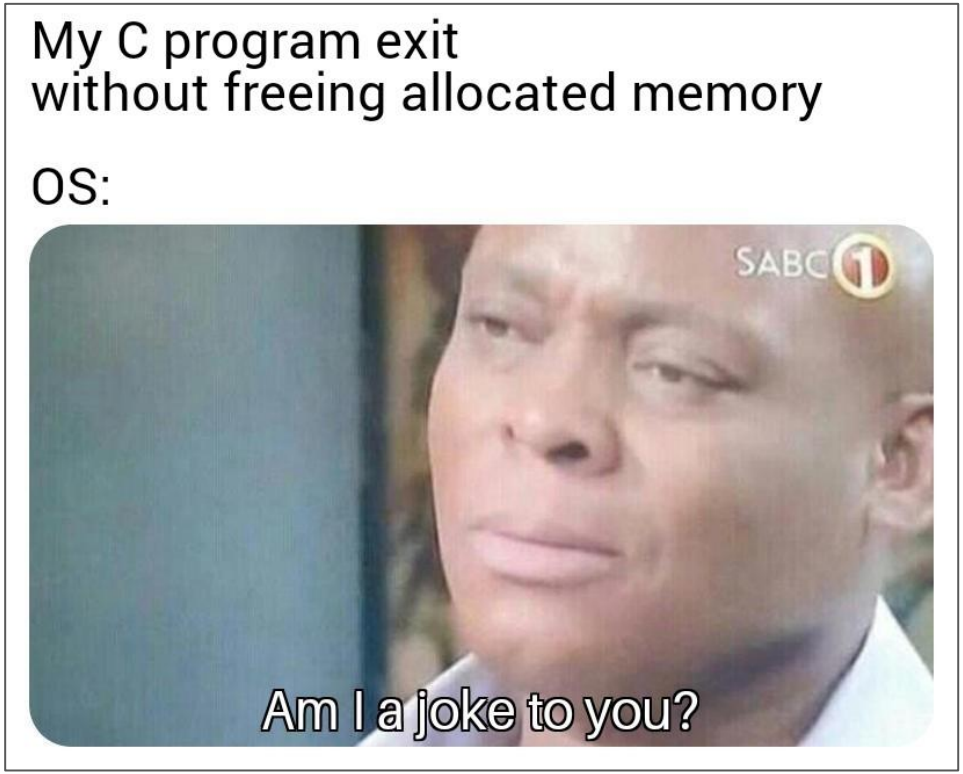
Memory Allocation I

CSE 351 Summer 2024

Instructor:
Ellis Haker

Teaching Assistants:
Naama Amiel
Micah Chang
Shananda Dokka
Nikolas McNamee
Jiawei Huang

