

C malloc implementation

CS351 : Saelee

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

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

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

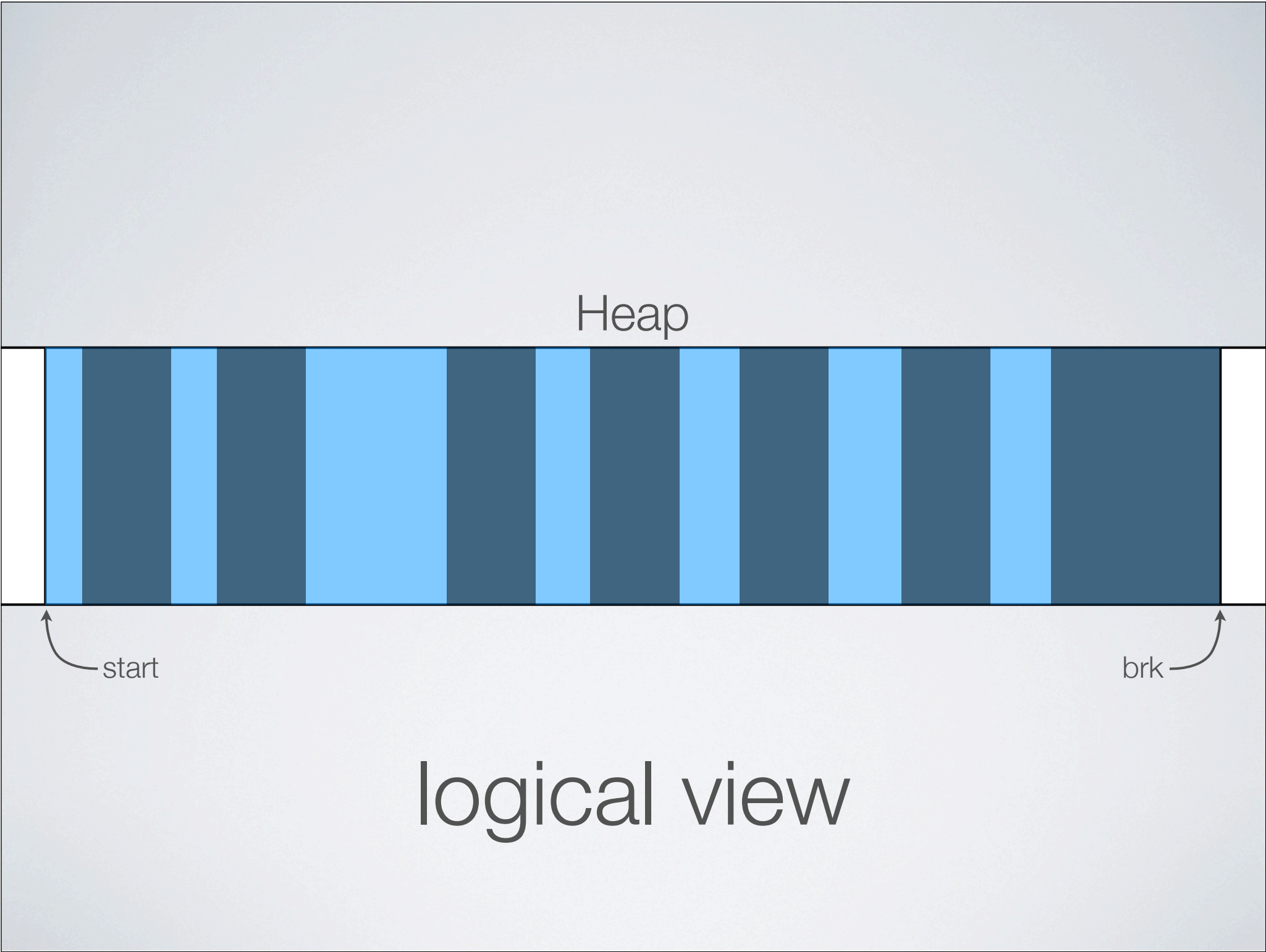
C Standard Library

requirements:

- alloc/dealloc in any order
- cannot modify or move allocated blocks

basic issues:

- tracking free space
- tracking allocated blocks
- storing block metadata



Heap

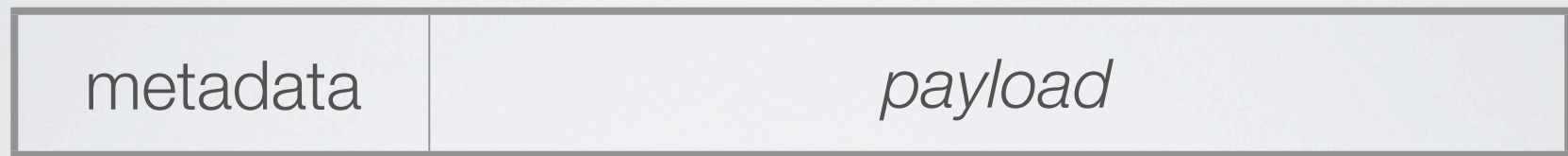
logical view

start

brk

heap records (blocks) must be
self-describing

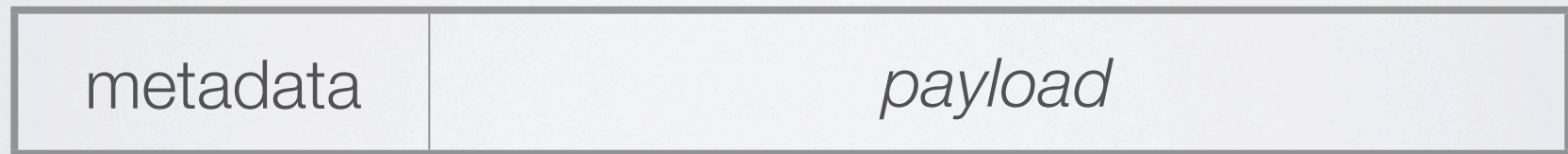
1. storing block **metadata**



↑
“header” field

block structure

`malloc(10)`



10 bytes

pointer returned to application

block size

`malloc(10)`



need to distinguish
allocated vs. free blocks

allocated *bit*

- use least significant bit
of header word

malloc(10)



1111



0x0000000F

payload

4 bytes

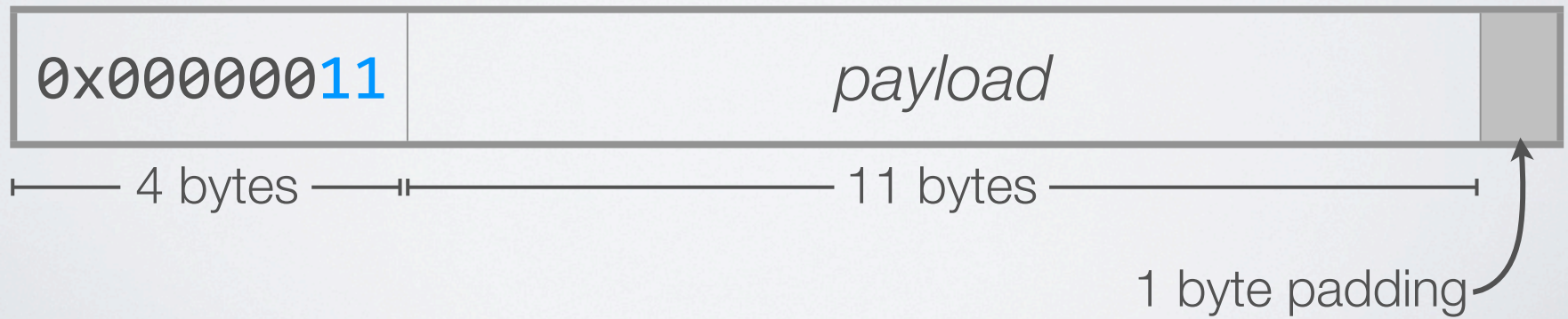
10 bytes

malloc(**11**)??



require block size *alignment*
to powers of 2

`malloc(11)`



2. tracking **allocated** blocks

not really necessary!

application responsibilities:

1. save pointers to payloads
2. call free to deallocate

otherwise, memory leak!

