# KERNEL MEMORY ALLOCATORS, PART 2

CS124 – Operating Systems

Fall 2018-2019, Lecture 17

# Last Time: Kernel Memory Allocators

- Began exploring kernel memory allocators:
  - Resource map allocators
  - Power-of-two free list allocators
  - McKusick-Karels allocator
  - Binary buddy allocators
- Each allocator is a refinement of previous allocators
- Buddy allocators are fast, nearly as fast as McKusick-Karels allocator
- Additionally, can coalesce space very easily
  - Coalescing is the extra time overhead of buddy allocators
- As with other power-of-two-based allocators, internal fragmentation is still a huge problem

# Kernel Allocators and Virtual Memory

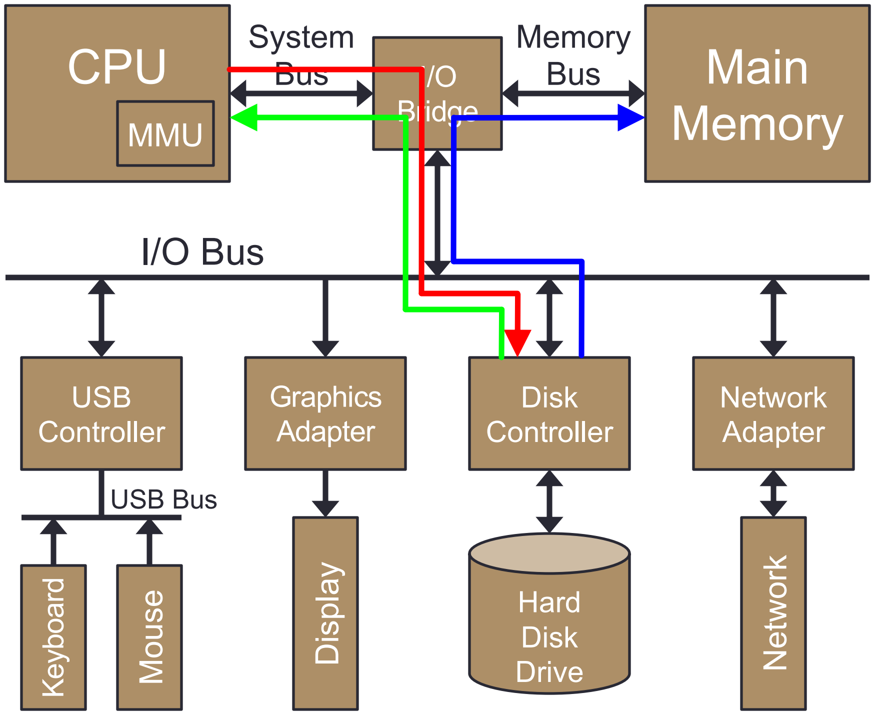
- So far, haven't considered virtual memory very deeply with our kernel allocators
  - Previous allocators are designed to work within power-of-two page size, and can also handle multiple-page allocations as needed
- Kernel allocators sometimes must care about whether the **physical** page frames are contiguous
  - i.e. a contiguous virtual address range maps to a contiguous physical address range
- This is unusual for user-space memory allocations
  - A user process sees a contiguous virtual address range, but virtual pages frequently map to non-contiguous physical page frames
  - Makes <u>absolutely no difference</u> to the user process whether the physical page frames are contiguous or not
- Two reasons why kernel allocators need to support this

# Kernel Allocators and Virtual Memory (2)

- Reason 1 (less critical): reduce overhead of kernel page-table management
- Every time a CPU's page table changes, the Translation Lookaside Buffers must be flushed
  - e.g. switching to a completely different page table (context switch)
  - e.g. changing entries in the existing page table (allocation/swap)
- When a kernel sets up a memory pool:
  - Reserve a contiguous array of physical page frames for the kernel
  - Generate a simple mapping from virtual pages to physical frames
- As long as continuity of page frames can be preserved, can avoid changing the kernel's page table
  - Can avoid flushing the Translation Lookaside Buffers as frequently

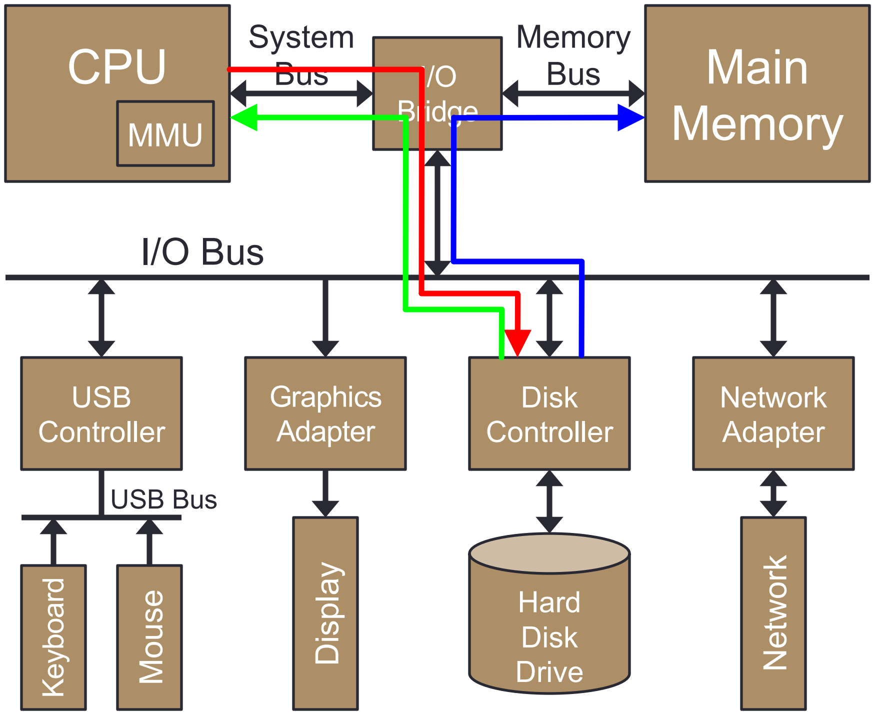
# Kernel Allocators and Virtual Memory (3)

