aicas technology brief

Using Dynamic Memory in Safety-Critical applications

Reliable Software Technologies
Ada-Europe 2013

Dr. James J. Hunt
CEO, aicas
June 2013

Dynamic Memory Management

Current practice

- DO-178B and DO-278 provided no guidance
- Just say no!
- but the following are being used:
    - stack allocation
    - object pooling

Future

- DO-332 provides guidance
- Perhaps, but how?

Is DMM Necessary?

What is the computational complexity?

- State Machine: no
- Push Down Automaton: limited
- Turing Machine: yes

Examples

- Brake control
- Path retracing
- Object recognition

New Standards

EUROCAE	RTCA	EUROCAE Title
ED-12C	DO-178C	Software Considerations in Airborne Systems and Equipment
ED-94C	DO-248C	Supporting Information for ED-12C and ED-109A
ED-109A	DO-278A	Software Integrity Assurance Considerations for Communication, Navigation, Surveillance and AirTrafficManagement (CNS/ATM) Systems
ED-215	DO-330	Software Tool Qualification Considerations
ED-216	DO-331	Model-Based Design and Verification Supplement to ED-12C and ED-109A
ED-217	DO-332	Object-Oriented Technology and Related Techniques Supplement to ED-12C and ED-109A
ED-218	DO-333	Formal Methods Supplement to ED-12C and ED-109A

* RTCA titles are identical except for the references to other standards.

Dynamic Memory Vulnerabilities

1. Ambiguous reference (reallocate live object)
2. Fragmentation starvation
3. Deallocation starvation (memory leak)
4. Premature deallocation (dangling reference)
5. Indeterministic allocation or deallocation
6. Lost update or stale reference
   (due to moving objects)
7. Heap memory exhaustion

Dynamic Memory Safety Objectives

1. Unique Allocation
2. Fragmentation Avoidance
3. Timely Deallocation
4. Reference Consistency
5. Deterministic Execution
6. Atomic Move
7. Sufficient Memory

Memory Management Techniques

Technique	Objectives						
	Unambiguous Reference	Fragment Avoidance	Timely Deallocation	Reference Consistency	Deterministic Deallocation	Atomic Move	Sufficient Memory
Object Pooling	AC	AC	AC	AC	MMI	N/A	AC
Stack Allocation	AC	MMI	MMI	AC	MMI	N/A	AC
Scope Allocation	MMI	MMI	MMI	AC	MMI	N/A	AC
Manual Heap Allocation	AC	?	AC	AC	MMI	MMI	AC
Garbage Collection	MMI	MMI	MMI	MMI	MMI	MMI	AC

AC = application code, MMI = memory management infrastructure,
N/A = not applicable, and ? = difficult to ensure by either AC or MMI.

Deterministic Garbage Collection

Saves development time

- No need to release objects explicitly
- Fewer memory errors
- Can easily track heap usage

Improves safety

- Reduces the danger of memory leaks
- Prevents premature object deallocation
- Does not interfere with realtime response

But what about Realtime?

Qualifying a Garbage Collector

Not possible for all collector

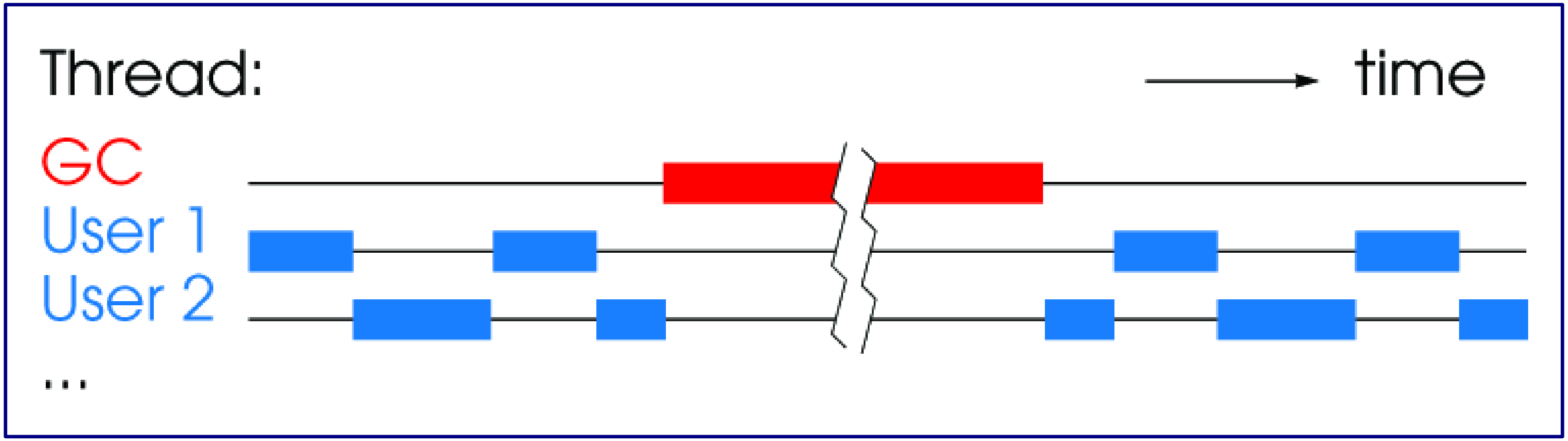
- Must be deterministic; no unbound steps
- Must assume maximum memory use
- Must consider allocation rate

