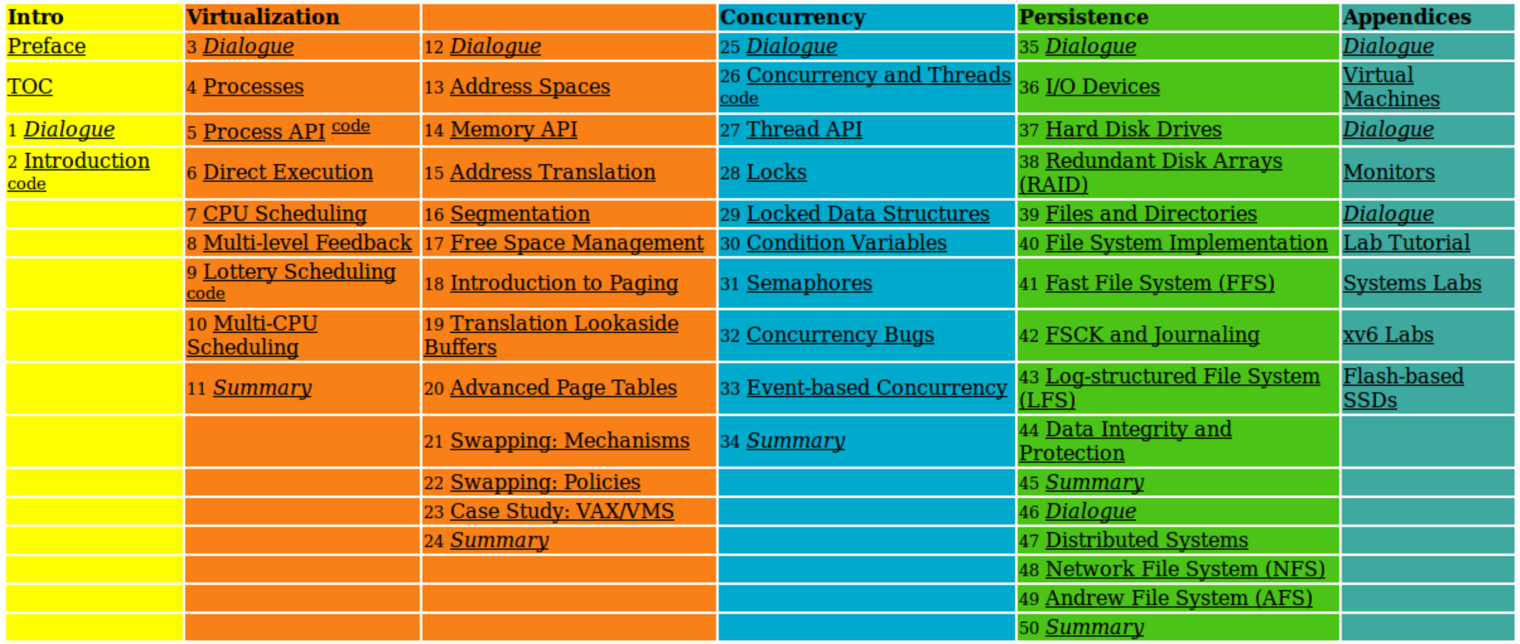
CSIS 429 Operating Systems

Lecture 11: Free Space Management

October 14th 2020

Textbook chapters

- Read “Free Space Management”



Intro	Virtualization		Concurrency	Persistence	Appendices
Preface	3 *Dialogue*	12 *Dialogue*	25 *Dialogue*	35 *Dialogue*	*Dialogue*
TOC	4 Processes	13 Address Spaces	26 Concurrency and Threads code	36 I/O Devices	Virtual Machines
1 *Dialogue*	5 Process API code	14 Memory API	27 Thread API	37 Hard Disk Drives	*Dialogue*
2 Introduction code	6 Direct Execution	15 Address Translation	28 Locks	38 Redundant Disk Arrays (RAID)	Monitors
	7 CPU Scheduling	16 Segmentation	29 Locked Data Structures	39 Files and Directories	*Dialogue*
	8 Multi-level Feedback	17 Free Space Management	30 Condition Variables	40 File System Implementation	Lab Tutorial
	9 Lottery Scheduling code	18 Introduction to Paging	31 Semaphores	41 Fast File System (FFS)	Systems Labs
	10 Multi-CPU Scheduling	19 Translation Lookaside Buffers	32 Concurrency Bugs	42 FSCK and Journaling	xv6 Labs
	11 *Summary*	20 Advanced Page Tables	33 Event-based Concurrency	43 Log-structured File System (LFS)	Flash-based SSDs
		21 Swapping: Mechanisms	34 *Summary*	44 Data Integrity and Protection	
		22 Swapping: Policies		45 *Summary*	
		23 Case Study: VAX/VMS		46 *Dialogue*	
		24 *Summary*		47 Distributed Systems	
				48 Network File System (NFS)	
				49 Andrew File System (AFS)	
				50 *Summary*	