- Reason 2 is due to Direct Memory Access (DMA) transfers from peripherals
- CPU sets up a DMA transfer from a peripheral to main memory, then does other things in the meantime
- The peripheral carries out the DMA transfer, interacting directly with memory
- The peripheral signals the CPU that DMA transfer is complete, via an interrupt
- **Does the peripheral use virtual addresses for its DMA transfers?**



# Kernel Allocator and Virtual Memory (5)

- Peripherals frequently use physical addresses for memory interactions
- Reason 2: If a DMA transfer requires a buffer of multiple pages, the pages must be contiguous in physical memory, not just in virtual memory
  - Frequently must also be within a specific address range




- For DMA support, kernel allocators often must provide a way to allocate physically contiguous regions

# Cached-Object Allocators

- Modern OSes tend to have kernel memory allocators based on large caches of frequently used objects
- Principle: allocate a large region of memory for each kind of object the kernel must dynamically allocate
  - Pack as many objects of each kind into the large region
  - Amortize cost of memory allocation over a large number of objects
  - Virtually eliminate both external and internal fragmentation
- For most allocations and releases, the operation will be constant-time
  - Rarely, need to request more memory for a given kind of object
  - Similarly, must release unused memory regions from time to time

# Cached-Object Allocators (2)

- Another benefit of such allocators: improved CPU cache usage as compared to power-of-two free list allocators
- All memory blocks returned by power-of-two free list allocators start on power-of-two address boundaries…
  - Exacerbates the issue of conflict-misses in direct-mapped and set-associative caches
  - Different blocks will be much more likely to map to same cache line
- By packing many objects into a larger contiguous memory region, addresses will be much more uniformly distributed
  - Far less likely to have blocks start on power-of-two address boundaries (as long as objects aren't a power of two in size…)
  - Generally much friendlier to set-associative caches

# Mach Zone Allocator

- The Mach kernel implements a **zone allocator**
  - A fast allocator that includes a page-level garbage collector
- The zone allocator manages memory areas called **zones**: regions of memory devoted to specific kinds of objects
  - Allows very tight packing of objects within a given zone
- Kernel keeps a zone for each kind of objects it needs
  - Process and thread control blocks, virtual memory map objects
  - Semaphores, pipes, kernel notification objects
  - Timer event details, alarm objects
  - Zone objects themselves are kept in a "zone of zones"
  - etc.
- Example: on Mac OS X, **`sudo zprint`** lists all zones

# Mach Zone Allocator (2)

- Each zone is a cache of objects of a specific type and size
- A **`zone`** structure records, among other things:
  - The name of the zone (e.g. "semaphores" or "threads")
  - The size of elements (i.e. objects) stored in the zone
  - The current size of the zone, the maximum size of the zone, and how much to increase the zone by when more space is needed
    - All these values are multiples of virtual memory pages
  - How many objects are in-use in the zone
- All zones are chained together into a linked list via the **`next_zone`** pointer
- Free elements in the zone are chained together via **`free_elements`** pointer
  - **`vm_offset_t`** is basically a **`void *`**

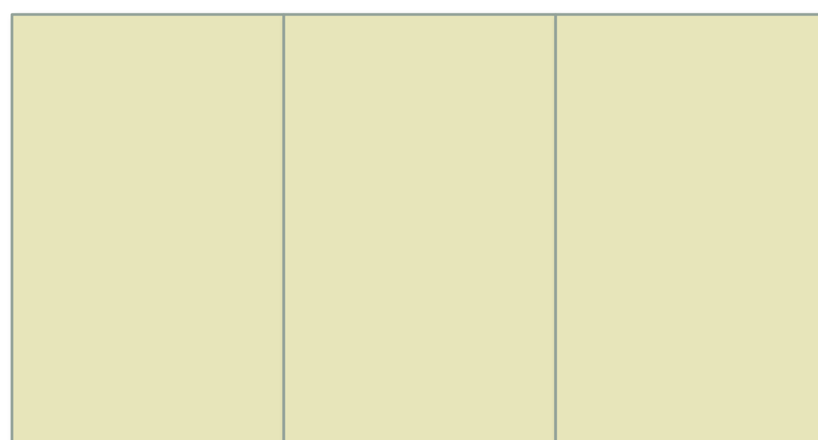
**`zone`** struct (partial):

```
const char *zone_name

vm_size_t cur_size
vm_size_t max_size
vm_size_t alloc_size

vm_size_t elem_size

vm_offset_t free_elements

zone *next_zone
```

# Mach Zone Allocator (3)

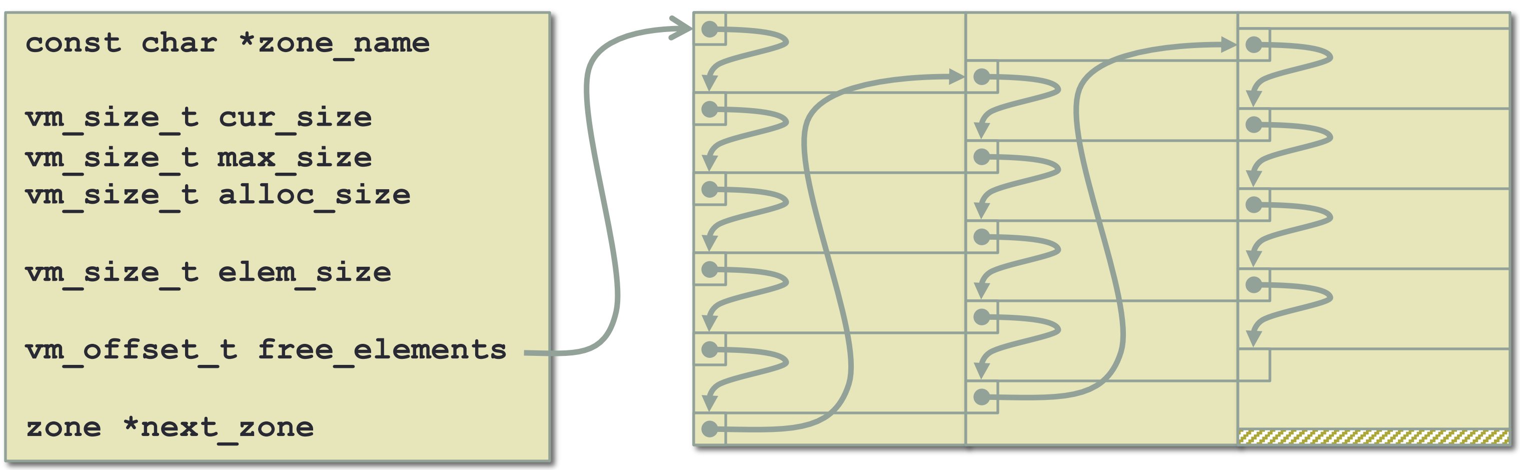
- Each zone has a set of (0+) virtual pages associated with it, for storing objects in that zone
- When a zone's available memory must be increased:
  - Zone allocator requests a contiguous region of virtual pages
  - Specific number of pages requested depends on system state
  - Normally, will be number of pages required for **`alloc_size`**
  - If low memory (or an earlier request for pages failed), will be **`elem_size`** rounded up to a whole number of virtual pages

```
const char *zone_name

vm_size_t cur_size
vm_size_t max_size
vm_size_t alloc_size

vm_size_t elem_size

vm_offset_t free_elements

zone *next_zone
```