3. tracking **free** blocks

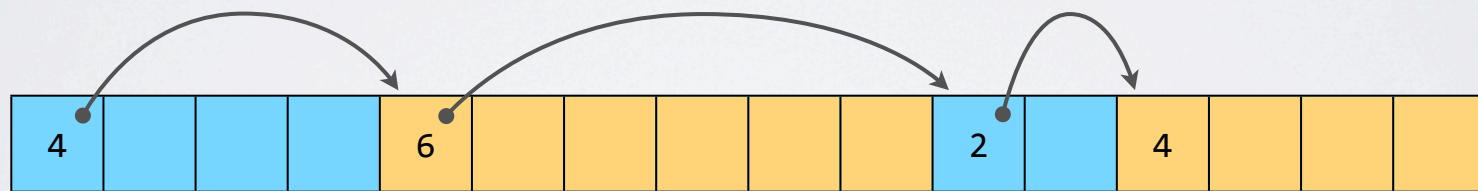
primary objective:

locate blocks in which
to allocate space for
incoming requests

other goals:

- fast search
- flexible & easy manipulation
- minimize fragmentation

strategy 1: “implicit” list



① search

```

void *find_fit(int length) {
    char *p = heap_start;           // search from beginning
    do {
        if (!(*p & 1) && *p >= length) // if free & big enough
            return p;                // found fit!

        p += *p & ~1;               // move to next block
    } while (p < heap_end);         // search till end

    return NULL;                    // no fit found!
}

```

simplification: using a single byte for header

“first fit”

$O(N)$

next fit

keep a static “cursor”

$O(N)$

best fit

always search entire heap

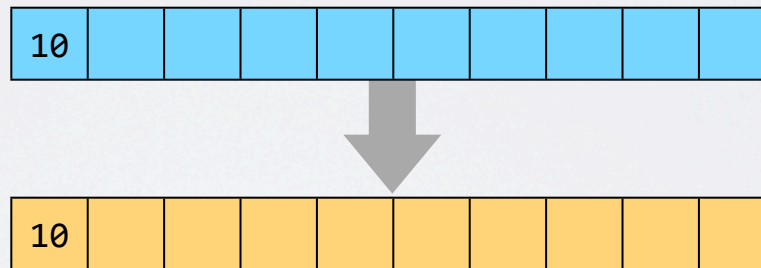
$\Theta(N)$

intuitively, **best fit** gives
the **best utilization**
(smallest “gaps”)

② allocate

```
*p |= 1; // set the allocated bit
```

```
malloc(2)
```



(bit of a waste)

should allow the free block
to be **split**, when possible

```

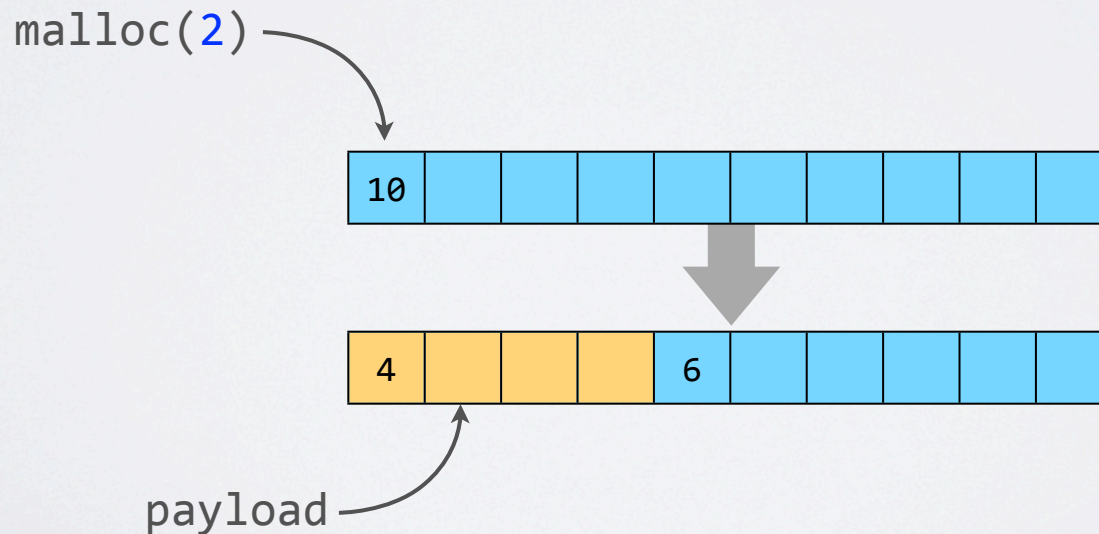
void *malloc(int size) {
    int newsize = ((size + 2) >> 1) << 1;    // round to next even size
    char *p = find_fit(newsize);             // find fit
    int oldsize = *p;                        // get size of block found

    *p = newsize | 1;                        // allocate and update size

    if (oldsize > newsize)                   // if block is too large
        *(p + newsize) = oldsize - newsize; // split it!

    return p + 1;                            // return pointer to payload
}

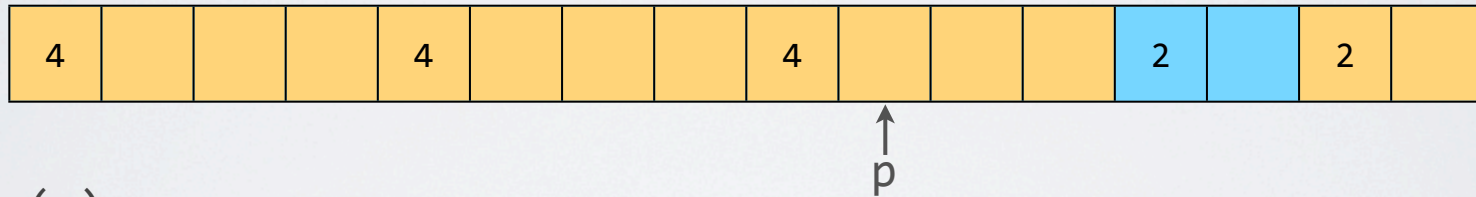
```



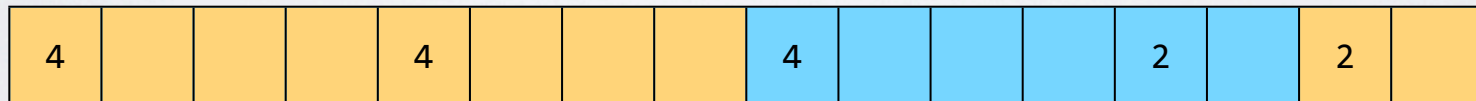
if `find_fit` fails, should call
`sbrk` to extend heap!

