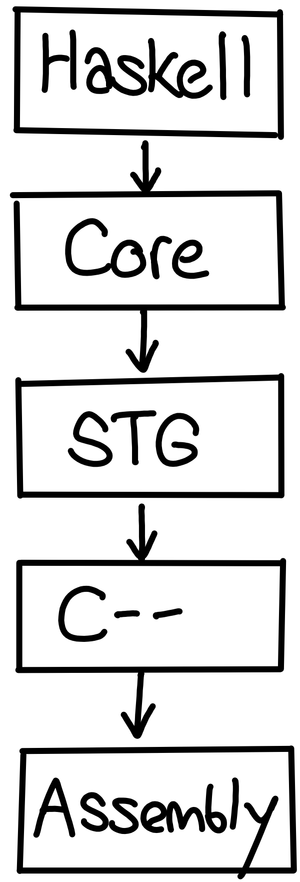
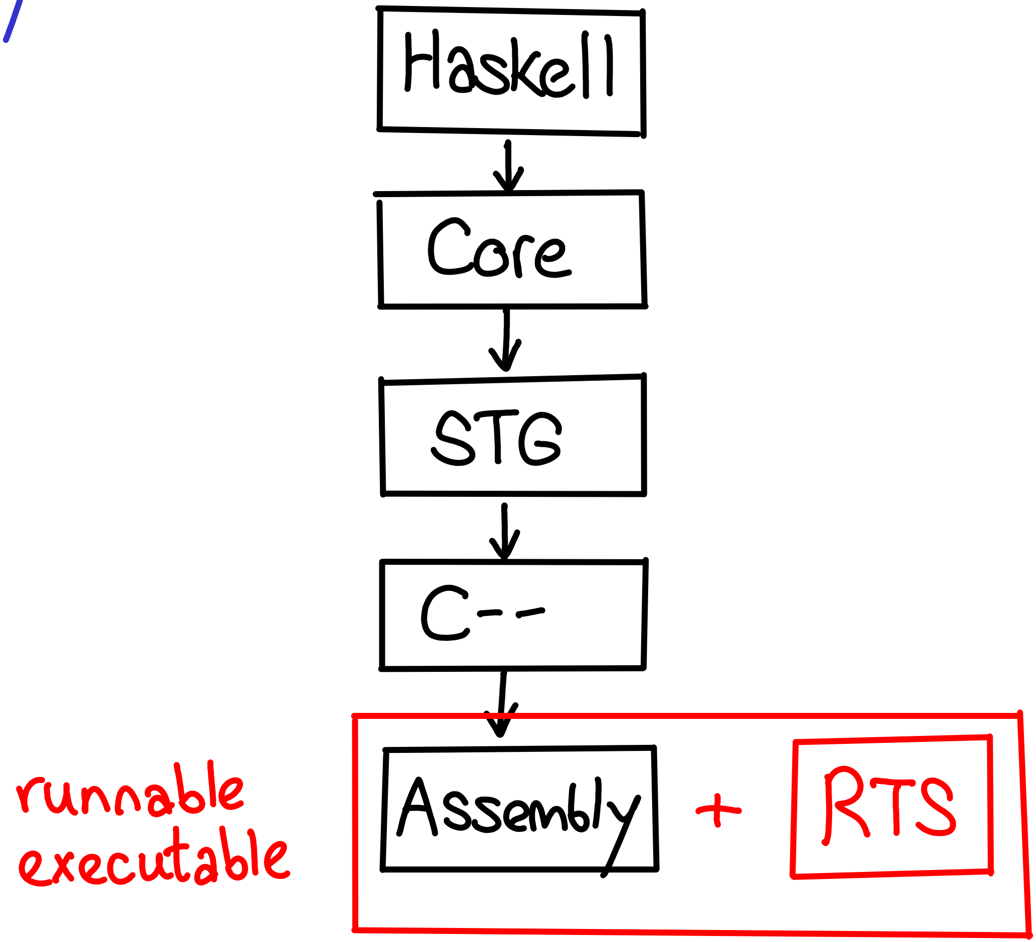
The GHC Runtime System

Edward Z. Yang

Last time



Today



Why learn about the RTS?

