Runtime Systems

Compiled code + runtime system = executable

The runtime system can include library functions for:

- I/O, for terminal, files, network, ...
- graphics
- math
- reflection
  - examining the static code & dynamic state of the running program itself
- threads, synchronization
- memory management
- system access, e.g. system calls
- ...

Can have more development effort put into the runtime system than into the compiler!

Memory management

Typically support the following operations:

- **allocate** a new (heap) memory block
- **deallocate** a memory block when it's done
  - blocks can be deallocated in any order
  - deallocated blocks will be recycled

Manual memory management:
the programmer decides when memory blocks are done, and explicitly deallocates them

Automatic memory management:
system automatically detects when memory blocks are done, and automatically deallocates them

Challenges:

- must avoid **dangling pointers**
- try to avoid **storage leaks**
- be efficient (time, space, locality, non-fragmentation)
- be convenient, reliable

Manual memory management

Maintain a **free list**: a linked list of deallocated blocks

- allocate: scan the list to find a block that's big enough
  - if no free blocks, allocate large chunk of new memory from OS
  - put any unused part of newly-allocated block back on free list
- deallocate: add to free list
  - store free-list links in the free blocks themselves!

Lots of interesting engineering details:

- allocate blocks using first-fit or best-fit?
- maintain multiple free lists, each for different size(s) of block?
  - when deallocating a block, must be able to determine its size
- combine adjacent free blocks into one larger block, to avoid fragmentation?

See Doug Lea's allocator for an excellent implementation

Regions

A different interface for manual memory management

Support:

- create a new (heap) memory *region*
- allocate a new (heap) memory block *from a region*
- deallocate a *region* and its contained blocks

+ deallocating a region is much faster than deallocating all its blocks individually
+ may be easier to know when all blocks in region are done than when any individual block is done
– must keep entire region allocated as long as any block in the region is still allocated

Best for applications with "phased allocations"

- create a region at the start of a "phase"
- allocate data used only in that phase to the region
- deallocate region when phase completes

(What applications have significant phased allocation?)

Automatic memory management

A.k.a. **garbage collection**

Automatically identify blocks that are done, deallocate them

+ no **dangling pointers**
+ no **storage leaks** (with caveats)
+ much more convenient
– can be less space-efficient, less time-efficient
+ can have faster allocation, better memory locality

General styles:

- reference counting
- tracing
    - mark/sweep
    - copying

Options:

- generational
- incremental, parallel, distributed

Accurate vs. conservative vs. hybrid

Reference counting

For each heap-allocated block,
maintain count of # of pointers to block

- when create block, ref count = 0
- when create new ref to block, increment ref count
- when remove ref to block, decrement ref count
- if ref count goes to zero, then delete block

Can even implement this without compiler support,
e.g. using C++ "smart pointers"

```
class Link { Link next; }
Link foo() {
   Link a = new Link();
   Link b = new Link();
   b.next = new Link();
   a.next = b;
   a = a.next;
   b = null;
   return a.next;
}
```


Evaluation of reference counting

+ local, incremental work
    - good for GC of distributed heaps
    - good for real-time systems
+ little/no language support required

– cannot reclaim cyclic structures
– uses malloc/free back-end ⇒ heap gets fragmented
– high run-time overhead (10-20%)
    - delay processing of ptrs from stack (deferred reference counting)
– space cost of counts
– thread-safety?

BUT: a surprising resurgence in recent research papers,
which fix almost all of these problems

Tracing collectors

Start with a set of **root** pointers

- global vars
- contents of stack & registers

Follow pointers in blocks, transitively,
starting from blocks pointed to by roots

- identifies all **reachable** blocks
- all unreachable blocks are garbage
    - unreachable ⇒ can't be accessed by program
    - (what about the converse?)

A question: how to identify pointers?

- which globals, stack slots, registers hold pointers?
- which slots of heap-allocated blocks hold pointers?

Identifying pointers

"**Accurate**": always know unambiguously where pointers are

Use some subset of the following to do this:

- static type info & compiler support
- run-time tagging scheme
- run-time conventions about where pointers can be

Conservative:
assume anything that *looks like* a pointer *is* a pointer

- consider target block reachable
- \+ supports GC in "uncooperative environments", e.g. C, C++

What "looks" like a pointer?

- most optimistic:
  just aligned pointer-sized memory words whose contents are the addresses of the beginning of allocated blocks
- what about interior pointers?
  off-the-end pointers?
  unaligned pointers?

Misses encoded pointers (e.g. xor'd ptrs), ptrs saved in files, some optimized code, ...

