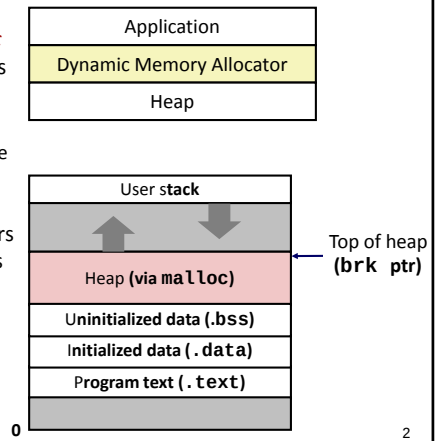
Dynamic Memory Allocation: Basic Concepts

Kai Shen

1

Dynamic Memory Allocation

- Programmers use *dynamic memory allocators* (such as **`malloc`**) to acquire memory at run time.
  - For data structures whose size is only known at runtime.
- Dynamic memory allocators manage an area of process virtual memory known as the *heap*.



2

Dynamic Memory Allocation

- Allocator maintains heap as collection of variable sized *blocks*, which are either *allocated* or *free*
- Types of allocators
  - ***Explicit allocator***: application allocates and frees space
    - E.g., **`malloc`** and **`free`** in C
  - ***Implicit allocator:*** application allocates, but does not free space
    - E.g. garbage collection in Java, ML, and Lisp
- We discuss explicit memory allocation today

