CSCI-UA.0201

Computer Systems Organization

Memory Management – Dynamic Allocation

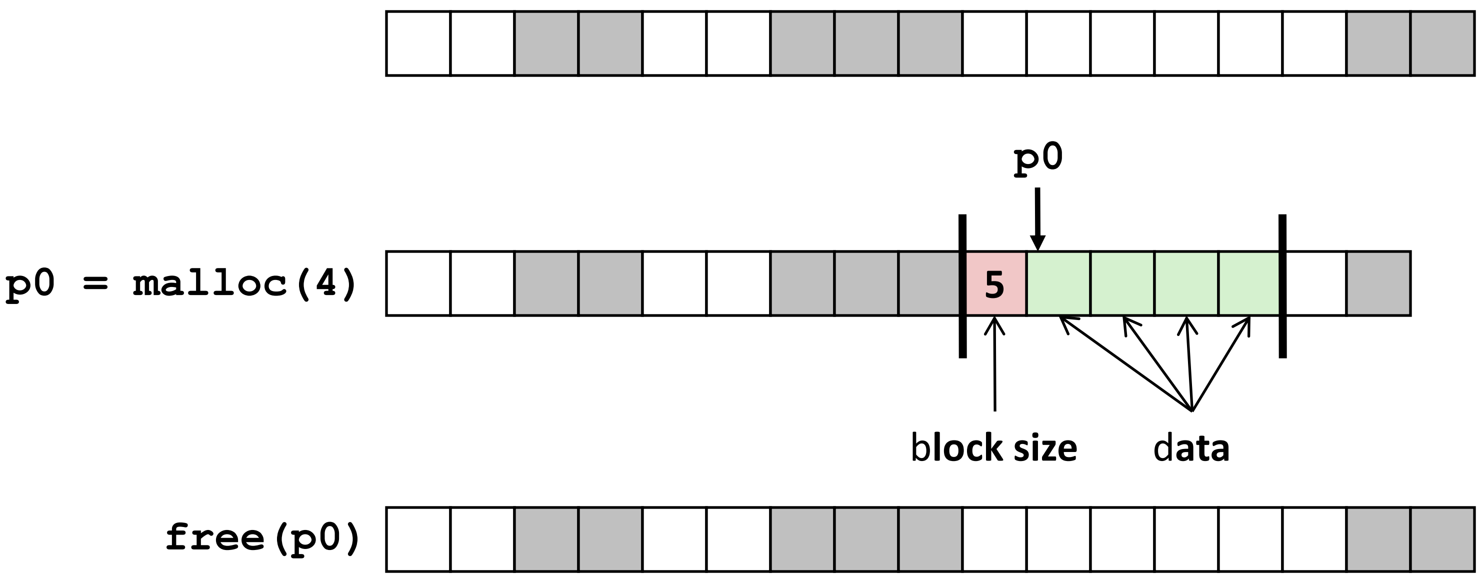
Thomas Wies
wies@cs.nyu.edu
https://cs.nyu.edu/wies

Malloc design choices

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

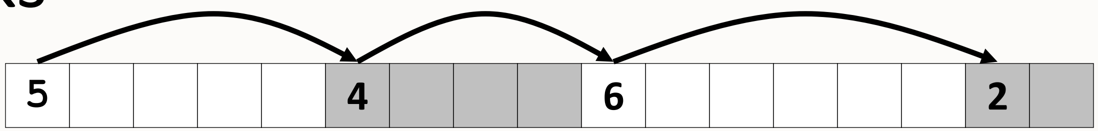
Knowing How Much to Free

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



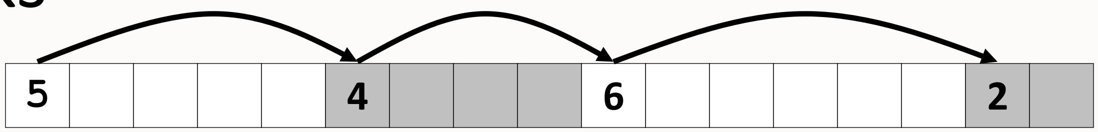
Keeping Track of Free Blocks

- Method 1: *Implicit list* using length—links all blocks

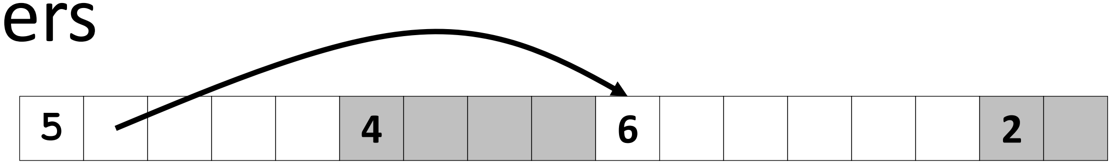


Keeping Track of Free Blocks

- Method 1: *Implicit list* using length—links all blocks

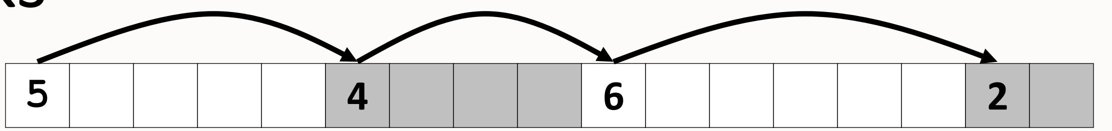


- Method 2: *Explicit list* among the free blocks using pointers

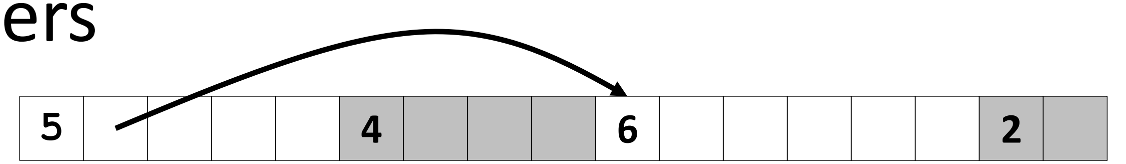


Keeping Track of Free Blocks

- Method 1: *Implicit list* using length—links all blocks



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

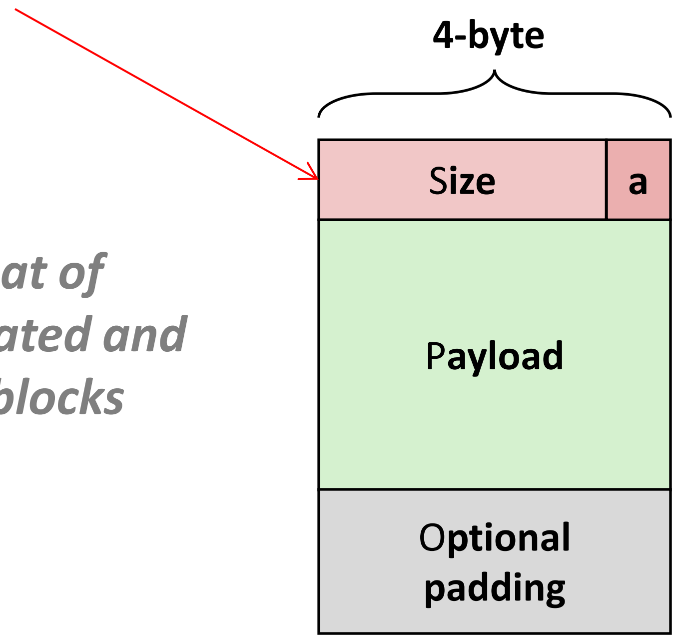


- Method 3: *Segregated free list*
  - Different free lists for different size classes

