The Hardware/Software Interface

Memory Allocation I

Instructors:

Amber Hu, Justin Hsia

Teaching Assistants:

Anthony Mangus
Grace Zhou
Jiuyang Lyu
Kurt Gu
Mendel Carroll
Naama Amiel
Rose Maresh
Violet Monserate
Divya Ramu
Jessie Sun
Kanishka Singh
Liander Rainbolt
Ming Yan
Pollux Chen
Soham Bhosale

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

Relevant Course Information

- HW 18 due tonight
- HW 19 due Friday (11/14)
  - Lab 4 preparation!
- HW 20 due Monday (11/17)

- Lab 4 due next Friday (11/21)
  - Section tomorrow intended to help prepare you for Lab 4

Mid-Quarter Feedback Survey Debrief

- Midterm was difficult
  - We will intentionally design the final to not be as difficult
- More practice with exam-style questions
  - Make sure you go to section for exam-style questions
  - Lab synthesis questions are short-answer
  - Look at past exams, and ask questions on Ed or OH about the solutions
- More answer explanations on Ed readings and homeworks
- Lectures are too fast
  - Especially Amber
- Scheduling office hours to make it more accessible
  - E.g. more online, weekend, weekday evening

Aside: Growth vs. Fixed Mindset

- Students can be thought of as having either a “growth” mindset or a “fixed” mindset (based on research by Prof. Carol Dweck)
  - “In a **fixed mindset** students believe their basic abilities, their intelligence, their talents, are just fixed traits. They have a certain amount and that's that, and then their goal becomes to look smart all the time and never look dumb.”
  - “In a **growth mindset** students understand that their talents and abilities can be developed through effort, good teaching and persistence. They don't necessarily think everyone's the same or anyone can be Einstein, but they believe everyone can get smarter if they work at it.”

- Midterm clobber policy demonstrates how we want you to adopt a **growth mindset**!

Lecture Outline (1/5)

- **Cache Performance, Revisited**
- Dynamically Allocated Memory in C
- Dynamic Memory Allocators
- Heap Fragmentation
- Heap Implementation Basics

AMAT, Revisited

- *Average Memory Access Time* (AMAT): average time to access memory considering both hits and misses

  AMAT = Hit time + Miss rate × Miss penalty

  (abbreviated AMAT = HT + MR × MP)

- We called this a *cache performance* ***metric***
  - This isn't the only metric we could have used!

Metrics in Computing

- Generally, folks care most about performance
  - Energy-efficiency is more important now since the plateau in 2004/2005
  - This is why we have so many specialized chips nowadays

- Really, this is just **efficiency** – making efficient use of the resources that we have
  - Performance: cycles/instruction, seconds/program
  - Energy efficiency: performance/watt
  - Memory: bytes/program, bytes/data structure

Metrics

- What do we do with metrics?
  - We tend to optimize along them!
  - Especially when jobs/funding depend on better performance along some metric
    - See all of Intel under "Moore's Law"
- Sometimes, strange incentives emerge
  - "Minimize the number of bugs on our dashboard"
    - Does it count if we make the bugs invisible?
  - "Make this faster for our demo in a week"
    - Shortcuts might hurt performance at scale
  - "Minimize our average memory access time"
    - What if we add *more* memory accesses that we know will hit?

Metrics and Success

- Success is *defined along metrics*
  - This affects how we measure and optimize

- Let's say that we choose **performance/program** or **performance/program set** (*i.e.*, benchmarks):
  1. Measure existing performance
  2. Come up with a bunch of optimizations that would improve performance
  3. Select a few to build into the "next version"

Metrics and Success

- Success is *defined along metrics*
  - This affects how we measure and optimize

- Let's say that we choose **profit/year** or **stock price**:
  - Success means earning more profit than last year
  - Improvement or optimizations might include:
    - Reduce expenses, cut staff
    - Sell more things or fancier things (*e.g.*, in-app purchases)
    - Make people pay monthly for things they could get for free
    - Increase advertising revenue:


The New York Times

Whistle-Blower Says Facebook 'Chooses Profits Over Safety'

Frances Haugen, a Facebook product manager who left the company in May, revealed that she had provided internal documents to journalists and others.

Metrics and Success

- Success is *defined along metrics*
  - This affects how we measure and optimize

- Let's say that we choose **minoritized participation in computing**:
  - What does success/participation mean (and dangers)?
    - Women? BIPOC? All minoritized lumped together?
      - Might optimize for one group at the expense of others
    - Taking intro? Passing intro? Getting a degree? Getting a job?
      - Says nothing about retention or participation/decision-making level

Design Considerations

- Regardless of what we build, the way that we define success shapes the systems we build
  - Choose your metrics carefully
  - There's more to choose from than performance (*e.g.,* usability, access, simplicity, agency)
- Metrics are a "heading" (in the navigational sense)
  - Best to reevaluate from time to time in case you're off course or your destination changes

Discussion Questions

- Discuss the following question(s) in groups of 3-4 students
  - I will call on a few groups afterwards so please be prepared to share out
  - Be respectful of others' opinions and experiences

