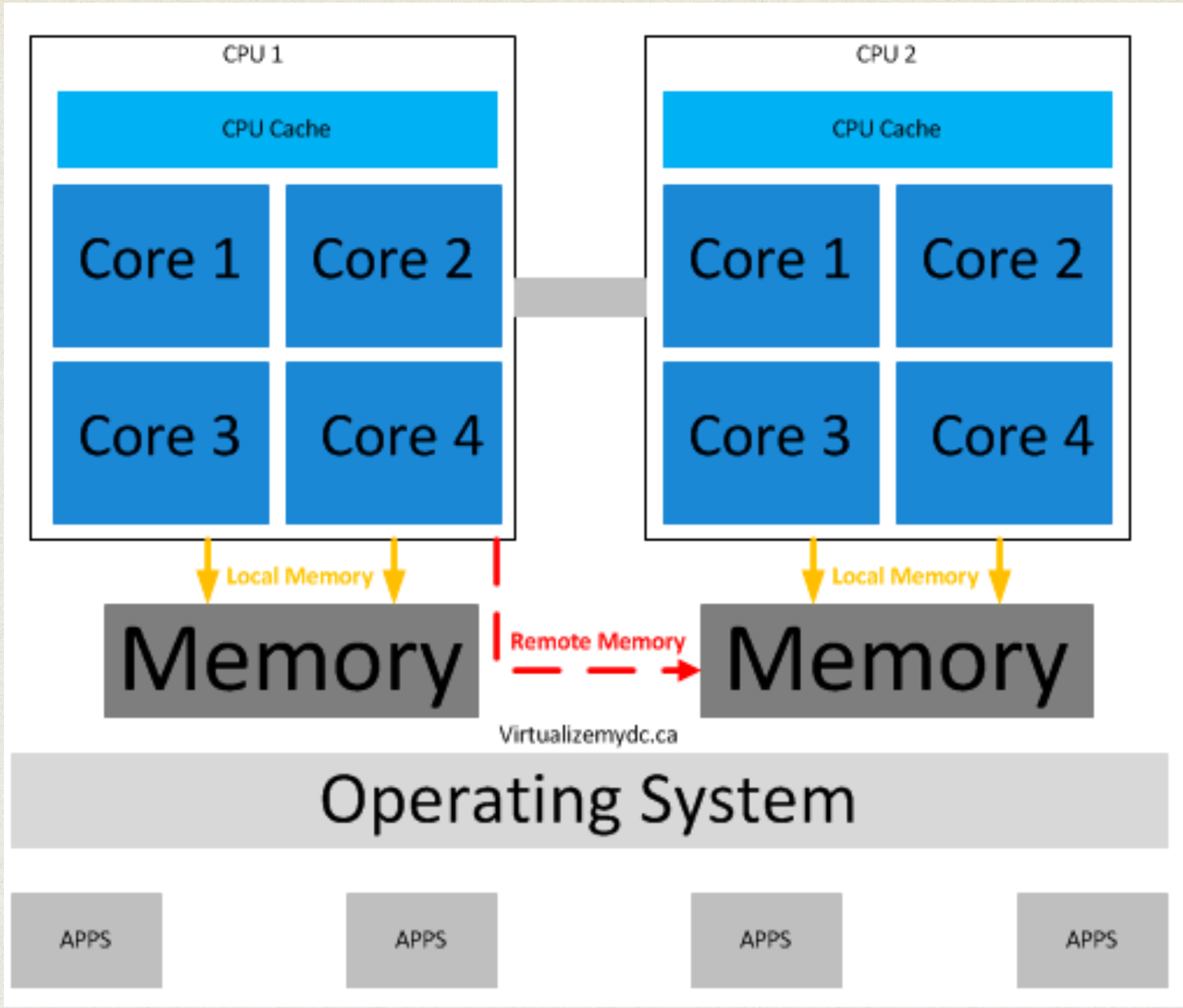
MEMORY MANAGEMENT WALK-THROUGH

Anirudh Jayakumar
March 4, 2016

TODAY'S SESSION

- Memory Management

NUMA ARCHITECTURE



NODES

- Memory arranged into banks that incur different cost of access
- Bank assigned to CPU, device cards for DMA
- Each bank is called a node
- Linux allocates a page from the node closest to the running CPU – node-local allocation
- Described by struct pglist_data – linux/mmzone.h

ZONES

- Each node is divided into a number of blocks called zones - struct zone_struct (linux/mmzone.h)

- Each zone has a type: ZONE_DMA, ZONE_NORMAL and ZONE_HIGHMEM

- ZONE_DMA - 0 -16 MB of RAM - legacy devices allocate to perform direct memory operations.

