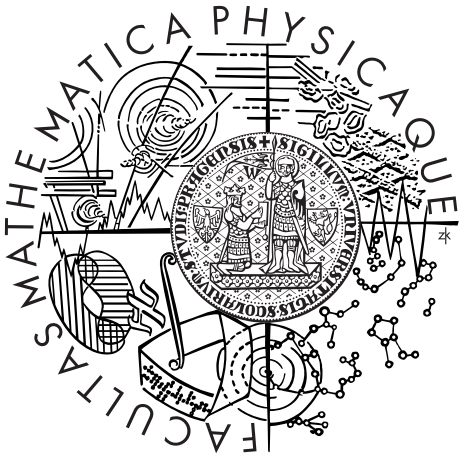
Charles University in Prague
Faculty of Mathematics and Physics

MASTER THESIS



Adam Hraška

Read-Copy-Update for HelenOS

Department of Distributed and Dependable Systems

Supervisor of the master thesis: Mgr. Martin Děcký
Study programme: Computer Science
Specialization: Software Systems

Prague 2013

First and foremost, I would like to thank my supervisor, Mgr. Martin Děcký, for giving me the opportunity to pursue a topic I was genuinely interrested in and the freedom to approach the research problems at hand in my own way.

Secondly, I would like to express sincere gratitude to Paul McKenney whose wonderful papers on the topics of synchronization and RCU made working on this thesis an enlightening experience.

Next, I have to give my deepest thanks to my closest family for their unmatched support and heroic patience.

Last but not least, I would like to thank Růženka for reminding me what is important in life and for her everlasting encouragements; Jan, for his swift help with the graphs; and Marcel, who spent a good part of his favorite day of the week travelling in the public transport of Prague just to make this thesis a reality.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Amsterdam on 29.07.2013 signature of the author

Název práce: Read-Copy-Update for HelenOS

Autor: Adam Hraška

Katedra: Katedra distribuovaných a spolehlivých systémů

Vedoucí diplomové práce: Mgr. Martin Děcký, Katedra distribuovaných a spolehlivých systémů

Abstrakt: Rozšírenie viacjadrových procesorov zvýšilo záujem o škálovateľné synchronizačné primitíva ako je Read-Copy Update. Zatiaľčo RCU je populárne v monolitických kerneloch operačných systémov, doposiaľ nebolo nasadené v prostredí mikrokernelov. V práci navrhneme a implementujeme RCU pre mikrokernelový operačný systém HelenOS. Navyše preskúmame možnosti použitia RCU v HelenOSe and demonštrujeme užitočnosť RCU tak v kerneli ako aj v user space. Merania ukazujú, že implementované RCU poskytuje lineárnu škalovateľnosť RCU čitateľov a RCU vyžaduje omnoho nižšiu réžiu pri vstupe do chránenej sekcie ako bežné zámky a to i v ideálnom prípade pre zámky. RCU sme využili v user space na 2.6 násobné zrýchlenie tradičných zámkov. V kerneli RCU zabezpečilo lineárnu škálovateľnosť futexového podsystému.

Klíčová slova: RCU, Read-Copy Update, HelenOS, concurrency

Title: Read-Copy-Update for HelenOS

Author: Adam Hraška

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Martin Děcký, Department of Distributed and Dependable Systems

Abstract: Multicore processors have become prevalent and spurred interest in scalable synchronization mechanisms, such as Read-Copy Update. While RCU is popular in monolithic operating system kernels it has yet to see an implementation in a microkernel environment. In this thesis we design and implement RCU for the microkernel operating system HelenOS. Moreover, we explore potential uses of RCU in HelenOS and illustrate its utility in both the kernel and user space. Benchmarks demonstrate that the RCU implementation provides linearly scalable read-sides and incurs significantly less overhead than traditional locking even if uncontended. Furthermore, RCU was used in user space to speed up traditional locking 2.6 times in the common case. In the kernel, RCU ensured linear scalability of a previously non-scalable futex subsystem.

Keywords: RCU, Read-Copy Update, HelenOS, concurrency

Contents

Chapter 1

Introduction

In recent years multicore processors have become entirely ubiquitous. They are no longer exclusively in the domain of high-end computers but have already overtaken the low-end computer segment and are now entering the smartphone market.

In order to harness the processing power of multicore machines, software has to run in physically parallel threads. These threads typically synchronize regularly, e.g. when accessing shared data, by means of standard synchronization primitives such as mutexes or reader-writer locks. Unfortunately, as the frequency of synchronization or the number of processing cores increases, standard synchronization primitives may introduce considerable overhead. What is more, without proper care they start limiting scalability[1] of the computation. For example, the naive approach of protecting a concurrently accessed data structure with a single mutex may easily lead to negative scaling (section 6.2). As more cores are added to the computation, the frequency of accesses of the shared data increases and the repeated locking and unlocking of the mutex from different cores results in an overall slower computation.

Read-Copy Update, or simply RCU, is a synchronization primitive targeted at concurrent read-mostly data structures that is scalable by design. It achieves its scalability by keeping multiple versions of data available – both new and old. As a result, it may propagate changes to other cpus only gradually and thereby avoid the expensive operations used in standard synchronization primitives to immediately make the changes visible on other cpus.

RCU already proved successful in the environment of the monolithic Linux kernel but has yet to see an implementation in a microkernel operating system. This thesis aims to bring RCU into the microkernel setting of the research oper-

[1] Scalability represents the increase in performance as the computation is extended to other processing cores of the computer.

ating system HelenOS. In particular, the following are the goals of the thesis:

- Give a thorough overview of existing RCU algorithms along with their strengths and weaknesses.
- Provide a prototype implementation of RCU that is suitable for HelenOS. Due to its microkernel design RCU must be available not only to kernel components but to user space servers and applications as well.
- Explore potential use of RCU in HelenOS and demonstrate its utility in a concrete subsystem.
- Evaluate the performance of the prototype.

1.1 Thesis overview

This thesis first describes the semantics of RCU in chapter 2. Next, chapter 3 reviews existing RCU algorithms that are of interest along with their advantages and disadvantages. Chapter 4 introduces the RCU algorithms that were implemented for HelenOS. Furthermore, chapter 5 explores how RCU was used in HelenOS. The focus of chapter 6 is experimental evaluation of both the implemented RCU algorithms as well as the use of RCU in HelenOS. Finally, chapter 7 concludes this thesis with a summary and suggests possible future work.

Chapter 2

What is Read-Copy Update?

Read-Copy Update is designed to enable scalable concurrent read access to data that is mostly read, i.e. where the fraction of updates does not exceed 10-40% of all operations ([13] and section 8.2 of [12]). Quite unlike traditional locking schemes, such as reader-writer locks, RCU allows readers to execute concurrently with not only other readers but with updaters as well. While an update is in progress RCU maintains older versions of the data and ensures readers always access coherent albeit possibly outdated data. Furthermore, RCU provides facilities to efficiently propagate new versions of objects and to easily defer reclamation of old objects until they are no longer referenced. As a result, RCU read-side critical sections incur little overhead and entering them is significantly faster than locking an uncontended lock.

Examples of data that changes infrequently but may benefit from fast concurrent reads include routing tables, security policies or hardware configuration. In addition, an access pattern suitable for RCU may be observed even in ordinary data strucutes; e.g. the authors of [31] report that the typical hash table usage pattern consists of 88% lookup, 10% insert and 2% delete operations.

2.1 Semantics

RCU readers may only reference RCU protected data within a *read-side critical section*, or simply a *reader section*. Read-side critical sections are delimited by Rcu-Read-Lock and Rcu-Read-Unlock, which are functions that never block. If a thread is not executing a read-side critical section it is said to be in a *quiescent state*, i.e. a state when the thread is not referencing any RCU protected objects. Any time period such that each thread passes through at least one quiescent state is called a *grace period*. By definition, any critical sections existing at the beginning of a grace period will have completed before the grace period ends. Moreover, threads continuously entering RCU critical sections do

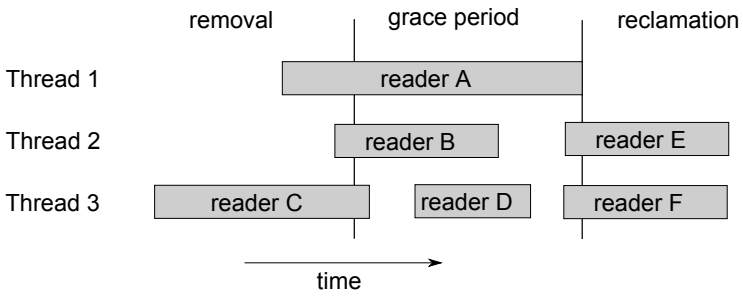


Figure 2.1: This figure portrays how grace periods extend at least until the last thread that is in a read-side critical section at the start of the grace period exits its critical section, in this case reader A. Notice that overlapping reader sections do not prolong the grace period.

not prolong a grace period even if the critical sections overlap with sections in other threads. Provided that critical sections have finite duration, each thread will eventually pass a quiescent state and a grace period will end.

Figure 2.1 illustrates how a grace period extends at least until all threads exit their preexisting critical sections, namely A, B and C. Thread 3 is the first to pass a quiescent state after leaving critical section C, so any further critical sections D and F do not affect the length of the grace period. Next, thread 2 enters a quiescent state after exiting B and stays in that state until it reaches E. Finally, the grace period may complete when thread 1, the last thread to pass a quiescent state since the start of the grace period, exits its critical section A.

Updaters use grace periods to effect deferred destruction. In particular, to remove an element from a data structure, an updater first unlinks the element from the data structure. Next, it invokes RCU-SYNCHRONIZE, which waits for a grace period to elapse. Once RCU-SYNCHRONIZE returns all readers using the element in a critical section at the start of the grace period must have completed. Because following readers find the element unlinked all readers that may have been using the element must have completed by the time RCU-SYNCHRONIZE returns. Therefore, it is safe to reclaim or free the unlinked element.

It is important to note that RCU only coordinates the concurrent execution of a reader with other readers or updaters. However, it in no way synchronizes updaters with other updaters. Therefore, updaters usually synchronize by other means, e.g. with locks.

2.2 Example usage

This section demonstrates how to use RCU with an example of working with a single linked null terminated list.

```
1 typedef struct item {
2     struct item *next;
3     int value;
4 } item_t;

6 typedef struct list {
7     item_t *first;
8     mutex_t update_mtx;
9 } list_t;
```

Line 6 defines a list whose chain of elements starting with `first` can be read in parallel as it is protected by RCU. In this example, RCU protects only the list's links `next` and assumes elements' contents, `value`, does not change after an element is inserted into the list. The list is protected from concurrent modifications with a single mutex `update_mtx`, defined on line 8.

```
1  int get_last_value(list_t *list)
2  {
3      rcu_read_lock();
4      item_t *cur = NULL;
5      item_t *next = rcu_access(list->first);
6      while (next) {
7          cur = next;
8          next = rcu_access(cur->next);
9      }
10     int val = (cur != NULL) ? cur->value : 0;
11     rcu_read_unlock();
12     return val;
13 }
```

Function `get_last_value()` traverses the list and returns the `value` stored in the last element or 0 if the list is empty. Before accessing any RCU protected fields, in this case `list->first`, the function enters a reader section on line 3. Next, it reads the protected field `list->first` via `rcu_access()`. Line 5 ensures that the compiler does not accidentally change `next`'s value in background in the middle of the reader section with compiler optimizations even in the face of concurrent updates of `list->first`. In other words, it guarantees `next` will point to the same element, i.e. the same version of the data, for the duration of the reader section as one would expect. Lines 6-9 search for the last element of the list and again load any RCU protected variables, `cur->next`, with `rcu_access()` as RCU requires. Next, last element's value is copied to `val` on line 10. However, reading the element's `cur->value` does not require `rcu_access()`. Firstly, in this example RCU protects only the list's structure and not the content of individual

elements. Secondly, after inserting elements into the list they remain immutable by convention. Therefore, if an element is accessible by the list's next pointers its `value` is certain to remain the same until it is freed.

Last, the function exits its reader section on line 11. After this point RCU may notify updaters that it is safe to reclaim elements used in the reader section. Consequently, the function must not access any elements of the list and it returns a copy of the last element's value `val` instead.

```
1  void insert_first(list_t *list, int val)
2  {
3      item_t *item = (item_t*) malloc(sizeof(item_t));
4      mutex_lock(&list->update_mtx);
5      item->value = val;
6      item->next = list->first;
7      rcu_assign(list->first, item);
8      mutex_unlock(&list->update_mtx);
9  }
```

Function `insert_first()` initializes a new element with value `val` and inserts it at the first position in the list. The list is changed with the mutex `list->update_mtx` locked only; therefore, updaters always have a consistent view of the list that cannot change during an update. As a result, line 6 retrieves the current head of the list without `rcu_access()`.

The new element is published for readers to see by means of `rcu_assign()` on line 7. `rcu_assign()` ensures that the contents of the element, i.e. `item->value`, have been completely initialized on weakly ordered architectures before it assigns `item` to `list->first`. Nevertheless, publishing a new element does not immediately make it accessible from the list's link `list->first`. It may take on the order of milliseconds before all new readers learn of the update to the list's link and start using the newly inserted element.

```
1  void delete_first(list_t *list)
2  {
3      mutex_lock(&list->update_mtx);
4      item_t *to_del = list->first;
5      if (to_del) {
6          list->first = to_del->next;
7          mutex_unlock(&list->update_mtx);
8          rcu_synchronize();
9          free(to_del);
10     } else
11         mutex_unlock(&list->update_mtx);
12 }
```

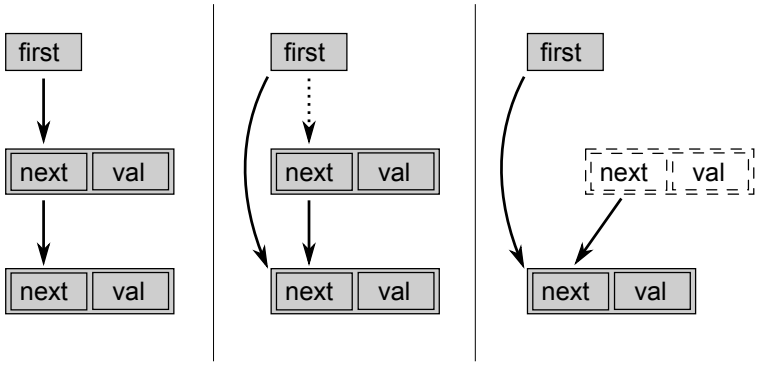


Figure 2.2: This figure shows how the first item of the example single linked list is removed and reclaimed with RCU. The left figure is the initial state. First, an updater removes the first element from the list, i.e. it changes the link from the list head `first` to the next element in the list. Next, it initiates a grace period via RCU-SYNCHRONIZE. The middle figure shows that during the grace period preexisting readers may still see `first` pointing to the removed element. Once the grace period ends, RCU-SYNCHRONIZE returns and the updater finds the list in the state depicted in the right most figure. Since no readers reference the removed element any more, the updater is free to reclaim or free the element.

Function `delete_first()` concludes this example and illustrates deferred destruction of the first element in the list. First, it removes the element from the list on line 6 while holding a mutex. The function only changes the list's structure but it does not publish new elements. Therefore, it is not necessary to `rcu_assign()` the address of the second element to `list->first` on line 6.

Second, it unlocks the mutex serializing updates of the list on line 7. Again, the mutex makes changes of the list's structure immediately visible to other updaters so no updaters will access the removed element after line 7. However, it is important not to modify the element's contents even after it had been removed from the list since readers running concurrently with `delete_first()` may still be reading the element.

Next, `rcu_synchronize()` waits for any readers that may have been accessing the removed element to complete and ensures that the removal of the first element is visible to any new readers. Finally, the element may be reclaimed on line 9 because neither other updaters nor any readers are referencing it any longer.

Figure 2.2 depicts the process in further detail.

2.3 Complete interface

The complete RCU interface features the listed functions:

RCU-READ-LOCK delimits the start of a read-side critical section, which may be nested.

Rcu-Read-Unlock marks the end of a read-side critical section.

Rcu-Synchronize blocks for an entire grace period. It waits for any preexisting readers, i.e. readers that may be accessing old versions of data, to complete. Moreover it guarantees any new readers, i.e. readers Rcu-Synchronize does not wait for, see changes introduced prior to calling Rcu-Synchronize in the same thread.

Rcu-Call is similar to Rcu-Synchronize. However, instead of blocking Rcu-Call returns immediately and asynchronously invokes a callback function when a grace period elapses.

Rcu-Barrier waits for all callbacks queued via Rcu-Call at the time of the call to Rcu-Barrier to complete.

Rcu-Dereference, Rcu-Access is used to load a pointer to an RCU protected object within a reader section. It prohibits the compiler to reload the pointer as part of optimizations. Otherwise the compiler could reload the pointer with an address to a different object within the same reader section. In other words, the compiler would unknowingly start referencing a different object in the middle of the reader section.

