CAPSTONE PROJECT REPORT

Title: SYSTEM MONITOR TOOL

Student Name: Indraneel Chakraborty

Department: Computer Science & Information Technology

Tools & Technologies Used: C++, ncurses, CMake, Linux/WSL, GCC, GitHub.

Abstract

This project is a terminal-based System Monitor Tool developed using C++. It displays real-time system metrics such as CPU utilization, memory usage, uptime, and process information. The tool uses the ncurses library to render a dynamic text-based user interface and reads data directly from the Linux /proc filesystem. The objective is to provide an efficient, lightweight, and educational tool for understanding Linux process and resource management internals.

Objective

- Develop a real-time system monitoring application using C++.
- Explore and parse data from the Linux /proc filesystem.
- Visualize CPU, memory, and process information dynamically using ncurses. • Demonstrate modular, object-oriented design in system-level programming.

Problem Statement

Most GUI-based system monitors such as GNOME System Monitor or top commands consume additional system resources or lack extensibility for educational use. This project aims to create a lightweight, customizable tool that can operate entirely within a terminal environment, providing developers a clear understanding of how system data is collected and processed in Linux.

Existing System

Existing tools like htop and top provide process monitoring capabilities but are monolithic and not easily extensible. Moreover, they do not serve as simple educational references for learning Linux system internals or C++ system programming.

Proposed System

The proposed System Monitor Tool is a modular and object-oriented implementation in C++. It reads live data from the /proc filesystem to display metrics like CPU utilization, memory consumption, system uptime, and per-process statistics. The ncurses library provides an efficient

and interactive display within a terminal window. The system is compatible with Linux and Windows Subsystem for Linux (WSL).

Novelty (Own Contribution)

- Built from scratch using object-oriented C++.
- Modular classes for CPU, Process, System, Parser, and Display.
- Real-time data refresh and progress bar visualization.
- Cross-platform compatibility via WSL on Windows.

Modules

- System Module - integrates CPU, memory, and process information and manages overall updates.
- Processor Module - calculates CPU utilization using time difference approach.
- Process Module - retrieves per-process information such as user, RAM, and command.
- Linux Parser Module - reads raw data from /proc and /etc directories.
- NcursesDisplay Module - displays formatted information using ncurses in real-time.

Technologies Used

Component	Technology
Language	C++
UI Library	ncurses
Compiler	GCC
Build System	CMake
Version Control	Git, GitHub
OS	Linux / Ubuntu

Implementation Details

- Reads data from /proc filesystem.
- Uses OOP principles for modularity.
- Displays CPU and memory bars with percentage.
- Lists processes with PID, user, CPU%, RAM, and command.
- Refreshes data every second using threads.

Full Source Code

main.cpp

```
#include <iostream>
```

```
#include "ncurses_display.h"
#include "system.h"
#include "process.h"

int main() {
  System system;
  NCursesDisplay::Display(system, 20);
}
```

all_processes.h

```
#ifndef ALL_PROCESSES_H
#define ALL_PROCESSES_H

#include "process.h"

#include <string>
#include <vector>

using std::string;
using std::vector;

class All_Processes {
private:   long Hertz;
vector<int> current_pids_;
  vector<Process> all_processes_;

  void UpdateProcesses();   void
AddNewProcesses(bool&);
vector<int> ReadFolders();
  void RemoveFinishedProcesses(bool&);

 public:
All_Processes();
  vector<Process>& GetProcesses();
};

#endif
```