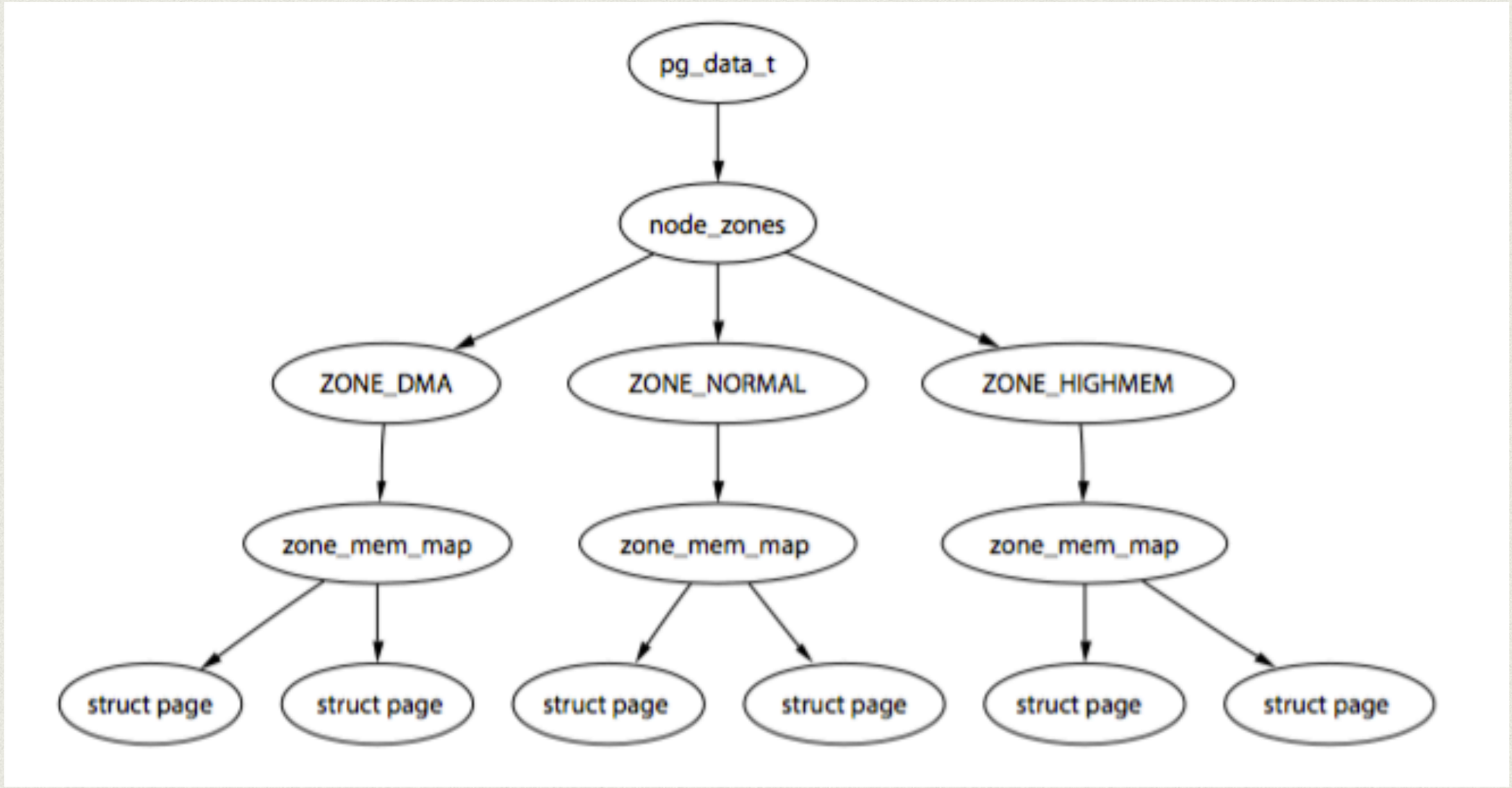
- ZONE_NORMAL - 16 MB -1 GB and is used by the kernel for internal data structures as well as other system and user space allocations

- ZONE_HIGHMEM - > 1 GB and is used exclusively for system allocations (file system buffers, user space allocations, etc).

