18-600 Foundations of Computer Systems

Lecture 16:
"Dynamic Memory Allocation"

October 23, 2017

18-600

SE

PL

OS

CA

➢ Required Reading Assignment:
- **Chapter 9 of CS:APP (3rd edition) by Randy Bryant & Dave O'Hallaron**

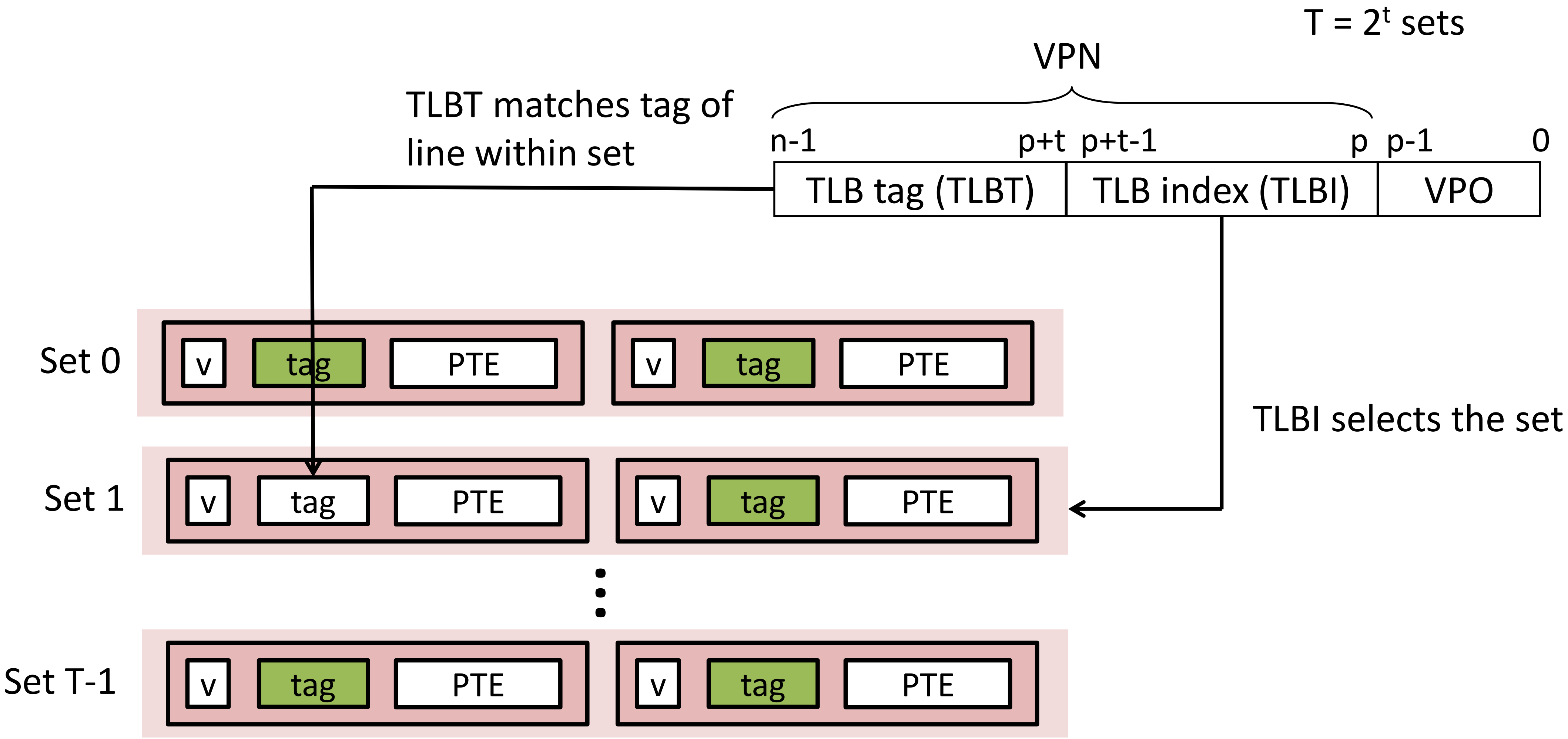
Electrical & Computer ENGINEERING

Socrative Experiment (Continuing)

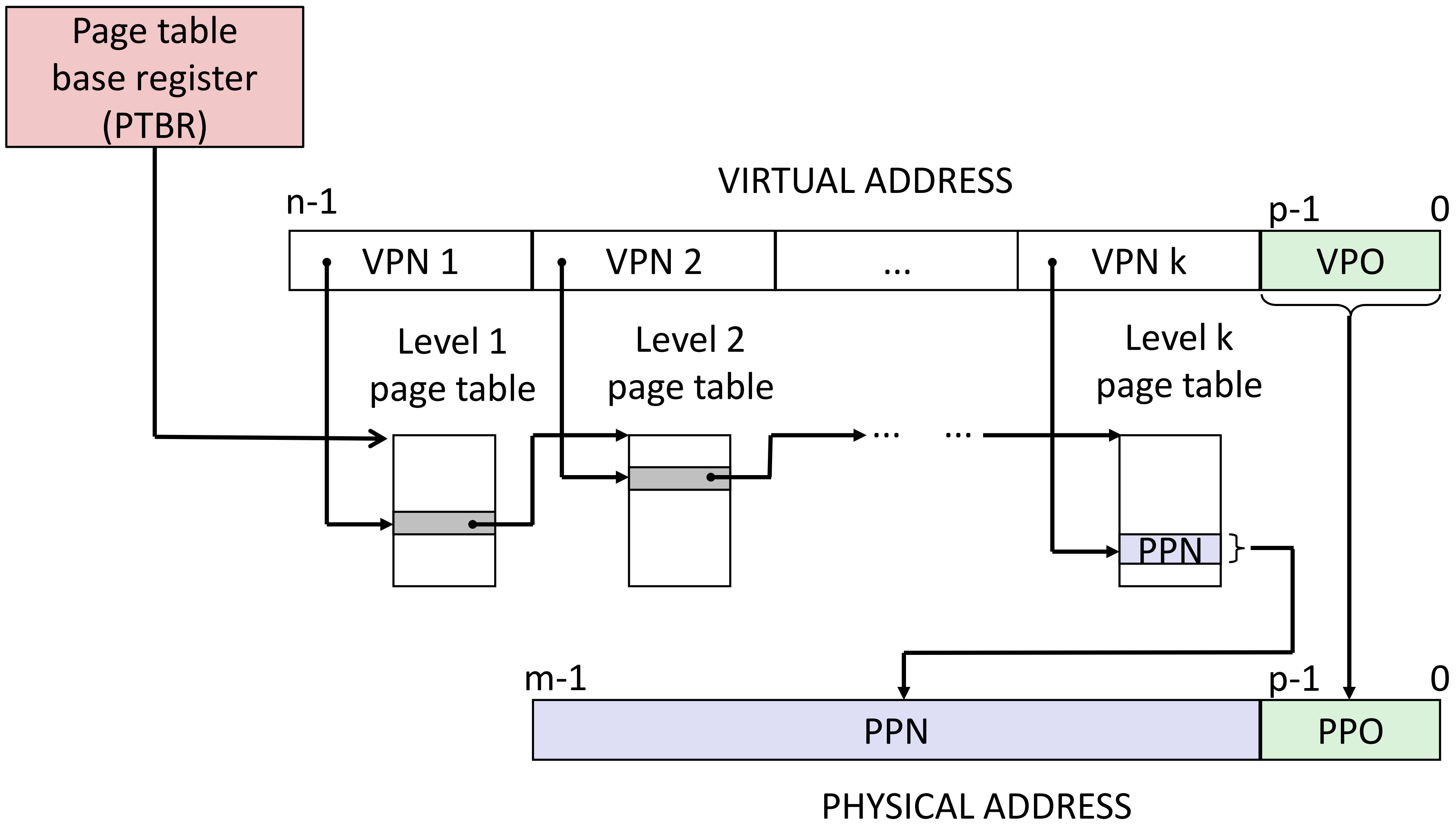
- Pittsburgh Students (18600PGH): https://api.socrative.com/rc/icJVVC
- Silicon Valley Students (18600SV): https://api.socrative.com/rc/iez85z
- Microphone/Speak out/Raise Hand: Still G-R-E-A-T!
- Socrative:
  - Let's me open floor for electronic questions, putting questions into a visual queue so I don't miss any
  - Let's me do flash polls, etc.
  - Prevents cross-talk and organic discussions in more generalized forums from pulling coteries out of class discussion into parallel question space.
    - Keeps focus and reduces distraction while adding another vehicle for classroom interactivity.
  - Won't allow more than 150 students per "room"
    - So, I created one room per campus
    - May later try random assignment to a room, etc.

Accessing the TLB

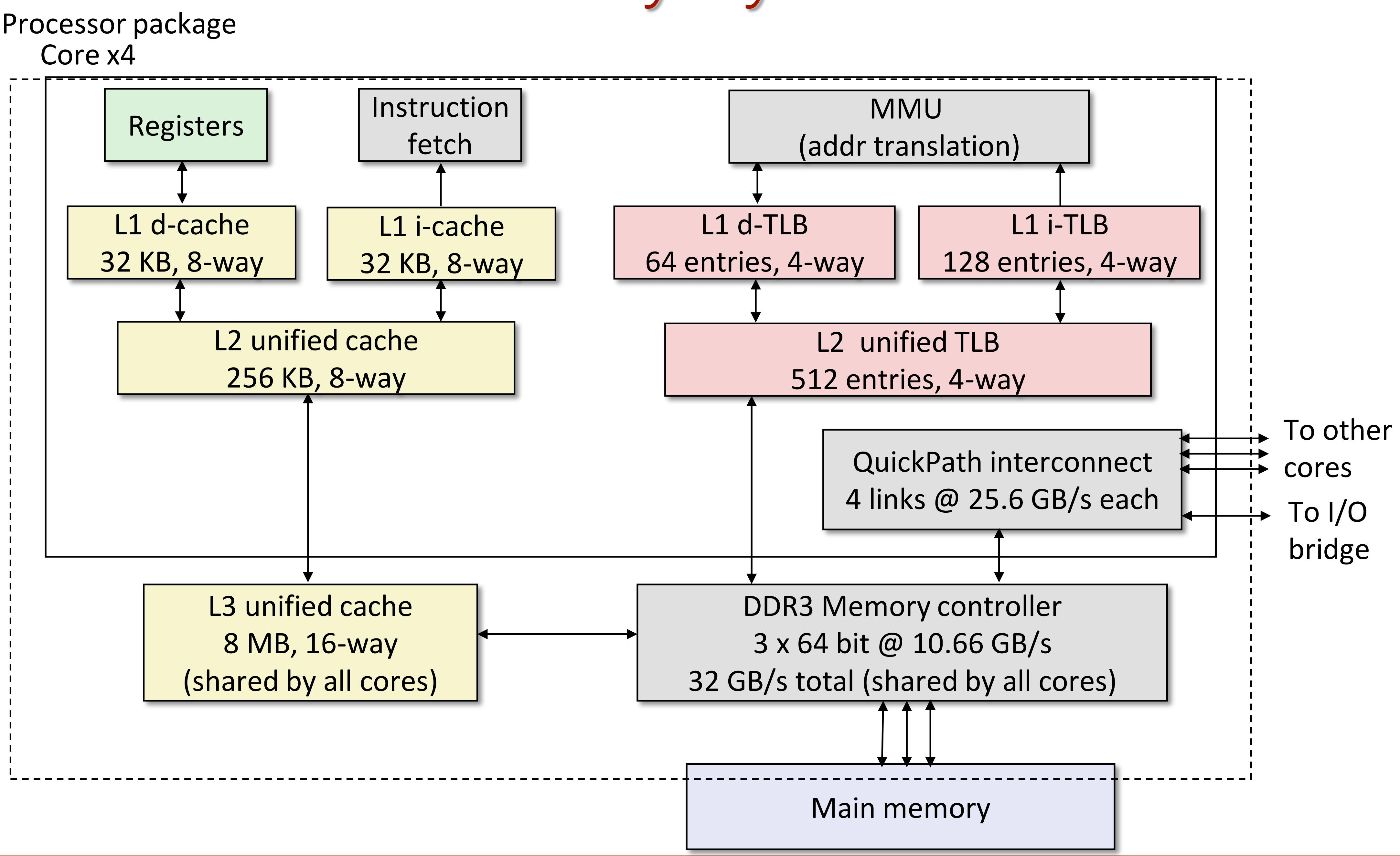
- MMU uses the VPN portion of the virtual address to access the TLB:



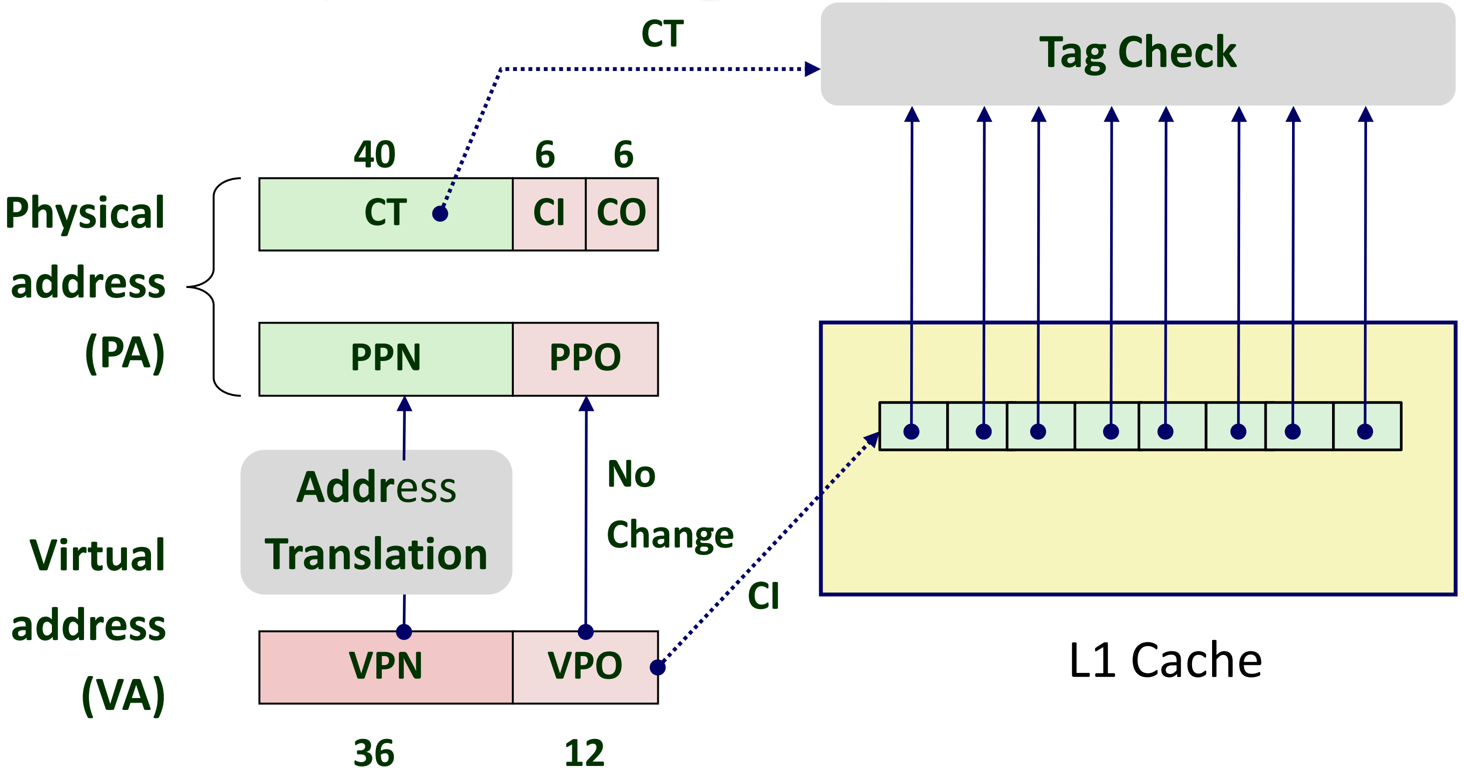
Translating with a k-level Page Table



Intel Core i7 Memory System

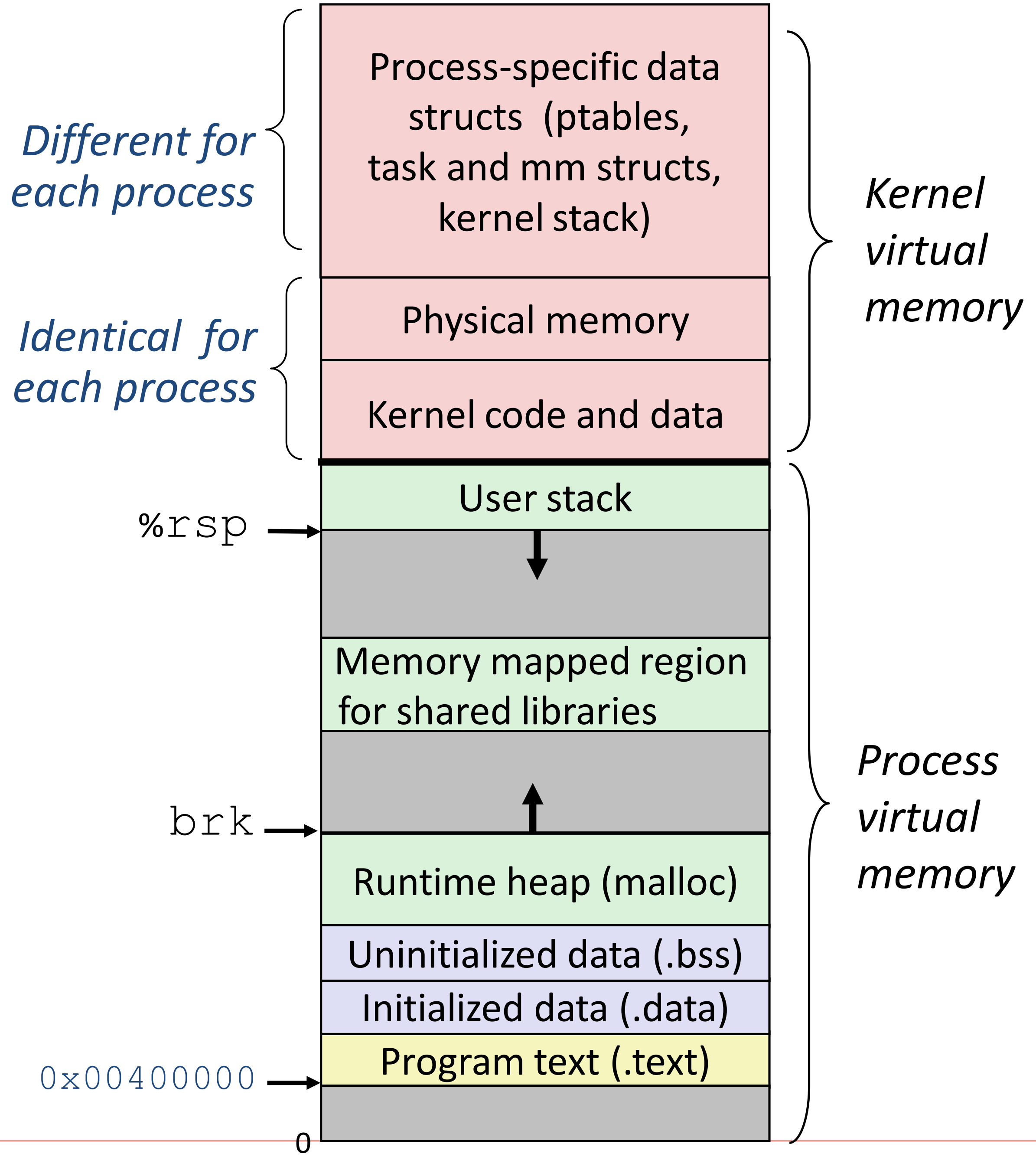


Cute Trick for Speeding Up L1 Access

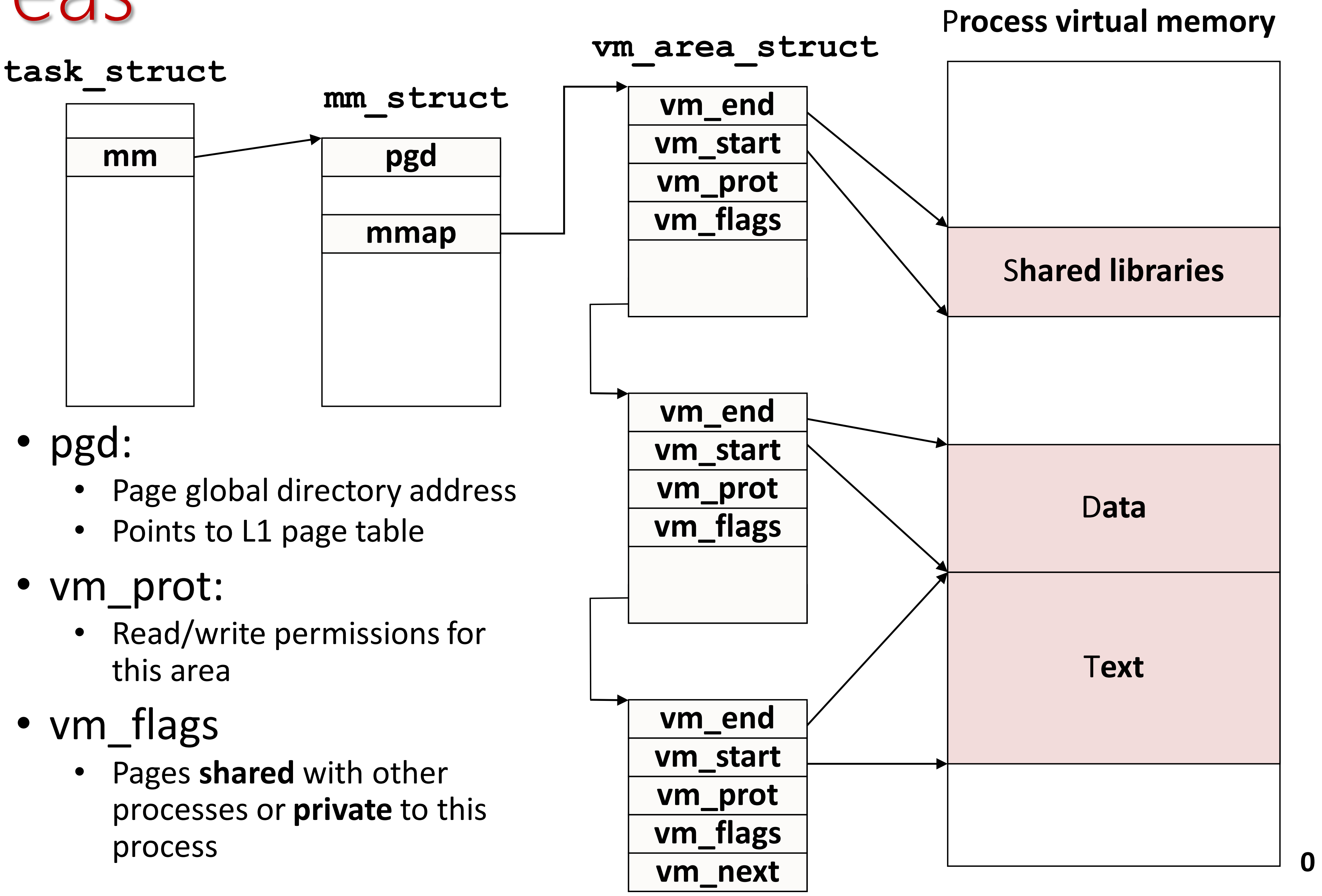


- Observation
  - Bits that determine CI identical in virtual and physical address
  - Can index into cache while address translation taking place
  - Generally we hit in TLB, so PPN bits (CT bits) available next
  - "Virtually indexed, physically tagged"
  - Cache carefully sized to make this possible

Virtual Address Space of a Linux Process

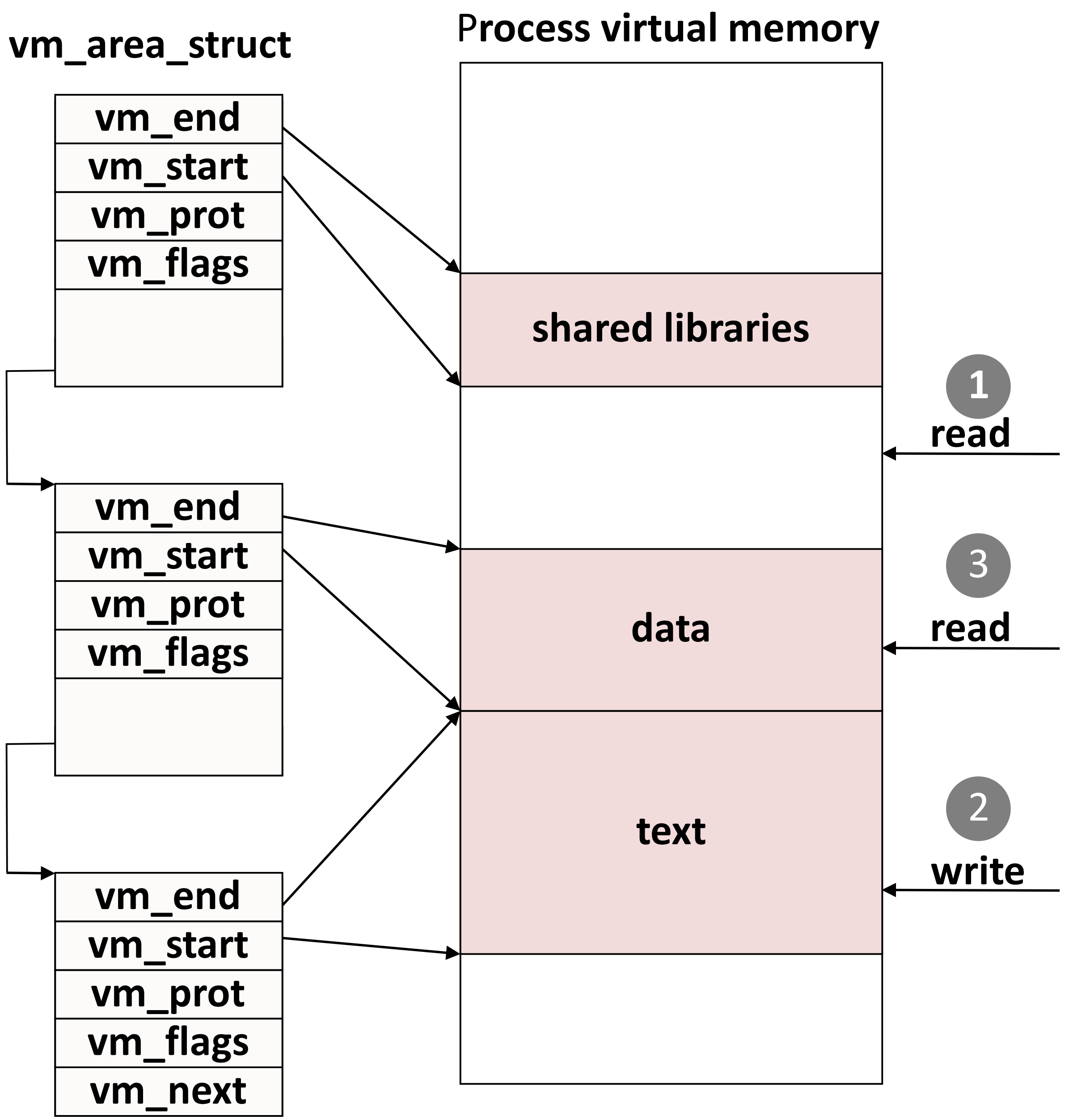


Linux Organizes VM as Collection of "Areas"



- pgd:
  - Page global directory address
  - Points to L1 page table
- vm_prot:
  - Read/write permissions for this area
- vm_flags
  - Pages **shared** with other processes or **private** to this process

Linux Page Fault Handling



Segmentation fault:
accessing a non-existing page

Normal page fault

Protection exception:
e.g., violating permission by writing to a read-only page (Linux reports as Segmentation fault)

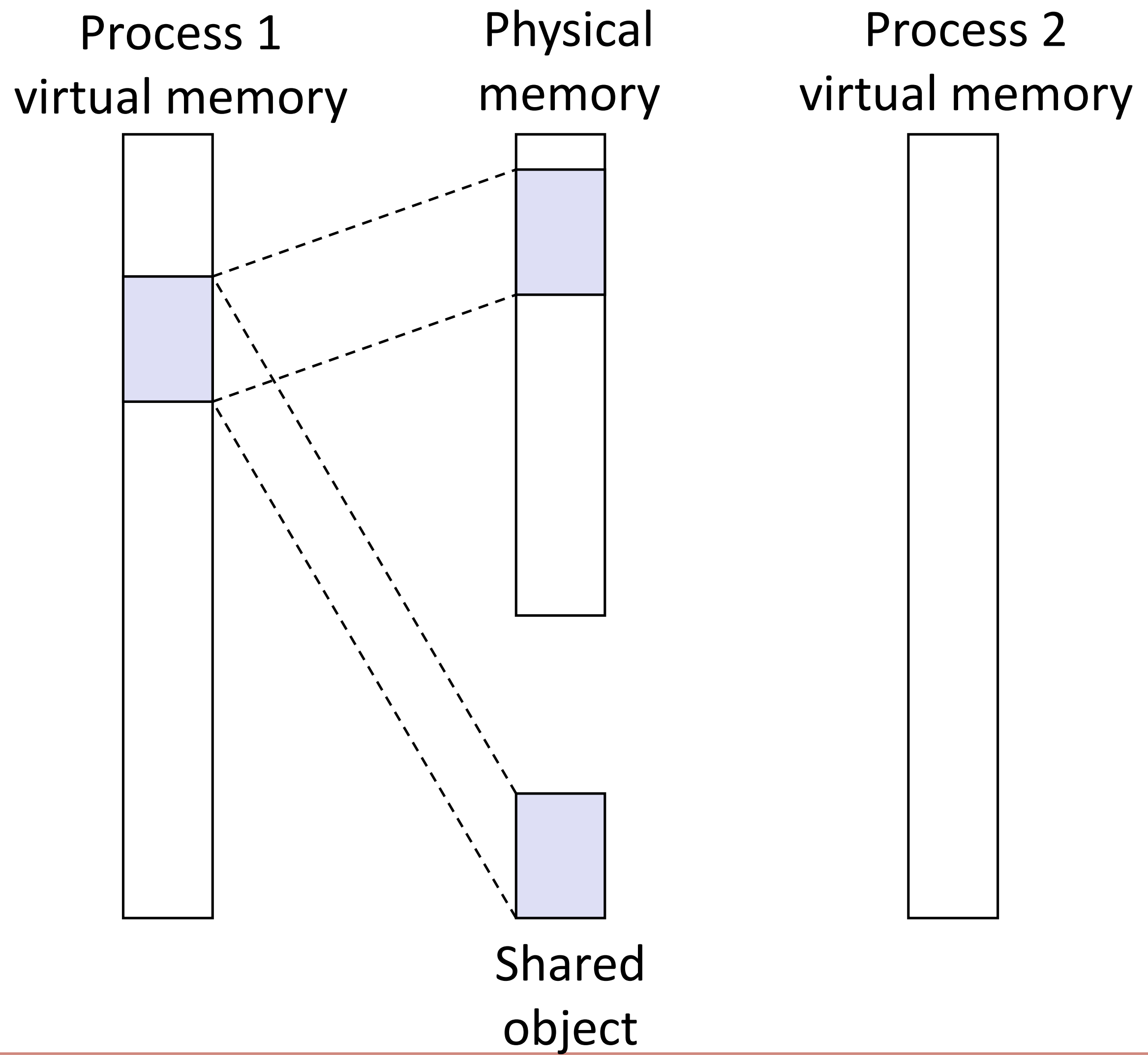
Memory Mapping

- VM areas initialized by associating them with disk objects.
  - Process is known as ***memory mapping***.

