Okay, I have analyzed the content of "Chapter -6.pdf" on The Memory System. I will now break down the chapter, explain each topic in detail, highlight key information, provide revision lists, and generate potential exam questions based on the content.

Here is a structured explanation of the chapter:

**Chapter 6: The Memory System**

This chapter discusses how operating systems manage memory, focusing on techniques and algorithms for both user and kernel space memory management. Efficiency and correctness are key concerns, especially regarding page replacement and kernel memory allocation.

**6.1 Traditional Heuristics for Page Allocation**

This section covers older methods of memory management, particularly relevant for systems without virtual memory or small embedded devices.

**6.1.1 Base-Limit Scheme**

* **Explanation:** In systems without virtual memory, memory for a process is allocated contiguously. Two registers, a **base register** and a **limit register**, are used for isolation and addressing. The base register stores the starting physical address of the process's memory region. The limit register stores the size of this region. Any logical address issued by the processor is translated into a physical address by adding the base address to the logical address (physical\_address=base+logical\_address). This physical address is then checked against the limit register; if it's less than the limit, the address is valid, otherwise, it's out of bounds.
* **Diagram:** Figure 6.1 illustrates this concept, showing allocated regions and "holes" in memory.
  + [Image 1]
* **Key Definitions:**
  + **Base Register:** Stores the starting physical address of a process's memory region.
  + **Limit Register:** Stores the size of a process's memory region.
  + **External Fragmentation:** The phenomenon in base-limit systems where free space exists between allocated memory regions in the form of "holes."
  + **Internal Fragmentation:** The phenomenon in virtual memory systems where memory space within a page is left empty. The wastage per page is typically limited (e.g., 4 KB).
* **Memory Allocation Heuristics (for choosing a hole when a new process is created):**
  + **Best Fit:** Choose the smallest hole that is just large enough to satisfy the memory request.
  + **Worst Fit:** Choose the largest available hole.
  + **Next Fit:** Start searching for a hole from where the last allocation was made and wrap around the memory space if necessary.
  + **First Fit:** Choose the first available hole that is large enough.
* **Why these schemes are less common now:** These schemes require knowing the maximum memory a process needs upfront, which is difficult for modern complex programs whose memory usage depends heavily on input. While still relevant for small embedded devices without virtual memory, they are largely considered outdated.
* **Quick Revision:**
  + Base-Limit uses base and limit registers for address translation and isolation.
  + Memory is allocated contiguously.
  + Address translation: physical=base+logical.
  + Checks address against limit.
  + Suffers from external fragmentation (holes between allocated regions).
  + Different heuristics (Best Fit, Worst Fit, Next Fit, First Fit) for hole selection.
  + Not widely used in modern systems due to unpredictable memory requirements.

**6.1.2 Classical Schemes to Manage Virtual Memory**

This section shifts to memory management in systems utilizing virtual memory, where the primary challenge is page replacement when memory is full.

* **Explanation:** In virtual memory systems, managing free memory frames is relatively straightforward using structures like a bit vector for free frames and an augmented tree to quickly find a free frame (e.g., in log(N)time). Internal fragmentation is a minor issue. The critical challenge is deciding which page to replace when physical memory (frames) is full. This significantly impacts the page fault rate, and page faults are expensive due to slow access to storage devices.
* **The Stack Distance:**
  + **Explanation:** A conceptual tool to understand temporal locality. All physical pages in memory are imagined as being in a stack. When a page is accessed, it's moved to the top of the stack. The stack distance is the distance between the top of the stack and the page's position before it was moved. This is used to evaluate temporal locality.
  + **Diagram:** Figure 6.2 visually represents the notion of stack distance.
    - [Image 2]

* + **Representative Plot:** Figure 6.3 shows a typical heavy-tailed distribution of stack distances. Low values are infrequent, followed by a peak in the low to medium range, and a heavy tail for large distances. This shape reflects both simultaneous access to multiple data/instruction streams (low values) and significant temporal locality (peak), as well as random accesses and phase changes (heavy tail). The log-normal distribution is often used to model this curve.
  + **Diagram:**
    - [Image 3]
  + **Significance:** Lower average stack distance indicates higher temporal locality (pages are re-accessed quickly). Higher average stack distance indicates lower temporal locality (pages are re-accessed after a long time).
* **Stack-based Algorithms:**
  + **Explanation:** A family of page replacement algorithms with predictable behavior. They conceptually organize pages in a stack based on a replacement priority. The top 'n' pages (where 'n' is the number of physical frames) are considered to be in main memory. Accessing a page changes its priority and moves it to a new position in the stack. The stack distance concept is an example where pages are ordered by last access time.
* **Optimal Page Replacement Algorithm:**
  + **Explanation:** An ideal, but impractical, algorithm that minimizes page faults by replacing the page that will be used furthest in the future. It requires knowing the future access pattern. This is a stack-based algorithm where priority is inversely proportional to the next-use time. It's provably optimal in minimizing page faults.
* **Least Recently Used (LRU) Algorithm:**
  + **Explanation:** A practical approximation of the optimal algorithm. It replaces the page that has not been accessed for the longest time. The assumption is that past access is a good predictor of future access. This is a stack-based algorithm where priority is inversely proportional to the last-used time. Accessing a page moves it to the top of the conceptual stack.
  + **Practical Implementation Challenges:** Maintaining accurate last-accessed timestamps for every page on every memory access is computationally expensive and introduces significant overhead. Storing timestamps in page table entries adds overhead. Finding the minimum timestamp in a priority queue is O(log(N)), which is still too slow for every memory access.
  + **Approximations using Access Bits:** Modern systems use approximations. A common technique leverages page protection bits (access bits) in the page table and TLB. Pages are periodically marked "not accessible" (access bit set to 0). The first access after resetting the bit causes a "soft page fault". An exception handler records the time and sets the access bit to 1 for subsequent seamless access. Pages with an access bit still at 0 after a period are considered not recently accessed and are candidates for replacement. This is a coarse-grained but fast approximation. Storing timestamps with page table entries and using the timestamp when the access bit transitions from 0 to 1 as a proxy for recency is another idea, but its accuracy is questionable in practice. The WS-Clock family of algorithms are real-world LRU approximations.
* **WS-Clock Algorithm:**
  + **Explanation:** A simple LRU approximation. Each physical page has an associated access bit. A pointer conceptually moves through the pages in a circular manner. If the pointer encounters a page with the access bit set to 1, it resets it to 0 and moves on. If the access bit is 0, that page is selected for replacement. The pointer stops at this page and continues from there next time. This approximates finding pages not recently used.
  + **Diagram:** Figure 6.4 illustrates the WS-Clock algorithm.
    - [Image 4]

* **WS-Clock Second Chance Algorithm:**
  + **Explanation:** An improved version that uses two bits: an access bit and a modified bit. The modified bit is set when the page is written to. This algorithm prioritizes unmodified pages for eviction because modified pages need to be written back to swap space, which is an overhead.
  + **State-Action Table:** Table 6.1 shows the actions taken based on the combination of access and modified bits:
    - (0,0): Replace the page.
    - (0,1): Schedule a write-back, set state to (0,0), and move forward.
    - (1,0): Set state to (0,0) and move forward (gives a second chance).
    - (1,1): Schedule a write-back, set state to (1,0). This indicates a frequently used, modified page that shouldn't be evicted immediately.
  + **Diagram:**
    - [Image 4] (Note: This diagram is for the basic WS-Clock, the table is key for the second chance algorithm)
  + **Overheads:** These LRU approximating algorithms are heavy because they introduce artificial page faults (soft page faults). While faster than hard page faults (fetching from storage), they still involve exception handling and bookkeeping.
* **FIFO Algorithm:**
  + **Explanation:** A simple algorithm that replaces the page that has been in memory the longest. It uses a queue to track the order pages were brought into memory. It has no runtime overhead for tracking usage. This algorithm is *not* stack-based.
  + **Belady's Anomaly:** FIFO suffers from this anomaly, where increasing the number of physical frames can *increase* the number of page faults. This happens because it's not stack-based; stack-based algorithms guarantee that the set of pages in a larger memory is a superset of the pages in a smaller memory. This anomaly can be arbitrarily large.
  + **Diagrams:** Figures 6.5 and 6.6 demonstrate Belady's Anomaly with examples of access sequences and page faults for 4 frames (10 faults) and 3 frames (9 faults).
    - [Image 5]
    - [Image 6]

* + **Consequences:** Belady's anomaly makes non-stack-based algorithms like FIFO ineffective, as their worst-case performance can be arbitrarily poor.
* **Quick Revision:**
  + Virtual memory management focuses on page replacement.
  + Page faults are expensive.
  + Stack distance measures temporal locality.
  + Stack-based algorithms have predictable behavior and avoid Belady's anomaly.
  + Optimal algorithm is ideal but impractical (requires future knowledge).
  + LRU is a practical approximation (replaces least recently used).
  + Pure LRU implementation is too slow.
  + LRU is approximated using access bits (soft page faults) or WS-Clock.
  + WS-Clock Second Chance uses access and modified bits to prioritize eviction of unmodified pages.
  + FIFO is simple but suffers from Belady's anomaly.
  + Belady's anomaly: more memory can lead to more page faults.
* **Mnemonics/Memory Tricks:**
  + **FIFO:** **F**irst **I**n, **F**irst **O**ut - like a queue. Remember the anomaly: Belady's **F**oolish **O**utcome.
  + **LRU:** **L**east **R**ecently **U**sed - think of something you haven't touched in a long time, you're least likely to need it now.

