CSO-Recitation 11

CSCI-UA 0201-007

R11: Assessment 09 & Dynamic memory allocation

Today's Topics

- Assessment 09
- Dynamic memory allocation
    - implement your malloc & free

Dynamic Memory Allocation

For when static memory isn’t enough

Why Dynamic Memory?

- You don’t always know how much memory you will need for your program
- What if you want to write a program that finds the average value in a column?
- If you did write such a program, how do you handle a user giving you a really big file, bigger than you expected?
- Even if you made sure you specified a really big global variable as a static buffer, people might still give you bigger files
  - And why go through that trouble anyway instead of just having dynamic memory?

Dynamic memory and the stack

- Does the stack give us dynamic memory?
    - In a sense, yes
    - However, it isn’t always suitable, because the memory gets reused after we return from a function call
    - By default the stack is also only a few megabytes in size

Dynamic memory on the heap

- We can use the sbrk syscall to ask the operating system to give us more heap space
- We can also use it to give back to the operating system
- However, in the real world, programmers don’t often do this themselves
  - Why?
- Instead, we usually use a library that handles things for us
  - API: malloc and free
  - Dynamic memory allocator

Malloc and Free

- Malloc allocates us a contiguous section of memory
  - It returns a void*, which is just “pointer to anything”
  - So you cast the result of malloc to what you want, e.g. int *
  - Malloc can return NULL if there was an error
- Free gives the memory back to the allocator
  - DO NOT call free twice on the same section of memory
    - This is undefined behavior
  - What you call free on must be the result of malloc
    - (or calloc or realloc, but I won’t discuss those)

Allocators

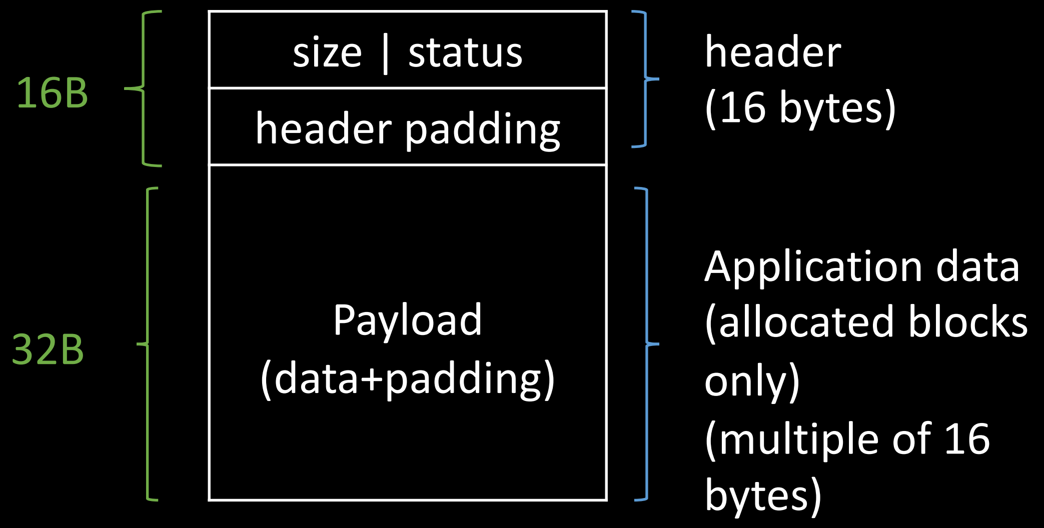
- Allocators can’t move data around
- How do you track what parts of the heap are freed or malloced? How do you track their sizes?
  - The trick is to store metadata along with the data in the heap to create a “linked list”
    - implicit list, explicit list
  - You store the status of the chunk (free or allocated), and the size of the data (which effectively points to the next chunk)
- When someone asks for memory, what do you give them?
  - There are a number of different strategies
- How to give back the memory?

Malloc using Implicit list

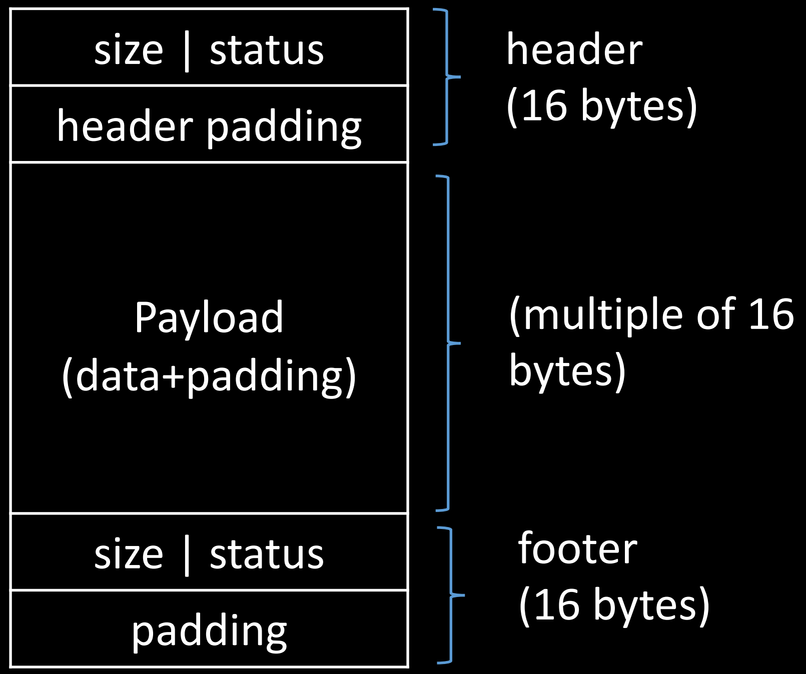
1. Structure of implicit list
2. Where to place an allocation?
3. Splitting a free block
4. Coalescing a free block