# Mach Zone Allocator (4)

- When zone's available memory must be increased (cont.)
  - New memory region is chopped up into elements of the specified **`elem_size`**
    - <u>Important Note</u>: individual elements may span adjacent virtual pages (fine, since the virtual memory region is contiguous)
  - Elements in the region are linked together into list of free elements
- This process of increasing memory can be repeated as needed, until the zone reaches its maximum size

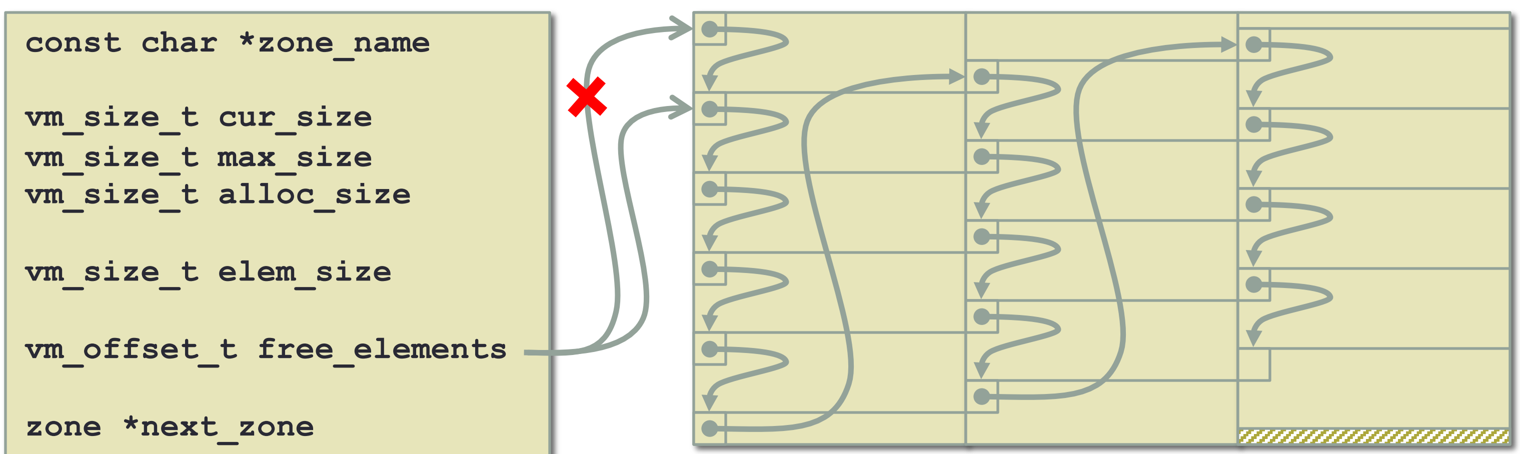


# Mach Zone Allocator (5)

- When an object of a particular type is required:

```
/* zone_t is a pointer to a zone struct */
void * zalloc(zone_t zone)
```

  - i.e. kernel subsystems will hold a pointer to their required zone(s)
- If zone has no free elements, zone memory is increased
- Zone allocator returns first object in list of free elements
  - Updates free list to point to next element in list



# Mach Zone Allocator (6)

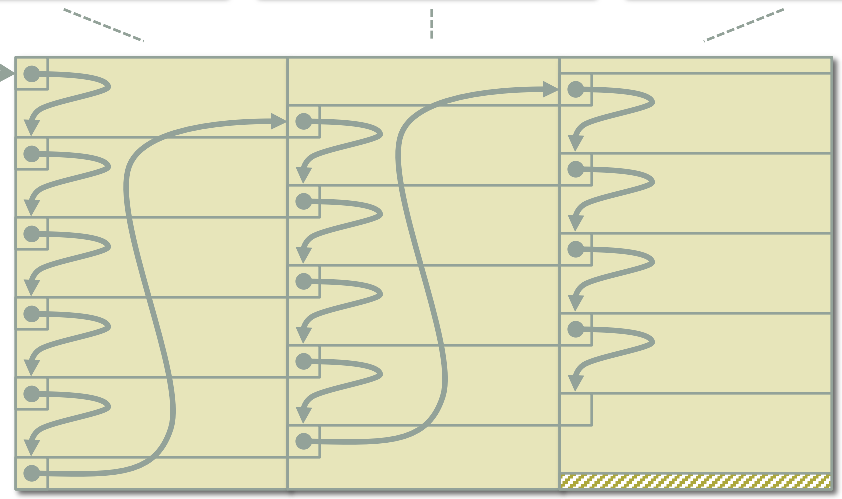
- The zone allocator has a garbage collection mechanism to reclaim pages that are unused
- Allocator maintains a **`zone_page_table_entry`** struct for every virtual page used by the zone allocator
- At initialization, each struct's **`alloc_count`** is set to the number of objects that fall within the corresponding page

```
alloc_count = 6
collect_count
next
```

```
alloc_count = 6
collect_count
next
```

