Lecture 14: Dynamic Memory

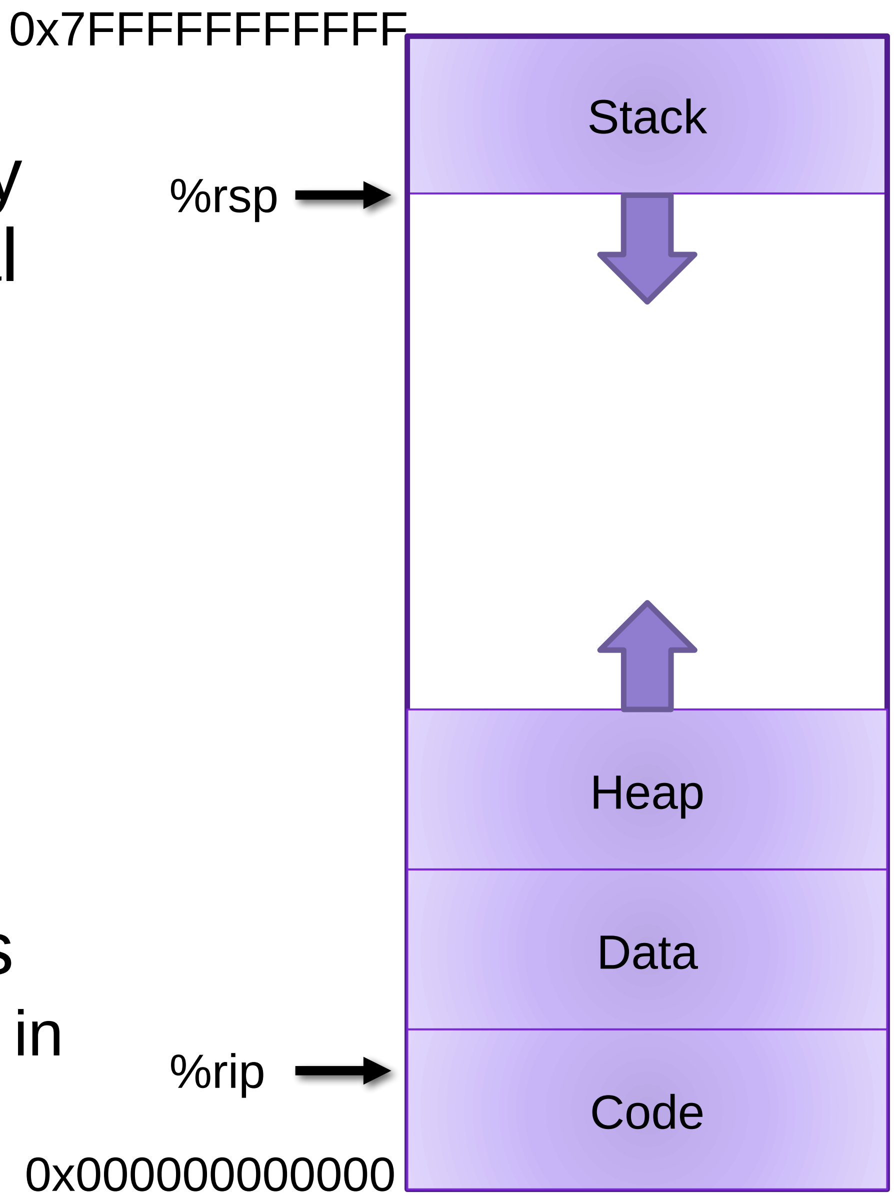
CS 105

Fall 2025

Memory

- byte addressable array made up of four logical segments
- attempt to access uninitialized address results in exception (segfault)

- **stack** provides local storage for procedures
  - "top" of the stack stored in register %rsp



Memory

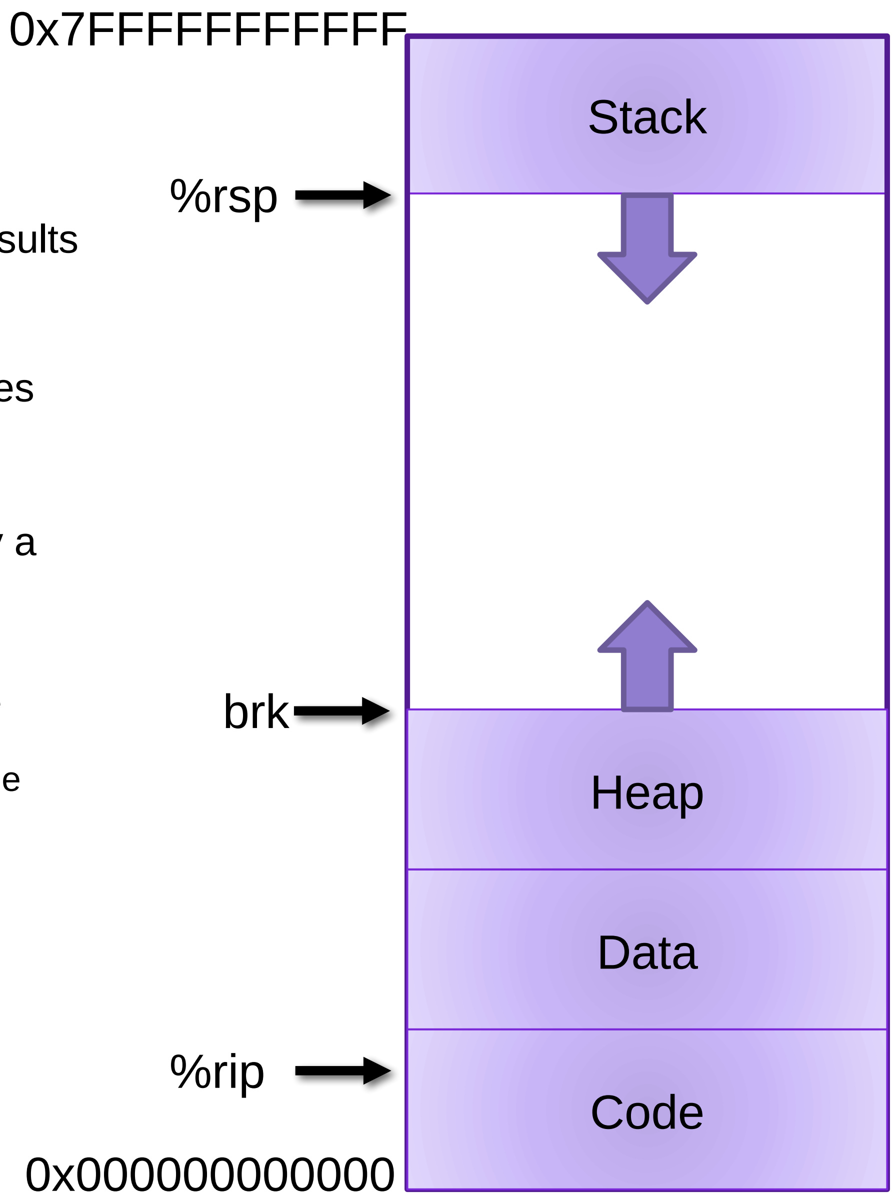
- byte addressable array made up of four logical segments
- attempt to access uninitialized address results in exception (segfault)

- **stack** provides local storage for procedures
  - "top" of the stack stored in register %rsp

- **heap** is an area of memory maintained by a dynamic memory allocator
  - operating system maintains variable brk that points to the top of heap
  - program can dynamically allocate/deallocate heap memory
  - program can use system call sbrk() to change size of heap

- **data** stores global variables

- **code** stores program instructions



Dynamic Memory Allocation

Dynamic memory allocator

- Manages the heap
  - organizes the heap as a collection of (variable-size) **blocks**, each of which is either **allocated** or **free**
  - allocates and deallocates memory
  - may ask OS for additional heap space using system call sbrk()
- Part of the process’s runtime system
  - Linked into program

Dynamic Memory Allocation

Dynamic memory allocator

- Manages the heap
  - organizes the heap as a collection of (variable-size) **blocks**, each of which is either **allocated** or **free**
  - allocates and deallocates memory
  - may ask OS for additional heap space using system call sbrk()
- Part of the process's runtime system
  - Linked into program

Example dynamic memory allocators

- **`malloc`** and **`free`** in C
- **`new`** and **`delete`** in C++
- object creation & garbage collection in Java
- object creation & garbage collection in Python

Dynamic Memory Allocation

Dynamic memory allocator

- Manages the heap
  - organizes the heap as a collection of (variable-size) **blocks**, each of which is either **allocated** or **free**
  - allocates and deallocates memory
  - may ask OS for additional heap space using system call sbrk()
- Part of the process's runtime system
  - Linked into program

Example dynamic memory allocators

- **`malloc`** and **`free`** in C
- **`new`** and **`delete`** in C++

explicit allocators

- object creation & garbage collection in Java
- object creation & garbage collection in Python

Dynamic Memory Allocation

Dynamic memory allocator

- Manages the heap
  - organizes the heap as a collection of (variable-size) **blocks**, each of which is either **allocated** or **free**
  - allocates and deallocates memory
  - may ask OS for additional heap space using system call sbrk()
- Part of the process's runtime system
  - Linked into program

Example dynamic memory allocators

- **`malloc`** and **`free`** in C
- **`new`** and **`delete`** in C++

explicit allocators

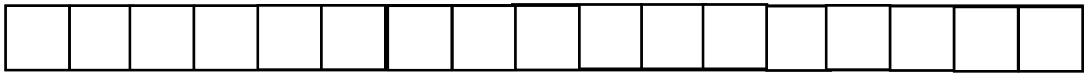
- object creation & garbage collection in Java
- object creation & garbage collection in Python

implicit allocators

Allocation Example using **malloc**

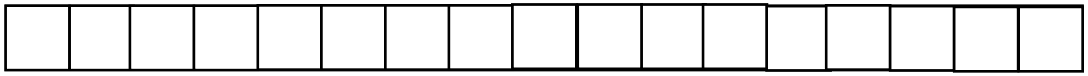
```
#include <stdio.h>
#include <stdlib.h>

void foo(int n) {

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

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

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

Allocation Example



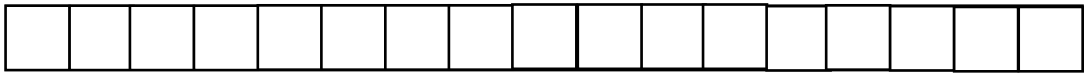
Assume each diagram block depicts 4 bytes

Allocation Example



Assume each diagram block depicts 4 bytes

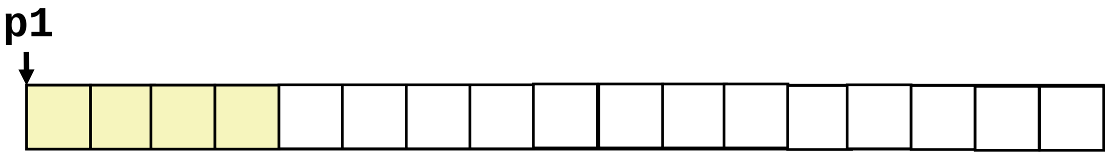
Allocation Example



Assume each diagram block depicts 4 bytes

```
p1 = malloc(16)
```

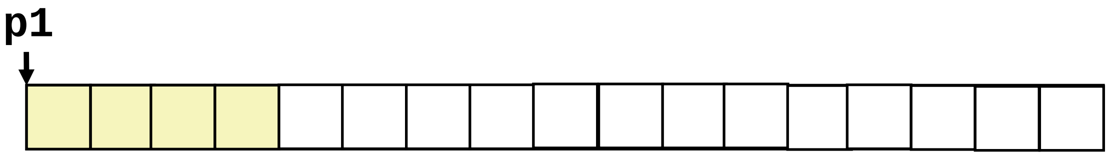
Allocation Example



Assume each diagram block depicts 4 bytes

```
p1 = malloc(16)
```

Allocation Example

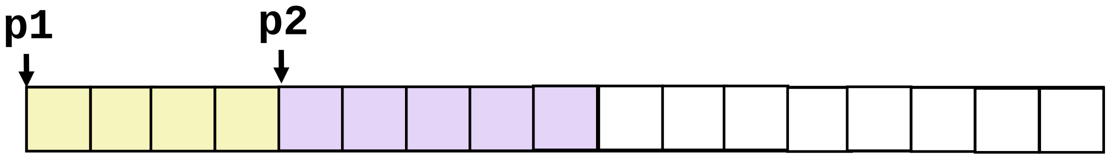


Assume each diagram block depicts 4 bytes

```
p1 = malloc(16)
```

```
p2 = malloc(20)
```

Allocation Example

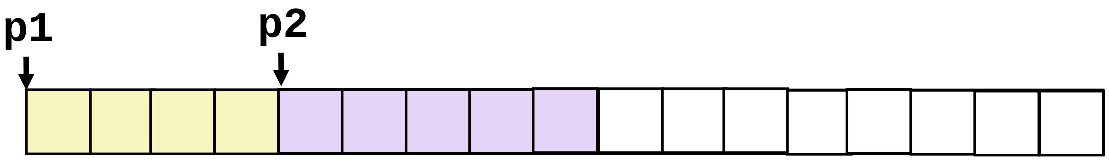


Assume each diagram block depicts 4 bytes

```
p1 = malloc(16)
```

```
p2 = malloc(20)
```

Allocation Example



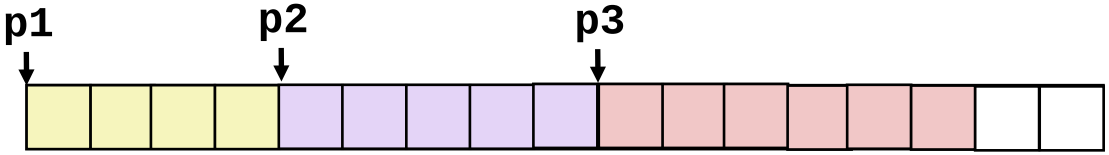
Assume each diagram block depicts 4 bytes

```
p1 = malloc(16)
```

```
p2 = malloc(20)
```

```
p3 = malloc(24)
```

Allocation Example



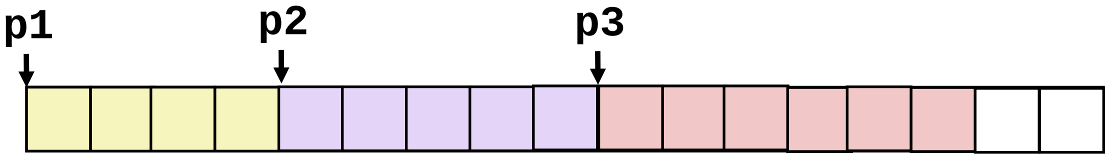
Assume each diagram block depicts 4 bytes

```
p1 = malloc(16)
```

```
p2 = malloc(20)
```

```
p3 = malloc(24)
```

Allocation Example



Assume each diagram block depicts 4 bytes

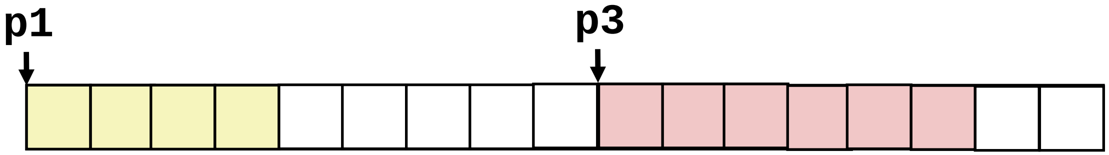
```
p1 = malloc(16)
```

```
p2 = malloc(20)
```

```
p3 = malloc(24)
```

```
free(p2)
```

Allocation Example



Assume each diagram block depicts 4 bytes

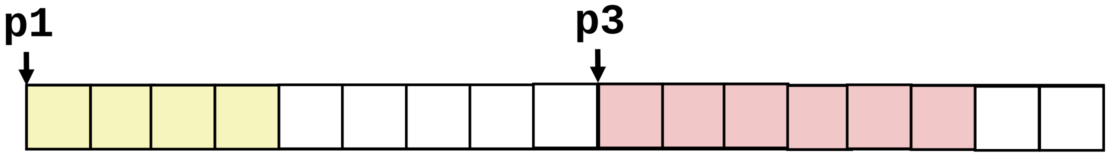
```
p1 = malloc(16)
```

```
p2 = malloc(20)
```

```
p3 = malloc(24)
```

```
free(p2)
```

Allocation Example



Assume each diagram block depicts 4 bytes

