Read-Copy-Update for HelenOS

http://d3s.mff.cuni.cz

Department of Distributed and Dependable Systems

D3S

Martin Děcký

decky@d3s.mff.cuni.cz

CHARLES UNIVERSITY IN PRAGUE

faculty of mathematics and physics

HelenOS

Introduction

- **HelenOS**
  - Microkernel multiserver operating system
  - Relying on asynchronous IPC mechanism
    - Major motivation for scalable concurrent algorithms and data structures
- **Martin Děcký**
  - Researcher in computer science (operating systems)
  - Not an expert on concurrent algorithms
    - But very lucky to be able to cooperate with hugely talented people in this area

In a Nutshell

Asynchronous IPC

=

Communicating parties may access the communication facilities concurrently

In a Nutshell

Asynchronous IPC

=

Communicating parties may access the communication facilities concurrently

→ **The state of the shared communication facilities needs to be protected by explicit synchronization means**

In a Nutshell

Asynchronous IPC

=

Communicating parties have to access the communication facilities concurrently

In a Nutshell

HelenOS

Asynchronous IPC

=

Communicating parties have to access the communication facilities concurrently

← In order to counterweight the overhead of the communication by doing other useful work while waiting for a reply

Department of Distributed and Dependable Systems

In a Nutshell

HelenOS

Asynchronous IPC

→

Communication facilities have to use concurrency-friendly (non-blocking) synchronization means

Department of Distributed and Dependable Systems

In a Nutshell

Asynchronous IPC

→

Communication facilities have to use concurrency-friendly (non-blocking) synchronization means

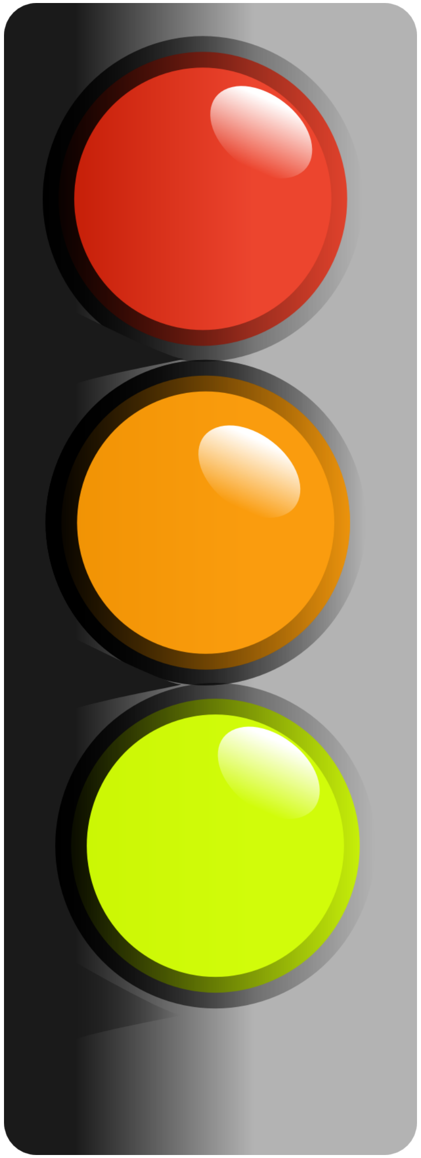
← In order to avoid limiting the achievable degree of concurrency

Department of Distributed and Dependable Systems D3S

Basic Synchronization Taxonomy

- **For accessing shared data structures**
- **Mutual exclusion synchronization**
  - Temporal separation of scheduling entities
  - Typical means
    - *Disabling preemption*, *Dekker's algorithm*, *direct use of atomic test-and-set operations*, etc.
  - Typical mechanisms
    - *Locks*, *semaphores*, *condition variables*, etc.
    - [+] Relatively intuitive semantics, well-known characteristics
    - [-] Overhead, restriction of concurrency, deadlocks

