CMPSCI 377 Operating Systems **Spring 2007**

Lecture 6: February 20

Lecturer: Emery Berger *Scribes: Diack Ba, John Gaquin*

This lecture covers garbage collection (GC). The basic concepts of GC will be covered, as well as many of the pros and cons of using GC versus normal memory allocation.

6.1 Garbage Collection

6.1.1 Terms To Know

Tracing
Copying
Conservative
Parallel GC
Concurrent GC
Live objects
Reachable objects
References

6.1.2 Explicit Memory Management

Traditionally, memory for program data is allocated using malloc/new and free/delete:

- `malloc / new`
  Allocates memory for an object
- `free / delete`
  Frees memory used by an object, allowing the space to be used by other objects

This form of memory management leaves efficiency up to the developer. Quite often developers will make mistakes when using these tools, which can cause many problems such as memory leaks. A solution to this is garbage collection.

6.1.3 Solution: Garbage Collection

A garbage collector is a process that will periodically scan the heap to determine which objects are live or garbage. It will reclaim space on the heap if it determines that an object is garbage, hence the term garbage collector.

- **Live Objects**
  An object in the heap is considered 'live' if it can still be referenced by any other object in the heap

- **Garbage**
  Conversely, an object is considered 'garbage' if it can no longer be reached by any other object. Logically, this would mean that it is no longer needed and the space can be reclaimed.

This process would free the developer from ever needing to worry about memory management. It would allow the developer to focus on the aspects of their own program, and the garbage collector would (hopefully) always be as efficient as possible.

- **Disadvantages of Garbage Collection**
  "Garbage Collection impairs performance!"

  - **Additional process to run**
  - **Degrades cache performance**
  - **Degrades page locality**
  - **Increase in memory needs**

- **Advantages of Garbage Collection**
  "GC enhances performance!"

  - **Faster memory allocation**
    Simple pointer bumping
  - **Increased cache performance**
    No need for headers
  - **Better page locality**
    Compacts data and reduces fragmentation

6.2 Classical Algorithms

There are three classical algorithms for garbage collection:

- **Mark-sweep**
- **Reference counting**
- **Semispace**

Additionally, there is Generational Garbage Collection which turns out to be the best.

6.2.1 Mark-sweep

Mark-sweep keeps track of live vs. garbage objects using a garbage bit on each object. Using this bit, it will mark each object as live or garbage depending on the environmental state it is in:

- Initially considers everything in the heap as garbage
- Starts with root objects; flags those as live objects.

- Recursively checks all objects that are referenced from within each object
  - If object is marked as garbage, mark it as live and continue recursing deeper
  - If object is already marked as live, stop. All objects referenced by this one will already be marked as live.
- All objects in heap that are left marked as garbage will be freed from the heap

6.2.2 Reference Counting

- For every object, maintain a reference count which is the number of incoming pointers
- $a- > ptr = x \Rightarrow refcount(x) + +$
- $a- > ptr = y \Rightarrow refcount(x - -); refcount(y) + +;$
- $reference\ count = 0 \Rightarrow no\ incoming\ pointers \Rightarrow garbage$

6.2.3 Semispace

Semispace divides the heap in half:

- **from-space**
  All memory is allocated in the from-space half of the heap
- **to-space**
  The to-space half is not used for normal memory allocation

When the from-space half is full, it runs a mark-sweep style garbage collection on that half and copies all live objects to the to-space half. The to-space half becomes the from-space, and the from-space becomes the to-space. The effective heap size is halved using this method.

6.2.4 Generational GC

- Optimization for copying collectors
- **Generational hypothesis**
- common case optimization
- Allocate into the **nursery**
- small region
- collect frequently copying survivors
- Keep track of pointers from mature space into nursery.

6.3 GC vs. malloc/free

- using GC in C/C++ is easy.

6.3.1 Other garbage collector

- Compares **malloc** to GC, but only **conservative**,non-copying collectors
  - Can't reduce fragmentation or reorder objects
  - faster,precise copying collectors
    - Incompatible with c/c++
    - Standard for Java

6.3.2 Oracular Memory Manager

- check oracle at each allocation
  - Oracle does not disrupt hardware state
  - When needed,modify instruction to call **free**
  - Extra costs hidden by simulator

6.3.3 Object Lifetime and Oracle Placement

- Oracles bracket placement of **frees**
  - **lifetime-based**: most aggressive
  - **Reachability-based** : most conservative

6.3.4 Liveness Oracle Generation

- **Liveness** : record allocations, memory accesses
  - Preserve code,type of objects
  - May use objects without accessing them

6.3.5 Reachability Oracle Generation

- **Reachability:**
  - illegal instructions mark heap events

6.4 Take-home practitioners

- Practitioners: GC is ok
  - if system has more than 3x needed RAM
  - if no competitions with other processes
- GC is not ok
  - Limited RAM
  - competition for memory
  - Depends on RAM for performance

6.4.1 Take-home: Researchers

- GC performance is good enough with enough RAM
- but performance suffers for limited RAM
- it is more susceptible to paging