Hazard Pointers for the Linux Kernel?

Paul E. McKenney
April 27, 2023
Updated January 10, 2024
Updated May 22, 2024

This document describes a possible design and implementation for hazard pointers in the Linux kernel. But first, a recap of implementation issues and an introduction to hazard pointers.

TL; DR: Issues

The biggest design issue surrounds module unloading. With RCU, a single `rcu_barrier()` call initiated after the last `call_rcu()` involving an in-module function suffices. This approach relies on the fact that RCU read-side critical sections are sharply bounded, witness RCU CPU stall warnings. (Not needed in Neeraj's use case.)

But one of the advantages of hazard pointers is the ability to hold a long-term reference without blocking reclamation of other hazard-pointer-protected objects. This means that a straightforward `hazptr_barrier()` that waits on all pre-existing hazard-pointer callbacks might never return. For more detail, please see the discussion of `hazptr_barrier()` below. (Neeraj's use case does not invoke `rcu_barrier()`, and so presumably would not need `hazptr_barrier()`.)

Another design issue surrounds diagnostics. The equivalent of an RCU CPU stall warning does not make sense, and the straightforward lockdep approaches are invalidated by the fact that it is perfectly legal to pass hazard pointers from one CPU to another and from one task to another. Note that one of the main benefits of hazard pointers is the ability to hold references to individual objects indefinitely.

A use-case issue is that there are not yet any known situations in the Linux kernel where memory contention favors use of hazard pointers over any of the other applicable synchronization mechanisms. But Neeraj located one here.

Note that the hazard-pointer grace-period mechanism compares hazard pointers. This means that there must either be a unique pointer value that corresponds to a given object on the one hand or that the hazard-pointers mechanism must understand what the memory allocator is doing.

There is no need for the saturation and underflow checks that are required for classic reference counting are not needed because hazard pointers uses "stroke arithmetic", where the value of

the implicit reference counter is the number of hazard pointers currently referencing the hazard-pointer-protected object.

What Are Hazard Pointers?

Hazard pointers can be thought of as an inside-out reference counter. Instead of an integer within a data element that counts references, each reference is instead denoted by a pointer that is stored in a CPU/task-local location, each such pointer being a “hazard pointer”. And I hope that you will agree with me that freeing a data element referenced by a hazard pointer would indeed be hazardous! The reference count for a given data element is then the number of hazard pointers referencing it.

As with RCU, once a given data element has been removed from all structures (excluding the hazard pointers themselves), readers can no longer find that element, and therefore no new hazard pointer can be created. Except for race conditions where readers store a hazard pointer to a given element just as that element is being removed. Hazard pointers handles this race by:

1. Reading the pointer to the target data element.
2. Storing it to a hazard pointer.
3. Enforcing full ordering (for example, `smp_mb()`, though there are other options).
4. Re-reading the original pointer. If the values read differ, the reader clears the hazard pointer and retries from the beginning. It is often the case that “from the beginning” means going back to an immortal pointer to the entire data structure, for example, a tree traversal will usually need to restart from the head of that tree.
5. Otherwise, the continued existence of the target data element is guaranteed, so the reader can start using that element.

This results in zero writes to the shared structure on the read path and no read-modify-write atomic operations. The call to `smp_mb()` can be replaced by `sys_membarrier()`-like IPIs, but `CONFIG_PREEMPT_RT=y` builds of the Linux kernel would probably prefer the read-side `smp_mb()` calls. Alternatively, RCU could be used to protect hazard-pointer acquisition, which, in contrast to “A marriage of pointer- and epoch-based reclamation” (where epoch-based reclamation is an implementation of RCU), this might be a beautiful friendship between the two, avoiding the traditional hazard-pointer-induced read-side memory barriers and IPIs while also avoiding the traditional RCU unbounded memory footprint in face of unbounded readers or grace-period blocking.

