Javan Pahlavan Mentorship

Course Assignment – T(2)

Prepared By:

Mahdi Siamaki

(siamaki.me@gmail.com)

Instructor:

Mr. Mahdi Bahrami

May 2025

Question 1: Understanding "Sectors" in fdisk Output

Question: In fdisk, when using the p (print) command, there is a column labeled "sectors." What does "sector" refer to in this context?

Answer:

In the context of the fdisk command's output (p option), the column labeled "Sectors" indicates the **total number of sectors** that a specific partition occupies on the disk. It represents the *size* or *length* of the partition, measured in units of disk sectors.

- **Disk Sector:** A sector is the smallest physical storage unit on a hard disk drive (HDD) or solid-state drive (SSD). Traditionally, the size of a sector was 512 bytes. Modern drives often use 4096-byte (4K) sectors (Advanced Format), though they may still emulate 512-byte logical sectors for compatibility (512e). fdisk typically operates based on the logical sector size reported by the drive.

- **Calculation:** The size of the partition in bytes can be calculated by multiplying the number shown in the "Sectors" column by the logical sector size of the disk (e.g., Number of Sectors * 512 bytes). fdisk also typically displays the start and end sector numbers for each partition, and the total number of sectors is essentially (End Sector - Start Sector) + 1.

Therefore, the "Sectors" column provides a precise measure of the partition's size based on the fundamental storage units of the disk.

Question 2: Partitioning Recommendations

Question: There are three types of servers mentioned. Please summarize your partitioning recommendations for each: Linux desktop systems, Linux servers for databases/web services with extensive logging, and Linux servers in university labs/staging environments.

Answer:

Partitioning strategies should align with the system's intended use, focusing on manageability, performance, security, and stability. Logical Volume Management (LVM) is generally recommended for server environments due to its flexibility in resizing and managing volumes.

- **A. Linux Desktop Systems:**
  - **Goal:** Simplicity, ease of use, user data separation (optional).
  - **Recommendation:**
    - /boot: (Optional but recommended) Small partition (e.g., 500MB – 1GB) for kernel images and bootloader files. Essential if using full disk encryption or complex RAID/LVM setups for /. Filesystem: ext2/ext4.
    - / (root): Contains the core OS, applications. Can encompass all other directories if simplicity is paramount. Filesystem: ext4 or Btrfs.

        - /home: (Recommended) Separate partition for user data. Allows OS reinstallation without losing user files, enables separate backups, and can have different mount options or quotas. Size depends on user needs. Filesystem: ext4 or Btrfs.
        - swap: Swap partition or swap file. Size often recommended as equal to RAM (for hibernation) or adjusted based on workload (e.g., sqrt(RAM) to 2x RAM).

- **B. Linux Servers (Databases/Web Services with Extensive Logging):**
    - **Goal:** Performance, stability (preventing log/data partitions from filling root), manageability.
    - **Recommendation (LVM Recommended):**
        - /boot: Small partition (e.g., 500MB – 1GB) outside LVM. Filesystem: ext2/ext4.
        - /: (OS Root) Relatively small, containing OS binaries and libraries (e.g., 20–50GB). Filesystem: ext4 or XFS.
        - /var: **Crucial Separate Partition.** Contains logs (/var/log), web server data (/var/www), database files (/var/lib/mysql, /var/lib/pgsql), mail spools, etc. Needs ample space and potentially performance tuning. Prevents runaway logs or databases from crashing the system by filling /. Filesystem: XFS or ext4 (XFS often preferred for large files/directories).

        - /tmp: Separate partition for temporary files. Can improve security (mount with noexec, nosuid) and prevent / from filling up. Size depends on application needs.
        - /opt: (Optional) Separate if installing large third-party applications here.
        - /srv: (Optional) Separate if serving data directly from here as per FHS.
        - Database Data Partition: (Strongly Recommended) A dedicated partition/logical volume mounted specifically for database files (e.g., /data/mysql or directly on /var/lib/mysql). Allows dedicated I/O tuning, specific filesystem choices (XFS), and easier management/backup.
        - swap: Swap partition/logical volume. Size depends on RAM and database tuning (some DBAs prefer minimal swap).

