ECE391
Computer System Engineering

Lecture 22

Dr. Jian Huang

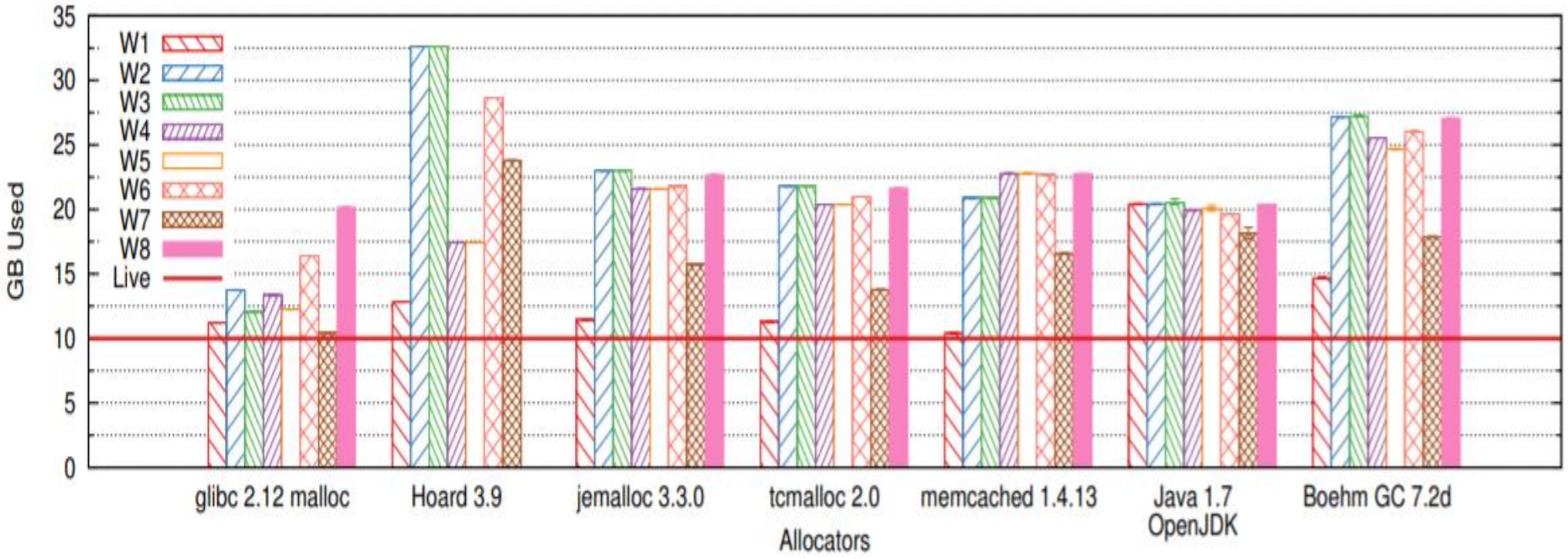
University of Illinois at Urbana- Champaign

Fall 2018

Lecture Topics

- Memory allocation interfaces in the kernel
  - kmalloc
  - slab caches
  - vmalloc
  - buddy system

Why Memory Allocation?



[Rumble et al., FAST'14]

Files for Memory Management

- headers (all under linux/): gfp.h, slab.h, vmalloc.h, slab_def.h, slub_def.h
- sources: mm/slab.c, mm/vmalloc.c
- swap-related: mm/swap.c, mm/swapfile.c, mm/page_alloc.c

Memory Allocation

- Overview
  - a few small items → kmalloc
  - a lot of items, repeatedly → slab cache
  - a big, physically contiguous region → free pages
  - a big area of virtual memory → vmalloc
    (not necessarily physically contiguous)

```
void * kmalloc ( size_t size,
                 gfp_t flags);
```

- flags/allocation priorities (common to all interfaces)

GFP_ATOMIC does not sleep; succeeds only if request can be satisfied with existing free resources (unlikely for large requests)

Memory Allocation

may sleep to wait for pages

GFP_KERNEL	by kernel, drivers, etc.
GFP_NOFS	no file system calls (avoids pushing pages to disk)
GFP_NOIO	no I/O operations at all
GFP_USER	on behalf of a user (low priority)

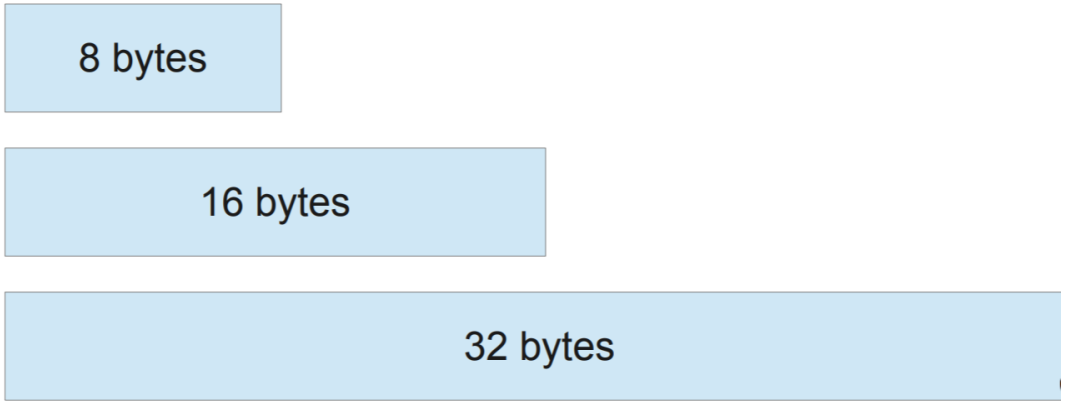
__GFP_DMA	DMA accessible (low physical addresses on some machines)
__GFP_HIGHMEM	high memory (PAE (phys. address extensions) on x86) is acceptable

