Assignment 3- Concurrency & Signals

****WARNING: PUT YOUR TERMINAL ON FULL SIZE BEFORE USE

How to compile

To compile the code based on the given makefile, follow these instructions:

1. Open a terminal or command prompt in the directory where the makefile and the source code files are located.
2. Type `make` and press Enter. This will execute the makefile and build the executable file named "mySystemStats".
3. If the build is successful, you can run the program by typing `./mySystemStats` in the terminal and pressing Enter. This should execute the program and display the system statistics.
4. If you want to clean up the executable file and other generated files, you can type `make clean` and press Enter. This will remove the "mySystemStats" file and any other generated files.

Command Line Argument Rules

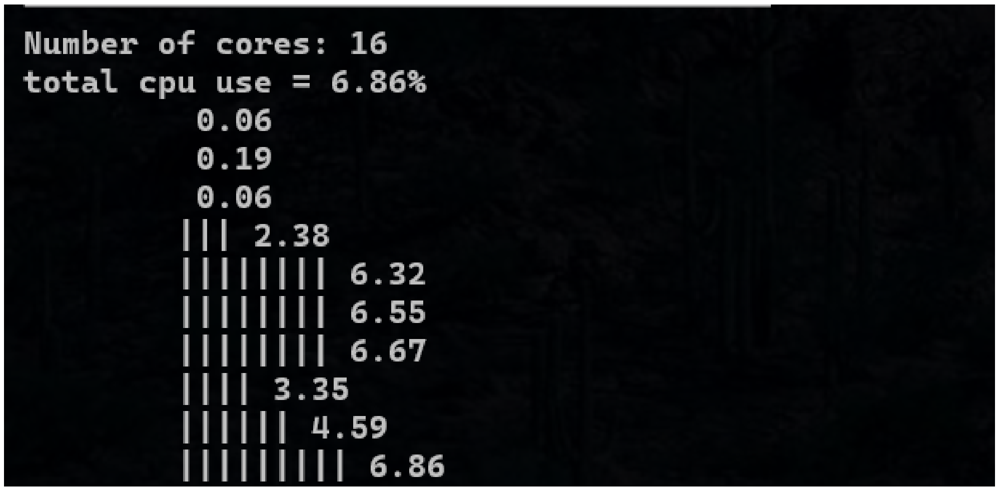
Here are the following rules for flags for ./mySystemStats. These were both approved by Marcelo and multiple TA’s

- The program accepts command line arguments for the following options: `--system`, `--user`, `--sequential`, `--samples=<value>`, `--tdelay=<value>`, and `--graphics`
  - You may **not** call `--user` and `--system` in the same call because this is contradictory behaviour. For example `./mySystemStats --user --system` is not valid
  - Values must be valid
  - If we have `--sequential` paired up with `--user` or `--system`, it will print the output of `--user` or `--system` in sequential fashion.

- The program also accepts positional arguments, which must be integers and specify the number of samples and time delay, respectively.
  - You may **not** use both positional arguements and `--samples=X/--tdelay=X`. For example `./mySystemStats 10 1 --samples=4` is not valid
  - You may **not** only use one positional arguement, both must be used. For example `./mySystemStats 10` or `./mySystemStats 10 1 1` is not allowed

Graphics Behaviour:

For CPU utilization:



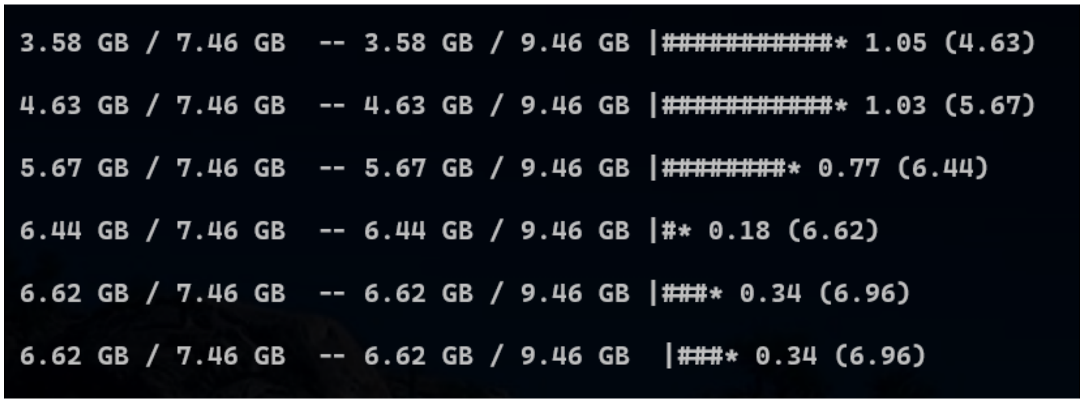
```
Number of cores: 16
total cpu use = 6.86%
         0.06
         0.19
         0.06
        ||| 2.38
        |||||||| 6.32
        |||||||| 6.55
        |||||||| 6.67
        |||| 3.35
        |||||| 4.59
        ||||||||| 6.86
```

Sample Output

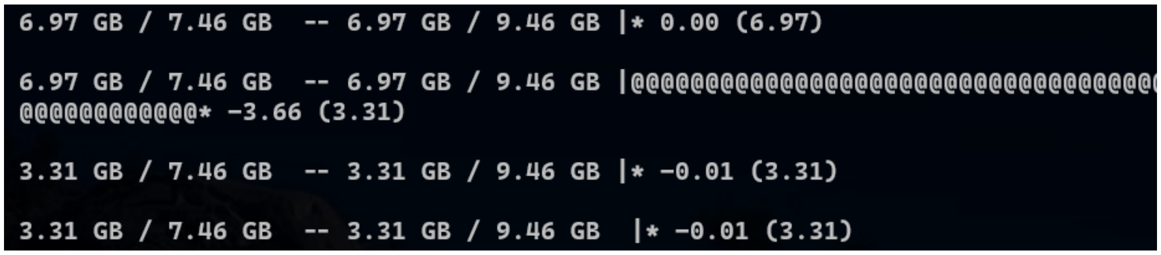
The way the bar system works for CPU utilization is that a bar appears for how many times we can add **0.75** without going over our cpu percentage.

Another way of representing this is `numBars = cpuUtil % .75`.

For Mem Utilization:



```
3.58 GB / 7.46 GB  -- 3.58 GB / 9.46 GB |###########* 1.05 (4.63)

4.63 GB / 7.46 GB  -- 4.63 GB / 9.46 GB |###########* 1.03 (5.67)

5.67 GB / 7.46 GB  -- 5.67 GB / 9.46 GB |########* 0.77 (6.44)

6.44 GB / 7.46 GB  -- 6.44 GB / 9.46 GB |#* 0.18 (6.62)

6.62 GB / 7.46 GB  -- 6.62 GB / 9.46 GB |###* 0.34 (6.96)

6.62 GB / 7.46 GB  -- 6.62 GB / 9.46 GB  |###* 0.34 (6.96)
```



```
6.97 GB / 7.46 GB  -- 6.97 GB / 9.46 GB |* 0.00 (6.97)

6.97 GB / 7.46 GB  -- 6.97 GB / 9.46 GB |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@@@@@@@@@@@@* -3.66 (3.31)

3.31 GB / 7.46 GB  -- 3.31 GB / 9.46 GB |* -0.01 (3.31)

3.31 GB / 7.46 GB  -- 3.31 GB / 9.46 GB  |* -0.01 (3.31)
```

Number of `#` or `@` is determined by how much we increase or decrease compared to the mem utilization in the most recently sampled memory utilization.

helpers.c

Here is a high-level description of the computational helpers we have used **(exactly the same as A1)**, as well as the structs used to be passed through the pipes (more on this later):

For a more in depth explanation, refer to my A1 doc on the last couple pages: https://drive.google.com/file/d/1LtNw_anMEaxfcHQvGW7DALra1PfwHDHD/view?usp=sharing

`clear_screen()`

This function clears the terminal screen and moves the cursor to the top-left corner of the screen using ANSI escape codes.

