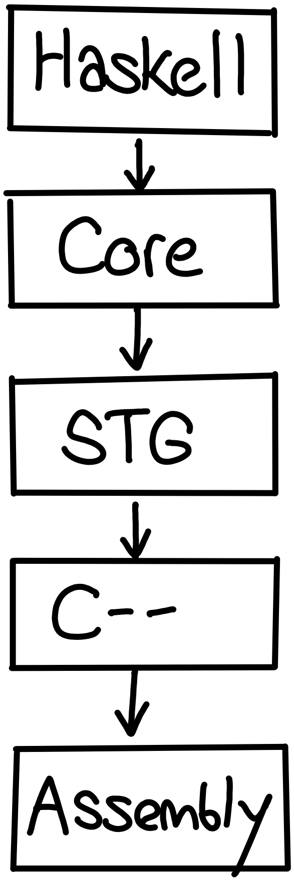
The GHC Runtime System

Edward Z. Yang

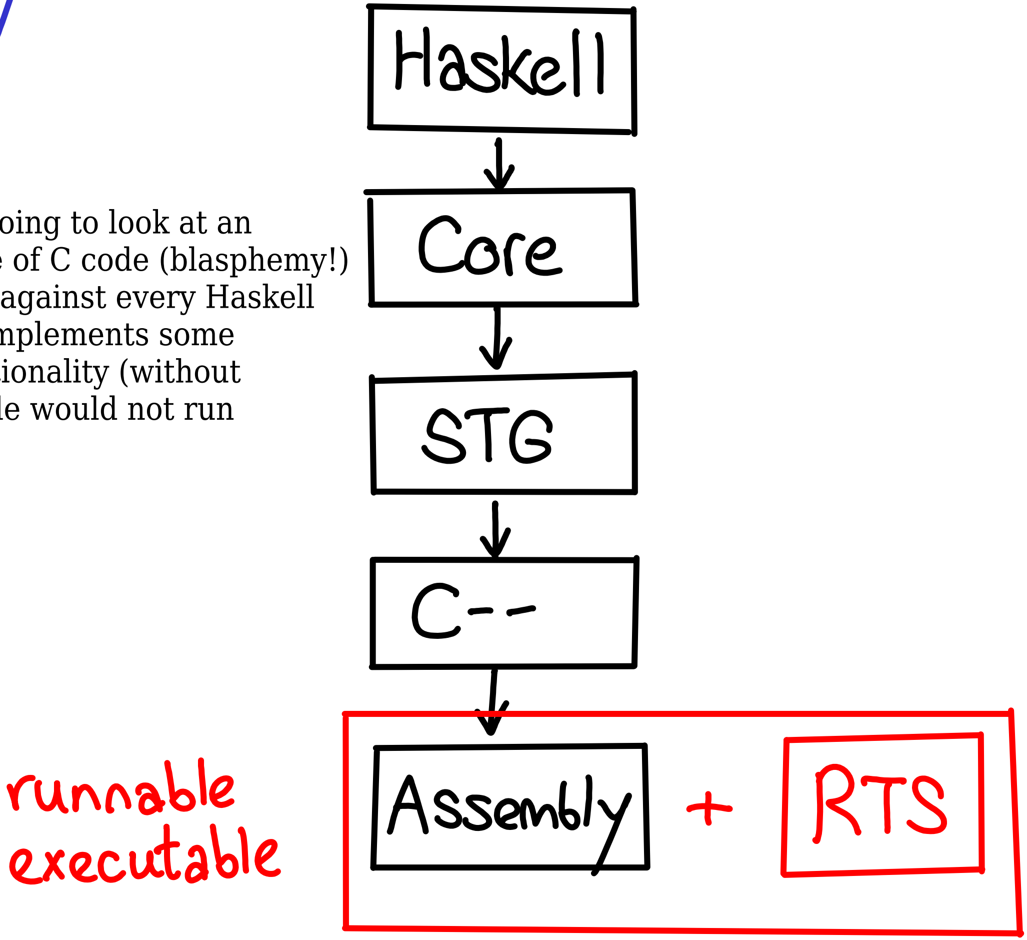
Last week



Last week, David Terei lectured about the compilation pipeline which is responsible for producing the executable binaries of the Haskell code you actually want to run.

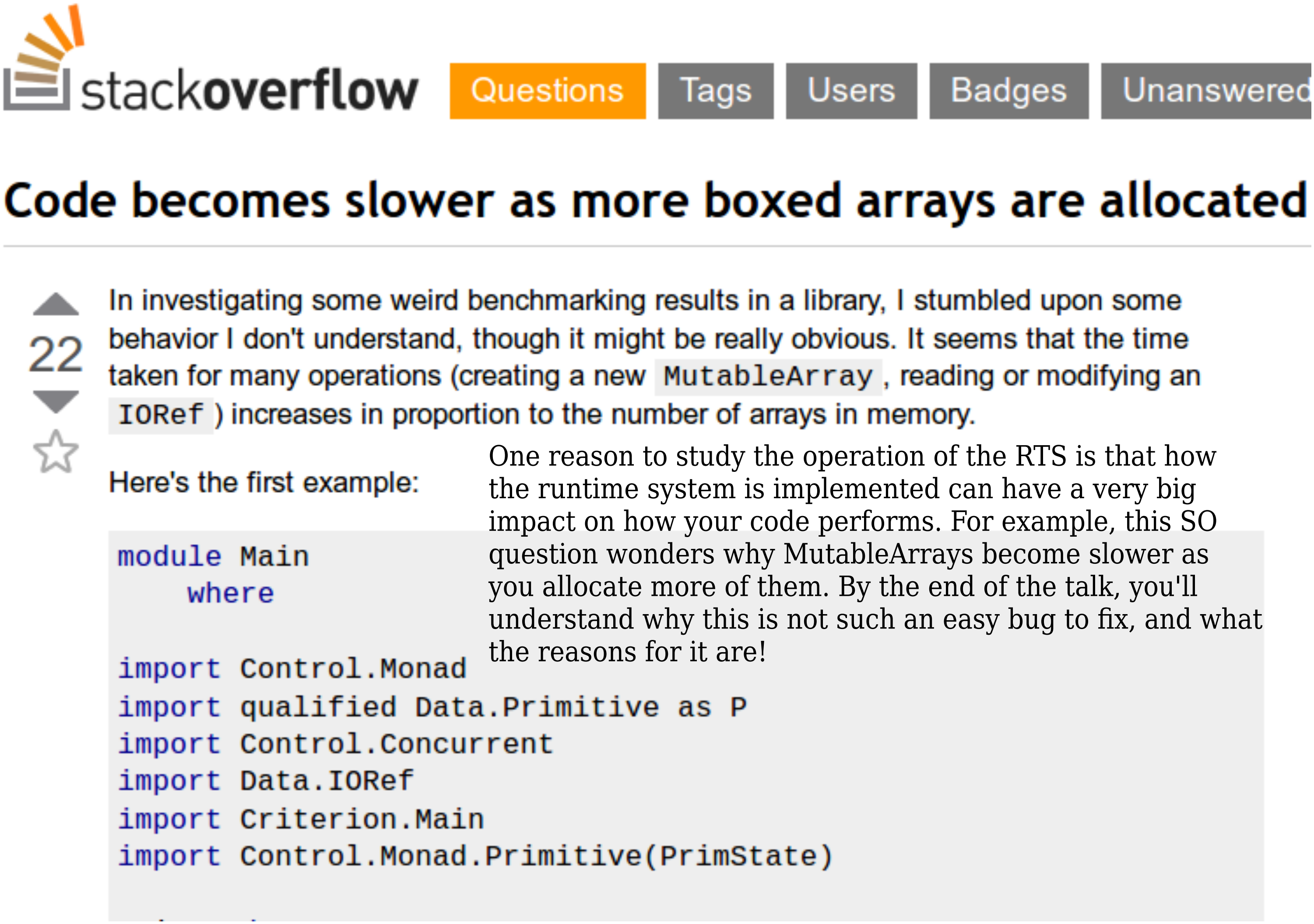
Today

Today, we are going to look at an important piece of C code (blasphemy!) which is linked against every Haskell program, and implements some important functionality (without which, your code would not run at all!)



Why learn about the RTS?

But first, an important question to answer: why should anyone care about a giant blob of C code that your Haskell code looks like? Isn't simply an embarrassing corner of Haskell that we should pretend doesn't exist?



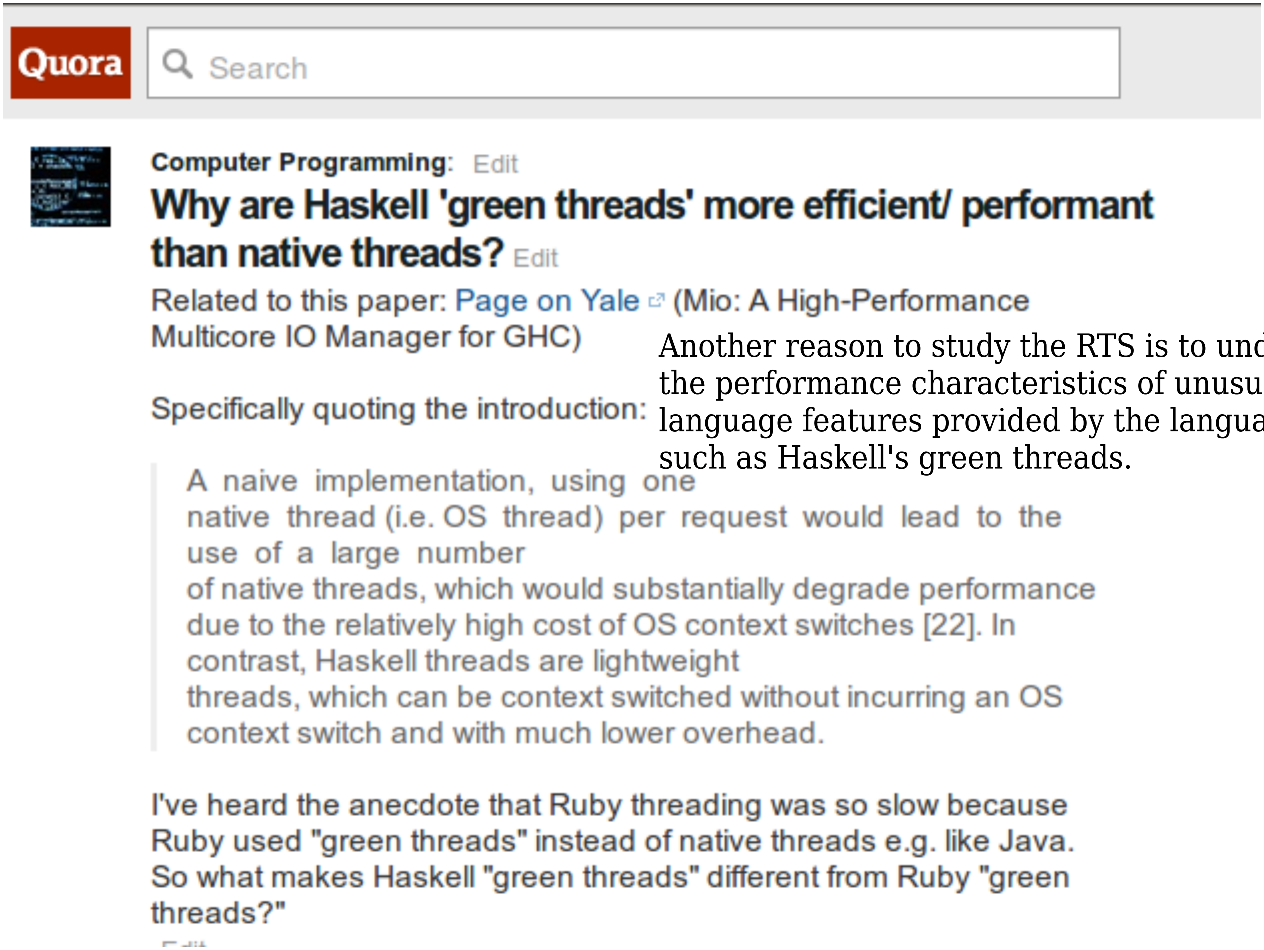
Code becomes slower as more boxed arrays are allocated

In investigating some weird benchmarking results in a library, I stumbled upon some behavior I don't understand, though it might be really obvious. It seems that the time taken for many operations (creating a new `MutableArray`, reading or modifying an `IORef`) increases in proportion to the number of arrays in memory.

Here's the first example:

```
module Main
    where

import Control.Monad
import qualified Data.Primitive as P
import Control.Concurrent
import Data.IORef
import Criterion.Main
import Control.Monad.Primitive(PrimState)
```

One reason to study the operation of the RTS is that how the runtime system is implemented can have a very big impact on how your code performs. For example, this SO question wonders why MutableArrays become slower as you allocate more of them. By the end of the talk, you'll understand why this is not such an easy bug to fix, and what the reasons for it are!



Computer Programming:

Why are Haskell 'green threads' more efficient/ performant than native threads?

Related to this paper: Page on Yale (Mio: A High-Performance Multicore IO Manager for GHC)

Specifically quoting the introduction:

> A naive implementation, using one native thread (i.e. OS thread) per request would lead to the use of a large number of native threads, which would substantially degrade performance due to the relatively high cost of OS context switches [22]. In contrast, Haskell threads are lightweight threads, which can be context switched without incurring an OS context switch and with much lower overhead.

I've heard the anecdote that Ruby threading was so slow because Ruby used "green threads" instead of native threads e.g. like Java. So what makes Haskell "green threads" different from Ruby "green threads?"

Another reason to study the RTS is to understand the performance characteristics of unusual language features provided by the language, such as Haskell's green threads.

In theory, only the semantics of Haskell's multithreading mechanisms should matter, but in practice, the efficiency and underlying implementation are important factors.

Perhaps after this class you will go work for some big corporation, and never write any more Haskell. But most high-level languages you will write code for are going to are going to have some runtime system of some sort, and many of the lessons from GHC's runtime are transferable to those settings. I like to think that GHC's runtime is actually far more understandable than many of these others (we believe in documentation!)

CLR

JVM

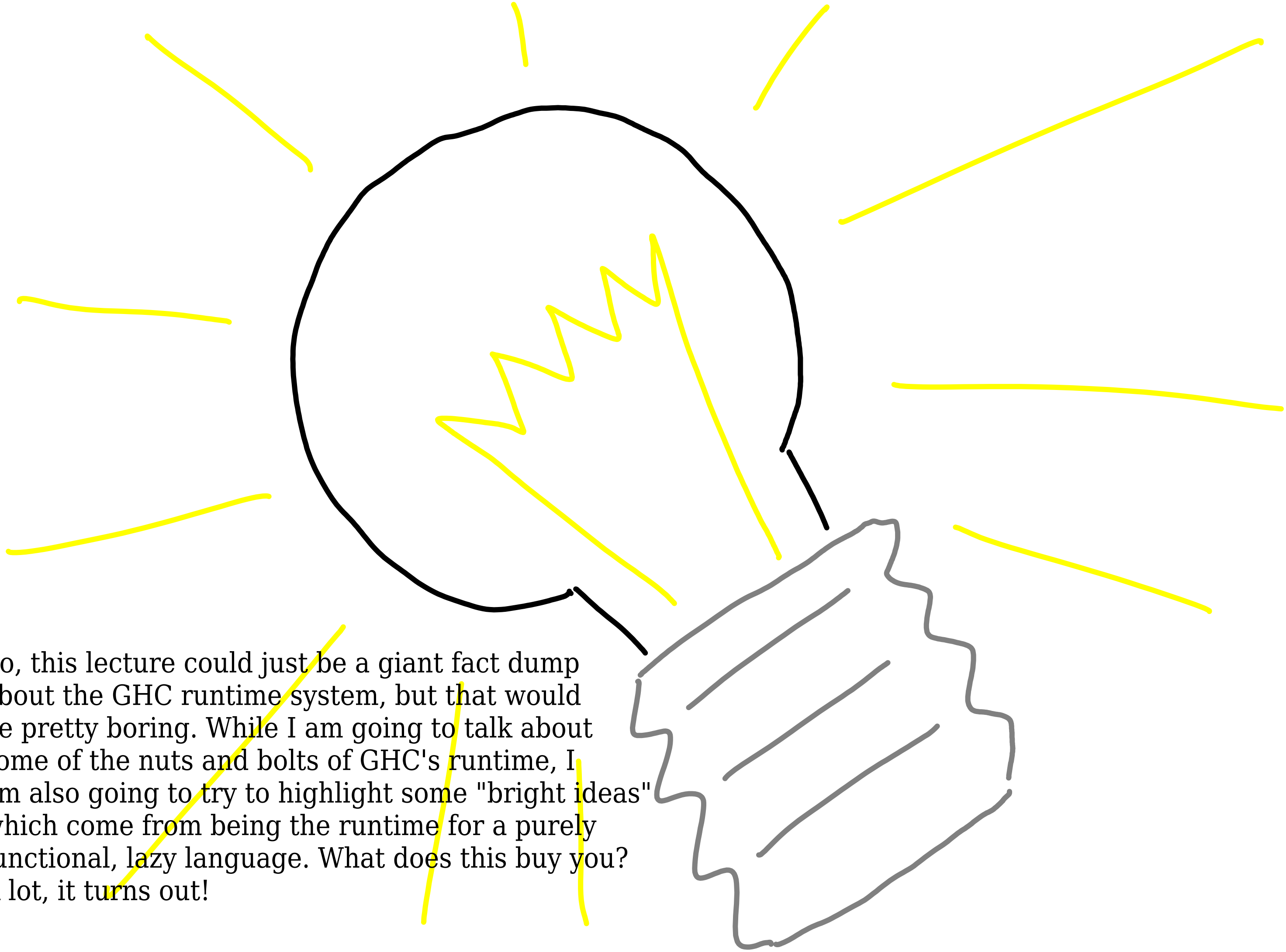
V8

GHC

Golang

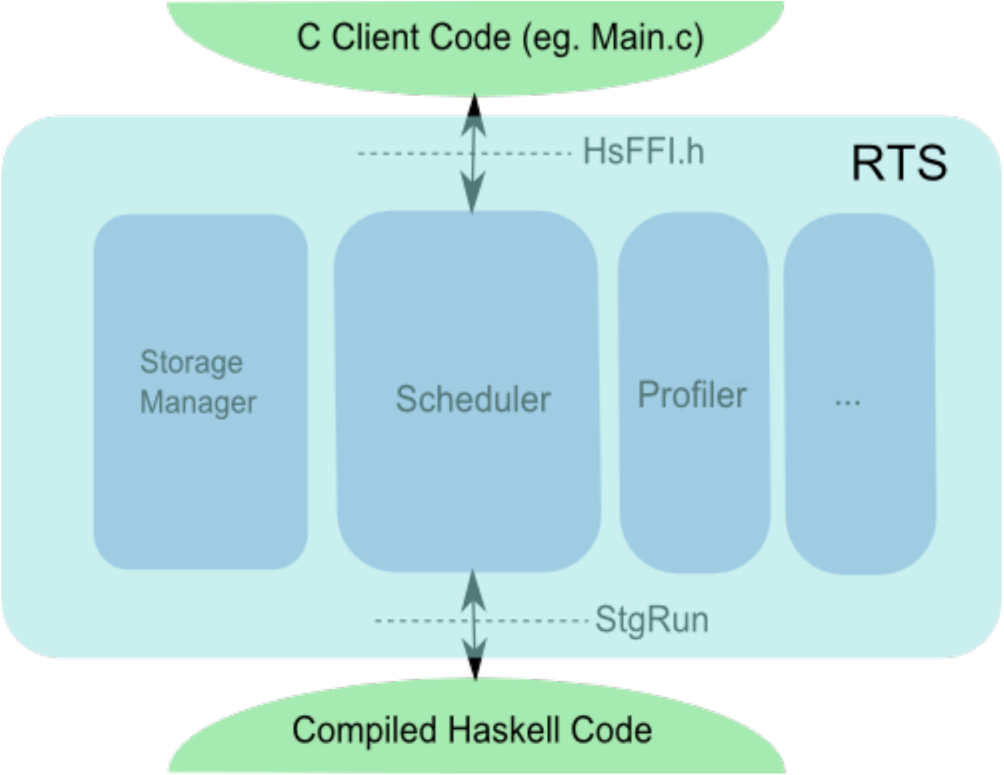
SpiderMonkey

[language]



So, this lecture could just be a giant fact dump about the GHC runtime system, but that would be pretty boring. While I am going to talk about some of the nuts and bolts of GHC's runtime, I am also going to try to highlight some "bright ideas" which come from being the runtime for a purely functional, lazy language. What does this buy you? A lot, it turns out!

Let's dive right in. Here's a diagram from the GHC Trac which describes the main "architecture" of the runtime. To summarize, the runtime system is a blob of code that interfaces between C client code (sometimes trivial, but you can call into Haskell from C) and the actual compiled Haskell code.



In a nutshell...

There is a lot of functionality that the RTS packs, let's go through a few of them.

→ Storage Manager (Garbage Collection)

The storage manager manages the memory used by a Haskell program; most importantly it includes the garbage collector which cleans up unused memory.

→ Scheduler

The scheduler is responsible for actually running Haskell code, and multiplexing between Haskell's green threads and managing multicore Haskell.

→ Bytecode Interpreter (GHCi)

When running GHCi, GHC typechecks and translates Haskell code into a bytecode format. This bytecode format is then interpreted by the RTS. The RTS also does the work of switching between compiled code and bytecode.

→ Dynamic Linker

The RTS sports a homegrown linker, used to load objects of compiled code at runtime. Uniquely, we can also load objects that were *statically* compiled (w/o -fPIC) by linking them at load-time. I hear Facebook uses this in Sigma.

→ Software Transactional Memory

A chunk of RTS code is devoted to the implementation of software transactional memory, a compositional concurrency mechanism.

→ Profiling

The RTS, esp. the GC, has code to dump profiling information when you ask for heap usage, e.g. +RTS -h

[and more...]