③ free

```
void free(void *p) {  
    char *bp = p-1; // get pointer to header  
    *bp = *bp & ~1; // zero out the allocated bit  
}
```



free(p)



malloc(5)

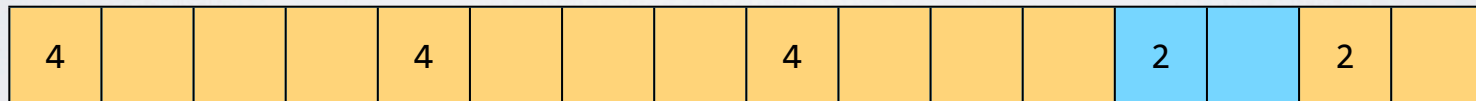
artificial fragmentation

④ coalesce

```

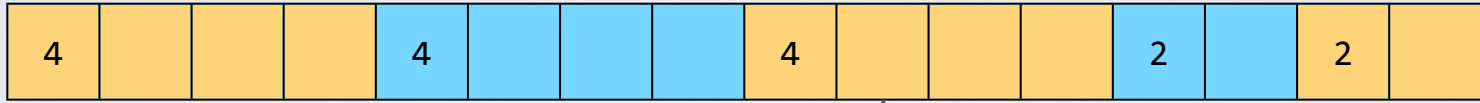
void free(void *p) {
    char *bp = p-1;           // get pointer to header
    *bp = *bp & ~1;          // free this block
    char *np = bp + *bp;     // find next block
    if (!(*np & 1))          // if next block is free
        *bp = *bp + *np;    // coalesce!
}

```



free(p)





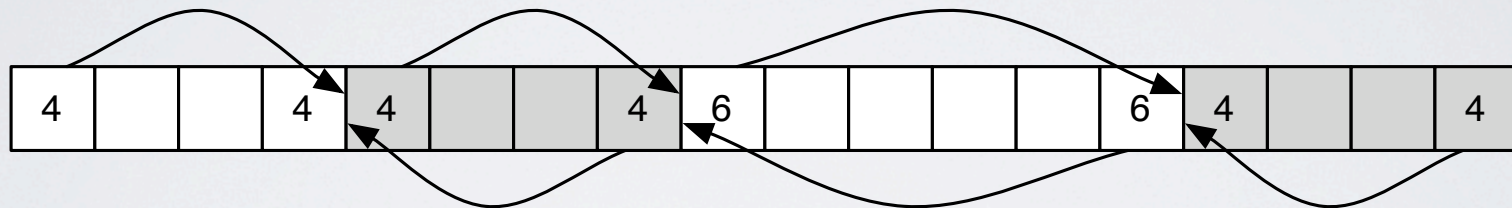
free(p)



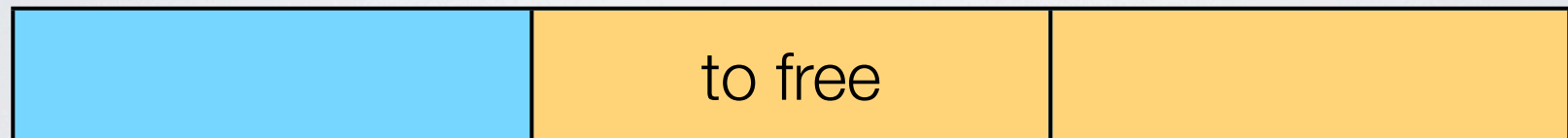
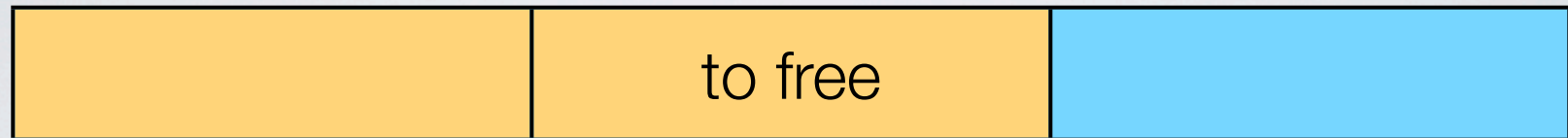
problem: cannot navigate to previous block!

boundary tags

header	“payload” + padding	footer
---------------	---------------------	---------------



effectively: doubly-linked list



$O(1)$ coalescing

```
void coalesce(void *bp) {
    void *next_bp = bp + (*bp & ~1),      // next block
        *prev_bp = bp - (*(bp-1) & ~1);  // prior block (via footer)

    int next_alloc = *next_bp & 1,
        prev_alloc = *prev_bp & 1;

    // coalesce if needed
    if (next_alloc && prev_alloc) {
        // case 1
    } else if (!next_alloc && prev_alloc) {
        // case 2
    } else if (next_alloc && !prev_alloc) {
        // case 3
    } else {
        // case 4
    }
}
```

alternative:

deferred coalescing

```
void *find_fit(size_t length) {
    char *p = heap_start,
          *np;
    do {
        if (!(*p & 1) && *p >= length)
            return p;

        np = p + (*p & ~1); // pointer to next

        if (!(*p & 1) && !(*np & 1)) // if both are free
            *p = *p + *np; // coalesce and retry
        else
            p += (*p & ~1); // else move to next
    } while (p < heap_end);

    return NULL;
}
```

problem with edge case (at end of heap)

Heap

header (allocated)

footer (allocated)

sentinel "prologue"
and "epilogue" blocks
(create on heap init
and expansion)

header (allocated)

start

brk



⑤ realloc

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

```
void *realloc(void *ptr, size_t size) {
    void *oldptr = ptr;
    void *newptr = malloc(size);           // find new block
    int copySize = *oldptr - 1;          // old payload size

    if (size < copySize)                 // if dest < source
        copySize = size;                 // adjust bytes to copy

    memcpy(newptr, oldptr, copySize);    // copy payload
    free(oldptr);                         // free old block

    return newptr;
}
```

naive approach!

always malloc = $O(N)$

+

copy payload ← expensive!

how to improve?

some ideas:

- try to grow/shrink block in place (no malloc)
- initially malloc more than necessary (trade-off!)

implicit list **analysis**

(a) **throughput**

search time + malloc = $O(N)$
N = total # blocks!

free + coalescing = $O(1)$

realloc = $O(N)$ + payload copy
(in worst case)

(b) utilization

depends on search
approach (`find_fit`)