(two underscores)

Basic Interface

```
void* kmalloc (size_t size, gfp_t flags);
```

- uses exponentially-sized slab caches (to be discussed)
  - ranging from 8B to several MB
  - up to 4MB in our kernel



- each allocation is contiguous in physical memory

Basic Interface

- Managing a private cache of objects (slab cache)
  - frequent allocations/deallocations
  - one cache per item type
  - physical memory is contiguous
  - protocol
    - Creation returns a page handle
    - Use handle to allocate/deallocate objects or to destroy the slab cache when done

Slab Allocator

- Kernels often allocate specific objects, not arbitrary sizes

- Initializing an object sometimes takes more time than allocating it
  - If possible, keep object initialized (e.g., call *mutex_init* just once)
  - Bring object back to its initial state at deallocation

- Key concept
  Pre-allocate caches of contiguous memory to make it efficient to allocate allocation requests for objects of a specific size.

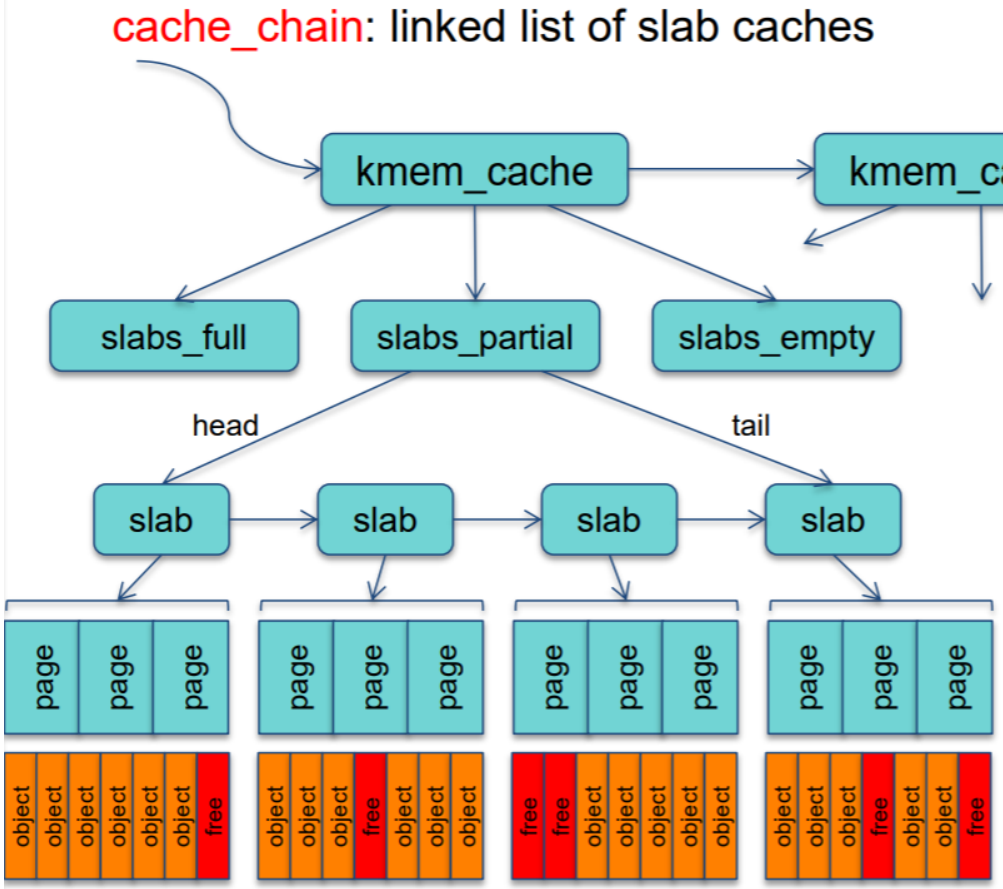
[Krzyzanowski et al. Rutgers University]

Components in Slab Allocator

- Terms
  - Object: requested unit of allocation
  - Slab: block of contiguous memory (often several pages)
    - Each slab caches similarly-sized objects
    - Avoids fragmentation problems
  - Cache: storage for a group of slabs for a specific object
    Each unique object type gets a separate cache
- Slab states
  - Empty all objects in the slab are marked as free
    - The slab can be reclaimed by the OS for other purposes
  - Full: all objects in the slab are marked as in-use
  - Partial: the slab contains free and in-use objects

[Krzyzanowski et al. Rutgers University]

Slab Allocator Structure



- Try to find a free location for the requested object on a partial slab in a cache for that type of object.
- If not found, then allocate a new slab and assign it to a cache

[Krzyzanowski et al. Rutgers University]

Slab Allocator Operation

- kmem_cache_create: create a new cache
  - Typically used when the kernel initializes or a kernel module is loaded
  - Identifies the name of the cache and size of its objects
  - Separate caches for inodes, directory entries, TCP sockets, etc.
- kmem_cache_destroy: destroy a cache
  - Typically called by a module when it is unloaded
- kmem_cache_alloc: allocate an object from a named cache
  - cache_alloc_refill may be called to add memory to the cache
- kmalloc / kfree: no object (cache) specified
  - Iterate through available caches and find one that can satisfy the size request