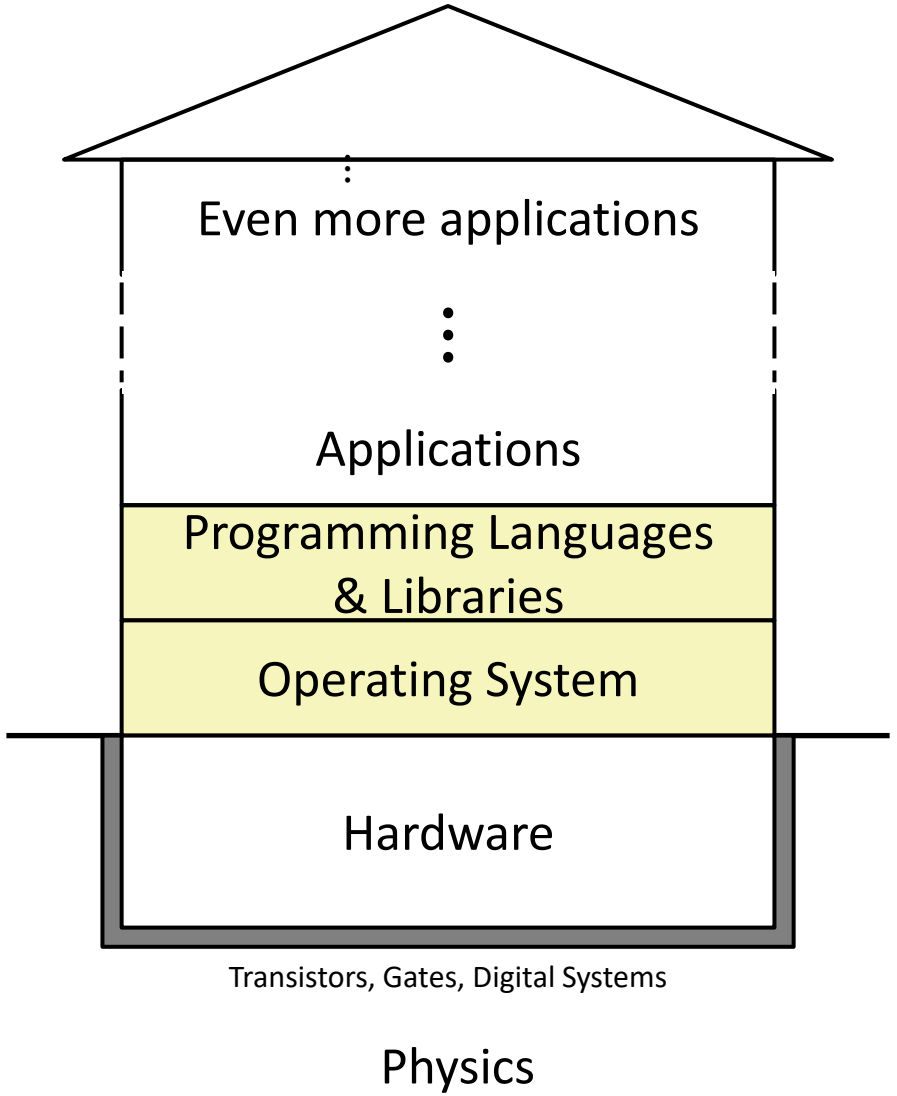
- Let's say your (main) metric for college is to get a 4.0 overall GPA.
  - What are some potential unintended consequences of this metric?

  - What are some other potential metrics you could use for college?

House of Computing Check-In

- Topic Group 3: **Scale & Coherence**
  - Caches, **Memory Allocation**, Processes, Virtual Memory

- How do we maintain logical consistency in the face of more data and more processes?
  - How do we support data access, including dynamic requests, across multiple processes?
  - How do we support control flow both within many processes and things external to the computer?



Lecture Outline (2/5)

- Cache Performance, Revisited
- **Dynamically Allocated Memory in C**
- Dynamic Memory Allocators
- Heap Fragmentation
- Heap Implementation Basics

Multiple Ways to Store Program Data (Review)

- Static global data
  - *Fixed size* (hard-coded into executable)
  - *Long lifetime*: entirety of the program
- Stack-allocated data
  - Local/temporary variables
  - *"Short" 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 global_array[1024];

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

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

Allocating Memory in C: `malloc` (Review)

- Need to `#include <stdlib.h>`
- **`void*`** `malloc(`**`size_t`** `size)`
  - Allocates a continuous block of `size` bytes of uninitialized memory, though different allocations are not necessarily adjacent
  - Returns a pointer to the beginning of the allocated block
    - Typically aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
    - Returns `NULL` if allocation failed (also sets `errno`) or `size==0`
- Good practices:

```
ptr = (int*) malloc(n*sizeof(int));
if (!ptr) { ... }
```

  - `sizeof` makes code more portable; explicit cast warns of pointer mismatches
  - Error check return value

Allocating Memory in C: Others (Review)

- Need to `#include <stdlib.h>`
- **`void*`** `malloc(`**`size_t`** `size)`
  - Allocates a continuous block of `size` bytes of uninitialized memory, though different allocations are not necessarily adjacent
  - Returns a pointer to the beginning of the allocated block
    - Typically aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
    - Returns `NULL` if allocation failed (also sets `errno`) or `size==0`
- Other allocation 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)

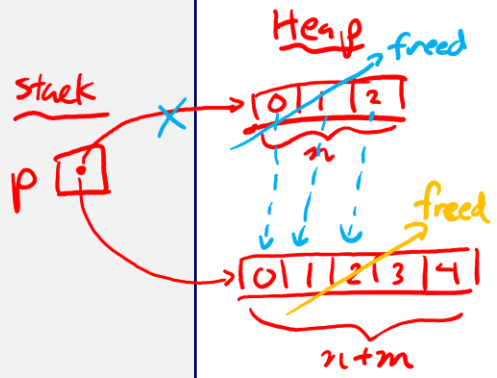
Deallocating Memory in C: `free` (Review)

