Malloc Lab

TA: Avimita Chatterjee
Recitation Date: 27th November 2018

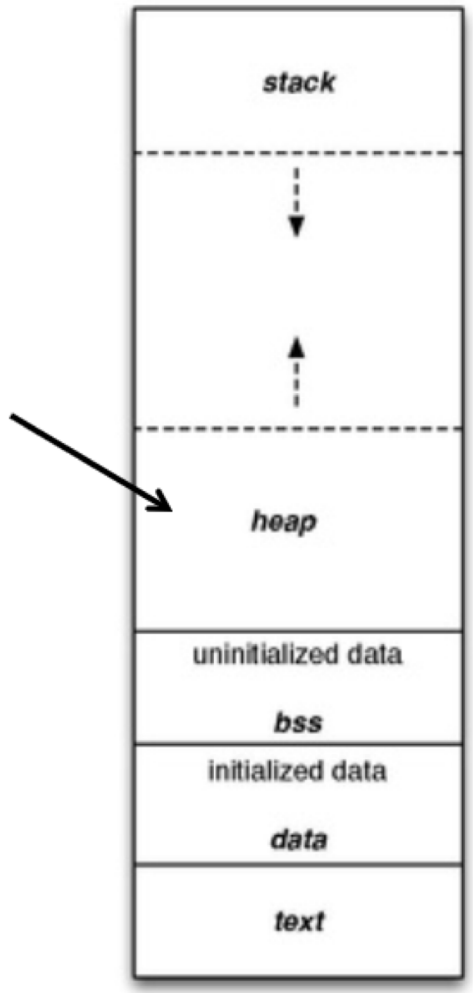
Introduction

- In this lab you will be writing a dynamic storage allocator for C programs, i.e., your own version the malloc, free and realloc routines.
- You are encouraged to explore the design space creatively and implement an allocator that is correct, efficient and fast.
- Malloc Lab Introduction Video: https://www.youtube.com/watch?v=v62G2ySAGxY
- You may work in a group of up to two people. Any clarifications and revisions to the assignment will be posted on the course Moodle.

What is malloc?

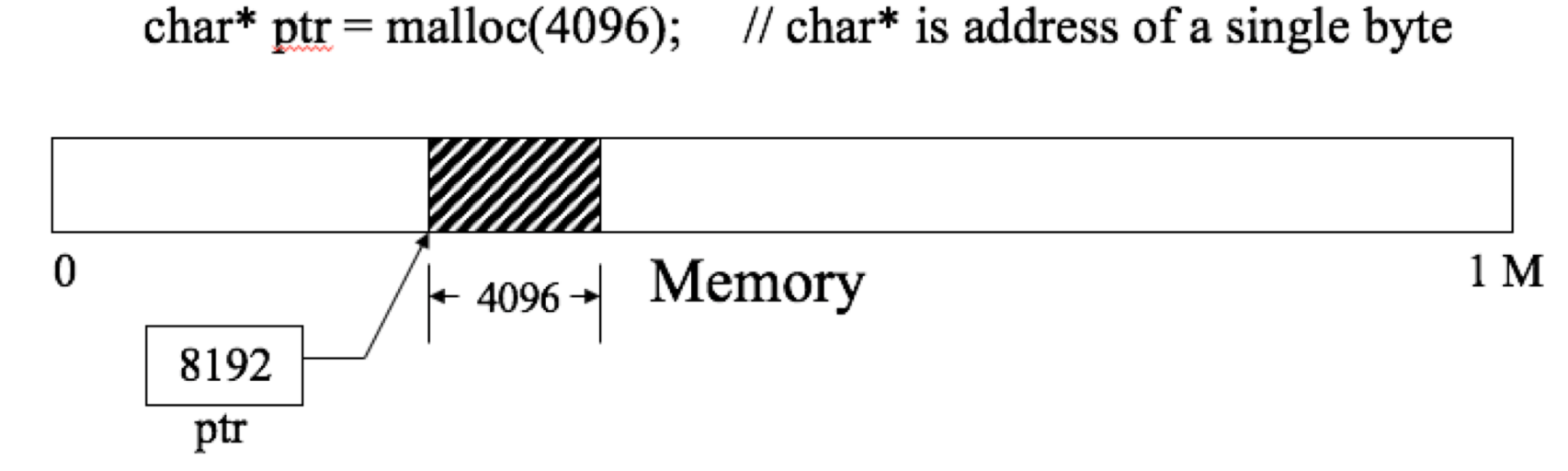
- **A function to allocate memory during runtime (dynamic memory allocation).**
  - More useful when the size or number of allocations is unknown until runtime (e.g. data structures)

- **The heap is a segment of memory addresses reserved almost exclusively for malloc to use.**
  - Your code directly manipulates the bytes of memory in this section.



