Lecture 13: Dynamic Memory Allocation

Vivek Kumar

Computer Science and Engineering

IIIT Delhi

vivekk@iiitd.ac.in

Last Lecture

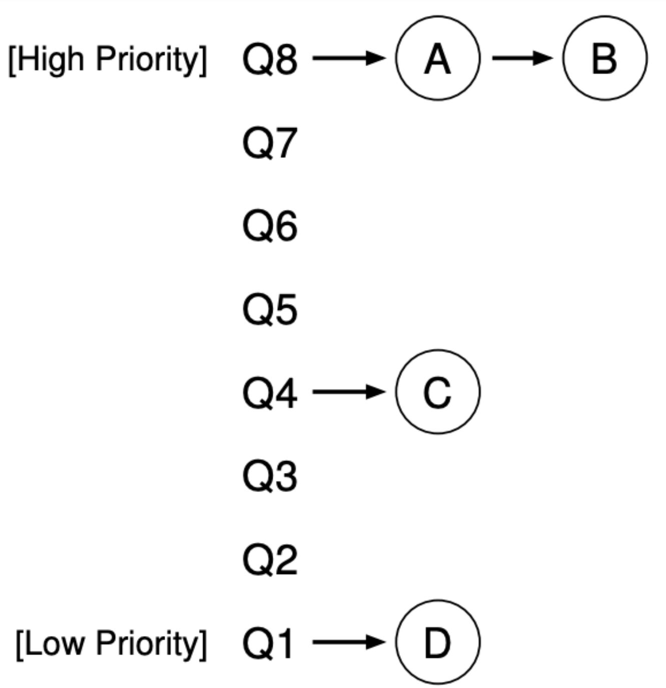


Figure 8.1: **MLFQ Example**

- Unfairness with IO jobs in RR queues
- Multi level feedback queue
- Completely fair scheduler
  - Variable scheduling latency depending on total number of processes
  - Minimum granularity for scheduling latency to reduce context switch overheads
  - Ready process having minimum vRuntime picked for running

Today's Class

- Dynamic memory allocations
- Fragmentation
- Allocation using implicit list
- Allocation using buddy allocator

Memory Allocations

- Static size, static allocation
  - Global and static variables
  - Linker allocates final addresses
  - Executable stores these allocated addresses

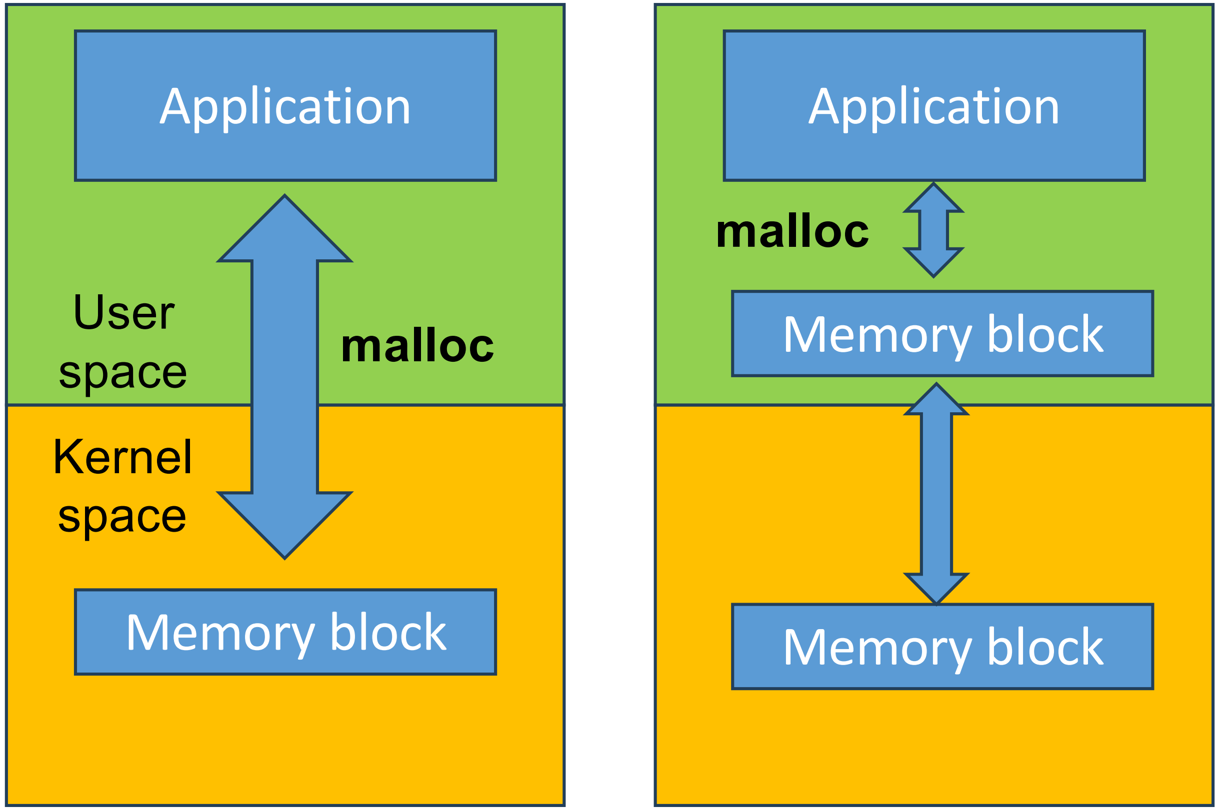
- Static size, dynamic allocation
  - Local variables
  - Compiler directs stack allocation

- Dynamic size, dynamic allocation
  - Programmer controlled
  - **Allocated in the heap – how?**

Dynamic Memory Allocation

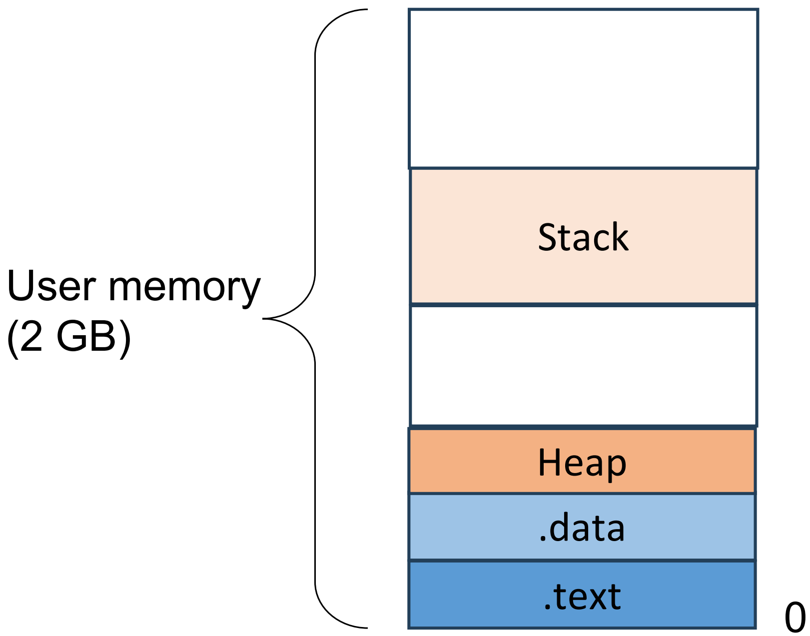
- Explicit vs. implicit memory allocator
  - Explicit: application allocates and *frees* space
    - e.g., malloc and free in C
  - Implicit: application allocates, but does not free space
    - e.g., garbage collection in Java or Python
- Allocation
  - In both cases the memory allocator provides an abstraction of memory as a set of blocks
  - Gives out free memory blocks to application

Which is Better and Why?



- Case-1
  - Switch from user space to kernel space at every call to malloc
  - Huge overheads in case of several calls to malloc
- Case-2
  - A big chunk of memory is allocated by the OS and mapped into process's address space
  - Calls to malloc then returns memory allocated inside the process address space
  - Switch from user space to kernel space is significantly reduced

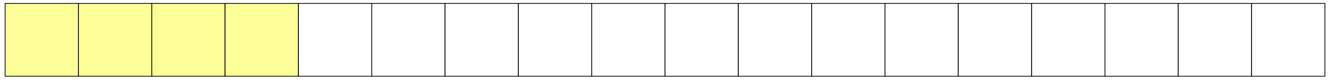
Process Memory Image



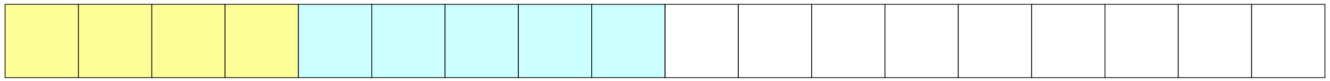
- `void *sbrk(intptr_t incr)`
  - Easiest to use function for allocators to request additional heap memory from the OS
  - `brk` used to initially set the fixed size of memory (e.g., data section)
  - `sbrk` provides convenience by allowing increment/decrement of heap by `incr` bytes (new memory is zero filled)
    - The `incr` can be negative, in which case the amount of allocated space is decreased

