# Dynamic Memory Allocation

Zhaoguo Wang

# Why dynamic memory allocation?

Do not know the size until the program runs (at runtime).

```
#define MAXN 15213
int array[MAXN];

int main(void)
{
      int i, n;
      scanf("%d", &n);
      if (n > MAXN)
            app_error("Input file too big");
      for (i = 0; i < n; i++)
            scanf("%d", &array[i]);
      exit(0);
}
```

# Why dynamic memory allocation?

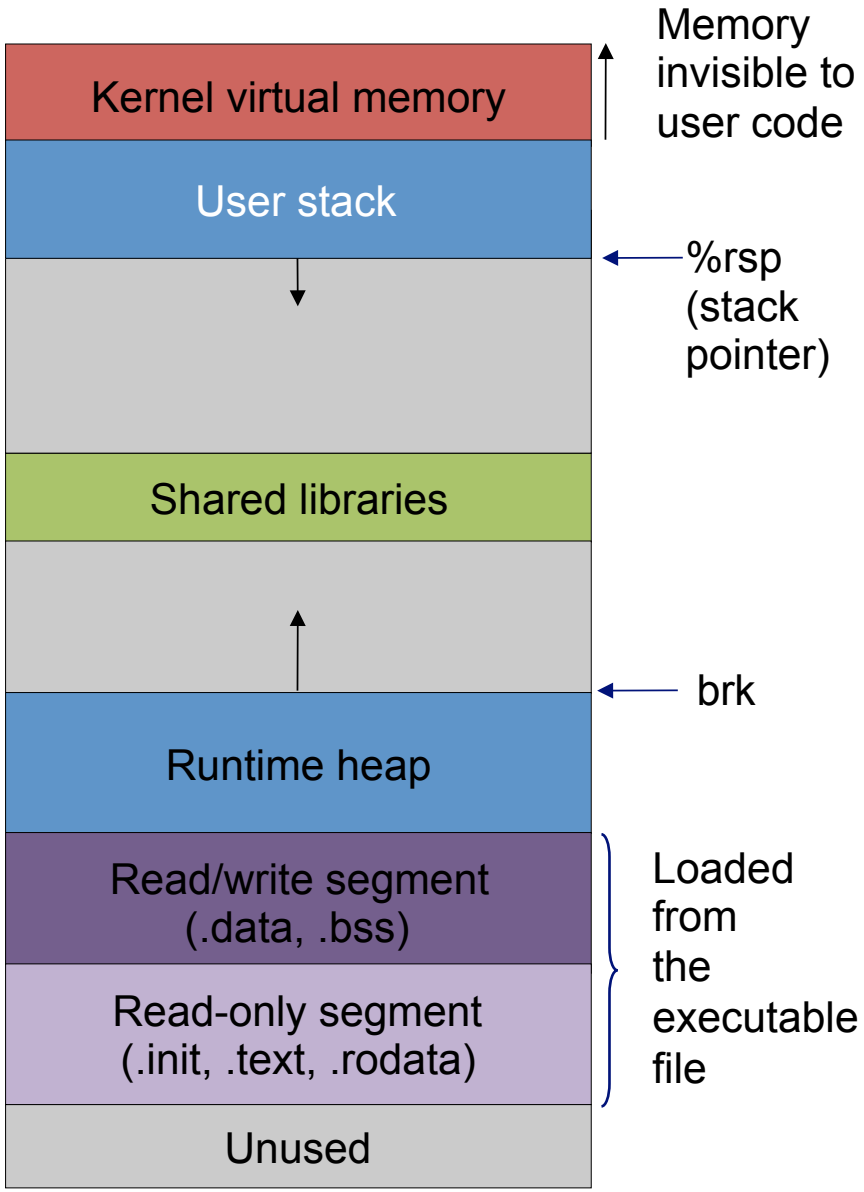
Do not know the size until the program runs (at runtime).

```
int main(void)
{
    int *array, i, n;

    scanf("%d", &n);
    array = (int *)malloc(n * sizeof(int));
    for (i = 0; i < n; i++)
        scanf("%d", &array[i]);
    exit(0);
}
```

# Dynamic allocation on heap

Question: Is it possible to dynamically allocate memory on stack?



# Dynamic allocation on heap

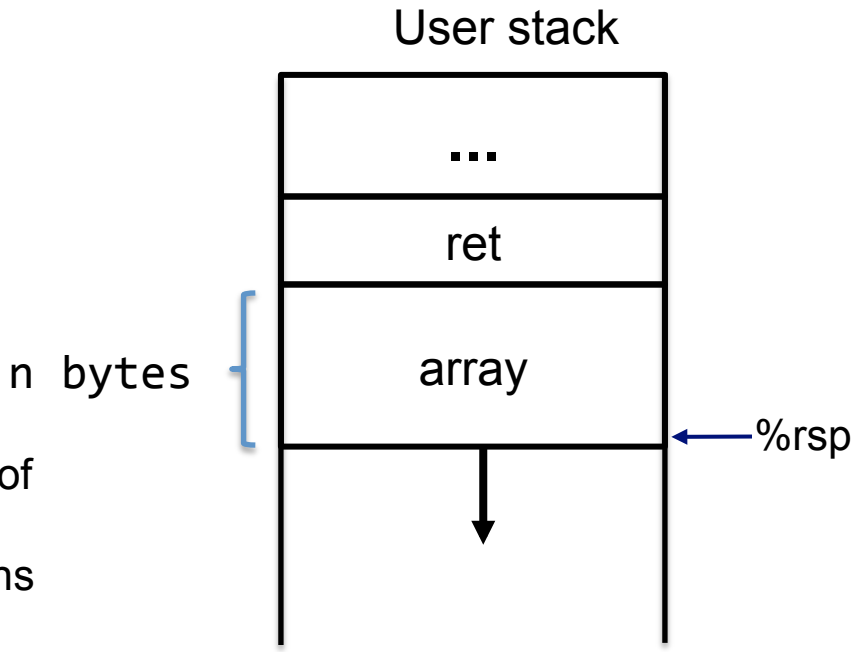
Question: Is it possible to dynamically allocate memory on stack?

Answer: Yes

```
void *alloca(size_t size);
```

Allocates size bytes of space in the stack frame of the caller. This temporary space is automatically freed when the function that called alloca() returns to its caller.

```
void func(int n) {
    array = alloca(n);
}
```



# Dynamic allocation on heap

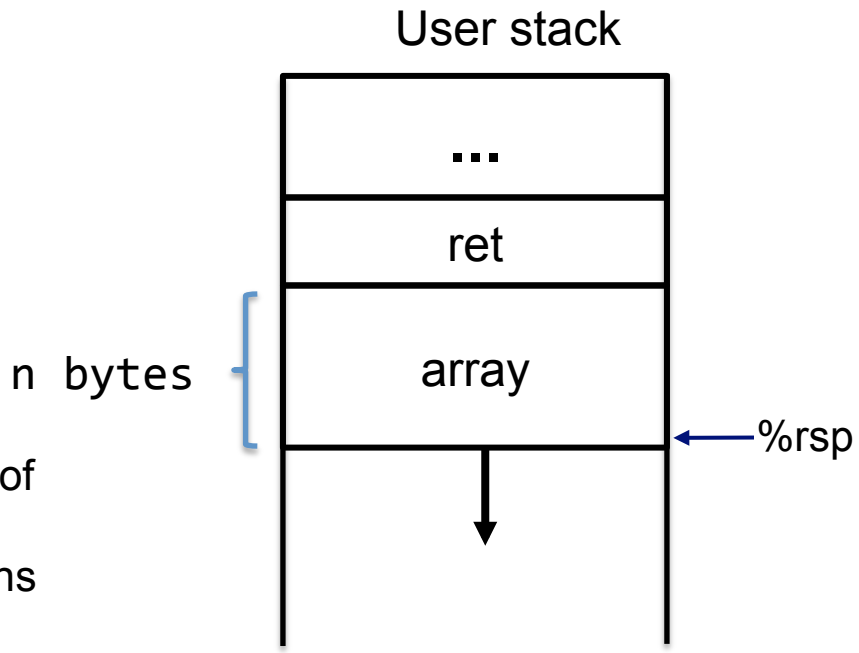
Question: Is it possible to dynamically allocate memory on stack?

Answer: Yes

```
void *alloca(size_t size);
```

Allocates size bytes of space in the stack frame of the caller. This temporary space is automatically freed when the function that called alloca() returns to its caller.

```
void func(int n) {
    array = alloca(n);
}
```



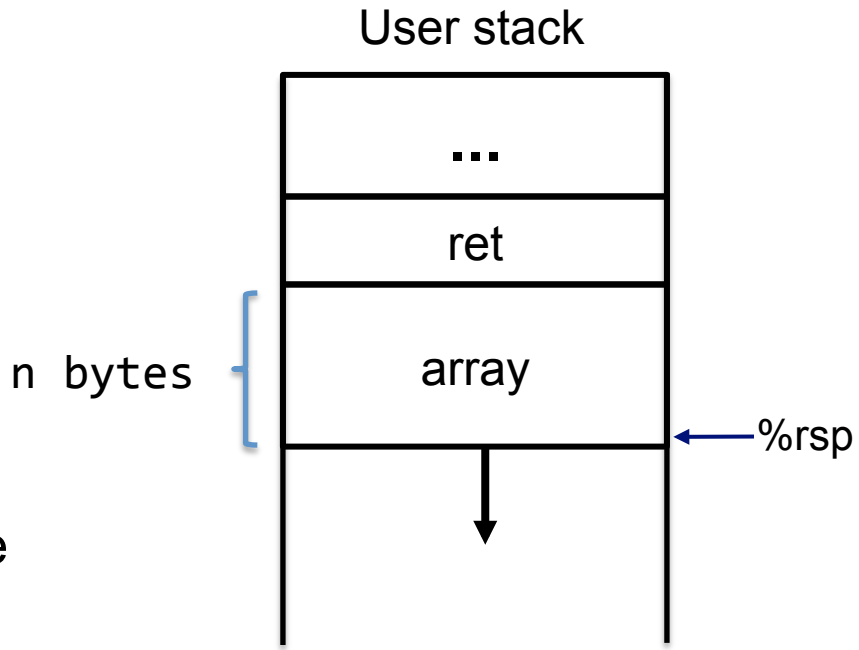

Not good practice!

# Dynamic allocation on heap

Question: Is it possible to dynamically allocate memory on stack?

Answer: Yes

```
void *alloca(size_t size);
```




Issue I – Can not free memory until current function returns → Blowing the stack up

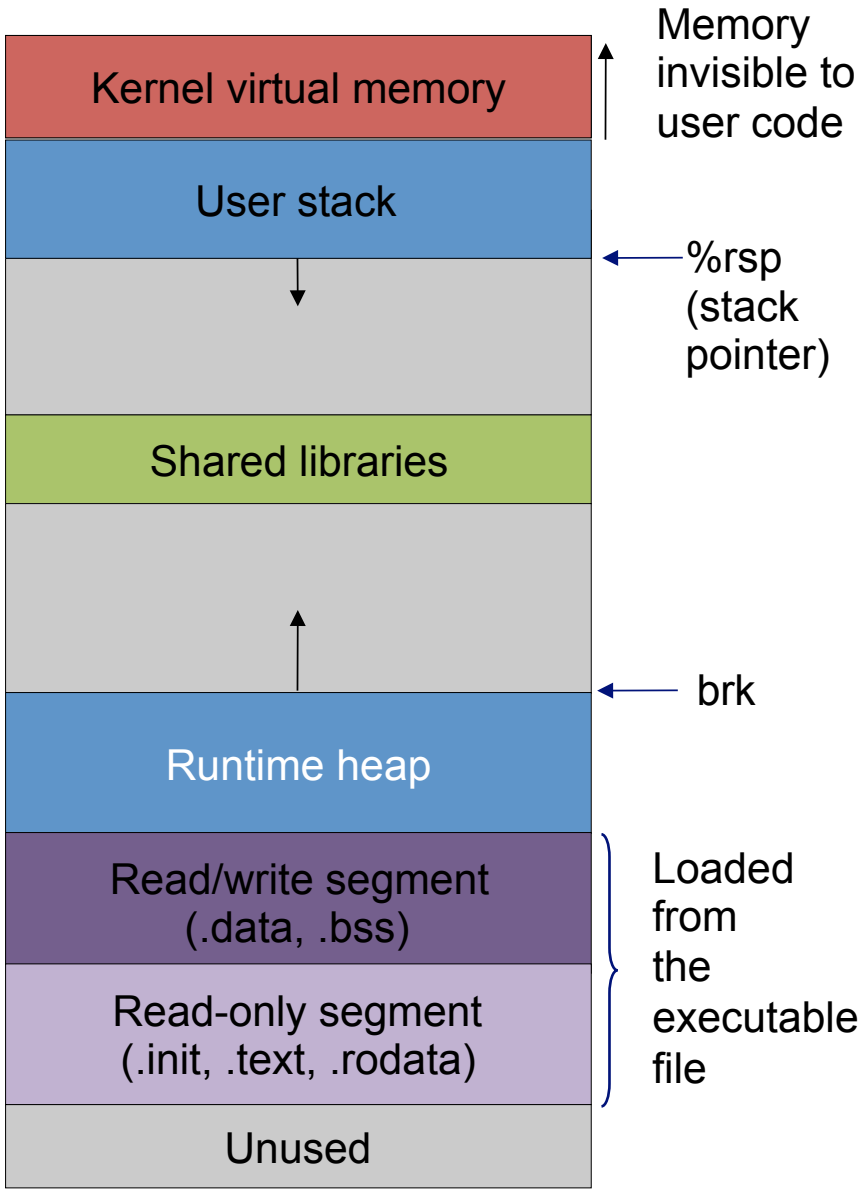
Issue II – Can not pass the memory out of the scope → Copy memories across different functions

Issue III – Return value points to the top of the stack → buffer overflow (attack)

Not good practice!

# Dynamic allocation on heap

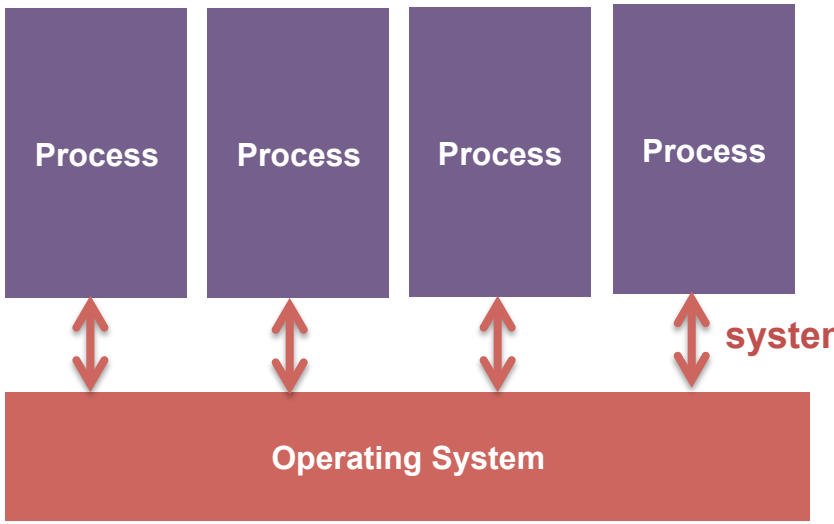
Question: How to allocate memory on heap?

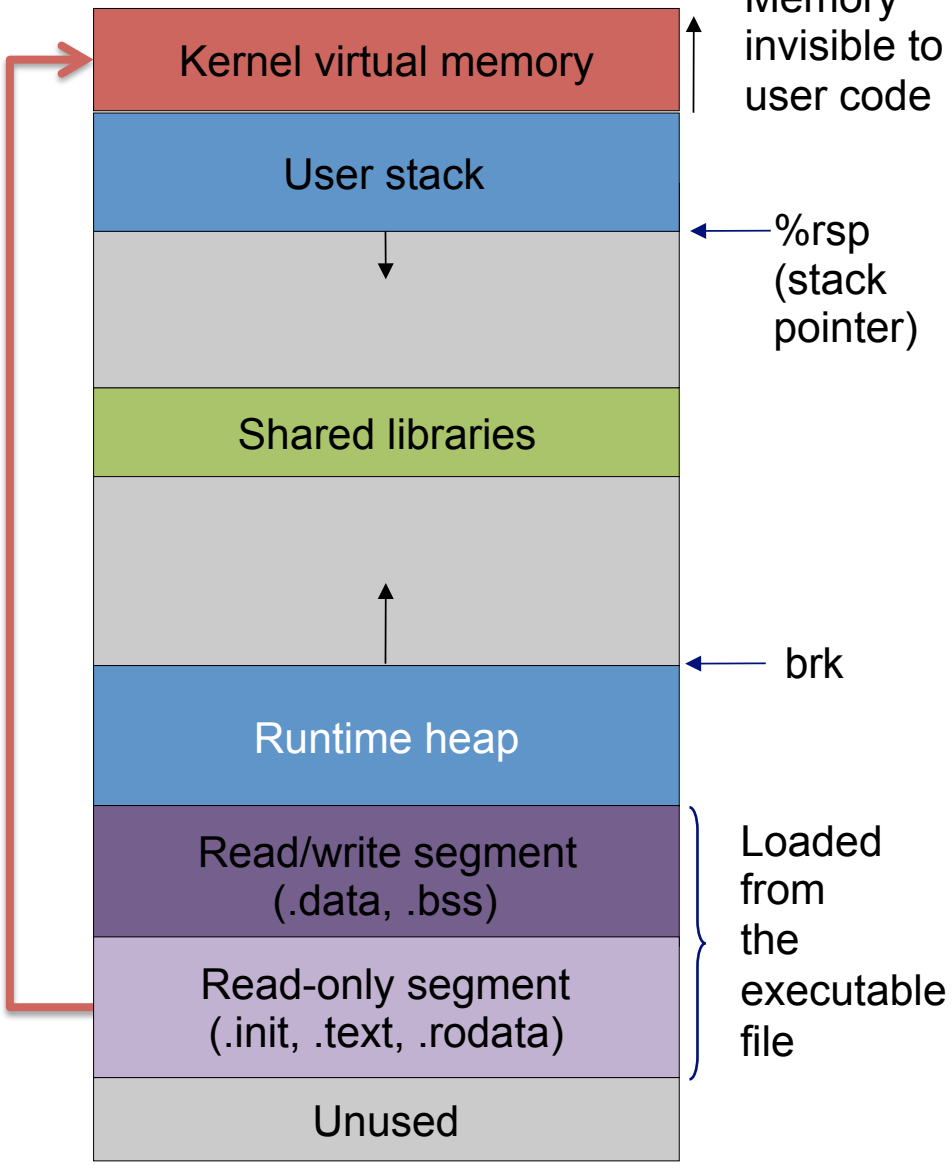


# Dynamic allocation on heap

Question: How to allocate memory on heap?

OS is responsible for heap.



# Dynamic allocation on heap

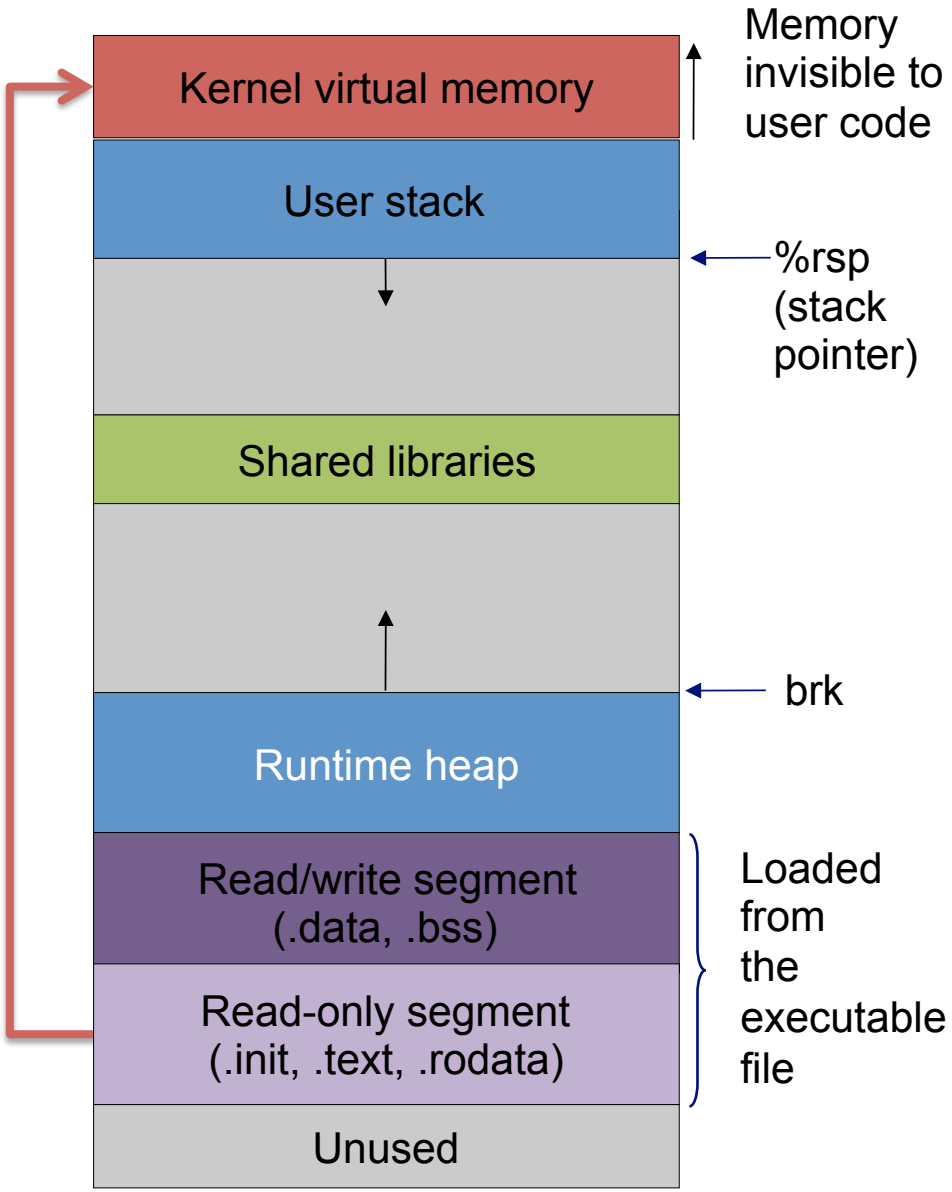
Question: How to allocate memory on heap?

OS is responsible for heap
– System calls

```
void *sbrk(intptr_t size);
```

sbrk

It increases the top of heap by “size” and returns a pointer to the base of new storage. The “size” can be a negative number.



# Dynamic allocation on heap

Question: How to allocate memory on heap?

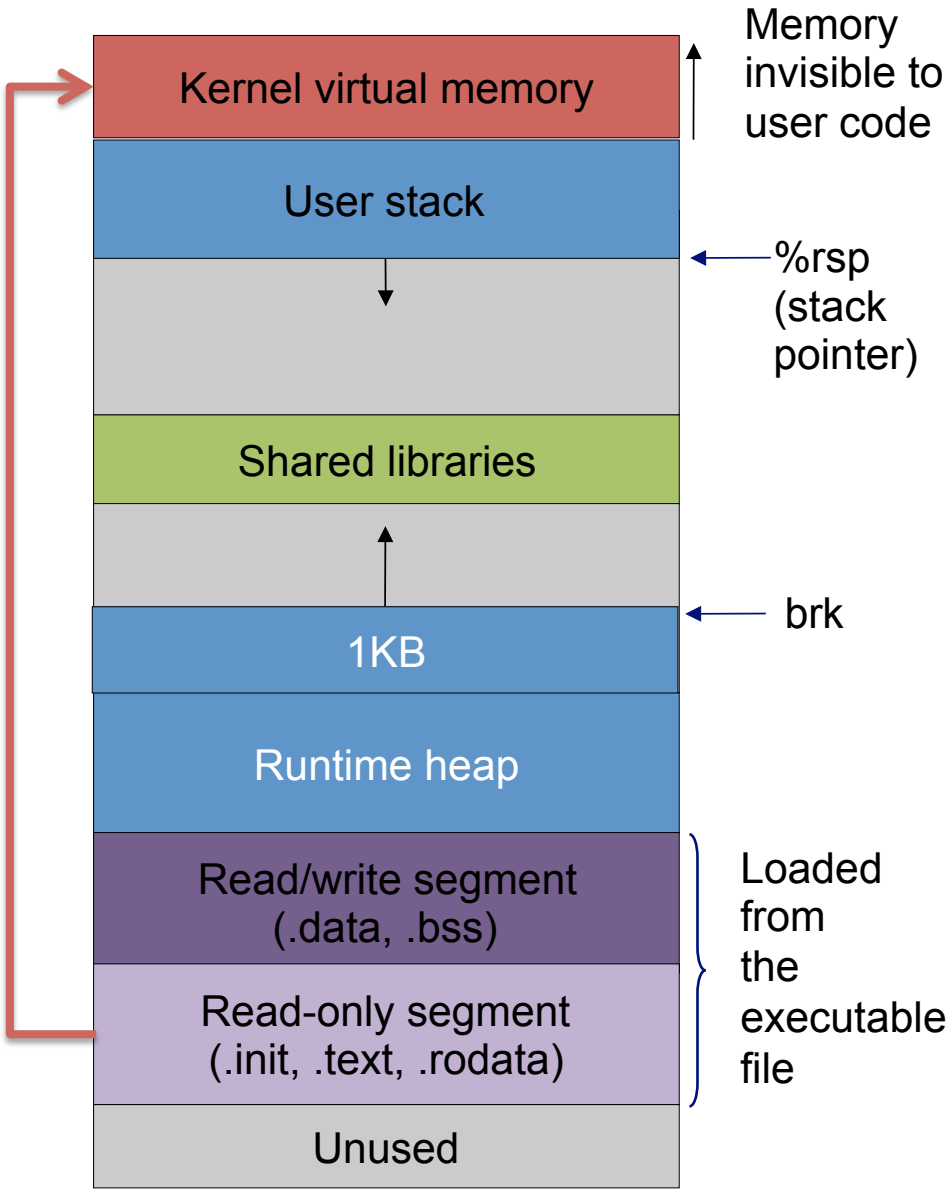
OS is responsible for heap
– System calls

```
void *sbrk(intptr_t size);
```

sbrk

It increases the top of heap by “size” and returns a pointer to the base of new storage. The “size” can be a negative number.

```
p = sbrk(1024) //allocate 1KB
```



# Dynamic allocation on heap

Question: How to allocate memory on heap?

