Part XI : Dynamic Memory Management

- Dynamic Memory
- Allocation
- Manual Deallocation
- Garbage Collection
  - Reference Counting
  - Mark & Sweep
  - Two-Space Collectors
  - Generational Collectors
- New Challenges:
  - Real-time collectors
  - Distributed collectors

Dynamic Memory

- Most languages need to allocate data on the heap.
- This is always necessary for data which
  - is not copied on assignment, parameter passing, and
  - lives longer than the function which created it.
- For instance in misc: Lists can live longer than the functions that create them.
- Two questions:
  - How is heap data allocated?
  - Once heap data is no longer used, how is the memory it occupies reclaimed?
- Both of these are tasks done by the runtime system.
- The runtime system is a set of system procedures and (possibly) threads which is linked with each executed program.

Object Allocation and Deallocation.

- Basic scheme:
  - The system keeps a free-list of unused memory blocks.
  - To allocate an object, the free-list is searched for a block which is large enough.
  - Many variations of this scheme have been explored:
    - first fit, best fit, exact fit with bounded number of freelist block sizes..
    - Address order, or FIFO, or LIFO search, or round robin.
  - The found block is removed from the free-list, and is used to store the object.
  - If no block is found, the program is aborted (for systems with only physical memory), or the heap size is increased.
  - Any remaining memory is returned to the freelist.
  - To deallocate an object, its memory is returned to the freelist.
  - Freelist blocks which are adjacent to each other are merged to create larger blocks.

Challenges

- Allocation and deallocation times.
  - How long does it take to search the freelist?
- Time for « book-keeping », i.e. entering and removing objects Memory fragmentation:
  - After a number of allocation and deallocation steps, one might end up in a state where the total size of free memory is larger than the size of an object to be allocated, yet there is no single reelist block which is large enough.
  - Two forms
    - *Internal fragmentation* is the wastage of memory because larger blocks than necessary are taken from the freelist.
    - *External fragmentation* is the wastage of memory from blocks on the freelist which are too small to allocate an object.

Measuring and Avoiding Fragmentation

- Measure of fragmentation during a program run:
  size of total heap required / max size of allocated objects.
- Best techniques depend very much on object size distribution:
  - Knuth, "The art of computer programming, Volume 1":
    - First fit better than best fit.
    - However, Knuth uses synthetic random distribution of object sizes.
  - Recent measurements with real world object-size distribution:
    - Best fit much better than first fit;
    - Address-ordered or FIFO best fit are best.
    - However, this is also slowest for free list search unless optimizations are added.
- Once the heap is too fragmented it can be *compacted* by moving objects around, but this is expensive.

Reasons against Manual Deallocation

- Traditional manual allocation is hard to get right.
  - Sometimes, objects are deallocated too early, then references point to random memory. ("Dangling pointers")
  - Sometimes, objects are deallocated too late or not at all, then heaps become too large ("Space leaks").
  - Estimate: About *half* of the time spent in error detection and fixing for C++ programs is spent on storage related bugs.
  - Popular (but expensive!) tools to detect such problems: Purify, or CenterLine.
- Manual deallocation seriously restricts API's.
  - It is not possible to define general values which are allocated on the heap, as one always needs to keep a handle for deallocation.
  - Compare for example strings in C/C++ with strings in Java.
- Manual deallocation is unsafe, hence not permitted

Garbage Collection

- Garbage collection is a method to reclaim unused storage automatic ally.
- Question: When is storage unused?
- Answer:
  - An *access path* is a sequence of references starting in a global reference or in a reference on some process stack, possibly followed by some field or array element selections.
  - An object is *reachable* if there is an *access path* leading to it.
  - The storage of all objects that are not reachable can be reused.
- Three main algorithms for GC:
  - reference counting
  - mark & sweep
  - copying collectors

Reference Counting

- Idea: Each heap object keeps a field `refcnt` in which the number of references to the object is kept.
- When the object is created, the count is set to 1.
- For each assignment `x = y`, the following code is executed:

```
if (x != null) x->refcnt--;
if (y != null) y->refcnt++;
```

- Each time a pointer is deallocated from the stack, we also decrement its `refcnt` (unless the pointer is null).
- Objects whose `refcnt` drops to 0 are reclaimed immediately.
- In that case all reference fields in a reclaimed objects get their `refcnt` decremented (this might lead to further objects being reclaimed).

