

Dynamic Memory Allocation

Jinyang Li

based on Tiger Wang's slides

What we've learnt: how C program is executed by hardware

- Compiler translates C programs to machine code
 - Basic execution:
 - Load instruction from memory, decode + execute, advance %rip
 - Control flow
 - Arithmetic instructions, cmp/test set RFLAGS
 - jge (...) changes %rip depending on RFLAGS
 - Procedure call
 - return address is stored on stack
 - %rsp points to top of stack (stack grows down)
 - call/ret
- Linking:
 - Combine multiple compiled object files together
 - Resolve and relocate symbols (functions, global variables)

Today's lesson plan

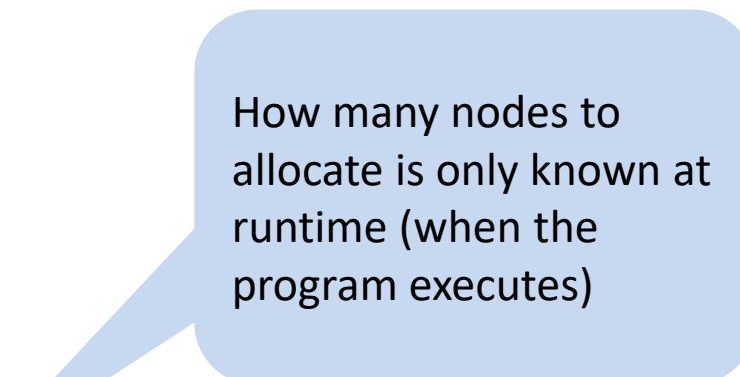
- dynamic memory allocation (malloc/free)

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;
}

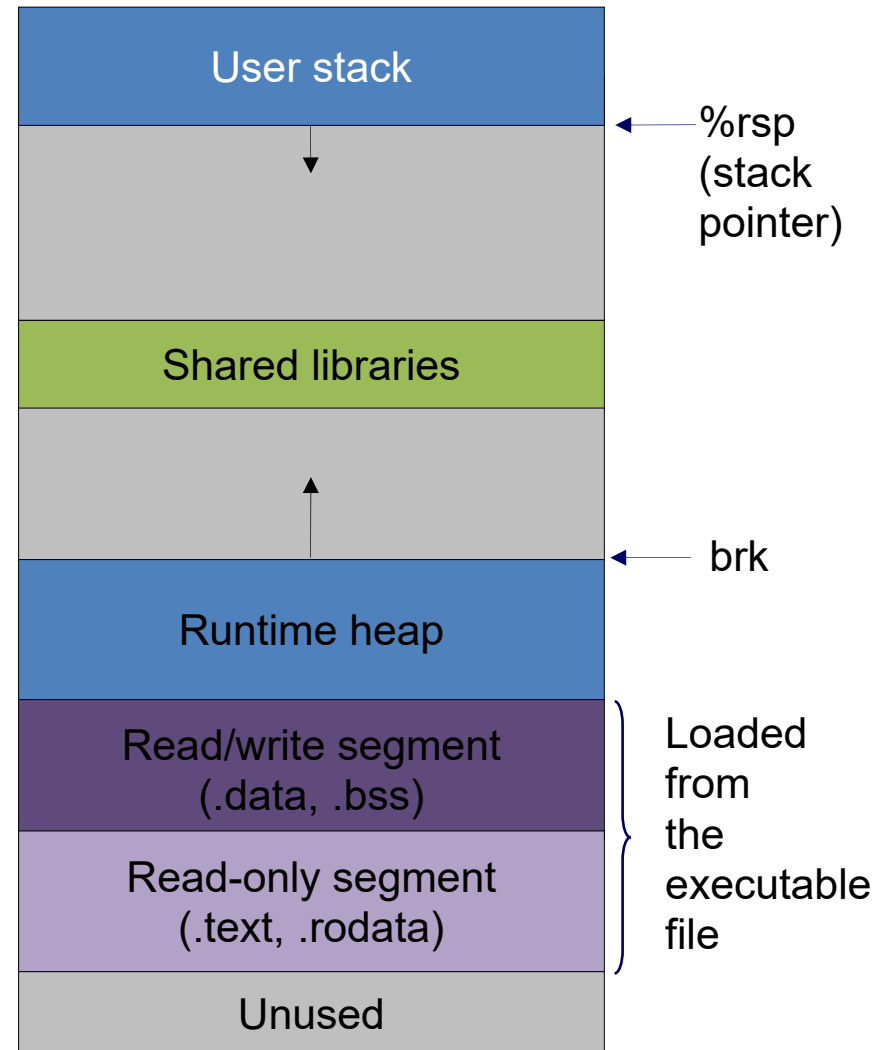
int main(void)
{
    char buf[100];
    node *head = NULL;
    while (fgets(buf, 100, stdin)) {
        list_insert(&head, atoi(buf));
    }
}
```



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?

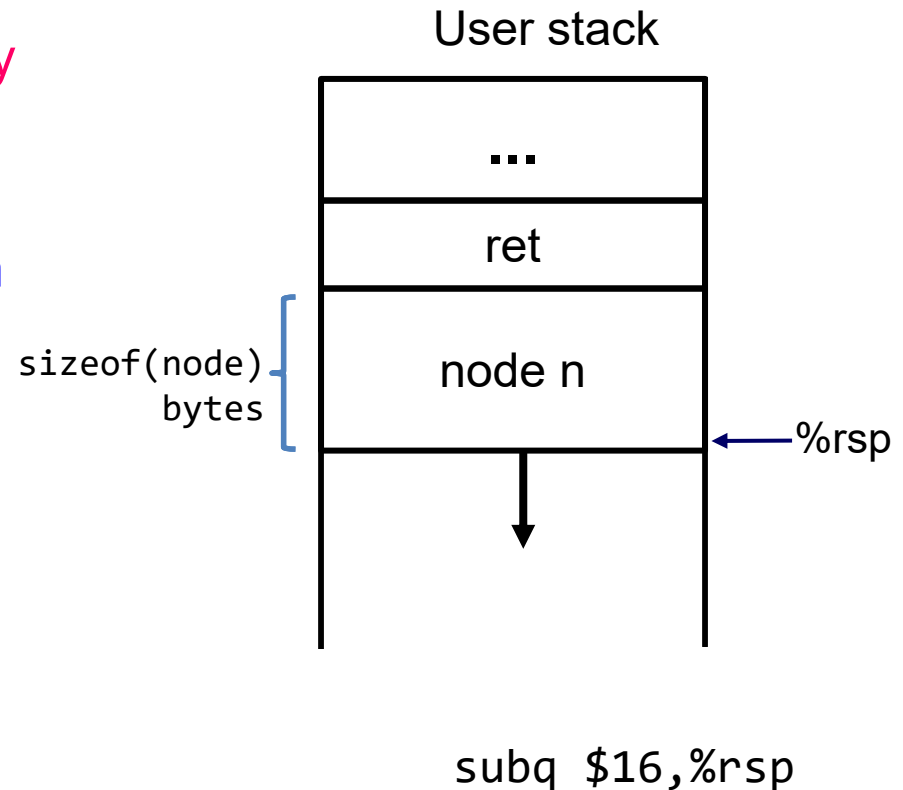


Dynamic allocation on heap

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

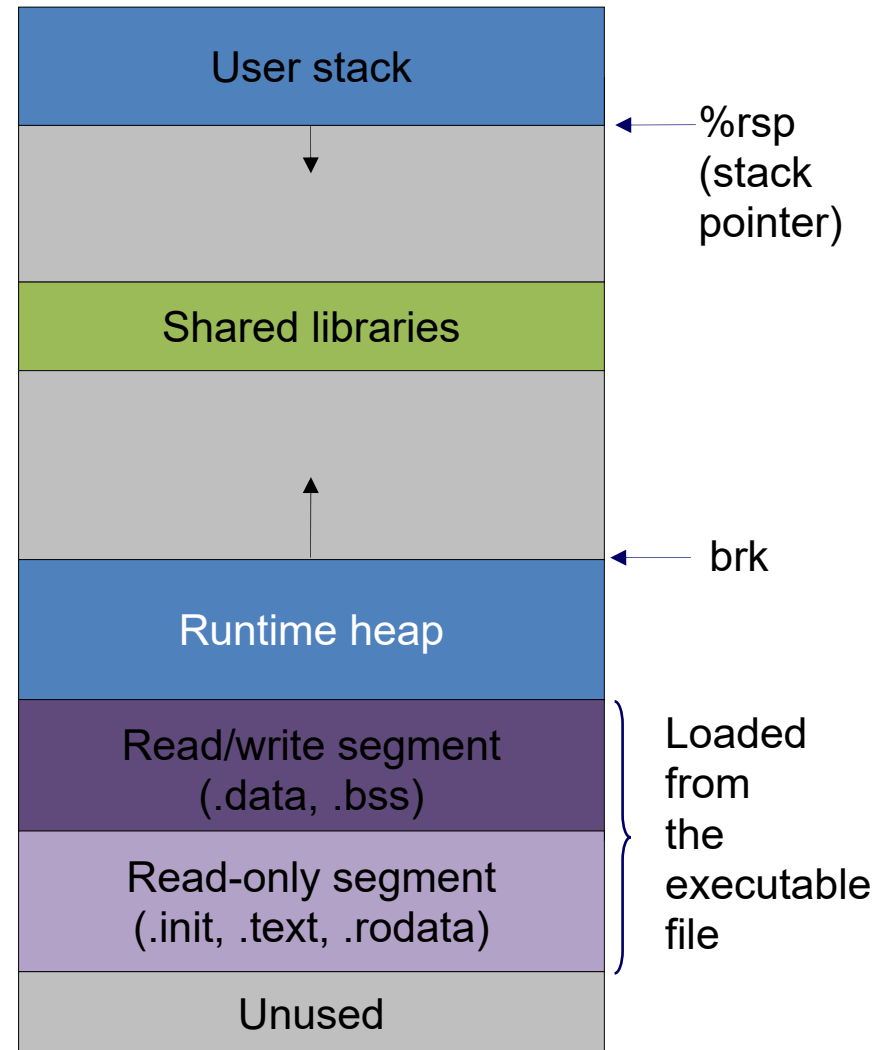
Answer: Yes, but space is freed upon function return

```
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?



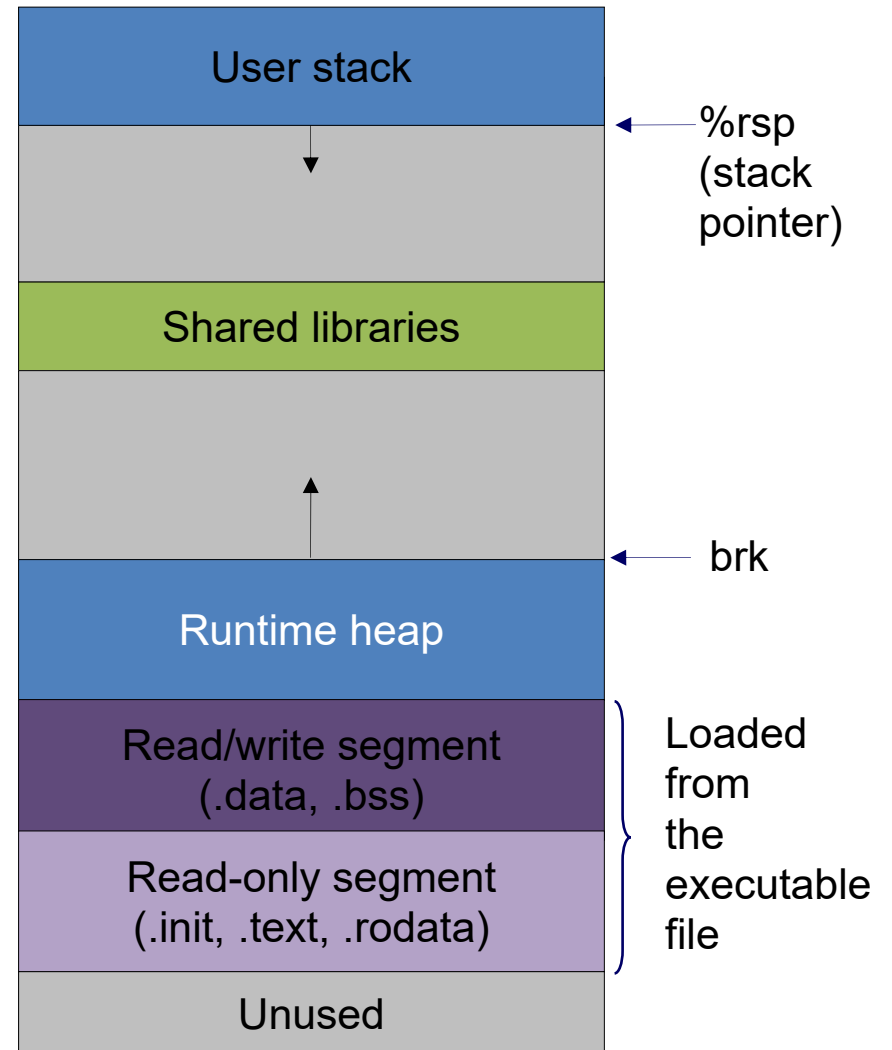
Dynamic allocation on heap

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

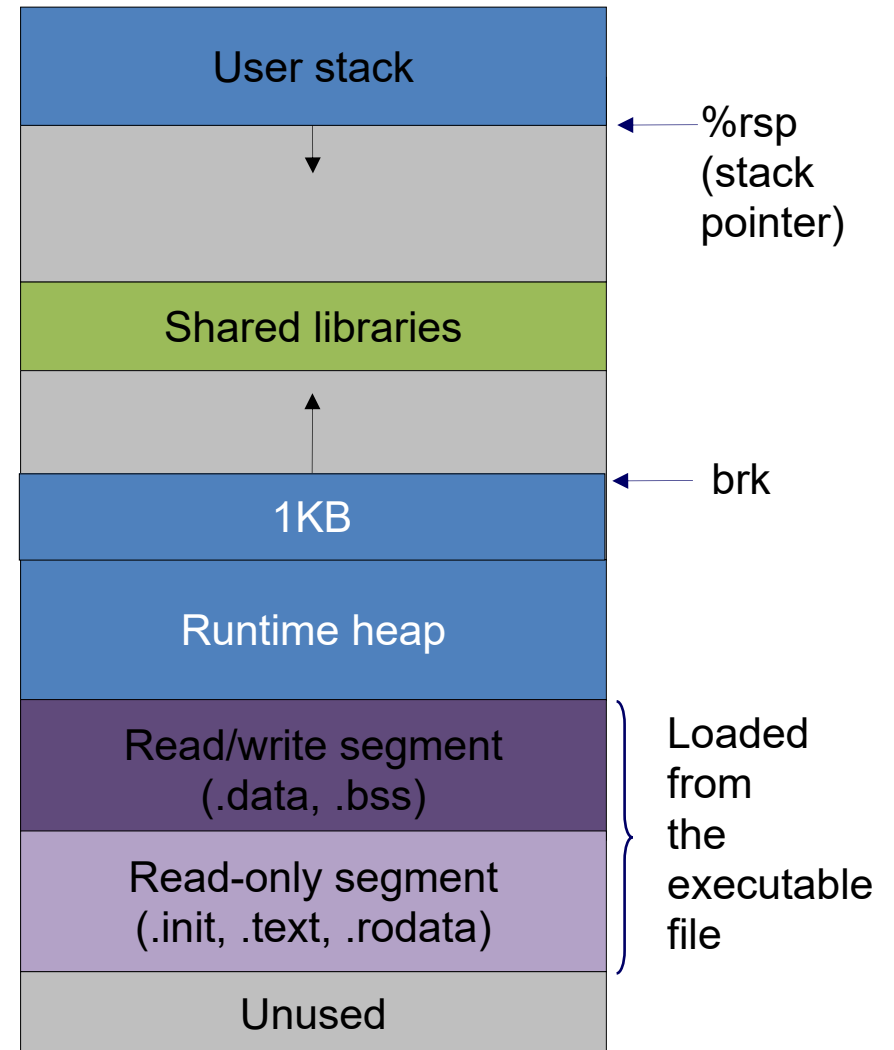
Question: How to allocate memory on heap?

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

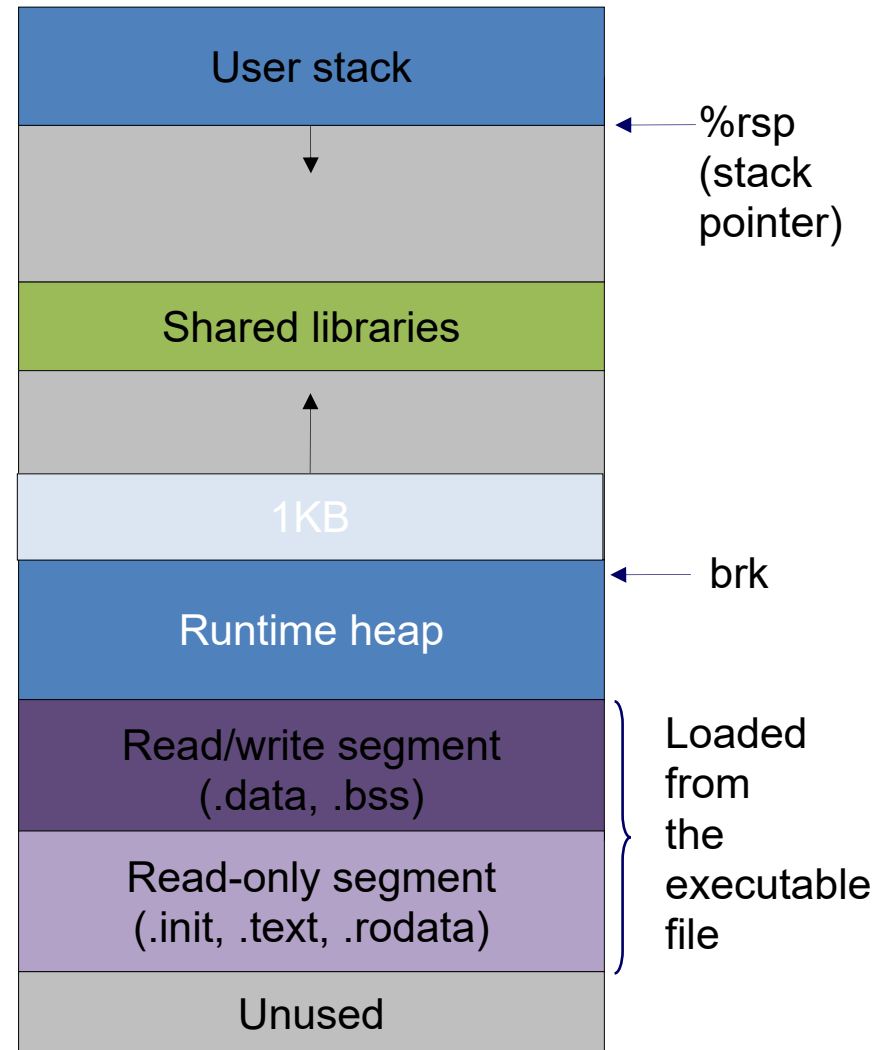
Question: How to allocate memory on heap?

