Drawing: Return Oriented Programming

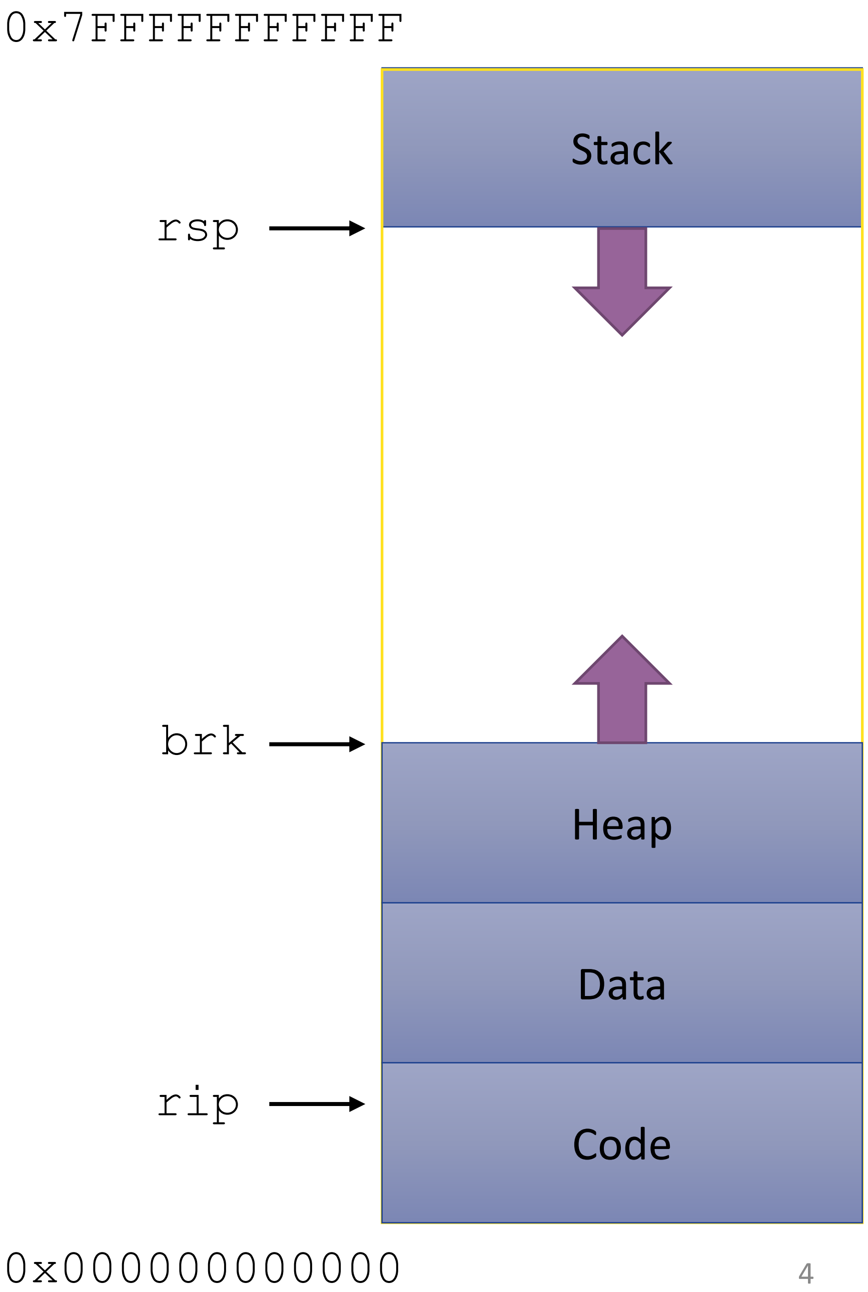
- Take three minutes to draw something related to “ROP”

- Some reminders
    - Stack smashing
    - Gadgets
    - Memory segments

Dynamic Memory

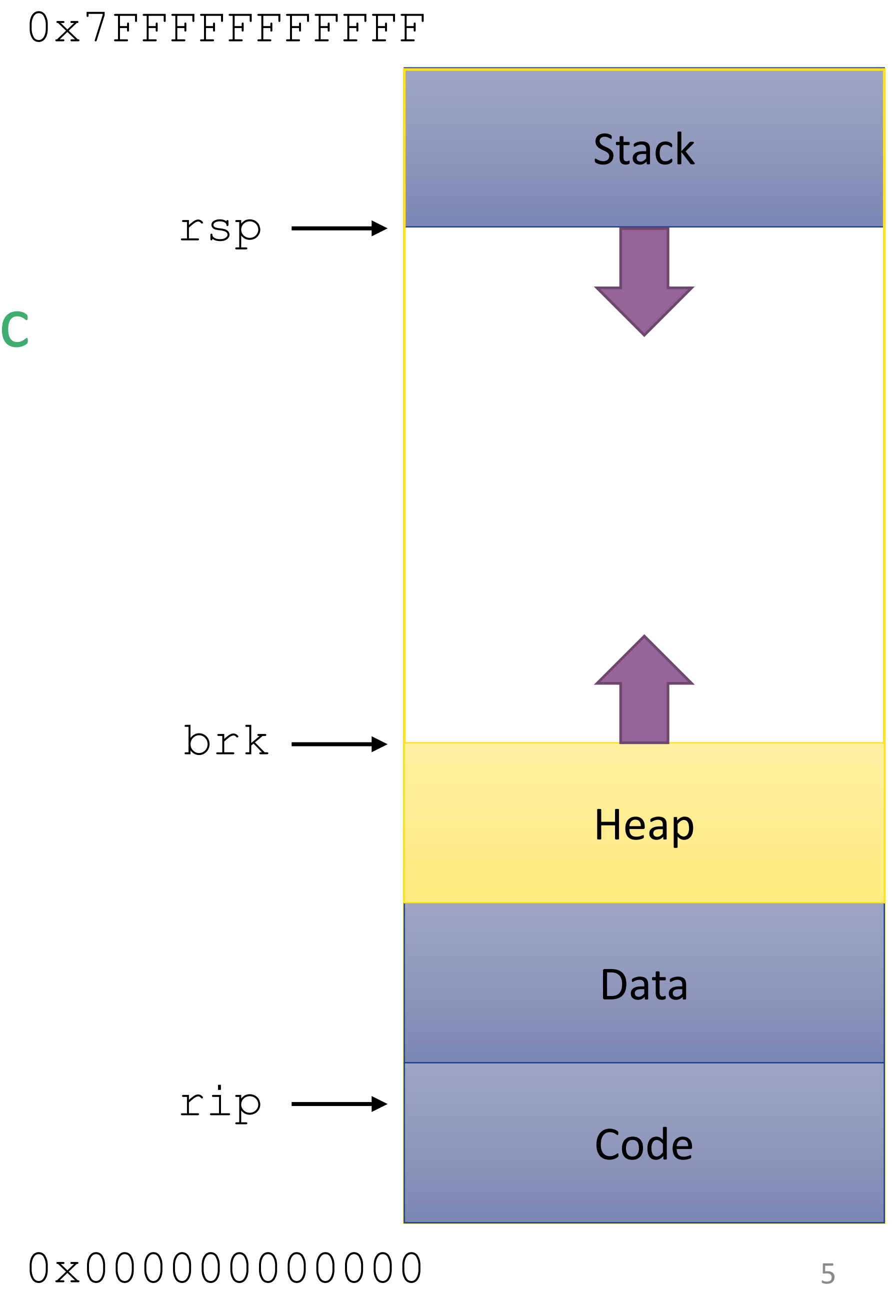
Memory

- Byte addressable array made up of four logical segments
- Stack provides local storage for procedures
- Heap is an area of memory maintained by a dynamic memory allocator (operating system maintains variable `brk` that points to the top of heap)
- Data stores global variables
- Code stores program instructions
- Attempt to access uninitialized addresses result in exceptions (segfault)



The Heap

- The heap is an area of memory for dynamic memory allocation
- Programmers can use a dynamic memory allocator to acquire additional memory at run time
- Programmers can use a system call to modify `brk` (but you should really just use `malloc` when programming in C)



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

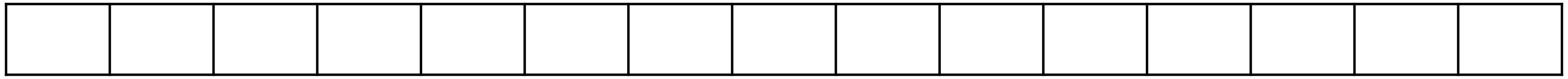
- object creation & garbage collection in Java
- object creation & garbage collection in Python

implicit allocators

Allocation Example using `malloc`

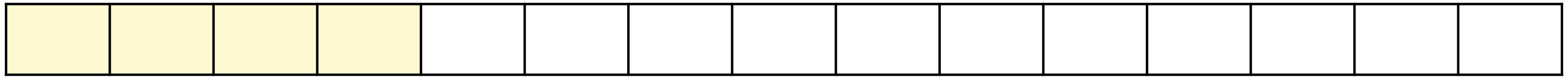
```c
#include <stdio.h>
#include <stdlib.h>

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);
}
```

