Implementing malloc

CS 351: Systems Programming

Michael Saelee <lee@iit.edu>

IIT College of Science
ILLINOIS INSTITUTE OF TECHNOLOGY

the API:

```
void *malloc(size_t size);

void free(void *ptr);

void *realloc(void *ptr, size_t size);
```

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

- returns a pointer to the payload (of min length `size` bytes) of a memory block
- this memory is *off-limits* to the DMA until released by the user

```c
void free(void *ptr);
```

- indicates to the DMA that the payload pointed to by `ptr` can be reused
- value of `ptr` must have been returned by a previous call to `malloc`

```c
void *realloc(void *ptr, size_t size);
```

- request to resize payload region pointed to by `ptr` to `size`
- DMA may allocate a new block
  - old data is copied to new payload
  - old payload is freed

realloc, by example

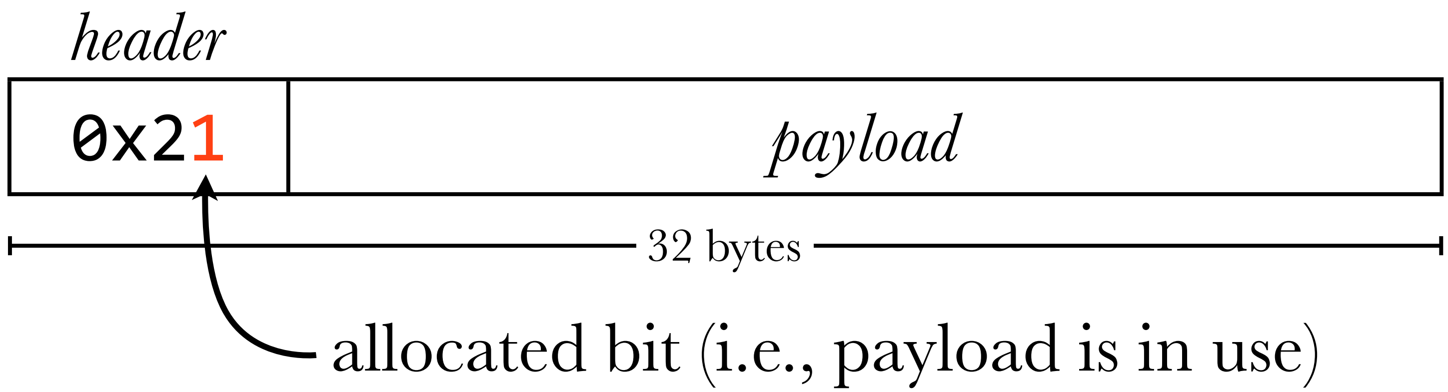
```
// allocate an array of 5 ints
int *arr = malloc(5 * sizeof(int));

// populate it
for (i=0; i<5; i++)
    arr[i] = i;

// sometime later, we want to "grow" the array
arr = realloc(arr, 10 * sizeof(int));

// arr may point to a new region of memory, but
// the old contents are copied over!
for (i=0; i<5; i++)
    printf("%d ", arr[i]); // => 0 1 2 3 4

// and now we have more room
for (i=5; i<10; i++)
    arr[i] = i;
```

basic implementation issues:

- tracking block metadata
- searching for and managing free space
- performing allocations

typical metadata = *size* & *allocation status*

- usually store in a block “header”
- if size is aligned to > 2 bytes, can use *bottom bit* of size for *allocated bit*


header
0x21
payload
32 bytes
allocated bit (i.e., payload is in use)

after `free`:


header
0x20
free for reuse
32 bytes

0x21	*payload*

important: payload should be *aligned* (i.e., begin on multiple of alignment size)

- usually means that header & block also be aligned

e.g., Linux requires 8-byte alignment

```c
#define ALIGNMENT 8 // must be a power of 2

#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~(ALIGNMENT-1))

for (i=1; i<=32; i+=2) {
    printf("ALIGN(%d) = %d\n",
           i, ALIGN(i));
```

→

```
ALIGN(1) = 8
ALIGN(3) = 8
ALIGN(5) = 8
ALIGN(7) = 8
ALIGN(9) = 16
ALIGN(11) = 16
ALIGN(13) = 16
ALIGN(15) = 16
ALIGN(17) = 24
ALIGN(19) = 24
ALIGN(21) = 24
ALIGN(23) = 24
ALIGN(25) = 32
ALIGN(27) = 32
ALIGN(29) = 32
ALIGN(31) = 32
```

```c
#define ALIGNMENT 8 // must be a power of 2

#define ALIGN(size) (((size) + (ALIGNMENT-1)) & ~(ALIGNMENT-1))

#define SIZE_T_SIZE (ALIGN(sizeof(size_t))) // header size

// super-naive allocator
void *malloc(size_t size) {
    size_t blk_size = ALIGN(size + SIZE_T_SIZE);
    size_t *header  = sbrk(blk_size);
    *header = blk_size | 1;   // mark allocated bit
    return (char *)header + SIZE_T_SIZE;
}

void free(void *ptr) {
    size_t *header = (char *)ptr - SIZE_T_SIZE;
    *header = *header & ~1L;  // unmark allocated bit
}
```

this implementation doesn’t reuse blocks!

to reuse blocks, must search the heap for a free block ≤ required size

```
void *find_fit(size_t size) {
    size_t *header = heap_start();
    while (header < heap_end()) {
        if (!(*header & 1) && *header >= size)
            return header;
        header = (char *)header + (*header & ~1L);
    }
    return NULL;
}

void *malloc(size_t size) {
    size_t blk_size = ALIGN(size + SIZE_T_SIZE);
    size_t *header  = find_fit(blk_size);
    if (header) {
        *header = *header | 1;
    } else {
        header  = sbrk(blk_size);
        *header = blk_size | 1;
    }
    return (char *)header + SIZE_T_SIZE;
}
```

```c
void *malloc(size_t size) {
    size_t blk_size = ALIGN(size + SIZE_T_SIZE);
    size_t *header  = find_fit(blk_size);
    if (header) {
        *header = *header | 1;
    } else {
        header  = sbrk(blk_size);
        *header = blk_size | 1;
    }
    return (char *)header + SIZE_T_SIZE;
}
```

very inefficient — when re-using a block, always occupies the *entire block*!

- better to *split* the block if possible and reuse the unneeded part later

```c
void *malloc(size_t size) {
    size_t blk_size = ALIGN(size + SIZE_T_SIZE);
    size_t *header  = find_fit(blk_size);
    if (header) {
        *header = *header | 1;
    } else {
        header  = sbrk(blk_size);
        *header = blk_size | 1;
    }
    return (char *)header + SIZE_T_SIZE;
}
```

```c
void *malloc(size_t size) {
    size_t blk_size = ALIGN(size + SIZE_T_SIZE);
    size_t *header  = find_fit(blk_size);
    if (header && blk_size < *header)
        // split block if possible (FIXME: check min block size)
        *(size_t *)((char *)header + blk_size) = *header - blk_size;
    else
        header = sbrk(blk_size);
    *header = blk_size | 1;
    return (char *)header + 8;
}
```

```c
void *find_fit(size_t size) {
    size_t *header = heap_start();
    while (header < heap_end()) {
        if (!(*header & 1) && *header >= size)
            return header;
        header = (char *)header + (*header & ~1L);
    }
    return NULL;
}
```