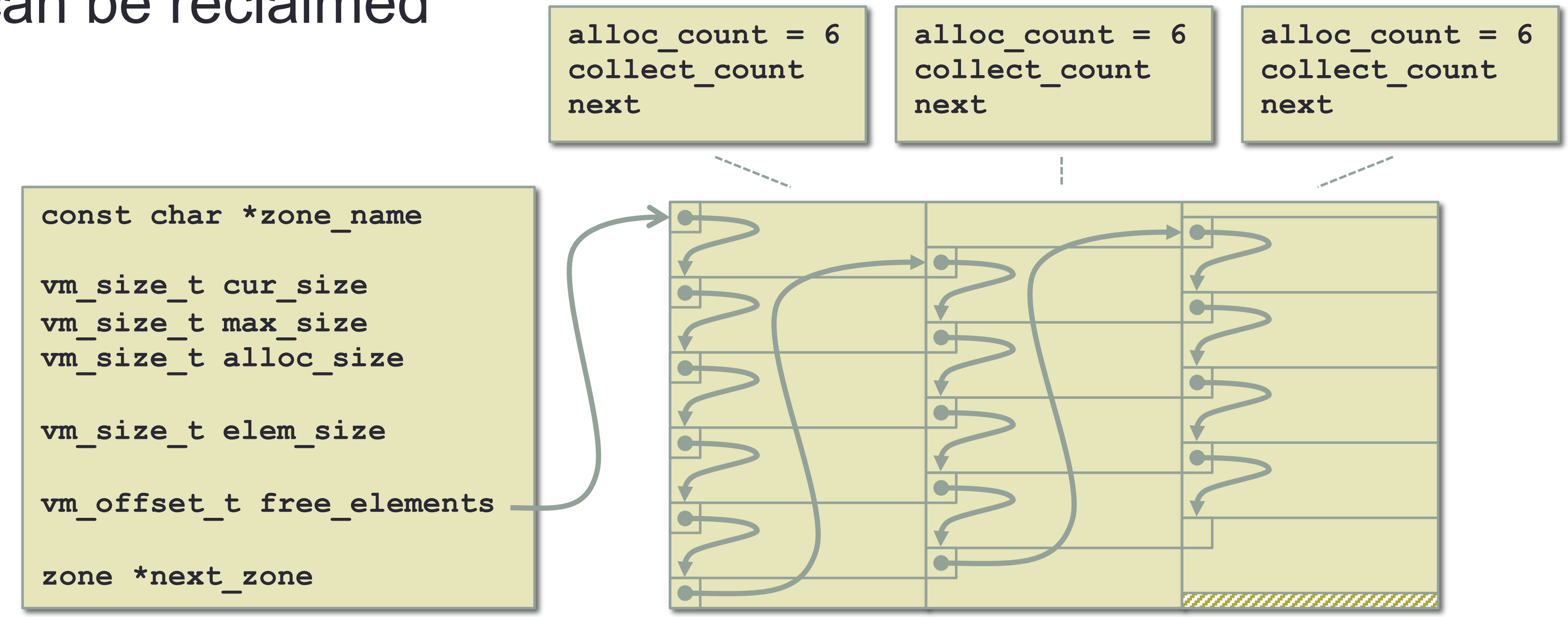
```
alloc_count = 6
collect_count
next
```

```
const char *zone_name

vm_size_t cur_size
vm_size_t max_size
vm_size_t alloc_size

vm_size_t elem_size

vm_offset_t free_elements

zone *next_zone
```



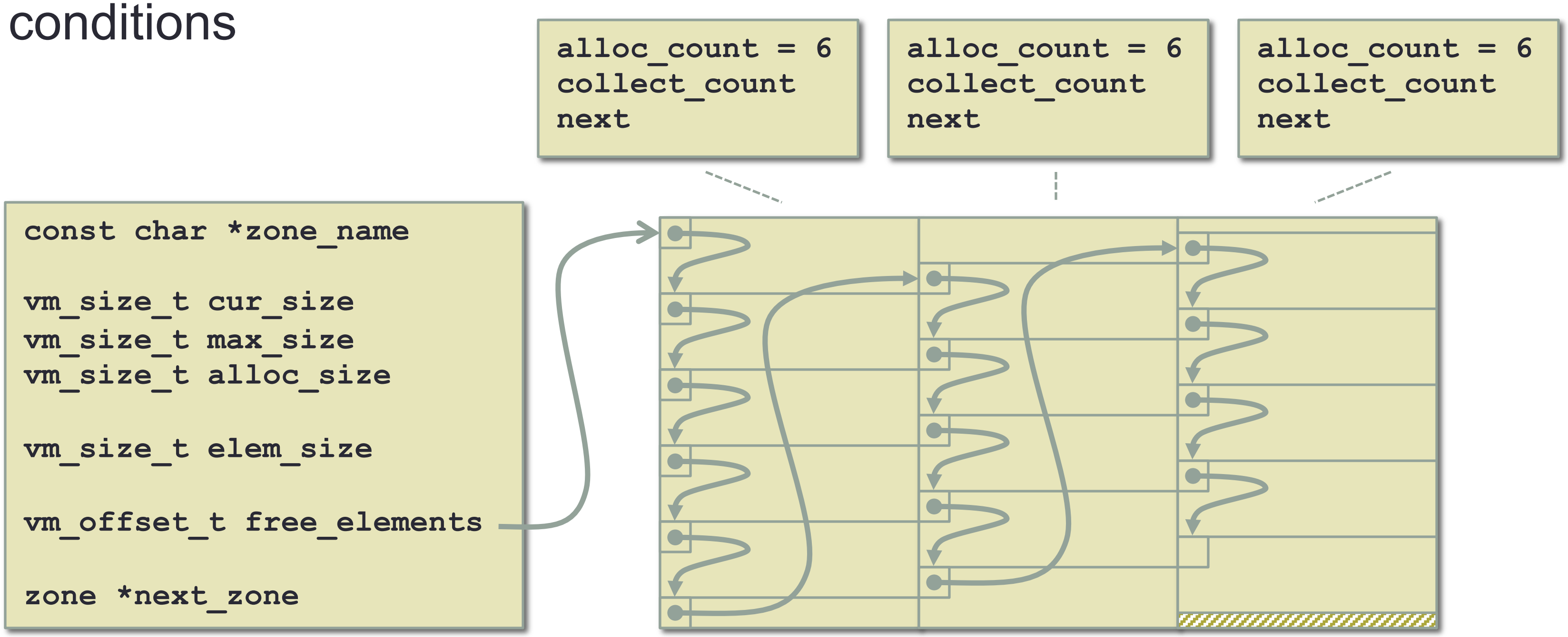
# Mach Zone Allocator (7)

