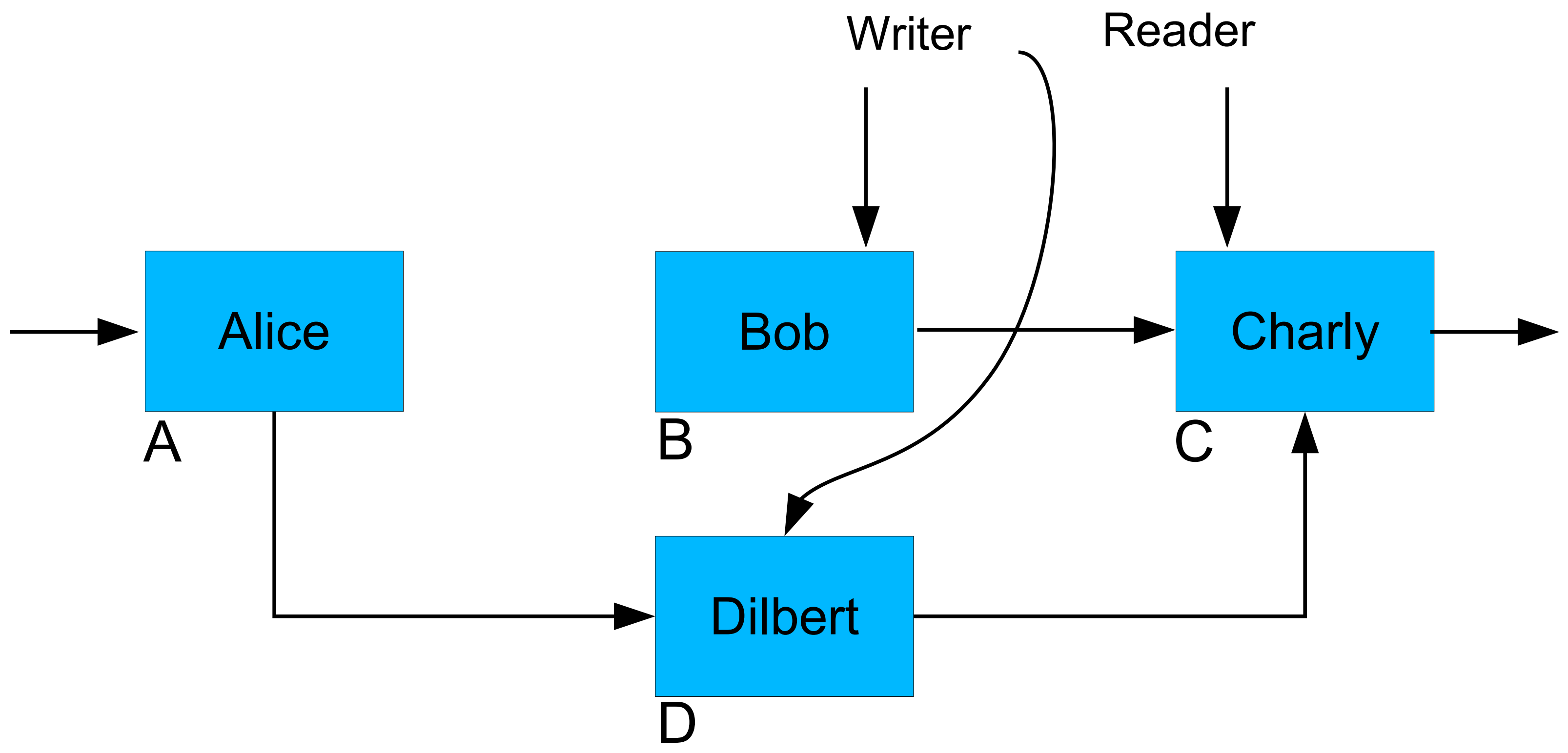
Copy & Update Phase: copy old version and update to new version

Example 1: List



Copy & Update Phase: make updated version visible

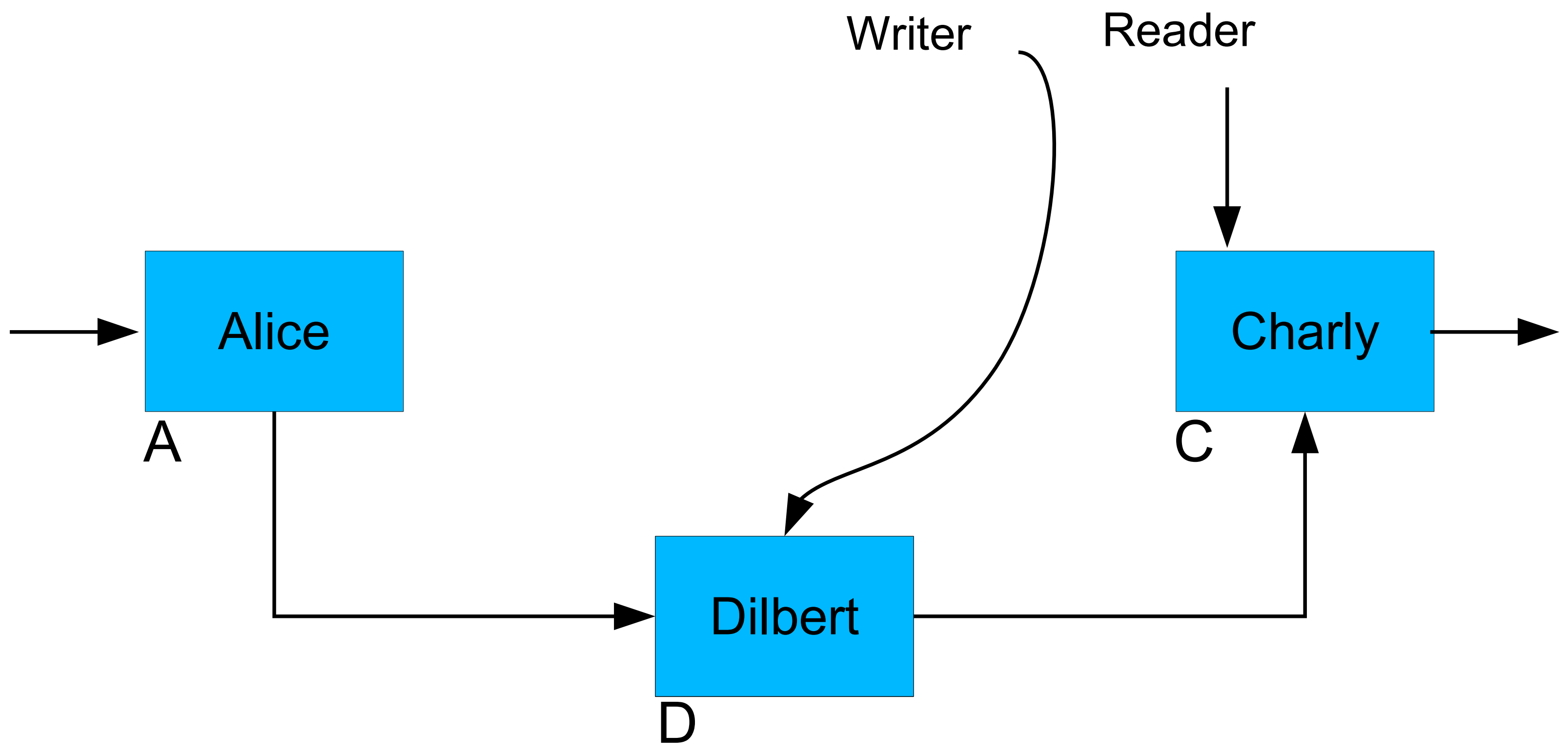
Example 1: List



Copy & update phase

Wait phase: concurrent reader proceeds

Example 1: List



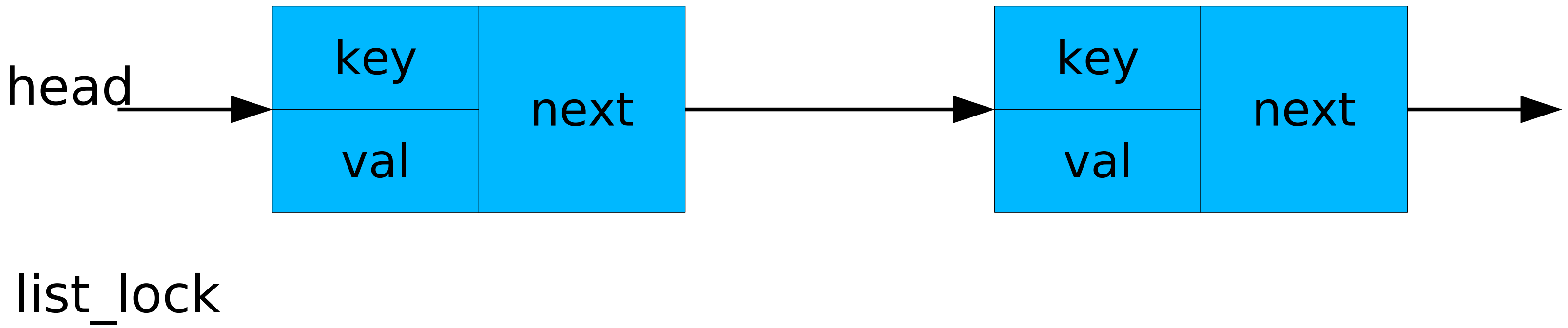
Copy & update phase

Wait phase

Reclamation phase: old version is deleted

Example 1: List

```
struct elem { long key; char *val; struct elem *next; };

struct elem *head; // pointer to first list element

lock_t list_lock;       // lock to synchronize access to list
```



Example1: List – Read Operation

```
int read(long key)
{
  lock(&list_lock);
  struct elem *p = head→next;
  while (p != head)
  {
     if (p→key == key)
     {
        /* read-only access to p */
        read unlock(&list_lock);
        return OK;
     }
     p = p→next;
  }
  unlock(&list_lock);
  return NOT_FOUND;
}
```

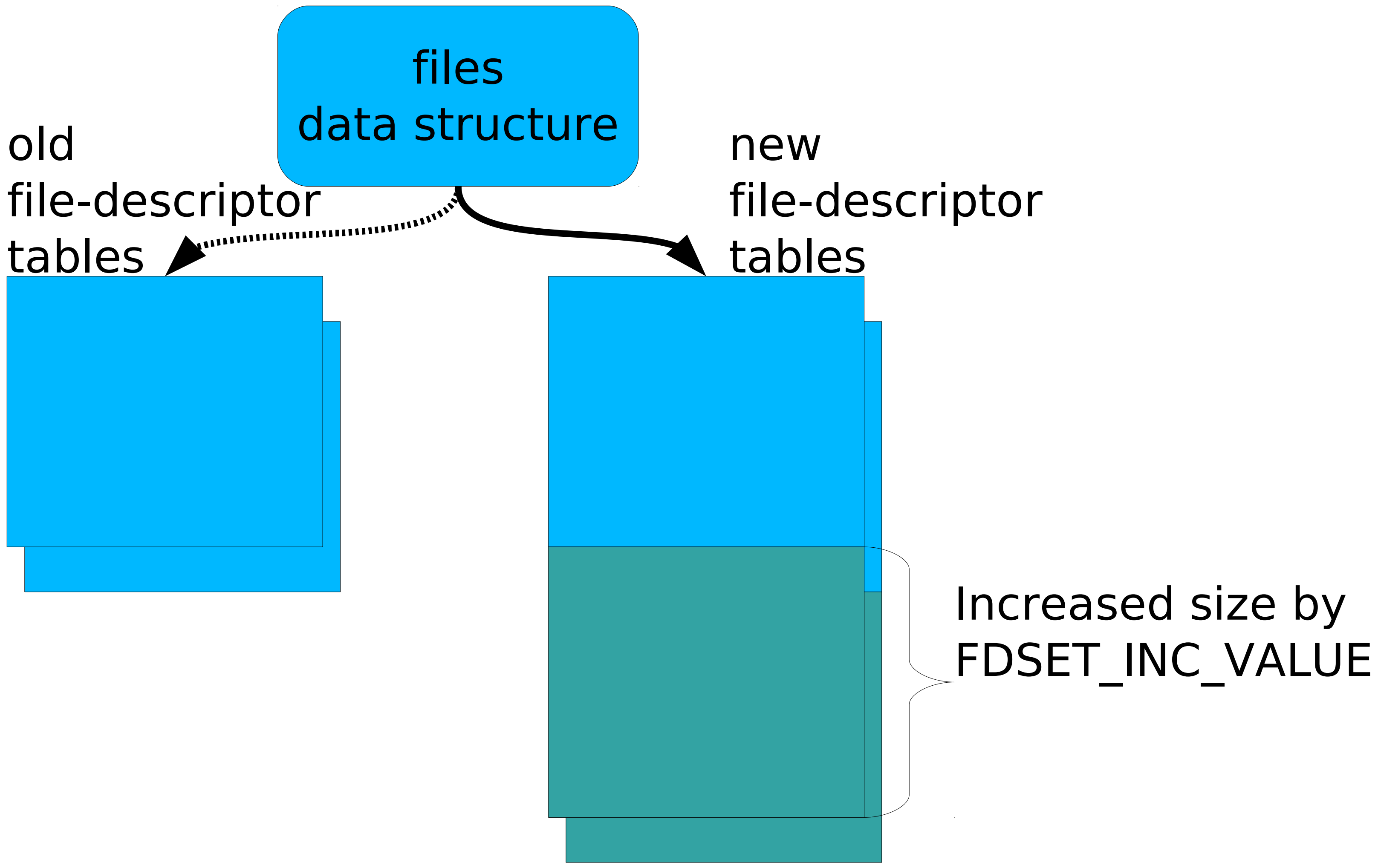
```
int read(long key)
{

  struct elem *p = head→next;
  while (p != head)
  {
     if (p→key == key)
     {
        /* read-only access to p */

        return OK;
     }
     p = p→next;
  }

  return NOT_FOUND;
}
```

Example1: List – Write Operation

```
int write(long key, char *val)
{
  struct elem *p = head→next;
  lock(&list_lock);
  while (p != head)
  {
    if (p→key == key)
    {
      /* write-access to p */

      p.val = val;



      unlock(&list_lock);



      return OK;
    }
    prev_p = p; p = p→next;
  }
  unlock(&list_lock);
  return NOT_FOUND;
}
```

```
int write(long key, char *val)
{
  struct elem *p = head→next;
  lock(&list_lock);
  while (p != head)
  {
    if (p→key == key)
    {
      /* copy & update */
      struct elem *new_p = copy(p);
      new_p.val = val;
      new_p->next = p→next;
      prev_p->next = new_p;
      unlock(&list_lock);
      wait_for_rcu(); /* wait phase */
      kfree(p); /* reclamation phase */
      return OK;
    }
    prev_p = p; p = p→next;
  }
  unlock(&list_lock);
  return NOT_FOUND;
}
```

