i. A device driver is missing after a kernel upgrade. How can the user verify whether the driver is loaded and load it if not? [2 Marks]

Answer:
To verify if the driver is loaded, the user can use the command:

bash
CopyEdit
```
lsmod | grep <driver_name>
```

If the driver is not listed, it means it is not loaded. To load the driver, use the command:

bash
CopyEdit
```
sudo modprobe <driver_name>
```

Replace `<driver_name>` with the actual name of the driver.

---

ii. Running the command `sudo smartctl -H /dev/<device>` shows: "SMART overall-health self-assessment test result: FAILED". What does this imply, and what logical next steps should be taken? [2 Marks]

Answer:
This message indicates that the hard drive has failed its self-assessment test and may soon become unusable.
Next steps:

- Immediately **backup** important data.
- Replace the disk as soon as possible.

Monitor the drive with further SMART tests using:

bash
CopyEdit
```
sudo smartctl -a /dev/<device>
```

-

---

iii. Your system performs multiple reads and writes to disk, and response time is critical. You can switch between CFQ, Deadline, and NOOP schedulers. Which scheduler should be used to minimize latency, and why? [1 Mark]

Answer:
Deadline scheduler should be used to minimize latency.
Reason: It is designed to provide fast response times by prioritizing I/O requests with deadlines, making it suitable for real-time or latency-sensitive workloads.

i.

c
CopyEdit

```c
if (fork() == 0) {
    printf("Child\n");
} else {
    printf("Parent\n");
}
printf("End\n");
return 0;
```

How many times each `printf` will be executed?
[1 Mark]

Answer:

- When `fork()` is called, it creates **one child process**, so there will be **two processes** in total.
- Both parent and child will execute the code after the fork.
- So:
    - `"Child\n"` will be printed once (by child process).
    - `"Parent\n"` will be printed once (by parent process).
    - `"End\n"` will be printed **twice** (once by parent and once by child).

✅ **Output summary:**

sql
CopyEdit

```
Child
```

```
End
Parent
End
```

(Note: Order may vary because of process scheduling.)

---

ii.

c
CopyEdit

```c
#include <stdio.h>
#include <unistd.h>

int main() {
    fork();
    fork();
    printf("BITS\n");
    return 0;
}
```

How many times will `"BITS\n"` be printed?
[1 Mark]

Answer:
Each `fork()` doubles the number of processes:

- First `fork()` → 2 processes
- Second `fork()` → Each of those 2 processes forks again → Total **4 processes**

So, `printf("BITS\n");` will be executed by **4 processes**.

✅ **Output:**
`BITS` will be printed **4 times**.

i. You are editing a configuration file in vi that has multiple sections, each starting with the word `Section:`. You want to list and jump between all these section headers quickly.
What command or method can you use in vi to list and jump through all lines matching a pattern like `Section:`?
[1 Mark]

Answer:
Use the command:

ruby
CopyEdit
`:g/Section:/`

This lists all lines containing "Section:".
To jump between them, use the `n` (next) or `N` (previous) after searching with `/Section:`.

---

ii. You've edited several parts of a file but now want to return to the last edited location after scrolling through the document. What command in vi lets you jump back to the last modification location?
[1 Mark]

Answer:
Use the command:

go
CopyEdit
`` `` (two backticks)``

This jumps to the exact location of the last change.

---

iii. You need to move (not copy) lines 50 to 60 to come after line 120 in vi. Which sequence of commands would you use to move this block of text efficiently?
[1 Mark]

Answer:
Use the following sequence in command mode:

arduino
CopyEdit
`:50,60move 120`

---

iv. While editing a large log file, you want to delete all lines between two markers, e.g., `#DEBUG START` and `#DEBUG END`. What command in vi allows you to delete a range of lines based on matching start and end patterns?
[1 Mark]

Answer:
Use this command in vi:

pgsql
CopyEdit
```
:/#DEBUG START/,/#DEBUG END/d
```

---

v. You are searching for a specific function name in a large source code file using vi, but you are unsure about the exact case (e.g., InitFunction, initfunction).
How can you perform a case-insensitive search for this function name in vi?
[1 Mark]