- First, garbage collector sets all **`collect_counts`** = 0
- Next, garbage collector traverses the free lists of <u>all</u> zones
  - For each free element:
    - Identify the virtual page that the element falls within
    - Increment that page's **`collect_count`** value by 1
- If a page's **`alloc_count`** == **`collect_count`**, the page can be reclaimed



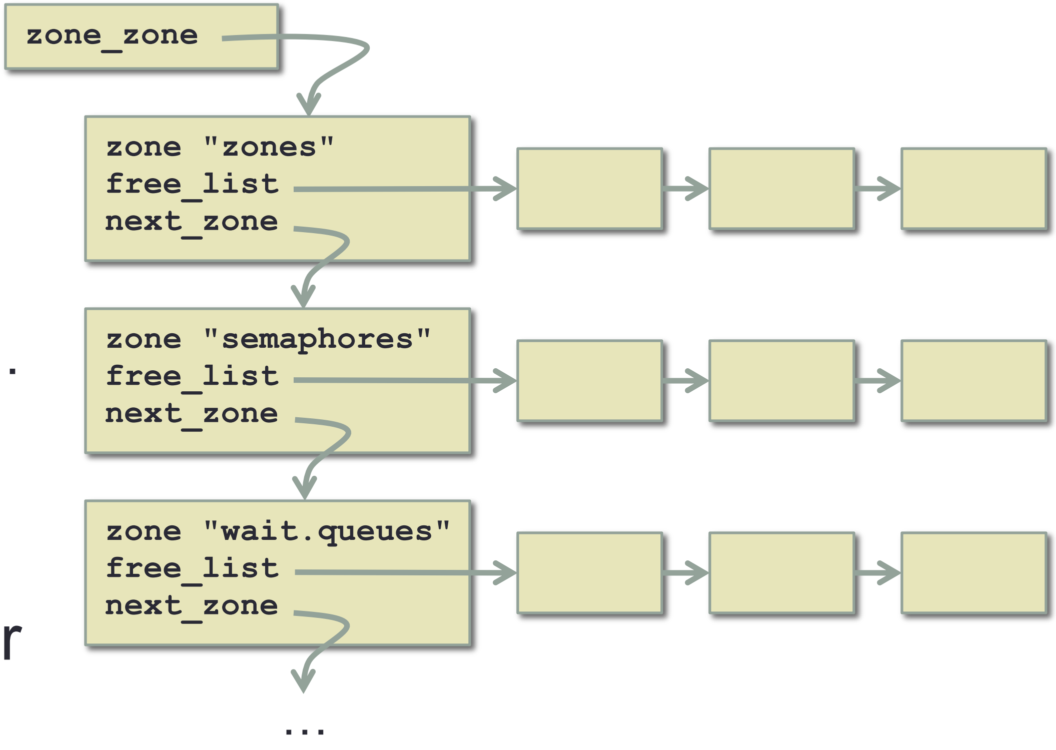
# Mach Zone Allocator (8)

- Finally, garbage collector makes a second sweep through all virtual pages used by the allocator
  - Any page with **`alloc_count`** == **`collect_count`** is queued up to eventually be freed
- GC is run when the virtual memory pager is invoked
  - Also, GC is sometimes run when objects are freed in low-memory conditions



# Mach Zone Allocator (9)

- As mentioned earlier, zone allocator maintains a zone for every kind of object the kernel needs to allocate
  - Including a "zone of zones" for **`zone`** structs used by the allocator
- Not every allocation is for a specific kind of object…
  - May be a buffer, or a small chunk of memory
- Zone allocator includes many zones for general allocation requests
  - e.g. kalloc.16, kalloc.32, …
  - e.g. buf.512, buf.1024, …
- Allows general kernel memory requests to be served by zone allocator



