Author: Samuel Brucker

SysInfo Version 3

GitHub: https://github.com/HaloSam296/SysInfo

5/1/24

Description:

SysInfo is a small utility tool built in C. Its purpose is to provide basic system information. With the new Version 2 release, there are expanded tools included for CPU information.

Setting it up:

The program must be ran on a Debian machine. The package “bc” is required to use option 8. If your machine does not have it, it will be automatically installed by the program, just make sure you have at least 1GB of free storage and an Internet connection.

To run the script, open your terminal and navigate to the folder containing the tool. This can be done through the use of the “cd” command. Then, compile the program using gcc.

Here’s an example: “gcc main.c -o run”

A computer code with text

Description automatically generated with medium confidence

If you used the example above, you can run the script by running the command “./run”, however if you didn’t use that exact command, the format to run the script is “./[name of compiled file]”. Please note this is the name of the compiled file, *not* the original main.c file.

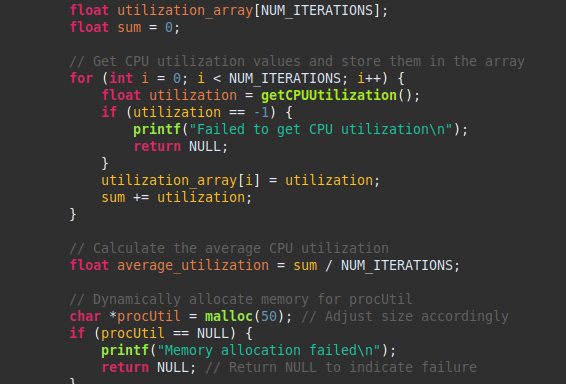
A screen shot of a computer

Description automatically generated

Array:

This program uses an array for option 8. This options runs a Bash script to acquire CPU utilization ten times, storing the outputs into an array. It then provides the array, as well as the average utilization.

Here is the array’s code:



Here is the output:

A screen shot of a computer

Description automatically generated

Options:

There are two main sections for the options, Miscellaneous, which include options 1-7, and CPU Options, which contains 8-10. Finally, there are two options outside of either section, 11 and 12, which are respectively used to check for the requisite packages and to exit the program.

**Please see the Bash section below to see more details on how options that run bash scripts work with their respective scripts. This sections describes the operations and I/O relevant to the C file only.**

Option 1 – Time and Date:  
This option grabs the current system time and date.

A screenshot of a phone

Description automatically generated

Option 2 – Hostname:  
Option 2 grabs the hostname of the system, which can be thought of as an internal designator/name for the machine.

A black background with white text

Description automatically generated

Option 3 – System Release Information:

This gives the information regarding what version the operating system is running. As this document is being written on a Kali Linux machine, this does not show up as Ubuntu.

A screenshot of a computer

Description automatically generated

Option 4 – Kernel Version:

Another simple option, this displays the kernel version of the system.

A close-up of a logo

Description automatically generated

Option 5 – Total System Memory:  
This option displays the amount of memory (RAM) that is on the machine. This is not the amount that is available, this is the total.

A screen shot of a computer

Description automatically generated

Option 6 – Show Terminal History  
This option modifies a bash script, *historyOG.sh,* and creates a new, edited copy. The script is simple, showing recent lines of terminal history. The number of lines shown is specified by the user. While the C program creates the copy of the original script, it edits code to show the number of lines set by the user. This is stored as *historyNEW.sh*.   
**Note:** If you request more lines of history than you have, the script will output every line of history.

A screenshot of a computer

Description automatically generated

Option 7 – Confirm Script Modification  
This option is paired with Option 6 and is used to confirm that the *historyNEW.sh* script exists and is a modified version of *historyOG.sh*. The gives the user a simple way to check if the operations of Option 6 were successful outside of running the entire option again. This process will not work unless Option 6 has been ran prior, as there would be no copied script to check.

A screen shot of a computer

Description automatically generated

Option 8 – CPU Core Count:  
Option 8 utilizes the *coreCount.sh* script to find and output the number of CPU cores on the system. Note that this displays the number of cores, not the total number of threads.

