Operating Systems

[19CSE213]

Capstone project:

Power consumption and system monitor

Team members:

Harine Vidyasekaran – CH.EN.U4CSE21121

Perla Sree Neha - CH.EN.U4CSE21147

Vyshnavi P – CH.EN.U4CSE21180

Department : CSE

Section : B

Submitted to:

Dr. A. Padmavathi

CSE Department

Abstract

Our project is a GUI application that monitors the **power consumption** and **system monitoring** and **system specifications** of a computer using the Psutil library and obtains information about running processes on the system. It is flexible and easy to use as it works both on macOS and Windows.

Objective

Monitoring Operating systems processes enables us to monitor and display process activity in real time. The objective is to monitor memory usage activity, power (if plugged or not) and more information for both **windows** and **mac os**. The language used is **python**.

Libraries used

1) Psutil

Psutil is a cross-platform library for retrieving information on running processes and system utilization (CPU, memory, disks, network, sensors) in Python. It supports Linux, Windows, macOS, and FreeBSD. With psutil, you can monitor system resources such as CPU usage, memory usage, disk I/O, network I/O, and process information such as process name, PID, CPU usage, memory usage, and more. It provides a simple and easy-to-use API for interacting with the system, making it a powerful tool for system monitoring and administration.

2) Os [for mac]

The `os` library in Python provides a way to interact with the operating system. It includes functions for accessing files and directories, launching external programs, and other system-related tasks. Here are some of the common tasks that the `os` library can be used for:

File and directory operations: functions for creating, deleting, renaming, and modifying files and directories; Process and system information: functions for retrieving information about the current process, as well as the system's environment variables; Interacting with the command line: functions for running command-line commands and scripts, including the ability to pass arguments, and read the output. Overall, the `os` library is a powerful tool for interacting with the operating system in Python, and it can be used for a wide range of system-related tasks.

Q) Why specifically for mac and why not windows?

The `os.getloadavg()` function is not only available on macOS, but also on other Unix-like operating systems, such as Linux and FreeBSD.

The reason why this function is not available on other operating systems, such as Windows, is because the concept of system load average is specific to Unix-like operating systems. The load average represents the average number of processes in the run queue over a certain period of time, and is used to indicate the current level of system utilization.

Since Windows has a different system architecture and does not use the concept of load average, the `os.getloadavg()` function is not available on that platform.

3) Pprint

`pprint` stands for "pretty print" and is a Python library used for formatting complex data structures like lists, dictionaries, and tuples in a more readable way. By default, Python's built-in `print` function prints out data structures in a compact and often hard-to-read format. `pprint` provides a simple way to output the same data in a more readable and aesthetically pleasing way.`pprint` is especially useful when working with nested data structures like dictionaries containing lists of other dictionaries or other complex objects. It helps to visualize the structure of the data and makes it easier to identify errors or inconsistencies.

In addition to the basic functionality of formatting data structures, `pprint` provides several options to customize the output. For example, you can adjust the indentation of nested data structures, control the width of the output, and choose between a number of different formatting styles.

4) Platform

The `platform` library in Python provides an interface to access various system-specific parameters and functions, including the underlying operating system, hardware architecture, and version information. It is a built-in module in Python and does not require any external installation. The `platform` library provides a set of functions that allow Python scripts to get information about the system on which they are running. This information includes the following:

- The name of the underlying operating system - version of os - hardware architecture (e.g., 32-bit or 64-bit) – python implementation (e.g., CPython, Jython, or IronPython) - version of the Python implementation - system's hostname - machine's processor - number of processors available and more

5) Tkinter

Tkinter is a standard Python library that provides a graphical user interface (GUI) toolkit for building desktop applications. It allows developers to create windows, buttons, menus, text boxes, and other GUI elements that users can interact with. Tkinter is based on the Tcl/Tk GUI toolkit and is included with most Python installations.

Tkinter provides a set of classes and functions that developers can use to build GUI applications. These include: widgets,dialogs,geometry manager,event handling. Tkinter is easy to learn and use, making it a popular choice for building simple desktop applications. It is also powerful enough to build complex applications with multiple windows and menus. Tkinter is cross-platform, meaning it works on Windows, Mac, and Linux, and supports many different programming languages, including Python 2 and Python 3.

