Design Tradeoffs in Memory Reclamation

Pedro Ramalhete - May 2023

Table of Contents

- What is an SMR and when do we need one
- Classification and Design Choices
- A couple of SMR algorithms
- What is left to explore

Memory Reclamation is a hot topic

The latest **Dijkstra Paper Prize on Distributed Computing** went to:

Maged Michael – **H**azard **P**ointers

Maurice Herlihy
Victor Luchangco
Mark Moir

Pass **T**he **B**uck

https://en.wikipedia.org/wiki/Dijkstra_Prize

The Problem

In a multi-threaded application, different threads may hold pointers to the same object.

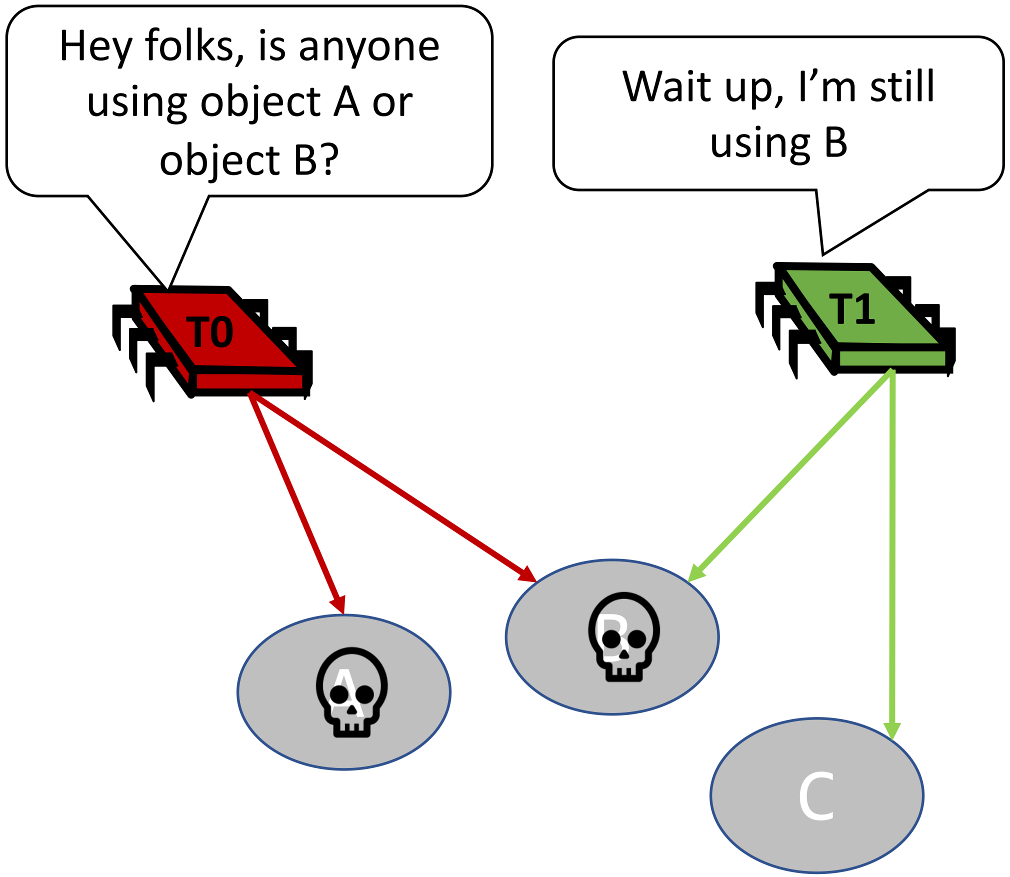
How do we know it is safe to delete/destroy/re-use an object?

What if another thread is still accessing the object?

In the example, destroying object A is ok but object B is not.

Thread 1 will have a **dangling pointer** to B and crash.

The concurrency mechanism that allows for the safe destruction of objects is named a **S**afe **M**emory **R**eclamation algorithm (**SMR**)


Hey folks, is anyone using object A or object B?
Wait up, I'm still using B
T0
T1
A
B
C

Can’t we use locks instead?

Q: What if we take a **mutual exclusion lock** when we access an object?

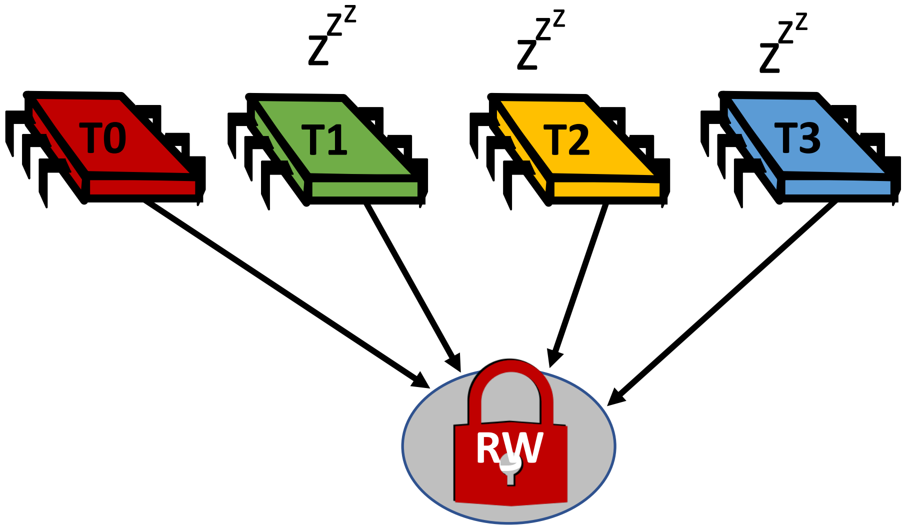
A: In many scenarios we may want multiple threads accessing the same objects but they are reading them. The **mutex becomes a bottleneck**.

Q: Ok, then use reader-writer locks and take a **read-lock**!

A: Most reader-writer locks don’t scale for contended reads. The ones that scale, use more memory.

Q: Fine, I’ll use one of the scalable reader-writer locks.

A: Sure, but what about optimistic access? And Lock-Free data structures?