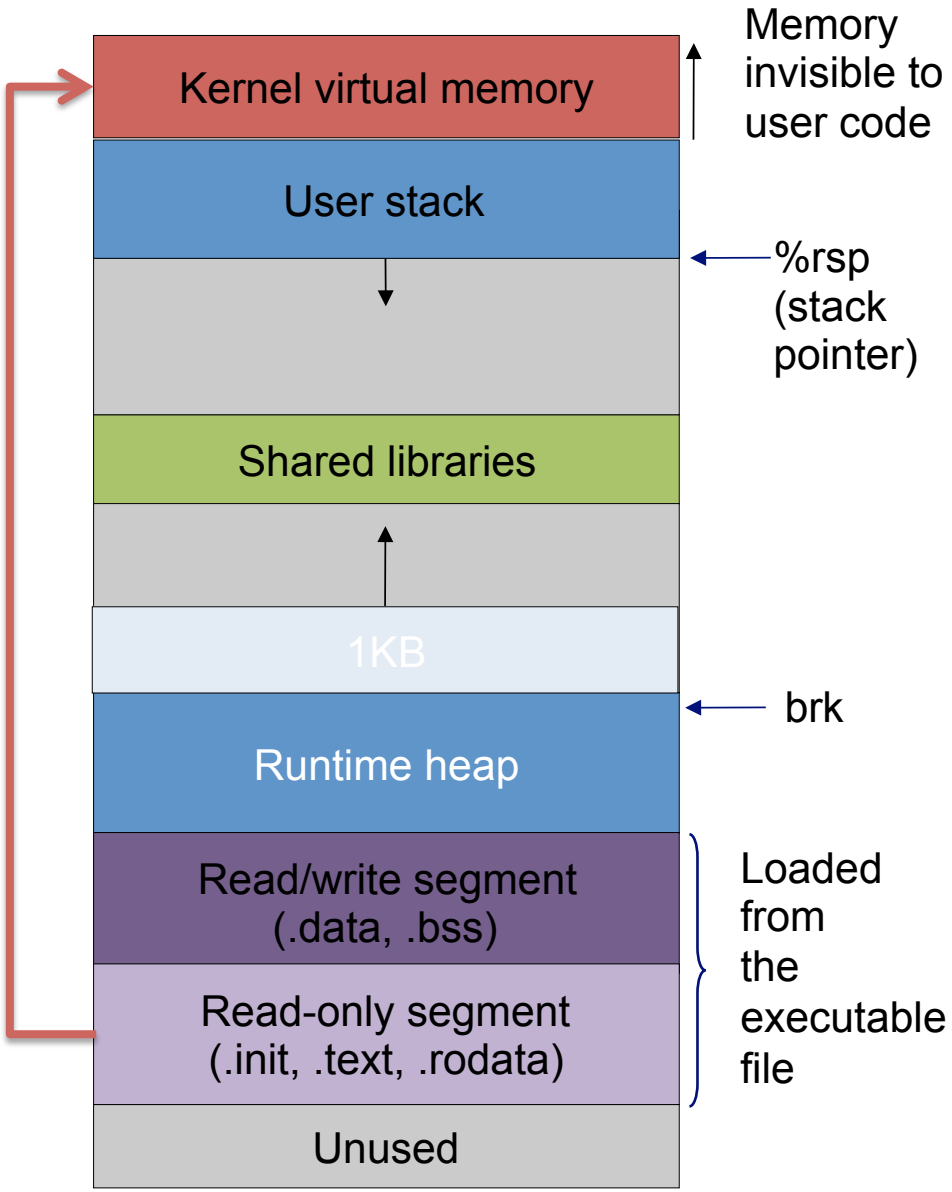
OS is responsible for heap
– System calls

```
void *sbrk(intptr_t size);
```

sbrk

It increases the top of heap by “size” and returns a pointer to the base of new storage. The “size” can be a negative number.

```
p = sbrk(1024) //allocate 1KB
sbrk(-1024) //free p
```



# Dynamic allocation on heap

Question: How to allocate memory on heap?

OS is responsible for heap
– System calls

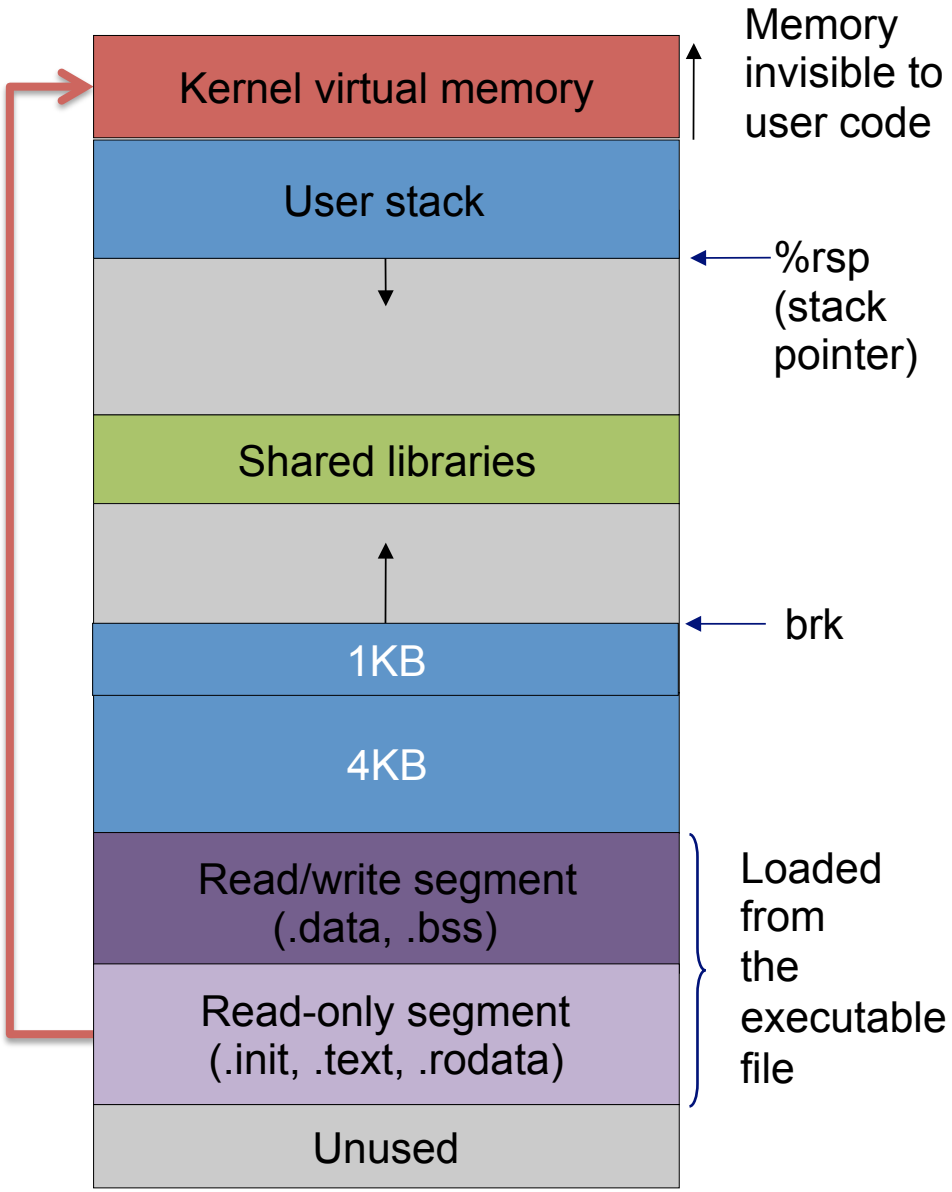
```
void *sbrk(intptr_t size);
```

sbrk

Issue I – can only free the memory on the top of heap

```
p1 = sbrk(1024) //allocate 1KB
p2 = sbrk(2048) //allocate 4KB
```

How to free p1?



# Dynamic allocation on heap

Question: How to allocate memory on heap?

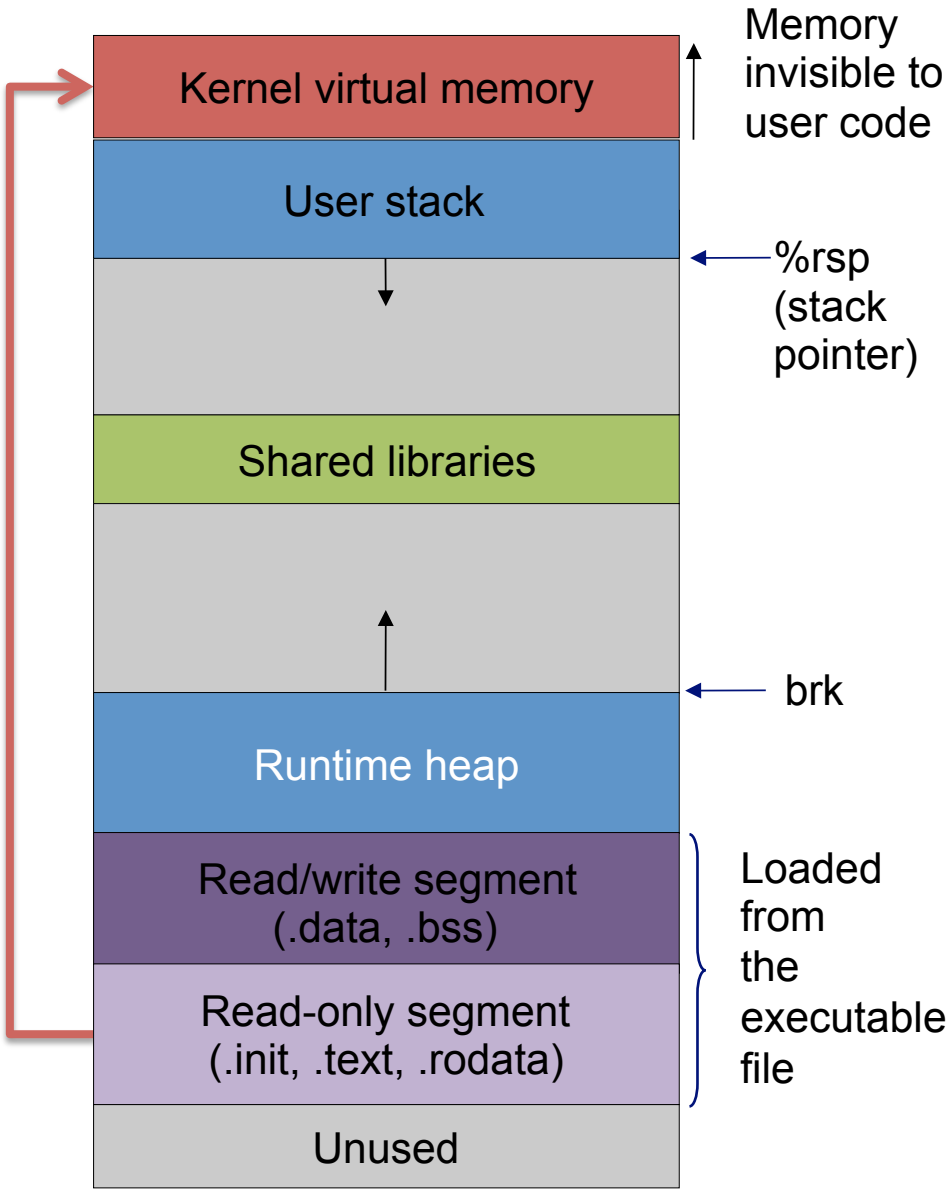
OS is responsible for heap
– System calls

```
void *sbrk(intptr_t size);
```

sbrk

Issue I – can only free the memory on the top of heap

Issue II – system call has high performance cost > 10X

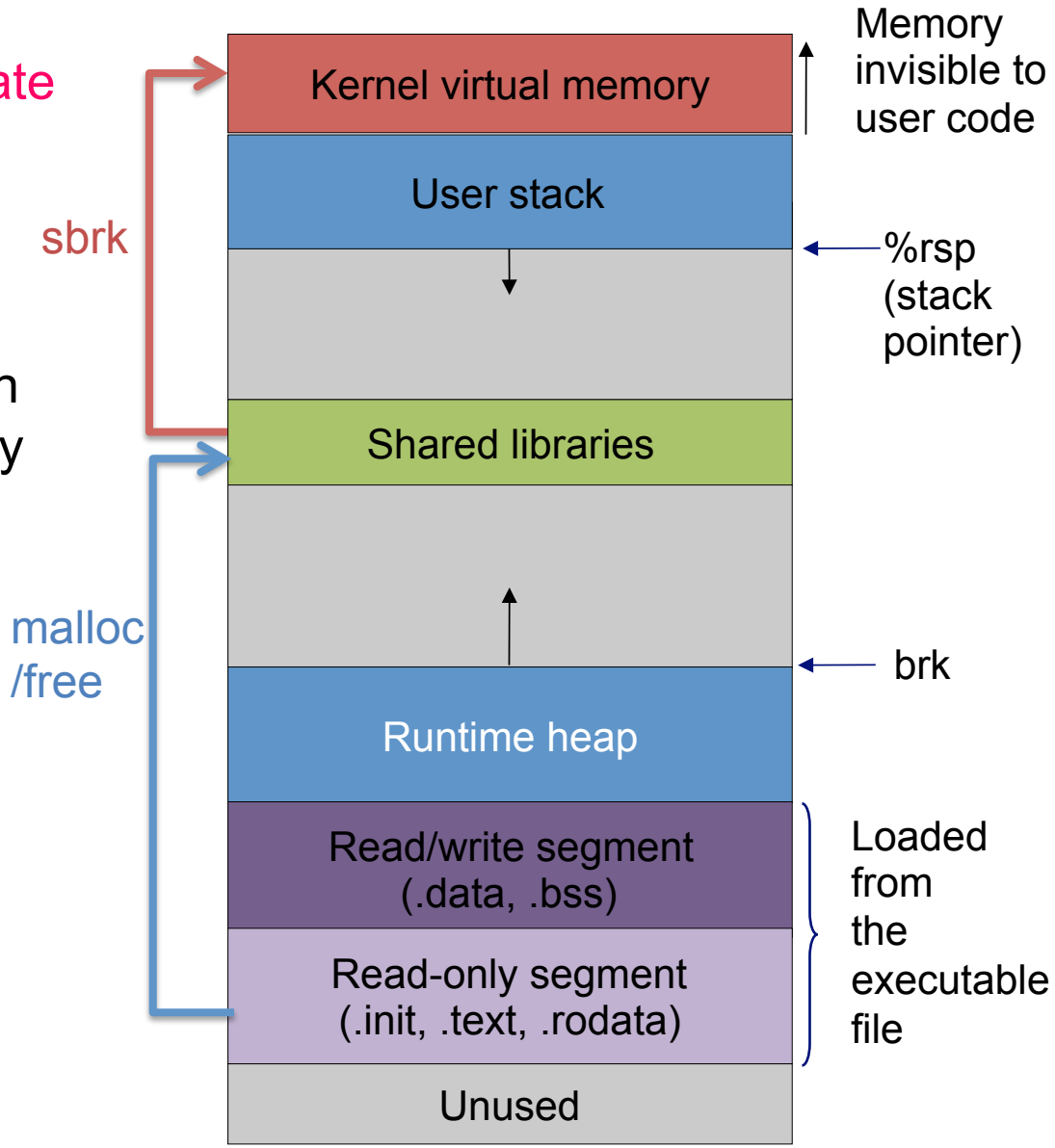


# Dynamic allocation on heap

Question: How to effciently allocate memory on heap?

Basic idea – request a large of memory region from heap once, then manage this memory region by itself. → allocator in the library

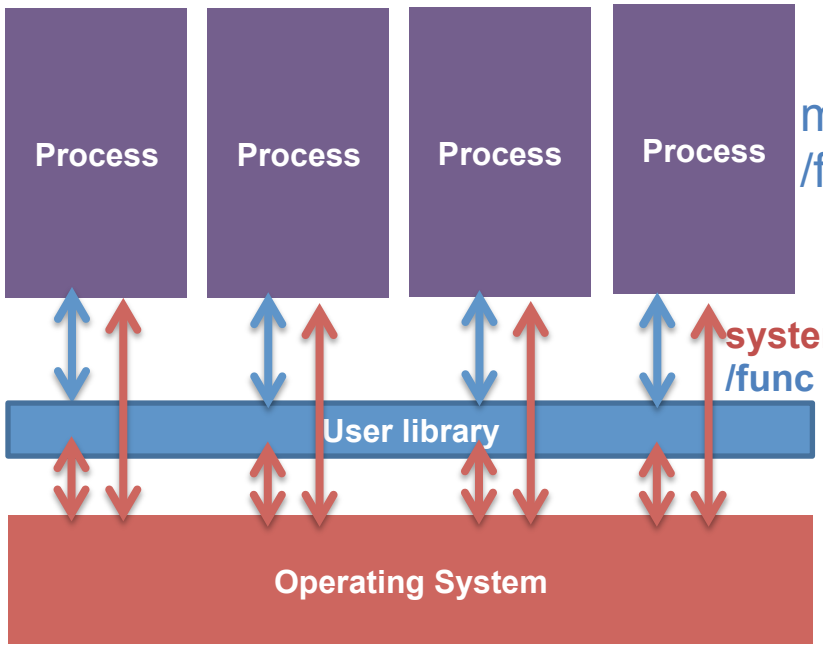
Memory management in the user space.

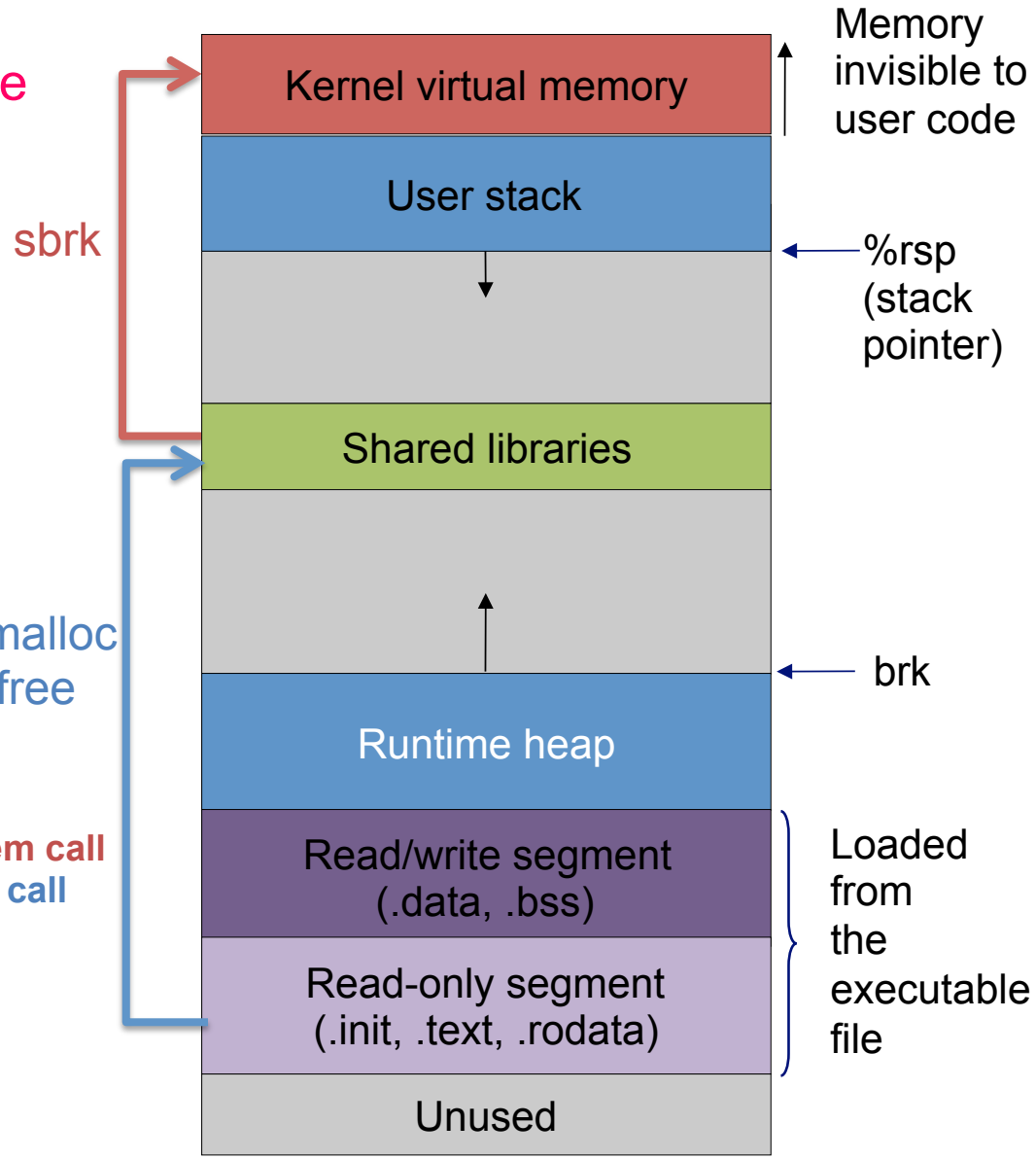


# Dynamic allocation on heap

Question: How to effciently allocate memory on heap?

Basic idea – request a large of memory region from heap once, then manage this memory region by itself. → allocator in the library



# Memory Allocator

Assumption in this lecture

- At the beginning, the allocator requests enough memory with `sbrk`

Goal

- Highly utilize the acquired memory with high throughput
  - high throughput – how many mallocs / frees can be done per second
  - high utilization – fraction of allocated size / total heap size

# Memory Allocator

Assumed behavior of applications

- Can issue arbitrary sequence of malloc (for arbitary sizes)/free
- The argument of free must be the return value of a previous malloc
- No double free

Requirements of allocator

- Must allocate from free memory (correctness)
- Once allocated, cannot be moved

# Questions

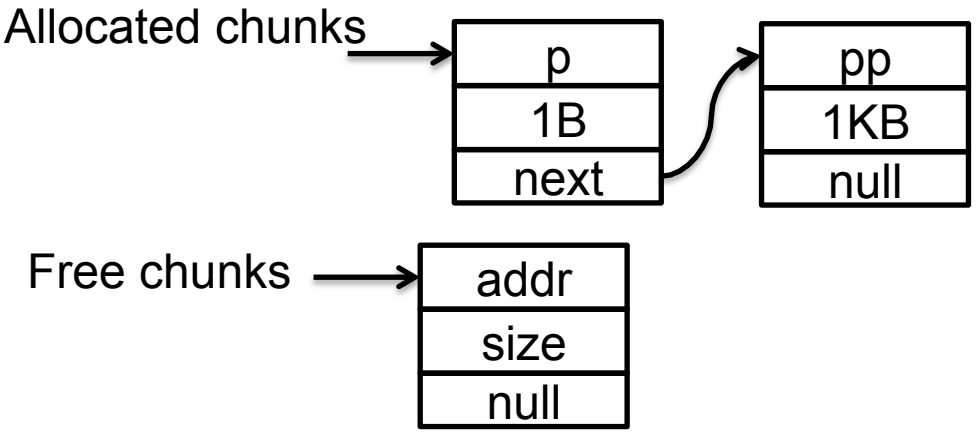
1. How to keep track which bytes are free and which are not?

2. Which of the free chunks to allocate?

3. free only supplies a pointer, how to know its corresponding chunk size?

