

Memory Allocation and Recycling



O(1) Addressing

- (Assume no "paging" nor "caching" for now).
- Linear Address Space:
Memory is effectively like a big array.
- Each word is accessible in the same amount of time.
 - A **decoder tree** in logic permits this.
 - The time bound for decoding is actually $O(\log n)$.
 - However, the clock interval is designed to be long enough so that it **practically** is $O(1)$.

How Memory is Used

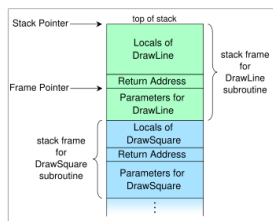
- Code
- Static variables:
remain allocated throughout execution
- "Automatic" variables:
e.g. arguments and local variables of nested functions
These **cease to exist** after the function returns.
- Dynamic variables:
instance variables of objects created **during** execution

Why "Automatic"?

- Automatic is not strictly necessary; dynamic could be used for it.
- However, due to **nested** calling discipline, reclamation of automatic is cheaper than dynamic in general.

Stack-Based Allocation

Low-overhead, but confining



From http://en.wikipedia.org/wiki/Call_stack

Heap-Based Allocation

More flexibility, but more overhead

Heap:



Space overheads:

Each block stores a size.

Free blocks also store a pointer to the "next" free block.

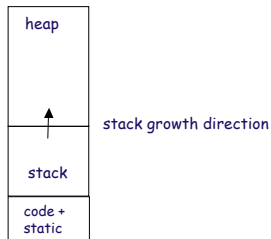
Time overheads:

Must *search* for an adequate free block.

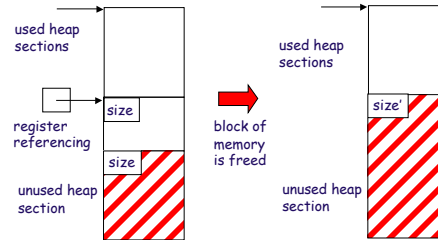
Must *sub-divide* free blocks that are bigger than requirement.

Must *coalesce* blocks as they become freed.

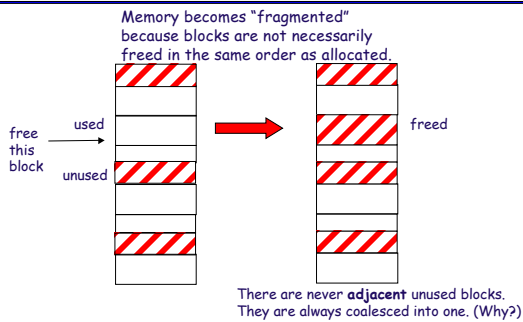
Memory is Pre-Divided



The Recycling Aspect

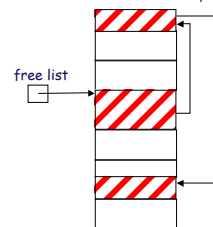


The Bigger Picture



Maintaining the "Free List"

Each unused block has a **size** field and a pointer to the **next** unused block.



Heap Issues

- Fragmentation ("checkerboarding")
 - Why is this an issue?
- Allocation Policy:
 - First-fit
 - Best-fit
 - ...

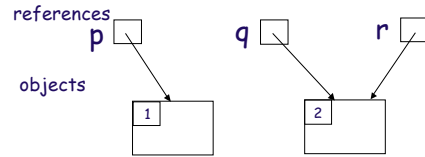
Approaches to Recycling Heap Memory

- Don't-do-it approach
- Programmer-burden approach
- Automatic approaches
 - Reference-Counting
 - Garbage collection
 - Mark-Sweep
 - Copying
 - Generational
 - others

Reference Counting

- Each object has a reference count, not normally shown.
- An **invariant** is maintained:
Reference count =
of references pointing to this object

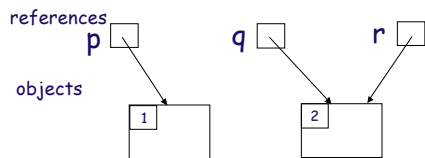
Reference Counting



What happens when we execute:

`q = p;`

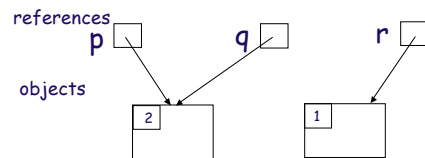
Reference Counting



What happens when we execute:

`q = p; // "make q point to where p points"`

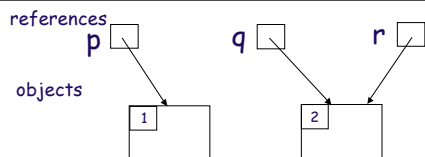
Reference Counting



What happens when we execute:

`q = p; // "make q point to where p points"`

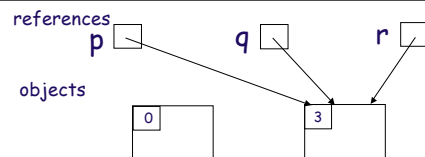
Reference Counting



What happens when we execute:

`p = q;`

Reference Counting

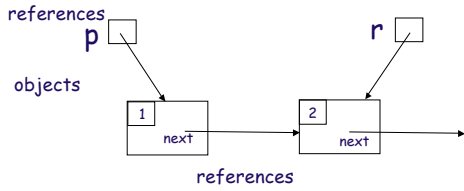


What happens when we execute:

`p = q;`

The block to which p formerly pointed is **reclaimable**.

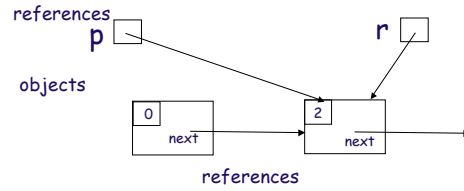
Linked List



What happens when we execute:

`p = p.next;`

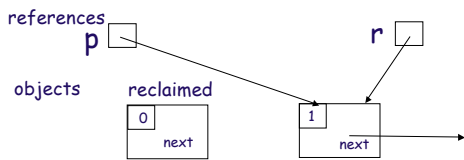
Linked List



What happens when we execute:

`p = p.next;`

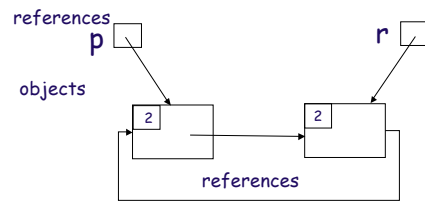
Linked List



What happens when we execute:

`p = p.next;`

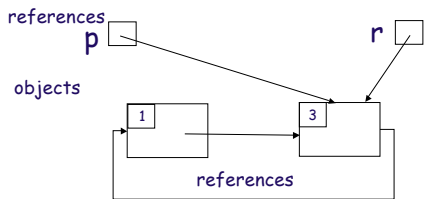
Circular Linked List



What happens when we execute:

`p = r;`

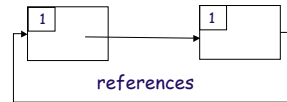
Circular Linked List



What happens when we execute:

`p = null;`
`r = null;`

Circular Linked List



Oops!

Garbage Collection (GC)

- Reference counting keeps track of what is accessible by modifying reference counts at each operation.
- Garbage collection does not wait until memory is scarce, then determines what is accessible. The rest can become free memory.
- Garbage collection does not suffer from the cyclic problem of reference counting.
- GC was invented by John McCarthy in conjunction with early Lisp implementation.

Garbage Collection

- is essentially a graph-search problem.
- Determine what is accessible from one or more "roots" of a directed graph.
- The **complement** of accessible is inaccessible, i.e. garbage.

Graph Search?

