# http://cs.LTH.se/EDA040

# Real-Time and Concurrent Programming

# Lecture 8 (F8): Real-time memory management for safe languages/Java. Examination hints.

Klas Nilsson

2016-10-18

1 Software correctness and safe languages
- Java deployment and motivation

2 Notes on embedded software
- Modularity outlook

3 Run-time systems
- Memory management

4 Hints for the exam
- Additional course content from this lecture
- Hints for the five hour written exam

# Some Real-Time Java approaches

- The real-time Java specification is overly complex, see spec at `http://www.rtsj.org/specjavadoc/book_index.html`
- There are various descriptions of the RTSJ available, such as the one referred to by this link.
- We contributed to the real-time JVM from Sun/Oracle, as reported on Using Real-time Java for Industrial Robot Control and as shown on YouTube, demonstrated in San Francisco.

Next, some development aspects...

# MULTI-STAGE DEPLOYMENT OF ROBOT CONTROL SOFTWARE

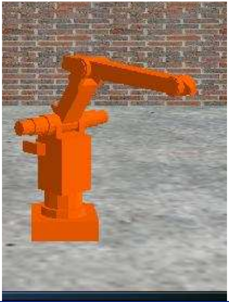
## A Java-based approach by

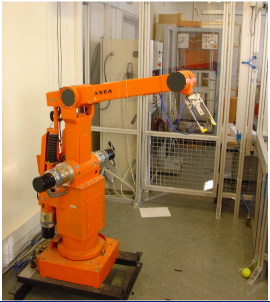

Sven Gestegård Robertz & Anders Nilsson

& ***Klas Nilsson*** & Mathias Haage

Dept. of Computer Science, Lund University, Sweden

**{sven|andersn|klas|mathias}@cs.lth.se**



# Robotic developments hampered by too complex/costly software engineering.

1. How can we trust that a deployed software component/function doesn't harm other (previously tested) parts of the control system?

- Use of a **safe** programming **language such as Java/C#** (with a strictly maintained sandbox model) for manually written code.

2. How can we stepwise deploy embedded control software such that the control and timing properties are verified in multiple small stages?

- **Multistage deployment** strategy, ranging from portable desktop-suitable simulation to cross compilation into target-specific hard real-time software functions.

Except for device drivers and automatically generated code we take a Java-based approach...... Why and how??......

# Deployment stages

1a) Running both the application and ***simulated environment*** in a ***standard JVM*** (J2SE from Sun) on a workstation.

1b) Running both the application and the simulated environment in a standard JVM on a workstation, using the ***free*** Java ***library*** classpath from GNU.

2) ***Natively compile*** application using LJRT, ***run on desktop*** in simulated environment.

3) Running natively compiled LJRT application with ***POSIX threads on target system***.

4a) Running natively compiled LJRT application with ***native RTOS threads in user space*** on target.

4b) Running natively compiled LJRT application with native RTOS threads, in **kernel space, on target system**.

Primary RTOS: www.xenomai.org

# Robot control depends on real-time software; use Java for portability and modularity!?

Typically experienced questions:

- **Why** would you do such a thing?
  The most industrially acceptable, freely available and portable way of imposing the modularity needed for robot software development: The Java **language**
- **How** would you do such a thing?
  As in the Lund Java-based Real-Time (LJRT) platform.

**Java-based**: *Full Java language with library subset.* ***Runs on any J2SE VM but not Java legally****. Cross-compilable to embedded targets by our compiler and run-time system.*

# Problems in software development

- Managing System Complexity
  Complex systems, weak structuring mecahnisms make it worse
- Managing System Development
  Late project, late errors makes it worse

What is the role of languages in this ?

- Errors detected eariler, in process, in tools
- Errors avoided

# Scalability(safety) $\rightarrow$ Java/C#

- Composing components and plugins during run-time for real-time: Safety and Modularity?
- Performance and predictability: Static type checking (whenever possible).
- Ensuring enabled error handling: Safe language required! (All possible executions are expressed by the program.)
- Automatic dynamic memory management.

