CS310 PROJECT

MINI TASK MANAGER FOR LINEX

Sayanth Sunil
2303133

Instructions : since it works using /proc file system ,this will only work on linux operating system and on WSL

USER INTERFACE

FULL VIEW

CPU and Core Monitoring

Overall CPU Usage:

3.4%

Usage Per Core:

Core 0: 1.0%
Core 1: 4.1%
Core 2: 8.5%
Core 3: 5.6%
Core 4: 3.1%
Core 5: 2.1%
Core 6: 0.5%
Core 7: 1.0%

Process Monitoring

All Processes | User Processes | System Processes

Pid	User	Cpu	Mem	Status	Command
833	root	2050.0	157.3M	S (sleeping)	/usr/lib/xorg/Xorg :0 -seat seat0 -auth /var/run/lightdm/r
16958	1000	850.0	40.2M	S (sleeping)	xfce4-screenshooter
1166	1000	450.0	122.6M	S (sleeping)	xfwm4 --display :0.0 --sm-client-id 20eff90d6-b87a-44€
4840	1000	450.0	24.7M	R (running)	python3 23.py
31	root	50.0	0.0M	S (sleeping)	migration/2
799	root	50.0	2.5M	S (sleeping)	/usr/sbin/VBoxService
1102	1000	50.0	3.0M	S (sleeping)	/usr/bin/VBoxClient --draganddrop
1232	1000	50.0	87.6M	S (sleeping)	/usr/lib/x86_64-linux-gnu/xfce4/panel/wrapper-2.0 /usr/l
1	root	0.0	12.8M	S (sleeping)	/sbin/init splash
2	root	0.0	0.0M	S (sleeping)	kthreadd
3	root	0.0	0.0M	S (sleeping)	pool_workqueue_release
4	root	0.0	0.0M	I (idle)	kworker/R-rcu_gp
5	root	0.0	0.0M	I (idle)	kworker/R-sync_wq
6	root	0.0	0.0M	I (idle)	kworker/R-slub_flushwq
7	root	0.0	0.0M	I (idle)	kworker/R-netns
10	root	0.0	0.0M	I (idle)	kworker/0:0H-kblockd
12	root	0.0	0.0M	I (idle)	kworker/R-mm_percpu_wq
13	root	0.0	0.0M	I (idle)	rcu_tasks_kthread
14	root	0.0	0.0M	I (idle)	rcu_tasks_rude_kthread
15	root	0.0	0.0M	I (idle)	rcu_tasks_trace_kthread
16	root	0.0	0.0M	S (sleeping)	ksoftirqd/0
17	root	0.0	0.0M	I (idle)	rcu_preempt
18	root	0.0	0.0M	S (sleeping)	rcu_exp_par_gp_kthread_worker/0
19	root	0.0	0.0M	S (sleeping)	rcu_exp_gp_kthread_worker
20	root	0.0	0.0M	S (sleeping)	migration/0
21	root	0.0	0.0M	S (sleeping)	idle_inject/0

Memory Monitoring

RAM Usage:

Used: 984.1 MB / 7946.7 MB (12.4%)

Swap Usage:

Used: 0.0 MB / 1024.0 MB (0.0%)

CPU AND CORE MONITORING PART

1. Overall CPU usage percentage will be showed here
2. Useage percentage of each core will be shown here

This is where the Operating System concepts come in. To calculate CPU usage, I read the `/proc/stat` file.

Function: `get_system_cpu_times`

- `with open('/proc/stat', 'r') as f`: I open the virtual file `/proc/stat`.
- `lines = f.readlines()`: I read the raw data.
- `overall_times`: The first line of this file represents the aggregate CPU usage.
- `core_times`: The subsequent lines (cpu0, cpu1, etc.) represent individual cores.

Function: `update_cpu_info`

- `prev_total = sum(prev_overall)`: I calculate the total time the CPU has spent doing *anything* in the previous check.
- `curr_total = sum(curr_overall)`: I do the same for the current check.
- `total_delta`: I find the difference (delta). This represents the total time passed.
- `idle_delta`: I calculate how much of that time the CPU was doing *nothing* (idle).
- `cpu_usage`: The formula is simple: **(Total Time - Idle Time) / Total Time**. This gives us the usage percentage.
- I repeat this exact loop for every single core to update the individual progress bars.

