# Memory Management:
# Free space and dynamic allocation

M1 MOSIG – Operating System Design

Renaud Lachaize

# Acknowledgments

- Many ideas and slides in these lectures were inspired by or even borrowed from the work of others:
  - Arnaud Legrand, Noël De Palma, Sacha Krakowiak
  - David Mazières (Stanford)
    - (most slides/figures directly adapted from those of the CS140 class)
  - Randall Bryant, David O'Hallaron, Gregory Kesden, Markus Püschel (Carnegie Mellon University)
    - Textbook: Computer Systems: A Programmer's Perspective ($2^{nd}$ Edition)
    - CS 15-213/18-243 classes (some slides/figures directly adapted from these classes)
  - Remzi and Andrea Arpaci-Dusseau (U. Wisconsin)
  - Textbooks (Silberschatz et al., Tanenbaum)

# Outline

- **Introduction**
  - Motivation
  - Fragmentation
- How to implement a memory allocator?
  - Key design decisions
  - A comparative study of several simple approaches
  - Known patterns of real programs
  - Some other designs
- Implicit memory management (garbage collection)

# Dynamic memory allocation – Introduction (1)

- **Almost every program uses it**
  - Gives very important functionality benefits
    - Avoids statically specifying complex data structures
    - Avoids static overprovisioning of memory
    - Allows having data grow as a function of input size
  - But can have a huge impact on performance

- **A general principle, used at several levels of the software stack**:
  - **In the operating system kernel**, to manage physical memory
  - **In the C library**, to manage the heap, a specific zone within the virtual memory of each process
  - (And also possibly) **within an application**, to manage a big chunk of virtual memory provided by the OS

# Dynamic memory allocation – Introduction (2)

- **Today's focus: how to implement it**

- **Some interesting facts (on performance)**

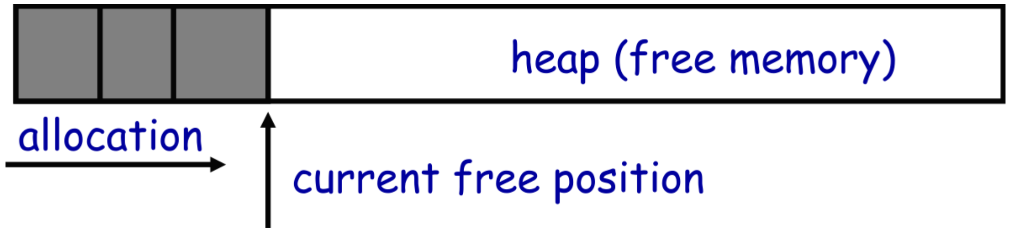
  – Changing a few lines of code can have huge, non-obvious impact on how well an allocator works (examples to come)

  – Proven: impossible to construct an "always good" allocator

  – Surprising result: after decades, memory management is still poorly understood

# Why is it hard?

- **Must satisfy arbitrary sequence of alloc/free operations**
- Easy without free:
  – Set a pointer to the beginning of some big chunk of memory (“heap”) and increment on each allocation




- **Problem: free creates holes (“*fragmentation*”). Result: lots of free space but cannot satisfy request**

- Why can’t we just move everything to the left when needed?
  – This requires to update memory references (and thus to know about the semantics of data)

# External Fragmentation

- **Occurs when there is enough aggregate heap memory, but no single free block is large enough**

