PURDUE
UNIVERSITY®

CS 25200: Systems Programming

Lecture 6: Memory Management and malloc()

Prof. Turkstra

Lecture 06

- brk() and sbrk()
- Free lists
- Fragmentation
- malloc() internals
- Segregated free lists
- Coalescing
- Fenceposts

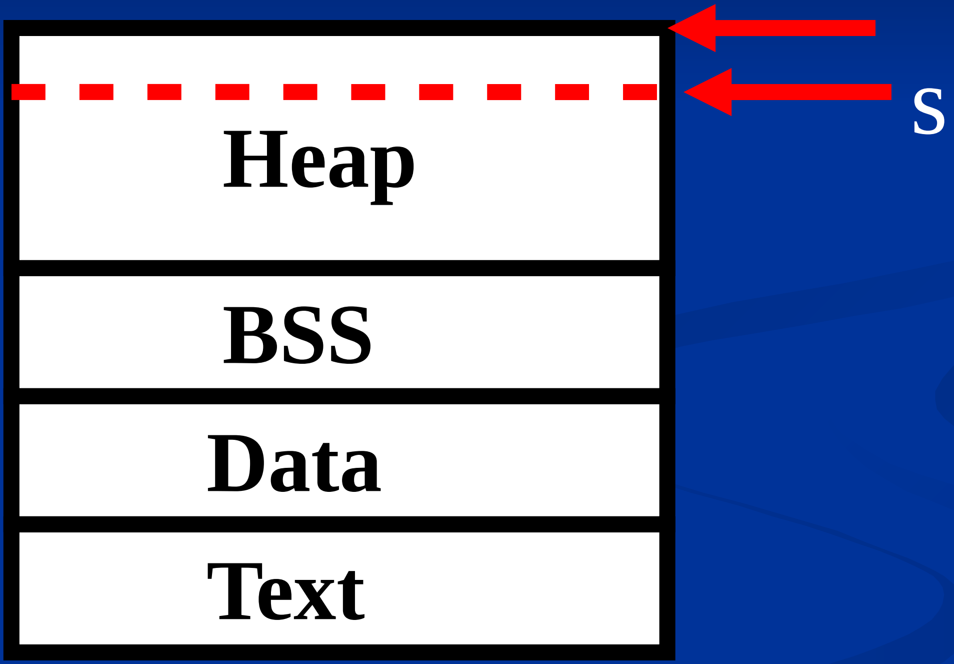
The Heap

- From the kernel’s standpoint, the heap is a single contiguous region of memory that grows linearly
- To request more memory, a userland process invokes the brk() system call
  - libc provides two wrapped versions of brk(): sbrk() and brk()

brk() vs sbrk()

- `int brk(void *addr);`
  Set the end of the data segment to the value specified by addr when *reasonable*
  - Cannot exceed maximum data size
- `void *sbrk(intptr_t increment);`
  Increment the program's data space by increment bytes.
  - Returns the previous program break or (void *) -1
  - sbrk(0) returns the current location
  - Argument can be negative

```
s = sbrk(n);
```