# Unsafe language mechanisms

- Manual memory management (malloc-free/new-delete)
  - When to do free? – "when last pointer removed"!
  - Too early – dangling pointers
  - Too late – memory leaks
- Cast as in C
- Pointer arithmetic
- Arrays with no boundchecks – Programmer error leads to chaos

⇒ Problems often show up late, sometimes after long execution times

⇒ Program-wide consistency problem

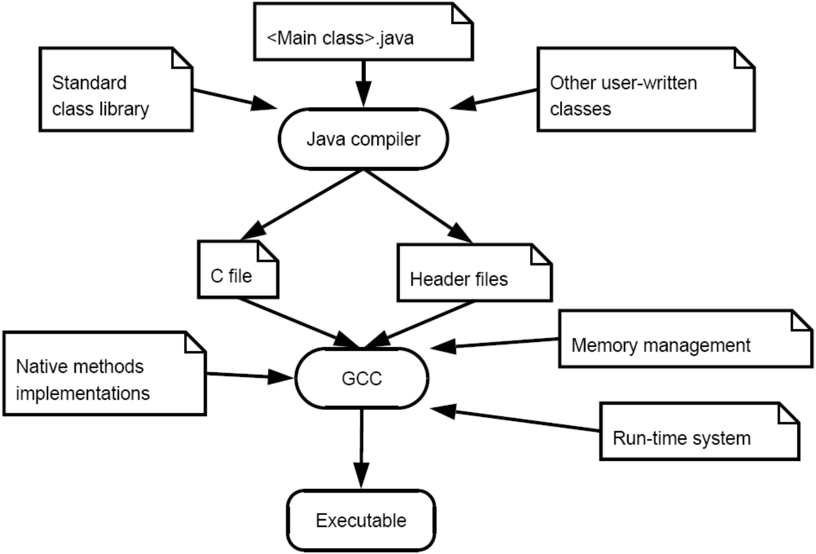
⇒ How to trust your robot?

# Answers from Safe languages

- Many errors caught by compiler
  - Remaining ones by runtime checks
  - Costs runtime efficiency
- Automatic memory management
  - When last pointer to object removed, object will be removed by Garbage Collector
  - Used to disrupt execution – not anymore
- Programming error leads to error message
  - no uncontrolled execution (i.e., no seg.fault, no illegal memory access, no blue-screen, . . . )
  - Uncontrolled execution would indicate an error in the platform – not in the program

Thus, a better separation between application and platform.

# Java compilation to native binaries via C-code



# Outlook on still valid topics of modularity

For your future profession in engineering,
but **not part of this course**,
the following 8 slides are provided for orientation
about the lack of modularity of software
compared to other engineering disciplines.

# Technical resources: embedded systems

## Resources in embedded systems

- Timing (CPU, HW,..)
- Memory (#bytes,..)
- Communication
- Device/unit-physical (energy, ..)
- Engineering effort

*Bounded and interconnected, use optimally for best possible product properties and profit*

**Cost:**
- Production
- Market opportunity

**Interface:**
- Interoperability
- Openness
- Usability
- Client satisfaction

**Adaptation:**
- Portability
- Modifiability
- Evolvability
- Expandability
- Flexibility
- Configurability
- Reusability
- Scalability

**Performance:**
- Performance (Speed)
- Timeliness (Deadlines)
- Determinism
- Security
- Robustness
- Reliability
- Availability
- Safety

**Design:**
- Feasibility
- Maintainability
- Understandability
- Correctness
- Simplicity
- Integrability
- Testability&Debugging

# The software crisis in robotics

”Architecture beats optimization”

”The mind exists to control the body”

Embedded information processing:

- Expensive monolitic systems today.
- Scalable software technology needed.

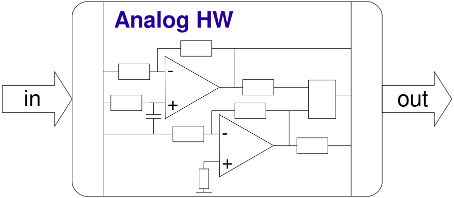
Current and future research:

1. Support for modular development and use of robotic software components, to enable modular robots that can assist humans in a flexible manner.
2. Enhanced technologies for implementation of control systems.

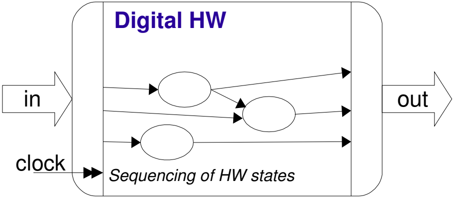
Resources for embedded systems, developments from past to present (for every improvement new drawbacks have resulted):

# Embedded information processing: Going from Analog to Digital (still hardware)

- \+ Direct effect
- \+ No quantization
- \+ Truly parallel
- – Cost
- – Repeatability (drift, etc.)