In a nutshell...

In this talk, we're going to focus on the storage manager and the scheduler, as they are by far the most important components of the RTS. Every Haskell program exercises them!

→ Storage Manager (Garbage Collection)

→ Scheduler

→ Bytecode Interpreter (GHCi)

→ Dynamic Linker

→ Software Transactional Memory

→ Profiling

[and more...]

Here's the agenda

→ Storage Manager

- Generational Copying GC
- Write barriers & promotion
- Parallel GC (briefly)

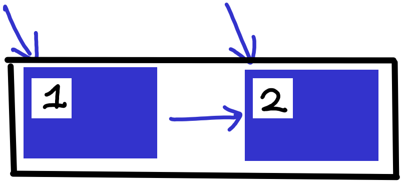
→ Scheduler

- Threads
- Load balancing
- MVars
- HECs
- Bound threads

Garbage Collection

Garbage Collection: Brief Review

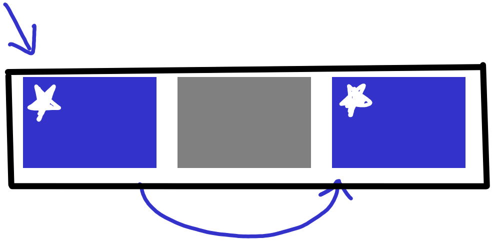
Reference Counting



- X Can't handle cycles

PHP, Perl, Python*

Mark and Sweep



- X Fragmentation
- X Needs to sweep entire heap

Golang, Ruby

Generational Copying Collector

JVM, V8, GHC

"Most objects die young"

—the Generational Hypothesis

If you are going to GC in a real world system, then there is basically one absolutely mandatory performance optimization you have to apply: generational collection. You've probably heard about it before, but the generational hypothesis states that most objects die young.

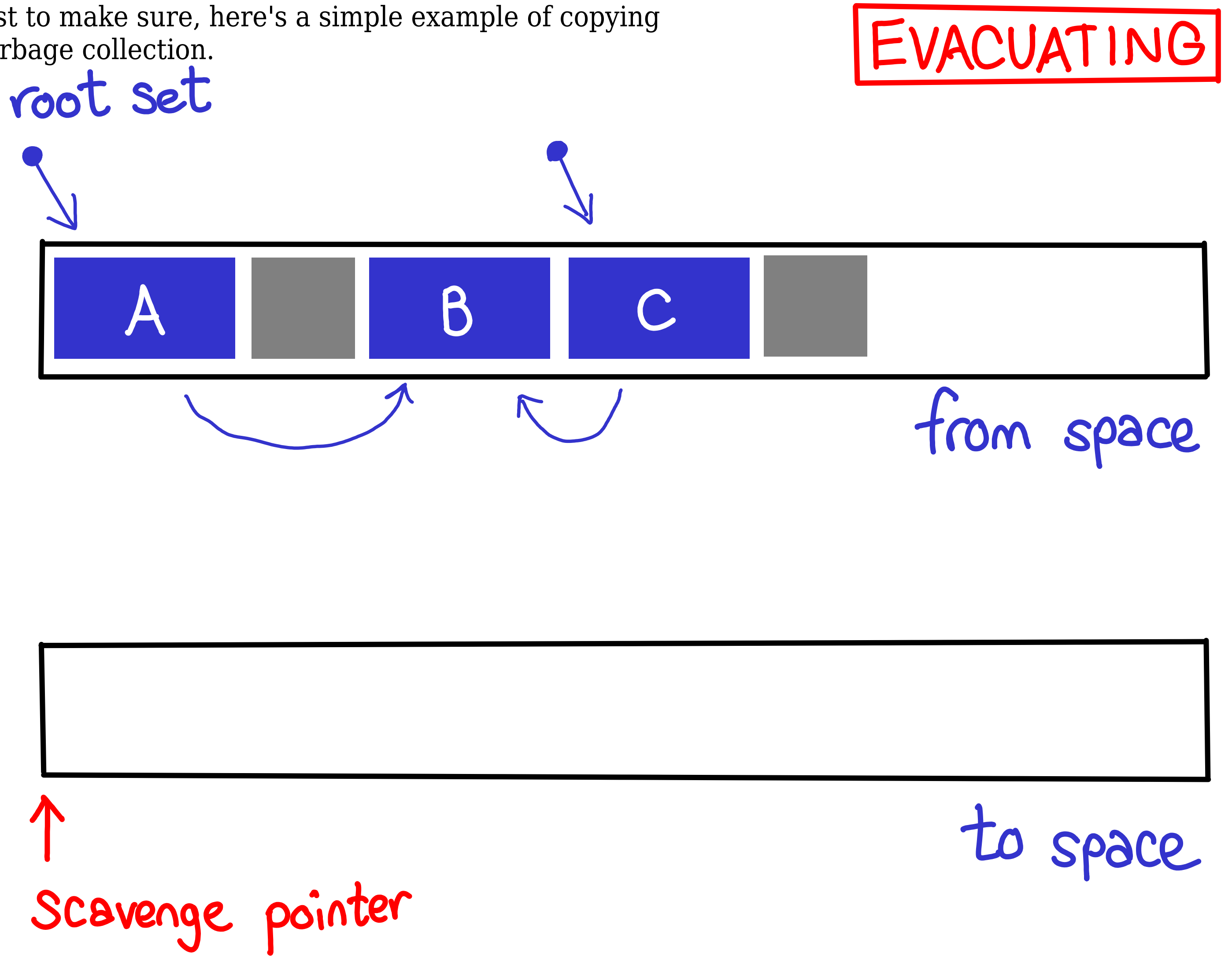
Generational Copying Collector

JVM, V8, GHC

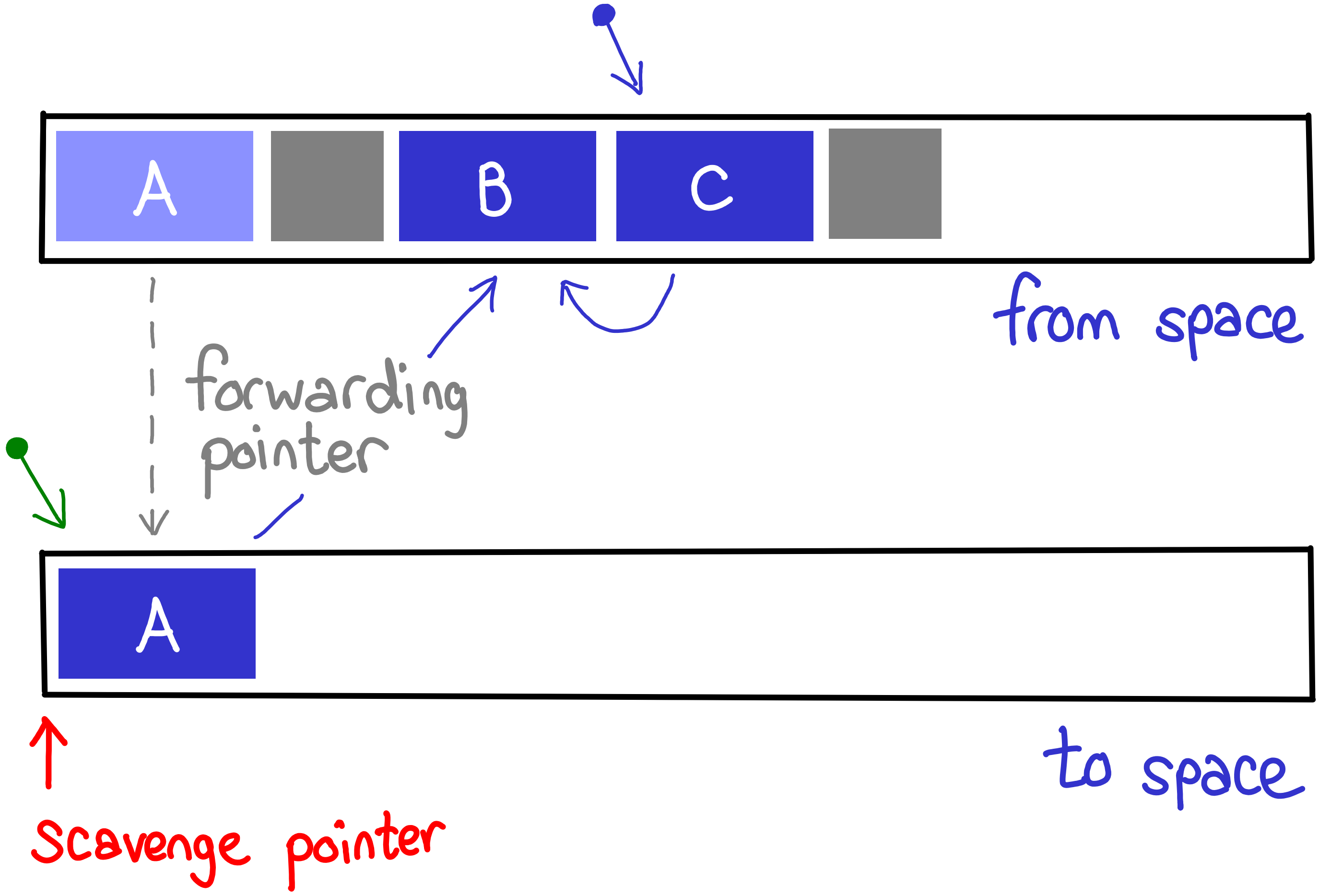
> “Most objects die young”
>
> especially in functional languages!
>
> —the Generational Hypothesis

This is especially true in pure functional languages like Haskell, where we do very little mutating a lot of allocating new objects when we do computation. (How else are you going to compute with immutable objects?!)

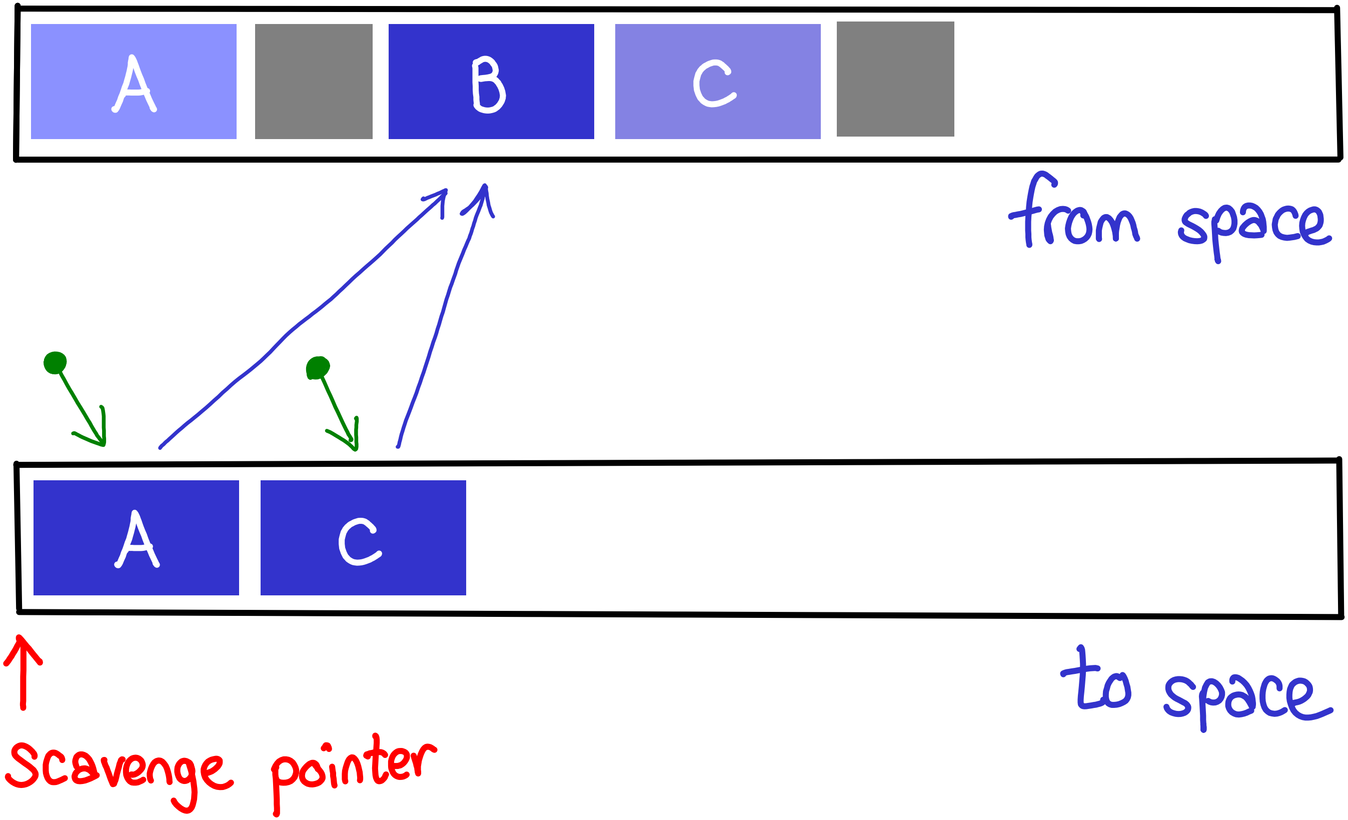
Just to make sure, here's a simple example of copying garbage collection.

