

Dynamic Memory Allocation

Jinyang Li

Some are based on Tiger Wang's slides

Why dynamic memory allocation?

```
typedef struct node {
    int val;
    struct node *next;
} node;

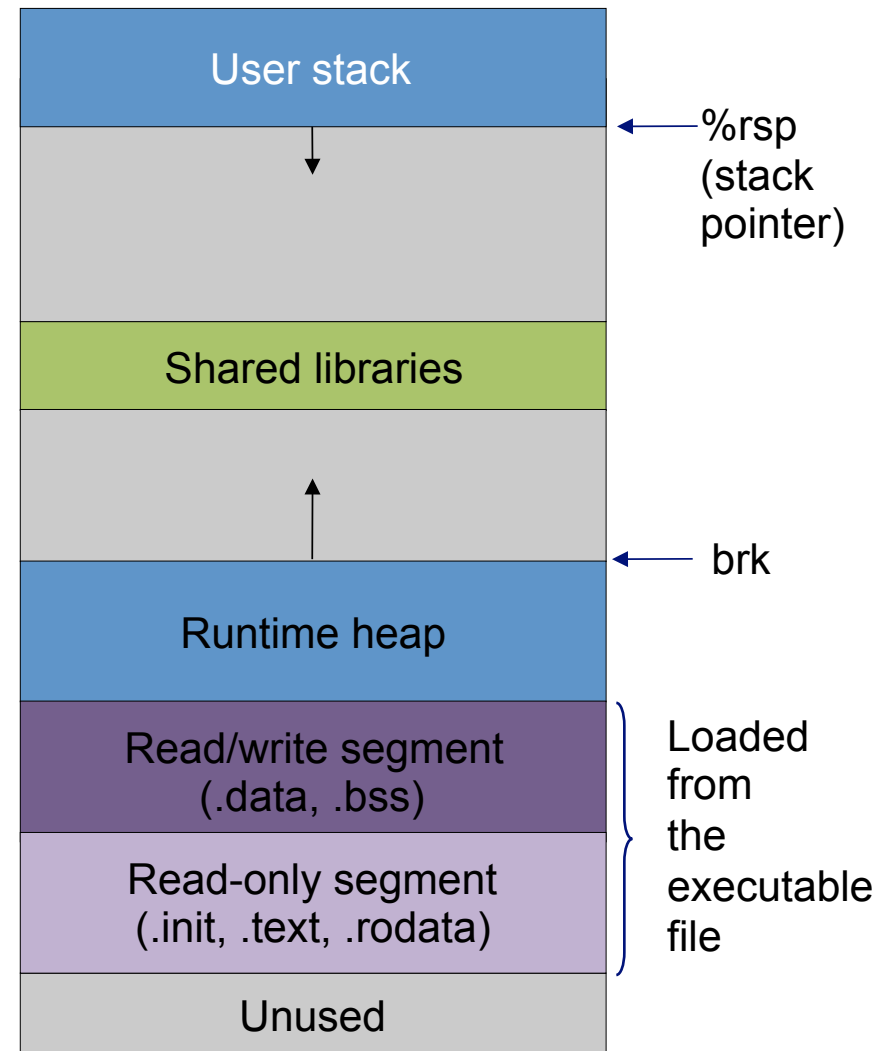
void
list_insert(node **head, int v) {
    node *np = malloc(sizeof(node));
    np->next = *head;
    np->val = v;
    *head = np;
}

int
main(void) {
    node *head = NULL;
    int n = atoi(argv[1]);
    for (int i = 0; i < n; i++) {
        list_insert(&head, i);
    }
}
```

How many nodes to allocate is only known at runtime (when the program executes)

Dynamic allocation on heap

Question: can one dynamically allocate memory on stack?

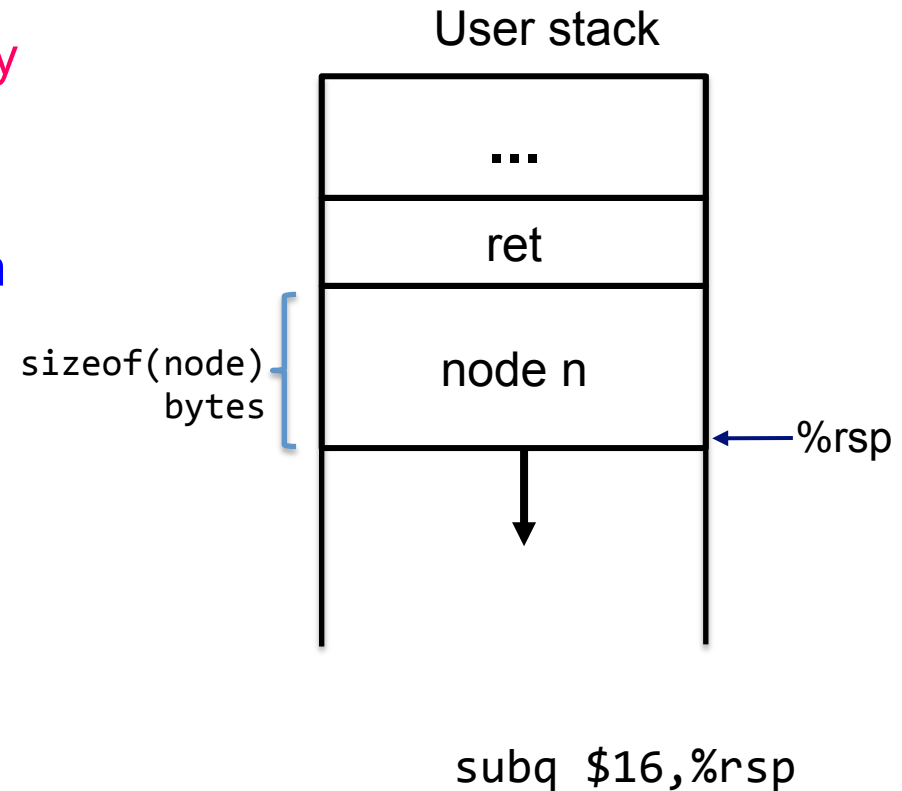


Dynamic allocation on heap

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

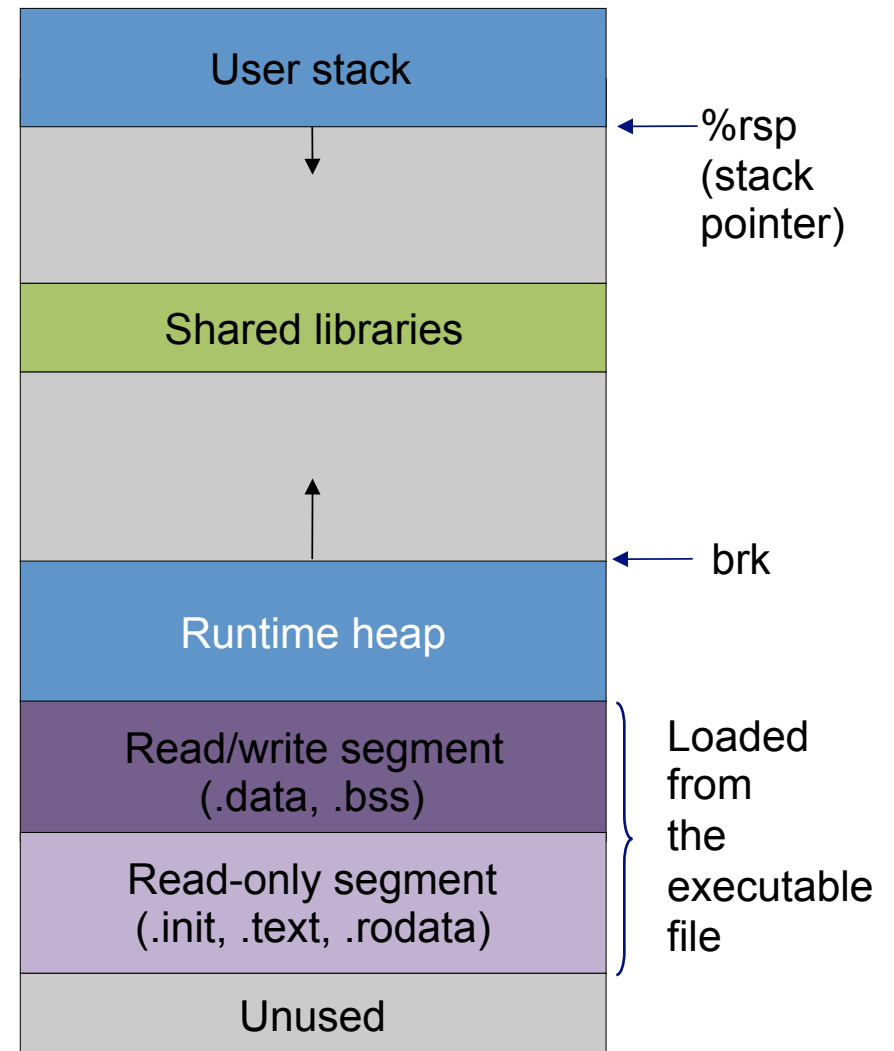
Answer: Yes, but space is freed upon function return

```
void  
list_insert(node *head, int v) {  
    node n;  
    node *np = &n;  
    np->next = head;  
    np->val = v;  
    *head = np;  
}
```



Dynamic allocation on heap

Question: How to allocate memory on heap?



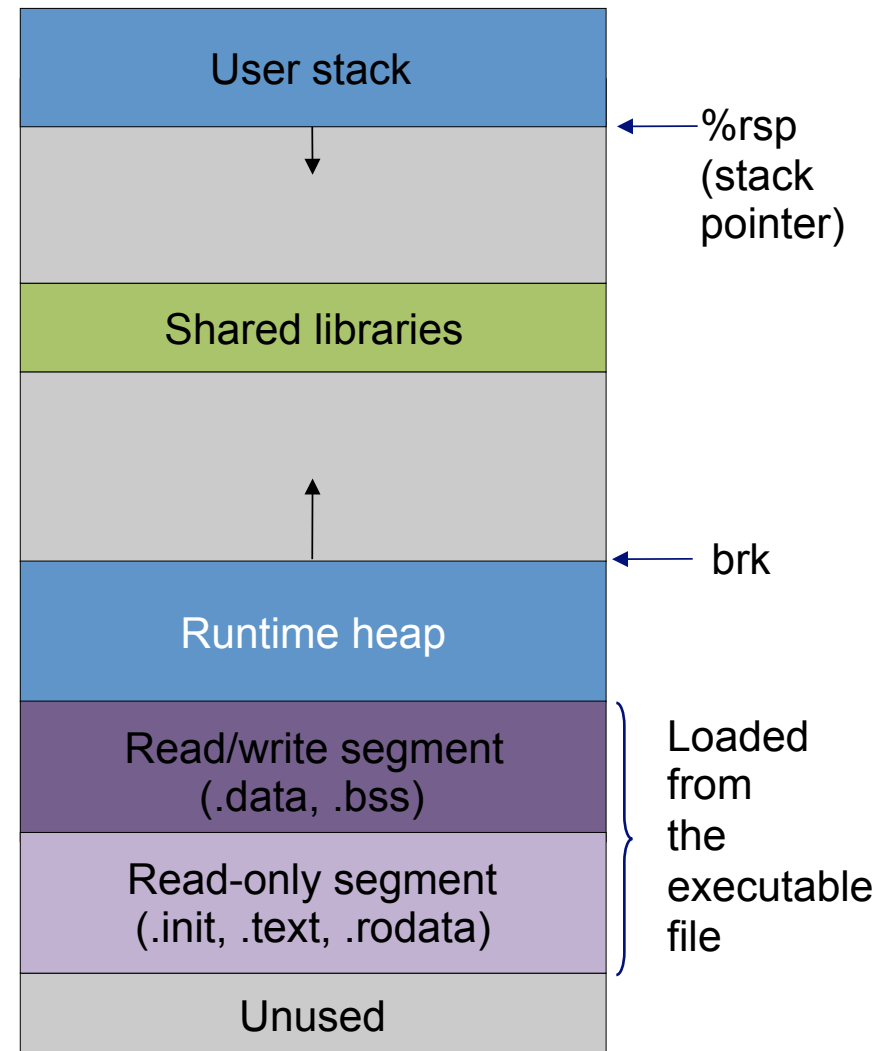
Dynamic allocation on heap

Question: How to allocate memory on heap?

