CSE 413 19wi Memory Management/Garbage Collection Notes

Memory categories / lifetimes

- Static – created when program loaded, lifetime is program execution
- Automatic – created on function call, lifetime is function activation (typically stack allocated since function calls are LIFO, but need to modify if closures can capture pointers to local environments – languages that use closures extensively sometimes allocate all function variables on the heap)
- Dynamic – created on demand, lifetime until no longer needed (heap)

Manual memory management: malloc/free (C); new/delete (C++), others

- Tight control over memory
- Error prone – who is responsible for freeing what
  - Memory leaks – storage allocated but not released when done
  - Dangling pointers – storage is released, but existing pointer to that address reused later

Automatic strategies

Reference counting

- Idea: associate a count with each piece of dynamic data: how many pointers (references) exist pointing to this data
  - Increment when new pointer value is created
  - Decrement when pointer changed or deleted
    - If reference count decremented to 0, delete object
- Example: manipulating reference counts on p=q assignment
- Pros: fairly simple to implement; precise discovery of when an object is free
- Cons:
  - Expensive relative to cheap pointer operations
  - Fails in the presence of cycles
    - Partial workaround: weak pointers/references. Requires programming discipline to avoid memory leaks or accidental deallocation
- But useful for resource allocation like file systems where overhead is low compared to other operations and when we have a guarantee of no cycles

Automatic garbage collection

- Basic idea
    - Mark all memory that is currently in use
    - Reclaim all memory previously allocated that is no longer in use
- Key concept: *reachable* (live) data:
    - *Root set*: all known static (global) and dynamic (local) variables. Everything they point to is reachable
        - Root set includes things like registers and anonymous compiler temporaries that hold a copy of a pointer
    - *Transitive closure*: if an object is referenced by some reachable object then it too is reachable

Classic mark/sweep garbage collection

- Associate a "mark bit" with each heap object
- When each new object is allocated (new, cons, etc.) ensure mark bit is 0.

Mark/sweep GC pseudo code:

Precondition: all mark bits on all heap objects are 0

Initialize worklist to empty. Every item on the worklist is an object that (a) is reachable and has had its mark bit set to 1 and (b) has not been examined to see what other objects it references

```
Gc() = mark_heap(); sweep();

mark_heap() =   // sketch – would need to check null ptrs in real gc, etc.
    for each variable r in the root set
        obj = *r
        if  mark_bit(obj) = 0
            mark_bit(obj) = 1
            add obj to worklist
    while worklist is not empty
        remove next object p from worklist
        for each reference variable (pointer) r in p
            obj = *r
            if  mark_bit(obj) = 0
                mark_bit(obj) = 1
                add obj to worklist
```

```
sweep() =
    for each heap object
        if mark bit is 0 then free object (add to free list)
        else set mark bit to 0 (reset for next gc)
```

postcondition: all unused heap objects have been freed and all mark bits are 0

example: show gc after

```
(define x '(a b))
(define n (length (append x x)))
(define y (cons 'c x))
```

Variations – lots

Compacting/copying collectors: idea:

- divide heap into two halves: old and new
- Allocate objects from old
- When old is used up, copy all reachable (live objects) to new
    - Need to put forwarding pointers in place from old objects that are live to new copies, then update all discovered pointers to point to new copies as other objects are collected/moved
- After all live objects are copied, swap old and new – previous old is now free space to be used to store copies of live data on next collection

Advantages

- Heap allocation is simple and very cheap – no free list needed, just keep a "next free" pointer in half where objects are being allocated
- Keeps live heap objects contiguous over time; avoids heap fragmentation and minimizes number of live pages

Disadvantages

- Hard to use all memory efficiently. Either half is not used, or, if we try to take advantage of virtual memory, we need to be very careful about thrashing (bad vm performance) because of excessive live pages, during gc especially

Generational collectors

- Idea: most program allocate lots of short-lived objects, so
- Biggest payoff is normally to run GC only on portion of memory with recently allocated objects

- Divide heap into small part for new objects – the "nursery" – and larger part for long-lived objects. GC nursery frequently, entire heap rarely. If a new object survives several GCs in the nursery, promote it to the longer-lived part of the heap

Concurrent collectors

- "Stop the world" isn't a great strategy for interactive or real-time computation. Want to allow GC and program ("mutator") to run concurrently, often with GC running in background cleaning up memory when time is available.
- But: much more complex, hard to get right (proofs needed, hard or impossible to reproduce timing-related bugs, can't debug into correctness), etc.

Real world: industrial-strength GCs these days use a mix of various strategies, particularly generational and concurrent collectors.