운영체제의 기초:
Dynamic Memory Allocation

2025년 5월 2, 4일

홍 성 수

sshong@redwood.snu.ac.kr

SNU RTOSLab 지도교수

서울대학교 전기정보공학부 교수

Seoul National University

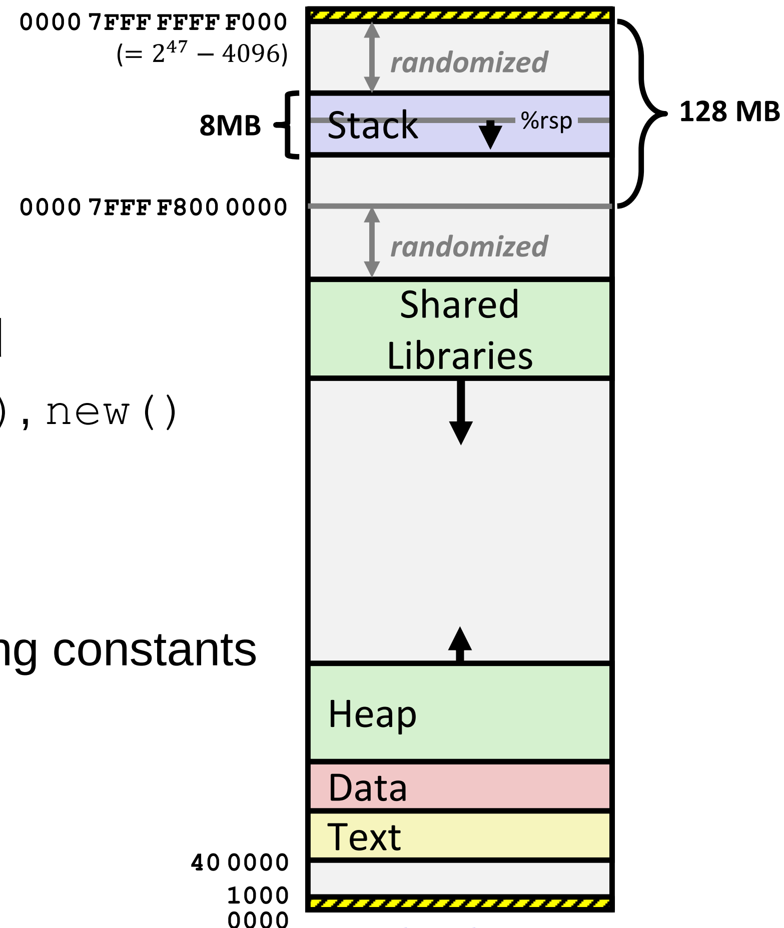
RTOS Lab

Agenda

I. Background
II. Heap
III. Dynamic Memory Allocation in Linux
IV. Garbage Collection

x86-64 Linux Memory Layout

- Stack
  - Runtime stack (8MB limit)
    - E.g., local variables
- Heap
  - Dynamically allocated as needed
    - When call `malloc()`, `calloc()`, `new()`
- Data
  - Statically allocated data
    - E.g., global vars, static vars, string constants
- Text/Shared libraries
  - Executable machine instructions
  - Read-only



I. Background

Static vs. Dynamic

- Static X
  - X is done at pre-runtime (or offline)
  - X could be analysis, synthesis, allocation, scheduling, etc.
- Dynamic X
  - X is done at runtime (or online)

Why Dynamic Allocation?

- Static allocation isn't sufficient for all…
  - Why? – Unpredictability and thus lack of space efficiency
    - Can't predict ahead of time how much memory, or in what form, will be needed
  - Example of memory request unpredictability:
    - Recursive procedures
      - Even regular procedures are hard to predict (data dependencies)
    - Complex data structures, e.g., linked lists and trees
    - Lack of space efficiency
      - If all storage must be reserved in advance (statically), then it will be used inefficiently (enough will be reserved to handle the worst possible case)
      - For example, OS doesn't know how many jobs there will be or which programs will be run

Dynamic Storage Allocation (1)

- Dynamic storage allocation is …
  - Needed both for "*main memory*" and for "*file space*" on disk
  - We'll touch upon disk space allocation in *Lecture* on *File Systems*

- Can be handled in one of two general ways
  1. "*Stack*" allocation
     - Restricted, but simple and efficient
  2. "*Heap*" allocation
     - More general, but less efficient
     - More difficult to implement

Dynamic Storage Allocation (2)

- Has two basic operations
  - Allocate and free (or deallocate)
- Stack organization
  - Memory allocation and freeing are partially predictable
    - Keeps all the free space together in one place
  - Allocation is hierarchical
    - Memory is freed in opposite order from allocation
    - E.g., alloc(A); alloc(B); alloc(C); free(C); free(B); free(A)
  - Examples
    - Function call frames, tree traversal, expression evaluation, parsing statements in program

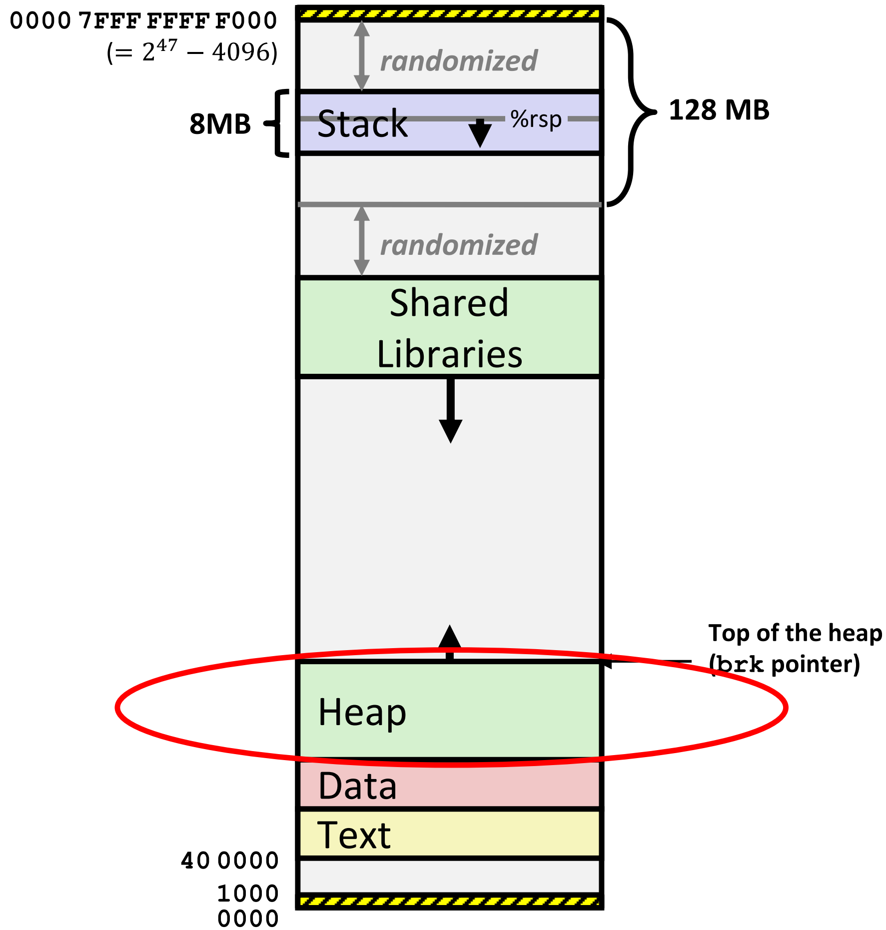
II. Heap

Why Heap?

- We've discussed two types of data allocation so far:
  - Global variables
  - Stack-allocated local variables

- Not sufficient!
  - How to allocate data whose size is only known at runtime?
    - E.g., when reading variable-sized input from network, file etc.
  - How to control the lifetime of allocated data?
    - E.g., a linked list that grows and shrinks as items are inserted/deleted

The Heap


0000 7FFF FFFF F000
(= 2^47 – 4096)
randomized
8MB
Stack
%rsp
128 MB
randomized
Shared Libraries
Top of the heap
(brk pointer)
Heap
Data
Text
40 0000
1000
0000

What is Heap?

