Linking & Runtime

CSE413

Autumn 2007

Agenda

- Linking
- Memory Management

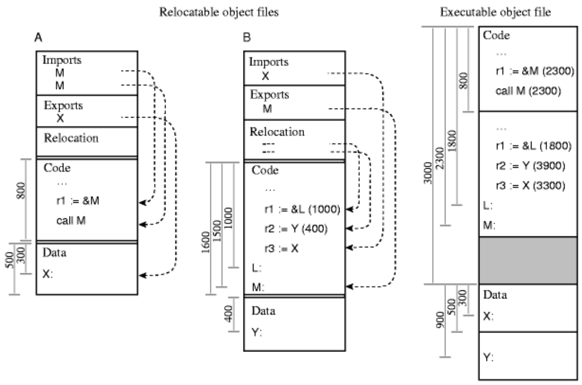
Runtime Systems

- Compiled code + runtime system = executable

The runtime system can include library functions for:

- I/O, for console, files, networking, etc.
- graphics libraries, other third-party libraries
- reflection: examining the static code & dynamic state of the running program itself
- threads, synchronization
- memory management
- system access, e.g. system calls

Static Linking Example



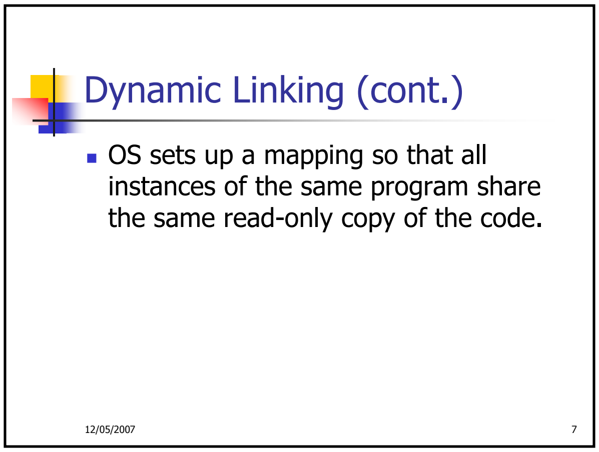
Libraries

- contain lots of code, you don't need all of it
- linkers search the library and only pull in the code that you need.
- libraries are often stored in a special format to make this easier.

Dynamic Linking

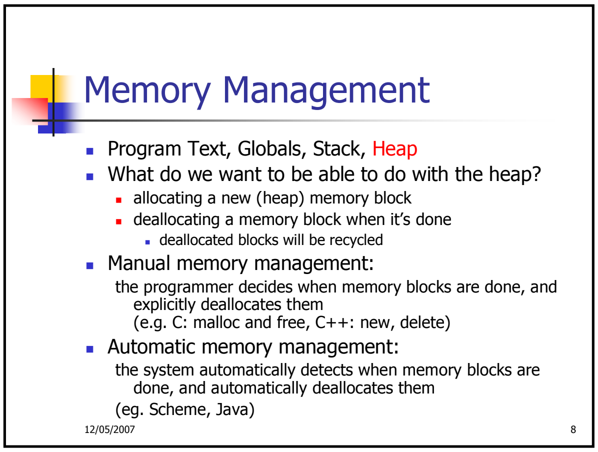
Observations:

- Several instances of a program are often live at the same time.
- Programs share code (graphics routines)
- Libraries often improve over time



Dynamic Linking (cont.)

- OS sets up a mapping so that all instances of the same program share the same read-only copy of the code.



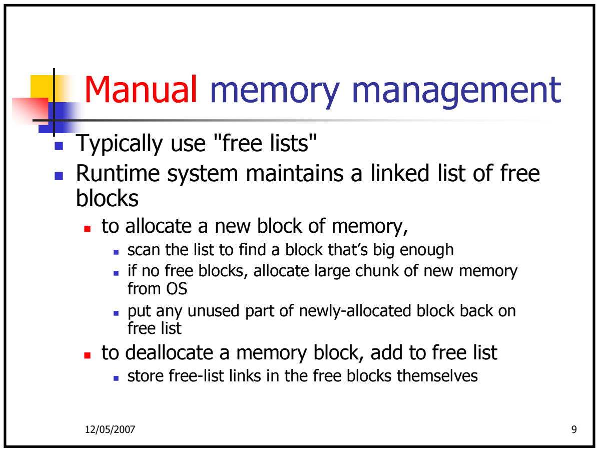
Memory Management

- Program Text, Globals, Stack, Heap
- What do we want to be able to do with the heap?
  - allocating a new (heap) memory block
  - deallocating a memory block when it's done
    - deallocated blocks will be recycled
- Manual memory management:

  the programmer decides when memory blocks are done, and explicitly deallocates them
  (e.g. C: malloc and free, C++: new, delete)
- Automatic memory management:

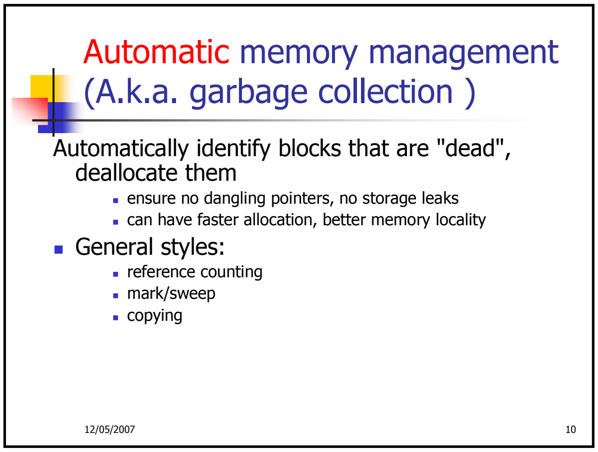
  the system automatically detects when memory blocks are done, and automatically deallocates them

  (eg. Scheme, Java)



Manual memory management

- Typically use "free lists"
- Runtime system maintains a linked list of free blocks
  - to allocate a new block of memory,
    - scan the list to find a block that's big enough
    - if no free blocks, allocate large chunk of new memory from OS
    - put any unused part of newly-allocated block back on free list
  - to deallocate a memory block, add to free list
    - store free-list links in the free blocks themselves

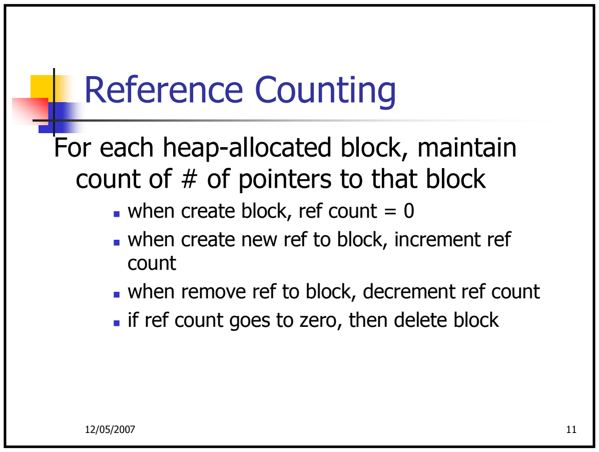


Automatic memory management (A.k.a. garbage collection)

Automatically identify blocks that are "dead", deallocate them

- ensure no dangling pointers, no storage leaks
- can have faster allocation, better memory locality

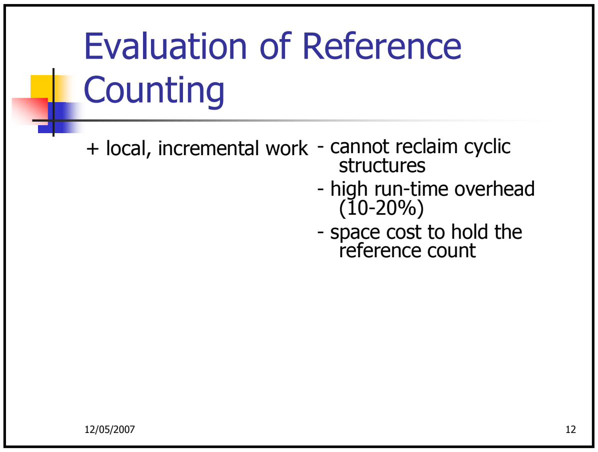
- General styles:
  - reference counting
  - mark/sweep
  - copying



Reference Counting

For each heap-allocated block, maintain count of # of pointers to that block

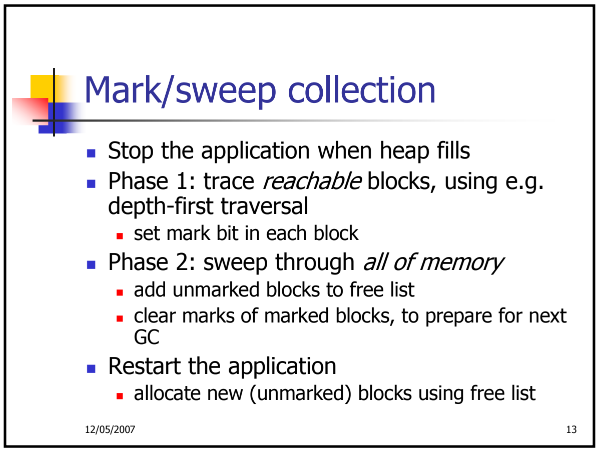
- when create block, ref count = 0
- when create new ref to block, increment ref count
- when remove ref to block, decrement ref count
- if ref count goes to zero, then delete block



