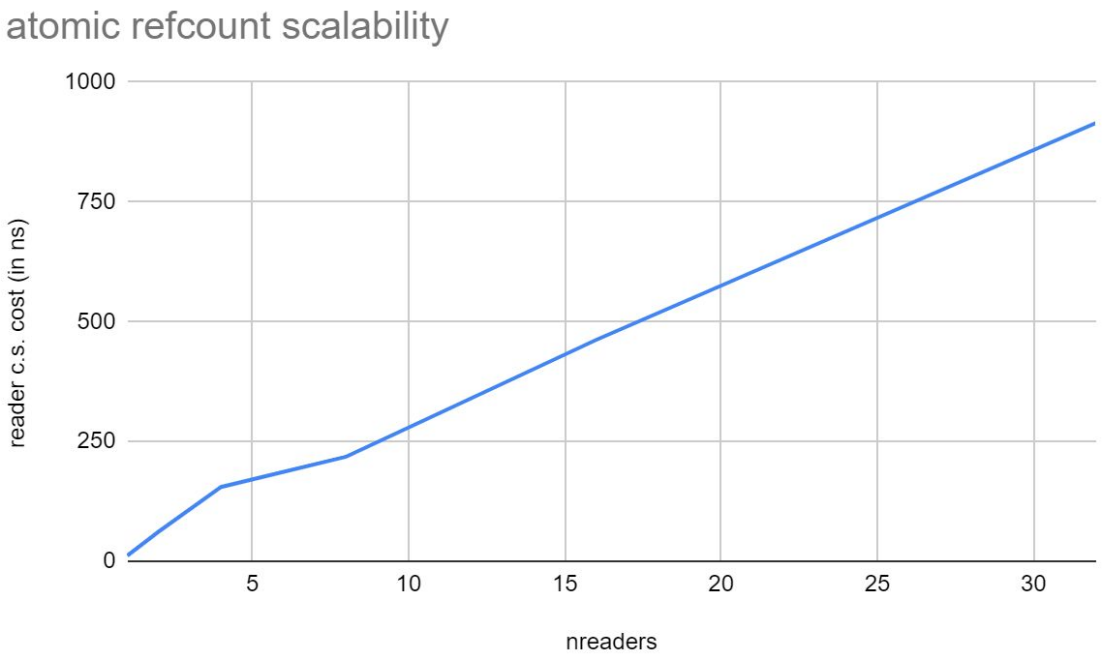
Hazard pointers in Linux kernel

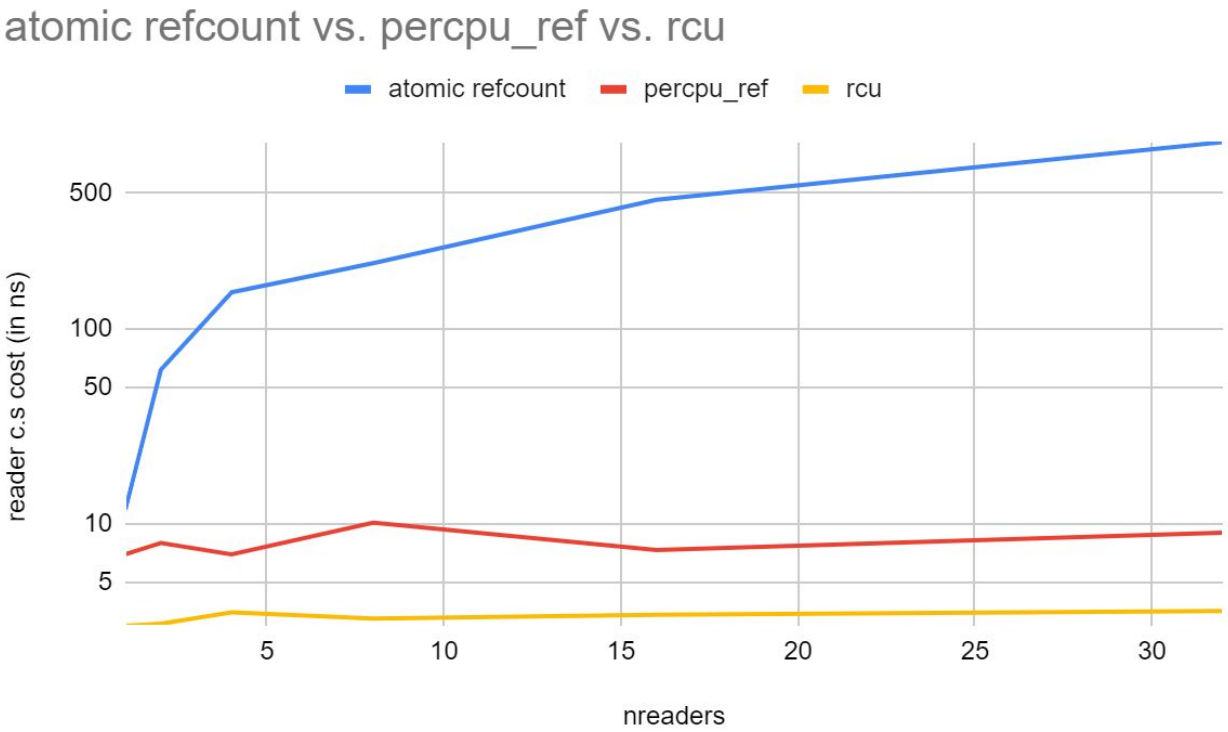
Boqun Feng 冯博群 (Microsoft)
Neeraj Upadhyay (AMD)
Paul McKenney (Meta)

