System Monitoring Tool -- Concurrency & Signals

This program monitors CPU and memory utilization in real-time and displays the data graphically in the terminal. It also provides information about the number of CPU cores and their maximum frequency in GHz. It does all this concurrently by using forking and pipes, and involves handling of Ctrl-C and Ctrl-Z signals.

Note that this readme is not ai-generated and written by me on a website called readme.so that makes your documentation look good in proper readme style. Thank you.

- Author: Sheharyar Meghani
- Date: March 24th , 2025
- References:
- Haved used information in lecture slides to implement forking and pipes, and handling of signals
- https://docs.kernel.org/filesystems/proc.html (https://docs.kernel.org/filesystems/proc.html) for cpu usage

- https://man7.org/linux/man-pages/man2/sysinfo.2.html (https://man7.org/linux/man-pages/man2/sysinfo.2.html) for getting ram info

- https://man7.org/linux/man-pages/man5/proc_cpuinfo.5.html (https://man7.org/linux/man-pages/man5/proc_cpuinfo.5.html) for getting cpu cores

- https://docs.kernel.org/cpu-freq/cpufreq-stats.html (https://docs.kernel.org/cpu-freq/cpufreq-stats.html) for getting max frequency

Makefile

- Has target A3 that creates the executable
- Has target A3.o that creates the object file from A3.c
- Has phony targets all that depends on A3 and phony target clean that removes the executable and object file To create executable use 'make' and to clean use 'make clean'

Requirements

- A Linux-based operating system (utilizes `/proc/stat` and `/proc/cpuinfo`).
- A terminal that supports ANSI escape sequences for cursor movement and screen clearing.

Compilation

Compile the program using the provided Makefile by entering in the command line:

```
make
```

Usage

Run the program with the following command:

```
./A3 [samples] [tdelay] [--memory] [--cpu] [--cores] [--sa
```

Command-line Options

Option	Description
`--samples`	Number of samples to display (default: 20).
`--tdelay`	Time delay between samples in microseconds (default: 500000).
`--memory`	Display only memory utilization
`--cpu`	Display only CPU utilization.
`--cores`	Display only CPU core information.

Example

Monitor only CPU utilization with 50 samples and a 2-second delay:

```
./A3 50 2000000 --cpu
```

Monitor only CPU utilization and Memory with 50 samples and a 2-second delay:

```
./A3 50  --cpu --tdelay=2000000 --memory
```

Monitor only CPU core information and Memory with 30 samples and a 2-second delay:

```
./A3 --tdelay=2000000 --memory --samples=30 --cores
```

Features and how problem was solved

1. **CPU Utilization**:

- Reads `/proc/stat` to calculate the percentage of CPU usage over time.
- **Formula**:
  `100.0 * (total_diff - idle_diff) / total_diff`
  where `total_diff` is the difference in total CPU usage between two samples taken at times T1 and T2 with tdelay difference , and `idle_diff` is the difference in idle time.

2. **Memory Utilization**:

- Uses the `sysinfo` system call to retrieve memory usage statistics.
- **Formula**:
  - `Total RAM (GB)` **=** `info.totalram / (1024 * 1024 * 1024)`
  - `RAM Used (GB)` **=** `(info.totalram - info.freeram) / (1024 * 1024 * 1024)`

3. **CPU Cores and Frequency**:

- Parses `/proc/cpuinfo` to determine the number of CPU cores.
- Parses '/sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_max_freq' to get the max frequency in KHz and converts to GHz
- Detects cores and frequency using the lines:
  - `processor` (for cores)
  - `"%f"` (for frequency)

4. **Graphical Display**:

- Utilizes ANSI escape codes to render graphs in the terminal.
- Metrics are displayed in different sections:
  - Memory: Starts at row 5
  - CPU: Starts at row 22
  - Cores: Starts at row 38
    Default row for output is row 5 if no memory is shown.

5. **Concurrency using forking and pipes**:

- In the plot_values function two pipes are created one for memory info and one for cpu info, the writing end is closed for the parent. For both cpu and memory, if the flag is active, a child process is forked and the child runs the memory or cpu child function. In this function the childs reading end is closed, and each child runs a loop over number of samples with tdelay, and writes into the pipe the respective information collected in each iteration. It closes the writing end and exits after its loop is over. The parent runs a while loop , reading from both the pipes in each iteration and plotting the respective values on the graphs.

6. **Plotting Samples with Delay**:.