all_processes.cpp

```
#include "all_processes.h"
#include <unistd.h>
#include <algorithm>
#include <vector>
#include "linux_parser.h"
#include "process.h"
```

```
using std::sort;
using std::vector;

bool compareProcesses(Process& p1, Process& p2) {
return (p1.RawRam() > p2.RawRam());
};

All_Processes::All_Processes() {
  Hertz = sysconf(_SC_CLK_TCK);
  UpdateProcesses();
}

vector<Process>& All_Processes::GetProcesses() {
UpdateProcesses();
  return all_processes_;
}

void All_Processes::UpdateProcesses() {
  current_pids_ = ReadFolders();

  bool changed = false;

  AddNewProcesses(changed);
  RemoveFinishedProcesses(changed);

  if (changed) {
    sort(all_processes_.begin(), all_processes_.end(), compareProcesses);
  }
}

vector<int> All_Processes::ReadFolders() { return LinuxParser::Pids(); };

void All_Processes::AddNewProcesses(bool& changed) {
for (std::size_t i = 0; i < current_pids_.size(); ++i) {     int
current_pid = current_pids_[i];

    if (std::find_if(all_processes_.begin(), all_processes_.end(),
                     [current_pid](Process& n) {
return n.Pid() == current_pid;                    })
== all_processes_.end()) {       changed =
true;
      Process process(current_pids_[i], Hertz);       all_processes_.emplace_back(process);
    }
  }
```

```
}

void All_Processes::RemoveFinishedProcesses(bool& changed) {
for (size_t i = 0; i < all_processes_.size(); i++) {
   int current_pid = all_processes_[i].Pid();

   if (std::find(current_pids_.begin(), current_pids_.end(), current_pid) ==
current_pids_.end()) {       changed = true;

     all_processes_.erase(all_processes_.begin() + i);
   }
  }
}
```

Format.h

```
#ifndef FORMAT_H
#define FORMAT_H

#include <string>

namespace Format { std::string
Format(int); std::string
ElapsedTime(long times); std::string
KBisMB(float kb);
};  // namespace Format

#endif
```

Format.cpp

```
#include "format.h"
#include <iomanip>
#include <sstream>
#include <string> using
std::string;
using std::to_string;

string Format::Format(int time) {
string timeAsString = to_string(time);
  return string(2 - timeAsString.length(), '0') + timeAsString;
}

string Format::ElapsedTime(long seconds) {
  int   hour   =   0;
int min = 0;    int
sec = 0;
```

```
  hour = seconds / 3600;
seconds = seconds % 3600;
min = seconds / 60;   seconds
= seconds % 60;
  sec = seconds;

  return Format(hour) + ':' + Format(min) + ':' + Format(sec);
}

string Format::KBisMB(float kb) {
float mb = kb / 1024;
std::stringstream mb_stream;
  mb_stream << std::fixed << std::setprecision(1) << mb;
return mb_stream.str();
}
```

Linux_parser.h

```
#ifndef SYSTEM_PARSER_H
#define SYSTEM_PARSER_H

#include <fstream>
#include <regex>
#include <string>

#include "parser_helper.h"

using std::string;
using std::vector;

namespace LinuxParser {
// System float
MemoryUtilization(); long
UpTime(); vector<int>
Pids(); int
TotalProcesses(); int
RunningProcesses();
string OperatingSystem();
string Kernel();
string UserByUID(int);

std::vector<string> CpuUtilization();
};  // namespace LinuxParser

#endif
```