**6.1.3 The Notion of the Working Set**

This section introduces the concept of a working set and its implications for system performance.

* **Explanation:** Informally, the working set of a program is the set of pages it actively accesses in a short time window. It's the set of pages the program repeatedly uses. A more formal definition is tied to the relationship between the number of pages allocated to a program and its page fault rate.
* **Diagram:** Figure 6.7 shows the page fault rate versus the number of pages in memory.
  + [Image 7]
* **Working Set Size:** The point on the graph where the page fault rate sharply decreases is considered the working set size. If a program is allocated fewer pages than its working set size, it will have a high page fault rate. If it has more, the page fault rate will be low. The pages in memory at this threshold constitute the working set. This behavior is observed in most real-world programs.
* **Thrashing:**
  + **Explanation:** Thrashing occurs when the total working sets of all active processes exceed the available physical memory. Processes constantly compete for memory, leading to frequent page faults as they evict each other's working set pages. The system spends most of its time swapping pages to and from storage, resulting in low CPU utilization (due to waiting for I/O) but high CPU load (due to servicing page faults). The kernel might incorrectly try to increase CPU utilization by scheduling more processes, which worsens the thrashing. This creates a vicious cycle that can slow the system down significantly and even lead to crashes.
  + **Real-world example:** The text describes the relatable scenario of a user opening many applications during boot-up, causing thrashing and slowing the system down. The solution is to wait and allow the system to load necessary pages.
  + **Prevention:** Modern operating systems detect thrashing using counters. The practical solution is to reduce the number of active processes so that their working sets fit in memory. Linux attempts to prevent thrashing by keeping working set pages in memory and not evicting recently loaded pages for a certain period (e.g., 1 second). If a new process needs memory but cannot get it due to these policies, the OOM killer might terminate it.
  + **Definition:** Thrashing is the phenomenon where too many applications run simultaneously, their working sets don't fit in memory, leading to constant page eviction and high page fault rates, ultimately slowing down or crashing the system.
* **Quick Revision:**
  + Working set is the set of actively used pages.
  + Working set size is the memory threshold for low page fault rate.
  + Thrashing occurs when working sets exceed physical memory.
  + Thrashing leads to high page faults, low CPU utilization, and system slowdown/crashes.
  + Kernel might worsen thrashing by scheduling more processes.
  + Prevention involves reducing active processes, protecting recent pages, and OOM killing.
* **Mnemonics/Memory Tricks:**
  + **Working Set:** Think of the set of tools you keep on your desk for your current task - that's your "working set."
  + **Thrashing:** Imagine the system is "thrashing" around trying to find pages, constantly struggling and getting nowhere.

**6.2 Virtual and Physical Address Spaces**

This section delves into the organization of virtual and physical memory in Linux, including page tables, struct page, folios, TLB management, and physical memory partitioning (NUMA and zones).

* **Explanation:** Linux memory management involves the design of the virtual memory space, page tables, and associated structures like struct page. Folios, groups of contiguous pages in both virtual and physical space, are a recent addition for efficiency.

**6.2.1 The Virtual Memory Map**

* **Explanation:** The virtual address space in Linux is partitioned between user processes and kernel threads, with no overlap. For example, in a 57-bit system, 64 PB can be for user space and 64 PB for the kernel. The user space has standard sections like text, stack, and heap. The kernel virtual memory map has several important regions.
  + **Direct-mapped region:** A large area where virtual and physical addresses are linearly related (or the same). This allows direct physical memory access, needed for I/O devices, DMA, booting, and storing page tables. It facilitates efficient access to large contiguous memory chunks, reducing TLB misses and translation overhead. Often used with huge pages (2MB, 1GB) to further reduce TLB pressure.
  + **Memory-mapped I/O region:** Maps I/O devices to memory.
  + **Per-CPU area:** Stores per-CPU data like current task info, context, and preemption flags.
  + **Kernel code + modules:** Region for the core kernel code and dynamically loaded modules (device drivers, file systems, etc.). Modules help keep the kernel small and modular, but security is a concern.
* **Diagram:** Figure 6.8 shows a representative layout of the kernel and user virtual memory map.
  + [Image 8]
* **Quick Revision:**
  + Virtual memory is split between user and kernel space.
  + Kernel virtual memory has specific regions: direct-mapped, memory-mapped I/O, per-CPU, code/modules.
  + Direct-mapped region allows direct physical memory access for kernel needs.
  + Huge pages reduce TLB pressure.
  + Modules allow dynamic loading of kernel code.

**6.2.2 The Page Table**

* **Explanation:** The page table (pgd\_t \*pgd in the mm\_struct) is crucial for translating virtual addresses to physical addresses. In Linux, it's also called the page directory. Modern systems use multi-level page tables (e.g., 5 levels for 57-bit addresses). The highest level is the Page Global Directory (PGD), pointed to by the CR3 register.
  + **CR3 Register:** Stores the physical address of the PGD for the current process. Switching processes requires changing CR3 and flushing the TLB, which is expensive. Optimizations exist to avoid flushing on user-to-kernel transitions.
  + **Multi-level structure:** The virtual address bits are divided to index each level of the page table. The goal is to reduce memory footprint and leverage sparsity in the virtual address space.
  + **Page Table Entry (PTE):** The lowest level entry, containing the mapping between a virtual page number and a physical page frame number, along with permission bits.
* **Diagram:** Figure 6.9 shows the structure of the mm\_struct and the 5-level page table organization.
  + [Image 9]
* **Page Table Levels:** Table 6.2 lists the acronyms and full names of the 5-level page table components.
* **Page Protection Bits (pgprot\_t):** Table 6.3 lists the permission bits stored in a PTE and TLB entries. These bits control access (read, write, execute, atomic operations, no access) for security and memory management (like LRU tracking).
* **Walking the Page Table:**
  + **Explanation:** The process of traversing the multi-level page table using the virtual address to find the corresponding PTE and physical address. Listing 6.1 shows the follow\_pte function, which demonstrates this traversal through the different levels (PGD, P4D, PUD, PMD, PTE). A spinlock is acquired to allow modification of the PTE.
  + **Code Example (Listing 6.1):**

C

int follow\_pte (struct mm\_struct \*mm, unsigned long address,

pte\_t \*\*ptepp, spinlock\_t \*\*ptlp) {

pgd\_t \*pgd;

p4d\_t \*p4d;

pud\_t \*pud;

pmd\_t \*pmd;

pte\_t \*ptep;

pgd = pgd\_offset(mm, address);

p4d = p4d\_offset(pgd, address);

pud = pud\_offset(p4d, address);

pmd = pmd\_offset (pud, address);

ptep = pte\_offset\_map\_lock(mm, pmd, address, ptlp);

\*ptepp = ptep;

return 0;

}

* + - **Explanation of Code:** This C function takes the memory structure (mm\_struct), the virtual address, pointers to a pte\_t pointer (ptepp) and a spinlock pointer (ptlp) as input. It retrieves the base address of the PGD from the mm\_struct. Then, using offset functions for each level (pgd\_offset, p4d\_offset, pud\_offset, pmd\_offset), it traverses down the page table hierarchy using parts of the virtual address to find the corresponding entry at each level. Finally, pte\_offset\_map\_lock finds the PTE and acquires a lock, returning a pointer to the PTE via ptepp. The function returns 0 on success.
  + **Accessing Table Entries (Listing 6.2):** Listing 6.2 shows how to access a PMD entry from a PUD table entry. This involves calculating the index within the table using bit shifts and masks on the virtual address (pmd\_index) and adding it to the base address of the table (pud\_pgtable).
  + **Code Example (Listing 6.2):**