we call this an ***implicit list*** based DMA

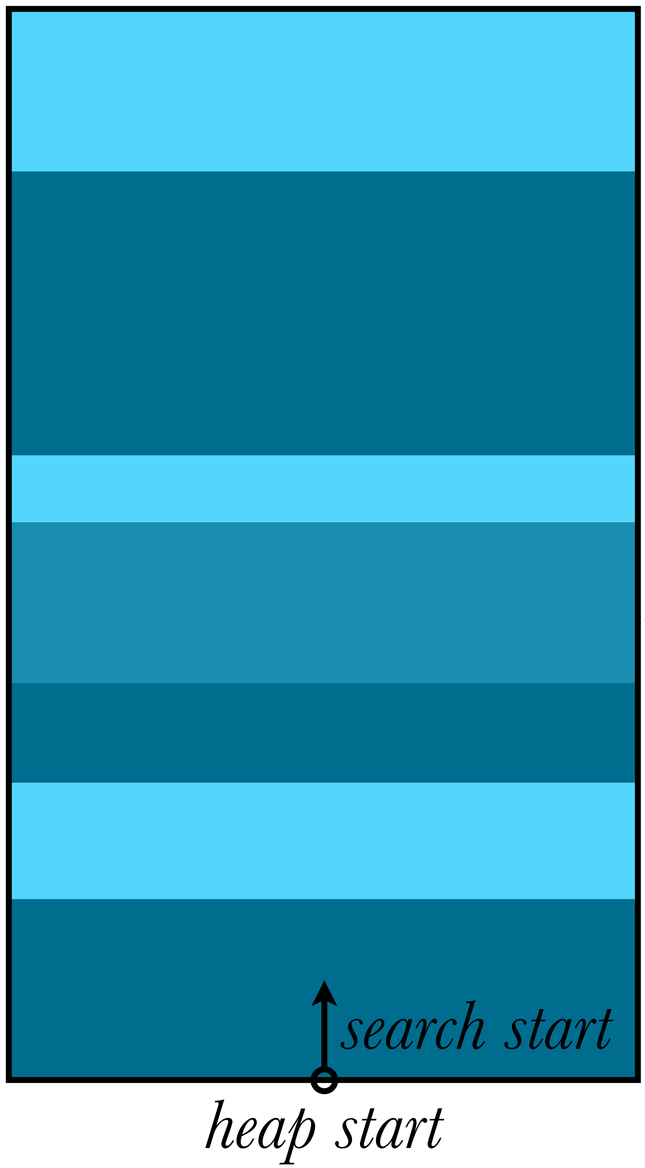
- navigating through blocks using sizes
- $O(n)$ search, where $n = \#$ blocks
    - n comprises allocated & free blocks!

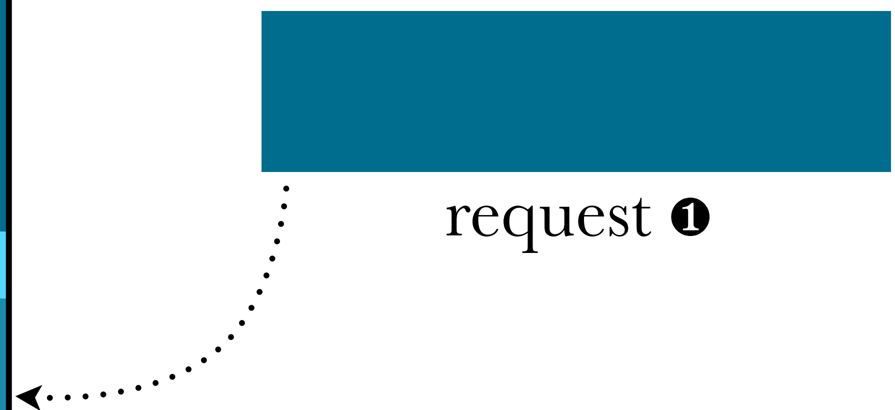
```
void *find_fit(size_t size) {
    size_t *header = heap_start();
    while (header < heap_end()) {
        if (!(*header & 1) && *header >= size)
            return header;
        header = (char *)header + (*header & ~1L);
    }
    return NULL;
}
```

to tune utilization & throughput, may pick from different search heuristics

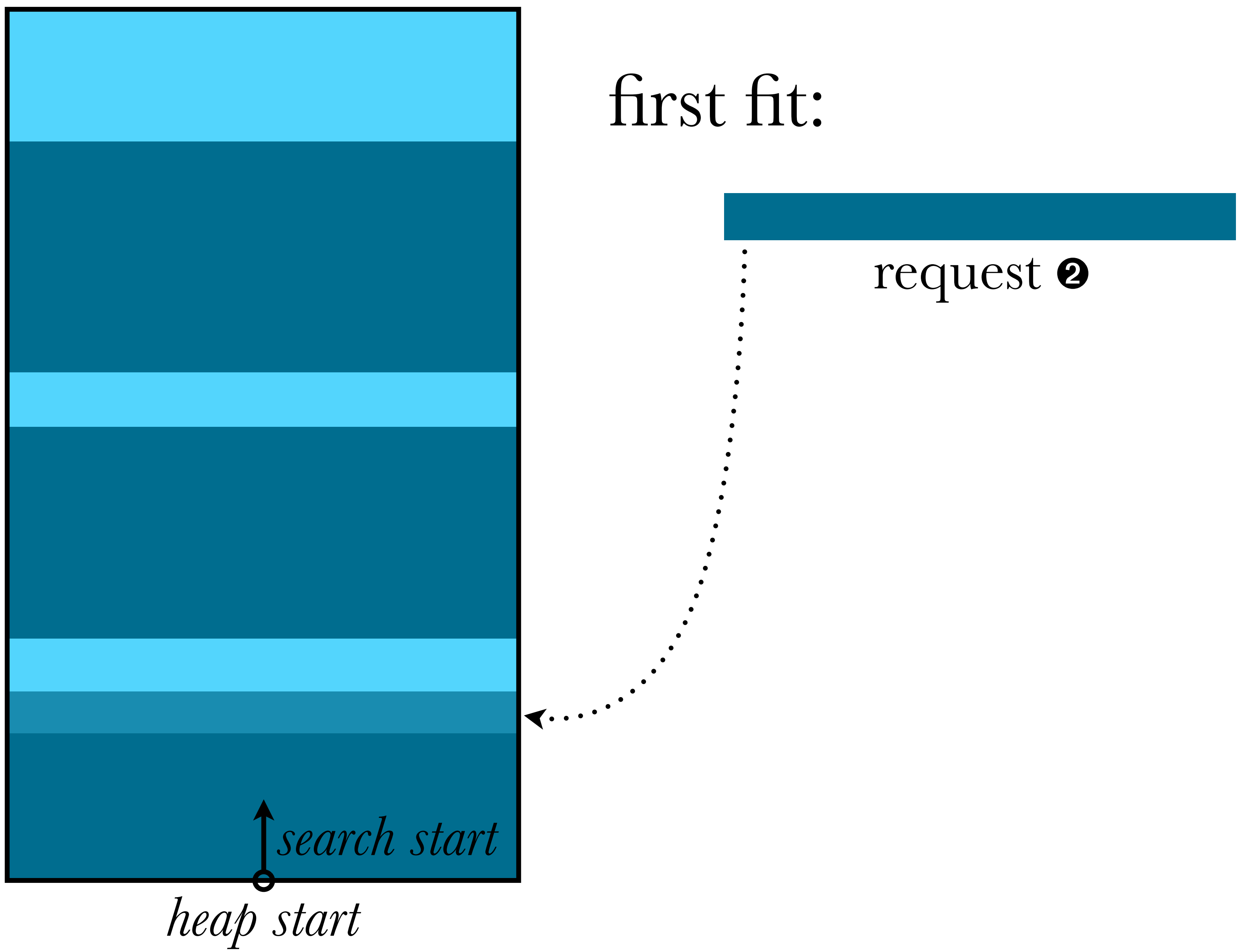
- *first-fit* (shown above)
- *next-fit* (requires saving last position)
- *best-fit* ($\Theta(n)$ time)

first fit:

