Virtual Memory Systems and Dynamic Memory Allocation

Computer Systems

Nov. 5, 2018

Troels Henriksen

Based on slides by:

Randal E. Bryant and David R. O'Hallaron

Today

- **Simple memory system example**
- Case study: Linux memory system
- Memory mapping
- Dynamic Allocation
- Bonus: Implicit Free Lists

Review of Symbols

- **Basic Parameters**
  - **N = 2^n** : Number of addresses in virtual address space
  - **M = 2^m** : Number of addresses in physical address space
  - **P = 2^p** : Page size (bytes)

- **Components of the virtual address (VA)**
  - **TLBI**: TLB index
  - **TLBT**: TLB tag
  - **VPO**: Virtual page offset
 - **VPN**: Virtual page number

- **Components of the physical address (PA)**
  - **PPO**: Physical page offset (same as VPO)
  - **PPN:** Physical page number
  - **CO**: Byte offset within cache line
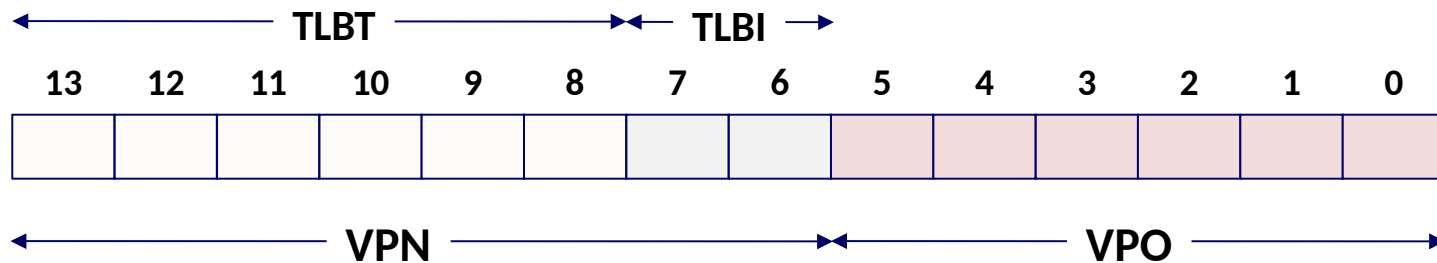 - **CI:** Cache index
  - **CT**: Cache tag

Simple Memory System Example

- **Addressing**
  - 14-bit virtual addresses
  - 12-bit physical address
  - Page size = 64 bytes

13	12	11	10	9	8	7	6	5	4	3	2	1	0

← ——————————— VPN ———————————— → ← ——————— VPO ——————— →

Virtual Page Number   **Virtual Page Offset**

11	10	9	8	7	6	5	4	3	2	1	0

← ——————— PPN ——————— → ← ——————— PPO ——————— →

Physical Page Number   **Physical Page Offset**
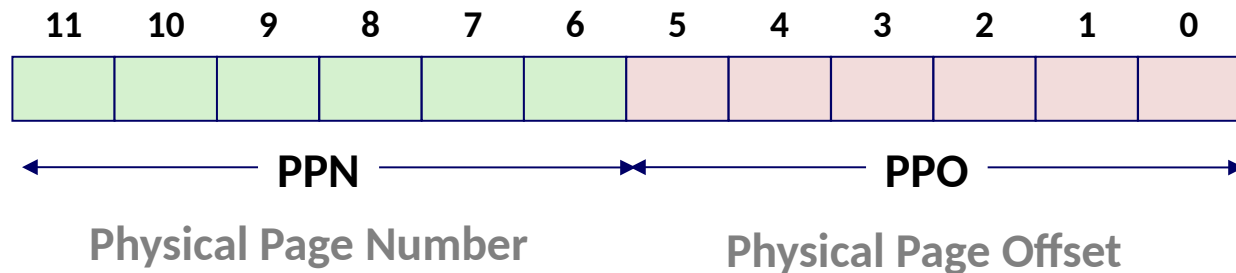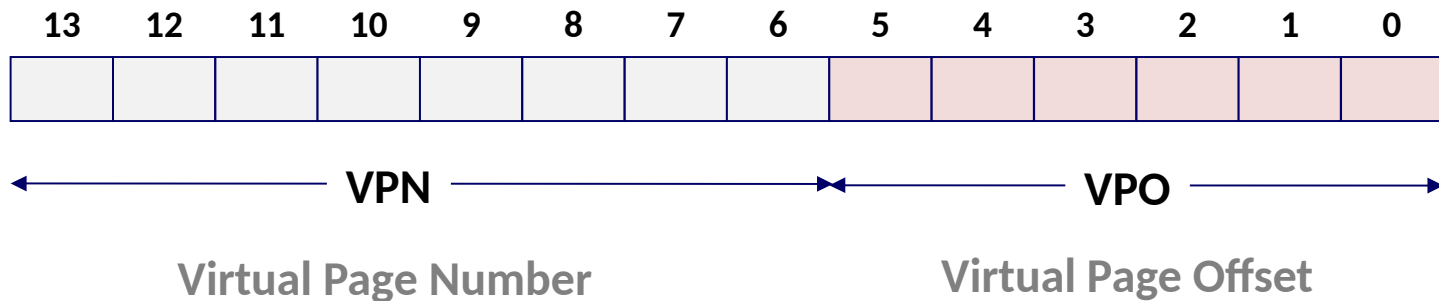
1. Simple Memory System TLB

- **16 entries**
- **4-way associative**

Set	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid	Tag	PPN	Valid
0	03	–	0	09	0D	1	00	–	0	07	02	1
1	03	2D	1	02	–	0	04	–	0	0A	–	0
2	02	–	0	08	–	0	06	–	0	03	–	0
3	07	–	0	03	0D	1	0A	34	1	02	–	0

2. Simple Memory System Page Table

Only show first 16 entries (out of 256)

VPN	PPN	Valid
00	28	1
01	–	0
02	33	1
03	02	1
04	–	0
05	16	1
06	–	0
07	–	0

VPN	PPN	Valid
08	13	1
09	17	1
0A	09	1
0B	–	0
0C	–	0
0D	2D	1
0E	11	1
0F	0D	1

3. Simple Memory System Cache

- **16 lines, 4-byte block size**
- **Physically addressed**
- **Direct mapped**
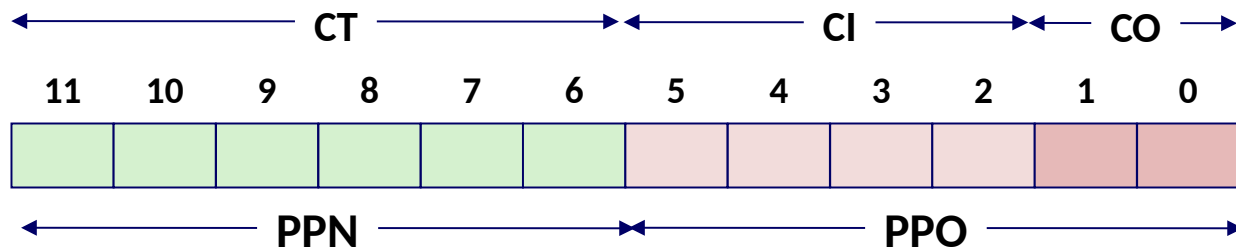
Idx	Tag	Valid	B0	B1	B2	B3
0	19	1	99	11	23	11
1	15	0	–	–	–	–
2	1B	1	00	02	04	08
3	36	0	–	–	–	–
4	32	1	43	6D	8F	09
5	0D	1	36	72	F0	1D
6	31	0	–	–	–	–
7	16	1	11	C2	DF	03

Idx	Tag	Valid	B0	B1	B2	B3
8	24	1	3A	00	51	89
9	2D	0	–	–	–	–
A	2D	1	93	15	DA	3B
B	0B	0	–	–	–	–
C	12	0	–	–	–	–
D	16	1	04	96	34	15
E	13	1	83	77	1B	D3
F	14	0	–	–	–	–

Bryant and O'Hallaron, Computer Systems: A Programmer's Perspective, Third Edition

Address Translation Example #1

Virtual Address: 0x03D4

	TLBT						TLBI						
13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	1	1	1	1	0	1	0	1	0	0

VPN ────────────────── VPO
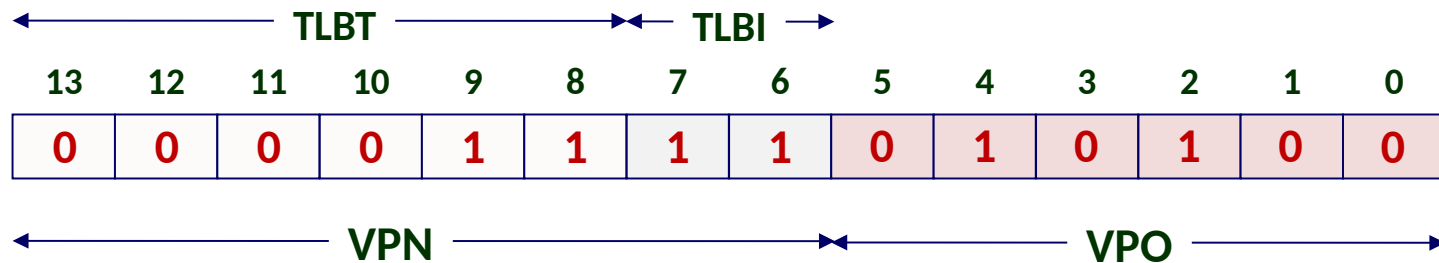
VPN **0x0F**   TLBI **0x3**   TLBT **0x03**   TLB Hit? **Y**   Page Fault? **N**   PPN: **0x0D**