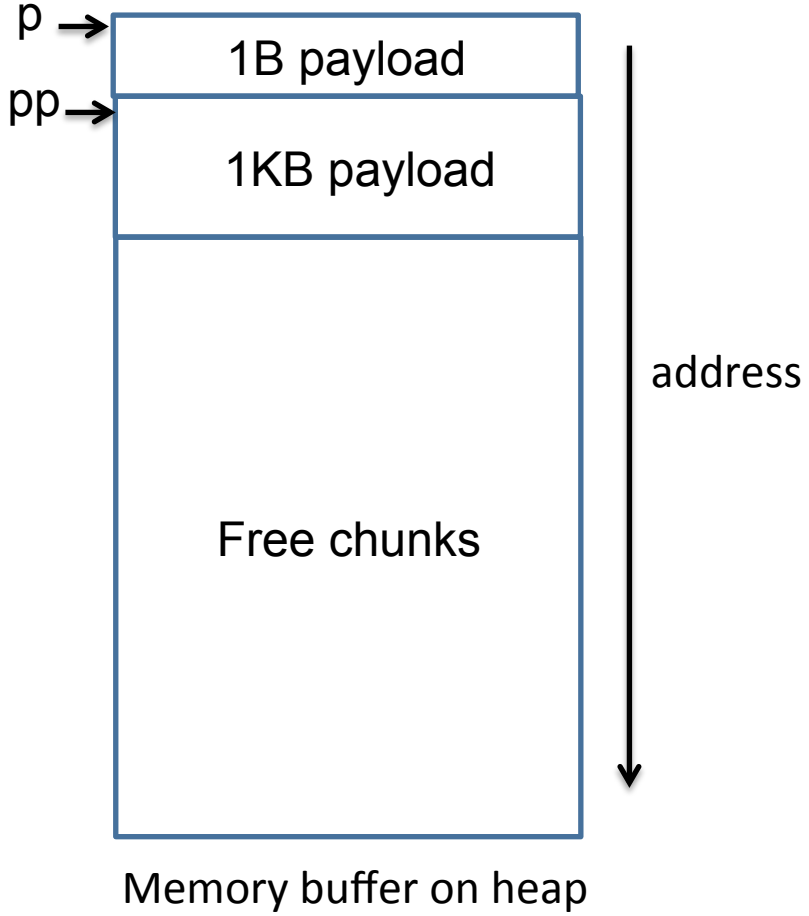
# Track allocated/free chunks

Keep the metadata (addr, size) in a linked list



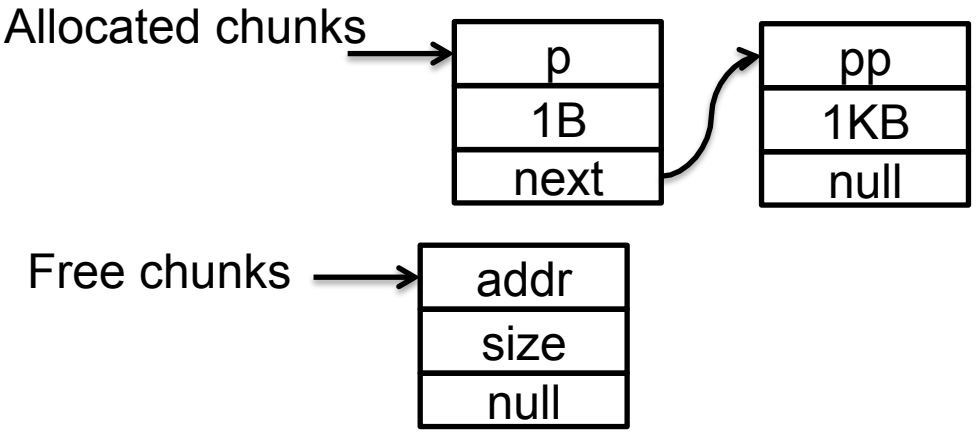

```
p  = malloc(1)
pp = malloc(1024)
```

What is the size of each linked list entry?

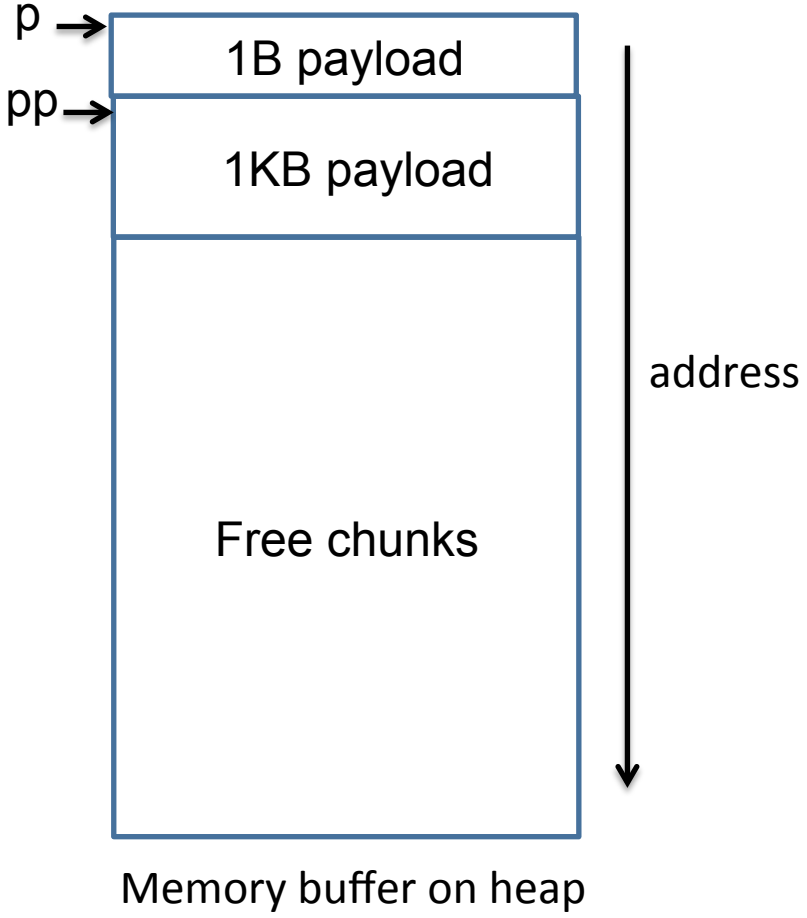


# Track allocated/free chunks

Keep the metadata (addr, size) in a linked list




```
p  = malloc(1)
pp = malloc(1024)
```




What is the size of each linked list entry? >= 20B

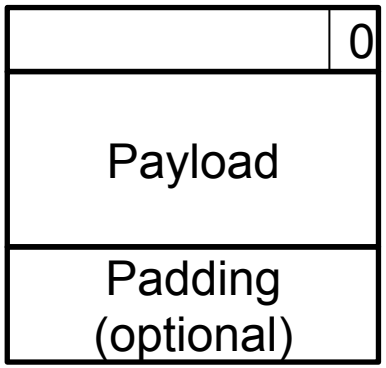
Can we further reduce the size of metadata?

# Implicit list

Embed the metadata in the chunks (blocks)

- Each block has a one-word (8 bytes) header
- Block is double-word (16 bytes) alignment

  → Size is multiple of 16




header (8 bytes)

allocated: header & 0x1
size: header & ~(0x1)

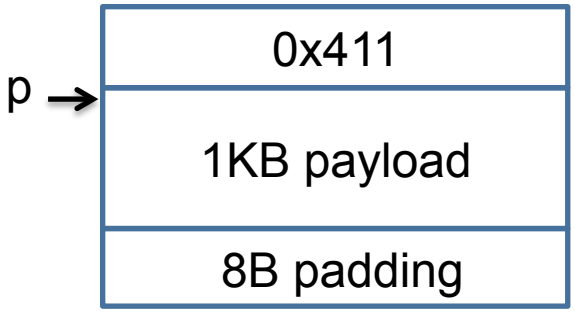
# Implicit list

Embed the metadata in the chunks (blocks)

- Each block has a one-word (8 bytes) header
- Block is double-word (16 bytes) alignment

  → Size is multiple of 16

```
p  = malloc(1024)
```



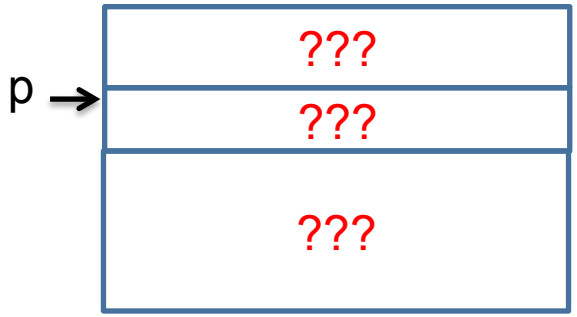
# Implicit list

Embed the metadata in the chunks (blocks)

- Each block has a one-word (8 bytes) header
- Block is double-word (16 bytes) alignment

  → Size is multiple of 16

```
p  = malloc(1)
```



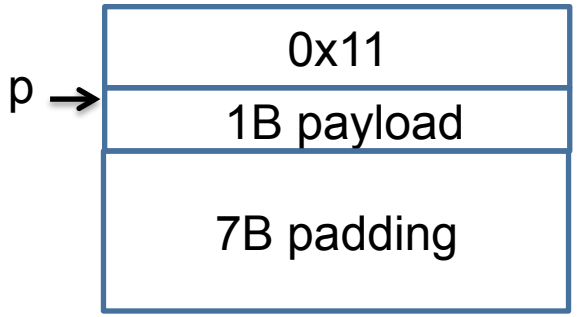
# Implicit list

Embed the metadata in the chunks (blocks)

- Each block has a one-word (8 bytes) header
- Block is double-word (16 bytes) alignment

  → Size is multiple of 16

```
p  = malloc(1)
```



# Exercises

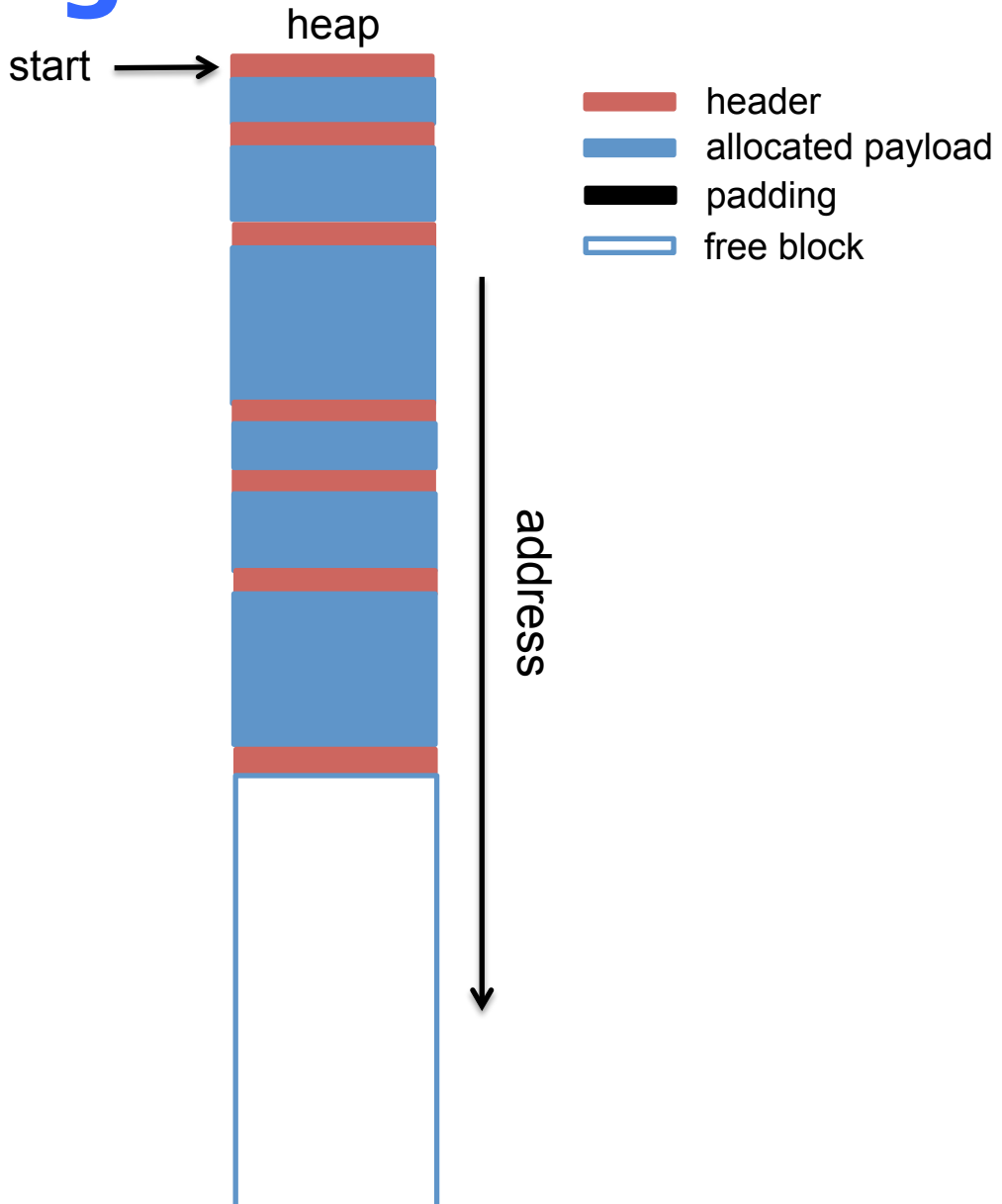
| Alignment | Request | Block size | Header (hex) |
|---|---|---|---|
| 8 bytes | malloc(5) | | |
| 4 bytes | malloc(13) | | |
| 16 bytes | malloc(20) | | |
| 8 bytes | malloc(3) | | |

# Exercises

| Alignment | Request | Block size | Header (hex) |
|---|---|---|---|
| 8 bytes | malloc(5) | 16 | 0x11 |
| 4 bytes | malloc(13) | 24 | 0x19 |
| 16 bytes | malloc(20) | 32 | 0x21 |
| 8 bytes | malloc(3) | 16 | 0x11 |

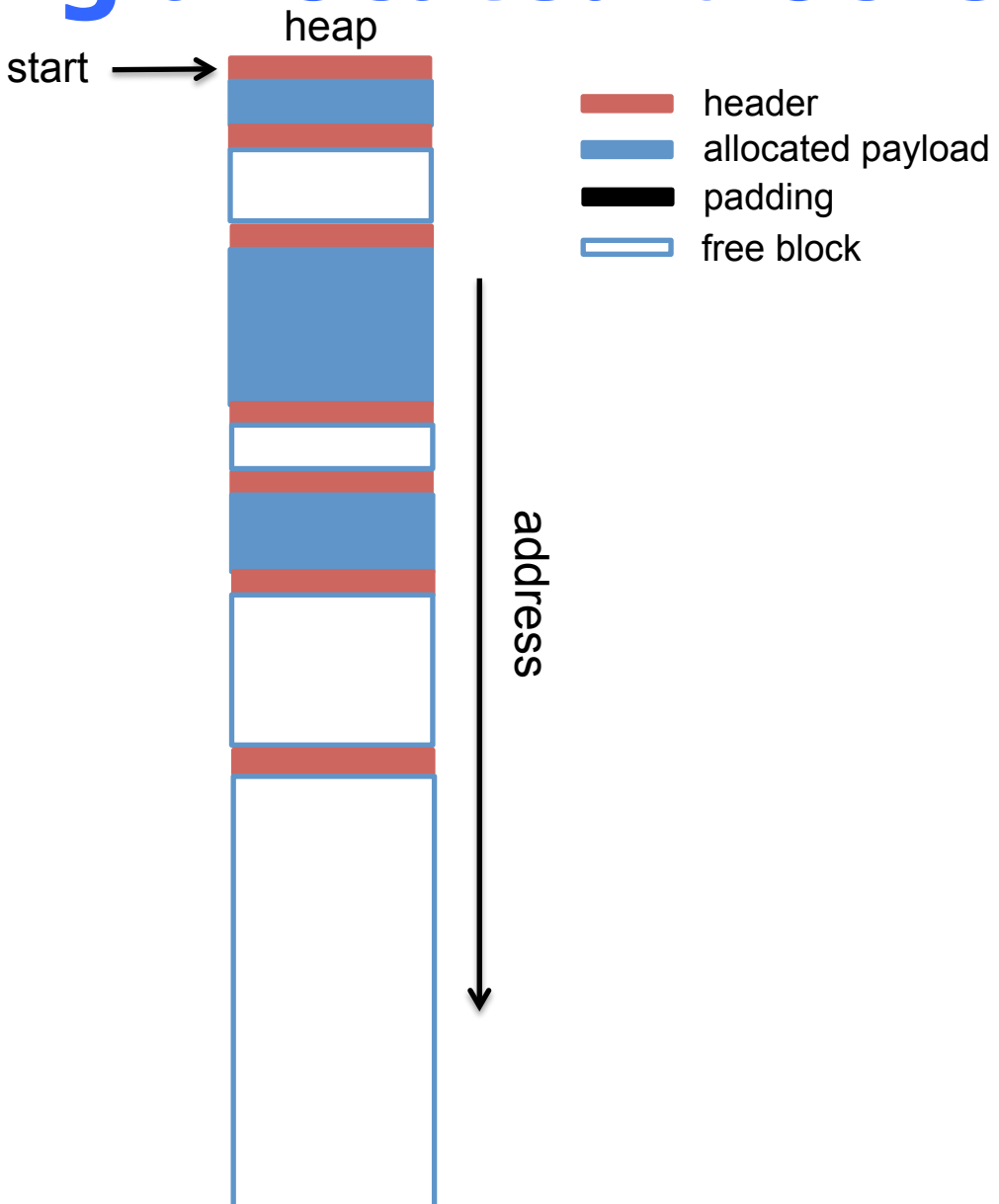
# Placing allocated blocks

```
p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
```



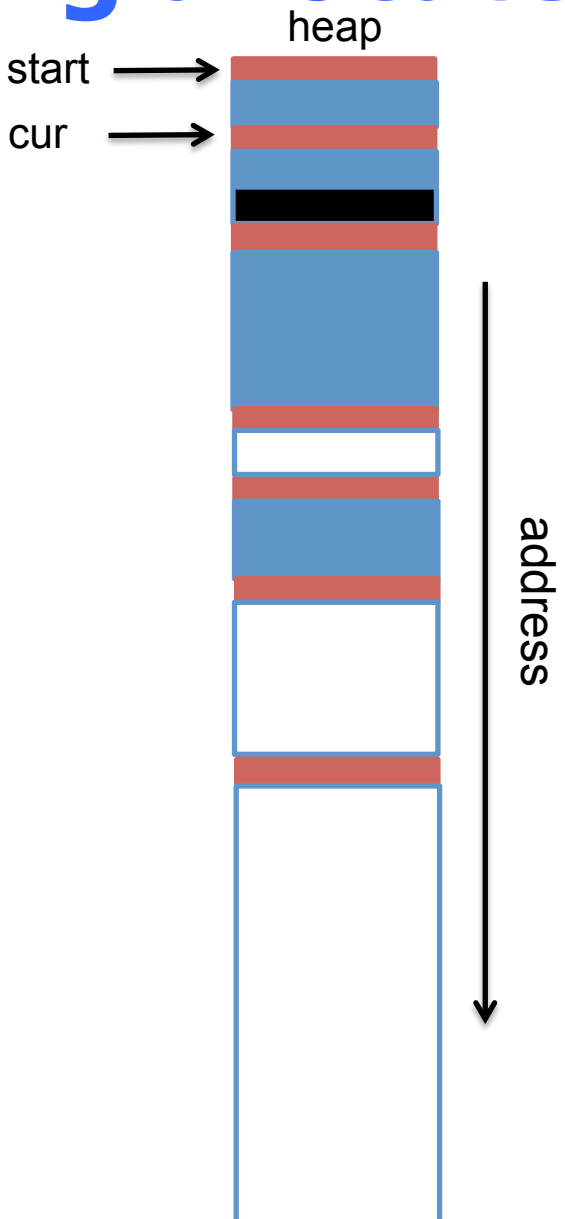
# Placing allocated blocks

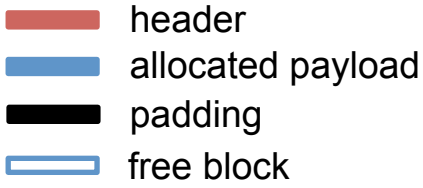
```
p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
```