Answer:
Use the following command:

CopyEdit
```
/\cinitfunction
```

The `\c` makes the search case-insensitive in vi.

i. Write a shell script using a case statement to accept a day of the week (Mon/Tue/…) from the user and display whether it's a weekday or weekend.
[2 Marks]

Answer:

bash
CopyEdit
```
#!/bin/bash
read -p "Enter day (Mon/Tue/...): " day

case $day in
  Mon|Tue|Wed|Thu|Fri)
```

```
    echo "Weekday"
    ;;
  Sat|Sun)
    echo "Weekend"
    ;;
  *)
    echo "Invalid input"
    ;;
esac
```

---

ii. Determine the output
[1 Mark]

bash
CopyEdit

```
#!/bin/bash
read -p "Enter start number: " num
while [ $num -le 15 ]; do
  echo $num
  ((num+=3))
done
```

Assume initial value of num as **4**.

Answer:
The loop starts at 4 and adds 3 each time until `num` is greater than 15.
So, the output will be:

CopyEdit

```
4
7
10
13
```

---

iii. Write a shell script that stores the names of five fruits in an array, and then uses a loop to display each fruit name with its position in the array (starting from index 0).
[2 Marks]

Answer:

bash
CopyEdit

```
#!/bin/bash
fruits=("Apple" "Banana" "Mango" "Grapes" "Orange")

for i in "${!fruits[@]}"; do
  echo "Fruit at index $i is ${fruits[$i]}"
done
```

i. Consider the absolute `path /var/log/syslog`.
The system starts resolving the path step-by-step using inode lookups.
What is the first inode that the system will read, and why?
[1 Mark]

Answer:
The first inode the system will read is the **root directory's inode (`/`)**.
Reason: Path resolution starts from the root (`/`) since `/var/log/syslog` is an absolute path. The system always starts from the root inode to locate each directory in sequence.

---

ii. A program opens a file using `open()`, reads 100 bytes with `read()`, and then duplicates the file descriptor using `dup()`. It then reads another 100 bytes using the duplicated file descriptor.
How many bytes of file data have been read? Justify your answer.
[2 Marks]

Answer:
A total of **200 bytes** have been read.
Justification:
`dup()` creates a new file descriptor pointing to the **same file offset**.
So after reading 100 bytes with the original descriptor, the offset moves to 100.
Reading another 100 bytes from the duplicate descriptor continues from offset 100 → total 200 bytes read.

---

iii. You run `unlink myfile.txt`. Will `myfile.txt` still appear in `lsof` as open by a process?
Justify your answer.
[2 Marks]

Answer:
Yes, `myfile.txt` will still appear in `lsof` if a process has it open.
Justification:
`unlink` removes the file **name** from the directory, but the **file content remains** on disk until all file descriptors referring to it are closed.
If a process has the file open, it can still access the data, and `lsof` will show it as open.

i. Initial Inode for Path Resolution

The first inode that the system will read to resolve the path `/var/log/syslog` is the inode for the **root directory (`/`)**.

Justification: Because `/var/log/syslog` is an **absolute path**, the kernel always begins the lookup process from the root directory. The root directory's inode contains the location of the `var` directory, which is the first step in resolving the full path.

---

ii. Bytes Read After `dup()`

A total of **200 bytes** of file data have been read.

Justification: The `dup()` system call creates a second file descriptor that points to the *exact same* underlying **open file description** in the kernel as the original. This means both descriptors share a single **file offset**. The first `read()` reads 100 bytes and advances the offset to 100. The second `read()`, using the duplicated descriptor, starts from that same offset (100) and reads the next 100 bytes, for a total of 200 bytes read from the file.

---

iii. `unlink` and `lsof`

Yes, `myfile.txt` will still appear in `lsof` as open by the process.

Justification: The `unlink()` command removes the file's name from the directory and decrements its **link count**. However, the kernel only deallocates the inode and its data

blocks when **both** the link count is zero **and** no processes have the file open. Since a process still holds an open file descriptor to the file, the file's data remains on disk, and `lsof` will correctly report it as an open file for that process, often with a `(deleted)` status.

i. Weekday/Weekend Classifier

This script prompts the user for a day of the week and uses a `case` statement to determine if it's a weekday or a weekend.