Scalability of atomic refcount

- refscale (preempt)
  - scale_type=refcnt
- Intel(R) Xeon(R) Platinum 8260 CPU @ 2.40GHz
  - 48 vcpus



Per-CPU refcount and RCU



50,000-Foot Level Reference Counting vs. RCU

Property	Reference Counting	Per-CPU Ref Count	RCU
Readers	Slow & unscalable	Fast and scalable	Fast and scalable
Memory Overhead	*O(Nobj)*	*O(Nobj*Ncpu)*	*O(Nobj*)*
Protection Duration	Can be long	Can be long	Bounded duration
Traversal Retries	If any object deleted	If any object deleted	Never
Deferred Memory	None	Switch to global	Can be large

*: assume that Nobj > Ncpu/Ntask and rcu_head is used.

50,000-Foot Level Reference Counting vs. RCU vs. ???

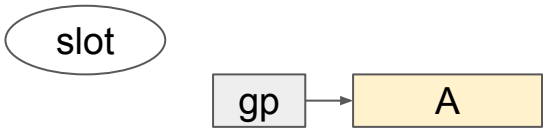
Property	Reference Counting	Per-CPU Ref Count	RCU	???
Readers	Slow & unscalable	Fast and scalable	Fast and scalable	Fast and scalable
Memory Overhead	*O(Nobj)*	*O(Nobj*Ncpu)*	*O(Nobj)*	*~O(Nobj)*
Protection Duration	Can be long	Can be long	Bounded duration	Can be long
Traversal Retries	If any object deleted	If any object deleted	Never	If any object deleted
Deferred Memory	None	Switch to global	Can be large	???

Hazard pointers

- Introduced at 2004:
  - M. M. Michael, "Hazard pointers: safe memory reclamation for lock-free objects," in IEEE Transactions on Parallel and Distributed Systems, vol. 15, no. 6, pp. 491-504, June 2004
- C++ standard library
  - https://www.open-std.org/jtc1/sc22/wg21/docs/papers/2024/n4981.pdf
- Other userspace libraries:
  - https://github.com/facebook/folly/blob/main/folly/synchronization/Hazptr.h
  - https://github.com/jonhoo/haphazard

Hazard pointers (slot)

- Hazard pointer slots
  - Can store one pointer value
  - Allocated by each reader
  - Updaters can access all the slots



