Distributed Operating Systems SS2009

Multiprocessor Synchronization using Read-Copy Update

Torsten Frenzel

TU Dresden Operating Systems Group

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

Torsten
Frenzel

TU Dresden
Operating
Systems Group

Introduction

- Multiprocessor OS's need to synchronize access to data structures
- → Fast synchronization primitives are crucial for performance and scalability
- Two important facts in OSs
  - Small critical sections (, that access data structures)
  - Data structures with many reads and few writes (updates)
- Goals
  - Reducing synchronization overhead
  - Reducing lock contention
  - Deadlock avoidance

Torsten
Frenzel

TU Dresden
Operating
Systems Group

Synchronization Primitives

- Coarse-grained locking (code-based locks)
  - Spinlock (called '*Big kernel lock*' in Linux)
  - Reader-writer lock (called '*Big reader lock*' in Linux)
- Fine-grained locking (data-based locks)
  - Spinlock
  - Reader-writer lock
  - Per-cpu reader-writer lock
- Lock-free synchronization
  - ➔ Fine grained
  - Use atomic operations to update data structures
  - Avoids disadvantages of locks
  - Hard (to do right) for complex data structures

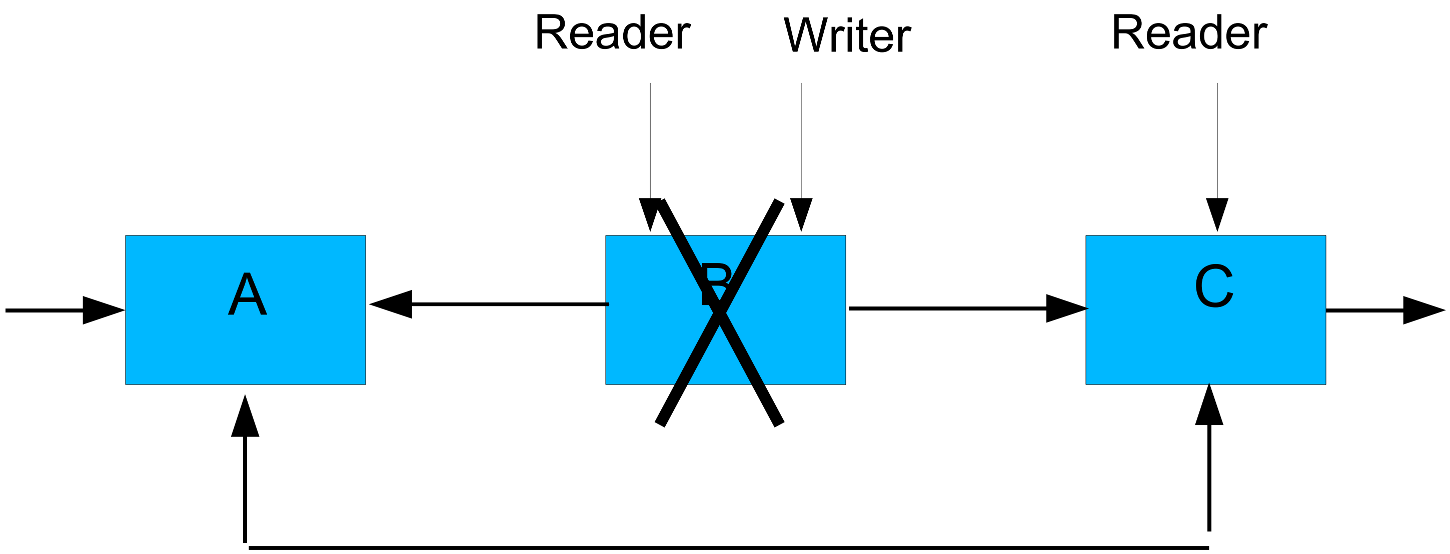
Lockless Synchronization

- Idea
  - No locks on reader side
  - Locks only on writer side (no concurrent update operations)
  - **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

Torsten
Frenzel

TU Dresden
Operating
Systems Group

Two-Phase Update - Example



Update Phase

Wait period

Reclamation Phase

Two-Phase Update - Principle

- Phase 1: Update Phase
  - Change data structure and make new state visible
- Wait period:
  - Allow existing read operations to proceed on the old state until completed
- Phase 2: Reclamation Phase
  - Remove old (invisible) state of data structure
- *Problem:*
  - **When to reclaim memory after update?**
  - **How long to wait?**
- **Read-Copy Update uses pessimistic approach:**

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

  → **Deferred memory reclamation**

Applications

- Scenarios
  - File descriptor table
  - Routing cache
  - Network subsystem policy changes
  - Hardware configuration
  - Module unloading
- Implementation
  - DYNIX
    - UNIX-based operating system from Sequent
  - Tornado
    - Operating system for large scale NUMA architectures
  - K42
    - Operating system from IBM for large scale architectures
  - Linux

Example 1: List – Read Operation

```
void read(long addr)
{
  read lock(&list_lock);
  struct elem *p = head->next;
  while (p != head)
  {
    if (p->address == addr)
    {
      /* read-only access to p */
      read unlock(&list_lock);
      return;
    }
    p = p->next;
  }
  read unlock(&list_lock);
  return;
}
```

```
void read(long addr)
{

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

      return;
    }
    p = p->next;
  }

  return;
}
```

Example 1: List – Delete Operation

```
void delete(struct elem *p)
{
  struct elem *p = head→next;
  write_lock(&list_lock);
  while (p != head)
  {
    if (p→address == addr)
    {
      p→next→prev = p→prev;
      p→prev→next = p→next;
      write_unlock(&list_lock);

      kfree(p);
      return;
    }
    p = p→next;
  }
  write_unlock(&list_lock);
  return;
}
```

```
void delete(struct elem *p)
{
  struct elem *p = head→next;
  spin_lock(&list_lock);
  while (p != head)
  {
    if (p→address == addr)
    {
      p→next→prev = p→prev;
      p→prev→next = p→next;
      spin_unlock(&list_lock);
      wait_for_rcu();
      kfree(p);
      return;
    }
    p = p→next;
  }
  spin_unlock(&list_lock);
  return;
}
```