6) Ttk

Tkinter ttk, on the other hand, is an extension to the Tkinter module, providing a set of additional widgets that are not available in the standard Tkinter module. These widgets have a modern look and feel and provide better functionality compared to the standard Tkinter widgets. The ttk module provides a set of widgets that are not only more aesthetically pleasing than the standard Tkinter widgets, but they also have improved functionality. Some of the additional widgets provided by ttk include: combobox, notebook, progressbar, treeview

In addition to the above widgets, ttk also provides a way to style Tkinter widgets, which allows developers to create custom looks for their applications. Overall, ttk is a great extension to the standard Tkinter module, providing additional widgets and improved functionality, while maintaining backward compatibility with existing Tkinter code.

Functions used

1) psutil.sensors_battery()

The psutil.sensors_battery() function is a part of the Python psutil module, which is used for retrieving system utilization information like CPU, memory, disk usage, network, etc. The sensors_battery() function specifically retrieves battery information on supported platforms (e.g., Windows and Linux).

When called, this function returns a named tuple that contains the following battery-related information: percent: The current battery charge percentage. secsleft: The approximate number of seconds left until the battery is fully discharged or fully charged (depending on whether the battery is currently charging or discharging). power_plugged: A boolean value indicating whether the battery is currently charging (True) or discharging (False). file: The path to the file that provides the battery information.

Here is an example of using psutil.sensors_battery() to retrieve and print the current battery information:

```
import psutil

battery = psutil.sensors_battery()

print(f"Battery percent: {battery.percent}")
print(f"Seconds left: {battery.secsleft}")
print(f"Power plugged in: {battery.power_plugged}")
```

Output:

```
============ RESTART: C:/Users/hp/Do
Battery percent: 75
Seconds left: -2
Power plugged in: True
>>>
```

2) psutil.virtual_memory()

Return statistics about system memory usage as a named tuple including the following fields, expressed in bytes.

Main metrics:

1-total: total physical memory (exclusive swap).

2-available: the memory that can be given instantly to processes without the system going into swap. This is calculated by summing different memory metrics that vary depending on the platform. It is supposed to be used to monitor actual memory usage in a cross-platform fashion.

3-percent: the percentage usage calculated as (total - available) / total * 100.

Other metrics: used, free, active, Inactive, buffers, cached, shared, slab, wired

Here's an example:

```
import psutil

print(psutil.virtual_memory())
```

Output:

```
============ RESTART: C:/Users/hp/Documents/IV SEM/OS 19CS6/demo.py ============
svmem(total=8470712320, available=1159172096, percent=86.3, used=7311540224, free=1159172096)
>>>
```

3) psutil.cpu_percent()

The psutil.cpu_percent() method returns the CPU utilization as a percentage. This method returns a float representing the percentage of CPU utilization by the system as a whole. It accepts an optional parameter interval which specifies the number of seconds to wait before

returning the CPU usage. By default, the interval is set to None, which means the method returns the current CPU utilization without any delay.

Here is an example:

```
import psutil
print(psutil.cpu_percent(4,percpu=True))
```

Output:

```
========== RESTART: C:/Users/hp/Documents/IV SEM/os 19cse/demo.py ==========
[8.0, 0.4, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
>>>
```

Here's another example:

```
import psutil
print(psutil.cpu_percent(4))
```

Output:

```
========== RESTART: C:/Users/hp/Documents
0.6
>>>
```

4) psutil.process_iter() is a function provided by the psutil module in Python that returns an iterator yielding a Process class instance for all running processes on the current system. Each Process instance returned by the iterator provides a number of methods and properties that can be used to retrieve information about the respective process, such as the process ID, name, CPU and memory usage, open files and sockets, and more.

psutil.process_iter(attrs=None, ad_value=None) returns an iterator yielding a Process class instance for all running processes on the local machine. This should be preferred over psutil.pids() to iterate over processes as it's safe from race condition. Every Process instance is only created once, and then cached for the next time psutil.process_iter() is called (if PID is still alive). Also it makes sure process PIDs are not reused.

attrs and ad_value have the same meaning as in Process.as_dict(). If attrs is specified Process.as_dict() result will be stored as a info attribute attached to the returned Process instances. If attrs is an empty list it will retrieve all process info (slow).