Dynamic Memory Allocation Examples

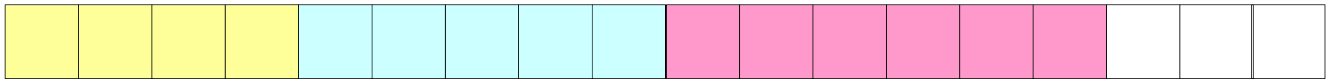
```
p1 = malloc(4*sizeof(int))
```



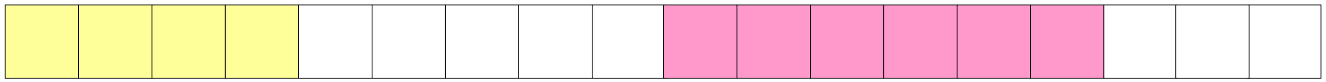
```
p2 = malloc(5*sizeof(int))
```



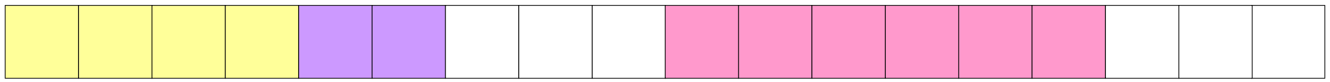
```
p3 = malloc(6*sizeof(int))
```



```
free(p2)
```



```
p4 = malloc(2*sizeof(int))
```



Goals of Good `malloc/free`

- Good performance for malloc and free
  - Ideally should take constant time (not always possible)
  - Should certainly not take linear time in the number of blocks
- Good space utilization
  - User allocated structures should use most of the heap
  - Want to minimize "**fragmentation**"

Today's Class

- Dynamic memory allocations
- Fragmentation
- Allocation using implicit list
- Allocation using buddy allocator

Fragmentation

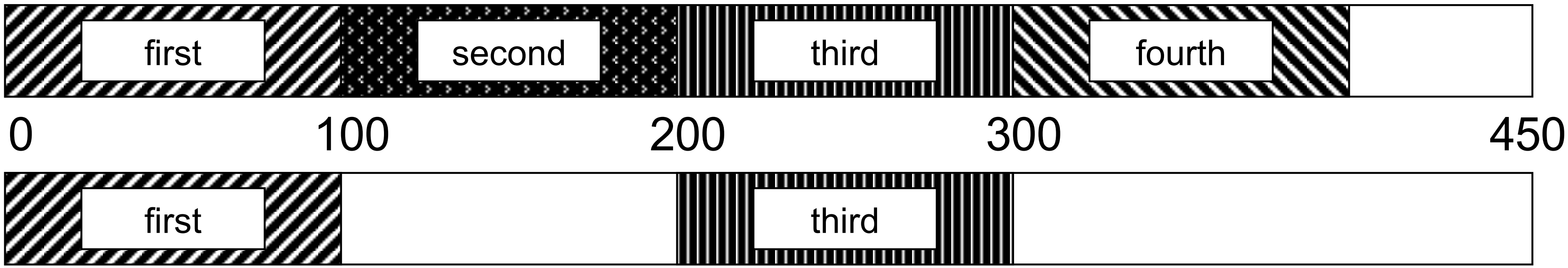
- Segments of memory can become unusable due to the result of allocation scheme
- Two types of fragmentation
  - External fragmentation
    - Memory remains unallocated
    - Variable allocation sizes
  - Internal fragmentation
    - Memory is allocated but unused
    - Fixed allocation sizes

External Fragmentation

- Imagine calling a series of *malloc* and *free*

```
char* first = malloc(100);
char* second = malloc(100);
char* third = malloc(100);
char* fourth = malloc(100);
free(second);
free(fourth);
char* problem = malloc(200);
```

- 250 free bytes of memory, only 150 contiguous
  - unable to satisfy final malloc request



Internal Fragmentation

- Imagine calling a series of *malloc*
  - Assume allocation unit is 100 bytes

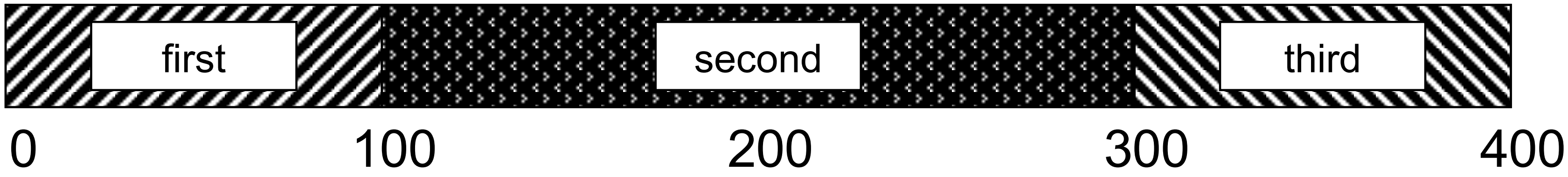
    char* first = malloc(90);

    char* second = malloc(120);

    char* third = malloc(10);

    char* problem = malloc(50);

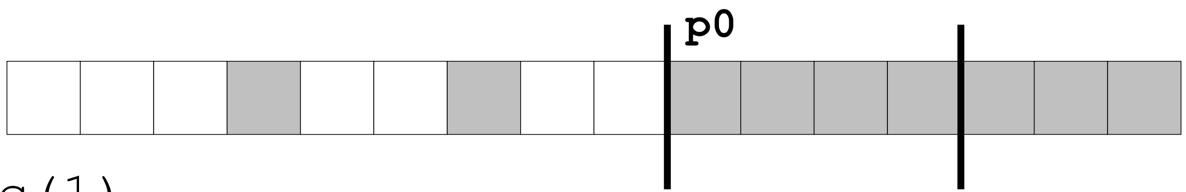
- All of memory has been allocated but only a fraction of it is used (220 bytes)
  - Unable to handle final memory request



Dynamic Memory Allocation Challenges

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

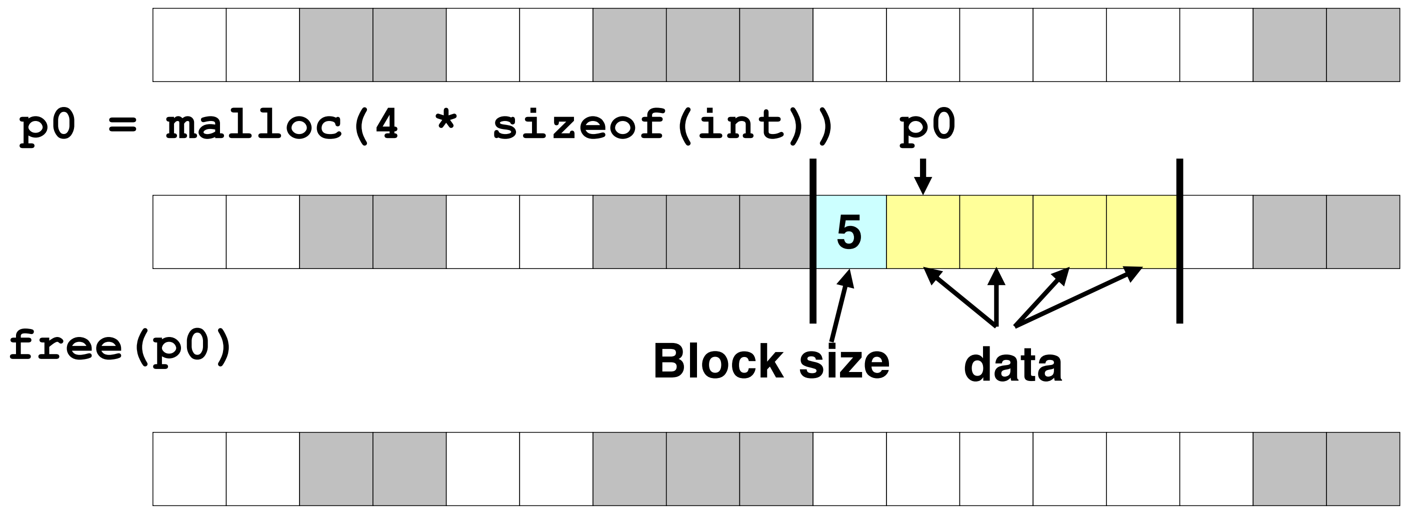
`free(p0)`



`p1 = malloc(1)`

Knowing How Much to Free

- Standard method
  - Keep the length of a block in the memory slot preceding the block
    - This slot is often called the header field or header
  - Requires a slot for every allocated block



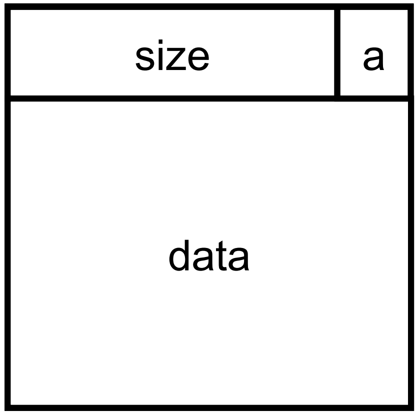
Today's Class

- Dynamic memory allocations
- Fragmentation
- Allocation using implicit list
- Allocation using buddy allocator

Implicit List: Keeping Track of Memory Blocks

- Need to identify whether each block is free or allocated
  - Block header at the start of each block of memory
    - Header is a structure with size and status

