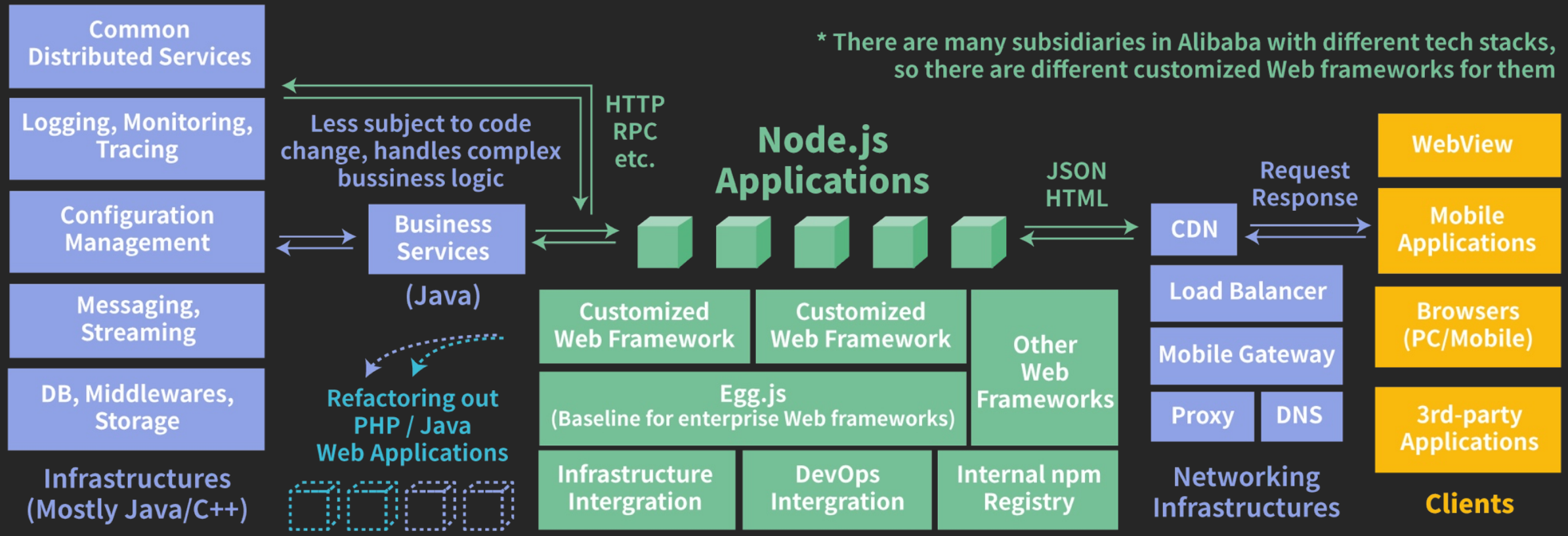

node
JS®
Interactive
NORTH AMERICA
Austin, Texas
November 28–December 2, 2016

Are Your V8 Garbage Collection Logs Speaking to You?

Joyee Cheung, Alibaba Cloud

What Prompted us to Look into them

Node.js at Alibaba



What Prompted Us to Look Into Them

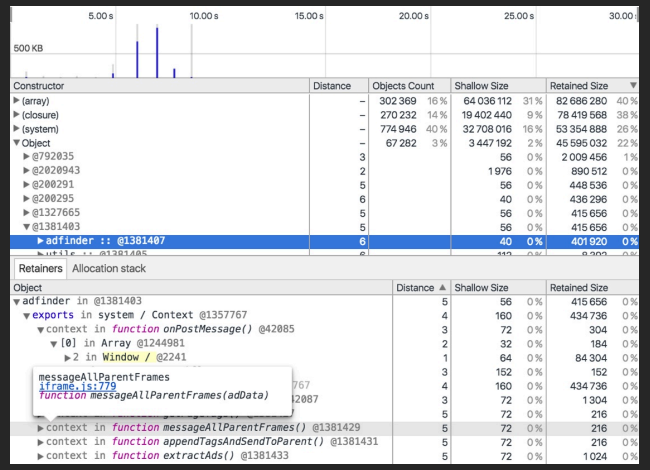
Why GC Logs?

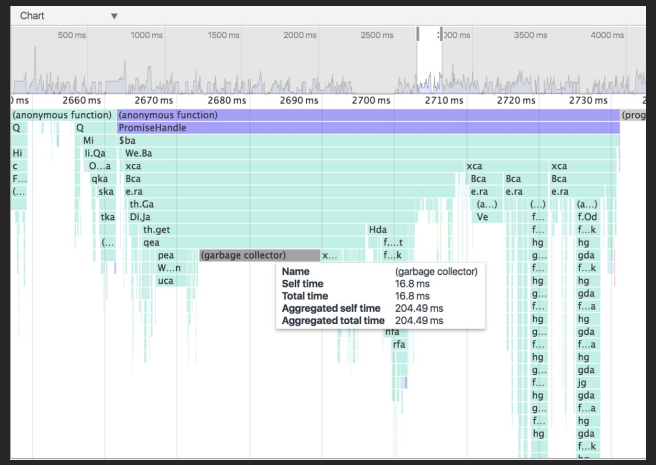
- **Java is the mainstream at Alibaba**
  - Chose Node.js for productivity, but it's meaningless without availability
  - Usually not the bottleneck, but with a huge traffic, every optimization counts
- **Garbage collection is important in long-running, data-intensive servers**
  - **High CPU usage**: hurts responsiveness
  - **High memory usage**: affects other processes
  - **Memory leaks**: misled by developers, crashes the process(OOM)
  - Java developers know their VM and its GC
  - ...
- **More about Node.js at Alibaba**
  - https://www.linux.com/blog/can-nodejs-scale-ask-team-alibaba

What Prompted Us to Look Into Them

Why GC Logs?