- Need to `#include <stdlib.h>`
- **`void free(void* p)`** doesn't change the pointer! (now points to deallocated memory)
  - 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
  - No action occurs if you call `free(NULL)`

Memory Allocation Example in C

```
1  void foo(int n, int m) {
2    int i, *p;
3    p = (int*) malloc(n*sizeof(int));       /* allocate block of n ints */
4    if (p == NULL) {                        /* check for allocation error */
5      perror("malloc");
6      exit(0);
7    }
8    for (i=0; i<n; i++)                     /* initialize int array */
9      p[i] = i;
10
11   p = (int*) realloc(p,(n+m)*sizeof(int));   /* add space for m ints to end of p block
12 */
13   if (p == NULL) {                        /* check for allocation error */
14     perror("realloc");
15     exit(0);
16   }
17   for (i=n; i < n+m; i++)                 /* initialize new spaces */
18     p[i] = i;
19   for (i=0; i<n+m; i++)                   /* print new array */
20     printf("%d\n", p[i]);
21   free(p);                                /* free p */
   }
```

prints message related to errno

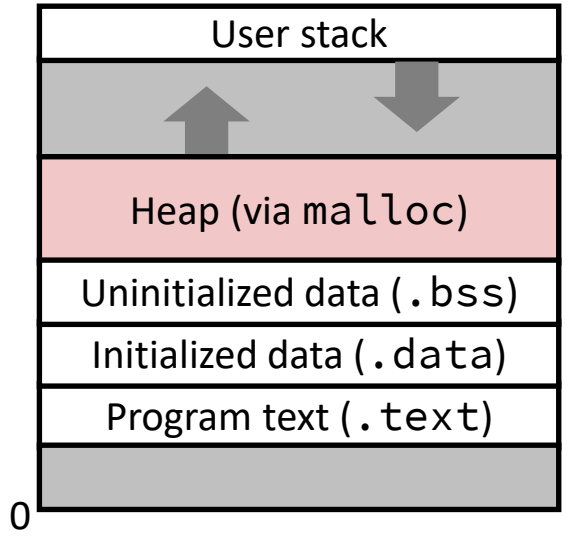


Lecture Outline (3/5)

- Cache Performance, Revisited
- Dynamically Allocated Memory in C
- **Dynamic Memory Allocators**
- Heap Fragmentation
- Heap Implementation Basics

Dynamic Memory Allocators: Types (Review)

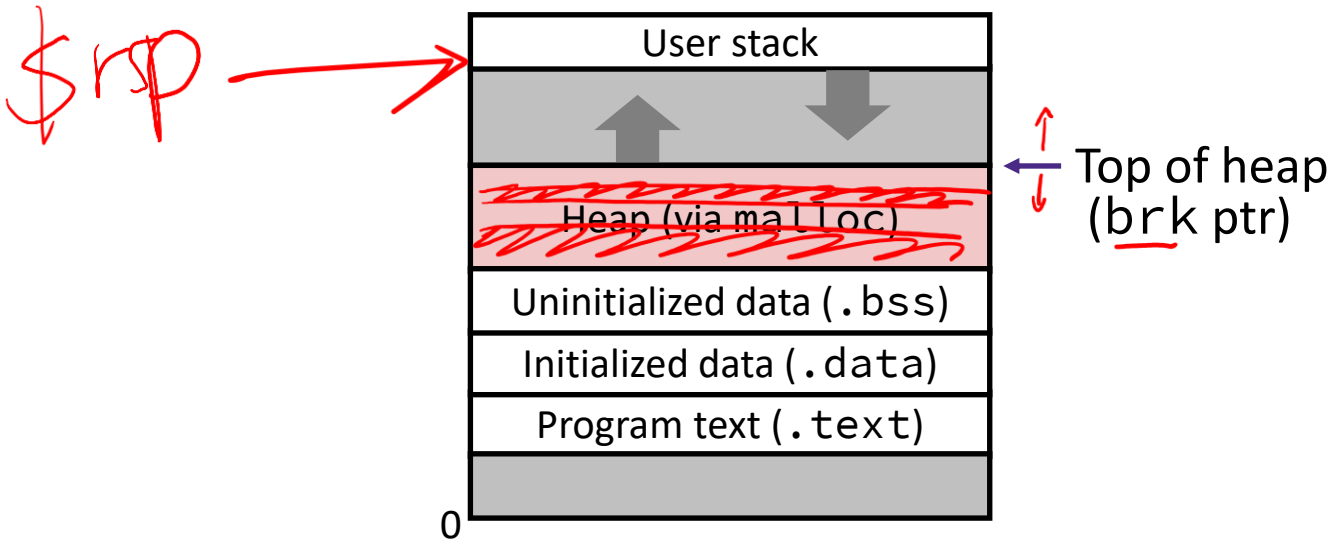
- Programmers use dynamic memory allocators to acquire memory at run time
  - For data structures whose size (or lifetime) is known only at runtime
  - Manage the heap of a process' memory:



- Types of allocators
  - ***Implicit* allocator:** programmer only allocates space (no free)
    - Example: garbage collection in Java, Caml, and Lisp
  - ***Explicit* allocator:** programmer allocates and frees space
    - Example: `malloc` and `free` in C

Dynamic Memory Allocators: Blocks (Review)