Hybrid: conservative for stack/regs, accurate for globals & heap

Mark/sweep collection

[McCarthy 60]: stop-the-world tracing collector

Stop the application when heap fills

Phase 1: trace reachable blocks, using e.g. depth-first traversal

- set mark bit in each block

Phase 2: sweep through all of memory

- add unmarked blocks to free list
- clear marks of marked blocks, to prepare for next GC

Restart the application

- allocate new (unmarked) blocks using free list

Evaluation of mark/sweep collection

\+ collects cyclic structures
\+ simple to implement
\+ no overhead during program execution

– "embarrassing pause" problem
– not suitable for distributed systems
– need to reserve space for depth-first traversal's stack, or do complicated pointer-reversal tricks
– fragmentation problems of free lists

Mark/compact collection

Like mark/sweep, but replaces sweep phase by compaction

- slide all marked blocks to one end of heap
- all free memory coalesced into one block at other end

\+ no free list needed!
\+ very fast allocation, directly from end of heap
\+ better memory locality, no fragmentation problems
– compaction is slower than sweeping
– redirects pointers $\Rightarrow$ requires accurate pointer info
  - some blocks may need to be "pinned" and not moved, e.g. OS I/O buffers

Challenge: must update all pointers to a moved block

- option 1: double-indirect pointers a.k.a. handles
- option 2:
  - compaction creates table of old→new addrs for moved blocks
  - extra scan patches pointers to moved blocks using table

Copying collection

Divide heap into two equal-sized **semi-spaces**
- application allocates in **from-space**
- **to-space** is empty

When from-space fills, do a GC:
- **visit** blocks referenced by roots
- when visit block from pointer:
  - **copy** block to to-space, redirect pointer to copy
  - leave **forwarding pointer** in from-space version;
    if visit block again, just redirect pointer to to-space copy
- **scan** to-space linearly to visit reachable blocks
  - to-space is queue for breadth-first search of reachable blocks
- when done scanning to-space:
  - **reset** from-space to be empty (akin to region deallocation)
  - **flip**: swap roles of to-space and from-space
- restart application

Evaluation of copying collection

+ collects cyclic structures
+ memory implicitly compacted at each collection
  $\Rightarrow$ no free list needed
  $\Rightarrow$ very fast allocation
  $\Rightarrow$ better memory locality
  $\Rightarrow$ no fragmentation problems
+ no separate table for updating pointers to copied blocks
+ no separate depth-first traversal stack required
+ only visits reachable blocks, ignores unreachable blocks

– requires twice the memory, during GC
  - more memory cost than compaction's table
  - could benefit from OS support, to avoid paging garbage after flip
– "embarrassing pause" problem still
– copying can be slower than marking
– redirects pointers $\Rightarrow$ requires accurate pointer info

Generational GC

Hypothesis: most blocks die soon after allocation
- e.g. closures, cons cells, stack frames, numbers, ...

Idea: concentrate GC effort on young blocks
- divide up heap into 2 or more **generations**
- GC each generation with different frequencies, algorithms

A generational collector

2 generations: **new-space** and **old-space**
- new-space managed using e.g. copying
  - fast allocation, good locality
- old-space managed using e.g. mark/sweep or .../compact
  - good space efficiency

To keep pauses short, make new-space relatively small
- will need frequent, but short, collections

If a block survives many new-space collections,
then **promote** it to old-space
- no more load on new-space collections

If old-space fills, do a full GC of both generations

Roots for generational GC

Must include pointers from old-space to new-space
as roots when collecting new-space

How to find these?

Option 1: scan old-space at each scavenge

Option 2: track pointers from old-space to new-space

Tracking old→new pointers

How to keep track of pointers from old-space to new-space?

- need a data structure to record them
- need a strategy to update the data structure

Option 0: use a purely functional language!

Option 1: keep list of all *locations* in old-space containing such cross-generation pointers (**remembered set**)

- instrument all assignments to update remset (**write barrier**)
  - can implement write barrier in sw or using page-protection hw
  - expensive: duplicates? space?

Option 2: same, but only track *blocks* containing such locations

- lower time and space costs, higher root scanning costs

Option 3: track fixed-size ***cards*** containing such locations

- use a bit-map as remembered set ⇒ very efficient to maintain

(Other options, too)

Evaluation of generation scavenging

+ new-space collections are short: fraction of a second
+ vs. pure copying:
  - less copying of long-lived blocks
  - less (virtual) memory space required
+ vs. pure mark/sweep:
  - faster allocation
  - better memory locality for frequently accessed blocks

– requires write barrier
– still have infrequent full GC's, with embarrassing pauses

Extensions:

- permanent-space as final generation of "eternal" data, e.g. code, constants
- large object space: allocate large objects separately, to avoid frequent copying in new-space
- one new-space per thread, in thread-local memory

Incremental, concurrent, and parallel GC

Avoid long pause times by running collector & application "simultaneously"

- really in parallel, on multiprocessor: **concurrent GC**
- simulate parallelism via time-slicing: **incremental GC**

Main issue: how to synchronize collector & application?

- need read barrier and/or write barrier, in hw and/or sw

A simpler alternative: stop-the-world, then collect in parallel
parallel GC

- exploits multiprocessors for faster GC

+ avoids synchronization costs
– requires efficient multiprocessor stop-the-world