Memory Allocation I

CSE 351 Spring 2022

Instructor:

Ruth Anderson

Teaching Assistants:

Melissa Birchfield

Jacob Christy

Alena Dickmann

Kyrie Dowling

Ellis Haker

Maggie Jiang

Diya Joy

Anirudh Kumar

Jim Limprasert

Armin Magness

Hamsa Shankar

Dara Stotland

Jeffery Tian

Assaf Vayner

Tom Wu

Angela Xu

Effie Zheng

Adapted from
https://xkcd.com/1093/

Relevant Course Information

❖ hw21 due Friday (5/20)

❖ Lab 4 due Friday (5/20)

   ▪ Cache parameter puzzles and code optimizations

❖ hw22 due Monday (5/23)

❖ hw24 due Wednesday (5/25)

❖ Lab 5 (on Mem Alloc) coming soon!

   ▪ Will want to look at readings for next week to get started

   ▪ Can be submitted at most ONE day late. (Sun 6/05)

Roadmap

C:

```
car *c = malloc(sizeof(car));
c->miles = 100;
c->gals = 17;
float mpg = get_mpg(c);
free(c);
```

Java:

```
Car c = new Car();
c.setMiles(100);
c.setGals(17);
float mpg =
      c.getMPG();
```

Memory & data
Integers & floats
x86 assembly
Procedures & stacks
Executables
Arrays & structs
Memory & caches
Processes
Virtual memory
Memory allocation
Java vs. C

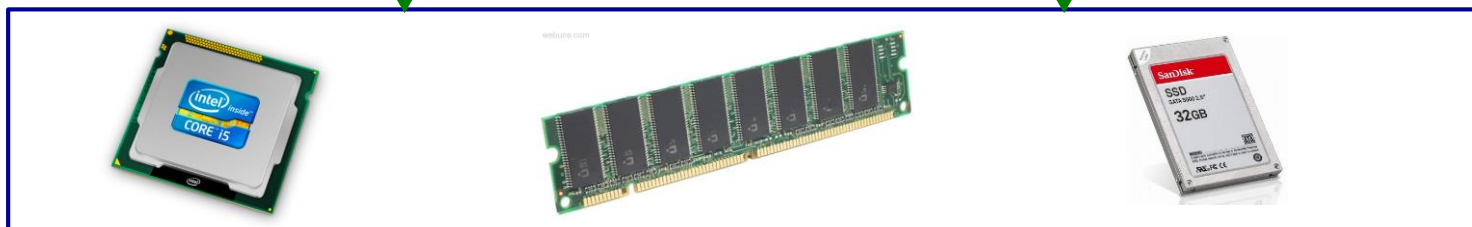Assembly language:

```
get_mpg:
    pushq    %rbp
    movq     %rsp, %rbp
    ...
    popq     %rbp
    ret
```
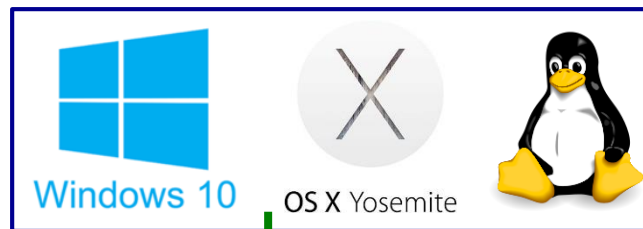
OS:

Machine code:

```
0111010000011000
100011010000010000000010
1000100111000010
110000011111101000011111
```

Computer system:

3

Reading Review

❖ Terminology:

- Dynamically-allocated data: malloc, free

- Allocators: implicit vs. explicit allocators, heap blocks, implicit vs. explicit free lists

- Heap fragmentation: internal vs. external

Multiple Ways to Store Program Data

❖ Static global data

- *Fixed size* at compile-time

- Entire *lifetime of the program* (loaded from executable)

- Portion is read-only (*e.g.* string literals)

