Lab 1: My Malloc Library

CS-350: Systems Programming

Instructor: Dr. Dorian Arnold

Computer Science Department, Emory University

Fall 2020

my_malloc library

```
void * my_malloc(uint32_t size)
```

allocates size bytes of memory

```
void my_free(void *ptr)
```

deallocates memory pointed to by `ptr`, previously allocated by `my_malloc()`

```
void coalesce_free_list(void)
```

merges adjacent chunks on the free list into single larger chunks

```
FreeListNode * free_list_begin( void )
```

retrieves the first node of the free list

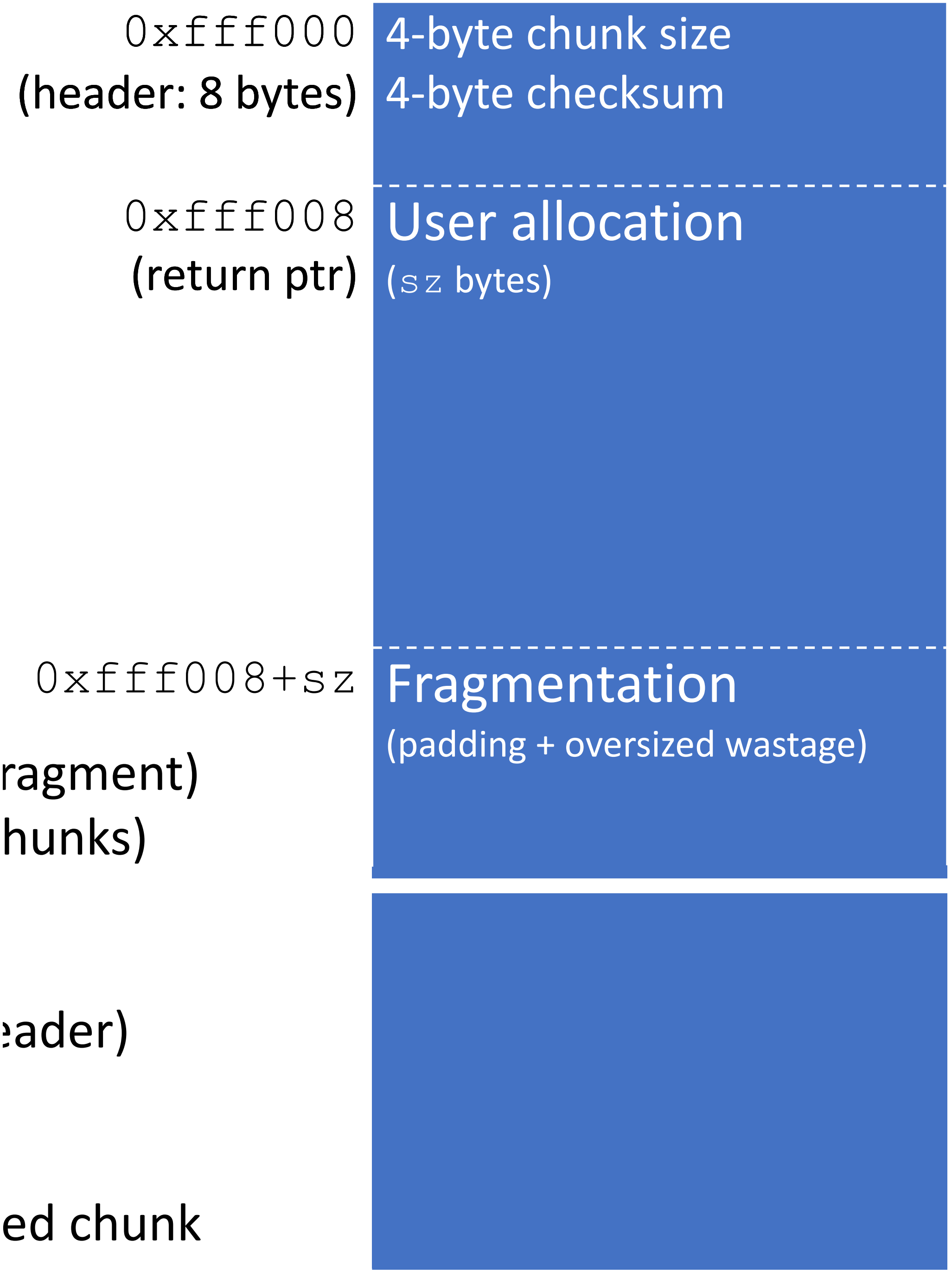
Memory Allocation

`my_malloc(sz)` : returns chunk to caller

1. Find chunk (from heap or free list)

2. Split chunk if too large
   - put remainder on free list

3. Chunk to return
   1. Header (for bookkeeping)
      - 1st 4-bytes: total chunk size (including header+fragment)
      - 2nd 4-bytes: magic number (validates malloc'd chunks)
   2. User allocation: `sz` bytes
      - Return chunk ptr: 8-bytes inside chunk (after header)
   3. Fragmentation:
      - Any necessary padding or wastage from oversized chunk