Example 2: File-descriptor Table

- 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);
```

Grace Periods and Quiescent States

- Definition of a grace period
  - *Intuitive*: duration until references to data are no longer hold 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*: induce quiescent state into CPU
  - *Detection*: Wait until CPU has passed through quiescent state

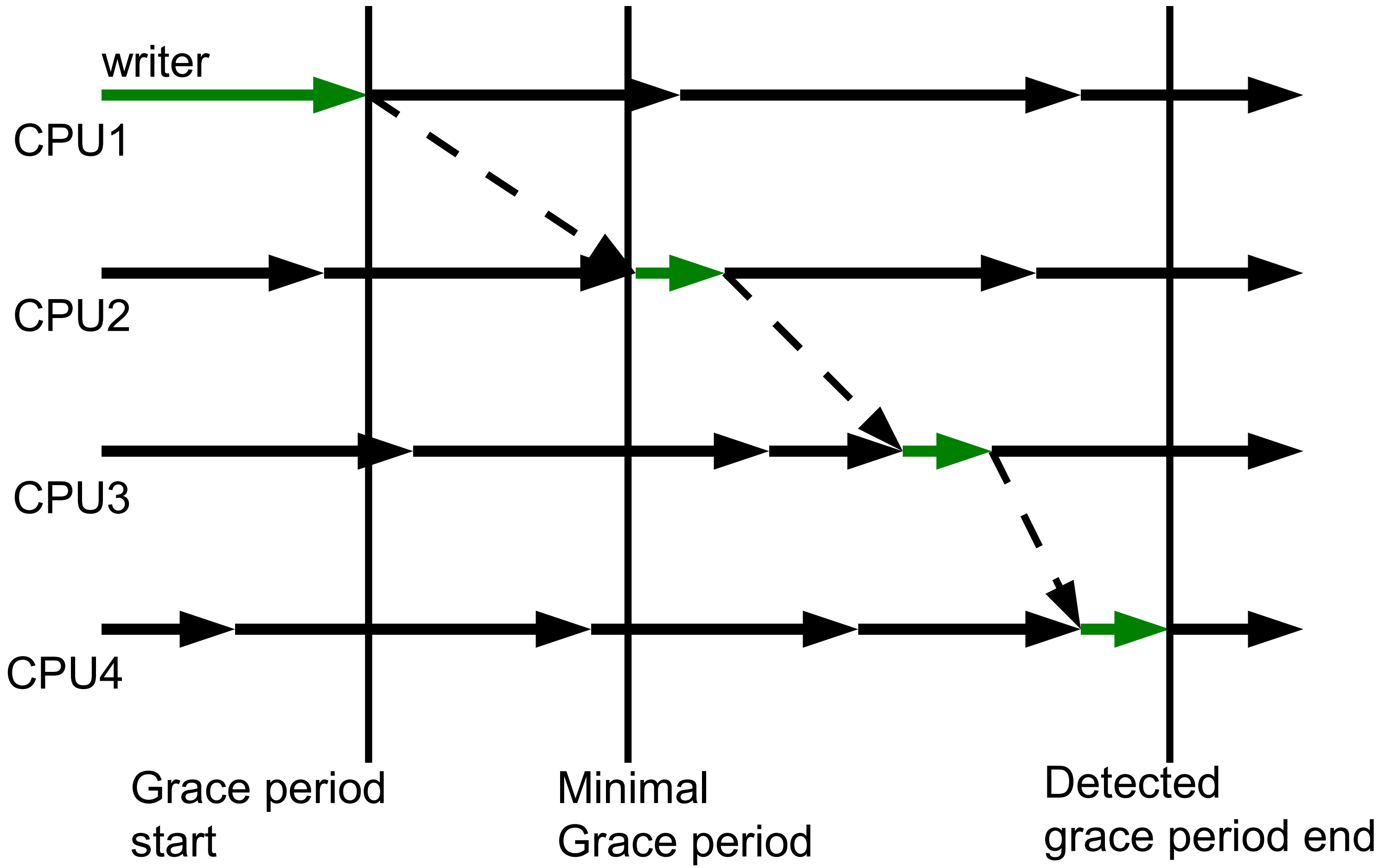
Torsten
Frenzel

TU Dresden
Operating
Systems Group

Quiescent State

- What are good quiescent states?
  - Should be easy to detect
  - Should occur not to frequent or infrequent
- **Per-CPU granularity**
  - For example: context switch, execution in idle loop, kernel entry/exit, CPU goes offline
  - OSs without blocking and preemption in read-side critical sections
- Per-thread granularity
  - OSs with blocking and preemption in read-side critical sections

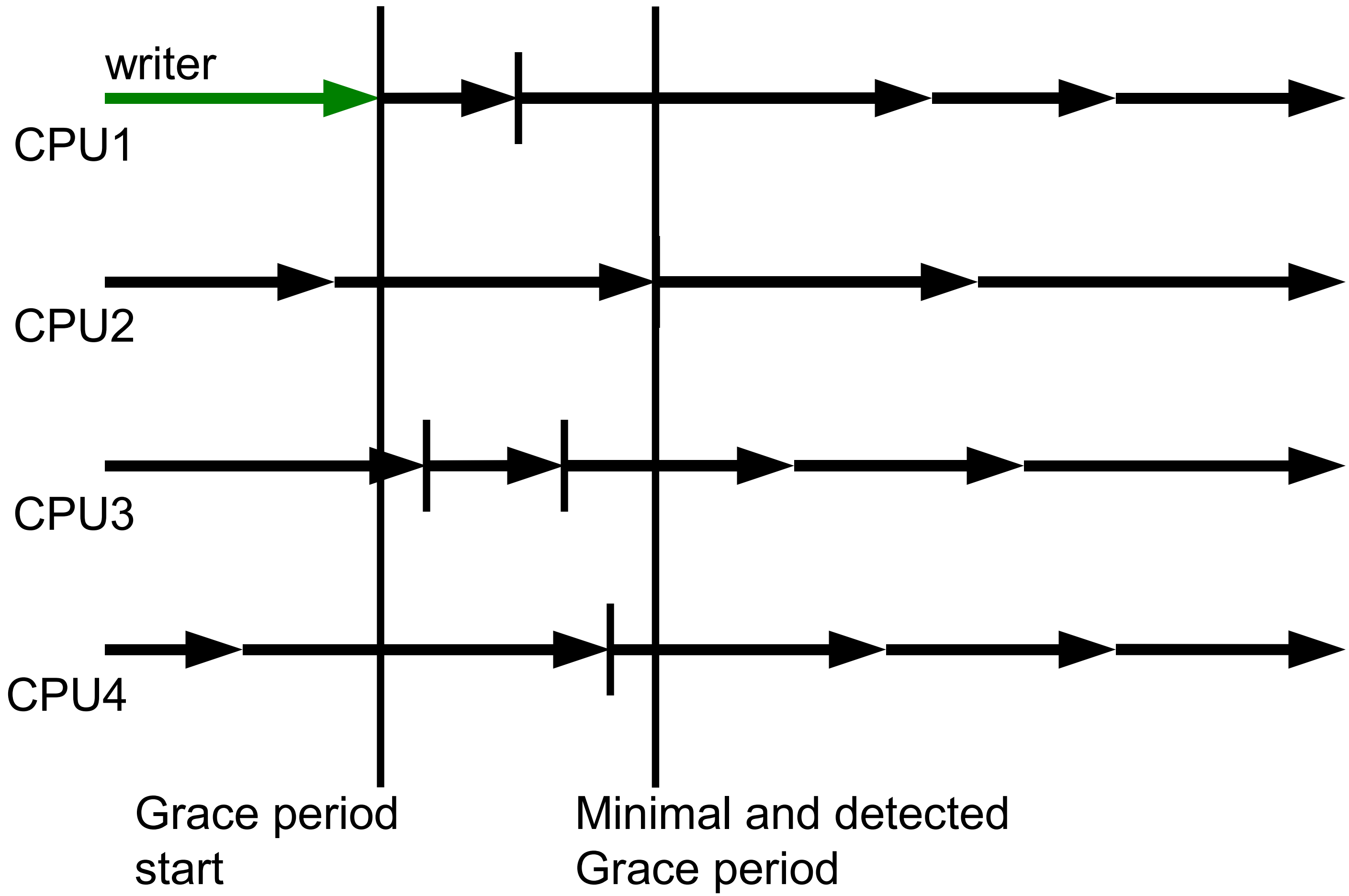
Quiescent State Enforement



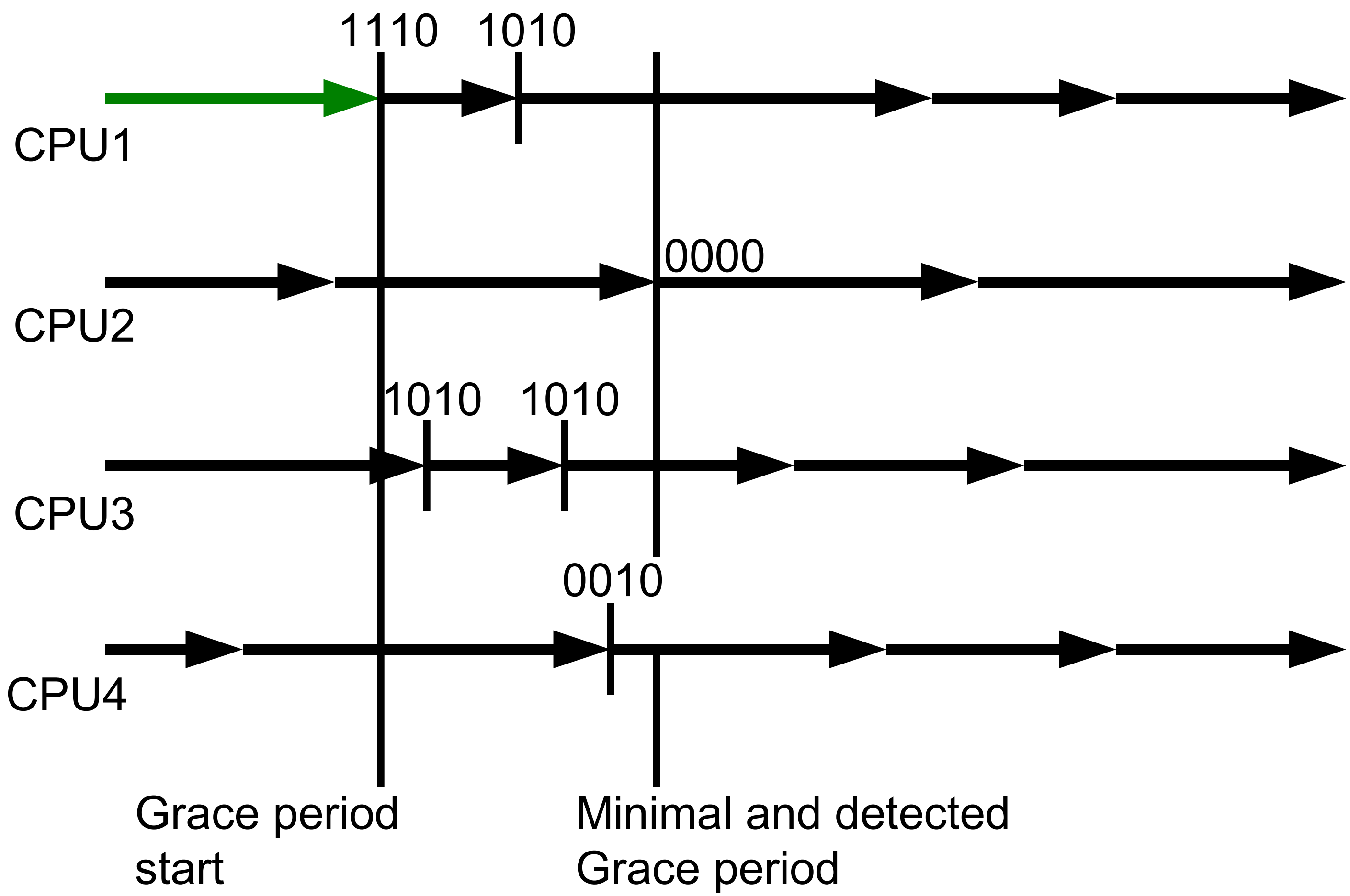
Torsten
Frenzel

TU Dresden
Operating
Systems Group

Quiescent State Detection



Quiescent State Bitmask



Enhancing RCU

- Two observations
  - Measuring grace periods adds overheads
  - Influence on system design
- Consequences
  - Batching of RCU requests
    - Single grace period can satisfy multiple requests
  - Maintaining per-CPU request lists
  - Callback functions for deferred memory reclamation
    - Avoids blocking
  - Low-overhead algorithm for measuring grace periods