❖ Stack-allocated data

- Local/temporary variables

  • *Can* be dynamically sized (in some versions of C)

- *Known lifetime* (deallocated on `return`)

❖ **Dynamic (heap) data**

- Size known only at runtime (*i.e.* based on user-input)

- Lifetime known only at runtime (long-lived data structures)

```
int array[1024];

int* foo(int n) {
  int tmp;
  int local_array[n];

  int* dyn =
    (int*)malloc(n*sizeof(int));
  return dyn;
}
```
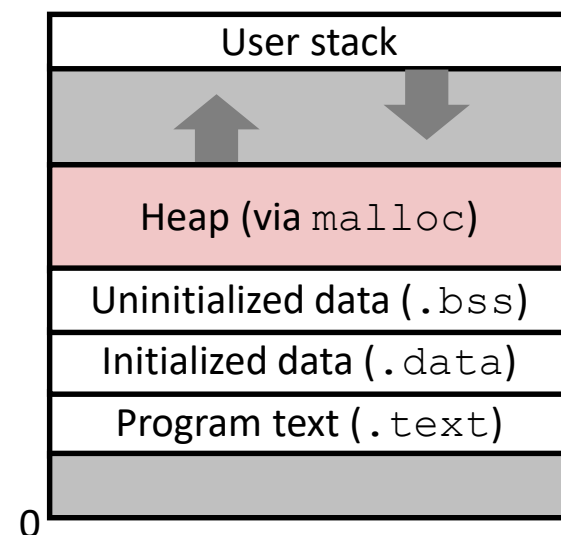
Memory Allocation

- ❖ **Dynamic memory allocation**
  - ▪ **Introduction and goals**
  - ▪ **Allocation and deallocation (free)**
  - ▪ **Fragmentation**
- ❖ Explicit allocation implementation
  - ▪ Implicit free lists
  - ▪ Explicit free lists (Lab 5)
  - ▪ Segregated free lists
- ❖ Implicit deallocation: garbage collection
- ❖ Common memory-related bugs in C

Dynamic Memory Allocation (Review)

❖ Programmers use dynamic memory allocators to acquire virtual memory at run time

- For data structures whose size (or lifetime) is known only at runtime

- Manage the heap of a process' virtual memory:

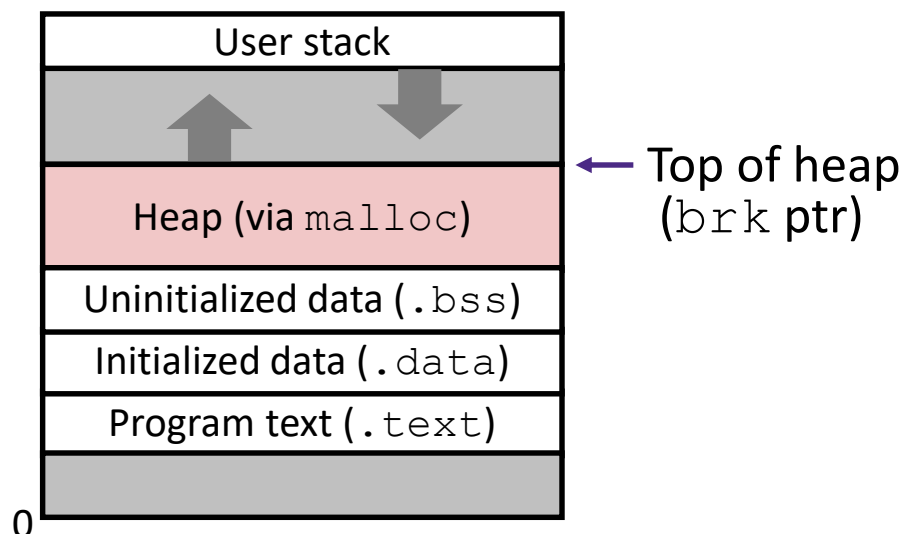| User stack |
| --- |
| |
| Heap (via `malloc`) |
| Uninitialized data (`.bss`) |
| Initialized data (`.data`) |
| Program text (`.text`) |
| |

