**Report: Multiprocessing-OS-sim**

**Description**

This project simulates the multiprocessing functionality of an operating system. It contains a multi-process and multi-thread task manager where all running processes and threads can be viewed, suspended, resumed, and terminated. It also simulates both interprocess and interthread communication using shared memory and message passing. Finally, the project demonstrates multiprocessing by processing a large text file by breaking it up into chunks and processing it in parallel to make every character uppercase and count the occurrence of each character.

# Code Structure

The code contains two main classes, ProcessManager and ThreadManager. They act as the task manager of the system by showing all processes and threads running on the system. They allow users to create, suspend, resume, and terminate tasks as well as send messages between them.

The code also contains a few dummy functions to use as a target for the processes and threads to work on to simulate the creation and communication between them. These functions include:

square\_numbers(): Squares each integer in a list and sums all the squares. It demonstrates IPC with shared memory since the result array and sum variables are shared by both the calling process and worker process

thread\_task(): Increments an integer in a shared array and puts the result back into a shared queue to demonstrate the shared memory communication between threads.

new\_thread() and new\_process() are used as targets for a newly created thread and process respectively which confirms that the new thread or process has been created successfully, simulates work and finishes.

Parallel text processing is done using the following functions:

parallel\_processing(file\_path, part\_size=1024):

This function reads the text file and splits it up into parts of 1024 bytes. Then it creates a pool of workers using the multiprocessing library in Python and maps the processing of each part to an available worker. Finally, it combines the work of each worker and sums up the result to give a final count of each character in the text file.

parts\_to\_process(parts):

This function takes in the parts of texts, converts them into upper case, and counts the occurrence of each character. The pool of threads is working on this function simultaneously with different parts of the text file.

The user interface is housed in the main function in the form of a menu. The menu lets users interact with each class and all available methods in the codebase. It contains elif statements to navigate the program according to the user's input.

**Instructions to use software.**

1. Make sure to have Python 3.7 or higher installed on your computer

2. Download the main.py file

3. Open the file in a Python interpreter

4. Compile and run the program

5. The program should run and display the menu in the console

6. Type in the integers next to the menu options and hit enter

7. Some menu options may ask for a process or thread ID. List all processes or threads and copy and paste a running thread or process ID for ease of use.

8. Some menu options may ask for messages to input.

1. Use space-separated texts. They will have many short messages

2. Use a long string of text (no spaces) to simulate a large message

9. For parallel text processing (Option 15), you will need a text file in the root directory.

1. Download "Text\_file.txt" before running the code

2. If you want to use your own text file, add it to your project directory and make sure to rename the file to "Text\_file.txt"

10. Enter 16 to end the program.

**Code Verification**

User Interface:

A screenshot of a computer

Description automatically generated

List all processes:

A screenshot of a computer

Description automatically generated

List Threads

**A screenshot of a computer

Description automatically generated**

Create Process

**A black rectangle with white dots

Description automatically generated**

**A black rectangular object with white dots

Description automatically generated**

Suspend Process

**A black rectangle with white dots

Description automatically generated**

**A black screen with white text

Description automatically generated**

Resume Process

**A black rectangle with white dots

Description automatically generated**

**A black rectangle with white dots

Description automatically generated**

Kill Process

A black rectangle with white dots

Description automatically generated

Create Thread

A black background with white text

Description automatically generated

A black screen with white text

Description automatically generated

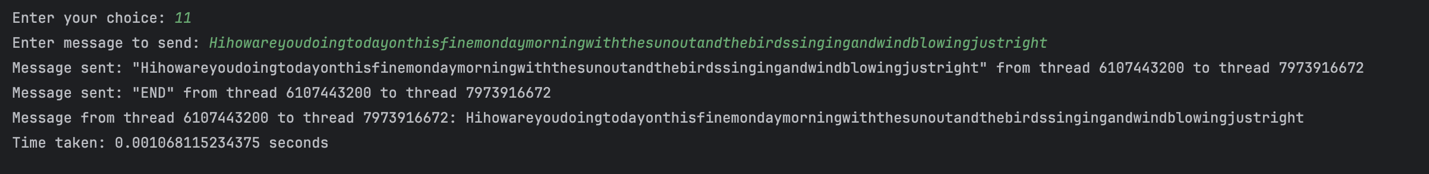
Send Message between Threads via message passing

Short messages:

A screenshot of a computer program

Description automatically generated

Long message:



Send Message between Threads via shared memory

A black background with white text

Description automatically generated

Send Message between Processes via Pipe

Short messages:

A screenshot of a computer

Description automatically generated

Long messages:

A screenshot of a computer

Description automatically generated

Send Message between Processes via shared memory

A black rectangle with white dots

Description automatically generated

Parallel Text File Processing

With 18kb text file:

A screenshot of a computer

Description automatically generated

With 74kb text file:

A screenshot of a computer

Description automatically generated

**Discussion**

This project was a great exercise in understanding the handling of multiple processes and threads in a system, including how to manage them by assigning them to various functions, viewing them, and communicating between them. It was also good to get hands-on experience building such a program because it taught me the intricacies of a multiprogramming machine. It was incredibly challenging getting all of the parts of this project to work, so I can only imagine the level of complexity that goes on behind the scenes in my computer to support the many different processes, threads, I/O operations, etc. It certainly piqued my interest in delving into the topic more and will help me greatly to understand the topics yet to come.

Based on the time taken by the different IPCs, I gathered that sending many short messages is better than a long message since long messages consistently take longer. This was surprising to me because I made each short message print out to console individually and they still took less time.

I also experienced the efficiency of multiprocessing in the text-processing part. Initially, I used an 18kb file to test my code and it worked as expected. However, I found a larger text file of 74kb and saw that increasing the text file 4 times only increased the time taken by 0.04 seconds.

There were a few challenges in this project for me, most of it had to do with simulating IPC. It was not that challenging to understand the different methods of communication but simulating them is where the challenge lay. It was especially challenging because I have them implemented as a menu option, rather than a stand-alone function call. I tried to find a clever function to target the threads and processes to where they would need communication to achieve the task, but I couldn’t come up with any. I also wanted existing processes or threads to communicate and spend a considerable time on it but to no avail. Finally, I landed on just creating the processes right when the IPC methods were called and having them send a few messages back and forth to simulate the communication. With more time, I would have liked to find a better way to simulate IPC.

It was also a great experience working with the multiprocessing library in Python. Being able to map a function to the pool was a convenient way to achieve multiprocessing. It seemed like the most difficult part of the project, but it turned out to be the easiest part for me.