Allocation Example



```
p1 = malloc(4)
```

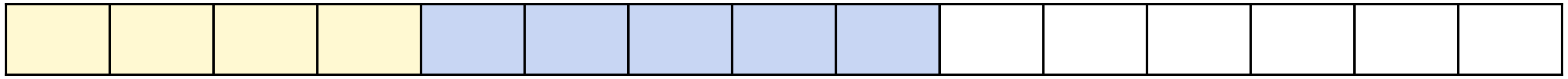
Allocation Example



```
p1 = malloc(4)
```

```
p2 = malloc(5)
```

Allocation Example

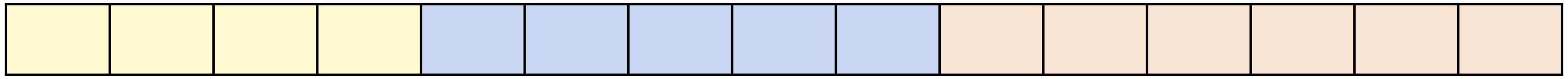


```
p1 = malloc(4)
```

```
p2 = malloc(5)
```

```
p3 = malloc(6)
```

Allocation Example



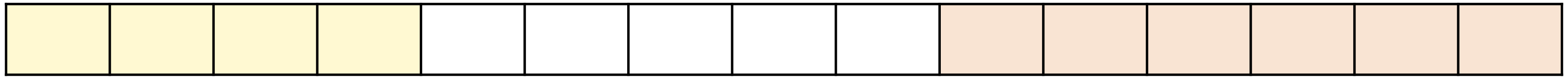
```
p1 = malloc(4)
```

```
p2 = malloc(5)
```

```
p3 = malloc(6)
```

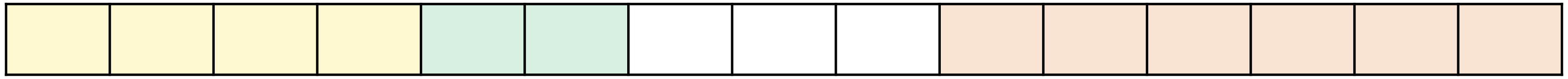
```
free(p2)
```

Allocation Example



```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(2)
```

Allocation Example



```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(2)
```

Allocator Requirements

1. Must handle arbitrary request sequences: cannot control number, size, or order of requests

2. Must respond immediately: no reordering or buffering requests

3. Must not modify allocated blocks: cannot modify or move blocks once they are allocated

4. Must align blocks: ensures that allocated blocks can hold any type of data

5. Must only use the heap: any helper data structures 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
}
```

Advantages
- Simple
- Blazing fast

Disadvantages
- Memory is never recycled
- Wastes a lot of space

Allocator Goals

High Throughput: number of requests completed per time unit

- Make the allocator fast
- Example: if your allocator processes 5,000 malloc calls and 5,000 free calls in 10 seconds then throughput is 1,000 operations/second

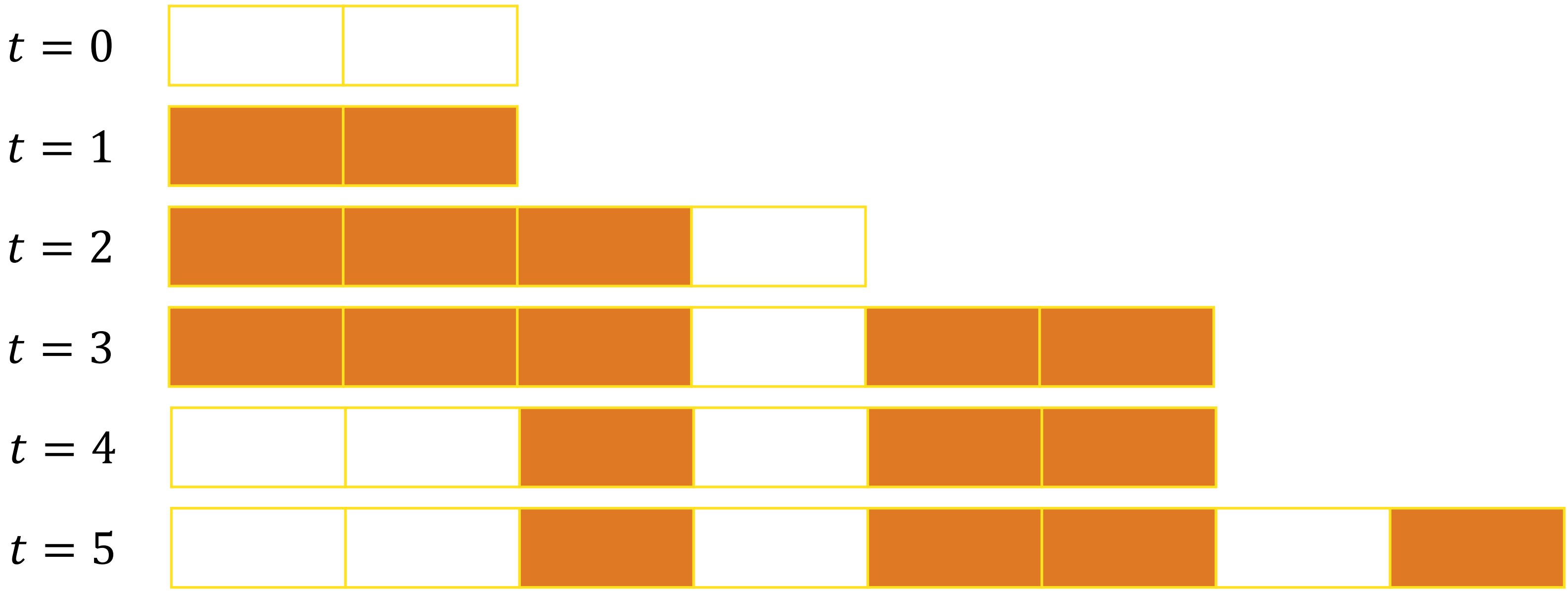
High Memory Utilization: fraction of heap memory allocated

- Minimize wasted space
- Maximize Peak Memory Utilization

$$U_t = \frac{\max_{i \le t} \mathit{space\ allocated\ at\ time\ i}}{\mathit{size\ of\ heap\ at\ time\ t}}$$

Practice with Memory Utilization

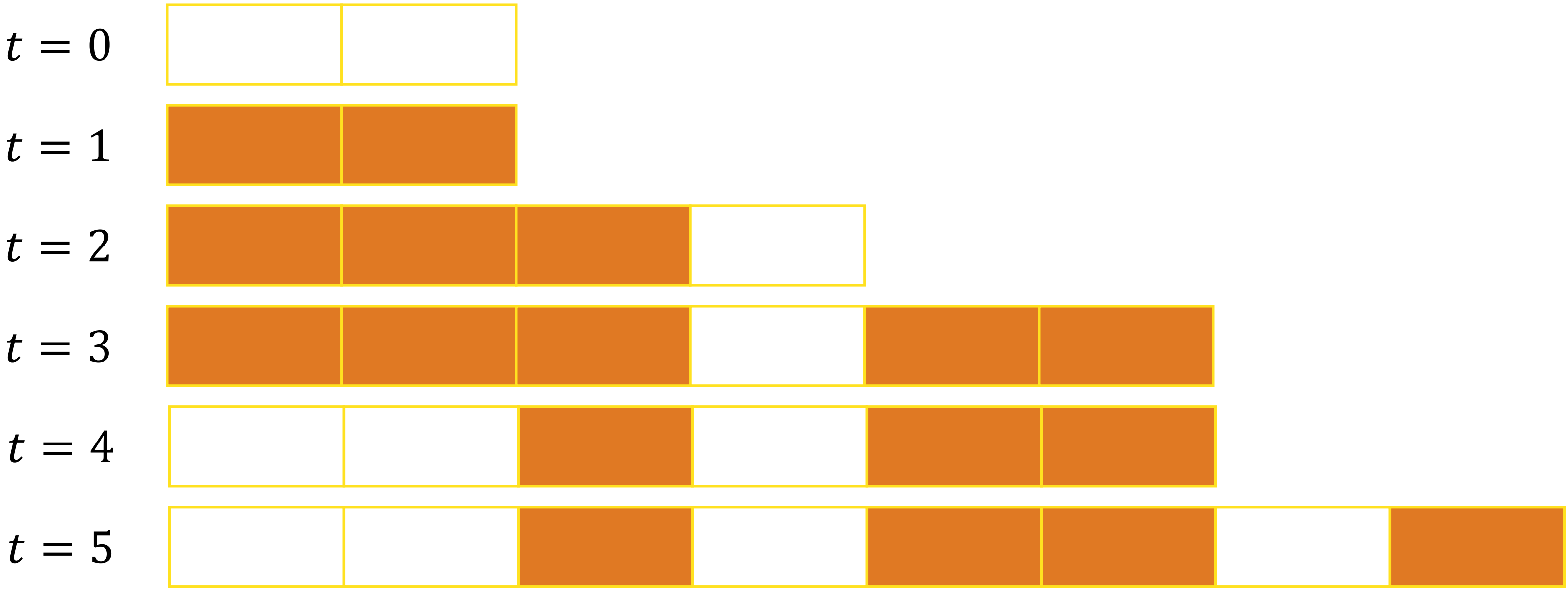
$$U_t = \frac{\max\limits_{i \le t} space\ allocated\ at\ time\ i}{size\ of\ heap\ at\ time\ t}$$



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

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

Allocator Goals

High Throughput: number of requests completed per time unit

- Make the allocator fast
- Example: if yo calls and 5,000 free calls in 10 sec ations/second

These goals are often conflicting

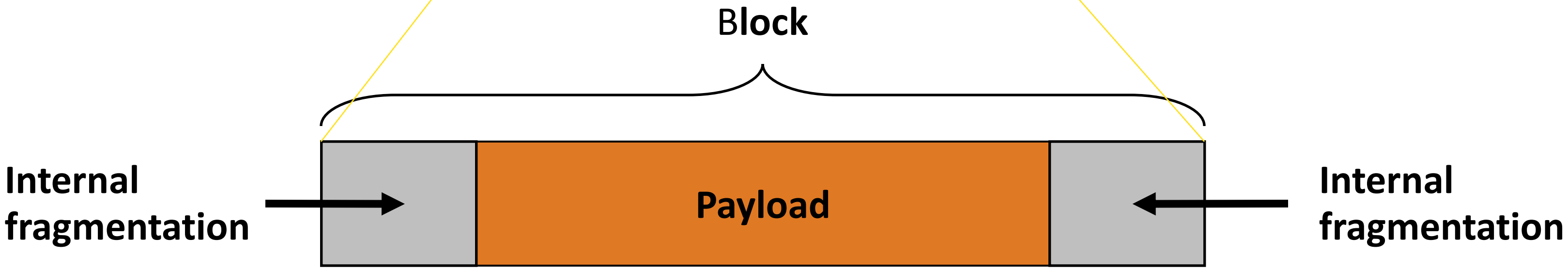
High Memory Utilization: fraction of heap memory allocated

- Minimize wasted space
- Maximize Peak Memory Utilization

$$U_t = \frac{\max_{i \le t} space\ allocated\ at\ time\ i}{size\ of\ heap\ at\ time\ t}$$

Utilization Blocker: Internal Fragmentation

For a given block, internal fragmentation occurs if the payload is smaller than the size of the block



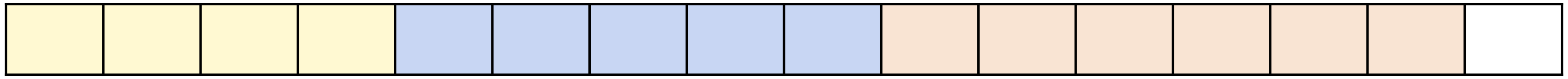
Caused by

- 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 and is easy to measure

Utilization Blocker: External Fragmentation

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

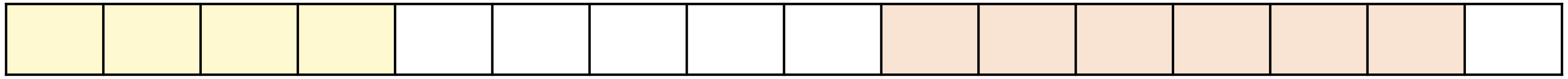