- Area can be *backed by* (i.e., get its initial values from) :
  - ***Regular file*** on disk (e.g., an executable object file)
    - Initial page bytes come from a section of a file
  - ***Anonymous file*** (e.g., nothing)
    - First fault will allocate a physical page full of 0's (***demand-zero page***)
    - Once the page is written to (***dirtied***), it is like any other page

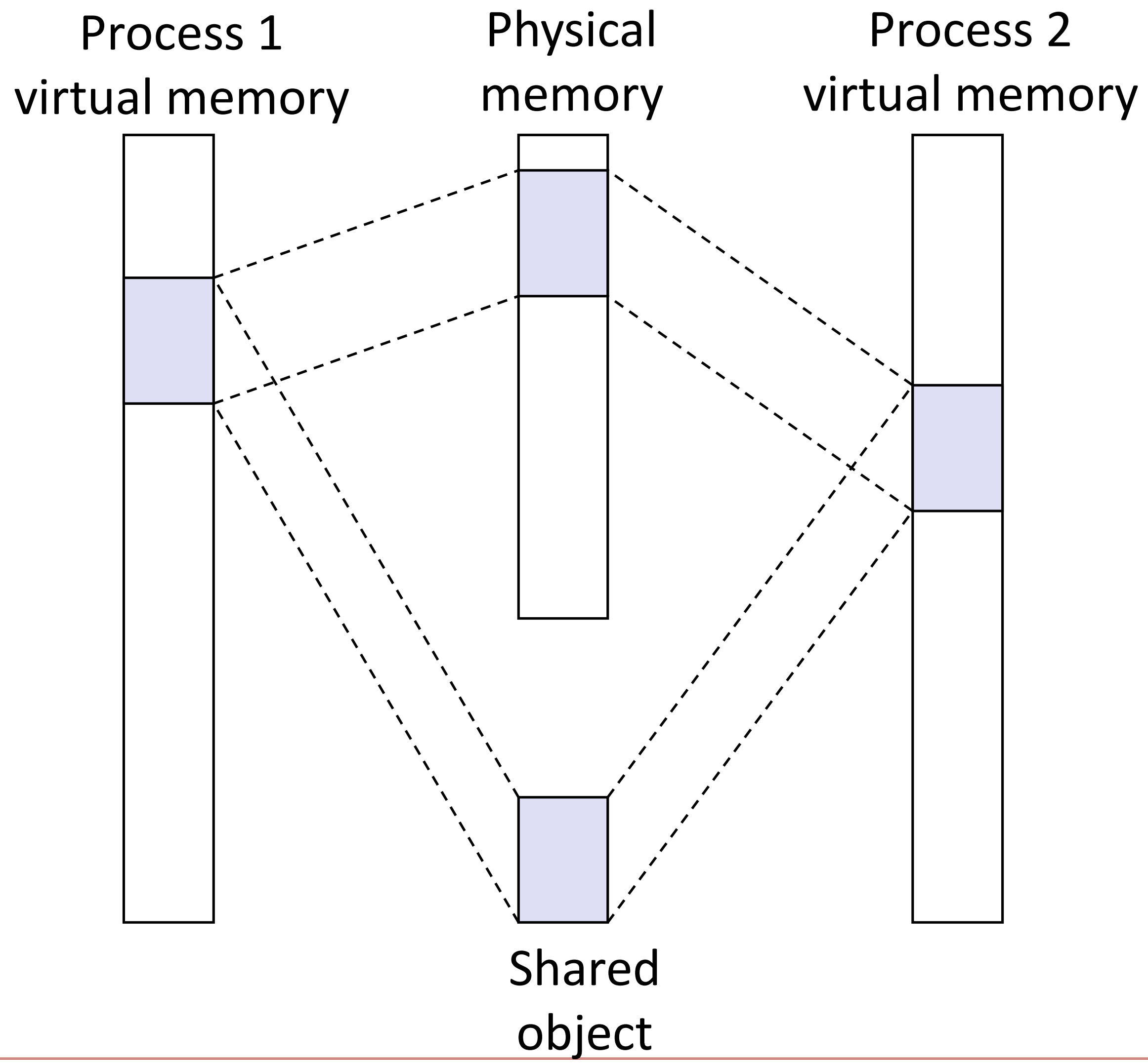
- Dirty pages are copied back and forth between memory and a special *swap file*.

Sharing Revisited: Shared Objects



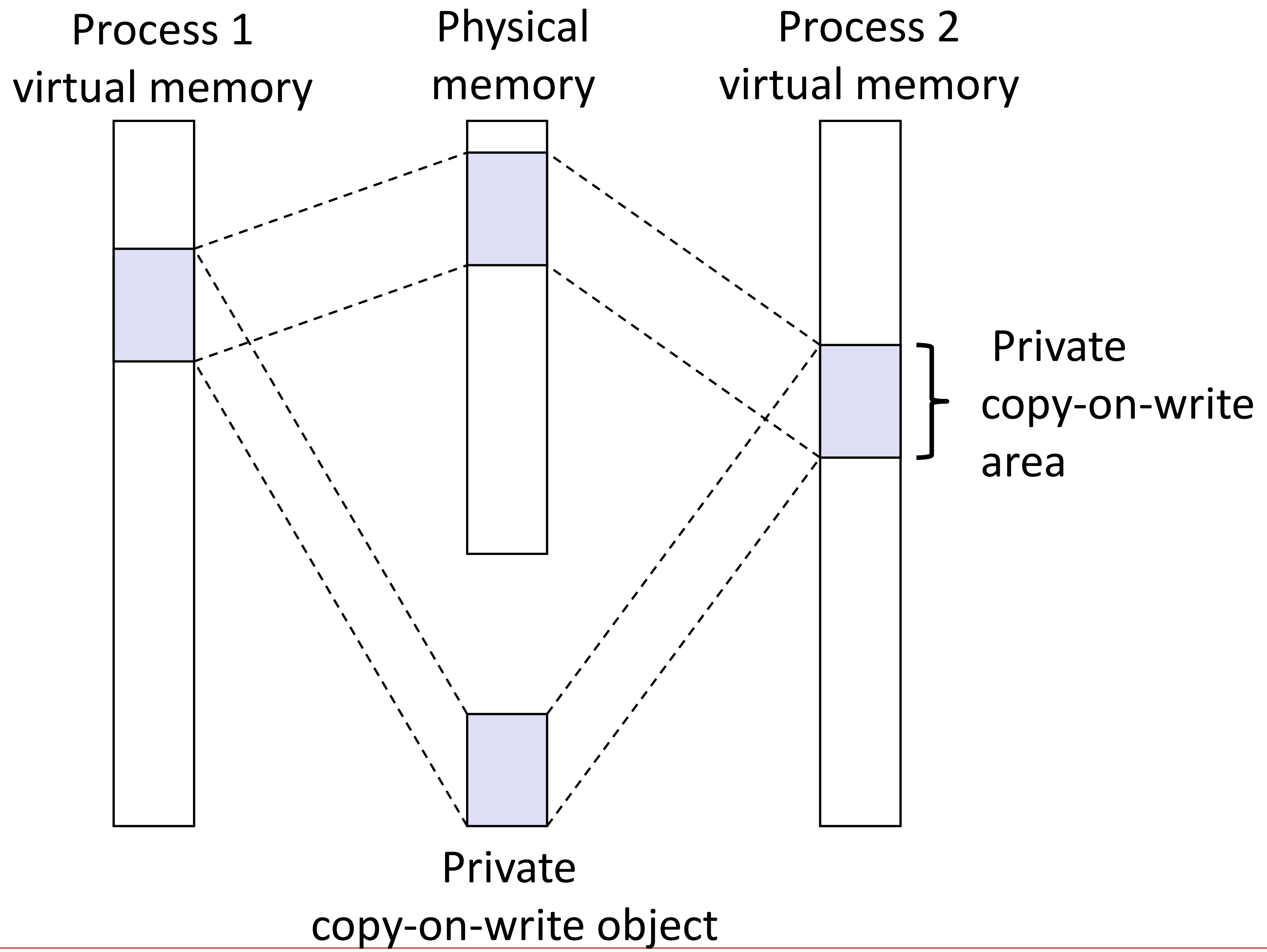
- Process 1 maps the shared object.

Sharing Revisited: Shared Objects



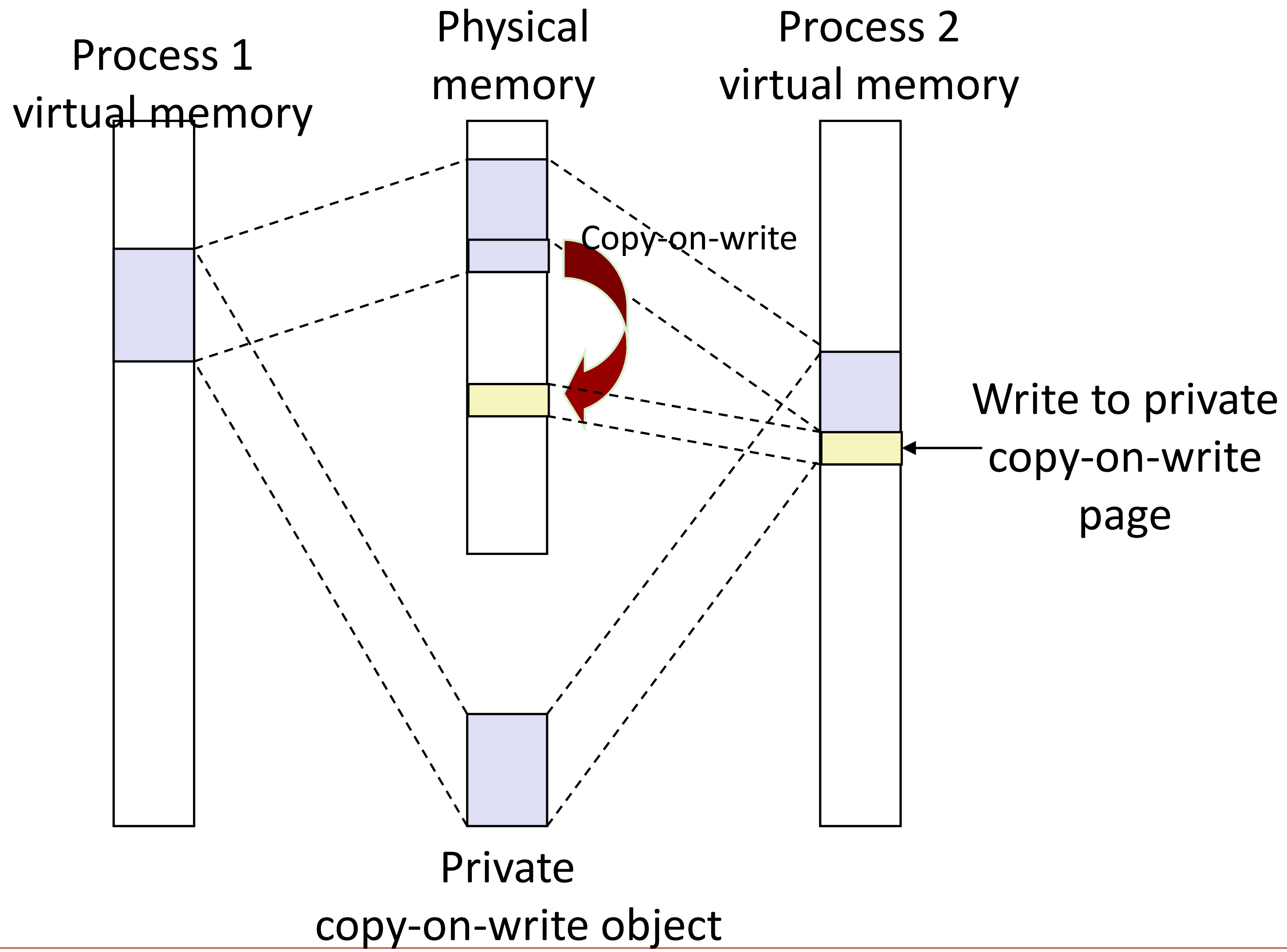
- **Process 2 maps the shared object.**
- Notice how the virtual addresses can be different.

Sharing Revisited: Private Copy-on-write (COW) Objects



- Two processes mapping a *private copy-on-write (COW)* object.
- Area flagged as private copy-on-write
- PTEs in private areas are flagged as read-only

Sharing Revisited: Private Copy-on-write (COW) Objects



- Instruction writing to private page triggers protection fault.
- Handler creates new R/W page.
- Instruction restarts upon handler return.
- Copying deferred as long as possible!

The `fork` Function Revisited

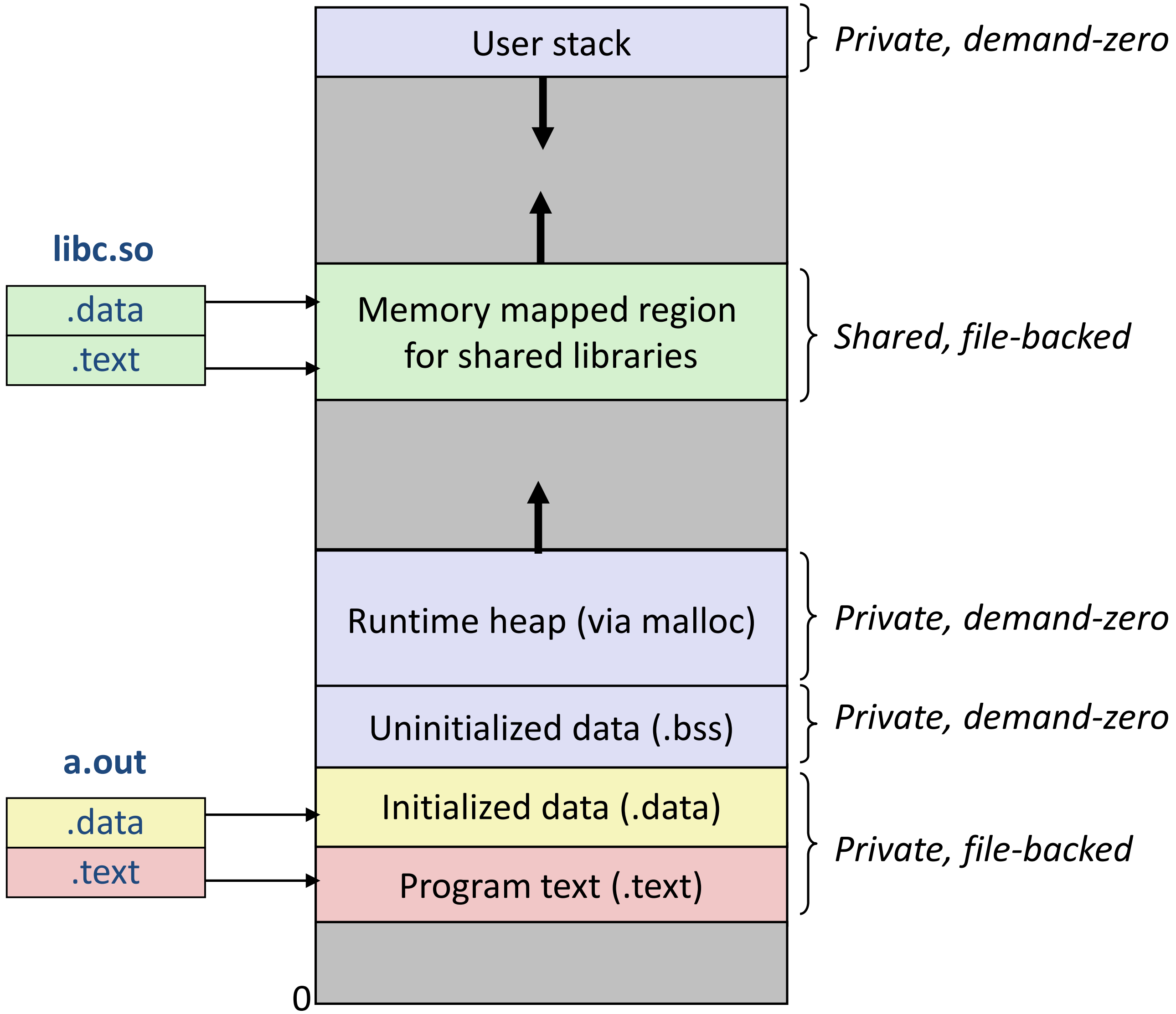
- VM and memory mapping explain how `fork` provides private address space for each process.