Rcu-Assign assigns the address of a new initialized element to an RCU protected pointer. On weakly ordered architectures[1] it ensures that initialization of the element is visible before it is accessible via the RCU protected pointer.

Notice that there are no guarantees as to when changes from a read-side critical section become visible to all other readers unless the critical section is followed by Rcu-Synchronize or Rcu-Call. Similarly, RCU does not guarantee to publish modifications from a reader section even if that reader section is followed by a grace period initiated with Rcu-Synchronize or Rcu-Call but in another thread on a different processor.

2.4 Constraints on implementations

McKenney [12] identified the primary sources of the slowdown of tranditional locking mechanism, i.e. mutexes and reader-writer locks:

- Cache misses due to lock variables.

[1] More on weakly ordered architectures, i.e. architectures without sequentially consistent memory models, in section 2.4

- Atomic operations.
- Memory barriers delimiting critical section code lead to pipeline stalls.

RCU implementations combat the first two sources of overhead in the performance sensitive read-sides with CPU or thread local variables.

Architectures without sequentially consistent memory models, e.g. ARM [1], IA64 [8], PowerPC [29], SPARC[2] [32], and to a lesser extend IA32 [9], are allowed to aggressively reorder memory accesses. In the next example both CPUs may load a value of 0 into their registers *r1* and *r2* when, initially, the memory locations X and Y contain 0.

CPU1	CPU2
X = 1;	Y = 1;
r1 = Y;	r2 = X;

The writes to X and Y occur before the loads of the variables in the instruction stream. In architectures with sequentially consistent memory models one would correctly expect at least one of the registers to store 1. However, CPUs in weakly ordered architectures may issue the loads of X and Y before writing the variables. In particular, both CPUs may first load the initial values 0 of X and Y into the registers and only then proceed to write the new values to memory.

Weakly ordered architectures provide memory barrier instructions that limit how memory operations may be reordered. A full memory barrier forces the CPU to complete any memory operations preceding the memory barrier in the instruction stream before it starts executing memory operations after the memory barrier. Locks surround critical section code with memory barriers that prevent the CPU from issuing loads and stores of protected shared variables outside the protected region. Without the barriers CPUs would be free to load a shared variable before a mutex is acquired; or write to a shared variable after a mutex is released.

RCU algorithms are faced with the same problem but tend to avoid issuing memory barriers in the performance critical Rcu-Read-Lock and Rcu-Read-Unlock. Without memory barriers in these functions, the CPU may be referencing RCU protected data even outside of the read-side critical section. As a consequence, RCU cannot declare that a thread is in a quiescent state just because it is not executing instructions within Rcu-Read-Lock and Rcu-Read-Unlock. Therefore, on a weakly ordered architecture a thread may only enter a quiescent state if it issues a memory barrier. RCU may issue memory barriers in threads on demand during grace period detection or it may wait for naturally occurring memory barriers, e.g. those involved in a context switch.

[2]In relaxed memory order mode.

A thorough explanation of the interactions between memory barrier instructions and CPUs is given in [22].

Chapter 3

Review of RCU algorithms

This chapter gives an overview of RCU algorithms that proved to be relevant to the design of RCU in HelenOS.

It is important to keep in mind that the main resposibility of any RCU is for Rcu-Synchronize and Rcu-Call to provide these guarantees:

- Both functions must wait for the so called preexisting readers to complete, i.e. any readers that may have referenced data prior to the start of the grace period. Preexisting readers complete once they exit their read-side critical section with Rcu-Read-Unlock *and* the CPU executed all of their memory accesses.
- They must ensure that any changes made prior to their invocation are visible to new readers, i.e. the readers for which the functions do not wait to complete before declaring the grace period to be over. Therefore, new readers enter their reader sections after the grace period started.

3.1 General purpose user space RCU

General purpose user space RCU [4], or General URCU, presented in 2012 is an implementation that tolerates preemption in read-side critical sections and does not expect any kernel instrumentation whatsoever. Therefore, it is suitable for user space.

Threads entering a reader section are associated with a reader group that identifies the thread with either preexisting or new readers once a grace period starts. Rcu-Synchronize can then separate preexisting from new readers by changing which group readers entering new critical sections are assigned. A grace period ends when there are no threads inside a critical section associated with the previous group.

Read-side Each RCU-READ-LOCK increments the thread's local RCU read-side critical section nesting counter, *nesting-cnt*. Moreover the outermost RCU-READ-LOCK also locally stores *cur-reader-group* which is a reader group identifier all new readers are assigned and has the value of either 0 or 1. Next, the function issues a memory barrier in order to separate the change of *nesting-cnt* from any memory references in the following reader section.

In order to atomically both increment *nesting-cnt* and affiliate the thread with *cur-reader-group* in the outermost RCU-READ-LOCK, the two logically distinct variables are packed in a single thread local word. For example, *reader-group* may occupy the lowest order bit while the rest of the word is reserved for *nesting-cnt*.

```
RCU-READ-LOCK
1  if thread.nesting-cnt == 0
2      atomically assign packed in a single word:
3          thread.nesting-cnt = 1
4          thread.reader-group = cur-reader-group
5      // Make nesting-cnt visible. Contain memory accesses in critical
6      // section and make changes prior to RCU-SYNCHRONIZE visible.
7      MEMORY-BARRIER      // L
8  else
9      thread.nesting-cnt++
```

Similarly, RCU-READ-UNLOCK contains late memory references of the reader section with a memory barrier before it decrements the read-side nesting counter. This protocol ensures that whenever another CPU notices the outermost decrement of *nesting-cnt*, i.e. that the thread exited its read-side critical section, memory operations in the reader section have already completed.

```
RCU-READ-UNLOCK
1  // Separate decrement from the critical section.
2  MEMORY-BARRIER
3  thread.nesting-cnt--
```

Write-side Simultaneous invocations of RCU-SYNCHRONIZE are serialized with a mutex. The function first executes a memory barrier which separates changes made prior to RCU-SYNCHRONIZE from the start of a new grace period. Next, it starts a new grace period by flipping *cur-reader-group* from 0 to 1, or from 1 to 0. As a result, new readers will associate their *nesting-cnt* with this new group. On the other hand, preexisting readers are assigned the previous group. Once the flip propagates to all CPUs, no new readers will be associated with the

previous group and the number of readers in the previous group will decrease. It eventually reaches zero as the preexisting readers exit their critical sections. Therefore, the function checks every thread if it belongs to preexisting readers and polls it until the thread's nesting count drops to zero.

However, a thread may be preempted before it stores the loaded *cur-reader-group* in RCU-READ-LOCK. While the thread is preempted RCU-SYNCHRONIZE may flip *cur-reader-group* and wait for all preexisting readers to exit. If the thread resumes afterwards its reader section will be erroneously associated with the preexisting reader group even though a grace period is no longer in progress. Should another RCU-SYCHRONIZE follow immediately, it would not wait for this reader to complete. To wait for such readers as well, RCU-SYNCHRONIZE waits in turn for preexisting as well as new readers, which prolongs the grace period.

In addition, the same technique applies even if RCU-READ-LOCK were not preempted but executed between when RCU-SYNCHRONIZE issues its first memory barrier S and flips *cur-reader-group* and, therefore, is incorrectly assigned the previous reader.

RCU-SYNCHRONIZE

```
1  with mutex locked
2      // Make prior changes visible in new reader sections.
3      MEMORY-BARRIER      // S
4      do twice
5          preexisting-group = cur-reader-group
6          // Separate preexisting and new readers.
7          cur-reader-group = ¬cur-reader-group
8          for each thread t
9              while t.IS-PREEXISTING-READER(preexisting-group)
10                 SLEEP(10 ms)
11     // Separate from following changes.
12     MEMORY-BARRIER
```

IS-PREEXISTING-READER(*preexisting-group*)

```
1  return 0 < thread.nesting-cnt
2      and  thread.reader-group == preexisting-group
```

Assessment

General URCU's strengths:

- Read-side critical sections may be preempted.

- First RCU that works in user space without having to explicitly announce quiescent states in client code [5].
- Simple.

Shortcomings:

- Readers have to issue a memory barrier both when entering and leaving read-side critical sections.
- Grace period may be twice as long as it needs to be in the common case.

3.2 Signal based user space RCU

Signal based user space RCU [4], or Signal URCU, is similar to General URCU. Unlike General URCU, Signal URCU does not include memory barriers in its reader sections. Instead it issues memory barriers on demand and only during grace periods by means of signals.

Read-side The authors of Signal URCU have noticed that for the purposes of the RCU algorithm issuing a memory barrier on demand in a signal handler has the effect of promoting compiler barriers, i.e. constructs that forbid the compiler from moving code across them, into full memory barriers. To see this imagine a memory barrier is executed in the reader thread after a compiler barrier. Other CPUs can be sure all accesses prior to the compiler barrier have completed. Similarly, if the memory barrier is executed before the compiler barrier, all accesses after the compiler barrier have not yet begun. The net effect is the same as having a memory barrier in place of the compiler barrier. As a consequence, Signal URCU borrows the implementation of Rcu-Read-Lock and Rcu-Read-Unlock from General URCU, but replaces memory with compiler barriers.

Rcu-Read-Lock

```
1  if thread.nesting-cnt == 0
2      atomically assign packed in a single word:
3          thread.nesting-cnt = 1
4          thread.reader-group = cur-reader-group
5      // Prevent compiler from leaking critical section code.
6      Compiler-Barrier    // L
7  else
8      thread.nesting-cnt++
```

RCU-READ-UNLOCK

```
1  // Prevent compiler from leaking critical section code.
2  COMPILER-BARRIER          // U
3  thread.nesting-cnt−−
```

Write-side Like the read-side, RCU-SYNCHRONIZE too is based on General URCU. The function's memory barrier S pairs up with the memory barriers issued on demand in other threads F-L which in turn promote compiler barriers L to full barriers.

RCU-SYNCHRONIZE

```
 1  with mutex locked
 2      // Make prior changes visible in new reader sections.
 3      MEMORY-BARRIER            // S
 4      // Make the most current nesting-cnt visible to this CPU.
 5      // Ensure new readers see changes prior to RCU-SYNCHRONIZE
 6      FORCE-MB-IN-THREADS       // F-L
 7      do twice
 8          preexisting-group = cur-reader-group
 9          // Separate preexisting and new readers.
10          cur-reader-group = ¬cur-reader-group
11          for each thread t
12              while t.IS-PREEXISTING-READER(preexisting-group)
13                  SLEEP(10 ms)
14      // Prevent any late memory accesses of preexisting readers
15      // from leaking past RCU-READ-UNLOCK.
16      FORCE-MB-IN-THREADS       // F-U
```

On the one hand, if the memory barrier F-L executes in the target thread while it is outside of a critical section, i.e. before the compiler barrier L in RCU-READ-LOCK, the memory barrier F-L paired up with S ensures that any reader section following F-L in the target thread sees all changes made prior to RCU-SYNCHRONIZE. Therefore, RCU-SYNCHRONIZE need not wait for such a reader section.

On the other hand, if the memory barrier F-L executes while the destination thread is in a reader section, i.e. between L of RCU-READ-LOCK and U of RCU-READ-UNLOCK, F-L makes the thread's most recent *nesting-cnt* as well as its group association *reader-group* visible to RCU-SYNCHRONIZE. As a result, RCU-SYNCHRONIZE will correctly wait for the preexisting reader section due to having a positive *nesting-cnt*.

Finally, RCU-SYNCHRONIZE forces a memory barrier in all threads after all preexisting readers have exited. The memory barriers that are the result of *F-U* follow any compiler barriers *U* in RCU-READ-UNLOCK of the preexisting readers and restrict their late memory accesses.

```
IS-PREEXISTING-READER(preexisting-group)
1  return 0 < thread.nesting-cnt
2         and  thread.reader-group == preexisting-group
```

```
FORCE-MB-IN-THREADS
1  send a signal to all threads
2  wait for an acknowledgement
```

```
SIGNAL-HANDLER
1  MEMORY-BARRIER
2  acknowledge
3  MEMORY-BARRIER
```

Assessment

Signal URCU's advantages:

- Works in user space.
- Light-weight read-side critical sections.

Disadvantages:

- Detecting grace periods disrupts readers.
- Grace period may be twice as long as it needs to be in the common case.
- Requires an implementation of signals, which HelenOS does not provide.

3.3 Classic kernel RCU

In Classic Linux kernel RCU [12] and its hierarchical extension [19] reader sections only disable preemption and a regular timer tick drives grace period detection.

```
RCU-READ-LOCK
1  DISABLE-PREEMPTION
```

```
RCU-READ-UNLOCK
1  ENABLE-PREEMPTION
```

The timer tick interrupt detects if a CPU reached a quiescent state, i.e. if the timer interrupted a thread in user mode, an idle loop or if a contex switch occurred since the last timer interrupt. As CPUs pass through quiescent states they are removed from the set of CPUs waiting for the current grace period to end, *wait-for-cpus*. The last one removed announces the end of the current grace period by incrementing the current grace period number, *cur-gp*. Other CPUs eventually notice this change – again in the timer handler – and dispatch waiting Rcu-Call callbacks. To start a new grace period a CPU simply adds all CPUs to the set *wait-for-cpus*.

The algorithm tolerates if the CPU reorders memory accesses out of Rcu-Read-Lock and Rcu-Read-Unlock delimited code because the detected quiescent states inherently include a memory barrier. For example a context switch or a switch to idle loop involve traditional locking which incorporates memory barriers. Similarly, switching to and from user space typically requires serializing instructions that act as memory barriers.

Timer-Handler

```
1  // Check without a lock if wait-for-gp ended.
2  if cpu.wait-for-gp < cur-gp
3       execute callbacks in cpu.cur-cbs
4  // New batch of callbacks is ready.
5  if cpu.cur-cbs == ∅ and cpu.next-cbs ≠ ∅
6       move cpu.next-cbs to cpu.cur-cbs
7       with mutex locked
8            // Callbacks batched during cur-gp wait for an entire g.p.
9            cpu.wait-for-gp = cur-gp + 1
10           // Grace period is not in progress.
11           if wait-for-cpus == ∅
12                wait-for-cpus = all cpus
13 if interrupted user mode, idle loop or context switch occurred
14      with mutex locked
15           wait-for-cpus = wait-for-cpus − this-cpu
16           // This was the last cpu to pass through a q.s.
17           if wait-for-cpus == ∅
18                // Announce that this grace period ended.
19                cur-gp++
20                // Start a new g.p. if some other CPU is waiting for it.
21                if cur-gp ≤ max_{c ∈ cpus} c.wait-for-gp
22                     wait-for-cpus = all cpus
```

Although the timer handler TIMER-HANDLER protects concurrent modifications of *cur-gp* with a mutex it reads the variable without synchronization. Consequently, it may load an older value of *cur-gp* and in the worst case locally queued callbacks *cpu.cur-cbs* will be delayed until the next context switch makes the most recent *cur-gp* visible to the CPU.

RCU-CALL(*callback*)
1 append *callback* to *cpu.next-cbs*

What is more, TIMER-HANDLER illustrates how to amortize grace period detection overhead over multiple RCU callbacks added via RCU-CALL. Instead of initiating a new grace period for each callback, TIMER-HANDLER batches incomming callbacks while a grace period is in progress in *next-cbs* and dispatches all of them at the end of the next grace period.

Assessment

Classic RCU's advantages:

- Extremely low overhead reader sections that just disable and enable preemption.
- Low overhead grace period detection, which only observes but never forces context switches, user or idle mode.
- All CPUs detect grace periods in parallel.

Disadvantages:

- Requires a regular sampling tick.
- Read-side critical sections may not be preempted.
- Grace period detection is held back if threads are alloted larger time quanta that they spend in the kernel.

3.4 Houston's RCU

Houston's RCU [7] removes Classic RCU's reliance on regular timer ticks. In Classic RCU the timer tick was used to batch callbacks, detect local quiescent states and dispatch callbacks whose grace periods had ended. Houston moves these resposibilities to RCU-CALL and RCU-READ-UNLOCK. It batches callbacks in RCU-CALL until a predefined number of queued callbacks is reached. In

addition, RCU-READ-UNLOCK announces quiescent states if a grace period is in progress. Both RCU-CALL and RCU-READ-UNLOCK detect that a grace period elapsed but leave it up to a designated thread to invoke callbacks.

For brevity the pseudocode below omits the interaction between RCU-CALL and the designated background thread. Instead it accurately portraits how the grace period detection code employs atomic instructions to communicate with the read-side critical section.