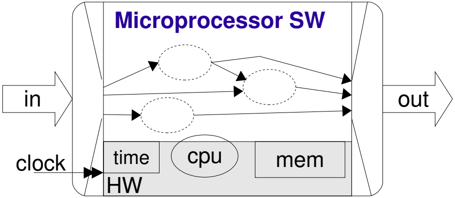



- \+ **Cost**
- \+ **Repeatability**
- \+ Truly parallel
- – Quantization
- – Latencies

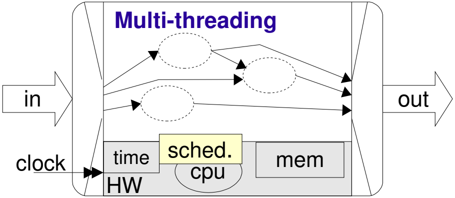


# Then programmable units (software), Multitasking RTOS (Real-Time Operating System)

- \+ **Cost reduction**
- \+ **Programability**
- – Modularity (resources)
- – Delays
- – Timing variations (jitter)

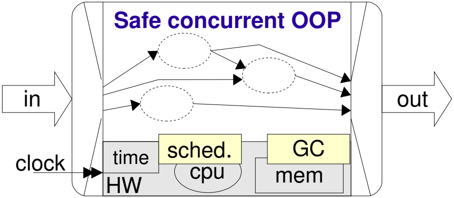



- \+ **Modularity of execution**
- \+ Flexibility
- – Predictability
- – Resource management
- – Modularity

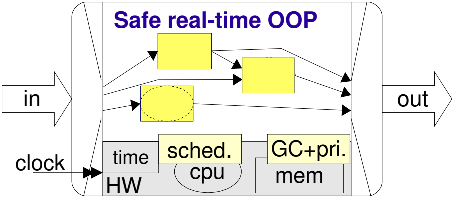


# Software components using safe language, first concurrency and then for real-time

- \+ **Modular reactivity**
- \+ **Safety**
- – Modularity (IO, memory)
- – Timing variations

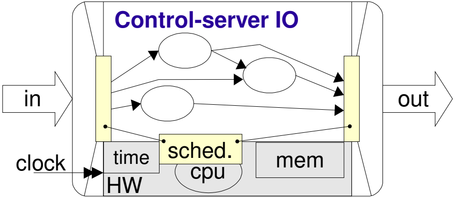



- \+ **Modular real-time**
- \+ **Robustness**
- \+ **Portable compilation**
- – Resource optimization
- – Timing variations for IO



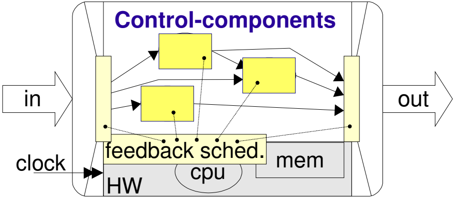
# Control components, scheduled IO, then feedback scheduling of resources

- \+ **Virtual CPUs**
- \+ **Composable IO**
- – Global memory
- – Resource optimization
- – Safety




---

- \+ **Performance tuning**
- \+ **Control components**
- – Global memory
- – Resource optimization
- – Safety



# Towards the "principle of superposition" for embedded software

Ongoing integration and further development:

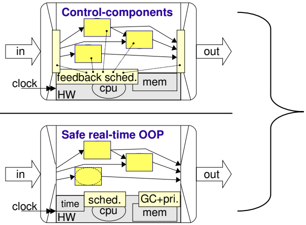
1) Object-oriented and portable safe real-time SW
2) Control components as composable SW

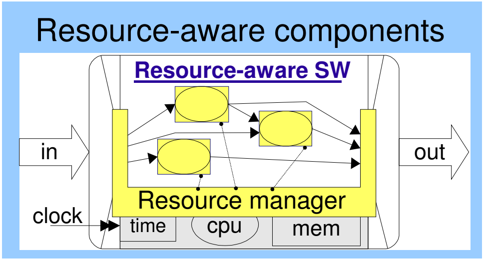
⇒ Resource-aware components & control systrems!

@LTH

Automatic Control:

Computer Science:



# RTGC

Next slides are shared with the Compiler Construction course EDA180.

(For details concerning a generation-based GC, see the CLR of Microsoft .NET folllowing this link.)

# Garbage Collection
## automatic memory management

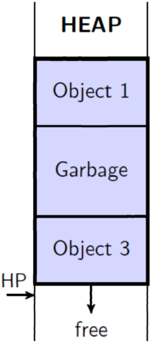

Roger Henriksson
Dept. of Computer Science,
LTH

# Presentation Outline

- Background
- Basic algorithms
- Generation-based GC
- C, C++, and conservative GC
- Incremental techniques
- Real-time systems and GC

# Memory organization

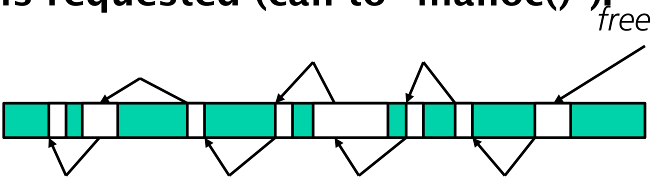
- Program code
- Global data
  - Static data
- Stacks
  - Activation records
  - LIFO – Last In First Out
- Heap
  - Random allocation/deallocation




How do we manage the heap efficiently?

