CSE 413 14au Memory management/Garbage Collection Notes

Memory categories / lifetimes

- Static – created when program loaded, lifetime is 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)
- Dynamic – created on demand, lifetime until no longer needed (heap)

Manual memory management: malloc/free; new/delete

- 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 accidental deallocations or memory leaks
- But useful for resource allocation like file systems where overhead is low compared to other operations and 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* data:

- Root set: all known static (global) and dynamic (local) variables. Everything they point to is reachable
- 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 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() =   // punted – would need to check null ptrs in real impl., 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 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
```

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 out of old
- When old is used up, copy all reachable (live objects) to new
    - Need to put forwarding pointers in place from old objects to new copies, and update all discovered pointers to point to new copies
- After all live objects copied, swap old and new – previous old is now free space to be used on next collection

Advantages

- Heap allocation is simple – 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

Generational collectors

- Idea: most program allocate lots of short-lived objects, so
- Biggest payoff is normally to GC only 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, bug prone, etc.

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