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?

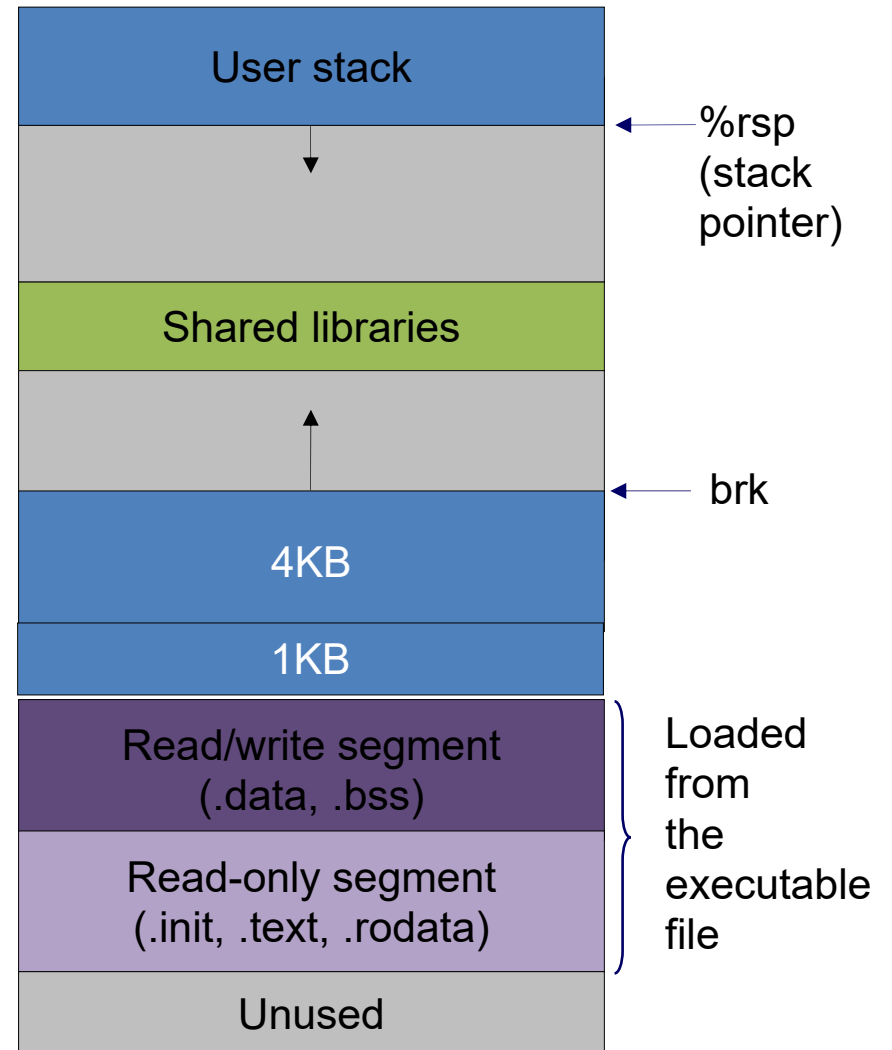
Ask OS for allocation on the heap via **system calls**

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

Issue I – 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

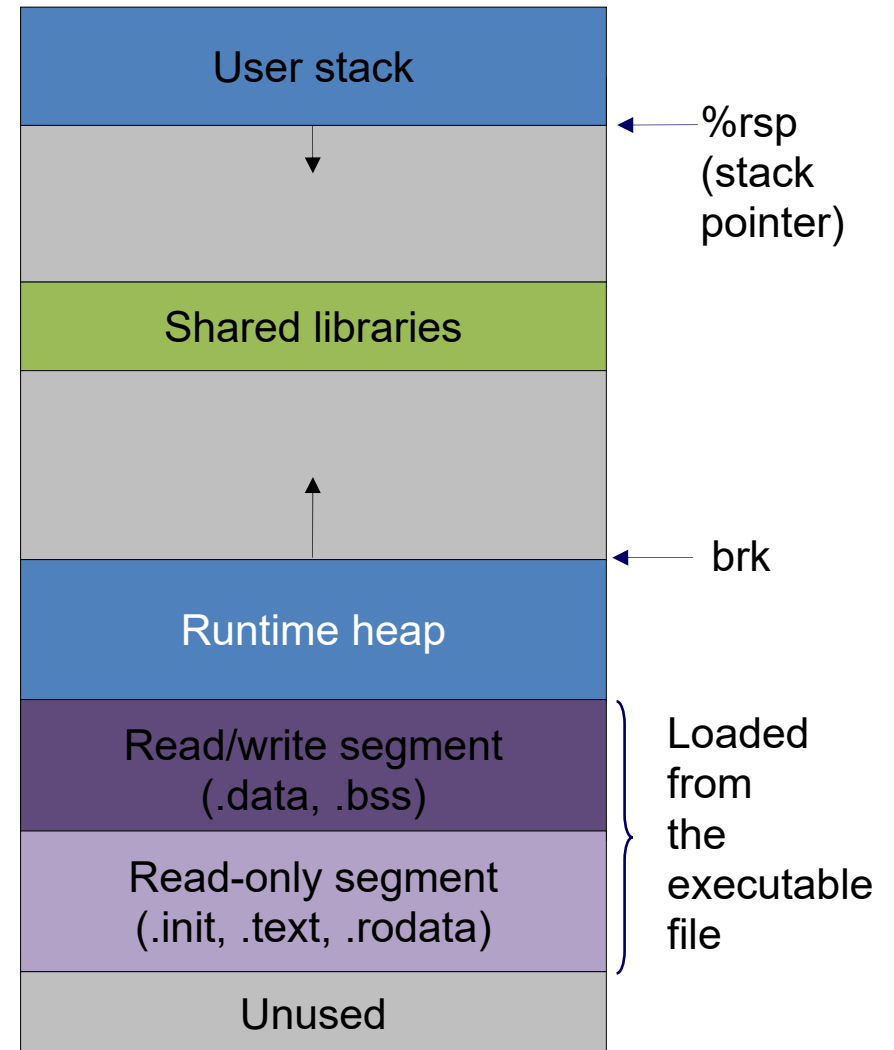
Question: How to allocate memory on heap?

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

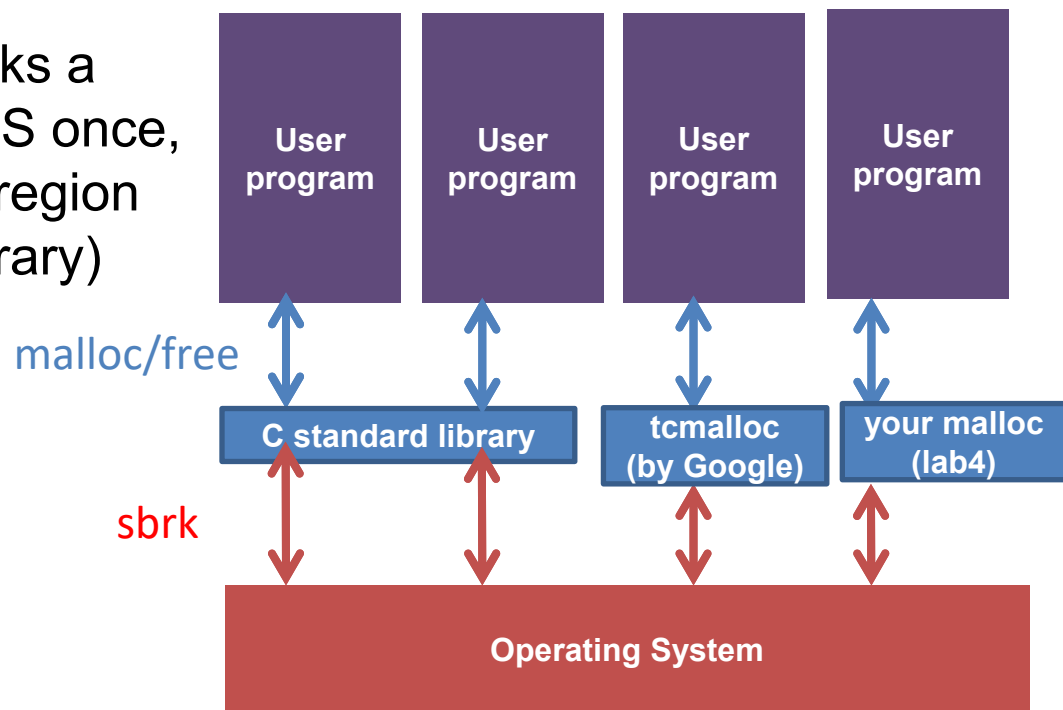
Issue II – system call has high performance cost > 10X



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?

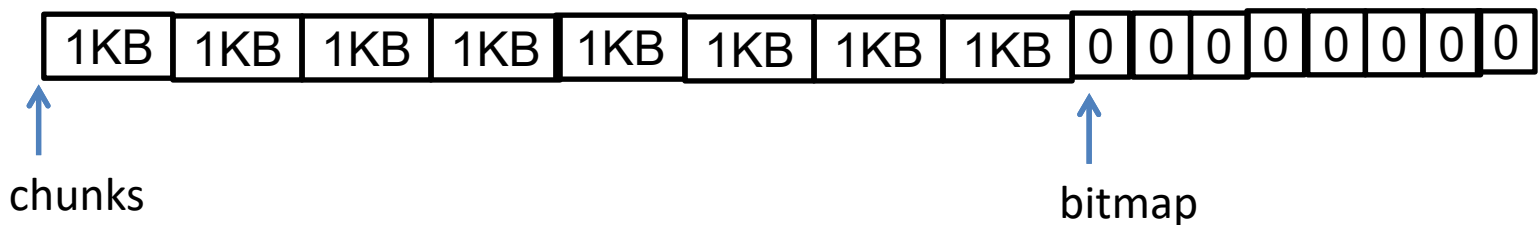
- Assumptions on application behavior:
 - Use APIs correctly
 - Argument of free must be the return value of a previous malloc
 - No double free
 - Use APIs freely
 - Can issue an arbitrary sequence of malloc/free
- Restrictions on the allocator:
 - Once allocated, space cannot be moved around

Questions

- (Basic book-keeping) How to keep track which bytes are free and which are not?
- (Allocation decision) Which free chunk to allocate?
- (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

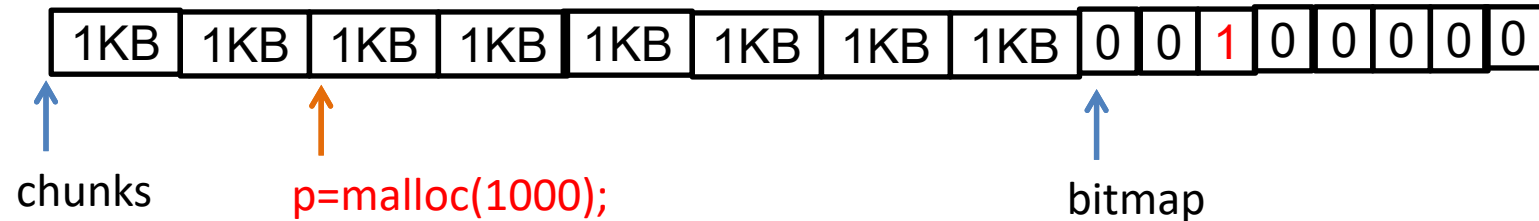


```
#define CHUNKSIZE 1<<10;
typedef char[CHUNKSIZE] chunk;
char *bitmap;
chunk *chunks;
size_t n_chunks;

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

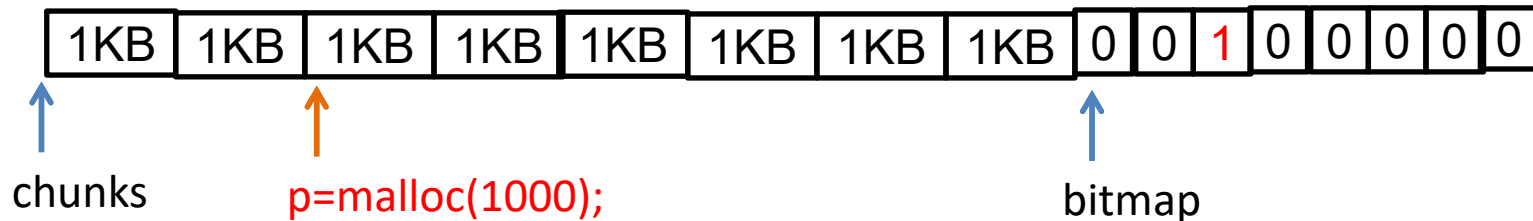
Assume allocator asks for enough memory from OS in the beginning

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.

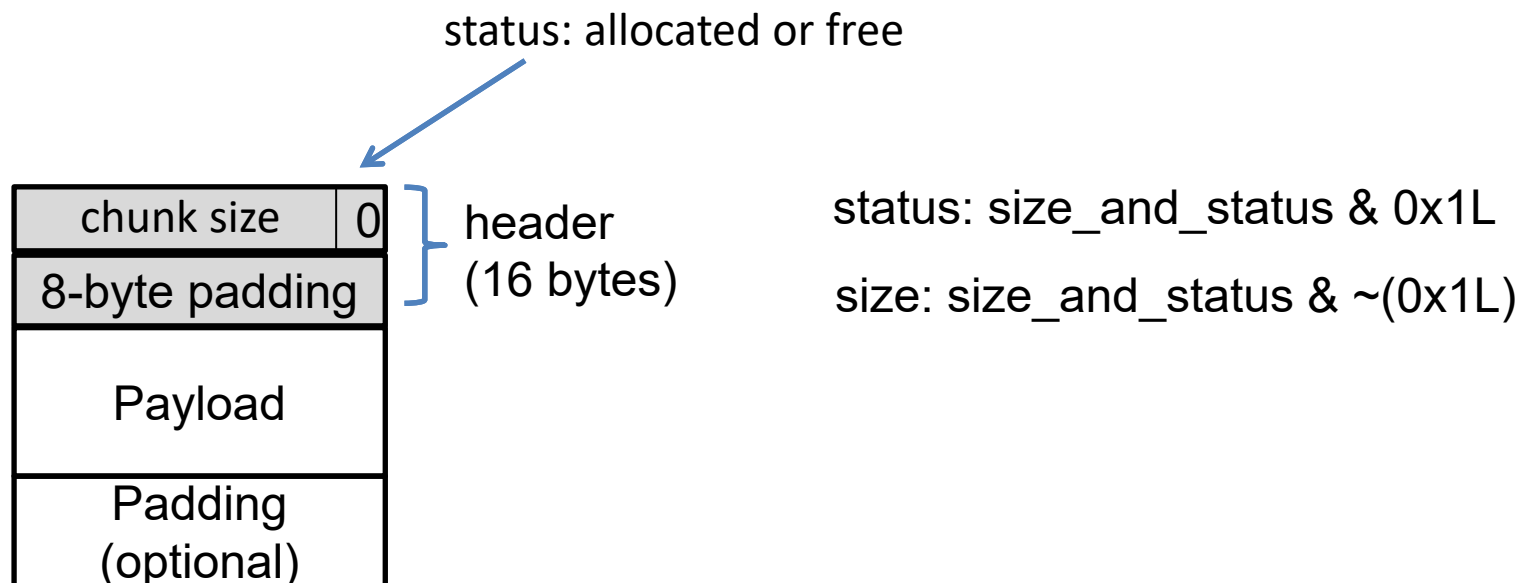
Problems of strawmans?

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

How to implement a “linked list” without use of malloc

Implicit list

- Embed chunk metadata in the chunks
 - Chunk has a header storing size and status
 - 16-byte aligned
 - Payload starting address must be some multiple of 16
 - To simplify design, assume header size is 16 byte, payload size is $x \times 16$

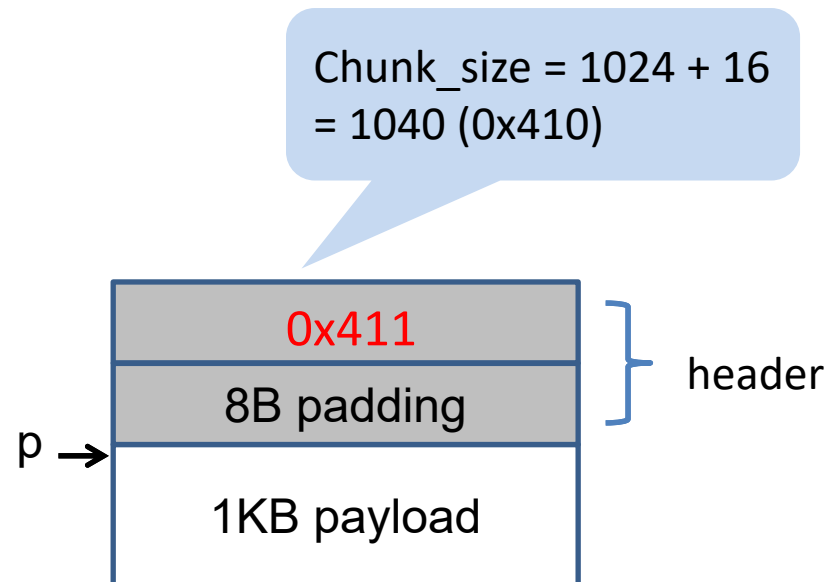


Implicit list

Embed chunk metadata in the chunks

- Chunk has a header storing size and status
- Payload is 16-byte aligned

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

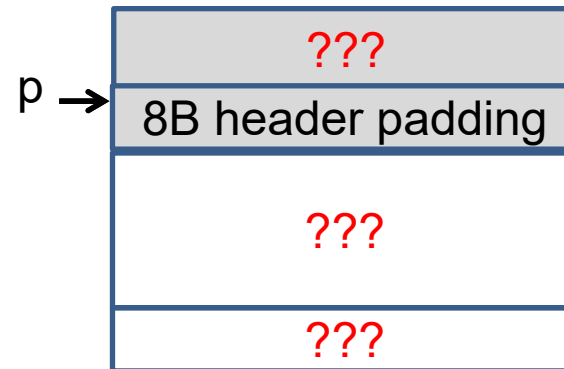


Implicit list

Embed chunk metadata in the chunks

- Chunk has a header storing size and status
- Payload is 16-byte aligned

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

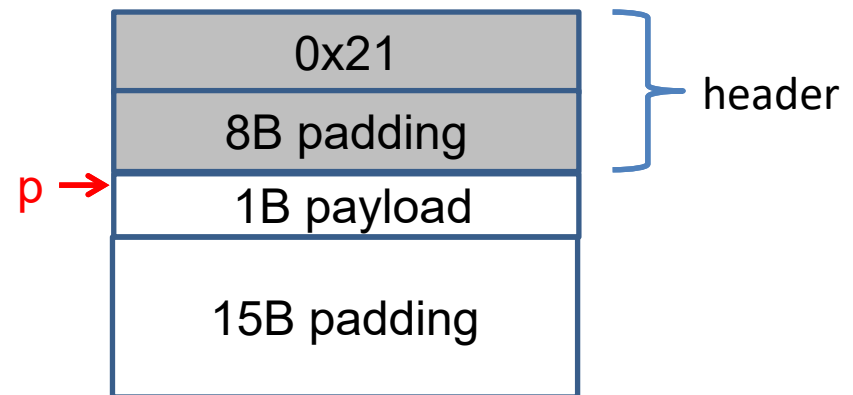


Implicit list

Embed chunk metadata in the chunks

- Chunk has a header storing size and status
- Payload is 16-byte aligned

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

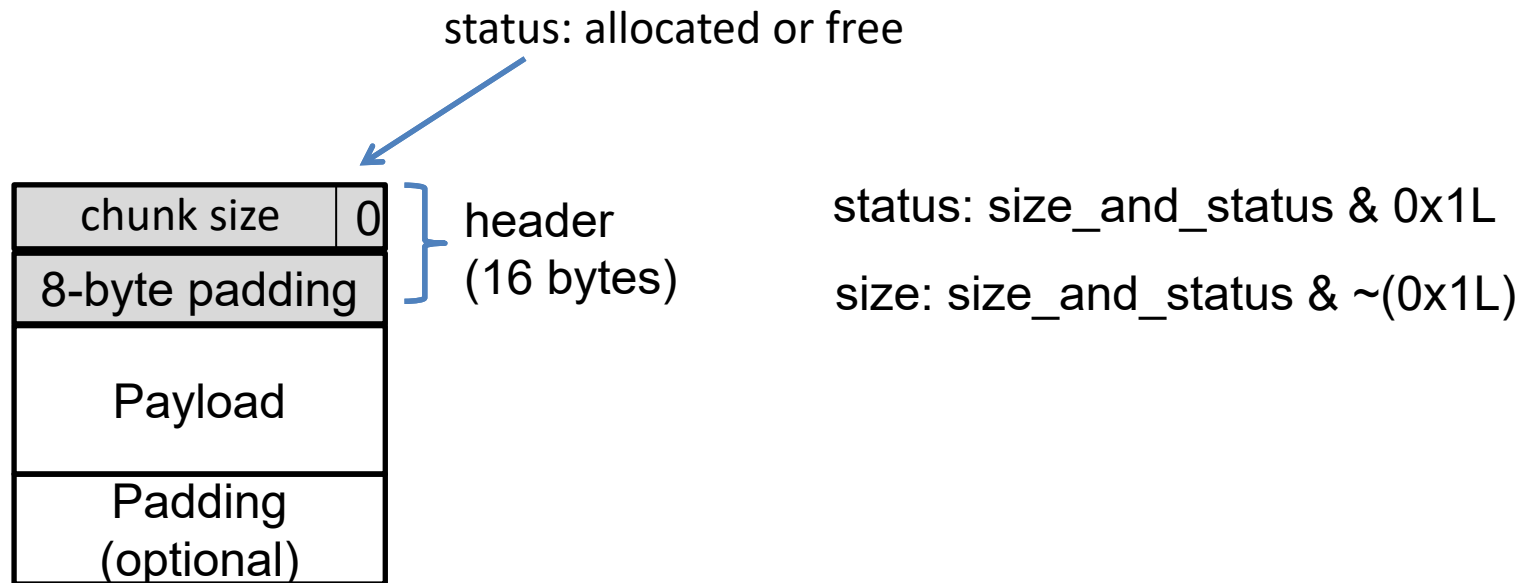


Today's lesson plan

- Previously:
 - Why dynamic memory allocation?
 - Design requirements and challenges
 - The basics of implicit list design.
- Today:
 - Implicit list
 - Explicit list

Implicit list

- Embed chunk metadata in the chunks
 - Chunk has a header storing size and status
 - 16-byte aligned
 - Payload starting address must be some multiple of 16
 - To simplify design, assume header size is 16 byte, payload size is $x \times 16$



How to initialize an implicit list