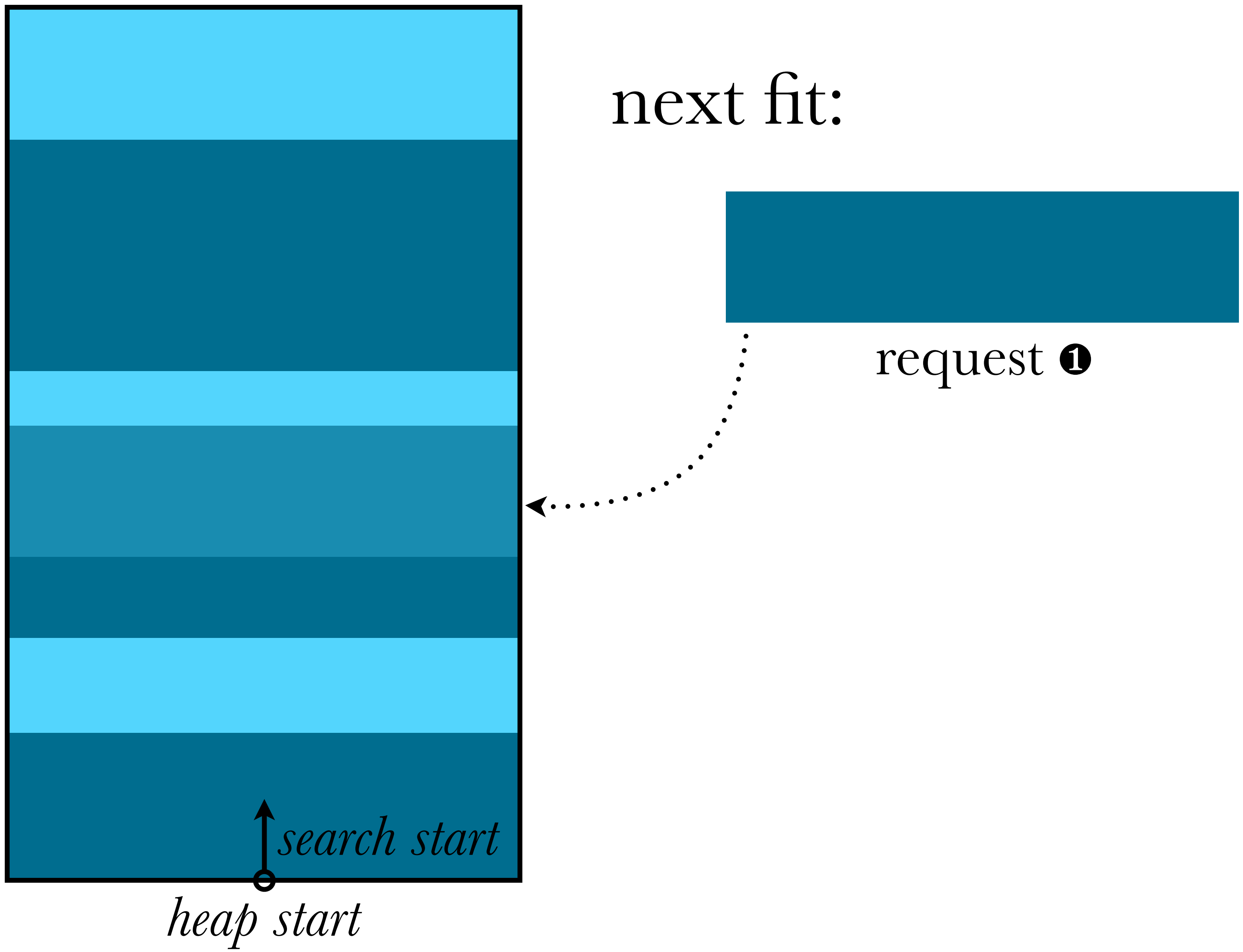


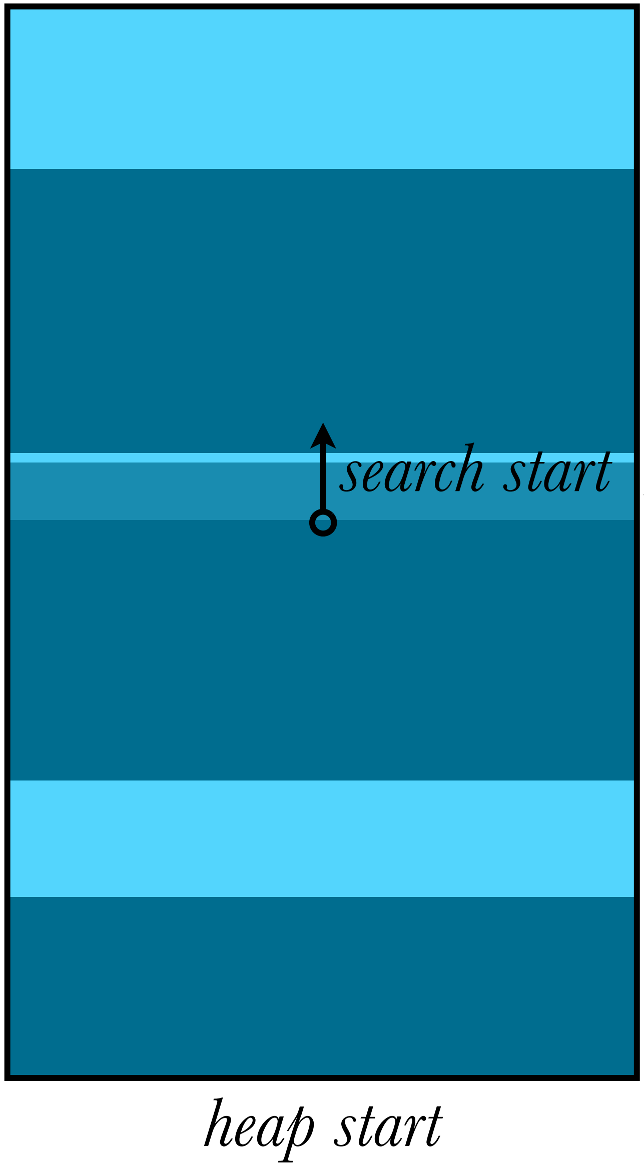


first fit:



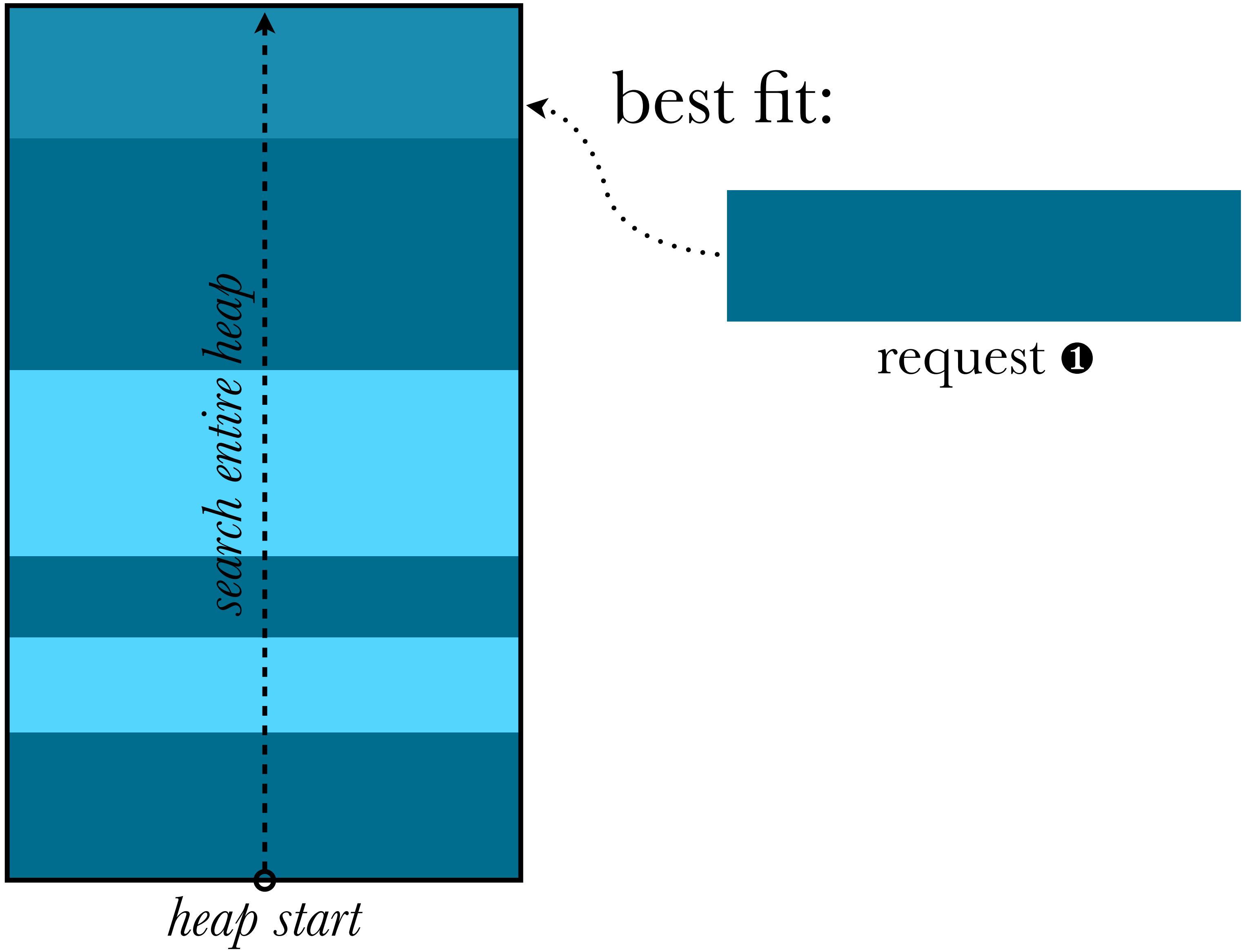
next fit:



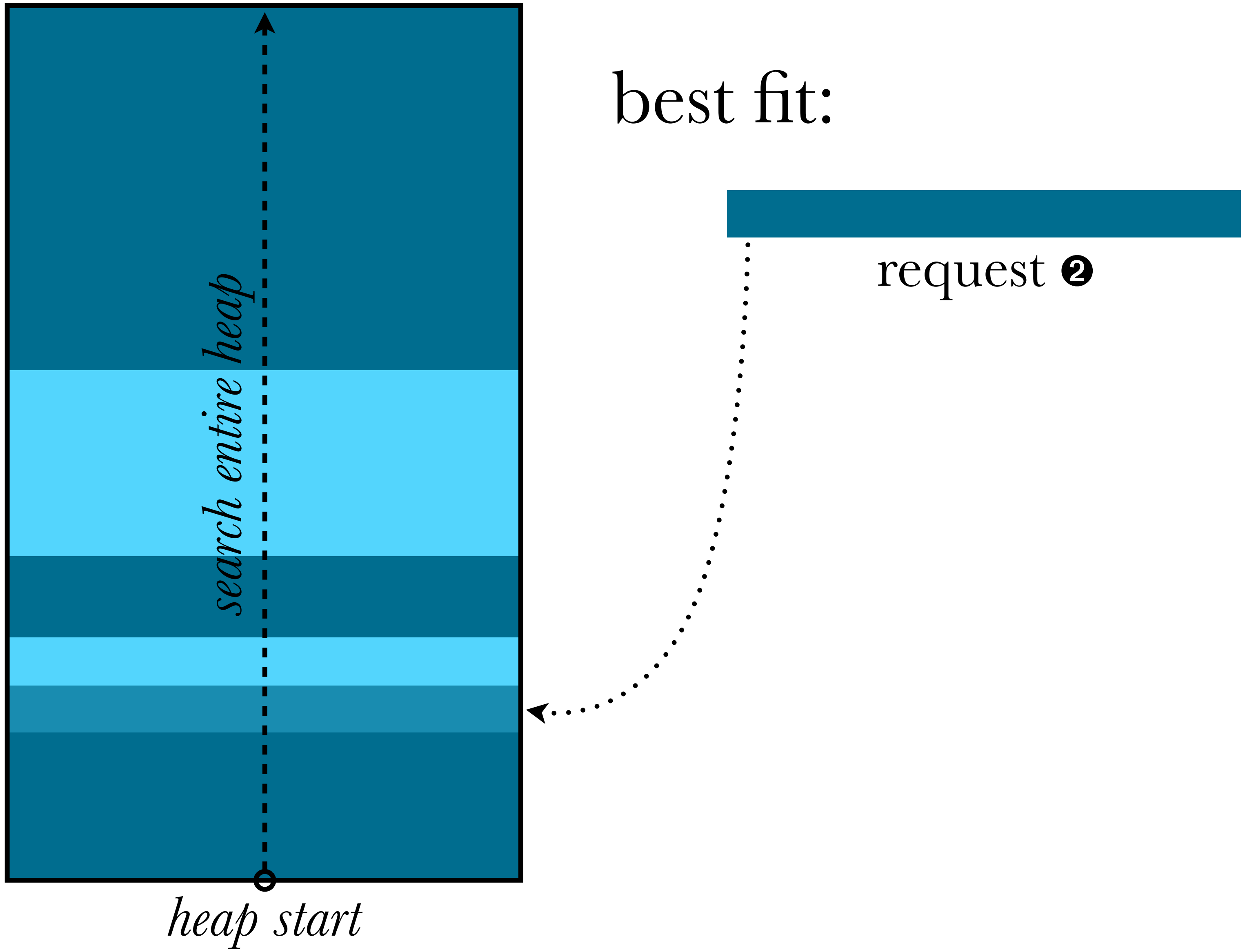


next fit:

best fit:



