Memory Allocation I

CSE 351 Spring 2024

Instructor:

Elba Garza

Teaching Assistants:

Ellis Haker
Adithi Raghavan
Aman Mohammed
Brenden Page
Celestine Buendia
Chloe Fong
Claire Wang
Hamsa Shankar
Maggie Jiang
Malak Zaki
Naama Amiel
Nikolas McNamee
Shananda Dokka
Stephen Ying
Will Robertson

Playlist: CSE 351 24Sp Lecture Tunes!

Announcements, Reminders

- Lab 3 due & Lab 4 releasing tonight
- HW17/18 due Friday, HW19 due Monday (13 May)
- Midterm due last night!
  - How’d it go?
  - Expect grades in a week-ish, more or less...
- Looking ahead: Guest lectures on May 15th and 17th

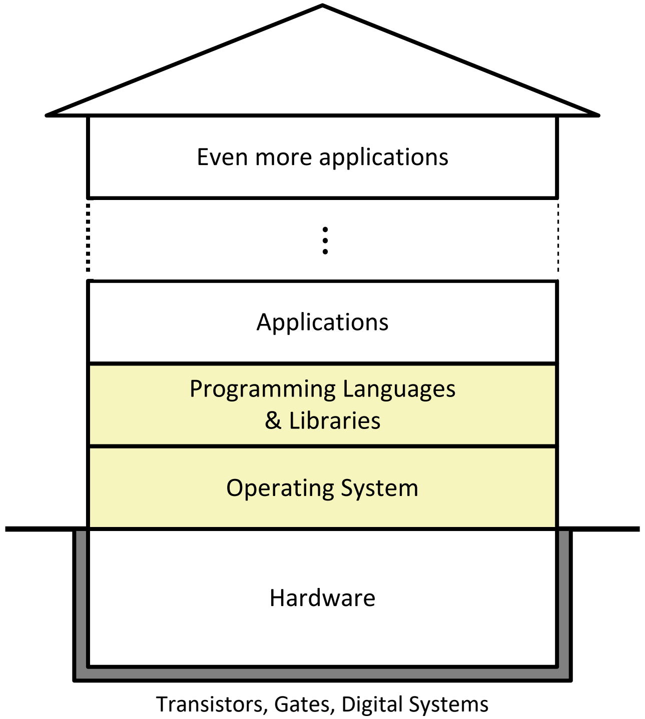
Current Events & CSE 351

- There may be interruptions to course resources:
  - Office Hours
  - Section
  - Grading

- Please bear with us as information comes in and the situation develops…

The Hardware/Software Interface

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


Even more applications
⋮
Applications
Programming Languages & Libraries
Operating System
Hardware
Transistors, Gates, Digital Systems

Physics

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

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)
  - Accessible anywhere in program
  - A 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** (*e.g.*, based on user-input)
  - **Lifetime known only at runtime** due to control by programmer (*e.g.*, `malloc`/`free` in C)

