System Programming Using Python Programming Language

Lab 01

Python is a high-level programming language that is often used for web development, scientific computing, data analysis, and machine learning. While Python is not typically thought of as a systems programming language, it can be used for system programming tasks.

System programming involves writing code that interacts with low-level system components such as the operating system, hardware, and network. Python provides several modules and libraries that make it possible to perform system programming tasks. Some of these include:

- `OS Module` **- This module provides a way to interact with the operating system, such as creating or deleting files and directories, changing permissions, and more.**
- `Subprocess Module` **- This module allows you to spawn new processes, connect to their input/output/error pipes, and obtain their return codes.**
- `Socket Module` **- This module provides a way to communicate with network sockets, allowing Python programs to interact with the internet, send and receive data through network connections.**
- `ctypes Module` **- This module provides a way to call functions in shared libraries and access C data types from Python code.**
- `multiprocessing Module` **- This module provides a way to create and manage multiple processes in a Python program.**

Using these modules and libraries, you can perform many system programming tasks using Python. For example, you can write a Python script to interact with the operating system, perform system administration tasks such as managing user accounts and groups, and manipulate network sockets to create a web server or a chat application.

Python can also be used to write system-level programs, such as device drivers, operating systems, and low-level libraries. While Python may not be the ideal choice for all system programming tasks, it can be a useful tool in many situations.

Handling the Current Working Directory

1. **Get the Current working directory**

In [ ]:

```
import os
```

In [ ]:

```
cwd = os.getcwd()
cwd
```

1. **Changing the Current working directory**

In [ ]:

```
# function which prints the current woring directory path
def cWDPath():
    print(os.getcwd())
```

```
# prints current dir. path
cWDPath()

# change the path
os.chdir("../")

# again prints the path
cWDPath()
```

1. **Creating a Directory**

In [ ]:

```
# name of directory You want to make
dirName = "System Programming"

# parent dir. path
parentDir = "C:/Users/hp/Google Drive/Fiverr Work/2022/34. System Programming Using Pytho
n/Lab 01/"

# join parent with new
path  = os.path.join(parentDir, dirName)

os.mkdir(path)
print(f"Directory {dirName} is created!")
```

1. **Listing out Files and Directories**

In [ ]:

```
# provide the path of dir.
path  = "C:\\Users\\hp\\Google Drive\\Fiverr Work\\2022"
```

In [ ]:

```
dirList = os.listdir(path)
dirList
```

In [ ]:

```
print(len(dirList))
```

1. **Deleting Directory or Files**

In [ ]:

```
# just give the path with the name of that file
path = "C:\\Users\\hp\Google Drive\\Fiverr Work\\2022\\34. System Programming Using Pyth
on\\Lab 01\\System Programming\\11.txt"
os.remove(path)
```

1. **os.name**

os.name is a variable in Python's built-in os module that returns the name of the operating system that the Python interpreter is running on.

The value of os.name can be one of two possible strings: "nt" for Windows, or "posix" for most Unix-based systems, including Linux and macOS.

In [ ]:

```
import os

if os.name == "nt":
    print("Running on Windows")
elif os.name == "posix":
    print("Running on UNIX based System")
else:
    print("Unknown Operating System")
```

1. **os.times()**

os.times() is a function in Python's built-in os module that returns a tuple of floating-point numbers representing the current process's user and system CPU times. The values in the tuple represent the following:

- `user:` **- The amount of CPU time spent in user-mode code by the current process, in seconds.**
- `system:` **- The amount of CPU time spent in system-mode code by the current process, in seconds.**
- `children_user:` **- The amount of CPU time spent in user-mode code by child processes of the current process, in seconds.**
- `children_system:` **- The amount of CPU time spent in system-mode code by child processes of the current process, in seconds.**

In [ ]:

```
os.times()
```

In [ ]:

```
import os

startTime = os.times()

# Perform some CPU-intensive work
for i in range(10000000):
    x = i * i

endTime = os.times()

userTime = endTime.user - startTime.user
systemTime = endTime.system - startTime.system

print("User CPU time:", userTime, "seconds")
print("System CPU time:", systemTime, "seconds")
```

The difference between the CPU times is calculated to get the amount of time spent by the process in user and system mode.

Note that os.times() may not be available on all platforms, and the values returned by the function may not be very accurate or precise. For more precise timing measurements, you may want to use the time or timeit modules instead.

In the example I provided, os.times() is used to measure the CPU time used by the current process before and after performing some CPU-intensive work. The user and system values returned by os.times() represent the amount of CPU time used by the current process in user-mode and system-mode code, respectively.