- Heap is …
  - Simply a kind of data structure meant to be used for dynamic memory allocation
    - Can better be explained as an ADT
  - Consists of "*allocated*" and "*free*" memory areas
  - Keeps track of the list of free memory areas
    - Allocated areas are accessed thru the pointers in your program anyway
      - No need for the heap to manage them
    - The free memory areas is called "*free list*"
    - Initially, the free list has only one big memory chunk that is the entire heap

Heap Organization

- Allocation and free are unpredictable
  - Heaps are used for arbitrary list structures, complex data organizations
    - Examples: **`new`** in C++, **`malloc()`** in C
- Heap memory consists of
  - Allocated areas and free areas (AKA holes)
  - Inevitably end up with lots of holes (*fragmentation*)

Challenge

- Reuse the space in holes to keep the number of holes small, their size large
  - Hopefully, group all the holes together into one big chunk

- Fragmentation
  - Leads to inefficient use of memory due to holes that are too small to be useful
  - No problem in stack allocation
  - Causes serious performance penalties
    - Drastic slowdown of smartphones after a long use
  - Anti-fragmentation approaches
    - *Buddy allocator*, *slab allocator*, *paging*, etc.

Anti-Fragmentation in Linux (1)

- Buddy allocator
  - Divides memory into partitions to try to satisfy a memory request as suitably as possible
    - Splits memory into halves to try to give a best-fit
    - Invented in 1963, Harry Markowitz
  - Effectively reduces external fragmentation with small compaction overhead

Harry Markowitz
American economist who won Nobel Memorial Prize in Economic Sciences

Anti-Fragmentation in Linux (2)

- Slab allocator
  - Caching frequently allocating and de-allocating data structures
  - Object creation and deletion are widely employed by the kernel which outweigh the cost of allocating memory

Free List Management (1)

- Free list is …
  - A list made by heap allocation schemes
    to keep track of the memory that is not in use
  - Algorithms differ in how they manage the free list
    - How to find a free area that suits for the allocation request?
    - What kind of data structure should be used for the free list?

Free List Management (2)

- Finding a free area
  - *Best-fit*
    - Keeps the linked list of free memory blocks
    - Search the whole list on each allocation
    - Choose the block that comes closest to matching the needs of the allocation
    - During release operations, merge adjacent free blocks
  - *First-fit*
    - Just scans the list for the first hole that is large enough
    - Also merge on releases
    - Most first-fit implementations are rotating first-fit

Free List Management (3)

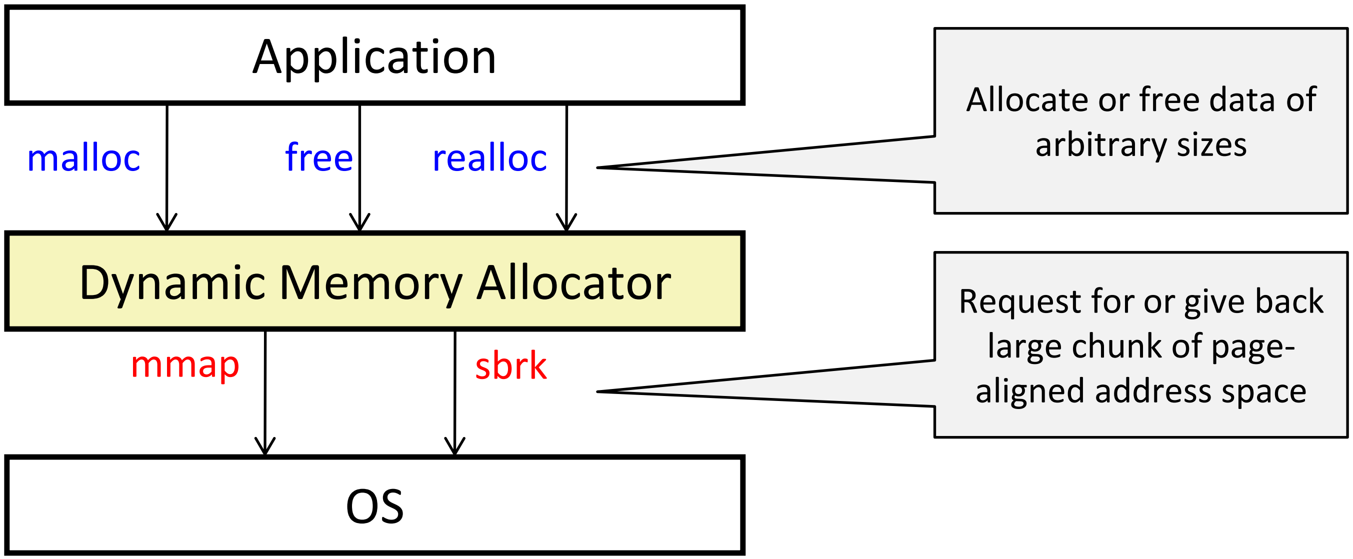
- Finding a free area (cont'd)
  - Best-fit is not necessarily better than first-fit
    - Suppose memory contains 2 free blocks of size 20 and 15
    - Suppose allocation ops are 10 then 20
    - Suppose ops are 8, 12, then 12
  - First-fit tends to leave "*average*" size holes while best-fit tends to leave some very large ones, some very small ones
    - The very small ones can't be used very easily
  - How about *Worst-fit*?

Free List Management (4)

- Data structures
  - Bitmap
    - Used for allocation of storage that comes in fixed-size chunks
    - Examples: disk blocks 32-byte chunks
    - Keep a large array of bits, one for each chunk
    - If bit is 0, it means chunk is in use
    - If bit is 1, bit means chunk is free
  - Segregated free list (seglist)
    - Keep a separate free list for each popular size
    - Allocation is fast, no fragmentation
    - May get some inefficiency if some lists run out while other lists have lots of free blocks
      - Get shuffled between pools

Implementation (1)

- Dynamic memory allocator
  - Part of user-level library
    - Why not implement its functionality in the kernel?



Implementation (2)

- Changing heap size

```
#include <unistd.h>
void *sbrk(int incr);
```

  - Adds **`incr`** bytes to the break value (i.e., brk pointer) and changes the allocated space accordingly
  - If **`incr`** is negative, the amount of allocated space is decreased by **`incr`** bytes
  - Returns the new value of the **`brk`** pointer

Implementation (3)

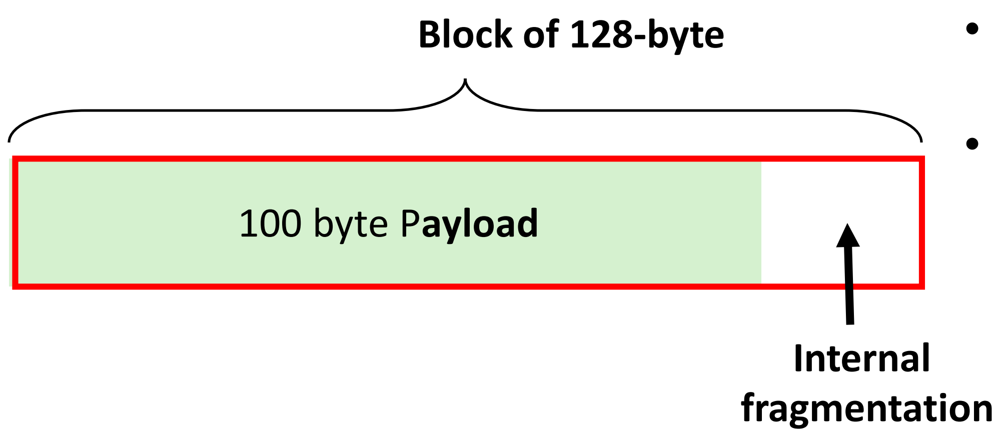
- Challenges facing a memory allocator
  - Achieve good *memory utilization*
    - Apps issue arbitrary sequence of malloc/free requests of arbitrary sizes
    - *Utilization* = sum of malloc'd data / size of heap
  - Achieve good *performance*
    - malloc/free calls should return quickly
    - *Throughput* = # ops/sec
  - Constraints:
    - Cannot touch/modify malloc'd memory
    - Can't move the allocated blocks once they are malloc'd
      - I.e., compaction is not allowed

Implementation (4)

- Fragmentation
  - Source of poor memory utilization
    - Internal fragmentation
    - External fragmentation

Implementation (5)