# Placing allocated blocks

```
p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
p7 = malloc(8)
```

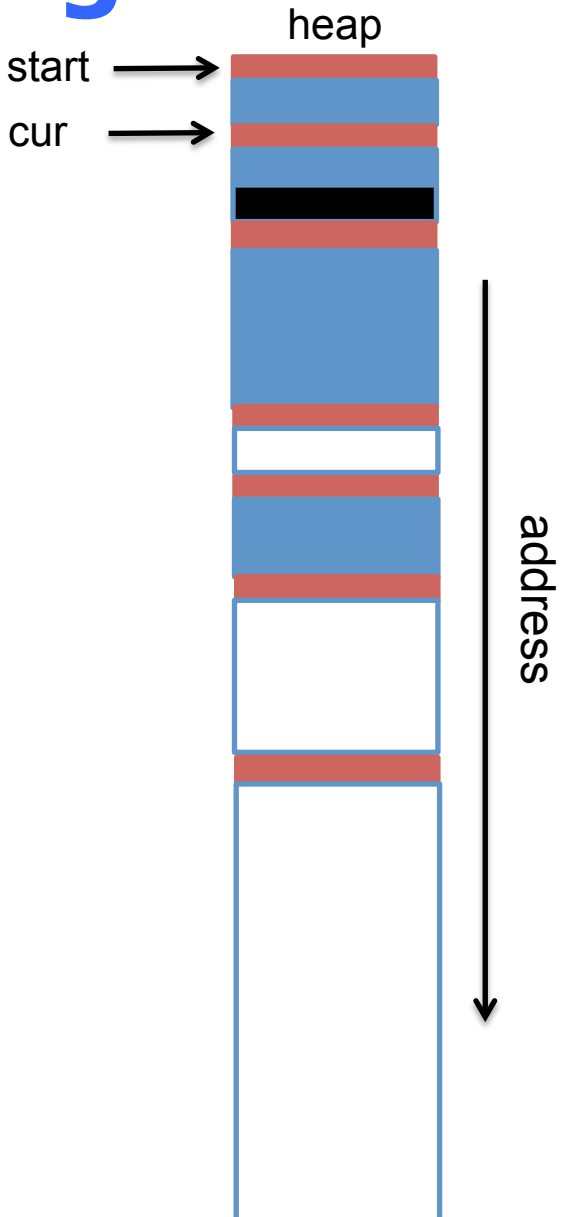


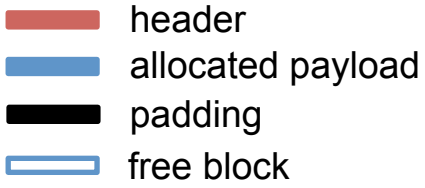




First fit – Search list from beginning, choose first free block that fits

# Placing allocated blocks

```
p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
p7 = malloc(8)
```



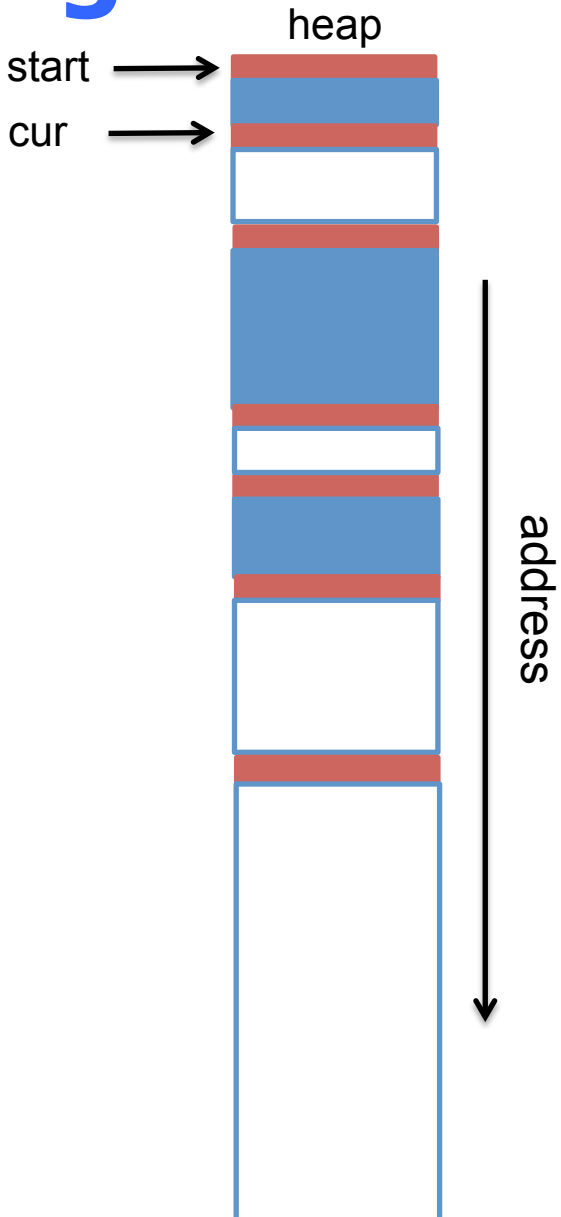



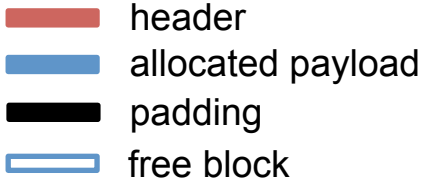

First fit – Search list from beginning, choose first free block that fits

Issue: cause “splinters/fragments” at beginning of the buffer

# Placing allocated blocks

```
p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
p7 = malloc(8)
```

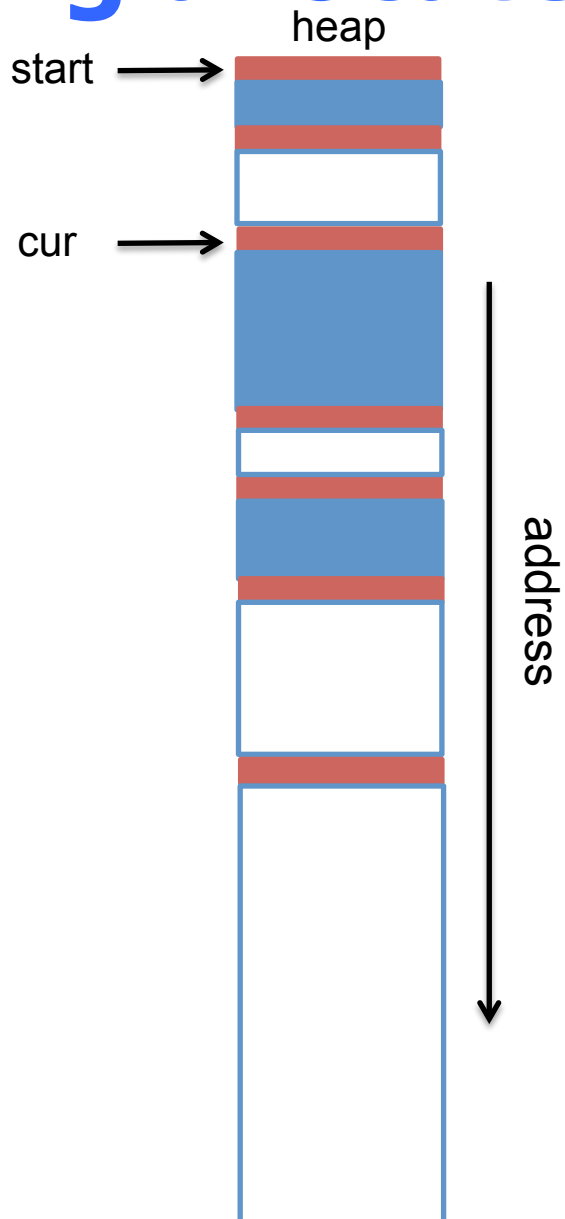


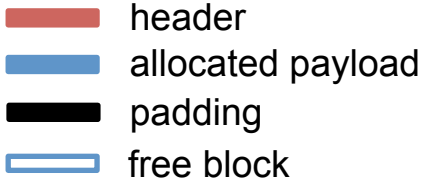




Best fit – choose the free block with the closest size that fits

# Placing allocated blocks

```
p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
p7 = malloc(8)
```

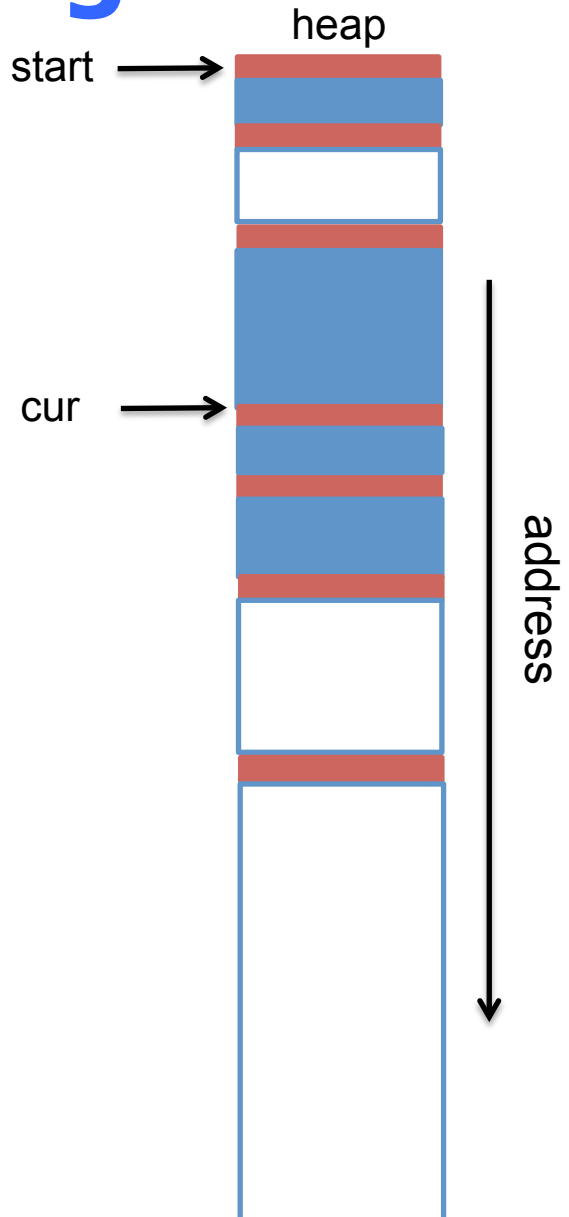


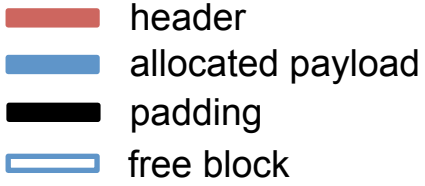




Best fit – choose the free block with the closest size that fits

# Placing allocated blocks

```
p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
p7 = malloc(8)
```



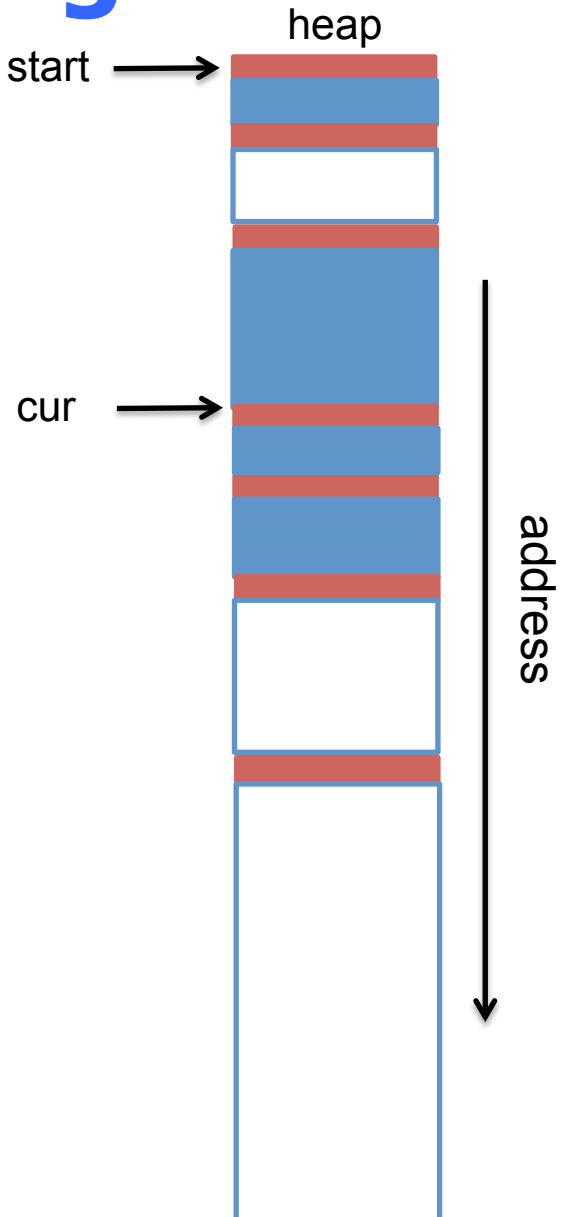



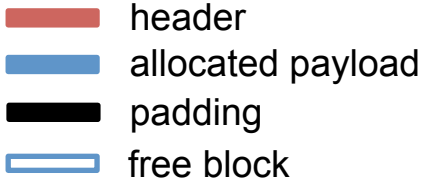

Best fit – Search list from beginning choose first free block that fits

Best fit keeps fragments small, but typically run slower than first fit.

# Placing allocated blocks

```
p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
p7 = malloc(8)
p8 = malloc(56)
```

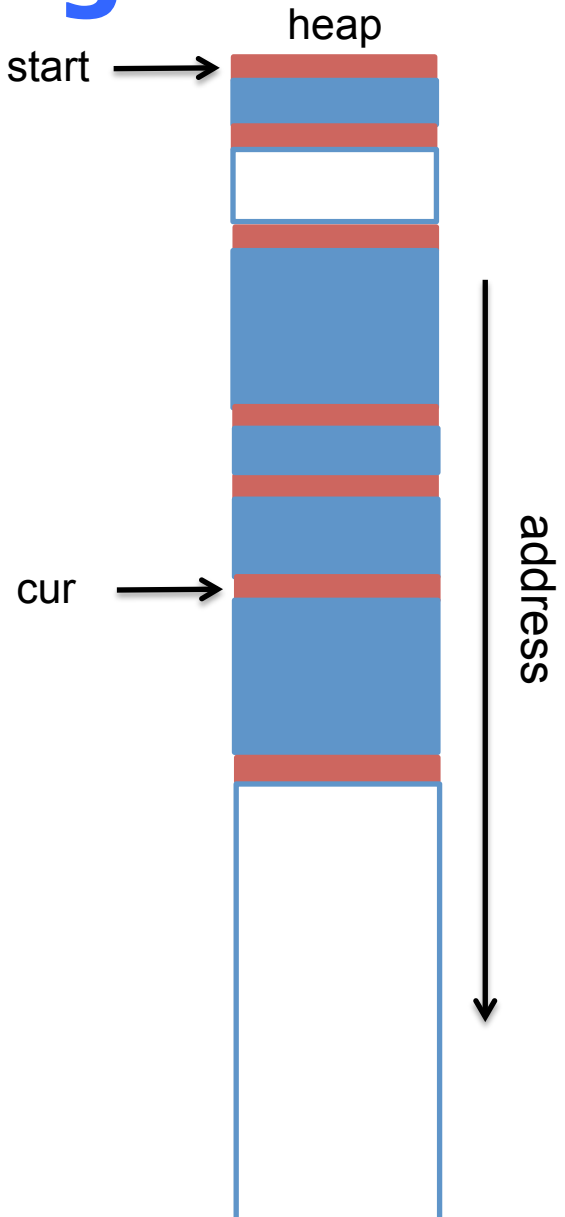


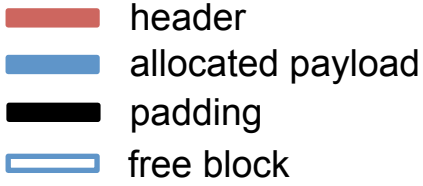




Next fit – like first-fit, but search list from the location where previous search left off.

# Placing allocated blocks

```
p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
p7 = malloc(8)
p8 = malloc(56)
```



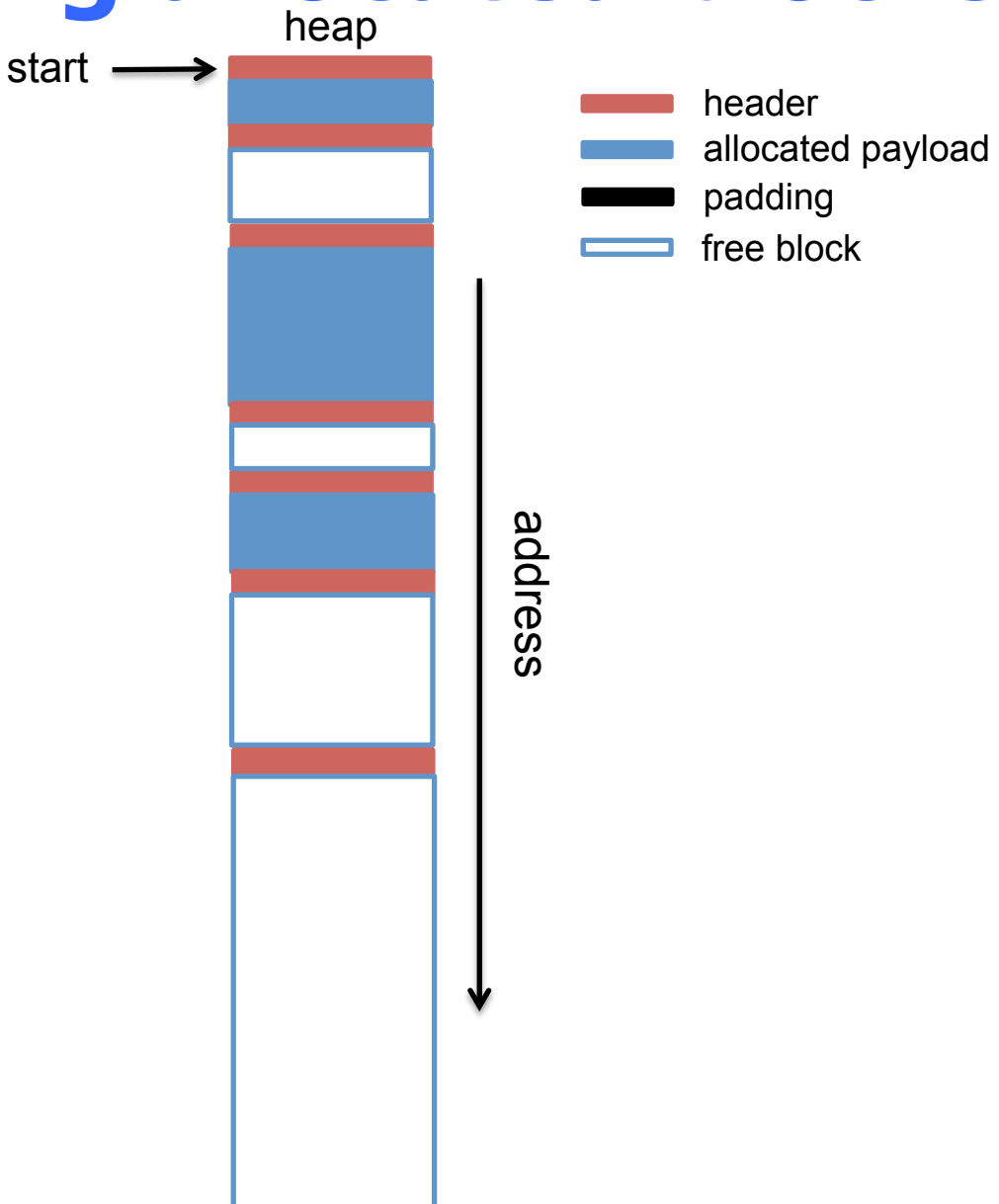




Next fit – like first-fit, but search list from from the location where previous search left off.

Next fit runs faster than first fit, but fragmentation is worse.

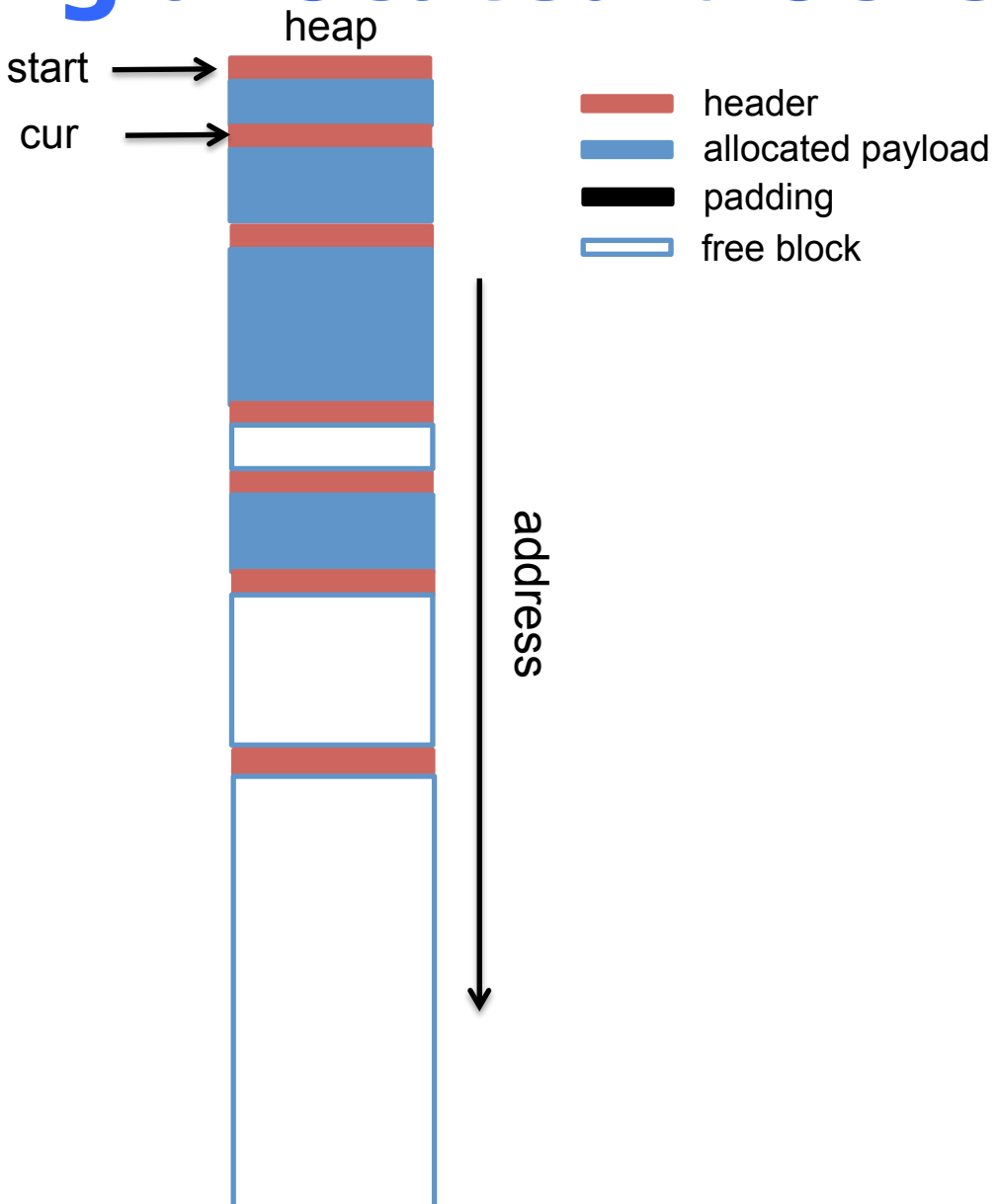
# Placing allocated blocks

```
p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
p7 = malloc(24)
p8 = malloc(24)
```



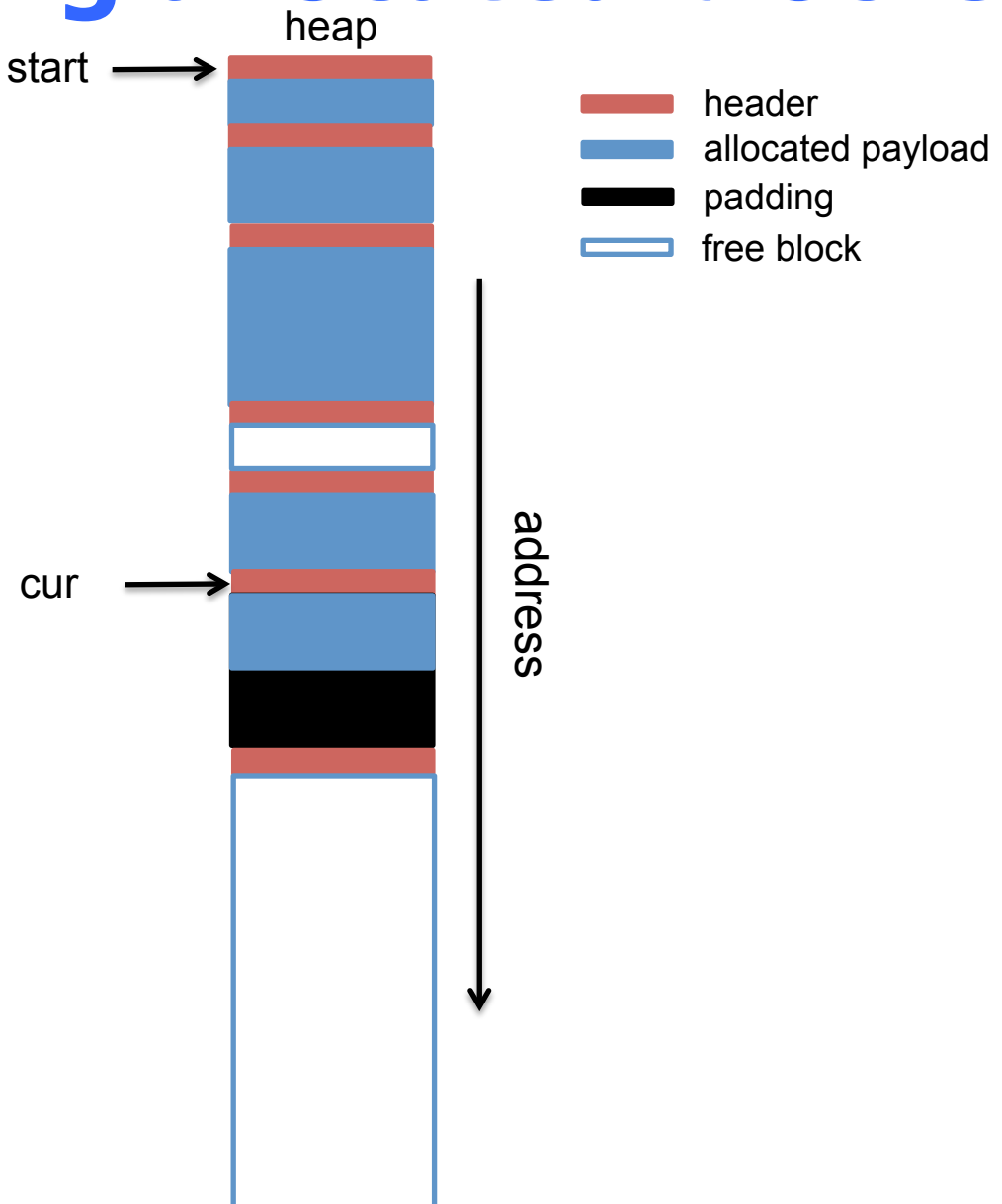
# Placing allocated blocks

```
p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
p7 = malloc(24)
p8 = malloc(24)
```



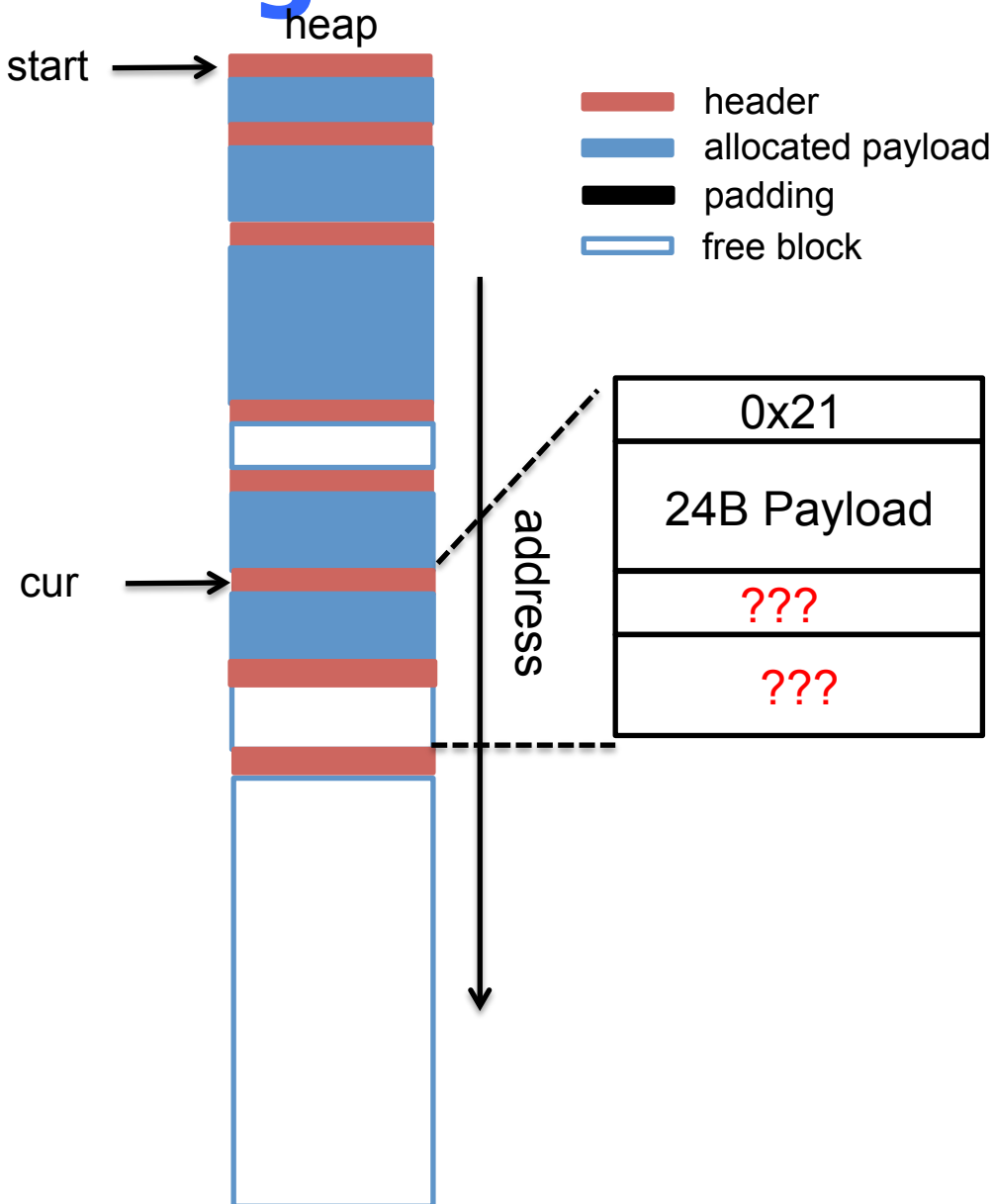
# Placing allocated blocks

```
p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
p7 = malloc(24)
p8 = malloc(24)
```