```
p1 = malloc(16)
```

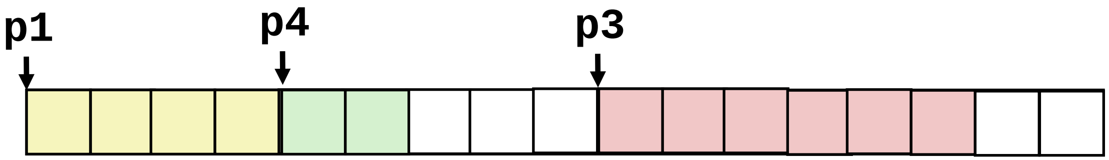
```
p2 = malloc(20)
```

```
p3 = malloc(24)
```

```
free(p2)
```

```
p4 = malloc(8)
```

Allocation Example



Assume each diagram block depicts 4 bytes

```
p1 = malloc(16)
```

```
p2 = malloc(20)
```

```
p3 = malloc(24)
```

```
free(p2)
```

```
p4 = malloc(8)
```

Allocator Requirements

Allocator Requirements

1) **Must handle arbitrary request sequences:**
   - cannot control number, size, or order of requests
   - (but we'll assume that each free request corresponds to an allocated block)

Allocator Requirements

1) **Must handle arbitrary request sequences:**
   - cannot control number, size, or order of requests
   - (but we'll assume that each free request corresponds to an allocated block)

2) **Must respond immediately:**
   - no reordering or buffering requests

Allocator Requirements

1) **Must handle arbitrary request sequences:**
   - cannot control number, size, or order of requests
   - (but we'll assume that each free request corresponds to an allocated block)

2) **Must respond immediately:**
   - no reordering or buffering requests

3) **Must not modify allocated blocks:**
   - can only allocate from free memory on the heap
   - cannot modify or move blocks once they are allocated

Allocator Requirements

1) **Must handle arbitrary request sequences:**
   - cannot control number, size, or order of requests
   - (but we'll assume that each free request corresponds to an allocated block)

2) **Must respond immediately:**
   - no reordering or buffering requests

3) **Must not modify allocated blocks:**
   - can only allocate from free memory on the heap
   - cannot modify or move blocks once they are allocated

4) **Must align blocks:**
   - 8-byte (x86) or 16-byte (x86-64) alignment on Linux
   - Ensures that allocated blocks can hold any type of data

Allocator Requirements

1) **Must handle arbitrary request sequences:**
   - cannot control number, size, or order of requests
   - (but we'll assume that each free request corresponds to an allocated block)

2) **Must respond immediately:**
   - no reordering or buffering requests

3) **Must not modify allocated blocks:**
   - can only allocate from free memory on the heap
   - cannot modify or move blocks once they are allocated

4) **Must align blocks:**
   - 8-byte (x86) or 16-byte (x86-64) alignment on Linux
   - Ensures that allocated blocks can hold any type of data

5) **Must only use the heap:**
   - any data structures used by the allocator must be stored in the heap

First Example: A Simple Allocator

```
void* malloc (size_t size) {
  return sbrk(align(size));
}

void free (void* ptr) {
  // do nothing
}
```

First Example: A Simple Allocator

```
void* malloc (size_t size) {
  return sbrk(align(size));
}

void free (void* ptr) {
  // do nothing
}
```

Advantages

- Simple
- Blazing fast

Disadvantages

- Memory is never recycled
- Wastes a lot of space

Allocator Goals

-

- **Throughput:** number of requests completed per time unit
  - Make allocator efficient
  - Example: if your allocator processes 5,000 `malloc` calls and 5,000 `free` calls in 10 seconds then throughput is 1,000 operations/second

- **Memory Utilization:** fraction of heap memory allocated
  - Minimize wasted space
  - Peak Memory Utilization $U_t = \frac{\max_{i \leq t} space\ allocated\ at\ time\ i}{size\ of\ heap\ at\ time\ t}$

- These goals are often conflicting

Exercise: Memory Utilization

- Recall that Peak Memory Utilization $U_t = \frac{\max_{i \le t} space\ allocated\ at\ time\ i}{size\ of\ heap\ at\ time\ t}$

- What is the Peak Memory Utilization at time $t = 2$?
- What is the Peak Memory Utilization at time $t = 5$?

Exercise: Memory Utilization

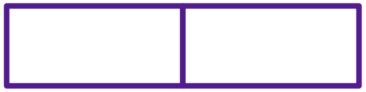
- Recall that Peak Memory Utilization $U_t = \frac{\max_{i \le t} space\ allocated\ at\ time\ i}{size\ of\ heap\ at\ time\ t}$

- What is the Peak Memory Utilization at time $t = 2$?
- What is the Peak Memory Utilization at time $t = 5$?

Exercise: Memory Utilization

- Recall that Peak Memory Utilization $U_t = \frac{\max_{i \leq t} \mathit{space\ allocated\ at\ time\ i}}{\mathit{size\ of\ heap\ at\ time\ t}}$

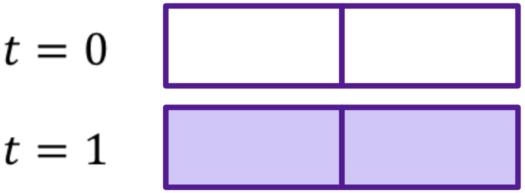
$t = 0$



- What is the Peak Memory Utilization at time $t = 2$?
- What is the Peak Memory Utilization at time $t = 5$?

Exercise: Memory Utilization

- Recall that Peak Memory Utilization $U_t = \frac{\max_{i \le t} space\ allocated\ at\ time\ i}{size\ of\ heap\ at\ time\ t}$



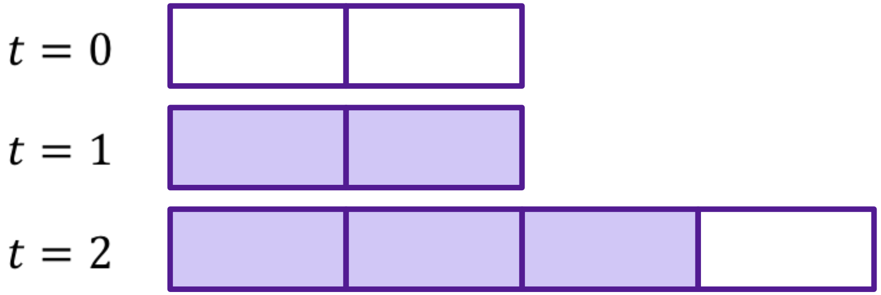
$t = 0$

$t = 1$

- What is the Peak Memory Utilization at time $t = 2$?
- What is the Peak Memory Utilization at time $t = 5$?

Exercise: Memory Utilization

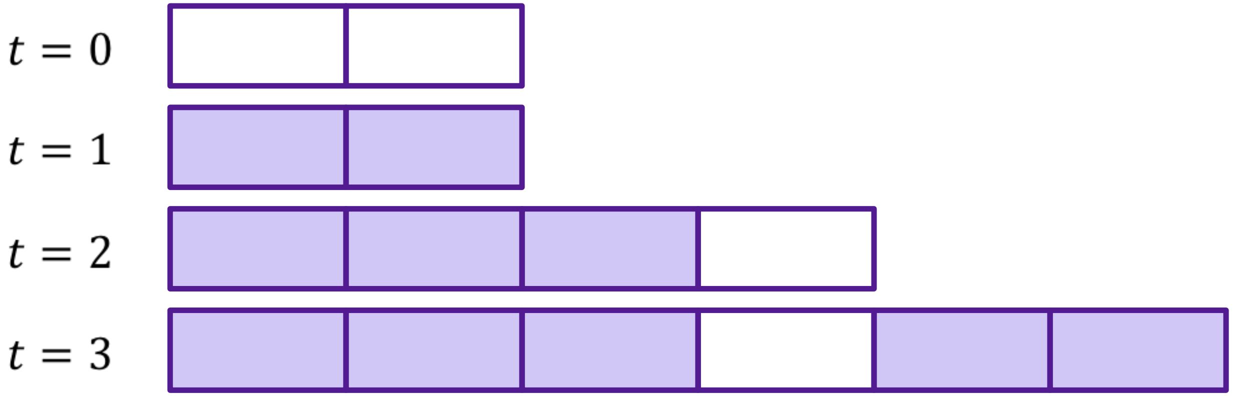
- Recall that Peak Memory Utilization $U_t = \frac{\max_{i \le t} \mathit{space\ allocated\ at\ time\ i}}{\mathit{size\ of\ heap\ at\ time\ t}}$



- What is the Peak Memory Utilization at time $t = 2$?
- What is the Peak Memory Utilization at time $t = 5$?

Exercise: Memory Utilization

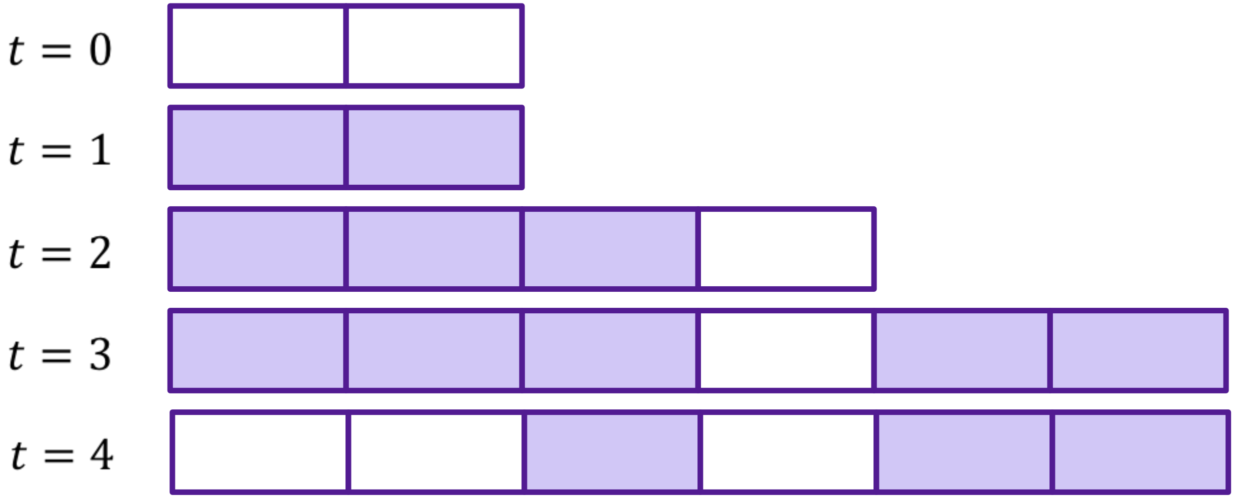
- Recall that Peak Memory Utilization $U_t = \frac{\max_{i \le t} \mathit{space\ allocated\ at\ time\ i}}{\mathit{size\ of\ heap\ at\ time\ t}}$



- What is the Peak Memory Utilization at time $t = 2$?
- What is the Peak Memory Utilization at time $t = 5$?

Exercise: Memory Utilization

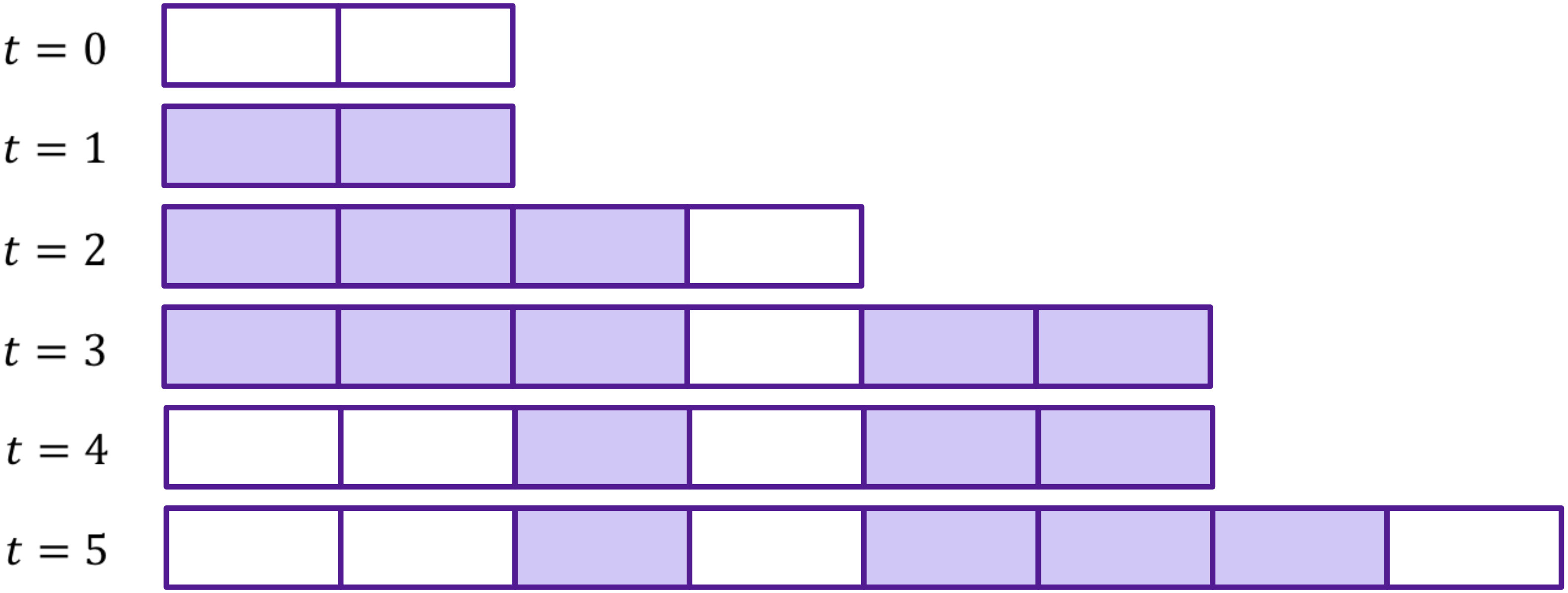
- Recall that Peak Memory Utilization $U_t = \frac{\max_{i \le t} \mathit{space\ allocated\ at\ time\ i}}{\mathit{size\ of\ heap\ at\ time\ t}}$



- What is the Peak Memory Utilization at time $t = 2$?
- What is the Peak Memory Utilization at time $t = 5$?

Exercise: Memory Utilization

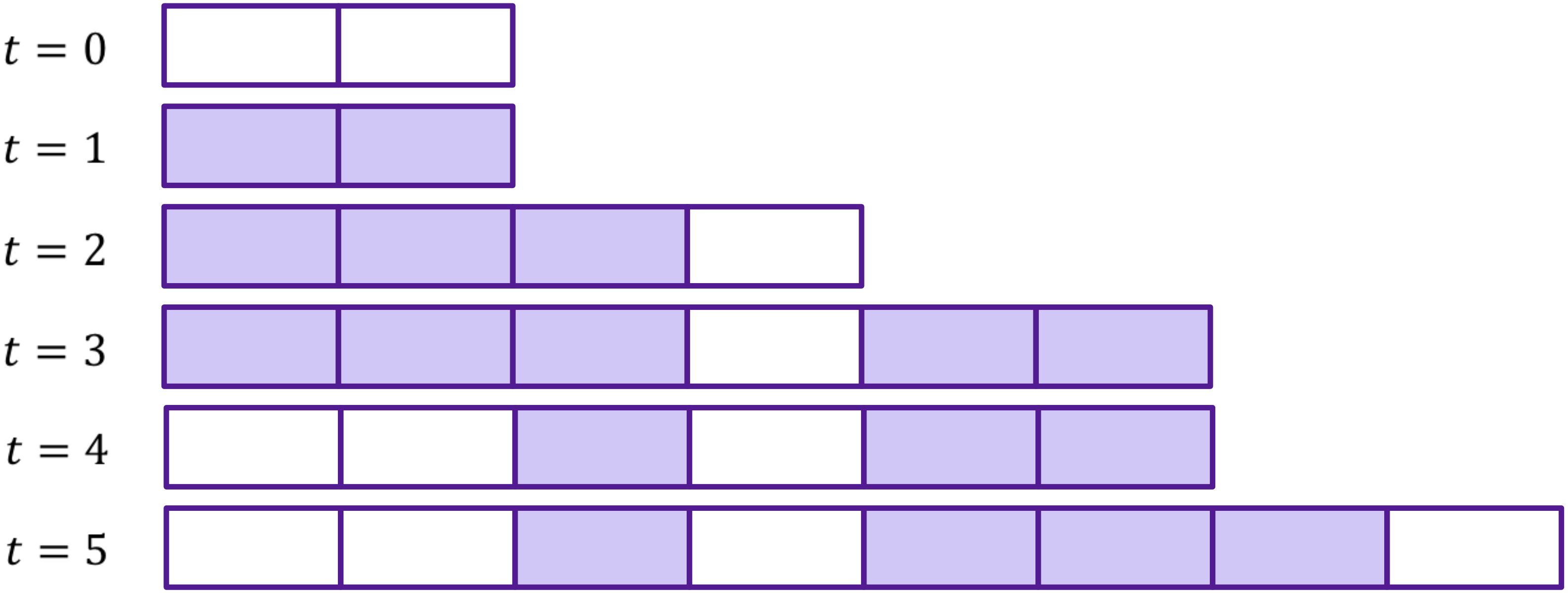
- Recall that Peak Memory Utilization $U_t = \frac{\max_{i \le t} \mathit{space\ allocated\ at\ time\ i}}{\mathit{size\ of\ heap\ at\ time\ t}}$



- What is the Peak Memory Utilization at time $t = 2$?
- What is the Peak Memory Utilization at time $t = 5$?

Exercise: Memory Utilization

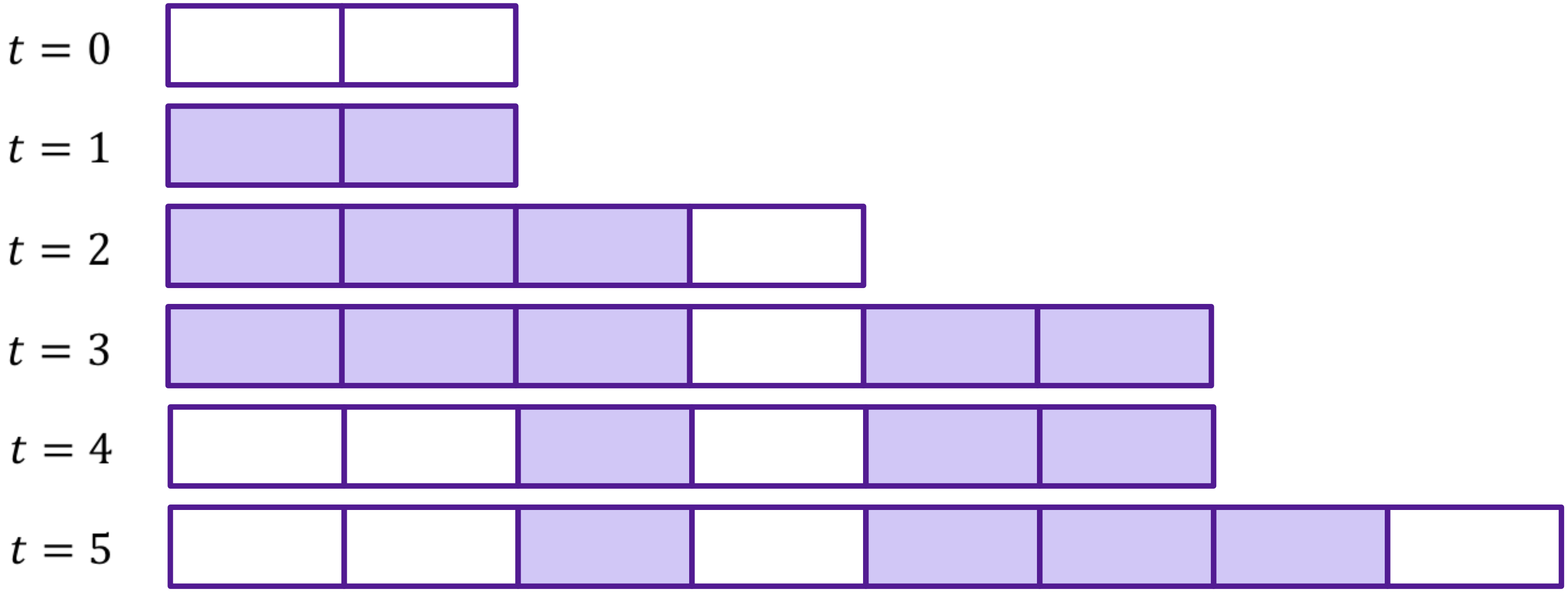
- Recall that Peak Memory Utilization $U_t = \dfrac{\max\limits_{i \le t} space\ allocated\ at\ time\ i}{size\ of\ heap\ at\ time\ t}$



- What is the Peak Memory Utilization at time $t = 2$?
- What is the Peak Memory Utilization at time $t = 5$?

Exercise: Memory Utilization

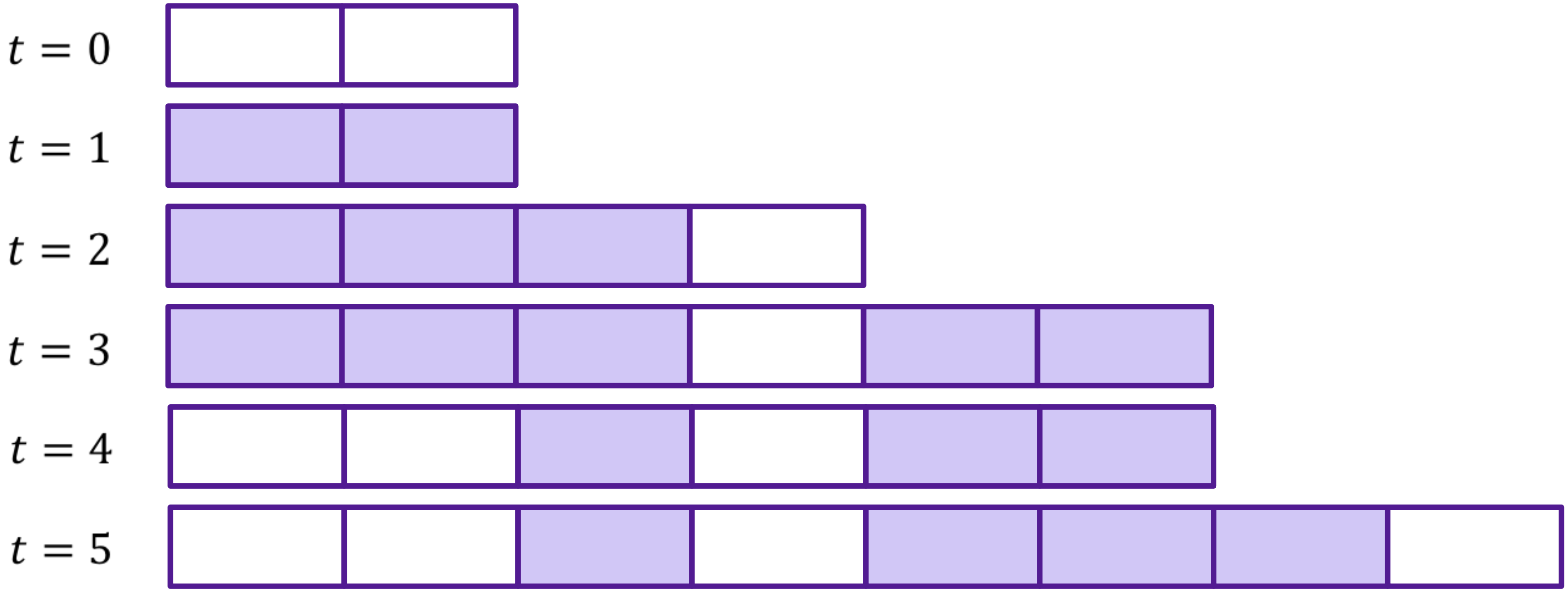
- Recall that Peak Memory Utilization $U_t = \frac{\max_{i \le t} space\ allocated\ at\ time\ i}{size\ of\ heap\ at\ time\ t}$



- What is the Peak Memory Utilization at time $t = 2$? $\mathbf{3/4}$
- What is the Peak Memory Utilization at time $t = 5$?

Exercise: Memory Utilization

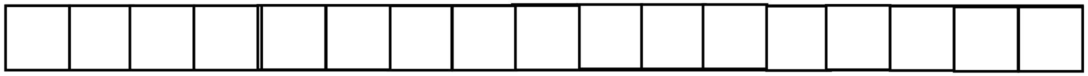
- Recall that Peak Memory Utilization $U_t = \frac{\max_{i \leq t} \textit{space allocated at time } i}{\textit{size of heap at time } t}$



- What is the Peak Memory Utilization at time $t = 2$? **$3/4$**
- What is the Peak Memory Utilization at time $t = 5$? **$5/8$**

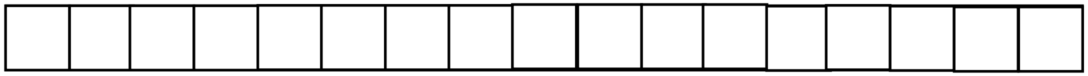
Utilization Blocker: External Fragmentation

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



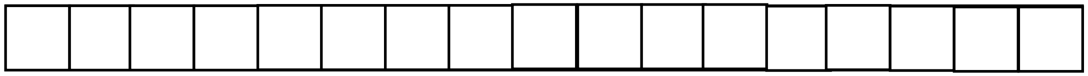
Utilization Blocker: External Fragmentation

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



Utilization Blocker: External Fragmentation

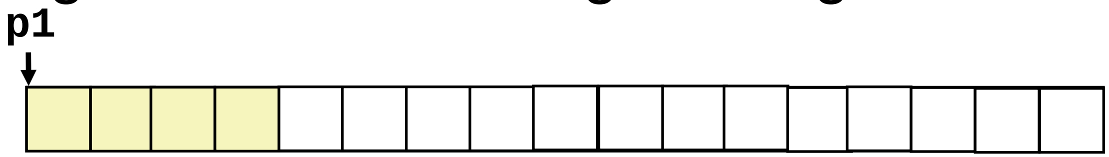
- Occurs when there is enough aggregate heap memory, but no single free block is large enough