Read-side Before RCU-READ-LOCK allows instructions of the reader section to execute it marks that the CPU is in a read-side critical section and orders the two actions with a memory barrier. Likewise, RCU-READ-UNLOCK clears the CPU's mark with ATOMIC-EXCHANGE which acts as a memory barrier and atomically clears the mark and returns its previous value. The previous mark indicates if a grace period started while the CPU was running a reader section. In that case, RCU-READ-UNLOCK wakes up any threads waiting for the grace period to end.

```
RCU-READ-LOCK
1  DISABLE-PREEMPTION
2  if ++cpu.nesting-cnt == 1
3       cpu.flags = IN-CS
4       MEMORY-BARRIER
```

```
RCU-READ-UNLOCK
1  if --cpu.nesting-cnt == 0
2       prev-flags = ATOMIC-EXCHANGE(cpu.flags, NOT-IN-CS)
3       if ANNOUNCE-QS ∈ prev-flags
4            preexisting-readers.SEMAPHORE-UP(1)
5  ENABLE-PREEMPTION
```

Write-side RCU-SYNCHRONIZE examines each CPU with an atomic compare and swap instruction. It atomically checks if the CPU is in a read-side critical section, and if it is it requests that the CPU announce a quiescent state once in RCU-READ-UNLOCK. Finally, the function waits for all preexisting readers to announce a quiescent state.

```
RCU-SYNCHRONIZE
1  with mutex locked
2      cur-gp++
3      preex-cnt = 0
4      for each cpu ∈ cpus
5          prev-flags = CAS(cpu.flags, IN-CS, IN-CS ∪ ANNOUNCE-QS)
6          if prev-flags == IN-CS
7              preex-cnt++
8      preexisting-readers.SEMAPHORE-DOWN(preex-cnt)
```

Assessment

Pros of Houston's RCU:

- No regular sampling tick.
- Low latency grace period end detection.

Cons may be:

- Noticeable overhead of RCU-READ-UNLOCK due to the atomic exchange instruction.
- Still requires a timer albeit an infrequent one.
- Non-preemptible reader sections.

3.5 Podzimek's RCU

Podzimek's RCU implementation for the OpenSolaris UTS kernel [30] does not rely on regular timer ticks to detect naturally occurring quiescent states like Classic RCU. Instead of sampling reader state each timer tick during grace period detection, Podzimek's RCU explicitly requests that the first reader to notice the start of a new grace period on each CPU announces a quiescent state in RCU-READ-LOCK or RCU-READ-UNLOCK. What is more, read-side critical sections are non-preemptible and context switches are recognized as naturally occurring quiescent states. Once all CPUs announce a quiescent state or perform a context switch the grace period ends.

Read-side Grace period detection starts by incrementing the global current grace period number *cur-gp*, i.e. the number of the grace period we are waiting to complete. To notice that a new grace period started, readers compare the changed

cur-gp to a value of it stored locally on each CPU, *last-seen-gp*. *last-seen-gp* is the last grace period number for which the CPU noted and announced a quiescent state. Readers check for the change of *cur-gp* in unnested RCU-READ-LOCK and RCU-READ-UNLOCK as these functions represent a quiescent state. In order to announce that it reached a quiescent state, a reader first executes a memory barrier that contains memory references within the critical section. Next, it notes that it passed a quiescent state by updating *last-seen-gp* to *cur-gp*.

```
RCU-READ-LOCK
1  DISABLE-PREEMPTION
2  CHECK-QS
3  cpu.nesting-cnt++
```

```
RCU-READ-UNLOCK
1  cpu.nesting-cnt−−
2  CHECK-QS
3  ENABLE-PREEMPTION
```

```
CHECK-QS
1  if 0 == cpu.nesting-cnt
2       if cpu.last-seen-gp ≠ cur-gp
3            gp = cur-gp
4            MEMORY-BARRIER
5            cpu.last-seen-gp = gp
```

CHECK-QS checks for and announces quiescent states at reader section boundaries. It first makes sure it is not invoked from a nested reader section on line 1. Next, line 2 verifies that a new grace period started and avoids issuing a memory barrier if the CPU is not holding up the current grace period. The memory barrier on line 4 represents a quiescent state. Firstly, it contains references to protected data within the critical section. Secondly, it makes changes prior to RCU-SYNCHRONIZE visible in the following critical section. Lastly, it captures delayed memory accesses from previous critical sections that however RCU-READ-UNLOCK exited without noticing that a new grace period had started. The function finally acknowledges that it passed a quiescent state by writing to *cpu.last-seen-gp* on line 5 for other CPUs to see.

Write-side RCU-SYNCHRONIZE waits a while after starting a grace period and then polls each CPU's *last-seen-gp* and context switch counter to see if the CPU reached a quiescent state in the meantime. If a CPU did not record one, e.g.

because it had been running a thread without any read-side critical sections, a forced context switch is scheduled on that CPU. Once all forced context switches finish, all CPUs must have passed a quiescent state and the grace period ends.

RCU-SYNCHRONIZE

```
1  // Separates prior changes of shared variables from new grace period.
2  MEMORY-BARRIER
3  with mutex locked
4      // Start new grace period.
5      cur-gp++
6      // Gather CPUs potentially in reader sections.
7      reader-cpus = ∅
8      for each cpu
9          if ¬cpu.idle and cpu.last-seen-gp ≠ cur-gp
10             cpu.last-ctx-switch-cnt = cpu.ctx-switch-cnt
11             reader-cpus = cpu ∪ reader-cpus
12     // Give preexisting reader sections a chance to exit.
13     SLEEP(10 ms)
14     // Force context switch on CPUs that may not have reached a q.s.
15     for each cpu ∈ reader-cpus
16         may-be-reading = ¬cpu.idle
17             and cpu.last-seen-gp ≠ cur-gp
18             and cpu.last-ctx-switch-cnt == cpu.ctx-switch-cnt
19         if may-be-reading
20             force and wait for a context switch on cpu
```

RCU-SYNCHRONIZE and CHECK-QS read *cpu.last-seen-gp* and *cur-gp* respectively without synchronization and rely on cache coherency protocols to propagate updates of these variables to all CPUs. Nevertheless, the algorithm does not rely on the variables for correctness and only uses them to make grace period detection less intrusive. In the worst case, if cache coherence fails to deliver the most recent *cur-gp* to a CPU or propagate its write of *last-seen-gp* back to RCU-SYNCHRONIZE, a context switch is forced on the CPU and the algorithm continues correctly.

While the above pseudo code captures the main idea of the algorithm, Podzimek's RCU is much more elaborate. First, it implements RCU-SYNCHRONIZE on top of RCU-CALL. Second, it batches multiple RCU callbacks during a grace period; therefore, spreading the overhead of detecting a grace period among multiple callbacks. Third, it processes the callbacks in CPU-bound background reclaimer

threads and leaves detection up to a dedicated detector thread. Last, the implementation is capable of speeding up the detection if a callback is declared to be expedited.

Assessment

Advantages of Podzimek's RCU:

- Low overhead read-sice critical sections with at most one memory barrier issued by each CPU per grace period.
- No regular sampling timer ticks are required.
- Does not needlessly wake idle CPUs up. As a result, it is more power efficient.
- Works in interrupt handlers.
- No instrumentation of the scheduler or the interrupt dispatching mechanism necessary.

Disadvantages of the algorithm may include:

- Intrusiveness of the detection depends on cache coherency.
- Read-side critical section may not be preempted.

3.6 Sleepable RCU

Sleepable RCU [10] introduced in the Linux kernel in 2012 allows threads in reader sections to not only be preempted but also to sleep.

Similarly to General URCU, readers in Sleepable RCU can be associated with two different reader groups. Once grace period detection starts Sleepable RCU changes the reader group of which new reader sections should be a member. Because no new readers will become members of the previous group the number of readers in that group eventually drops to zero. A grace period then ends as all preexisting readers must have been associated with the previous reader group.

Read-side Because Sleepable RCU operates in a kernel environment it does not assign a reader group to individual threads. Instead it maintains two counters per each CPU tracking the number of readers in each group on that CPU and modifies the counters with disabled preemption.

RCU-READ-LOCK

```
1  with disabled preemption
2      thread.reader-group = cur-reader-group
3      cpu.reader-cnt[thread.reader-group]++
4      MEMORY-BARRIER      // L
5      cpu.cs-seq[thread.reader-group]++
```

RCU-READ-LOCK associates the upcoming reader section with the current group with an increment on line 3. Next, it issues a memory barrier L to separate the increment from code of the reader section. Last, line 5 increases the sequence number of completed calls to RCU-READ-LOCK on the current processor. The sequence number enables RCU-SYNCHRONIZE to detect new readers that have been mistakenly associated with the previous readers group.

RCU-READ-UNLOCK

```
1  with disabled preemption
2      MEMORY-BARRIER      // U
3      cpu.reader-cnt[thread.reader-group] − −
```

RCU-READ-UNLOCK issues a memory barrier U on line 2 to contain memory accesses of the reader section before pronouncing the reader section finished with a decrement on line 3.

Write-side Quite unline General URCU, Sleepable RCU does not require two group flips per grace period. Exactly as in General URCU, the current reader group *cur-reader-group* may be flipped between when RCU-READ-LOCK loads the value of *cur-reader-group* on line 2 and associates the thread with the group on line 3. Such a reader would incorrectly become a member of the previous reader group although it is a new reader, i.e. it sees all changes prior to RCU-SYNCHRONIZE and will not be waited for.

RCU-SYNCHRONIZE

```
1  with mutex locked
2      // Make prior changes visible to new readers.
3      MEMORY-BARRIER
4      delayed-reader-grp = ¬cur-reader-group
5      WAIT-FOR-READERS(delayed-reader-grp)
6      // Do not mix group flip with WAIT-FOR-READERS.
7      MEMORY-BARRIER
8      cur-reader-group = ¬cur-reader-group
9      // Do not mix group flip with WAIT-FOR-READERS.
10     MEMORY-BARRIER
11     preex-reader-grp = ¬cur-reader-group
12     WAIT-FOR-READERS(preex-reader-grp)
```

Instead of flipping *cur-reader-group* twice in a row, Sleepable RCU maintains an invariant that RCU-SYNCHRONIZE finds the previous reader group empty upon entry before flipping the current group. In particular, it explicitly waits for such readers on line 5 with WAIT-FOR-READERS before flipping the current reader group on line 8. Then, a grace period elapses when all preexisting readers have exited their critical sections with line 12.

What is more, given that readers will be assigned an incorrect group only rarely one can expect that there will be no delayed readers in the previous group when entering RCU-SYNCHRONIZE. Therefore, compared to General URCU the grace period length is effectively cut in half.

WAIT-FOR-READERS(*preex-reader-grp*)

```
1  forever
2      prev-seq-sum = Σ_{c ∈ cpus} c.cs-seq[preex-reader-grp]
3      MEMORY-BARRIER              // A
4      approx-active-cnt = Σ_{c ∈ cpus} c.reader-cnt[preex-reader-grp]
5      if approx-active-cnt == 0
6          MEMORY-BARRIER          // B
7          cur-seq-sum = Σ_{c ∈ cpus} c.cs-seq[preex-reader-grp]
8          if cur-seq-sum == prev-seq-sum
9              return
10     SLEEP(10 ms)
```

WAIT-FOR-READERS waits for preexisting readers of the group *preex-reader-grp* to exit their reader sections. It determines the approximate number of preexist-

ing readers in the group *preex-reader-grp*, *approx-active-cnt*, on line 4. It first issues a memory barrier A that pairs up with memory barrier L in RCU-READ-LOCK. For preexisting readers L happens before A; therefore, the increment of *reader-cnt* in RCU-READ-LOCK is incorporated into the sum *approx-active-cnt*. Consequently, *approx-active-cnt* will be positive and WAIT-FOR-READERS will wait for the preexisting readers to exit their reader sections with line 5.

On the other hand, when L happens after A *approx-active-cnt* may not reflect the increment of *reader-cnt*. However, because A happened before L the reader will see all changes prior to RCU-SYNCHRONIZE and is, therefore, a new reader that WAIT-FOR-READERS does not have to wait for.

Whereas the function may not see a new reader's increment in RCU-READ-LOCK while summing up *reader-cnt*s, it may incorrectly incorporate its decrement of *reader-cnt* in RCU-READ-UNLOCK into the sum if the new reader is migrated to a different CPU and immediately invokes RCU-READ-UNLOCK while WAIT-FOR-READERS is adding up *reader-cnt*s. In that case, memory barrier B pairs up with U. Because changes to the sequence number *cs-seq* are separated from decrements of *reader-cnt* with U, B guarantees the function will see the current sequence number on line 7. What is more, the previous sum of sequence numbers on line 2 could not have included the new reader's update of the sequence because the change is separated from the sum on line 2 with memory barries A and L. As a result, if a new reader skews *approx-active-cnt* with only a decrement of *reader-cnt* the sums of sequence numbers before and after line 4, *prev-seq-sum* and *cur-seq-sum*, will differ. Line 8 detects this case and avoids exiting prematurely.

Assessment

Advantages of Sleepable RCU:

- Reader sections may be preempted and even sleep.
- Small grace period detection latency.

Disadvantages may include:

- Both entering and existing read-side critical section involves memory barriers.

3.7 Preemptible RCU in Linux

The preemptible Linux kernel RCU [15, 25], or simply Preemptible RCU, allows readers to be preempted within their reader sections.

The algorithm evolved from a complex design [15, 25] to the current simplified version[1]. After finishing the implementation of A-RCU, which is our implementation of RCU in HelenOS, Preemptible RCU moved away from running RCU callbacks in softirq context [24]. Instead, callbacks are processed in dedicated kernel threads similarly to the design employed in Podzimek's RCU. Furthermore [24] introduced another background kernel thread that is responsible for starting new grace periods and announcing the end of old ones; a concept first presented by Podzimek's RCU. What is more, the current version of Preemptible RCU was inspired by user mode RCU algorithms described in the previous sections as mentioned in [23, 20]. Although A-RCU was developed independently from the presented Preemptible RCU, it is based on Signal URCU and concepts of Podzimek's RCU. Consequently, Preemptible RCU's read-side in the most recent incarnation resembles A-RCU's read-side.

Read-side Entering and leaving a reader section involves only incrementing or decrementing the thread's reader section nesting count, *nesting-cnt*.

```
RCU-READ-LOCK
1  thread.nesting-cnt++
2  COMBILER-BARRIER

RCU-READ-UNLOCK
 1  if thread.nesting-cnt ≠ 1
 2      thread.nesting-cnt−−
 3  else
 4      COMBILER-BARRIER
 5      thread.nesting-cnt = INT-MIN
 6      COMBILER-BARRIER
 7      if thread.special-unlock-needed
 8          RCU-READ-UNLOCK-SPECIAL
 9      COMBILER-BARRIER
10      thread.nesting-cnt = 0
```

If a reader section is preempted or RCU wishes to detect the end of a grace period as soon as possible RCU-READ-UNLOCK finds *special-unlock-needed* set to true. In turn, RCU-READ-UNLOCK-SPECIAL records a quiescent state for the current CPU and clears *special-unlock-needed*.

In order to avoid a race with interrupt handlers utilizing RCU, the outermost RCU-READ-UNLOCK first sets the nesting count to the smallest integer value

[1] Version 3.10.4 released on the 28th of July 2013

on line 5. If an interrupt handler were to interrupt the outermost Rcu-Read-Unlock and execute a reader section the handler's Rcu-Read-Unlock would not enter the else branch on lines 3-10 because *nesting-cnt* would be either negative or equal to 2 due to line 1 of the handler's Rcu-Read-Lock. Therefore, Rcu-Read-Unlock-Special is guaranteed to be invoked only once for each time *special-unlock-needed* is set to true.

Write-side Preemptible RCU's write side resembles the write side of Classic RCU. It too is driven by a regular clock tick. Firstly, the timer handler is responsible for detecting that a new batch of callbacks is ready and, therefore, a grace period should be started. The handler wakes up a background kernel thread that initiates a new grace period if one is not already in progress.

Secondly, the handler samples threads running on the CPU and records that the CPU passed a quiescent state in a bitmap if the interrupted thread is outside of a reader section. Because memory accesses are separated from changes of the nesting count with compiler barriers on line 2 of Rcu-Read-Lock and line 4 of Rcu-Read-Unlock the interrupted thread or interrupt handler is guaranteed to be outside of a reader section whenever *nesting-cnt* is zero.

Thirdly, the timer handler detects that the current grace period had ended. A grace period ends once all CPUs pass a quiescent state and all threads preempted in a reader section have exited their reader sections. Then, the timer handler or Rcu-Read-Unlock-Special wake up the background kernel thread that started the current grace period to clean up data structures used during the grace period. Finally, the kernel thread updates the number of completed grace periods thereby announcing that the current grace period ended.

Lastly, the timer handler instructs another dedicated background kernel thread to dispatch the relevant callbacks that were queued on the current CPU for grace periods which had already completed.

All in all Preemptible RCU's write side is rather intricate as it must handle CPU hotplugging and tries to avoid interrupting idle CPUs with the regular clock tick. The resulting design involves instrumentation of interrupt and exception handling code as well as transitions to the idle mode.

Assessment

Advantages of Preemptible RCU in Linux:

- Reader sections may be preempted.
- Functions in interrupt and exception handlers.

- Low overhead reader sections; RCU-READ-LOCK consists of an increment while RCU-READ-UNLOCK contains only simple instructions and a handful of conditional statements.

Disadvantages might include:

- Instrumentation of exception and interrupt dispatching is necessary.

3.8 Summary

While other RCU algorithms exist [18, 14, 17, 21, 16] the reviewed algorithms proved to be relevant to the design of RCU for HelenOS.

