Operating System (OS) CS232

Memory Management: Free-Space Management

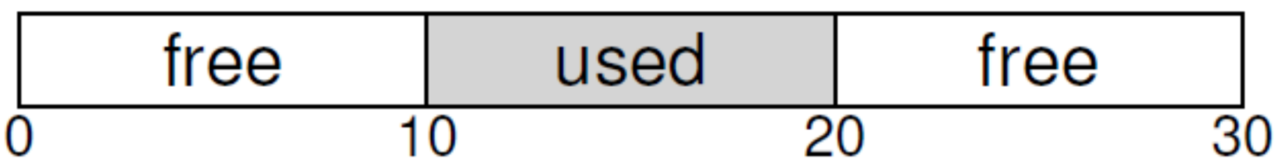
Dr. Muhammad Mobeen Movania

Outlines

- Issues with Segmentation
- Memory Management functions
- Assumptions
- Low-level Mechanisms
- Tracking Size of Allocated Regions
- Examples of several allocators
- Summary

Issues with Segmentation

- We concluded that irrespective of the method used, we still encounter external fragmentation
  - Allocated memory segments are of **unequal size**
  - As memory portions are allocated and deallocated, the memory quickly becomes full of small “holes”
- Consider the example:



- What if a request for 15 bytes arrive?

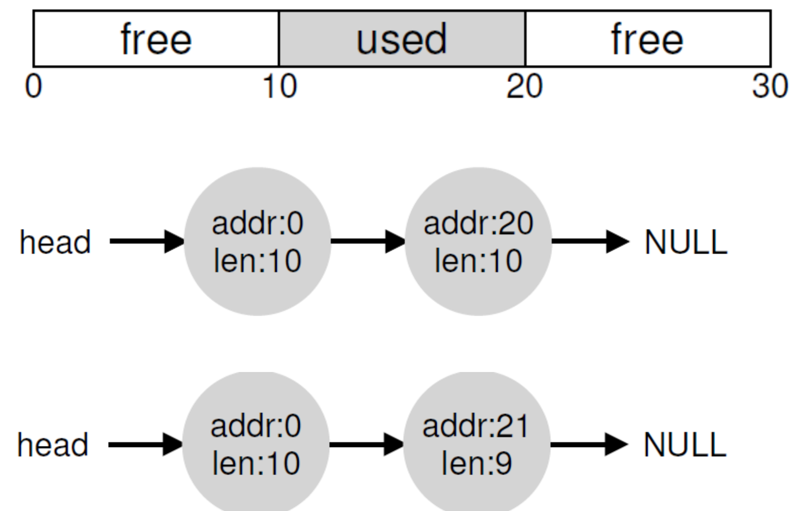
Assumptions

- Compaction is not possible
  - Once memory is handed out to a client, it cannot be relocated to another location in memory