- Internal fragmentation
  - Malloc allocates data from blocks of certain sizes
  - Occurs if payload is smaller than block size

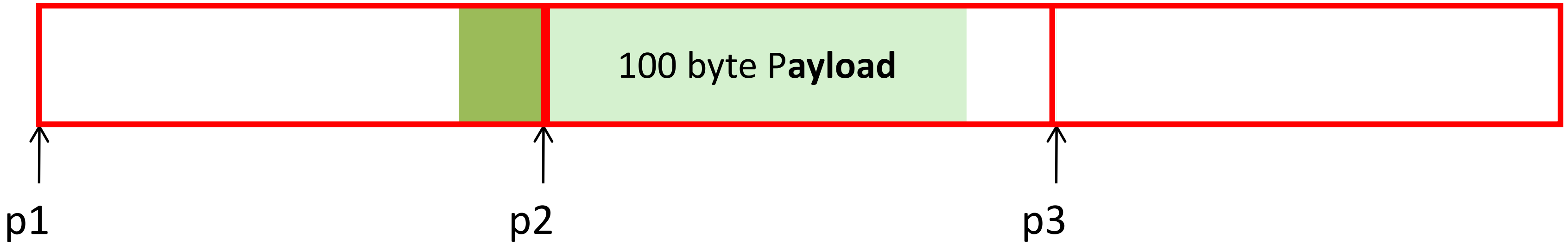


- Block size decided by allocator's designer
- Payload is the number of bytes you want when you call malloc()

  - May be caused by
    - Limited choices of block sizes
    - Padding for alignment purposes
    - Other space overheads

Implementation (6)

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



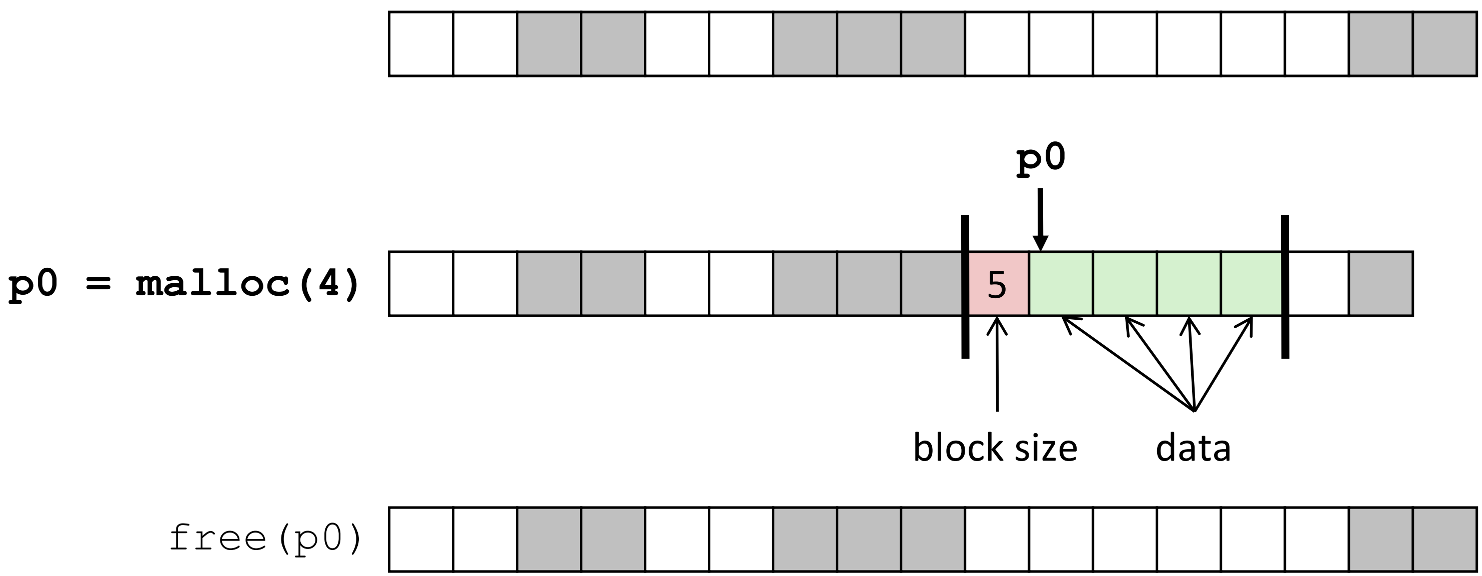

```
P1 = malloc(100);
p2 = malloc(100);
p3 = malloc(100);
free(p1);
free(p3);
malloc(200)?
```

Heap Design Choices

- Questions to answer
  1. How do we know how much memory to free given just a pointer?
  2. How do we keep track of the free blocks?
  3. What do we do with the extra space when allocating a space that is smaller than the free block it is placed in?
  4. How do we pick a block to use for allocation
     - Many might fit?
  5. How do we reinsert freed block?

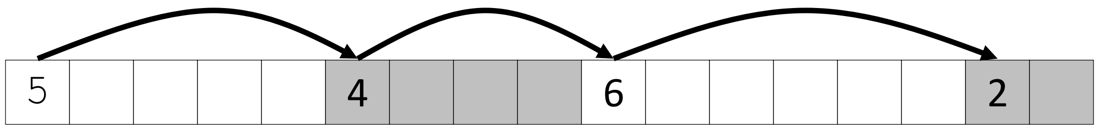
Q1. Knowing How Much to Free

- Standard method
  - Keep the length of a block in the *header field* preceding the block
    - Requires header overhead for every allocated block

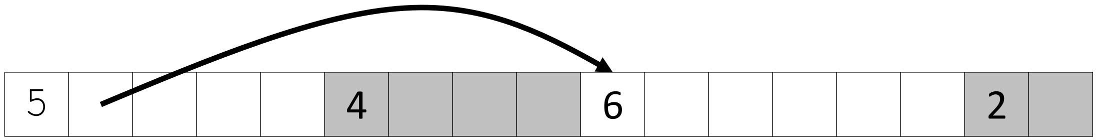


Q2. Keeping Track of Free Blocks

- Method 1: *Implicit list* using length
  - Links all blocks



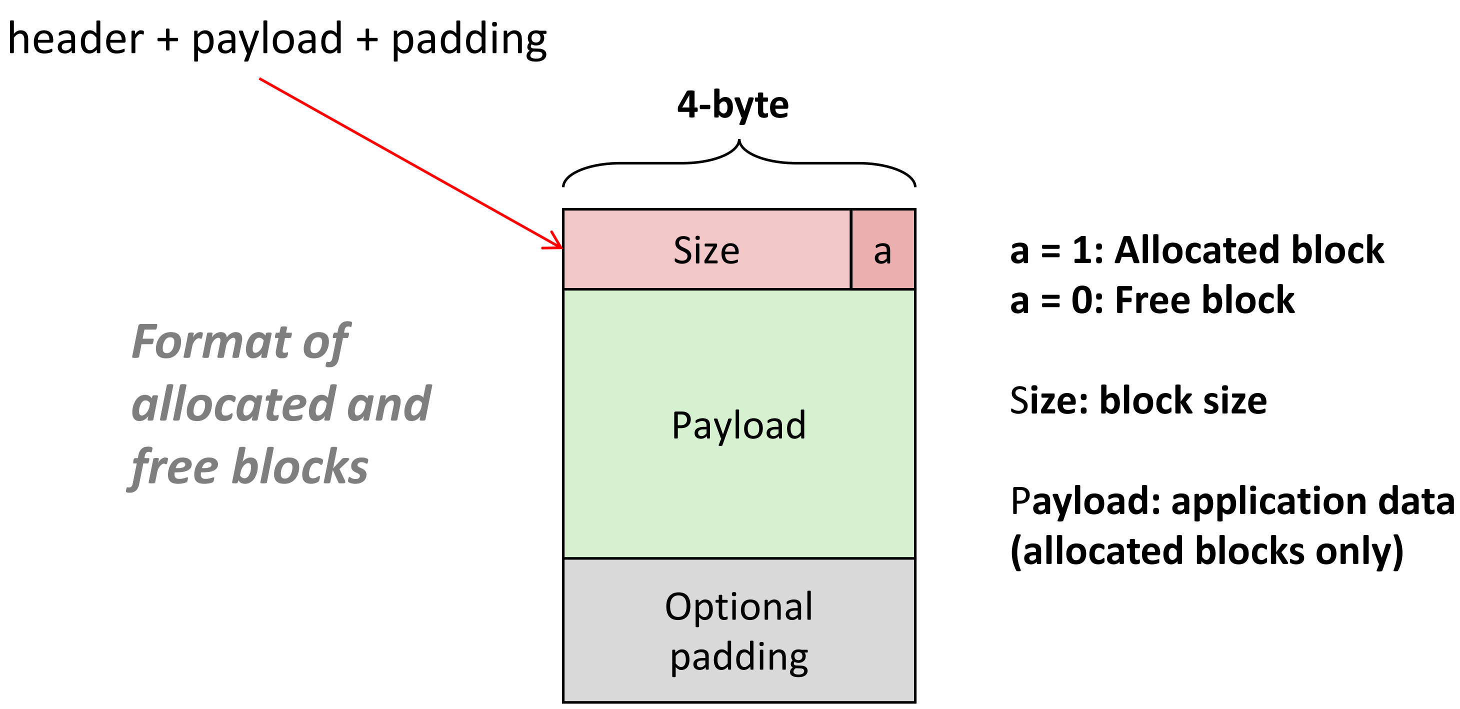
- Method 2: *Explicit list* among the free blocks using pointers



