csci 3411: Operating Systems

Memory Management

Gabriel Parmer

Slides evolved from Silberschatz and West

Memory

- Memory/Storage Hierarchy

	size	speed	managed by
Registers	<1K	1 cyc	*?*
Cache	<16M	3-50	*?*
Memory	<64G	150-500	OS → today
Disk	>100G	forever	OS

- We *want* everything to be as fast as registers

  ...but as large as disks!

  - *Why can't we have this, but how can we try anyway?*

Memory Management

- Memory = array of bits from 0 → MAX_MEM

- *Can programs execute with this simple memory?*

  ...so aren't we done?

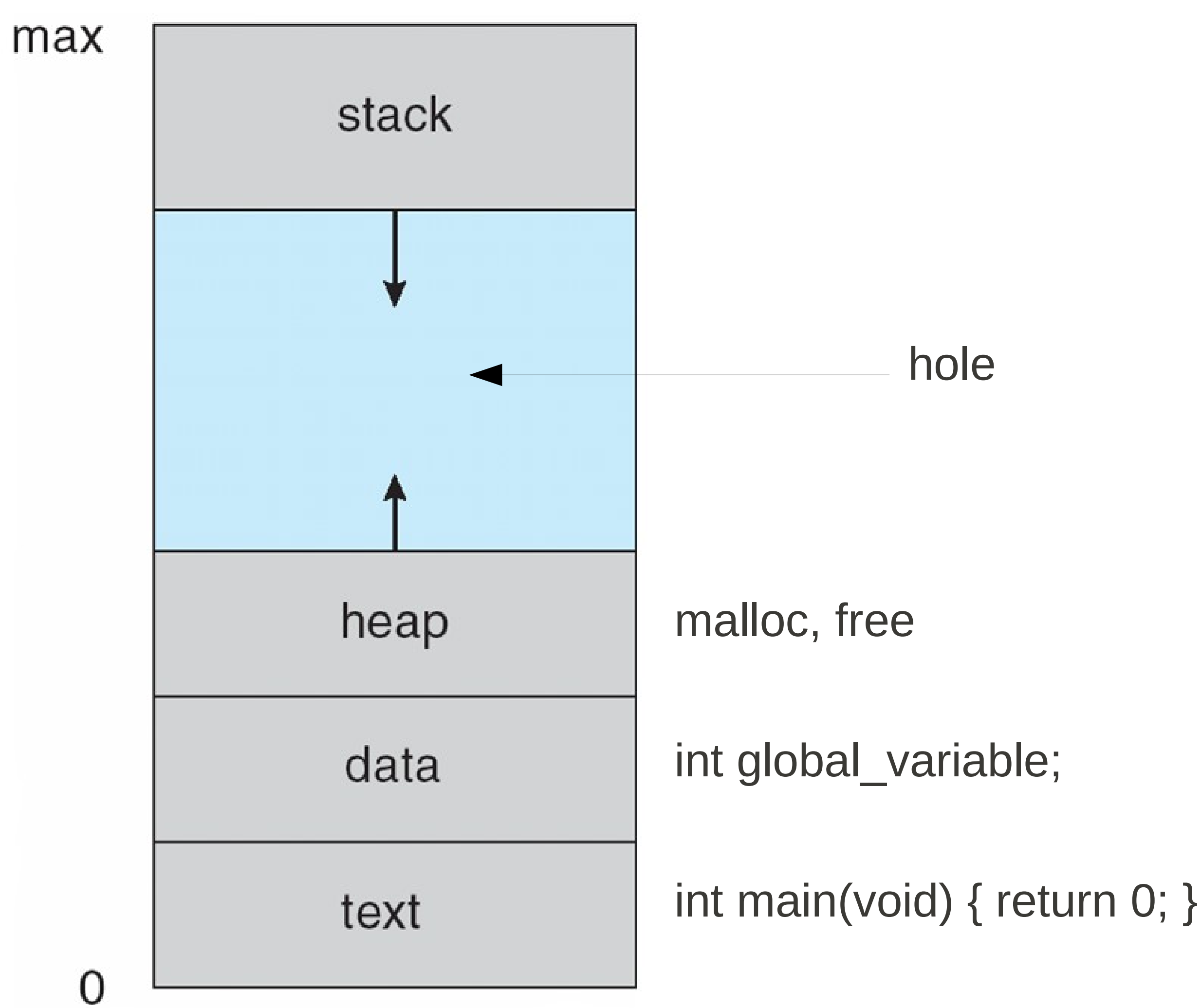
- *How* should the OS manage memory
  - *Problems with memory as an array split between processes?*

