University of Glasgow | School of Computing Science

Garbage Collection

Advanced Operating Systems
Tutorial 4

Tutorial Outline

- Review of exercise 2
- Review of lectured material
- Discussion: real-time garbage collection

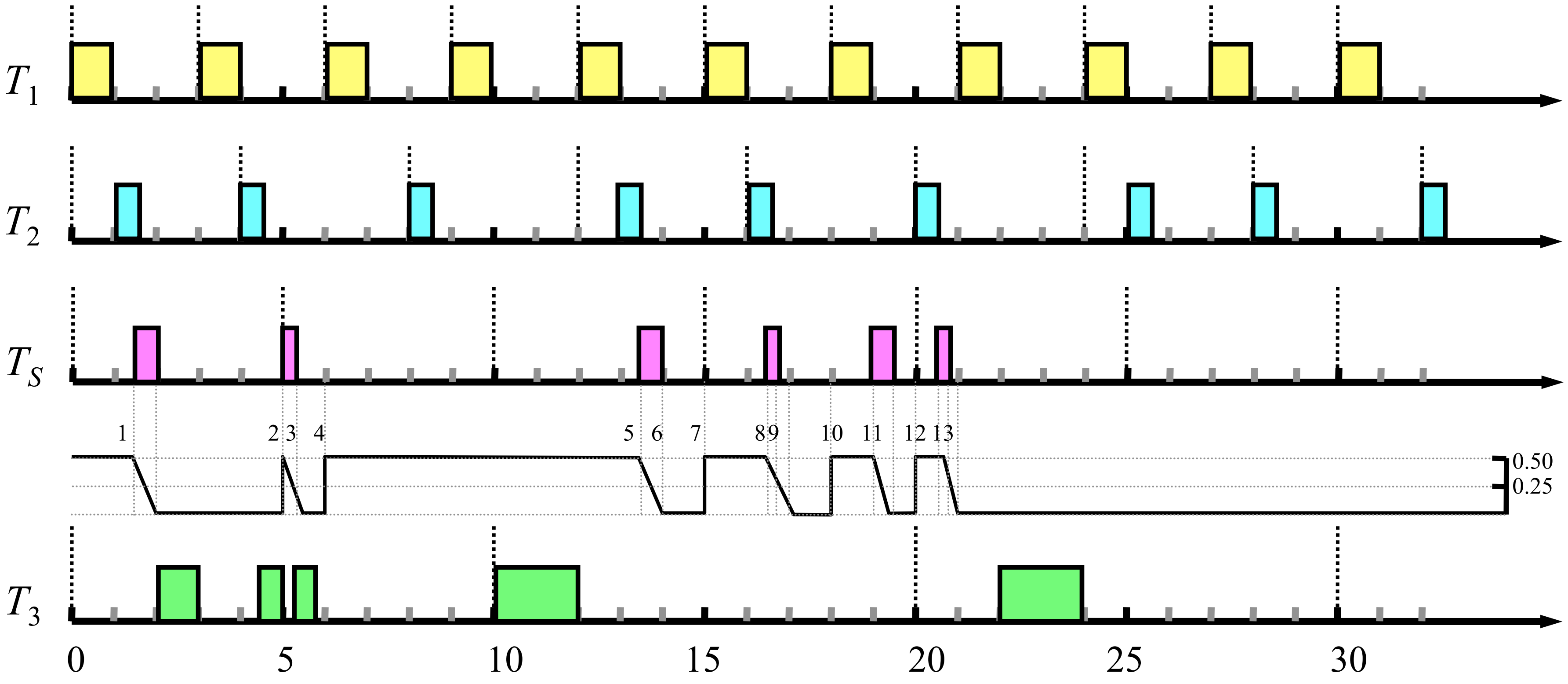
Review of Exercise 2

- Consider a system of periodic tasks: $T_1 = (3, 1)$, $T_2 = (4, 0.5)$, $T_3 = (10, 2)$. The system must support three aperiodic jobs:
    - A_1 is released at time 0.5
    - A_2 is released at time 12.25
    - A_3 is released at time 17
- The aperiodic jobs execute for 0.75 units of time. The system is scheduled using the rate monotonic algorithm, with a simple sporadic server $T_s = (5, 0.5)$ supporting the aperiodic jobs.
- Simulate the system for sufficient time to show how the aperiodic jobs are scheduled. What is the response time for each of the aperiodic jobs?

Review of Exercise 2: Worked Answer

1) C1; R2 $\Rightarrow t_e = \mathrm{MAX}(t_r, \mathit{BEGIN}) = 0$; replenish at $t_e{+}p_s = 5$
2) Replenished due to previous R2; executes according to C1
   R2 $\Rightarrow t_e = t_f = 5$ since $\mathit{END} < t_f$; replenish at $t_e{+}p_s{=}10$
3) Job A_1 ends, but T_s continues according to C2
4) Replenished early due to R3(b)
5) C1; R2 $\Rightarrow t_e = \mathrm{MAX}(t_r, \mathit{BEGIN}) = 12$; replenish at $t_e{+}p_s{=}17$
6) Budget exhausted (R3(a) does not apply, already replenished at step 4)
7) Replenished early due to R3(b)
8) C1; R2 $\Rightarrow \mathrm{t_e = MAX(t_r, BEGIN) = 15}$; replenish at $\mathrm{t_e{+}p_s{=}20}$
9) C2
10) Replenished early due to R3(b)
11) C1; R2 $\Rightarrow \mathrm{t_e = MAX(t_r, BEGIN) = 18}$; replenish at $\mathrm{t_e{+}p_s{=}23}$
12) Replenished early due to R3(b)
13) C1