```
p1 = malloc(16)
```

Utilization Blocker: External Fragmentation

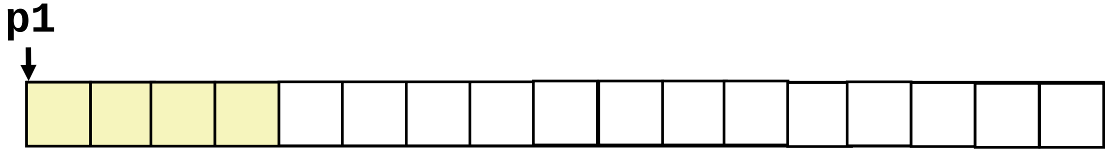
- Occurs when there is enough aggregate heap memory, but no single free block is large enough



`p1 = malloc(16)`

Utilization Blocker: External Fragmentation

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

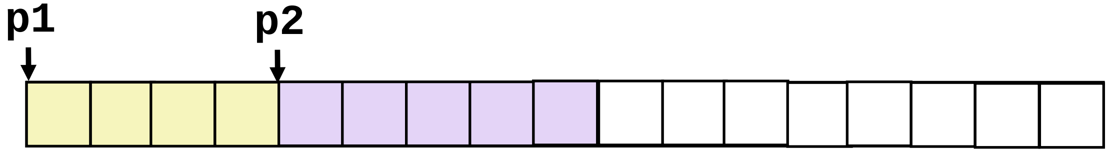


`p1 = malloc(16)`

`p2 = malloc(20)`

Utilization Blocker: External Fragmentation

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

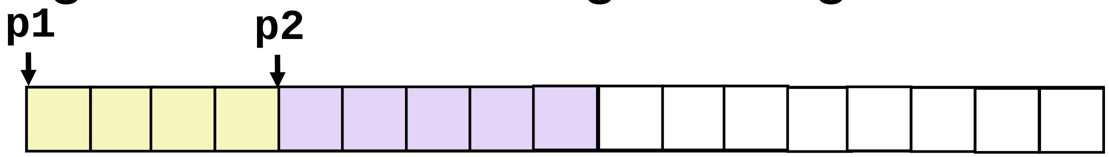


```
p1 = malloc(16)
```

```
p2 = malloc(20)
```

Utilization Blocker: External Fragmentation

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



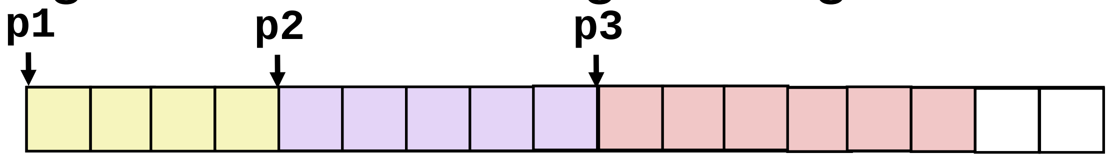
```
p1 = malloc(16)
```

```
p2 = malloc(20)
```

```
p3 = malloc(24)
```

Utilization Blocker: External Fragmentation

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



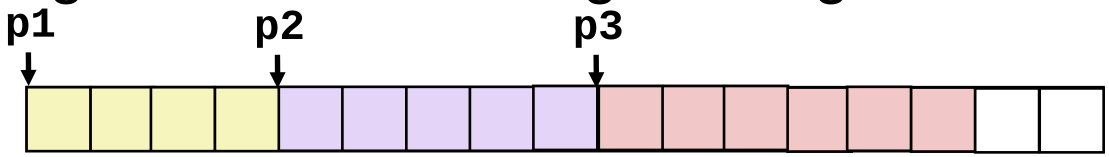
```
p1 = malloc(16)
```

```
p2 = malloc(20)
```

```
p3 = malloc(24)
```

Utilization Blocker: External Fragmentation

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



```
p1 = malloc(16)
```

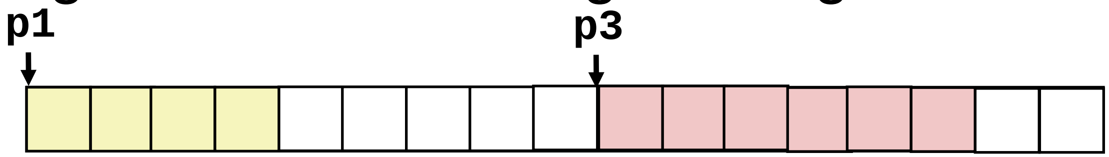
```
p2 = malloc(20)
```

```
p3 = malloc(24)
```

```
free(p2)
```

Utilization Blocker: External Fragmentation

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



```
p1 = malloc(16)
```

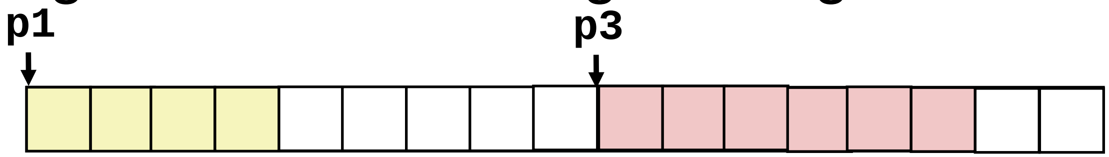
```
p2 = malloc(20)
```

```
p3 = malloc(24)
```

```
free(p2)
```

Utilization Blocker: External Fragmentation

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



```
p1 = malloc(16)
```

```
p2 = malloc(20)
```

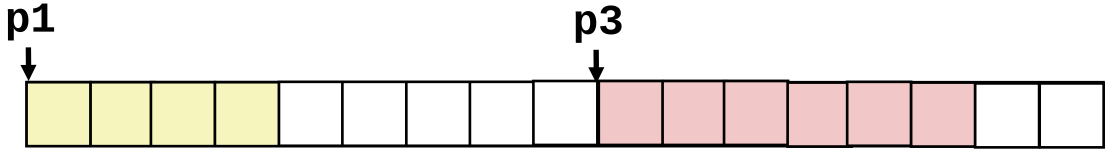
```
p3 = malloc(24)
```

```
free(p2)
```

```
p4 = malloc(24)
```

Utilization Blocker: External Fragmentation

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