```
typedef struct {
    unsigned long size_and_status;
    unsigned long padding;
} header;

void init_chunk(header *p, unsigned long sz, bool status)
{
    p->size_and_status = sz | (unsigned long) status;
}

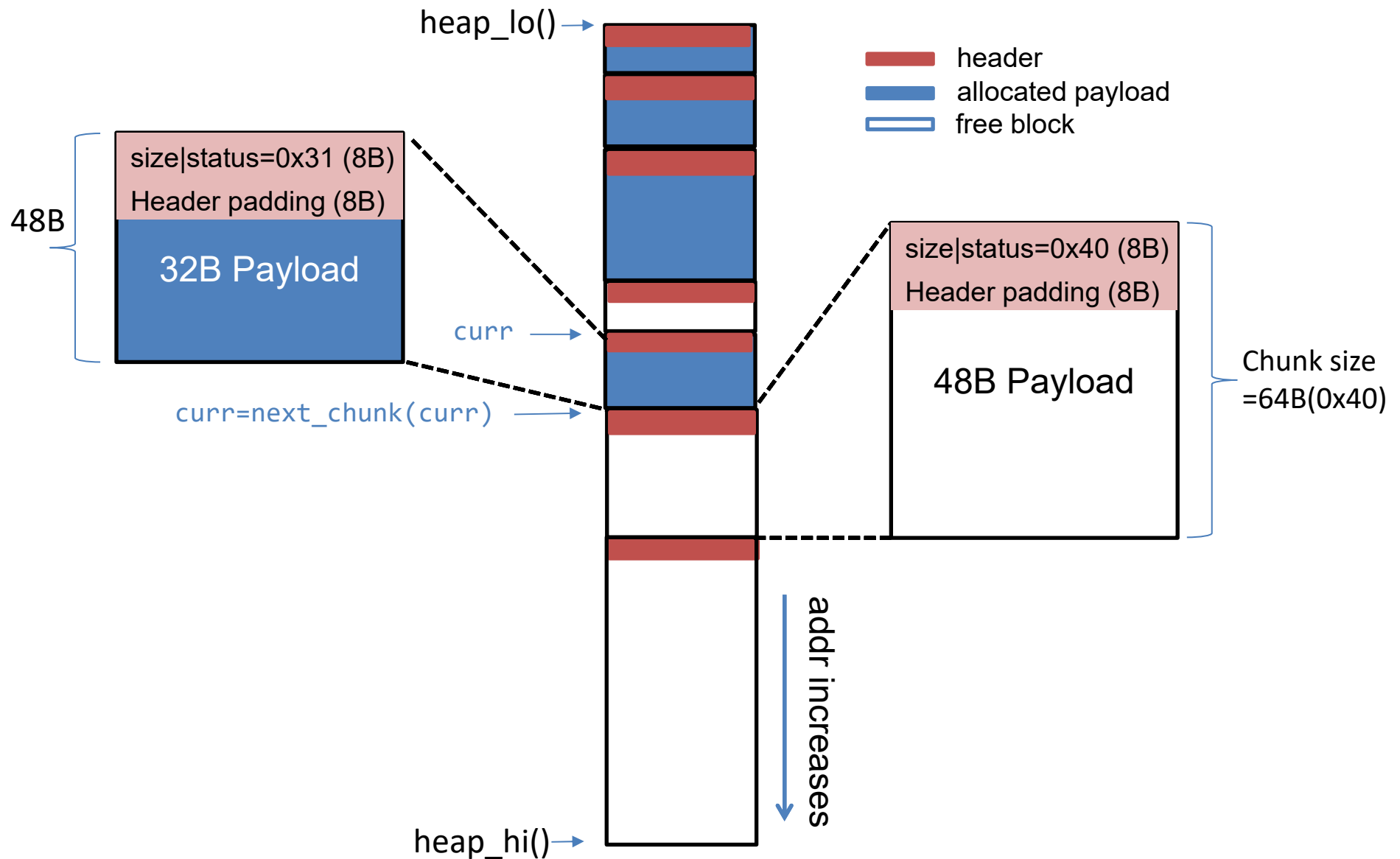
void init()
{
    header *p;
    p = ask_os_for_chunk(INITIAL_CSZ);
    init_chunk(p, INITIAL_CSZ, status);
}
```

How to traverse an implicit list

```
bool get_status(header *h) {  
    // return status of the chunk  
}  
size_t get_size(header *h) {  
    // return size of the chunk  
}  
  
header *next_chunk(header *curr) {  
    // How to set curr to point to next chunk?  
}  
  
void traverse_implicit_list() {  
    header *curr = (header *)heap_lo();  
    while ((char *)curr < heap_high()) {  
        bool allocated = get_status(curr);  
        size_t csz = get_chunksz(curr);  
        printf("chunk size=%d status=%d\n", csz, allocated);  
        curr = next_chunk(curr);  
    }  
}
```

```
typedef struct {  
    unsigned long size_and_status;  
    unsigned long padding;  
} header;
```

How to traverse an implicit list

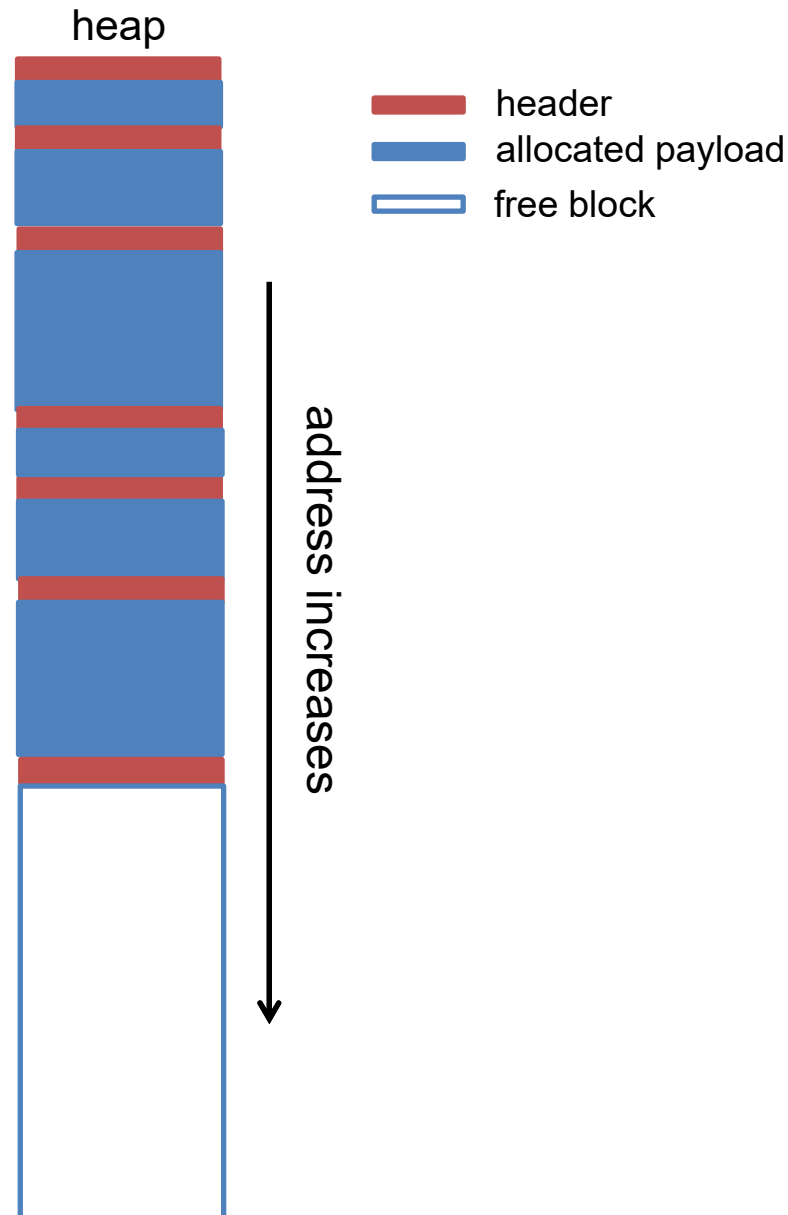


malloc() in an implicit list

```
void malloc(unsigned long size) {  
    unsigned long chunk_sz = align(size) + sizeof(header);  
    header *h = find_fit(chunk_sz);  
    //split if chunk is larger than necessary  
    split(h, chunk_sz);  
    set_status(h, true);  
}
```

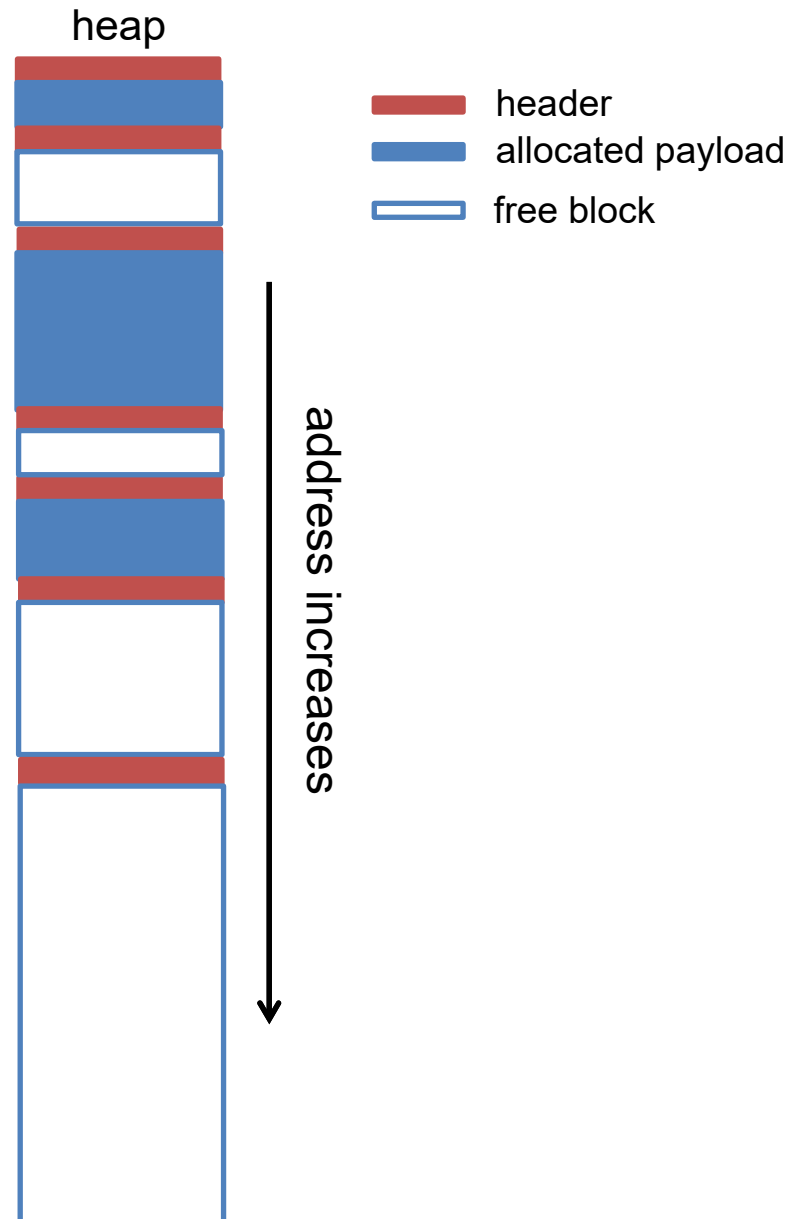
Where to place an allocation?

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



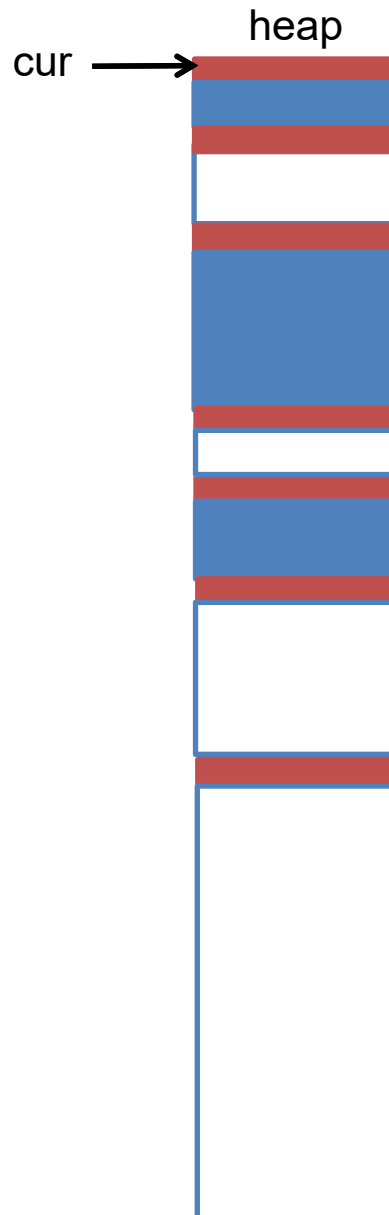
Where to place an allocation?