Physical Address

	CT						CI				CO	
11	10	9	8	7	6	5	4	3	2	1	0
0	0	1	1	0	1	0	1	0	1	0	0

PPN ────────────── PPO
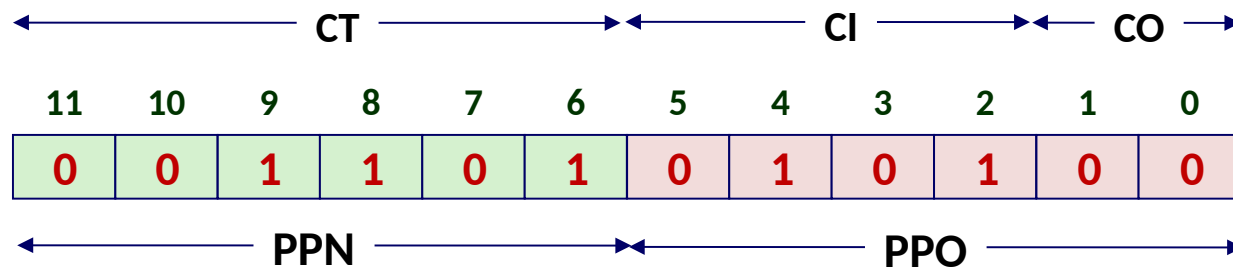
CO **0**   CI **0x5**   CT **0x0D**   Hit? **Y**   Byte: **0x36**

Address Translation Example #2

Virtual Address: 0x0020

TLBT ← → TLBI →

13	12	11	10	9	8	7	6	5	4	3	2	1	0
0	0	0	0	0	0	0	0	1	0	0	0	0	0

← VPN → ← VPO →
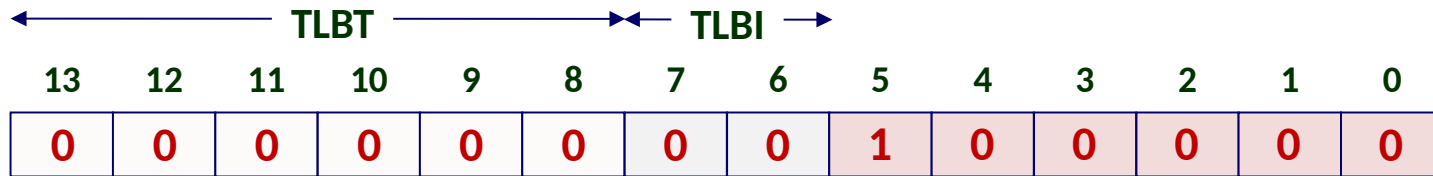
VPN __0x00__   TLBI __0__   TLBT __0x00__   TLB Hit? __N__   Page Fault? __N__   PPN: __0x28__

Physical Address

CT ← → CI → ← CO →

11	10	9	8	7	6	5	4	3	2	1	0
1	0	1	0	0	0	1	0	0	0	0	0

← PPN → ← PPO →
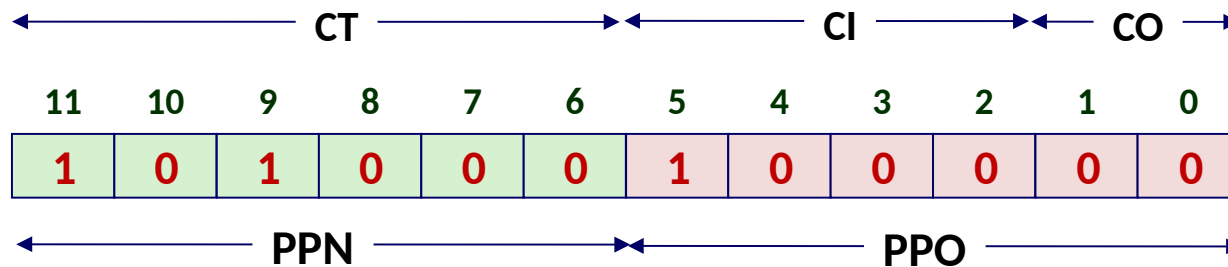
CO __0__   CI __0x8__   CT __0x28__   Hit? __N__   Byte: __Mem__

Today

- **Simple memory system example**
- **Case study: Linux memory system**
- **Memory mapping**
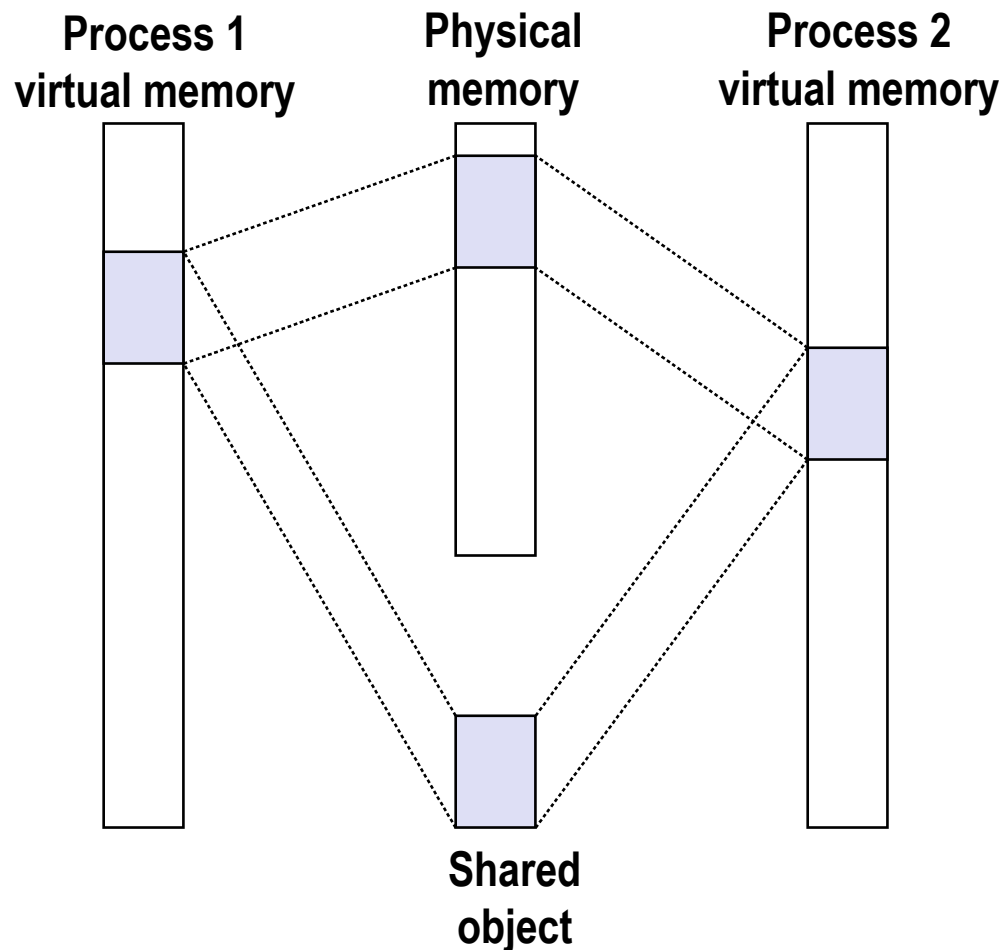- **Dynamic allocation**
- **Bonus: Implicit free lists**

Virtual Address Space of a Linux Process

Different for each process

Process-specific data structs (ptables, task and mm structs, kernel stack)

Identical for each process

Physical memory

Kernel code and data

Kernel virtual memory

User stack

`%rsp` →

Memory mapped region for shared libraries

`brk` →

Runtime heap (malloc)

Uninitialized data (.bss)

Initialized data (.data)

`0x00400000` → Program text (.text)

0

Process virtual memory

Linux Organizes VM as Collection of "Areas"

