CS354: Machine Organization and Programming

Lecture 34

Friday the November 20^{th} 2015

Section 2

Instructor: Leo Arulraj

Class Announcements

Programming Assignment 4 is due on coming Wednesday 11/25 . If you have not yet started on it, get started and put in more effort.

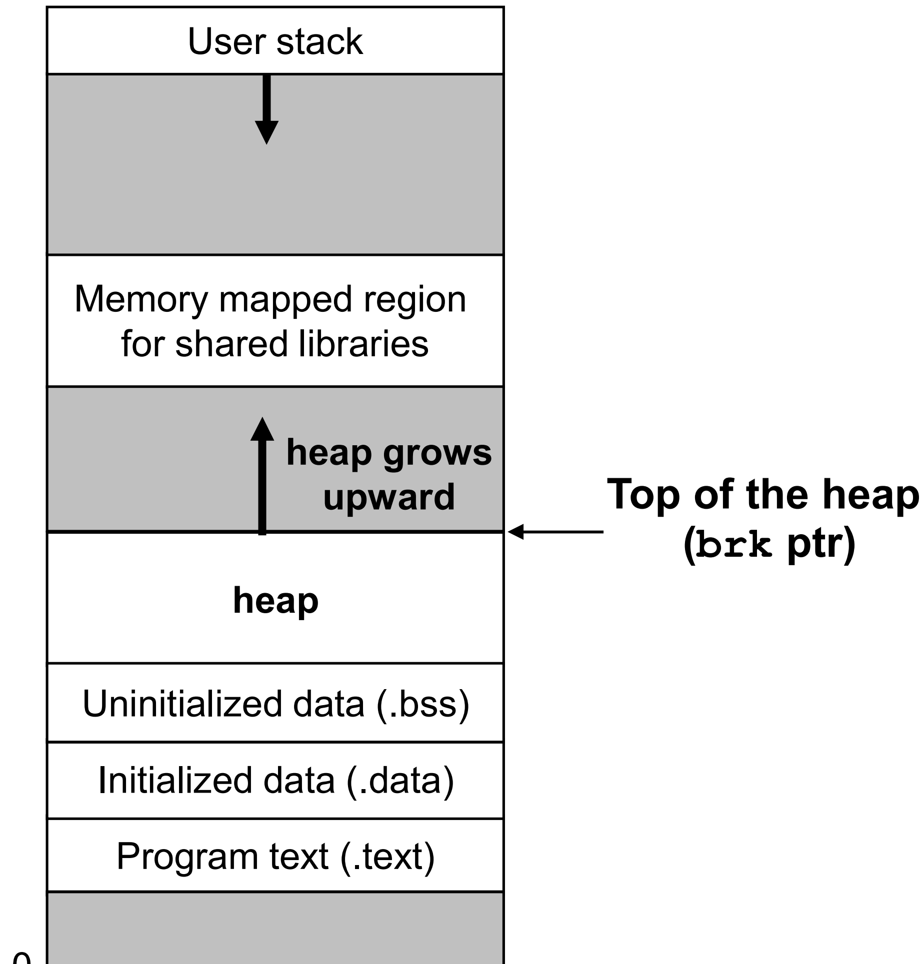
Please use piazza or come see us during office hours, if you need help.

Lecture Overview

1. malloc and free functions
2. Why Dynamic Memory Allocation?
3. Allocator requirements and goals
4. Fragmentation
5. Implementation issues

Dynamic Memory Allocation

Heap space as part of memory layout along with Kernel's "brk" pointer.



Dynamic Memory Allocation

Allocators maintain the heap as a collection of various-sized blocks.

Each block is a contiguous chunk of virtual memory that is either allocated or free.

Each free block is available for allocation.

Each allocated block needs to be freed before it can be reallocated.

Dynamic Memory Allocation

Explicit allocators: requires application to explicitly free any allocated blocks. E.g. C, C++

Implicit allocators: allocator detects when an allocated block is no longer being used by the application and then frees it during "garbage collection" process. E.g. Java (high level language)

We will be focusing on explicit allocators.

malloc() , free() and sbrk()

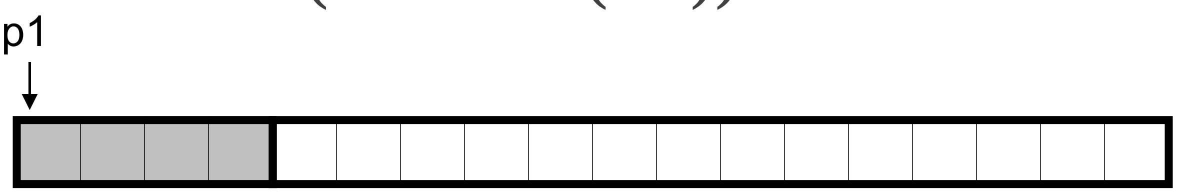
void *malloc(size_t size) : returns pointer to allocated memory on success. calloc , realloc variants.

void free(void *ptr) : frees memory previously allocated using malloc()