- For each sample, calculated the Y-coordinate for memory and CPU utilization by scaling the values to fit the graph.
  - For memory, the Y-coordinate was calculated as `(used_memory / total_memory) * 12`, where 12 is the height of the memory graph.
  - For CPU, the Y-coordinate was calculated as `(CPU_utilization / 100) * 10`, where 10 is the height of the CPU graph.
- Used the `plot_point` function to plot the Y-coordinate at the corresponding X-coordinate (sample number).
- Drew axes for each category (memory and CPU) with the X-axis representing the sample number (size: `samples`) and the Y-axis representing the scaled values (size: 12 for memory and 10 for CPU).
- Used `usleep(tdelay)` to introduce a delay between each sample point, ensuring real-time visualization in the child functions.

7. **Drawing CPU Cores**:

- The show cores function creates two pipes , one for number of cores and one for max frequency, creates child processes for each that each call their own child function, in which they extract the info and write it into the pipes. Closing their write end and exiting after done. Parent runs a while loop, reading from the pipes in each iteration and filling the values of cores and max frequency.
- then calls the draw cores function.
- Used the `coresrow` parameter to determine the starting row for drawing the CPU cores.
- Calculated the height of the core layout as the square root of the number of cores (`sqrt(cores)`), ensuring a balanced grid.
- Calculated the width of the core layout as `cores / height`.
- Used a nested loop to draw each core in a grid-like structure:
  - The outer loop iterated over the number of cores.
  - The inner loop adjusted the position of each core based on the calculated height and width.
  - Used the `draw_core` function to render each core at the appropriate position.
- Displayed the total number of cores and their maximum frequency above the core layout.

---

8. **User Input Handling**:

- Accepts command-line arguments to customize the number of samples, delay between samples, and metrics to display.
- Uses the flag struct to store the flags for each utilization, 1 if to show and 0 to not. Also stores tdelay and number of samples

---

9. **Handling Signals**:

- In the main function , used sigaction to set the signal handlers of both SIGSTP and SIGINT.
- The SIGSTP handler simply returns and ignores the signal
- The SIGINT handler prompts user for an input Y or N. If Y it exits the program. If N it returns and does nothing and our program continues

- To stop the question being asked in all processes including children. I set the signal handler of SIGINT to SIGIGN using signal in the child processes to ignore the signals.

Input Format

The program accepts arguments in the following format:

1. **Positional Arguments**:
    - **`samples`**: Number of samples to collect (default: 20).
    - **`tdelay`**: Delay between samples in microseconds (default: 500000).
2. **Flags**:
    - `--memory`: Display only memory utilization.
    - `--cpu`: Display only CPU utilization.
    - `--cores`: Display only CPU core information.
3. **Keyword Arguments**:
    - `--samples=N`: Set the number of samples to `N`.
    - `--tdelay=T`: Set the delay between samples to `T` microseconds.

Input Constraints

1. **Order of Arguments**:
    - Positional arguments (`samples` and `tdelay`) must be the first two arguments.
    - Flags (`--memory`, `--cpu`, `--cores`) and keyword arguments (`--samples=N`, `--tdelay=T`) can follow.
2. **Restrictions**:

- `tdelay` and `samples` can only be changed once.
- flags (`--memory`, `--cpu`, or `--cores`) can only be inputted once.

Build and clean

- Build: make
- Cleam: make clean

Overview of Functions

`clear_screen()`

- Clears the terminal screen and moves the cursor to the top-left corner.
- **Usage**: Called at the start of the show function to refresh the display and make space for the incoming graphs.

`reset_cursor(int rows)`

- Resets the cursor position to avoid overwriting the graph.
- **Parameters**:
  - `rows`: Number of rows to move the cursor down.
- **Usage**: Called after plotting to ensure the next output doesn't overlap the graph.

`get_ram()`

- Retrieves the total RAM in the system using `sysinfo`.
- **Returns**: Total RAM in GB.

- **Usage**: Used to label the memory graph.

`draw_axes(int rows, int cols, char *o_label, char *y_label, int row_start)`

- Draws the X and Y axes for the graph.
- **Parameters**:
  - `rows`: Number of rows for the graph.
  - `cols`: Number of columns for the graph starting at column 8. Colums are the amount of samples
  - `o_label`: Label for the origin (e.g., "0 GB").
  - `y_label`: Label for the Y-axis (e.g., "100%").
  - `row_start`: Starting row for the graph.
- **Usage**: Called to initialize the graph layout.

`plot_point(int x, int y, int rows, int row_start, char *label)`

- Plots a point on the graph at the specified coordinates.
- **Parameters**:
  - `x`: The X-coordinate of the point, representing the sample number.
  - `y`: The Y-coordinate of the point, representing the scaled value (memory or CPU usage).
  - `rows`: The total number of rows in the graph, used to adjust the Y-coordinate for terminal positioning.
  - `row_start`: The starting row of the graph, ensuring the point is plotted within the correct section of the terminal.