Process virtual memory

`vm_area_struct`

`task_struct`

`mm_struct`

| mm |

| pgd |
| mmap |

vm_area_struct:

| vm_end |
| vm_start |
| vm_prot |
| vm_flags |
| |

Shared libraries

| vm_end |
| vm_start |
| vm_prot |
| vm_flags |
| |

Data

Text

| vm_end |
| vm_start |
| vm_prot |
| vm_flags |
| vm_next |

0

- **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

vm_area_struct

Process virtual memory

| vm_end |
| vm_start |
| vm_prot |
| vm_flags |
| |

shared libraries
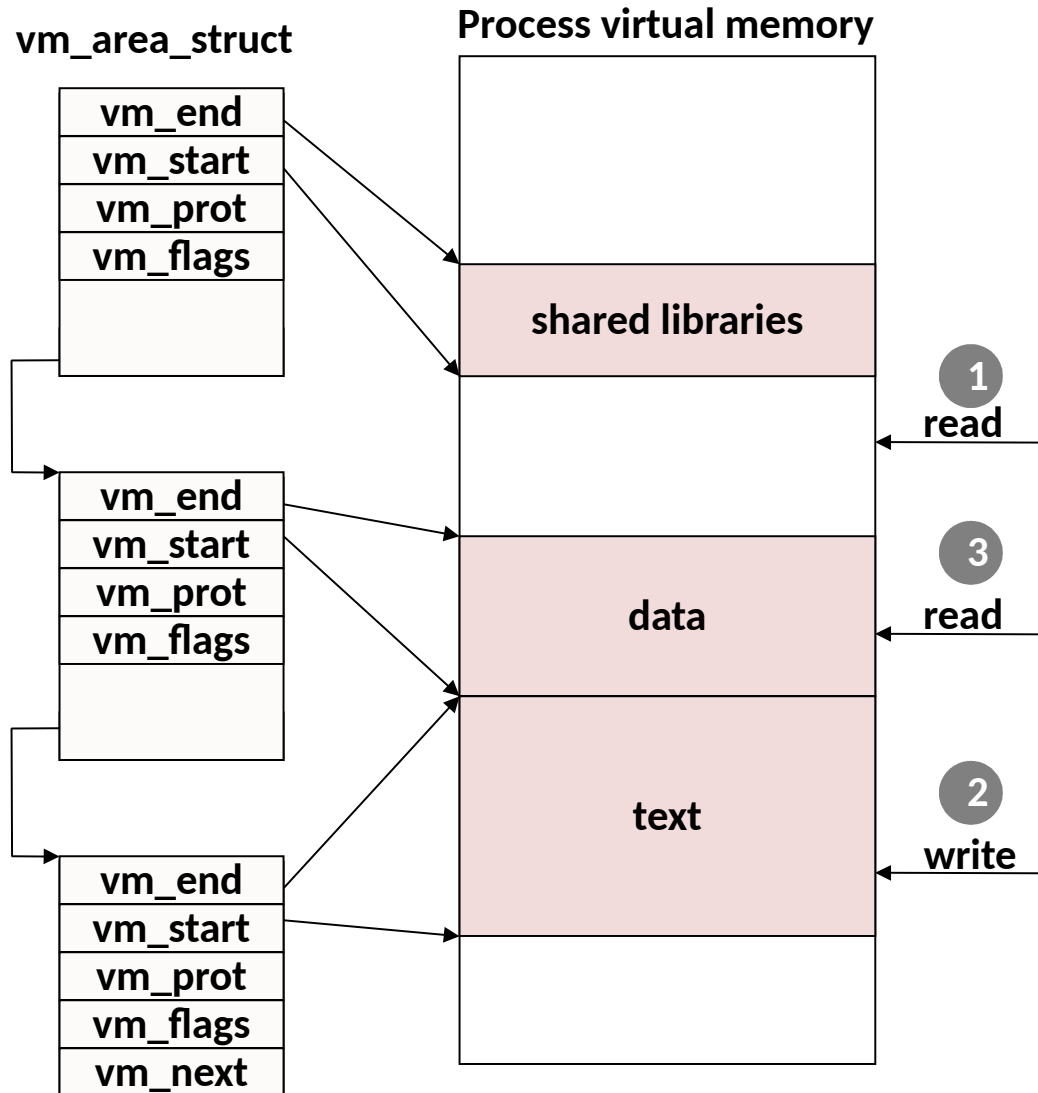
(1) read — **Segmentation fault:** accessing a non-existing page

| vm_end |
| vm_start |
| vm_prot |
| vm_flags |
| |

data

(3) read — **Normal page fault**

text

| vm_end |
| vm_start |
| vm_prot |
| vm_flags |
| vm_next |

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

Today

- **Simple memory system example**
- **Case study: Linux memory system**
- **Memory mapping**
- **Dynamic allocation**
- **Bonus: Implicit free lists**

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 virtual memory

Physical memory

Process 2 virtual memory
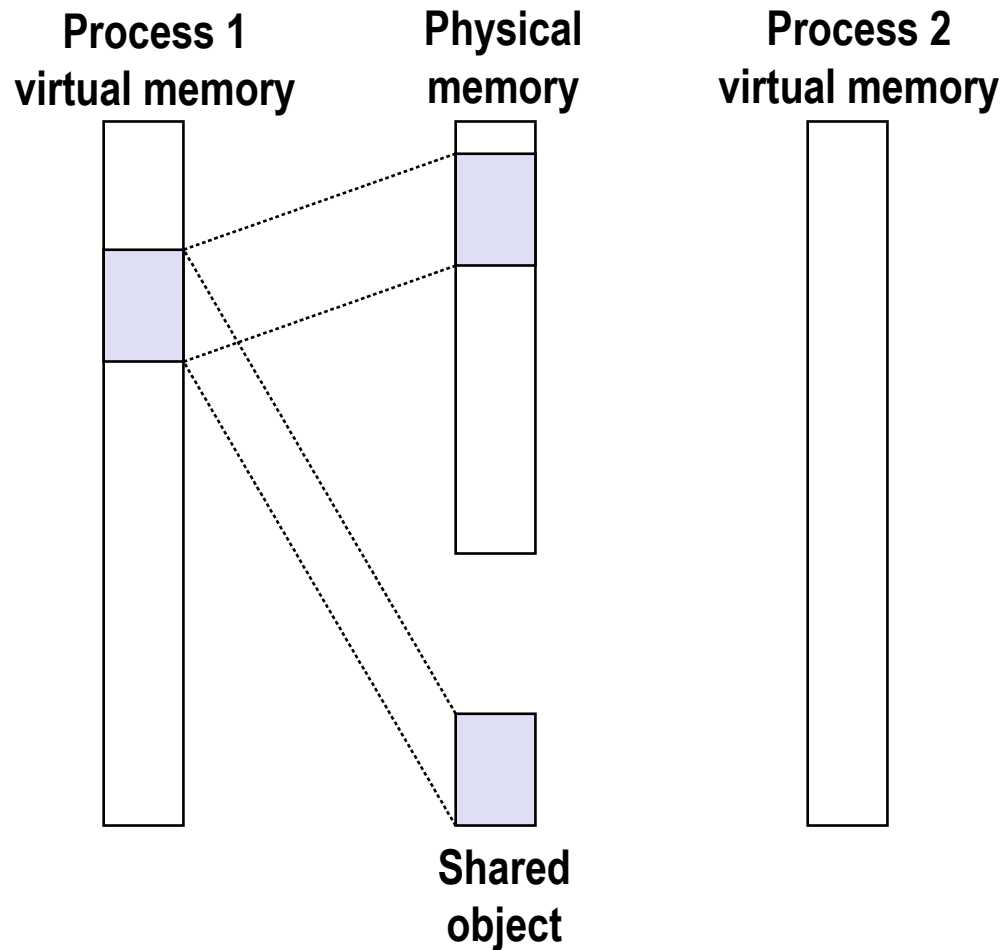
- **Process 1 maps the shared object.**

Shared object

Sharing Revisited: Shared Objects

Process 1 virtual memory

Physical memory

Process 2 virtual memory

Shared object

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

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

Process 1 virtual memory

Physical memory

Process 2 virtual memory

Private copy-on-write area

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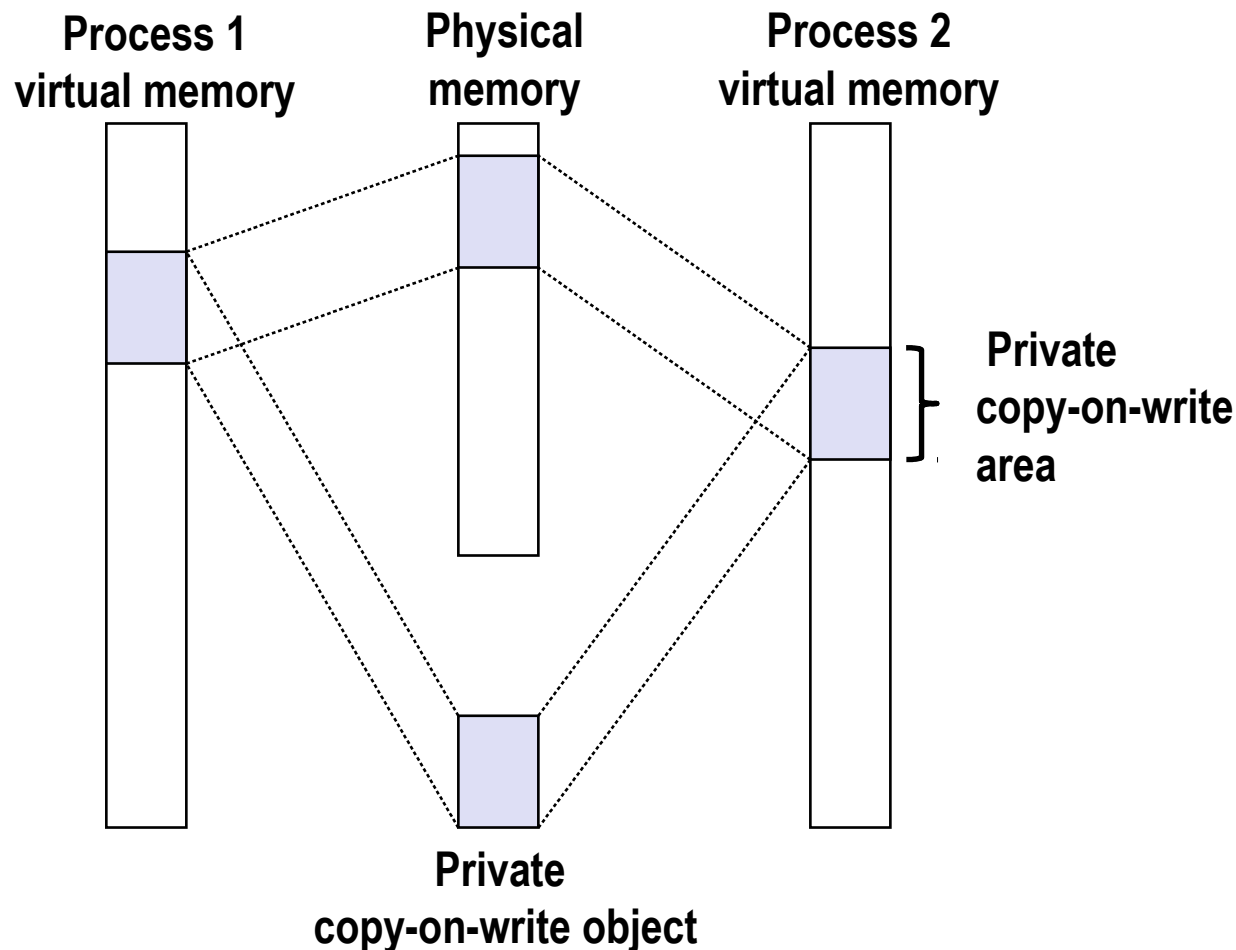
- **Two processes mapping a *private copy-on-write (COW)* object.**
- **Area flagged as private copy-on-write**
- **PTEs in private areas are flagged as read-only**