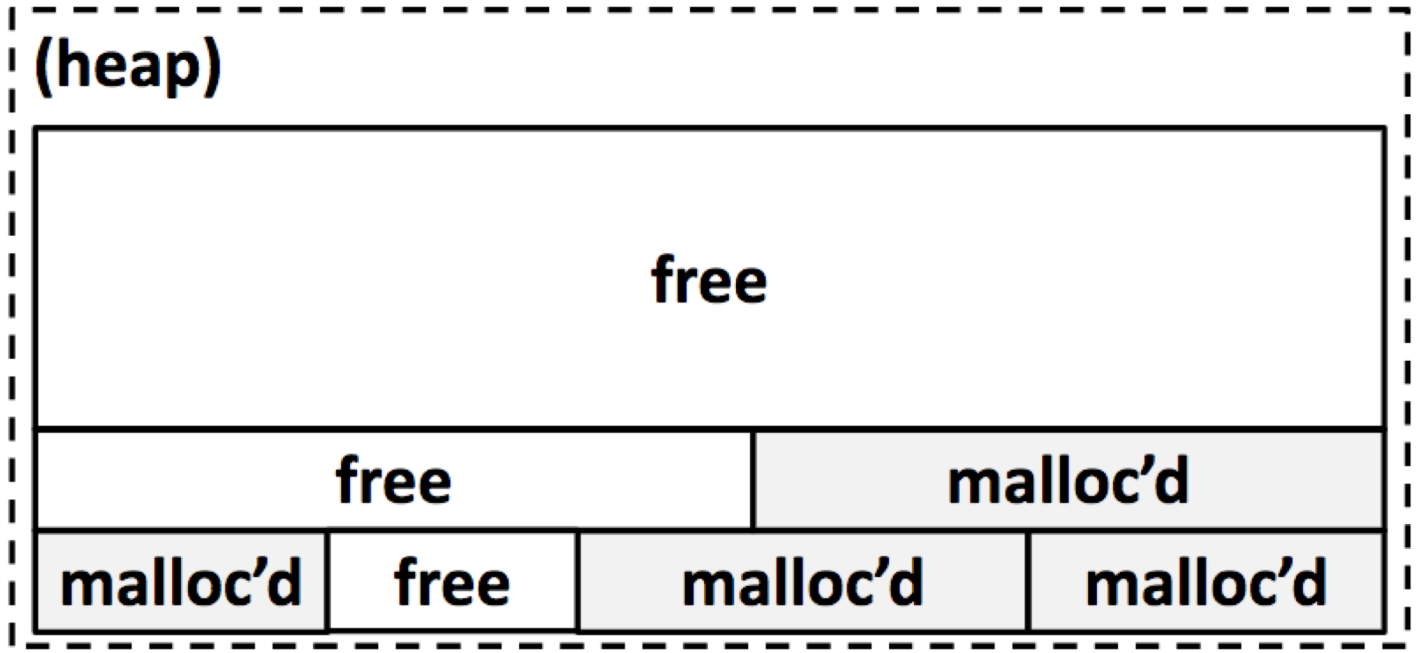
Allocating Memory

- Example



Malloc Internals

- The heap consists of blocks of memory



Concept

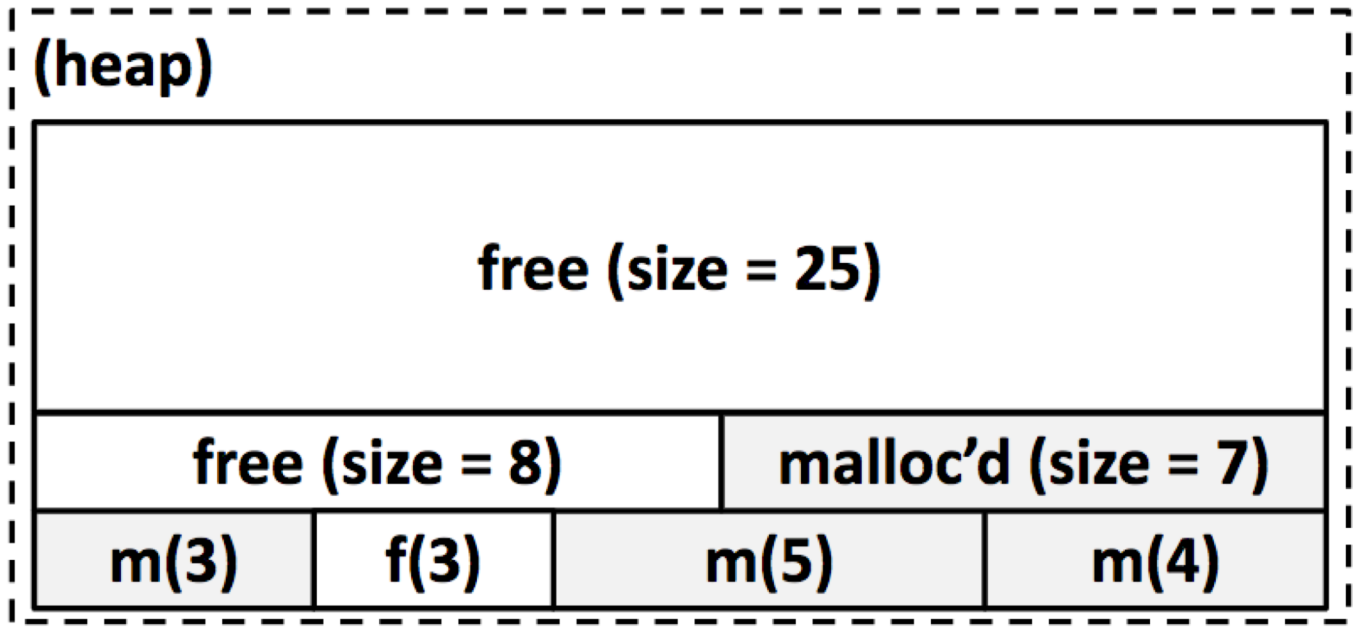
- Overall, malloc does three things:

1. Organizes all blocks and stores information about them in a structured way.
2. Uses the structure made to choose an appropriate location to allocate new memory.
3. Updates the structure when the user frees a block of memory.

This process occurs even for a complicated algorithm like segregated lists.