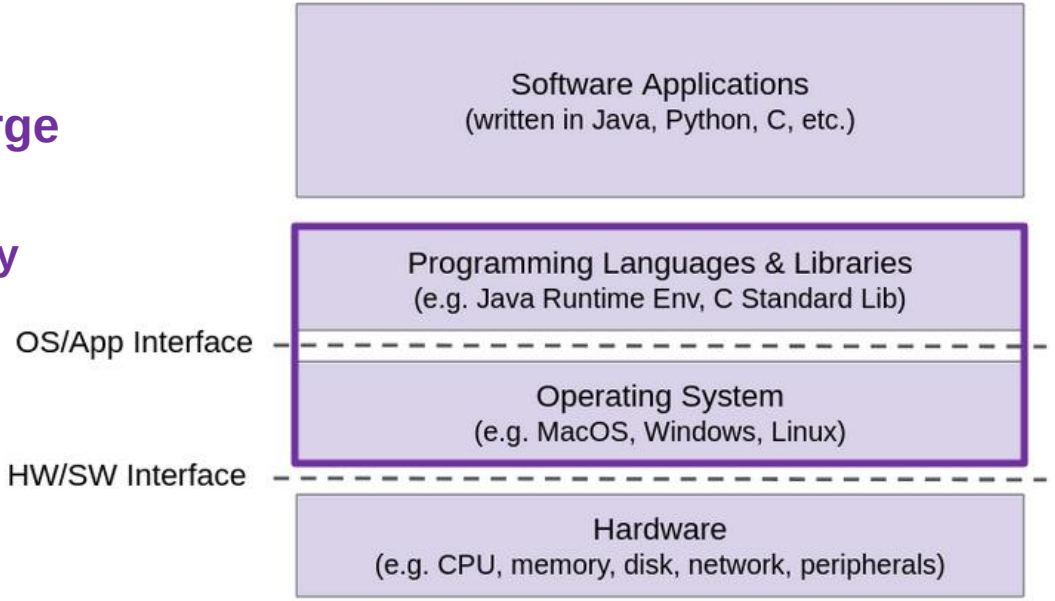


Administrivia

- Today
  - HW17 due (11:59pm)
- Friday, 8/2
  - RD20 due (1pm)
  - No HW :)
  - **Lab 5 released (due 8/14)**
- Monday, 8/5
  - RD21 due (1pm)
  - HW19 due (11:59pm)

Topic Group 3: Scale & Coherence

- How do we make memory accesses faster?
- **How do programs manage large amounts of memory?**
  - **How can we allocate memory *dynamically* (i.e. at runtime)**
- How does your computer run multiple programs at once?

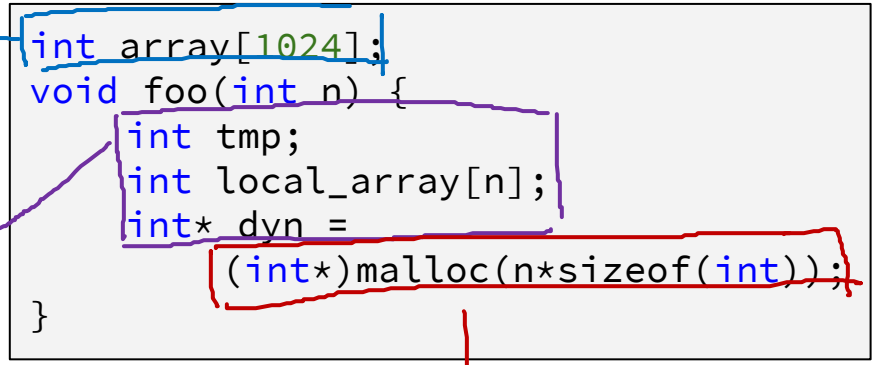


Dynamic Memory Allocation

- **Overview**
  - **Introduction and goals**
  - **Allocation and deallocation (free)**
- **Malloc and free in C**
  - **Common memory-related bugs in C**
- Fragmentation
- Explicit allocation implementation
  - Implicit free lists
  - Explicit free lists (Lab 5)
  - Splitting and coalescing
- Implicit deallocation: garbage collection

Multiple Ways to Store Program Data

- **Static global data**
  - **Fixed size** at compile-time
  - Lasts **entire lifetime** of the program
  - Accessible anywhere in the program
  - A portion is read-only (literals)
- **Stack-allocated data**
  - Local/temporary variables
    - Can be **dynamically allocated** (in some versions of C)
  - **Known lifetime** (deallocated when function returns)
- **Dynamic (heap) data**
  - Size and lifetime only **known at runtime**