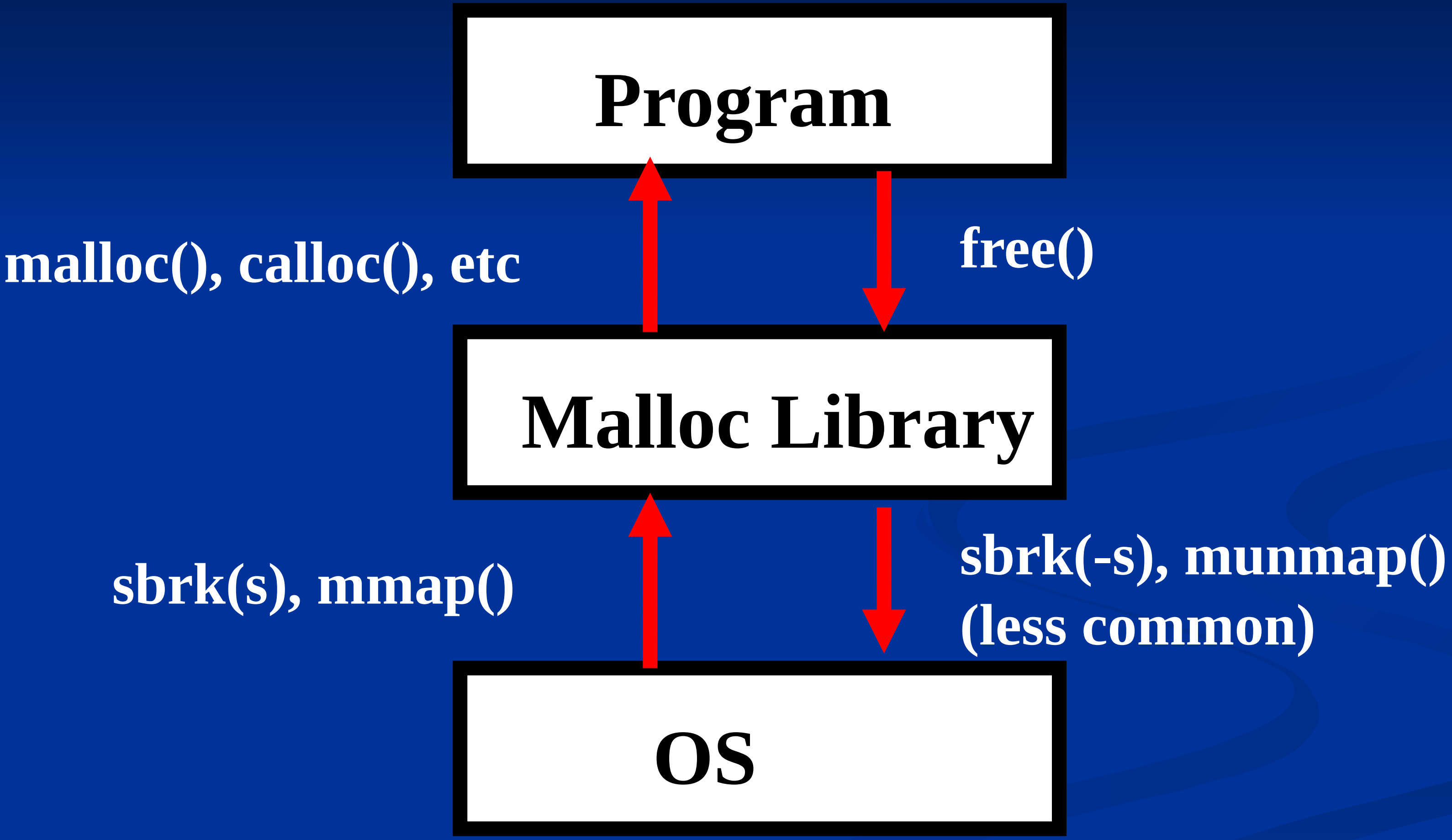
malloc

- The portable and “comfortable” way to allocate memory in C is by using the provided memory allocation package
  - malloc(), calloc(), realloc(), and free()
- The program break is managed for you, internally
- Programmer explicitly invokes the above functions

How?

- Memory is requested from the OS in large “chunks” (e.g., 64KiB)
- These chunks are then managed internally
  - Added to a free list
  - Subsequent requests are satisfied from the free list when possible
- Decreases number of times the OS must be invoked (via system call)

Memory management



Program

malloc(), calloc(), etc

free()

Malloc Library

sbrk(s), mmap()

sbrk(-s), munmap()
(less common)

OS

Implementation

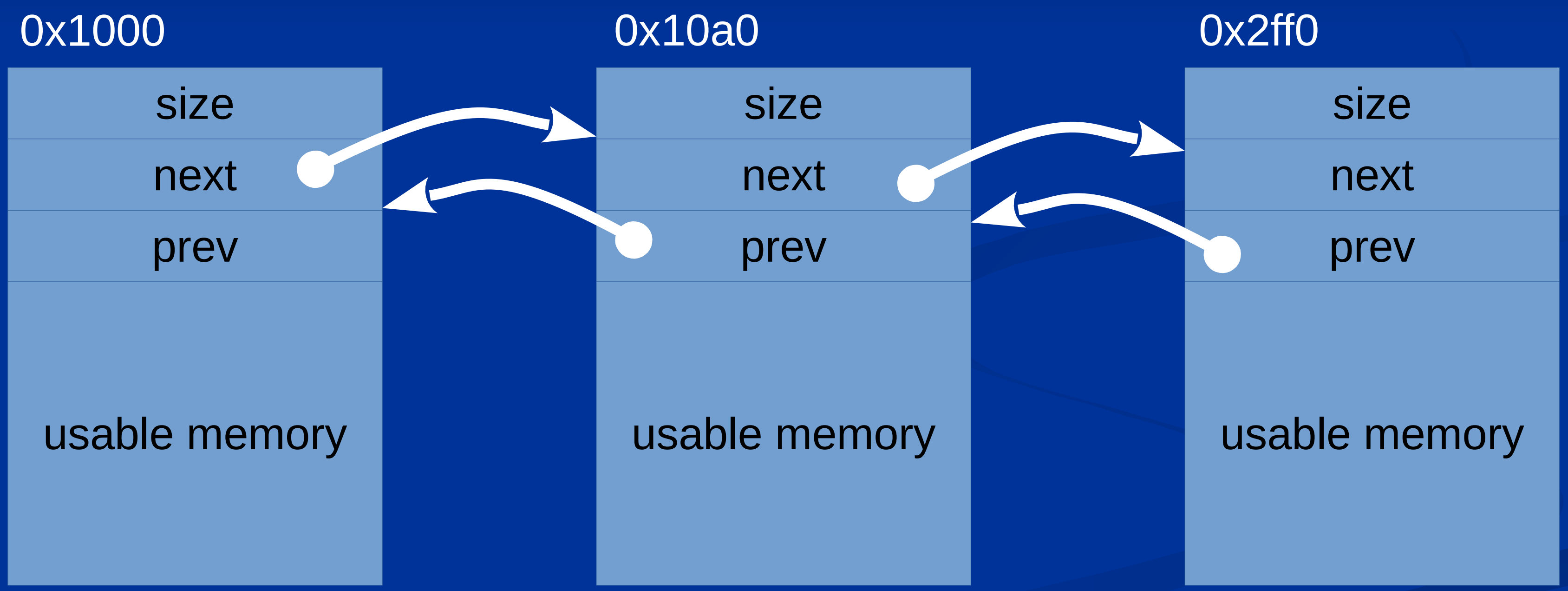
- There are many different ways to implement allocators
- Some useful data structures…
  - Single free list
  - Segregated free lists
  - Cartesian trees
  - Boundary tags