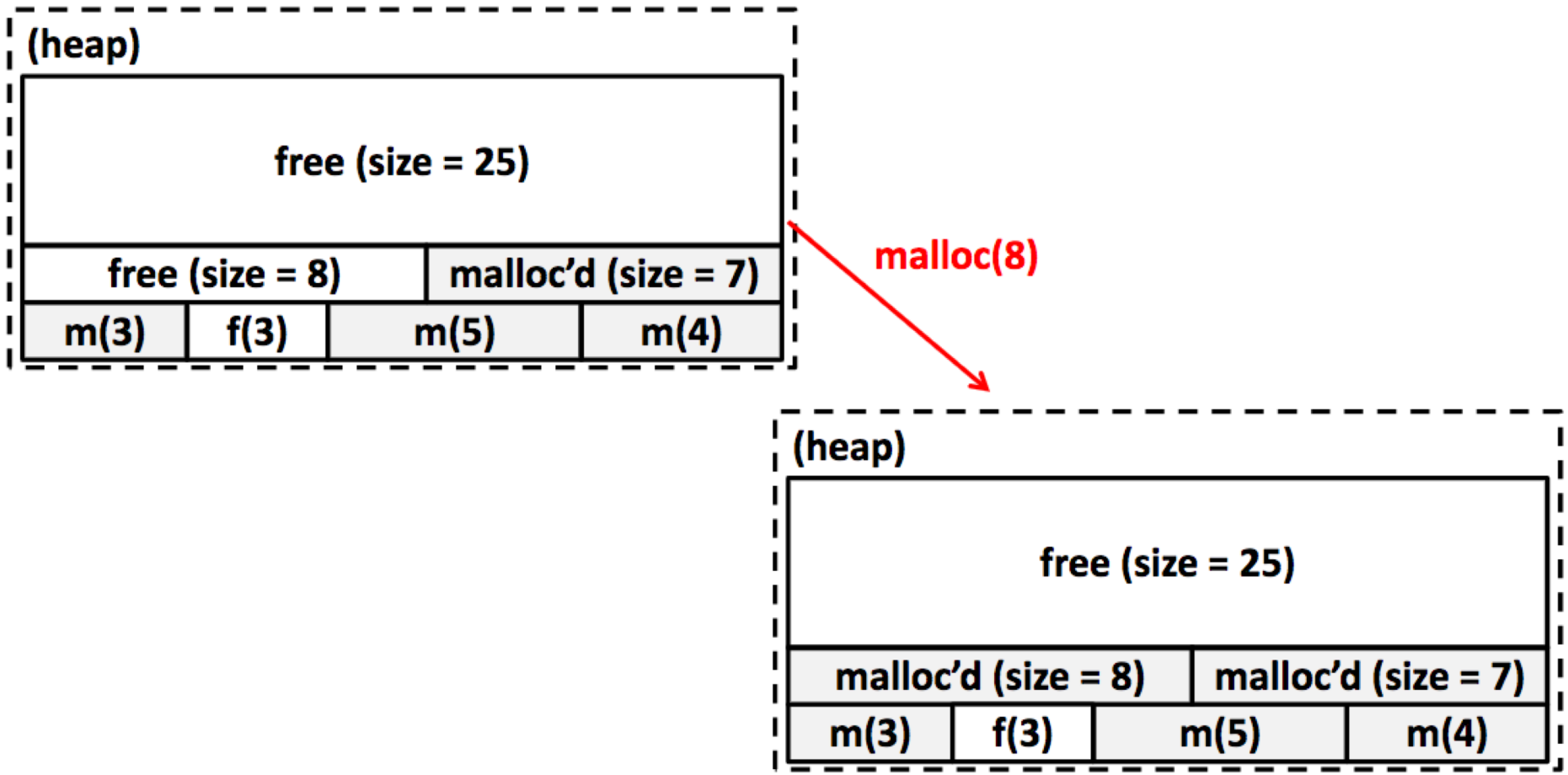
Concept (Implicit list)

1. **Organizes all blocks and stores