`struct mem_info`

This struct holds information about memory usage, including the total amount of memory and the current memory usage. It also includes an array to store the memory usage samples for calculating the average usage.

`struct memory_display`

This struct holds a string that represents the formatted memory usage display.

`struct session_info`

This struct holds information about the users who are currently logged into the system, including their usernames and the number of users.

`struct cpu_info`

This struct holds information about CPU usage, including the CPU utilization and the number of bars to display when rendering the CPU usage graph.

`typedef MachineInfo`

This typedef defines a struct `MachineInfo` that contains information about the machine, including the system name, node name, release version, machine type, and version.

`logCores()`

This function logs the number of cores in the system using the `sysconf()` system call.

`sleep()`

This function pauses the execution of the program for a specified number of seconds.

`struct cpustat`

This struct holds CPU usage statistics, including the amount of time the CPU has spent in various states (user, nice, system, idle, iowait, irq, and softirq).

`skip_lines()`

This function reads a file and skips a specified number of lines.

`get_stats()`

This function reads the `/proc/stat` file and retrieves CPU usage statistics for a specific CPU core.

`calculate_load()`

This function calculates the CPU utilization by comparing the CPU usage statistics for two different points in time.

`logCpuUsage()`

This function logs the current CPU usage by calling `get_stats()` and `calculate_load()`.

`get_cpu_times()`

This function retrieves CPU usage statistics from the `/proc/stat` file, including the amount of time the CPU has spent in various states (user, nice, system, idle, iowait, irq, softirq, and steal) and the amount of time spent waiting for I/O.

stats_functions.h

`sigint_handler(int sig)`

This function is a signal handler that handles the SIGINT signal (which is generated by the user pressing Ctrl+C). It sets the signal to be ignored, prompts the user to confirm if they really want to quit, and exits the program if the user confirms by typing "y" or "Y". If the user enters any other character, the function sets the signal to be handled again.

`logSessional(int users_pipe[2], int NUM_SAMPLES, int SLEEP_TIME)`

This function writes the users and their sessions to a pipe using the utmp file. It takes in a pipe for writing the information, the number of times the function should run, and the amount of time to sleep between each run. Within the function, it initializes a `session_info` structure to hold information about the users and their sessions. It then opens the utmp file, retrieves information about the sessions, and stores that information in the `session_info` structure. It then writes the structure to the pipe and sleeps for the specified amount of time before repeating the process until the specified number of runs is completed.

`getMachineInfo(int pipefd[2])`

This function retrieves information about the machine, such as the system name, node name, release version, and machine type. It takes in a pipe for writing the information. Within the function, it initializes a `MachineInfo` structure to hold the information and uses the `uname` function to retrieve the system information. It then copies the information into the `MachineInfo` structure and writes the structure to the pipe.

`getMemUtil(int NUM_SAMPLES, int SLEEP_TIME, int pipefd[2])`

This function retrieves information about the memory usage of the system. It takes in a pipe for writing the information, the number of times the function should run, and the amount of time to sleep between each run. Within the function, it uses the `sysinfo` function to retrieve information about the system memory, including the total amount of memory and the amount of free memory. It calculates the used memory by subtracting the free memory from the total memory and writes the used memory to the pipe. It then sleeps for the specified amount of time before repeating the process until the specified number of runs is completed.

`getCpuUsage(int samples, int tdelay, int pipe[])`

This function retrieves information about the CPU usage of the system. It takes in a pipe for writing the information, the number of times the function should run, and the amount of time to delay between each run. Within the function, it initializes an array of `cpu_info` structures to hold the information and uses the `get_cpu_utilization` function to retrieve the CPU utilization. It calculates the number of bars (for a graphical representation of the utilization) and writes the information to the pipe. It then sleeps for the specified amount of time before repeating the process until the specified number of runs is completed.

`void graphicalRefresh(int samples, int tdelay, int graphics, int sequential)`

This code defines a function named `graphicalRefresh` that takes four integer parameters named `samples`, `tdelay`, `graphics`, and `sequential`. It then declares several integer variables and initializes them to 0, which are used to store the process IDs and pipe file descriptors for child processes.