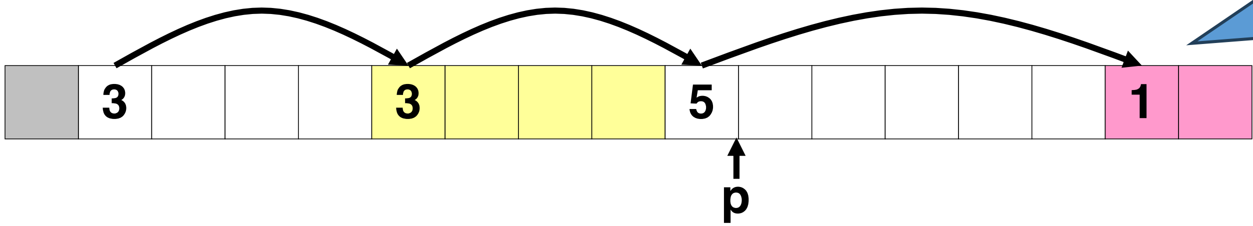
Format of allocated and free blocks



a = 1: allocated block
a = 0: free block

size: block size

data: application data (allocated blocks only)



Blocks of memory (both free or used) are connected to each other as linked list

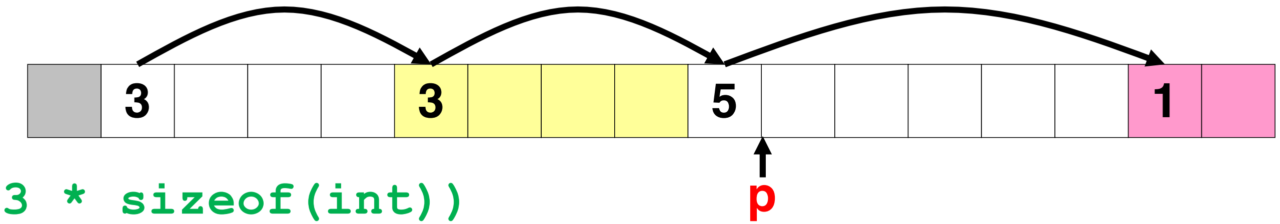
Implicit List: **Finding a Free Block**

- First fit:
  - Search list from beginning, choose first free block that fits
    - May miss a block with closest size that fits
  - Can take linear time in total number of blocks (allocated/free)
- Next fit:
  - Like first-fit, but search list from end of previous search
  - Research suggests that fragmentation is worse
    - May miss a block with closest size that fits
- **Best fit**:
  - Choose the free block with the closest size that fits
    - Requires complete search of the list
  - Keeps fragments small – usually helps fragmentation
  - Will typically run slower than first-fit

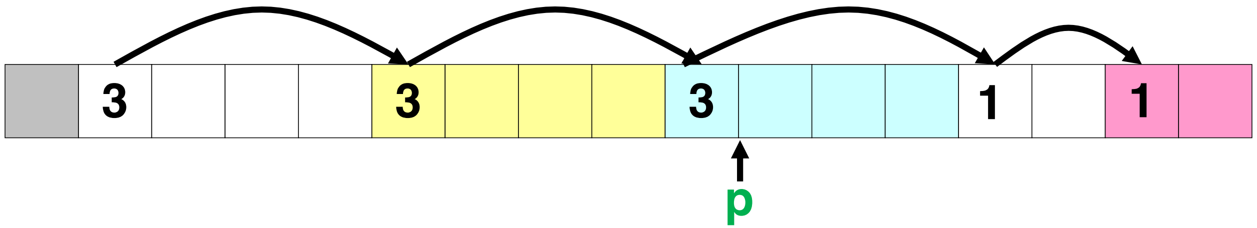
Implicit List: **Allocating in Free Block**

- Allocating in a free block
  - Since allocated space might be smaller than free space, we might want to split the block (Splitting)

```
p = malloc(3 * sizeof(int))
```

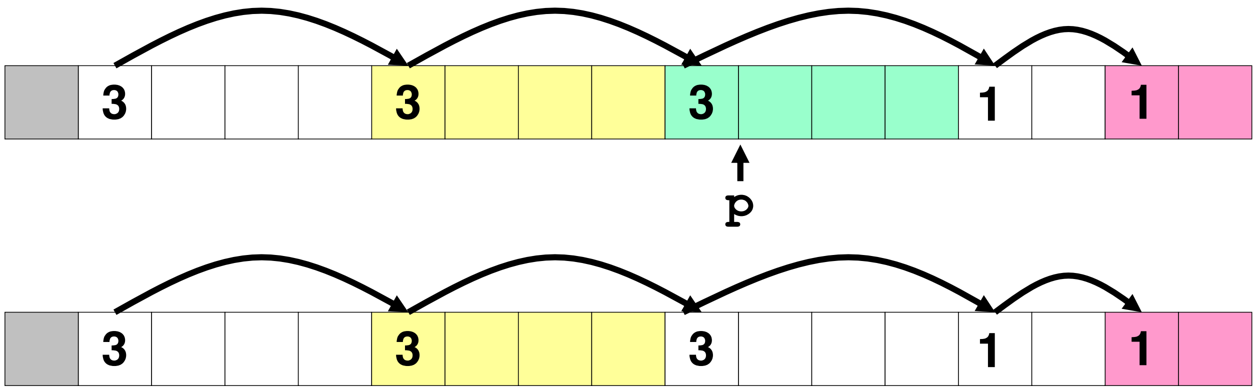



```
p = malloc(3 * sizeof(int))
```



Implicit List: **Freeing a Block**

- Simplest implementation:
  - Only need to clear allocated flag
  - But can lead to "false fragmentation"



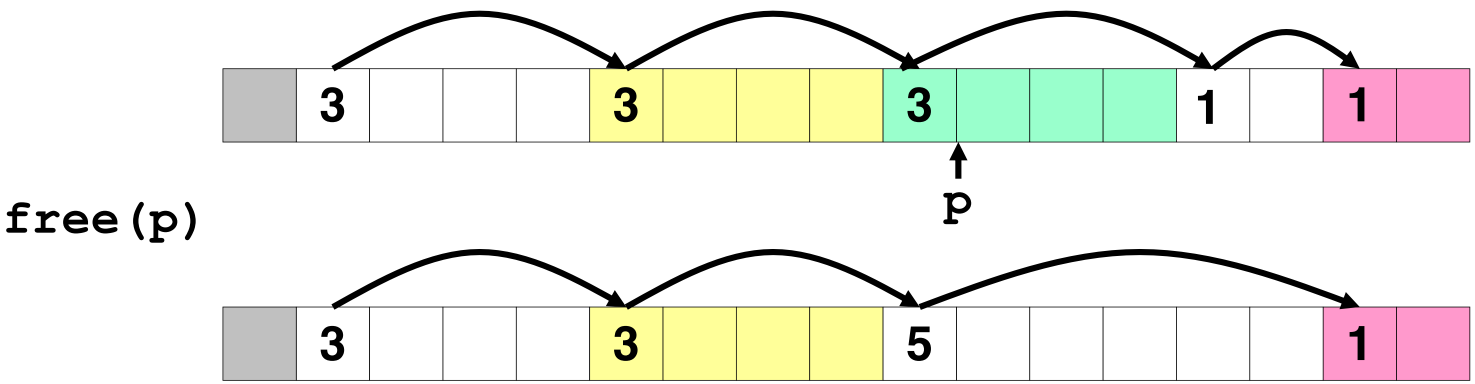
`free(p)`

`malloc(4 * sizeof(int))` **Oops!**

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

Implicit List: **Coalescing**

- Join (coalesce) with next and/or previous block **if free**
  - Coalescing with **next** block

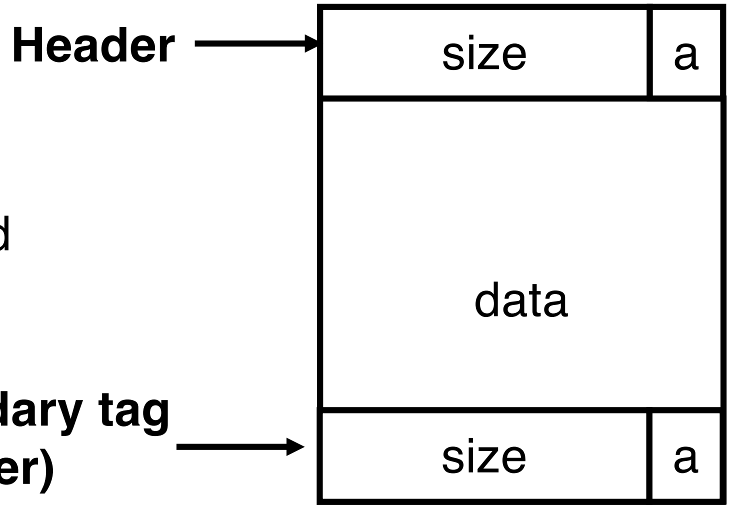


  - But how do we coalesce with previous block?