Reference material:

1. Safe Memory Reclamation for Dynamic Lock-Free Objects Using Atomic Reads and Writes (Maged Michael’s original paper).

2. The Repeat Offender Problem: A Mechanism for Supporting Dynamic-Sized, Lock-Free Data Structures (Herlihy's, Luchangco's, and Moir's original paper). These two papers were officially declared to be in a tie. ;-)
3. Folly library hazard-pointers implementation (see files in this directory whose names are prefixed by "Hazptr").
4. Hazard Pointers for C++26 (C++ working paper, contains background and use cases).
5. Is Parallel Programming Hard, And, If So, What Can You Do About It? (Section 9.3).

Why Hazard Pointers in the Linux Kernel?

At a high level, RCU is a scalable replacement for reader-writer locking and the hazard pointers technique is a scalable replacement for reference counting. Now there are already a great number of scalable reference-counting use cases in the Linux kernel, including the following:

- "Just use kref!" This works extremely well in a great many use cases, but imposes constraints on acquiring references, such as holding a lock, already holding a reference, or, in the case of `kref_get_unless_zero()`, being within an RCU read-side critical section. Furthermore, in this RCU-protected case, you cannot acquire the first reference, which can be OK in the not-uncommon case where a reference is "held" by the enclosing data structure. And in all cases, kref has scalability limitations due to memory contention when there is a popular element.
- "Just use RCU!" The point here is that `rcu_read_lock()` can be thought of as acquiring a reference to all RCU-protected data elements, and doing so atomically with near-zero cost. This works great! That is, assuming that there is no need to block while holding a reference and that no one is going to hold a reference for very long.
- "Just use SRCU!!!". This allows both blocking while holding a reference and also long-running references. But this assumes that there is no need to distinguish between someone holding a reference to one element from someone holding a reference to some other element. As might happen if it was necessary to free part of the data structure during the course of a long-running reference.
- "Just use percpu-ref!" This refers to the `percpu_ref` structure that supports calls to `percpu_ref_get()` and friends. This can work extremely well, but a data structure comprised of many small elements might not be well-served by a per-element set of per-CPU variables. And of course using SRCU with a per-element `srcu_struct` structure suffers from this same memory-size issue.
- "Just follow rcuref.rst!" This document suggests using RCU to guarantee a data element's continued existence while acquiring a per-element reference. This works quite well, and has long been heavily used in open-coded form. However, it is often preferable to avoid the unconditional `atomic_inc()` in the read-side path in favor of a conditional primitive, in order to avoid acquiring references to things that have just now been removed. In addition, it would have scalability limitations if applied to a data structure with a popular element (such as the root of a search tree), which could potentially suffer memory contention. The usual way around such memory-contention

issues is to have RCU protect the search structure (including the popular element), and obtain the reference only of the destination element. Which works quite well. That is, unless the destination element itself happens to be the popular element.

- Finally, the patch series that prompted this document. This is similar to the aforementioned rcuref.rst approach, but does provide conditional reference acquisition. However, it has similar issues with popular elements, and to this point, one of the features of this patch series is that it avoids atomic compare-and-swap operations due to their quadratic behavior on contention. This suggests that the popular-element problem might soon be more than a theoretical issue.
- More reference-counting use cases here!

Given all of these alternatives, why would something new be required?

Running through the alternatives above, the **potential** unmet needs include:

1. Handling frequent reference acquisition from many CPUs to a single popular data element. The fact that the aforementioned patch series improved performance by avoiding cmpxchg operations hints that this unmet need might be important.
2. Allowing other data elements to be freed while a given reference is being held or acquired.

And one way of handling these needs happens to be hazard pointers. It is quite possible that the design and implementation process will identify a useful alternative, or that additional Linux-kernel use cases will appear. If and when this happens, they will be added to the appropriate lists.

At this point, the odds against something like this going in any time soon might be ten-to-one against. But that is probable enough to be worth thinking things through, especially given that an unanticipated bottleneck could change the odds at any time.