Work Based Collector

- No root scan and compaction (unbound)
- Mark and sweep steps on fixed size blocks
- Automatically tracks allocation rate

Conventional Garbage Collection

GC can interrupt execution for long periods of time:

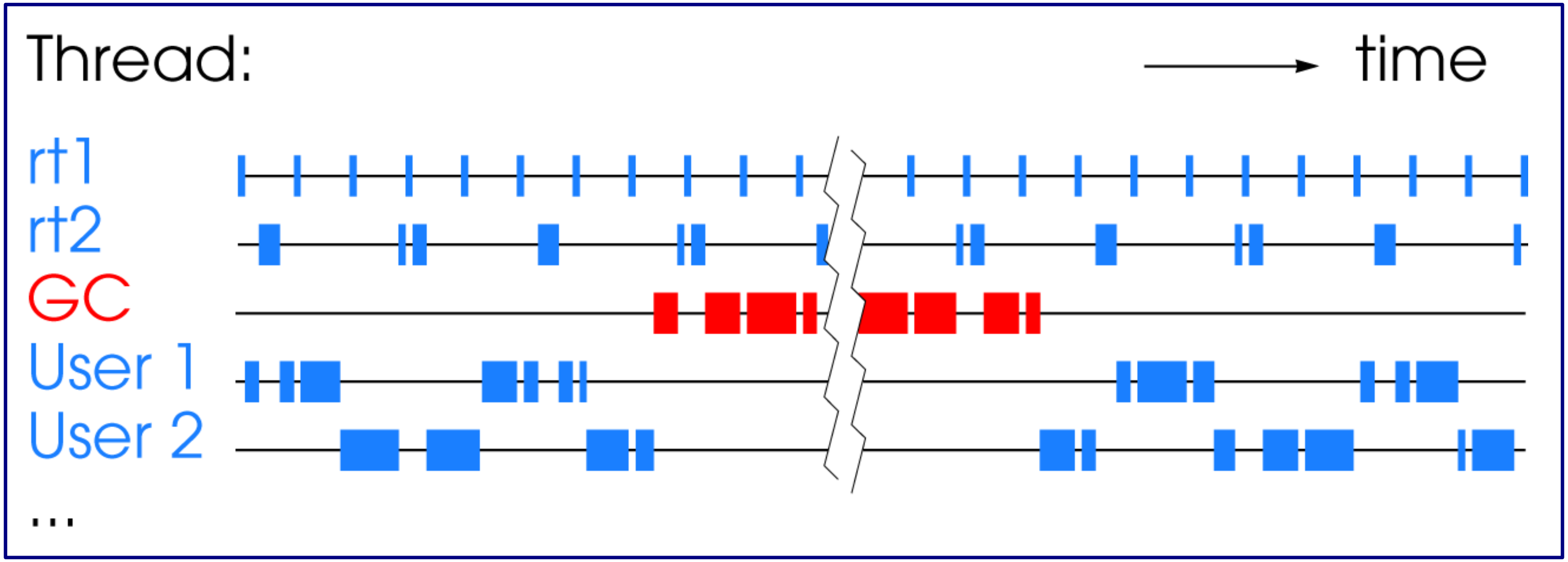


Problem

long, unpredictable pauses during execution

RTSJ & Conventional Garbage Collection

No heap threads can interrupt garbage collector:



The application must be split into a realtime and a nonrealtime part.