Hazard pointers (reader acquire)

```
1: tmp1 = READ_ONCE(gp); // snapshot the global pointer value
2: WRITE_ONCE(*slot, tmp1); // store the value into a hazard pointer slot
3: smp_mb();
4: tmp2 = READ_ONCE(gp);  // re-snapshot the global pointer value
5: if (tmp1 == tmp2) {
     return tmp2;
} else {
5b:  WRITE_ONCE(*slot, NULL); // reset hazard pointer slot
     <continue step 1 or abort>
}
```

Hazard pointers (reader release)

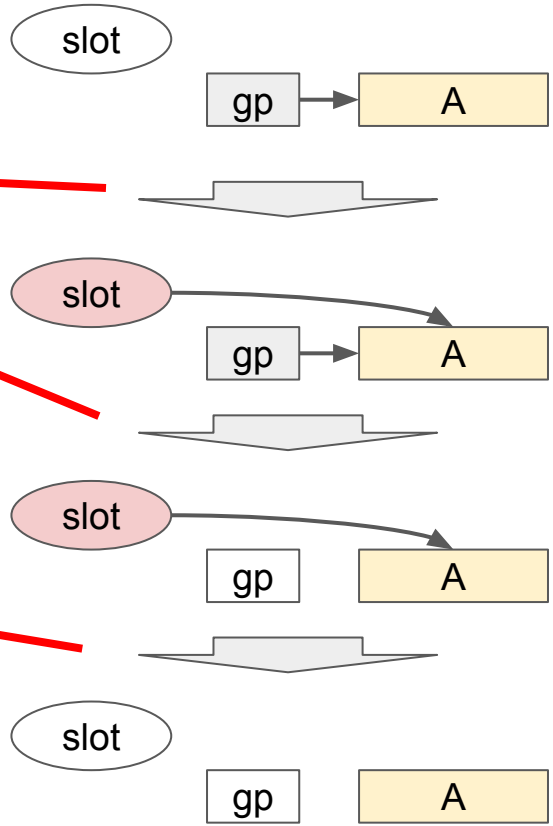
```
1: smp_store_release(slot, NULL);
```

Hazard pointers (updater)

```
1: todo = READ_ONCE(gp);
2: WRITE_ONCE(gp, NULL); // unpublish todo
3: smp_mb();
4: ptr = READ_ONCE(*slot); // fetch the pointer that a hazard pointer is protecting
5: if (ptr == todo) {
     <continue to step 4>
} else {
     <check more slots>
}
```

Hazard pointers (synchronization case #1)

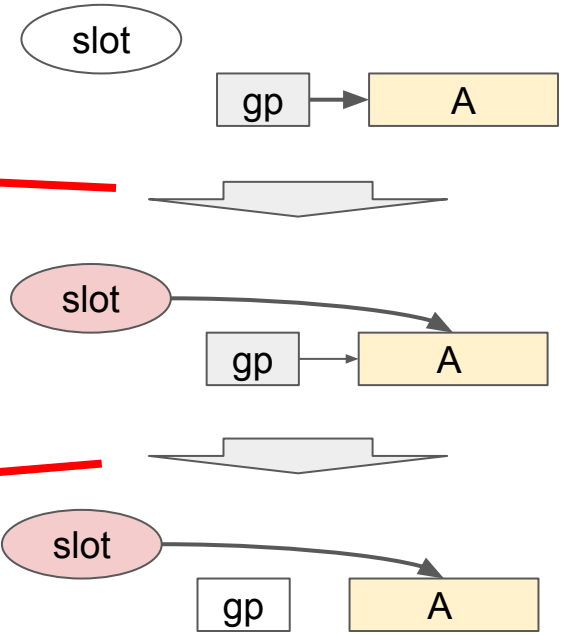
```
<reader>                          <updater>
1: tmp1 = READ_ONCE(gp);
2: WRITE_ONCE(*slot, tmp1);

                                  2: WRITE_ONCE(gp, NULL);
                                  ...

3: smp_mb();
4: tmp2 = READ_ONCE(gp);
5: if (tmp1 == tmp2) { // false, reader will try again
} else {
5b:  WRITE_ONCE(*slot, NULL);


                                  4: ptr = READ_ONCE(*slot);

}
```



