Javan Pahlavan Mentorship

Course Assignment – T(4)

Prepared By:

Mahdi Siamaki

(siamaki.me@gmail.com)

Instructor:

Mr. Mahdi Bahrami

April 2025

Question 1:

The command netstat -nat lists all current TCP connections. How can I count the number of connections in each state?

To determine the number of current TCP connections categorized by their respective states using the netstat -nat command, you can construct a pipeline of command-line utilities. The netstat -nat command itself initiates this process, where -n ensures numerical display of addresses and port numbers, -a includes all listening and non-listening sockets, and -t filters for TCP connections exclusively. The output of this command, which lists all TCP connections, is then typically piped to grep '^tcp'. This step filters the stream to ensure only lines representing TCP connections are processed, effectively removing headers or other protocol lines. Subsequently, awk '{print $6}' is used to extract the sixth column from the filtered output, as this column conventionally contains the state of the TCP connection (e.g., ESTABLISHED, LISTEN, TIME_WAIT). To count these states, the extracted list is first passed to sort, which arranges the states alphabetically. This sorting is a prerequisite for the final command, uniq -c, which then collapses the sorted list into unique state entries, prepending each with a count of its occurrences.

A typical command sequence would be:

```
netstat -nat | grep '^tcp' | awk '{print $6}' | sort | uniq -c
```

An example output might look like this:

```
65 ESTABLISHED
      5 LISTEN
      12 TIME_WAIT
```

This output would indicate, for instance, 65 connections in the ESTABLISHED state, 5 in LISTEN, and 12 in TIME_WAIT.

Question 2:

How can we read the stdin, stdout, and stderr streams of a running process using the cat command?

The /proc filesystem in Linux offers a mechanism to inspect running processes, including their standard I/O streams: stdin (standard input), stdout (standard output), and stderr (standard error). For any given process with a Process ID (PID) <pid>, these streams correspond to file descriptors 0, 1, and 2, respectively. These are accessible as symbolic links within the /proc/<pid>/fd/ directory: /proc/<pid>/fd/0 for stdin, /proc/<pid>/fd/1 for stdout, and /proc/<pid>/fd/2 for stderr.

While you can attempt to use the cat command to read these file descriptors, the practical utility and behavior vary. For stdout (/proc/<pid>/fd/1) and stderr (/proc/<pid>/fd/2), if the process is actively writing to these streams and they are directed to a file or pipe, cat might display the content as it's being written or from the current buffer. However, if these streams are connected to a terminal, reading them via /proc might yield no useful data, or could even interfere with the terminal, especially if output is buffered by the process and not yet flushed. Accessing these files typically requires appropriate permissions, such as being the process owner or root.

Reading stdin (/proc/<pid>/fd/0) using cat is generally not a practical way to observe what a process will read. If stdin is a pipe or file, cat would attempt to read from that source, potentially consuming input intended for the target process. If stdin is connected to a terminal, the interaction is complex and usually not what one intends when trying to "spy" on input. For actively monitoring output streams, especially stdout and stderr, tools like strace (to observe write() system calls) or redirecting the process's output to a file and then using tail -f on that file are often more effective and reliable methods.

Question 3:

What is an inode and what is its purpose?

An inode, short for "index node," is a fundamental data structure within Unix-style filesystems such as ext4, XFS, and Btrfs. Its primary purpose is to store all metadata about a file or directory, with the notable exceptions of the filename itself and the actual data content. Each file and directory residing on the filesystem is uniquely represented by an inode. This inode holds a rich set of attributes critical for file management and system operation.

The metadata stored within an inode includes the file type (e.g., regular file, directory, symbolic link, device file), its permissions (read, write, execute for owner, group, and others), the User ID (UID) of the owner, and the Group ID (GID) of the group. It also records the file's size in bytes, several timestamps (last access time or atime, last content modification time or mtime, and last inode metadata change time or ctime), and a link count, which indicates how many hard links or directory entries point to this particular inode. Crucially, the inode contains pointers to the actual data blocks on the disk where the file's content is stored, or for very small files, sometimes the data itself (in an "inline" fashion on some filesystems).