Adding locks to optimistic data structures or lock-free data structures kind of defeats the purpose…

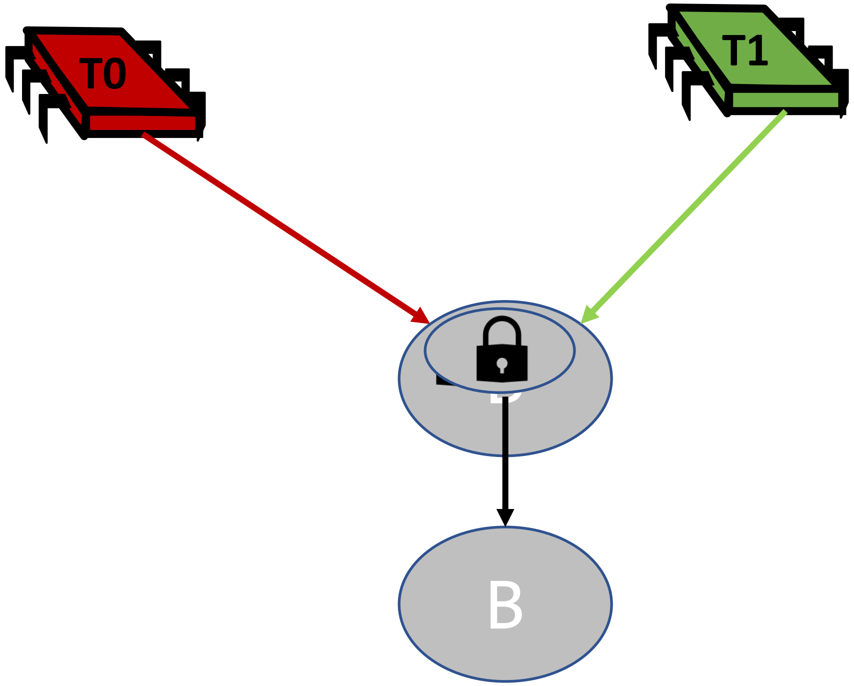
Lock-Free data structure
??

Even when using locks, you may need memory reclamation

If the **lock is inside the object**, then the destruction of the object implies the destruction of the lock.

```
struct Foo {
  pthread_mutex_t lock;
  int data1;
  long data2;
  ...
};
```

If the lock is destroyed but there is still a pointer to the object+lock in another thread, this is a problem.



If we put the lock in a separate object, the problem still exists because we will have to destroy the object where the lock is located.

Software Transactional Memory typically uses a **pre-allocated array of locks** for all objects in the application, which solves this problem.

When do we need a Safe Memory Reclamation scheme?

Executing read optimistic accesses:

- Reading object/records/nodes optimistically
- Optimistic (blocking) data structures
- Concurrency Controls with optimistic reads (OCC and related)
- Sequence locks
- **Lock-free data structures**
- …

Executing pessimistic access but the lock’s lifetime is bounded to the *shared* object, or has an unknown lifetime

- The lock that protects the object/record/node is located inside the object
- The lock is outside the object but has a dynamic lifetime
- …

In practice, most multi-threaded applications with dynamic memory allocation need a memory reclamation scheme.

Classification and Design choices

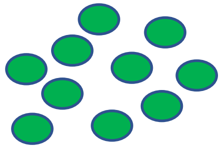
Families of Reclamation Schemes

Memory reclamation schemes can be *roughly* divided into four groups:

Pessimistic

- Quiescent-based:

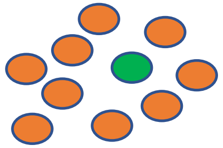
Protects **all** objects



Examples: Epoch-Based-Reclamation, Userspace-RCU, QSBR

- Pointer-based:

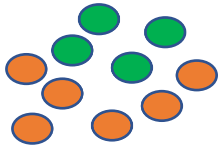
Protects a **single** object



Examples: Hazard Pointers, Pass The Buck, Pass The Pointer

- Group-based:

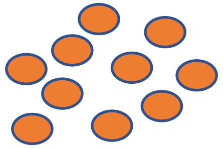
Protects a **group** of objects



Examples: Hazard Eras, Drop The Anchor, Margin Pointers

- Optimistic:

Reads **without** protection and validates after



Examples: OneFile Optimistic, Version-Based Reclamation

Some schemes combines features from multiple groups and therefore can not be placed in a single group.

API of a reclamation scheme

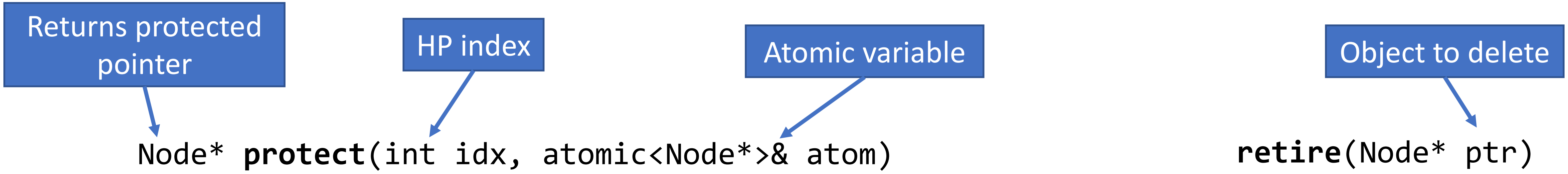
Memory reclamation schemes usually have two APIs, the `protect()` and the `retire()`.

`protect()` takes an associated index or some cookie to represent where this object is to be announced and it takes an address of an atomic variable from which the pointer is read. It returns a pointer that is protected.

`retire()` takes a pointer to an object that has been unlinked, i.e. is no longer reachable from the root pointers of the data structure.

Quiescent schemes do **not** have a `protect()`. They have an `arrive()/depart()`. After `arrive()` is called, all *reachable* objects are protected, therefore they can be safely read.

Optimistic schemes do **not** have a `protect()`. Instead, whenever a piece of data is read, they do post-validation to check whether or not that data read is valid. If it's not, we need to **restart** the operation.