Each Process has its Own Little World

Picture from “The Matrix”, Warner Bros. Pictures

- Virtual Address Space
  - What does this mean? Address Space? Virtual?
- What benefits does this provide?
  - Hint: “*The matrix is control*”

Process Memory Layout

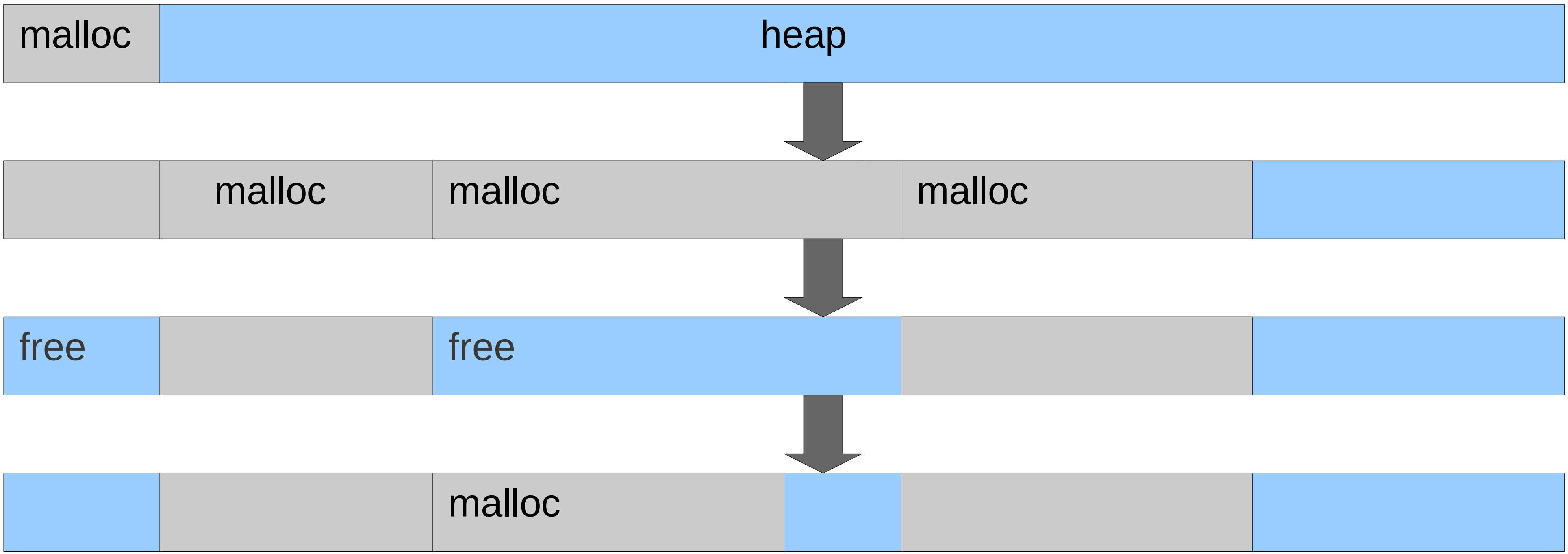


Processes' Memory

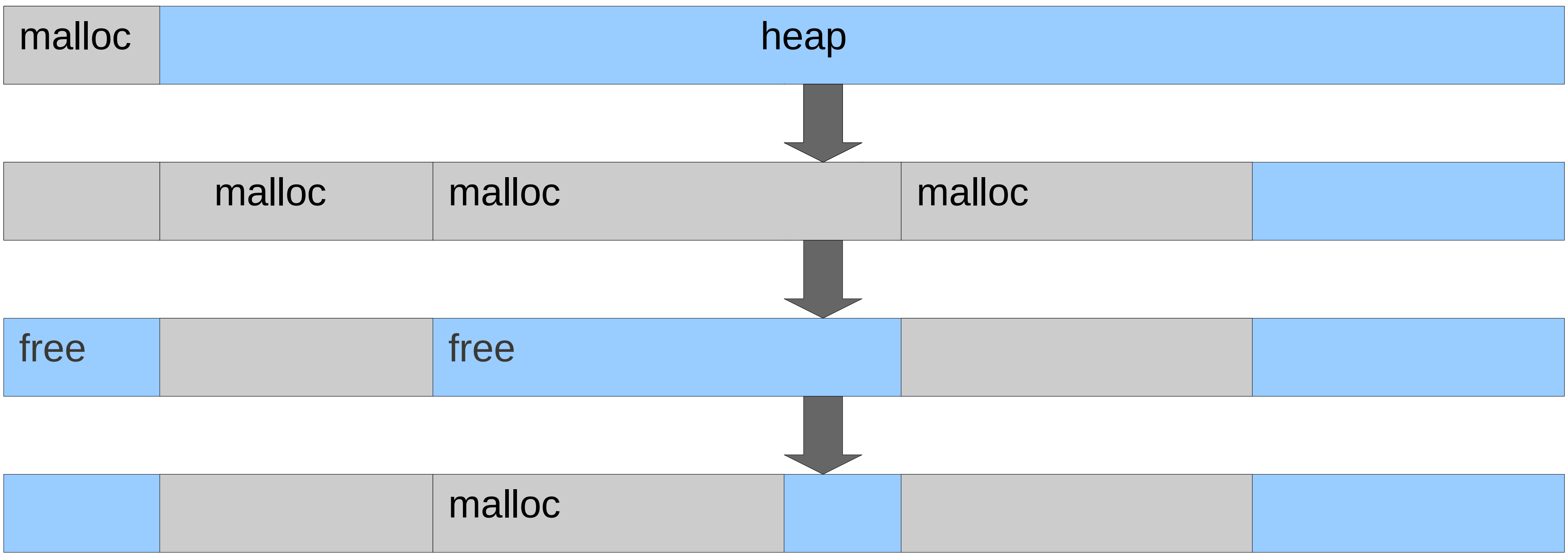
- Stack – grows down (on x86 at least)
  - What manages the stack? Uses its memory, and grows it?
- Heap
