Lecture Overview

- Linux memory management
  - *This part of the Linux kernel is relatively complex and is only presented in overview, the point is to familiarize yourself with the names and terminology*
  - Paging
  - Physical and logical memory layout
  - Contiguous frame management
  - Noncontiguous frame management
  - Process address space
    - Memory descriptors
    - Memory regions
    - Page faults

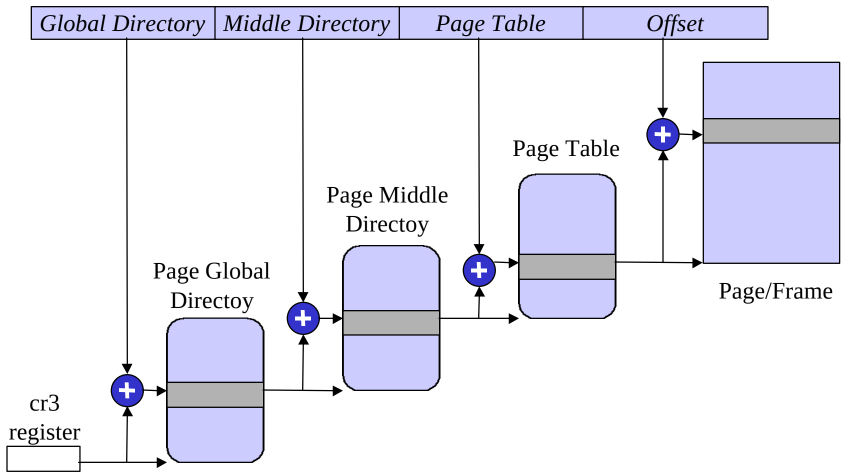
Linux Memory Management

- Intel x86 processes have segments
- Linux tries to avoid using segmentation
  - Memory management is simpler when all processes use the same segment register values
  - Using segment registers is not portable to other processors
- Linux uses paging
  - 4k page size
  - A three-level page table to handle 64-bit addresses
  - On x86 processors
    - Only a two-level page table is actually used
    - Paging is supported in hardware
    - TLB is provided as well

Linux Memory Management

View of a logical address in Linux

(For x86 processors, *Middle Directory* is 0 bits)



Linux Kernel Memory Management

- Approximately the first two megabytes of physical memory are reserved
  - For the PC architecture and for OS text and data
  - The rest is available for paging
- The logical address space of a process is divided into two parts
  - `0x00000000` to `PAGE_OFFSET-1` can be addressed in either user or kernel mode
  - `PAGE_OFFSET` to `0xffffffff` can be addressed only in kernel mode
  - `PAGE_OFFSET` is usually `0xc0000000`

Linux Page Frame Management

- The kernel keeps track of the current status of each page frame in an array of `struct page` descriptors, one for each page frame
  - Page frame descriptor array is called `mem_map`
  - Keeps track of the usage count (== 0 is free, > 0 is used)
  - Flags for dirty, locked, referenced, etc.
- The kernel allocates and release frame via
  - `__get_free_pages(gfp_mask, order)` and `free_pages(addr, order)`

Linux Page Frame Management

- In theory, paging eliminates the need for contiguous memory allocation, but…
  - Some operations like DMA ignores paging circuitry and accesses the address bus directly while transferring data
    - As an aside, some DMA can only write into certain addresses
  - Contiguous page frame allocation leaves kernel paging tables unchanged, preserving TLB and reducing effective access time
- As a result, Linux implements a mechanism for allocating contiguous page frames
  - *So how does it deal with external fragmentation?*

Contiguous Page Frame Allocation

- *Buddy system* algorithm
  - All page frames are grouped into 10 lists of blocks that contain groups of *1*, *2*, *4*, *8*, *16*, *32*, *64*, *128*, *256*, and *512* contiguous page frames, respectively
    - The address of the first page frame of a block is a multiple of the group size, for example, a 16 frame block is a multiple of 16×2^{12}
  - The algorithm for allocating, for example, a block of 128 contiguous page frames
    - First checks for a free block in the *128* list
    - If no free block, it then looks in the *256* list for a free block
    - If it finds a block, the kernel allocates 128 of the 256 page frames and puts the remaining 128 into the *128* list
    - If no block it looks at the next larger list, allocating it and dividing the block similarly
    - If no block can be allocated an error is reported