Linux_parser.cpp

```
#include <dirent.h>
#include <sstream>
#include <string>
#include <vector>
#include "linux_parser.h"
#include "parser_consts.h"
#include "parser_helper.h"

using std::stof; using
std::string; using
std::to_string;
using std::vector;

string LinuxParser::OperatingSystem() {
string line;   string key;   string value;
  std::ifstream filestream(ParserConsts::kOSPath);
  if (filestream.is_open()) {     while
(std::getline(filestream, line)) {
std::replace(line.begin(), line.end(), ' ', '_');
std::replace(line.begin(), line.end(), '=', ' ');
std::replace(line.begin(), line.end(), '"', ' ');
std::istringstream linestream(line);       while
(linestream >> key >> value) {        if (key ==
"PRETTY_NAME") {          std::replace(value.begin(),
value.end(), '_', ' ');          return value;
       }
     }
   }
  }
  return value;
}

string LinuxParser::Kernel() {
string os, version, kernel;   string
line;
  std::ifstream stream(ParserConsts::kProcDirectory +
ParserConsts::kVersionFilename);
  if (stream.is_open()) {
std::getline(stream, line);
std::istringstream linestream(line);
   linestream >> os >> version >> kernel;
  }
  return kernel;
}
```

```
vector<int> LinuxParser::Pids() {
  vector<int> pids;
  DIR* directory = opendir(ParserConsts::kProcDirectory.c_str());
struct dirent* file;   while ((file = readdir(directory)) != nullptr) {
// Is this a directory?
    if (file->d_type == DT_DIR) {
string filename(file->d_name);
      if (std::all_of(filename.begin(), filename.end(), isdigit)) {
int pid = stoi(filename);
        pids.push_back(pid);
      }
    }
  }
  closedir(directory);
  return pids;
}

float LinuxParser::MemoryUtilization() {
  float memTotal = ParserHelper::GetValueByKey<int>(
      ParserConsts::filterMemTotalString, ParserConsts::kMeminfoFilename);
float memFree = ParserHelper::GetValueByKey<int>(
      ParserConsts::filterMemFreeString, ParserConsts::kMeminfoFilename);

  float memory = (memTotal - memFree) / memTotal;

  return memory;
}

long LinuxParser::UpTime() {
string line;
  long upTime = ParserHelper::GetValue<long>(ParserConsts::kUptimeFilename);
return upTime;
}

int LinuxParser::TotalProcesses() {
  return ParserHelper::GetValueByKey<int>(ParserConsts::filterProcesses,
                                          ParserConsts::kStatFilename); }

int LinuxParser::RunningProcesses() {
  return ParserHelper::GetValueByKey<int>(ParserConsts::filterRunningProcesses,
                                          ParserConsts::kStatFilename); }

string LinuxParser::UserByUID(int UID) {
  string line, user, x;
int fileUid;
```

```
std::ifstream
filestream(ParserCons
ts::kPasswordPath);
 if (filestream.is_open()) {    while
(std::getline(filestream, line)) {
std::replace(line.begin(), line.end(), ':', ' ');
     std::istringstream linestream(line);
    while (linestream >> user >> x >> fileUid) {
     if (fileUid == UID) {
        return user;
      }
     }
    }
  }
  return user;
}
```

ncurses_display.h

```
#ifndef NCURSES_DISPLAY_H
#define NCURSES_DISPLAY_H
#include <curses.h>
#include "process.h"
#include "system.h"

namespace NCursesDisplay { void Display(System& system,
int n = 10); void DisplaySystem(System& system,
WINDOW* window);
void DisplayProcesses(std::vector<Process> processes, WINDOW* window, int n);
std::string ProgressBar(float percent);
};

#endif
```