Single free list

- …or sequential fit
- Structure that is sequentially searched to find the needed size
  - First-fit
  - Best-fit
  - Next-fit
  - Worst-fit
- Can be a singly-linked list, doubly-linked list, tree, etc

Where is the list stored?

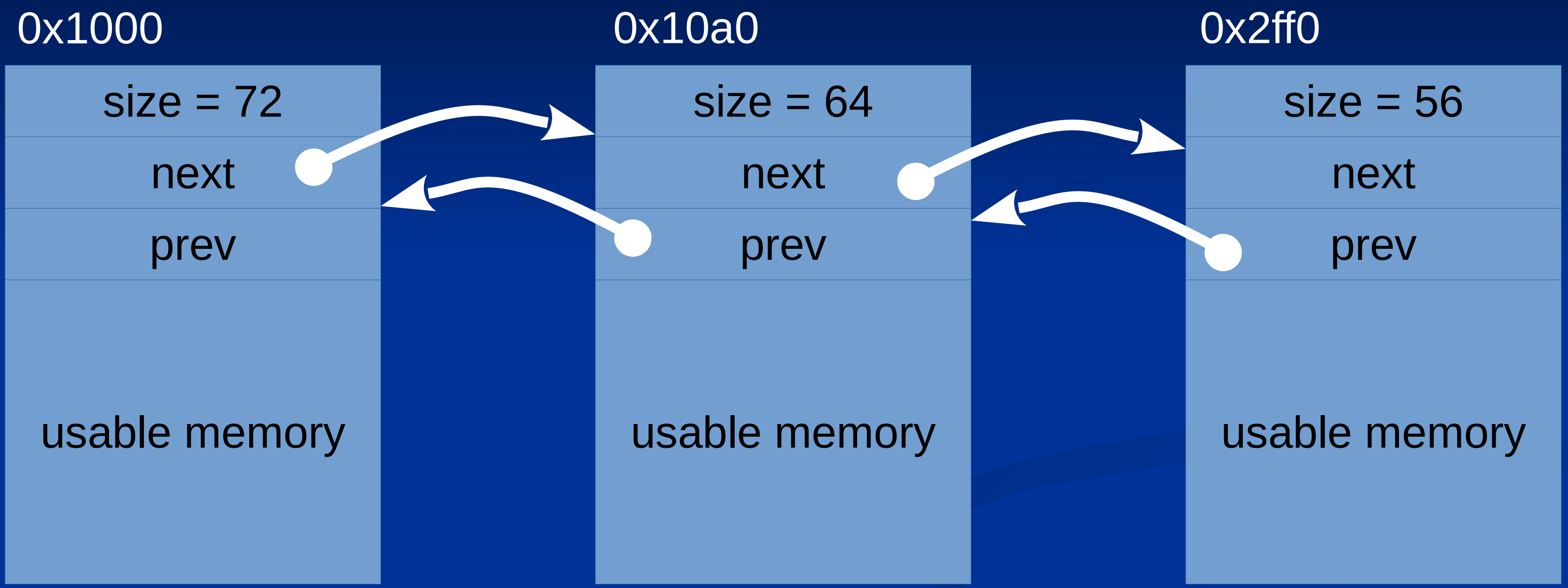
- In the free blocks!
- Each block has a header



Fragmentation

- Memory that is too small to be usefully allocated
  - External: visible to allocator
  - Internal: visible to requester
- Want to minimize fragmentation

External fragmentation



0x1000

size = 72

next

prev

usable memory

0x10a0

size = 64

next

prev

usable memory

0x2ff0

size = 56

next

prev

usable memory

```
p = malloc(100);
```

- 104 + 16 bytes (header) = 120 bytes total

p = ??