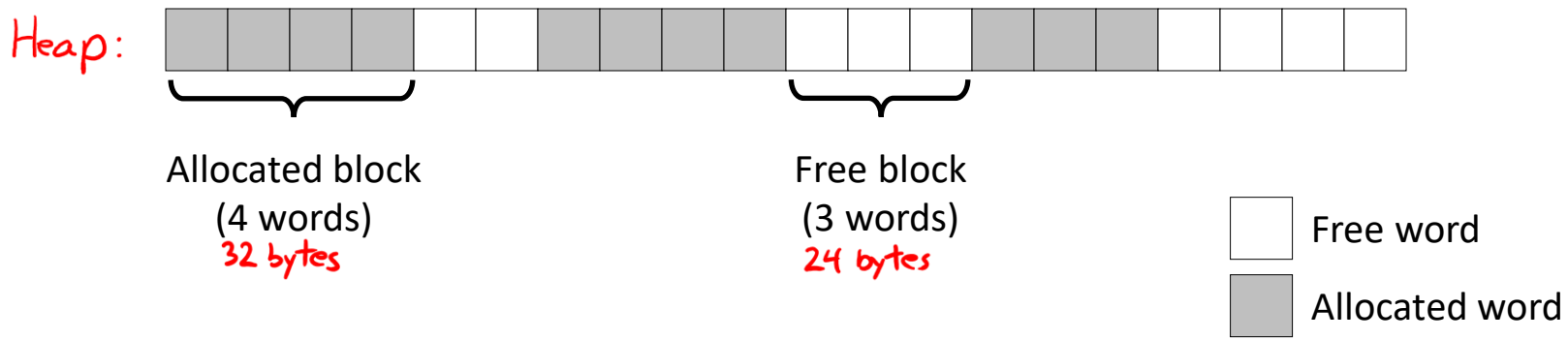
- Allocator organizes heap as a collection of variable-sized *heap blocks*, which are either *allocated* or *free*
  - What happens if we run out of heap space?
    - Ask the Operating System for more memory and increment brk!



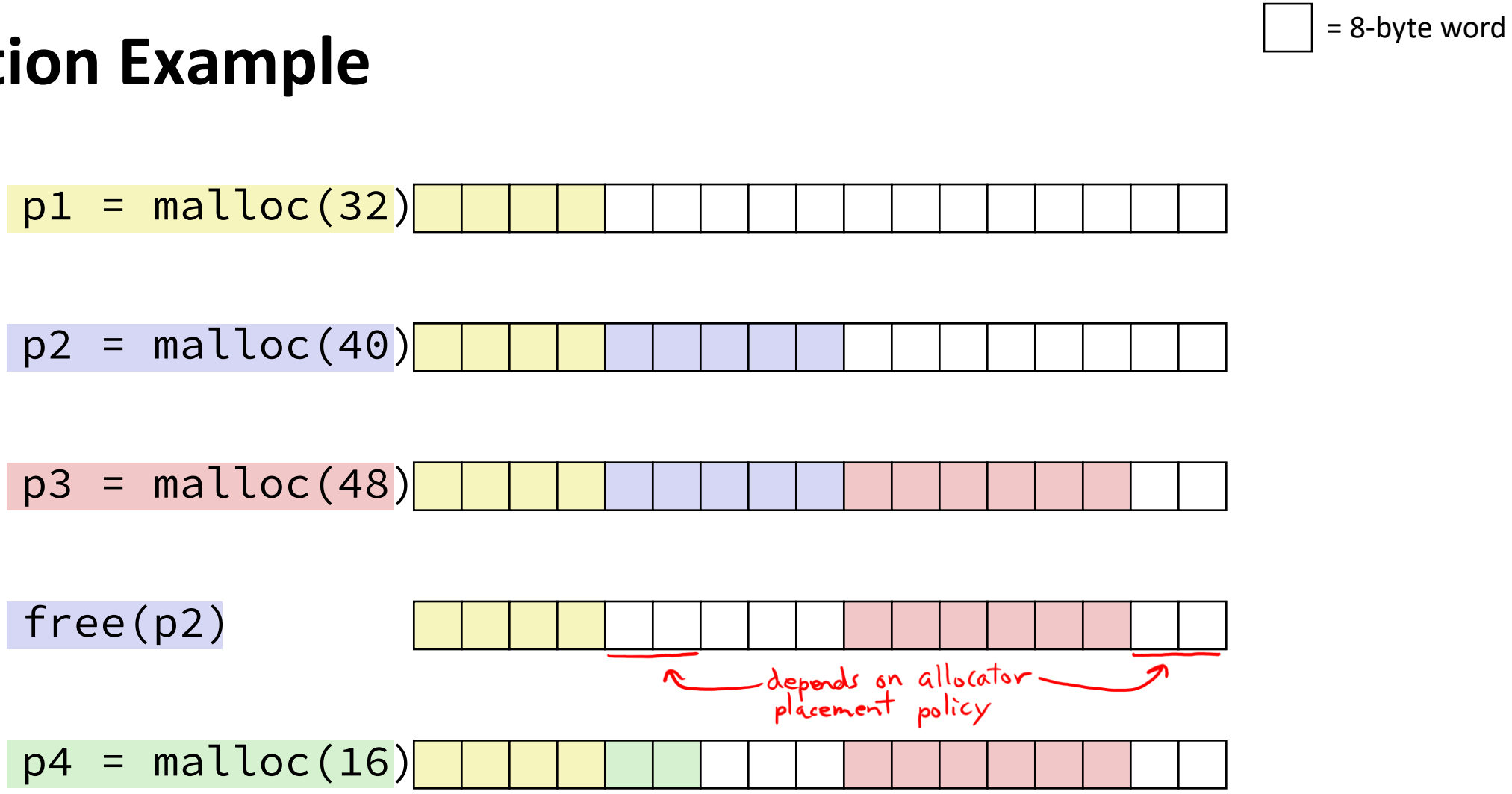
= 1 word = 8 bytes

Notation

- 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 😕



Allocation Example



Implementation Interface (Review)

