CMSC 433 – Programming Language Technologies and Paradigms
Spring 2005

Memory Management
May 10, 2005

Memory Management in Java

- Local variables live on the stack
  - Allocated at method invocation time
  - Deallocated when method returns
- Other data lives on the heap
  - Memory is allocated with new
  - But never explicitly deallocated
    - Java uses automatic memory management

19

Memory Mgmt and the JVM

- The JVM Specification doesn't say how to manage the heap
- Simplest valid memory management strategy: never delete any objects
  - Not such a bad idea in some circumstances (when?)

20

Garbage Collection (GC)

- At any point during execution, can divide the objects in the heap into two classes:
  - *Live* objects will be used later
  - *Dead* objects will never be used again
    - They are garbage
- Idea: Can reuse memory from dead objects

21

Many GC Techniques

- We can't know for sure which objects are really live or dead
  - Undecidable, like solving the halting problem
- Thus we need to make an approximation
  - OK if we decide something is live when it's not
  - But we'd better not deallocate an object that will be used later on

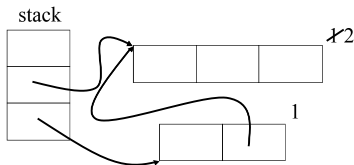
22

Reference Counting

- Old technique (1960)
- Each object has count of number of pointers to it from other objects and from the stack
  - When count reaches 0, object can be deallocated