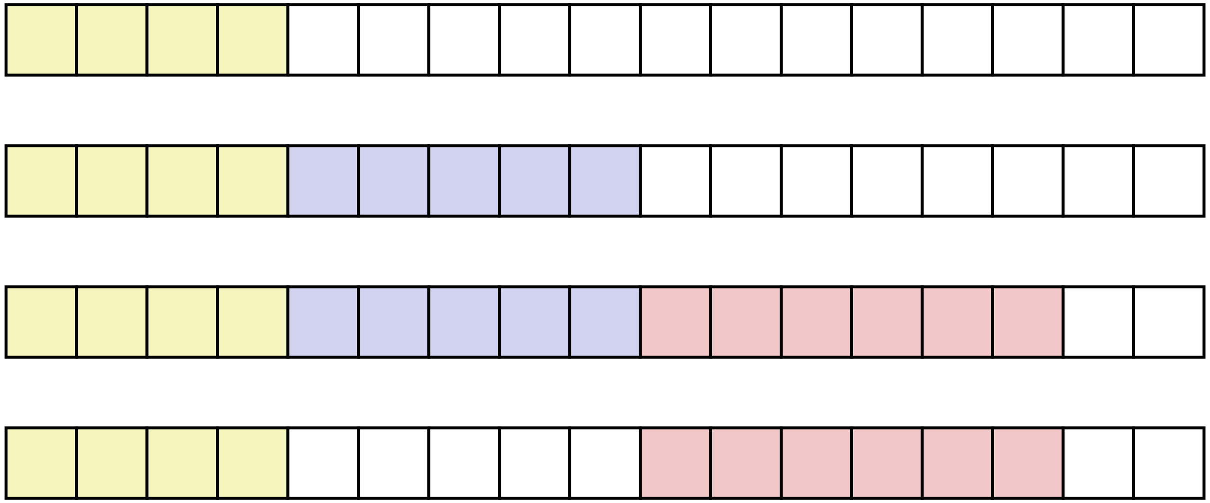
```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(6)
```

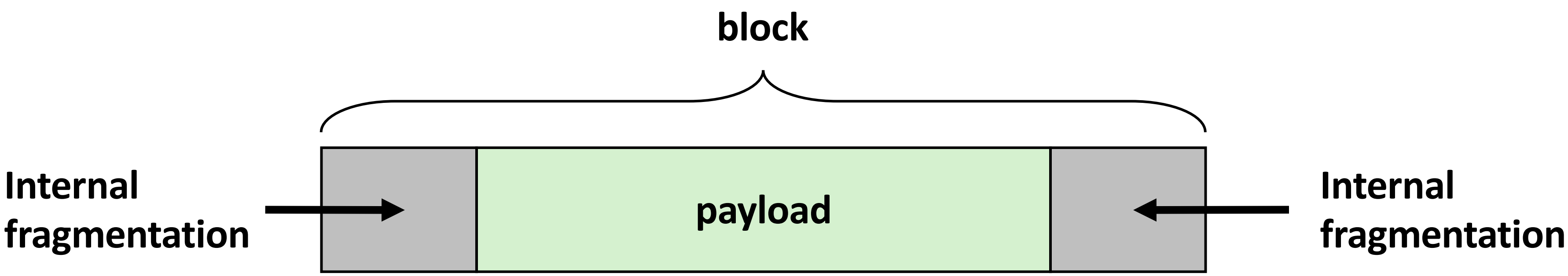


*Oops! (what would happen now?)*

(adapted from the following source: Carnegie Mellon University – 15-213/18-243 class)

# Internal Fragmentation

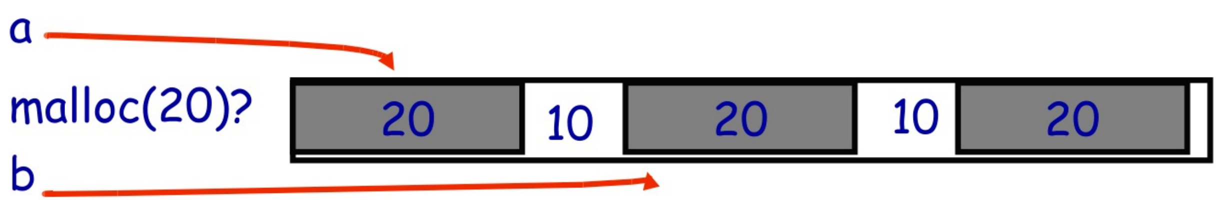
- <u>**For a given block**</u>**,** ***internal fragmentation*** **occurs if payload is smaller than block size**