Desiderata

Because most hazard-pointer uses are in userspace libraries and research code, it is well worth laying out some of the constraints inherent in the Linux kernel:

1. CPU and memory overhead must be reasonably low.
2. It is necessary to preserve the energy-efficiency measures adopted by RCU, perhaps by sharing some of RCU's code. It might be good to consolidate the laziness timers.
3. Allocating memory on the free path should be avoided, except when used as an optimization, as in the case of `kfree_rcu_mightsleep()`.
4. Bottlenecks should be avoided, though early prototypes might have some performance and scalability limitations.

5. Should idle and offline CPUs be permitted to be hazard-pointer readers? If so, then there has to be some other mechanism to drive forward processing on behalf of these CPUs. (The preferred answer is of course “yes”.)
6. Other kernel functionality must be accounted for, including CPU hotplug (thus requiring deferred processing to be done off-CPU when needed), module unload (thus requiring an `rcu_barrier()` or similar), and `PREEMPT_RT` (thus requiring some approach not relying excessively on IPIs and preemption disabling, at least for kernels built with `CONFIG_PREEMPT_RT=y`).
7. It should not be necessary to double the number of pointers passed through APIs (the pointer to the object itself and its hazard pointer). In many use cases, the hazard pointer is manipulated within a single function so that there is no problem. In other use cases, it may be helpful to pass a reference to the hazard pointer in place of the pointer itself. However, this works better in C++ than in C due to the type erasure inherent to hazard pointers. C can handle this by creating a hazard-pointer type for each hazard-pointer-protected type, which is also not wonderful. (Full disclosure: C++ also creates duplicate types, but automatically and behind the scenes. Too bad about the increased build times…)
8. It should be possible to past call_hazptr() an in-module function, and to still be able to safely unload that module. This is harder than it might seem.
9. It would be nice for hazard-pointer acquisition to be unconditional, but this might or might not be feasible.

More desiderata will be added as needed while design and implementation progress.

Design

Identifying Data Elements

Unlike RCU, hazard pointers must know the address of the data element being protected. This can be an issue in cases where some readers maintain pointers to the interior of that data element. Here are some ways of handling this:

1. Use the address of the beginning of the element as a canonical pointer.
2. Use the address of the element’s `rcu_head` field as a canonical pointer.
3. Make the `call_hazptr()` function take a range of addresses rather than a single address.
4. Refer to the memory-management system to obtain the range of addresses.

The initial prototype will use the element's `rcu_head` field as a canonical pointer, as this permits any storage to be used (not just slab memory) and allows the `rcu_head` structure to be reused, thus in turn allowing easy reuse of RCU's real-time and energy-efficiency mechanisms. Longer term, this choice rules out a `kfree_hazptr_mightsleep()`, but one thing at a time. Perhaps longer term there will be a `hazptr_head` structure that provides space for a pointer/size pair.

Data Structures

Tracking Hazard Pointers

Hazard pointers will be allocated in cache-friendly blocks in a `hazptr_block` structure. This structure also contains an identically sized array of `hazptr_rbtree_node` structures that are used to create a hazard-pointer search structure that is used during each hazard-pointers reclamation cycle. This is not exactly memory-efficient, but it does avoid any need to allocate memory on the free path. Longer term, perhaps a hash table can be substituted for the current rbtree, thus halving the size of the `hazptr_rbtree_node` structure, but in the short term use of the rbtree sidesteps hash-table-(re)sizing issues.

Different execution contexts will need to trace hazard pointers, and a `hazptr_context` structure is provided for this purpose. This structure contains a list of `hazptr_block` structures and the head of a freelist threading through the unused hazard pointers. Perhaps each CPU would pre-allocate a `hazptr_context` structure, though this would require disabling preemption across a hazard-pointers traversal. It is easy enough to allocate per-task `hazptr_context` structures if doing so makes sense.