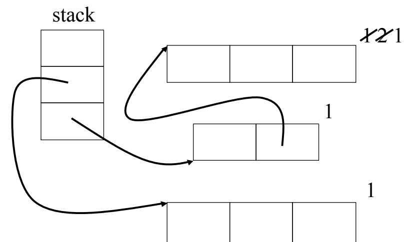
23

Reference Counting Example



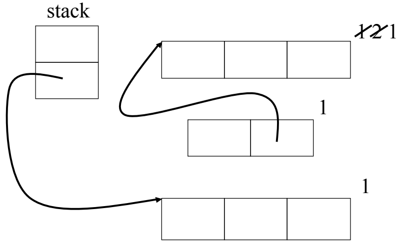
24

Reference Counting Example



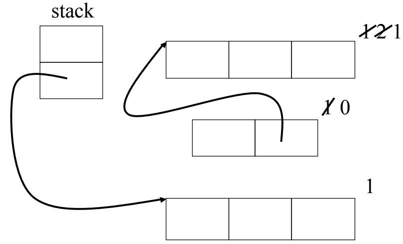
25

Reference Counting Example



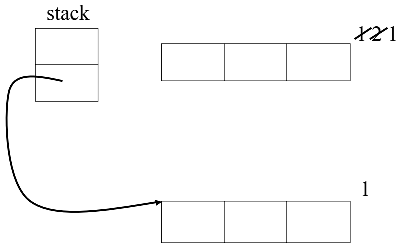
26

Reference Counting Example



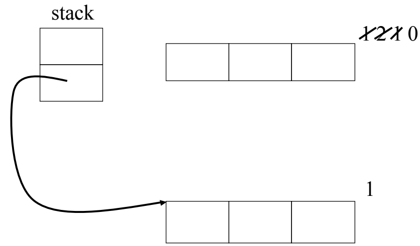
27

Reference Counting Example



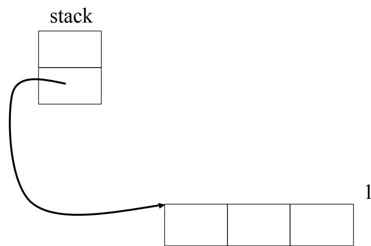
28

Reference Counting Example



29

Reference Counting Example



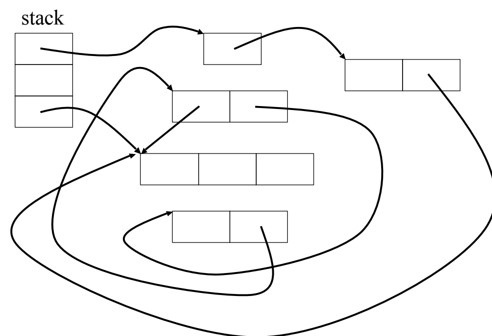
Tradeoffs with Ref Counting

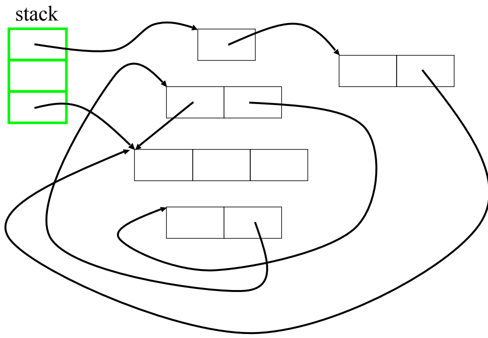
- Advantage: Incremental technique
  - Small amount of work per memory write
  - With more effort, can bound running time
    - Useful for real-time systems
- Problem: Data on cycles can't be collected
  - Counts never go to zero

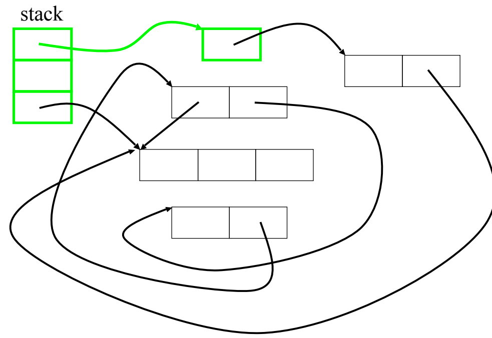
Mark and Sweep GC

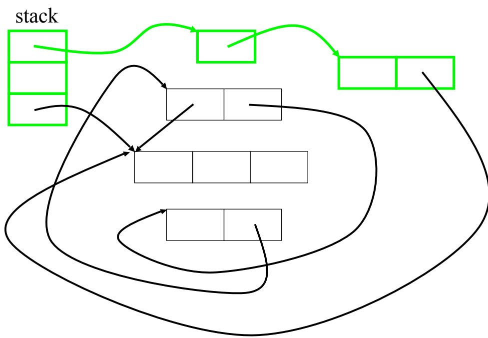
- Idea: Only objects reachable from stack could possibly be live
  - Every so often, stop the world and do GC:
    - Mark all objects on stack as live
    - Until no more reachable objects,
      - Mark object reachable from live object as live
    - Deallocate any non-reachable objects
- This is a *tracing* garbage collector