Ask OS for allocation on the heap via **system calls**

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

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

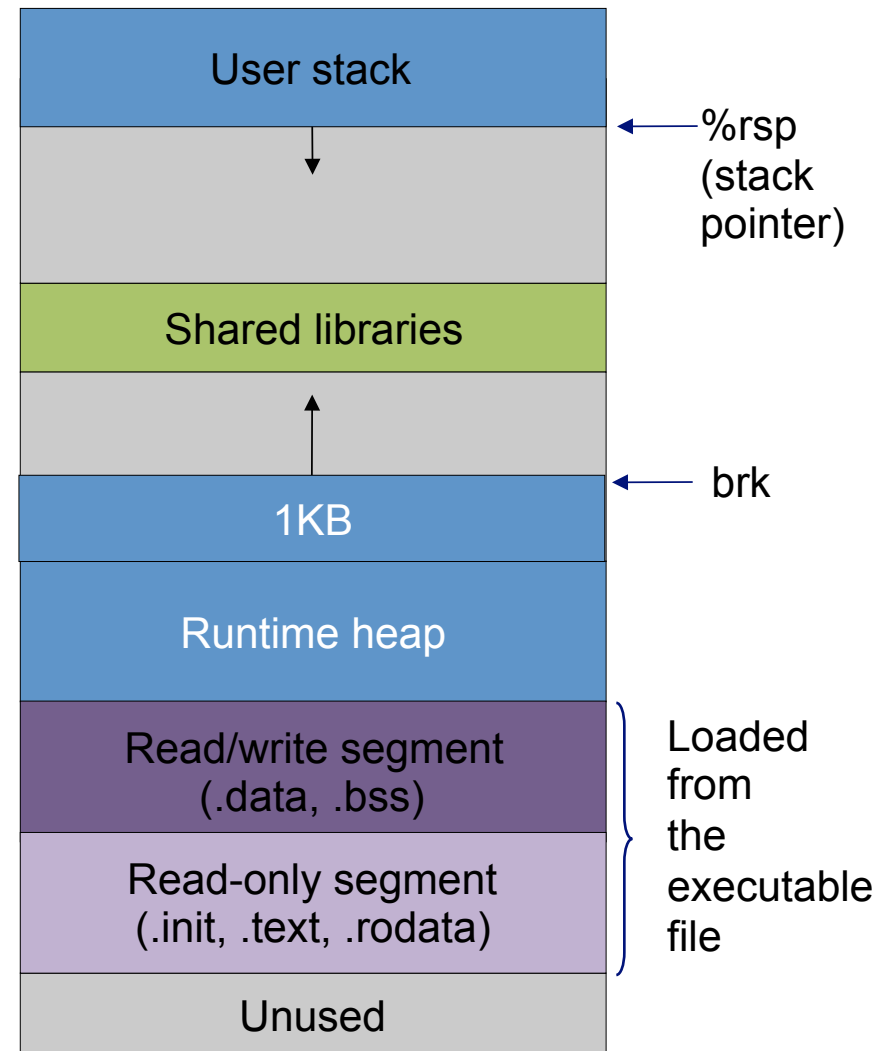
Question: How to allocate memory on heap?

Ask OS for allocation on the heap via **system calls**

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

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

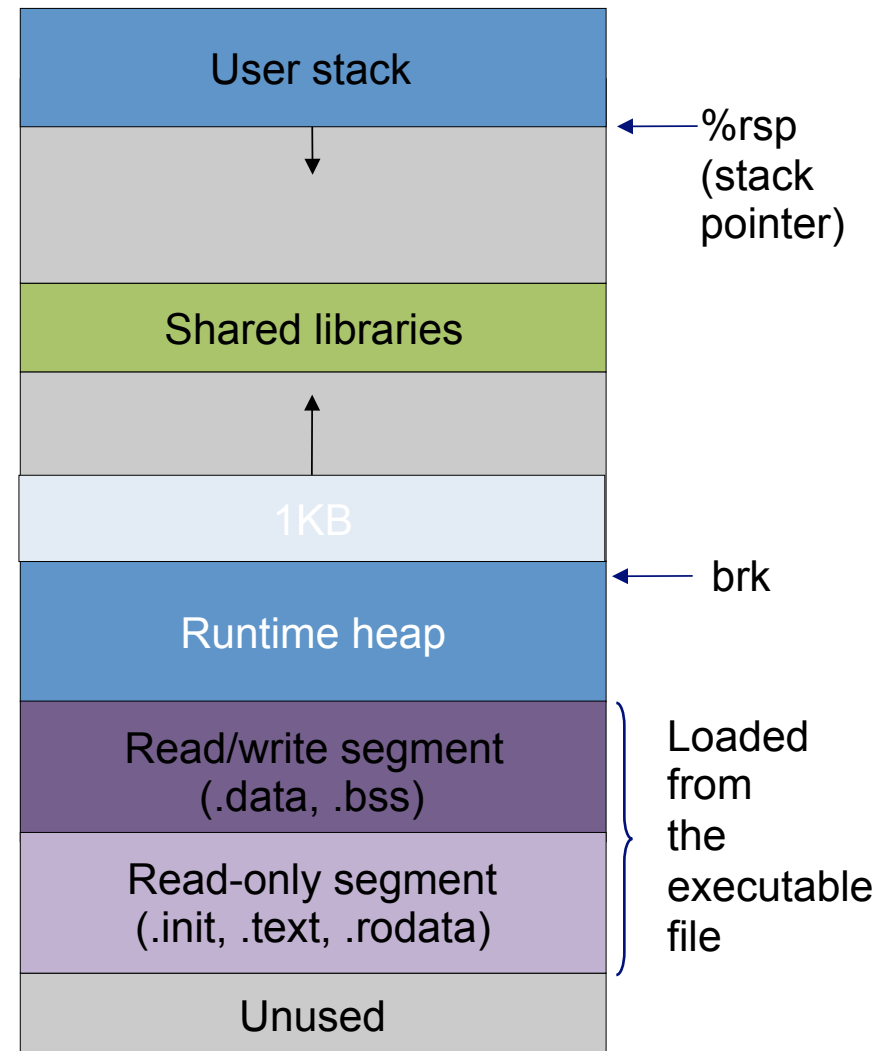
Question: How to allocate memory on heap?

Ask OS for allocation on the heap via **system calls**

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

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?

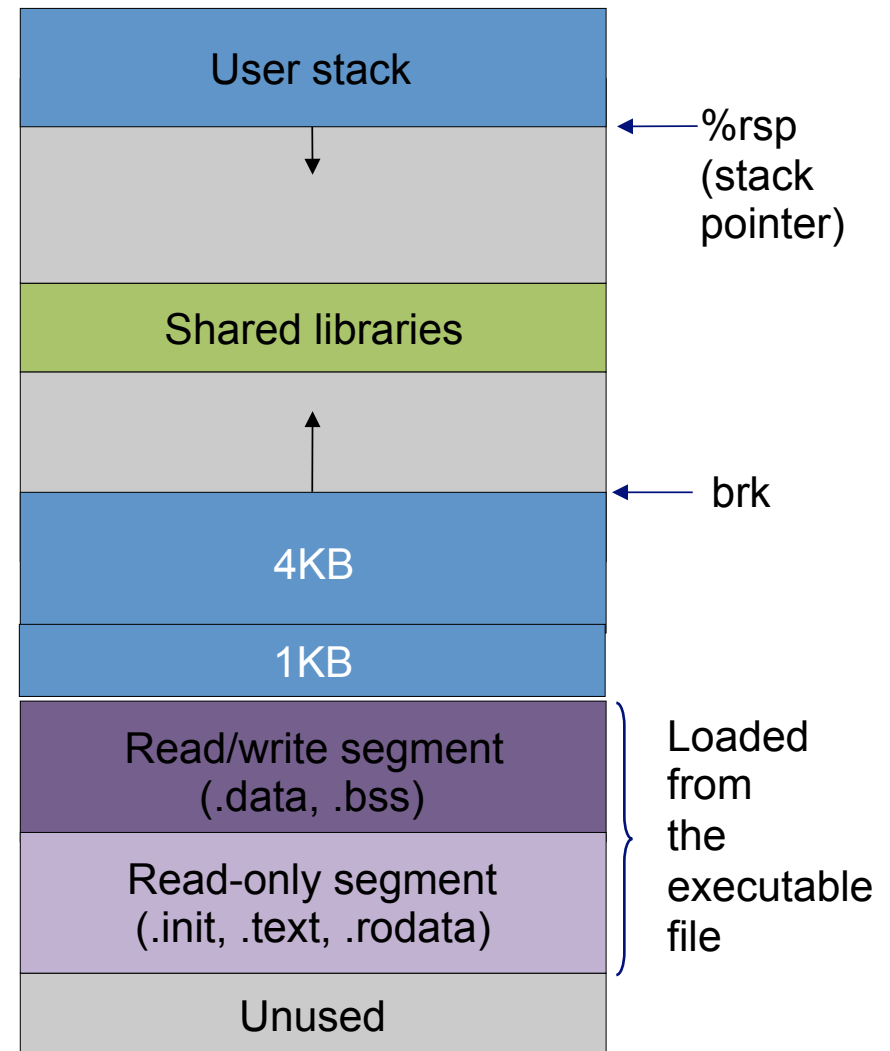
Ask OS for allocation on the heap via **system calls**

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

Issue 1 – can only free the memory on the top of heap

```
p1 = sbrk(1024) //allocate 1KB  
p2 = sbrk(4096) //allocate 4KB
```

How to free p1?