- Caused by:
  - overhead of maintaining heap data structures
    - (e.g., memory footprint of metadata headers/footers)
  - padding for alignment purposes
  - explicit policy decisions
    - (e.g., decision to return a big block to satisfy a small request, in order to make the operation go faster)

# More abstractly

- **What an allocator must do**:
  - Track which parts of memory are in use, and which parts are free
  - Ideally: no wasted space, no time overhead
- **What the allocator <u>cannot</u> do**:
  - Control order, number and size of the requested blocks
  - Change user pointers (therefore, bad placement decisions are permanent)




- The core fight: **<u>minimize fragmentation</u>**
  - Application frees blocks in any order, creating holes in "heap"
  - If holes are too small, future requests cannot be satisfied

# What is (external) fragmentation?

- **Inability to use memory that is free**
- **Two factors required for <u>external</u> fragmentation**
  - **Different lifetimes**:
    - If adjacent objects die at different times, then fragmentation

    - If they die at the same time, then no fragmentation

  - **Different sizes**:
    - If all requests have the same size, then no fragmentation

    - (As we will see later, in the context of virtual memory, paging relies on this to remove external fragmentation)

# Outline

- Introduction
  - Motivation
  - Fragmentation
- **How to implement a memory allocator?**
  - Key design decisions
  - A comparative study of several simple approaches
  - Known patterns of real programs
  - Some other designs
- Implicit memory management (garbage collection)

# Important design decisions (1/5)

- **Free block organization:** How do we keep track of free blocks?
- **Placement:** How do we choose an appropriate free block in which to place a newly allocated block?
- **Splitting:** After we place a newly allocated block in some free block, what do we do with the remainder of the free block?
- **Coalescing:** What do we do with a block that has just been freed?

# Important design decisions (2/5)

- **Free block organization**: *How do we keep track of free blocks?*
  - Common approach: **"free list"** i.e., linked list of descriptors of free blocks
  - **Multiple strategies to sort the free list**
  - For space efficiency, **the free list is stored within the free space!**
  - (There are also other approaches/data structures beyond free lists, e.g., balanced trees)

# Important design decisions (3/5)

- **Placement**: *How do we choose an appropriate free block in which to place a newly allocated block?*
  - **Placement strategies have a major impact on external fragmentation**
  - We will study  several examples soon
    - (best fit, first fit, …)
  - Ideally: put block where it will not cause fragmentation later
    - Impossible to guarantee in general: requires knowledge about the future

# Important design decisions (4/5)

- **Splitting**: *After we place a newly allocated block in some free block, what do we do with the remainder of the block?*

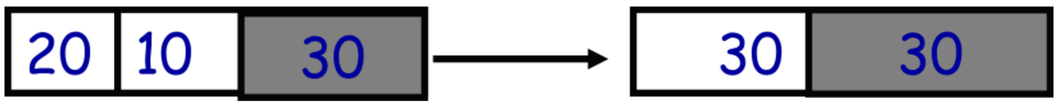
**Two choices:**

  - **Keep the remainder within the chosen block**
    - Simple, fast
    - but introduces more internal fragmentation
  - **Split the chosen block in two** and insert the remainder block in the free list
    - Better with respect to internal fragmentation (less wasted space)
    - … But requires more work (and thus more time), which may be wasted if most remainder blocks are useless (too small)

# Important design decisions (5/5)

- **Coalescing**: *What do we do with a block that has just been freed?*
  - The adjacent blocks may be free
  - Coalescing the newly freed block with the adjacent free block(s) yields a larger free block

  - **This helps avoiding "false external fragmentation"**




  - **Different strategies**:
    - **Immediate** coalescing: systematic attempt whenever a block is freed
      - This may sometimes work "too well"
    - **Deferred**: only on some occasion (e.g., when we are running out of space) or periodically

# Impossible to “solve” fragmentation

- If you read research/technical papers to find the best allocator
  - All discussions revolve around trade-offs
  - Because **there cannot be a best allocator**

- **Theoretical result**
  - ***For any possible allocation algorithm, there exists streams of allocation and deallocation requests that defeat the allocator and force it into severe fragmentation***