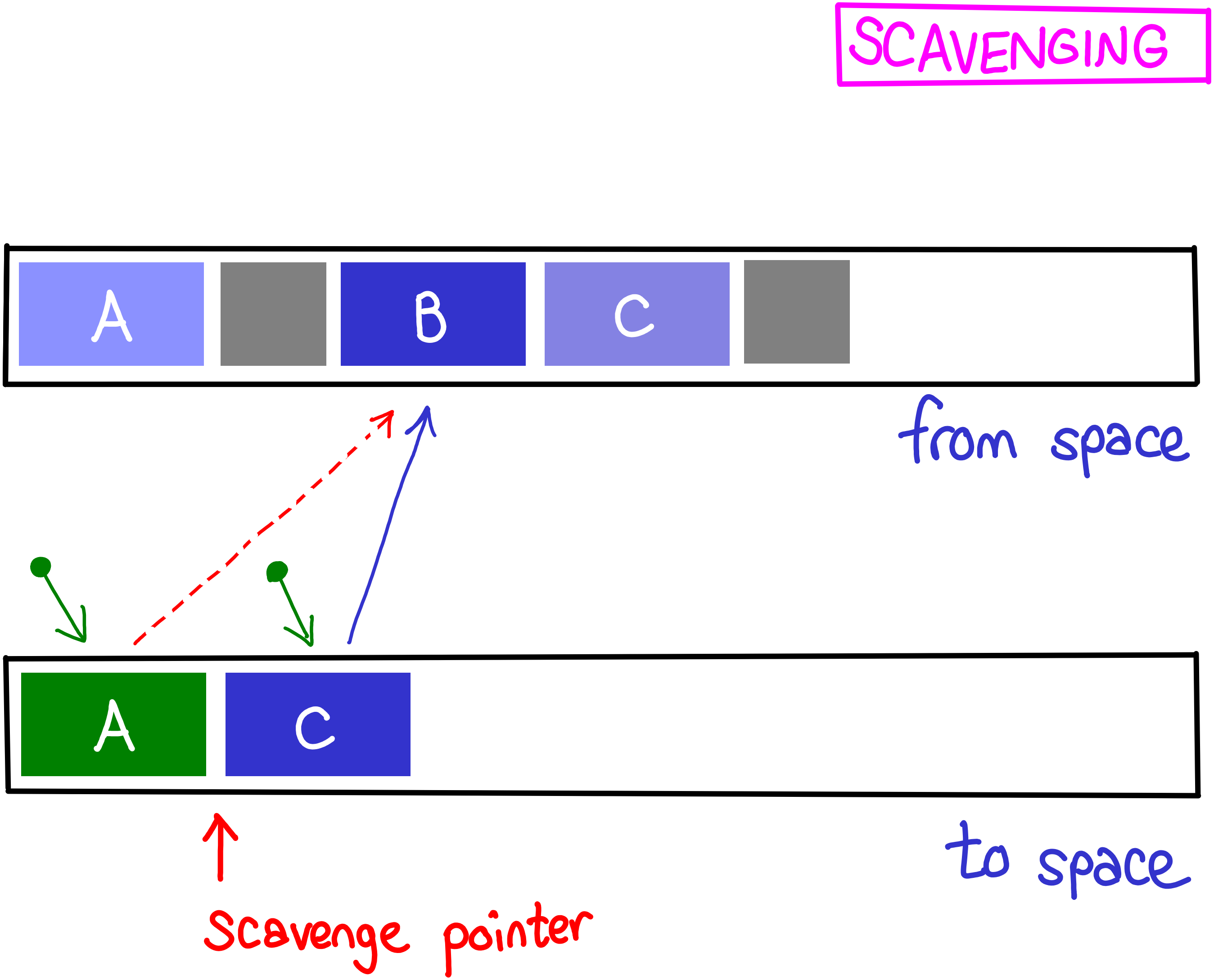


EVACUATING

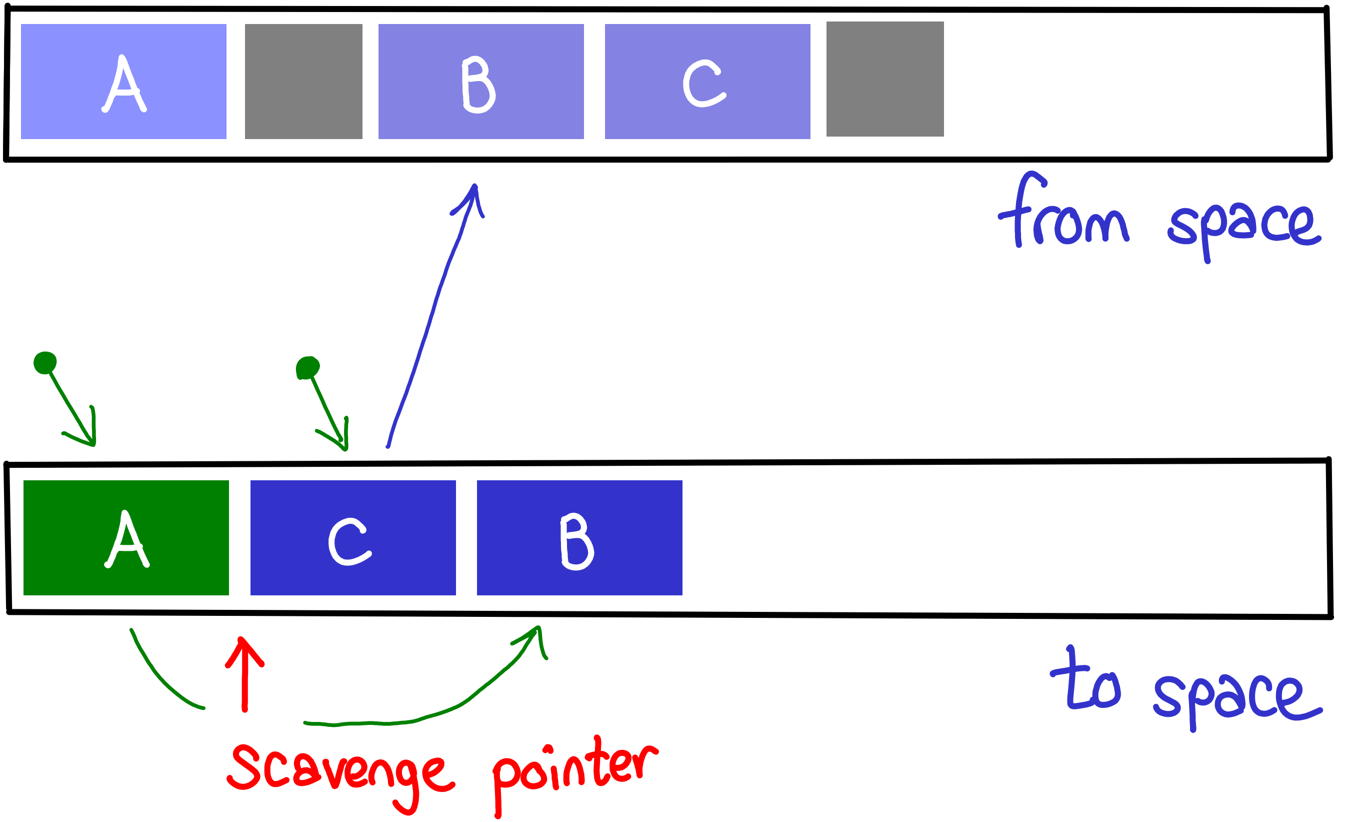

A
B
C
from space
forwarding pointer
A
to space
scavenge pointer

EVACUATING

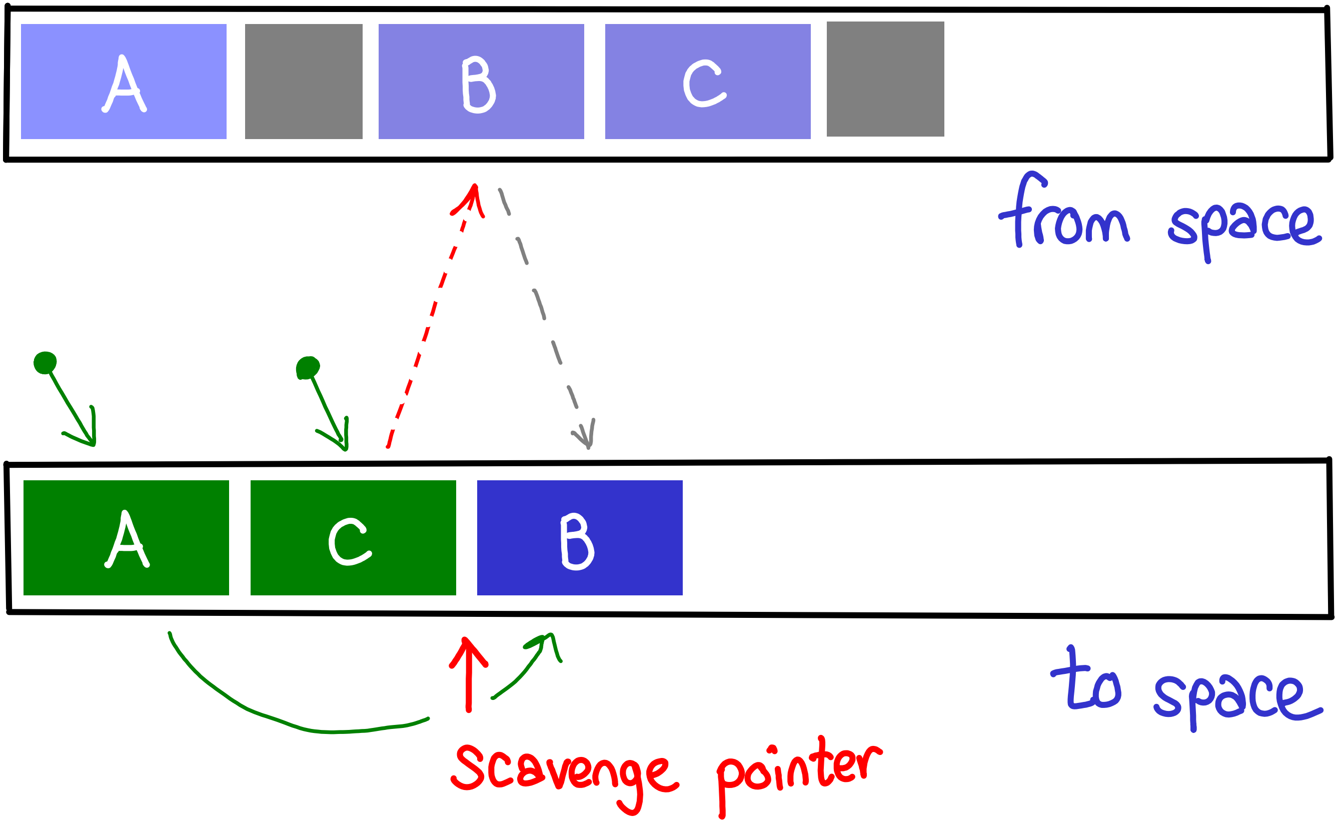



SCAVENGING
A
B
C
from space
A
C
to space
scavenge pointer

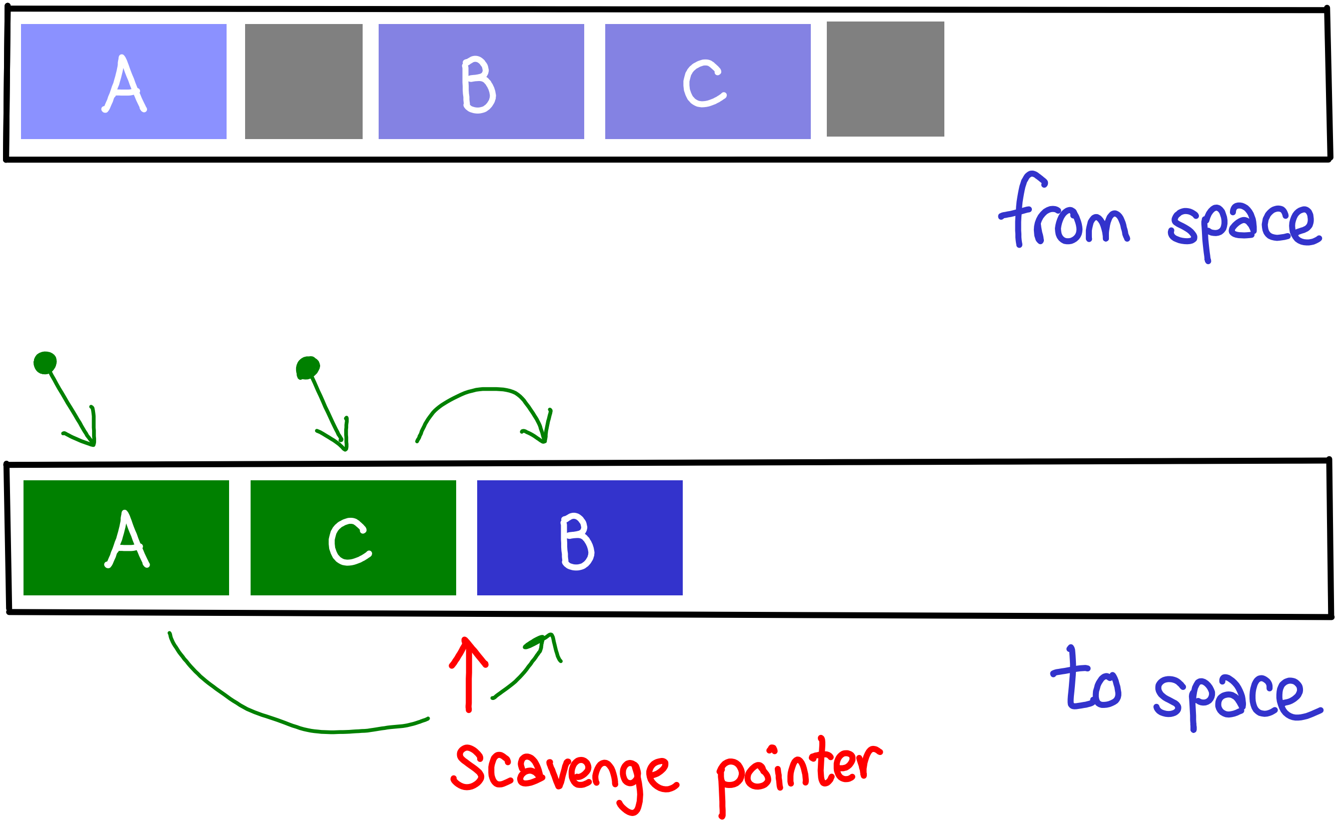
EVACUATING



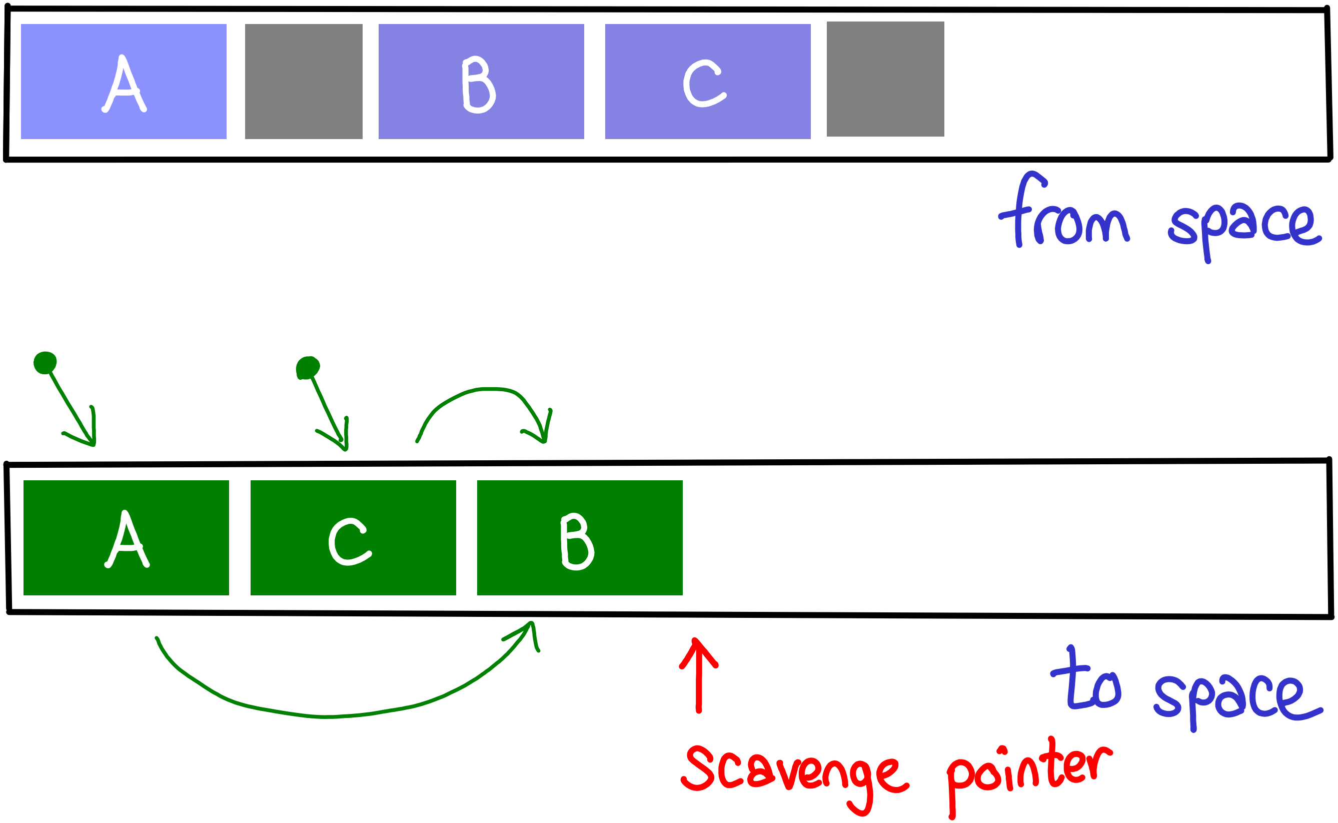
SCAVENGING



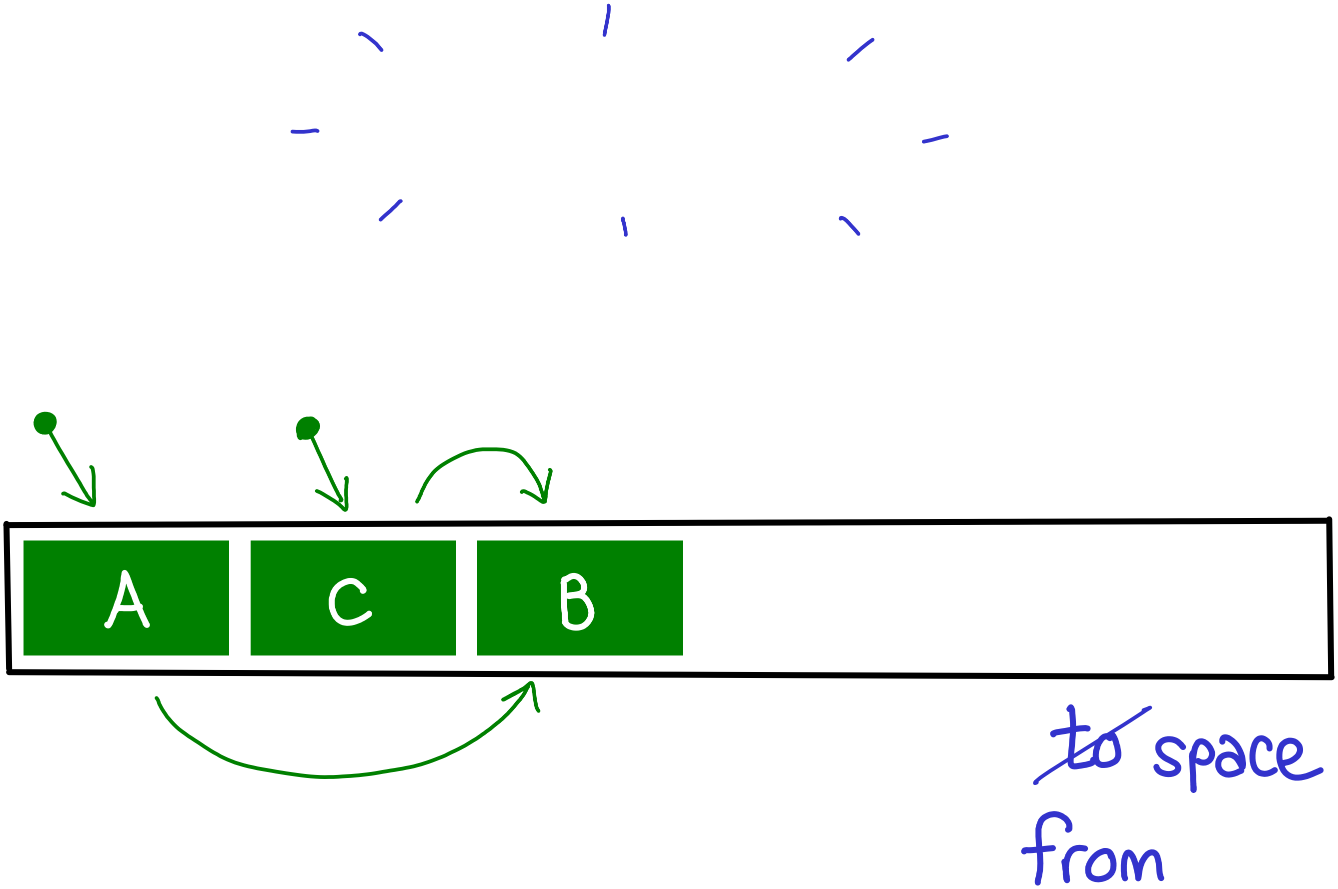
EVACUATING



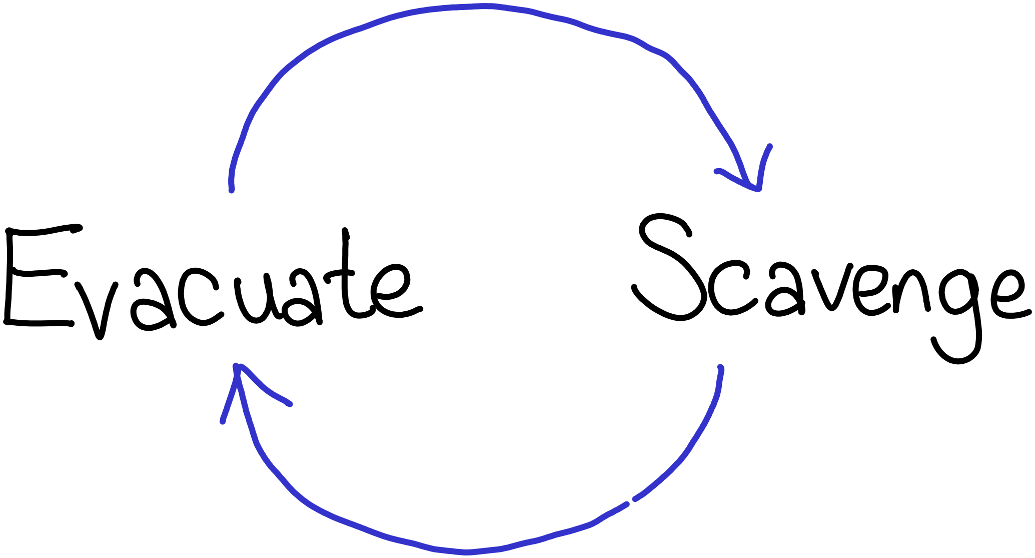
SCAVENGING



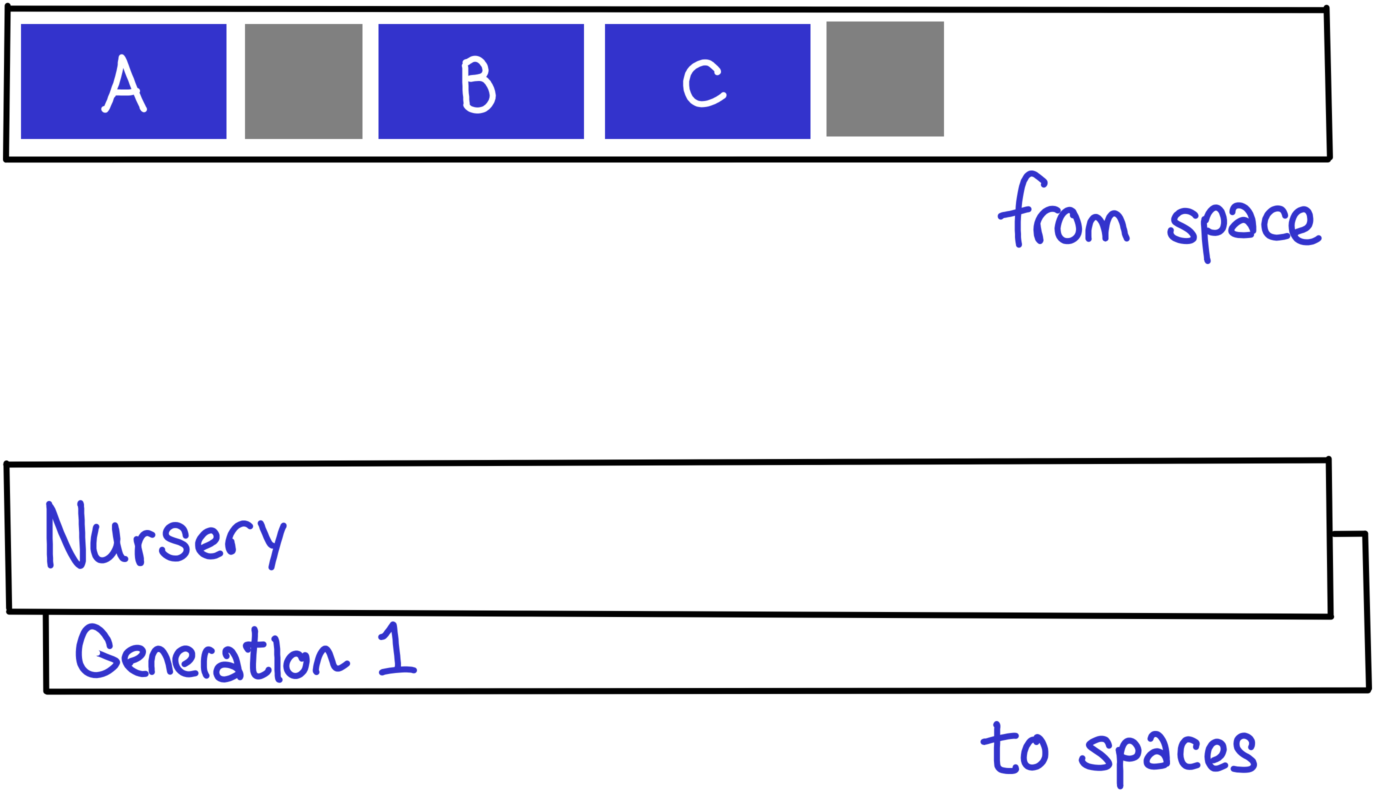
The more garbage you have, the faster GC runs.



Roughly, you can think of copying GC as a process which continually cycles between evacuating and scavenging objects.

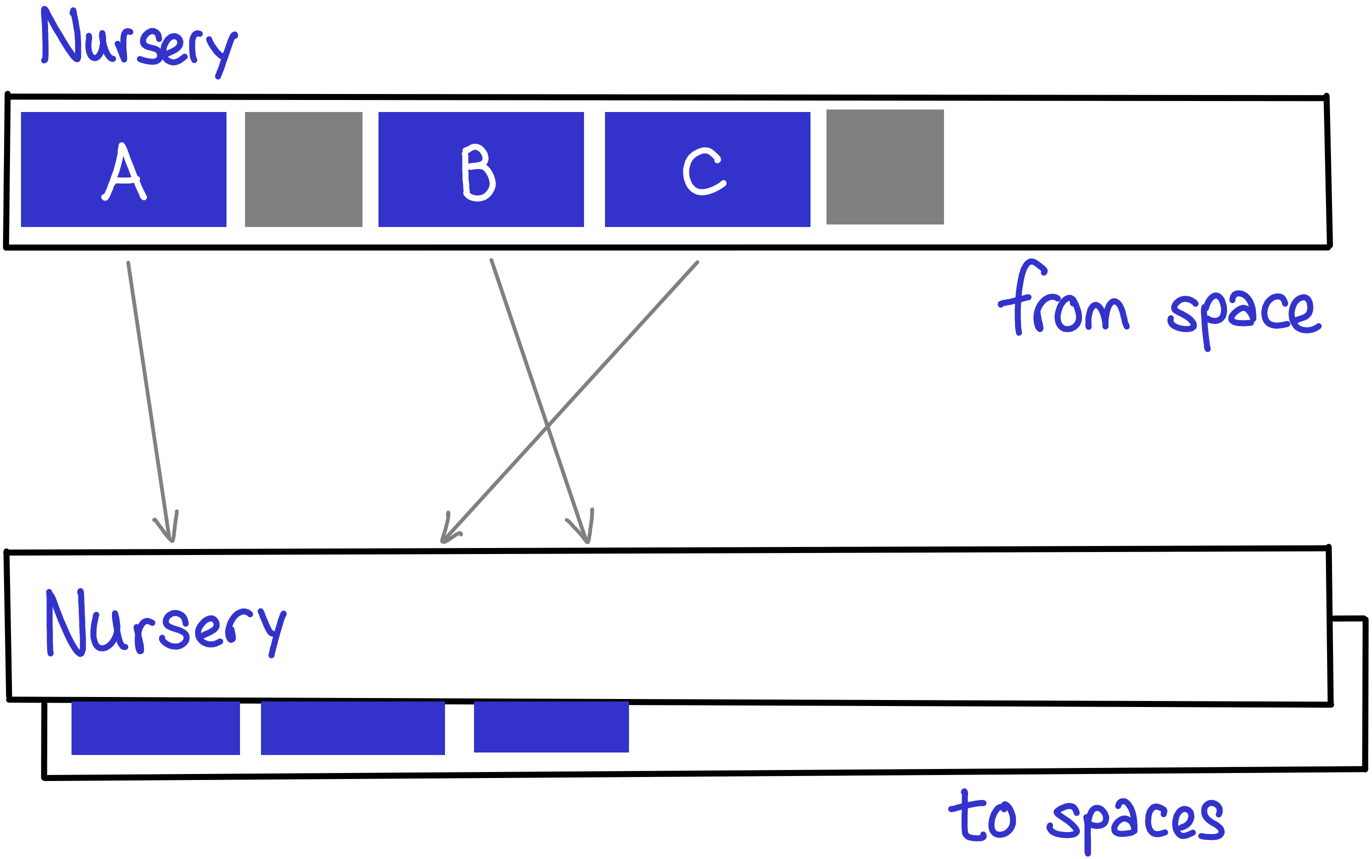


With this knowledge in hand, we can explain how generational copying collection works. Let's take the same picture as last time, but refine our view of the to spaces so that there are now to regions of memory: the nursery (into which new objects are allocated), and the first generation.