Example 2: File-descriptor Table Expansion



Example 2: File-descriptor Table Expansion

- Expansion of file-descriptor table (files)
  - Current fixed-size (max_fdset)
  - Pointer to fixed-size array of open files (open_fds)
  - Pointer to fixed-size array of open files closed on exit (close_on_exec)

```
spin_lock(&files→file_lock);
nfds = files→max_fdset + FDSET_INC_VALUE;
/* allocate and fill new_open_fds */
/* allocate and fill new_close_on_exec */
...
old_openset = xchg(&files->open_fds, new_open_fds);
old_execset = xchg(&files->close_on_exec, new_close_on_exec);
...
nfds = xchg(&files->max_fdset, nfds);
spin_unlock(&files→file_lock);
wait_for_rcu();
free_fdset(old_openset, nfds);
free_fdset(old_execset, nfds);
```

Other Examples

- Routing cache
  - Copy & update phase: change the network routing topology
  - Reclamation phase: clear old routing information data
- Network subsystem policy changes
  - Copy & update phase: add new policy rules and make old rules inaccessible
  - Reclamation phase: clear data structures of old policy rules
- Hardware configuration
  - Copy & update phase: hot-unplug a CPU or device and remove any reference to the device specific data structures
  - Reclamation phase: free the memory of the data structures
- Module unloading
  - Copy & update phase: remove all references to the module
  - Reclamation phase: remove the module from the system

Implementations

- DYNIX
  - UNIX-based operating system from Sequent
- K42
  - Operating system kernel from IBM for large scale parallel architectures
- K42/Tornado
  - Operating system for large scale NUMA architectures from University of Toronto
- Linux
- L4-based Microkernels: Fiasco, Nova, Pistachio

Two-Phase Update - Principle

- Phase 1: Copy & Update Phase
  - Copy relevant data of old state
  - Update copy to new state
  - Use atomic operation to:
    - Make new state visible
    - Make old state inaccessible
- Wait period:
  - Allow ongoing read operations to proceed on the old state until completed
- Phase 2: Reclamation Phase
  - Remove old (invisible) state of data structure
  - Reclaim the memory

Deferred Memory Reclamation

- Problem:
  - **When to reclaim memory after update phase?**
  - **How long to wait?**
- Read-Copy Update uses pessimistic approach:

  „Wait until every concurrent read operation has completed and no pending references to the data structure exist“

Grace Periods and Quiescent States

- Definition of a grace period
  - *Intuitive*: duration until references to data are no longer held by any thread
  - *More formal*: duration until every CPU has passed through a **quiescent state**
- Definition of a quiescent state
  - State of a CPU without any references to the data structure
- How to measure a grace period?
  - *Enforcement*: actively induce quiescent state into CPU
  - *Detection*: passively wait until CPU has passed quiescent state

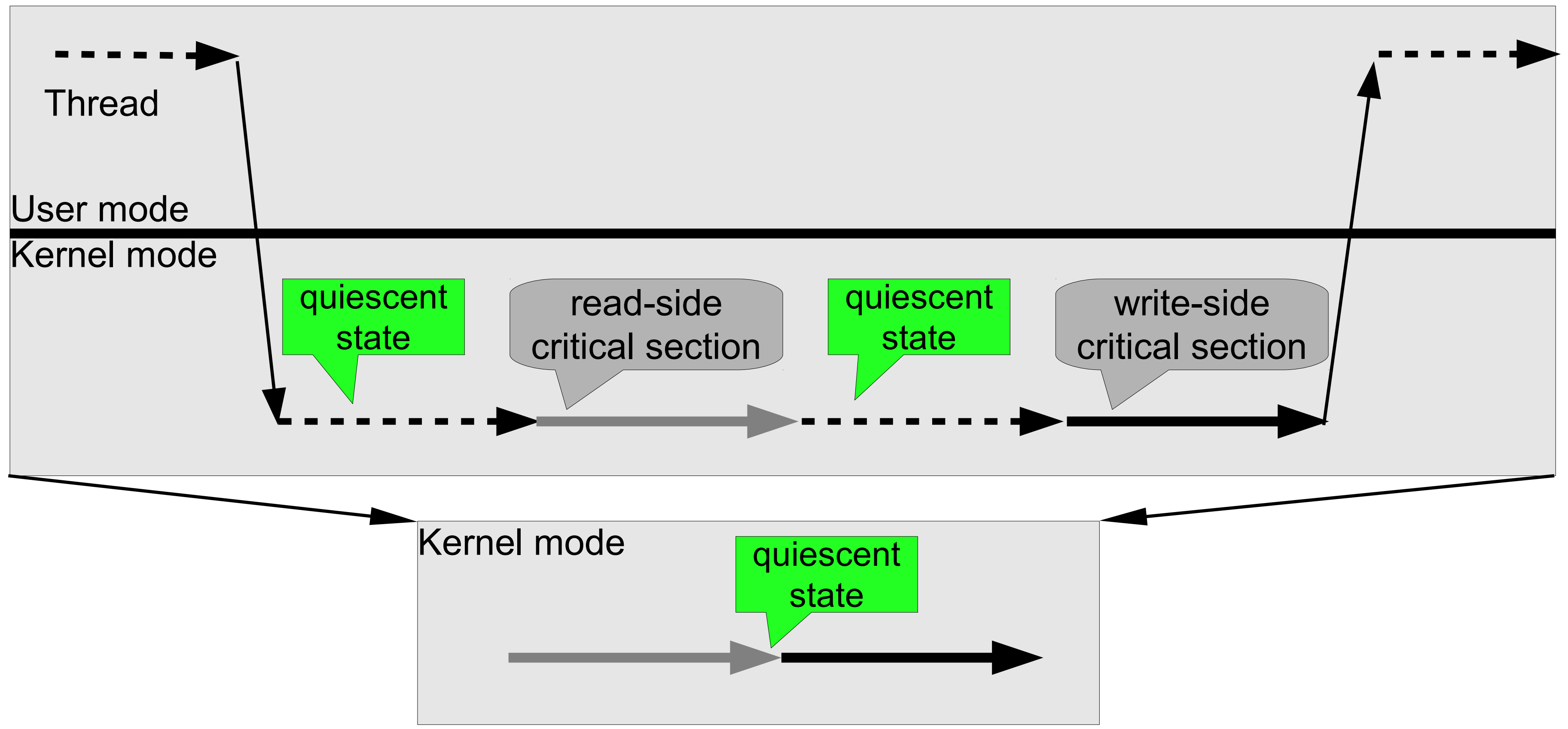
Quiescent State

- What are good quiescent states?
  - Should be easy to detect
  - Should occur not to frequently or infrequently
- **Per-CPU granularity**
  - OSs without blocking and preemption in read-side critical sections
  - For example: context switch, execution in idle loop, kernel entry/exit
- Per-thread granularity
  - OSs with blocking and preemption in read-side critical sections
  - Counting of the number of threads inside read-side critical sections

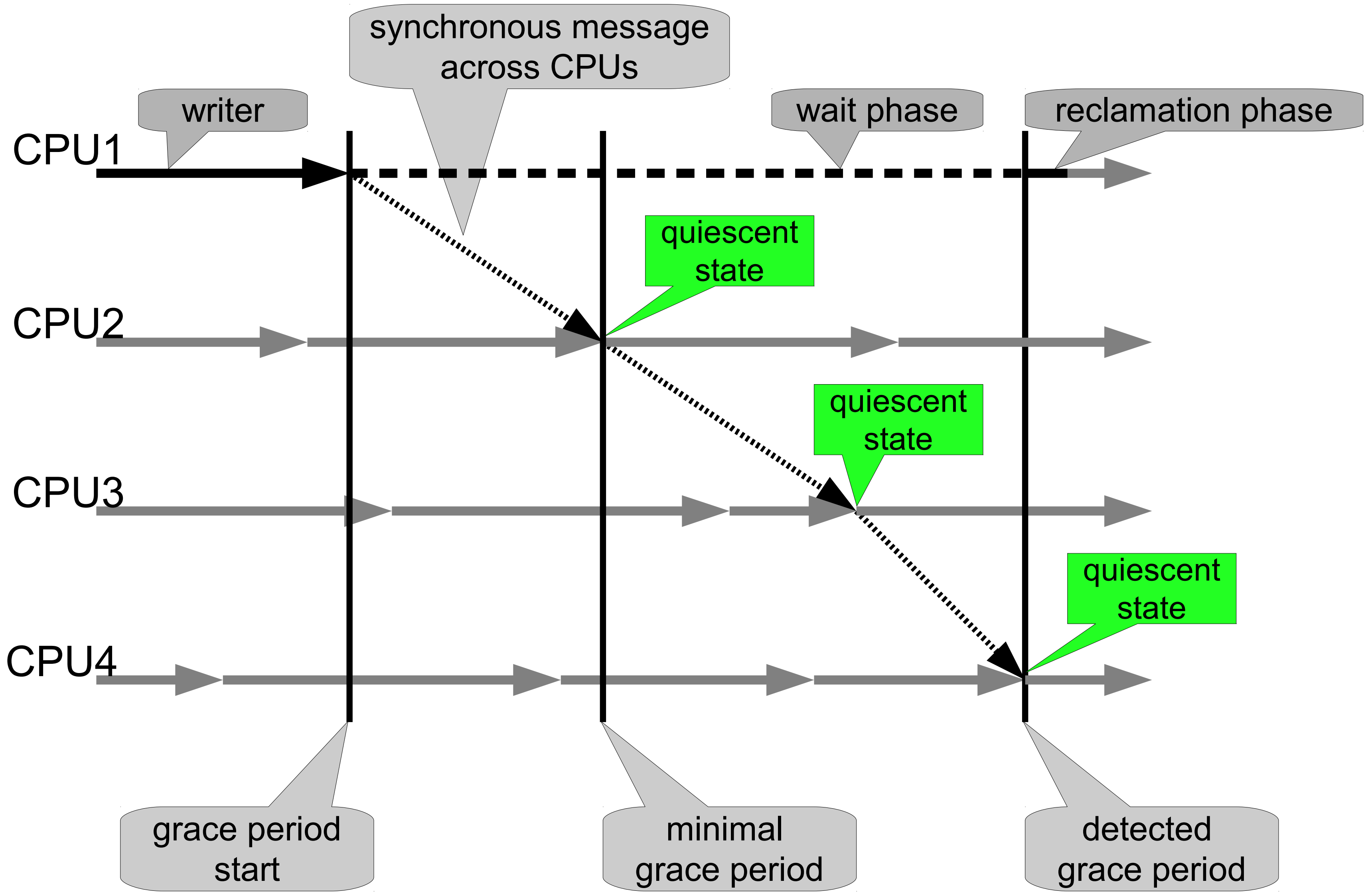
➔ *Not discussed in this lesson!*

Modelling of Critical Sections

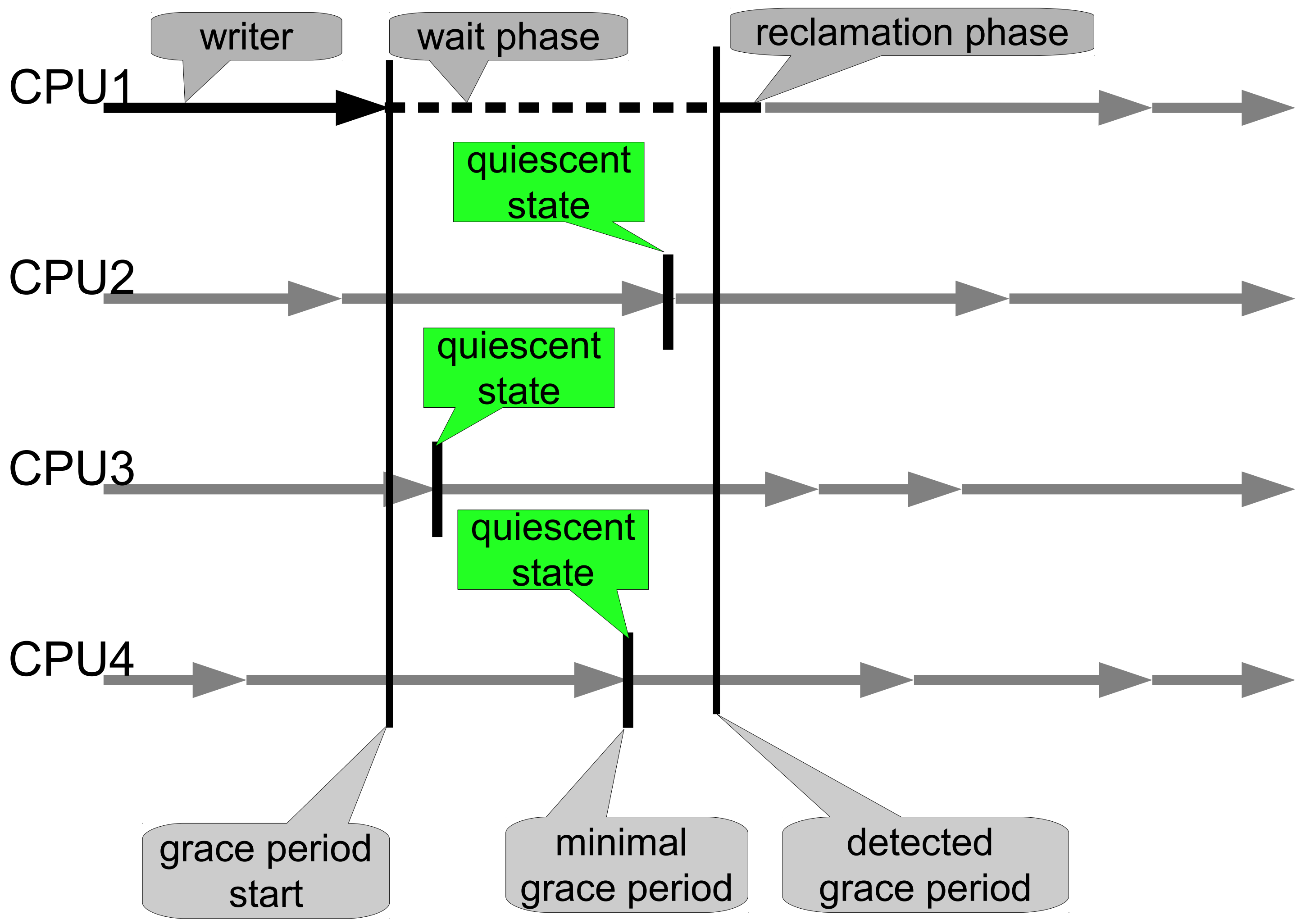
- User-level code path of threads are ignored
  - Threads execute only in the kernel
- Non-critical sections of threads are ignored
  - Threads execute continously critical sections