- The allocator manages a continuous region of bytes

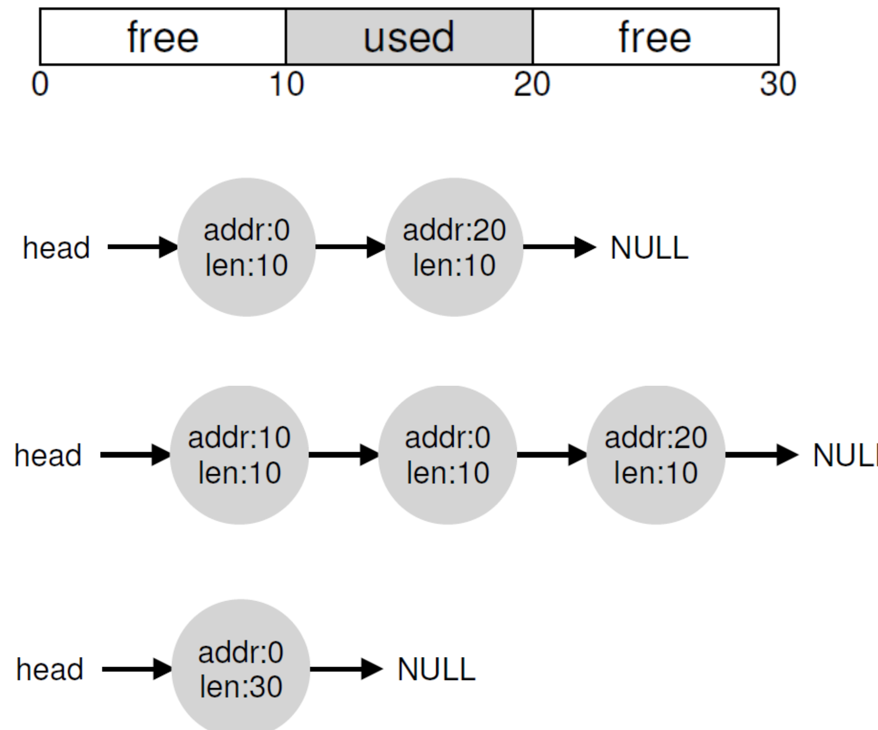
Low-level Mechanisms

- Splitting
  - Assume a 30 bytes heap
  - Request for 10 bytes is easy
  - Request for <10 bytes?
  - Request for 1 byte



Low-level Mechanisms

- Coalescing
  - Free (10)
  - Entire list is free but unavailable!
  - Coalescing



Memory Management functions

- We saw two functions earlier,

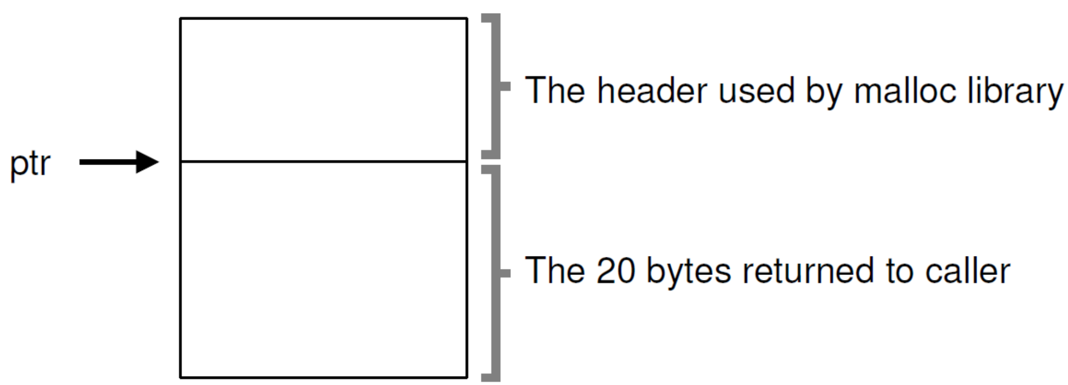
```
void *malloc(size_t size)
void free(void *ptr)
```

- Notice that free doesn’t take a size parameter.
- How does it know how big is the chunk being free’d?
  - There is additional header stored with each dynamic allocation that contains size of allocation
- The allocator library maintains a free list

Tracking Size of Allocated Regions

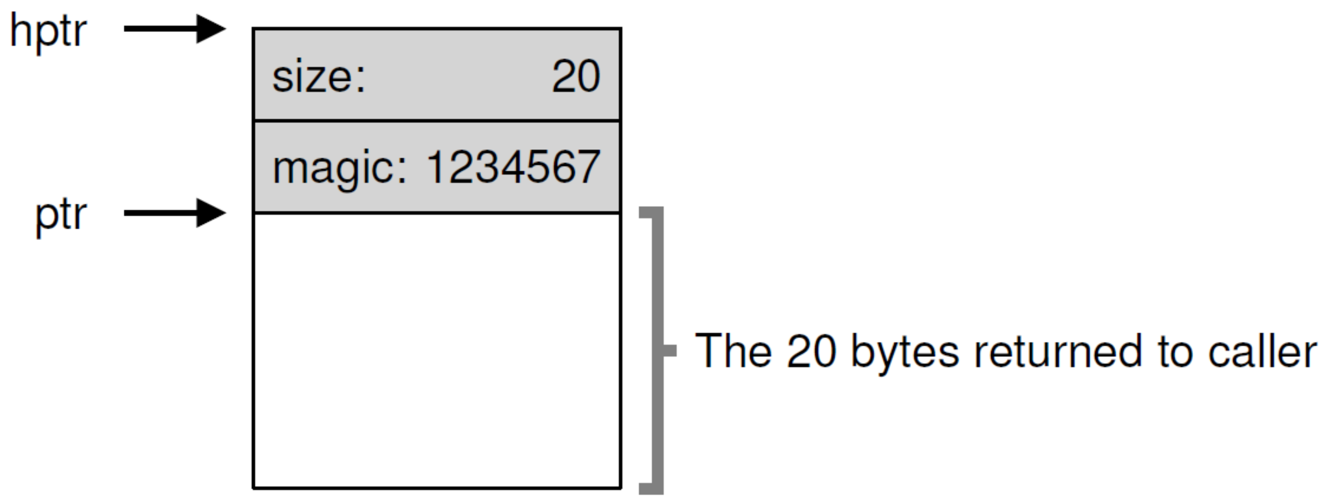
- Most allocators store a little bit of extra information in a **header** block which is kept in memory
  - when a user requests N bytes, the library does not search for a free chunk of size N, rather, it searches for a free chunk of size N + size of header

```
ptr = malloc(20);
```