[Krzyzanowski et al. Rutgers University]

Basic Interface

```
kmem_cache* kmem_cache_create (const char* name,
        size_t size, size_t align, unsigned long flags,
        void (*constructor) (void*, kmem_cache*,
                             unsigned long ignored),
        void (*ignored)([same as constructor]));
```

- name used to avoid >1 cache for same structure
- size is object size; cache grows/shrinks automatically
- alignment specified for individual objects

- macro for creation

```
kmem_cache_t* KMEM_CACHE (<structure>, flags);
```

Flags for slab cache allocation

SLAB_HWCACHE_ALIGN	align objects to cache lines (makes accesses a little faster)
SLAB_CACHE_DMA	use DMA-accessible memory
SLAB_POISON	fill new memory with 0xA5A5A5A5
SLAB_RED_ZONE	bound objects with “red zones” (test buffer overruns)

- Slab cache constructor function
  - Callback function used when memory is allocated for the slab cache
  - called for each object in new slab
  - NOT called for each object allocation (*kmem_cache_alloc*)
  - third argument used to be flags (now always 0)

Slab cache API

- Slab cache allocation/deallocation and destruction

```
void* kmem_cache_alloc (kmem_cache*,
                        gfp_t flags);

void* kmem_cache_zalloc (kmem_cache*,
                         gfp_t flags);
```

  - flags are passed to lower allocator (*kmalloc*) iff a new slab is allocated
  - zalloc version zeroes memory in new object

```
void kmem_cache_free (kmem_cache*, void*);
```

Slab cache API

```
void kmem_cache_destroy (kmem_cache*);
```

– fails silently (logs error message) if not empty

```
int kmem_cache_shrink (kmem_cache*);
```

– frees empty slabs; returns 0 if all slabs released

Getting big chunks of memory

- multiples of page size (4kB on x86; ISA-dependent)
- physically contiguous

```
unsigned long get_zeroed_page (gfp_t flags);
unsigned long __get_free_page (gfp_t flags);
unsigned long __get_free_pages
            (gfp_t flags, unsigned int order);
```

- flags are same as for kmalloc
- order is log (base 2) of number of pages requested
- (the latter two function names start with two underscores)

Getting big chunks of memory

```
void free_page (unsigned long);
void free_pages (unsigned long, int order);
```

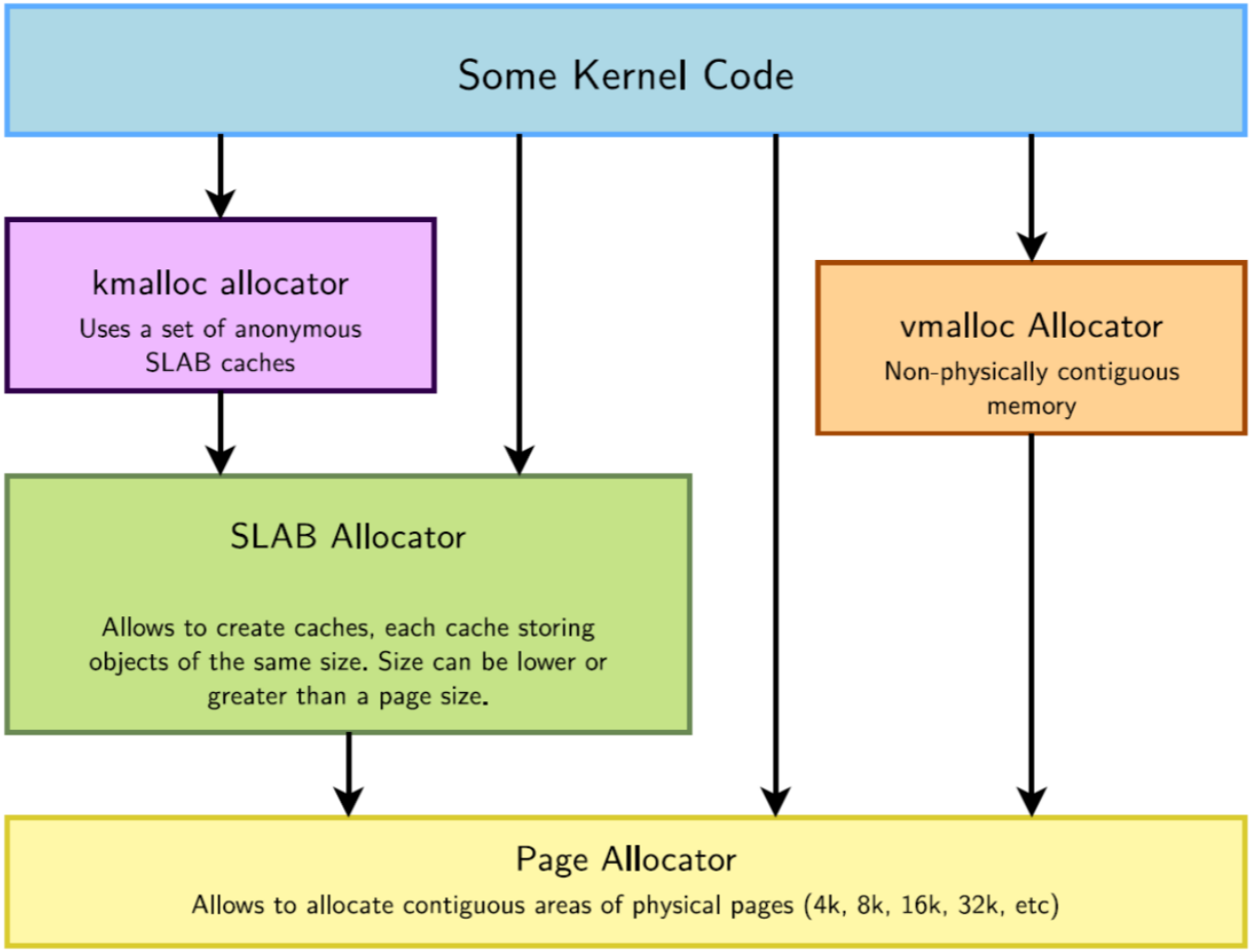
- order must match value used when allocated!
- These functions do not check for you!