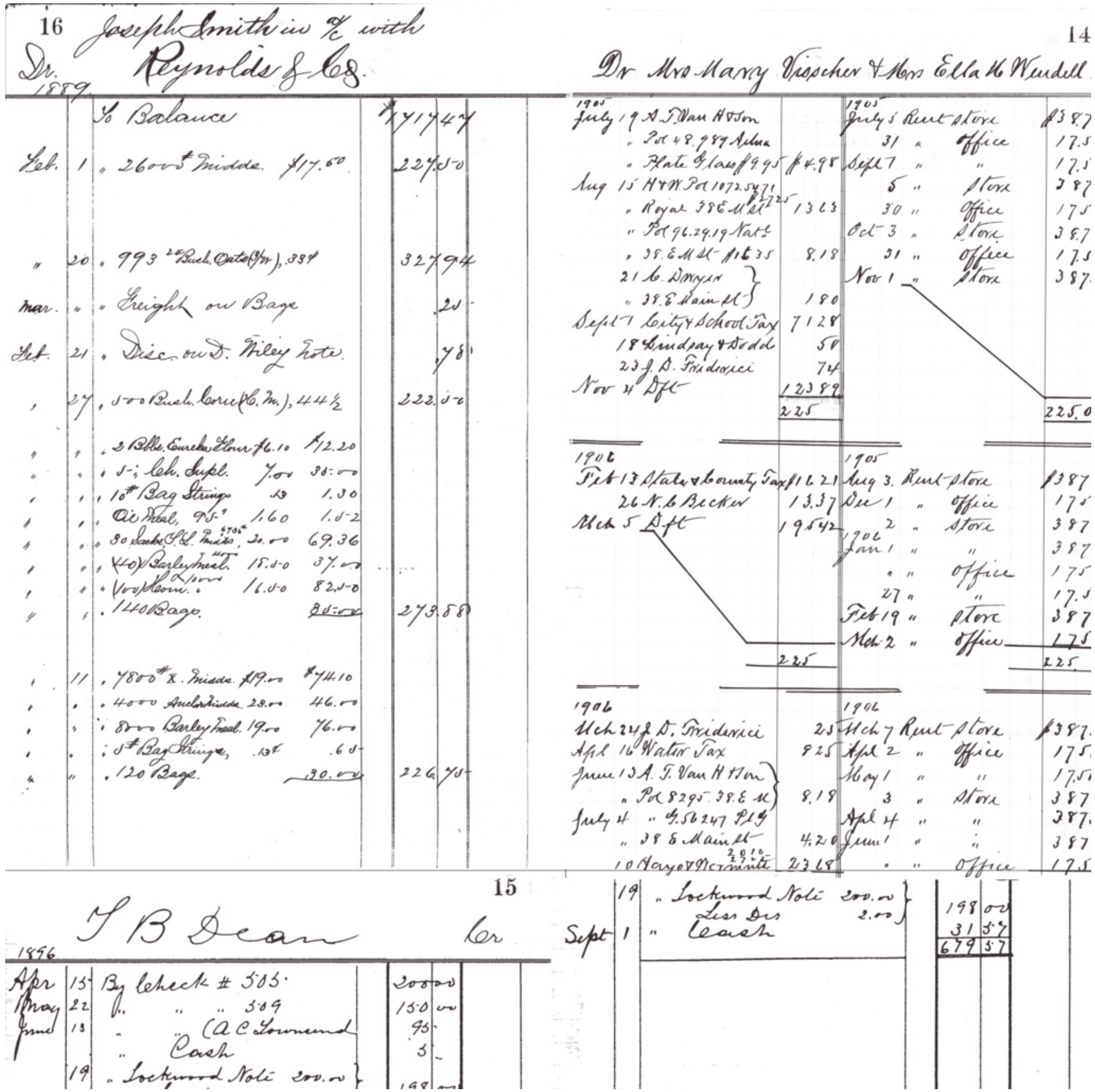
Mutual Exclusion Synchronization



Basic Synchronization Taxonomy

- **Non-blocking synchronization**
  - Replace temporal separation by sophisticated means that guarantee logical consistency
  - Typical means
    - *Atomic writes, direct use of atomic read-modify-write operations*, etc.
  - Typical mechanisms
    - *Transactional memory, hazard pointers*, *Read-Copy-Update*, etc.
  - [+] Reasonable (almost no) overhead and restriction of concurrency in favorable cases, guarantee of progress
  - [-] Less intuitive semantics, sometimes non-trivial characteristics, non-favorable cases, livelocks

Non-blocking Synchronization



Non-blocking Synchronization

- **Wait-freedom**
  - Guaranteed system-wide progress and starvation-freedom (all operations are finitely bounded)
  - Wait-freedom algorithms always exist [1], but the performance of general methods is usually inferior to blocking algorithms
  - Wait-free queue by Kogan & Petrank [2]
- **Lock-freedom**
  - Guaranteed system-wide progress, but individual threads can starve
  - Four phases: Data operation, assisting obstruction, aborting obstruction, waiting
- **Obstruction-freedom**
  - Guaranteed single thread progress if isolated for a bounded time (obstructing threads need to be suspended)

From Means to Mechanism

HelenOS

Synchronization means

Individual instance of usage

→

Synchronization mechanism

Generic reusable pattern

Department of Distributed and Dependable Systems

From Means to Mechanism

HelenOS

Synchronization means

Individual instance of usage

E.g. non-blocking list implementation using atomic pointer writes

Synchronization mechanism

Generic reusable pattern

E.g. non-blocking list implementation using Read-Copy-Update

Department of Distributed and Dependable Systems

What Is Read-Copy-Update

HelenOS

- **Non-blocking synchronization mechanism**
  - Targeting synchronization of read-mostly pointer-based data structures with immutable values
    - Favorable case: R/W ratio of ~ 10:1 (but even 1:1 is achievable)
    - Unlimited number of readers without blocking (not waiting for other readers or writers)
    - Little overhead on the reader side (smaller than taking an uncontended lock)
    - Readers have to tolerate “stale” data and late updates
    - Readers have to observe “safe” access patterns
    - Synchronization among writers out of scope of the mechanism
    - Optional provisions for asynchronous reclamation

Department of Distributed and Dependable Systems

What Is Read-Copy-Update (2)

- **Read-side critical section**
  - Delimited by **read_lock()** and **read_unlock()** operations (non-blocking)
    - Protected data can be referenced only inside the critical section
  - Safe **access()** methods for reading pointers
    - Avoiding unsafe compiler optimizations (reloading the pointer)
    - Not necessary for reading values
  - **Quiescent state** (a thread outside a critical section)
  - **Grace period** (all threads pass through a quiescent state)

What Is Read-Copy-Update (3)

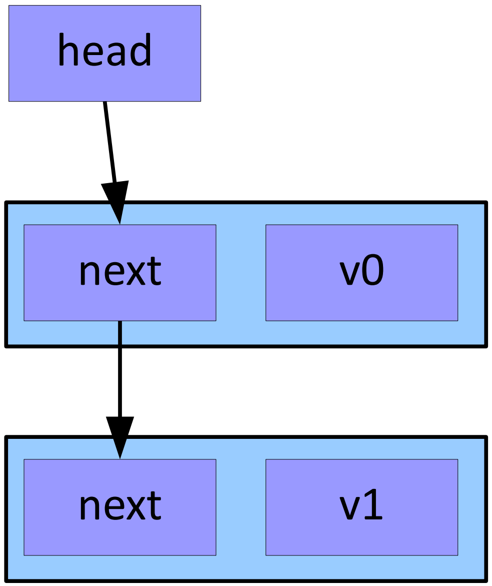
HelenOS

- **Synchronous write-side update**
  - Atomically unlinking an old element
  - Calling a **synchronize()** operation
    - Blocks until a grace period elapses (all readers pass a quiescent state, no longer referencing the unlinked data)
    - Possibility to reclaim or free the unlinked data
  - Inserting a new element using safe **assign()** operation
    - Avoiding unsafe compiler optimizations and store reordering on weakly ordered architectures