Header struct

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




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

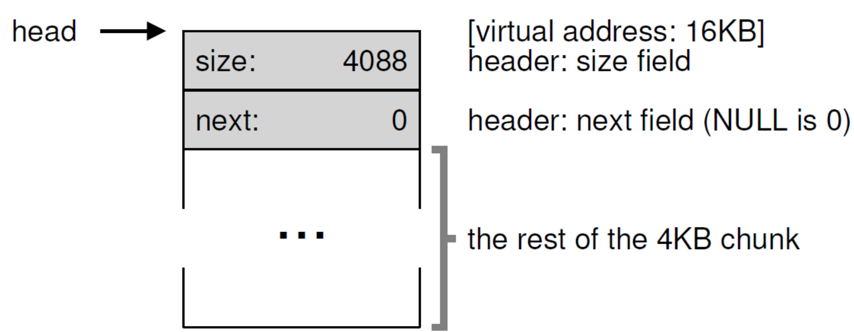
Embedding a free list

- We’ve to keep track of allocations via malloc
- Usually in a list when making a new node we call malloc()
- We need to build the free list inside the free space itself!

Example: Heap with one free chunk

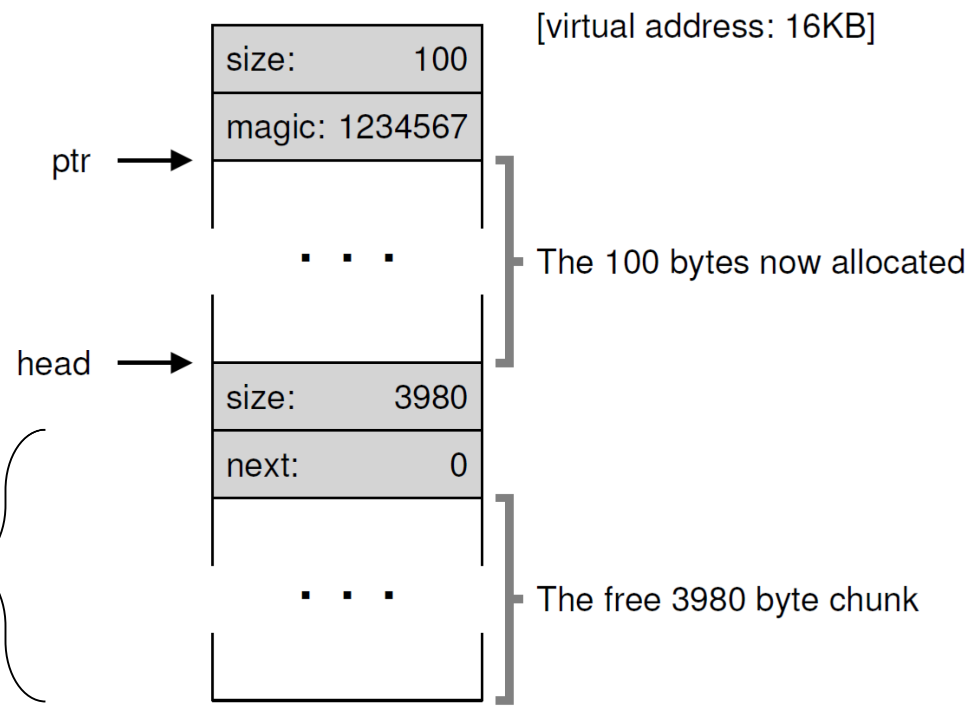
- Heap: 4KB = 4096 bytes
- Initially list has just one entry of 4088 bytes
  – 4096-sizeof(header) = 4096-8 = 4088 bytes
- Memory allocation code