Method 1: Implicit List

- Malloc grows a contiguous region of heap by calling sbrk()
- Heap is divided into variable-sized blocks
- For each block, we need both size and allocation status

header + payload + padding



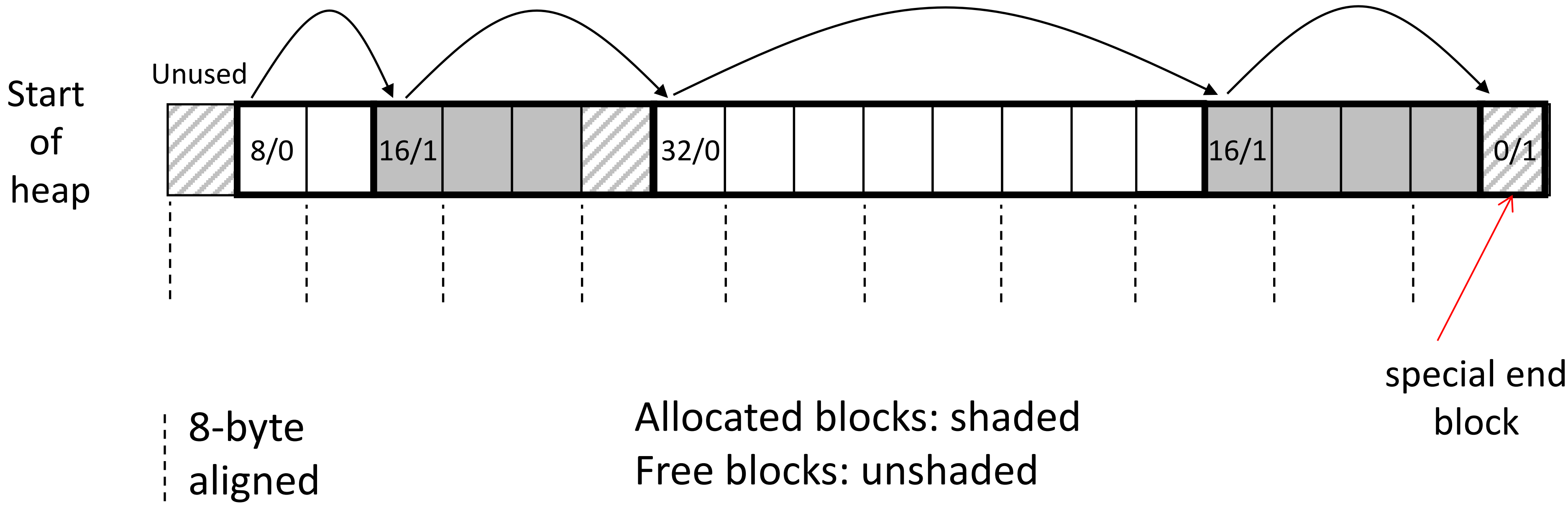
Format of allocated and free blocks

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

Size: block size

Payload: application data (allocated blocks only)

Detailed Implicit Free List Example



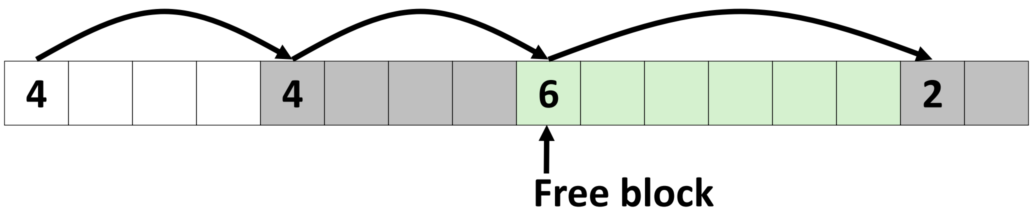
Allocated blocks: shaded
Free blocks: unshaded
Headers: labeled with size in bytes/allocated bit
Each square represents 4 bytes

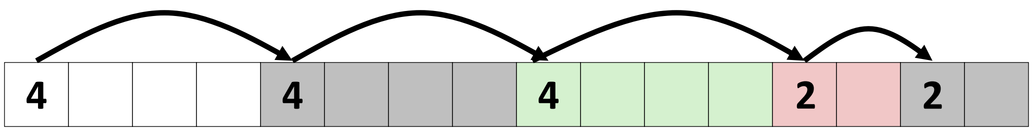
Implicit List: Finding a Free Block

- *First fit:*
  - Search from beginning, choose ***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

Implicit List: Allocating in Free Block

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

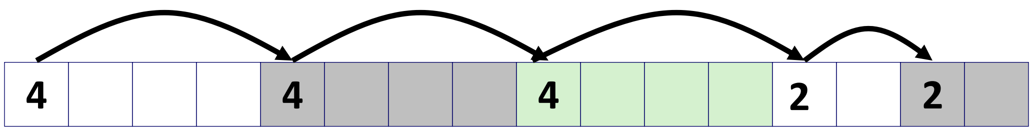




After malloc(4)

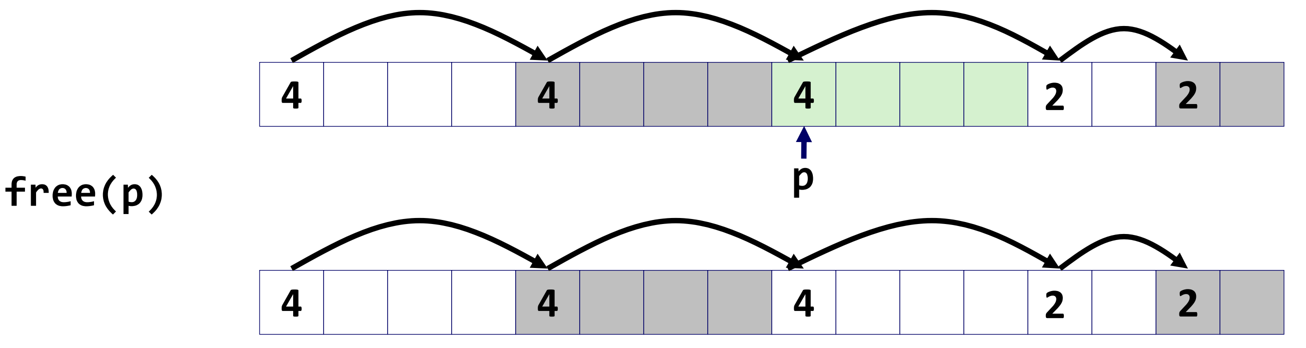
Implicit List: Freeing a Block

- Simplest implementation:
  - Need only clear the "allocated" flag
  - But can lead to "false fragmentation"



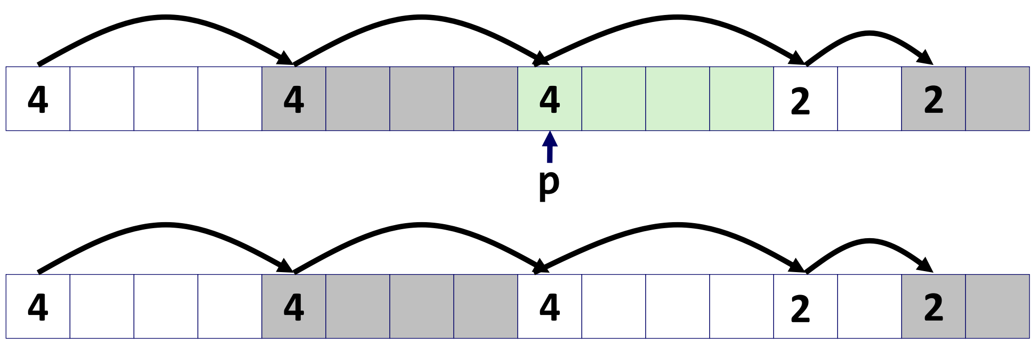
Implicit List: Freeing a Block

- Simplest implementation:
  - Need only clear the "allocated" flag
  - But can lead to "false fragmentation"



Implicit List: Freeing a Block

- Simplest implementation:
  - Need only clear the "allocated" flag
  - But can lead to "false fragmentation"

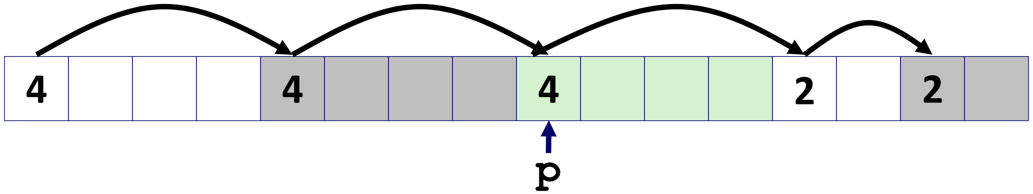


`free(p)`

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

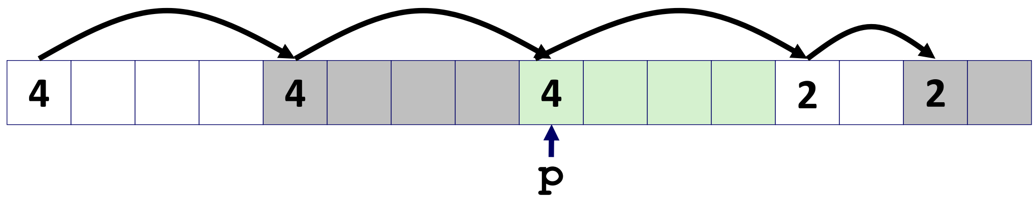
Implicit List: Coalescing

- Join *(coalesce)* with next/previous blocks, if they are free
  - Coalescing with next block

