```
CC = gcc

CFLAGS = -Wall -Werror

all: my_program

my_program: sys_stats.o modular.o

	$(CC) -o $@ $^

sys_stats.o: sys_stats.c

	$(CC) $(CFLAGS) -c -o $@ $<

modular.o: modular.c

	$(CC) $(CFLAGS) -c -o $@ $<

clean:

	rm -f my_program *.o
```

How to run:

1. Have to two files ready downloaded

Use: ***gcc sys_stats.c modular.c -o my_program***

2. Run the program
3. ./my_program + CLA etc….

How problem solving approach: Each task (CPU utilization, memory utilization, and user sessions) is handled by a separate child process created through `fork()`. This design allows each task to be executed concurrently:
By dedicating a separate process for each task, the program isolates the execution contexts. This separation enhances maintainability and scalability.
Concurrent execution is inherent as each child process can run independently.
Pipes are established between the parent process and each child process. After gathering the required information, each child process writes it to the respective pipe. The parent process then reads from these pipes, ensuring that data is correctly relayed.

The parent process uses `waitpid()` to synchronize with child processes, ensuring it only attempts to read from the pipes after the child processes have completed their execution and written data to the pipes.
The program checks for potential errors in critical operations like `fork()`, `pipe()`, and `read()`. This robustness is crucial for building reliable applications.
Properly closing unused pipe ends in both child and parent processes helps prevent resource leaks and ensures that EOF (End-of-File) conditions are correc as child processes complete their execution.
The display functions (`display_system_info`, `display_system_info_system`, `graphical_system_info`, `display_user_info`, and `display_system_info_sequential`) are adapted to accept the gathered data as parameters. This design allows them to directly use the data read from the pipes, facilitating a clean separation between data gathering and presentation logic.
The actual data gathering happens concurrently due to the earlier forking of child processes. The sequential calls in `main` are for reading from the pipes and displaying the information, which happens after the concurrent data gathering phase.

Overview of Functions(Including documentation and how to use/run for each function and in total)

- ```
  void gather_cpu_info(char *cpu_info_buffer, int buffer_size, int tdelay, double *cpu_usage);
  ```
- gather_cpu_info is a function designed to calculate and return the overall CPU usage percentage of a system. The function's main role is to read and analyze CPU time metrics from the /proc/stat file on Linux systems.

Resource used: https://www.idnt.net/en-US/kb/941772
Where I learned about the different cpu lines, which is (in default order):
User: Time spent with normal processing in user mode.
Nice: Time spent with nice processes in user mode.
System: Time spent running in kernel mode.
idle: Time spent in vacations twiddling thumbs.
Iowait: Time spent waiting for I/O to complete.(Considered idle time too)
Irq: Time spent serving hardware interrupts.
Softirq: Time spent serving software interrupts.
Also introduced static long idle_time, total_time,prev_idle_time,prev_total_time
How It Works:
File Reading: Opens the /proc/stat file to read CPU time statistics.
Parsing: Reads from /proc/stat, parses the relevant times using strtok (`#include <string.h>`)
to tokenize the string and strtol to convert these tokens to long integers.
  ```

https://www.geeksforgeeks.org/strtok-strtok_r-functions-c-examples/ (learned the use of strtok)
https://www.tutorialspoint.com/c_standard_library/c_function_strtol.htm (learned use of strol)
Error parsing using stderr. stderr link
Then assign each parsed numeric value to a variable that represents a specific CPU time metric using switch command. switch link
After successfully reads from /proc/stat, and successfully assign the parsed value to the corresponding variable,
set idle_time = idle
total_time = user + nice + system + idle + iowait + irq + softirq
Use static variables to keep track of the prev_idle_time and prev_total_time to calculate the difference between the current and previous readings.
Lastly, assign the calculated CPU usage percentage to the variable pointed by cpu_usage.
Used double *cpu_usage: A pointer to a double where the function will store the calculated CPU usage percentage.
Error Handling : If /proc/stat not open or error in parsing the file, sets *cpu_usage to 0 and returns. It uses standard error output to report these issues.
How to use:

```
double cpu_usage;
char cpu_info_buffer[256];
gather_cpu_info(cpu_info_buffer, sizeof(cpu_info_buffer), config.time_delay,
&cpu_usage);
```