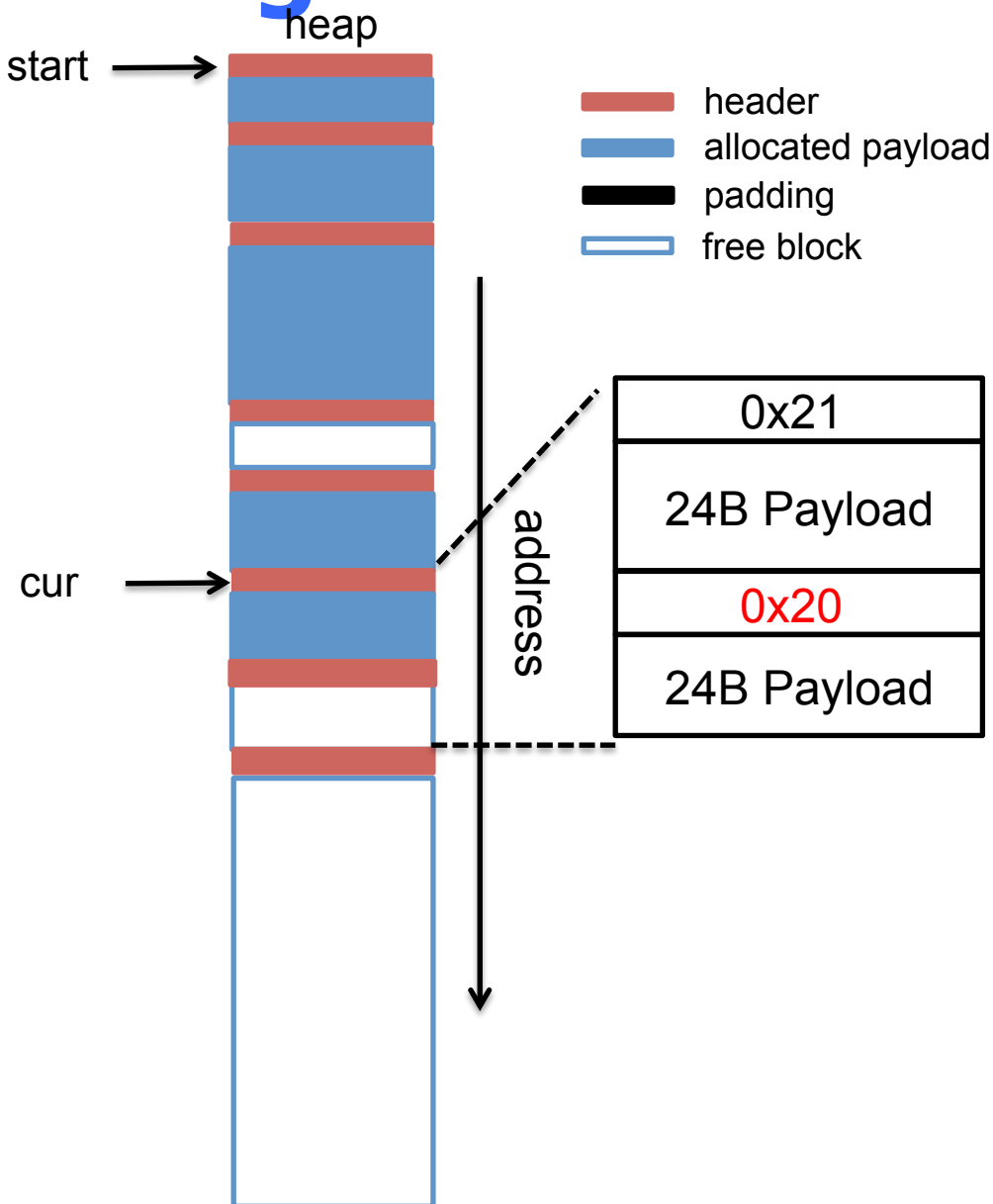
# Splitting free block

```
p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
p7 = malloc(24)
p8 = malloc(24)
```



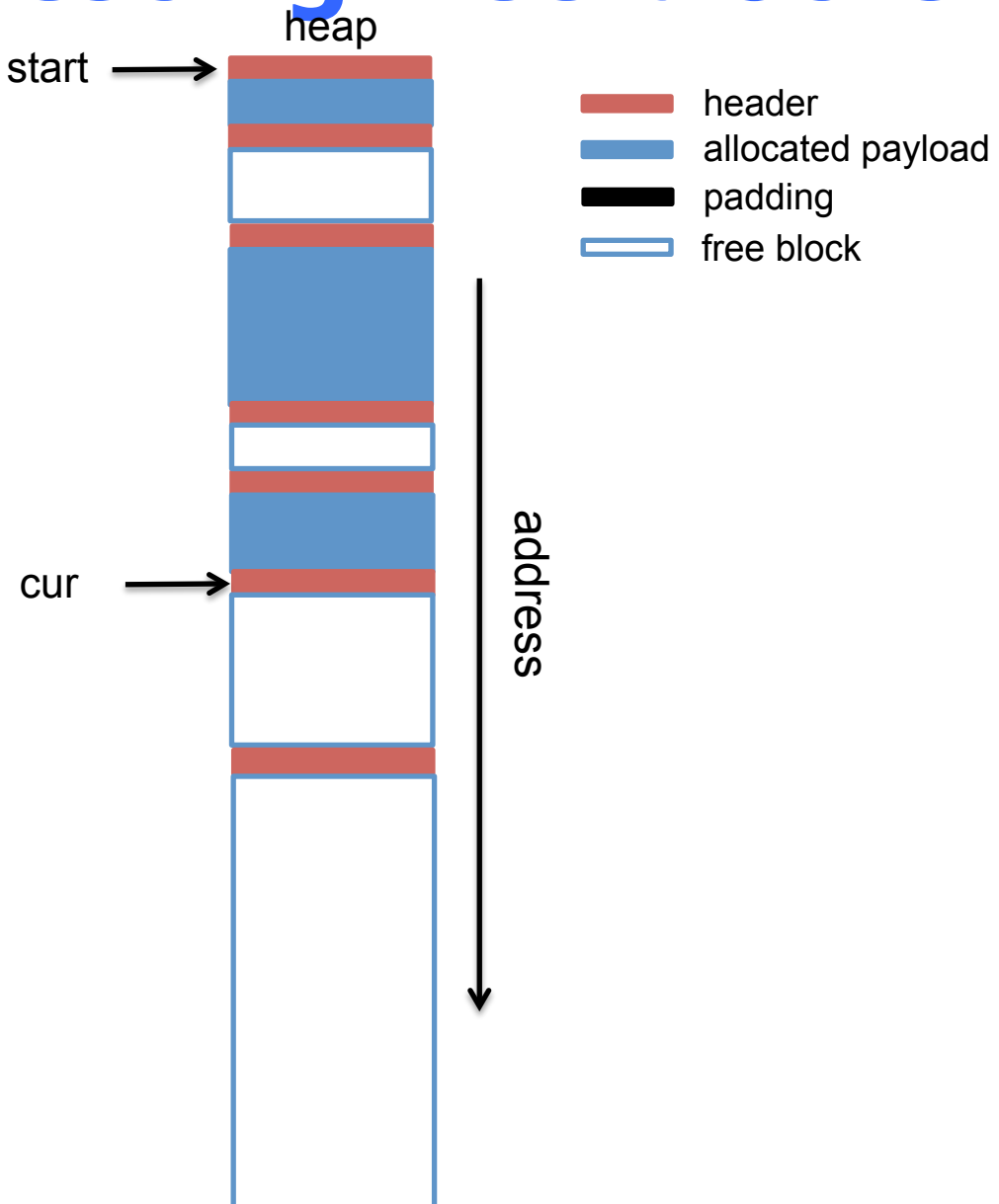
# Splitting free block

```
p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
p7 = malloc(24)
p8 = malloc(24)
```



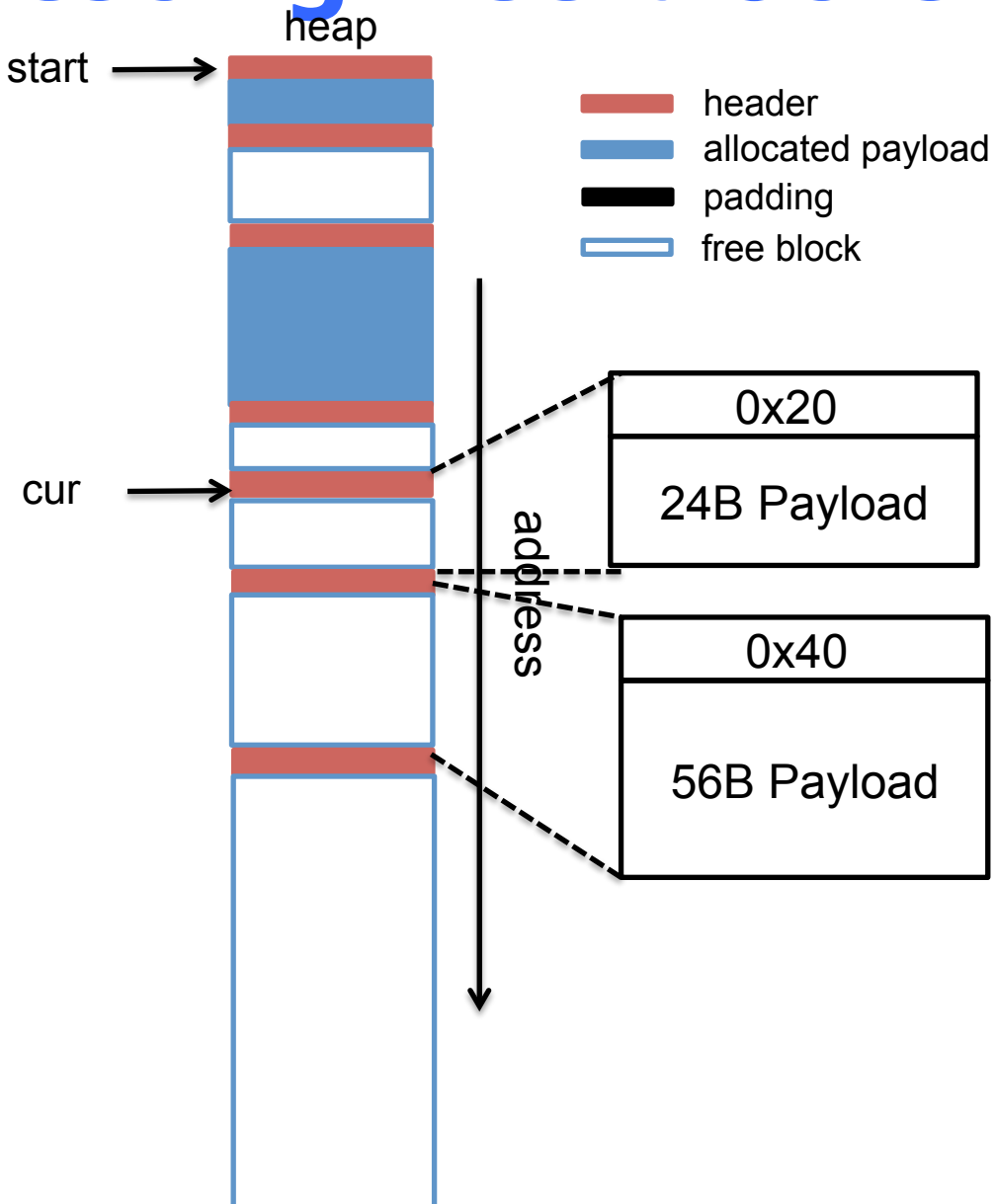
# Coalescing free blocks

```
p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
free(p5)
```



# Coalescing free blocks

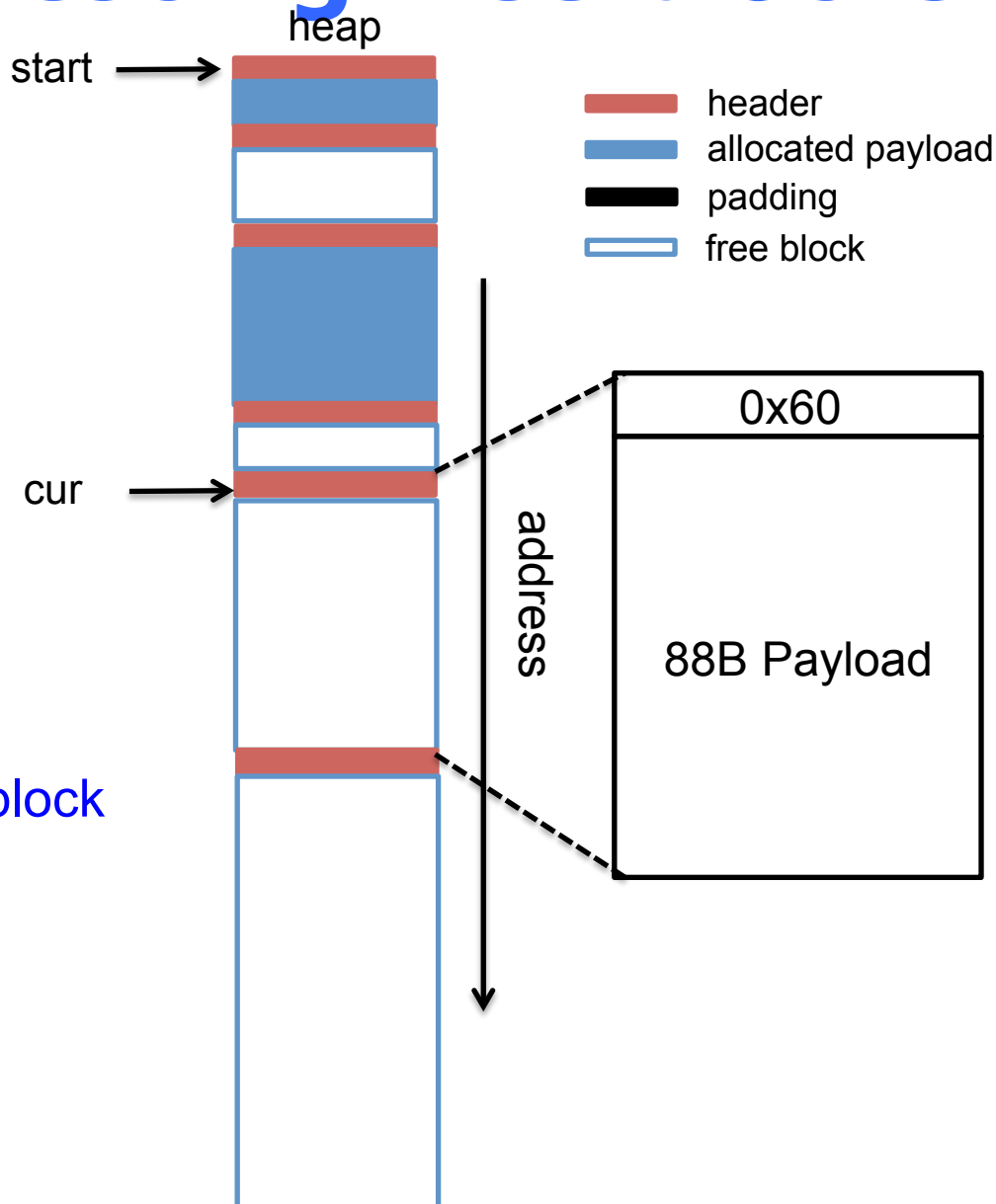
```
p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
free(p5)
```



# Coalescing free blocks

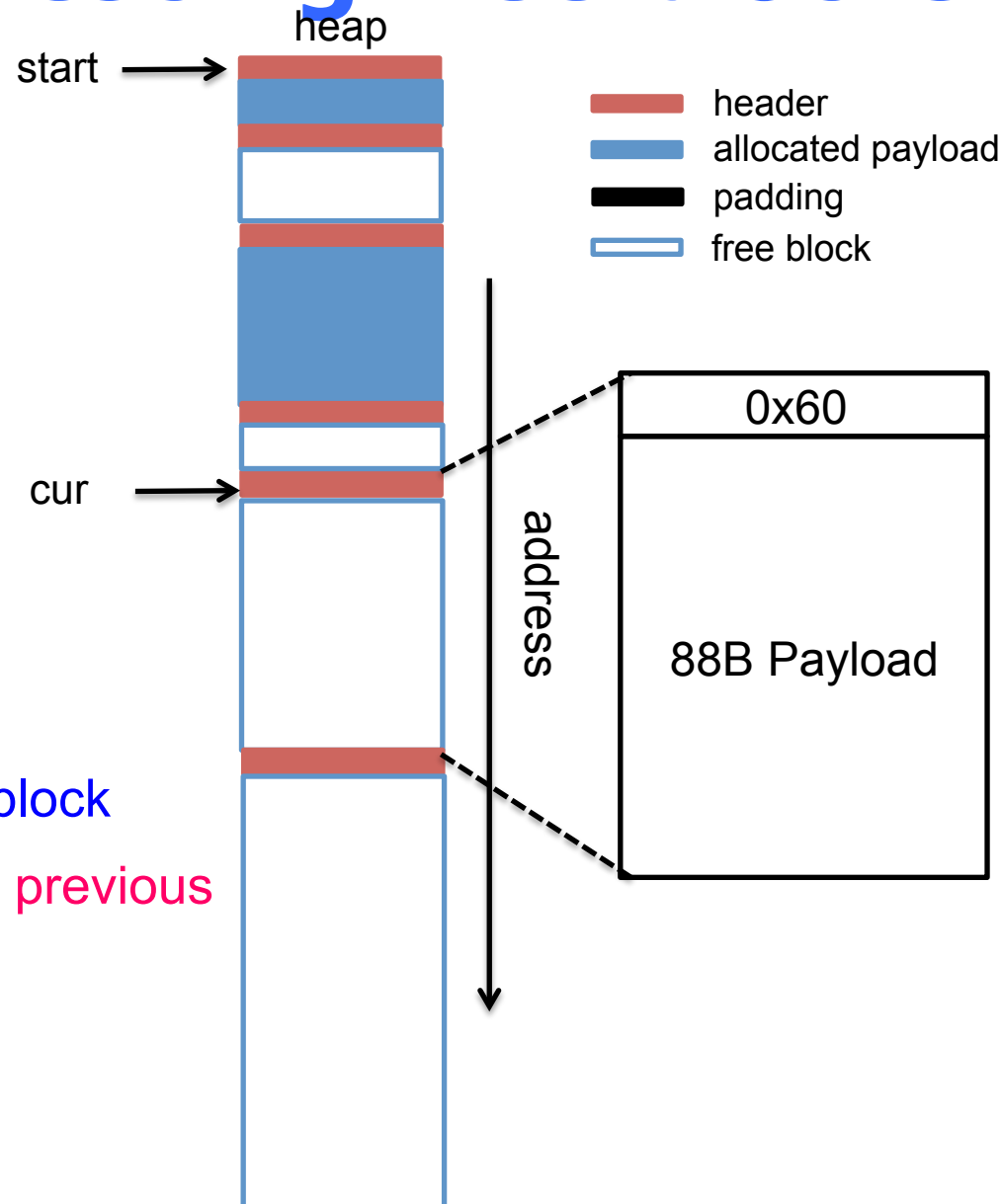
```
p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
free(p5)
```

Coalescing with next block



# Coalescing free blocks

```
p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
free(p5)
```

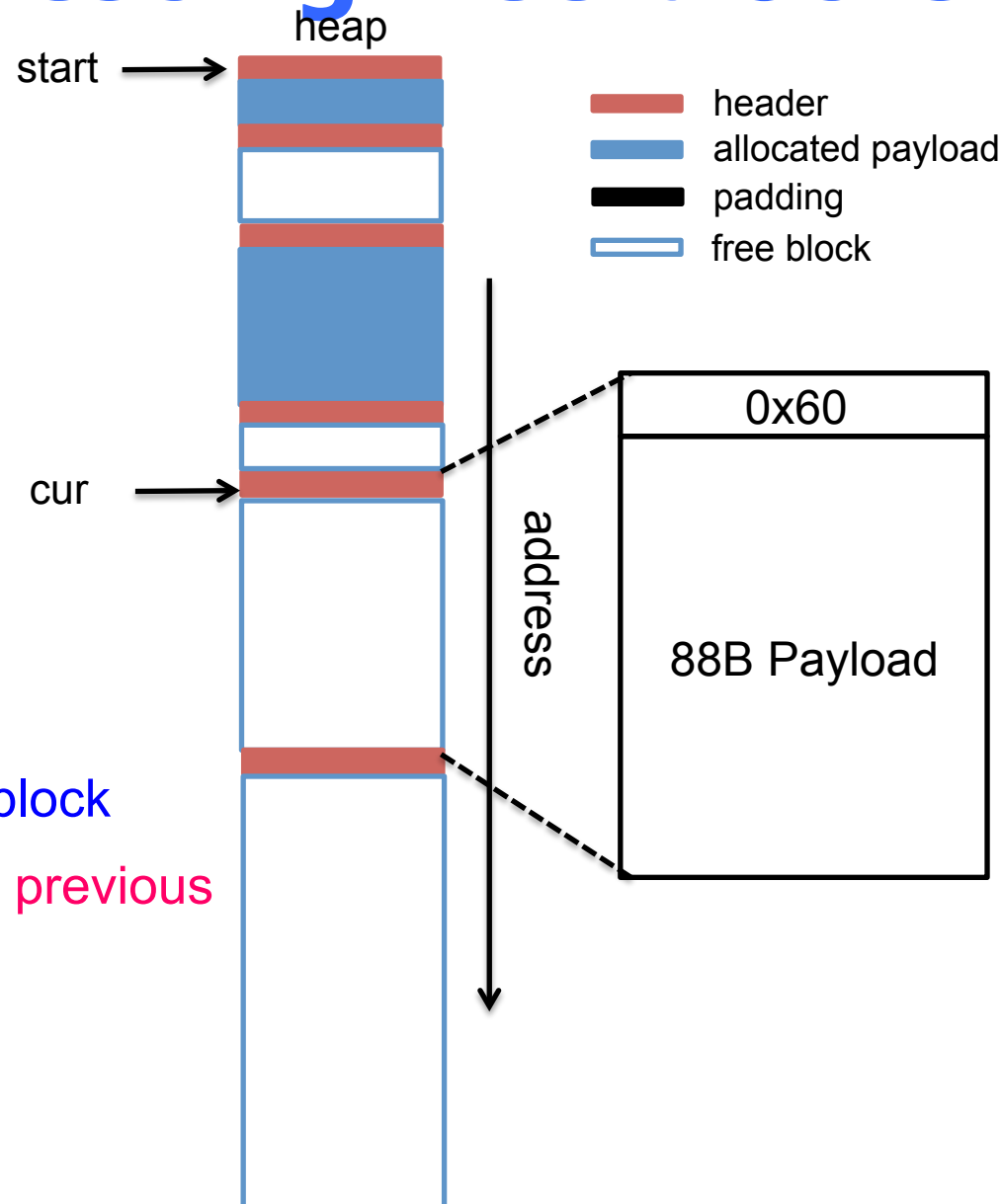



Coalescing with next block

How to coalesce with previous block?

# Coalescing free blocks

```
p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
free(p5)
```




Coalescing with next block

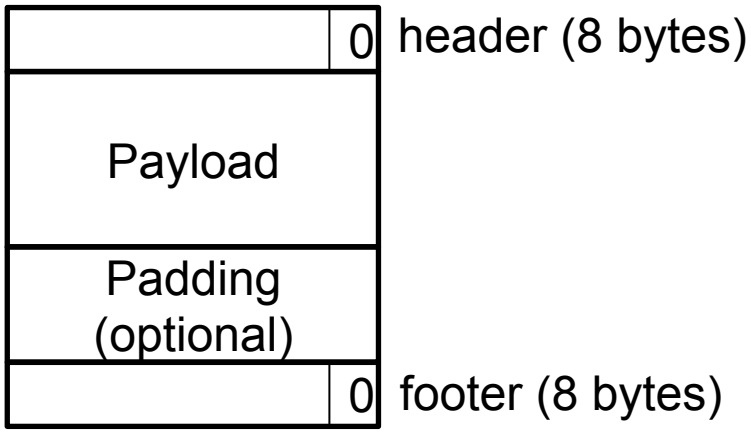
How to coalesce with previous block?

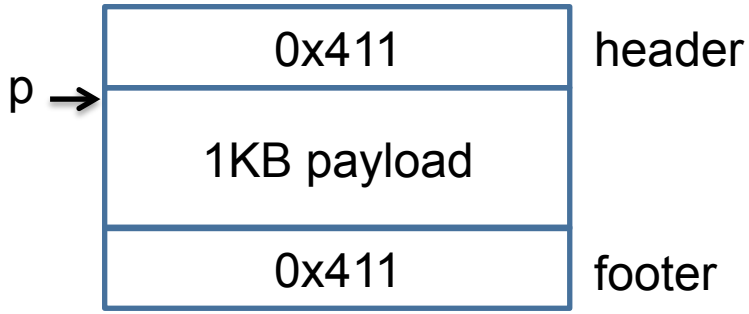
-- search from start?