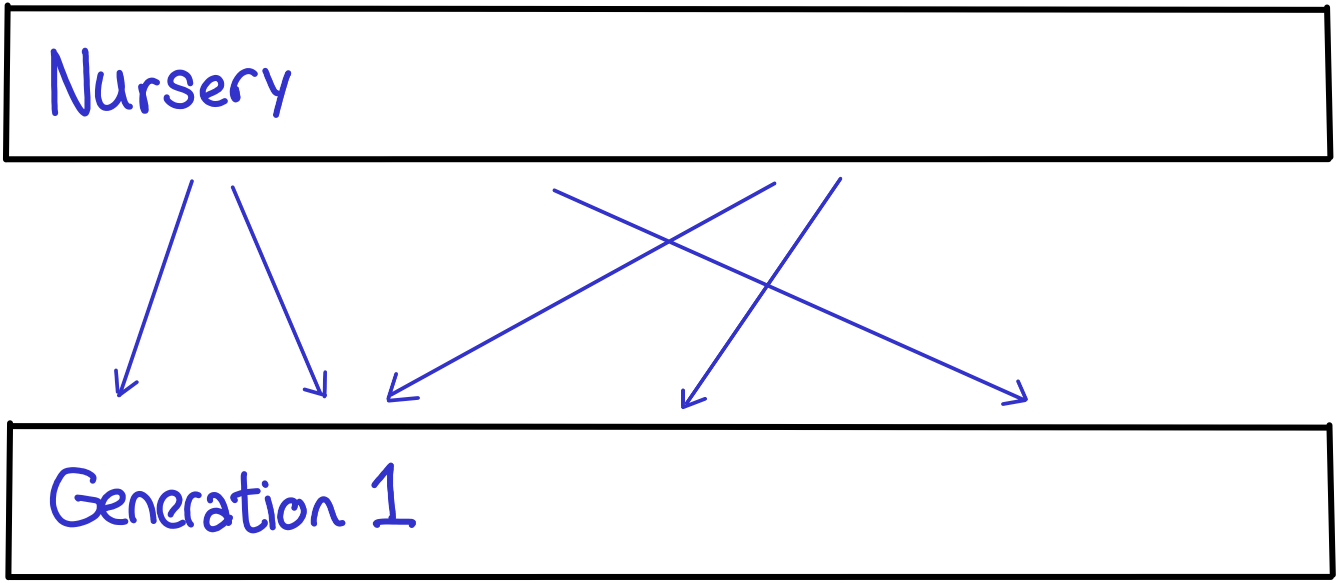


The difference now is that when we do copying collection, we don't move objects into the nursery: instead, we *tenure* them into the first generation.

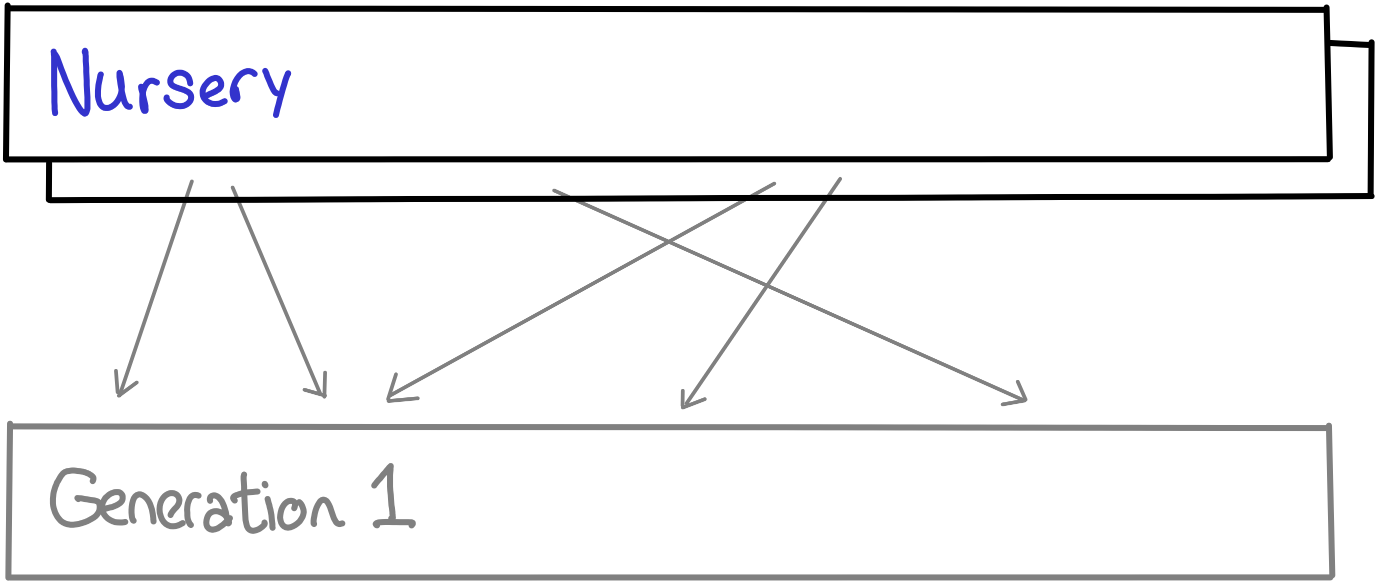
Tenuring



In generational garbage collection, we maintain an important invariant, which is that pointers only ever go from the nursery to the first generation, and not vice versa. It's easy to see that this invariant is upheld if all objects in your system are immutable (+1 for Haskell!)



If this invariant is maintained, then we can do a partial garbage collection by only scanning over things in the nursery, and assuming that the first generation is live. Such a garbage collection is called a "minor" garbage collection. Then, less frequently, we do a major collection involving all generations to free up garbage from the last generation.



Minor GC

Generational Copying Collector

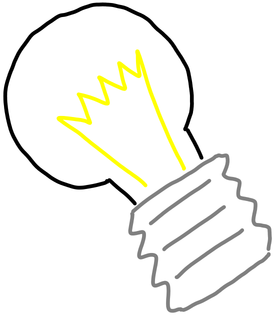
- The more garbage you have, the faster it runs
- Free memory is contiguous

The key points.

Having contiguous memory to allocate from is a big deal: it means that you can perform allocations extremely efficiently. To allocate in Haskell, you only need to do an addition and a compare.

```
mk_exit()
    entry:
        Hp = Hp + 16;
        if (Hp > HpLim) goto gc;

        v::I64 = I64[R1] + 1;

        I64[Hp - 8] = GHC_Types_I_con_info;
        I64[Hp + 0] = v::I64;

        R1 = Hp;
        Sp = Sp + 8;
        jump (I64[Sp + 0]) ();

    gc: HpAlloc = 16;
        jump stg_gc_enter_1 ();
}
```

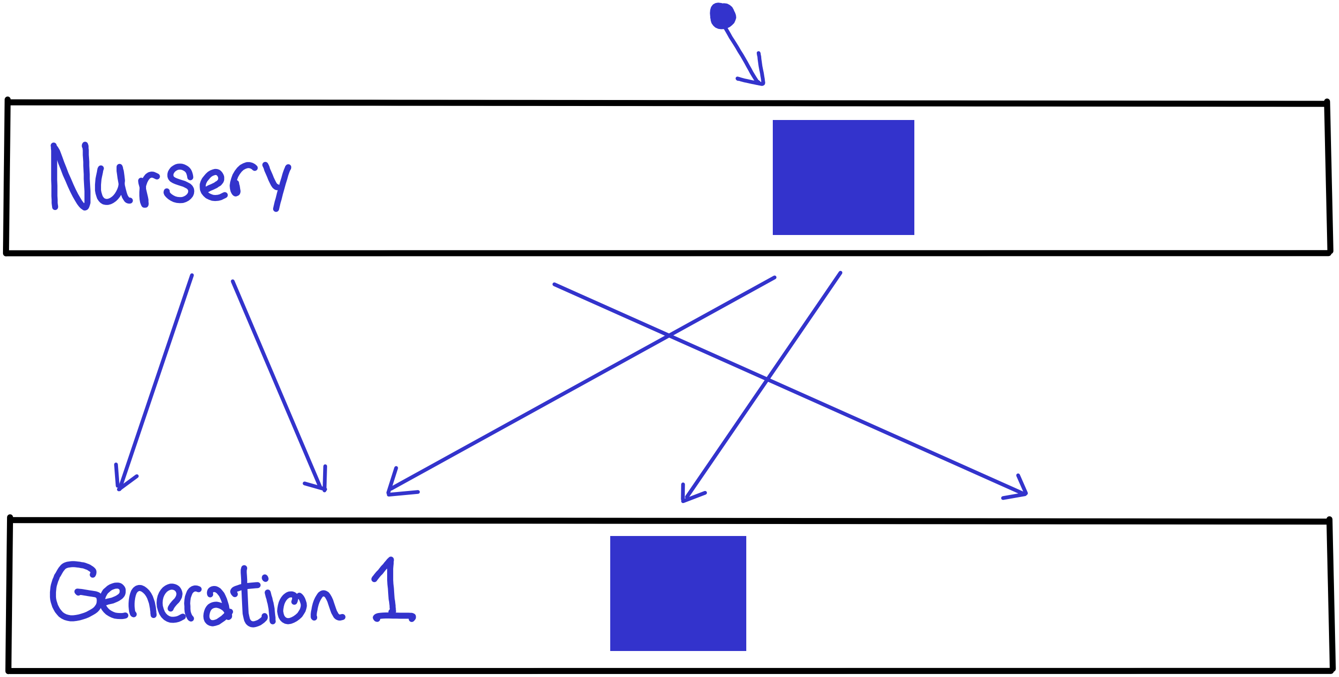
What about Purity?



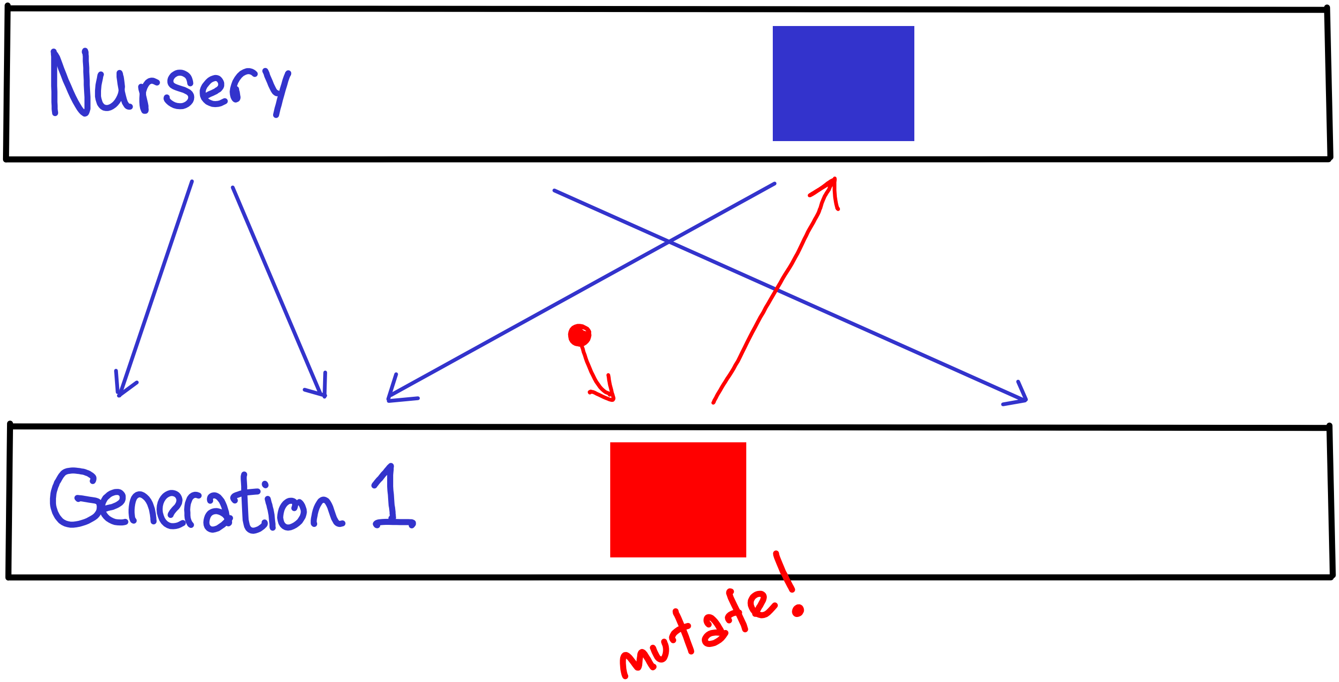
→ Write Barriers

→ Parallel Garbage Collection

I promised you I would talk about the unique benefits we get for writing an RTS for Haskell code, and now's the time. I'm going to talk how Haskell's purity can be used to good effect.

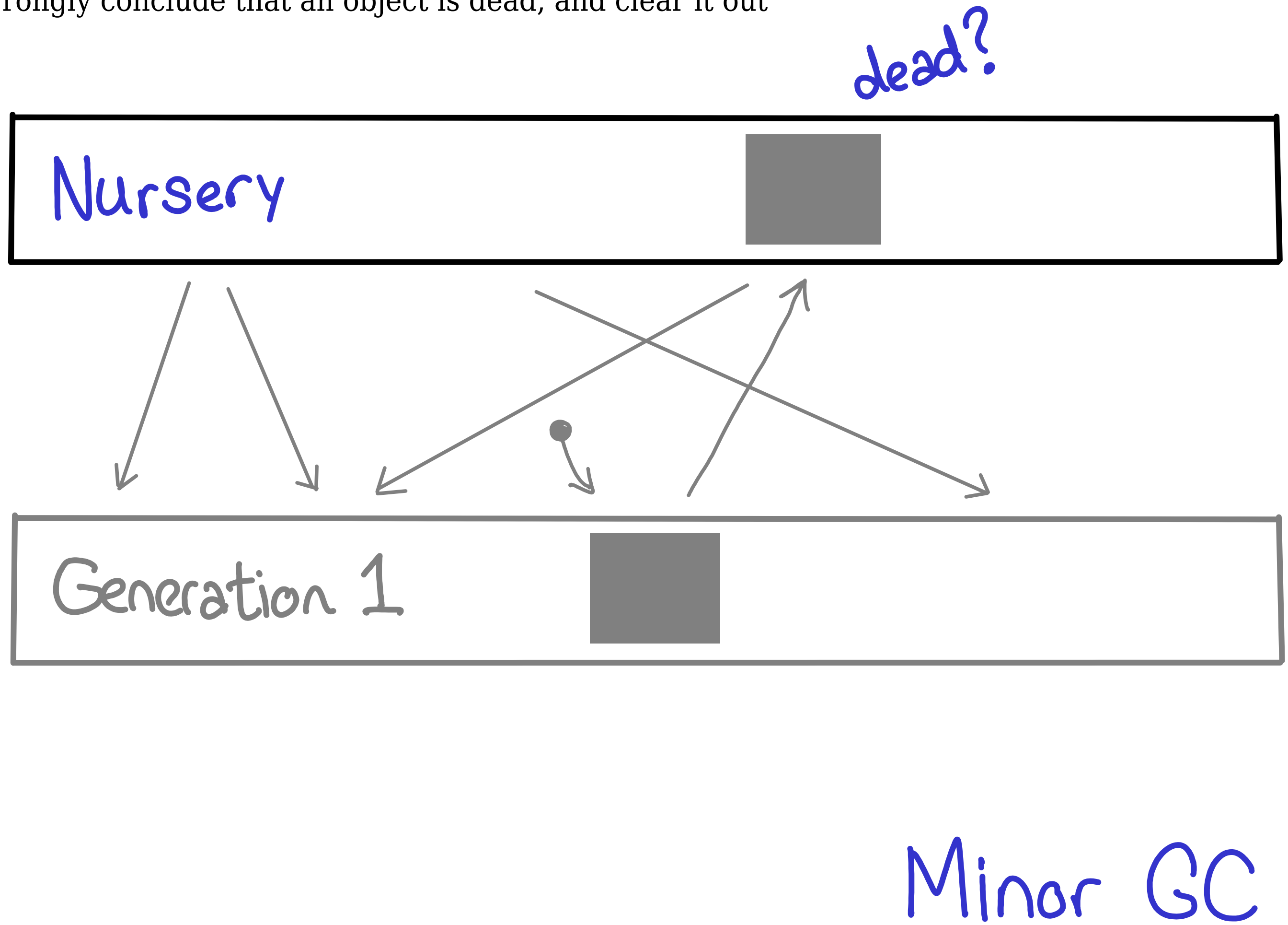


To talk about write barriers, we have to first go back to our picture of generations in the heap, and recall the invariant we imposed, which is that pointers are only allowed to flow from the nursery to the first generation, and not vice versa.

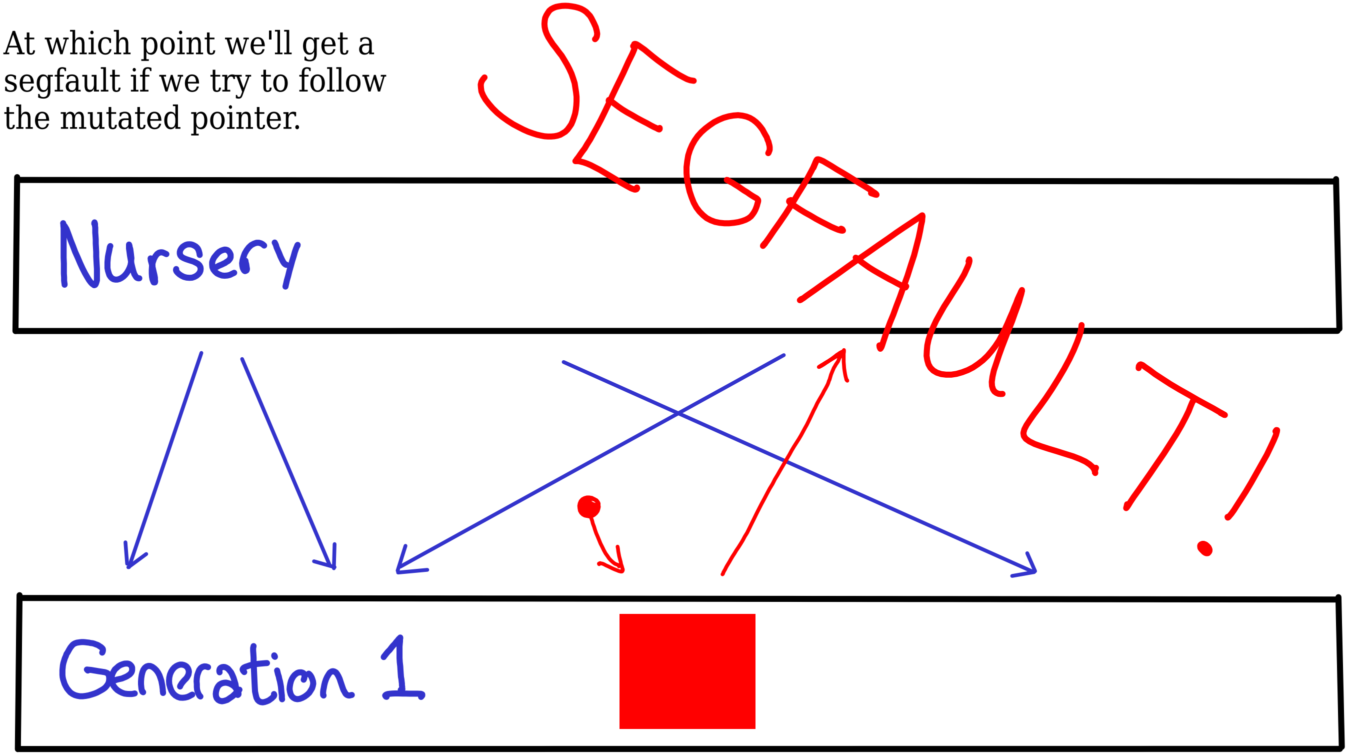


When mutation comes into the picture, there's a problem: we can mutate a pointer in an old generation to point to an object in the nursery.