ncurses_display.cpp

```
#include <curses.h>
#include <chrono>
#include <string>
#include <thread>
#include <vector>
#include <algorithm>
#include "format.h"
#include "ncurses_display.h"
#include "system.h" using
std::string;
using std::to_string;
```

```
std::string NCursesDisplay::ProgressBar(float percent) {
std::string result{""};   int size{50};
  float bars{percent * size};

 for (int i{0}; i < size; ++i) {
   result += (i <= bars) ? '|' : ' ';
 }

 string display{to_string(percent * 100).substr(0, 4)};  if
(percent < 0.1 || percent == 1.0)
    display = " " + to_string(percent * 100).substr(0, 3);
return result + " " + display + "/100%";
}

void NCursesDisplay::DisplaySystem(System& system, WINDOW* window) {
int row{0};
  mvwprintw(window, ++row, 2, "%s", ("OS: " + system.OperatingSystem()).c_str());
mvwprintw(window, ++row, 2, "%s", ("Kernel: " + system.Kernel()).c_str());
mvwprintw(window, ++row, 2, "CPU: ");   wattron(window, COLOR_PAIR(1));
  mvwprintw(window, row, 10, "%s", ProgressBar(system.Cpu().Utilization()).c_str());
wattroff(window, COLOR_PAIR(1));   mvwprintw(window, ++row, 2, "Memory: ");
wattron(window, COLOR_PAIR(1));
  mvwprintw(window, row, 10, "%s", ProgressBar(system.MemoryUtilization()).c_str());
wattroff(window, COLOR_PAIR(1));
  mvwprintw(window, ++row, 2, "%s", ("Total Processes: " +
to_string(system.TotalProcesses())).c_str());
  mvwprintw(window, ++row, 2, "%s", ("Running Processes: " +
to_string(system.RunningProcesses())).c_str());
mvwprintw(window, ++row, 2, "%s", ("Up Time: " +
Format::ElapsedTime(system.UpTime())).c_str());
wrefresh(window);
}

void NCursesDisplay::DisplayProcesses(std::vector<Process> processes,
                            WINDOW* window, int n) {   int
row{0};   int const pid_column{2};   int const
user_column{9};   int const cpu_column{20};   int const
ram_column{28};   int const time_column{37};   int
const command_column{48};   wattron(window,
COLOR_PAIR(2));   mvwprintw(window, ++row,
pid_column, "PID");   mvwprintw(window, row,
user_column, "USER");   mvwprintw(window, row,
cpu_column, "CPU[%%]");   mvwprintw(window, row,
ram_column, "RAM[MB]");   mvwprintw(window, row,
time_column, "TIME+"); mvwprintw(window, row,
command_column, "COMMAND");
```

```
 wattroff(window, COLOR_PAIR(2));  int to_print =

std::min(static_cast<int>(processes.size()), n);

 int win_x = getmaxx(window);  for (int i = 0; i < to_print; ++i) {    mvwprintw(window,
++row, pid_column, "%s", to_string(processes[i].Pid()).c_str());
   mvwprintw(window, row, user_column, "%s", processes[i].User().c_str());
   float cpu = processes[i].CpuUtilization() * 100;     mvwprintw(window, row,
cpu_column, "%s", to_string(cpu).substr(0, 4).c_str());     mvwprintw(window, row,
ram_column, "%s", processes[i].Ram().c_str());
   mvwprintw(window, row, time_column, "%s",
Format::ElapsedTime(processes[i].UpTime()).c_str());

   int max_cmd_len = std::max(0, win_x - command_column - 2);
string cmd = processes[i].Command();     if
(static_cast<int>(cmd.size()) > max_cmd_len) {      if
(max_cmd_len > 3)
      cmd = cmd.substr(0, max_cmd_len - 3) + "...";
else
      cmd = cmd.substr(0, max_cmd_len);
   }
   mvwprintw(window, row, command_column, "%s", cmd.c_str());
 }
}

void NCursesDisplay::Display(System& system, int n) {
initscr();        noecho();        cbreak();
 start_color();

 int x_max{getmaxx(stdscr)};
 WINDOW* system_window = newwin(9, x_max - 1, 0, 0);
WINDOW* process_window =
    newwin(3 + n, x_max - 1, getmaxy(system_window) + 1, 0);

 while (1) {
   init_pair(1, COLOR_BLUE, COLOR_BLACK);
init_pair(2, COLOR_GREEN, COLOR_BLACK);
   box(system_window, 0, 0);
box(process_window, 0, 0);
   DisplaySystem(system, system_window);
   DisplayProcesses(system.Processes().GetProcesses(), process_window, n);
wrefresh(system_window);
   wrefresh(process_window);
   refresh();
  std::this_thread::sleep_for(std::chrono::seconds(1));
 }
```

```
 endwin();
}
```

parser_consts.h

```
#ifndef CONSTS_PARSER_H
#define CONSTS_PARSER_H

#include <string>

using std::string;

namespace ParserConsts { const string
kProcDirectory{"/proc/"}; const string
kCmdlineFilename{"/cmdline"}; const string
kCpuinfoFilename{"/cpuinfo"}; const string
kStatusFilename{"/status"}; const string
kStatFilename{"/stat"}; const string
kUptimeFilename{"/uptime"}; const string
kMeminfoFilename{"/meminfo"}; const string
kVersionFilename{"/version"}; const string
kOSPath{"/etc/os-release"};
const string kPasswordPath{"/etc/passwd"};

const string filterProcesses("processes"); const string
filterRunningProcesses("procs_running"); const string
filterMemTotalString("MemTotal:"); const string
filterMemFreeString("MemFree:");
const string filterCpu("cpu"); const
string filterUID("Uid:");
const string filterProcMem("VmData:");

}  // namespace ParserConsts

#endif
```