- **Tools from V8/Chromium are more about optimizing frontend applications**





- Heap snapshots and timelines are more about allocations, less about collections
- Lack of high-level overview of the heap, too much details

CPU profiles are more about function calls, less about objects

Hard to use in staging/production environment

What Prompted Us to Look Into Them

Why GC Logs?

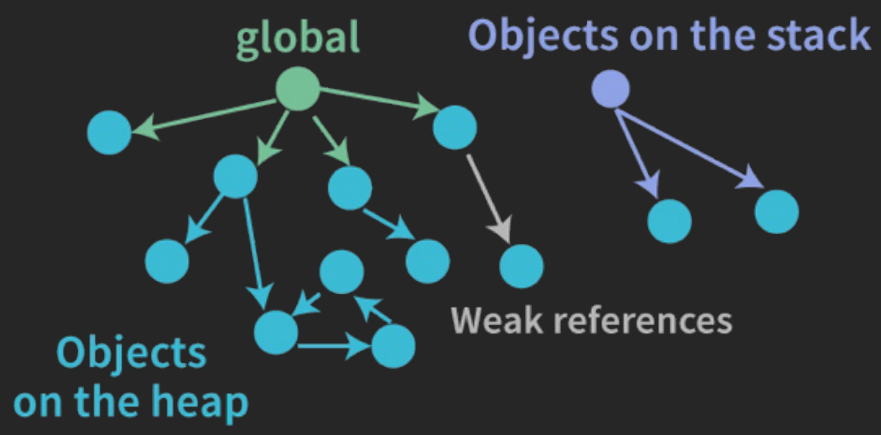

```
[15368:0x102004600]     6314 ms: Scavenge 181.6 (213.4) -> 180.5 (224.4) MB, 7.3 / 0.0 ms [allocation failure].
[15368:0x102004600] Memory allocator,   used: 229828 KB, available: 1236540 KB
[15368:0x102004600] New space,          used:   3903 KB, available:  12216 KB, committed:  32768 KB
[15368:0x102004600] Old space,          used:  62965 KB, available:   6373 KB, committed:  70612 KB
[15368:0x102004600] Code space,         used:   1270 KB, available:     43 KB, committed:   3072 KB
[15368:0x102004600] Map space,          used:    267 KB, available:    802 KB, committed:   1104 KB
[15368:0x102004600] Large object space, used: 116408 KB, available: 1235499 KB, committed: 119200 KB
[15368:0x102004600] All spaces,         used: 184814 KB, available: 1254935 KB, committed: 226756 KB
[15368:0x102004600] External memory reported:     16 KB
[15368:0x102004600] Total time spent in GC  : 63.6 ms
```

- **V8 provides garbage collection logs**
  - Not documented, not available in the devtools, need to use command line args
  - Higher-level than heap profiles, lower-level than the number of memory usuage
  - Help verify the fixes by displaying patterns of GC
  - Not silver bullet, but nice to have
- **This talk is based on the versions of V8 used by Node v4 & v6 LTS**
  - LTS plan stablizes the V8 versions for each LTS and its GC logs

An Introduction to V8 GC

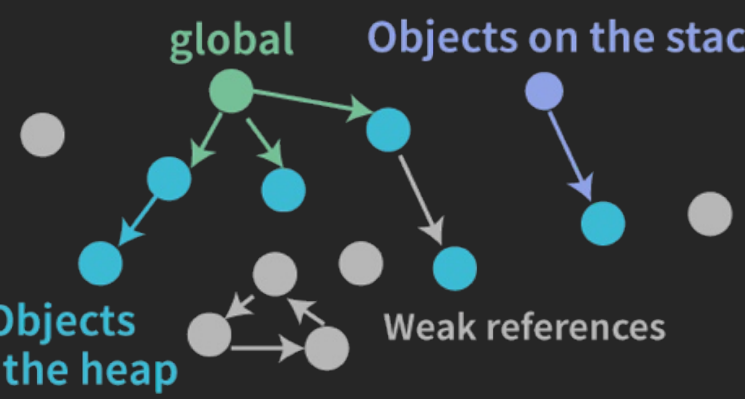
What is Garbage Collection

In the begining




```
function foo() {
  var bar = global.bar = { a: 1 };
  bar.x = { c: 3 };
  var baz = { b: 2 };
  ...
}
```

After a while...

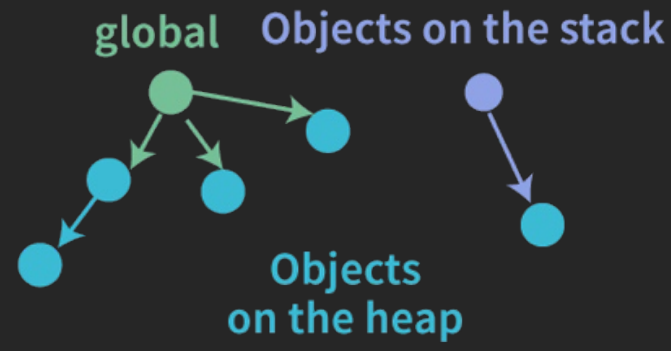



```
bar.x = null;
delete f;

var d = new SomeClass(10);
```

