CSCB09 A3: Concurrently running system monitoring tool

Assignment creator: Professor Marcelo Ponce

Arthur: Alexander Zhang

Date of completion: 2025/03/28

1. Objectives

This project(assignment) is about recreating the existing system monitoring project using concurrency. It shares the exact same behaviour as its parent version, where users can use different commands to control the program to print their current RAM and memory usage, together with core-related information through CLAs. Only, this project uses concurrency to collect data, where the child process(s) will be collecting information and the parent will be printing it. Additionally, the program will change its behaviour upon receiving SIGTSTP and SIGINT signals.

2. Approach

To accomplish these objectives, I refactored my code in A1 so that all the printing functions now have additional “ports”, which are file descriptors(ends of pipes) that will enable them for cross-process operation. Additionally, new driver functions are implemented in which fork() and pipe() are used to achieve the concurrent objective. Specifically, in each of these driver functions, different numbers of processes and pipes are created. There would be one main process which focuses on printing the information, and a couple of child processes(whose number equals the number of grafts needed) that collect information and send it to the parent through pipes. Additionally, to ensure synching in the case of handling signals, there are pipes that enable a parent to write to its children to indicate one printing cycle(in the case of memory and CPU) has ended, after which the child will send newly collected information. All information collecting functions are now modularized and have their own header files.

3. Implementation

As noted above, the refactored module contains the following modules based on their functions: CPU**u information collecting**: Module for collecting CPU-related information, embodied by cpu.h and cpu.c. Since CPU utilization relies on delay, an additional function is defined in this module to help with the collection of CPU usage.

RAM information collecting: Module responsible for collecting RAM-related information, embodied by memory.h and cpu.c.

Printing module(uses information from the previous two modules): Contains the refactored printing functions from A1, embodied by present.h and present.c. Additionally, it also grabs and prints information on the core, since there is little to no helper function required for this functionality, no separate module is created for it

Concurrency module(The driving module):This module uses all previous modules to achieve concurrency, hence it is the driving module. Additionally, it enables the program to handle SIGINT signals differently. Embodied by print_con.h and print_con.c.

Driver file: a .c file which contains the main function to use all the above-definedd modules. This is why the SIGTSTP handler is installed.

CPU information collecting:

```
//Pause the program for the given microsecond
//microsecond: Number of microseconds to sleep for
void sleep_microsecond(long microsecond);
```

Implemented to satisfy the need to pause the program for given microseconds; since sleep() accepts only whole seconds, and usleep() is considered obsolete by std99 standard. To satisfy the condition of using nanosleep(), a timespec structure is defined, so that the need for pausing for whole seconds and microseconds may be satisfied simultaneously.

```
//Return the ratio between the CPU on time and the total time of a given
time interval
////delay: the time interval between two samples in microseconds
float get_cpu_utilization(long delay);
```

Calculate the ratio between the CPU on time and total time of a given time interval. The formula used is: CPU utilization = (active_2 - active_1)/(total_2 - total_1), where active time is the total time(since system boot) - idel_time(CPU not working) - iowait(time spent waiting for IO). The second sample is collected after waiting for a delay of microseconds. Hence, active_1/total_1:

active time of CPU/total time of CPU of the first sample, active_2/total_2: active time of CPU/total time of CPU of the second sample.

RAM information collecting:

```
//Calculate the used RAM and total RAM in terms of GB and return their
ratio
////delay, the flag user inputs to control the gap between sampling
float get_Memory_ratio();
```

Calculate the used RAM and total RAM in terms of GB and return their ratio by reading information from the system using sysinfo function. This function is implemented so that the amount of memory used can be reflected in the graph. To be specific, the ratio is multiplied by the height of the matrix to determine its appropriate position.

```
//Get the total RAM and return it as an integer
int get_total_Memory();
//Return the amount of memory currently in use
float get_current_Memory();
```

These two functions use the same function sysinfo(). The only difference is their return type; get_total_memory() returns an integer because the upper limit of the memory chart is expressed as the total RAM on the machine in integers. float get_current_Memory() returns float because the current amount of memory in use needs to be displayed in 2-decimal precision above the chart

Printing module:

```
// When the user inputs too large of a sample size, do a sanity check
void sanity_check();
```

Used during program argument parsing. Warn the user that the number they input for the size of the sample is too large(in this program, greater than or equal to 100 is considered too large). This was initially designed to ensure that the user does not mistakenly input delay before sample size when using positional arguments, since for positional arguments the order must be sample size then delay. This is used later in more general cases to urge the user not to input unreasonably large values for samples.

```
// Return the maximum frequency of the cores in GHz
float get_Max_freq();
```