- **Applications**
  - Can issue arbitrary sequence of allocation and deallocation requests
  - Must never access memory not currently allocated
  - Must never deallocate memory not currently allocated

- **Allocators**
  - Can't control number or size of allocated blocks
  - Must respond immediately to allocation requests (can't reorder or buffer)
  - Must allocate blocks from free memory (blocks can't overlap)
  - Can't move the allocated blocks (defragmentation not allowed) would break your pointers!
  - Must align blocks so they satisfy all alignment requirements

Performance Goals: Throughput

- **Goals:** Given some sequence of allocation and deallocation 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
- Example:
  - If 5,000 `malloc` calls and 5,000 `free` calls completed in 10 seconds, then throughput is 1,000 operations/second

Performance Goals: Memory Utilization

- Definition: *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
- Definition: *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 $U_k = (\max_{i \le k} P_i)/H_k$ after k+1 requests
- Goal: maximize utilization for a sequence of requests
- Why is this hard? And what happens to throughput? pack fast or pack tight?

Lecture Outline (4/5)

- Cache Performance, Revisited
- Dynamically Allocated Memory in C
- Dynamic Memory Allocators
- **Heap Fragmentation**
- Heap Implementation Basics

Fragmentation (Review)

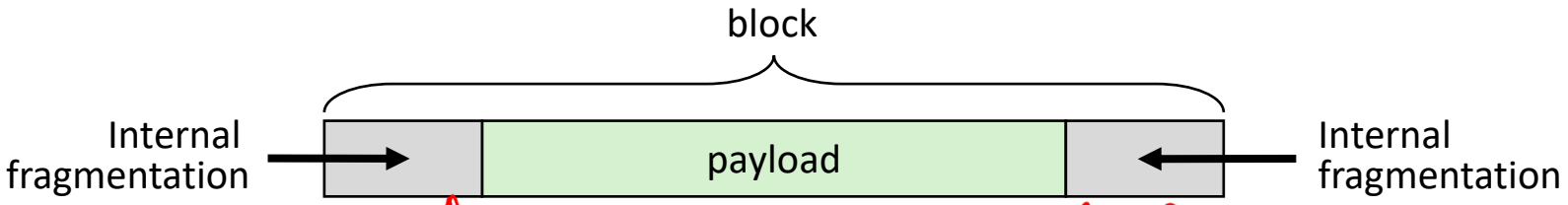
- Poor memory utilization is caused by *fragmentation*
  - Sections of memory are not used to store anything useful, but cannot satisfy allocation requests
  - Two types: *internal* and *external*

- **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 (Review)

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



- **Causes:**
  - Padding for alignment purposes
  - Overhead of maintaining heap data structures (inside block, outside payload)
  - Policy decisions (*e.g.*, minimum block size) faster throughput to not individually size every block

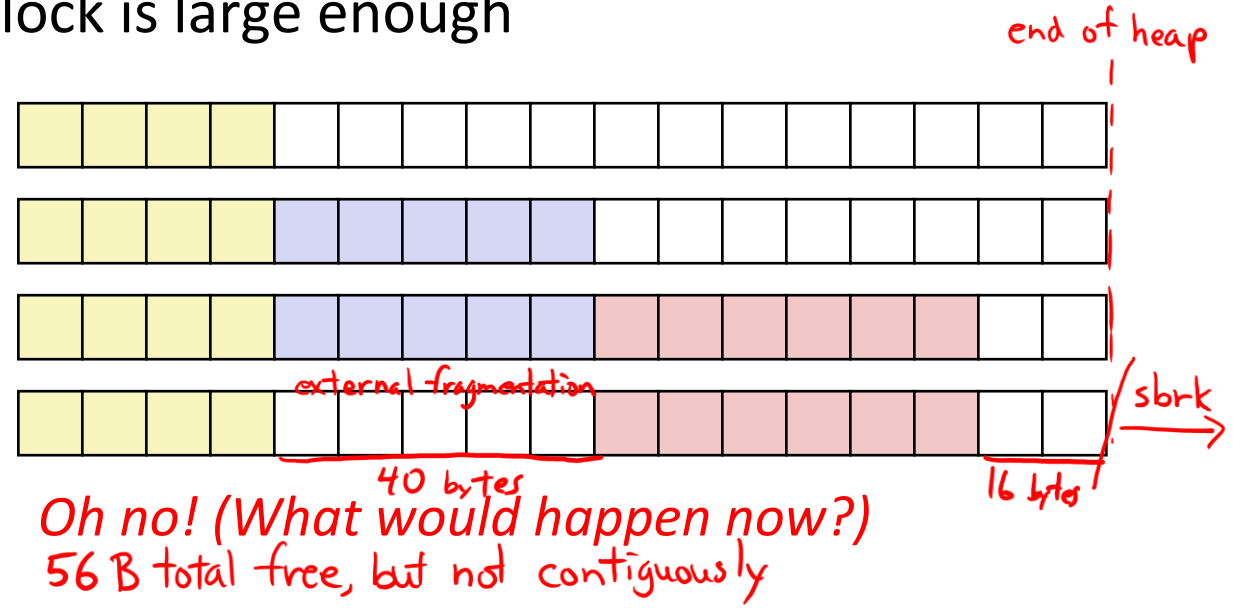
- Easy to measure because only depends on *past* requests