C

pmd\_t \*pmd\_offset(pud\_t \*pud, unsigned long address) {

return pud\_pgtable(\*pud) + pmd\_index(address);

}

unsigned long pmd\_index (unsigned long address) {

return (address >> PMD\_SHIFT) & (PTRS\_PER\_PMD - 1);

}

pmd\_t \*pud\_pgtable (pud\_t pud) {

return (pmd\_t \*) \_\_va (pud\_val (pud) & pud\_pfn\_mask (pud));

}

#define \_\_va(x) ((void \*) ((unsigned long)(x)+PAGE\_OFFSET))

* + - **Explanation of Code:**
      * pmd\_offset: Takes a pointer to a PUD entry (pud) and a virtual address. It calculates the index of the PMD entry within the PMD table using pmd\_index(address) and adds it to the base virtual address of the PMD table (obtained from pud\_pgtable(\*pud)) to get a pointer to the desired PMD entry. Pointer arithmetic automatically accounts for the size of pmd\_t.
      * pmd\_index: Takes a virtual address. It shifts the address right by PMD\_SHIFT bits to get the relevant bits for indexing the PMD table. It then performs a bitwise AND with (PTRS\_PER\_PMD - 1) to isolate the lower bits corresponding to the index within the table.
      * pud\_pgtable: Takes a PUD entry (pud). It extracts the physical address bits from the PUD entry using pud\_val(pud) and masks them with pud\_pfn\_mask(pud) to get the page frame number (PFN) of the PMD table. It then converts this physical address to a virtual address using the \_\_va macro.
      * \_\_va(x): A macro that converts a physical address x to a virtual address by adding PAGE\_OFFSET. PAGE\_OFFSET is the starting address of the direct-mapped kernel memory region where page tables are stored.
* **Quick Revision:**
  + Page table translates virtual to physical addresses.
  + Linux uses multi-level page tables.
  + CR3 register points to the current process's PGD.
  + PTEs store virtual-to-physical mappings and permission bits.
  + Walking the page table involves traversing down the levels using parts of the virtual address.
  + Kernel code uses bit shifts and masks to calculate indices within page tables.
  + \_\_va macro converts physical to virtual addresses in the direct-mapped region.

**6.2.3 Pages and Folios**

* **Explanation:**
  + **struct page:** A data structure in the kernel that stores metadata for each physical page (frame). This includes information like the page's type (anonymous, memory-mapped), status (locked, modified, active), and a reference count. It's a complex structure often using unions to accommodate different types of data a page can represent. The reference count tracks how many entities use the page; it must be zero before the page is recycled.
  + **Folios:** A newer concept (v5.18+) representing a compound or aggregate page consisting of two or more contiguous physical pages. They were introduced to handle the large number of pages in modern systems and reduce translation and metadata overhead. A folio points to the first page in the group and stores the number of pages it contains.
  + **Folio Advantages:** Originally focused on contiguous virtual pages for prefetching, the concept evolved to also include contiguous physical addresses for benefits with I/O, DMA, and reduced translation overheads. Folios integrate well with huge pages, simplifying their management and benefiting I/O/DMA devices that need contiguous physical memory access. Usage and replacement information (LRU stats) are maintained at the folio level. When a process forks, folios are treated as a single unit for copy-on-write.
* **Mapping PFN to struct page and vice versa (Listing 6.3):** The kernel needs to convert between a page frame number (PFN) and its corresponding struct page.
  + **Code Example (Listing 6.3):**

C

#define pte\_pfn(x) phys\_to\_pfn(x.pte)

#define phys\_to\_pfn(p) ((p) >> PAGE\_SHIFT)

#define \_\_pfn\_to\_page(pfn) \

({ unsigned long pfn = (pfn); \

struct mem\_section \*\_sec = \_pfn\_to\_section(\_\_pfn); \

\_\_section\_mem\_map\_addr(\_\_sec) + pfn; \

})

* + - **Explanation of Code:**
      * pte\_pfn(x): A macro that extracts the PFN from a page table entry x by calling phys\_to\_pfn.
      * phys\_to\_pfn(p): A macro that converts a physical address p to a PFN by right-shifting by PAGE\_SHIFT (which is typically 12, effectively dividing by the page size of 212=4096).
      * \_\_pfn\_to\_page(pfn): A macro that converts a PFN to a pointer to the corresponding struct page. The simple version assumes a linear array of struct page and adds the PFN as an offset. The code shown is a more complex variant for systems that divide memory into sections (mem\_section). It determines the section the PFN belongs to (\_pfn\_to\_section), finds the base address of that section's mem\_map (which stores the PFN-to-struct pagemappings), and adds the PFN to get the address of the struct page.
* **Quick Revision:**
  + struct page stores metadata for each physical page.
  + Folios group contiguous physical pages (and often virtual pages).
  + Folios reduce overhead and benefit I/O/DMA.
  + Folios are managed as single units for LRU, COW, etc.
  + Macros like pte\_pfn and \_\_pfn\_to\_page convert between PTEs, PFNs, and struct page.
* **Mnemonics/Memory Tricks:**
  + **Folio:** Think of a "folio" as a collection of pages, like a book.

**6.2.4 Managing the TLB**

* **Explanation:** The Translation Lookaside Buffer (TLB) is a cache of recent virtual-to-physical address translations. TLB misses are expensive as they require walking the page table. Efficient TLB management is critical for performance.
  + **TLB Design:** TLBs are typically associative caches with multiple levels (i-TLB for instructions, d-TLB for data, shared L2 TLB). Each entry stores a virtual page number mapping to a physical frame number, along with permission bits.
  + **TLB Flushes:** When a process switches, the CR3 register changes to point to the new process's page table. The TLB entries from the previous process are no longer valid and must be flushed, which is expensive.
  + **Optimizations:**
    - **Global (G) bit:** Intel processors allow marking kernel space entries as global so they are not flushed during process switches.
    - **invlpg instruction:** Allows selectively invalidating specific TLB entries instead of flushing the entire TLB.
    - **Virtual Address Space Partitioning:** Splitting user and kernel space avoids flushing kernel entries when switching to the kernel.
    - **Address Space IDs (ASIDs) / Processor Context IDs (PCIDs):** Annotating TLB entries with a process ID allows the TLB to hold entries for multiple processes simultaneously. Memory accesses include the current process's ID, and only matching TLB entries are used. This blurs the line between software (process ID) and hardware. Intel x86 processors use PCIDs (typically 4096 available, one reserved), stored in the top bits of CR3. Linux supports ASIDs (architecture-independent) and maps them to PCIDs on x86-64. The kernel might use a smaller number of ASIDs for broader architecture compatibility (e.g., 6 in v6.2). The INVPCID instruction invalidates entries for a specific PCID.
  + **Lazy TLB Mode:** For multithreaded processes sharing an address space across multiple cores, TLB modifications on one core must be broadcast to others (via Inter-Processor Interrupts - IPIs) for consistency. In lazy TLB mode, invalidations can be deferred for cores not currently running the affected process. Consistency checks are performed at specific kernel entry points (like copy\_from\_user, copy\_to\_user) or before switching back to a process with deferred invalidations. This avoids interrupting high-priority tasks.
* **Quick Revision:**
  + TLB caches virtual-to-physical translations.
  + TLB misses are costly page walks.
  + Process switches require TLB flushes (expensive).
  + Optimizations include global bits, selective invalidation (invlpg), and ASIDs/PCIDs.
  + ASIDs/PCIDs tag TLB entries with process IDs.
  + Lazy TLB mode defers TLB invalidations for efficiency.
* **Mnemonics/Memory Tricks:**
  + **TLB:** **T**ranslation **L**ookaside **B**uffer - it helps "look aside" the page table for quick translations.
  + **ASID/PCID:** Think of a unique "ID" for each address space (process) to keep their TLB entries separate.

