Operating Systems II

Memory Management

Why Memory Management?

- Force tasks to communicate over fixed interfaces.
- Security and protection of tasks from one another (and kernel).
- Trick applications to provide more memory than available.
- When sharing memory, provides synchronization amongst operations.

MM Responsibilities

- Allocate blocks of the system’s primary (or executable) memory on request
- Ensure exclusive control of allocated blocks
- Provide a means for cooperating processes to share blocks
- Automatically move data between the primary and secondary memories

Demand Paged Virtual Memory

- Architecture-Independent Memory model.
  - This model is adapted for specific implementation.
- Each process is allocated it’s own virtual address space.
  - Processes reference virtual addresses.
  - System maps each reference into a physical address.
- The kernel *and* hardware together ensure that the virtual and physical addresses are correctly corresponding.

Page by Page

- All memory management in the kernel is done on the basis of a page.
  - For the i386, a page is typically 4K (4096 bytes).
- The virtual addressing space is 32 bits wide
  - 4 Gb of virtual address space for a task.
  - Up to 2^{20} pages per task.

Virtual Address Space

- A virtual address space is broken into two sections.
  - User segment (3 Gb) to contain the applications code and data. Addressable by the user.
    - Unmapped virtual addresses are simply not used.
  - Kernel segment (1 Gb) permanently mapped and associated with fixed physical memory addresses used by the kernel.
- System calls execute in kernel segment (mode).

Virtual Address Space

- Each of these segments are further broken down into sectors.
  - *Code* and *Data* sectors.
- The kernel segment (physical pages) are mapped after the user segment.
  - The boundary is described by `TASK_SIZE` macro.

Kernel to/from User

- From the kernel, there are several macros defined to allow access to the user segment of a processes virtual address space...
  - get_user(val, ptr) /* read scalar value */
  - put_user(val, ptr) /* write scalar value */
  - copy_from_user(to, from, n)
  - copy_to_user(to, from, n)
  - strncpy_from_user(to, from, n)
  - strnlen_user(str, n) /* length of string (max n) */
  - clear_user(mem, len)

access_ok() ?

- The access_ok() macro allows the kernel to query a virtual address to see if it is legitimate for the task.
- Each of the kernel -> user memory access macros have a __ equivalent (e.g. __get_access(val, ptr)) that performs the action without the address check.
- WHY?

Let's Get Physical!

- Converting addresses from virtual to physical is a three level process in an architecture independent model.
  - It can be less in a *real* implementation depending on what the hardware supports.
  - This is dictated by the hardware's MMU...
  - For the x86 it only supports a two level conversion of the address.

Virtual to Physical

Page Directories

- Kernel has to manage page directories (and tables) for both the user and kernel segments of the processes.
- Short identifiers
  - pgd - page directory, pmd - page middle directory
- The data types are `pgd_t` and `pmd_t` respectively.
  - Defined as structs to avoid mistaken casts to int.

Page Directory Support

- `pgd_val(), pmd_val()`
  - Allow access to the real value of the directory entry.
- `pgd_alloc(), pmd_alloc()`
  - Provide a memory page for the respective directory.
- `pgd_free(), pmd_free()`
  - The directory entries are freed

Page Directory Support

- `pgd_clear(), pmd_clear()`
  - Deletes the entry in the page directory.
- `pgd_none(), pmd_none()`
  - Test whether a directory entry is valid
- `pgd_present(), pmd_present()`
  - Negation result of the p{gm}d_none() macro
- `pgd_bad(), pmd_bad()`
  - Check the correctness of the entries that refer to subsequent page structs (directory, mid directory, table)

Page Directory Support

- `pgd_page(), pmd_page()`
  - Return reference to next level page structure
- `pgd_offset()`
  - Return reference to page directory for the given address.
- `pmd_offset()`
  - Return the reference to page middle directory for the given address.
- `set_pgd(), set_pmd()`
  - Fill the page directory with entries.

The Page Table

- The Page table is the lowest level of the memory model.
- An entry in the page table is defined as `pte_t` structure.
- This is the final mapping, so it addresses a page in the physical memory.

Page Table Support

- `pte_val()`
  - Returns the value of a page table entry.
- `pte_alloc()`
  - Using an entry of the pmd, returns the referenced page table. Creates one if non-existant.
- `pte_free()`
  - Free the page table and makes sure it does not contain an initialized value.

Page Table Support

- `pte_page()`
  - Returns reference to physical page
- `pte_offset()`
  - Given an intermediate page directory and virtual address, returns associated page table.
- `set_pte()`
  - Sets the given position of the page table entry.

