F7: Memory management, Real Time Java

Memory management

- Static
- Manual
- Automatic (Garbage Collection)

RTGC: Real-Time Garbage Collection

- Automatic memory management for safe object oriented hard real-time systems.

Real Time Java

- Languages for embedded real time systems
- A proposed real-time Java standard
- Language requirements
- Real-time Java activities in Lund

Memory organization

Global data

Data available to all threads. Lives throughout the program execution.

Stacks

Local variables, temporary storage.

LIFO - Last In First Out

An activation record is created on the top of the stack each time a method is invoked.

Separate stack for each thread.

Heap

Shared between all threads. Objects are typically stored here.

Stacks can be allocated on the heap.

Random allocation/deallocation during runtime - unordered.

How do we manage the heap efficiently and in a predictable way?

Static Memory Management

- Simple, but not very flexible.
- The programmer or compiler determines the exact memory layout in advance.
- Not possible to dynamically allocate objects/memory during runtime.
- Should stacks be accepted?
  Example: Pascal D80 (Viggen fighter)

Pros

- Highly deterministic behaviour.
- No need for runtime system support.

Cons

- Very limited expressiveness - difficult to write efficient programs.

Suitable for very small systems or systems with extreme demands on predictability.

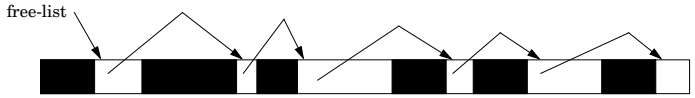
Manual dynamic memory management

Memory is assigned as requested by the application program.

Two operations:

- NEW (malloc) - Locates a suitable memory area and returns a pointer to the block.
- FREE - Returns a piece of memory to the pool of free memory.

All free memory blocks are linked together in a data structure - often called the *free-list*



Flexible, but:

- Complex code to manually administrate the memory.
  - Dangling pointers, memory leaks, fragmentation
  - Unpredictable cost for allocation/deallocation

Automatic dynamic memory management

Properties

- The runtime system is responsible for administrating the memory.
  A *garbage collector* reclaims memory not used by the application any more.
- No explicit deallocation.
- Easy to combine with compaction in order to avoid memory fragmentation.

Advantages

- Less complex application code.
- Less memory-related programming errors.
- Compaction => Deterministic allocation cost.

Problems

- Need type-safe languages.
- A challenge to implement non-intrusive garbage collectors.

Garbage Collection

Batch garbage collection

Stop-the-world

Incremental garbage collection

GC work performed in small increments interleaved with program execution.

Allocation, pointer reads, and pointer writes can trigger GC increments.

Concurrent garbage collection

Garbage collection work is performed by a separate thread. How do we guarantee progress?

Real-time

Traditionally based on incremental techniques:

- Each memory management operation may trigger GC work.
- GC work will cluster together.
- Accumulated costs for GC makes it hard to guarantee short enough response times.

Scheduling Real-Time Garbage Collection

Traditional

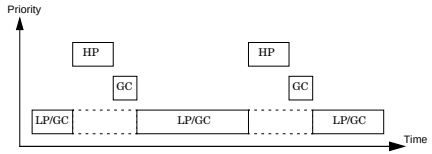
Interleave execution of application with garbage collection.

Long worst-case response times.

Our approach

Minimize response times for high-priority processes by removing GC work from high-priority processes - hard real-time.

Perform GC when no high-priority process executes.



Interleave low-priority processes with GC work - soft real-time.

Scheduling analysis can be used to prove schedulability.

Real-Time Java!

Robot video.

Prototype

Environment

VME-based control computer, 25 MHz Motorola 68040.

Real-time kernel developed at Dept. of Automatic Control, Lund.

Measurements

Worst-case costs for high-priority processes:

Pointer assignments: 10 µs

Allocation: 22-58 µs (100-1000 byte objects)

Locking: 60 µs

Comparison

malloc/free:

Malloc: 130-150 µs, Free: 106-154 µs (typical)

Malloc: 483 µs - 40 ms !!! (free-list with 50-5000 holes)

Embedded Real-Time Java

Languages in use

General-purpose languages: C, C++, Pascal, Ada, Modula, ...

Domain-specific languages: IEC 1131, Rapid, ...

Why Java? - Rationale

- Object-oriented safe language
  - Type safe
  - Automatic memory management
- Built-in threads and synchronization primitives
- Exception handling
- Platform independent

Make modern languages useful for embedded systems!

Real-Time Java standards

The Java standard must be adapted for meeting real-time demands.

Two competing standards

- *Real-Time Specification for Java (RTSJ)* by The Real-Time for Java Expert Group. backed by Sun Microsystems. *http://www.rtj.org*
- *Real-Time Core Extensions* by J-Consortium. *http://www.j-consortium.org*

Some good ideas, but also bad ones

- "There are no such thing called real-time garbage collection."
- Numerous memory and thread types.
- Effectively puts the responsibility for memory management on the programmer.

Sun Microsystems - LTH collaborate research

LTH cooperates with Sun Microsystems to develop an implementation of RTSJ which supports real-time garbage collection.

Available in the Sun Real-Time Java System version 2.0.

Lund Java-based Real-Time System (LJRT)

Design goals

- Low memory footprint
- Speed
- External code
- Determinism and latency
- Portability
- Keep the original thread/memory model of Java.

LJRT Approach

Small memory footprint and high performance
-> Natively compiled Java

Any hardware comes with a C compiler
-> Use ANSI C as intermediate (high-level assembly) language

Development platform

Standard GNU/Linux.

Target platform examples

- Atmel AVR Atmega 128. 8 bit RISC at 6 MHz. 128 kB flash and 36 kB RAM.
- Power PC G3. 32 MB RAM running GNU/Linux with Xenomai. Controller for ABB IRB6 industrial robot written completely in Java.
- Axis ETRAX. 32 MB RAM and 8 MB flash.

LJRT Compilation Process