information about them in a structured way.**



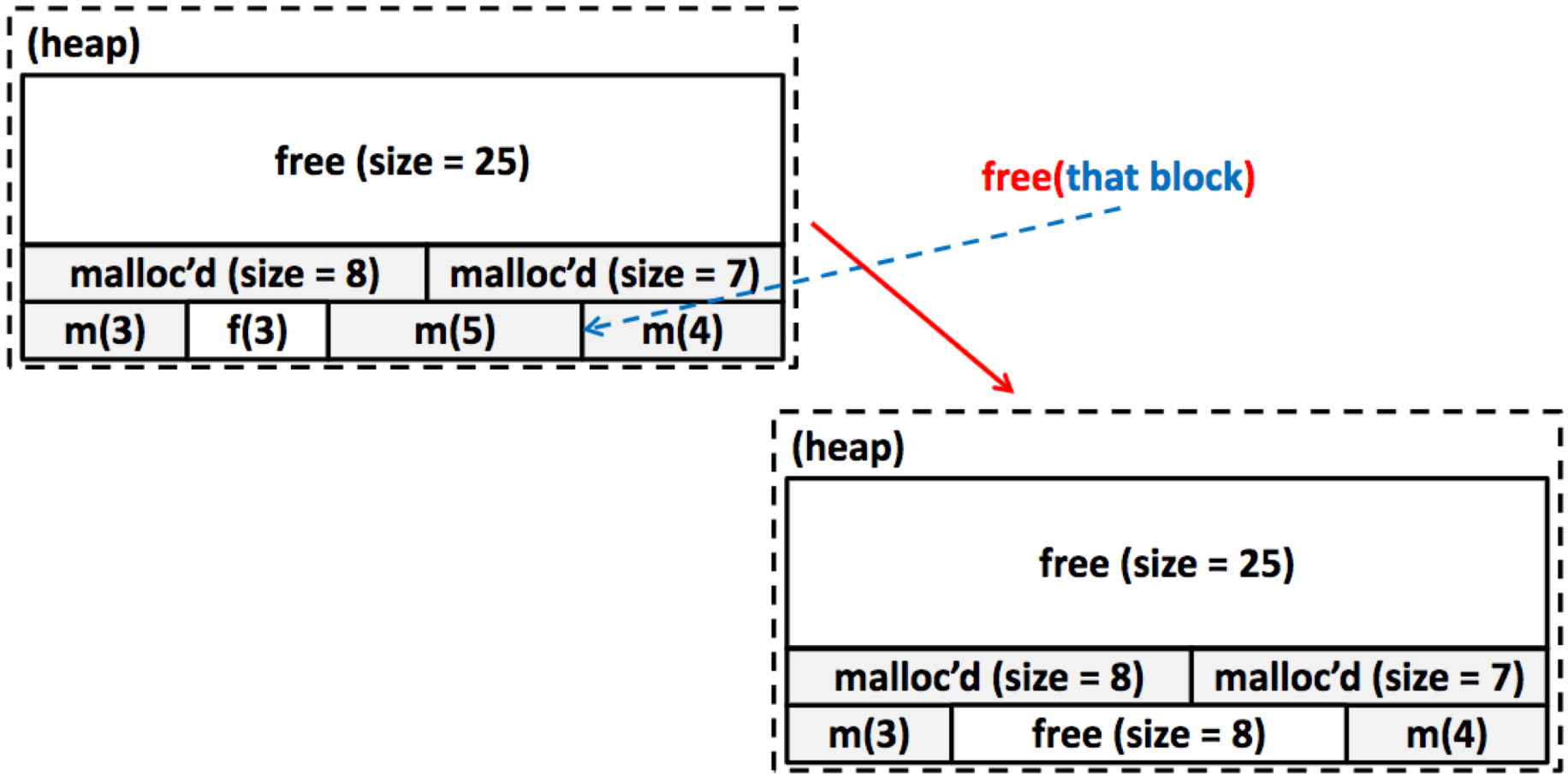
Concept (Implicit list)