Page Table Flags

- Flags in the page table entry indicate
  - The legal access modes into the page.
  - The pages status.
- A page’s status can give vital information for how memory management is performed.

Page Table Flags

- PAGE_NONE – No physical memory page associated with entry.
- PAGE_SHARED – All types of access permitted.
- PAGE_READONLY – No writing. “Copy-on-Write” can be used.
- PAGE_KERNEL – kernel segment only allowed access.
- PAGE_KERNEL_RO – kernel read-only access.

Modifier Functions

- `pte_modify()` – modify the descriptor
- `pte_present()` – tests presence attribute
- `pte_dirty()` – tests dirty attribute
- `pte_read()` – tests read attribute
- `pte_exec()` – tests exec attribute
- `pte_young()` – tests age attribute

Modifier Functions

- `pte_mkexec(), pte_exprotect()`
  - Sets execute attribute
- `pte_mkdirty(), pte_mkclean()`
  - Sets dirty attribute
- `pte_mkread(), pte_mkwrite()`
  - Sets read attribute
- `pte_mkyoung(), pte_mkold()`
  - Sets age attribute

Task’s Virtual Space

- A task’s virtual space is divided between user and kernel segments.
- Since the user segment contains data and code belonging to a task it must be unique to the task.
  - This can be done using different directories or page tables for a process.
- fork - copies the page directory, tables, and pages
- clone - shares the same memory structure.

Structure of User Segment

- This is dependent on the binary format being executed. (a.out, ELF, S19, ...)
- Contains the code of user program, data space for execution, and any shared libraries used.
- This also includes program filename, environment, arguments, ...

Virtual Memory Areas

- Shared libraries and applications are sometimes large. Even if it can fit in the addressable space, it is wasteful to load.
- Virtual memory can also provide a mechanism for sharing memory.
  - Quicker to write to a memory location than to perform a system call.
- Read Only data can be shared between tasks (Copy_on_write).
- Just because memory is requested, it may not be needed!

vm_area_struct

- vm_start, vm_end – beginning and ending of virtual memory.
- vm_page_prot – protection attributes of area.
- vm_flags – memory type.
- vm_file – file or device mapped to virtual memory.
- vm_opps – structure of function pointers (handlers for page errors).

Operational Details

- The virtual memory structs for a task can be stored in both an AVL tree and a singly linked list for management.
- If you try to access any of the addresses that are defined by this virtual memory area, then the appropriate function is called.

Operational Details (cont.)

- open
  - Memory region is added to the set of regions owned by the task.
- close
  - Memory region is removed from the set of regions owned by the task.
- nopage(area, address, write_access)
  - Obtains physical page, copies data, corrects page directory/page table. Invoked by Page Fault exception handler

do_mmap_pgoff()

- Creates a new VM area.
- Arguments include:
  - The file to map and offset into the file.
  - Address to start searching for free linear interval.
  - Length of the file to map.
  - Any permissions and memory region flags.

Kernel Segment

- On an interrupt 0x80, the processor sets registers to allow kernel memory access.
  - Likewise, resets registers on return to user task
- Memory management in the kernel is in some ways easier than in the user segment.
  - No shared libraries
  - All memory is always kept in physical pages.
  - All tasks share the same kernel segment!
- Still have to deal with dynamic memory.