```
p1 = malloc(16)
```

```
p2 = malloc(20)
```

```
p3 = malloc(24)
```

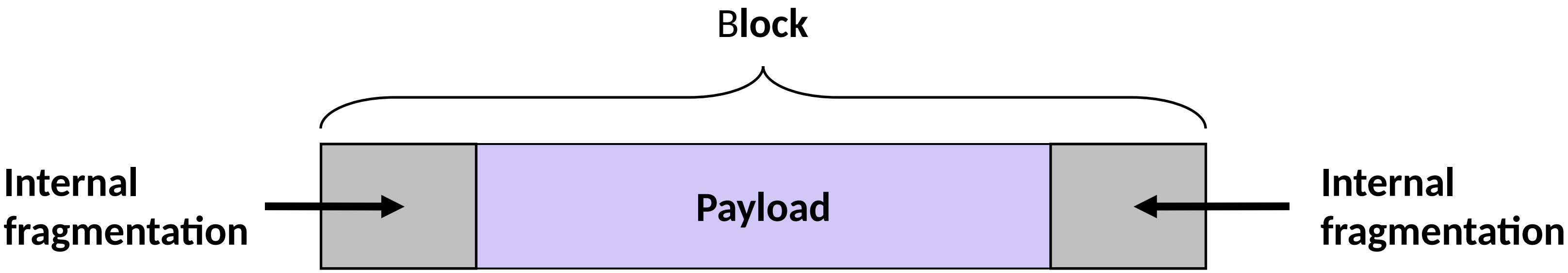
```
free(p2)
```

```
p4 = malloc(24)
```

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

Utilization Blocker: Internal Fragmentation

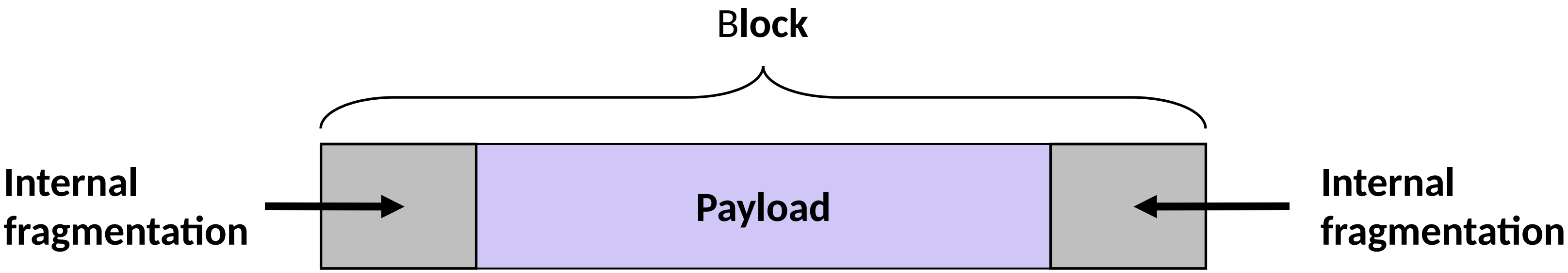
- For a given block, ***internal fragmentation*** occurs if payload is smaller than block size



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

Utilization Blocker: 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
    (for example, returning a big block to satisfy a small request)
- Depends only on the pattern of previous requests
  - Thus, easy to measure

Challenges

- Goal: maximize throughput and peak memory utilization

- Implementation challenges:

Challenges

- Goal: maximize throughput and peak memory utilization

- Implementation challenges:
  - How do we know how much memory to free given just a pointer?

Challenges

- Goal: maximize throughput and peak memory utilization

- Implementation challenges:
  - How do we know how much memory to free given just a pointer?
  - How do we keep track of the free blocks?

Challenges

- Goal: maximize throughput and peak memory utilization

- Implementation challenges:
  - How do we know how much memory to free given just a pointer?
  - How do we keep track of the free blocks?
  - How do we pick a block to use for allocation?

Challenges

- Goal: maximize throughput and peak memory utilization

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

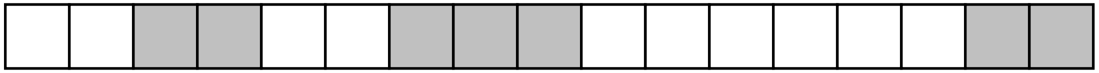
Challenges

- Goal: maximize throughput and peak memory utilization

- Implementation challenges:
  - How do we know how much memory to free given just a pointer?
  - How do we keep track of the free blocks?
  - How do we pick a block to use for allocation?
  - 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 reinsert a freed block?

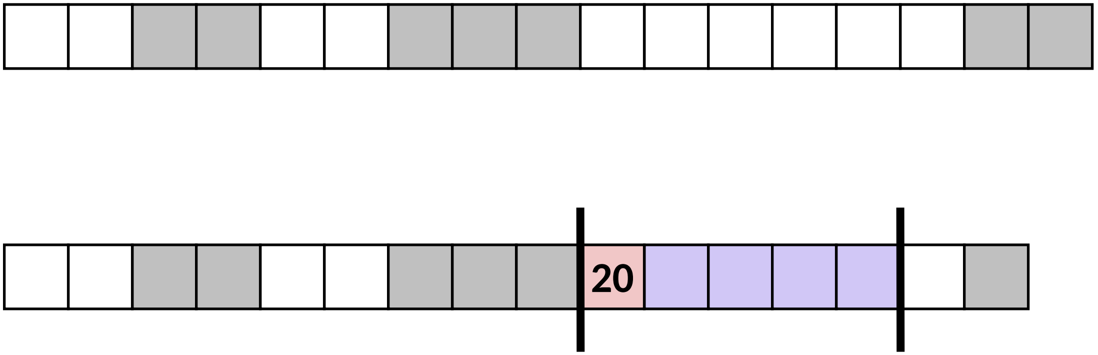
Knowing How Much to Free

- 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 (4 byte) header for every allocated block



Knowing How Much to Free

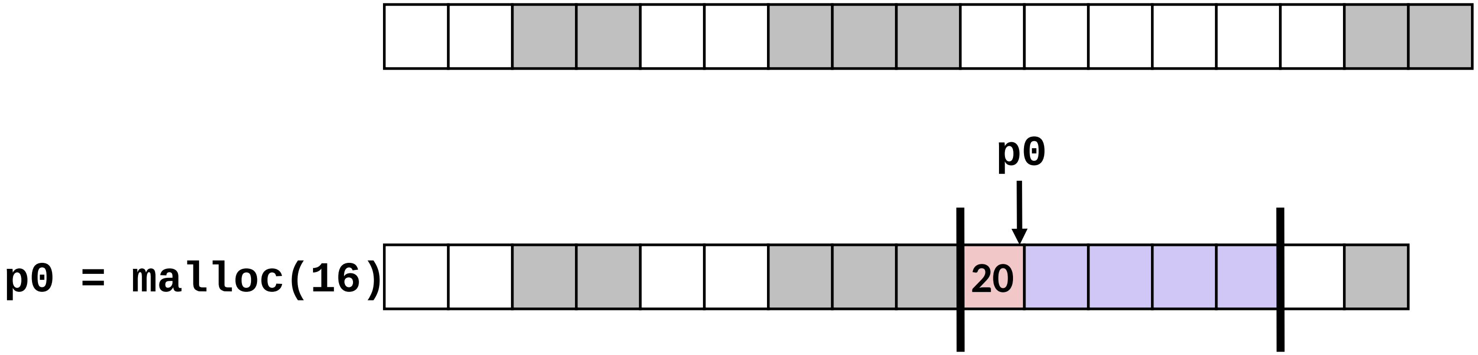
- Standard method
  - Keep the length of a block in the word preceding the block.
    - This word is often called the ***header field*** or ***header***
  - Requires an extra (4 byte) header for every allocated block



`p0 = malloc(16)`

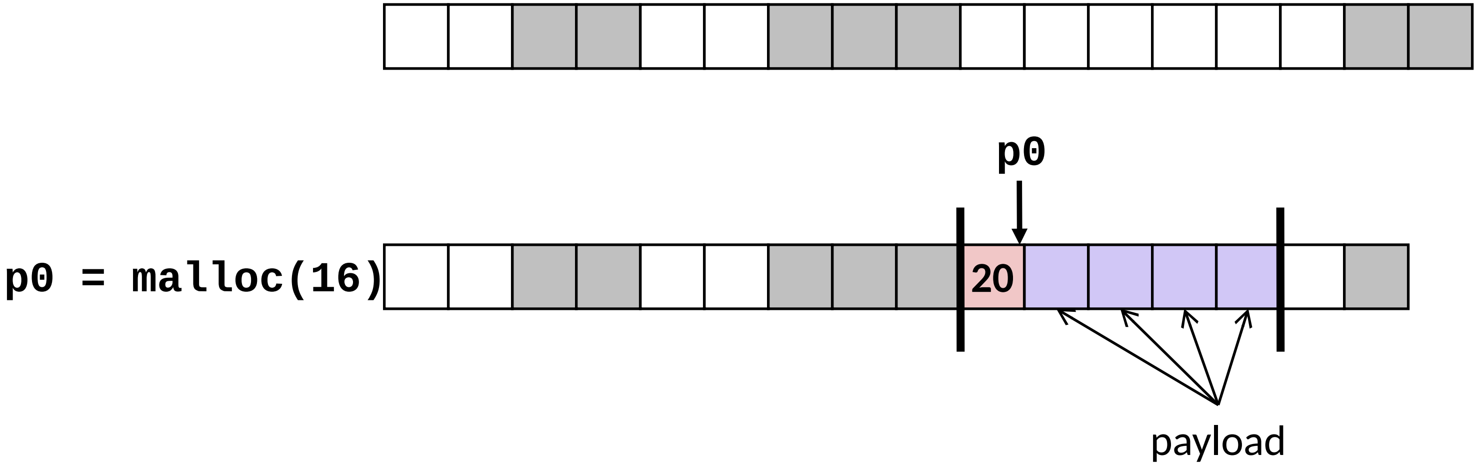
Knowing How Much to Free

- 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 (4 byte) header for every allocated block



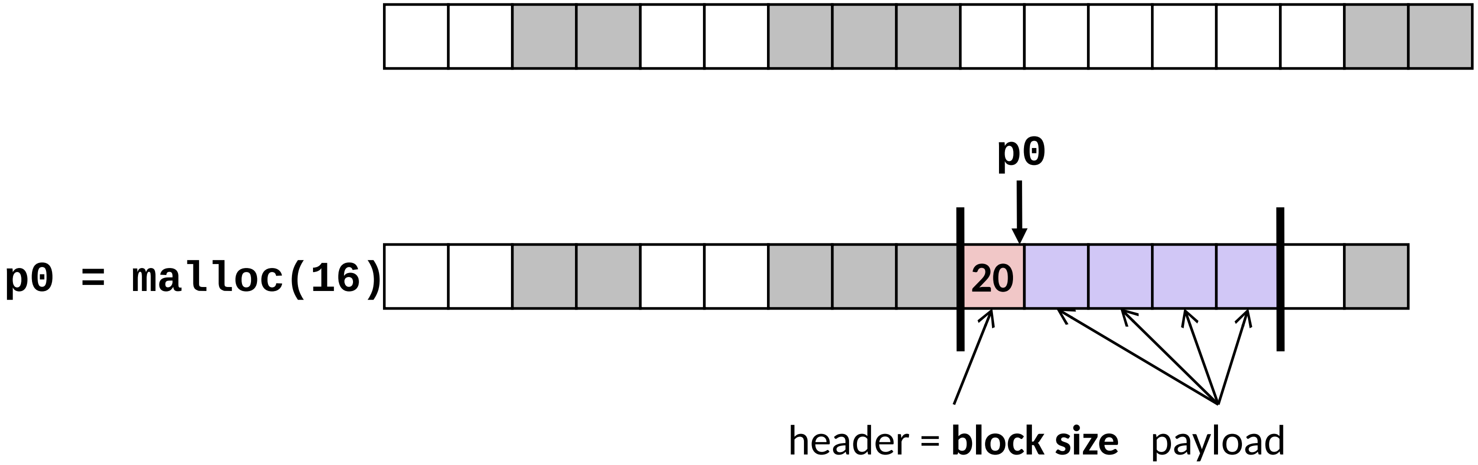
Knowing How Much to Free

- 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 (4 byte) header for every allocated block



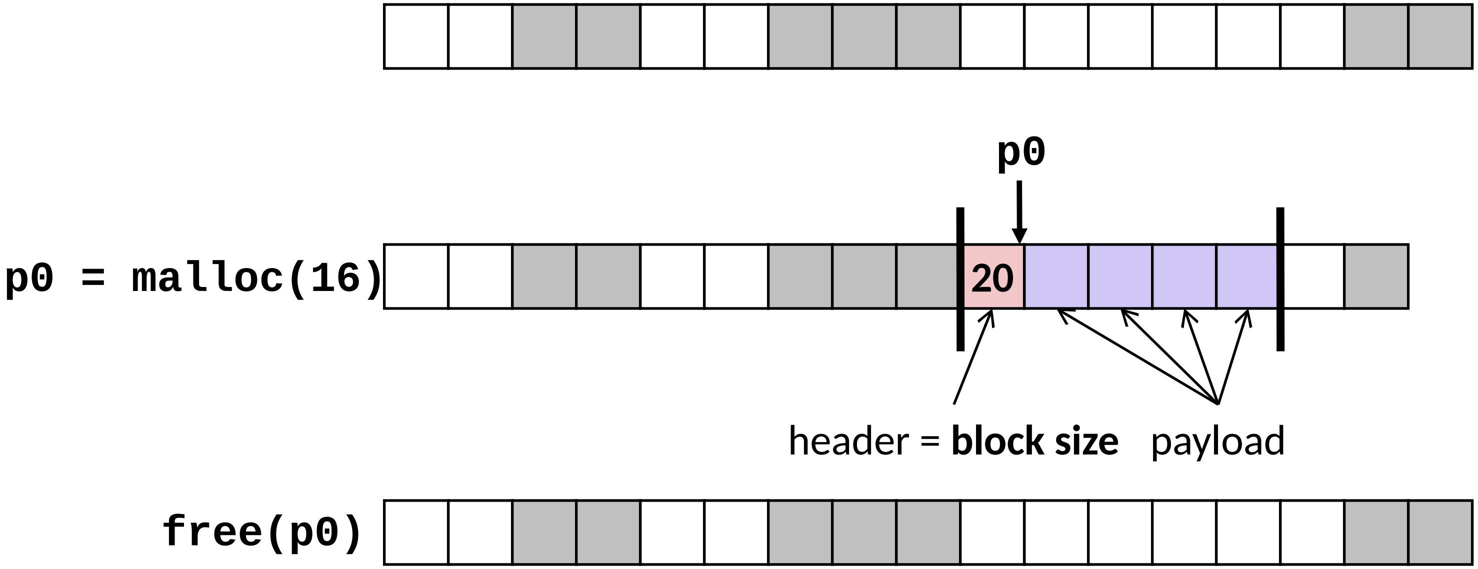
Knowing How Much to Free

- 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 (4 byte) header for every allocated 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 (4 byte) header for every allocated block



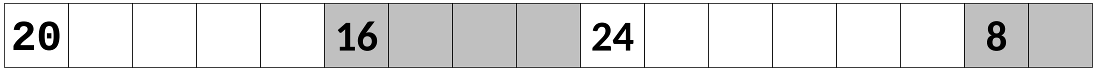
Challenges

- Goal: maximize throughput and peak memory utilization

- Implementation Challenges:
  - How do we know how much memory to free given just a pointer?
  - How do we keep track of the free blocks?
  - How do we pick a block to use for allocation?
  - 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 reinsert a freed block?

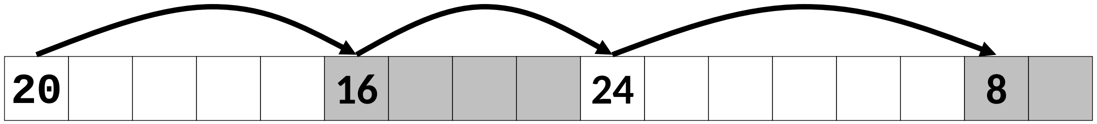
Keeping Track of Free Blocks

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



Keeping Track of Free Blocks

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



Method 1: Implicit List

- For each block we need both size and allocation status
  - Could store this information in two ints: wasteful!

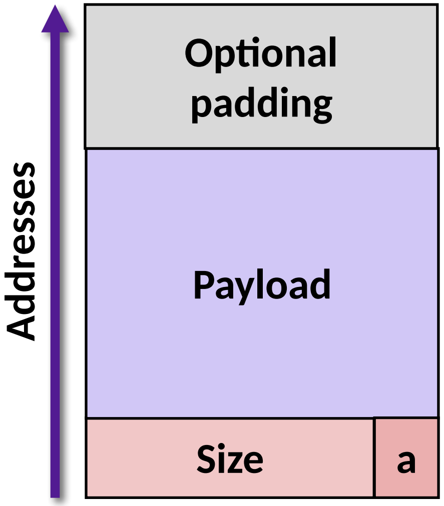
Method 1: Implicit List

- For each block we need both size and allocation status
  - Could store this information in two ints: 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 an allocated/free flag
  - When reading size word, must mask out this bit

Method 1: Implicit List

- For each block we need both size and allocation status
  - Could store this information in two ints: 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 an allocated/free flag
  - When reading size word, must mask out this bit

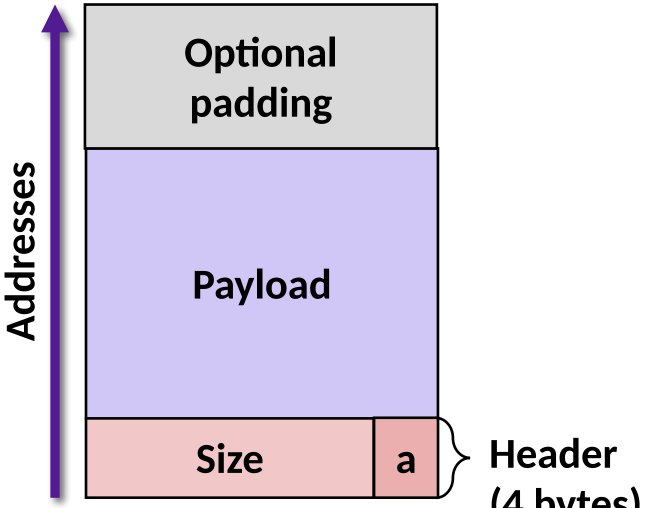
Format of allocated and free blocks



Method 1: Implicit List

- For each block we need both size and allocation status
  - Could store this information in two ints: 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 an allocated/free flag
  - When reading size word, must mask out this bit

Format of allocated and free blocks



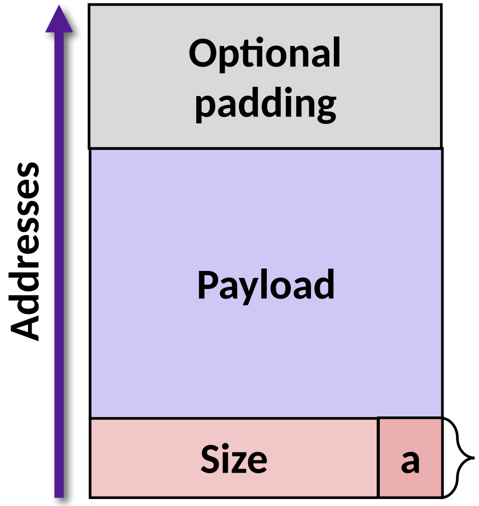
Size: total block size (incl header + padding)

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

Method 1: Implicit List

- For each block we need both size and allocation status
  - Could store this information in two ints: 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 an allocated/free flag