  - *malloc*, *free* – how are these implemented? syscalls?

- Memory allocation/deallocation is difficult
  - Efficiency
  - Good usage of memory (minimal waste)
  - Where do you keep the data-structures to describe memory?

Memory Allocation Algorithms

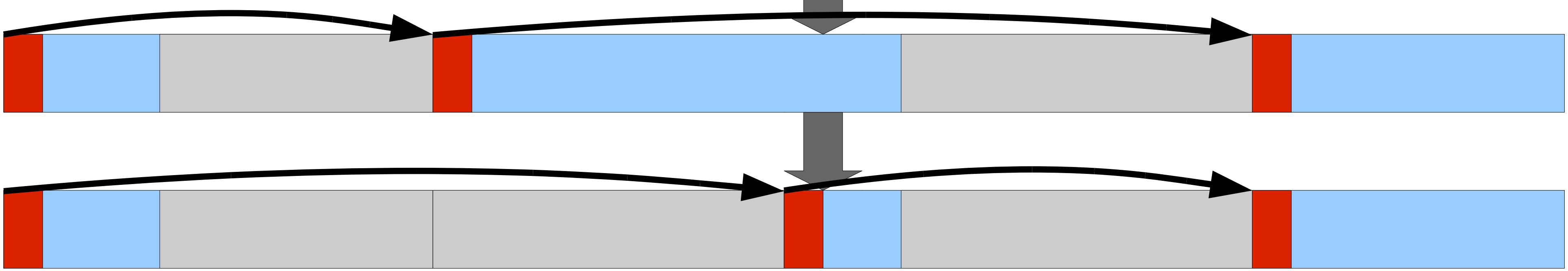


Memory Allocation Algorithms



- How do we track the free “holes”?
- When *free* is called, how do we know how large the memory chunk to free is?

How do we track the “holes”