Code becomes slower as more boxed arrays are allocated

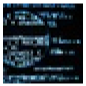
22

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)
```

Quora Search



Computer Programming: Edit

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

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

Specifically quoting the introduction:

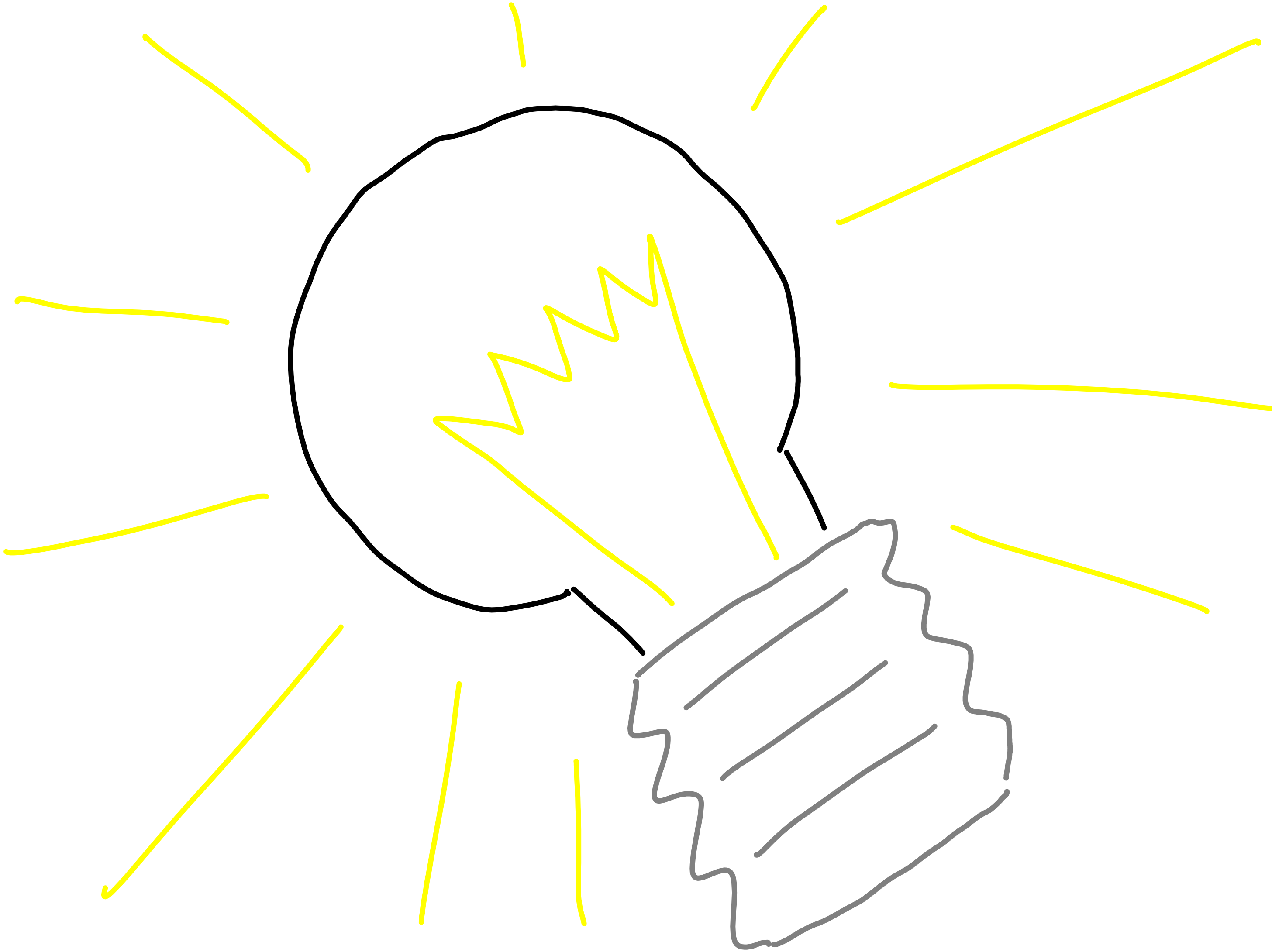
> A naive implementation, using one
> native thread (i.e. OS thread) per request would lead to the use of a large number
> of native threads, which would substantially degrade performance due to the relatively high cost of OS context switches [22]. In contrast, Haskell threads are lightweight
> threads,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

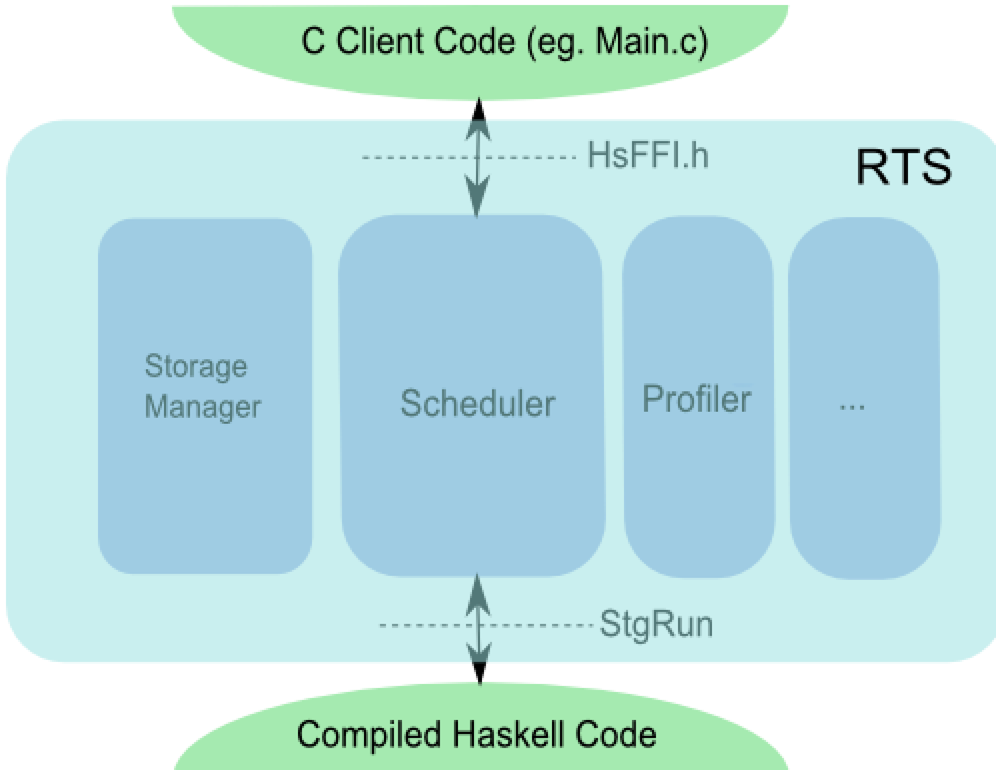
I've heard the anecdote that Ruby threading was so slow because Ruby used "green threads" instead of native threads e.g. like Java. So what makes Haskell "green threads" different from Ruby "green threads?"

CLR JVM

V8 GHC Golang

SpiderMonkey [language]





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

In a nutshell...

- → Storage Manager (Garbage Collection)