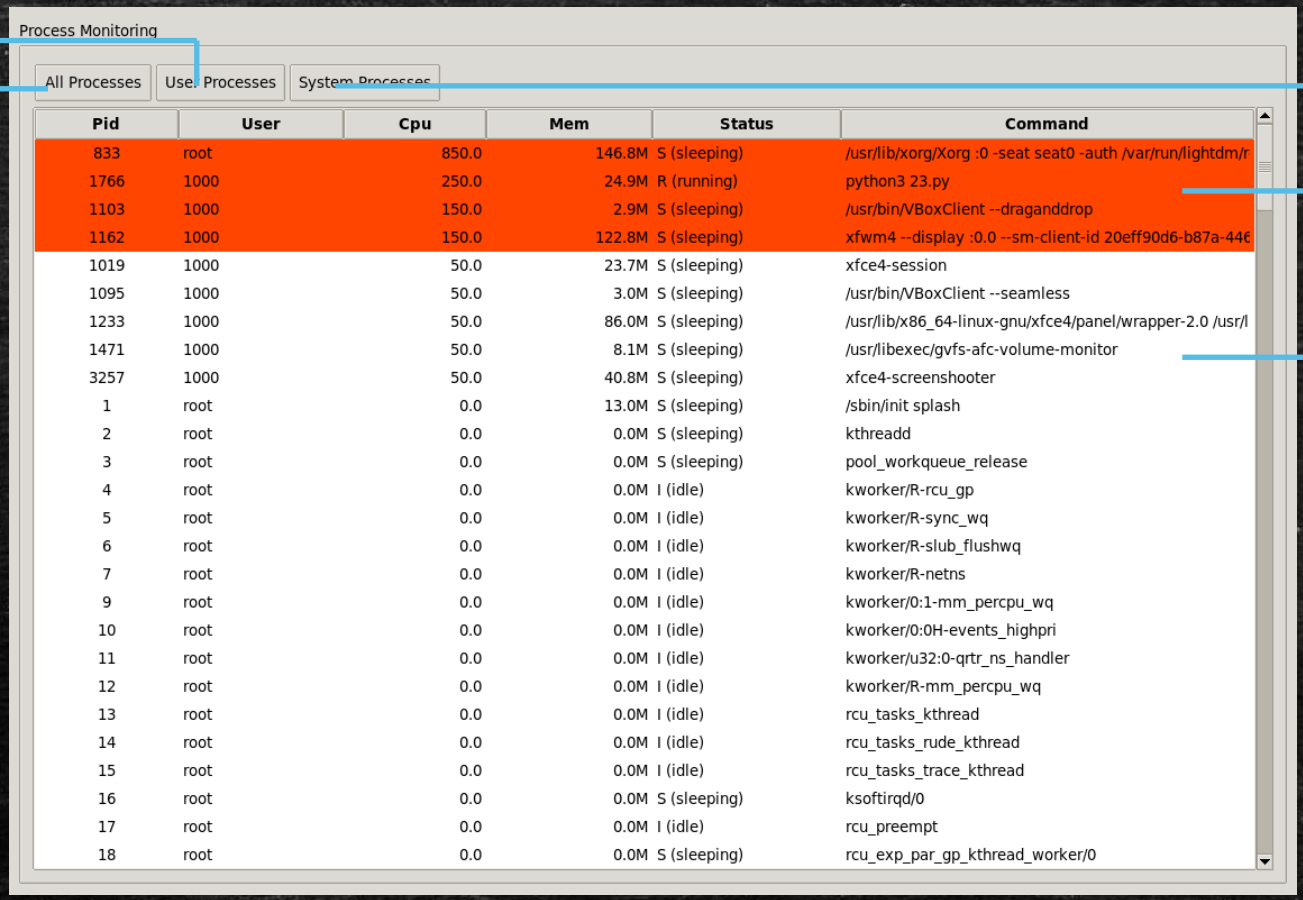
Code

```python
def get_system_cpu_times(self):
    """Reads /proc/stat to get total and per-core CPU times."""
    try:
        with open('/proc/stat', 'r') as f:
            lines = f.readlines()

        # Overall is the first line, 'cpu'
        overall_times = [int(p) for p in lines[0].split()[1:]]

        # Per-core lines start with 'cpuX'
        core_times = []
        for i in range(self.cpu_core_count):
            core_line = lines[i + 1]
            core_times.append([int(p) for p in core_line.split()[1:]])
        return {'overall': overall_times, 'cores': core_times}
    except (IOError, ValueError):
        return {'overall': [0]*10, 'cores': [[0]*10 for _ in range(self.cpu_core_count)]}
```

```python
def update_cpu_info(self):
    current_sys_cpu_times = self.get_system_cpu_times()

    # --- Overall CPU Calculation Logic ---
    prev_overall = self.prev_sys_cpu_times['overall']
    curr_overall = current_sys_cpu_times['overall']

    prev_total = sum(prev_overall)
    curr_total = sum(curr_overall)
    prev_idle = prev_overall[3]
    curr_idle = curr_overall[3]

    total_delta = curr_total - prev_total
    idle_delta = curr_idle - prev_idle

    cpu_usage = 0.0
    if total_delta > 0:
        cpu_usage = 100.0 * (1.0 - idle_delta / total_delta)

    self.overall_cpu_bar['value'] = cpu_usage
    self.overall_cpu_label['text'] = f"{cpu_usage:.1f}%"
```

```
# --- Per-Core CPU Calculation Logic ---
for i in range(self.cpu_core_count):
    prev_core = self.prev_sys_cpu_times['cores'][i]
    curr_core = current_sys_cpu_times['cores'][i]

    prev_total_core = sum(prev_core)
    curr_total_core = sum(curr_core)
    prev_idle_core = prev_core[3]
    curr_idle_core = curr_core[3]

    total_delta_core = curr_total_core - prev_total_core
    idle_delta_core = curr_idle_core - prev_idle_core

    core_usage = 0.0
    if total_delta_core > 0:
        core_usage = 100.0 * (1.0 - idle_delta_core / total_delta_core)

    self.core_bars[i]['value'] = core_usage
    self.core_labels[i]['text'] = f"{core_usage:.1f}%"

self.prev_sys_cpu_times = current_sys_cpu_times
```

