CS 341 | Lecture #10
Building an allocator II

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

```
typedef struct _metadata_entry_t {
   void *ptr;
   size_t size;
   int free; //0(in use) or 1(available)
   struct _metadata_entry_t *next;
} entry_t;

static entry_t* head = NULL;
```

*size_t requested*

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

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

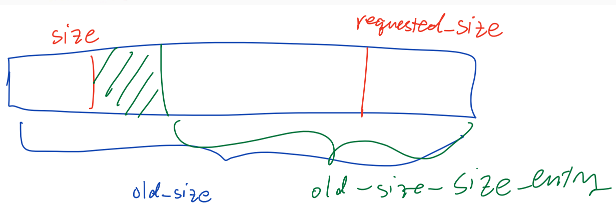
```
void* realloc(void *old, size_t newsize) {
if(old == NULL) { return malloc(newsize);}
entry_t* entry = ((entry_t*)old) -1;
assert( entry->ptr == old);
assert( entry->free == 0);
ssize oldsize = entry->size;
if( oldsize > 2*newsize && (oldsize-newsize)> 1024/*THRESHOLD*/) {
    entry_t *newentry = entry + newsize;
    newentry->ptr = newentry + 1;
    newentry->free = 1;
    newentry->size = newsize- oldsize - sizeof(entry);

    newentry->next = entry->next;
    entry->next = newentry;

}
if( oldsize > newsize) {
   return;
}
void* result = malloc(newsize);
ssize_t minsize = min(newsize, oldsize);
memcpy(result, old, minsize);
free(old);
```

*optimize*

*→ wrong pointer arithmetic*




## 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"

*we can add another variable in entry_t so to record requested bytes.*

```
void printMallocStats() {
size_t allocated_bytes = 0;
size_t entry_count = 0;
size_t unavailable = 0;
size_t available_fragments;
entry_t * p = head;
while(p) {
    if(p->free ==0) { allocated_bytes += p->size;}
    if(p->free ==0) { unavailable +=  p->size - p->requested;}
    if(p->free) available_fragments ++;
    entry_count++;
    p=p->next;
}
size_t overhead_bytes = entry_count * sizeof(entry_t);
```

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

... what is natural alignment?

*int * p = malloc(...)+3 → end up in weird odd address which could be slow or even cause crashes.*

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

*size_t r = (size+15)/16 (number of blocks of size 16)*

*when size=0, r=0*

*when 0<size≤15, r=1*

*⋮*

## 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){

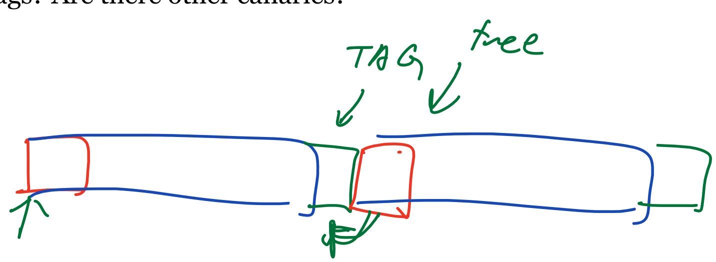
int request_blocks = request_bytes / 8? Is this good?

// 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* result = buffer + used
    used += bytes
    return result;
}

void free_all_the_things() {
    used = 0
}
```

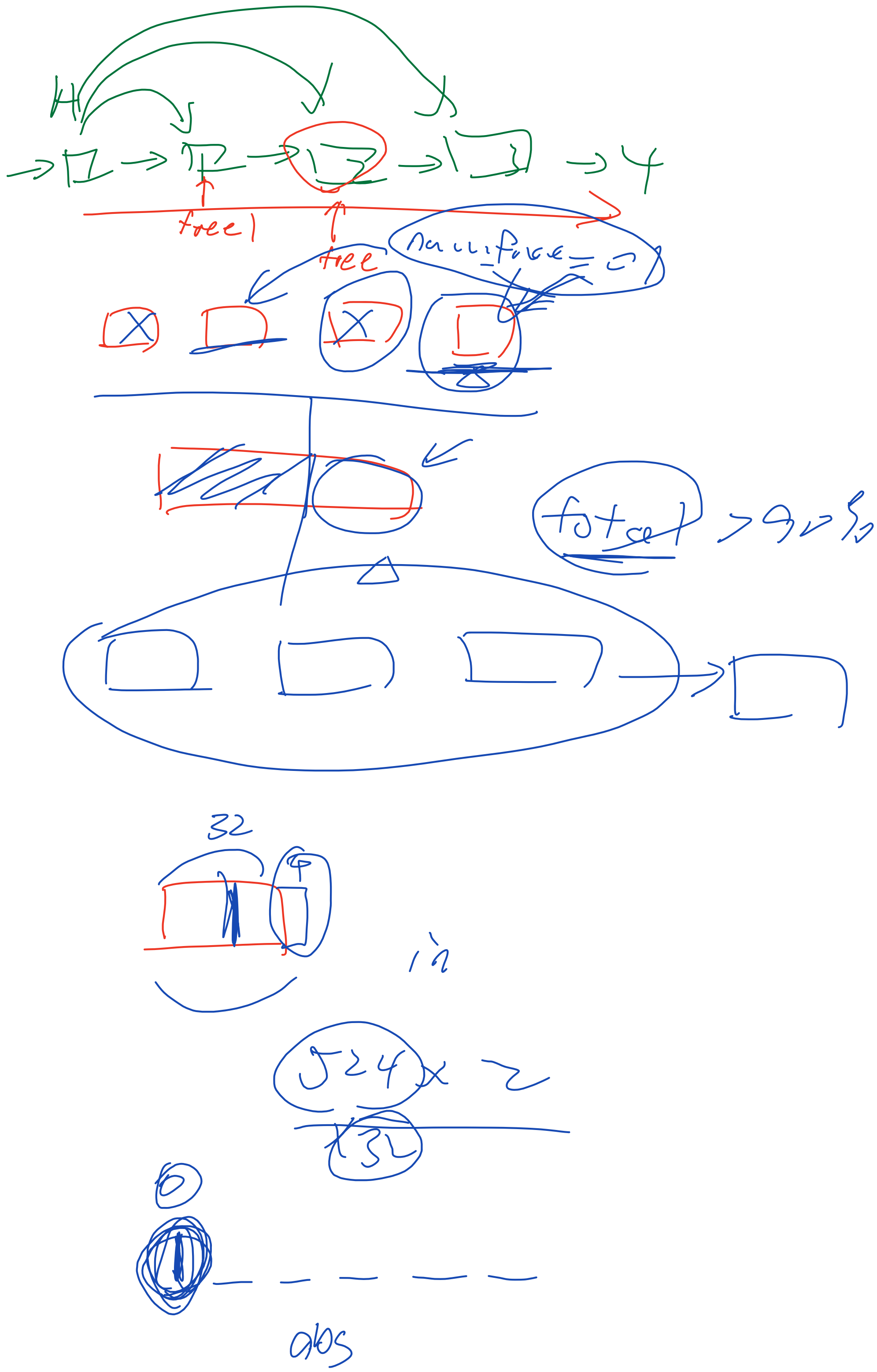
## 8. How can I beat malloc?

a) Efficiency of representation

b) Speed of allocation

c) Speed of "recycling"

d) Utilization of memory



free
free
total
32
abs