Dynamic allocation on heap

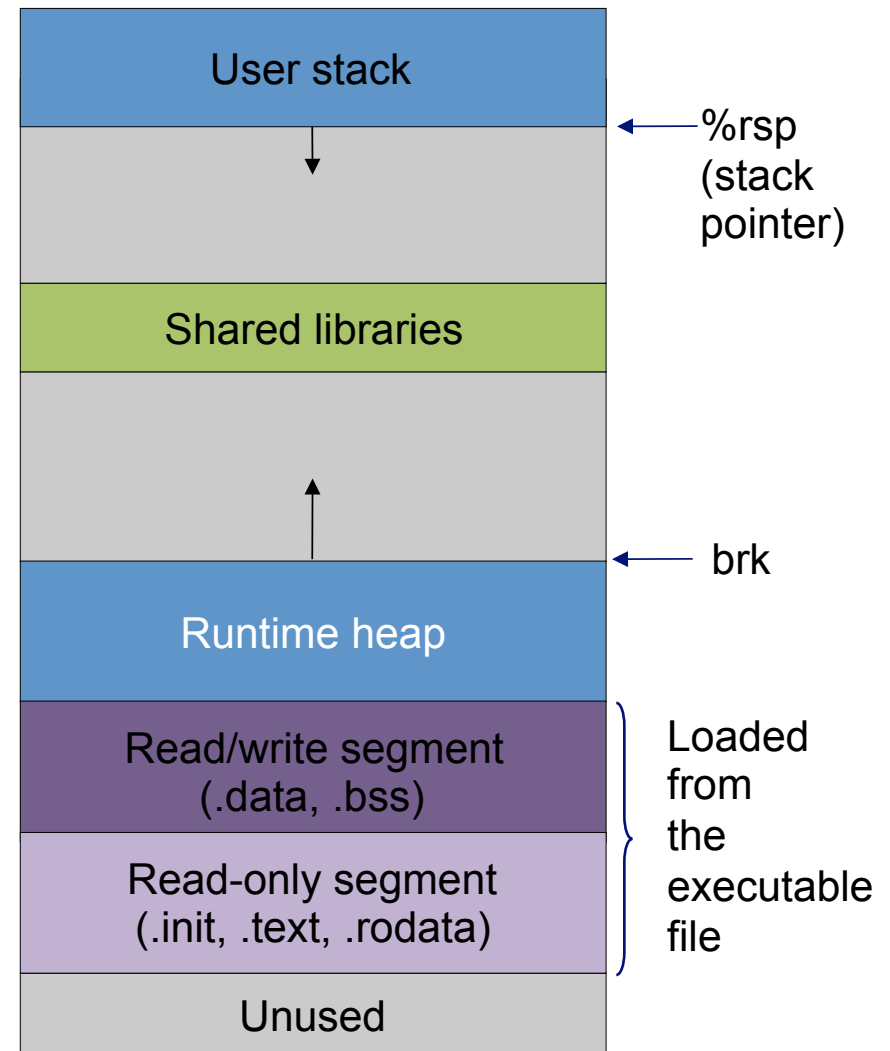
Question: How to allocate memory on heap?

Ask OS for allocation on the heap via **system calls**

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

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

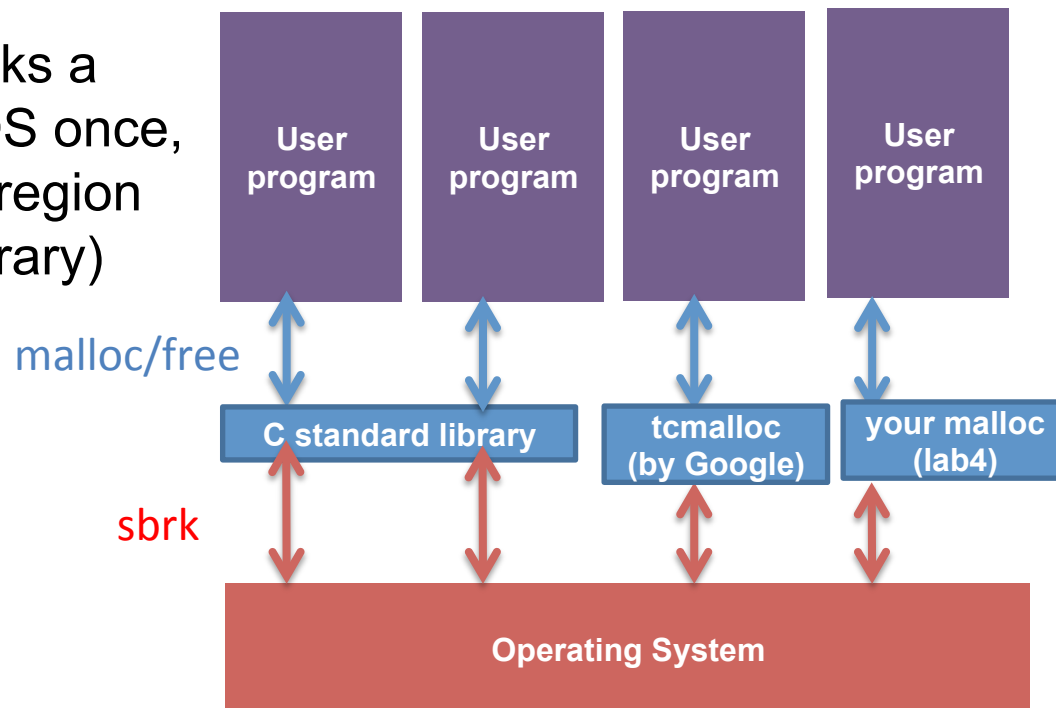
Issue II – system call has high performance cost > 10X



Dynamic allocation on heap

Question: How to efficiently allocate memory on heap?

Basic idea: user program asks a large memory region from OS once, then manages this memory region by itself (using a “malloc” library)



How to implement a memory allocator?

API:

- `void* malloc(size_t size);`
- `void free(void *ptr);`

Goal:

- Efficiently utilize 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

How to implement a memory allocator?

Assumed behavior of applications:

- Issue an arbitrary sequence of malloc/free
- Argument of free must be the return value of a previous malloc
- No double free

Restrictions on the allocator:

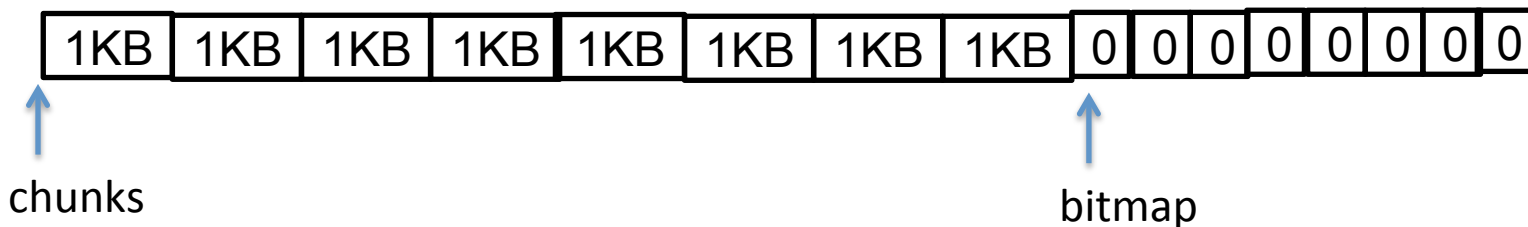
- Once allocated, space cannot be moved around

Malloc design challenges

1. (Basic book-keeping) How to keep track which bytes are free and which are not?
2. (Allocation decision) Which free chunk to allocate?
3. (API restriction) free is only given a pointer, how to find out the allocated chunk size?

How to bookkeep? Strawman #1

- Structure heap as n 1KB chunks + n metadata

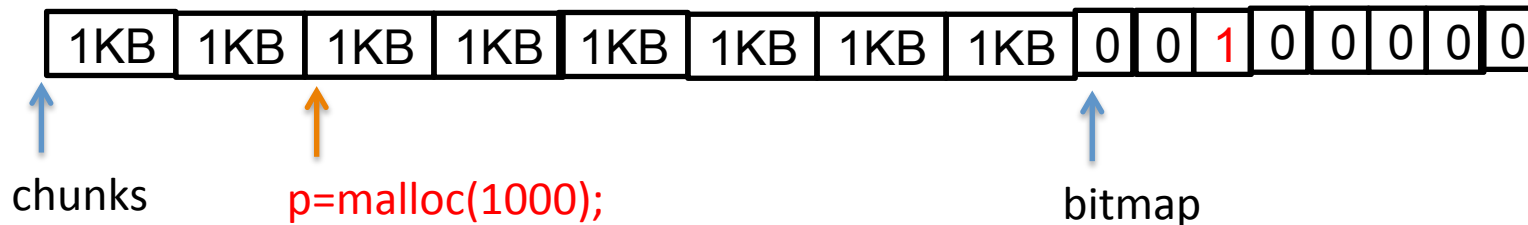


```
#define CHUNKSIZE 1<<10;
typedef char[CHUNKSIZE] chunk;
char *bitmap;
chunk *chunks;
size_t n_chunks;

void init() {
    n_chunks = 128;
    sbrk(n_chunks*sizeof(chunk)+ n_chunks/8);
    chunks = (chunk *)heap_lo();
    bitmap = heap_lo() + n_chunks *CHUNKSIZE;
}
```

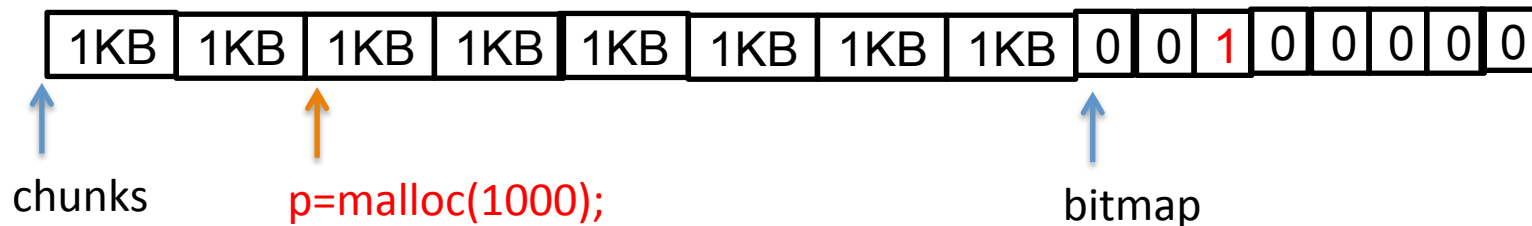
Assume allocator asks for
enough memory from OS
in the beginning

How to bookkeep? Strawman #1



```
void* malloc(size_t sz) {  
    // find out # of chunks needed to fit sz bytes  
    CSZ = ...  
  
    //find csz consecutive free chunks according to bitmap  
    int i = find_consecutive_chunks(bitmap);  
  
    // return NULL if did not find csz free consecutive chunks  
    if (i < 0)  
        return NULL;  
  
    // set bitmap at positions i, i+1, ... i+csz-1  
    bitmap_set_pos(bitmap, i, csz);  
    return (void *)&chunks[i];  
}
```


How to bookkeep? Strawman #1



```
void free(void *p) {  
    i = ((char *)p - (char *)chunks)/sizeof(chunk);  
    bitmap_clear_pos(bitmap, i); //how many bits to clear??  
}
```

- Problem with strawman?
 - free does not know how many chunks allocated
 - wasted space within a chunk (internal fragmentation)
 - wasted space for non-consecutive chunks (external fragmentation)

How to bookkeep? Other Strawmans

- How to support a variable number of variable-sized chunks?
 - Idea #1: use a hash table to map address → [chunk size, status]
 - Idea #2: use a linked list in which each node stores [address, chunk size, status] information.

