[CSED211] Introduction to Computer Software Systems

Lab 8: Malloc Lab

Hyeongmin Oh

COMPUTER ARCHITECTURE & OPERATING SYSTEMS LABORATORY

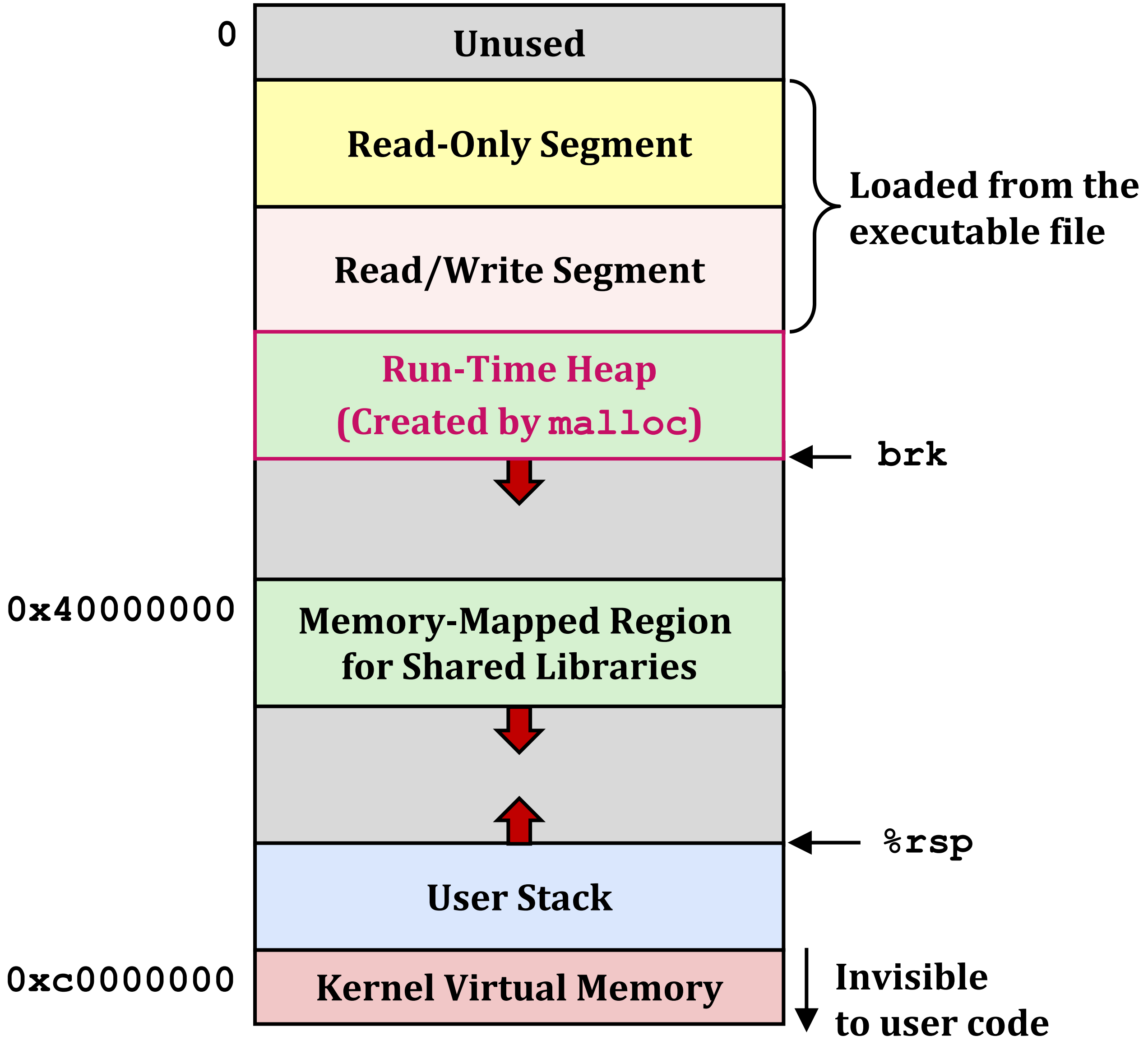
2024.12.12

Today's Agenda

- Background
  - Dynamic memory allocation
  - Example: Implicit free list
- Malloc Lab
- Quiz

Dynamic Memory Allocation

- Programmers use dynamic memory allocators (e.g., `malloc`) to acquire heap memory at runtime



Dynamic Memory Allocation (Cont.)