first fit, next fit,

best fit

ideal

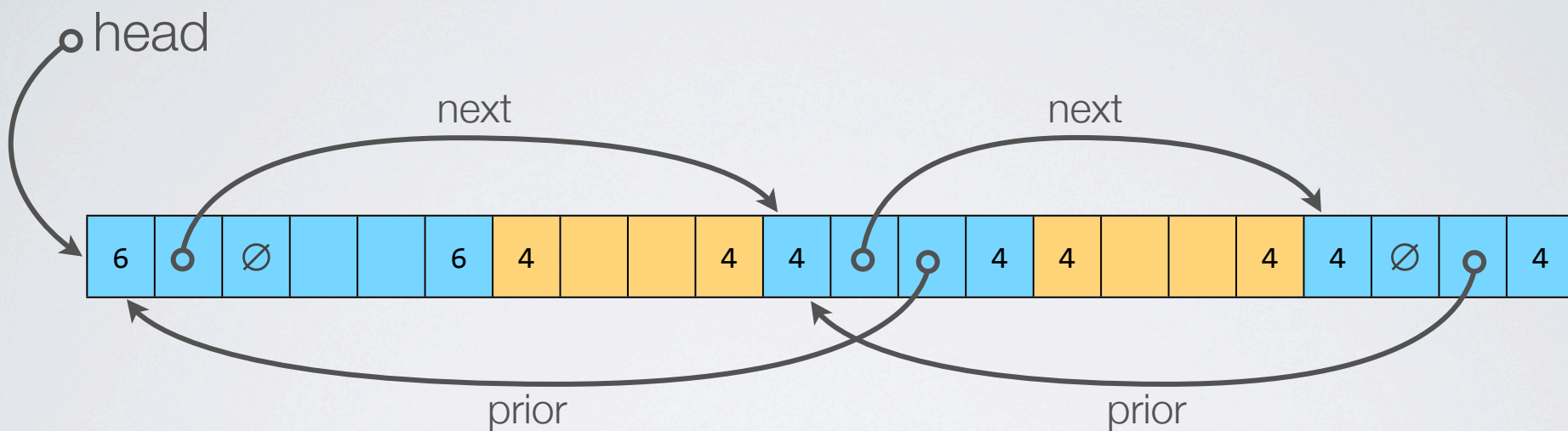


best fit = $\Theta(N)$

N = total # blocks!

ideally, search *free blocks only*

strategy 2: **explicit list**



additional overhead for
next/prior pointers
(+ internal fragmentation)

boundary tags still needed
for coalescing!

search = $O(N)$

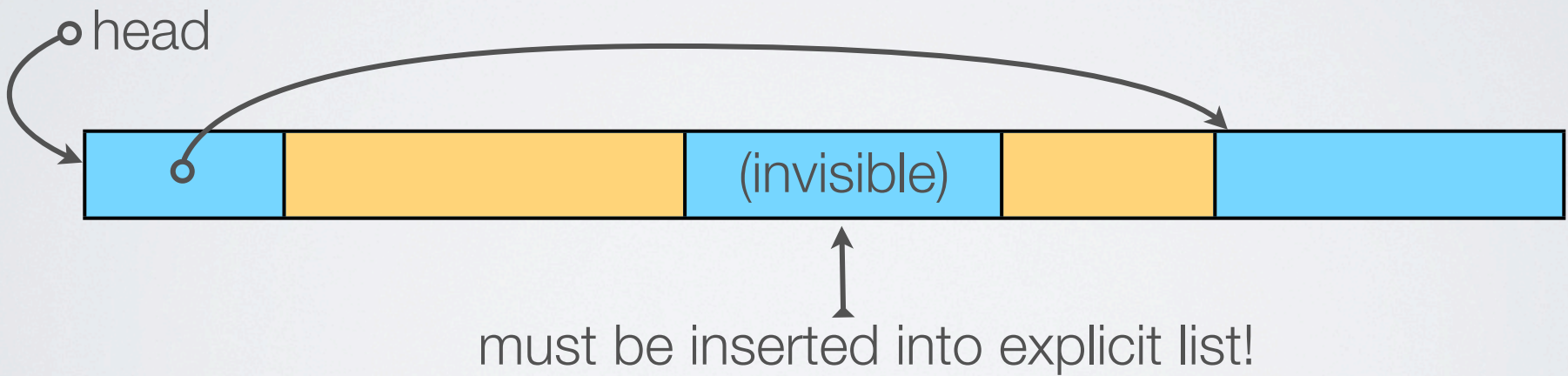
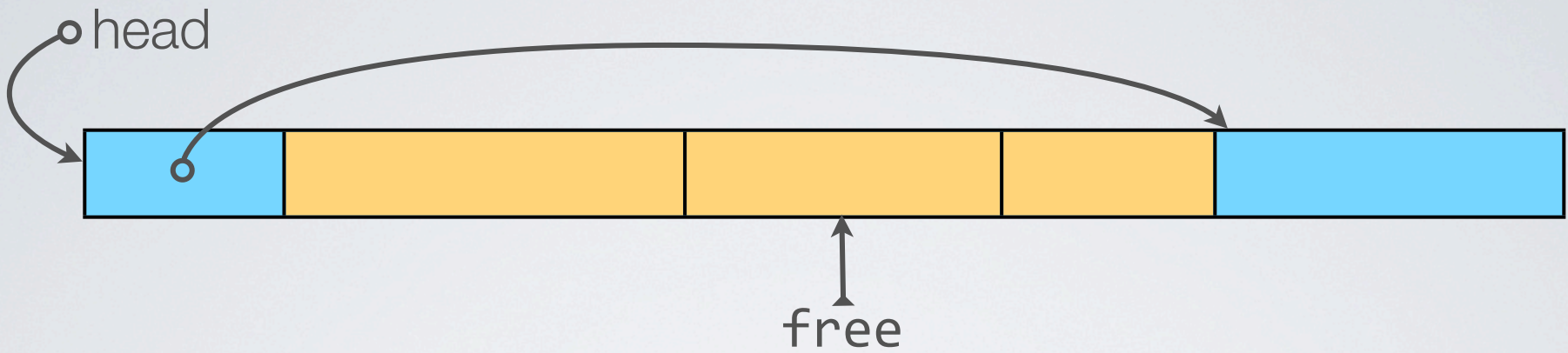
N = # of **free** blocks

```
typedef struct {  
    int size;  
    void *nextp;  
    void *priorp;  
} freeBlkHeader;
```

```
void *find_fit(int length) {  
    freeBlkHeader *bp = heap_start;  
  
    for (bp = heap_start;  
         bp != NULL && (bp->size & ~1) < length;  
         bp = bp->nextp) ;  
  
    return bp;  
}
```

malloc, free, realloc
mostly unchanged

but ... what to do with
new free blocks?



free blocks created when:

- freeing an allocated block
- coalescing free blocks
- splitting during allocation

insertion policy:

- FIFO (queue)
- LIFO (stack)
- address ordered
- sorted by size

when coalescing, may need
to remove and reinsert

deferred coalescing



explicit list **analysis**

similar to implicit list, but
 $O(N)$, $N = \#$ free blocks

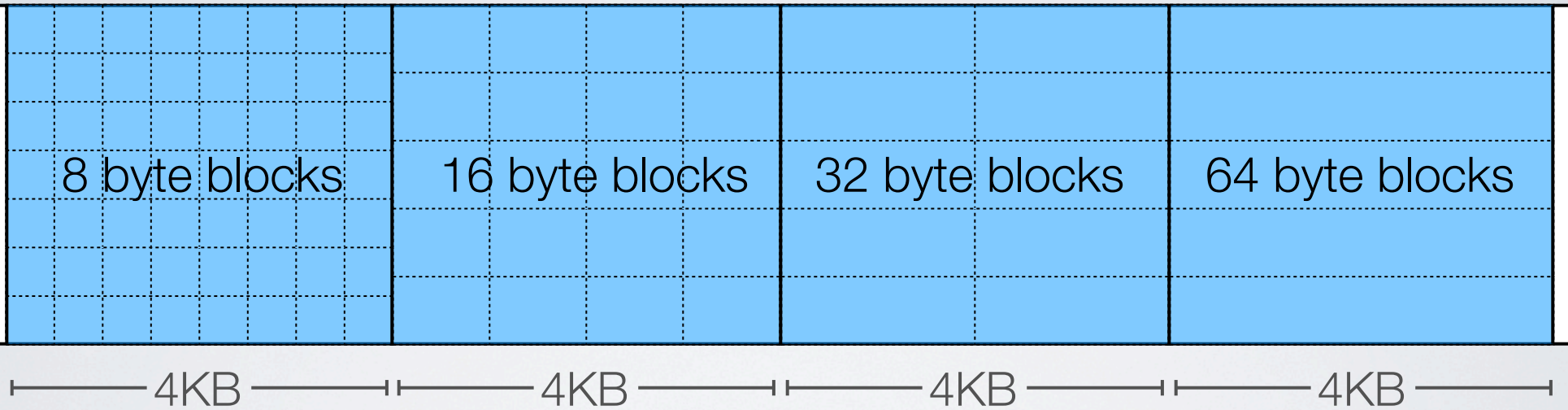
best fit is still inefficient ...
linked/linear data structure

strategy 3: **segregated lists**

(a) **simple** segregated
storage

allocate separate pages for
pre-set block *size classes*

Heap



each page = free list for size k

keep track of free blocks in
each page with
free-space bitmap

malloc(k):