Getting big chunks of memory

- Virtual memory allocation (request size in bytes, but allocates pages)
  - all functions return virtual addresses
  - but all other functions discussed today allocate physically contiguous regions
  - what if we don't care (or need a bigger region)?
  - use vmalloc (see linux/vmalloc.h)

```
void* vmalloc (unsigned long size);
void vfree (void* addr);
```

Kernel Memory Allocation Overview

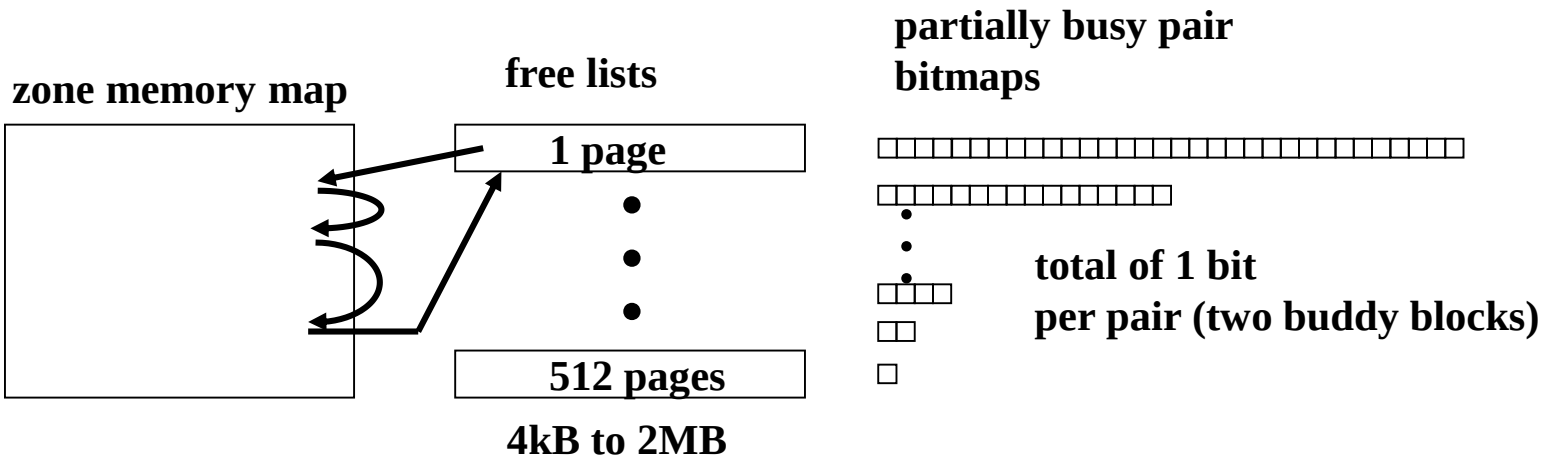


[Merlin Koglin, Memory Subsystem in the Linux Kernel]

The Buddy System

- Problem: how to implement memory allocation inside the kernel
  - need page alignment for allocations
  - may need contiguous regions of physical memory
  - need flexible allocation granularity
  - want to avoid always rewriting page tables

The Buddy System

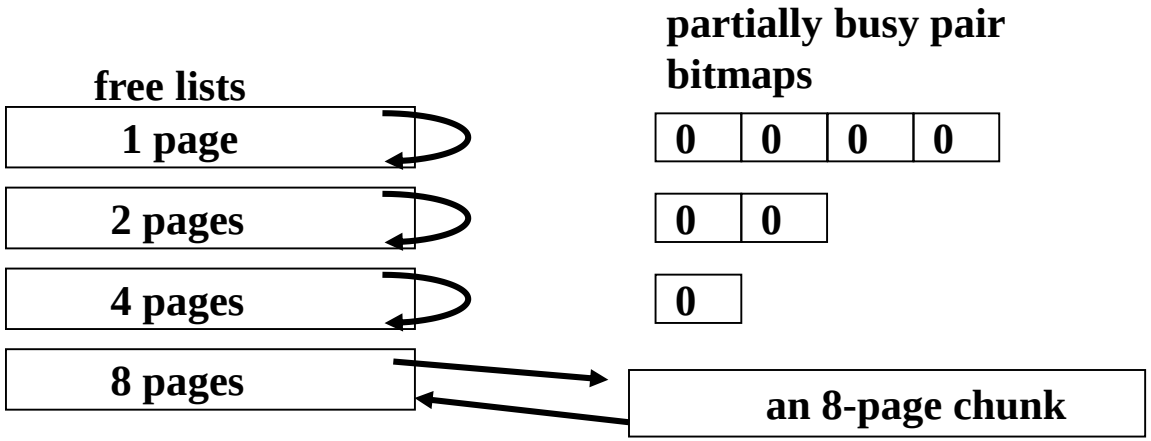


The Buddy System

- Traditional simple answer
  - exponential bins: 1 page, 2 pages, 4 pages, etc.
  - buddy system extends with dynamic motion between bins
- Partially busy bit: 1 if exactly one buddy in use (0 if both/neither in use)
- Example using one group of eight pages
  - view also as two groups of four, four groups of two, or eight single pages
  - initially appears as a single chunk in 8-page free list
  - all other free lists are empty
  - all partially free bits are 0