Review: Virtual Addresses

Every address generated by a process is treated as a "virtual address" by the OS

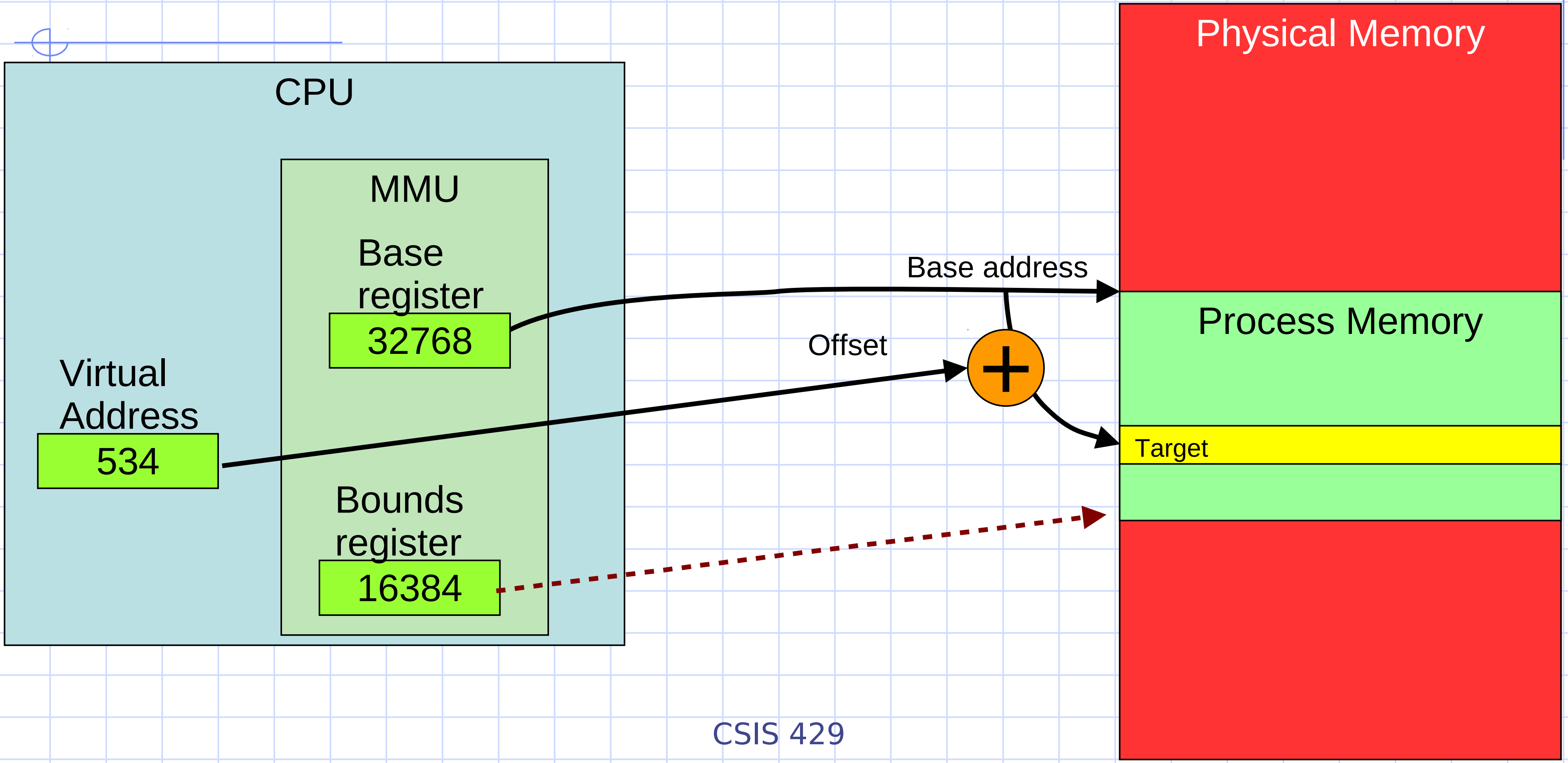
OS translates each virtual address to a DRAM address.

The "crux" of the problem:

How can the OS build the abstraction of a private address space while actually having many processes use physical memory?

And doing so securely? And efficiently?

Dynamic Relocation with Base and Bounds



Managing Free Space

Can be easy with Paging:

- ✔ Space is divided into page frames
- ✔ Keep a list of free frames
- ✔ When requested, return first free frame

Difficult:

- ✗ When free space consists of variable-sized units
- ✗ OS manages physical memory using segmentation
- ✗ In user-space memory allocation like `malloc`/`free`
- ✗ Main problem: External Fragmentation

Crux of the Problem: How to manage free space

How should free space be managed, when satisfying variable-sized requests?

Strategies to minimize fragmentation?

Time and space overheads of the options?

malloc and free

```
void* malloc(size_t size)
```

- `size` is the number of bytes being requested
- `void*` means “untyped pointer” or “pointer to any type”

  not “pointer to void”, not “pointer to nowhere”

`malloc` and `free`

```
void free(void* ptr)
```

- `ptr` is a pointer obtained from malloc

  Note: calling program does not have to specify the size of memory to be freed!

  This implies that the implementation of malloc/free have to track the sizes of every allocation
- `void` return type → nothing returned.

`malloc` and `free`

The space that `malloc`/`free` manage is known as the "heap"

A "free list" is used to manage free space in the heap.

→ does not have to be a list data structure

→ any data structure that tracks free space and has references to all the free chunks of space

`malloc` and `free`

Once space is allocated using malloc, it cannot be relocated to some other location in memory.

In C, it would be hard to figure out all the pointers that may point "into" some allocated chunk.

`malloc` and `free`

- Once space is allocated using malloc, it cannot be relocated to some other location in memory.
- In C, it would be hard to figure out all the pointers that may point “into” some allocated chunk.
- Java has strong typing and garbage collection.

`malloc` and `free`

Allocated space is owned by the program which is responsible for freeing it.

→ No compaction or defragmentation can take place

An allocator could ask for the heap to “grow” using e.g. `sbrk`

Fragmentation

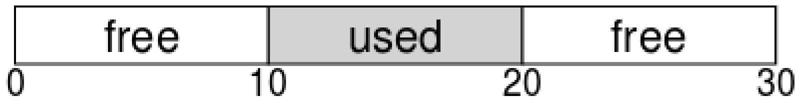
Memory allocators have to worry about two kinds of fragmentation:

External fragmentation occurs when the space between allocated chunks become too small to be usable

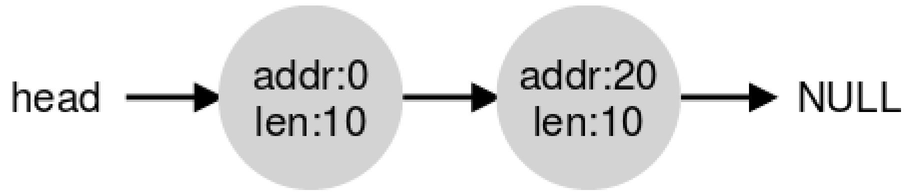
Internal fragmentation occurs when allocated chunks are only partly used.

