

## Lecture 13: Garbage Collection

COS 320

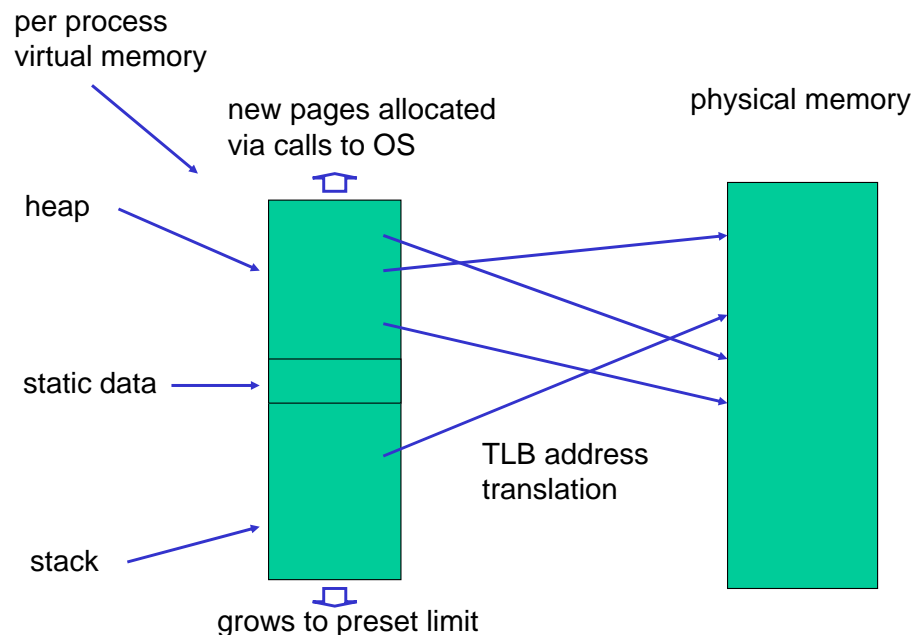
Compiling Techniques

Princeton University  
Spring 2007

Prof. David August

- Every modern programming language allows programmers to allocate new storage dynamically
  - New records, arrays, tuples, objects, closures, etc.
- Every modern language needs facilities for reclaiming and recycling the storage used by programs
- It's usually the most complex aspect of the run-time system for any modern language (Java, ML, Lisp, Scheme, Modula, ...)

### Memory Layout



### GC

- What is garbage?
  - A value is garbage if it will not be used in any subsequent computation by the program
- Is it easy to determine which objects are garbage?

- What is garbage?
  - A value is garbage if it will not be used in any subsequent computation by the program
- Is it easy to determine which objects are garbage?
  - No. It's undecidable. Eg:
 

```
if long-and-tricky-computation then use v
else don't use v
```

## Explicit Memory Management

User library manages memory; programmer decides when and where to allocate and deallocate

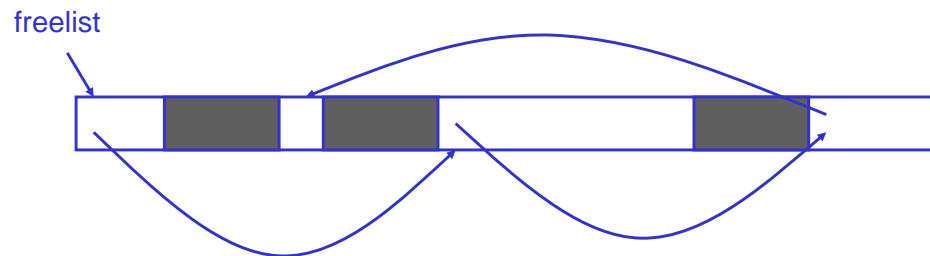
- `void* malloc(long n)`
- `void free(void *addr)`
- Library calls OS for more pages when necessary
- Advantage: people are smart
- Disadvantage: people are dumb and they really don't want to bother with such details if they can avoid it

Since determining which objects are garbage is tricky, people have come up with many different techniques

- It's the programmers problem:
  - Explicit allocation/deallocation
- Reference counting
- Tracing garbage collection
  - Mark-sweep, copying collection
  - Generational GC