Return the maximum frequency of the cores in GHz by reading and comparing the frequency of each processor provided by /proc/cpuinfo file. This is needed for the print_Core() function since the maximum frequency of the cores must be displayed alongside the number of cores.

```
/*
Initialize and return a chart matrix with the given parameters
//height: The height of the matrix
//sample_size: The length of the matrix
*/
char **initialize_matrix(int height, int sample_size);
```

Initialize and return a chart matrix with the given parameters. Since the screen needs to be cleaned each time for the graph to update, I need to memorize the previous graph so that I can print the updated graph. A 2D matrix would be the best choice since each entry can represent a “pixel” in the chart. The matrix is dynamically allocated to ensure fit and ease of transfer between functions. A function is created because both CPU chart and Memory charts need the matrix, and their initial setup is the same(X-axis, Y-axis).

```
/*
Print the given CPU matrix
//to_print: Pointer to the target matrix
//ratio: the ratio of the usage of CPU and the total time
//height: length of the matrix
//sample_size: length of the matrix
//Returns nothing
*/
void print_CPU_matrix(char **to_print, float ratio, int height, int
sample_size);

/*
Print the given CPU matrix
//to_print: Pointer to the target matrix
//current: the amount of memory currently in use
//total: total amount of memory
//height: length of the matrix
//sample_size: length of the matrix
//Returns nothing
*/
void print_MEMORY_matrix(char **to_print, float current, int total, int
height, int sample_size);
```

These two functions are essentially the same, with some minor differences in parameters since the CPU graph and memory graph require different readings to be printed. In the Memory chart, the upper limit and current memory (max-memory) are dynamic(different on each machine) and hence need to be passed in an argument, whereas in CPU chat, the upper and lower are set(100% and 0%), so only the ratio of the CPU time and height and length of the array needs to be passed in because the percentage of the current CPU utilization can be obtained in percentage by simply multiplying the ratio by 100.

```
/*
Print the dynamic graph of the memory usage
//sample: The total number of samples to print
//delay: The time interval between collecting samples measured in seconds
//fd_read: file descriptor for getting data from child
//fd_write: fd for writing back to child process(to indicate the finishing
of printing, needed to handle signals)
//Returns nothing
*/
void print_MEMORY(int sample, long delay, int fd_read, int fd_write);
```

Print the dynamic graph of RAM usage utilizing initialize_matrix(), get_Memory_ratio(), print_memory_matrix(), get_total_Memory() and, get_current_Memory(). This function essentially serves as a combiner and driver of all these helper functions, while handling some business logic(which row and column to put the “#” according to the data from helper functions, determining the height and length of the matrix given the sample size, determining the delay, refreshing the graph, etc). Additionally, pipe ends are added as parameters so that the function supports concurrency operations.

```

/*
Print the dynamic graph of CPU usage
//sample: The total number of samples to print
//delay: The time interval between collecting samples measured in seconds
//fd_read: file descriptor for getting data from parent
//fd_write:  writing back to child process(to indicate the finishing of
printing, needed to handle signals)
//Returns nothing
*/
void print_CPU(int sample, long delay, int fd_read, int fd_write);
```

Similar to print_Memory(). Except the functions it utilizes are slightly different: get_cpu_utilization(). Additionally, pipe ends are added as parameters so that the function supports concurrency operations.

```
/*
```

```
Print the graphs for Memory and CPU asynchronously
//sample: The total number of samples to print
//delay: The time interval between collecting samples measured in seconds
//fd_read_1: file descriptor for getting data from child(memory)
//fd_read_2: fd to read from child 2(CPU)
//cpu_fd: fd for  writing back to child process(to indicate the finishing
of printing, needed to handle signals)
//Returns nothing
//mem_fd: fd for writing back to child process for collecting memory
*/
void print_MEMORY_AND_CPU(int sample, long delay, int fd_read_1, int
fd_read_2, int cpu_fd, int mem_fd);
```

Combines the functionality of print_Memory() and print_CPU(). Note this function is created rather than simply calling print_Memory() and print_CPU() because the refresh of the screen must be controlled inside print_Memory() and print_CPU() to satisfy the need of only printing one of the graphs(else, the main function will seem clumsy and filled with loops). The functions essentially do the same thing as initializing, printing, and updating the graphs only; it does it for CPU and Memory simultaneously and controls the refresh of the screen for both graphs at the same time. Additionally, pipe ends are added as parameters so that the function supports concurrency operations.

```
// Print the number of logical cores and the maximum frequency of
processors on the machine
//Returns nothing
void print_Core(int to_plot, float freq);
```

Print the number of logical cores and the maximum frequency of processors on the machine, The parameters, which will be provided by the child process using get_nprocs_conf(), which is a built-in function for the sys/sysinfo.h library, and get_Max_freq().