```
p1 = malloc(4)
p2 = malloc(5)
p3 = malloc(6)
free(p2)
```

- Depends on the pattern of future requests and is difficult to predict

Utilization Blocker: 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)
```

- Depends on the pattern of future requests and is difficult to predict

Practice with Utilization

$$U_t = \frac{\max_{i \le t} space\ allocated\ at\ time\ i}{size\ of\ heap\ at\ time\ t}$$

- Assume your heap is initially of size zero and you then run the following sequence of requests using the given allocator on a system with 4-byte alignment (all pointers start at a multiple of 4 address).
- What is the peak memory utilization after you complete the last request?

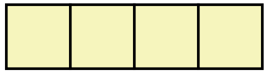
```
p1 = malloc(4)
p2 = malloc(5)
p3 = malloc(6)
free(p2)
p4 = malloc(2)
```

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

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

Practice with Utilization

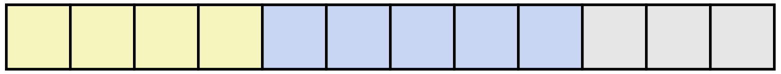
$$U_t = \frac{\max_{i \le t} space\ allocated\ at\ time\ i}{size\ of\ heap\ at\ time\ t}$$

```
p1 = malloc(4)
```



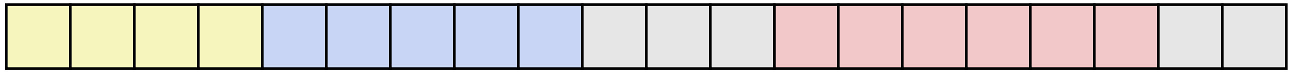
$$U_1 = \frac{4}{4}$$

```
p2 = malloc(5)
```



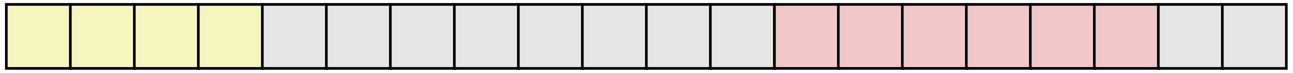
$$U_2 = \frac{9}{12} = \frac{3}{4}$$

```
p3 = malloc(6)
```



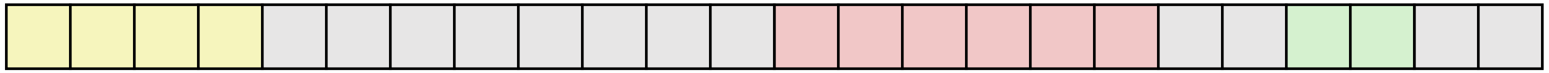
$$U_3 = \frac{15}{20} = \frac{3}{4}$$

```
free(p2)
```



$$U_4 = \frac{15}{20} = \frac{3}{4}$$

```
p4 = malloc(2)
```



$$U_5 = \frac{15}{24} = \frac{5}{8}$$

Challenges

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

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

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?

Challenges

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

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

Goal: maximize throughput and peak memory utilization

Implementation challenges:

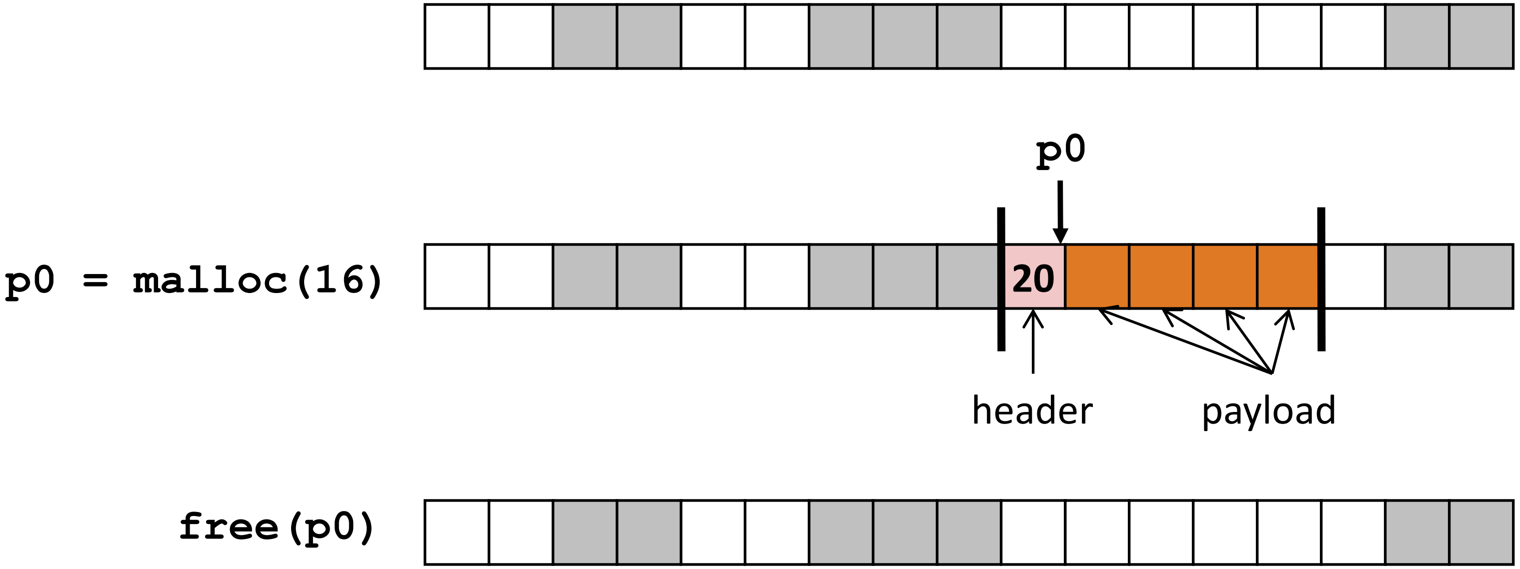
- How do we know how much memory to free given just a pointer?
- How do we keep track of the free blocks?
- How do we pick a block to use for allocation?
- What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
- How do we reinsert a freed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

Knowing How Much to Free

Why the preceding block?
Why not have the pointer point to the header?