= 8-byte word

External Fragmentation (Review)

- For the heap, *external fragmentation* occurs when the allocation/free pattern leaves "holes" between blocks
  - 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)
```



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

Polling Questions (1/2)

- Which of the following statements are FALSE?

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

should allocate on the Stack

B. `malloc` returns an address of a payload that is filled with mystery data

allocates only; no initialization

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

$\frac{\text{aggregate payload}}{\text{heap size}}$

D. An allocation failure will cause your program to stop

just returns NULL

E. We're lost…

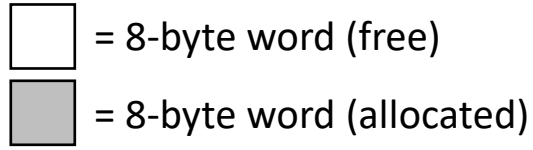
Lecture Outline (5/5)

- Cache Performance, Revisited
- Dynamically Allocated Memory in C
- Dynamic Memory Allocators
- Heap Fragmentation
- **Heap Implementation Basics**

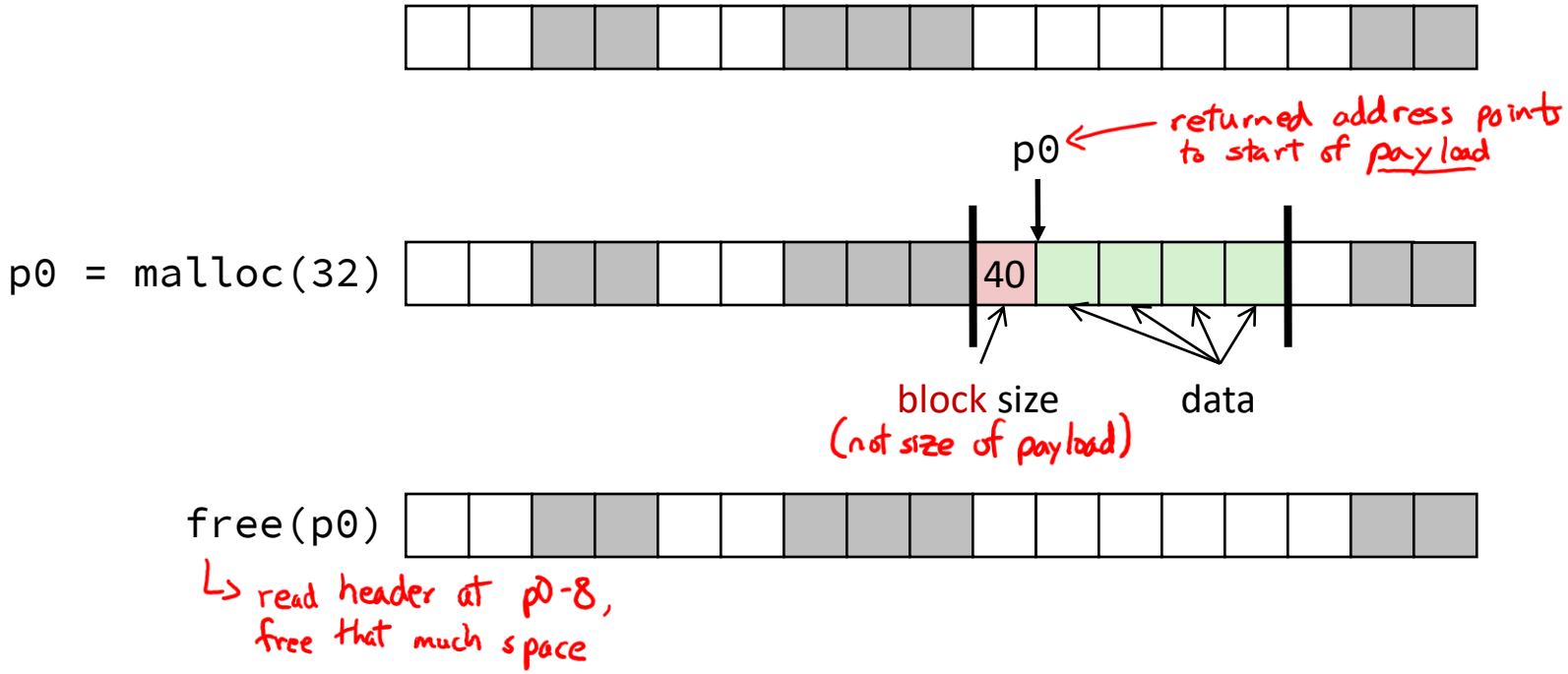
Implementation Issues

- **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



- 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



Header Information

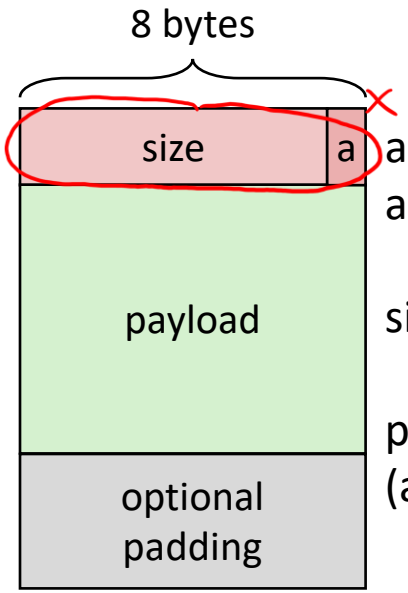
address is multiple of 8=0b1000

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?** (1 word, 1 bit)
- 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!

Format of allocated and free blocks:



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;
```