  - Measurement framework for long-running critical sections

Torsten
Frenzel

TU Dresden
Operating
Systems Group

Linux's RCU Implementation

- **Batching** with per-CPU request list
  - ***Closed batch*** per CPU waiting for completion of (current or next) grace period
  - ***Open batch*** per CPU for new requests
- Separation of CPU-local and global data structures
  - CPU-local quiescent state detection and batch handling
  - Global grace period measurement with CPU-bitmask
- Low overhead if RCU system is idle
- Support for CPU hotplugging
- Support for preemptible read-side critical section
- Support for weak memory consistency

Torsten
Frenzel

TU Dresden
Operating
Systems Group

Data Structures

- Global data: `rcu_ctrlblk`

`nr_curr`	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: `rcu_data`

`nr_curr`	grace period this CPU thinks as current (should be equally global `nr_curr`)
`qs_pending`	CPU needs to pass through a quiescent
`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

Functional Separation

- Interface
  - `call_rcu()` add RCU callback to batch request list
  - `synchronize_rcu()` wait for grace period to complete
- Tasklet (implements RCU core)
  - Batch processing
    - Invokes callbacks after grace period
  - Finish and start new grace period
  - Quiescent state handling
- Timer-interrupt handler
  - Updates variable `qs_passed` of CPU
  - Schedules tasklet of RCU work is pending
- Scheduler
  - Updates variable `qs_passed` of CPU

Batch Processing