```
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)
```



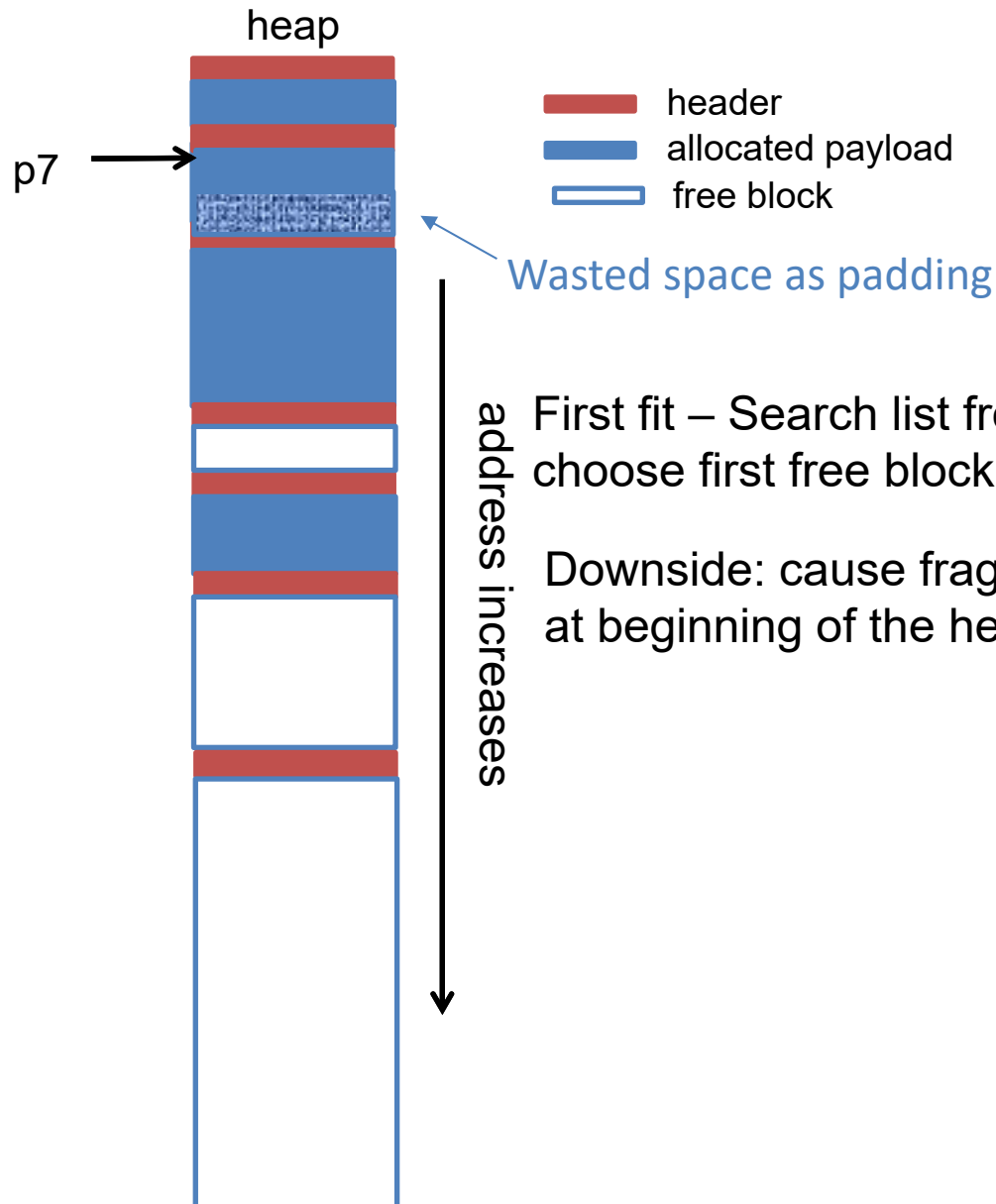
header
allocated payload
free block

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

address increases

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)
```

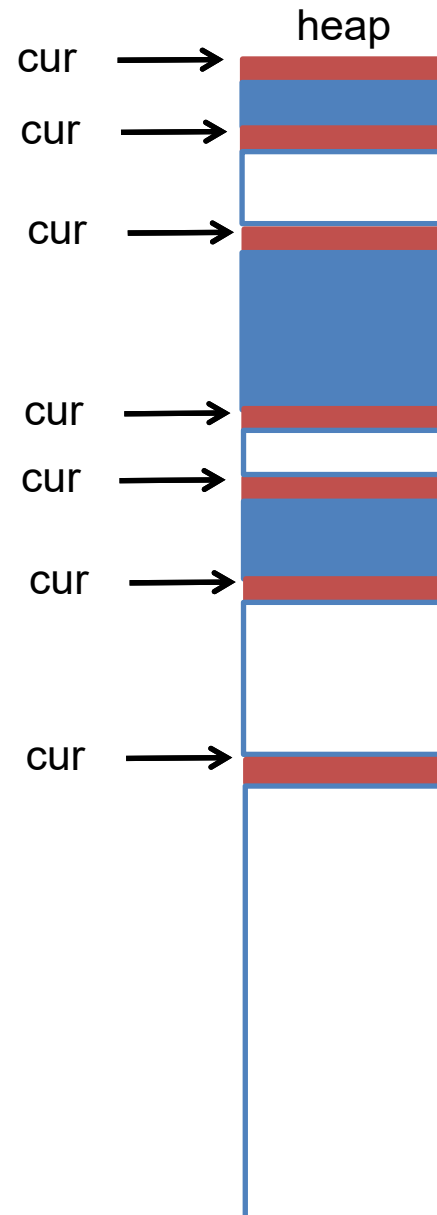


First fit – Search list from beginning, choose first free block that fits

Downside: cause fragmentation at beginning of the heap

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)
```



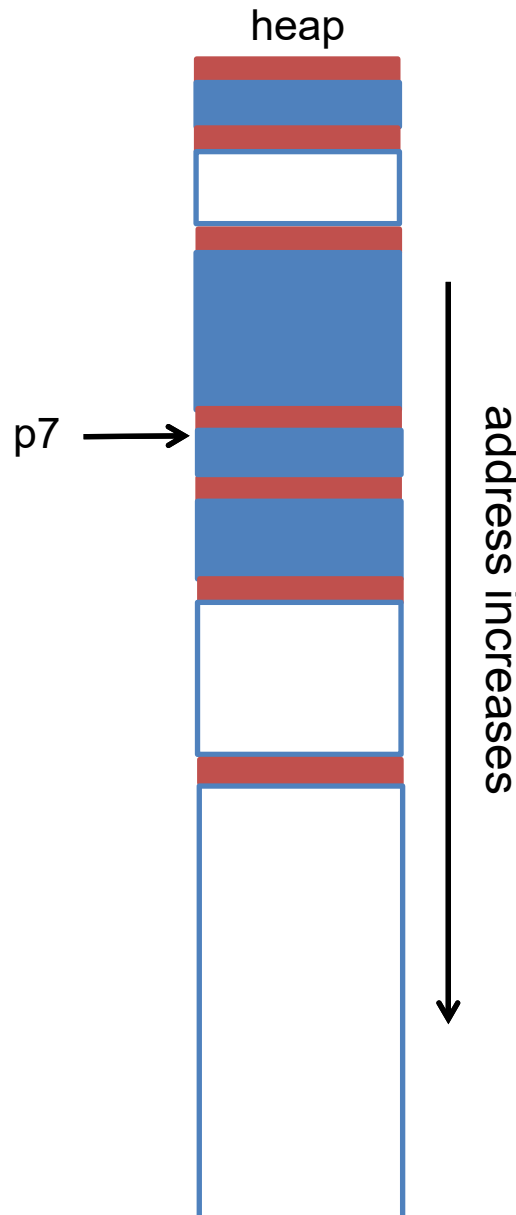
header
allocated payload
free block

address increases

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

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)
```



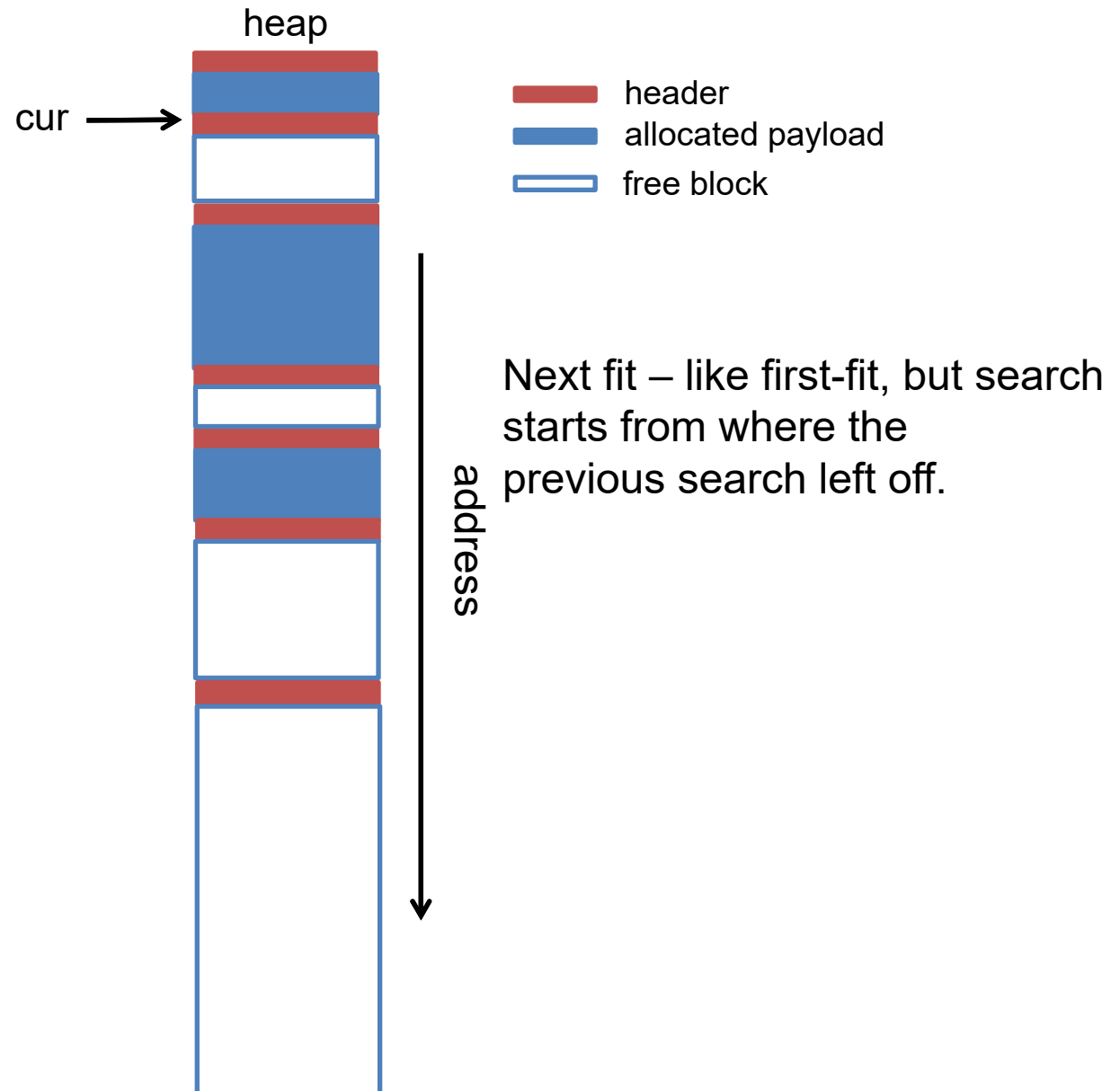
header
allocated payload
free block

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

Downside: run slower than first fit.

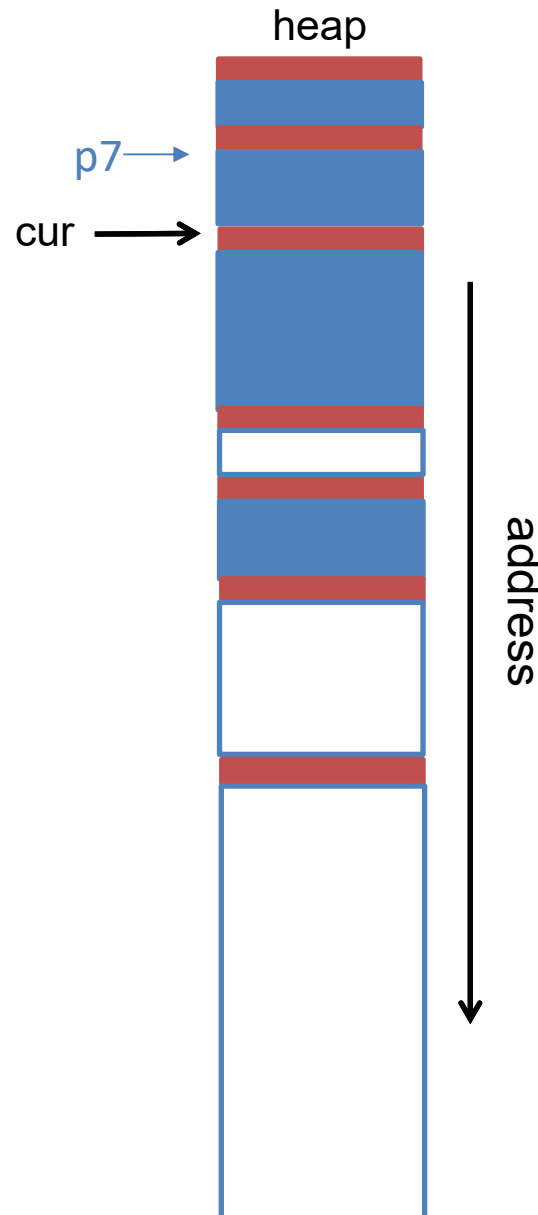
Next fit

```
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)
```



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)
```

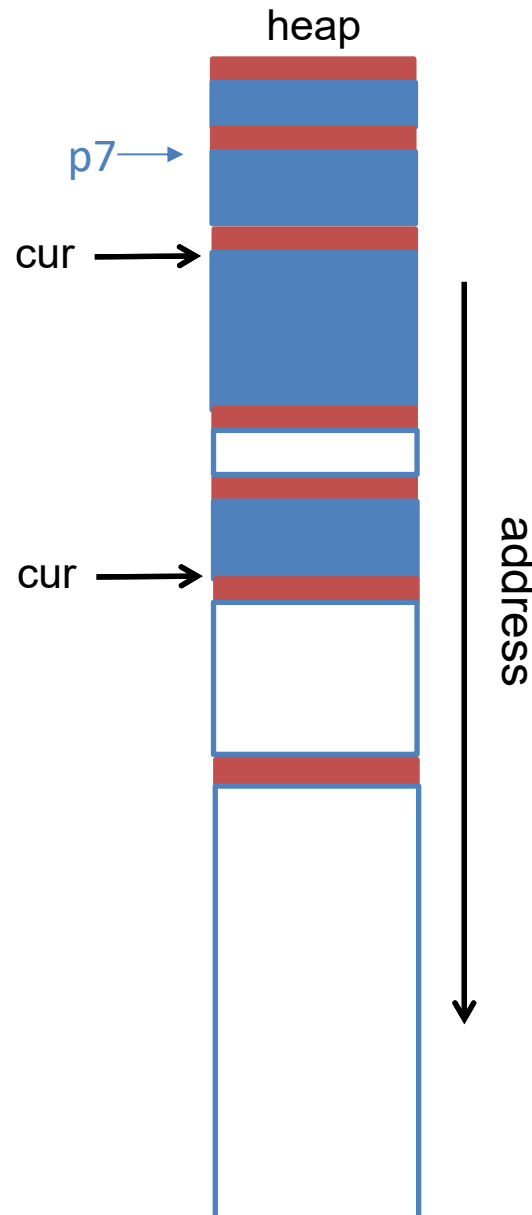


header
allocated payload
free block

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)
```

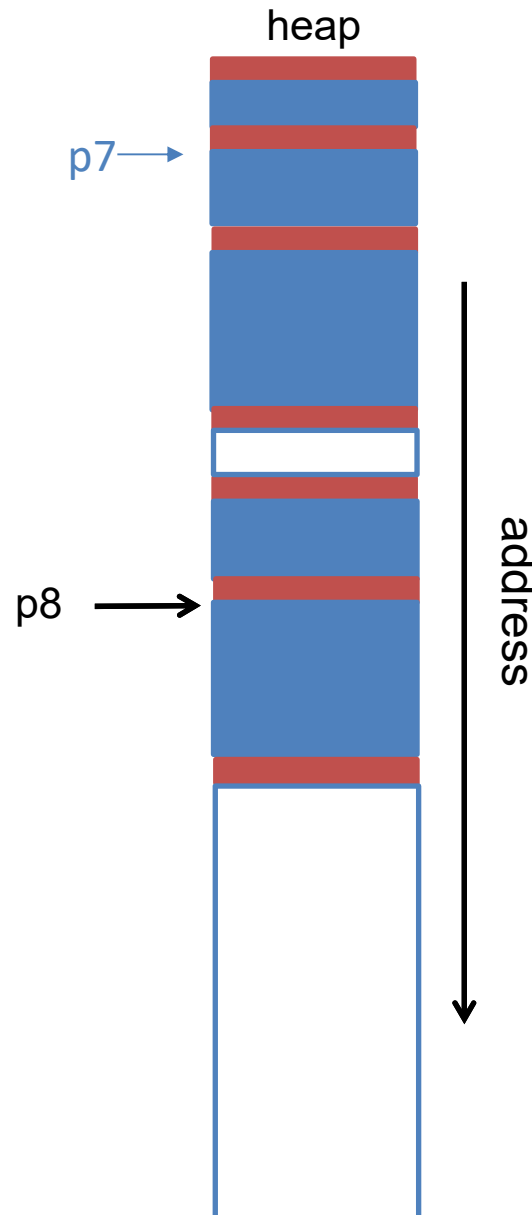


header
allocated payload
free block

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
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.