Returns protected pointer

HP index

Atomic variable

Object to delete

```
Node* protect(int idx, atomic<Node*>& atom)
```

```
retire(Node* ptr)
```

Using a reclamation scheme

Quiescent-based

```
rcu_arrive();
 obj1->x++;
 obj2->x++;
 obj3->y++;
 ...
 obj99->x++;
rcu_depart();
```

- All objects protected for read
- All objects protected for write
- Must unlink before retire
- Objects are protected for the future
- Not resilient to failures

Pointer-based / Group-based

```
obj1 = protect(0,node->next);
 obj1->x++;
obj2 = protect(1,obj1->next);
 obj2->x++;
obj2 = protect(2,obj2->next);
 obj3->y++;
```

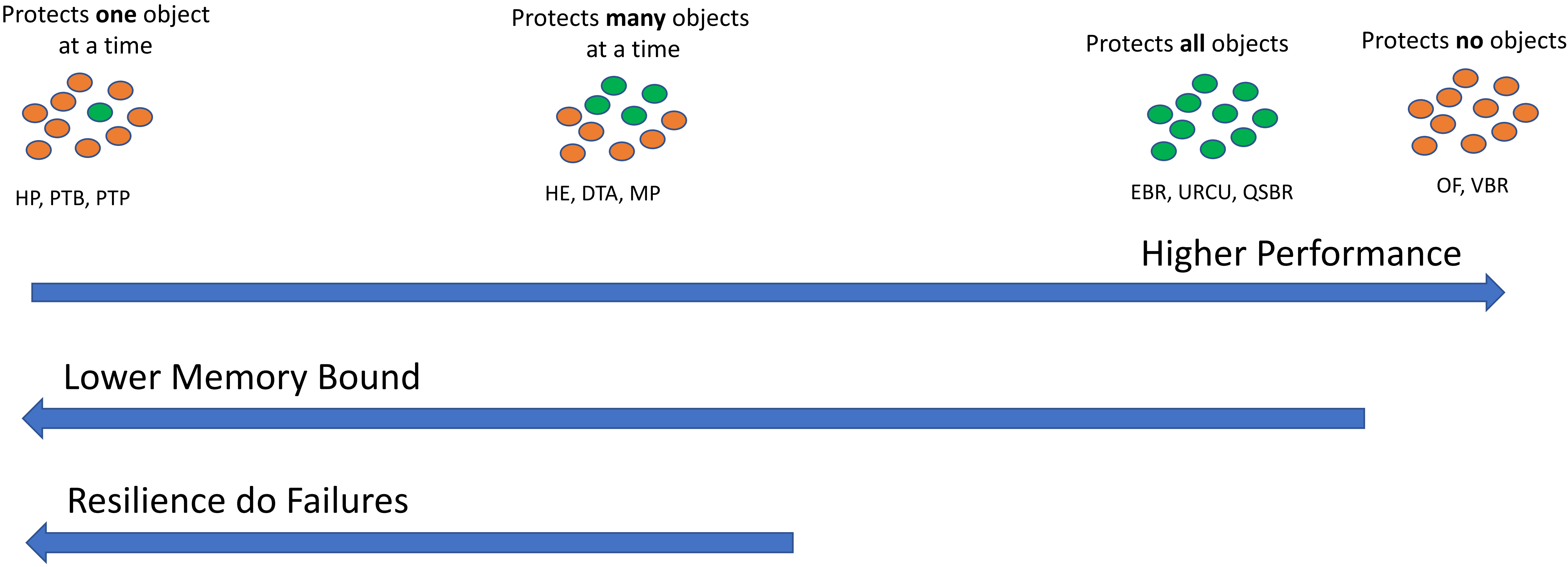
- Individual/group-of objects protected for read
- Individual/group-of objects protected for write
- Must unlink before retire

Optimistic

```
label start:
tmp1 = obj1->x;
if (!post_validate(obj1)) goto start;
tmp2 = obj2->x;
if (!post_validate(obj2)) goto start;
tmp3 = obj3->y;
if (!post_validate(obj3)) goto start;
```

- No need for unlink before free
- No protection for writes is possible
- Restarts can occur
- Can’t return memory to the OS
- Objects/pointers need some validation

Performance vs Memory (bound)



Protects **one** object at a time

HP, PTB, PTP

Protects **many** objects at a time

HE, DTA, MP

Protects **all** objects

EBR, URCU, QSBR

Protects **no** objects

OF, VBR

Higher Performance

Lower Memory Bound

Resilience do Failures

Each call to `protect()` or to `arrive()` will typically cost one store-load fence.

What makes a *good* reclamation scheme?

Five typical categories:

- **Performance**: Should have high throughput and low tail latency;
- **Bounded retired objects**: Must have a bound on the number of retired objects. The lower the better;
- **Usability**: Should be easy to deploy, with a small number of annotations/modifications to the code;
- **Applicability**: Should be deployable on as many data structures as possible;
- **Consistency**: Should not drastically change the latency profile of the workload;

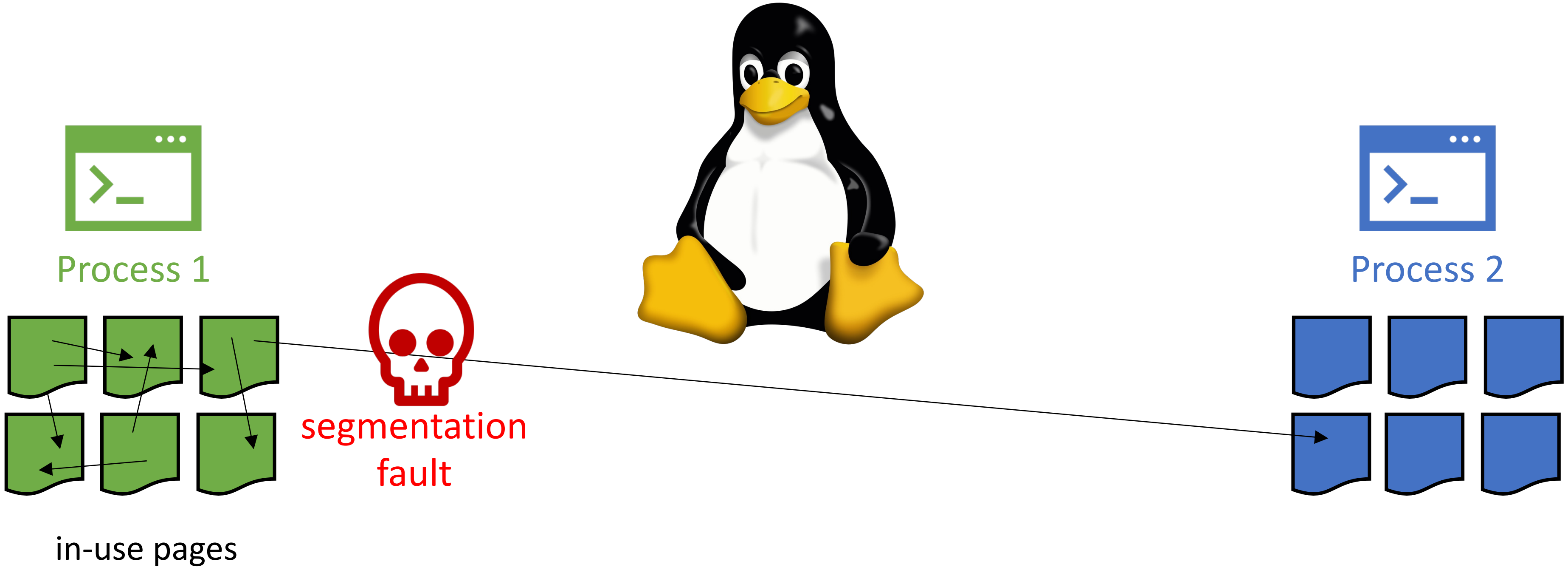
Depending on the use-case there are specific questions that may need to be considered:

Can we use this scheme with the system's allocator?
How much extra memory is taken per object due to scheme metadata?
Can I use it with any kind of objects?
Does it do restarts?
What are the performance-memory trade-offs?
What is the average memory usage?
Is it a manual scheme, partial automatic, or fully automatic?
etc…

Any Allocator *vs* Custom Allocator

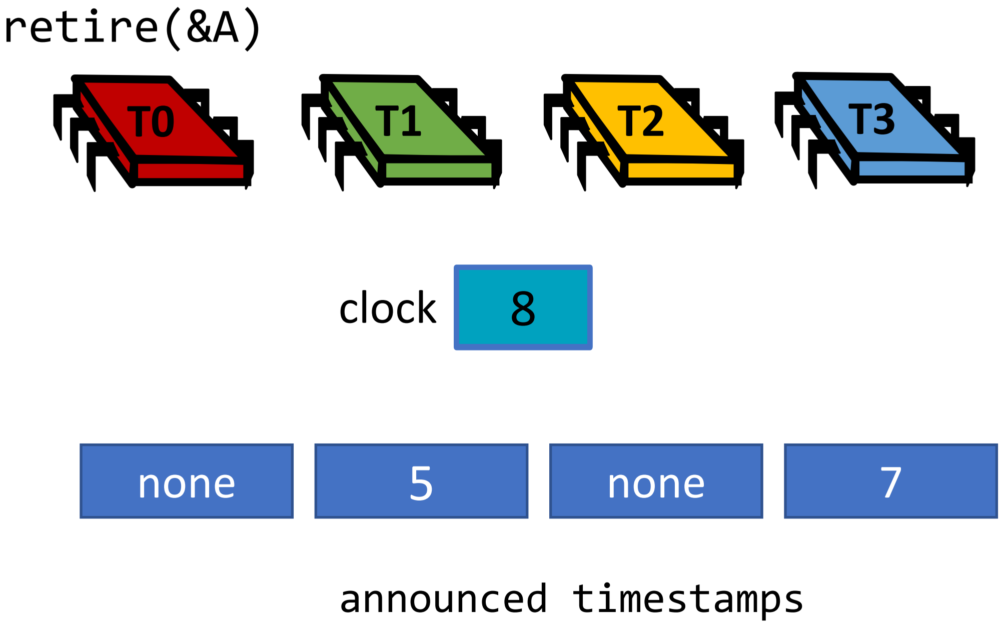
Certain schemes may need to read data from memory blocks which have been de-allocated. For example, Automatic Optimist Access, Free Access, and Version Based Reclamation require a customized allocator.

This typically means a large region has to be pre-allocated (or mmaped) from the Operating System.



Central clock Quiescence

Central Clock Quiescence



clock 8

none	5	none	7

announced timestamps

`arrive``()`:

1. Read the timestamp in the clock
2. Announce the timestamp

`depart``()`:

1. Clear the announced timestamp

`retire``(ptr)`:

1. Read the current timestamp in the clock and increment it with `fetch_add()`
2. Scan the announced timestamps and save the oldest announced timestamp
3. If the oldest > current, `free(ptr)`
4. Else, wait until it becomes true, *or* save the object in a thread-local retired list

Memory usage is **unbounded** *or* `retire()` is blocking.

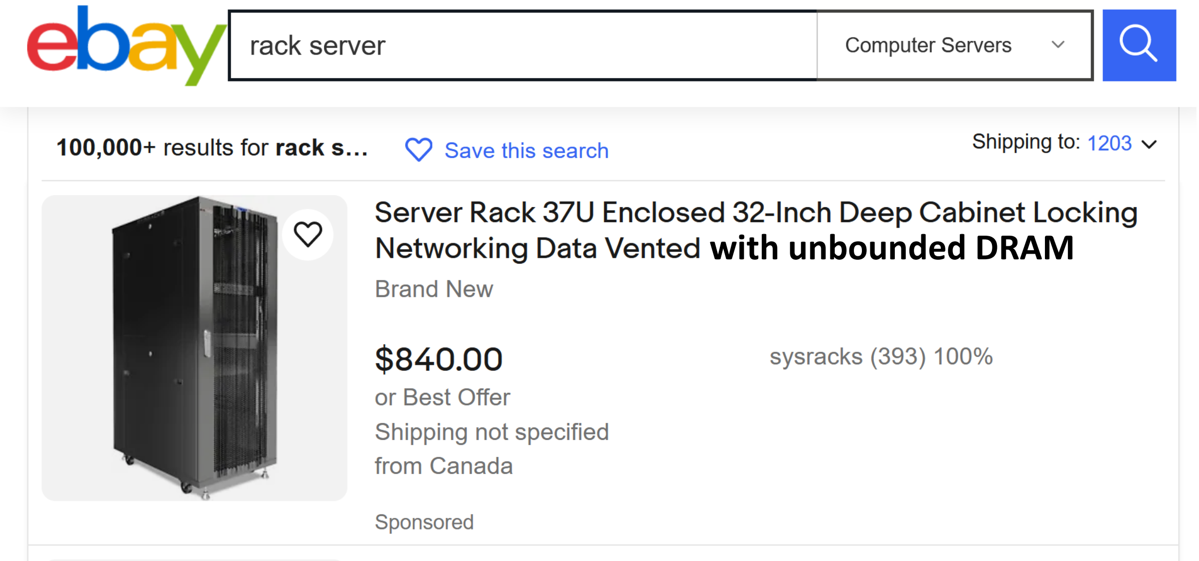
`arrive()` and `depart()` can be wait-free.