Buddy system example

- Initial configuration

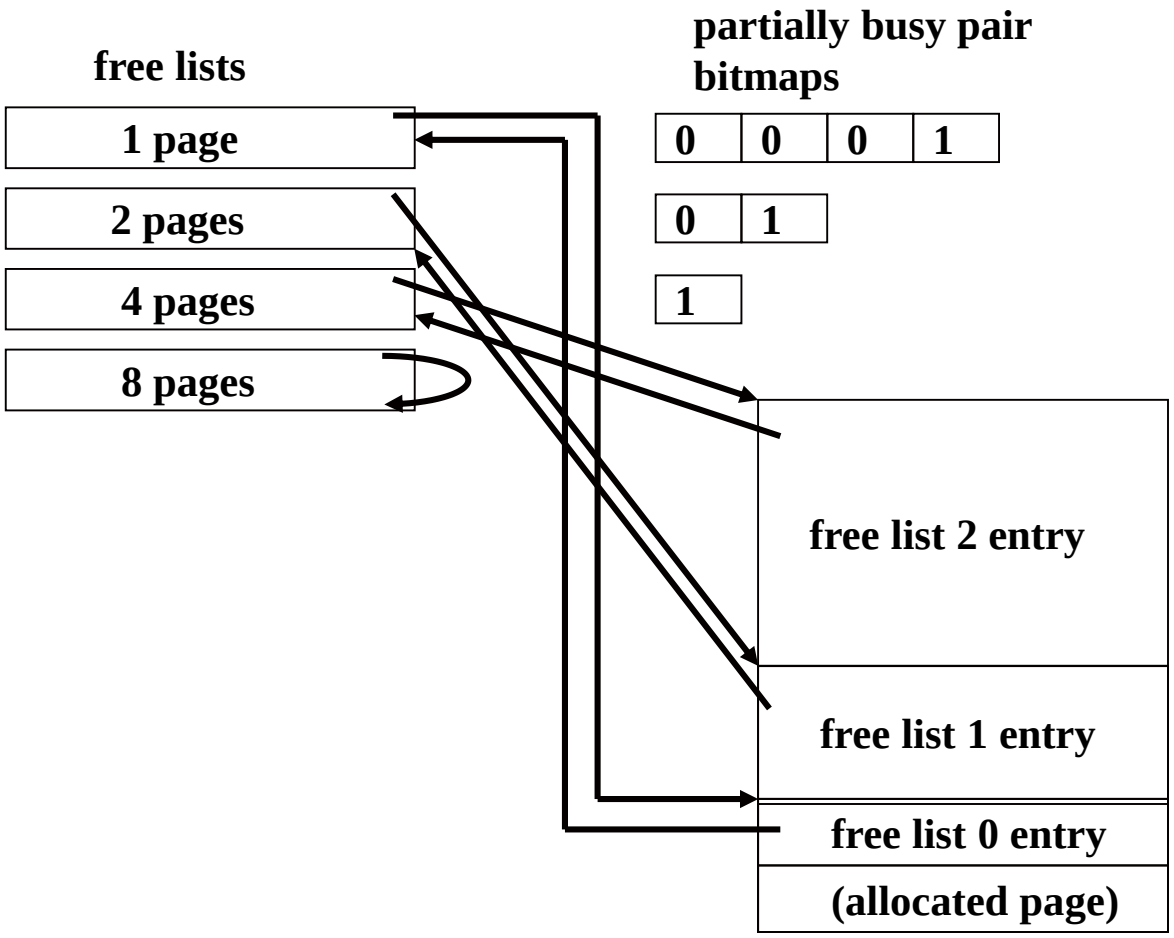


- Allocation
  - try the correct size free list
  - if empty, try the next larger size and break up a chunk

Buddy system example

- Request one page of order 0
  (order is log of # of pages, i.e., one page here)
  - any in free list 0 (1 page)? no…
  - any in free list 1 (2 pages)? no…
  - any in free list 2 (4 pages)? no…
  - any in free list 3 (8 pages)? yes! split it up recursively…