## Explicit MM

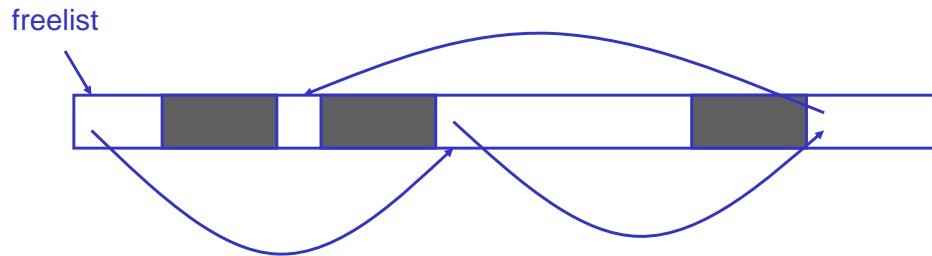
- How does malloc/free work?
  - Blocks of unused memory stored on a freelist
  - malloc: search free list for usable memory block
  - free: put block onto the head of the freelist



## Explicit MM

### Drawbacks

- malloc is not free: we might have to do a search to find a big enough block
- As program runs, the heap fragments leaving many small, unusable pieces



## Automatic MM

Languages with explicit MM are harder to program

- Always worrying about dangling pointers, memory leaks: a huge software engineering burden
- Impossible to develop a secure system, impossible to use these languages in emerging applications involving mobile code
- New languages tend to have automatic MM
  - eg: Microsoft is pouring \$\$\$ into developing safe language technology, including a new research project on dependable operating system construction

## Explicit MM

Solutions:

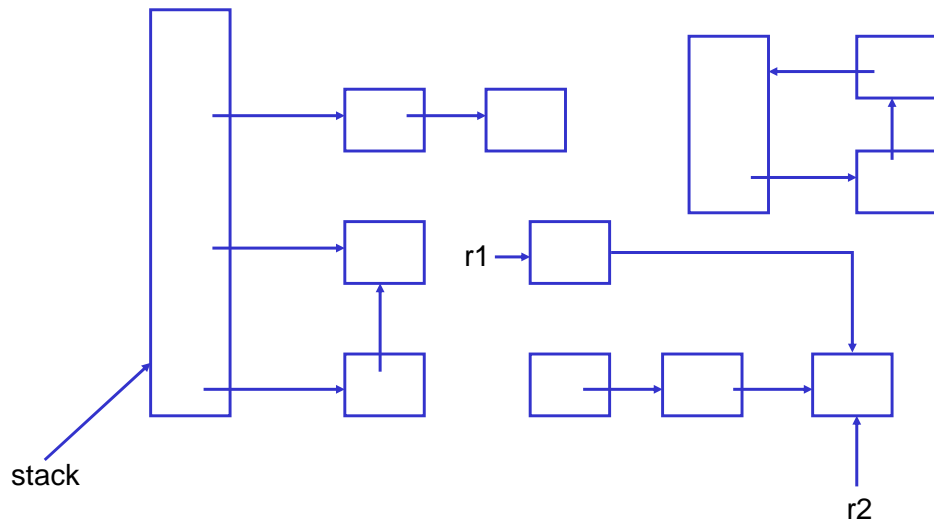
- Use multiple free lists, one for each block size
  - Malloc and free become  $O(1)$
  - But can run out of size 4 blocks, even though there are many size 6 blocks or size 2 blocks!
- Blocks are powers of 2
  - Subdivide blocks to get the right size
  - Adjacent free blocks merged into the next biggest size
  - still possibly 30% wasted space
- Crucial point: there is no magic bullet. Memory management always has a cost. We want to minimize costs and, these days, maximize reliability.

## Automatic MM

Question: how do we decide which objects are garbage?