- Allocator maintains heap as collection of variable sized blocks
  - Which can be either allocated or free

- There are two types of allocator
  - Explicit allocator: Application allocates and frees space
    - e.g., `malloc` and `free` in C
  - Implicit allocator: Application only allocates, but does not free space
    - e.g., garbage collection in Java

Dynamic Memory Allocation (Cont.)

- Allocator maintains heap as collection of variable sized blocks
  - Which can be either allocated or free

- There are two types of allocator
  - Explicit allocator: Application allocates and frees space
    - e.g., `malloc` and `free` in C
  - Implicit allocator: Application only allocates, but does not free space
    - e.g., garbage collection in Java

- In this lab, we will focus on explicit allocator

Explicit Allocator in C (`malloc` package)

- Included in `stdlib.h`

- `void *malloc(size_t size)`
  - Allocate `size` bytes, and return a pointer to the address of the allocated object

- `void free(void *ptr)`
  - Free the memory space pointed by `ptr`

- `void *realloc(void *ptr, size_t size)`
  - Change the size of a previously allocated block into `size` bytes

- For more detail, check https://linux.die.net/man/3/malloc

Explicit Allocator in C (`malloc` package)

```c
#include <stdio.h>
#include <stdlib.h>

void foo(int n){
     int i, *p, *p_new;
     /* Allocate a block of n ints */
     p = (int *) malloc(n * sizeof(int));

     /* Initialize allocated block */
     for(i=0; i<n; i++)
         p[i] = i;

     /* Reallocate memory */
     p_new = (int *) realloc(p, 2 * n * sizeof(int));

     /* Return allocated block to the heap */
     free(p);
}
```

We will implement these functions!

Conditions for Good Memory Allocator

- To implement a good memory allocator, we should consider two things:
    - Throughput and memory utilization
        - They are often in a tradeoff relationship

Performance Goal: Throughput

- Throughput
  - The number of completed requests per unit time
  - e.g., 5,000 `malloc` calls and 5,000 `free` calls in 10 seconds
    → throughput: 1,000 operations per second

- To maximize throughput, the allocator should
  - Find a free memory block efficiently
  - Release an allocated memory block quickly

Performance Goal: Peak Memory Utilization

- Peak memory utilization
  - For a given sequence of malloc and free requests $R_0, R_1, \dots, R_k, \dots, R_{N-1}$
  - Aggregate payload P_k: Sum of currently allocated payloads after completing R_k
  - Current heap size H_k: Monotonically increase when allocator uses `sbrk()`
  - Peak memory utilization U_k after completing $(k+1)$ requests: $U_k = \max_{i \le k}(P_i)/H_k$

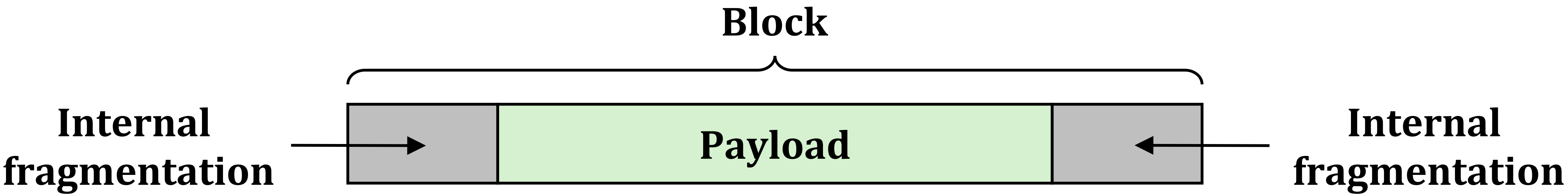
- To maximize peak memory utilization, allocator should avoid fragmentation

Fragmentation

- There are two kinds of fragmentation
  - Internal fragmentation
  - External fragmentation