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 (e.g., 4 byte integer) for every allocated block



Challenges

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

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

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?

Challenges

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

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

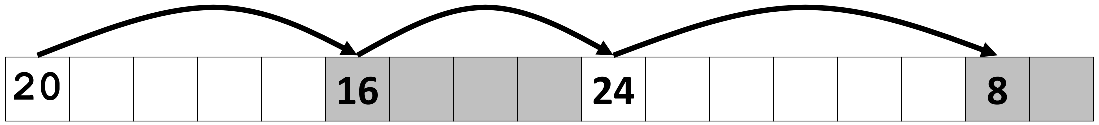
Goal: maximize throughput and peak memory utilization

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



Method 1: Implicit List

For each block we need both size and allocation status

- Could store this information in two words (one word for the size; one word for the status), but this is needlessly 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

Address (Hex)	Address (Binary
00	0000 0000
04	0000 0100
08	0000 1000
0C	0000 1100
10	0001 0000
14	0001 0100
18	0001 1000
1C	0001 1100
20	0010 0000

Method 1: Implicit List

For each block we need both size and allocation status

- Could store this information in two words (one word for the size; one word for the status), but this is needlessly 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

Address (Hex)	Address (Binary
00	0000 0000
04	0000 0100
08	0000 1000
0C	0000 1100
10	0001 0000
14	0001 0100
18	0001 1000
1C	0001 1100
20	0010 0000

Method 1: Implicit List

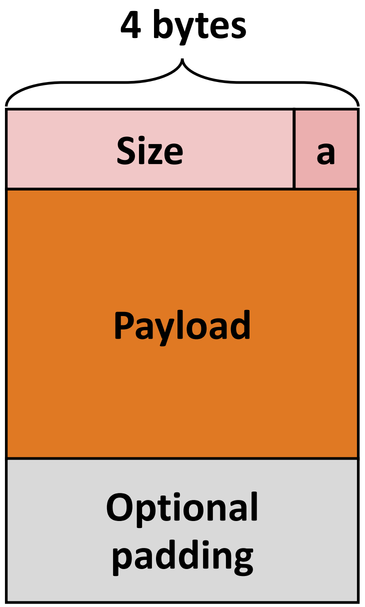
For each block we need both size and allocation status

- Could store this information in two words (one word for the size; one word for the status), but this is needlessly 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

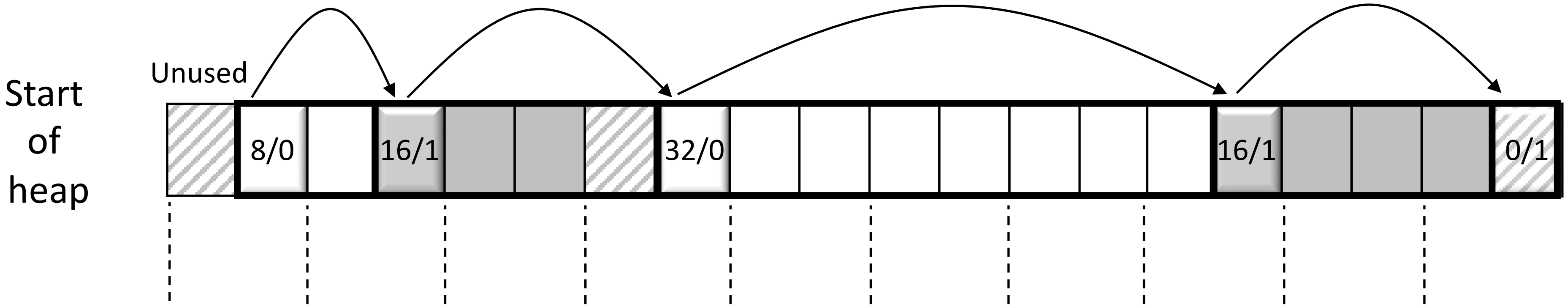


a = 1: Allocated block
a = 0: Free block

Size: block size

Payload: application data (allocated blocks only)

Detailed Implicit Free List Example



¦ 4-byte words
¦ 8-byte aligned

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

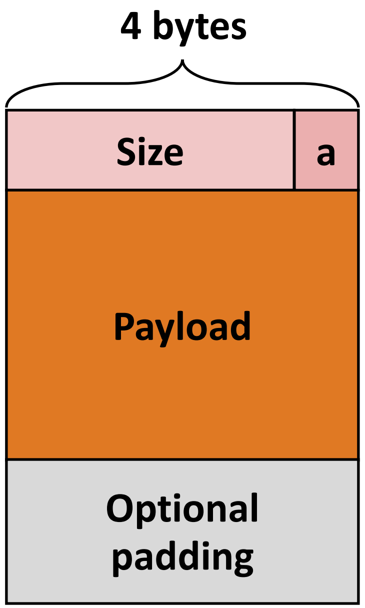
Practice with Block Headers

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.

Header
+ Payload
+ Padding

Size and Status

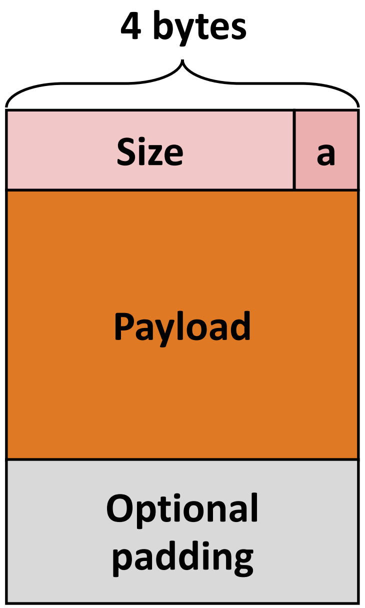
Request	Block size (decimal)	Block header (hex)
`malloc(1)`		
`malloc(5)`		
`malloc(12)`		



Practice with Block Headers

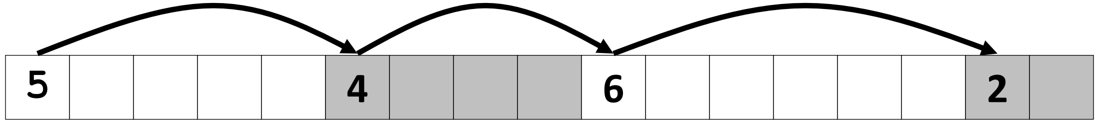
Determine the block sizes and header values that would result from the following sequence of malloc requests. Assume that the allocator uses an implicit list implementation with the block format just described and maintains 8-byte alignment.

	Header + Payload + Padding	Size and Status
Request	**Block size (decimal)**	**Block header (hex)**
`malloc(1)`	8	00 00 00 09
`malloc(5)`	16	00 00 00 11
`malloc(12)`	16	00 00 00 11

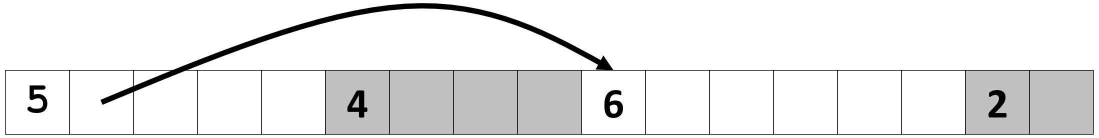


Keeping Track of Free Blocks

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

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

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

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?

Challenges

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

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

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

```
p = start;
while ((p < end) && ((*p & 1) || (*p  <= len)))
  p = p + (*p & -2);
```

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

- Can take linear time in total number of blocks (allocated and free)
- In practice it can cause fragmentation at the beginning

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

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

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

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?