A black background with white text

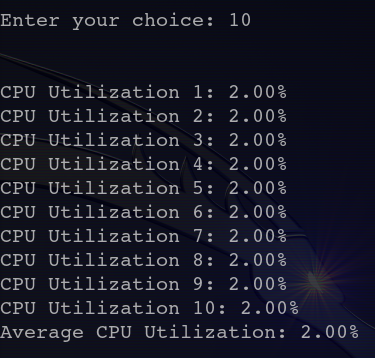
Description automatically generated

Option 9 – Average CPU Temperature:  
This option works with *avgTemp.sh* to output the average temperature of your CPU cores. **This option is incompatible with virtual machines, as they largely do not have access to their own CPU temperature.**

A screen shot of a television screen

Description automatically generated

Option 10 – Approximate CPU Total Utilization:  
To approximate the CPU utilization, Option 10 takes the outputs of *procUtil.sh,* organizes them into an array, and outputs that information as well as the average of them all.



Option 11 – Check for Packages  
Option 11 uses *checkPackages.sh* to confirm if the system has the bc and lm-sensors packages, which are needed for a few options. If they are not, the program automatically installs them.

A screen shot of a computer

Description automatically generated

Option 12 – Exit  
Very simply, this option exits the program.

A black background with white text

Description automatically generated

**Bash Scripts:**  
This program uses three Bash scripts to perform a variety of operations. All bash scripts are stored in the *bash/*directory contained within the program’s directory. There is no need for the user to *chmod* them and set them as executables, this is handled automatically by the program. *In the following images, please ignore any red error indicators. Many of these scripts were developed first on Linux, then ported to Windows for further work. My IDE on Windows does not like how the characters from Linux are formatted and reads them as errors, but they work perfectly.*

The first is *avgTemp.sh*, which reads the /sys/class/thermal/thermal\_zone0/temp file to get the CPU’s temperature. It then converts this temperature from millidegrees to regular degrees Celsius. This script is incompatible with VMs.

A computer screen shot of text

Description automatically generated  
  
Next is *coreCount.sh*, which uses a mixture of lscpu and grep to grab the amount of cores on the system.

A computer screen shot of a code

Description automatically generated  
  
There also is *procUtil.sh*. Its purpose is to calculate the current CPU utilization of the system, which it captures at .5 second intervals. It outputs the result of each interval, as well as the overall average. This script needs the package bc installed.

A screen shot of a computer program

Description automatically generated

This is followed by *historyOG.sh*, but a copy called *historyNEW.sh* is made upon running Option 6. While *historyOG.sh* is not intended to be ran and will error out, the *historyNEW.sh* copy displays the user’s terminal history. The number of lines set to be displayed in the script is dependent on the C program.

A screenshot of a computer program

Description automatically generated

Finally, there is *checkPackages.sh*, which checks if the packages bc and lm-sensors are installed. If not, it confirms there is at least 1Gb of storage and that there is a network connection, and if both these conditions are met, automatically installs either or both packages if they are not already.

A screen shot of a computer program

Description automatically generated

A computer screen with orange and white text

Description automatically generated

Project 2 Improvements:

Several improvements were made on the Project 2 code. There was some cleanup of the code and comments, but most notably the CPU Temperature and CPU Utilization options were overhauled and improved. It was confirmed that VMs cannot access the temperature of their host CPUs, which helped stop wasted efforts and continued rewrites of that option. Finally, there was a great improvement in documentation, as the user How-To guide is much more complete.

Error Handling:

There is decent error handling throughout the program. I’m sure it can be broken in twenty different ways by a skilled – or very unskilled – user, but it’s robust enough to survive most contact. Here are some examples of how the error handling works, this is not an exhaustive list:

In the bash scripts that rely on packages, the scripts of course have error handling to make sure those packages exist.

A screen shot of a computer code

Description automatically generated

There is error handling for memory allocation and management throughout the C program:  
A computer screen shot of text

Description automatically generated

Given this C program relies heavily on a multitude of Bash scripts, there of course are plenty of checks for those Bash scripts:

A computer screen with text

Description automatically generated

Finally, for the Terminal History options, a lot of error handling was required to make constant checks that the files exist, can be edited, etc.

A computer screen with text

Description automatically generated

A screen shot of a computer program

Description automatically generated

Testing:

This program was developed on a mixture of Windows and Kali machines, but most testing was done on Kali. Still, two Ubuntu VMs ran using Proxmox were used test the program on actual Ubuntu platforms. The files were easily shared between each other using a GitHub repository.

The first machine was a simple Ubuntu LTS 22.04. This machine gave a quick and dirty CLI that I could more efficiently navigate than a full desktop environment:

A screenshot of a computer

Description automatically generated

The second was a full Ubuntu 22.04 VM. This machine included the desktop environment and was used for full tests to gauge what the average user’s experience would be like:

A screenshot of a computer program

Description automatically generated