Department of Distributed and Dependable Systems

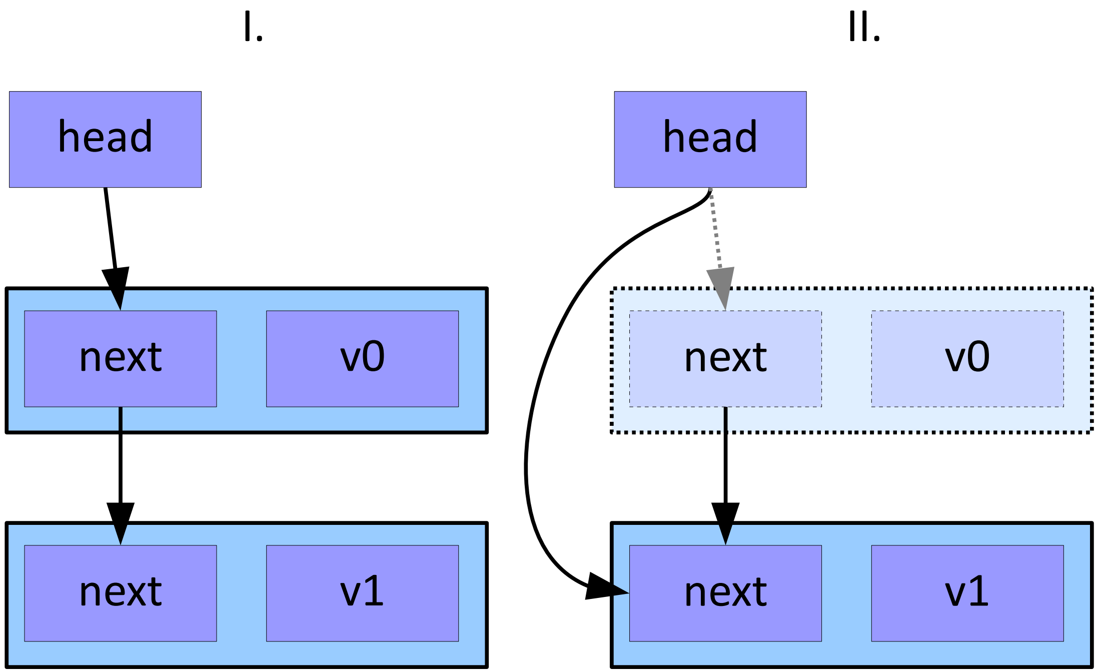
Synchronous Update Example

I.



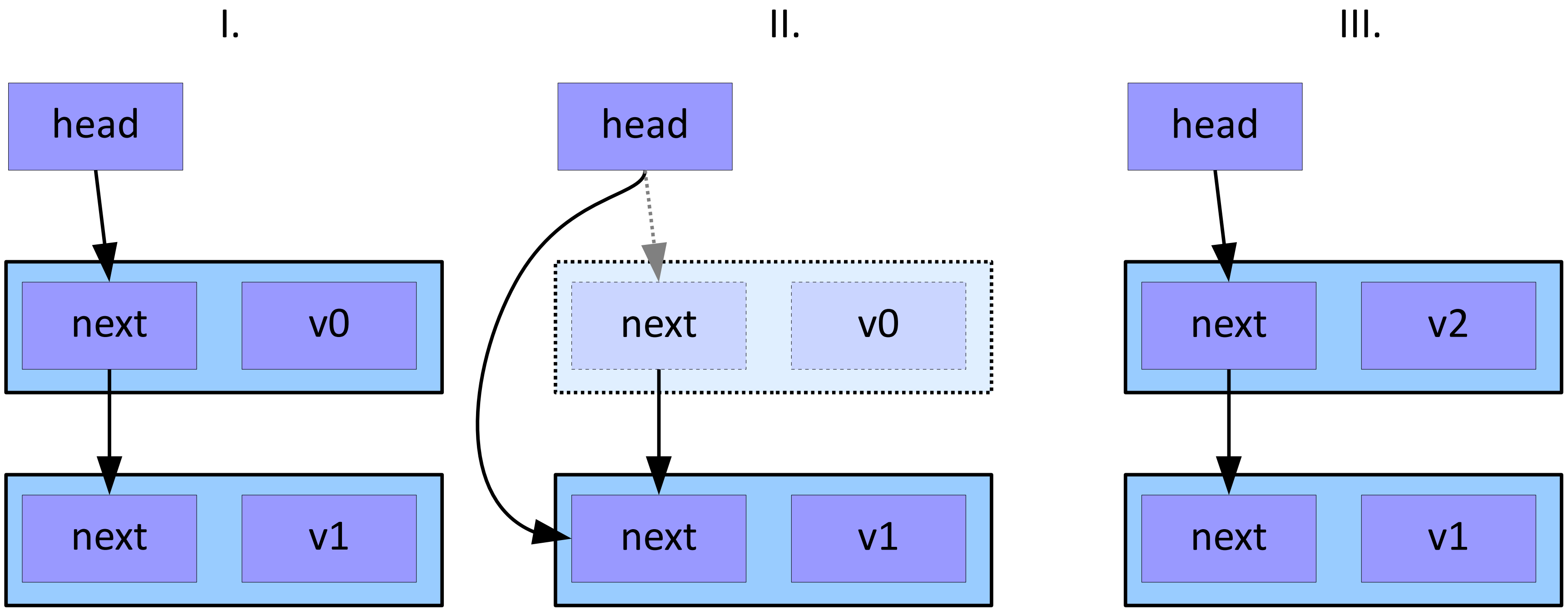
Atomic pointer update to remove the element with v0 from the list

Synchronous Update Example



Blocking on **synchronize()**
During the grace period preexisting readers can still access the “stale” element with v0

Synchronous Update Example



No reader can reference the element with v0 anymore – it can be reclaimed
New element with v2 can be atomically inserted

What Is Read-Copy-Update (4)

- **Asynchronous write-side update**
  - Using a **call()** operation
    - Non-blocking operation registering a callback
    - Callback is executed after a grace period elapses
  - Using a **barrier()** operation
    - Waiting for all queued asynchronous callbacks