Challenges

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

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

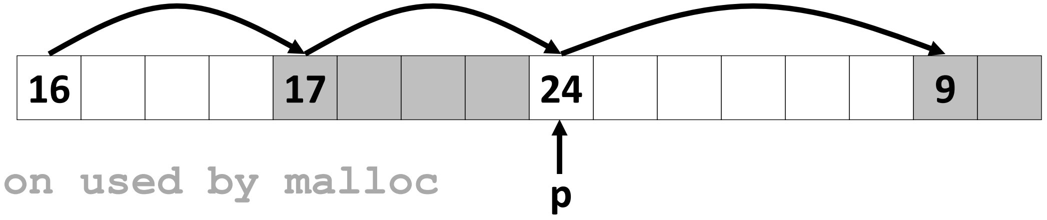
Goal: maximize throughput and peak memory utilization

Implementation challenges:

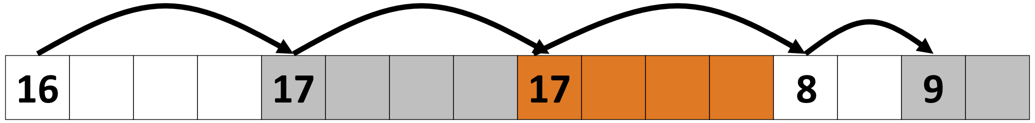
- How do we know how much memory to free given just a pointer?
- How do we keep track of the free blocks?
- How do we pick a block to use for allocation?
- What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
- How do we reinsert a freed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



```
// utility function used by malloc
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

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

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

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?

Challenges

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

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

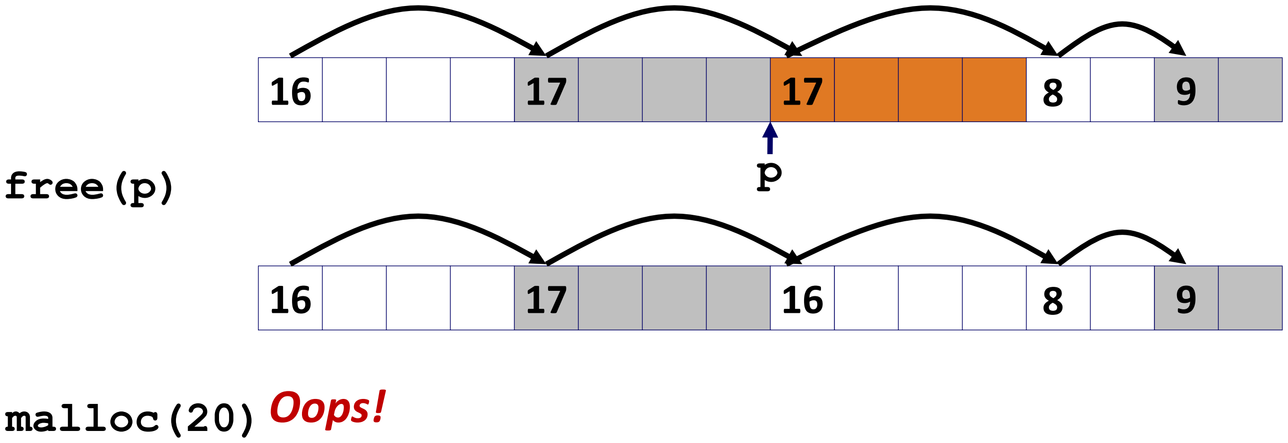
Goal: maximize throughput and peak memory utilization

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

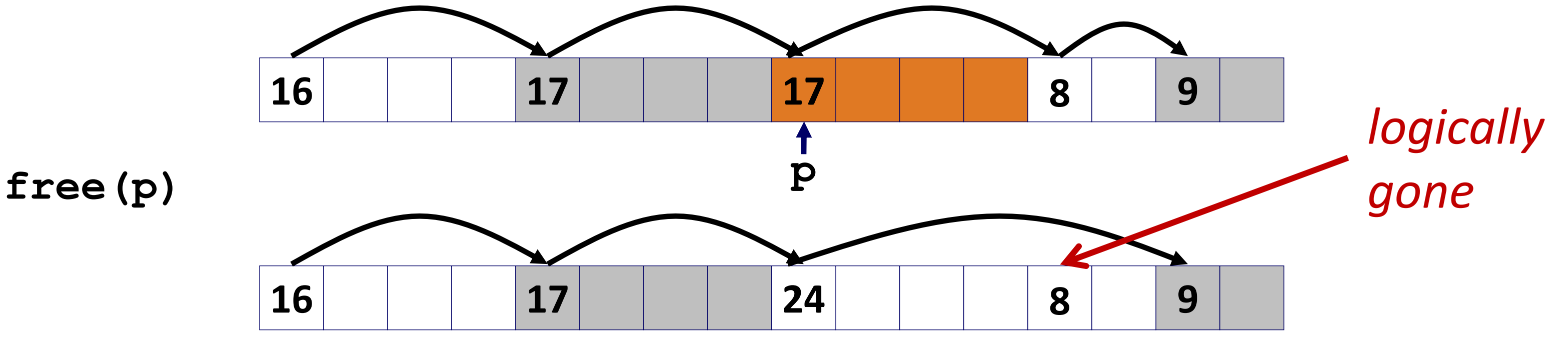
- Simplest implementation:
  - Need only clear the “allocated” flag
    - `void free_block(ptr p) { *p = *p & -2 }`
  - But can lead to “false fragmentation”



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

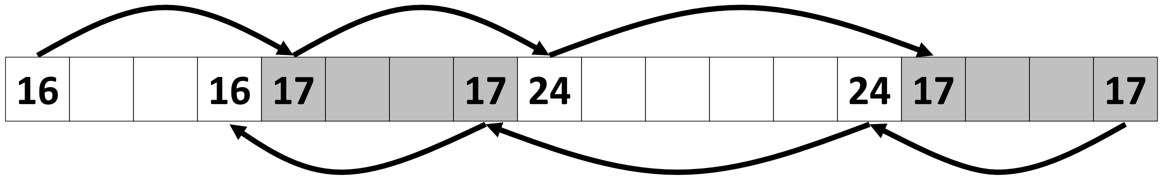


```
void free_block(ptr p) {
     *p = *p & -2;           // clear allocated flag