Firstly, Podzimek dispatches RCU callbacks in background reclaimer threads which places little restrictions on the callbacks and does not require interrupt handler instrumentation. Therefore, we adopted the concept.

Moreover Signal URCU offers read-side critical sections with little overhead. Consequently, the implemented user space RCU in HelenOS is based on Signal URCU. What is more, we used the ideas in Signal URCU when designing a preemptible kernel RCU.

Lastly, the next chapter demonstrates that an invariant of Sleepable RCU can be applied to Signal URCU as well and remove the need for two counter flips.

Chapter 4

Implemented RCU algorithms

4.1 Requirements

Due to the research nature of HelenOS a user space RCU implementation must take into account the following.

- URCU must not impose design concepts of legacy systems on HelenOS, e.g. it must not require signals for its operation.

The requirements listed next must guide the implementation of a kernel RCU for HelenOS.

- First and foremost, a kernel RCU must tolerate reader sections in both interrupt as well as exception handlers in order to be universally usable.
- Second, it should allow Rcu-Call in interrupt and exception contexts for the same reason.
- Third, it should not disable preemption for the duration of reader sections unless such an implementation would incur a high performance penalty. Otherwise, disabling preemption might unnecessarily limit scheduling latency.

4.2 Preemptible kernel A-RCU

Preemptible kernel A-RCU is based on Signal URCU and the ideas of Podzimek's RCU. It meets all the set requirements. Not only does it support reader sections and Rcu-Calls in interrupt and exception handlers, it also leaves preemption enabled in reader sections without loss of performance.

Similarly to Signal URCU, A-RCU reader sections do not issue memory barriers and in the common case only increment and decrement a reader section

nesting counter. Like Signal URCU, A-RCU drives grace period detection from a single thread that polls other threads until they have exited their reader sections. However, A-RCU exploits that it runs in the kernel and that it can keep track of preempted readers explicitly. Therefore, instead of polling all threads in the system, A-RCU polls only threads actually running on CPUs and tracks threads that were preempted in a reader section in a linked list.

Inexpensive tracking of preempted readers In performance sensitive applications of RCU, one expects a thread to enter many reader sections within a time slice. Therefore, context switches will preempt relatively few reader sections and requiring expensive operations when preempted readers exit their reader sections will not introduce a significant overhead.

As a result, to detect a grace period, A-RCU first samples each CPU to determine if it passed a quiescent state. Once all CPUs pass a quiescent state A-RCU checks if there are any preempted readers holding up the current grace period. If there are it sleeps and requests that the last preempted reader that is holding up the current grace period wake up A-RCU's grace period detecting thread from Rcu-Read-Unlock.

Sampling CPUs for quiescent states A-RCU samples CPUs for quiescent states somewhat similarly to Classic RCU. Instead of relying on regular timer interrupts, A-RCU interrupts the sampled CPU with an interprocessor interrupt (IPI). In the IPI handler of the sampled CPU, A-RCU checks the interrupted thread's reader section nesting count and if it finds the thread outside of a reader section the handler issues a memory barrier. The memory barrier represents a quiescent state on weakly ordered architectures.

While A-RCU's IPIs are more intrusive than Classic RCU's timer interrupts, A-RCU only interrupts other processors when grace period detection is in progress. In order to further reduce the overhead of polling other processors, it non-intrusively detects context switches on other processors and only samples CPUs that have not yet announced that they performed a context switch. CPUs announce the event with Podzimek's protocol of announcing quiescent states. Each processor locally stores the number of the last grace period for which it entered a quiescent state, e.g. during a context switch. If the current global grace period number does not match the locally stored value during a context switch, the CPU enters a quiescent state by issuing a memory barrier. Then, it announces that the CPU passed a quiescent state by updating the locally stored value for other CPUs to read.

Tracking preempted readers When a reader section is preempted for the first time, A-RCU must decide whether the preempted reader is a preexisting reader and it should hold up the current grace period; or it is a new reader that may hold up the next grace period if it does not exit its reader section in time. Because it is the first time the reader was preempted, it only ran on one CPU. If that CPU had not passed a quiescent state since the start of the grace period the preempted reader may have started before the grace period did. Therefore, it is considered to be a preexisting reader and it is placed into a list of such readers.

Otherwise, the CPU noted a quiescent state after the start of the grace period and the preempted reader will definitely see all changes prior to the grace period. The reader is, therefore, a new reader. Although the new reader will not hold up the current period, it is placed into a list of readers that may hold up the next grace period if they do not exit their reader sections before the next period starts.

It is important to note that A-RCU only waits for preempted readers holding up the current grace period after all CPUs passed a quiescent state. Since all CPUs passed a quiescent state all future preempted readers will be associated with the new reader list. Consequently, the number of threads in the preexisting reader list will monotonically decrease and once the last reader in the list exits its reader section the grace period is truly over.

Reclaimer and detector threads A-RCU borrows the concept of Podzimek's reclaimer threads. Each CPU has a single bound reclaimer thread that dispatches batched callbacks in the background. The first reclaimer thread to notice new locally queued callbacks starts a new grace period and takes on the role of a detector thread. When the grace period is over the detector thread signals the event to other reclaimer threads. In other words, A-RCU does not have a dedicated detector thread like Podzimek's RCU.

Pseudo code

The pseudo code below illustrates the algorithm in greater detail.

Read-side Rcu-Read-Lock and Rcu-Read-Unlock only adjust the level of nested reader sections. If a reader section is preempted the reader's nesting count is marked with a Was-Preempted flag. Rcu-Read-Unlock checks for this unlikely event and invokes Preempted-Unlock.

```
Rcu-Read-Lock
1  thread.nesting-cnt++
2  Combiler-Barrier
```

```
RCU-READ-UNLOCK
1  COMBILER-BARRIER
2  thread.nesting-cnt−−
3  if thread.nesting-cnt == WAS-PREEMPTED
4      PREEMPTED-UNLOCK

PREEMPTED-UNLOCK
1  if WAS-PREEMPTED == ATOMIC-EXCHANGE(thread.nesting-cnt, 0)
2      with preempt-lock locked
3          remove thread from some preempted list
4          if cpu.cur-preempted is now empty and det-waiting
5              det-sema.SEMAPHORE-UP
```

PREEMPTED-UNLOCK atomically clears the flag WAS-PREEMPTED which ensures that in a race between the thread's PREEMPTED-UNLOCK and an interrupt handler with a reader section only one succeeds in clearing the flag and properly unlocks the section. On lines 2-6, the function checks if it unlocked the last preempted reader that was holding up the current grace period and if so, notifies the sleeping detector thread.

Write-side RCU-CALL adds a new callback represented as a linked list node *item-ptr* at the end of the arriving callbacks list of the local CPU, *arriving-cbs*. It first adjusts the list tail pointer *arriving-cbs-tail* atomically with respect to interrupts on the local CPU on line 2. The line ensures RCU-CALL will not corrupt the list even in the presence of nested RCU-CALLs in exception handlers. Next, the function attaches the callback's node to the list on line 3. If this was the first callback queued locally, line 6 notifies the CPU's reclaimer thread that a new batch is ready for processing.

```
RCU-CALL(item-ptr)
1  with disabled preemption
2      prev-tail = ATOMIC-EXCHANGE(cpu.arriving-cbs-tail, &item-ptr.next)
3      *prev-tail = item-ptr
4      first-cb = (prev-tail == &cpu.arriving-cbs)
5      if first-cb
6          cpu.cbs-arrived-sema.SEMAPHORE-UP
```

Reclaimer threads run in a never-ending loop. First, a reclaimer waits for the current grace period to end with WAIT-FOR-GP-END on line 5. Second, it makes sure that there are RCU-CALLs that have yet to be dispatched on line

2. If there are none Wait-For-Callbacks waits for Rcu-Call to add a new callback and wake the reclaimer up via a semaphore. Next, the thread runs any callbacks for the grace period that has just elapsed and, finally, waits for another grace period to end on line 5.

```
Reclaimer-Thread
1  forever
2      Wait-For-Callbacks
3      Exec-Callbacks
4      Advance-Callbacks
5      Wait-For-GP-End
```

A-RCU makes use of three lists of callbacks. *cur-gp-cbs* represents the functions that should be run after the current grace period ends, i.e. functions that have been added before the current grace period started. These are the functions Exec-Callbacks invokes. Furthermore, *next-gp-cbs* holds Rcu-Calls that can only be satisfied once the next grace period elapses. Finally, *arriving-cbs* contains the callbacks that have been added since the last call to Advance-Callbacks.

After every grace period Advance-Callbacks moves the items of *next-gp-cbs* to *cur-gp-cbs* and the elements in *arriving-cbs* to *next-gp-cbs*. Placing items into the intermediary list *next-gp-cbs* is necessary in order to avoid executing callbacks prematurely. In particular, when the reclaimer calls Advance-Callbacks another reclaimer may have already reached line 5 and started a new grace period. Therefore, some items of *arriving-cbs* may have been added when a new grace period was already in progress. If the function were to move *arriving-cbs* directly to *cur-gp-cbs*, the callbacks added after the current grace period started but before it had a chance to complete would be invoked prematurely.

Reclaimer threads wait for a grace period with Wait-For-GP-End. The function first checks if a grace period is already in progress on line 3 by comparing the last completed grace period number *completed-gp* to the current grace period number *cur-gp*. If they match, a new grace period is started by incrementing *cur-gp* on line 10 and moving readers that were preempted after the previous grace period started in *next-preempted* to *cur-preempted* which is the list of preexisting readers for the newly initiated grace period.

```
WAIT-FOR-GP-END
 1  with gp-lock locked
 2      // A grace period is already in progress
 3      if completed-gp ≠ cur-gp
 4          wait for "end of grace period" signal
 5          return
 6      else
 7          // Start a new grace period
 8          with preempt-lock locked
 9              move readers in next-preempted to cur-preempted
10              cur-gp++
11  WAIT-FOR-READERS
12  with gp-lock locked
13      completed-gp = cur-gp
14  signal "end of grace period"
```

Next, WAIT-FOR-READERS waits for preexisting readers to complete before WAIT-FOR-GP-END declares the grace period to be over. WAIT-FOR-READERS first ensures all CPUs reached a quiescent state on lines 2-7. Only then can the function be sure no future readers will be associated with the list of readers delaying the current grace period *cur-preempted*. Therefore, the list will eventually become empty and it is safe to wait for the last reader in *cur-preempted* to notify WAIT-FOR-READERS that it exited its reader section.

```
WAIT-FOR-READERS
 1  // Poll CPUs until all reach a quiescent state
 2  reader-cpus = active cpus
 3  while reader-cpus ≠ ∅
 4      SLEEP(10 ms)
 5      reader-cpus = reader-cpus ∩ all non-idle cpus
 6          where cpu.last-seen-gp ≠ cur-gp
 7      sample reader-cpus with an IPI
 8  with preempt-lock locked
 9      // Preempted readers delaying the current g.p. exist
10      if cur-preempted ≠ ∅
11          det-waiting = TRUE
12  if cur-preempted was nonempty
13      det-sema.SEMAPHORE-DOWN
```

The sampling IPI-HANDLER checks if it interrupted a thread in a reader section on a CPU that has yet to note a quiescent state and if not it enters a

quiescent state with a memory barrier on line 4. The memory barrier contains any late accesses of reader sections that have just exited.

```
IPI-Handler
1  if 0 < thread.nesting-cnt and cpu.last-seen-gp ≠ cur-gp
2      // Still in a reader section. Poll again later.
3  else
4      Memory-Barrier
5      cpu.last-seen-gp = cur-gp
```

The context switch handler Context-Switch marks preempted reader's nesting count with the flag Was-Preempted to instruct the reader it may have to notify a detector waiting for it to complete in Preempted-Unlock. If the reader was preempted for the first time, Context-Switch associates it with the preexisting preempted readers list *cur-preempted* or with new preempted readers list *next-preempted*. Acquiring the *preempt-lock* guarantees Context-Switch has the most up-to-date value of the current grace period number *cur-gp* and associates the preempted reader with the correct list.

Lastly, Context-Switch may record a quiescent state for this CPU on lines 8-11 because a potentially preempted reader was already noted globally in the proper list.

```
Context-Switch
 1  if 0 < thread.nesting-cnt and
 2      Was-Preempted ∉ AtomicMark(nesting-cnt, Was-Preempted)
 3          with preempt-lock locked
 4              if cpu.last-seen-gp ≠ cur-gp
 5                  add thread to cur-preempted
 6              else
 7                  add thread to next-preempted
 8  if cpu.last-seen-gp ≠ cur-gp
 9      gp = cur-gp
10      Memory-Barrier
11      cpu.last-seen-gp = gp
```

Assessment

Advantages:

- Reader sections may be preempted.
- Functions in interrupt and exception handlers.

- Low overhead reader sections.
- Clearly separated from the rest of the system; no need to instrument exception and interrupt handlers.
- RCU processing is contained in dedicated threads which simplifies keeping track of the cost of RCU when both detecting grace periods and running callbacks.

Disadvantages might include:

- Polling CPUs with interprocessor interrupts may become disruptive in large systems with tens of CPUs.

4.3 Preemptible Podzimek's kernel RCU

Preemptible Podzimek's RCU, or simply PP-RCU, is our modification to Podzimek's RCU to make reader sections preemptible. It borrows the mechanism to track preempted readers from A-RCU. Moreover PP-RCU does not force context switches on CPUs delaying a grace period. Instead, it requests such CPUs to wake the detector up when they pass a quiescent state.

Read-side PP-RCU's read-side is very similar to Podzimek's RCU. Rcu-Read-Lock is exactly the same as Podzimek's but it also enables preemption before continuing with the reader section code.

```
Rcu-Read-Lock
1  with disabled preemption
2      if 0 == thread.nesting-cnt
3          Record-QS
4      thread.nesting-cnt++
```

Unlike in Podzimek's RCU Rcu-Read-Unlock must first disable preemption that had been enabled when leaving Rcu-Read-Lock. In addition to Podzimek's original function lines 4-5 check if the reader should signal to a detector thread that it had exited its critical section.

```
Rcu-Read-Unlock
1  with disabled preemption
2      if 0 == − − thread.nesting-cnt
3          Record-QS
4          if thread.was-preempted or cpu.is-delaying-gp
5              Signal-Unlock
```

```
RECORD-QS
1  if cpu.last-seen-gp ≠ cur-gp
2      gp = cur-gp
3      MEMORY-BARRIER
4      cpu.last-seen-gp = gp

SIGNAL-UNLOCK
1  if true == ATOMIC-EXCHANGE(cpu.is-delaying-gp, false)
2      remaining-readers.SEMAPHORE-UP
3  if true == ATOMIC-EXCHANGE(thread.was-preempted, false)
4      with preempt-lock locked
5          remove thread from some list of preempted threads
6          if cpu.cur-preempted is now empty and preempted-blocking-gp
7              preempted-blocking-gp = false
8              remaining-readers.SEMAPHORE-UP
```

RCU-READ-UNLOCK invokes SIGNAL-UNLOCK if the CPU is delaying the current grace period or if the thread was preempted while inside a reader section. In the former case, the function wakes up the detector via a semaphore on line 2 because it has just passed a quiescent state. The latter case is handled in line with A-RCU. In particular, if the thread was the last thread in the list of preempted preexisting readers for the grace period, *cur-preempted*, and the detector is blocked waiting for the such a thread, SIGNAL-UNLOCK wakes the detector up in line 8.

Furthermore the function checks for both conditions with an ATOMIC-EXCHANGE in order to execute each of the branches at most once even in the face of a nested SIGNAL-UNLOCK running in an exception handler.

Write-side Although PP-RCU employs the same callback batching and dispatching mechanism as A-RCU and implements RCU-SYNCHRONIZE in terms of RCU-CALL, we list the pseudocode for RCU-SYNCHRONIZE as if all the steps were invoked in place and not in a detector thread. We hope to highlight PP-RCU's algorithmic differences from Podzimek's RCU and reduce duplication with A-RCU.

RCU-SYNCHRONIZE increments the grace period number *cur-gp* to start a new grace period. Incrementing *cur-gp* instructs readers to note a proper quiescent state with a memory barrier in RECORD-QS.

Next, line 8 gives CPUs some time to pass through a quiescent state[1] before proceeding to sample CPUs with interprocessor interrupts. Lines 10-12 interrupt

[1] And also to batch callbacks in the actual implementation.

those CPUs which may yet have to pass a quiescent state as viewed from RCU-SYNCHRONIZE's detecting thread.