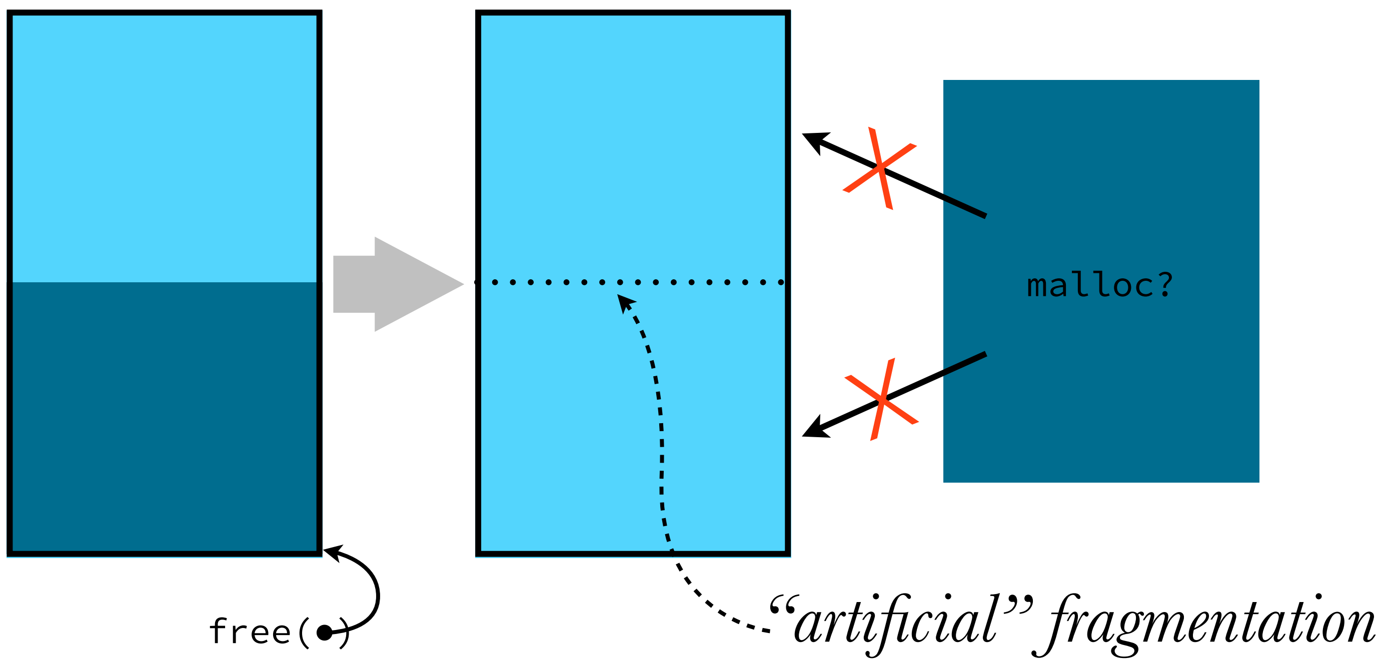
best fit:



intuitively, best fit *likely* improves utilization

- but at the expense of throughput and higher likelihood of scattering blocks
- note: “best fit” is not a complete strategy — what to do in case of a tie?

```c
void free(void *ptr) {
    size_t *header = (char *)ptr - SIZE_T_SIZE;
    *header = *header & ~1L;
}
```



need to *coalesce* adjacent free blocks

have a choice of when to do this:

1. at search time: *deferred* coalescing
2. when freeing: *immediate* coalescing

1. deferred coalescing

```
void *find_fit(size_t size) {
    size_t *header = heap_start(),
           *next;
    while (header < heap_end()) {
        if (!(*header & 1)) {
            if (*header >= size)
                return header;
            next = (char *)header + *header;
            // merge with next block if available & free
            if (next < heap_end() && !(*next & 1)) {
                *header += *next;
                continue;
            }
        }
        header = (char *)header + (*header & ~1L);
    }
    return NULL;
}
```

to pick up all free blocks, requires the entire heap to be searched from the start

1. deferred coalescing

```
void *find_fit(size_t size) {
    size_t *header = heap_start(),
           *next;
    while (header < heap_end()) {
        if (!(*header & 1)) {
            if (*header >= size)
                return header;
            next = (char *)header + *header;
            // merge with next block if available & free
            if (next < heap_end() && !(*next & 1)) {
                *header += *next;
                continue;
            }
        }
        header = (char *)header + (*header & ~1L);
    }
    return NULL;
}
```

also may result in a cascade of merges during search — *indeterminate performance*

2. immediate coalescing

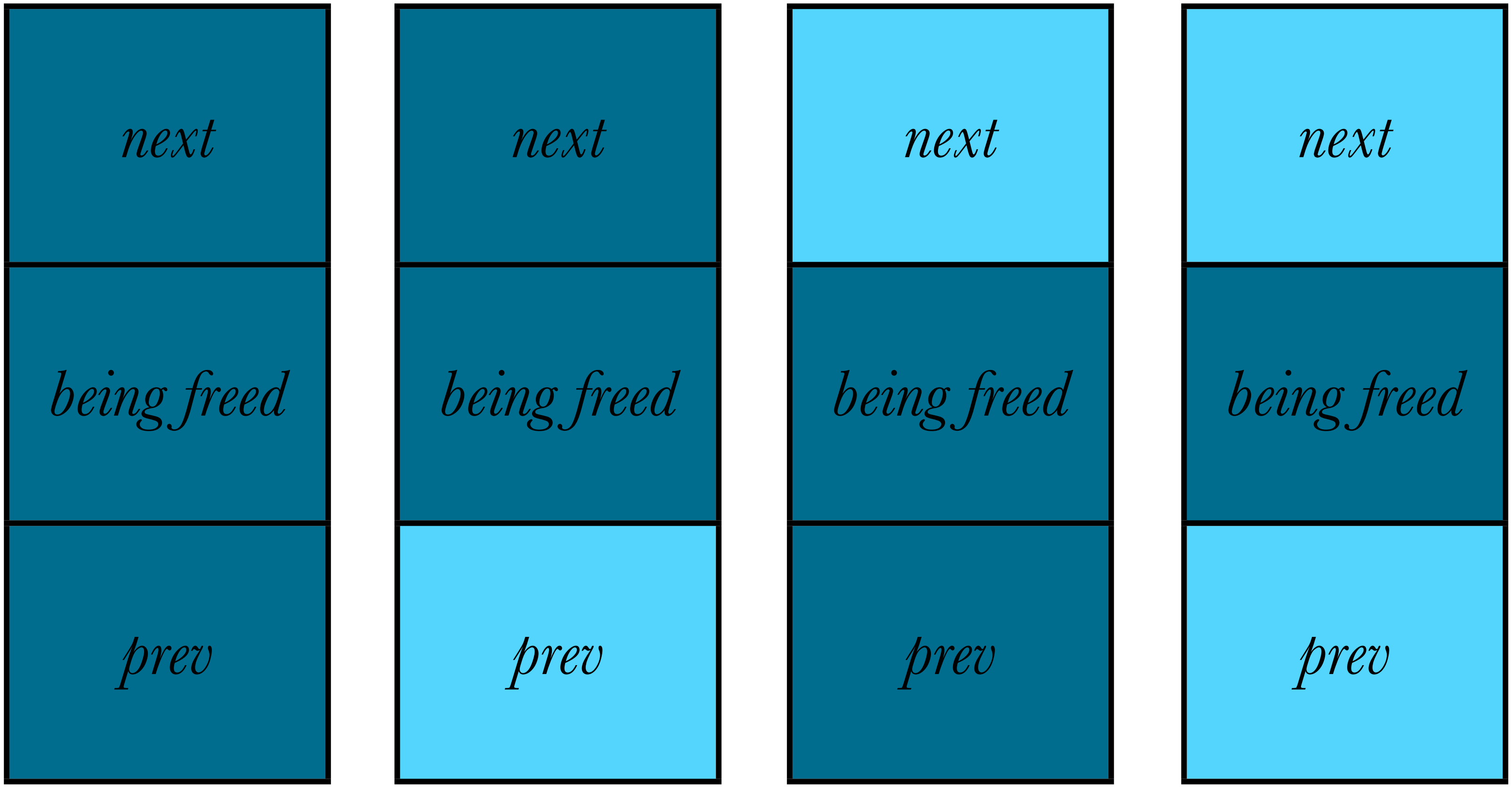
```
void free(void *ptr) {
    size_t *header = (char *)ptr - SIZE_T_SIZE,
           *next;
    *header = *header & ~1L;

    // coalesce if possible
    next = (char *)header + *header;
    if (next <= heap_end() && !(*next & 1)) {
        *header += *next;
    }
}
```

but what about the previous block?