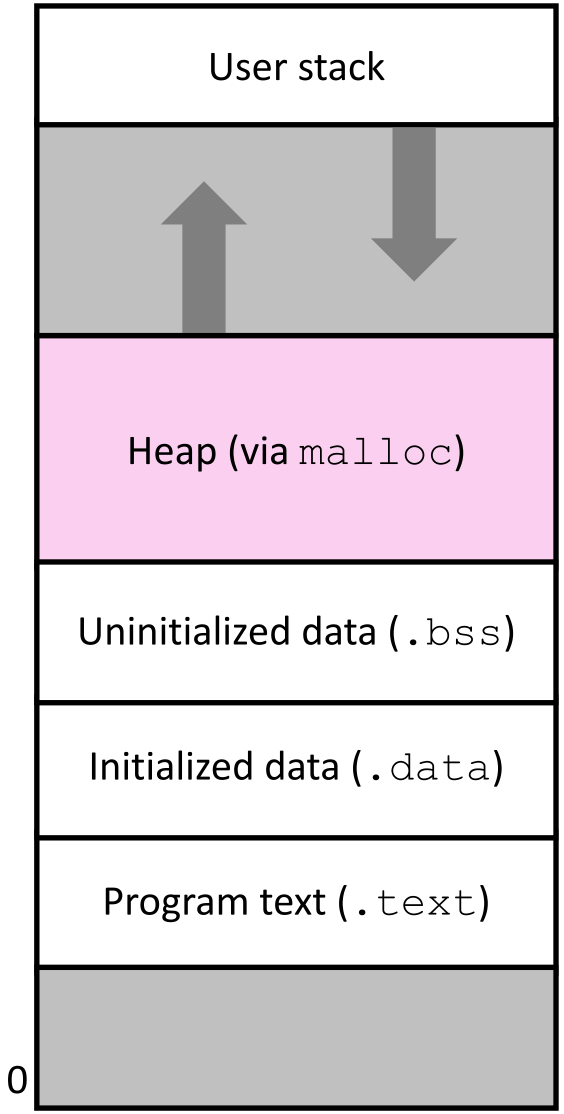
```
int array[1024];

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

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

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

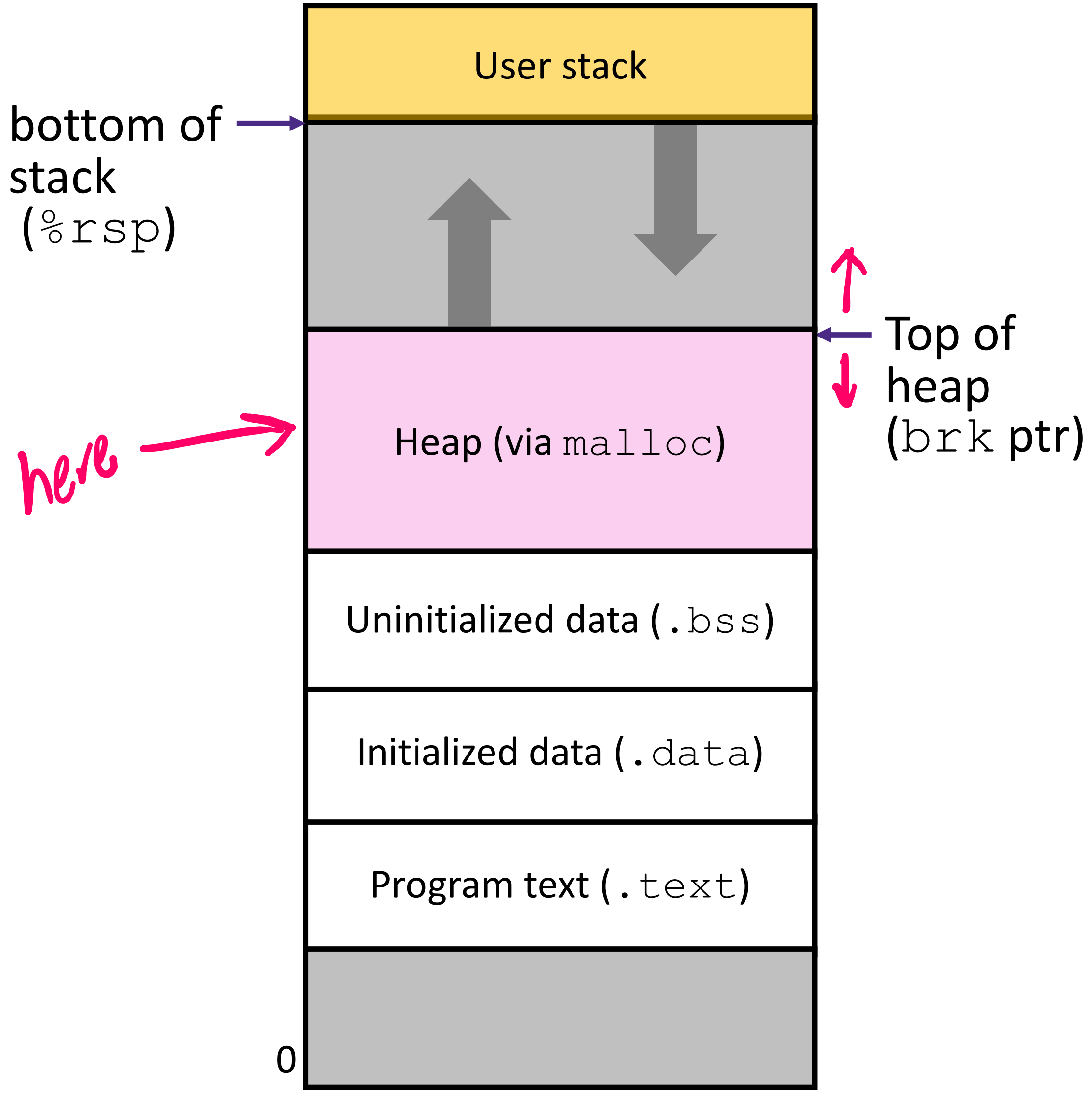
Dynamic Memory Allocation (Review)

- 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:
- Types of allocators
  - **Explicit** allocator: programmer allocates and frees space
    - Example: `malloc` and `free` in C
  - **Implicit** allocator: programmer only needs to allocate space (no free)
    - Example: use `new`, and garbage collection is done for you in Java, Ruby, and Python



Dynamic Memory Allocation

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



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
  - `size_t`?! Simple `typedef` for an unsigned 8-byte integer
  - Returns a pointer to the beginning of the allocated block; `NULL` if request failed
    - 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
- Best practices:
  - `ptr = (int*) malloc(n*sizeof(int));` // allocate space for n-many ints