# Mach Zone Allocator (10)

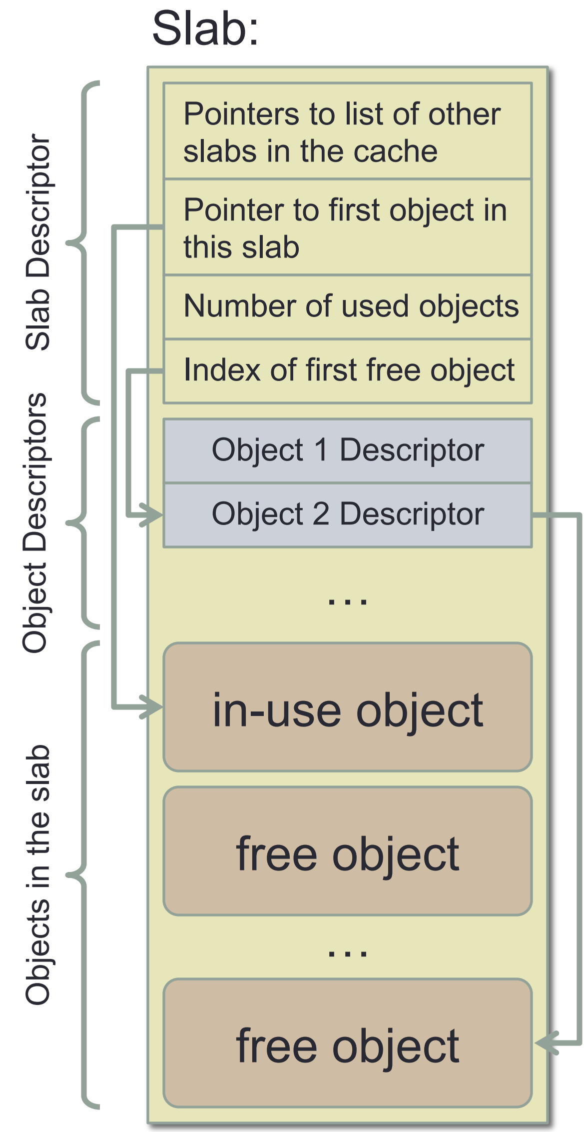
- Generally, the zone allocator is extremely fast
  - Cost of acquiring pages and allocating memory is amortized over many objects of the same type and size
  - Very good at handling allocate-free-reallocate memory interactions
- Any issues center around the garbage collector
  - Designed to handle scenarios when memory usage is bursty, so that pages aren't held by the zone allocator for a long time
  - Does add some overhead to virtual memory paging system (the zone allocator's GC is invoked by the pager)
    - If the system has to swap some pages to disk, might as well look for zone-allocator pages we aren't using anymore, while we're at it…
  - Can impact system performance in unpredictable ways
  - (Problems can't be *that* severe – Mac OS X uses it after all…)

# Slab Allocators

- Sun Solaris 2.4 kernel introduced a **slab allocator**
  - Identical concept to zone allocator; different implementation details
- A **slab** is a sequence of virtual memory pages
  - Often constrained to be physically contiguous as well
- A **cache** holds kernel objects of a specific type
  - Similar to Mach's zones
  - A cache is created for each kind of object the kernel needs to dynamically allocate
- A cache has zero or more slabs for storing kernel objects
  - e.g. when the cache is initially created, it has no slabs
  - All slabs in a given cache hold the same kind of kernel object
- As with the zone allocator, a kernel object may span multiple adjacent pages in a given slab

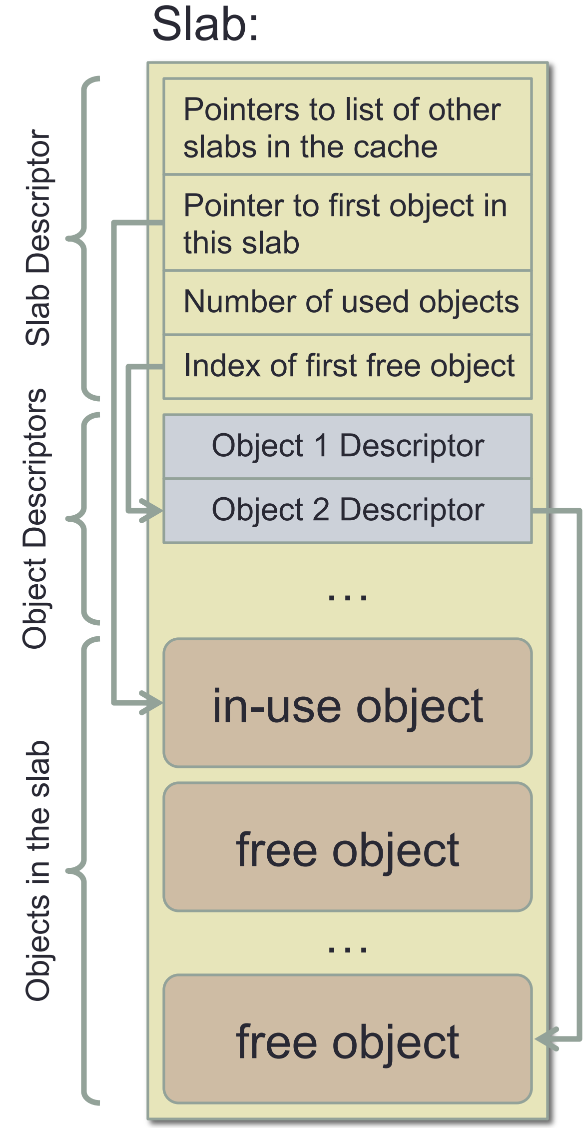
# Slab Allocators (2)

- Slabs hold some details about the objects they contain
  - (In zone allocator, memory regions don't hold any additional info)
  - Pointers to other slabs in the same cache
  - Pointer to first object in the slab
    - i.e. pointer to start of the array of objects
  - How many objects are currently in use
  - Index of the first free object in the slab (or a special constant if slab is full)
- Each object also has its own descriptor
  - If the object is free, descriptor holds the index of the next free object in the slab
  - (A special constant indicates end of list)



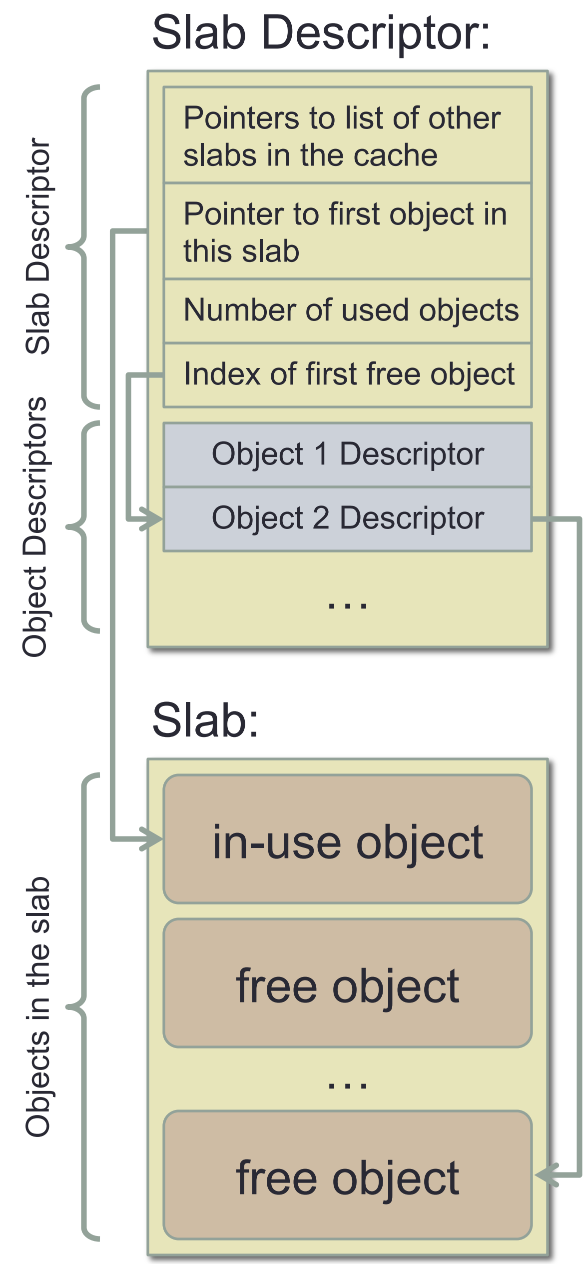
# Slab Allocators (3)

