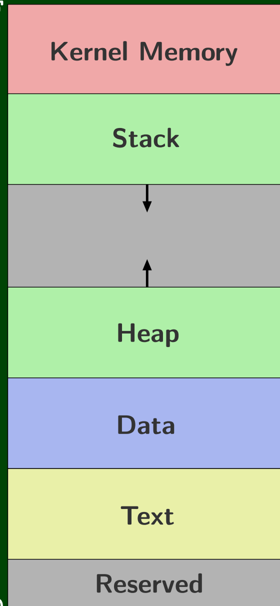
CS 24

Introduction to Computing Systems

Dynamic Memory

Outline


0xFFFFFFFFFFFF
Kernel Memory
Stack
Heap
Data
Text
Reserved
0x000000000000

What Does `malloc` Do?

An **allocator** maintains heap as collection of variable sized blocks, which are either allocated or free.

- `void *malloc(size_t size)`: On success, returns a pointer to a memory block of at least size bytes aligned to a 16-byte boundary (on `x86-64`). If `size == 0`, returns `NULL`. On failure, returns `NULL`.
- `void free(void *p)`: Returns the block pointed at by `p` to pool of available memory. `p` must come from a previous call to `malloc`, `calloc`, or `realloc`.
- `calloc`: Version of `malloc` that initializes allocated block to zero.
- `realloc`: Changes the size of a previously allocated block.

Getting Memory from the OS: `sbrk`

```
NAME
     sbrk -- change data segment size

SYNOPSIS
     #include <unistd.h>

     void *sbrk(int incr);

RETURN VALUE
     The sbrk function returns a pointer to the base of
     the new storage if successful; otherwise -1 with
     errno set to indicate why the allocation failed.
```

“Slide Code”

Warning: The code in these slides is all “slide code”. It generally is not the quality we expect of you, because it has to fit on a slide.

The full code is provided on the website.

Allocator Requirements

- Correctness
  - Can't control number or size of allocated blocks
  - Must respond immediately to `malloc` requests
  - Must allocate blocks from free memory
  - Must align blocks so they satisfy all alignment requirements
  - Can manipulate and modify only free memory
  - Can't move the allocated blocks once they are `malloc`ed
- Performance
  - Throughput: How many requests can be completed per unit time?
  - Utilization: How much of the heap is used for program data?

Attempt 0: Bump Allocator


```
1 void *malloc(size_t size) {
2    return sbrk(size);
3 }
4
5 void free(void *ptr) {
6
7 }
```

Attempt 1: "Simple Implicit List"


```
1  typedef struct {
2     bool is_allocated;
3     word_t size;
4     uint8_t payload[];
5  } block_t;
6
7  static block_t *mm_heap_first = NULL;
8  static block_t *mm_heap_last = NULL;
9
10 void *malloc(size_t size) {
11    size_t asize = round_up(size + D_SIZE, D_SIZE);
12    block_t *block = find_fit(asize);
13    if (!block) {
14       block = mm_heap_last = sbrk(asize);
15       block->size = asize;
16    }
17
18    block->is_allocated = true;
19    return block->payload;
20 }
21
22 void free(void *ptr) {
23    block_t *block = (block_t *)(ptr - offsetof(block_t, payload));
24    block->is_allocated = false;
25 }
```

Saving Space


```
1 typedef struct {
2     bool is_allocated;
3     word_t size;
4     uint8_t payload[];
5 } block_t;
```

```
1 typedef struct {
2     word_t header;
3     uint8_t payload[];
4 } block_t;
```

```
 1 static size_t get_size(block_t *block) {
 2     return block->header & ~0xF;
 3 }
 4
 5 static void set_header(block_t *block, size_t size, bool is_alloc) {
 6     block->header = size | is_alloc;
 7 }
 8
 9 static bool is_allocated(block_t *block) {
10     return block->header & 0x1;
11 }
```

Your mental health is important.

Please spend the effort and time to take care of yourself.

Finding a Free Block

- First fit:
  - Search list from beginning, choose first free block that fits:
  - Can take linear time in total number of blocks (allocated and free)
  - In practice it can cause "splinters" at beginning of list
- Best fit:
  - Search the list, choose the best free block: fits, with fewest bytes left over
  - Keeps fragments smallusually improves memory utilization
  - Will typically run slower than first fit
  - Still a greedy algorithm. No guarantee of optimality

Coalescing

- When do we coalesce?
- What do we coalesce with?
- How do we coalesce with the previous block?

Explicit List

Summary of Trade-Offs

- Free-List Implementation:
  - Implicit List
  - Explicit List
  - Segregated List
  - Balanced Tree
- Search Strategy:
  - First-fit, next-fit, best-fit, etc.
  - Trades off lower throughput for less fragmentation
- Splitting policy:
  - When do we go ahead and split free blocks?
  - How much internal fragmentation are we willing to tolerate?
- Coalescing policy:
  - Immediate coalescing: coalesce each time free is called
  - Deferred coalescing: try to improve performance of free by deferring coalescing until needed.