    - `sizeof` makes code more portable (`int`s aren't the same size in all machines…)
    - `void*` is implicitly cast into any pointer type; explicit typecast will help you catch coding errors when pointer types don't match e.g. (int*)

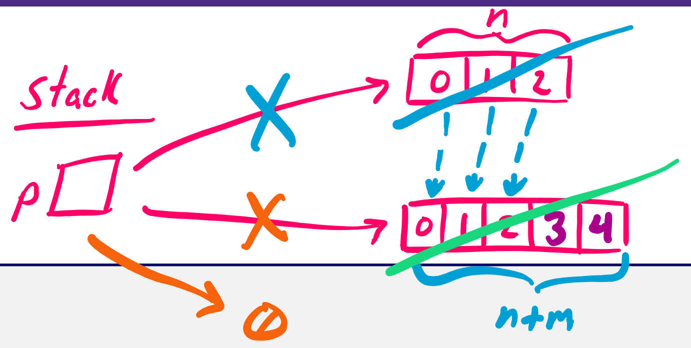
Allocating Memory in C (Review)

- Need to `#include <stdlib.h>`
- `void* malloc(size_t size)`
  - Allocates a continuous block of `size` bytes of uninitialized memory
  - **`size_t`**?! Simple `typedef` for an unsigned 8-byte integer
  - Returns a pointer to the beginning of the allocated block; `NULL` if request failed
    - 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)` //Note: freeing doesn't change pointer value! It will still point to (now) deallocated memory.
  - Releases whole block pointed to by `p` back to the pool of available memory
  - Pointer `p` must be the address originally returned by `(m|c|re)alloc` (*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)`

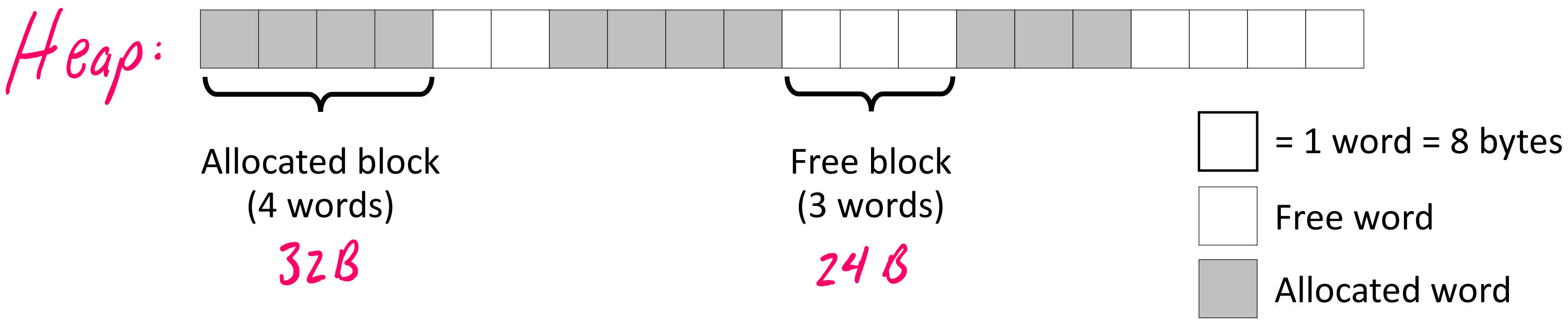
Memory Allocation Example in C