Hazard pointers (synchronization case #2)

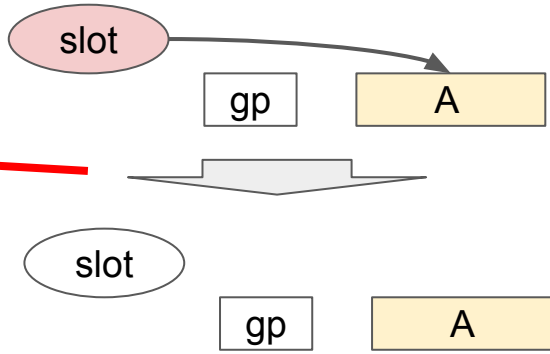
```
<reader>
1: tmp1 = READ_ONCE(gp);
2: WRITE_ONCE(*slot, tmp1);
3: smp_mb();
4: tmp2 = READ_ONCE(gp);
5: (tmp1 == tmp2) {
     return tmp2;


}
```

```
<updater>


1: todo = READ_ONCE(gp);
2: WRITE_ONCE(gp, NULL);
3: smp_mb();
4: ptr = READ_ONCE(*slot);
5: if (ptr == todo) {
     <continue to step 4>
}
```



Hazard pointers (synchronization case #3)

```
<reader>                    <updater>

smp_store_release(slot, NULL);
                            4: ptr = smp_load_acquire(*slot);
                            5: if (ptr == todo){
                                  ...
                            } else {
                                 <clean up ptr if it's the last slot>.
                            }
```



Hazard pointers

- Summary
  - Users of hazard pointers need to allocate hazard pointer slots before protection - users bring their own counter
  - One hazard pointer can protect different objects
  - Updaters of hazard-pointer-protected objects can check the readers of a particular object.

100,000-Foot-Level Hazard Pointers vs. RCU

RCU can be thought of as a fast and scalable replacement for reader-writer locking

Hazard pointers can be thought of as a fast and scalable replacement for reference counting

There is significant overlap in the use cases for hazard pointers and RCU

50,000-Foot Level Reference Counting, RCU, & Hazptr

Property	Reference Counting	Per-CPU Ref Count	RCU	Hazard Pointer
Readers	Slow & unscalable	Fast & scalable	Fast & scalable	Fast & scalable
Memory Overhead	*O(Nobj)*	*O(Nobj*Ncpu)*	*O(Nobj)*	*~O(Nobj)*
Protection Duration	Can be long	Can be long	Bounded duration	Can be long
Traversal Retries	If any object deleted	If any object deleted	Never	If any object deleted
Deferred Memory	None	Switch to global	Can be large	Depends on scan interval

Implementation of hazptr

- hazptr_context: how hazard pointer slots are maintained.
- reader scan: how updaters scan the slots to readers of a particular object.

hazptr_context

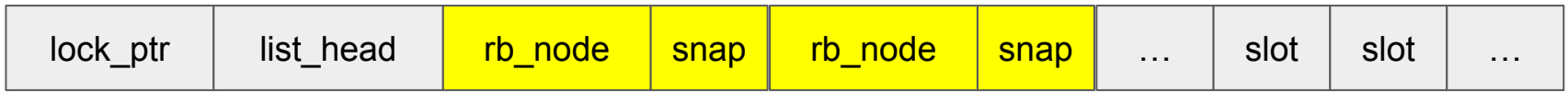
- Hazard pointer slots: “Allocated by each reader” and “Updaters can access all the slots”
- Provides fixed amount of slots (for allocation).
- Each context can add any number of hazptr_context into a global list (for reader queries from updaters).