Within a given filesystem, every inode is assigned a unique inode number, which serves as the internal identifier for the file or directory. This separation of the filename (stored in a directory entry) from the inode and its associated data allows for powerful features. For example, hard links are possible, where multiple filenames can refer to the exact same inode and thus the same data and metadata. Renaming or moving a file within the same filesystem typically only involves modifying directory entries to point to the same inode, making these operations very fast. Inodes are also vital for filesystem integrity; utilities like fsck rely on inode information to check and repair inconsistencies. Thus, the inode is the core filesystem object that describes a file or directory's properties and location, independent of how it is named or accessed.

Question 4:

When a new file is created using the touch command, what happens step-by-step in the filesystem, especially in terms of the inode?

When the touch filename command is executed to create a new file (i.e., filename does not already exist), a sequence of operations occurs within the filesystem, with significant involvement of inodes. First, the filesystem must allocate a free inode from its inode table. This newly allocated inode is assigned a unique inode number, which will serve as its identifier.

Once allocated, this inode is populated with essential metadata for the new file. The file type is set to "regular file." Its permissions are determined by the system's default permissions for new files (often 0666), as modified by the current user's umask. The owner (User ID) and group (Group ID) are set to those of the process executing the touch command. The link count for the inode is initialized to 1, as one directory entry will now point to it. Timestamps, specifically the access time (atime), modification time (mtime), and inode change time (ctime), are all set to the current time. Since touch creates an empty file, the file size is set to 0 bytes, and consequently, no data blocks are allocated on the disk, meaning the inode's pointers to data blocks will be null or indicate an absence of data.

Concurrently, a new directory entry must be created in the parent directory where filename is located. This directory entry serves as the bridge between the human-readable filename and the system's internal representation, storing the filename itself alongside the inode number of the newly created inode. The creation of this new entry in the parent directory also means that the parent directory's mtime (modification time, because its content changed) and ctime (change time, because its metadata changed) are updated.

If, however, filename already exists when touch is run, the command's behavior changes. Instead of creating a new file and inode, touch typically updates the atime and mtime of the existing file's inode to the current time. The ctime of this existing inode is also updated to reflect these timestamp modifications. No new inode is allocated, and the link count and other metadata generally remain unchanged. The parent directory's timestamps are usually not affected in this scenario, as its list of entries has not been altered.

Question 5:

What does the stat command show about a file's inode, and how should we interpret that information?

The stat command is a utility that displays detailed status information about a specified file or filesystem, a significant portion of which is directly sourced from or related to the file's inode. For instance, running stat myfile.txt might produce output similar to:

```
  File: myfile.txt
  Size: 1024       Blocks: 8       IO Block: 4096   regular file
Device: 801h/2049d   Inode: 1234567    Links: 1
Access: (0644/-rw-r--r--)  Uid: ( 1000/ username)   Gid: ( 1000/ username)
Access: 2023-10-27 10:00:00.000000000 +0000
Modify: 2023-10-27 09:30:00.000000000 +0000
Change: 2023-10-27 09:30:00.000000000 +0000
 Birth: -
```

Interpreting this output provides deep insight into the file's characteristics, largely defined by its inode. The File field shows the filename, which stat resolves for convenience, though the name itself is stored in a directory entry, not the inode. The Size indicates the file's content size in bytes. Blocks refers to the number of disk blocks allocated for the file's content, a value influenced by the filesystem's IO

Block size. The file type, here "regular file," is another critical piece of metadata stored in the inode.

The Device field specifies the device ID (in hexadecimal and decimal) on which the file resides; this, combined with the Inode number (e.g., 1234567), uniquely identifies the inode and thus the file system-wide. The Inode number itself is the direct unique identifier for this inode within its particular filesystem. The Links field shows the hard link count, indicating how many directory entries point to this single inode; a count of 1 means only myfile.txt refers to it.

Permissions are displayed in both octal (0644) and symbolic (-rw-r--r--) formats, alongside the Uid (User ID and name of the owner) and Gid (Group ID and name of the group), all of which are stored in the inode. The timestamps are also direct inode attributes: Access (atime) is the last time the file's content was read; Modify (mtime) is the last time the content was changed; and Change (ctime) is the last time the inode's metadata (like permissions, ownership, or other timestamps) was modified. The Birth time, or creation time, may be shown if the filesystem supports it. By examining these fields, an administrator gains a comprehensive understanding of a file's fundamental properties, crucial for management, security, and troubleshooting.