**6.2.5 Partitioning Physical Memory**

* **Explanation:** Modern large systems often have non-uniform memory access (NUMA) architectures where memory access latency varies depending on the CPU's proximity to the memory. The OS needs to manage this heterogeneity for performance.
  + **NUMA Machines:** Systems with multiple CPU clusters (nodes) connected by an interconnect. Memory within a node (local memory) is faster to access than memory in another node (remote memory). The goal is to keep data and code within local memory to minimize remote accesses and lower average access time. Each cluster is a "node." Physical addresses are managed hierarchically.
  + **Diagram:** Figure 6.10 shows a diagram of a NUMA machine with nodes and interconnect.
    - [Image 10]

* + **Zones:** Linux partitions physical memory (page frames) into non-overlapping sets called zones to manage different types of memory or memory with different characteristics. This extends the NUMA concept to include different types of memory devices (DRAM, NVM, flash, etc.) with varying performance characteristics. Memory-mapped I/O and DMA pages can also be assigned to specific zones.
  + **Zone Types (Listing 6.4):**
    - **ZONE\_DMA:** Reserved for physical pages accessible by the DMA controller. Pages here are typically uncached to avoid consistency issues with DMA devices.
    - **ZONE\_NORMAL:** For regular kernel and user pages.
    - **ZONE\_HIGHMEM:** (If configured) For physical memory exceeding the virtual address space size (relevant on older/embedded systems).
    - **ZONE\_MOVABLE:** Contains pages that can be easily moved or reclaimed by the kernel. This helps create large contiguous free regions for allocations benefiting from physical contiguity (e.g., for huge pages, databases). Pages pinned by kernel processes can hinder this.
    - **ZONE\_DEVICE:** (If configured) For pages stored on novel memory devices like NVMs, graphics card memory, etc.. Pages here are managed by the device, typically not cached, and can be accessed directly by DMA controllers without CPU involvement. NVMs offer persistence and are faster than disks but slower than DRAM.
  + **Code Example (Listing 6.4 - enum zone\_type):**

C

enum zone\_type {

/\* Physical pages that are only accessible via the DMA

\* controller \*/

ZONE\_DMA,

/\* Normal pages \*/

ZONE\_NORMAL,

/\* Useful in systems where the physical memory exceeds

\* the size of max virtual memory.

\* We can store the additional frames here \*/

#ifdef CONFIG\_HIGHMEM

ZONE\_HIGHMEM,

#endif

/\* It is assumed that these pages are freely movable and

\* reclaimable \*/

ZONE\_MOVABLE,

/\* These frames are stored in novel memory devices like

\* NVM devices, \*/

#ifdef CONFIG\_ZONE\_DEVICE

ZONE\_DEVICE,

#endif

/\* Dummy value indicating the number of zones \*/

\_\_MAX\_NR\_ZONES

};

* + - **Explanation of Code:** This is an enumeration defining the different types of memory zones supported by the Linux kernel. Each member represents a distinct category of physical memory pages managed by the system. Conditional compilation (#ifdef) is used for ZONE\_HIGHMEM and ZONE\_DEVICE, meaning these zones are only included if the corresponding kernel configuration options are enabled. \_\_MAX\_NR\_ZONES is a placeholder to indicate the total number of zone types.
  + **Sections:** Zones can be further divided into sections (mem\_section). This creates a 2-level hierarchy for better management of free frames within smaller structures. Sections also handle noncontiguous zones by breaking them into contiguous chunks and can accommodate intra-zone heterogeneity (e.g., different latencies within a zone). Each section has a mem\_map to store the PFN-to-struct page mapping.
  + **Diagram:** Figure 6.11 illustrates the relationship between zones, sections, PFNs, and struct page.
    - [Image 11]

* + **struct zone (Listing 6.5):** Contains details about a specific zone, including its associated NUMA node (node, zone\_pgdat), the range of PFNs it spans (zone\_start\_pfn, spanned\_pages), the number of pages it currently manages (managed\_pages, present\_pages), its name, and a list of free areas (free\_area) used by the buddy allocator for contiguous memory allocation. Note that spanned\_pages indicates the total range of PFNs, while present\_pages indicates the actual number of physical pages in the zone.
  + **Code Example (Listing 6.5 - struct zone):**

C

struct zone {

int node; /\* NUMA node\*/

/\* Details of the

\* NUMA node\*/

struct pglist\_data \*zone\_pgdat;

/\* zone\_end\_pfn

\* zone\_start\_pfn + spanned\_pages - 1\*/

unsigned long zone\_start\_pfn;

atomic\_long\_t managed\_pages;

unsigned long spanned\_pages;

unsigned long present\_pages;

/\* Name of the zone

\*/

const char \*name;

/\* List of the free

\* areas in the zone (managed by the

\* buddy allocator)

\*/

struct free\_area free\_area [MAX\_ORDER];

};

* + - **Explanation of Code:** This C structure defines a zone in the Linux kernel's memory management. node and zone\_pgdat link the zone to a specific NUMA node. zone\_start\_pfn and spanned\_pagesdefine the range of physical frame numbers covered by the zone. managed\_pages and present\_pages track the number of pages actively managed and physically present in the zone, respectively. name is a descriptive string for the zone. free\_area is an array used by the buddy allocator to manage free blocks of different sizes within the zone.
  + **struct pglist\_data (Listing 6.6):** Contains details about a NUMA node, including its ID (node\_id), an array of zones within that node (node\_zones), a global list of zones across all nodes (node\_zonelists), the number of zones in the node (nr\_zones), and the number of pages owned by the node (node\_present\_pages, node\_spanned\_pages). It also includes a pointer to the kswapd process (a background daemon for swapping pages) and LRU state information (lruvec).
  + **Code Example (Listing 6.6 - struct pglist\_data):**

C

typedef struct pglist\_data {

/\* NUMA node id \*/

int node\_id;

/\* Hierarchical organization of zones \*/

struct zone node\_zones [MAX\_NR\_ZONES];

struct zonelist node\_zonelists [MAX\_ZONELISTS];

int nr\_zones;

/\* #pages owned by the NUMA node (node\_id) \*/

unsigned long node\_present\_pages;

unsigned long node\_spanned\_pages;

/\* Pointer to the page swapping daemon \*/

struct task\_struct \*kswapd;

/\* LRU state information\*/

struct lruvec \_\_lruvec;

} pg\_data\_t;

* + - **Explanation of Code:** This C structure, aliased as pg\_data\_t, holds data for a NUMA node. node\_id is the identifier. node\_zones is an array of zones belonging to this node. node\_zonelists is a global list of zones. nr\_zones counts zones in this node. node\_present\_pages and node\_spanned\_pages track the page count. kswapd points to the swapping daemon for this node. \_\_lruvec stores LRU data.
* **Quick Revision:**
  + NUMA systems have varying memory access latencies based on proximity.
  + Physical memory is partitioned into zones.
  + Zones manage different types of memory or memory with different characteristics.
  + Zone types include ZONE\_DMA, ZONE\_NORMAL, ZONE\_HIGHMEM, ZONE\_MOVABLE, ZONE\_DEVICE.
  + Zones can be divided into sections for better management and to handle non-contiguity.
  + struct zone contains detailed information about a zone.
  + struct pglist\_data (or pg\_data\_t) holds information about a NUMA node and its zones.
  + kswapd is a background process for swapping pages.
* **Mnemonics/Memory Tricks:**
  + **NUMA:** **N**on-**U**niform **M**emory **A**ccess - not all memory is equally fast to access.
  + **Zones:** Think of dividing the physical memory into different "zones" based on their properties or purpose.

**6.3 Page Management**

This section focuses on page management techniques, particularly reverse mapping and the MGLRU page replacement algorithm used in Linux.

* **Explanation:** Efficient page management requires understanding concepts like Bloom filters and reverse mapping. Bloom filters are probabilistic data structures for set membership testing that allow false positives but no false negatives. Reverse mapping tracks which processes map to a given physical page.

**6.3.1 Reverse Mapping**

* **Explanation:** Reverse mapping is needed to find all the virtual pages (across different processes) that map to a specific physical page. This is essential when a physical page is swapped out or its permissions change, as the corresponding page table entries in all mapping processes need to be updated. Page sharing, even without explicit shared memory, is common in Linux due to the fork operation and copy-on-write (COW).
  + **vm\_area\_struct (vma):** Represents a contiguous region of a process's virtual memory address space and stores information like start/end addresses and permissions. Page sharing happens at the vma granularity.
  + **Anonymous vs. File-backed pages:** Pages can be anonymous (not file-backed, like stack/heap) or file-backed (memory-mapped files, managed by the page cache). Reverse mapping mechanisms differ for these types.
  + **The Problem:** Given a physical page, finding all processes that map to it is challenging. Storing a list of processes in each struct page is inefficient due to variable list size and redundancy for shared pages. Adding a pointer to a vma in each struct page is better but problematic because vmas split and merge frequently.
  + **anon\_vma:** An intermediary structure introduced to virtualize vmas and handle the many-to-many relationship between vmas and pages. A physical page points to a single anon\_vma via its mapping field. Multiple vmas (across processes, e.g., after a fork) can point to the same anon\_vma.
  + **Diagrams:** Figure 6.12 shows pages pointing to an anon\_vma. Figure 6.13 illustrates many-to-one (vmas to anon\_vma) and one-to-many (vma to anon\_vmas after COW) relationships. Figure 6.14 depicts the overall many-to-many mapping.
    - [Image 12]
    - [Image 13]