```
static void __rcu_process_callbacks(struct global_data *global,
                                    struct loca_data *local)
{
    if (not is_empty(local→batch_closed) and        /* Is the closed batch list not empty? */
       (global→nr_compl >= local→nr_batch))        /* Grace geriod this batch is waiting for completed? */
    {
        ... move closed batch list to completed batch ...
    }

    if (not is_empty(local→batch_open) and          /* Is the open batch full? */
        is_empty(local→batch_closed)                /* Is the closed batch empty? */
    {
        ... move open batch to closed batch ...
        local→nr_batch = global→nr_curr + 1;  /* After the next grace period has completed
                                                 this batch can be processed */

        if (not global→next_pending)                /* Is a new grace period aleady requested? */
        {
            global→next_pending = 1;                /* A new grace period has to be started */
            rcu_start_batch(global);                /* Try to start a new grace period immediately */
        }
    }

    rcu_check_quiescent_state(global, local);      /* Check if this CPU gone through a quiescent state */

     ... if there is a non-empty completed batch, process RCU callbacks ....
}
```

Quiescent State Handling

```
static void rcu_check_quiescent_state(struct global_data *global,
                                       struct local_data  *local)
{
        if (local→nr_curr != global→nr_curr) {       /* Has a new grace period started? */
                local→qs_pending = 1;                 /* Yes, Reset, for new grace period */
                local→qs_passed = 0;                  /* Reset, for new grace period */
                local→nr_curr = global→nr_curr;      /* Grace period this cpu is passing through */
                return;
        }

        if (!local→qs_pending)                       /* Is this cpu waiting for quiescent state */
                return;                               /* No, go on with work */

        if (!local→qs_passed)                        /* Has this cpu passed a quiescent state */
                return;                               /* No, come back later */

        local→qs_pending = 0;                         /* This cpu has passed through a quiescent state! */

        if (local→nr_curr == global→nr_curr)         /* sanity check */
                cpu_quiet(local→cpu, global);         /* update cpu bitmask and check if
                                                         grace period completed */
}
```

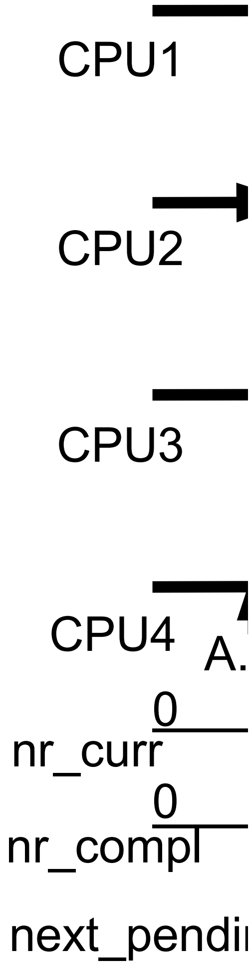
Finish and Start of Grace Period