3

The `malloc` Package

```
#include <stdlib.h>
void *malloc(size_t size)
```

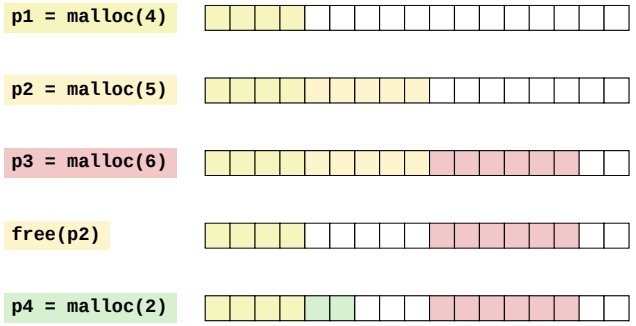
- Successful:
  - Returns a pointer to a memory block of at least **`size`** bytes (typically) aligned to 8-byte boundary
  - If **`size==0`**, returns NULL
- Unsuccessful: returns NULL (0) and sets **`errno`**

```
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`**

4

Allocation Example



5

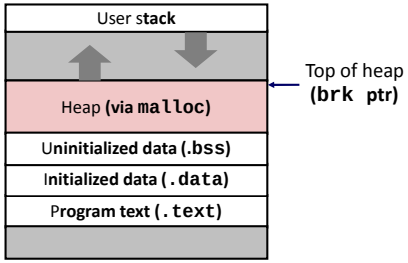
Constraints

- Applications
  - Can issue arbitrary sequence of **`malloc`** and **`free`** requests
  - **`free`** request must be to a **`malloc`**'d block
- Allocators
  - Must respond immediately to **`malloc`** requests
  - Can manipulate and modify only free memory
  - Must allocate blocks from free memory
  - Must align blocks so they satisfy all alignment requirements
    - 8 byte alignment for GNU **`malloc`** (**`libc malloc`**) on Linux boxes
  - Can't move the allocated blocks once they are **`malloc`**'d
    - *i.e.*, compaction is not allowed

6

Interaction with OS

- Change heap size
  - **`brk`** system call



7

Performance Goal: Runtime Speed

- Given some sequence of **`malloc`** and **`free`** requests:
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$
- Speed – throughput:
  - Number of completed requests per unit time
  - Example:
    - 5,000 **`malloc`** calls and 5,000 **`free`** calls in 10 seconds
    - Throughput is 1,000 operations/second

8

Performance Goal: Memory Utilization

- Given some sequence of **`malloc`** and **`free`** requests:
  - $R_0, R_1, ..., R_k, ..., R_{n-1}$
- *Def:* *Aggregate payload* P_k
  - **`malloc(p)`** results in a block with a ***payload*** of **`p`** bytes
  - After request R_k has completed, the ***aggregate payload*** P_k is the sum of currently allocated payloads
- *Def:* *Heap size* H_k
  - Maximum heap size so far
- *Def:* *Peak memory utilization after k requests*
  - $U_k = (max_{i<k} P_i) \ / \ H_k$
- Goals: maximize runtime speed and memory utilization
  - Somewhat conflicting

9

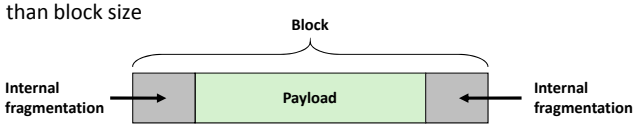
Fragmentation

- Poor memory utilization caused by *fragmentation*
  - ***internal*** fragmentation
  - ***external*** fragmentation

10

Internal Fragmentation

- For a given block, *internal fragmentation* occurs if payload is smaller than block size

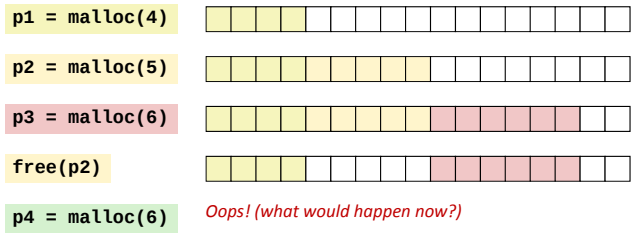


- Caused by
  - Overhead of maintaining heap data structures
  - Padding for alignment purposes
  - Explicit policy decisions (e.g., to return a big block to satisfy a small request)

11

External Fragmentation

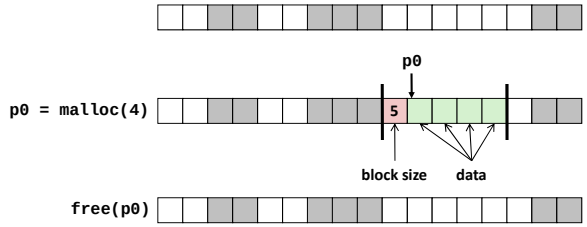
- Occurs when there is enough aggregate heap memory, but no single free block is large enough



12

Structure of Allocated Block

- Standard method
  - Keep the length of a block in the word preceding the block.
    - This word is often called the ***header field*** or ***header***
  - Requires an extra word for every allocated block



13

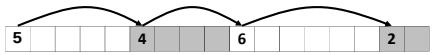
Free Block Management

- How to keep track of free blocks?
- Allocate from free blocks:
  - How do we pick a block to use for allocation – many might fit?
  - What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
- How do we reinsert freed block?

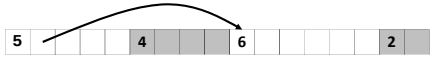
14

Keeping Track of Free Blocks

- Method 1: *Implicit list* using length—links all blocks



- Method 2: *Explicit list* among the free blocks using pointers



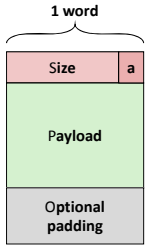
- Method 3: *Segregated free list*
  - Different free lists for different size classes
- Method 4: *Blocks sorted by size*
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key

15

Method 1: Implicit Free Block List

- For each block we need both size and allocation status
  - Could store this information in two words: wasteful!
- Standard trick
  - If blocks are aligned, some low-order address bits are always 0
  - Instead of storing an always-0 bit, use it as a allocated/free flag
  - When reading size word, must mask out this bit

Format of allocated and free blocks



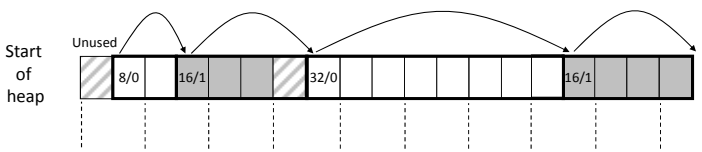
a = 1: Allocated block
a = 0: Free block

Size: block size

Payload: application data (allocated blocks only)

16

Detailed Implicit Free List Example



Double-word aligned

Allocated blocks: shaded
Free blocks: unshaded
Headers: labeled with size in bytes/allocated bit

17

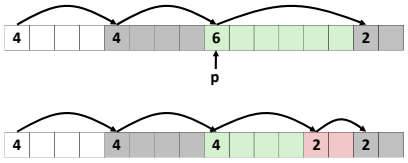
Implicit List: Finding a Free Block

- *First fit:*
  - Search list from beginning, choose ***first*** free block that fits:
  - Can take linear time in total number of blocks (allocated and free)
  - It may cause "splinters" at beginning of list
- *Next fit:*
  - Like first fit, but search list starting where previous search finished
  - Should often be faster than first fit: avoids re-scanning unhelpful blocks
  - Some research suggests that fragmentation is worse
- *Best fit:*
  - Search the list, choose the ***best*** free block: fits, with fewest bytes left over
  - Keeps fragments small—usually helps fragmentation
  - Will typically run slower than first fit

18

Implicit List: Allocating in Free Block

- Allocating in a free block: *splitting*
  - Since allocated space might be smaller than free space, we might want to split the block



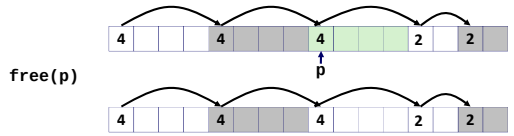
19

Implicit List: Freeing a Block

- Simplest implementation:
  - Need only clear the "allocated" flag

