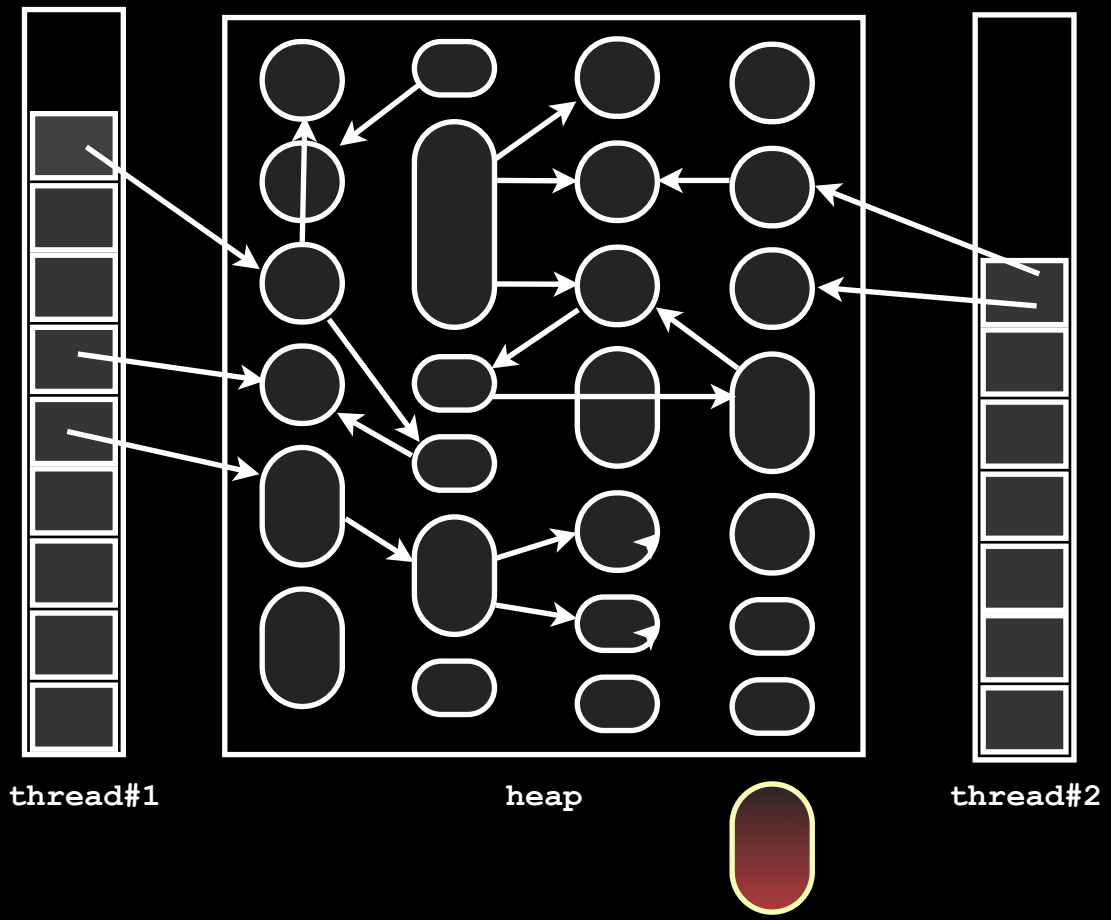
Garbage Collection

Garbage Collection

- A Garbage Collector (GC) is an algorithm that automatically finds unused objects in the memory of an application and prepares them for reuse
- GC frees programmers from worrying about the exact lifetime of objects and
  ensures that the heap will not be corrupted by access to previously freed data
- *... but introduces pauses that may be* ***O****(heap) and can increase the memory required.*
  Moreover, pauses occur at unpredictable times, especially in concurrent programs

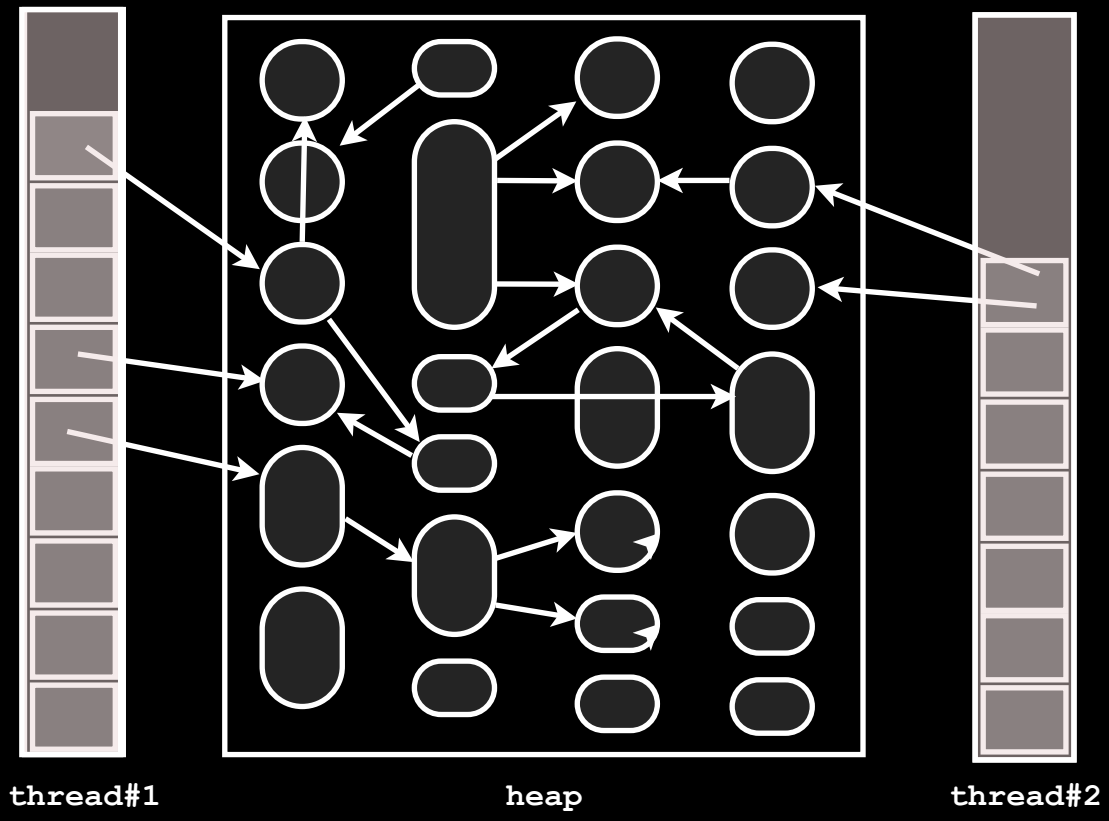
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