Allocators: Free List

Memory allocators have to track free space:



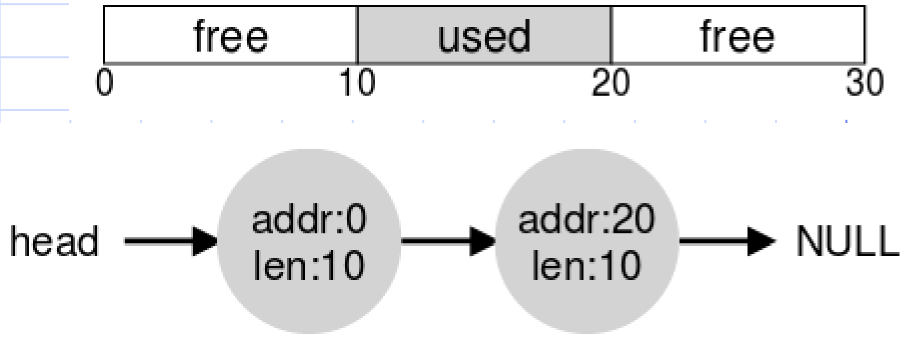
using some kind of "free list":



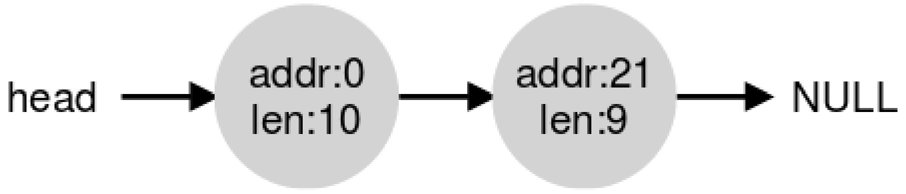
In this case a request for over 10 units will fail.

Allocators: Splitting Memory

Memory allocators will have to split free space when given a doable request

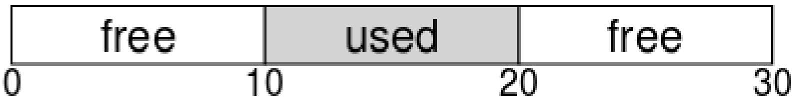


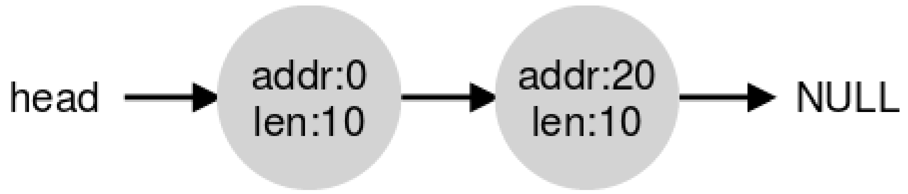
e.g. for 1 unit of space:

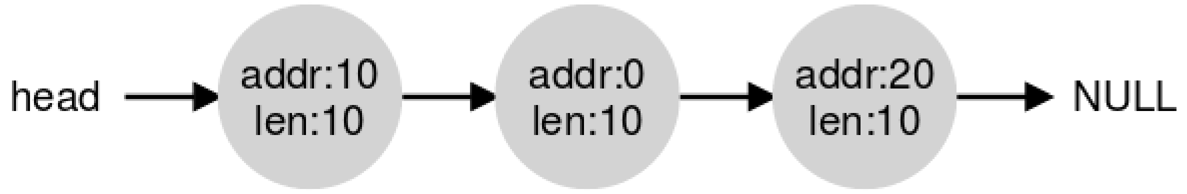


Allocators: Coalescing Free Space

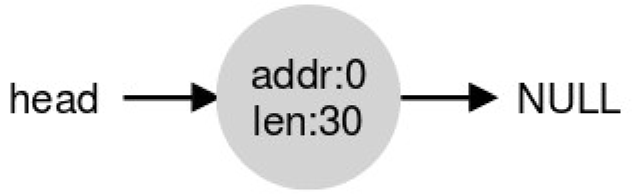
- If a “used” space is deallocated or freed, the allocator will need to not only track it:







But also coalesce contiguous chunks of free space

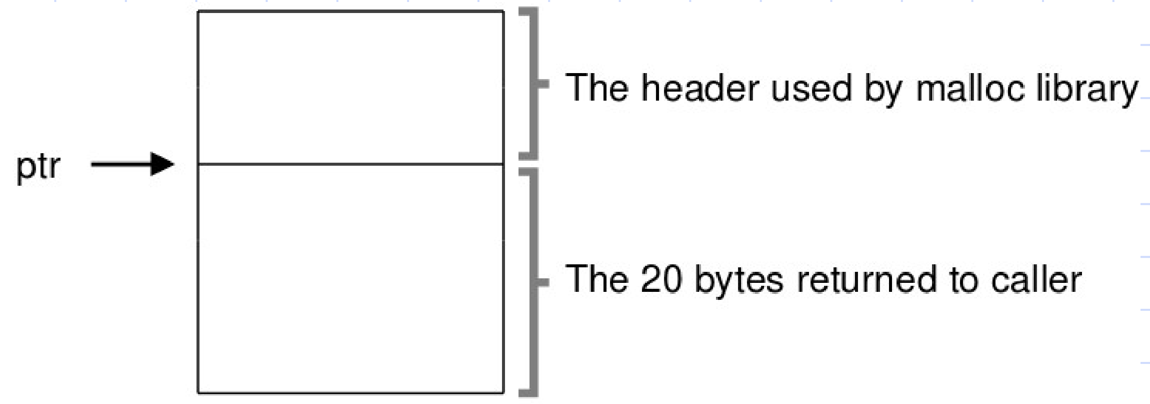


Tracking sizes of allocated regions

Recall: `free` only specifies the location of a chunk of memory to be deallocated, not its size.