```
void free_block(ptr p) { *p = *p & -2 }
```

  - But can lead to "false fragmentation"



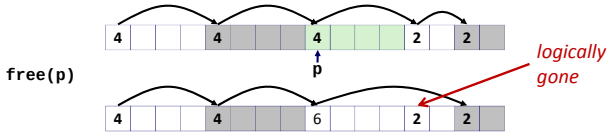
`malloc(5)` ***Oops!***

There is enough free space, but the allocator won't be able to find it

20

Implicit List: Coalescing

- Join *(coalesce)* with next/previous blocks, if they are free
  - Coalescing with next block

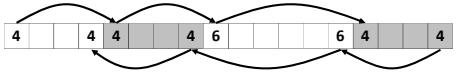


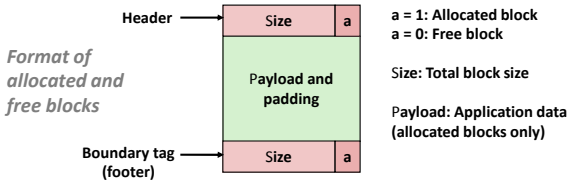
  - But how do we coalesce with *previous* block?

21

Implicit List: Bidirectional Coalescing

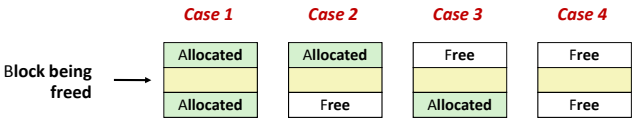
- *Boundary tags* [Knuth73]
  - Replicate size/allocated word at "bottom" (end) of free blocks
  - Allows us to traverse the "list" backwards, but requires extra space
  - Important and general technique!





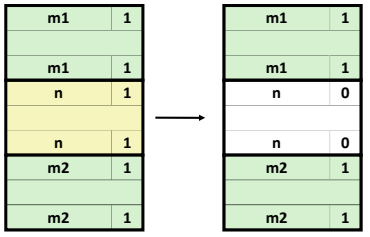
22

Constant Time Coalescing



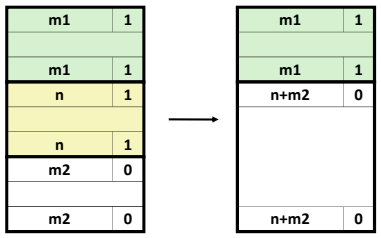
23

Constant Time Coalescing (Case 1)



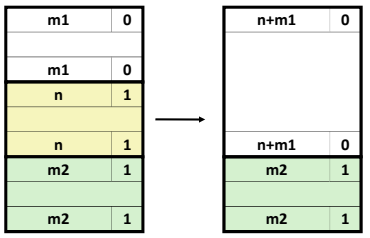
24

Constant Time Coalescing (Case 2)



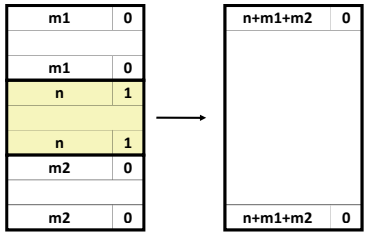
25

Constant Time Coalescing (Case 3)



26

Constant Time Coalescing (Case 4)



27

Space use of footers

- Additional space consumption
- Can it be optimized?
  - Footer tag is free in free blocks
  - No need to coalesce in allocated blocks

28

Implicit Free Block Lists: Summary

- Implementation: very simple
- Allocate cost:
  - linear time worst case
- Free cost:
  - constant time worst case
  - even with coalescing
- Memory usage:
  - Depend on placement policy: first-fit, next-fit or best-fit
- Not used in general allocator because of linear-time allocation
- Segregated free lists (next lecture) approximate a best fit placement policy without having to search entire free list

29

Disclaimer

These slides were adapted from the CMU course slides provided along with the textbook of "Computer Systems: A programmer's Perspective" by Bryant and O'Hallaron. The slides are intended for the sole purpose of teaching the computer organization course at the University of Rochester.

30