- `User CPU time:` **- The user value returned by os.times() represents the amount of CPU time spent executing code in user-mode. This includes time spent executing the code in the current process, as well as any user-mode code executed by child processes.**
- `System CPU time:` **- The system value returned by os.times() represents the amount of CPU time spent executing code in kernel-mode (i.e., system-mode). This includes time spent executing system calls, interrupts, and other operating system functions, as well as any system-mode code executed by child processes.**

In the example I provided, the CPU-intensive work is performed by the current process, so the majority of the CPU time will be in user-mode. However, if the process were to make a lot of system calls, or if child processes were created and did a significant amount of work, then the system CPU time would increase as well.

In general, user CPU time represents the amount of time spent executing your application code, while system CPU time represents the amount of time spent executing operating system code on behalf of your application.

Both can be important to monitor and optimize for different types of applications.

1. **Check if file exist**

In [ ]:

```
import os

try:
    fileName = "tt.txt"
    f = open(fileName, "r")
    text = f.read()
    print(text)
    f.close()
except IOError:

    print(f"Problem in reading: {fileName}")
```

File Traversing

1. **os.walk()**

`os.walk()` - is a function in Python's built-in os module that generates the file names in a directory tree by walking the tree either top-down or bottom-up. The function returns a generator that yields a 3-tuple (dirpath, dirnames, filenames) for each directory in the tree, including the starting directory.

The parameters of os.walk() are:

- `top:` **- The directory at the top of the directory tree to start the walk from.**
- `topdown:` **- A Boolean flag indicating whether to generate the directory tree top-down (i.e., start from the top-level directory and descend into subdirectories) or bottom-up (i.e., start from the leaf directories and work up to the top-level directory). The default is True, which means the directory tree is generated top-down.**
- `onerror:` **- An optional function that is called when an error occurs while walking the directory tree.**

The 3-tuple (dirpath, dirnames, filenames) returned by os.walk() contains the following:

- `dirpath:` **- The path of the directory currently being visited.**
- `dirnames:` **- A list of the names of the subdirectories in dirpath.**
- `filenames:` **- A list of the names of the files in dirpath.**

In [ ]:

```
import os
fileNames = []
dirNames = []
fileSize = []
filePath = []
topDirPath = "C:\\Users\\hp\Google Drive\\Fiverr Work\\2022\\34. System Programming Usin
g Python\\Lab 01"

for dirPath, dirName, fileName in os.walk(topDirPath):

    for file in fileName:
        if file is not None:
            fileNames.append(file)
            filePath.append(dirPath)
            fileSize.append(os.path.getsize(dirPath+f"\\{file}"))

    for dir in dirName:
       if dir is not None:
            dirNames.append(dir)
            # dirSize = os.path.getsize(dir)
```

```
print(f"{fileSize} bytes")
# print(f"{dirSize} bytes")

print(f"Total number of files: {len(fileNames)}","\n", fileNames)

print(f"\n Total number of directories: {len(dirNames)}","\n", dirNames)
```

In [ ]:

```
import os

# Set the directory path
directory_path = "C:\\Users\\hp\\Google Drive\\Fiverr Work\\2022"

# Traverse the directory
for root, directories, files in os.walk(directory_path):
    # Loop through the files in the directory
    for file in files:
        # Get the file path
        file_path = os.path.join(root, file)
        # Print the file path
        print(file_path)
```

In [ ]:

```
import os
import matplotlib.pyplot as plt

def get_folder_sizes(folder):
    sizes = {}
    for root, dirs, files in os.walk(folder):
        path = root.split(os.sep)
        for file in files:
            size = os.path.getsize(os.path.join(root, file))
            ext = os.path.splitext(file)[1][1:].upper()
            sizes[ext] = sizes.get(ext, 0) + size
    return sizes

def display_pie_chart(sizes):
    labels = []
    values = []
    for key, value in sizes.items():
        labels.append(key)
        values.append(value)
    fig, ax = plt.subplots()
    ax.pie(values, labels=labels, autopct='%1.1f%%')
    ax.axis('equal')
    plt.show()

if __name__ == '__main__':
    folder = "C:\\Users\\hp\\Google Drive\\Fiverr Work\\2022\\34. System Programming Usi
ng Python"
    sizes = get_folder_sizes(folder)
    display_pie_chart(sizes)
```

In [ ]:

```
os.system("ls -l")
```

In [ ]:

```

```