  - When reading size word, must mask out this bit

Format of allocated and free blocks



Header (4 bytes)

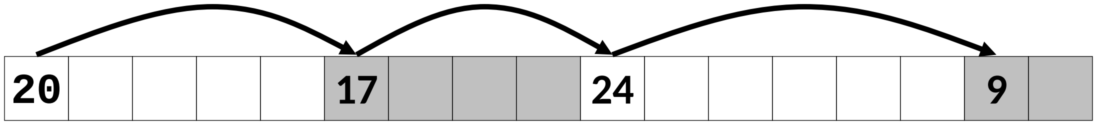
Payload: application data (allocated blocks only)

Size: total block size (incl header + padding)

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

Keeping Track of Free Blocks

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



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

Exercise: Block Headers

- Determine the block sizes and header values that would result from the following sequence of malloc requests. Assume that the allocator uses an implicit list implementation with the block format just described and maintains 8-byte alignment.

Request	Block size (decimal)	Block header (hex)
malloc(1)		
malloc(5)		
malloc(12)		

Exercise: Block Headers

- Determine the block sizes and header values that would result from the following sequence of malloc requests. Assume that the allocator uses an implicit list implementation with the block format just described and maintains 8-byte alignment.

Request	**Block size (decimal)**	**Block header (hex)**
malloc(1)		
malloc(5)		
malloc(12)		

Exercise: Block Headers

- Determine the block sizes and header values that would result from the following sequence of malloc requests. Assume that the allocator uses an implicit list implementation with the block format just described and maintains 8-byte alignment.

Request	Block size (decimal)	Block header (hex)
malloc(1)	8	0x00000009
malloc(5)		
malloc(12)		

Exercise: Block Headers

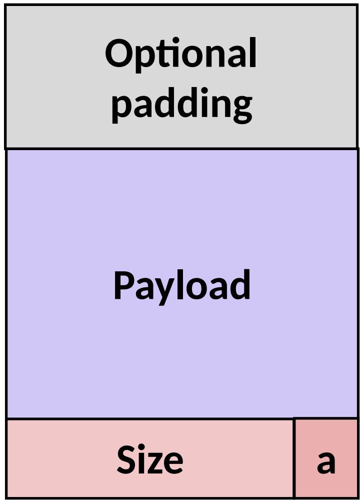
- Determine the block sizes and header values that would result from the following sequence of malloc requests. Assume that the allocator uses an implicit list implementation with the block format just described and maintains 8-byte alignment.

Request	Block size (decimal)	Block header (hex)
malloc(1)	8	0x00000009
malloc(5)	16	0x00000011
malloc(12)		

Exercise: Block Headers

- Determine the block sizes and header values that would result from the following sequence of malloc requests. Assume that the allocator uses an implicit list implementation with the block format just described and maintains 8-byte alignment.

Request	Block size (decimal)	Block header (hex)
malloc(1)	8	0x00000009
malloc(5)	16	0x00000011
malloc(12)	16	0x00000011



Optional padding

Payload

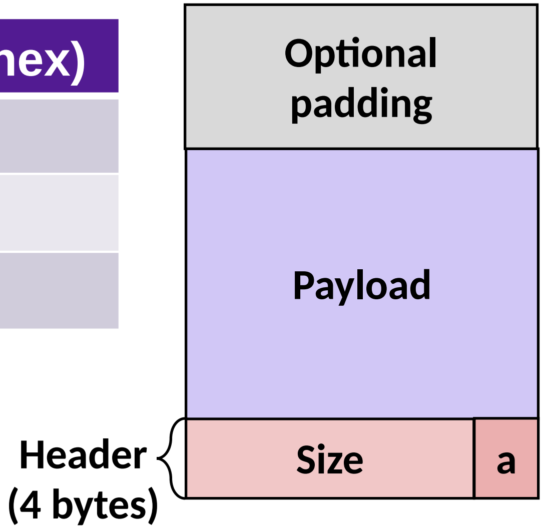
Size

a

Exercise: Block Headers

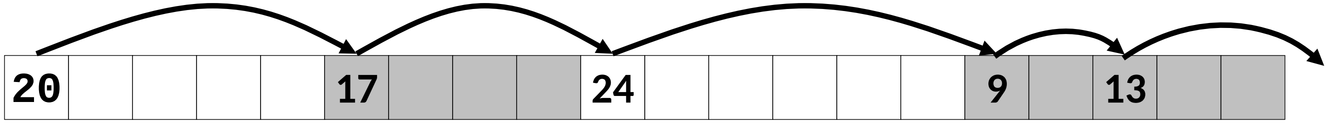
- Determine the block sizes and header values that would result from the following sequence of malloc requests. Assume that the allocator uses an implicit list implementation with the block format just described and maintains 8-byte alignment.

Request	Block size (decimal)	Block header (hex)
malloc(1)	8	0x00000009
malloc(5)	16	0x00000011
malloc(12)	16	0x00000011



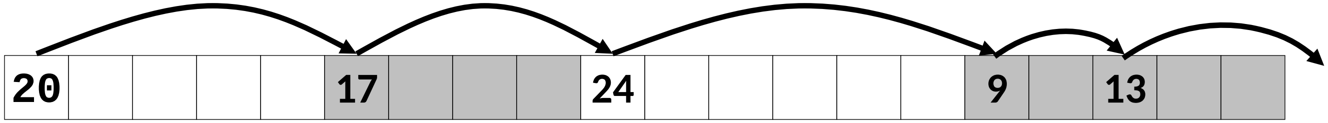
Keeping Track of Free Blocks

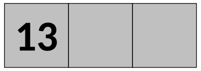
- Method 1: ***Implicit list*** using length—links all blocks



Keeping Track of Free Blocks

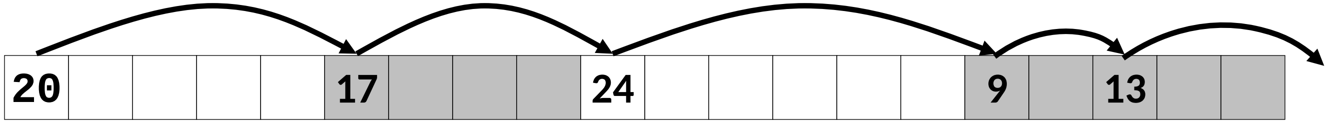
- Method 1: ***Implicit list*** using length—links all blocks



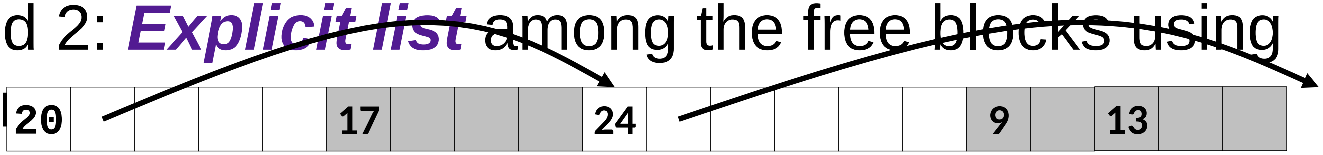


Keeping Track of Free Blocks

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

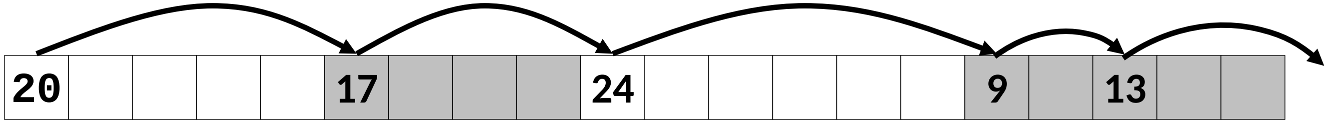


- Method 2: ***Explicit list*** among the free blocks using pointe

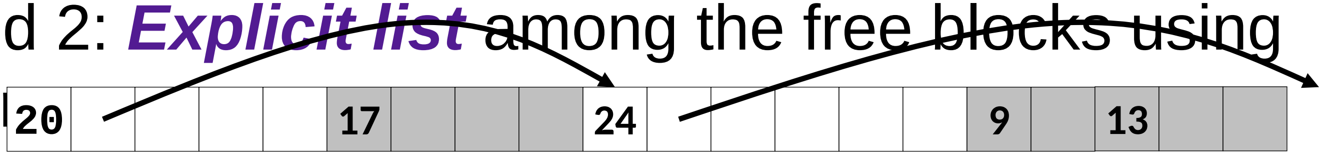


Keeping Track of Free Blocks

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



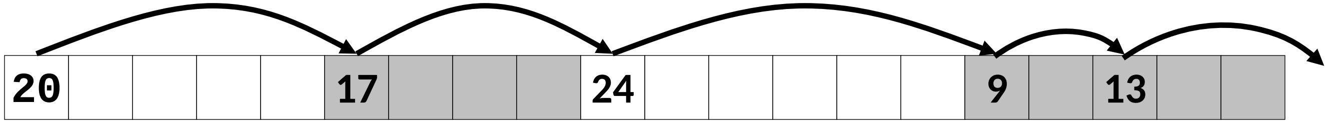
- Method 2: ***Explicit list*** among the free blocks using pointer



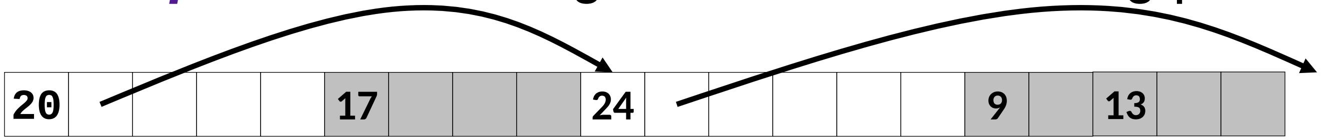
- Method 3: ***Segregated free list***
  - Different free lists for different size classes

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

Challenges

- Goal: maximize throughput and peak memory utilization

- Implementation Challenges:
  - How do we know how much memory to free given just a pointer?
  - How do we keep track of the free blocks?
  - How do we pick a block to use for allocation?
  - 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 reinsert a freed block?

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

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

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

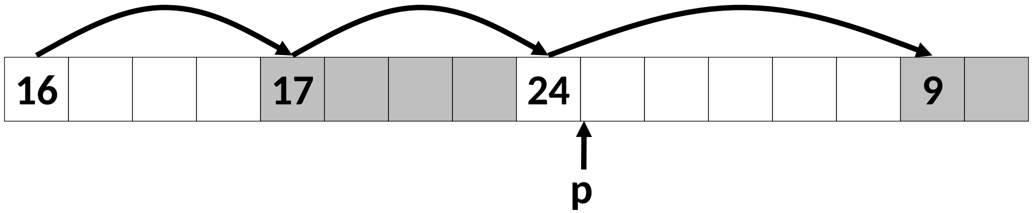
Challenges

- Goal: maximize throughput and peak memory utilization

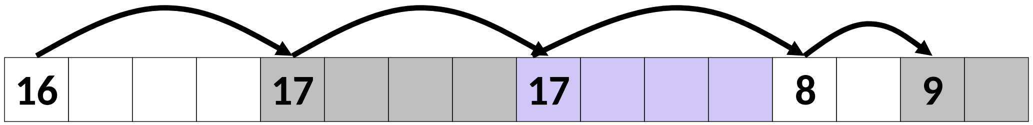
- Implementation Challenges:
  - How do we know how much memory to free given just a pointer?
  - How do we keep track of the free blocks?
  - How do we pick a block to use for allocation?
  - What do we do with the extra space when allocating a structure that is smaller than the free block it is placed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

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
```