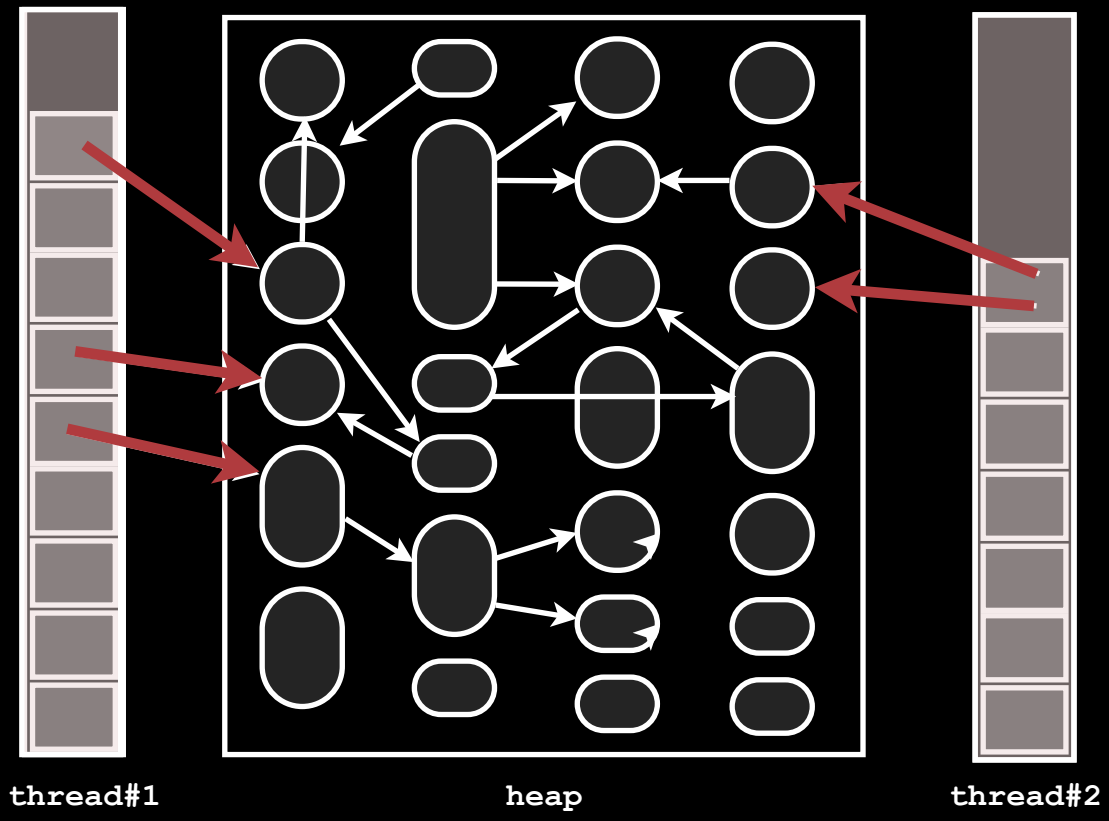
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

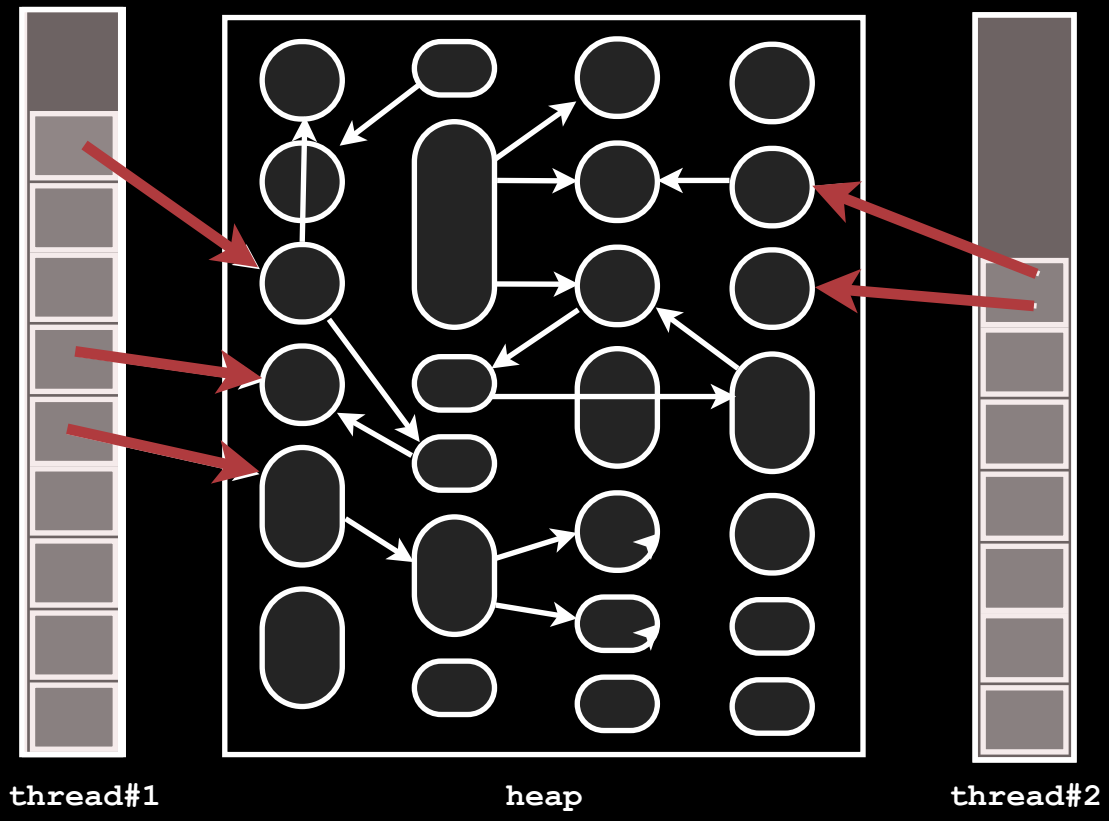
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

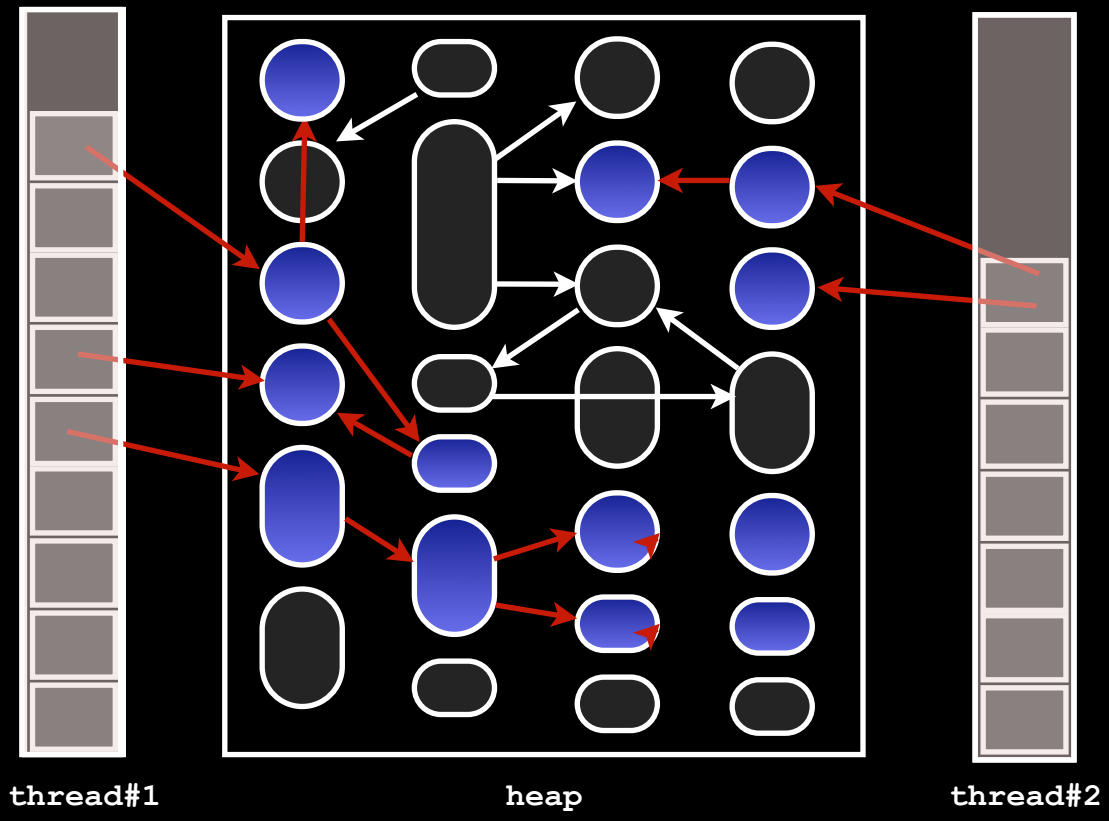
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

