Briefing Document: Linux Memory Management

Introduction: This document synthesizes information from three sources to provide a detailed overview of memory management concepts within the Linux operating system. Key areas covered include process contexts, virtual memory, address spaces, the Memory Management Unit (MMU), kernel memory, the use of mmap, and memory-mapped I/O.

I. Process Contexts and Memory Organization

- **Process Context:** When the kernel is entered due to a system call or processor exception (like a page fault), it operates in a *process context*. Execution is synchronous, meaning the flow is linear and predictable. The kernel executes code and manipulates kernel data to complete the requested operation, starting from the system call or exception.
- As the source notes, this is a "*synchronous (top down)*" approach.
- **Interrupt Context:** When the kernel is entered due to a hardware interrupt from a peripheral, it operates in an *interrupt context*. Execution here is asynchronous, as interrupts can happen at any time and preempt the current execution flow. This is considered a "*bottom up*" approach.
- This means, the source states, "*interrupts can occur at any time, making them asynchronous.*"
- **User Address Space (VAS) Segments:** The user VAS is structured into segments or mappings. Key segments include:
- **Text Segment:** Stores machine code, read-only and executable (*r-x*).
- **Data Segments:** Stores global and static data variables. This is further divided into:
- *Initialized Data Segment:* Pre-initialized variables, static.
- *Uninitialized Data Segment (BSS):* Uninitialized variables, auto-initialized to zero at runtime, static.
- **Heap Segment:** Used for dynamic memory allocation via malloc and related APIs.
- **Libraries:** Shared libraries dynamically linked to the process.
- **Stack:** Used for function calls, local variables, and parameter passing. It operates on a LIFO basis and 'grows' towards lower addresses. The text specifies that "*On all modern processors (including the x86 and ARM families), the stack 'grows' toward lower addresses (called a fully descending stack)*"
- **Stacks per Thread:** Each user-space thread has two stacks: a *user-mode stack* for user-mode code and a *kernel-mode stack* when the thread executes in kernel mode (due to a system call or exception). A pure kernel thread only has a kernel-mode stack. The source clarifies "*we require one stack per thread per privilege level supported by the CPU...Thus, on Linux, every user space thread alive has two stacks*."
- **Kernel-Mode Stacks:** Unlike dynamic user-mode stacks, kernel-mode stacks are fixed in size, static, and small (typically 2-4 pages, depending on the architecture).

- **Per-CPU Interrupt Stacks:** Many architectures have a separate stack per CPU for handling hardware interrupts.

II. Virtual Memory

- **Indirection Layer:** Virtual memory provides a layer of indirection, allowing programs to allocate more memory than physically available. The text states that "*With virtual memory, programs running on the system can allocate far more memory than is physically available; indeed, even a single process can have a virtual address space larger than the system's physical memory.*"
- **Process Isolation:** Virtual memory allows each process to have its own memory mapping, protecting memory spaces from other processes. Kernel memory is also invisible to user-space processes. The briefing states, "*Each process can have different memory mapping - One process's RAM inaccessible to other process - Kerneal RAM invisible to user space process*."
- **Swapping:** Memory can be swapped to disk, extending the usable memory beyond physical RAM.
- **Mapping to Devices:** Virtual memory allows mapping of program memory to device memory.
- **MMU (Memory Management Unit):** The hardware responsible for translating virtual addresses to physical addresses. The source mentions that the MMU "*Sits between CPU core and memory...Transpalently handles all memory access from load/store instructions*". The MMU also enforces memory access permissions.
- **Address Spaces:**
- On a 32-bit system, the highest virtual address is 4GB. On a 64-bit system, it's 16EB (exabytes). The document points out that "*On a Linux OS running on (and compiled for) a 64-bit processor, the highest virtual address will be 264 = 16 EB.*"
- The 48-bit addressing scheme on 64-bit systems makes the VAS very sparse, leaving the vast majority of the address space unused.
- **TLB (Translation Lookaside Buffer):** A hardware cache of virtual-to-physical address mappings within the MMU. It stores permission bits. The text explains that "*The TLB is part of MMU systems...if the virtual address is in TLB the MMU can look up the physical resource (RAM or hardware)*"
- **Page Fault:** A CPU exception generated when software tries to use an invalid virtual address. The doc provides the following: "*A page fault is CPU exception, generated when software attempts to use an invalid virtual address. There are 3 cases...*"
- Address is not mapped for the process
- Process lacks permission for the address
- Address is valid but swapped out.
- **Vsyscall Page**: An optimization for performing system calls that don't really need a mode switch to the kernel.