Internal Fragmentation

- Occurs if the payload is smaller than the block size



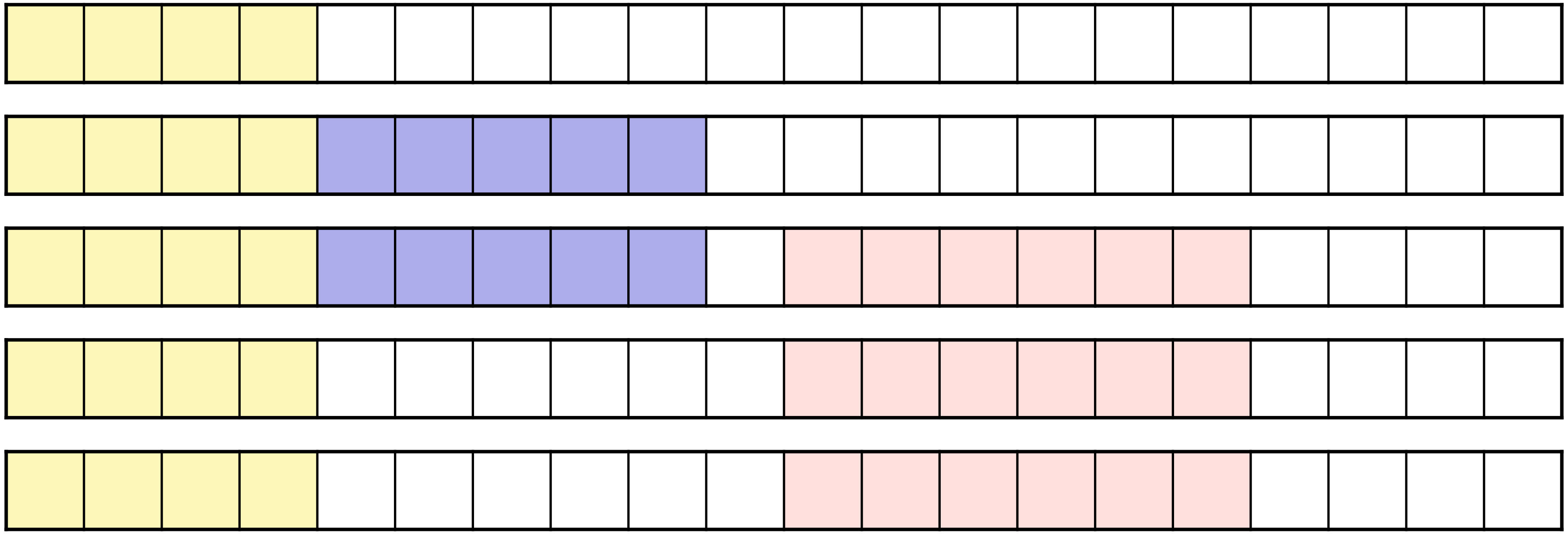
- Caused by
  - Overhead of maintaining heap data structures (e.g., boundary tag, pointer)
  - Padding for alignment purposes
  - Explicit policy decisions (e.g., return a big block to satisfy a small request)

External Fragmentation

- Occurs when allocating blocks and freeing the middle one, leaving small pieces of free space that may be too small to reuse

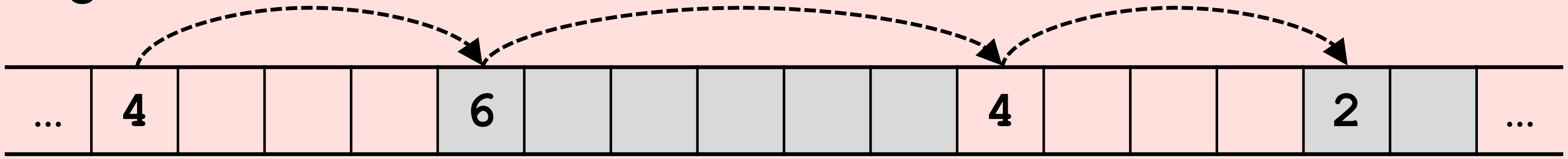
```
#define SZ sizeof(long)

p1 = malloc(4 * SZ)
p2 = malloc(5 * SZ)
p3 = malloc(6 * SZ)
free(p2)
p4 = malloc(7 * SZ)
```

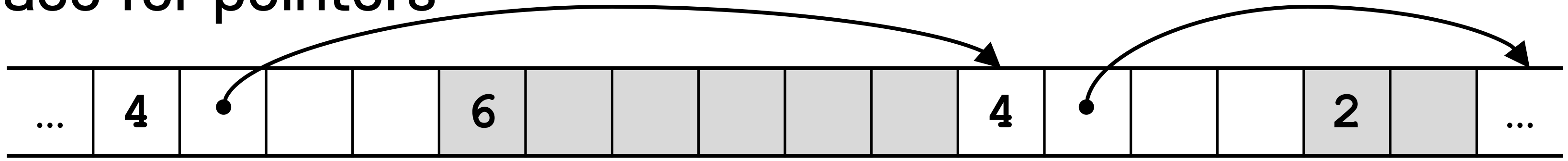


Free Block Management Schemes

- Method 1: Implicit list using length – links all blocks
  - Needs to tag each block as allocated or free



- Method 2: Explicit list among the free blocks using pointers
  - Needs space for pointers