- To create virtual address for new new process
  - Create exact copies of current `mm_struct`, `vm_area_struct`, and page tables.
  - Flag each page in both processes as read-only
  - Flag each `vm_area_struct` in both processes as private COW

- On return, each process has exact copy of virtual memory

- Subsequent writes create new pages using COW mechanism.

The `execve` Function Revisited



- To load and run a new program `a.out` in the current process using `execve`:
- Free `vm_area_struct`'s and page tables for old areas
- Create `vm_area_struct`'s and page tables for new areas
  - Programs and initialized data backed by object files.
  - `.bss` and stack backed by anonymous files .
- Set PC to entry point in `.text`
  - Linux will fault in code and data pages as needed.

User-Level Memory Mapping

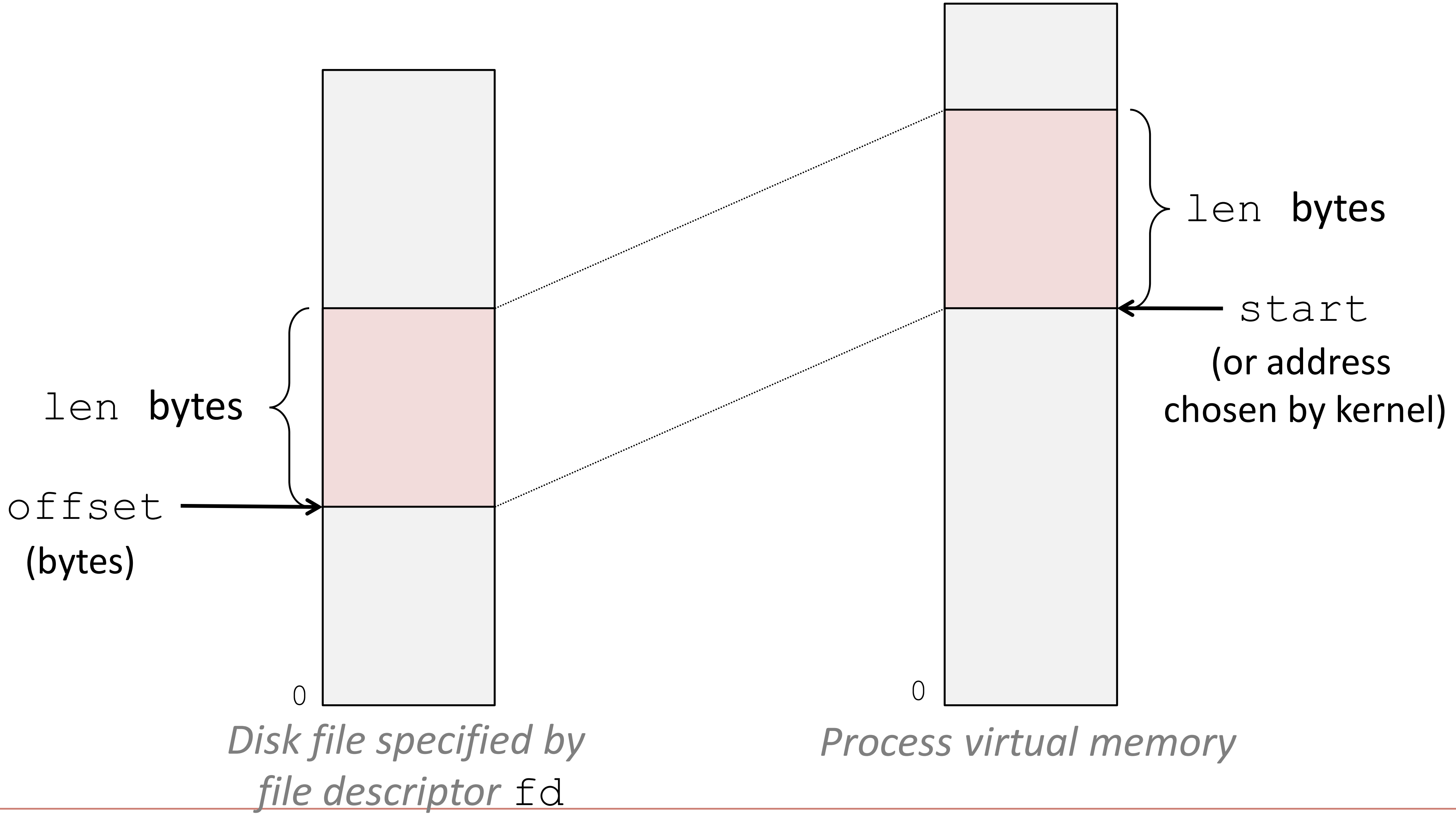
```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```

- Map **`len`** bytes starting at offset **`offset`** of the file specified by file description **`fd`**, preferably at address **`start`**
  - **`start:`** may be 0 for "pick an address"
  - **`prot`**: PROT_READ, PROT_WRITE, ...
  - **`flags`**: MAP_ANON, MAP_PRIVATE, MAP_SHARED, ...

- Return a pointer to start of mapped area (may not be **`start`**)

User-Level Memory Mapping

```
void *mmap(void *start, int len,
           int prot, int flags, int fd, int offset)
```



Example: Using mmap to Copy Files

- Copying a file to `stdout` without transferring data to user space .

```
#include "csapp.h"

void mmapcopy(int fd, int size)
{

   /* Ptr to memory mapped area */
   char *bufp;

   bufp = Mmap(NULL, size,
            PROT_READ,
            MAP_PRIVATE,
            fd, 0);
   Write(1, bufp, size);
   return;
}
```

mmapcopy.c

```
/* mmapcopy driver */
int main(int argc, char **argv)
{
   struct stat stat;
   int fd;

   /* Check for required cmd line arg */
   if (argc != 2) {
      printf("usage: %s <filename>\n",
           argv[0]);
      exit(0);
   }

   /* Copy input file to stdout */
   fd = Open(argv[1], O_RDONLY, 0);
   Fstat(fd, &stat);
   mmapcopy(fd, stat.st_size);
   exit(0);
}
```

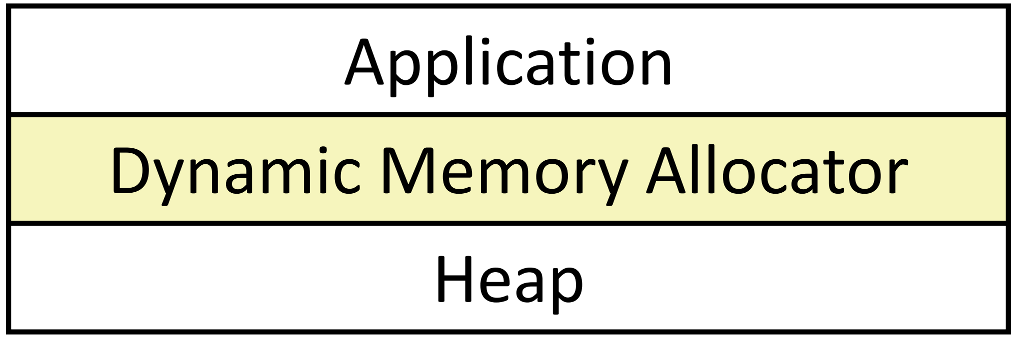
mmapcopy.c

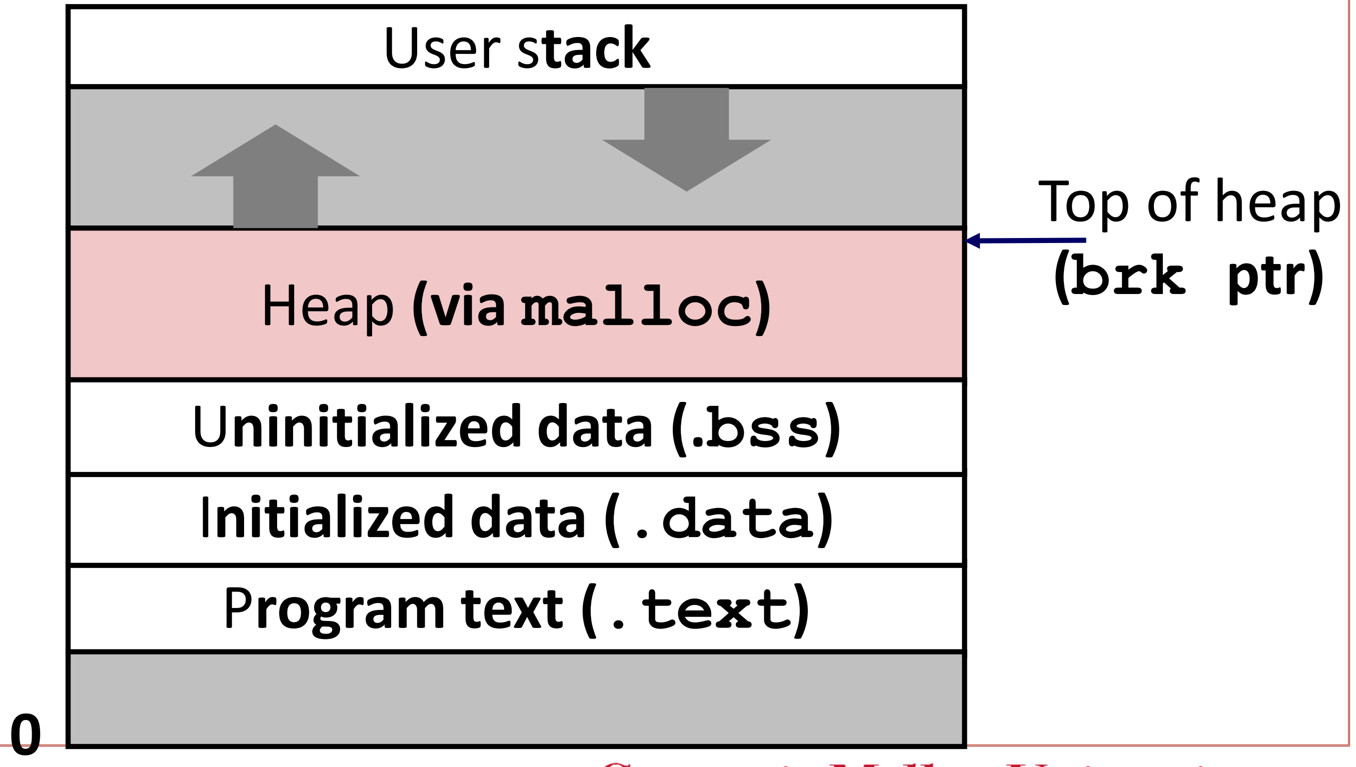
Today: User-Level Memory Allocation

- Basic concepts
- Implicit free lists
- Explicit free lists
- Segregated free lists

Dynamic Memory Allocation

- Programmers use *dynamic memory allocators* (such as `malloc`) to acquire VM at run time.
  - For data structures whose size is only known at runtime.
- Dynamic memory allocators manage an area of process virtual memory known as the *heap*.
- Allocator maintains heap as collection of variable sized *blocks*, which are either *allocated* or *free*
- Types of allocators
  - ***Explicit allocator***: application allocates and frees space
    - E.g., `malloc` and `free` in C
  - ***Implicit allocator:*** application allocates, but does not free space
    - E.g. in Java, ML, and Lisp





The `malloc` Package

`#include <stdlib.h>`

`void *malloc(size_t size)`

- Successful:
  - Returns a pointer to a memory block of at least **`size`** bytes aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
  - If **`size == 0`**, returns NULL
- Unsuccessful: returns NULL (0) and sets **`errno`**

`void free(void *p)`

- Returns the block pointed at by **`p`** to pool of available memory
- **`p`** must come from a previous call to **`malloc`** or **`realloc`**

Other functions

- **`calloc`**: Version of **`malloc`** that initializes allocated block to zero.
- **`realloc`**: Changes the size of a previously allocated block.
- **`sbrk`**: Used internally by allocators to grow or shrink the heap

malloc Example

```
#include <stdio.h>
#include <stdlib.h>

void foo(int n) {
   int i, *p;

   /* Allocate a block of n ints */
   p = (int *) malloc(n * sizeof(int));
   if (p == NULL) {
      perror("malloc");
      exit(0);
   }

   /* Initialize allocated block */
   for (i=0; i<n; i++)
            p[i] = i;