```
static void cpu_quiet(int cpu, struct global_data *global)
{
    cpu_clear(cpu, global→cpumask);          /* Clear bit of this cpu in cpu bitmask */

    if (cpus_empty(global→cpumask))          /* Has a grace period completed? */
    {
        global→nr_compl = global→nr_curr;   /* Set completed to current grace period */
        rcu_start_batch(global);             /* Try to start a new grace period */
    }
}

static void rcu_start_batch(struct global_data *global)
{
    if (global→next_pending and              /* Should a new grace period be started? */
       global→nr_compl == global→nr_curr)   /* Is completed equal current grace period? */
    {
        global→next_pending = 0;             /* Reset grace period trigger */
        global→nr_curr++;                    /* A new global grace period starts */

                                             /* Update cpu bitmask */
        cpus_andnot(global→cpumask, cpu_online_map);
    }
}
```

When to invoke the RCU Core?

```
static int __rcu_pending(struct global_data *global, struct local_data *local)
{
        /* This cpu has pending rcu entries and the grace period
           for them has completed. */
        if (not is_empty(local→batch_closed) and
           global→nr_compl >= local→nr_batch)
              return true;

        /* This cpu has no pending entries, but there are new entries */
        if (is_empty(local→batch_closed) and
           not is_empty(local→batch_open)
              return true;

        /* This cpu has finished callbacks to invoke */

        /* The rcu core waits for a quiescent state from the cpu */
        if (local→nr_curr < global→nr_curr or local→qs_pending)
              return true;