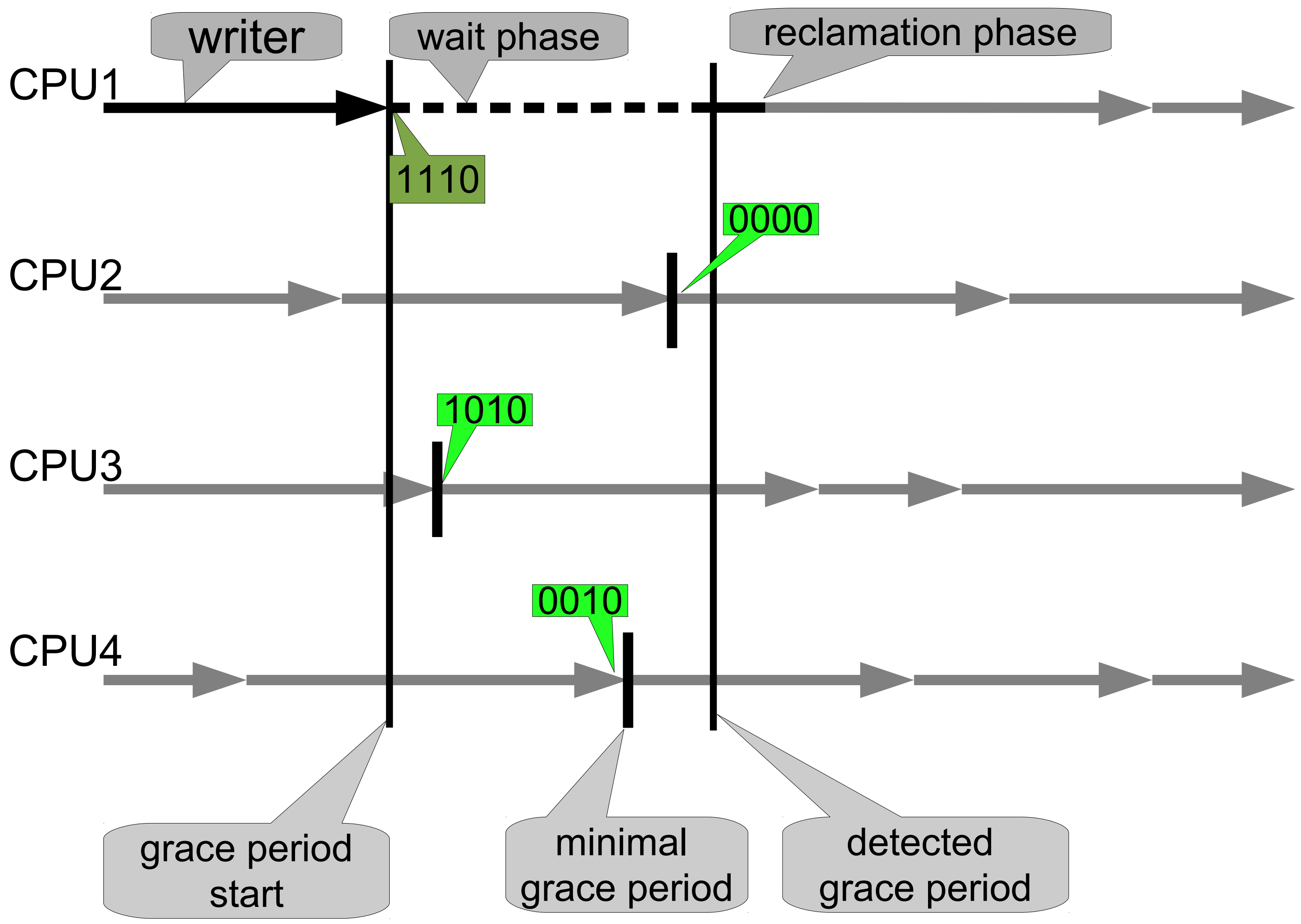
Quiescent State Enforcement



Quiescent State Detection



Quiescent State Detection Using a Bitmask

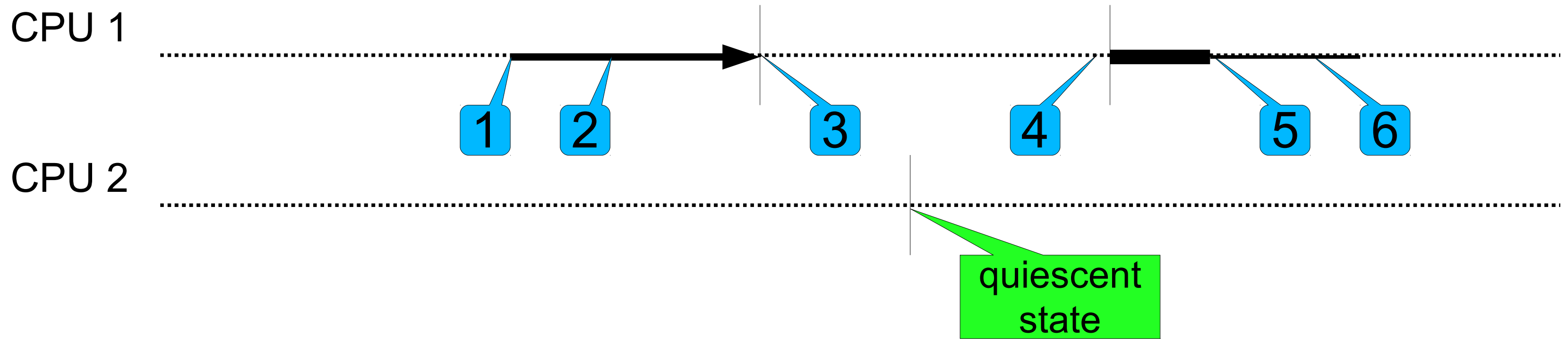


How to Make RCU Scalable?

- Observation
  - Measuring grace periods adds overheads
- Consequences
  - Generate RCU requests using callbacks instead of waiting
  - Batching: Measure on grace period for multiple RCU requests
  - Maintaining per-CPU request lists
  - Measuring of grace periods globally for all CPUs
  - Separation of RCU-related data structures into CPU-local and global data
    - CPU-local: quiescent state detection and batch handling
    - Global: grace period measurement with CPU-bitmask
  - Low overhead for detecting quiescent states
  - Minimal overhead if RCU subsystem is idle

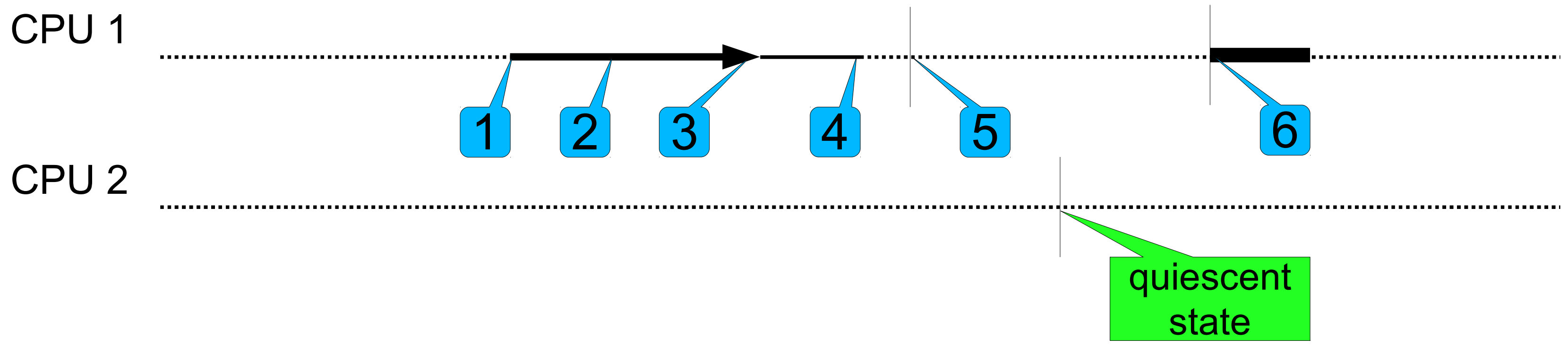
Memory Reclamation with RCU

- Memory reclamation is most important use case
  - Recall single-linked list example
- Waiting for end of grace period blocks thread:
  1. Start of operation
  2. Modify data structure
  3. **Block** current operation and start grace period
  4. Grace period completed and reclamation of memory
  5. Continue operation
  6. End of operation



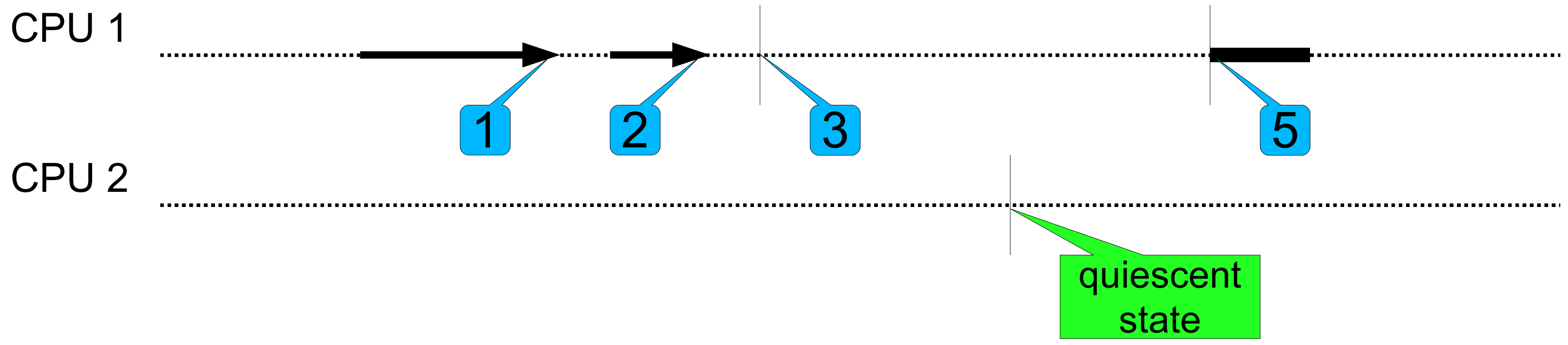
Using Callbacks

- Callback is a function that is invoked to perform the memory reclamation after the grace period completed
- A callback defines an RCU **request**
  1. Start of operation
  2. Modify data structure
  3. Register callback and continue operation **without blocking**
  4. End of operation
  5. Start of grace period measurement
  6. Grace period completed and reclamation of memory



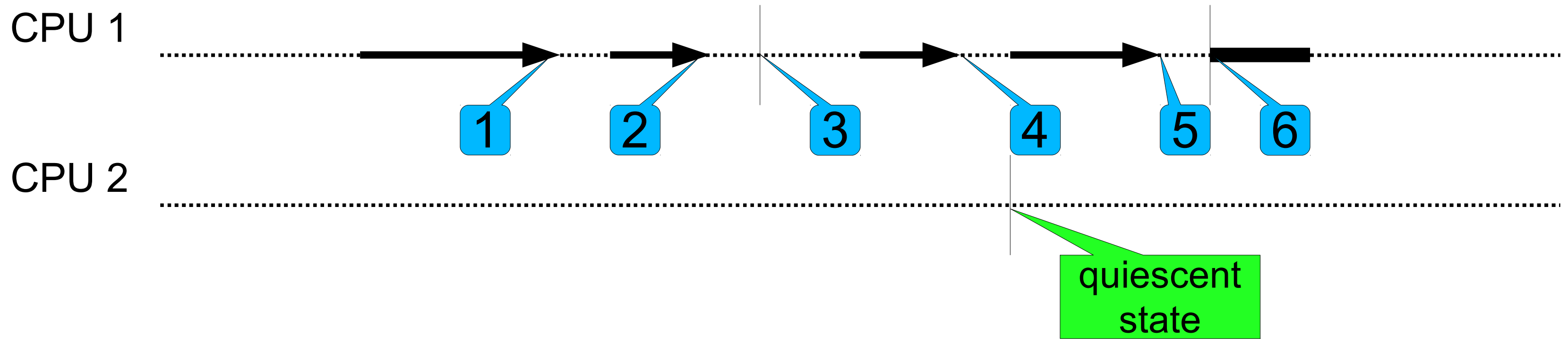
Batching for Multiple Requests

- Batch contains a set of request which wait for the same grace period to complete
- RCU requests must be registered before measurement of grace period starts
  1. Register RCU request 'A' into batch
  2. Register RCU request 'B' into batch
  3. Start new grace period
  4. Grace period completed, execute request 'A' and 'B' of batch



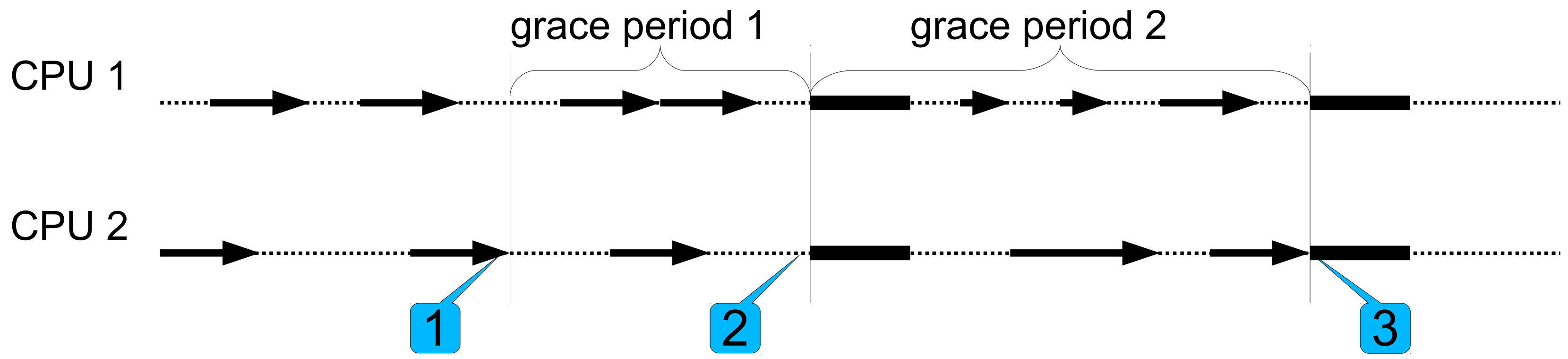
Closed and Open Batches

- Closed batch holds requests that are waiting for current grace period to complete
- Open batch holds requests that are waiting for next grace period to complete
  1. Register RCU request 'A' into open batch
  2. Register RCU request 'B' into open batch
  3. Close current open batch and start new grace period
  4. Register RCU request 'C' into open batch
  5. Register RCU request 'D' into open batch
  6. Grace period completed, execute requests 'A' and 'B' of closed batch



