acmqueue

Structured Deferral: Synchronization via Procrastination

We simply do not have a synchronization mechanism that can enforce mutual exclusion.

Paul E. McKenney, IBM

Developers often take a proactive approach to software design, especially those from cultures valuing industriousness over procrastination. Lazy approaches, however, have proven their value, with examples including reference counting, garbage collection, and lazy evaluation. This structured deferral takes the form of synchronization via procrastination, specifically reference counting, hazard pointers, and RCU (read-copy-update).

Synchronization via procrastination extends back to H. T. Kung and Philip Lehman's 1980 paper,[12] with the general principle articulated by Henry Massalin in 1992.[13] Although these ideas have been used in production for decades,[9] they are still unfamiliar to many. This article provides an introduction to structured deferral by means of a fanciful example described in the next section.

MOTIVATING EXAMPLE

In this example, Schrödinger would like to construct an in-memory database to keep track of the animals in his zoo. Births would of course result in insertions into this database, while deaths would result in deletions. The database is also queried by those interested in the health and welfare of Schrödinger's animals.

Schrödinger has numerous short-lived animals such as mice, resulting in high update rates. In addition, there is a surprising level of interest in the health of Schrödinger's cat,[19] so much so that Schrödinger sometimes wonders whether his mice are responsible for most of these queries. Regardless of their source, the database must handle the large volume of cat-related queries without suffering from excessive levels of contention. Both accesses and updates are typically quite short, involving accessing or mutating an in-memory data structure, and therefore synchronization overhead cannot be ignored.

Schrödinger also understands, however, that it is impossible to determine exactly when a given animal is born or dies. For example, suppose that his cat's passing is to be detected by heartbeat. Seconds or even minutes will be required to determine that the poor cat's heart has in fact stopped. The shorter the measurement interval, the less certain the measurement, so that a pair of veterinarians examining the cat might disagree on exactly when death occurred. For example, one might declare death 30 seconds after the last heartbeat, while another might insist on waiting a full minute, in which case the veterinarians disagree on the state of the cat during the second half of the minute after the last heartbeat.

Fortunately, Heisenberg[8] has taught Schrödinger how to cope with such uncertainty. Furthermore, the delay in detecting the cat's passing permits use of synchronization via procrastination. After all, given that the two veterinarians' pronouncements of death were separated by a full 30 seconds, a few additional milliseconds of software procrastination is perfectly acceptable.

The next section illustrates synchronization via procrastination with reference counting, using

this well-understood mechanism to demonstrate some of the less-familiar properties of structured deferral.

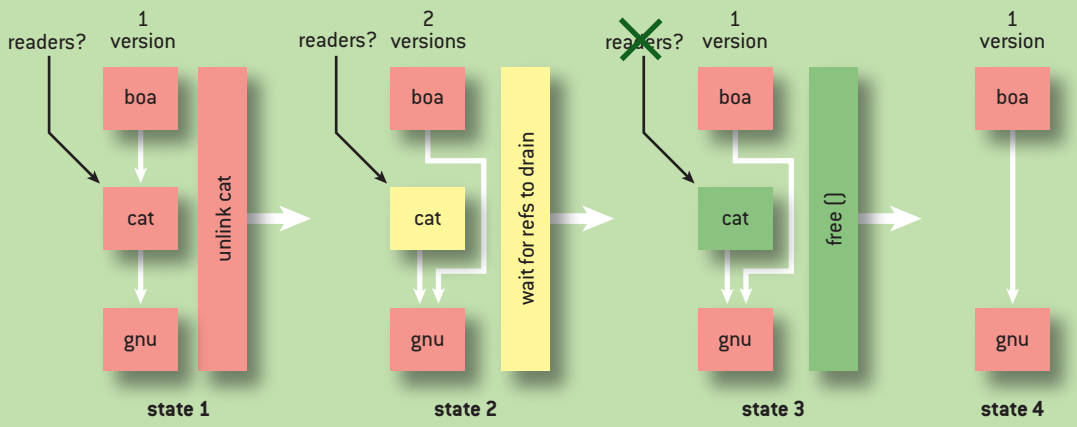
REFERENCE COUNTING

A simple solution for Schrödinger's zoo is to place a reference counter in each animal's data element within a hash table, with collisions handled by chaining. Readers atomically increment the reference before accessing an animal's data element and atomically decrement it afterwards. This provides synchronization only between readers and updaters; updaters must synchronize among themselves using other mechanisms such as locking, nonblocking synchronization, or transactional memory.

The four-state process of removing the data element corresponding to Schrödinger's poor cat is shown in figure 1. The initial state (1) shows one chain of the hash table, representing Schrödinger's boa, cat, and gnu. As indicated by the red color of each box, any number of readers might be referencing these data elements; therefore, updates must be carried out carefully to avoid disturbing these readers. To transition to state 2, the updater stores a pointer to the gnu's data element in the `->next` pointer of the boa's data element. This store must be atomic in the sense that any concurrent reader must see either the old value or the new value, not some mashup of the two. Such a store may be carried out using a C11/C11++ relaxed atomic variable[2] or in older C/C++ compilers, a volatile cast.[3] Note that the cat's `->next` pointer continues referencing the gnu's data element to accommodate readers still referencing the cat.

From this point forward, there is no path to the cat's data element (indicated by its yellow color), so new readers cannot gain access to it. Once the cat's reference counter reaches zero, transitioning to state 3, all readers that had a reference to the cat's data structure have released their references, indicated by the green color of the cat's box. Because there is still no path to the cat's data element, new readers still cannot gain a reference, so it is now safe to transition to state 4 by freeing the late cat's data element.

FIGURE 1 **Deferred Deletion Via Reference Counting**



This sequence of state transitions has therefore safely removed the cat's data element from the hash table, despite the presence of concurrent readers. There remains, however, the problem of obsolete references, to say nothing of correctness and performance. These are discussed in the following sections.

OBSOLETE REFERENCES

While in state 2, old readers can still hold old references to the cat's data structure, but new readers cannot gain a reference. These concurrent readers can therefore disagree as to whether the cat is still alive, just as the veterinarians in the example disagreed. This disagreement is therefore not a bug, but rather a faithful reflection of external reality.

In other cases, disagreements can be detected and rejected. For example, if accesses and updates to a given data element were protected by mutual exclusion, a "deleted" field would allow readers to reject deleted data by acting as if the search had failed.[1]

Finally, suppose that an algorithm uses the common idiom that makes a decision while holding a lock, and then relies on that decision after releasing the lock. This algorithm is in fact relying on obsolete information because some other CPU might acquire the lock and change the data on which the decision was based—while the first CPU is still relying on its now-obsolete decision.

To see how this idiom works, consider a networking stack that acquires a lock, makes a routing decision, transmits a packet, releases the lock, and then updates statistics. Because the statistics need to reflect where you actually sent the packet, as opposed to where you might have sent it had you waited, using "obsolete" data is in fact correct. Furthermore, in most cases, the software does not actually transmit the packet but instead causes the hardware to queue the packet for later transmission. By the time the hardware actually transmits the packet, the routing decision might well have changed. Worse yet, it can take hundreds of milliseconds for a packet to travel from its source to a distant destination, providing even more opportunity for the routing decision to change. Therefore, it is reasonable to make the routing decision while holding the lock, and then release the lock before transmitting the packet—or to use synchronization via procrastination.