0

❖ Types of allocators

- **Explicit allocator:** programmer allocates and frees space
  - Example: `malloc` and `free` in C

- **Implicit allocator:** programmer only allocates space (no free)
  - Example: garbage collection in Java, Caml, and Lisp

7

Dynamic Memory Allocation

❖ Allocator organizes heap as a collection of variable-sized *blocks*, which are either *allocated* or *free*

  ▪ Allocator requests pages in the heap region; virtual memory hardware and OS kernel allocate these pages to the process

  ▪ Application objects are typically smaller than pages, so the allocator manages blocks *within* pages

    • (Larger objects handled too; ignored here)

User stack
↑ ↓
Heap (via `malloc`)
Uninitialized data (`.bss`)
Initialized data (`.data`)
Program text (`.text`)

← Top of heap (`brk` ptr)

0

Allocating Memory in C (Review)

❖ Need to `#include <stdlib.h>`

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

- Allocates a continuous block of `size` bytes of uninitialized memory

- Returns a pointer to the beginning of the allocated block; NULL indicates failed request

  - Typically aligned to an 8-byte (x86) or 16-byte (x86-64) boundary

  - Returns `NULL` if allocation failed (also sets `errno`) or `size==0`

- Different blocks not necessarily adjacent

❖ Good practices:

- `ptr = (int*) malloc(n*sizeof(int));`

  - `sizeof` makes code more portable

  - `void*` is implicitly cast into any pointer type; explicit typecast will help you catch coding errors when pointer types don't match

Allocating Memory in C (Review)

❖ Need to `#include <stdlib.h>`

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

  ▪ Allocates a continuous block of `size` bytes of uninitialized memory

  ▪ Returns a pointer to the beginning of the allocated block; `NULL` indicates failed request

    • Typically aligned to an 8-byte (x86) or 16-byte (x86-64) boundary

    • Returns `NULL` if allocation failed (also sets `errno`) or `size==0`

  ▪ Different blocks not necessarily adjacent

❖ Related functions:

  ▪ **void*** `calloc(`**size_t** `nitems, `**size_t** `size)`

    • "Zeros out" allocated block

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

    • Changes the size of a previously allocated block (if possible)

  ▪ **void*** `sbrk(`**intptr_t** `increment)`

    • Used internally by allocators to grow or shrink the heap

Freeing Memory in C (Review)

❖ Need to `#include <stdlib.h>`

❖ **void** `free`(**void*** `p`)

  ▪ Releases whole block pointed to by `p` to the pool of available memory

  ▪ Pointer `p` must be the address *originally* returned by `m/c/realloc` (*i.e.* beginning of the block), otherwise system exception raised