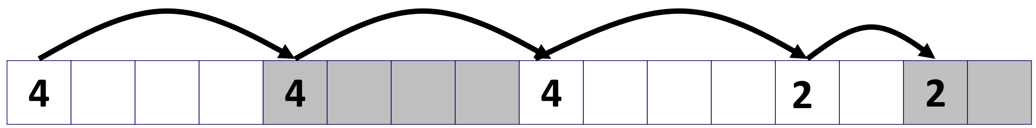


Implicit List: Coalescing

- Join *(coalesce)* with next/previous blocks, if they are free
  - Coalescing with next block

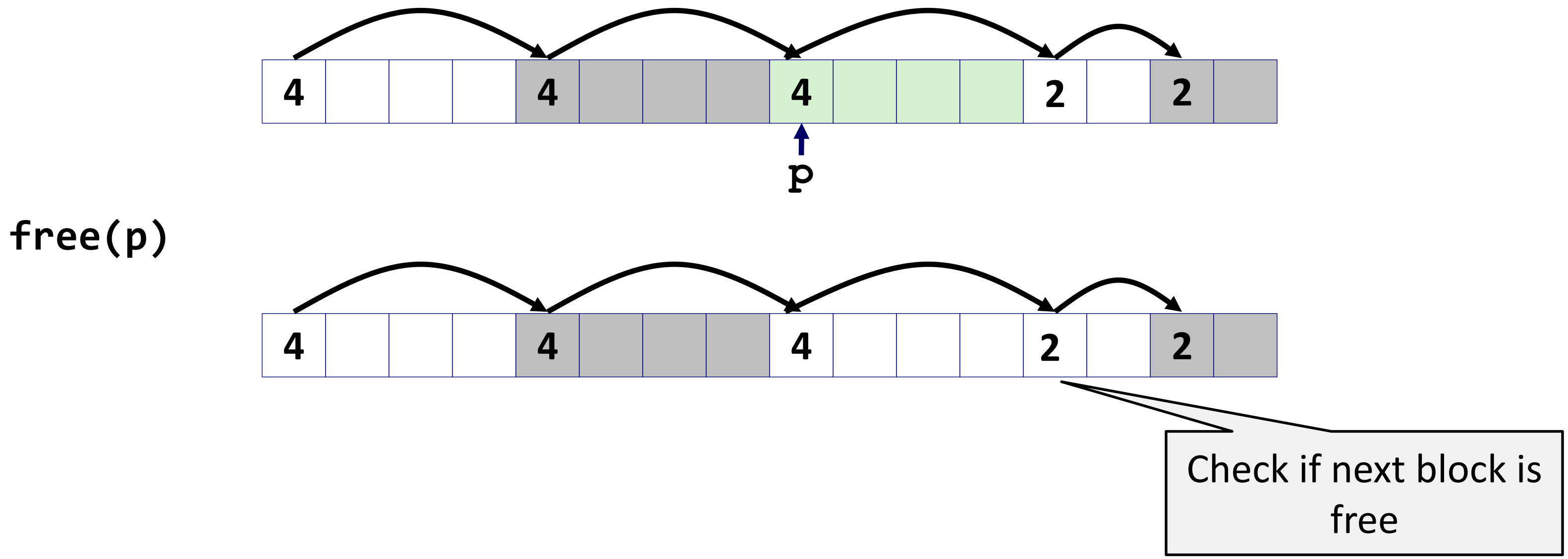


`free(p)`



Implicit List: Coalescing

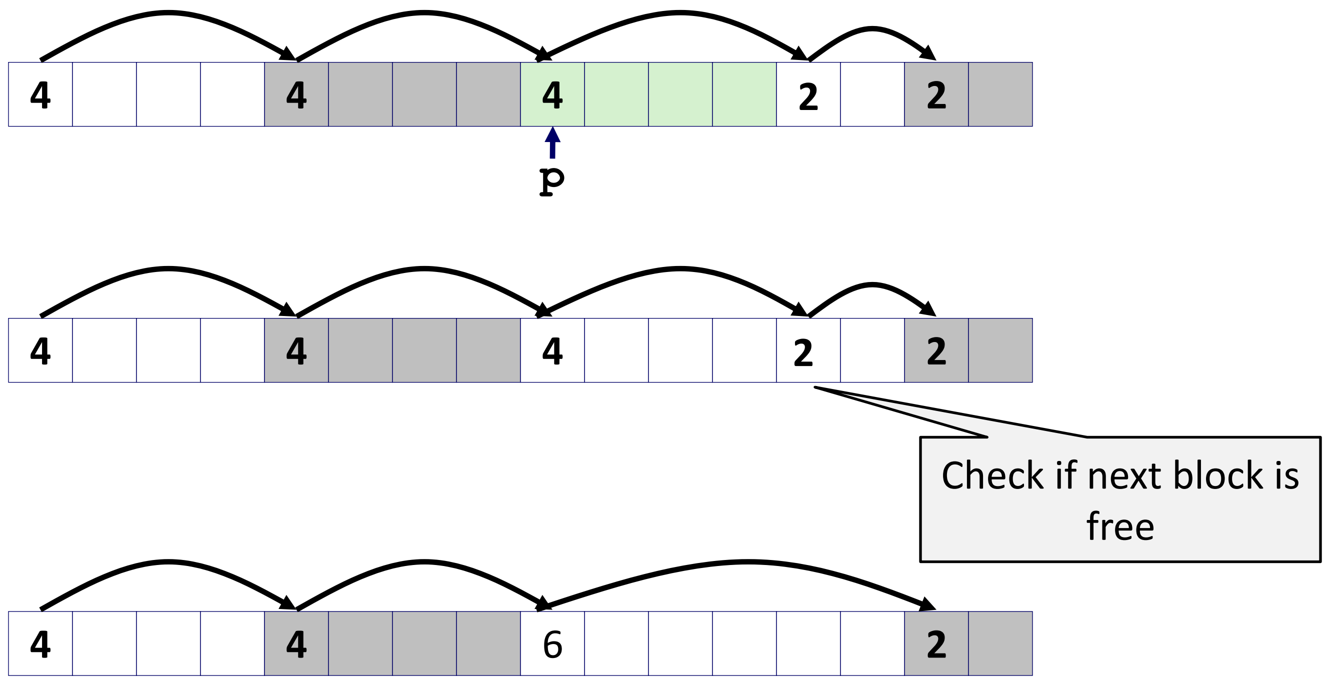
- Join *(coalesce)* with next/previous blocks, if they are free
  - Coalescing with next block