Buddy system example

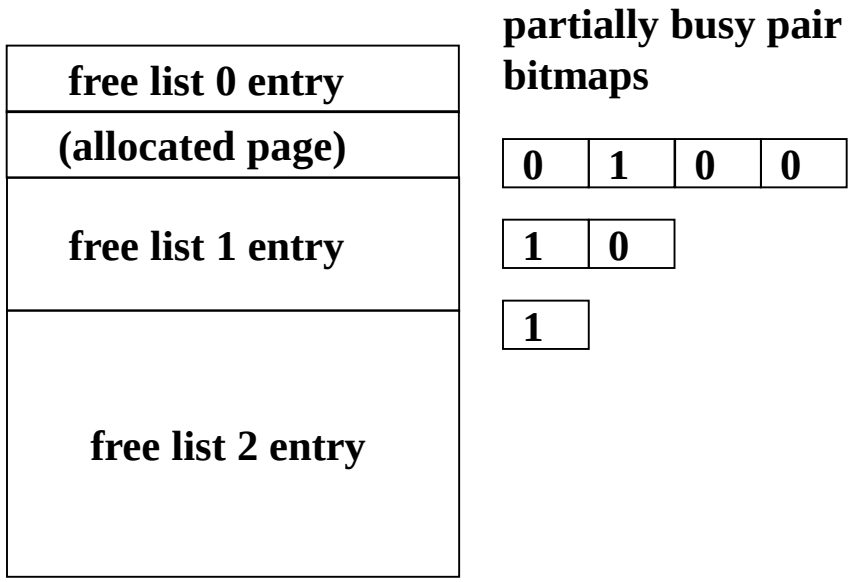


Coalescence in Buddy System

- When a block is freed, see if we can merge buddies
- Two blocks are buddies if:
  - They are the same size, *b*
  - They are contiguous
  - The address of the first page of the lower # block is a multiple of *2b × page_size*
- If two blocks are buddies, they are merged
- Repeat the process.

Buddy System Deallocation

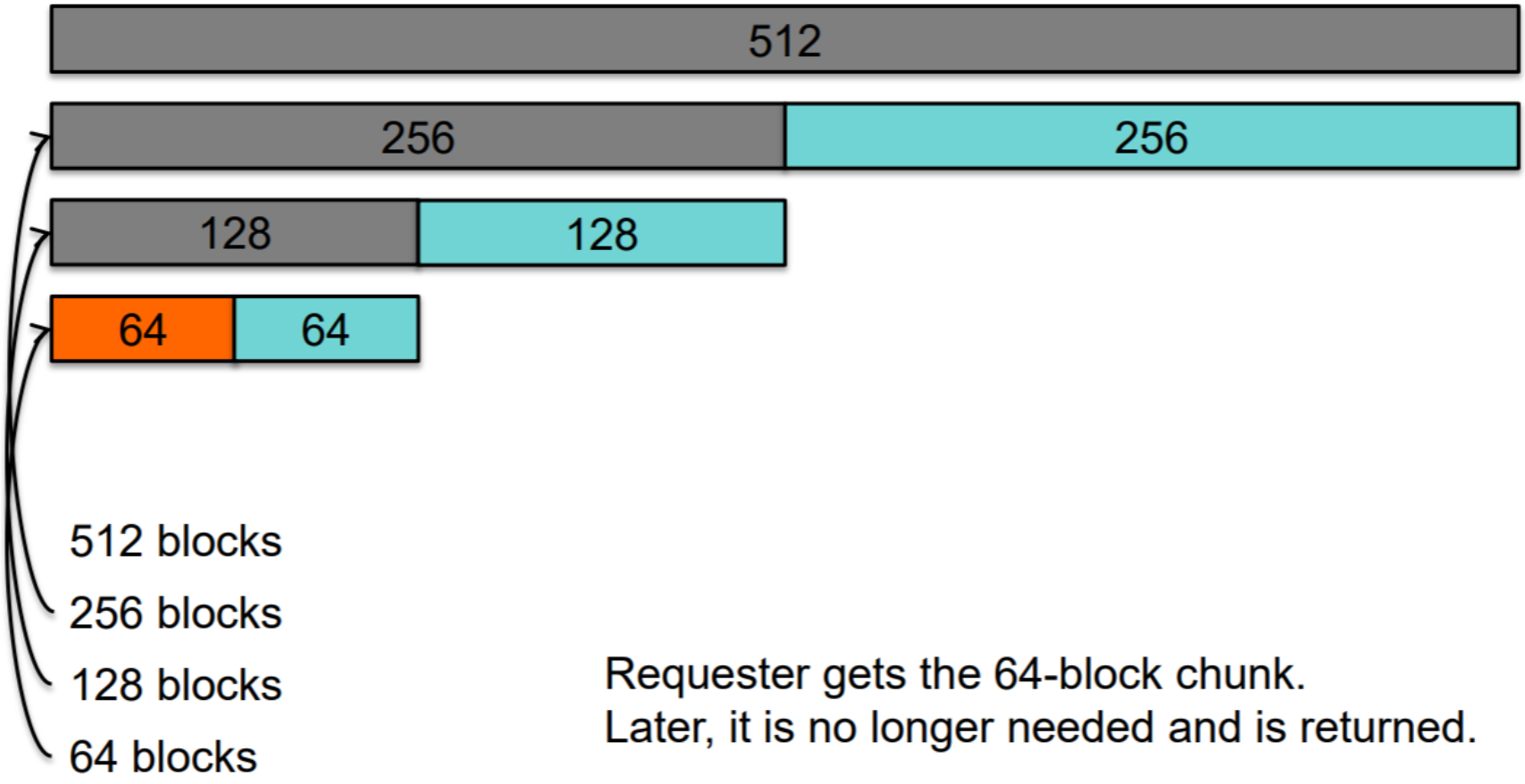
- Deallocation
  - check if buddy is free (is pair bit = 1?)
  - if both free, merge and check again (recursively)
- initial configuration for free example



Buddy System Deallocation

- Free element (order is 0)
  - check (& flip) buddy bit in order 0
  - buddy was free → remove buddy from free list and merge (address remains the same for now)
  - check (& flip) buddy bit in order 1
  - buddy was free → remove buddy from free list and merge (address changes to start of first 4-page block)
  - repeat for order 2, then done; end with initial (all free) configuration

