--From Subrina Jahan ☺

--7th Batch

Ques Solve -> Final (18-19), (17-18), (16-17), MID

2018-19

1.a) Write short notes:

i) System Programming ii) Unix Kernel API iii) Shell Scripting iv) File Attributes.

i) System Programming

System programming is the process of designing and developing system software that provides services to the computer hardware or operating system. It includes programming in languages like C and focuses on creating low-level tools such as compilers, device drivers, and operating systems.

ii) Unix Kernel API

The Unix Kernel API provides system calls that allow user-level programs to interact with the kernel. Examples include open(), read(), write(), fork(), and exec(), which perform tasks like file management, process control, and memory handling.

iii) Shell Scripting

Shell scripting is the process of writing scripts using a Unix shell (e.g., Bash) to automate tasks. It simplifies repetitive tasks such as file backup, automation, and system monitoring using conditional statements, loops, and system commands.

iv) File Attributes

File attributes in Unix include metadata about a file such as permissions (rwx), owner, group, timestamp (modification, access), inode number, size, etc. Use ls -l or stat to view file attributes.

b) A file whose file descriptor is fd contains the following sequence of bytes: 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5. The following system calls are made:

Iseek(fd, 3, SEEK_SET); read(fd, &buffer, 4);

where the Iseek call makes a seek to byte 3 of the file. What does buffer contain after the read has completed?

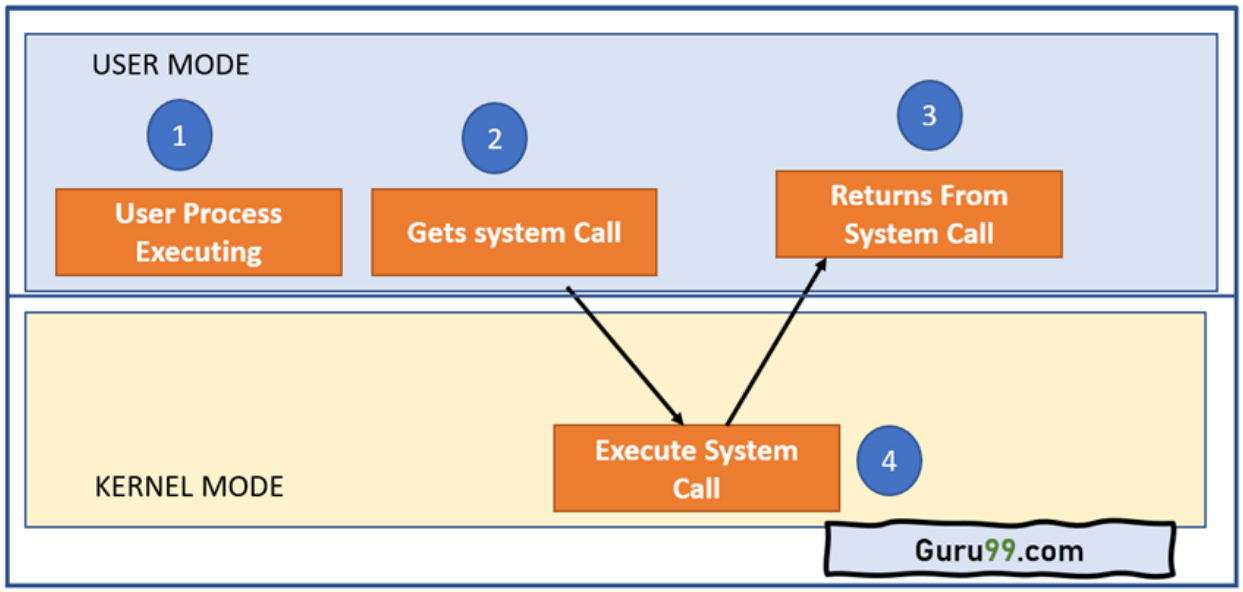
The contents of the file descriptor fd are: 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5.

lseek(fd, 3, SEEK_SET);: This system call sets the file offset to byte 3 from the beginning of the file (SEEK_SET). So, the file pointer is moved to byte 3, which is the number 1 in the sequence.

read(fd, &buffer, 4);: This system call reads 4 bytes from the current file offset (byte 3) and stores them in the buffer. The buffer will contain the bytes starting from the current offset: 1, 5, 9, 2.

So, after the read has completed, the buffer will contain the sequence of bytes: 1, 5, 9, 2.

c) Depict the control flow of system calls. Do you agree/disagree that the system calls are time consuming? Give your statement. How do you change file permissions in Unix Explain with example.



Yes, system calls are time-consuming. This is because:

1. Mode Switching Overhead: The transition between user mode and kernel mode requires extra CPU cycles.
2. Context Switching: The OS may need to save and restore process states, adding latency.
3. Hardware Interaction: Some system calls require disk or network access, which is slower than CPU operations.

However, optimizations like system call batching and caching can reduce their impact.

File permissions in Unix determine who can read, write, or execute a file. The chmod command is used to modify these permissions.

Syntax:
chmod [permissions] [file]

Each permission is represented as a number. For example:

- Read (r) = 4
- Write (w) = 2
- Execute (x) = 1

Now,

chmod 755 myfile.txt

Here,

7 (4+2+1) → Owner has read, write, execute permissions.
5 (4+0+1) → Group has read, execute permissions.
5 (4+0+1) → Others have read, execute permissions.