- ```
  void gather_memory_info(long *phys_used_mem, long *total_phys_mem, long
  *virt_used_mem, long *total_virt_mem, char *memory_info_buffer, size_t
  buffer_size);
  ```

Calculates and reports both physical and virtual memory usage by reading from /proc/meminfo, which introduces the following long parameters mem_total, mem_free, buffers, and cached swap_total, swap_free.Char pointer endptr and char buffer.
Resource used link: meminfo
Also parameter includes:
->long *phys_used_mem: Pointer to a long where the function will store the calculated used physical memory (in kilobytes).
->long *total_phys_mem: Pointer to a long where the function will store the total physical memory available (in kilobytes).
->long *virt_used_mem: Pointer to a long where the function will store the calculated used virtual memory (swap usage, in kilobytes).
->long *total_virt_mem: Pointer to a long where the function will store the total virtual memory available (total swap space, in kilobytes).
  ```

First opens /proc/meminfo to access memory statistics. Then reads and parses the lines of /proc/meminfo, which includes total memory (MemTotal), free memory (MemFree), buffers (Buffers), cached memory (Cached), total swap space (SwapTotal), and free swap space (SwapFree). Reads the file line by line, stores it in the buffer declared above, and checks for specific memory-related keywords. When it finds a line that matches one of these keywords, it extracts the numerical value following the keyword. All mentioned above are converted from a string to a long integer using strtol. strtol
Calculation:

```
total_tick[1] = fields[0] + fields[1] + fields[2] + fields[3] + fields[4] +
fields[5] + fields[6];
total_idle[1] = fields[3]; // idle ticks count

// Calculate CPU usage
total_usage[0] = total_tick[0] - total_idle[0];
total_usage[1] = total_tick[1] - total_idle[1];
*cpu_usage = 100.0 * (total_usage[1] - total_usage[0]) / (total_tick[1] -
total_tick[0]);
```

Error handle:reports errors to stderr if /proc/meminfo cannot be opened or if errors in parsing the memory information.Cases includes: 1.Conversion error: if endptr points to a character not (\0), ( ), (\n), indicates string contains additional characters beyond the number,then can't be parsed as part of the number. 2.want to skip linesdon't need, such as the VmallocTotal, by using strncmp(buffer, "VmallocTotal:", 13): Compares the first 13 characters of buffer (the current line read from /proc/meminfo) with the string "VmallocTotal:".If such occurs, returns or continue depending on the situation.
How to use: gather_memory_info(&physUsedMem, &totalPhysMem, &virtUsedMem, &totalVirtMem);
How to use: `long phys_used_mem, total_phys_mem, virt_used_mem, total_virt_mem;`

```
char memory_info_buffer[256];
gather_memory_info(&phys_used_mem, &total_phys_mem, &virt_used_mem,
&total_virt_mem, memory_info_buffer, sizeof(memory_info_buffer));
```

- read_uptime(char *uptime_str, size_t max_size): Reads system uptime from /proc/uptime and formats it into a readable string, provides the time the system has been up and running in seconds, converts this duration into a more human-readable format, specifying the uptime in days, hours, minutes, and seconds.

Parameter
-> char *uptime_str: A pointer to a character array where the formatted uptime string will be stored.
-> size_t max_size: The maximum size of the uptime_str buffer, ensuring the function does not write beyond this buffer.

Opens and reads /proc/uptime to access the system's uptime data, which contains a single line with two numbers. First number is total number of seconds the system has been up, and the second number is the amount of time the system has been idle.
Then, reads the line from /proc/uptime and use strtod to convert the uptime from a string representation in seconds to a double. strtod link
Then, Converts to a readable format with days, hours, minutes, and seconds from the total uptime seconds. Formats the calculated uptime into a string.
Used snprintf here because can store the result in a variable for later use rather than displaying it immediately. *This also applies to all snprintf written below*. snprintf link
Calculation/Conversion:

```
int days = (int)uptime_seconds / (24 * 3600);
uptime_seconds -= days * (24 * 3600);
int hours = (int)uptime_seconds / 3600;
uptime_seconds -= hours * 3600;
int minutes = (int)uptime_seconds / 60;
int seconds = (int)uptime_seconds % 60;
```

Last, Format the uptime string to include days, hours, minutes, and seconds

```
snprintf(uptime_str, max_size, "%d days %02d:%02d:%02d (%d:%02d:%02d)", days,
hours, minutes, seconds, hours + (days * 24), minutes, seconds);
```