2. **Uses the structure made to choose an appropriate location to allocate new memory.**



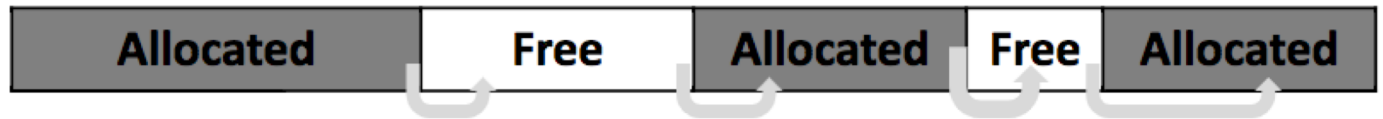
Concept (Implicit list)

3. **Updates the structure when the user frees a block of memory.**



Allocation methods in a nutshell

- **Implicit list: a list is implicitly formed by jumping between blocks, using knowledge about their sizes.**



- **Explicit list: Free blocks explicitly point to other blocks, like in a linked list.**
  - Understanding explicit lists requires understanding implicit lists

- **Segregated list: Multiple linked lists, each containing blocks in a certain range of sizes.**
  - Understanding segregated lists requires understanding explicit lists

Fragmentation

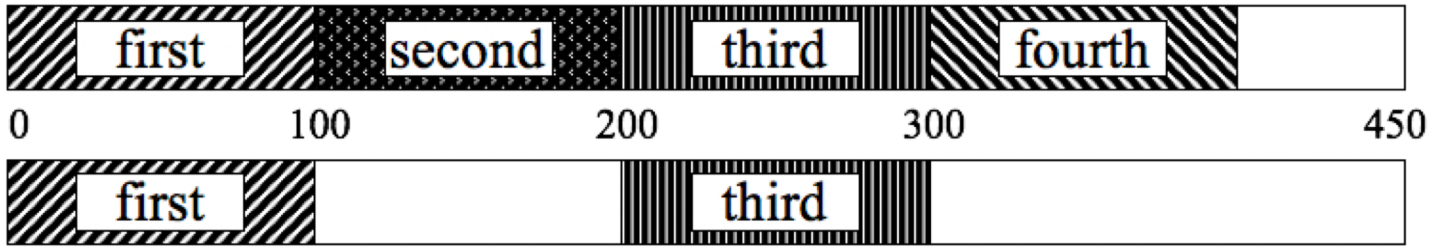
- Segments of memory can become unusable
  - FRAGMENTATION
  - result of allocation scheme
- Two types of fragmentation
  - external fragmentation
    - memory remains unallocated
    - variable allocation sizes
  - internal fragmentation
    - memory is allocated but unused
    - fixed allocation sizes