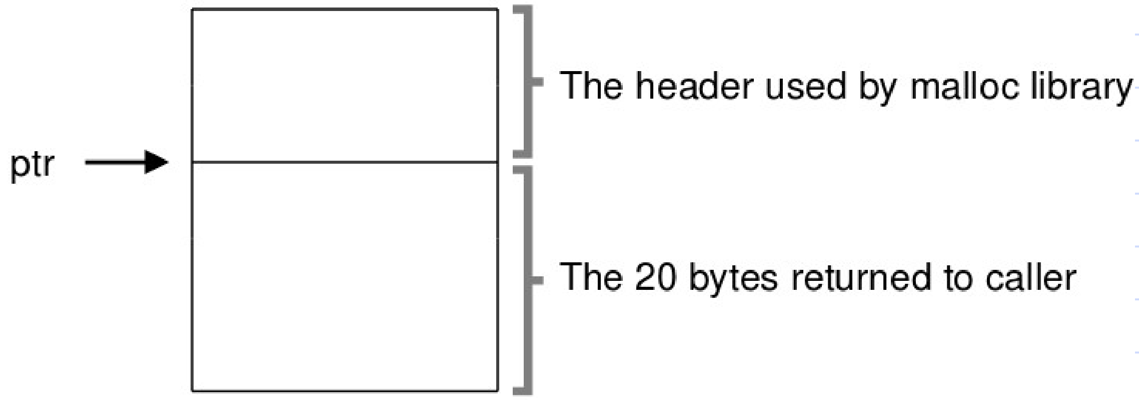
malloc/free have to figure out the size.

Allocators store accounting information in a header block, usually just before the pointer to the allocated space.


ptr
The header used by malloc library
The 20 bytes returned to caller

Tracking sizes of allocated regions

What goes into the headers?


ptr
The header used by malloc library
The 20 bytes returned to caller

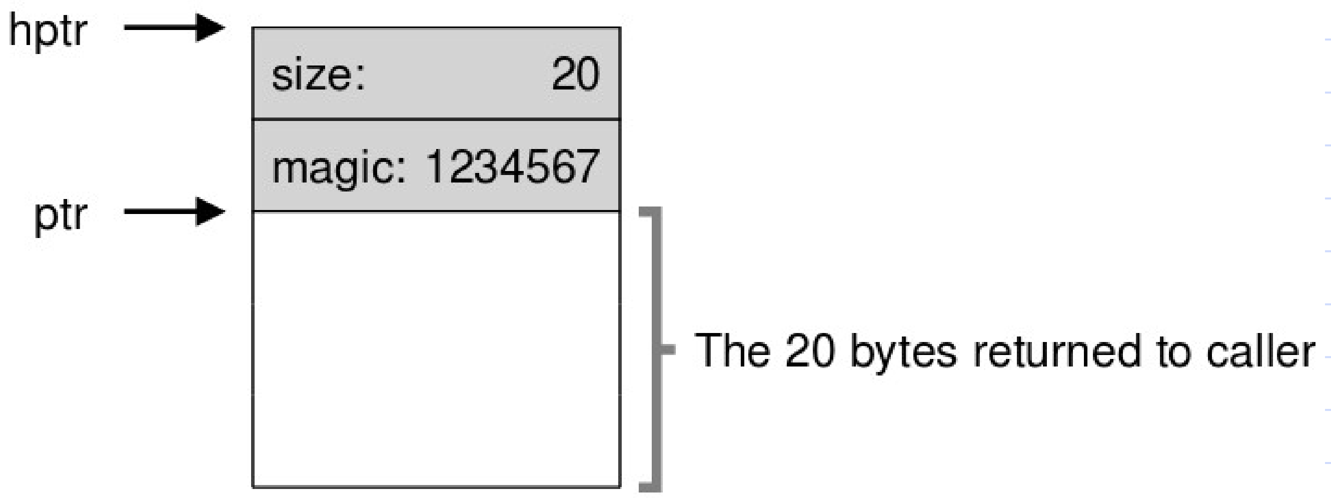
```
typedef struct header{
    int size;
    int magic;
} header_t;
```

Magic numbers for integrity checking

Tracking sizes of allocated regions

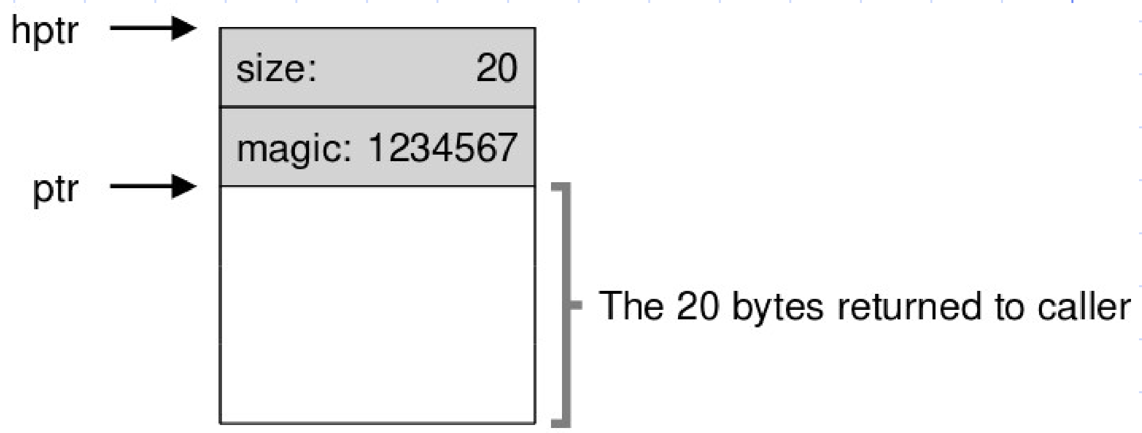
Header structure:

```
typedef struct header{
    int size;
    int magic;
} header_t;
```



