Memory Allocation I
CSE 351 Summer 2021

Instructor:
Mara Kirdani-Ryan

Teaching Assistants:
Kashish Aggarwal
Nick Durand
Colton Jobs
Tim Mandzyuk

Adapted from
https://xkcd.com/1093/
Gentle, Loving reminders

- hw18, 19 due tonight!
- hw20 due Friday (8/13)
- hw21 due Monday (8/16)

- Lab 5 is out!
  - Section tomorrow should help!
  - Lecture Friday should help too!
  - *If you’re using late days, send us an email! We need to submit final grades!*
Unit 3: Scale, Coherence

- Caches, Process, Virtual Memory
  - Multiple programs?
  - Larger programs?
- Metrics & Structures
- Memory Allocation
New topic, what’s implicit?
Malloc: Everything all at once

- Both technically, and sociotechnically!
  - Builds on alignment, memory access, pointer arithmetic, structs
- Memory allocation is structural
  - *Much like this entire course!*
  - Historic, ideological

- Y’all will probably experience more historic structures, built from ideology & metrics
  - I want to make sure y’all have practice with analysis
  - Ideally yours, not mine
Let’s try a new thing!

- For this lecture (and beyond, if you so choose), if you see something, say something!
  - Use 🤖, or 🏡
- I’ll lovingly ask you to explain what’s implicit, what’s assumed
  - This, at some level, is a personal interpretation
  - How you’re feeling about the material, now.
- *Disclaimer: I really don’t know how this will go!*
  - We can try it together?
Socio-Technical Callouts

- **Embedded values**: CS tends to emphasize efficiency, performance, minimalism, ruggedness
  - *We shape our tools, and our tools shape us*
- **Ideology**: What’s taken as fact, so much so that we don’t even need to ask?
  - *Neoliberalism*: individualism, self-sufficiency, self-reliance, emphasize individuals over structures
  - Building upon structures without examining them!
- **Access**: What structures exist? Who were they designed for? Who can use them?
- **Metrics**: Choice of metric is ideological, and shapes structures; optimizing for the average case harms people, knowledge & positivism
Sound good?
Feel ok?
Anything you’d like to be different?
Back to memory allocation!
Learning Objectives

Understanding this lecture means you can:

- Differentiate between explicit and implicit memory allocators, and utilize C’s memory allocation interface
- Define throughput and utilization, how fragmentation affects utilization, and how both determine allocator implementations
- Understand implicit free lists allocators
- Point out ideological assumptions in this lecture, with some help!
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)
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

- **Common memory-related bugs in C**
Dynamic Memory Allocation

- 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 & frees space
    - **Example**: `malloc` and `free` in C
  - **Implicit allocator**: programmer only allocates space
    - **Example**: garbage collection in Java, Caml, and Lisp
Dynamic Memory Allocation

- Allocator organizes heap as a collection of variable-sized blocks, either allocated or free
  - Allocator requests pages in the heap region; VM hardware and OS kernel allocate pages to the process
  - Application objects are typically smaller than pages, so the allocator manages blocks within pages
    - (Larger objects handled too; ignored here)
Allocating Memory in C

- 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

- 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

- Need to `#include <stdlib.h>`

- `void free(void* p)`
  - Releases whole block ref'd 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 or on `NULL`!
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

- We will draw memory divided into words
  - Each word is 64 bits = 8 bytes
  - Allocations will be in sizes that are a multiple of boxes (i.e. multiples of 8 bytes)
  - Book and old videos still use 4-byte word
    - Holdover from 32-bit version of textbook 😞
Allocation Example

\[
p_1 = \text{malloc}(32)
\]

\[
p_2 = \text{malloc}(40)
\]

\[
p_3 = \text{malloc}(48)
\]

\[
\text{free}(p_2)
\]

\[
p_4 = \text{malloc}(16)
\]

Each box represents an 8-byte word.
Implementation Interface

○ Applications
  • Issue arbitrary sequence of `malloc` and `free` requests
  • Must never access memory not currently allocated
  • Must never free memory not currently allocated
    • Can 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 alignment requirements
  • Can’t move the allocated blocks
Performance Goals

**Goals:** Given some sequence of `malloc` and `free` requests $R_0, R_1, ..., R_k, ..., R_{n-1}$, 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 $U_k = (\max_{i \leq k} P_i)/H_k$ after $k+1$ requests
- Goal: maximize utilization for a sequence of requests
- Why is this hard? And what happens to throughput?
Fragmentation

- Poor memory utilization 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

**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

- 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
Checking in!

- Which of the following statements is FALSE?
  - 🐶 Arrays that are sized at runtime should be stored on the heap
  - 🐇 malloc returns an address of a block that is filled with garbage
  - 🐐 Peak memory utilization is a measure of both internal and external fragmentation
  - 🦄 An allocation failure will cause your program to stop
  - 😎 Help!
Principles of memory allocation, feeling ok?
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 block length in the word preceding the data
    - This word is often called the *header field* or *header*
  - Requires an extra word for every allocated block

```
free(p0)
p0 = malloc(32)
```

- □ = 8-byte word (free)
- ■ = 8-byte word (allocated)
Tracking 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

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

---

[Diagram showing memory allocation with different sizes and states]
Implicit Free Lists

- 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!

**Format of allocated and free blocks:**

- **size**
  - block size (in bytes)
- **payload**
  - application data
  - (allocated blocks only)
- **optional padding**
- **a**
  - allocated block: $a = 1$
  - free block: $a = 0$

Let $X$ = header (first word)

- $x = $size $| a$
- $a = x \& 1$
- $size = x \& \sim 1$

**Example values for size:**

- 8 bytes: 00001000
- 16 bytes: 00010000
- 24 bytes: 00011000
- ...
Implicit Free List Example

- Each block begins with header (size in bytes and allocated bit)
- Sequence of blocks in heap (size|allocated): 16|0, 32|1, 64|0, 32|1
- 16-byte alignment for payload
  - May require initial padding (internal fragmentation)
  - Note size: padding is considered part of previous block
- Special one-word marker (0|1) marks end of list
  - Zero size is distinguishable from all other blocks
Finding a Free Block

**First fit**

- **Search list from beginning**, choose first free block that fits:

  ```c
  p = heap_start;
  while ((p < end) && // not past end
      ((*p & 1) || // already allocated
       (*p <= len))) { // too small
      p = p + (*p & -2); // go to next block (UNSCALED +)
  } // p points to selected block or end
  ```

  - Can take time linear in total number of blocks
  - In practice can cause “splinters” at beginning of list

---

| 16|0 | 32|1 | 64|0 |
|---|---|---|---|---|
| Free word | Allocated word | Allocated word unused |
Implicit List: Finding a Free Block

- **Next fit**
  - Like first-fit, but search list starting where previous search finished
  - Should often be faster than first-fit: avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is worse

- **Best fit**
  - Search the list, choose the *best* free block: large enough AND with fewest bytes left over
  - Keeps fragments small—usually helps fragmentation
  - Usually worse throughput
Implicit free lists ok?
A few thoughts on allocators
Dynamic Memory Allocation

- 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 & frees space
    - **Example**: `malloc` and `free` in C
  - **Implicit allocator**: programmer only allocates space
    - **Example**: garbage collection in Java, Caml, and Lisp
Performance Goals

Goals: Given some sequence of \texttt{malloc} and \texttt{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 \texttt{malloc} calls and 5,000 \texttt{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 $U_k = (\max_{i \leq k} P_i) / H_k$ after $k+1$ requests
- Goal: maximize utilization for a sequence of requests
- Why is this hard? And what happens to throughput?
Fragmentation

- Poor memory utilization 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