Implicit List: Coalescing

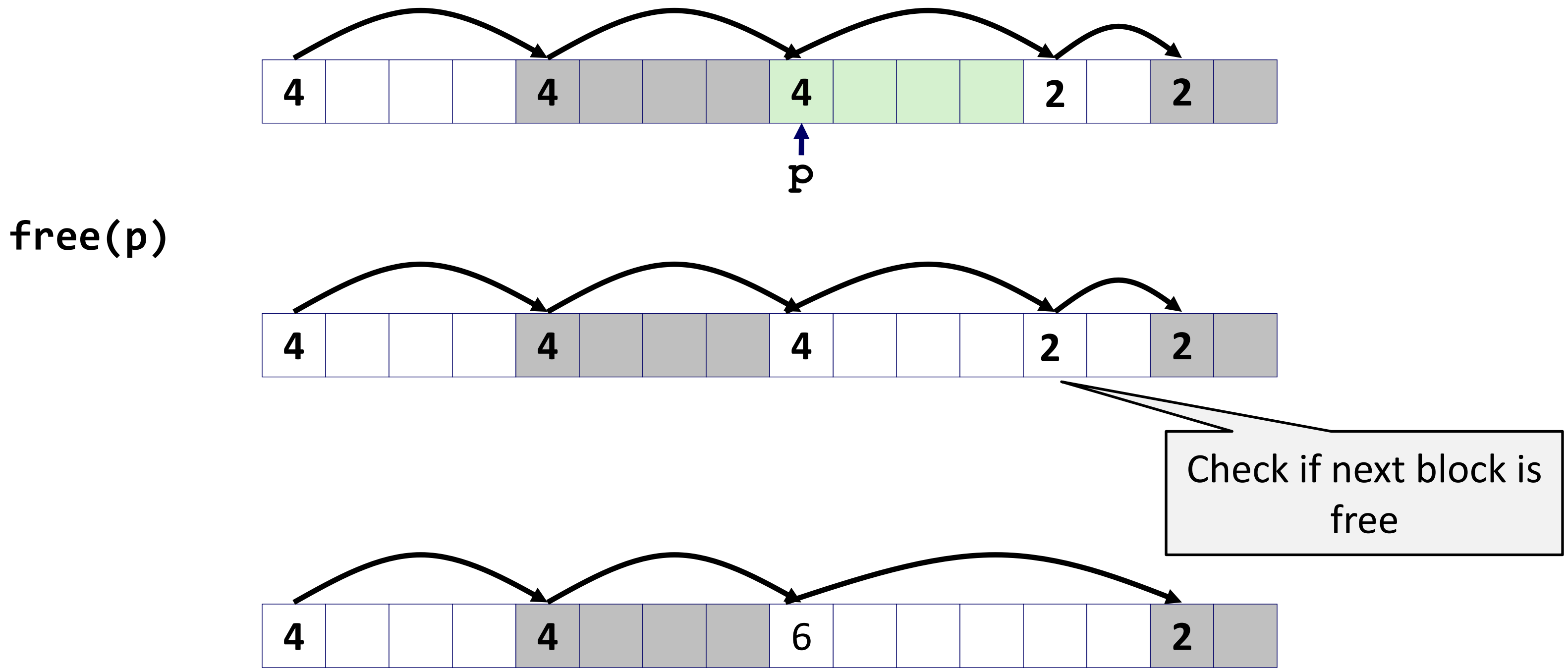
- Join *(coalesce)* with next/previous blocks, if they are free
  - Coalescing with next block

`free(p)`



Implicit List: Coalescing

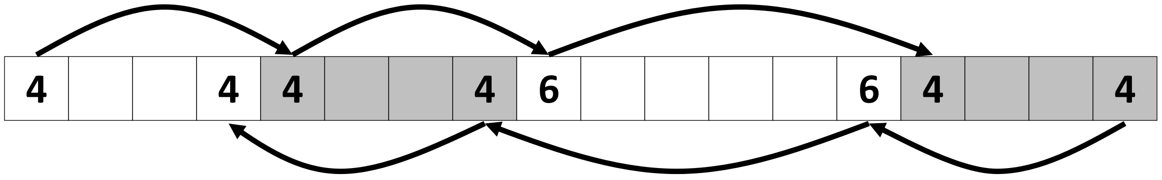
- Join *(coalesce)* with next/previous blocks, if they are free
  - Coalescing with next block



How to coalesce with a previous block?

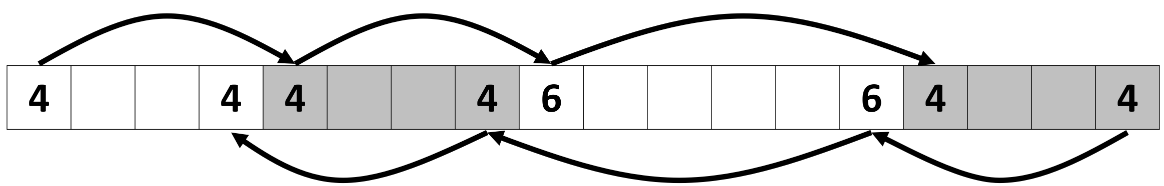
Implicit List: Bidirectional Coalescing

- *Boundary tags* [Knuth73]
  - Replicate size/allocated header at "bottom" (end) of blocks
  - Allows us to traverse the "list" backwards, but requires extra space

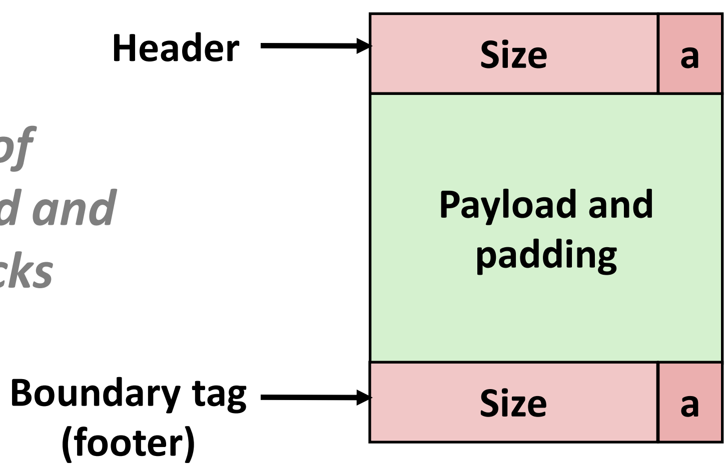


Implicit List: Bidirectional Coalescing

- ***Boundary tags*** [Knuth73]
  - Replicate size/allocated header at "bottom" (end) of blocks
  - Allows us to traverse the "list" backwards, but requires extra space