In addition, detecting hardware failure often takes significant time, resulting in a period during which it is uncertain whether or not the hardware has failed. Furthermore, many updates have a wide timing window—for example, a regulatory change might require a security-configuration update within a 90-day period. In both cases and in many similar situations, a few extra milliseconds of software procrastination during the update are quite acceptable. In fact, software procrastination can enable readers to become aware of external changes sooner[16] (figure 17 in citation #16), thus reducing response time.

In short, synchronization via procrastination is especially useful when interacting with external state.

CORRECTNESS AND PERFORMANCE

Unfortunately, naïve use of reference counters results in several problems. The first problem is a race between deletion and read-side reference acquisition. To illustrate, suppose a reader fetches the boa's `->next` pointer while in the first state of figure 1 and is then preempted. Before this reader resumes, an updater sequences through the rest of the states in the figure. When the reader resumes, it will atomically increment what used to be the cat's reference counter, but which might now be

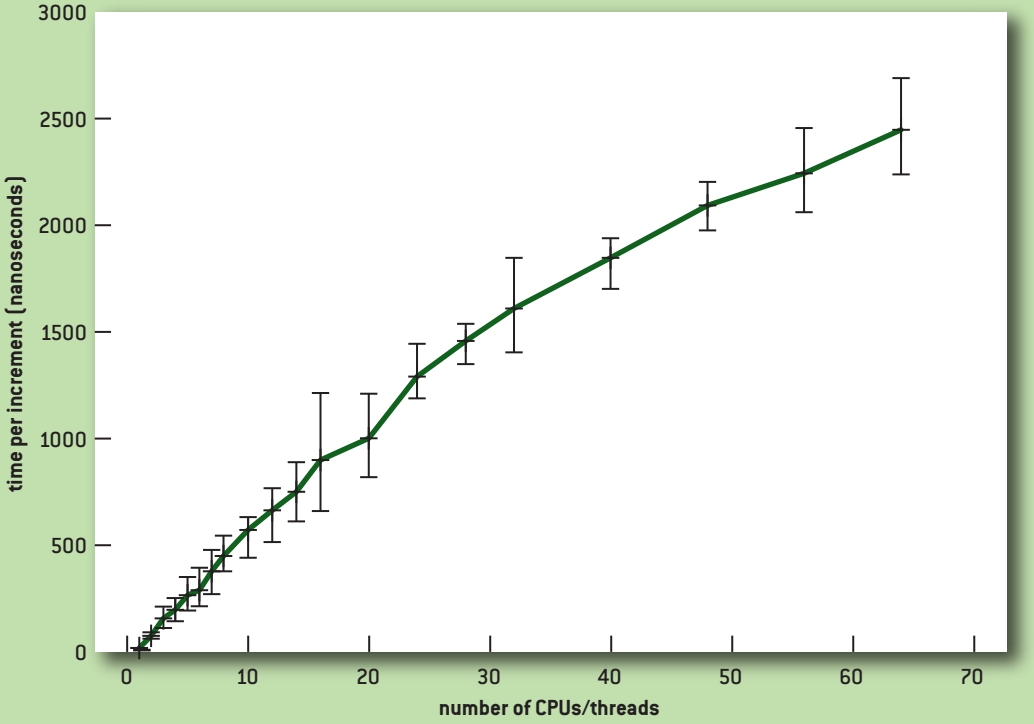
something else entirely, corrupting the application's state. This problem can be avoided via complex schemes based on compare-and-swap[21] (corrected by Maged Michael and Michael Scott[18]), but this results in abysmal performance.[7] Another way of avoiding this problem is to use a garbage collector so that the cat's data structure persists for as long as it is referenced.

If there is no garbage collector, another approach is to reference-count the entire hash chain rather than the individual data elements. This works for a fixed-size hash table but falls prey to a similar race condition if the hash table can be resized. In theory, resizing can be handled using a global reference counter covering the entire hash table, but in practice this increases the probability that a continuous stream of concurrent readers would prevent the counter from ever reaching zero. If the counter never reaches zero, data elements removed from the hash table cannot be freed, eventually resulting in failure caused by memory exhaustion.

Furthermore, as shown in figure 2, the atomic increment of a single variable simply does not scale: cross-thread references can be extremely expensive, because electrons are not infinitely fast. If a single thread attempts to increment a variable atomically, the cost on an Intel Xeon Westmere-EX system is about 10 nanoseconds for a single thread, rising to about 2,500 nanoseconds for 64 threads (the change in slope of the line is caused by hardware multithreading). This means that the number of accesses drops from about 100 million for a single thread to about 400,000 on 64 threads. This is not the sort of scalability that Schrödinger requires.

FIGURE 2

Atomic Increment Does Not Scale



Two methods for addressing these issues are hazard pointers and RCU, which are discussed in the next two sections. Another approach, called proxy collectors (http://atomic-ptr-plus.sourceforge.net/), amortizes the reference-count overhead but falls outside the scope of this article.

HAZARD POINTERS

The key insight behind hazard pointers is that reference counters can be implemented inside-out. Rather than incrementing an integer associated with a given data element, you instead record a pointer to that data element in a per-thread list of hazard pointers. The number of times a given data element's pointer occurs in the concatenation of these lists is that element's reference count. When a data element is to be freed, the free operation is deferred until there are no hazard pointers referencing it. Free operations are batched to minimize the number of expensive cross-thread references to the hazard pointers[17] (independently invented by others[10] and available from a number of sources, including http://concurrencykit.org/).

Because hazard pointers are thread-local, they avoid the performance and scalability problems faced by many reference-counter implementations.[7] They also avoid the race condition just described, as can be seen in `hp_acquire()` on lines 1-14 of figure 3. The key points here are the need to allocate a hazard pointer (line 3), the need to avoid the race condition (the reload and check on line 9), and the need to defeat compiler code-motion optimizations. Compiler code motion is addressed by `ACCESS_ONCE()` on lines 6, 7, and 9, which may be implemented either as volatile casts or as C11/C++11 volatile relaxed atomic loads and stores. Compiler and CPU code motion is addressed by the memory barrier on line 8, which prevents reordering the prior store on line 7 with the subsequent load on line 9 and is thus required even on relatively strongly ordered systems such as x86. The NULL return on line 11 signals the caller that it must restart the hazard-pointer traversal from the beginning. All of this taken together ensures that the updater will have a correct view of the state of the hazard pointers.

Releasing a hazard pointer is straightforward: simply set it to `NULL`, either preceded by a memory barrier or using a C11/C++11 store-release operation; then free it so that it can be reused, as shown in `hp_release()` on lines 16-21 of figure 3. The `hp_release()` function's sole argument is the pointer returned by `hp_acquire()`.

Most programs written using explicit reference counters can be easily converted to use `hp_acquire()` and `hp_release()`. However, algorithms that require only a fixed number of hazard pointers can allocate them statically, thus avoiding the overhead of `hp_alloc()`'s and `hp_free`'s dynamic allocation, as shown in `hp_record()` on lines 23-35 of figure 3. Note that this approach does not release a given hazard pointer until that pointer is reused. In the worst case, this approach prevents a small fixed amount of memory from being freed, which is normally harmless. Although there are algorithms that require unbounded numbers of hazard pointers, simple searches and traversals of many data structures require at most two hazard pointers.

Hazard pointers work extremely well in many situations, but they do have some shortcomings.