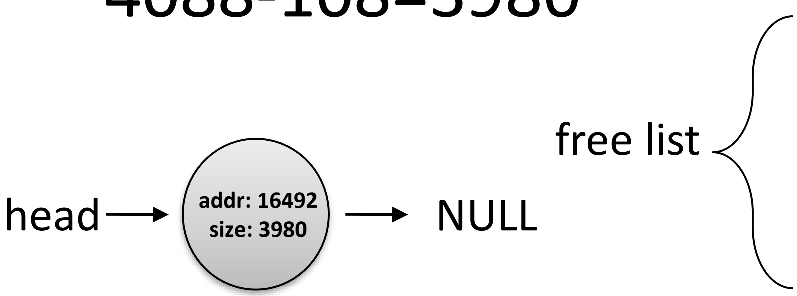
```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                    MAP_ANON|MAP_PRIVATE, -1, 0);
head->size   = 4096 - sizeof(node_t);
head->next   = NULL;
```



Example: Heap after one allocation

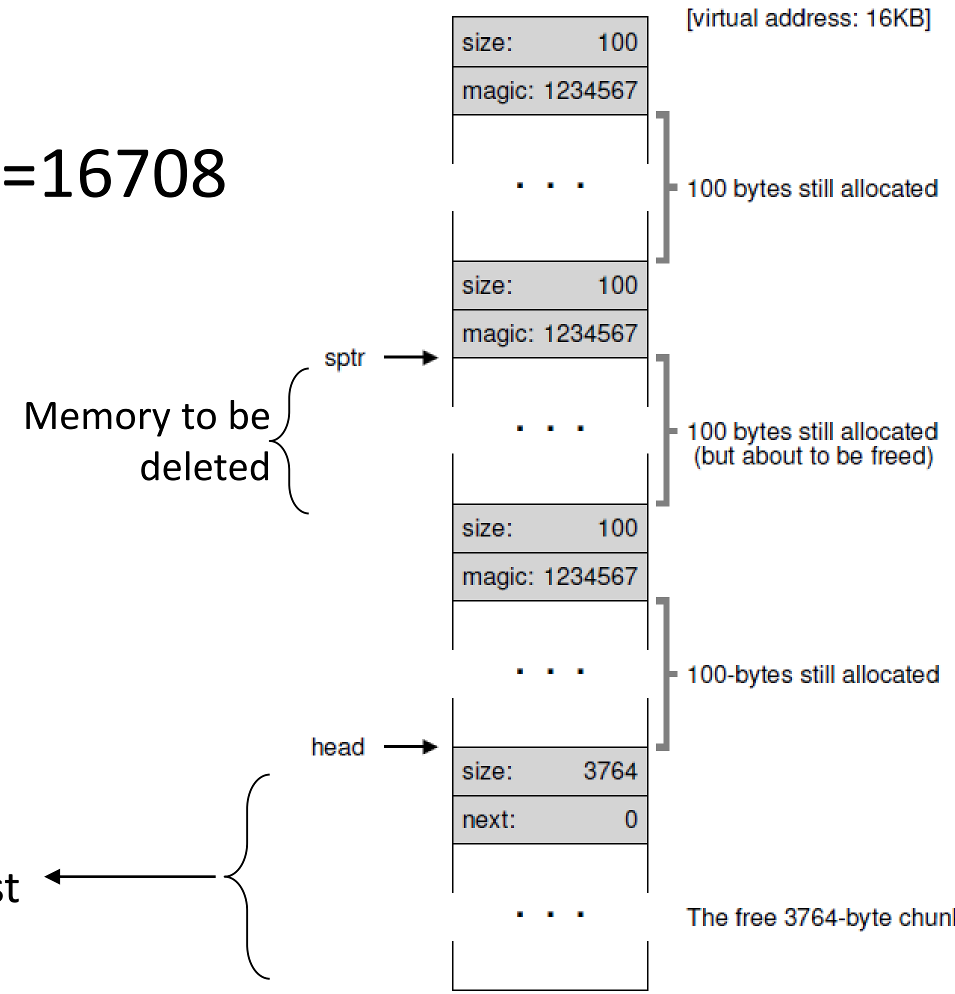
- After calling malloc(100)
  - A split was done
  - Actual allocation of 108 bytes
- Address
  - 16384+108=16492
- Size
  - 4088-108=3980