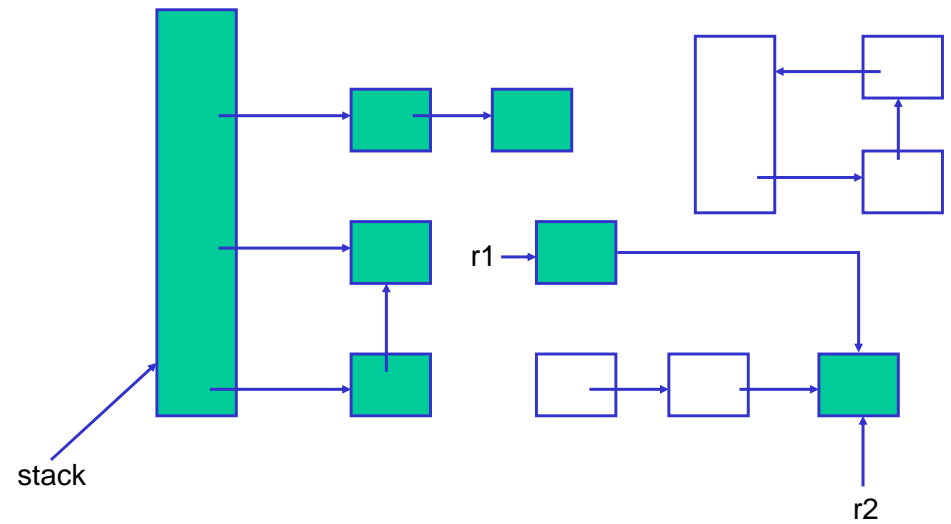
- Can't do it exactly
- Therefore, we conservatively approximate
- Normal solution: an object is garbage when it becomes unreachable from the roots
  - The roots = registers, stack, global static data
  - If there is no path from the roots to an object, it cannot be used later in the computation so we can safely recycle its memory

## Object Graph



- How should we test reachability?

## Object Graph

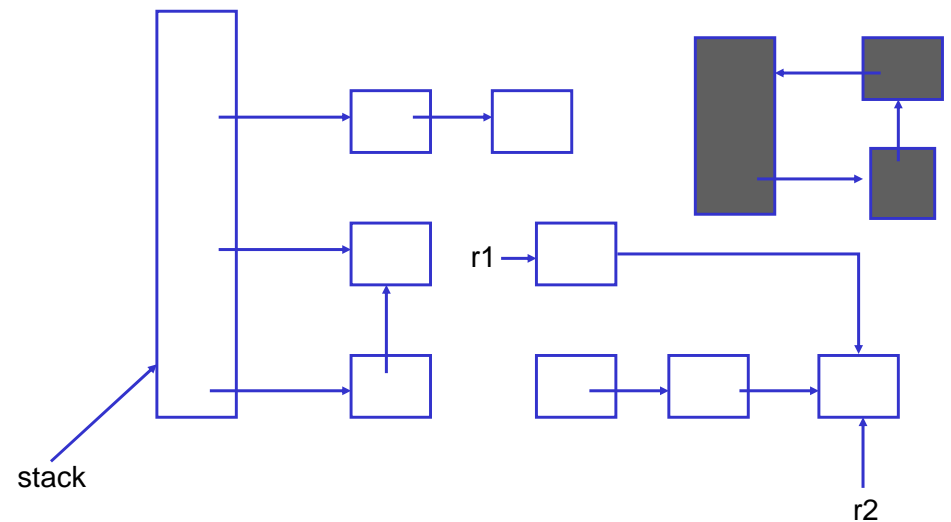


- How should we test reachability?

## Reference Counting

- Keep track of the number of pointers to each object (the reference count).
- When the reference count goes to 0, the object is unreachable garbage

## Object Graph



- Reference counting can't detect cycles

## Reference Counting

In place of a single assignment  $x.f = p$ :

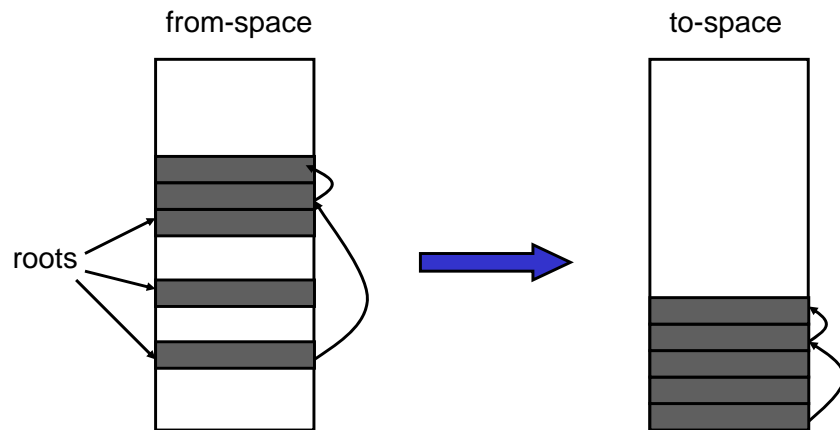
```
z = x.f
z.count = z.count - 1
If z.count = 0 call putOnFreeList(z)
x.f = p
p.count = p.count + 1
```