        return false;
}
```

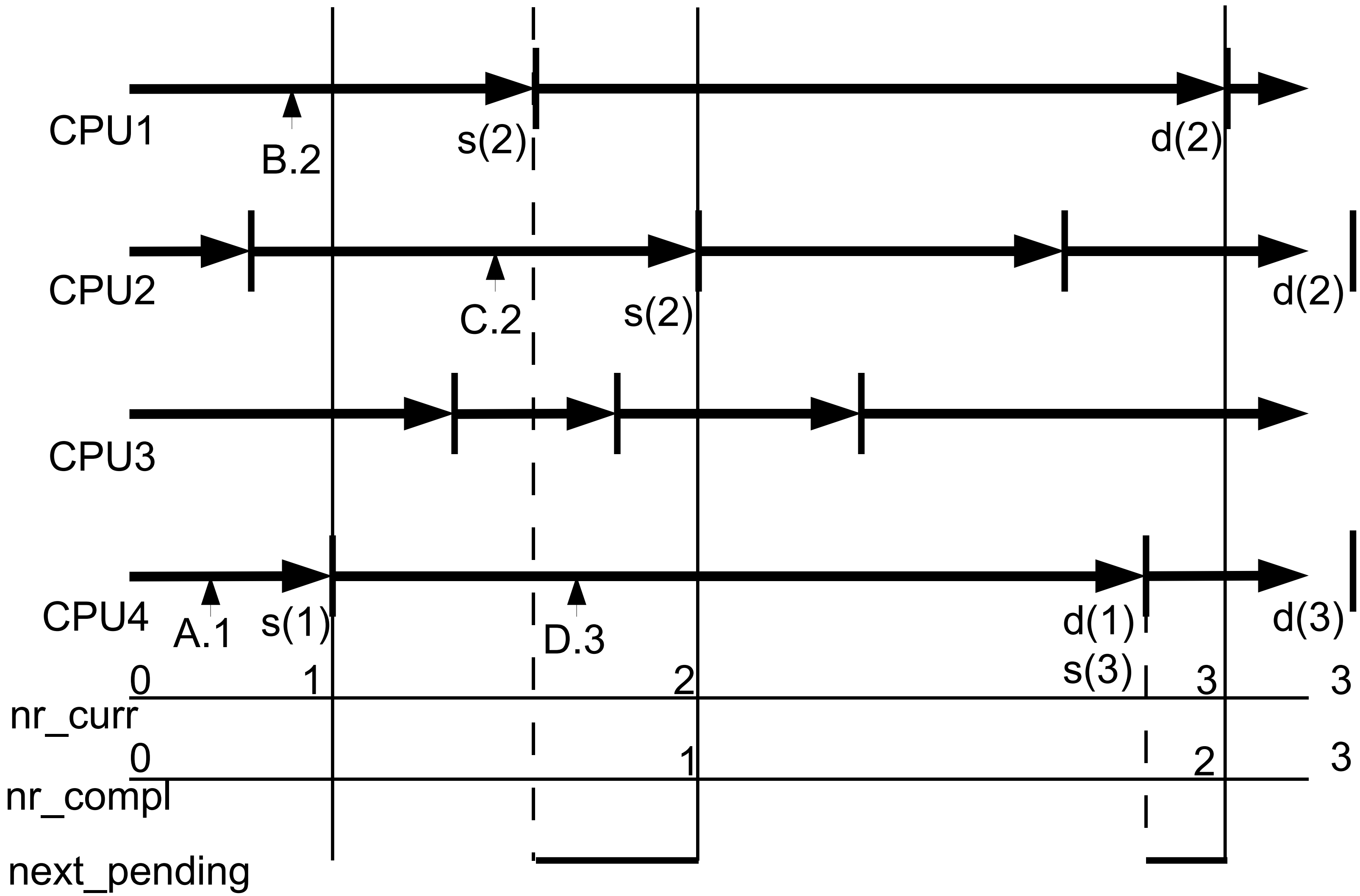
Linux RCU Example



Torsten
Frenzel

TU Dresden
Operating
Systems Group

Linux RCU Example

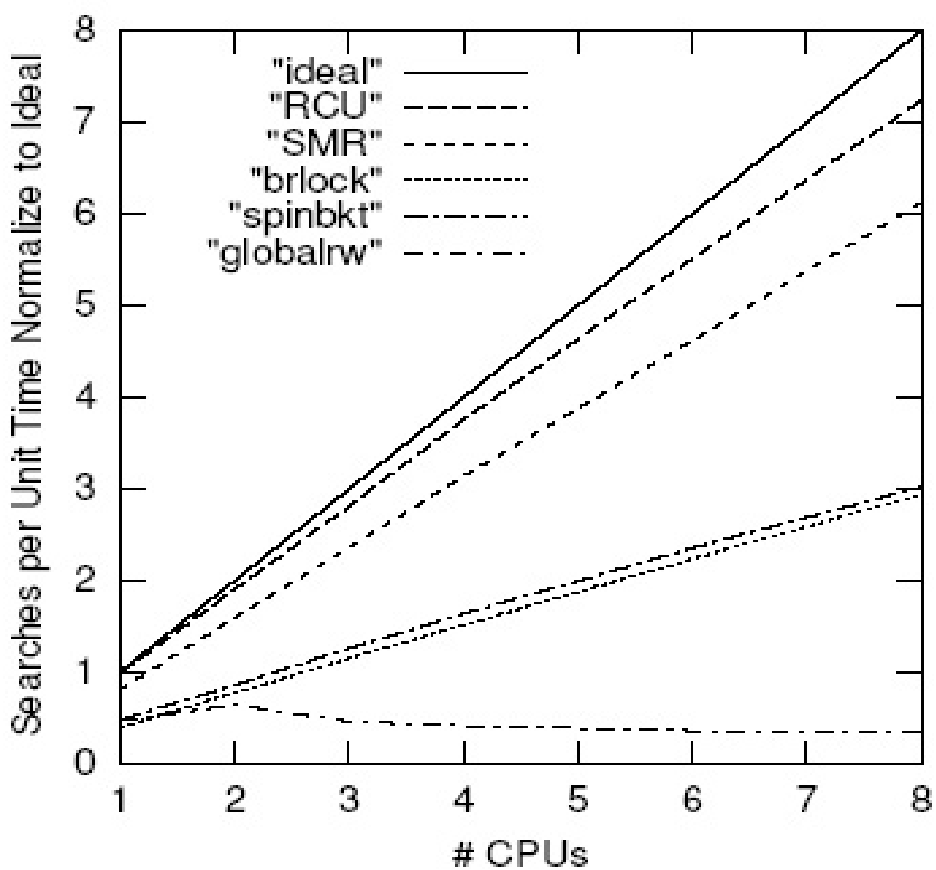


Scalability and Performance

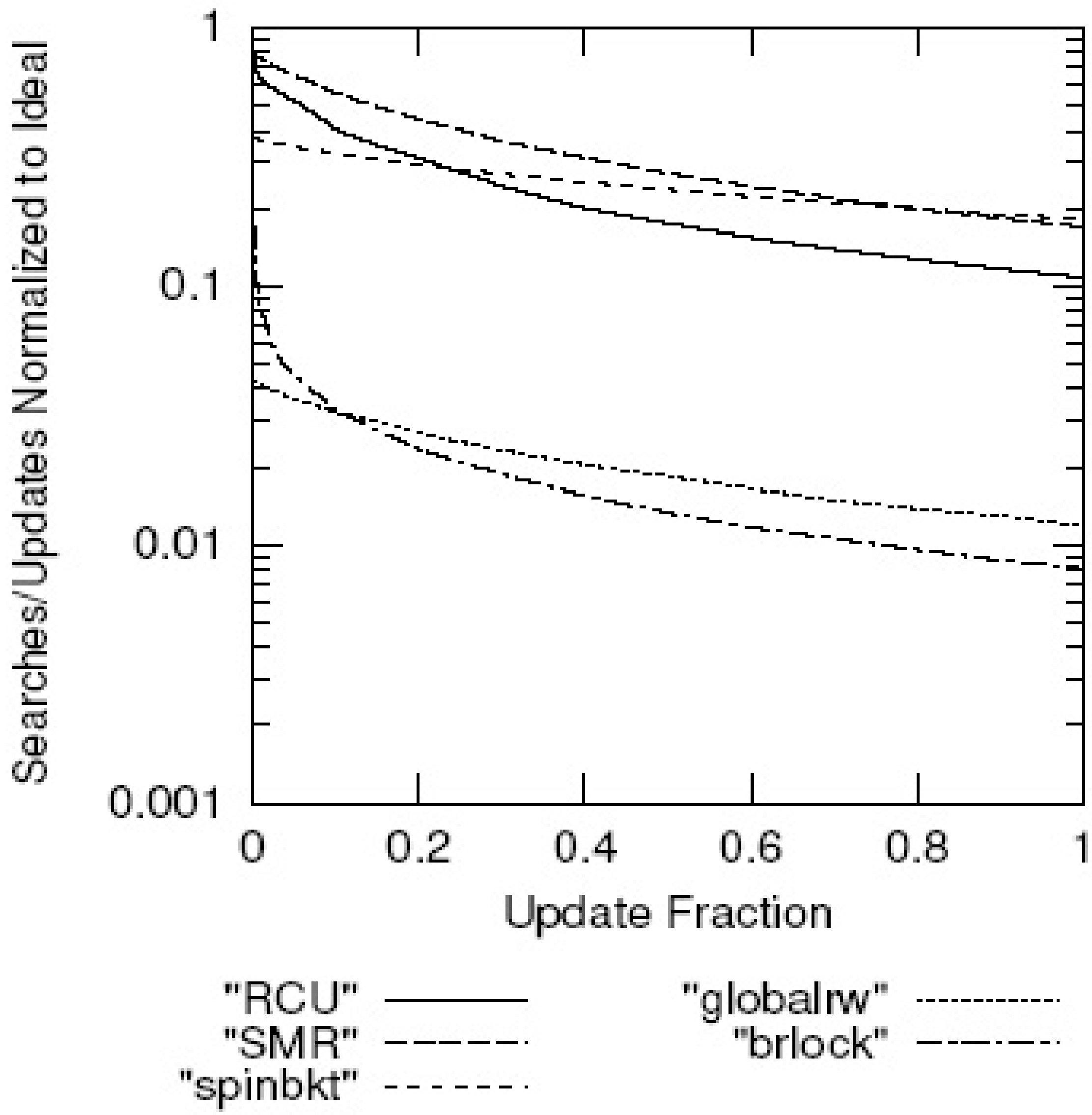
- How does RCU scale?
  - Number of CPUs (n)
  - Number of read-only operations
- How does RCU perform?
  - Fraction of accesses that are updates (f)
  - Number of operations per unit
- What other algorithms to compare to?
  - Global reader-writer lock (*globalrw*)
  - Per-CPU reader-writer lock (*brlock*)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

Scalability