Realtime Garbage Collection

Paced garbage collector

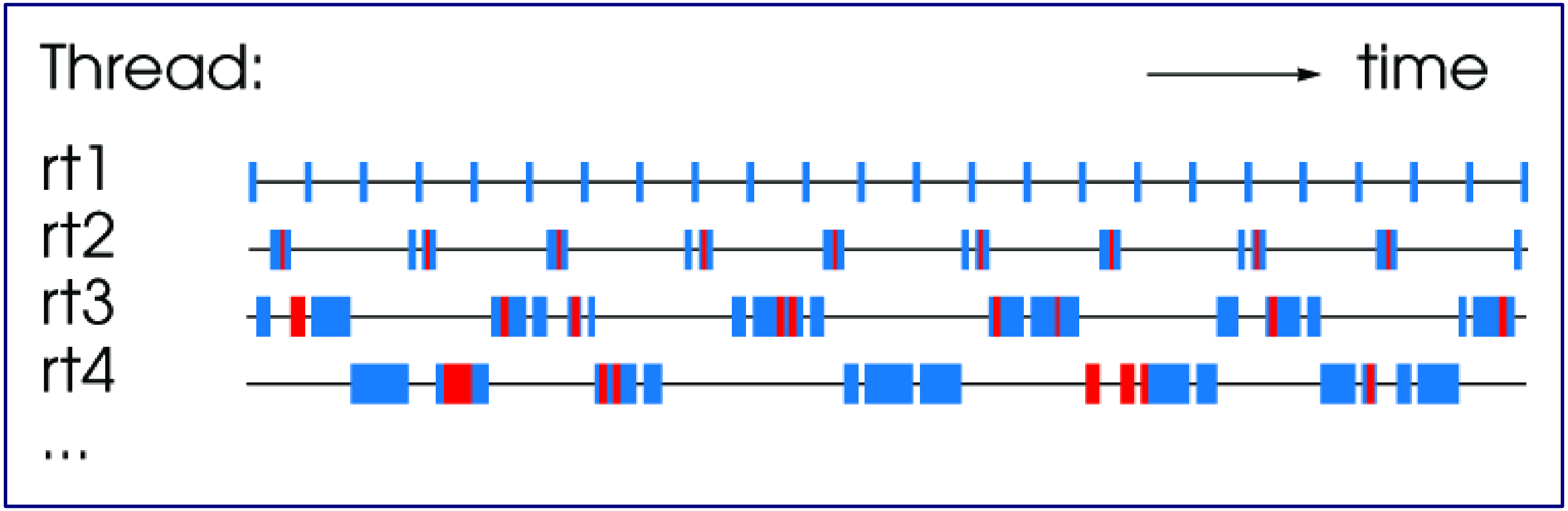
- Run GC at a high priority
- Runs at given interval, for given duration (Scheduling!)
- Programmer must provide both maximum memory use and maximum allocation rate

Slack garbage collector

- Run GC at lower priority than realtime tasks
- Runs when processor cycles are available (Scheduling!)
- Programmer must provide both maximum memory use and maximum allocation rate

Work-Based Garbage Collector

All Java Threads are realtime threads



- GC work is performed at allocation time
- GC work must be sufficient to recycle enough memory before free memory is exhausted
- Execution time of all allocations must be bound

Realtime Garbage Collection

Work based garbage collector

- No GC thread (not schedule!)
- GC borrows application thread
- Need only determine maximum memory use
- No read barriers needed
- Low latency

Also need priority inversion avoidance, realtime scheduling, and other concepts that the Real Java (RTSJ) has taken from Ada.

Real-Specification for Java

Conventional Java + deterministic GC !=
Realtime Java

Also need Real-Time Specification for Java

- priority inversion avoidance,
- realtime scheduling, and
- other concepts taken from Ada.

Example Period Task on LinuxRT

Period 200.00us, 7500 iterations

- min = 167.90us (84%)
- max = 231.80us (116%)
- average = 199.99us
- standard deviation = 965ns (0%)

Execution

- both calculation & allocation in each release
- 2–4 us

Conclusion

Generate state machines if you can, but

deterministic garbage collection is the best solution for complex safety-critical programs

Not just any Java implementation will do!