External Fragmentation

- Imagine calling a series of *malloc* and *free*

```
char* first = malloc(100);
char* second = malloc(100);
char* third = malloc(100);
char* fourth = malloc(100);
free(second);
free(fourth);
char* problem = malloc(200)
```

- 250 free bytes of memory, only 150 contiguous
  - unable to satisfy final malloc request

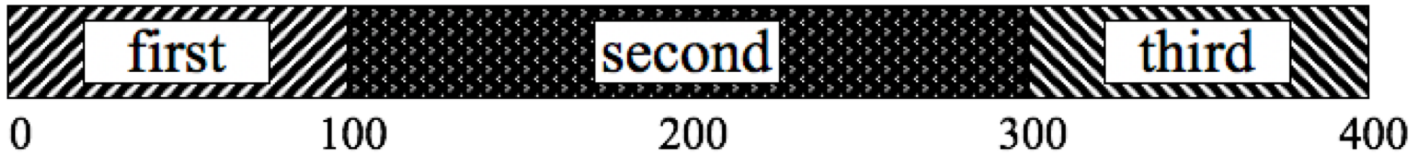


Internal Fragmentation

- Imagine calling a series of *malloc*
  - assume allocation unit is 100 bytes

```
char* first = malloc(90);
char* second = malloc(120);
char* third = malloc(10);
char* problem = malloc(50);
```

- All of memory has been allocated but only a fraction of it is used (220 bytes)
  - unable to handle final memory request



Instructions for the Lab

- Download the *malloclab-handout.tar* file from the Moodle assignment page.
- Start by copying *malloclab-handout.tar* to a directory in which you plan to do your work. Then give the command: ***tar xvf malloclab-handout.tar***
- This will cause a number of files to be unpacked into the directory.
- The only file you will be modifying and handing in is ***mm.c***.
  The ***mdriver.c*** program is a driver program that allows you to evaluate the performance of your solution.
- Use the command *make* to generate the driver code and run it with the command *./mdriver -V*. (The -V flag displays helpful summary information.)
- Looking at the file *mm.c* you'll notice a C structure team into which you should insert the requested identifying information about the one or two individuals comprising your programming team. **Do this right away so you don't forget.**
- When you have completed the lab, you will hand in only one file (*mm.c*), which contains your solution.

Getting Started with the Lab

- Your dynamic storage allocator will consist of the following four functions, which are declared in mm.h and defined in mm.c.
- int mm_init(void);
- void *mm_malloc(size_t size);
- void mm_free(void *ptr);
- void *mm_realloc(void *ptr, size_t size);
- The mm.c file we have given you implements the simplest but still functionally correct malloc package that we could think of.
- Using this as a starting place, modify these functions (and possibly define other private static functions), so that they obey the following semantics:

Semantics

- ***mm_init:*** Before calling mm_malloc mm_realloc or mm_free, the application program (i.e., the trace-driven driver program that you will use to evaluate your implementation) calls mm_init to perform any necessary initializations, such as allocating the initial heap area. The return value should be -1 if there was a problem in performing the initialization, 0 otherwise.
- ***mm_malloc:*** The mm_malloc routine returns a pointer to an allocated block payload of at least size bytes. The entire allocated block should lie within the heap region and should not overlap with any other allocated chunk.
- ***mm_free:*** The mm_free routine frees the block pointed to by ptr. It returns nothing. This routine is only guaranteed to work when the passed pointer (ptr) was returned by an earlier call to mm_malloc or mm_realloc and has not yet been freed.
- ***mm_realloc:*** The mm_realloc routine returns a pointer to an allocated region of at least size bytes with the following constraints.
  - if ptr is NULL, the call is equivalent to mm_malloc(size);
  - if size is equal to zero, the call is equivalent to mm_free(ptr);
  - if ptr is not NULL, it must have been returned by an earlier call to mm_malloc or mm_realloc. The call to mm_realloc changes the size of the memory block pointed to by ptr (the {\em old block}) to size bytes and returns the address of the new block. Notice that the address of the new block might be the same as the old block, or it might be different, depending on your implementation, the amount of internal fragmentation in the old block, and the size of the realloc request. The contents of the new block are the same as those of the old ptr block, up to the minimum of the old and new sizes. Everything else is uninitialized. For example, if the old block is 8 bytes and the new block is 12 bytes, then the first 8 bytes of the new block are identical to the first 8 bytes of the old block and the last 4 bytes are uninitialized. Similarly, if the old block is 8 bytes and the new block is 4 bytes, then the contents of the new block are identical to the first 4 bytes of the old block