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

Process 1 virtual memory

Physical memory

Process 2 virtual memory

Copy-on-write

Write to private copy-on-write page

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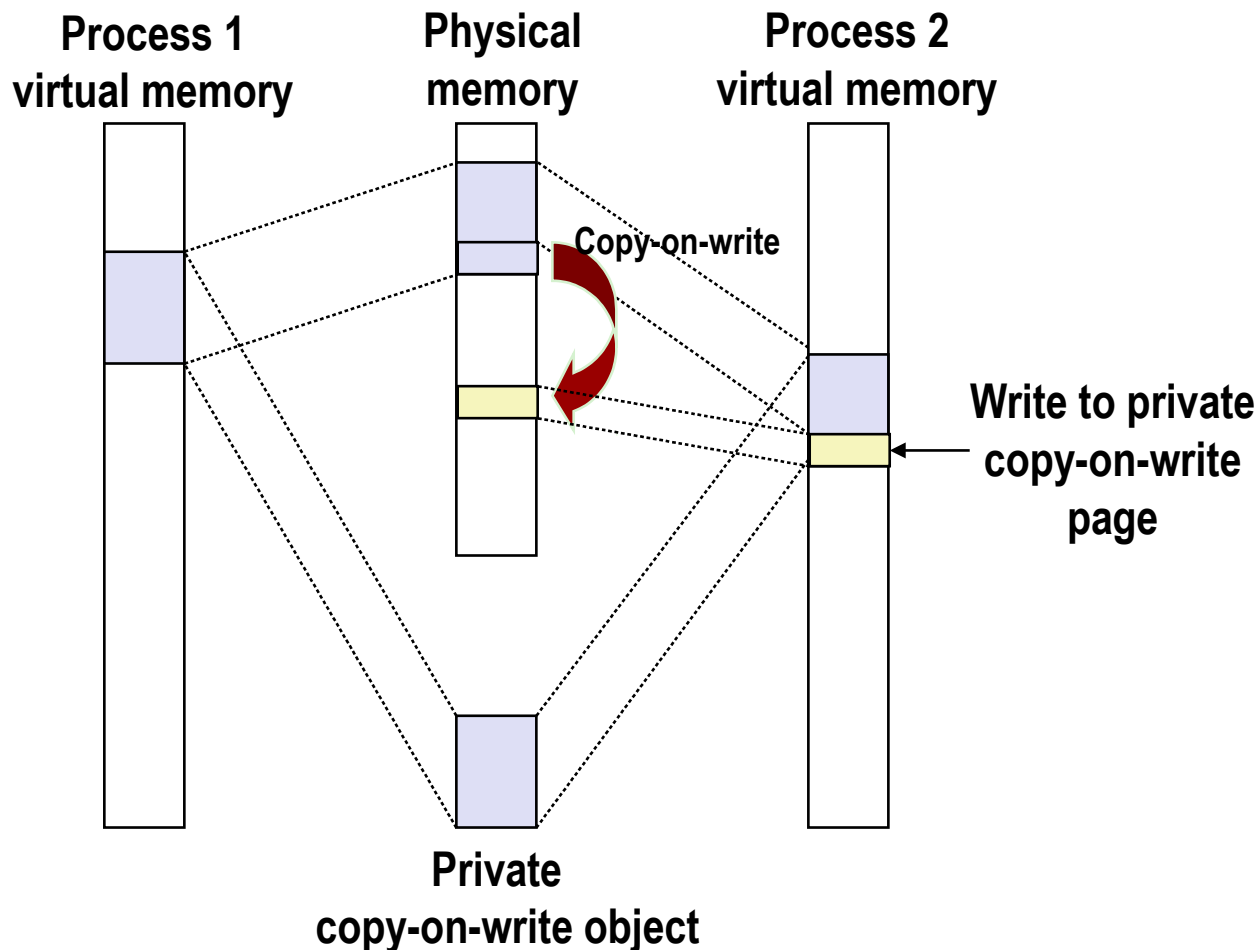
- **Instruction writing to private page triggers protection fault.**
- **Handler creates new R/W page.**
- **Instruction restarts upon handler return.**
- **Copying deferred as long as possible!**

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
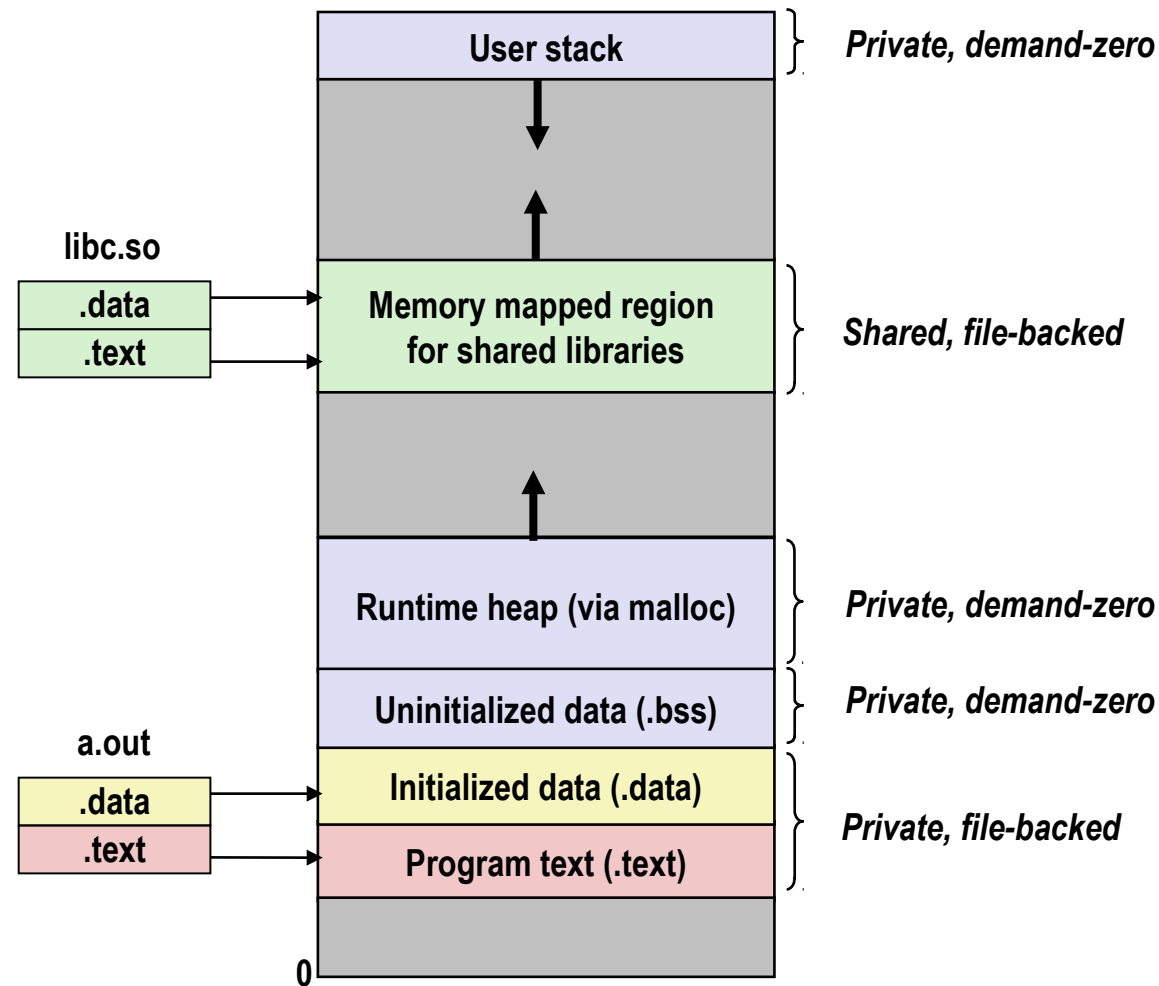
- **To load and run a new program `a.out` in the current process using `execve`:**