Implicit List: **Bidirectional Coalescing**

- Boundary tags [Knuth73]
  - Replicate header at end of block
  - Allows us to traverse the “list” backwards, but requires extra space

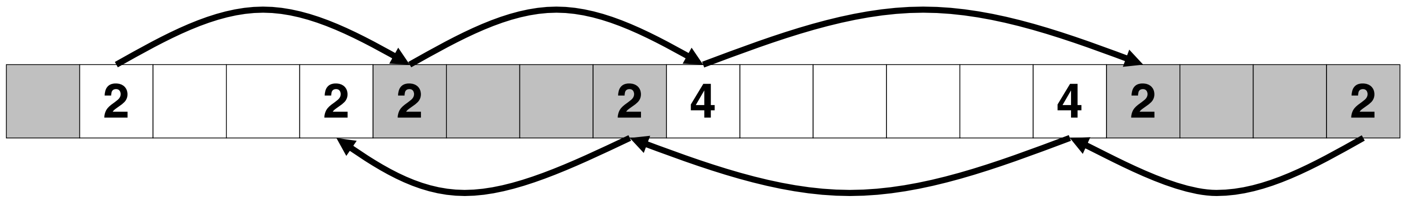
Format of allocated and free blocks



a = 1: allocated block
a = 0: free block

size: total block size

application data
(allocated blocks only)



Today's Class

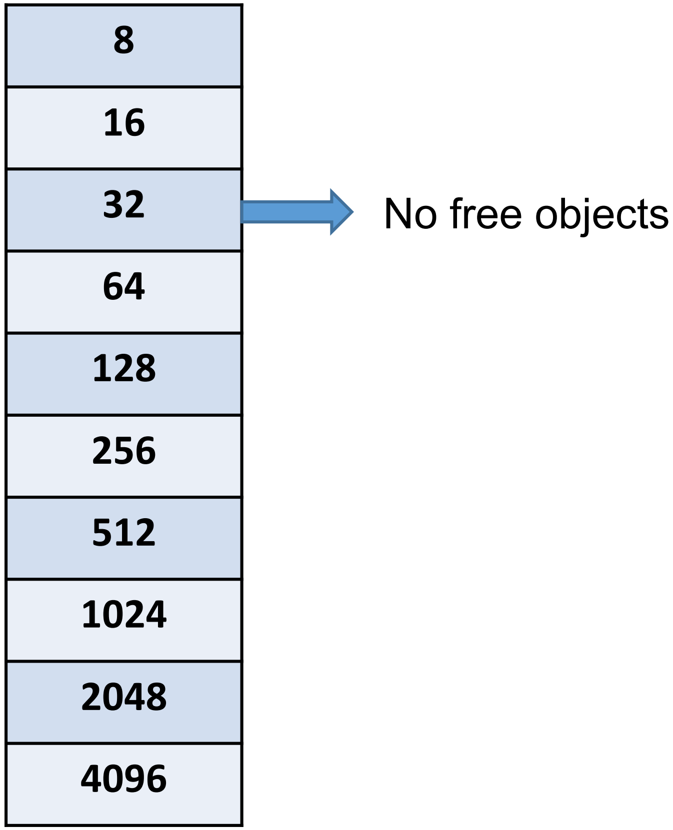
- Dynamic memory allocations
- Fragmentation
- Allocation using implicit list
- Allocation using buddy allocator

Buddy Allocator

8
16
32
64
128
256
512
1024
2048
4096

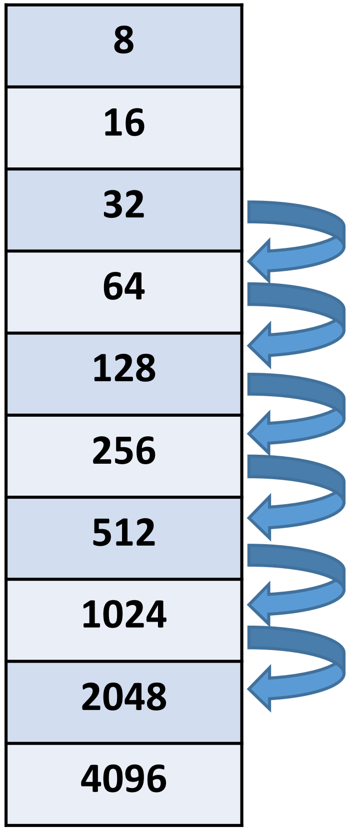
- Implicit list
  - Memory usage depends on placement policy
    - First fit, next fit, or best fit
  - Prone to fragmentation
- **Buddy allocator**
  - It maintains free buckets of memory objects of different allocation sizes (2^k)
    - Initially all buckets are empty
  - Faster allocation
  - Reduces fragmentation

Buddy Allocator



- A= malloc(24)
  - Rounded size = 32
    - Little bit of fragmentation, but let's ignore that
  - No free objects in that bucket

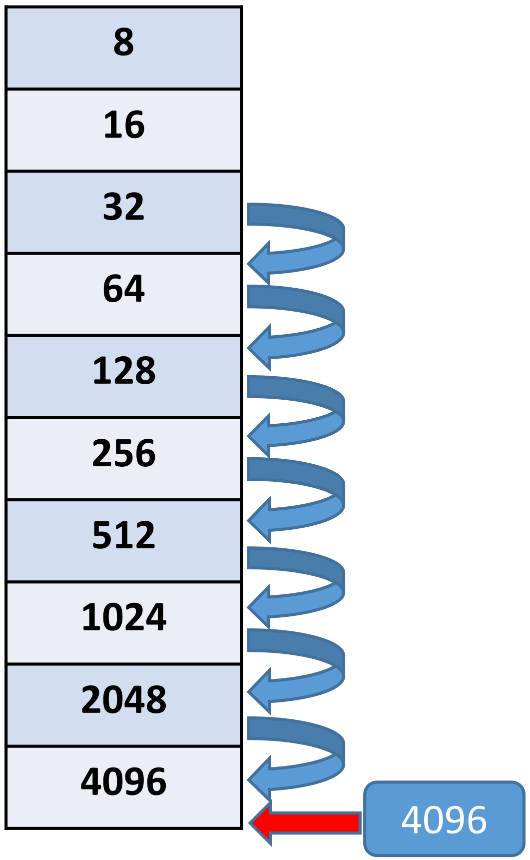
Buddy Allocator



- Allocator recursively calls itself to allocate an object that is twice the size of the bucket size
- Call would eventually lead to bucket 4096

Source: Piyus Kedia, https://piyus.github.io/OS/index.html

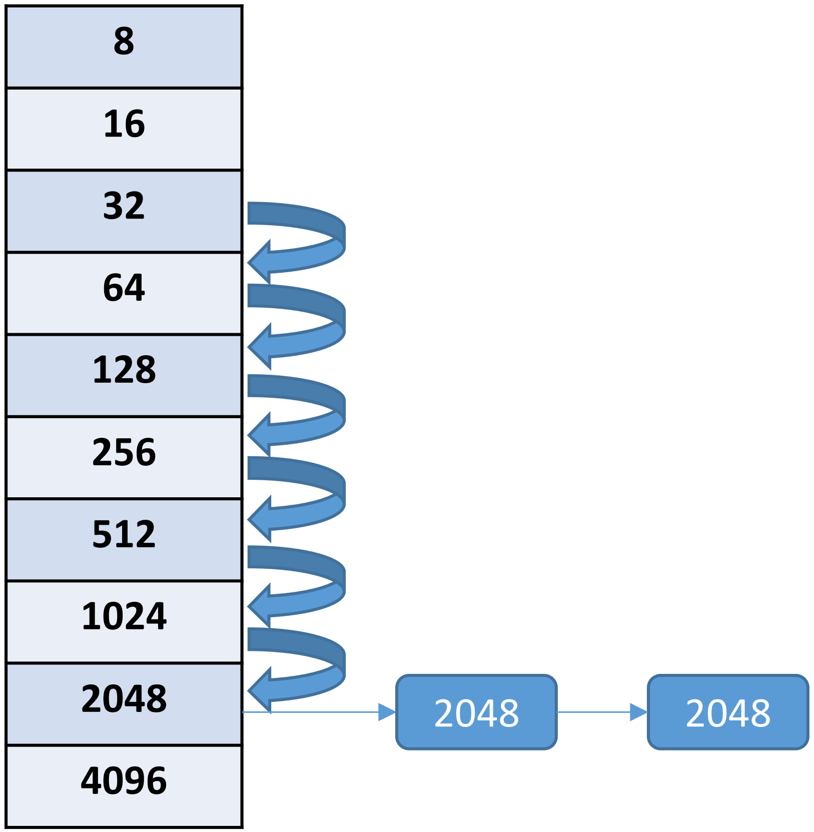
Buddy Allocator



No free objects. Hence, allocate 4Kb from RAM

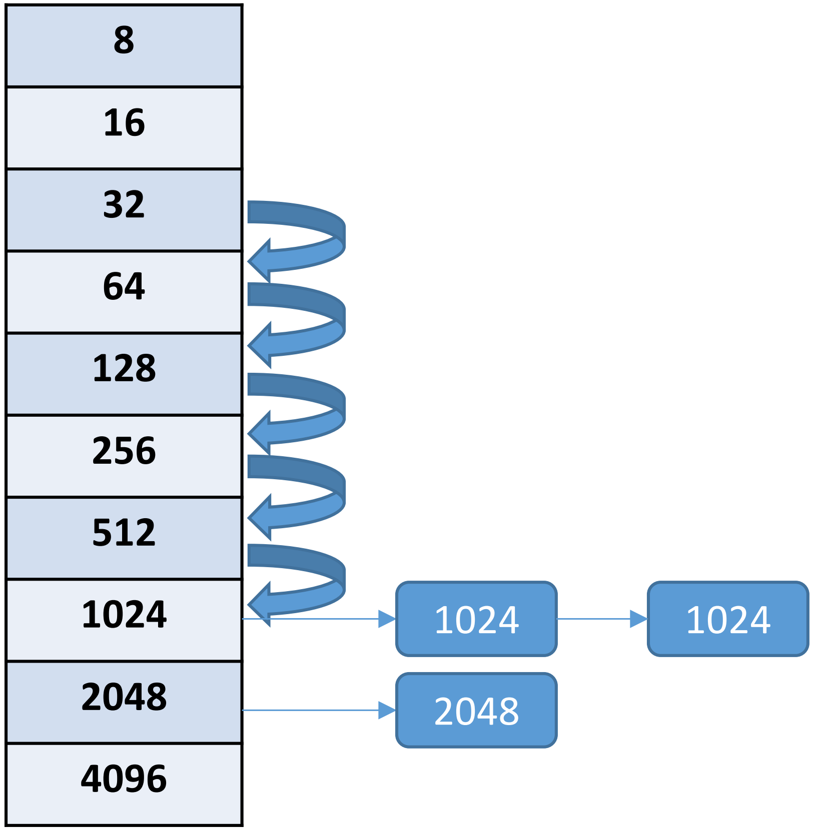
- There is no free object in the bucket size 4096
  - Request OS to allocate a chunk of size 4096 bytes from the RAM and return to its caller
    - System call
    - 4096 bytes is also called as page size in Linux (will be covered in Lecture 17)