— can't get to it! (singly-linked list issues) update block structure: include footer to support bi-directional navigation

header	payload + padding	**footer**

referred to as block “boundary tags”



4 scenarios; coalescing = $O(1)$ operation

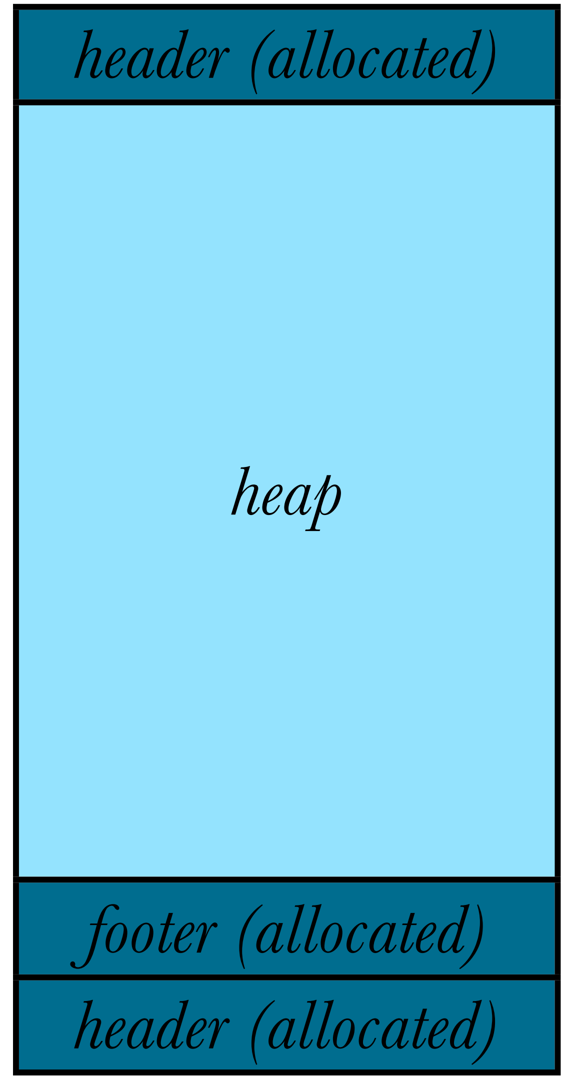
```c
// given pointer to free block header, coalesce with adjacent blocks
// and return pointer to coalesced block
void *coalesce(size_t *bp) {
    size_t *next = (char *)bp + (*bp & ~1L),
           *prev = (char *)bp - (*(size_t *)((char *)bp-SIZE_T_SIZE) & ~1L);
    int next_alloc = *next & 1,  // FIXME: potential segfault!
        prev_alloc = *prev & 1,  // FIXME: potential segfault!

    if (prev_alloc && next_alloc) {
        return bp;
    } else if (!prev_alloc && next_alloc) {
        *prev += *bp; // header
        *(size_t *)((char *)bp + *bp - SIZE_T_SIZE) = *prev; // footer
        return prev;
    } else if (prev_alloc && !next_alloc) {
        ...
    } else {
        ...
    }
}
```

```
// given pointer to free block header, coalesce with adjacent blocks
// and return pointer to coalesced block
void *coalesce(size_t *bp) {
    size_t *next, *prev;
    int next_alloc, prev_alloc;

    // must deal with edge cases!
    if (heap_start() < bp) {
        prev = (char *)bp - (*(size_t *)((char *)bp-SIZE_T_SIZE) & ~1L)
        prev_alloc = *prev & 1;
    } else {
        prev_alloc = 1; // sane choice
    }

    // same for next and next_alloc

    ...
}
```

edge cases arise everywhere!

convenient to introduce *sentinel prologue & epilogue blocks*

- simplify test cases
- create on heap init and move on expansion



finally, realloc:

```
void *realloc(void *ptr, size_t size) {
    // note: not dealing with footers
    size_t *header = (size_t *)((char *)ptr - SIZE_T_SIZE);
    size_t oldsize = *header & ~1L,
           newsize = ALIGN(size + SIZE_T_SIZE);
    void *newptr;

    if (oldsize >= newsize) {
        return ptr;
    } else {
        newptr = malloc(size);
        memcpy(newptr, ptr, oldsize - SIZE_T_SIZE);
        free(ptr);
        return newptr;
    }
}
```

```
newptr = malloc(size);
memcpy(newptr, ptr, oldsize - SIZE_T_SIZE);
free(ptr);
```

$= O(n)$ malloc, $n =$ total # blocks

$+ O(m)$ copy, $m =$ size of payload

very expensive! (and `realloc` is intended to provide room for optimization) ideas for optimization:

- try to "grow" block in place
    - always possible if at end of heap
- *pre-allocate* more then required; quite reasonable if already `realloc`'d

implicit-list summary:

- $O(n)$ malloc; n = total # blocks
- $O(1)$ free (with immediate coalescing)
- $O(n+m)$ realloc; n driven by malloc,
  m payload size

would greatly improve performance
to search *only free blocks*

use an ***explicit list***

i.e., store size & pointers in free blocks to create a doubly-linked list

note: allocated blocks still store just size
& allocated bit

```c
typedef struct free_blk_header {
    size_t size;
    struct free_blk_header *next;
    struct free_blk_header *prior;
} free_blk_header_t;

// init heap with a permanent (circular) free list head
void init_heap() {
    free_blk_header_t *bp = sbrk(ALIGN(sizeof(free_blk_header_t)));
    bp->size  = 0;
    bp->next  = bp;
    bp->prior = bp;
}

void *malloc(size_t size) {
    // instead of the following, use mm_init in the malloc lab!
    static int heap_inited = 0;
    if (!heap_inited) {
        heap_inited = 1;
        init_heap();
    }
    ...
}
```

```
typedef struct free_blk_header {
    size_t size;
    struct free_blk_header *next;
    struct free_blk_header *prior;
} free_blk_header_t;

void *find_fit(size_t length) {
    free_blk_header_t *bp = heap_start();
    for (bp = bp->next; bp != heap_start(); bp = bp->next) {
        // find first fit
        if (bp->size >= length) {
            // remove from free list and return
            bp->next->prior = bp->prior;
            bp->prior->next = bp->next;
            return bp;
        }
    }
    return NULL;
}
```

```c
// blocks must be able to accommodate a free block header
#define MIN_BLK_SIZE ALIGN(sizeof(free_blk_header_t))


void *malloc(size_t size) {

    // init_heap stuff from before goes here

    size_t *header;
    int blk_size = ALIGN(size + SIZE_T_SIZE);

    blk_size = (blk_size < MIN_BLK_SIZE)? MIN_BLK_SIZE : blk_size;

    header = find_fit(blk_size);
    if (header) {
        *header = ((free_blk_header_t *)header)->size | 1;
        // *header = *header | 1; <-- also works (why?)

        // FIXME: split if possible
    } else {
        header = sbrk(blk_size);
        *header = blk_size | 1;
    }
    return (char *)header + SIZE_T_SIZE;
}
```

when freeing (or splitting), must manually add freed block to the explicit list (vs. just updating allocated bit in implicit list)

```c
void free(void *ptr) {
    free_blk_header_t *header = (char *)ptr - SIZE_T_SIZE,
                      *free_list_head = heap_start();

    // add freed block to free list after head
    header->size  = *(size_t *)header & ~1L;

    // add freed block to free list after head
    header->next  = free_list_head->next;
    header->prior = free_list_head;
    free_list_head->next = free_list_head->next->prior = header;

    // FIXME: coalesce! (requires adding footers, too)
}
```

adding freed block at head = LIFO search

other policies: FIFO & address ordered

but search is always $O(n)$, n = # *free* blocks

(linear linked structure)

(still a huge potential throughput increase over implicit list!)

how to improve search speed (esp. best-fit)?

can make this arbitrarily complex:

```
typedef struct free_blk_header {
    size_t size;
    struct free_blk_header *next;
    struct free_blk_header *prior;
} free_blk_header_t;
```

e.g., for a tree structure:

```
typedef struct free_blk_header {
    size_t size;
    struct free_blk_header *parent;
    struct free_blk_header *left;
    struct free_blk_header *right;
} free_blk_header_t;
```

we can view this as a straightforward data structure implementation

but this is a perilous path!