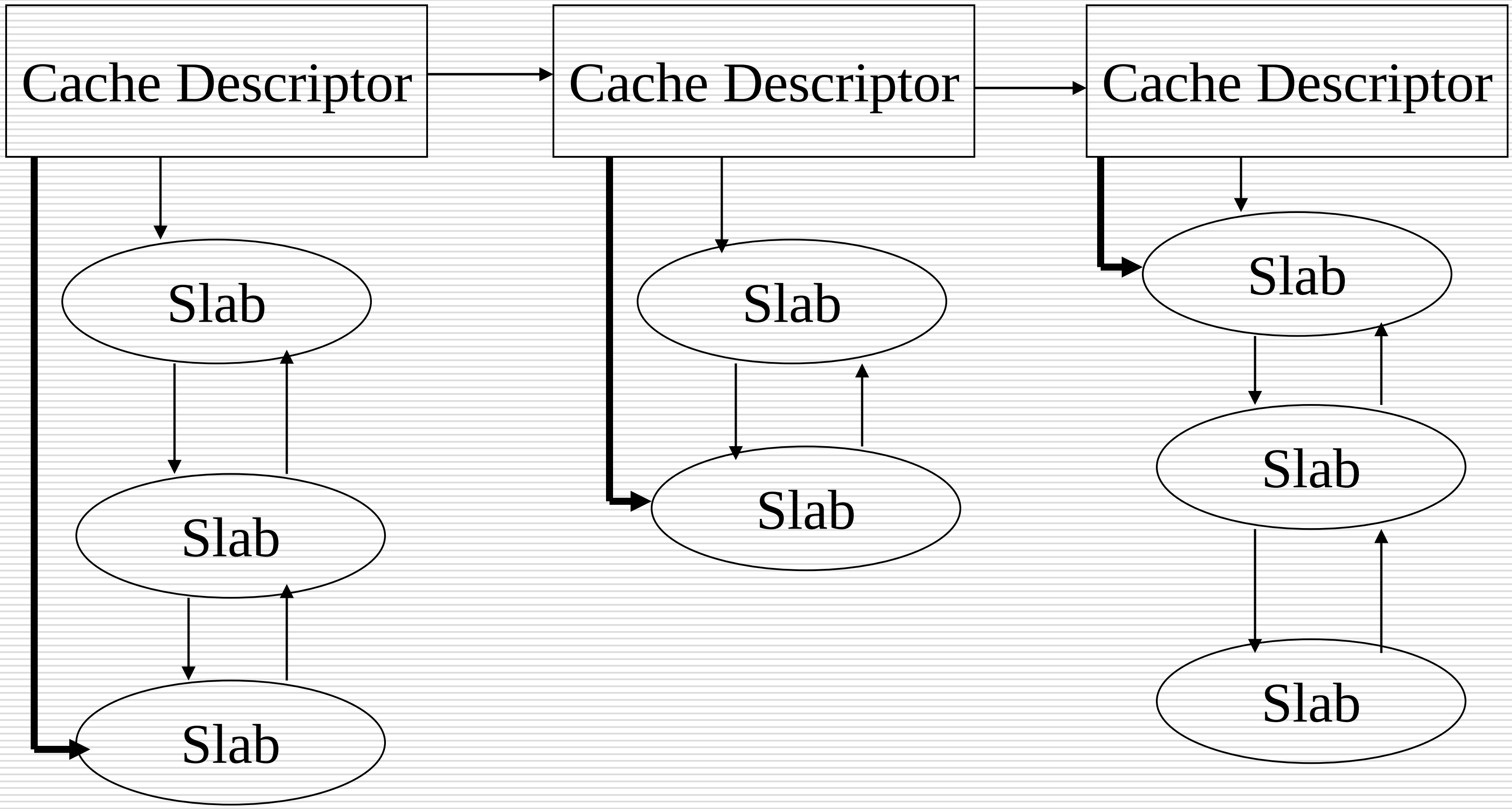
Kernel's Dynamic Memory

- Once up, the kernel can acquire memory through
  - `kmalloc(size, priority)`
  - maximum of 128K
  - Does not clear the allocated memory
- Free the memory through
  - `kfree(obj)`
- Dynamic memory in the kernel is performed on a slab basis instead of pages.

Slabs

- A Caching mechanism exists for managing different sizes of slabs.
- Each of the slabs can be a multiple of a memory page in size.
  - Starting from 64 bytes up to 128K
- Depending on the memory space, the slab management structure can be inside or outside the slab.

Slab Caches


Cache Descriptor
Cache Descriptor
Cache Descriptor
Slab
Slab
Slab
Slab
Slab
Slab
Slab
Slab

Slab Allocation

- `kmem_cache_create(name, size, offset, flags, `**`ctor()`**`, `**`dtor()`**`)`
  - `name` – name of the cache
  - `size` – creates a cache for objects of this size.
  - `offset` – any special alignment needed.
  - `ctor()` and `dtor()` – construct and destruct functions to call on (de)allocating objects.

Slab Allocation (cont.)

- `kmem_cache_destroy()`
  - Closes the cache and releases all associated memory.
- `kmem_cache_shrink()`
  - Shrink the size of a cache.
- `kmem_cache_alloc()`
  - Allocate an object from the cache.
- `kmem_cache_free()`
  - Free an object back into the cache.

Paging

- Paging can occur either to a fixed length file or to a partition.
  - Page file or swap partition.
- Pages of physical memory that have been altered need writing to the swap memory.
- Pages that have not been altered can just be discarded.
- Why is kernel memory never swapped?

Partition vs. File

- For swapping, a partition is more efficient than a file.
  - Partition is in consecutive blocks, thus no searching for blocks.
  - File can be split across non-consecutive blocks resulting in longer lookups.
  - Filesystem block size may not correspond to a page size!
- Swap file is more flexible! Easier to allocate a file than a partition after initial setup.