- **C. Linux Servers (University Labs/Staging – Multi-User):**
    - **Goal:** User isolation, storage quotas, manageability, preventing user activity from impacting the OS.
    - **Recommendation (LVM Recommended):**
        - /boot: Small partition (e.g., 500MB – 1GB) outside LVM. Filesystem: ext2/ext4.
        - /: (OS Root) Moderate size (e.g., 20-50GB). Filesystem: ext4 or XFS.

- /home: **Essential Separate Partition.** Contains individual user directories. Needs significant space depending on the number of users and expected storage. Allows for implementing filesystem quotas, easier backups of user data, and prevents user files from filling the root filesystem. Filesystem: ext4 or XFS (ensure quota support is enabled).
- /tmp: Separate partition. Important in multi-user environments for isolation and security (mount with noexec, nodev, nosuid).
- /var: Separate partition for logs and system data.
- swap: Swap partition/logical volume. Size depends on expected usage patterns.

Question 3: Running Applications with Insufficient RAM (Virtual Memory)

Question: How does an operating system run multiple applications when the total available RAM is less than the combined memory requirements? Please summarize the key aspects.

Answer:

Operating systems (OS) handle situations where the total memory demand of running processes exceeds the available physical RAM through a mechanism called **Virtual**

Memory. This creates the illusion that the system has more memory than physically installed. Key aspects include:

- **CPU Scheduling:** The OS rapidly switches the CPU between different processes (multitasking/time-sharing). Each process gets small time slices to execute. This allows multiple applications to make progress concurrently, but it doesn't solve the memory shortage itself.
- **Process Data Loading (Paging):** Processes are divided into fixed-size blocks called **pages**. Physical RAM is also divided into blocks of the same size called **frames**. The OS's Memory Management Unit (MMU) maps the virtual addresses used by a process to physical addresses in RAM.
- **Demand Paging:** Not all pages of a process need to be in RAM simultaneously. Using demand paging, the OS only loads a page from disk into a RAM frame when the process actually tries to access it (i.e., on demand). This requires only the *working set* (currently needed pages) of a process to be resident in RAM.
- **Insufficient RAM Scenario:** When a running process needs a page loaded (due to a page fault) or a new process starts, and there are no free frames in RAM, the OS must make space.
- **Using Disk Space (Paging/Swapping):**
  - The OS designates a portion of the hard disk as **swap space** (can be a dedicated partition or a file).

  - **Paging Out:** To free a RAM frame, the OS selects a page currently in RAM (typically one not recently used, based on a page replacement algorithm like LRU – Least Recently Used) and writes its contents to the swap space on disk.
  - **Paging In:** The freed RAM frame can then be used to load the newly required page from disk (either from the application's executable file or from the swap space if it was previously paged out).
  - **Swapping (Less Common Now):** Historically, swapping referred to moving an *entire* process out to disk. Modern systems primarily use paging, which is more granular and efficient.

- **Memory Management Strategies:** Core strategies include demand paging and page replacement algorithms. These algorithms are crucial for deciding which page(s) to evict from RAM when space is needed, aiming to minimize the performance impact by choosing pages least likely to be needed soon.
- **Partial Loading:** As highlighted by demand paging, **not all pages** of a process must be loaded into RAM for execution. Only the pages containing the code and data currently being accessed are required.

In essence, the OS uses the disk (swap space) as an extension of RAM, moving less-used pages out to disk to make room for actively used pages. This allows more processes to run than physical RAM would otherwise permit, but it comes at a performance cost because disk I/O is significantly slower than RAM access. Excessive paging (known as "thrashing") occurs when the system spends more time moving pages than executing code, leading to severe performance degradation.

Question 4: Translation Lookaside Buffer (TLB)

Question: What is the Translation Lookaside Buffer (TLB), and what role does it play in memory management?

Answer:

The **Translation Lookaside Buffer (TLB)** is a specialized, small, fast hardware cache integrated within the CPU or its Memory Management Unit (MMU). Its primary role is to **speed up the translation of virtual memory addresses to physical memory addresses**.