2. a) cd [<dir>] means that change pwd to HOME directory, or <dir> if it is supplied. However, "Changing directories" and "being in a directory" are imprecise phrases. When you cd to a directory named dir, you may think of yourself as being "in dir", but this is not true. What is the true meaning for cd [<dir>] in order to resolve the relative pathnames?

The cd [<dir>] command changes the current working directory to the specified directory (if provided) or to the user's home directory if no directory is given. When we use cd to change to a directory named <dir>, we are not physically moving into that directory. Instead, the shell updates its internal record of the current working directory to the specified directory. The shell then uses this updated information to resolve relative pathnames.

Relative pathnames are pathnames that do not start with the root directory (/) but instead are based on the current working directory.

For example:

1. Before changing directory

   ```
   pwd         # Output: /home/user
   ls test.txt  # Resolves to: /home/user/test.txt
   ```

2. After cd Document

```
pwd         # Output: /home/user/Document
ls test.txt  # Resolves to: /home/user/Document/ test.txt
```

b) In multi-user operating system, the operating system must protect users from each other and protect itself from users. However, while providing an operating environment for all users, the operating system creates this illusion (permission for read, write, execution, etc.) by creating data paths between user processes and devices and files. How does UNIX create this illusion?

In a multi-user operating system like UNIX, the system must ensure that users do not interfere with each other's processes, data, or system resources. UNIX creates the illusion of a private, secure environment for each user through the following mechanisms:

- UNIX assigns read (r), write (w), and execute (x) permissions to the owner, group, and others. Example: drwxr-x--x means the owner has full access, Group can read & execute, Others can execute.
- Each user has a User ID (UID) and belongs to a Group ID (GID). The system ensures users cannot modify others' files unless given permission.
- Each process runs runs under the UID of the user who owns it, preventing access to another user's processes. Unauthorized access is blocked by the kernel's access control mechanisms.
- When accessing files or devices, the kernel verifies if the user has the all necessary permissions. Unauthorized actions are denied.
- UNIX treats everything as a file. Restricted device file access prevents unauthorized users from manipulating hardware.

c) Each user has a username and a number, the UID, why? Wouldn't it be simpler to record the username of the user as the owner of a file? Why not have a single identifier for each user?

Using a UID (User ID) instead of a username for file ownership and system processes provides several key advantages:

1. Numeric UIDs are faster to process and compare than strings (usernames). The system can quickly check permissions and ownership using integer comparisons rather than string lookups.
2. A user may change their username, but their UID remains constant. This ensures consistent file ownership even if a username is modified.
3. Some systems might allow duplicate usernames, leading to confusion in ownership. UIDs are unique, preventing ambiguity in identifying users.
4. In distributed systems, UIDs are used across multiple machines to maintain file ownership. If only usernames were used, conflicts could arise if different users on separate systems had the same name.

A single identifier (like just a username) could lead to security risks, conflicts, and inefficiencies. Using both a username and UID allows flexibility while ensuring system integrity and security.

3. a. A system call is like a conventional function call in that it causes a jump to a subroutine followed by a return to the caller. But it is significantly different. Distinguish between system call and function call based on the following keywords: privileged/kernel mode, trap instruction, system dependent. (mark 5)

System calls and function calls both transfer control to another code block, but they differ in how they interact with the operating system. System calls allow user programs to request services from the kernel, while function calls work within the program itself.

- **Privileged/Kernel Mode:**
  - A system call runs in privileged mode (kernel mode). This allows it to access hardware or critical system functions like reading a file or allocating memory.
  - A function call runs in user mode, which has limited access and cannot directly control hardware or the operating system.
- **Trap Instruction:**
  - A system call uses a trap instruction to switch from user mode to kernel mode. This is a special instruction that tells the CPU to enter the operating system.
  - A function call does not use any trap. It just jumps to another part of the program within the same mode (user mode).
- **System Dependent:**
  - A system call is system dependent. Its behavior, number, and use depend on the specific operating system (e.g., Linux, Windows).
  - A function call is mostly system independent. It works the same way in most systems if the programming language is the same.

b. Unix files have a full set of permission bits, including a set-user-ID bit and a set- group-ID bit. If you turn on the set-group-ID bit for a directory, does it have any effect? If so, what and why? If not, could you think of some use for this bit? (mark 5)

Yes, turning on the **set-group-ID (SGID)** bit for a **directory** does have an effect in Unix. When the **SGID bit is set on a directory**, all **new files and subdirectories** created inside it will automatically inherit the **group ownership** of the directory, **not** the primary group of the user who created the file.

This behavior is helpful for **collaborative work** in shared directories. Without the SGID bit, files created by different users may belong to different groups, causing permission issues.

With SGID set:

- All new files and subdirectories **inherit the directory's group**, keeping permissions consistent.
- It **reduces manual effort** (no need to run chgrp or adjust group settings repeatedly).
- It ensures **smooth automation**, where scripts or tools can run without group mismatch errors.

Example:

Assume there is a directory /shared/project owned by group devs. To enable SGID:

chmod g+s /shared/project

Now, any file created inside will automatically belong to group devs, regardless of which user creates it (as long as they are members of devs).

c. Analyze the following three commands:

i) ps-ax | grep cron

ii) ps-ax | tee processes.txt | more

i) ps -ax | grep cron

Purpose: Lists all running processes and filters out only those containing the word "cron".