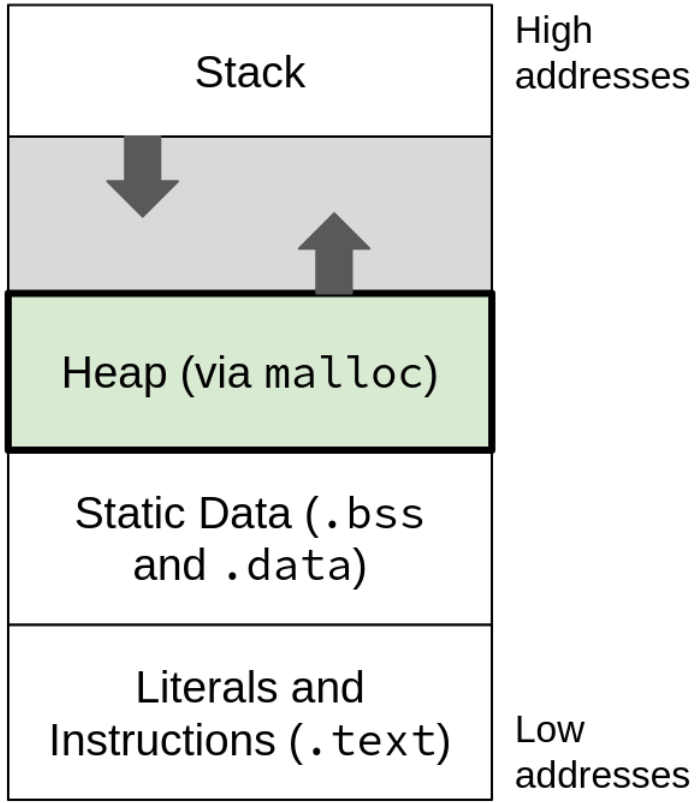
Dynamic Memory Allocation

- Programmers use **dynamic memory allocators** to acquire memory at runtime
  - For data structures whose size or lifetime is known only at runtime
  - Stored in **heap** segment of memory
- Types of allocators
  - **Implicit**: language handles **garbage collection**, programmer only needs to allocate the data
    - Ex: `new` in Java
  - **Explicit**: programmer needs to allocate *and* deallocate memory
    - Ex: `malloc` and `free` in C

Dynamic Memory Allocation (pt 2)

- Allocator organizes data into variable-sized **blocks**, which can be allocated or free

- What happens when the heap runs out of space?
  - Allocator asks the OS for more!
    - `sbrk` in Unix



Allocating Memory in C

- Need to `#include <stdlib.h>`
- `void* malloc(size_t size)`
  - Allocates a **contiguous** 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
    - Returns **`NULL`** if allocation failed, or `size=0`
      - Pointer to an invalid address (represented as address 0)
  - Blocks typically aligned to 8 or 16 bytes
- Other versions:
  - `calloc`: initializes memory to 0
  - `realloc`: moves existing block to a larger one

blocks returned from multiple calls to malloc are not guaranteed to be contiguous with each other

In C, NULL always = 0

Translates to 0 in x86, not true in all ISAs

Freeing Memory in C

- `void free(void* p)`
  - Releases block pointed to by `p` back to the pool of available memory
  - Pointer `p` must be the address originally returned by `malloc`/`calloc`/`realloc` (i.e., beginning of the block)
  - **Don’t call `free` on a block that has already been released!**
    - Undefined behavior - can even introduce buffer overflow vulnerabilities!
  - No action occurs if you call `free(NULL)`

Best Practices for `malloc` and `free`

- `malloc`
  - Using `sizeof` makes code more portable (`int`s aren't 4B on all machines…)
  - `void*` is implicitly cast, but explicitly casting will help you catch errors
  - Ex: `int* ptr = (int*) malloc(n * sizeof(int));`
- `free`
  - After calling `free`, set the pointer to `NULL`
    - Avoids double-free errors
  - Ex:

```
free(ptr);
ptr = NULL;
```

malloc and free Example

```
void foo(int n) {
    int i, *p;
    p = (int*) malloc(n*sizeof(int)); // allocate space for array 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;
    free(p);                                     // free p
    p = NULL;                                    // good practice to
set to NULL after free
}
```

check for malloc failure

print error message

`malloc` and `free` Interface

- Applications
  - Must never access memory not currently allocated
  - Must never free memory not currently allocated
    - Must only use `free` with previously `malloc`'ed blocks
- Allocators
  - 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

