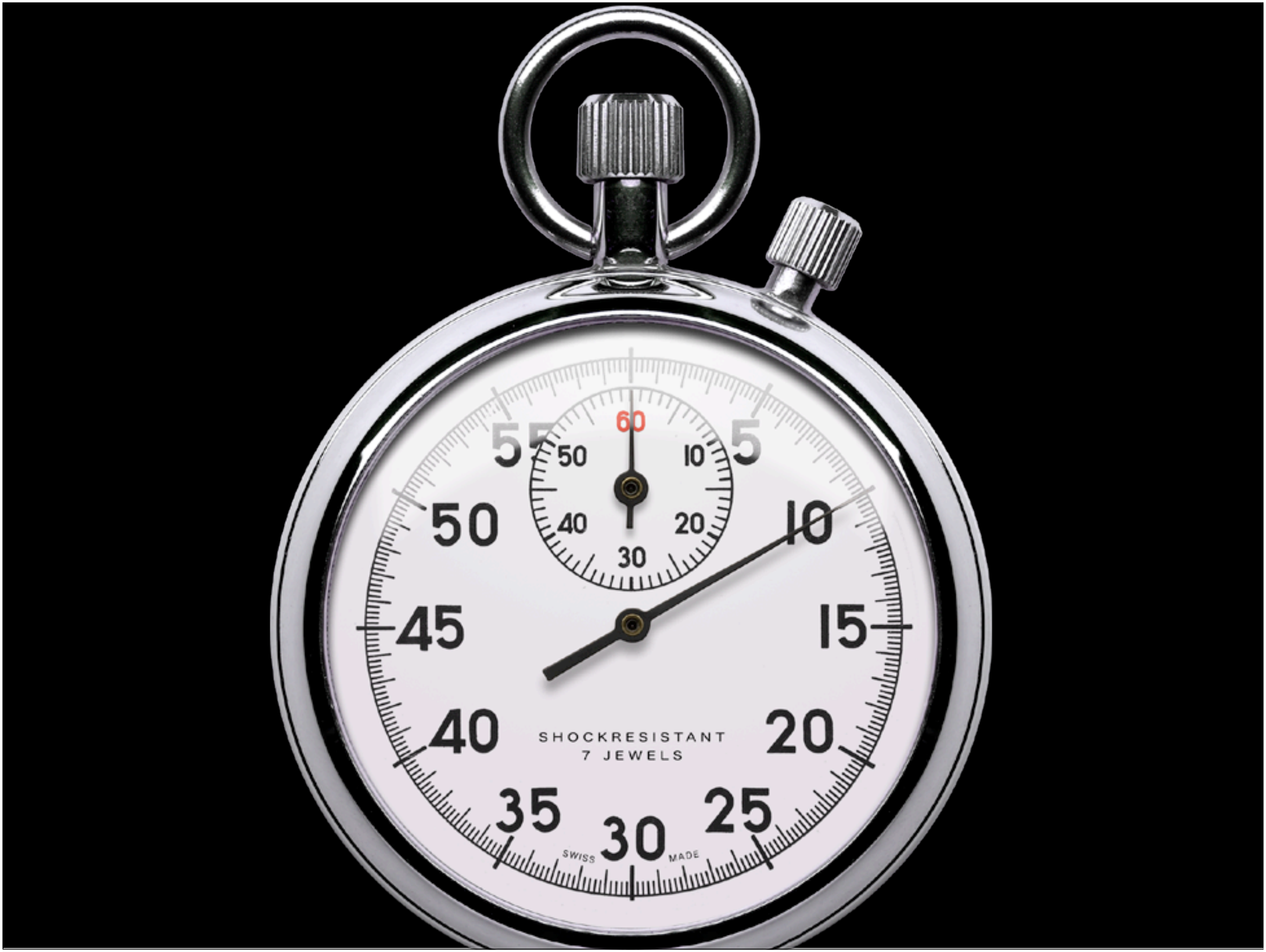

SHOCKRESISTANT
7 JEWELS
SWISS
MADE

Memory management and programming models

- The choice of memory management affects productivity
- Object-oriented languages naturally hide allocation behind abstraction barriers
  - Taking care of de-allocation manually is more difficult in OO style
- Concurrent algorithms usually emphasize allocation
  - because freshly allocated data is guaranteed to be thread local
  - "transactional" algorithms generate a lot of temporary objects
- … but garbage collection is a global, costly, operation that introduces unpredictability

Alternative 1: No Allocation

- If there is no allocation, GC does not run.
    - This approach is used in JavaCard

Alt 2: Allocation in Scoped Memory

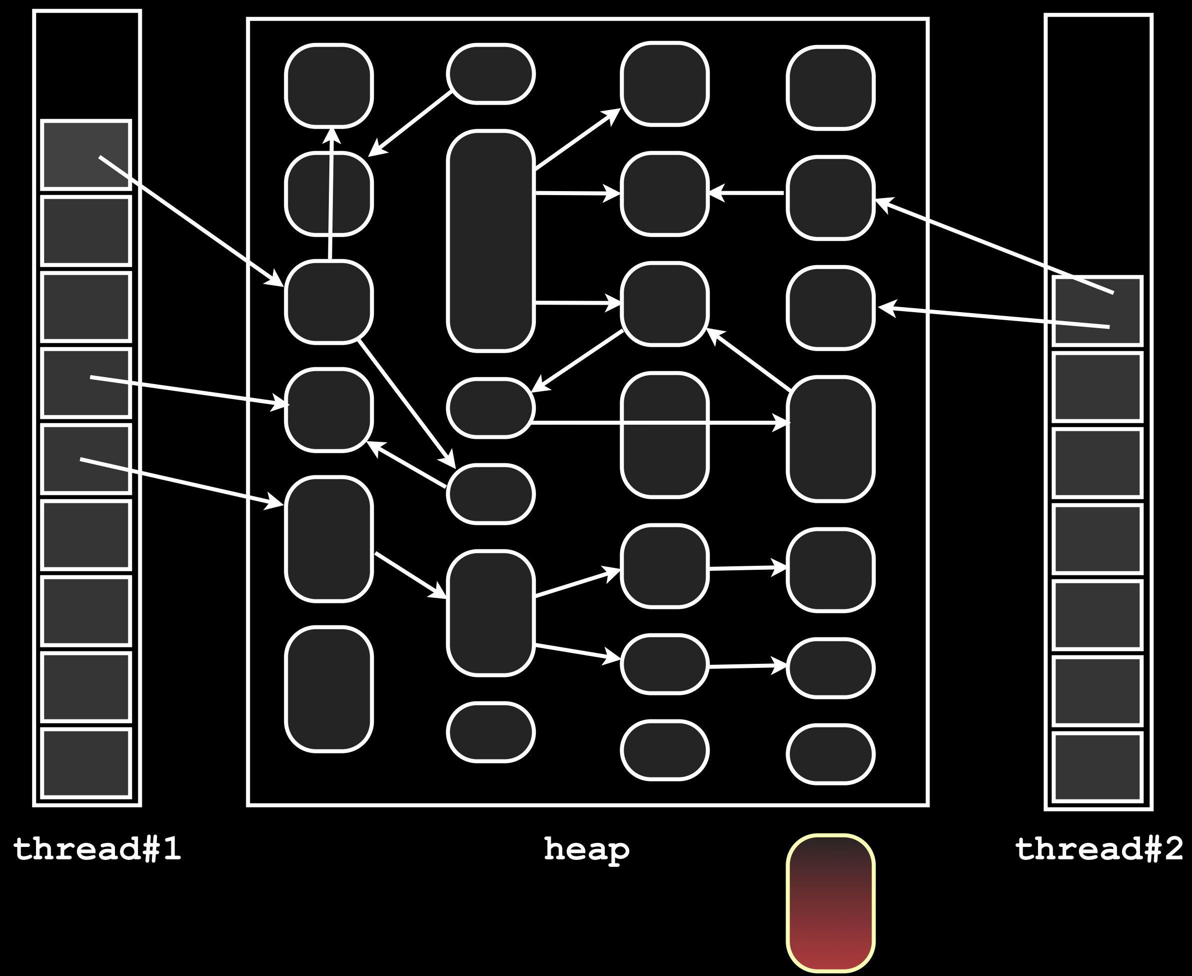
- RTSJ provides scratch pad memory regions which can be used for temporary allocation
  - Used in deployed systems, but tricky as they can cause exceptions

```
s = new SizeEstimator();
s.reserve(Decrypt.class, 2);
...
shared = new LTMemory(s.getEstimate());
shared.enter(new Run(){ public void run(){
    ...d1 = new Decrypt() ...
}});
```