- Ouch, that hurts performance-wise!
- Dataflow analysis can eliminate some increments and decrements, but many remain
- Reference counting used in some special cases but not usually as the primary GC mechanism in a language implementation

## Copying Collection

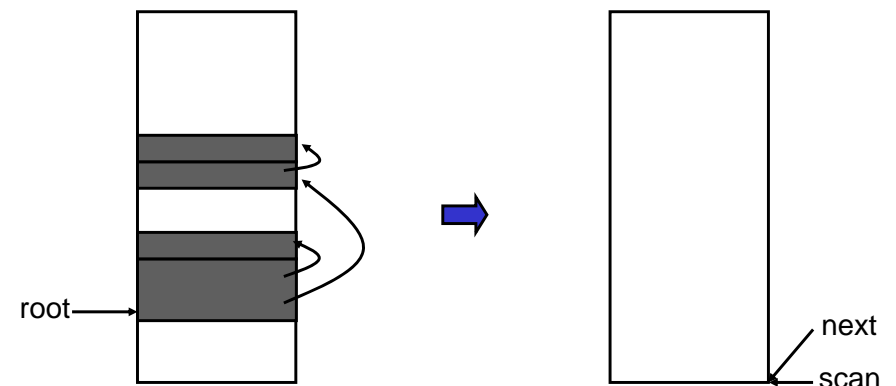
- Basic idea: use 2 heaps
  - One used by program
  - The other unused until GC time
- GC:
  - Start at the roots & traverse the reachable data
  - Copy reachable data from the active heap (from-space) to the other heap (to-space)
  - Dead objects are left behind in from space
  - Heaps switch roles