Tracking sizes of allocated regions

How free uses the header information:



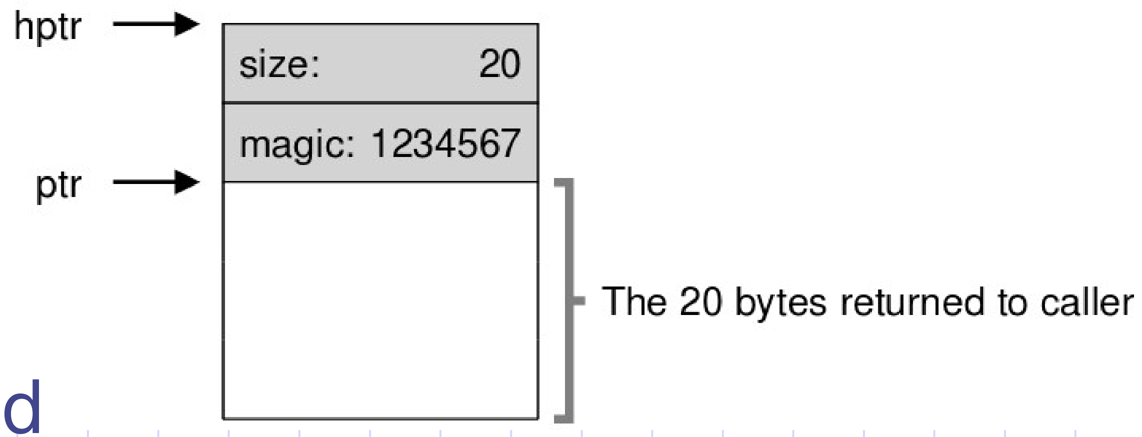
```
void free(void* ptr) {
    header_t* hptr = (void*)ptr – sizeof(header_t);

    assert(hptr->magic == 1234567);
  …
```

Pointer arithmetic

Tracking sizes of allocated regions

How `malloc` uses the header information:



When `malloc` is invoked for a chunk of size N, it will look for a chunk of size "`N+sizeof(header_t)`" and returns a pointer to the space past the header.

Embedding a free list in the heap

Recall: when adding a new node to a linked list or tree, we can call `malloc` to allocate space for the node.

Cannot do that if we are implementing `malloc` itself!

We have to embed the free list in the free space.

Free list example

Suppose we have a 4096 byte heap.

→ Initially the free list will have 1 node of size 4096 – the size of the header.

The node declaration will look like:

```
typedef struct node {
    int          size;
    struct node* next;
} node_t;
```

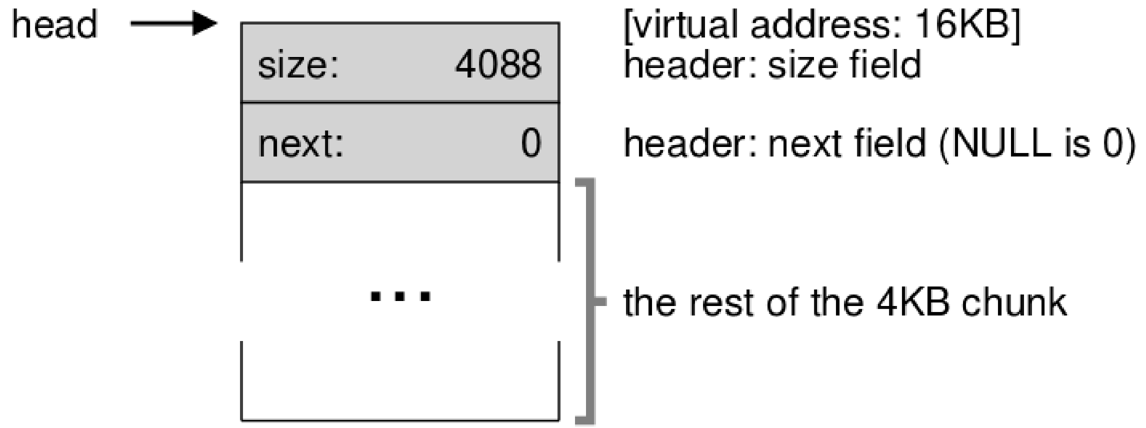
Free list example: initialization

Let’s assume that the heap is built using free space acquired using mmap:

```
// mmap() returns a pointer to a chunk of free space
node_t* head = mmap(NULL, 4096,
               PROT_READ|PROT_WRITE,
               MAP_ANON|MAP_PRIVATE, -1, 0);
head->size   = 4096 - sizeof(node_t);
head->next   = NULL;
```

