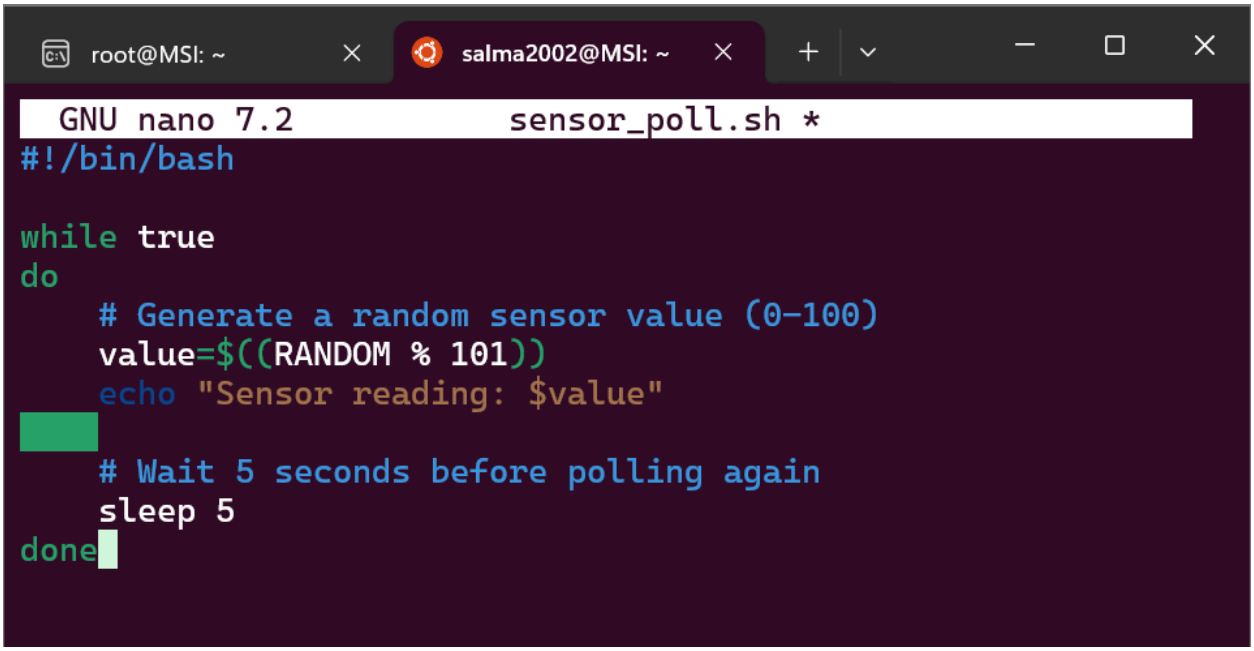
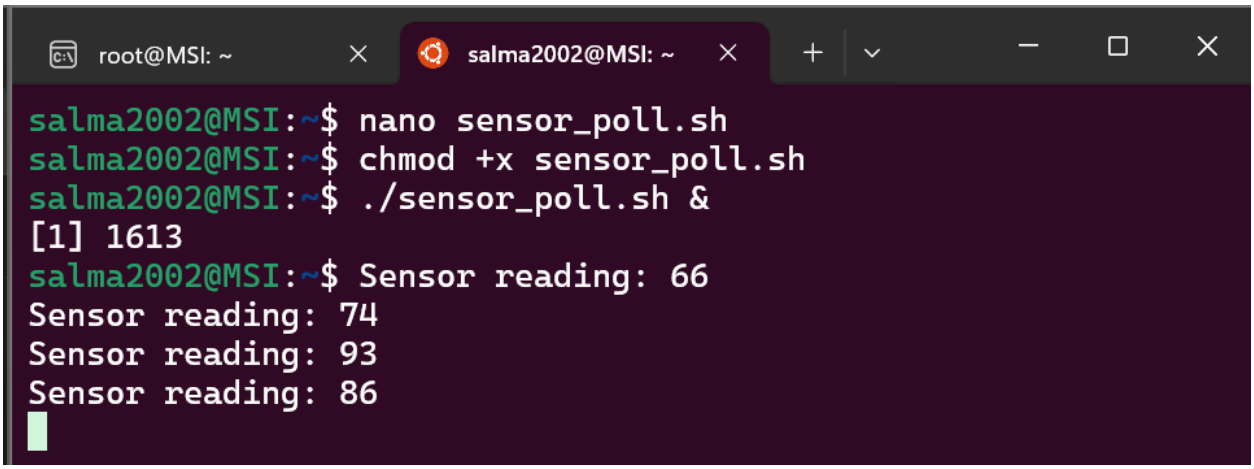
Day 4 – Phase 4: Process and Network Monitoring

Boss's Request: Monitor the system while simulating sensor activity.

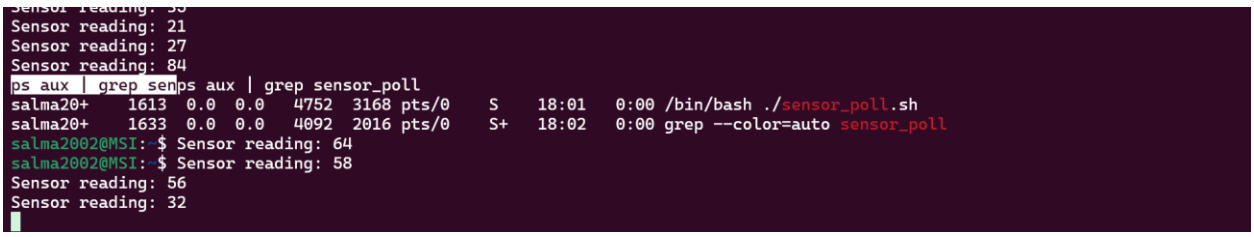
Tasks:

• Run a background task to simulate sensor polling.