- → Scheduler
- → Bytecode Interpreter (GHCi)
- → Dynamic Linker
- → Software Transactional Memory
- → Profiling

[and more...]

In a nutshell...

- → $\boxed{\text{Storage Manager (Garbage Collection)} \\ \text{Scheduler}}$
- → Bytecode Interpreter (GHCi)
- → Dynamic Linker
- → Software Transactional Memory
- → Profiling

[and more...]

→ Storage Manager

Generational Copying GC

Write barriers & promotion

Parallel GC (briefly)

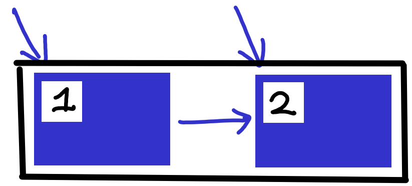
→ Scheduler

Threads	HECs
Load balancing	Bound threads
MVars	

Garbage Collection

Garbage Collection: Brief Review

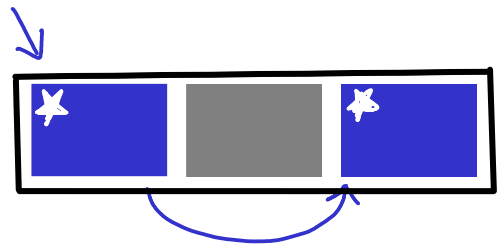
Reference Counting



- 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

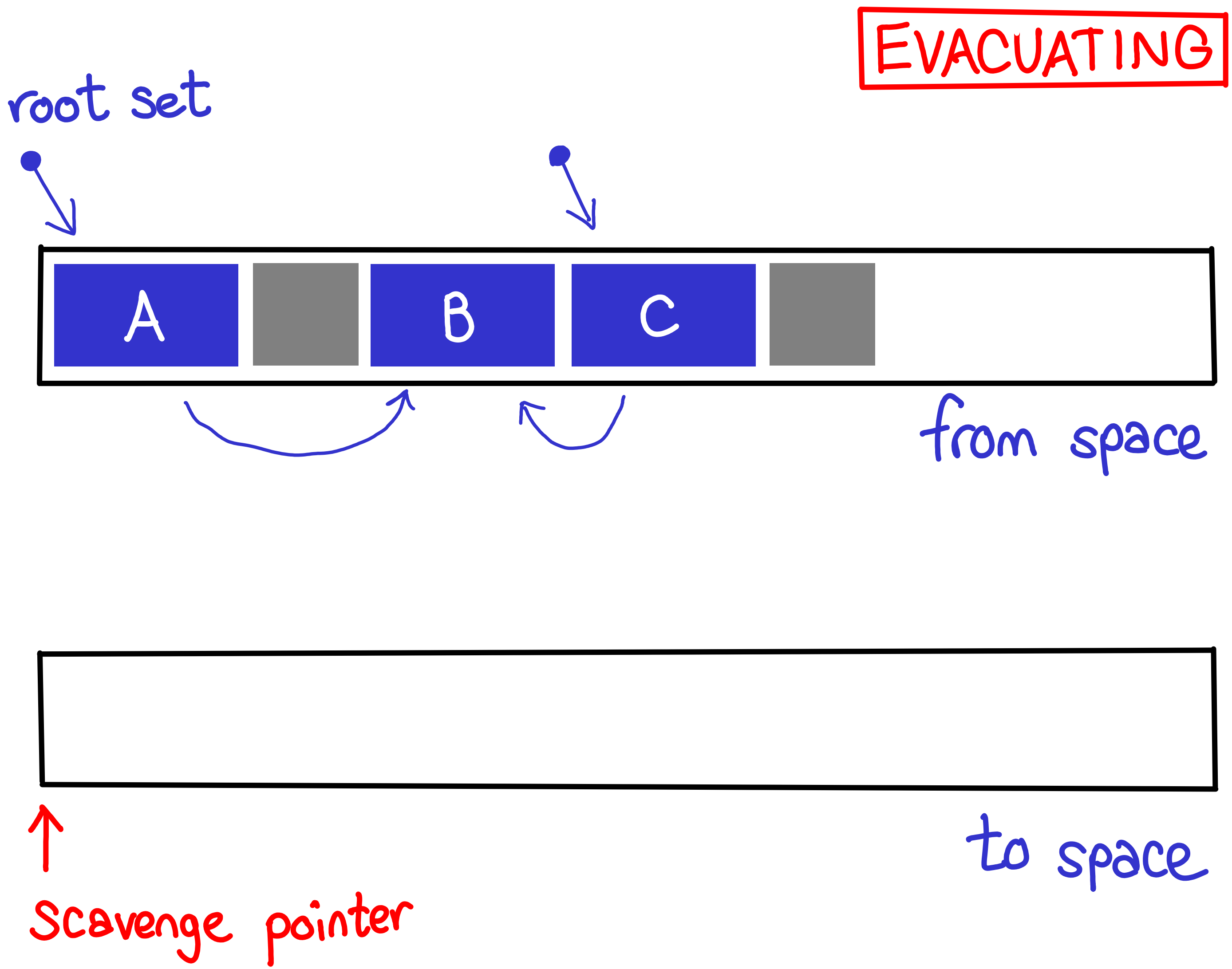
Generational Copying Collector

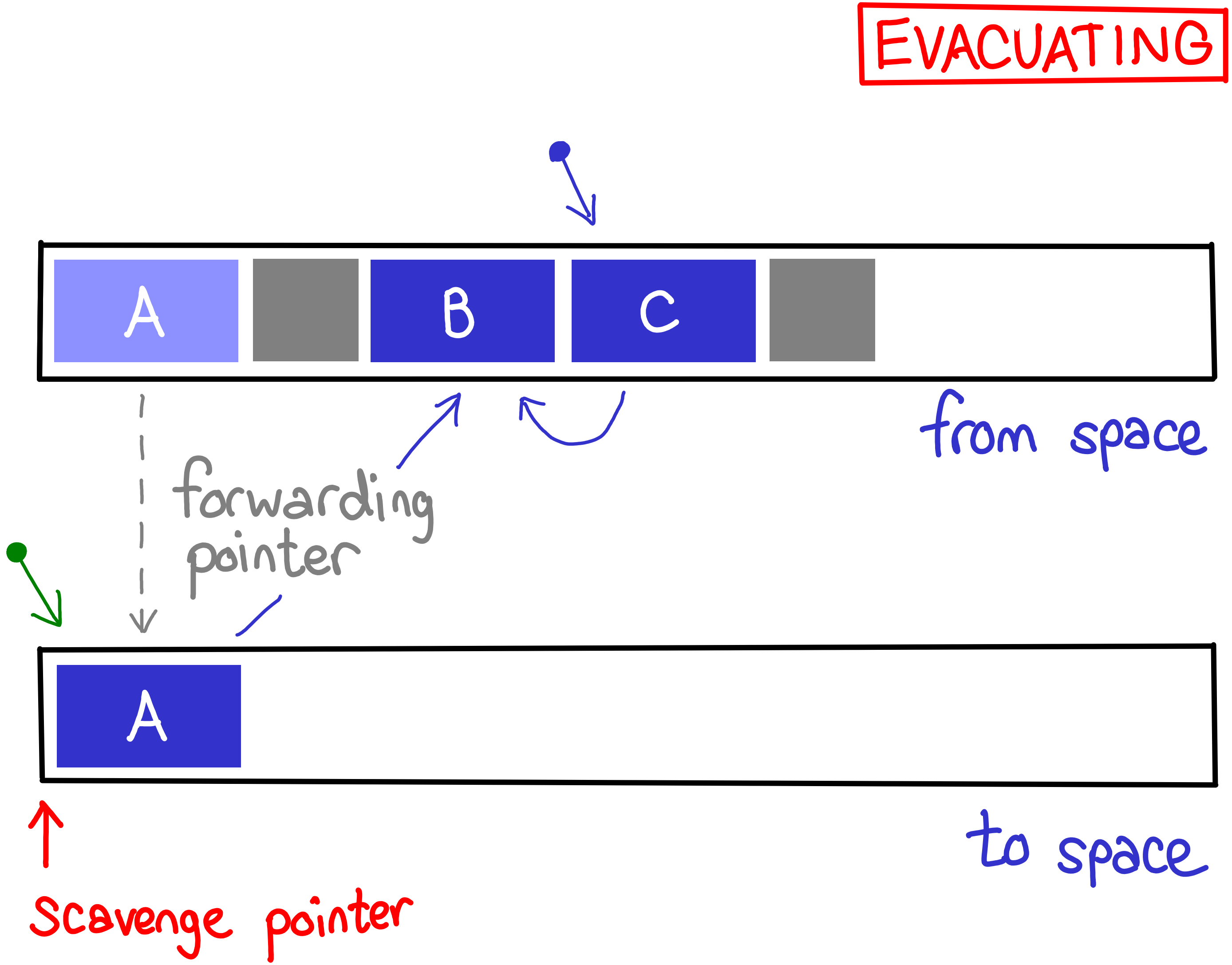
JVM, V8, GHC

"Most objects die young"

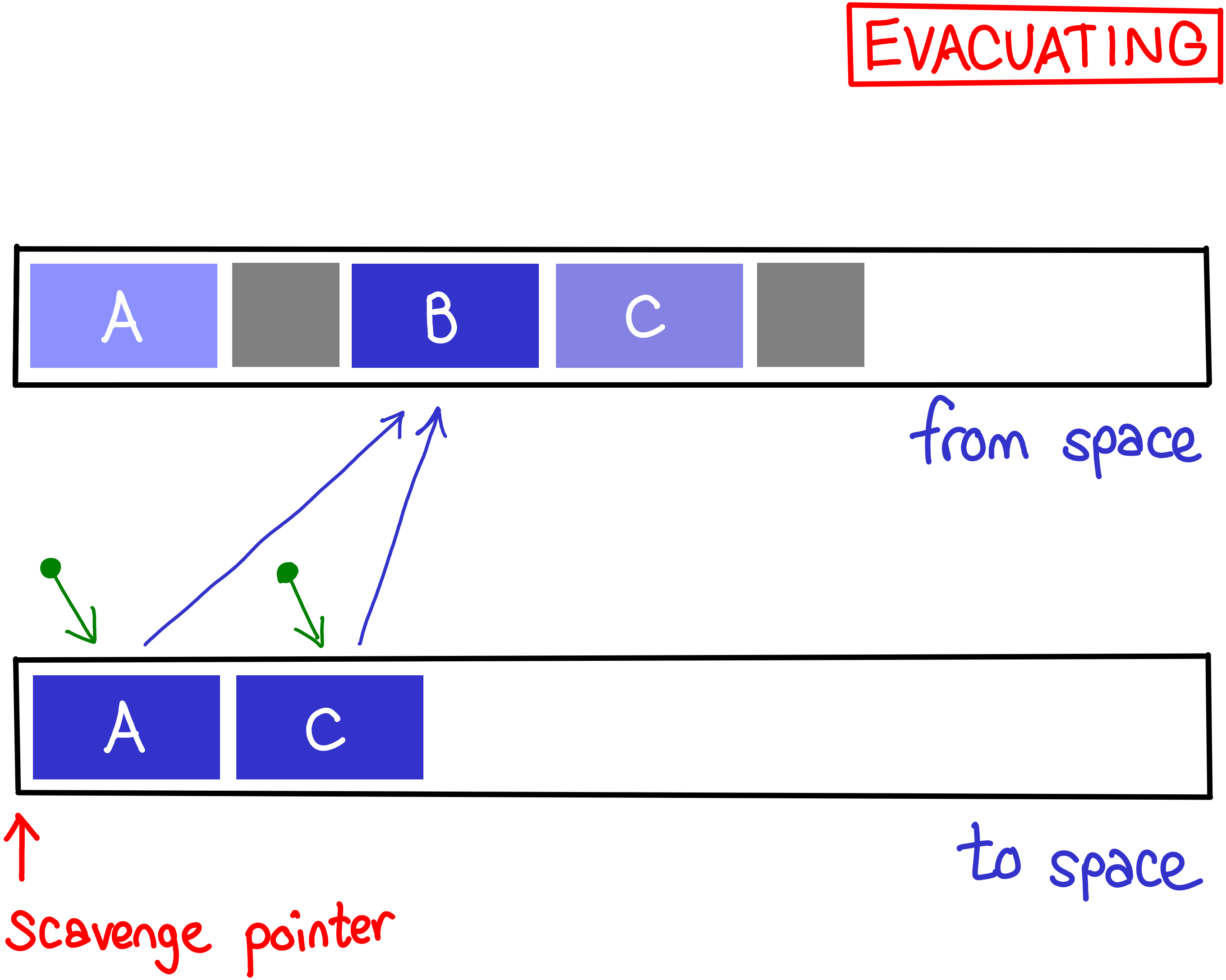
especially in functional languages!

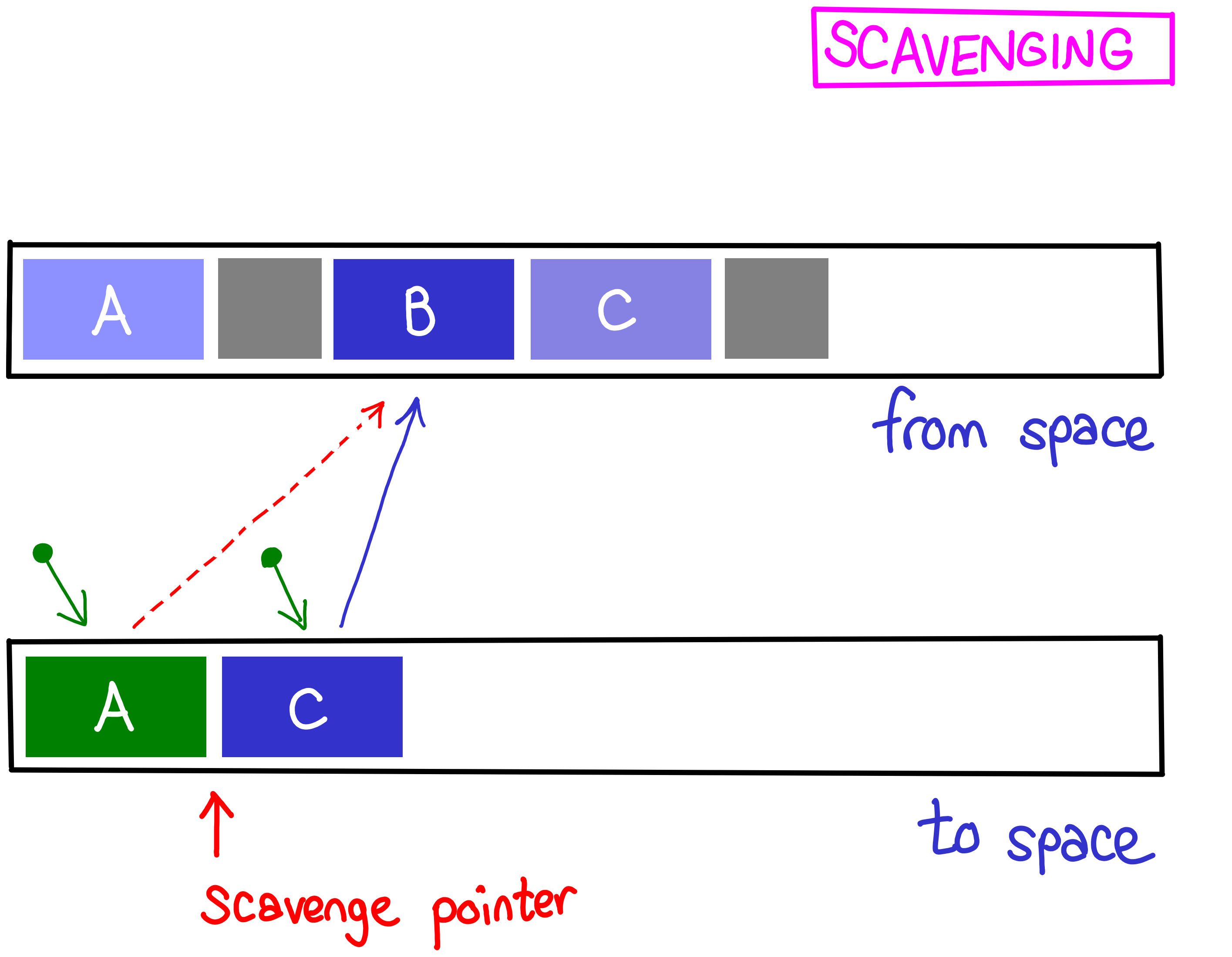
—the Generational Hypothesis

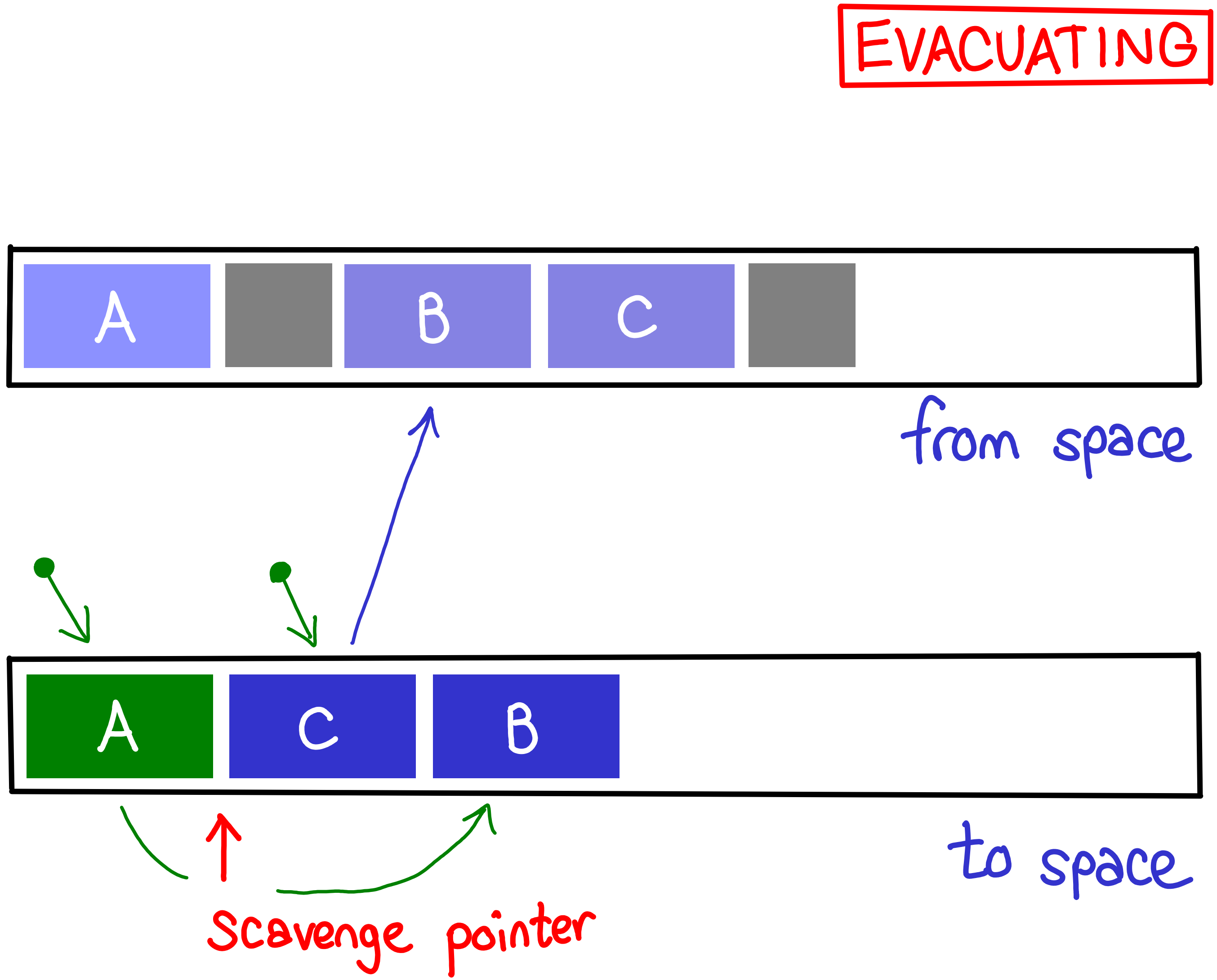

EVACUATING
root set
A
B
C
from space
to space
scavenge pointer