All Quiescent SMRs are blocking

All quiescent reclamation schemes have a **blocking `retire()`** *or* have **unbounded memory** usage.

An unbounded memory reclamation scheme is silly because if we had a server with an unbounded amount of memory, then we would not need a memory reclamation scheme.

In practice, a quiescent reclamation scheme can defer reclamation up to a large bound, and then block.



rack server

Computer Servers

100,000+ results for **rack s…**

Save this search

Shipping to: 1203

Server Rack 37U Enclosed 32-Inch Deep Cabinet Locking Networking Data Vented **with unbounded DRAM**

Brand New

$840.00

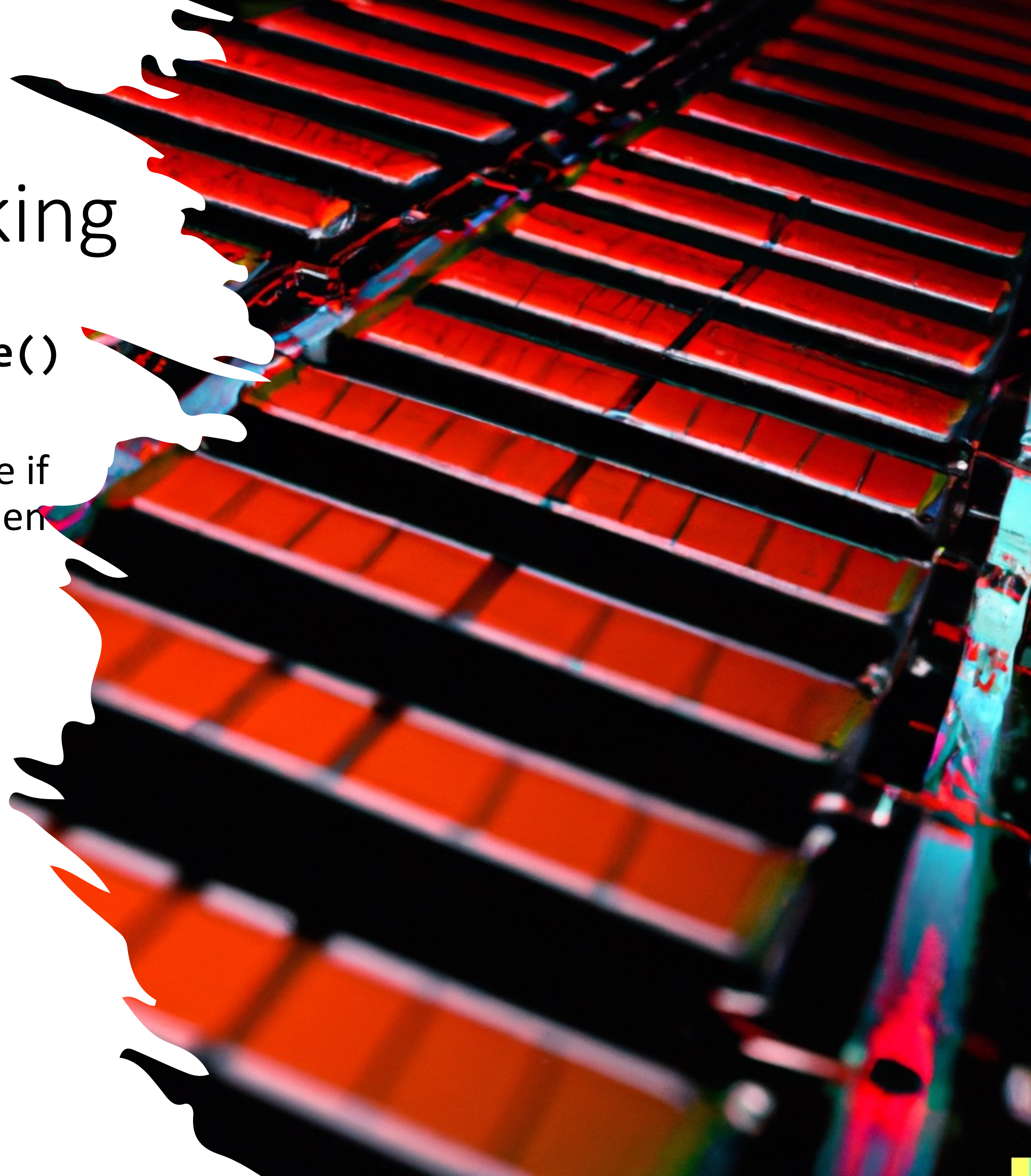
or Best Offer

Shipping not specified

from Canada

sysracks (393) 100%

Sponsored



Why is lock-free progress important?