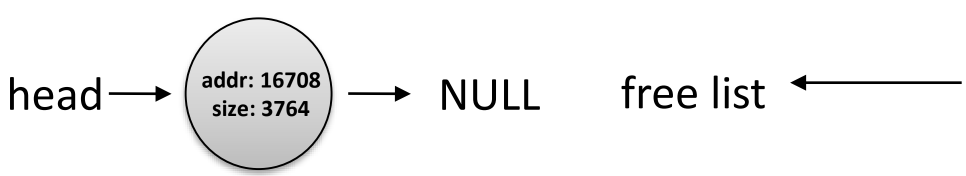




Example: Heap with 3 allocations

- Address
  - 16384+108+108+108=16708
- Size
  - 4088-108-108-108
  - 3764

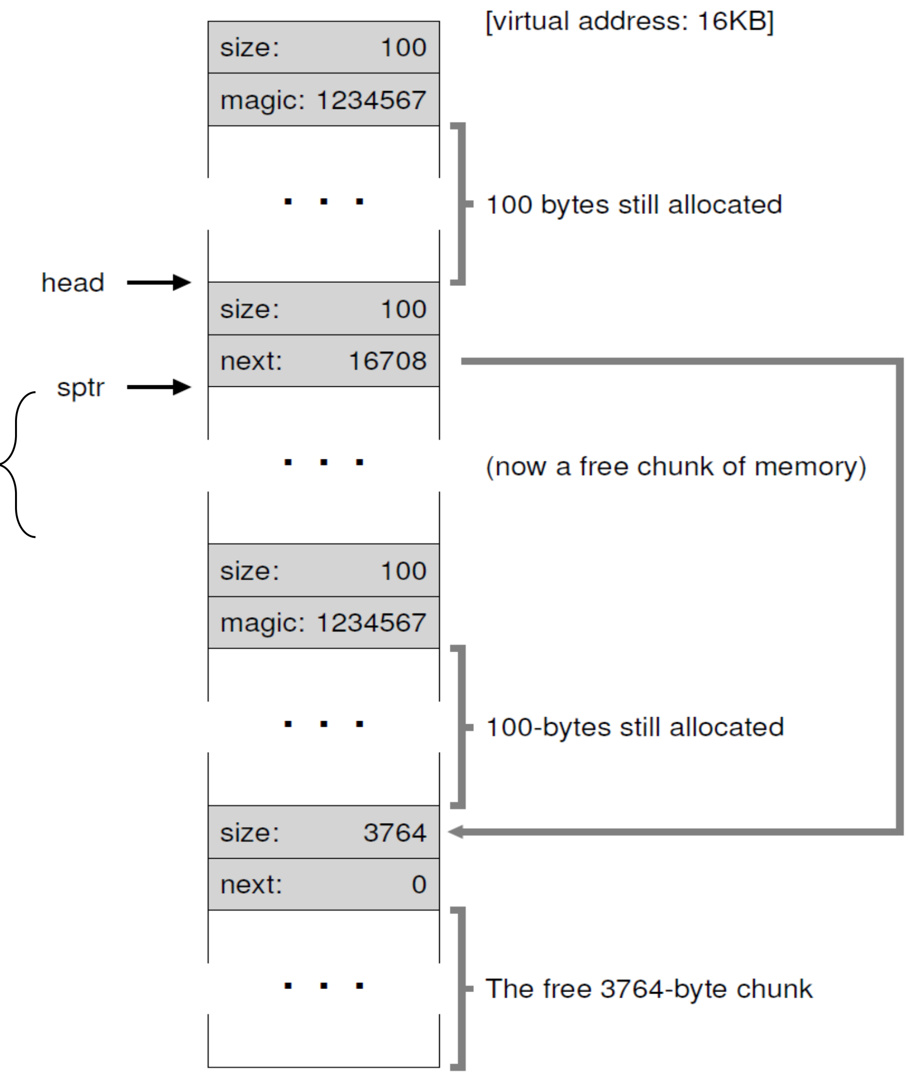


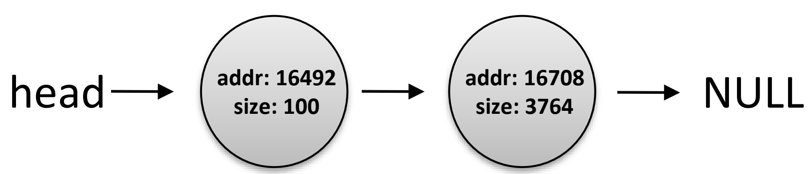


Example: Heap with deletion of one allocated node

- sptr=16500
- free(sptr)
- head=sptr-8

100 bytes memory
to be deleted