     ptr next = p + *p;      // find next block

     if ((*next & 1) == 0)   // add to this block if
       *p = *p + *next;      //    not allocated
}
```

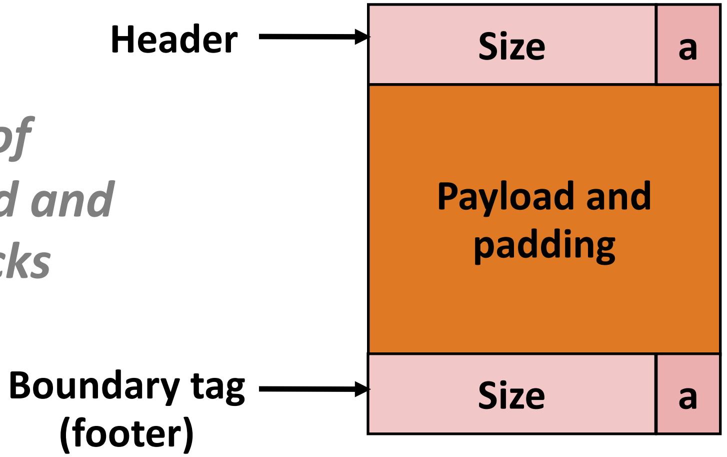
But how do we coalesce with *previous* block?

Implicit List: Bidirectional Coalescing

- Boundary
  - Replicate size/allocated word at "bottom" (end) of free blocks
  - Allows us to traverse the "list" backwards, but requires extra space



Format of allocated and free blocks



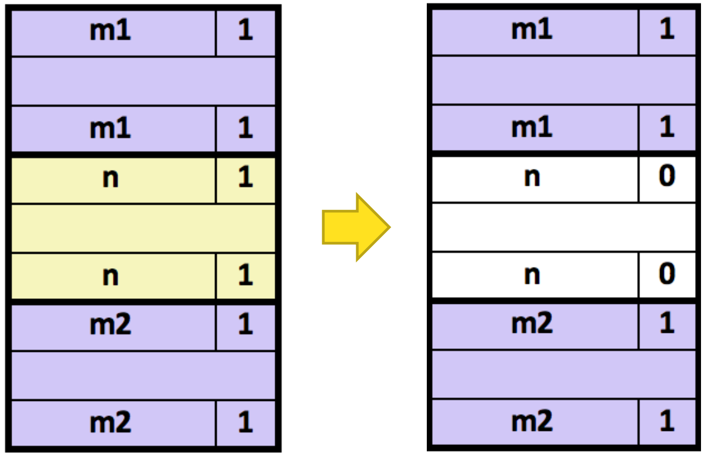
a = 1: Allocated block
a = 0: Free block

Size: Total block size

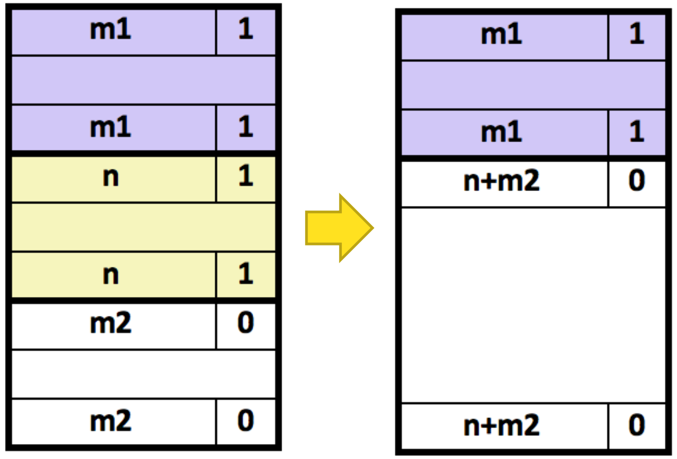
Payload: Application data
(allocated blocks only)

Constant-Time Coalescing

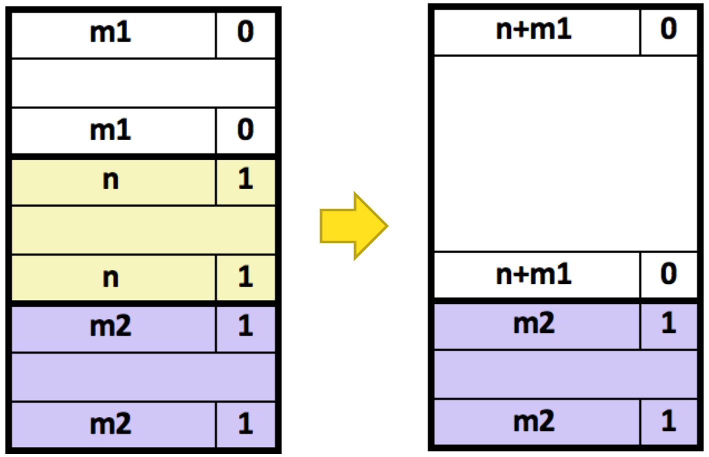
Case 1: Blocks above and below allocated



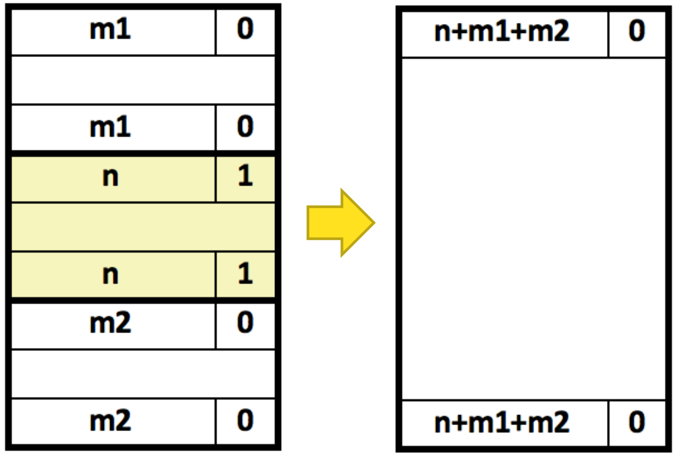
Case 2: Block above allocated, block below free



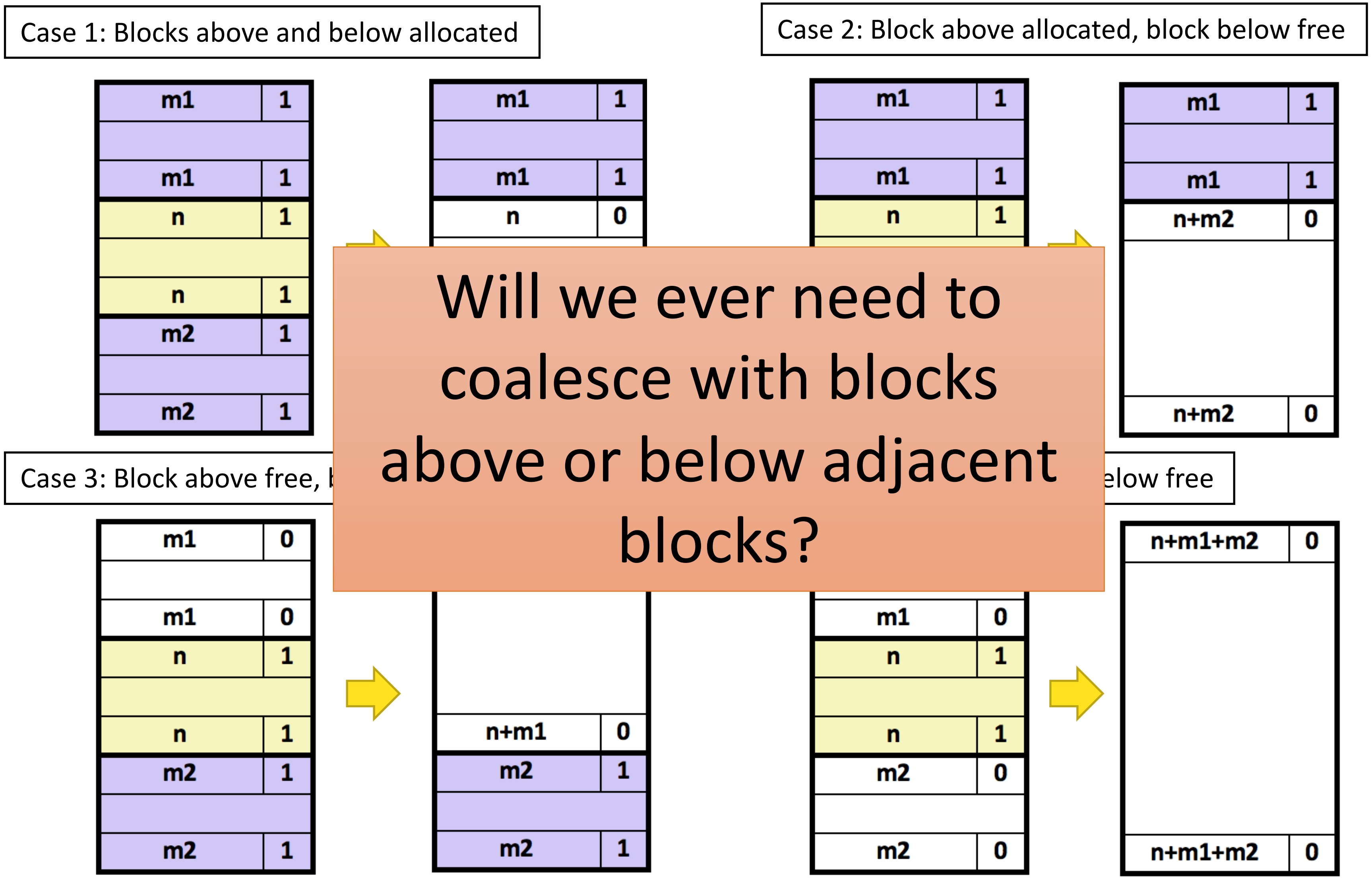
Case 3: Block above free, block below allocated



Case 4: Blocks above and below free



Constant-Time Coalescing



Case 1: Blocks above and below allocated

Case 2: Block above allocated, block below free

Case 3: Block above free, b

elow free

Will we ever need to coalesce with blocks above or below adjacent blocks?

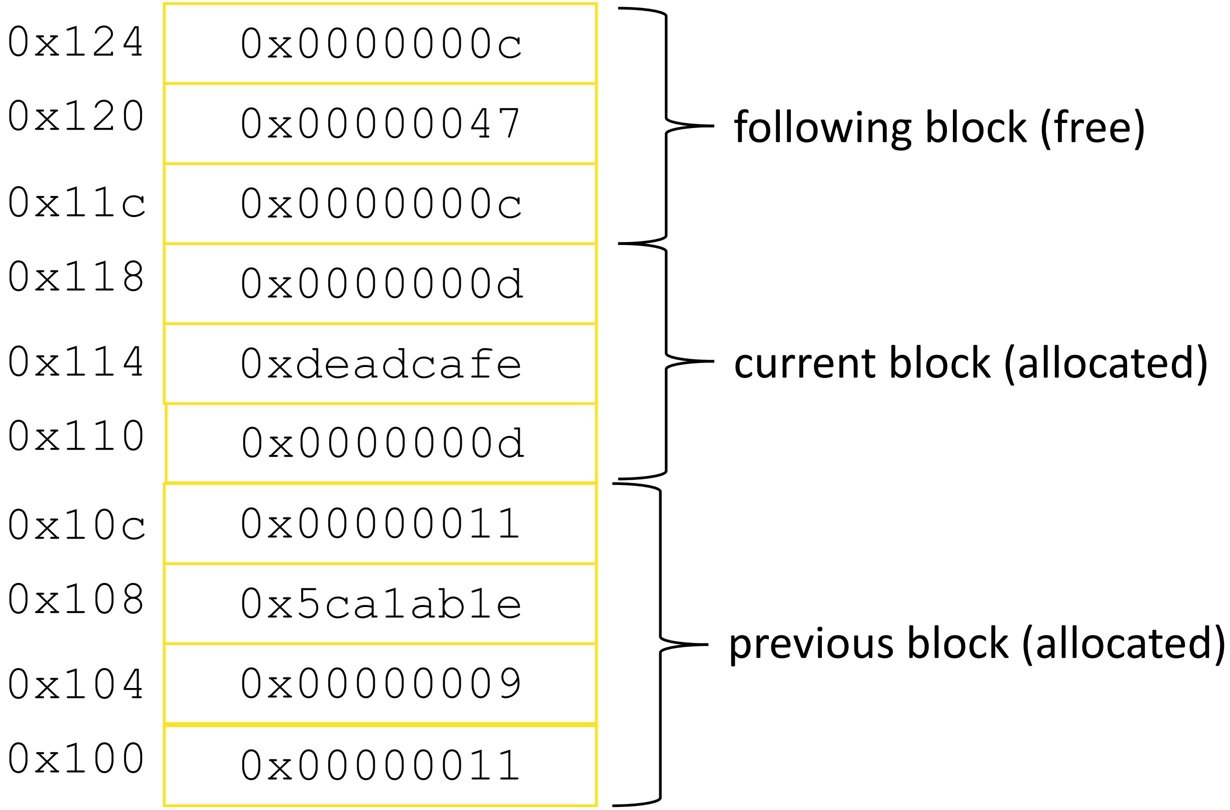
Practice with Coalescing

Assume the current state of the heap is shown. What is the state of the heap after a call “`free(0x114)`” is executed?

0x124	0x0000000c
0x120	0x00000047
0x11c	0x0000000c
0x118	0x0000000d
0x114	0xdeadcafe
0x110	0x0000000d
0x10c	0x00000011
0x108	0x5ca1ab1e
0x104	0x00000009
0x100	0x00000011

Practice with Coalescing

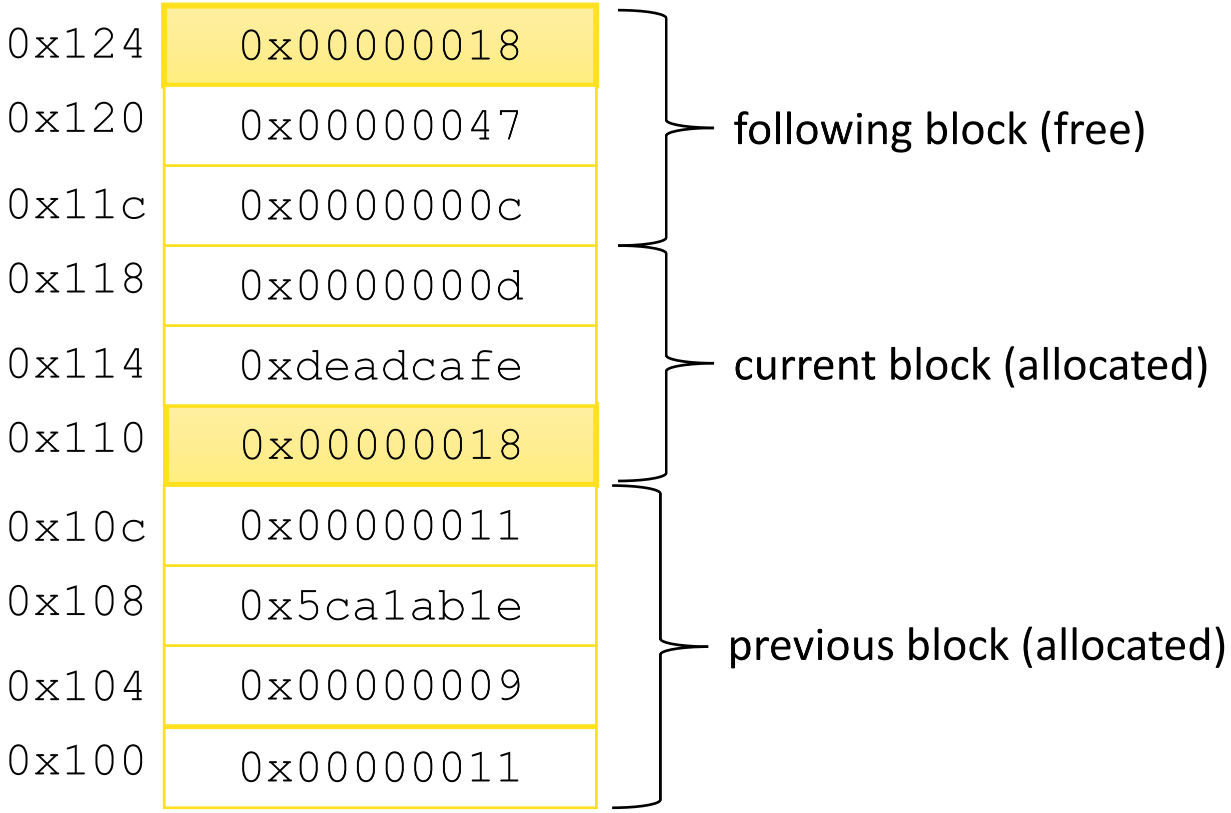
Assume the current state of the heap is shown. What is the state of the heap after a call “`free(0x114)`” is executed?



0x124	0x0000000c	following block (free)
0x120	0x00000047	
0x11c	0x0000000c	
0x118	0x0000000d	current block (allocated)
0x114	0xdeadcafe	
0x110	0x0000000d	