- How much fragmentation should we tolerate?
  - Let M = bytes of live data, $n_{min}$ = smallest allocation size, $n_{max}$ = largest allocation size
  - How much gross memory required?
  - Bad allocator: M . ($n_{max}$ / $n_{min}$)
    - (uses maximum size for any size)
  - Good allocator: ~ M . log($n_{max}$ / $n_{min}$)

# Pathological example

- Example: Given allocation of 7 20-byte blocks


  - What is a bad stream of frees and then allocates?
  - Free every one block out of two, then alloc 21 bytes

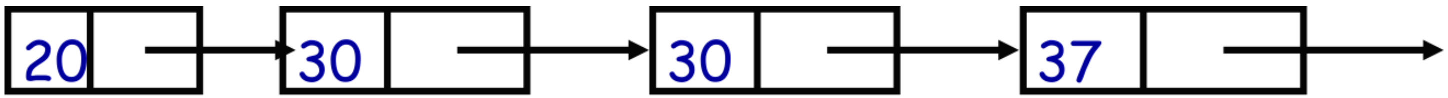
- Next: we will study two allocators (placement strategies) that, in practice, work pretty well: “best fit” and “first fit”

# Outline

- Introduction
  - Motivation
  - Fragmentation
- How to implement a memory allocator?
  - Key design decisions
  - **A comparative study of several simple approaches**
  - Known patterns of real programs
  - Some other designs
- Implicit memory management (garbage collection)

# Best fit

- **Placement strategy: minimize fragmentation by allocating space from block that leaves smallest fragment**
  - **Data structure:** heap is a list of free blocks, each has a header holding block size and pointer to next




  - **Code:**
    - Search free list for block closest in size to the request (exact match is ideal)
    - During free, (usually) coalesce adjacent blocks

- **Problem:** Sawdust
  - Remainder so small that, over time, we are left with "sawdust" everywhere

- Implementation? (go through the whole list? maintain sorted list?)

# First fit

- **Strategy: pick the first block that fits**
  - Data structure: free list
  - Code: scan list, take the first one
  - **Implementation: Multiple strategies for sorting the free list: LIFO, FIFO or by address** (see below)

- **LIFO:** put free block on front of list
  - Simple but causes higher fragmentation (see details on next slide)
  - Potentially good for cache locality

- **Address sort:** order free blocks by address
  - Makes coalescing easy (just check if next block is free)
  - Also preserves empty/idle space (locality good when paging)

- **FIFO:** put free block at end of list

# Subtle pathology: LIFO first fit [Advanced]

- An example of the subtle impact of simple design decisions

- LIFO first fit seems good:
  - Put object on front of list (cheap), hope same size used again (cheap + good locality)

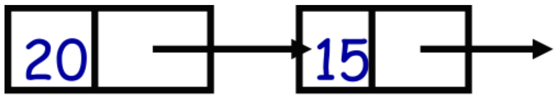
- But has big problems for simple allocation patterns
  - E.g., repeatedly intermix short-lived allocations of *2n* bytes, with long-lived allocations of *(n+1)* bytes
  - Each time a large object is freed, a small chunk will be quickly taken, leaving useless fragment. Pathological fragmentation

# First fit: Nuances [Advanced]

- First fit sorted by address order, in practice:
  - Blocks at front preferentially split, ones at back only split when no larger one found before them
  - Result? Seems to roughly sort free list by size
  - So? Makes first fit operationally similar to best fit: a first fit of a sorted list = best fit!

- Problem: sawdust at beginning of the list
  - Sorting of list forces a large request to skip over many small blocks.