(or +; ~1 = -2)

Polling Questions (2/2)

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

all multiples of 16 have lowest 4 bits as zeros. ⇒ 16 = 0b10000

4 flags

- 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

two steps: ① mask out bit ②
② shift into LSB ①

(header & 2) >> 1

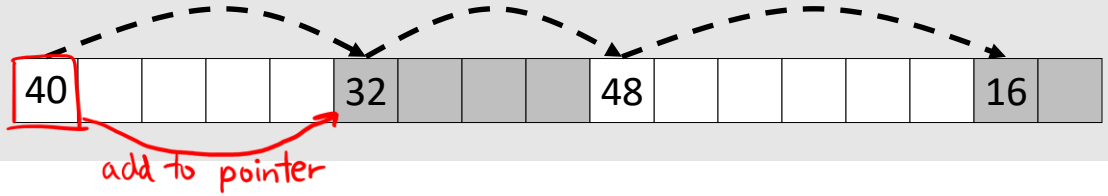
(header >> 1) & 1

Keeping Track of Free Blocks

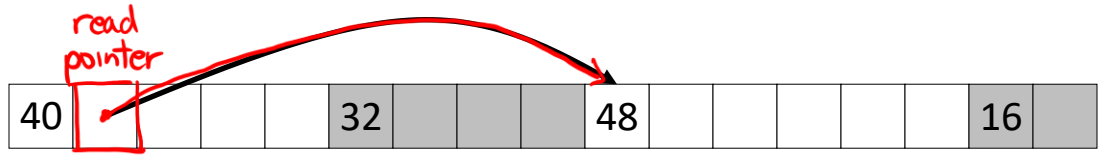
= 8-byte word (free)

= 8-byte word (allocated)

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



2) *Explicit free list* among only the free blocks, using pointers



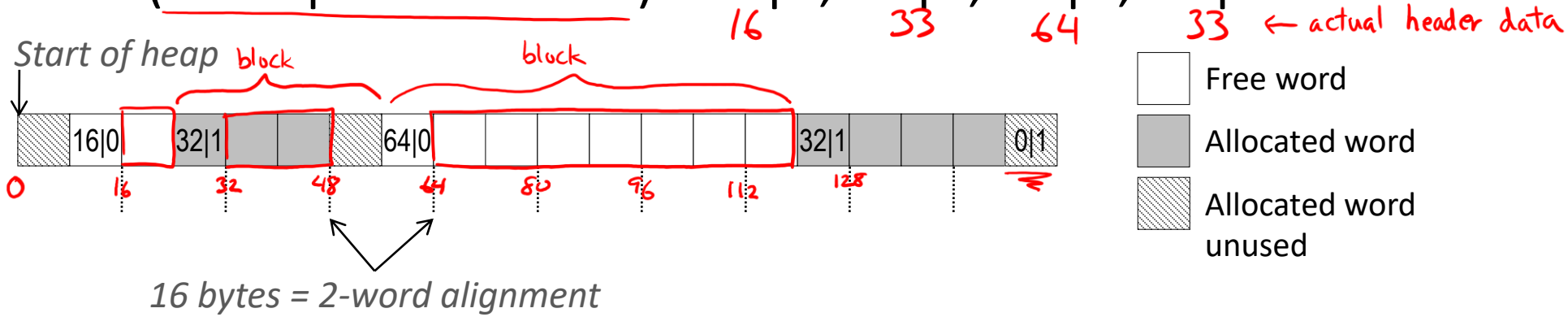
3) *Segregated free list*
   - Different free lists for different size "classes"

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 List Example

= 8-byte word

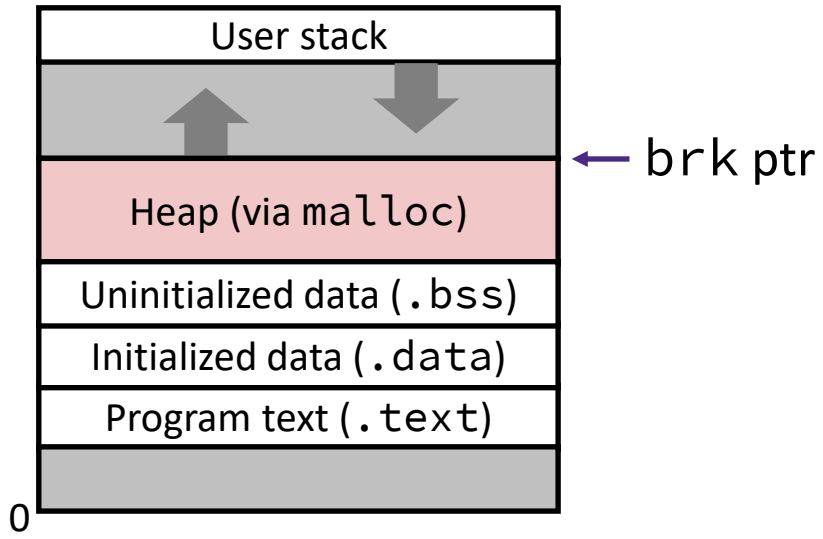
- Each block begins with header (size in bytes and is-allocated? bit)
- Heap blocks (`size`|is-allocated?): 16|0, 32|1, 64|0, 32|1