- Retries and memory barriers result in degraded performance.
- If hazard-pointer protection is required for a large group of data elements in a linked structure, then a separate hazard pointer must be acquired for each and every data element.
- Acquiring a hazard pointer requires memory, which must be managed. Although the static-allocation strategy used by Michael[17] works well in small programs for some types of data structures,

FIGURE 3

Hazard-pointer Acquisition and Release

```
1 void **hp_acquire(void **p)
2 {
3    void **hp = hp_alloc();
4    void *tmp;
5
6    tmp = ACCESS_ONCE(*p);
7    ACCESS_ONCE(*hp) = tmp;
8    smp_mb();
9    if (tmp != ACCESS_ONCE(*p)) {
10      hp_free(hp);
11      return NULL;
12   }
13   return hp;
14 }
15
16 void hp_release(void **hp)
17 {
18   smp_mb();
19   ACCESS_ONCE(*hp) = NULL;
20   hp_free(hp);
21 }
22
23 int hp_record(void **p, void **hp)
24 {
25   void *tmp;
26
27   tmp = ACCESS_ONCE(*p);
28   ACCESS_ONCE(*hp) = tmp;
29   smp_mb();
30   if (tmp != ACCESS_ONCE(*p)) {
31      ACCESS_ONCE(*hp) = NULL;
32      return 0;
33   }
34   return 1;
35 }
```

in general allocation and freeing are required.

• Hazard pointers provide no provision for taking additional action when freeing memory—for example, shutting down threads or hardware associated with a given data element.

• Finally, in the general case, the code must keep track of the hazard pointers and explicitly release them when they are no longer needed.

Some of these shortcomings are inherent in hazard pointers' strengths, which include nonblocking updates, strong ordering properties, and reduced memory overhead.[7] Nevertheless, in operating system kernels and large applications, hazard pointers' shortcomings can outweigh their strengths. The next section presents an alternative design that addresses these shortcomings—albeit by sacrificing some of hazard pointers' strengths. The lesson is, use the right tool for the job!

READ-COPY-UPDATE

The goal is to produce a reference-counting scheme that (1) has low (and preferably zero) overhead; (2) needs no memory allocation when acquiring references; (3) can protect arbitrary numbers of data elements with a single operation; (4) permits additional actions to be taken prior to freeing a given data element. One mechanism that meets these goals is RCU (read-copy-update).[16]

To ensure that the goal of low overhead is met, let's define acquiring a reference (`rcu_read_lock()`) and releasing it (`rcu_read_unlock()`) as no-ops that generate no code, thus achieving the best conceivable performance, scalability, realtime response, wait freedom, and energy efficiency. This admittedly unconventional approach has the additional benefit of dispensing with memory allocations. Given that `rcu_read_lock()` and `rcu_read_unlock()` take no arguments, there is no way for them to specify a particular data element, so they also meet the third goal of protecting all data elements.

Skeptics might argue that if `rcu_read_lock()` and `rcu_read_unlock()` generate no code, they cannot affect machine state and therefore cannot possibly act as synchronization primitives. Let us press on nonetheless: after all, only those who have gone too far can possibly know how far they can go.

Figure 1 shows how an element can be removed from a linked list despite the presence of concurrent readers, which indicates that Schrödinger does not need mutual exclusion. The transition from state 2 to state 3, however, requires waiting for all preexisting readers to complete. Therefore, the challenge is to implement an operation that waits for preexisting readers, even though `rcu_read_lock()` and `rcu_read_unlock()` generate no code.

To surmount this challenge, let's first consider the nonpreemptive software environment, where a given thread continues to execute until it voluntarily relinquishes the CPU. Examples of nonpreemptive software environments include the Linux kernel when built with `CONFIG_PREEMPT=n`, DYNIX/ptx, and numerous embedded systems. This type of environment can impose the convention that a thread is forbidden from relinquishing its CPU if it has executed `rcu_read_lock()` but has not yet reached the matching `rcu_read_unlock()`. (This same convention is required when holding pure spinlocks to avoid deadlock.) Such a thread is said to be in an *RCU read-side critical section*. Just as with reference counting and locking, if a reference to a given data element is obtained within a given RCU read-side critical section, that reference must be dropped before exiting that section, unless the data element has been handed off to some other synchronization mechanism.

Now suppose that the updater thread wishing to transition from state 2 to state 3 in figure 1 sees CPU 0 execute a context-switch operation. Given that RCU readers must refrain from relinquishing their CPUs, this context switch implies all of CPU 0's prior RCU read-side critical sections have completed—in other words, that CPU 0 is in a *quiescent state*. In addition, subsequent readers running on CPU 0 cannot acquire a reference to Schrödinger's cat because there is no longer a path to the cat. Therefore, once the updater has observed a context switch on each CPU, there can no

longer be any readers referencing the cat—in other words, a *grace period* will have elapsed.

This procrastination procedure is encapsulated in the RCU primitive `synchronize_rcu()`, the simplest implementation of which simply runs in turn on each CPU, thus ensuring that each CPU has executed at least one context switch. The operation of `synchronize_rcu()` is shown in figure 4, where each horizontal double-ended arrow represents an RCU read-side critical section beginning with `rcu_read_lock()` and ending with `rcu_read_unlock()`, and where each circle represents a context switch. The updater, running on CPU 3, first removes the cat's element using `rcu_assign_pointer()` and then invokes `synchronize_rcu()` in order to wait for each CPU to execute a context switch, thereby guaranteeing the completion of all preexisting readers that might have held a reference to the cat. Once `synchronize_rcu()` completes, the updater is free to execute any additional actions required (for example, shutting down associated threads or hardware) and then free the element. Production-quality implementations also provide an asynchronous counterpart, `call_rcu()`, that invokes a specified function at the end of a grace period. During the early part of the grace period, the CPUs might well disagree on the cat's state, which is only fitting for Schrödinger's cat.

In short, `rcu_read_lock()` and `rcu_read_unlock()` do not affect machine state, but they do not need to. Instead, they act on the developer, who is required to follow the convention that RCU readers must not release their CPUs. This form of RCU can therefore be thought of as synchronization via social engineering. (Other synchronization mechanisms also rely on social engineering; examples include prohibiting data races and a specific lock protecting all uses of a given object.)

A "textbook" RCU implementation is quite simple, as can be seen from the 20 lines of code in figure 5. Furthermore, in an SC (sequentially consistent) environment, only `rcu_read_lock()`,