- Hashtable benchmark



Performance



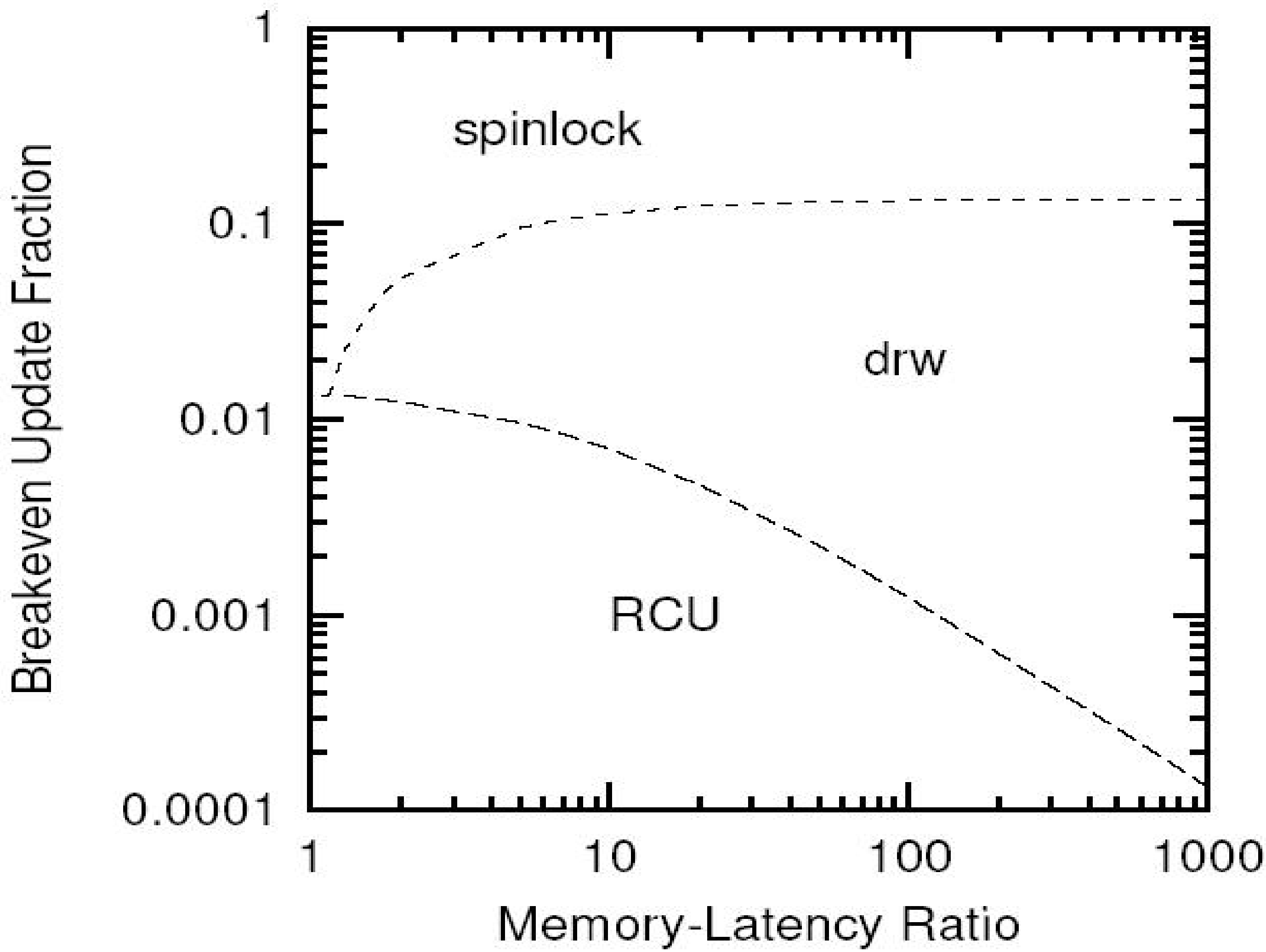
Torsten
Frenzel

TU Dresden
Operating
Systems Group

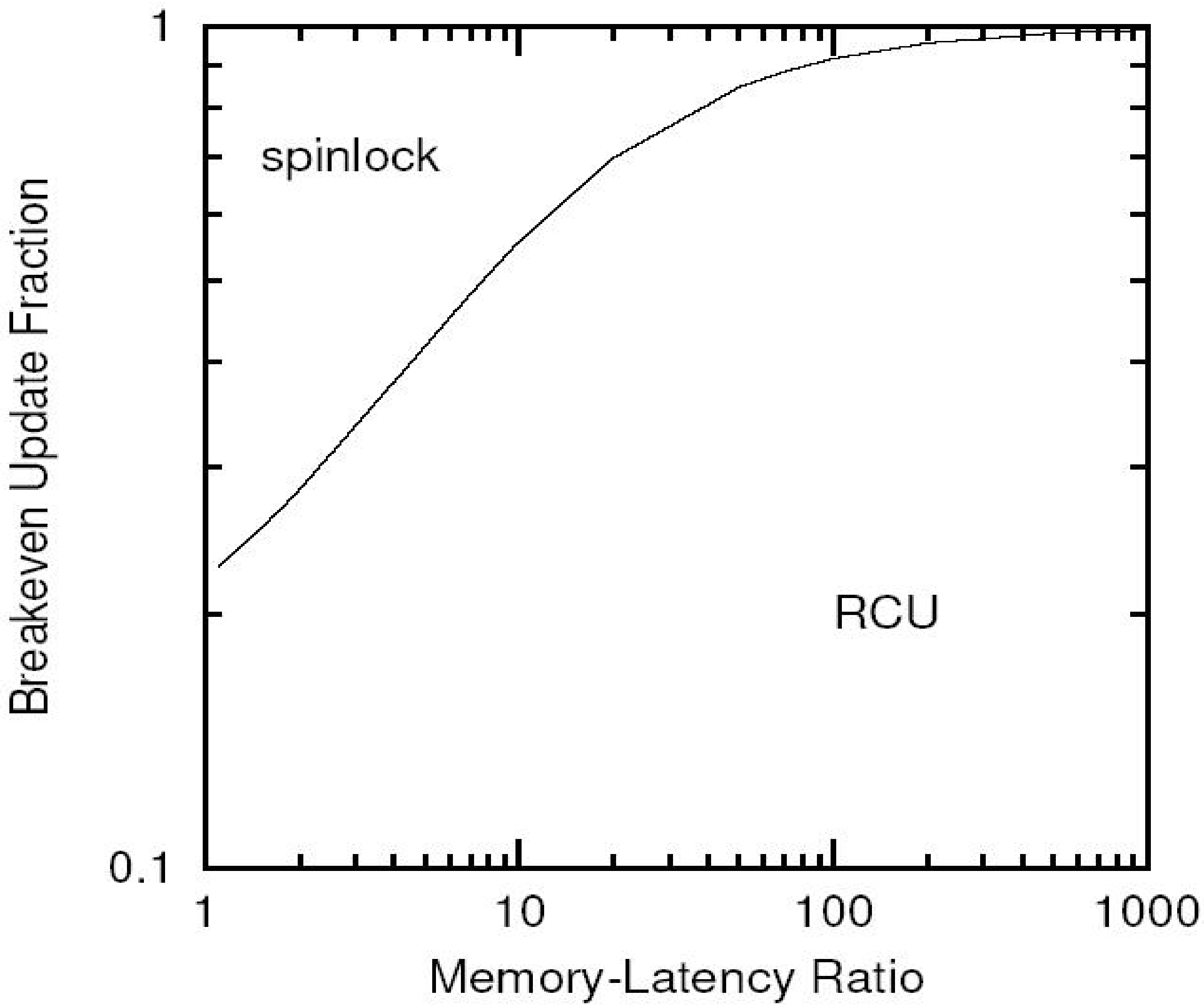
Performance vs. Complexity

- 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 access that are updates (f)
  - Number of updates per grace period (λ = {small, large})

Few Updates per Grace Period



Many Updates per Grace Period



Torsten
Frenzel

TU Dresden
Operating
Systems Group

Concluding Remarks

- RCU performance and scalability
  - Near-optimal scaling with increasing number of CPUs
  - Very good performance under high contention
- RCU modifications
  - Support for weak consistency models