External fragmentation

- Can be calculated:
  Ext_Frag = 100 * (1 – size(largest_free_block) / sum(free_mem))
- Previous example:
  100 * (1 – 72 / (72 + 64 + 56)) = 62.5%
- Only one block? 0%

Comparison

- Fragmentation depends on the algorithm and the workload
- Best fit tends to leave some very large holes and some very small holes
  - Can’t use small holes easily
  - Computationally more expensive
- First fit tends to leave “average” sized holes
  - Also faster than best fit
- Next fit often used in practice
  - Prevents accumulation of small chunks at the beginning

Mechanics

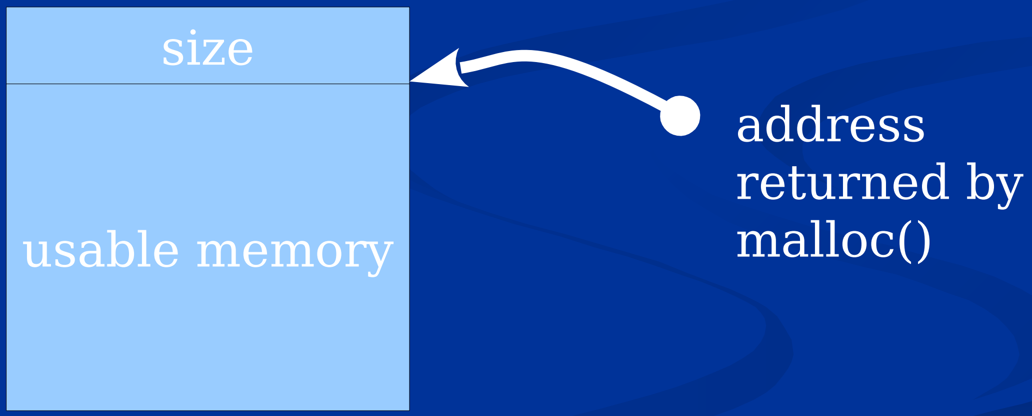
- malloc(): search the free list for the appropriate size
  - Found? Split if necessary (add the remainder to free list) and return the block
  - Not found? Request more memory (chunk) from the OS, add to free list, repeat
    - Usually done in “large” increments (e.g., 4KiB)

free()

- Coalesce if possible
  - If not, insert into list
- How does free() know the size of the memory chunk passed to it?

Header

- Remember the free list? malloc()’d chunks also have a header
  - But no list



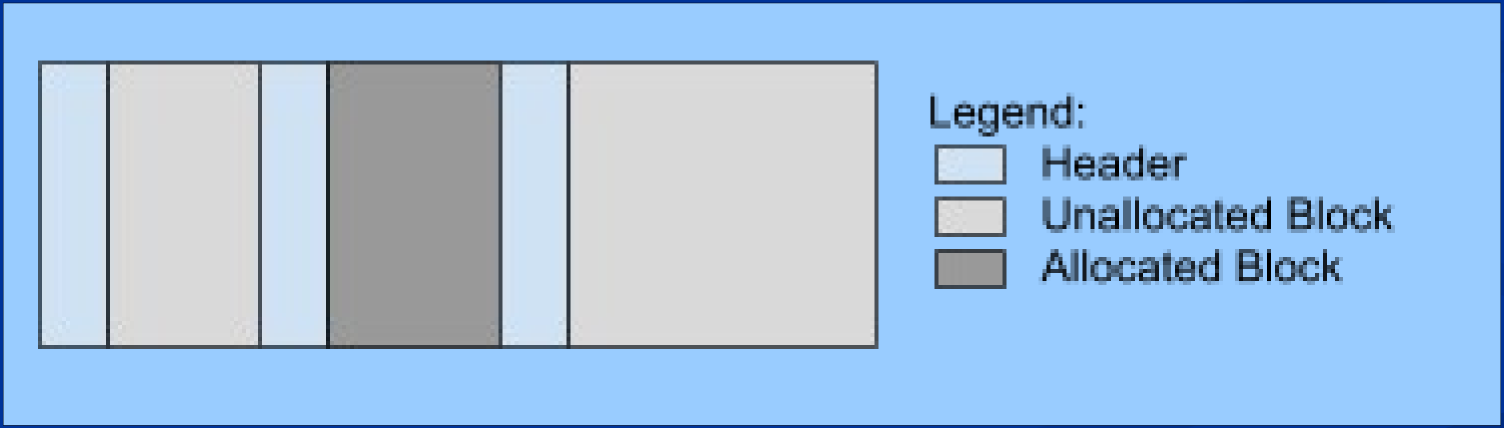
  - malloc() returns a pointer that points after the header

Headers

- Easy to get the next block header

```
(header *) ((char *) block_addr +
                ALLOC_HEADER_SIZE + size)
```

- What about the previous block?