Alt 3: Real-time Garbage Collection

- There are three main families of RTGC implementations
- **Work-based**
  - *Aicas JamaicaVM*
- **Time-triggered, periodic**
  - *IBM Websphere*
- **Time-triggered, slack**
  - *SUN Java Real Time System*

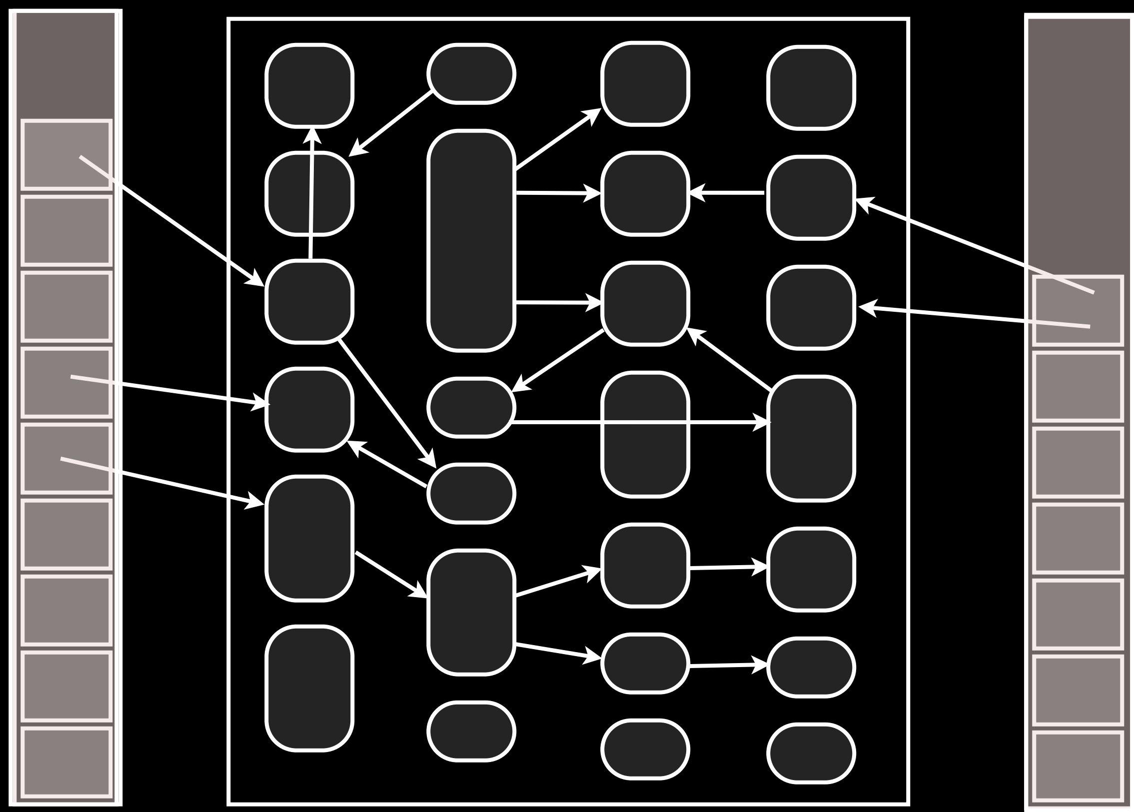
Garbage Collection



Phases

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

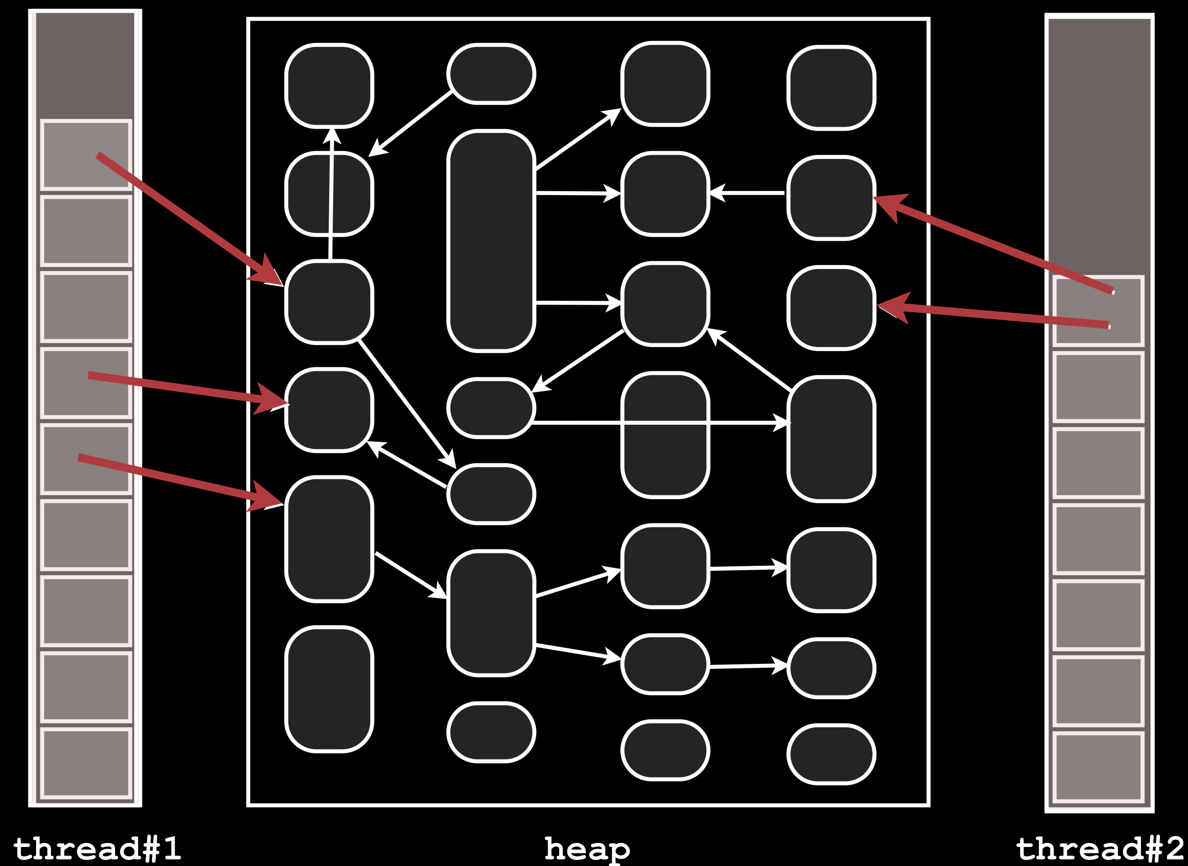