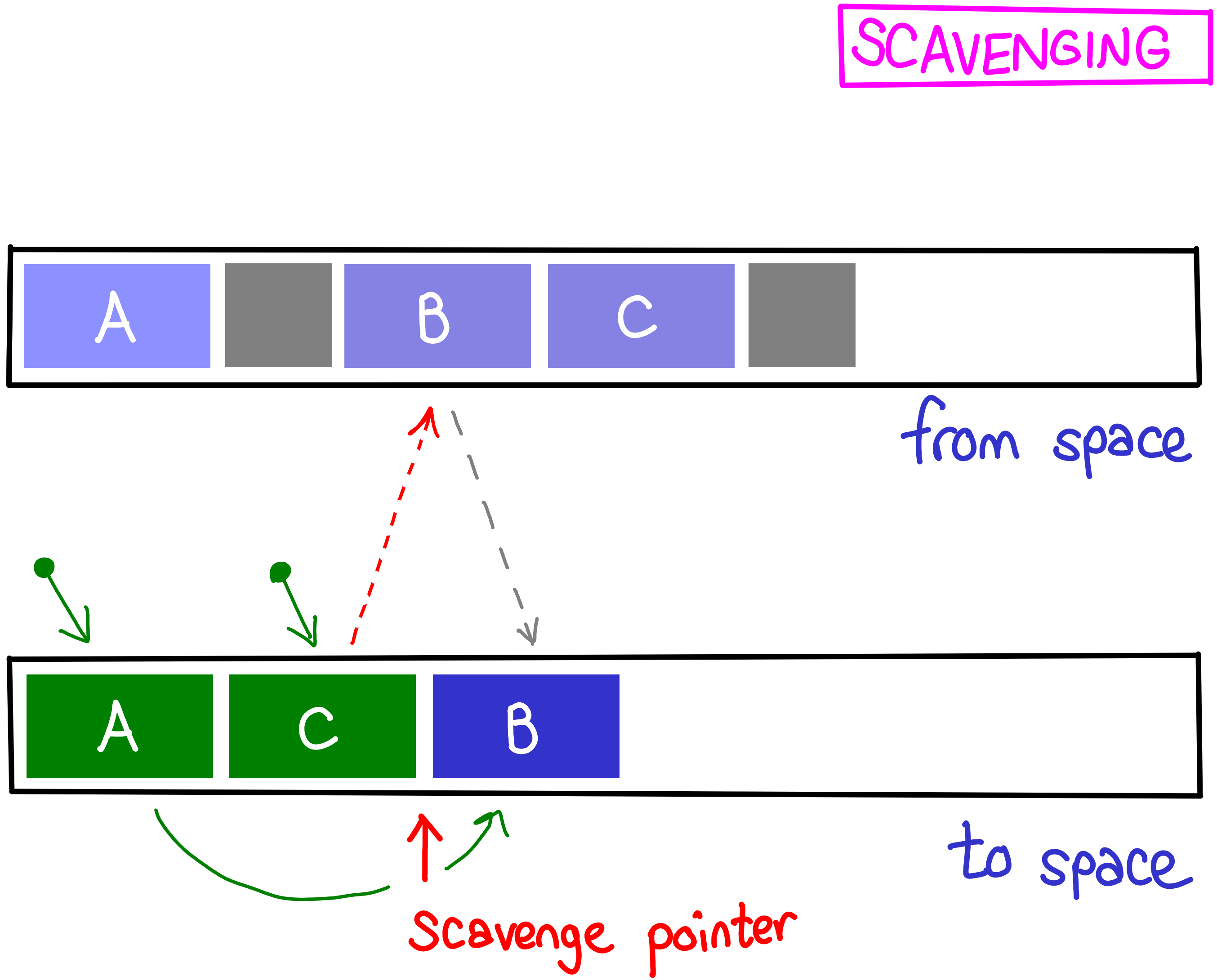


EVACUATING
A
B
C
from space
forwarding pointer
A
to space
scavenge pointer

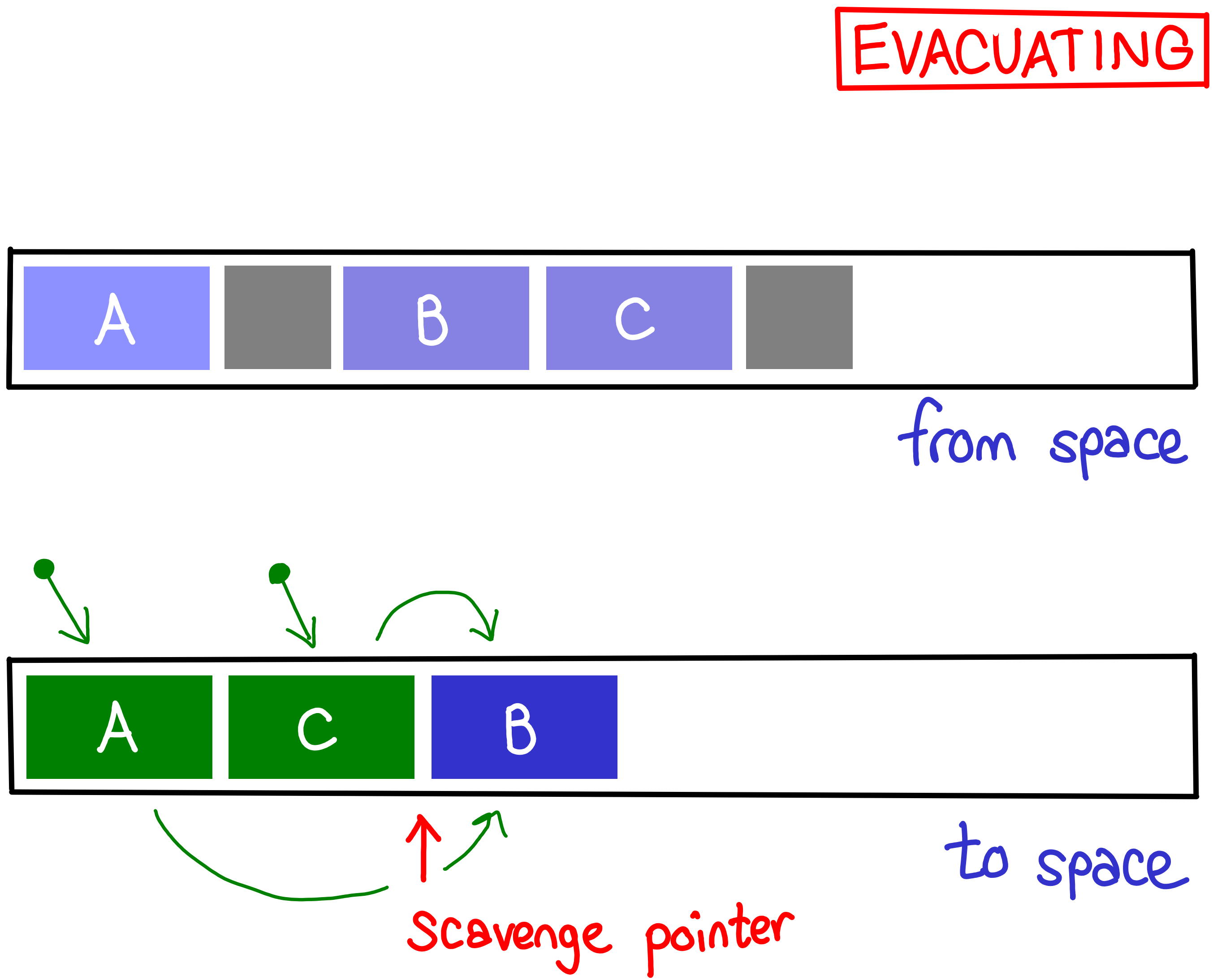

EVACUATING
A
B
C
from space
A
C
to space
scavenge pointer

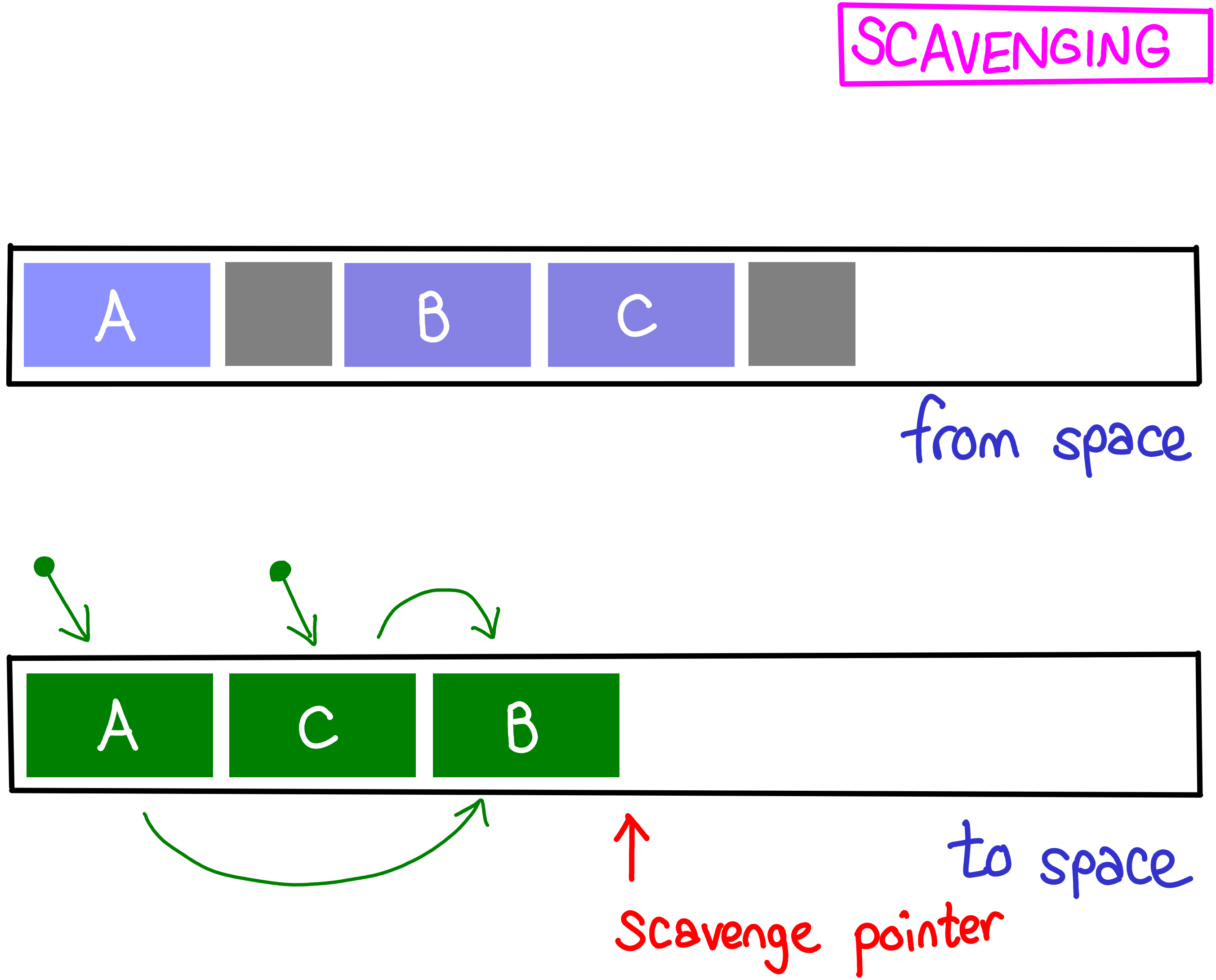

SCAVENGING
A
B
C
from space
A
C
to space
scavenge pointer

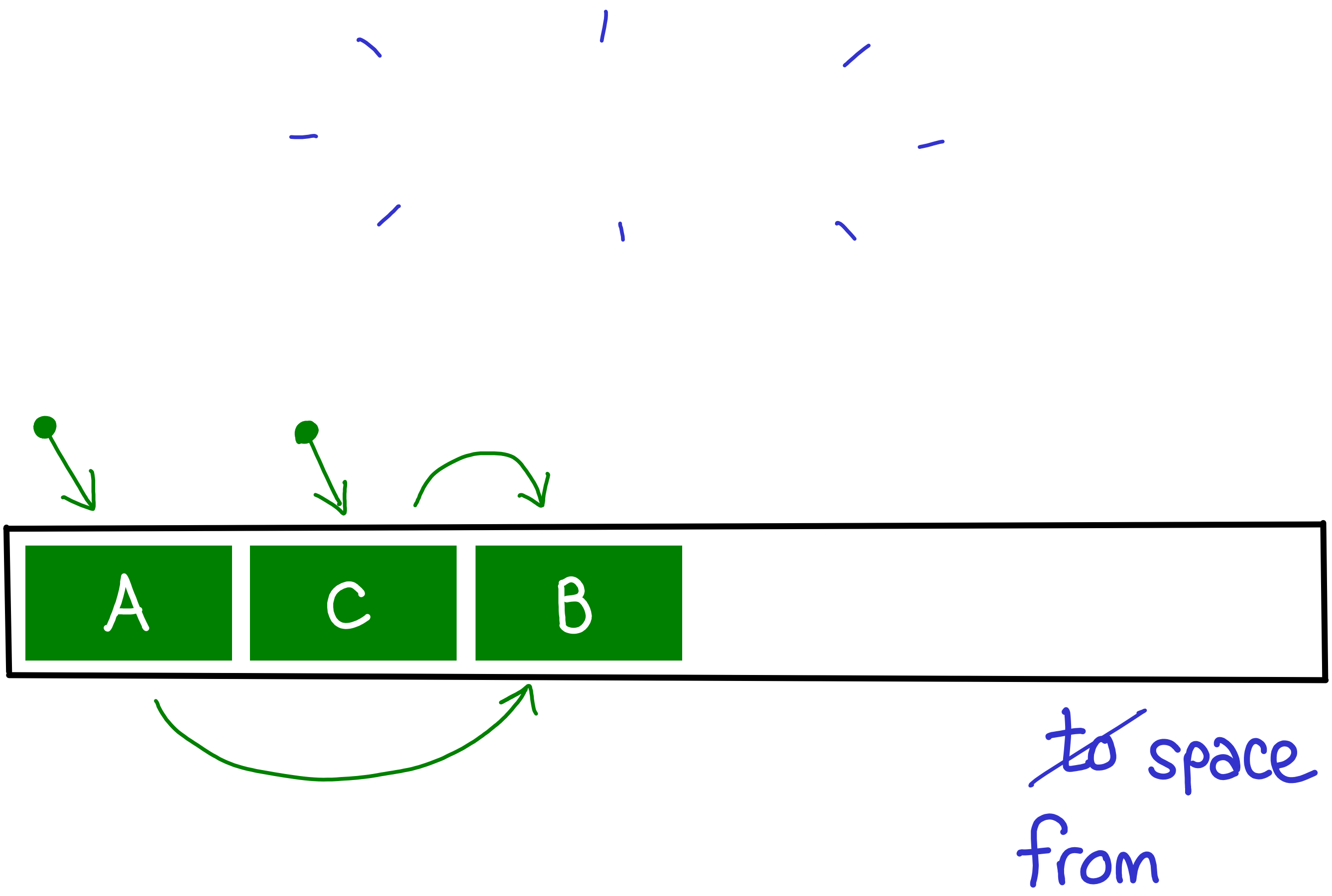

EVACUATING
A
B
C
from space
A
C
B
to space
scavenge pointer

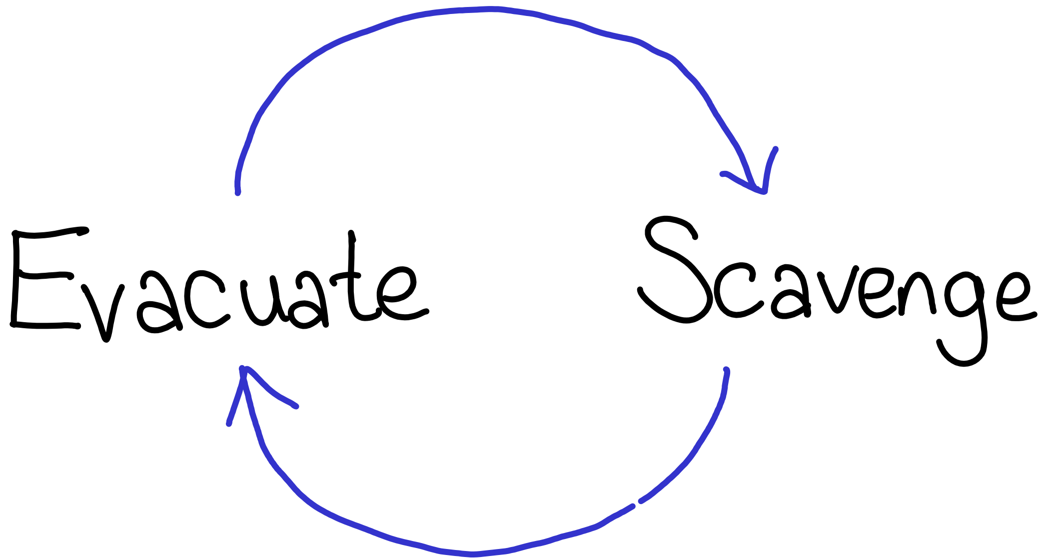


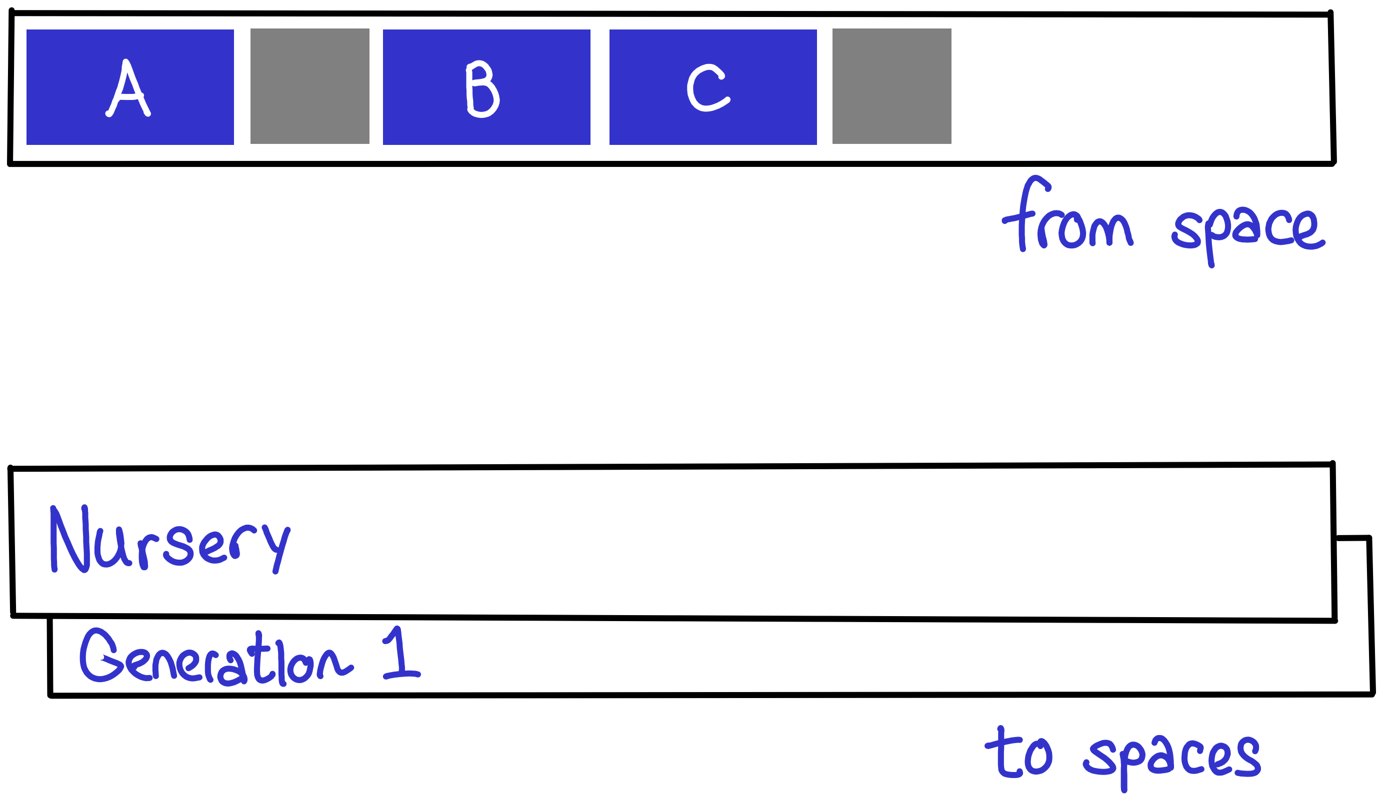
SCAVENGING
A
B
C
from space
A
C
B
to space
Scavenge pointer


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

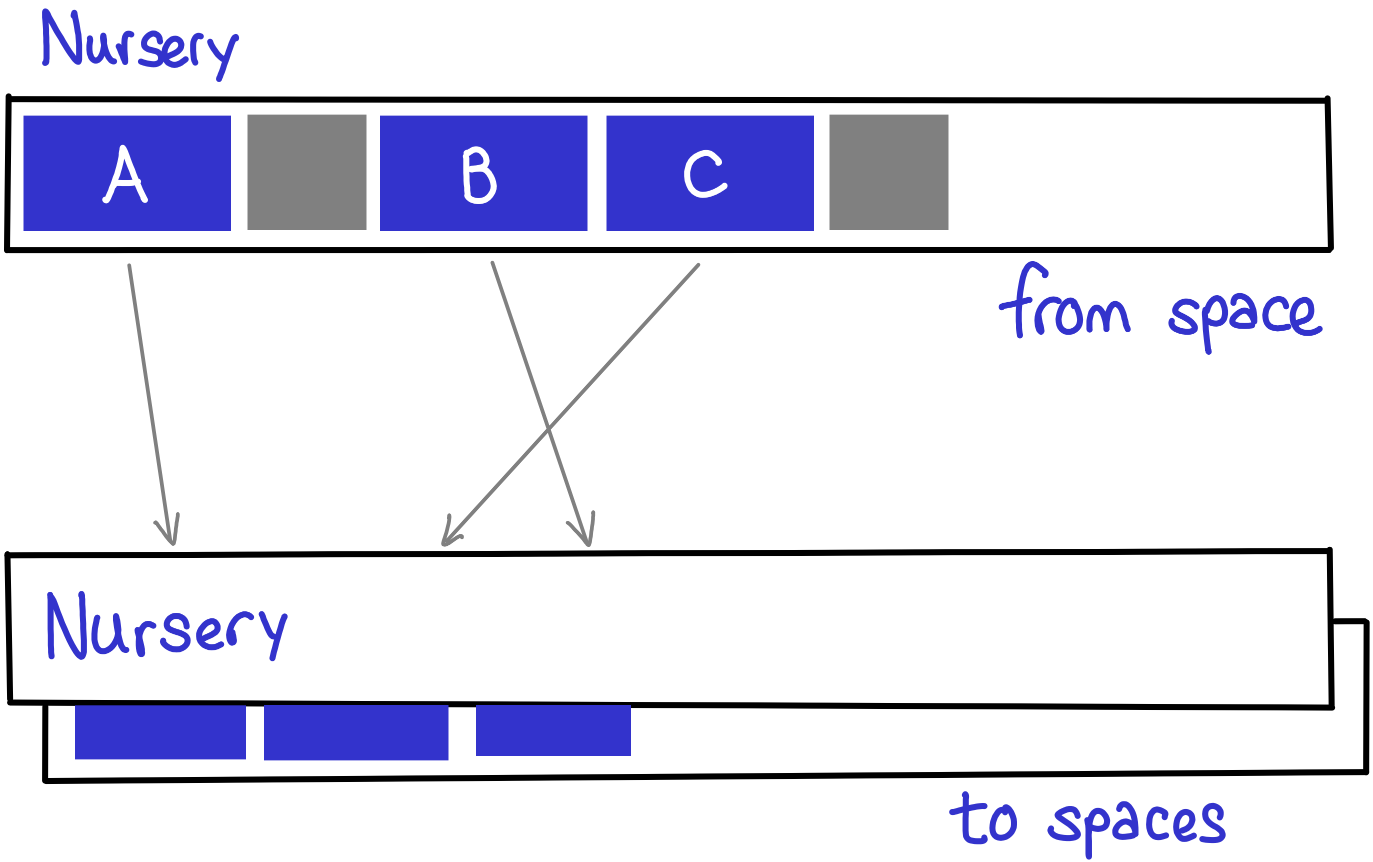

SCAVENGING
A
B
C
from space
A
C
B
to space
scavenge pointer

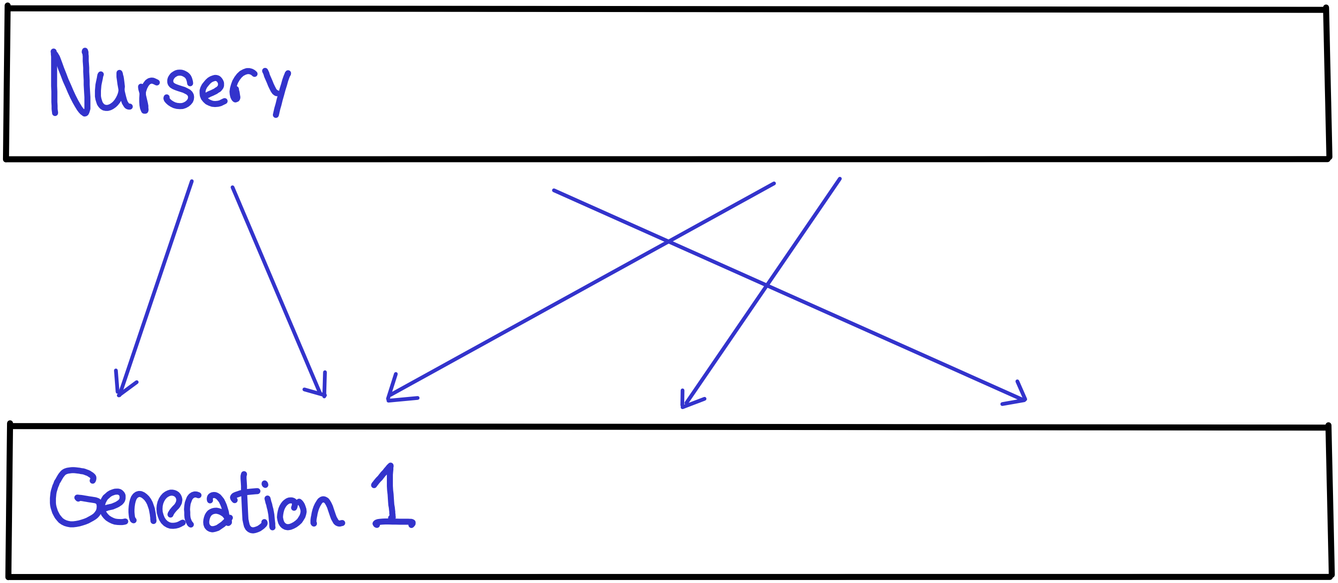

A
C
B
~~to~~ space
from

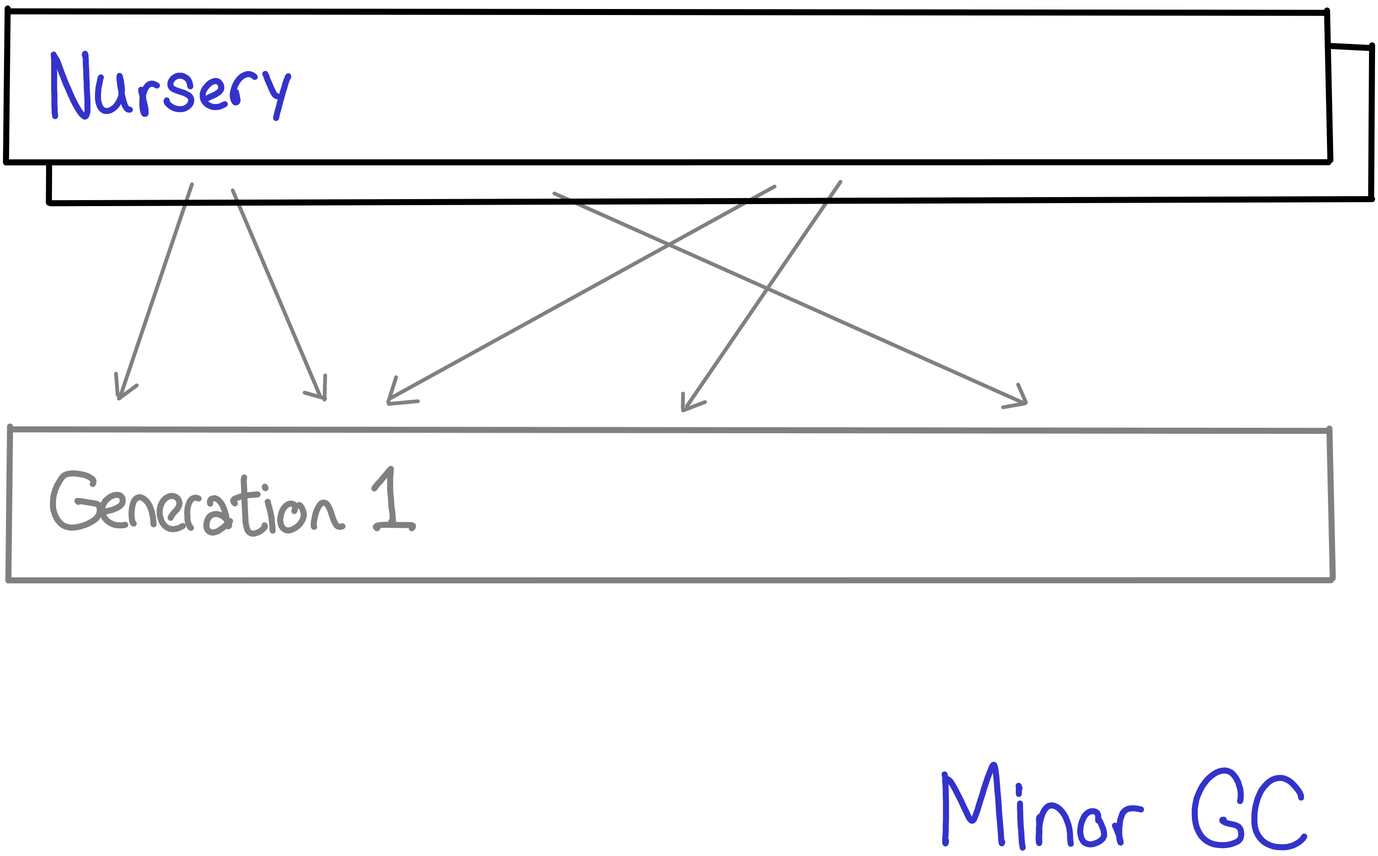

Evacuate
Scavenge


A
B
C
from space
Nursery
Generation 1
to spaces

Tenuring




Nursery
Generation 1


Nursery
Generation 1
Minor GC

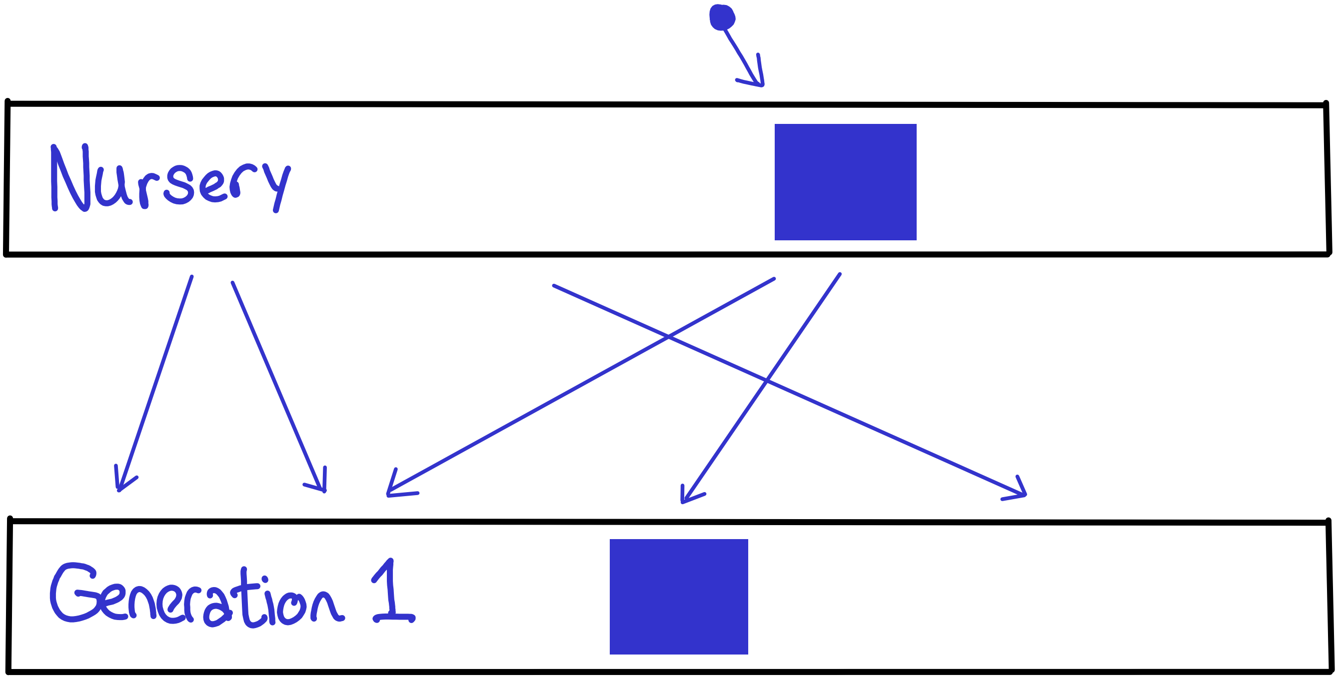
Generational Copying Collector

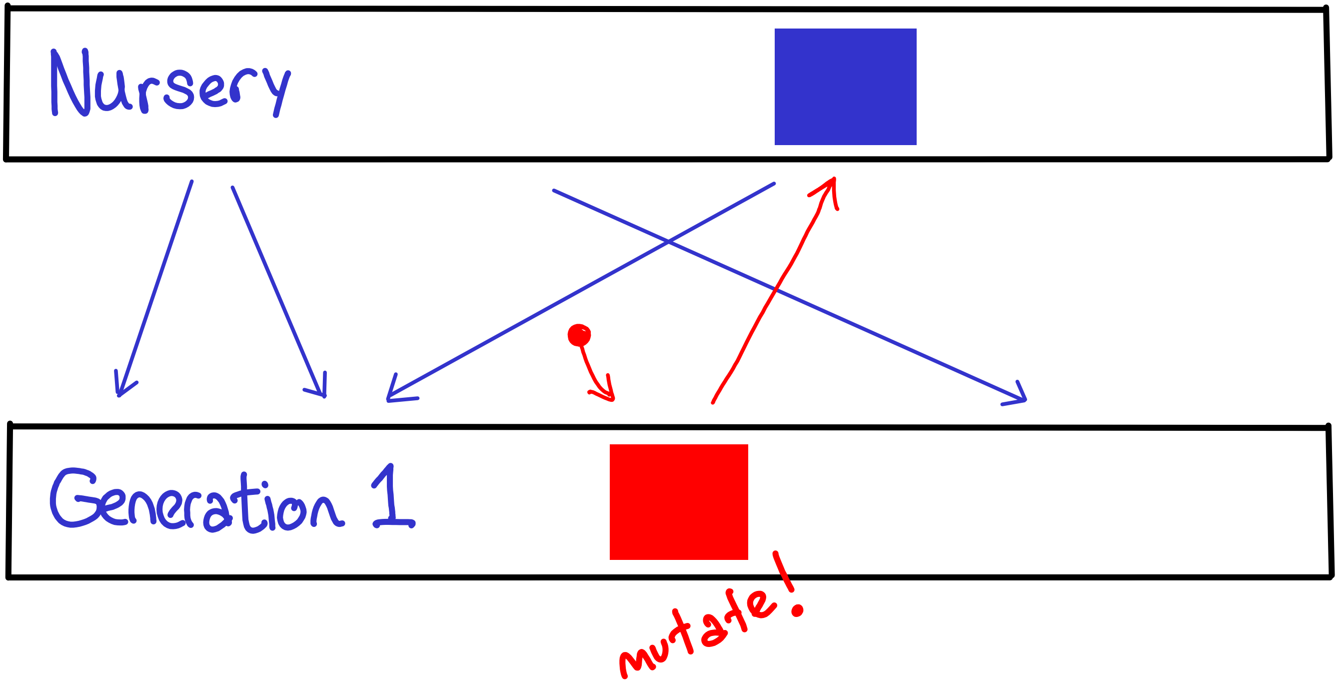
- The more garbage you have, the faster it runs
- Free memory is contiguous