* + **Copy-on-Write (COW):** When a COW page shared via an anon\_vma is written to by a child process, a new private copy of the page is created for the child. This new page points to a *new* anon\_vma exclusive to the child. The child's vma is typically *not* split to avoid excessive vma structures.
  + **anon\_vma\_chain (Listing 6.7):** An intermediary structure linking a vma and an anon\_vma. This avoids scalability issues with direct linked lists between vmas and anon\_vmas. An anon\_vma\_chain contains pointers to a vma (vma) and an anon\_vma (anon\_vma), and is part of lists (same\_vma) and red-black trees (rb) for efficient traversal.
  + **Code Example (Listing 6.7 - anon\_vma\_chain):**

C

struct anon\_vma\_chain {

struct vm\_area\_struct \*vma; /\* pointer to a vma \*/

struct anon\_vma \*anon\_vma; /\* pointer to an anon\_vma

\*/

struct list\_head same\_vma; /\* pointer to other

\* anon\_vma\_chains corresponding to the same vma \*/

struct rb\_node rb; /\* pointer to a node in

\* the red-black tree \*/

};

* + - **Explanation of Code:** This structure acts as a link between a vm\_area\_struct (vma) and an anon\_vma. vma points to the virtual memory area. anon\_vma points to the associated anonymous vma structure. same\_vma is a list head used to link multiple anon\_vma\_chain nodes that correspond to the *same* vma. rb is a node for inclusion in a red-black tree, which allows for efficient searching of anon\_vma\_chain nodes associated with a particular anon\_vma.
  + **Diagrams:** Figure 6.15 shows the relationship between vma, anon\_vma, and anon\_vma\_chain (avc). Figure 6.16 shows multiple avcs linked to the same vma.
    - [Image 14]
    - [Image 15]

* + **struct anon\_vma (Listing 6.8):** This structure primarily contains pointers to other structures, notably a red-black tree (rb\_root) that stores the anon\_vma\_chain nodes pointing to this anon\_vma. This tree allows quickly finding all the vmas associated with a given anon\_vma and the virtual address ranges they cover. It also has pointers to a root anon\_vma (for hierarchical organization) and a parent anon\_vma. The root anon\_vma holds a lock for managing the anon\_vma\_chain list.
  + **Code Example (Listing 6.8 - anon\_vma):**

C

struct anon\_vma {

/\* Arrange all the anon vmas corresponding to a vma

\* hierarchically \*/

/\* The root node holds the lock for accessing the chain

\* of anon\_vmas \*/

struct anon\_vma \*root;

struct anon\_vma \*parent;

struct rb\_root\_cached rb\_root;

/\* interval tree of

\* anon\_vmas \*/

};

* + - **Explanation of Code:** This structure represents an anonymous vma. It has pointers to a root and parent anon\_vma for organizing them hierarchically. The rb\_root is the root of a red-black tree that stores anon\_vma\_chain nodes associated with this anon\_vma, enabling efficient reverse mapping.
  + **Revisiting vm\_area\_struct (Listing 6.9):** The vm\_area\_struct (vma) now includes fields to link it with the reverse mapping structures: a list head for anon\_vma\_chain nodes (anon\_vma\_chain) and a pointer to its exclusive anon\_vma (anon\_vma).
  + **Code Example (Listing 6.9 - fields in vm\_area\_struct):**

C

struct vm\_area\_struct {

struct list\_head anon\_vma\_chain;

struct anon\_vma \*anon\_vma;

}

* + - **Explanation of Code:** This snippet shows the relevant fields added to the vm\_area\_struct for reverse mapping. anon\_vma\_chain is a list head for linking multiple anon\_vma\_chain structures that connect this vma to various anon\_vmas (relevant in COW scenarios). anon\_vma points to the primary anon\_vma associated with this vma.
  + **Diagrams:** Figure 6.17 shows the updated relationship between vma, anon\_vma, and avc. Figure 6.18 illustrates an anon\_vma associated with multiple vmas across processes.
    - [Image 16]
    - [Image 17]

* + **Example Walkthrough (fork and COW):** Figures 6.19, 6.20, 6.21, and 6.22 illustrate how the reverse mapping structures change during fork and COW operations, showing how avcs and anon\_vmas are created and linked to track shared and private pages. The red-black tree in the anon\_vma is key for the reverse lookup (finding vmas from a physical page/anon\_vma). Modern kernels optimize space by reusing existing anon\_vma/avc pairs where possible.
  + **Diagrams:**
    - [Image 18]
    - [Image 19]
    - [Image 20]
* **Quick Revision:**
  + Reverse mapping finds vmas/processes mapping to a physical page.
  + Needed when physical pages are swapped or permissions change.
  + Page sharing is common due to fork and COW.
  + vm\_area\_struct (vma) represents a virtual memory region.
  + anon\_vma is an intermediary for reverse mapping of anonymous pages.
  + anon\_vma\_chain links vmas and anon\_vmas to handle many-to-many relationships.
  + anon\_vma uses a red-black tree of anon\_vma\_chain nodes for efficient reverse lookup.
  + fork and COW operations modify these reverse mapping structures.
* **Mnemonics/Memory Tricks:**
  + **rmap:** **R**everse **Map**ping - mapping from physical back to virtual/processes.
  + **anon\_vma:** Think of an "anonymous" link in the virtual memory chain, connecting pages to vmas.
  + **anon\_vma\_chain:** A "chain" that links the anon\_vma and the vma together.

**6.3.2 The MGLRU Algorithm for Page Replacement**

* **Explanation:** The Multi-Generational LRU (MGLRU) algorithm is a sophisticated page replacement algorithm used in Linux, building upon the WS-Clock Second Chance algorithm. It aims to leverage temporal and spatial locality and be efficient for large workloads.
  + **Key Features:**
    - Pages are grouped into generations based on recent access. Recently accessed pages are promoted to the latest generation.
    - Pages from the oldest generations are reclaimed (swapped out) in the background.
    - Pages are "aged" intelligently based on workload.
    - Integrates with the concept of folios.
  + **Key Data Structures:**
    - **struct lruvec (Listing 6.10):** Stores LRU-related information for a NUMA node. Includes a pointer to pglist\_data (zone info), refaults count (for anonymous and file-backed pages, indicating accesses after eviction), and structures for LRU state (lrugen, mm\_state).
    - **Code Example (Listing 6.10 - struct lruvec):**

C

struct lruvec {

/\* contains the physical memory layout of the NUMA

\* node \*/

struct pglist\_data \*pgdat;

/\* Number of refaults \*/

unsigned long refaults [ANON\_AND\_FILE];

/\*LRU state \*/

struct lru\_gen\_struct lrugen;

struct lru\_gen\_mm\_state mm\_state;

};

* + - * **Explanation of Code:** This structure holds LRU vector information for a NUMA node. pgdatpoints to the NUMA node's page list data. refaults is an array tracking page accesses after eviction for anonymous and file-backed pages. lrugen and mm\_state contain detailed state information for the multi-generational LRU algorithm.
    - **struct lru\_gen\_struct (Listing 6.11):** Contains sequence numbers (max\_seq, min\_seq for anon/file), timestamps per generation, and a 3D array of linked lists (lists). The 3D array indexes pages by generation, type (anon/file), and zone. Each list contains struct page entries for pages of the same generation, type, and zone.
    - **Code Example (Listing 6.11 - struct lru\_gen\_struct):**

C

struct lru\_gen\_struct {

/\* sequence numbers \*/

unsigned long max\_seq;

unsigned long min\_seq [ANON\_AND\_FILE];

unsigned long timestamps [MAX\_NR\_GENS];

/\* 3D array of lists \*/

struct list\_head lists [MAX\_NR\_GENS] [ANON\_AND\_FILE] [

MAX\_NR\_ZONES];

};

* + - * **Explanation of Code:** This structure manages the multi-generational aspects of the LRU algorithm. max\_seq is the highest generation number. min\_seq tracks the lowest generation number for anonymous and file-backed pages separately. timestamps can potentially store timestamps for each generation. The lists array is the core data structure, organizing pages into linked lists based on their generation, type (anonymous/file), and memory zone.
    - **struct lru\_gen\_mm\_state (Listing 6.12):** Stores the state of a page walk (traversal to find pages for eviction). Includes the current sequence number (seq), number of page walkers (nr\_walkers), head and tail pointers for a list of mm\_struct (processes being scanned), and an array of Bloom filters to speed up the page walk.
    - **Code Example (Listing 6.12 - struct lru\_gen\_mm\_state):**