Parser_helper.h

```
#ifndef HELPER_PARSER_H
#define HELPER_PARSER_H
#include <fstream>
#include <regex>
#include <string>
#include "linux_parser.h"
#include "parser_consts.h"

using std::string; using
std::vector;
```

```
namespace
ParserHelper {
template <typename
T>
T GetValueByKey(string const &filter, string const &filename) {
string line, key;
  T value;

  std::ifstream stream(ParserConsts::kProcDirectory + filename);
if (stream.is_open()) {     while (std::getline(stream, line)) {
std::istringstream linestream(line);       while (linestream >> key
>> value) {        if (key == filter) {
         return value;
       }
     }
   }
  }
  return value;
};
template <typename T> T
GetValue(string const &filename) {
string line;
  T value;

  std::ifstream stream(ParserConsts::kProcDirectory + filename);
if (stream.is_open()) {     std::getline(stream, line);
std::istringstream linestream(line);     linestream >> value;
  }
  return value;
};
}  // namespace ParserHelper
#endif
```

process.h

```
#ifndef PROCESS_H
#define PROCESS_H
#include <string>
#include <vector>

using std::string;
using std::vector;

/*
Basic class for Process representation
It contains relevant attributes as shown below
```

```
*/ class Process {
private:   int pid_;
long Hertz_;   float
utime_ = 0.0;   float
stime_ = 0.0;   float
cutime_ = 0.0;   float
cstime_ = 0.0;
 float starttime_ = 0.0;

 vector<string> ReadFile(int);

 public:
Process(int, long);
int Pid();
 string User();   string
Command();   double
CpuUtilization();   float
RawRam();   string
Ram();
 long int UpTime();
};

#endif
```

process.cpp

```
#include <unistd.h>
#include <cctype>
#include <iostream>
#include <iterator>
#include <sstream>
#include <string>
#include <vector>

#include "format.h"
#include "linux_parser.h"
#include "parser_consts.h" #include
"parser_helper.h"
#include "process.h"

using namespace std;
```

```
Process::Process(int pid, long Hertz) : pid_(pid), Hertz_(Hertz) {
 vector<string> cpuNumbers = ReadFile(pid);  utime_ =
 stof(cpuNumbers[13]);
  stime_ = stof(cpuNumbers[14]); cutime_
  = stof(cpuNumbers[15]);
 cstime_ = stof(cpuNumbers[16]);
 starttime_ = stof(cpuNumbers[21]);
}

// Return this process's ID
int Process::Pid() { return pid_; }

// Return this process's CPU utilization
double Process::CpuUtilization() {
long uptime = LinuxParser::UpTime();
  double total_time = utime_ + stime_ + cutime_ + cstime_;

  double seconds = uptime - (starttime_ / Hertz_);
  double cpu_usage = (total_time / Hertz_) / seconds;

  return cpu_usage;
}

// Return the command that generated this process string
Process::Command() {   string cmd =
ParserHelper::GetValue<string>(to_string(pid_) +
                                ParserConsts::kCmdlineFilename);
  size_t maxSize = 50;
if(cmd.size() > maxSize) {
cmd.resize(maxSize - 3);     cmd
= cmd + "...";
  }
  return cmd;
}

float Process::RawRam() {
  float memInKB = ParserHelper::GetValueByKey<float>(
ParserConsts::filterProcMem,       to_string(pid_) +
ParserConsts::kStatusFilename);
  return memInKB;
}
```

```
// Return this process's memory utilization
string Process::Ram() {   float memInKB =
RawRam();
  return Format::KBisMB(memInKB);
}

// Return the user (name) that generated this process
string Process::User() {
 int UID = ParserHelper::GetValueByKey<int>(    ParserConsts::filterUID, to_string(pid_)
+ ParserConsts::kStatusFilename);

 string user = LinuxParser::UserByUID(UID);
 return user;
}

// Return the age of this process (in seconds)
long int Process::UpTime() {   long uptime =
LinuxParser::UpTime();
  long seconds = uptime - (starttime_ / Hertz_);

  return seconds;
}

vector<string> Process::ReadFile(int pid) {
  string line, skip;

  std::ifstream stream(ParserConsts::kProcDirectory + to_string(pid) +
                  ParserConsts::kStatFilename);

  getline(stream, line);   istringstream
linestream(line);   istream_iterator<string>
beg(linestream), end;   vector<string>
cpuNumbers(beg, end);   return cpuNumbers;
};
```

