LUNDS TEKNISKA HÖGSKOLA
Lunds universitet

# 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**



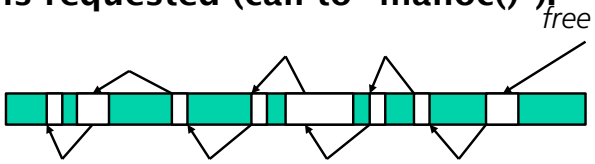
## Memory organization

- Program code
- Global data
  - Static data
- Stacks
  - Local variables, return address, etc.
  - 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

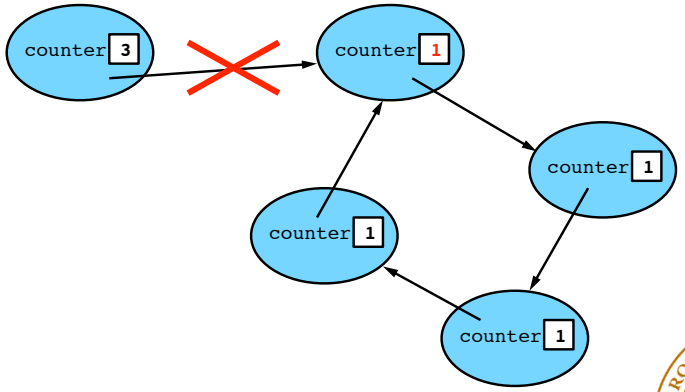
- Easy to implement. Usually short pauses.

## 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

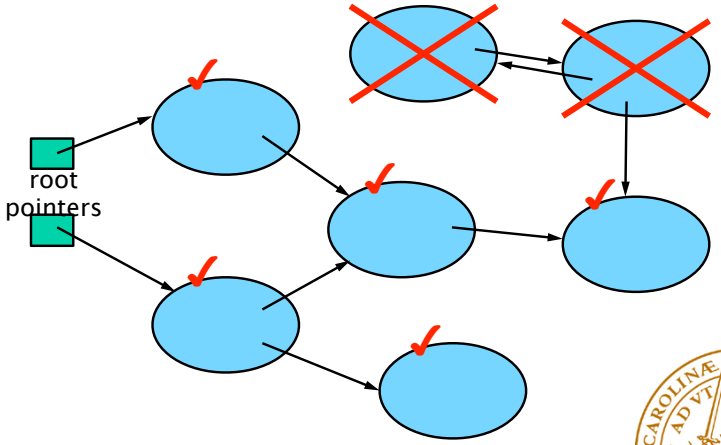
- Periodically search (traverse) the entire pointer graph of the application, marking encountered objects.
- Objects not encountered during the marking phase is dead.
- Recursively traverse the pointer graph starting from a number of root pointers. A root pointer is a pointer located outside the garbage 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





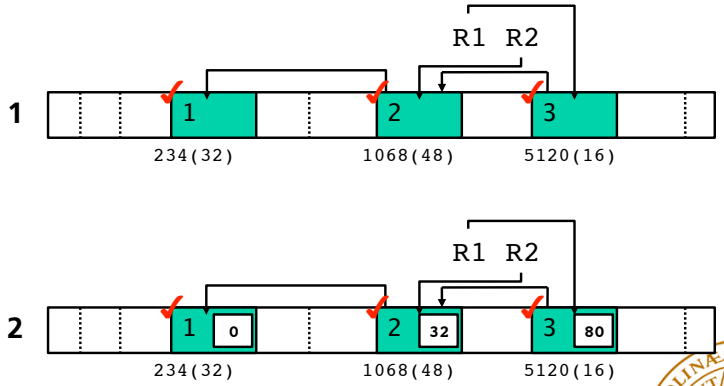

# Mark-Compact

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

## Algorithm

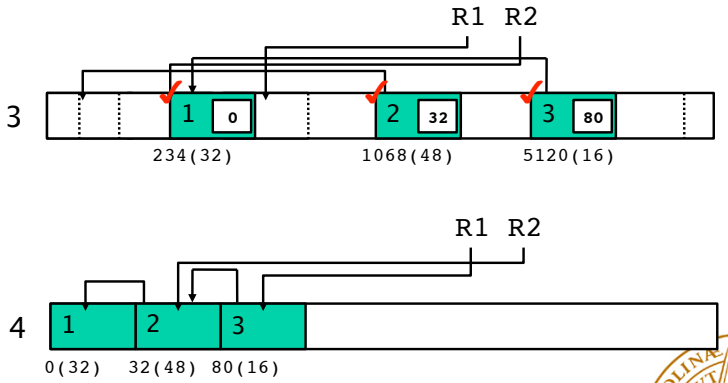
- Wait until no free memory remains on the heap.
- Pass 1: Recursively traverse the pointer graph starting from the root pointers and mark all encounterd objects (Mark).
- Pass 2: Determine where (address) each marked object will be located after compaction. Store the new address within the object.
- 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.