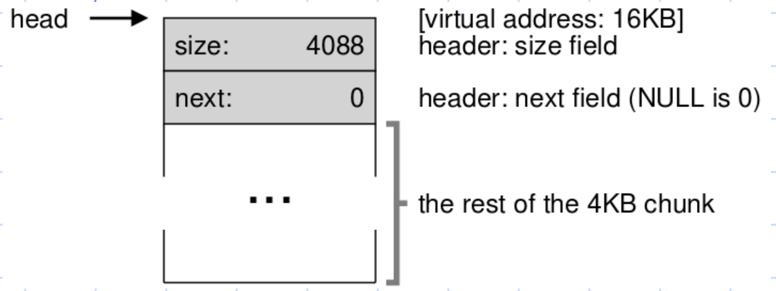
Free list example: initialization

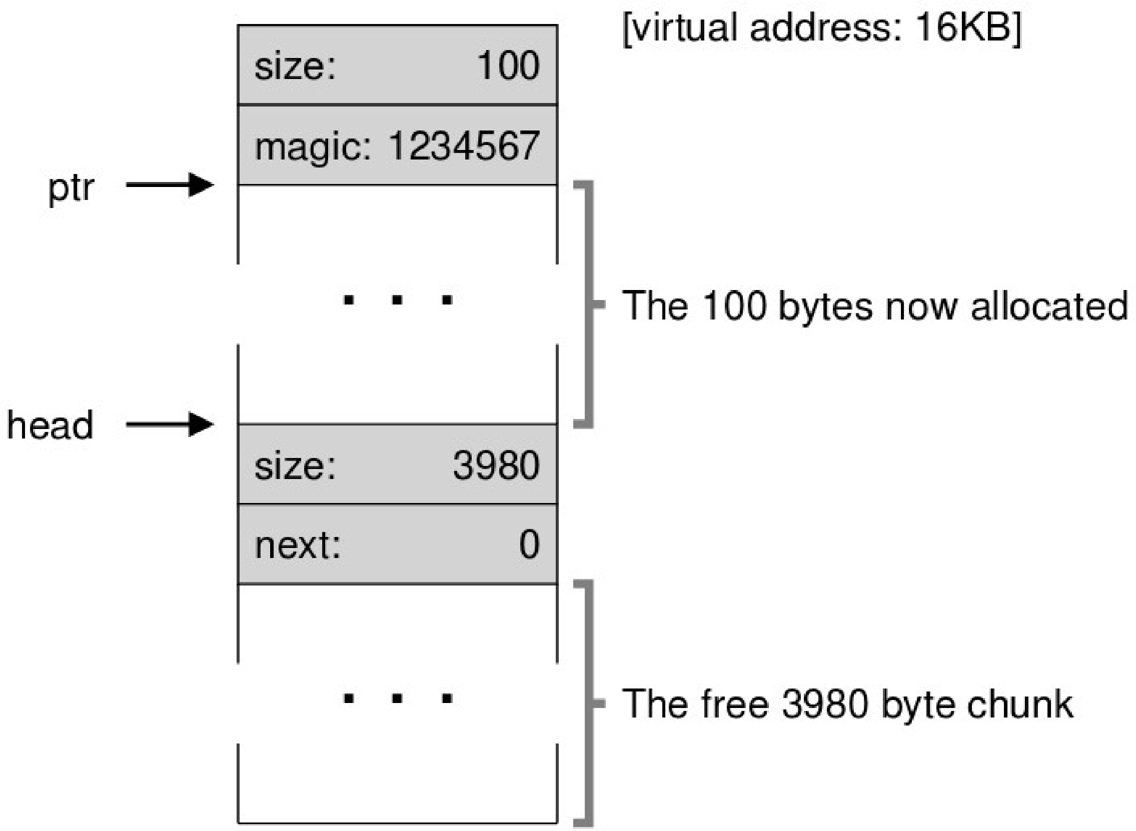
```
node_t* head = mmap(NULL, 4096,
                    PROT_READ|PROT_WRITE,
                    MAP_ANON|MAP_PRIVATE, -1, 0);
head->size   = 4096 - sizeof(node_t);
head->next   = NULL;
```