FIGURE 4 **Nonpreemptible Grace Period**

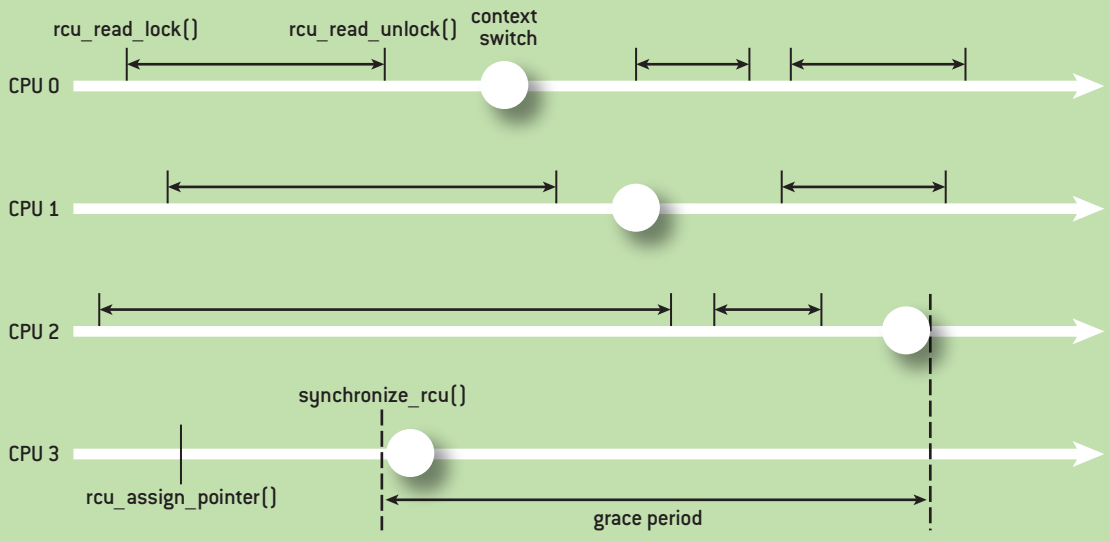


FIGURE 5

Textbook Implementation of RCU

```
1 #define rcu_read_lock()
2 #define rcu_read_unlock()
3 void synchronize_rcu(void)
4 {
5    int cpu;
6
7    for_each_online_cpu(cpu)
8      run_on(cpu);
9 }
10 #define rcu_dereference(p) \
11 ({ \
12   typeof(p) _p1 = ACCESS_ONCE(p)); \
13   smp_read_barrier_depends(); \
14   _p1; \
15 })
16 #define rcu_assign_pointer(p, v) \
17 ({ \
18   smp_wmb(); \
19   ACCESS_ONCE(p) = (v); \
20 })
```

`rcu_read_unlock()`, and `synchronize_rcu()` are required, consisting of only nine lines of code. Production-quality implementations are larger to meet severe performance, scalability, response time, and energy efficiency requirements. User-mode RCU implementations are also available,[4] as are preemptible-kernel implementations.[5] (The full Linux-kernel RCU implementation is beyond the scope of this article.[14])

Because no mainstream computing system is SC, however, `rcu_dereference()` must be used when fetching an RCU-protected pointer, and `rcu_assign_pointer()` must be used when mutating an RCU-protected pointer. In SC systems, `rcu_dereference()` and `rcu_assign_pointer()` reduce to a simple load and store, respectively. In C11/C++11, `rcu_dereference()` is a load consume, and `rcu_assign pointer()` is a store release. Figure 6 shows the deletion process in terms of the RCU primitives.

RCU therefore meets its goals of low overhead, read-side memory-allocation avoidance, arbitrarily large scope of protection, and support of cleanup actions. Of course, there is no free lunch: the price of these goals is giving up hazard pointers' nonblocking, bounded-memory, and ordering properties. As can be seen in figure 7, however, RCU is nonetheless heavily used within the Linux kernel. That said, RCU is specialized, so that locking is used roughly an order of magnitude more heavily than is RCU.

FIGURE 6

Deferred Deletion Via RCU

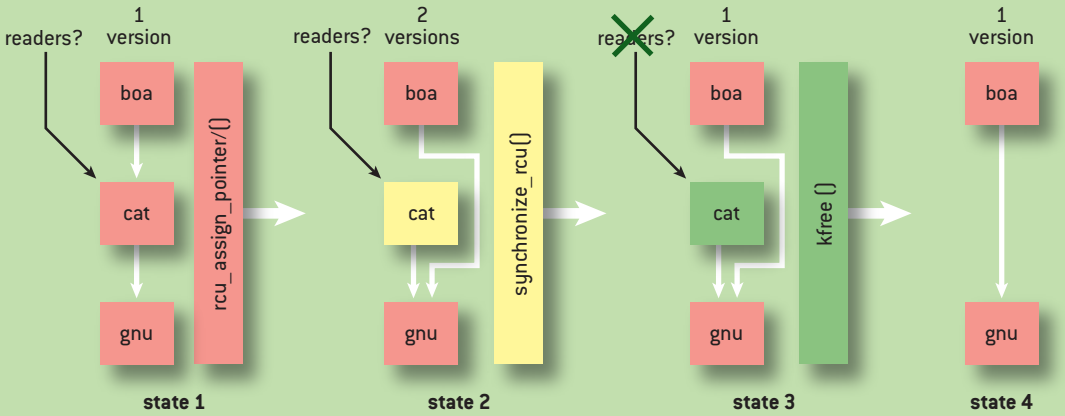
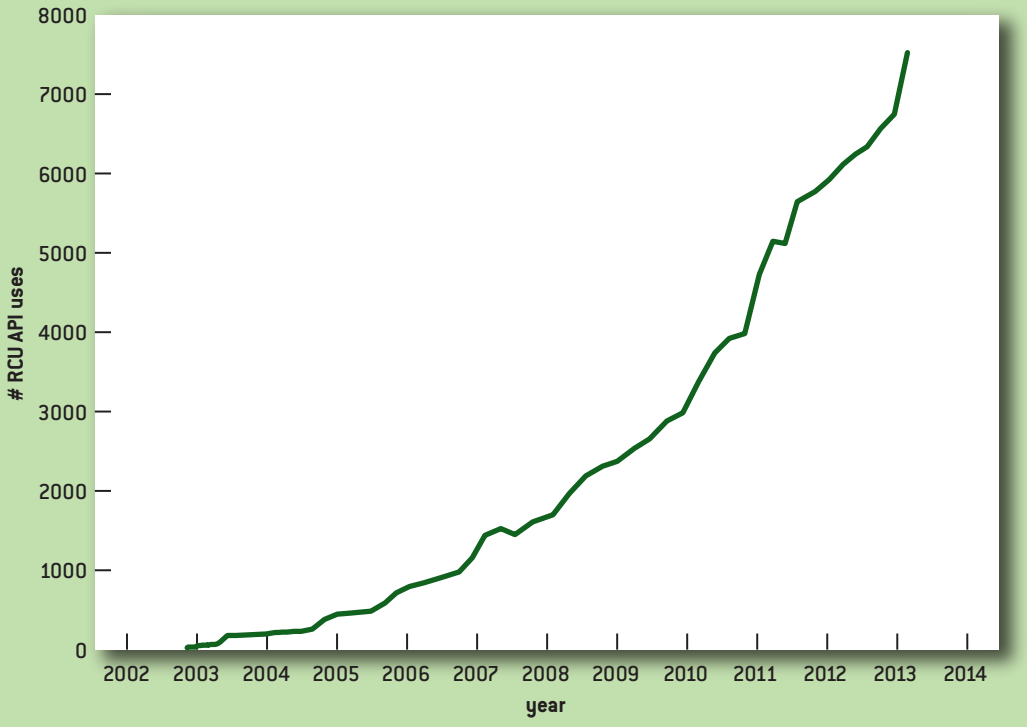


FIGURE 7

RCU Applicability to the Linux Kernel



COMPARISON

This section gives a brief overview of the performance of hazard pointers and RCU for Schrödinger's zoo, providing qualitative and quantitative comparisons based on the example. This is followed by some rules of thumb that help determine when to use synchronization via procrastination.