Bash

```
#!/bin/bash


# Prompt the user to enter a day of the week
read -p "Enter a day of the week (e.g., Mon, Tue, etc.): " day


# Use a case statement to check the input and handle different cases
case "$day" in
    Mon|mon|Tue|tue|Wed|wed|Thu|thu|Fri|fri)
        echo "$day is a weekday."
        ;;
    Sat|sat|Sun|sun)
        echo "$day is a weekend."
        ;;
    *)
        echo "Invalid input. Please enter a valid day abbreviation."
        ;;
esac
```

---

ii. Output Determination

The script initializes `num` to 4 and enters a `while` loop that continues as long as `num` is less than or equal to 15. In each iteration, it prints the current value of `num` and then increments it by 3.

The execution trace is:

- Prints **4** (num becomes 7)
- Prints **7** (num becomes 10)
- Prints **10** (num becomes 13)
- Prints **13** (num becomes 16)
- Loop terminates because 16 is not less than or equal to 15.

The final output will be:

4
7
10
13

---

iii. Looping Through an Array

This script creates an array of five fruits and then uses a `for` loop to iterate through it, printing each fruit's name along with its corresponding index.

Bash
#!/bin/bash

# 1. Store the names of five fruits in an array
fruits=("Apple" "Banana" "Cherry" "Date" "Elderberry")

2. Use a C-style for loop to iterate and display each fruit with its index
echo "Displaying fruits with their array indices:"
for ((i=0; i<${#fruits[@]}; i++)); do
    echo "Index $i: ${fruits[i]}"
done

i. List and Jump Between Pattern Matches

To list all lines containing "Section", you can use the global command:

:g/Section/nu

To jump between the matches, use the search command. Type `/Section` and press Enter. Then, use **`n`** to jump to the next match and **`N`** to jump to the previous one.

---

ii. Jump to Last Edited Location

To jump to the exact cursor position of the last modification, use the following command in normal mode:

`.

(That's a backtick followed by a period.)

---

iii. Move a Block of Lines

The most efficient way to move lines 50 through 60 to appear after line 120 is with the `:move` command:

:50,60m120

---

iv. Delete Lines Between Two Patterns

You can use search patterns to define a range for the delete command (`:d`). To delete all lines from `#DEBUG START` to `#DEBUG END` inclusive, use:

:/#DEBUG START/,/#DEBUG END/d

---

v. Perform a Case-Insensitive Search

To perform a case-insensitive search for a single term, append `\c` to your search pattern.

/initfunction\c

Alternatively, you can make all subsequent searches in the session case-insensitive by running the command `:set ignorecase` (or the shorter `:set ic`).

First Snippet (`if/else` block)

- `printf("Child\n");`: **1 time** (only in the child process).
- `printf("Parent\n");`: **1 time** (only in the parent process).
- `printf("End\n");`: **2 times** (once by the parent and once by the child).

Second Snippet (Two `fork()` calls)

"BITS" will be printed **4 times**.

Each `fork()` call doubles the number of active processes (1 process → 2 processes → 4 processes). The final `printf` statement is executed by all four resulting processes.

The `raise()` Function in C Signal Handling

The `raise()` function is a standard library function in C, declared in the `<signal.h>` header. Its primary purpose is to allow a program to send a signal to itself.

1. Definition and Syntax

The `raise()` function sends the signal specified by `sig` to the calling process.

- **Header:** `<signal.h>`
- **Syntax:** `int raise(int sig);`
- **Parameter:** `sig` is the integer value of the signal to be sent (e.g., `SIGINT`, `SIGABRT`, `SIGUSR1`).
- **Return Value:** It returns `0` on success and a non-zero value if an error occurs.