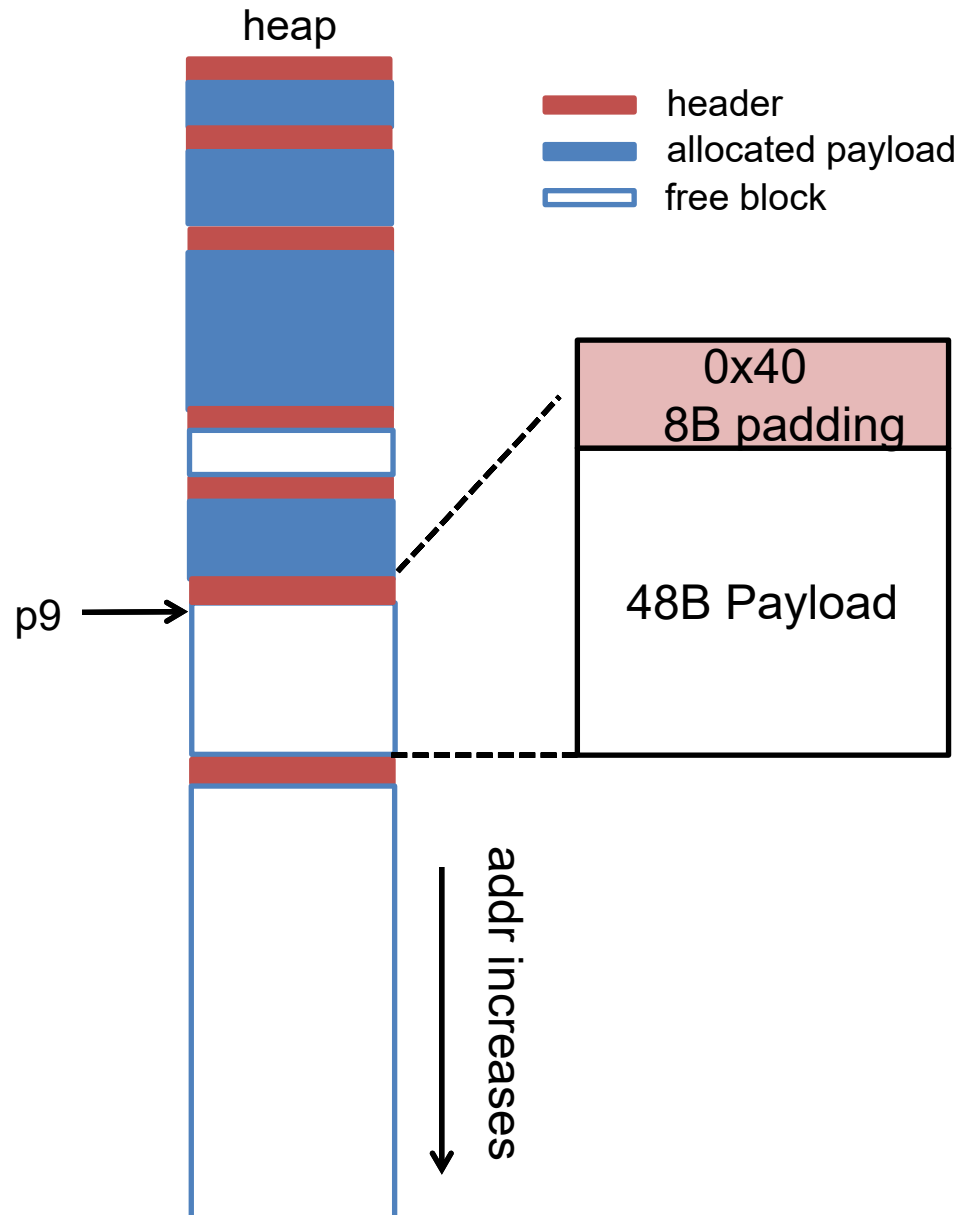
malloc() in an implicit list

```
void malloc(unsigned long size) {  
    unsigned long chunk_sz = align(size) + sizeof(header);  
    header *h = find_fit(chunk_sz);  
    //split if chunk is larger than necessary  
    split(h, chunk_sz);  
    set_status(h, true);  
}
```

Splitting a free block

...

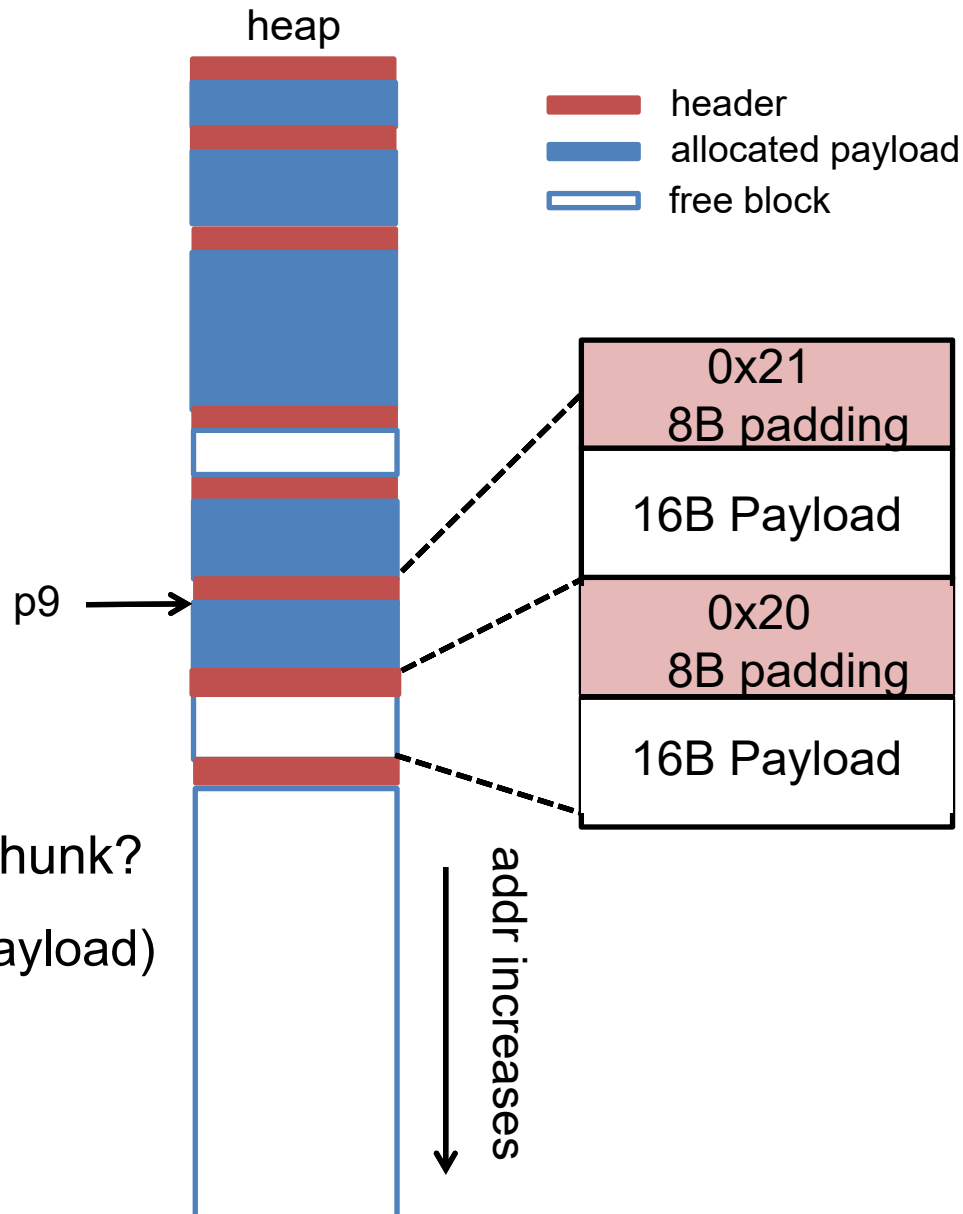
p9 = malloc(16)



Splitting a free block

...

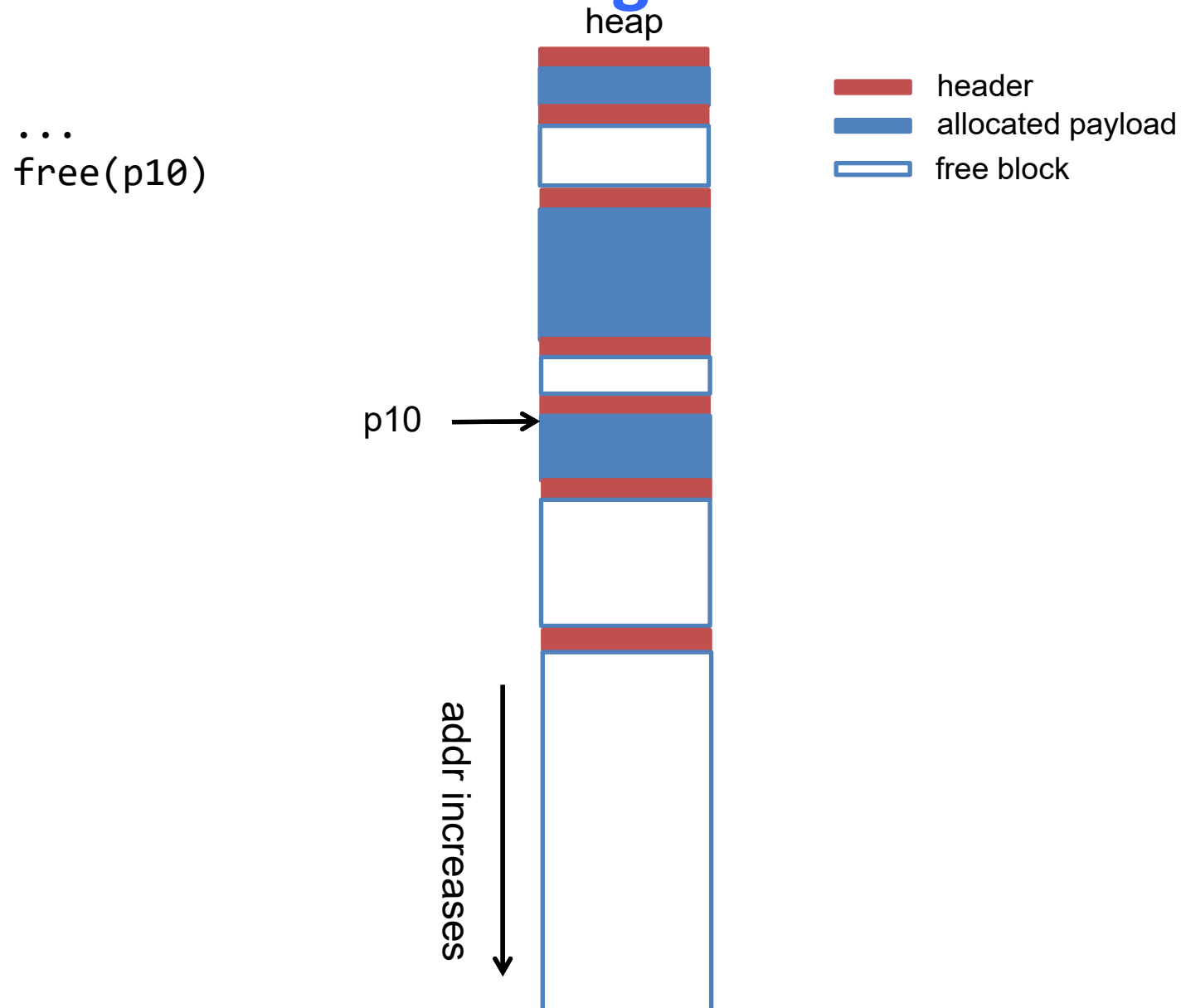
p9 = malloc(16)



Q: what's the smallest chunk?

A: 16 (header)+16(min payload)
= 32B

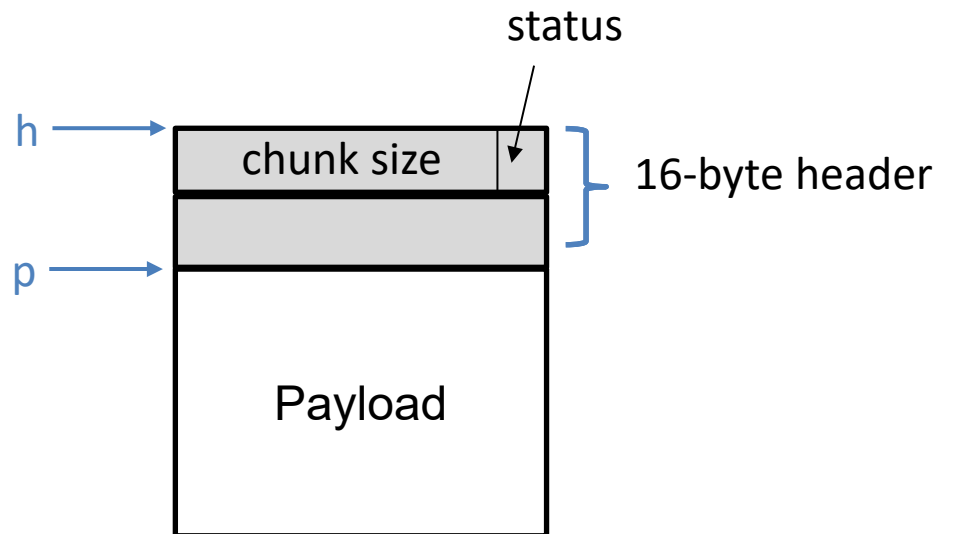
Coalescing a free block with its next free neighbor



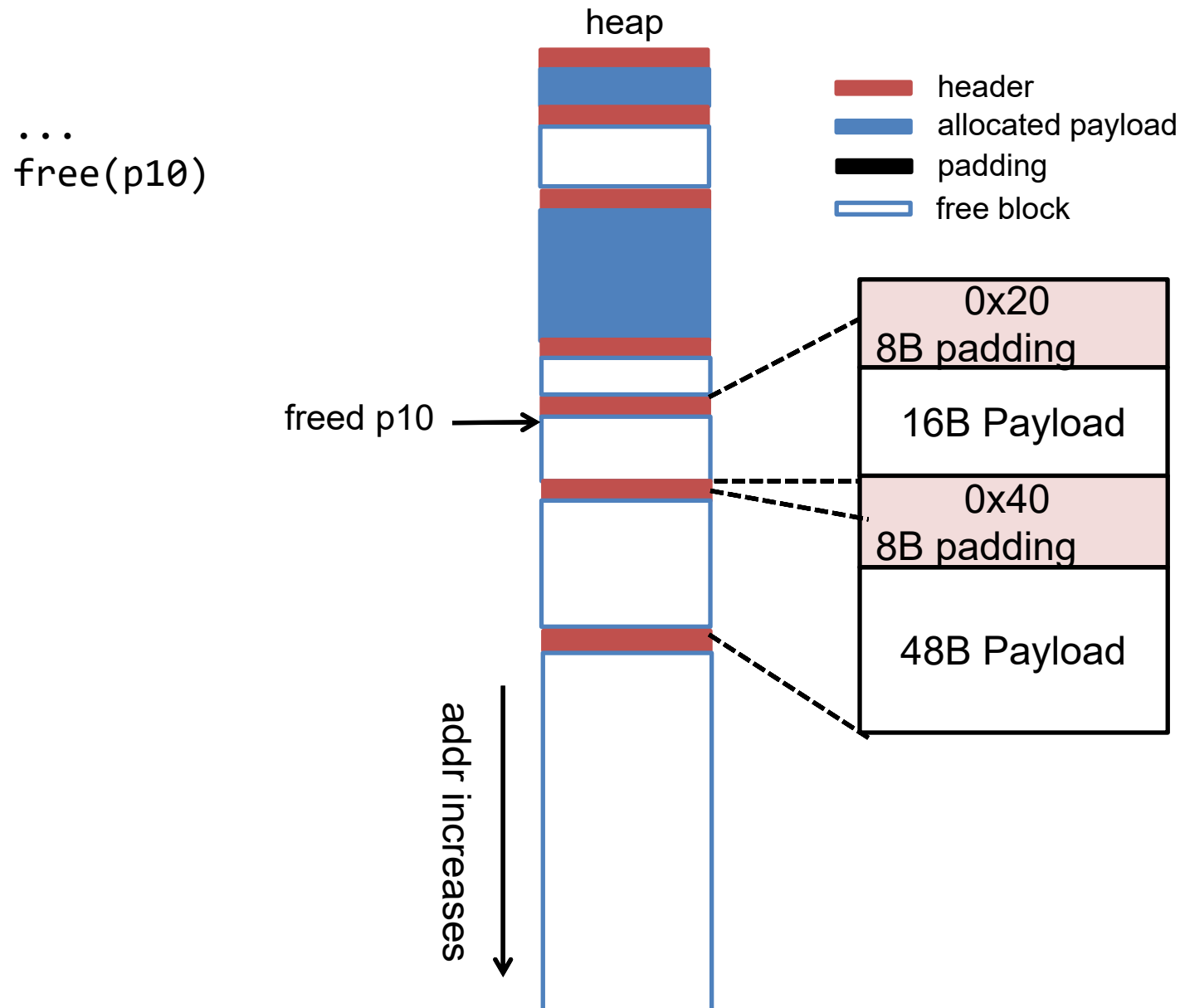
free() in an implicit list

```
void free(void *p) {  
    header *h = payload2header(p);  
    set_status(h, false);  
    coalesce(h);  
}
```