   /* Return allocated block to the heap */
   free(p);
}
```

Allocation Example

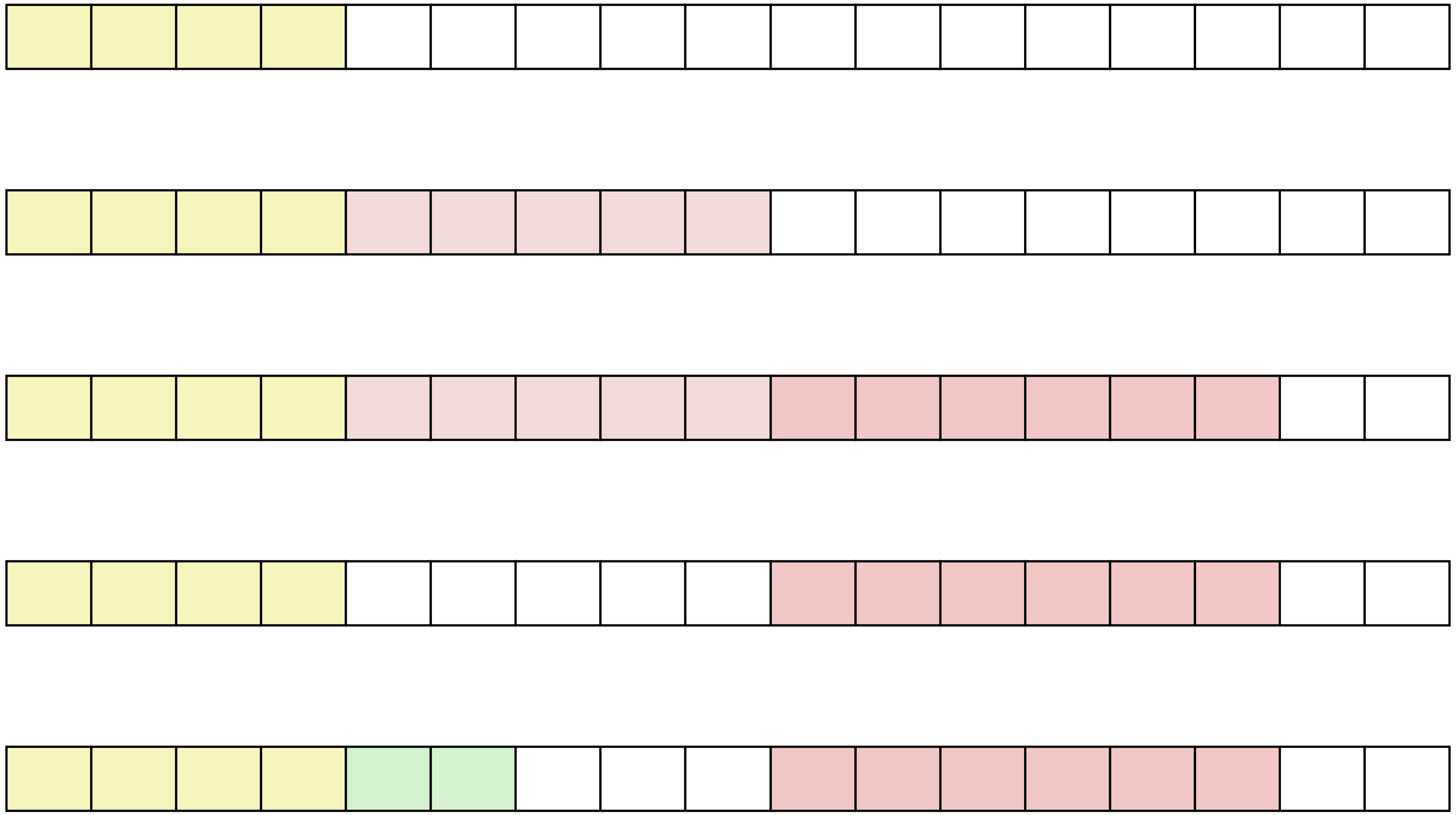
```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(2)
```



- Assumption
  - Memory is word addressed.
  - Words are int-sized.

Constraints

- Applications
  - Can issue arbitrary sequence of **`malloc`** and **`free`** requests
  - **`free`** request must be to a **`malloc`**'d block
- Allocators
  - Can't control number or size of allocated blocks
  - Must respond immediately to **`malloc`** requests
    - *i.e.*, can't reorder or buffer requests
  - Must allocate blocks from free memory
    - *i.e.*, can only place allocated blocks in free memory
  - Must align blocks so they satisfy all alignment requirements
    - 8-byte (x86) or 16-byte (x86-64) alignment on Linux boxes
  - Can manipulate and modify only free memory
  - Can't move the allocated blocks once they are **`malloc`**'d
    - *i.e.*, compaction is not allowed

Performance Goal: Throughput

- Given some sequence of `malloc` and `free` requests:
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$
- Goals: maximize throughput and peak memory utilization
  - These goals are often conflicting
- Throughput:
  - Number of completed requests per unit time
  - Example:
    - 5,000 **`malloc`** calls and 5,000 **`free`** calls in 10 seconds
    - Throughput is 1,000 operations/second

Performance Goal: Peak Memory Utilization

- Given some sequence of `malloc` and `free` requests:
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$

- *Def: Aggregate payload P_k*
  - **`malloc(p)`** results in a block with a ***payload*** of **`p`** bytes
  - After request R_k completed, the ***aggregate payload*** P_k is the sum of currently allocated payloads

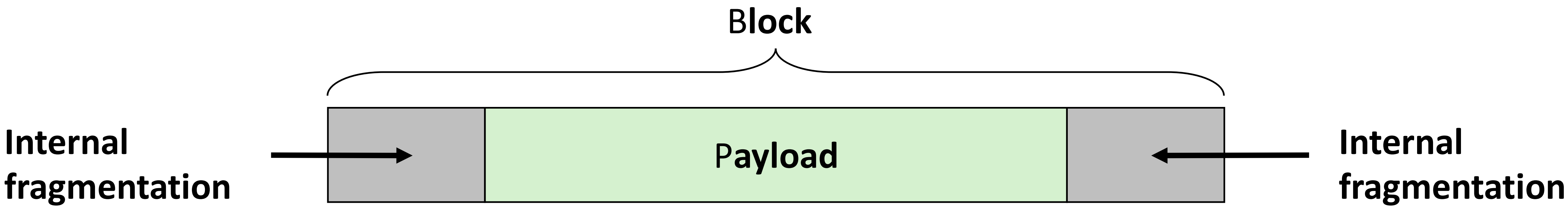
- *Def: Current heap size H_k*
  - Assume H_k is monotonically nondecreasing
    - i.e., heap only grows when allocator uses **`sbrk`**

- *Def: Peak memory utilization after k+1 requests*
  - $U_k = (\max_{i<=k} P_i) \,/\, H_k$

- Poor memory utilization caused by *fragmentation*: ***internal*** fragmentation and ***external*** fragmentation

Internal Fragmentation

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



- Caused by
  - Overhead of maintaining heap data structures
  - Padding for alignment purposes
  - Explicit policy decisions
    (e.g., to return a big block to satisfy a small request)

- Depends only on the pattern of *previous* requests
  - Thus, easy to measure

External Fragmentation

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

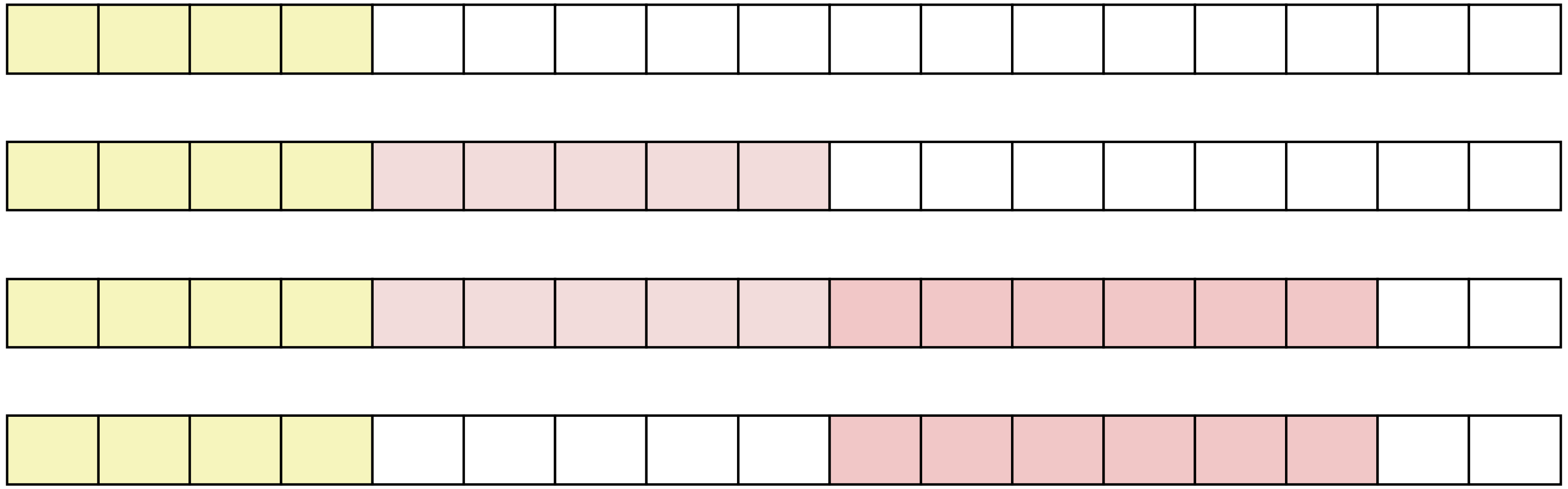
```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(6)
```



Oops! (what would happen now?)

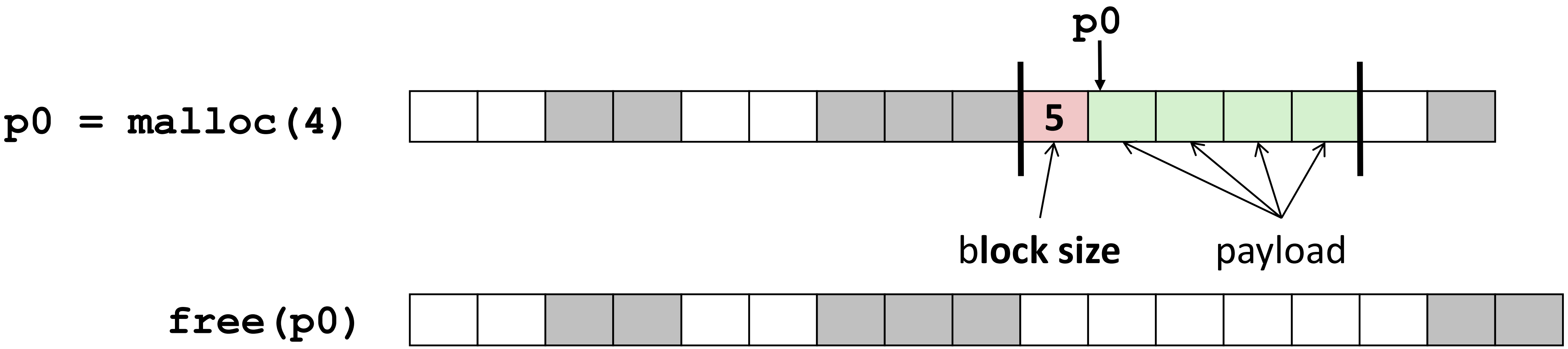
- Depends on the pattern of future requests
  - Thus, difficult to measure

Implementation Issues

- 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 structure 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?

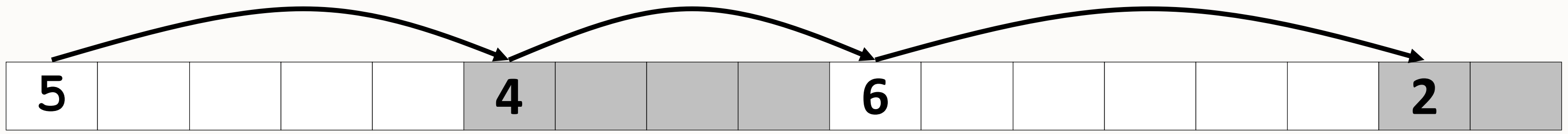
Knowing How Much to Free

- Standard method
  - Keep the length of a block in the word preceding the block.
    - This word is often called the ***header field*** or ***header***
  - Requires an extra word for every allocated block

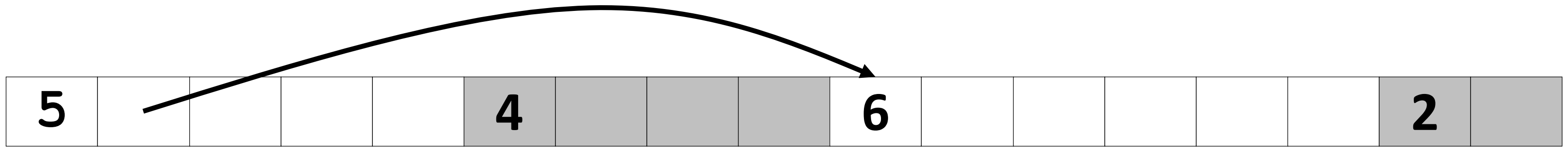


Keeping Track of Free Blocks

- 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 4: *Blocks sorted by size*
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

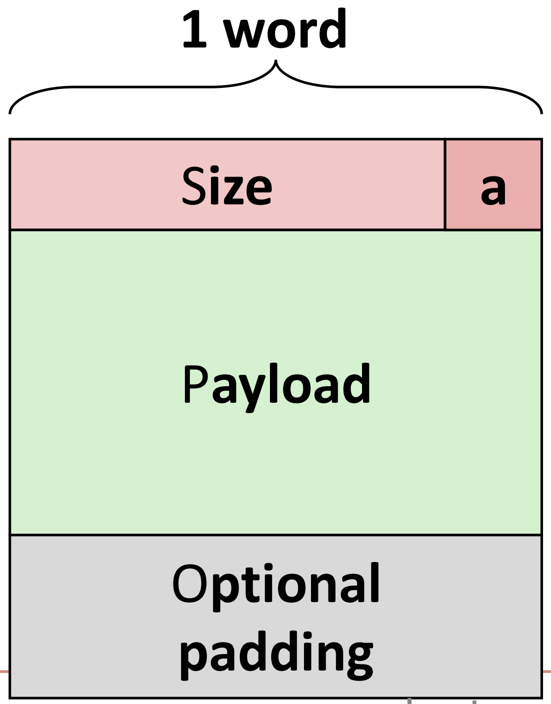
Today

- Basic concepts
- Implicit free lists
- Explicit free lists
- Segregated free lists

Method 1: Implicit List

- For each block we need both size and allocation status
  - Could store this information in two words: wasteful!
- Standard trick
  - If blocks are aligned, some low-order address bits are always 0
  - Instead of storing an always-0 bit, use it as a allocated/free flag
  - When reading size word, must mask out this bit

Format of allocated and free blocks

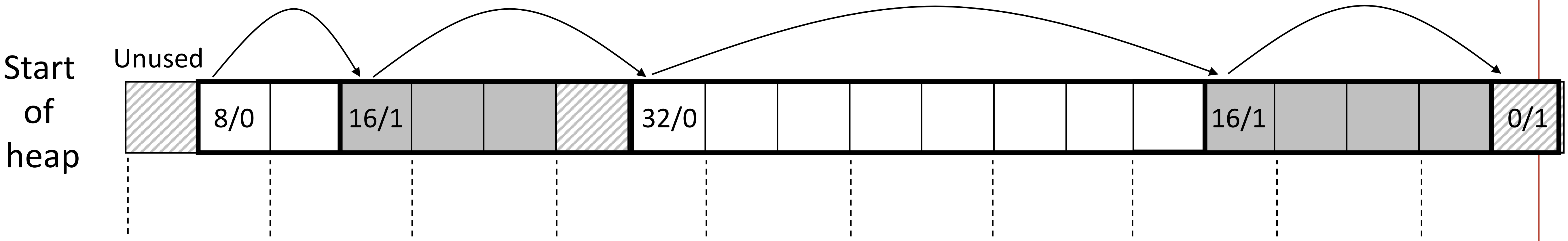


a = 1: Allocated block
a = 0: Free block

Size: block size

Payload: application data (allocated blocks only)

Detailed Implicit Free List Example



Double-word aligned

Allocated blocks: shaded
Free blocks: unshaded
Headers: labeled with size in bytes/allocated bit

Implicit List: Finding a Free Block

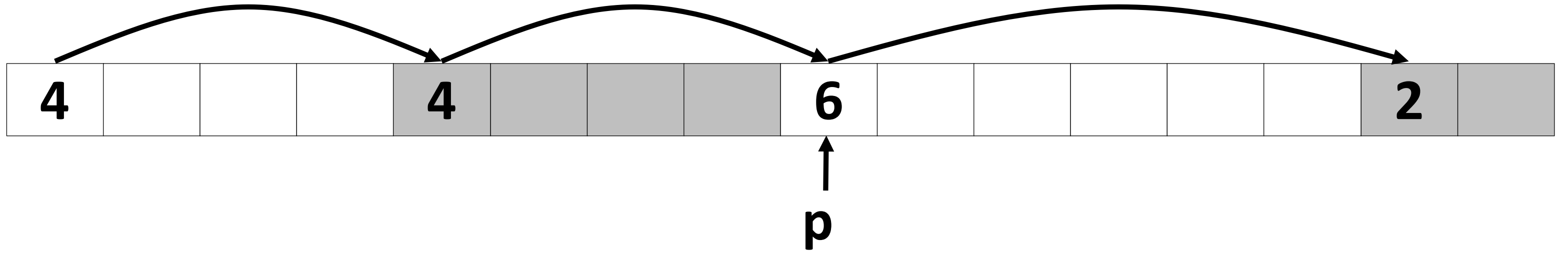
- *First fit:*
  - Search list from beginning, choose ***first*** free block that fits:

```
p = start;
while ((p < end) &&        \\ not passed end
       ((*p & 1) ||        \\ already allocated
       (*p  <= len)))      \\ too small
  p = p + (*p & -2);       \\ goto next block (word addressed)
```

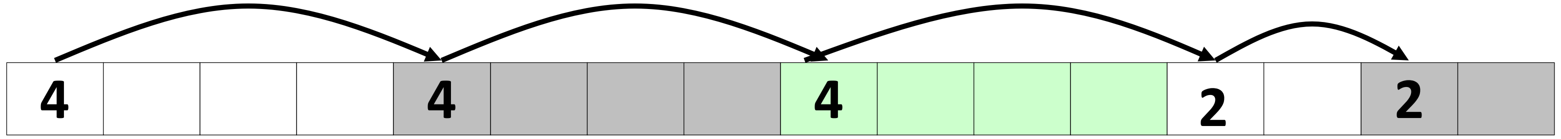
  - Can take linear time in total number of blocks (allocated and free)
  - In practice it can cause "splinters" at beginning of list
- *Next fit:*
  - Like first fit, but search list starting where previous search finished
  - Should often be faster than first fit: avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is worse
- *Best fit:*
  - Search the list, choose the ***best*** free block: fits, with fewest bytes left over
  - Keeps fragments small—usually improves memory utilization
  - 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



`addblock(p, 4)`



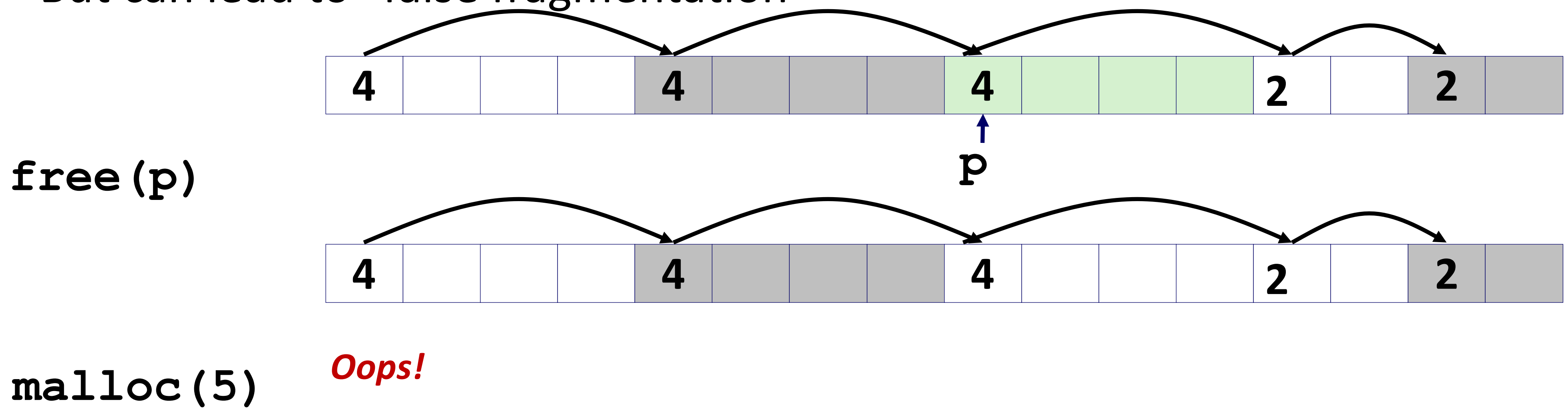