Hazptr basic usage

```
<initialization>
init_hazptr_context(ctx);
hazptr_t *hptr = hazptr_alloc(ctx); // readers allocate the hazard pointer ahead of time

<reader>
if (p = hazptr_tryprotect(hptr, gp /* a global pointer */, ..)) {
      // p is valid reference to gp, until hazptr_clear(hptr).
      hazptr_clear(hptr);
}

<updater>
ptr = READ_ONCE(gp); // another synchronization between different updaters.
WRITE_ONCE(gp, NULL);
call_hazptr(&ptr->hazptr_head, func);
```

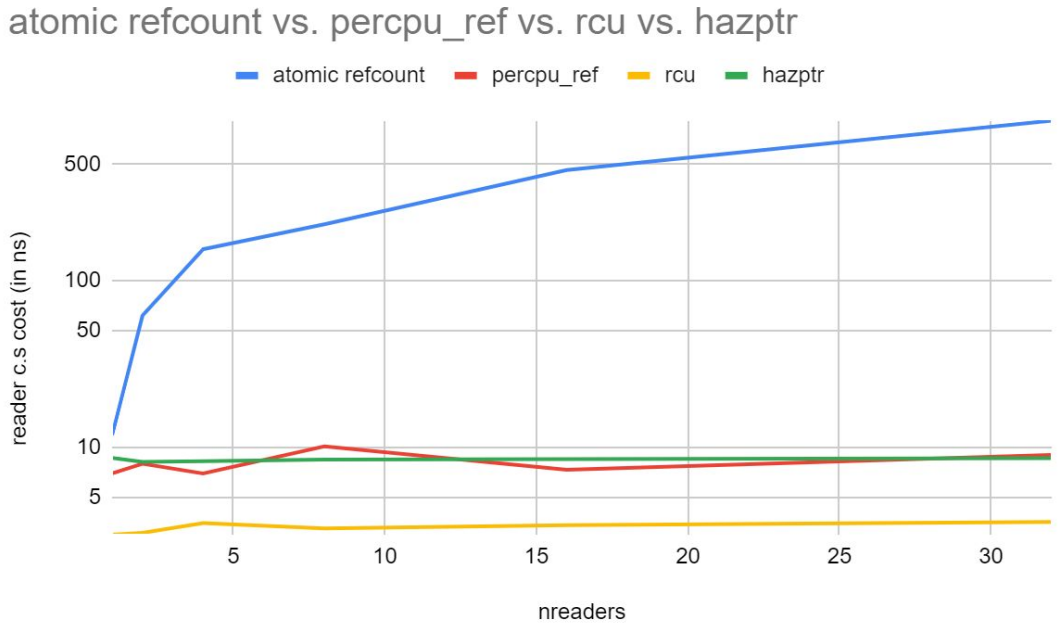
Reader scan

- If there are M objects to be freed and N active slots, the time complexity of a full scan would be M x N.
- Some user-space implementation uses BTree or other search trees to store the scan result of all the slots.
  - But it’s an “allocate memory to free memory” situation.
- In the current implementation, a rbtree node is allocated ahead of the time to avoid allocating memory in hazptr callback handling.
  - Because readers can change the slot at any time, updaters need to store the snapshot value of a slot into the rbtree.
  - Readers don’t touch the rbtree (unless it’s a context removal).