- This command uses a pipe (|) to pass the output of ps -ax as input to next command grep.
- ps -ax lists all running processes in the system.
- grep cron filters the list to show only lines containing the word "cron.
- **Shell Feature Used:** Piping and filtering (using | and grep).

ii) ps -ax | tee processes.txt | more

Purpose: Displays all running processes, saves them to a file (processes.txt), and shows them page by page

- ps -ax lists all current processes.
- tee processes.txt writes the output to the file processes.txt and passes it along to the next command (dual output).
- more displays the result page-by-page, useful for viewing long outputs interactively.
- This command view and save the process list at the same time.
- **Shell Features Used:** Piping, tee duplicates the output stream to a file and standard output simultaneously, pagination with more

4. a. The kernel had to locate a free inode and free disk blocks when it created a new file. How does the kernel know which blocks are free? How does the kernel know which inodes are free? What method does the file system on your machine use to keep track of unused blocks and inodes? (9mark)

When a file is created in a UNIX-like system, the **kernel must allocate a free inode** (to store metadata) and **free disk blocks** (to store the file content). To do this efficiently, the kernel uses **internal data structures** and the help of the file system.

1. How Does the Kernel Know Which Disk Blocks Are Free?

- The kernel uses a **block allocation bitmap** maintained by the file system.
- In a **bitmap**, each bit represents a block:
    - **0** = block is **free**
    - **1** = block is **in use**
- When the file system needs a new block it scans the bitmap to find the first 0 bit, allocates that block, and convert the bit to 1.
-
- The bitmap is kept in memory for efficiency and updated when blocks are allocated or freed
- Some file systems (like older ext2) also used **free block lists**, but this was slower and less space-efficient.

2. How Does the Kernel Know Which Inodes Are Free?

- The file system also maintains an **inode bitmap**:
    - Each bit corresponds to an inode.
    - **0** = inode is free
    - **1** = inode is in use.
- The kernel checks this bitmap when creating a new file or directory to find a free inode.

3. What Method Does the File System on Your Machine Use?

Most modern Linux file systems use **bitmaps** for both inodes and blocks.

Examples:

- **ext3/ext4 (common Linux file systems):**
    - **Block Bitmap** for tracking free blocks.
    - **Inode Bitmap** for tracking free inodes.

- These keep track the usage status of blocks and inodes.

b. A directory is just a node in a set of linked nodes. Using the pwd command you can simply know the current directory. Based on the functionalities of pwd answer the following:

i) What repetitive steps are performed to compute its current directory?

ii) How do we know when we read the top of the tree?

iii) How do we print the directory names in the correct order?

The pwd (print working directory) command shows the **absolute pathname** of the current working directory. Internally, the system performs the following steps to compute and print this path:

i) What repetitive steps are performed to compute its current directory?

The system starts at the current directory and repeatedly:

1. Reads the inode number of the current directory.
2. Moves to the parent directory (..).
3. Scans entries in the parent directory to match the inode of the current directory.
4. Stores the matched name (the actual name of the current directory).

This process continues recursively up the tree until the root is reached.

ii) How do we know when we read the top of the tree?

The root directory (/) is the only directory whose inode number is equal to its parent's inode. When the system reads . and .. in a directory and finds they have the same inode number, it has reached the root.

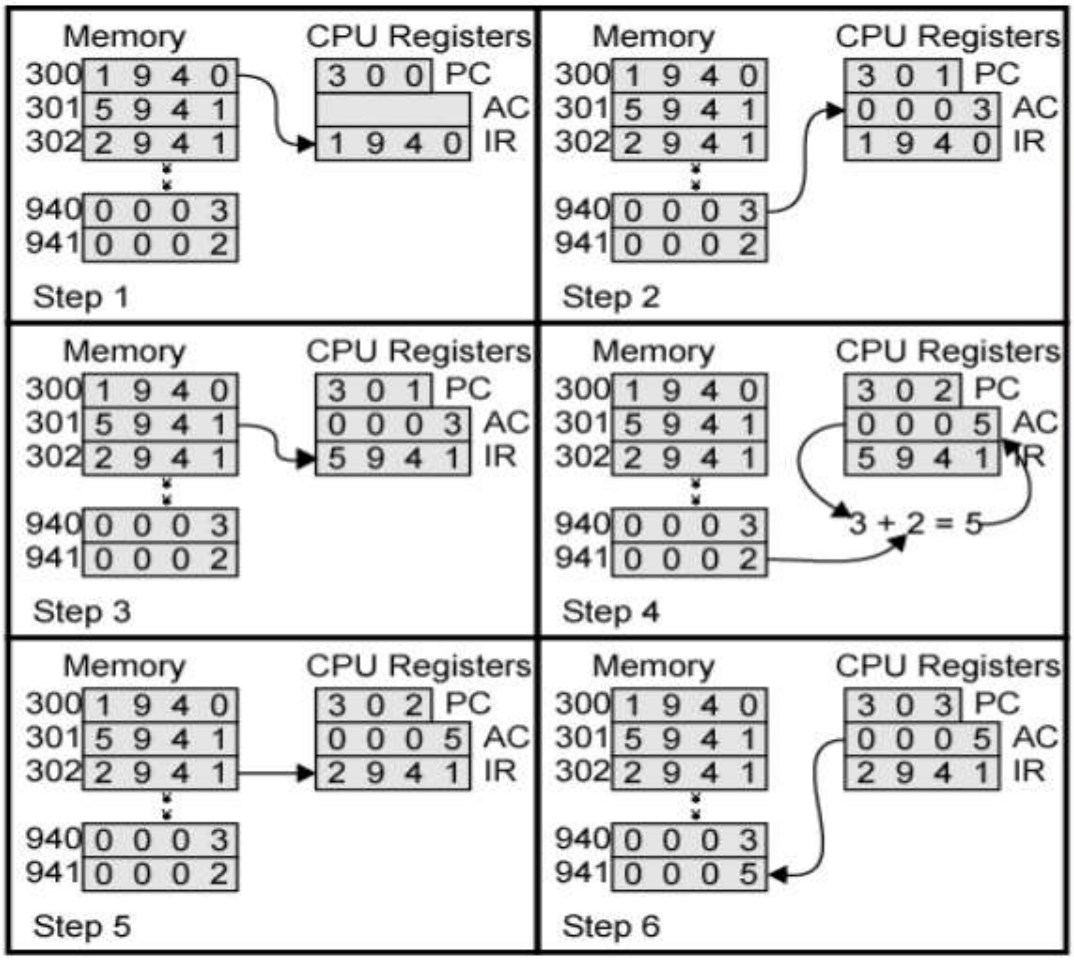
iii) How do we print the directory names in the correct order?