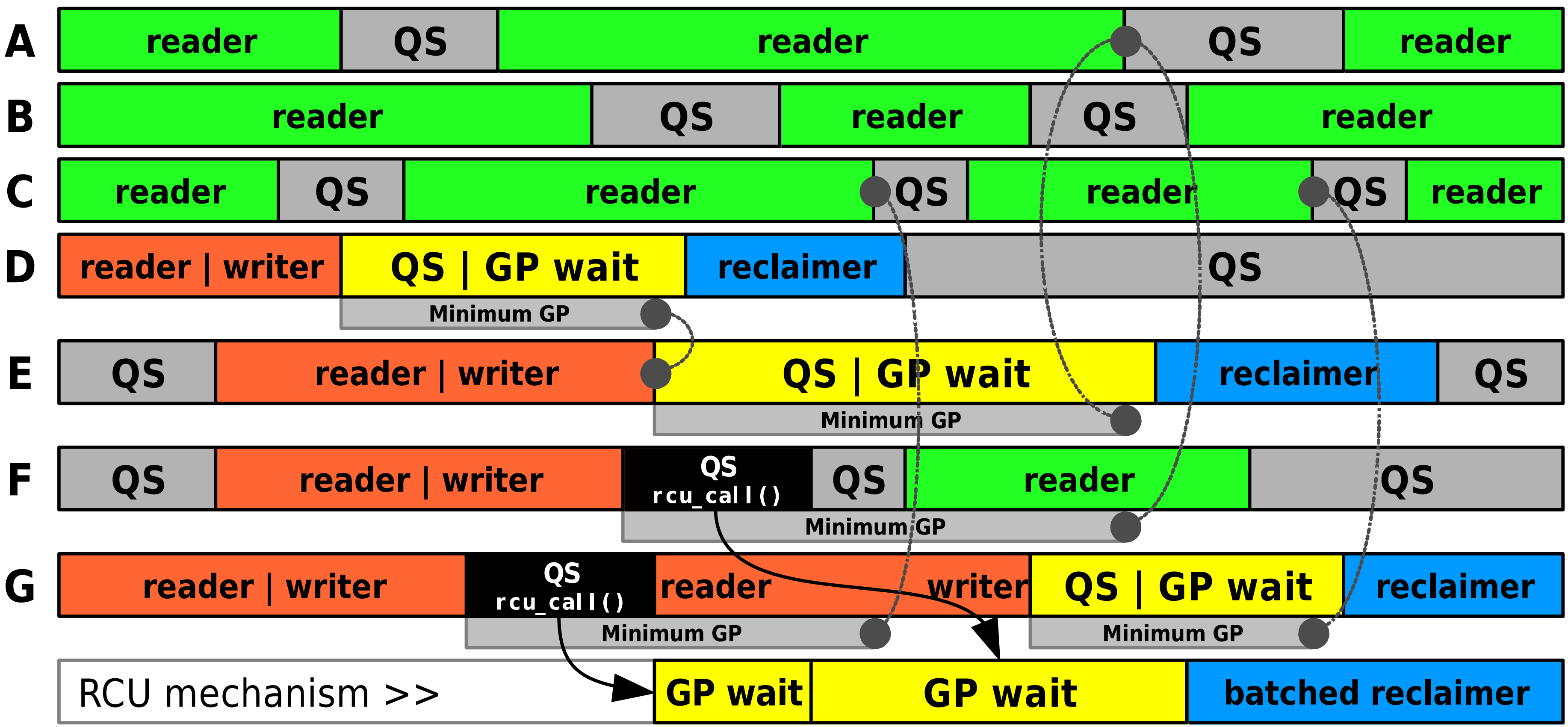
Grace Period Detection

HelenOS

- **Cornerstone of any RCU algorithm**
  - Implicit trade-off between precision and overhead
    - Any extension of a grace period is also a grace period
    - Long (imprecise) grace periods
      - Blocking synchronous writers for a longer time
      - Increasing memory usage due to unreclaimed elements
    - Short (precise) grace periods
      - Increasing overhead on the reader side (need for memory barriers, atomic operations, other heavy-weight operations, etc.)
  - Usual compromise
    - Identifying *naturally occurring quiescent states* for the given RCU algorithm
      - Context switches, exceptions (timer ticks), etc.

Department of Distributed and Dependable Systems

The Big Picture ...



Motivation for RCU in HelenOS

- **Foundation for a scalable concurrent data structure**
- **Developing a microkernel-specific RCU algorithm**
  - Specific requirements, constraints and use cases
  - Last well-known RCU implementation for a microkernel in 2003 (K42)

Credits

- **AP-RCU**
  - Non-intrusive, portable RCU algorithm
  - Developed and implemented by Andrej Podzimek for UTS (OpenSolaris) [3] [4]
- **AH-RCU**
  - Inspired by AP-RCU and several other RCU algorithms
  - Developed and implemented by Adam Hraška for SPARTAN (HelenOS) [7]
  - Foundation for the Concurrent Hash Table in HelenOS [8]
  - Additional variants (preemptible AP-RCU, user space RCU)

HelenOS requirements

- **The RCU algorithm must not impose design concepts of legacy systems on HelenOS**
  - E.g. a specific way how the timer interrupt handler is implemented
- **The kernel space RCU algorithm must support**
  - Read-side critical sections in interrupt and exception handlers
  - Asynchronous reclaimation (**call()**) in interrupt and exception handlers
  - Read-side critical sections with preemption enabled (not affecting scheduling latency)

HelenOS requirements (2)

- **Concurrent Hash Table implementation**
  - Growing and shrinking
  - Interrupt and non-maskable interrupt tolerant
    - Suitable for a global page hash table
  - Concurrent reads with low overhead
  - Concurrent inserts and deletes

AH-RCU

- **Basic characteristics**
  - Kernel space algorithm
  - Read-side critical sections are preemptible (without loss of performance)
    - Multiple read-side critical sections within a time slice
    - Expensive operations when a thread was preempted do not make much harm
  - Support for asynchronous reclamation in interrupt and exception handlers
  - No reliance on periodic timer

AH-RCU (2)

- **Grace period detection**
  - Test if all CPUs passed a quiescent state
    - Sending an interprocessor interrupt (IPI) to each CPU
      - If the interrupt handler detects a nesting count of 0, it issues a memory barrier (representing a natural quiescent state)
    - Avoid sending IPI if context switch is detected
  - Detect any preempted readers holding up the current grace period
    - Sleep and wait for the last preempted reader holding up the grace period to wake the detector thread

AH-RCU (3)

- **Advantages**
  - Low overhead and preemptible read-side critical section, suitable for exception handlers
  - No regular sampling
- **Disadvantages**
  - Polling CPUs using interprocessor interrupts might be disruptive in large systems

HelenOS Concurrent Hash Table

- **Basic characteristics**
  - Inspired by Triplett's relativistic hash table [5] and Michael's lock-free lists [6]
    - Hash collisions resolved using separate RCU-protected bucket lists
    - Buckets organized as lock-free lists without hazard pointers
      - RCU still protects against accessing invalid pointers and the ABA problem
    - Concurrent lookups and concurrent modifications
      - Tolerance for nested concurrent modifications from interrupt and exception handlers
  - Growing and shrinking using background resizing by a factor of 2
    - Concurrent with lookups and updates
    - Requires four grace periods
  - Deferred element freeing using RCU **call()**

Enough talk!