Contiguous Page Frame Allocation

- *Buddy system* algorithm
  - When a block is released, the kernel attempts to merge together pairs of free *buddy* blocks of size *b* into a single block of size *2b*
    - Two blocks are considered buddies if
      - Both have the same size
      - They are located in contiguous physical addresses
      - The physical address of the first page from of the first block is a multiple of $2b \times 2^{12}$
    - The merging is iterative

Contiguous Page Frame Allocation

- Linux makes use of two different buddy systems, one for page frames suitable for DMA (i.e., addresses less than 16MB) and then all other page frames
- Each buddy system relies on
  - The page frame descriptor array `mem_map`
  - An array of ten `free_area_struct`, one element for each group size; each `free_area_struct` contains a doubly linked circular list of blocks of the respective size
  - Ten bitmaps, one for each group size, to keep track of the blocks it allocates

Contiguous Memory Area Allocation

- The buddy algorithm is fine for dealing with relatively large memory requests, but it how does the kernel satisfy its needs for small memory areas?
  - In other words, the kernel must deal with internal fragmentation
- Linux 2.2 introduced the *slab allocator* for dealing with small memory area allocation
  - View memory areas as objects with data and methods (i.e., constructors and destructors)
  - The slab allocator does not discard objects, but caches them
  - Kernel functions tend to request objects of the same type repeatedly, such as process descriptors, file descriptors, etc.

Contiguous Memory Area Allocation

- Slab allocator
  - Groups objects into caches
  - A set of specific caches is created for kernel operations
  - Each cache is a "store" of objects of the same type (for example, a file pointer is allocated from the `filp` slab allocator)
    - Look in `/proc/slabinfo` for run-time slab statistics
  - Slab caches contain zero or more slabs, where a slab is one or more contiguous pages frames from the buddy system
  - Objects are allocated using `kmem_cache_alloc(cachep)`, where `cachep` points to the cache from which the object must be obtained
  - Objects are released using `kmem_cache_free(cachep, objp)`

Contiguous Memory Area Allocation

- Slab allocator
  - A group of general caches exist whose objects are geometrically distributed sizes ranging from 32 to 131072 bytes
  - To obtain objects from these general caches, use `kmalloc(size, flags)`
  - To release objects from these general caches, use `kfree(objp)`

Noncontiguous Memory Area Allocation

- Linux tries to avoid allocating noncontiguous memory areas, but for infrequent memory requests sometimes it makes sense to allocate noncontiguous memory areas
  - This works similarly as the lecture discussions on paging
  - Linux uses most of the reserved addresses above `PAGE_OFFSET` to map noncontiguous memory areas
  - To allocate and release noncontiguous memory, use `vmalloc(size)` and `vfree(addr)`, respectively

Linux Kernel Memory Allocation Review

- Kernel functions get dynamic memory in one of three ways
  - `__get_free_pages()` to get pages from the buddy system
  - `kmem_cache_alloc()` or `kmalloc()` to use slab allocator to get specialized or general objects
  - `vmalloc()` to get noncontiguous memory areas

- *What about processes?*

Process Address Spaces

- To the kernel, user mode requests for memory are
  - Considered non-urgent
    - Unlikely to references all of its pages
    - Allocated memory may not be accessed for a while
  - Considered untrustworthy
    - Kernel must be prepared to catch all addressing errors
- As a result, the kernel tries to defer allocation of dynamic memory to processes

Process Address Spaces

- The *address space* of a process consists of all logical addresses that the process is allowed to use
  - Each process address space is separate (unless shared)
  - The kernel allocates logical addresses to a process in intervals called *memory regions*
    - Memory regions have an initial logical address and a length, which is a multiple of 4096
- Typical situations in which a process gets new memory regions
  - Creating a new process (`fork()`), loading an entirely new program (`execve()`), memory mapping a file (`mmap()`), growing its stack, creating shared memory (`shmat()`), expanding its heap (`malloc()`)