Question 6:

When using the mv command, what changes occur at the inode level? Under what conditions does the inode stay the same or change?

The changes occurring at the inode level when using the mv (move) command are fundamentally determined by whether the source and destination paths reside on the same filesystem or on different filesystems.

If the move operation occurs **within the same filesystem** (e.g., renaming a file like mv oldname newname, or moving it to a different directory on the same partition like mv source_file /somedir/dest_file), the inode number of the file itself *does not change*. The mv command essentially manipulates directory entries. A new directory entry is created for the destination name/path, pointing to the original inode. The old directory entry corresponding to the source name/path is then removed. Because the inode's metadata is affected (its link from directory entries is effectively altered, and the parent directories' contents change), the ctime (change time) of the file's inode is updated. However, the actual data blocks on disk are not moved, and the atime (access time) and mtime (modification time) of the file typically remain unchanged. This process is very fast as it only involves metadata updates in the directory structures.

Conversely, if the mv command is used to move a file **between different filesystems** (e.g., mv /fs1/file.txt /fs2/file.txt), the file's inode number *will change*. This is because inodes are unique only within their specific filesystem. In this scenario, mv performs an operation equivalent to a copy followed by a delete. A new inode is allocated on the destination filesystem, and the file's data is physically copied from the source location to the data blocks associated with this new inode. The metadata (permissions, ownership, timestamps) from the original file is typically preserved on the new file as much as possible, subject to user permissions and system policies; the mtime might be preserved, while atime would reflect the copy operation, and ctime would be set for the new inode's creation. Once the copy is successfully completed and verified, the original file (its directory entry and eventually its inode, once the link count drops to zero after removal) is deleted from the source filesystem. This operation is inherently slower as it involves reading and writing the entire file content.

In summary, an inode remains the same for moves within the same filesystem, involving only directory entry changes, and changes for moves across filesystems, which necessitate a full copy and new inode allocation.

Question 7:

When you run ls, how does it retrieve the names of all files and directories in the current directory?

When the ls command is executed, it retrieves the names of files and directories by interacting with the filesystem's directory structure, primarily through a series of system calls. The process begins with ls determining the target directory. If no arguments are provided, it defaults to the current working directory (.); otherwise, it uses the specified path.

Next, ls must open this target directory. This is typically done using the opendir(3) library function, which in turn uses system calls like open(2) or openat(2). Successfully opening a directory, which requires execute (or search) permission on that directory, returns a directory stream pointer. This pointer is an opaque handle used for subsequent operations.

With the directory open, ls iteratively calls the readdir(3) library function. Each call to readdir() reads the next available directory entry from the opened stream. A directory entry, often represented by a struct dirent, fundamentally contains the filename (as d_name) and its associated inode number (d_ino). Some systems also provide the file type (d_type) directly in the directory entry, which can optimize ls by avoiding extra system calls for basic listings. ls collects these names as it reads through the directory entries.

If ls is run with options that require more detailed information, such as -l for a long listing, it needs more than just the name and inode. For each entry, ls will then perform an additional system call, typically stat(2) or lstat(2) (the latter is used to get information about a symbolic link itself rather than the file it points to). These calls use the filename (often constructed from the directory path and the d_name) or the directory file descriptor and d_name (with fstatat(2)) to retrieve a struct stat. This structure contains detailed metadata from the file's inode, such as its size, permissions, owner, timestamps, and link count.

After collecting all necessary information, ls usually sorts the names (alphabetically by default, or by other criteria if specified by options like -t for time or -S for size). Finally, it formats this data and prints it to standard output. Once all entries have been processed, ls calls closedir(3) to close the directory stream and release associated resources. In essence, ls reads the content of a directory file, which is a list of (filename, inode number) pairs, and then optionally uses those inode numbers to fetch further details for each item.

Question 8:

When you run the top command, it shows process states such as running, sleeping, zombie, and stopped. Can you explain what each state means and when it occurs?

The top command, and similar utilities like htop, provide a dynamic real-time view of running processes, including their current state, typically indicated by a single letter in the S (State) column. These states are crucial for understanding system activity and process behavior.