Status

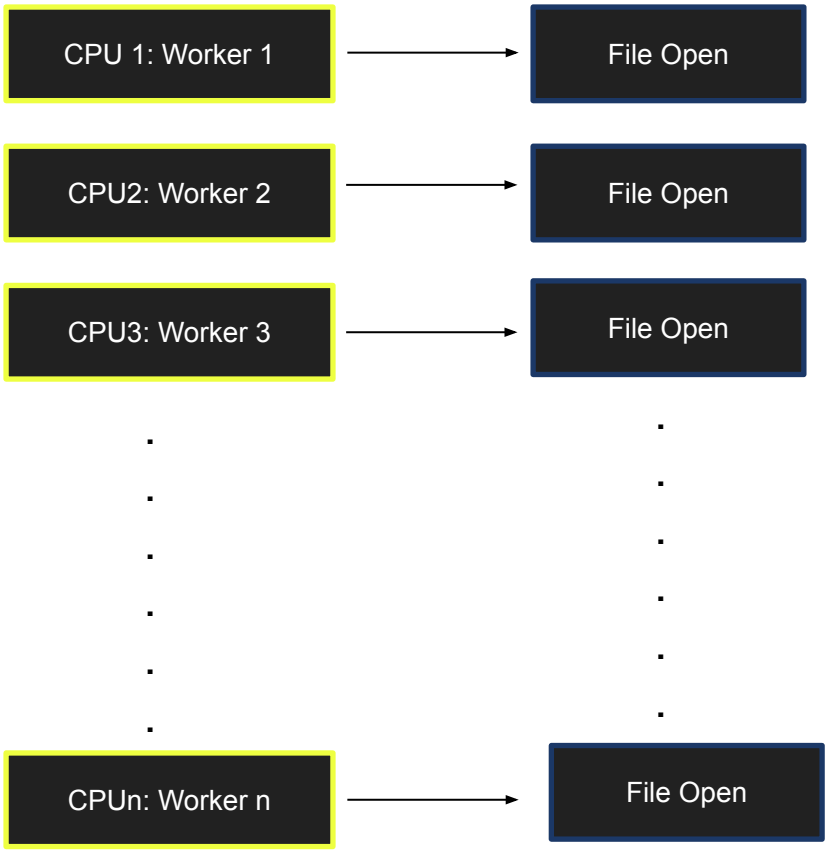
- https://lore.kernel.org/lkml/20240917143402.930114-1-boqun.feng@gmail.com/
- Memory overhead
  - O(Nobj) due to hazptr_head
  - O(NTask * Nref-per-task)



Case Study

- Nginx Workers Scaling Issue with Apparmor

Nginx Workers Scaling Issue with Apparmor




```
nginx-12151   [006] ...1. 436426.250125: <stack trace>
=> kretprobe_trace_func
=> kretprobe_dispatcher
=> kretprobe_rethook_handler
=> rethook_trampoline_handler
=> arch_rethook_trampoline_callback
=> arch_rethook_trampoline
=> security_file_open
=> do_dentry_open
=> path_openat
=> do_filp_open
=> do_sys_openat2
=> __x64_sys_openat
=> do_syscall_64
```

Use Hazard Pointers for apparmor_file_open()

```
static int apparmor_file_open(struct file *file)
{
...
+       struct hazptr_context ctx;
+       hazptr_t *hptr;

...
-       label = aa_get_newest_cred_label(file->f_cred);
+       init_hazptr_context(&ctx);
+       hptr = hazptr_alloc(&ctx);
+       label = aa_get_newest_cred_label_hazptr(file->f_cred, hptr);
...
-       aa_put_label (label);

+       hazptr_clear(hptr);
+       cleanup_hazptr_context(&ctx);
        return error;
 }
```

Label Acquire Path

```
static inline struct aa_label *aa_get_newest_cred_label_hazptr(const struct cred *cred, hazptr_t
*hptr)
{
        return aa_get_newest_label_hazptr(aa_cred_raw_label(cred), hptr);
}


static inline struct aa_label *aa_get_newest_label_hazptr(struct aa_label *l, hazptr_t *hptr)
{
       if (!l)
               return NULL;

       if (label_is_stale(l)) {
               struct aa_label *tmp;

               …
               tmp = aa_get_label_try_hazptr(&l->proxy->label, hptr);
               …

               return tmp;
       }

       return hazptr_protect(hptr, l, rcu);
}
```

Label Acquire Path

```
static inline struct aa_label *aa_get_label_try_hazptr(struct aa_label __rcu **l,
hazptr_t *hptr)
{
	struct aa_label *c;

	do {

		c = hazptr_tryprotect(hptr, *l, rcu));
	} while (!c);

	return c;
}
```

