

Garbage Collection

1

What is garbage collection?

- Automatic recycling of allocated heap memory that can not be used again
 - i.e., is garbage
 - i.e., is not reachable (from roots)
- Convenient
- Avoids memory leaks (usually)
- Required for type safety

2

Java can leak memory

```
// Find the error
class Stack {
    Object a[] = new Object[10];
    int top = -1;
    Object pop() {
        if (top == -1)
            throw new NoSuchElementException();
        return a[top--];
    }
    void push(Object o) {
        if (top+1 == a.length)
            throw new StackOverflowException();
        a[++top] = o;
    }
}
```

3

Reachable != will be used again

```
// Error fixed in green
class Stack {
    Object a[] = new Object[10];
    int top = -1;
    Object pop() {
        if (top == -1)
            throw new NoSuchElementException();
        Object r = a[top];
        a[top--] = null;
        return r;
    }
    void push(Object o) {
        if (top+1 == a.length)
            throw new StackOverflowException();
        a[++top] = o;
    }
}
```

4

Finding pointers can be hard

- What are root pointers?
 - Pointers that can be accessed by program, even though no pointers point to them
 - For Java:
 - all local and stack variables of all threads
 - all loaded classes (class unloading?)
 - static variables of loaded classes
 - What is the type of a Java stack variable?
 - » Need stack map, from PC to local/stack variable types
 - » Can also take into account dead references

5

Finding pointers in objects can be hard

- Type unsafe languages are a nightmare
 - variant records (unions)
 - pointers into middle of objects
 - various bit-twiddling tricks
 - Even in typesafe languages
 - need run-time access to type information for objects

6

Reference counting

- Different approach to garbage collection
- For each object, keep track of number of references to it
- Cost to adjust counts (some optimization possible)
 - {Object tmp = a; a = b; b = tmp}
 - generates:
a.refCount++; tmp = a;
b.refCount++; a.refCount--; a = b;
tmp.refCount++; b.refCount--; b = tmp;
tmp.refCount--;
- Can't reclaim circular structures
- Use only in special circumstances
 - not generally recommended

7

Mark and Sweep [McCarthy 1960]

- From roots
 - Roots are pointers known to be accessible
 - e.g., registers, stack
- perform a depth-first search to mark live nodes
- Sweep through all memory
 - if node is unmarked, it is garbage, put on free list
 - if marked, is in use, unmark
 - to prepare it for next garbage collection

8

Problems with Mark and Sweep

- Many of these are problems with other garbage collection schemes
 - Stack required for DFS
 - could be big
 - Finding roots
 - Finding pointers
 - Easy for CONS cells
 - Stops the world

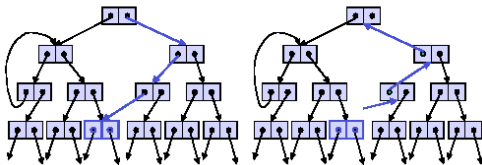
9

Pointer Reversal

- In depth first search, need stack as big as depth of data structure
 - for a long list, depth == length
- Often, garbage is being collected when we are short on space
- Can avoid storing DFS stack separately using pointer reversal
 - Efficient way to implement mark and sweep

10

Doing pointer reversal



11

Is pointer reversal worthwhile?

- Requires additional bits
 - $\log f$ bits, where f is number of pointer fields
- Requires additional visits and modifications of each node
- Generally not worth while
 - Instead, can do tail-call optimization for last field from object
 - when following last field out of an object, don't put object on stack for a return visit
 - helps a little

12

Variations on Mark-and-sweep

- Mark and compact
 - In sweep phase, compact live cells
- Mark and incremental sweep
 - Only mark phase needs to halt the world
 - When allocating, sweep just enough to satisfy allocation
- Snapshot mark-and-sweep
 - Set of unreachable objects doesn't shrink
 - quickly make copy of entire address space
 - use copy-on-write page mapping
 - in a separate process, perform mark and sweep on copy
 - when done, give list of garbage to main process as list of free cells

13

Copying Collectors

- Mark and sweep doesn't move objects
- Copying collectors:
 - traverse the live objects
 - copying them to new space
 - after all objects moved, swap new and old space
- Advantages
 - doesn't touch garbage
 - makes free space contiguous; allocation very cheap
- Disadvantages
 - "wastes" half of memory
 - requires accurate pointer identification
- Allows for locality optimization