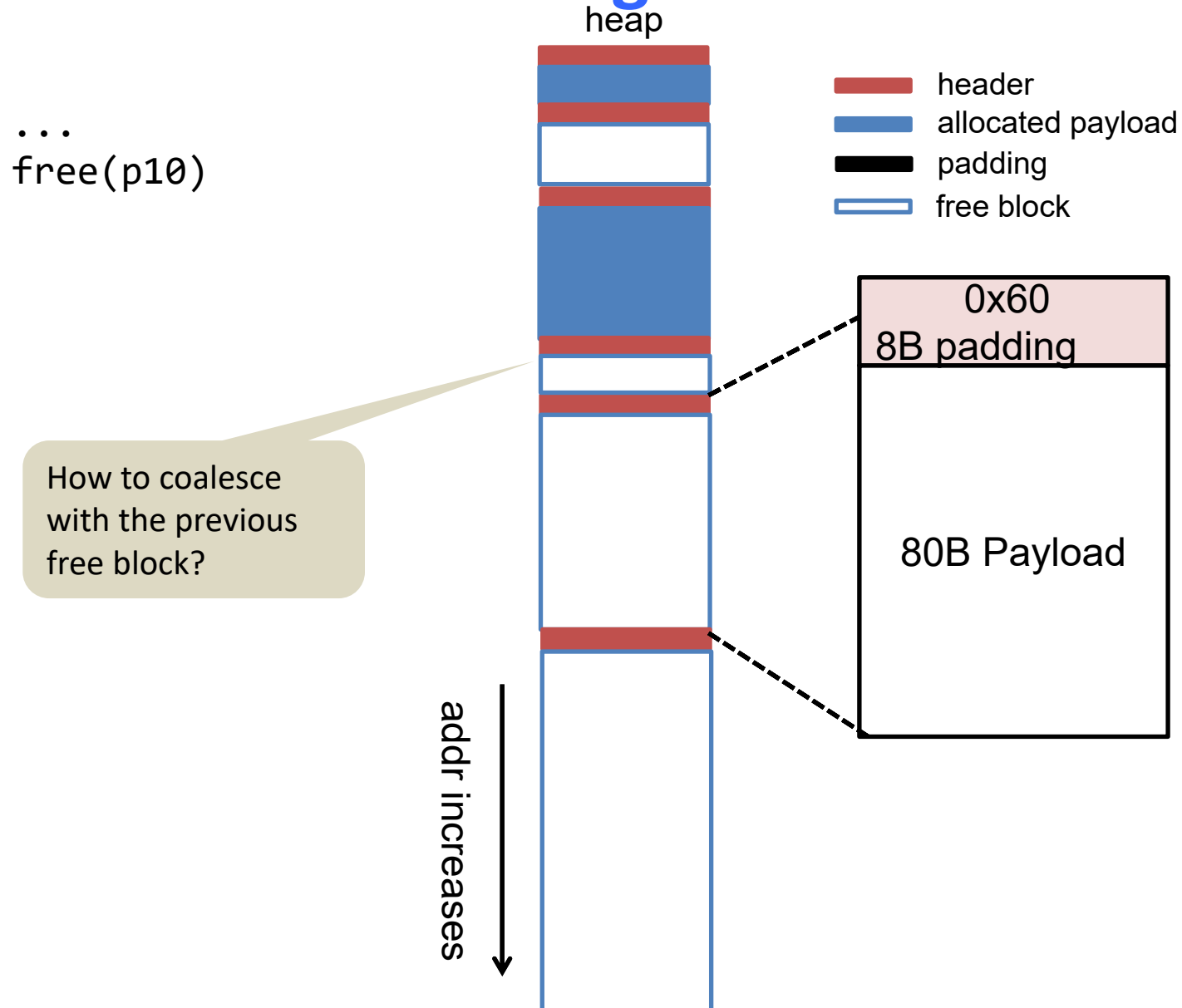
```
header *payload2header(void *p)  
{  
  
}
```



Coalescing a free block with next free neighbor

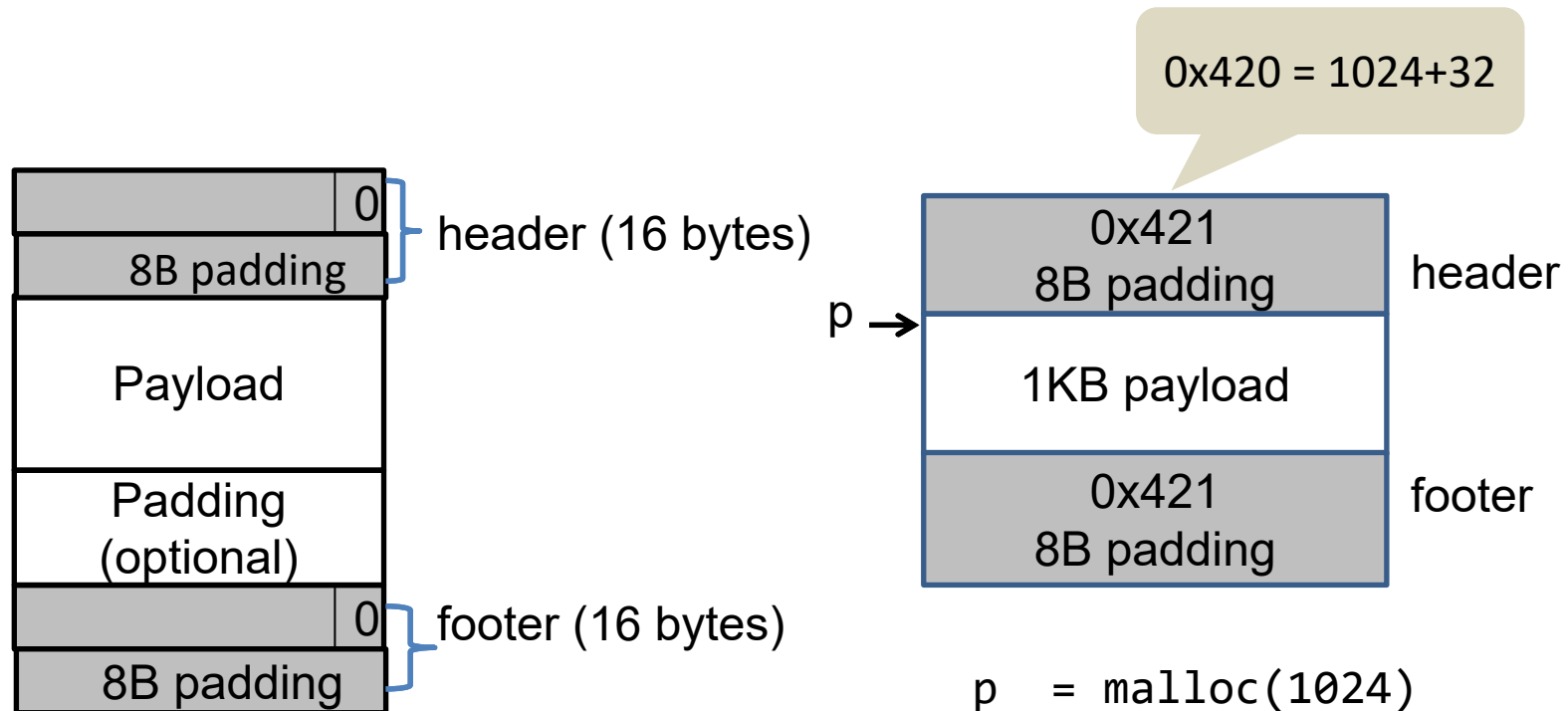


Coalescing a free block with its next free neighbor

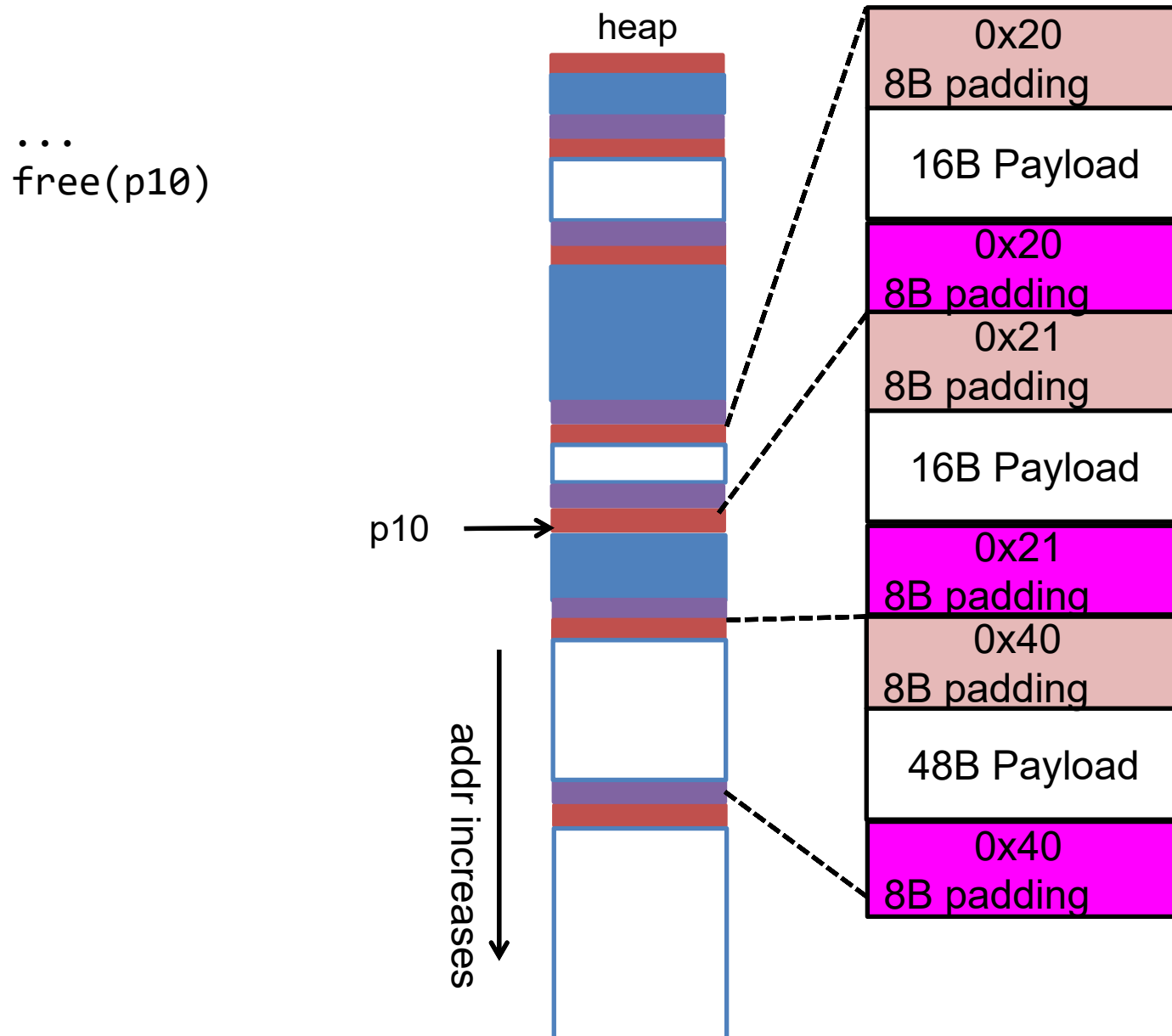


Use footer to coalesce with previous block

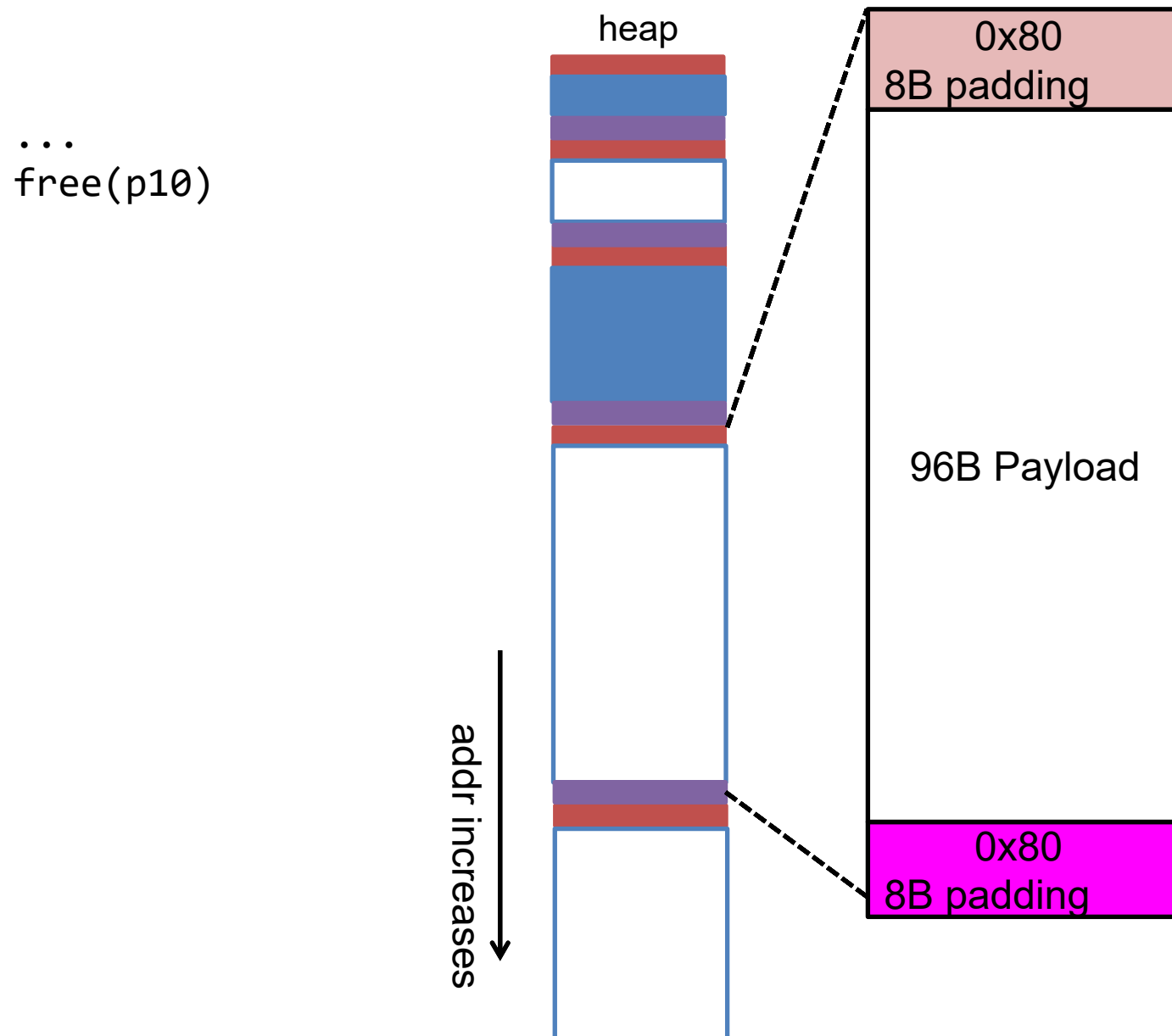
- Duplicate header information into the footer



Coalescing prev and next blocks



Coalescing prev and next blocks



Explicit free lists

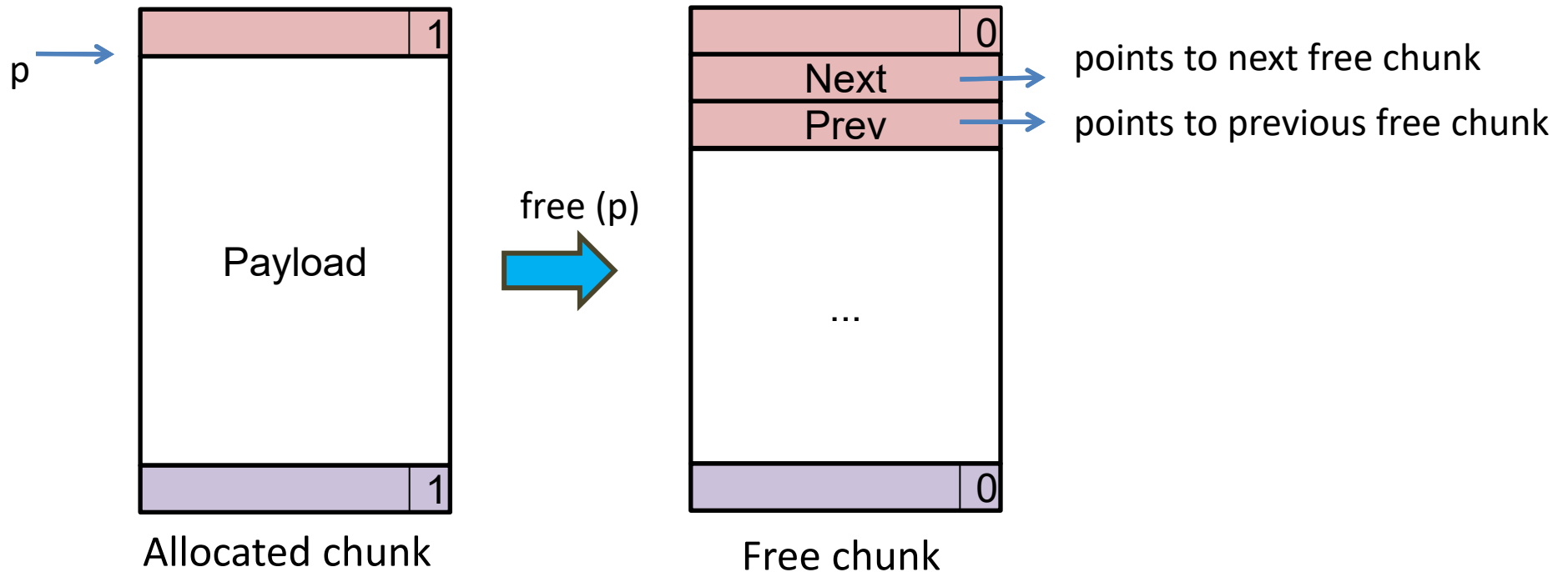
Problems of implicit list:

- Allocation time is linear in # of total (free and allocated) chunks

Explicit free list:

- Maintain a linked list of free chunks only.

Explicit free list



- Question: do we need next/prev fields for allocated blocks?

Answer: No. We do not need to chain together allocated blocks. We can still traverse all blocks (free and allocated) as in the case of implicit list.

- Question: what's the minimal size of a chunk?