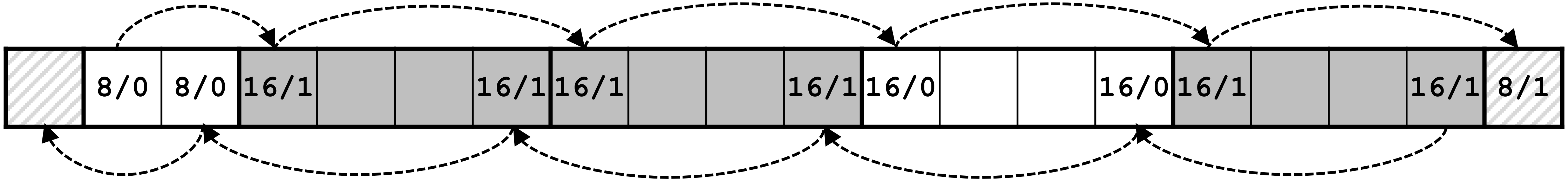


- Method 3: Segregated free list - different free lists for different size classes

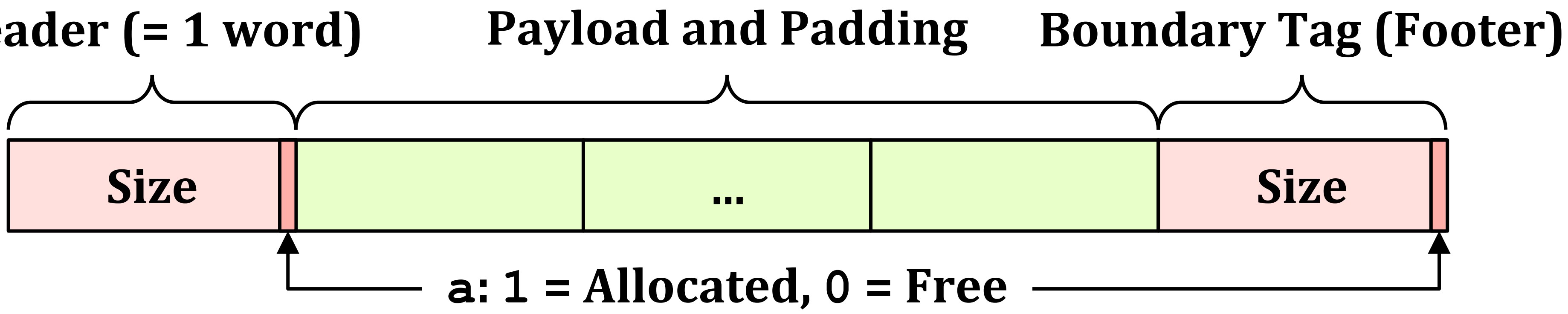
- Method 4: Blocks sorted by size - can use a balanced tree (e.g., red-black tree) with pointers within each free block, and the length used as a key

Implicit Free List

- Implicit: No explicit pointer to reference adjacent blocks



- Instead, location of the adjacent block can be inferred using boundary tags
  - Two boundary tags (header, footer) are needed to traverse bidirectional



Implicit Free List: Design Space

- There are multiple decision points when implementing implicit free list

- Placement policy
  - During `malloc`, what free block should be used for allocation

- Splitting policy
  - After `malloc`, split the remaining free space as a separate block or not

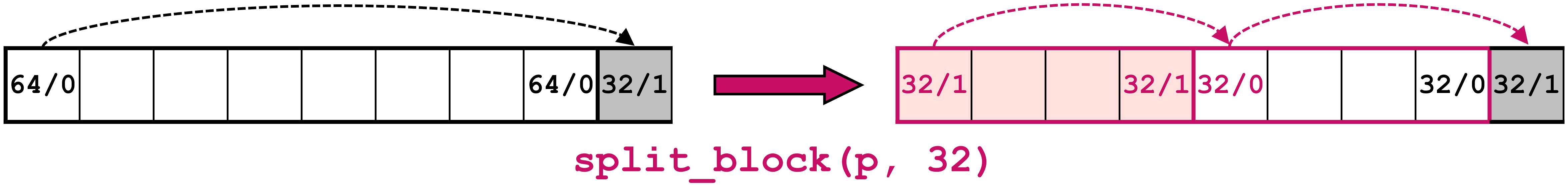
- Coalescing policy
  - After `free`, merge adjacent free block immediately or not

Implicit Free List: Design Space

- Placement policy
  - First fit
    - Searches the list from the beginning and chooses the first free block that fits
  - Next fit
    - Like first fit, but searches the list starting where previous search finished
  - Best fit
    - Searches the list and chooses the best free block that fits with the fewest bytes left