Example:

```
import psutil
import pprint
process_list = []
for process in psutil.process_iter():
    process_info = process.as_dict(attrs=['name', 'cpu_percent'])
    process_list.append(process_info)
    pprint.pprint(process_list)
```

Output:

```
{'cpu_percent': 0.0, 'name': 'IntelCpHDCPSvc.exe'},
 {'cpu_percent': 0.0, 'name': 'svchost.exe'},
 {'cpu_percent': 0.0, 'name': 'svchost.exe'},
 {'cpu_percent': 0.0, 'name': 'RuntimeBroker.exe'},
 {'cpu_percent': 0.0, 'name': 'svchost.exe'},
 {'cpu_percent': 0.0, 'name': 'svchost.exe'},
 {'cpu_percent': 0.0, 'name': 'chrome.exe'},
 {'cpu_percent': 0.0, 'name': 'svchost.exe'},
 {'cpu_percent': 0.0, 'name': 'LockApp.exe'},
 {'cpu_percent': 0.0, 'name': 'svchost.exe'},
 {'cpu_percent': 0.0, 'name': 'svchost.exe'},
 {'cpu_percent': 0.0, 'name': 'svchost.exe'},
 {'cpu_percent': 0.0, 'name': 'svchost.exe'},
 {'cpu_percent': 0.0, 'name': 'svchost.exe'},
 {'cpu_percent': 0.0, 'name': 'svchost.exe'},
 {'cpu_percent': 0.0, 'name': 'unsecapp.exe'},
 {'cpu_percent': 0.0, 'name': 'svchost.exe'},
 {'cpu_percent': 0.0, 'name': 'svchost.exe'},
 {'cpu_percent': 0.0, 'name': 'WmiPrvSE.exe'},
 {'cpu_percent': 0.0, 'name': 'SynTPEnhService.exe'},
 {'cpu_percent': 0.0, 'name': 'svchost.exe'},
 {'cpu_percent': 0.0, 'name': 'svchost.exe'},
 {'cpu_percent': 0.0, 'name': 'svchost.exe'},
 {'cpu_percent': 0.0, 'name': 'svchost.exe'},
 {'cpu_percent': 0.0, 'name': 'AdobeCollabSync.exe'}]
[{'cpu_percent': 0.0, 'name': 'System Idle Process'},
 {'cpu_percent': 0.0, 'name': 'System'},
 {'cpu_percent': 0.0, 'name': 'Registry'},
 {'cpu_percent': 0.0, 'name': 'smss.exe'},
 {'cpu_percent': 0.0, 'name': 'csrss.exe'},
 {'cpu_percent': 0.0, 'name': 'wininit.exe'},
 {'cpu_percent': 0.0, 'name': 'csrss.exe'},
 {'cpu_percent': 0.0, 'name': 'winlogon.exe'},
 {'cpu_percent': 0.0, 'name': 'services.exe'},
 {'cpu_percent': 0.0, 'name': 'lsass.exe'},
 {'cpu_percent': 0.0, 'name': 'svchost.exe'},
 {'cpu_percent': 0.0, 'name': 'fontdrvhost.exe'},
 {'cpu_percent': 0.0, 'name': 'fontdrvhost.exe'},
 {'cpu_percent': 0.0, 'name': 'WUDFHost.exe'},
```

5) platform.system()

The platform.system() function is used to retrieve the system/OS name. It returns a string representing the operating system name such as 'Windows', 'Linux', 'Darwin' (for macOS), etc. This function belongs to the platform module in Python, which provides an interface to retrieve information about the underlying platform, such as the hardware architecture, operating system, and version.

Example for system:

```
import platform
print(platform.system())
```

Output:

```
Windows
>>>
```

6) platform.machine()

`platform.machine()` is a function in the Python `platform` module which returns the machine type where the Python interpreter is running. The return value will be one of the following strings:

- 'x86_64' (for a 64-bit Intel processor)

- 'i386' or 'i686' (for a 32-bit Intel processor)

- 'armv7l' or 'armv6l' (for a Raspberry Pi or other ARM-based device)

- 'aarch64' (for a 64-bit ARM processor)

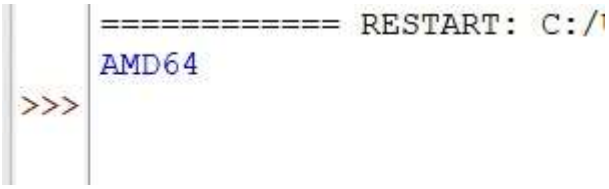
- 'ppc64le' (for a PowerPC 64-bit Little Endian processor)

This function is useful when you need to write platform-specific code or when you want to determine which binary to download for a particular platform.