    - `label`: The character or symbol used to represent the point (e.g., "#" for memory, ":" for CPU).
- **Usage**: This function is used to plot individual data points for memory and CPU utilization on their respective graphs.

---

get_ram_y()

- Calculates the Y-coordinate for the current memory usage based on the total and used RAM obtained using sysinfo
- **Parameters**:
- **Returns**: The Y-coordinate for the memory usage point, scaled to fit within the graph's height (12 rows).
- **Usage**: This function is used to determine the vertical position of the memory usage point on the graph everytime memory child needs a sample of memory

---

get_cpu_utilization()

- Retrieves CPU usage statistics by reading and parsing the `/proc/stat` file.
- Read the first line using fgets and updated variables to use formula: long int idle_time = idle + iowait long int non_idle_time = user + nice + system + irq + softirq + steal + guest
  stats.total = idle_time + non_idle_time.

stats.idle = idle_time

- **Returns**: A `CPUStats` struct containing:
    - `total`: The total CPU time (idle + non-idle).
    - `idle`: The idle CPU time.
- **Usage**: This function is used to gather raw CPU data for calculating CPU utilization percentages.

---

get_cpu_percentage(CPUStats prev, CPUStats curr)

- Calculates the CPU utilization percentage by comparing the difference in CPU times between two samples.
- Uses formula 100.0 * (total_diff - idle_diff) / total_diff
- **Parameters**:
  - `prev`: The previous CPU statistics (total and idle time).
  - `curr`: The current CPU statistics (total and idle time).
- **Returns**: The CPU utilization percentage as a floating-point value.
- **Usage**: This function is used to determine the CPU usage percentage, which is then scaled and plotted on the CPU graph.

---

plot_values(int mrow, int cpurow, info flags)

- Plots memory and CPU utilization over time, updating the graph in real-time.

- **Parameters**:

  - `mrow`: The starting row for the memory graph.
  - `cpurow`: The starting row for the CPU graph.
  - `flags`: A data structure containing flags to enable or disable memory and CPU plotting, as well as the number of samples to plot and the tdelay.

- **Usage**: This function creates pipes for memory and CPU utilization, forks child processes to collect the respective data, and plots the values in real-time on the terminal using a while loop over the write end of the pipes and

using function plot_point. It reads memory and CPU values from the pipes, scales them to their Y-axis, and updates the graphs accordingly. Then closes read end of pipes.

`draw_core(int x, int y)`

- Draws a graphical representation of a CPU core at the specified coordinates.
- **Parameters**:
  - `x`: The X-coordinate for the core's starting position.
  - `y`: The Y-coordinate for the core's starting position.
- **Usage**: This function is used to visually represent each CPU core in a grid-like structure, making it easy to see the number of cores at a glance.

`plot_cores(int cores, float max_frequency, int coresrow)`

- Plots the number of CPU cores and their maximum frequency in a graphical format.
- **Parameters**:
  - `cores`: The total number of CPU cores.
  - `max_frequency`: The maximum frequency of the CPU cores in GHz.
  - `coresrow`: The starting row for the cores graph.
- **Usage**: This function calculates the layout of the cores (using `sqrt(cores)` for height) and draws each core in a grid. It also displays the total number of cores and their maximum frequency.

`show_cores(int coresrow)`

- Creates two pipes and child processes for cores and frequency , each child calls its information function
- Parent uses a while loop to read from the pipes and get number of cores and max frequency
- **Parameters**:
  - `coresrow`: The starting row for the cores graph.
- **Usage**: This function reads the number of cores and their frequencies, then calls `plot_cores` to display them in a structured format.

---

show(info Flags)

- The main function to display the system monitor, including memory, CPU, and core information.
- **Parameters**:
- struct info with fields:
  - `samples`: The number of samples to display.
  - `tdelay`: The delay between samples in microseconds.
  - `memory`: A flag to enable or disable memory monitoring.
  - `cpu`: A flag to enable or disable CPU monitoring.
  - `cores`: A flag to enable or disable core information display.
- **Usage**: This function initializes the display, draws the axes for memory and CPU graphs, and calls the necessary functions to plot data or display core information based on the flags provided. Keeps track of the starting rows for Memory, CPU, and Cores, updating them if memory is displayed.

