# Question 1: Understanding Hardware Configuration via /proc Filesystem

***Command:***

```
more /proc/cpuinfo
```

This command provides detailed information about your CPU. Key terms to note:

- **processor**: Represents the logical processors (threads) in your system.
- **cores**: Refers to the physical cores of the CPU. Each core may support multiple threads if hyperthreading is enabled.

***Verify with `lscpu`:***

```
lscpu
```

- `lscpu` displays a concise summary of CPU architecture:
  - Core count
  - Thread count
  - Hyperthreading status
- **Hyperthreading**: Allows a single physical core to function as multiple logical processors.

***Tips:***

- Use `lscpu` to answer subquestions (b) to (e) regarding cores, processors, frequency, and architecture.
- For parts (f) to (h), use these additional commands:
  - Total and free memory: `free` or `cat /proc/meminfo`
  - Forks and context switches: `grep` specific fields in `/proc/stat`.

# Question 2: Monitoring a Running Process Using `top`

***Steps:***

1. **Compile and Run the Program**:

   ```
   gcc cpu.c -o cpu
   ./cpu
   ```

   a. This runs the program `cpu` in an infinite loop.
2. **Open `top` in Another Terminal**:

   ```
   top
   ```

The `top` command displays a list of all running processes with their resource usage.

***Analyze the Output:***

- **(a) Find the PID**:
  - Look for the process named `cpu` in the `COMMAND` column. The PID is shown in the first column.
- **(b) Check CPU and Memory Usage**:
  - `%CPU`: Percentage of CPU being used by the process.
  - `%MEM`: Percentage of memory consumed by the process.
- **(c) Observe the Process State**:
  - The STATE column indicates the current process state:
    - R: Running
    - S: Sleeping (Blocked)
    - Z: Zombie

***Exit Commands:***

- Quit `top`: Press q.
- Stop the `cpu` program: Press `Ctrl+C` in the terminal where it is running.

Question 3:

# Assignment Help: Shell Behavior - Child Processes and I/O Redirection

## 1. Compile and Run the Program

- **Task**: Compile and run the cpu-print.c program.
    - Use gcc to compile the program: gcc cpu-print.c -o cpu-print

    - Execute the compiled program: ./cpu-print

The program will run indefinitely, printing output to the terminal.

## 2. Process Identification and Parent Processes

### *(a) Identify the PID of the cpu-print Process*

- Use the ps command to check the list of running processes:
    - ps aux | grep <process-name>
    - Look for the PID (Process ID) of cpu-print from the output.

### *(b) Trace Parent and Ancestor Processes*

- To find the parent process (PPID) of cpu-print:
    - ps -o ppid= -p <PID-of-cpu-print>

- Use pstree to trace the ancestors of the process:
    - pstree -p <PPID>

## 3. I/O Redirection to a File

### *(c) Redirect the Output of cpu-print to a File*

- Run the program and redirect output to a file: ./cpu-print > /tmp/tmp.txt &
- To explore file descriptors, use the lsof command: lsof -p <PID-of-cpu-print>
    - Look for entries for standard input (0), standard output (1), and standard error (2).
    - The output file (/tmp/tmp.txt) should be associated with 1 (stdout).

## 4. Pipe Implementation

### *(d) Implement a Pipe and Examine Processes*

- Run the program with a pipe: ./cpu-print | grep hello &
- Use ps to check the processes: ps aux | grep -E "cpu-print|grep"
- Check the file descriptors for each process: lsof -p <PID-of-cpu-print>
  lsof -p <PID-of-grep>

## 5. Investigate Built-in Commands

### *(e) Explore cd, ls, history, and ps*

- To identify whether the commands are built-in or external executables: which cd ls history ps
- To verify if a command is a shell built-in or external: type cd ls history ps

### Important Notes:

- **Avoid Overwriting Files**: Be cautious when redirecting output to files. If running a command like cpu-print indefinitely, the file can grow large and fill up the disk.
- **Stop Running Processes**: Use kill to stop a process if needed: kill -9 <PID>

- **Check Disk Space**: To see how much disk space is available: df -h

## Question 4 : Virtual vs. Physical Memory Usage

### Overview

This question focuses on understanding how virtual and physical memory usage differ based on program behavior. The programs `memory1.c` and `memory2.c` allocate large arrays of the same size. However:

- **`memory1.c`** allocates the array but does not access it.
- **`memory2.c`** allocates the array and accesses/modifies its elements.