QUALITATIVE COMPARISON

Table 1 compares the properties of hazard pointers and RCU, demonstrating that these two synchronization mechanisms represent different design points:

- Each advantage is inherently intertwined with a corresponding disadvantage. Hazard pointers' bounded memory overhead advantage over RCU requires that developers carefully acquire a hazard pointer for each data element that readers traverse. This constitutes the corresponding disadvantage versus RCU's ability to protect all data elements with a single `rcu_read_lock()`.
- Similarly, hazard pointers' nonblocking advantage (coupled with bounded memory) requires that updaters force concurrent hazard-pointer readers to retry their hazard-pointer acquisitions. This retrying constitutes the corresponding disadvantage versus RCU.
- Finally, hazard pointers' linearizability advantage requires read-side memory barriers during hazard-pointer acquisition. The resulting reduced read-side performance constitutes the corresponding disadvantage versus RCU. (There is some debate as to whether linearizability is universally useful.[6,22])

Other differences appear to be implementation choices rather than inherent properties of the underlying mechanisms. For example, hazard pointers offer no way to invoke cleanup actions when a given data element is finally reclaimed. This precludes shutting down hardware, threads, or other active components that might be associated with that data element. However, the hazard-pointer's

TABLE 1: **Comparison of hazard pointers and RCU**

		Hazard Pointers		RCU
Memory	+	Bounded memory overhead.	–	Blocked readers can exhaust memory (avoid via RCU priority boosting).
	–	Individual hazard-pointer acquisition required for each data element.	+	Single rcu read lock() protects all data elements.
Blocking	+	Completely nonblocking.	–	Updaters can block (but can use non-blocking subset of RCU API [4]).
	–	Updates can force concurrent readers to retry.	+	Readers and updaters make unconditional concurrent forward progress, deterministic wait-free readers.
Linearizability	+	Linearizable.	–	Non-linearizable (but often not observable).
	–	Readers require heavy-weight memory barriers, but still low overhead.	+	Lower read-side overhead: no readside memory barriers.

+ = advantage – = disadvantage

mechanism could be augmented to provide the possibility of RCU-like cleanup actions if desired. Of course, providing cleanup actions would have other consequences, including prohibiting the common usage pattern where hazard pointers are not cleaned up immediately after a traversal. This usage pattern is harmless if no cleanup actions are in place, as it simply retains a small amount of memory that could otherwise be freed. In the presence of cleanup actions, however, this usage pattern could indefinitely defer cleanup, which would have the possibly unacceptable side effect of preventing reuse of the corresponding hardware or thread data.

Current implementations of hazard pointers record only the beginning of a structure when freeing it, causing difficulties when pointers reference structures nested within other structures. This nesting is quite common in some environments, including the Linux kernel. Extending hazard pointers to handle internal pointers is quite straightforward, however: when freeing a structure, you should pass its size, as well as its address, allowing hazard pointers to that structure's interior to be properly handled.

Because hazard pointers reference only specific structures, races with updates must be handled by restarting the read-side traversal from the beginning. This need to restart stems from the fact that once a structure has been removed, updates no longer change pointers emanating from that structure, so those pointers can no longer be trusted.

Restarting traversals is straightforward for simple data structures but can pose a significant software engineering challenge for large multilinked structures with deeply nested access functions or methods. Because this article focuses on simple hash tables with chaining, it does not explore these issues. Perhaps traversals can be conveniently restarted using exceptions in languages supporting them.

In short, the choice between hazard pointers and RCU depends on the workload's requirements, including the other synchronization mechanisms used in the program. For example, if the program were memory-constrained, then hazard pointers would likely be the right choice. In contrast, if the program used large multilinked structures with deeply nested access functions or methods, then RCU might be the right choice.

QUANTITATIVE COMPARISON

This section presents the results of benchmarks that were run on fixed-size hash tables protected by a single global lock, per-bucket locks, hazard pointers, and RCU. The tests were run on a Westmere-EX x86 system with 32 cores (64 hardware threads) running at 2 GHz. To ease comparisons, hazard-pointer updates were protected using per-bucket locking. Blocking was avoided by providing each thread with its own CPU.

The tests used a signal-based RCU variant from liburcu[4] (available on a number of recent Linux distributions, including Debian, Fedora, OpenSUSE, and Ubuntu). This is slower than the zero-cost (QSBR) implementation described earlier, but the zero-cost implementation requires that every thread periodically reside in a quiescent state. Because not all applications are structured to meet this requirement, and because hazard pointers do not require any particular application structure, fairness dictates that the RCU implementation used in these tests also not require any particular structure. For purposes of comparison, the QSBR RCU implementation offers roughly 10 percent better performance than does signal-based RCU.

Because the hash tables are simple linked structures, hazard pointers are statically allocated and

FIGURE 8

Read-Only Performance on Simple Hash Table

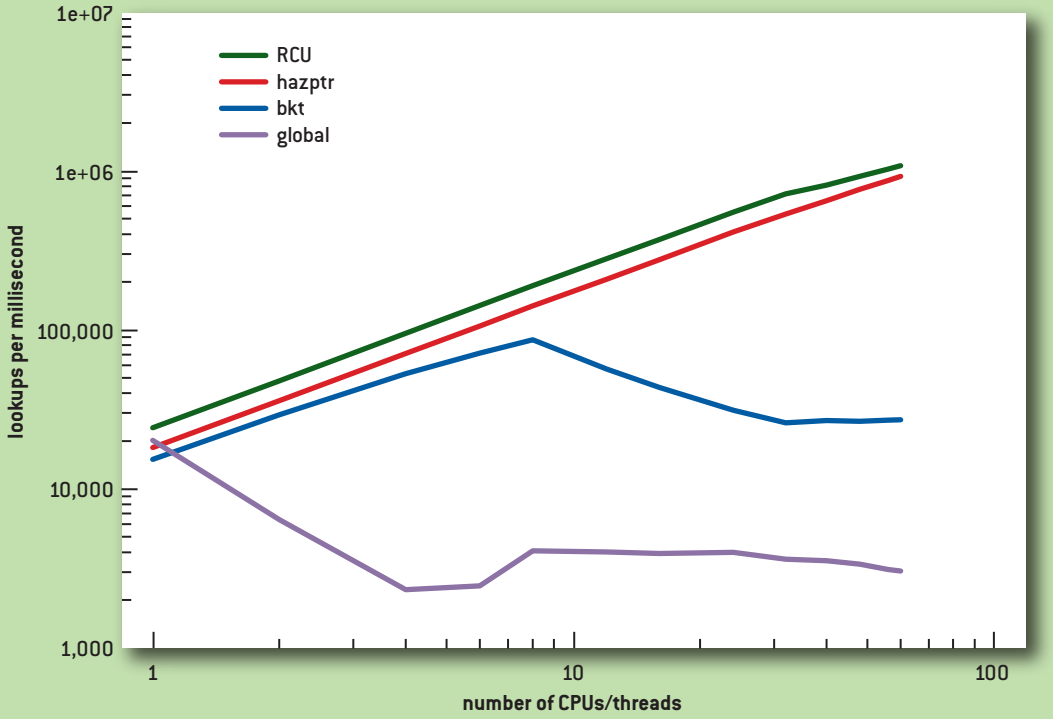
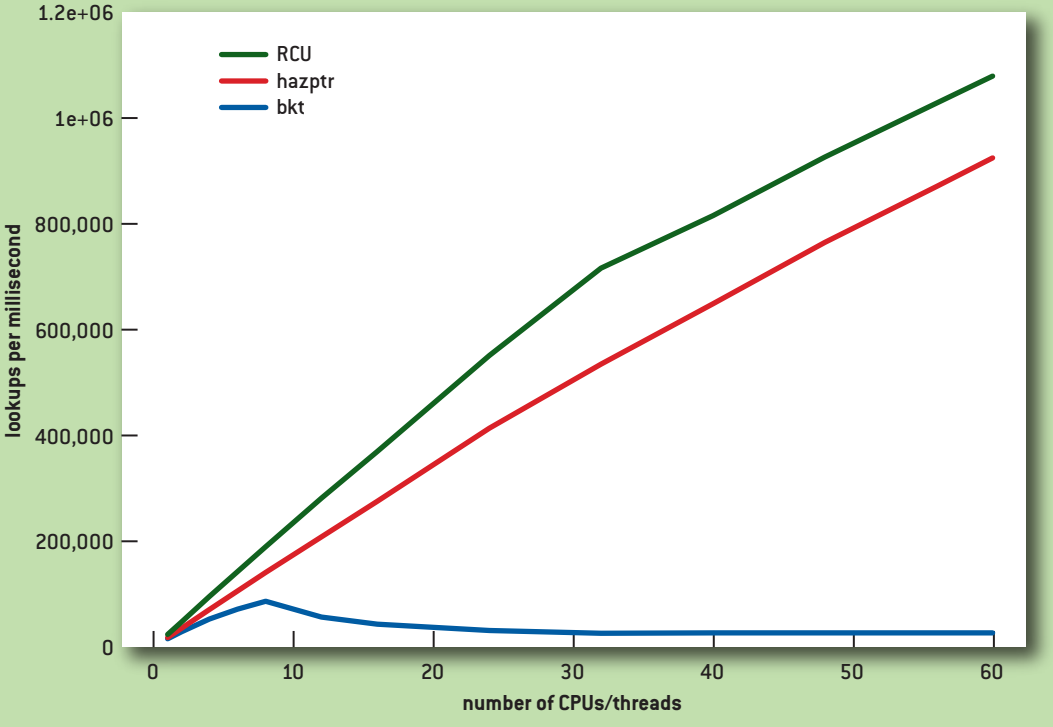