Boundary tag

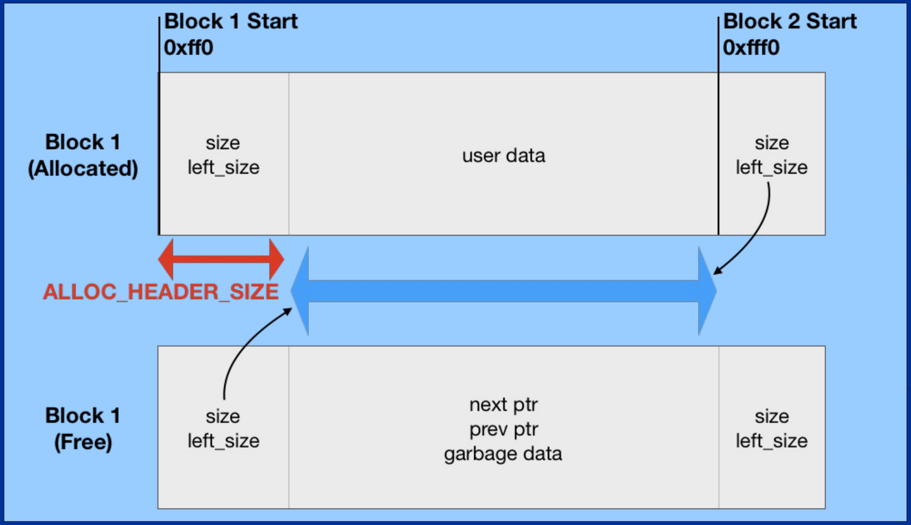
- Track the size of the "left" block
  - …in the current block's header
- Previous block header is now:

```
(header *) ((char *) block_addr – left_size –
                ALLOC_HEADER_SIZE)
```