Buddy Allocator



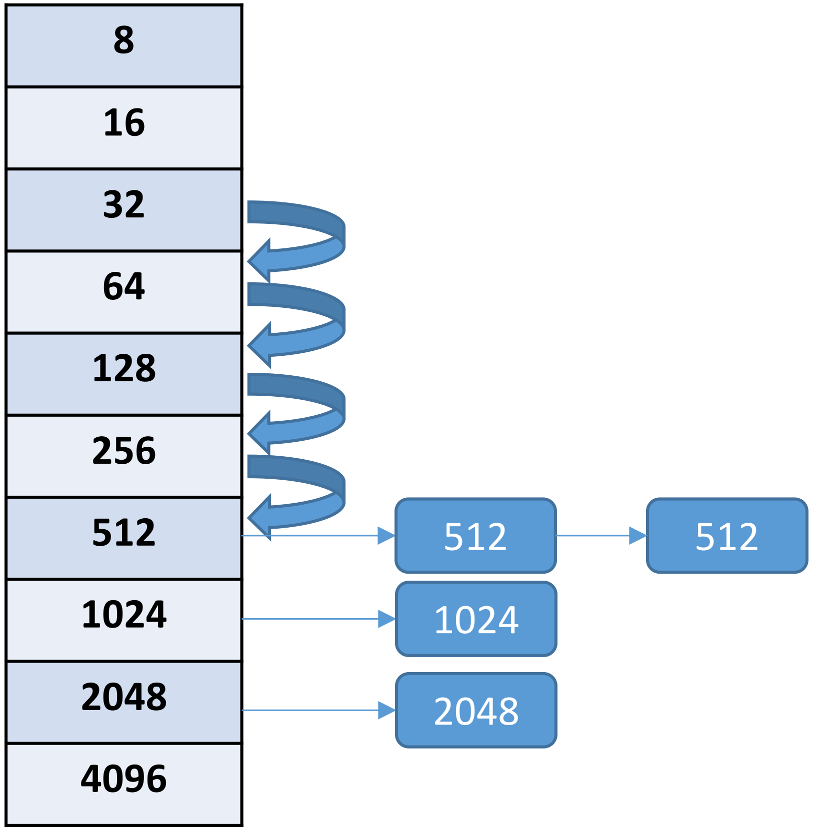
- Split the 4096 object into two halves (two 2048 objects), add into the parent bucket
  - Bucket 2048

Buddy Allocator



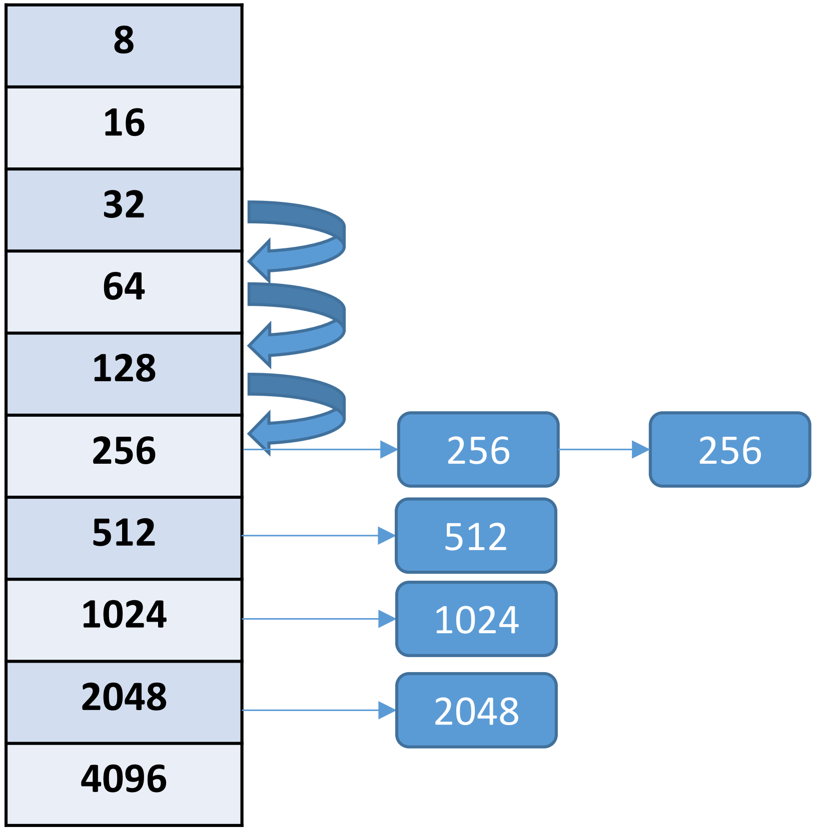
- One of the 2048 object is promoted to the bucket 1024 where it is split into two objects of size 1024 each

Buddy Allocator



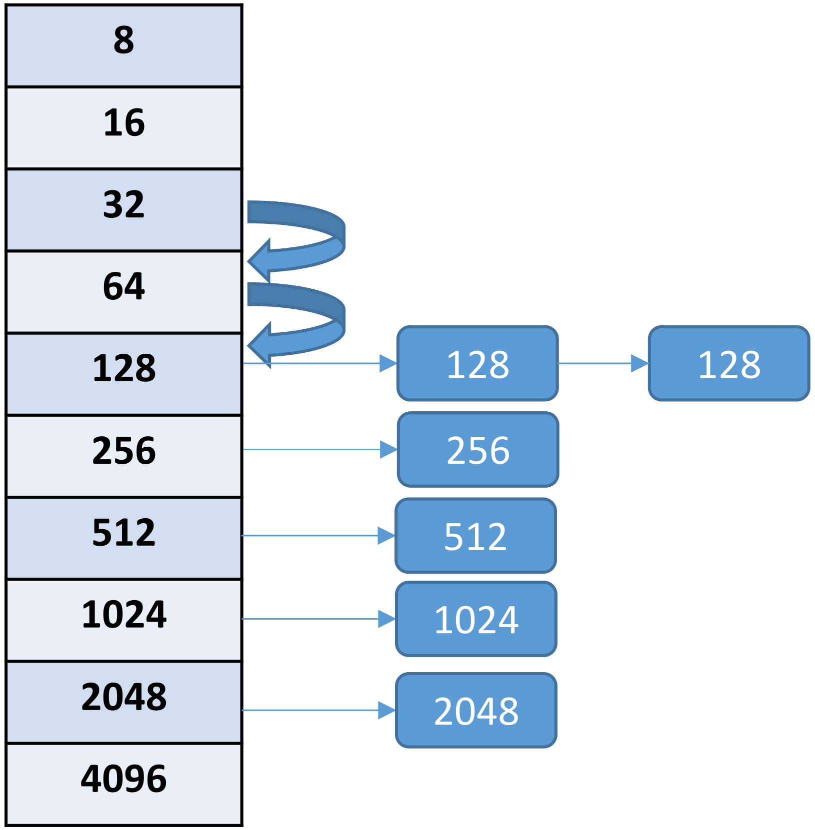
- One of the 1024 object is promoted to the bucket 512 where it is split into two objects of size 512 each

Buddy Allocator



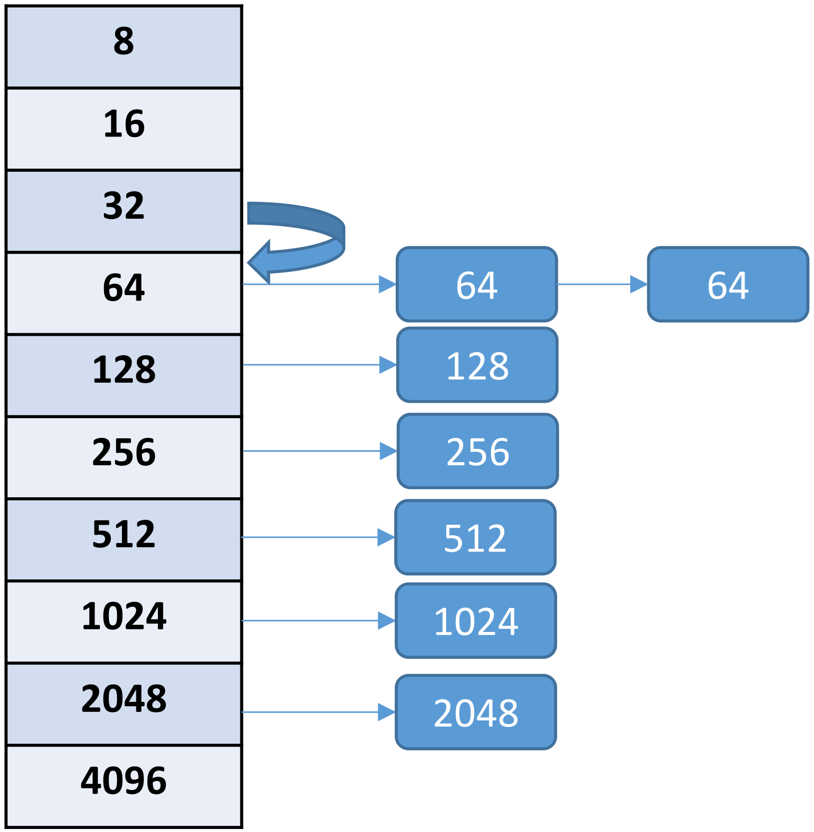
- One of the 512 object is promoted to the bucket 256 where it is split into two objects of size 256 each