Garbage Collection



Phases

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

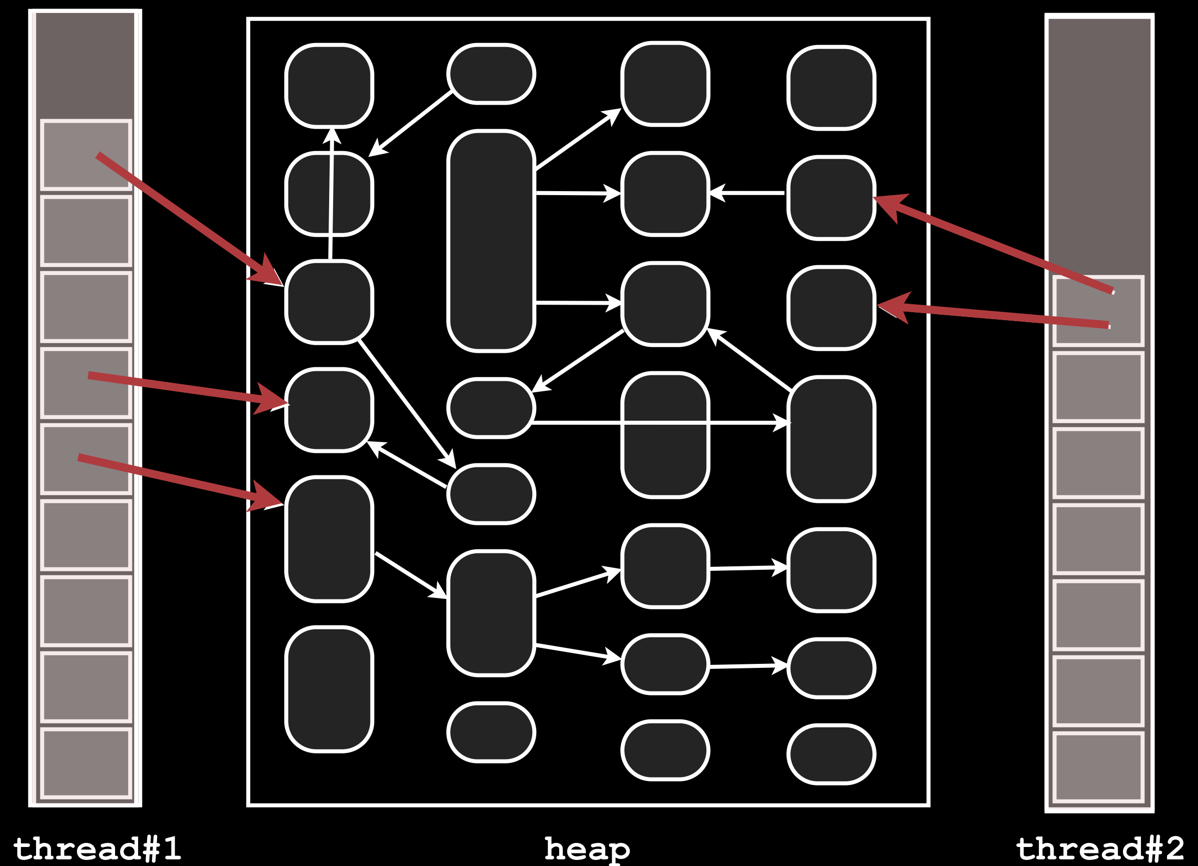
Garbage Collection



Phases

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

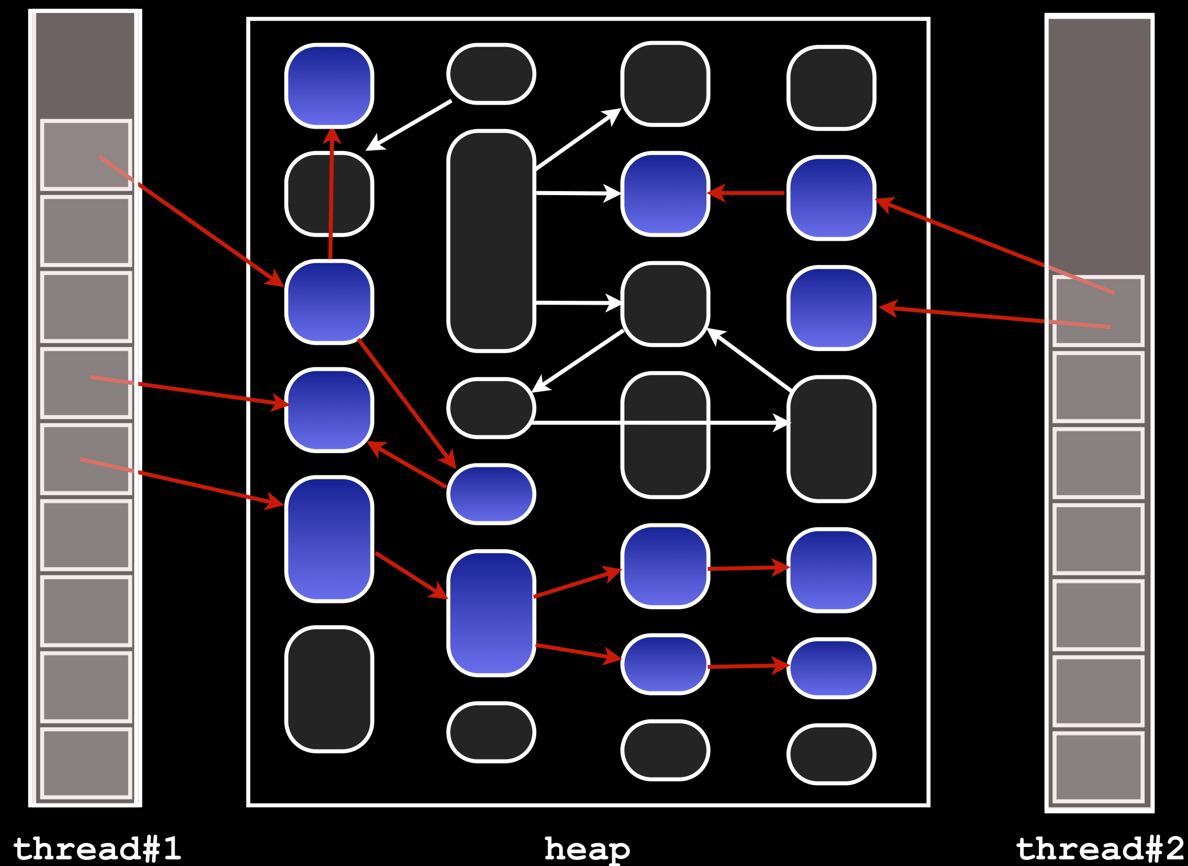
Garbage Collection



Phases

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

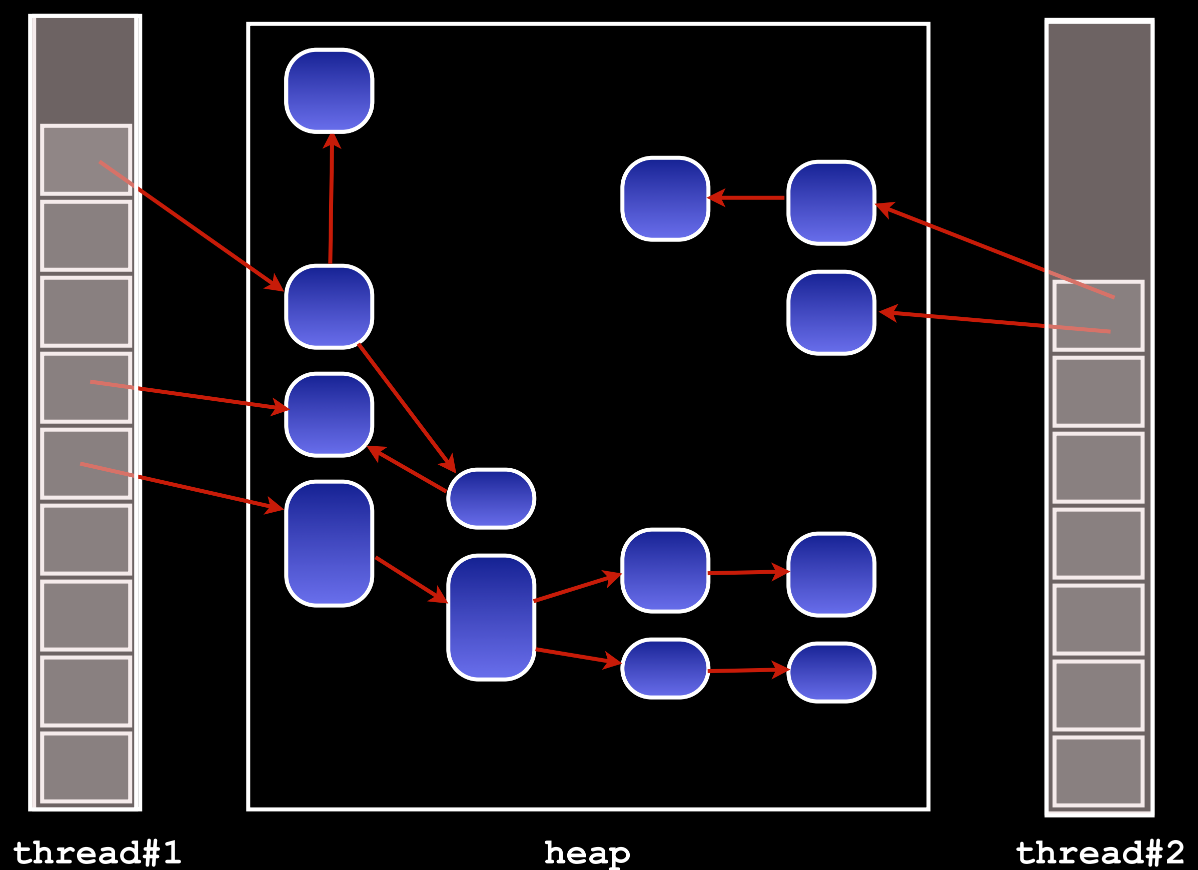
Garbage Collection



Phases