# Coalescing free blocks

Embed the metadata in the chunks (blocks)

- Each block has a one-word (8 bytes) header and (8 bytes) footer
- Block is double-word (16 bytes) alignment
  → Size is multiple of 16

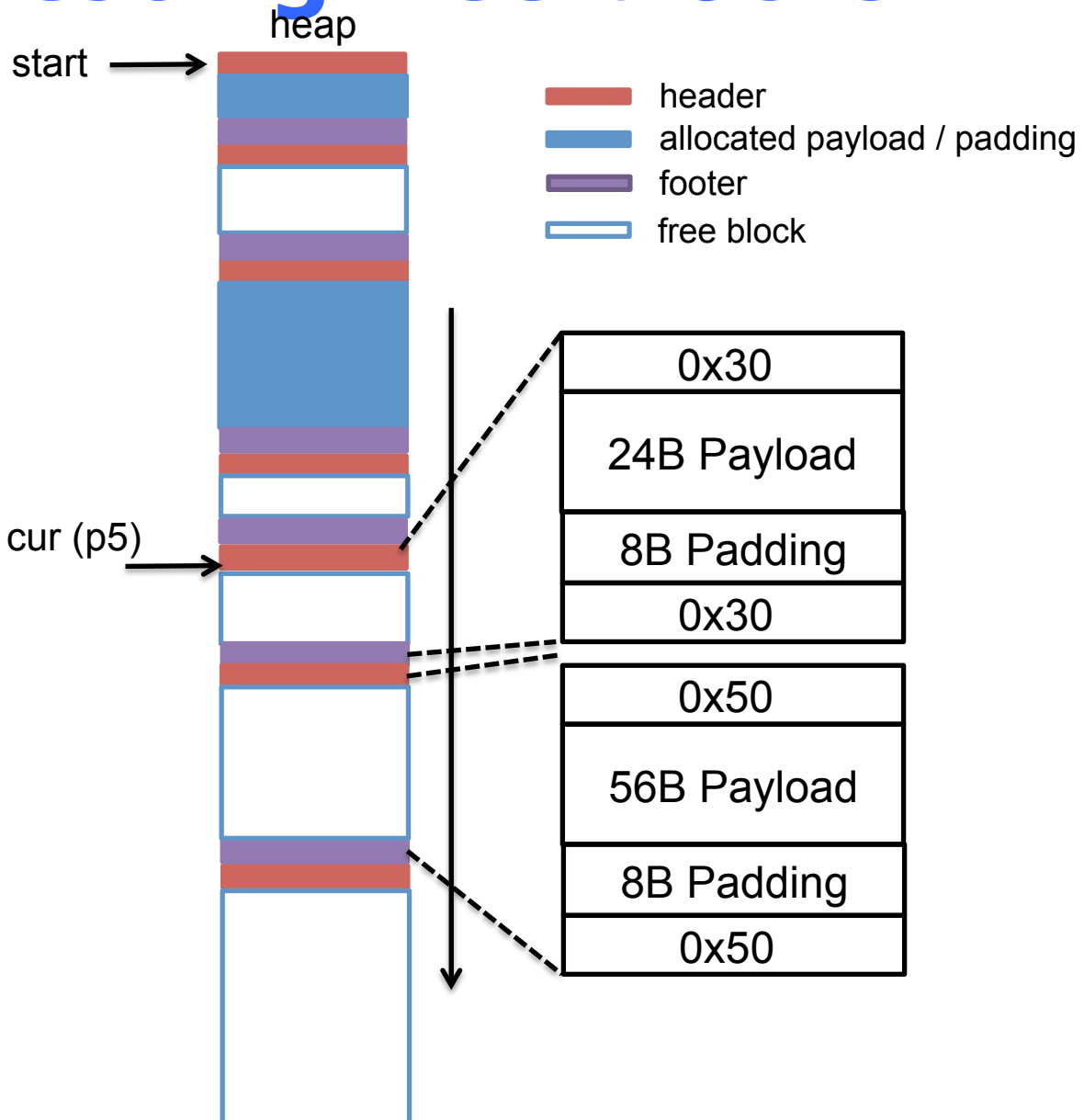



```
p  = malloc(1024)
```

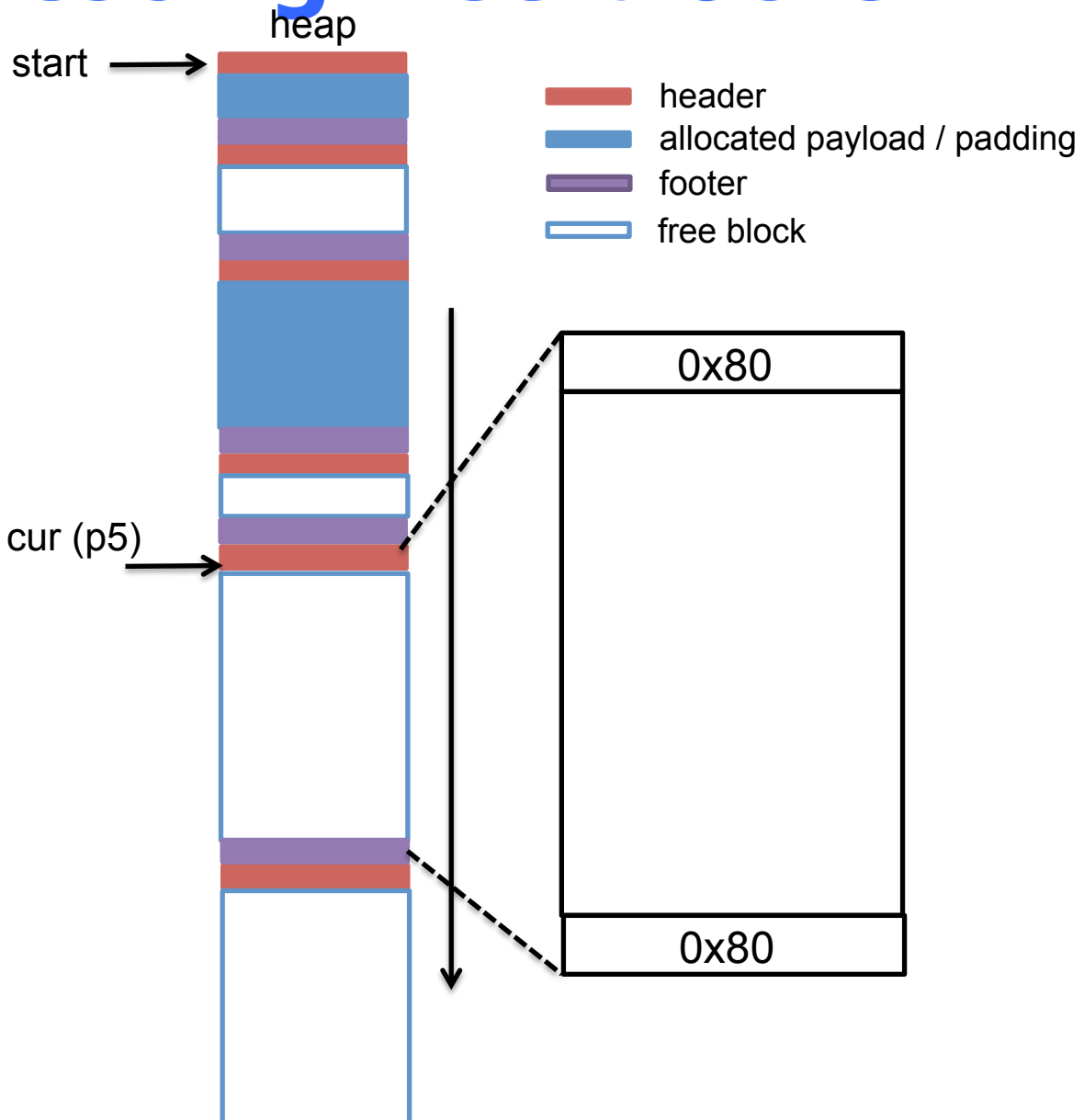
# Coalescing free blocks

```
p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
free(p5)
```



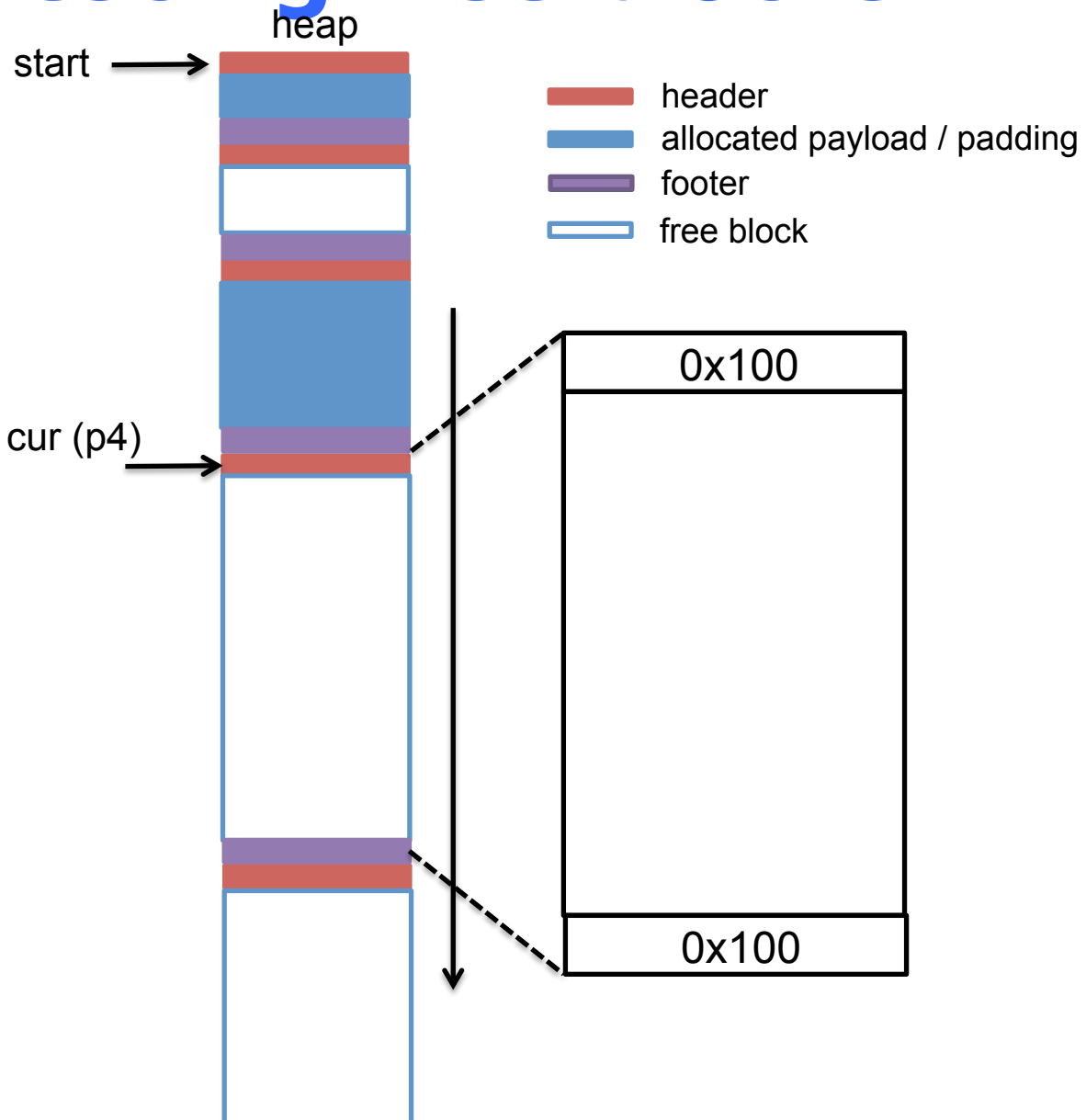
# Coalescing free blocks

```
p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
free(p5)
```



# Coalescing free blocks

```
p1 = malloc(8)
p2 = malloc(24)
p3 = malloc(56)
p4 = malloc(8)
p5 = malloc(24)
p6 = malloc(56)
free(p2)
free(p4)
free(p6)
free(p5)
```



# Exercises

| Alignment | Request | Block size | Header (hex) |
|---|---|---|---|
| 8 bytes | malloc(5) | | |
| 4 bytes | malloc(13) | | |
| 16 bytes | malloc(20) | | |
| 8 bytes | malloc(3) | | |

Each block has both header and footer

# Exercises

| Alignment | Request | Block size | Header (hex) |
|---|---|---|---|
| 8 bytes | malloc(5) | 24 | 0x19 |
| 4 bytes | malloc(13) | 32 | 0x19 |
| 16 bytes | malloc(20) | 32 | 0x21 |
| 8 bytes | malloc(3) | 24 | 0x19 |

Each block has both header and footer

# Explicit free lists

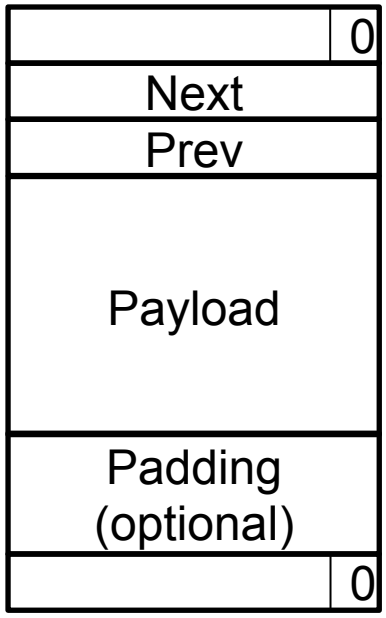
Problems of implicit free list

- Block allocation time is linear in the total number of heap blocks

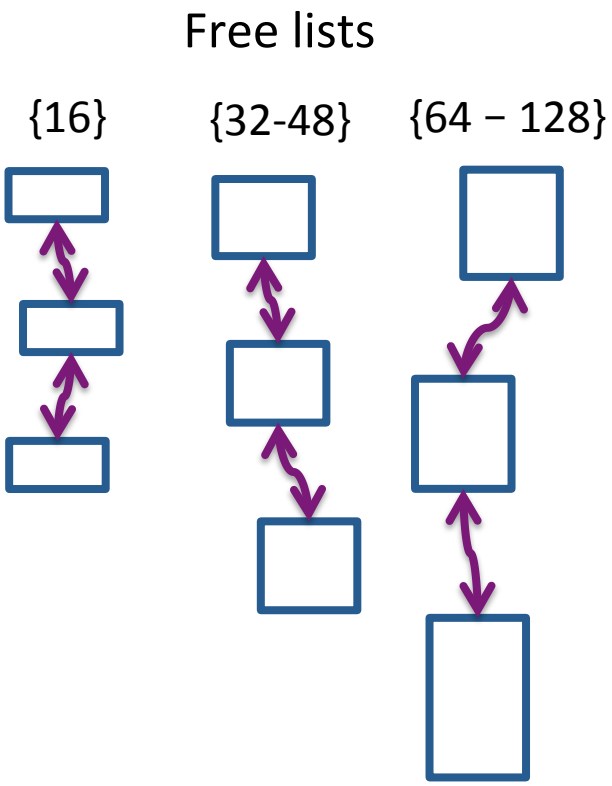
Basic idea – segregated list

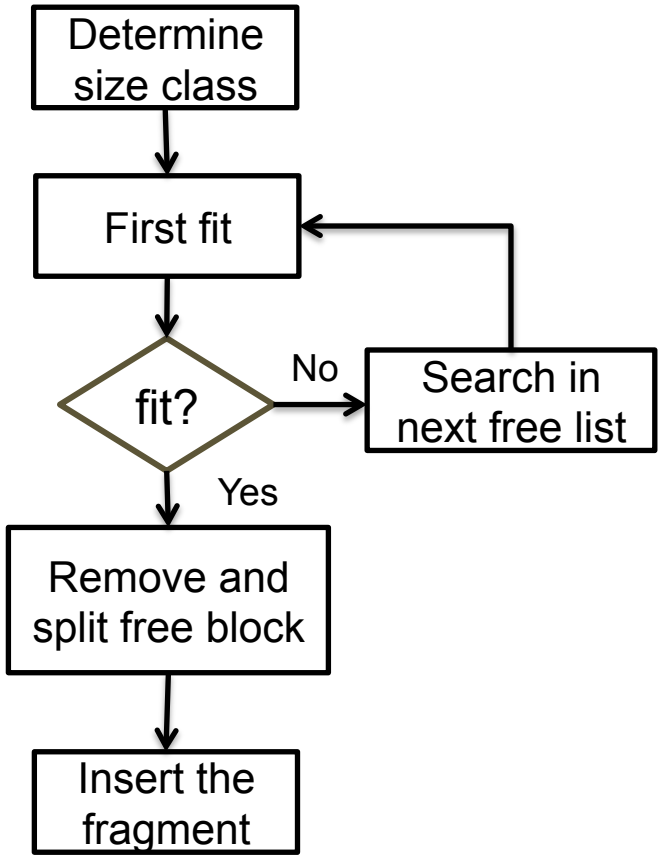
- Maintain multiple free list, each list holds blocks that are roughly the same size

# Segregated list

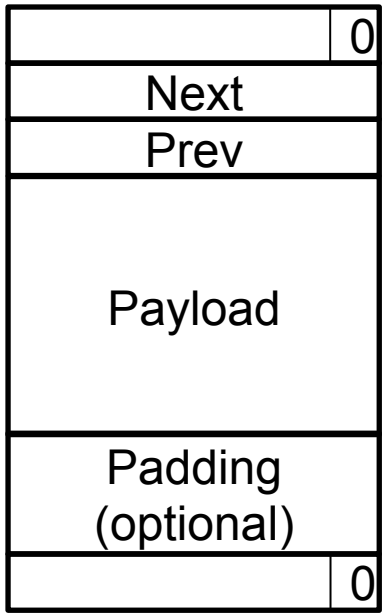



Block layout

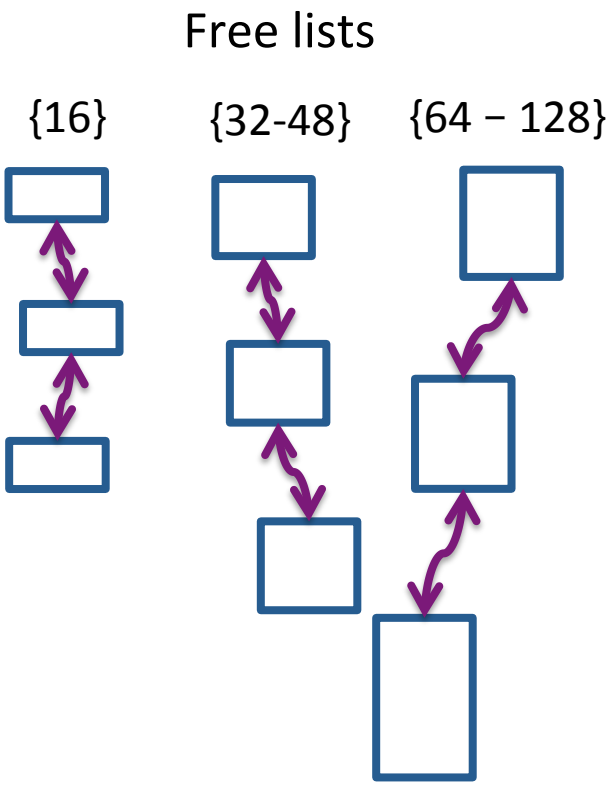


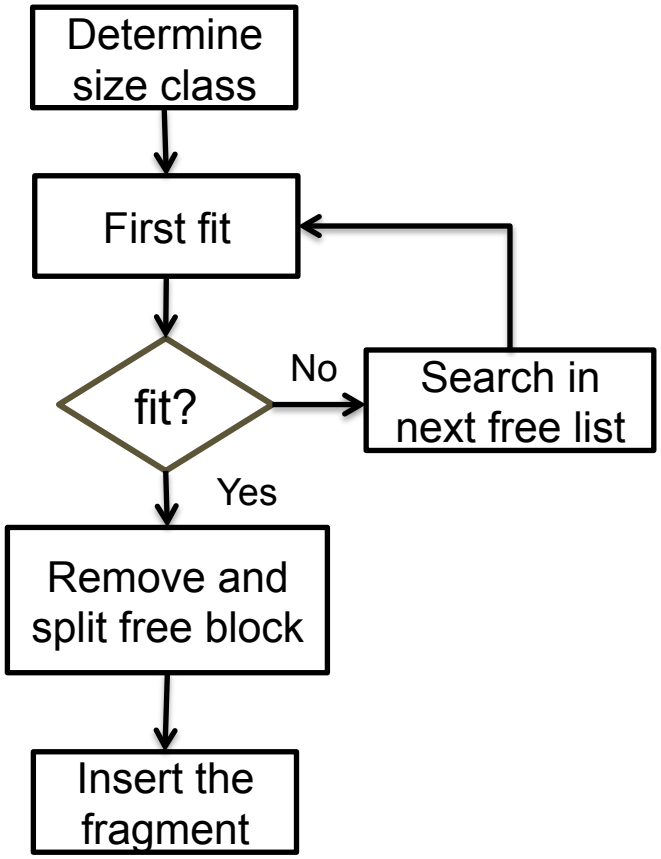



# Segregated list



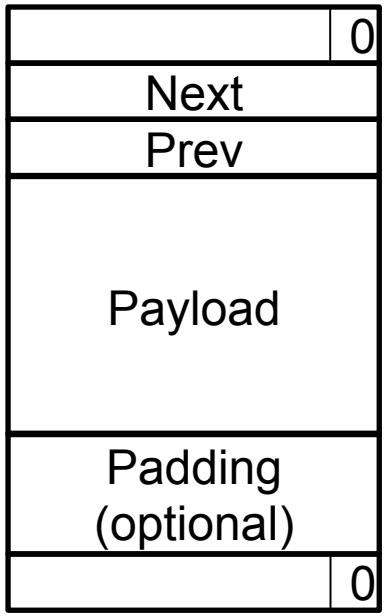

Block layout



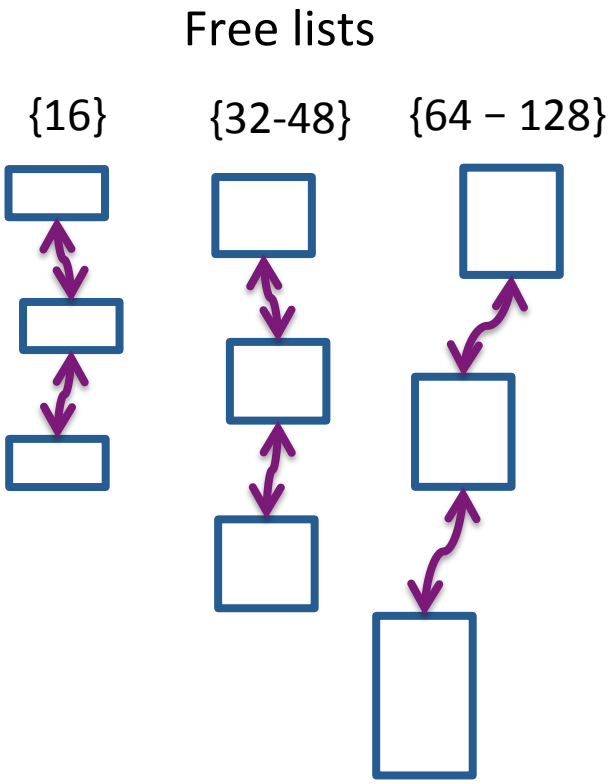



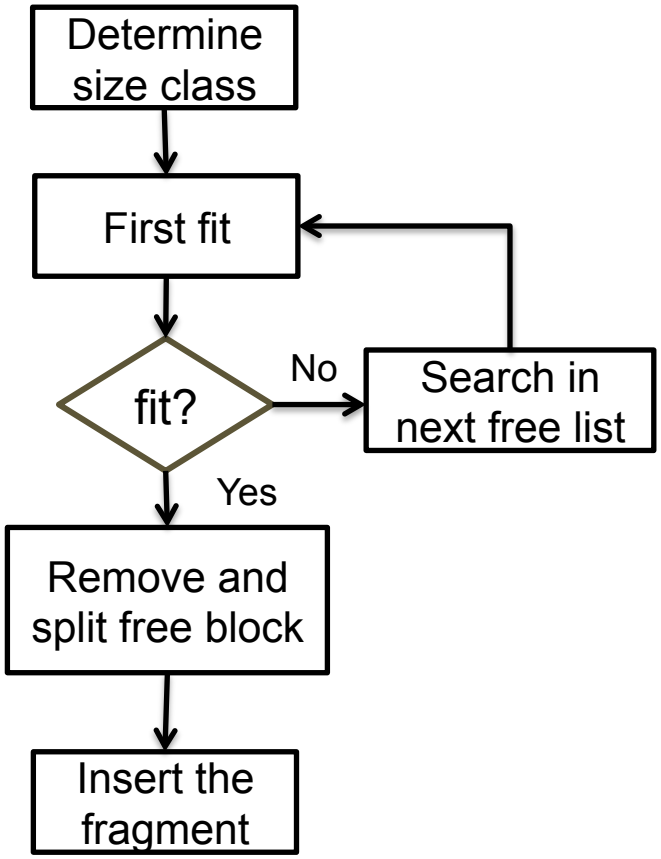

Do we need an allocated lists?

# Segregated list




Block layout

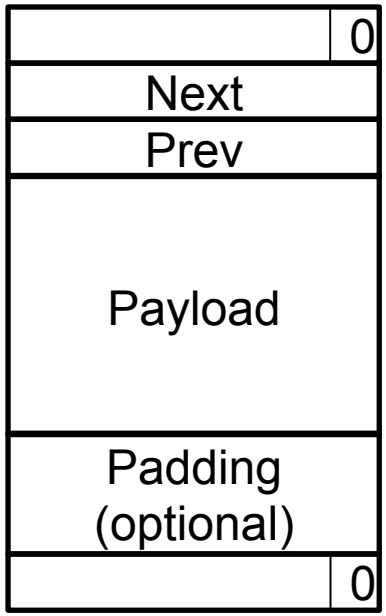


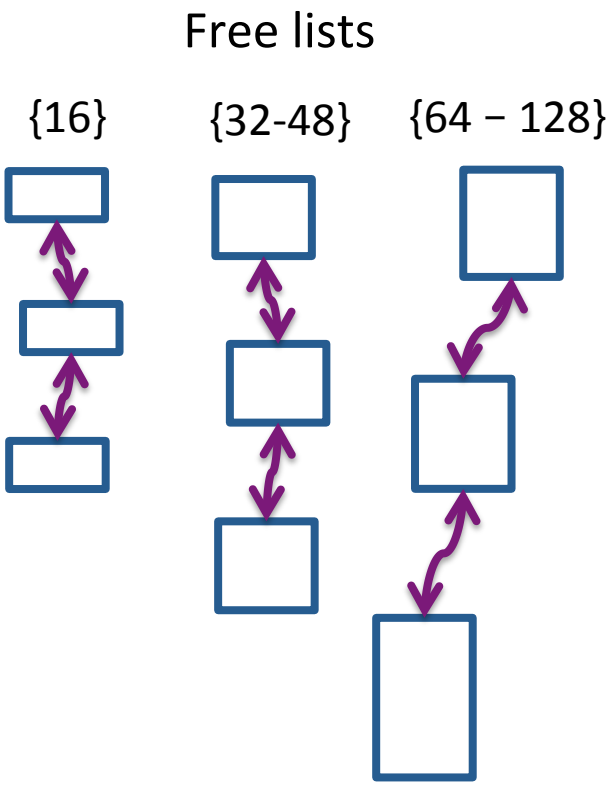

Do we need an allocated lists?