Answer: 16 (header) + 16 (footer) + 8 (next pointer) + 8 (previous pointer) = 48 bytes

Explicit list: types, basic helpers

```
typedef struct {
    unsigned long size_and_status;
    unsigned long padding;
} header;

typedef struct free_hdr {
    header common_header;
    struct free_hdr *next;
    struct free_hdr *prev;
} free_hdr;
```

```
bool get_status(header *h) {
    ...
}

size_t get_size(header *h) {
    ...
}

void set_status(header *h, bool status){
    ...
}


Void set_size(header *h, size_t sz) {
    ...
}
```

Explicit list: globals, initialization

```
free_hdr *freelist;
```

```
#define MIN_OS_ALLOC_SZ 1024  
void init() {  
    free_hdr *h = get_block_from_OS(MIN_OS_ALLOC_SZ);  
    init_free_chunk(h, MIN_OS_ALLOC_SZ);  
    insert(&freelist, h);  
}
```

Explicit list: allocate

```
void *
malloc(size_t s) {  assume s >= 16 and is 16-byte aligned
    size_t csz = s + 2*sizeof(header); //min chunk size required
    free_hdr *n = first_fit(csz);
    //if n is NULL, then ask OS to increase heap size
    free_hdr *newchunk = split(n, csz);
    if (newchunk)
        insert(&freelist, newchunk);
    set_status(n, true);
    return (char *)n+sizeof(header);
}
```

```
free_hdr *
first_fit(size_t sz) {
    free_hdr *n = freelist;
    while (n) {
        if (get_size(&(n->common_header)) >= sz) {
            delete(&freelist, n);
            break;
        }
        n = n->next;
    }
    return n;
}
```


Explicit list: free

```
void free(void *p) {  
    header *h = get_header_from_payload(p);  
    init_free_chunk((free_hdr *)h, get_size(h));  
  
    header *next = get_next_header(h);  
    if (!get_status(next))  
        h = coalesce((free_hdr *)h, (free_hdr *)next);  
    header *prev = get_prev_header(h);  
    if (!get_status(prev))  
        h = coalesce((free_hdr *)h, (free_hdr *)prev);  
  
    insert(&freelist, (free_hdr *)h);  
}
```

Today's lesson plan

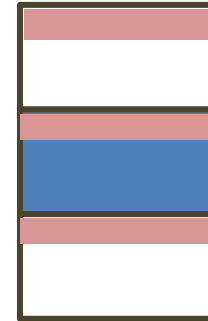
- Quick review of implicit, explicit list design
- Segregated list
- Buddy system

Review: implicit → explicit

Implicit list (header-only)



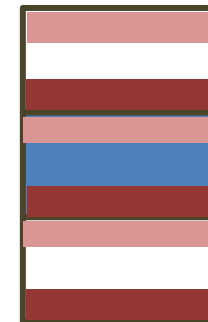
Problem: cannot coalesce with previous free block
→ contiguous free blocks



Implicit list (header+footer)

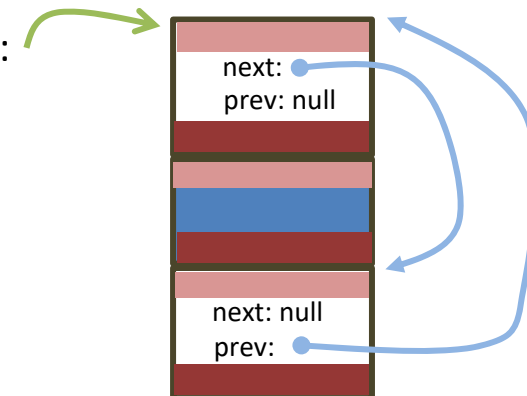


Problem: search for free block scans over allocated blocks



explicit list (header+footer)

head of freelist:

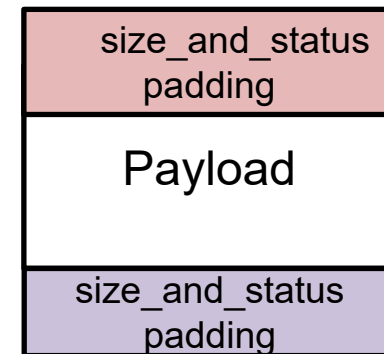


Q: Why use a doubly linked list?

Explicit list: implementation

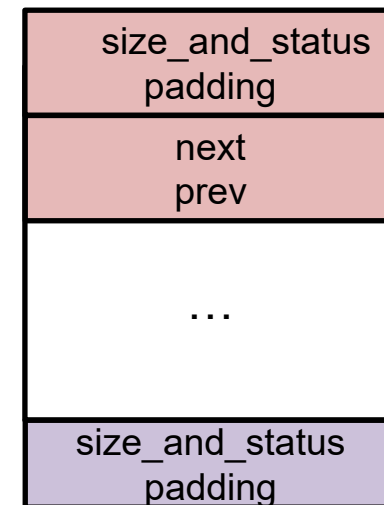
```
typedef struct {  
    unsigned long size_and_status;  
    unsigned long padding;  
} header;
```

Allocated chunk:



```
typedef struct free_hdr {  
    header common_header;  
    struct free_hdr *next;  
    struct free_hdr *prev;  
} free_hdr;
```

Free chunk:



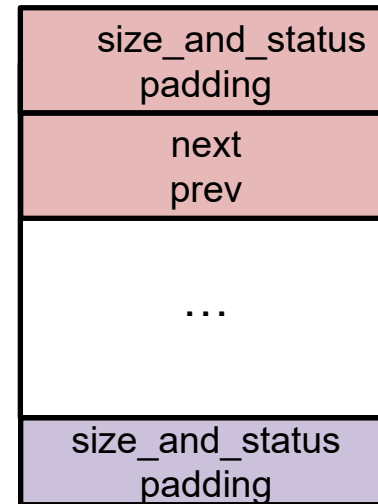
Explicit list: initialize a free chunk

```
typedef struct free_hdr {
    header common_header;
    struct free_hdr *next;
    struct free_hdr *prev;
} free_hdr;
```

```
free_hdr *freelist;
```

```
//initialize a region of memory of size 'sz' //with
start address 'h' as a free chunk
void init_free_chunk(free_hdr *h, size_t sz)
{

```



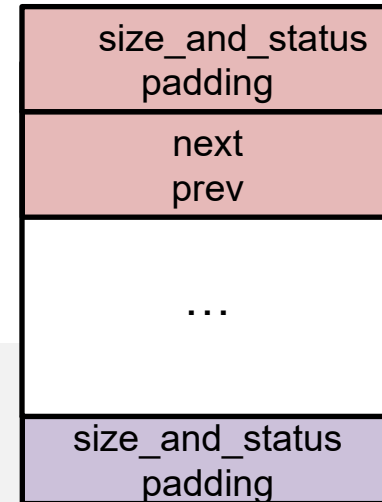
Explicit list: initialization

```
typedef struct free_hdr {  
    header common_header;  
    struct free_hdr *next;  
    struct free_hdr *prev;  
} free_hdr;
```

```
free_hdr *freelist = NULL;
```

```
//initialize a region of memory of size 'sz'  
//with start address 'h' as a free chunk  
void init_free_chunk(free_hdr *h, size_t sz)  
{  
  
    set_size_status(&h->common_header, sz, false);  
    h->prev = h->next = NULL;  
    set_size_status(get_footer_from_header(h->common_header), sz, false);  
}
```

```
free_hdr *get_block_from_OS(size_t sz) {  
    free_hdr *h = sbrk(sz);  
    init_free_chunk(h, sz); //init header and footer  
    return h;  
}  
  
void init() {  
    free_hdr *h = get_block_from_OS(INIT_ALLOC_SZ);  
    insert(&freelist, h);  
}
```



Explicit list: allocate

assume $s \geq 16$ and is 16-byte aligned

```
void *malloc(size_t s){  
    size_t csz = s + 2*sizeof(header); //min chunk size required  
    free_hdr *n = first_fit(csz); //I ignored n=NULL (not enough space)  
    free_hdr *newchunk = split(n, csz);  
    if (newchunk)  
        insert(&freelist, newchunk);  
    set_status(n, true);  
    return (char *)n+sizeof(header);  
}
```

```
free_hdr *first_fit(size_t sz) {
```

```
}
```

Explicit list: allocate

assume $s \geq 16$ and is 16-byte aligned

```
void *malloc(size_t s){
    size_t csz = s + 2*sizeof(header); //min chunk size required
    free_hdr *n = first_fit(csz); //I ignored n=NULL (not enough space)
    free_hdr *newchunk = split(n, csz);
    if (newchunk)
        insert(&freelist, newchunk);
    set_status(n, true);
    return (char *)n+sizeof(header);
}

free_hdr *first_fit(size_t sz) {
    free_hdr *n = freelist;
    while (n) {
        if (get_size(&n->common_header) >= sz) {
            delete(&freelist, n);
            break;
        }
        n = n->next;
    }
    return n;
}
```


Explicit list: allocate

assume $s \geq 16$ and is 16-byte aligned

```
void *malloc(size_t s){  
    size_t csz = s + 2*sizeof(header); //min chunk size required  
    free_hdr *n = first_fit(csz); //I ignored n=NULL (not enough space)  
    free_hdr *newchunk = split(n, csz);  
    if (newchunk)  
        insert(&freelist, newchunk);  
    set_status(n, true);  
    return (char *)n+sizeof(header);  
}
```

```
free_hdr *split(free_hdr *n, size_t csz)  
{
```

```
}
```

Explicit list: allocate

assume $s \geq 16$ and is 16-byte aligned

```
void *malloc(size_t s){
    size_t csz = s + 2*sizeof(header); //min chunk size required
    free_hdr *n = first_fit(csz); //I ignored n=NULL (not enough space)
    free_hdr *newchunk = split(n, csz);
    if (newchunk)
        insert(&freelist, newchunk);
    set_status(n, true);
    return (char *)n+sizeof(header);
}
```

```
free_hdr *split(free_hdr *n, size_t csz)
{
    size_t remain_sz = get_size(&n->common_header) - csz;
    if (remain_sz < MIN_CHUNK_SZ)
        return NULL;
    set_size(&n->common_header, csz);
    set_size((header *)((char *)n+csz-sizeof(header)), csz);
    free_hdr *newchunk = (free_hdr *)((char *)n+csz);
    init_free_chunk(newchunk, remain_sz);
    return newchunk;
}
```

Segregated list

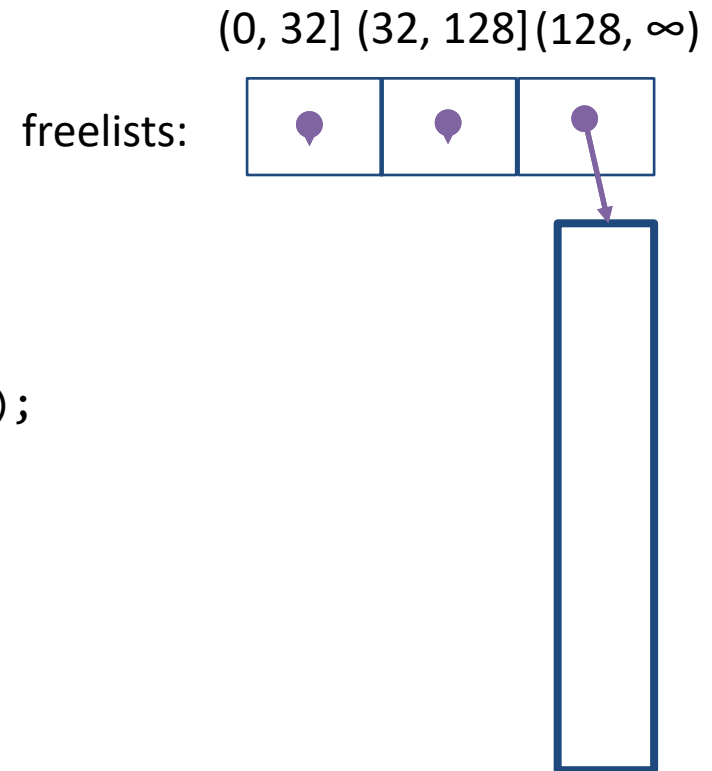
- Idea: keep multiple freelists
 - each freelist contains chunks of similar sizes

Segregated list: initialize

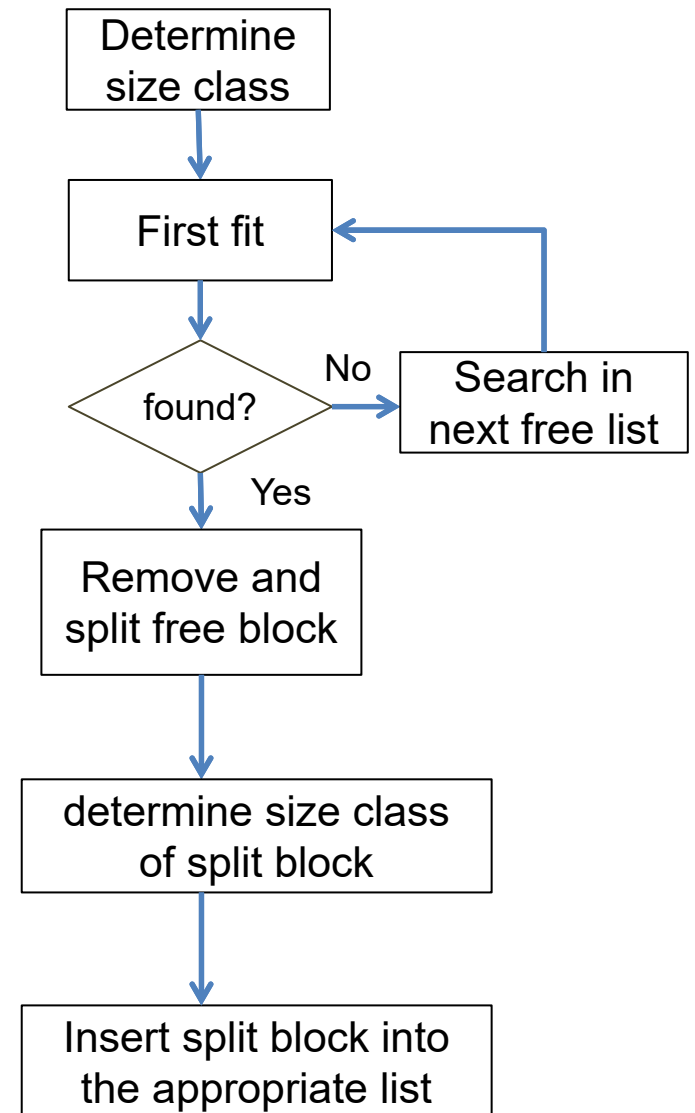
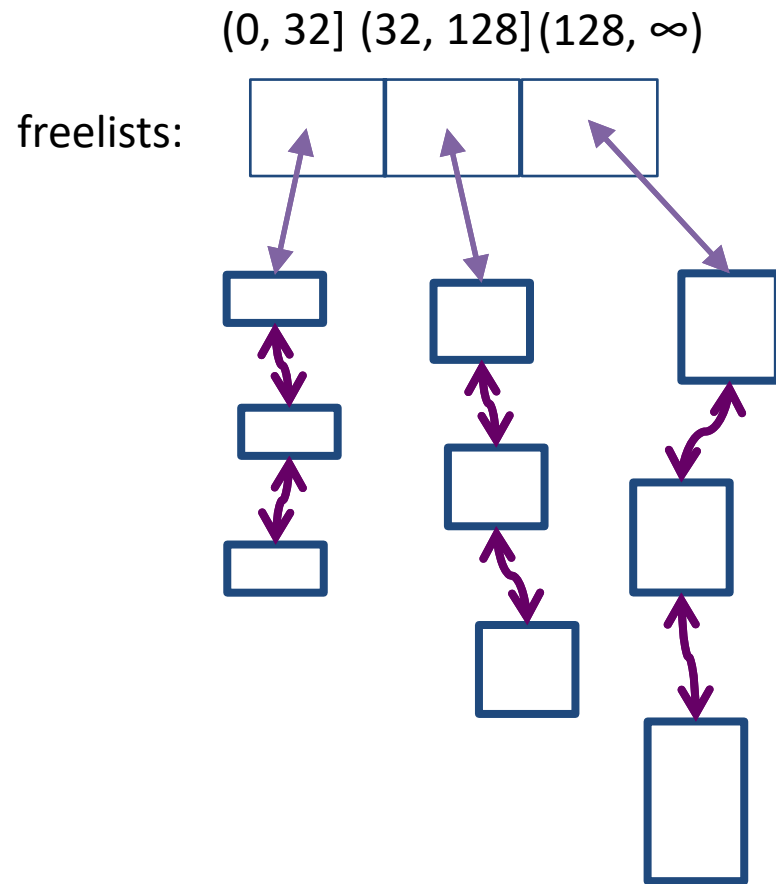
```
#define NLISTS 3
free_hdr* freelists[NLISTS];
size_t size_classes[NLISTS] = {32, 128, (size_t)-1};
```

```
int which_freelist(size_t s) {
    int ind = 0;
    while (s > size_classes[ind])
        ind++;
    return ind;
}
```

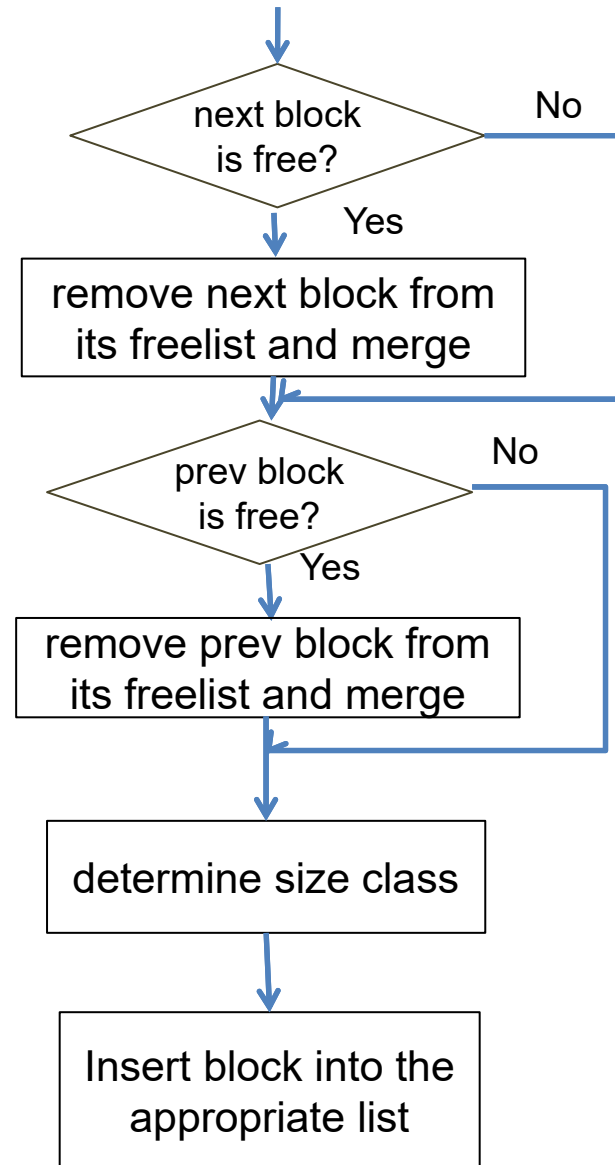
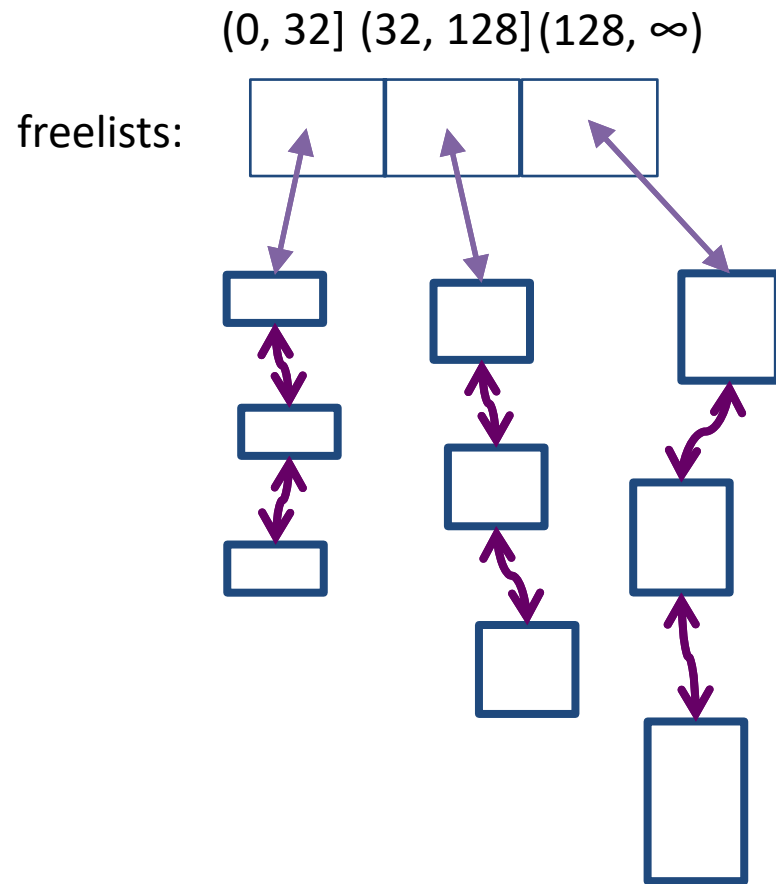
```
void init() {
    free_hdr *h = get_block_from_OS(1024);
    freelist[which_freelist(1024)] = h;
}
```