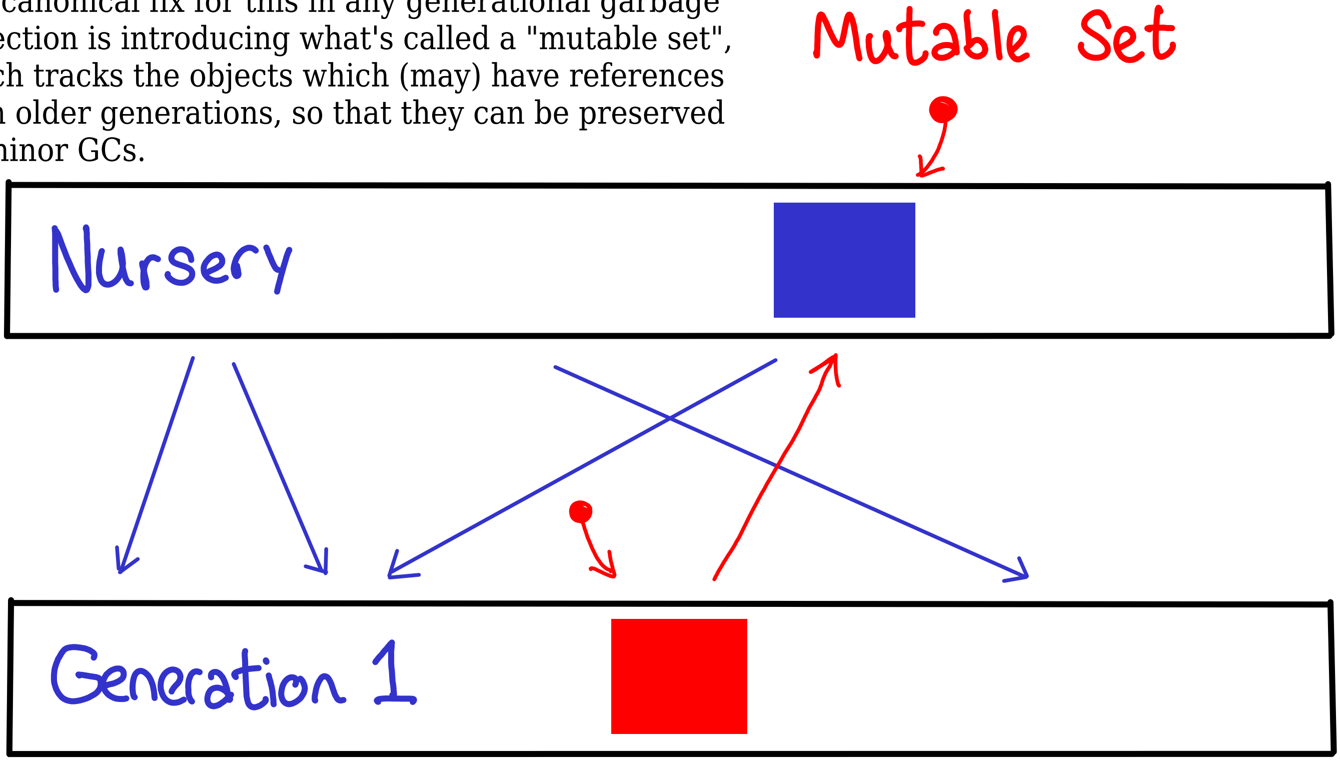
If we perform a minor garbage collection, we may wrongly conclude that an object is dead, and clear it out



At which point we'll get a segfault if we try to follow the mutated pointer.



The canonical fix for this in any generational garbage collection is introducing what's called a "mutable set", which tracks the objects which (may) have references from older generations, so that they can be preserved on minor GCs.



There is a big design space in how to build your mutable sets, with differing trade offs. If garbage collection is black magic, the design of your mutable set mechanism probably serves as the bulk of the problem.

Why is generational GC hard?

This.

For example, if you're Java, your programmers are modifying pointers on the heap ALL THE TIME, and you really, really, really need to make sure adding something to the mutable set is as fast as possible. So if you look at, say, the JVM, there are sophisticated card marking schemes to minimize the number of extra instructions that need to be done when you mutate a pointer.

Why is generational GC hard in Java?

This.

Purity to the rescue

Haskell doesn't have many of these optimizations (simplifying its GC and code generation)... and, to a large extent, it doesn't need them!

- Mutation is rare

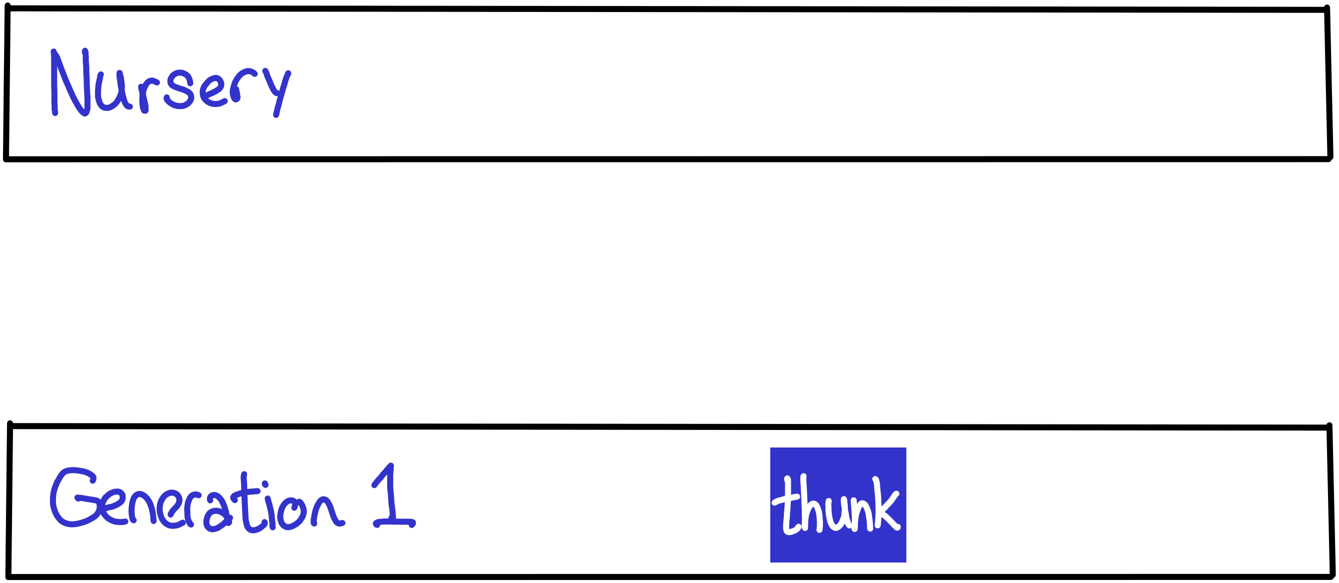
Idiomatic Haskell code doesn't mutate. Most executing code is computing or allocating memory. This means that slow mutable references are less of a "big deal."

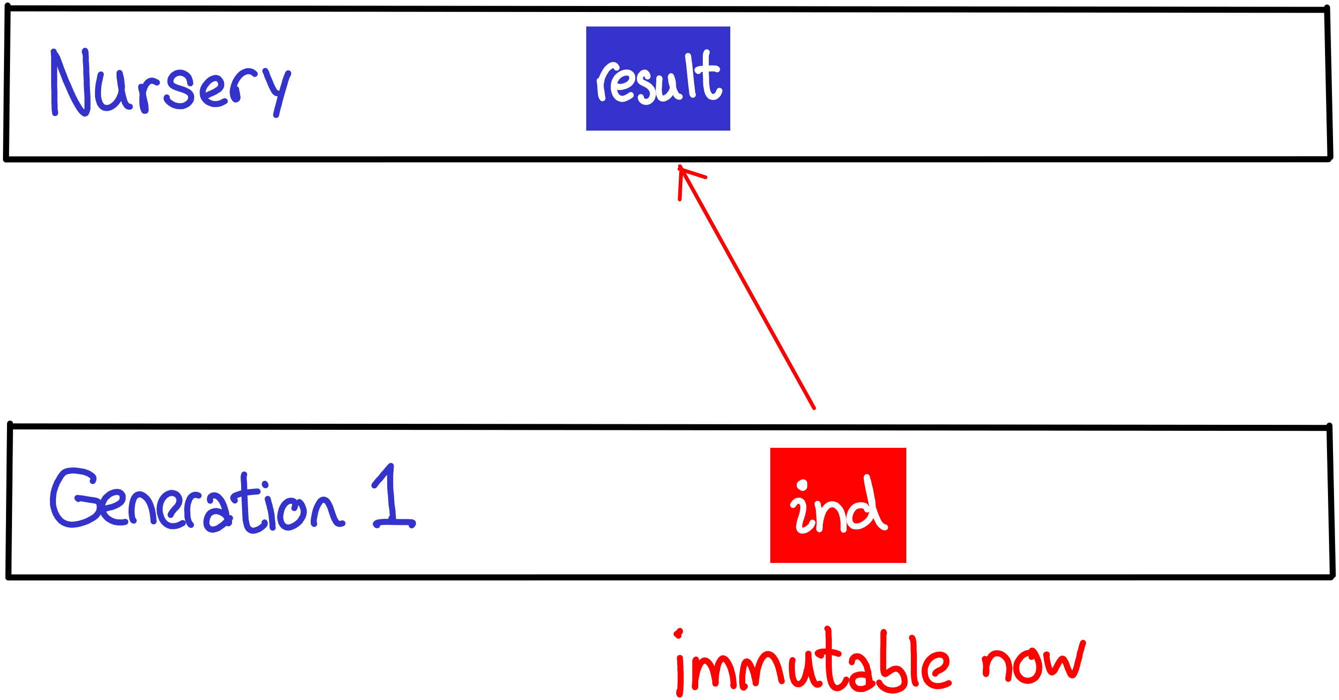
- IORefs are slow anyway

Perhaps this is not a good excuse, but IORefs are already pretty sure, because their current implementation imposes a mandatory indirection. "You didn't want to use them anyway."

- Laziness is a special kind of mutation

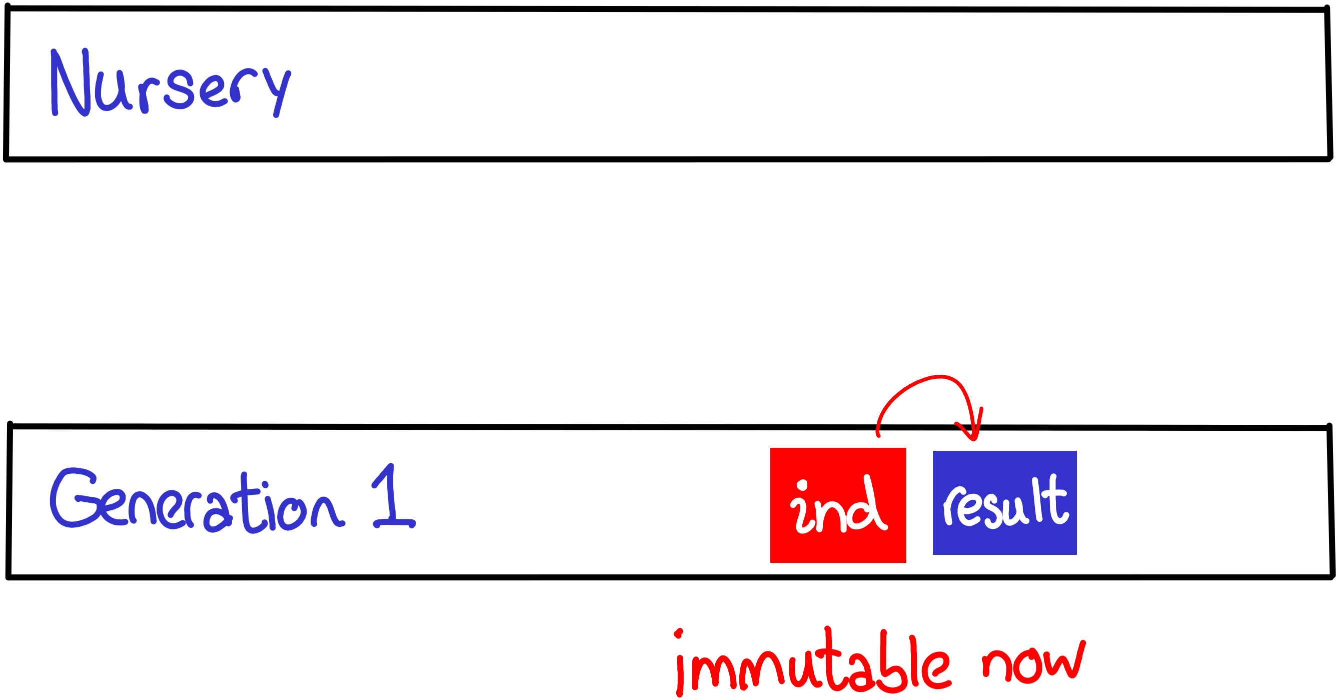
Now, it is patently not true that Haskell code does not, under the hood, do mutation: in fact, we do a lot of mutation, updating thunks with their actual computed values. But there's a trick we can play in this case.


Nursery
Generation 1
thunk



Once we evaluate a thunk, we mutate it to point to the true value precisely once. After this point, it is immutable.

Promotion



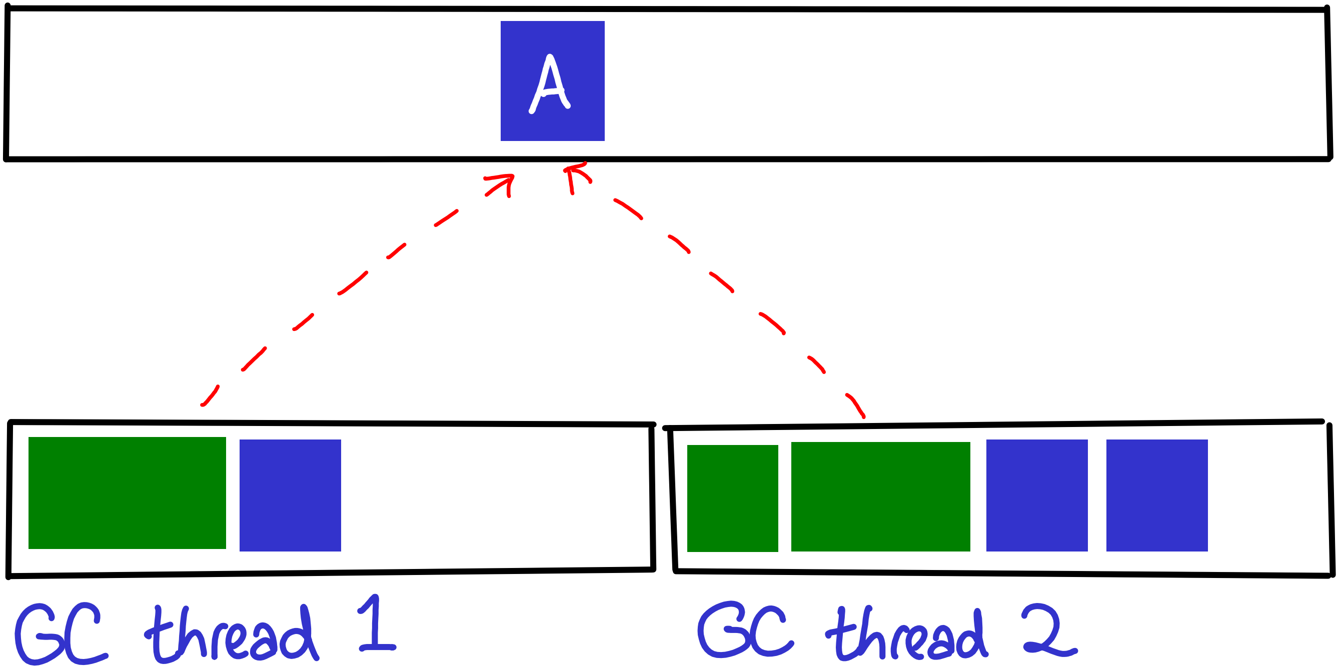
Since it is immutable, result cannot possibly become dead until ind becomes dead. So, although we must add result to the mutable set, upon the next GC, we can just immediately promote it to the proper generation.

Parallel GC

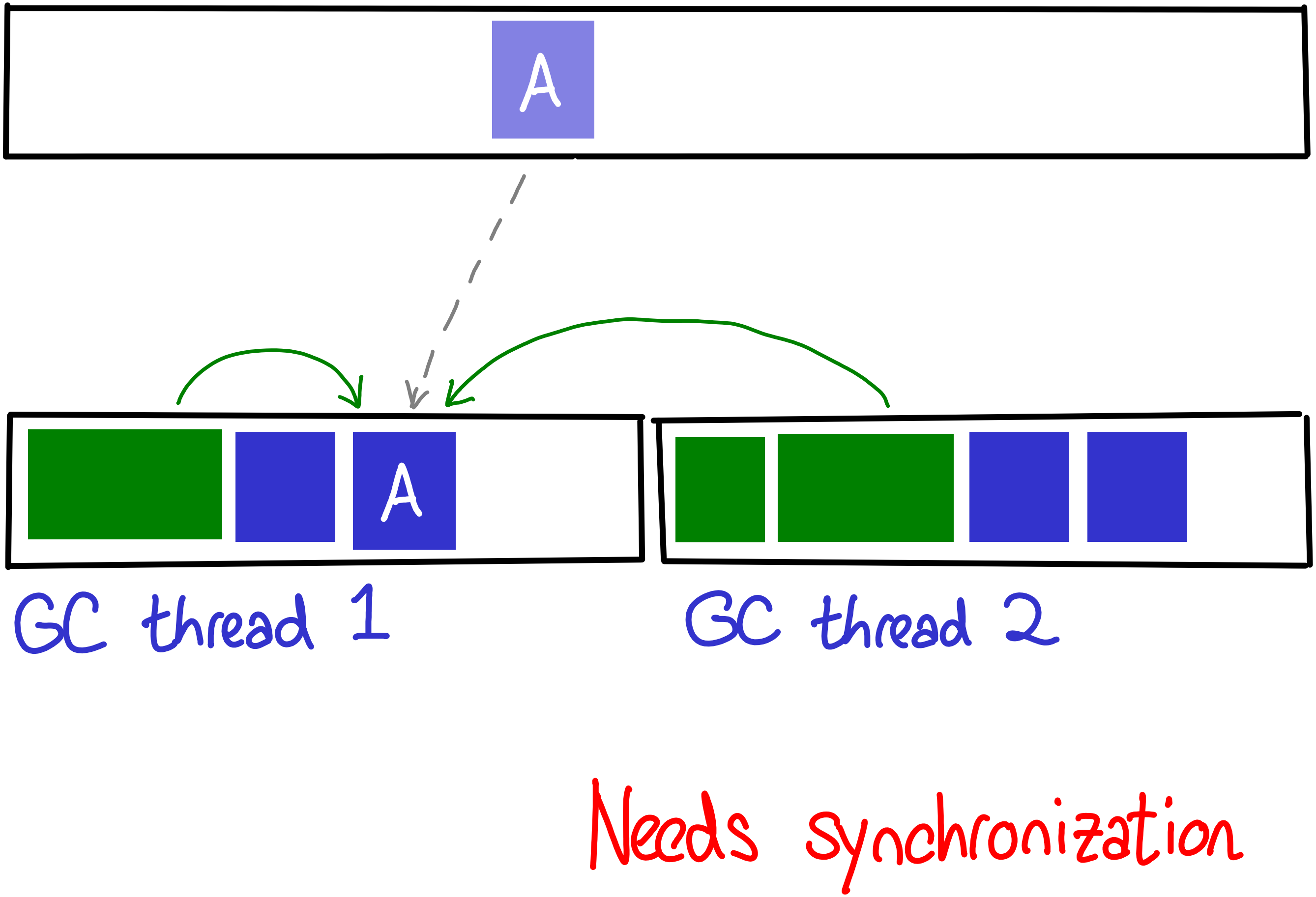
Haskell programs spend a lot of time garbage collecting, and while running the GC in parallel with the mutators in the program is a hard problem, we can parallelize GC. The basic idea is that the scavenging process (that's where we process objects which are known to be live to pull in the things that they point to) can be parallelized.

Idea: Split heap into blocks, and parallelize the scavenging process

Now, here's a problem. Suppose that you have two threads busily munching away on their live sets, and they accidentally end up processing two pointers which point to the same object.

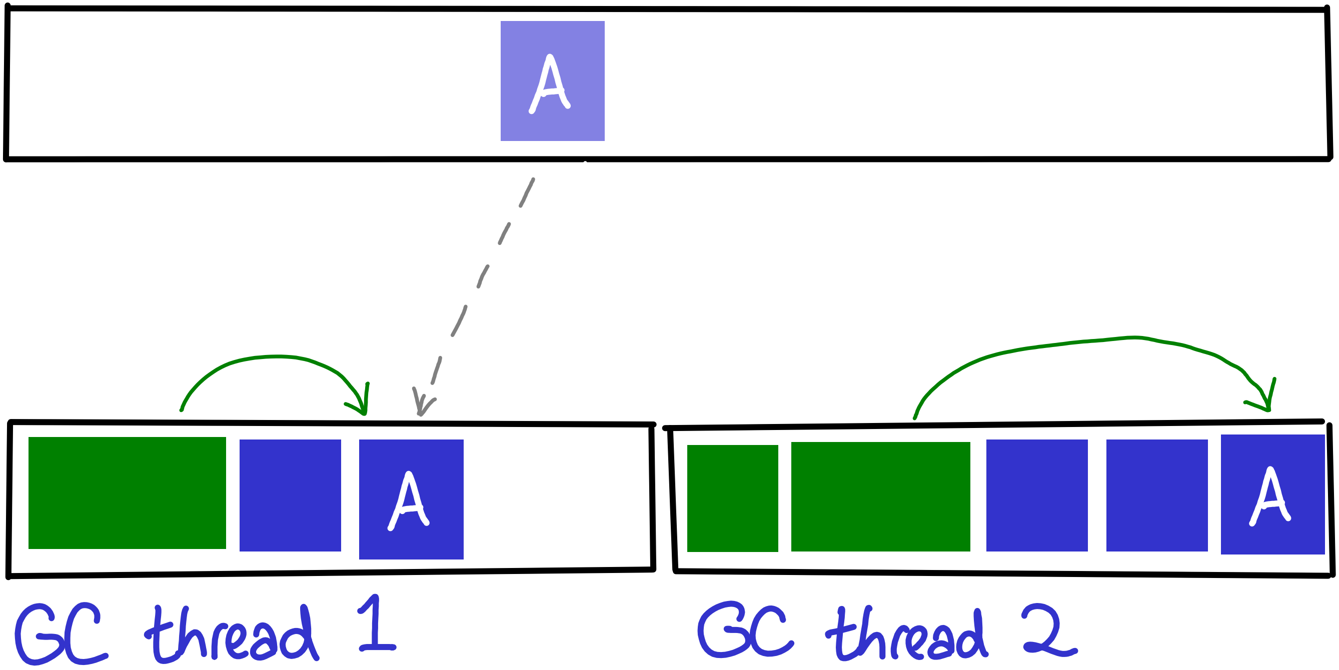


In an ideal world, only one of the threads would actually evacuate the object, and the other thread would update its pointer to point to its sole copy. Unfortunately, to do this, we'd have to add synchronization here. That's a BIG DEAL; most accesses to the heap here have no races and we really don't want to pay the cost of synchronization.