## Copying Collection



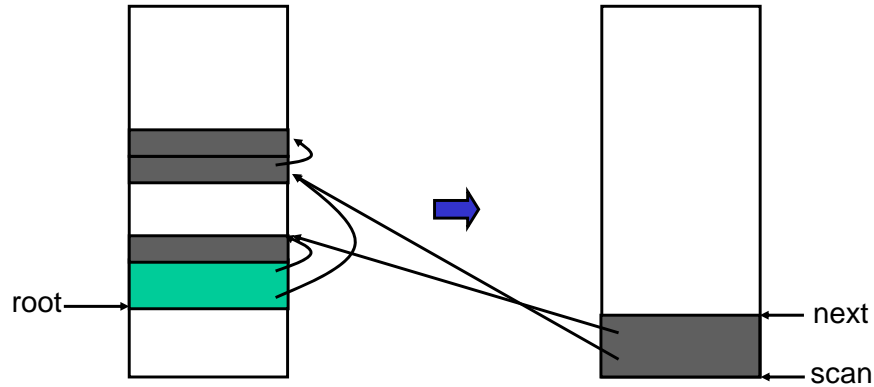
## Copying GC

- Cheny's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space



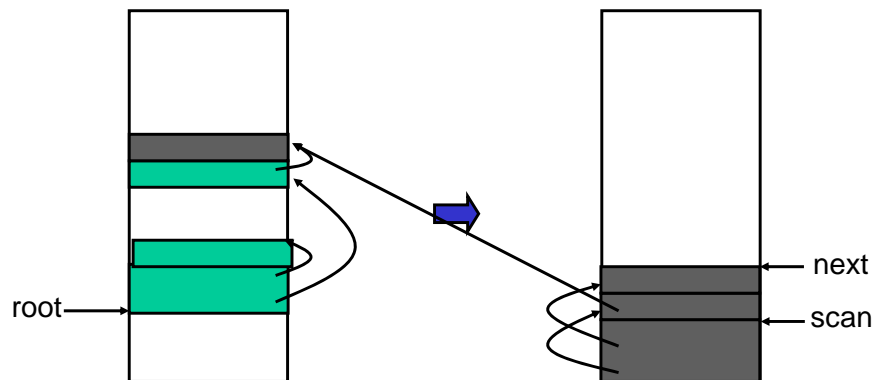
## Copying GC

- Cheny's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space



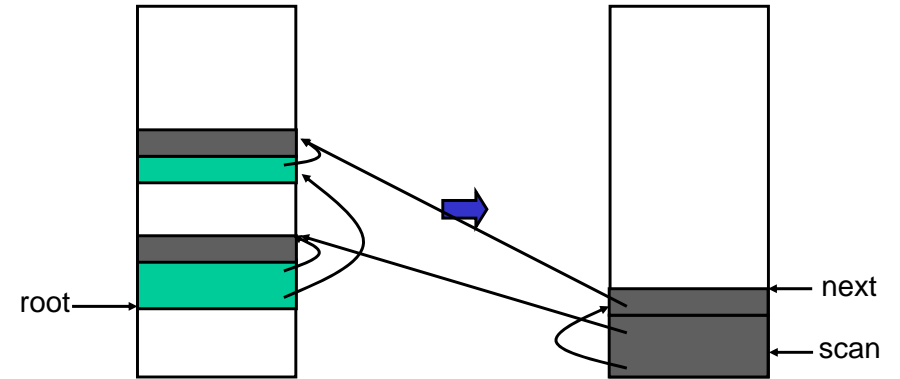
## Copying GC

- Cheny's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space



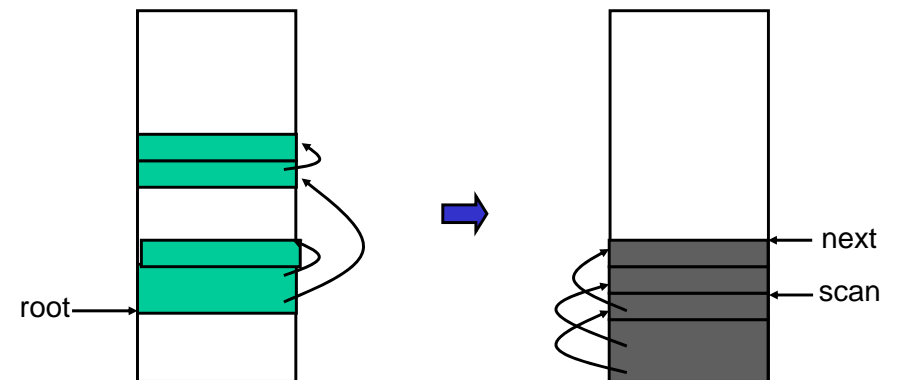
## Copying GC

- Cheny's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space



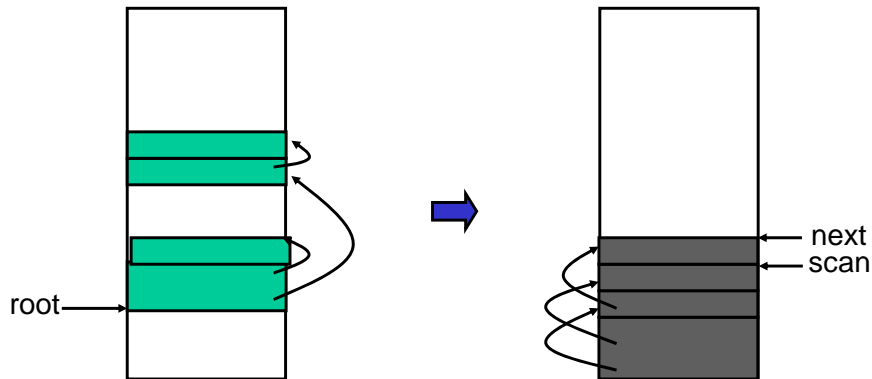
## Copying GC

- Cheny's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space



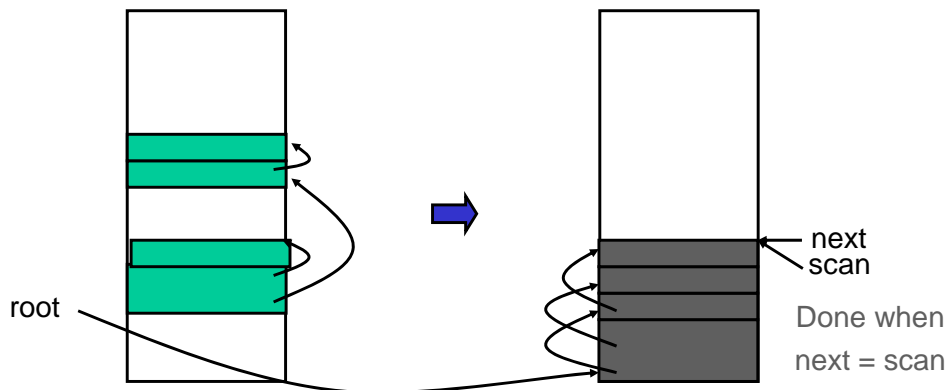
## Copying GC

- Cheney's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space



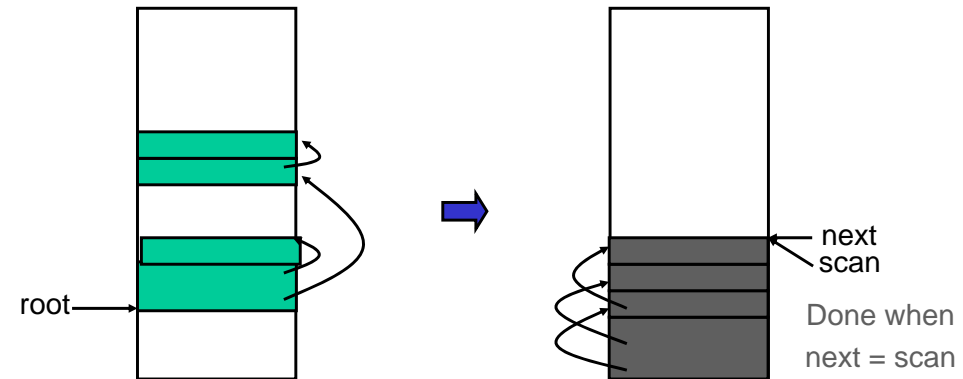
## Copying GC

- Cheney's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space



## Copying GC

- Cheney's algorithm for copying collection
  - Traverse data breadth first, copying objects from from-space to to-space



## Copying GC

- Pros
  - Simple & collects cycles
  - Run-time proportional to # live objects
  - Automatic compaction eliminates fragmentation
  - Fast allocation: pointer increment by object size
- Cons
  - Precise type information required (pointer or not)
    - Tag bits take extra space; normally use header word
  - Twice as much memory used as program requires
    - Usually, we anticipate live data will only be a small fragment of store
    - Allocate until 70% full
    - From-space = 70% heap; to-space = 30%
  - Long GC pauses = bad for interactive, real-time apps

## Baker's Concurrent GC

- GC pauses avoided by doing GC incrementally
- Program only holds pointers to to-space
- On field fetch, if pointer to from-space, copy object and fix pointer
  - Extra fetch code = 20% performance penalty
  - But no long pauses ==> better response time
- On swap, copy roots as before