Error Handle: If the file cannot be opened, or if there is an error reading the file or parsing the uptime, the function sets the uptime_str to "Unknown" and reports the error. Same error handling logic as above regarding conversion error.
How to use:
Invoke read_uptime, passing it the buffer and its size.
read_uptime(uptimeStr, sizeof(uptimeStr));

- ```
  void list_user_sessions(int *session_count, char *users_info_buffer,
  size_t buffer_size);
  ```

Lists active user sessions by reading utmp files. identifies sessions users are logged in (USER_PROCESS entries) User_Process link , prints username and terminal for each session. If the remote host is in session, also displayed, and IP. Total count of user sessions is returned via a pointer argument.
Parameter: int *session_count: A pointer to an integer where the function will store the total count of active user sessions found.
Functionality

- utmp File Reading: Initiates a read operation on utmp file, records login sessions. setutxent() is used to rewind to the start of the utmp file, ensuring that the enumeration starts from the beginning of the file. Resource: setutxent() link

  ```
  #include <utmpx.h>
  ```

- Session Enumeration: Iterates through utmp records using getutxent(), filtering for entries with the type USER_PROCESS,or an active user session. For each
  ```

valid session found, the function outputs the user's name, terminal device, and optionally the remote host name if present. Resource: [setutxent() link]

- Counting Sessions: provided to the caller through the session_count parameter. Comes in handy later on when in the display and graphics function.
- After completing the enumeration, the utmp file is closed using endutxent(), ensuring that system resources are released appropriately. Resource: [endutxent()]
- System Information: Additionally, displays static system information obtained via uname

How to Use:

```
char users_info_buffer[256];
int user_session_count = 0;
list_user_sessions(&user_session_count,users_info_buffer,
sizeof(users_info_buffer));
```

- initialize_config(ProgramConfig *config): Sets default configuration for the program, including flags and counters.

int system_flag:
int user_flag:
 int graphics_flag:
int sequential_flag:
int sample_count:
int time_delay:
Details of the flags and what each flag does can be found in the function overview below.

- `void display_system_info(const ProgramConfig *config, char *cpu_info_buffer,char *memory_info_buffer,char *users_info_buffer);`

display the default information, which is when the input command line is just ./a.out with no argument behind it. Use configuration.initializes to setup ProgramConfig with default values. Takes a pointer to an instance of ProgramConfig as its parameter, allowing it to modify the members of the instance directly.Includes memory usage, CPU load, user sessions, and system uptime.
Parameter: const ProgramConfig *config`: A pointer to a `ProgramConfig` structure that contains configuration flags and settings influencing the scope and format of the displayed information.
Functionality:
Prints the required information based on demo video from assignment.
Screen Clearing: Clears the console screen using ANSI escape codes.
Access memory usage by opening /proc/self/status to read the memory usage of the current process.

Memory and CPU Usage: Retrieves and displays the current physical and virtual memory usage, as well as the CPU usage percentage. Use `gather_memory_info` and `gather_cpu_info`. Refer to above readme for more detail.
User Sessions: Enumerates and lists active user sessions by invoking `list_user_sessions`. Refer to list_user_session function readme above for more detail.
System Uptime: Fetches and formats the system uptime using `read_uptime'.Refer to read_uptime function readme above for more detail.
ANSI Escape Codes: Utilizes ANSI escape codes to manage the cursor position on the console, ensuring that updates are made to the correct screen locations without reprinting the entire screen content. Allows dynamic refreshing of displayed data.
ESCAPE code usage example, also applies to all Escape code usage above and below

```
printf("\033[%dB",4+user_session_count);
printf("\033[2K");
printf("\rtotal cpu use = %.2f%%", cpu_usage);
printf("\033[%dA",4+user_session_count);
printf("\033[%dD",21);
```

Use this to update total cpu use while having the correct formatting.
Dynamic Updates: Use loop to update the memory and CPU usage information based on the console for the number of samples specified or if not indicated, use the default value, with a delay between updates as configured.
System Information: Additionally, displays static system information obtained via uname
Calculation/Conversion:

- Memory usage is presented in gigabytes (GB) which requires conversion from kilobytes (as read from /proc/meminfo and /proc/self/status).
- CPU usage is calculated as a percentage, showing the proportion of CPU resources being utilized. (refer to gather_cpu_info for details)