If A is immutable...

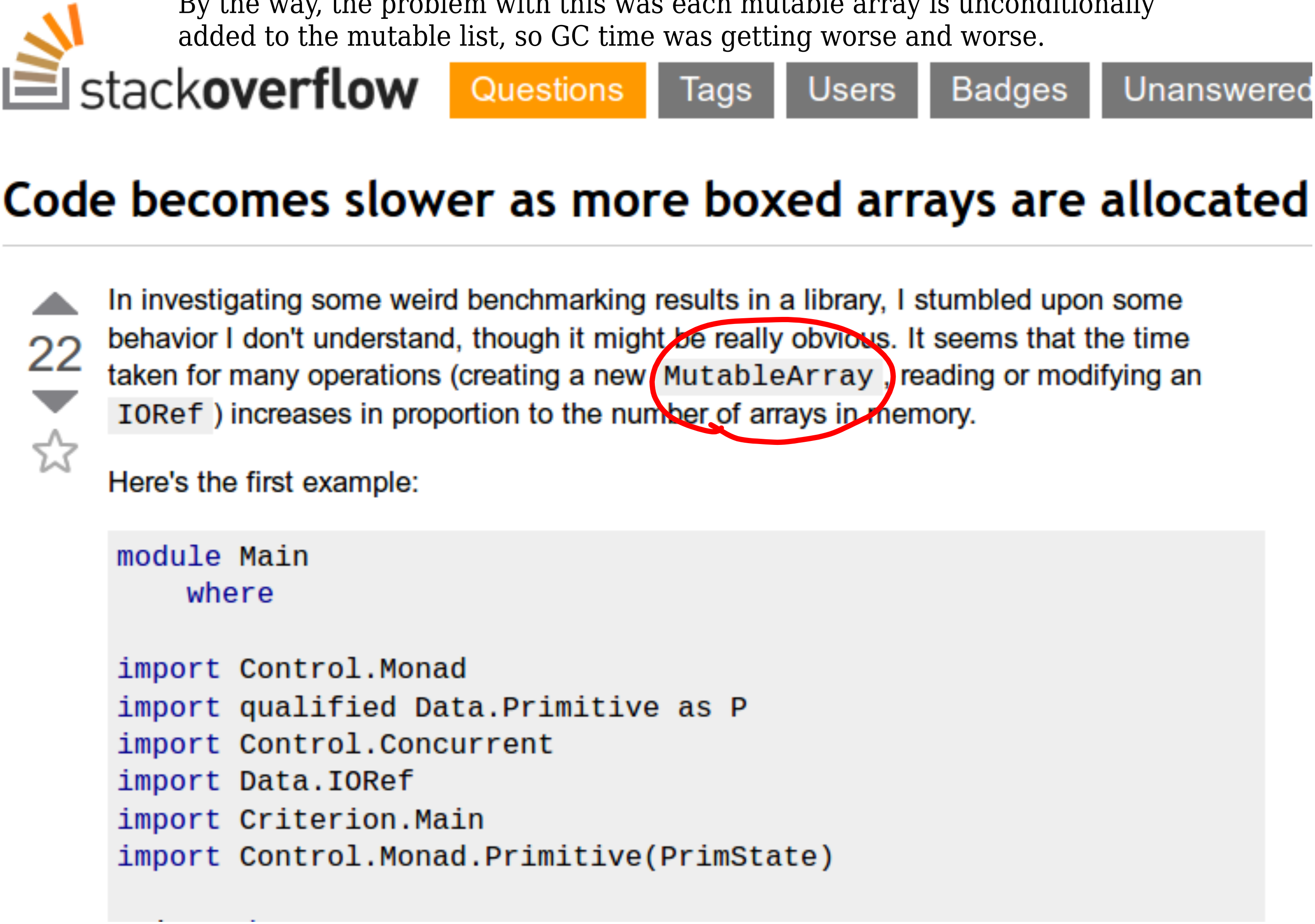
If A is an immutable object, there's an easy answer: just let the two threads race, and end up with duplicates of the object! After all, you can't observe the difference.



...observationally indistinguishable!

Purity = Flexibility

By the way, the problem with this was each mutable array is unconditionally added to the mutable list, so GC time was getting worse and worse.



Code becomes slower as more boxed arrays are allocated

In investigating some weird benchmarking results in a library, I stumbled upon some behavior I don't understand, though it might be really obvious. It seems that the time taken for many operations (creating a new `MutableArray`, reading or modifying an `IORef`) increases in proportion to the number of arrays in memory.

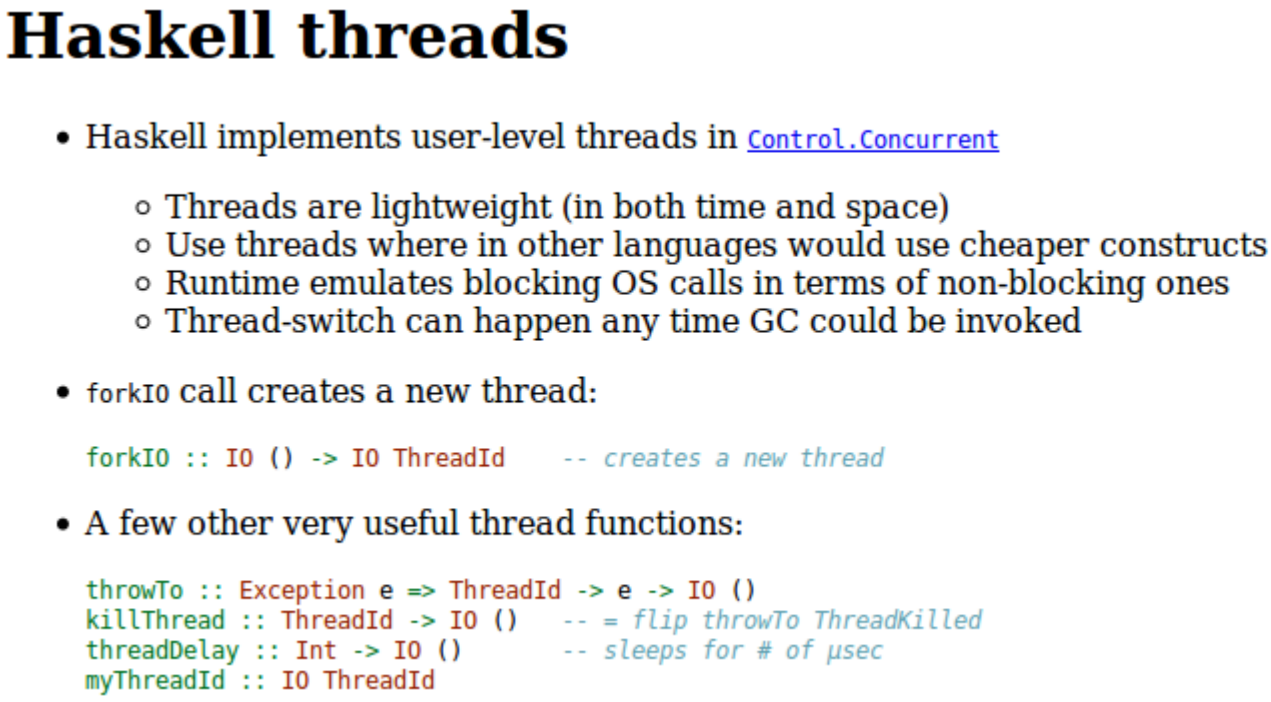
Here's the first example:

```
module Main
    where

import Control.Monad
import qualified Data.Primitive as P
import Control.Concurrent
import Data.IORef
import Criterion.Main
import Control.Monad.Primitive(PrimState)
```

For the second part of this lecture, I want to talk about the scheduler.

Scheduler

In case your final project doesn't involve any concurrency, it's worth briefly recapping the user visible interface for threads.



Haskell threads

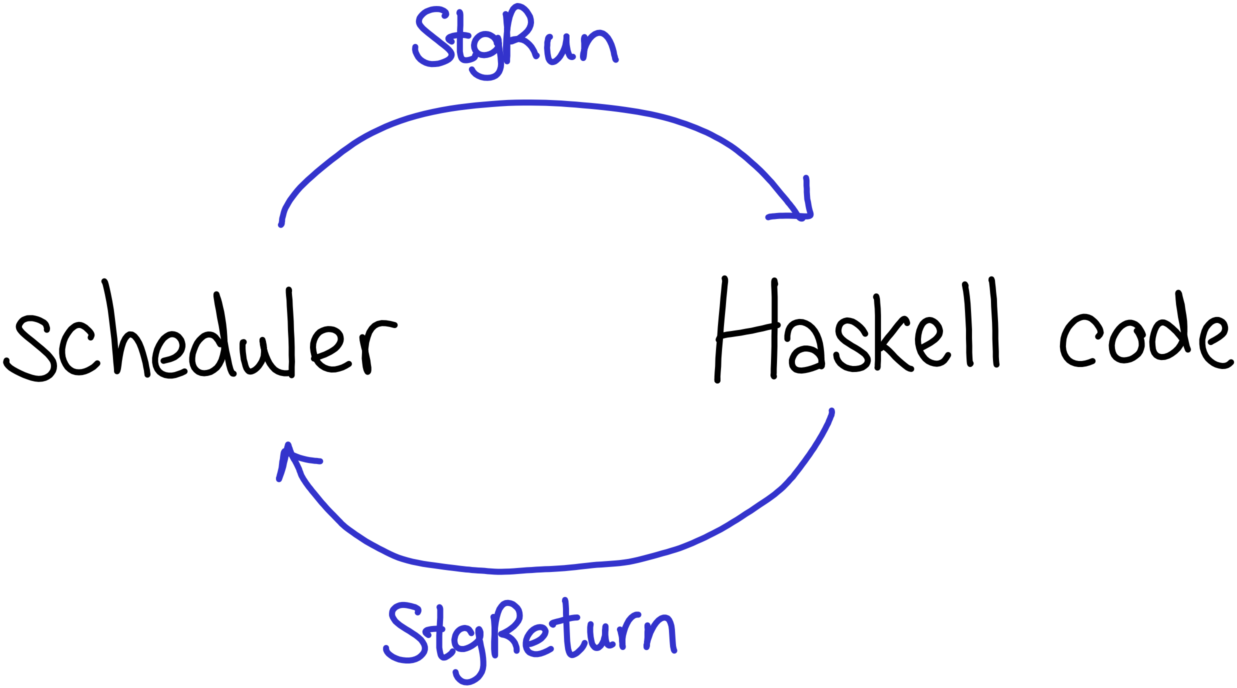
- Haskell implements user-level threads in `Control.Concurrent`
  - Threads are lightweight (in both time and space)
  - Use threads where in other languages would use cheaper constructs
  - Runtime emulates blocking OS calls in terms of non-blocking ones
  - Thread-switch can happen any time GC could be invoked
- `forkIO` call creates a new thread:

```
forkIO :: IO () -> IO ThreadId    -- creates a new thread
```

- A few other very useful thread functions:

```
throwTo :: Exception e => ThreadId -> e -> IO ()
killThread :: ThreadId -> IO ()   -- = flip throwTo ThreadKilled
threadDelay :: Int -> IO ()       -- sleeps for # of μsec
myThreadId :: IO ThreadId
```

The scheduler mediates the loop between running Haskell code, and getting kicked back to the RTS (where we might run some other Haskell code, or GC, etc...)

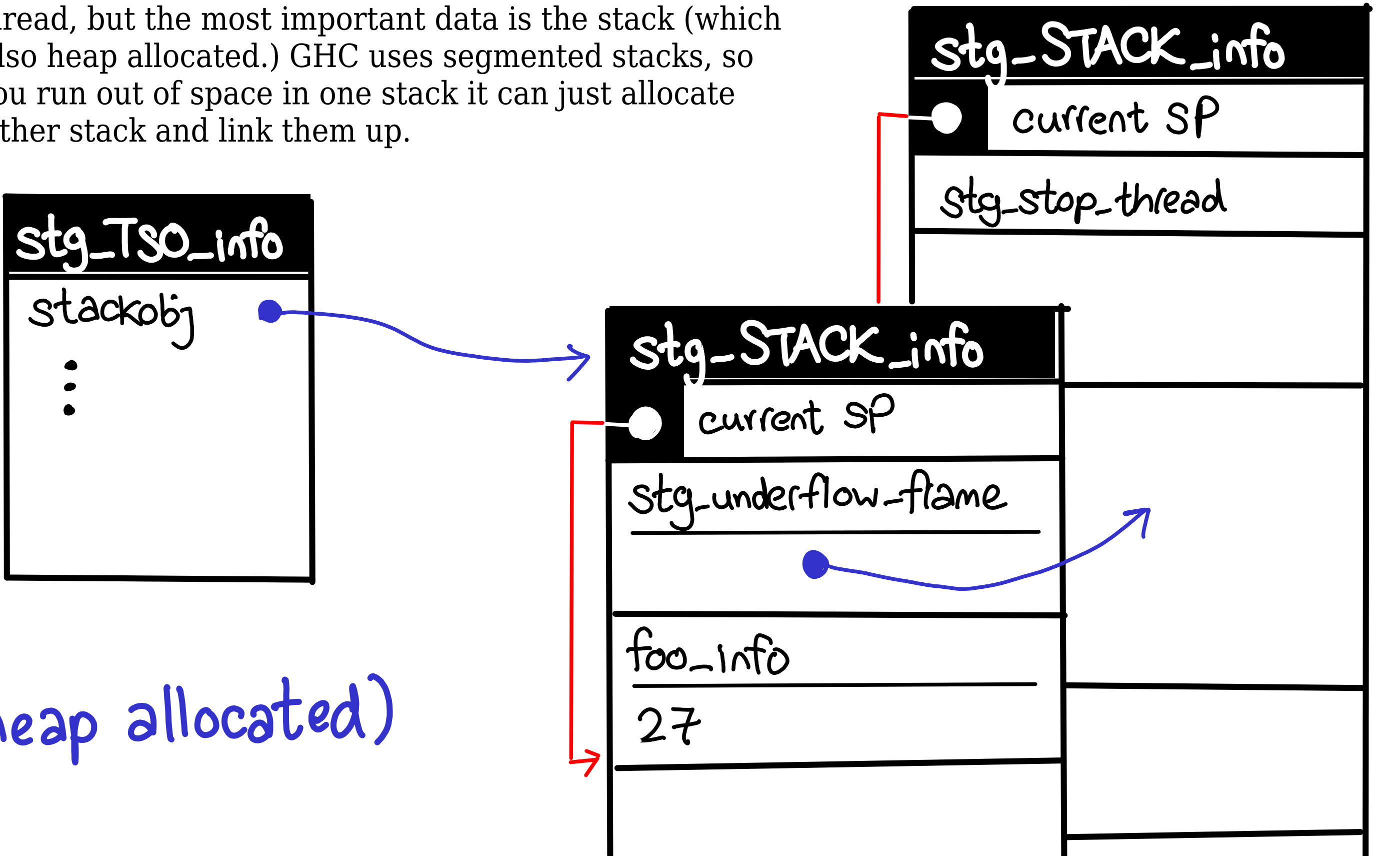


```
mk_exit()
    entry:
        Hp = Hp + 16;
        if (Hp > HpLim) goto gc;

        v::I64 = I64[R1] + 1;

        I64[Hp - 8] = GHC_Types_I_con_info;
        I64[Hp + 0] = v::I64;

        R1 = Hp;
        Sp = Sp + 8;
        jump (I64[Sp + 0]) ();

    gc: HpAlloc = 16;
        jump stg_gc_enter_1 ();
}
```

set to zero

Anatomy of a thread

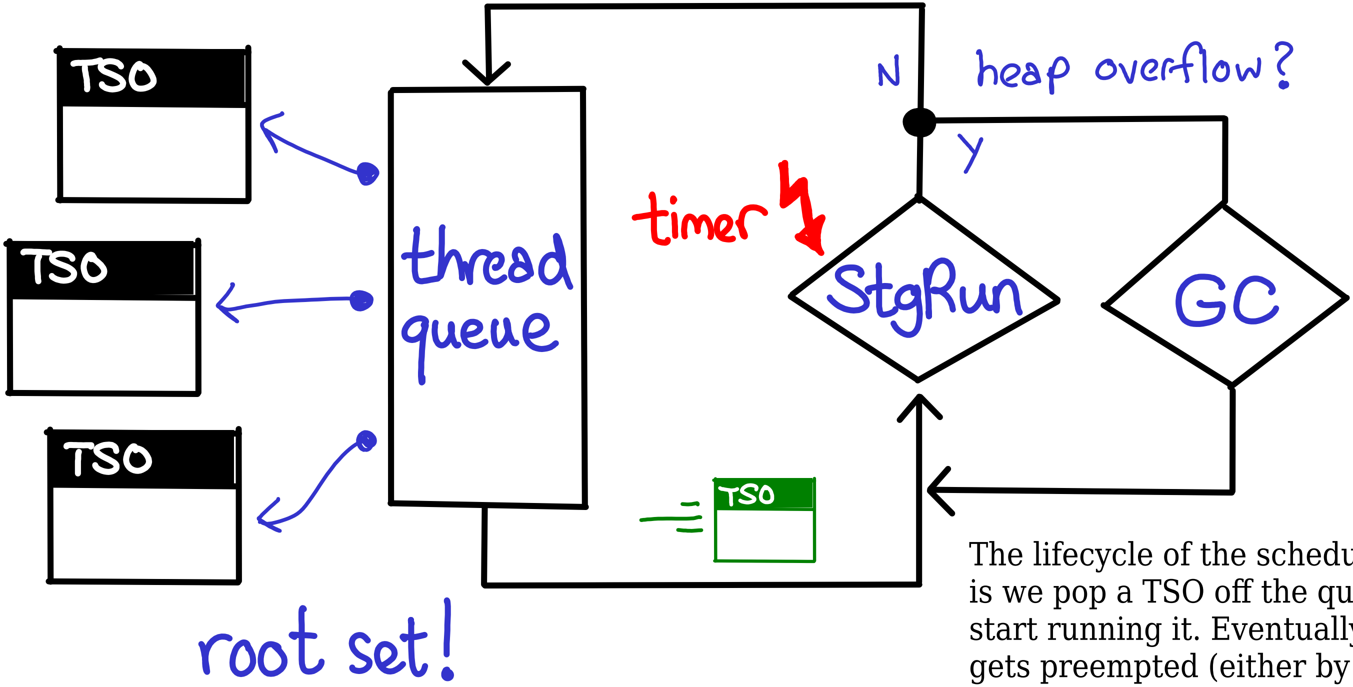
So, what is a thread anyway? Very simply, a thread is just another heap object! There are a number of metadata associated with a thread, but the most important data is the stack (which is also heap allocated.) GHC uses segmented stacks, so if you run out of space in one stack it can just allocate another stack and link them up.



(heap allocated)

In a single-threaded Haskell program, these TSO objects are managed by a thread queue.

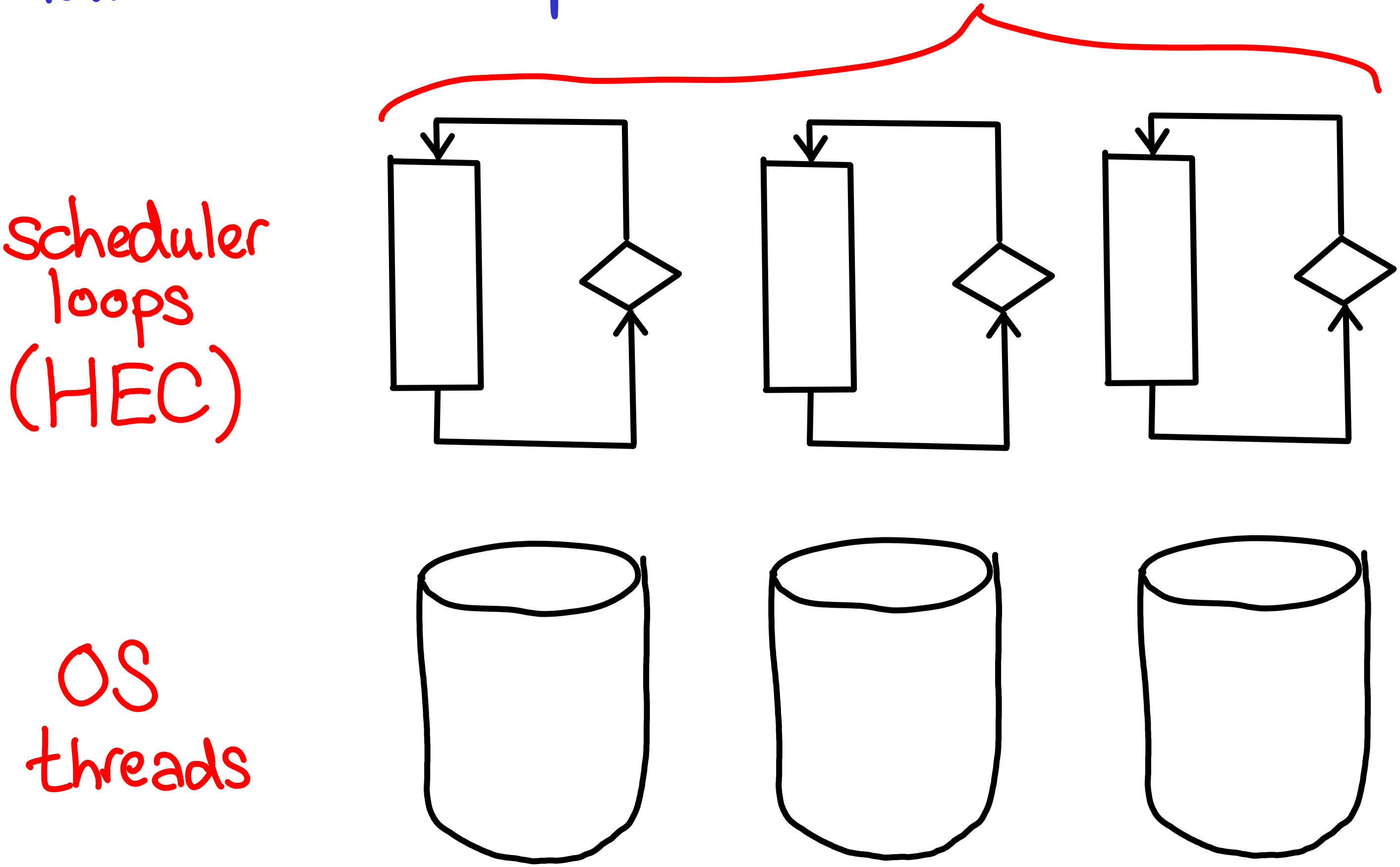
Single-threaded operation



Scheduler Loop

The lifecycle of the scheduler loop is we pop a TSO off the queue and start running it. Eventually, it gets preempted (either by running out of memory, getting flagged by the timer, or blocking) in which case we pop out and run the GC or head to the next thread queue.