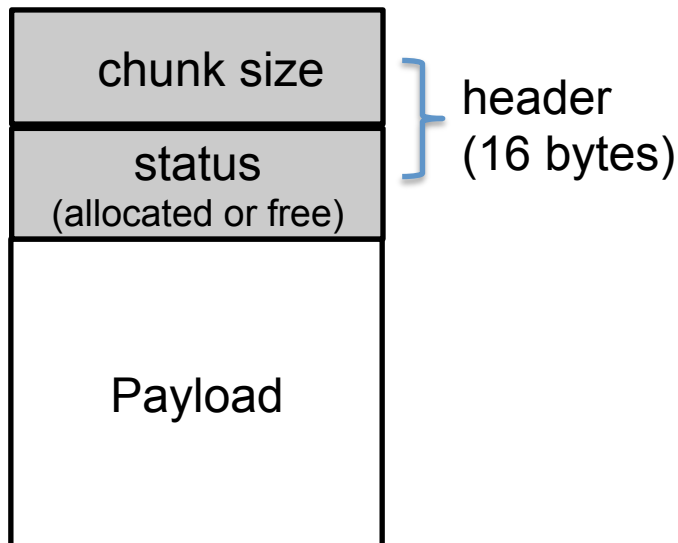
Problems of strawmans?

Implementing a hash table and linked list requires use of a dynamic memory allocator!

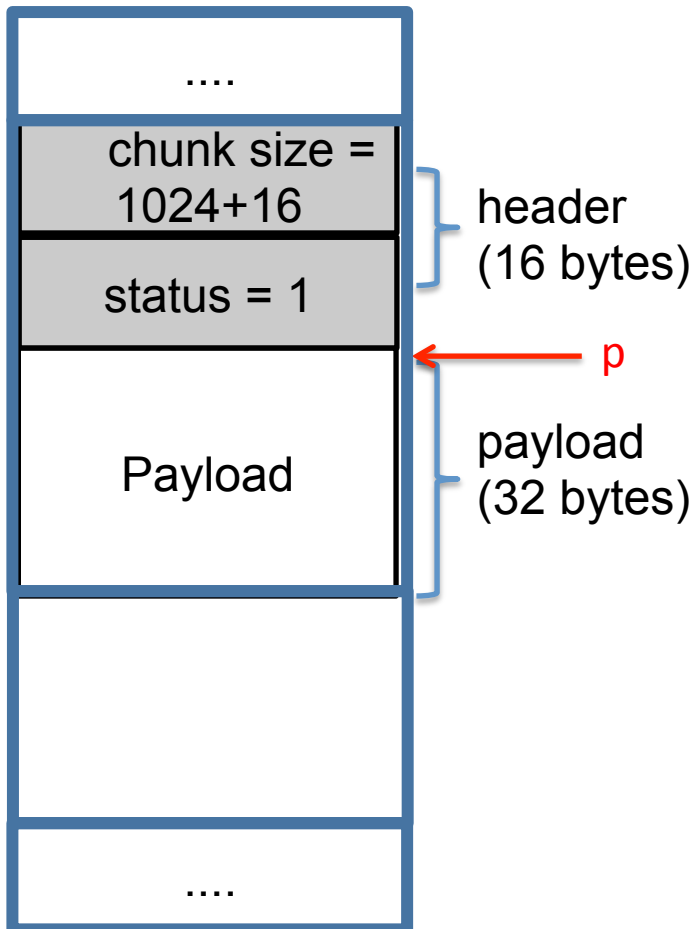
How to implement a “list” without use of malloc

Implicit list

- Each chunk contains metadata + payload
 - Metadata (chunk header) stores chunk size and status
- Alignment requirement: 16-byte (aka the starting address of payload must be multiples of 16).
 - Make header 16-byte in size
 - Make chunk multiples of 16 in size.

