Runtime Systems

Compiled code + runtime system = executable

The runtime system can include library functions for:
- I/O, for console, files, networking, etc.
- graphics libraries, other third-party libraries
- 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

Support
- allocating a new (heap) memory block
- deallocating a memory block when it's done
  - 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

Manual memory management

Typically use "free lists"

Runtime system maintains a linked list of free blocks
- to allocate a new block of memory, 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
- to deallocate a memory block, 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?
- combine adjacent free blocks into one larger block, to avoid fragmentation of memory into lots of little blocks?

See Doug Lea's allocator for an excellent implementation

Regions

A different interface for manual memory management

Support:
- creating a new (heap) memory region
- allocating a new (heap) memory block from a region
- deallocating an entire region when all blocks in that region are done

Deallocating a region is much faster than deallocating all its blocks individually

Less need for complicated reasoning of when individual blocks are done

But 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 "dead", deallocate them
- ensure no **dangling pointers**, no **storage leaks**
- 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"

```
Cons foo() {
   a = new Cons();
   b = new Blob();
   c = bar(a, b);
   return c;
}

Cons bar(Cons x, Blob y) {
   Cons l = x;
   l.head = y;
   t = l.tail;
   return t;
}
```

Evaluation of reference counting

+ local, incremental work
+ little/no language support required
+ local ⇒ feasible for distributed systems

– 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
– no bound on time to reclaim
– 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

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 might a pointer,
& mark target block reachable

+ supports GC in "uncooperative environments", e.g. C, C++

What "looks" like a pointer?

- most optimistic:
  just aligned pointers to beginning of blocks
- what about interior pointers?
  off-the-end pointers?
  unaligned pointers?

Miss encoded pointers (e.g. xor'd ptrs), ptrs in files, ...

A hybrid: conservative for roots, but accurate for heap blocks

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

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
  - may copy more blocks to end of to-space,
    which will be scanned in turn, à la breadth-first search
- when done scanning to-space:
  - reset from-space to be empty
  - **flip**: swap roles of to-space and from-space
- restart application

Evaluation of copying collection

+ collects cyclic structures
+ allocation directly from end of from-space
  - no free list needed ⇒ very fast allocation
+ memory implicitly compacted at each collection
  ⇒ better memory locality
  ⇒ no fragmentation problems
+ no separate traversal stack required
+ only visits reachable blocks, ignores unreachable blocks

– requires twice the (virtual) memory,
  physical memory sloshes back and forth
  - could benefit from OS support

– “embarrassing pause” problem still
– copying can be slower than marking
– redirects pointers ⇒ 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 copying
- old-space managed using mark/sweep

To keep pauses low, 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

- instrument all assignments to update list (a **write barrier**)
  - can implement write barrier in sw or using page-protected 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 "list" ⇒ very efficient to maintain

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

+ vs. pure mark/sweep:

- faster allocation
- better memory locality

– requires write barrier

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

Incremental & parallel GC

Avoid long pause times by running collector & mutator in parallel

- really concurrent: parallel GC
- simulate parallelism via time-slicing: incremental GC

Main issue: how to synchronize collector & application?

- need read barrier and/or write barrier

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

- exploits multiprocessors for faster GC
- avoids synchronization costs