Enough talk!
Show me the (pseudo)code!

AP-RCU Reader Side

read_lock():

```
disable_preemption()
check_qs()
cpu.nesting_cnt++
```

read_unlock():

```
cpu.nesting_cnt--
check_qs()
enable_preemption()
```

check_qs():

```
if (cpu.nesting_cnt == 0) {
    if (cpu.last_seen_gp != cur_gp) {
        gp = cur_gp
        memory_barrier()
        cpu.last_seen_gp = gp
    }
}
```

AP-RCU Reader Side

read_lock():

```
disable_preemption()
check_qs()
cpu.nesting_cnt++
```

read_unlock():

```
cpu.nesting_cnt--
check_qs()
enable_preempti
```

check_qs():

```
if (cpu.nesting_cnt == 0) {
    if (cpu.last_seen_gp != cur_gp) {
        gp = cur_gp
        memory_barrier()
        cpu.last_seen_gp = gp
    }
}
```

The first reader to notice the start of a new grace period on each CPU announces its quiescent state.
Once all CPUs announce a quiescent state or perform a context switch (a naturally occurring quiescent state due to disabled preemption), the grace period ends.

Note: Writer forces a context switch on CPUs where no read-side critical section was not observed for a while.

Note: Except **memory_barrier()** only inexpensive operations.

Department of Distributed and Dependable Systems

Preemptible AP-RCU Reader Side

```
read_lock():
    disable_preemption()
    if (thread.nesting_cnt == 0)
        record_qs()
    thread.nesting_cnt++
    enable_preemption()

read_unlock():
    disable_preemption()
    if (thread.nesting_cnt-- == 0) {
        record_qs()
        if ((thread.was_preempted) ||
           (cpu.is_delaying_gp))
            signal_unlock()
    }
    enable_preemption()
```

```
record_qs():
    if (cpu.last_seen_gp != cur_gp) {
        gp = cur_gp
        memory_barrier()
        cpu_last_seen_gp = gp
    }

signal_unlock():
    if (atomic_exchange(cpu.is_delaying_gp, false)
       == true)
        remaining_readers_semaphore.up()
    if (atomic_exchange(thread.was_preempted, false)
       == true) {
        preempt_mutex.lock()
        preempted_list.remove(thread)
        if ((is_empty(cpu.cur_preempted)) &&
           (preempted_blocking_gp))
            remaining_readers_semaphore.up()
        preempt_mutex.unlock()
    }
```

AH-RCU Reader Side


```
read_lock():
    thread.nesting_cnt++
    compiler_barrier()

read_unlock():
    compiler_barrier()
    thread.nesting_cnt--
    if (thread.nesting_cnt == was_preempted)
        preempted_unlock()

preempted_unlock():
    // avoid race between thread and interrupt handler
    if  (atomic_exchange(thread.nesting_cnt, 0) == was_preempted) {
        preempt_lock.lock()
        preempted_list.remove(thread)
        if ((is_empty(cpu.cur_preempted)) && (detection_waiting))
            detection_semaphore.up()        // notify the detector thread about the grace period
        preempt_lock.unlock()
    }
```

Note: Except **preempted_unlock()** only inexpensive operations.

AP-RCU Writer Side

synchronize():

```
	memory_barrier()
	mutex.lock()

	cur_gp++                                        // start a new grace period
	reader_cpus = []                                // gather CPUs potentially in read-side CS
	foreach cpu in cpus {
		if ((!cpu.idle) && (cpu.last_seen_gp != cur_gp)) {
			cpu.last_ctx_switch_cnt = cpu.ctx_switch_cnt
			reader_cpus += cpu
		}
	}

	wait(10ms)                                      // longest acceptable grace period duration (tunable)

	foreach cpu in reader_cpus {                    // enforce a quiescent state
		if ((!cpu.idle) && (cpu.last_seen_gp != cur_gp) &&
		   (cpu.last_ctx_switch_cnt == cpu.ctx_switch_cnt))
			cpu.ctx_switch_force_wait()
	}

	mutex.unlock()
```

AH-RCU Writer Side

HelenOS

```
detector_thread:
    forever {
        wait_for_callbacks()

        // run callbacks added before the current grace period
        execute_callbacks()

        // push callbacks registered since last processing to the queue
        advance_callbacks()

        wait_for_gp_end()
    }
```