Challenges

- Goal: maximize throughput and peak memory utilization

- Implementation Challenges:
  - How do we know how much memory to free given just a pointer?
  - How do we keep track of the free blocks?
  - How do we pick a block to use for allocation?
  - 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 reinsert a freed block?

Implicit List: Freeing a Block

- Simplest implementation:
  - Need only clear the “allocated” flag

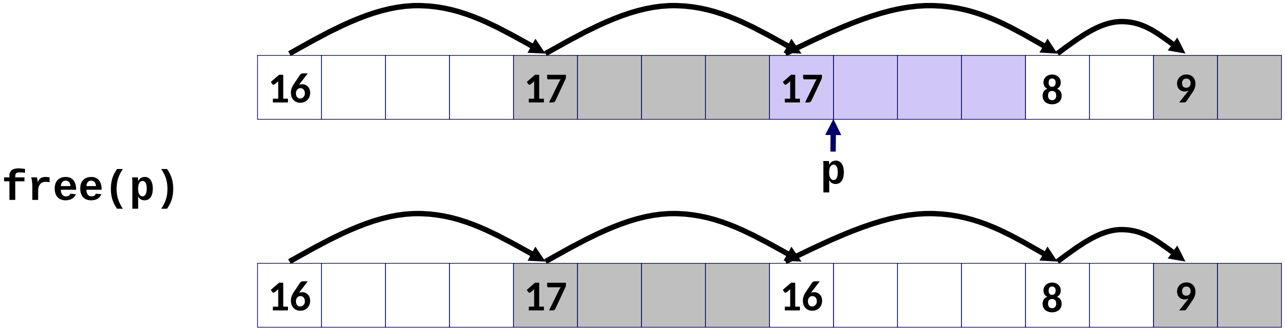
```
        void free_block(ptr p) { *p = *p & -2 }
```

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”

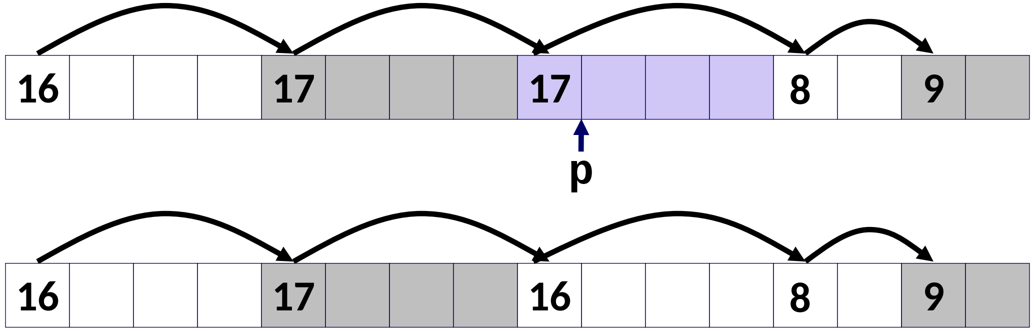


Implicit List: Freeing a Block

- Simplest implementation:
  - Need only clear the “allocated” flag

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

  - But can lead to “false fragmentation”



`free(p)`

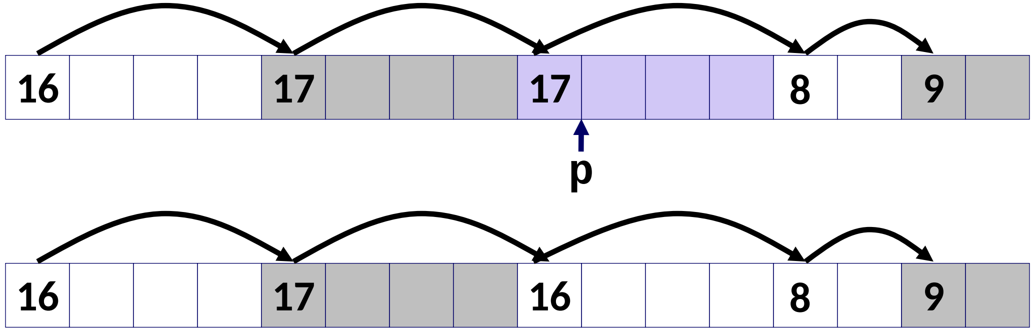
`malloc(20)` *Oops!*

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”



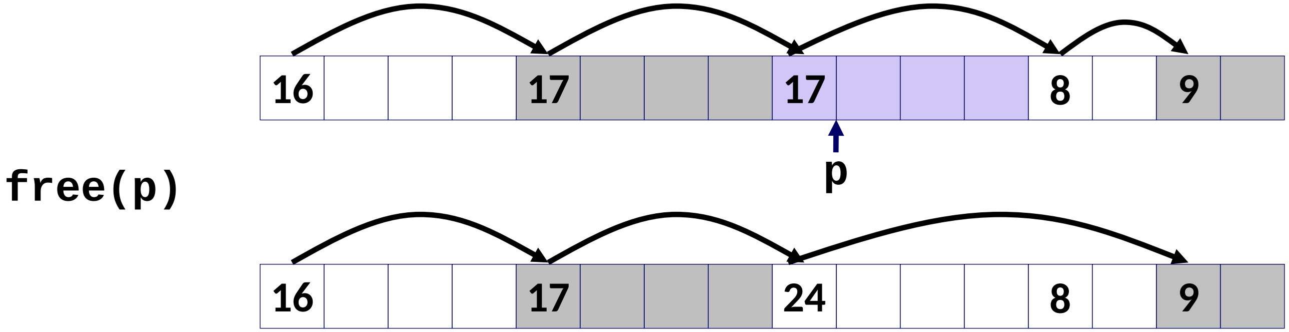
`free(p)`

`malloc(20)` *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

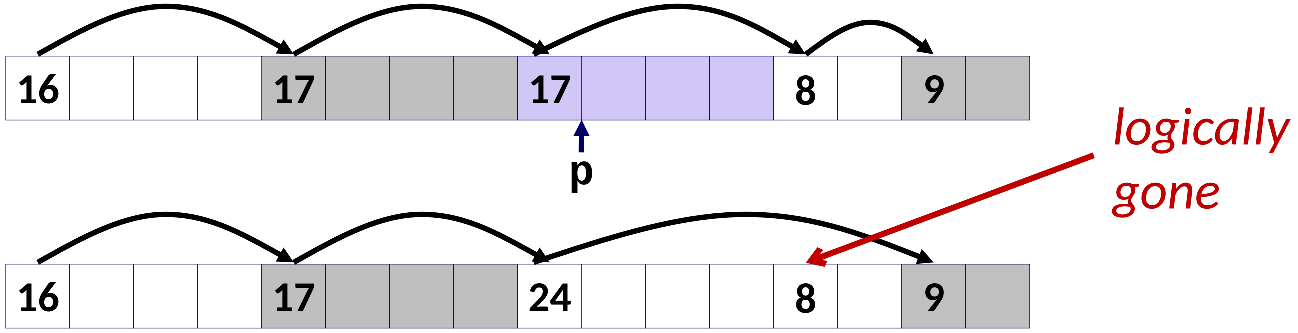


But how do we coalesce with previous block?

Implicit List: Coalescing

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

`free(p)`

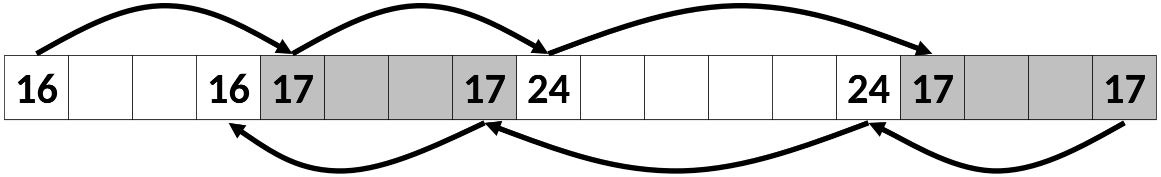


logically gone

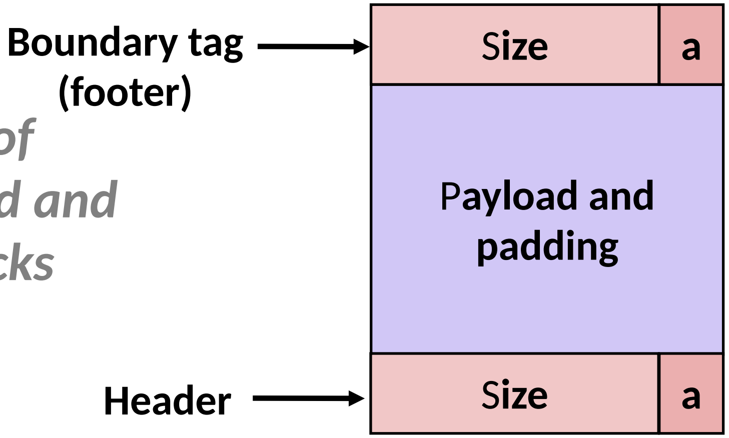
But how do we coalesce with previous block?

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



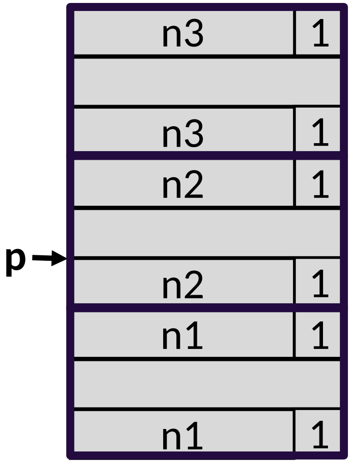
a = 1: Allocated block
a = 0: Free block

Size: Total block size

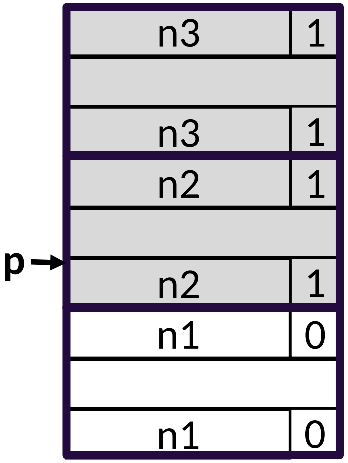
Payload: Application data (allocated blocks only)

Constant-Time Coalescing

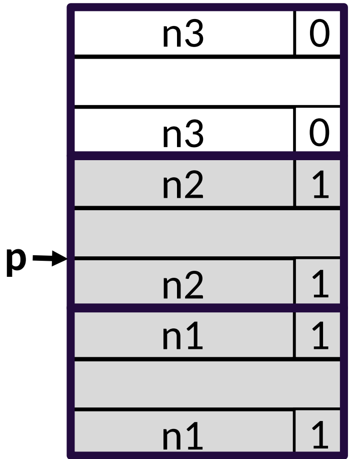
Case 1: Prev and next block allocated



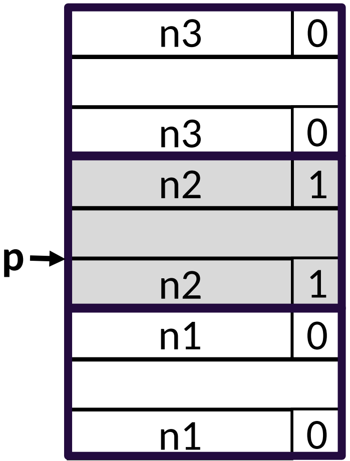
Case 2: Prev block free, next block allocated



Case 2: Prev block allocated, next block free

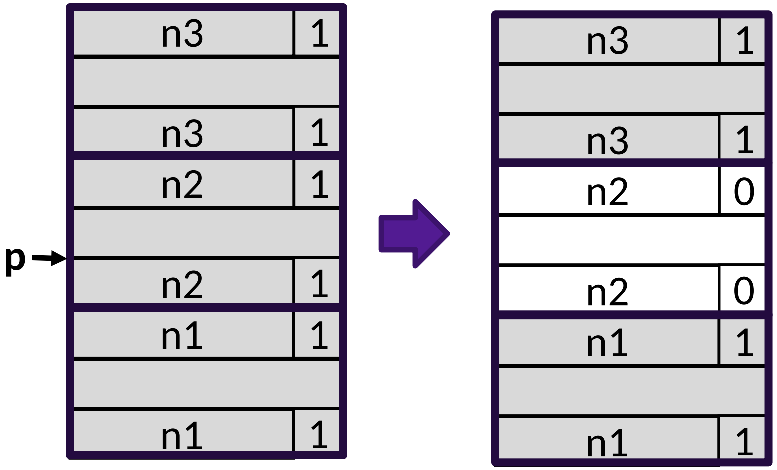


Case 4: Prev and next block free

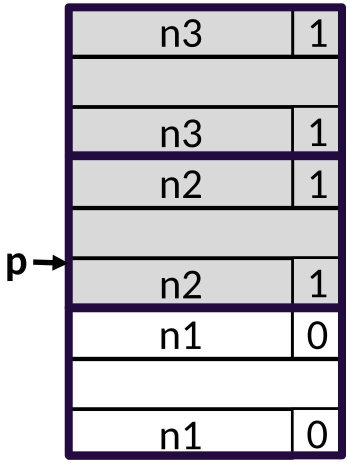


Constant-Time Coalescing

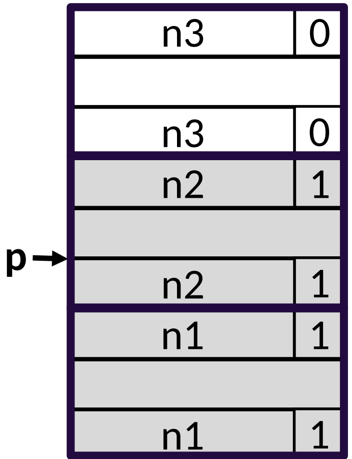
Case 1: Prev and next block allocated



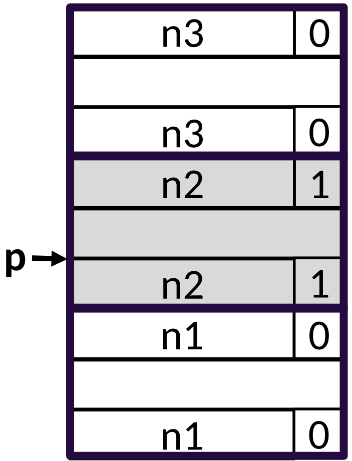
Case 2: Prev block free, next block allocated



Case 2: Prev block allocated, next block free

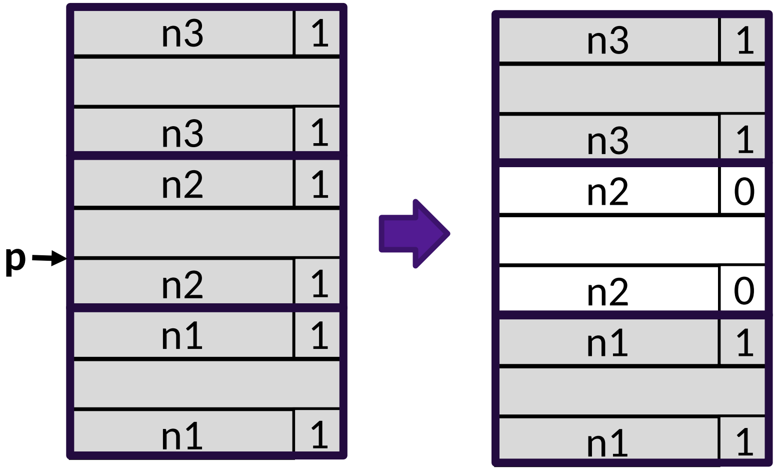


Case 4: Prev and next block free

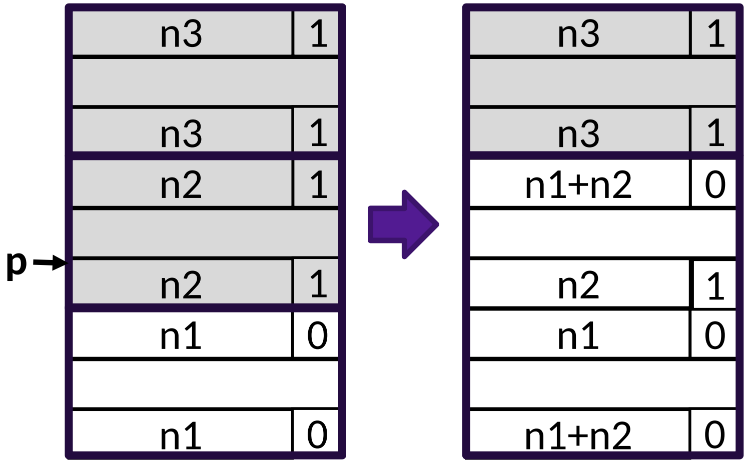


Constant-Time Coalescing

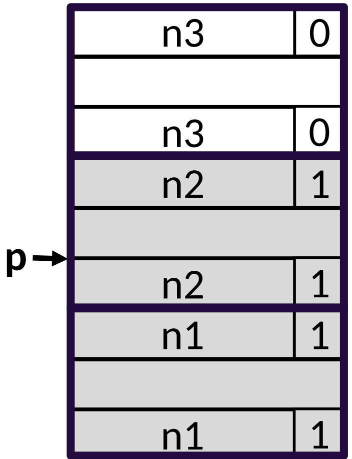
Case 1: Prev and next block allocated



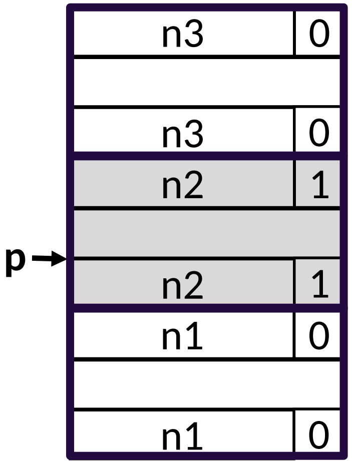
Case 2: Prev block free, next block allocated