# Manual memory management

- The application program is responsible for releasing objects not needed any longer (explicit calls to "free()").
- Blocks of free memory are linked into a list called a free list until new memory is requested (call to ”malloc()”).



# Common programming errors

## Dangling pointer

```
char *a,*b;

a = malloc(10);
b = a;
...
free(a);
...
printf("%s",b);
```

## Memory leak

```
char *a;
int i;

for(i=0;i<10;i++){
   a = malloc(10);
   sprintf(a,"%d",i);
   printf("%s",a);
}
free(a);
```

# Who should deallocate?

## From the Xlib API:

```
char *XGetAtomName(display,atom)
   Display display;
   Atom atom;

char *XGetDefault(display,program,option)
   Display display;
   char *program;
   char *option;
```

**Both functions returns a string, but who should deallocate it? The caller?**

# Who should deallocate?

## From the Xlib API:

```
XSetIconName(display,w,icon_name)
   Display display;
   Window w;
   char *icon_name;
```

**The function takes a string as parameter, but…**

**…can we deallocate the string after calling XSetIconName?**

# Fragmentation

Allocating/deallocating objects of varying size cause fragmentation. A request for a large block of memory might not be satisfied because only small blocks exist.

- Manual memory management ⇨ fragmentation
- Garbage collection without compaction ⇨ fragmentation
- Compaction requires less memory in the worst case.

## Example

- Max 100 KB live memory at any time, maximum object size 256 bytes ⇨ In the worst case 900 KB heap is required. [Rob71]

# Automatic memory management

**garbage collection** (eng., 'skräpsamling'), i databehandlingssammanhang, process vid dynamisk minnesanvändning där tidigare utnyttjade minnesceller, som ej längre kan nås från det exekverande programmet, automatiskt identifieras och anges vara tillgängliga för återanvändning.

Nationalencyklopedin

# Basic algorithms

- Reference counting
- Traversing algorithms
  - Mark-Sweep / Mark-Compact
  - Copying algorithms

# Reference counting

## Idea

- Each object contains a counter indicating the number of pointers referencing the object.
- The object can be deallocated when the counter becomes zero.

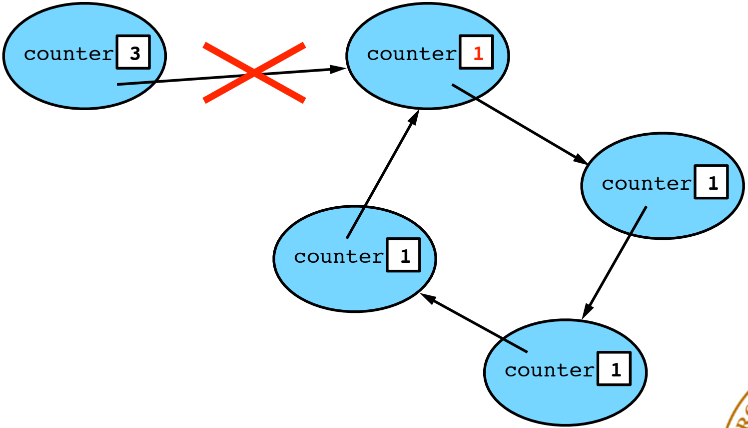
## Advantage

- Easy to implement. Usually short pauses.

## Disadvantages

- Expensive. The counters of affected objects must be updated whenever a pointer assignment is performed.
- No compaction ⇨ fragmentation.
- Fails to detect circular structures of garbage objects.

# Circular structures



# Traversing algorithms

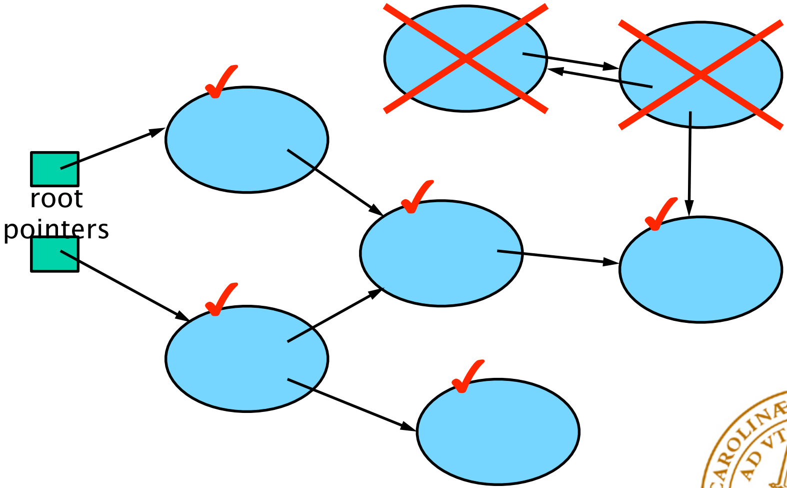
## Idea

- Periodically search (traverse) the entire pointer graph of the application, marking encountered objects.
- Objects not encountered during the marking phase is dead.
- Recursively traverse the pointer graph starting from a number of root pointers. A root pointer is a pointer located outside the garbage collected heap pointing to an object on the heap. Example: global pointers, pointers located on the stack or in the registers of the microprocessor.