```
void addblock(ptr p, int len) {
  int newsize = ((len + 1) >> 1) << 1;  // round up to even
  int oldsize = *p & -2;                // mask out low bit
  *p = newsize | 1;                     // set new length
  if (newsize < oldsize)
    *(p+newsize) = oldsize - newsize;   // set length in remaining
}                                       //   part of block
```

Implicit List: Freeing a Block

- Simplest implementation:
  - Need only clear the "allocated" flag

```
void free_block(ptr p) { *p = *p & -2 }
```

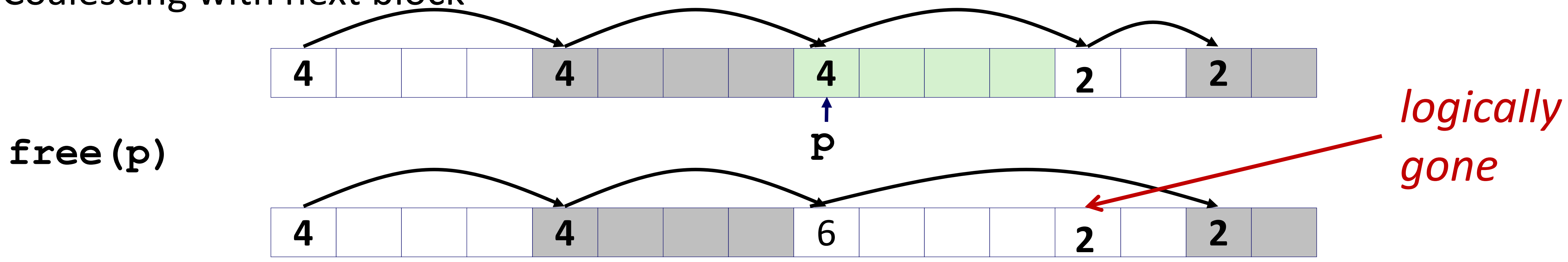
  - But can lead to "false fragmentation"



There is enough free space, but the allocator won't be able to find it

Implicit List: Coalescing

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

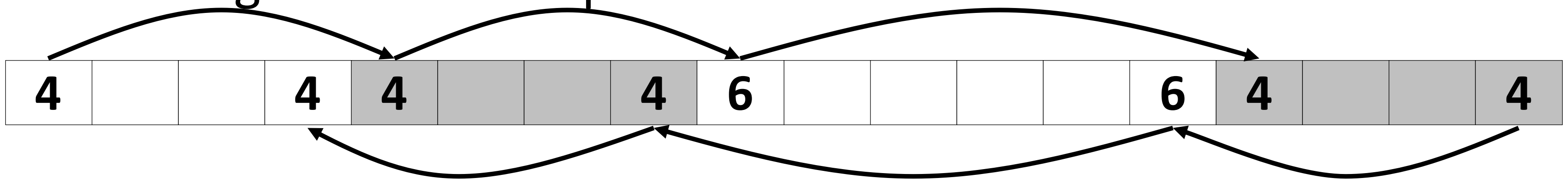



