An Introduction to Garbage Collection
tobias.ritzau@ida.liu.se

What is a Garbage Collector?
- The task of a garbage collector is to reclaim memory when it is of no more use to the application
- In most cases this requires both compiler support and a runtime system

Why use GC?
- Many memory related bugs that are hard to find and correct are eliminated
  - Pre-mature freeing
  - Memory leaks
  - Duplicate freeing of the same memory area
- Often the symptom occurs far away from the cause of the bug

Still Memory Leaks?
```java
class Stack {
  Object[] stack = new Object[128];
  int sp = 0;
  void push(Object obj) { stack[sp++] = obj; }
  Object pop() { return stack[--sp]; }
}
```

What is the Cost?
- Usually an execution time overhead of 5 % - 15%
- Memory overhead is more complicated
  - Depending on technique 33 % - 100 % of the heap can be used to store objects
  - If all runtime info is taken into account some systems require 600 % memory overhead...
  - As usual more memory gives faster execution and vice versa

Definition of Terms
- **Object graph** – memory areas form the nodes and the pointers between them are the edges
- **Child** – any area that is accessed by another is a child of the accessing one
- **Parent** – A is a child of B → B is a parent of A
- **Root** – an object that is in use by the application
- **Live** – an object that is always alive (e.g. static, global, the runtime stack)
- **Reachable** – object that can be reached from a root in the object graph (live → reachable)
Three Basic Techniques

- All widely used garbage collection techniques can be derived from one of or a combination of three basic techniques
  - Mark and Sweep
  - Copying Garbage Collection
  - Reference Counting

Mark and Sweep

- Mark objects recursively starting at the roots
- All reachable objects are now marked and thus all live ones
- All non-marked objects can now be swept

Mark-Sweep

- The heap is divided into two sub-heaps
- One is labeled from-space and the other is labeled to-space
- All objects are allocated in to-space
- When to-space becomes full a flip is made
- To-space now becomes from-space and from-space becomes to-space
- Objects are now copied recursively from from-space to to-space starting at the roots

Copying

- All objects contain a reference counter
- The reference counter records how many references point to the object
- The reference counter needs to be updated by the application
- When the reference counter falls to zero the object can be reclaimed

Reference Counting
Reference Counting

Incremental GC
- So far the collectors have been "stop-the-world-collectors" which is not good for interactive systems (for example)
- An incremental GC performs the GC work in multiple increments
- It is important to maintain the GC information while the application is running

Tri-color Marking
- White objects have not yet been found
- Gray objects have been found but all children have not been investigated
- Black objects have been fully analyzed
- The application must almost never create edges from black to white objects!

Barriers
- Barriers are used to protect application code that updates the object graph
- Barriers exist in two variations:
  - Read barriers are inserted when references are read
  - Write barriers are inserted when references are written

Baker’s Copying
- An incremental copying collector with a read barrier
- All objects that are read can be stored in a black object and if the object is white it will create a black to white edge
- To ensure that this does not happen the read barrier copies the object (makes it gray)

Brooks’ Copying
- An incremental copying collector with a write-barrier
- All objects that are written are copied into to-space
- All objects are accessed via a forward pointer stored in a field of every object
Some other Collectors

- Baker’s Treadmill
- Generation Scavenging
- The Train Algorithm
- Beltway Collectors
- Replication Copying
- Mark and Compact

Baker’s Treadmill

- Instead of using two sub-heaps two doubly linked list represents to- and from-space
- Since a double linked list is used it is fast and easy to move an object between the spaces
- Problems:
  - Doubly linked list requires much memory
  - All objects are of the same size

Generation Scavenging

- Objects tend to “die” young
- Objects are moved to “older generations” when they grow old
- Older generations are scanned less often
- Pointers from older to newer generations must be recorded (write barrier)
- When an older generation is collected all younger is too

The Train Algorithm

- Collecting the oldest generation can take a lot of time
- Here the oldest generation is divided into trains which in turn are divided into cars
- Using the train algorithm only one car need to be collected at every full invocation

Beltway Collector

- Beltway collector is a generalization of copying garbage collectors
- It can be configured at runtime to act like most copying collectors
- Two new configurations are presented and they outperform the other configurations in the benchmark

Replication Copying

- Using replication copying the application always use the from-space object
- No expensive synchronization is required
- The write-barrier must keep a replica in to-space up to date (to use after the flip)
- Keeping the replica up to date is expensive
Mark and Compact

- Mark and Compact eliminates external fragmentation and use far less memory than copying collectors do
- Live objects slide to one side of the heap
- Incremental mark and compact exist but there is no perfect solution to allocate new objects

GC without Type Information

- Conservative collectors guarantee that data that can be interpreted as pointers is
- Some leaks occur
- Some optimization may cause memory to be prematurely reclaimed
- GC for C/C++
  - www.hpl.hp.com/personal/Hans_Boehm/gc/

Real-Time GC

- Everything has to be predictable
- Both memory usage and execution time!
- Three presented solutions:
  - Real-time Brooks’
  - Real-time mark and sweep
  - Real-time reference counting

RT Copying GC

- An analysis that guarantees that given a specific memory usage the system will not run out of memory
- The system does GC work while leaving high priority processes and while allocating in low priority processes

RT Mark Sweep

- To eliminate external fragmentation the heap is divided into equally sized blocks
- These blocks are collected using a standard incremental mark and sweep collector
- GC work is performed during allocation

RT Reference Counting

- Same block structure as real-time mark and sweep
- GC work is performed during allocation but can also be deferred during high priority processes
RT Comparison

- RT Copying was first but requires up to 6 times the used memory.
- RT Mark and Sweep has slower object access (block structure) but "only" requires up to 4 times the used memory.
- RT Reference Counting can not handle cycles and has slow object access but does well with twice the used memory.

All figures include all runtime information required by modern OO languages like Java.

Static Garbage Collection

- By statically analyzing the program it is sometimes possible to find out where memory can be reclaimed.
  - Reduce or eliminate the performance loss caused by runtime GC.
  - Predict memory usage.

Static Garbage Collection

By statically analyzing the program it is sometimes possible to find out where memory can be reclaimed.

- Reduce or eliminate the performance loss caused by runtime GC.
- Predict memory usage.

Escape Analysis

- Analyzes the data flow to discover if a reference to an object escapes the method where it is allocated.
- If it does not escape:
  - It can be allocated on the stack.
  - It can only be used by one thread.
- Tests suggests that around 20% of the allocations can be allocated on the stack.

Extended Escape Analysis

- Further analysis can also handle objects that does escape.
- The analysis performed in HIDOORS determines how high up in the stack a reference escapes.
- Tests suggests that about 20% more allocations can be handled.

Reasons for SGC failure

- Objects that are:
  - Passed to native methods.
  - Thrown as exceptions.
  - Accessed by multiple threads.
  - Accessed by a finalizer.
  - Passed to global scope.

Eliminating the Runtime GC

- Native methods can be marked as safe.
- Thrown objects can often be handled by allocating one exception object per thread.
- Objects that are accessed by multiple threads can be allocated statically.
- Finalizers could (should?) be avoided.
- No object should be stored in global scope if it is not permanent.
Further Reading

www.cs.kent.ac.uk/people/staff/rej/gc.html