0x10c	0x00000011	previous block (allocated)
0x108	0x5ca1ab1e	
0x104	0x00000009	
0x100	0x00000011	

Practice with Coalescing

Assume the current state of the heap is shown. What is the state of the heap after a call “`free(0x114)`” is executed?



0x124	0x00000018	following block (free)
0x120	0x00000047	
0x11c	0x0000000c	
0x118	0x0000000d	current block (allocated)
0x114	0xdeadcafe	
0x110	0x00000018	
0x10c	0x00000011	previous block (allocated)
0x108	0x5ca1ab1e	
0x104	0x00000009	
0x100	0x00000011	

Challenges

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

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

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?

Summary of Key Allocator Policies

Storage policy:

- What data structure will you use to keep track of the free blocks?

Placement policy:

- First-fit, next-fit, best-fit, etc.
- Trades off lower throughput for less 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:

- 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

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)

Tools for Dealing With Memory Bugs

- Debugger: gdb
  - Good for finding bad pointer dereferences
  - Hard to detect the other memory bugs

- Heap consistency checker (e.g., mcheck)
  - Usually run silently, printing message only on error
  - Can be used to detect overreads, double-free
  - glibc malloc contains checking code
    - setenv MALLOC_CHECK_ 3

- Binary translator: valgrind
  - Powerful debugging and analysis technique
  - Rewrites text section of executable object file
  - Checks each individual reference at runtime
    - Bad pointers, overwrites, refs outside of allocated block

Memory Bugs Persist…

Apr 2020

Feb 2020

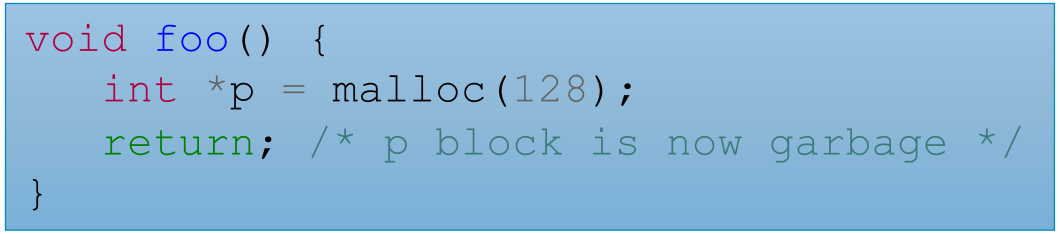
Mar 2020

Oct 2019

Implicit Allocators

Implicit Allocators: Garbage Collection

- Garbage collection: automatic reclamation of heap-allocated storage—application never has to free