## Requirements

- Runtime type information must be available for all objects:
  - How large is the object?
  - Where is the pointers within the object located?

# Traversing a pointer graph



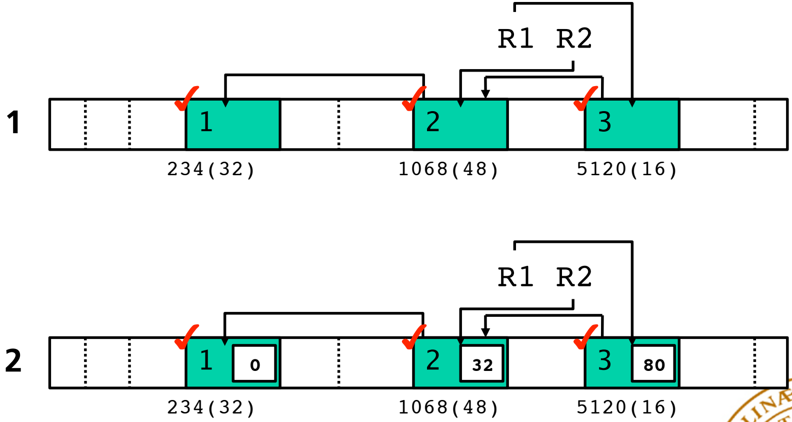
# Mark–Compact

**The compacting LISP 2–algorithm [Knu73] requires four passes.**

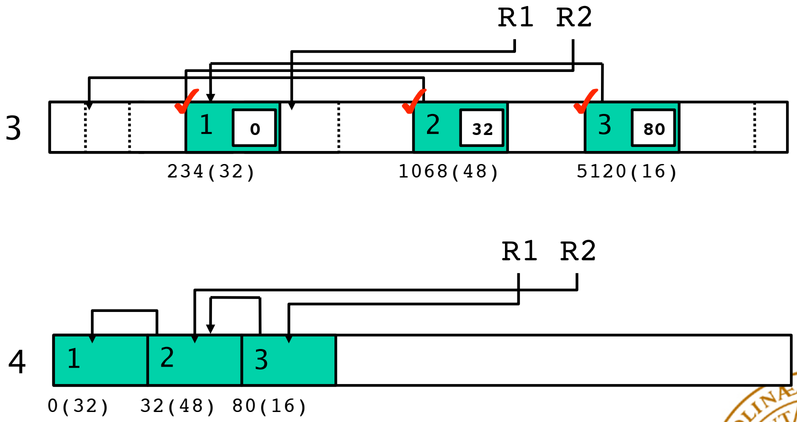
## Algorithm

- **Wait until no free memory remains on the heap.**
- **Pass 1: Recursively traverse the pointer graph starting from the root pointers and mark all encounterd objects (Mark).**
- **Pass 2: Determine where (address) each marked object will be located after compaction. Store the new address within the object.**
- **Pass 3: Update all pointers to point at the new addresses.**
- **Pass 4: Slide (move) objects into their new positions.**

# LISP 2, example



# LISP 2, example



# A copying algorithm

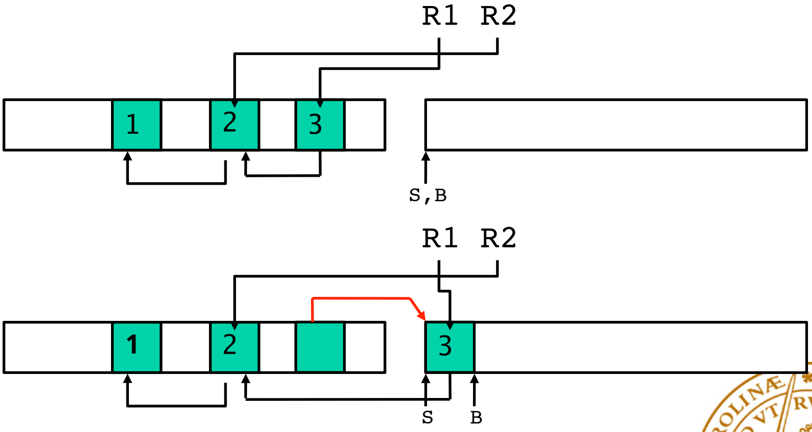
Memory is partitioned into two subheaps used alternating. When a subheap is full, the live objects are copied (or evacuated) into the empty subheap.