`0xfff000`
(header: 8 bytes)

4-byte chunk size
4-byte checksum

`0xfff008`
(return ptr)

User allocation
(`sz` bytes)

`0xfff008+sz`

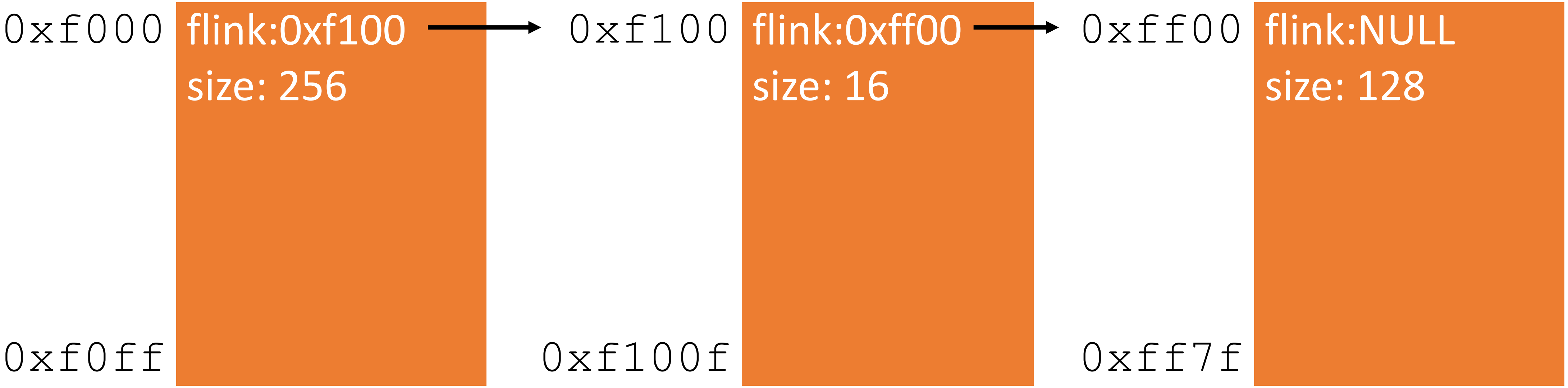
Fragmentation
(padding + oversized wastage)

Free List Management

```
typedef struct freelistnode {
    struct freelistnode *flink;
    uint32_t size;
} * FreeListNode;
```

`my_free( ptr )` places chunk on free list

1. Check for valid checksum
2. Embed `struct freelistnode` at start of chunk
3. Insert free list node into free list



0xf000

flink:0xf100

size: 256

0xf0ff

0xf100

flink:0xff00

size: 16

0xf100f

0xff00

flink:NULL

size: 128

0xff7f

Auxiliary Functions

```
void coalesce_free_list(void)
```

Merges adjacent chunks on the free list into single larger chunks

Free list nodes should be kept in order

No coalescing unless this function is called!

```
FreeListNode * free_list_begin( void )
```

retrieves the first node of the free list

Requirements and Constraints

1. **sbrk() is the only allowed third party library or system call allowed**

2. Always call `sbrk(8192)` except if `my_malloc()` needs more bytes

3. Assume that other library routines also may call sbrk().

4. You may not use more than 8 bookkeeping bytes.

5. You may use **one** global variable for the first free list node

6. Free list should always be sorted in ascending order by chunk address.

7. Use *first fit* strategy to search the free list, i.e. return the first usable chunk.

You may call sbrk(0) to identify the heap's current end.

What I did! (Not necessarily what you have to do)

1. Implemented malloc()
    1. find_chunk(): returns address of appropriately sized chunk to use
        - first only from the heap, then later from the free list first then heap
    2. split_chunk(): returns address of chunk to use
        - if needed, split chunk and put remainder on free list
    3. Bookkeep: place chunk size and magic number in header
    4. Return ptr to user allocation

2. Implemented free()
    1. implemented singly-linked list for `struct freelistnode`
    2. check for magic number
    3. embed struct freelistnode into chunk
    4. insert chunk into free list

3. Implemented coalesce()
    1. wrote function to test if two nodes are adjacent
    2. wrote function to merge two adjacent nodes
    3. traverse free list testing and merging adjacent nodes

Other Hints and Tips

- Start early!
- Don’t start programming until you fully understand the concepts:
  - this lab is complex, but not a lot of code
- Remember, pointer arithmetic is based on pointer type
- Build and test incrementally
- Consider a driver program that validates heap and free list after each malloc()/free()
- Don’t forget “my_errno”
- Memory debuggers (gdb, valgrind, etc.) are your friends