PROCESS MONITORING PART



1.The processes which exceed the threshold for CPU is highlighted in orange colour and will be listed in the top
2.All other processes happening will be listed down
3.This is default ,this button will show all processes
4.This button will filter and show the user processes
5.This button will filter and shows the system processes

I iterate through the `/proc` directory to find every running process.

Function: `get_process_data`

- **`os.listdir('/proc')`:** This lists every folder in `/proc`. If the folder name is a number (isdigit), it represents a Process ID (PID).
- Reading `/proc/[pid]/stat`: "I read this file to get the process's specific CPU time (user time + system time)."
- **`cpu_percent`**: Just like with the global CPU, I calculate the percentage by comparing how much time this specific process used vs. the total system time passed.
- Reading `/proc/[pid]/status`: This file is human-readable. I use it to grab the Process Name, State (Sleeping/Running), User ID (UID), and RAM usage .
- `current_proc_times`: I store the times in a dictionary so I can compare them in the next update cycle.

code

```
def set_filter(self, filter_type):
    self.current_filter = filter_type

def get_process_data(self):
    processes = []
    pids = [pid for pid in os.listdir('/proc') if pid.isdigit()]

    current_proc_times = defaultdict(int)

    # Get total system CPU time delta
    prev_total_sys_time = sum(self.prev_sys_cpu_times['overall'])
    curr_total_sys_time = sum(self.get_system_cpu_times()['overall'])
    sys_time_delta = curr_total_sys_time - prev_total_sys_time
    if sys_time_delta == 0: sys_time_delta = 1
```