Swap Space

- Within the kernel, `MAX_SWAPFILES` (8 is reasonable) defines the maximum number of space space entries allowed.
- To map a new swap space, the system call `sys_swapon(char* specialfile, int flags)` is invoked.
  - `specialfile` points to the swap device/file.
  - `flags` indicates the priority of swap device.
- `sys_swapoff(char *specialfile)` to remove the swap space.

swap_info table

- Each swap device added creates an entry in the swap_info table. Entries contain
  - flags – Is the swap space available?
  - swap_device – device number if swap partition.
  - sdev_lock – synchronization lock for swap space.
  - swap_file – file pointer if swap file used.
  - swap_vfsmnt – mount point of the swap space.
  - swap_map – table of swap space pages.
  - prio – priority of the swap space.
  - next – priority list of swap spaces.
  - pages - #pages that can be written to.

Swap Management

- For each page of memory in swap, a `mem_map_t` structure is used.
- Different fields include:
  - `list` - All swap pages are kept in a doubly linked circular list. One list for used, clean, and dirty pages.
  - `mapping` - Points to the address_space object which the page belongs.
  - `count` - number of users of the page in swap.

Physical Page Management

- Page frames are often 4Kb on Intel architecture.
- The kernel must keep track of the current status of each frame.
  - Is the page free?
  - process or kernel pages?
- An array of page frame descriptors is used for each page (`struct page`).

Page Frame Descriptors

- Each descriptor has several fields:
  - count - equals 0 if the frame is free, >0 otherwise.
  - flags - an array of 32 bits for the frame status.
    - Example flag values:
      - `PG_locked` - page cannot be swapped out.
      - `PG_reserved` - page reserved for kernel or unuseable.
      - `PG_slab` - included in a slab.

The mem_map Array

- All page frame descriptors are stored in the `mem_map` array.
- Descriptors are less than 64 bytes, so about 4 pages of memory are needed for each Mb of RAM.
- The `virt_to_page()` macro computes the number of the page frame whose address is passed as a parameter.
  - `#define virt_to_page(addr) (__pa(addr) >> PAGE_SHIFT)`
  - `__pa` macro converts logical address to physical.

Requesting Page Frames

- Main routine for requesting page frames:
  - `__get_free_pages(gfp_mask, order)`
  - request 2^{order} contiguous page frames.
- `gfp_mask` specifics how to look for page frames
  - `GFP_WAIT` - Allows kernel to discard page frame contents to satisfy request.
  - `GFP_IO` - Allows kernel to write pages to disk to free pages frames for new request.
  - `GFP_HIGH/MED/LOW` - Request priority.

Releasing Page Frames

- Main routine for freeing pages is: `free_pages(addr, order)`
  - Check descriptor of frame at addr.
  - If not reserved -> decrement the count field.
  - If count == 0 -> free 2^{order} contiguous frames
  - `free_pages_ok()` inserts 1st page into list of free page frames.

External Fragmentation

- External fragmentation is a problem when small blocks of free page frames are scattered between allocated page frames.
- Becomes impossible to allocate large blocks of contiguous page frames.
- Solution:
  - Use paging HW to group non-contiguous page frames into contiguous linear (virtual) addresses, or …
  - Track free blocks of contiguous frames & attempt to avoid splitting large free blocks to satisfy requests.

The Buddy System

- Free page frames are grouped into lists of blocks containing 2^n contiguous page frames.
  - Maintaining 11 lists of 1, 2, 4, ..., 1024 contiguous page frames can often service ALL memory size requests.
  - Physical address of 1st frame in a block is a multiple of the group size
    - e.g. multiple of $16*2^{12}$ for a 16 page frame block.

Buddy Allocation

- Example: Need to allocate 65 contiguous page frames.
  - Look in the list of free 128 page frame blocks.
  - If free block exists, allocate it
  - Else, if free block found in 256 page frame list, split it into 2 128-page frame blocks. Allocate one block, and put remaining block into the lower order list.
- What is the worst case fragmentation?

Buddy De-Allocation

- When blocks of page frames are released the kernel tries to merge pairs of “buddy” blocks of size b into blocks of size 2b.
- Two blocks are buddies if:
  - They have equal size b.
  - They are located at contiguous physical addresses.
  - The address of the first page frame in the first block is aligned on a multiple of $2b*2^{12}$
- The process repeats by attempting to merge buddies of size 2b, 4b, 8b etc…

Buddy Data Structures

- Different zones used for page frames.
- An array of MAX_ORDER * elements (one for each group size) of type free_area_struct.
  - free_area[order] contains a free_list and a bitmap for groups of blocks and 2^{order} page frames.
- MAX_ORDER is set to maximum required size.