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

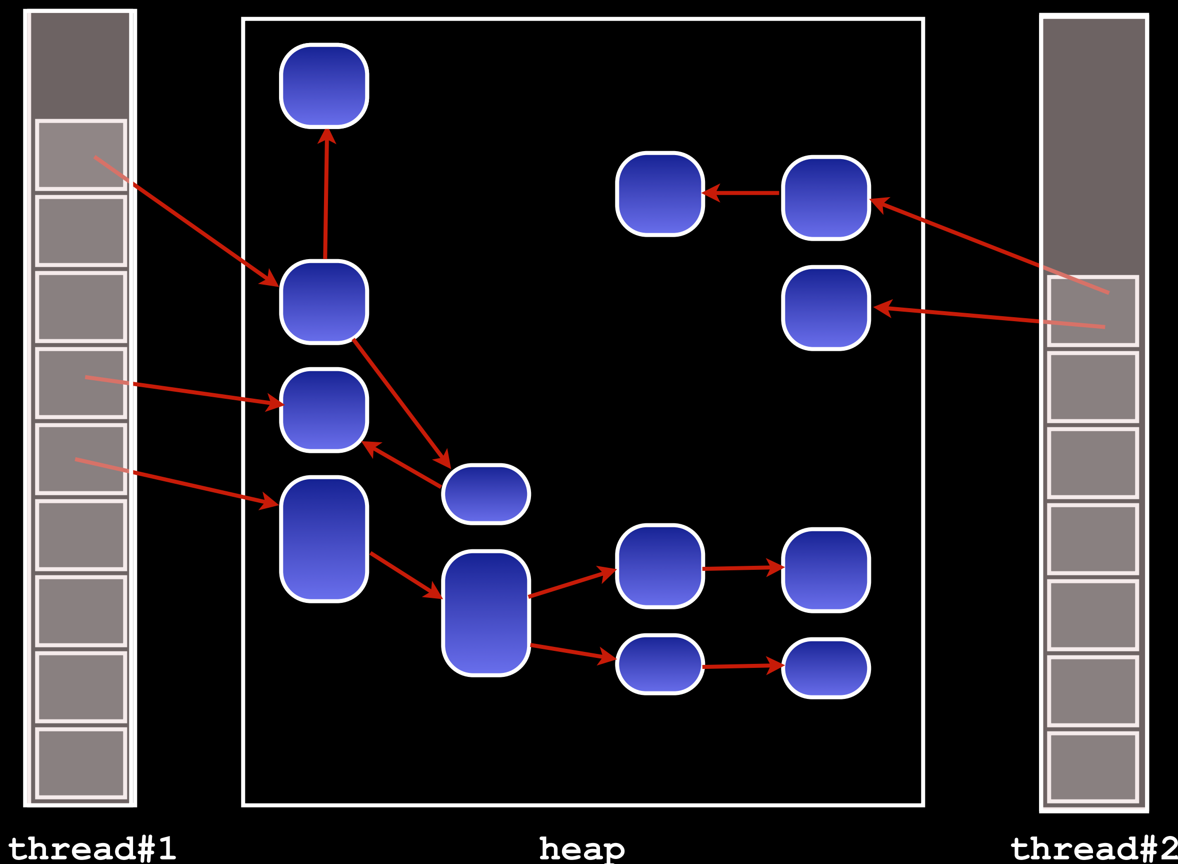
Garbage Collection



Phases

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

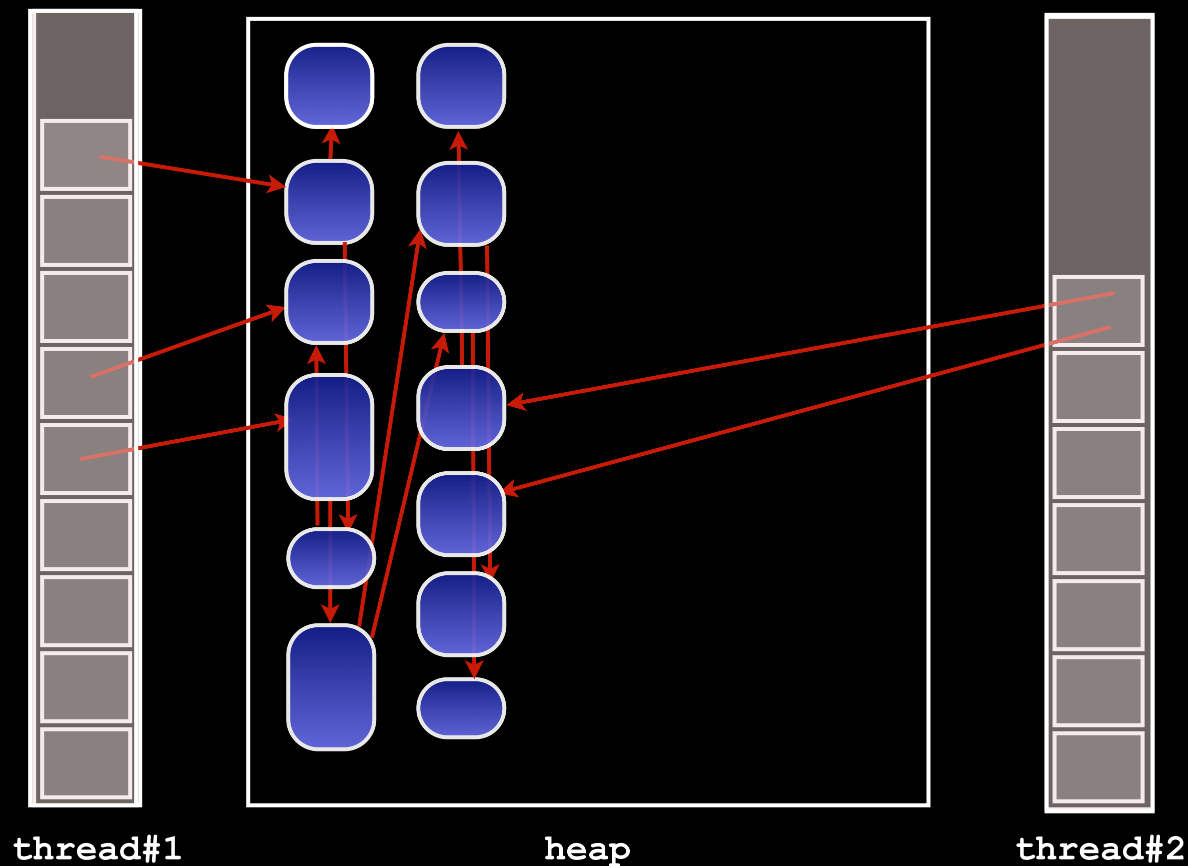
Garbage Collection



Phases

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

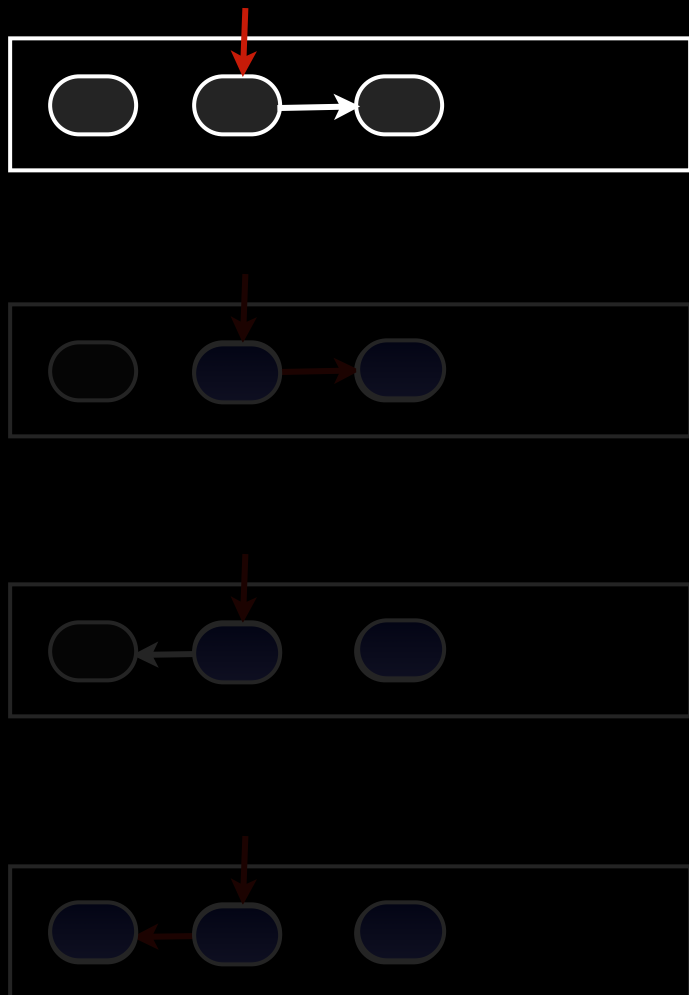
Garbage Collection



Phases

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