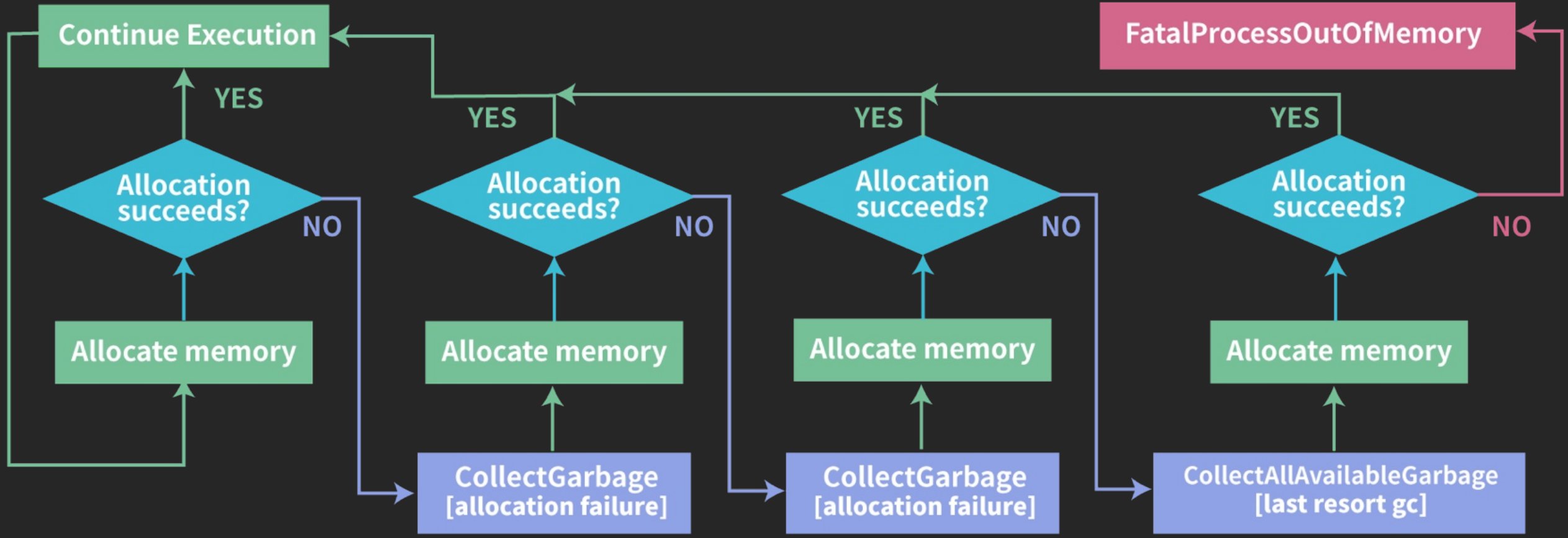
We need to allocate new objects,
but there is not enough memory