C

struct lru\_gen\_mm\_state {

unsigned long seq;

/\* Number of page walkers \*/

int nr\_walkers;

/\* head and tail pointers in the linked list \*/

struct list\_head \*head;

struct list\_head \*tail;

/\* array of Bloom filters \*/

unsigned long \*filters [NR\_BLOOM\_FILTERS];

};

* + - * **Explanation of Code:** This structure tracks the state during a page walk (scanning memory for pages to reclaim). seq indicates the current generation being considered. nr\_walkerscounts threads performing page walks. head and tail pointers manage a list of mm\_structstructures (representing processes) being scanned. filters is an array of Bloom filters used to optimize the scanning process by quickly identifying PMD regions likely to contain young pages.
  + **Page Access Tracking (Listing 6.13):** MGLRU needs to know if a page has been recently accessed. This is done using a "recently accessed bit" in the PTE. Hardware can set this automatically on access. Without hardware support, pages are marked inaccessible, causing soft page faults on access, allowing the kernel to record the access and make the page accessible again. These access bits are periodically cleared ("aged").
    - **Code Example (Listing 6.13 - pte\_mkold):**

C

pte\_t pte\_mkold(pte\_t pte)

{

return pte\_clear\_flags (pte, \_PAGE\_ACCESSED);

}

* + - * **Explanation of Code:** This function demonstrates how the "recently accessed bit" in a page table entry (pte\_t) is cleared to mark the page as "old" during the aging process. It takes a pte\_t as input and returns a modified pte\_t with the \_PAGE\_ACCESSED flag cleared using the pte\_clear\_flags function.

* + **Memory Footprint Compression (Page Reclamation):** Linux dynamically manages the number of pages in memory per zone based on pressure. When pressure is high, pages are evicted or reclaimed to free space.
    - **Passive Process:** The kswapd daemon periodically ages and evicts pages. It calls evict\_folios.
    - **Active Process:** Other kernel functions can directly call evict\_folios.
    - **Page Reclamation vs. Eviction:** Reclamation is a broader term. Eviction is one type (swapping a dirty page to storage). Reclamation can also involve releasing pages from kernel page buffers or dynamically shrinking memory-mapped files.

* + **The Need to Age (Listing 6.14):** Aging updates the generation of pages. The sequence number indicates the generation, lower being older. max\_seq is the youngest generation, min\_seq (per type) is the oldest. The aging algorithm aims to maintain a target number of generations. Incrementing max\_seq effectively ages all pages without modifying individual page data. The algorithm checks if aging is needed based on the number of generations and the proportion of young/old pages. Constants (α,β) and thresholds are used for practical tuning.
    - **Code Example (Listing 6.14 - Aging Logic):**

C

1 if (min\_seq + MIN\_NR\_GENS > max\_seq) return true; /\* Too few generations. Aging is required\*/

2

3

4 /\* There are too many generations. Aging is not required \*/

5 if (min\_seq < max\_seq - MIN\_NR\_GENS) return false;

6 /\* Come here only if min\_seq + MIN\_NR\_GENS <= max\_seq \*/

7

8

9 /\* The aging logica \*/

10 if (young \* MIN\_NR\_GENS > total)

11 return true;

12

13 if (old \* (MIN\_NR\_GENS + 2) < total)

14 return true;

15

16 /\* default\*/

17 return false;

* + - * **Explanation of Code:** This snippet outlines the logic for determining if aging is necessary. It first checks if the number of generations (max\_seq - min\_seq + 1) is below a minimum threshold (MIN\_NR\_GENS + 1); if so, aging is required (line 1). It then checks if the number of generations is above a certain point relative to the minimum; if so, aging is not needed (line 5). If the number of generations is around the minimum, it checks the proportion of "young" pages (belonging to the latest generation) and "old" pages (belonging to a specific older generation) relative to the total number of pages (total) using inequalities involving MIN\_NR\_GENS and a tuning constant (2 for old pages) to decide if aging should occur to maintain a balance between generations (lines 10, 13). Otherwise, aging is not triggered (line 17).

* + **The Aging Process:** Involves walking through page tables of active processes. Bloom filters are used to accelerate this by skipping PMD regions likely containing mostly old pages. If a PMD is not in the Bloom filter (no false negatives), it's skipped. If it is in the filter (possible false positive), its pages are scanned to confirm their recency. For young pages found, the accessed bit is cleared, and their generation is set to max\_seq. PMD addresses predominantly containing young pages are added to the Bloom filter.
  + **Overview of the Eviction Algorithm:** The evict\_folios function is called when physical memory pressure is high. It evicts pages from the oldest generations.
    - **Inputs:** LRU state (struct lruvec) and swappiness factor.
    - **swappiness:** An integer (1-200) that influences the priority of evicting file-backed vs. anonymous pages. Lower swappiness (e.g., 1) prioritizes file-backed eviction; higher (e.g., 200) prioritizes anonymous eviction. Default (0) evicts file-backed.
    - **Code Example (Listing 6.15 - Choosing Eviction Type):**

C

1 if (!swappiness)

2 type = LRU\_GEN\_FILE;

3 else if (min\_seq [LRU\_GEN\_ANON] < min\_seq [LRU\_GEN\_FILE])

4 type = LRU\_GEN\_ANON;

5 else if (swappiness == 1)

6 type = LRU\_GEN\_FILE;

7 else if (swappiness == 200)

8 type = LRU\_GEN\_ANON;

9 else if (!(sc->gfp\_mask & GFP\_IO)) /\* I/O operations are

10 \* involved \*/

11 type = LRU\_GEN\_FILE;

12 else

13 type = get\_type\_to\_scan (lruvec, swappiness, &tier);

* + - * **Explanation of Code:** This snippet shows the logic for deciding whether to prioritize evicting anonymous (LRU\_GEN\_ANON) or file-backed (LRU\_GEN\_FILE) pages. It first checks the swappiness value (line 1). If 0, file-backed are prioritized (line 2). If not, it compares the minimum sequence numbers (oldest generations) for anonymous and file-backed pages; the type with the lower (older) sequence number is prioritized (line 3). It handles edge cases for swappiness being 1 (prioritize file-backed) or 200 (prioritize anonymous) (lines 5, 7). If I/O operations are involved and it's not allowed in the current context (!(sc->gfp\_mask & GFP\_IO)), file-backed pages are preferred as they are already backed by storage (line 9). Otherwise, a more elaborate function get\_type\_to\_scan is called to determine the type based on refault statistics and swappiness (line 12).
    - **Rationale for Eviction Choice:** File-backed pages have a home on disk, making eviction simpler. Anonymous pages need swap space management. Conversely, anonymous pages (stack/heap) might not contain valid data and can have higher temporal locality; file-backed pages (code/data) might be more critical. Linux offers both options, controlled by swappiness. The algorithm also considers which type has an older generation.

* + **Finding the Type of Pages to Evict (Control-Theoretic Approach):** A control-theoretic algorithm balances the eviction of anonymous and file-backed pages. It uses a ctrl\_pos structure with refaults, evictions, and a gain factor. The gain reflects the eviction preference based on swappiness (swappiness for anonymous, 200-swappiness for file-backed). The algorithm compares a "process variable" (file pages) to a "set point" (anonymous pages) to balance their eviction. It uses the normalized refault rate (#refaults / #evictions) and the gain. The eviction probability is inversely proportional to the normalized refault rate and proportional to the gain. Equation 6.1 shows the ideal comparison; Equation 6.2 shows the practical kernel implementation with constants (α=1,β=64) for real workload performance. An additional check prioritizes file eviction if file refaults are low (e.g., < 64).
  + **Scan the Folios:** The algorithm iterates through zones and scans folios of the chosen eviction type with the lowest sequence number. Folios are skipped if pinned, being written back, recently accessed, or involved in race conditions.
  + **Tiers:** To ensure fairness within a generation, folios are grouped into tiers based on their approximate reference count (log(refs+1)). Lower tiers have fewer references and are preferred for eviction. The algorithm performs a tier-wise comparison of normalized refault rates and gain (using Equation 6.2) to decide which folios to evict or promote. Folios in higher tiers deemed "more useful" are given a "second chance" by incrementing their generation. Their reference count can optionally be reset.
  + **Diagram:** Figure 6.23 illustrates the concept of tiers and promoting higher-tier folios to newer generations.
    - [Image 23]