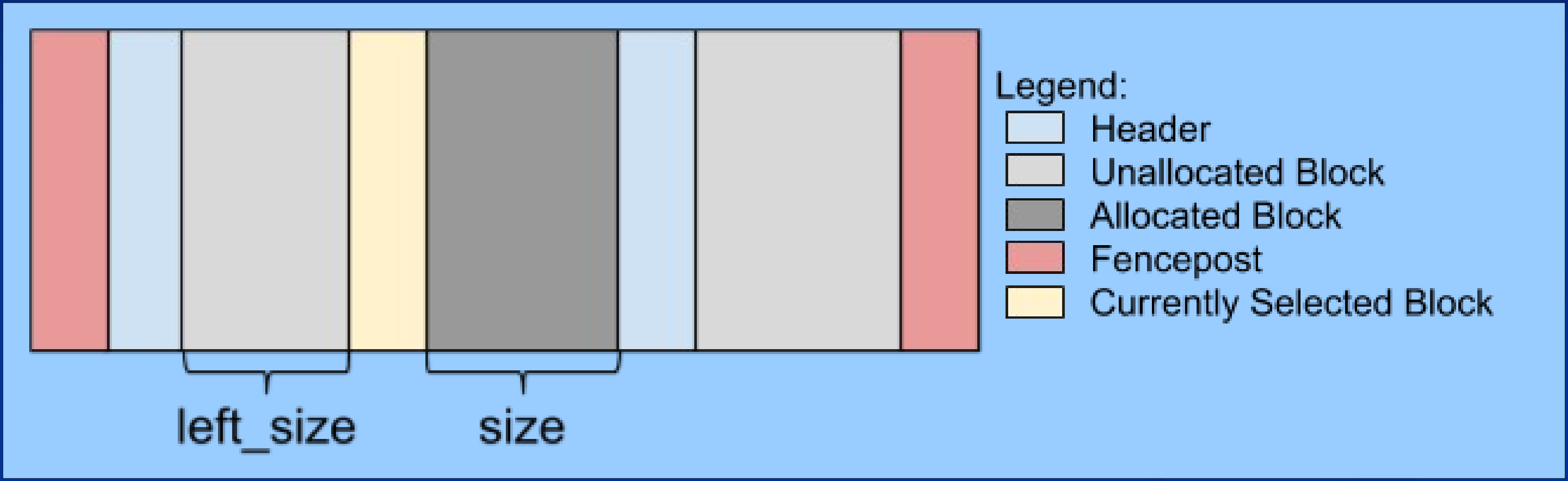
Constant time coalescing

  - Donald Knuth

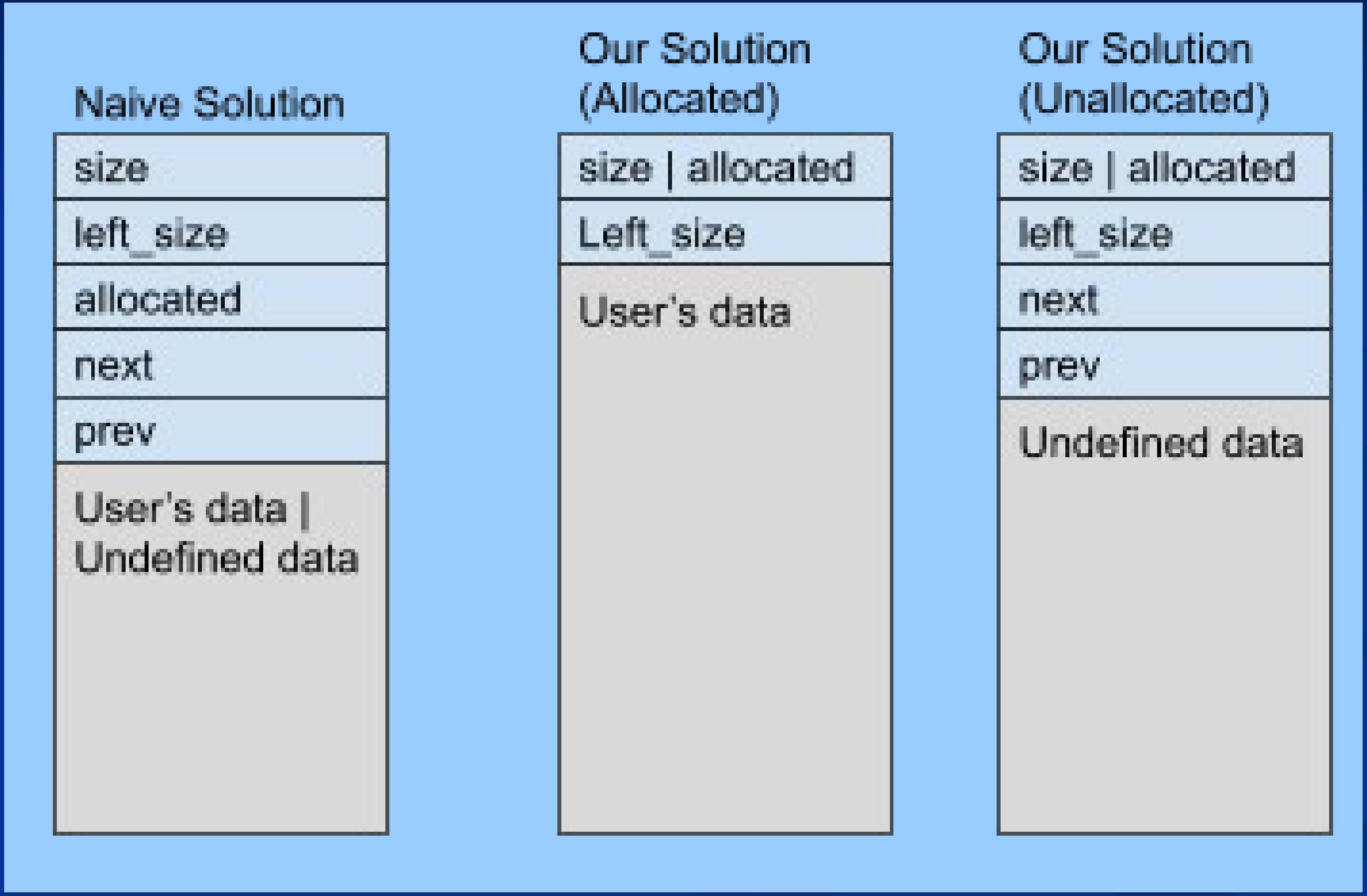
- Note that both size and left_size include the next and previous pointers



  - This is the maximum possible usable memory when allocated


left_size
size
Legend:
Header
Unallocated Block
Allocated Block
Fencepost
Currently Selected Block

How to tell allocation status?



Naive Solution

- size
- left_size
- allocated
- next
- prev
- User's data | Undefined data

Our Solution (Allocated)

- size | allocated
- Left_size
- User's data

Our Solution (Unallocated)

- size | allocated
- left_size
- next
- prev
- Undefined data

Bitwise operations!

- How do you set a bit?

```
size |= 0x1
```

- How do you clear them?

```
size &= ~0b111
```

Header

```
typedef struct header {
  size_t size;
  size_t left_size;
  union {
    struct {
      struct header *next;
      struct header *prev;
    };
    char *data;
  }
} header;
```

Alignment

- Many RISC architectures simply cannot handle an unaligned access
  - Sparc: SIGBUS
- x86 can, but it is slow
- Our malloc() should always return a MIN_ALLOCATION-aligned address
  - 8 bytes for now, could change!

Minimum allocated size

- Suppose the user requests 1 byte: malloc(sizeof(char));
  - Must be a multiple of MIN_ALLOCATION, so round to 8 bytes
  - Good enough?

Internal fragmentation

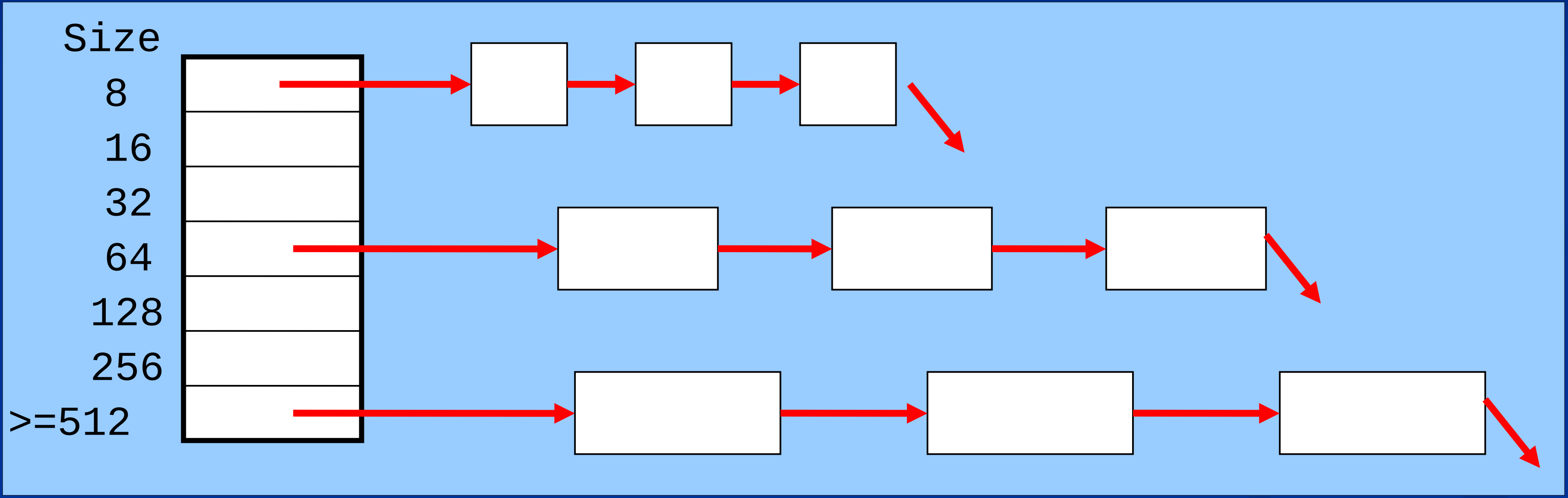
- Waste of memory due to allocator returning a larger block than requested