Segregated list: allocation



Segregated list: free



Buddy System

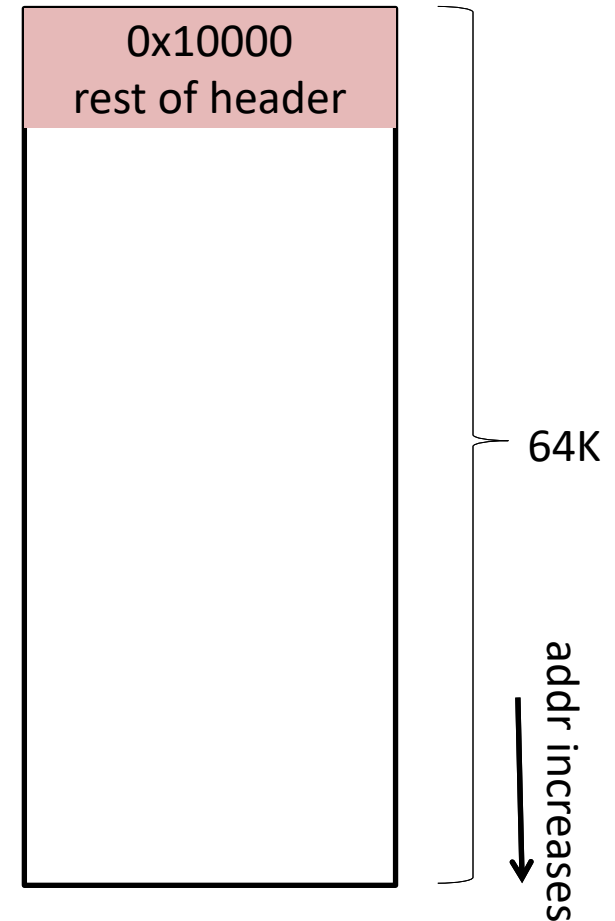
- A special case of segregated list
 - each freelist has *identically-sized* blocks
 - block sizes are powers of 2
- Advantage over a normal segregated list?
 - Less search time (no need to search within a freelist)
 - Less coalescing time
- Adopted by Linux kernel and jemalloc

Simple binary buddy system

Initialize:

- assume heap starts at the address of all zeros
 - Implementation can add an offset

$(0000\ 0000\ 0000\ 0000)_2$



Binary buddy system: allocate

```
p = malloc(16000);
```

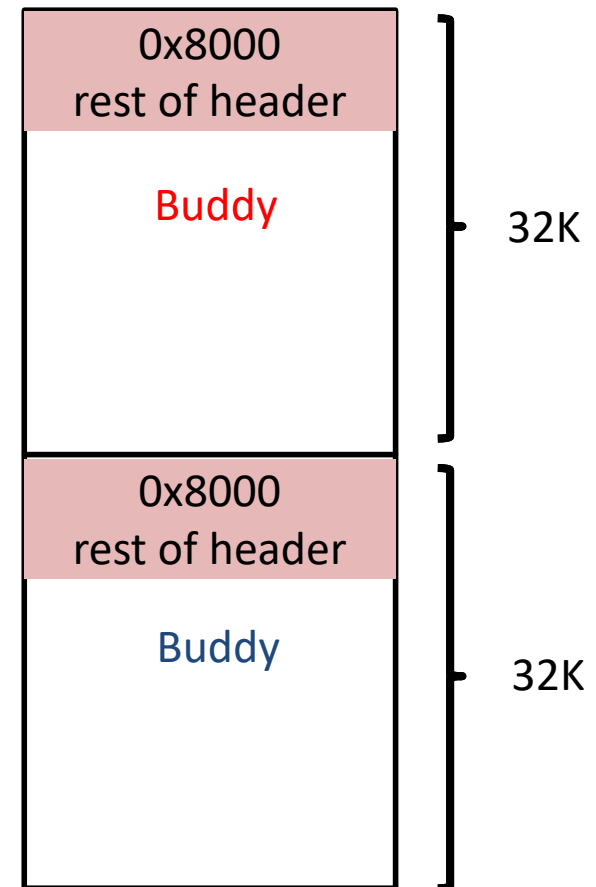
Recursive split in half until having the right size

- insert free buddy into appropriate freelist

Addresses of buddies at size 2^m differ in exactly 1-bit at position m (from right)

$(0000\ 0000\ 0000\ 0000)_2$

$(1000\ 0000\ 0000\ 0000)_2$

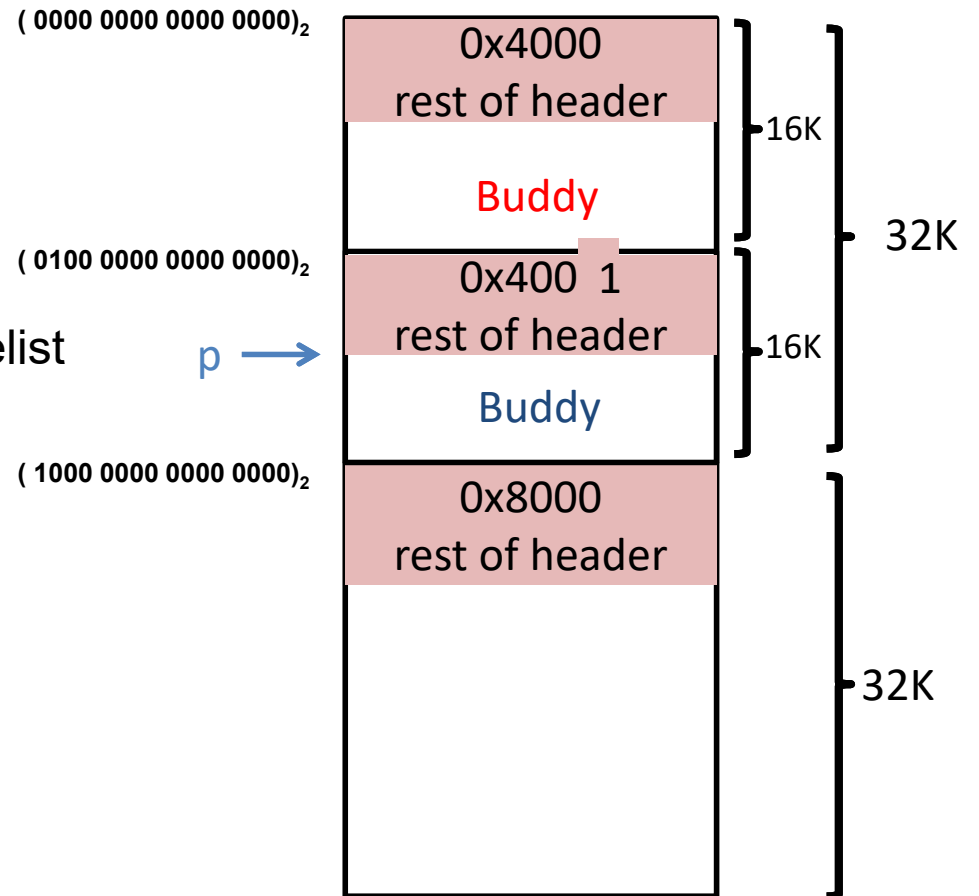


Binary buddy system: allocate

```
p = malloc(16000);
```

Recursive split in half until having the right size

- insert free buddy into appropriate freelist



Binary buddy system: free

`free(p);`

Recursively merge block with buddy

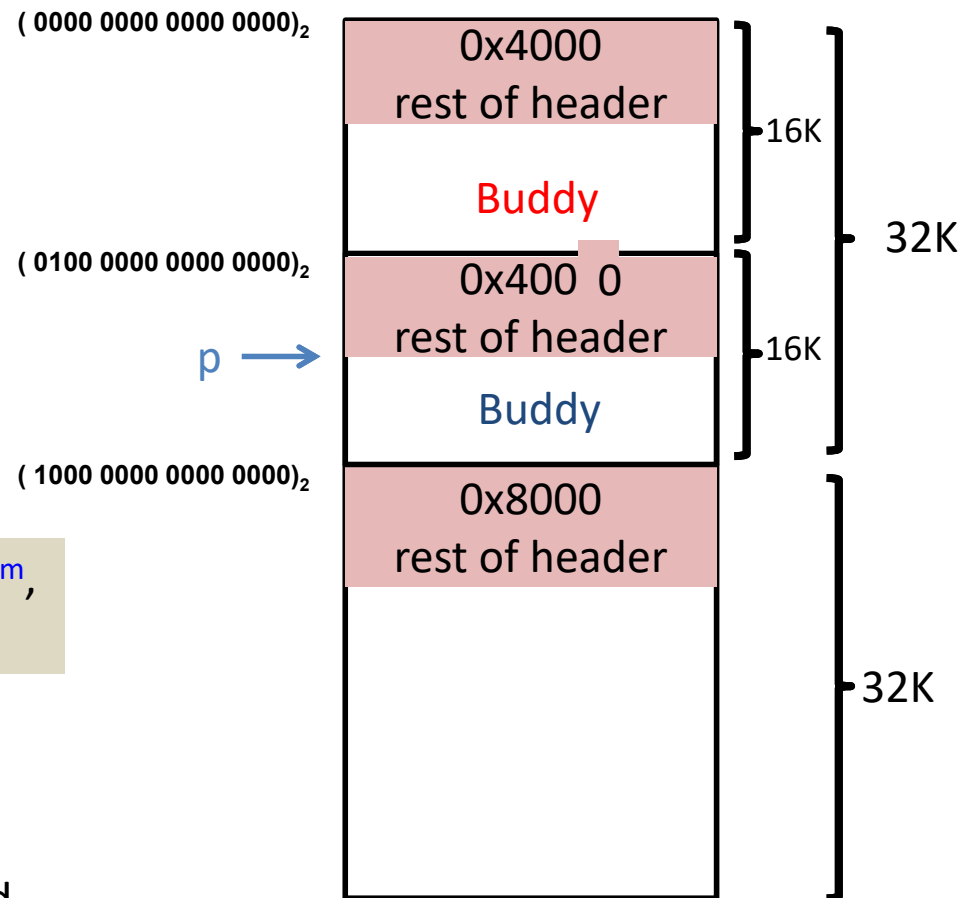
1. Calculate addr of buddy block,
determine buddy status

Question: given addr a of block with size 2^m ,
how to calculate its buddy's address?

$$a \oplus (1 \ll m)$$



any bit XOR 0 = unchanged
any bit XOR 1 = flipped

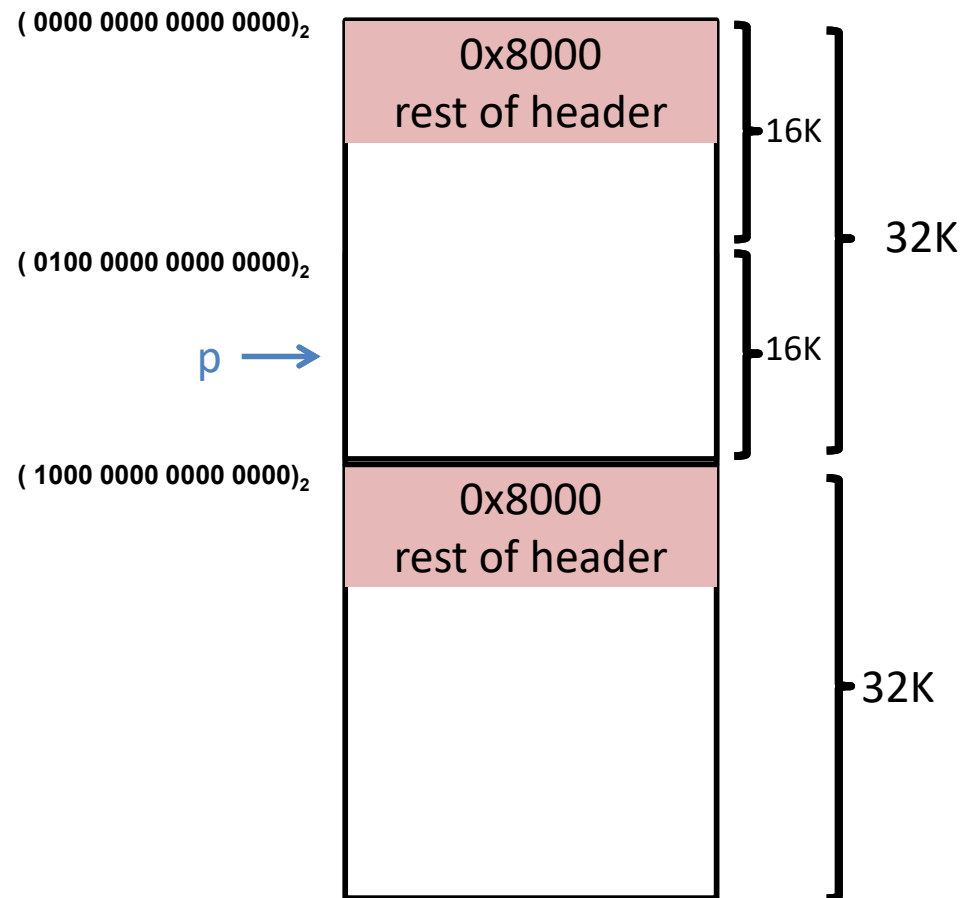


Binary buddy system: free

`free(p);`

If buddy is free:

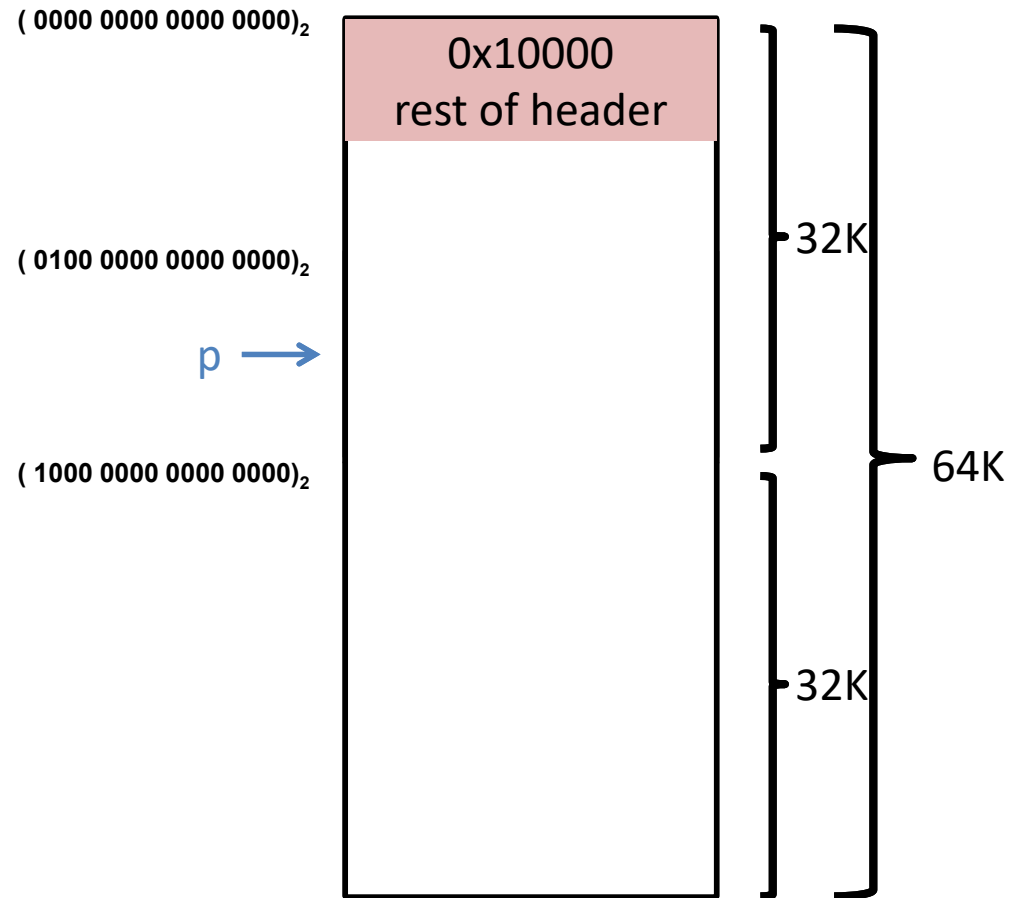
2. Detach free buddy from its list
3. Combine with current block



Binary buddy system: free

`free(p);`

Repeat to merge with larger buddy
Insert final block into appropriate
freelist



Summary

- Dynamic memory allocation
- Design constraints:
 - Free API does not include size
 - Space cannot be moved around
- Evolution of designs
 - Implicit list
 - Explicit list
 - Segregated list
 - Buddy system