Global Grace Periods

- Grace periods are measured globally for all CPUs
  - Maintaining per CPU request lists
  - One CPU starts next grace periods
  - CPU that executes quiescent state last, ends grace period
- Once a grace period has completed all CPUs can execute their own requests
  1. Start of next grace period 1
  2. End of grace period 1 and start of grace period 2
  3. End of grace period 2



Data Structures

- CPU-Global data:

`nr_curr_global`	number of current grace period
`cpumask`	bitfield of CPUs, that have to pass through a quiescent state for completion of current grace period
`nr_compl`	number of recently completed grace period
`next_pending`	flag, requesting another grace period

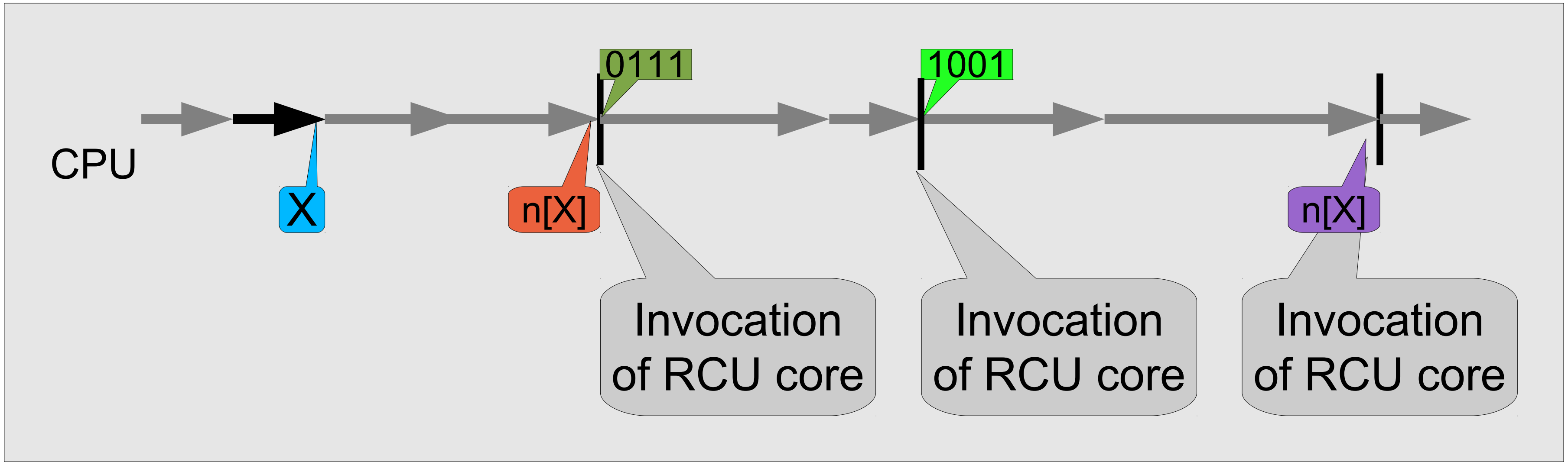
- CPU-local data:

`nr_curr_local`	local copy of global `nr_curr`
`qs_pending`	CPU needs to pass through a quiescent state
`qs_passed`	CPU has passed a quiescent state
`batch_closed`	***closed batch*** of RCU requests
`nr_batch`	grace period the closed batch belongs to
`batch_open`	***open batch*** of RCU requests

Components

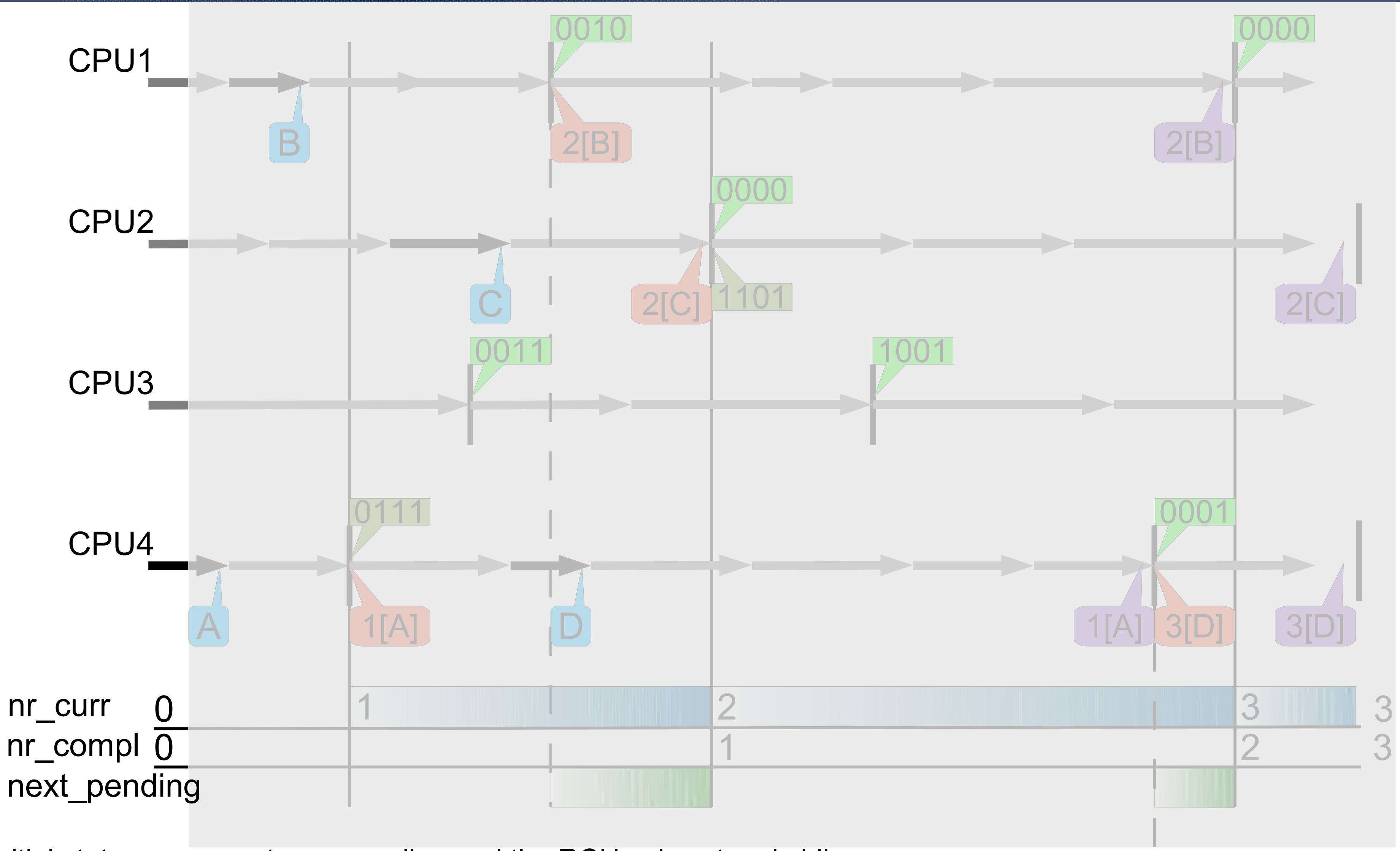
- Interface
  - `wait_for_rcu()` wait for grace period to complete
  - `call_rcu()` add RCU callback to open batch request list
- RCU core
  - Creates closed batch from open batch and assign grace period to be completed
  - Invokes callbacks in closed batch after grace period completed
  - Clear bit in CPU bitmask after quiescent state has detect
  - Requests new grace period, if required
  - Starts and finishes grace periods
- Timer-interrupt handler and scheduler
  - Detect quiescent states
  - Update variable CPU-local `qs_passed` of CPU
  - Schedule RCU core if work is pending

Modelling of Batches and Grace Periods



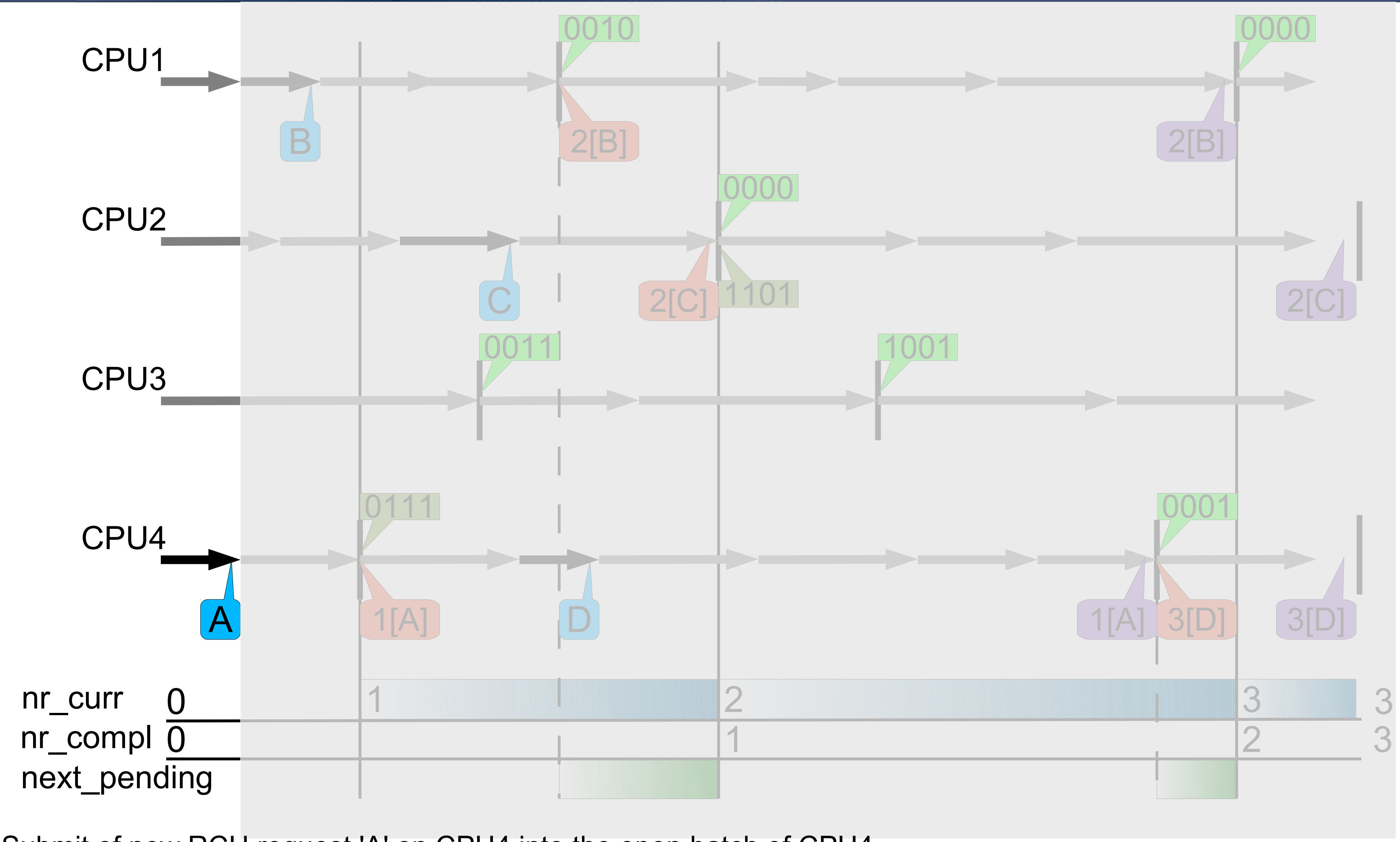
- Explanation:
  - X — Insertion of a callback *X* into the open batch
  - n[X] — move requests from the open batch to the closed batch; the closed batch can be processed after grace period *n* has elapsed
  - n[X] — grace period *n* has been elapsed and the corresponding closed batch can be processed
  - 0111 — Start new grace period and reset CPU bitmask
  - 1001 — Set bit to 0 for this CPU in CPU bitmask

Linux RCU Example (1)



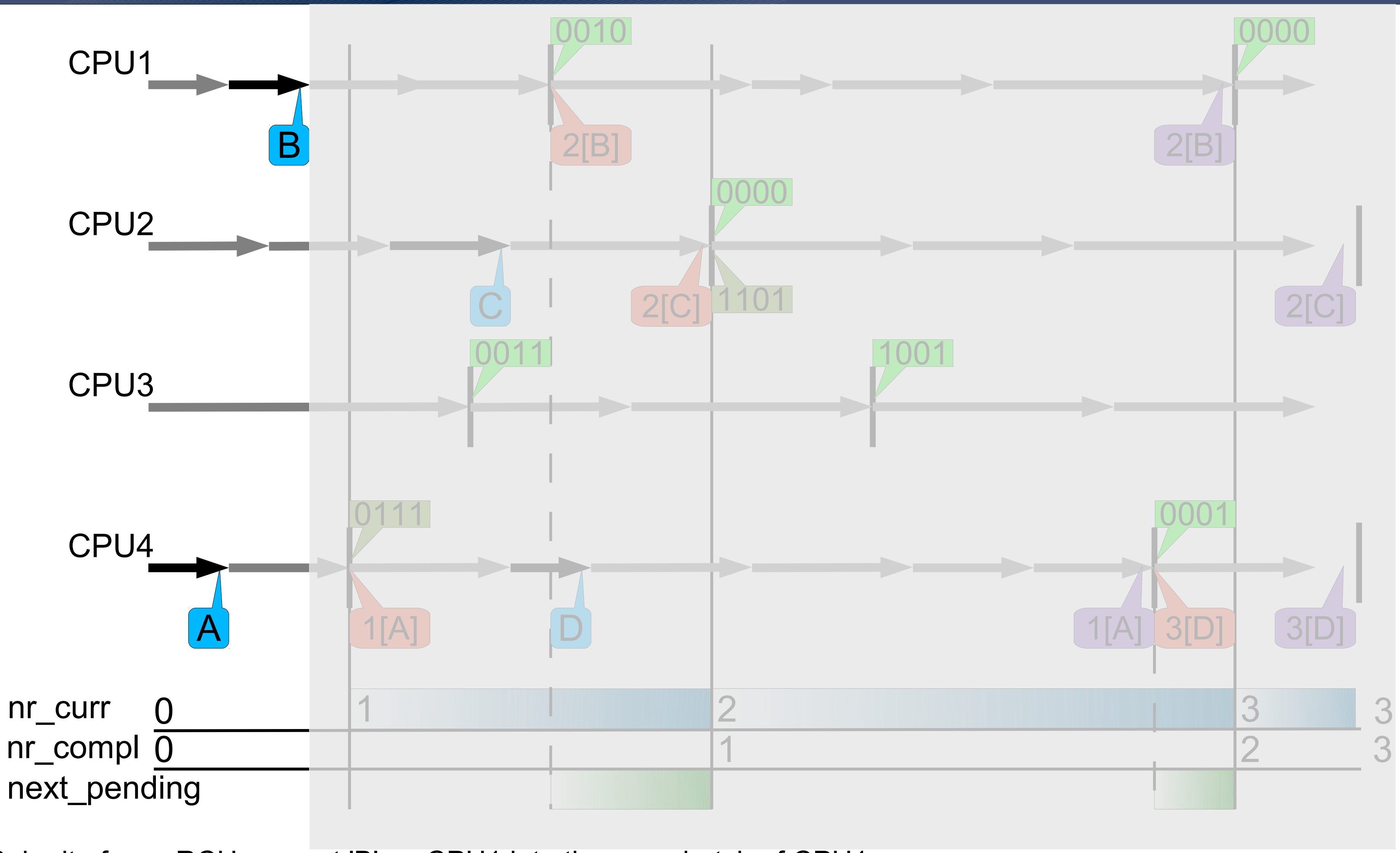
Initial state, no requests are pending and the RCU subsystem is idle