Department of Distributed and Dependable Systems

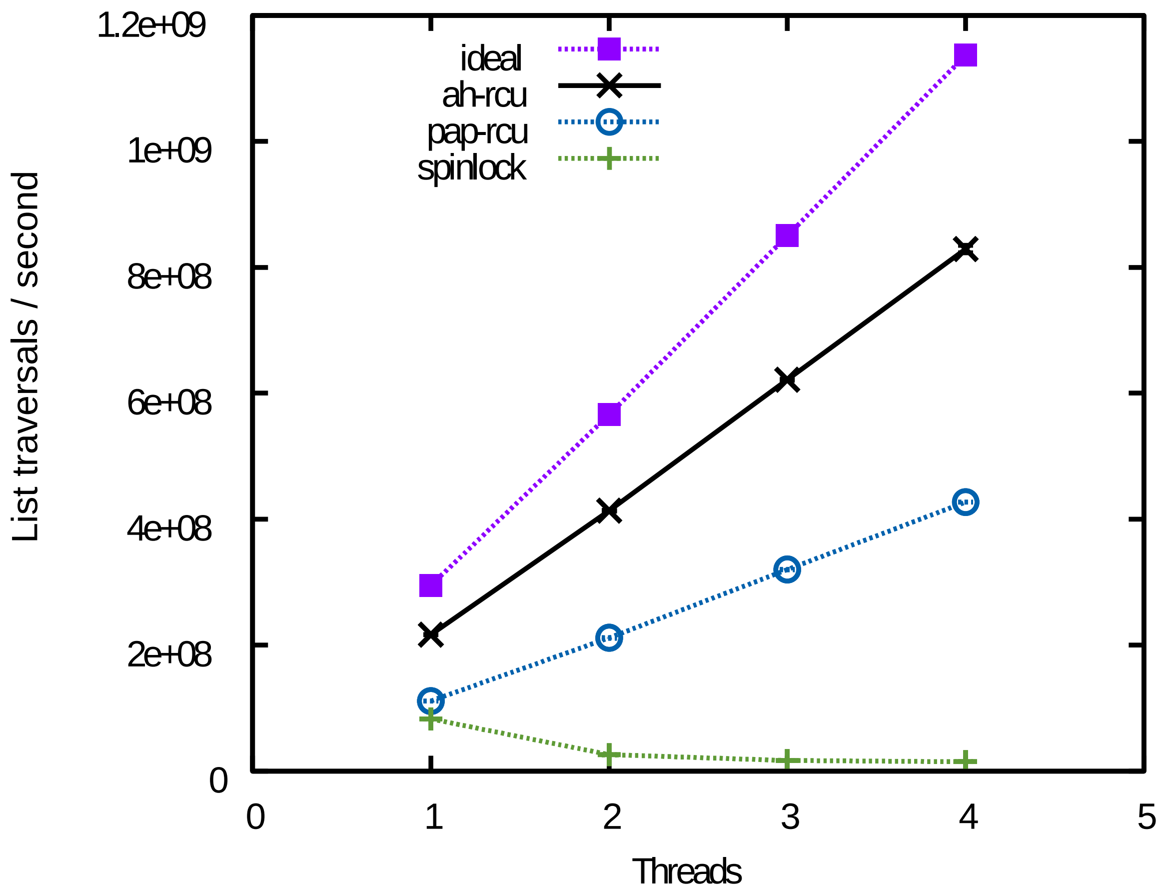
AH-RCU Writer Side (2)

```
wait_for_gp_end():
    gp_mutex.lock()
    if (completed_gp != cur_gp) {      // a grace period is already in progress
        wait_for_gp_end_signal()
        goto out
    } else {                           // start a new grace period
        preempt_lock.lock()
        cur_gp++
        preempt_lock.unlock()
    }
    gp_mutex.unlock()

    wait_for_readers()

    gp_mutex.lock()
    completed_gp = cur_gp

    out:
        gp_mutex.unlock()
```

Department of Distributed and Dependable Systems

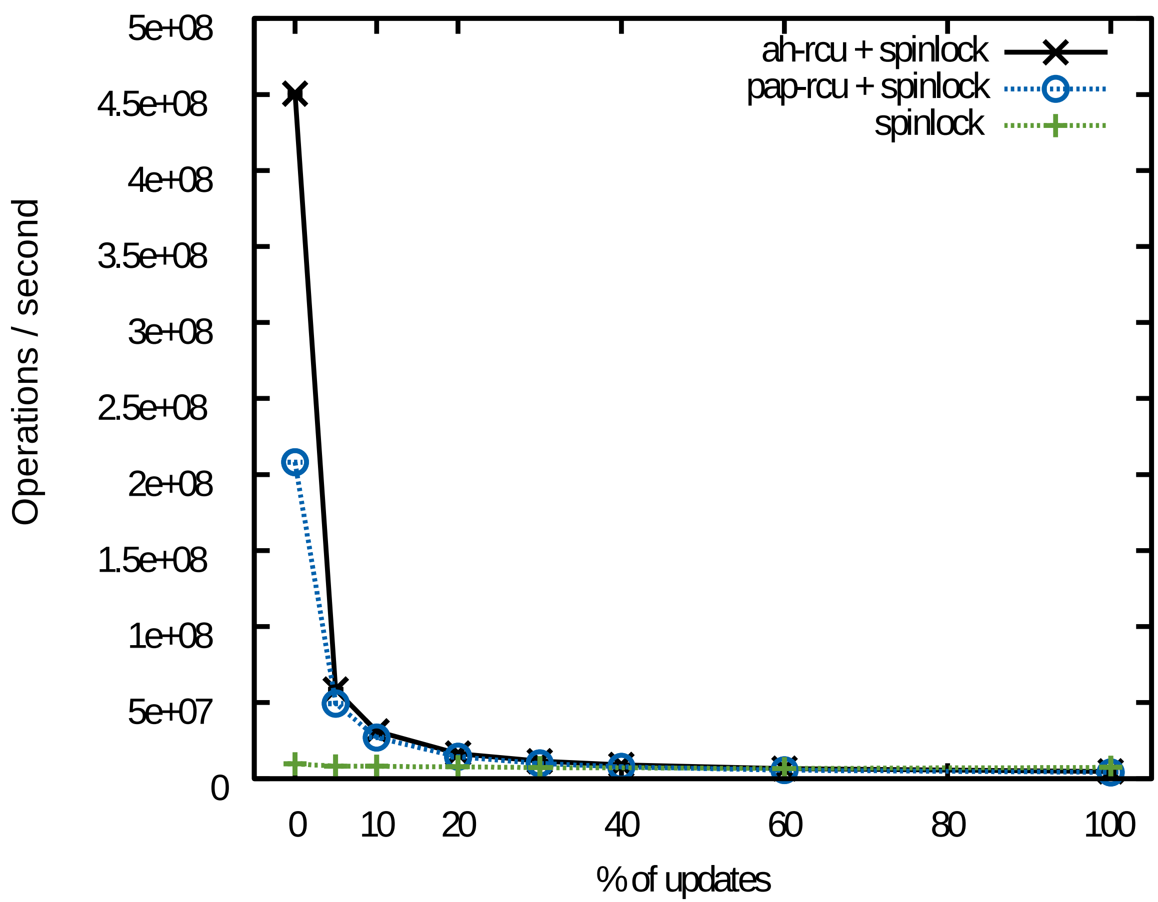
Evaluation

Read-side critical section scalability: Traversal of a five-element list
The list is protected as a whole, it is only read, never modified.