* + **Eviction of a Folio:** Once candidates are identified, folios are evicted. Delays may be inserted. Additional bookkeeping is done, such as scanning nearby virtual addresses for other old pages (spatial locality) and adding PMD addresses of young regions to the Bloom filter. Large folios can be split. This piggybacking of bookkeeping on non-critical operations is a common OS pattern. Eviction involves writing back (if modified), clearing kernel state, freeing buffers, flushing TLB entries, and writing to storage.
  + **The Process of Looking Around:** The lru\_gen\_look\_around function scans nearby pages to identify candidates for aging/eviction, leveraging spatial locality. Called during folio eviction or by kswapd. It ages young pages and increments generations of young entries. Bloom filters optimize this by skipping PMD regions confirmed to be old. PMDs found in the filter are scanned (with potential false positives), and young folios within them can be promoted.
* **Quick Revision:**
  + MGLRU is a multi-generational LRU approximation.
  + Pages are aged into generations based on access.
  + Oldest generations are preferred for eviction.
  + struct lruvec stores LRU state per NUMA node.
  + refaults track pages accessed after eviction.
  + struct lru\_gen\_struct organizes pages by generation, type, and zone in lists.
  + struct lru\_gen\_mm\_state tracks page walk state and uses Bloom filters.
  + Page access is tracked using accessed bits (soft page faults if no hardware support).
  + Aging updates page generations, often by incrementing max\_seq.
  + Page reclamation includes eviction and releasing pages from buffers/mapped files.
  + swappiness influences eviction preference for anonymous vs. file-backed pages.
  + A control-theoretic approach balances eviction based on normalized refault rates and gain.
  + Folios are grouped into tiers based on reference count.
  + Higher-tier folios in the oldest generation are given a second chance (promoted).
  + Eviction involves bookkeeping and writing back modified folios.
  + "Looking around" scans nearby pages for aging/eviction candidates, optimized by Bloom filters.
* **Mnemonics/Memory Tricks:**
  + **MGLRU:** **M**ulti-**G**enerational **LRU** - pages have different "ages" or generations.
  + **Swappiness:** How "swappy" the system is with anonymous pages (higher swappiness = more likely to swap anonymous).
  + **Tiers:** Think of different "levels" or "tiers" of importance or activity within a generation.

**6.4 Kernel Memory Allocation**

This section discusses how the kernel manages its own memory, which differs significantly from user space due to the need for contiguous physical memory in many cases.

* **Explanation:** Kernel memory allocation is distinct from user space allocation because the kernel often requires contiguous physical memory for interacting with hardware (I/O, DMA) and managing internal structures. While virtual memory solves many user-level problems, the kernel's needs sometimes necessitate managing physical memory directly, which can reintroduce issues like external fragmentation (similar to the old base-limit schemes). However, with more regularity in access patterns, more efficient mechanisms than simple heuristics can be used.

**6.4.1 Buddy Allocator**

* **Explanation:** A popular mechanism for managing contiguous memory regions (physical or virtual). It works with blocks of memory whose sizes are powers of 2.
  + **Allocation:** A large free region is repeatedly split into two equal-sized "buddies" until a block just large enough for the requested size is created. The allocated size is rounded up to the nearest power of 2. This implicitly overlays a binary tree structure on the contiguous memory. Allocations happen at the leaf nodes. The allocated memory in a leaf node is always more than 50% of its capacity (otherwise it would have been split further).
  + **Freeing:** When a block is freed, the system checks if its "buddy" (the adjacent block of the same size at the same level in the conceptual tree) is also free. If so, the two buddies are merged into a single larger free block (their parent in the tree). This coalescing process continues up the tree as long as buddies are free.
  + **Diagrams:** Figure 6.24 illustrates the allocation process (splitting). Figure 6.25 shows the freeing process (merging buddies).
    - [Image 24] (Note: The image labels refer to Figure 6.24)
    - [Image 25] (Note: The image labels refer to Figure 6.25)

* + **Implementation:** The buddy tree is represented by an array of linked lists, one for each "order" (level in the tree, where order 0 is the smallest block size - one page). Each list in the free\_area array of struct zone (free\_area[MAX\_ORDER]) holds free blocks (aggregate pages) of that order. There are separate lists for different "migration types" of pages (e.g., movable, unmovable, reclaimable).
  + **Diagram:** Figure 6.26 visualizes zones and their free\_area lists for different migration types.
    - [Image 26]

* + **Parent-Child Relationship:** Parent-child relationships in the buddy tree are implicit and can be calculated using pointer arithmetic, avoiding the need for explicit pointers.
  + **Allocation Code (Listing 6.17):** The rmqueue\_smallest function iterates through the free\_area lists starting from the requested order up to the maximum order. It tries to get a free block (get\_page\_from\_free\_area) of the required size and migration type. If found, it's removed from the list (del\_page\_from\_free\_list) and returned.
  + **Code Example (Listing 6.17 - rmqueue\_smallest snippet):**

C

for (current\_order = order; current\_order < MAX\_ORDER; ++

current\_order) {

area = &(zone->free\_area[current\_order]);

page = get\_page\_from\_free\_area(area, migratetype);

if (!page)

continue;

del\_page\_from\_free\_list(page, zone, current\_order);

...

return page;

}

* + - **Explanation of Code:** This loop attempts to find a free block of memory of at least the requested order (size). It iterates through the free\_area array in the zone, starting from the requested order. get\_page\_from\_free\_area tries to find a free page (block) of the current order and migratetype. If a block is found (page is not NULL), it's removed from the free list using del\_page\_from\_free\_list and returned. If no block of the current order is found, the loop continues to the next higher order (larger block size).
  + **Freeing Code (Listing 6.18):** The \_\_free\_one\_page function attempts to coalesce the freed page with its buddy. It finds the buddy's PFN (find\_buddy\_page\_pfn). If the buddy is free, it's removed from its free list (del\_page\_from\_free\_list), and the two blocks are merged mathematically to get the PFN of the combined parent block. Pointer arithmetic is used to find the starting address of the combined block in the struct page array. The process repeats up the tree. Finally, the merged block's order is set, and it's added to the free list at that order.
  + **Mathematical Explanation of Buddy PFNs:** For buddies with PFNs A and B at order ϕ, A⊕B=2ϕ. The PFN of their parent is A & B=min(A,B).
  + **Code Example (Listing 6.18 - \_\_free\_one\_page snippet):**

C

void \_\_free\_one\_page(struct page \*page, unsigned long pfn,

struct zone \*zone, unsigned int order, ...)

{

while (order < MAX\_ORDER - 1) {

buddy = find\_buddy\_page\_pfn(page, pfn, order, &

buddy\_pfn);

if (!buddy)

goto done\_merging;

del\_page\_from\_free\_list(buddy, zone, order);

....

combined\_pfn = buddy\_pfn & pfn;

page = page + (combined\_pfn - pfn);

pfn = combined\_pfn;

order++;

}

done\_merging:

/\* set the order of the new \*/

set\_buddy\_order(page, order);

add\_to\_free\_list(page, zone, order, migratetype);

}

* + - **Explanation of Code:** This function frees a page (block) and attempts to merge it with its buddy. It loops while the current order is less than the maximum order. find\_buddy\_page\_pfn finds the buddy page and its PFN. If a free buddy is found, it's removed from its free list. combined\_pfn = buddy\_pfn & pfn calculates the PFN of the merged block (parent). page = page + (combined\_pfn - pfn) calculates the starting struct page pointer for the merged block using pointer arithmetic. The PFN and order are updated, and the loop continues, attempting to merge with the next level's buddy. If no free buddy is found, it jumps to done\_merging. Finally, the order of the merged block is set, and it's added to the appropriate free list.
* **Quick Revision:**
  + Buddy allocator manages contiguous memory blocks of power-of-2 sizes.
  + Allocation splits blocks; freeing merges free buddies.
  + Implicit binary tree structure.
  + Implemented using free\_area array of lists in struct zone.
  + Separate lists for different page migration types.
  + Buddy PFNs and parent PFNs can be calculated using XOR and AND operations.
  + Kernel code uses pointer arithmetic to manage blocks and their relationships.
* **Mnemonics/Memory Tricks:**
  + **Buddy Allocator:** Think of memory blocks having a "buddy" of the same size; if both are free, they team up and become a bigger block.