- Suppose memory has free blocks:




  - If allocation operations are 10 then 20, best fit wins
  - When is first fit better than best fit?
  - Suppose allocation operations are 8, 12, 12. Then first fit wins

# Some other placement strategies

- **Worse fit**
  - Strategy: fight against sawdust by splitting block to maximize leftover size
  - However, seems to ensure that there are no large blocks

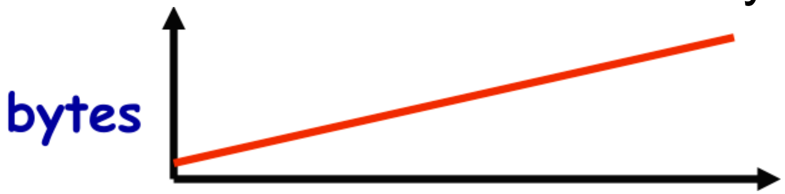
- **Next fit**
  - Strategy: use first fit, but remember where we found the last thing and start searching from there
  - Seems like a good idea, but tends to break down entire list

# Outline

- Introduction
  - Motivation
  - Fragmentation
- How to implement a memory allocator?
  - Key design decisions
  - A comparative study of several simple approaches
  - **Known patterns of real programs**
  - Some other designs
- Implicit memory management (garbage collection)

# Known patterns of real programs

- So far, we have treated programs as black boxes
- Most real programs exhibit 1 or 2 (or all 3) of the following patterns of alloc/dealloc:
  - Ramps: accumulate data monotonically over time




  - Peaks: allocate many objects, use briefly, then free all


  - Plateaus: allocate many objects, use for a long time

# Outline

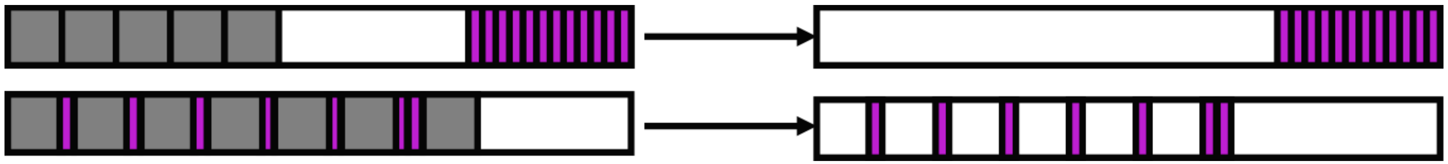
- Introduction
  - Motivation
  - Fragmentation
- How to implement a memory allocator?
  - Key design decisions
  - A comparative study of several simple approaches
  - Known patterns of real programs
  - **Some other designs**
- Implicit memory management (garbage collection)

# Beyond simple free lists [Advanced]

- We will study a few examples of other approaches:
  - Segregated lists
  - Slab caches
  - Buddy allocation