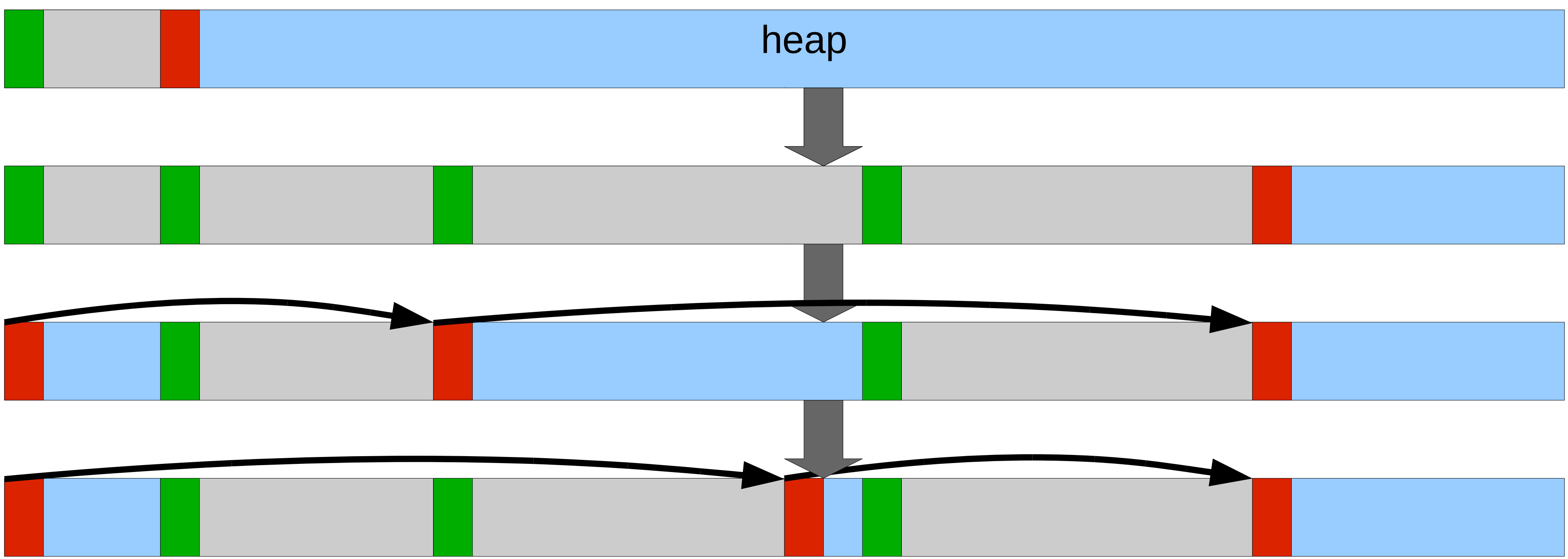



```
struct free_list {
    int size_of_hole;
    struct free_list *next;
}
```

- A *freelist* is born!
  - Linked list...
  - of free memory

free(*mem*): size of *mem*?

- When *free* is called, how does the system know how large the memory chunk to free was?




```
struct header {
  int memory_chunk_size;
}
```

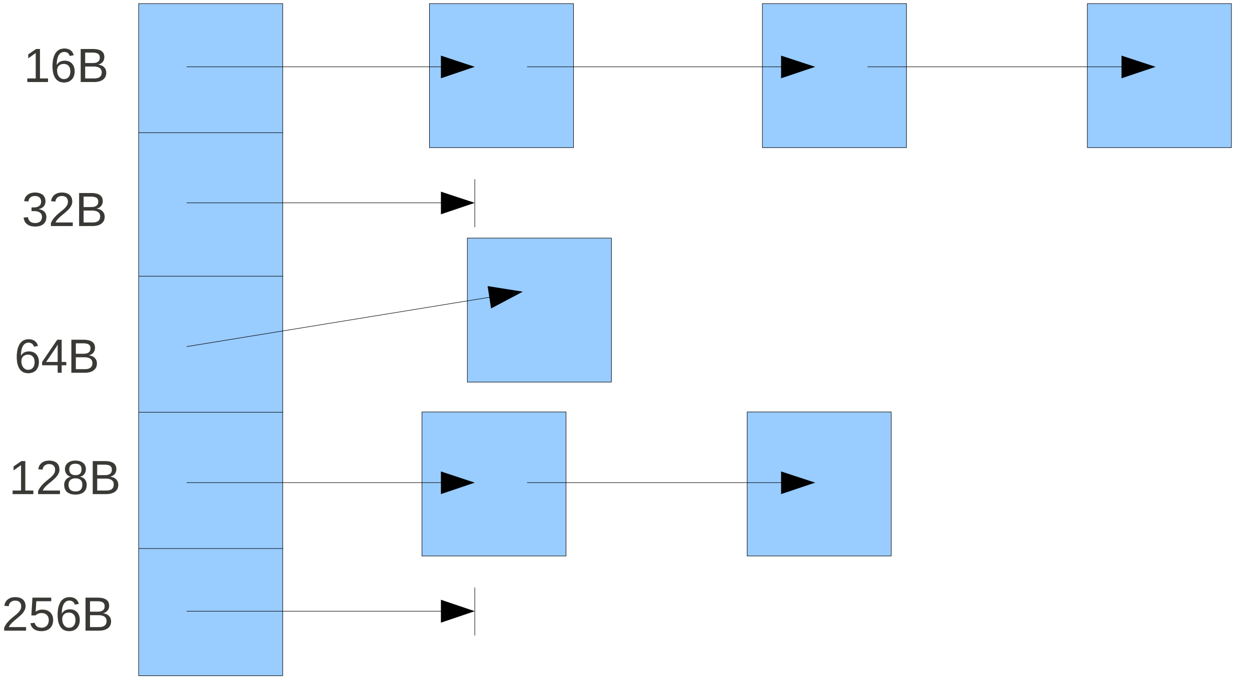
– structure directly *before* allocated memory to track size