```
gather_cpu_info(&cpu_usage);
gather_memory_info(&phys_used_mem, &total_phys_mem,
&virt_used_mem, &total_virt_mem);
```

Refer to gather_memory_info for calculation
Before is the conversion:
Physical used memory:(double)phys_used_mem / 1024.0 / 1024
Physical total memory: (double)total_phys_mem / 1024.0 / 1024,
Virtual used memory:((double)virt_used_mem / 1024.0 / 1024)+(double)phys_used_mem / 1024.0 / 1024
Virtual total memory:(double)total_virt_mem / 1024.0 / 1024+(double)total_phys_mem / 1024.0 / 1024)
Error Handling:error checking for data retrieval and reading and parsing memory usage. In case of conversion error, use the method mentioned in gather_memory_info.
How to Use

```
display_system_info(&config,read_buffer_cpu,read_buffer_mem,read_buffer_user);
```

- `void display_system_info_system(const ProgramConfig *config, char *cpu_info_buffer,char *memory_info_buffer);`

This function is used for command ./a.out –system where it generates on the system usage based on requirement: report how much utilization of the CPU is being done

- report how much utilization of memory is being done (report used and free memory)
    - Total memory is the actual physical RAM memory of the computer.
    - Virtual memory accounts for the physical memory and swap space together -- swap is the amount of space (usually in disk or any other storage device) assigned by the OS to be used as memory in case of running out of physical space.

Parameter

- const ProgramConfig *config: A pointer to a ProgramConfig structure that specifies the number of samples to display (sample_count) and the delay between each sample (time_delay)

Functionality

Please refer to the void display_system_info function above as this function does the exact same job for just displaying the system usage. (skip user session)

System Information: Additionally, displays static system information obtained via uname

Calculation/Conversion

- Memory usage is presented in gigabytes (GB) which requires conversion from kilobytes (as read from /proc/meminfo and /proc/self/status).

For detailed calculation refer to void_display_system_info function above, which has the exact same calculation.

- CPU usage is calculated as a percentage, showing the proportion of CPU resources being utilized. (refer to gather_cpu_info for details.

Error Handle:

- Exactly the same as the void display_system_info function above.

How to Use:

```
display_system_info_system(&config,read_buffer_cpu,read_buffer_mem);
```

```
void display_system_info_sequential(const ProgramConfig *config, char *cpu_info_buffer,char *memory_info_buffer,char *users_info_buffer);
```

sequentially displays detailed system information, including memory and CPU usage, user sessions and system information. Use the command ./a.out –sequential. However, instead of doing real-time dynamic updates, this mode provides a snapshot per iteration, allowing for a thorough inspection of system changes over time.

Parameter

- const ProgramConfig *config: A pointer to a ProgramConfig structure that dictates the number of iterations (sample_count) and the interval between each iteration (time_delay), along with other configurable options.

Functionality

- Please refer to the void display_system_info function above as this function does the exact same job but instead does it sequentially. Such that instead of overwriting/clearing the previous iteration, you print the whole thing again below, however, the memory usage line will be different and the result of it is printed one line below the previous iteration in terms of placeholder space as indicated in the sample video from assignment.
- Iterative Display: For each iteration, defined by the sample_count, the function:
    - Displays the iteration number and static configuration details (number of samples and delay between samples).
    - Collects and displays current memory usage (physical and virtual) and CPU usage statistics.
    - Enumerates and lists active user sessions, enhancing visibility into current system access.
    - Repeat the above steps, pausing for the specified time_delay between iterations.
- System Information: Additionally, displays static system information obtained via uname

Calculation/Conversion

- Converts memory usage statistics from kilobytes to gigabytes for readability.
- Calculates CPU usage as a percentage of total CPU resources utilized.
- For detailed calculation refer to void_display_system_info function above, which has the exact same calculation.

Error Handling

- Includes validation for file access and parsing operations. If any error occurs (e.g., failure to open /proc/self/status), appropriate error messages are displayed, and the function exits the current iteration to maintain the integrity of the display process.
- Exactly the same as the void display_system_info function above.

How to use:

```
display_system_info_sequential(&config,read_buffer_cpu,read_buffer_mem,read_buf
fer_user);
```

