Usage

Start the CPU and Memory monitor:

`python3 monitor.py`

`python3 stress_test.py [CPU Percentage] [Execution time] [Memory in MB]`

Default

`python3 stress_test.py`

Default CPU: 100%

Default Time: 60 sec

Default Memory: Total PC memory

Example

`python3 stress_test.py 50 10 7000`

Stresses CPU at 50% and 7GB memory for 10 sec.

`python3 stress_test.py 60`

Stresses CPU at 60% and all memory for 60 sec.

Walkthrough:

The fundamental idea is to control the amount of CPU usage by stressing a fraction of cores at maximum percentage. To do this we use the `multiprocessing` module. We do the necessary imports:

```
 from multiprocessing import Process, active_children, Pipe
import os
import signal
import sys
import time
import psutil
```

The `psutil` module enables user to monitor the system statistics from an outside view independent of the process. We provide the default values for time, cpu count, memory. We also define gigabyte and megabyte variables.

```
 DEFAULT_TIME = 60
TOTAL_CPU = psutil.cpu_count(logical=True)
DEFAULT_MEMORY = (psutil.virtual_memory().total >> 20)*1000
PERCENT = 100
GIGA = 2 ** 30
MEGA = 2 ** 20
```

`psutil.cpu_count()` is a method that returns the number of cores available on the system. The default argument for the `logical` parameter is `True`. Most systems have different physical cores and logical cores. We're retrieving the total usable cores (logical or physical) for further calculation. `psutil.virtual_memory()` Return statistics about system memory usage as a named tuple including the following fields, expressed in bytes.

Main metrics:

total: total physical memory (exclusive swap).

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

Other metrics:

used: memory used, calculated differently depending on the platform and designed for informational purposes only. total - free does not necessarily match used.

free: memory not being used at all (zeroed) that is readily available; note that this doesn't reflect the actual memory available (use available instead). total - used does not necessarily match free.

The following piece of code: `DEFAULT_MEMORY = (psutil.virtual_memory().total >> 20)*1000` takes the total memory and raising it up in Megabytes and multiplying it by 1000 to get an accurate floating point value of total memory available in GB. The default memory is total available memory on the system.

The arguments for CPU, time and memory are passed through the commandline and the usage is mentioned in the Usage section.

```
def get_args():
  '''
    Function to assign commandline arguments if passed.

    Returns:
        exec_time  : Execution time in seconds, default = 60
        proc_num   : Number of processors required according
                     to the percentage input by the user.
                     Default = total cpu count of the system
        memory     : Memory in Megabytes to be consumed.
                     Default = Total system memory
        percent    : Percentage of CPU to be used
    '''
    exec_time = DEFAULT_TIME
    proc_num = TOTAL_CPU
    percent = 100
    memory = DEFAULT_MEMORY
    if(len(sys.argv) > 4):
        raise
    if(len(sys.argv) == 2):
        percent = int(sys.argv[1])
        if(percent > 100):
            raise
        proc_num = (percent * TOTAL_CPU)/100
    if(len(sys.argv) == 3):
        percent = int(sys.argv[1])
        if(percent > 100):
            raise
        proc_num = (percent * TOTAL_CPU)/100
        exec_time = int(sys.argv[2])
    if(len(sys.argv) == 4):
        percent = int(sys.argv[1])
        proc_num = (percent * TOTAL_CPU)/100
        exec_time = int(sys.argv[2])
        memory = int(sys.argv[3])
        if(percent > 100 or memory > DEFAULT_MEMORY):
            raise

    return exec_time, proc_num, percent, memory
```

The following `try-except` block in the `_main()` takes care of raised exceptions:

```
def _main():
    try:
        exec_time, proc_num, cpu_percent, memory = get_args()
        global PERCENT
        PERCENT = cpu_percent
    except:
        msg = "Usage: stress_test [CPU percent] [exec_time] [Memory in MB]"
        sys.stderr.write(msg)
        constraints = "\nCPU < 100 and memory < "+str(DEFAULT_MEMORY)
        sys.stderr.write(constraints)
        sys.exit(1)
```

Memory Stress:

The memory stressing is done by assigning a string of blank spaces in increments of 256 MB.

```
def alloc_max_str(memory):
    '''
    Function to load memory by assigning string of requested size

    Arguments:
        memory: amount of memory to be utilized in MB
    Returns:
        a : String of size 'memory'
    '''
    i = 0
    a = ''
    while True:
        try:
            a = ' ' * (i * 256 * MEGA)
            if((psutil.virtual_memory().used >> 20) > memory):
                break
            del a
        except MemoryError:
            break
        i += 1
    return a
```

`a` is an empty string which is initialized and then a `while` block is initiated. The `try` and `except` block is put in place incase the program runs out of memory. `a = ' ' * (i * 256 * MEGA)` creates a string of size 256 MB in multiples of `i`. With each iteration, `i` will increase and the subsequent string variable will be in multiples of `256`. The size of the current used memory is compared with the requested memory using `(psutil.virtual_memory().used >> 20) > memory` and if the string assigned is of the requested size, the loop breaks and returns the string to the calling method. If the loop iterates, it deletes the previously created object and assigns a new string variable (`del a`).