FIGURE 9

Read-Only Performance on Simple Hash Table (Linear)



are not explicitly released, thus eliminating the overhead of allocation, free, and release operations. Schrödinger recognizes that other algorithms and data structures may require dynamically allocated hazard pointers, which would reduce their performance.

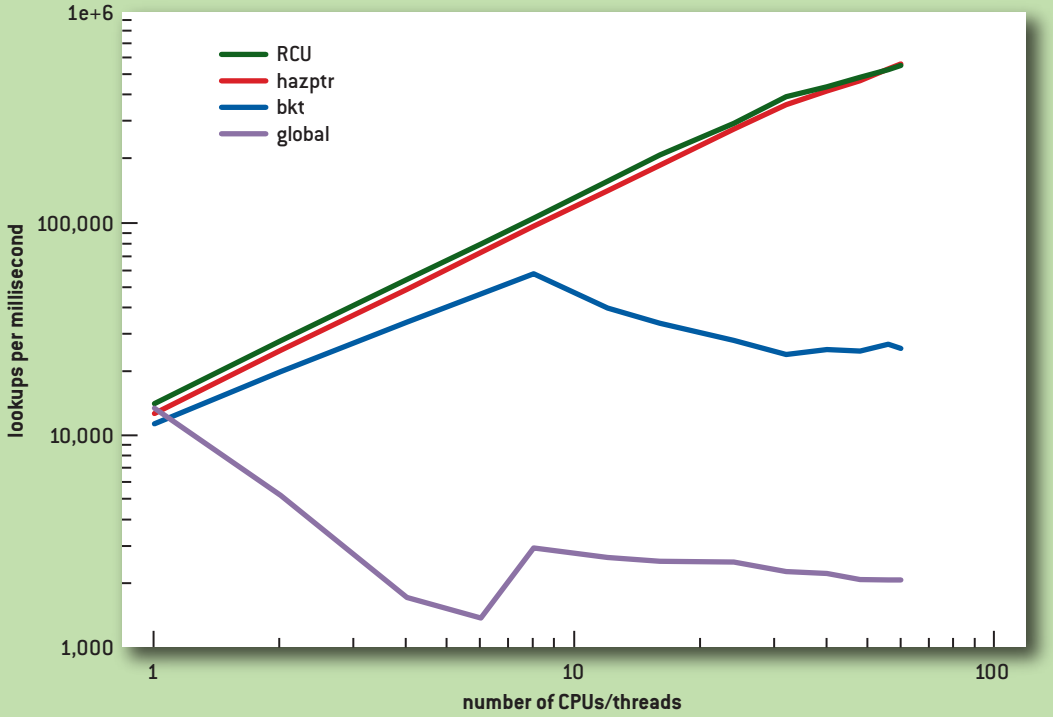
Figure 8 shows the results of a lookup-only test of a hash table with simple integers for keys. This hash table has 1,024 buckets with chaining, and it contains 1,024 elements out of a total population of 2,048. Lookup keys were randomly selected so that half of the lookups succeeded. As expected, global locking performs quite poorly. Per-bucket locking (*bkt*) scales linearly up to about eight CPUs, then drops off as a result of increasing lock and memory contention as the number of CPUs increases relative to the number of hash buckets. Increasing the number of buckets results in better scalability and performance, as expected. Both hazard pointers (*hazptr*) and RCU scale very nearly linearly with excellent performance.

Figure 9 plots the same data on a linear scale, which more clearly shows RCU's and hazard pointers' performance and scalability advantages compared with per-bucket locking. Note also RCU's inflection at 32 CPUs: hazard pointers take better advantage of this particular system's dual-threaded hardware than does RCU. (The runs using 32 or fewer CPUs run each thread on its own core, while the runs with more CPUs run two threads per core.) Nevertheless, RCU enjoys a 14 percent performance advantage at 60 CPUs and a 23 percent performance advantage at 32 CPUs.

Figure 10 shows read-only performance of a prototype of Schrödinger's in-memory database, which is a 1,024-bucket hash table with chaining that uses ASCII strings of up to 31 characters as

FIGURE 10

Read-Only Performance on Schrödinger's Hash Table



keys. The results are similar to those for the integer-keyed hash table, except that RCU's advantage over hazard pointers decreases to about 8 percent at 32 CPUs and to nil at 60 CPUs because of the heavier-weight key operations.

Figure 11 shows the effect of increasing fractions of the queries accessing Schrödinger's cat for 60 CPUs. Global locking performs poorly throughout, as expected. Per-bucket locking's effectiveness is decreased by the increasing level of contention on the hash bucket containing the cat. The decrease becomes catastrophic when more than about 50 CPUs execute cat-related queries. RCU's throughput is unaffected by the fraction of cat-related queries, but hazard pointers' throughput decreases with increasing fractions of cat-related queries, ranging from parity with RCU down to about a 25 percent performance penalty. This decrease comes as a result of the memory barriers required by hazard pointers. Removing these memory barriers restores performance parity with RCU, but also results in an unsafe hazard-pointer implementation. This behavior suggests the hypothesis that memory barriers are more expensive when multiple CPUs access the same memory locations, which seems plausible given that CPUs accessing disjoint memory locations cannot detect each others' memory-access order.