```
void display_user_info(const ProgramConfig *config,char *users_info_buffer);
```

display user-specific system information, using the command ./a.out –user, including active user sessions and the memory usage of the monitoring tool itself. Basically a version of display_system_info.

Parameter

- const ProgramConfig *config: A pointer to a ProgramConfig structure that influences the display, particularly regarding the number of samples and the

delay between updates, although for this function, the focus is more on user sessions than on dynamic sampling.

Functionality

- Please refer to the void display_system_info function above as this function does the exact same job for just displaying the user usage.
- User Sessions Display: counts active user sessions by invoking list_user_sessions, which scans the utmp file to find logged-in users. Then displays the username and terminal for each active session,and ip address.
- System Information: Additionally, displays static system information obtained via uname

Calculation/Conversion

- Converts memory usage statistics from kilobytes to gigabytes for readability.
- Calculates CPU usage as a percentage of total CPU resources utilized.
- For detailed calculation refer to void_display_system_info function above, which has the exact same calculation.

Error Handling

- Includes checks for potential errors when opening and reading /proc/self/status. If an error occurs (e.g., the file cannot be opened or there's an issue parsing the memory usage), it reports the problem and exits the current operation to prevent incorrect or incomplete information display.
- Exactly the same as the void display_system_info function above.

How to Use