— distances us from the problem domain

some domain-specific issues:

- real-world programs (that use the allocator) exhibit exploitable patterns
  - e.g., allocation ramps, plateaus, peaks, and common request sizes
- locality of allocations is important!

but must also take care to not *overspecialize* a general-purpose allocator!

viz., "*premature optimization is the root of all evil*" (D. Knuth)

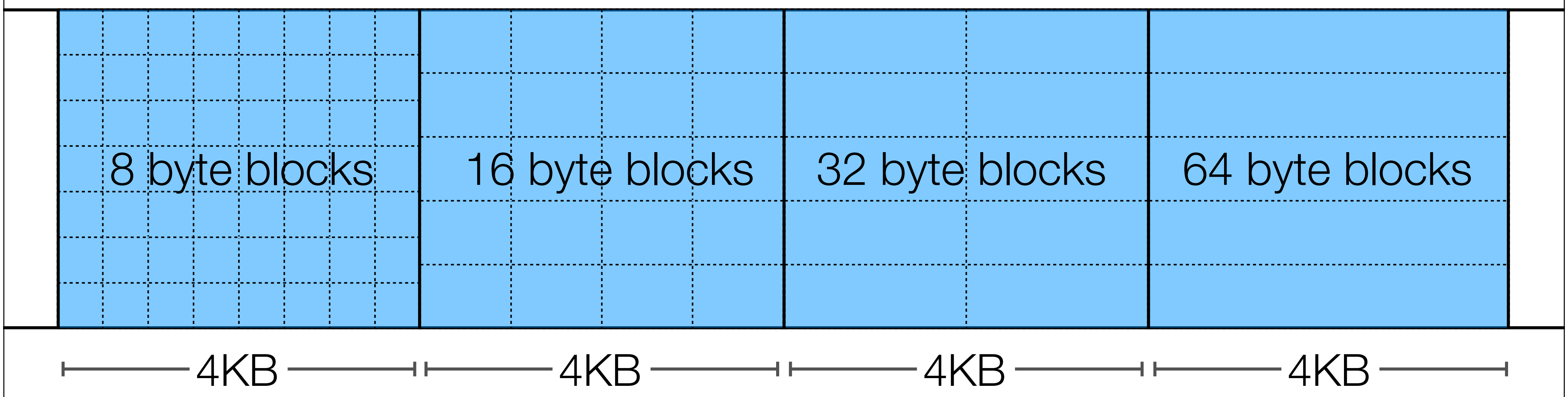
— different programs will likely exhibit different request patterns/distributions other common implementation strategies:

1. simple segregated storage
2. segregated fits
3. buddy systems

1. *simple segregated storage*

- pre-allocate lists of fixed block sizes, where a list occupies full VM page(s)
- track lists in a small, fast lookup table

Heap


8 byte blocks
16 byte blocks
32 byte blocks
64 byte blocks
4KB
4KB
4KB
4KB

malloc(k):

- allocate first free block in list for smallest size $\geq k$
- if list is empty, set aside a new page for blocks of matching size

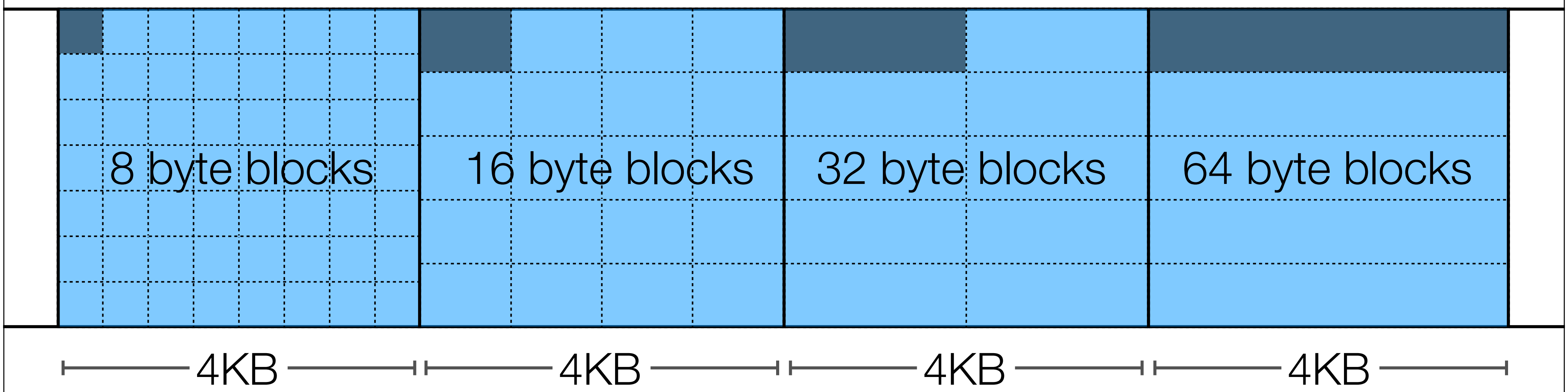
free:

- mark as free; don't coalesce
- if page becomes empty, can reuse for another size

simple & fast search and allocation

also: low metadata overhead & good locality for similarly sized requests

tradeoff: massive fragmentation!



by itself not a viable general-purpose allocator, but may be used to service frequent requests of predictable size

- i.e., as a "caching" allocator
- Linux kernel internally uses something like this (known as *slab allocator*)

2. *segregated fits*

- maintain separate explicit free lists of varying *size classes*
- dynamically manage blocks in lists

malloc(k):

- look in list of size $\geq k$
- allocate first empty block
- split if possible (using some threshold), putting leftover on appropriate list

free:

- free and, if possible, coalesce
- add block to the appropriate list (may result in moving coalesced blocks)

approximates best fit (i.e., *good* fit) with high speed by reducing search space

- may choose not to coalesce (or defer coalescing) for smaller, common sizes

```c
#define NUM_SIZE_CLASSES 5

size_t min_class_size[] = { MIN_BLK_SIZE, 64, 128, 256, 1024 };

typedef struct free_blk_header {
    size_t size;
    struct free_blk_header *next;
    struct free_blk_header *prior;
} free_blk_header_t;

// global array of pointers to doubly-linked free lists
free_blk_header_t *free_lists;


void init_heap() {
    int i;
    free_lists = sbrk(NUM_SIZE_CLASSES * sizeof(free_blk_header_t));
    for (i=0; i<NUM_SIZE_CLASSES; i++) {
        free_lists[i].size  = 0;
        free_lists[i].next  = free_lists[i].prior = &free_lists[i];
    }
    return 0;
}
```

```c
size_t min_class_size[] = { MIN_BLK_SIZE, 64, 128, 256, 1024 };
free_blk_header_t *free_lists;


void *find_fit(size_t size) {
    int i;
    free_blk_header_t *fp;
    for (i=0; i<NUM_SIZE_CLASSES; i++) {
        // locate the first suitable list that isn't empty
        if (min_class_size[i] >= size
                && free_lists[i].next != &free_lists[i]) {
            // take the first block (no searching!)
            fp = free_lists[i].next;
            // remove it from the free list
            free_lists[i].next = fp->next;
            fp->next->prior = &free_lists[i];
            // and try to split it
            try_split(fp, size);
            return fp;
        }
    }
    // FIXME: do a full search of "top" list if not found!
    return NULL;
}
```