RCU-SYNCHRONIZE

```
 1  // Separates prior changes of shared variables from new grace period.
 2  MEMORY-BARRIER
 3  with mutex locked
 4      // Start new grace period
 5      with gp-lock locked
 6          cur-gp++
 7          move next-preempted to cur-preempted
 8      SLEEP(10 ms)
 9      // Request CPUs without a q.s. to notify us when they reach one.
10      for each cpu
11          if ¬cpu.idle and cpu.last-seen-gp ≠ cur-gp
12              Sample cpu with an IPI
13      // Wait for CPU delaying the g.p. to pass a q.s.
14      remaining-readers.SEMAPHORE-DOWN(delaying-cpu-cnt)
15      // Wait for preempted readers holding up the g.p.
16      with preempt-lock locked
17          // Preempted readers delaying the current g.p. exist
18          if cur-preempted ≠ ∅
19              preempted-delaying-gp = TRUE
20      if cur-preempted was nonempty
21          remaining-readers.SEMAPHORE-DOWN
```

The interrupt handler IPI-HANDLER runs on the interrupted CPU. Therefore, it does not have to rely on cache coherence to determine whether the CPU has reached a quiescent state. The IPI handlers enter a quiescent state on line 6 if the CPU is not in a reader section. Otherwise, the handler instructs its CPU to notify RCU-SYNCHRONIZE of a quiescent state as soon as possible, e.g. from SIGNAL-UNLOCK right after the reader exits the critical section. In addition, the handlers determine the number of the CPUs that are delaying the grace period.

When the IPI handlers return, RCU-SYNCHRONIZE waits for the CPUs that have yet to reach a quiescent state to pass one with line 14. After all CPUs have recorded a quiescent state, lines 15-21 follow A-RCU's protocol of waiting for the last preempted reader holding up the current grace period to exit.

IPI-Handler

```
1  if cpu.last-seen-gp ≠ cur-gp
2      if 0 < thread.nesting-cnt
3          cpu.is-delaying-gp = true
4          Atomic-Inc(delaying-cpu-cnt)
5      else
6          Record-QS
```

When a reader section is preempted for the first time Context-Switch adds its thread to the relevant list of preempted readers copying A-RCU's approach. Once the preempted reader has been noted globally, the CPU may record a quiescent state because it is no longer running any readers. Lastly, Context-Switch notifies any Rcu-Synchronize waiting on line 14 via a semaphore that it had passed through a quiescent state.

Context-Switch

```
1  if 0 < thread.nesting-cnt and ¬thread.was-preempted
2      thread.was-preempted = true
3      Place in appropriate preempted reader list, i.e.
4          lines 3-7 of A-RCU's Context-Switch
5  Record-QS
6  if cpu.is-delaying-gp
7      cpu.is-delaying-gp = false
8      remaining-readers.Semaphore-Up
```

Assessment

Advantages:

- Reader sections may be preempted.
- Functions in interrupt and exception handlers.
- Clearly separated from the rest of the system; no need to instrument exception and interrupt handlers.
- RCU processing is contained in dedicated threads which simplifies keeping track of the cost of RCU when both detecting grace periods and running callbacks.

Disadvantages:

- Increases reader section compared overhead to Podzimek's RCU with one additional if statement and an Enable-Preemption and Disable-Preemption pair.

4.4 User space RCU

The implemented user space RCU in HelenOS is a variant of Signal URCU. The implemented algorithm is modified in three ways.

Firstly, it does not signal each thread of the process in order to issue a memory barrier in the thread on demand. Instead, we introduced a new syscall that issues memory barriers on CPUs that are running threads of the current process. As a result, the algorithm does not require the system to schedule and run every thread of the process just to detect a grace period. Moreover, the syscall is implemented in terms of interprocessor interrupts which are arguably less intrusive than switching contexts to the desired threads and entering user space.

Secondly, the algorithm is further modified to require a single instead of two reader group changes per grace period. Signal URCU flips the reader group and waits for readers of the previous group twice in succession in order to wait for new readers that were delayed and mistakenly associated with the previous reader group. If Signal URCU did not flip the reader group twice the following Rcu-Synchronize could find the now new reader group non-empty and it would incorrectly avoid waiting for readers in that group. The modified algorithm ensures that the new reader group is always empty by explicitly waiting for it to become empty first, which is an idea adopted from Sleepable RCU. Only then does it flip the reader group and wait for preexisting readers of the old reader group. Because the misassociation of readers is expected to be a rare event, waiting for the new reader group to become empty will almost always end immediatelly.

Thirdly, we introduced the ability for Rcu-Synchronize to exit early without starting a new grace period if a grace period already elapsed while the function was waiting to acquire the algorithm's global mutex in order to intiate a new grace period.

Pseudo code

HelenOS runs user mode code in threads scheduled in user space, so called *fibrils*, that execute on top of the kernel provided threads. Therefore, the implemented algorithm tracks individual fibrils instead of threads.

Read-side The algorithm's read-side is the same as in Signal URCU but we include it here again for completeness.

RCU-READ-LOCK

```
1  if fibril.nesting-cnt == 0
2      atomically assign packed in a single word:
3          fibril.nesting-cnt = 1
4          fibril.reader-group = cur-reader-group
5      COMPILER-BARRIER    // L
6  else
7      fibril.nesting-cnt++
```

RCU-READ-UNLOCK

```
1  COMPILER-BARRIER        // U
2  fibril.nesting-cnt--
```

Write-side Simultaneous invocations of RCU-SYNCHRONIZE are synchronized with a global mutex. The function first checks on line 6 whether a complete grace period elapsed while it was waiting to acquire the mutex. If not it increments the current grace period number, *cur-gp*, and starts a new grace period.

RCU-SYNCHRONIZE

```
 1  // Contain following load of cur-gp outside of mutex
 2  MEMORY-BARRIER              // E
 3  gp-in-progress = cur-gp
 4  with mutex locked
 5      completed-gp = cur-gp
 6      if completed-gp > gp-in-progress
 7          return
 8      cur-gp++
 9      MEMORY-BARRIER          // S
10      // Make the most current nesting-cnt visible to this CPU.
11      FORCE-MB-IN-THREADS     // F-L
12      MEMORY-BARRIER          // A-F-L
13      delayed-reader-grp = ¬cur-reader-group
14      WAIT-FOR-READERS(delayed-reader-grp)
15      MEMORY-BARRIER
16      cur-reader-group = ¬cur-reader-group
17      MEMORY-BARRIER
18      preex-reader-grp = ¬cur-reader-group
19      WAIT-FOR-READERS(preex-reader-grp)
20      FORCE-MB-IN-THREADS     // F-U
```

Following Signal URCU's example, FORCE-MB-IN-THREADS on line 11 forces memory barriers in other threads of the process. On the one hand, if the forced memory barriers are issued inside a reader section, i.e. between compiler barriers L and U, they pair up with A-F-L and make the positive nesting count visible to RCU-SYNCHRONIZE. On the other hand, if the barriers are issued outside of a reader section, F-L pairs up with S and makes changes prior to RCU-SYNCHRONIZE visible in future readers of such a thread and, therefore, also its fibrils.

Next, on line 13 the function waits for those readers to complete which have been mistakenly associated with the non-current reader group as discussed in section 3.2 and section 3.6. Only then can RCU-SYNCHRONIZE flip the current reader group to separate preexisting from new readers and proceed to wait for the preexisting readers with WAIT-FOR-READERS on line 18.

The function separates the change of the current reader group from WAIT-FOR-READERS with memory barriers in order not to intermingle the write of *cur-reader-group* with waiting for readers. However, the memory barriers surrounding the group update do not guarantee that new readers will immediately see the update. Until the readers encounter a memory barrier or a context switch it is up to cache coherence protocols to gradually make the updated *cur-reader-group* visible on all CPUs. As a result, new readers may still associate with the preexisting readers group even after the group flip. The situation is, however, benign. In the worst case WAIT-FOR-READERS will also wait for some of the new readers to complete.

Finally, RCU-SYNCHRONIZE contains any late memory references of preexisting readers' critical sections with line 20. Then, it unlocks its mutex and the grace period ends. Another RCU-SYNCHRONIZE that has waited for the global mutex on line 4 during this grace period can now start.

When a grace period ends, another RCU-SYNCHRONIZE blocked waiting on line 4 acquires the global mutex. Right after aquiring the mutex *cur-gp* represents the number of the most recently completed grace period, *completed-gp*. If that number is greater than *gp-in-progress*, i.e. the number of the grace period that was in progress when this RCU-SYNCHRONIZE started waiting for the global mutex, a full grace period must have elapsed while waiting for the mutex. In turn, the function exits early without initiating another grace period.

Although *gp-in-progress* reads *cur-gp* without any locking memory barrier E guarantees that exiting early is safe. In other words, RCU-SYNCHRONIZE exits via line 7 only if an entire grace period elapsed since we started waiting for the mutex and all prior changes propagated properly. Consider two executions of RCU-SYNCHRONIZE, A and B, possibly running in parallel. A reaches line 4 first

and successfully acquires the mutex whereas B gets blocked. If B is invoked when A's grace period is already in progress B's memory barrier E happens after the most recent S issued by A. Therefore, *gp-in-progress* reflects the most recently incremented *cur-gp*. When B locks the mutex, *completed-gp* will be equal to *gp-in-progress* and it initiates another grace period instead of exiting early.

However, if A's barrier S happens after B's E, E will pair up with A's F-L and propagate B's changes to all the new readers of the grace period that A has just initiated. As a result, B does not have to wait for another full grace period and it may exit immediately when A's grace period ends. In this case, B's *gp-in-progress* most likely does not include A's increment of *cur-gp*. When B finally acquires the mutex it will determine that *gp-in-progress* is less than *cur-gp* and exit early as desired.

```
FORCE-MB-IN-THREADS
1  Issue memory barriers on CPUs with threads of this process via a syscall.

WAIT-FOR-READERS(preex-reader-grp)
1  for each fibril fib
2       while fib.IS-PREEXISTING-READER(preex-reader-group)
3            SLEEP(10 ms)

IS-PREEXISTING-READER(preex-reader-grp)
1  return 0 < fibril.nesting-cnt
2       and fibril.reader-group == preex-reader-grp
```

Assessment

Advantages:

- Light-weight read-side critical sections.

Disadvantages might include:

- Required the introduction of a new syscall.

Chapter 5

Use of RCU in HelenOS

We demonstrate the use of RCU in two different roles – first to improve scalability, second as a waiting mechanism.

First, we introduce a novel concurrent hash table based on RCU. We incorporate the hash table in the kernel futex subsystem to increase its scalability.

Next, we illustrate how to wait for the completion of multiple tasks in a lightweight manner with RCU. We apply the technique to user space futex locking and decrease the locking's impact on performance in singlethreaded applications.

5.1 Resizable concurrent hash table

Instead of applying RCU to a single existing HelenOS subsystem we designed a high performance RCU concurrent hash table that may be employed throughout the system. What is more, the microkernel already makes use of a simple non-resizable hash table that is accessed concurrently e.g. in the futex subsytem or as a global page hash table and is, therefore, a canditate to be replaced with a more sophisticated design.

In order for the new concurrent hash table to be useful even in existing contexts, it must meet a number of requirements:

- It must be resizable and grow and shrink with the number of elements in the table.
- It must tolerate interrupts and non-maskable interrupts, e.g. for the hash table to potentially serve as a global page hash table.
- It should allow concurrent reads ideally with low overhead.
- Preferably, it should allow concurrent inserts and deletes.

5.1.1 Existing concurrent hash tables

The implemented RCU concurrent hash table, or CHT for short, combines original protocols with techniques and algorithms from previous hash tables. The relevant existing hash tables are described in short for comparison and reference.

Java ConcurrentHashMap

Lea's Java ConcurrentHashMap [11] is a variation on stripped locking [6]. The table is partitioned into a small number of independent segments of linked lists. Each segment is protected by a lock that prevents concurrent updates or resizing of the segment. Readers never block but traversing a collision chain requires at least one memory barrier per node. Furthermore, resizing a segment involves copying certain nodes of the linked lists and relies on garbage collection to ensure readers always access valid nodes in a bucket list.

Advantages:

- Readers never block and search the table in parallel
- Resizable

Disadvantages:

- Inserts and deletes into the same segment block
- Resizing blocks all updates of a segment until completed
- Reader overhead of one memory barrier per node of a bucket list
- Relies heavily on garbage collection
- Only grows, never shrinks

Michael's hash table

Michael introduced the first practical lock-free[1] single linked list [26] in 2002. He then built a simple nonresizable hash table with lock-free bucket lists.

Elements in a lock-free list are kept in a fixed order, e.g. sorted by their hash. The essential idea of Michael's lists is that a node is first marked as deleted before it can be unlinked from the list. As a result, it is always safe to insert a new node after a node that is not marked as deleted or to unlink the successor of such a

[1] A data structure is lock-free if one of the many threads attempting to perform an operation is guaranteed to complete the operation in a finate number of steps. This is a stronger guarantee than that which is provided by locks. See section 3.7 of [6]

node. What is more, checking if a node is marked as deleted and changing its next pointer if it is not deleted can be performed atomically with a compare-and-swap operation and leads to a lock-free design.

Michael ensures that traversed nodes remain valid with hazaard pointers [27, 28]. Therefore, list traversals incur a cost of at least one memory barrier per node.

Advantages:

- Lock-free table lookups, inserts and deletes

Disadvantages:

- Nonresizable
- Reader overhead of at least one memory barrier per node of a bucket list due to hazard pointers

Click's hash table

Click proposes a completely lock-free resizable hash table [2, 3] in Java that resolves collisions with linear open addressing. Instead of searching and modifying a linked list upon a collision it probes successive cells of the table.

The core of Click's hash table is a lock-free protocol to move a word from one table to another in the presence of concurrent modifications of the word. To move a value V from A to B:

1. B has to be initialized to an invalid value $V_{invalid}$ before the protocol can be applied.
2. A is atomically marked as immutable[2] if it still contains the value V with a Compare-And-Swap(A, V, $V \cup Immutable$). Any delayed concurrent updaters will detect that the word is being moved with a failing Compare-And-Swap(A, V, V_{new})
3. Next, we can save V in B by means of Compare-And-Swap(B, $V_{invalid}$, V) which guarantees that we do not accidentally overwrite B's content in case another thread completed the move in the meantime and has already updated B to a new value.

[2] The flag must be part of the word containing the moved value. As a result, the domain of values the word may contain must be smaller than number of values the word can actually store. For example, if the word stores a pointer the flag may be saved in the least significant bit as it is always zero for memory allocators available in practice.

4. Finally, we signal that the word has been moved successfully with a COMPARE-AND-SWAP(A, $V \cup Immutable$, $V_{invalid} \cup Immutable$).

Once the protocol starts the value in A will remain to differ from its original value V. Therefore, an updater can always detect that a word it wishes to modify is being moved. Moreover any thread can easily complete moving of a word on behalf of others at any step of the algorithm.

Advantages:

- Resizable
- Lock-free lookups, inserts, removals and even resize

Disadvantages:

- A cost of at least one memory barrier per visited node
- Linear probing tends to build longer strides of taken cells
- Relies heavily on garbage collection
- Only grows, never shrinks

Split-ordered hash table

Split ordered hash table [31] stores all elements in a single lock-free Michael's list. Nodes of the list are sorted so that elements belonging to the same bucket are grouped together. Therefore, buckets are represented as groups of elements in the single list and table pointers serve to quickly locate the first element of a bucket in the list.

What is more, nodes are sorted in a way that guarantees elements do not have to be moved when the table grows and each bucket has to be split into two. In particular, the elements are sorted by the reverse of their hash and the bottom k bits of the hash indicate the bucket where the element resides in a table with size 2^k. As a result of the chosen ordering of nodes in the list, when a bucket needs to be split elements of the two new buckets are already stored in separate groups in the list Moreover, nodes of the new bucket with the smaller ordinal number immediately precede the nodes of the other new bucket. Therefore, when the table grows it does not have to move any elements and it just adds an entry to the table pointing to the first node of the other new bucket.

Advantages:

- Lock-free lookups, inserts, deletes and resize
- Incremental gradual resize

Disadvantages:

- Overhead of multiple memory barriers in addition to at least one memory barrier per traversed node
- Space overhead of one dummy node at the beginning of each bucket that can never be freed
- Extra indirection when accessing elements due to a 2-level main table and dummy nodes
- Only grows, never shrinks

Relativistic hash table

Triplett et al. built a resizable hash table based on RCU [33] which handles hash collisions with a separate RCU protected list per bucket. RCU reader sections enclose list traversals and updaters publish list modifications via RCU-SYNCHRONIZE. What is more, RCU-SYNCHRONIZE separates bucket node removals from the nodes' destruction. Consequently, RCU defers freeing of nodes until all list traversals involving the node have completed and noone can access the node. As a result, table lookups are fast and are guaranteed to access only valid nodes without having to resort to locking, explicit tracking of valid nodes with e.g. hazard pointers, or issuing memory barriers.

Furthermore, the hashing scheme ensures that when the table grows elements of a bucket in the old table are divided between exactly two buckets of the new larger table. Both of the buckets of the new larger table initially point to the same original bucket containing elements of both new buckets. Next, a background resizing thread adjusts the next pointer of the first element of the first new bucket to skip any following elements of the other new bucket and point directly to the second element of the first new bucket. The process is repeated one node of the original bucket at a time until the bucket is unzipped into two.

