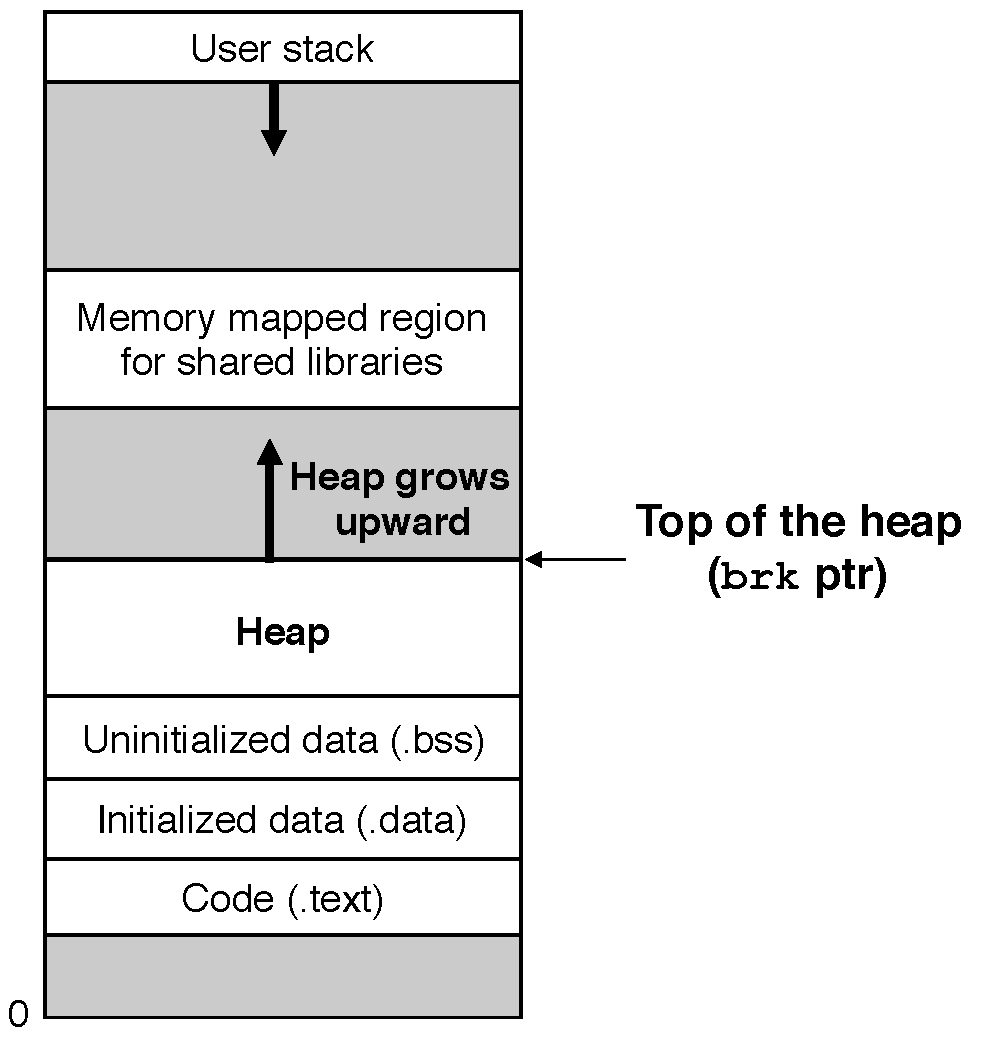
Week 6 Lecture

# Dynamic Memory Allocation (B&O 9.9-9.11)

## Recall from an earlier lecture



## Overview

* **Dynamic memory allocators** maintain an area of a process’s virtual memory, which is known as the **heap**.
* An allocator maintains the heap as a collection of various sized **blocks**.
* Each block is a contiguous chunk of virtual memory that is either **allocated** or **free**.
  + **Allocated block** – a block that has been reserved explicitly for use by the application
  + **Free block** – a block that is available to be allocated
* There are two basic allocator styles, **explicit allocators** and **implicit allocators**.
  + Both require the application to explicitly allocate blocks.
  + The difference between them is which entity they hold responsible for freeing allocated blocks after the information stored in those blocks is no longer being used by the application.
    - **Explicit allocators** – require the application to explicitly free any allocated blocks
      * **Example**: The C Standard Library’s malloc package
        + Allocate blocks by calling the **malloc** function.
        + Free blocks by calling the **free** function.
    - **Implicit allocators** – require the allocator to detect when an allocated block is no longer being used by the program and then free the block
      * These are also known as garbage collectors
      * **Example**: Java’s garbage collector
* If an application requests a block of bytes, the resulting allocated block has a **payload** of bytes.