```
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 ();
}
```

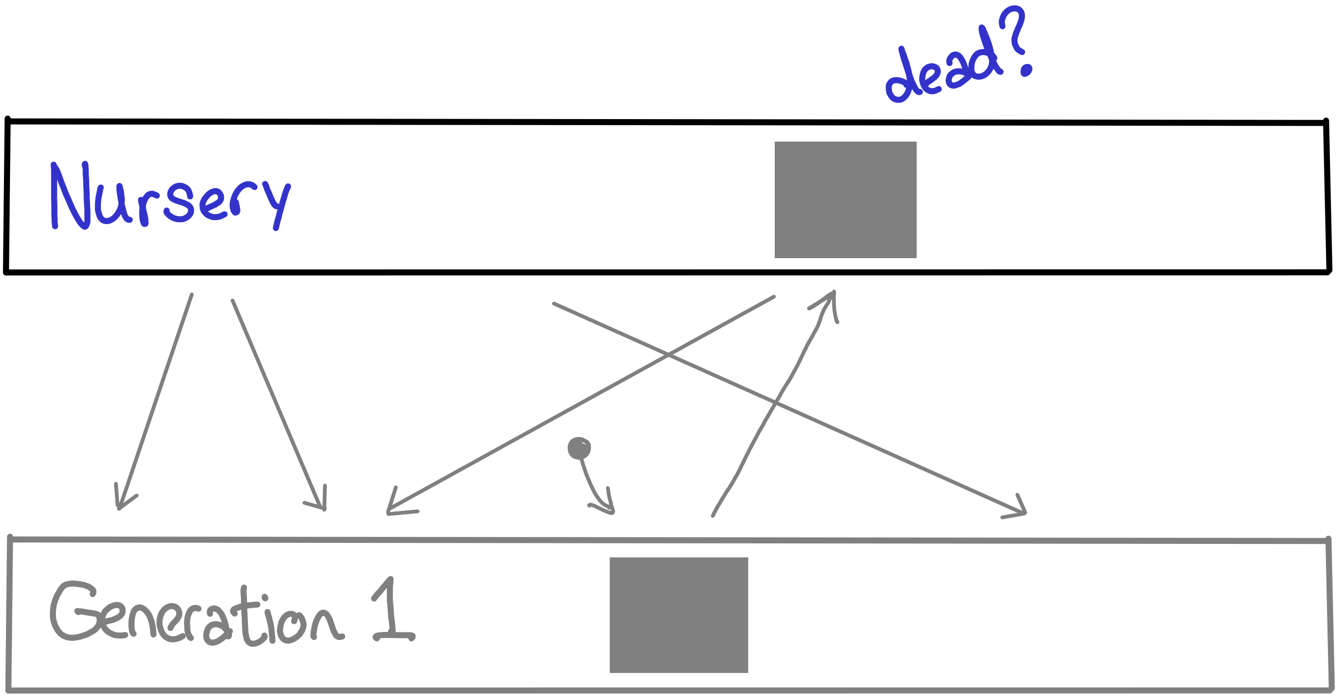
What about Purity?

- → Write Barriers
- → Parallel Garbage Collection

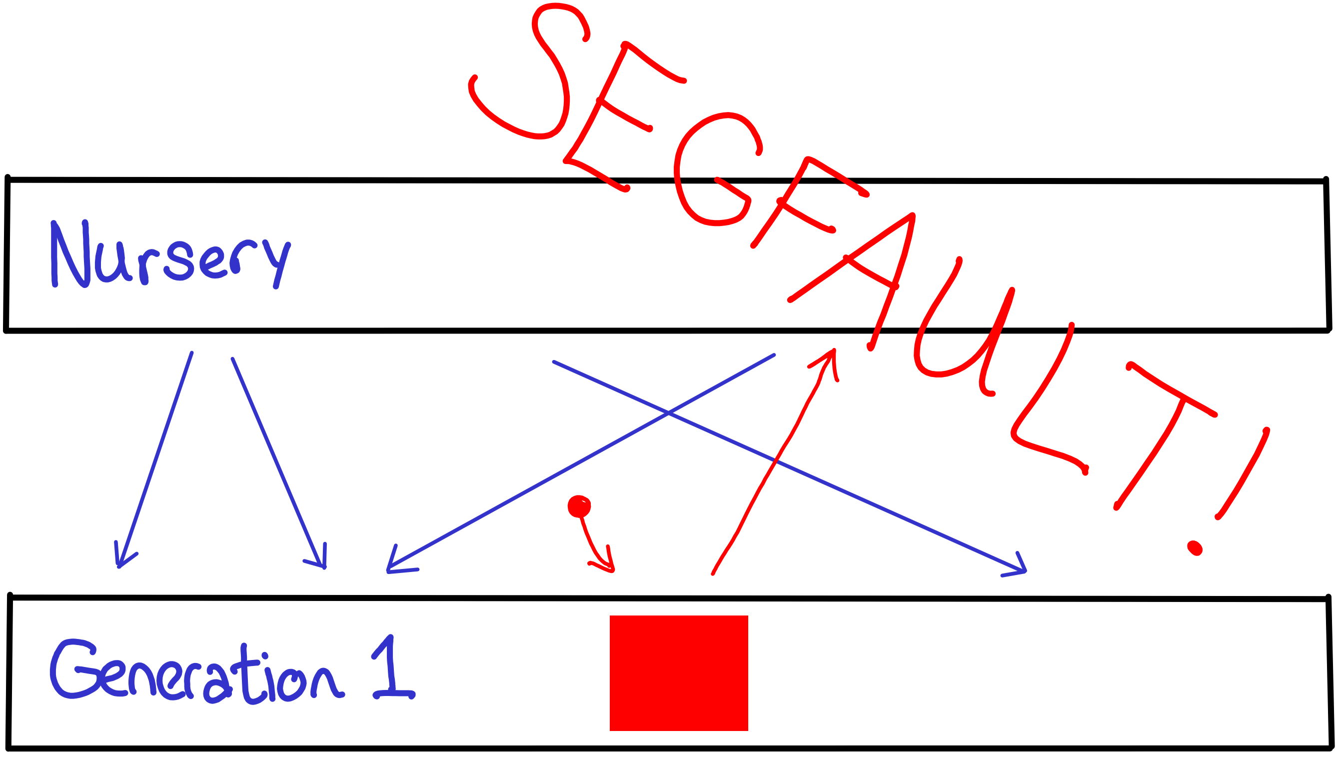

Nursery
Generation 1

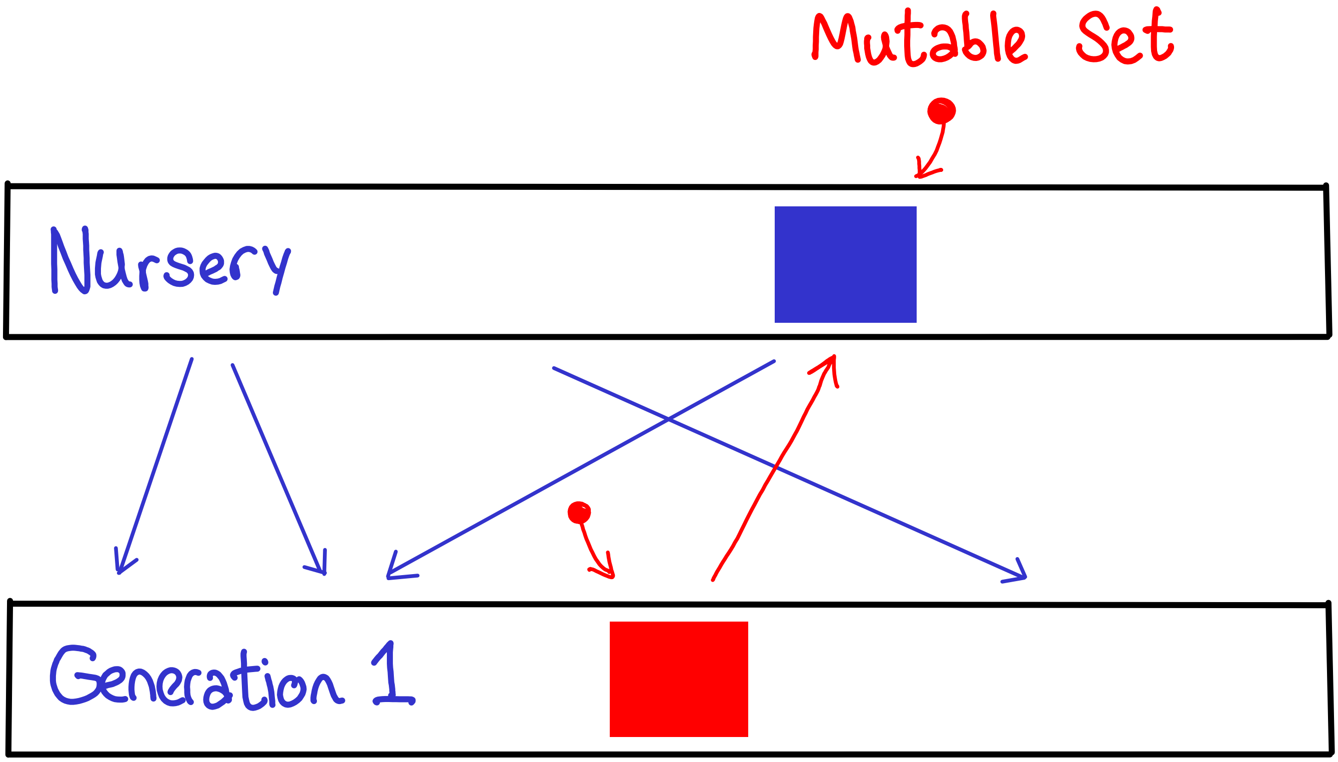


Nursery
Generation 1
mutate!



Minor GC


SEGFAULT!
Nursery
Generation 1



Mutable Set
Nursery
Generation 1

Why is generational GC hard?

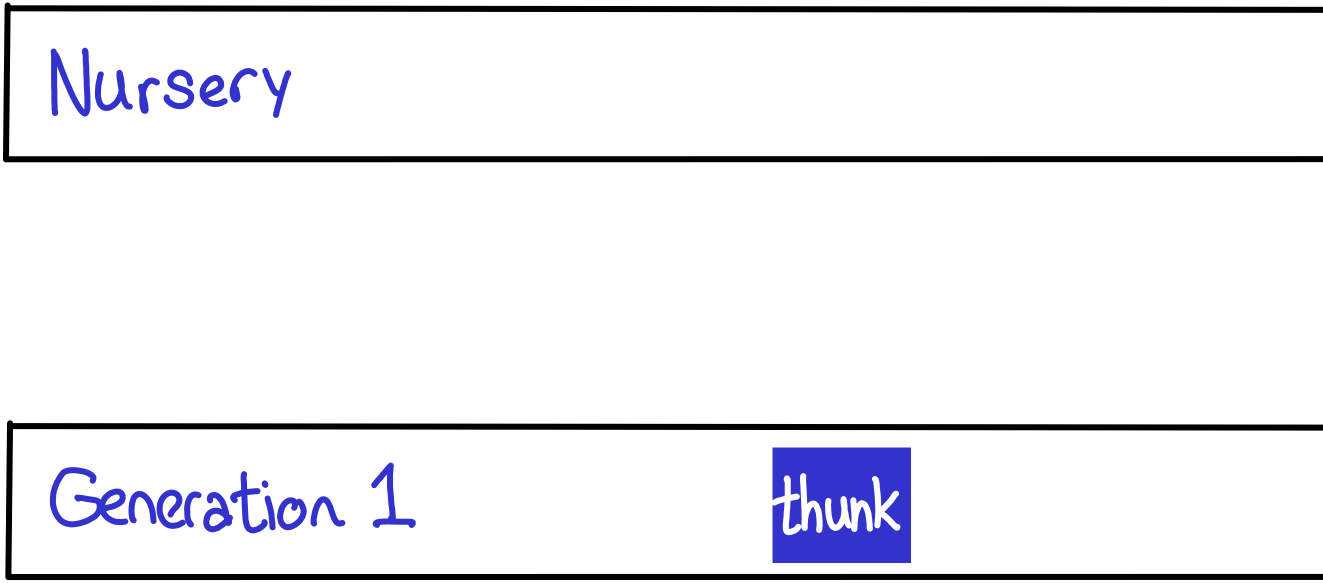
<u>This</u>.

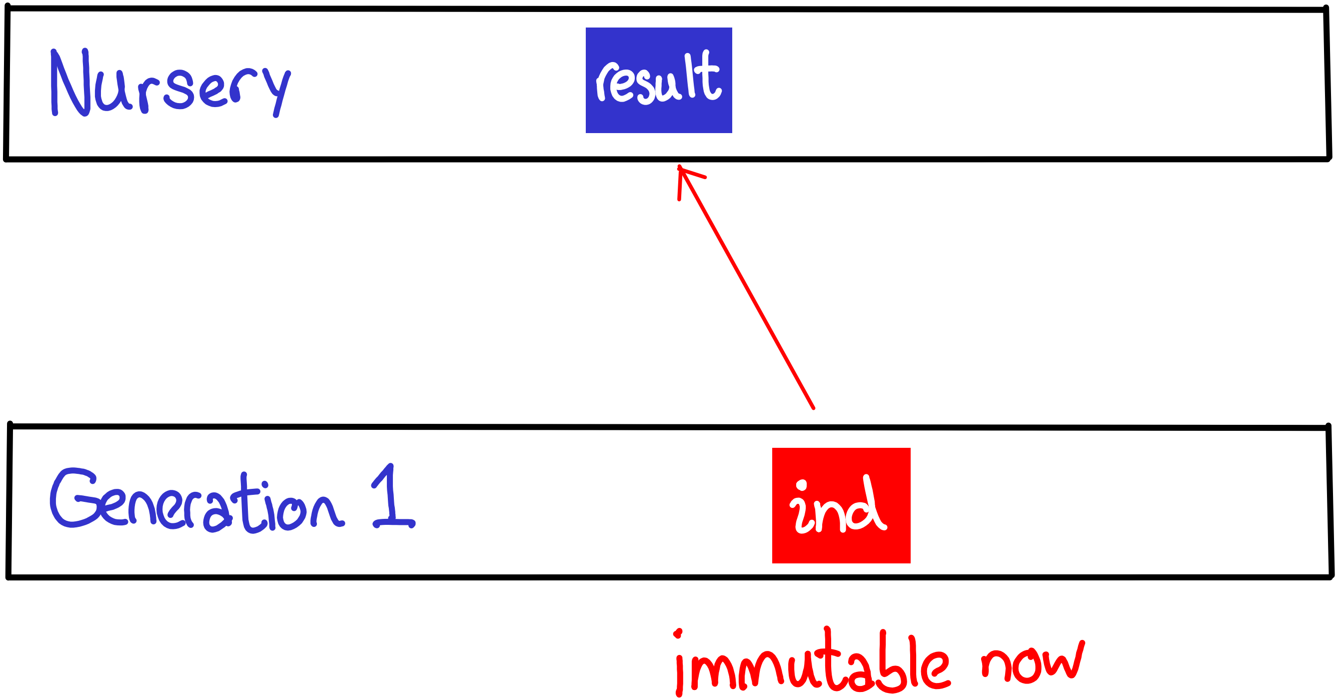
Why is generational GC hard in Java?

This.

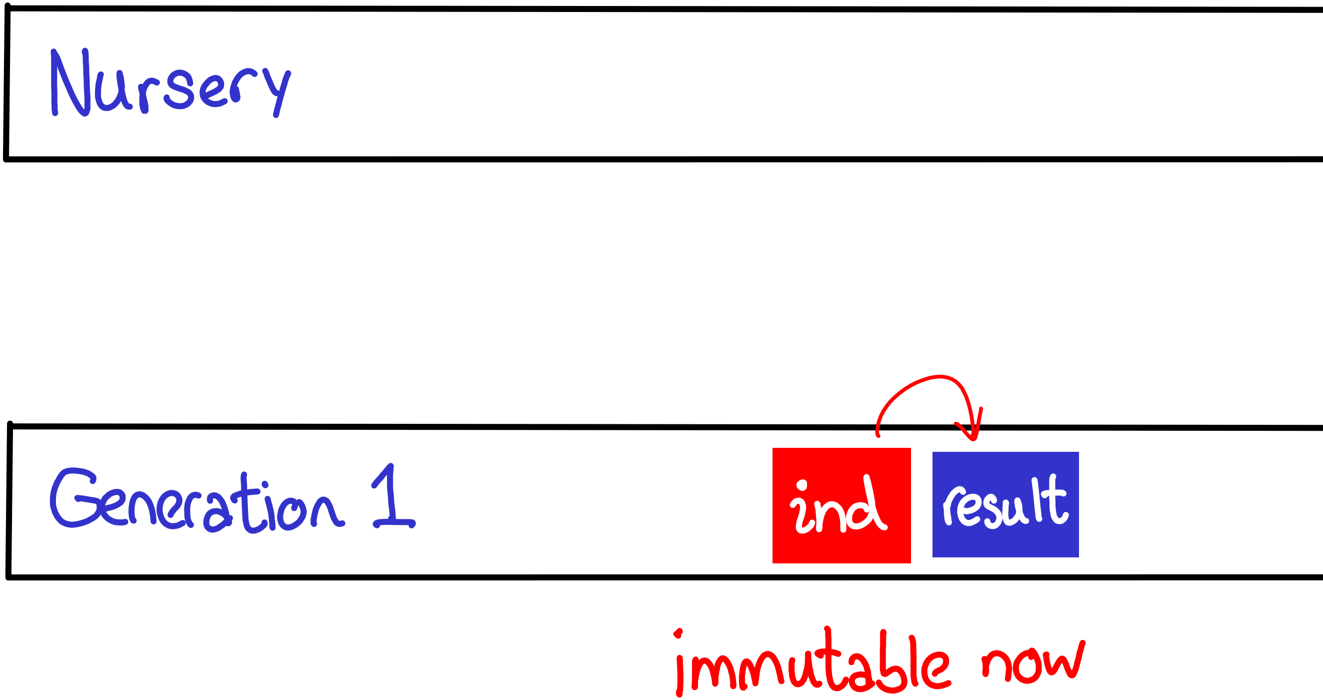
Purity to the rescue

- Mutation is rare
- IORefs are slow anyway
- Laziness is a special kind of mutation


Nursery
Generation 1
thunk

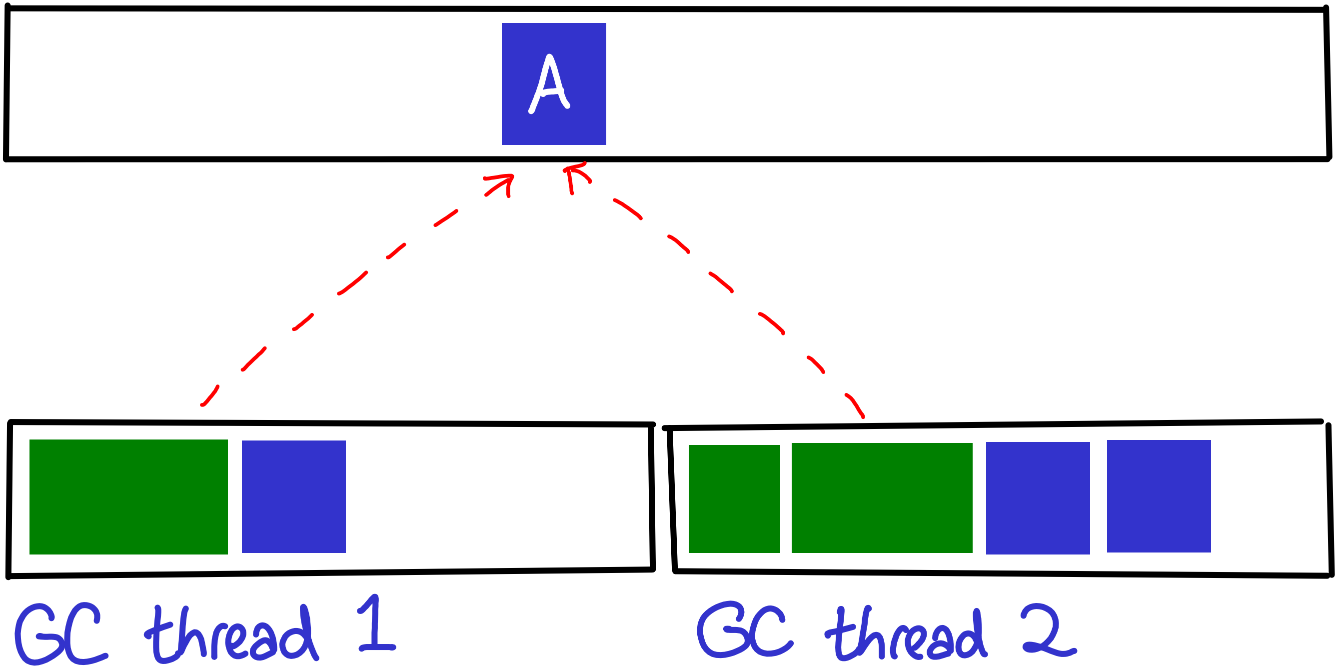

Nursery
result
Generation 1
ind
immutable now

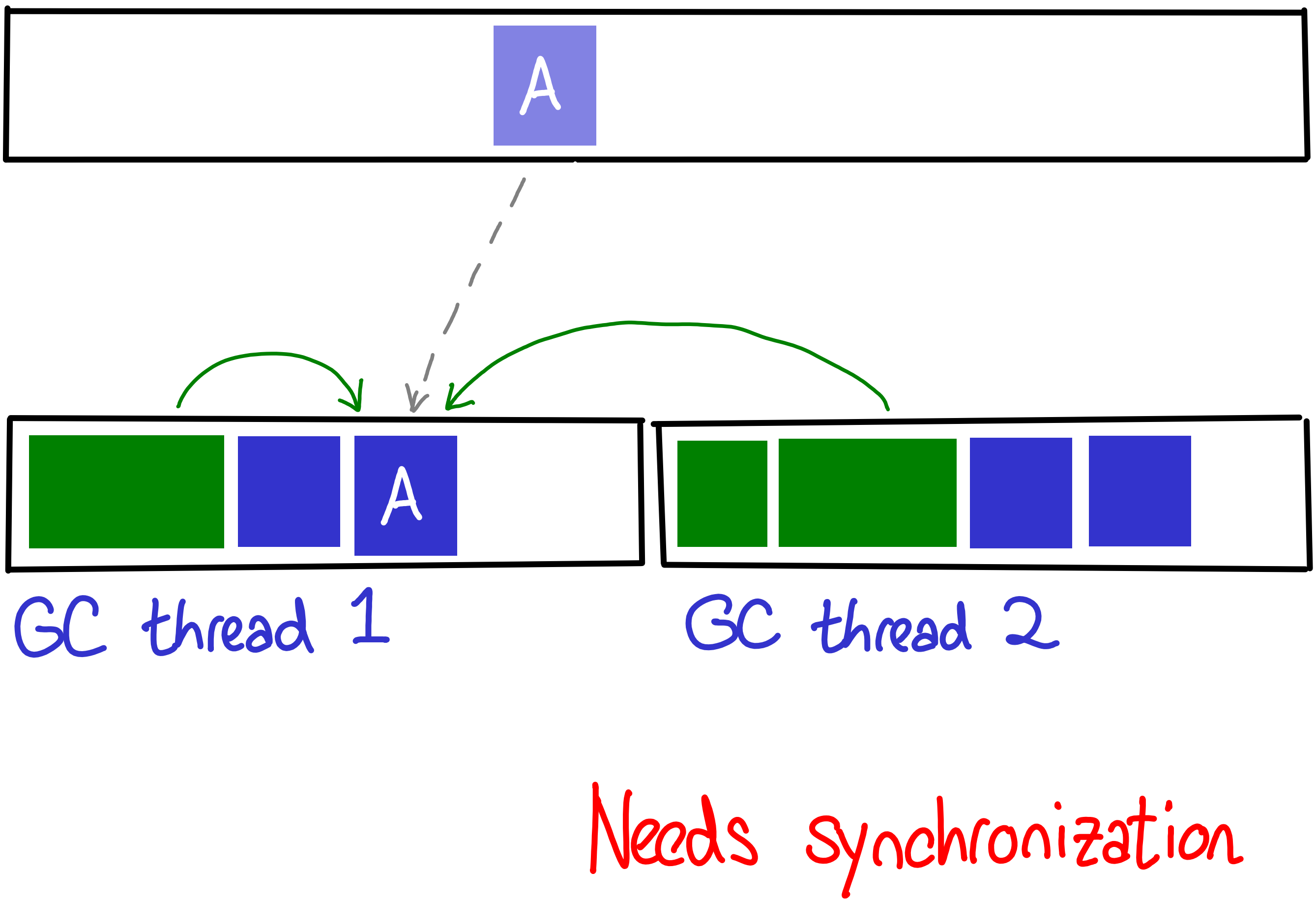
Promotion



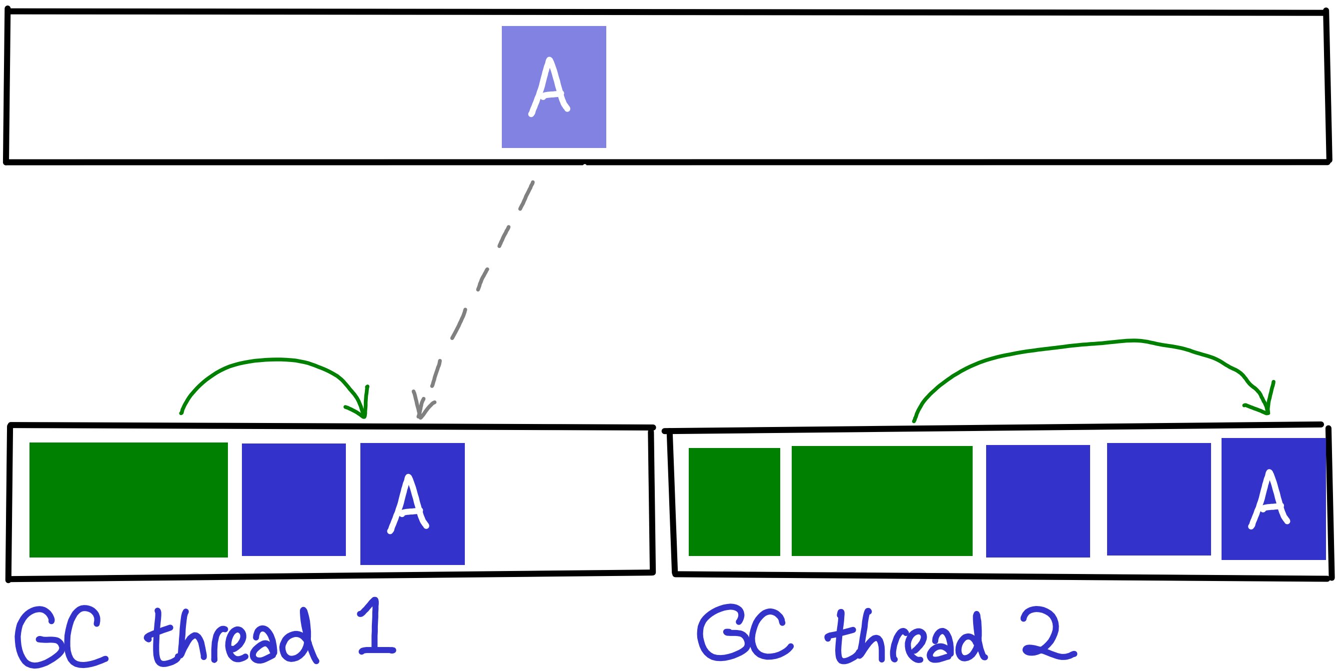
Parallel GC

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


A
GC thread 1
GC thread 2


A
A
GC thread 1
GC thread 2
Needs synchronization

If A is immutable...



...observationally indistinguishable!

Purity = Flexibility

Code becomes slower as more boxed arrays are allocated

22

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)
```