Concurrency module:

```
// Handler function for the main process. It will pause the program
// code: signal code to intercept
void sig_int_handler_parent(int code);
```

Handler to be installed into the parent process of all driving functions in this module. It intercepts the SIGINT signal and checks whether the user wants to quit.

```
// Function for printing memory using concurrency
// sample: The total number of samples to print
// delay: The time interval between collecting samples measured in seconds
```

```
// Returns nothing
void mem_driver(int sample, long delay);

// Function for printing cpu using concurrency
// sample: The total number of samples to print
// delay: The time interval between collecting samples measured in seconds
// Returns nothing
void cpu_driver(int sample, long delay);


// Print the core-related information using concurrency
// Returns nothing
void core_driver();

// Function for printing cpu and memory using concurrency
// sample: The total number of samples to print
// delay: The time interval between collecting samples measured in seconds
// Returns nothing
void mem_cpu_driver(int sample, long delay);
```

These functions, as their names suggest, print the RAM, CPU, Core and RAM and CPU together using Concurrency. For the former three, one child process is created for information collecting, for the last, two children processes are created to retrieve mem , core, and cpu information respectively(using the corresponding functions in memory, cpu, and present modules). The functions call print_MEMORY, print_CPU, print_Core and print_MEMORY_AND_CPU respectively in their main processes.

```
// Function for printing memory and core using concurrency
// sample: The total number of samples to print
// delay: The time interval between collecting samples measured in seconds
// Returns nothing
void core_mem_driver(int sample, long delay);

// Function for printing cpu and core using concurrency
// sample: The total number of samples to print
// delay: The time interval between collecting samples measured in seconds
// Returns nothing
void core_cpu_driver(int sample, long delay);

// Function for printing cpu and memory and core using cucurrency
// sample: The total number of samples to print
```

```
// delay: The time interval between collecting samples measured in seconds
// Returns nothing
void core_mem_cpu_driver(int sample, long delay);
```

Similar to mem_driver, cpu_driver, and mem_cpu_driver, except now in each of these an additional process it created to retrieve information for cores

4. PseudoCode

```
myMornitoringTool
{
    // In the case of one argument
    if (no argument)
    {
        print_Core_Memory_CPU(default size and sample); // default
behaviour
        exit;
    }
    // if one argument
    if (one argument)
    {
        if (chage_delay or change_sample_size)
        {
            print_Core_Memory_CPU(updated size or sample);
        }
        if (--Cores or --cpu or --memory)
        {
            // print the corresponding graph
        }
        // The input is not valid
        else
        {
            warn_invalid_input;
            exit;
        }
    }

    // Two or more arguments

    //Case 1: first two are positional
    if (first_two_positional)
```

```
    {
        for (all_other_arguments)
        {
            if (--Cores or --cpu or --memory)
            {
                change_corresponding_flag_true;
            }
            //The user can change the delay or sample size later
            if(--tdelay or --samples){
                update_delay or update_sample;
            }

            //Invalid use case
            else{
                warn_invalid_input;
                exit;
            }

        }
    }

    //First two are not postional
    else{
        for (all_other_arguments)
        {
            if (--Cores or --cpu or --memory)
            {
                change_corresponding_flag_true;
            }
            //The user can change the delay or sample size later
            if(--tdelay or --samples){
                update_delay or update_sample;
            }

            //Invalid use case
            else{
                warn_invalid_input;
                exit;
            }
```

```
        }
    }

    //If core flag is marked true, then every thing will be core-related
    if(core_flag == true){
        if(cpu_flag == true){
            print_core_cpu();
            exit
        }
        else if(memory_flag == true){
            print_core_memory();
            exit
        }
        else if(cpu_flag == true && memory_flag == true){
            print_core_cpu_memory();
            exit
        }
        else{
            print_core();
            exit;
        }
    }
    else{
        if(cpu_flag == true){
            print_cpu();
            exit
        }
        else if(memory_flag == true){
            print_memory();
            exit
        }
        else if(cpu_flag == true && memory_flag == true){
            print_cpu_memory();
            exit
        }
        else{
            print_memory_cpu();
            exit;
    }
}
```

```
//Additionally, for each print... function, here is the pseudocode
print...(sample, delay){
    Create_pipe;
    count = 1

    For(number of graphs){
        fork;
    }

    if(child){
        while(count < sample){
        collect_related_info
        write_info_to_write_end
        wait_for_write_back
        sleep(delay)
        }
        exit

    }
    if(parent){
        while(count < num_samples){
        read_info_from_read_end_of_pipe
        print_info
        write_to_child_for completion
        }
        wait for child to finish
        exit
    }
}
```

5. Compilation instruction

1. Must compile using make file. To compile the executable, simply type “make”.
2. If wish to remove all source files, executable, one shall type “make clean”
3. When compiling, all module files, the driver file, and the Makefile **must be in the same folder**

6. Expected result:

The program accepts several command line arguments:

--memory
to indicate that only the memory usage should be generated

--cpu
to indicate that only the CPU usage should be generated

--cores
to indicate that only the core information should be generated

--samples=N
If used the value N will indicate how many times the statistics are going to be collected and results will be average and reported based on the N number of repetitions.
If no value is indicated, the default value will be 20.