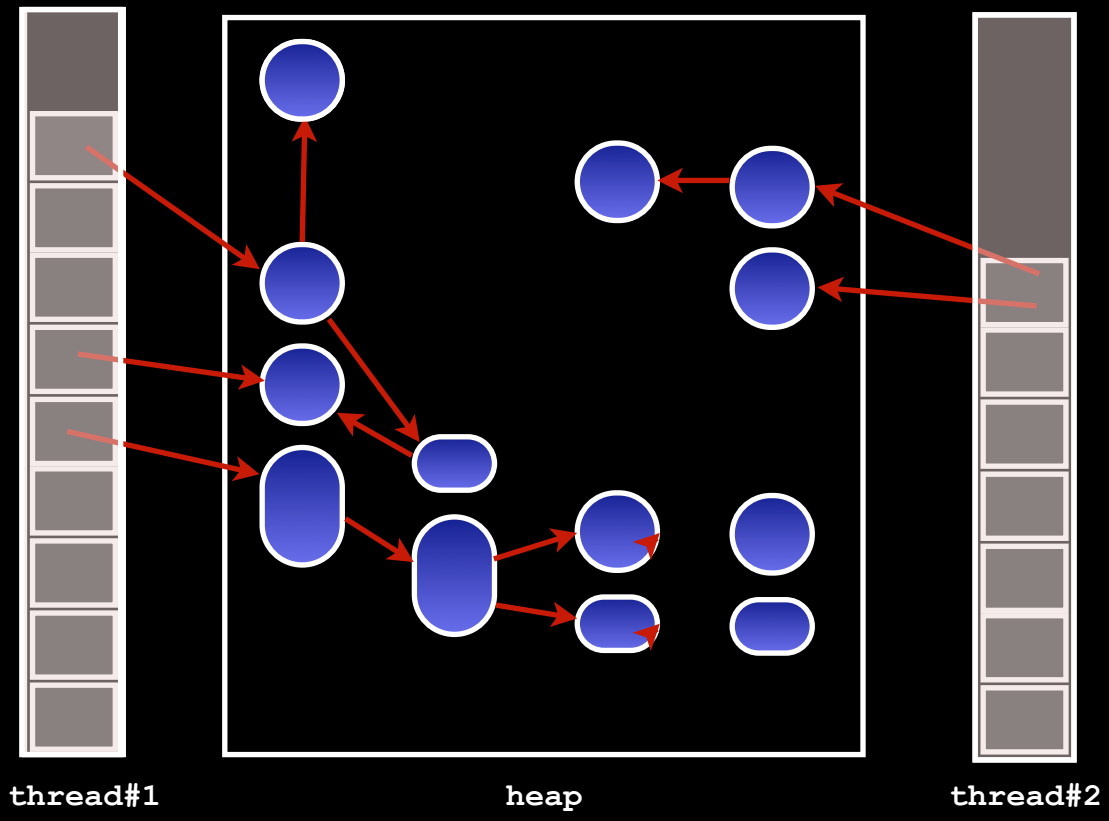
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

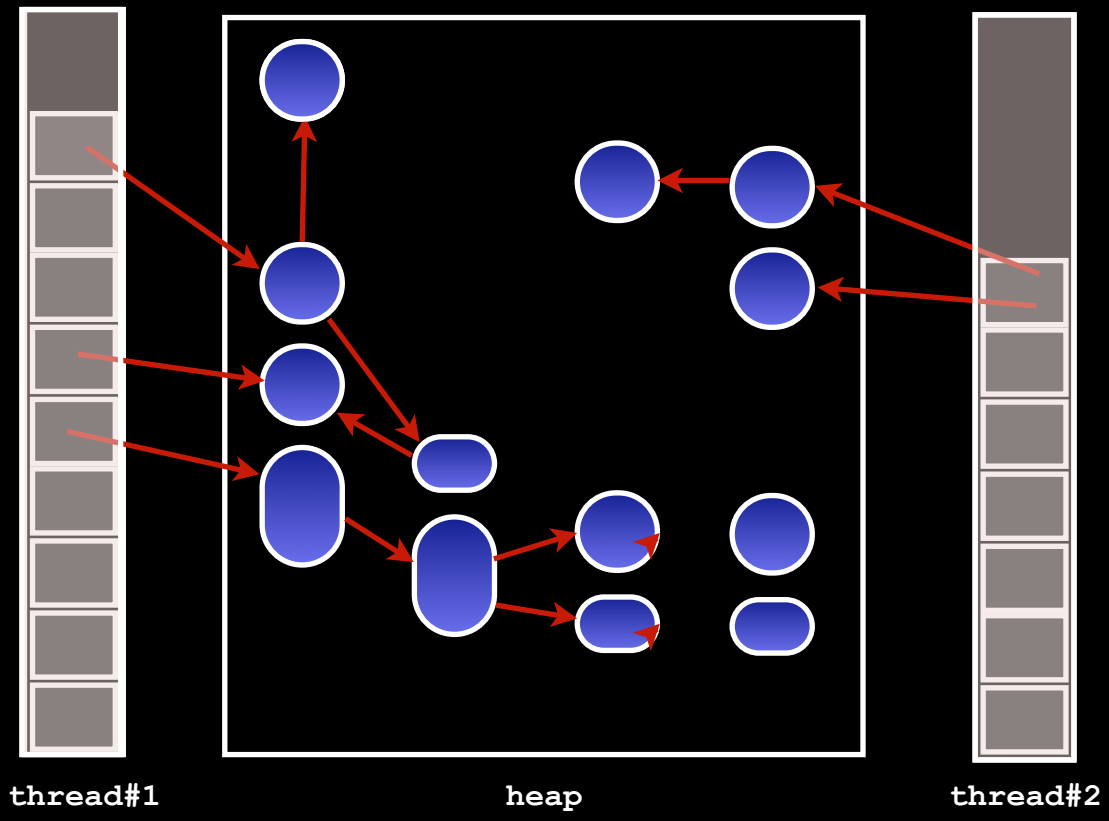
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