Figure 12 shows data taken from the same runs as for figure 11, but shows only the throughput of cat-related queries. The global-lock, hazard pointer, and RCU results scale as expected: the more cat-related queries attempted, the more that get done. Per-bucket locking does not fare as well because

FIGURE 11

Read-Only Cat-Heavy Performance on Schrödinger's Hash Table

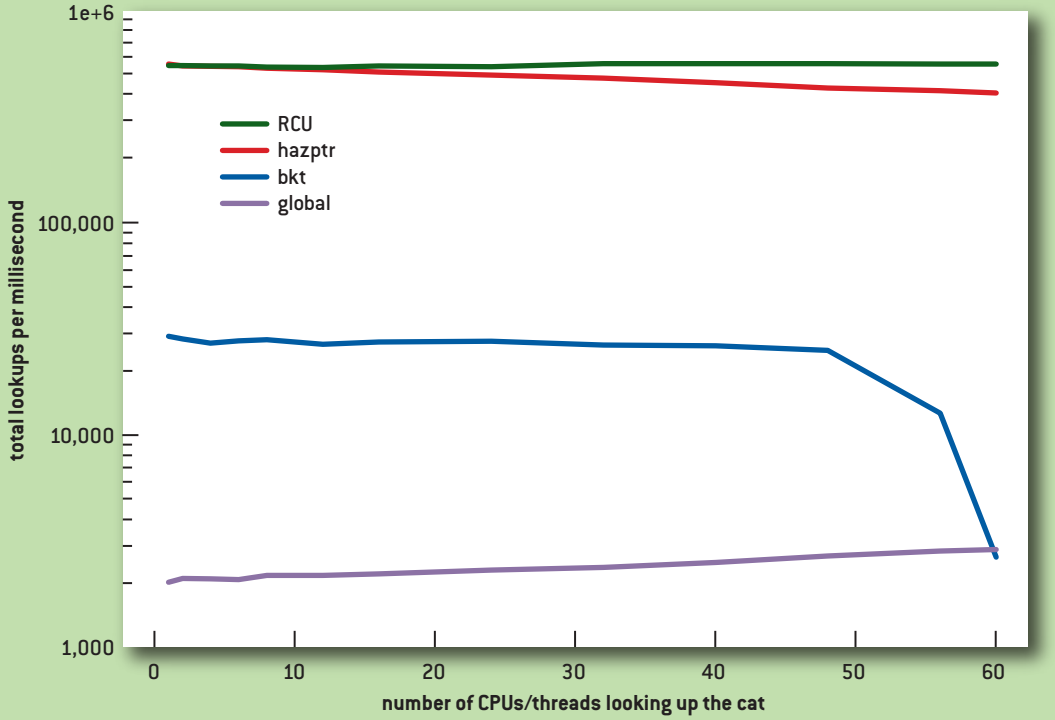
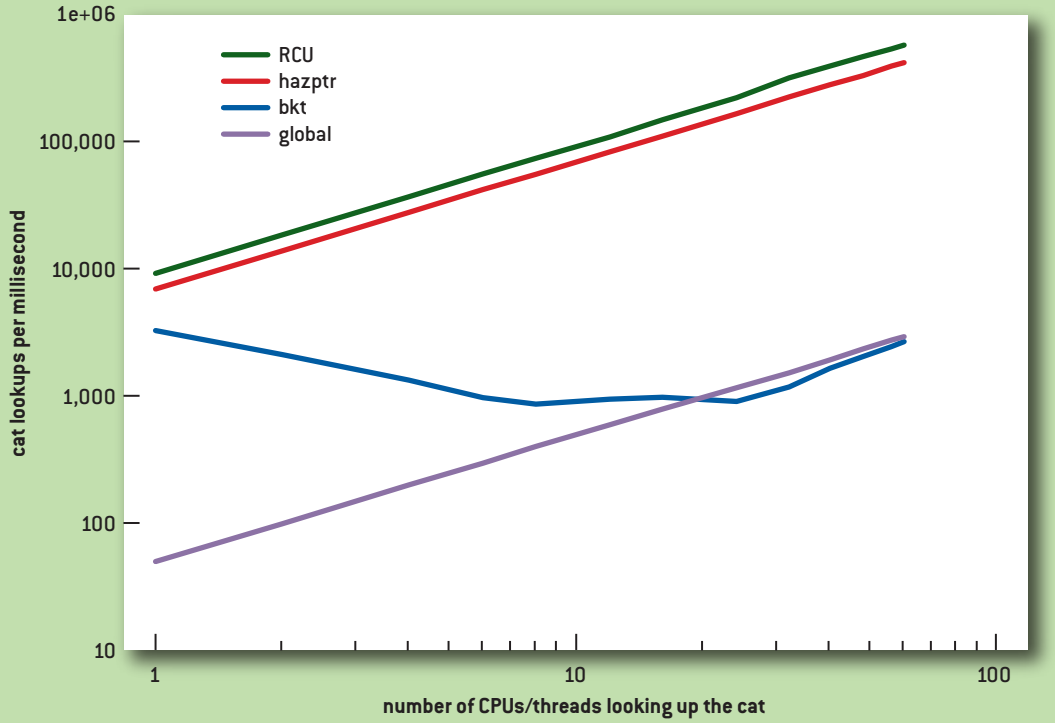


FIGURE 12

Read-Only Cat-Heavy Performance on Schrödinger's Hash Table: Cat Queries



the contention on the cat's hash bucket increases as the number of attempted cat-related queries increases. Therefore, beyond about 20 CPUs, per-bucket locking performance is similar to global locking.

Of course, Schrödinger needs to do updates. Table 2 shows the results of a test with 15 threads doing updates, 15 threads querying the cat, and 30 threads querying random animals. All numbers are events per millisecond, consistent with the earlier figures. Although the cat was always present during this test, the other animals were randomly added and deleted, so there was a 50 percent probability of any given animal being present at any given time. Therefore, 50 percent of the non-cat queries could be expected to fail, which was the case here.

Global locking performed poorly, as expected. Per-bucket locking was outperformed by hazard pointers by a factor of three, and hazard pointers were in turn outperformed by RCU by an additional factor of two. In contrast, updates (adds and deletes) slowed by 10 percent from per-bucket locking to hazard pointers and in turn slowed by an additional 50 percent from hazard pointers to RCU. This is not unexpected, given that both hazard pointers and RCU intentionally sacrifice update performance in favor of read-side performance. The increase in read-side performance is much larger than the decrease in update-side performance, compared with per-bucket locking.

This does raise the question of whether the low read-side performance of per-bucket locking and hazard pointers was in fact caused by their faster update rates. Additional testing therefore throttled per-bucket locking and hazard pointer update rates to that of RCU. Per-bucket locking read-side

TABLE 2: **Schrödinger's zoo with updates (operations per millisecond)**

Mechanism	Reads	Failed Reads	Cat Reads	Adds	Deletes
Global Locking	799	80	639	77	77
Per-Bucket Locking	13,555	6,177	1,197	5,370	5,370
Hazard Pointers	41,011	6,982	27,059	4,860	4,860
RCU	85,906	13,022	59,873	2,440	2,440

throughput did increase in response to the decrease in update-side lock contention, but only to about 17,000 updates per millisecond, which is nowhere near hazard pointers' 41,011 reads per millisecond, let alone RCU's 85,906 reads per millisecond (zero-cost RCU achieves roughly 100,000 reads per millisecond). Hazard pointer read-side throughput did not change significantly in response to the throttled update rates.

The throughput of both hazard pointers and RCU is quite a bit lower than in the read-only workload shown in figure 10. This is because the updates result in read-side cache misses, reducing throughput.

An additional question remains: namely, why RCU's throughput is double that of hazard pointers in this benchmark, given that there was at most a 25 percent difference in the read-only tests. Removing hazard pointers' read-side memory barriers increased throughput to about 80,000 reads per millisecond, demonstrating that memory barriers were the culprit. This in turn suggests that memory barrier overhead is an increasing function of cache-miss rate, so the presence of updates increases the cost of the hazard pointer mechanism.

The data in this section clearly demonstrates the substantial read-side performance benefits of synchronization via procrastination mechanisms such as hazard pointers and RCU. RCU's update-side performance benefits have been demonstrated elsewhere.[4]

WHEN TO PROCRASTINATE?