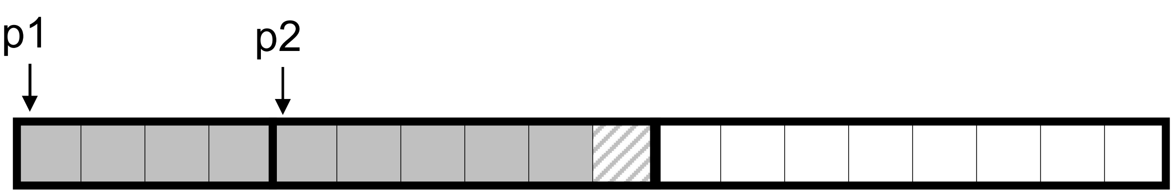
void *sbrk(intptr_t incr) : grows or shrinks the heap by adding incr to kernel's brk pointer.

Allocating and freeing blocks

After p1 = malloc(4*sizeof(int))



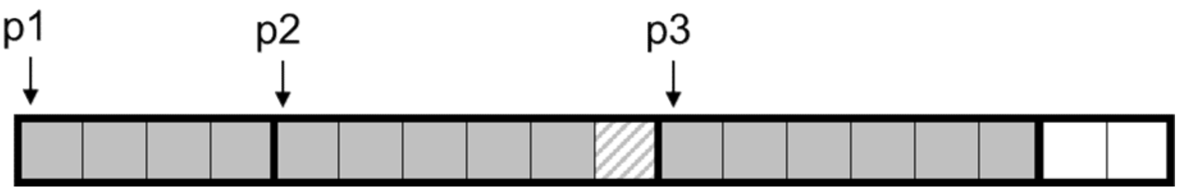
After p2 = malloc(5*sizeof(int))



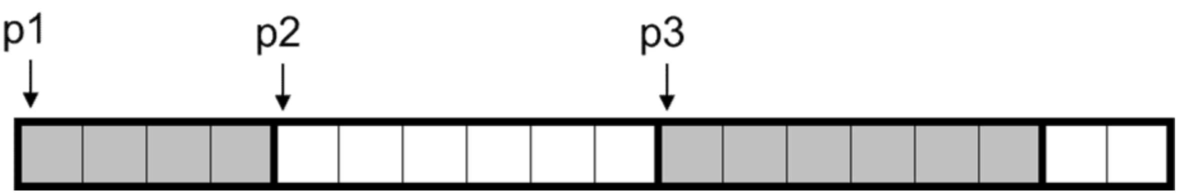
Six blocks allocated for 8-byte alignment

Allocating and freeing blocks

After p3 = malloc(6*sizeof(int))

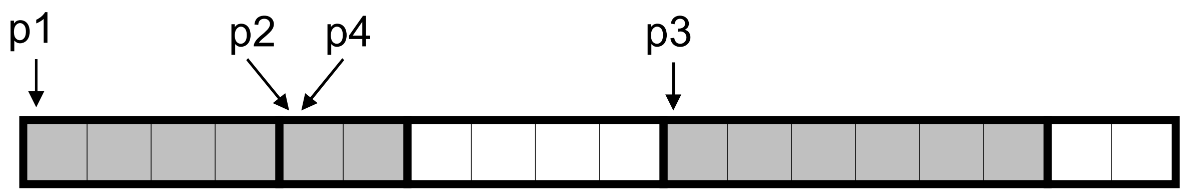


After free(p2)



Allocating and freeing blocks

After p4 = malloc(2*sizeof(int))



Why Dynamic Mem. Allocation?

Consider an example of allocating an array?

int array[MAXN];

Resizing is hard.

Instead, programmer can use malloc() and free()

Allocator constraints

1. Handling arbitrary request sequences
2. Making immediate responses to requests
3. Using only the heap
4. Aligning blocks (alignment requirement)
5. Not modifying allocated blocks

Allocator requirements and goals

1. Goal 1: Maximizing throughput which is defined as the total number of requests that the allocator completes per unit of time.

2. Goal 2: Maximizing memory utilization which is defined after a certain number of alloc & free requests as total memory requested by the alloc() requests that are not yet freed divided by the total heap space used right now.

Internal Fragmentation

Occurs when an allocated block is larger than the payload.

Can happen due to alignment restrictions or due to a minimum allocation unit enforced by the allocator.

Easy to quantify: sum of the differences between sizes of the allocated blocks and their payloads.

External Fragmentation

Occurs when there is enough aggregate free memory to satisfy an allocate request but no single free block is large enough to handle the request.

Occurs due to repeated alloc() and free() that lead to small free spaces.

Difficult to quantify and impossible to predict. Its effect depends on future request sizes too!

Implementation Issues

1. Free block organization: How do we keep track of free blocks?
2. Placement: How do we choose an appropriate free block in which to place a newly allocated block?
3. Splitting: After we place a newly allocated block in some free block, what do we do with the remainder of the free block?
4. Coalescing: What do we do with a block that has just been freed?

Free list

All free blocks are kept organized as part of a "free list" by the allocator.

Upon a new alloc() request, a free block that can hold the requested memory size is chosen and used.

Where to store the free list itself?