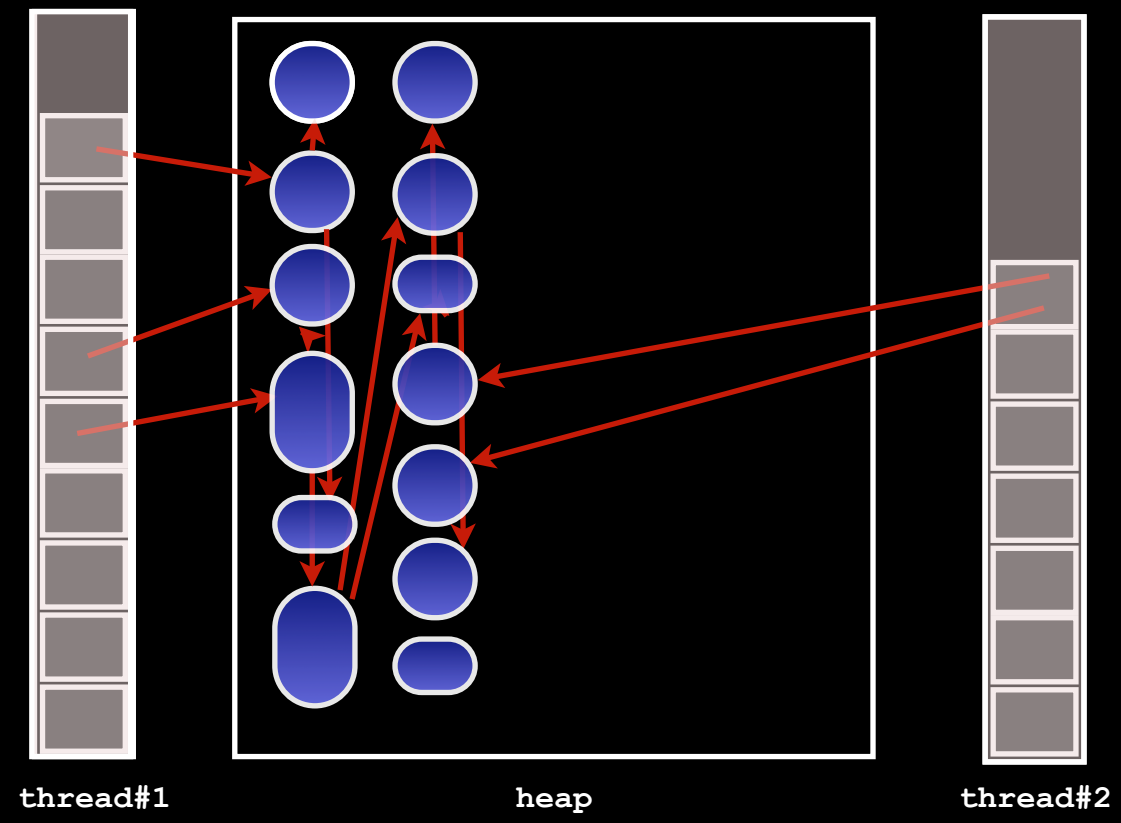
Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

Garbage Collection



Phases

- Mutation
- Stop-the-world
- Root scanning
- Marking
- Sweeping
- Compaction

GC is Easy

- If responsiveness is not an issue,
  the GC can complete in one long pause under the assumption that there is no interleaved application activity
- Marking is easy
  if the graph does not change while you are searching it.
- Copying/compacting objects and fixing up the heap is easy
  if the application is prevented from accessing the heap

Real-time Garbage Collection

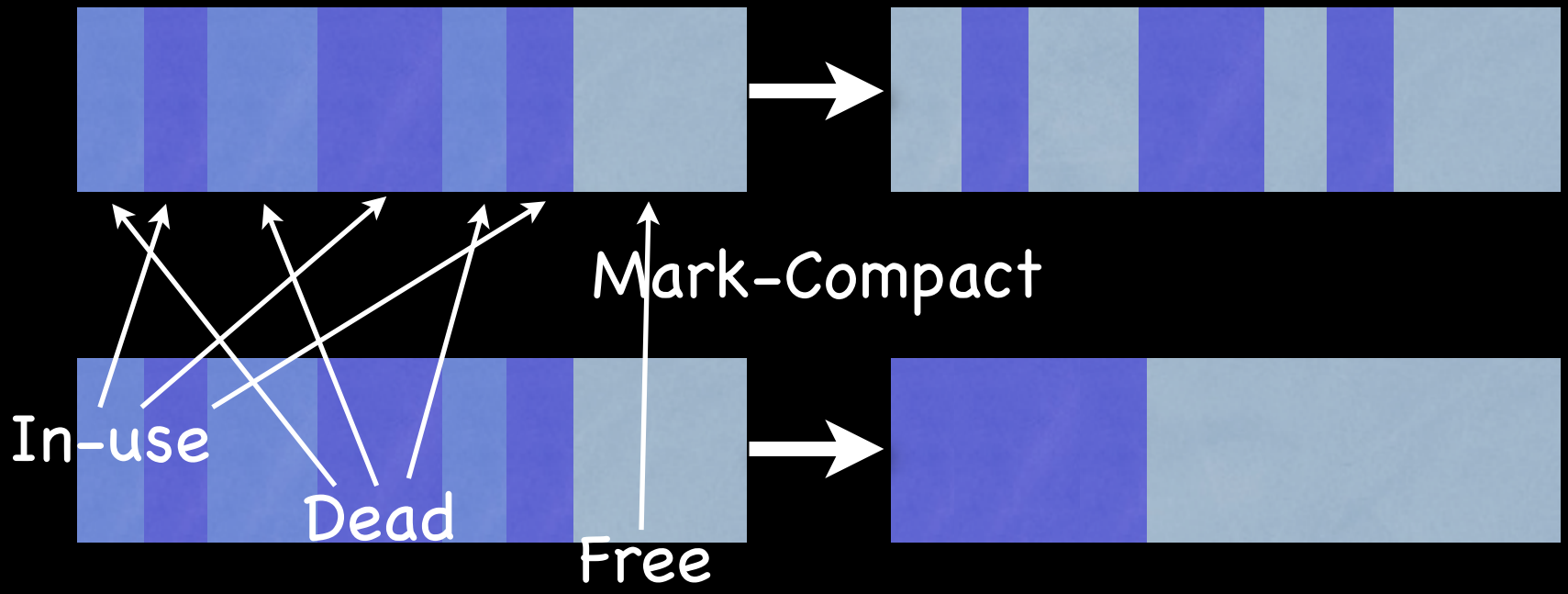
Real-time GC

- A Real-time GC must provide time and space predictability
  - provide a performance model that can be used to guarantee that programs do not run out of memory or experience pauses that violate their timing constraints
- A Real-time-GC must support defragmentation of the heap if it is to be used with long-lived applications
- Multi-processor support is unavoidable
- Throughput should not degrade overly

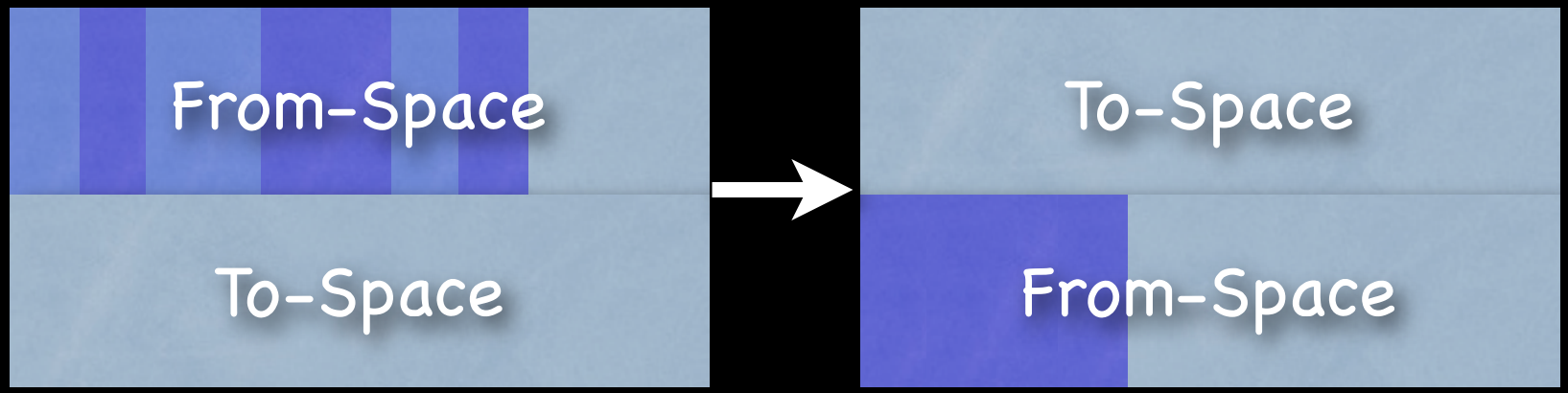
Main Collector Types

- Reference counting
  - keep count of incoming references. Requires auxiliary GC to reclaim cycles.
- Mark-Sweep
  - one phase to mark used objects, another to sweep unused space. No copying is performed.
- Mark-Compact
  - one phase to mark used objects, another to copy all used objects to one side of the heap.
- Copying
  - one phase to simultaneously mark and evacuate all used objects; the space that previously held all objects is then completely free.