- **KASAN (Kernel Address Sanitizer):** A tool used to detect memory errors such as Use After Free (UAF) and Out Of Bounds (OOB) access. According to the source, "*KASAN: The modern kernel (4.0 onward for x86_64, 4.4 for ARM64) employs a powerful mechanism to detect and report memory issues*."
- **Address Types**: The document identifies various types of addresses:
- **User virtual addresses:** The addresses seen by user-space programs.
- **Physical addresses:** The addresses used between the processor and the system's memory.
- **Bus addresses:** The addresses used between peripheral buses and memory.
- **Kernel logical addresses:** The normal address space of the kernel. They have a direct mapping to physical addresses via a constant offset.
- **Kernel virtual addresses:** A mapping from a kernel-space address to a physical address. They do not necessarily have a linear one-to-one mapping.
- **Physical Memory and Pages:**
- Physical memory is divided into units called *pages*. Memory is handled per page by the system.
- A memory address (virtual or physical) is divided into a page number and an offset within the page. The lower bits of the address are the offset. If you discard the offset, the remaining bits form the *page frame number (PFN)*
- **High and Low Memory:**
- **Low memory**: Memory for which logical addresses exist in kernel space. Typically, all memory is low memory on systems encountered.
- **High memory**: Memory for which logical addresses do not exist. This is memory beyond the address range reserved for kernel virtual addresses. The document states that "*Before accessing a specific high-memory page, the kernel must set up an explicit virtual mapping to make that page available in the kernel's address space*." High memory is mainly for user space process pages.
- **ZONE_NORMAL**: The lower 896 MB of memory, which is directly accessed by the kernel.
- **ZONE_HIGHMEM**: High memory used for page cache and process memory.
- **ZONE_DMA and ZONE_DMA32**: Additional physical memory zones for devices that can perform DMA to a limited amount of physical memory.
- **struct page:**
- A kernel data structure used to manage physical memory. There is one struct page for every physical page. This struct keeps track of everything the kernel needs to know about the page.
- Fields include atomic_t count (reference count), void *virtual (kernel virtual address if mapped), and unsigned long flags (status of the page).

- Functions exist to translate between struct page pointers and virtual addresses: virt_to_page(), pfn_to_page(), and page_address().
- **kmap() and kunmap()**:
- kmap is used to map any page into the kernel address space and returns the virtual address. For low memory, it returns a logical address, and for high memory, it creates a mapping in a dedicated region of kernel address space. The source makes it clear that "*Mappings created with kmap should always be freed with kunmap; a limited number of such mappings is available, so it is better not to hold on to them for too long*."
- kmap can sleep if no mappings are available.
- **Page Tables:** Data structures used by the processor to translate virtual to physical addresses. They are essentially multilevel tree-structured arrays containing these mappings.
- **Virtual Memory Areas (VMAs):**
- Kernel data structure used to manage distinct memory regions of a process address space.
- Represents a contiguous range of virtual addresses with the same permission flags, backed by the same object (file, swap space).
- VMAs can be viewed in the /proc/<pid>/maps file.
- The struct includes fields like vm_start, vm_end, vm_file, vm_pgoff, vm_flags, vm_ops and vm_private_data.