```
                 3         2
void foo(int n, int m) {
  int i, *p;
1 p = (int*) malloc(n*sizeof(int));          /* allocate block of n ints for an array */
  if (p == NULL) {                           /* check for allocation error */
    perror("malloc");
    exit(0);
  }
  for (i=0; i<n; i++)                        /* initialize int array */
    p[i] = i;

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

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)
  - **Note:** Book and old videos still use 4-byte word
    - Holdover from 32-bit version of textbook 🙁



Heap:

Allocated block
(4 words)
32B

Free block
(3 words)
24 B

□ = 1 word = 8 bytes

□ Free word

▨ Allocated word

Allocation Example

□ = 8-byte word

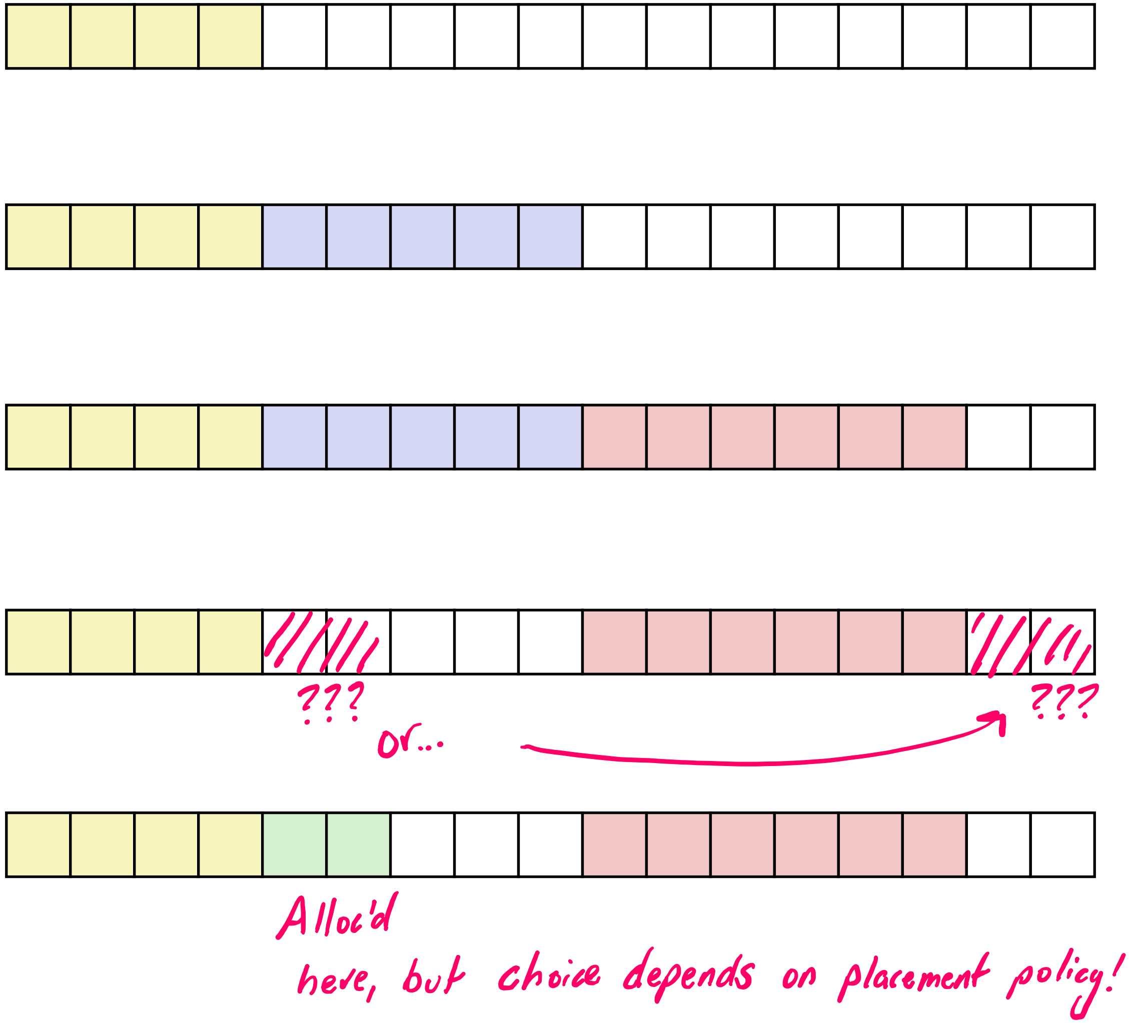
`p1 = malloc(32)`

`p2 = malloc(40)`

`p3 = malloc(48)`

`free(p2)`

`p4 = malloc(16)`