- **Virtual-to-Physical Address Translation:** In a virtual memory system, processes use virtual addresses. To access actual data in RAM, these virtual addresses must be translated into physical addresses. This translation process typically involves looking up information in data structures called **page tables,** which are stored in main memory (RAM).

- **The Problem:** Accessing page tables stored in RAM for every memory reference would be very slow, as it could potentially require one or more extra memory accesses per desired memory access.
- **The TLB Solution:** The TLB caches recently used virtual-to-physical address translations (page table entries). When the CPU generates a virtual address:
    1. The MMU first checks the TLB for a matching virtual page number.
    2. **TLB Hit:** If the translation is found in the TLB (a "hit"), the corresponding physical frame number is retrieved directly from the TLB very quickly. The physical address is constructed, and memory access proceeds without consulting the main memory page table.
    3. **TLB Miss:** If the translation is *not* found in the TLB (a "miss"), the MMU (often with OS assistance) must perform a full lookup in the page table stored in RAM. Once the physical address is determined, the translation information (virtual page number -> physical frame number) is typically loaded into the TLB, potentially replacing an older entry. The memory access then proceeds.
- **Role in Memory Management:** The TLB is a crucial performance optimization component of virtual memory systems. By caching frequently used address translations, it significantly reduces the overhead associated with the translation process, making virtual memory practical and efficient. A high TLB hit rate is essential for good system performance.

Question 5: Page, Virtual Page, Context Switch & Swap Impact

Question: What are a page, a virtual page, and a context switch? Additionally, how does increasing swap space affect context-switching performance? How does page size influence these effects?

Answer:

- **Page:** A **page** (or physical page, page frame) is a fixed-size block of **physical memory (RAM)**. The OS divides the system's RAM into these frames to manage memory allocation.
- **Virtual Page:** A **virtual page** is a fixed-size block within a process's **virtual address space**. It is the unit of memory that the OS and MMU manage for mapping virtual addresses to physical addresses. The size of a virtual page is typically the same as the size of a physical page frame (e.g., 4KB on many systems).
- **Context Switch:** A **context switch** is the process performed by the OS kernel to switch the CPU from executing one process (or thread) to executing another. This involves:
    1. Saving the complete state (context) of the currently running process (including CPU register values, program counter, stack pointer, process status, and memory management information like page table pointers).

2. Loading the saved state of the incoming process that is scheduled to run next.
3. Updating CPU/MMU structures (like the pointer to the current page table, potentially flushing the TLB). Context switches are essential for multitasking but incur overhead.

- **Swap Space and Context Switching Performance:**
    - **Increasing swap space** itself does *not* directly slow down or speed up the mechanical act of a context switch (saving/loading registers, etc.).
    - However, **heavy utilization of swap space** (meaning the system is actively paging frequently due to memory pressure) *indirectly* and significantly degrades the *effective* performance associated with context switching.
    - When the OS switches to a new process (Process B), if Process B's necessary pages (its working set) are currently swapped out to disk, the context switch completes, but Process B immediately triggers page faults. The system must then perform slow disk I/O operations to page in the required data *after* the context switch, delaying the actual execution of Process B.
    - Therefore, while more swap space allows the system to run more processes or larger processes under memory pressure, it doesn't alleviate the performance penalty of paging. High swap *usage* indicates memory

bottlenecks that make task switching appear slow because processes often can't run immediately after being switched in.

- **Page Size Influence:**
    - **Larger Page Sizes (e.g., 2MB "Huge Pages" vs. 4KB standard):**
        - *TLB Performance*: Larger pages mean fewer pages are needed to cover the same amount of memory. This reduces the size of page tables and increases the likelihood of a TLB hit (one TLB entry maps a larger memory region). This can reduce address translation overhead, potentially improving performance during context switches (less chance of TLB misses immediately after a switch).
        - *Paging I/O*: Paging in or out a larger page involves a larger, potentially more efficient single disk I/O operation, but also takes longer per operation. If only a small part of a large page is needed, paging it in might still be slow.
        - *Internal Fragmentation*: Larger pages increase the potential for wasted memory within the last page allocated to a process segment (internal fragmentation).
        - *Swap Impact*: With larger pages, fewer page faults might occur if the working set fits well within them, potentially reducing swap activity. However, swapping out a large page takes longer. The TLB benefits might slightly mitigate the perceived slowness after a context switch in

a swapping environment compared to smaller pages with more TLB misses.