  - Support for NUMA architectures
  - Without stale data tolerance
  - Support for preemptible critical sections
- Other memory reclamation schemes
  - Lock-free reference counting
  - Hazard-pointer-based recalamation
  - Epoch-based reclamation

References

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

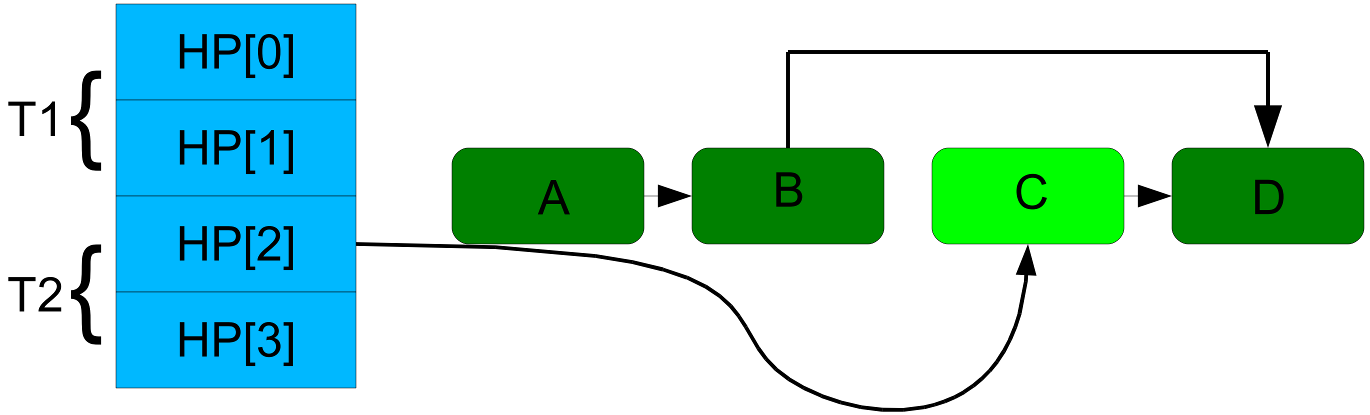
Hazard-Pointer-Based Reclamation

- Introduces H=NK hazard pointers

  N ... number of threads

  K ... data structure dependent (K=2 for queues and lists)

- Memery can only be reclaimed, when no hazard pointer to the location exist



Torsten
Frenzel

TU Dresden
Operating
Systems Group