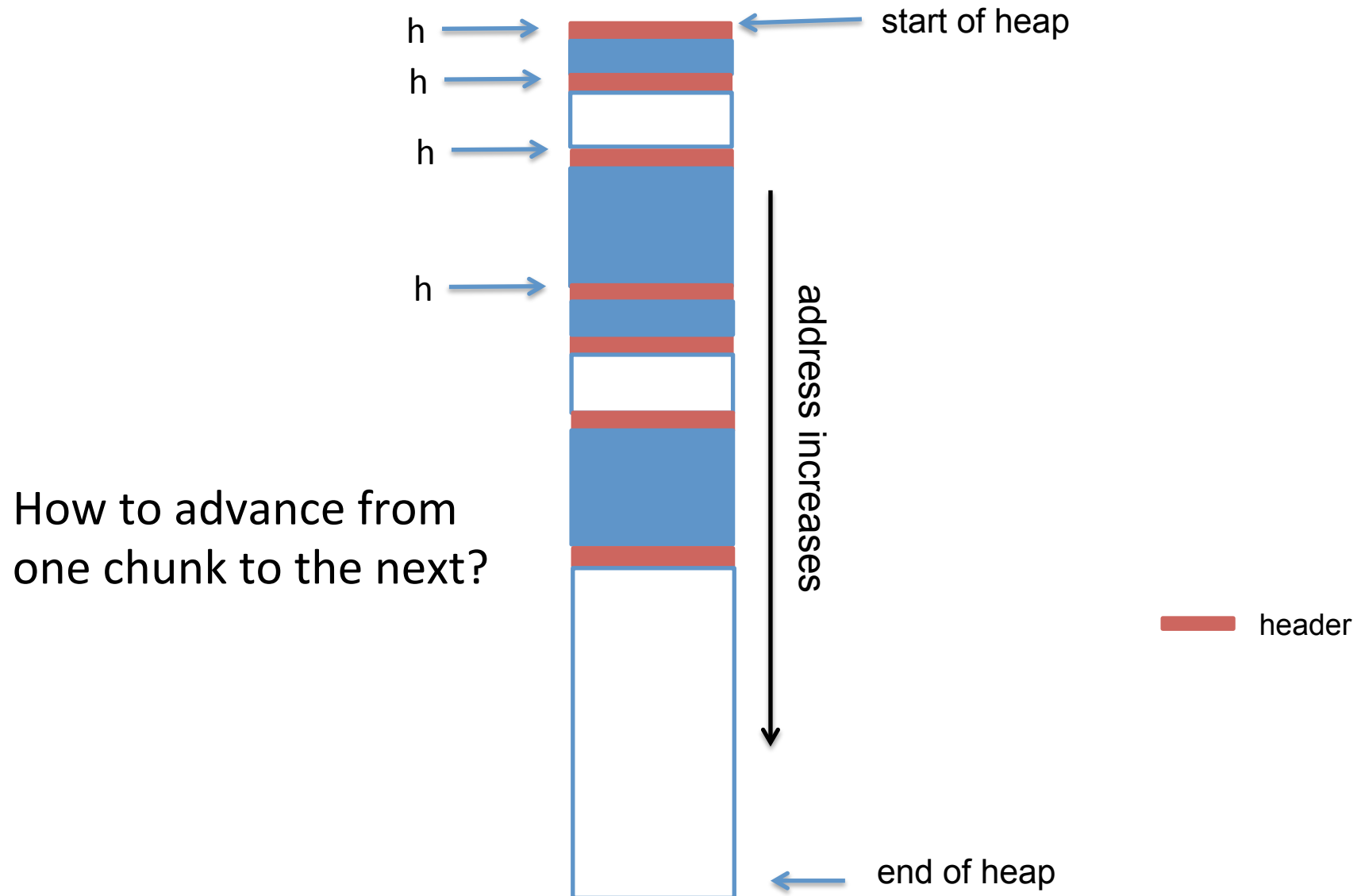


Implicit list: chunk format



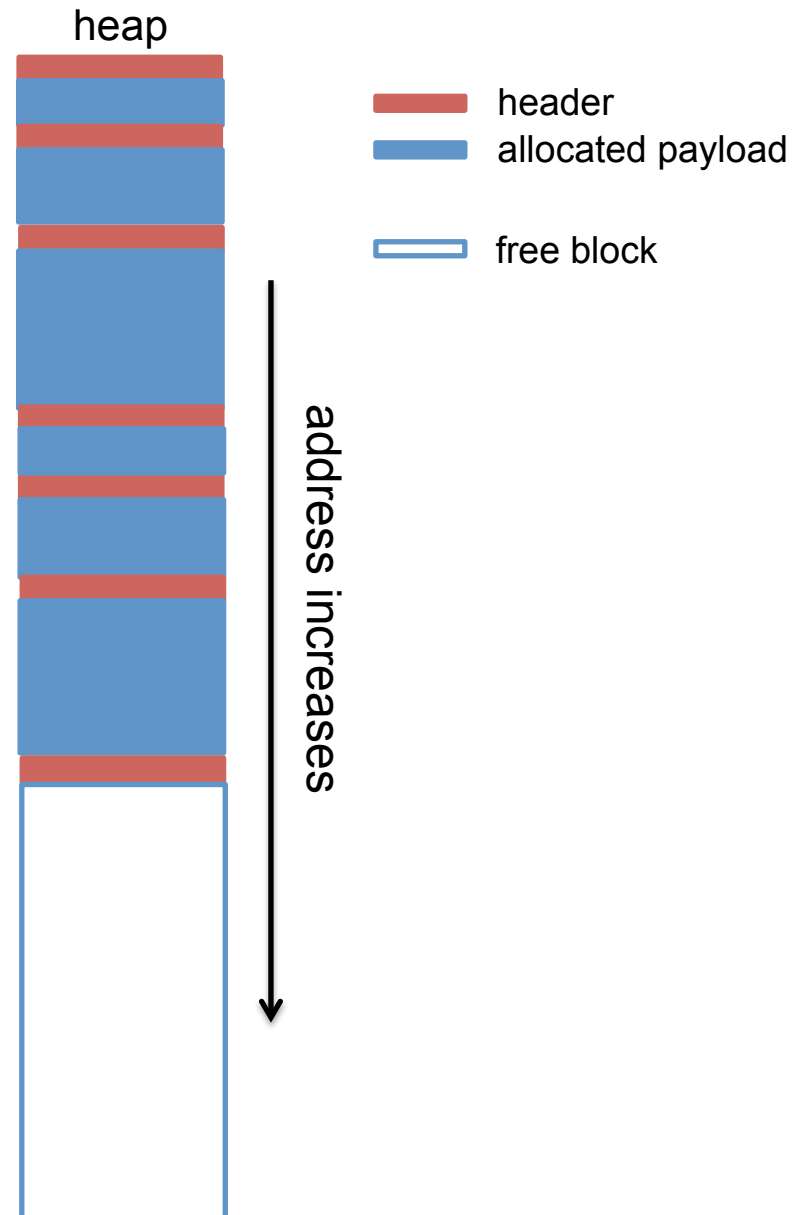
```
p = malloc(20)
```

Implicit list: heap traversal



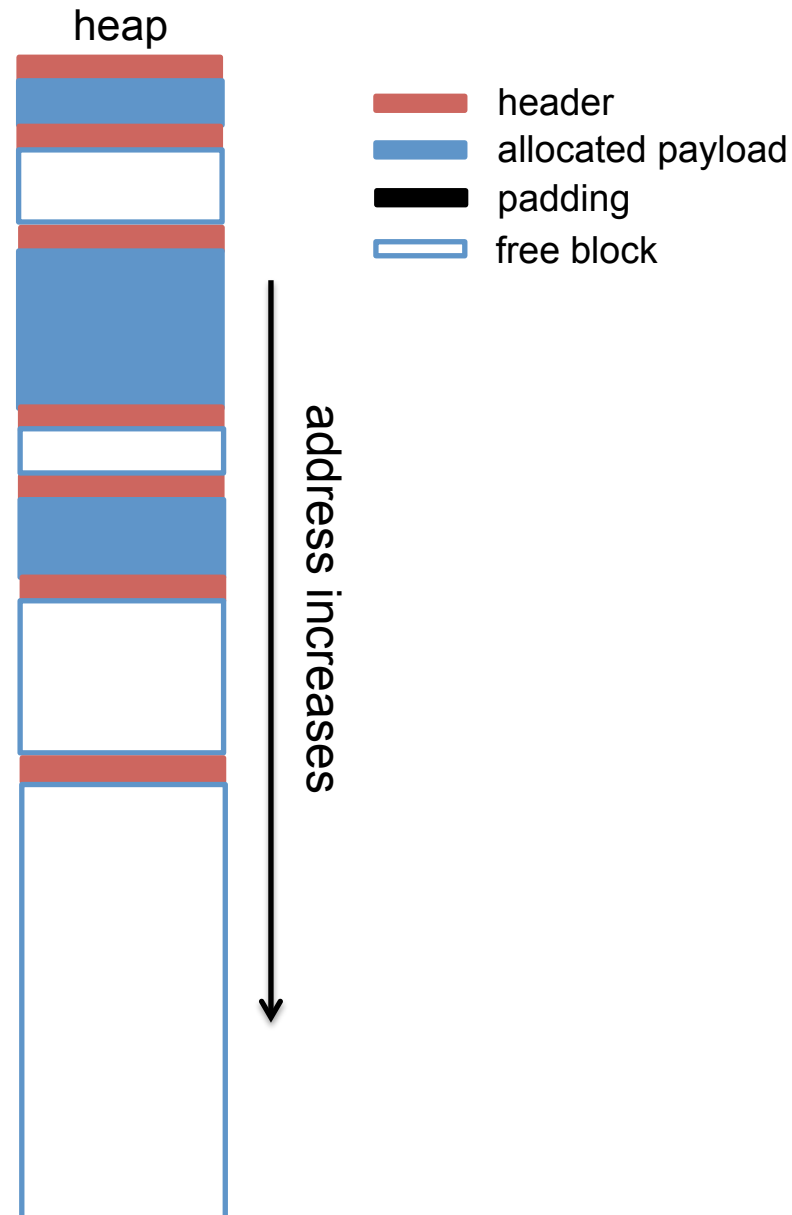
Which chunk to allocate?

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



Where to place an allocation?

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



First fit

```
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)
```

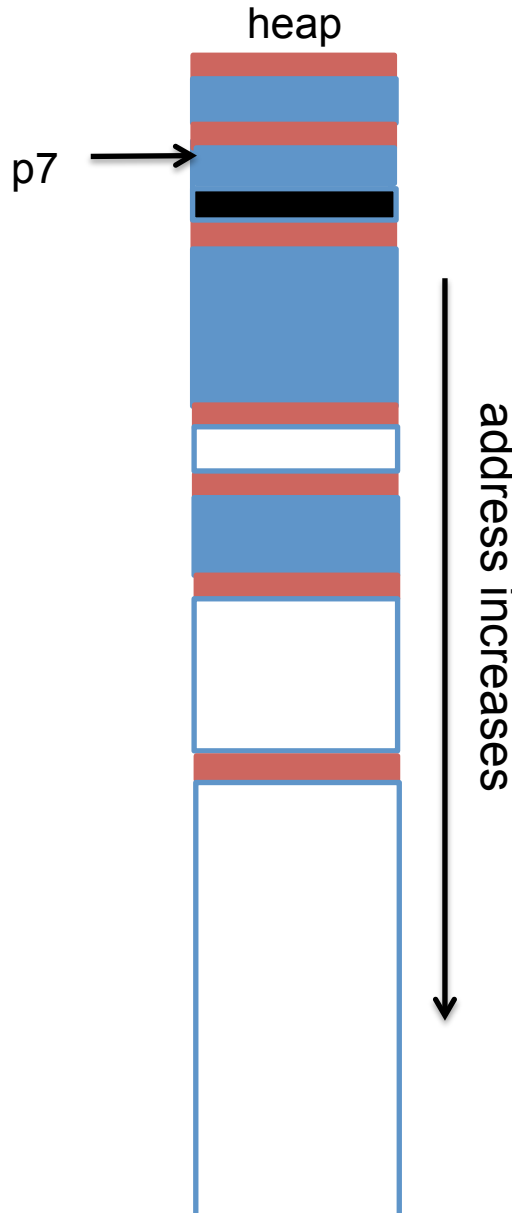


- header
- allocated payload
- padding
- free block

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