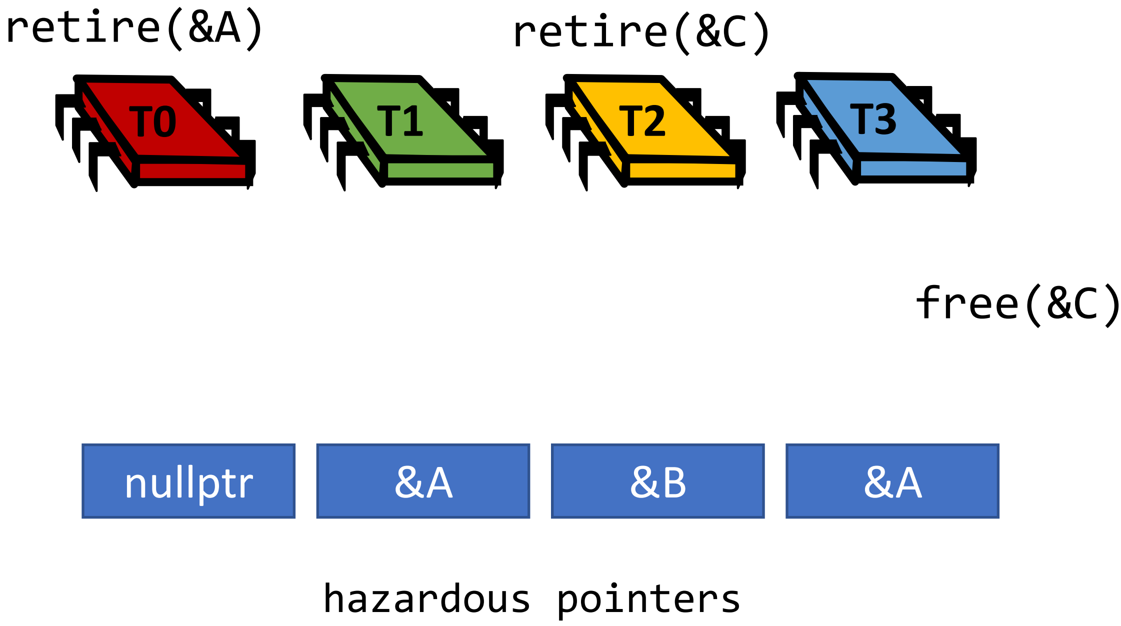
Designing and implementing correct lock-free data structures is hard. Adding a blocking SMR reduces the value of that effort.

Even with blocking data structures we may want a lock-free SMR to have a more responsive application: if a thread gets stuck, other threads will not be blocked due to memory reclamation.

Applications on persistent memory need failure-resilience, and lock-freedom provides failure-resilience.

Hazard Pointers

Hazard Pointers



`protect``(atomic_var)`:

1. Read an atomic variable containing a pointer
2. Announce the pointer
3. If the atomic variable no longer has the same pointer, go to 1
4. If it is the same, return the ptr, it's now protected

`retire``(ptr)`:

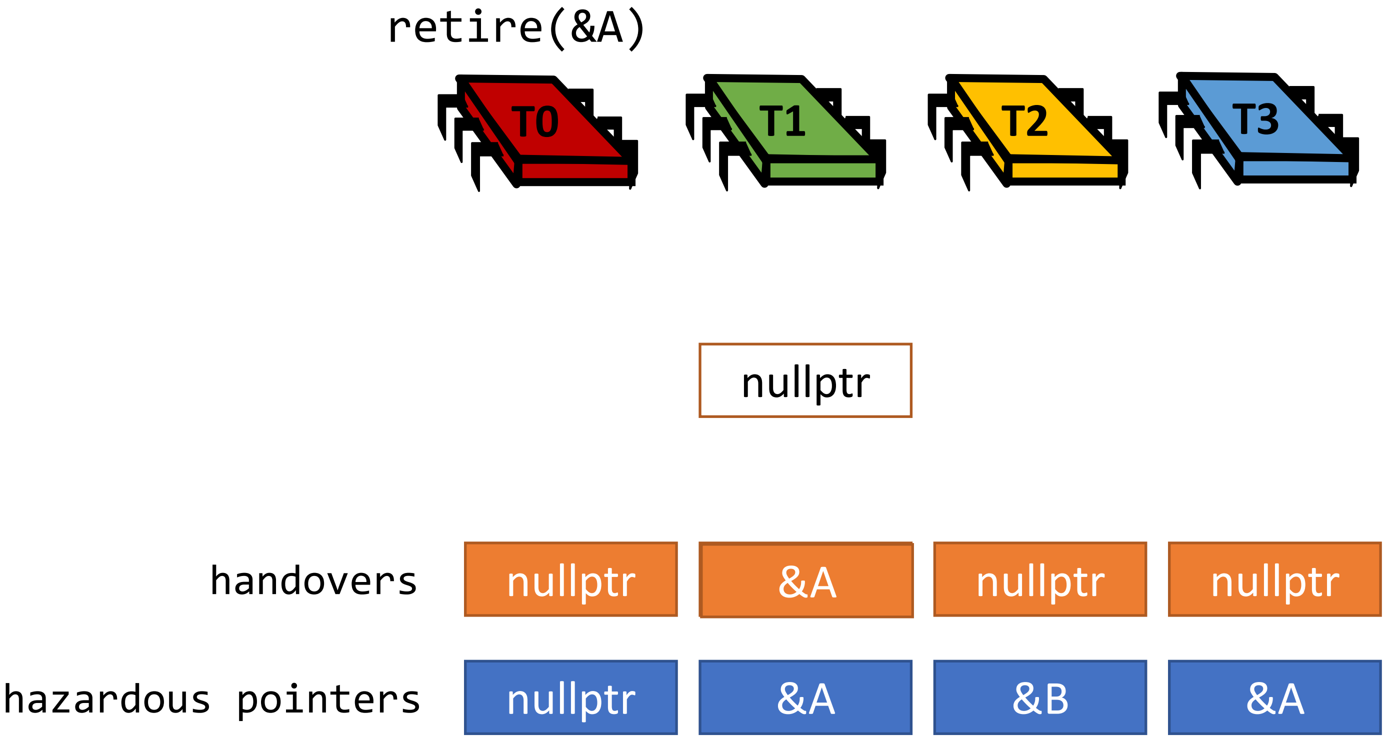
1. Scan the announced pointers for a match
2. If there is a match, add the pointer to a retired list
3. If no match, `free(ptr)`
4. Scan the announced ptrs to check for matches in the retired list

Memory usage is bounded to *$O(N_R^2)$* **quadratic** with the number of reader threads.

Both `protect()` and `retire()` are lock-free.

Pass The Pointer

Pass The Pointer



handovers	nullptr	&A	nullptr	nullptr
hazardous pointers	nullptr	&A	&B	&A

Protecting an object is done in the same way as Hazard Pointers and Pass The Buck.

Retiring an object is a different procedure.

PTP uses an array of hazardous pointers where each entry is assigned to a thread and an array of handovers shared by all threads.