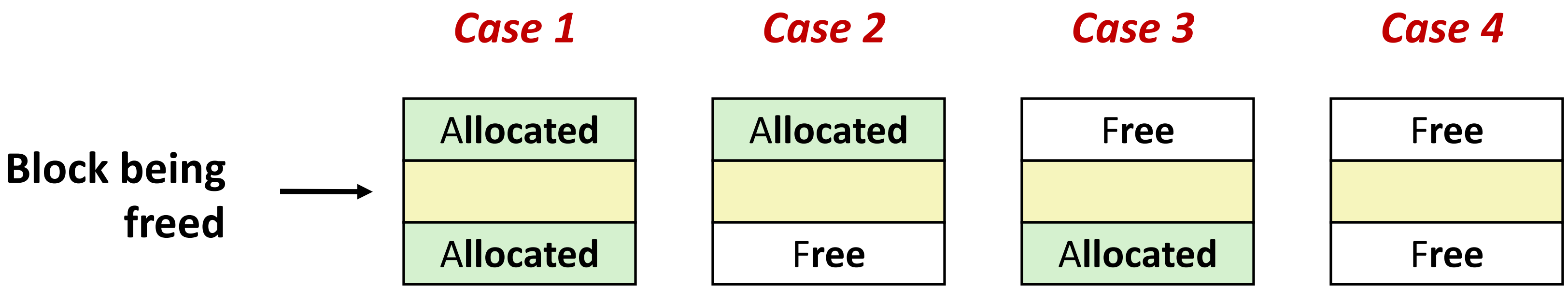
Format of allocated and free blocks



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

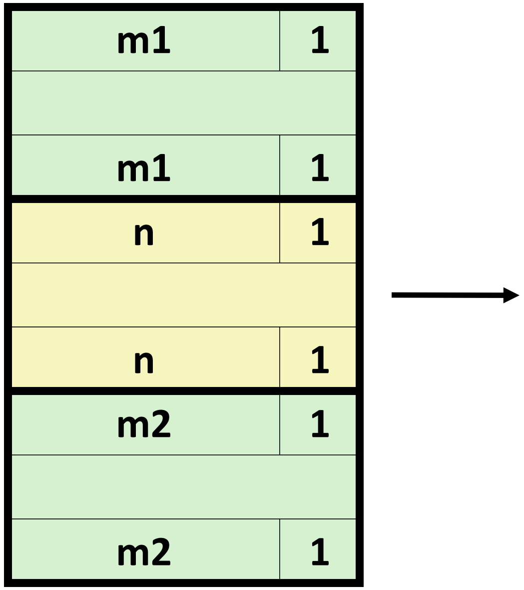
Size: Total block size

Coalescing

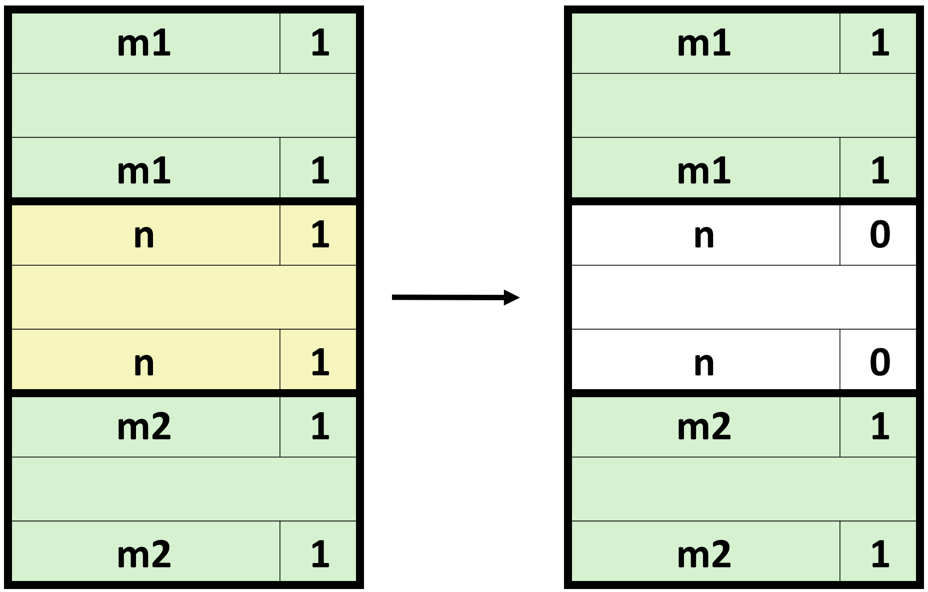


Case 1
Case 2
Case 3
Case 4
Block being freed
Allocated
Allocated
Allocated
Free
Free
Allocated
Free
Free

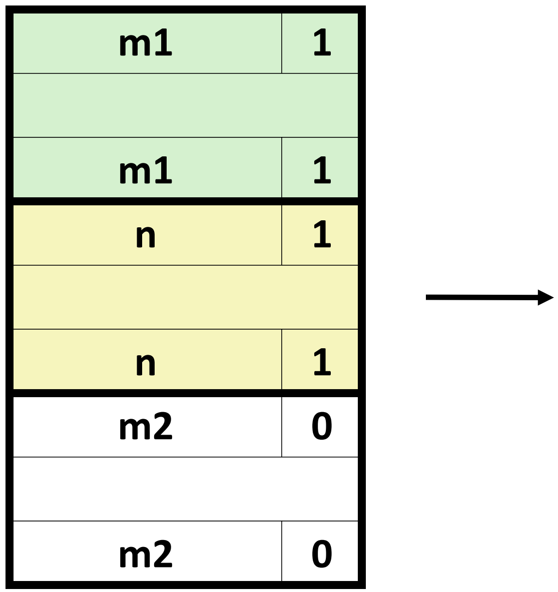
Coalescing (Case 1)



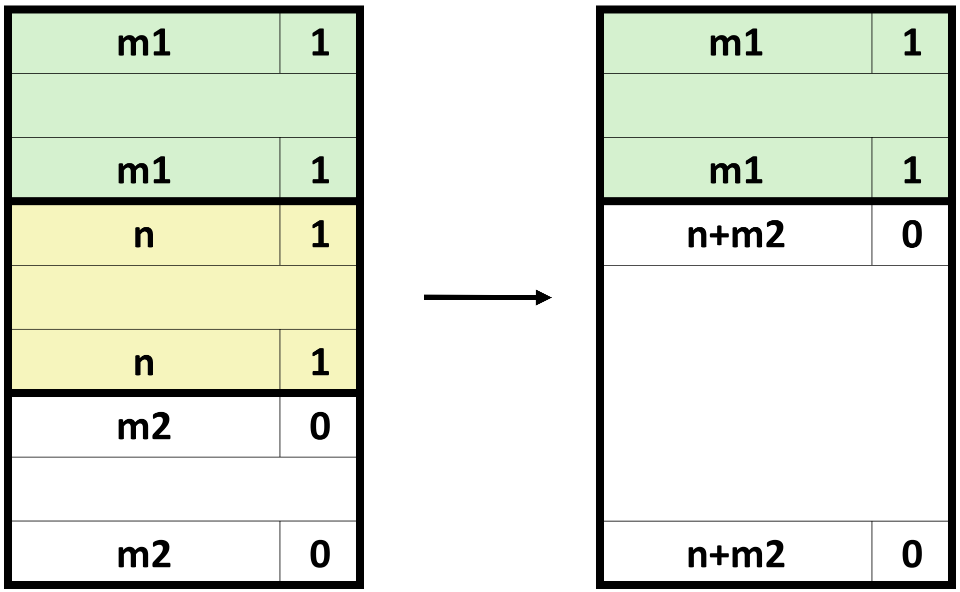
Coalescing (Case 1)



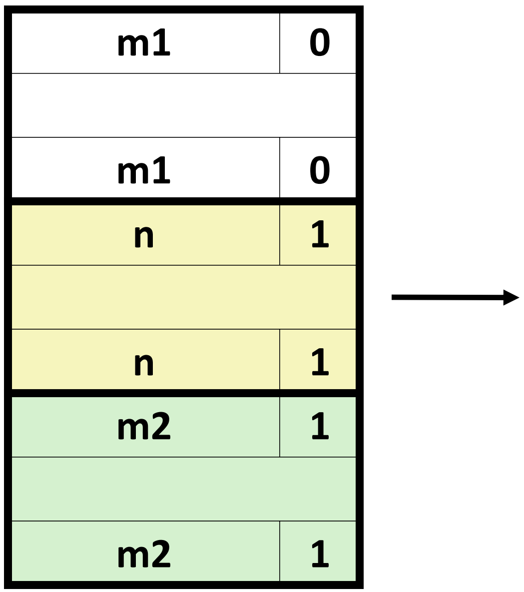
Coalescing (Case 2)



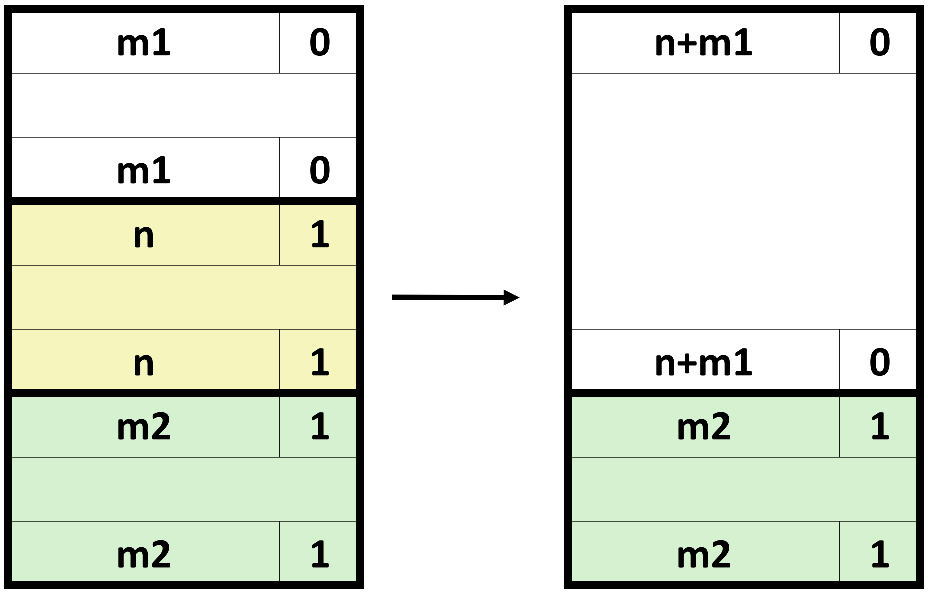
Coalescing (Case 2)



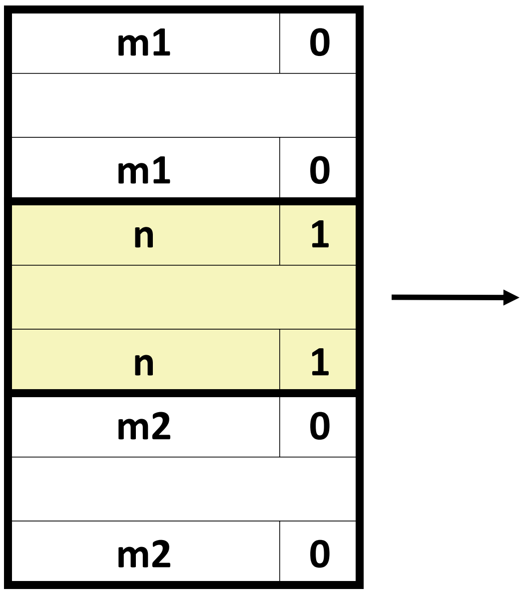
Coalescing (Case 3)