Linux RCU Example (2)



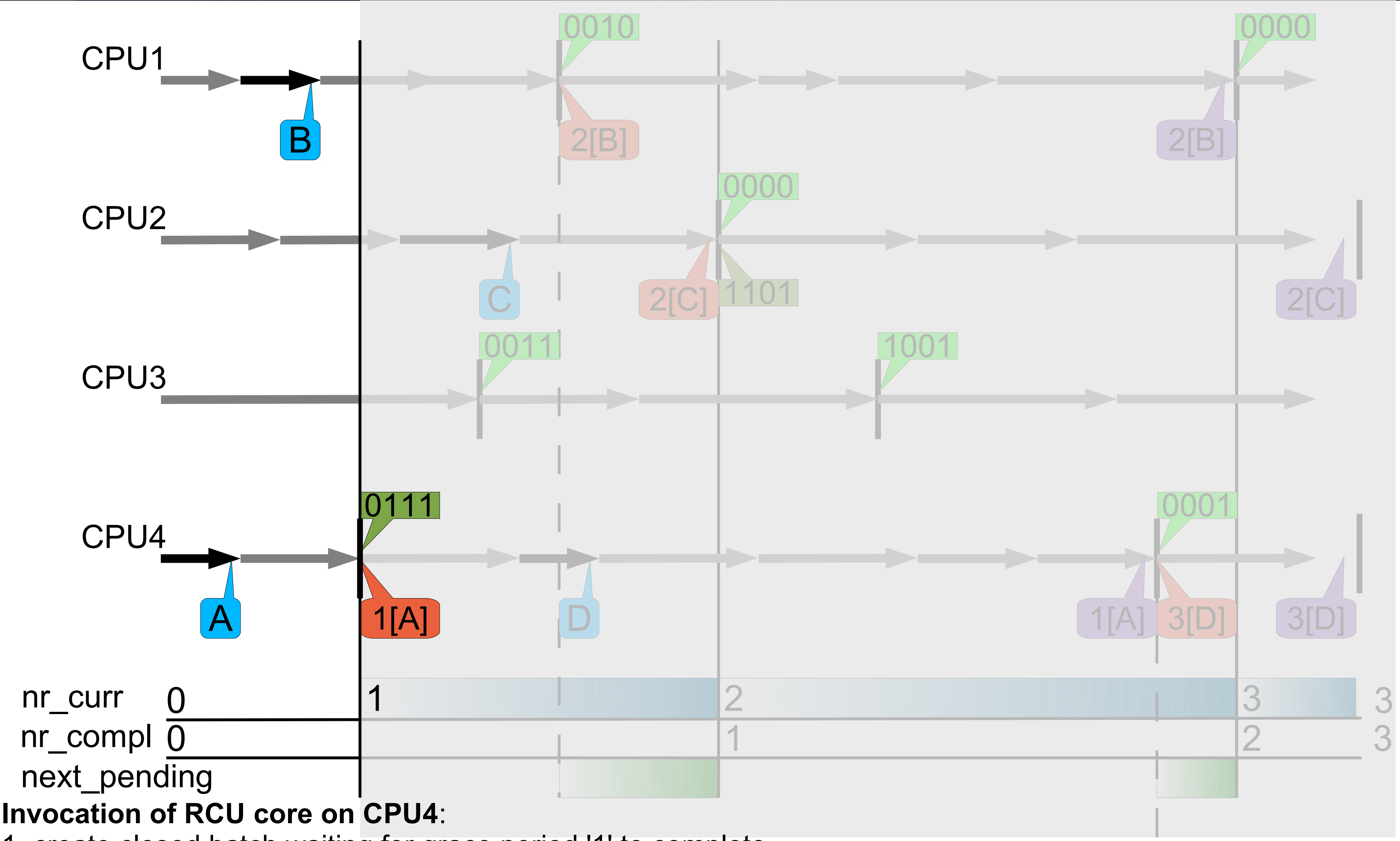
Submit of new RCU request 'A' on CPU4 into the open batch of CPU4

Linux RCU Example (3)



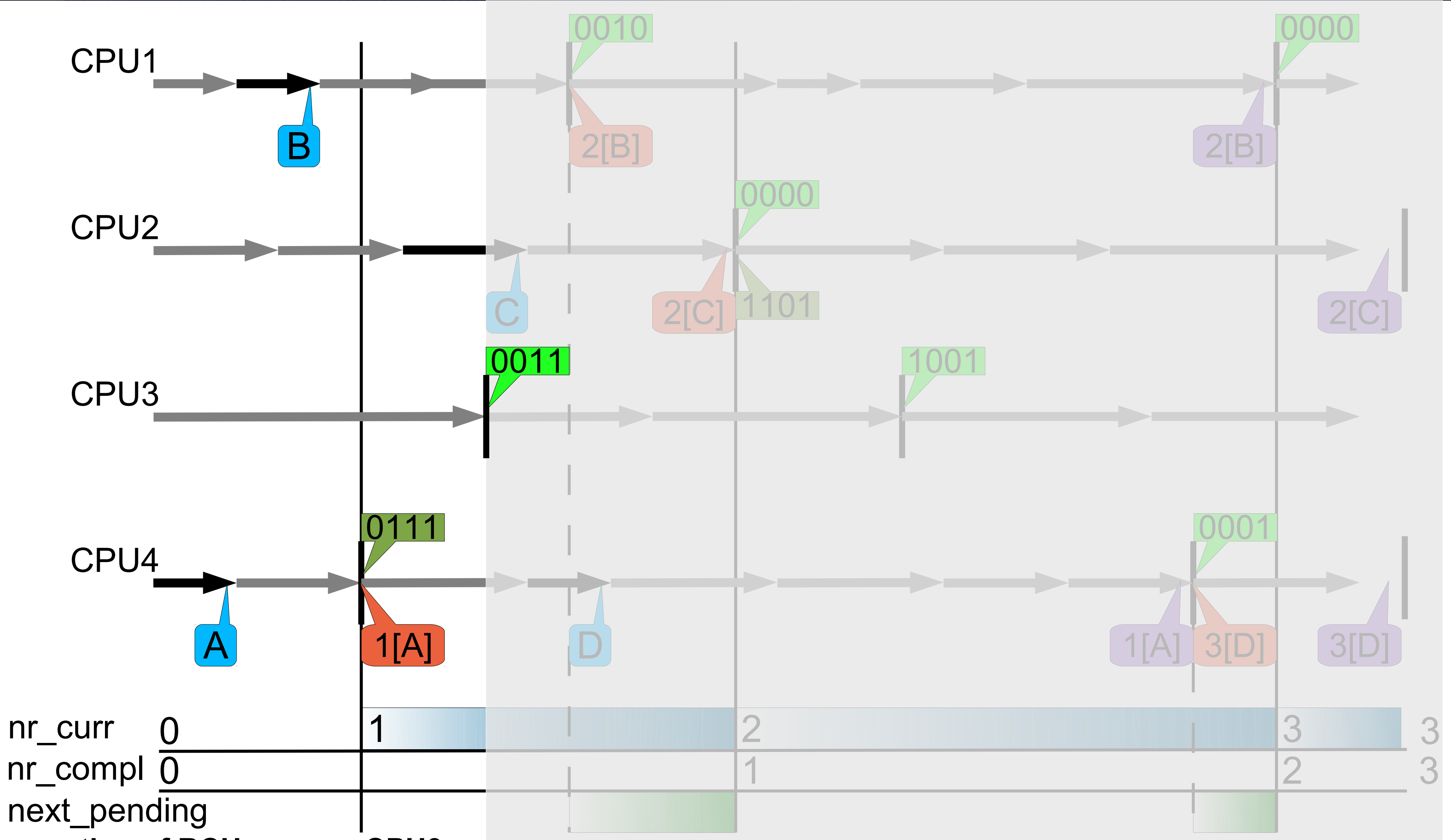
Submit of new RCU request 'B' on CPU1 into the open batch of CPU1

Linux RCU Example (4)



Invocation of RCU core on CPU4:
1. create closed batch waiting for grace period '1' to complete
2. start of new grace period '1' and set bitmask to wait for quiescent states

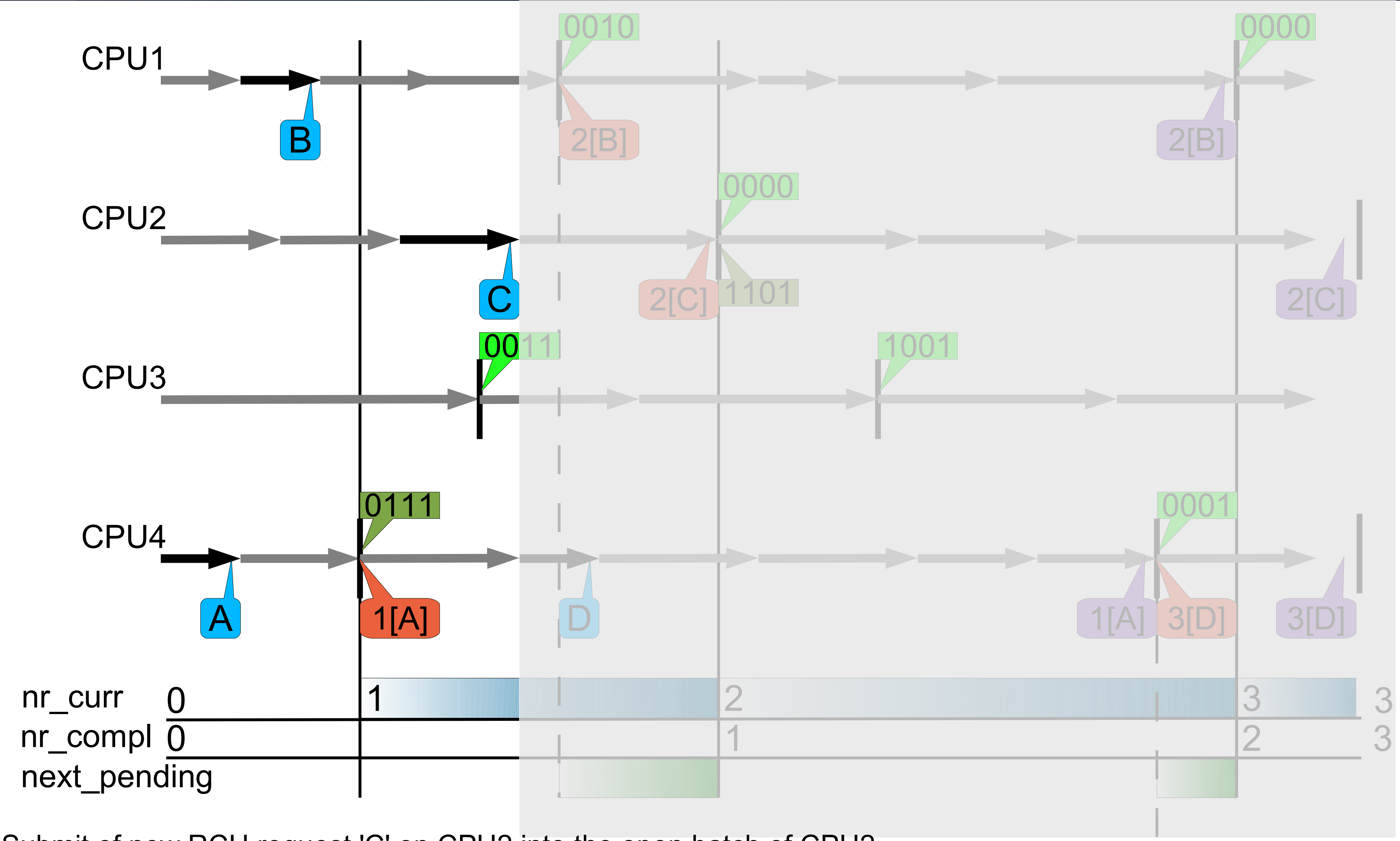
Linux RCU Example (5)



Invocation of RCU core on CPU3:

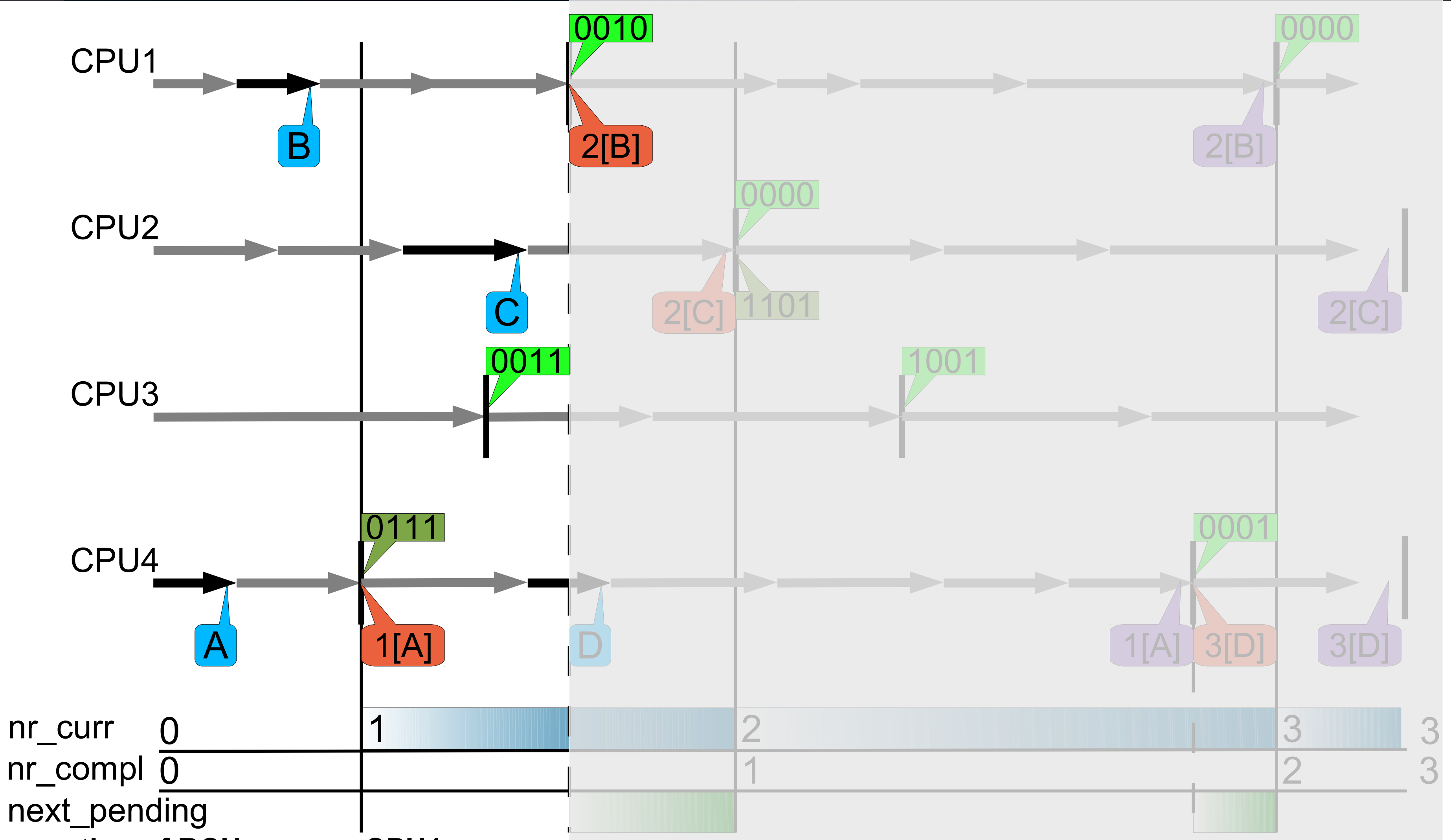
1. quiescent state detected, clear CPU bit in bitmask for grace period '1'

Linux RCU Example (6)



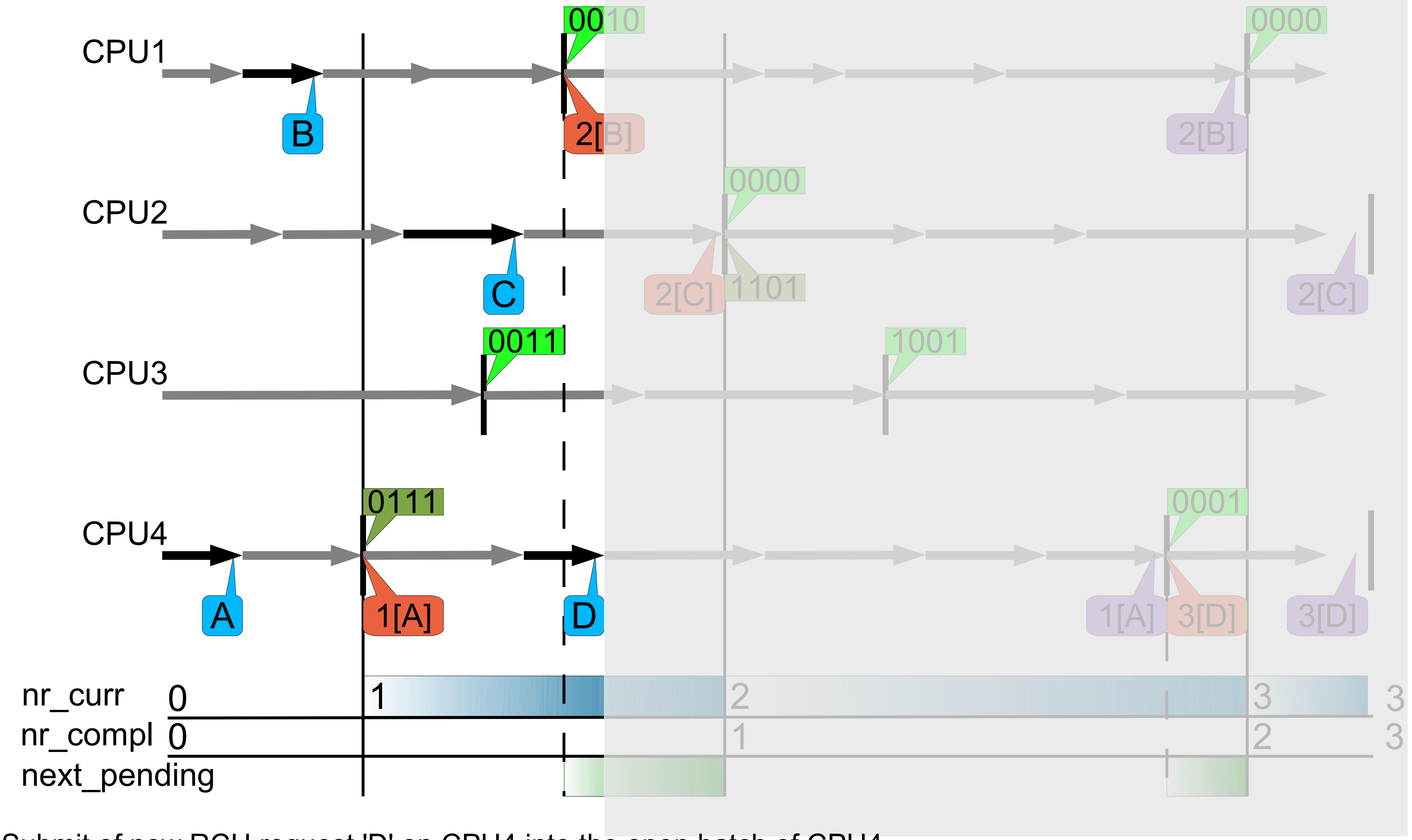
Submit of new RCU request 'C' on CPU2 into the open batch of CPU2

Linux RCU Example (7)



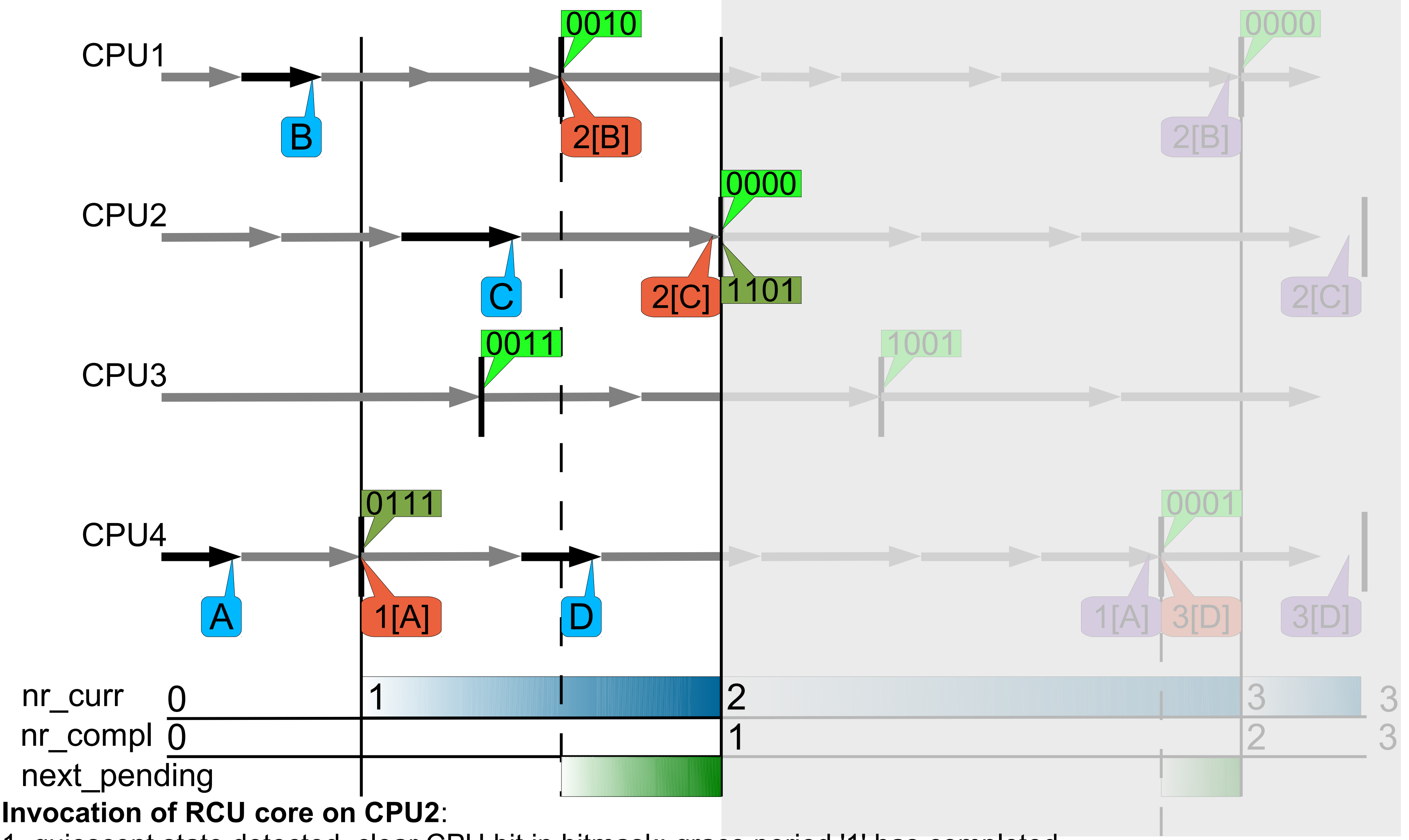
Invocation of RCU core on CPU1:
1. quiescent state detected, clear CPU bit in bitmask for grace period '1'
2. create closed batch waiting for grace period '2' to complete
3. request another grace period

Linux RCU Example (8)



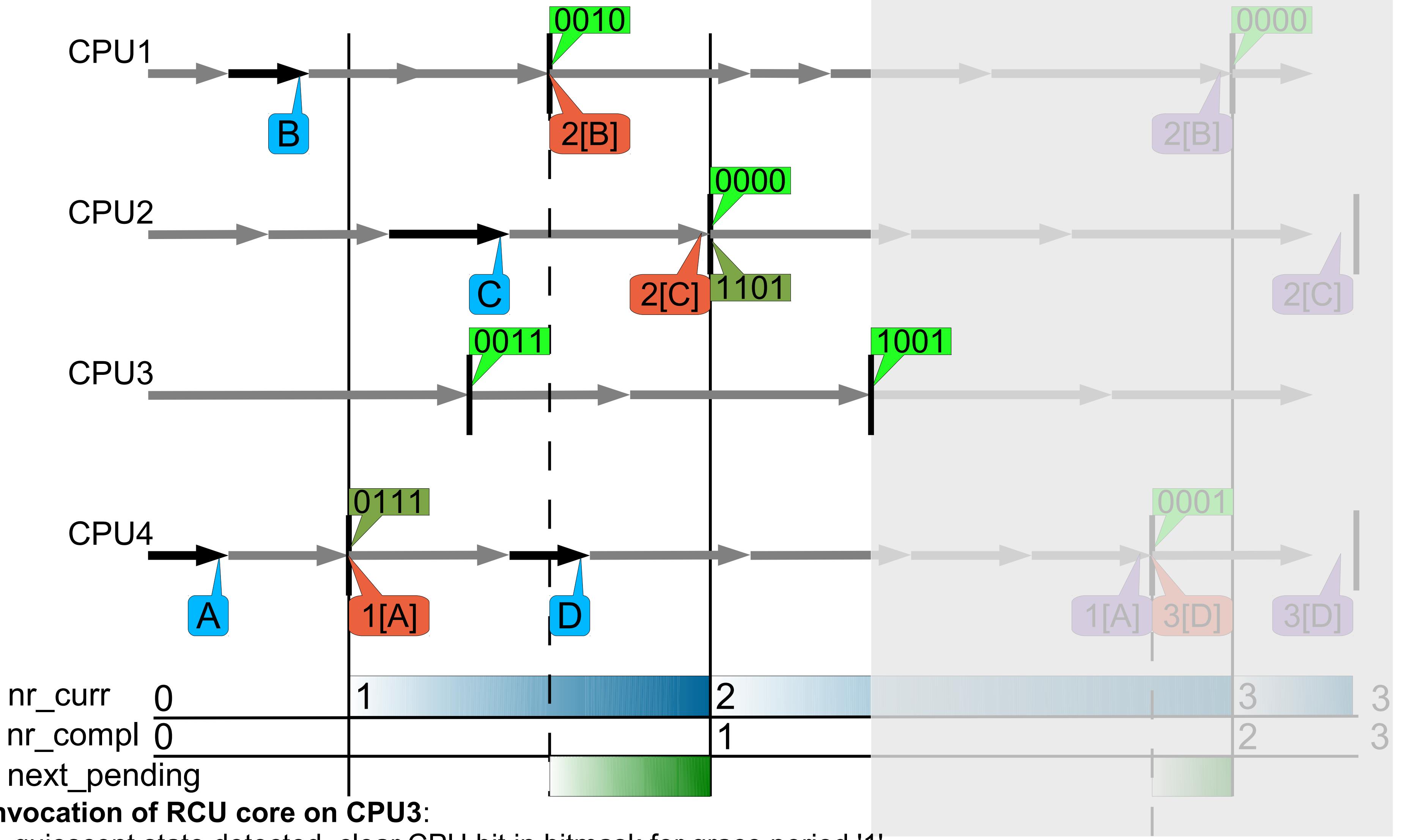
Submit of new RCU request 'D' on CPU4 into the open batch of CPU4