You will observe memory usage differences using the `ps` command and analyze why the **Resident Set Size (RSS)** varies when the array is accessed in `memory2.c`.

### Compiling and Running the Programs

1. Compile the programs using the following commands:

   ```
   gcc memory1.c -o memory1
   gcc memory2.c -o memory2
   ```

2. Run each program in separate terminals:

```
./memory1

./memory2
```

3. Each program will display its **Process ID (PID)**. Note down the PID for both programs.

### Observing Memory Usage

1. Open another terminal and use the `ps` command to check the memory usage of each program:

```
ps -p <PID> -o pid,vsz,rss,comm
```

Replace `<PID>` with the PID of `memory1` or `memory2`.

   a. **VSZ (Virtual Memory Size):** The total memory reserved for the program, including memory that may not yet be physically allocated.
   b. **RSS (Resident Set Size):** The amount of physical memory (RAM) currently allocated to the process.

## Q5: Help Guide for Disk Access Programs

### 1. Setting Up the Environment

- **Create Folder**: Create a directory named disk-files:

```
mkdir disk-files
```

- **Add Sample File**: Place foo.pdf inside the disk-files folder:

```
cp foo.pdf disk-files/
```

- **Generate Multiple Files**: Use the make-copies.sh script to create 5000 copies of foo.pdf with unique filenames in the folder. Make the script executable with the following command:

```
chmod +x make-copies.sh
./make-copies.sh
```

### 2. Compiling and Running the Programs

- **Compilation**: Use GCC to compile disk.c and disk1.c:

```
gcc disk.c -o disk
```

- **Execution**: Run the program in the terminal:

```
./disk
```

- **Monitor Disk Usage**: Use tools like iostat, dstat, or iotop to monitor disk metrics.

### 3. Monitoring Disk Utilization

- Install iostat using your package manager.
- Use the following command to observe real-time disk activity: iostat -d 1

### 4. Clearing Disk Buffer Cache

To ensure you're reading from the disk and not from memory, clear the disk buffer cache using the following commands (Linux):

- **Clear page cache**:

```
sudo sync; sudo echo 1 > /proc/sys/vm/drop_caches
```

- **Clear dentries and inodes**:

```
sudo sync; sudo echo 2 > /proc/sys/vm/drop_caches
```

- **Clear both page cache and dentries/inodes**:

```
sudo sync; sudo echo 3 > /proc/sys/vm/drop_caches
```

# *Question 6:*

# *Debugging Programs Using GDB*

### Part 1: Debugging `pointers.cpp`

1. **Compile with Debug Symbols**:

```
g++ -g pointers.cpp -o pointers
```

2. **Run GDB**:

```
gdb ./pointers
```

3. **Set a Breakpoint**:

```
break main
```

4. **Start the Program**:

```
run
```

5. **Step Through the Code**: Use `next` to execute line by line.
6. **Find the Faulty Line**: If the program crashes, use `backtrace` to locate the issue.

## Part 2: Debugging `fibonacci.cpp`

1. **Compile with Debug Symbols**:

```
g++ -g fibonacci.cpp -o fibonacci
```

2. **Run GDB**:

```
gdb ./fibonacci
```

3. **Set Breakpoints**:
   a. At main: `break main`
   b. Inside the loop (e.g., line 13): `break <line_number>`
4. **Start the Program**:

```
run
```

5. **Print Variables**: Use `print <variable>` to observe variable values during execution.
6. **Step Through Code**: Use `next` to debug the loop and monitor logic.

### Common Commands Recap

- `break <line>`: Set a breakpoint.
- `run`: Start execution.
- `next`: Execute the next line.
- `print <variable>`: Show variable values.
- `backtrace`: Trace crash points.

# Question 7 : System Calls and strace

strace is a diagnostic, debugging, and instructional tool in Linux that allows users to trace the system calls made by a program during its execution. The objective is to trace the system calls made by an executable (e.g., a program or a Linux command like ls) using strace, analyze the trace output, and answer specific questions.

For example, to trace the system calls made by a program named cpu.c:

gcc cpu.c -o cpu.exe

strace <executable>

The output will contain detailed information about system calls made during the execution of the program or command.

<syscall_name>(<arguments>) = <return_value>

After running strace, count the number of lines representing system calls.

Find a list of supported system calls for your operating system. A good resource is the **man page for syscall** :

man 2 <syscall_name>