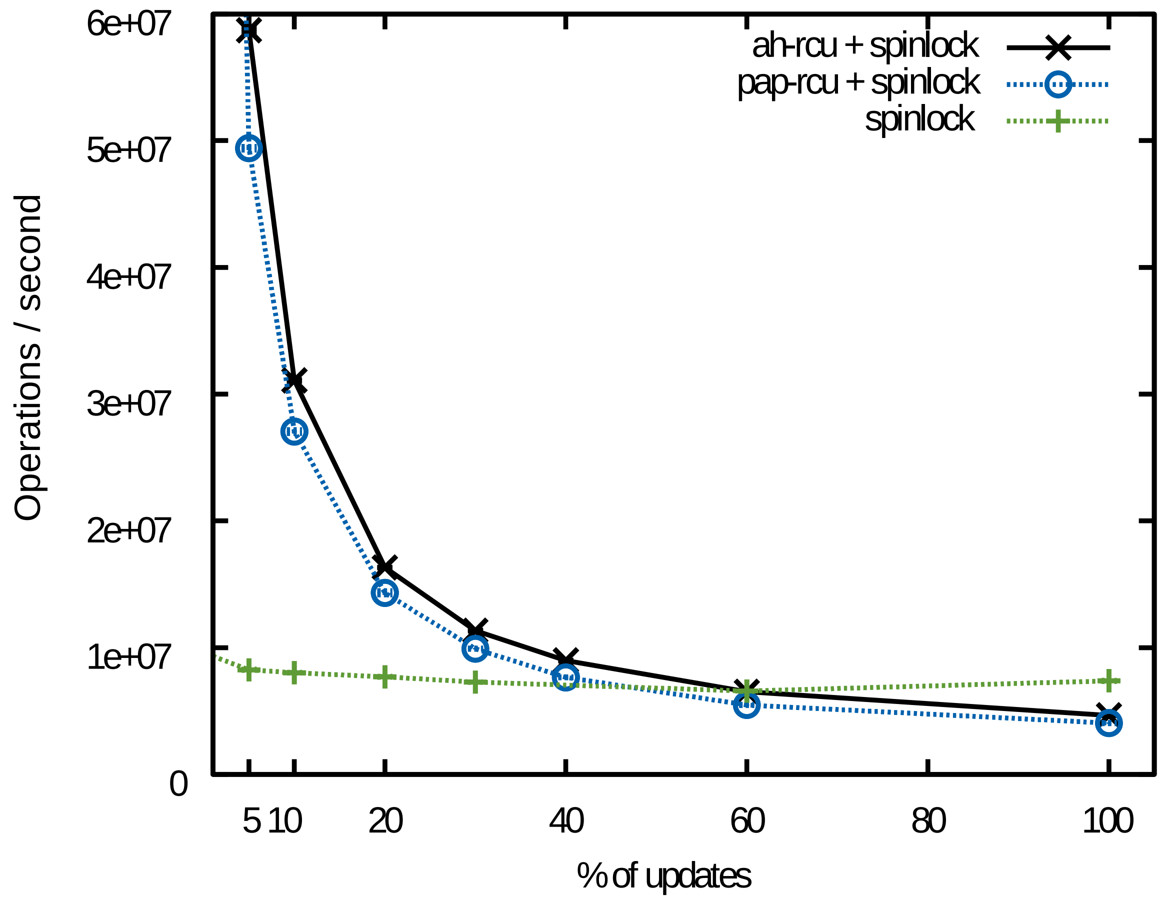
Evaluation (2)

Write-side overhead: Different ratios of updates vs. lookups
Five-element list, four threads running in parallel. Updates are always synchronized by a spinlock.



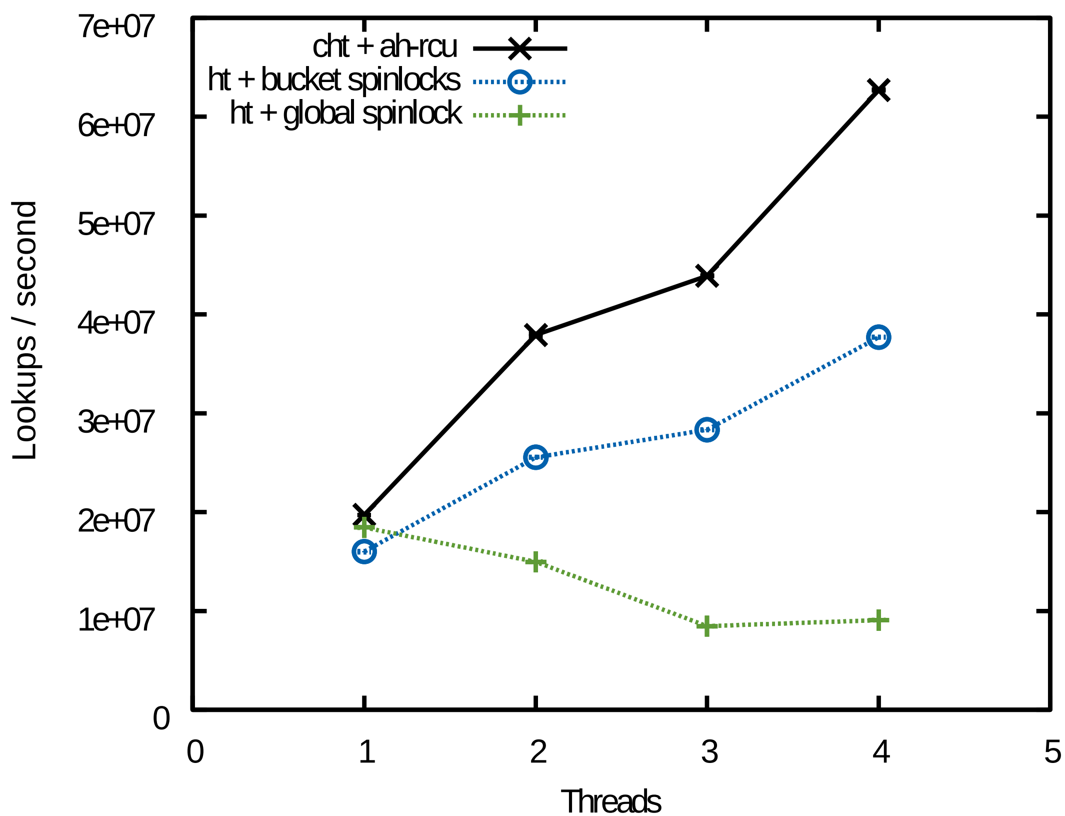
Evaluation (3)

Read-side scalability vs. write-side overhead: Crossover point
Data points from previous figure with low fraction of updates are discarded.