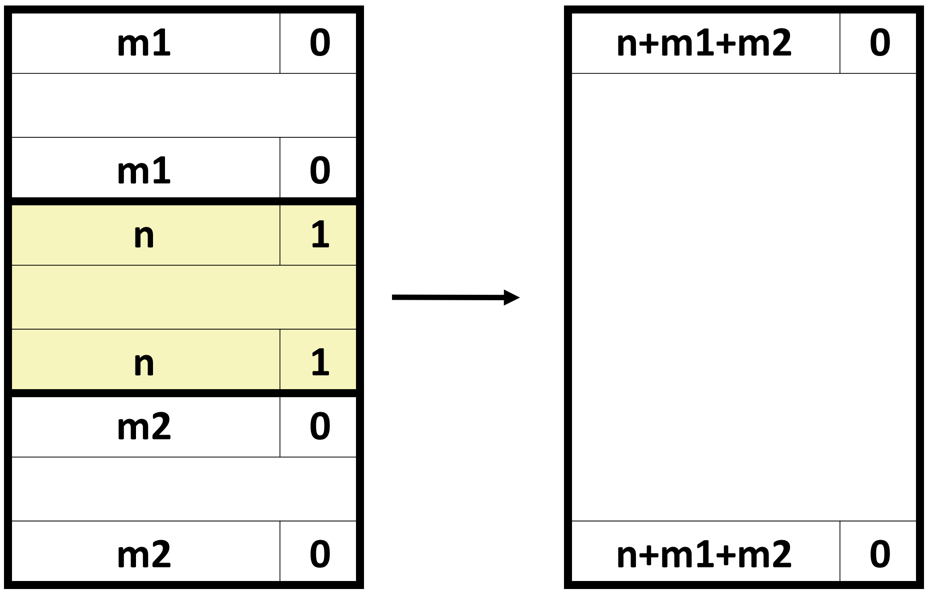
Coalescing (Case 3)



Coalescing (Case 4)



Coalescing (Case 4)



When to coalesce?

- 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

Implicit Lists: 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

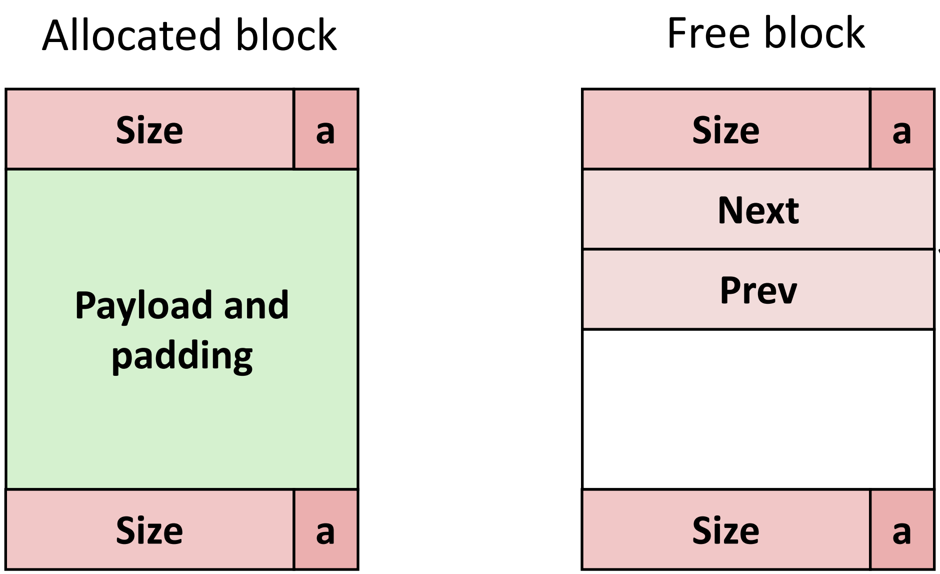
- Not used in practice for `malloc/free` because of high runtime cost for allocation
  - used in many special purpose applications

Explicit Free list

- 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.

Explicit Free Lists



Store next/prev pointers in "payload" of free block.

Does this increase space overhead?

Freeing With Explicit Free Lists

- 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

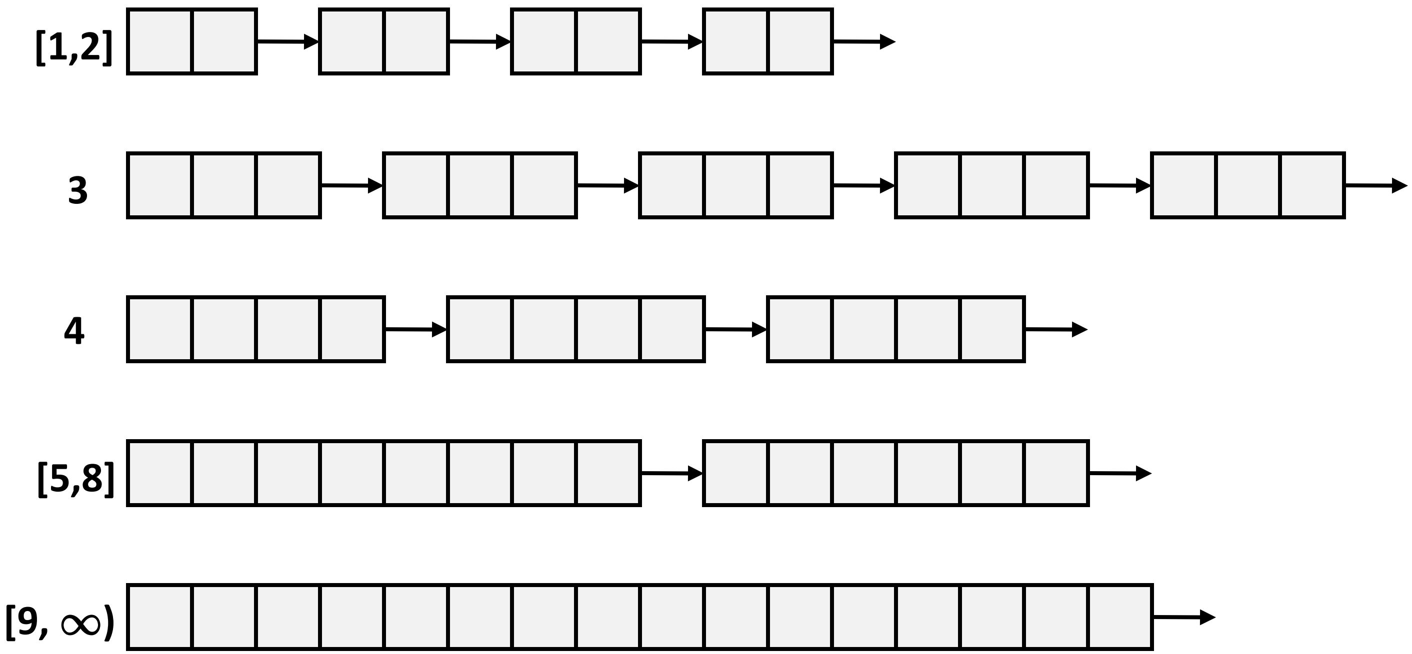
Explicit List

- ✓ 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?

Segregated List (Seglist) Allocators

- Multiple free lists each linking free blocks of similar sizes



Seglist Allocator

- 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 remainder as a single free block in largest size class.

Seglist Allocator (cont.)

- To free a block:
  - Coalesce and place on appropriate list

- Advantages of seglist allocators
  - Fast allocation
  - Better memory utilization
    - First-fit search of segregated free list approximates a best-fit search of entire heap

A Word About Garbage Collection

A Word About Garbage Collection

- In C, it is the programmer's responsibility to free any memory allocated by malloc/calloc/…

A Word About Garbage Collection

- In C, it is the programmer's responsibility to free any memory allocated by malloc/calloc/…
- A garbage collection is a dynamic storage allocator that automatically frees allocated blocks that are no longer needed by the program.

A Word About Garbage Collection

- In C, it is the programmer's responsibility to free any memory allocated by malloc/calloc/…
- A garbage collection is a dynamic storage allocator that automatically frees allocated blocks that are no longer needed by the program.
- Allocated blocks that are no longer needed are called garbage.