- Method 3: *Segregated free list* (*seglist*)
  - Different free lists for different size classes

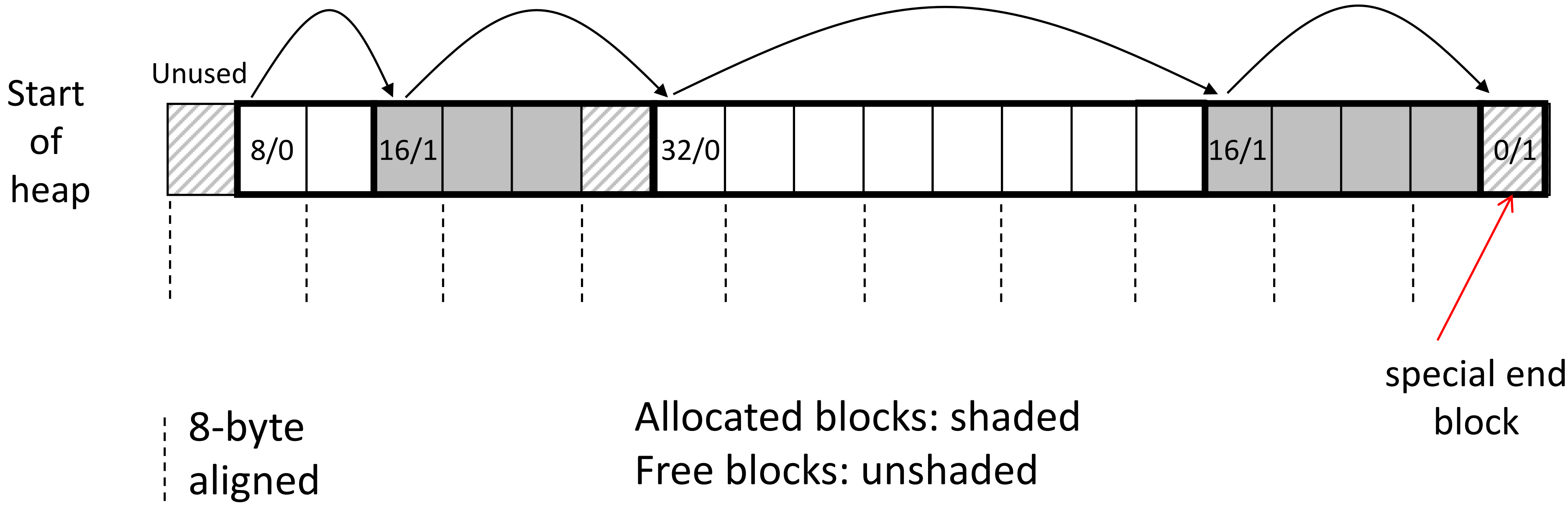
Method 1: Implicit List (1)

- Heap is divided into variable-sized blocks
- Each block has size and allocation status



Method 1: Implicit List (2)

- Detailed example



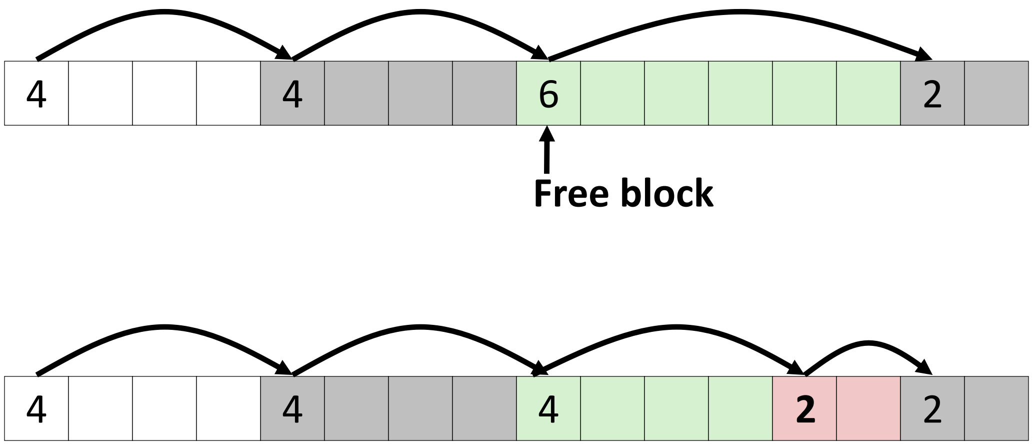
Allocated blocks: shaded
Free blocks: unshaded
Headers: labeled with size in bytes/allocated bit
Each square represents 4 bytes

Method 1: Implicit List (3)

- Q4. Finding a free block
  - *First-fit*:
    - Search from the beginning, choose the first free block that fits
  - *Next-fit*:
    - Like first-fit, except search starts where previous search finished
  - *Best-fit*:
    - Search the list, choose the best free block: fits, with fewest bytes left over (i.e., pick the smallest block that is big enough for the payload)
    - Keeps fragments small
    - Will typically run slower than first-fit

Method 1: Implicit List (4)

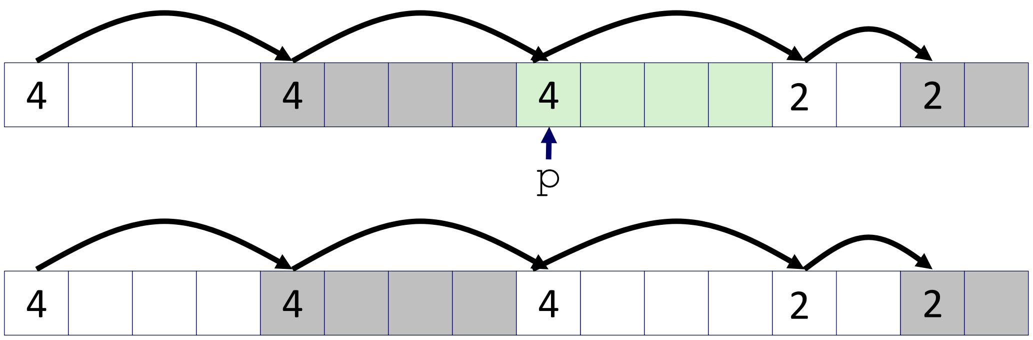
- Q3. Allocating in a free block: *splitting*
  - Since allocated space might be smaller than free space, we might want to split the block



After malloc(4)

Method 1: Implicit List (5)

- Q5. Freeing a free block with no coalescing
  - Simplest implementation:
    - Need only clear the “allocated” flag
    - But can lead to “*false fragmentation*”