  Int_Frag = 100 * (1 – size(request) / sum(mem_allocated))

  - E.g., malloc(1) → 8 + roundup8(1) = 40 bytes for 1 byte of memory

Segregated free lists

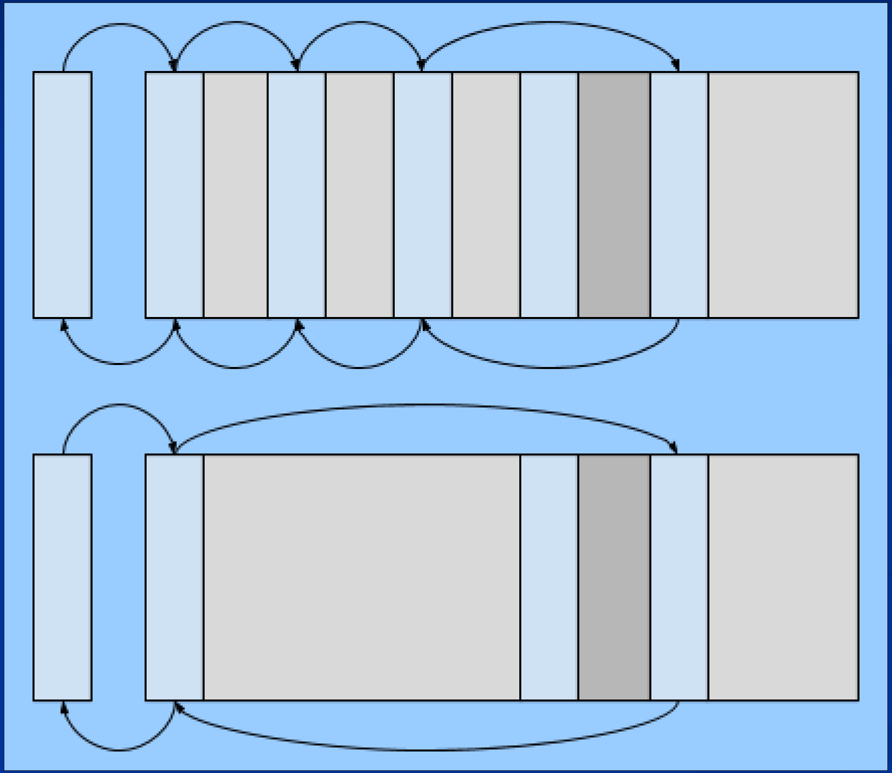
- Multiple free lists, one for each size



- Objects allocated from the free list of nearest size
- Empty or no size large enough? Get more memory from the OS and populate the appropriate free list

Coalescing

- Some implementations do not coalesce
  - Ours will
- Without a footer or boundary tag, requires traversal to find out if neighboring blocks are free