Buddy Allocator



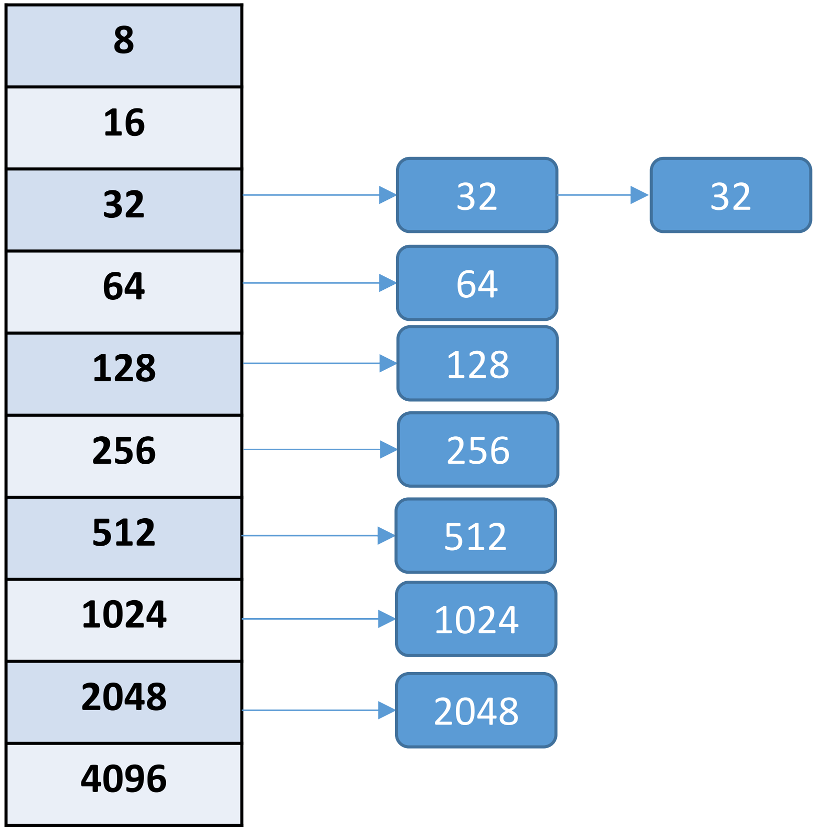
- One of the 256 object is promoted to the bucket 128 where it is split into two objects of size 128 each

Buddy Allocator



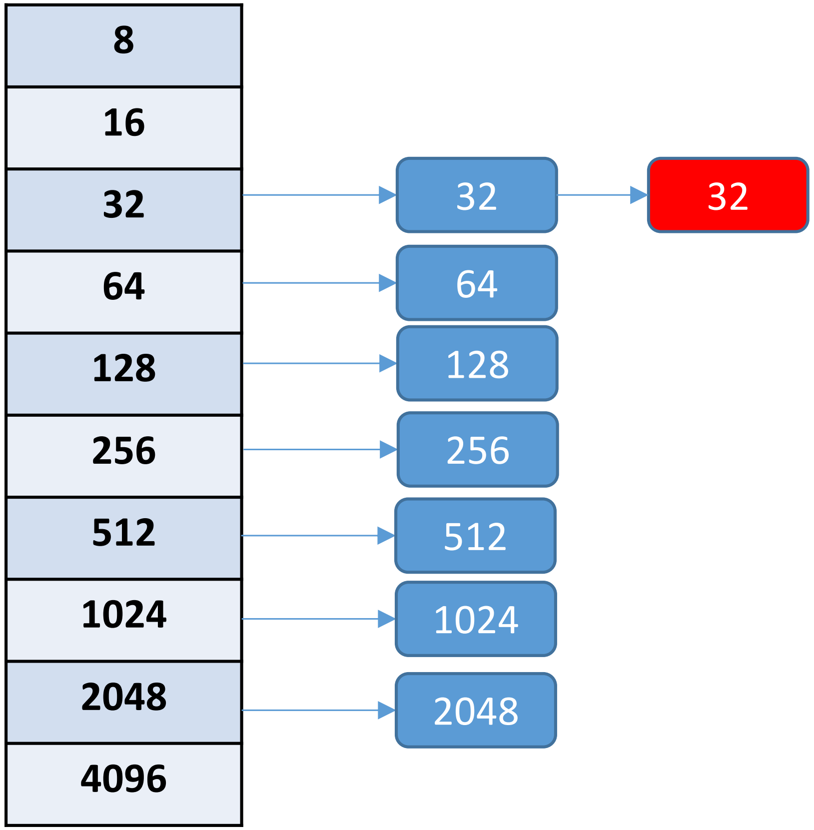
- One of the 128 object is promoted to the bucket 64 where it is split into two objects of size 64 each

Buddy Allocator



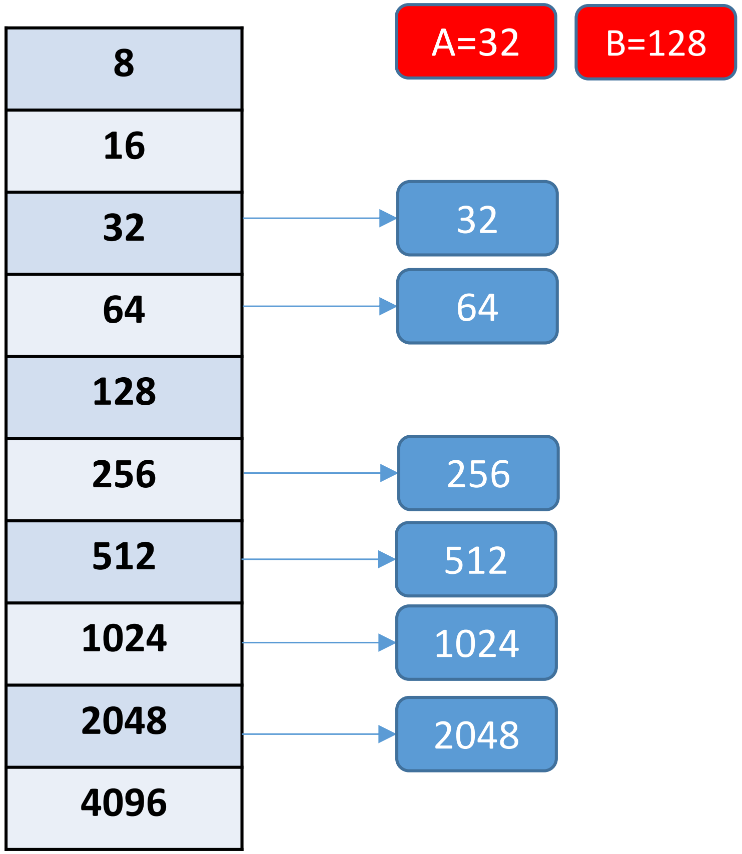
- One of the 64 object is promoted to the bucket 32 where it is split into two objects of size 32 each

Buddy Allocator



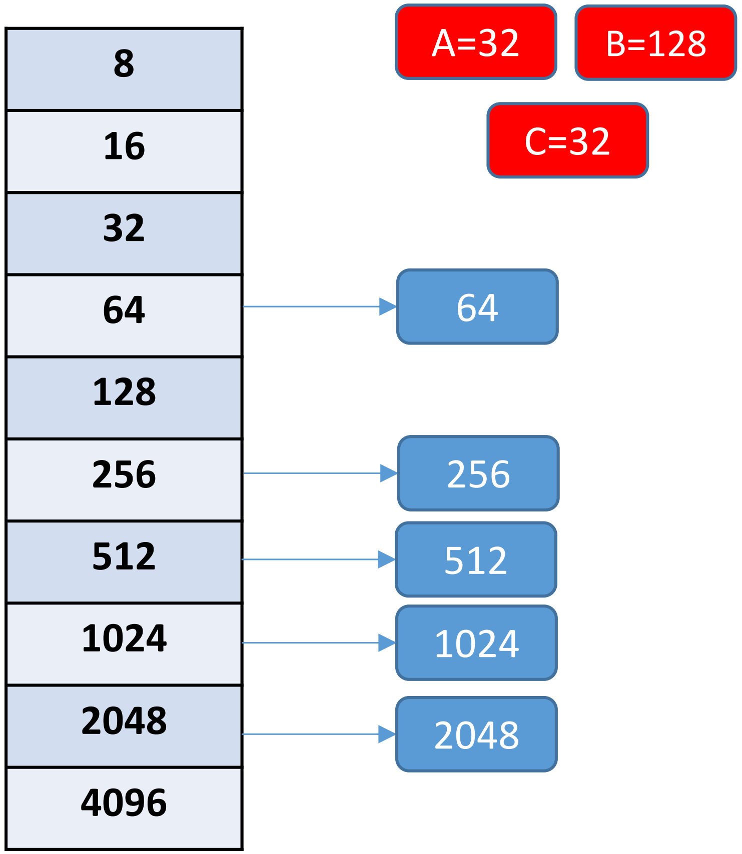
- One of the 32-byte object in bucket 32 will now be returned to the malloc(32) call