Incrementalizing marking



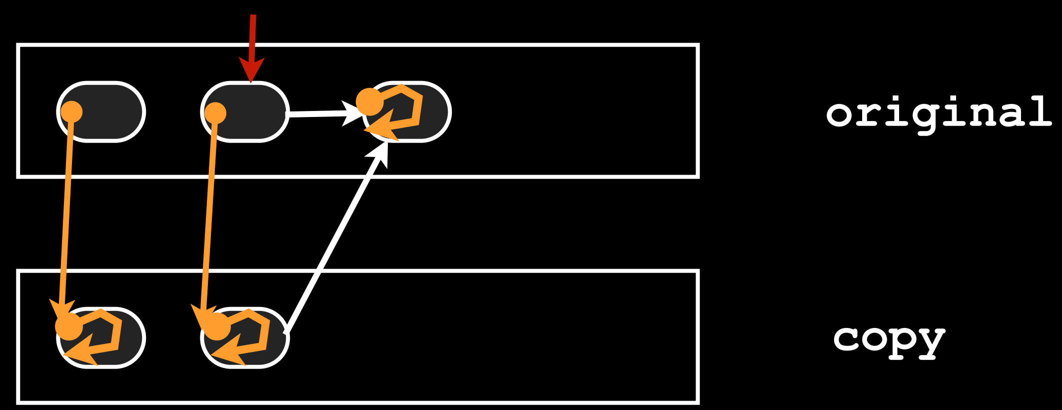
Collector marks object

Application updates
reference field

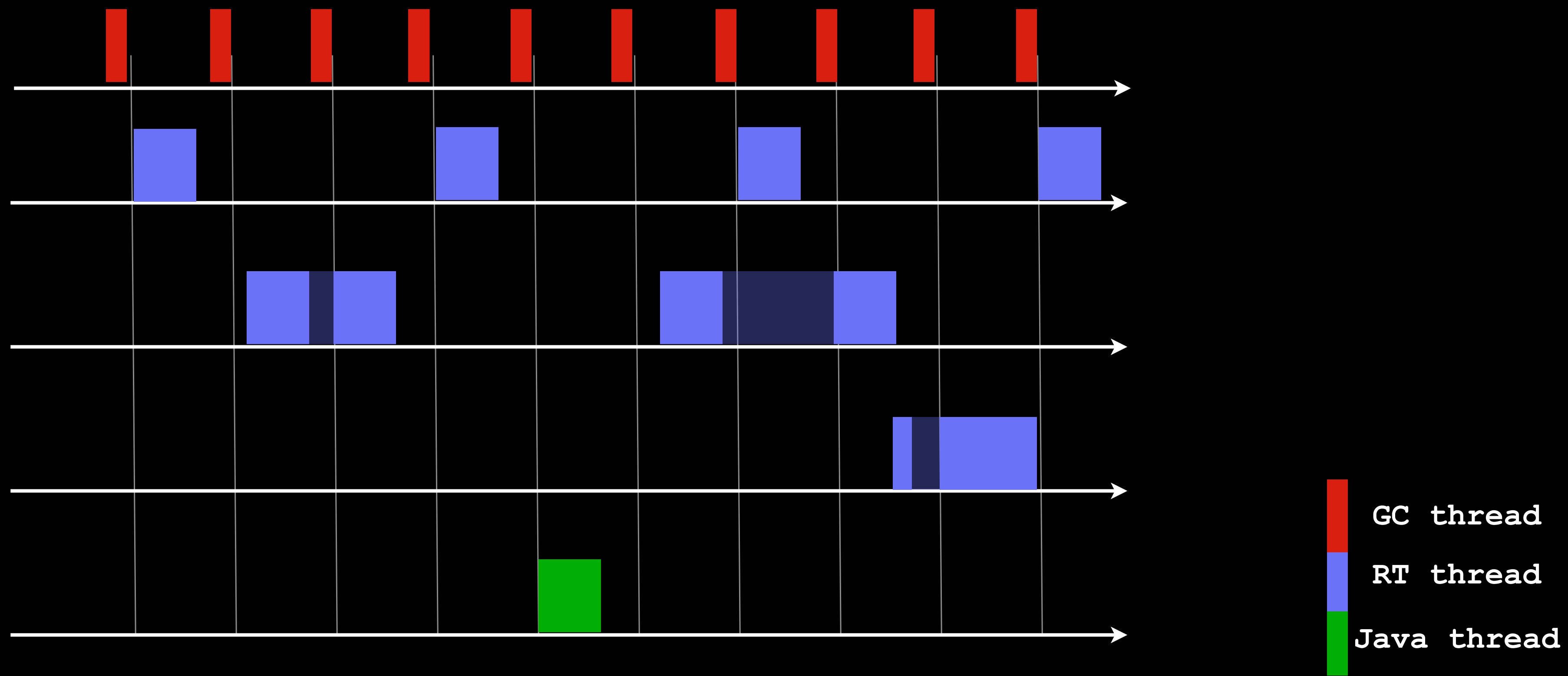
Compiler inserted
write barrier marks object

Incrementalizing compaction

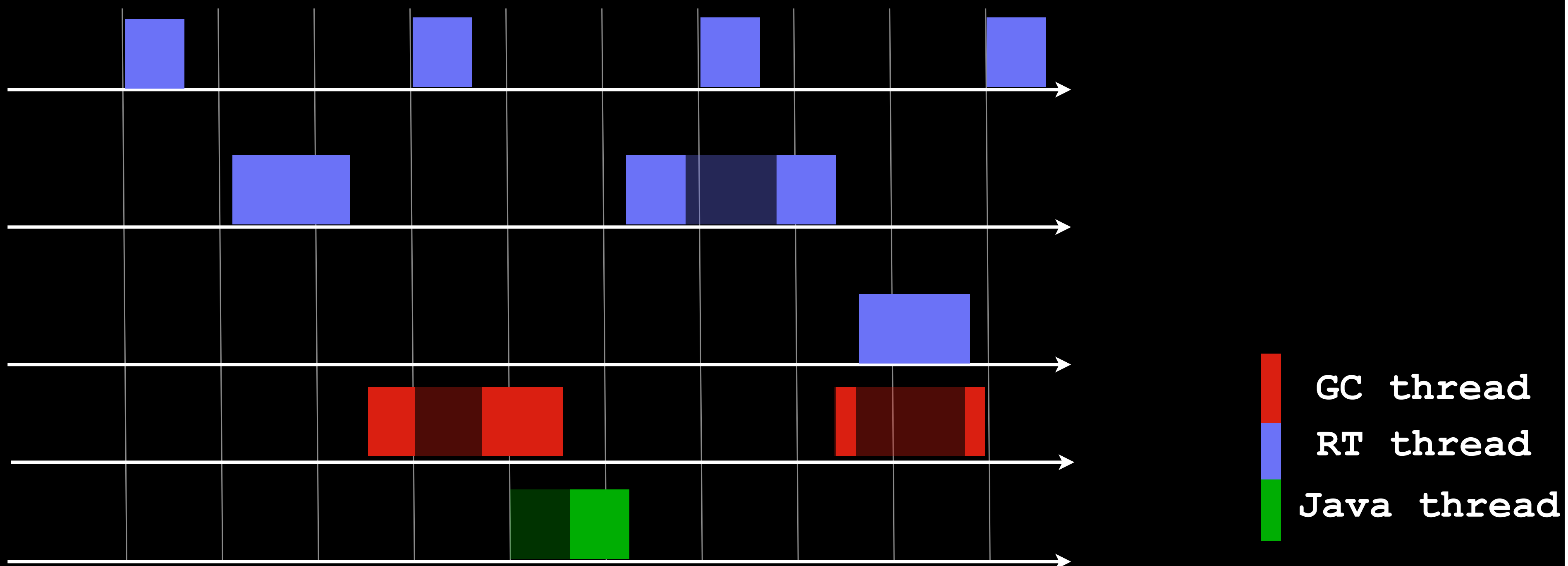
- Forwarding pointers refer to the current version of objects
- Every access must start with a derefence