Process Memory Descriptor

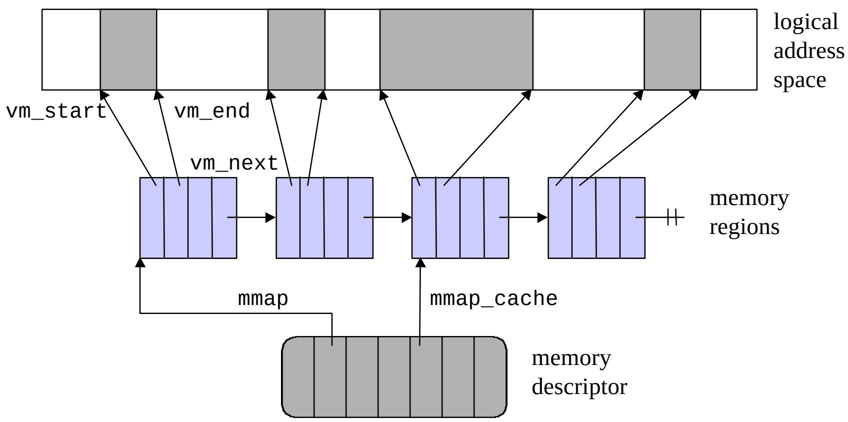
- All information related to the process address space is included in the *memory descriptor* (`mm_struct`) referenced by the `mm` field of the process descriptor
- Some examples of included information
  - A pointer to the top level of the page table, the Page Global Directory, in field `pgd`
  - Number of page frames allocated to the process in field `rss`
  - Process' address space size in pages in field `total_vm`
  - Number of locked pages in field `locked_vm`
  - Number of processes sharing the same `mm_struct`, i.e., lightweight processes
- *Memory descriptors are allocated from the slab allocator cache using `mm_alloc()`*

Process Memory Region

- Linux represents a memory region (i.e., an interval of logical address space) with `vm_area_struct`
  - Contains a reference to the memory descriptor that owns the region (`vm_mm` field), the start (`vm_start` field) and end (`vm_end` field) of the interval
  - Memory regions never overlap
  - Kernel tries to merge contiguous regions (if their access rights match)
  - All regions are maintained on a simple list (`vm_next` field) in ascending order by address
    - The head of the list and the size of the list are in the `mmap` field and the `map_count` fields, respectively, of the `mm` memory descriptor
    - If the list of regions gets large (usually greater than 32), then it is also managed as an AVL tree for efficiency

Process Memory Region

Abstract view of memory descriptor, regions, and logical address space



Process Memory Region

- To allocate a logical address interval, the kernel uses `do_mmap()`
  - Checks for errors and limits
  - Tries to find an unmapped logical address interval in memory region list
  - Allocates a `vm_area_struct` for new interval
  - Updates bookkeeping and inserts into list (merging if possible)
- To release a logical address interval, the kernel uses `do_munmap()`
  - Locates memory region that overlaps, since it may have been merged
  - Removes memory region, splitting if necessary
  - Updates bookkeeping

Page Fault Handler

- When a process requests more memory from the kernel, it only gets additional logical address space, not physical memory
- When a process tries to access its new logical address space, a page fault occurs to tell the kernel that the memory is actually needed (i.e., demand paging)
  - The page fault handler compares the logical address to the memory regions owned by the process to determine if
    - The memory access was an error
    - Physical memory needs to be allocated to the process
  - An address may also not be in physical memory if the kernel has swapped the memory out to disk

Copy on Write

- When the kernel creates a new process, it does not give it a completely new address space
  - They share the address space of their parent process
  - The kernel write protects all shared pages frames
  - Whenever either the parent or the child tries to write a shared page frame, an exception occurs
  - The kernel traps the exception and makes a copy of the frame for the writing process

Managing the Heap

- Processes can acquire dynamic memory on their *heap*
  - The `start_brk` and `brk` fields of the memory descriptor delimit the starting and ending address of the heap, respectively
- The C functions `malloc()`, `calloc()`, `free()`, and `brk()` modify the size of the heap
- `brk()` is the root of all these functions
  - It is the only one that is a system call
  - It directly modified the size of the heap
  - It is actually allocating or releasing logical address space
- One the process actually gets a page frame, the actual memory allocation into small chunks (i.e., `malloc(sizeof(char) * 50)`) is done in user space