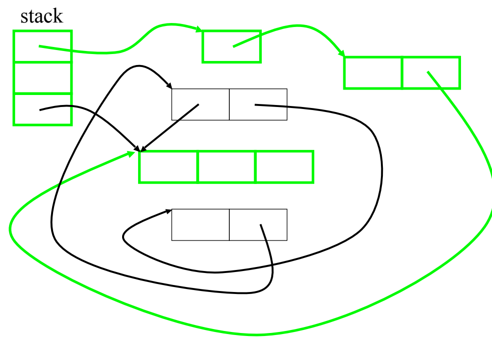
Mark and Sweep Example



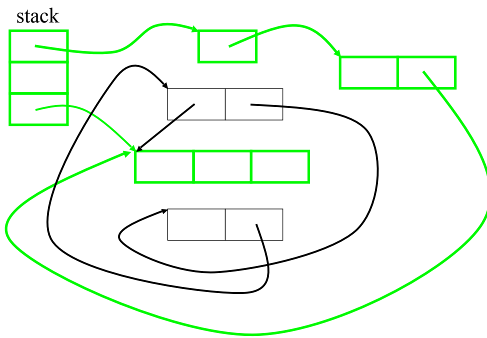

Mark and Sweep Example
stack
34


Mark and Sweep Example
stack
35


Mark and Sweep Example
stack
36

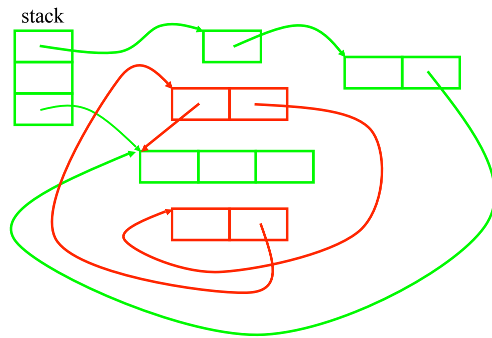

Mark and Sweep Example
stack
37

Mark and Sweep Example



38

Mark and Sweep Example



39

Tradeoffs with Mark and Sweep

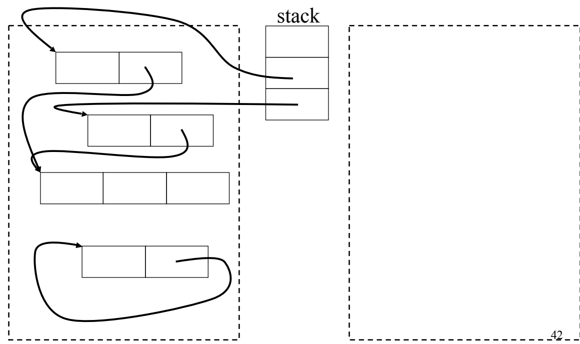
- Pros:
  - No problem with cycles
  - Memory writes have no cost
- Cons:
  - Fragmentation
  - Cost proportional to heap size
    - Sweep phase needs to traverse whole heap

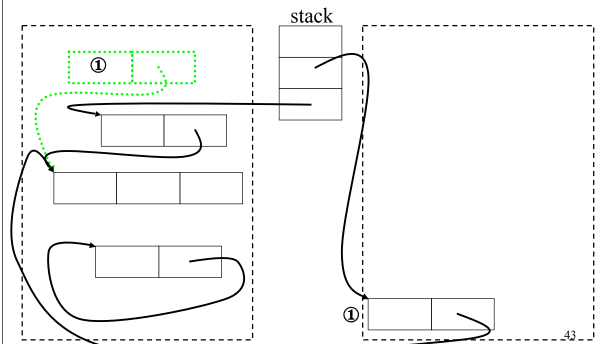
40

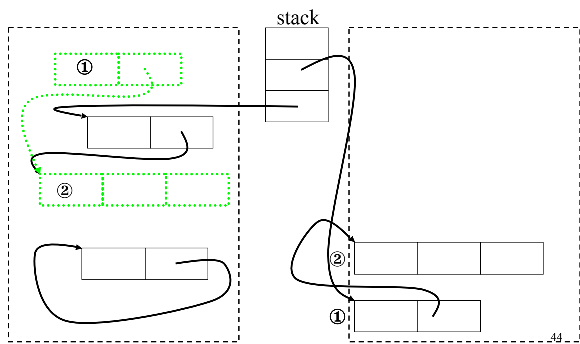
Stop and Copy GC

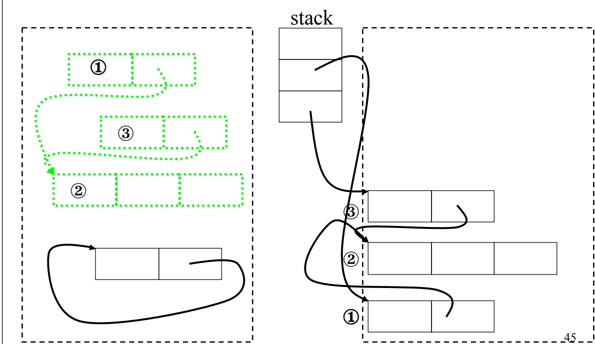
- Like mark and sweep, but only touches live objects
  - Divide heap into two equal parts (semispaces)
  - Only one semispace active at a time
  - At GC time, flip semispaces
    - Trace the live data starting from the stack
    - Copy live data into other semispace
    - Declare everything in current semispace dead; switch to other semispace