PAGES

- Physical pages as basic unit of memory management
  - Processor's smallest addressable unit is a word
  - MMU typically deals in pages
    - The hardware that manages memory and performs virtual to physical address translations
- 32-bit arch 4KB pages, 64-bit arch 8KB pages
  - 1GB physical memory on a 32-bit machine, 262,114 distinct pages

Relationship Between Nodes, Zones and Pages



WHY PAGE STRUCT?

- uses this structure to keep track of all the pages in the system
- the kernel needs to know whether a page is free
- If a page is not free, the kernel needs to know who owns the page
- Possible owners include user-space processes, dynamically allocated kernel data, static kernel code, the page cache, and so on.

PAGE STRUCTURE

- `struct page`, the kernel representation of a physical page
  - Defined in `<linux/mm.h>`

```
struct page {
        page_flags_t            flags;
        atomic_t                _count;
        atomic_t                _mapcount;
        unsigned long           private;
        struct address_space    *mapping;
        pgoff_t                 index;
        struct list_head        lru;
        void                    *virtual;
};
```

PAGE STRUCTURE

```
struct page {
        page_flags_t            flags;
        atomic_t                _count;
        atomic_t                _mapcount;
        unsigned long           private;
        struct address_space    *mapping;
        pgoff_t                 index;
        struct list_head        lru;
        void                    *virtual;
};
```

- `flags`: status of the page (e.g. dirty or locked)
  - bit flag, at least 32 simultaneous flags
  - flag values are defined in `<linux/page-flags.h>`

PAGE STRUCTURE

```
struct page {
        page_flags_t            flags;
        atomic_t                _count;
        atomic_t                _mapcount;
        unsigned long           private;
        struct address_space    *mapping;
        pgoff_t                 index;
        struct list_head        lru;
        void                    *virtual;
    };
```

- `_count`: usage count of the page
  - "How many references to this page"
  - Check this field by `page_count()`
    - Zero means the page is free, positive integer means it is in use.

PAGE STRUCTURE

```
struct page {
        page_flags_t            flags;
        atomic_t                _count;
        atomic_t                _mapcount;
        unsigned long           private;
        struct address_space    *mapping;
        pgoff_t                 index;
        struct list_head        lru;
        void                    *virtual;
};
```

- `virtual`: the page's virtual address
- **`page` struct is associated with physical pages, not virtual pages**
  - Transient, because of swapping
  - Its goal is to describe physical memory, not the data contained therein.

PAGE STRUCTURE COST ANALYSIS

- "What a lot of memory used!"
  - Assume struct page consumes 40 bytes of memory
  - the system has 4KB physical pages,
  - the system has 128MB physical memory
  - All the page structures in the system consume slightly more than 1MB of memory
    - *Not too high a cost for managing all the physical pages*

GETTING PAGES

- All interfaces allocate memory with page-sized granularity are declared in `<linux/gfp.h>`
- `struct page * alloc_pages(unsigned int gfp_mask, unsigned int order)`
  - Allocates 2^order contiguous physical pages and returns a pointer to the first page's page structure; on error it returns `NULL`.
- `unsigned long __get_free_pages(unsigned int gfp_mask, unsigned int order)`
- `struct page * alloc_page(unsigned int gfp_mask)`
  - Same as `alloc_pages` with order 0
- `unsigned long __get_free_page(unsigned int gfp_mask)`
- `unsigned long get_zeroed_page(unsigned int gfp_mask)`

FREEING PAGES

- `void __free_pages(struct page *page, unsigned int order)`
- `void free_pages(unsigned long addr, unsigned int order)`
- `void free_page(unsigned long addr)`
- You must be careful to free only pages you allocate.
  - Passing the wrong struct page or address, or the incorrest order, can result in corruption.