Heap Management in C

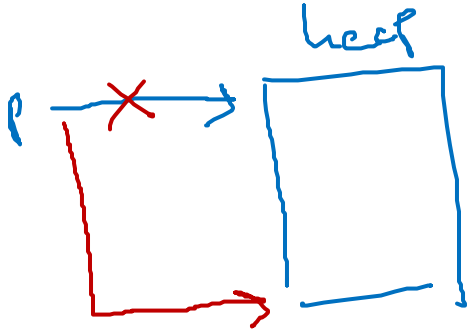
- Likely very different from what you're used to
- Programmer has to remember to free data when they're done with it
  - Otherwise, causes a **memory leak**
- Requires keeping track of where your data is stored
- Bugs are common!

Find that Bug!

```
int* p = (int*)malloc(N * sizeof(int));
for (int i = 0; i < N; i++) {
    *p = i;
    p++;
}
free(p);
```

changed p

address we're freeing isn't the same as what malloc returned



Freeing a pointer not returned by malloc

Find that Bug! (pt 2)

```
x = (int*)malloc(N * sizeof(int));
// manipulate x
free(x);
...
y = (int*)malloc(N * sizeof(int));
// manipulate y
free(x);
```

Double-free

Find that Bug! (pt 3)

```
x = (int*)malloc(N * sizeof(int));
// manipulate x
free(x);
...
y = (int*)malloc(M * sizeof(int));
for (i=0; i<M; i++) {
    y[i] = x[i]++;
}
```

we already freed x!

Use after free

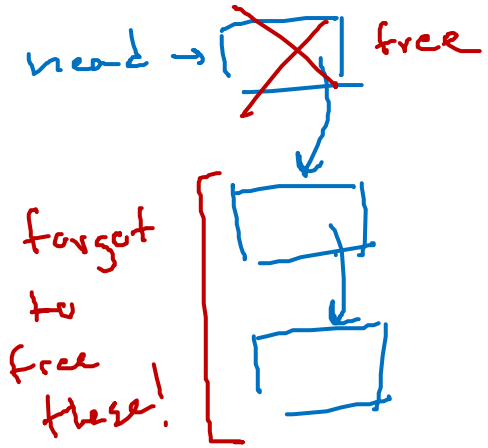
Find that Bug! (pt 4)

```
typedef struct L {
    int val;
    struct L* next;
} list;   // linked list node
void main() {
    list* head = (list*) malloc(sizeof(list));
    head->val = 0;
    head->next = NULL;
    // create and manipulate the rest of the list
    ...
    free(head);
    return;
}
```

allocate 1st node

call malloc more times to make more nodes (not shown)

only free the head!