- **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)
```

len bytes

len bytes

start
(or address
chosen by kernel)

offset
(bytes)

Disk file specified by
file descriptor **fd**

Process virtual memory
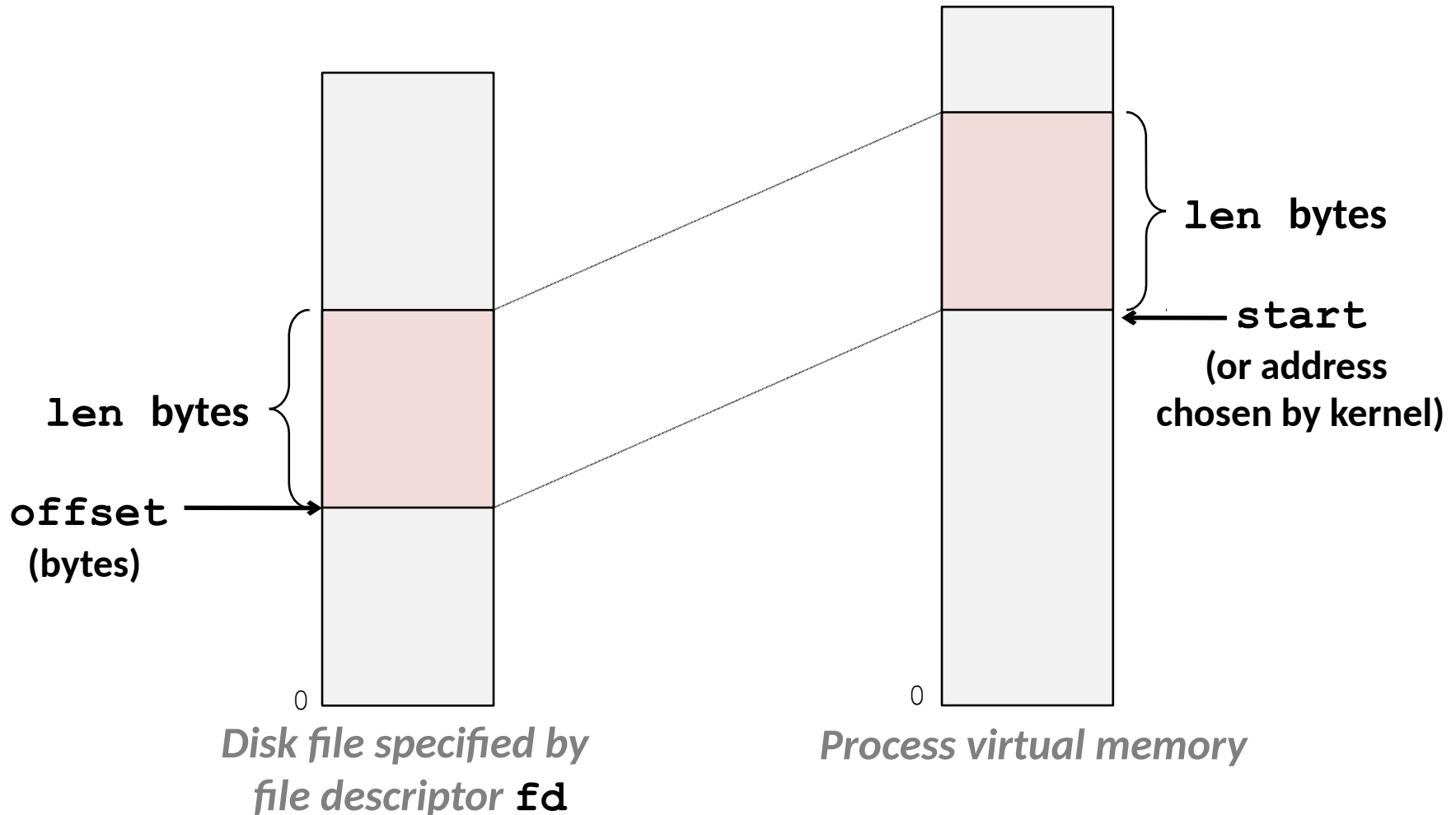
Example: Using `mmap` to Copy Files

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

```c
#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

```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

- **Simple memory system example**
- **Case study: Linux memory system**
- **Memory mapping**
- **Dynamic allocation**
- **Bonus: Implicit 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*.**

Application
Dynamic Memory Allocator
Heap

User stack
Heap (via `malloc`)
Uninitialized data (`.bss`)
Initialized data (`.data`)
Program text (`.text`)

Top of heap (`brk ptr`)

0
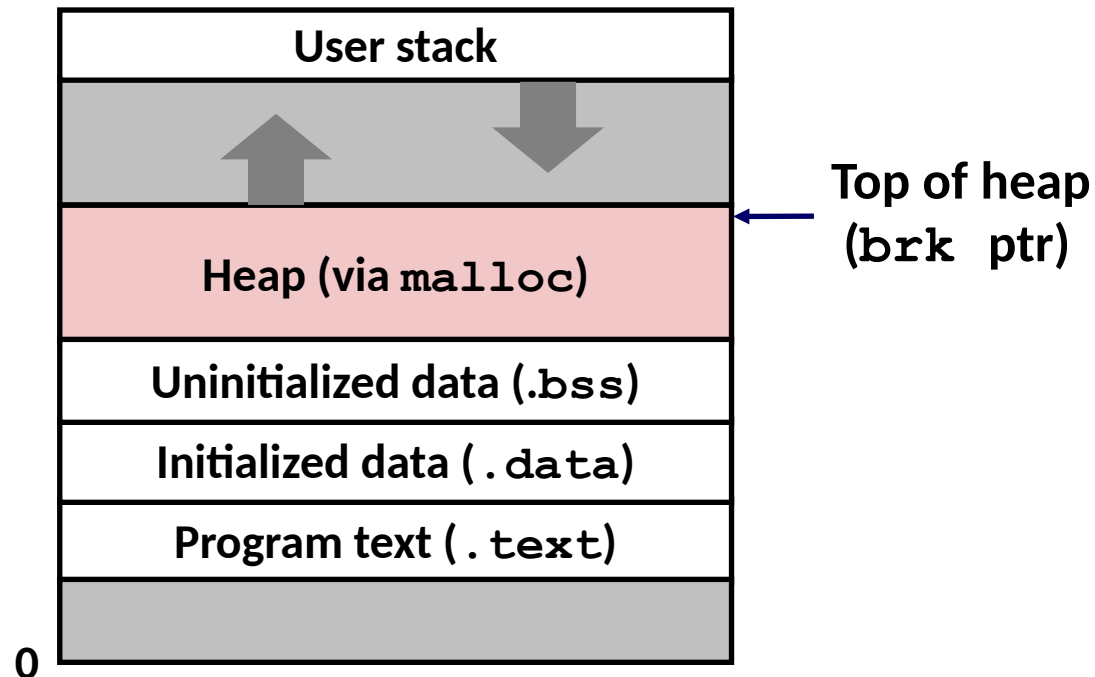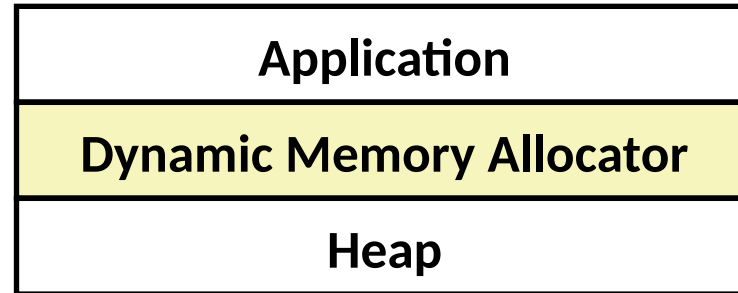
Dynamic Memory Allocation

- **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. garbage collection in Java, ML, and Lisp

- **Will discuss simple explicit memory allocation today**

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