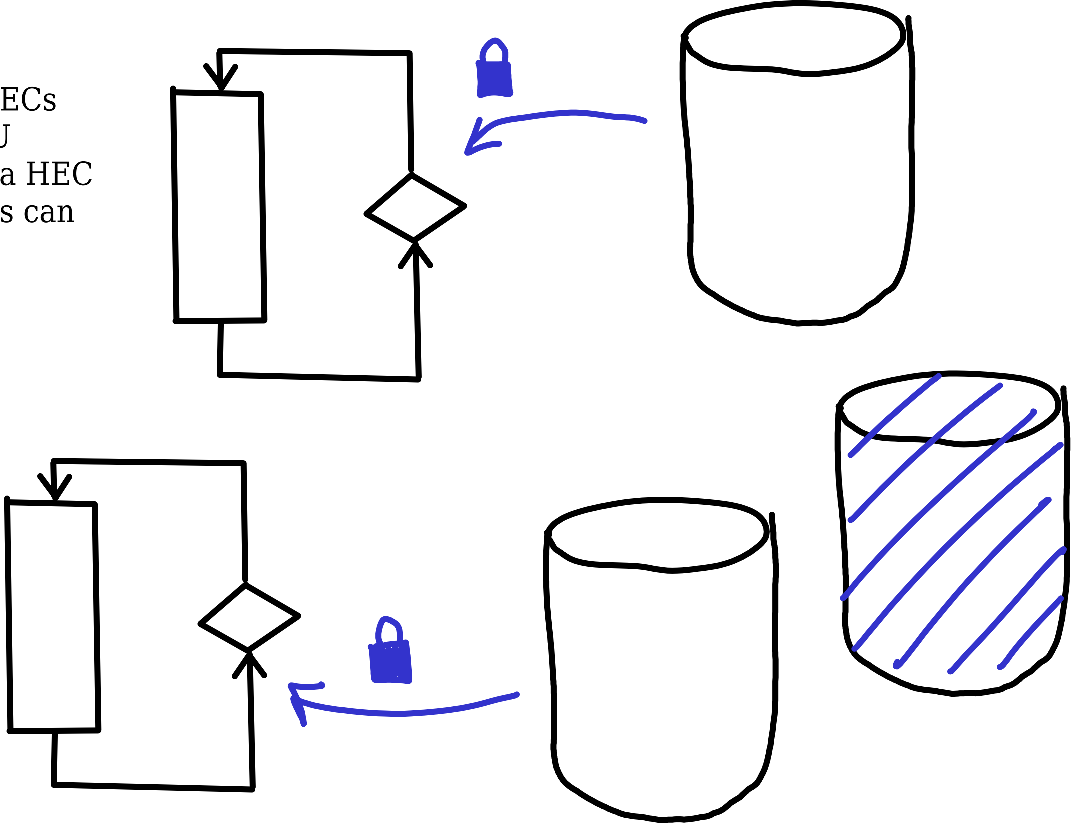
Multi-threaded operation -N3



Multithreaded operation simply involves allocating one of these schedule loops to each operating system core you want to run. We refer to a scheduler loop as a HEC.

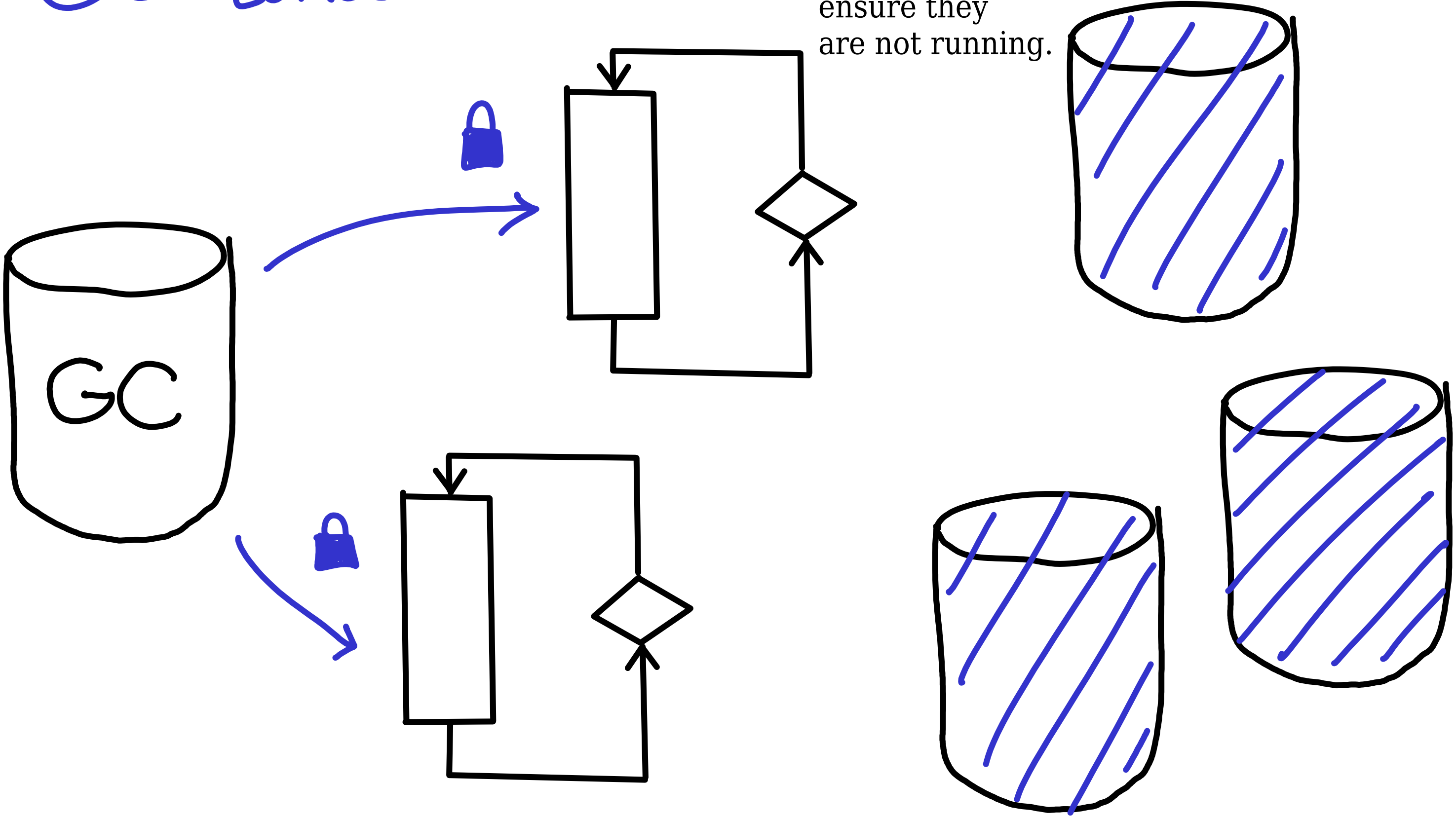
HECs are locks

A useful interpretation of HECs is that they are locks: a CPU core can take out a lock on a HEC in which case no other cores can use it.



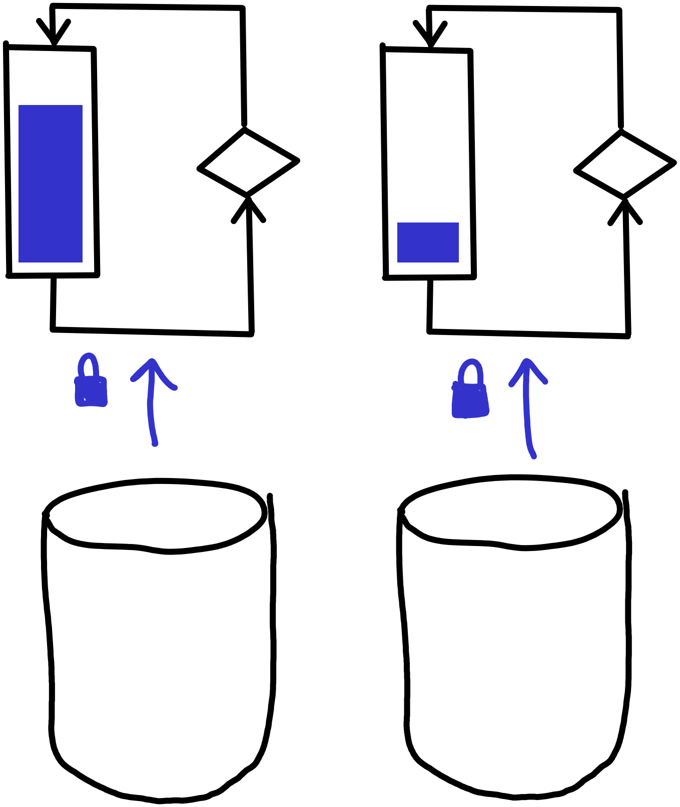
GC takes all locks

Because garbage collection cannot run concurrently with Haskell code, the GC process takes out locks on all HECs to ensure they
are not running.



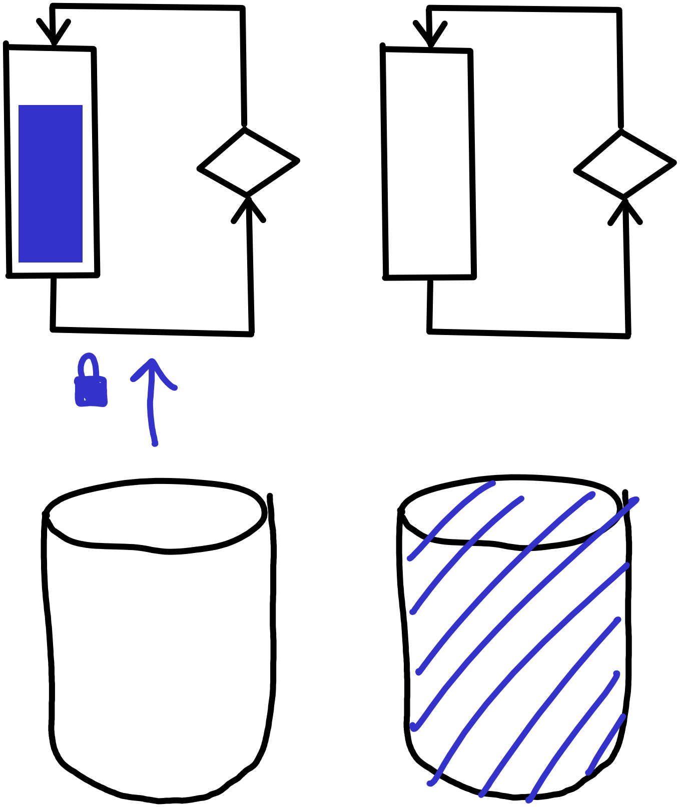
Work imbalance

One problem with running multiple scheduler loops is that their respective event queues can get unbalanced.



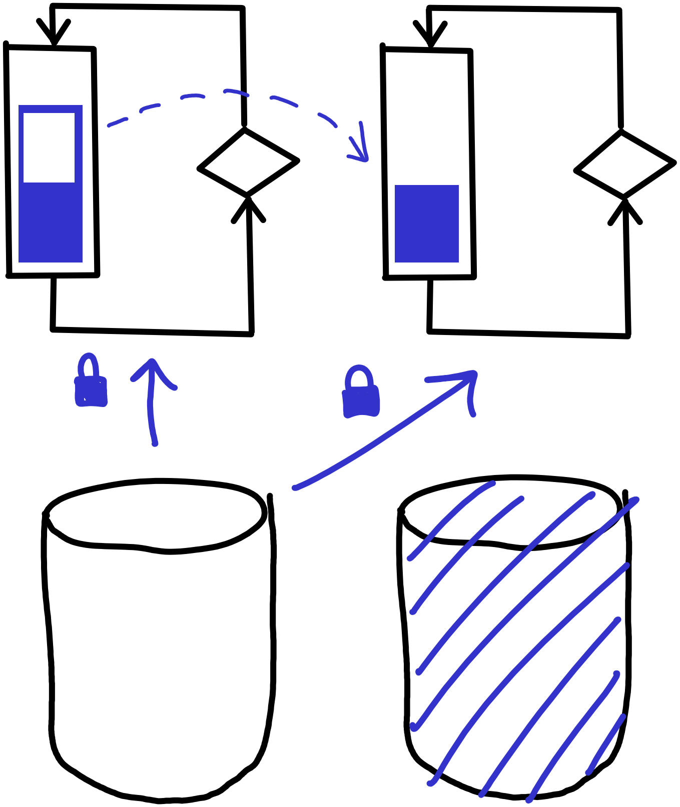
Work imbalance

If a core runs out of work to do, it releases the HEC and goes to sleep.



Work imbalance

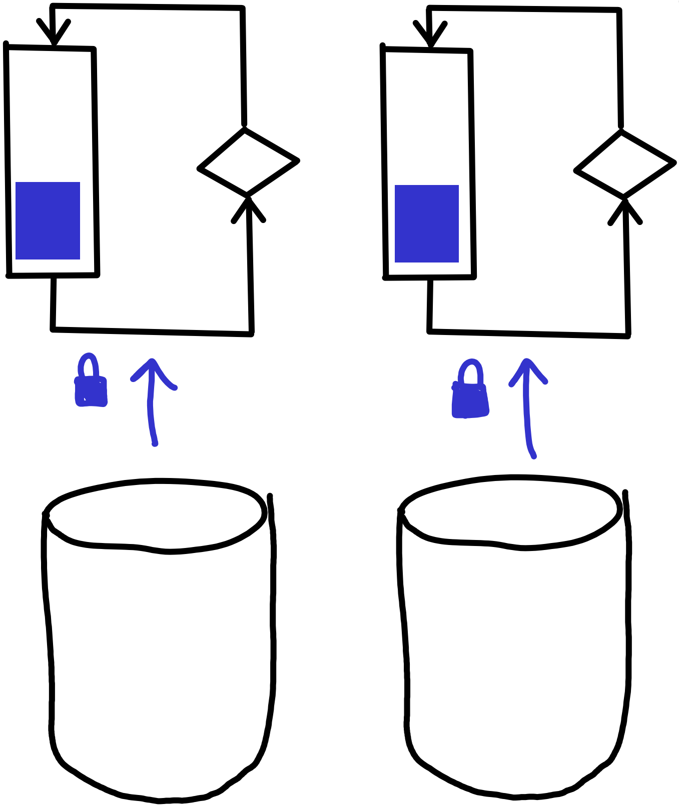
Every time we come around the schedule loop, a core does a quick check to see if there are any free HECs. If there are, it snarfs them up, and then distributes some of its own work to those queues. No heavy synchronization necessary!



Work imbalance

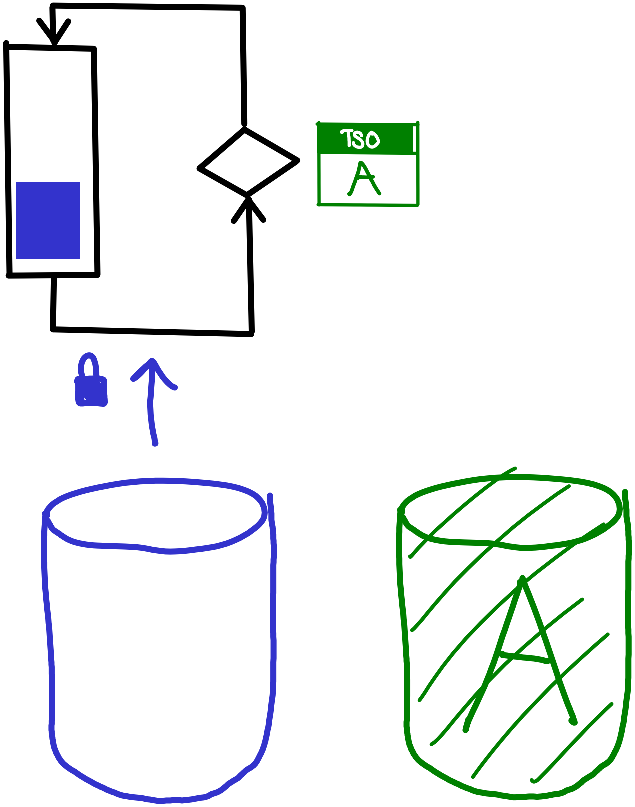
Throughput First!

This scheme is not very fair, and you won't get very good latency guarantees from it, but it is great for throughput.

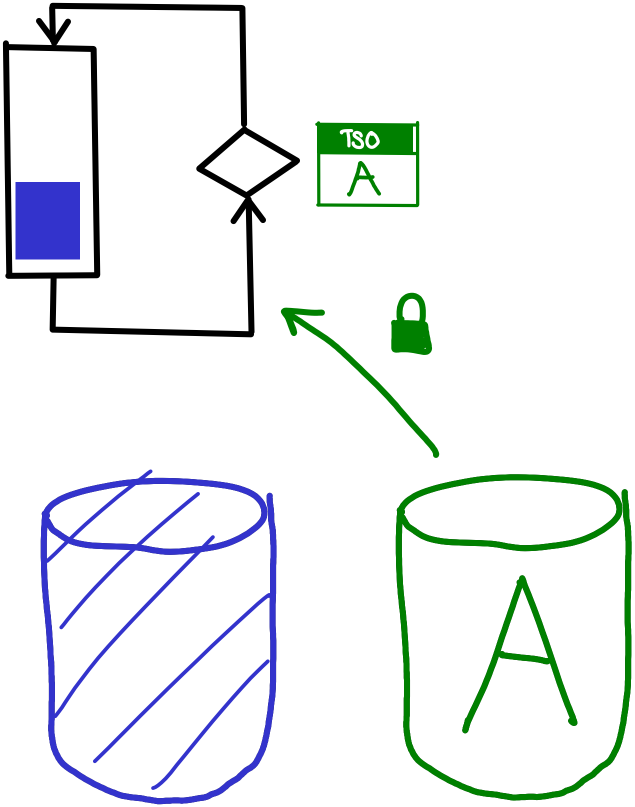


Bound threads

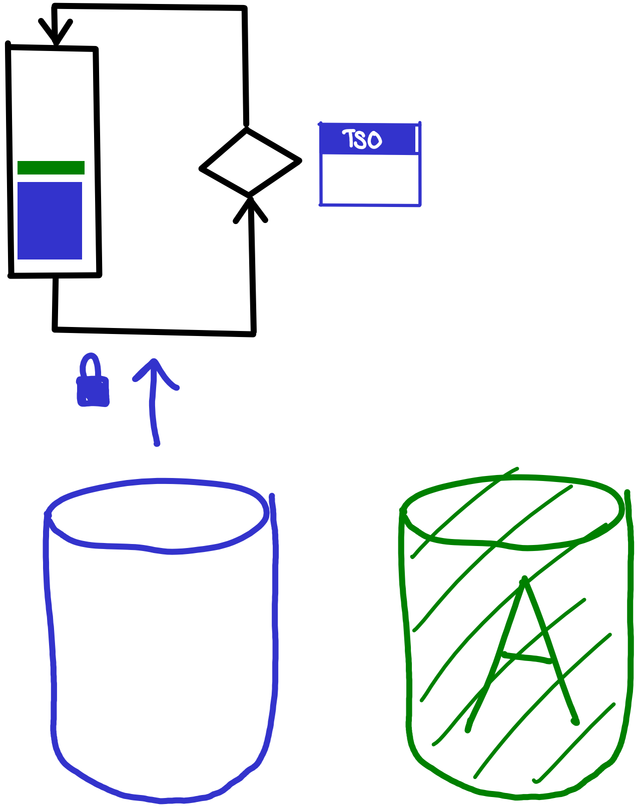
Here's how bound threads are implemented with HECs.