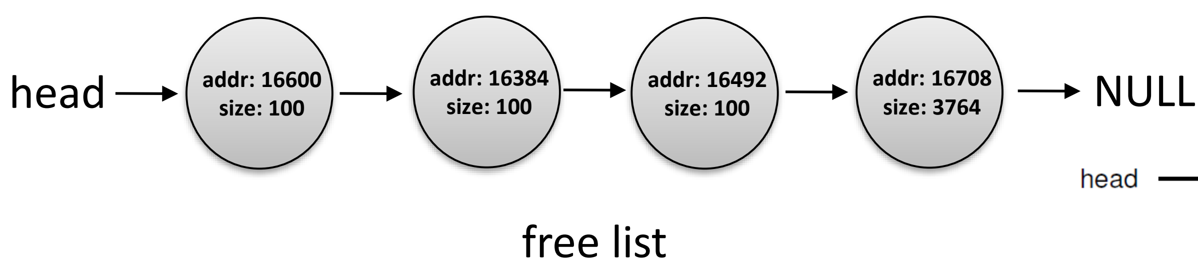


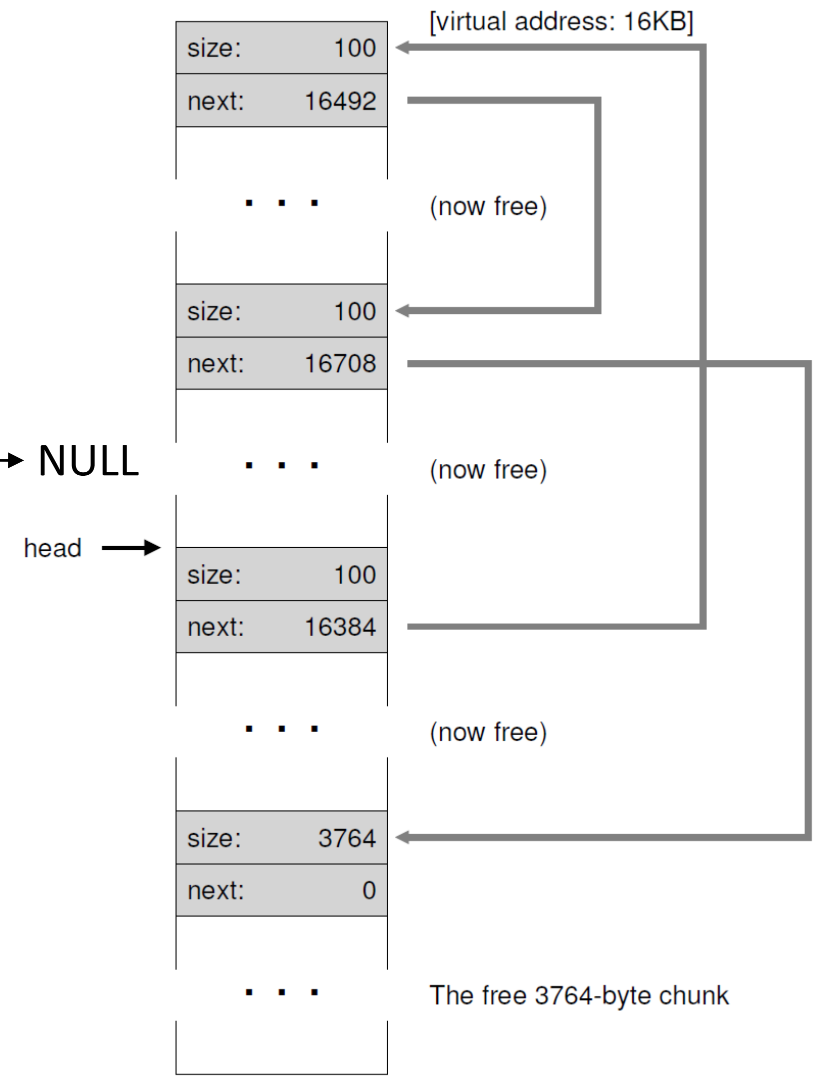


free list

Example: Deletion of all allocated nodes

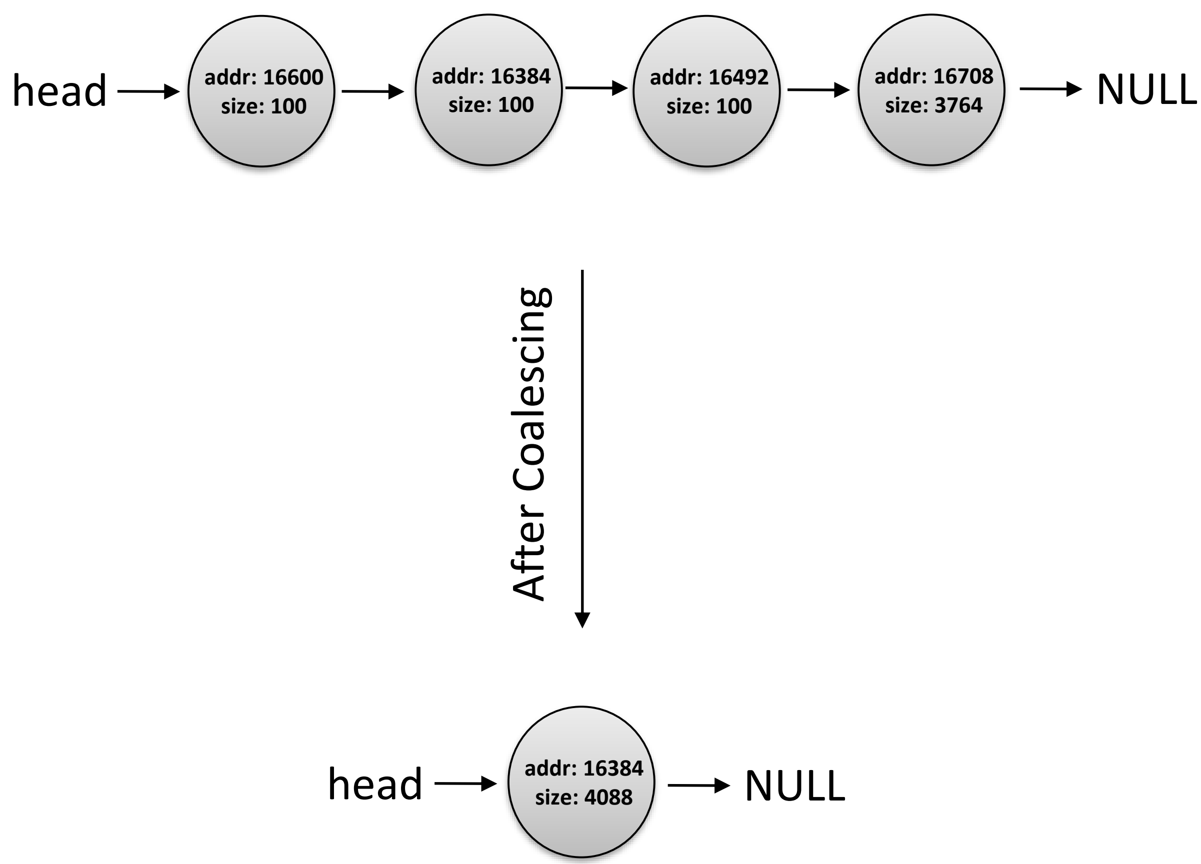
- Coalescing can help remove fragmentation





Example: Deletion of all allocated nodes

- Coalescing can help remove fragmentation



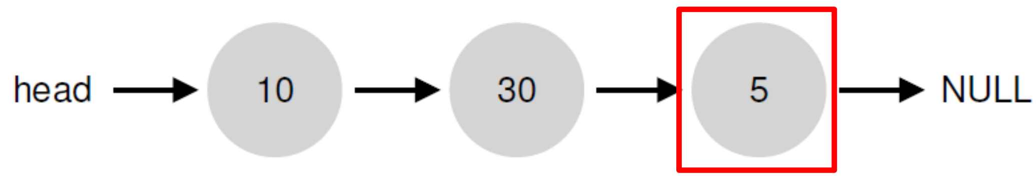
Block allocation strategies (policies)

- Best fit
  - Returns the smallest block that satisfies our request
- Worst fit
  - Returns the biggest chunk that satisfies our request
- First fit
  - Returns the first chunk that satisfies our request
- Next fit
  - Returns the next block (from the last block allocated) that satisfies our request

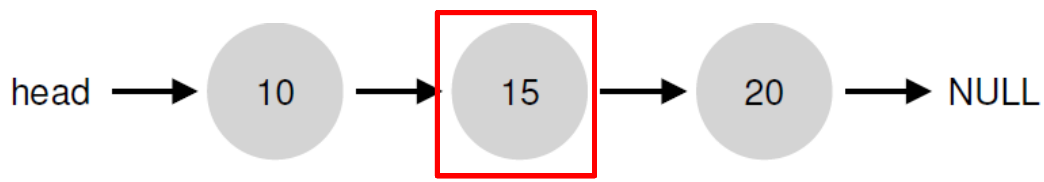
Examples (15 bytes alloc. req.)