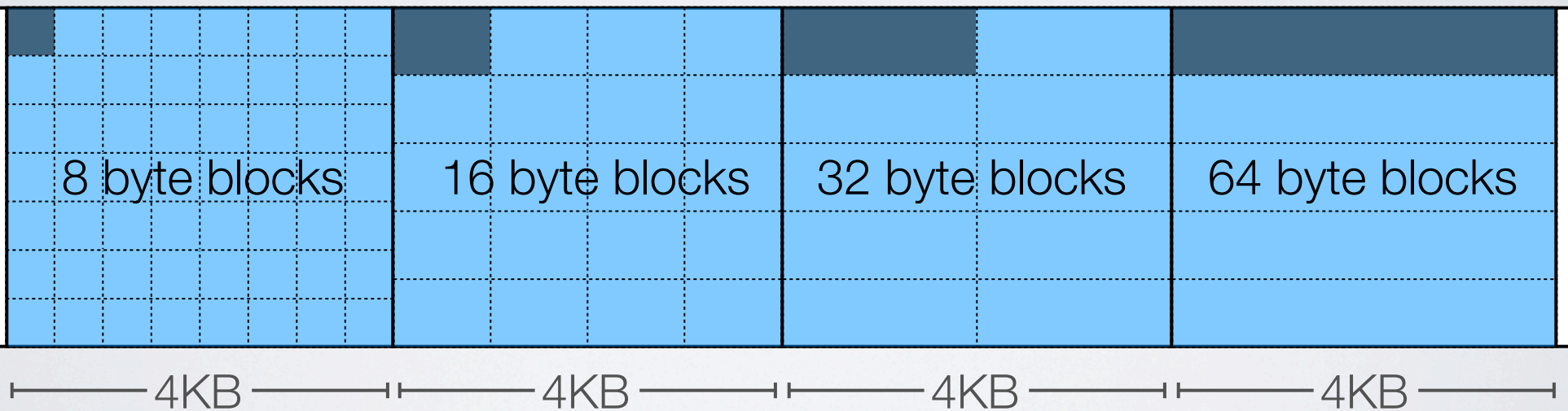
- allocate first free block on list for smallest size $\geq k$
- if list is empty, allocate new page

free:

- mark as free in bitmap
- if page is empty, can use for another size
- no coalescing!

constant time alloc/free!

tradeoff:
massive **fragmentation!**



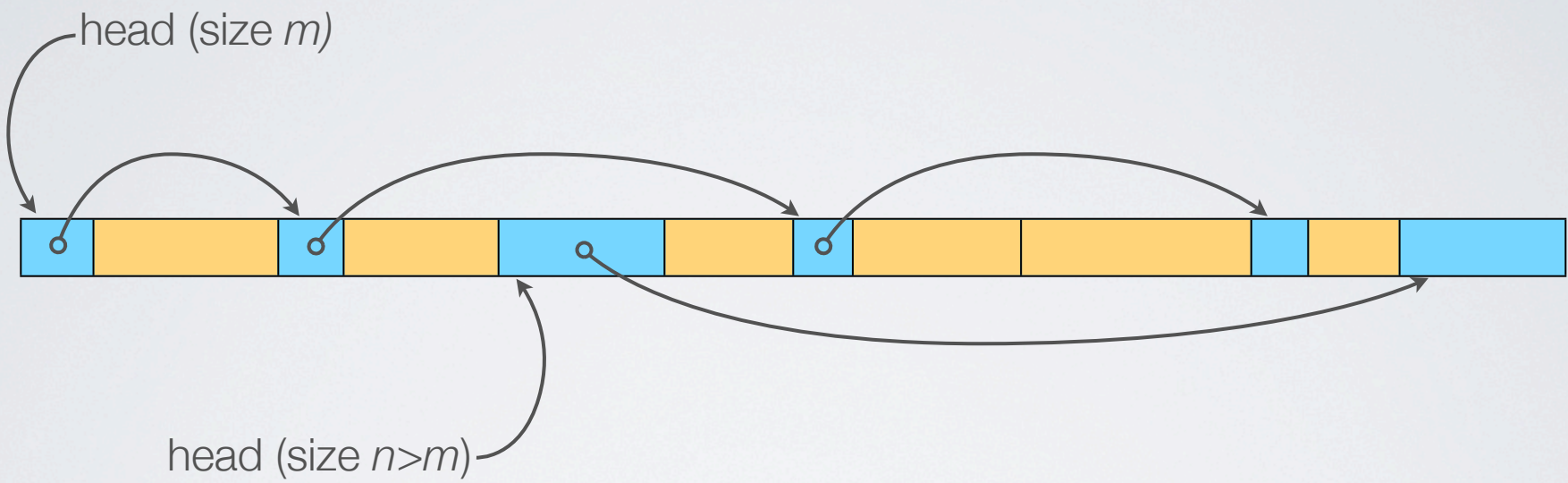
may be suitable when:

- many requests for the same size / set of sizes
- fast malloc & free is critical
- e.g., real time/embedded systems

can be used in conjunction
with other techniques
(i.e., hybrid approach)

(b) segregated **fits**

maintain **separate explicit lists** for each size class



malloc(k):

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

free:

- free and coalesce
- add coalesced block to appropriate list

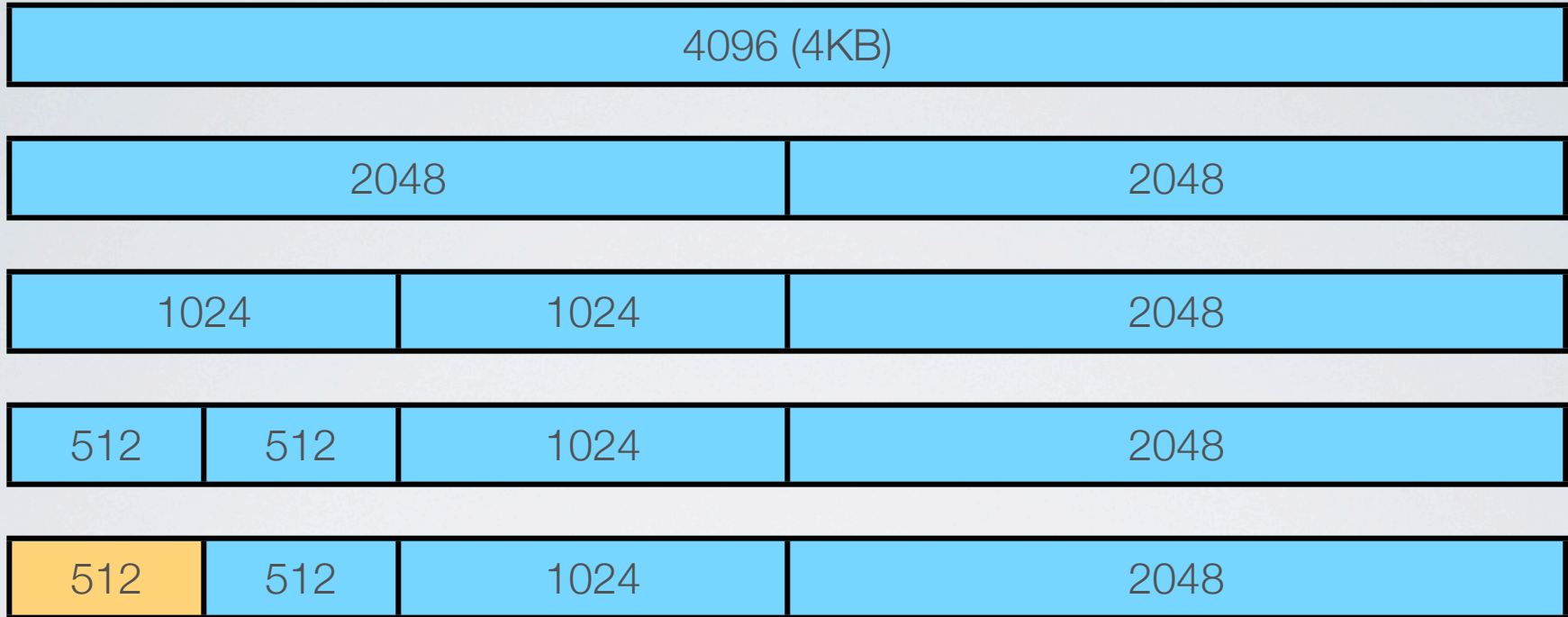
good balance between
complexity, speed, utilization

(c) buddy system

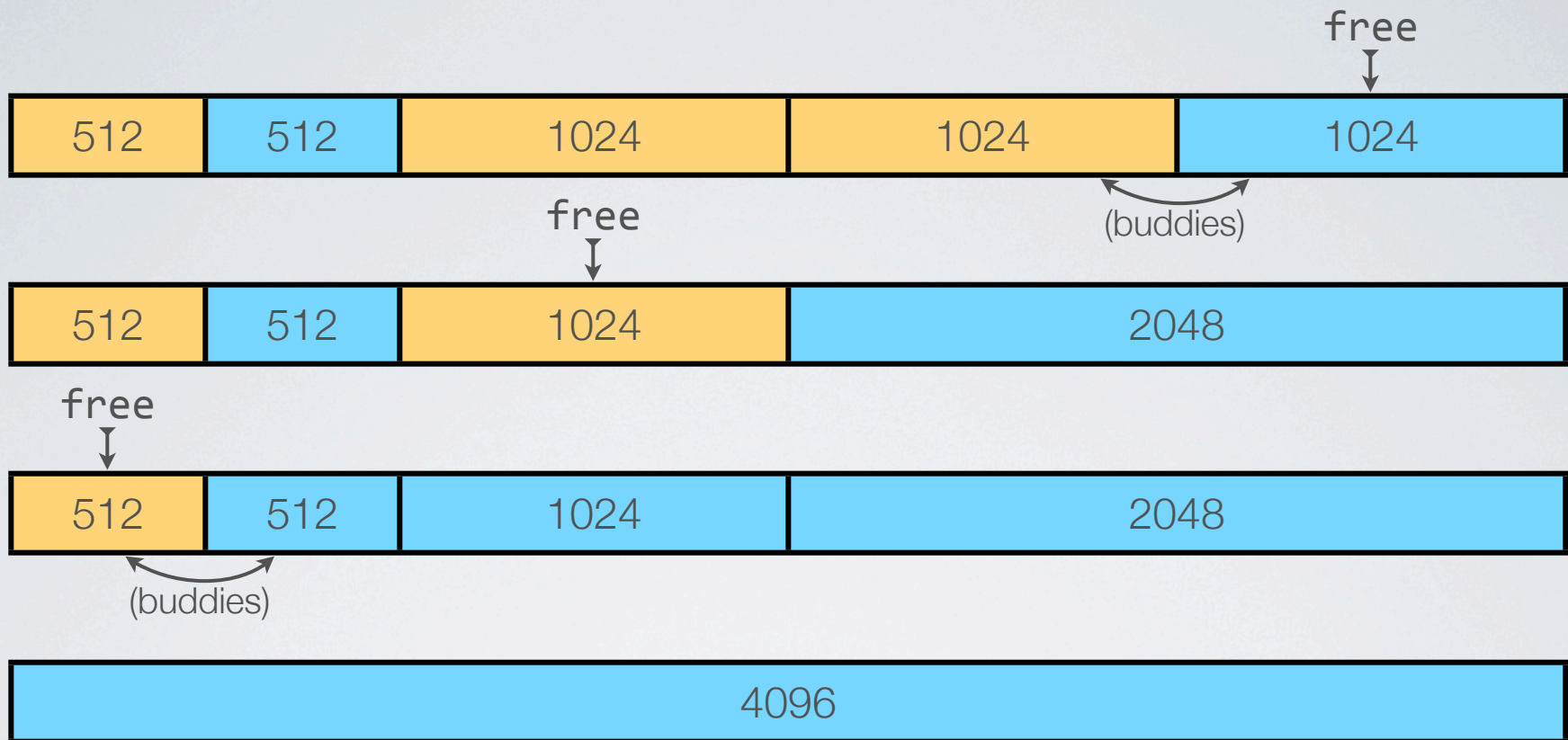
blocks can only be split into
two “buddy” sub-blocks with
pre-set sizes

a block can only be merged
with its buddy

malloc(450)



e.g., binary buddies

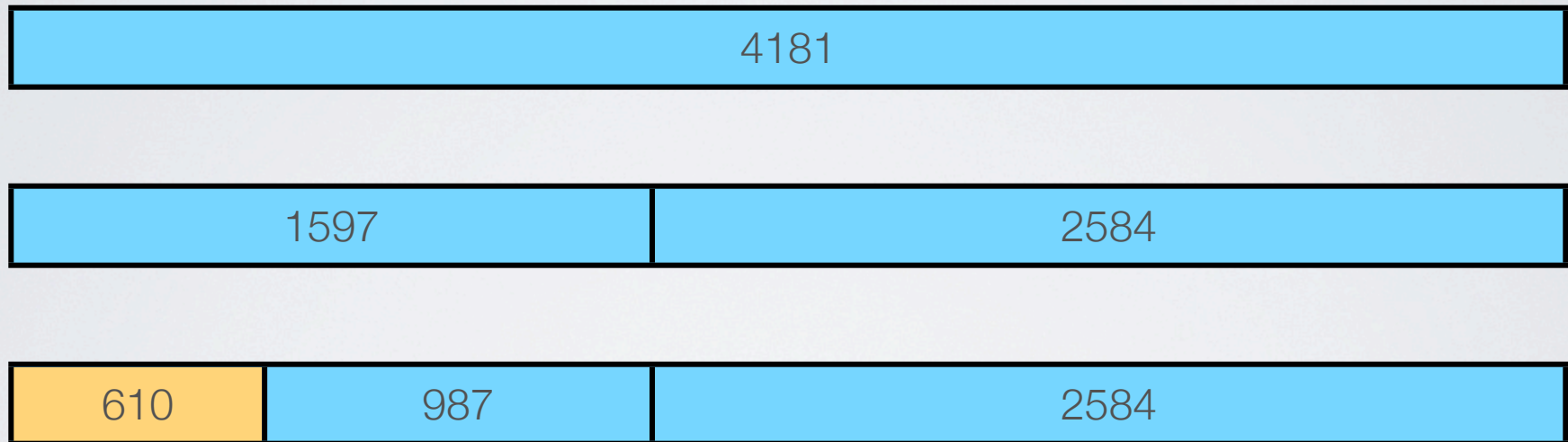


e.g., binary buddies

Fibonacci sequence:

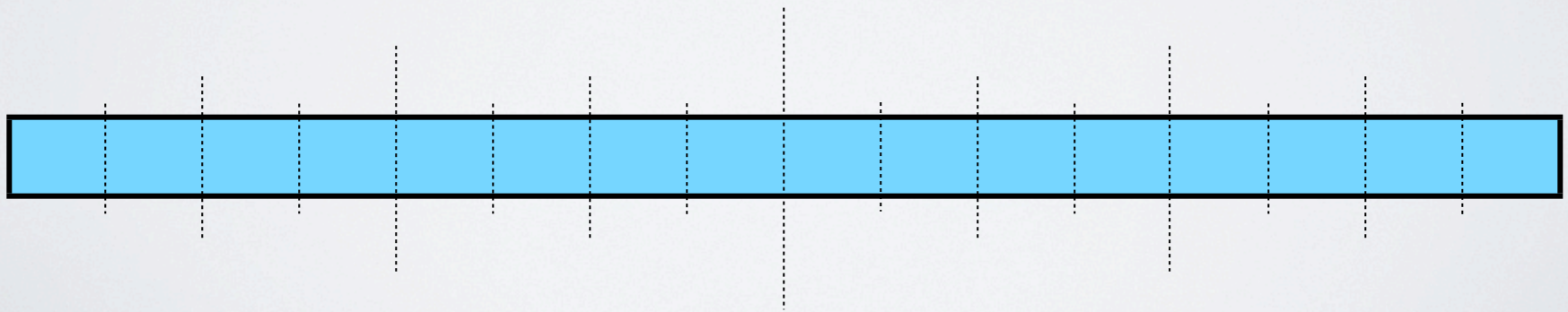
1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, 2584, 4181

`malloc(450)`



e.g., Fibonacci buddies

all block boundaries are
known in advance!



very little block overhead:

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

in practice, internal
fragmentation worse than
segregated fits

strategy 4: *

data structure
& optimization problem!

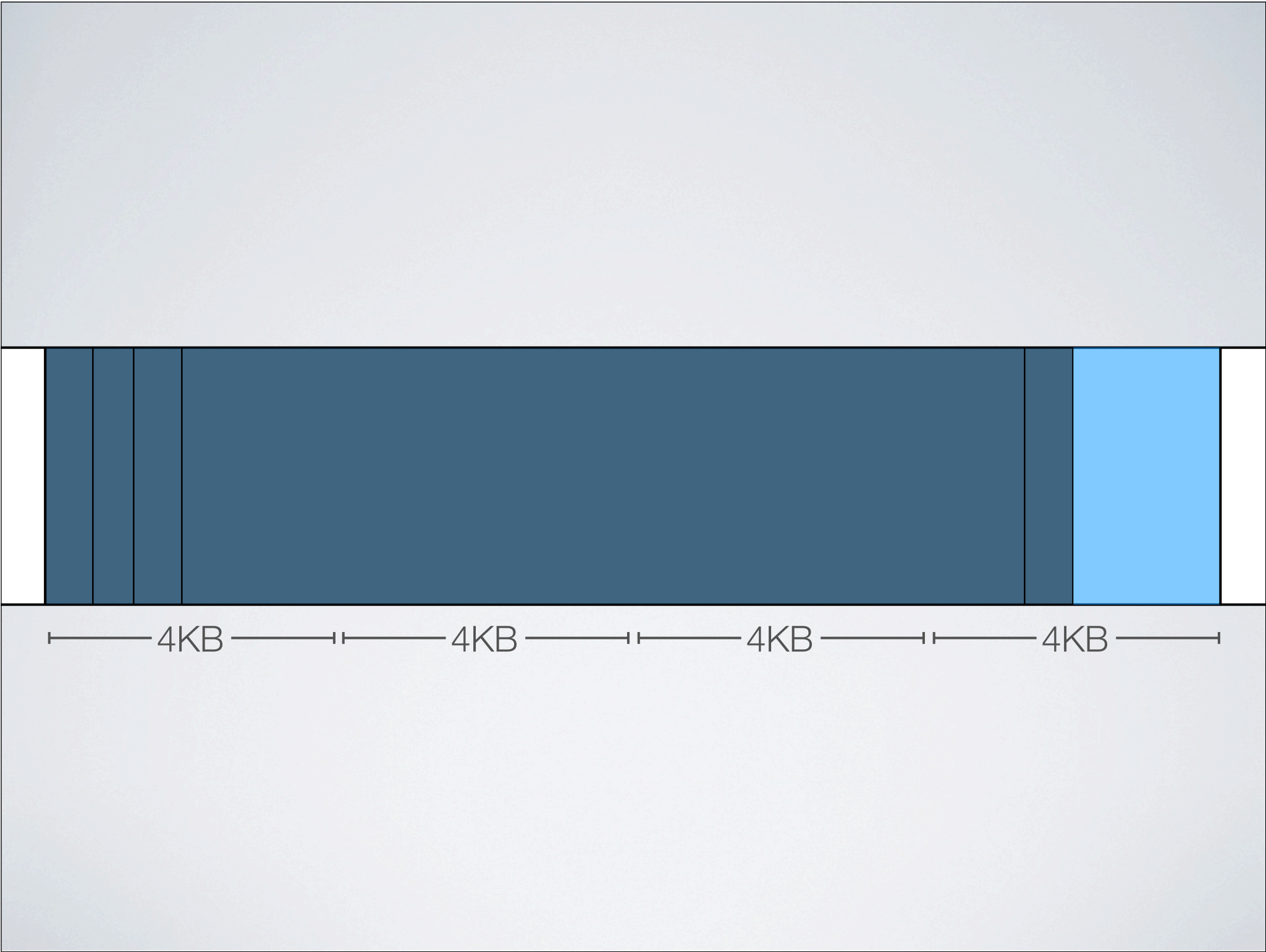
binary trees (balanced),
skip lists,
etc.