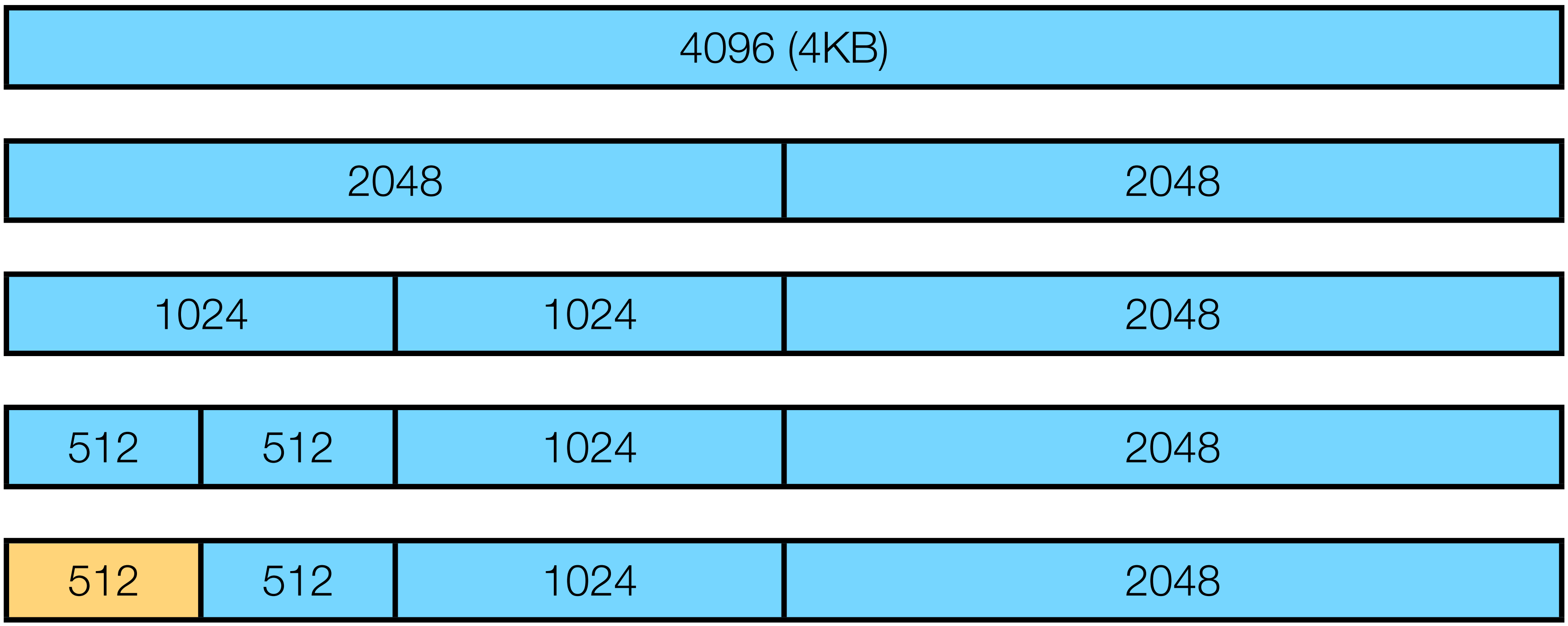
```c
size_t min_class_size[] = { MIN_BLK_SIZE, 64, 128, 256, 1024 };
free_blk_header_t *free_lists;


void try_split(free_blk_header_t *fp, size_t needed) {
    int i, remaining = fp->size - needed;
    free_blk_header_t *sp;
    if (remaining < MIN_BLK_SIZE)
        return;
    // split the block ...
    fp->size = needed;
    sp = (free_blk_header_t *)((char *)fp + needed);
    sp->size = remaining;

    // ... and put the leftover free block in the correct list
    for (i=NUM_SIZE_CLASSES-1; i>0; i--)
        if (min_class_size[i] <= remaining) {
            sp->prior = &free_lists[i];
            sp->next = free_lists[i].next;
            free_lists[i].next = free_lists[i].next->prior = sp;
            break;
        }
}
```

3. *buddy systems*

- each block (starting with the whole heap) may be split into two sub-blocks at a preset boundary

e.g., “binary buddies”

`malloc(450)`



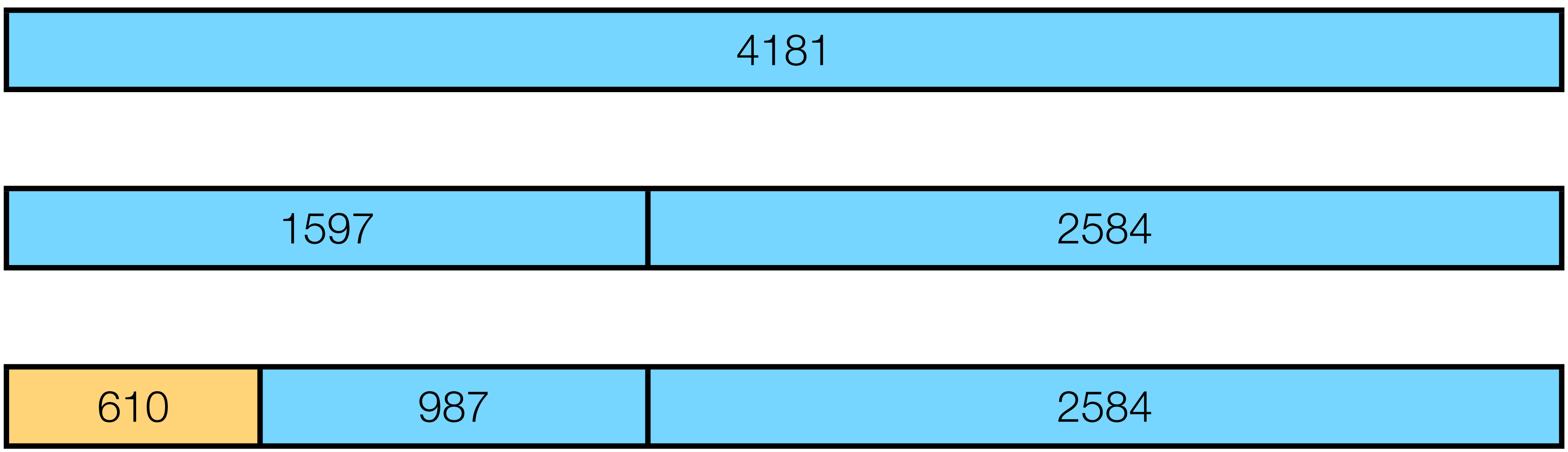
e.g., "binary buddies"

Fibonacci sequence:

1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181

e.g., “Fibonacci buddies”

`malloc(450)`



very little block overhead:

- free/allocated bit
- is block "whole" or split?
- (size not needed!)

in practice, however, internal fragmentation is much worse than segmented fits good reading: "Doug Lea's malloc"

http://gee.cs.oswego.edu/dl/html/malloc.html hybrid allocator:

- best fit; segregated fits
  - LRU for tie-breaking
- deferred coalescing
- “mmap” for large requests