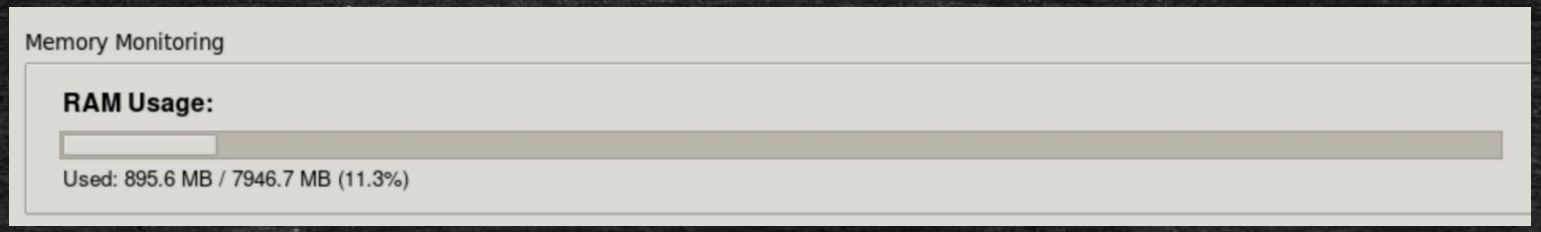
```
for pid in pids:
    try:
        # CPU time from /proc/[pid]/stat
        with open(f'/proc/{pid}/stat', 'r') as f:
            stat_data = f.read().split()
        proc_total_time = int(stat_data[13]) + int(stat_data[14])
        current_proc_times[pid] = proc_total_time

        proc_time_delta = proc_total_time - self.prev_proc_times.get(pid, 0)
        cpu_percent = 100.0 * (proc_time_delta / sys_time_delta)

        # Other info from /proc/[pid]/status
        proc_info = {}
        with open(f'/proc/{pid}/status', 'r') as f:
            for line in f:
                key, value = line.split(':', 1)
                if key.strip() in ['Name', 'State', 'Uid', 'VmRSS']:
                    proc_info[key.strip()] = value.strip()
```

```
            # Command from /proc/[pid]/cmdline
            cmdline = proc_info.get('Name', 'N/A')
            with open(f'/proc/{pid}/cmdline', 'r') as f:
                full_cmd = f.read().replace('\0', ' ').strip()
                if full_cmd: cmdline = full_cmd

            mem_kb = int(proc_info.get('VmRSS', '0 kB').replace('kB', '').strip())
            uid = int(proc_info.get('Uid', '0').split()[0])

            proc_data = {
                'pid': pid, 'user': 'root' if uid == 0 else str(uid), 'cpu': f"{cpu_percent:.1f}",
                'mem': f"{mem_kb/1024:.1f}M", 'status': proc_info.get('State', '?'),
                'command': cmdline, 'uid': uid, 'mem_raw': mem_kb, 'cpu_raw': cpu_percent
            }
            processes.append(proc_data)
        except (IOError, IndexError, ValueError):
            continue

    self.prev_proc_times = current_proc_times
    return processes
```

