CS 241 | Lecture #10
Building an allocator II

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

```
01      typedef struct _metadata_entry_t {
02          void *ptr;
03          size_t size;
04          int free; //0(in use) or 1(available)
05          struct _metadata_entry_t *next;
06      } entry_t;
07
08      static entry_t* head = NULL;
09
```

2. Implement an efficient realloc to avoid memory copying when possible?

Assume the above entry_t structure is immediately before the user's pointer

```
01      void* realloc(void *old, size_t newsize) {
```

3. Instrumenting malloc
Case study: Fragmentation & Memory overhead & utilization?

How can we modify our malloc implementation so that we write an instrumentation function below to print how efficient our memory allocator is? "123456 bytes allocated. 280 byte overhead. 352 unavailable bytes in 6 fragments"

```
01      void printMallocStats() {
02
```

4. Memory alignment and BUS Signals?
... aka why malloc writers care about CPUs

... what is natural alignment?

... How can we round up allocations to nearest 16 bytes?

5. Block Coalescing & Thinking in sizeof(size_t) blocks...

Goodbye bytes. Memory = one big “array” of size_t entries

Use Knuth Boundary Tags:

100	.....	100	64	.....	64	132	.....	132

malloc(size_t request_bytes){

int request_blocks = request_bytes / 8? Is this good?

// enough space -

void* ptr = sbrk(

6. Implementing Canaries

How (and when can we detect buffer overflow/ underflow using boundary tags? Are there other canaries?

7. Fast Memory pools

static char buffer[10000];

size_t used=0;

void* malloc(size_bytes) {

}

void free_all_the_things() {

}

8. How can I beat malloc?

a) Efficiency of representation

b) Speed of allocation

c) Speed of "recycling"

d) Utilization of memory