Buddy Allocator



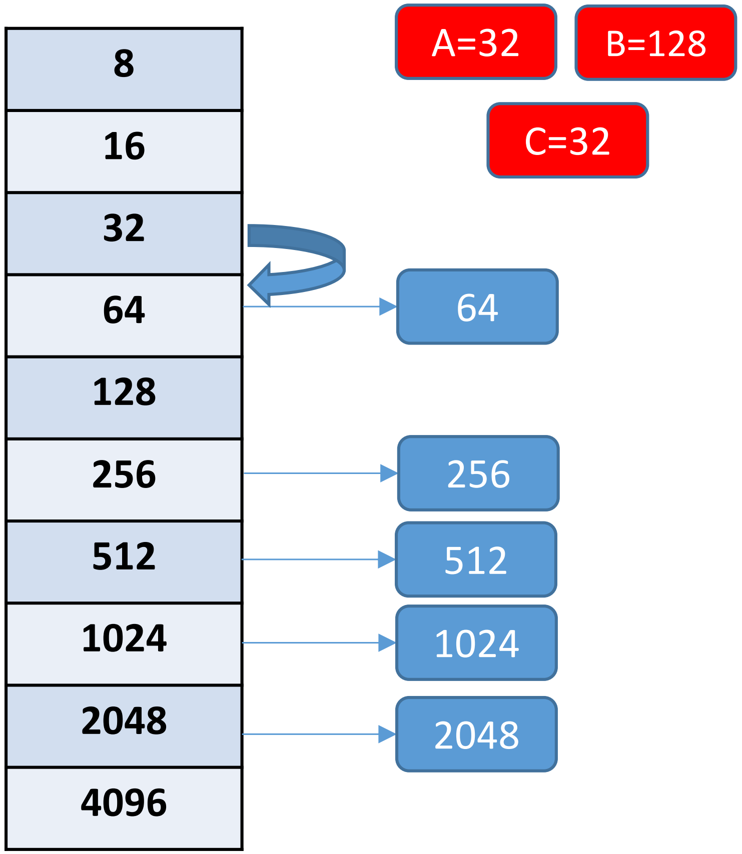
- B = malloc(128)
  - One free object already in that bucket 128
  - Return that free object from the bucket to the malloc(128)
  - Now no more free object in bucket 128

Buddy Allocator



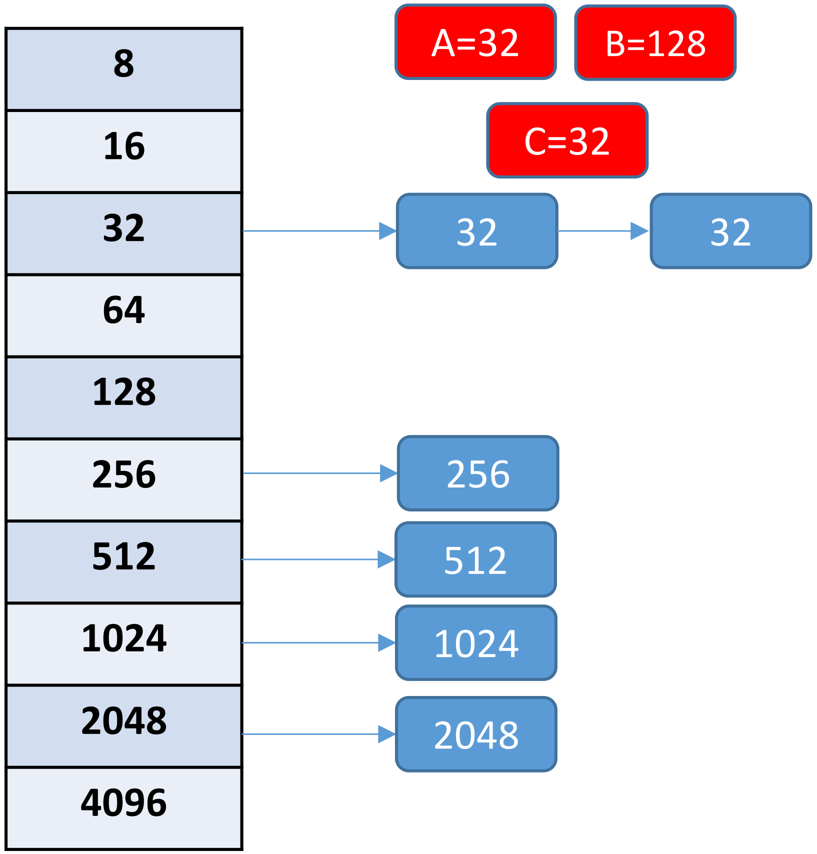
- C = malloc(32)
  - One free object already in that bucket
  - Return that free object from the bucket to the malloc(32)
  - Now no more free object in bucket 32

Buddy Allocator



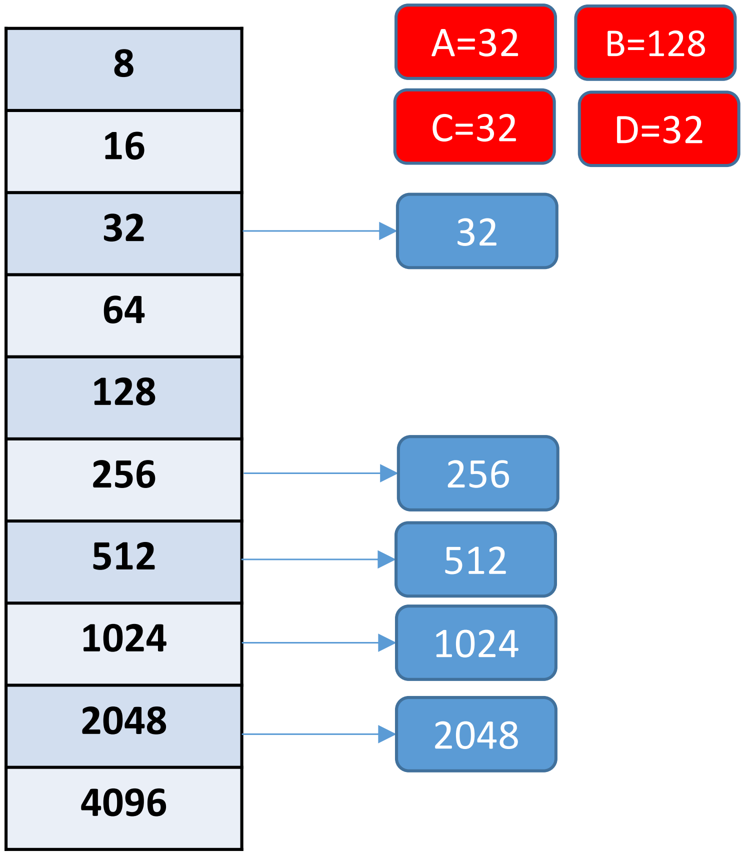
- D = malloc(32)
  - No more free object exists in the bucket 32
  - Fetch one object from the buddy bucket 64

Buddy Allocator



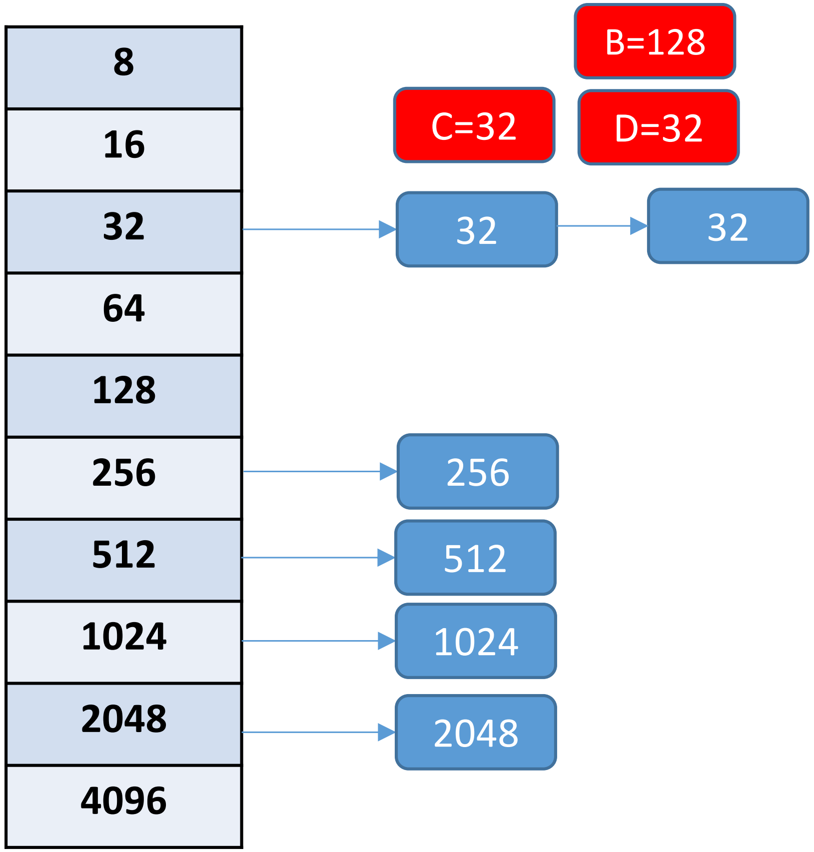
- Object of 64 bytes received from buddy bucket 64 will be split into two 32 byte free objects and added into the bucket 32