Implicit Free List: Design Space

- Splitting policy
  - Since allocated space might be smaller than the free space, it might need to split the free block



- Coalescing policy
  - Immediate coalescing: Coalesce each time `free` is called
  - Deferred coalescing: Try to improve performance of `free` by deferring coalescing until needed

Implicit Free List: Implementation

- You can implement implicit free list by referring CS:APP textbook
  - Placement policy: First-fit
  - Splitting policy: Enable splitting
  - Coalescing policy: Immediate coalescing
  - Check pp.883-897 for more detail

```
Results for mm malloc:
trace  valid  util     ops      secs   Kops
 0       yes   99%    5694  0.022506    253
 1       yes  100%    5848  0.019721    297
 …
24       yes   89%      12  0.000000   40000
25       yes   89%      12  0.000000   40000
Total          75%  209279  2.882481     73

Perf index = 45 (util) + 5 (thru) = 50/100
```

Implicit Free List: Performance Analysis

```
Perf index = 45 (util) + 5 (thru) = 50/100
```

- Moderate memory utilization (45/60)
  - Internal fragmentation: Good
    - Small data structure overhead compared to other free block management schemes
  - External fragmentation: Can be bad
    - First-fit does not care about remaining free block capacity
    - Fragmented free block keeps accumulated

- Low throughput (5/40)
  - If tiny `malloc()` requests are dominant, small blocks are accumulated at the front
    - Since implicit free list also traverses allocated block when finding available free block, throughput can be significantly decreased
  - Even though `coalescing()` is constant-time procedure, immediate coalescing can degrade throughput
    - If free block is used for the subsequent request, coalescing is unnecessary

Today's Agenda

- Background
- Malloc Lab
- Quiz

Malloc Lab: Assignment

- Goal: Gain hands-on experience about dynamic memory allocation

- In this lab, you will implement five functions declared in `mm.c`
  - `mm_init()`
  - `mm_malloc()`
  - `mm_free()`
  - `mm_realloc()`
  - `mm_check()` (for debugging, disable when you submit)

- Complete `mm.c` and run:
  - `$ make && ./mdriver -V`

Malloc Lab: Submission Guideline

- Due: 12/23 (Mon) 23:59
  - Late submission will not be accepted

- Two files must be uploaded to PLMS: mm.c and lab report (in pdf)

- Submission format (0 point if violated)
  - Code name: [student id]_mm.c (e.g., 2023xxxx_mm.c)
  - Report name: [student id].pdf (e.g., 2023xxxx.pdf)

- No page, font limit on lab report
  - Show every work you did!

Malloc Lab: Evaluation

- Score evaluation: Quiz (10%) + Code (40%) + Report (50%)
  - Code evaluation criteria
    - Performance index will be translated based on the score table

Perf. Index	<50	50~59	60~69	70~79	80~89	90~100
Score	10 pts	20 pts	25 pts	30 pts	35 pts	40 pts

  - Report evaluation criteria

function	mm_malloc()	mm_free()	mm_realloc()	mm_init()	mm_check()
Score	15 pts	15 pts	10 pts	5 pts	5 pts

Malloc Lab: Coding Rules (0 point if violated)

- Your code should compiled & run successfully

- Return pointers of your allocator must be 8-byte aligned
  - Malloc lab assumes 32-bit system, so double word is 8-byte

- Do not
  - Change the interface (function header) in `mm.c`
  - Use any memory-management related library or system calls
    - e.g., `malloc`, `free`, `realloc`, `calloc`, `sbrk`, `brk`
      - Do not implement `mm_malloc` using `malloc`, `mm_free` using `free`, …
    - `memcpy` is okay
  - Define any global/static compound data structure in `mm.c`
    - e.g., arrays, structs, trees, lists, …

Malloc Lab: Lab Report Guideline

- Please clarify the type of memory allocator you implemented
  - e.g., implicit free list, explicit free list, segregated free list, …

- For each function, please provide
  1. Screenshot of your code
  2. How does your code work (detailed comment is enough)

- Provide the rationale behind your throughput / memory utilization score
  - e.g., why your memory allocator shows poor memory utilization,
    why your memory allocator shows high throughput, …

Cheating Policy

- You can refer
  - Textbook source code
  - Malloc lab writeup, lab slides, lecture slides
  - Internet sources that doesn’t involve direct answers or codes about malloc lab

- You should not refer
  - ChatGPT
  - Code and report from a senior who has already taken this course
  - Blogs or github repository that contain solution codes
  - Every other references that violate POSTECHIAN’s Honor

Quiz

- Go to the PLMS, and start the Quiz!