The **R (Running or Runnable)** state signifies that a process is either currently executing instructions on a CPU core or is in the scheduler's run queue, ready and waiting for a CPU core to become available. This state occurs when a process has active computations to perform and is not blocked waiting for any external event or resource.

A very common state is **S (Interruptible Sleep)**. A process in this state is paused, waiting for an event to occur or a resource to become available. It is "interruptible" because it can be awakened by signals. This state is typical for processes waiting for I/O operations (like disk reads/writes or network activity), user input, or those that have been explicitly put to sleep by system calls such as sleep(), select(), poll(), or epoll_wait(), or are waiting on synchronization primitives like mutexes or semaphores.

The **D (Uninterruptible Sleep)** state indicates that a process is sleeping in a way that cannot be interrupted by signals. This usually happens when a process is in the middle of a critical I/O operation, often directly interacting with hardware drivers (especially disk I/O, but sometimes also network I/O like NFS, or other hardware interactions). The process remains in this state until the hardware operation completes. Processes in state D cannot be killed, even with SIGKILL, and a high number of them can indicate I/O system problems, such as a failing disk or overloaded storage.

A process enters the **Z (Zombie or Defunct)** state after it has terminated (i.e., called exit()) but before its parent process has acknowledged its termination by calling a wait() family system call. The process entry is kept in the process table to allow the parent to retrieve the child's exit status. While zombie processes themselves consume very few resources (primarily a slot in the process table), a large accumulation of them can indicate a bug in the parent process, which is failing to "reap" its children. If a parent terminates without reaping its children, these zombies are typically inherited by the init process (PID 1) or a designated subreaper process, which should then clean them up.

Lastly, the **T (Stopped or Traced)** state means the process's execution has been suspended. This can occur if it receives a job control signal like SIGSTOP or SIGTSTP (often initiated by pressing Ctrl+Z in a terminal), SIGTTIN, or SIGTTOU. A stopped process can be resumed with a SIGCONT signal (e.g., using fg or bg shell commands). Alternatively, a process is in a traced state if it is being controlled by a debugger (like gdb or strace) and has hit a breakpoint or is being single-stepped.

Other, less frequently encountered states might include I for idle kernel threads on some newer kernels, or modifiers like < for high-priority, N for low-priority, and s for a session leader. Understanding these states is fundamental for diagnosing system performance and behavior.

Question 9:

When you run the top command, it shows process states such as running, sleeping, zombie, and stopped. Can you explain what each state means and when it occurs?

A high load average on a server suggests that the system is heavily utilized, with many processes either actively consuming CPU resources or waiting in an uninterruptible state (often for I/O). Simultaneously, a high number of context switches indicates that the CPU is frequently shifting its focus from executing one process or thread to another. When both these metrics are elevated, it generally signals a system under significant stress, where the CPU spends a considerable amount of time managing tasks rather than performing productive work. Several underlying conditions could lead to this scenario.

One common cause is **CPU contention**, where numerous processes are in a runnable state, actively competing for limited CPU cores. The scheduler rapidly switches between these processes to provide them with CPU time, leading to high context switches. This often occurs if there are too many active applications, applications with inefficient algorithms, or simply an insufficient number of CPU cores for the given workload. Tools like vmstat (checking the r column for runnable processes and cs for context switches) and top/htop (observing CPU usage per process and many R state processes) can help confirm this.

Another possibility is an **I/O bottleneck**. Processes that frequently block on I/O operations (such as disk reads/writes or network communication) and then become ready again after the I/O completes, cause the scheduler to switch them in and out, increasing context switches. The load average can also be inflated by processes in an uninterruptible sleep (D state) waiting for I/O. This situation can arise from slow storage devices, a saturated network, poorly optimized database queries, or applications performing excessive small I/O operations. iostat (for disk I/O statistics like %util and await), vmstat (for block I/O via bi/bo columns and I/O wait CPU time via wa), and network monitoring tools like iftop or nload are useful for diagnosing I/O issues.