--tdelay=T
to indicate how frequently to sample in microseconds (1 microsec = 10 -6 sec.)
If no value is indicated, the default value will be 0.5 sec = 500 millisec = 500000 microsec.

These last two arguments can also be specified as positional arguments if no flag is indicated, in the corresponding order: samples tdelay.
In this case, these arguments should be the first ones to be passed to the program.

CLA Syntax:

./myMonitoringTool [samples [tdelay]] [--memory] [--cpu] [--cores] [--samples=N] [--tdelay=T]

Default Behavior

- If no arguments are passe,d the program should present all the information about memory utilization, CPU utilization, and cores.
- Default values for samples=20, tdelay=500000 microsec.

Note:

- ***Due to how sampling works, the program will wait for –tdelay time before printing the graphs. For example: ./mymonitoringtool –tdelay=20000000 –cpu will lead to the program wait for 20 seconds before printing the graph***

- ***As long as user inputs an unreasonably large sample amount, regardless of where they inputted it, it will trigger the sanity check. For example: ./myMonitoringTool 2000 --samples=10 will trigger the sanity check even the sample size will actually later on be set to 10***

- ***Since the length of the y-axis is dynamically allocated depending on the sample size provided, when the sample size is too big, it might cause the title("Nbr of samples…..") and part of the graph to be not present in the default-sized terminal window. In this case, expanding the terminal window would resolve the issue.***

- ***If the user inputs one positional argument, like "./ a.out 40", then it will set the sample size***

- ***If the user sets the input to larger than 100, it will trigger a sanity test to remind the user of their choice***

- ***If repetitive flags are passed in, then the program will use the rightmost flag provided, for example, "./a.out 50 –samples=25" will the sample size to 25***

- ***Sample size and delay must be positive integers***

- ***When using positional arguments, they must be the first to be inputted. Numerical values without flags will not be allowed anywhere else in the arguments, for example:***
  , "./a. out 50 –core 50" is prohibited

- ***Users can input the same flag multiple times***

- ***If none of –core, –cpu, –memory flags is provided, then the program will print all their information, regardless of the rest of the command line arguments for example: "./a.out 50 250000 –samples=98 –tdelay=2500 –samples=60" will lead the program to print Memory, CPU, and core information with sample size 60 and delay of 2500 microseconds***

- ***When only core information is printed, the program will NOT print the title "Nbr of samples — X microSecs delay", as sample size and delay are unreasonable under this context.***

- ***For both the CPU chat and Memory charts, the minimal value is printed in a way that it would replace the X-axis(the dashes at the bottom of the graph)***

7. Test cases

The program has been tested extensively with various combinations of flags. Some unexpected output include: When an invalid flag is provided, the program prints the invalid flag and exits.

8. Disclaimers

1. Since the program relies on multiple system files to complete its execution, if any of these files is corrupted or the user does not have the right permission, the program will fail
2. Since the program uses pipes and pipes are system-managed resources, the program could fail if pipes cannot be created or closed.
3. Since the program relies on pipes to communicate between processes, if one of the process crashes, it is possible that the program will enter a deadlock state(the program will get stuck).

9. References

1. https://linux.die.net/man/2/fstat
2. https://man7.org/linux/man-pages/man2/sysinfo.2.html
3. https://stackoverflow.com/questions/11706563/how-can-i-programmatically-find-the-cpu-frequency-with-c
4. https://piazza.com/class/m5j5smmmxyx2s0/post/50 (For core frequency)
5. https://man7.org/linux/man-pages/man0/signal.h.0p.html
6. https://man7.org/linux/man-pages/man3/waitpid.3p.html
7. https://www.tutorialspoint.com/c_standard_library/c_function_fscanf.htm
8. https://stackoverflow.com/questions/3748136/how-is-cpu-usage-calculated
9. https://man7.org/linux/man-pages/man2/nanosleep.2.html
10. https://www.geeksforgeeks.org/c-nanosleep-function/
11. https://man7.org/linux/man-pages/man2/sigaction.2.html
12. https://man7.org/linux/man-pages/man2/pipe.2.html
13. https://cms-pcrs.utsc.utoronto.ca/cscb09w25/content/quests (For use of pipes and sigactions)
14. https://unix.stackexchange.com/questions/607109/when-i-do-double-fork-a-process-is-the-detached-process-still-in-the-same-proce
15. https://man7.org/linux/man-pages/man2/kill.2.html
16. https://docs.oracle.com/cd/E19683-01/806-4125/6jd7pe6bo/index.html#:~:text=A%20pipe%20between%20two%20processes,h%3E%20…
17. https://man7.org/linux/man-pages/man7/pipe.7.html