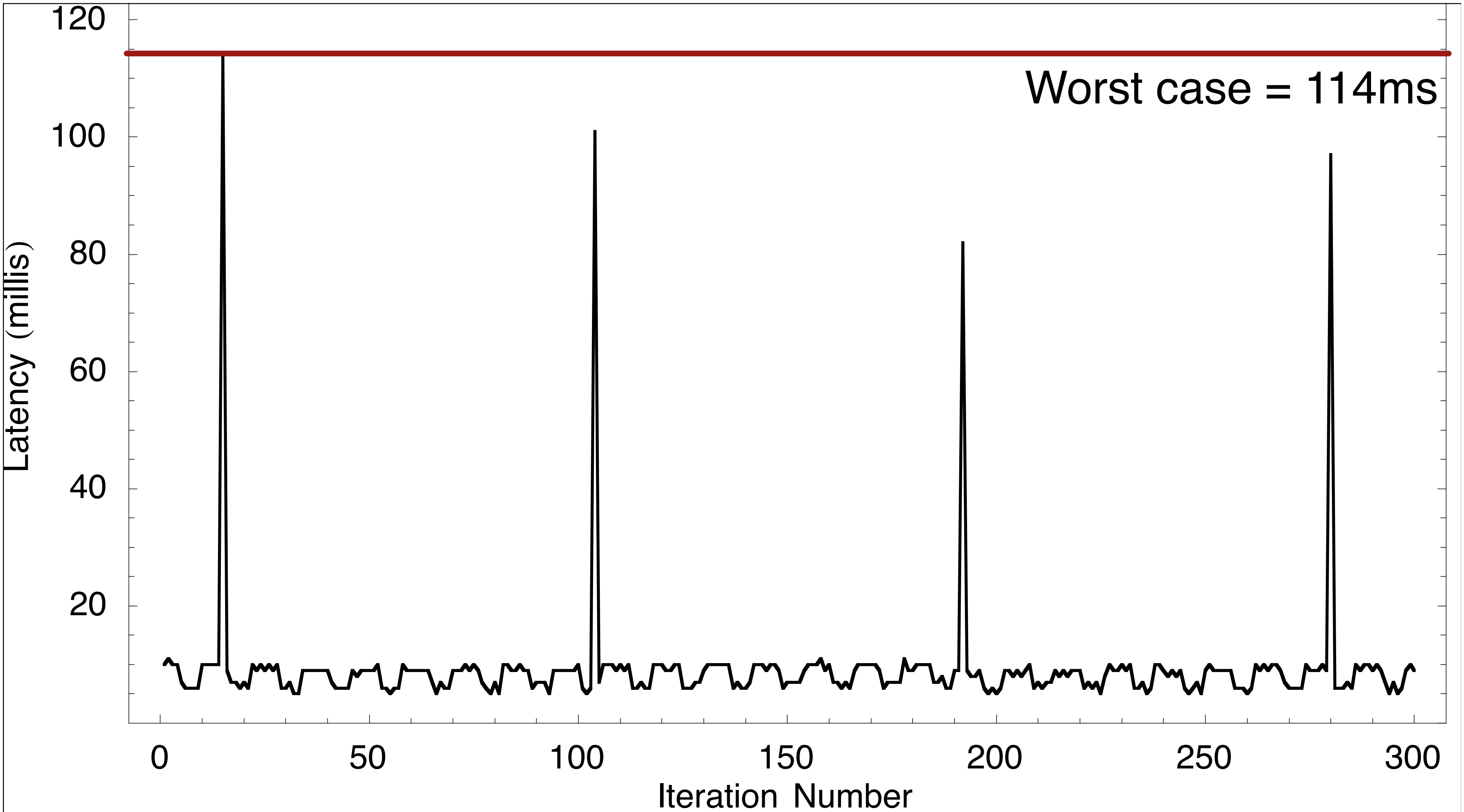


Time-based GC Scheduling

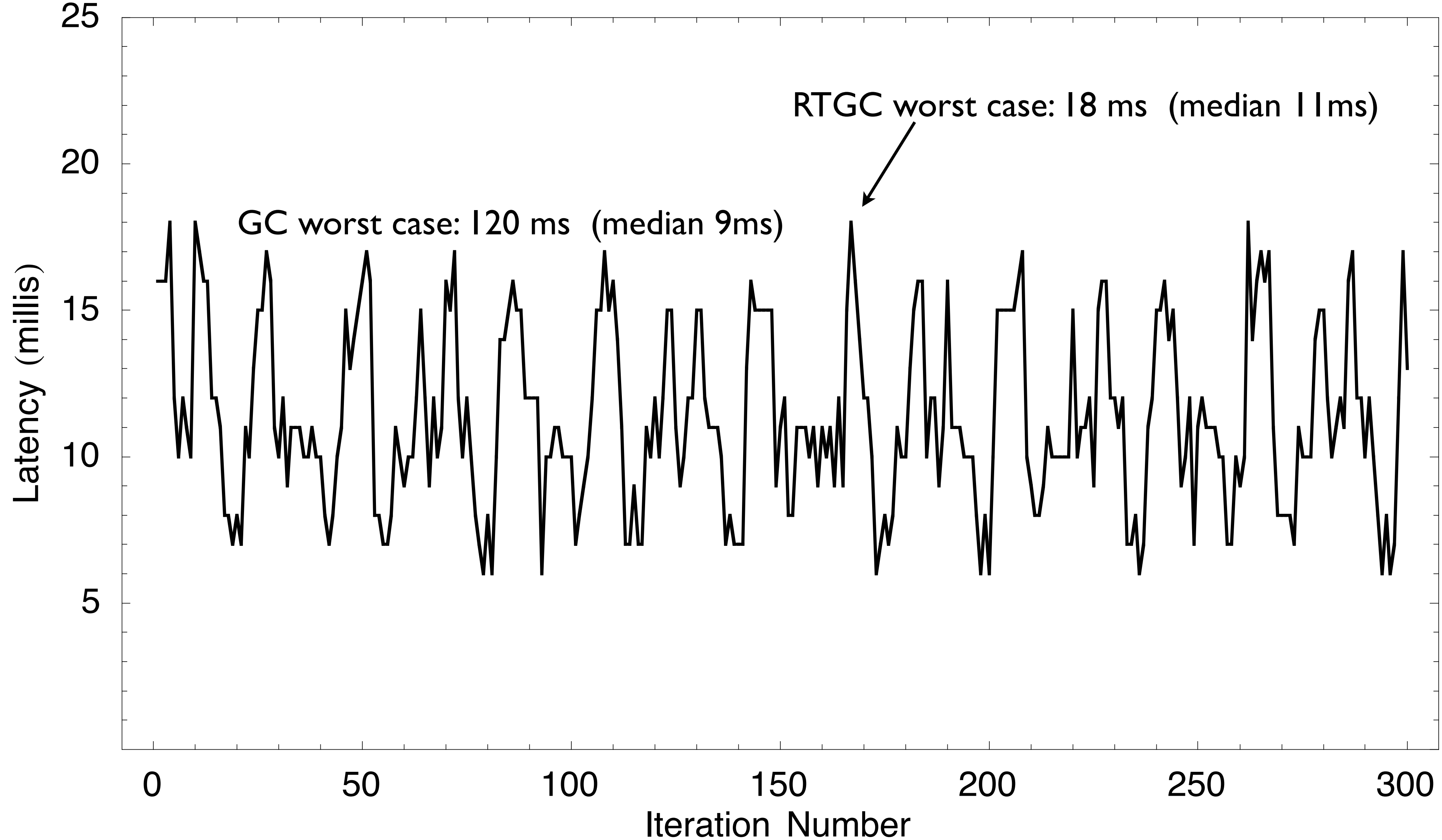


Slack-based GC Scheduling

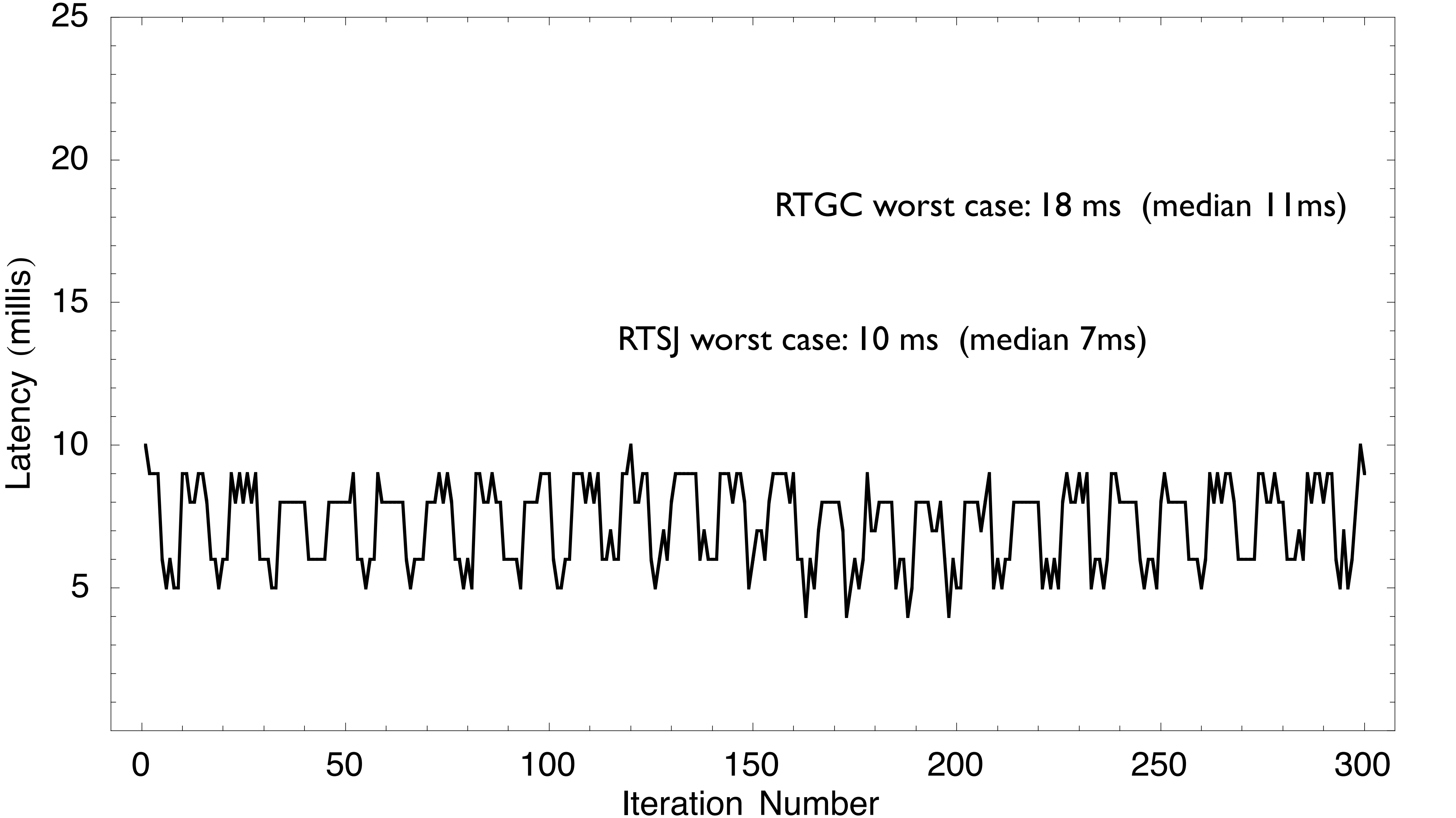




- GC pauses cause the collision detector to miss deadlines...
  and this is not a particularly hard problem should support KHz periods



CD with periodic RTGC



Slack-based GC

Scheduling GC

- Basic parameters needed to schedule a set of task that rely on a real-time garbage collection

C_i	[seconds]	computation time	task
T_i	[seconds]	period	task
A_i	[bytes]	allocation	task
G_i	[seconds]	GC work generated	task
H	[bytes]	heap size	system
L_{max}	[bytes]	live memory	system
T_{gc}	[seconds]	GC cycle duration (period)	system
G_0	[seconds]	GC cycle overhead	system

Traditional response time analysis

- C and T are the traditional task WCET and deadline
- Computing the response time analysis can be done as usual by:

$$R_i = C_i + \sum_{j=1}^{i-1} \left(\left\lceil \frac{R_i}{T_j} \right\rceil C_j \right)$$

Schedulability analysis with GC

- The period of the GC task T_{gc} must be chosen so that it larger than a GC cycle (from start collection to finish)
- The constant G_i is the upper bound on the amount of GC work that a user task *i* can generate in release
- The constant G_0 is the task independent work performed during each GC cycle
- The maximum heap size *L* is a chosen by the user
- The maximum amount of live memory L_{max} depend on the program
- The upper bound on allocation A_i per release of task *i*

Schedulability tests

- In the presence of GC two new tests are added to the schedulability equations:

- *T1 The mutator tasks meet their deadline (modified)*

- *T2 The system does not run out of memory*

- *T3 The GC task meets its deadline and keeps with up mutator tasks*