API

The initial hazard-pointers API members are the `hazptr_context` structure, `DEFINE_HAZPTR_CONTEXT()`, `DECLARE_HAZPTR_CONTEXT()`, `init_hazptr_context()`, `cleanup_hazptr_context()`, `hazptr_alloc()`, `hazptr_free()`, `hazptr_tryprotect()`, `hazptr_protect()`, `hazptr_swap()`, `hazptr_clear()`, `call_hazptr()`, and `hazptr_barrier()`.

`struct hazptr_context;`

This is a structure that holds a set of hazard pointers for a given context, where the context might be a CPU, a task, and so on. It may be declared statically or dynamically allocated. If it is dynamically allocated, then it must be passed to `init_hazptr_context()` before its first use, otherwise `DEFINE_HAZPTR_CONTEXT()` must be used to statically allocate it.

`DEFINE_HAZPTR_CONTEXT(name);`

Define a statically allocated `hazptr_context` structure with the specified name. The keyword `static` may be placed in front of this if desired in order to limit visibility to the current translation unit. This structure is initially empty, but may be passed to `hazptr_alloc()` in order to associate newly allocated hazard pointers with it.

`DECLARE_HAZPTR_CONTEXT(name);`

Make the specified statically allocated non-static-storage-class `hazptr_context` structure visible to other translation units.

`void init_hazptr_context(struct hazptr_context *hzcp);`

Initializes the specified `hazptr_context` structure. This structure is initially empty.

`void cleanup_hazptr_context(struct hazptr_context *hzcp);`

Frees all hazard pointers contained within the specified `hazptr_context` structure. If this structure was dynamically allocated, it may now be freed. If it was statically allocated, it is now back in the empty state and it may be passed to `hazptr_alloc()` in order to start using it again.

Note that `cleanup_hazptr_context()` will complain bitterly if any of the hazard pointers that it contains are still in use.

`hazptr_t *hazptr_alloc(struct hazptr_context *hzcp);`

Returns a pointer to a new `hazptr_t` or `NULL` if memory is exhausted. Note that the specified `hazptr_context` structure must already have been initialized, either via `DEFINE_HAZPTR_CONTEXT()` or `init_hazptr_context()`.

`void hazptr_free(struct hazptr_context *hzcp, hazptr_t *hzp);`

Frees a hazard pointer that was previously returned from `hazptr_alloc()`.

`bool hazptr_tryprotect(hazptr_t *hzp, T **p, field);`

Uses the hazard pointer referenced by `hzp` to protect the hazard-pointer-protected structure referenced by `*p`, where `field` is the offset of the `rcu_head` structure within the enclosing hazard-pointer-protected structure. If the structure referenced by `*p` is removed in the meantime, this clears the hazard pointer and returns `false`. Otherwise, the structure is now under hazard-pointer protection and this function returns `true`.

`void hazptr_protect(hazptr_t *hzp, T **p, field);`

Uses the hazard pointer referenced by `hzp` to protect the hazard-pointer-protected structure referenced by `*p`, where `field` is the offset of the `rcu_head` structure within the enclosing hazard-pointer-protected structure. The caller is responsible for ensuring that the pointer referenced by `p` cannot be removed, for example, `p` might reference a linked-list header and `*p` might reference that list's first element.

`void hazptr_swap(hazptr_t *hzp1, hazptr_t *hzp2);`

Swaps the roles of the two hazard pointers. This is useful when doing hand-over-hand traversal of a linked data structure.

`void hazptr_clear(hazptr_t *hzp);`

Arranges for the specified hazard pointer to no longer be protecting anything. Note that `hazptr_clear()` is a no-op when used immediately before a call to either `hazptr_tryprotect()` or `hazptr_protect()`. Why not just pass a `NULL` pointer to `hazptr_protect()`? Diagnostics, my friend, diagnostics!

`void call_hazptr(struct rcu_head *head, rcu_callback_t func);`

Passes the specified `rcu_head` pointer to the specified function some time after there are no longer any hazard pointers protecting the enclosing structure.