# Fighting fragmentation
# Exploiting ordering and size dependencies [Advanced]

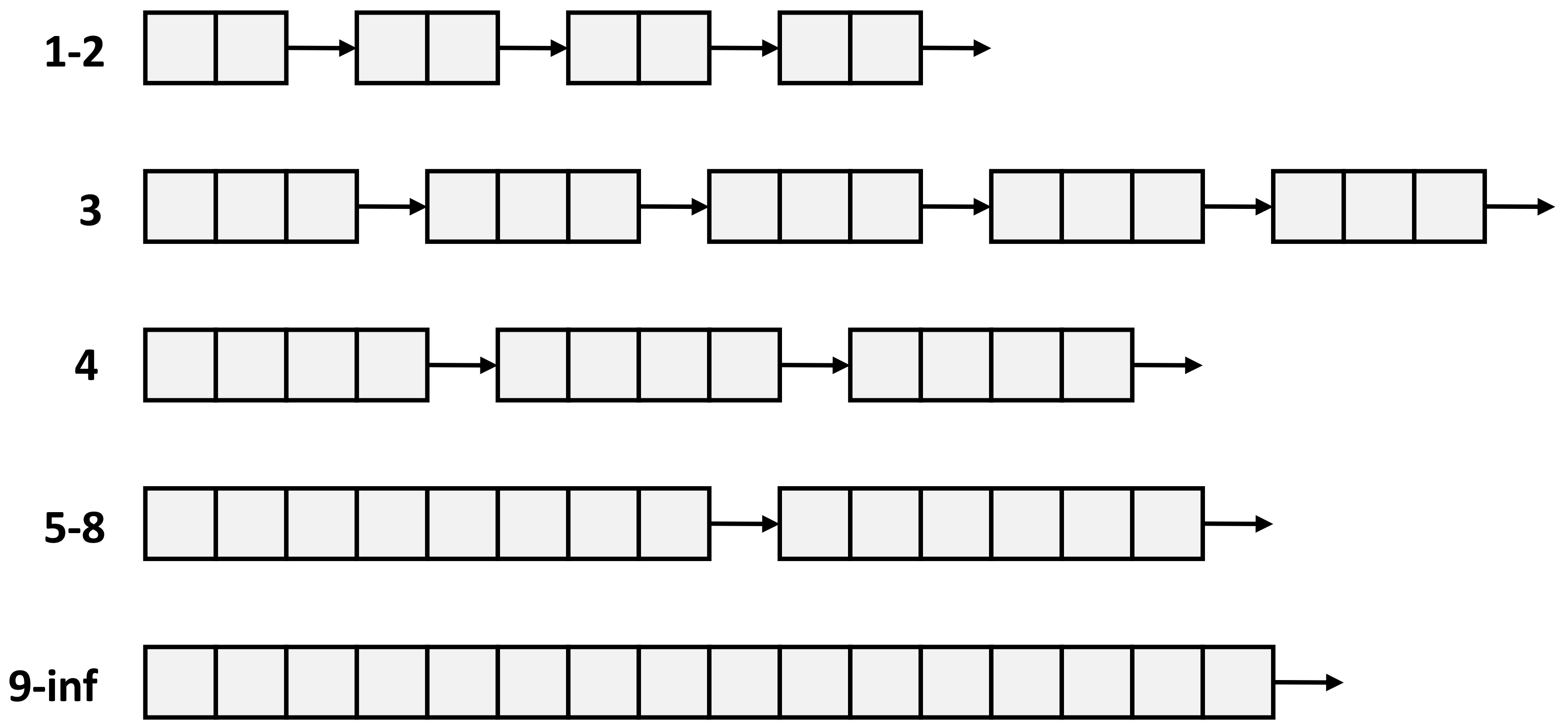
- **Segregation = reduced fragmentation**
  - Allocated at same time ~ freed at same time
  - Different type ~ freed at different time



- Implementation observations
  - Programs allocate small number of different sizes
  - Fragmentation at peak use is more important than at low
  - Most allocations are small (< 10 words)
  - Work done with allocated memory increases with size
  - Implications?

# Segregated List (Seglist) Allocators [Advanced]

- Each ***size class*** of blocks has its own free list




- Often have separate classes for each small size
- For larger sizes: One class for each two-power size

(adapted from the following source: Carnegie Mellon University – 15-213/18-243 class)

# Slab allocation [Advanced]

- **Remember what we told earlier : if all requests have the same size, then no fragmentation**
- The kernel allocates many instances of the same structures
  - E.g., a 1.7 kB `task_struct` for every process on the system
  - And often needs contiguous physical memory
- **Slab allocation optimizes for this case:**
  - A slab is multiple pages of contiguous physical memory
  - A cache contains one or more slabs
  - Each cache stores only one kind of object (fixed size)
- Each slab is full, empty or partial
- Example: need new `task_struct`?
  - Look in the `task_struct` cache
  - If there is a partial slab, pick free `task_struct` in that
  - Else use empty, or may need to allocate new slab for cache
- Advantages: speed and no internal fragmentation [Bonwick]

# Buddy allocation [Advanced]

- **A special form of segregated allocator**

- Here we only discuss the most common type of buddy system: **binary buddies**

- **Relies on specific rules to make management faster**:
  - Rounds up all allocation sizes to powers of 2
  - Imposes specific rules/restrictions on splitting/coalescing procedures
  - Fast but may result in heavy internal fragmentation