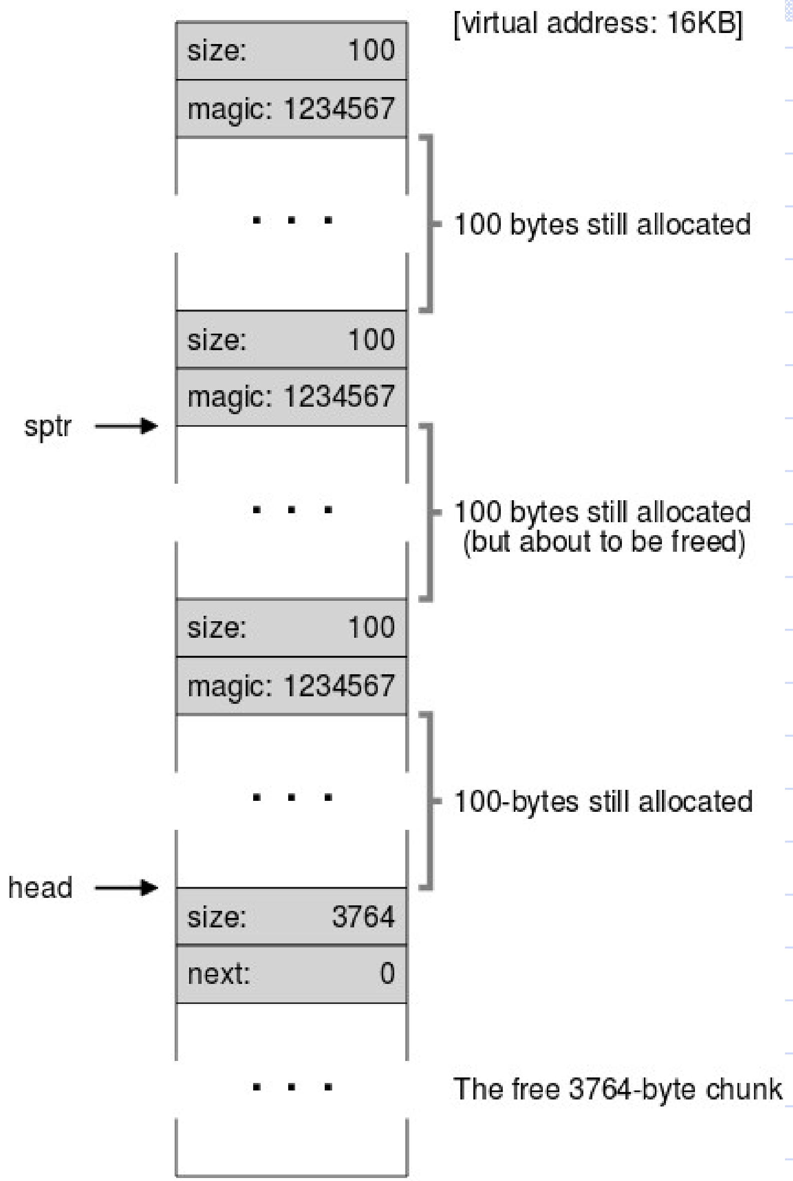
Example: After a 100-byte request

- Remember, the allocator will look for a 108 byte chunk
- It will do a Split



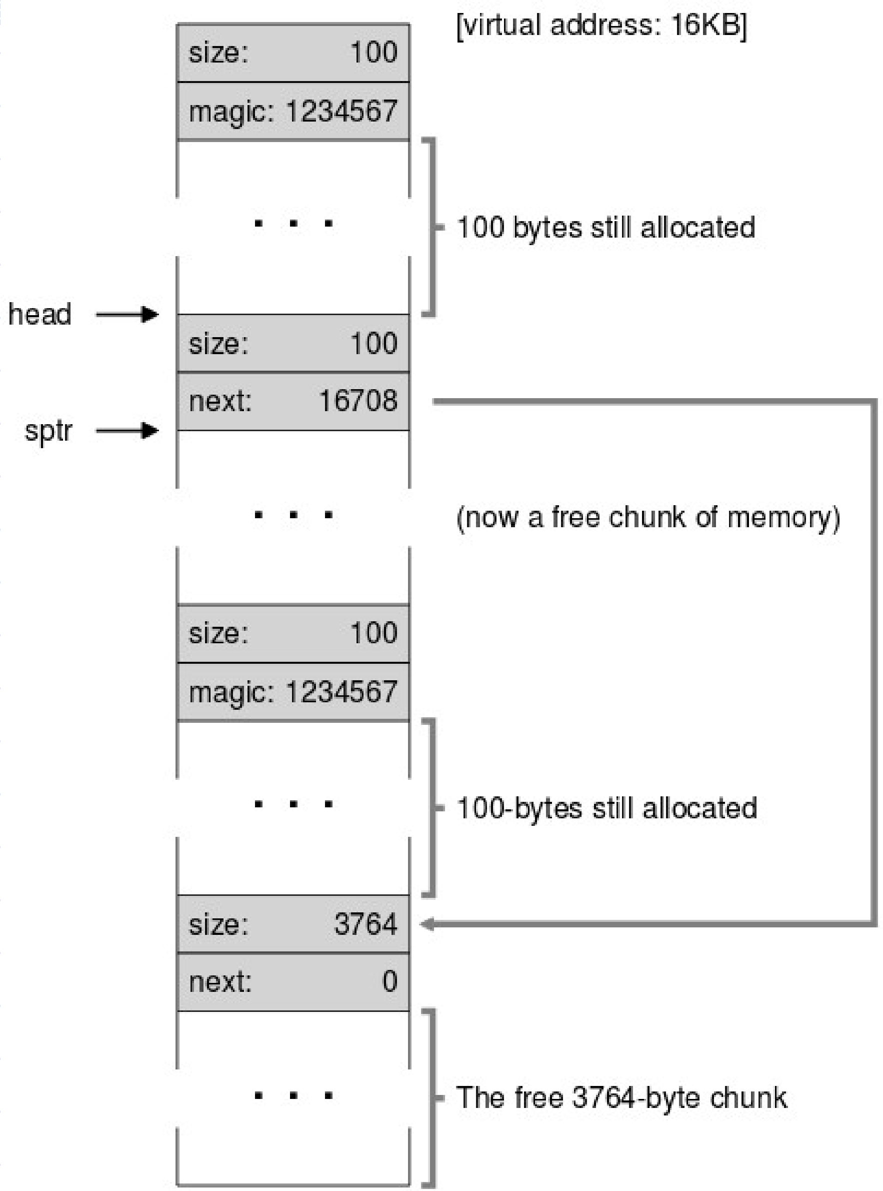


Example: After 3 100-byte requests



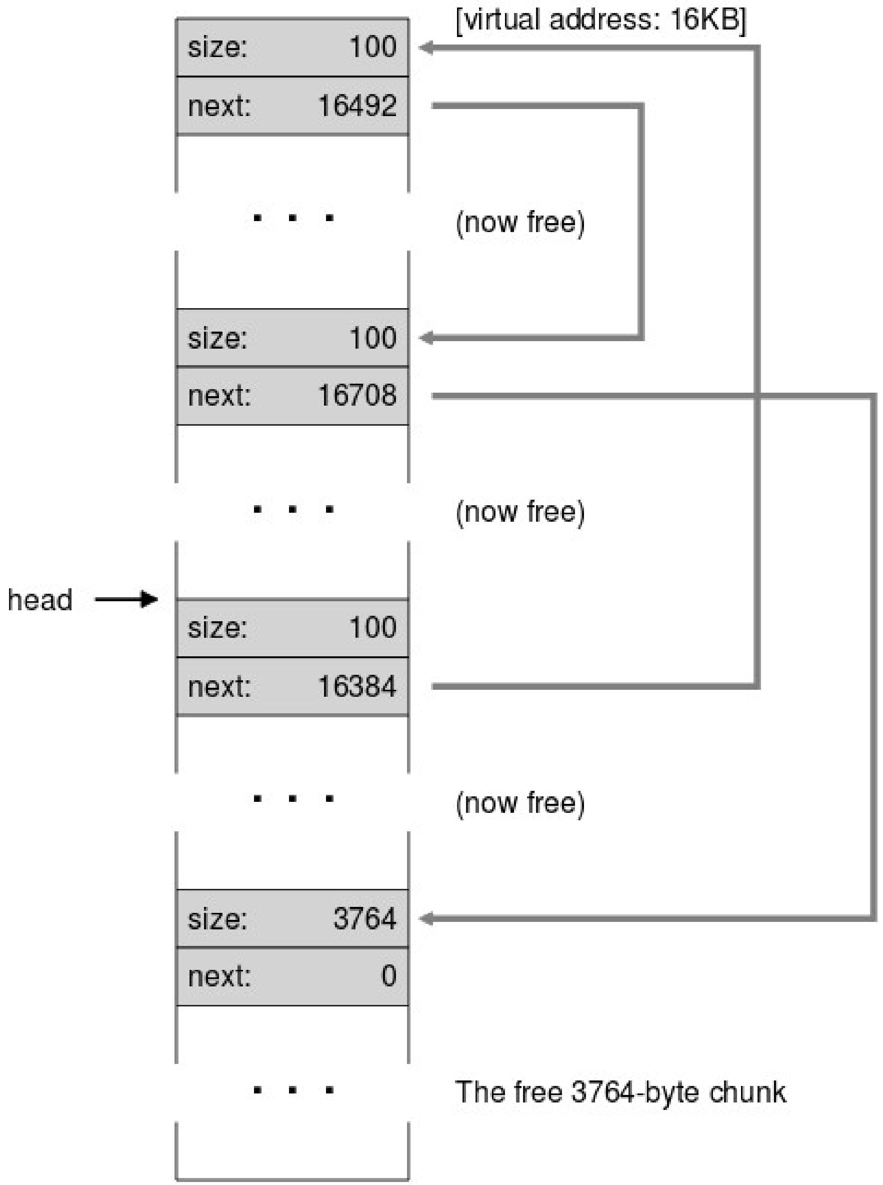
Example: After `free(sptr)`

We now have a free list with two nodes.



Example: After freeing but no coalescing

We now have a free list with four nodes.



Allocator Strategies: First Fit

First Fit: Search the free list and return the first node that fits the request.

→ Fast!

Problem: may result in a lot of small nodes at the beginning of the free list

If the allocator uses address-based ordering of nodes in the free list, coalescing free nodes will be easy.

Allocator Strategies: Next Fit

Next fit: keep a pointer to the node in the free list where the last good node was found.

Search the free list starting from this pointer and return the “next” node that fits the request.

→ Also fast!

May fix the problem of having many fragments at the beginning.

Allocator Strategies: Best Fit

Best fit: On each allocation, search the entire free list.
If exact match is found, stop searching.
Choose the smallest node that fits the request.

Problem: may end up giving us a lot of very small fragments.

Allocator Strategies: Worst Fit

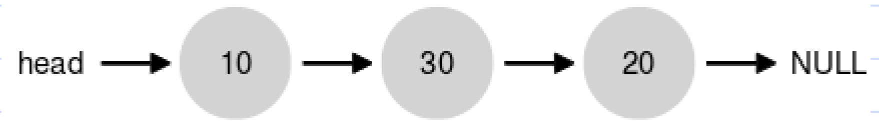
Worst fit: Sounds bad but …

Idea is to find the largest chunk and split it.

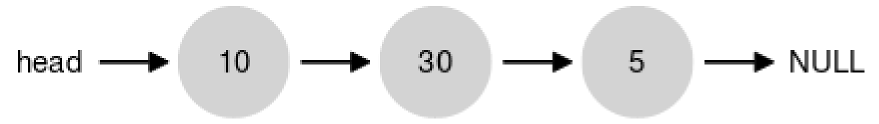