Advantages:

- Near zero overhead of lookups even when resizing because readers never synchronize and need not issue memory barriers
- Resizable, grows and shrinks

Disadvantages:

- Inserts and deletes synchronize by means of locks; therefore, block
- All updates are blocked until a resize completes
- Resize requires a number of grace periods to elapse which results in a longer time to completely resize the table

5.1.2 Implemented concurrent hash table

The implemented concurrent hash table in HelenOS, CHT, has a structure similar to Triplett's relativistic hash table. CHT also resolves hash collisions with separate RCU protected bucket lists. Moreover, CHT too defers freeing list nodes by means of RCU, namely Rcu-Call. Therefore, walking the lists in RCU reader sections guarantees that any nodes reachable by following the next pointers are valid.

Quite unlike Triplett's hash table, CHT organizes its bucket lists as Michael's lock-free lists. What is more, CHT's lock-free lists do not require hazard pointers because RCU already ensures nodes remain valid for as long as they might be accessed[3]. Therefore, CHT's lock-free lists do not incur the cost of issuing at least one memory barrier per visited node.

The combination of RCU with lock-free buckets results in near zero overhead concurrent lookups all the while allowing concurrent modifications of the table. In addition, the table tolerates nested concurrent modifications from interrupt and exception handlers due to the lock-free property of its buckets.

Resizing algorithm In short the table resizes in three main steps. Firstly, it moves pointers to the first nodes of buckets, or simply bucket heads, to a new bucket head array. Next, CHT splits or joins the buckets of the new array so that the table can make use of the extra bucket head slots in the new array. Lastly, the new array replaces the original bucket array.

CHT grows or shrinks by a factor of two. Because it assigns buckets to elements based on the top k bits of their hash, when the table grows each bucket is split into two buckets and all elements of the two new buckets come from a single bucket in the original table. Furthermore, bucket nodes are sorted by their elements' hashes much like nodes of a split ordered hash table. While elements at

[3] RCU also prevents the ABA problem (section 10.6 of [6]) when an atomic compare and swap of a pointer fails to notice a change of the pointer and mistakenly succeeds. Consider this scenario: the pointer is first updated from A to B; next, the object at A is freed; finally a new object at A is constructed and the pointer is set to point to A again although it represents a different object.

the beginning of the original bucket fall into the first new bucket the remaining elements will end up in the second new bucket. Therefore, there is a single link where to split the original linked list into two correctly sorted linked lists of the new buckets.

With a single split point CHT can resize the table in parallel with lookups and lock-free updates also in a lock-free manner. A single background thread guides resizing of the table. However, if an updater detects that the bucket it is about to access is resizing, i.e. its head is moving or it is waiting to be split or joined, the updater steps in and completes the head move or bucket split on behalf of the resizing thread.

In particular, the background resizing thread increases the size of the table as follows:

1. First, the resizer allocates a new bucket head array that is twice the size of the original one. Next, it marks the bucket heads as invalid as Click's lock-free move protocol requires. Then, it waits for a grace period to elapse to ensure all updaters see the array properly initialized.

2. Second, the resizer moves the bucket heads from the original array to the new array with Click's lock-free protocol. The original bucket heads remain marked as immutable. In addition, owing to the selected ordering of list nodes elements at the beginning of the moved list are exactly the elements that should be stored at the new bucket head. The rest of the list contains elements that still have to be linked to the new additional bucket head in the second half of the array, which remains marked as invalid.

3. Third, CHT splits the buckets in the new array. It attaches the first node that does not belong in the new bucket, F, to the correct new additional bucket head. Then, CHT severs the link between F and the node immediately preceeding it, L, to complete splitting of the bucket. L is also the last node that should be stored at the new bucket head. In greater detail:

   (a) The resizer marks L as *join follows* similarly to how Michael marks nodes of lock-free lists as deleted. Marking a node as join follows ensures that updaters unaware that the table is resizing do not succeed in inserting new nodes between L and F. What is more, it guarantees that such updaters unlink neither L nor F.

   (b) Next, the table marks F as *join node*.

   (c) Then, CHT points the correct bucket head, which is still marked invalid, to F. Because L also links to F, F is now part of two buckets

and cannot be unlinked atomically from both lists. However, F is a join node which indicates the node may only be marked deleted but must not be unlinked.

(d) The resizer waits one grace period for all updaters to see that all new bucket heads are valid. Therefore, future updaters will not retry steps (a)-(c) in order to split a bucket on behalf of the resizer. Consequently, the resizer can safely unlink F from L and also atomically remove L's mark join follows.

(e) Another grace period makes it visible that the link between L and F has been broken. As a result, F is no longer part of two lists and its join node label is removed.

4. Next, the resizer initiates another grace period to make all participants aware that the table is now free of any node follows or join node marks.

5. Last, the resizer replaces the old bucket head array with the new one and publishes the change with a RCU-SYNCHRONIZE. Finally, the old bucket head array may be freed.

The table shrinks with a similar protocol.

Assessment

Advantages:

- Near zero overhead of lookups even when resizing because readers never synchronize and need not issue memory barriers
- Resizable, grows and shrinks
- Inserts and deletes are lock-free even in the face of a resize
- No space overhead of additional dummy nodes or multi-level main table

Disadvantages:

- Resize requires four grace periods to elapse which results in a longer time to completely resize the table
- Algorithm complexity

Technical details The actual implementation extends the algorithmic description above with a number of practical improvements. Firstly, CHT handles insertions of multiple equivalent items. Secondly, the table memoizes hashes and terminates buckets with a single global sentinel node to further improve performance. Thirdly, user generated hashes are mixed to guard against hashes not suitable for a power of two table. Lastly, the table manages to encode six different marks of the next pointer into the four values available in the two lower order bits of the pointers, namely deleted, join follows, join node, invalid, immutable and normal.

5.1.3 Improving futex subsystem scalability

The original futex kernel subsystem employed a single hash table protected with a global passive mutex. The kernel made use of the table to look up a kernel object that is associated with the futex and is necessary to perform operations on the futex.

In order to improve the subsystem's scalability, we modified the implementation to instead use CHT. Therefore, the global passive mutex no longer represents a performance bottleneck when operating on distinct futexes.

5.2 Upgradable user space futexes

5.2.1 Futexes in singlethreaded programs

HelenOS runs user space code in fibrils which are threads implemented entirely in user space. The user space library, libc, schedules fibrils cooperatively on top of ordinary kernel-level threads. For example, whenever a program calls `printf()` the fibril manager in libc may schedule a different fibril to run in the current kernel-level thread.

While HelenOS[4] supports concurrent execution in user space with both fibrils and threads the existing user space servers and programs are predominantly singlethreaded. Only two[5] user space drivers add new threads via `thread_create()` and only a limited number[6] of user space programs create additional fibrils explicitly via `fibril_create()`.

However fibrils are allowed to execute on top of multiple threads. As a result, fibril synchronization as well as entering the fibril manager involve futexes to

[4] As of version 0.5.0; in particular revision r1723

[5] Keyboard drivers for Ski and Niagara

[6] Applications kbd, nterm, trace, vuhid, klog, ping; drivers xtkbd, ps2mouse, usb stack; servers devman, remcons, isdv4_tablet, networking stack

ensure thread-safety in future multithreaded programs. Nonetheless, the use of futexes introduces unnecessary overhead to singlethreaded programs in terms of memory barriers and atomic instructions.

```
1  void fibril_add_ready(fid_t fid)
2  {
3      fibril_t *fibril = (fibril_t *) fid;
4      futex_lock(&fibril_futex);
5      if ((fibril->flags & FIBRIL_SERIALIZED))
6          list_append(&fibril->link, &serialized_list);
7      else
8          list_append(&fibril->link, &ready_list);
9      futex_unlock(&fibril_futex);
10 }
```

For example, the function `fibril_add_ready()` protects its internal structures with a futex as it notifies the fibril manager that a fibril is ready to be scheduled. The function does not enter the fibril manager with an acquired futex and, therefore, cannot be switched with another fibril. As a result, accessing the internal data is atomic in singlethreaded programs even without the use of the futex. What is more, the function unlocks the futex before it allows the fibril to enter the fibril manager and switch to a different fibril. As a consequence, any fibril in the singlethreaded program will always find the futex unlocked.

In general whenever futexes are employed with mutex semantics in singlethreaded programs the futexes may be removed. Given that all fibrils run in a single thread they must always find each futex unlocked. Otherwise the fibril attempting to acquire a locked futex would block the single thread. Because the program would have no more runnable threads it would not be able to schedule the fibril holding the futex to run and unlock the futex. Consequently, the entire application would come to a halt.

Ideally, only multithreaded programs would include futexes with mutex semantics and singlethreaded programs would utilize an empty futex implementation in order to minimize overhead.

5.2.2 RCU as a waiting mechanism in libc

We exploited that if libc is deadlock free in singlethreaded programs it always finds futexes unlocked even when running with multiple fibrils. We modified libc to start programs with an empty futex implementation. Only once the first additional thread is created the futex implementation is switched dynamically to the thread-safe original version. However before the new thread can run all

critical sections protected with the empty futex implementation must first exit. RCU allows us to cheaply wait for all existing futex critical sections to exit and to instruct future critical sections to make use of the thread-safe futex version.

Locking and unlocking upgradable futexes The functions FUTEX-LOCK and FUTEX-UNLOCK illustrate how to choose between a thread-safe and an empty futex implementation in a light weight manner.

```
FUTEX-LOCK(futex)
1  RCU-READ-LOCK
2  futex.upgraded = RCU-ACCESS(use-thread-safe-futexes)
3  if futex.upgraded
4       FUTEX-DOWN(futex)
```

```
FUTEX-UNLOCK(futex)
1  if futex.upgraded
2       FUTEX-UP(futex)
3  RCU-READ-UNLOCK
```

The functions enclose the entire futex critical section in an RCU reader section. FUTEX-LOCK first reads the global RCU protected variable *use-thread-safe-futexes* to decide if it should fall back on the thread-safe version represented by the call to FUTEX-DOWN. Furthermore, it notes in *futex.upgraded* on line 2 how the futex was locked so that FUTEX-UNLOCK can do the inverse.

When a program starts *use-thread-safe-futexes* is set to false and locking and unlocking a futex is equivalent to entering and exiting an RCU reader section and evaluating two if statements. On the other hand, the reader section and the two branches represent an additional overhead introduced to multithreaded programs that previously invoked the thread-safe FUTEX-DOWN and FUTEX-UP directly.

Section 6.7 shows that the additional overhead results in a 15% performance penalty in multithreaded programs whereas the change speeds up futex locking 2.6 times in the more common singlethreaded case.

Waiting for locks to upgrade Before the thread entry function THREAD-ENTRY executes the user supplied thread function it waits for UPGRADE-ALL-FUTEXES-AND-WAIT to upgrade all futexes to a thread-safe version.

```
THREAD-ENTRY
1  // ... fibril and thread setup
2  UPGRADE-ALL-FUTEXES-AND-WAIT
3  // Run the desired thread function
```

```
UPGRADE-ALL-FUTEXES-AND-WAIT
1  FUTEX-DOWN(upgrade-and-wait-futex)
2  if ¬use-thread-safe-futexes
3      RCU-ASSIGN(use-thread-safe-futexes, true)
4      RCU-SYNCHRONIZE
5  FUTEX-UP(upgrade-and-wait-futex)
```

UPGRADE-ALL-FUTEXES-AND-WAIT first checks if futexes have already been upgraded on line 2. Next, it instructs FUTEX-LOCK to start using a thread-safe implementation on line 3. Due to futex critical sections being enclosed in RCU reader sections, RCU-SYNCHRONIZE waits for all existing critical sections that may have been using an empty futex implementation to exit. Moreover, line 4 publishes the assignment to *use-thread-safe-futexes* on line 3 to future RCU readers and, therefore, ensures all future calls to FUTEX-LOCK see the assignment and switch to the thread-safe version.

What is more, the function protects *use-thread-safe-futexes* from simultaneous access from multiple threads that are starting up with an always thread-safe futex *upgrade-and-wait-futex*.

Technical details HelenOS libc exposes futex variables with semaphore semantics via `futex_down()` and `futex_up()`. We extended the futex libc interface with `futex_lock()` and `futex_unlock()`. These functions make it explicit that mutex semantics apply to a given futex. In particular, the futex must be unlocked in the same fibril where it was locked. In addition, a `futex_unlock()` may never precede a `futex_lock()`.

Whereas `futex_lock()` and `futex_unlock()` implement upgradable futexes `futex_down()` and `futex_up()` retained their original properties.

The following futexes in libc were converted to upgradable futexes:

- `fibril_futex` in uspace/lib/c/generic/fibril.c
- `ipc_futex` in uspace/lib/c/generic/ipc.c
- `malloc_futex` uspace/lib/c/generic/malloc.c

The futexes listed below, however, could not be converted because they unlock the futex from a different fibril:

- `async_futex` in uspace/lib/c/generic/async.c

Chapter 6

Evaluation

The proposed RCU algorithms as well as their use were evaluated experimentally. Section 6.1 details the methodology of running the experiments and the hardware setup used. First, section 6.2 focuses on the scalability of the each of the implemented RCU algorithms' read-side. Second, section 6.3 examines RCU's write-side overhead and compares it with the overhead introduced by a spinlock. Third, section 6.4 and section 6.5 compare the performance of our novel RCU based concurrent hash table to a traditional scalable lock based hash table. Next, section 6.6 evaluates the scalability of the original and the RCU improved futex kernel subsystem Last, section 6.7 contrasts the performance of upgradable futexes with plain thread-safe futexes in a singlethreaded and multithreaded setting.

6.1 Methodology

The algorithms were benchmarked on a computer with an Intel Core i7 920 2.67 GHz processor that includes four physical cores each with two hyper-threading logical cores.

All benchmarks were run for at least ten seconds. Moreover each benchmark was repeated twenty times after five initial warm-up runs. Unless noted otherwise, the graphs that follow include error bars at a distance of one standard deviation from the mean[1]. However, the variance of the measured runs was negligible and, therefore, the error bars are not visible in the graphs with the naked eye.

What is more, the system was rebooted as rarely as possible in order to mitigate the effects of slightly different clock tick calibration. In fact the system was only rebooted if the next series of benchmarks required recompilation of the source codes.

[1]The standard deviation of the underlying distribution (not the deviation of the mean) was estimated with an unbiased sample standard deviation.

In order to measure RCU's read-side scalability and write-side's overhead we simulated situations where operations are short and frequent but still require synchronization, e.g. searching a routing table. In such settings synchronization contributes noticeably to the operations running time. We chose protecting traversals of a five element list as a representative workload. While walking the entire list is quick it is already an example of a real world workload which is similar to searching a rather long hash table bucket. Furthermore, if we were to measure performance with a long critical section the length of the critical section alone would dominate the cost of the operation independent of the used synchronization mechanism and hide the performance characteristics in question.

Moreover we compared RCU to spinlocks and not reader-writer locks because for shorter critical sections spinlocks outperform reader-writer locks, see [4, 13] and section 2.2.8 of [12]. Reader-writer locks are best suited for longer critical sections where the primary cost of synchronization is due to contention and not cache misses on the lock variables. In addition, HelenOS does not provide a performant implementation of reader-writer locks.

Next, although HelenOS offers passive mutexes we opted for spinlocks. The implementation of spinlocks consistently outperformed passive mutexes by a large margin in every test we tried.

Lastly, all benchmarks but one were limited to a maximum of four cores to avoid skewing results with hyper-threading.

6.2 Read-side scalability

We used a five element linked list to benchmark the scalability of RCU read-side critical sections. The same list was traversed in a tight loop by up to four CPU bound threads. Although the list was not modified during the benchmark, each list traversal was protected in a reader section or a spinlock for comparison.

Figure 6.1 shows that both A-RCU and the preemptible version of Podzimek's RCU scale linearly. While both algorithm implementations demonstrate optimal linear scaling to more processors A-RCU fares better than PP-RCU. A-RCU achieves 74% and PP-RCU 37% of the ideal number of list traversals, i.e. when no synchronization is used whatsoever.

What is more, A-RCU introduces significantly lower overhead compared to an uncontended spinlock. Even when running in a single thread, A-RCU allows 2.6 times more operations to complete that an uncontended spinlock.