The function then creates four pipes using the `pipe` system call and assigns the file descriptors to the appropriate variables. If any of the pipe creations fail, the program exits with an error message.

The next section of the code sets up a signal mask and blocks the `SIGTSTP` signal. It then forks a new process, which becomes the watchdog process, and checks for any errors during the fork process. If the process ID of the child process returned by `fork` is negative, it means that the fork failed. If the process ID is 0, then this is the child process, and it sets up a new signal mask and a variable named `parent` to store the parent process ID. The child process then enters an infinite loop in which it checks if the parent process is still alive by comparing the parent process ID to the stored `parent` variable. If the parent process ID has changed, the child process sends a `SIGTERM` signal to all child processes and exits.

The function then forks three more child processes named `users_pid`, `memory_pid`, and `cpu_pid`, which are used to collect data about user sessions, memory utilization, and CPU utilization, respectively. If any of the fork calls fail, the program exits with an error message.

The child process named `users_pid` sets up a new signal mask and calls a function named `logSessional` with the `users_pipe`, `samples`, and `tdelay` parameters. This function collects data about user sessions and writes the results to the `users_pipe` pipe. The child process then exits.

The child process named `memory_pid` sets up a new signal mask and closes the write end of the `memory_pipe` pipe. It then calls a function named `getMemUtil` with the `samples`, `tdelay`, and `memory_pipe` parameters, which collects data about memory utilization and writes the results to the `memory_pipe` pipe. The child process then closes the read end of the pipe and exits.

The child process named `cpu_pid` sets up a new signal mask and closes the read end of the `cpu_pipe` pipe. It then calls a function named `getCpuUsage` with the `samples`, `tdelay`, and `cpu_pipe` parameters, which collects data about CPU utilization and writes the results to the `cpu_pipe` pipe. The child process then closes the write end of the pipe and exits.

The parent process is responsible for reading and displaying system information, such as memory usage and CPU usage.

Concurrency is achieved through the use of pipes, which allow for interprocess communication between the parent process and the child process using structs (as mentioned before). The parent process creates four pipes: memory_pipe, machine_pipe, user_pipe and cpu_pipe. These pipes are used to send data from the child process to the parent process.

In the parent process, after the child process has finished executing, the memory_pipe, users_pipe, and machine_pipe are closed on their write ends using the close() function, indicating that the parent process will only read from them. The SIGINT signal is then set to be handled by a custom function sigint_handler() using the signal() function.

The parent process then reads the total system memory from memory_pipe using the read() function and stores it in mem_info.memory_total. It then creates an array memory_usage of size samples to store the memory usage of the system at each sample. The memory usage for each sample is read from memory_pipe using the read() function and stored in mem_info.memory_used[i], where i is the index of the current sample. The memory usage is also stored in memory_usage[i].

Similarly, the parent process reads the CPU usage information for each sample from cpu_pipe using the read() function and stores it in the array cpu_info.

The parent process then displays the system information, such as the number of samples and the sampling interval, as well as the memory usage for the current sample. It also calculates and displays the change in memory usage since the previous sample, and if graphics mode is enabled, it displays a graphical representation of the change using hash symbols as defined previously.

This process continues until all the samples have been read and displayed, at which point the parent process exits.

`void usersRefresh` and `void systemRefresh`

These do the exact same thing as `graphicalRefresh` except `usersRefresh` just outputs the Users/Sessions info and systemRefresh just outputs the system usage.

parseArgs.h

This code parses the command line arguments. It uses the argc and argv arguments from main() and updates the values of several variables, such as systemm , user , sequential , samples , and tdelay , based on the arguments specified
by the user. The code supports both positional arguments and named options with the format --flag=value .

The code checks for the presence of flags such as --system and --user , and updates the values of the corresponding variables. For
named options such as --samples= and --tdelay= , the code extracts the integer value following the equal sign and updates the
samples and tdelay variables.

The code also performs several checks to ensure that the arguments passed by the user are valid, such as checking that the values

for samples and tdelay are positive integers and that the correct number of positional arguments are provided.
If an invalid
argument is found, the code will print an error message and exit the program with a status code of 1.