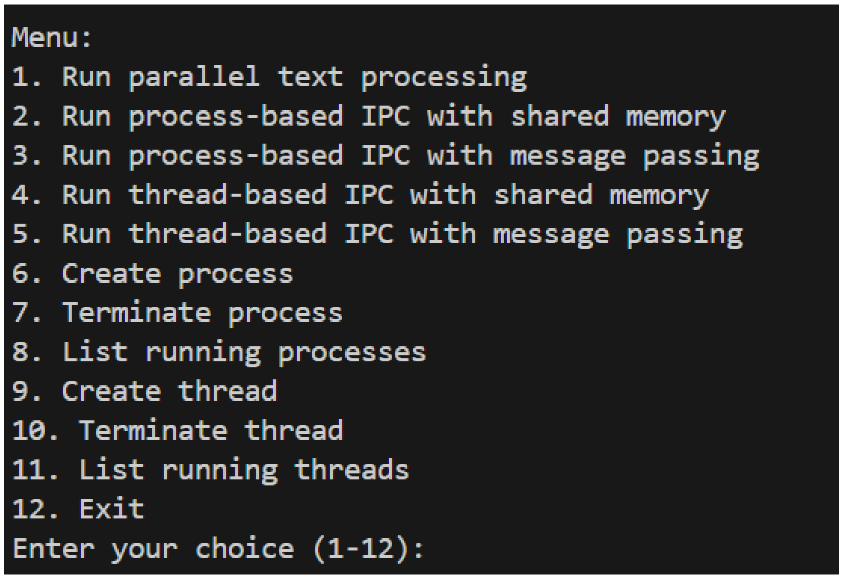
Navid Hoque

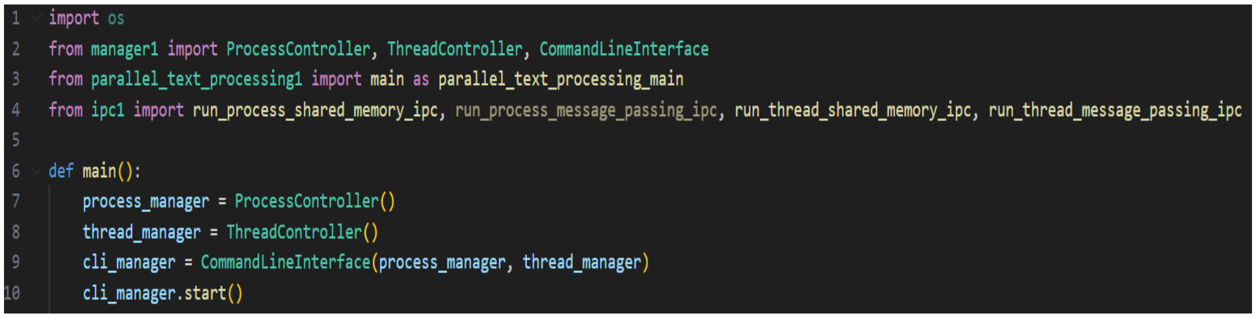
The following project developed a comprehensive operating system simulation that included a multi-process and thread task manager, inter-process communication mechanism, and parallel mathematical operations. The project provided the implementation of this simulation. The project was done using python with the packages and the main packages that would be need to be installed would be psutil. Numpy, and time.

As a whole the whole project runs through the main1.py file which is the CLI for the whole project. This CLI basically collaborates with all of the other files and runs them through this file making a very easy to use application.



```
Menu:
1. Run parallel text processing
2. Run process-based IPC with shared memory
3. Run process-based IPC with message passing
4. Run thread-based IPC with shared memory
5. Run thread-based IPC with message passing
6. Create process
7. Terminate process
8. List running processes
9. Create thread
10. Terminate thread
11. List running threads
12. Exit
Enter your choice (1-12):
```