Example for machine:

```
import platform
print(platform.machine())
```

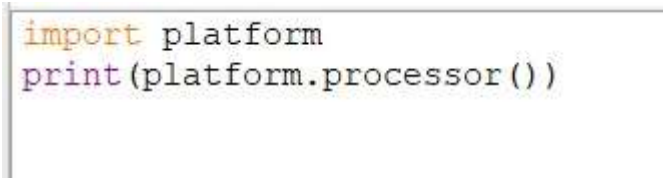
Output:



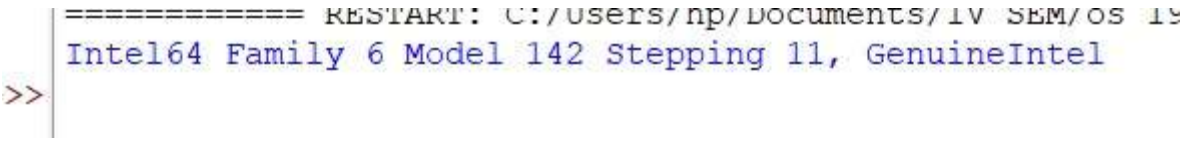
7) platform.processor()

platform.processor() is a method in the platform module in Python that returns the processor name of the current system. The processor name is a string that describes the type of processor, including its architecture and clock speed.

Example:



Output:



8) `os.getloadavg()`

This is a method in Python's built-in `os` module that returns the average system load over a given period of time. Specifically, it returns a tuple of three floating-point numbers, representing the load averages over the last 1, 5, and 15 minutes, respectively.

The load average is a measure of the amount of work that a computer system is performing. It represents the average number of processes that are either in a runnable state or waiting for I/O operations to complete. A higher load average indicates that the system is under heavier load and may be less responsive.

By using `os.getloadavg()`, a Python program can obtain information about the current system load and use this information to make decisions about how to allocate resources or schedule tasks. For example, a program that runs background jobs may use the load average to decide whether to start new jobs or wait until the system load decreases.

Example:

```
hell.py - /Users/sreeneha/D
import psutil
import os

load1, load5, load15 = psutil.getloadavg()
cpu_usage = (load15/os.cpu_count())*100
print(cpu_usage)
print(os.getloadavg())
```

Output:

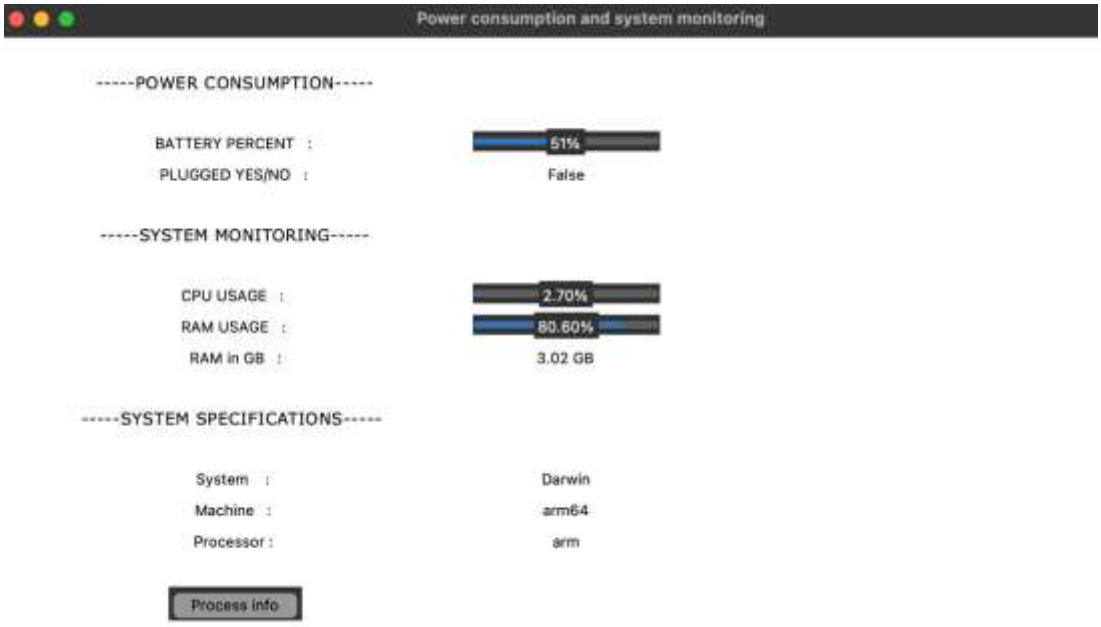
```
27.0263671875
(1.734375, 2.03076171875, 2.162109375)
>>>
```