2. Purpose and Use Case

The `raise()` function is used when a program needs to trigger its own signal handling logic based on an internal state or error, rather than waiting for an external event. Key use cases include:

- **Testing:** It allows developers to test their custom signal handler functions without needing to manually send signals from the terminal (e.g., with `Ctrl+C`).
- **Error Handling:** A program can raise a specific signal like `SIGABRT` (abort signal) when it detects a critical, unrecoverable internal error, ensuring a controlled but immediate shutdown.
- **Self-Interruption:** It can be used to interrupt the program's normal flow to perform a specific action defined in a signal handler.

3. Example C Program

The following program demonstrates how to use `raise()`. It first registers a custom handler for the `SIGUSR1` signal (a user-defined signal), and then uses `raise()` to trigger it.

C

```
#include <stdio.h>    // For printf
#include <signal.h>   // For signal() and raise()
#include <stdlib.h>   // For exit()


/**
 * @brief A custom signal handler function.
 * This function will be executed when the signal it's registered for is raised.
 * @param signum The integer representing the signal number caught.
 */
void custom_signal_handler(int signum) {
    printf("\n>>> Inside the custom signal handler for signal %d.\n", signum);
    printf(">>> The handler has finished its task. Control will now return to the main
program.\n");
}


int main() {
    printf("Main Program: Registering a custom handler for the SIGUSR1 signal...\n");


    // Use the signal() function to associate SIGUSR1 with our custom handler.
    // If SIGUSR1 is raised, custom_signal_handler() will be called.
    signal(SIGUSR1, custom_signal_handler);


    printf("Main Program: The handler is registered. About to raise the SIGUSR1 signal.\n\n");


    // Use raise() to send the SIGUSR1 signal to this program itself.
    raise(SIGUSR1);


    printf("\nMain Program: Control has returned from the signal handler.\n");
    printf("Main Program: Resuming normal execution and finishing.\n");
```

```
    return 0;
}
```

4. Explanation of Program Flow and Output

When the above C code is compiled and executed, it will produce the following output, demonstrating the flow of control:

Expected Output:

```
Main Program: Registering a custom handler for the SIGUSR1 signal...
Main Program: The handler is registered. About to raise the SIGUSR1 signal.

>>> Inside the custom signal handler for signal 10.
>>> The handler has finished its task. Control will now return to the main program.

Main Program: Control has returned from the signal handler.
Main Program: Resuming normal execution and finishing.
```

- **Step 1:** The `main` function starts and registers `custom_signal_handler` as the function to call when `SIGUSR1` is received.
- **Step 2:** The `raise(SIGUSR1)` line is executed. This immediately pauses the execution of `main`.
- **Step 3:** The operating system transfers control to the registered handler, `custom_signal_handler`. The code inside this function runs, printing the lines that start with `>>>`.
- **Step 4:** Once the handler function completes, control returns to the `main` function at the point right after the `raise()` call was made.
- **Step 5:** The rest of the `main` function executes, printing the final messages and exiting normally.

2. Search, Sort, and Deduplicate Errors

This command finds all case-insensitive lines containing "error" in `app.log`, sorts them, removes any duplicates, and saves the result.

Bash

```
grep -i "error" app.log | sort | uniq > unique_sorted_errors.txt
```

---

ii. Sort and Paginate a File

This command sorts `data.txt` in reverse order, then uses a pipe to skip the first 5 lines (`tail`) and show the next 10 (`head`).

Bash

```
sort -r data.txt | tail -n +6 | head -n 10
```

---

iii. Combine and Clean Multiple Files

This command combines three files, removes any blank lines, sorts the combined content, removes duplicates, and saves the clean data.

Bash

```
cat file1.txt file2.txt file3.txt | grep -v '^$' | sort | uniq > clean_sorted.txt
```

---

iv. Count Total Occurrences Across Multiple Files

This command finds all lines containing "CRITICAL" across all `.log` files in `/var/logs/`, pipes the output to `wc -l` to count the total number of matching lines, and saves that count to a file.

Bash

```
grep "CRITICAL" /var/logs/*.log | wc -l > critical_count.txt
```

---

v. Find Lines Not Containing a Word

This command uses `grep` with the `-v` (invert match) flag to display all lines from `somefile.log` that **do not** contain the word "warning".

Bash

```
grep -v "warning" somefile.log
```