Beyond P&P
Memory Management

Dynamic Memory Management

Two basic approaches

Explicit (*e.g.*, `malloc/free`)

- Programmer explicitly allocates and frees memory
- \+ Can be very efficient
- – But error prone

Automatic (*e.g.*, garbage collection)

- Programmer allocates memory
- But runtime system reclaims it
- \+ Very simple for programmer
- – But may slow execution (figuring out what to reclaim)

How Does `malloc` work?

Before calling `main()`

- C runtime system asks OS for large chunk of memory
- This will serve as the heap for that program

`malloc`

- Carves out a piece of the heap
- Records the fact that it is allocated
- Subsequent calls to `malloc` will not use this storage

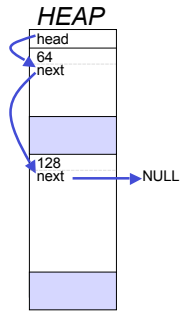
`free`

- Records the fact that piece of heap is no longer allocated
- Subsequent calls to `malloc` can use it

How Does `malloc` Manage the Heap?

Free data is arranged in a linked list (*free list*)

- Next pointer embedded in free data itself (rather than building a separate "list" structure)
- Each chunk is "tagged" with size (prefix or auxiliary table)



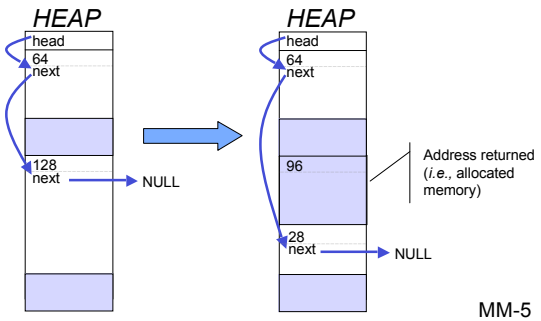
How Does `malloc` Manage the Heap?

When `malloc` called

- Scan list for sufficiently large chunk
  Chunk may need to be split
- Returned chunk also "tagged" with size

Example

- `malloc(96)`

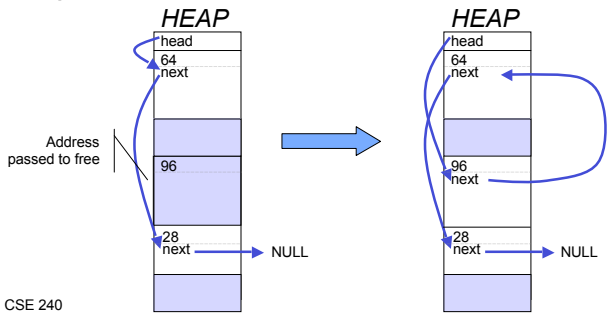


How Does `malloc` Manage the Heap?

When `free` called

- Adds chunk of data back onto the free list (at head)
- Size available in "tag"

Example



Problems

What if wrong address is freed?

```
p = malloc(x);
p++;
free(p);
```

- Size info will be wrong!
- Program will likely (later) fail in mysterious ways

What if same chunk is freed twice?

- Often results in free-list cycle

More Problems

Fragmentation

- After a while, we end up with lots of small chunks
- May have enough memory for some request, but not contiguous
- Solutions
  - Allow heap to grow (ask OS for more memory)
  - Choose chunks wisely: best fit v. first fit

Slow allocation

- Must traverse long list to find suitable chunk of memory
- Solutions
  - Sort free list (now `free` is slower)
  - Multiple free lists (*e.g.*, one for large chunks, one for small, *etc.*)

Bugs

- Wild writes can corrupt size or next pointers
- Solution?
  - Use auxiliary structure to record size (doesn't actually solve problem)
  - Valgrind (provides its own implementation of `malloc`)

It Looks Simple But. . .

Many implementations of malloc

- BSD Malloc
- CSRI UToronto Malloc
- GNU Malloc
- G++ Malloc
- Hoard
- Mmalloc
- ptmalloc
- QuickFit Malloc
- Vmalloc
- And many more. . . (yours?)

Many ways to perfect and customize

- Often best malloc determined by how it is to be used

Garbage Collection

Automatic memory management

- Ideal goal: automatically free/reclaim storage that will not be used in the future (*i.e.,* dead/garbage data)
- Practical alternative: free non-reachable data b/c can't be used again (approx. of ideal goal)

\+ Simple for programmer

– Runtime overhead

Basic approaches to garbage collection

- Reference counting
- Tracing

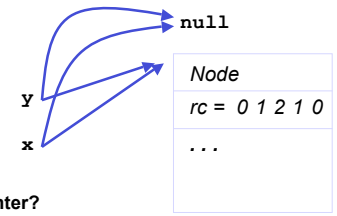
Reference Counting

Idea

- For each heap allocated object, maintain count of # of pointers to it
- When creating object o, rc[o] = 0
- When creating new reference to object o, rc[o]++
- When removing reference to object o, rc[o]--
- If ref count goes to zero, free object (*i.e.,* place on free list)

Example

```
Node x,y;
x = new Node(3,null);
y = x;
x = null;
y = x;
```



Complication

- What if freed object contains pointer?

Reference Counting (cont.)