Collect the garbage



An Introduction to V8 GC

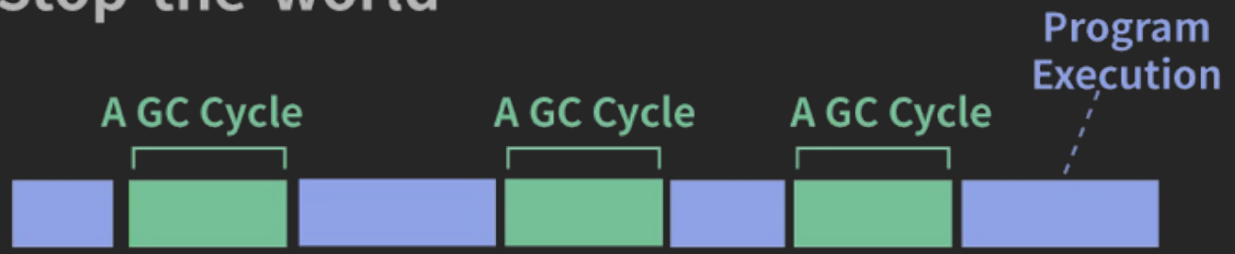
How Most GCs are triggered



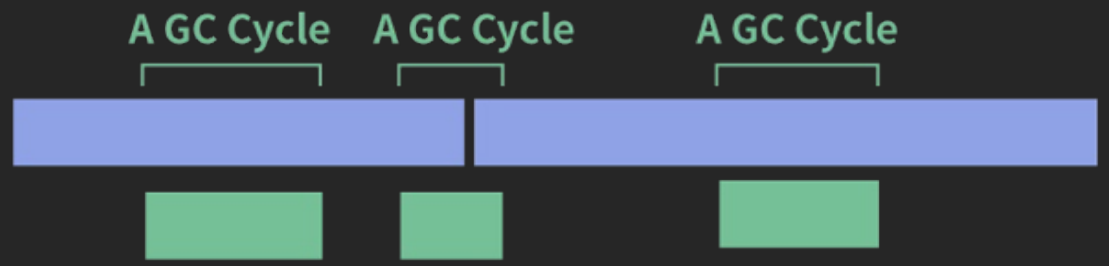
An Introduction to V8 GC

Stop-The-World, Parallel, Concurrent GC

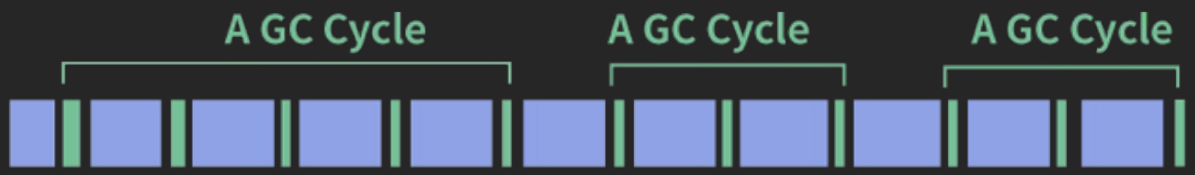
Stop-the-world



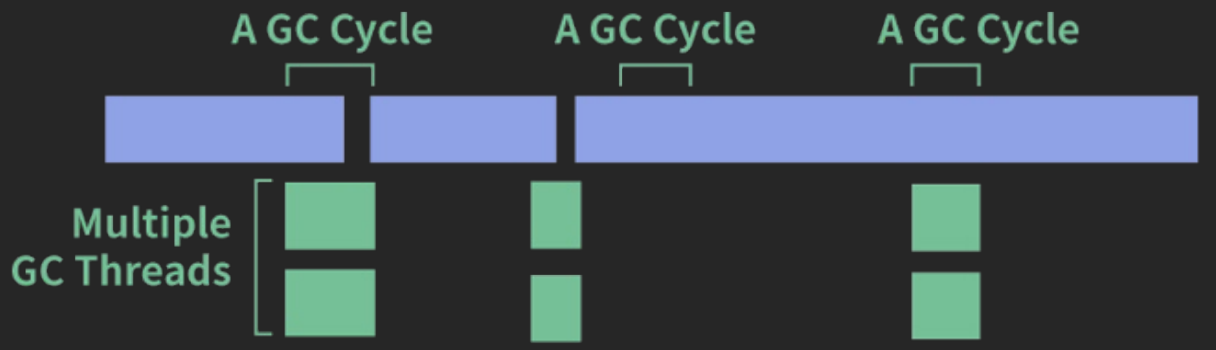
Concurrent



Incremental



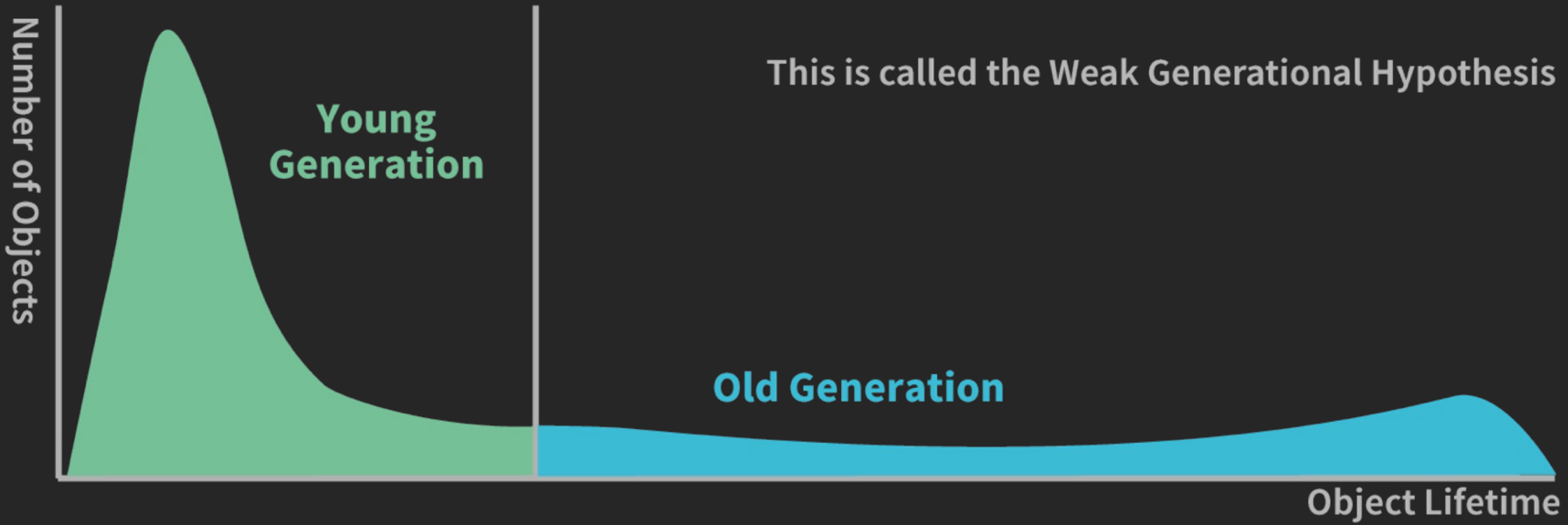
Parallel + Concurrent



V8 uses different strategies for different stages of GC

An Introduction to V8 GC

Generational GC



This is called the Weak Generational Hypothesis

Most objects die young

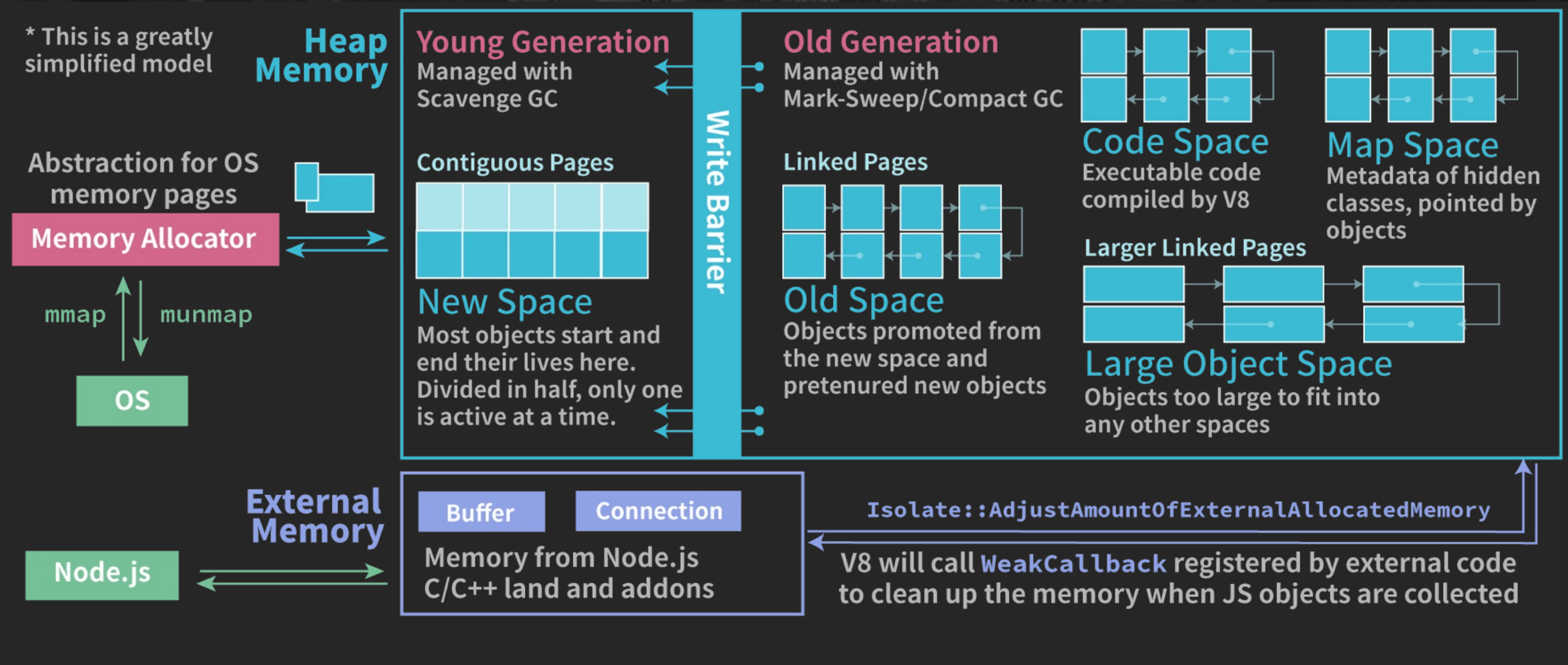
We can reclaim memory in time by checking on them more frequently

Those who don't tend to live forever

We can save CPU by checking on them less frequently