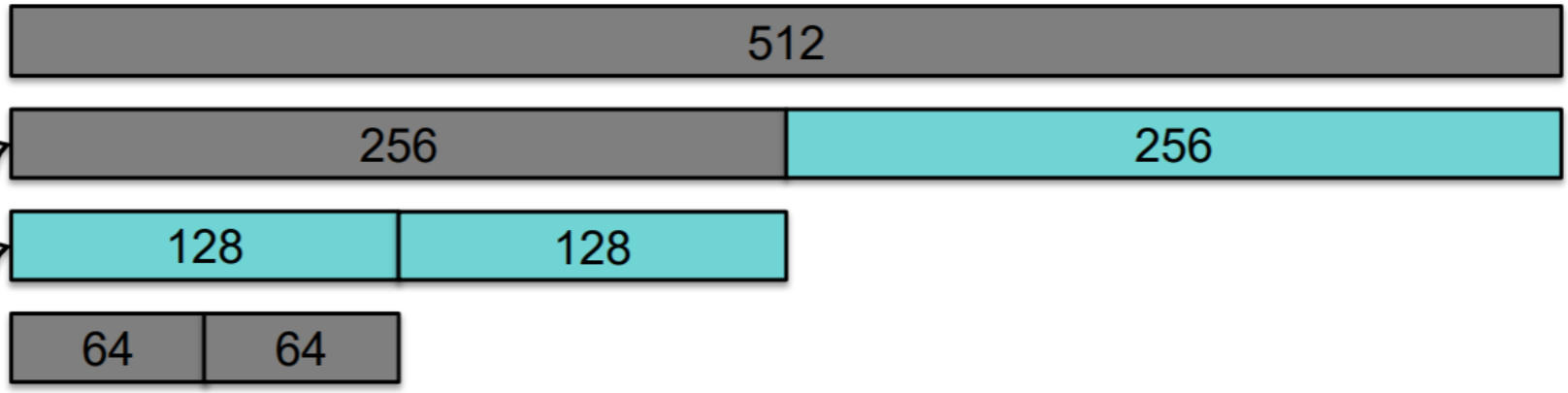
Buddy System Deallocation Example



Requester gets the 64-block chunk.
Later, it is no longer needed and is returned.

[Krzyzanowski et al. Rutgers University]