First fit

```
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)
```

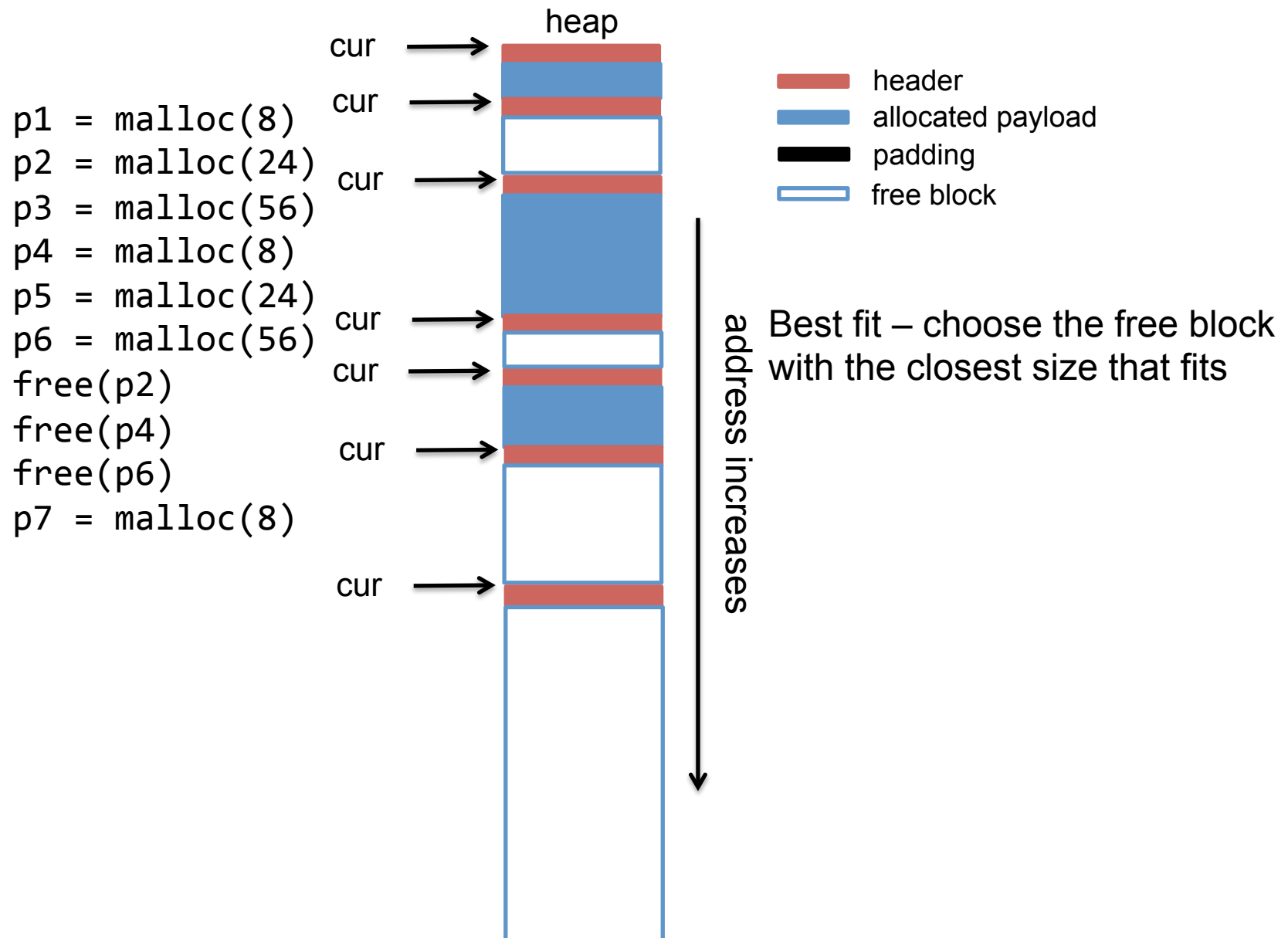


- header
- allocated payload
- padding
- free block

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

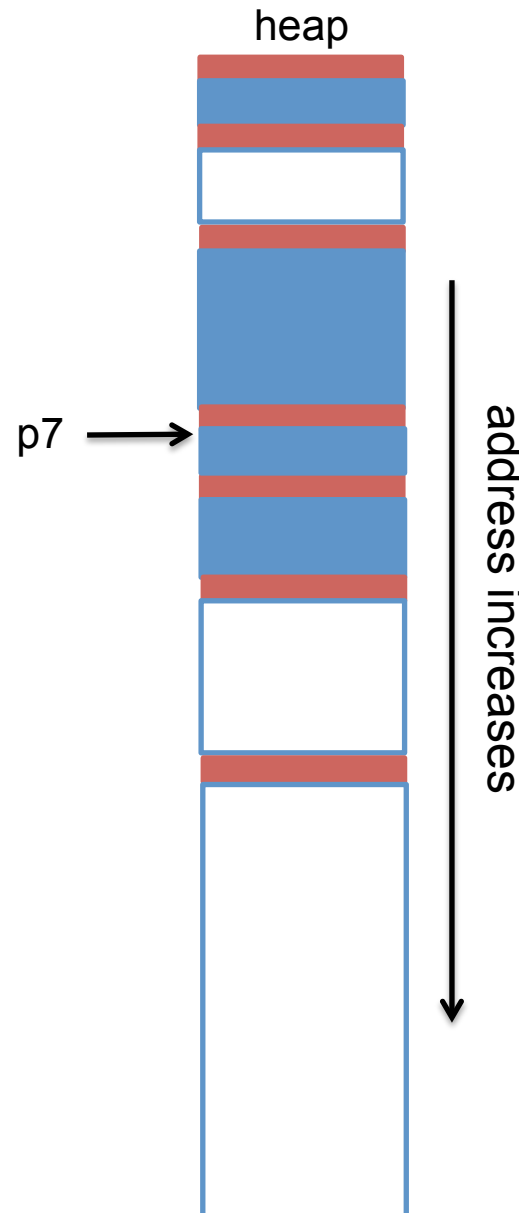
Downside: cause fragmentation at beginning of the heap

Best fit



Best fit

```
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)
```



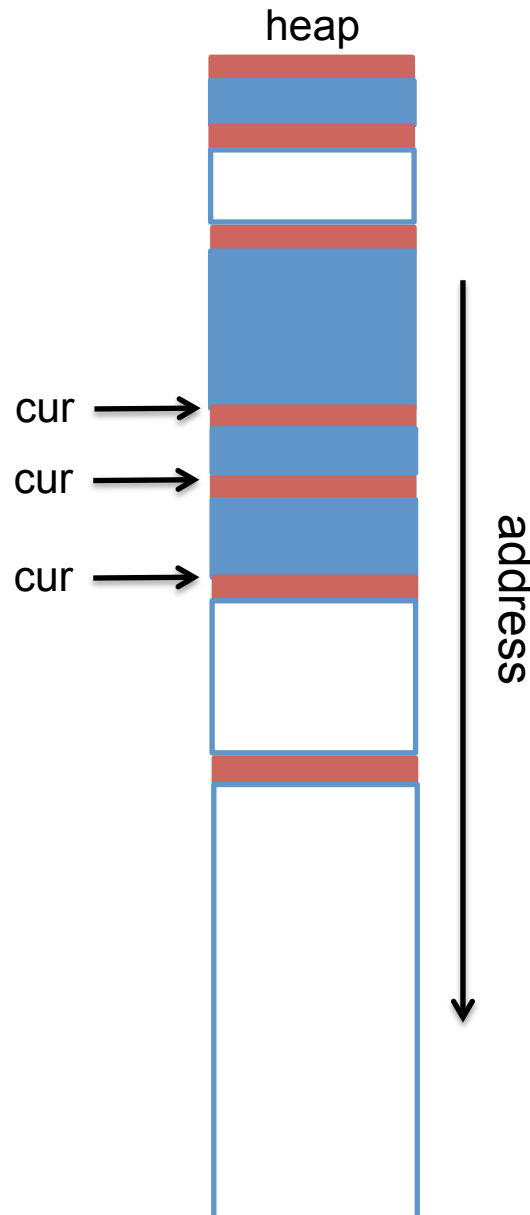
- header
- allocated payload
- padding
- free block

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

Downside: run slower than first fit.