- 16-byte alignment for (1) *heap block size* and (2) *payload address*
  - Padding for size is considered part of *previous* heap block (internal fragmentation)
  - May require initial padding at start of heap
- Special one-word marker (0|1) marks end of list
  - Zero `size` is distinguishable from all other blocks

Summary (1/3)

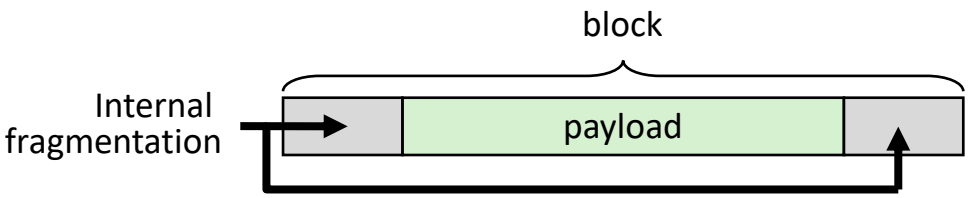
- **Dynamic memory allocation** is used when size or lifetime is not known until runtime
  - Memory allocated in the heap segment of memory:
  - In C: `void* malloc(size_t size)`
  - In C: `void free(void* p)`
  - In Java: **`new`**

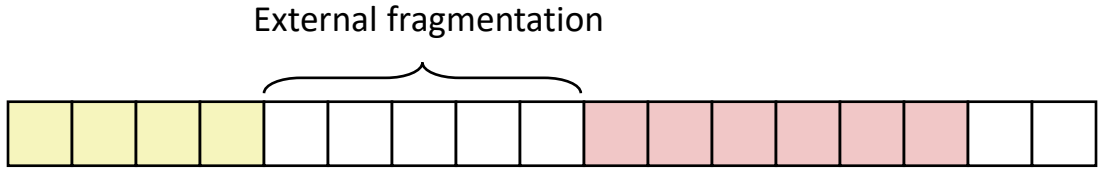


- Managed by dynamic memory allocator
  - Implicit: automatic deallocations, Explicit: manual deallocations
  - Performance metrics: throughput, memory utilization

Summary (2/3)

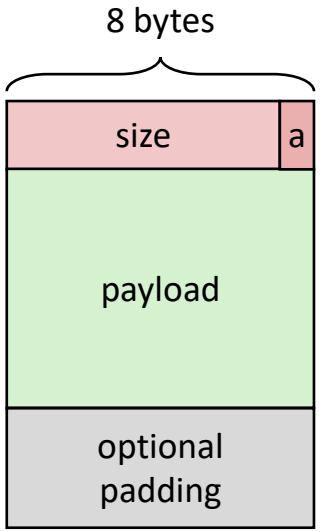
- The heap is divided into allocated and free **heap blocks**
  - Fragmentation: internal is non-payload space within blocks, external is free space between allocated blocks





  - Blocks have headers with size and is-allocated? Information:

Format of allocated and free blocks:



a = 1: allocated block
a = 0: free block

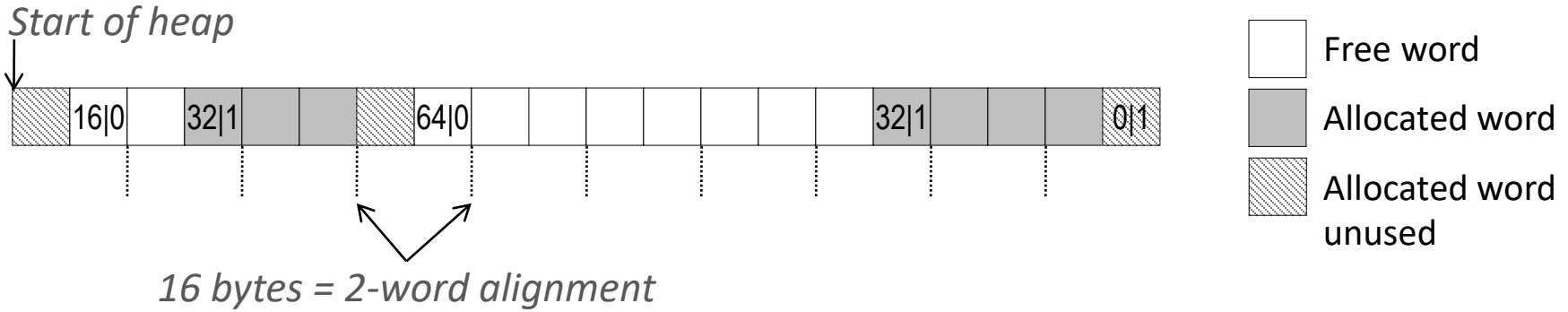
size: block size (in bytes)

payload: application data (allocated blocks only)

= 8-byte word

Summary (3/3)

- Implicit free list example
- Heap blocks (`size`|is-allocated?): 16|0, 32|1, 64|0, 32|1



- 16-byte alignment for (1) *heap block size* and (2) *payload address*
  - Padding for size is considered part of *previous* heap block (internal fragmentation)
  - May require initial padding at start of heap
- Special one-word marker (0|1) marks end of list
  - Zero `size` is distinguishable from all other blocks