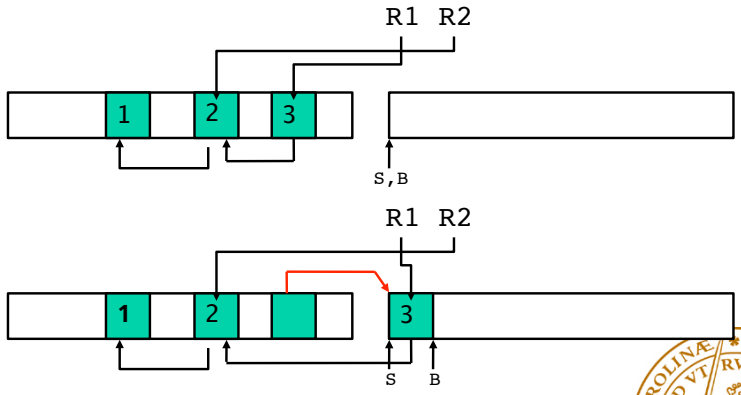
**Algorithm** (according to Cheney [Che70])

- 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.

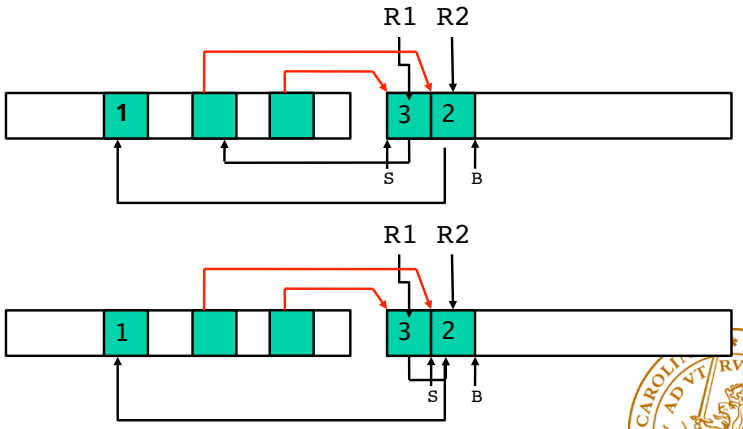
A pointer to the evacuated version of the object is stored in the old version to facilitate updating other pointers to the object.



# Copying GC – example

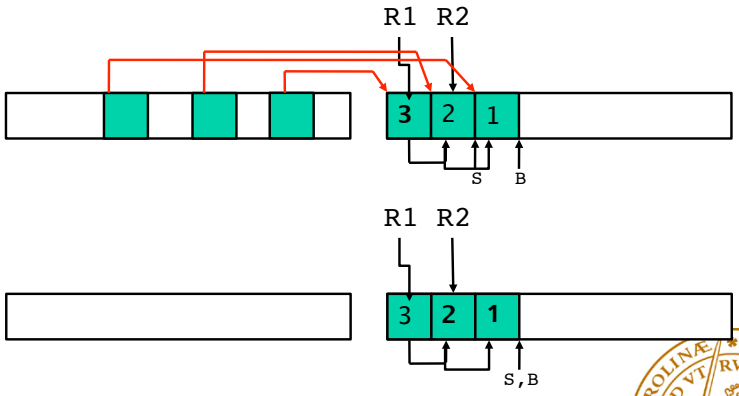


# Copying GC, example






# Copying GC, example






# Generation-based GC

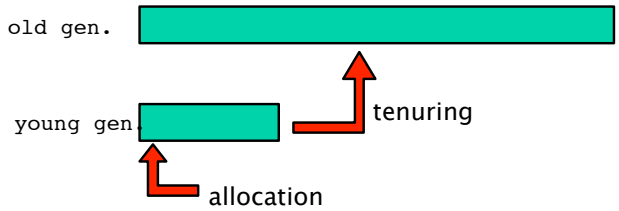
- Objects usually die young.
- Objects not affected by ”infant mortality” usually lives for a long time.