```c
#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);
}
```

Assumptions Made in This Lecture

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

**Allocated block
(4 words)**

**Free block
(3 words)**

☐ **Free word**

🟩 **Allocated word**
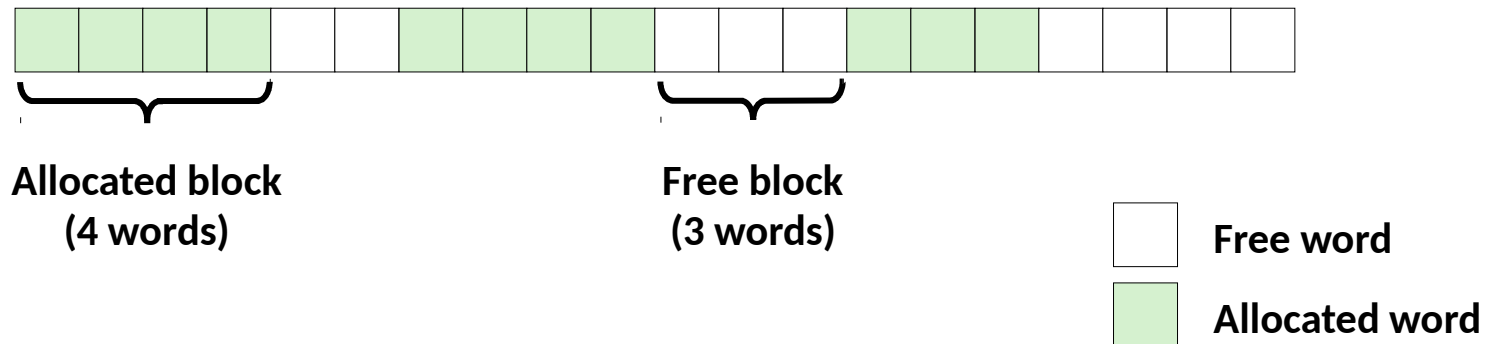
Allocation Example

p1 = malloc(4)
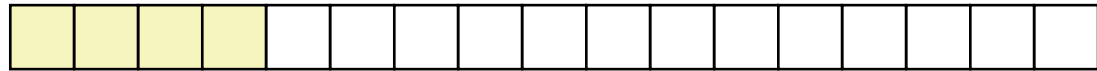
p2 = malloc(5)
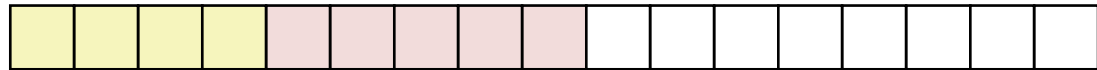
p3 = malloc(6)
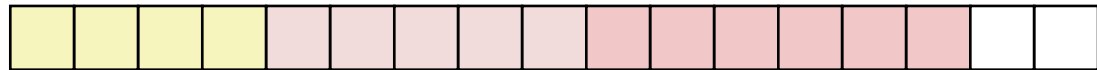
free(p2)
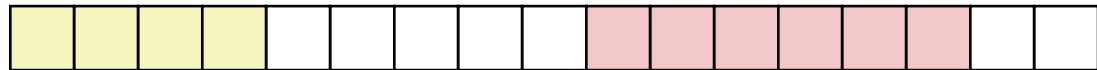
p4 = malloc(2)
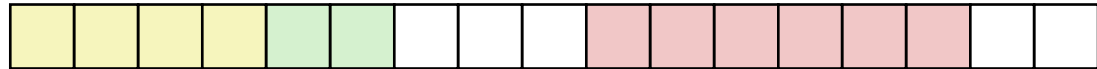
Constraints

- **Applications**
  - Can issue arbitrary sequence of **malloc** and **free** requests
  - **free** request must be to a **malloc**'d block (or NULL)

- **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 (why?)

Performance Goal: Throughput

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

- **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:**
  - $R_0, R_1, ..., R_k, ... , R_{n-1}$

- *Def: Aggregate payload P_k*
  - `malloc(p)` results in a block with a *payload* of `p` bytes
  - After request R_k has 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*
  - $U_k = (max_{i<=k} P_i) / H_k$

Fragmentation

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

Internal Fragmentation

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

Block

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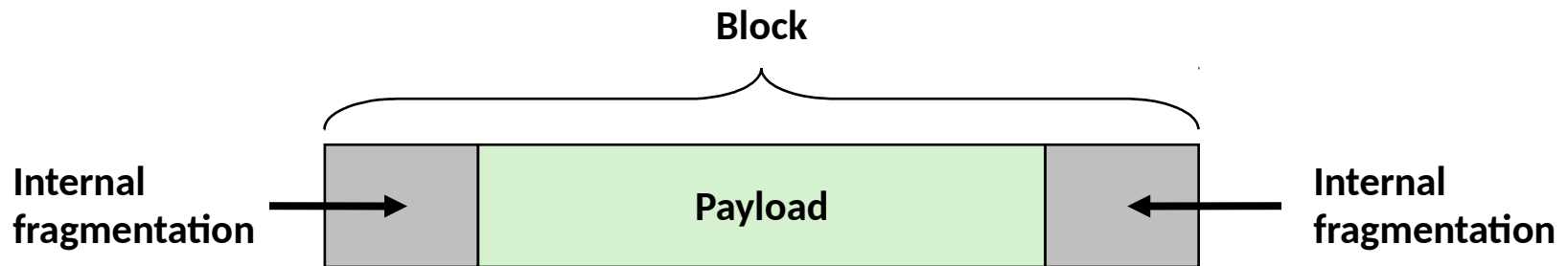
- **Caused by**
  - Overhead of maintaining heap data structures
  - Padding for alignment purposes
  - Explicit policy decisions
    (e.g., to return a big block to satisfy a small request)

- **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)
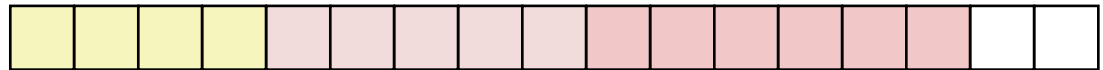
p3 = malloc(6)

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

p0

p0 = malloc(4)   5

block size   payload

free(p0)
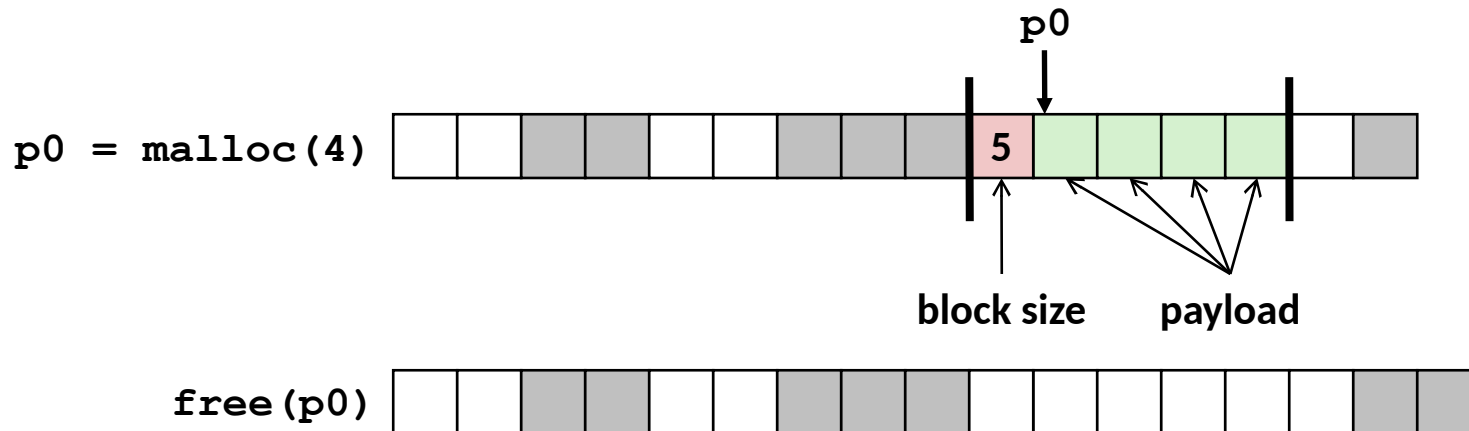
Keeping Track of Free Blocks

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

| 5 | | | | | 4 | | | | 6 | | | | | | 2 | |
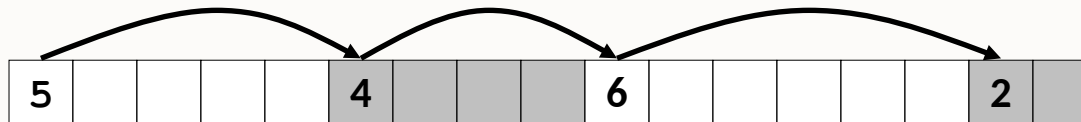
- **Method 2:** *Explicit list* **among the free blocks using pointers**

| 5 | | | | 4 | | | | 6 | | | | | 2 | |
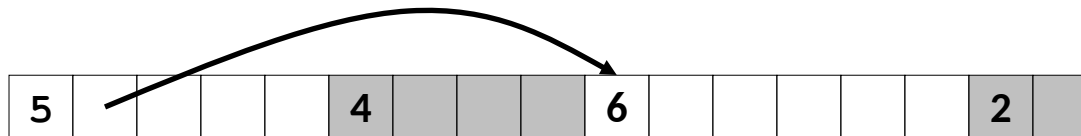
- **Method 3:** *Segregated free list*
  - Different free lists for different size classes

- **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

- **Simple memory system example**
- **Case study: Linux memory system**
- **Memory mapping**
- **Dynamic allocation**
- **Bonus: Implicit 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

1 word

Format of allocated and free blocks

Size	a
Payload	
Optional padding	
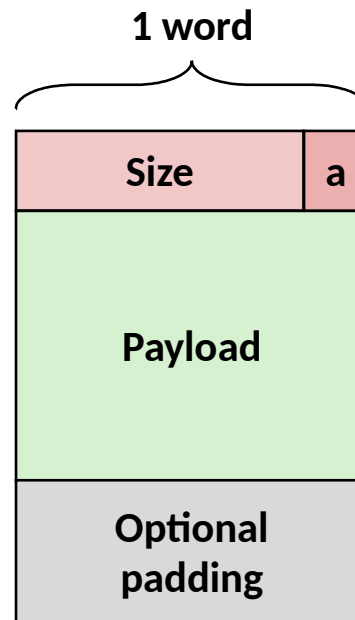
a = 1: Allocated block
a = 0: Free block

Size: block size

Payload: application data
(allocated blocks only)

Detailed Implicit Free List Example

Start of heap

Unused

| 8/0 | | 16/1 | | | | 32/0 | | | | | | | 16/1 | | | | 0/1 |
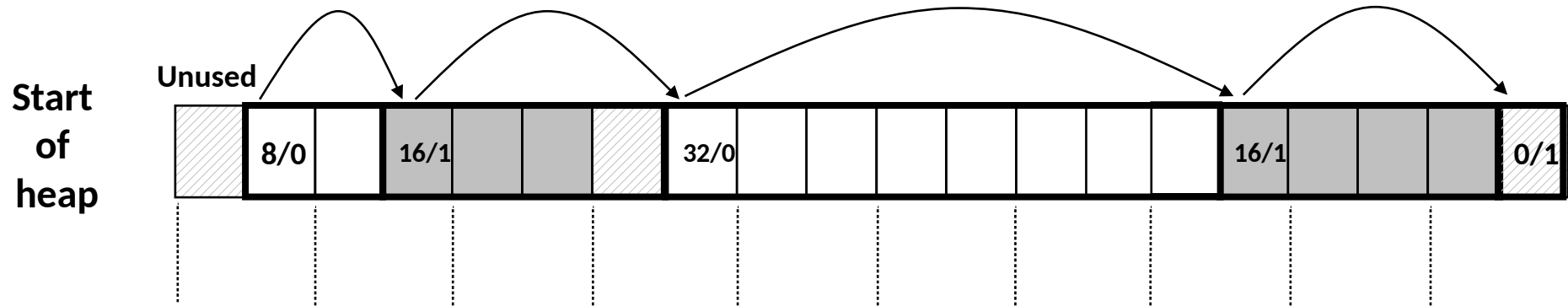
Double-word aligned

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
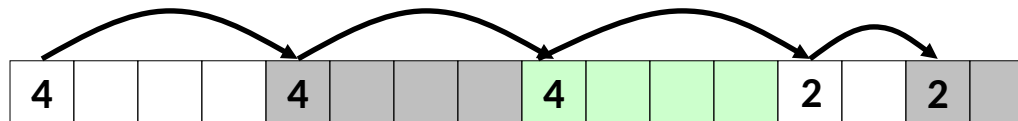 - ■ Will typically run slower than first fit

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
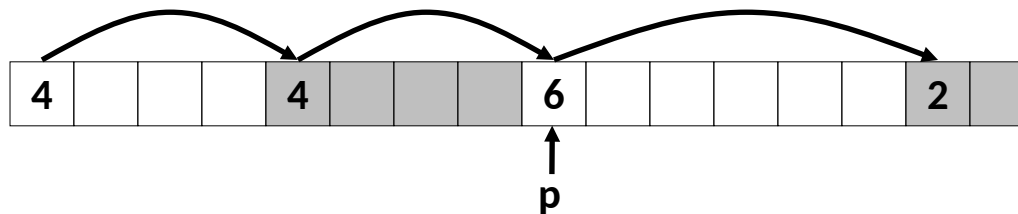
`addblock(p, 4)`