- **Smaller Page Sizes (e.g., 4KB):**
    - *TLB Performance*: More pages required, potentially leading to more TLB misses and higher translation overhead.
    - *Paging I/O*: Paging individual small pages is faster per page but may require more numerous, less efficient I/O operations overall.
    - *Internal Fragmentation*: Reduced internal fragmentation compared to large pages.
    - *Swap Impact*: May lead to more frequent but smaller paging operations. Context switching into a process requiring many small pages currently swapped out could trigger numerous small page faults.

In summary, page size affects the trade-offs between TLB efficiency, page table size, I/O characteristics during paging, and internal fragmentation, all of which can influence overall system performance, including the effective speed of task switching in memory-constrained environments.

Question 6: Huge Pages

Question: What is a huge page? Please explain its purpose and when it is used.

Answer:

A **huge page** is a memory page that is significantly larger than the standard page size used by the operating system's memory management. While the standard page size is commonly 4KB on architectures like x86-64, huge pages can be much larger, typically 2MB or even 1GB.

- **Purpose:** The primary purpose of using huge pages is to **improve performance** for applications that manage very large amounts of memory. This performance improvement comes mainly from:

    1. **Reduced Memory Management Overhead:** By using larger pages, the total number of pages required to map a large memory region is drastically reduced. This shrinks the size of the page tables that the OS needs to manage in RAM.

    2. **Increased TLB Effectiveness:** Since each TLB entry can now map a much larger region of memory (e.g., 2MB instead of 4KB), the TLB can cover a significantly larger total amount of memory. This greatly increases the TLB hit rate for applications with large working sets, reducing the frequency of slow page table walks and significantly speeding up virtual-to-physical address translation.

- **When Used:** Huge pages are typically used for performance-critical applications that allocate and frequently access large, contiguous (or semi-contiguous) memory regions. Common use cases include:
    - **Database Management Systems (DBMS):** Databases often manage large buffer pools or caches in memory. Using huge pages for these pools can significantly improve database performance.
    - **Java Virtual Machines (JVM):** The Java heap can become very large; configuring the JVM to use huge pages can reduce overhead and improve application performance.
    - **High-Performance Computing (HPC):** Scientific and engineering applications often work with massive datasets in memory.
    - **Virtualization:** Hypervisors can use huge pages for guest VM memory to improve virtualization performance.
- **Considerations:** Huge pages are often statically allocated (pre-allocated at boot or runtime by an administrator) and typically cannot be swapped out to disk (unlike standard pages). This requires careful capacity planning. Applications may also need to be explicitly configured or coded to take advantage of huge pages (e.g., via mmap flags or specific library calls like libhugetlbfs).

Question 7: Memory Fragmentation in RAM

Question: What is memory fragmentation in RAM, and what problems can it cause?

Answer:

Memory fragmentation in RAM refers to a situation where the available free memory is divided into many small, non-contiguous blocks, rather than being available as large, contiguous chunks. Over time, as processes allocate and free memory blocks of varying sizes, the initially contiguous free memory space becomes broken up.

There are two main types of fragmentation:

1. **External Fragmentation:** This occurs when there is enough *total* free memory available in the system to satisfy a request for a large block, but it is not *contiguous*. The free memory is scattered throughout RAM in many small chunks, none of which is large enough on its own to fulfill the allocation request. This is a common issue for memory allocators that handle variable-sized requests directly in the main memory space (e.g., kernel memory allocations via buddy system or slab allocator).

2. **Internal Fragmentation:** This occurs when memory is allocated in fixed-size blocks (like pages or blocks from a slab allocator), and the requested amount is smaller than the allocated block size. The unused space *within* the allocated block is wasted and cannot be used for other allocations. For example, allocating a 4KB

page for only 1KB of data results in 3KB of internal fragmentation within that page. Using larger page sizes (like huge pages) can potentially increase internal fragmentation.