CPU Stress:

The CPU is stressed by creating multiple processes and connecting each processes in a Parent-child pipe for a two way communication. The number of processes to be created depends on the number of processors that are calculated according to user percentage.

```
     actual_cores = int(proc_num)
    last_core_usage = round((proc_num-actual_cores),2)*100
    proc_num = actual_cores
```

This piece of code block splits the number of cores (currently fractional) into an integer and a floating point value of the remainder core. (e.g. if the CPU usage entered corresponds to 3.6 cores, `actual_cores` will be `3` and `last_core_usage` will be `60%`.

A process is created for each core along with a `Pipe()` for communication between parent and child.

```
for i in range(proc_num-1):
        parent_conn, child_conn = Pipe()
        p = Process(target=loop, args=(child_conn,[i], False))
        p.start()
        procs.append(p)
        conns.append(parent_conn)
```

This creates child processes which will be executing the method `loop()` parallelly. The `Process()` method from the `multiprocessing` module enables creation of child processes with `Pipe()` and `Queue()` objects. The `Pipe()` will ensure a communication between the parent and child process for getting the `PID` and the `core affinity` of that particular process in form of a message. The process is targetted to run `loop()` method:

```
def loop(conn, affinity, check):
    '''
    Function to stress cores to run at 100%

    Arguments:
        conn    : child connection which is an object of Pipe()
        affinity: list of cores to assign affinity for the process
        check   : conditional flag to enable real time calibration
    '''
    proc = psutil.Process()
    proc_info = proc.pid
    msg = "Process ID: "+str(proc_info)+" CPU: "+str(affinity[0])
    conn.send(msg)
    conn.close()
    proc.cpu_affinity(affinity)
    while True:
        if(check and psutil.cpu_percent()>PERCENT):
            time.sleep(0.01)            #change to fine tune
        1*1
```

The `cpu_affinity()` assigns a core affinity to a process. The result is that the process runs on the cores specified in the `affinity` list. Currently, we pass one core for every process. This disables sharing of cores and will allow us to control the average usage across all cores. The process executes a simple computation of `1*1` infinitely. If the `check` parameter is true, the core will balance the percentage the core runs at by delaying the calculation with `10ms` and restart the loop according to the total CPU percentage for calibration. For e.g. if the total CPU percentage goes above the desired CPU percentage, the core will reduce its usage to balance the load. We use one of the cores which was supposed to run at `100%` for this task using this snippet:

```
parent_conn, child_conn = Pipe()
    p = Process(target=loop, args=(child_conn,[proc_num-1], True))
    p.start()
    procs.append(p)
    conns.append(parent_conn)
```

We pass the third argument as `True` for this core which is the value for `check` parameter.

Lastly, this snippet is used for fractional load:

```
if(proc_num!=TOTAL_CPU):
        last_core = proc_num
        parent_conn, child_conn = Pipe()
        p = Process(target=last_core_loop, args=(child_conn, [last_core], last_core_usage))
        p.start()
        procs.append(p)
        conns.append(parent_conn)
```

The method `last_core_loop()` allows execution of a core at a specified percentage by conditionally delaying the exection of loop statements.

```
def last_core_loop(conn, affinity, percent):
    '''
    Function to stress the last core at fractional percentage.
    e.g. core 5 at 45% Usage

    Arguments:
        conn    : child connection which is an object of Pipe()
        affinity: list of cores to assign affinity for the process
        percent   : fractional percentage to run the core at
    '''
    proc = psutil.Process()
    proc_info = proc.pid
    msg = "Process ID: "+str(proc_info)+" CPU: "+str(affinity[0])
    conn.send(msg)
    conn.close()
    proc.cpu_affinity(affinity)
    while True:
        if(psutil.cpu_percent(percpu=True)[affinity[0]] > percent):
            time.sleep(0.04)          #change to fine tune
        1*1
```

Here, the usage of the particular core is monitored and if the usage rises above the `percent` value, the loop is delayed by `40ms`. These values are arbitrary and can be fine-tuned for accurate results. For e.g. if the total percentage entered by the user equals to a consumption of `3.6` cores, 3 cores will be running at `100%` and the 4th core will run at `60%`.

Future prospects:

Currently, only a single core is engaged for recalibration according to CPU usage. Meaning, if we want to use 50% of the CPU on a 12 core machine, we're running 5 cores at 100% and the 6th core is balancing itself by reducing usage while monitoring the complete CPU usage. This is done by simply delaying the core execution by 10ms. This approach is not accurate. Furthermore, only one CPU core is engaged for auto balancing the load as opposed to all others in order to avoid undershooting. Implementation of a sophisticated PID algorithm may help reduce error and keep the CPU at a constant load while minimizing the error.

Another odd behaviour that is encountered on my system is when I plug in the laptop to charge. The CPU usage overshoots and doesn't always come down. But this overshooting is only seen in the Task Manager and the `monitor.py` shows the desired results.