```
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
```
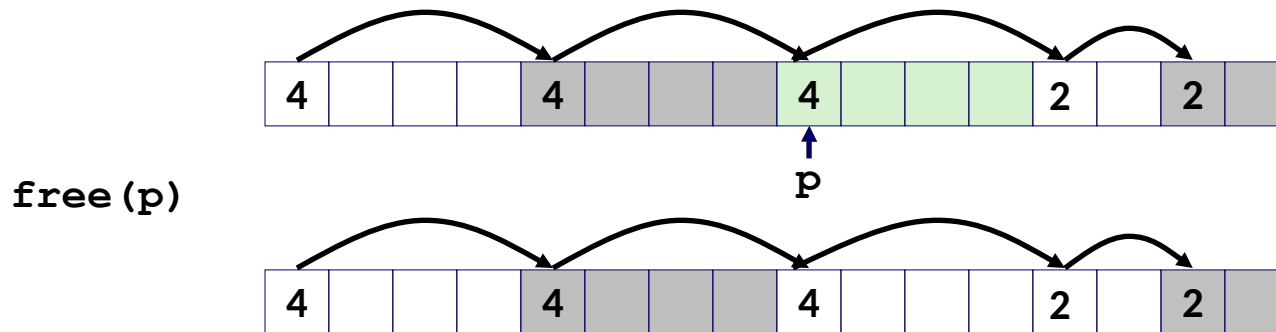
Implicit List: Freeing a Block

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

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

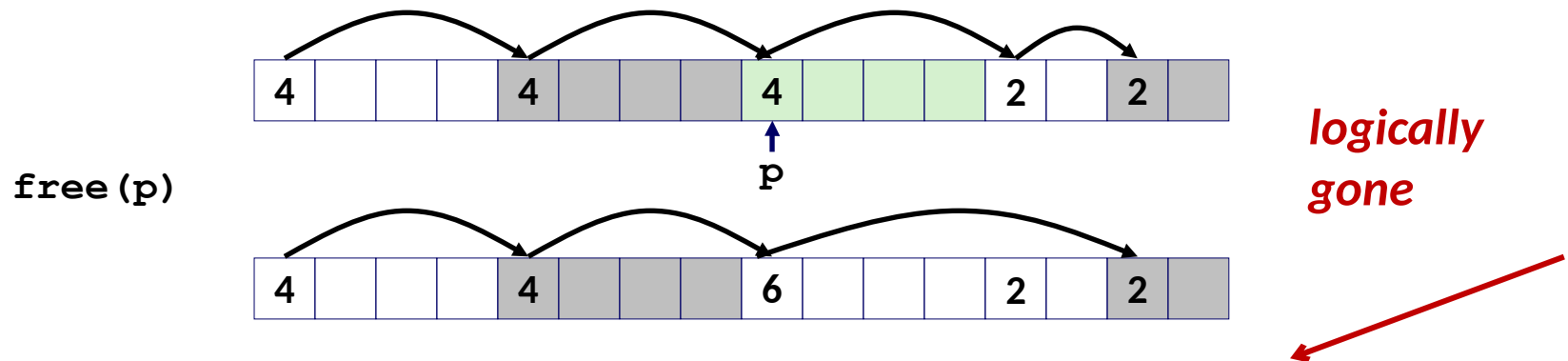 - But can lead to "false fragmentation"

free(p)

malloc(5) *Oops!*

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

free(p)

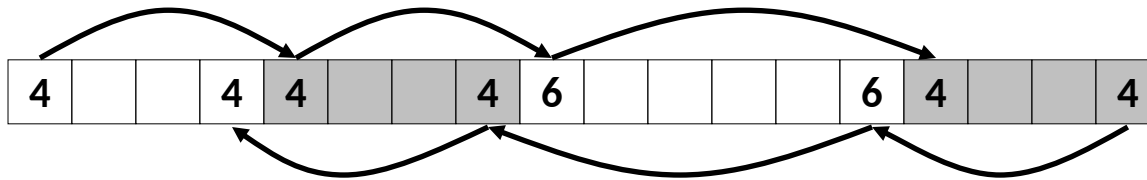p

logically gone