41


Stop and Copy Example
stack
42


Stop and Copy Example
stack
①
①
43


Stop and Copy Example
stack
①
②
②
①
44


Stop and Copy Example
stack
①
③
②
③
②
①
45

Stop and Copy Tradeoffs

- Pros:
  - Only touches live data
  - No fragmentation; automatically compacts
    - Will probably increase locality
- Cons:
  - Requires twice the memory space
  - Like mark and sweep, need to stop the world

46

Improving Stop and Copy

- Long lived objects get copied over and over
  - Idea: Have more than one semispace, divide into generations
    - Objects that survive copying passes get pushed into older generations
    - Older generations collected less often
- One popular setup
  - Generational stop and copy

47

More Issues in GC (cont'd)

- Stopping is world is a big hit
  - Unpredictable performance
    - Bad for real-time systems
  - Need to stop all threads
    - Without a much more sophisticated GC
- One-size fits all solution
  - Sometimes, GC just gets in the way
  - But correctness comes first

48

What Does GC Mean to You?

- Ideally, nothing
  - It should make your life easier
  - And shouldn't affect performance too much
    - May even give better performance than you'd have with explicit deallocation
- If GC becomes a problem, hard to solve
  - Not much control over it

49

Increasing Memory Performance

- Don't allocate as much memory
  - Less work for your application
  - Less work for the garbage collector
  - Should improve performance
    - (Why only "should"?)
- Don't hold on to references
  - Null out pointers in data structures
  - Or use weak references

50

Find the Memory Leak

```
class Stack {
  private Object[] stack;
  private int index;
  public Stack(int size) {
    stack = new Object[size];
  }
  public void push(Object o) {
    stack[index++] = o;
  }
  public void pop() {
    return stack[index--];
  }
}
```

– From Haggar, Garbage Collection and the Java Platform Memory Model

51

Bad Ideas (Usually)

- Calling System.gc()
  - This is probably a bad idea
  - You have no idea what the GC will do
  - And it will take a while
- Managing memory yourself
  - Object pools, free lists, object recycling
  - GC's have been heavily tuned to be efficient

52

java.lang.ref

- Package that lets you interact with GC
  - If there's a *strong* (normal) reference to an object, GC will consider it live
  - Can use java.lang.ref to make references to object that "don't count"

```
public abstract class Reference {
    void clear();
    public Object get();
    ...
}
```

53

SoftReference

- Constructor SoftReference(Object o)
  - Make a soft reference to object o
  - Access using get() method
- GC *may* free object if
  - Only soft, weak, or phantom refs to it
    - No strong refs
- GC invokes clear() on SoftRef if freed

54

WeakReference

- Constructor WeakReference(Object o)
- GC *will* free object if
  - Only weak or phantom refs to it
    - No strong or soft refs
- GC invokes clear() on WeakRef if freed

55

PhantomReference

- We won't talk about these
  - Need to be used with ReferenceQueues
  - ...which we also haven't discussed

56

Uses of Soft and Weak Refs

- Use soft references for caches
  - Data can be recomputed/reconstructed
  - GC flushes cache (clears soft refs) if memory full
- Use weak references for mappings
  - E.g., object reuse (if you're going to do it...)
  - GC frees object once strong refs are gone

57

Caveat for Weak References

- Which is correct?

```
WeakReference wr;              WeakReference wr;
obj = wr.get();                obj = wr.get();
if (obj == null) {             if (obj == null) {
  obj = new A();                 wr = new WeakRef(new A());
  wr = new WeakRef(obj);         obj = wr.get();
}                               }
```

- From Haggar, Garbage Collection and the Java Platform Memory Model

58

Caveat for Weak Refs (cont'd)

- Left side is correct
  - Consider `new WeakRef(new A())`
  - Object A allocated
  - Put into WeakReference
  - Suppose GC happens right after
  - Then A may be deallocated before `wr.get()`

59

Dealing with GC Problems

- Best idea: Measure where your problems are coming from
  - Find a Java implementation that has lots of heap profiling information
  - Understand thoroughly what's happening
  - Find solution
- No easy solution
  - But don't try to optimize too early

60