Case 2: Prev block allocated, next block free

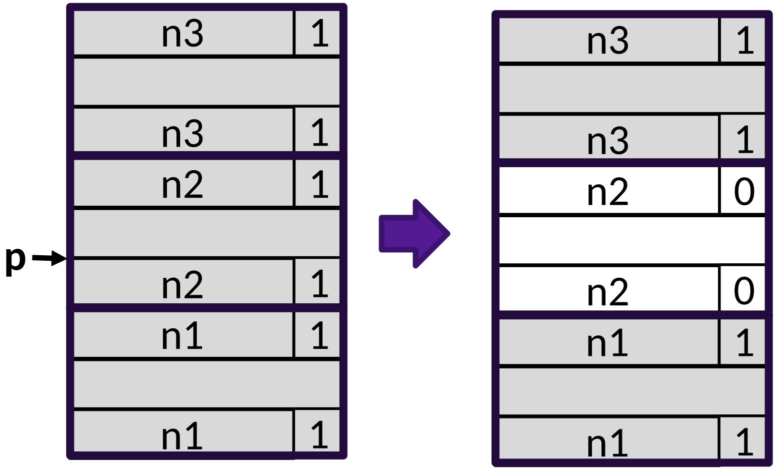


Case 4: Prev and next block free

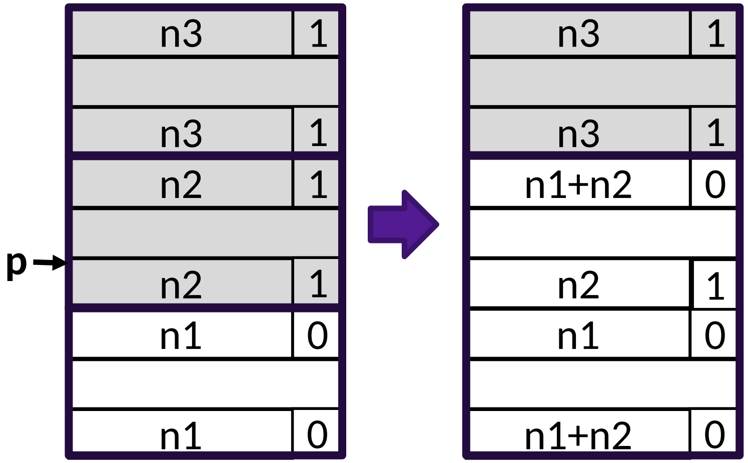


Constant-Time Coalescing

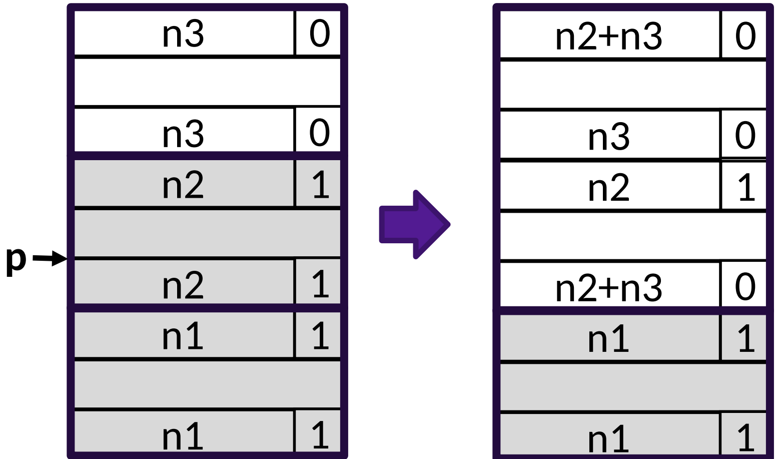
Case 1: Prev and next block allocated



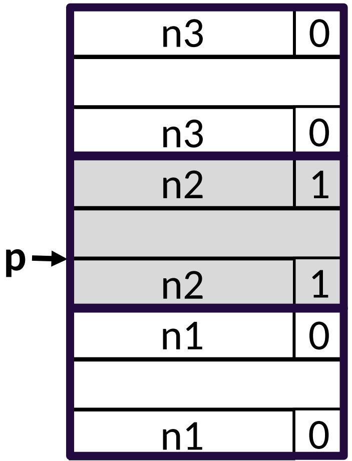
Case 2: Prev block free, next block allocated



Case 2: Prev block allocated, next block free

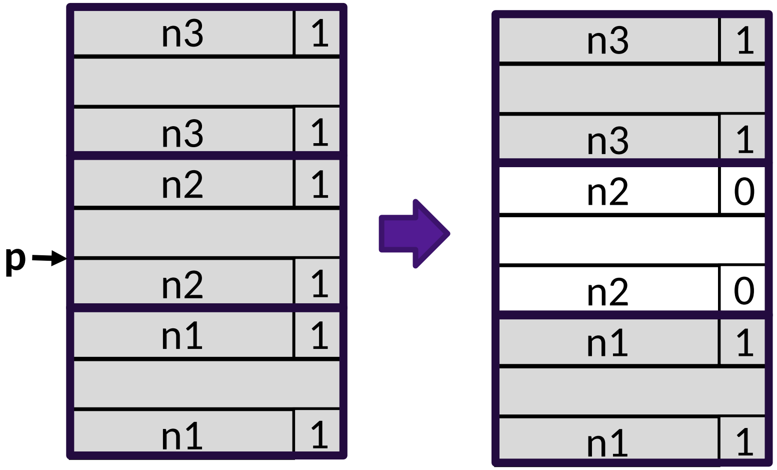


Case 4: Prev and next block free

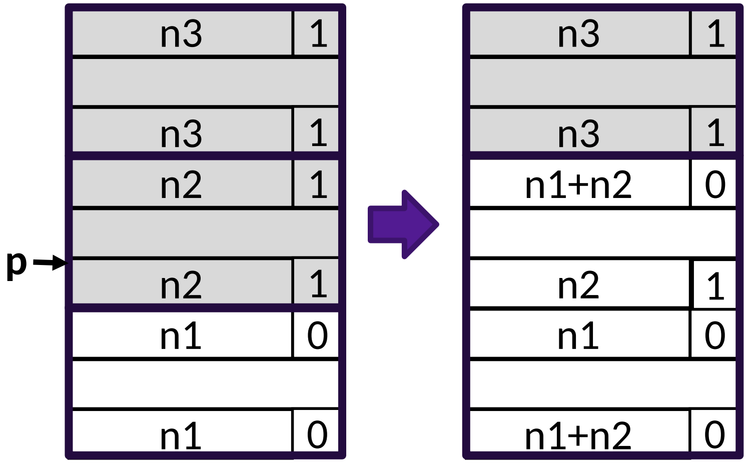


Constant-Time Coalescing

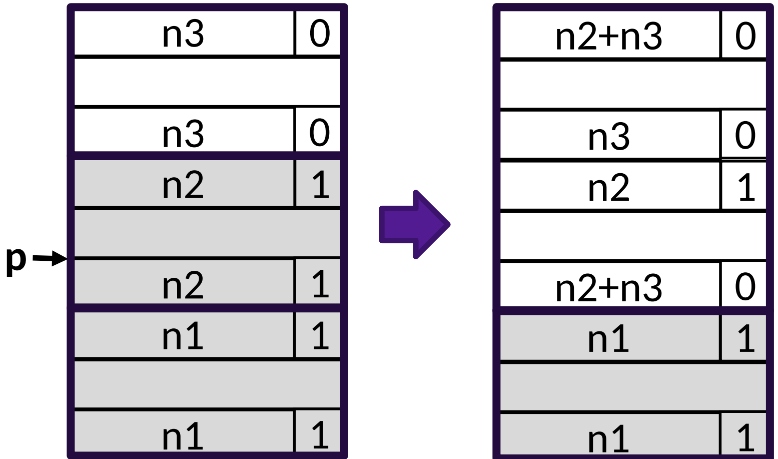
Case 1: Prev and next block allocated



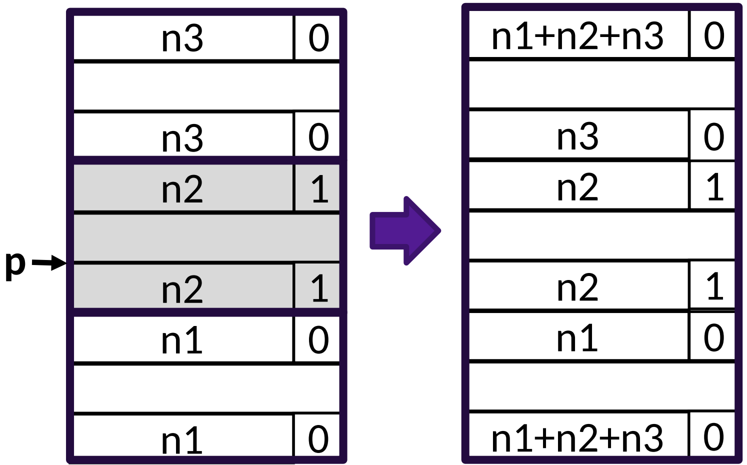
Case 2: Prev block free, next block allocated



Case 2: Prev block allocated, next block free

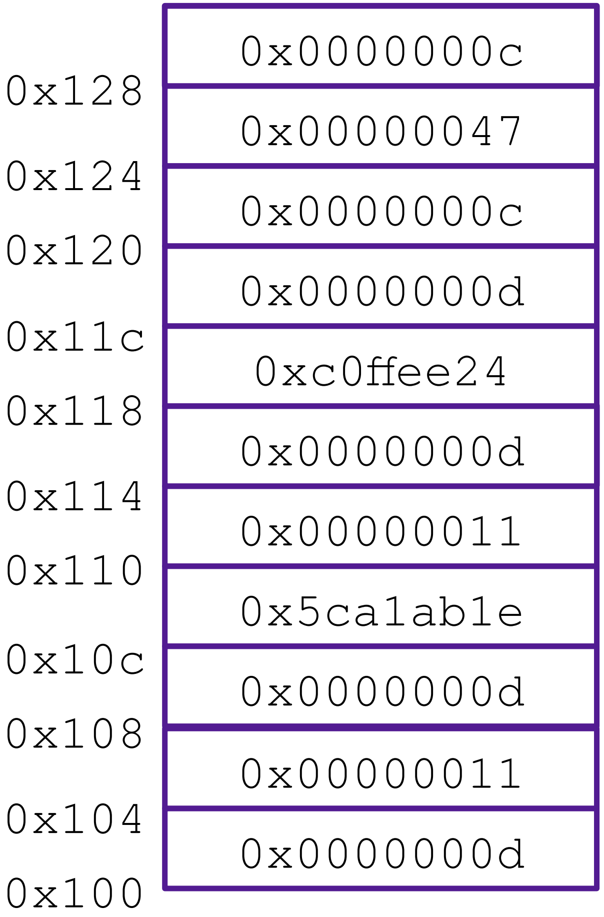


Case 4: Prev and next block free



Exercise: Coalescing

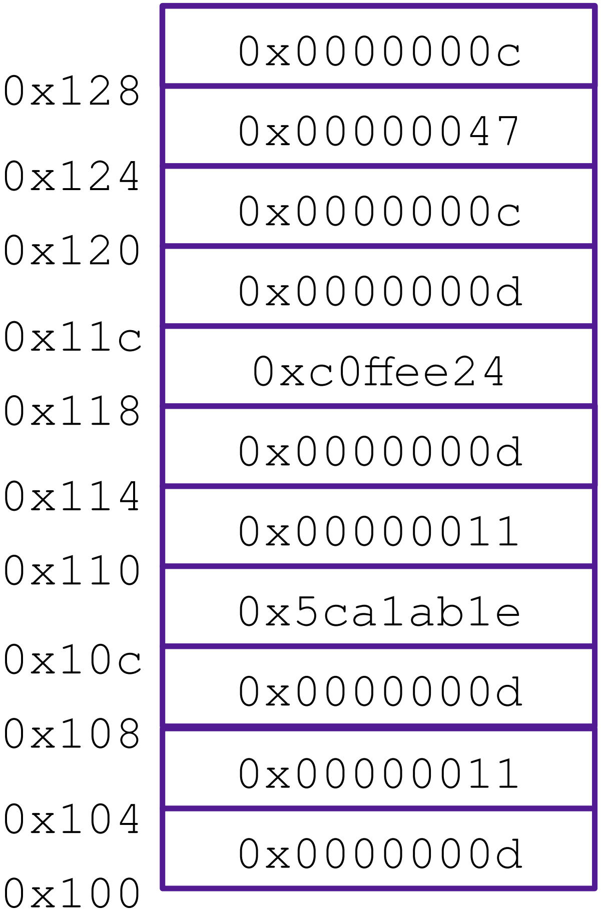
- Assume the current heap is shown below. What would be the state of the heap after the function `free(0x118)` is executed?



	0x0000000c
0x128	0x00000047
0x124	0x0000000c
0x120	0x0000000d
0x11c	0xc0ffee24
0x118	0x0000000d
0x114	0x00000011
0x110	0x5ca1ab1e
0x10c	0x0000000d
0x108	0x00000011
0x104	0x0000000d
0x100	

Exercise: Coalescing

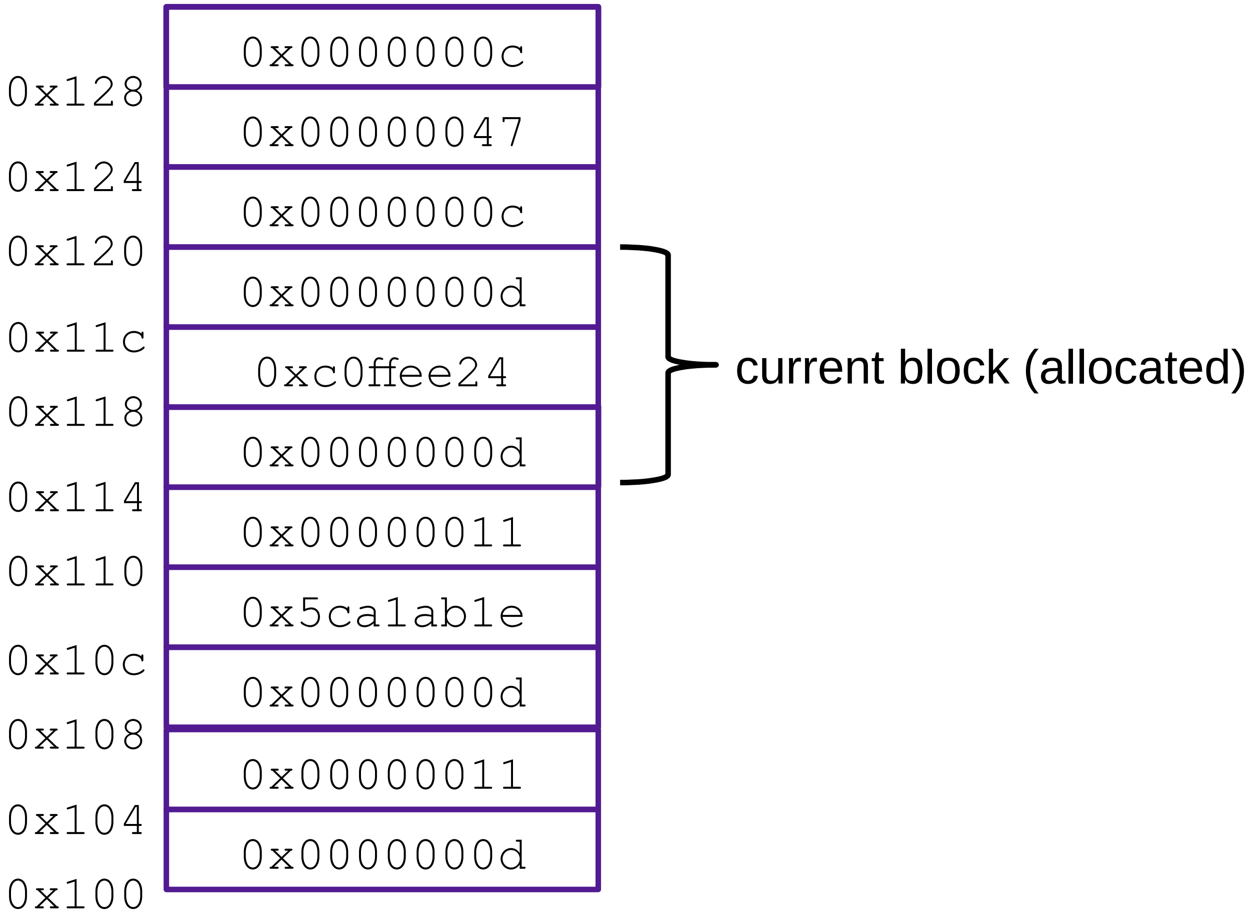
- Assume the current heap is shown below. What would be the state of the heap after the function `free(0x118)` is executed?