`void hazptr_barrier(void);` (Questionable)

Waits until all pre-existing `call_hazptr()` callbacks have been invoked. This is useful for the same reasons that rcu_barrier() is useful, for example, at module teardown time. One possible issue with `hazptr_barrier()` is that people are likely to be happy to hold hazard pointers for extended periods of time. One hope is that people refrain from long-term references to elements that have been removed from their respective data structures.

It might be necessary to do something special for `hazptr_barrier()`, for example:

- There might need to be a `call_hazptr()` variant that checks to see if there are any outstanding callbacks for a specified callback function. More thought is required here, especially given how easy it is to add another `call_hazptr()` and to forget to add the corresponding `hazptr_barrier()`. But this is the approach that could be made to work quickly if hazard pointers was needed on an emergency basis.
- There might need to be a `call_hazptr()` for the specific case of module unloading (as opposed to things like filesystem unmount).
    - This function might take a range of text addresses indicating the group of functions that are to be unloaded with the to-be-unloaded module. However, this

approach might be complicated by use of common callback functions that are outside of the module, but which use data within the module (which might be referenced by the object being deferred-freed)
    - This function might take some identifier indicating which module is being unloaded, so that it waits for all callback functions from that module.
    - Each in-module call to `call_hazptr()` might record the function being invoked, then make a variant of `hazptr_barrier()` that waits for all callbacks having that function. Doing this at build time has problems with function pointers. Doing this either at build time or run time can wait unnecessarily on functions that are in the main kernel. Attempts to distinguish between functions in relevant modules and in the main kernel might have problems with deciding whether the data involved survives module unloading.
    - This function might take a timeout.
    - There might be a single-shot polling version of this function.
    - All of these variants might have difficulty with `DECLARE_STATIC_CALL()` and similar things.
    - All of these variants might have difficulty with modules that force-load other modules (as rcutorture does with torture).
- It might be necessary to allow users to declare groups of `call_hazptr()` callback functions and to have a `hazptr_barrier()` variant that waits for a specified group.

Another alternative is to forbid use of `hazptr_barrier()` from module code. However, given that there is no reasonable implementation of `synchronize_hazptr()`, this alternative for all intents and purposes forbids the use of hazard pointers from code that can be built as a module, including things like rcutorture. For example, any hazard-pointer torture tests would need to be built into the kernel, not running as modules.

Sample Use Case

This section was intended to look at the read-mostly set implementation quoted by Hazard Pointers for C++26, implemented using hazard pointers, Thomas Gleixner's RCU-protected reference-count-acquisition patch, and pure RCU. However, that example as shown would simply demonstrate RCU's ease of use, and we need an example that plays more clearly to hazard-pointers strengths. This section needs a better example, perhaps one where the element is returned rather than an indication of presence or absence.

Please also see Neeraj's Linux-kernel AppArmor example, which he posted to LKML here, and includes analysis of alternative concurrency designs. This use case requires frequent reference acquisition on a Nginx AA policy centralized reference count. This encounters memory contention on 96-CPU systems, and systems with 512 CPUs are becoming (relatively) cheaply and readily available. Neeraj's patch series makes careful use of per-CPU reference counting, but might benefit further from hazard pointers.

At the 2024 LSF/MM summit, a number of developers and maintainers expressed a strong interest in hazard pointers, so it is time to get this going. Boqun Feng released a prototype at branch boqun-dev/hazptr of the shiny new (as of May 2024) shared RCU git tree.

Execution Flows

Hazard-Pointers Reclamation Cycle

This is to be driven by a dedicated kthread, similar to RCU's `rcu_gp_kthread()`.