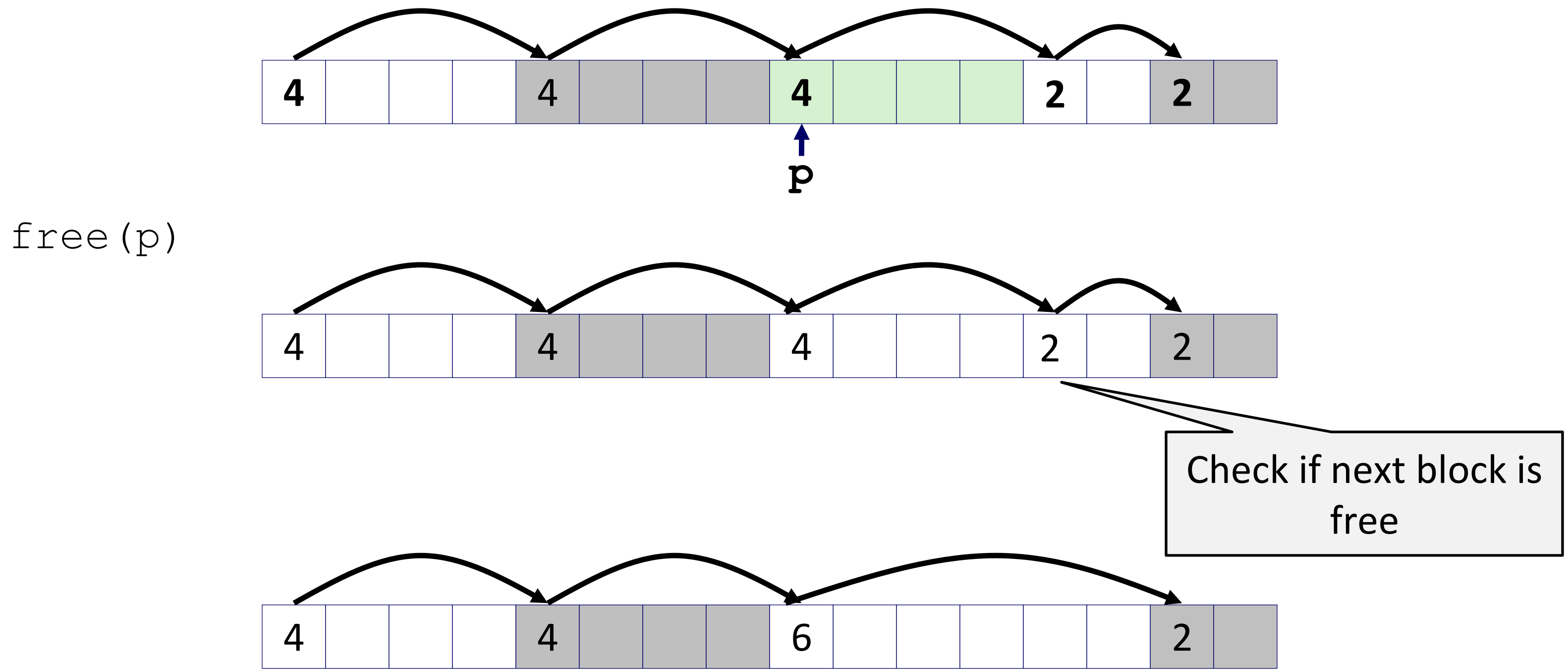


`free(p)`

`malloc(5)` ***Oops!***

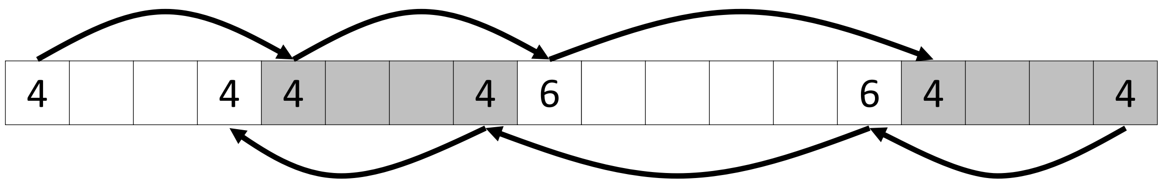
Method 1: Implicit List (6)

- Q5. Freeing a free block with coalescing
  - Join (coalesce) with next/previous blocks, if they are free
    - Coalescing with the next block

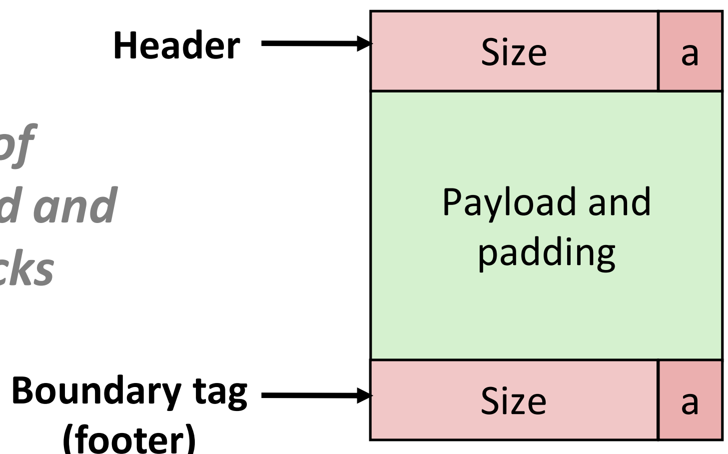


Method 1: Implicit List (7)

- Q5. Freeing a free block with bidirectional coalescing
  - *Boundary tags* [Knuth73]
    - Replicate size/allocated header at “bottom” (end) of blocks
      - Allows us to traverse the “list” backward, but requires extra space



Format of allocated and free blocks

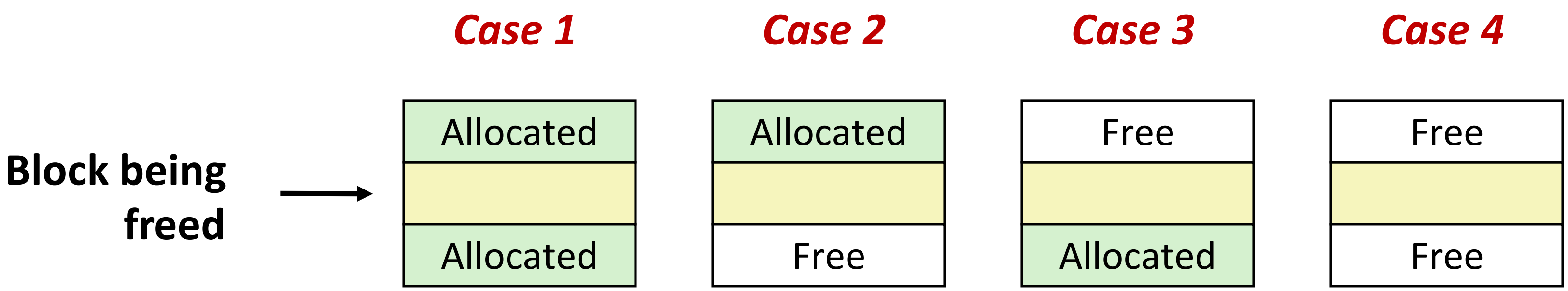


a = 1: Allocated block
a = 0: Free block

Size: Total block size

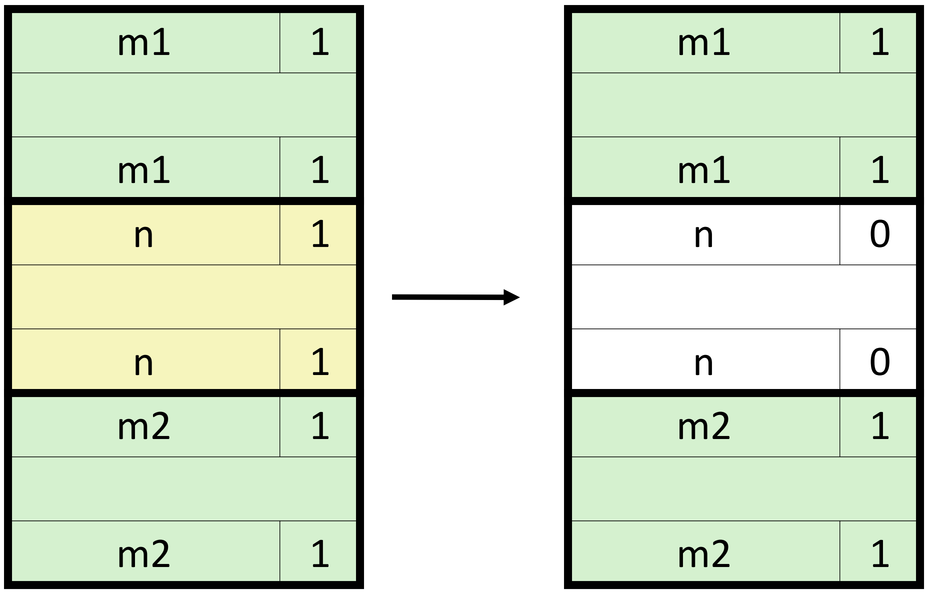
Method 1: Implicit List (8)