Since names are stored from child to parent during the bottom up traversal, the system must:

1. Reverse the list of directory names.
2. Adds slashes (/) to form the full path.

For example, if it stored: ["project", "user", "home"], it prints /home/user/project.

5.a) Suppose the values 3 and 2 are stored in memory locations 940 and 941 respectively. Now, explain the program execution with figures to store the sum of the two values in the memory location 941.



- **Step-1:** The PC contains 300, the address of the first instruction. This instruction (the value 1940 in hexadecimal) is loaded into the instruction register IR, and the PC is incremented. Note that this process involves the use of a memory address register and a memory buffer register. For simplicity, these intermediate registers are ignored
- **Step-2:** The first 4 bits (first hexadecimal digit) in the IR indicate that the AC is to be loaded. The remaining 12 bits (three hexadecimal digits) specify the address (940) from which data are to be loaded
- **Step-3:** The next instruction (5941) is fetched from location 301, and the PC is incremented
- **Step-4:** The old contents of the AC and the contents of location 941 are added, and the result is stored in the AC.
- **Step-5:** The next instruction (2941) is fetched from location 302, and the PC is incremented
- **Step-6:** The contents of the AC are stored in location 941

b) In UNIX, there is a single file hierarchy and every accessible file is in this single file hierarchy, no matter how many disks are attached. Similar to windows, there is no such thing as the "C"

drive" or "E" drive" in UNIX. If any new file system arrive then how UNIX adapt itself? Give example(s). (4mark)

When a new file system is added, UNIX **does not assign it a new drive letter** like Windows. Instead, it **mounts** the new file system **into the existing directory structure**.

How UNIX Adapts to New Filesystems are given below:

1. The new file system is attached to a specific directory, known as a **mount point**.
2. This mount point could be any empty directory (e.g., /mnt/usb)
3. Once mounted, files from the new file system appear under that directory as if they were part of the original hierarchy.

This allows users and programs to access files on the new storage device by navigating the standard directory tree, without needing to know the physical location of the data.

Example:

Suppose a USB drive is inserted:

mount /dev/sdb1 /mnt/usb

- /dev/sdb1 is the device representing the USB.
- /mnt/usb becomes the **access point** for all files on that USB.
- Now, accessing /mnt/usb/file.txt gives you access to a file on the new file system.

c) How does 1s -1 work? Briefly explain the data flow in who command with figures. (5ark)

The ls -l command in UNIX lists directory contents in long listing format. Here's the step-by-step process:

- The user runs ls -l in the terminal. The shell interprets the command and starts the ls program with the -l option.
- ls opens the specified directory and reads its contents using system calls like opendir() and readdir().
- For each file/subdirectory, ls collects metadata using stat(). This includes:
  - File type and permissions
  - Owner, group
  - Size
  - Modification time
- The collected information is formatted according to the long listing format.
- Finally, the formatted data is printed line by line to the terminal.

Output Format of ls -l:
drwxr-xr-x 2 user group 4096 Mar 10 12:30 mydir

Explanation:

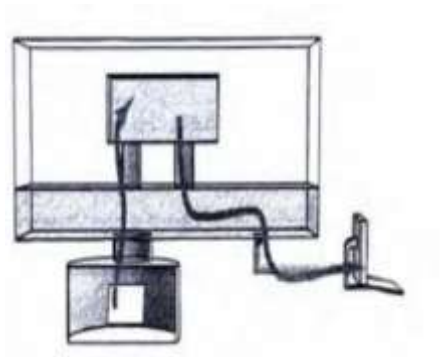
- **File Type:** d (directory) or - (file).
- **Permissions:** rwxr-xr-x (owner, group, others).
- **Links:** Number of hard links.
- **Owner & Group:** Shows who owns the file.
- **Size:** In bytes.
- **Modification Time:** Last modified date.
- **Filename:** Name of the file/directory.

The who command shows the list of currently logged-in users.

- It reads records from the file /var/run/utmp. Each record contains info about one user session (username, terminal, login time).
- These records are stored as structures in an array.