Label Release Path

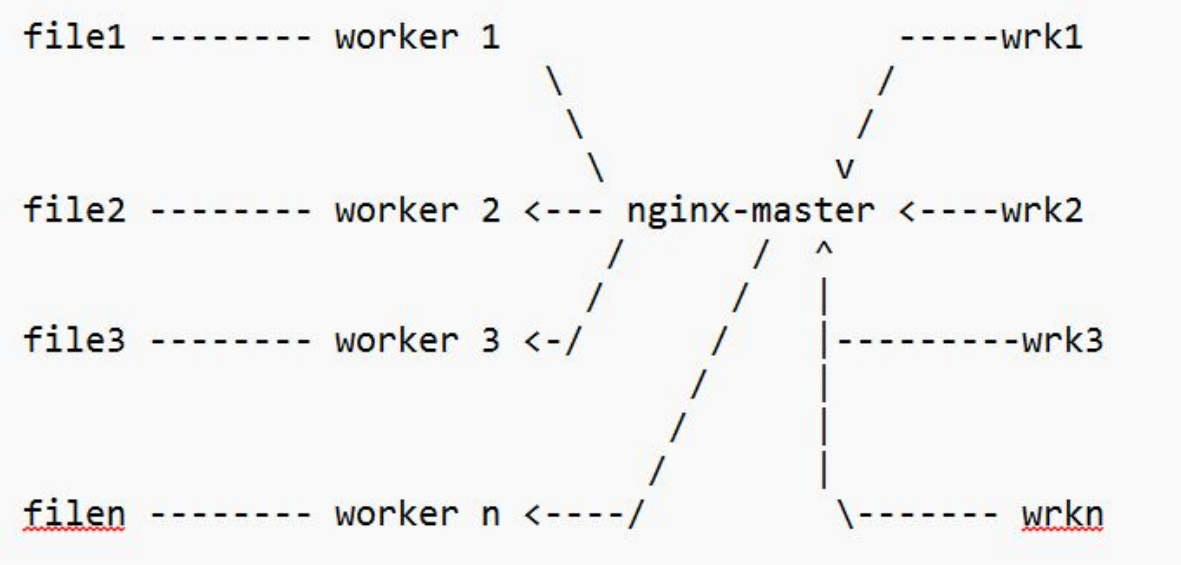
```
-void aa_label_kref(struct kref *kref)
+static void label_hazptr_func(struct callback_head *head)
 {

+       struct  aa_label *label = container_of(head, struct aa_label,
rcu);

        call_rcu(&label->rcu, label_free_rcu);
 }

+void aa_label_kref(struct kref *kref)
+{
+       struct aa_label *label = container_of(kref, struct aa_label,
count);

...
+       call_hazptr(&label->rcu, label_hazptr_func);
+}
+
```

Test Environment

- 1. Run Nginx [1] in web serving mode.
- 2. Use wrk2 load generator [2] so that 192 clients send request for 192 different files. The wrk clients run on socket 1.
- 3. There are 192 nginx workers and each worker handles request for a different client. The workers run on socket 0.
- Platform: 2 Socket 4th Generation EPYC Processor with 96 Cores, 192 threads per socket.

[1] https://nginx.org/en/download.html
[2] https://github.com/giltene/wrk2

Test Environment



Throughput (Higher is better)

Baseline	Apparmor Disabled	Hazard Pointers
Kref	+11.6%	+7.4%

COPYRIGHT AND DISCLAIMER

The information presented in this document is for informational purposes only and may contain technical inaccuracies, omissions, and typographical errors. The information contained herein is subject to change and may be rendered inaccurate releases, for many reasons, including but not limited to product and roadmap changes, component and motherboard version changes, new model and/or product differences between differing manufacturers, software changes, BIOS flashes, firmware upgrades, or the like. Any computer system has risks of security vulnerabilities that cannot be completely prevented or mitigated. AMD assumes no obligation to update or otherwise correct or revise this information. However, AMD reserves the right to revise this information and to make changes from time to time to the content hereof without obligation of AMD to notify any person of such revisions or changes.