- Depth-First Search
- Breadth-First Search
- Iterative Deepening

Some GC Techniques

- Mark/Sweep
- Copying
- Generational

Mark/Sweep Garbage Collection

- Do a search (say depth-first) from the roots of the "memory graph".
- Mark any reachable nodes as you go.
- (By making a pass over *all* nodes linearly through memory) Sweep up any unmarked nodes into the free list.
- (This all supposes that node entities are clearly identifiable. It requires that memory be maintained appropriately.)

Memory Overhead Comparison

- Reference counting:
 - One integer per node
- Mark/sweep:
 - One bit per node

Time Overhead Comparison

- Reference counting:
 - A small tax on every operation
- Mark/sweep:
 - A big tax when memory runs out

Copying Garbage Collection

- Divide the memory into two **half-spaces**, say A and B.
- Work within one half-space (A or B) at any given time.
- Assuming using A now. When it comes time to collect garbage, perform a depth-first search, **copying** each used record from A to B. Then switch to using B.

Copying Advantages

- Trivial to **compact** as you copy. Relative locations do not get maintained.
- Caution: Cannot rely on any absolute addresses, because memory is generally **relocated**.
- This results in a single large unused chunk of memory on each collection.
- **Real-time** versions of copying have been devised (cf. Henry Baker article).

Copying GC Disadvantages?

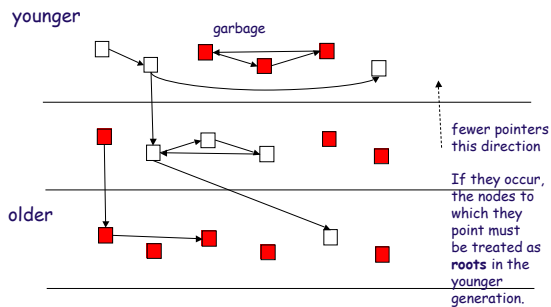
Generational Garbage Collection

- This is one of many heuristics used to reduce overhead in GC.
- It can be observed that some nodes are **ephemeral** (temporary and quickly become garbage), while others have great **longevity**.
- Thus devise a way to do a **quick partial** collection to pick up the ephemeral nodes, reserving a full GC until more desperate.

Generational Garbage Collection

- The extension of the dichotomy ephemeral vs. long-lived is achieved by assigning nodes to **generations**.
- A node in the **youngest** generation is usually the more likely to become garbage.
- References generally point from a younger to an older generation, but not so much vice-versa.
- If a node survives collection at one generation, it is **promoted** to the next older generation.
- A generation is collected only if there is not enough memory freed by collecting younger generations.

Generational Garbage Collection



Food for Thought (or indigestion?)

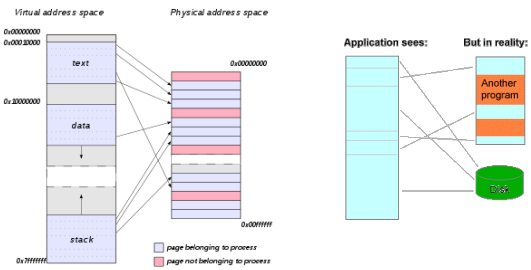
- How does the need for memory recycling impact the desire to use threads?

More Considerations

- Approaches to making memory larger/faster:
 - Paging, Virtual Memory
 - Caching
 - Each comes with its own set of issues

Paging

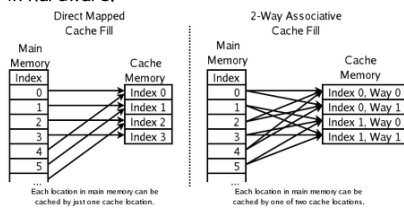
Purpose: Virtual memory, Sharing physical memory



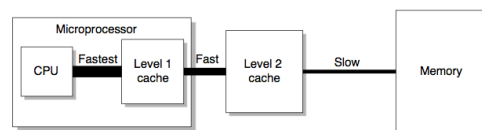
Caching

Purpose: Faster memory for faster execution

Analogous to paging and to hashing in some ways, but done in hardware.



Multi-Level Cache



Computer Memory Hierarchy