processor.h

```
#ifndef PROCESSOR_H
#define PROCESSOR_H

#include <string>
#include <vector>

using std::string;
using std::vector;
```

```
class Processor {
private:
  int previdle;
int previowait;
int prevuser;
int prevnice;
int prevsystem;
int previrq;  int
prevsoftirq;  int
prevsteal; void
AssignPrevValue
s(vector<double
>);
vector<double>
ReadFile();

 public:   double
Utilization();
};
```

#endif **processor.cpp**

```
#include "processor.h"
#include <sstream>
#include <string>
#include <vector>

#include "parser_consts.h"
#include "parser_helper.h"

using std::string;
using std::vector;

double Processor::Utilization() {
vector<double> values = ReadFile();
double user = values[0];   double
nice = values[1];   double system =
values[2];   double idle = values[3];
double iowait = values[4];   double
irq = values[5];   double softirq =
values[6];
  double steal = values[7];

  double PrevIdle = previdle + previowait;
  double Idle = idle + iowait;
```

```
  double PrevNonIdle =
     prevuser + prevnice + prevsystem + previrq + prevsoftirq + prevsteal;
double NonIdle = user + nice + system + irq + softirq + steal;

  double PrevTotal = PrevIdle + PrevNonIdle;
double Total = Idle + NonIdle;

  double totald = Total - PrevTotal;

  double idled = Idle - PrevIdle;

 double CPU_Percentage = (totald - idled) / totald;

 AssignPrevValues(values);
  return CPU_Percentage;
}

void Processor::AssignPrevValues(vector<double> newValues) {
prevuser = newValues[0];   prevnice = newValues[1];
prevsystem = newValues[2];   previdle = newValues[3];
previowait = newValues[4];   previrq = newValues[5];
prevsoftirq = newValues[6];   prevsteal = newValues[7];
}

vector<double> Processor::ReadFile() {
  string line, key;   double value;   vector<double>
cpuNumbers;   std::ifstream
stream(ParserConsts::kProcDirectory +
                  ParserConsts::kStatFilename);
if (stream.is_open()) {     while
(std::getline(stream, line)) {
std::istringstream linestream(line);
while (linestream >> key) {        if (key ==
ParserConsts::filterCpu) {          while
(linestream >> value) {
cpuNumbers.emplace_back(value);
        }
       }
     }
    }
  }
  return cpuNumbers;
}
```