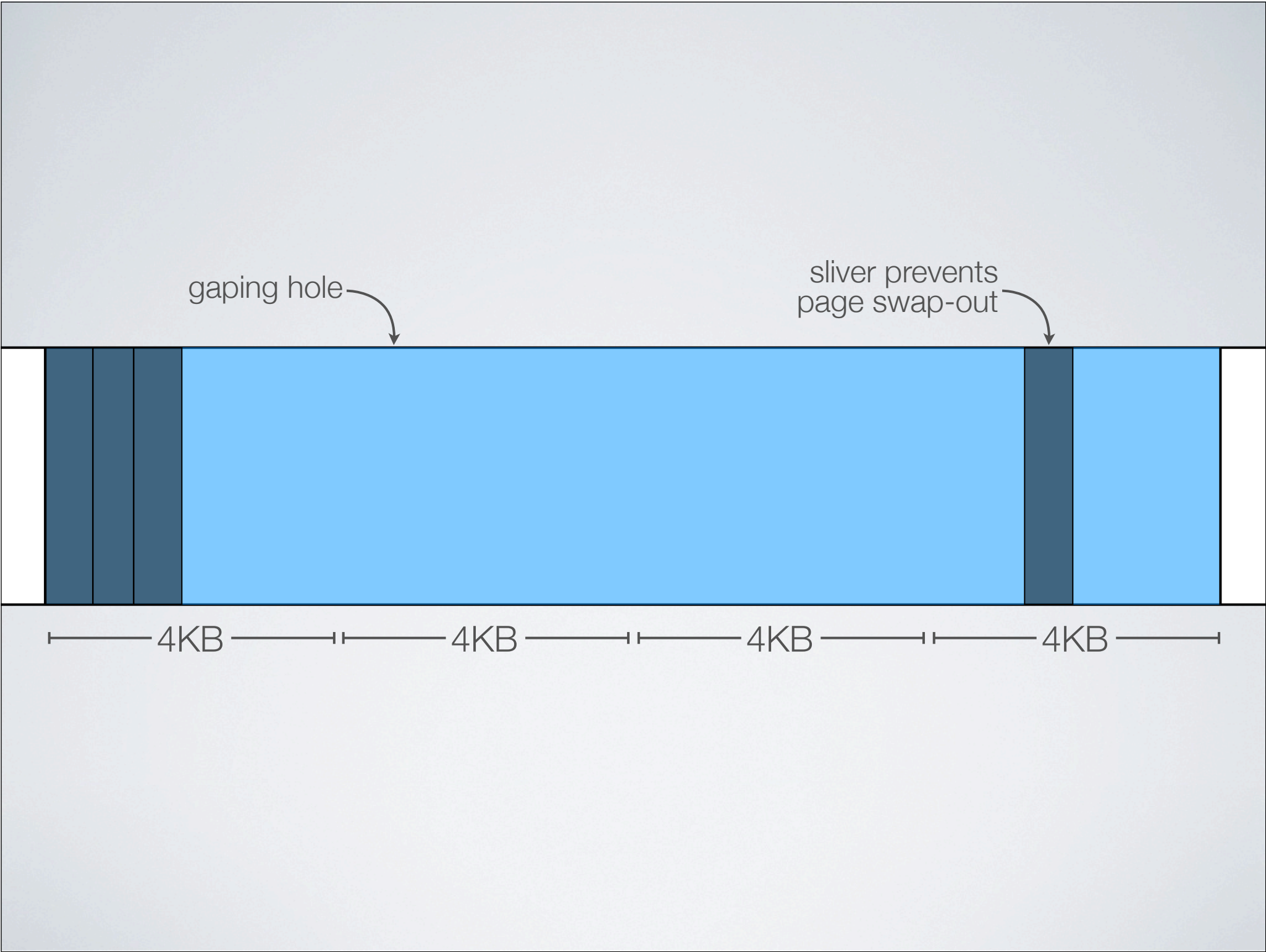
need to carefully balance
throughput & utilization!

</malloc strategies>

one more thing ...

how to deal with really large
malloc requests?
(e.g., $>$ page size)



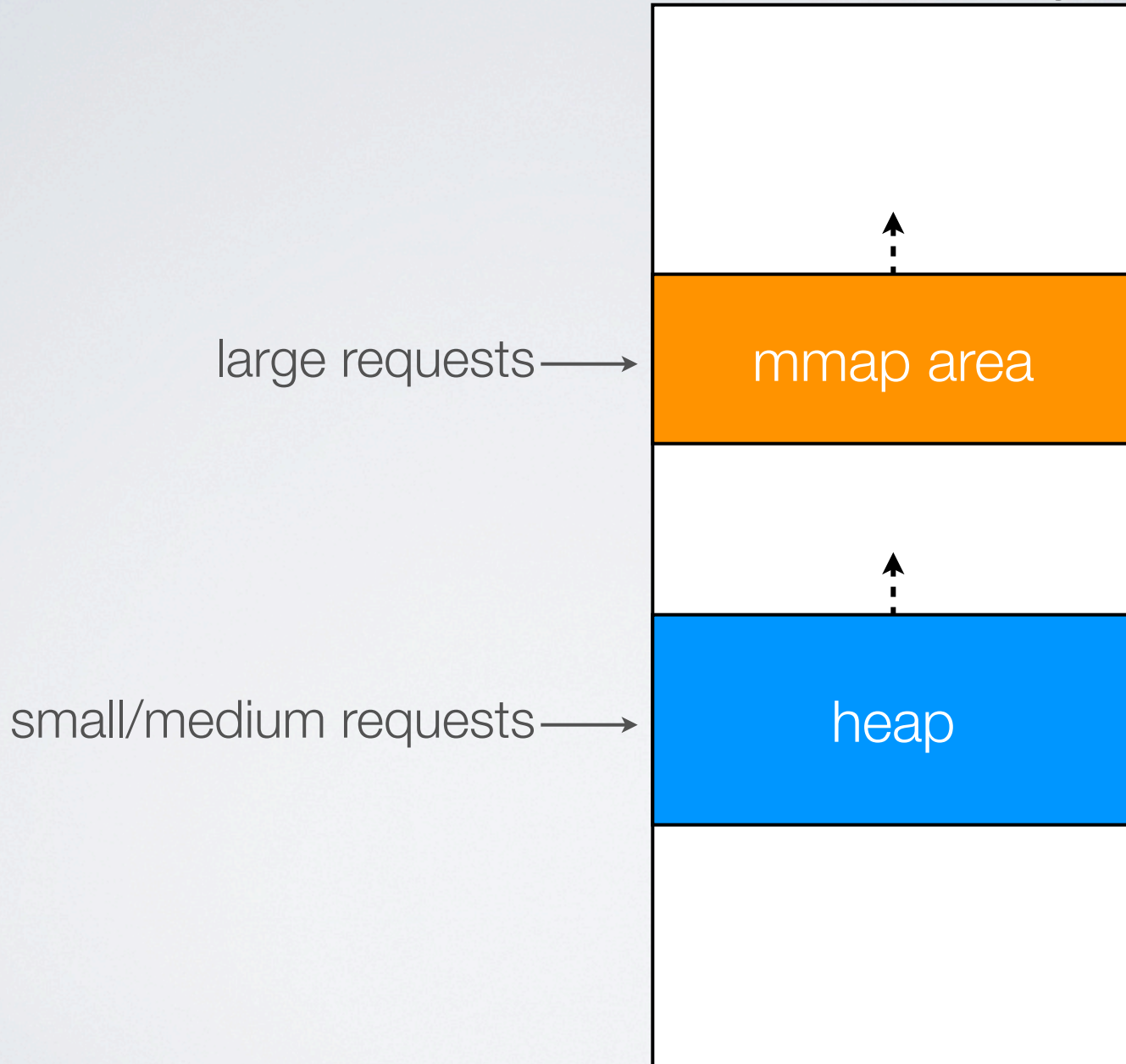


ideally, allocate super large
blocks outside heap space

`mmap` syscall maps page-aligned block of memory into virtual address space

`munmap` to delete mapping

Virtual Memory



`mmap` is much less efficient
than manually carving out
heap space!

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