Memory pressure leading to swapping can also manifest as high load and context switches. If the system is low on physical RAM, it will resort to using swap space on disk, which is significantly slower. Processes get paged out to swap and then must be paged back into RAM when needed, causing I/O waits and context switches. This can be due to insufficient RAM for the running applications or memory leaks. vmstat (checking si/so for swap-in/swap-out activity), free -m, and top/htop (for memory and swap usage) are key tools here.

A **high rate of short-lived processes or threads** can contribute as well. If applications are rapidly creating and destroying processes or threads, each such event, along with the associated scheduling, incurs context switches. This might stem from application design choices, such as forking a new process for every minor request instead of utilizing a thread pool. Monitoring tools like pidstat or even execsnoop can help identify high process creation rates.

Furthermore, **inefficient application-level locking or synchronization** mechanisms can cause threads or processes to frequently contend for shared resources. This contention forces threads to sleep and wake up often, resulting in increased context switches. This is usually an application design issue, diagnosable with application-specific profilers or system-wide tools like perf if symbols are available.

Finally, an **interrupt storm**, where a hardware device or its driver generates an excessive number of interrupts, can also lead to high context switches, as each interrupt may preempt the currently running task. This could point to faulty hardware or a buggy driver, and can be investigated by examining /proc/interrupts or the in column in vmstat. Diagnosing the precise cause often involves using a combination of these monitoring tools to pinpoint the resource contention or bottleneck.

Question 10:

Try running atop, htop, and nmon. Where do these tools retrieve their data from?

System monitoring utilities like atop, htop, and nmon provide comprehensive, real-time views of system performance and resource usage primarily by reading and interpreting data from the **/proc virtual filesystem**. This special filesystem in Linux acts as an interface to kernel data structures, exposing a vast amount of information about nearly every aspect of the system.

These tools parse numerous files within /proc to gather their statistics. For **process information**, they access subdirectories like /proc/[pid]/ (where [pid] is a process ID) to read files such as stat (for CPU usage, memory, status), status (for more human-readable details like UID, GID, memory specifics, context switch counts), io (for I/O bytes read/written by the process), and cmdline (command line arguments).

For **CPU utilization and statistics**, they read /proc/stat, which contains overall CPU time spent in various states (user, system, idle, iowait, etc.), as well as counts for context switches, interrupts, and processes created. Information about CPU hardware, like model and core count, is found in /proc/cpuinfo. **Memory information** is largely sourced from /proc/meminfo (total/free memory, buffers, cache, swap details) and /proc/vmstat (detailed virtual memory statistics like page faults and swap activity).

Disk I/O statistics are commonly retrieved from /proc/diskstats, which provides per-device counts of reads/writes, sectors transferred, and time spent on I/O, allowing tools like atop to calculate disk busy percentages. **Network statistics** are gathered from files like /proc/net/dev (for per-interface byte/packet counts and errors), /proc/net/tcp and /proc/net/udp (for active connections and listening sockets), and /proc/net/snmp (for protocol-level statistics). The system's **load average** is read from /proc/loadavg.

While /proc is the predominant source, these tools may also employ other kernel interfaces and system calls. For instance, the sysinfo(2) system call can provide some overlapping system-wide information. Tools like htop might use getpriority(2)/setpriority(2) for managing process nice values and kill(2) for sending signals to processes. atop has an additional capability: it can leverage kernel process accounting, if enabled, by reading from an accounting file (e.g., /var/log/pacct). This allows atop to log historical data and provide more accurate resource consumption figures for processes that may have exited between its regular sampling intervals.

In general, these utilities operate by periodically reading and parsing the relevant text-based data from these /proc files and other sources. They then calculate changes and rates (e.g., CPU usage, I/O throughput) by comparing current readings with previous ones, and finally format and present this consolidated information in their respective user interfaces, often using libraries like ncurses for terminal-based UIs.