An Introduction to V8 GC

Heap Organization



An Introduction to V8 GC

Heap Limits(Default)

- deps/v8/src/heap/heap.h: Heap::Heap()

```
// semispace_size_ should be a power of 2 and old_generation_size_ should
// be a multiple of Page::kPageSize.
max_semi_space_size_(8 * (kPointerSize / 4) * MB),
initial_semispace_size_(Page::kPageSize),
max_old_generation_size_(700ul * (kPointerSize / 4) * MB),
initial_old_generation_size_(max_old_generation_size_ /
                             kInitalOldGenerationLimitFactor),
old_generation_size_configured_(false),
max_executable_size_(256ul * (kPointerSize / 4) * MB),
```

16MB

1MB

1.4GB

0.7GB

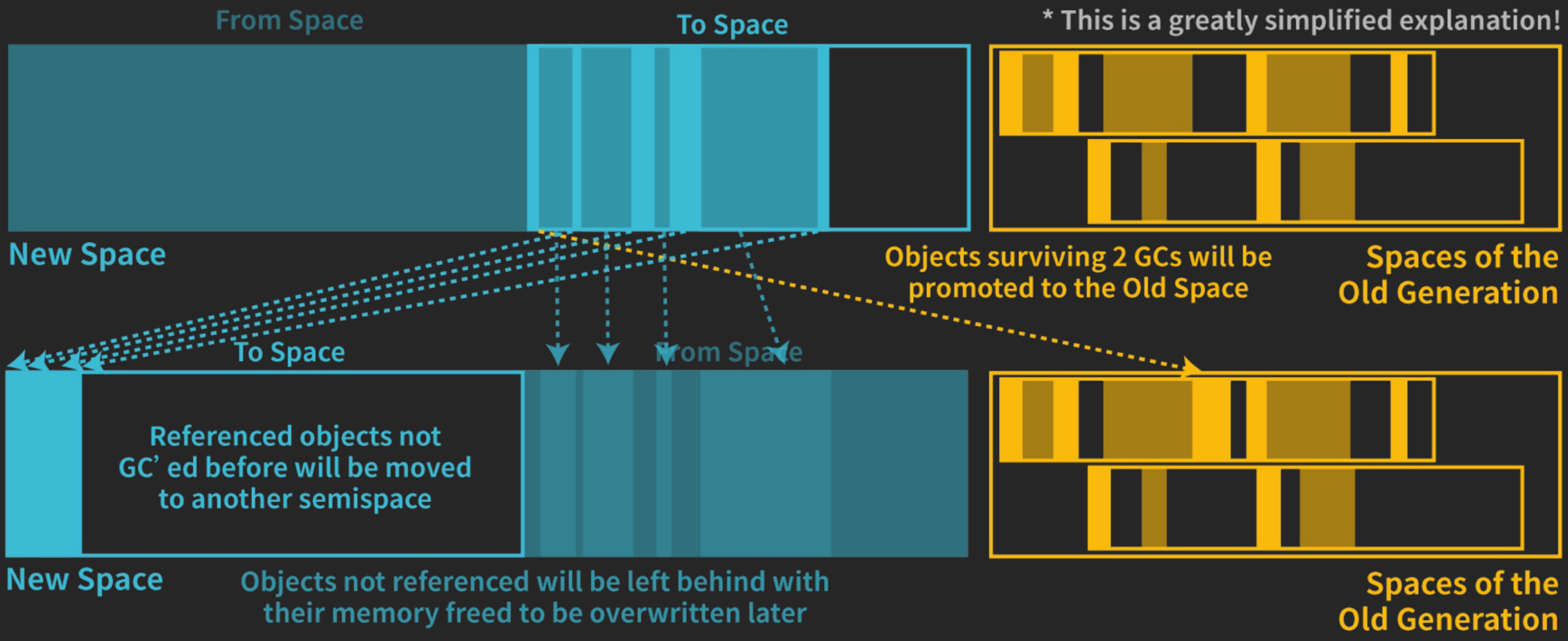
512MB

```
const int kPointerSize   = sizeof(void*);     // NOLINT
```

