CS 341 | Lecture #9
Building an allocator

1. The following allocator will use this linked list structure:

```
01    typedef struct _metadata_entry_t {
02       void *ptr;
03       int size;
04       int free; //0(in use) or 1(available)
05       struct _metadata_entry_t *next;
06    } metadata_entry_t;
```

*Global variable:*

```
07    static metadata_entry_t * head = NULL;
```

2. *Complete* `malloc()`

```
08    void *malloc(size_t size) {
09
10       /* See if we have free space of enough size. */
11       metadata_entry_t *p = head;
12       metadata_entry_t *chosen = NULL;
13
14       while (p != NULL) {
15          if (p->free && _____________________________) {
16             if (chosen == NULL || (chosen && p->size < chosen->size)) {
17                chosen = p;
18             }
19          }
20          p = p->next;
21       }
22
23       if (chosen) {
24          chosen->free = 0;
25          return chosen->ptr;
26       }
27
28       /* Add our entry to the metadata */
29       chosen = sbrk(0);
30       sbrk(sizeof(metadata_entry_t));
31       chosen->ptr = sbrk(0);
32       if (sbrk(size) == (void*)-1) {
33          return NULL;
34       }
35       chosen->size = size;
36       chosen->free = 0;
37
38       chosen->next = head;
39       head = chosen;
40       return chosen->ptr;
41    }
```

3. Complete `free()`

```
01    void free(void *ptr) {
02       if (!ptr) return;
03
04
05          metadata_entry_t *p = _____________________
06          while (p) {
07             if (p->ptr == ptr) {
08
09             }
10             p = p->next;
11          }
12
13          return;
14    }
```

Which placement algorithm does this malloc( )use?
Is calling sbrk 4 tims necessary?
What is the order of growth running time for this implementation of free?

4 i) Why does this implementation suffer from false fragmentation?
ii) When should we split blocks?

iii) Does this implementation use an explicit or implicit linked list?

5. How would you change `malloc()` to use a *first-fit* placement allocation?

```
01       while (p != NULL) {
02          if (p->free && _____________________________) {
03             if (chosen == NULL || (chosen && p->size < chosen->size)) {
04                chosen = p;
05             }
06          }
07          p = p->next;
08       }
```

## 8. Towards a better allocator

Implementing `realloc` & improving performance of `free`()

Hint: Can we ensure this structure is immediately before the user's pointer?

```
01    typedef struct _metadata_entry_t {
02       void *ptr;
03       int size;
04       int free;
05       struct _metadata_entry_t *next;
06    } metadata_entry_t;
```

We want an O(1) deallocator!

```
01    void free(void*user) {
02      if(user == NULL) return; // No-op
03       ?
```

End of the allocator challenge?

1. Block Spitting & Block Coalescing
2. Memory pools
3. Advanced: Slab allocator and Buddy allocator
4. Internal vs External Fragmentation
5. How we use Boundary Tags to implement coalescing?

## 9. Puzzle:

Complete this code to read in values from stdin into heap memory. Can you beat CS225 code by using C and `realloc` to increase the size of the array? Fix any errors you notice.

```
01  #define quit(mesg) {puts(mesg); exit(1);}
02
03  size_t capacity = 256;
04  size_t count =  0;
05  int* data = malloc( capacity );
06  if( ! data ) quit("Out of memory");
07
08  while( !feof(stdin) && !ferror(stdin)) {
09    if( count == capacity) {
10       capacity *= 2;
11
12    }
13    if( fscanf(stdin, "%d", data+count) != 1) break;
14    count++;
15  }
16  // can now reduce capacity to the number actually read
17  printf("%d values read",(int) count);
18  data = realloc(data, count);
```