```
void foo() {
   int *p = malloc(128);
   return; /* p block is now garbage */
}
```

- Common in many dynamic languages:
  - Python, Java, Ruby, Perl, ML, Lisp, Mathematica

- Variants (“conservative” garbage collectors) exist for C and C++
  - However, cannot necessarily collect all garbage

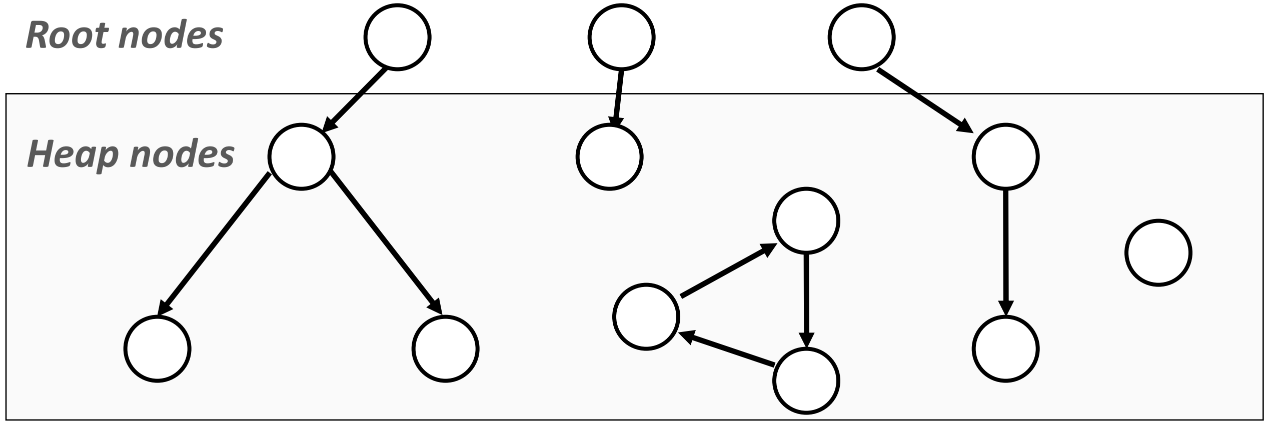
Garbage Collection

- How does the memory manager know when memory can be freed?
  - In general we cannot know what is going to be used in the future since it depends on conditionals
  - But we can tell certain blocks cannot be used if there are no pointers to them

- Must make certain assumptions about pointers
  - Memory manager can distinguish pointers from non-pointers
  - All pointers point to the start of a block
  - Cannot hide pointers (e.g., by coercing them to an long, and then back again)

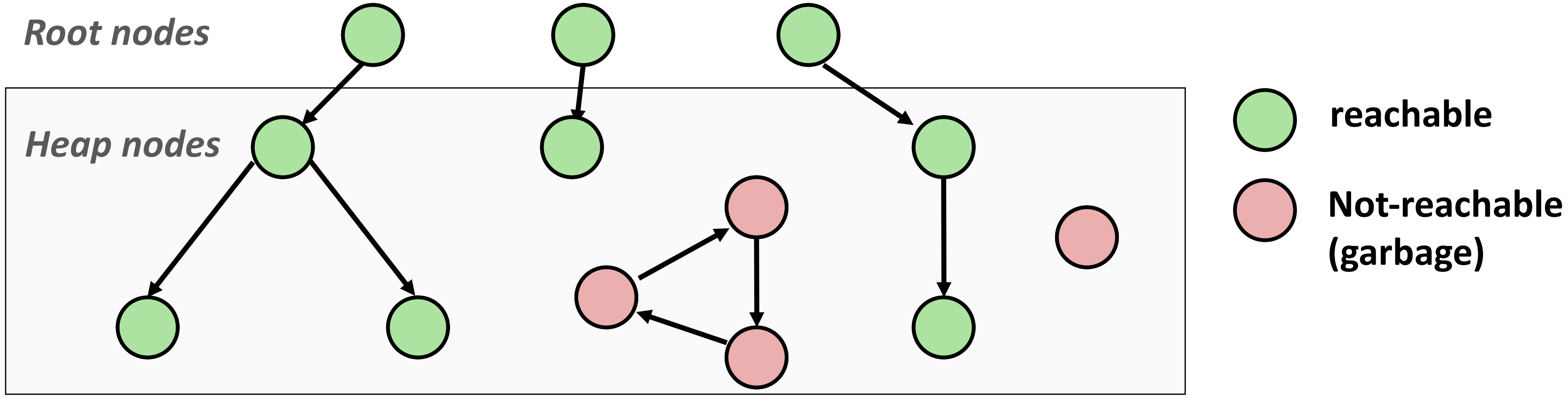
Memory as a Graph

- We view memory as a directed graph
  - Each block is a node in the graph (called a heap node)
  - Extra root nodes correspond to locations not in the heap that contain pointers into the heap
    - registers, local stack variables, or global variables
  - Each pointer is an edge in the graph



Memory as a Graph

- A node <u>is reachable</u> if there exists a directed path from some root node to the node
- Unreachable heap nodes are garbage
  - they can never again be used by the application
  - they should be freed ("garbage collected")



Garbage Collection

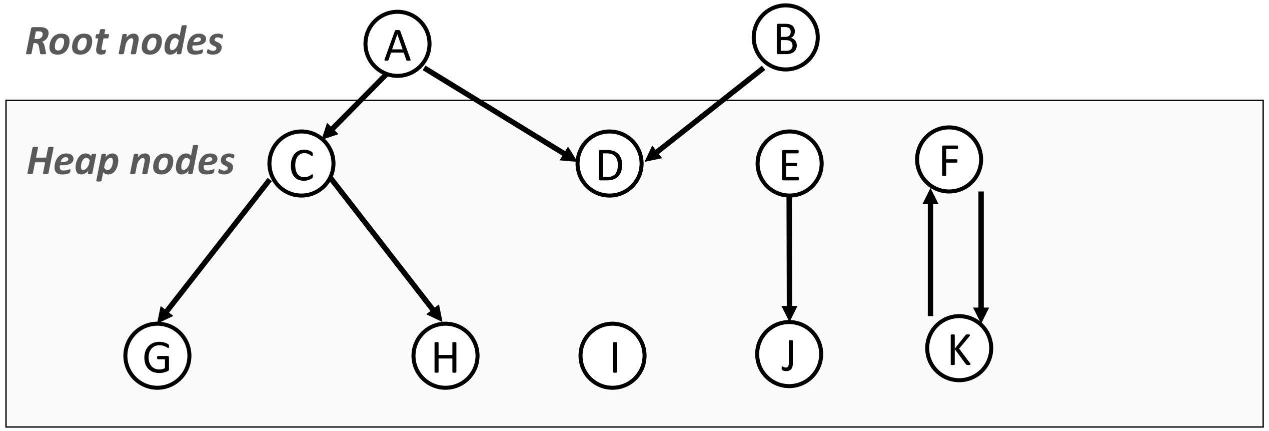
- The role of a garbage collector is
  - to maintain some representation of the reachability graph
  - to reclaim the unreachable nodes by freeing them
    - this can happen periodically, or collector can run in parallel with application)

- Languages that maintain tight control over how applications create and use pointers (e.g., Java, Python, OCaml) can maintain an exact representation of the graph

- Garbage collectors for languages like C/C++ will be conservative

Practice Garbage Collection

- Consider the following graph representation of memory. Which nodes correspond to blocks that should be freed by the garbage collector?



Classical GC Algorithms

- Mark-and-sweep collection (McCarthy, 1960)
    - Does not move blocks (unless you also “compact”)

- Reference counting (Collins, 1960)
    - Does not move blocks

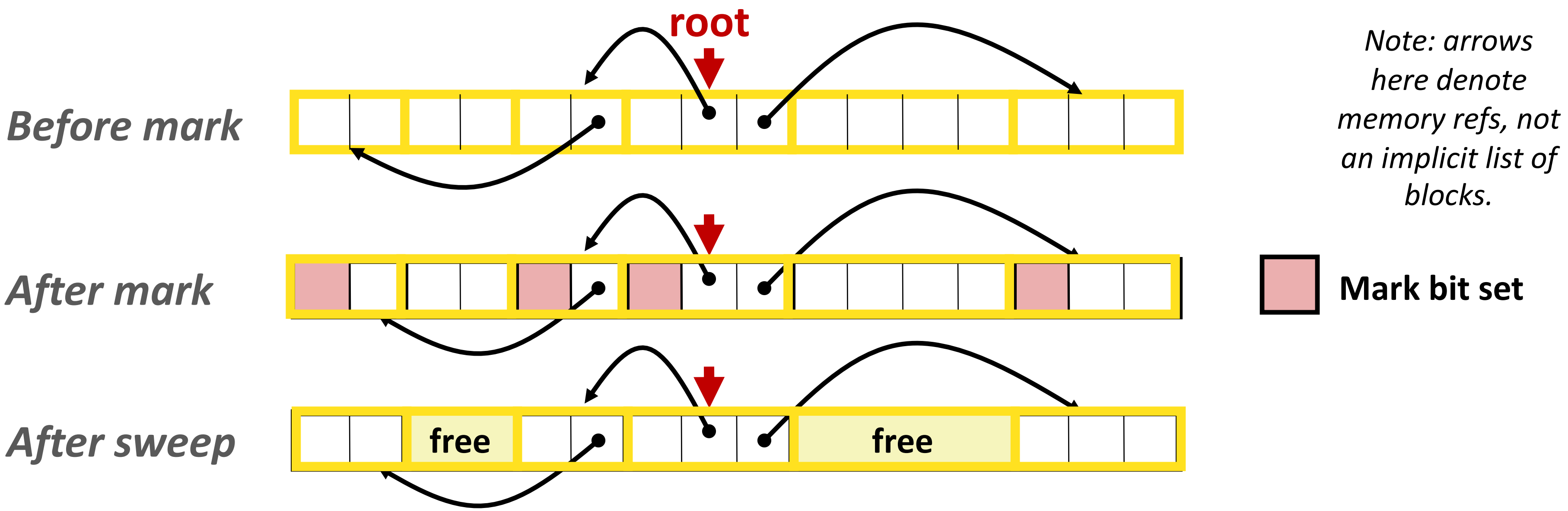
- Copying collection (Minsky, 1963)
    - Moves blocks

- Generational Collectors (Lieberman and Hewitt, 1983)
    - Collection based on lifetimes
        - Most allocations become garbage very soon
        - So focus reclamation work on zones of memory recently allocated

Mark and Sweep Collector

Each block header has an extra mark bit (can use spare low-order bits)

- Two phase protocol
  1. Mark: Start at roots and set mark bit on each reachable block
  2. Sweep: Scan all blocks and free blocks that are not marked



Note: arrows here denote memory refs, not an implicit list of blocks.

Mark and Sweep Collector

Mark using depth-first traversal of the memory graph

```
ptr mark(ptr p) {
   if (!is_ptr(p)) return;         // do nothing if not pointer
   if (markBitSet(p)) return;      // check if already marked
   setMarkBit(p);                  // set the mark bit
   for (i=0; i < length(p); i++)   // call mark on all words
     mark(p[i]);                   //   in the block
   return;
}
```

Sweep using lengths to find next block

```
ptr sweep(ptr p, ptr end) {
   while (p < end) {
      if markBitSet(p)
         clearMarkBit();
      else if (allocateBitSet(p))
         free(p);
      p += length(p);
}
```