## 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.



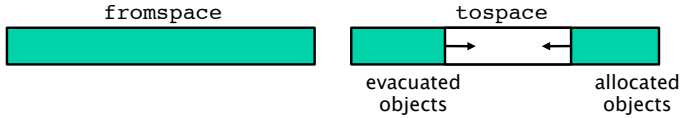
# Incremental GC

- 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.



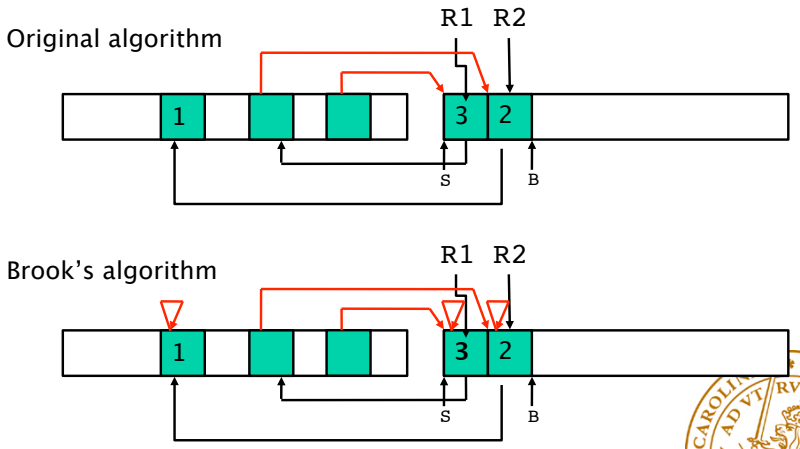
# Brook’s algorithm

- 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






# 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

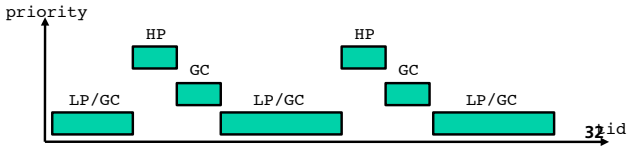
- Control systems (JAS, industrial robots)
  - Small number of periodic threads with high priority. Hard real-time requirements.
  - Large number of low-priority threads. Soft real-time requirements.
- Requirements
  - Minimal response time for high priority threads.
  - Minimal latency for high priority 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

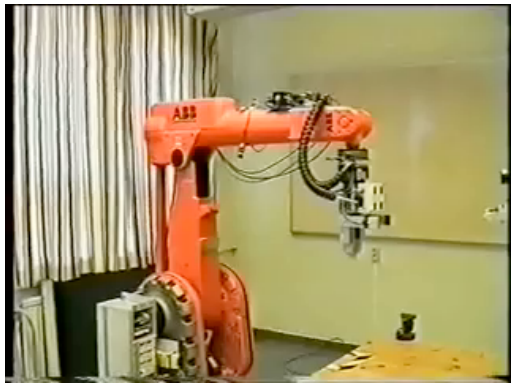
- VME-based control computer, 25 MHz Motorola 68040.
- Real-time kernel developed at Dept. of Automatic Control.
- Controls an ABB IRB-2000 industrial robot.

- 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





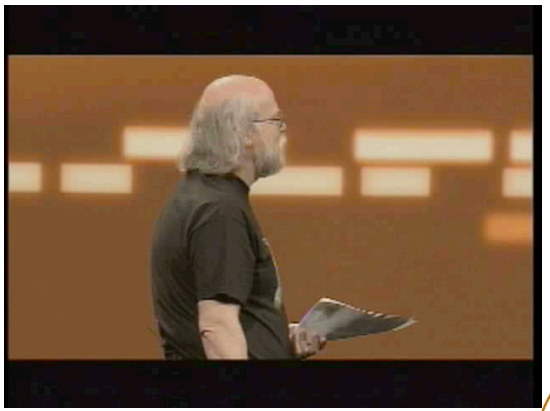

# Real-Time Java

- 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.

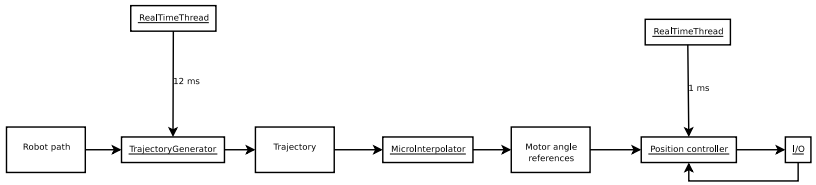


# JavaOne 2007 Demo





# 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.