- Four cases of coalescing



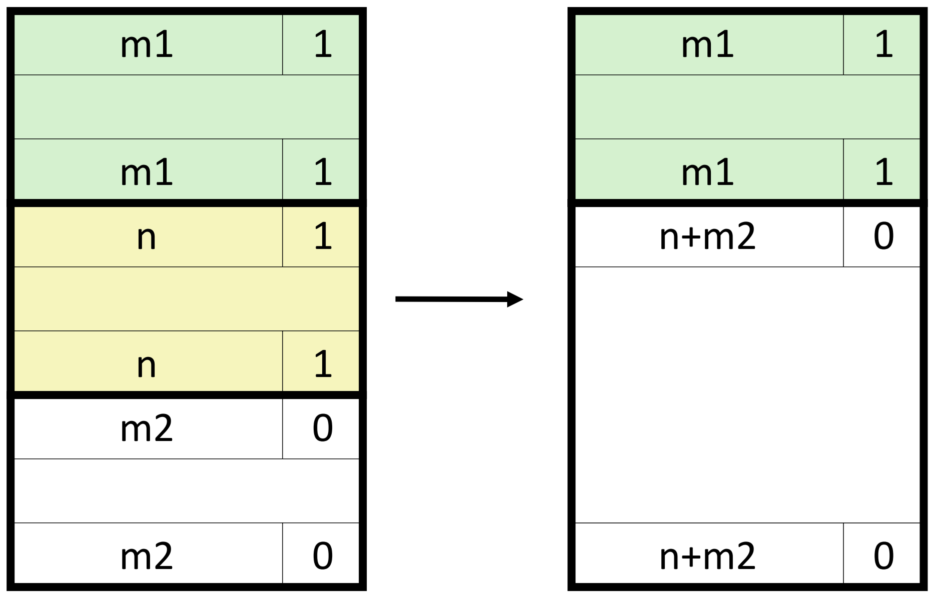
Method 1: Implicit List (9)

- Case 1



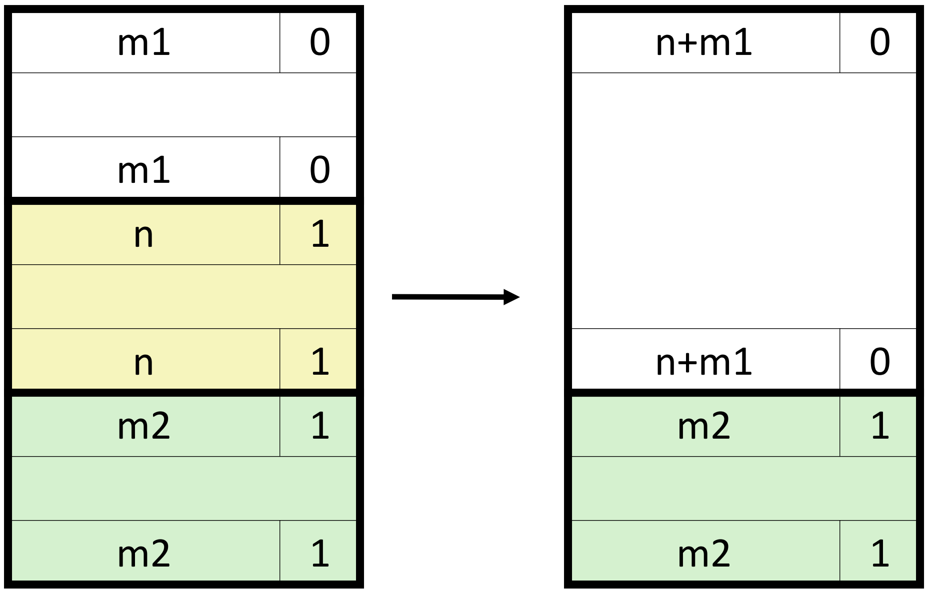
Method 1: Implicit List (10)

- Case 2



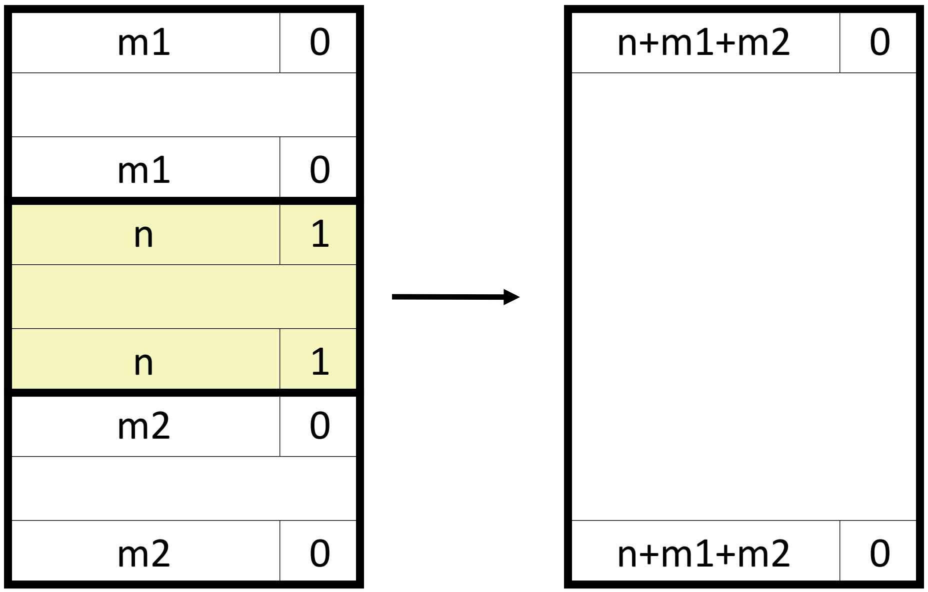
Method 1: Implicit List (11)

- Case 3



Method 1: Implicit List (12)

- Case 4



Method 1: Implicit List (13)

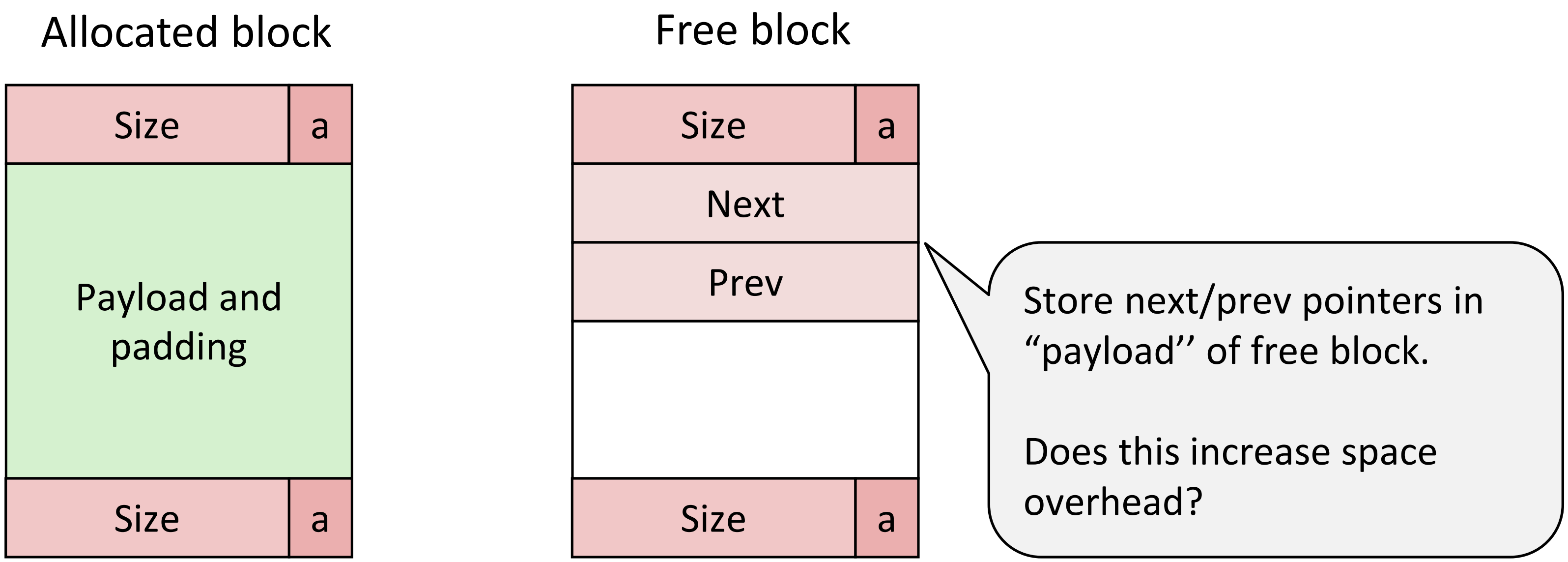
- When to coalesce?
  - *Immediate* coalescing:
    - Coalesce each time **`free()`** is called
  - *Deferred* coalescing:
    - Try to improve the performance of free by deferring coalescing until needed
    - Examples:
      - Coalesce as you scan the free list for malloc()
      - Coalesce when the amount of external fragmentation reaches some threshold