**6.4.2 Slab Allocator**

* **Explanation:** The slab allocator is designed for efficient allocation and deallocation of objects of a *single, fixed size*. It's often used on top of the buddy allocator, which provides large contiguous memory regions to the slab allocator. The slab allocator manages pools of objects of the same type.
  + **Key Concepts:**
    - **Slab:** A contiguous region of memory that stores a set of objects of the same type. It contains space for objects and a freelist to track indices of free objects within the slab.
    - **Slab Cache (kmem\_cache):** A system-wide pool for a specific object type, managing multiple slabs.
  + **Structure:** A slab cache manages three lists of slabs for each NUMA node: full (no free objects), partial(some free objects), and free (all objects free).
  + **Allocation:** When an object is requested, the allocator first checks a per-CPU array cache of recently freed objects for quick reuse. If not found there, it looks for a partial slab. If a free object is found in a partialslab, it's allocated and initialized. If no partial slabs exist, a free slab is taken and an object is allocated from it, moving the slab to the partial list.
  + **Deallocation:** A freed object is typically added to the per-CPU array cache. The state of its encapsulating slab is updated (e.g., a full slab becomes partial). The slab is moved to the appropriate list (partial or free). If a slab becomes empty, it might be returned to the underlying memory system (e.g., buddy allocator).
  + **Diagram:** Figure 6.27 shows the high-level structure of the slab allocator with slabs, slab caches, and per-CPU caches.
    - [Image 27]

* + **Memory Region:** Slabs for a given object type are stored in a contiguous kernel memory region, often allocated by the buddy allocator, to allow using pointer arithmetic to find a slab from an object's address.
  + **Criticism:** The slab allocator can involve frequent movement of slabs between lists and overhead in updating slab states when objects in the per-CPU cache are allocated/freed.
* **Quick Revision:**
  + Slab allocator manages fixed-size objects.
  + Uses slabs (contiguous memory regions for objects) and slab caches (kmem\_cache) to manage pools.
  + Slab caches have full, partial, and free lists per NUMA node.
  + Uses a per-CPU array cache for fast allocation/deallocation.
  + Slabs are stored in a contiguous region.
  + Can have overheads with list management and state updates.
* **Mnemonics/Memory Tricks:**
  + **Slab Allocator:** Think of a "slab" of concrete with pre-defined slots for objects of the same size.

**6.4.3 Slub Allocator**

* **Explanation:** The Slub allocator is a simpler and more optimized version of the slab allocator, emphasizing regularity and relying heavily on pointer arithmetic. It also manages fixed-size objects and uses slabs and slab caches, but with a different organization.
  + **Structure:** Each CPU has a *private slab* assigned to it, stored in its per-CPU region. Unlike the slab allocator, it doesn't have a separate per-CPU array cache; the slab itself is the primary unit for fast allocation.
  + **Allocation:** Objects are allocated directly from the per-CPU slab's freelist. If the per-CPU slab is full, the CPU gets a new free slab. If no free slabs are available, it looks for a partial slab in the slab cache. If a partial slab becomes full, it's removed from the partial list and forgotten (not tracked in a full list).
  + **Deallocation:** When an object is freed, it's returned to its slab's freelist. The inuse count in the slab is decremented. The slab's state is updated implicitly. Pointer arithmetic is used to find the slab from the object's address. If a formerly full slab becomes partially full upon deallocation, it's added to the slab cache's partial list.
  + **Memory Management:** Slub doesn't maintain lists of full or empty slabs. Full slabs are forgotten. Empty slabs are returned to the buddy allocator. New slabs are fetched from the buddy allocator on demand.
  + **Diagram:** Figure 6.28 shows the structure of the Slub allocator.
    - [Image 28]

* + **Advantages:** Simpler design, less state to manage, changes are localized to the slab (no need to update slab state when objects are in per-CPU cache), relies on pointer arithmetic for efficiency. Reduced list management overhead compared to slab allocator.
* **Quick Revision:**
  + Slub is an optimized slab allocator.
  + Relies heavily on pointer arithmetic.
  + Each CPU has a private slab.
  + Doesn't use a separate per-CPU array cache; slab is the unit.
  + Only tracks partial slabs in the slab cache.
  + Full slabs are forgotten; empty slabs are returned to the buddy allocator.
  + Simpler and more memory efficient than the slab allocator.
* **Mnemonics/Memory Tricks:**
  + **Slub Allocator:** A "slub" is a thick, messy lump - think of the simplified, less list-heavy structure compared to the Slab allocator. (A bit counter-intuitive, but might help remember the difference).

**6.5 Summary and Further Reading**

**6.5.1 Summary**

* Memory management is a core OS function with efficiency and correctness concerns.
* Traditional base-limit schemes use contiguous allocation, base/limit registers, and suffer from external fragmentation; they are less common in modern systems due to unpredictable memory needs.
* Virtual memory uses page tables for address translation and faces the challenge of page replacement.
* Page replacement algorithms aim to minimize page faults. Optimal is ideal but impractical. LRU is a widely used approximation, often implemented using access bits and soft page faults or algorithms like WS-Clock and MGLRU.
* FIFO is simple but suffers from Belady's anomaly, where increasing memory can increase page faults.
* The working set is the set of pages a program actively uses. Thrashing occurs when total working sets exceed physical memory, leading to excessive page faults; prevention involves limiting active processes and protecting recently used pages.
* Linux virtual memory is partitioned for user and kernel space. The kernel map includes direct-mapped regions for efficient physical access. Multi-level page tables and PTEs manage virtual-to-physical mappings and permissions. TLBs cache translations, and their management (flushing, ASIDs/PCIDs, lazy mode) is crucial for performance during process switches.
* Physical memory is partitioned into zones to handle NUMA architectures and different memory device types. Zones can be divided into sections. Structures like struct page, folios, struct zone, and struct pglist\_datamanage physical page metadata and organization.
* Kernel memory allocation differs from user space, often requiring contiguous physical memory.
* The Buddy allocator manages contiguous memory blocks of power-of-2 sizes, using splitting for allocation and coalescing for freeing.
* Slab and Slub allocators manage fixed-size objects, often using memory provided by the buddy allocator. Slab uses lists of full, partial, and free slabs. Slub is a simpler, more efficient version using per-CPU slabs and relying more on pointer arithmetic.

**6.5.2 Further Reading**

(This section refers to external sources not provided in the document, so I cannot include specific links or details.)

**Exercises**

Here are potential exam questions based on the chapter content:

1. **Theory:** Explain the concept of fragmentation in memory management. Differentiate clearly between internal and external fragmentation, providing examples of when each occurs.
2. **Theory:** Describe the Stack Distance and its significance in evaluating memory system performance. How does the typical distribution of stack distances (as shown in Figure 6.3) reflect program behavior?
3. **Theory:** Compare and contrast the FIFO and LRU page replacement algorithms. Explain why FIFO suffers from Belady's Anomaly while LRU does not. Use the concept of stack-based algorithms in your explanation.
4. **Theory:** What is "thrashing" in the context of memory management? Explain its causes and the negative impacts on system performance. Describe at least two mechanisms used in modern operating systems (like Linux) to detect and prevent thrashing.
5. **Theory:** Describe the organization of the kernel's virtual memory address space in Linux, highlighting the purpose of key regions such as the direct-mapped region and the memory-mapped I/O region.
6. **Theory:** Explain the purpose of multi-level page tables in modern operating systems. How does the kernel walk the page table to translate a virtual address? Reference the role of the CR3 register and the different levels (PGD, P4D, etc.).
7. **Application:** Given a system using the Base-Limit scheme with a base register value of 1000 and a limit register value of 5000. For the following logical addresses issued by the processor, determine the physical address and whether the memory access is valid or invalid:
   * Logical address: 3000
   * Logical address: 6000
   * Logical address: 0
   * Logical address: 4999
8. **Theory:** Discuss the practical challenges of implementing a pure LRU page replacement algorithm. How do operating systems approximate LRU using mechanisms like accessed bits and soft page faults?
9. **Theory:** Explain the concept of Address Space IDs (ASIDs) or Processor Context IDs (PCIDs) and how they are used to optimize TLB management during process switches. What are the trade-offs involved in making process IDs visible to hardware?
10. **Theory:** Describe the purpose of Zones in Linux physical memory management. Explain at least three different zone types and the characteristics of the memory managed within each zone.
11. **Theory:** Explain the core idea behind reverse mapping in Linux memory management. Describe the roles of vm\_area\_struct (vma), anon\_vma, and anon\_vma\_chain in implementing reverse mapping, particularly in the context of copy-on-write.
12. **Theory:** Compare the Buddy allocator, Slab allocator, and Slub allocator in terms of their primary use cases and key design principles. What are the advantages of Slub over the traditional Slab allocator?