Problems Caused by Fragmentation (primarily External Fragmentation):

- **Allocation Failures:** The most significant problem is that the system may be unable to allocate a large contiguous block of memory, even if the total amount of free RAM is sufficient. This can cause applications or kernel subsystems to fail when requesting memory.
- **Reduced Performance:**
  - Memory allocators may need to expend more effort (CPU time) searching for a suitable free block or attempting to coalesce adjacent small free blocks.
  - In severe cases, the inability to allocate contiguous memory might force the system to resort to more paging/swapping than would otherwise be necessary, indirectly impacting performance.
- **System Instability:** In extreme cases, particularly with kernel memory fragmentation, the inability to allocate critical kernel structures can lead to system slowdowns or even crashes.

Modern operating systems and memory allocators employ various techniques (e.g., paging for user space, buddy system, slab allocation for kernel space) to manage and mitigate fragmentation, but it can still be a concern, especially in long-running systems with complex memory allocation patterns or under heavy memory pressure.

References

1. Silberschatz, A., Galvin, P. B., & Gagne, G. (2018). *Operating System Concepts* (10th ed.). Wiley. (Covers virtual memory, paging, swap, context switching, fragmentation, TLB, page sizes).

2. Bovet, D. P., & Cesati, M. (2005). *Understanding the Linux Kernel* (3rd ed.). O'Reilly Media. (Detailed explanation of Linux kernel internals including memory management).

3. LPIC-1: Linux Professional Institute Certification Study Guide. (Covers exam objectives related to partitioning, filesystems, and basic OS concepts).

4. Linux fdisk manual page (man fdisk). (Definitive source for command options and output interpretation, including the 'sectors' column).

5. Arch Linux Wiki. (2024). *fdisk*. (Provides practical examples and explanations of fdisk usage and output).

6. Wikipedia contributors. (2024). *Disk sector*. Wikipedia, The Free Encyclopedia. (General explanation of disk sectors, including 512-byte and 4K standards).

7. Red Hat Enterprise Linux Documentation. (Various versions). *Storage Administration Guide - Advanced Format (4K) Disk Sectors*. (Explains handling of 4K sector disks).

8. Ubuntu Documentation. (Various versions). *Partitioning*. (Provides recommendations for desktop partitioning schemes).

9. Red Hat Enterprise Linux Documentation. (Various versions). *Performing a standard RHEL installation – Recommended Partitioning Scheme*. (Offers guidance for server partitioning, including LVM and separate mount points like /home, /var, /tmp).

10. Linux Documentation Project (TLDP). *Linux System Administrators' Guide – Filesystems*. (Discusses filesystem choices and partitioning strategies for servers).

11. Red Hat Enterprise Linux Documentation. (Various versions). *Managing storage devices – Adding swap space*. (Discusses swap space configuration and sizing considerations).

12. Crucial.com. (n.d.). *What is Virtual Memory?*. (General overview of virtual memory, paging, and swapping concepts).

13. Wikipedia contributors. (2024). *Context switch*. Wikipedia, The Free Encyclopedia. (Explanation of the context switching process and its overhead).

14. Wikipedia contributors. (2024). *Translation lookaside buffer*. Wikipedia, The Free Encyclopedia. (Detailed explanation of TLB function, hits, misses, and relation to page tables).

15. Tanenbaum, A. S., & Bos, H. (2014). *Modern Operating Systems* (4th ed.). Pearson. (Covers memory management units, TLBs, and virtual memory).

16. Kernel.org Documentation. *HugeTLB Pages*. (Official Linux kernel documentation on the purpose, use, and configuration of huge pages).

17. Wikipedia contributors. (2024). *Memory fragmentation*. Wikipedia, The Free Encyclopedia. (Explanation of internal and external fragmentation and their consequences).

18. Red Hat Enterprise Linux Documentation. (Various versions). *Monitoring and managing system status and performance - Configuring Huge Pages*. (Practical guidance on using huge pages in RHEL).

19. Oracle Documentation. (Various versions). *Database Performance Tuning Guide - Using HugePages*. (Specific guidance on using huge pages for Oracle databases on Linux).

20. LWN.net articles on Linux memory management. (Often contain in-depth discussions on topics like fragmentation and kernel allocators).