- **Can be configured via command line arguments**
  - --max_old_space_size, --max_semi_space_size, --max_executable_space_size

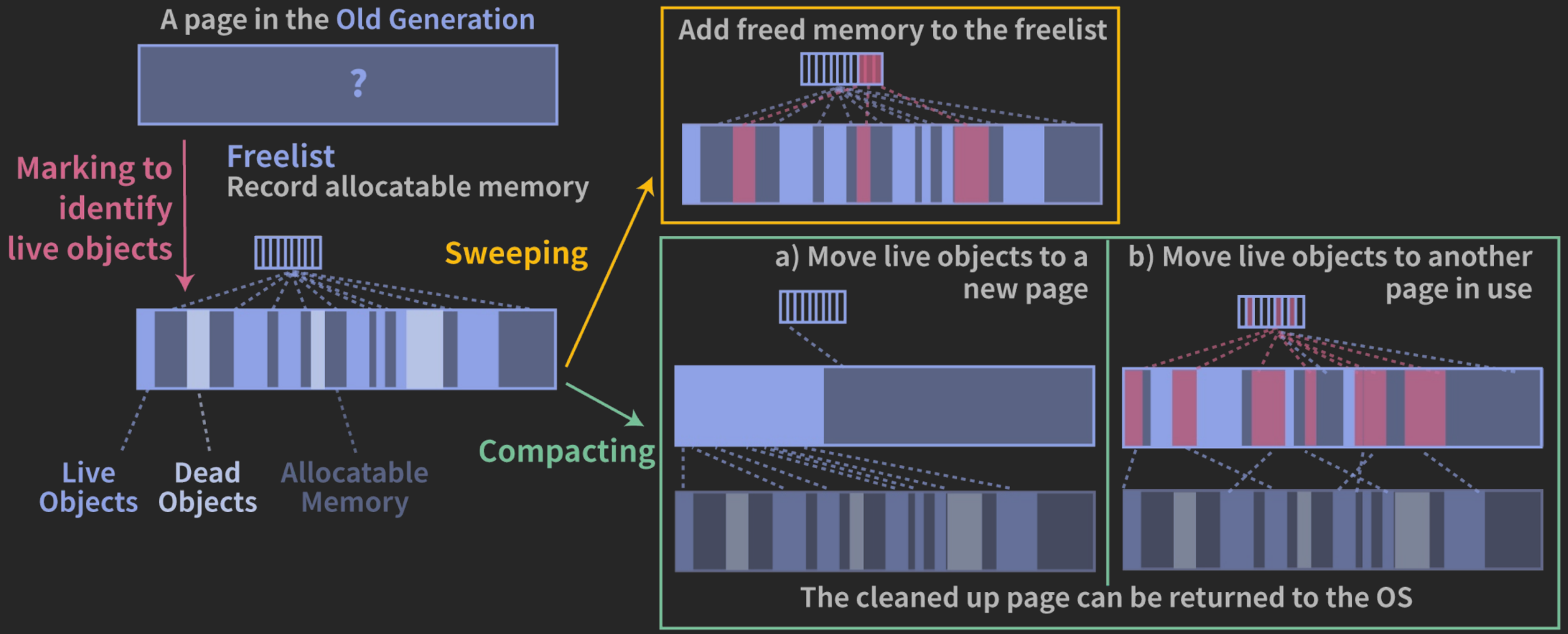
An Introduction to V8 GC

New Generation – Scavenge GC



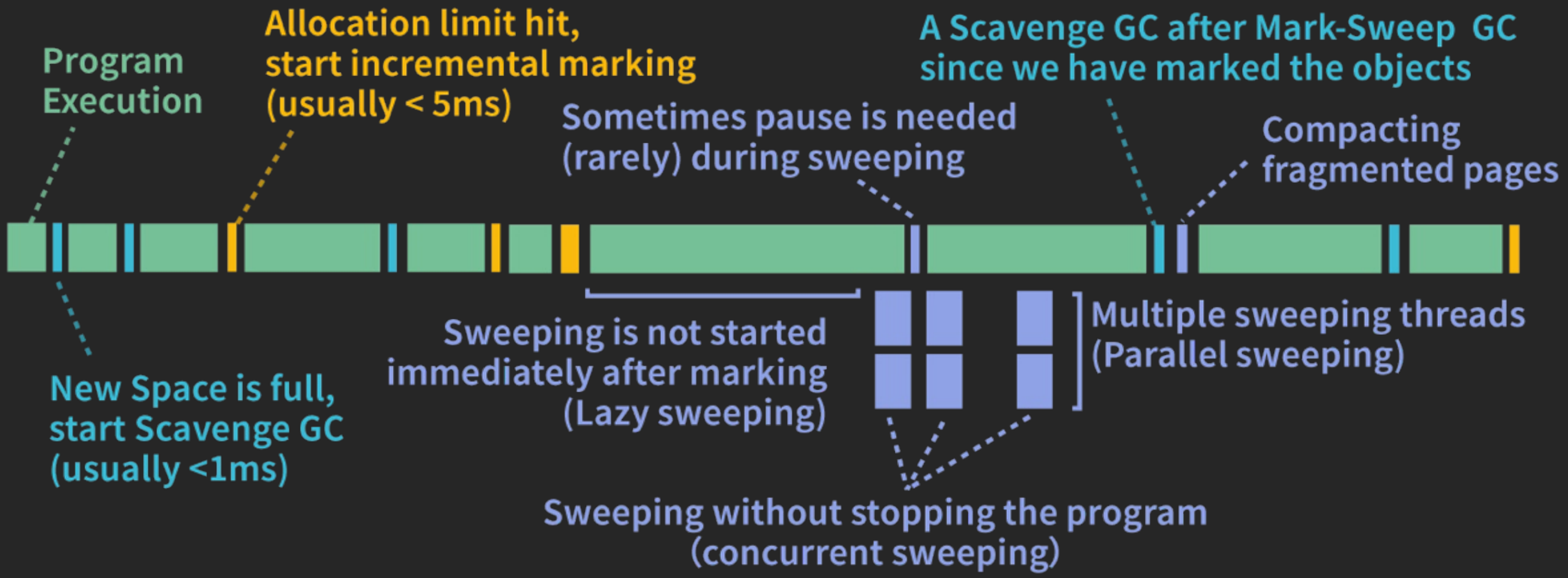
An Introduction to V8 GC

Old Generation – Mark-Sweep/Compact GC



An Introduction to V8 GC

GC Pauses in V8



How to Get The GC Logs

- Plenty of options (`node --v8-options| grep gc`)
- We'll be focusing on
  - `--trace_gc`
    - Print a line briefly describing when, how, what and why for each GC
  - `--trace_gc_verbose`
    - Combined with `--trace_gc`, summarizes about each space and the heap after each GC
  - `--trace_gc_nvp`
    - Print name-value pairs describing how each step of GC goes
- In case you do want to read the code
  - `deps/v8/src/heap`, start with `gc-tracer.cc`

How to Read The GC Logs

- `--trace_gc`

WHERE: [PID:Isolate] **WHAT: Type of GC**

HOW LONG:
GC pause/Time spent on external memory

```
[43417:0x102004600]    11361 ms: Scavenge 279.0 (320.1) -> 276.6 (323.1) MB, 4.4 / 0.0 ms [allocation failure].
[43417:0x102004600]    11454 ms: Scavenge 313.8 (355.9) -> 312.6 (365.9) MB, 7.8 / 0.0 ms [allocation failure].
[43417:0x102004600]    11647 ms: Scavenge 395.7 (439.3) -> 393.7 (444.3) MB, 5.5 / 0.0 ms [allocation failure].
[43417:0x102004600]    11759 ms: Scavenge 438.0 (483.4) -> 436.7 (492.4) MB, 8.6 / 0.0 ms [allocation failure].
[43417:0x102004600]    11974 ms: Scavenge 513.6 (560.3) -> 511.6 (565.3) MB, 6.8 / 0.0 ms [allocation failure].
[43417:0x102004600]    12132 ms: Scavenge 562.0 (609.9) -> 560.6 (618.9) MB, 9.9 / 0.0 ms [allocation failure].
[43417:0x102004600]    12318 ms: Scavenge 632.4 (682.2) -> 630.4 (688.2) MB, 8.4 / 0.0 ms [allocation failure].
