User Level Memory Manager

Serena Hogan

Overview

- Implement a user level memory manager
- Controls memory allocation on the heap
  - Alternative implementations for malloc and free
- Use mmap to request new memory from the system
  - ```
    void* p = mmap(0, getpagesize()*4, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0);
    ```
- Uses a memMan struct to keep track of all of the memory management overhead
- Allocate four pages at a time to minimize the overhead of calling mmap (syscall)
- Buddy memory allocation
    ```

Buddy Memory Allocator

- Start with blocks of page size
- When requesting new memory
  - Split each block in half until it is the smallest it can be while still being large enough to fill the memory request
- Each block is a buddy to the half is it was split from
- When freeing memory
  - Join a block with its buddy, if it is free

Buddy Memory Allocator

Blocks 2^0 are 64K

1. A reqs 34K
2. B reqs 66K
3. C reqs 35K
4. D req 67K
5. B is freed
6. D is freed
7. A is freed
8. C is freed

https://en.wikipedia.org/wiki/Buddy_memory_allocation

<table>
<tr><td>1</td><td colspan="8">2^3</td></tr>
<tr><td></td><td colspan="4">2^2</td><td colspan="4">2^2</td></tr>
<tr><td></td><td colspan="2">2^1</td><td colspan="2">2^1</td><td colspan="4">2^2</td></tr>
<tr><td></td><td>2^0</td><td>2^0</td><td colspan="2">2^1</td><td colspan="4">2^2</td></tr>
<tr><td></td><td>A: 2^0</td><td>2^0</td><td colspan="2">2^1</td><td colspan="4">2^2</td></tr>
<tr><td>2</td><td>A: 2^0</td><td>2^0</td><td colspan="2">B: 2^1</td><td colspan="4">2^2</td></tr>
<tr><td>3</td><td>A: 2^0</td><td>C: 2^0</td><td colspan="2">B: 2^1</td><td colspan="4">2^2</td></tr>
<tr><td>4</td><td>A: 2^0</td><td>C: 2^0</td><td colspan="2">B: 2^1</td><td colspan="2">2^1</td><td colspan="2">2^1</td></tr>
<tr><td></td><td>A: 2^0</td><td>C: 2^0</td><td colspan="2">B: 2^1</td><td colspan="2">D: 2^1</td><td colspan="2">2^1</td></tr>
<tr><td>5</td><td>A: 2^0</td><td>C: 2^0</td><td colspan="2">2^1</td><td colspan="2">D: 2^1</td><td colspan="2">2^1</td></tr>
<tr><td>6</td><td>A: 2^0</td><td>C: 2^0</td><td colspan="2">2^1</td><td colspan="2">2^1</td><td colspan="2">2^1</td></tr>
<tr><td></td><td>A: 2^0</td><td>C: 2^0</td><td colspan="2">2^1</td><td colspan="4">2^2</td></tr>
<tr><td>7</td><td>2^0</td><td>C: 2^0</td><td colspan="2">2^1</td><td colspan="4">2^2</td></tr>
<tr><td>8</td><td>2^0</td><td>2^0</td><td colspan="2">2^1</td><td colspan="4">2^2</td></tr>
<tr><td></td><td colspan="2">2^1</td><td colspan="2">2^1</td><td colspan="4">2^2</td></tr>
<tr><td></td><td colspan="4">2^2</td><td colspan="4">2^2</td></tr>
<tr><td></td><td colspan="8">2^3</td></tr>
</table>

Chunks and Bins

- Chunks keep track of their size and whether they are valid or not
  - Multiply the size by -1 to mark a chunk as valid/invalid
  - Negative sizes denote valid blocks
- Empty chunks will keep track of the previous and next chunk in their bin
  - Payload needs to be large enough to store 2 void* (16 bytes)
- Bins keep track of doubly linked lists of chunks of a specific size that aren’t in use

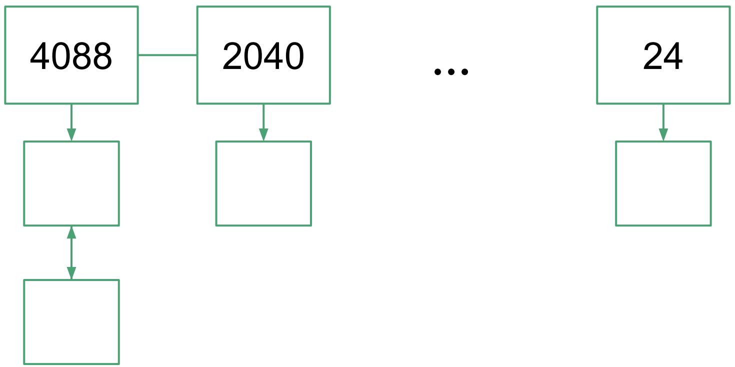
(i) size
Ptr to prev chunk
Ptr to next chunk
...
size

Chunk not in-use

(v) size
payload
size

Chunk in-use

Bins



Memory Manager

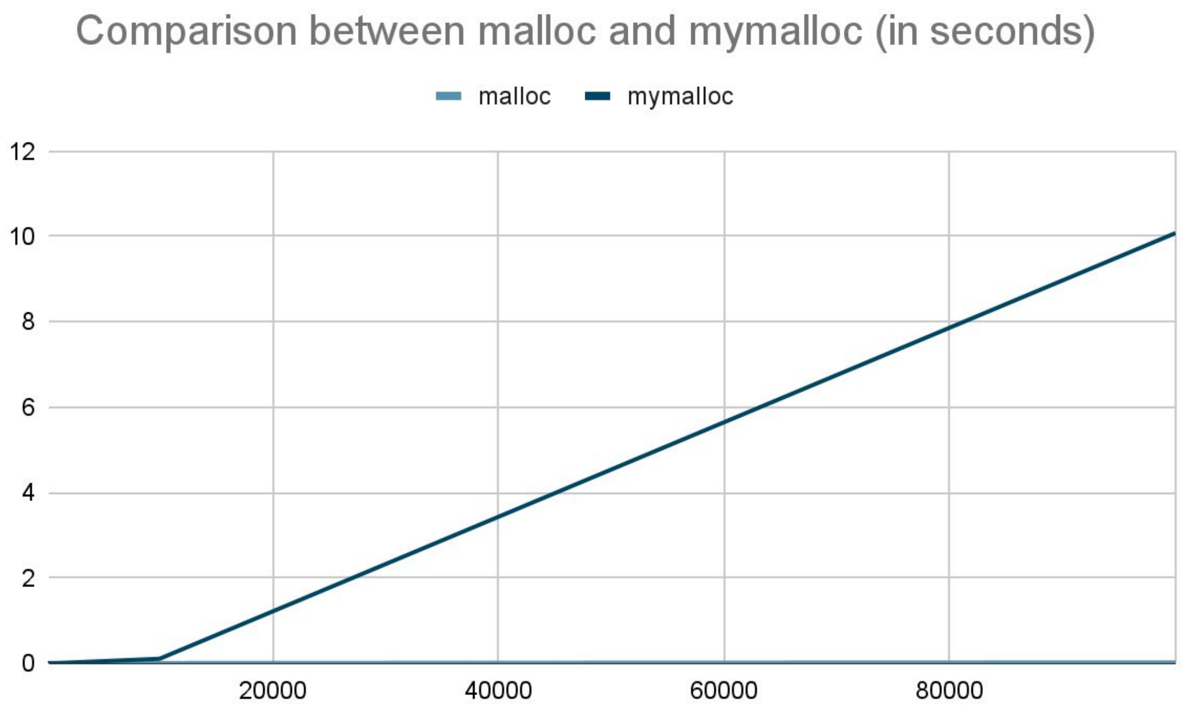
- Each bin has an associated size, which denotes the size of the payload for the chunks in that bin
- Each bin also keeps track of the head and tail of the doubly linked list of chunks
- The memory manager keeps track of the bins of chunks
  - It also keeps track of the highest and lowest address corresponding to the memory it has allocated