  ▪ Don't call `free` on a block that has already been released
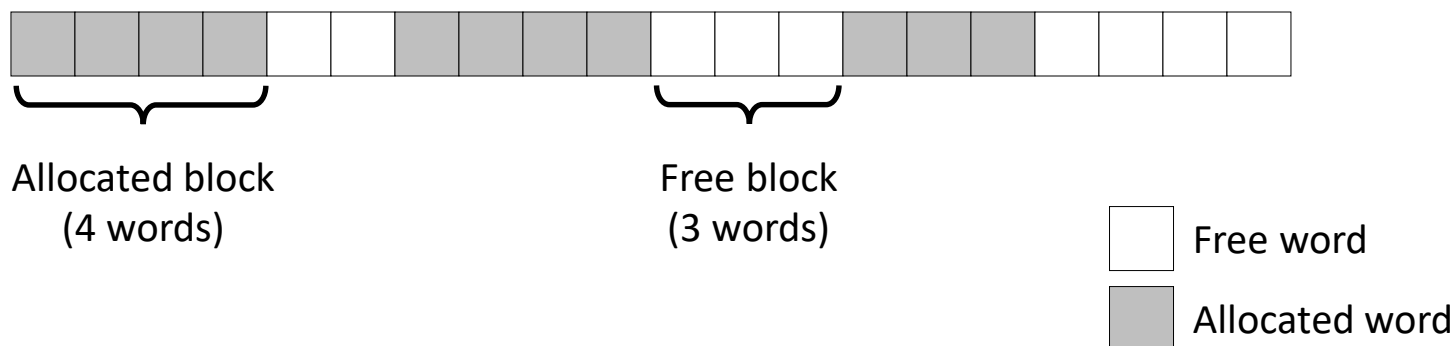
Memory Allocation Example in C

```c
void foo(int n, int m) {
  int i, *p;
  p = (int*) malloc(n*sizeof(int));   /* allocate block of n ints */
  if (p == NULL) {                    /* check for allocation error */
    perror("malloc");
    exit(0);
  }
  for (i=0; i<n; i++)                 /* initialize int array */
    p[i] = i;

                                      /* add space for m ints to end of p block */
  p = (int*) realloc(p,(n+m)*sizeof(int));
  if (p == NULL) {                    /* check for allocation error */
    perror("realloc");
    exit(0);
  }
  for (i=n; i < n+m; i++)             /* initialize new spaces */
    p[i] = i;
  for (i=0; i<n+m; i++)               /* print new array */
    printf("%d\n", p[i]);
  free(p);                           /* free p */
}
```
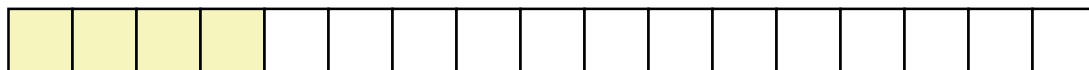
Notation

☐ = 1 word = 8 bytes

❖ We will draw memory divided into *words*

▪ Each *word* is 64 bits = 8 bytes

▪ Allocations will be in sizes that are a multiple of words (*i.e.* multiples of 8 bytes)

▪ Book and old videos still use 4-byte *word*

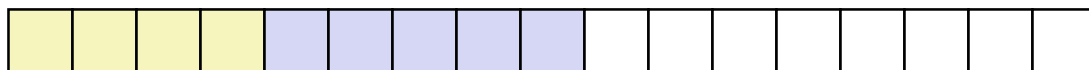• Holdover from 32-bit version of textbook ☹

Allocated block
(4 words)

Free block
(3 words)

☐ Free word

▨ Allocated word
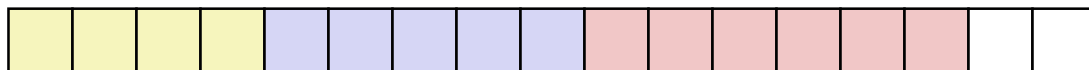
Allocation Example

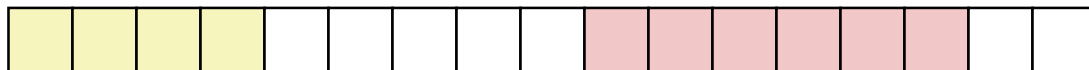☐ = 8-byte word

`p1 = malloc(32)`

`p2 = malloc(40)`

`p3 = malloc(48)`

`free(p2)`

`p4 = malloc(16)`

Implementation Interface (Review)

❖ **Applications**

- Can issue arbitrary sequence of `malloc` and `free` requests
- Must never access memory not currently allocated
- Must never free memory not currently allocated
  - Also must only use `free` with previously `malloc`'ed blocks

❖ **Allocators**

- Can't control number or size of allocated blocks
- Must respond immediately to `malloc`
- Must allocate blocks from free memory
- Must align blocks so they satisfy all alignment requirements
- Can't move the allocated blocks

Performance Goals (Review)

❖ **Goals:** Given some sequence of `malloc` and `free` requests $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$, maximize throughput and peak memory utilization