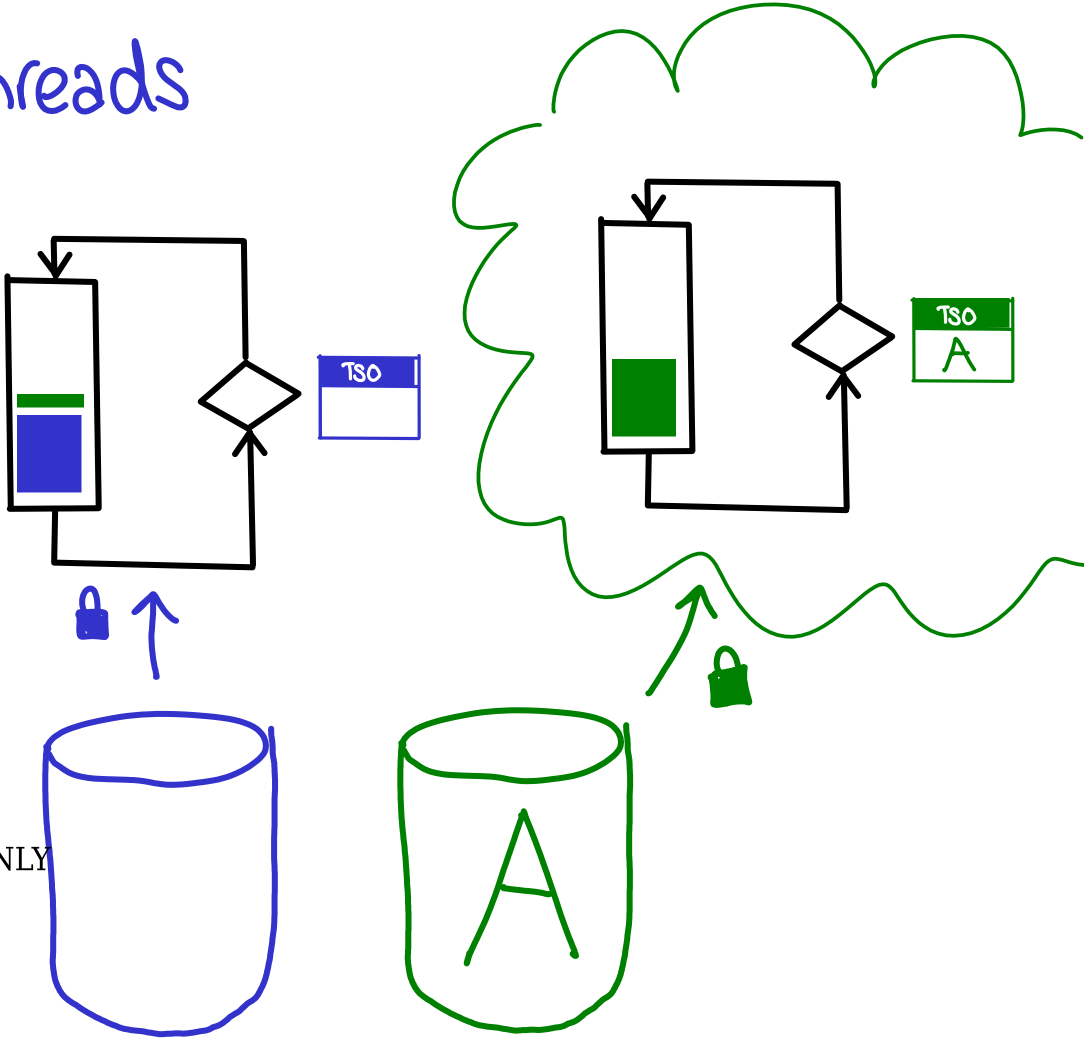
Bound threads



Bound threads



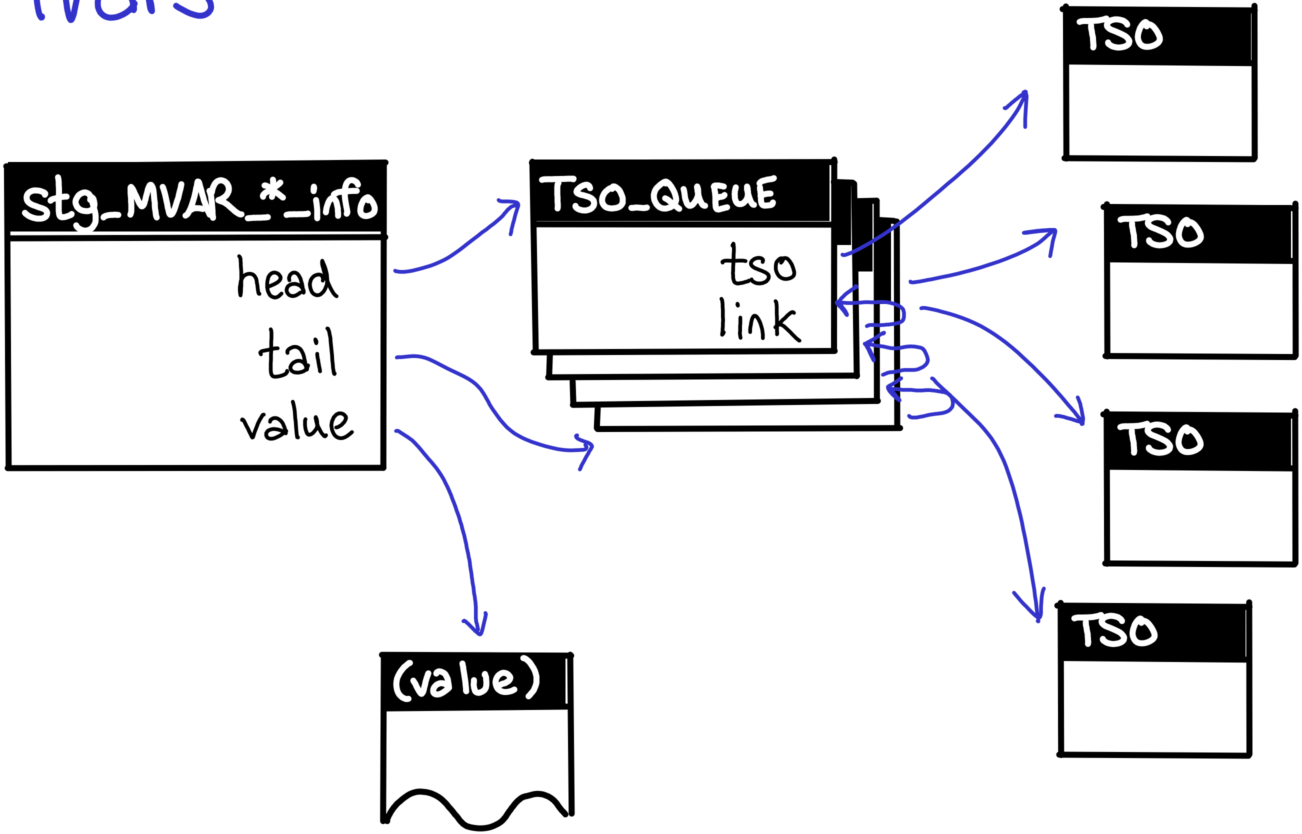
Bound threads



You want to avoid running ordinary TSOs on bound threads, since they are the ONLY thread that can service TSOs bound to that thread.

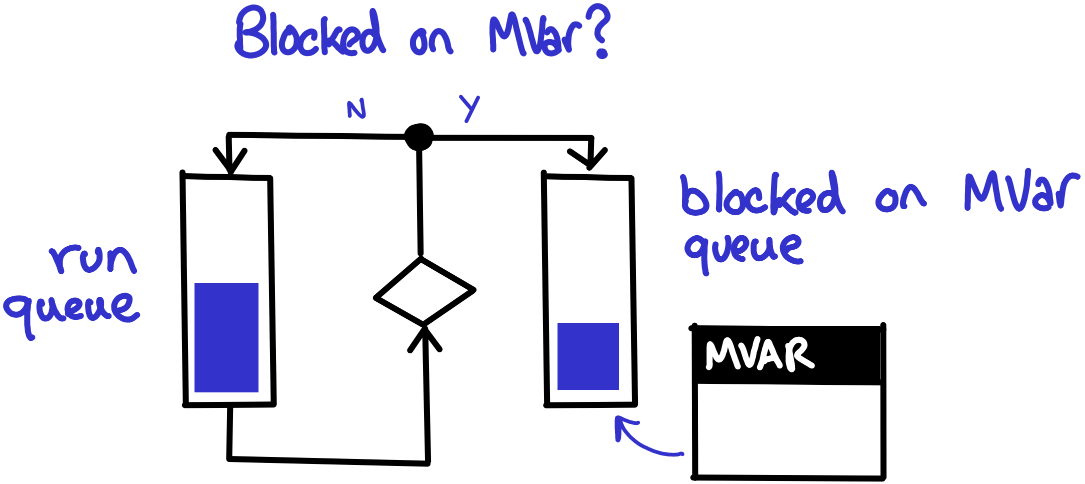
MVars

Let's talk about how MVars are implemented.



MVars

MVars essentially contian another thread queue, the "blocked on this MVar" thread queue. When you block on an MVar, a TSO is removed from the main run queue and put on the MVar queue.



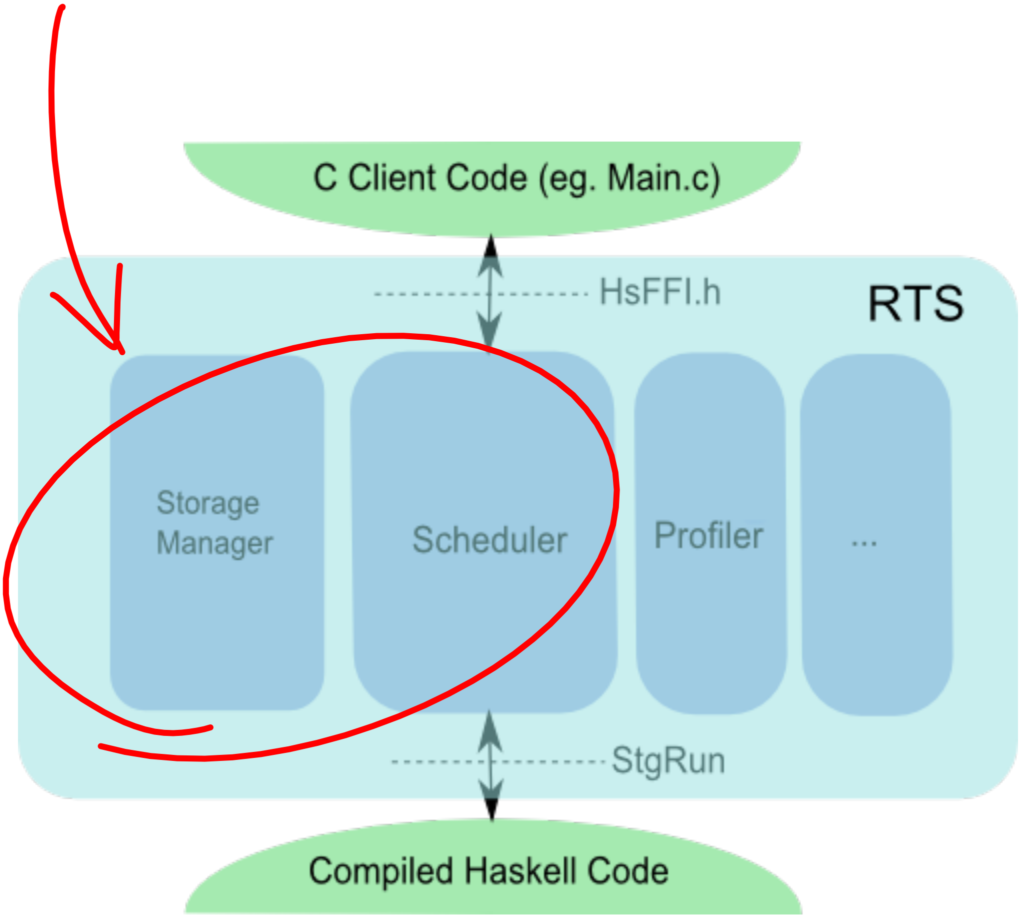
Fun fact: If the MVar becomes garbage, the threads in its queue die too

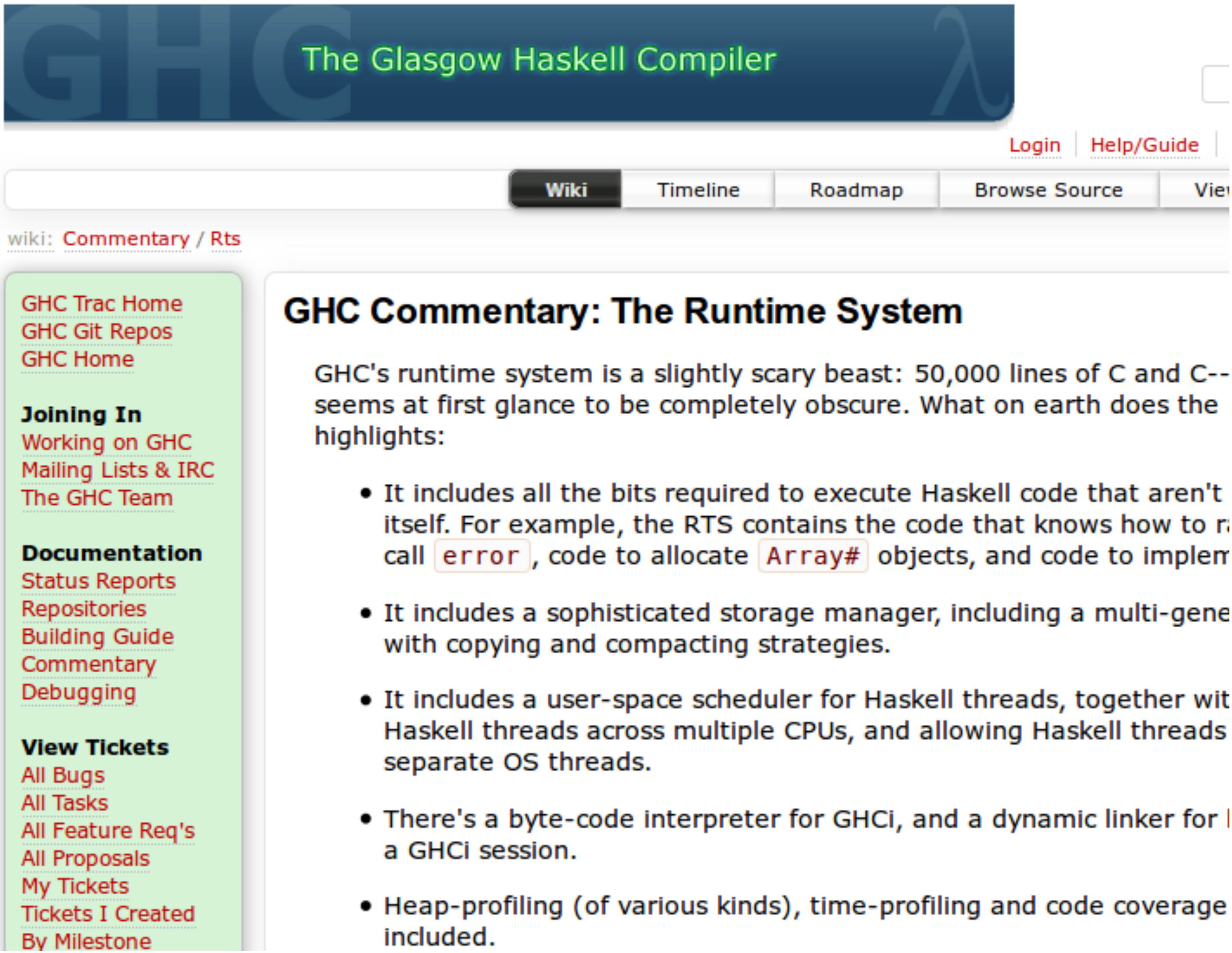
Scheduler in a nutshell

Everything lives on the heap

Small initial stack segments
= cheap green threads

Purity = most code threadsafe
by default


C Client Code (eg. Main.c)
HsFFI.h
RTS
Storage Manager
Scheduler
Profiler
...
StgRun
Compiled Haskell Code



The Glasgow Haskell Compiler

Login | Help/Guide |

Wiki | Timeline | Roadmap | Browse Source | Vie

wiki: Commentary / Rts

GHC Trac Home
GHC Git Repos
GHC Home

Joining In
Working on GHC
Mailing Lists & IRC
The GHC Team

Documentation
Status Reports
Repositories
Building Guide
Commentary
Debugging

View Tickets
All Bugs
All Tasks
All Feature Req's
All Proposals
My Tickets
Tickets I Created
By Milestone

GHC Commentary: The Runtime System

GHC's runtime system is a slightly scary beast: 50,000 lines of C and C--
seems at first glance to be completely obscure. What on earth does the
highlights:

- It includes all the bits required to execute Haskell code that aren't
itself. For example, the RTS contains the code that knows how to ra
call `error`, code to allocate `Array#` objects, and code to implem

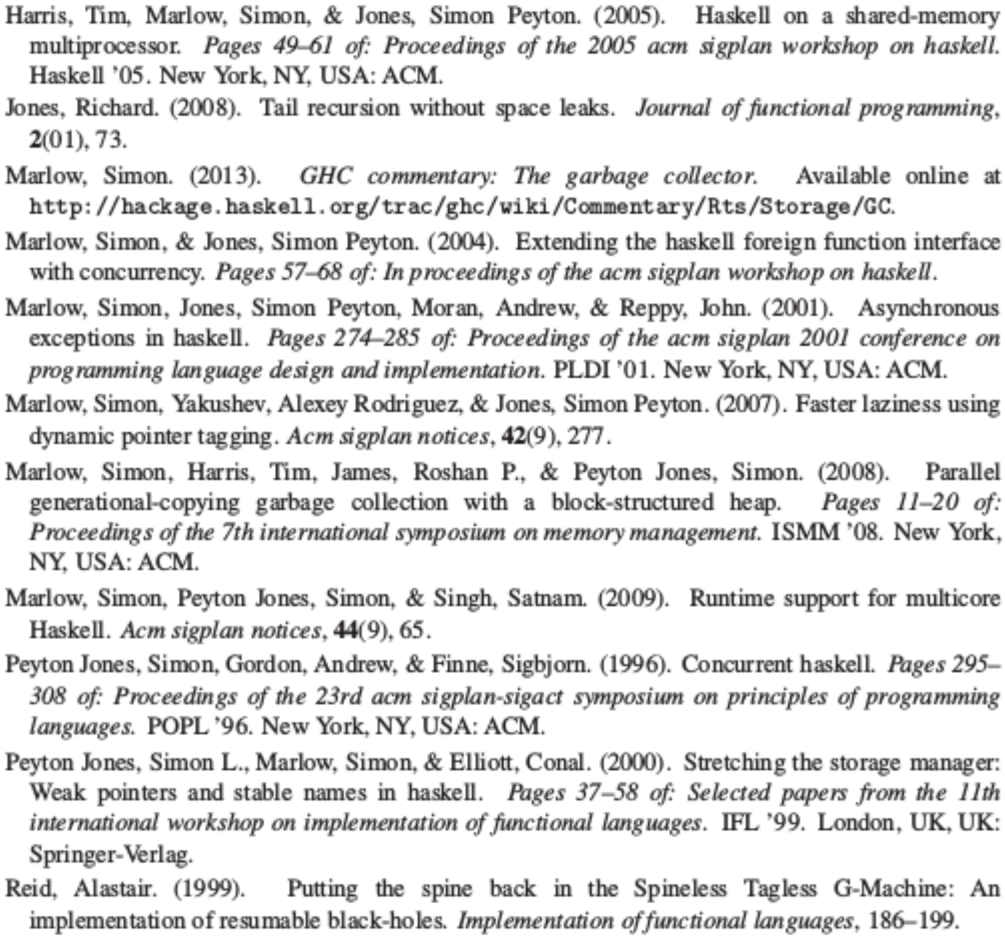
- It includes a sophisticated storage manager, including a multi-gene
with copying and compacting strategies.

- It includes a user-space scheduler for Haskell threads, together wit
Haskell threads across multiple CPUs, and allowing Haskell threads
separate OS threads.

- There's a byte-code interpreter for GHCi, and a dynamic linker for
a GHCi session.

- Heap-profiling (of various kinds), time-profiling and code coverage
included.

Related Work



Harris, Tim, Marlow, Simon, & Jones, Simon Peyton. (2005). Haskell on a shared-memory multiprocessor. *Pages 49–61 of: Proceedings of the 2005 acm sigplan workshop on haskell*. Haskell '05. New York, NY, USA: ACM.

Jones, Richard. (2008). Tail recursion without space leaks. *Journal of functional programming*, **2**(01), 73.

Marlow, Simon. (2013). *GHC commentary: The garbage collector*. Available online at `http://hackage.haskell.org/trac/ghc/wiki/Commentary/Rts/Storage/GC`.

Marlow, Simon, & Jones, Simon Peyton. (2004). Extending the haskell foreign function interface with concurrency. *Pages 57–68 of: In proceedings of the acm sigplan workshop on haskell*.

Marlow, Simon, Jones, Simon Peyton, Moran, Andrew, & Reppy, John. (2001). Asynchronous exceptions in haskell. *Pages 274–285 of: Proceedings of the acm sigplan 2001 conference on programming language design and implementation*. PLDI '01. New York, NY, USA: ACM.

Marlow, Simon, Yakushev, Alexey Rodriguez, & Jones, Simon Peyton. (2007). Faster laziness using dynamic pointer tagging. *Acm sigplan notices*, **42**(9), 277.

Marlow, Simon, Harris, Tim, James, Roshan P., & Peyton Jones, Simon. (2008). Parallel generational-copying garbage collection with a block-structured heap. *Pages 11–20 of: Proceedings of the 7th international symposium on memory management*. ISMM '08. New York, NY, USA: ACM.

Marlow, Simon, Peyton Jones, Simon, & Singh, Satnam. (2009). Runtime support for multicore Haskell. *Acm sigplan notices*, **44**(9), 65.

Peyton Jones, Simon, Gordon, Andrew, & Finne, Sigbjorn. (1996). Concurrent haskell. *Pages 295–308 of: Proceedings of the 23rd acm sigplan-sigact symposium on principles of programming languages*. POPL '96. New York, NY, USA: ACM.

Peyton Jones, Simon L., Marlow, Simon, & Elliott, Conal. (2000). Stretching the storage manager: Weak pointers and stable names in haskell. *Pages 37–58 of: Selected papers from the 11th international workshop on implementation of functional languages*. IFL '99. London, UK, UK: Springer-Verlag.

Reid, Alastair. (1999). Putting the spine back in the Spineless Tagless G-Machine: An implementation of resumable black-holes. *Implementation of functional languages*, 186–199.

http://ezyang.com/jfp-ghc-rts-draft.pdf

WIP