Output screenshot

Windows os

Output screenshot

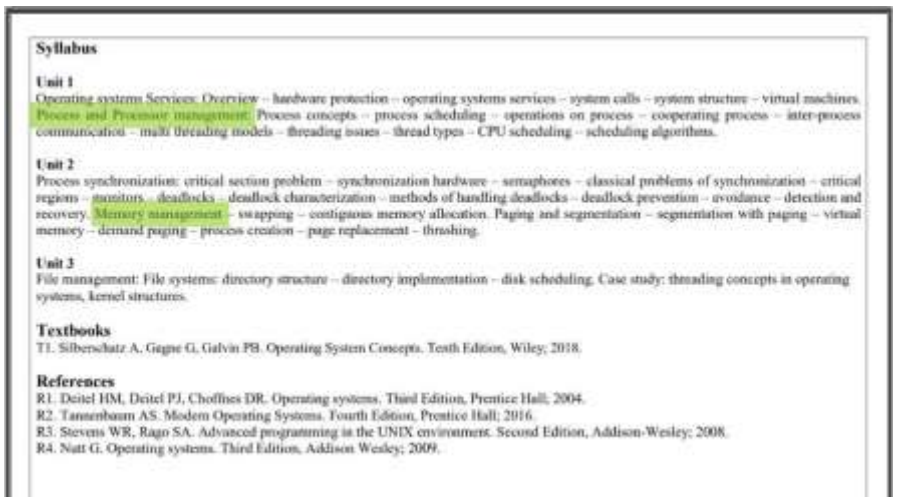
Mac os



Relevance to OS:

This code is relevant to OS because it uses the "psutil" module to obtain system information such as CPU usage, memory usage, and battery status. These parameters are important factors to monitor the health of the OS. Furthermore, it uses the "platform" module to obtain the system specifications such as system name, machine type, and processor type.

Additionally, the code creates a graphical user interface using the Tkinter module to display the system information in real-time. It provides a visual representation of the current state of the system, which is useful for monitoring and maintaining the health of the OS.



Syllabus

Unit 1
Operating systems Services: Overview – hardware protection – operating systems services – system calls – system structure – virtual machines. Process and Processor management: Process concepts – process scheduling – operations on process – cooperating process – inter-process communication – multi threading models – threading issues – thread types – CPU scheduling – scheduling algorithms.

Unit 2
Process synchronization: critical section problem – synchronization hardware – semaphores – classical problems of synchronization – critical regions – monitors – deadlocks – deadlock characterization – methods of handling deadlocks – deadlock prevention – avoidance – detection and recovery. Memory management – swapping – contiguous memory allocation. Paging and segmentation – segmentation with paging – virtual memory – demand paging – process creation – page replacement – thrashing.

Unit 3
File management: File systems: directory structure – directory implementation – disk scheduling. Case study: threading concepts in operating systems, kernel structures.

Textbooks
T1. Silberschatz A, Gagne G, Galvin PB. Operating System Concepts. Tenth Edition, Wiley; 2018.

References
R1. Deitel HM, Deitel PJ, Choffnes DR. Operating systems. Third Edition, Prentice Hall; 2004.
R2. Tanenbaum AS. Modern Operating Systems. Fourth Edition, Prentice Hall; 2016.
R3. Stevens WR, Rago SA. Advanced programming in the UNIX environment. Second Edition, Addison-Wesley; 2008.
R4. Natt G. Operating systems. Third Edition, Addison Wesley; 2009.

Conclusion:

To conclude, our project is a simple and useful application that helps users to monitor their system performance, power consumption and system specifications. It is flexible and easy to use as it works both on macOS and Windows. It is also well structured and user friendly with a Graphical User Interface (GUI).