EXAMPLE

```
unsigned long page;

page = __get_free_pages(GFP_KERNEL, 3);
if (!page) {
        /* insufficient memory: you must handle this error! */
        return ENOMEM;
}

/* 'page' is now the address of the first of eight contiguous
pages ... */

free_pages(page, 3);

/*
 * our pages are now freed and we should no
 * longer access the address stored in 'page'
 */
```

kmalloc()

- Very similar to user-space's `malloc()` routine
- Interface for obtaining kernel memory in byte-sized chunks
  - If you need whole pages, use the previously discussed interfaces
  - For most kernel allocations, `kmalloc()` is the preferred interface.
- Declared in `<linux/slab.h>`
  - `void * kmalloc(size_t size, int flags)`
    - Returns the pointer to physically contiguous memory of length size bytes; `NULL` on error.
    - You must check `NULL` after all calls to `kmalloc()`, and handle the error appropriately.

kfree()

- The other end of `kmalloc()` is `kfree()`, declared in `<linux/slab.h>`
  - `void kfree(const void *ptr)`

EXAMPLE

```
struct dog *ptr;

ptr = kmalloc(sizeof(struct dog), GFP_KERNEL);
if (!ptr)
        /* handle error ... */

/* ... */

/* Later, when you no longer need the memory*/
kfree(ptr);
```

GFP_MASK FLAGS

- The flags are broken up into three categories:
  - **Action modifier**
    - Specify *how* the kernel is supposed to allocate the requested memory.
    - For example, interrupt handlers must instruct the kernel not to sleep in the course of allocating memory
  - **Zone modifier**
    - Specify from *where* to allocate memory.
  - **Type**
    - Specify a combination of action and zone modifiers as needed by a certain *type* of memory allocation.
    - The `GFP_KERNEL` is a type flag, which is used for code in process context inside the kernel.

ACTION MODIFIERS

Flag	Description
`__GFP_WAIT`	The allocator can sleep.
`__GFP_HIGH`	The allocator can access emergency pools.
`__GFP_IO`	The allocator can start disk I/O.
`__GFP_FS`	The allocator can start filesystem I/O.
`__GFP_COLD`	The allocator should use cache cold pages.
`__GFP_NOWARN`	The allocator will not print failure warnings.
`__GFP_REPEAT`	The allocator will repeat the allocation if it fails, but the allocation can potentially fail.
`__GFP_NOFAIL`	The allocator will indefinitely repeat the allocation. The allocation cannot fail.
`__GFP_NORETRY`	The allocator will never retry if the allocation fails.
`__GFP_NO_GROW`	Used internally by the slab layer.
`__GFP_COMP`	Add compound page metadata. Used internally by the hugetlb code.

ZONE MODIFIERS

Flag	Description
`__GFP_DMA`	Allocate only from `ZONE_DMA`
`__GFP_HIGHMEM`	Allocate from `ZONE_HIGHMEM` or `ZONE_NORMAL`

TYPE FLAGS

Flag	Description
`GFP_ATOMIC`	The allocation is high priority and must not sleep. This is the flag to use in interrupt handlers, in bottom halves, while holding a spinlock, and in other situations where you cannot sleep.
`GFP_NOIO`	This allocation can block, but must not initiate disk I/O. This is the flag to use in block I/O code when you cannot cause more disk I/O, which might lead to some unpleasant recursion.
`GFP_NOFS`	This allocation can block and can initiate disk I/O, if it must, but will not initiate a filesystem operation. This is the flag to use in filesystem code when you cannot start another filesystem operation.
`GFP_KERNEL`	This is a normal allocation and might block. This is the flag to use in process context code when it is safe to sleep. The kernel will do whatever it has to in order to obtain the memory requested by the caller. This flag should be your first choice.
`GFP_USER`	This is a normal allocation and might block. This flag is used to allocate memory for user-space processes.
`GFP_HIGHUSER`	This is an allocation from `ZONE_HIGHMEM` and might block. This flag is used to allocate memory for user-space processes.
`GFP_DMA`	This is an allocation from `ZOME_DMA`. Device drivers that need DMA-able memory use this flag, usually in combination with one of the above.