# Dynamic memory management
# Recap

- **(External) Fragmentation is caused by**:
  - Size heterogeneity
  - Isolated deaths
  - Time-varying behavior
- **Allocator should try to**:
  - Exploit memory patterns
  - Be evaluated under real workloads
  - Have smart and efficient (in space and time) implementation

# Summary of Key Allocator Policies

- **Placement policy**:
  - First-fit, next-fit, best-fit, etc.
  - Trades off lower throughput for less fragmentation
  - ***Interesting observation***: segregated free lists approximate a best fit placement policy without having to search entire free list
- **Splitting policy**:
  - When do we go ahead and split free blocks?
  - How much internal fragmentation are we willing to tolerate?
- **Coalescing policy**:
  - ***Immediate coalescing:*** coalesce each time **`free()`** is called
  - ***Deferred coalescing:*** try to improve performance of **`free()`** by deferring coalescing until needed. Examples:
    - Coalesce as you scan the free list for **`malloc()`**
    - Coalesce when the amount of external fragmentation reaches some threshold

(adapted from the following source: Carnegie Mellon University – 15-213/18-243 class)

# Outline

- Introduction
  - Motivation
  - Fragmentation
- How to implement a memory allocator?
  - Key design decisions
  - A comparative study of several simple approaches
  - Known patterns of real programs
  - Some other designs
- **Implicit memory management (garbage collection)**

# Implicit Memory Management: Garbage Collection

- ***Garbage collection***:
  - **Automatic reclamation of heap-allocated storage**
  - **The application never has to free**
    - Avoids many memory management bugs
      - Examples: double free bugs, some forms of dangling pointers, some forms of memory leaks
    - … but not all of them
    - Usually yields lower performance than manual memory management

- **Common in many languages**
  - Functional languages (e.g., Lisp, ML)
  - Scripting languages (e.g., Perl)
  - Modern object-oriented languages (e.g., Java)
- Variants (“conservative” garbage collectors) exist for C and C++
  - However, cannot necessarily collect all garbage

# Garbage collection

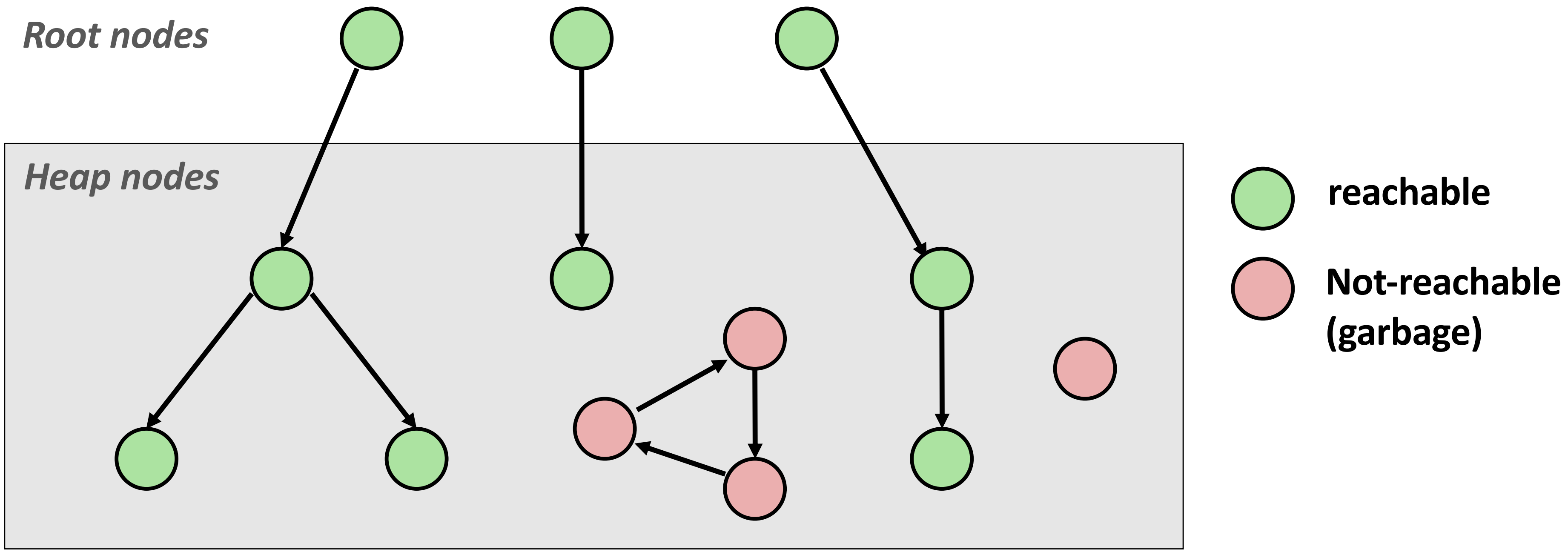
- **Main principle:** How does the memory manager know when a memory block can be freed?
  - In general we cannot know what is going to be used in the future since it depends on conditionals
  - But we can tell that certain blocks cannot be used if there are no pointers to them
  - **A (dynamically allocated) memory block becomes garbage (i.e., useless) when it cannot be reached anymore by the application**