A Word About Garbage Collection

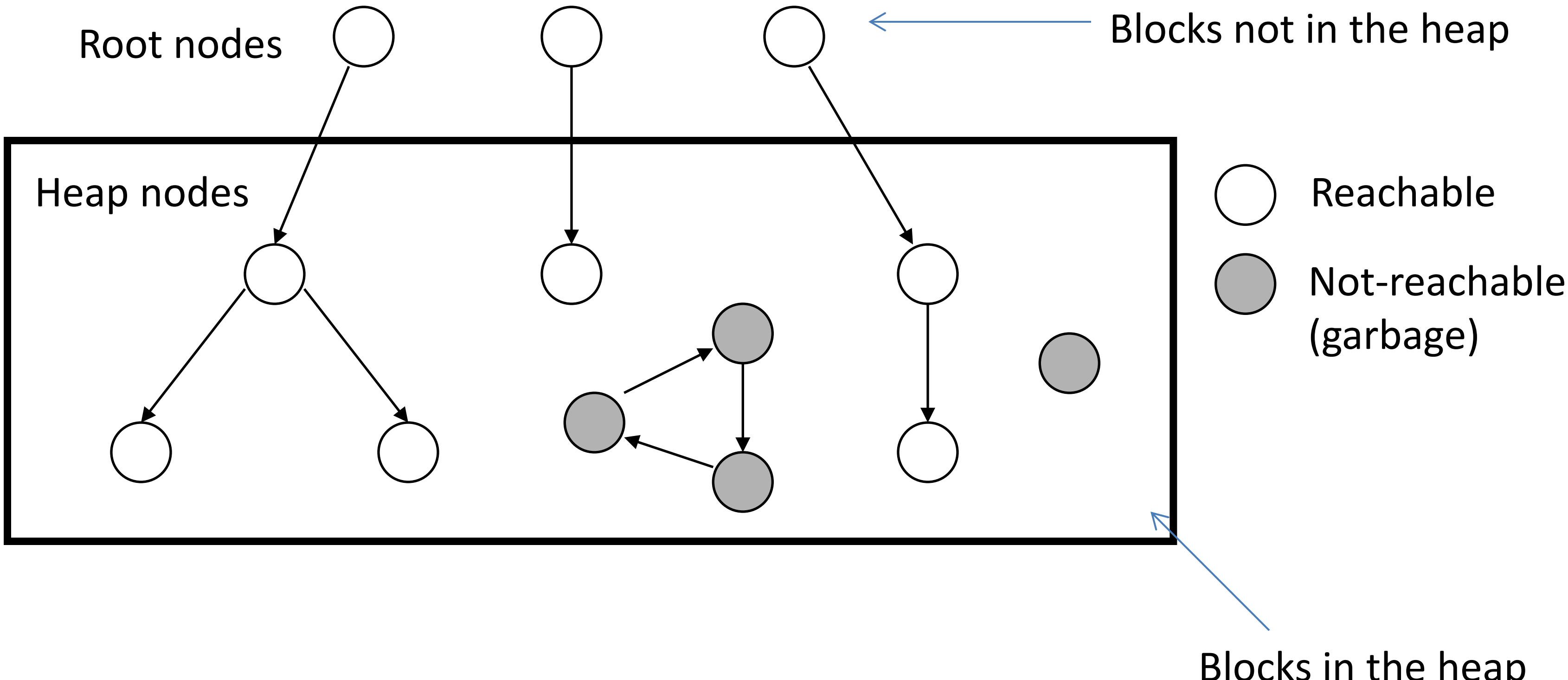
- In systems that support garbage collection (e.g. Java, Perl, Mathematica, …)
  - Applications explicitly allocate heap blocks
  - But never free them!
- The garbage collector periodically identifies garbage and make appropriate calls to free.

A Word About Garbage Collection

- In systems that support garbage collection (e.g. Java, Perl, Mathematica, …)
  - Applications explicitly allocate heap blocks
  - But never free them!
- The garbage collector periodically identifies garbage and make appropriate calls to free.

How does the garbage collector recognizes blocks that are no longer needed?

A Word About Garbage Collection



Reachability Graph

Conclusions

- Dynamic memory allocator manages the heap.
- Dynamic memory allocator is part of the user-space
- The allocator has two main goals:
  - reaching higher throughput (operations per second)
  - better memory utilization (i.e. reduces fragmentation).

Conclusions (cont'd)

- Explicit allocator
  - Works in terms of blocks
  - Keeping track of free blocks
    - Implicit list
    - Explicit list
    - segregated list
    - blocks sorted by size
- Implicit allocator