1. Start a hazard-pointers reclamation cycle.
2. Collect callbacks from each CPU. (Does this need to be a separate pass from collecting hazard pointers? Seems like migration could make this necessary. But could do it with the hazard-pointer collection phase if the callbacks collected are for the next cycle rather than the current one.)
3. Collect hazard pointers from each CPU.
4. Use IPIs if needed to accelerate collection. (Can use locking and remote access, especially in `PREEMPT_RT` kernels.) Or just do the collection sequentially, at least in the initial prototype.
5. Build search structure (rbtree? If so, `include/linux/rbtree_types.h` and `include/linux/rbtree.h`). This step will initially be sequential, but it might need to be parallelized later.
6. Check collected callbacks against search structure. If found, put back in the waiting list, otherwise in the done list, with the done list being handed off to RCU. This can be done concurrently at each CPU. (New `call_hazptr()` invocations would add to a third list in order to bound lock contention.)
7. Wait for all checking to to complete, using IPIs if needed to accelerate checking. (Can use locking and remote access, perhaps preferentially for small numbers of callbacks, but especially for `PREEMPT_RT` kernels.)
8. End hazard-pointers reclamation cycle.

This procedure should be initiated based on time (but only if there are callbacks queued and modulated by laziness), based on memory-shrinker needs (but only if there are callbacks queued), and based on number of callbacks queued (similar to RCU's `call_rcu()` heuristics).

For best results, in the case of laziness, the hazard-pointers reclamation cycle should be commenced at the same time as is the RCU grace period. The point is to avoid unnecessary wakeups.

Queueing a Hazard-Pointers Callback

The initial prototype will go directly to per-CPU callback queuing, given the generally short-lived nature of global callback queuing in the various flavors of RCU.

Each per-CPU pool will have three lists of callbacks, presumably re-using the unsegmented `rcu_cblist`:

1. For newly queued callbacks, protected by its own lock in order to reduce contention during callback floods.
2. For callbacks that have been seen by the hazard-pointers reclamation cycle, but which still had hazard pointers referencing them.
3. For callbacks that are ready to invoke. In non-lazy configurations, this list might be unused in favor of the `RCU_DONE_TAIL` segment of the CPU's `rcu_data` structure's `->cblist`.

If need be, an additional lock would protect the last two lists.

Avoiding Hazard-Pointer Acquisition Failure and Memory Barriers

Hazard pointers has traditionally had several disadvantages:

1. Hazard-pointer acquisition failure.
2. Read-side ordering, which has traditionally been provided either by read-side full memory barriers or by real-time-unfriendly update-side IPIs interrupting readers.

Hazard-Pointer Acquisition Failure

To see the traditional need for hazard-pointer acquisition failure in non-trivial data structures, consider the following linked list:

A -> B -> C -> D

Suppose that a reader holds a hazard pointer referencing B, which is then removed. B's pointer to next might still reference C, but suppose that prior to that reader advancing to C, C is also removed. Because there is no hazard pointer referencing C, it may be freed immediately. But this means that B's pointer to next can no longer be traversed, because it now references the freelist. For this reason, hazard-pointer traversals are subject to failure, so that the reader in this case must abandon its traversal, perhaps retrying it from the beginning. For an example, please see Section 3.3 of Hazard Pointers for C++26.

Hazard-pointer acquisition failure can be avoided using link counts, as demonstrated by Folly-library usage. But without these link counts, another trick is to protect the load from the to-be-traversed pointer and the store to the hazard pointer in an RCU read-side critical section. The hazard-pointer update code can then gather the retired lists, wait for an RCU grace period, and then pick up the hazard pointers. This approach frees the hazard-pointer reader from the need to reload the traversed pointer, along with the need to double-check it against the stored hazard pointer.

This works straightforwardly for trivial linked data structures that have at most one element linked from an immortal header pointer. More involved linked data structures can use a combination of link counts and RCU. @@@ Check @@@

Read-Side Ordering

@@@ RCU to avoid need for read-side memory barriers. @@@

@@@ Use hazard pointer instead of link count when cleaning up. Traversals gain hazard pointer only on head element or some other proxy for the entire structure. Destructor drops hazard pointer? Doesn't help if destructor isn't invoked until deleter time… @@@

Execution Contexts

Make `rcu_core()` call out to hazard-pointer work. This likely also requires similar call outs from `rcu_sched_clock_irq()`, if nothing else, to ensure that `rcu_core()` runs when hazard pointers needs it to.