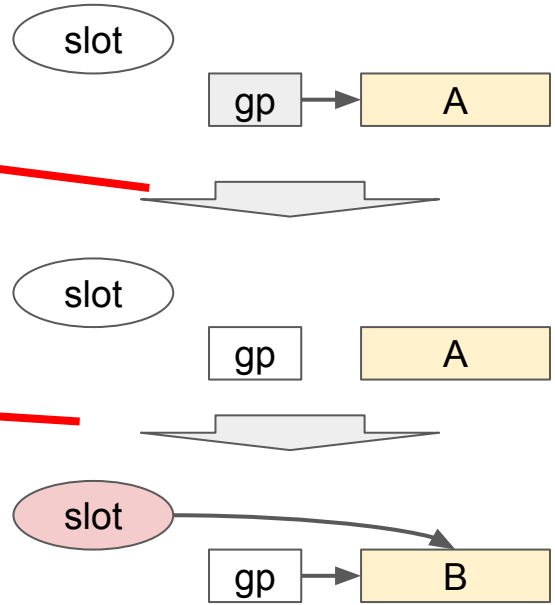
THIS INFORMATION IS PROVIDED 'AS IS." AMD MAKES NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE CONTENTS HEREOF AND ASSUMES NO RESPONSIBILITY FOR ANY INACCURACIES, ERRORS, OR OMISSIONS THAT MAY APPEAR IN THIS INFORMATION. AMD SPECIFICALLY DISCLAIMS ANY IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR ANY PARTICULAR PURPOSE. IN NO EVENT WILL AMD BE LIABLE TO ANY PERSON FOR ANY RELIANCE, DIRECT, INDIRECT, SPECIAL, OR OTHER CONSEQUENTIAL DAMAGES ARISING FROM THE USE OF ANY INFORMATION CONTAINED HEREIN, EVEN IF AMD IS EXPRESSLY ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

Q & A

Hazard pointers (synchronization case #1 ABA?)

```
<reader>
1: tmp1 = READ_ONCE(gp);


2: WRITE_ONCE(*slot, tmp1);
3: smp_mb();
4: tmp2 = READ_ONCE(gp);
5: if (tmp1 == tmp2) { // true
     <Neither A nor B accessed yet>
} else {

}
```

```
<updater>

2: WRITE_ONCE(gp, NULL);
...
4: ptr = READ_ONCE(*slot);
```



hazptr_tryprotect(slot, gp, ..)

```
tmp = READ_ONCE(gp); // fetch the global pointer value

// tmp could be freed here, which would make tmp an invalid pointer.

WRITE_ONCE(*slot, tmp); // store the value into the hazptr slot.

smp_mb(); // pairs with the reader checking at callback handling.

// At this point, hazard pointers guarantees that tmp cannot be freed.

tmp1 = READ_ONCE(gp);  // Thus tmp1 cannot become invalid until hazptr_clear()

if (tmp1 == tmp) {

     return tmp1;

} else

     return NULL;
```

What is Lifetime-End Pointer Zap?

```
// C pointers are not just pointers!

p = malloc(sizeof(*p));

do_something(p); // might free(p), and if so, p is now an invalid pointer

q = malloc(sizeof(*q));  // might have same address as p

assert(p != q); // Compiler can optimize to assert(true)

// Inequality implies that p is invalid, thus compiler's choice

// Pointer "provenance" in addition to pointer's "value bits"
```

What is Lifetime-End Pointer Zap?

// C pointers are not just pointers!

p = malloc(sizeof(*p));

do_something(p This invalidates ow an invalid pointer

q = mallo concurrent algorithms

assert(going back to the 1970s

// Inequality implies that p is invalid, thus compiler's choice

// Pointer "provenance" in addition to pointer's "value bits"

Current Lifetime-End Pointer Zap Proposals

P2434R1 (“Nondeterministic pointer provenance”)

Davis Herring’s “angelic provenance” paper on which the next two are based.

P2414R4 (“Pointer lifetime-end zap proposed solutions”)

Atomics and volatile operations erase provenance, as does usable_ptr<T> class and a `make_ptr_prospective()` function.

P3347R0 (“Invalid/Prospective Pointer Operations”)

Operations on invalid pointers must produce bit values consistent with those of the invalid pointer. Dereferencing an invalid pointer is still undefined behavior and comparison involving at least one invalid pointer is still implementation defined

Maged Michael and I started working this issue back in 2017…