Next fit

```
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)
```

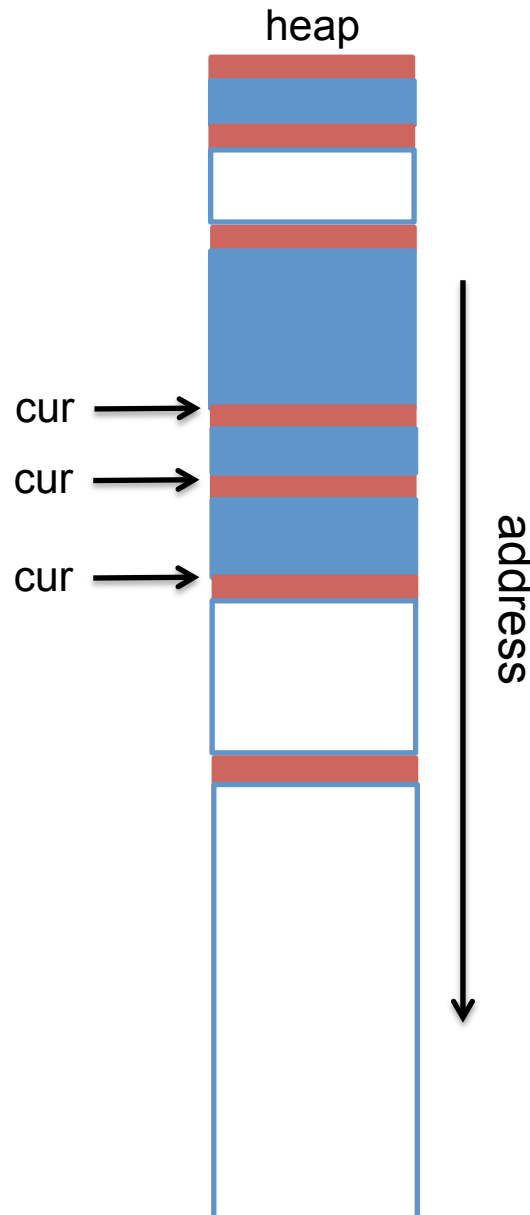


header
allocated payload
padding
free block

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

Next fit

```
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)
```

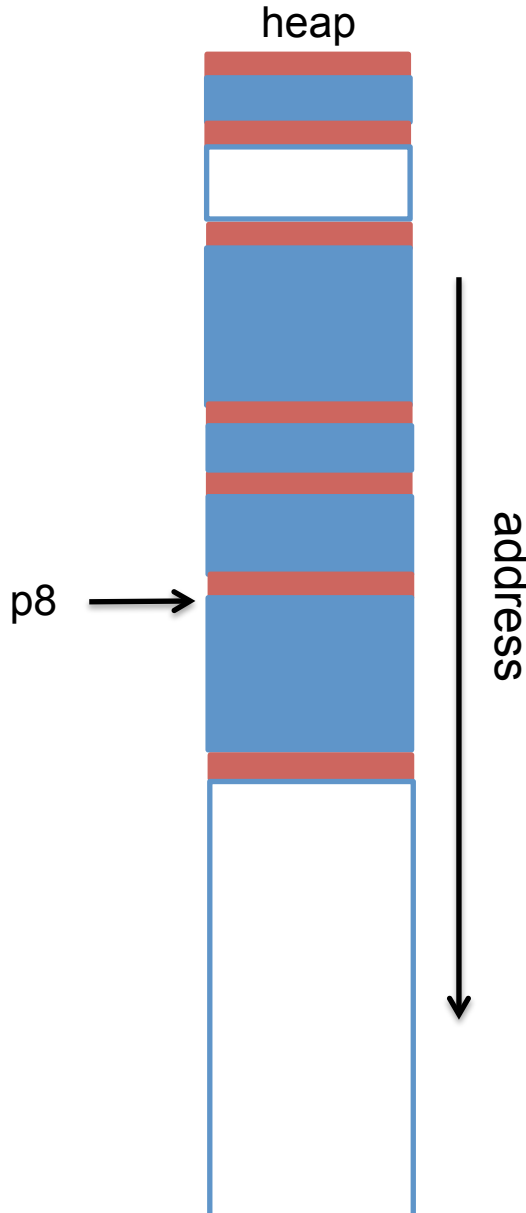


header
allocated payload
padding
free block

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

Next fit

```
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)
```

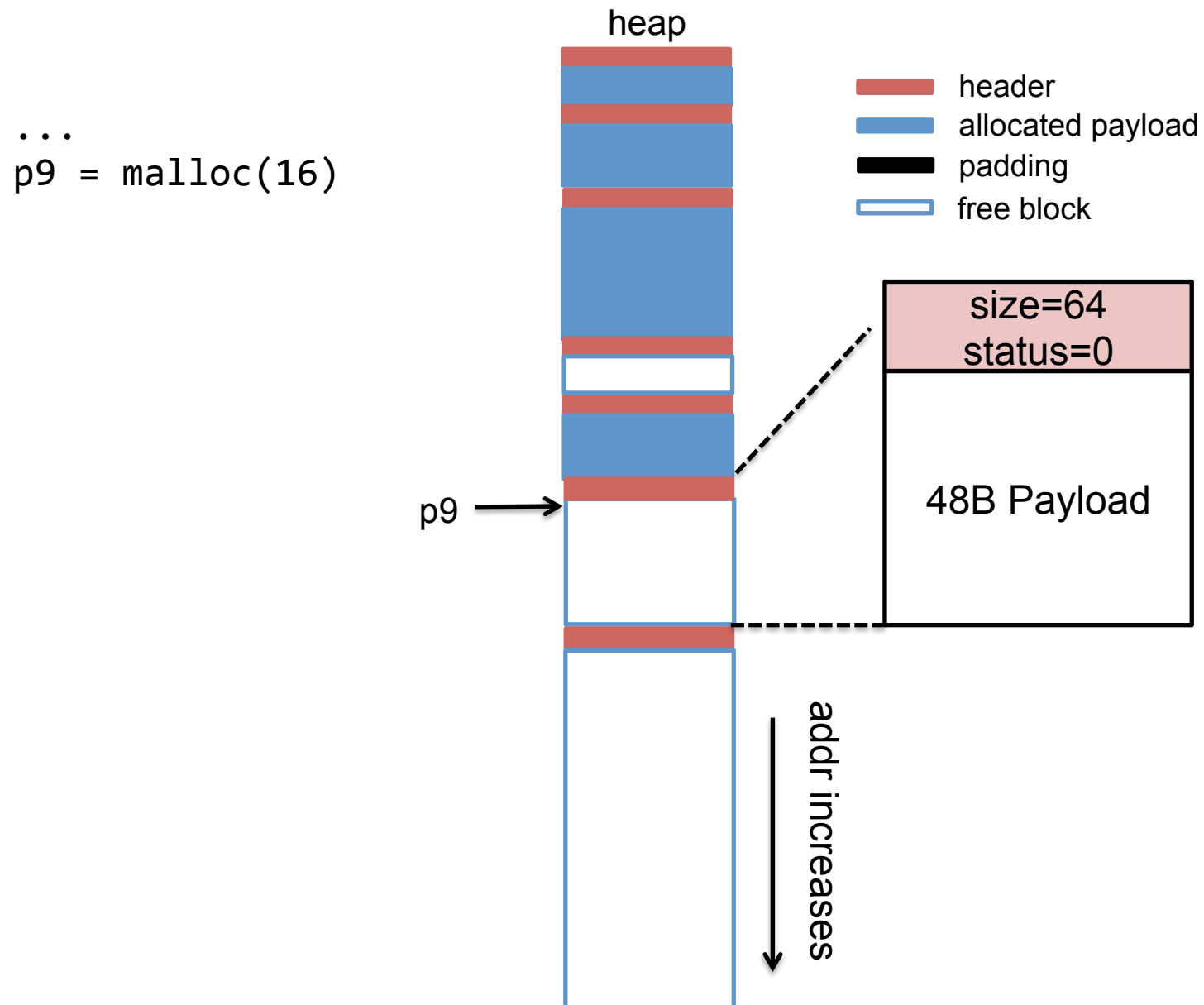


- header
- allocated payload
- padding
- free block

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

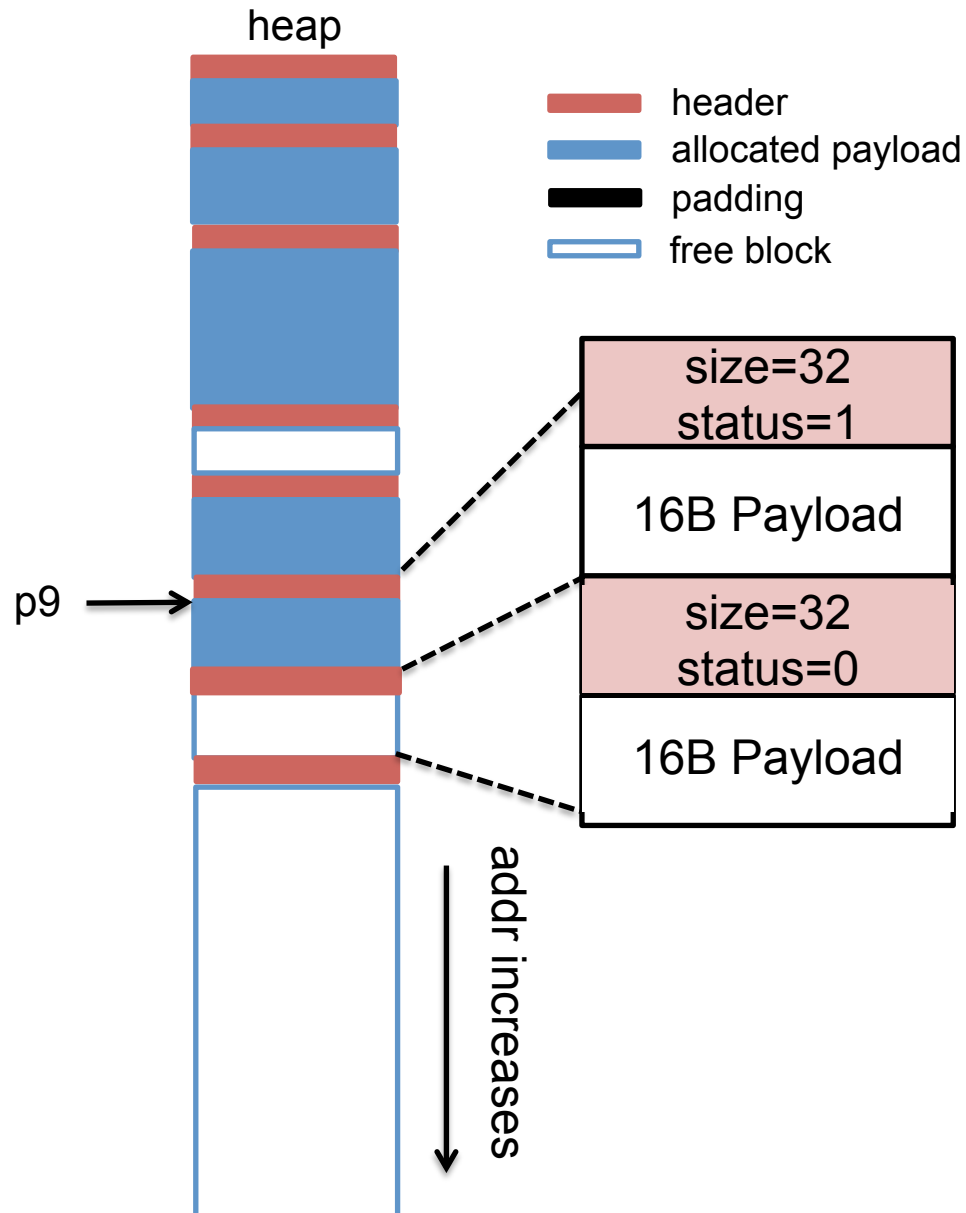
Next fit runs faster than first fit,
but fragmentation is worse.

Splitting a free block

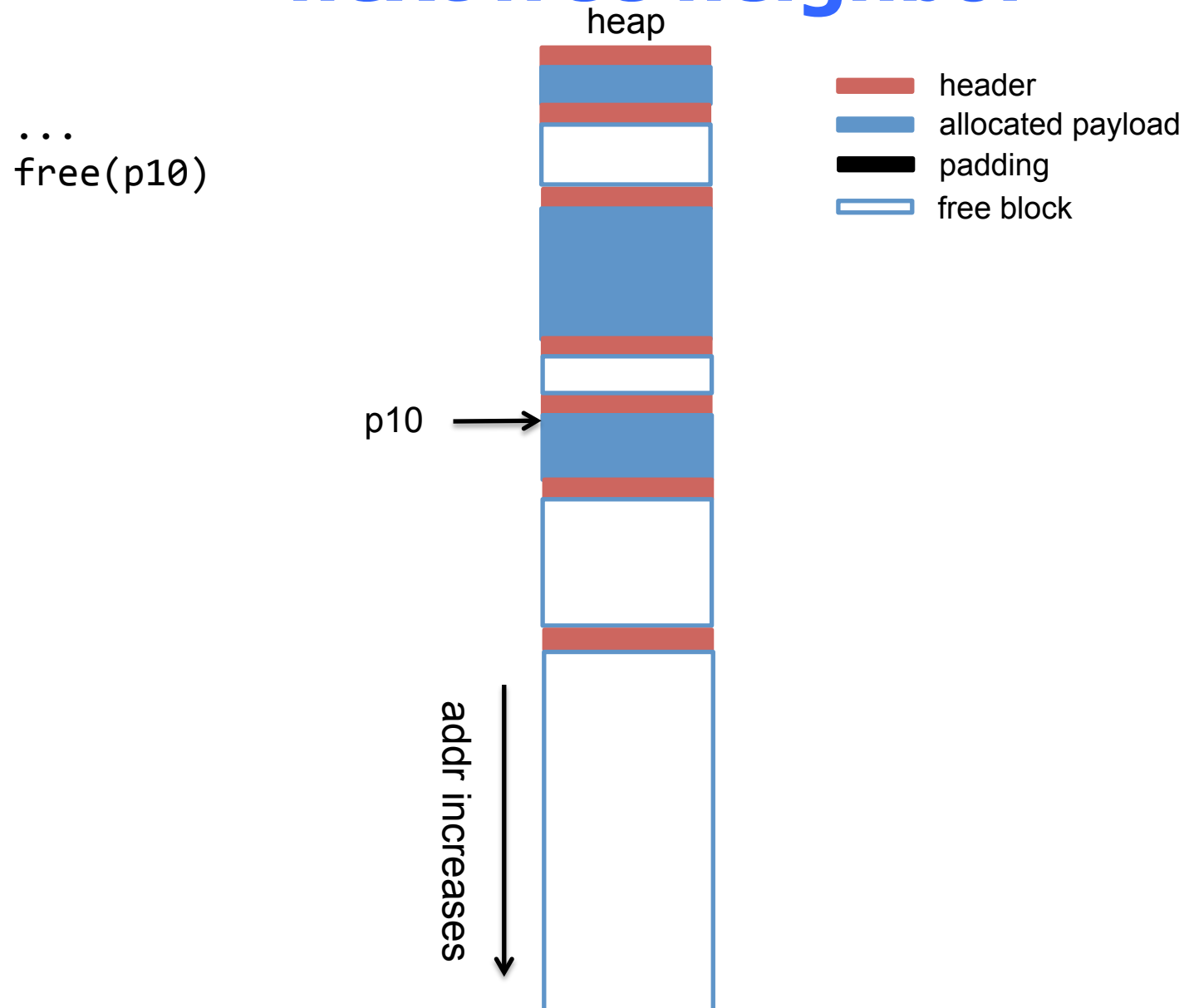


Splitting a free block

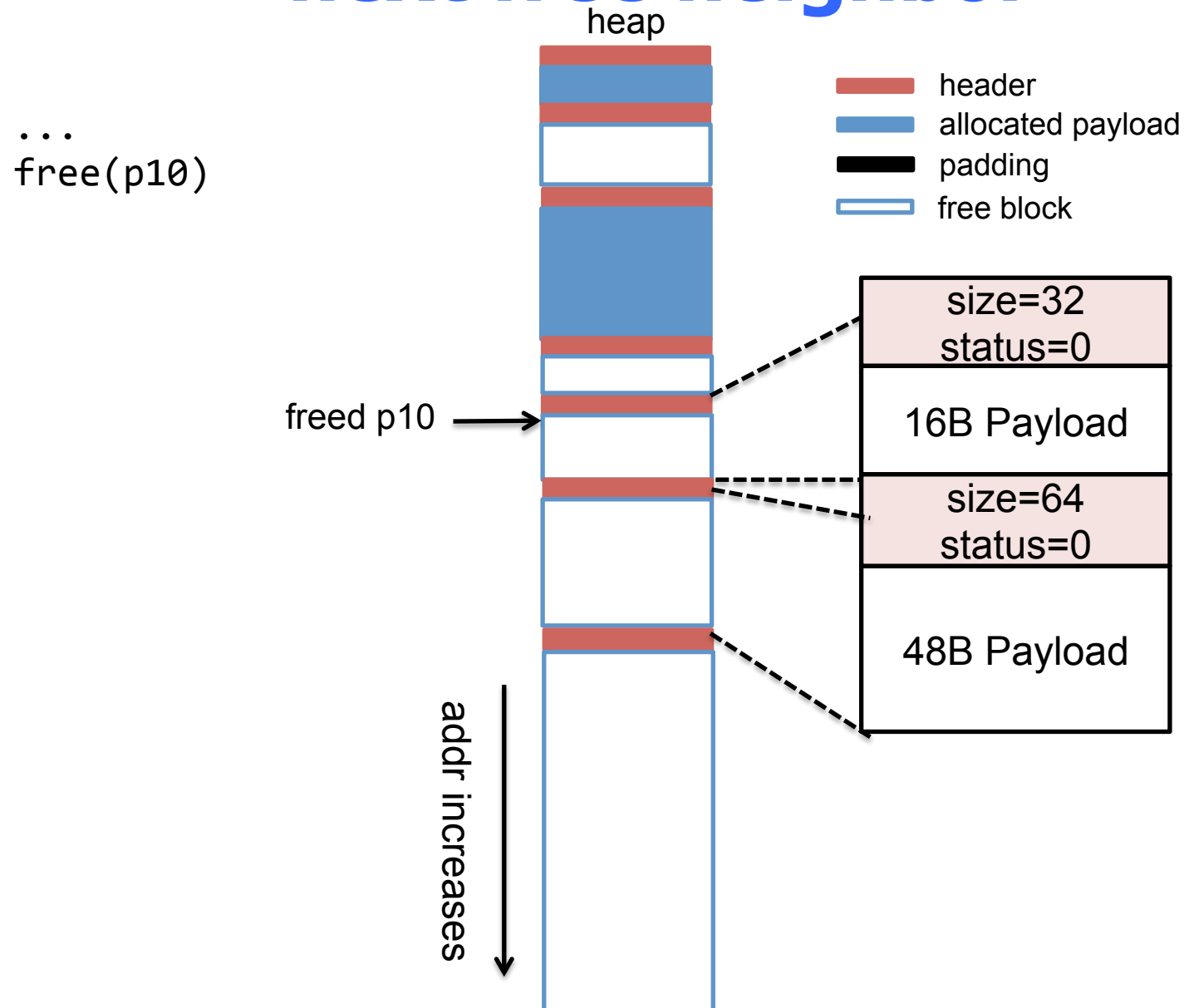
...
p9 = malloc(16)