Alloc'd here, but choice depends on placement policy!

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` (can't reorder or buffer them!)
  - Must allocate blocks from free memory (no overtaking or overlapping)
  - Must align blocks so they satisfy all alignment requirements
  - Can't move the allocated blocks Boo!

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
- Example:
  - If 5,000 `malloc` calls and 5,000 `free` calls completed in 10 seconds, then throughput is 1,000 operations/second

Performance Goals

- 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

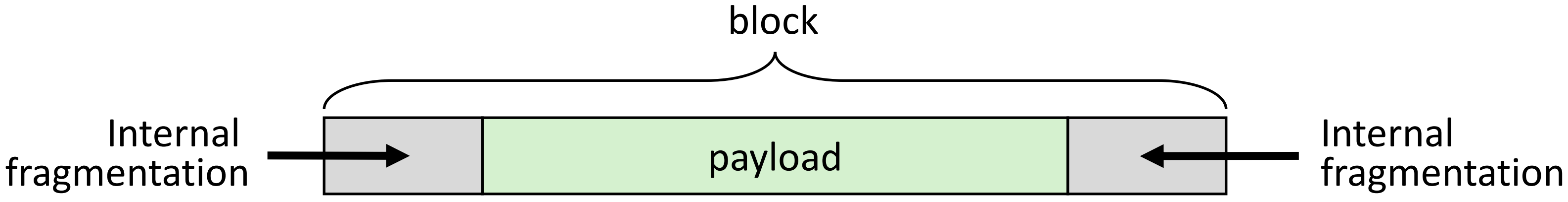
- Defined as $U_k = (\max_{i \le k} P_i)/H_k$ after k+1 requests
- Goal: maximize utilization for a sequence of requests
- Why is this hard? And what happens to throughput?

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 `struct`s
  - 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