	0x0000000c
0x128	0x00000047
0x124	0x0000000c
0x120	0x0000000d
0x11c	0xc0ffee24
0x118	0x0000000d
0x114	0x00000011
0x110	0x5ca1ab1e
0x10c	0x0000000d
0x108	0x00000011
0x104	0x0000000d
0x100	

Exercise: Coalescing

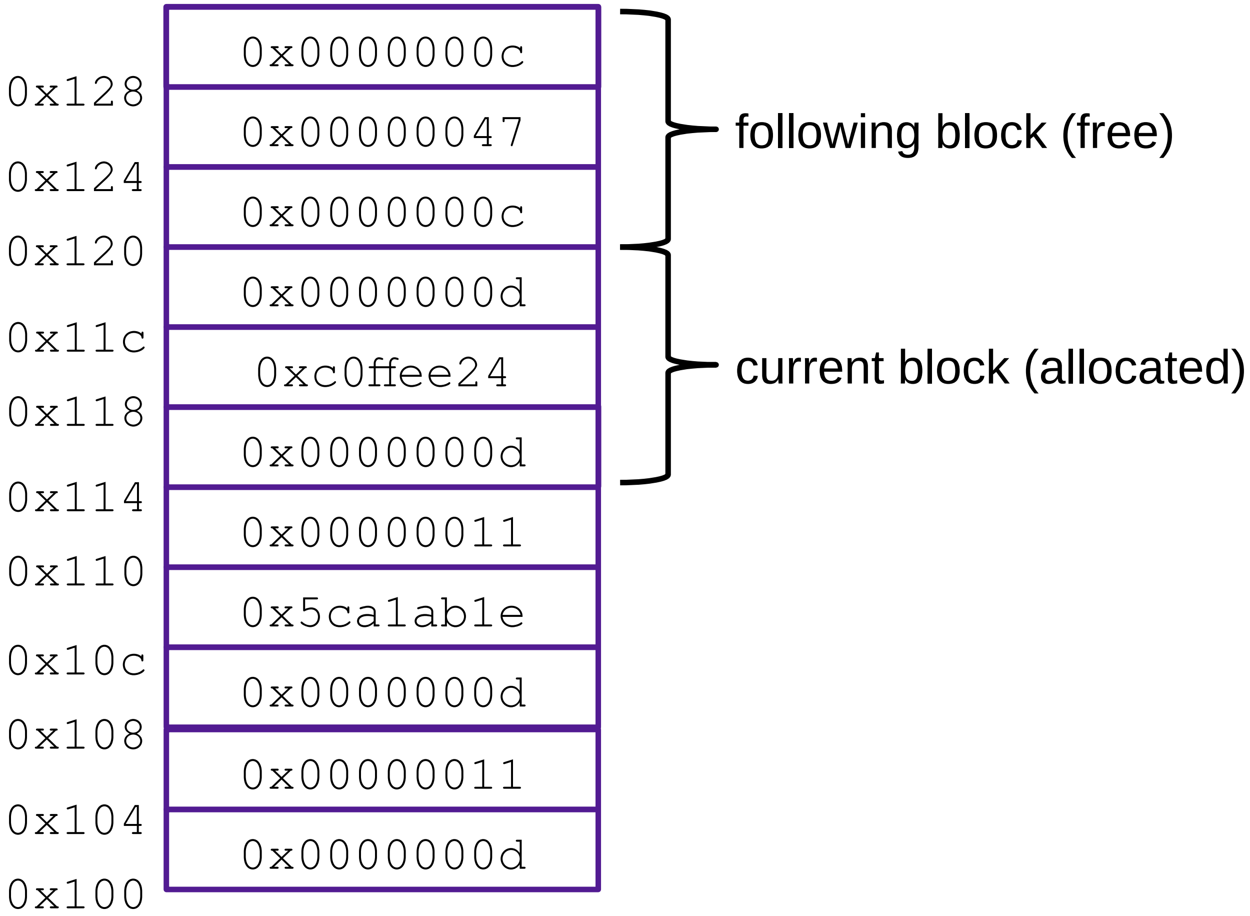
- Assume the current heap is shown below. What would be the state of the heap after the function `free(0x118)` is executed?



0x128	0x0000000c
0x124	0x00000047
0x120	0x0000000c
0x11c	0x0000000d
0x118	0xc0ffee24
0x114	0x0000000d
0x110	0x00000011
0x10c	0x5ca1ab1e
0x108	0x0000000d
0x104	0x00000011
0x100	0x0000000d

Exercise: Coalescing

- Assume the current heap is shown below. What would be the state of the heap after the function `free(0x118)` is executed?



0x0000000c

0x128

0x00000047

0x124

0x0000000c

0x120

0x0000000d

0x11c

0xc0ffee24

0x118

0x0000000d

0x114

0x00000011

0x110

0x5ca1ab1e

0x10c

0x0000000d

0x108

0x00000011

0x104

0x0000000d

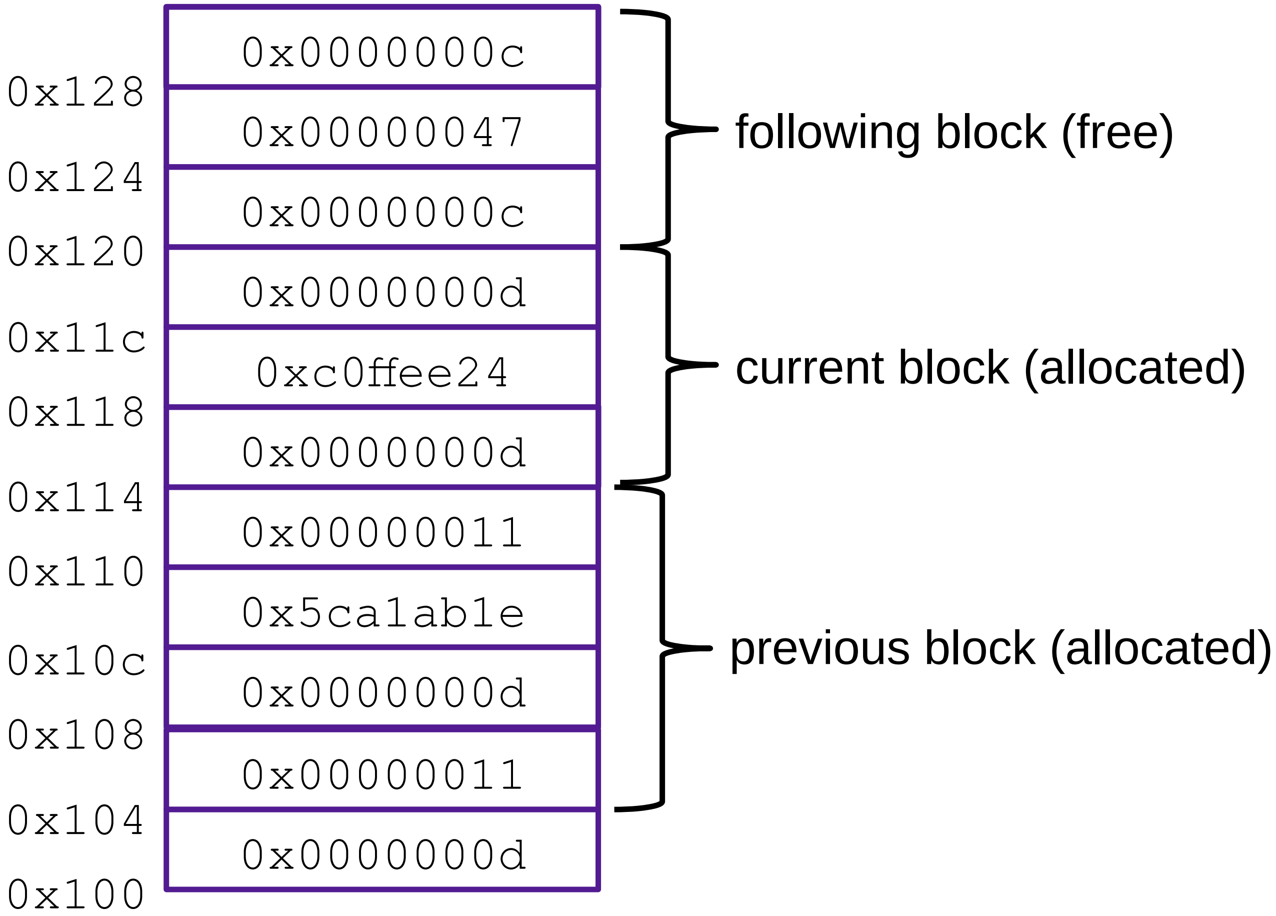
0x100

following block (free)

current block (allocated)

Exercise: Coalescing

- Assume the current heap is shown below. What would be the state of the heap after the function `free(0x118)` is executed?



```
         0x0000000c
0x128
         0x00000047
0x124
         0x0000000c
0x120
         0x0000000d
0x11c
         0xc0ffee24
0x118
         0x0000000d
0x114
         0x00000011
0x110
         0x5ca1ab1e
0x10c
         0x0000000d
0x108
         0x00000011
0x104
         0x0000000d
0x100
```

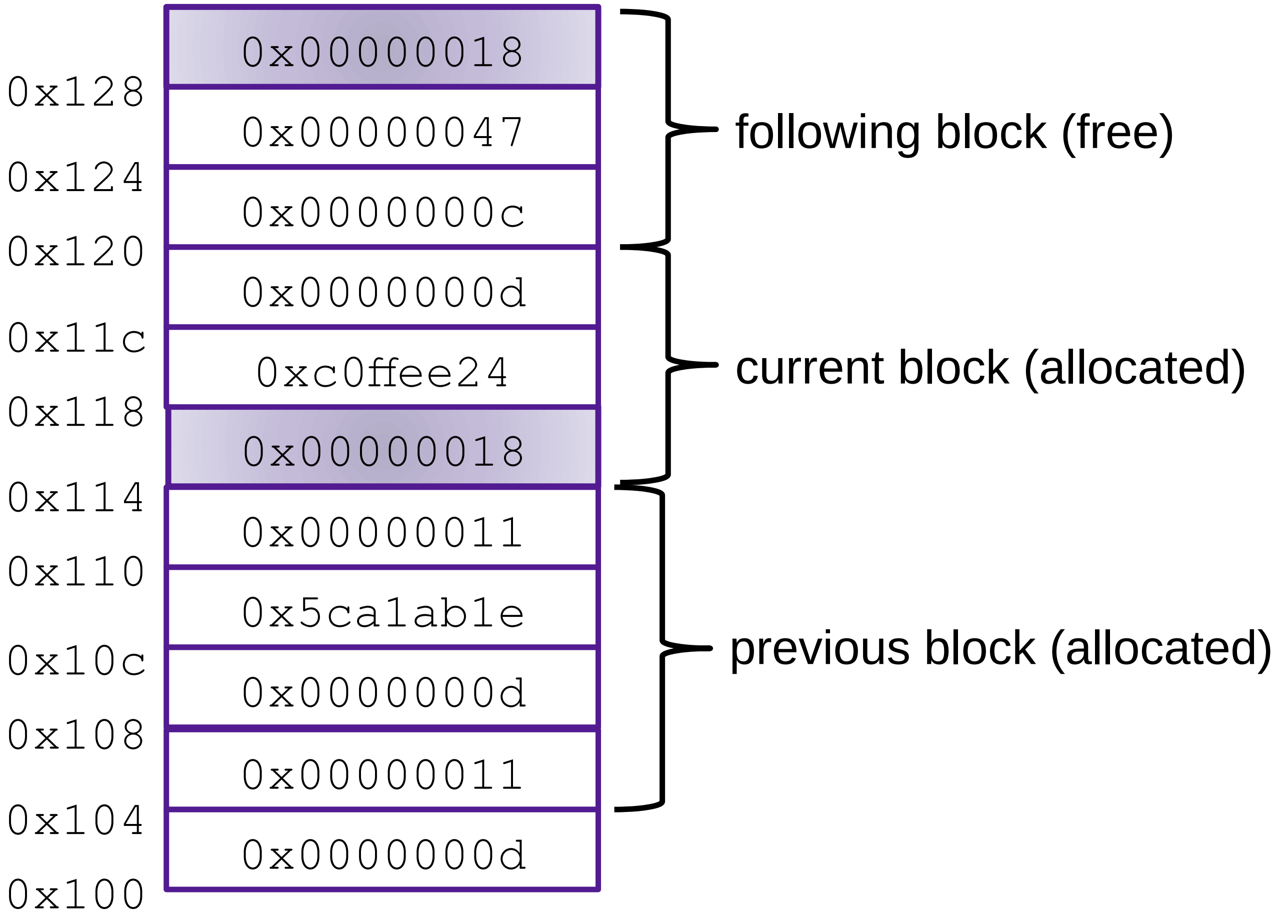
following block (free)

current block (allocated)

previous block (allocated)

Exercise: Coalescing

- Assume the current heap is shown below. What would be the state of the heap after the function `free(0x118)` is executed?



0x00000018
0x00000047
0x0000000c
0x0000000d
0xc0ffee24
0x00000018
0x00000011
0x5ca1ab1e
0x0000000d
0x00000011
0x0000000d

Summary of Key Allocator Policies

- Free-block storage policy:
  - Implicit lists, with boundary tags (nice and simple)
  - Explicit lists, exclude free blocks (faster, but more overhead)
  - Segregated lists (different lists for different sized blocks)
  - Fancy data structures (red-black trees, for example)

Summary of Key Allocator Policies

- Free-block storage policy:
  - Implicit lists, with boundary tags (nice and simple)
  - Explicit lists, exclude free blocks (faster, but more overhead)
  - Segregated lists (different lists for different sized blocks)
  - Fancy data structures (red-black trees, for example)

- Placement policy:
  - First-fit (simple, but lower throughput and higher fragmentation)
  - Next-fit (higher throughput, higher fragmentation)
  - Best-fit (lower throughput, lower fragmentation
  - segregated free lists approximate a best fit placement policy without having to search entire free list

Summary of Key Allocator Policies

- Free-block storage policy:
  - Implicit lists, with boundary tags (nice and simple)
  - Explicit lists, exclude free blocks (faster, but more overhead)
  - Segregated lists (different lists for different sized blocks)
  - Fancy data structures (red-black trees, for example)

- Placement policy:
  - First-fit (simple, but lower throughput and higher fragmentation)
  - Next-fit (higher throughput, higher fragmentation)
  - Best-fit (lower throughput, lower fragmentation
  - segregated free lists approximate a best fit placement policy without having to search entire free list

- Splitting policy:
  - When do we go ahead and split free blocks?
  - How much internal fragmentation are we willing to tolerate?

Summary of Key Allocator Policies

- Free-block storage policy:
  - Implicit lists, with boundary tags (nice and simple)
  - Explicit lists, exclude free blocks (faster, but more overhead)
  - Segregated lists (different lists for different sized blocks)
  - Fancy data structures (red-black trees, for example)

- Placement policy:
  - First-fit (simple, but lower throughput and higher fragmentation)
  - Next-fit (higher throughput, higher fragmentation)
  - Best-fit (lower throughput, lower fragmentation
  - segregated free lists approximate a best fit placement policy without having to search entire free list

- Splitting policy:
  - When do we go ahead and split free blocks?
  - How much internal fragmentation are we willing to tolerate?

- Coalescing policy:
  - No coalescing (bad choice)
  - ***Immediate coalescing:*** coalesce each time **`free`** is called
  - ***Deferred coalescing:*** coalesce on allocate or after fixed time

Memory-Related Perils and Pitfalls

- Dereferencing bad pointers
- Reading uninitialized memory
- Overreading memory
- Overwriting memory
- Referencing freed blocks
- Freeing blocks multiple times
- Failing to free blocks

Memory-Related Perils and Pitfalls

- Dereferencing bad pointers (Correctness)
- Reading uninitialized memory (Correctness)
- Overreading memory
- Overwriting memory
- Referencing freed blocks
- Freeing blocks multiple times
- Failing to free blocks

Memory-Related Perils and Pitfalls

- Dereferencing bad pointers (Correctness)
- Reading uninitialized memory (Correctness)
- Overreading memory (Security)
- Overwriting memory (Security)
- Referencing freed blocks
- Freeing blocks multiple times
- Failing to free blocks

Memory-Related Perils and Pitfalls

• Dereferencing bad pointers	(Correctness)
• Reading uninitialized memory	(Correctness)
• Overreading memory	(Security)
• Overwriting memory	(Security)
• Referencing freed blocks	(Security)
• Freeing blocks multiple times	(Security)
• Failing to free blocks	

Memory-Related Perils and Pitfalls

• Dereferencing bad pointers	(Correctness)
• Reading uninitialized memory	(Correctness)
• Overreading memory	(Security)
• Overwriting memory	(Security)
• Referencing freed blocks	(Security)
• Freeing blocks multiple times	(Security)
• Failing to free blocks	(Performance)

Memory Bugs Persist…

Memory Bugs Persist…

Memory Bugs Persist…

Memory Bugs Persist…

Memory Bugs Persist…