- These goals are often conflicting

1) Throughput

- Number of completed requests per unit time
- <u>Example</u>:
  - If 5,000 `malloc` calls and 5,000 `free` calls completed in 10 seconds, then throughput is 1,000 operations/second

Performance Goals

❖ <u>Definition</u>: *Aggregate payload P_k*

- ▪ `malloc(p)` results in a block with a *payload* of `p` bytes
- ▪ After request R_k has completed, the *aggregate payload P_k* is the sum of currently allocated payloads

❖ <u>Definition</u>: *Current heap size H_k*

- ▪ Assume H_k is monotonically non-decreasing
  - • Allocator can increase size of heap using `sbrk`

2) Peak Memory Utilization

- ▪ Defined as $U_k = (\max_{i \leq k} P_i)/H_k$ after k+1 requests
- ▪ Goal: maximize utilization for a sequence of requests
- ▪ Why is this hard? And what happens to throughput?

Fragmentation (Review)

❖ Poor memory utilization is caused by *fragmentation*

- Sections of memory are not used to store anything useful, but cannot satisfy allocation requests

- Two types: *internal* and *external*

❖ **Recall:** Fragmentation in structs

- Internal fragmentation was wasted space *inside* of the struct (between fields) due to alignment

- External fragmentation was wasted space *between* struct instances (*e.g.* in an array) due to alignment

❖ Now referring to wasted space in the heap *inside* or *between* allocated blocks

Internal Fragmentation

❖ For a given block, *internal fragmentation* occurs if payload is smaller than the block

block

Internal fragmentation → | payload | ← Internal fragmentation
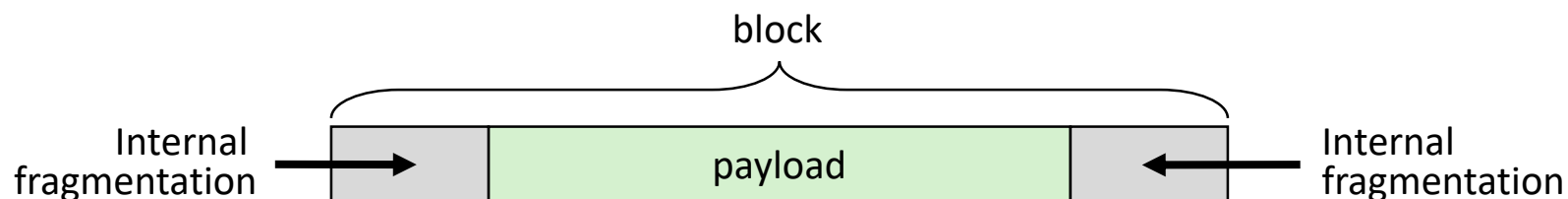
❖ **Causes:**
  ▪ Padding for alignment purposes
  ▪ Overhead of maintaining heap data structures (inside block, outside payload)
  ▪ Explicit policy decisions (*e.g.* return a big block to satisfy a small request)

❖ Easy to measure because only depends on past requests
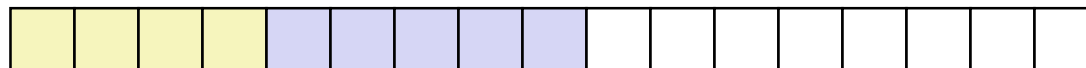
External Fragmentation

☐ = 8-byte word

❖ For the heap, *external fragmentation* occurs when allocation/free pattern leaves "holes" between blocks
  ▪ That is, the aggregate payload is non-continuous
  ▪ Can cause situations where there is enough aggregate heap memory to satisfy request, but no single free block is large enough
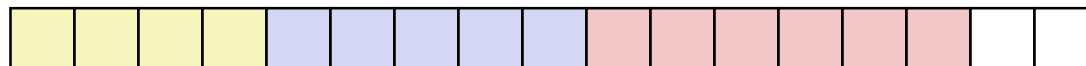
`p1 = malloc(32)`