Scheduler

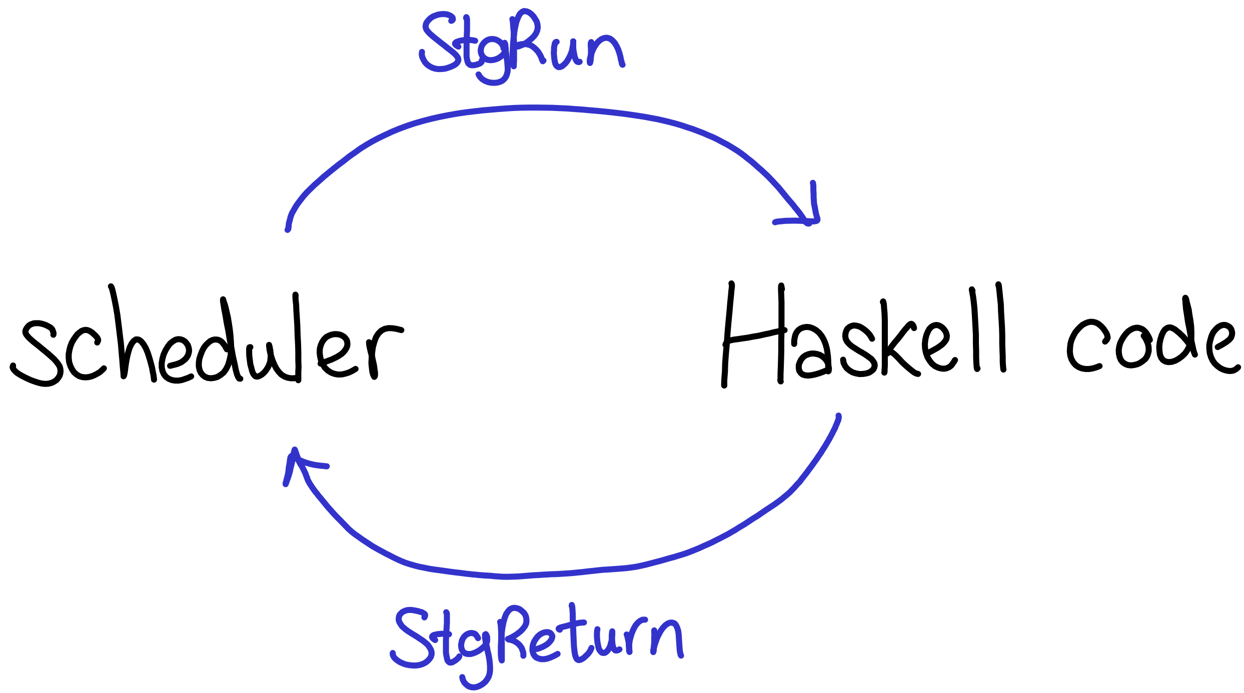
Haskell threads

- 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:

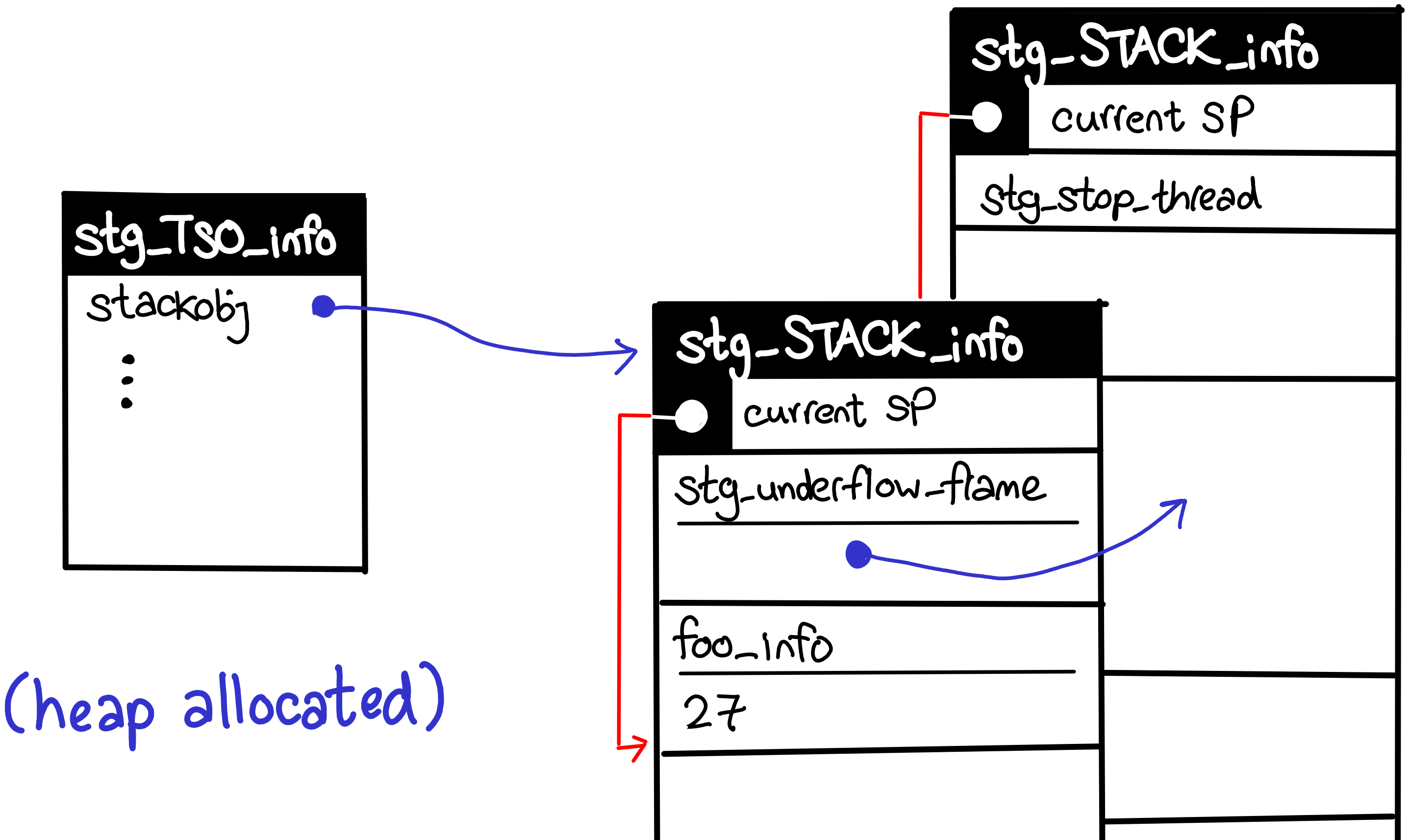
```
throwTo :: Exception e => ThreadId -> e -> IO ()
killThread :: ThreadId -> IO ()   -- = flip throwTo ThreadKilled
threadDelay :: Int -> IO ()       -- sleeps for # of µsec
myThreadId :: IO ThreadId
```


StgRun
scheduler
Haskell code
StgReturn

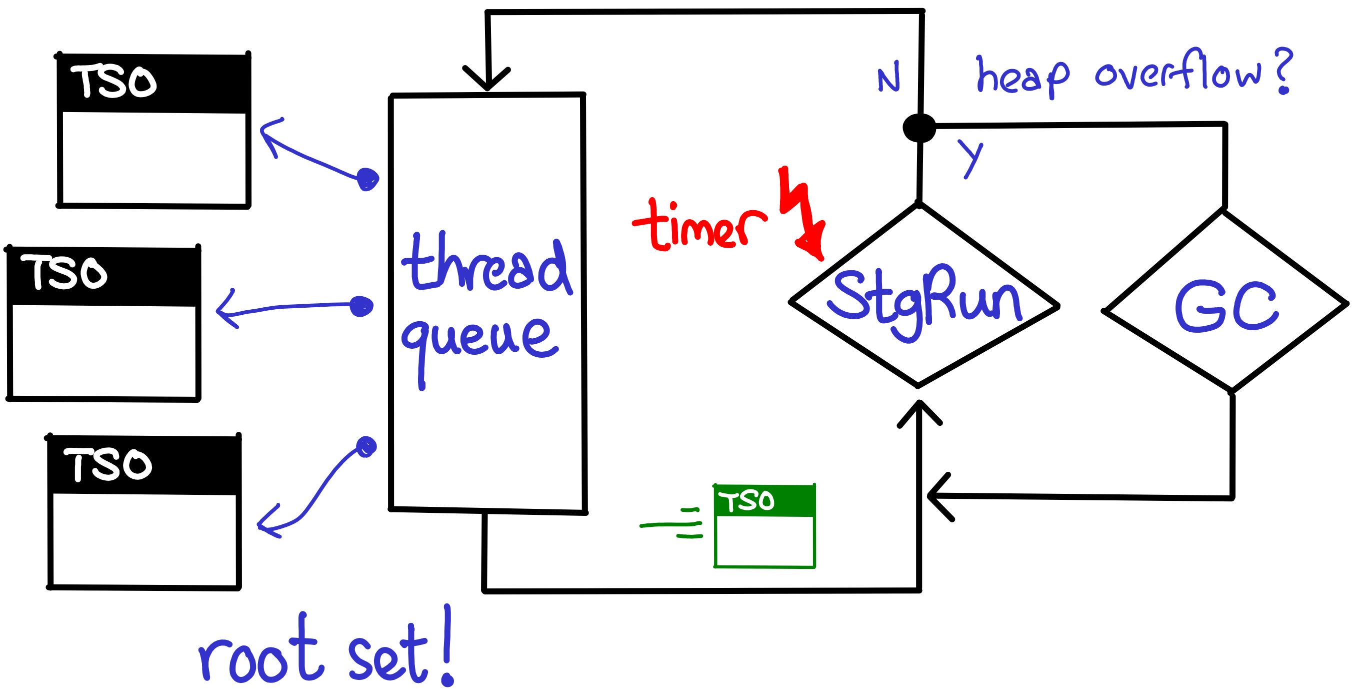
set to zero

```
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 ();
}
```