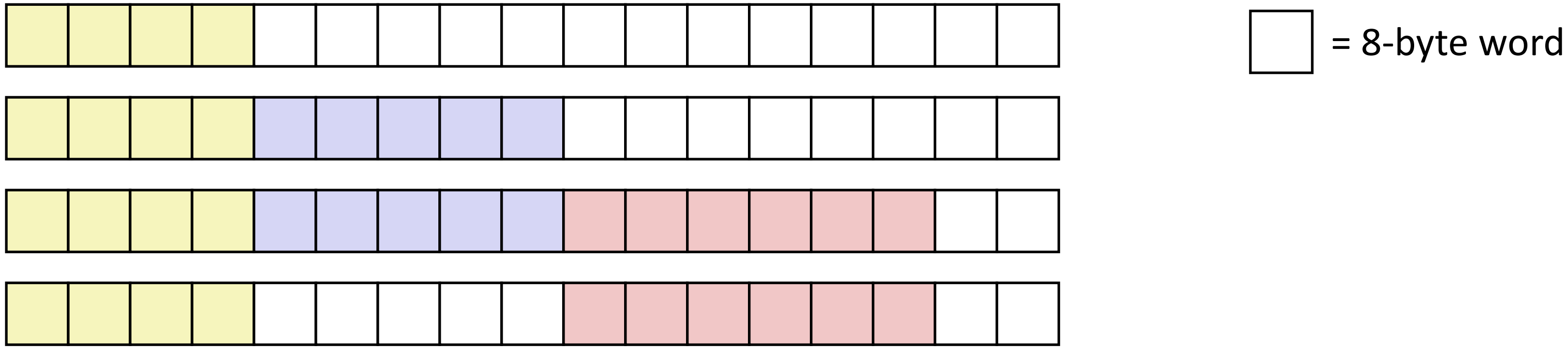
- **Causes:**
  - Padding for alignment purposes
  - Overhead of maintaining heap data structures (inside block, outside payload)
  - Explicit policy decisions (*e.g.*, return a big block to satisfy a small request)
- Easy to measure because only depends on past requests

Why? Faster in throughput, but bad for utility.

External Fragmentation

- 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?
  - A. Temporary arrays should not be allocated on the Heap yep, the stack!
  - B. `malloc` returns an address of a block that is filled with mystery data yep!
  - C. Peak memory utilization is a measure of both internal and external fragmentation yep!
  - [✓ D.] An allocation failure will ~~cause your program to stop~~ nah, just returns NULL
  - E. We're lost…

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?

Implementation Issues

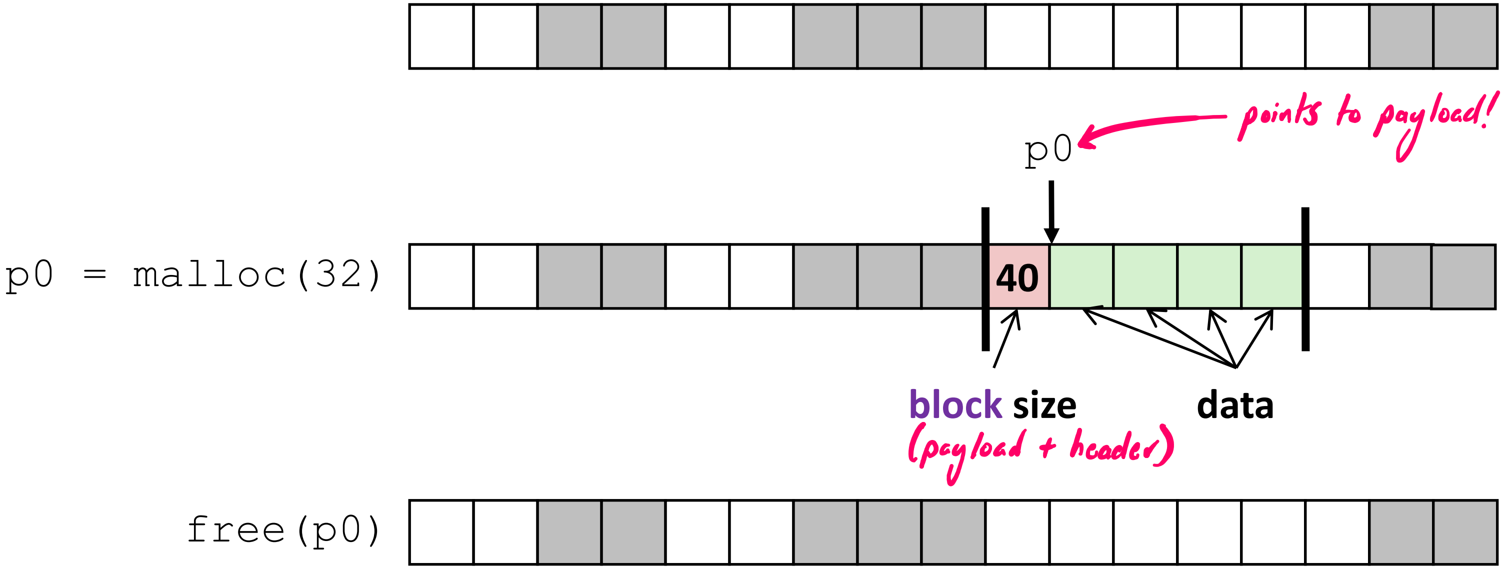
- **How do we know how much memory to free given just a pointer?**
- **How do we keep track of the free blocks?**
- How do we pick a block to use for allocation (when many might fit)?
- What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
- 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 just, **header**
  - Requires an <u>extra word for every allocated block</u>