`p2 = malloc(40)`

`p3 = malloc(48)`

`free(p2)`

`p4 = malloc(48)`   *Oh no! (What would happen now?)*

❖ Don't know what future requests will be
 ▪ Difficult to impossible to know if past placements will become problematic

Polling Question

- ❖ Which of the following statements is FALSE?
  - ■ Vote in Ed Lessons

  A. **Temporary arrays should *not* be allocated on the Heap**

  B. **`malloc` returns an address of a block that is filled with random data**

  C. **Peak memory utilization is a measure of both internal and external fragmentation**

 D. **An allocation failure will cause your program to stop**

Implementation Issues

❖ How do we know how much memory to free given just a pointer?

❖ How do we keep track of the free blocks?

❖ How do we pick a block to use for allocation (when many might fit)?

❖ What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?

❖ How do we reinsert a freed block into the heap?
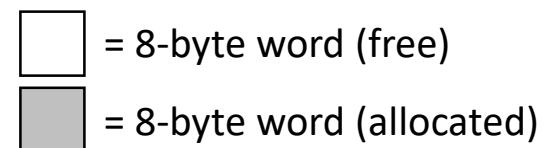
Knowing How Much to Free

☐ = 8-byte word (free)

▨ = 8-byte word (allocated)
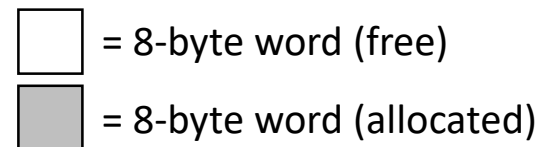
❖ Standard method

▪ Keep the length of a block in the word preceding the data

• This word is often called the *header field* or *header*

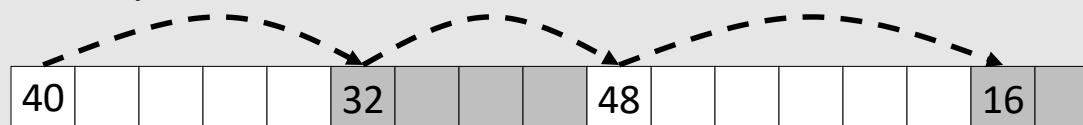▪ Requires an extra word for every allocated block

p0

p0 = malloc(32)