## Generational GC

- Empirical observation: if an object has been reachable for a long time, it is likely to remain so
- Empirical observation: in many languages (especially functional languages), most objects died young
- Conclusion: we save work by scanning the young objects frequently and the old objects infrequently

## Generational GC

- Assign objects to different generations G0, G1, ...
  - G0 contains young objects, most likely to be garbage
  - G0 scanned more often than G1
  - Common case is two generations (new, tenured)
  - Roots for GC of G0 include all objects in G1 in addition to stack, registers

## Generational GC

How do we avoid scanning tenured objects?

- Observation: old objects rarely point to new objects
  - Normally, object is created and when it initialized it will point to older objects, not newer ones
  - Only happens if old object modified well after it is created
  - In functional languages that use mutation infrequently, pointers from old to new are very uncommon
- Compiler inserts extra code on object field pointer write to catch modifications to old objects
- Remembered set is used to keep track of objects that point into younger generation. Remembered set included in set of roots for scanning.

### Other issues

- When do we promote objects from young generation to old generation
  - Usually after an object survives a collection, it will be promoted
- How big should the generations be?
  - Appel says each should be exponentially larger than the last
- When do we collect the old generation?
  - After several minor collections, we do a major collection

### Other issues

- Sometimes different GC algorithms are used for the new and older generations.
  - Why? Because they have different characteristics
- Copying collection for the new
  - Less than 10% of the new data is usually live
  - Copying collection cost is proportional to the live data
- Mark-sweep for the old
  - Mark reachable
  - Sweep that not marked