- Slab and object descriptors consume some useful space within each slab…
- Depending on object size, descriptors may be factored into a separate object
  - External slab descriptors are stored in their own caches, but will vary in size due to the number of objects that fit within the slab
  - Internal slab descriptors are simpler, but consume useful slab space
- For "large" objects (e.g. objects > 1/8 the page size), separate out descriptors
  - Because cached objects are large, count of object descriptors will be small
  - External descriptor objects will be small



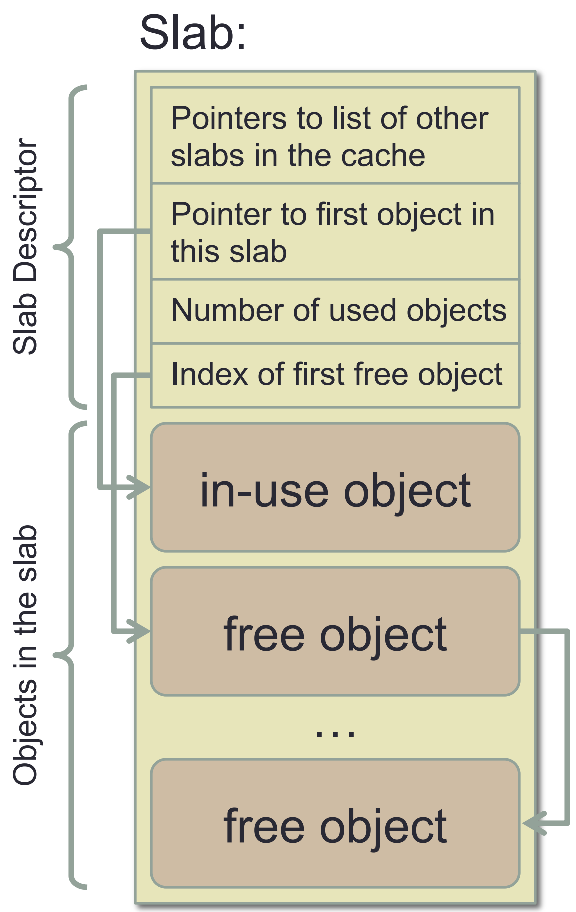
# Slab Allocators (4)

- For "large" objects (e.g. objects > 1/8 the page size), separate out descriptors
- Cached objects are large, so fewer of them will fit within a given slab
  - Count of object descriptors will be small…
  - External slab descriptors will be small
- Slab allocator can use a separate cache for external slab descriptor objects



# Slab Allocators (5)

- For "small" objects (e.g. object < 1/8 the page size), object descriptor table will consume significant space…
  - Many objects will fit in the slab, so will require many object descriptors
- Store the free-object list within the free objects themselves
  - The space isn't being used for anything else…



# Slab Allocators (6)

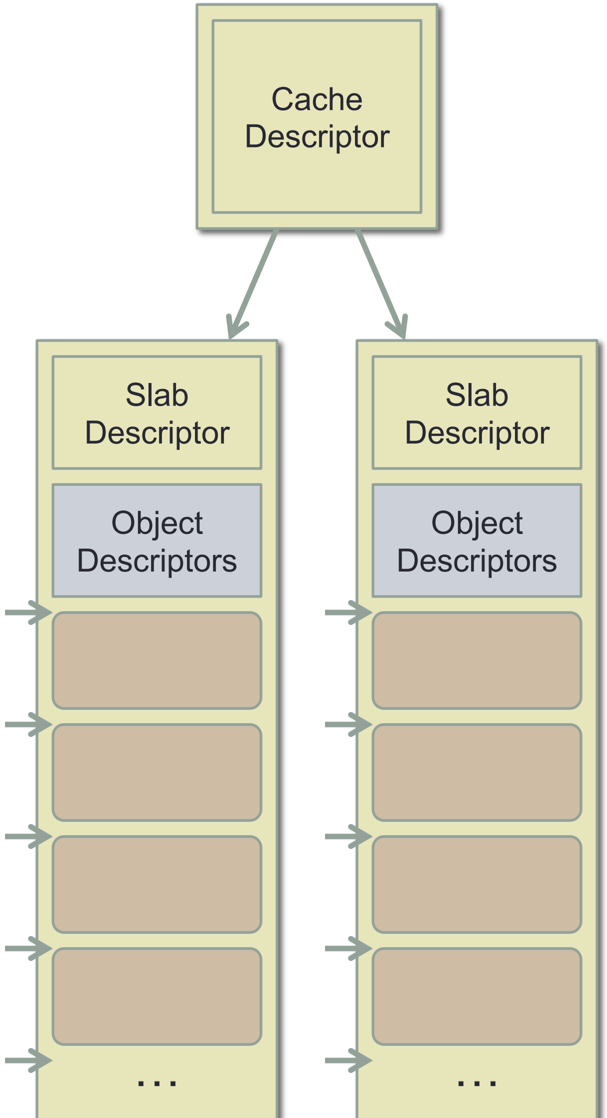
- As with the Mach zone allocator, slab allocators frequently include caches for general kernel memory allocations
  - e.g. 16-byte allocations, 32-byte allocations, etc.
- Can also include caches for DMA-friendly buffers