## Algorithm (according to Cheney [Che70])

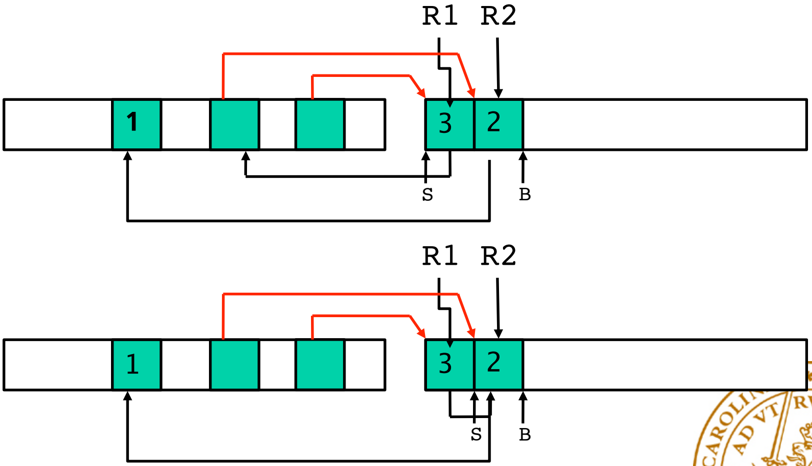
- Wait until the heap is full.
- Evacuate the objects referenced by the root pointers.
- Search the evacuated objects for pointers. Evacuate the objects referenced by these pointers if not already evacuated. Update the pointers to point to the evacuated version of the object.

A pointer to the evacuated version of the object is stored in the old version to facilitate updating other pointers to the object.

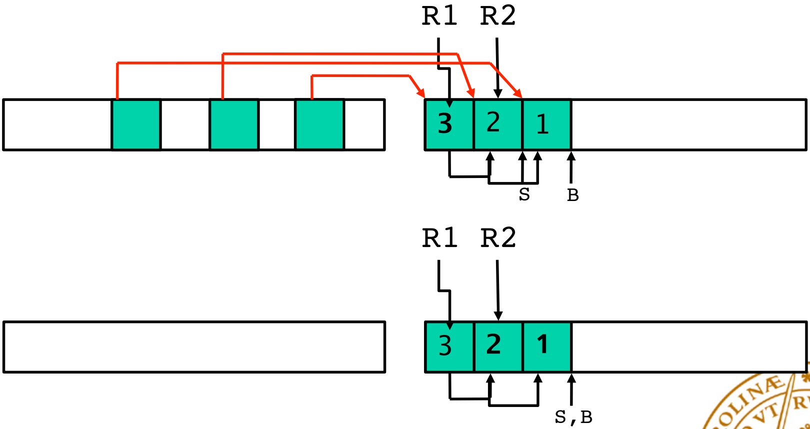
# Copying GC – example



# Copying GC, example



# Copying GC, example



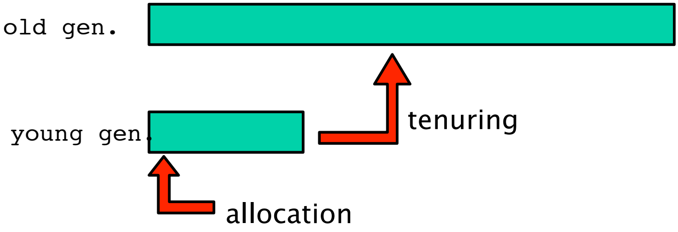
# Generation-based GC

- **Objects usually die young.**
- **Objects not affected by ”infant mortality” usually lives for a long time.**

## Generation-based GC! [Ung84]

- **Partition the heap into several ”generations” garbage collected separately.**
- **New objects are allocated in the young generation.**
- **Ageing (surviving) objects is promoted into the next generation (”tenuring”).**

# Generation-based GC




- **Efficient! Most pauses short. Most garbage collection work is performed in the young generation.**
- **Complex: Must keep track of inter-generation pointers.**

# Conservative GC

- ”Hostile” environment:
  no runtime type information available
- Example: C, C++.

**How do you identify pointers?**

# Conservative GC

## Strategy

- Assume that every word on the heap, stack, and statically allocated memory is a potential pointer.
- If a bit pattern can be interpreted as a pointer, regard it as a pointer!