```
display_user_info(&config,read_buffer_user);
```

```
void graphical_system_info(const ProgramConfig *config, char
*cpu_info_buffer,char *memory_info_buffer,char *users_info_buffer);
```

provides a graphical representation of system information, including memory and CPU usage, using command line ./a.out –graphical. It uses a dynamic console to resemble a real-time graph of system resource utilization, alongside static system information and user sessions.

Parameter: const ProgramConfig *config: A pointer to a ProgramConfig structure that determines the number of updates (sample_count) and the time interval between each update (time_delay).

Functionality:

Screen Preparation: Clears the console screen using ANSI escape codes to ensure a clean slate for displaying the graphical information.

Resource Usage Monitoring: Refer to display_system_info and gather_cpu_info and gather_memory_info for details.

Use memset to fill a block of memory with a particular value using snprintf as well.[memset link]

Graphical Updates: For each sample iteration, as dictated by sample_count,which can be default or given by the user, the function:

Collects current memory and CPU usage data using gather_memory_info and gather_cpu_info.
Graphically represents memory and CPU usage changes using custom graphics (e.g., bars or lines), dynamically updating these on the console for real-time visualization.Scale/magnitude of this is based on cpu_usage:
int change_magnitude_cpu = (int)(cpu_usage * 40);
Displays user session count and system core count as part of the system overview.
System Information: Additionally, displays static system information obtained via uname
Dynamic Refresh: Use ANSI escape codes to manage cursor positions and refresh only specific parts of the console output.

Calculation/Conversion:
Converts memory usage figures from kilobytes to gigabytes.
For detailed calculation refer to void_display_system_info function above, which has the exact same calculation.
Represents CPU usage as a percentage, graphically depicting the load over time.
Error Handling:
Implements error checking for file access and parsing operations related to retrieving the tool's memory usage.
Exactly the same as the void display_system_info function above.
How to
Use:`graphical_system_info(&config,read_buffer_cpu,read_buffer_mem,read_buffer_user);`

---

parse_arguments(int argc, char *argv[], ProgramConfig *config)
Checks the command-line arguments provided to the program, updating a ProgramConfig structure with given/required setting. Also allows users to change the number of samples, the delay between samples.
Parameters
int argc: The number of command-line arguments passed to the program.
char *argv[]: An array of character pointers listing all the arguments passed to the program.
ProgramConfig *config: A pointer to a ProgramConfig structure that will be updated based on the parsed arguments
Functionality:
Arguments Parsing: Identifies numeric arguments that specify sample_count and time_delay. It expects these numbers to appear as the first two arguments after the program name, if not then it must be (-), or command arguments.
Flag Arguments Parsing: Recognizes specific flag arguments (--system, --user, --graphics, --sequential) that indicate the desired mode of operation for the monitoring

tool. Each flag sets a corresponding boolean field in the ProgramConfig structure to 1 (true), enabling that feature.
Error Handle: Use straightforward error handling approach by ignoring unrecognized arguments and proceeding with the parsing process.
How to run: parse_arguments(argc, argv, &config);
Example of command line, (a few special cases)
./a.out 5 2.
This means the default ./a.out display_system_info will be runned and sample size is changed from default size of 10 and time delay is changed to 2 seconds from 1 seconds.
./a.out –sequential –graphics
This means the program will print the required lines for –sequential first using display_system_info_sequential, then after finish, it will print the required lines for –graphics using graphical_system_info.
How to Run:
./a.out sample_count[optional] time_delay [optional] –flags[one or more or none]
Options:
--system
--user
--graphics
--sequential
By default

---

sigint_handler(int signum)
handle the SIGINT signal, typically generated by pressing Ctrl+C in the terminal.
first saves the current cursor position in the terminal. Then, it asks the user whether they want to quit the program by displaying a message and reading the response from stdin. This interaction requires the user to type "yes" to confirm the exit.
If the user types "yes" (followed by hitting Enter), the program calls exit(0) to terminate immediately.
If the user types anything else, the program resets the SIGINT signal handler to sigint_handler itself, ensuring that future SIGINT signals continue to be handled in the same manner. It then reads any remaining input until a newline character is encountered, to clear the input buffer, and restores the cursor position to where it was when the signal was received.
Failure to Read Input: If fgets fails to read input, the function tries to reset the signal handler for SIGINT to itself again.

—---

handle_sigtstp(int signum):
pressing Ctrl+Z in the terminal. The SIGTSTP signal suspends (stops) a process.
The function immediately re-registers itself as the handler for SIGTSTP using signal(signum, handle_sigtstp). This ensures that subsequent SIGTSTP signals are still handled by this function.

The function doesn't explicitly define any behavior (like printing a message or modifying program state) beyond re-registering itself as the handler. It essentially sets up to catch SIGTSTP signals without defining a custom response, relying on the default system behavior to suspend the process. The commented-out call to fflush(stdout) suggests there may have been or could be additional intended behavior, like ensuring all output is flushed to the terminal before the process is suspended.

Signal Handling Setup: It sets up signal handlers for SIGINT (usually sent by pressing Ctrl+C) and SIGTSTP (usually sent by pressing Ctrl+Z). These handlers allow the program to respond to these signals in custom ways, such as cleaning up resources or prompting the user before exiting.

Configuration Initialization: It initializes a ProgramConfig structure, presumably to hold configuration settings that might be modified based on command-line arguments.

Command-Line Arguments: It parses command-line arguments to potentially adjust the program's configuration, such as setting operation modes or parameters for the information gathering.

Pipe Creation: It creates three sets of pipes for inter-process communication (IPC) between the parent process and each of the child processes. Pipes are used to send data from the child processes back to the parent process.

Forking Child Processes:

- For each area of system information (CPU, memory, user sessions), the program forks a new child process.
- Each child process performs its specific task (gathering CPU, memory, or user session information), writes the results into the write end of its corresponding pipe, and then exits.
- Each child process closes the read end of its pipe since it only needs to write to the parent.

Reading from Pipes:

- After forking the child processes, the parent process closes the write ends of the pipes (since it only needs to read from the children) and waits for each child process to complete using waitpid.
- It then attempts to read the data sent by each child process from the pipes. This data includes CPU utilization, memory utilization, and user sessions information.

Displaying Information: Based on the command-line arguments and the retrieved data from the child processes, the program decides which information to display:

- General System Information: If no specific flags are provided, or under certain conditions (like argc == 1), it displays general system information using the data read from the pipes.
- Specific Flags: If specific flags are set (system_flag, graphics_flag, user_flag, sequential_flag), it displays detailed information tailored to each flag. This could include more focused information on system stats, graphical displays, user-specific data, or sequential data presentation, all utilizing the data gathered from the child processes.

Note:
-> for time 1%, since when pipe, fork, child process etc is used, when CLA argument is imputed and pressed, note that there is a 2 second delay before executing, such that for example time ./my_program, real time should be 20.12 second, however, the actual output is 22.12 second when taking into account the 2 second delay. Or for ex: time ./my_program, real time should be 30.12 second, however, output is 32.12 for the same reason.
->for all the display and graphic function, in order to achieve the printing memory change and cpu change in a similar manner to the sample video, introduced the concept of placeholder, where I first have placeholder for memory or cpu (depends on the function), then afterwards use ESCAPE code to updates within the placeholder. For details or examples please refer to above in each individual function overview and documentation.
->When testing cpu usage for ./a.out ./a.out –system, can also set the cpu usage result to .5f or .4f as the changes can be very minimal. However, this raises another issue of formatting, if you change the cpu usage to .5f or .4f, formatting might be different.
->for memory change for graphics, I have asked the professor, he said we can always assume the first memory_change is always gonna be 0 so that why in graphical _ system_info I had:

```
if (i==0){
   memory_change = 0;
}
```