Finally, enclosing list traversals in a spinlock protected critical section displays negative scaling. As expected, the loss of parallelism and the extensive interprocessor communication severely degrade performance and the more pro-

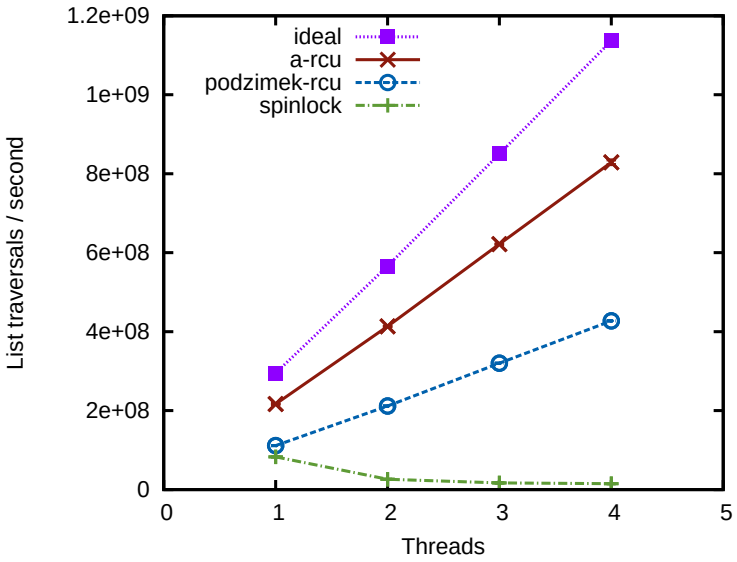


Figure 6.1: The graph shows the scalability of RCU read-side critical sections. y-axis indicates the total number of traversals of a five element list with respect to the number of threads walking the same list in parallel. Nothing, A-RCU, PP-RCU and a spinlock protect each traversal respectively although the list is only read and never modified. *ideal* represents walking the list without any synchronization whatsoever.

cessors were involved in the benchmark the less total number of operations was completed in a second.

6.3 Write-side overhead

RCU provides facilities to efficiently publish new versions of RCU protected data but it does not deal with coordination of updaters themselves. Updaters typically employ traditional locking to avoid data structure corruption from concurrent modifications. In order to measure the overhead introduced by RCU's grace period detection on top of the synchronization between updaters, we compared the performance of a spinlock protected list with a list where only updates were synchronized with a spinlock and reading the list involved only an RCU reader section.

Again we used a five element list. Updates of the list consisted of replacing the first element with a new one. Each thread had its own list of preallocated free elements; therefore, allocating a new element did not require any locking and did not involve `malloc()`.

Figure 6.2 depicts the total number of completed operations in four threads as the fraction of updates on the total number of operations varies. For read mostly data RCU's light weight and scalable reader sections clearly outweigh

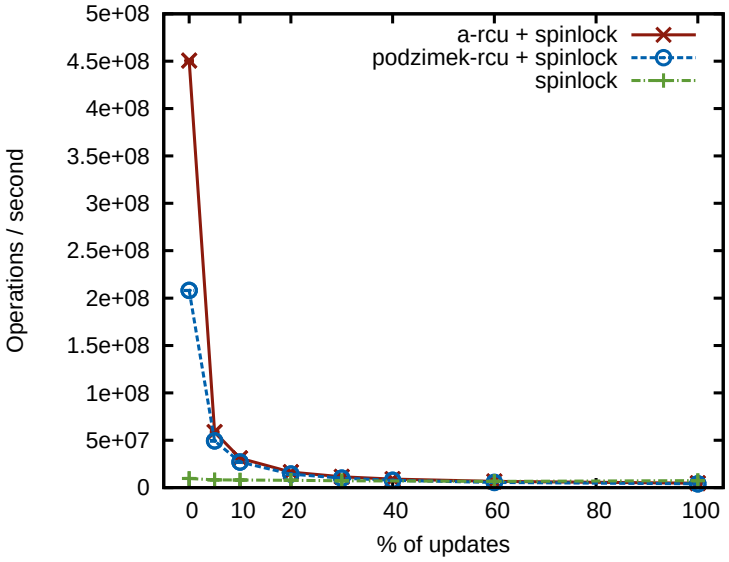


Figure 6.2: The figure illustrates RCU write-side overhead with the total number of operations performed on a five element list from four threads running in parallel with respect to the fraction of updates. *a-rcu + spinlock* and *podzimek-rcu + spinlock* guard walking the list with A-RCU and PP-RCU respectively but coordinate updates with a single spinlock. *spinlock* test runs access the list with an acquired spinlock when both reading and updating the list.

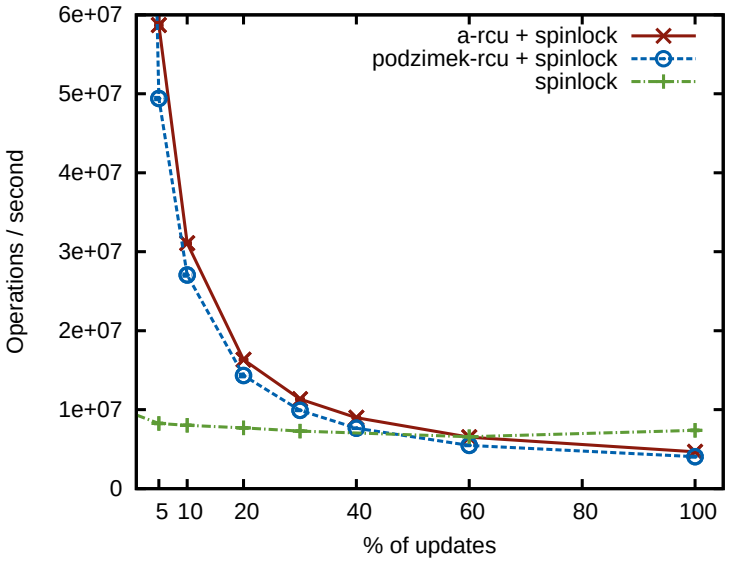


Figure 6.3: This figure details the point when the speed of RCU read-sides no longer outweighs RCU write-side overhead. It discards data points of figure 6.2 with low fraction of updates in order to make the crossover point visible.

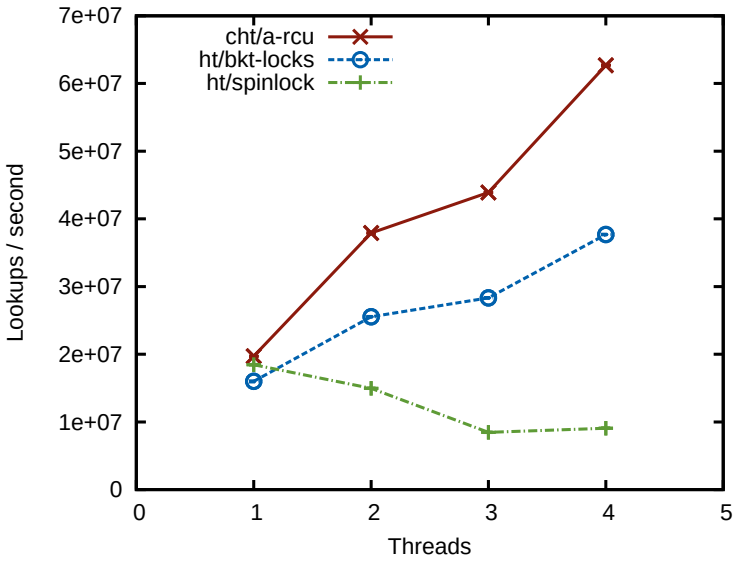


Figure 6.4: The figure illustrates hash table lookup scalability of CHT run in conjunction with A-RCU, *cht/a-rch*, a hash table with a dedicated spinlock per each bucket, *ht/bkt-locks*, and a hash table guarded by a single spinlock, *ht/spinlock*.

the overhead of grace period detection. As the fraction of updates increases performance quickly degrades.

Figure 6.3 shows that eventually the overhead incurred by grace period detection is greater than the benefits of fast read-side sections. A-RCU again fares better that PP-RCU and the crossover point for A-RCU and PP-RCU is 60% and 40% of updates respectively.

6.4 Hash table lookup scalability

In this section we compare lookup scalability of our novel concurrent hash table with ordinary spinlock guarded hash tables. The test consisted of searching for keys in tables with an average load factor of four elements per bucket. Half of the lookups were searching for keys present in the table; the other half was for nonexistent keys. In addition, each thread searched for a separate group of keys. Therefore, any contention was purely the result of protecting the tables' internal structure in anticipation of updates and was not inherent in the lookup pattern. Consequently, in the ideal case the hash tables should have introduced no contention.

In particular, figure 6.4 compares:

cht/a-rcu CHT that uses 128 buckets and A-RCU. To index a bucket CHT first mixes the bits of the user supplied hash and then takes the most significant 7 bits.

ht/bkt-locks A non-resizible hash table with 127 buckets which are each guarded by a distinct spinlock. To index a bucket the table calculates the remainder of the user supplied hash after division by the number of buckets.

ht/spinlock A hash table similar to *ht/btk-locks* that however guards the entire table with a single spinlock.

Note that CHT was set up with a resize triggering load factor high enough for CHT to never resize. However, CHT was still tracking the number of elements in the table and still checking if it should resize – quite unlike the spinlock protected tables that are not resizable and, therefore, do not need to track the number of elements nor check if a resize is in order.

What is more, the hash tables use different hash functions. While CHT's design forces it to mix user supplied hashes to produce a good hash, spinlock based hash tables divide user hashes by a prime number and use the remainder as the final bucket index. This influences both the distribution of elements in the buckets as well as the time to actually compute a hash. Keys were selected to favor traditional hash tables which achieved an optimal element distribution, i.e. exactly four element in each bucket. On the other hand, in CHT buckets contained from 0 to 9 items. Furthermore, the hash mixing function employed by CHT was approximately 10% slower than diving by a prime.

Figure 6.4 contrasts the reading scalability of the hash tables. As expected, the naive approach to guarding a table with a single lock leads to negative scaling. On the other hand, both CHT and the hash table with per bucket spinlocks display near linear scaling. Nonetheless, CHT outperforms the spinlock based table by 66% when accessing the table concurrently from four processors thanks to fast RCU protected read-side code paths. Moreover CHT performs slightly better than both spinlock based hash tables in the base case of running on a single processor core.

The results show a larger performance increase in all hash tables tested when adding a second processor to the benchmark than when adding a third processor. We can only speculate about the source of the anomaly, especially because each processor core has a dedicated L2 cache and can only share data via a single L3 cache.

6.5 Hash table update overhead

In order to evaluate the behavior of CHT in presence of concurrent updates we let four CPU bound threads update the table with a varying probability and compared the results with hash tables with per bucket spinlocks or a single table

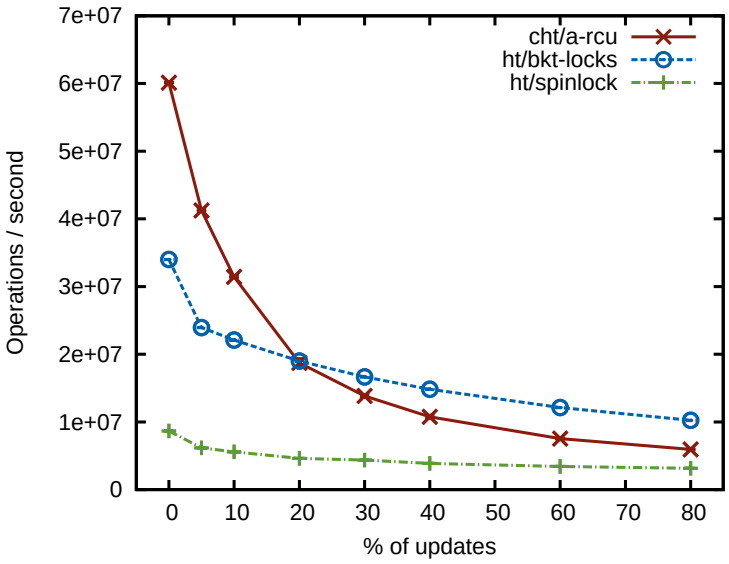


Figure 6.5: The graph captures the gradual decline in performance of hash tables with the increase in the fraction of concurrent updates. Four threads were accessing the same table in parallel and the y-axis represents the total number of operations completed. *cht/a-rcu* used CHT with A-RCU. *ht/bkt-locks* denotes a hash table with a spinlock per each bucket and *ht/spinlock* is a hash table protected with a single spinlock.

spinlock. When updating the table a thread selects a key from a universe of keys that is twice as large as the desired number of elements in the table. If the key already resides in the table the associated element is removed; otherwise a new element with such a key is inserted. Threads keep track of inserted keys in thread local bitmaps in order to quickly determine if a new key should be inserted or an existing removed.

Having a universe of keys with twice the desired number of keys in the table produces variantions in the number of elements in the table but leads to the targetted average number of keys in the table throughout the run of the benchmark. If more keys are stored in the table removals become more probable. Similarly, as the element count drops below the targetted average inserts become more prevalent.

The test setup si similar to the ones used before. Firstly, the benchmark starts with a table with the desired average load. Secondly, each thread utilizes a separate space of the universe of keys. Lastly, unlike in previous benchmarks elements are not preallocated in order to model the effects of increased memory consumption due to delays in memory reclamation by RCU.

Figure 6.5 shows that CHT significantly outperforms hash tables with per bucket spinlocks in read mostly scenarios where the fraction of updates does not exceed 10%. RCU grace period detection overhead as well as the complexity

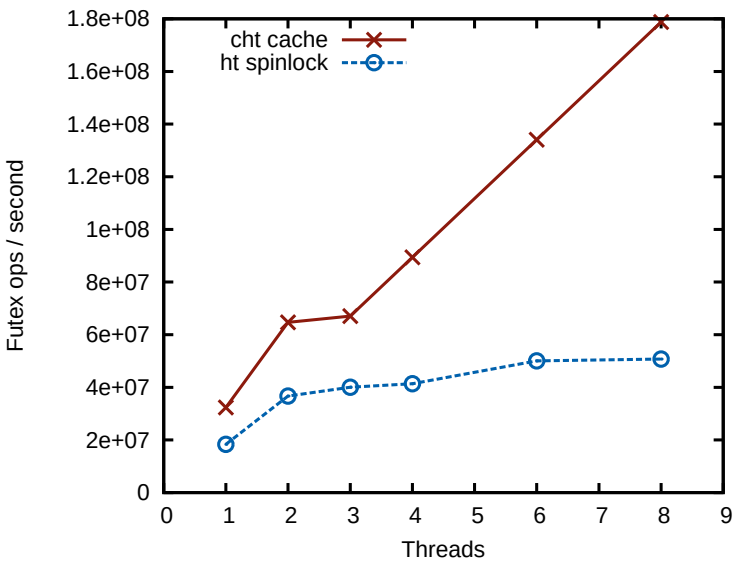


Figure 6.6: The figure captures the scalability of the kernel futex subsystem. The graph portaits the total number of futex syscalls completed in a tight loop depending on the number of user space threads in use. *cht cache* is the new futex subsystem that incorporates CHT while *ht spinlock* denotes the modified version of the original futex subsystem that had passive mutexes replaced with spinlocks.

of CHT's updates gradually lower its performance. At 20% updates CHT is no different that a hash table with per bucket spinlocks.

6.6 Futex kernel subsystem performance

We also evaluated the scalability and performance of the futex kernel subsystem based on CHT and compared it with the previous design. We varied the number of user space threads that each invoked the futex syscalls in a tight loop. Each of the threads operated on a different futex variable to avoid contention of the benchmarking threads.

The original design employed passive mutexes to exclude concurrent accesses of a single system global hash table of kernel futex objects. Unfortunately, the subsystem performed abysmally in our benchmark and the number of syscalls completed per second copied the x-axis. Consequently, we benchmarked a modified version of the original design where we replaced the passive mutexes with spinlocks.

Figure 6.6 clearly shows that the previous locking scheme employed in the futex kernel subsystem limited its scalability and performance. After adding two processor cores the performance improves only marginally with additional processors. For example, utilizing six more cores to a total of eight cores, i.e. an increase of 300% in computing power, increases the total number of futex syscalls

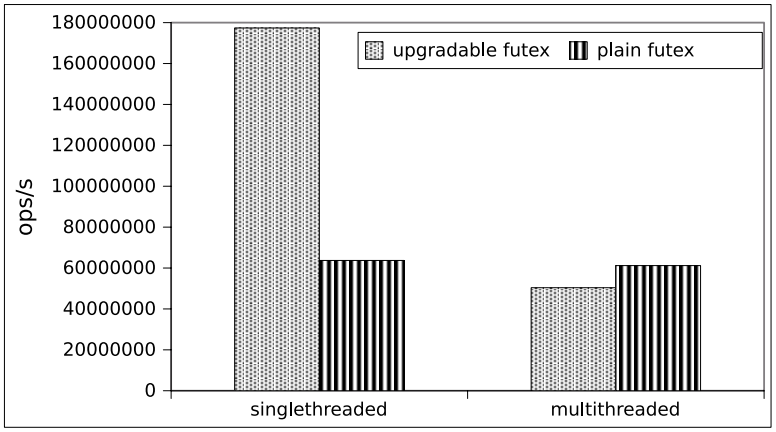


Figure 6.7: This figure compares the performance of a light weight upgradable futex with that of an ordinary futex in both a singlethreaded setting as well as when used from two threads. The y-axis is the number of futex operations executed in a single thread per second.

by only 82% over two processor cores.

On the other hand, not only does the CHT based futex backend demonstrate linear scalability, it also improves single threaded performance of the subsystem.

Furthermore, the figure also includes a sudden performance bump when going from one to two processors, which was already mentioned in section 6.4.

6.7 Libc futex performance

In order to gauge the performance benefits of light weight upgradable futexes, one and two user space threads locked and unlocked distinct futexes in a loop. Next, the experiment was repeated with ordinary futexes that always use atomic instructions.