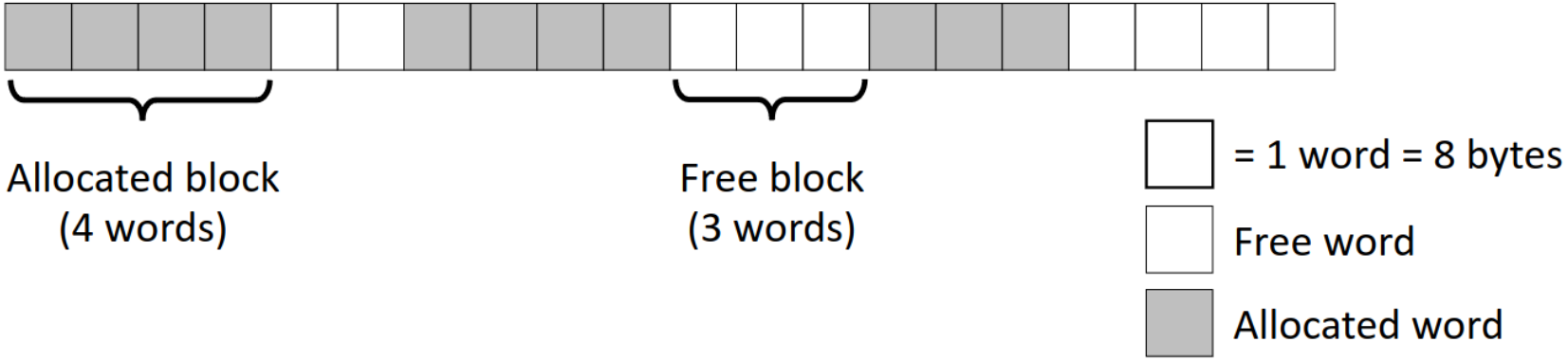
Memory leak

Dynamic Memory Allocation

- Overview
  - Introduction and goals
  - Allocation and deallocation (free)
- Malloc and free in C
  - Common memory-related bugs in C
- **Fragmentation**
- Explicit allocation implementation
  - Implicit free lists
  - Explicit free lists (Lab 5)
  - Splitting and coalescing
- Implicit deallocation: garbage collection

Notation

- We will draw memory divided into **words**
  - Each word is 64 bit = 8 bytes
  - Note: textbook and old videos still use 32-bit words
- Allocations will be aligned and in sizes that are a multiple of words



Memory Allocation Example

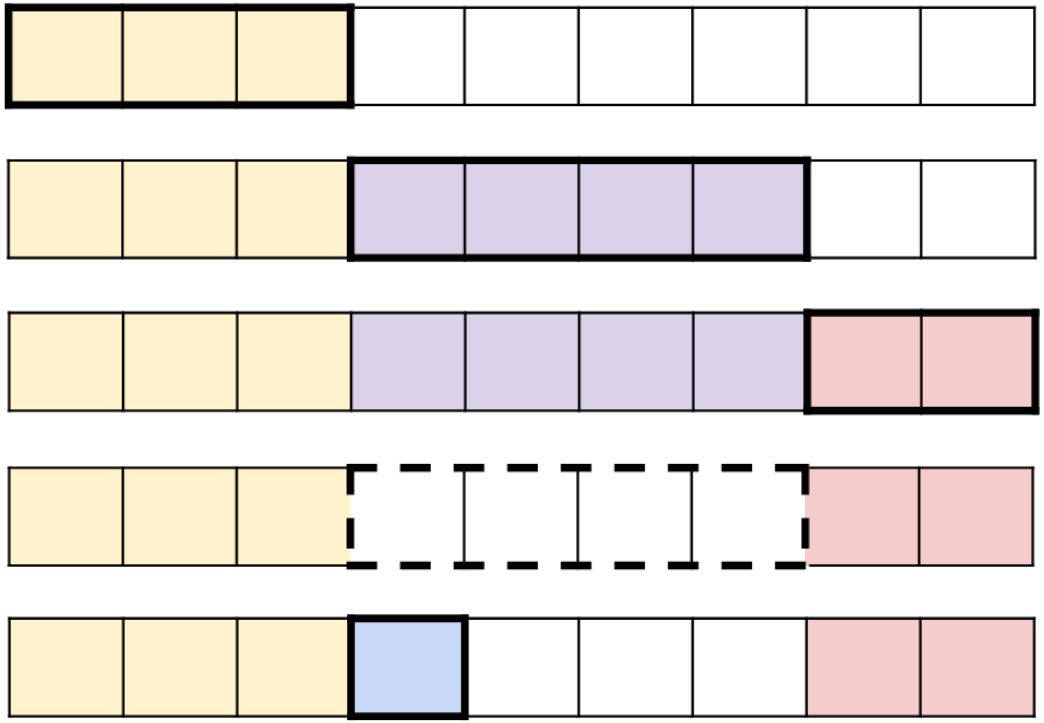
```
p1 = malloc(24)
```

```
p2 = malloc(32)
```

```
p3 = malloc(16)
```

```
free(p2)
```

```
p4 = malloc(8)
```



Performance Goals

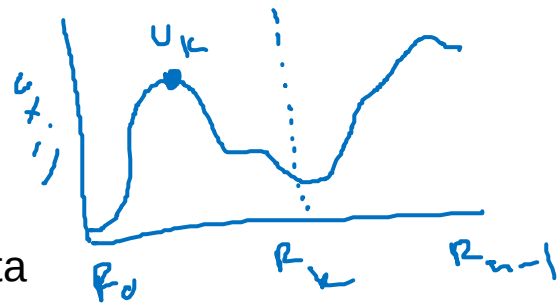
- Given some sequence of malloc and free requests $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$, maximize **throughput** and **peak memory utilization**
  - Often conflicting goals!