Allocation Algorithms

- Given a freelist
  - *First fit* – allocate the first hole that is big enough
  - *Best fit* – allocate the hole that results in the smallest hole after allocation
- Tradeoffs? WRT what metrics?
- *More intelligent freelists?*

Allocation Algorithms

- More intelligent freelists
  - Example: *Power-of-two allocator* – multiple freelists, one for each power of 2

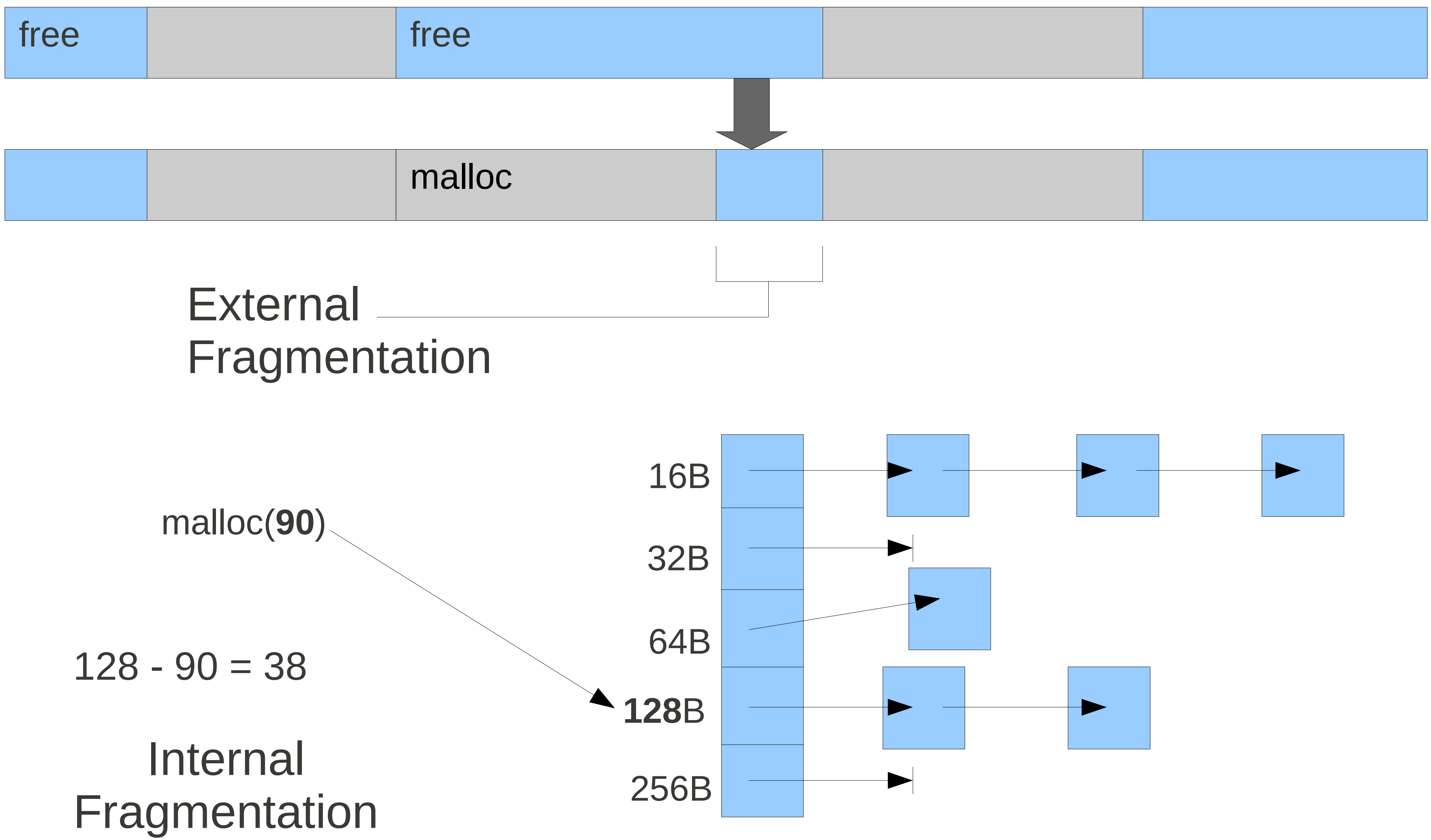


- Tradeoffs? WRT what metrics?

Allocation Algorithms: Goals

- Efficiency
  - Low asymptotic *AND* constant-time costs
- Minimize wasted memory – *Fragmentation*
  - External Fragmentation
    - “Holes” left after allocation when freelist chunk is larger than allocation amount
  - Internal Fragmentation
    - Difference between the amount of allocation requested, and that size of the allocation made
      - e.g. most allocation algorithms won't allocate less than ~16B
- Evaluate the allocation algorithms