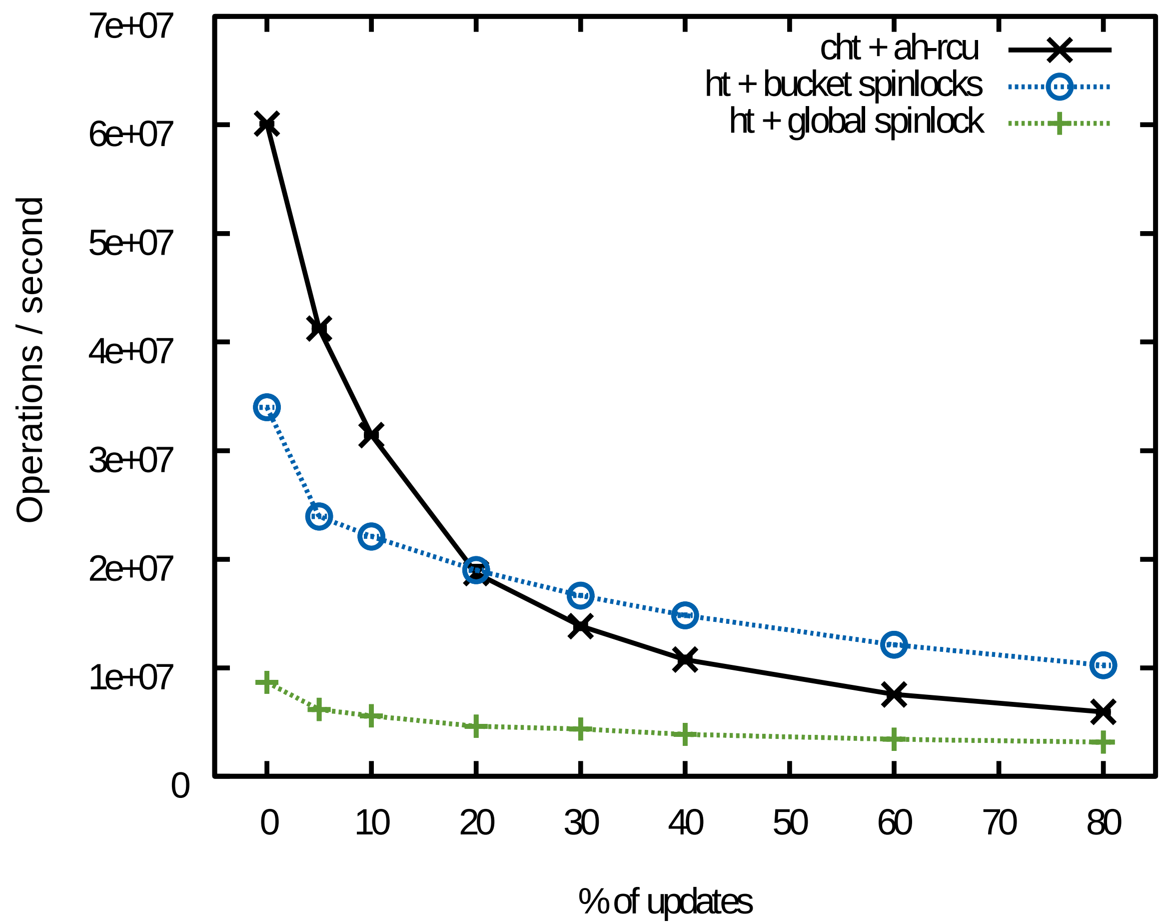
Evaluation (4)

Concurrent hash table lookup scalability
128 buckets, average load factor of 4 elements per bucket, 50 % of lookups for hitting keys, 50 % of lookups for missing keys (each thread used a separate list). The resize condition was checked, but never executed.



Evaluation (5)

Concurrent hash table update overhead: Different ratios of concurrent updates vs. lookups
Four threads running in parallel.



Conclusion

HelenOS

- **Novel scalable algorithms**
  - Preemptible AP-RCU for HelenOS
  - Preemptible AH-RCU for HelenOS
  - Resizeable Concurrent Hash Table for HelenOS
    - Suitable as a basic data structure for asynchronous HelenOS IPC
    - Suitable for other kernel uses (e.g. global page table)
- **Thorough evaluation**
  - Promising behavior

Department of Distributed and Dependable Systems

Q&A

HelenOS

www.helenos.org

Department of Distributed and Dependable Systems

References

[1] Herlihy M. P.: *Impossibility and universality results for wait-free synchronization*, in Proceedings of 7th Annual ACM Symposium on Principles of Distributed Computing, ACM, 1988

[2] Kogan A., Petrank E.: *Wait-free queues with multiple enqueuers and dequeuers*, in Proceedings of 16th ACM Symposium on Principles and Practice of Parallel Programming, ACM, 2011

[3] Podzimek A., Děcký M., Bulej L., Tůma P.: *A Non-Intrusive Read-Copy-Update for UTS*, in Proceedings of 18th IEEE International Conference on Parallel and Distributed Systems, IEEE, 2012, http://d3s.mff.cuni.cz/publications/download/PodzimekDeckyBulejTuma-ICPADS-2012.pdf

[4] http://d3s.mff.cuni.cz/software/rcu/rcu.patch

[5] Triplett J., McKenney P. E., Walpole J.: *Resizable, scalable, concurrent hash tables via relativistic programming*, in Proceedings of the 2011 USENIX Annual Technical Conference, ACM, 2011

[6] Michael M. M.: High performance dynamic lock-free hash tables and list-based sets, in Proceedings of 14th Annual ACM Symposium on Parallel Algorithms and Architectures, ACM, 2002

[7] Hraška A.: *Read-Copy-Update for HelenOS*, master thesis, Charles University in Prague, 2013, http://www.helenos.org/doc/theses/ah-thesis.pdf

[8] https://code.launchpad.net/~adam-hraska+lp/helenos/cht-bench