May be the “solution” to the problem of Best fit

Allocator Examples

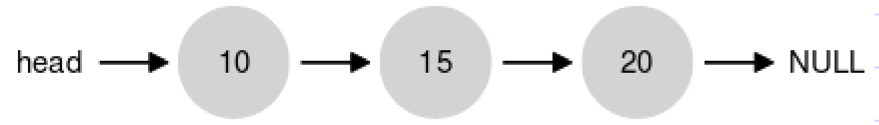
Suppose we start with:



Best fit:



Worst fit:



Buddy Allocation

Fast, simple allocation for blocks that are 2^N bytes

Allocation restrictions:

- Block sizes are restricted to 2^N

Allocation strategy for K byte request:

- Raise allocation request to nearest 2^M
- Search free list and if no perfect match, recursively divide large blocks until correct block is obtained
- “Buddy” block stays free

Free strategy

- Recursively coalesce block with free “buddys”

Which allocator is best?

Depends completely on use.

Slab allocation scheme tried to “pre-allocate” and “pre-initializing” chunks based on past usage patterns.

Characteristics of Allocators

Best fit: Exhaustive search may be slow.

- Tends to leave very large holes, very small ones
- The very small holes → fragmentation.

First fit: Fast

- Tends to leave “average” sized holes

Worst fit:

- Tends to fragment and has high overhead too.

Buddy: Internal fragmentation if not a power of 2

Real world: C malloc function

- Calls sbrk to request more contiguous memory from OS
- Adds small header to each block of memory with
  - Pointer to next free block
  - Size of current block
- Keeps a separate free list for each popular size
  - For fast allocation and to avoid fragmentation
  - Can be inefficient
- First fit used for free list of irregular block sizes