## Problems

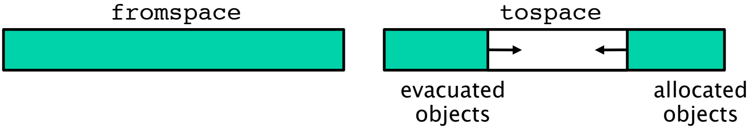
- Compaction difficult, we dare not alter pointers.
- Data can be misinterpreted as pointers - potential memory leak.
- Pointers can in some cases avoid detection - potential dangling pointers.

# Incremental GC

- Stop-the-world, long pauses (seconds)
- Incremental algorithms
  - Splits the GC work into many small increments (milliseconds).
  - Distributes the work over the execution of the application program (parallel GC).
  - Incremental variants av Mark-Sweep, Mark-Compact and copying algorithms.
  - Reference counting incremental by nature.

# Brook's algorithm

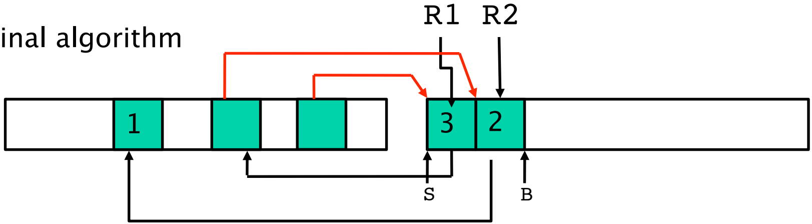
- Incremental copying algorithm.



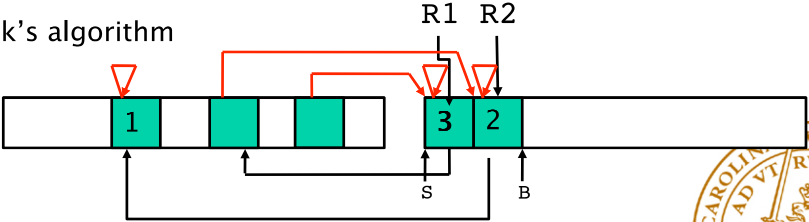

- Perform enough GC work in connection with every allocation to empty fromspace before tospace fills up (or deadlock).
- Incrementality requires
  - Fine-granular interruptibility. Heap consistency.
  - Read barrier: pointer access indirect via forwarding pointer
  - Write barrier: garantuees that the application does not change the pointer graph without the GC knowing it.

# Brook's algorithm

Original algorithm




Brook's algorithm



# Real-time systems and GC

- Response time requirements
  - Soft real-time systems
  - Hard real-time systems
- Individual pauses must be short and not too close together in time.
- Incremental algorithms required.
- Compaction?
- Hard real-time has been considered incompatible with GC, but…

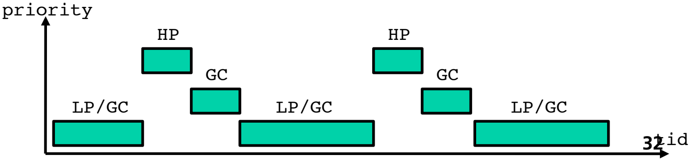
# Embedded control systems

- Control systems (JAS, industrial robots)
  - Small number of periodic threads with high priority. Hard real-time requirements.
  - Large number of low-priority threads. Soft real-time requirements.
- Requirements
  - Minimal response time for high priority threads.
  - Minimal latency for high priority threads (jitter).
  - Predictable (and low) worst-case response times.
  - Guarantee schedulability for the system.

# GC in hard real-time systems

## Idea [Hen98]

- Avoid doing GC work when high-priority threads execute. Perform GC in the pauses. Memory always available.
- Low-priority threads: standard incremental techniques.
- Minimize the cost for pointer operations for the high-priority threads.
- Interruptible garbage collection, minimum locking.
- Theory for a priori schedulability analysis.



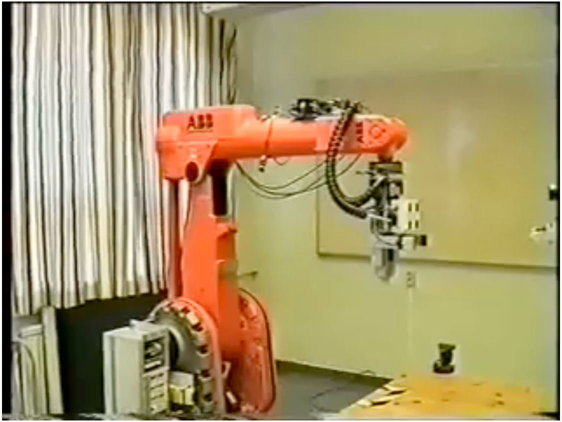
# Prototype

- VME-based control computer, 25 MHz Motorola 68040.
- Real-time kernel developed at Dept. of Automatic Control.
- Controls an ABB IRB-2000 industrial robot.