---

process_flags(int argc, char **argv, info *flags)

- Processes command-line arguments to configure the `info` structure.
- Ensures that `samples` and `tdelay` are only set once using control variables (`s` and `t`).
- **Parameters**:

    - `argc`: The number of command-line arguments.
    - `argv`: An array of command-line arguments.
    - `flags`: A pointer to an `info` structure where parsed values are stored.
- **Processing Logic**:
    - Checks if the first (`argv[1]`) or second (`argv[2]`) argument is an integer using `strtol`.
    - If valid, assigns `argv[1]` to `samples` and `argv[2]` to `tdelay`, ensuring each is only set once by changing s and t to 1 signalling that samples or tdelay have been changed.
    - Recognizes `--samples=N` and `--tdelay=T`, extracts `N` and `T` using `strtol`, and assigns values only if they have not been set before(if s and t = 0).
    - Handles exclusive flags:
        - `--memory`: Disables `cpu` and `cores` or enables memory if it was turned off
        - `--cpu`: Disables `memory` and `cores` or enables cpu if it was turned off
        - `--cores`: Disables `memory` and `cpu` or enables cores if it was turned off
    - If an invalid format is detected, prints an error message and exits.
- **Usage**:
    - Example 1:

```
./program 10 5 --cpu
```

        - `samples = 10`
        - `tdelay = 5`
        - `cpu = 1, memory = 0, cores = 0`
    - Example 2:

```
./program --samples=15 --tdelay=3 --memory
```

        - `samples = 15`
        - `tdelay = 3`
        - `memory = 1, cpu = 0, cores = 0`

---

memory_child(int mem_pipe[], info flags)

- Collects memory utilization data and writes it to the `mem_pipe`.
- **Parameters**:
  - `mem_pipe[]`: An array of 2 integers representing a pipe, where:
    - `mem_pipe[0]` is the read end of the pipe (closed in this function).
    - `mem_pipe[1]` is the write end of the pipe (used to send the memory utilization values).
  - `flags`: A data structure containing the number of samples (`samples`) and the time delay (`tdelay`) between each sample.
- **Usage**: This function calculates memory utilization at regular intervals, writing each sample to the `mem_pipe`. It uses `get_ram_y()` to get the memory utilization value and sends it to the pipe. The function sleeps for the specified delay (`tdelay`) between samples using usleep.

---

cpu_child(int cpu_pipe[], info flags)

- Collects CPU utilization data and writes it to the `cpu_pipe`.
- **Parameters**:
  - `cpu_pipe[]`: An array of 2 integers representing a pipe, where:
    - `cpu_pipe[0]` is the read end of the pipe (closed in this function).

        - `cpu_pipe[1]` is the write end of the pipe (used to send the CPU utilization values).
    - `flags`: A data structure containing the number of samples (`samples`) and the time delay (`tdelay`) between each sample.

- **Usage**: This function calculates CPU utilization at regular intervals, writing each sample to the `cpu_pipe`. It uses `get_cpu_utilization()` to get the current CPU stats and `get_cpu_percentage(prev_cpu, curr)` to calculate the utilization using 2 known cpu stats. The function sleeps for the specified delay (`tdelay`) between samples using usleep.

`cores_child(int cores_pipe[])`

- Has Parameter `cores_pipe`, an array of 2 integers representing a pipe
- Close read end `cores_pipe[0]`
- Use `fopen` to open the file `/proc/cpuinfo` to get the number of CPU cores
- Count the number of occurrences of "processor" in the file to determine the number of cores
- Write the number of cores to `cores_pipe[1]`
- Close the write end of the pipe and the file

`freq_child(int freq_pipe[])`

- Has Parameter `freq_pipe`, an array of 2 integers representing a pipe.
- Closes the read end `freq_pipe[0]`.
- Uses `fscanf` to extract the maximum frequency from the file `/sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_max_freq`.
- Writes the extracted frequency value to `freq_pipe[1]`.
- Closes the write end of the pipe and the file.

---

handle_z(int signal)

- Simply returns to ignore Ctrl Z

---

handle_c(int signal)

- Prompts user for an input after printing "Do you want to exit? (y/n)"
- If answer is Y, clear screen and exit program
- Else, clear the question line, return and ignore

---

main(int argc, char **argv)

- Uses sigaction to set signal handlers of Ctrl C and Ctrl Z
- Parses command-line arguments by calling the process_flags function and starts the system monitoring tool.
- Uses data structure info to store these values:
- samples,tdelay,memory,cpu,cores , the last three are flags to keep track of which metrics to show
- Sets these to their default values: 20,500000,1,1,1 in respective order where 1 means show
- **Parameters**:
  - `argc`: The number of command-line arguments.
  - `argv`: An array of command-line arguments.
- Calls show function and passes it the info variable so that it can display the correct metrics
- **Usage**: This is the entry point of the program. It processes user inputs (e.g., number of samples, delay, and which metrics to display) and calls the `show` function to start the monitoring process.