Steps:

- Open the utmp file
- Loop through each record
- Read the user's login information
- Display the data in the terminal
- Close the file



6. a. The stat system call is passed the name of a file and a pointer to a stritet and fills the struct with information about the file. Explain, using the directory, inode, and data model, how stat works? (3)

The stat() system call retrieves metadata about a file. It is passed the name of a file and a pointer to a structure (struct stat) where file information will be stored. Here's how it works internally:

1. Search Directory for Filename

- When stat("file.txt", &buf) is called, the OS **searches the directory** where the file is located.
- Each directory is a **list of entries**, mapping **filenames** to **inode numbers**.

2. Get the Inode

- Once the filename is matched, the system finds the corresponding **inode number**.
- The inode is a structure that stores metadata such as:
  - File type and permissions
  - Owner and group
  - File size
  - Timestamps (creation, access, modification)
  - Pointers to the actual data blocks

3. Step 3: Fill the stat Structure

- stat() reads the inode and **fills the struct stat** with this metadata.
- stat() does **not** read the actual file contents—only metadata from the inode.

b. How can we design a multithreaded program to count and print the total number of words in two files? Give two use case scenarios for clarification. (6mark)

Creating Threads:

- Create two threads, one for each file using pthread_create().

Counting Words:

- Each thread opens its assigned file and reads the content.
- Words are counted by checking character sequences (e.g., between white spaces and alphanumeric characters).

Handling Shared Data:

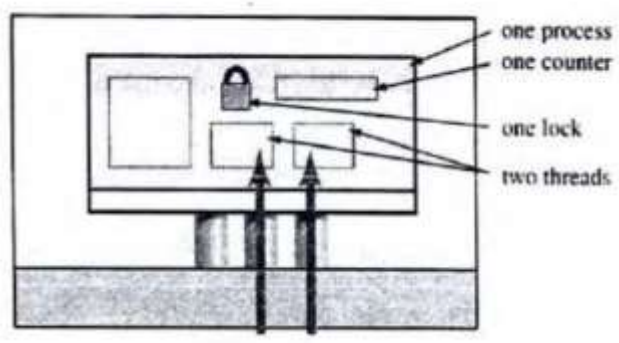
- A shared global variable total_words is used to store the total count.
- To avoid race conditions, use a mutex (pthread_mutex_lock/unlock) when updating the shared count.

```
void *count_words(void *filename) {
  // Open file and read character by character
  // On word detection:
  pthread_mutex_lock(&counter_lock);
```

```
    total_words++;
    pthread_mutex_unlock(&counter_lock);
    return NULL;
}
```

Final Output:

- The main thread waits for both threads using pthread_join().
- After both threads complete, the total word count is printed.



1. Large File Processing on multicore CPU

Problem: Sequentially processing two large files might be slow.

Solution: Two threads process files parallelly on different CPU cores, improving performance.

2. Responsive GUI Application

Problem: Word counting on the main thread can freeze the UI.

Solution: Use background threads for counting to keep the UI responsive; protect shared data with a mutex.

c. Write short notes on the following:

i) pthread_create, pthread_join

ii) pthread_mutex_lock, pthread_mutex_unlock (5mark)

i) pthread_create, pthread_join:

pthread_create:

- This function is used to create a new thread in a program.
- It takes four arguments: a thread identifier, optional thread attributes (or NULL), a function pointer to the thread's start routine, and an argument to pass to that function.
- It starts executing the specified function concurrently in a new thread.

pthread_join:

- This function is used to wait for a specific thread to finish execution.
- It blocks the calling thread until the target thread terminates.
- It can also retrieve the return value from the thread, allowing proper synchronization and cleanup.

ii) pthread_mutex_lock, pthread_mutex_unlock:

pthread_mutex_lock:

- This function is used to acquire a mutex lock, ensuring that only one thread can access a shared resource at a time.
- If the mutex is already locked, the calling thread blocks until it becomes available.

pthread_mutex_unlock:

- This function releases a mutex lock previously acquired.
- It allows other threads to acquire the mutex and continue.
- It ensures mutual exclusion and helps in preventing deadlocks in critical sections.

2017-18

1. a) Define the following terms: i) Users and Groups ii) System calls versus system libraries. (4mar)

i) Users and Groups

- Users:

A user in UNIX/Linux is an individual who interacts with the system and has a unique User ID. Each user has specific permissions and ownership over files and processes.

Example: root (superuser), Name (regular user).

- Groups:

A group is a collection of users who share common permissions and privileges. Each group has a unique Group ID (GID). Groups help in access control by allowing multiple users to access files and resources without setting permissions individually.

Example: A directory /shared/project owned by group devs

ii) System Calls vs. System Libraries

System Calls	System Libraries
Low-level interface between user programs and the operating system.	High-level wrapper functions that make system calls easier to use.
Directly operates in the kernel mode.	Operates in user mode and calls system calls when needed.
Slower because they involve context switching to the kernel.	Faster as they optimize multiple system calls.
Perform low-level OS services (e.g., read, write)	Provide high-level API using system calls internally
read(), write(), fork(), open()	printf(), scanf(), fopen(), malloc()

c) A very significant issue for system programming is the handle of I/O devices for accessing the abstraction. Differentiate the access criteria of I/O devices for UNIX and C. (5mark)

In UNIX, devices are treated as files, and access is handled via system calls. In C, I/O devices are typically accessed using higher-level library functions that abstract the underlying system calls.

UNIX I/O (System Calls)	C I/O (Standard Library)
Provides low-level access to devices and files.	Provides high-level, user-friendly file access.
Uses system calls like `open()`, `read()`, `write()`.	Uses standard functions like `fopen()`, `fread()`, `fwrite()`.
Works with file descriptors (integers).	Works with `FILE*` pointers.
No built-in buffering; programmer must manage data flow.	Provides internal buffering automatically.