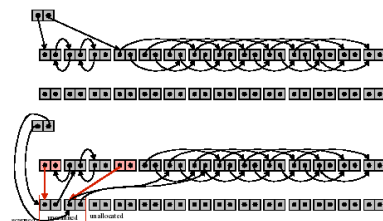
14

Cheney's algorithm

- Algorithm given for Cons cells
- As object in from space seen
 - copy object to unscanned to-space
 - store forwarding point in from-space version of object
- Four kinds of objects:
 - from-space, unscanned - pointers to from-space
 - from-space, scanned - 1 pointer to to-space replacement
 - to-space, unscanned - pointers to from-space
 - to-space, scanned - pointers to to-space

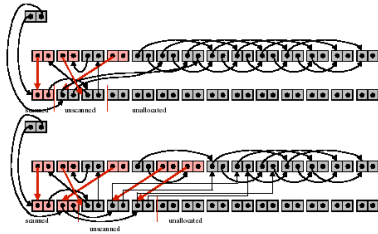
15

Cheney example



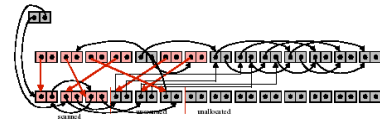
16

example, continued



17

Example, continued



18

Forwarding pointers

```
forward(p) {  
  // compute new value for pointer  
  if p is not a pointer return p  
  // check to see if cell already forwarded  
  if M[p] points to-space then return M[p]  
  // forward cell pointed to by p  
  M[next] = M[p]  
  M[next+1] = M[p+1]  
  M[p] = next  
  next += 2;  
  return M[p]  
}
```

19

Cheney's Garbage Collection

```
GarbageCollect() {  
  scan = next = beginning of to-space  
  foreach root r do  
    r = forward(r)  
  while scan < next do {  
    M[scan] = forward(M[scan])  
    scan++  
  }  
}
```

20

Cheney's algorithm

- A copying garbage collection
- Requires no auxiliary storage (e.g., stack for DFS search)
- Linear time, low constant factor
- Easy to implement
- Forms basis of many later algorithms
- However, does unpleasant things to locality

21

Asymptotic cost of garbage collection

- Assume A objects accessible at any one time
- Assume memory can hold M objects
- Mark and sweep:
 - $c_1 A$ for mark, $c_2 M$ for sweep
 - Can allocate $M-A$ objects between GCs
 - Cost per allocation: $(c_1 A + c_2 M) / (M-A)$
 - Cost is c_2 as M goes to infinity
- Copying collection
 - $c_3 A$ for copy
 - Cost per allocation $c_3 A / (M-A)$
 - Cost is 0 as M goes to infinity

22

Weak pointers

- An object only pointed to by weak pointers is garbage
 - example use: pointers from cache
- Set to null when object is GC'd
 - perhaps fire off other code

23

Finalization/destruction

- Not guaranteed to be prompt
- If a garbage A contains a ref to a B that is also garbage
 - in what order are the finalizers run?
 - finalizing A might invoke code on B
 - but if B was finalized, might not be valid
- Finalization might resurrect an object
 - hard to detect
 - In Java, finalizers are only run once
- Objects with finalizers might not be collected until GC'd a second time

24

Conservative collection

- Doesn't rely on being able to exactly identify all pointers
 - if it looks like a pointer, treat it like a pointer
 - can't move objects
 - can't accidentally change something that isn't a pointer
- Have to figure out set of rules for what a pointer might be
 - pointer into middle of object?
 - non-aligned pointers
- Possible to deceive a conservative collector
- Initial Java JDK GC was conservative
 - due to difficulty of identifying pointers in stack/local variables

25

Language differences

- A lot of this depends on the memory allocation behavior
- The programming language has a big influence:
 - In some languages, there is no stack; everything is heap allocated
 - Some languages allocate memory at almost every statement
 - In purely functional languages, you never mutate existing heap objects
 - In mostly functional languages, rarely mutate
- C does least amount of allocation/computation
- C++ does much more (10 times more?)
- GC overheads in an interpreted language don't seem as high

Reducing Pause Times

- Asymptotic complexity pretty good
 - if you have sufficiently more memory than is live at any time
 - And don't swap memory out to disk
- But pause times can be unacceptable
 - particularly for real-time systems

27

Incremental/Concurrent Collection

- Mutator/Application and Collector run in parallel
- Perhaps running in parallel, perhaps context switching
- Done to reduce garbage collection pause time
- We'll come back to these if we have time