Mark-Sweep



Two-space Copying



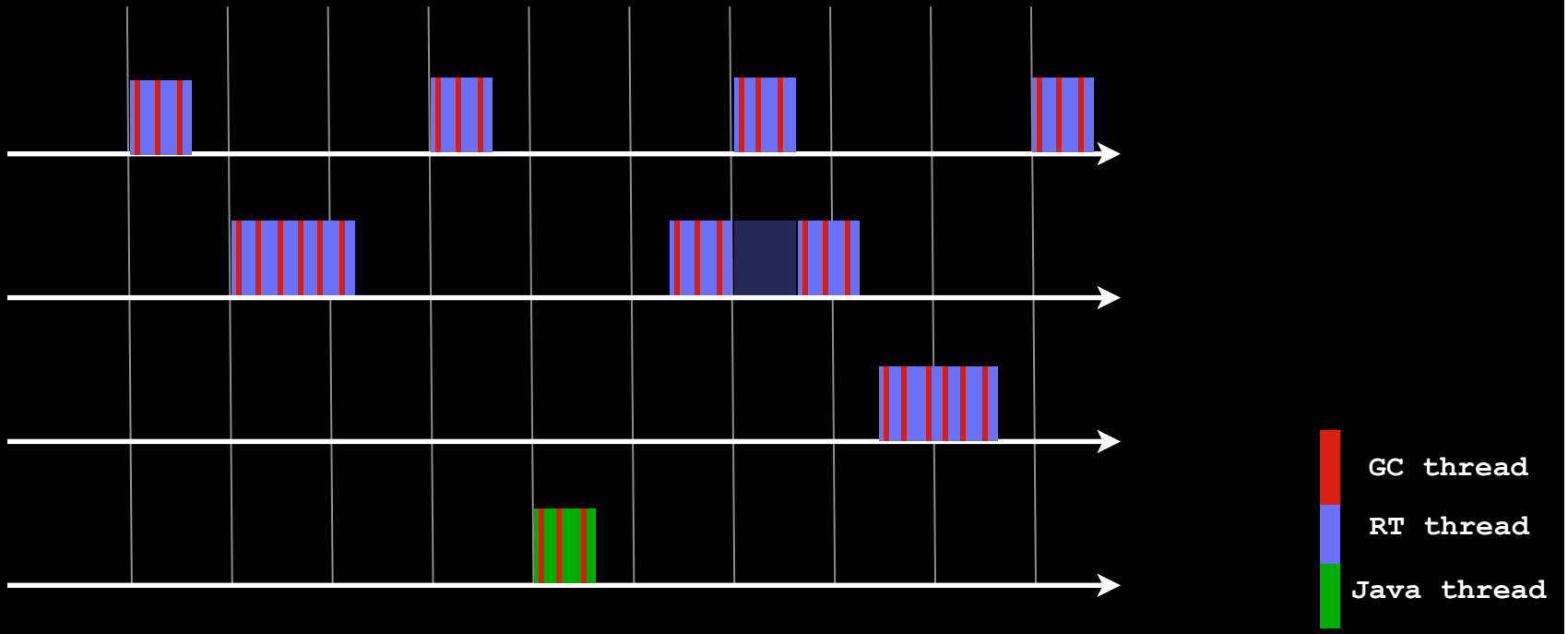
RTGC: state of the art

- Sort-of Real-time GCs:
  - Too many to list, Baker (1978) was first
- Really Real-time GCs:
  - JamaicaVM (1999) - *work-based*
  - Henriksson (1998) - *slack-based*
  - Metronome (2003) - *time-based*

Work-based RTGC

Baker'78, Siebert'99

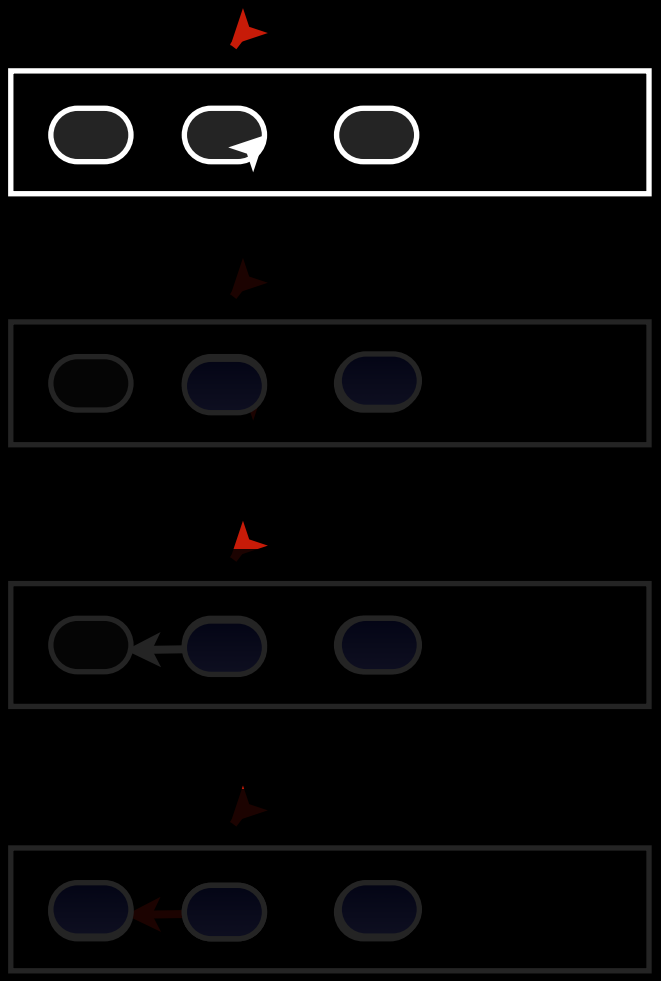
Work-based scheduling



Baker

- First attempt at RTGC by Baker '78
- For LISP where all objects are the same, small, size
- Copying collector that instruments the mutator to maintain:
  - To-space invariant:
    - mutator uses to-space, never sees from-space, objects are copied on first read
  - Steady state:
    - collector keeps up with mutator allocations, collection work performed on each allocation

Incrementalizing marking



Collector marks object

Application updates reference field

Compiler inserted write barrier marks object

Dissecting Baker

- The chief problem with Baker is the to-space invariant
  - Reads are frequent, a heavy read barrier leads to poor performance
  - Variable sized objects in Java make costs harder to bound
- What RT programmers fear:
  - a burst of copying reads, causing substantial slow-downs