T2 Memory test

- If an object is freed during one GC cycle, the earliest time it can be found and freed is during the following GC cycle
- Assume that we compute A_{max}, the maximum amount allocated by all mutators during one GC cycle
- The maximum of floating garbage is A_{max}, plus the maximum allocated in the cycle A_{max} and the maximum live memory L_{max} must be less than the heap size H

$$A_{\max} \leq \frac{H - L_{\max}}{2}$$

T2 Memory test

- If we assume that the GC period is a multiple of the hyperperiod of the mutator tasks, then we can compute:

$$
\begin{aligned}
G_{\max} &= G_0 + \sum_{i=1}^{n} \frac{T_{\mathrm{gc}}}{T_i} G_i \\
A_{\max} &= \sum_{i=1}^{n} \frac{T_{\mathrm{gc}}}{T_i} A_i
\end{aligned}
$$

T3 GC test for Slack-scheduled

- For a slack-scheduled RTGC, the response time of the GC is the maximal amount of GC work that has to be performed per cycle plus the sum of the interruptions of all other tasks.
- The response time of the mutator tasks (T1) is computed as usual

$$R_{\mathrm{gc}} = G_{\mathrm{max}} + \sum_{i=1}^{n} \left(\left\lceil \frac{R_{\mathrm{gc}}}{T_i} \right\rceil \cdot C_i \right)$$

T1 for Periodic

- A periodic GC is run according to a pattern, e.g.
  CMMCMM…
- For any window of time *t* the minimum mutator utilization is the least ratio of time available to the mutator threads, written *mmu(t)*
- mcu(t) = 1-*mmu*(*t*) is the ratio used by the collector during that window

$$R_i = C_i + \sum_{j=1}^{i-1} \left(\left\lceil \frac{R_i}{T_j} \right\rceil \cdot C_j \right) + \qquad (1 - mmu(R_i)) \cdot R_i$$

T3 GC test for Periodic

- The response time for the GC is computed by finding:

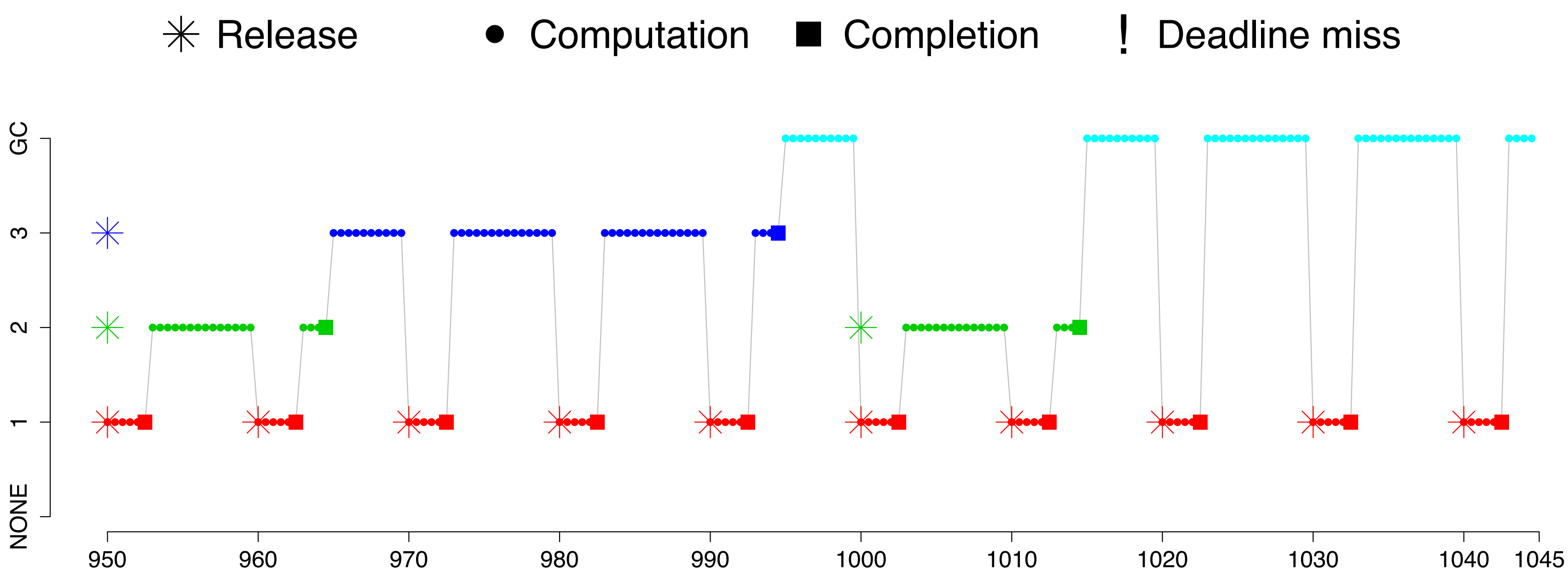
Let R_{gc} be the smallest t such that $t \cdot mcu(t) \geq G_{\max}$ and $t \leq T_{\mathrm{gc}}$