Fragmentation



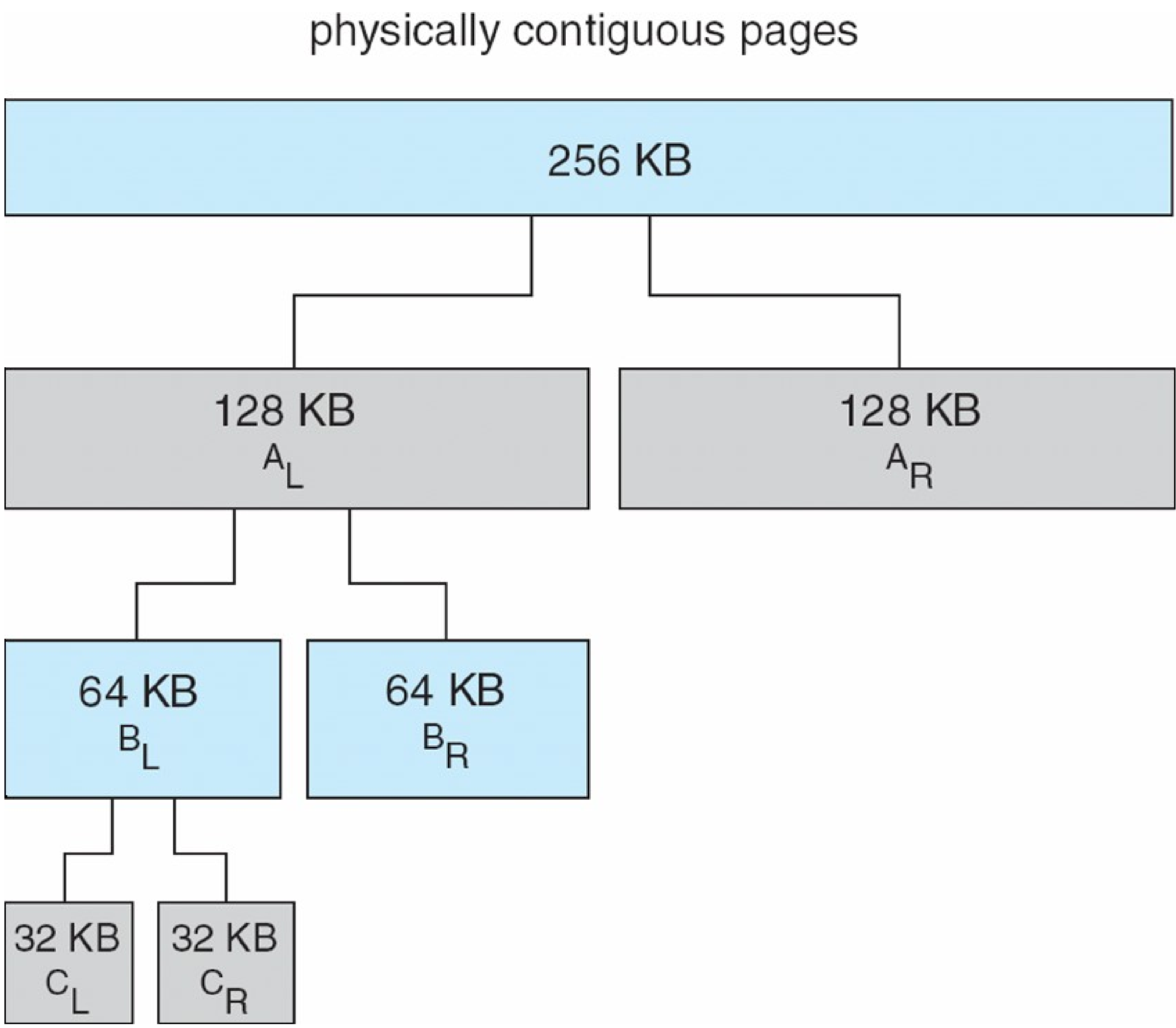
Kernel Memory Allocation

- Physical memory = big array of bytes
  - Often really chunks of some larger size = 4K
- How can we allocate these chunks?
  - Memory requests can be > 4K

- Bitmap
  - An array of bits, one per 4k chunk
  - 1: allocated, 0: free
  - Allocation: Scan for N chunks
  - *Costs?*

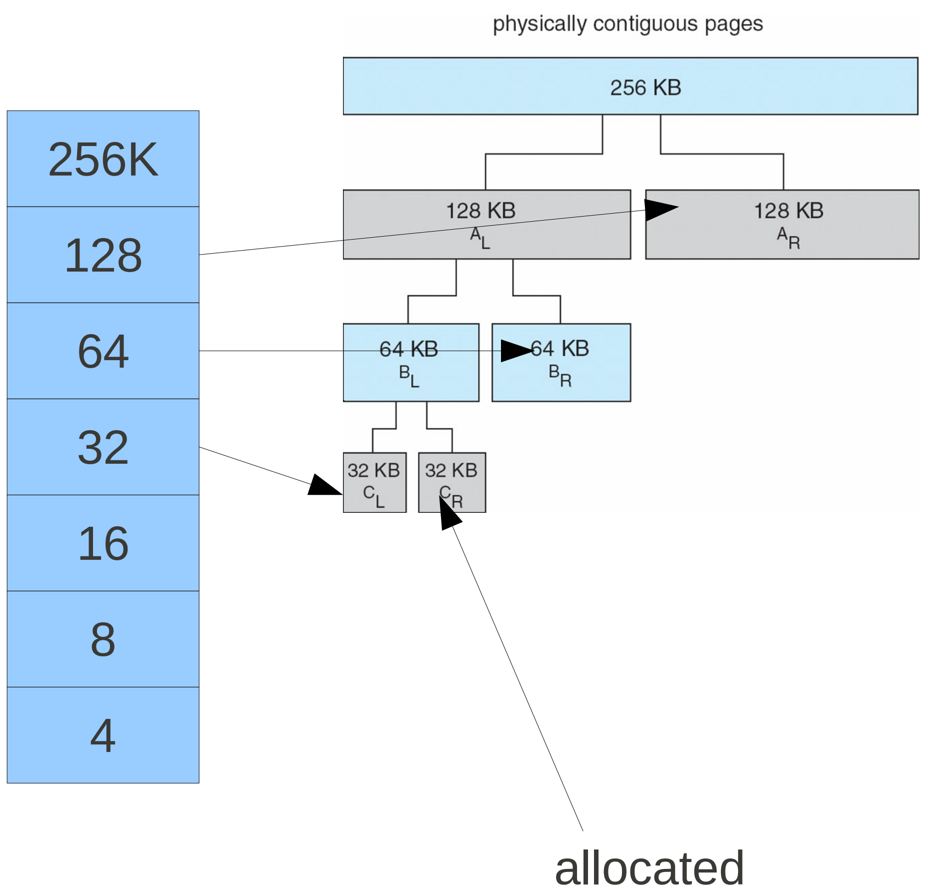
Buddy Allocation

- Power of 2 allocator
  - Start with a given amount of memory
  - Assume 4K alloc granularity
  - For a request
    - recursively break up memory (div 2)
    - Till we have chunk of smallest size
- Difficult to provide higher-order allocations
  - *Coalesce* unallocated siblings
- Downsides/Benefits?
- Used to allocate *orders* of pages for user *or* kernel lvl



Buddy Allocation II

- Implementation
  - Freelists?
    - Cost of allocation?
    - Cost of coalescing?
  - Other options?
    - Free: how find allocation size?
      - Can't store meta-data in memory chunk (!= pow 2)
    - Bitmap per allocate size?
      - Cost of allocation
      - Cost of coalescing
      - Memory wasted?