Malloc using Implicit list

- Structure of implicit list
  - Implicit list means that it does not use pointers explicitly, but it can find the next node just like a linked list.



e.g. p=malloc(20);



Malloc using Implicit list

- Where to place an allocation?
- Different algorithms:
  - First fit → easy & fast; cause fragmentation at beginning of the heap
  - Best fit → good for utilization; slower
  - Next fit → faster than first fit; even worse fragmentation

Malloc using Implicit list

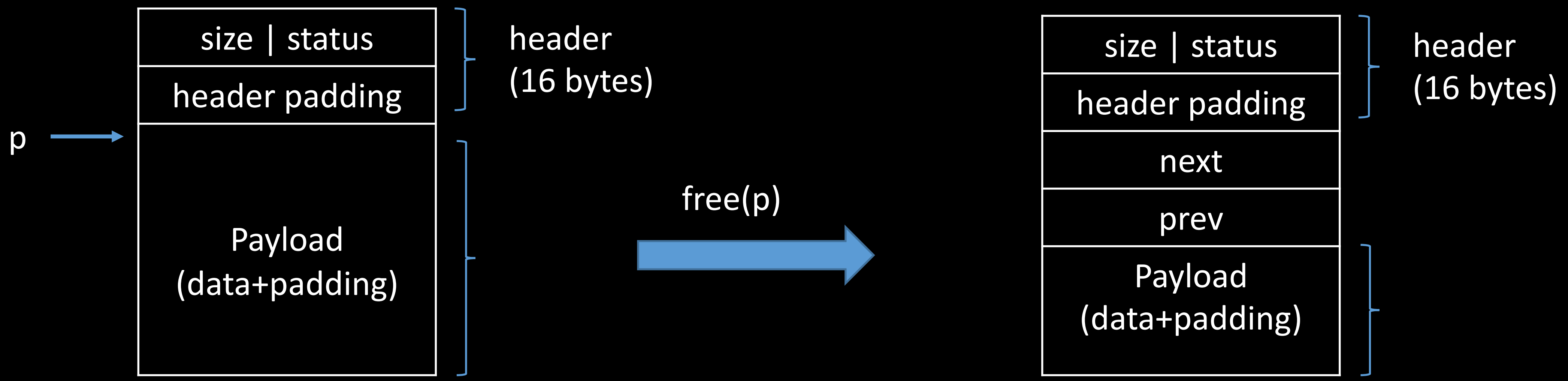
- Splitting a free block
  - Happens when we do memory allocation, e.g. p10=malloc(16)
  - (I have found a suitable free block)
  - ➢ find the next chunk (according to the size you want)
  - ➢ set size & status
- Coalescing a free block
  - Happens when we do free memory, e.g. free(p10)
  - After free, merge this free block with its next(& prev) free neighbor
  - ➢ find the next and prev chunk, and check its status
  - ➢ set size & status
    - ➢ (don’t forget the footer)

Implement Malloc using Implicit list

- Malloc <malloc(size)>
  - get the size of the chunk to allocate
  - find a free chunk
  - ask the OS for chunk
    - Split chunk if necessary
  - set this chunk status to be allocated
  - return pointer to the payload
- Free <free(p)>
  - go to the header from payload
  - set this chunk status to be free
  - coalesce

Malloc using Explicit free list

- Based on the implicit list, because the implicit list is too slow
- Structure of explicit free list
  - Maintain a linked list by adding 2 pointers: next & prev
    - points to the next/previous free chunk
    - only the free chunk needs to be recorded, and the allocated ones do not need



Implement Malloc using Explicit free list

- Malloc <malloc(size)>
  - get the size of the chunk to allocate
  - find a free chunk (in your linked list – linked list traverse)
    - delete this chunk from the linked list
  - ask the OS for chunk
    - Split chunk if necessary
    - insert the new free chunk to the linked list
  - set this chunk status to be allocated
  - return pointer to the payload
- Free <free(p)>
  - go to the header from payload
  - free the chunk
    - set this chunk status to be free
    - initial the next & prev pointer
  - coalesce
    - delete some chunk(s) from the linked list
  - insert this new free block into the linked list