1) **Throughput**:
   - Number of completed requests per some unit of 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 (pt 2)

- **Aggregate Payload** (P_k)
  - `malloc(p)` returns a **payload** of `p` bytes
  - After request R_k has completed, P_k = the sum of currently allocated payloads
- **Current Heap Size** (H_k)
  - Allocator can increase heap size using `sbrk`
    - Assume allocator cannot decrease heap size

2) **Peak Memory Utilization** (U_k):



- Memory utilization = aggregate payload ÷ heap size
  - i.e., how much of our heap contains payload data
- $U_k = (\max_{i \le k} P_k) \div H_k$ after $k+1$ requests
  - i.e., the maximum utilization at any point up through when R_k has completed

Fragmentation

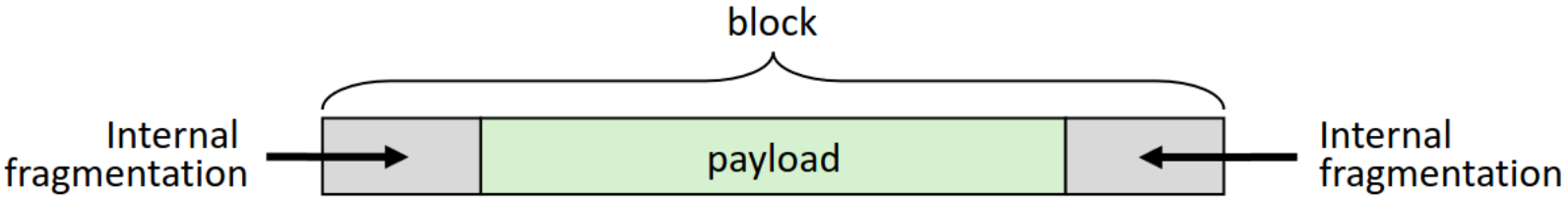
- Recall fragmentation in structs
  - Used to preserve alignment
  - **Internal**: unused space *inside* of a struct (between fields)
  - **External**: unused space *between* struct instances (after fields)

- The heap is similar
  - **Internal fragmentation**: extra space *inside* an allocated block
  - **External fragmentation**: extra space between allocated blocks

can be for alignment, or other reasons

Internal Fragmentation

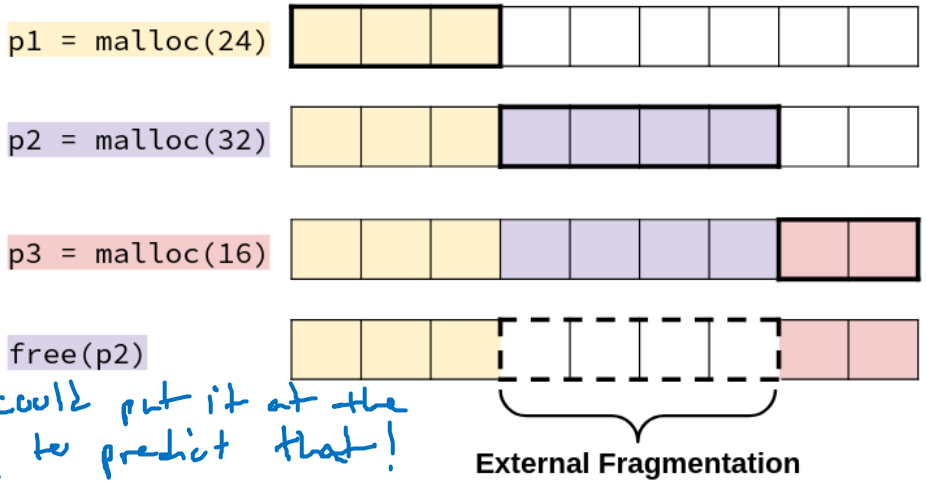
- Additional space inside of an allocated block not used to store a payload



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

External Fragmentation

- Occurs when allocation/free pattern leaves “holes” between blocks
  - Can cause situations where there is enough aggregate heap space to satisfy a request, but no single free block is large enough



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

ex: if we knew p2 would be freed 1st, could put it at the end of the heap, but there's no way to predict that!