[43417:0x102004600]    12449 ms: Scavenge 685.3 (736.6) -> 683.8 (744.6) MB, 8.4 / 0.0 ms [allocation failure].
[43417:0x102004600]    12613 ms: Scavenge 752.0 (804.8) -> 750.1 (811.8) MB, 9.0 / 0.0 ms [allocation failure].
[43417:0x102004600]    12765 ms: Scavenge 807.6 (862.5) -> 806.1 (874.5) MB, 7.3 / 0.0 ms (+ 8.8 ms in 130 steps
since last GC) [allocation failure].
[43417:0x102004600]    12861 ms: Mark-sweep 841.5 (906.8) -> 259.7 (317.8) MB, 10.7 / 0.0 ms (+ 10.2 ms in 210
steps since start of marking, biggest step 2.2 ms) [GC interrupt] [GC in old space requested].
```

WHY

WHEN: Time since the process has started

HOW MUCH:
Size of all objects(Size of memory allocated from OS)

How to Read The GC Logs

- `--trace_gc_verbose && --trace_gc`

All memory reserved for this space (inside dedicated pages)

Memory V8 allocated from the OS

```
[43748:0x103000000] Memory allocator,  used:  78948 KB, available: 1387420 KB
[43748:0x103000000] New space,         used:   5135 KB, available:   2924 KB, committed:  16384 KB
[43748:0x103000000] Old space,         used:  14119 KB, available:    817 KB, committed:  15316 KB
[43748:0x103000000] Code space,        used:   1445 KB, available:     88 KB, committed:   3072 KB
[43748:0x103000000] Map space,         used:    268 KB, available:    800 KB, committed:   1104 KB
[43748:0x103000000] Large object space, used:  39063 KB, available: 1386379 KB, committed:  40000 KB
[43748:0x103000000] All spaces,        used:  60031 KB, available: 1391010 KB, committed:  75876 KB
[43748:0x103000000] External memory reported:    16 KB
[43748:0x103000000] Total time spent in GC  : 16.7 ms
[43748:0x103000000] Heap growing factor 1.7 based on mu=0.970, speed_ratio=76 (gc=1873432, mutator=24752)
[43748:0x103000000] Grow: old size: 21301 KB, new limit: 53307 KB (1.7)
```