```
void free_block(ptr p) {
    *p = *p & -2;          // clear allocated flag
    next = p + *p;         // find next block
    if ((*next & 1) == 0)
      *p = *p + *next;     // add to this block if
}                          //    not allocated
```

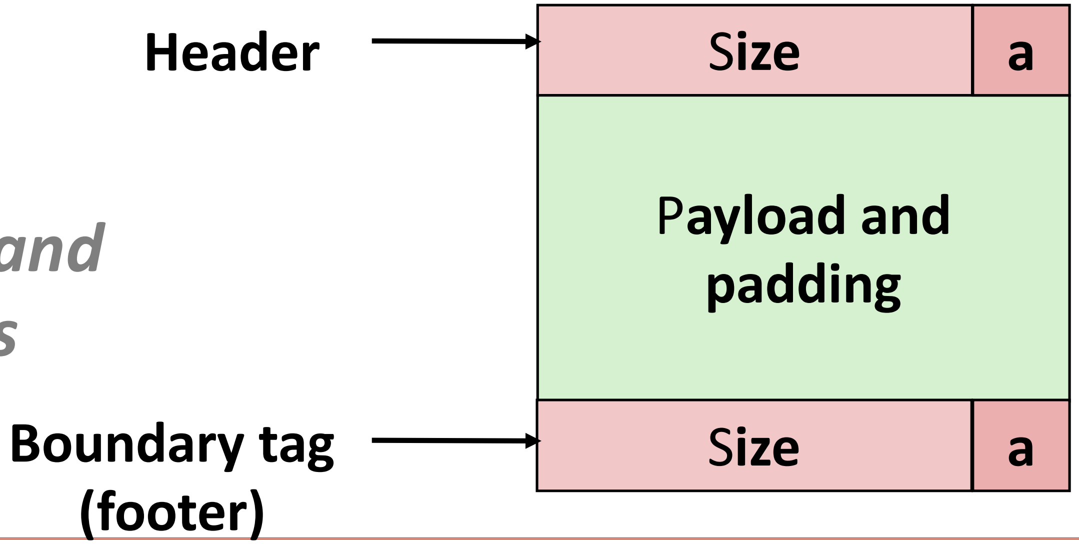
- But how do we coalesce with *previous* block?

Implicit List: Bidirectional Coalescing

- *Boundary tags* [Knuth73]
  - Replicate size/allocated word at "bottom" (end) of free blocks
  - Allows us to traverse the "list" backwards, but requires extra space
  - Important and general technique!



Format of allocated and free blocks

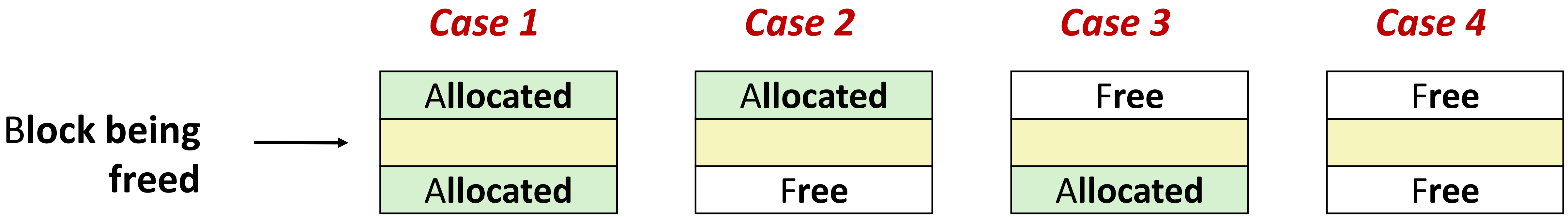


a = 1: Allocated block
a = 0: Free block

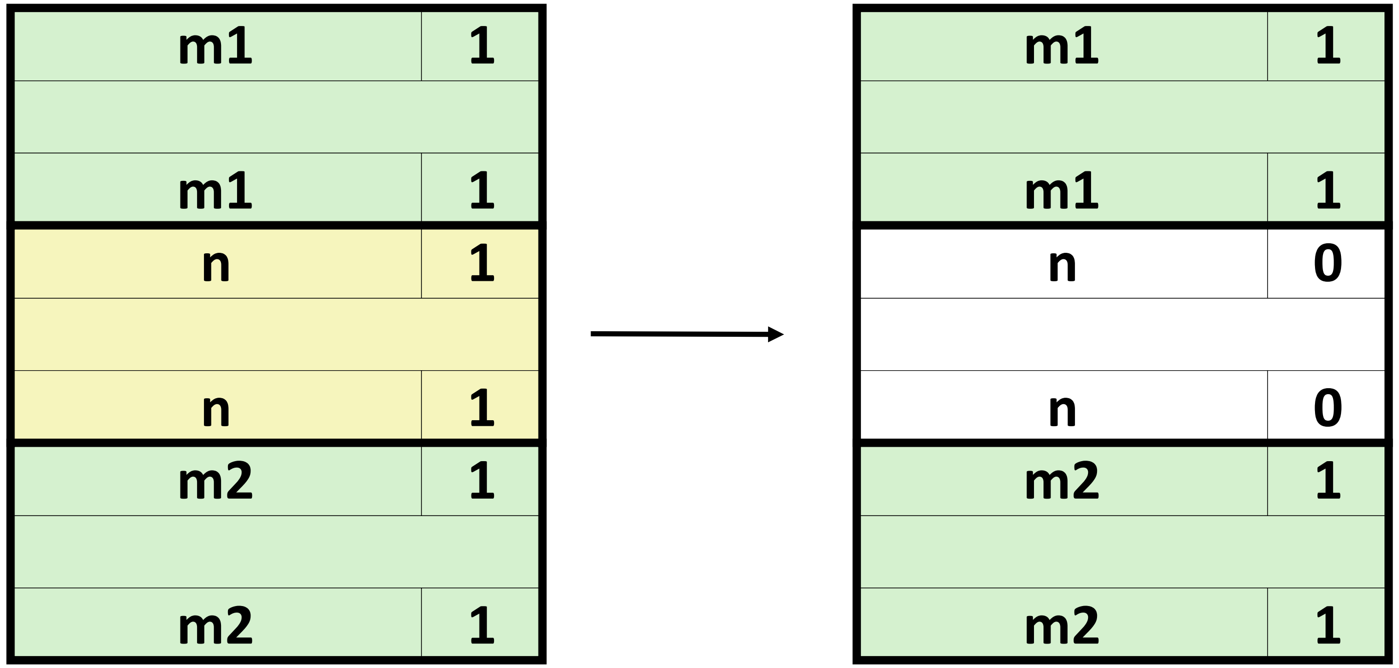
Size: Total block size

Payload: Application data (allocated blocks only)

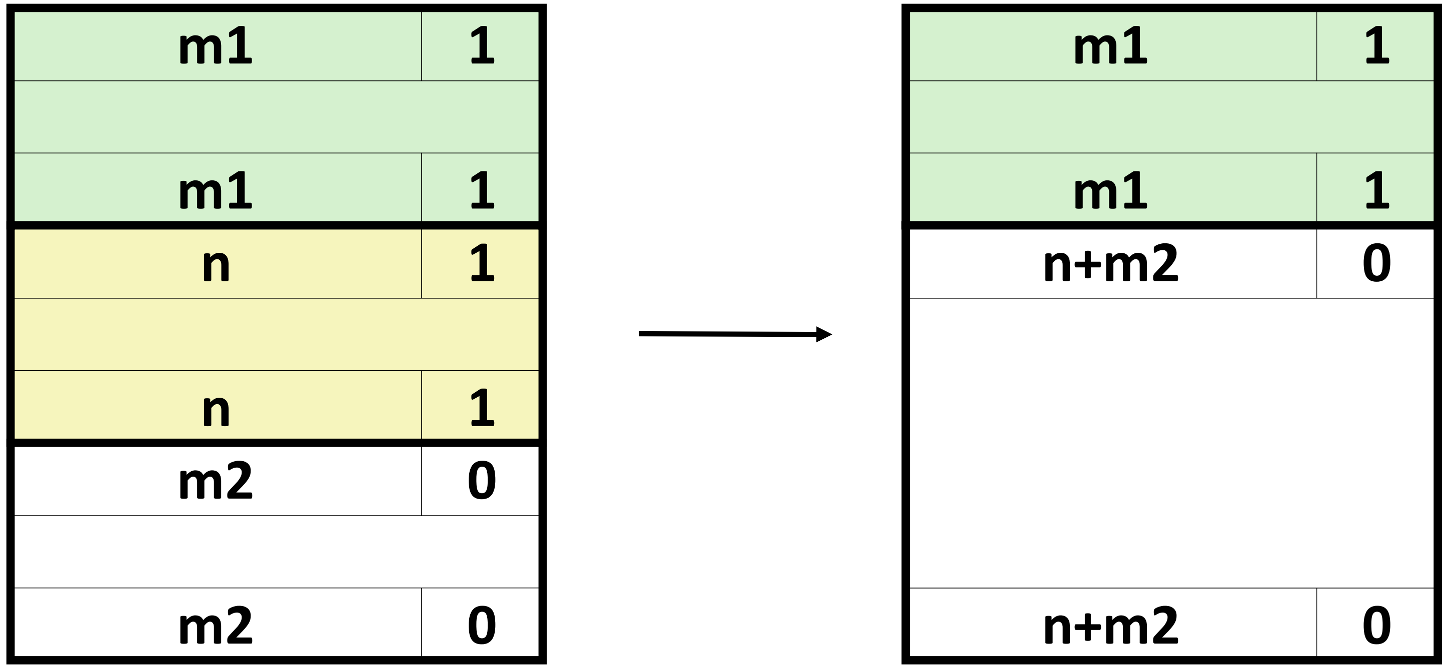
Constant Time Coalescing



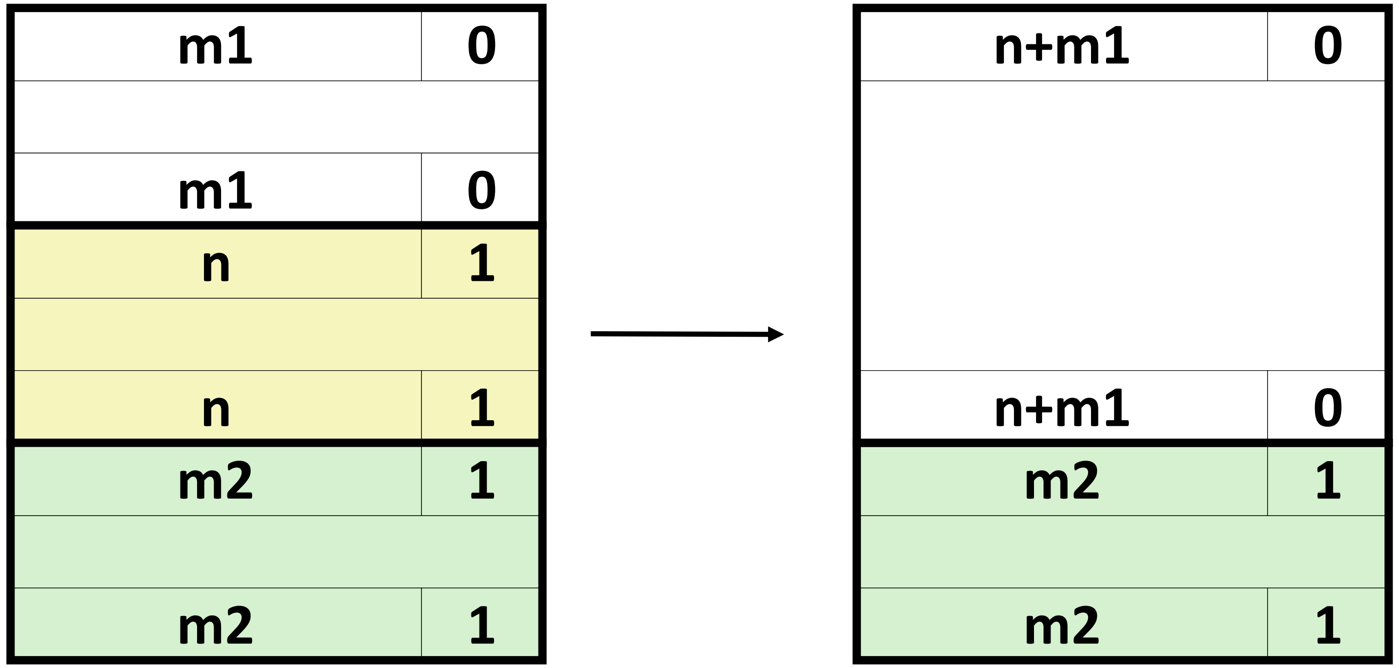
Constant Time Coalescing (Case 1)



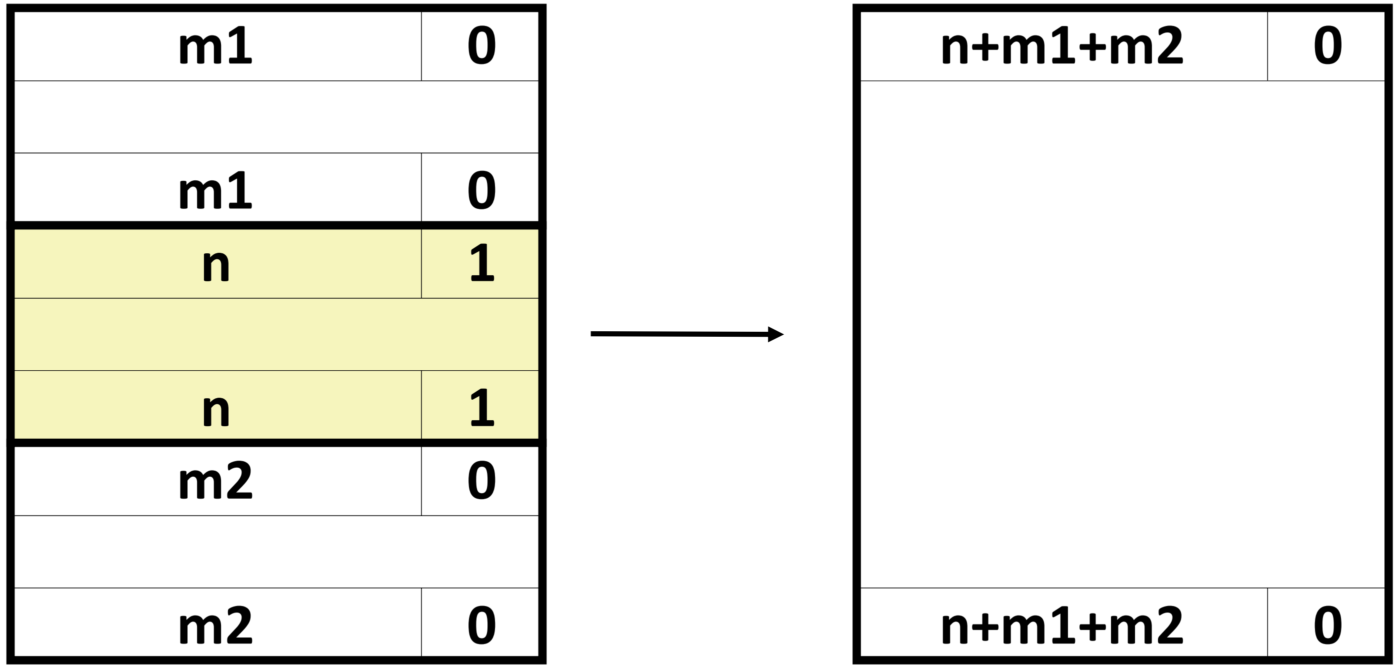
Constant Time Coalescing (Case 2)



Constant Time Coalescing (Case 3)



Constant Time Coalescing (Case 4)



Disadvantages of Boundary Tags

- Internal fragmentation

- Can it be optimized?
  - Which blocks need the footer tag? Only free blocks
  - What does that mean? Use another free bits to indicate free/allocated blocks

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 (next lecture) 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

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 placement policy
  - 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

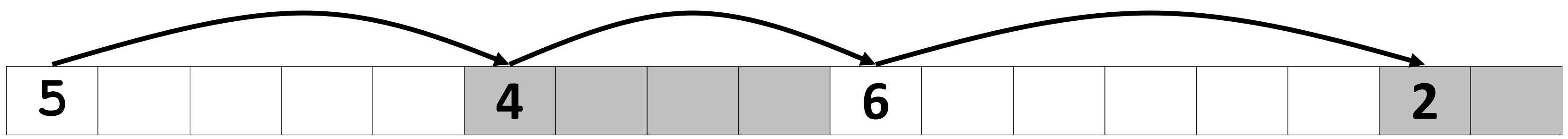
- However, the concepts of splitting and boundary tag coalescing are general to *all* allocators

Today

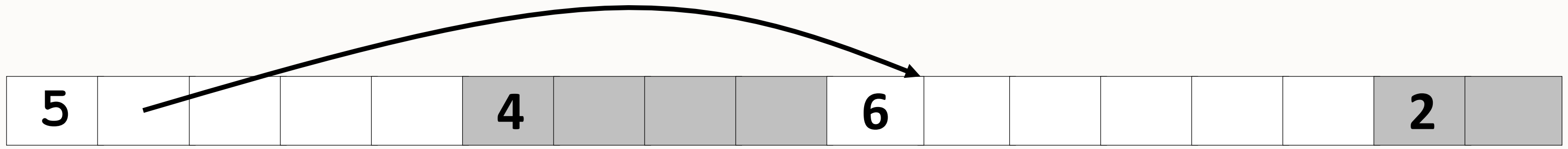
- Basic concepts
- Implicit free lists
- Explicit free lists
- Segregated free lists

Keeping Track of Free Blocks

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

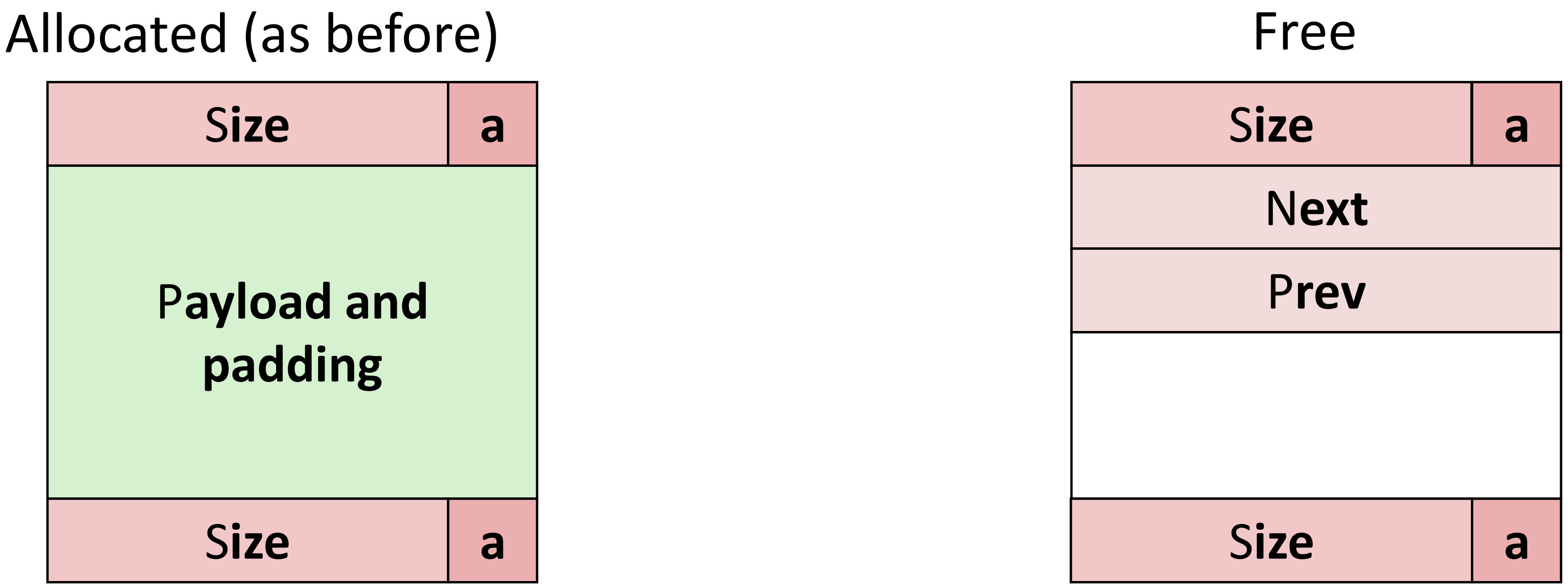


- Method 2: *Explicit free list* among the free blocks using pointers



- Method 3: *Segregated free list*
  - Different free lists for different size classes
- Method 4: *Blocks sorted by size*
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

Explicit Free Lists

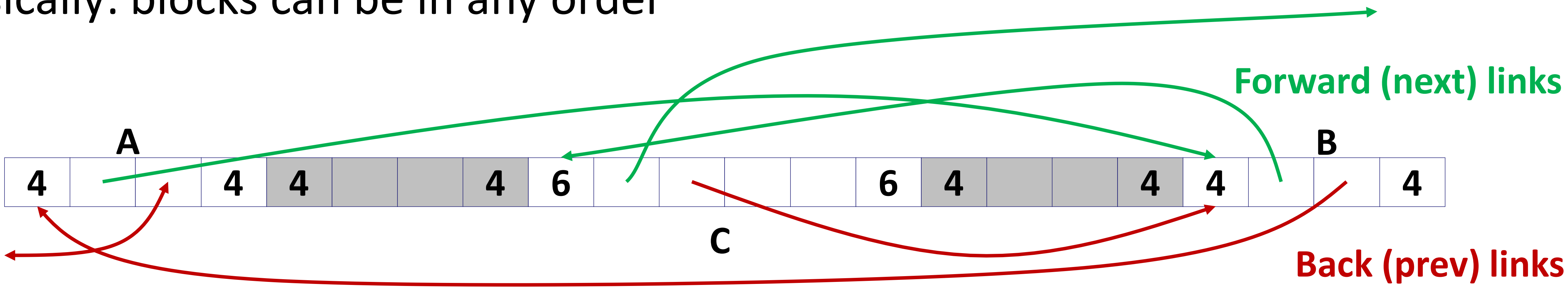


- Maintain list(s) of *free* blocks, not *all* blocks
  - The "next" free block could be anywhere
    - So we need to store forward/back pointers, not just sizes
  - Still need boundary tags for coalescing
  - Luckily we track only free blocks, so we can use payload area

Explicit Free Lists

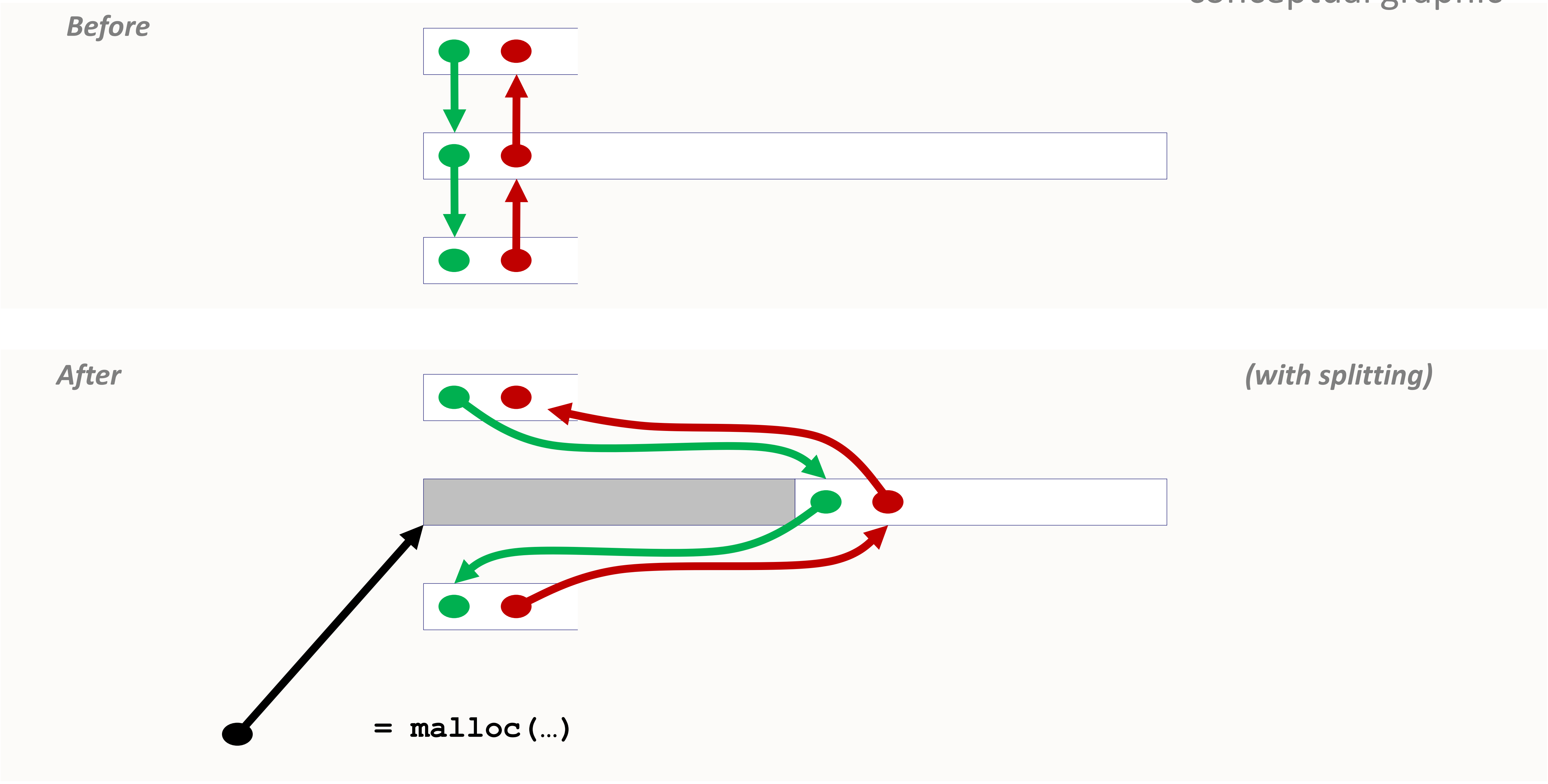
- Logically:

- Physically: blocks can be in any order



Allocating From Explicit Free Lists

conceptual graphic

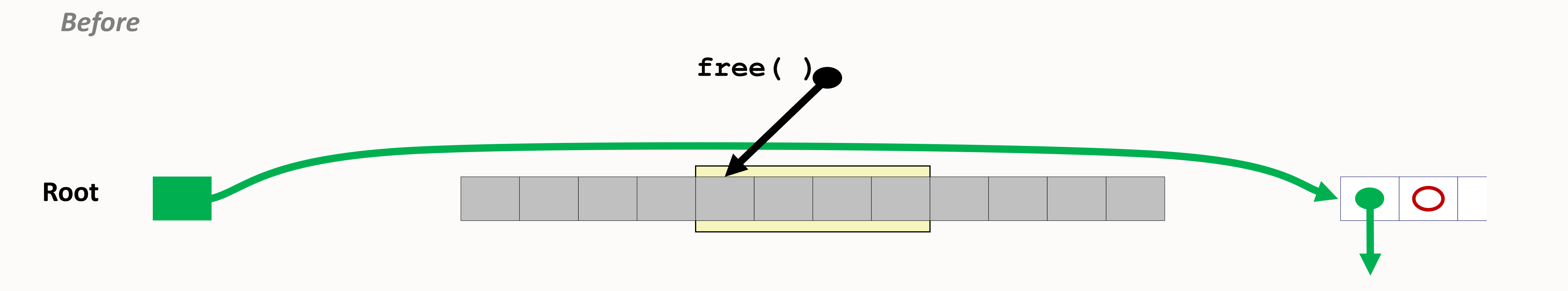


Freeing With Explicit Free Lists

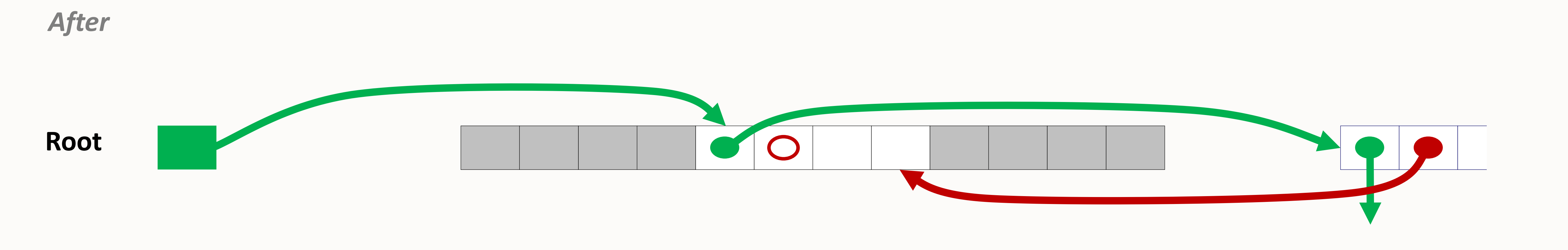
- *Insertion policy*: Where in the free list do you put a newly freed block?
- **LIFO (last-in-first-out) policy**
  - Insert freed block at the beginning of the free list