Pros

- Simple
- Local, incremental work

Cons

- High runtime overhead
  (must manipulate ref counts for every reference update)
- Cannot reclaim cyclic data structures (shifts burden to programmer)
- Space cost (for counts)
- Allocation is expensive (search freelist)
- Fragmentation
- Complicates compilation

Improvements to ref. counting

- Optimize counter updates (combine several into one update)
- Combine ref counting w/ more sophisticated scheme (next)

Trace Collecting

Observation

- Rather than keep track of references to objects we can follow pointers to identify all reachable objects (and discard the rest)
- Faster (when not collecting)

Details (when collecting garbage)

- Start with a set of root pointers (program vars)
  - Global pointers
  - Pointers on stack and in registers
- Traverse objects recursively from root set
  - Visit reachable objects
  - Unvisited objects are garbage
- We might visit object even if it's dynamically dead (*i.e.,* we are only conservatively approximating dead object discovery)

When do we collect?

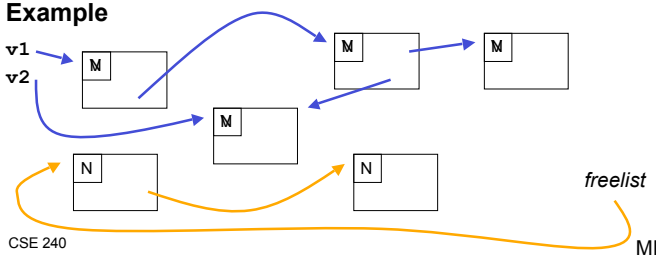
- When the heap is full

Mark-Sweep Collecting

Simple trace collector

1. Trace reachable objects, marking them as we go
2. Sweep through all of heap
   - Add unmarked objects to free list
   - Clear marks of marked objects

Example



Mark-Sweep Collecting (cont.)

Pros

- Simple
- Collects cyclic structures

Cons

- "Embarrassing pause" problem
- Fragmentation
- Must be able to dynamically identify pointers in vars and objects
- More complex runtime system

Improvement: Mark-Compact

+ No fragmentation problem, allocation is very fast (no free list to search)
– Slower sweep
– Need to change pointers because when an object is moved all pointers to it must be updated

Pointers

Issues

- In order to trace reachable objects, must be able to dynamically determine what is a pointer
  - Imagine doing this in C!
  - Easier in Java
- How?
  - Compiler support for tagging data
- What if we're not certain about what is a pointer?
  - Be conservative; assume anything that may be a point is
  - May keep extra garbage
  - Cannot move objects (mark-compact)
  - Conservative garbage collectors can be used with C

Copy Collecting

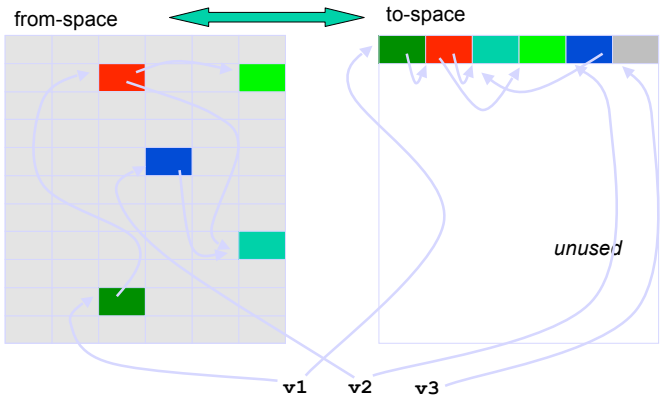
Idea

- Move objects to "new" heap while tracing

Details

- Divide heap in half (prog. allocs. in from-space, to-space is empty)
- When from-space is full...
  - Copy non-garbage from from-space to to-space when visiting object during tracing
  - To-space is now compact

Copy Collecting Example



Copy Collecting (cont.)

Pros

- Collects cyclic structures
- Supports compaction (no free-list, fast allocation)
- Only visits reachable objects, not all objects in heap

Cons

- Requires twice the (virtual) memory
- "Embarrassing pause" problem
- Copying can be slow (especially for large objects)
- Need to update pointers because objects are moving

Hybrid Collectors

Idea

- Different collection techniques may be combined

Example: Mark-Sweep/Copy collector

- Big objects managed with mark-sweep (avoids copy time)
- Small objects managed with copy collector

\+ May be more efficient

– More complex

Generational Collecting

Observation

- "Young" objects are likely to die soon
- "Old" objects are more likely to live on

Idea

- Exploit this fact by concentrating collection on "young" objects

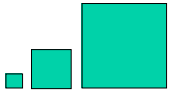
Details

- Divide heap in generations (G_0, G_1, ...; G_0 for youngest objects)
- Collect G_0 most frequently, G_1 less frequently, *etc.*
- Object is "tenured" from one gen. to next after surviving several GCs

Result

- Usually only have to collect a small sub-heap

Generational Collecting (cont.)



Issues

- Need to encode "age" in object
- Generation G_i should be *k* times larger than (younger) G_{i-1}
- Each generation may be collected with different algorithm
- Root set for objects in one generation may come from another gen.

Dealing with cross-generation pointers

- Younger to older (*i.e.,* G_y to G_o for o>y) are very common
  - Collect (younger) G_y when collecting (older) G_o
- Older to younger (*i.e.,* G_o to G_y for o>y) are uncommon
  - Search all of G_o?
  - Need to limit search by detecting writes into G_o
    - Remembered lists (compiler support)
    - Card marking (compiler support)
    - Page marking (no compiler support)

Generational Collecting (cont.)

Pros

- Most collections are short (less of an "embarrassing pause")
- Lower runtime overhead
- Less memory required (if use mark-sweep for older generations)

Cons

- Still sometimes do full, slow collections (embarrassing pause!)
- Need to record age with each object

Memory Management

Easy to ignore when it works well!

Alternatives

- Hybrids
  - Manage some of heap explicitly
  - Manage some automatically
- Regions
  - Group related data in region
  - Free all objects in region at once
- . . .

Next Time

Review and Lunch?