`vmalloc()`

- Similar as `kmalloc()`, except it allocates memory that is only virtually contiguous
  - This is how `malloc()` works; The pages returned are contiguous within the virtual address space of the processor, but there is no guarantee that they are actually contiguous in physical RAM.
- `vmalloc()` is declared in `<linux/vmalloc.h>`
  - `void * vmalloc(unsigned long size)`
  - `void vfree(void *addr)`

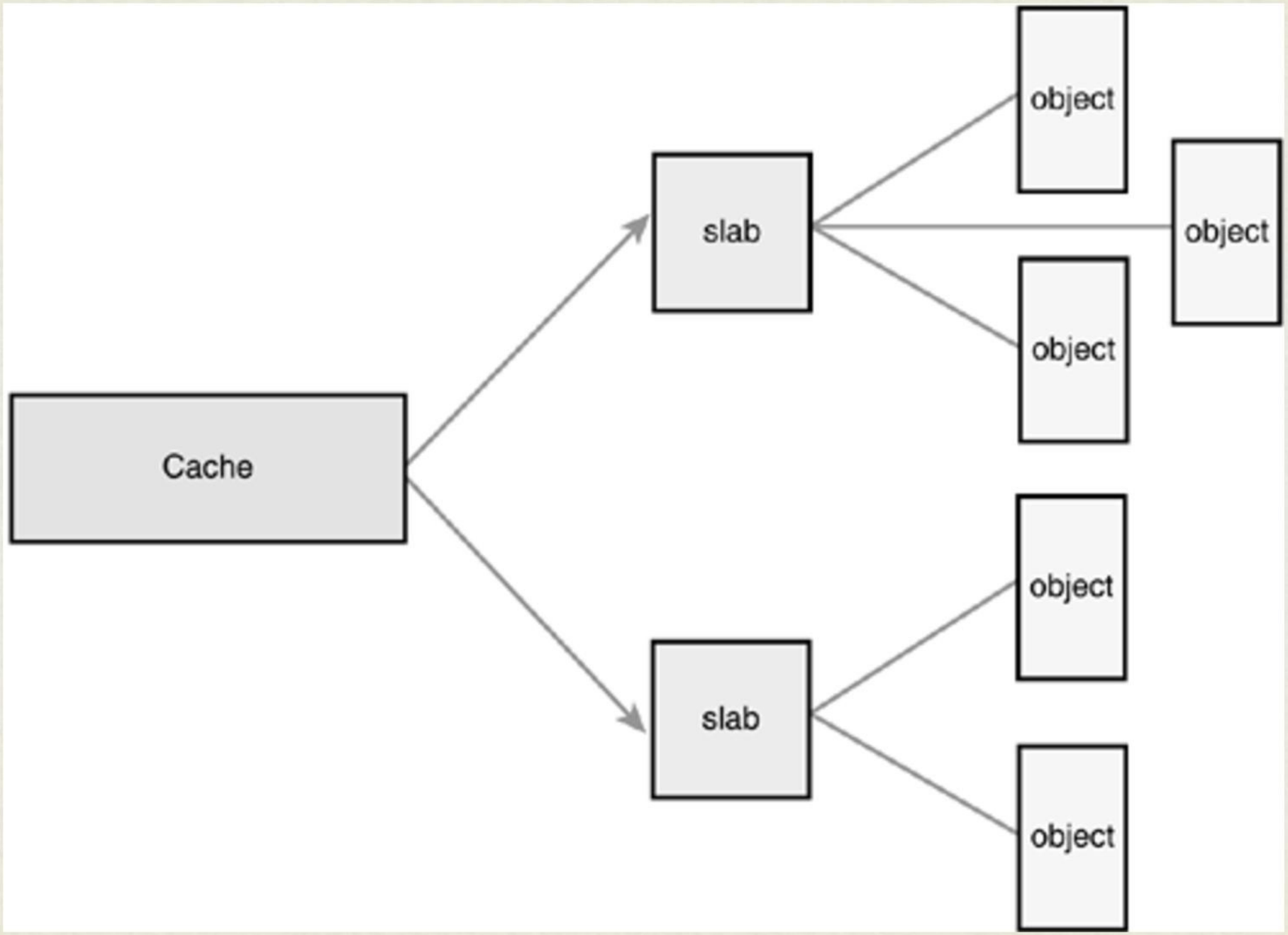
SLAB LAYER

- *"free lists"* -- no global control!
- Slab layer acts as a generic data structure-caching layer.
  - Divides different objects into groups called **caches**.
    - One cache per object type
    - For example, for process descriptors, a free list of `task_struct` structures
- Caches are divided into **slabs**, composed of one or more physically contiguous pages.