```c
typedef struct bin {
    int size;
    void* head;
    void* tail;
} bin;

typedef struct memMan {
    void* highestAddress;
    void* lowestAddress;

    bin miscSzBin;

    bin* bins;
    int nBins;
} memMan;

memMan memoryManager;
```

Fragmentation

- Because chunks will frequently be larger than the requested memory, chunks themselves will have internal fragmentation
    - Smallest chunk is 24 bytes, allocating an int leads to 20 bytes of fragmentation
- Buddy allocation is supposed to prevent internal fragmentation as much as possible by using the smallest chunk that it is a power of 2
    - but it is still inevitable
- All blocks also have 8 bytes of overhead to store the size

Comparison to malloc




```c
clock_t start, end;
start = clock();
int max = 100000;

for(int n = 100; n <= max; n *= 10) {
    int** x = malloc(sizeof(int*)*n);
    for(int i = 0; i < n; i++) {
        x[i] = malloc(sizeof(int));
    }
    for(int i = 0; i < n; i++) {
        free(x[i]);
    }
    free(x);
    end = clock();
    printf("for %d took %lf seconds using malloc\n", n,
        ((double)(end-start)) / CLOCKS_PER_SEC);
}
```

	malloc	mymalloc
100	0.000019	0.000044
1000	0.000218	0.001074
10000	0.001734	0.102336
100000	0.017399	10.080175

Downsides

- Internal fragmentation
- Uses malloc to dynamically allocate space for bins
- Need to call mymalloc_init() and mymalloc_destroy() to set-up and destroy the bins for the memory manager
- Buffer overflow
  - Writing past the end of a array can overwrite metadata of other chunks
- Does not give memory back to the system
  - All of the allocated memory is kept in bins and not returned to the system
  - Calling myfree will mark those chunks as usable but they will not be returned to the system
  - Allocating too much memory may cause the program to crash