References

I. Foundational Operating Systems and Linux Internals Textbooks:

1. **Bovet, D. P., & Cesati, M. (2005).** ***Understanding the Linux Kernel*** **(3rd ed.). O'Reilly Media.**
    - *Relevance*: Comprehensive details on the Linux kernel's internal workings, including process management (states, scheduling), memory management, the virtual filesystem (VFS), inodes, the /proc filesystem, and system calls. Particularly relevant for answers concerning inodes (Q3, Q4, Q5, Q6), process states (Q8), and data sources for monitoring tools (Q2, Q10).
2. **Stevens, W. R., & Rago, S. A. (2013).** ***Advanced Programming in the UNIX Environment*** **(3rd ed.). Addison-Wesley Professional.**
    - *Relevance*: The definitive guide to Unix system programming. It covers file I/O, file metadata (inodes, stat structure), directory operations (opendir, readdir), process control, signals, and inter-process communication. Essential for understanding how commands like ls, mv, touch, stat interact with the filesystem at a low level (Q2, Q3, Q4, Q5, Q6, Q7).

3. **Tanenbaum, A. S., & Bos, H. (2015).** ***Modern Operating Systems*** **(4th ed.). Pearson Education.**
    - *Relevance*: A standard textbook covering general operating system principles such as processes, threads, memory management, file systems, I/O, and deadlocks. Provides the theoretical underpinnings for many of the concepts discussed, like process states (Q8) and filesystem structures (Q3, Q4).
4. **Silberschatz, A., Galvin, P. B., & Gagne, G. (2018).** ***Operating System Concepts*** **(10th ed.). Wiley.**
    - *Relevance*: Another widely-used textbook offering comprehensive coverage of operating system theory, similar in scope to Tanenbaum & Bos. Useful for understanding general concepts related to processes, scheduling, memory, and file systems (Q3, Q8, Q9).

II. Networking Textbooks:

1. **Stevens, W. R. (1994).** ***TCP/IP Illustrated, Volume 1: The Protocols*****. Addison-Wesley Professional.**
    - *Relevance*: The classic reference for understanding the TCP/IP protocol suite in detail, including TCP connection states (ESTABLISHED, LISTEN, TIME_WAIT, etc.). Authoritative for the concepts behind netstat output (Q1).

III. Official Manual Pages and System Documentation:

The primary and most direct authoritative source for the behavior of specific Linux commands and system interfaces are their manual (man) pages and the kernel's own documentation.

1. **The Linux man-pages project (various authors, maintained by Michael Kerrisk).**
   - *How to access*: Via the man command on any Linux system (e.g., man ls, man proc, man stat.h) or online resources like man7.org.
   - *Relevance*:
     - **proc(5):** Describes the /proc filesystem, its structure, and the information contained within files like /proc/stat, /proc/meminfo, /proc/[pid]/stat, etc. This is critical for understanding data sources for tools like top, htop, nmon, and netstat (Q1, Q2, Q8, Q10).
     - **Command man pages:** ls(1), mv(1), touch(1), stat(1), netstat(8), top(1), ps(1), vm stat(8), iostat(1), atop(1), nmon(1). These detail the options, behavior, and output of each command (Q1, Q4, Q5, Q6, Q7, Q8, Q9, Q10).
     - **System call man pages:** open(2), read(2), write(2), stat(2), lstat(2), opendir(3), rea ddir(3), closedir(3), sysinfo(2), kill(2), wait(2). These explain the

underlying kernel interfaces used by higher-level commands and system utilities (Q2, Q4, Q6, Q7, Q8).

- **Header file documentation:** e.g., stat.h for the structure of inode information.

2. **GNU Coreutils Documentation.** (Accessible online at www.gnu.org/software/coreutils/manual/)
   - *Relevance*: The GNU Core Utilities package includes fundamental commands like ls, mv, stat, touch. Their official documentation provides detailed explanations of their behavior and options, complementing the man pages (Q4, Q5, Q6, Q7).
3. **procps-ng Project Documentation and Source Code.** (Project providing ps, top, htop, vmstat, etc.)
   - *Relevance*: The source code and accompanying documentation for these tools are authoritative for their specific implementation details, particularly how they parse /proc and format their output (Q8, Q10).

IV. Kernel Source Code and Documentation:

1. **Linux Kernel Source Code and Documentation.** (Accessible via kernel.org and in the Documentation/ directory of the kernel source tree).
   - *Relevance*: For the deepest understanding, the kernel source code itself is the ultimate authority on how filesystem operations, process scheduling,

and other system behaviors are implemented. The accompanying documentation in the source tree also provides valuable insights. This is especially true for subtle behaviors or advanced topics (relevant to nearly all questions at the most fundamental level).