Virtual Memory and Isolation

Isolation

User applications

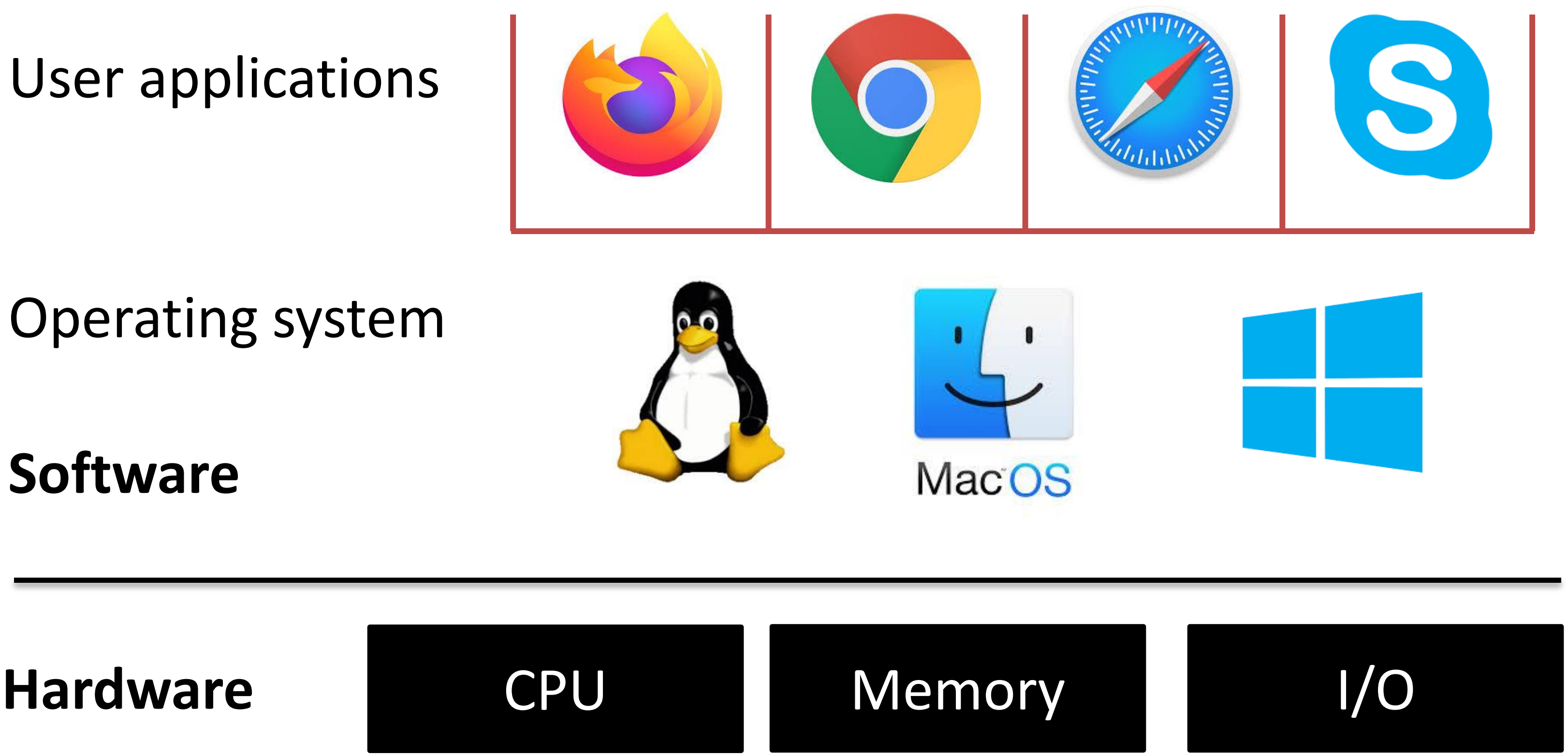
Operating system

Software

Hardware

CPU	Memory	I/O

Isolation


User applications
Operating system
Software
MacOS
Hardware
CPU
Memory
I/O

Isolation

User applications

Operating system

MacOS

Software

Hardware

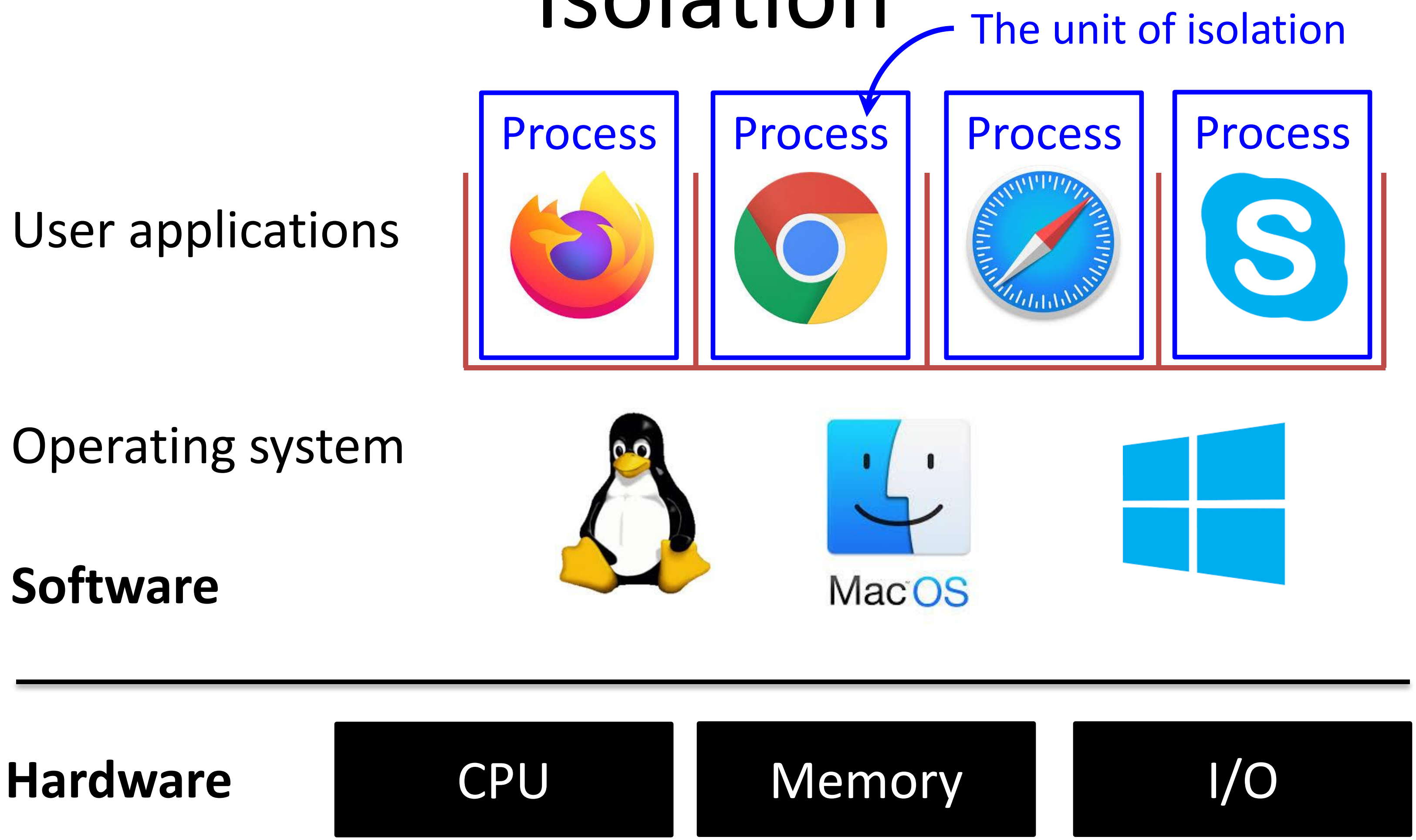
CPU

Memory

I/O

Isolation – Enforced separation to contain effects of failures

Isolation

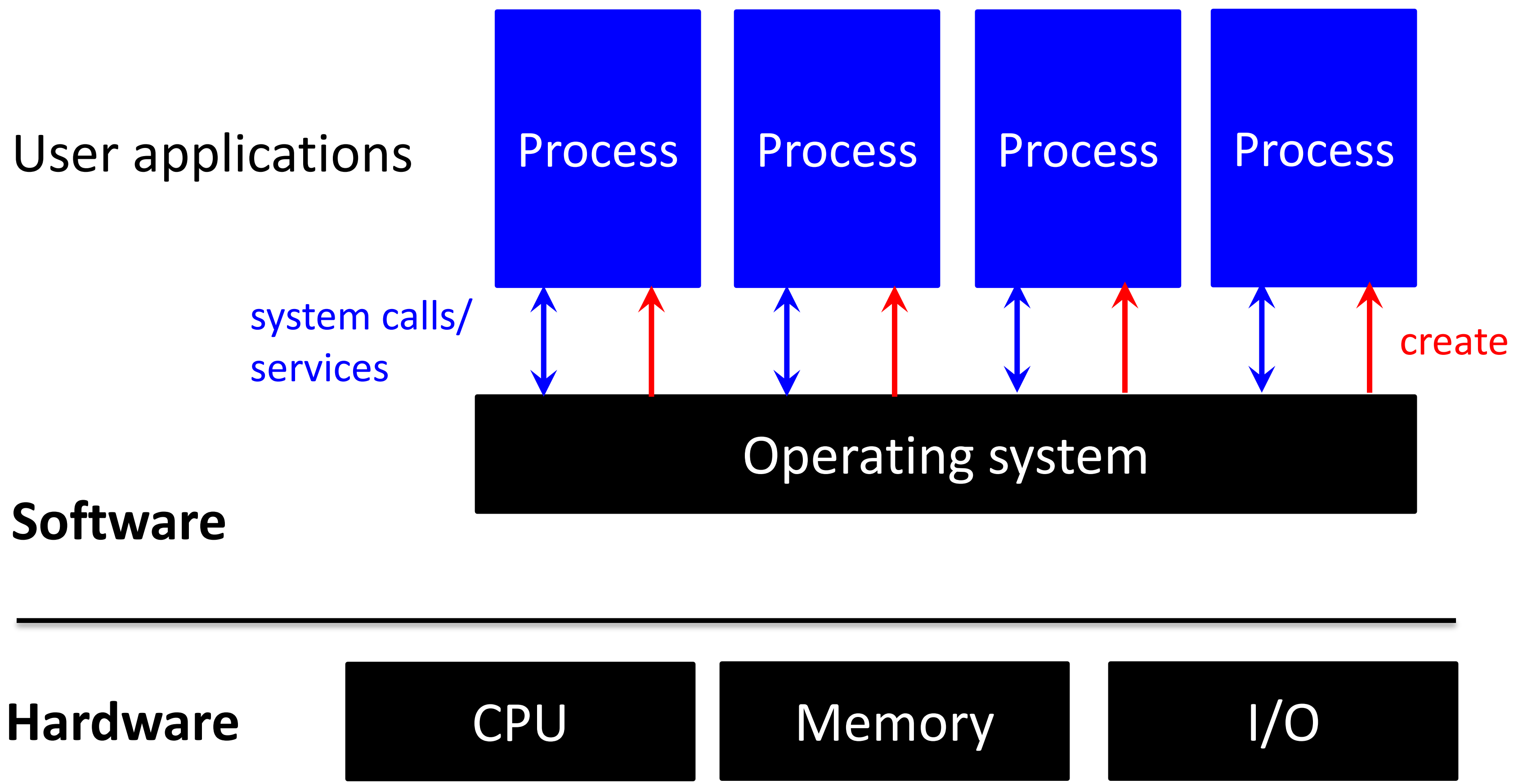


Isolation – Enforced separation to contain effects of failures

Process

- An instance of a computer program that is being executed
- Program vs. Process
  - Program: a passive collection of instructions
  - Process: the actual execution of those instructions
- Different processes have different process id
  - `getpid()`: function that returns id of current process
  - Command `ps`: list all processes

Isolation



To run a program, OS starts a process and provide services through system calls (`getpid()`, `fopen()`).