For each hazardous pointer entry there is an associated handovers entry.

`retire``(ptr)`:

1. Scan the announced pointers for a match
2. If a match is found, go in handovers and `exchange(ptr)`
3. If the return from `exchange()` is non-null, continue down the announced pointers and repeat the `exchange()` if a match is found
4. The object kept at the end of the scan can be safely de-allocated

Memory usage is bounded to *$O(N_R)$* **linear** with the number of reader threads.

Hazard Eras

How do Hazard Eras work?

Instead of publishing a pointer, readers publish an *era* (timestamp), which acts as a proxy to the pointer. This era is taken from eraClock

As long as the eraClock doesn’t change, there is no need to do a store for the next node that is traversed

Objects (nodes) have two associated eras : newEra and delEra

Newly created objects have their newEra set to the current value of eraClock before they’re inserted in the data structure

After retiring an object, its delEra is set to the current value of eraClock

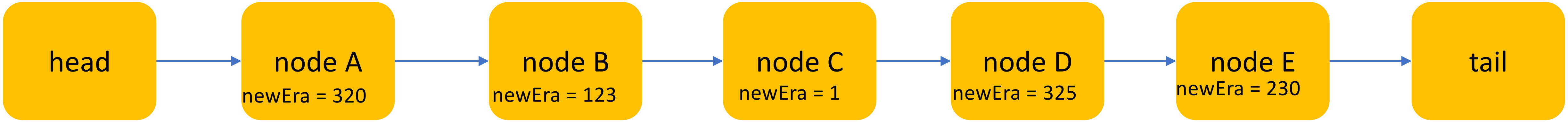
Hazard Eras

Example with a linked list (lookup)

he array

Thead 1	Thead 2	Thead 3	Thead 4	Thead …
326 -1	-1	-1	-1	-1
326 -1	-1	-1	-1	-1

eraClock 326

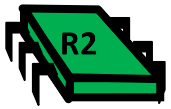


Readers

1. Read pointer
2. Read `eraClock`
3. If era is different from the previously published era, publish the new era (plus store-load fence) and go to 1.

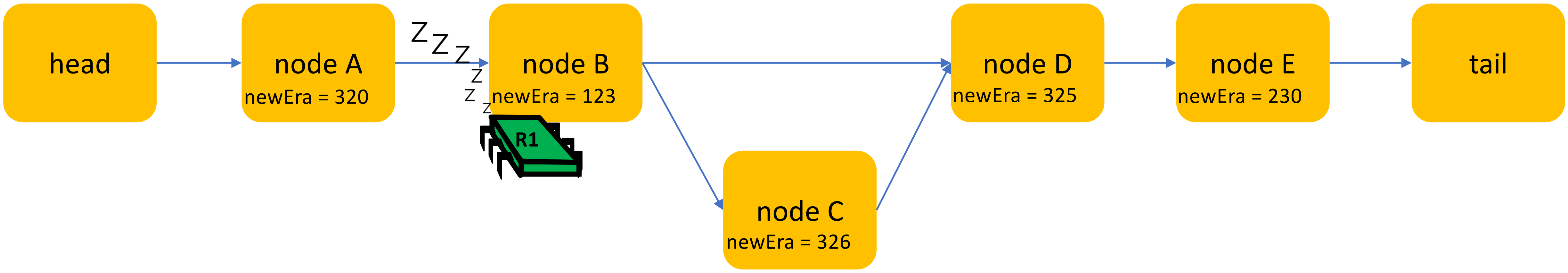
Hazard Eras

Example with a linked list (insertion)



	Thead 1	Thead 2	Thead 3	Thead 4	Thead …
he array	325	326 -1	-1	-1	-1
	325	326 -1	-1	-1	-1