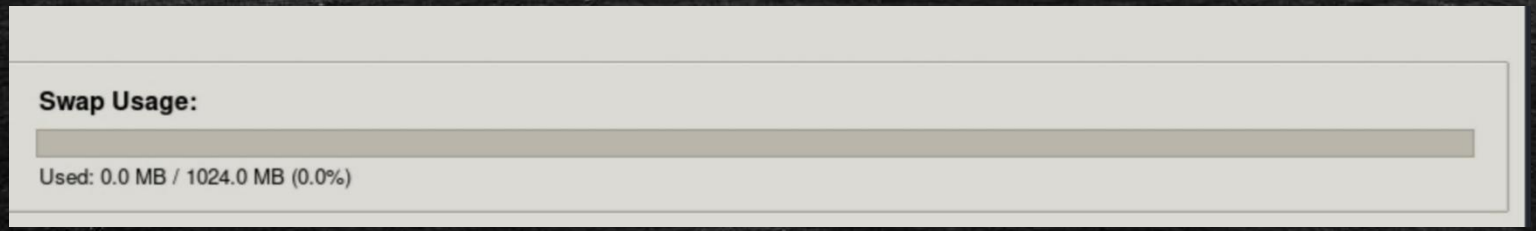
```python
def update_process_info(self):
    # Clear existing entries
    for i in self.tree.get_children():
        self.tree.delete(i)

    processes = self.get_process_data()

    # Apply filter
    if self.current_filter == 'user':
        processes = [p for p in processes if p['uid'] >= USER_UID_MIN]
    elif self.current_filter == 'system':
        processes = [p for p in processes if p['uid'] < USER_UID_MIN]

    # Sort by CPU
    processes.sort(key=lambda p: p['cpu_raw'], reverse=True)

    # Populate the treeview
    for proc in processes:
        tags = []
        if proc['cpu_raw'] > CPU_HIGH_THRESHOLD:
            tags.append('high_cpu')
        if proc['mem_raw'] > MEM_HIGH_THRESHOLD_KB:
            tags.append('high_mem')

        self.tree.insert('', 'end', values=(
            proc['pid'], proc['user'], proc['cpu'], proc['mem'], proc['status'], proc['command']
        ), tags=tuple(tags))
```

MEMORY MONITORING PART

1



2



1. Overall RAM usage will be showed here
2. This refers to the amount of Swap Space currently being used by the operating system.

For memory, I parse the `/proc/meminfo` file. This file contains keys like `MemTotal` and `MemAvailable`.

Function: `update_memory_info`

- `meminfo = { ... }`: This list comprehension reads the file and creates a dictionary so I can easily grab values like 'MemTotal'."
- `mem_used = mem_total - mem_available`: Linux doesn't explicitly give 'Used' memory in this file, so I calculate it by subtracting available memory from the total.
- `swap_total`: I do the same for Swap memory. Swap is space on the hard drive used as emergency RAM."

code

```python
def update_memory_info(self):
    try:
        with open('/proc/meminfo', 'r') as f:
            lines = f.readlines()

        meminfo = {line.split(':')[0]: int(line.split(':')[1].strip().split()[0]) for line in lines}

        # RAM Calculation
        mem_total = meminfo.get('MemTotal', 1)
        mem_available = meminfo.get('MemAvailable', 1)
        mem_used = mem_total - mem_available
        mem_percent = (mem_used / mem_total) * 100

        self.mem_labels["RAM"]['bar']['value'] = mem_percent
        self.mem_labels["RAM"]['label']['text'] = f"Used: {mem_used/1024:.1f} MB / {mem_total/1024:.1f} MB ({mem_percent:.1f}%)"

        # Swap Calculation
        swap_total = meminfo.get('SwapTotal', 1)
        swap_free = meminfo.get('SwapFree', 1)
        swap_used = swap_total - swap_free
        swap_percent = 0
        if swap_total > 0:
            swap_percent = (swap_used / swap_total) * 100

        self.mem_labels["Swap"]['bar']['value'] = swap_percent
        self.mem_labels["Swap"]['label']['text'] = f"Used: {swap_used/1024:.1f} MB / {swap_total/1024:.1f} MB ({swap_percent:.1f}%)"

    except (IOError, ValueError):
        pass
```

UPDATING THE LIST AND LOOP

Finally, filter and sort the data to display it.

Function: `update_process_info`

- `if self.current_filter == 'user':` check the UID. If it is 1000 or greater, it's a User process. If it's less than 1000 (usually 0), it's a System/Root process.
- `processes.sort(...):` sort the list so the most CPU-hungry processes appear at the top.
- `self.tree.insert`: I add the data to the table.
- **Color Coding:** "Here I check if CPU > 75% or Memory > 500MB. If so, I tag the row to turn it orange or blue."

Function: `update_all_info`

- `self.root.after(UPDATE_INTERVAL_MS, ...):` "This creates a recursive loop. Every 2 seconds, it calls itself again to refresh the dashboard."

ACKNOWLEDGEMENT

The graphical user interface setup is built by using Ai tools like Gemini and Chat gpt

THANK YOU