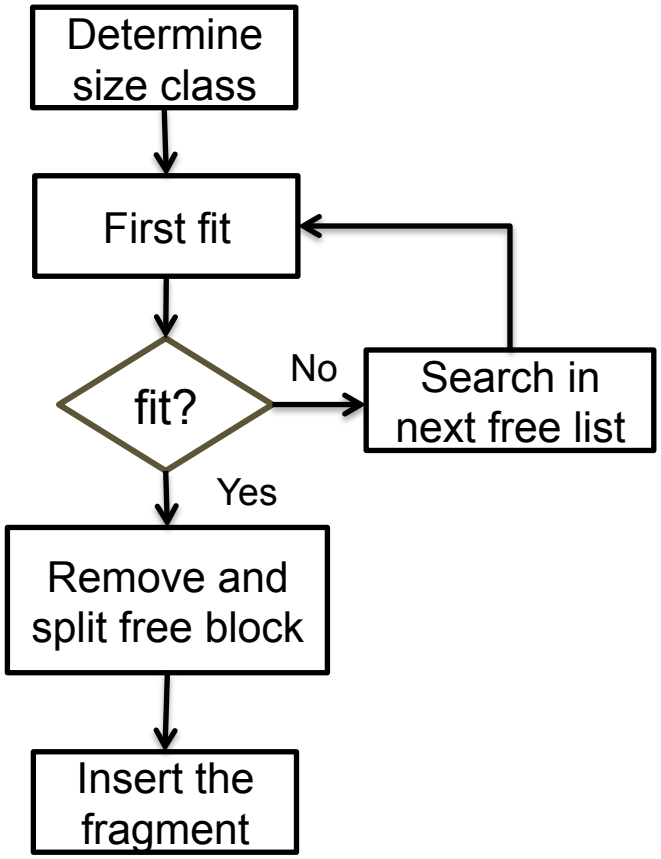
No, we are able to calculate the start of the block with p passed by free()

# Segregated list



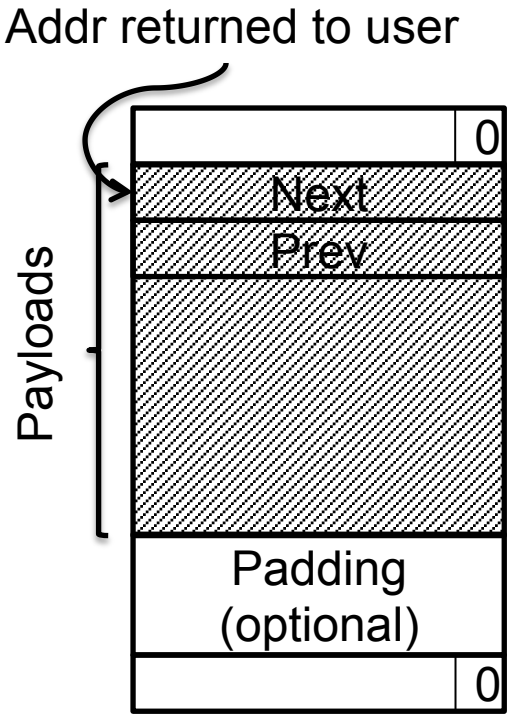

Block layout

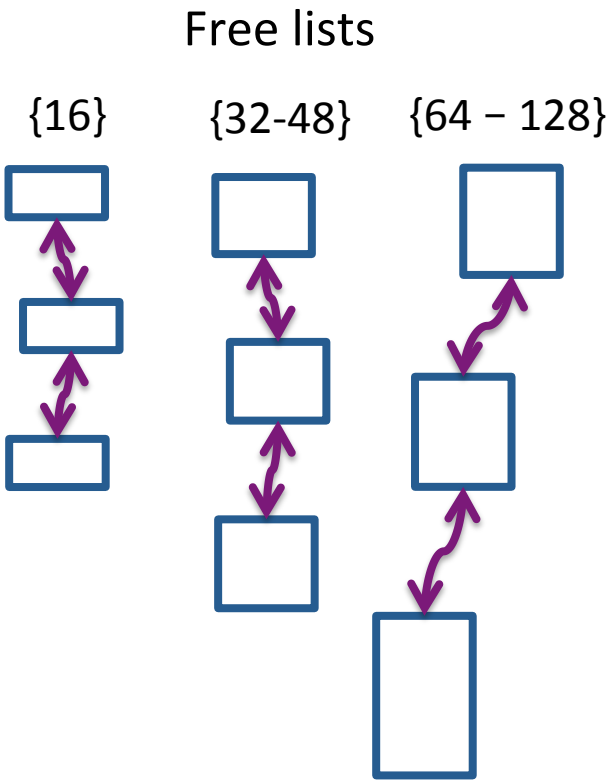


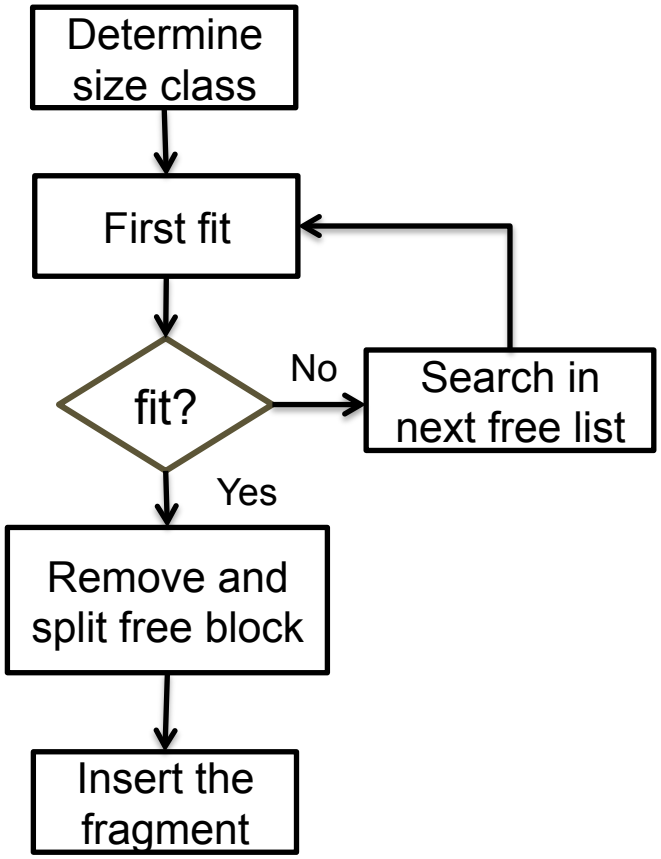




Can we get rid of Next and Prev?

# Segregated list



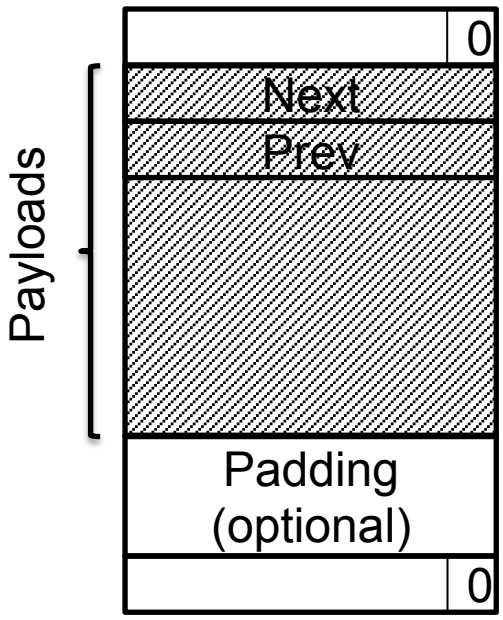



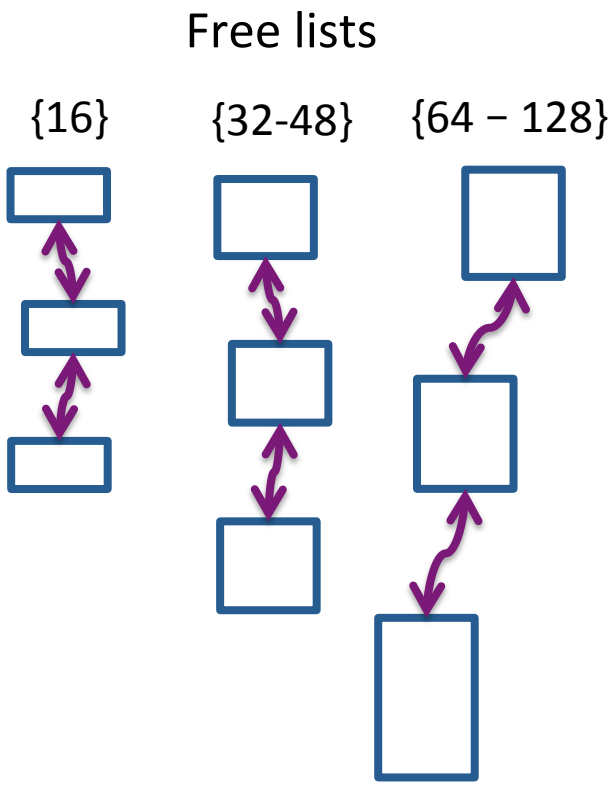

Can we get rid of Next and Prev?

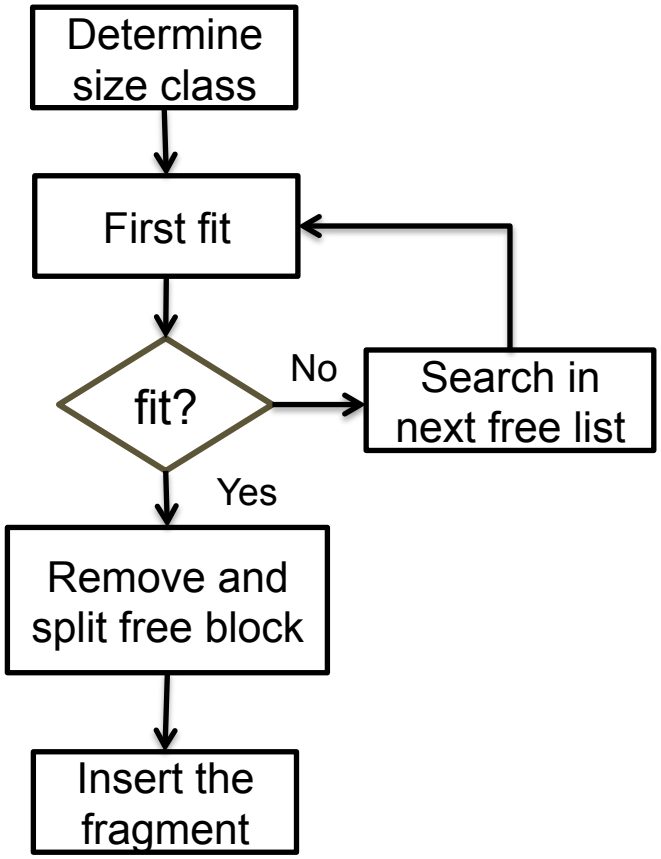
Allocated blocks do not need Next and Prev

# Segregated list




Block layout




With this design, what is the minimal size for a free block?

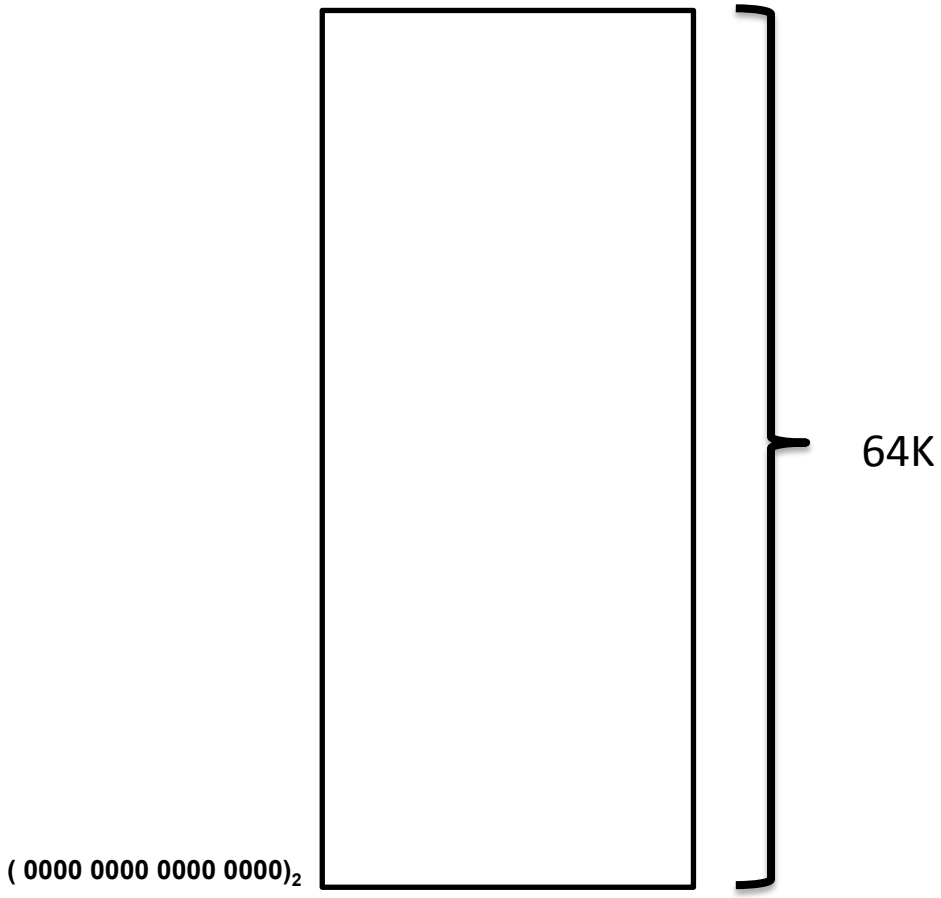
# Buddy System

Adopted by Linux kernel and jemalloc

This lecture

- A simplified binary buddy allocator

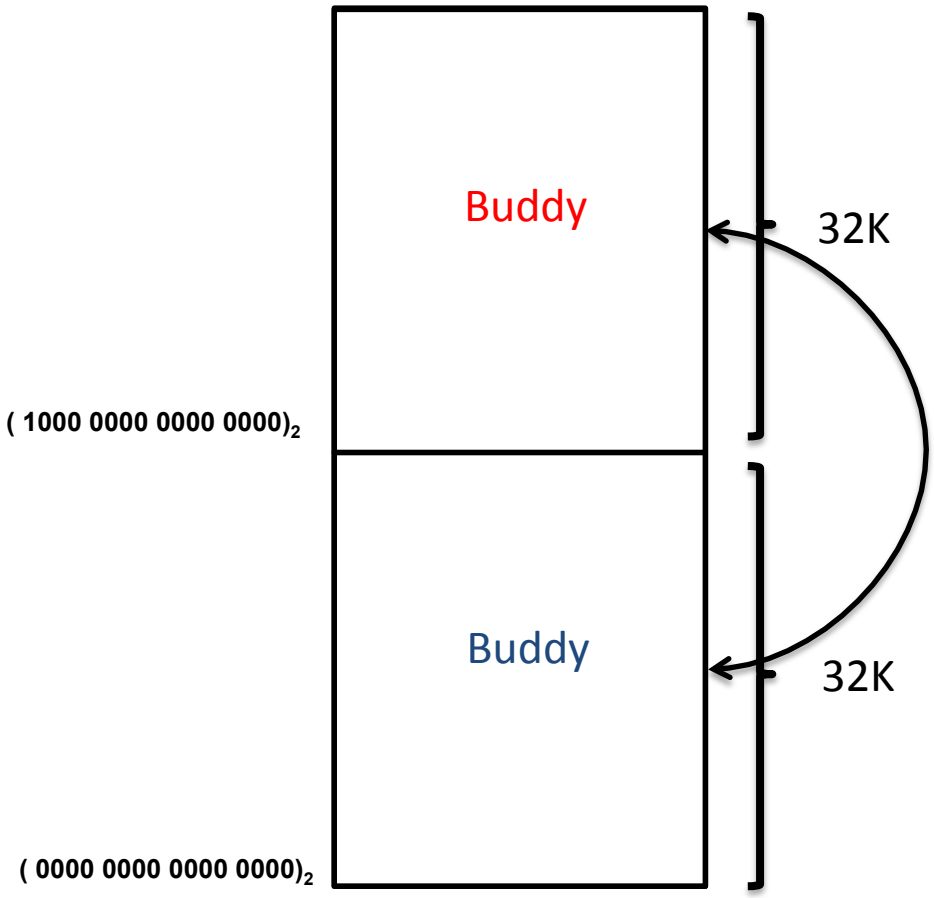
# Binary buddy system



# Binary buddy system

Split

- Split exactly in half

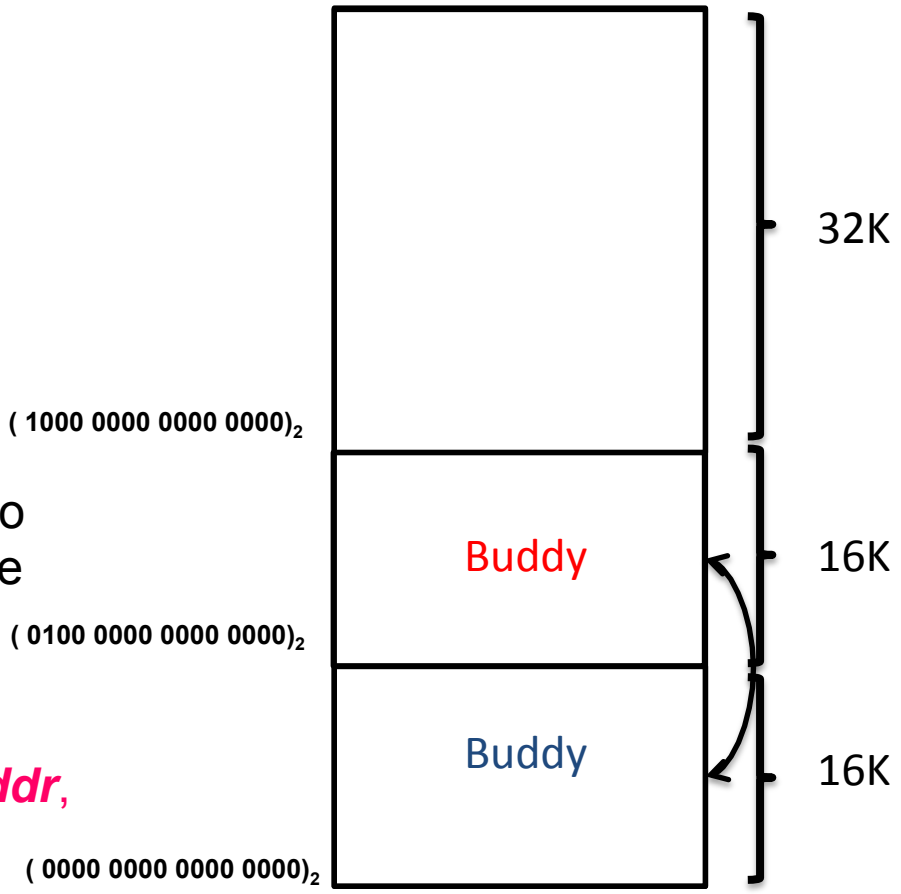


# Binary buddy system

Split
- Split exactly in half
- Each half is the buddy of the other

Address
- Block of size $2^n$ begin at memory addresses where the ***n*** least significant bits are zero
- When a block of size $2^{n+1}$ is split into two blocks of size $2^n$, the addresses of these two blocks will differ in exactly one bit, bit **n**.

If a block of size $2^n$ begins at address ***addr***, what is its buddy address and size?

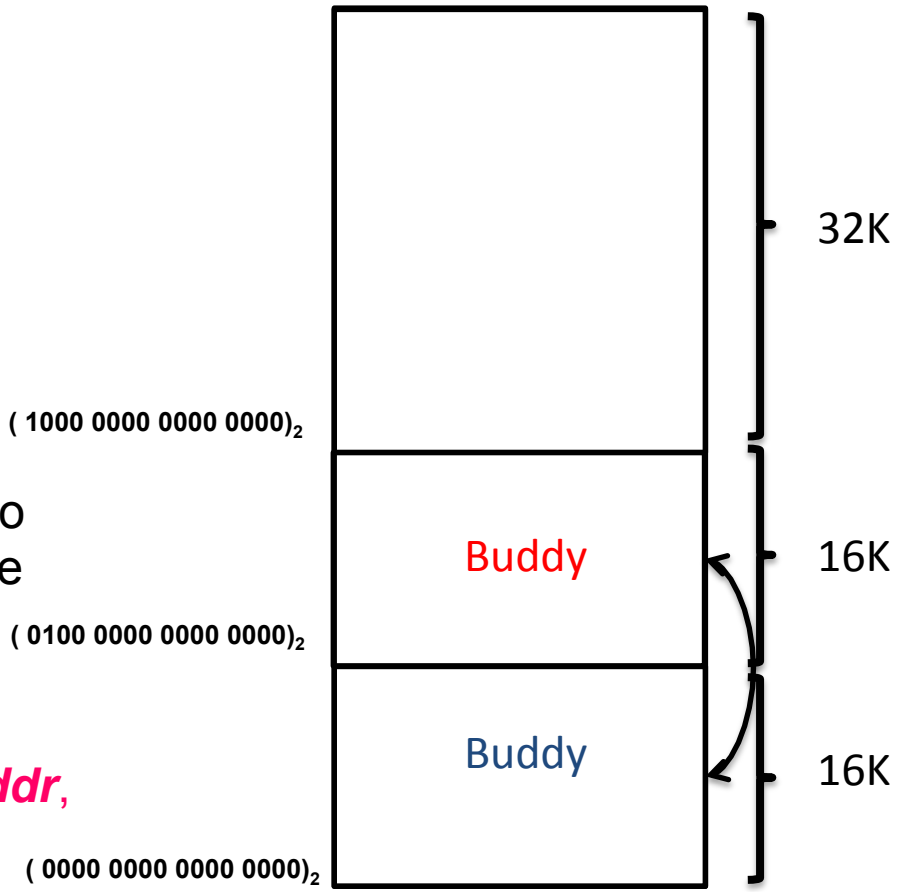


# Binary buddy system

Split
- Split exactly in half
- Each half is the buddy of the other

Address
- Block of size $2^n$ begin at memory addresses where the ***n*** least significant bits are zero
- When a block of size $2^{n+1}$ is split into two blocks of size $2^n$, the addresses of these two blocks will differ in exactly one bit, bit **n**.

If a block of size $2^n$ begins at address ***addr***, what is its buddy address and size?
addr of buddy = addr ^ (n<<1)

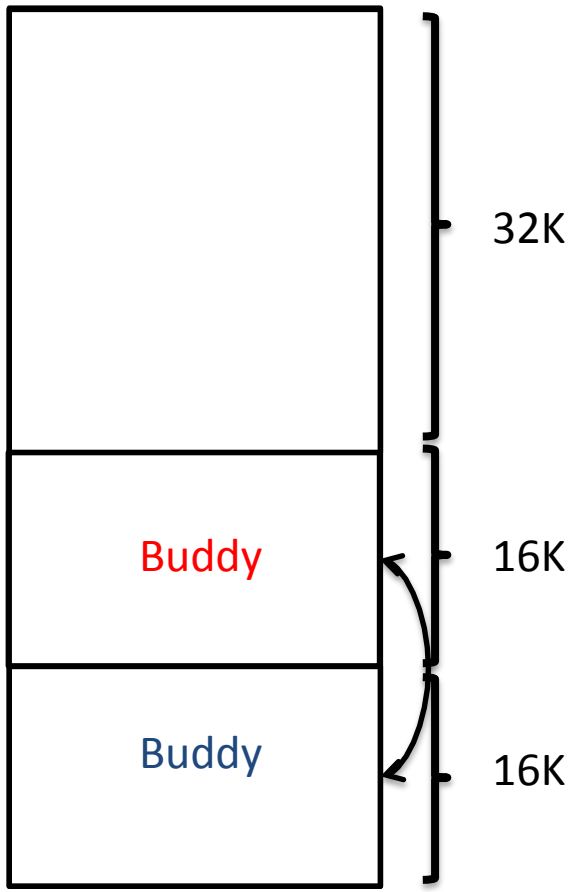


# Binary buddy system

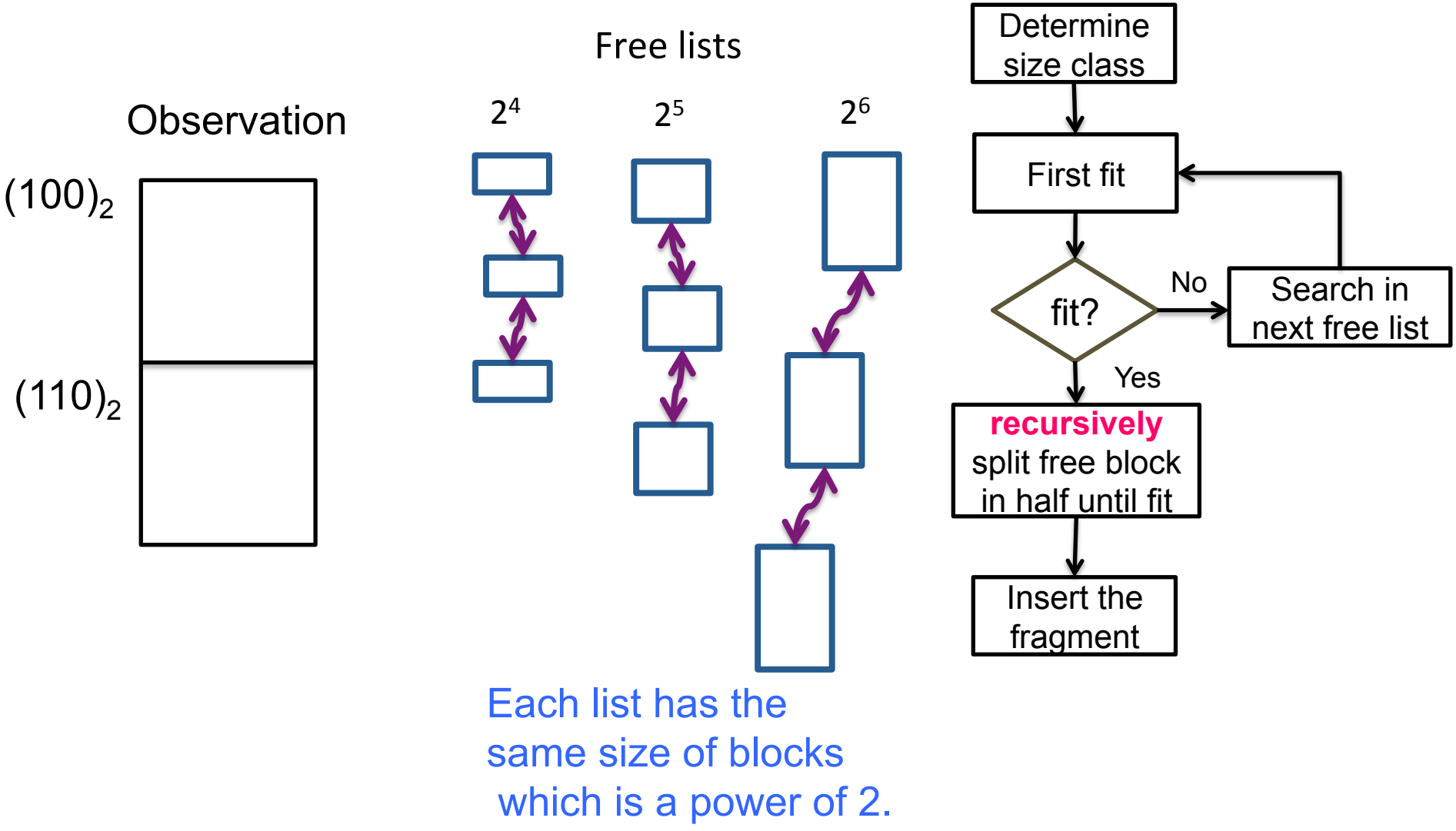
Split
- Split exactly in half
- Each half is the buddy of the other

Address
- Block of size $2^n$ begin at memory addresses where the ***n*** least significant bits are zero
- When a block of size $2^{n+1}$ is split into two blocks of size $2^n$, the addresses of these two blocks will differ in exactly one bit, bit **n**.