## Explicit Allocators

* **Example**: Allocating and freeing blocks with malloc package
  + Package functions
    - **void \*malloc(size\_t size);**
      * If successful:
        + Returns a pointer to a memory block of at least size bytes
        + If size == 0, returns NULL
      * If unsuccessful: 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 or realloc.
    - **void \*realloc(void \*p, size\_t size);**
      * Changes size of block p and returns pointer to new block.
      * Contents of new block unchanged up to min of old and new size.
    - **void \*calloc(size\_t nmemb, size\_t size);**
      * Version of malloc that initializes allocated block to zero.
  + malloc example

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  void foo(int n) {  int i, \*p;  /\* Allocate a block of n ints \*/  p = (int \*) malloc(n \* sizeof(int));  if (p == NULL) {  perror("malloc");  exit(0);  }  /\* Initialize allocated block \*/  for (i=0; i<n; i++)  p[i] = i;  /\* Return allocated block to the heap \*/  free(p);  } |

* Assumptions we will make in this lecture:
  + Memory is word addressed.
  + Word size is equal to sizeof(int).

### Allocator Requirements/Constraints

* Must be able to handle arbitrary request sequences
  + The allocator cannot make assumptions about the ordering of allocate and free requests since there isn’t necessarily any meaningful pattern there.
    - As long as the application frees what it allocates and allocates before it tries to free, then the application may make these request in an arbitrary sequence.
* Must respond immediately to requests
  + This means that the allocator is not allowed to reorder or buffer requests in order to improve performance.
* Must align blocks in such a way that they can hold any type of data object
* Must only manipulate and change free blocks
  + Allocators cannot modify or move blocks once they are allocated
    - This means that techniques such as compaction of allocated blocks are not allowed.

### Allocator Goals

* Maximize throughput
  + **Throughput** – the number of requests completed per unit time
  + In general this goal is achieved by minimizing the average time to satisfy allocate and free requests
* Maximize memory utilization
  + **Memory utilization** – essentially just how efficiently the allocator uses the heap
    - The total amount of memory allocated to the application divided by the heap size

### Fragmentation

* **Fragmentation** – a phenomenon which occurs when otherwise unused memory is not available to satisfy allocate requests
* This is a primary cause of poor heap utilization.
* There are two types of fragmentation, **internal fragmentation** and **external fragmentation**.

#### Internal Fragmentation

* This occurs when the size of an allocated block is larger than the payload size.
* Why does this happen?
  + There are a number of reasons why it could happen.
  + Two examples are as follows.
    - The implementation of the allocator imposes a minimum size for allocated blocks that is greater than the size of the requested payload.
    - The allocator might increase block size in order to satisfy alignment constraints
* Quantifying internal fragmentation
  + It is simply the sum of the differences between the sizes of the allocated blocks and their payload.

#### External Fragmentation

* This 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.
* Quantifying external fragmentation
  + Much more difficult to quantify than internal fragmentation is.
  + This is because it depends not only on the pattern of previous requests and the allocator implementation but also the pattern of future requests.

### Allocator implementation issues

* Difficult to find a balance between throughput and memory utilization.
* Overview of what issues to consider:
  + Free block organization
    - How do we keep track of free blocks?
  + Placement of allocated blocks
    - How do we choose an appropriate free block in which to place a newly allocated block?
  + Splitting free blocks
    - After we place a newly allocated block in some free block, what do we do with the remainder of the free block?
  + Coalescing
    - What do we do with a block that has just been freed?

#### Implicit free lists

* Free block organization
  + Each block contains a header with some extra information.
    - Allocated bit indicates whether block is allocated or free.
    - Size field indicates entire size of block (including the header)
    - Trick: Allocation bit is just the high-order bit of the size word
* Placement of allocated blocks
  + No explicit structure tracking location of free/allocated blocks.
    - Rather, the size word (and allocated bit) in each block form an implicit “block list”
  + How do we find a free block in the heap?
    - Start scanning from the beginning of the heap.
    - Traverse each block until (a) we find a free block and (b) the block is large enough to handle the request.
    - This is called the **first fit** strategy.