system.h

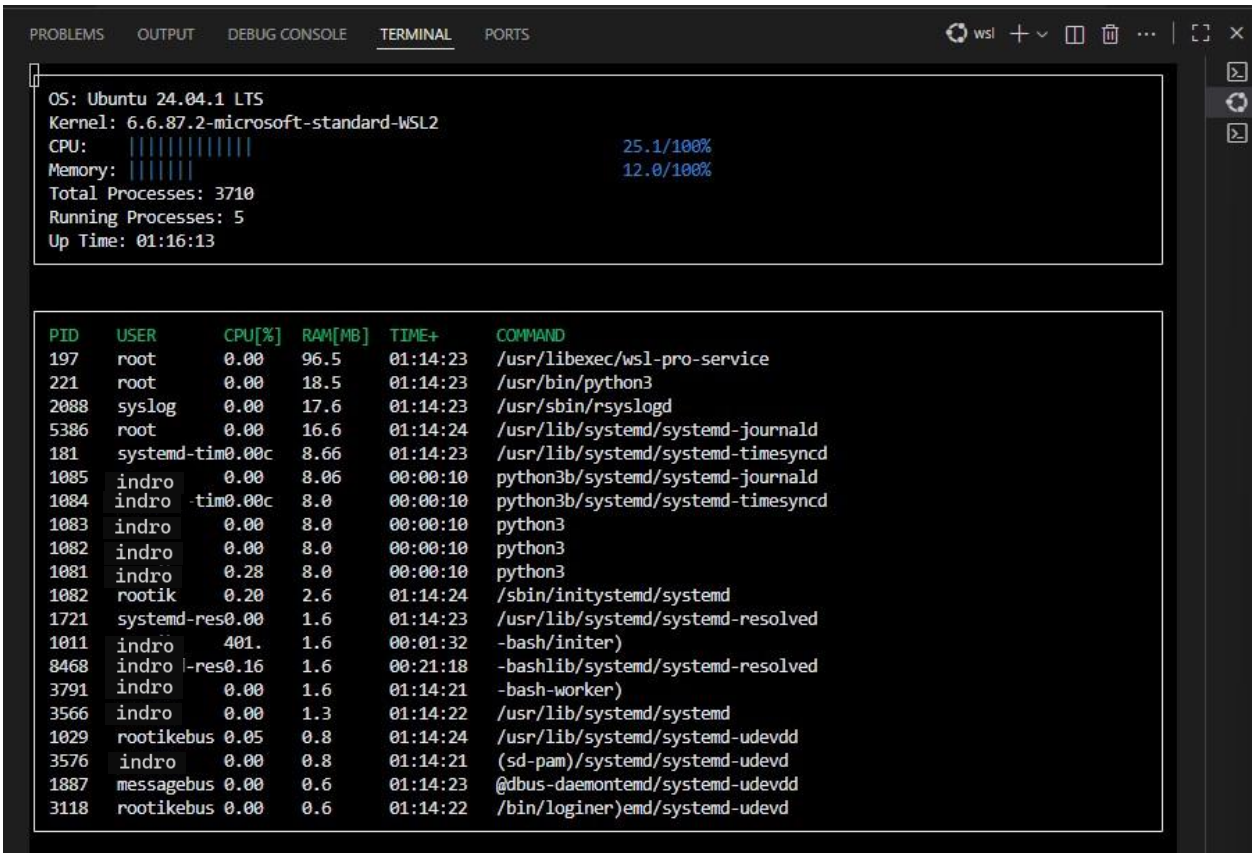
```
#ifndef SYSTEM_H
#define SYSTEM_H
#include <string>
#include <vector>
#include "process.h"
#include "processor.h"
#include "all_processes.h"

class System {
public:
 Processor& Cpu();
 All_Processes& Processes();
  float MemoryUtilization();         long
   UpTime();
  int TotalProcesses();
int RunningProcesses();
std::string Kernel();
  std::string OperatingSystem();

 private:
Processor cpu_;
  All_Processes processes_;
};

#endif
```

system.cpp

```
#include <string>
#include <vector>
#include "linux_parser.h"
#include "process.h"
#include "processor.h"
#include "all_processes.h"
#include "system.h"
#include "format.h"

using std::string;
using std::vector;

Processor& System::Cpu() { return cpu_; }

All_Processes& System::Processes() { return processes_; }

string System::Kernel() { return string(LinuxParser::Kernel()); }
```

```
float System::MemoryUtilization() { return LinuxParser::MemoryUtilization(); }

string System::OperatingSystem() { return LinuxParser::OperatingSystem(); }

int System::RunningProcesses() { return LinuxParser::RunningProcesses(); }

int System::TotalProcesses() { return LinuxParser::TotalProcesses(); }

long int System::UpTime() { return LinuxParser::UpTime(); }
```

Screenshots



Result

Successfully implemented a responsive and lightweight Linux System Monitor that displays real-time CPU, memory, and process statistics.

Conclusion

This project demonstrates how system-level data can be captured and visualized using standard Linux interfaces. It bridges theory (OS concepts) with hands-on C++ programming.