Combine
- We assume only combine with its buddy block in our lecture



# Buddy system



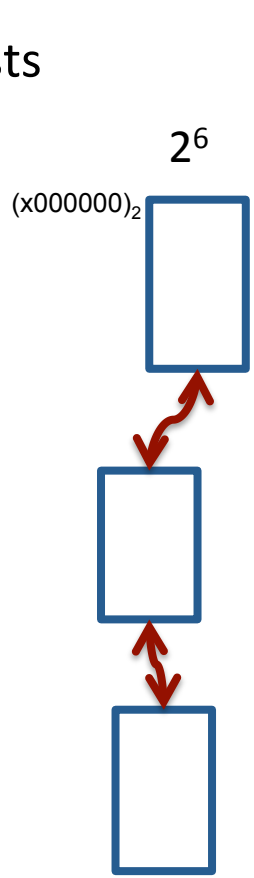

Each list has the same size of blocks which is a power of 2.

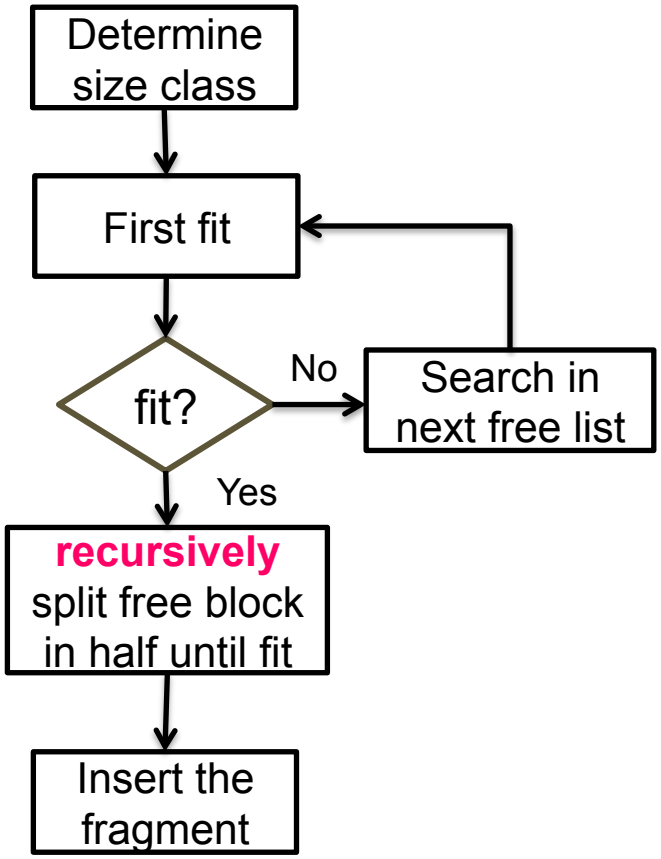
# Buddy system

`p = malloc(1)`

Step 1. search in $2^6$ list

Step 2. recursive split



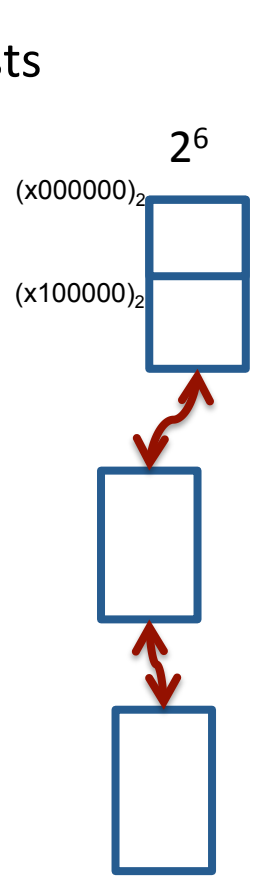

Each list has the same size of blocks which is a power of 2.

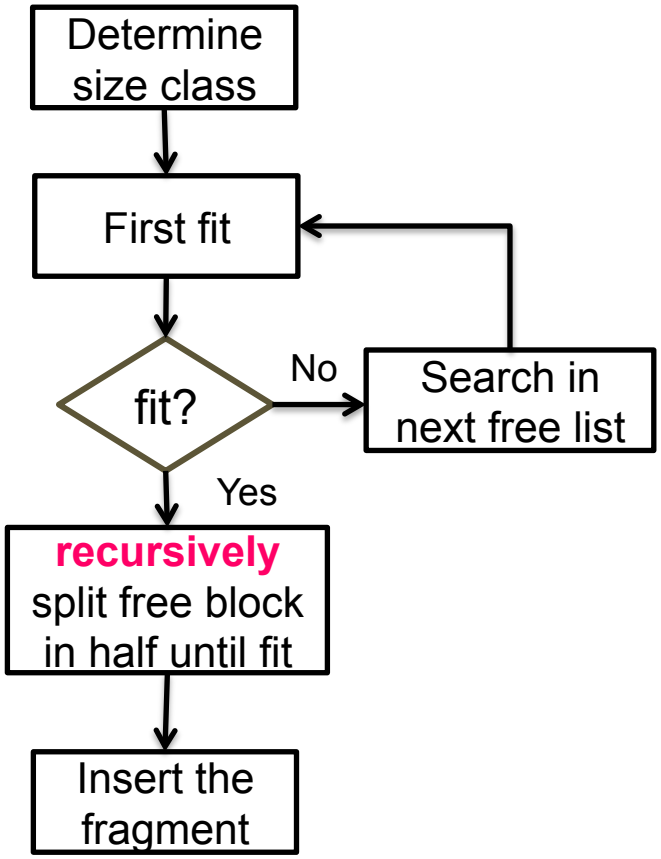
# Buddy system

```
p = malloc(1)
```

Step 1. search in $2^6$ list

Step 2. recursive split



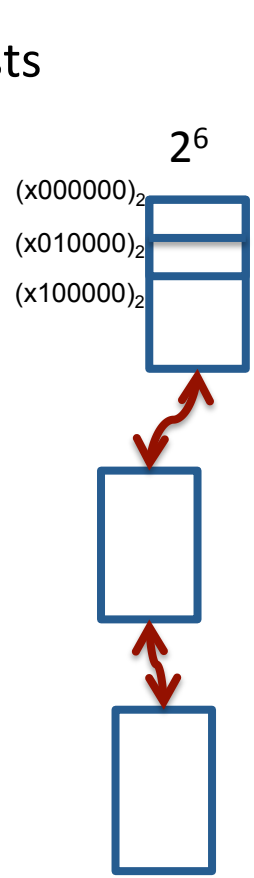



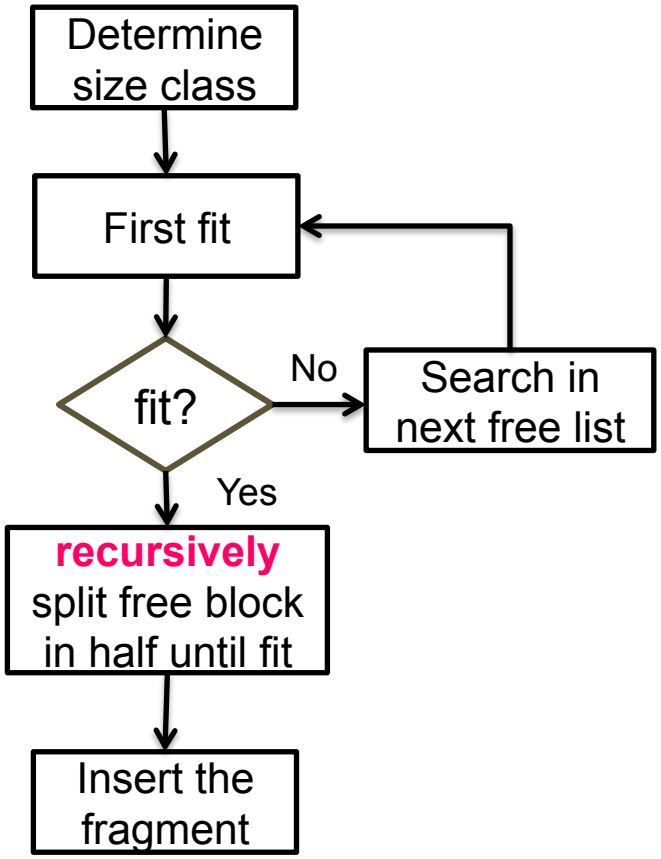
# Buddy system

```
p = malloc(1)
```

Step 1. search in $2^6$ list

Step 2. recursive split



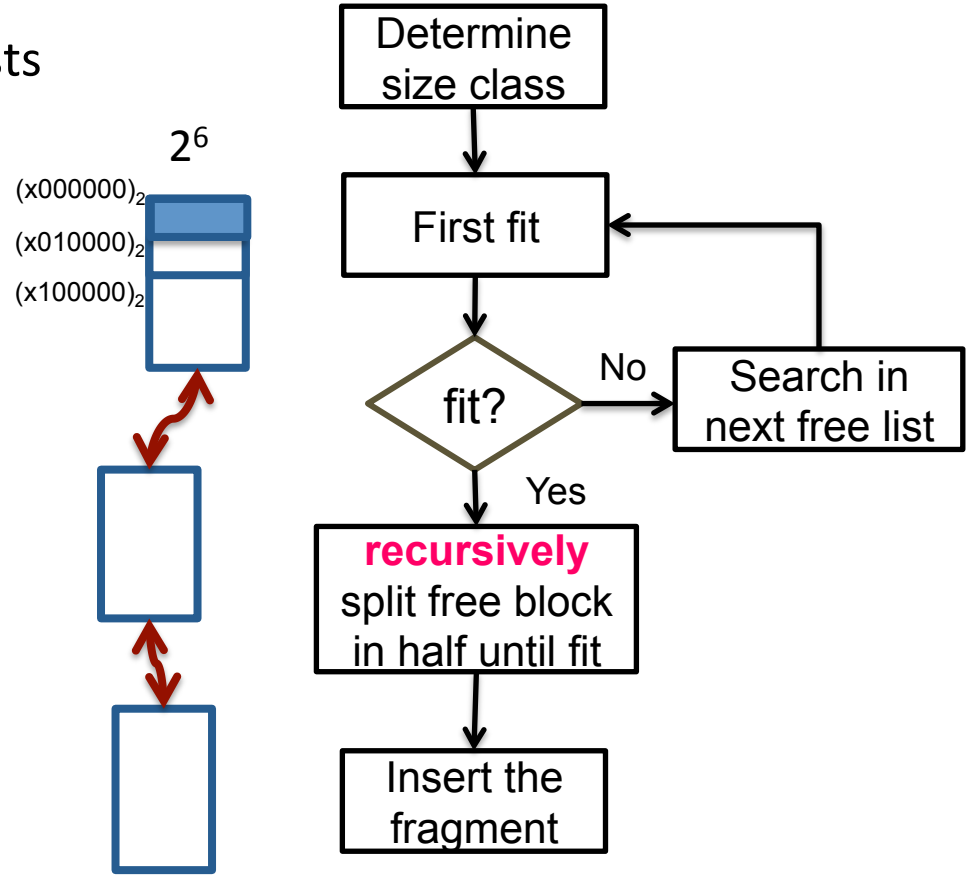



# Buddy system

`p = malloc(1)`

Step 1. search in $2^6$ list

Step 2. recursive split



# Buddy system

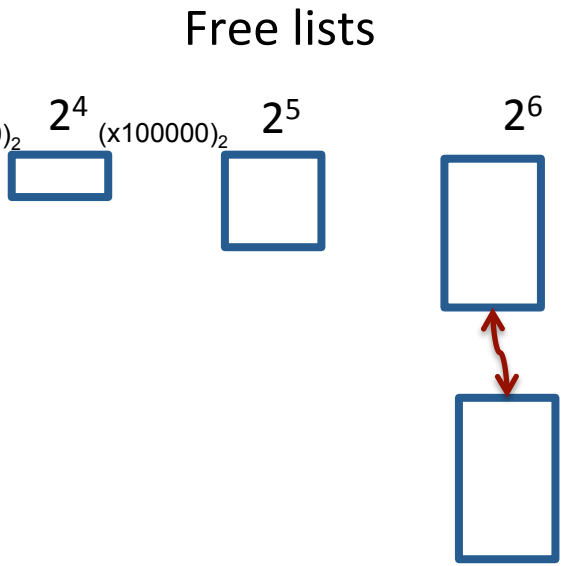
`p = malloc(1)`

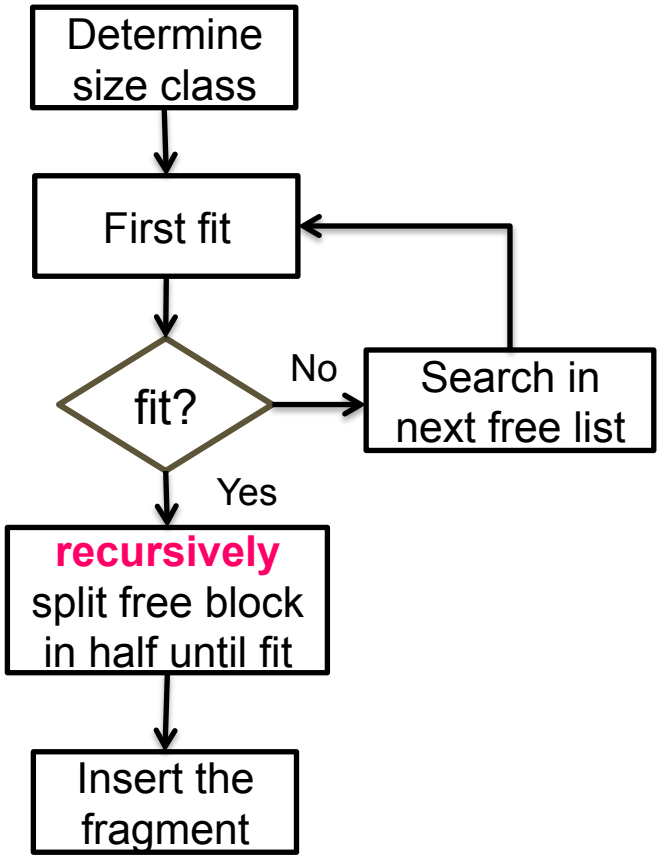
Step 1. search in $2^6$ list

Step 2. recursive split

Step 3. insert free blocks into the list

Step 4. return, p is $(x000000)_2$

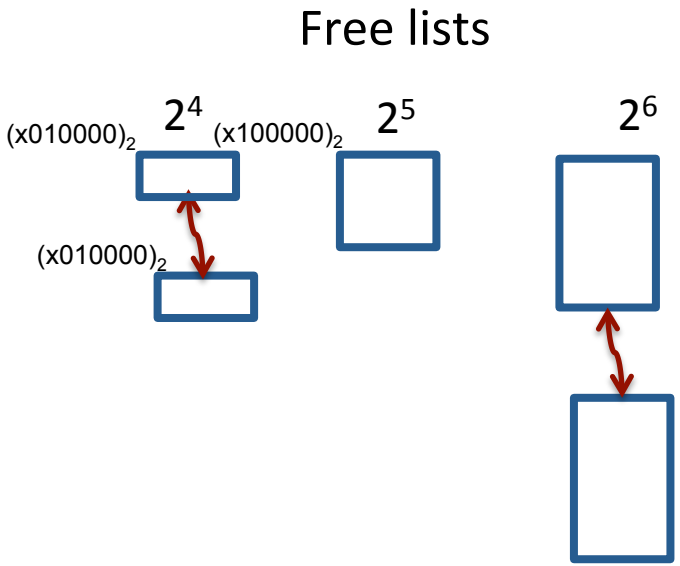


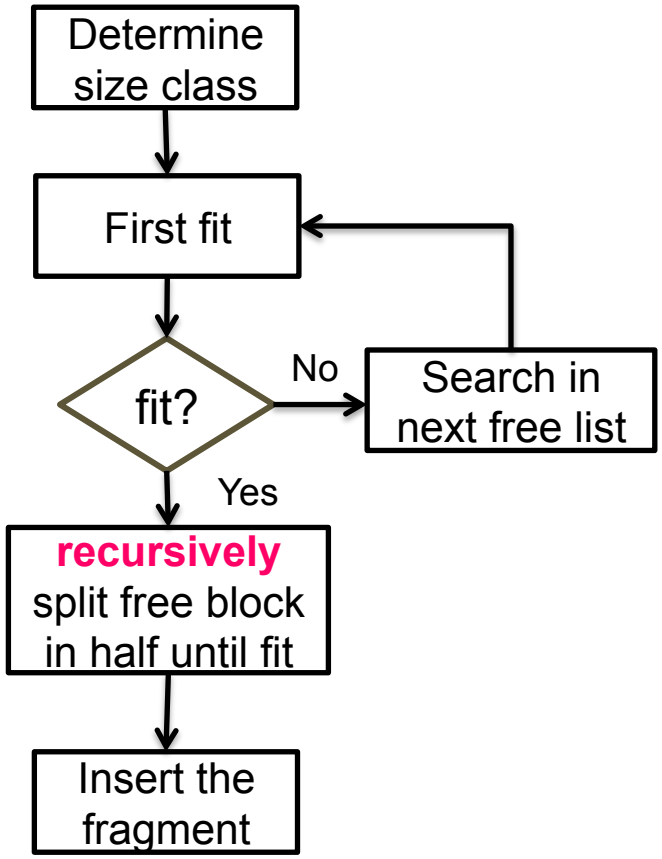



# Buddy system

```
p = malloc(1)

free(p)
```

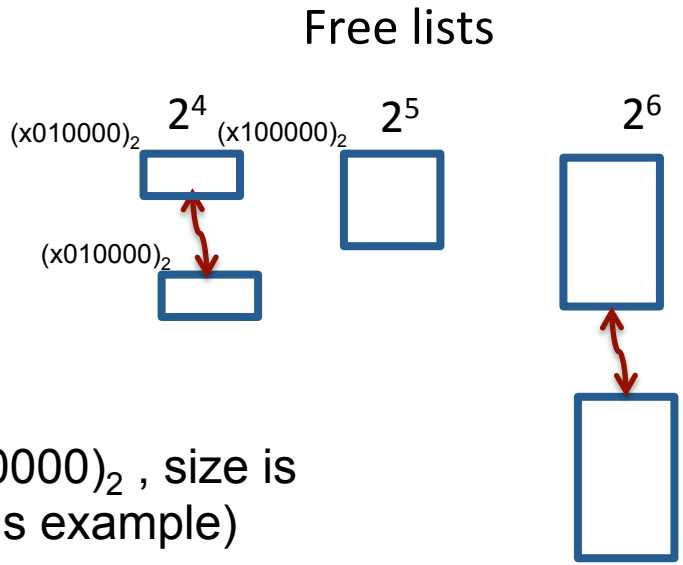


# Buddy system

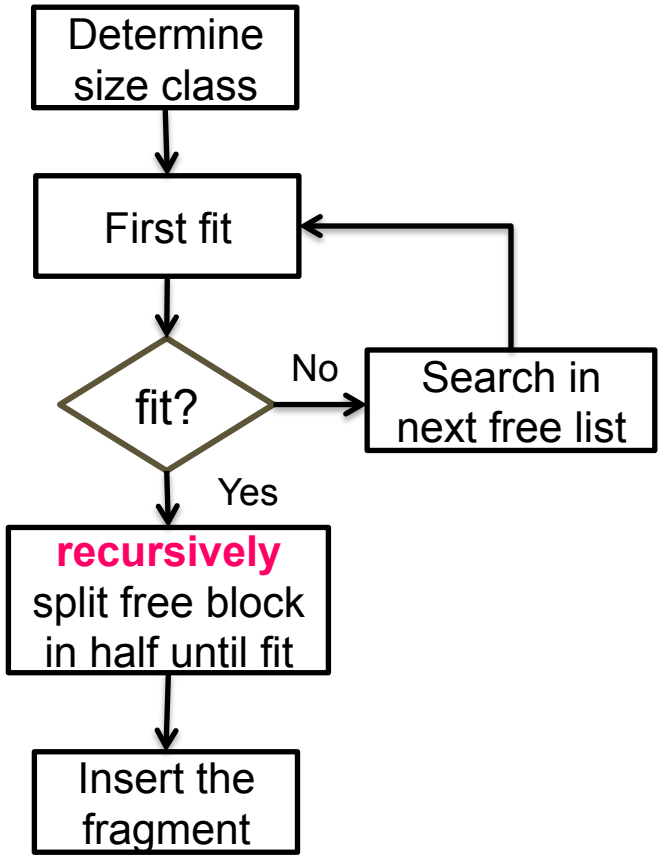
`p = malloc(1)`

free(p)




p's address is $(x000000)_2$ , size is 16B (no footer in this example)

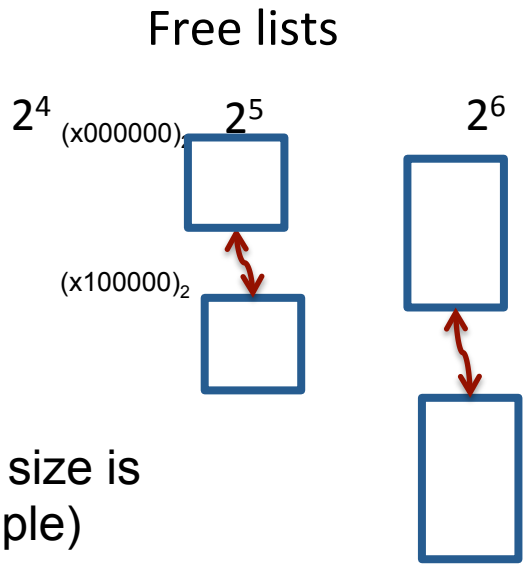
→ p's buddy is 16 B block begins at x000000 ^ (1<<4) which is $(x010000)_2$



# Buddy system

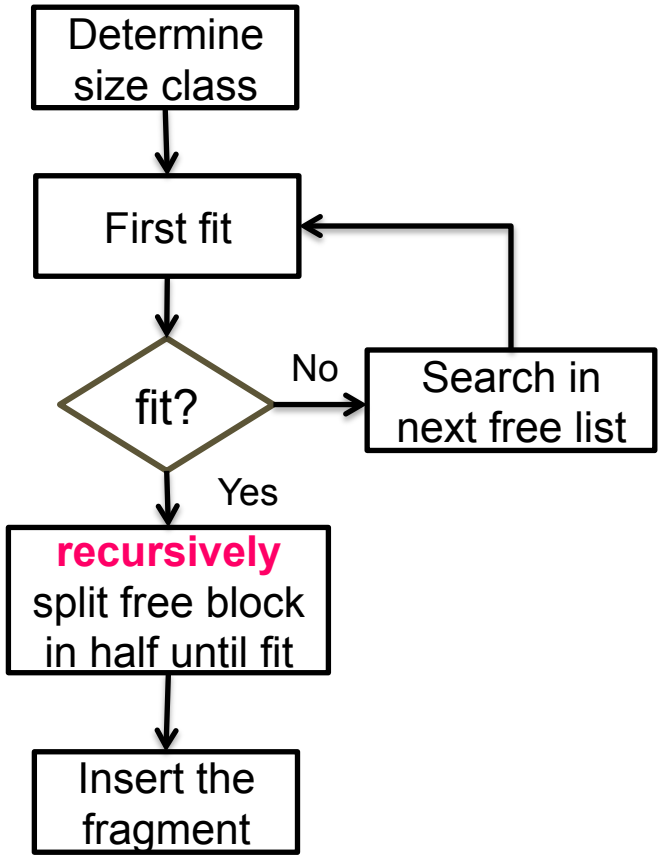
```
p = malloc(1)
```

free(p)




p's address is $(x000000)_2$ , size is 32B (no footer in this example)

→ p's buddy is 32 B block begins at x000000 ^ (1<<5) which is $(x100000)_2$



# Buddy system

`p = malloc(1)`

free(p)

p's address is $(x000000)_2$ , size is 32B

→ p's next block address is p + 32B which is $(x100000)_2$