A1 : 0.5 → 5.25 response time = 4.75
A2 : 12.25 → 16.75 response time = 4.5
A3 : 17.0 → 20.75 response time = 3.75



Review of Lectured Material

- Automatic memory management
  - Stack allocation
- Reference counting
  - Simple, incremental, problems with cycles
- Garbage collection
  - Mark-sweep
  - Mark-compact
  - Copying collectors
  - Generational collectors
  - Real-time collectors
- Practical factors

Key Learning Outcomes

- Concepts of automatic memory management
- Reference counting: what, when, and why?
- Garbage collection concepts
  - Basic mark-sweep algorithm
  - Limitations, and rationale for copying collectors
  - Generational collectors: concepts, advantages and disadvantages
  - Incremental collectors
    - Tricolour marking
    - Read- and write-barriers
    - For real-time use
  - Practical limitations

Discussion: Real-time Garbage Collection

- Problems with prior work
  - Fragmentation and inability to handle large data structures
  - High-space overhead
  - Uneven mutator (program) utilisation: garbage collector consumes significant fraction of available CPU time
- Basic operation of the real-time collector
  - Free lists for different size blocks
  - Non-copying (mostly) - arraylets
  - Incremental mark-sweep algorithm, with read barrier
  - Occasional copies, for defragmentation
- Real-time scheduling
  - Analytical analysis to show performance bounds
- Practical factors and implementation issues

A Real-time Garbage Collector with Low Overhead and Consistent Utilization

David F. Bacon
dfb@watson.ibm.com

Perry Cheng
perryche@us.ibm.com

V.T. Rajan
vtrajan@us.ibm.com

IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

ABSTRACT

Now that the use of garbage collection in languages like Java is becoming widely accepted due to the safety and software engineering benefits it provides, there is significant interest in applying garbage collection to hard real-time systems. Past approaches have generally suffered from one of two major flaws: either they were not provably real-time, or they imposed large space overheads to meet the real-time bounds. We present a mostly non-moving, dynamically defragmenting collector that overcomes both of these limitations: by avoiding copying in most cases, space requirements are kept low; and by fully incrementalizing the collector we are able to meet real-time bounds. We implemented our algorithm in the Jikes RVM and show that at real-time resolution we are able to obtain mutator utilization rates of 45% with only 1.6–2.5 times the actual space required by the application, a factor of 4 improvement in utilization over the best previously published results. Defragmentation causes no more than 4% of the traced data to be copied.

General Terms

Algorithms, Languages, Measurement, Performance

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.3.2 [**Programming Languages**]: Java; D.3.4 [**Programming Languages**]: Processors—*Memory management (garbage collection)*

Keywords

Read barrier, defragmentation, real-time scheduling, utilization

1. INTRODUCTION

Garbage collected languages like Java are making significant inroads into domains with hard real-time concerns, such as automotive command-and-control systems. However, the engineering and product life-cycle advantages consequent from the simplicity of programming with garbage collection remain unavailable for use in the core functionality of such systems, where hard real-time constraints must be met. As a result, real-time programming requires the use of multiple languages, or at least (in the case of the Real-Time Specification for Java [9]) two programming models within the same language. Therefore, there is a pressing practical need for a system that can provide real-time guarantees for Java without imposing major penalties in space or time.

We present a design for a real-time garbage collector for Java, an analysis of its real-time properties, and implementation results that show that we are able to run applications with high mutator utilization and low variance in pause times.

The target is uniprocessor embedded systems. The collector is therefore concurrent, but not parallel. This choice both complicates and simplifies the design: the design is complicated by the fact that the collector must be interleaved with the mutators, instead of being able to run on a separate processor; the design is simplified since the programming model is sequentially consistent.

Previous incremental collectors either attempt to avoid overhead and complexity by using a non-copying approach (and are therefore subject to potentially unbounded fragmentation), or attempt to prevent fragmentation by performing concurrent copying (and therefore require a minimum of a factor of two overhead in space, as well as requiring barriers on reads and/or writes, which are costly and tend to make response time unpredictable).

Our collector is unique in that it occupies an under-explored portion of the design space for real-time incremental collectors: it is a *mostly non-copying* hybrid. As long as space is available, it acts like a non-copying collector, with the consequent advantages. When space becomes scarce, it performs defragmentation with limited copying of objects. We show experimentally that such a design is able to achieve low space and time overhead, and high and consistent mutator CPU utilization.

In order to achieve high performance with a copying collector, we have developed optimization techniques for the Brooks-style read barrier [10] using an “eager invariant” that keeps read barrier overhead to 4%, an order of magnitude faster than previous software read barriers.

Our collector can use either time- or work-based scheduling. Most previous work on real-time garbage collection, starting with Baker’s algorithm [5], has used work-based scheduling. We show both analytically and experimentally that time-based scheduling is superior, particularly at the short intervals that are typically of interest in real-time systems. Work-based algorithms may achieve short individual pause times, but are unable to achieve consistent utilization.

The paper is organized as follows: Section 2 describes previ-

POPL’03, January 15–17, 2003, New Orleans, Louisiana, USA.
Copyright © 2003 ACM 1-58113-628-5/03/0001 $5.00.