Linux RCU Example (9)



Invocation of RCU core on CPU2:
1. quiescent state detected, clear CPU bit in bitmask; grace period '1' has completed
2. create closed batch waiting for grace period '2' to complete
3. start new grace period '2'

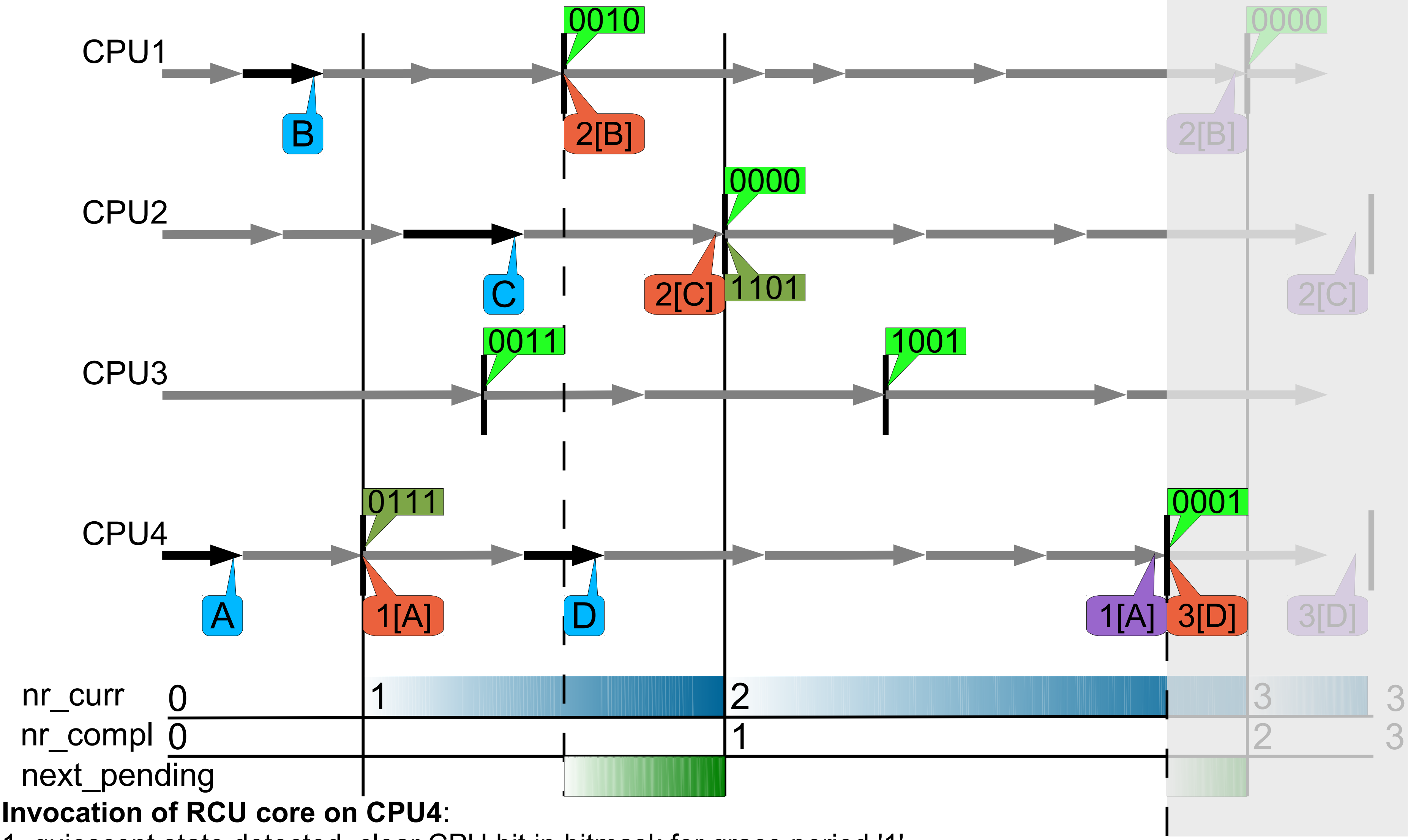
Linux RCU Example (10)



Invocation of RCU core on CPU3:

1. quiescent state detected, clear CPU bit in bitmask for grace period '1'

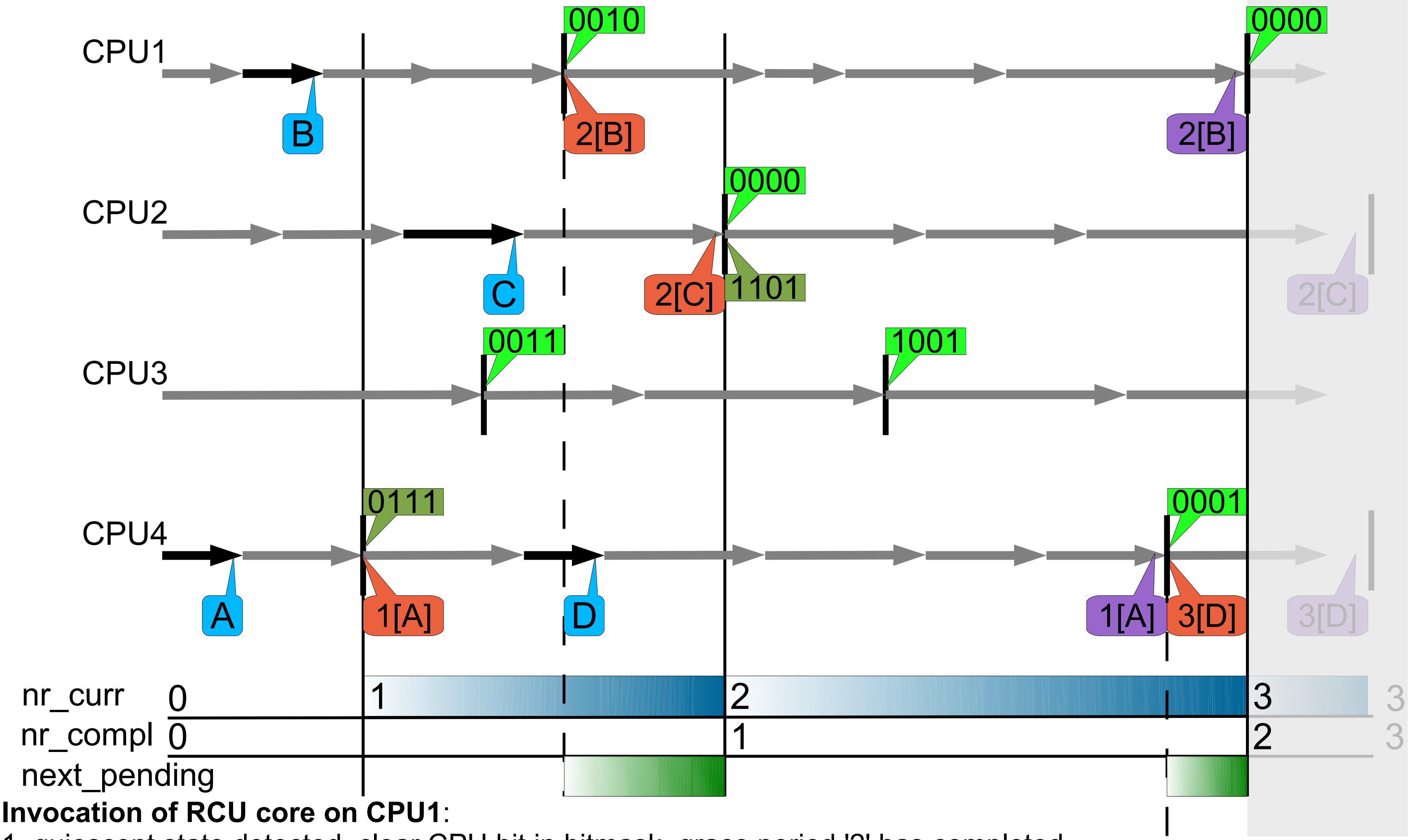
Linux RCU Example(11)



Invocation of RCU core on CPU4:

1. quiescent state detected, clear CPU bit in bitmask for grace period '1'
2. process closed batch for grace period '1'
3. create closed batch waiting for grace period '3' to complete

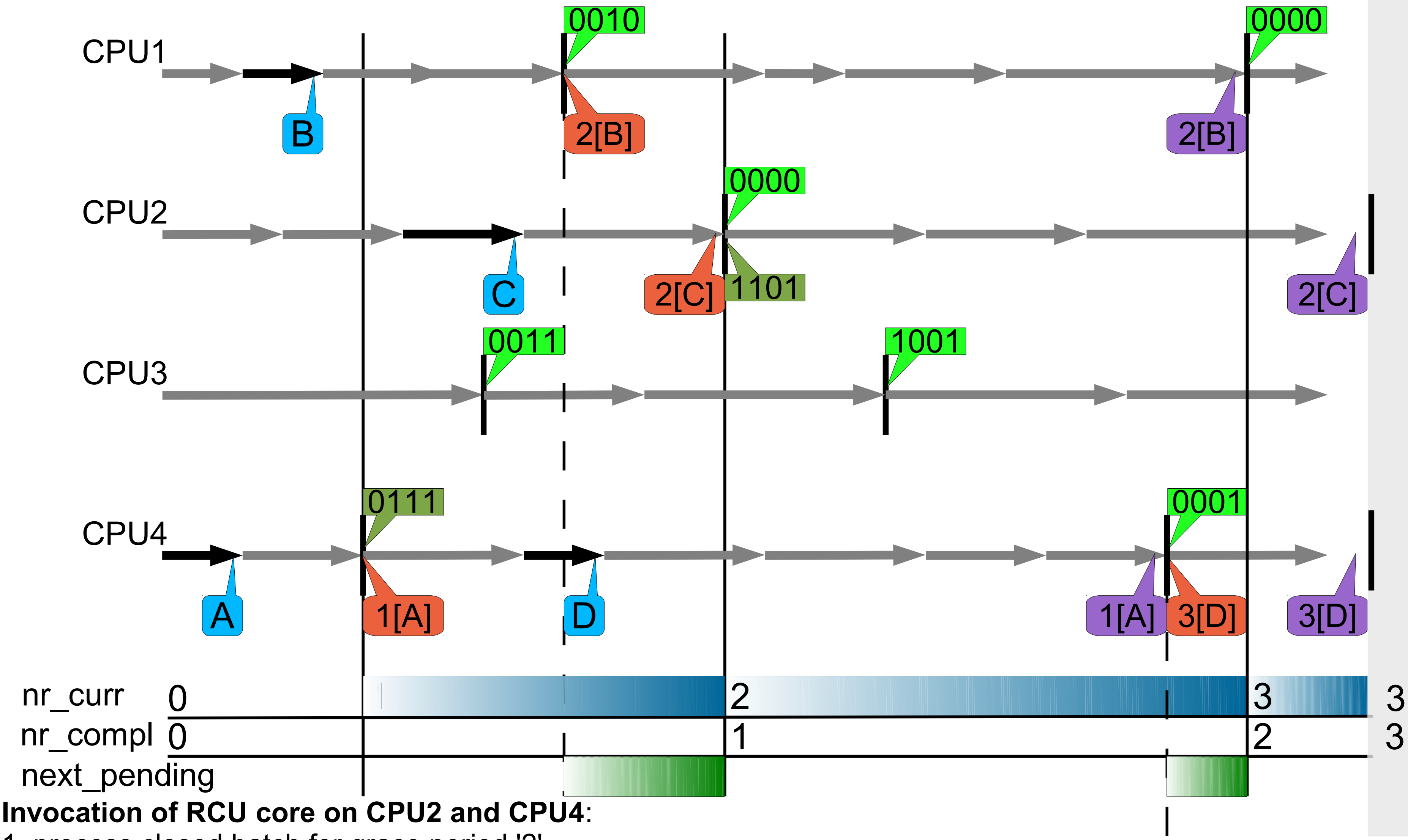
Linux RCU Example (12)



Invocation of RCU core on CPU1:

1. quiescent state detected, clear CPU bit in bitmask, grace period '2' has completed
2. process closed batch for grace period '2'

Linux RCU Example (13)



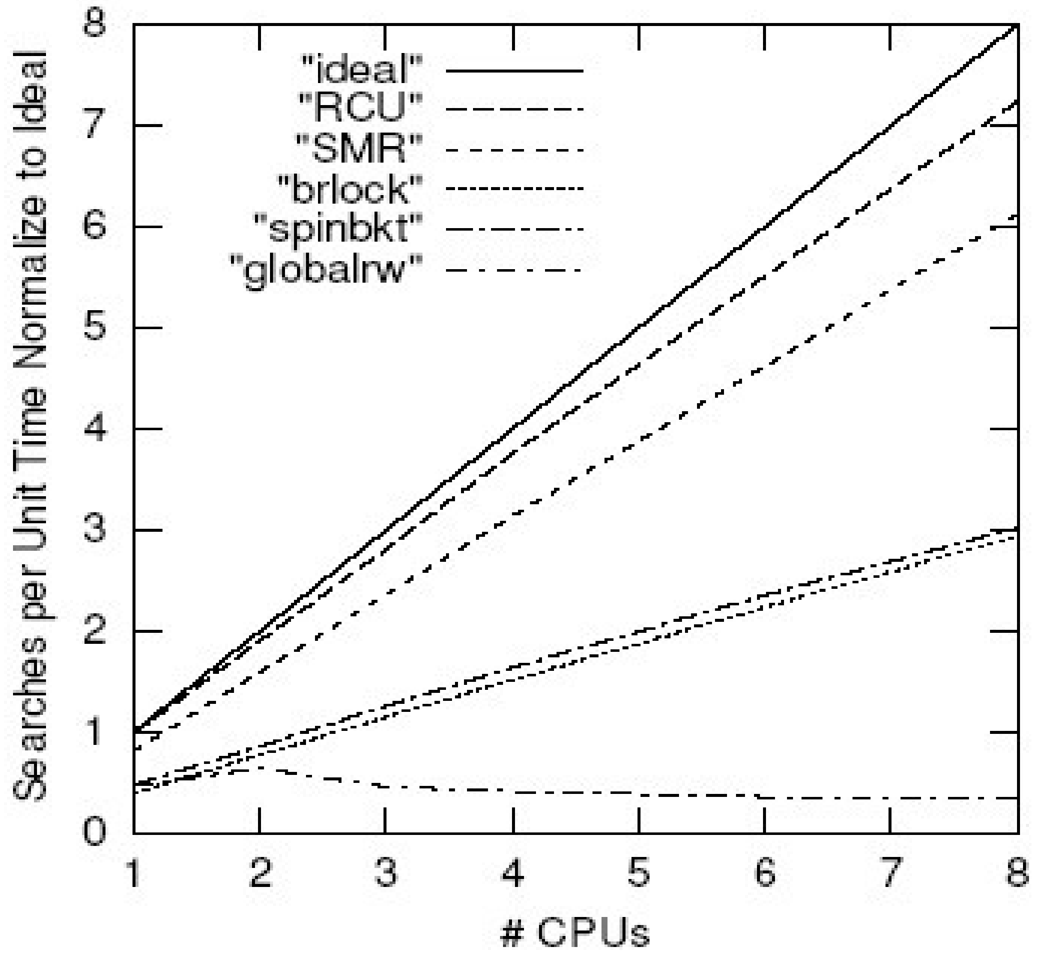
Invocation of RCU core on CPU2 and CPU4:
1. process closed batch for grace period '2'