• List processes and filter for the background task.



• Check network states (established connections).

```
Sensor reading: 70
netstat -ant | grep ESTABLISHED
salma2002@MSI:~$ Sensor reading: 72
Sensor reading: 47
```

• Try foreground and background switching.

```
Sensor reading: 37
fg %1
./sensor_poll.sh
Sensor reading: 78
Sensor reading: 54
Sensor reading: 59
```

• Kill a process if needed.

```
salma2002@MSI:~$ kill -9 1613
salma2002@MSI:~$ jobs -l
[1]+  1613 Killed                  ./sensor_poll.sh
salma2002@MSI:~$
```

Open-Ended Questions:

• What happens step by step when you type a command in bash (e.g., ls) until you see the output?

Step 1: Reading the Command

The shell starts by reading the command line you've typed, from the point you hit the first character to when you press the **Enter** key. It stores this input as a single string of characters.

Step 2: Tokenization and Parsing

Next, Bash breaks the input string into a list of **tokens**. Tokens are the fundamental building blocks of the command, separated by spaces. For example, if you type ls -l /home, the tokens are ls, -l, and /home.

Following tokenization, the shell parses these tokens to determine what they mean. It checks for special characters and operators (like |, >, &), which dictate how the command should be executed.

Step 3: History Expansion

Bash checks for history expansion characters, most notably !. If present, it replaces the history event with the corresponding command from your command history. For example, !$ expands to the last argument of the previous command.

Step 4: Alias Expansion

The shell checks if the first token is an **alias**. An alias is a shortcut for a longer command. For instance, ll is often aliased to ls -alF. If an alias is found, Bash replaces the alias with its full command and restarts the parsing process from Step 2.

Step 5: Command Lookup

The shell now tries to figure out what the first token is. It follows a specific order of precedence to find a match:

- **Keyword**: It first checks if the command is a **shell keyword** (like if, while, for).
- **Alias**: If it isn't a keyword, it checks for **aliases** again (as mentioned in Step 4).
- **Shell Builtin**: It checks if the command is a **builtin shell command** (like cd, echo, pwd). These are commands compiled directly into the shell executable, making them very fast.
- **Executable File**: If the command is none of the above, Bash searches for an executable file with that name in the directories listed in the **$PATH environment variable**. It checks each directory in the order they appear until it finds a match.

Step 6: I/O Redirection and Expansions

Before executing the command, Bash handles several types of **expansions**:

- **Tilde Expansion**: Replaces ~ with the path to the current user's home directory.
- **Variable Expansion**: Replaces variables like $USER with their values.
- **Command Substitution**: Executes a command within backticks (` `) or $(...) and replaces it with the command's output.
- **Filename Expansion (Globbing)**: Replaces wildcard characters like * and ? with matching filenames.
- **Word Splitting**: Breaks the results of expansions into separate words.
- **Quote Removal**: Removes any quotation marks that were used to prevent expansions.

It also handles **I/O redirection**, which is the process of changing where the input and output of a command go. The shell sets up pipes for the standard input (stdin), standard output (stdout), and standard error (stderr) streams.

Step 7: Executing the Command

Once all the preparations are complete, Bash executes the command.

- **For builtin commands**, the shell executes them directly without creating a new process.
- **For external commands** (executables found in $PATH), the shell creates a new process by **forking** itself. The **parent process** (the original shell) then waits for the new **child process** to finish. The child process then executes the command using the exec system call, replacing its own image with the new program.

Step 8: Waiting and Cleanup

After the command finishes, the child process exits and returns a status code (0 for success, non-zero for an error). The parent shell process then checks this status code and performs any necessary cleanup, such as updating the command history and displaying the command prompt again, ready for your next command.

• Explain the types of processes in Linux: daemon, zombie, orphan. How can you detect them?

A zombie process is a process that has been terminated but still has an entry in the process table while an orphan process refers to a process whose parent process has terminated but continues to run. On the other hand, Daemon process are background processes that start with system boot and run until the shutdown.

• Why do we need Inter-Process Communication (IPC)? List some IPC mechanisms and real-life examples.

Processes need to communicate with each other in many situations. **Inter-Process Communication or IPC** is a mechanism that allows processes to communicate.

- It helps processes synchronize their activities, share information and avoid conflicts while accessing shared resources.
- There are two method of IPC, shared memory and message passing. An operating system can implement both methods of communication.

1. **Pipes -** A pipe is a unidirectional communication channel used for IPC between two related processes. One process writes to the pipe, and the other process reads from it.
   Types of Pipes are Anonymous Pipes and Named Pipes (FIFOs)

   Example: In a Linux shell, the command ls | grep .txt uses a pipe (|) to send the output of the ls command (a list of files) as the input to the grep command, which then filters for files ending in .txt.

2. **Sockets -** Sockets are used for network communication between processes running on different hosts. They provide a standard interface for communication, which can be used across different platforms and programming languages.

   Example: A web browser (client process) uses a socket to communicate with a web server (server process) to request and receive web pages.

3. **Shared memory -** In shared memory IPC, multiple processes are given access to a common memory space. Processes can read and write data to this memory, enabling fast communication between them.
4. **Semaphores -** Semaphores are used for controlling access to shared resources. They are used to prevent multiple processes from accessing the same resource simultaneously, which can lead to data corruption.
5. **Message Queuing -** This allows messages to be passed between processes using either a single queue or several message queue. This is managed by system kernel these messages are coordinated using an API.

   Example: A producer-consumer system where one process generates data (the producer) and places it into a message queue, and another process (the consumer) reads the data from the queue to process it.