Hazard-pointer callbacks (`rcu_head` structures) invoked by current RCU mechanisms by feeding these callbacks into the appropriate `RCU_DONE_TAIL` segments. Carefully, so as to avoid energy-efficiency issues.

Validation

Given the recent unearthing of an embarrassing day-one bug in perfbook's hazard-pointers implementation, something more aggressive will be needed, even in prototype stage.

Torture Testing

As with rcutorture, the testing must recycle elements of an array to avoid compiler interference due to lifetime-end pointer-zap issues. Specific tests must include:

- Demonstrating that holding a hazard pointer on one element protects that element.
- Demonstrating that holding multiple hazard pointers on multiple elements protects them all.
- Demonstrating that elements not referenced by a hazard pointer can be reclaimed.
- Demonstrating that elements not referenced by a hazard pointer can be reclaimed, even when hazard pointer is held on some other element throughout.
- Demonstrating that hazard pointers function properly in the face of concurrent CPU-hotplug operations.
- Demonstrating use of hazard pointers from idle and offline CPUs (most likely pro-forma testing).
- Should it prove feasible, demonstrating `hazptr_barrier()` functionality.
- Demonstrating emergency reclamation in response to (for example) memory shrinkers.

- Demonstrating that lazy hazard-pointer callbacks do not result in needless wakeups. (To be fair, this testing is still TBD for RCU.)
- Measuring the performance and scalability of hazard-pointer acquisition and release, both in real-time mode (`smp_mb()` in read-side code) and otherwise (IPIs in update-side code).
- Measuring the performance and scalability of hazard-pointer reclamation.

Use-Case Diagnostics

One use-case bug is failure to release a hazard pointer.

The equivalent of an RCU CPU stall warning does not make sense because, as with SRCU, there are perfectly reasonable use cases that hold a hazard pointer for arbitrary lengths of time. In addition, the straightforward lockdep approaches are invalidated by the fact that it is perfectly legal to pass hazard pointers from one CPU to another and from one task to another.

If a suspected use-case bug is reproducible, one approach is to attach a BPF program to the hazard-pointer acquisition code and to keep a stack trace in a BPF map for each hazard pointer.

Another use-case bug is premature reuse of a hazard pointer. This ends protection of the object referenced by the previous use of that hazard pointer, with the resulting use-after-free issues. One way to catch at least some of these bugs would be to require that hazard pointers be cleared using `hazptr_clear()` before being reused. Another might be to add a `hazptr_check()` function that verifies that the hazard pointer still contains the desired pointer.

Interestingly enough, one potential advantage of hazard pointers over in-object reference counting is improved diagnostic, but with emphasis on “potential”.

LKMM Modeling

What has to happen for LKMM to be useful to hazard-pointers users and implementers?

Modeling Uses of Hazard Pointers

Here, the user models their hazard-pointer uses as if these hazard pointers were normal reference counters. The reason this works is because LKMM doesn’t know about either C-language structures or memory allocation, and LKMM can therefore safely model a reference counter within the structure.

Modeling Implementations of Hazard Pointers

The implementer must carefully partition the implementation into pieces that LKMM’s tooling can handle, just as with any other non-trivial synchronization primitive.

Optimizations

- Invoke callbacks directly from main kthread if there are not very many of them.
- Have a time delay between consecutive hazard-pointer scans unless there are lots of not-yet-scanned callbacks or we are low on memory.
- Use a combining tree for requests to scan callbacks, similar to starting RCU grace periods.
- Single list of callbacks vs. multiple queues. (As with Tasks RCU, perhaps best to start with a single list.)