Buddy System Deallocation Example



512 blocks

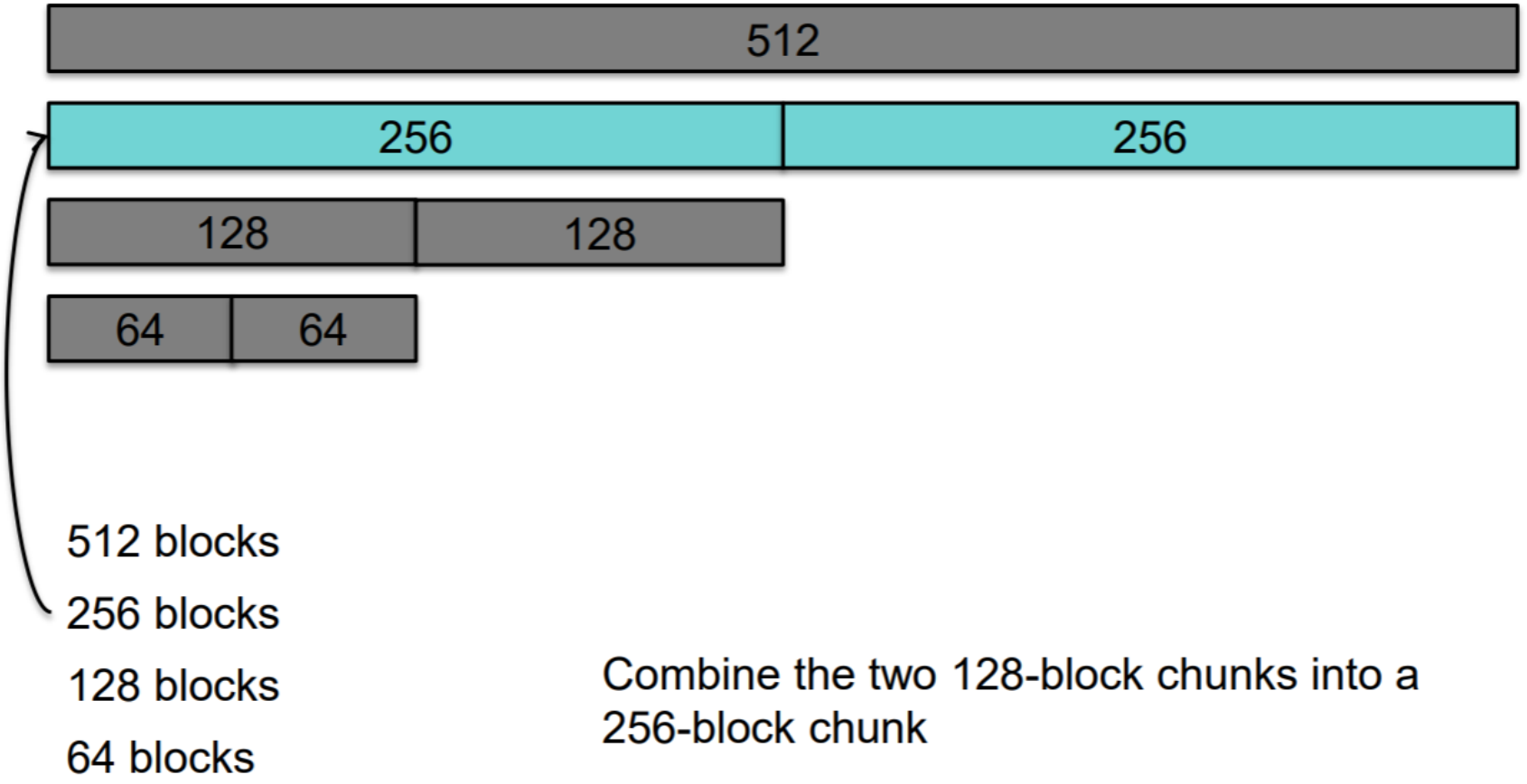
256 blocks

128 blocks

64 blocks

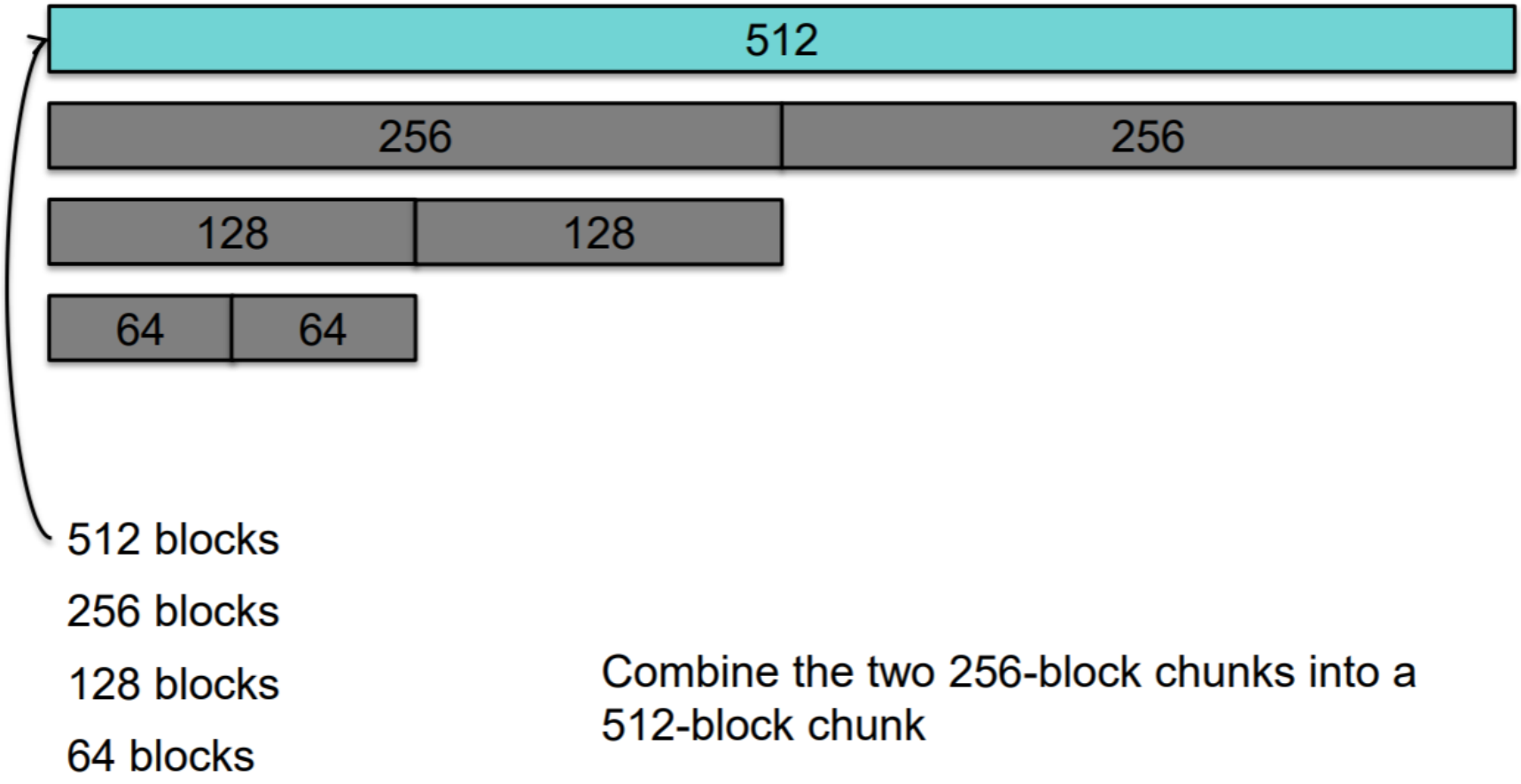
Combine (coalesce) the two 64-block chunks into a128-block chunk

Buddy System Deallocation Example



[Krzyzanowski et al. Rutgers University]

Buddy System Deallocation Example



[Krzyzanowski et al. Rutgers University]

Memory Zones

- Location of zone memory map in 2.6.22.5
- One structure to hold all information about contiguous physical pages
  - **`struct pglist_data contig_page_data;`**

- Inside of the page list structure, an array of zone structures
  - hold information about each zone (type) of memory
  - ZONE_DMA and ZONE_NORMAL for class' kernel
  - **`struct zone node_zones[MAX_NR_ZONES];`**

Memory Zones

- Inside each zone structure, an array of free area structures
  - these are the exponential bins
  - indexed by page order (0 to 10; 4kB to 4MB)
  - **`struct free_area free_area[MAX_ORDER];`**

- Inside each free area structure
  - a doubly-linked list of chunks of memory (**`free_list.prev/next`**)
  - a count of chunks (**`nr_free`**)