Example 1

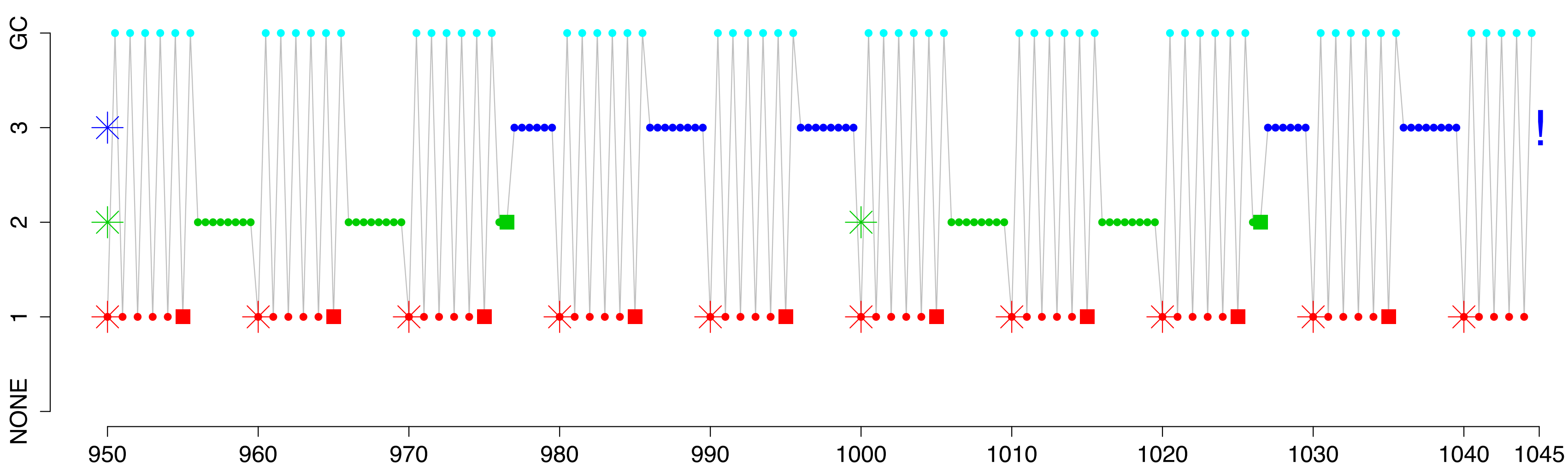
i	T_i	C_i	A_i	G_i
1	10	3	72	1
2	50	9	302	5
3	95	21	256	4

$L_{\max}$	G_0	H	T_{gc}
300	10	25500	730

Quantum size: 0.5
Window size: 10
MC pattern: `MCMCMCMCMCMCMMMMMMMM`



(a) Slack GC does not cause a deadline miss

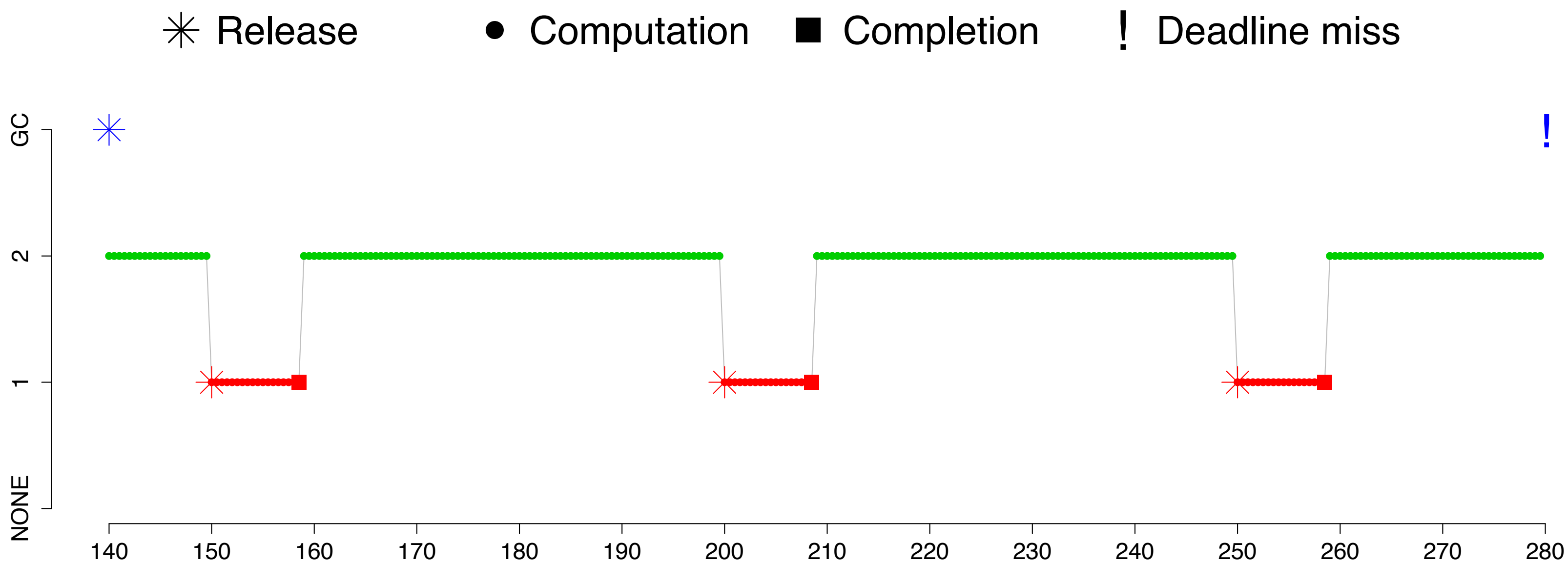


Example 2

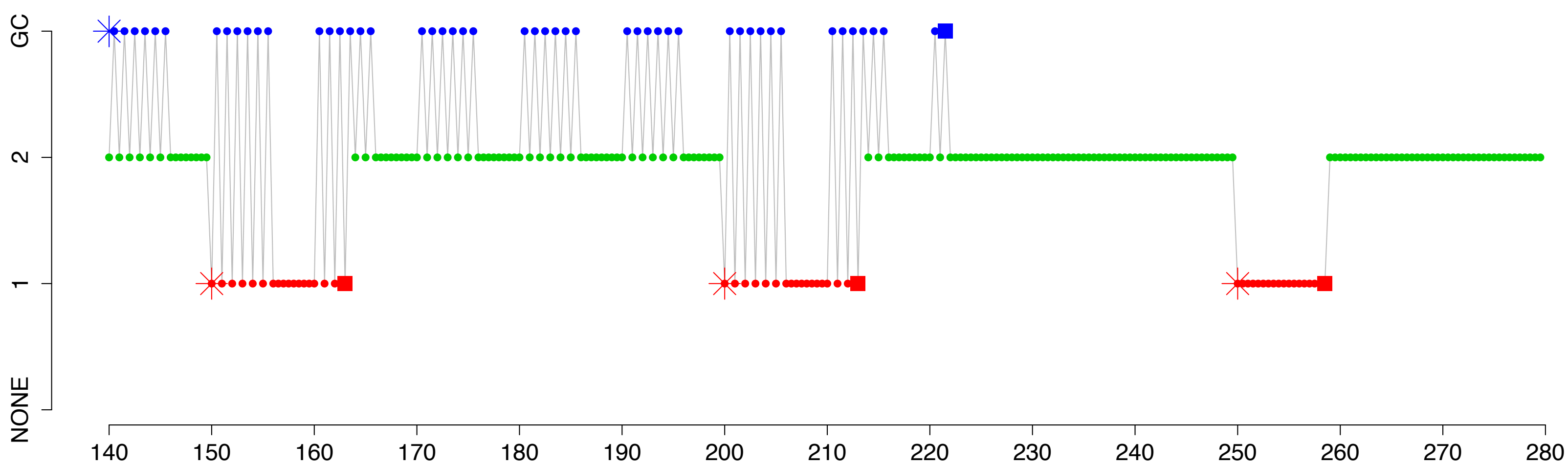
i	T_i	C_i	A_i	G_i
1	50	9	302	5
2	980	490	65	4

L_{max}	G_0	H	T_{gc}
300	10	3000	140

Quantum size: 0.5
Window size: 10
MC pattern: `MCMCMCMCMCMCMMMMMMMM`



(a) Slack GC misses its deadline, cannot keep up with application



(b) Periodic GC does not miss a deadline

References and acknowledgements

- Team
  - *J. Baker, T. Cunei, T. Kalibera, T. Hosking, F. Pizlo, M. Prochazka*

- Paper trail
  - *Scheduling Real-time Garbage Collection on Uni-processors.* TOCS / RTSS, 2009
  - *Accurate Garbage Collection in Uncooperative Environments.* CC:P&E, 2009
  - *Memory Management for Real-time Java: State of the Art.* ISORC, 2008
  - *Garbage Collection for Safety Critical Java.* JTRES, 2007
  - *Hierarchical Real-time Garbage Collection.* LCTES, 2007
  - *Scoped Types and Aspects for Real-time Java Memory management.* RTS, 2007
  - *Accurate Garbage Collection in Uncooperative Environments with Lazy Stacks.* CC, 2007
  - *An Empirical Evaluation of Memory Management Alternatives for Real-time Java.* RTSS, 2006