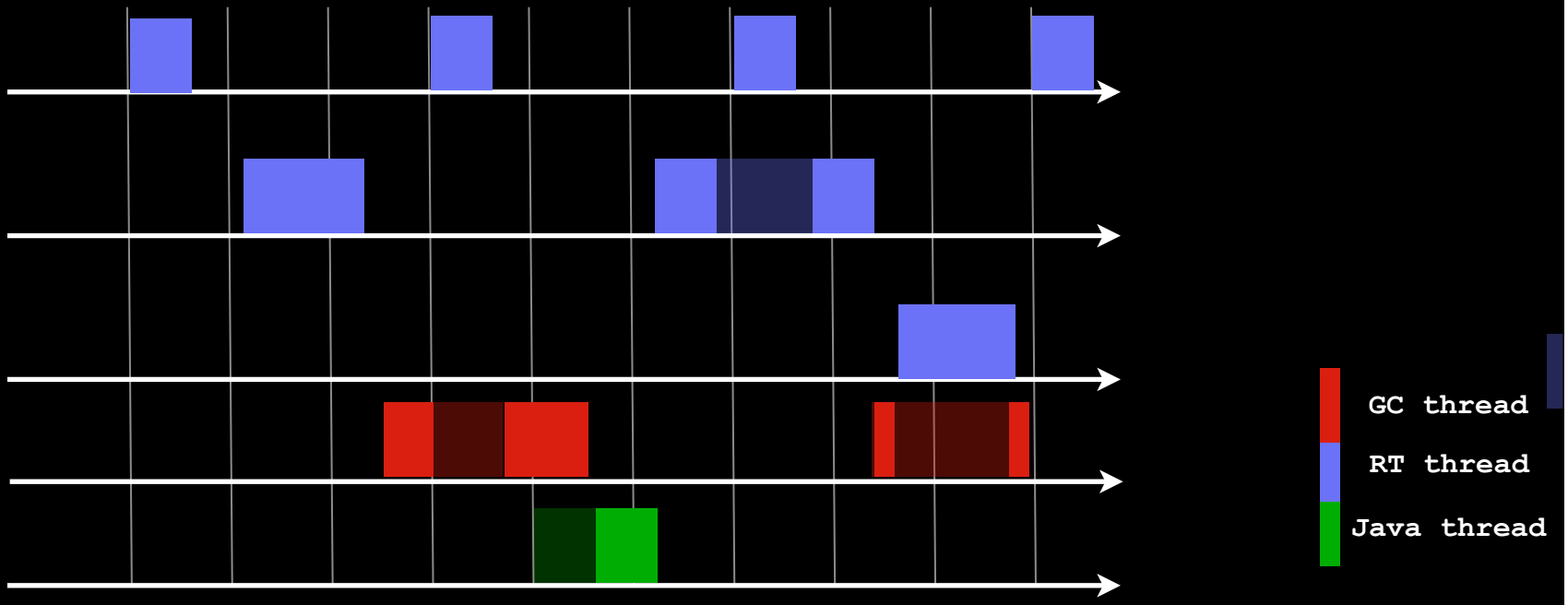
Siebert

- Modern work-based collector in the JamaicaVM:
  - Mark-Sweep, no copying
  - All objects are same size, 32 bytes
    (large objects⇒ lists, arrays⇒ tries) Stack is logically an object
  - Root scanning is fully incremental
  - Write barrier to mark objects (including the stack)
  - Allocation triggers a bounded amount of collector work

Slack-based RTGC

Henriksson'98

Slack-based GC Scheduling

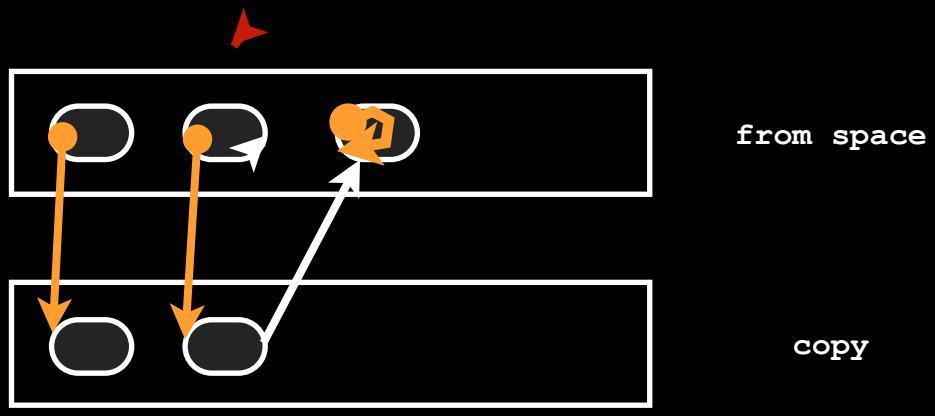


Slack-based GC

- Henriksson takes a different approach to real-time guarantees:
  - tries to guarantee real-time tasks never experience GC interference
- How is this done?
  - RT tasks can always preempt GC
  - Objects have normal representation.
  - Read & write barriers are used to ensure marking/copying soundness.
  - Copying collector, with roll backs when the mutator preempts GC

Incrementalizing copying/compaction

- Forwarding pointers are used to refer to the current version of the object.
- Every access must start with a derefence



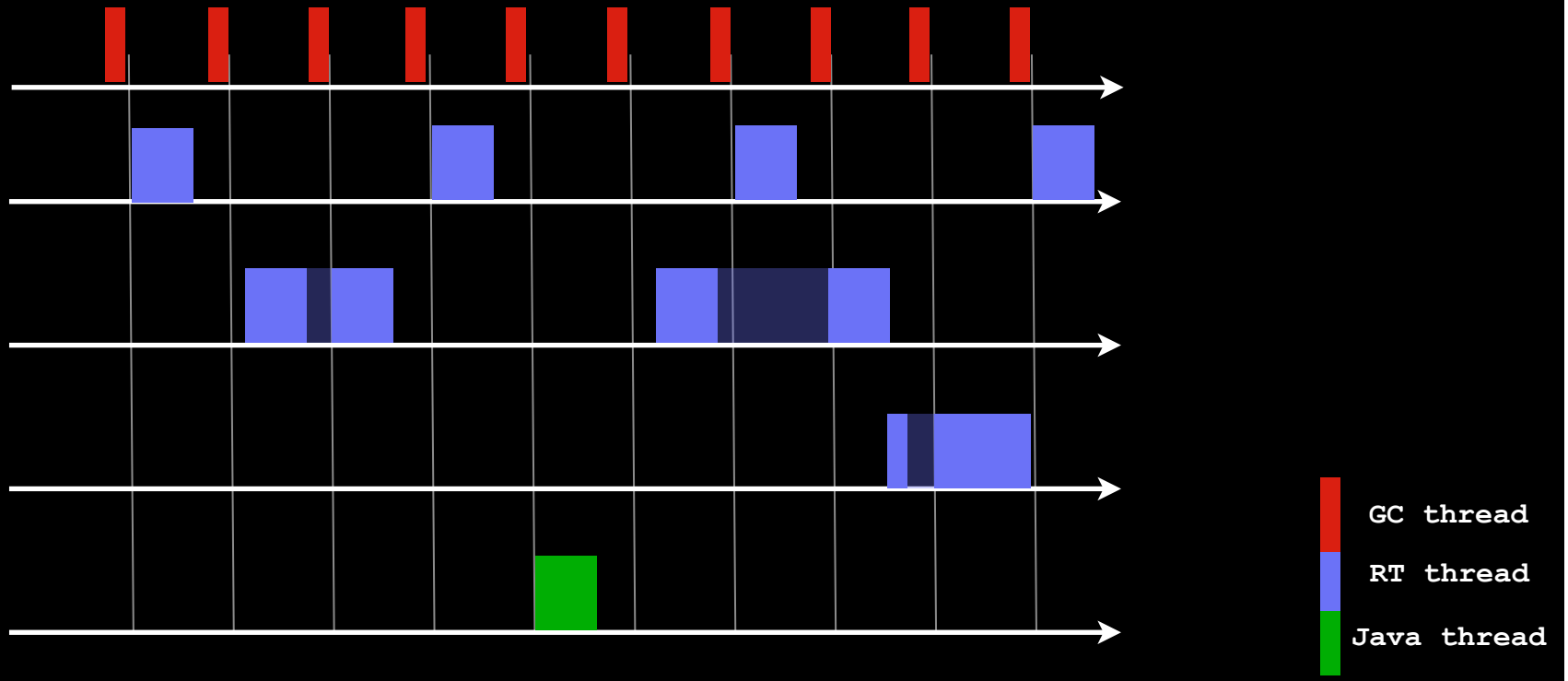
Dissecting Henriksson

- Pros:
  - Gives the programmer full control over the GC schedule
  - SUN uses a variant of this collector in their RTSJVM
- Cons:
  - Read barrier are expensive
  - Mutator induced copy aborts make collector progress hard to ensure
  - Programmer must know allocation rate, and to budget for GC work
  - Priority inversion can lead to GC interference