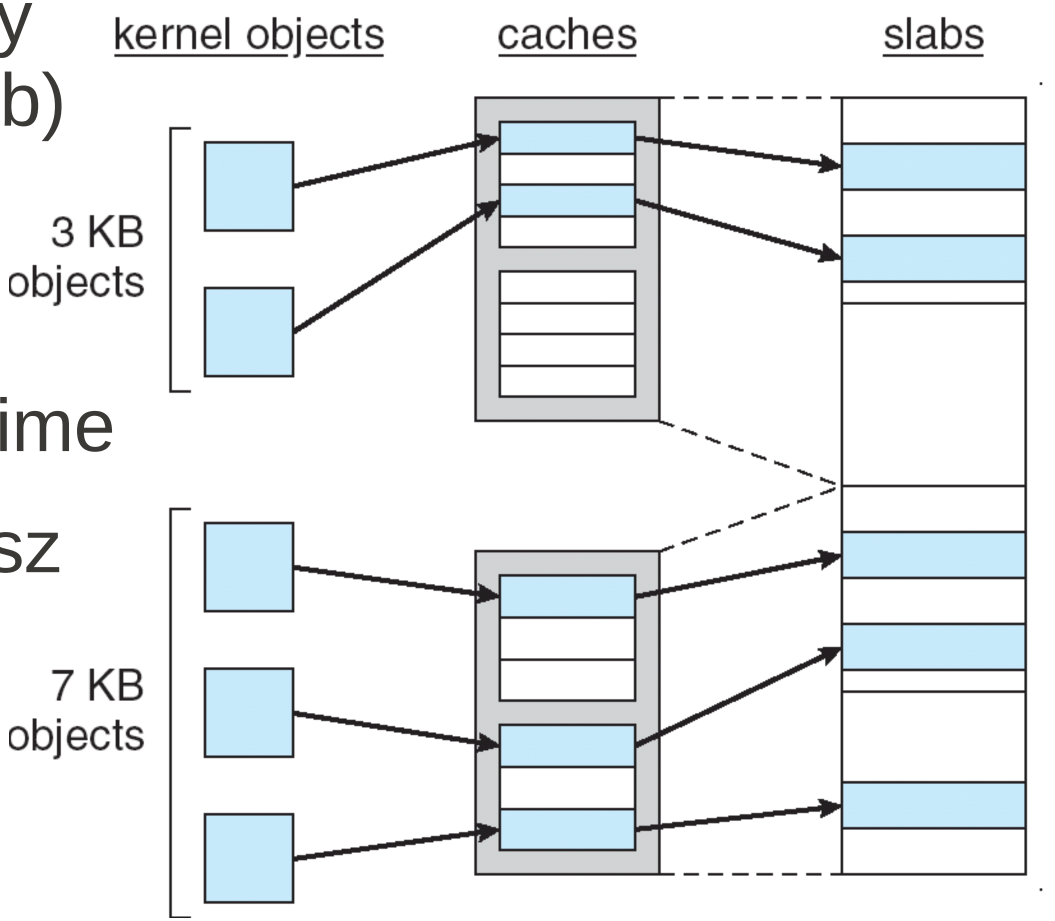


Slab Allocation

- Goals:
  - Allocation of exact memory size needed; no extern. frag.
    - Larger/smaller than page
  - Fast allocation/deallocation
- *Objects* are the actual used memory
- *Caches* consist of one or more slabs
  - One cache per object *type/size* (like pow2, but diff sz)
  - Tracks allocated objects
- Allocate *slabs* of memory using buddy allocation

Slab Allocation II

```
struct tcb { … };
struct tcb *thd = slab_alloc_tcb();
// thd memory size exactly
// equal to sizeof(struct tcb)
slab_free_tcb(thd);
```



- Slab created at compile-time
- Objects exactly the right sz
- *slab* = array of obj
- *cache* = freelist of slabs

Slab Allocation III

- E.g.: Object is 3K, Slab size is 12K
- Cache tracks 4 (12/3) objects per slab
  - Every 4 objects allocated → ask buddy alloc for slab
- When *all* objects in slab are *freed*, free slab
- When allocate object, which slab should we use?
  - *Most full slab? Empty? In between?*
  - *Temporal/spatial locality of caches?*
- Fragmentation with slab? (e.g. slab is 16K)
- *What's best slab size? Larger/Smaller? Tradeoffs?*