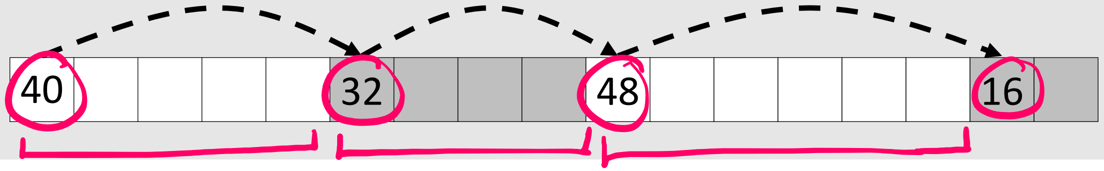
□ = 8-byte word (free)

▨ = 8-byte word (allocated)



Keeping Track of Free Blocks

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

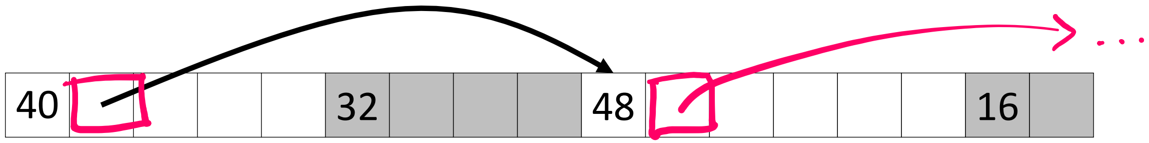


□ = 8-byte word (free)

▨ = 8-byte word (allocated)

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

Lab 5 funness!



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

Out of scope of 351

Implicit Free Lists

1 word 1 bit!!

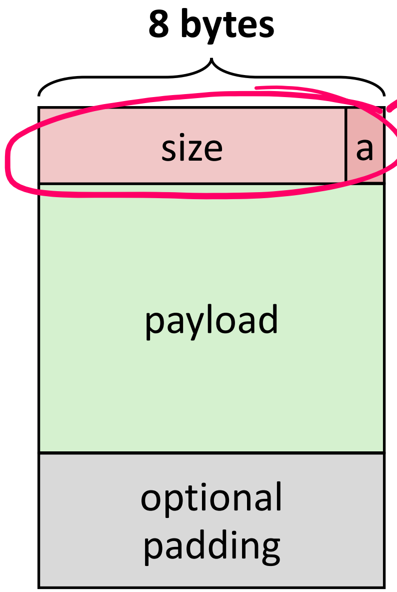
- For each block we need: **`size`**, **is-allocated?**
  - Could store using two words, but kinda wasteful…
- Standard trick
  - If blocks are aligned, some low-order bits of `size` are always 0
  - Use lowest bit as an allocated/free flag (fine as long as aligning to K>1)
  - When reading `size`, must remember to mask out this bit! Don't forget!

e.g., with 8-byte alignment, possible values for size:

```
00001000 = 8 bytes
00010000 = 16 bytes
00011000 = 24 bytes
. . .
```

Which is guaranteed!

Format of allocated and free blocks:



x

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

size: block size (in bytes)

payload: application data (allocated blocks only)

If `x` is the 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?

8-byte → 3 least significant bits are all 0, so up to 3 flags

16-byte → 4 least significant bits are all 0, so up to $\underline{4}$ 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 the `header`!

Option #1: (header >> 1) & 1

Option #2: (header & 2) >> 1