Advantages of Reference Counting:

- Easy to implement
- Can be implemented by by hand, or by special classes (e.g. in C++: "smart pointers").
- Storage is reclaimed immediately.
- Hence, better locality of allocated heap data.

Disadvantages of Reference Counting:

- Additional overhead for simple pointer operations.
- Additional space required in objects.
- Cyclic references cannot be recovered (this is clearly the most serious problem).

Mark and Sweep

- Garbage Collection proceeds in two phases:
- Mark Phase: Starting with the global variables and the stack (the so-called *roots*), follow all access paths and mark every visited object (e.g. by setting a bit in its descriptor).
- Sweep Phase: Go linearly through the heap and recycle any objects that do not have their mark bits set.
- Problem: When marking objects along a path, we need to keep track of where to go afterwards, for instance by recursion or by pushing object addresses on a stack.
- If a path is very long, this leads to large additional space overhead for the stack.
- Solution: Keep track of return addresses in the visited objects themselves, for instance by pointer rotation.

Advantages of Mark and Sweep:

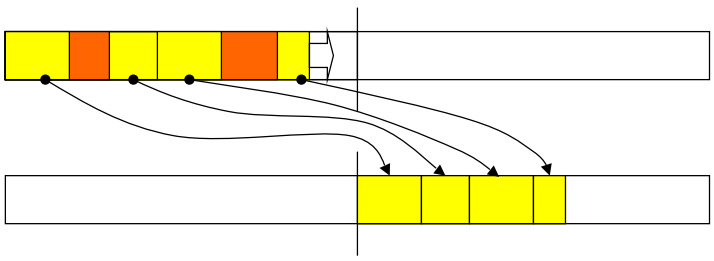
- Can reclaim cyclic structures.
- Fairly easy to implement.
- Low space overhead.

Disadvantages of Mark and Sweep:

- Fragmentation can be a problem; to solve it, an additional copying phase is required.
- Allocation from a freelist can be slow.

Copying Collectors

- Idea: Split the heap in two spaces.
- Objects are allocated linearly in one space.
- When that space is full, we copy all reachable objects to the other space.



Copying Collector Algorithm

```
scan = free = Start of To-Space
for R in Roots: R = copy(R)
while scan < free:
  for f in fields(object at scan):
    scan.f = copy(scan.f)
  scan = scan + size(scan)

copy(P) =
  if P.forward != null:
    return P.forward
  else
    copy size(P) bytes from P to free
    P.forward = free
    free = free + size(P)
    return P.forward
```


Advantages of Copying Collectors

- Can reclaim cyclic structures.
- Very easy to implement.
- Extremely fast allocation: Increment heap pointer and check against limit; can be done inline.
- Compaction is automatic.

Disadvantages of Copying Collectors:

- Double the space overhead, because two spaces are needed.
- Terrible locality, if space is bigger than cache.

Generational Garbage Collection

- Problem: Garbage collecting the whole heap memory makes long pause times.
- Empirical observation: Most objects die young, i.e

  > The longer an object lives, the longer is the likelihood that it will survive the next collection.

- Therefore, benefit of garbage collection is highest for young objects.
- Idea: Try to keep young objects in a smaller space, which is searched more often.
- Then GC takes less time, and yields proportionally more free space.

- Generational GC Scheme: Rather than having two spaces of a copying collector, we have N spaces, where N is the number of generations.
- We start by collecting the youngest generation from space 0 to space 1.
- If this succeeds, GC is finished; we have reclaimed a free space.
- Otherwise, if space 1 becomes full during a collection cycle, we copy its life data to space 2.
- ...and so on.
- Problem: To avoid having to search all of the heap for referenced objects, we need to keep track of references from older generations to younger ones.

Modern Garbage Collectors

- For PC's and servers the currently fastest garbage collectors use a multi-generation copying scheme.
  - The youngest generation's size is chosen small enough to fit in cache.
  - Older (and larger) generations are often subdivided into pieces to increase locality and decrease pause times (=> train algorithm).
- For embedded systems with tighter memory constraints, mark and sweep is more often used.

Current Challenges

Garbage Collection is still an active research area.

Among the topics currently investigated are:

- Concurrent GC. How can a garbage collector run concurrently with a program which allocates data and modifies the heap?
  - This is particularly attractive for multi-threaded processors.
- Real-time GC. How can pause times of a garbage collector be kept small enough for real time.
- Distributed GC. How can we trace references between systems with acceptable overhead?