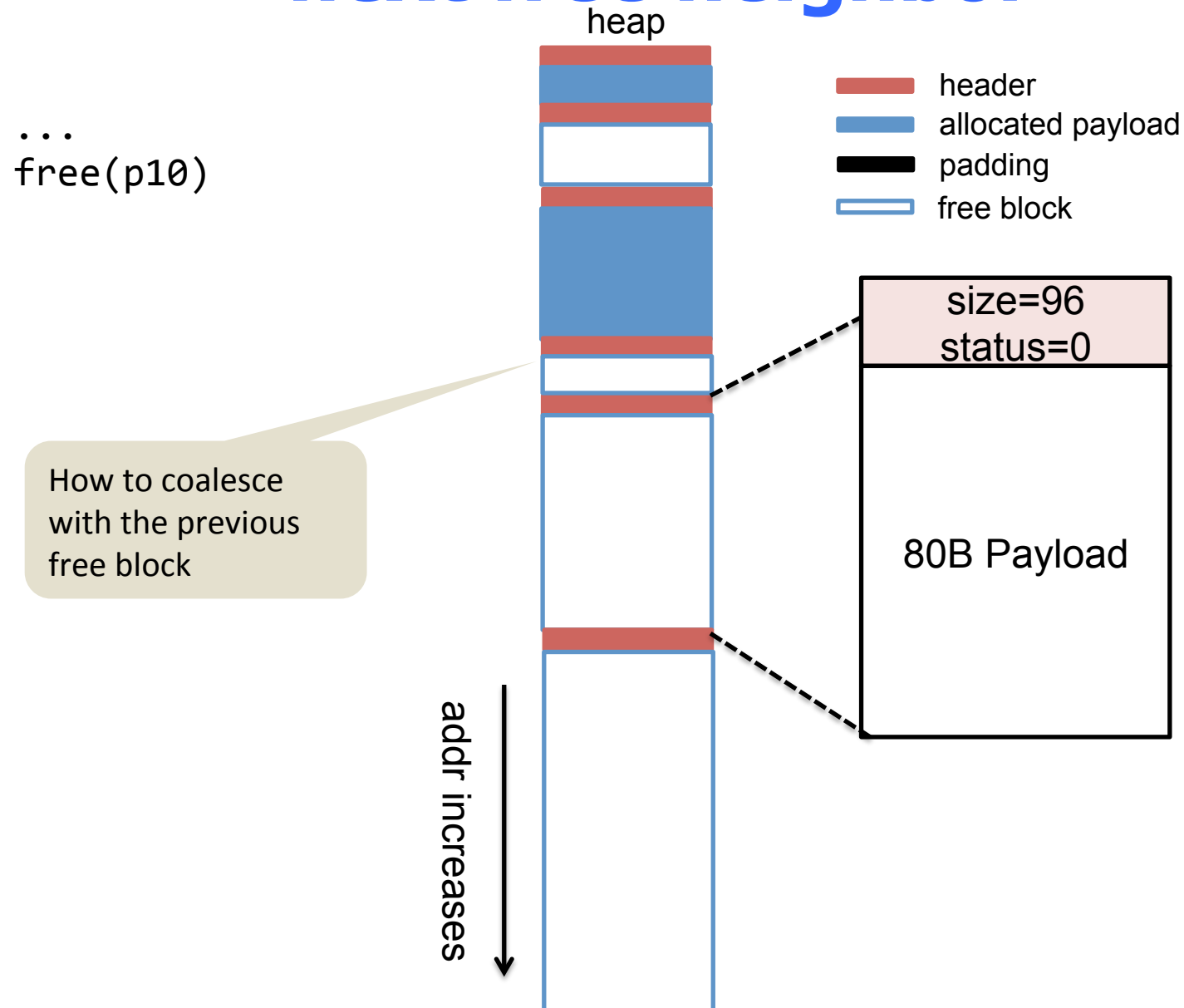
Coalescing a free block with its next free neighbor



Coalescing a free block with its next free neighbor



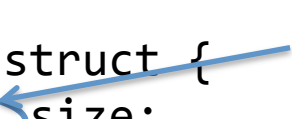
Coalescing a free block with its next free neighbor



implicit list impl (lab 4)

```
typedef struct {  
    size_t size;  
    size_t allocated;  
} header_t;
```

unsigned long



```
int mm_init();  
void *mm_malloc(size_t size);  
void mm_free(void *p);
```

implicit list implementation

```
size_t hdr_size = sizeof(header_t);
```

```
int init() {  
    return 0; //start with empty heap  
}
```

```
void *mm_malloc(size_t size) {  
  
    size = align(size);  
    size_t csz = size + hdr_size;  
    header_t *h = first_fit(csz); //find a free chunk  
    if (!h) {  
        h = ask_os_for_chunk(csz);  
    } else {  
        split(h, csz); //split if necessary  
    }  
    ... // set chunk status to be allocated  
}
```

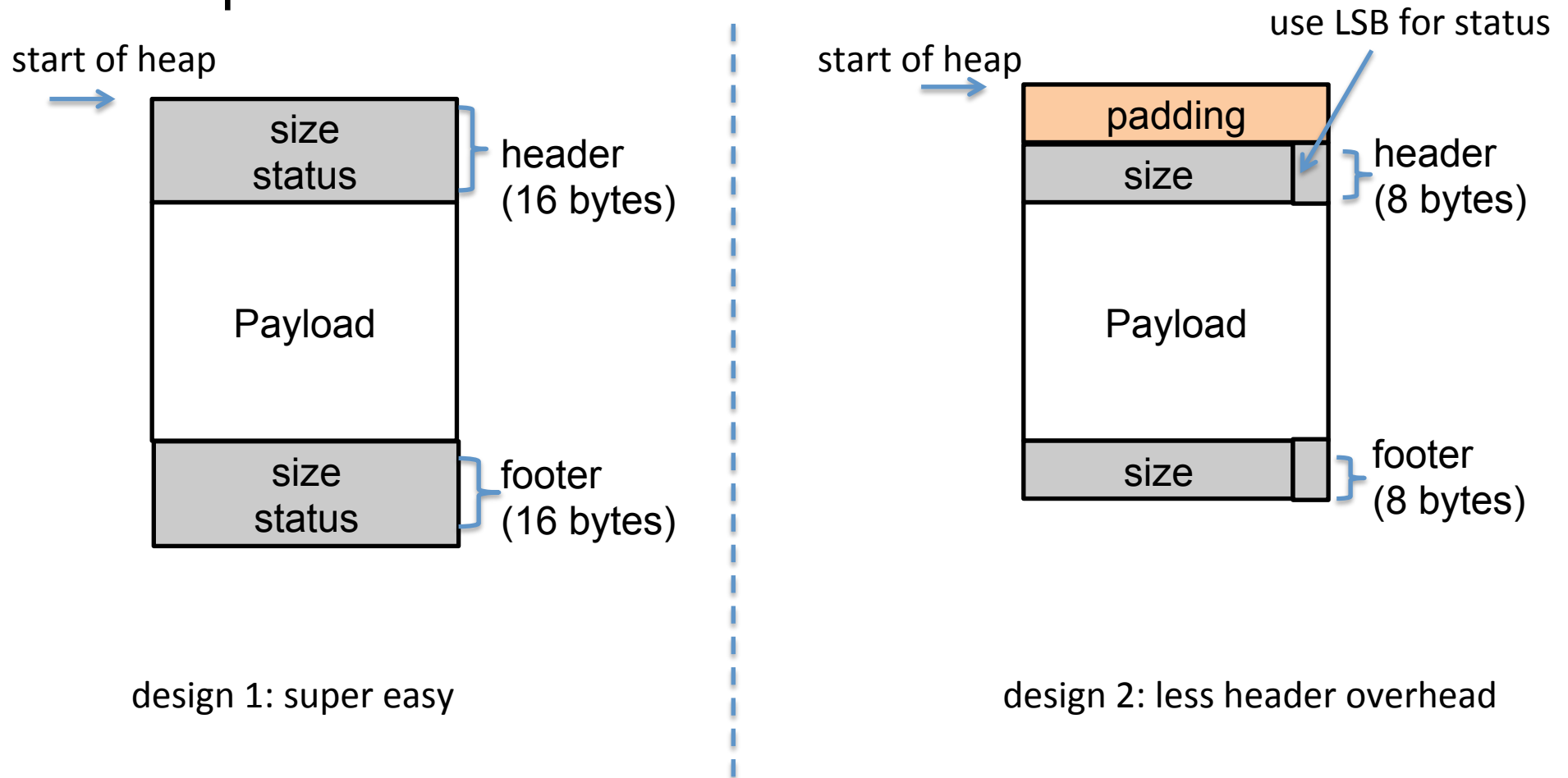
implicit list implementation

```
header_t *payload2header(void *p) {  
  
}
```

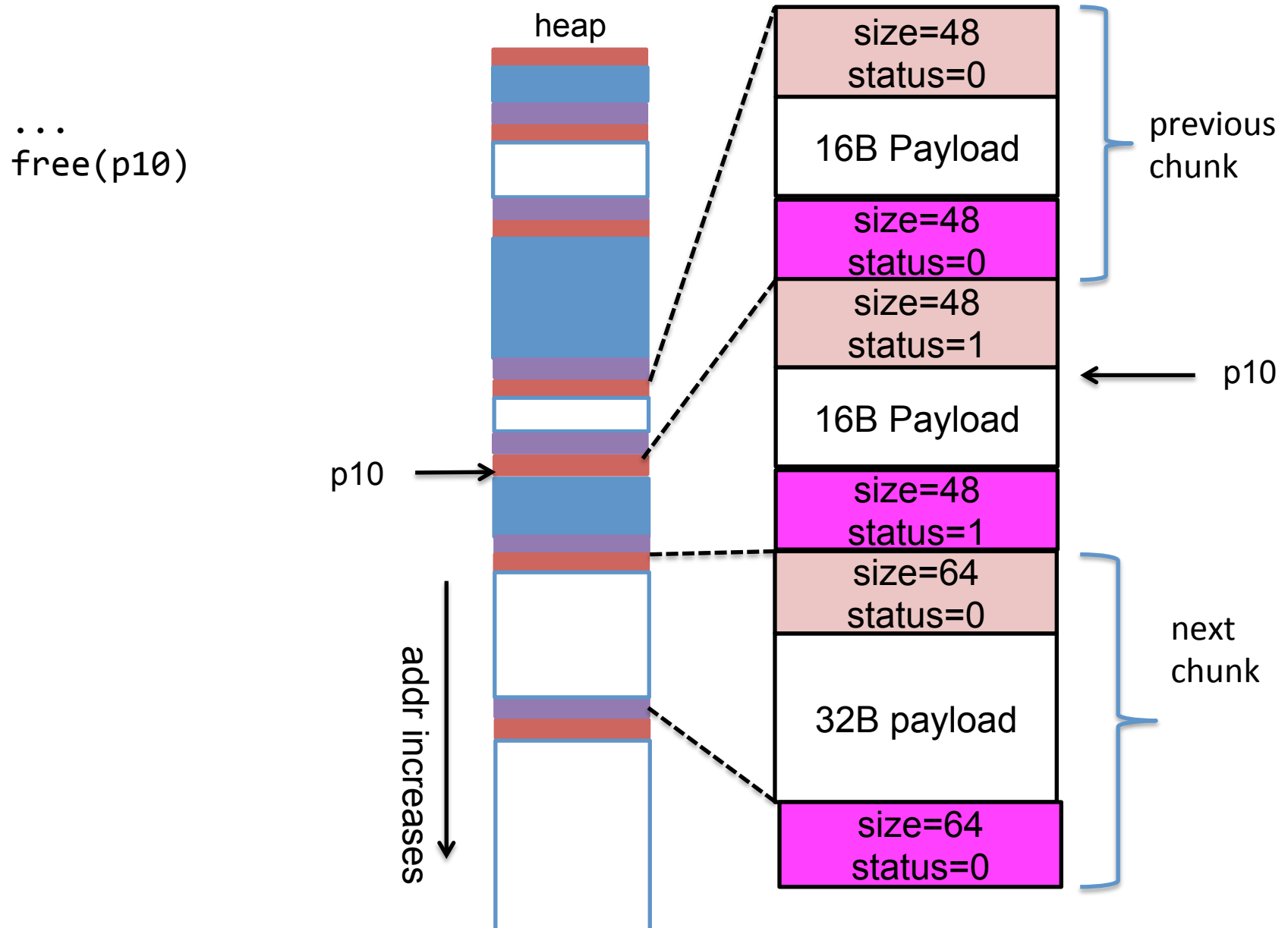
```
void *mm_free(void *p) {  
    header_t *h = payload2header(p);  
    ... //set chunk status to be free  
    coalesce(h);  
}
```

Use footer to coalesce with previous block

- Duplicate header information into the footer



Coalescing prev and next blocks



Coalescing prev and next blocks

...
free(p10)