- Worst-case costs for high-priority threads
  - Pointer assignment: < 10 μs
  - Allocation: 32-76 μs (100-1000 bytes)
  - Locking: 60 μs

- Comparison to malloc/free:
  - malloc: 130-150 μs, free: 106-154 μs (typical)
  - malloc: 483 μs - 40 ms !!! (provoced worst-case)

# Inverted pendulum control

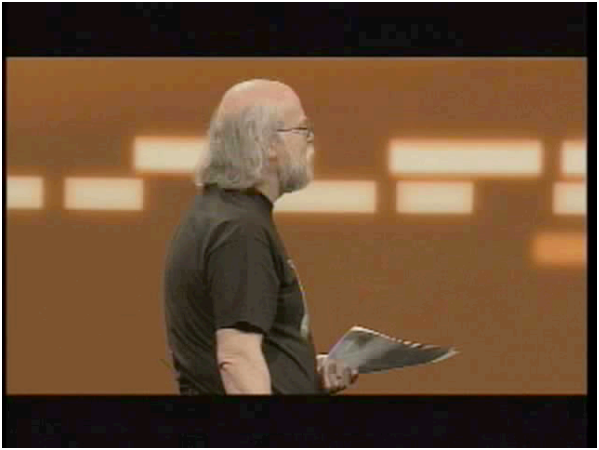


# Real–Time Java

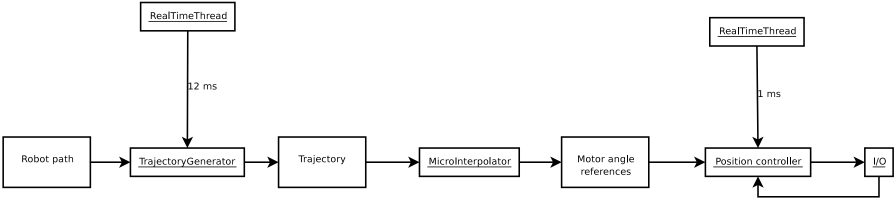
- RTSJ - Real-Time Specification for Java
  - New libraries
  - New thread and memory model
  - Predictable JVM

- Sun Java Real-Time System 2.0 (Sun JRTS 2.0)
  - Released May 2007.
  - Real-time GC from Lund University.
  - Industrial robot control project Sun/LU.

# JavaOne 2007 Demo



# Java robot control



# Bibliography

## Surveys


- **Richard Jones, Rafael Lins. "Garbage Collection – Algorithms for Automatic Dynamic Memory Management", John Wiley & Sons, 1996.**
- **Paul R. Wilson. "Uniprocessor Garbage Collection Techniques", IWMM´92, St. Malo, Frankrike, september 1992.
ftp://ftp.cs.utexas.edu/pub/garbage/gcsurvey.ps**

# Bibliography

## References


[Che70] C. J. Cheney. "A Nonrecursive List Compacting Algorithm", Communications of the ACM, 13(11), november 1970.

[Hen98] R. Henriksson. "Scheduling Garbage Collection in Embedded Systems", doktorsavhandling, Inst. för datavetenskap, Lunds Tekniska Högskola, september 1998. http://www.cs.lth.se/~roger/thesis.html.

[Knu73] D. E. Knuth. "The Art of Computer Programming, Vol 1", Addison-Wesley, 1973.

[Rob71] J. M. Robson. "An Estimate of the Storage Size Necessary for Dynamic Storage Allocation", Journal of the ACM, 18(3), juli 1971.

[Ung84] D. Ungar. "Generation Scavenging: A Non-disruptive High Performance storage Reclamation Algorithm", ACM Sigplan Notices, 19(3), maj 1984.

# Things you should know

**How to know what a program does/means;**
**Why Java (ur C# without use of `unsafe`[1])?**

- Safe languages: The notion of strong type safety for improving modularity, supported at compile-time and run-time.
- Under controlled deployment of open-source, compilation via C is an attractive option for portability to systems without an available RT-JVM.

**Memory management and Real-Time Garbage Collection (RTGC)**

- Dynamic memory allocation: Manual and Automatic.
- Properties of Automatic memory management; of GC algorithms.
- The fundamental RTGC principle: The Medium priority GC thread serving the High-priority threads, and the Low-priority threads performing the GC in their own context

[1] References to Microsoft .NET, such as C#, is not part of the course.

# The written examination

Hints:

- Notice the hints (8 items) on the Exam page of this course.
- Study problems and solutions of the December exam 2010
- Study the character of problems in various older exams.

Comments:

- Understanding the problem is part of the problem (in industry too).
- To understand what to solve in detail, and how to use the available time, is part of the problem (in industry too).

Old exam, orally and on board:

- How to structure concurrent software.
- Form of examination, grades and retakes.

Think threads and good luck!