Time-based RTGC

Metronome'03

Time-based GC Scheduling



Time-based GCs

- Metronome runs periodically for a bounded amount of time
  - Amount of collector interference is easy to understand
  - Progress is easier to guarantee than in Henriksson
- A Mark-Sweep-and-sometimes-Copy collector:
  - Write barrier to ensure marking soundness
  - Light read barrier to ensure copying soundness
  - Almost-empty pages evacuated in response to fragmentation
  - Arrays are split into tries with page-sized leaves
  - Collector increments lower-bounded by time to copy a page-size objects

Suming up

Siebert

Local ref. assignment	mark(b) a = b
Primitive heap load	a = b->f
Reference heap load	a = b->f mark(a)
Primitive heap store	a->f = b
Reference heap store	mark(b) a->f = b

Henriksson

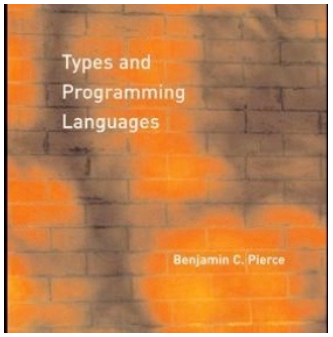
Local ref. assignment	a = b
Primitive heap load	a = b->forward->f
Reference heap load	a = b->forward->f mark(a)
Primitive heap store	a->forward->f = b
Reference heap store	mark(b) a->forward->f = b->forward

Metronome

Local ref. assignment	a = b
Primitive heap load	a = b->forward->f
Reference heap load	a = b->forward->f
Primitive heap store	a->forward->f = b
Reference heap store	mark(a->forward->f) a ->forward->f = b->forward

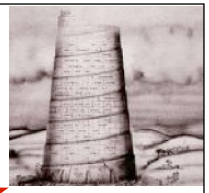
The Essence of JavaScript

by Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi



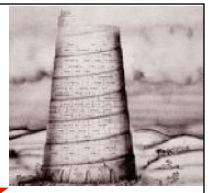
http://www.cs.brown.edu/people/arjun/lambdajs

A small step operational semantics



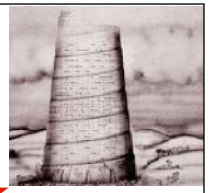
- λJS is a small-step, operational semantics for the core of JavaScript
- The full language is obtained by desugaring into λJS
- An implementation is available (in Haskell)
- Treatment of with is particularly interesting…
- Some security applications which we will not discuss here.

Functions and Objects



$$
\begin{array}{rl}
c = & \texttt{num} \mid \texttt{str} \mid \texttt{bool} \mid \texttt{undefined} \mid \texttt{null} \\
v = & c \mid \texttt{func(}x \cdots\texttt{) \{ return } e \texttt{ \}} \mid \texttt{\{ } str\texttt{:}v \cdots \texttt{ \}} \\
e = & x \mid v \mid \texttt{let (}x \texttt{ = } e \cdots\texttt{) } e \mid e\texttt{(}e \cdots\texttt{)} \mid e\texttt{[}e\texttt{]} \mid e\texttt{[}e\texttt{] = } e \mid \texttt{delete } e\texttt{[}e\texttt{]} \\
E = & \bullet \mid \texttt{let (}x \texttt{ = } v \cdots\ x \texttt{ = } E\texttt{, } x \texttt{ = } e \cdots\texttt{) } e \mid E\texttt{(}e \cdots\texttt{)} \mid v\texttt{(}v \cdots\ E\texttt{, } e \cdots\texttt{)} \\
& \mid \texttt{\{}str\texttt{: } v \cdots\ str\texttt{:}E\texttt{, } str\texttt{:}e \cdots\ \texttt{\}} \mid E\texttt{[}e\texttt{]} \mid v\texttt{[}E\texttt{]} \mid E\texttt{[}e\texttt{] = } e \mid v\texttt{[}E\texttt{] = } e \\
& \mid v\texttt{[}v\texttt{] = } E \mid \texttt{delete } E\texttt{[}e\texttt{]} \mid \texttt{delete } v\texttt{[}E\texttt{]}
\end{array}
$$

Functions and Objects



$$\texttt{let (x = v } \cdots\texttt{) e} \hookrightarrow e[x/v] \cdots \qquad \text{(E-Let)}$$

$$\texttt{(func(}x_1 \cdots x_n\texttt{) \{ return e \})(}v_1 \cdots v_n\texttt{)} \hookrightarrow e[x_1/v_1 \cdots x_n/v_n] \qquad \text{(E-App)}$$

$$\texttt{\{ } \cdots \texttt{ str: v } \cdots \texttt{ \}[str]} \hookrightarrow v \qquad \text{(E-GetField)}$$

$$\frac{str_x \notin (str_1 \cdots str_n)}{\texttt{\{ }str_1\texttt{: }v_1 \cdots str_n\texttt{: }v_n\texttt{ \} [}str_x\texttt{]} \hookrightarrow \texttt{undefined}} \qquad \text{(E-GetField-NotFound)}$$

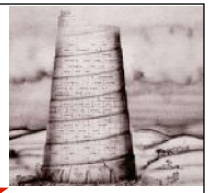
$$\begin{array}{c}\texttt{\{ }str_1\texttt{: }v_1 \cdots str_i\texttt{: }v_i \cdots str_n\texttt{: }v_n\texttt{ \} [}str_i\texttt{] = v} \\ \hookrightarrow \texttt{\{ }str_1\texttt{: }v_1 \cdots str_i\texttt{: }v \cdots str_n\texttt{: }v_n\texttt{ \}}\end{array} \qquad \text{(E-UpdateField)}$$

$$\frac{str_x \notin (str_1 \cdots)}{\texttt{\{ }str_1\texttt{: }v_1 \cdots\texttt{\} [}str_x\texttt{] = }v_x \hookrightarrow \texttt{\{ }str_x\texttt{: }v_x\texttt{, }str_1\texttt{: }v_1 \cdots\texttt{ \}}} \qquad \text{(E-CreateField)}$$

$$\begin{array}{c}\texttt{delete \{ }str_1\texttt{: }v_1 \cdots str_i\texttt{: }v_x \cdots str_x\texttt{: }v_n\texttt{ \} [}str_x\texttt{]} \\ \hookrightarrow \texttt{\{ }str_1\texttt{: }v_1 \cdots str_i\texttt{: }v \cdots str_n\texttt{: }v_n\texttt{ \}}\end{array} \qquad \text{(E-DeleteField)}$$

$$\frac{str_x \notin (str_1 \cdots)}{\texttt{delete \{ }str_1\texttt{: }v_1 \cdots\texttt{\} [}str_x\texttt{]} \hookrightarrow \texttt{\{ }str_1\texttt{: }v_1 \cdots\texttt{\}}} \qquad \text{(E-DeleteField-NotFound)}$$

Functions and Objects



- Dot syntax is desugared to string lookup
- ...

```
let (obj = { "x" : 500, "y" : 100 })
  let (select = func(name) { return obj[name] })
    select("x") + select("y")
↪* 600
```

– A program that looks up a non-existent field does not result in an error; instead, JavaScript returns the value `undefined` (E-GetField-NotFound):

```
{ "x" : 7 }["y"] ↪ undefined
```