40

block size   **data**

free(p0)
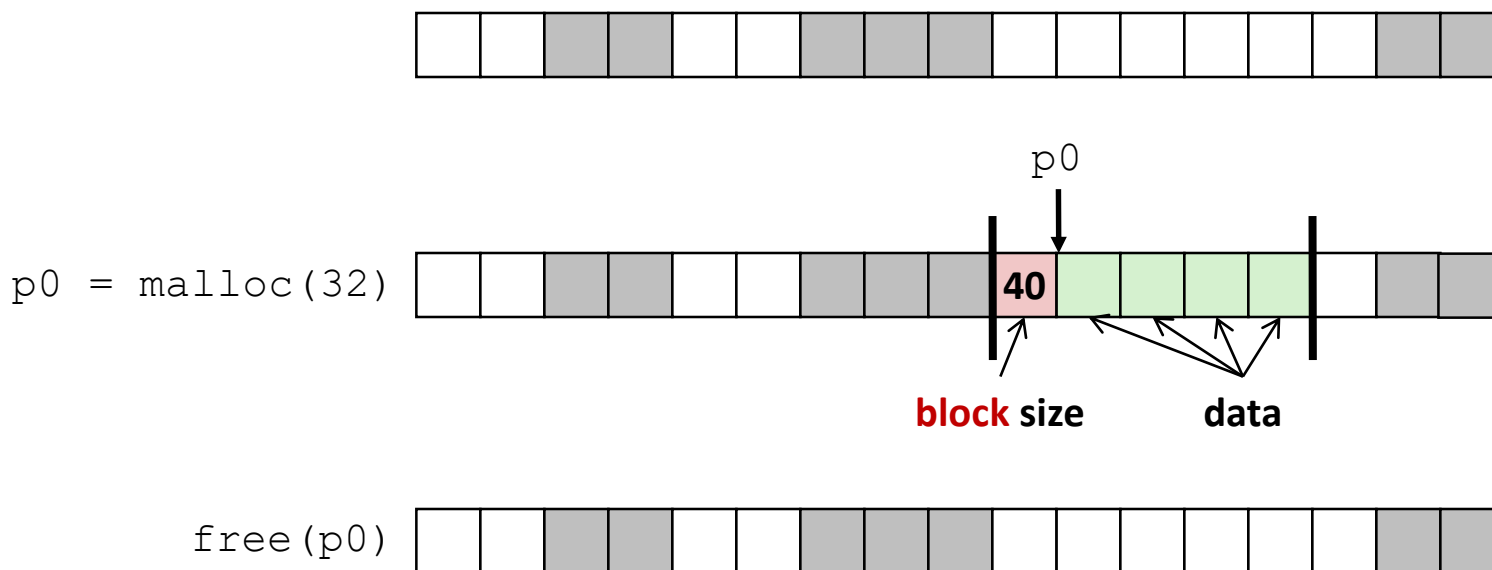
Keeping Track of Free Blocks

□ = 8-byte word (free)
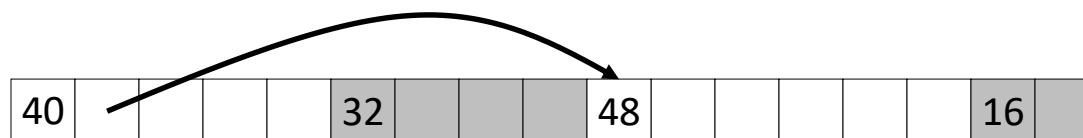
▦ = 8-byte word (allocated)

1) *Implicit free list* using length – links <u>all</u> blocks using math
   - No actual pointers, and must check each block if allocated or free

2) *Explicit free list* among <u>only the free blocks</u>, using pointers

3) *Segregated free list*
   - Different free lists for different size "classes"

4) *Blocks sorted by size*
   - Can use a balanced binary tree (*e.g.* red-black tree) with pointers within each free block, and the length used as a key

Implicit Free Lists

e.g. with 8-byte alignment, possible values for size:
  00001000 = 8 bytes
  00010000 = 16 bytes
  00011000 = 24 bytes
  ...

❖ For each block we need: `size`, **is-allocated?**

  ▪ Could store using two words, but wasteful

❖ Standard trick

  ▪ If blocks are aligned, some low-order bits of $size$ are always 0

  ▪ Use lowest bit as an allocated/free flag (fine as long as aligning to $K>1$)

  ▪ When reading $size$, must remember to mask out this bit!

8 bytes

Format of allocated and free blocks:

size	a
payload	
optional padding	
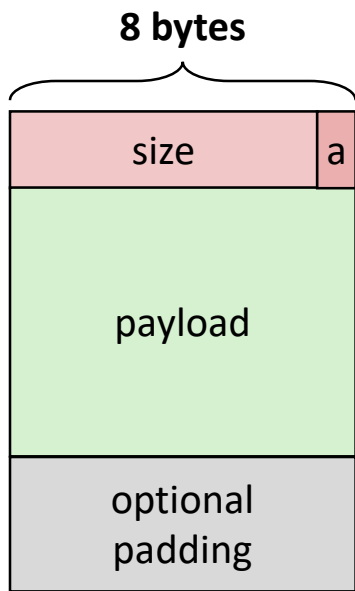
a = 1: allocated block
a = 0: free block

size: block size (in bytes)

payload: application data (allocated blocks only)

If `x` is first word (header):

```
x = size | a;

a = x & 1;

size = x & ~1;
```

Header Questions

❖ How many "flags" can we fit in our header if our allocator uses 16-byte alignment?

❖ If we placed a new "flag" in the second least significant bit, write out a C expression that will extract this new flag from `header`