Polling Question

1. Which of these statements is **FALSE**?
   A) Temporary arrays should not be allocated on the Heap
   B) malloc returns an address of a block that is filled with mystery data
   C) Peak memory utilization is a measure of both internal and external fragmentation
   D) An allocation failure will cause your program to stop

usually better to put on the stack

uninitialized

false, returns NULL on failure

Utilization = sum of all payloads / heap size

fragmentation is not part of a payload

Dynamic Memory Allocation

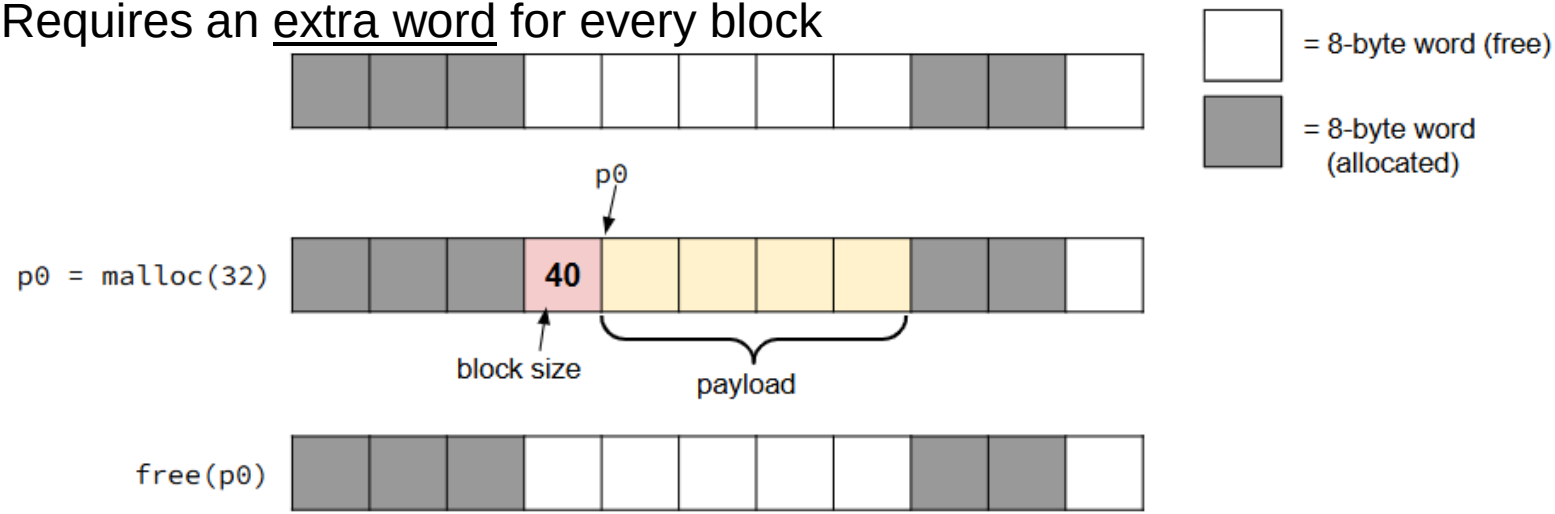
- Overview
  - Introduction and goals
  - Allocation and deallocation (free)
- Malloc and free in C
  - Common memory-related bugs in C
- Fragmentation
- **Explicit allocation implementation**
  - **Implicit free lists**
  - Explicit free lists (Lab 5)
  - Splitting and coalescing
- Implicit deallocation: garbage collection

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 reclaim free blocks?