Although the read-side performance benefits of both hazard pointers and RCU can be sizable, these are specialized mechanisms that are typically used in conjunction with other mechanisms. This raises the question as to when they should be used. Extensive use of RCU in the Linux kernel[15] has led to the following rules of thumb, which may also apply to hazard pointers:

1. Procrastination works extremely well in read-mostly situations where disagreement among readers is acceptable. What constitutes "read-mostly" depends on the workload, but 90 percent reads to 10 percent updates is a good rule of thumb.
2. Procrastination works reasonably well in read-mostly situations where readers must always agree.
3. Procrastination sometimes works well in cases where the numbers of reads and updates are roughly equal and where readers must agree.
4. Procrastination rarely works well in update-mostly situations where readers must agree. There are currently two known exceptions to this rule: (1) providing existence guarantees for update-friendly mechanisms, and (2) providing low-overhead wait-free read-side access for realtime use.

Note that the traditional definition of *read* may be generalized to include writes. An example within Schrödinger's application would be if each animal's data structure included an array of

per-thread cache-aligned variables that count accesses to that animal, thus allowing Schrödinger to evaluate his cat's popularity. This works because these read-side writes do not conflict. Further generalization is not only possible, but also heavily used in practice—for example, permitting conflicting writes that are protected by some other synchronization mechanism.[1] This is permissible because reference counters, hazard pointers, and RCU all permit a wide range of code in their read-side critical sections, including atomic operations, transactions, and locking.

Schrödinger's application permits the traditional definition of *read* and thus falls into the first rule of thumb. It is therefore eminently suitable for synchronization via procrastination. These rules will continue to be refined as experience accumulates. In particular, better mechanisms are needed for update-heavy situations, and there is some promising work in progress in this area.[11,20] Finally, table 1 might be a first step toward rules of thumb for choosing between hazard pointers and RCU.

CONCLUSION

This article has presented an overview of synchronization via procrastination, discussing some of the consequences, such as read-side disagreements on current state, and how these consequences are actually a faithful reflection of the reality external to the computer. This is to be expected: there is simply no synchronization mechanism that can enforce mutual exclusion on any significant fraction of the physical universe. Use of the two popular procrastination mechanisms described here, hazard pointers and RCU, has increased over the past few decades and can be expected to increase further as the use of multicore systems increases.

ACKNOWLEDGMENTS

I owe thanks to Steven Rostedt for the analogy between RCU and Schrödinger's cat (https://lkml.org/lkml/2013/3/29/248); to Maged Michael for many fruitful discussions of hazard pointers and RCU; and to Arjan van de Ven, Peter Sewell, and Susmit Sarkar for discussions on concurrency and memory ordering. I am indebted to Samy Al Bahra, Eduardo Bacchi Kienetz, Seth Jennings, Wade Cline, Mathieu Desnoyers, Ram Pai, Kent Yoder, and Phil Estes for helping render this article human-readable. I am grateful to Jim Wasko for his support of this effort.

REFERENCES

1. Arcangeli, A., Cao, M., McKenney, P. E., Sarma, D. 2003. Using read-copy update techniques for System V IPC in the Linux 2.5 kernel. In *Proceedings of the 2003 Usenix Annual Technical Conference*: 297-310.
2. Becker, P. 2011. Working draft, standard for programming language C++; http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3242.pdf.
3. Corbet, J. 2012. ACCESS_ONCE(); http://lwn.net/Articles/508991/.

4. Desnoyers, M., McKenney, P. E., Stern, A., Dagenais, M. R., Walpole, J. 2012. User-level implementations of read-copy-update. *IEEE Transactions on Parallel and Distributed Systems* 23: 375-382.
5. Guniguntala, D., McKenney, P. E., Triplett, J., Walpole, J. 2008. The read-copy-update mechanism for supporting realtime applications on shared-memory multiprocessor systems with Linux. *IBM Systems Journal* 47(2): 221-236.
6. Haas, A., Kirsch, C. M., Lippautz, M., Payer, H. 2012. How FIFO is your concurrent FIFO queue? In *Proceedings of the 2012 ACM Workshop on Relaxing Synchronization for Multicore and Manycore Scalability*.
7. Hart, T. E., McKenney, P. E., Brown, A. D., Walpole, J. 2007. Performance of memory reclamation for lockless synchronization. *Journal of Parallel and Distributed Computing* 67(12): 1270-1285.
8. Heisenberg, W. 1927. Über den anschaulichen Inhalt der quantentheoretischen Kinematik und Mechanik. *Zeitschrift für Physik* 43(3-4): 172-198. English translation in *Quantum Theory and Measurement*. Eds. J. A. Wheeler and W. H. Zurek, Princeton University Press, 1984.
9. Hennessy, J. P., Osisek, D. L., Seigh II, J. W. 1989. Passive serialization in a multitasking environment. U.S. Patent 4,809,168 (lapsed).
10. Herlihy, M., Luchangco, V., Moir, M. 2002. The repeat offender problem: a mechanism for supporting dynamic-sized, lock-free data structures. In *Proceedings of the 16th International Conference on Distributed Computing*: 339-353.
11. Jacobi, C., Slegel, T., Greiner, D. 2012. Transactional memory: architecture and implementation for IBM System z. The 45th Annual IEEE/ACM International Symposium on MicroArchitecture; http://www.microsymposia.org/micro45/talks-posters/3-jacobi-presentation.pdf.
12. Kung, H. T., Lehman, Q. 1980. Concurrent manipulation of binary search trees. *ACM Transactions on Database Systems* 5(3): 354-382.
13. Massalin, H. 1992. Synthesis: an efficient implementation of fundamental operating system services. Ph.D. thesis, Columbia University, New York, NY.
14. McKenney, P. E. 2010. The RCU API; http://lwn.net/Articles/418853/.
15. McKenney, P. E., Boyd-Wickizer, S., Walpole, J. 2013. RCU usage in the Linux kernel: one decade later; http://rdrop.com/users/paulmck/techreports/RCUUsage.2013.02.24a.pdf.
16. McKenney, P. E., Slingwine, J. D. 1998. Read-copy-update: using execution history to solve concurrency problems. In *Parallel and Distributed Computing and Systems*: 509-518.
17. Michael, M. M. 2004. Hazard pointers: safe memory reclamation for lock-free objects. *IEEE Transactions on Parallel and Distributed Systems* 15(6): 491-504.
18. Michael, M. M., Scott, M. L. 1995. Correction of a memory management method for lockfree data structures. Technical Report TR599, University of Rochester, Rochester, NY.
19. Schrödinger, E. 1935. Die gegenwärtige Situation in der Quantenmechanik. *Naturwissenschaften* 23: 807-812; 823-828; 844-949. English translation: http://www.tuhh.de/rzt/rzt/it/QM/cat.html.
20. Sutton, A. 2013. Concurrent programming with the Disruptor; http://lca2013.linux.org.au/schedule/30168/view_talk.
21. Valois, J. D. 1995. Lock-free linked lists using compare-and-swap. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*: 165-172.
22. Vogels, W. 2009. Eventually consistent. *Communications of the ACM* 52: 40-44.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

PAUL E. MCKENNEY is a Distinguished Engineer in IBM's Linux Technology Center, where he maintains the Linux kernel's RCU implementation and contributes to userspace RCU. Prior to that, he worked on Sequent's DYNIX/ptx kernel. He received his Ph.D. in 2004 from Oregon Health and Sciences University in computer science and engineering. He received his master's in 1988 from Oregon State University in computer science and a pair of bachelor's degrees from Oregon State University in computer science and mechanical engineering. He blogs at http://paulmck.livejournal.com/ and works on the book entitled "Is Parallel Programming Hard, and, If So, What Can You Do About It?" at http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html.