```
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
```
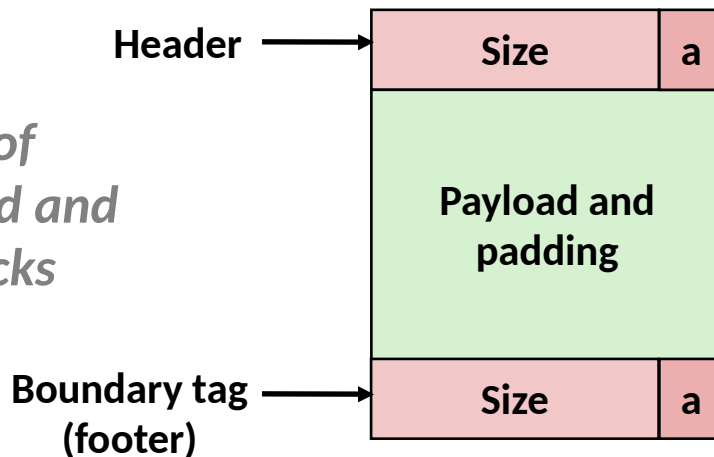
Implicit List: Bidirectional Coalescing

- ***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!

| 4 | | | 4 | 4 | | | 4 | 6 | | | | | 6 | 4 | | | 4 |

Format of allocated and free blocks

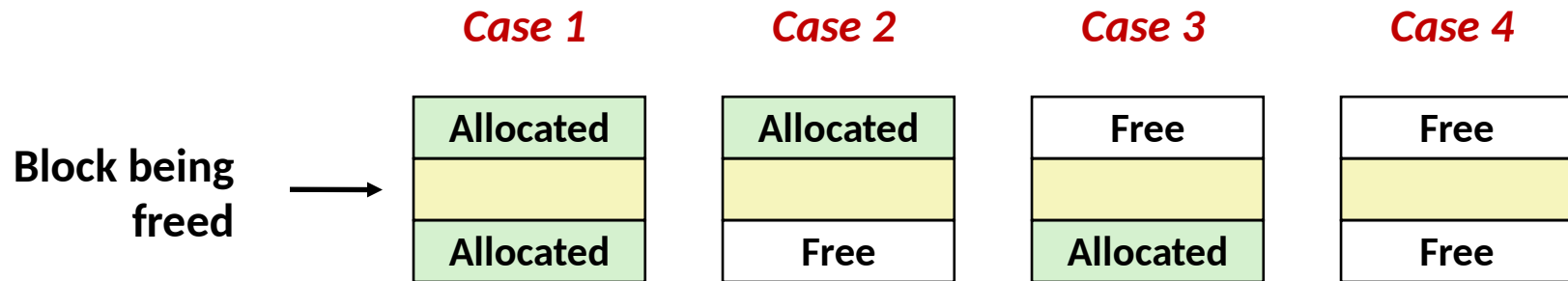| Header →	Size	a
	Payload and padding	
Boundary tag (footer) →	Size	a

a = 1: Allocated block
a = 0: Free block
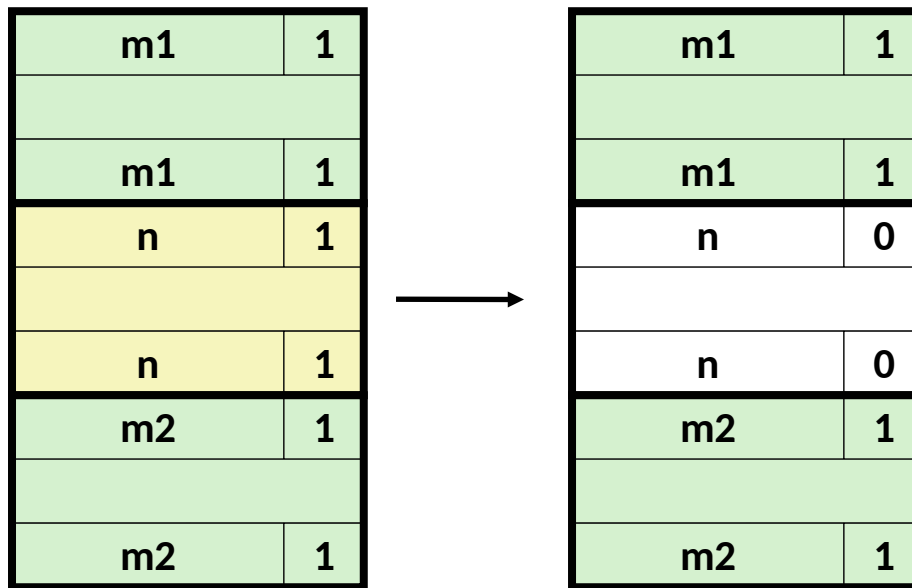
Size: Total block size

Payload: Application data
(allocated blocks only)

Constant Time Coalescing
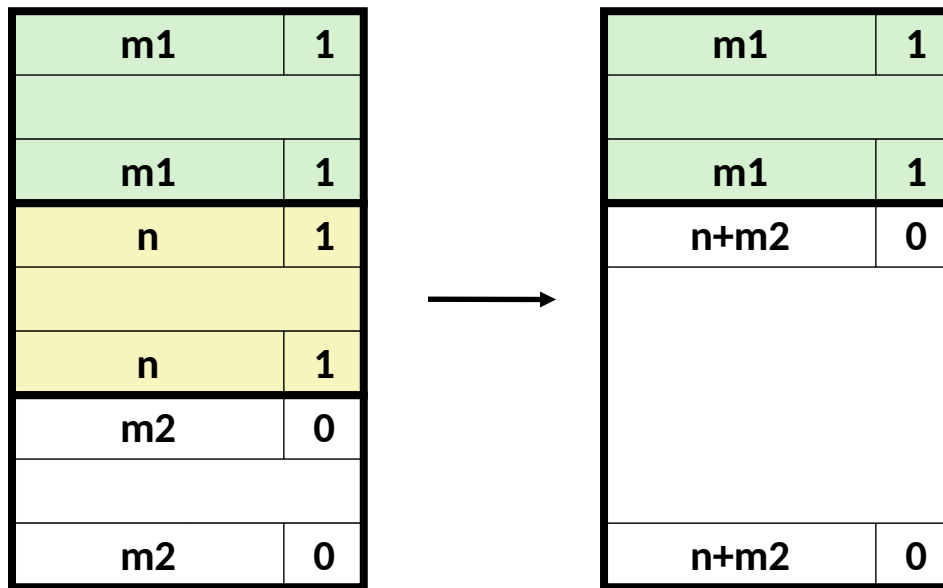
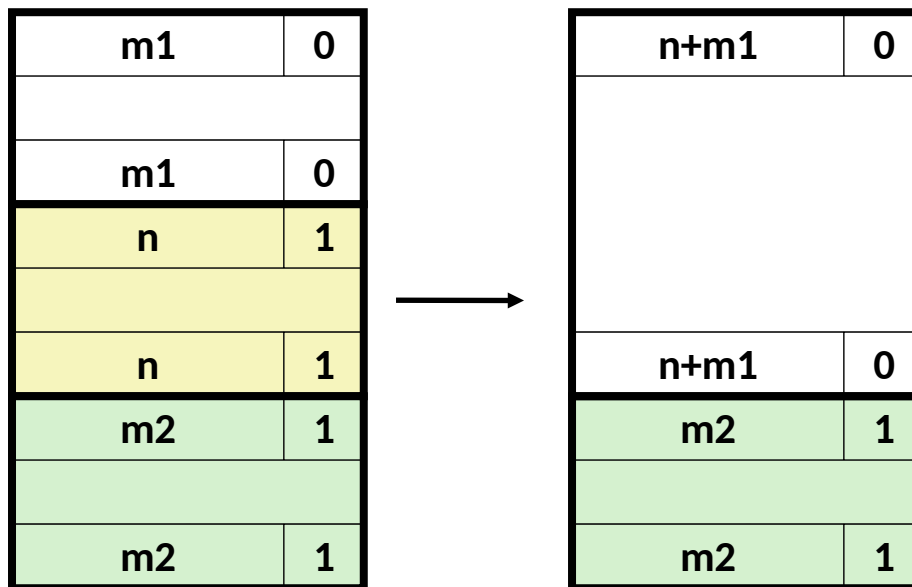|	Case 1	Case 2	Case 3	Case 4

Block being freed →

Case 1	Case 2	Case 3	Case 4
Allocated	Allocated	Free	Free
Allocated	Free	Allocated	Free

Constant Time Coalescing (Case 1)

Constant Time Coalescing (Case 2)

m1	1
m1	1
n	1
n	1
m2	0
m2	0

$\longrightarrow$

m1	1
m1	1
n+m2	0
n+m2	0

Constant Time Coalescing (Case 3)

m1	0
m1	0
n	1
n	1
m2	1
m2	1

→

n+m1	0
n+m1	0
m2	1
m2	1

Constant Time Coalescing (Case 4)
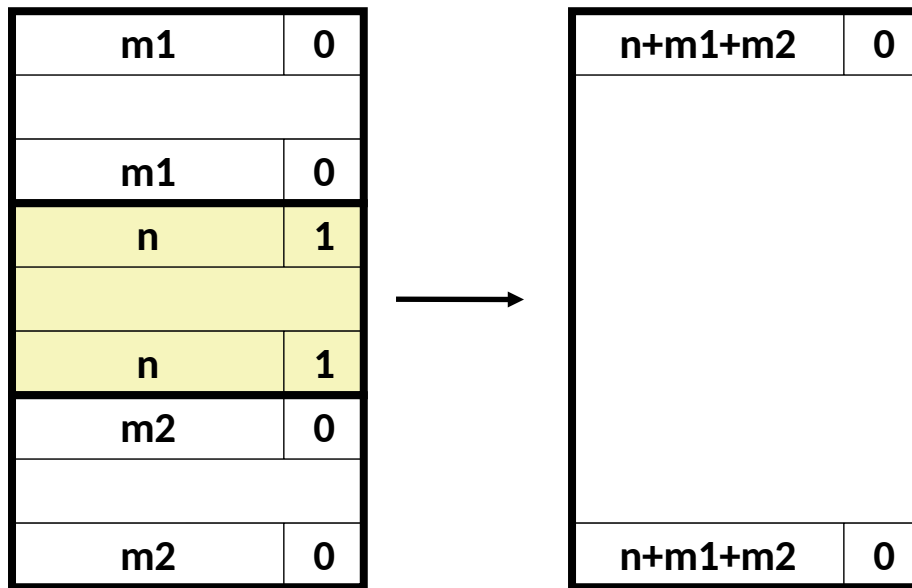
Disadvantages of Boundary Tags

- **Internal fragmentation**

- **Can it be optimized?**
    - Which blocks need the footer tag?
    - What does that mean?

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 (see book) 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**