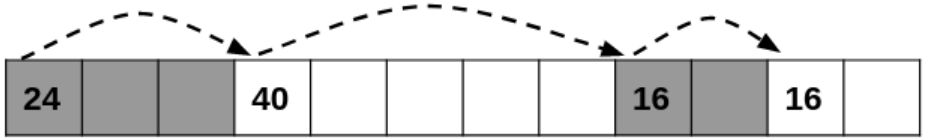
Knowing how much to free

- Store the length of the block in a **header**
  - The word preceding the data
  - Also may contain other info, like whether the block is allocated
- Requires an extra word for every block



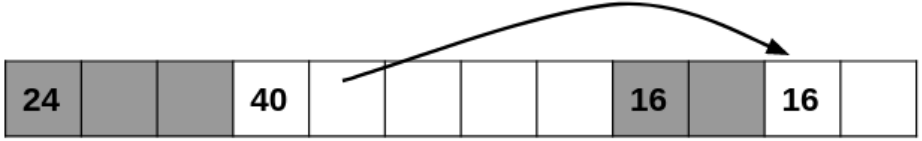
Keeping track of free blocks

- **Implicit Free List**
  - Use pointer arithmetic to traverse through entire heap until we find a free bock



pros: saves memory
cons: finding free block is slower

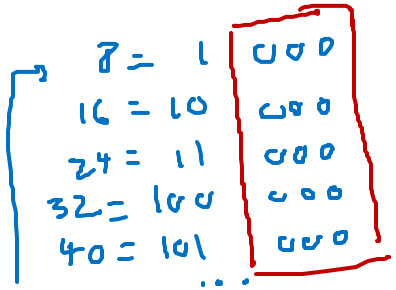
- **Explicit Free List**
  - Free block stores pointer to the next free block, forming a linked list



pros: finding free block is fast
cons: pointers take up memory

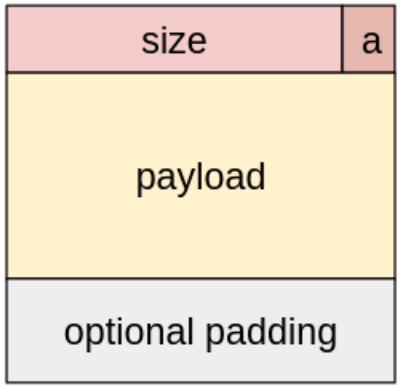
- Others not covered in this class
  - **Segregated free lists** (different lists for each object type), sorting blocks by size

Implicit Free Lists (i.e. no pointers)



- For each block, we need to store: **`size`**, **`is_allocated`**
  - Could use two works, but kinda wasteful…
- Recap: if size is a multiple of 2^n, then lowest *n* bits of the size are always 0
  - Use the lowest bit of header word to store **`is_allocated`** flag
  - When reading **`size`**, mask this bit out

Block format:



a = 1 if allocated, 0 if free

size = total block size in bytes

payload for allocated blocks only

preserves lowest bit, 0s out the rest

If the header value is h:

```
h = size | a
a = h & 1
size = h & ~1
```

= 0b11…10, 0s out lowest bit

Header Questions

- How many “flags” (boolean values) could we fit in our header if our allocator uses 16-byte alignment? 4

  16 = 2^4, lowest 4 bits will be 0

- 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!

  header & 2

  2 = 0b00…010

Summary

- The **heap** is a segment of memory used to **dynamically allocate** data
  - Useful when we don’t know the size or when it can be freed until runtime
- **Fragmentation** is space in the heap that is not used to store payloads
  - **Internal**: inside of a block
  - **External**: between blocks
- C uses an **explicit allocator**, meaning the programmer decides when heap data is freed
- Different heap implementations
  - **Implicit free list**: only store header and payload
  - **Explicit free list**: free blocks store pointers to the next free block (next lecture)