- Example:  Linux slab allocator
  - Includes caches for DMA-friendly buffers of varying sizes
  - Also includes caches for more general-purpose buffers of the same sizes as DMA-friendly buffer caches

# Slabs and CPU Caches

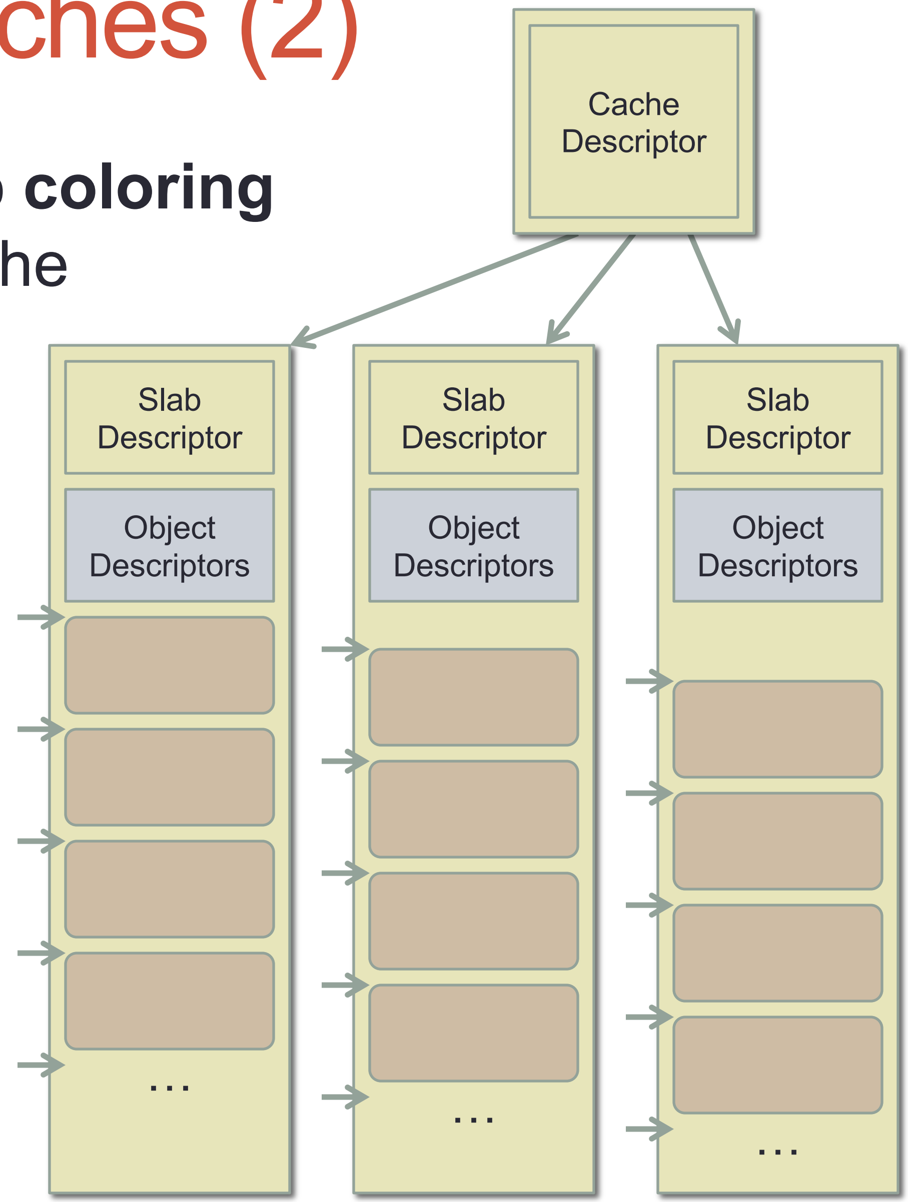
- Slabs can cause CPU cache issues…
- Scenario: multiple slabs in a particular object cache
- Slabs' starting addresses are aligned with power-of-two boundaries…
- Individual objects in the slabs are at same offsets from start of slabs

- Greatly increased likelihood of CPU cache conflict-misses ☹



# Slabs and CPU Caches (2)

- Slab allocator can apply **slab coloring** to new slabs in an object cache
- For each new slab, add a different offset to the start of objects in the slab
  - Offset is called the slab's **color**
- Each object cache specifies min/max color, increment
  - Ensures that objects are still properly word-aligned
  - Takes advantage of left-over space in slabs when a whole number of objects don't fit



# Linux Slab Allocator

- Linux began to use a slab allocator in the 2.2 kernel
- Problem: over time, can become very difficult to allocate large regions of physically contiguous virtual pages
- Linux uses a binary buddy allocator to allocate physically contiguous sequences of virtual pages for slabs
  - Sits beneath the slab allocator, for handling requests for new slabs
  - Supports size orders from 1 page, up to 1024 contiguous pages
  - Can easily coalesce adjacent regions of physically contiguous pages when slabs are released back to the memory pool
- In Linux slab allocator, each cache contains slabs that are all the same size
  - (In zone allocator, different zones may have memory regions with varying numbers of pages in them)

# Linux Slab Allocator (2)

- In Linux, slabs can be in three different possible states:
  - Full – all objects in the slab are marked as used
  - Empty – all objects in the slab are marked as free
  - Partial – the slab contains both used and free objects
- Slabs in various states are maintained in different linked lists within the object cache
  - Original Solaris slab allocator maintained each cache's slabs in a single list, ordered on the slab's state
- When an allocation request is made:
  - Slab allocator tries to use a partial slab first, to satisfy the request
  - Avoids external fragmentation across multiple slabs; preserves empty slabs so they can be reclaimed more easily, when necessary
  - An empty slab is only used if the cache has no partial slabs