SLAB

- Each slab contains some number of **objects**, cached data structures.
- Each slab is in one of three states: *full*, *partial*, or *empty*.
- When the kernel requests a new object,
  - the request is satisfied from a partial slab, if one exists
  - otherwise, it is satisfied from an empty slab, if one exists
  - otherwise, one empty slab is created
- The above strategy reduces fragmentation

RELATIONSHIP BETWEEN CACHES, SLABS, AND OBJECTS



SLAB DESCRIPTOR

```
struct slab {
        struct list_head  list;         /* full, partial, or empty list */
        unsigned long     colouroff;    /* offset for the slab coloring */
        void              *s_mem;       /* first object in the slab */
        unsigned int      inuse;        /* allocated objects in the slab */
        kmem_bufctl_t     free;         /* first free object, if any */
};
```

SLAB ALLOCATOR

- Create a new cache:

```
kmem_cache_t * kmem_cache_create(const char *name, size_t size,
                    size_t align, unsigned long flags,
                    void (*ctor)(void*, kmem_cache_t *, unsigned long),
                    void (*dtor)(void*, kmem_cache_t *, unsigned long))
```

- Destroy a cache:

```
int kmem_cache_destroy(kmem_cache_t *cachep)
```

- Obtain an object from a cache:

```
void * kmem_cache_alloc(kmem_cache_t *cachep, int flags)
```

- Free an object and return it to its originating slab:

```
void kmem_cache_free(kmem_cache_t *cachep, void *objp)
```

EXAMPLE

- Real-life example uses the `task_struct` structure, adapted from `kernel/fork.c`
- First, the kernel has a global variable storing a pointer to the `task_struct` cache:
  - `kmem_cache_t *task_struct_cachep;`
- During kernel initialization, in `fork_init()`, the cache is created:

```
task_struct_cachep = kmem_cache_create("task_struct",
                                       sizeof(struct task_struct),
                                       ARCH_MIN_TASKALIGN,
                                       SLAB_PANIC,
                                       NULL,
                                       NULL);
```

EXAMPLE

- Each time a process calls `fork()`, a new process descriptor must be created (recall our last walk-through lecture)
  - This is done in `dup_task_struct()`, which is called from `do_fork()`:

```
struct task_struct *tsk;

tsk = kmem_cache_alloc(task_struct_cachep, GFP_KERNEL);
if (!tsk)
        return NULL;
```

EXAMPLE

- After a task dies, if it has no children waiting on it, its process descriptor is freed and returned to the `task_struct_cachep` slab cache.
- This is done in `free_task_struct()`
- `kmem_cache_free(task_struct_cachep, tsk);`

EXAMPLE

- The `task_struct_cachep` cache is never destroyed!
- If it were, you would destroy the cache via:

```
int err;

err = kmem_cache_destroy(task_struct_cachep);
if (err)
        /* error destroying cache */
```

/proc/sys/vm

Proc File	Description
admin_reserve_kbytes	The amount of free memory in the system that should be reserved for users with the capability cap_sys_admin.
dirty_background_bytes	Contains the amount of dirty memory at which the background kernel flusher threads will start writeback.
dirty_bytes	Contains the amount of dirty memory at which a process generating disk writes will itself start writeback.
drop_caches	Writing to this will cause the kernel to drop clean caches, as well as reclaimable slab objects. 1 - To free pagecache , 2 - To free reclaimable slab objects
stat_interval	The time interval between which vm statistics are updated. The default is 1 second.
swappiness	This control is used to define how aggressive the kernel will swap memory pages. Higher values will increase agressiveness
vfs_cache_pressure	This percentage value controls the tendency of the kernel to reclaim the memory which is used for caching of directory and inode objects.

REFERENCES

- http://ptgmedia.pearsoncmg.com/images/0131453483/downloads/gorman_book.pdf

- http://www.makelinux.net/books/lkd2/