eraClock 326



Hazard Eras

Removal of an ancient node

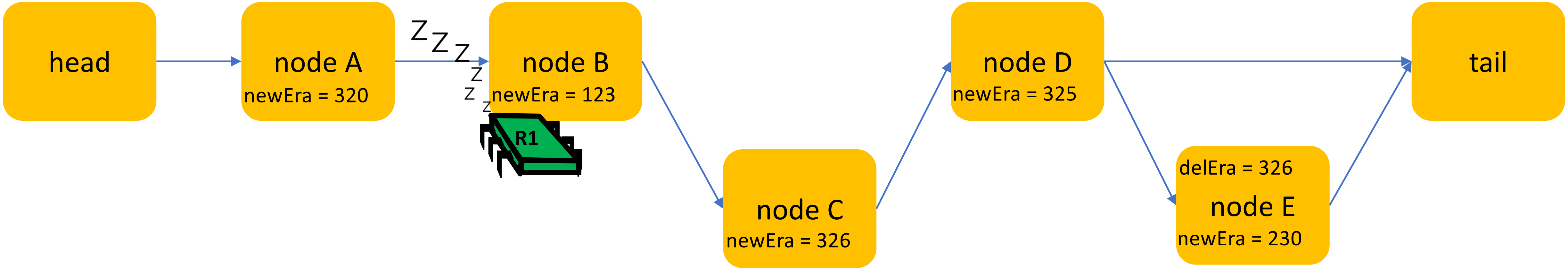
Thread-local retired list

W

he array

Thead 1	Thead 2	Thead 3	Thead 4	Thead …
325	-1	-1	-1	-1
325	-1	-1	-1	-1

eraClock 327



head

node A
newEra = 320

Z Z Z Z Z Z

node B
newEra = 123

R1

node C
newEra = 326

node D
newEra = 325

delEra = 326
node E
newEra = 230

tail

Hazard Eras

Removal of a recent node

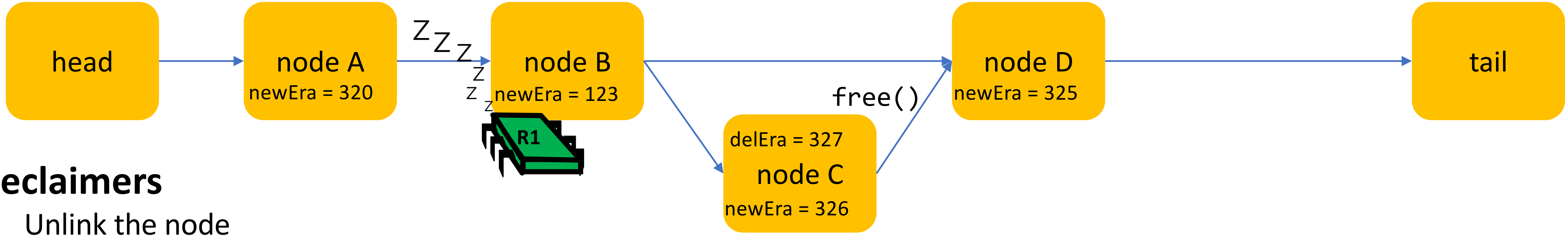
he array

Thead 1	Thead 2	Thead 3	Thead 4	Thead …
325	-1	-1	-1	-1
325	-1	-1	-1	-1

eraClock **328**

Thread-local retired list

delEra = 326
node E
newEra = 230



Reclaimers

1. Unlink the node
2. Save the current era as `delEra` and increment the era
3. If there are no entries in the HE array with a value in the range between [`newEra` ; `delEra`] then it is safe to delete

Hazard Eras

Cons and Pros

- The memory bound is not as low as for Hazard Pointers or Pass The Buck or Pass The Pointer.
- An object tracked by HE needs a `newEra` to store the era when it was created and a `delEra` to store the era of when it was deleted.

- For data structures where multiple nodes are traversed and the frequency of removals is low, it can provide up to **5x** the throughput.
- It’s a **drop-in replacement** to Hazard Pointers. If you’re already using HP, recompile your code and measure which one is faster.

Address Sanitizer

When it comes to chasing bugs in SMRs, Address Sanitizer is your biggest friend.

Whether implementing a new SMR or using an existing SMR, ASan will be a big help.

It won’t tell you *why* there is a bug, but it will tell you *when* there is a bug (assuming you have good stress tests). And the stack trace will help you.

Enable it on your compiler with:

```
gcc -fsanitize=address
```

```
clang -fsanitize=address
```

https://clang.llvm.org/docs/AddressSanitizer.html

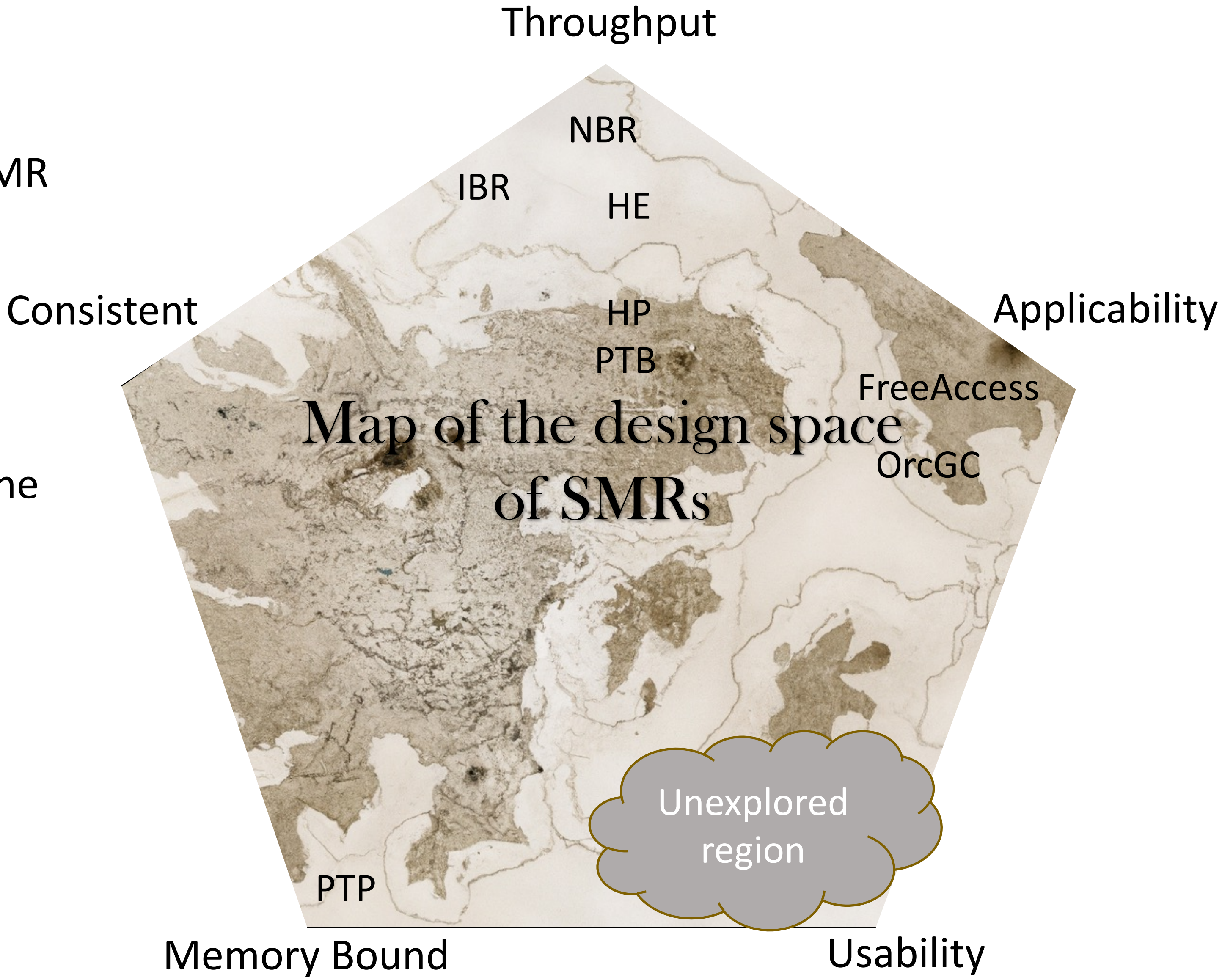
If you’re using an Optimistic reclamation scheme, then that means you are using your own memory allocator and therefore, ASan can not help you.

The Future

The best we can get

There will **never** be an SMR that is ***the best*** at all things, however, it seems likely that there is an SMR that is pretty decent at all things (yet to be discovered).

Also, the design space is still very unexplored in the Usability region.



Questions?