Segregated lists

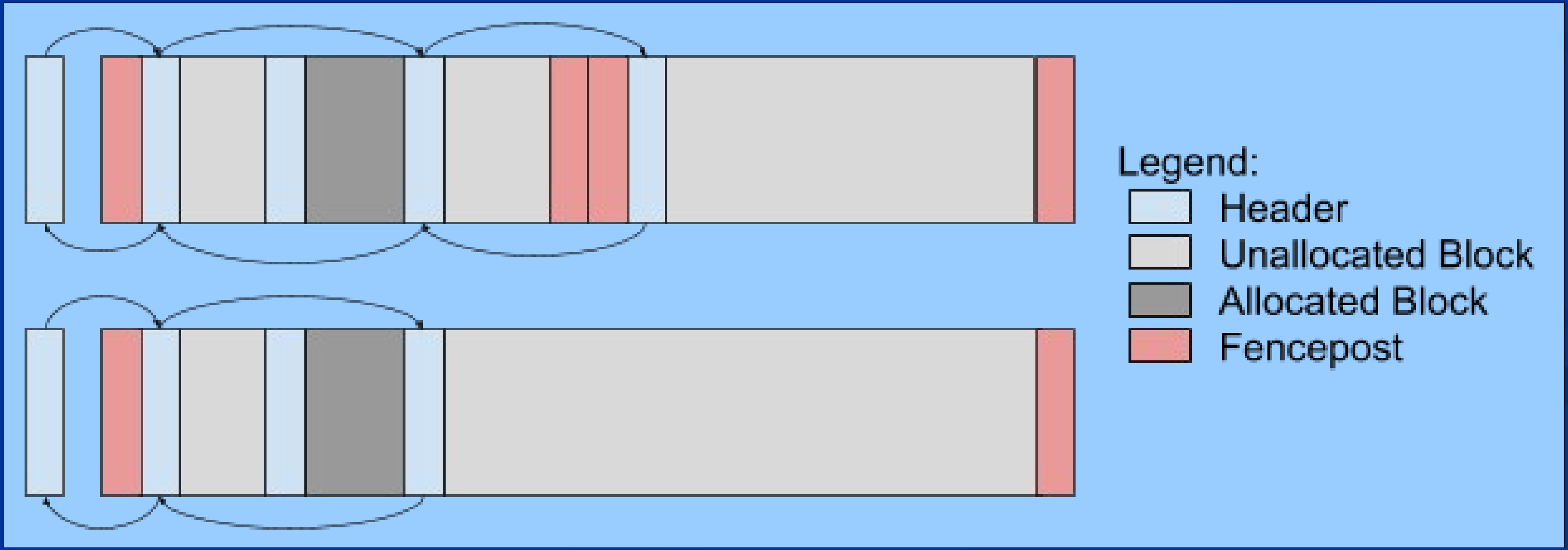
- With segregated lists,
  - Allocation is often O(1)
  - free() would be O(1) without coalescing
- Segregated free lists are fast
  - But use more memory
  - Even more memory if they don’t coalesce

Project 2

- You will implement a malloc library that can be used as a substitute for libc’s malloc
- Gain an understanding of malloc internals
- Also better understand memory errors – premature free, double free, wild free, etc

Fenceposts

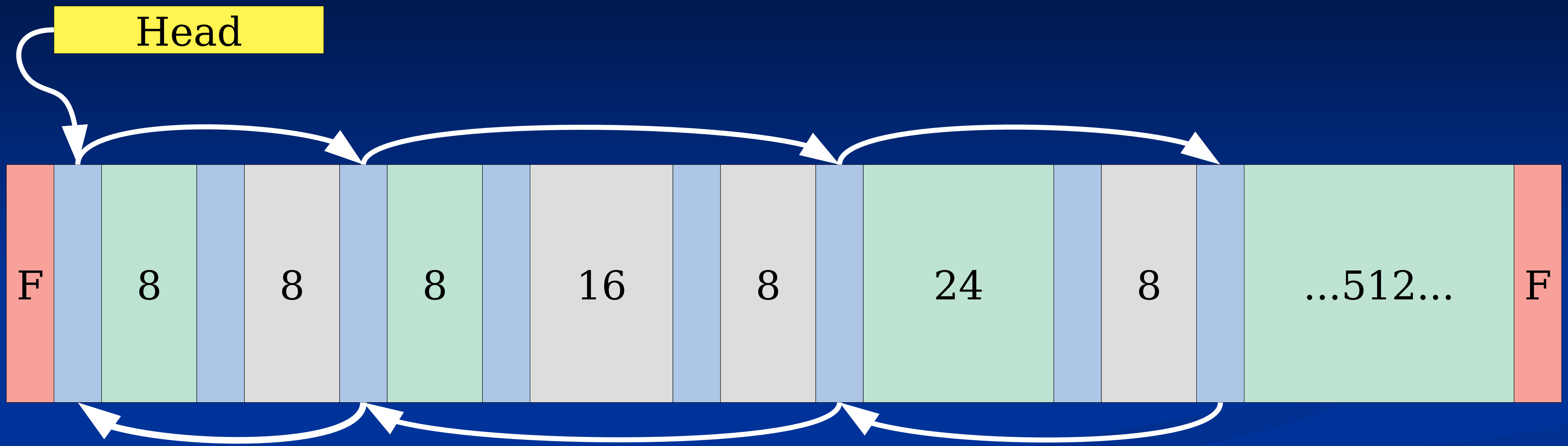
- Memory inside malloc is obtained a "chunk" at a time
- Chunks can also be coalesced
- How do we know when not to?



Obtaining chunks

- Round to nearest ARENA_SIZE
  - Don’t forget to include size of fenceposts and header in request
- Call sbrk()
  - Determine if the new request is contiguous with previous brk
    - Yes? Eliminate fencepost
    - No? Don’t
- Initialize fenceposts (set_fenceposts())
  - May have to fix left_size
- Initialize header
- Add to free list
  - Possibly coalesce

All together



Fencepost

Header

Footer

Free

Allocated

Questions?