Scalability and Performance

- How does RCU scale?
  - Number of CPUs (n)
  - Number of read-only operations
- How does RCU perform?
  - Fraction of accesses that are updates (f)
  - Number of operations per unit
- What other algorithms to compare to?
  - Global reader-writer lock (*globalrw*)
  - Per-CPU reader-writer lock (*brlock*)
  - Data spinlock (*spinbkt*)
  - Lock-free using safe memory reclamation (*SMR*)

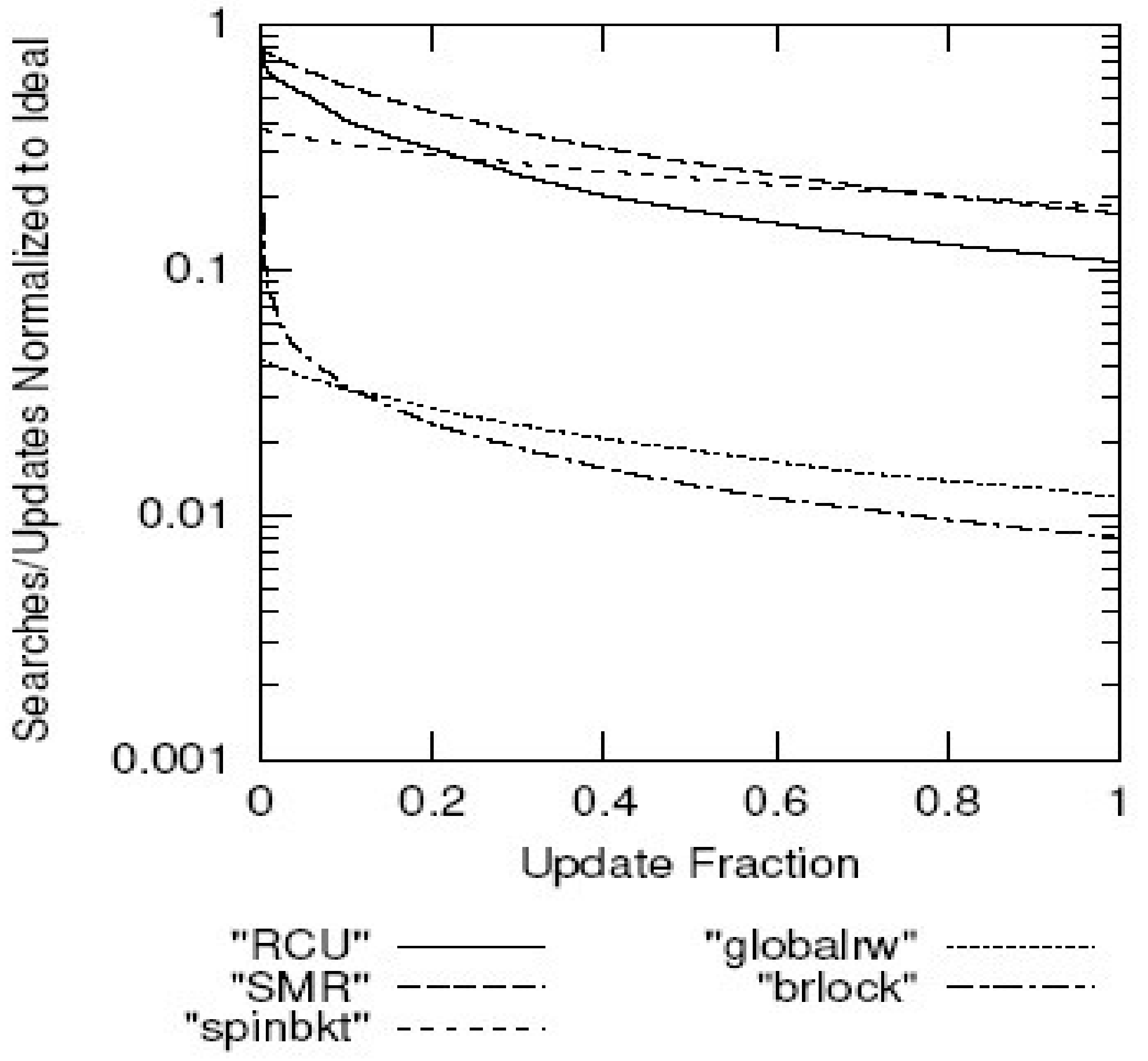
Scalability [2]

- Hashtable benchmark
  - Reading entries in a hashtable



Performance [2]

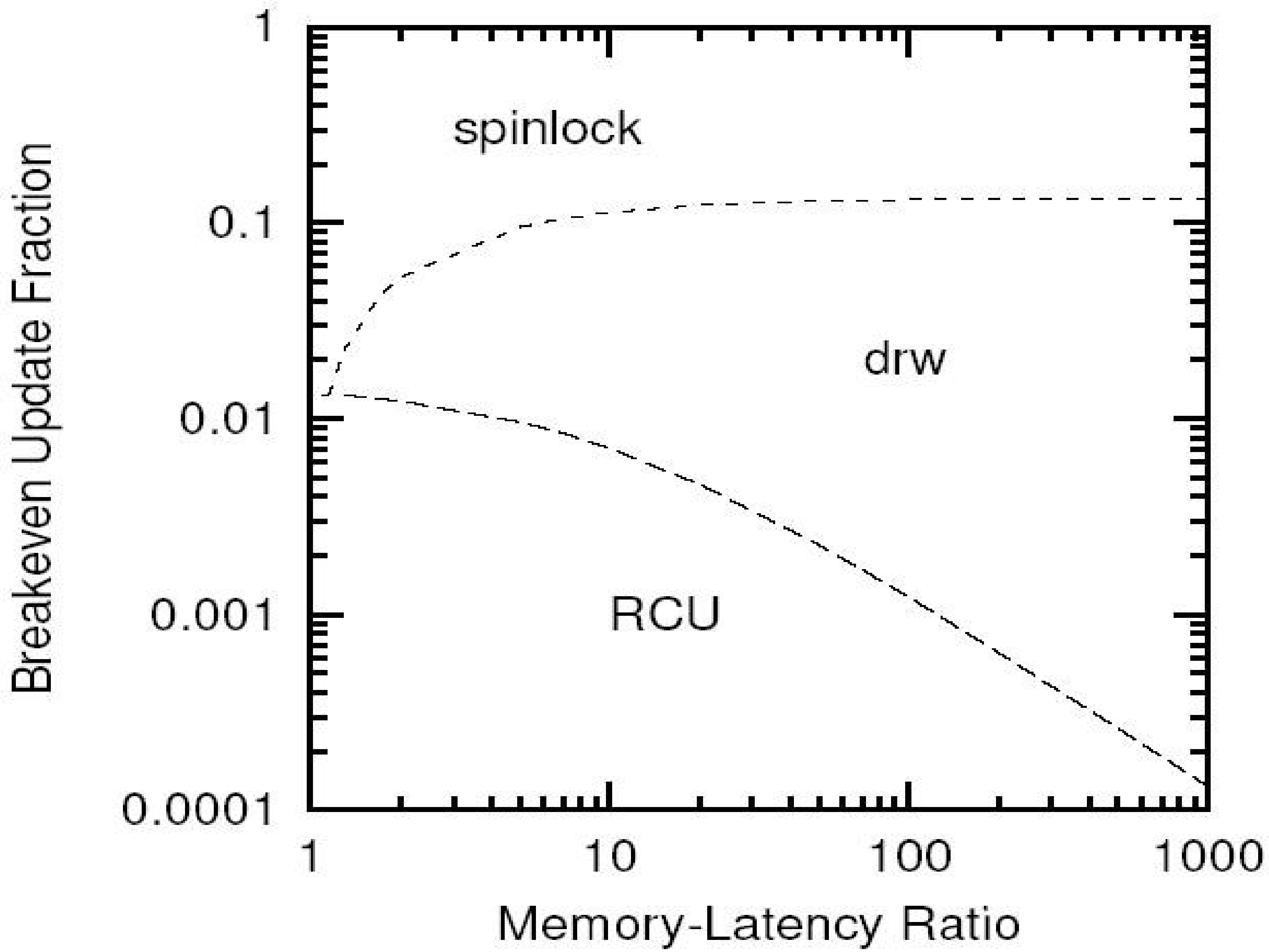
- Changing entries in a hashtable with 4 CPUs



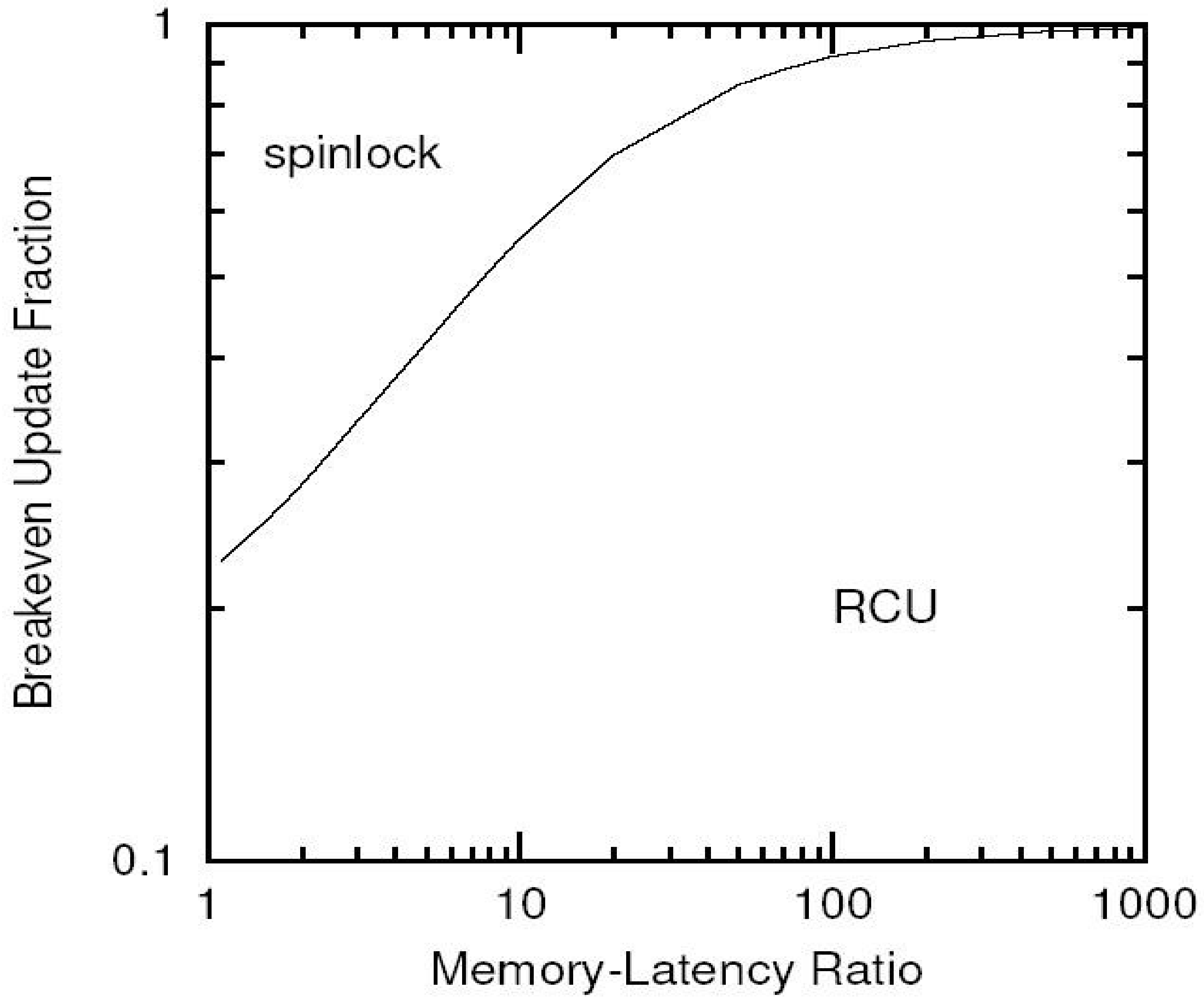
Performance vs. Complexity

- When should RCU be used?
  - Instead of simple spinlock? (spinlock)
  - Instead of per-CPU reader-writer lock? (drw)
- Under what conditions should RCU be used?
  - Memory-latency ratio (r)
  - Number of CPUs (n = 4)
- Under what workloads?
  - Fraction of accesses that are updates (f)
  - Number of updates (batch size) per grace period (λ = {small, large})

Few Updates per Grace Period [2]



Many Updates per Grace Period [2]



Concluding Remarks

- RCU performance and scalability
  - Near-optimal scaling with increasing number of CPUs for read-only workloads
  - Performance depends on many factors
- RCU modifications
  - Support for weak consistency models
  - Support for NUMA architectures
  - Without stale data tolerance
  - Support for preemptible read-side critical sections
  - Support for CPU hotplugging
- Other memory reclamation schemes
  - Lock-free reference counting
  - Hazard-pointer-based recalamation
  - Epoch-based reclamation

References

- [1] Read-Copy Update: Using Execution History to Solve Concurrency Problems; McKenney, Slingwine; 1998
- [2] Read-Copy Update; McKenney, Karma, Arcangeli, Krieger, Russel; 2003
- [3] Making Lockless Synchronization Fast: Performance Implications of Memory Reclamation; Hart McKenney; Brown; 2006
- [4] Linux Journal: Introduction to RCU; McKenney 2004; http://linuxjournal.com/article/6993
- [5] Linux Journal: Scaling dcache with RCU; McKenney; 2004; http://linuxjournal.com/arcticle/7124