Evaluation of Reference Counting

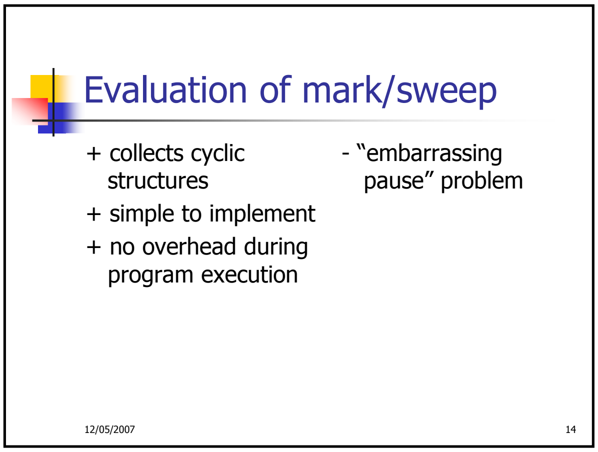
+ local, incremental work

- cannot reclaim cyclic structures
- high run-time overhead (10-20%)
- space cost to hold the reference count



Mark/sweep collection

- Stop the application when heap fills
- Phase 1: trace *reachable* blocks, using e.g. depth-first traversal
  - set mark bit in each block
- Phase 2: sweep through *all of memory*
  - add unmarked blocks to free list
  - clear marks of marked blocks, to prepare for next GC
- Restart the application
  - allocate new (unmarked) blocks using free list



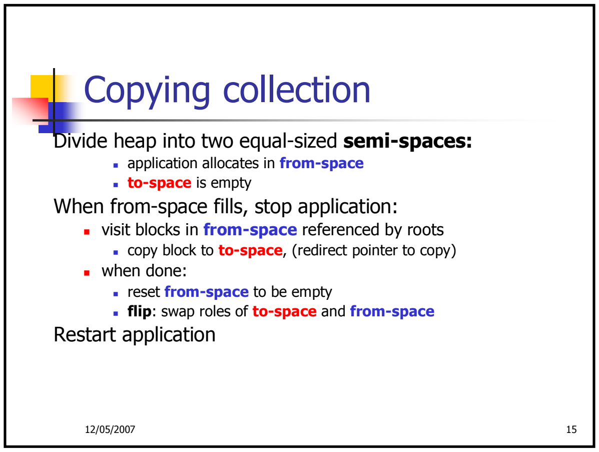
Evaluation of mark/sweep

+ collects cyclic structures

+ simple to implement

+ no overhead during program execution

- "embarrassing pause" problem



Copying collection

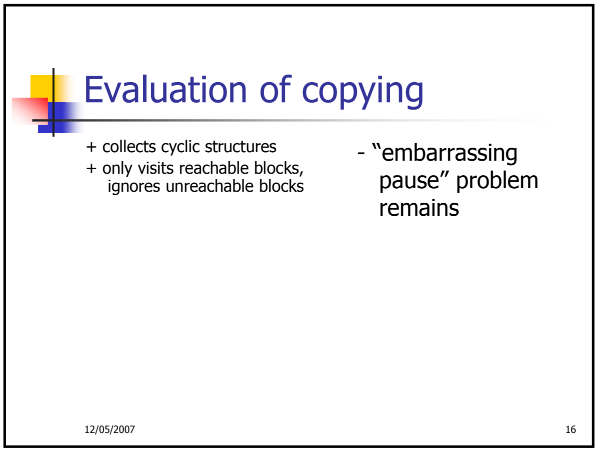
Divide heap into two equal-sized **semi-spaces:**

- application allocates in **from-space**
- **to-space** is empty

When from-space fills, stop application:

- visit blocks in **from-space** referenced by roots
  - copy block to **to-space**, (redirect pointer to copy)
- when done:
  - reset **from-space** to be empty
  - **flip**: swap roles of **to-space** and **from-space**

Restart application



Evaluation of copying

+ collects cyclic structures

+ only visits reachable blocks, ignores unreachable blocks

- "embarrassing pause" problem remains