The number of operations completed per thread is depicted in figure 6.7. Upgradable futexes offer 2.6 times the performance of ordinary futexes in the singlethreaded case. In multithreaded programs light weight futexes are upgraded to use atomic instruction just like ordinary futexes but have to check if a futex was upgraded or not. The incurred additional overhead over ordinary futexes is however only 15%.

6.8 Discussion of results

The benchmark results of this chapter clearly demonstrate that RCU was successfully used to significantly improve scalability in HelenOS (section 6.6). What is more, it markedly speeds up fibril locking in HelenOS's singlethreaded user space programs (section 6.7).

Moreover, section 6.2 and section 6.3 establish that the implemented RCU algorithms exhibit the expected performance characteristics consistent with existing RCU algorithms [12].

Lastly, section 6.4 and section 6.5 prove that RCU allowed us to build a concurrent hash table that dramatically outperforms the best lock based hash table designs in read mostly situations.

Chapter 7

Summary

7.1 Conclusion

The primary goal of this thesis was to design and implement an RCU algorithm suitable for the microkernel operating system HelenOS. Secondly, the thesis aimed to explore some of the possible uses of RCU within HelenOS. Last but not least, the thesis was to give an overview of the relevant RCU variants.

Firstly, we briefly described the semantics of RCU along with an example usage.

Secondly, the thesis gave an extensive review of relevant RCU algorithms. Reviews included pseudocode of the algorithms that exemplified their main ideas. What is more, we highlighted the strengths and weaknesses of the individual RCU variants.

Thirdly, we focused on designing RCU algorithms suitable for HelenOS and implemented three RCU algorithms. One of the kernel algorithms, A-RCU, builds on the ideas of previously existing user space RCU and exploits the kernel environment to provide a novel algorithm. Furthermore, the thesis extends the non-preemptible Podzimek's kernel RCU into a fully preemptible RCU with the techniques developed for A-RCU. What is more, we combined the existing user space Signal URCU algorithm with an idea introduced by the kernel Sleepable RCU to shorten grace period detection duration in the common case and implemented the resulting user space algorithm.

Fourthly, the thesis examined the potential uses of RCU in HelenOS and employed RCU both to increase scalability of the kernel as well as a cheap waiting mechanism in user space. In addition, it mentions that RCU allowed us to introduce a novel resizable concurrent lock-free hash table, CHT. CHT successfully replaced the original kernel futex subsystem design which resulted in dramatic

scalability improvements and even reduced the subsystem's singlethreaded base cost. Moreover we note that HelenOS's user space programs currently do not utilize multithreading. As a result, they do not represent ideal candidates for RCU to increase their scalability. We exploited this fact to speed up fibril locking with the help of user space RCU, this time in the role of a light weight waiting mechanism.

Next, the thesis includes a thorough experimental evaluation of the implemented work. First, we analyzed RCU's read-side scalability and write-side overhead and determined that A-RCU offers the best performance. Second, we compared the speed of lookups and updates of CHT with a lock based hash table that is the least prone to contention – a table that includes one spinlock per each bucket. We concluded that CHT significantly outperforms the lock based hash table in read mostly situations, as is expected of an RCU protected data structure. Next, we explored the scalability of the new CHT based kernel futex subsystem. The presented results demonstrate that the new futex subsystem is linearly scalable which is a substantial improvement over the previous design. Furthermore, the new futex subsystem proved to be faster in the singlethreaded uncontended base case. Last but not least, we determined that RCU allowed us to considerably speed up fibril locking of singlethreaded user programs.

Lastly, we provided testing code of not only the RCU implementations and CHT but also of other new utility components.

In summary, the goals of the thesis were fulfilled. We provided high performance RCU prototype implementations and successfully employed and evaluated RCU within HelenOS.

7.2 Future work

There are multiple ways in which the current work may be extended.

Firstly, HelenOS could possibly benefit from other RCU variants, e.g. an RCU implementation that is specifically optimized for single processor machines like Linux's TinyRCU [21]; or an RCU variant that allows sleeping in the read-side critical section should such a need arise.

Secondly, the current implementation of kernel RCU algorithms uses a single lock to protect global RCU data. The single lock is involved in grace period processing and in turn limits write-side scalability to tens of processors. Should HelenOS support larger number of processors, current RCU's write-side scalability could be improved with hierarchical locking [19].

Thirdly, selected user space servers could be converted to a multithreaded

design in order to make use of RCU's properties.

Lastly, an interesting avenue for future research is to encapsulate the use of RCU in other concurrent data structures that are arguably easier and less error prone to use than intricate custom made RCU designs.

Appendix A

Getting started with HelenOS

In order to start HelenOS, first check out the source files (or use the pregenerated cd-rom images in images/):

```
$ bzr co lp:~adam-hraska+lp/helenos/cht-bench
```

Second, build the cross compiler necessary to build HelenOS itself:

```
$ cd cht-bench
$ ./tools/toolchain.sh ia32 amd64
```

Third, build the HelenOS sources (substitute amd64 for ia32 if you are running a 32 bit x86 machine):

```
$ make PROFILE=amd64
```

Now you can boot the generated image in e.g. qemu:

```
$ qemu-system-x84_64 -cdrom image.iso -smp 4
```

After a while the system boots and the user is presented with a graphical user interface. To enter the kernel console, type:

```
$ kcon
```

When in the kernel console, the user may display statistics gathered by the added components, i.e.

```
$ rcu
$ workq
```

What is more, the kernel console allows testing the individual components, e.g.run:

```
$ test smpcall1
$ test workqueue
$ test rcu1
$ test cht
```

Moreover, you can run benchmarks in the kernel console. See the following command's output to determine how to run a specific benchmark:

```
$ chtbench 0 0 0
```

Finally, you can exit the kernel console:

```
$ continue
```

When in the regular console, you can user space benchmarks. Consult the output of:

```
$ app/rcubench
```

Appendix B

Navigating the source tree

Kernel RCU The implementation of A-RCU and PP-RCU can be found in:

- kernel/generic/include/synch/rcu.h
- kernel/generic/include/synch/rcu_types.h
- kernel/generic/src/synch/rcu.c
- kernel/test/synch/rcu1.c

CHT The comments in CHT's source codes explain handling of various edge cases:

- kernel/generic/include/adt/cht.h
- kernel/generic/src/adt/cht.c
- kernel/test/cht/cht1.c

User space RCU For more information about URCU consult these source codes:

- uspace/include/lib/urcu/rcu.h
- uspace/lib/urcu/rcu.c
- kernel/generic/include/synch/smp_memory_barrier.h
- kernel/generic/src/synch/smp_memory_barrier.c

Upgradable futexes In order to view the implementation of upgradable futexes navigate to these files:

- uspace/lib/c/include/futex.h
- uspace/lib/c/generic/futex.c
- uspace/lib/c/generic/thread.c

Work queues To support CHT we extended HelenOS with work queues, which are automatically expanding thread pools that support insertions of new work items without blocking.

- kernel/generic/include/synch/workq.h
- kernel/generic/src/synch/workq.c
- kernel/test/synch/workq-test-core.h
- kernel/test/synch/workqueue2.c
- kernel/test/synch/workqueue3.c

SMP calls Instead of dealing with interprocessor interrupts directly, we encapsulated the functionality with SMP calls. The facility allows invoking functions on specific CPUs via IPIs.

- kernel/generic/include/smp/smp_call.h
- kernel/generic/src/smp/smp_call.c
- kernel/generic/include/cpu/cpu_mask.h
- kernel/generic/src/cpu/cpu_mask.c
- kernel/test/smpcall/smpcall1.c

Benchmarks Benchmarking source codes:

- kernel/generic/include/bench/cht_bench.h
- kernel/generic/src/bench/cht_bench.c
- uspace/app/rcubench/rcubench.c

Appendix C

Numerical results

	Read-side scalability List traversals/second vs threads (mean ± std. deviation)			
	ideal	a-rcu	pp-rcu	spinlock
1	2950902579 ± 573954	2169385779 ± 267817	1114574029 ± 1748201	828663398 ± 383819
2	5662887116 ± 8110248	4134548275 ± 5385807	2116331765 ± 5735110	261233049 ± 1159017
3	8501670707 ± 33501314	6216024064 ± 11330622	3200677642 ± 2925867	170511564 ± 176933
4	11367232307 ± 344921	8290290892 ± 59306393	4273952849 ± 442214	148989542 ± 55264

	Write-side overhead Operations/second vs update fraction (mean ± std. deviation)		
	a-rcu + spinlock	pp-rcu + spinlock	spinlock
0	4506399539 ± 8960894	2080584852 ± 169921	96003686 ± 57382
5	586999398 ± 1745016	494056295 ± 1481447	82575360 ± 452994
10	310876569 ± 681910	270534246 ± 1473085	80314368 ± 303091
20	163309158 ± 293817	143357109 ± 1731122	76952371 ± 409621
30	113429708 ± 383135	99209392 ± 410279	72908800 ± 269769
40	89836632 ± 318718	76577104 ± 437364	71039168 ± 323794
60	65188659 ± 338496	54841776 ± 421772	65811251 ± 236898
100	46766489 ± 388576	40461506 ± 546131	73813196 ± 453047

Hash table lookup scalability
Lookups/second vs threads
(mean ± std. deviation)

	cht/a-rcu	ht/bkt-locks	ht/spinlock
1	197079859 ± 126061	159868518 ± 803451	184588697 ± 443519
2	379099545 ± 1431777	255446220 ± 454991	149671116 ± 308785
3	438947020 ± 391574	283325235 ± 502110	84567654 ± 264590
4	626969804 ± 651086	376936857 ± 480696	90754252 ± 85448

	Hash table update overhead Operations/second vs update fraction (mean ± std. deviation)		
	cht/a-rcu	ht/bkt-locks	ht/spinlock
0	601161728 ± 339271	339935232 ± 150875	86573056 ± 29308
5	412352512 ± 1484582	239403008 ± 122606	61603840 ± 35895
10	314376192 ± 913359	220921856 ± 54831	55705600 ± 46340
20	187236352 ± 6183132	189857792 ± 85448	46071808 ± 29308
30	138333388 ± 2111642	166592512 ± 35895	43646976 ± 65536
40	107557683 ± 2289730	148294860 ± 344921	38784204 ± 71791
60	75274649 ± 473948	121189171 ± 181855	34288435 ± 85448
100	59323187 ± 863733	102288588 ± 255506	31457280 ± 65536

	Futex kernel subsystem performance Futex operations/second vs threads (mean)	
	cht cache	ht spinlock
1	4084050	1877756
2	7918158	3032343
3	11970898	2924545
4	16310713	2543497
6	19126115	2042172
7	23803651	1664986

Bibliography

[1] ARM Limited. *ARM Architecture Reference Manual: ARMv7-A and ARMv7-R Edition*, 2010.

[2] Cliff Click. A lock-free hash table. In *JavaOne Conference*, 2007.

[3] Cliff Click. Towards a scalable non-blocking coding style. In *JavaOneSM Conference*, 2008.

[4] Mathieu Desnoyers, Paul E. McKenney, Alan Stern, Michel R. Dagenais, and Jonathan Walpole. User-level implementations of read-copy update. *IEEE Transactions on Parallel and Distributed Systems*, 23:375–382, 2012.

[5] Thomas E. Hart, Paul E. McKenney, Angela Demke Brown, and Jonathan Walpole. Performance of memory reclamation for lockless synchronization. *J. Parallel Distrib. Comput.*, 67(12):1270–1285, December 2007.

[6] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

[7] Jim Houston. [RFC&PATCH] alternative rcu implementation]. Available: `https://lkml.org/lkml/2004/8/30/87` [Viewed 30.11.2012], August 2004.

[8] Intel, Santa Clara, CA, USA. *Intel Itanium Architecture Developer's Manual, Volume 2: System Architecture*, May 2010. Available: `http://www.intel.com/content/dam/www/public/us/en/documents/manuals/itanium-architecture-software-developer-rev-2-3-vol-2-manual.pdf` [Viewed 30.11.2012].

[9] Intel, Santa Clara, CA, USA. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*, June 2012. Available: `http://download.intel.com/products/processor/manual/253668.pdf` [Viewed 30.11.2012].

[10] Lai Jiangshan. [RFC PATCH 5/5 single-thread-version] implement per-domain single-thread state machine call_srcu(). Available: `https://lkml.org/lkml/2012/3/6/586` [Viewed 30.11.2012], March 2012.

[11] Doug Lea. java.util.concurrent.concurrenthashmap. http://gee.cs.oswego.edu/cgi-bin/viewcvs.cgi/jsr166/src/main/java/util/concurrent/ConcurrentHashMap.java?revision=1.118, 2012. Revision 1.118.

[12] Paul E. McKenney. *Exploiting Deferred Destruction: An Analysis of Read-Copy-Update Techniques in Operating System Kernels*. PhD thesis, OGI School of Science and Engineering at Oregon Health and Sciences University, 2004. Available: http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf [Viewed 30.11.2012].

[13] Paul E. McKenney. RCU vs. locking performance on different CPUs. In *linux.conf.au*, Adelaide, Australia, January 2004. Available: http://www.linux.org.au/conf/2004/abstracts.html#90 http://www.rdrop.com/users/paulmck/rclock/lockperf.2004.01.17a.pdf [Viewed 30.11.2012].

[14] Paul E. McKenney. Sleepable RCU. Available: http://lwn.net/Articles/202847/ Revised: http://www.rdrop.com/users/paulmck/RCU/srcu.2007.01.14a.pdf [Viewed 30.11.2012], October 2006.

[15] Paul E. McKenney. The design of preemptible read-copy-update. Available: http://lwn.net/Articles/253651/ [Viewed 30.11.2012], October 2007.

[16] Paul E. McKenney. Priority-boosting RCU read-side critical sections. Available: http://lwn.net/Articles/220677/ Revised: http://www.rdrop.com/users/paulmck/RCU/RCUbooststate.2007.04.16a.pdf [Viewed 30.11.2012], February 2007.

[17] Paul E. McKenney. QRCU with lockless fastpath. Available: http://lwn.net/Articles/223752/ [Viewed 30.11.2012], February 2007.

[18] Paul E. McKenney. Using Promela and Spin to verify parallel algorithms. Available: http://lwn.net/Articles/243851/ [Viewed 30.11.2012], August 2007.

[19] Paul E. McKenney. Hierarchical RCU. Available: http://lwn.net/Articles/305782/ [Viewed 30.11.2012], November 2008.

[20] Paul E. McKenney. [patch rfc -tip 4/4] merge preemptable-rcu functionality into hierarchical rcu. Available: https://lkml.org/lkml/2009/7/23/303 [Viewed 30.11.2012], July 2009.

[21] Paul E. McKenney. Re: [patch fyi] rcu: the bloatwatch edition. Available: http://lkml.org/lkml/2009/1/14/449 [Viewed 30.11.2012], January 2009.

[22] Paul E. Mckenney. Memory barriers: a hardware view for software hackers, June 2010.

[23] Paul E. McKenney. Simplicity through optimization. In *linux.conf.au 2010*, Wellington, New Zealand, January 2010. Available: http://www.rdrop.com/users/paulmck/RCU/SimplicityThruOptimization.2010.01.21f.pdf [Viewed 30.11.2012].

[24] Paul E. McKenney. The new visibility of rcu processing. Available: http://lwn.net/Articles/518953/ [Viewed 30.11.2012], October 2012.

[25] Paul E. McKenney, Dipankar Sarma, Ingo Molnar, and Suparna Bhattacharya. Extending RCU for realtime and embedded workloads. In *Ottawa Linux Symposium*, pages v2 123–138, July 2006. Available: http://www.rdrop.com/users/paulmck/RCU/OLSrtRCU.2006.08.11a.pdf [Viewed 30.11.2012].

[26] Maged M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '02, pages 73–82, New York, NY, USA, 2002. ACM.

[27] Maged M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, PODC '02, pages 21–30, New York, NY, USA, 2002. ACM.

[28] Maged M. Michael. Hazard pointers: Safe memory reclamation for lock-free objects. *IEEE Trans. Parallel Distrib. Syst.*, 15(6):491–504, June 2004.

[29] Motorola and IMB Corporation. *PowerPC microprocessor family: the programming environments*, 1994.

[30] Andrej Podzimek. Read-copy-update for opensolaris. Master's thesis, Charles University in Prague, 2010. Available: https://andrej.podzimek.org/thesis.pdf [Viewed 30.11.2012].

[31] Ori Shalev and Nir Shavit. Split-ordered lists: Lock-free extensible hash tables. *J. ACM*, 53(3):379–405, May 2006.

[32] Inc. SPARC International. *The SPARC architecture manual (version 9).* Upper Saddle River, NJ, USA, 1994. Available: `http://www.sparc.org/standards/SPARCV9.pdf` [Viewd 30.11.2012].

[33] Josh Triplett, Paul E. McKenney, and Jonathan Walpole. Resizable, scalable, concurrent hash tables via relativistic programming. In *Proceedings of the 2011 USENIX Annual Technical Conference*, pages 145–158, Portland, OR USA, June 2011. The USENIX Association.