  - ***Pro:*** simple and constant time
  - ***Con:*** studies suggest fragmentation is worse than address ordered
- **Address-ordered policy**
  - Insert freed blocks so that free list blocks are always in address order:
    addr(prev) < addr(curr) < addr(next)
  - ***Con:*** requires search
  - ***Pro:*** studies suggest fragmentation is lower than LIFO

Freeing With a LIFO Policy (Case 1)

conceptual graphic

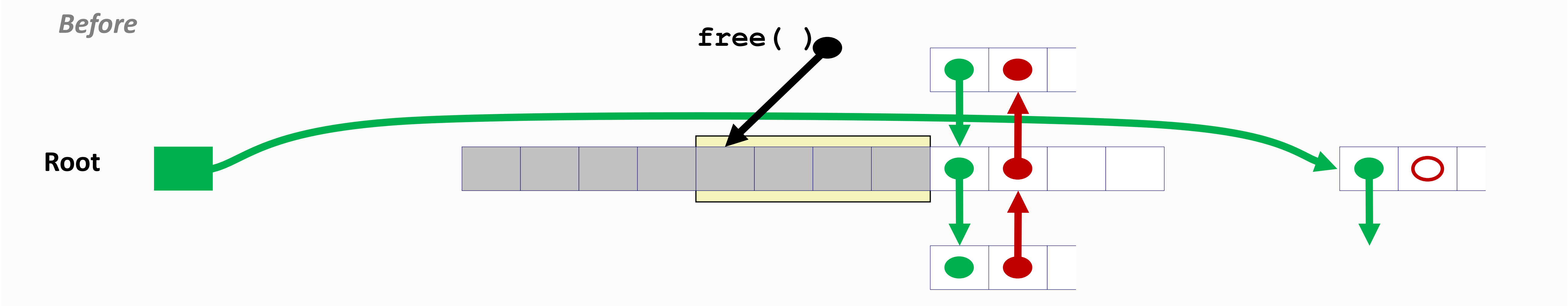


- Insert the freed block at the root of the list

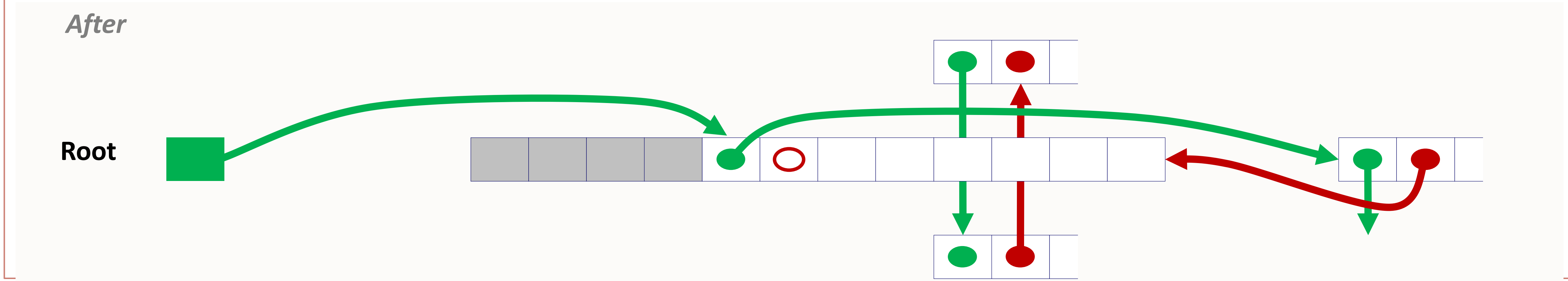


Freeing With a LIFO Policy (Case 2)

conceptual graphic

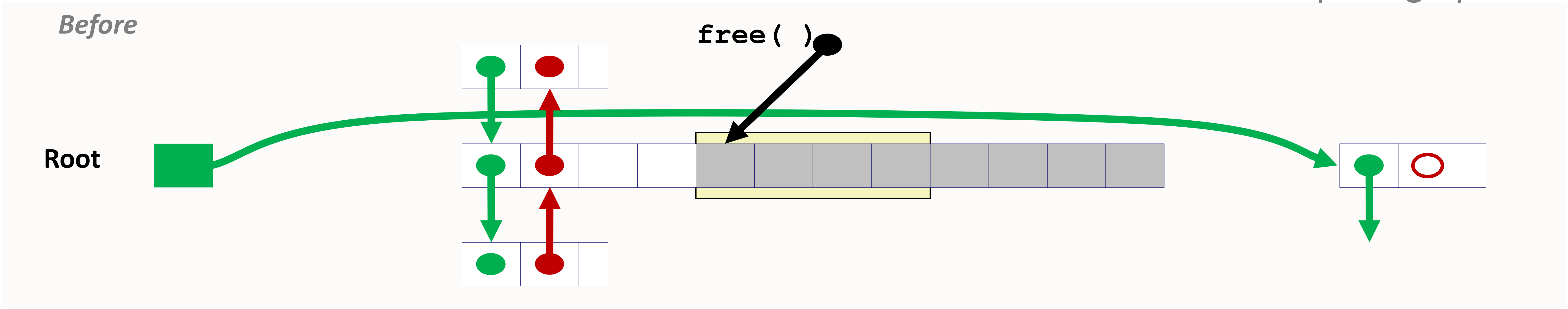


- Splice out successor block, coalesce both memory blocks and insert the new block at the root of the list

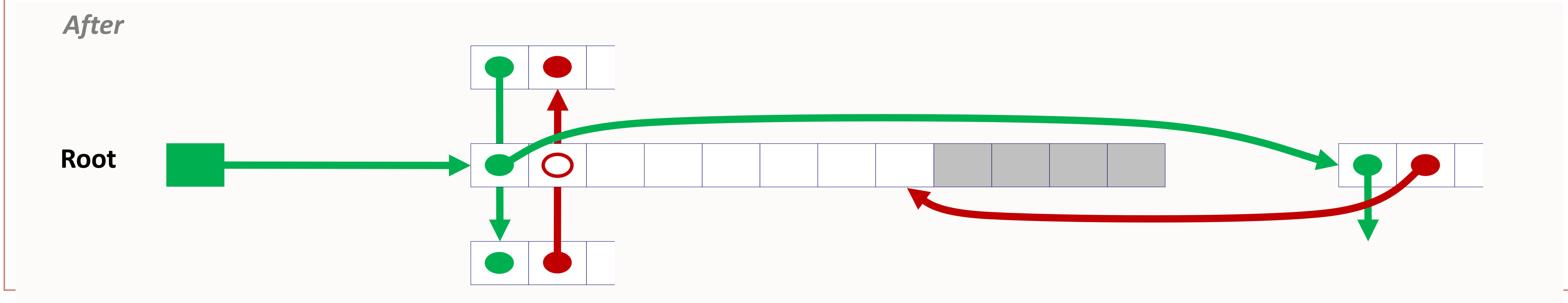


Freeing With a LIFO Policy (Case 3)

conceptual graphic

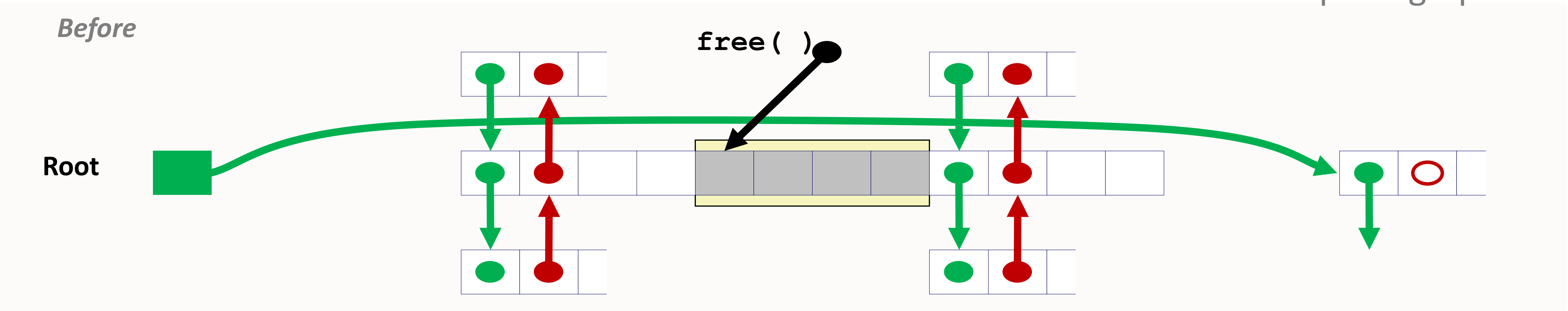


- Splice out predecessor block, coalesce both memory blocks, and insert the new block at the root of the list

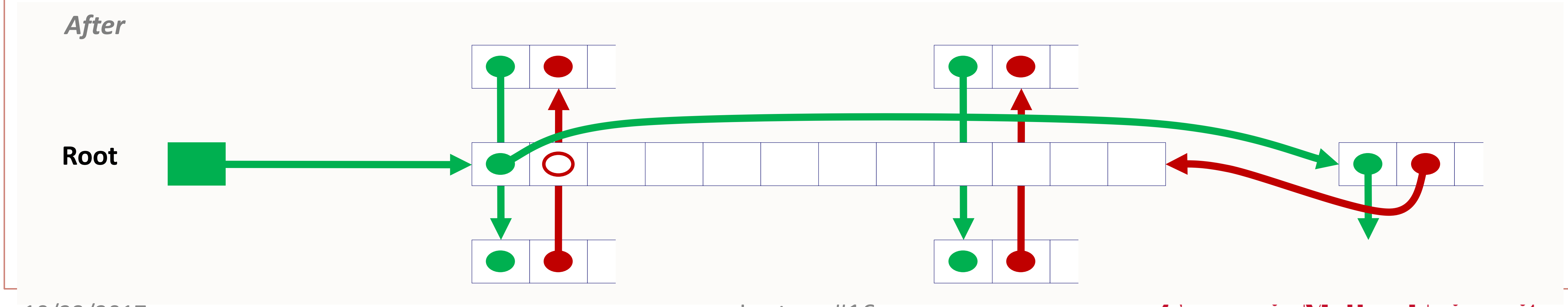


Freeing With a LIFO Policy (Case 4)

conceptual graphic



- Splice out predecessor and successor blocks, coalesce all 3 memory blocks and insert the new block at the root of the list



Explicit List Summary

- Comparison to implicit list:
  - Allocate is linear time in number of ***free*** blocks instead of ***all*** blocks
    - ***Much faster*** when most of the memory is full
  - Slightly more complicated allocate and free since needs to splice blocks in and out of the list
  - Some extra space for the links (2 extra words needed for each block)
    - Does this increase internal fragmentation?

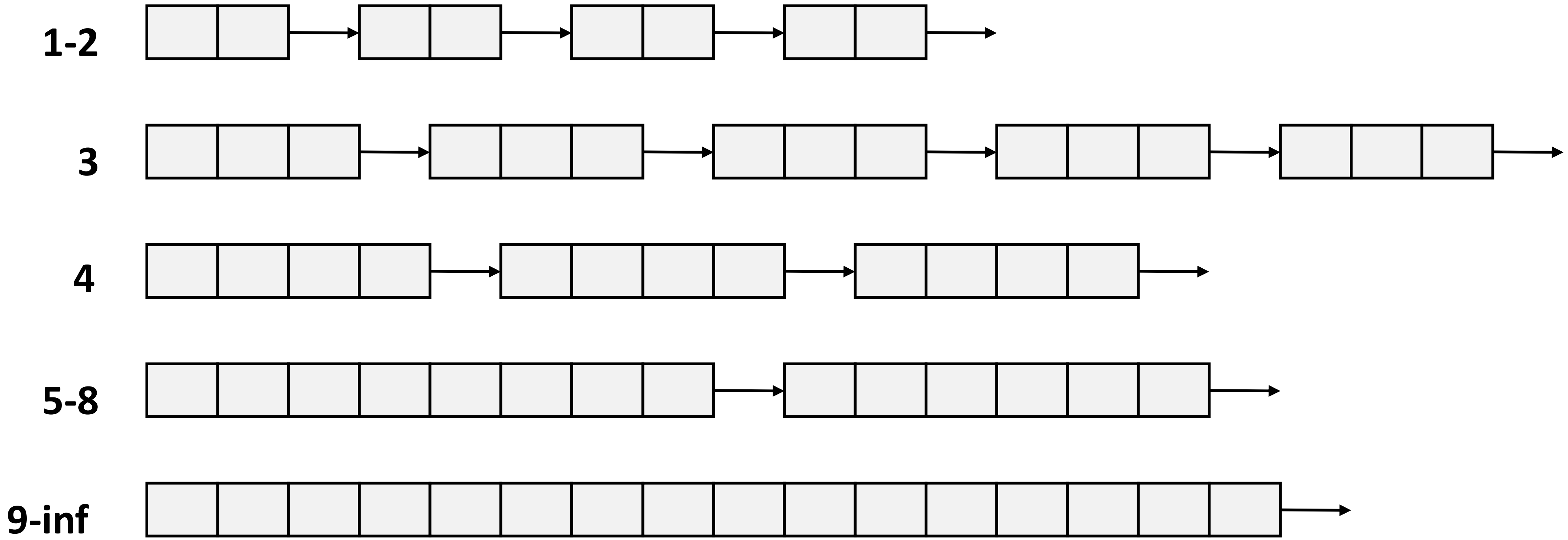
- Most common use of linked lists is in conjunction with segregated free lists
  - Keep multiple linked lists of different size classes, or possibly for different types of objects

Today

- Basic concepts
- Implicit free lists
- Explicit free lists
- Segregated free lists

Segregated List (Seglist) Allocators

- 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

Seglist Allocator

- Given an array of free lists, each one for some size class

- To allocate a block of size *n*:
  - Search appropriate free list for block of size *m* > *n*
  - If an appropriate block is found:
    - Split block and place fragment on appropriate list (optional)
  - If no block is found, try next larger class
  - Repeat until block is found

- If no block is found:
  - Request additional heap memory from OS (using **`sbrk()`**)
  - 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
  - Higher throughput
    - log time for power-of-two size classes
  - Better memory utilization
    - First-fit search of segregated free list approximates a best-fit search of entire heap.
    - Extreme case: Giving each block its own size class is equivalent to best-fit.

Today

- Basic concepts
- Implicit free lists
- Explicit free lists
- Segregated free lists

18-600 Foundations of Computer Systems

Lecture 17:
"Multicore Cache Coherence"

October 25, 2017

Next Time …

Electrical & Computer ENGINEERING