III. mmap() and Memory-Mapped I/O

- **mmap() System Call:** The mmap() system call creates a mapping between a file (or device) and a region in memory. The briefing document quotes the prototype as "*void *mmap(void * addr , size_t length , int prot , int flags , int fd , off_t offset);*". It returns the starting address of the mapping on success.
- **munmap() System Call**: Unmaps a region of memory. As the document shows the prototype is "*int munmap(void * addr , size_t length);*". It returns 0 on success, or -1 on failure.
- **Benefits of mmap():**
- Simplifies application logic by replacing read() and write() with direct memory access.
- Improves performance by avoiding system calls. The document states that "*memory-mapping facility allows direct memory access to a user space device*".
- **Mapping Modes:**
- **MAP_PRIVATE:** Creates a private mapping. Modifications are not visible to other processes using the same mapping and are not carried over to the underlying file.
- **MAP_SHARED:** Creates a shared mapping. Modifications are visible to other processes using the same mapping and, for file mappings, are carried over to the underlying file.

- **msync() System Call:** Controls synchronization of shared memory mappings with the underlying file. The prototype is "*int msync(void * addr , size_t length , int flags);*".
- MS_SYNC: Synchronous file write.
- MS_ASYNC: Asynchronous file write.
- **mlock() and munlock():** System calls used to lock/unlock parts of a process's virtual memory, ensuring pages are always in physical memory.
- **Device Driver mmap():**
- Implemented in the driver's file_operations structure.
- The driver allocates memory (using kmalloc(), vmalloc(), etc.) and maps it to the user space.
- Helper functions like remap_pfn_range() are used for mapping physical memory to virtual addresses. The document notes that "*A driver should allocate memory (using kmalloc(), vmalloc(), alloc_pages()) and then map it to the user address space as indicated by the vma parameter using helper functions such as remap_pfn_range()*."
- **remap_pfn_range()**: A kernel function that creates page tables to map physical addresses to a virtual address range. It is used by a device driver during the mmap call. Its prototype is "*int remap_pfn_range(struct vm_area_struct *vma, unsigned long virt_addr, unsigned long pfn, unsigned long size, pgprot_t prot)*".
- vma: Virtual memory area being mapped into.
- virt_addr: Virtual address where remapping begins.
- pfn: Page frame number corresponding to physical memory.
- size: Size of the area being remapped.
- io_remap_page_range is similar but used when phys_addr points to I/O memory.
- **VMA Operations:** VMAs have functions for managing their lifecycle, called open, close and nopage which is called when the page is not in physical memory.

IV. Memory-Mapped I/O vs. Port I/O

- **Memory-Mapped I/O:** Device registers are accessed by mapping them to memory addresses.
- **Port I/O:** Devices are accessed via specialized I/O ports and instructions like inb/outb
- **Accessing Memory-Mapped I/O:**Can be done from user space via /dev/mem with mmap(). The source highlights the following lines of code "*virt_addr = (uint8_t *)mmap(NULL, size , PROT_READ | PROT_WRITE, MAP_SHARED, fd, base_addr)*" to map I/O memory into virtual space.
- In kernel space using devm_ioremap_resource(). The document mentions that "*devm_ioremap_resource() managed API performs the job of (validity) checking the requested I/O memory region, requesting it from the kernel*".

- **Memory Barriers:** Used to enforce ordering of memory reads and writes across hardware boundaries (e.g., between CPU and peripheral devices) to prevent compiler/processor reordering from creating issues. Macros such as rmb(), wmb(), and mb() are used to insert these barriers. The text states that "*Memory barriers can be placed into the code path by using the following macros: #include <asm/barrier.h> :*".

V. Kernel Memory Allocation

- The kernel uses a buddy system allocator with power of two page sized chunks when allocating memory.
- Requests for memory are handled by first attempting to allocate memory from the requested order, otherwise larger blocks are recursively taken, split, and placed into the available orders until the requested memory is available.
- vmstat -m command can be used to show the kernel allocation pools.

Conclusion: This briefing document has covered many key facets of the Linux memory management system, including different memory contexts, virtual memory, memory mapping via mmap, the MMU, and kernel memory concepts. It has also detailed the use of structures and operations required in kernel mode to provide memory access to user space. These concepts are critical for understanding how Linux manages memory resources and interacts with hardware devices.