Buddy Allocator



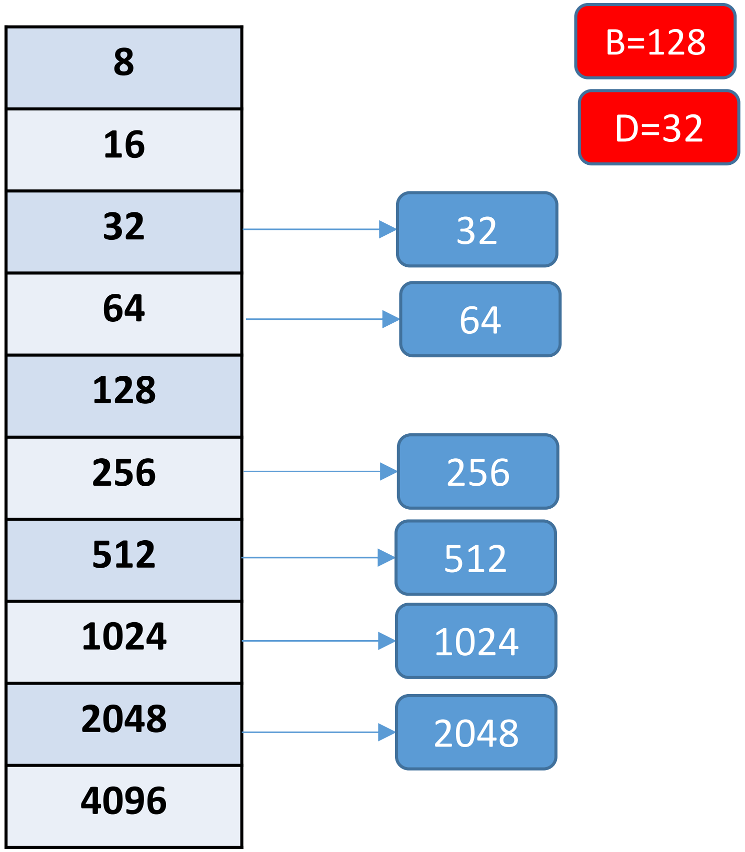
- One free object from bucket 32 returned to malloc(32) call

Buddy Allocator



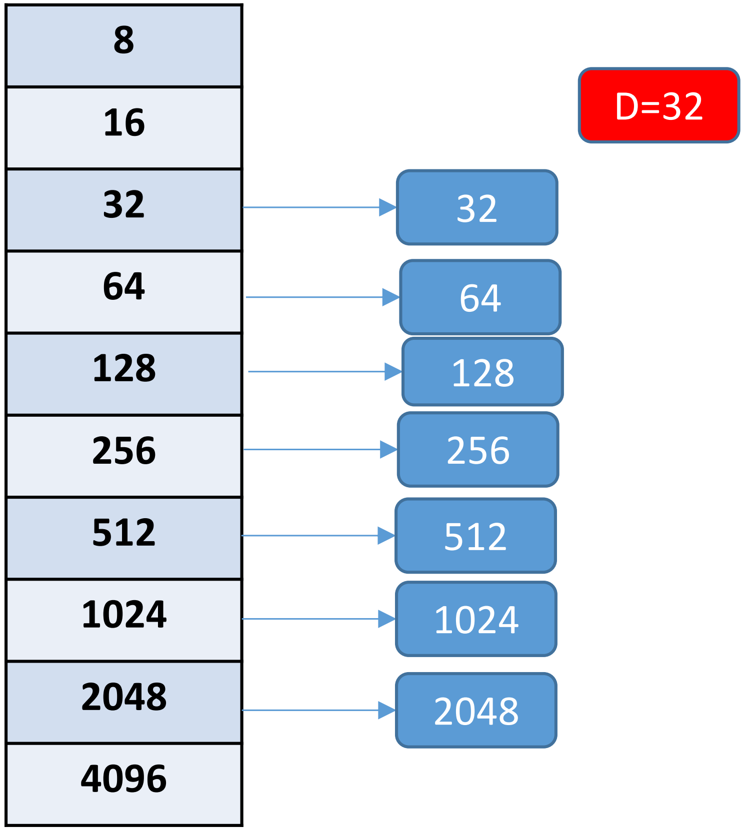
- free(A)
  - Append to the free object list in bucket 32

Buddy Allocator



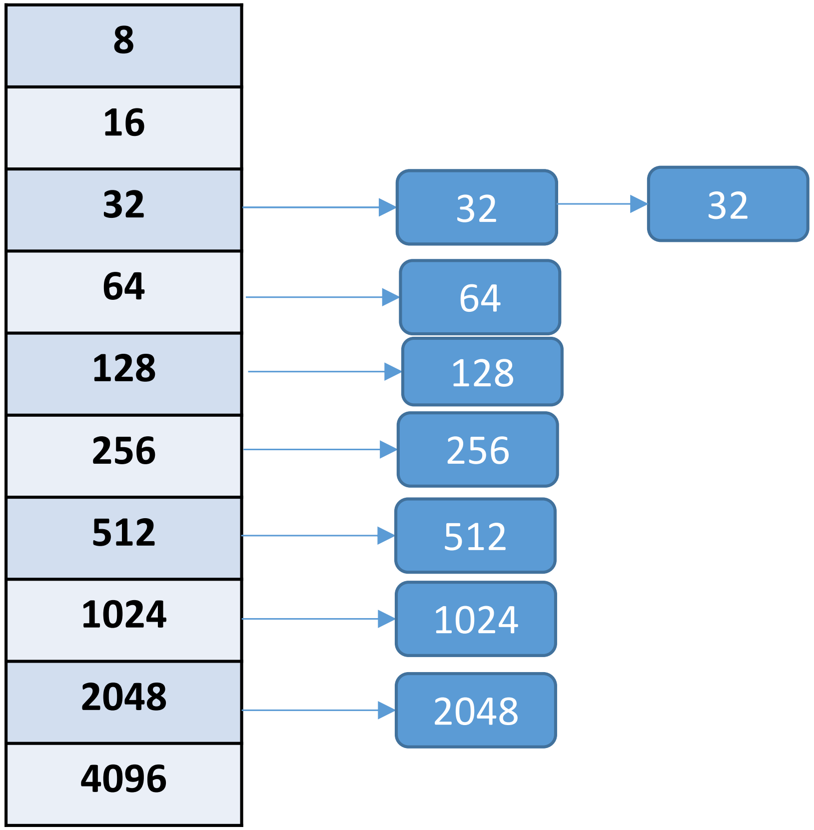
- free(C)
  - One more memory object appended to the free list in bucket 32 which results in a total of 3 free objects in that bucket
  - Two of the free objects are coalesced (merged) and promoted to bucket 64
- If merging and promotion doesn't happen then it will result in unnecessary calls to OS for 4Kb objects
  - System call invocation, thereby leading to overheads!

Buddy Allocator



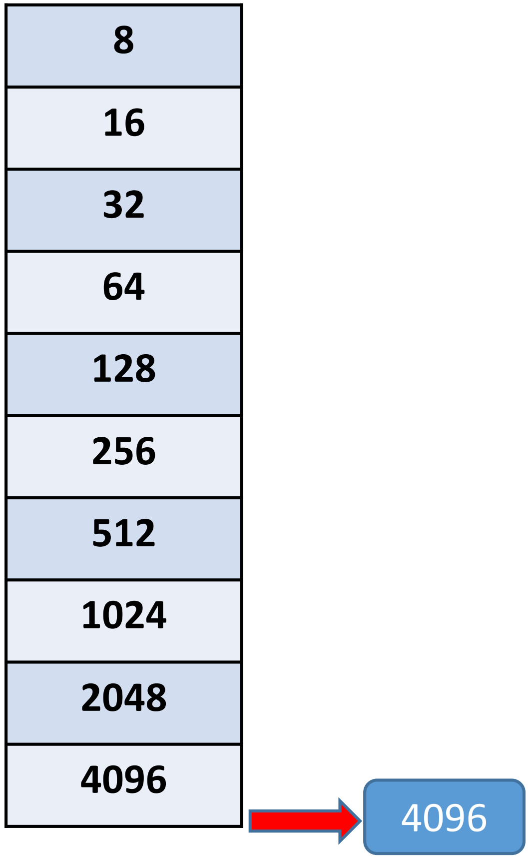
- free(B)
  - Append to the bucket 128

Buddy Allocator



- free(D)
  - Append the memory object back to the bucket (32)
- Finally, during process termination, all free objects are coalesced back into 4096 bucket and returned to the OS

Buddy Allocator



- All free objects are coalesced back into 4096 bucket and returned to the OS at process termination
  - What would happen if memory not returned to OS?
- How to handle allocations greater than 4096 bytes?
  - The size is rounded up to the nearest multiples of 4096 and allocated directly from OS

Next Lecture

- Mid semester review