Method 1: Implicit List (14)

- Summary
  - Implementation: very simple
  - Allocate cost:
    - Linear time worst case
  - Free cost:
    - Constant time worst case, even with coalescing
  - Memory usage:
    - Will depend on first-fit, next-fit or best-fit
  - Not used in practice for **`malloc`**/**`free`** because of linear-time allocation
    - Used in many special-purpose applications

Method 2: Explicit List (1)

- Maintain list(s) of free blocks instead of all blocks
- Need to store forward/back pointers in each free block, not just sizes
  - Because free blocks may not be contiguous in heap



Method 2: Explicit List (2)

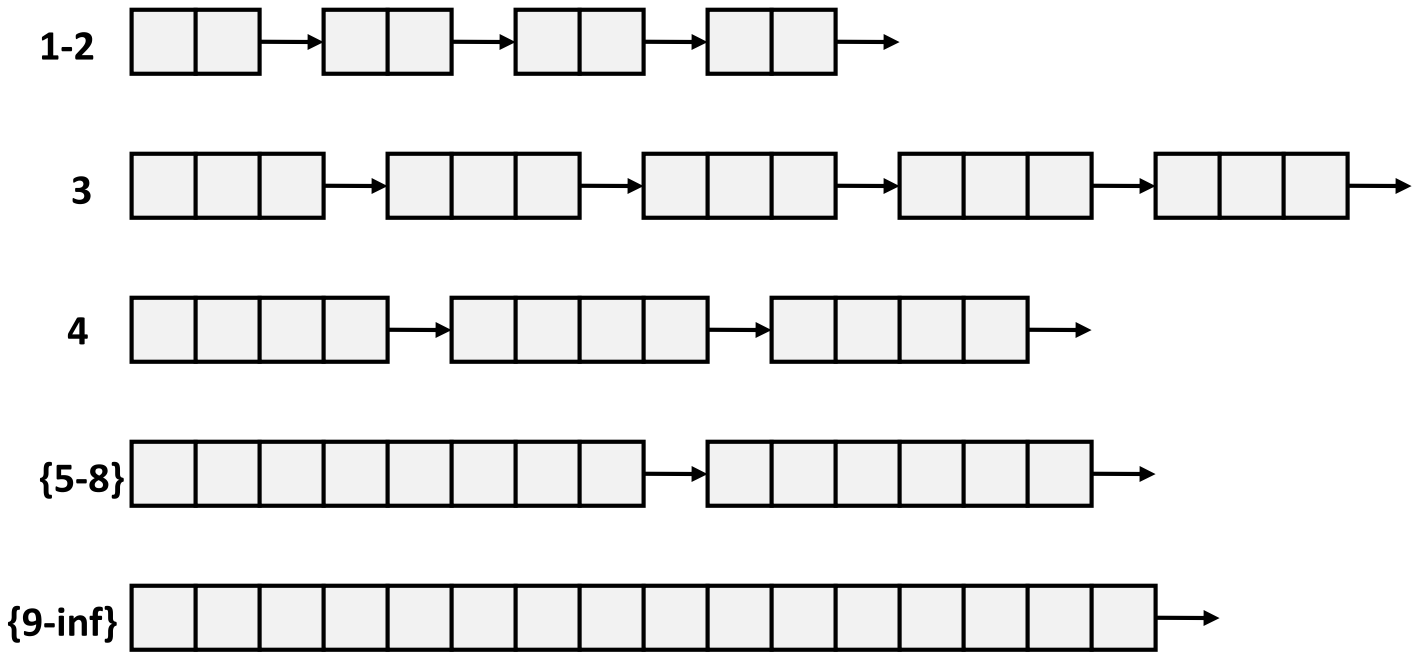
- Where in the free list to put a newly freed block?
  - Insert freed block at the beginning of the free list (LIFO)
    - Pro: simple and constant time
  - Insert freed blocks to maintain address order:
    - *addr(prev)* < *addr(curr)* < *addr(next)*
    - Pro: may lead to less fragmentation than LIFO

Method 2: Explicit List (3)

- Summary
  - Allocation is linear time in # of free blocks instead of all blocks
  - Still expensive to find a free block that fits
  - How about keeping multiple linked lists of different size classes?

Method 3: Segregated List (1)

- Seglist
  - Multiple free lists each linking free blocks of similar sizes



Method 3: Segregated List (2)

- Seglist
  - Given an array of free lists, each one for some size class
  - To allocate a block of size *n*:
    - Search in appropriate free list containing size *n*
    - Split found block and place fragment on appropriate list
    - Try next larger class if no blocks found
  - If no block is found:
    - Request additional heap memory from OS
    - Allocate block of *n* bytes from this new memory
    - Place the remainder as a single free block in the largest size class

Method 3: Segregated List (3)

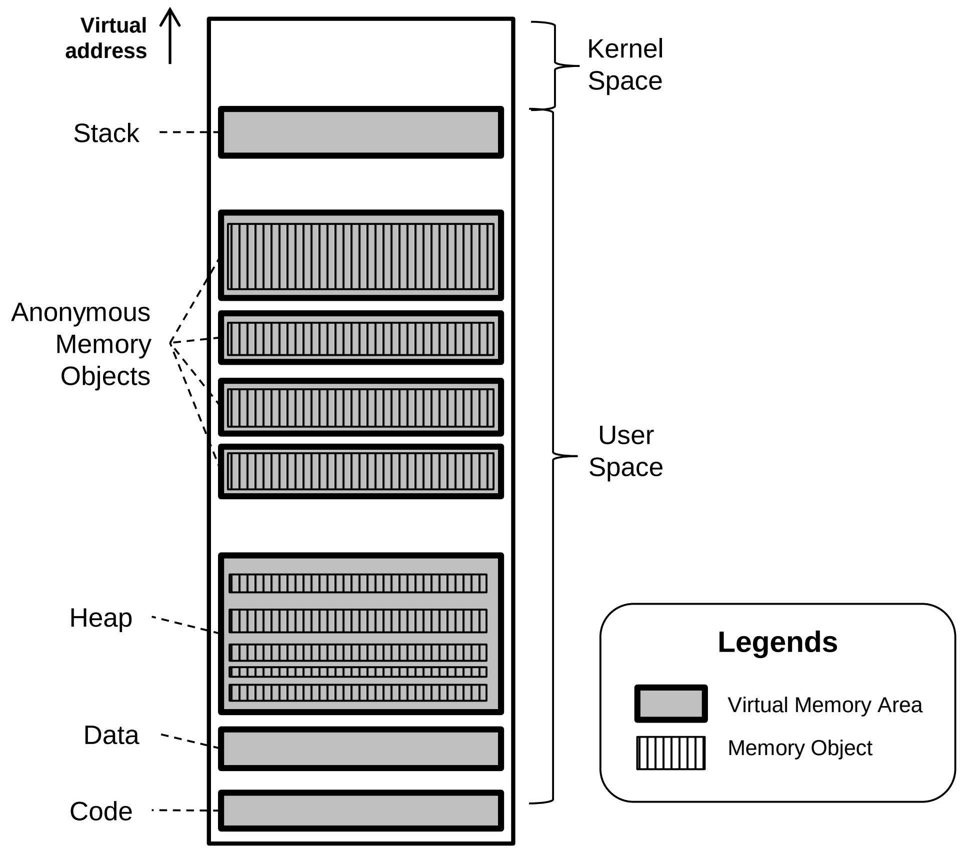
- Seglist
  - To free a block:
    - Coalesce and place on the appropriate list
  - Advantages of seglist allocators
    - Fast allocation
    - Better memory utilization
      - First-fit search of segregated free list approximates a best-fit search of the entire heap