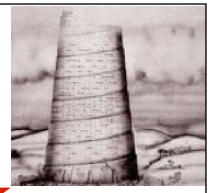
– Field update in JavaScript is conventional (E-UpdateField)—

```
{ "x" : 0 }["x"] = 10 ↪ { "x" : 10 }
```

—but the same syntax also creates new fields (E-CreateField):

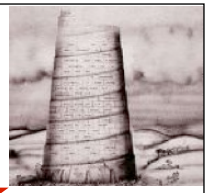
```
{ "x" : 0 }["z"] = 20 ↪ {"z" : 20, "x" : 10 }
```

State



$l = \cdots$	Locations
$v = \cdots \mid l$	Values
$\sigma = \langle (l, v) \cdots \rangle$	Stores
$e = \cdots \mid$ `e = e` $\mid$ `ref e` $\mid$ `deref e`	Expressions
$E = \cdots \mid$ `E = e` $\mid$ `v = E` $\mid$ `ref E` $\mid$ `deref E`	Evaluation Contexts

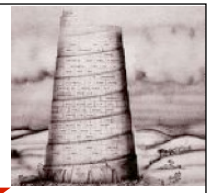
Example



```
function sum(arr) {
  var r = 0;
  for (var i = 0; i < arr["length"]; i = i + 1) {
    r = r + arr[i] };
  return r };

sum([1,2,3]) // returns 6
var a = [1,2,3,4];
delete a["3"];
sum(a) // returns NaN!
```

State



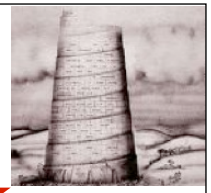
$$\frac{e_1 \hookrightarrow e_2}{\sigma E[e_1] \longrightarrow \sigma E[e_2]}$$

$$\frac{l \notin dom(\sigma) \qquad \sigma' = \sigma, (l, v)}{\sigma E[\texttt{ref v }] \longrightarrow \sigma' E[\texttt{l}]} \quad \text{(E-Ref)}$$

$$\sigma E[\texttt{deref l }] \longrightarrow \sigma E[\sigma(l)] \quad \text{(E-Deref)}$$

$$\sigma E[\texttt{l = v }] \longrightarrow \sigma[l/v] E[l] \quad \text{(E-SetRef)}$$

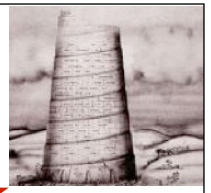
Prototypes



$$\frac{str_x \notin (str_1 \cdots str_n) \qquad \texttt{"\_\_proto\_\_"} \notin (str_1 \cdots str_n)}{\{\ str_1\ :\ v_1\ ,\ \cdots\ ,\ str_n\ :\ v_n\ \}\ [str_x] \hookrightarrow \texttt{undefined}}\ \text{(E-GetField-NotFound)}$$

$$\frac{str_x \notin (str_1 \cdots str_n)}{\{\ str_1\ :\ v_1 \cdots\ \texttt{"\_\_proto\_\_"}\colon\ p\ \cdots\ str_n\ :\ v_n\ \}\ [str_x] \hookrightarrow (\texttt{deref}\ p)[str_x]}\ \text{(E-GetField-Proto)}$$

Desugaring

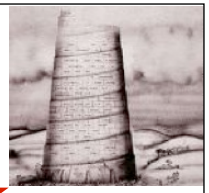


$desugar[\![$`{prop : e ···}`$]\!]$ = `ref {`
`  prop :` $desugar[\![e]\!]\cdots$`,`
`  "__proto__": (deref Object)["prototype"]`
`}`

$desugar[\![$`function(`$x\cdots$`) { `$stmt\cdots$` }`$]\!] =$ `ref {`
`  "code": func(`$this, x\cdots$`) { return `$desugar[\![stmt\cdots]\!]$` },`
`  "prototype": ref { "__proto__":`
`                     (deref Object)["prototype"]`
`                   } }`

$desugar[\![$`new `$e_f(e\cdots)$`]\!] =`
`  let (constr = deref `$desugar[\![e_f]\!]$
`       obj = ref { "__proto__" : constr["prototype"] })`
`    constr["code"](obj, `$desugar[\![e]\!]\cdots$`);`
`    obj`

Desugaring



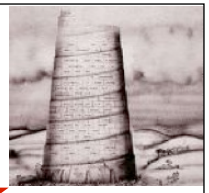
$desugar[\![$`obj[field](`$e \cdots$`)`$]\!]$ =

```
  let (obj = desugar[[obj]])
    let (f = (deref obj)[field])
      f["code"](obj, desugar[[e]] ···)
```

$desugar[\![e_f$`(`$e \cdots$`)`$]\!]$ =

```
  let (obj = desugar[[e_f]])
    let (func = deref obj)
      func["code"](window, desugar[[e]] ···)
```

Desugaring

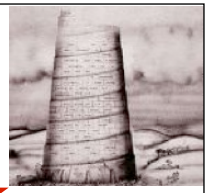


desugar⟦`obj instanceof constr`⟧ =

```
  let (obj = ref (deref desugar⟦obj⟧),
       constr = deref desugar⟦constr⟧)
    done: {
      while (deref obj !== null) {
        if ((deref obj).__proto__ === constr.prototype) {
          break done true }
        else { obj = (deref obj).__proto__ } };
      false }
```

desugar⟦`this`⟧ = `this` (an ordinary identifier, bound by functions)

Local variables



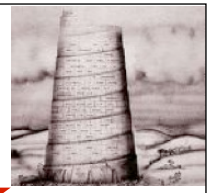
- Local variables are automatically lifted to the top of the function:

```
function foo(){ if (true) { var x = 10; } return x }
foo() // returns 10
```

the `return` statement has access to the variable that appeared to be defined inside a branch of the `if`. This can result in unintuitive answers:

```
function bar() { x = 10; var x = x * x; return x }
bar() // returns 100
x // error: unbound identifier
```

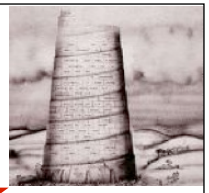
Local variables



- All local variable declarations `var x = e` are desugared into simple assignments, `x = e`.
- Furthermore, locals are desugared to let-bindings at the top of functions:

```
let (x = ref undefined) ···
```

Global Variables



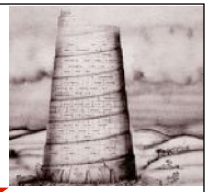
- Global variables are properties of the root scope object, called `window`. `window` has a field `window.window` that references itself:

```
window.window === window // returns true
```

A program can obtain a reference to the global scope object by simply writing `window`. As a consequence, globals seem to break lexical scope, since we can observe that they are properties of `window`:

```
var x = 0;
window.x = 50;
x // returns 50
x = 100;
window.x // returns 100
```

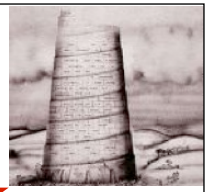
Global Variables



However, `window` is the only scope object that is directly accessible to JavaScript programs [6, Section 10.1.6]. We can maintain lexical scope by abandoning global variables. That is, we simply desugar the code above to the following:

```
window.x = 0;
window.x = 50;
window.x // returns 50
window.x = 100;
window.x // returns 100
```

with



- The `with` statement adds an arbitrary object to the front of the scope chain:

```
function(x, obj) {
 with(obj) {
    x = 50; //if obj.x exists then obj.x=50 else x=50
    return y } }//return either obj.y, or window.y
```

We desugar `with` by turning comments above into code:

```
function(x, obj) {
if(obj.hasOwnProperty("x")) {obj.x=50} else {x=50}
if (obj.hasOwnProperty("y")) { ...
```