Suitable for device files and raw I/O access.	Suitable for text/data file operations.
Offers more control, used in system-level programming.	Easier to use, preferred in application-level coding.

2. a) Analyze the following commands and describe which shell feature(s) these are used:
i) ls mydir > outfilel &
ii) vi *
iii) ls mydir> outfilel & date outfile2 & (4mark)

i) ls mydir > outfile1 &

ls mydir → Lists the contents of mydir.

> outfile1 → Redirects output to a file (outfile1).

& → Runs the command in the background, allowing the user to continue using the shell without waiting for the command to finish.

ls mydir > outfile1 & → Saves output to a file and runs the command asynchronously.

ii) vi *

* (Wildcard) → Expands to match all files in the current directory.

vi → Opens all matched files in the vi editor.

vi * → matches all files in the current directory and opens them in `vi`.

iii) ls mydir > outfile1 & date > outfile2 &

ls mydir > outfile1 & → Lists mydir contents, redirects output to outfile1, and runs in the background.

date > outfile2 & → Gets the current date/time, redirects output to outfile2, and runs in the background.

ls mydir > outfile1 & date > outfile2 & → Executes both commands concurrently, saving their outputs to separate files.

b) Jumping off the end of a file lets you set the current position to locations after the end of the file. For example, lseek (fd, 100, SEEK_END) moves the position 100 bytes past the end of a file. Now. Answer the following questions with an explanation.
i) What happens if you read data after the end of a file?
ii) What happens if you write data after the end of a file?

i) if we read data after the end of a file:

- Attempting to read beyond the end of a file results in an End of File (EOF) condition.
- The read() system call returns 0 bytes, indicating that there is no more data to read.
- No error occurs, but the read operation does not retrieve any data.
- The file size remains unchanged.

ii) if we write data after the end of a file:

- Writing beyond the end of a file causes the file size to increase.
- The gap between the old EOF and the new data is filled with null bytes (\0), creating a sparse file.
- Sparse files do not physically store the empty space, saving disk space.
- The new data is written at the specified position, and future reads return actual stored data.

Example:

```
lseek(fd, 100, SEEK_END);

write(fd, "A", 1);
```

This creates a hole of 100 bytes filled with zeroes before writing 'A' at the new end. The file size increases by 101 bytes, but only 1 byte contains actual data.

3. a) Consider the race condition when two people are login to the unix system at the same time. The important facts are that Unix is a time-sharing system and this procedure requires two separate steps.

i) What system calls are involved?
ii) Why is this situation created?
iii) How to solve this issue?

In a multi-user, time-sharing Unix system, race conditions can occur when two users log in concurrently and the system accesses or modifies shared resources in a non-atomic way.

i) What system calls are involved?

Several system calls are involved in the login process:

- fork() → Creates a new process for each login session.
- exec() → Replaces the current process with a new one.
- open() – Opens the password and user database files
- fread() / read() → Reads user credentials from these files.
- fwrite()/write() → Writes logs or updates session records.
- setuid() → Sets the user ID to match the logged-in user.

ii) Why is this situation created?

- Two login processes tries to access or modify the same file/resource concurrently.
- Time-sharing allows concurrent execution, increasing race risk
- Without synchronization, processes may read/write inconsistent or partial data

iii) How to solve this issue?

To prevent race conditions during login:

- Use **file locking mechanisms** (e.g., flock() or fcntl()) to ensure only one process can access critical files at a time.
- Implement **atomic operations** where possible.
- Design login processes to be **thread-safe** and avoid shared writes without coordination
- Use synchronization tools (mutexes, semaphores) to coordinate processes.

b) The sample listing of ls -l shows directories with permission mode drwxr-xr-x, typically. From left to right, this patterns means the item is a directory, That the owner of the file may read, write, and execute the directory, and that group members and anyone else may read or execute the directory.

1) What does the ”execute” permission mean for a directory?
2) What does the “execute bit” mean for a directory? Why is it useful?

1) “execute” permission

- The execute (x) permission on a directory allows access inside the directory (e.g., using cd mydir).
- If execute is missing, users cannot enter the directory, even if they have read permission.
- If only execute (--x) is set, users can access known files but cannot list directory contents.
- Example:

```
chmod -x mydir  # Removes execute permission
cd mydir        # Permission denied
```

2) execute bit

- The execute bit (x) allows a user to traverse the directory without listing its contents.
- If a user knows a filename inside, they can access it, but cannot see other files without read permission.
- Example:

```
chmod 111 mydir  # Only execute permission
cd mydir         # Allowed
ls mydir         # Permission denied (no read permission)
cat mydir/file.txt # Allowed if file.txt has read permission
```

Execute bit is useful because:

- Prevents unauthorized users from listing directory contents but allows access to specific files.
- Allows users to interact with files inside without exposing the entire directory.

4.a. In Unix file system, there is no limit to tree structure. All files in this system reside in this structure, and includes many programs for working with the objects of this structure. (6 mark)

i) How does a file know which directory it is in?

ii) How does pwd figure out where you are?

i) How does a file know which directory it is in?

In Unix, a file does not store information about which directory it is in. Instead:

1. A directory holds mappings between file names and their inode numbers.
2. Each file is represented by an inode, which stores metadata (permissions, owner, timestamps, etc.), but not its directory path or name.
3. Thus, a file "resides" in a directory because the directory contains an entry that links a file name to the inode — not the other way around

ii) How does pwd figure out where you are?