28

Generational garbage collection

- Many objects die young
 - 50-90% of Common Lisp objects die before 10K bytes
 - 75-95% of Haskell objects die before 10K bytes
 - 99% of Cedar objects die before 731K bytes
 - >50% of C objects die before 10K bytes
 - >90% of C objects die before 32K bytes
- Objects that do not die young often live long
 - Objects don't have a half-life

29

Hand-waved situation

- Consider using a copying collector
- With the following distribution of objects
 - 40% have been around for a long time
 - 5% allocated recently, but before most recent GC
 - 50% allocated since most recent GC
 - Mostly garbage
- If we use a copying collector, and the 40% that has been around for a long die doesn't change:
 - we'll keep moving it, and moving it, and moving it

30

Generational Collection

- Could we sweep just the 50% allocated since the last GC?
- Only 1-3% likely to have survived, will be very easy to sweep
 - only pay cost for live objects
 - Will not have to move old objects
- Benefits
 - Diminish pause time
 - Decreased overall GC cost
 - Effect on locality?

31

Issues?

- How do we sweep just the new objects?
- How do we find roots of new objects?
 - Including pointers from old objects
- What happens to an object after we sweep the new objects
 - Mark them as old objects?
 - Even the object that was allocated moments before the GC?
- Nepotism by old objects
 - If a new object is pointed to by a garbage old object, it won't be collected

32

Old->new pointers

- Pointers to new objects contained in old objects
- Created two ways:
 - mutating old objects
 - promoting one object but not the object it points to
- Don't care about mutated old-old pointers
 - until we garbage collect old objects
- We assume all old objects are live, and any new object pointed to by an old object is therefore live
 - a conservative approximation

33

Recording old->new pointers

- Number of hardware-software solutions
- Some depend on special hardware
 - e.g., automatic forwarding
 - Lisp machines are dead
 - I won't talk about them
- Remembered sets
- Page marking with VM support

34

Remembered sets

- Every time you store a reference to a new object into an old object
 - append the old object to a list
- Avoid duplicate entries
- Compiler optimizations
 - Don't record stores into newly allocated objects

35

Page marking with VM support

- Use VM dirty bits
 - a little tricky
 - pages with dirty bits != pages that might have have old->new pointers
- Write protect pages containing old objects
 - Set bit for pages as you get a page fault, unprotect page
- In both of these approaches, scan entirety of dirty pages
- Might consider using write protected to produce remembered set
- Maintain write protection even after page fault
 - But probably won't work
 - cost of page fault too high

36

Card Marking

- User-level version of page marking
- Memory is divided into cards
 - how big is a card?
- Each time a pointer in a card is changed, mark the card
 - 3-6 instructions
 - Scan all marked cards

37

Combining Card marking and remembered sets

- With multi-generational GC, a pointer might remain an old->young pointer after a GC
 - can't erase card mark
 - have to scan entire card each sweep
- Idea: when you scan a card, record old->young pointers
 - reset card mark

38

Tenure policies

- When does an object become tenured (considered old)?
- After it survives one collection?
 - Easy to implement, simple memory model
 - But objects very recently allocated will be promoted even though they are likely to die soon
- Not too bad if short lifetimes much less than size of heap
 - falsely promoting 10K isn't too bad if you have 10 Megabytes of memory
- Promote objects after they survive two collections
 - Gets almost all of the easy cases
- Use something more sophisticated
 - Perhaps based on number of objects that actually survived⁹⁹

Heap organization

- Appel: spaces for old objects, new objects, reserved Aging space
 - Creation space
 - Two aging spaces (only one used at a time, except during GC)
 - old space
- During a sweep of creation space + aging space
 - objects in creation space move to new aging space
 - objects in aging space promoted

40

Train collection

- Old generation is a set of trains
 - each train is a set of cars
- Collect one car at a time
 - only scan that car and pointers to that car
- If an object has pointers from a later train (or root)
 - move to last train (or start a new train)
- If object has pointers only from same train
 - move to end of current train

41

Parallel collection

- So, you just bought a 64 processor SMP with 40 Gigabytes of RAM
- Run a single JVM with a heap size of 37 Gigabytes
- 64 processors perform a lot of allocations
- GC algorithm runs on only a single processor
 - in earlier versions of Sun's VM
 - experimental versions of parallel collectors are available in 1.4

42