Anatomy of a thread

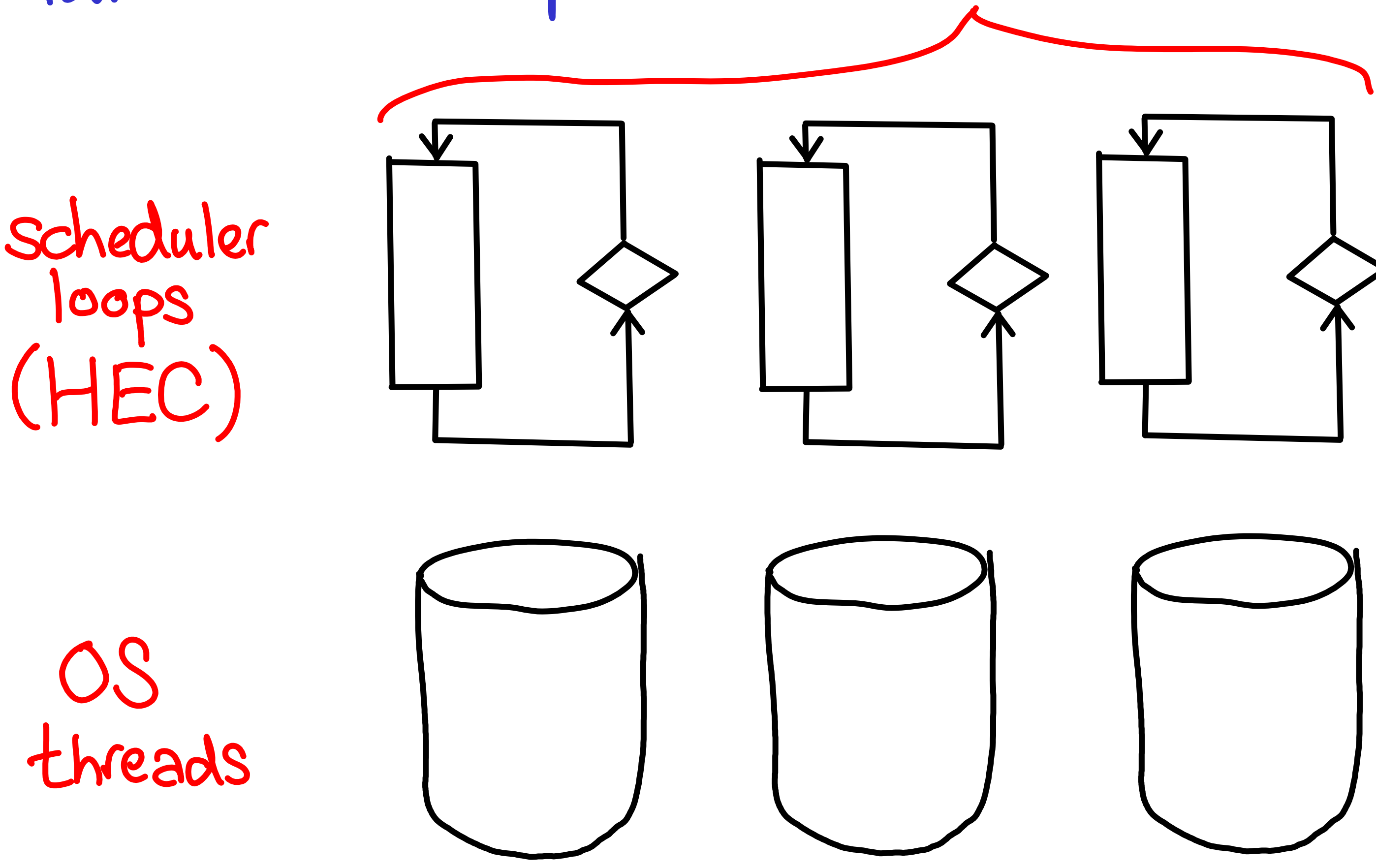


Single-threaded operation

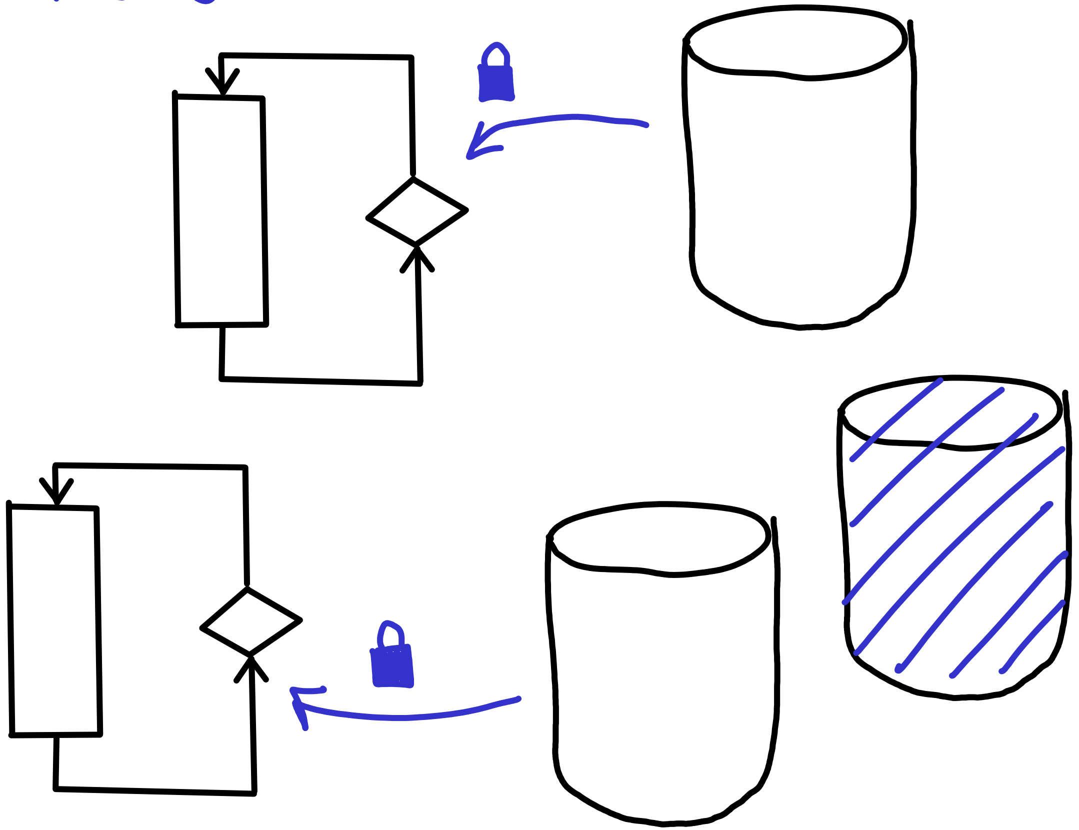


Scheduler Loop

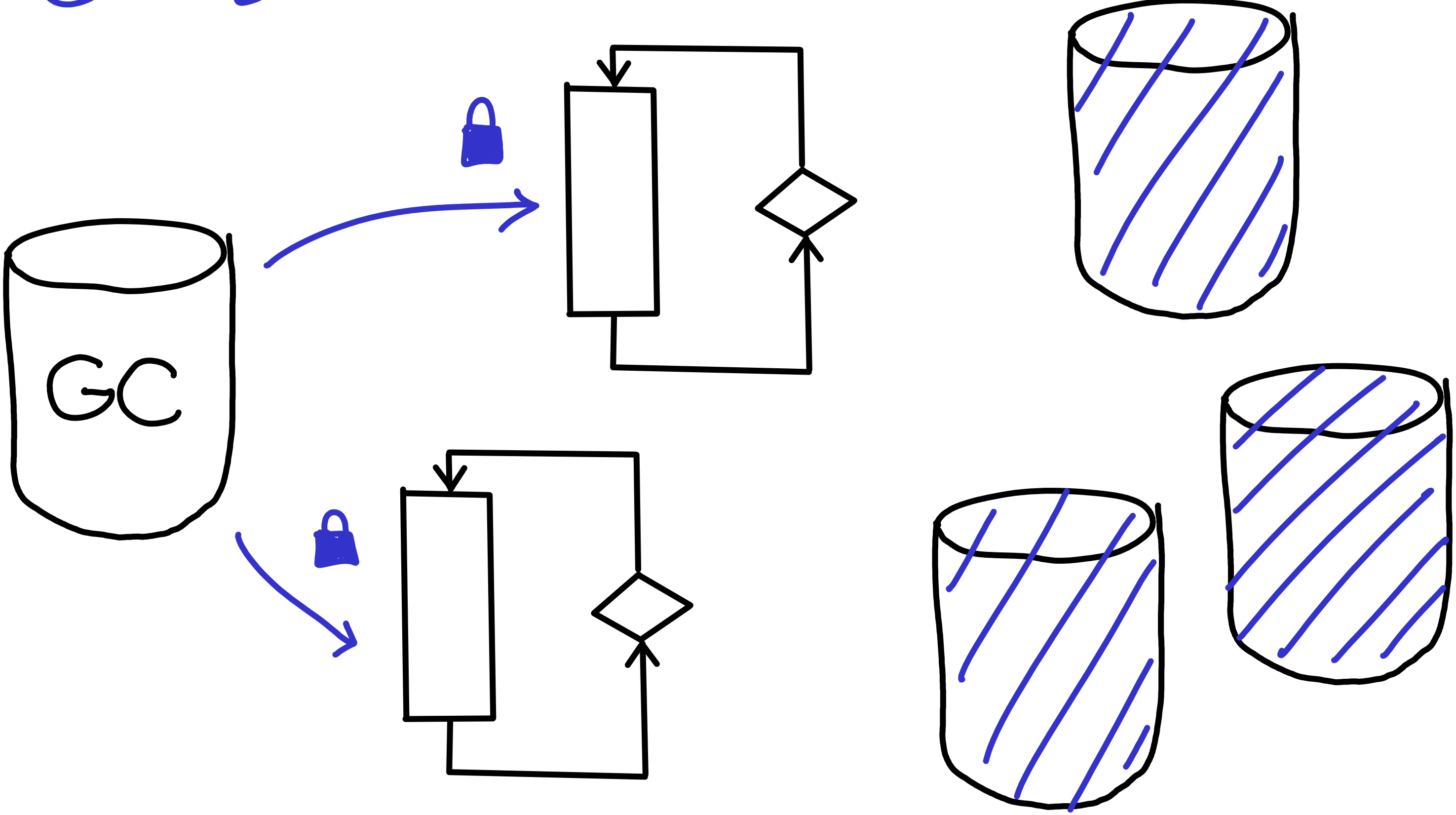
Multi-threaded operation -N3



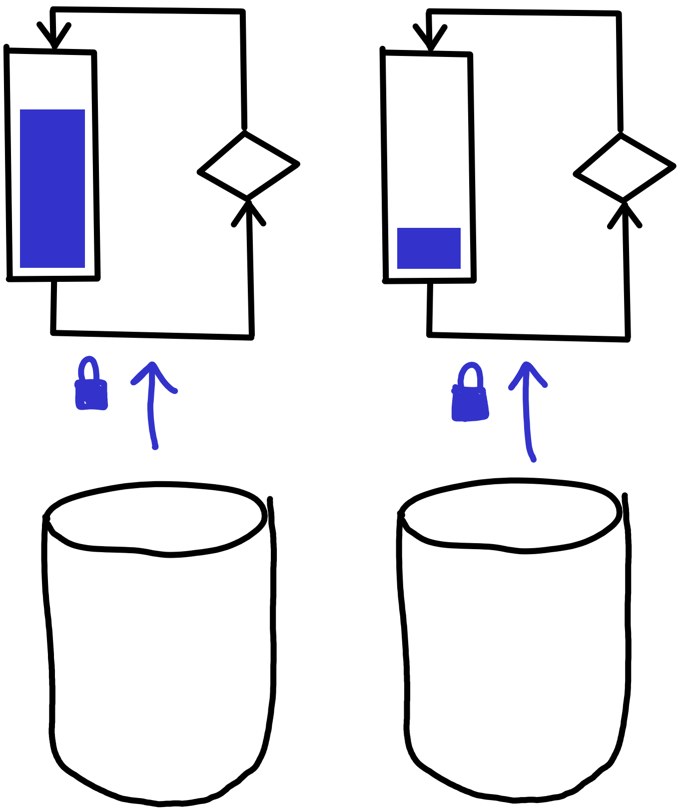
HECs are locks



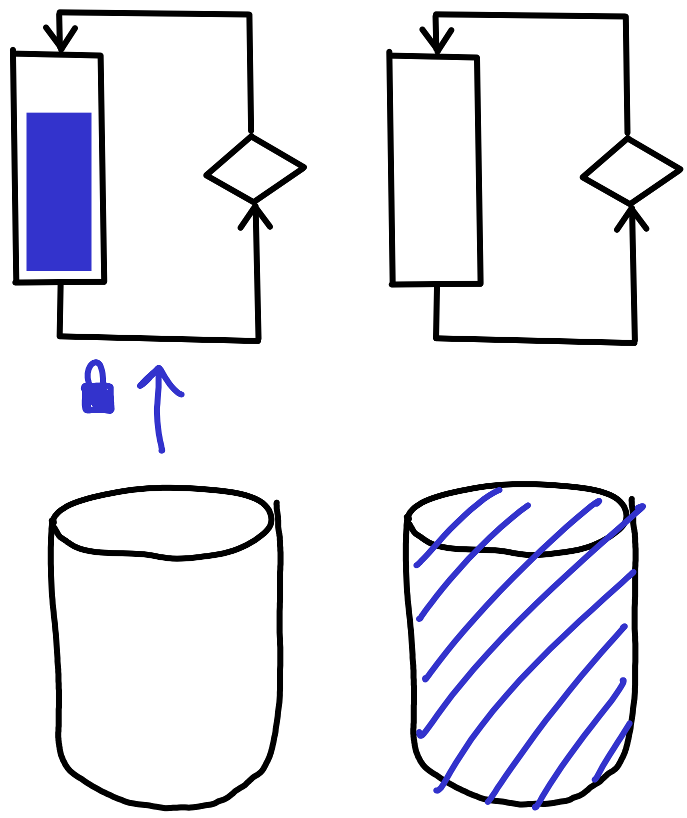
GC takes all locks



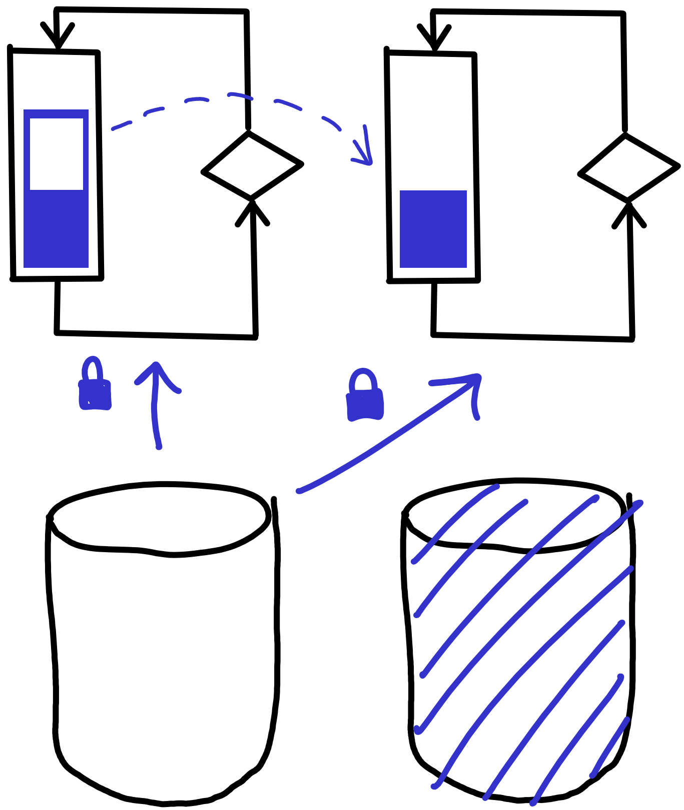
Work imbalance



Work imbalance

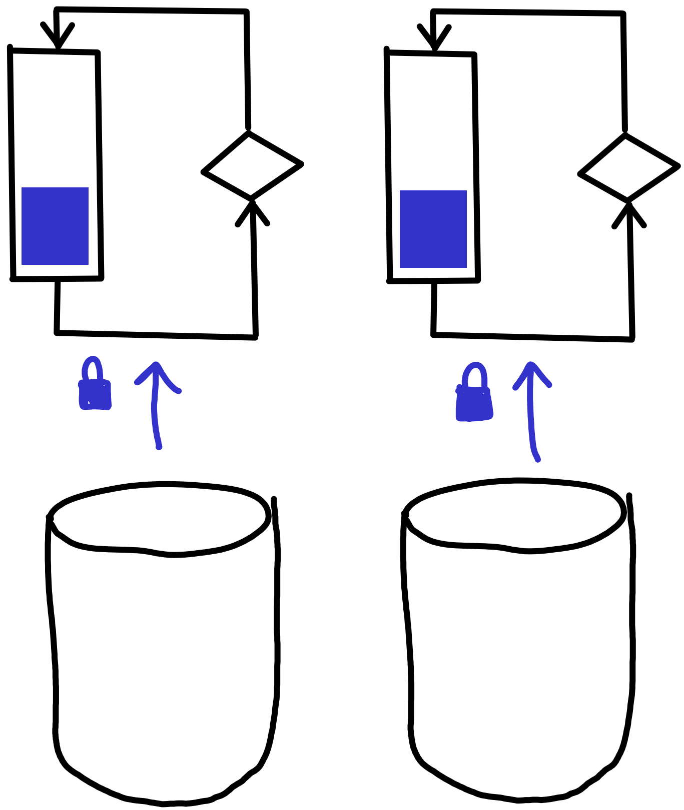


Work imbalance

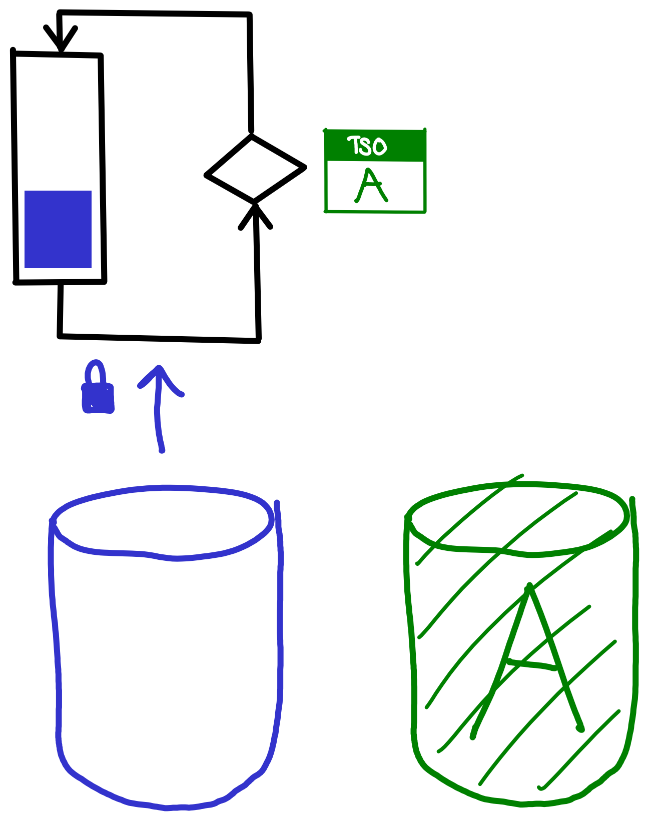


Work imbalance

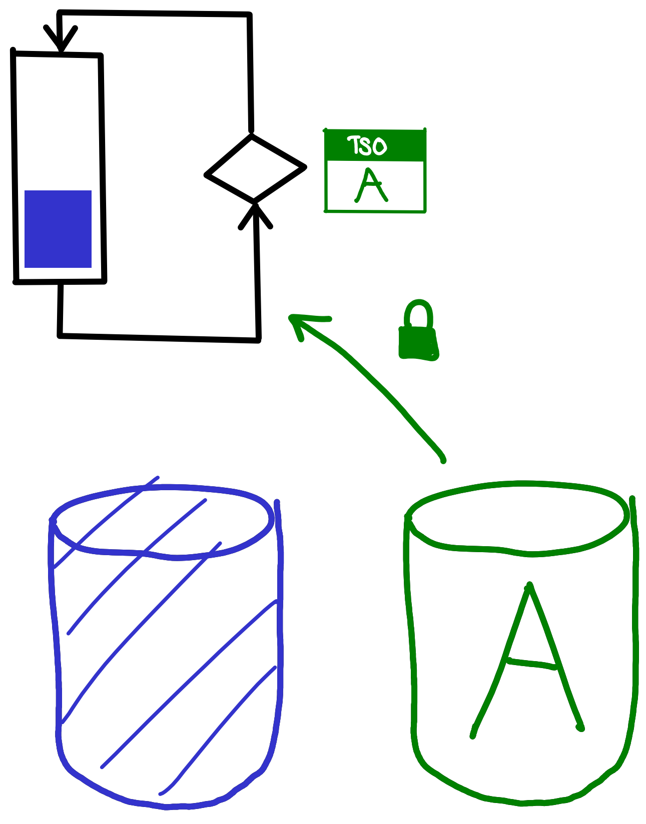
Throughput First!