The pwd command (print working directory) determines the absolute path of the current directory using the following steps:

1. It first gets the inode number of the current directory using the stat system call.
2. Then it moves to the parent directory ("..") using the chdir system call.
3. Inside the parent directory, it reads entries (opendir, readdir) to find the filename whose inode matches the previous directory's inode.
4. This process repeats until the root directory is reached.
5. The collected names are reversed to print the full path from root. This way pwd reconstructs the full absolute path.

b. Unix organizes disk storage into file systems. A file system is a collection of files and directories. A directory is a list of names and pointers. Each entry in a directory point to a file or directory. A directory contains entries that point to its parent directory and to its subdirectories. Now the question is how the Unix file system keeps track of large files.

i) What are the facts?

ii) What are the possible problems and solutions? (6mark)

i) What are the facts?

- Unix uses **inodes** to store metadata about files (size, ownership, permissions, timestamps).
- Each inode contains **pointers** to data blocks that hold the actual content.
- For small files, all data can be accessed using **direct pointers** from the inode.
- For large files, the inode also includes:
    - **Single indirect pointer** → points to a block that contains more pointers to data blocks.
    - **Double indirect pointer** → points to a block that contains pointers to other blocks of pointers.
    - **Triple indirect pointer** → allows even deeper levels of referencing.
- This multi-level pointer system allows Unix to track **very large files efficiently**, even if the inode has limited direct pointers.

ii) What are the possible problems and solutions?

Problems:

1. **Access time increases** with file size due to multiple levels of indirection.
2. **Corruption in indirect blocks** can make large portions of the file inaccessible.
3. Managing very large files increases **metadata overhead**.

Solutions:

1. Use **buffer caching** and **read-ahead techniques** to speed up access.
2. Use **journaling file systems** (e.g., ext3/ext4) to reduce risk of corruption.

3. **Regular defragmentation** help maintain performance and recoverability.

5.c. Standard Unix shells do not die when the user sends the interrupt or quit signal when the child is running. How does a standard Unix shell respond to these signals when regarding a command line?

Standard Unix shells handle interrupt and quit signals in a specific way when a child process is running. When a user sends an interrupt or quit signal (e.g., Ctrl+C for SIGINT, Ctrl+\ for SIGQUIT), the shell typically does not terminate itself. Instead,

- It intercepts these signals and then sends them to the currently running child process.
- The child process then receives the signal and is usually terminated.
- After the child process terminates, the shell resumes its execution and re-displays its prompt, allowing the user to enter further commands.
- This behavior ensures that the user's interactive session remains active even if a running command is interrupted

6.a. What do you know about thread of execution? Explain multiple threads of execution with proper example. (4 mark)

A **thread of execution** is the path followed by a single sequence of instructions in a program. In a single-threaded program, only one task is executed at a time.

Multithreading allows a program to run **multiple tasks concurrently** within the same process.

- All threads share the same memory, file descriptors, and process ID.
- Each thread has its own **stack and CPU registers**, enabling independent execution.
- Threads can communicate easily using shared global variables.

Example: Multi-threaded "Hello World" Program

In a multithreaded version of a Hello World program:

- One thread prints "hello ".
- Another thread prints "world".
- Outputs shows hello world, showing concurrent execution.

```
#include <stdio.h>
#include <pthread.h>

// Function for Thread 1
void *thread1_func(void *arg) {
```

```
    printf("Hello\n");
    return NULL;
}

// Function for Thread 2
void *thread2_func(void *arg) {
    printf("World\n");
    return NULL;
}

int main() {
    pthread_t t1, t2;

    // Create two threads
    pthread_create(&t1, NULL, thread1_func, NULL);
    pthread_create(&t2, NULL, thread2_func, NULL);

    // Wait for both threads to finish
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);

    return 0;
}
```

Here, pthread_create() creates threads, and pthread_join() waits for them to finish. This illustrates how multithreading achieves **parallel output without waiting** for one task to finish first.

MID1 (2017-18)

3. Distinguish system program based on the following keywords: API, operating system, system level service, resource, service, and functionality of the operating system.

- **API (Application Programming Interface):**
  System programs provide APIs that allow user programs to access low-level OS features like file handling, process control, etc., in a controlled and abstracted way.
- **Operating System:**
  The OS is the core software managing hardware and resources. System programs act as a **bridge between the user and the OS**, enabling easier interaction.
- **System-Level Service:**
  System programs perform system-level tasks such as managing files, user accounts, and processes. Examples include compilers, shells, and backup utilities.

- **Resource:**
  System programs help in **allocating and managing system resources** (CPU, memory, disk, I/O) effectively, often using OS-provided mechanisms.
- **Service:**
  They provide essential services to users and applications—such as file editing (vi, nano), compilation (gcc), and automation (shell scripts).
- **Functionality of the Operating System:**
  System programs extend the OS's functionality by providing user-friendly tools for performing complex tasks, making OS features more accessible.

4. What is one pass and two pass compilers Explain.

A compiler translates high-level source code into machine code. The way it processes the source code determines whether it is a one-pass or two-pass compiler.

1) One-Pass Compiler

A one-pass compiler processes the source code in a single pass, meaning it reads the code from start to finish once, translating it into machine code without going back.