Maintained by external C++ code

Adjust heap growing factor and the allocation limit that triggers the next incremental marking

How to Read The GC Logs

- `--trace_gc_nvp`

ms=Mark-Sweep, s=Scavenge

```
[51548:0x102004600]    35050 ms: pause=8.4 mutator=1181.6 gc=ms reduce_memory=0 clear=0 clear.code_flush=0.0
clear.dependent_code=0.0 clear.global_handles=0.0 clear.maps=0.0 clear.slots_buffer=0.0 clear.store_buffer=0.1
clear.string_table=0.2 clear.weak_cells=0.0 clear.weak_collections=0.0 clear.weak_lists=0.1 evacuate=7.5
evacuate.candidates=0.0 evacuate.clean_up=0.0 evacuate.copy=7.2 evacuate.update_pointers=0.4
evacuate.update_pointers.between_evacuated=0.0 evacuate.update_pointers.to_evacuated=0.0
evacuate.update_pointers.to_new=0.3 evacuate.update_pointers.weak=0.0 external.mc_prologue=0.0
external.mc_epilogue=0.0 external.mc_incremental_prologue=0.0 external.mc_incremental_epilogue=0.0
external.weak_global_handles=0.0 finish=0.1 mark=0.2 mark.finish_incremental=0.1 mark.prepare_code_flush=0.0
mark.roots=0.0 mark.weak_closure=0.0 mark.weak_closure.ephemeral=0.0 mark.weak_closure.weak_handles=0.0
mark.weak_closure.weak_roots=0.0 mark.weak_closure.harmony=0.0 sweep=0.1 sweep.code=0.0 sweep.map=0.0
sweep.old=0.1 incremental_finalize=0.0 steps_count=124 steps_took=9.9 longest_step=2.8
finalization_steps_count=2 finalization_steps_took=0.3 finalization_longest_step=0.2
incremental_marking_throughput=14624500 total_size_before=251989056 total_size_after=171775680
holes_size_before=59072 holes_size_after=8720000 allocated=56826536 promoted=3614424 semi_space_copied=6769264
nodes_died_in_new=63 nodes_copied_in_new=2 nodes_promoted=0 promotion_ratio=30.6% average_survival_ratio=88.4%
promotion_rate=97.3% semi_space_copy_rate=57.3% new_space_allocation_throughput=6537.5
context_disposal_rate=0.0 compaction_speed=2137628
```

What We've Found

- Tons of Scavenge GC in New Space eating up the CPU
  - Cache misses caused by inappropriate caching strategies
  - Excessive deep clones/object merging
- Large Object Space keeps growing
  - Serialization/deserialization of huge RPC payload
- Code Space keeps growing
  - Bugs in templating engines
- Old Space keeps growing
  - Closures in long connections

Tools

- **Our parser**
- Up on NPM
- For people who already have applications running in production

https://github.com/aliyun-node/v8-gc-log-parser

http://alinode.aliyun.com/blog
(translation coming soon)

Visualization coming soon!

Tools

- `--trace_gc_object_stats`
  - Feed the output to https://mlippautz.github.io/v8-heap-stats/
    - Requires Node >= v7
  - More about the memory usuage, less about GC
- `--track_gc_object_stats`
  - Combined with `--log_timer_events`, writes a log file to disk
  - Can be viewed with chrome://tracing (Incomplete???)
  - Pending PR: https://github.com/nodejs/node/pull/9304

THANKS

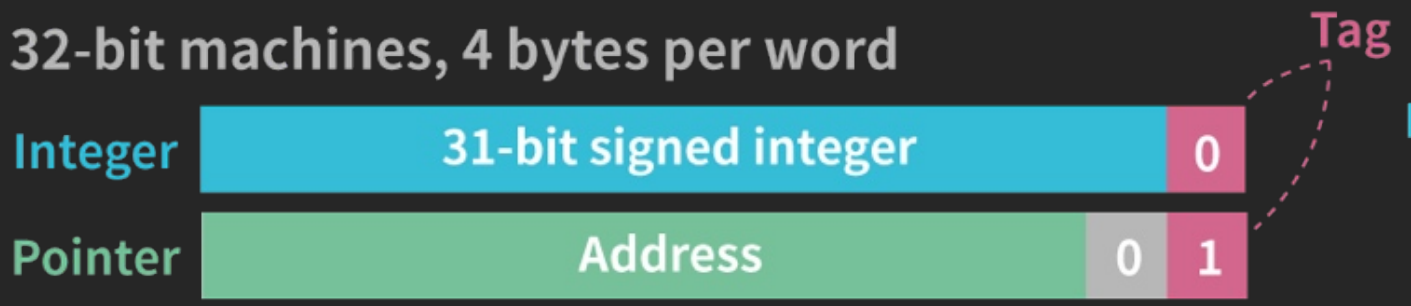
Twitter/Github: @joyeecheung
joyeec9h3@gmail.com
https://alinode.aliyun.com

An Introduction to V8 GC

Accurate GC

SMI: Smuggling Data in Pointers

32-bit machines, 4 bytes per word



Integers out of the 31-bit range will be boxed into objects

A 4-bytes-aligned address % 4 === 0

The 2nd last bit must be 0

64-bit machines, 8 bytes per word

Integers out of the 32-bit range will be boxed into objects

An 8-bytes-aligned address % 8 === 0. The 2nd and 3rd last bits must be 0

We can know if a word contains a pointer or an integer for sure by looking at the tag instead of being conservative about the content. Hence the GC is accurate.