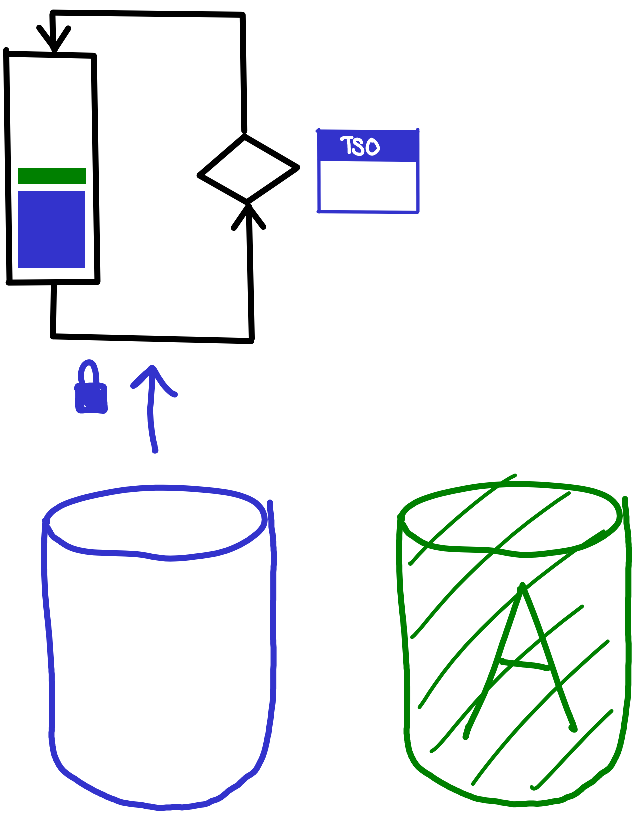
Bound threads



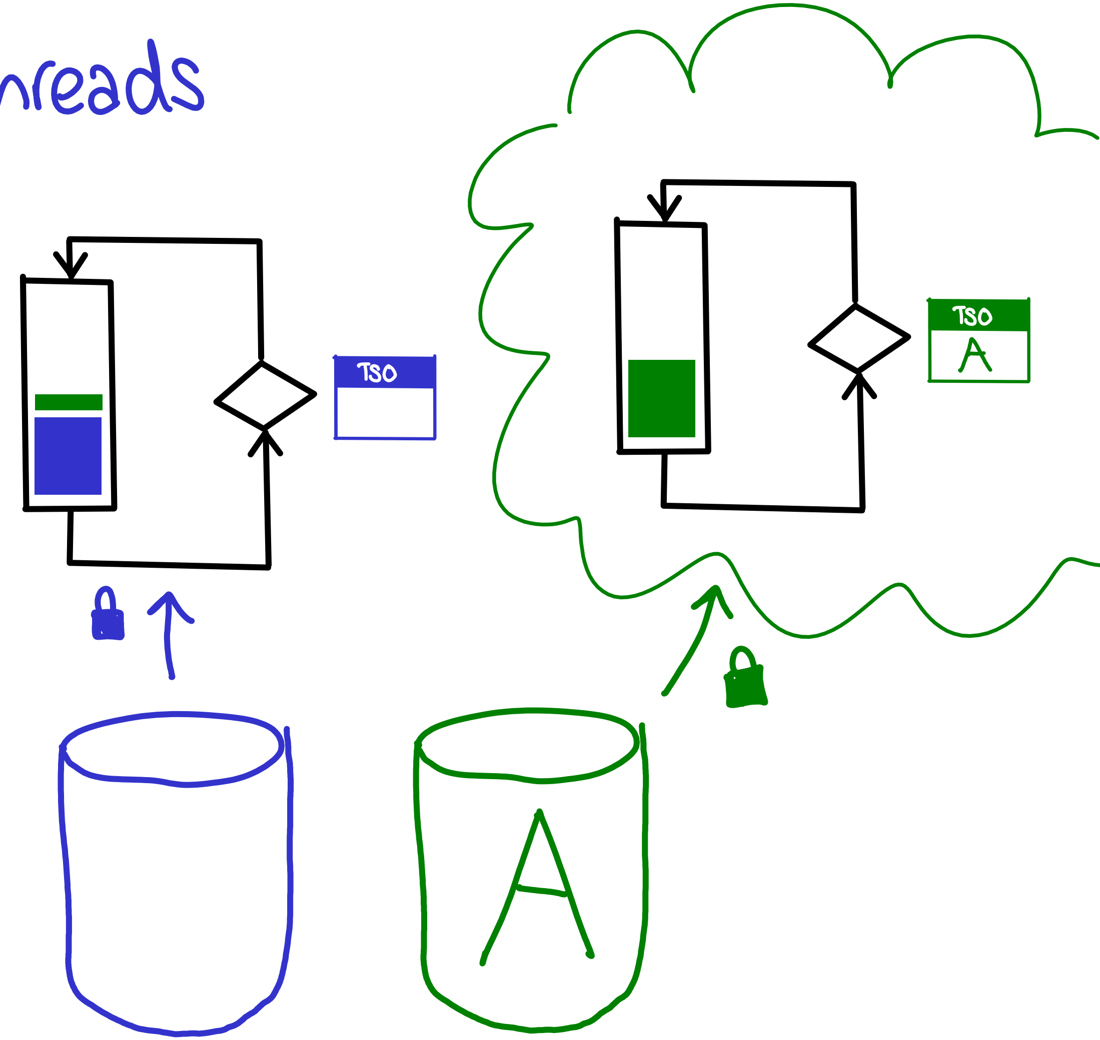
Bound threads



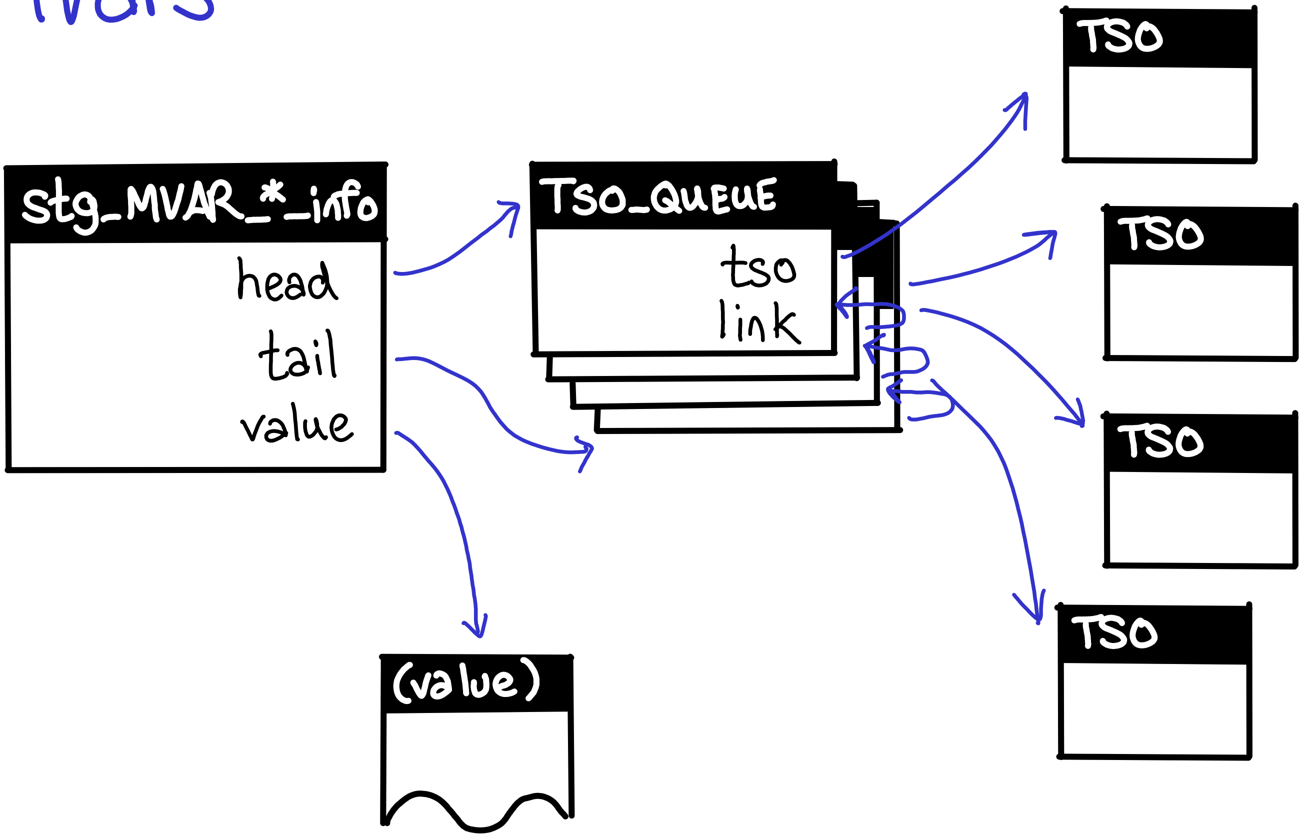
Bound threads



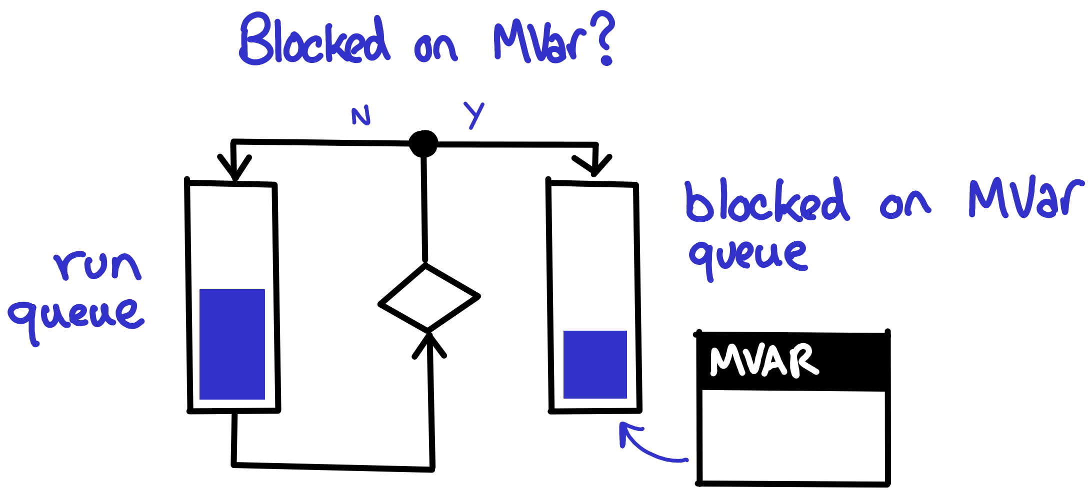
Bound threads



MVars


stg_MVAR_*_info
head
tail
value
TSO_QUEUE
tso
link
TSO
TSO
TSO
TSO
(value)

MVars



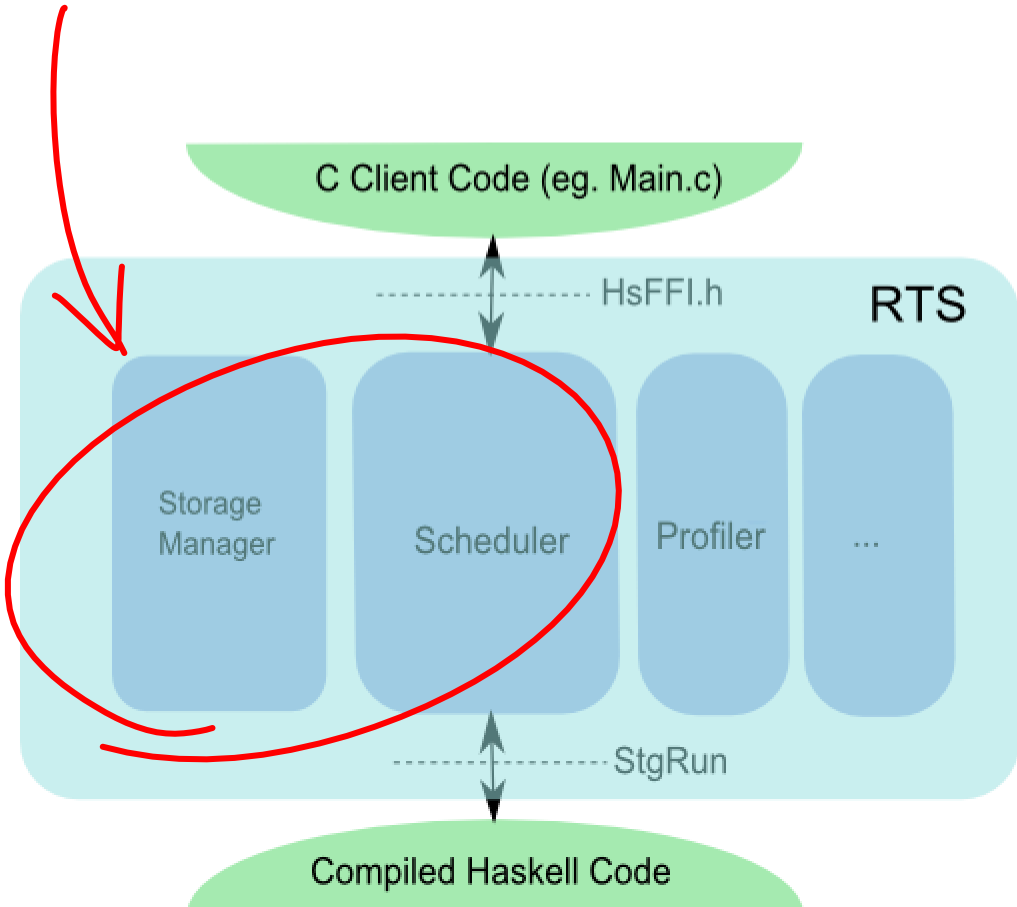
Fun fact: If the MVar becomes garbage, the threads in its queue die too

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

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 r 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