- Original list

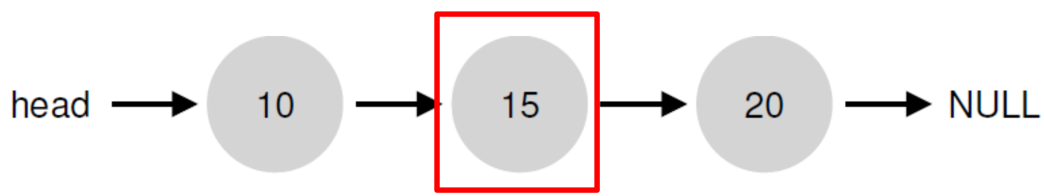
- Best fit



- Worst fit



- First fit



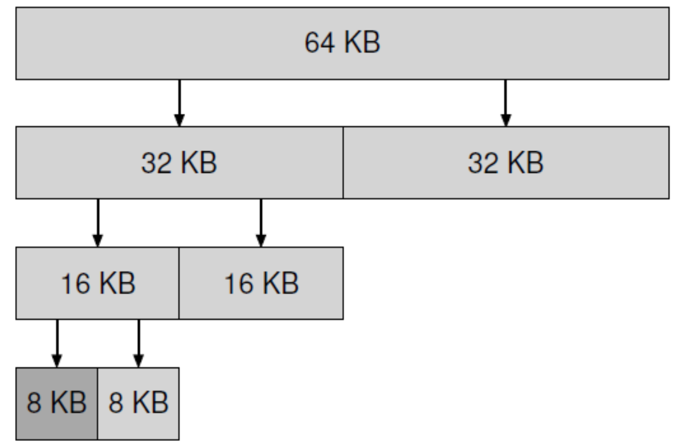
Other approaches

- Segregated lists
  - If a particular application has one (or a few) popular-sized request, keep a separate list just to manage objects of that size.
  - This results in less fragmentation on average.

- Slab allocator in Solaris
  - Allocates object-caches for frequently requested kernel structures i.e. locks, file system inodes, etc.
  - When object-caches run low, they request more *slabs* from *general allocator*
  - When they have too few allocated objects, the general allocator can reclaim memory
  - It keeps free objects in pre-initialized state

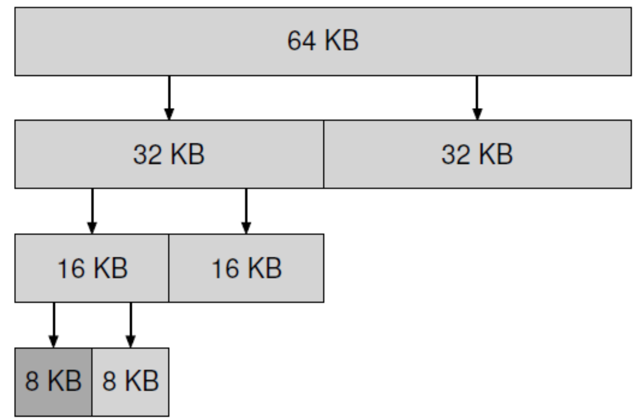
Other approaches

- Binary Buddy Allocator
- Memory is a space of 2^N
- When a request arrives
  - The space is recursively divided by 2
  - Until we reach a size that satisfies
  - Further division will be too small
  - Here a request of 7 KB
  - May suffer from internal fragment.



Other approaches

- Binary Buddy Allocator … contd.
- What happens when a block is free'd?
- We see if its buddy is free, coalesce them!
- Continue upper level, recursively!
- Free() only receives address; so how do we determine the address of buddy?
  - Buddy's address differs by one bit; determined by the node level in tree



Summary

- We learned about basic forms of memory allocators
- Making a fast, space-efficient, scalable allocator that works well for a broad range of workloads remains an on-going challenge