III. Dynamic Memory Allocation in Linux

VMA and Memory Objects

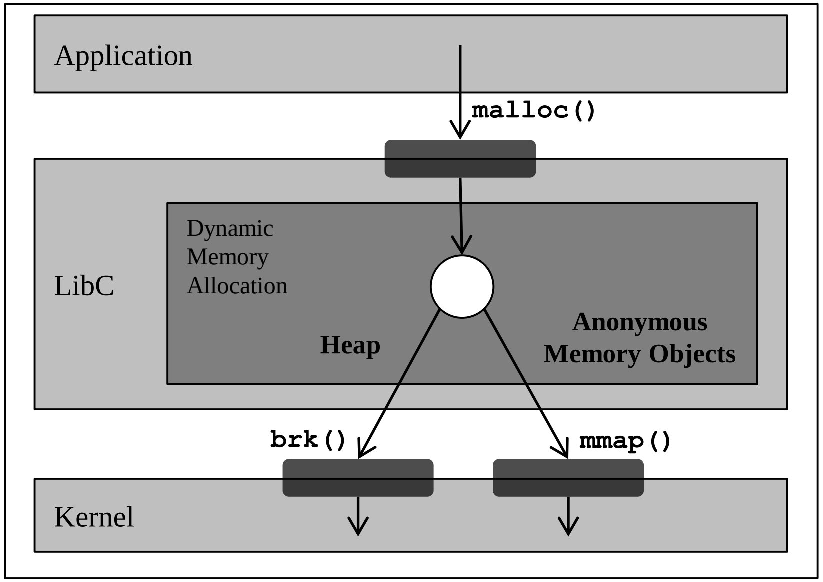
- Virtual memory area (VMA)
  - Logical memory region
    that consists of a set of contiguous pages
  - Unit of virtual memory management in the Linux kernel
  - Created by an **`mmap()`** system call
- Memory object
  - Unit of dynamic memory allocation
  - Created by **`malloc()`** call

Virtual Address Space Layout



`malloc()` in Linux

- `malloc()` is serviced differently according to the size of the requested memory object
  - Heap vs. anonymous memory object (AMO)



Heap vs. Anonymous Memory Object

- Heap
  - Heap is used when the size of requested memory object is smaller than **`mmap`** threshold
  - There is only one heap for each process
  - Heap is the VMA created by **`mmap()`** during the process is created by **`fork()`**

Heap vs. Anonymous Memory Object

- Anonymous memory object (AMO)
  - Anonymous **`mmap()`** is invoked when the size of requested memory object is equal to or greater than mmap threshold
    - An independent VMA serves exactly one AMO
    - Anonymous **`mmap()`** creates a VMA which consists of anonymous pages
  - When an anonymous page is created
    - It has *neither* a page table entry *nor* a physical frame yet
    - These are allocated to the page later via a *minor page fault*

Two Types of Page Faults

- Major page fault
  - Page fault that incurs *page mapping* and *page read* (*disk I/O*) from disk
- Minor page fault
  - Page fault that incurs page mapping *only* without disk I/O
  - Used for anonymous pages
  - Steps for minor page fault handling

1. Process attempts to access an anonymous page
↓
2. If a corresponding page table entry is empty, then a minor page fault occurs
↓
3. Get a physical frame from dynamic memory allocator of kernel
↓
4. Restart the instruction that was interrupted by the page fault

Why Two Mechanisms for Servicing One API (`malloc`)?

- Trade-off between memory efficiency & performance
  - Heap
    - Good for performance because page mappings of a memory object can be reused after it is **`free()`**-ed (avoids minor page faults)
    - Bad for memory efficiency (physical frames of **`free()`**-ed memory objects cannot be returned to the kernel)
  - AMO
    - Bad for performance because page mappings of a memory object cannot be reused
    - Good for memory efficiency (physical frames of **`free()`**-ed memory objects are immediately returned to the kernel)

In Memory-Constrained Devices

- AMOs are favored over heap
  - Typically, the **`mmap`** threshold in smartphones is much smaller than that of desktops/servers
    - In Android Jellybean, **`mmap`** threshold = 64 KB (16 pages)
    - In Ubuntu 12.04 (64 bits), **`mmap`** threshold = 4 MB

IV. Garbage Collection

Why Garbage Collection? (1)

- Memory reclamation
  - Act of collecting and freeing unused memory
  - Very important in dynamic memory management
  - How do you know when dynamically allocated memory can be freed?
    - It's easy when the chunk is used only in one place
    - Reclamation is hard when the chunk is shared
      - It can't be recycled until all the sharers are finished
      - Sharing is indicated by the presence of pointers to the chunk
      - Without a pointer, can't access or can't find it, anyway

Why Garbage Collection? (2)

- Memory reclamation (cont'd)
  - What will happen if unused memory is not reclaimed?
    - *Memory leak*
  - Who should perform memory reclamation?
  - How about *automatic* memory reclamation, instead of manual one?
    - *Automatic garbage collection*

Automatic Garbage Collection (1)

- Garbage is …
  - Data objects in program that can't be accessed in the future
- Garbage collection (from Wikipedia)
  - Attempts to reclaim garbage
  - A form of *automatic* memory management in comparison with manual management
  - Invented by John McCarthy around 1959 to solve problems in Lisp

Automatic Garbage Collection (2)

- How GC works
  - When available memory goes low,
    the garbage collector searches through all of the pointers and collects unused or unreached data objects
    - Must be able to find as many pointers as possible in code

- Pros and cons
  - Makes life easier on application programmers
  - Garbage collectors are difficult to program and debug, especially if compaction is also done

Automatic Garbage Collection (3)

- GC and programming language support
  - GC must have the ability to find pointer variables in code
    - Needs runtime supports from your programming language
  - There even exist garbage collected languages
    - They require GC to be part of the language specification
    - E.g., Java, Python, C#, most scripting languages

Automatic Garbage Collection (4)

- Benefits of GC
  - Can eliminate certain types of potentially serious bugs
    - *Memory leaks*
      - Program fails to free memory occupied by objects that have become unreachable, which can lead to memory exhaustion
      - Can causes shutdown of essential national infrastructures like telephony switching systems
    - *Dangling pointers bugs*
      - A piece of memory is freed while there are still pointers to it, and one of those pointers is dereferenced
      - By then the memory may have been re-assigned to another use, with unpredictable results
    - *Double free bugs*
      - Program tries to free a region of memory that has already been freed, and perhaps already been allocated again

Finding Garbage (1)

- Mark-and-sweep approach
  - Preconditions
    - Must be able to find all objects
    - Must be able to find all pointers to objects
  - Pass 1: *Mark*
    - Go through all global and local variables, looking for pointers
    - Mark each object pointed to and recursively mark all objects it points to
    - Compiler has to cooperate by saving information about where the pointers are stored
  - Pass 2: *Sweep*
    - Go through all objects, free up those that aren't marked

Finding Garbage (2)

- *Reference counter* approach
  - Preconditions
    - Must be able to find all objects
    - Must be able to find all pointers to objects
  - Operations
    - Keep track of the number of outstanding pointers to each chunk of allocated memory
    - When it goes to zero, free the memory
  - Reference counters must be managed carefully by the GC
    - No mistakes during incrementing and decrementing them
    - Problems with circular data structures
  - Example: File descriptors in Unix

Problems with Garbage Collection

- Garbage collection is often expensive
  - 20% or more of all CPU time in systems that use it
- Resulting in stalls scattered throughout a session
  - The moment when the garbage is actually collected can be unpredictable
  - Unpredictable stalls can be unacceptable in real-time environments, in transaction processing, or in interactive programs