Heap Consistency Checker

- Dynamic memory allocators are notoriously tricky beasts to program correctly and efficiently. They are difficult to program correctly because they involve a lot of untyped pointer manipulation. You will find it very helpful to write a heap checker that scans the heap and checks it for consistency.
- Some examples of what a heap checker might check are:
- Is every block in the free list marked as free?
- Are there any contiguous free blocks that somehow escaped coalescing?
- Is every free block actually in the free list?
- Do the pointers in the free list point to valid free blocks?
- Do any allocated blocks overlap?
- Do the pointers in a heap block point to valid heap addresses?
- Your heap checker will consist of the function int mm_check(void) in mm.c. It will check any invariants or consistency conditions you consider prudent. It returns a nonzero value if and only if your heap is consistent. You are not limited to the listed suggestions nor are you required to check all of them. You are encouraged to print out error messages when mm_check fails.

Support Routines

The *memlib.c* package simulates the memory system for your dynamic memory allocator. You can invoke the following functions in memlib.c:

- ***void *mem_sbrk(int incr):*** Expands the heap by incr bytes, where incr is a positive non-zero integer and returns a generic pointer to the first byte of the newly allocated heap area. The semantics are identical to the Unix sbrk function, except that mem_sbrkaccepts only a positive non-zero integer argument.
- ***void *mem_heap_lo(void):*** Returns a generic pointer to the first byte in the heap.
- ***void *mem_heap_hi(void):*** Returns a generic pointer to the last byte in the heap.
- ***size_t mem_heapsize(void):*** Returns the current size of the heap in bytes.
- ***size_t mem_pagesize(void):*** Returns the system's page size in bytes (4K on Linux systems).

Programming Rules

- You should not change any of the interfaces in mm.c.
- You should not invoke any memory-management related library calls or system calls. This excludes the use of malloc, calloc, free, realloc, sbrk, brk or any variants of these calls in your code.
- For consistency with the libc malloc package, which returns blocks aligned on 8-byte boundaries, *your allocator must always return pointers that are aligned to 8-byte boundaries*. The driver will enforce this requirement for you.

Hints

- **Use the mdriver -f option.** During initial development, using tiny trace files will simplify debugging and testing. We have included two such trace files (`short{1,2-bal.rep}) that you can use for initial debugging.
- **Use the mdriver -v and -V options.** The -v option will give you a detailed summary for each trace file. The -V will also indicate when each trace file is read, which will help you isolate errors.
- **Compile with gcc -g and use a debugger.** A debugger will help you isolate and identify out of bounds memory references.
- **Understand every line of the malloc implementation in the textbook.** The textbook has a detailed example of a simple allocator based on an implicit free list. Use this is a point of departure. Don't start working on your allocator until you understand everything about the simple implicit list allocator. That's good for 70% on this assignment.
- **Encapsulate your pointer arithmetic in C preprocessor macros or inline functions.** Pointer arithmetic in memory managers is confusing and error-prone because of all the casting that is necessary. You can reduce the complexity significantly by writing macros or, better yet, inline functions for your pointer operations. See the text for examples and look at the handout for provided samples.
- **Do your implementation in stages.** The first 9 traces contain requests to malloc and free. The last 2 traces contain requests for realloc, malloc, and free. We recommend that you start by getting your malloc and free routines working correctly and efficiently on the first 9 traces. Only then should you turn your attention to the reallocimplementation. For starters, we've built a realloc on top of your existing malloc and free implementations. But to get really good performance, you will need to build a stand-alone realloc.
- **Start early** It is possible to write an efficient malloc package with a few pages of code. However, we can guarantee that it will be some of the most difficult and sophisticated code you have written so far in your career. So start early, and good luck!

Thank you and Good Luck!