# Garbage collection (continued)

- **Assumptions**
  - Pointers (i.e., memory addresses) can be distinguished from other types of variables
  - A pointer can only point to the beginning of a memory block (i.e., not to the middle of a block)
  - A pointer cannot be “hidden” in another data type
- Languages such as C and C++ do not comply with the above assumptions
  - But some restricted forms of garbage collection can nonetheless be implemented with these languages

# Tracing garbage collectors
# Memory as a Graph

- **We view memory as a directed graph**
  - Each block is a node in the graph
  - Each pointer is an edge in the graph
  - Locations not in the heap that contain pointers into the heap are called ***root*** nodes (e.g., registers, locations on the stack, global variables)




A node (block) is *reachable* if there is a path from any root to that node.

Non-reachable nodes are *garbage* (not needed by the application)

(adapted from the following source: Carnegie Mellon University – 15-213/18-243 class)

# Garbage collection algorithms (1/2)

- **Tracing collectors** (example: Mark-and-sweep)
  - Usually triggered when heap runs out of free space
  - Some important criteria
    - **Moving (a.k.a. “compacting”) versus non-moving**
      - Note that, in a safe language (e.g., Java), the runtime system knows about all pointers
      - So an object can be moved if all the related pointers are updated accordingly
      - Good: helpful for fighting fragmentation and improving locality
      - Bad: performance impact of memory copies
    - **Stop-the-world versus incremental versus concurrent**
      - Different trade-offs depending on the requirements of programs (interactivity/reactivity, need to reclaim memory fast, …)
    - **Precise versus conservative**
      - See previous discussions on C/C++

# Garbage collection algorithms (2/2)

- **Reference counting**
  - Another approach (different from tracing collectors)
  - Each object has an internal field (“ref count”), which keeps tracks of the current number of pointers to it
  - The ref count is incremented when a pointer is set to this object
  - The ref count is decremented when a pointer is set to another object or destroyed
  - The object can be reclaimed when the ref count reaches zero

  - Pros
    - No need to halt program when running collector
    - Immediate reclamation of available memory
  - Cons
    - Need to update the ref counts (negative performance effects)
    - Problems with circular data structures (leaks)

# References


- Andrea & Remzi Arpaci-Dusseau. **OSTEP textbook** (http://www.ostep.org). Chapters:
  - *"Memory API"*
  - *"Free space management"*

- [CSAPP (book)] Randall Bryant, David O' Hallaron. ***Computer Systems: A Programmer's Perspective***. Pearson.
  - See chapter on "Virtual memory", section on "Dynamic memory allocation" (Also covers garbage collection)

- [Wilson et al.] P. R. Wilson, M. R. Johnstone, M. Neely, D. Boles. ***Dynamic Storage Allocation: A Survey and Critical Review***. University of Texas at Austin, 1995.

- [Bonwick] J. Bonwick. ***The Slab Allocator: An Object-Caching Kernel Memory Allocator***. Usenix Summer 1994 Technical Conference.