Characteristics:

- A one-pass compiler reads the source code only once from beginning to end.
- It processes syntax analysis, semantic checks, and code generation in a single pass.
- Faster and uses less memory, but cannot handle forward references (e.g., using a function before it's declared).
- Used in simple or small language compilers.

2) Two-Pass Compiler

A two-pass compiler scans the source code twice before generating machine code.

Pass 1 (Analysis Pass):

Scans the entire program to collect information, such as variable declarations and function definitions. Then builds a symbol table to store variable names, types, and memory locations.

Pass 2 (Code Generation Pass):

Uses the information from Pass 1 to generate machine code. It can optimize code since it has a complete view of the program.

Characteristics:

- More flexible because it allows functions and variables to be used before their declaration.
- Supports optimizations, making programs more efficient.
- Used in modern compilers like GCC.

Mid 2 (2017-18)

To a programmer, a system call looks like any other call to a library procedure. Is it important that a programmer know which library procedures result in system calls? Under what circumstances and why? (2)

Yes, it is important for a programmer to know which library procedures result in system calls because:

- System calls involve context switching between user and kernel mode, which is slower than normal function calls. Avoiding unnecessary system calls can improve performance.
- Proper error handling is critical, as system calls can fail due to resource limits, permissions, or hardware issues. Knowing when a system call is made helps handle such errors correctly.
- Some system calls require special privileges (e.g., root access). Being aware helps avoid unexpected permission errors or security issues.
- Overuse of system calls can reduce portability across platforms. Standard library functions may offer more portable alternatives.
- System calls interact directly with hardware and kernel resources, such as memory, files, and devices. This requires the programmer to manage these resources carefully, ensuring proper allocation and deallocation.
- During debugging or profiling, knowing which functions trigger system calls helps identify performance bottlenecks and optimize code.
- For example, fread() uses read() internally. Choosing the right one depends on the need for buffering or raw performance.

2016-17

2.c. What is Shell Scripting? Explain and write some commands.

A shell script is a text file containing a sequence of commands for a UNIX/Linux shell to execute. It is used to automate repetitive tasks, manage system operations, and perform batch processing. Shell scripting is commonly done in Bash but also works in other shells like sh, ksh, and zsh.

Supports variables, loops (for, while), conditionals (if, case), and functions.

Used for:

- Automating backups
- System monitoring
- Batch file processing
- Installing packages

Command	Description
ls	Lists files and directories
pwd	Prints current working directory
cd	Changes directory
echo	Displays messages or variables
cat	Displays contents of a file
mv	Moves or renames files
mkdir	Creates a new directory
rmdir	Removes an empty directory
chmod	Changes file or directory permissions
chown	Changes file or directory ownership
grep	Searches text using patterns (regular expressions)

5.c. The return value from **fork()** allows a process to determine if it is the parent or child process. What other technique can a process use?

Apart from the return value of fork(), a process can use the getpid() and getppid() system calls to determine information about itself and its relationship to other processes.

Techniques:

getpid() – Get Process ID

- Returns the process ID (PID) of the calling process.
- Every process, including child and parent, has a unique PID.

getppid() – Get Parent Process ID

- Returns the parent process ID (PPID) of the calling process.
- A child process can use this to identify its parent.

```
pid_t pid = fork();

if (pid == 0) {
```

```
    // Child process

    printf("Child PID: %d, Parent PID: %d\n", getpid(), getppid());

} else {

    // Parent process

    printf("Parent PID: %d\n", getpid());

}
```

6.b. List the common functions performed by UNIX APIs. Give example.

UNIX APIs provide system-level access for managing resources. Common functions include:

1. **Process Management**
    - Create, execute, and terminate processes.
    - Example: fork(), exec(), wait()
2. **File Management**
    - Create, read, write, and delete files.
    - Example: open(), read(), write(), close()
3. **Device I/O**
    - Communicate with hardware devices using file descriptors.
    - Example: read(), write() on device files
4. **Memory Management**
    - Allocate and manage memory.
    - Example: mmap(), brk()
5. **Directory Management**
    - Create and manage directories.
    - Example: mkdir(), rmdir(), chdir()
6. **User and Group Management**
    - Access and modify user credentials.
    - Example: getuid(), getgid()
7. **Interprocess Communication (IPC)**
    - Enable communication between processes.
    - Example: pipe(), msgget()

Example:

```
int fd = open("data.txt", O_RDONLY);
read(fd, buffer, 100);
close(fd);
```

This example opens a file, reads data, and closes it using UNIX system calls.

c. Write the syntax of if unconditional statement in UNIX. Example with an example.

```
if [ condition ]

then

  commands

fi
```

- condition: A test expression (e.g., file check, string comparison, number check)
- then: Begins the block of commands to be executed if the condition is true.
- fi: Marks the end of the if block.

Example:

```
#!/bin/bash

num=10

if [ $num -gt 5 ]

then

  echo "Number is greater than 5"

fi
```

Output: Number is greater than 5

2015-16

Explain syntax awk command

The awk command is a powerful text-processing tool in UNIX/Linux used for pattern scanning, filtering, and reporting.

Syntax:

```
awk 'pattern {action}' filename
```

- **pattern**: A condition to match (e.g., a specific string).

- **action**: The command to execute when the pattern matches.
- **filename**: The file to be read.

Common Usage Examples:

1. **Print all lines**:

```
awk '{print}' file.txt
```

2. **Print the first column**:

```
awk '{print $1}' file.txt
```