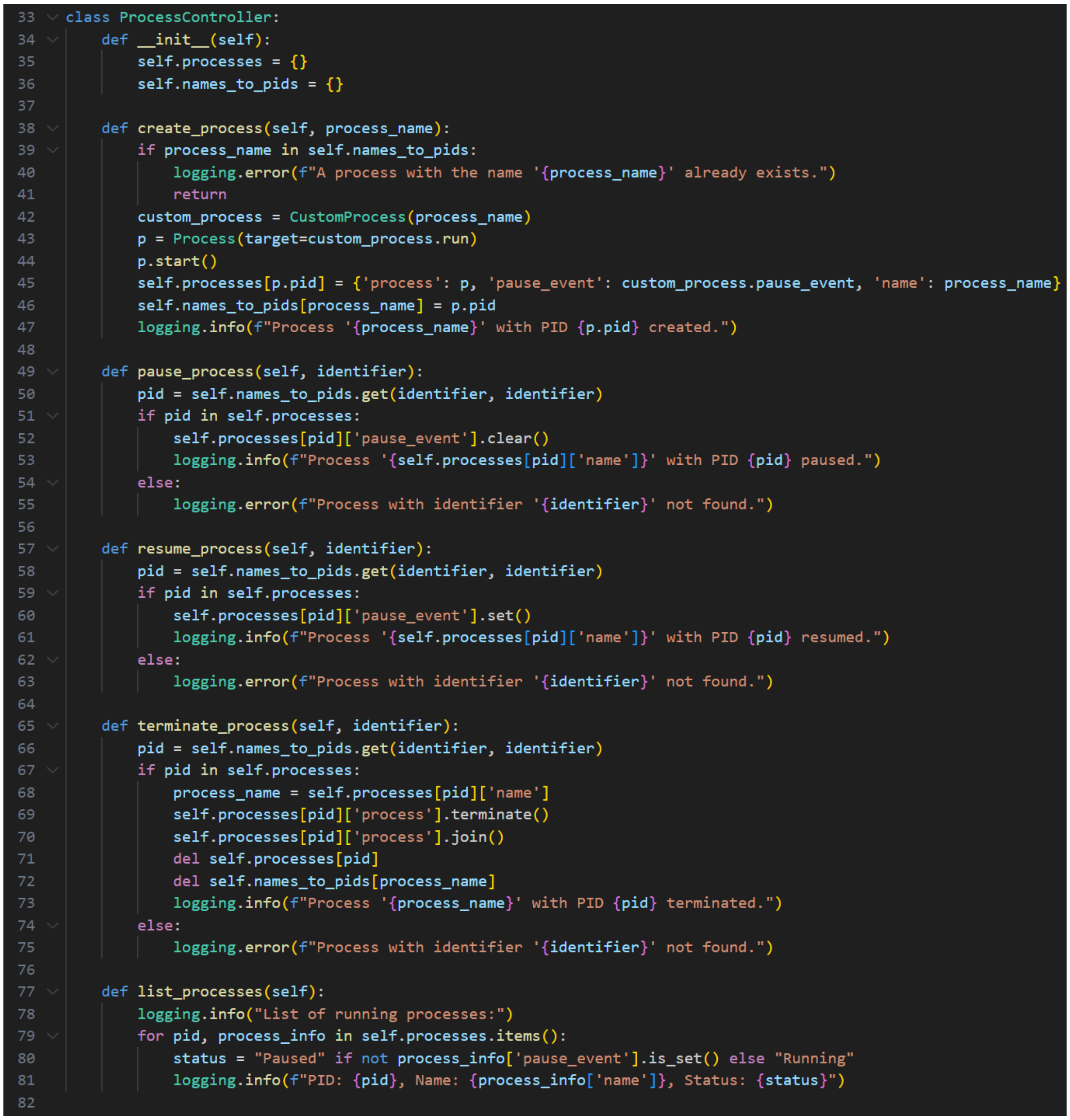
Above is the output of the CLI. Overall by entering a number 1-12 the respective command will be executed.



```python
import os
from manager1 import ProcessController, ThreadController, CommandLineInterface
from parallel_text_processing1 import main as parallel_text_processing_main
from ipc1 import run_process_shared_memory_ipc, run_process_message_passing_ipc, run_thread_shared_memory_ipc, run_thread_message_passing_ipc

def main():
    process_manager = ProcessController()
    thread_manager = ThreadController()
    cli_manager = CommandLineInterface(process_manager, thread_manager)
    cli_manager.start()
```

Above shows the accessing of the other files to be ran through this main1.py file

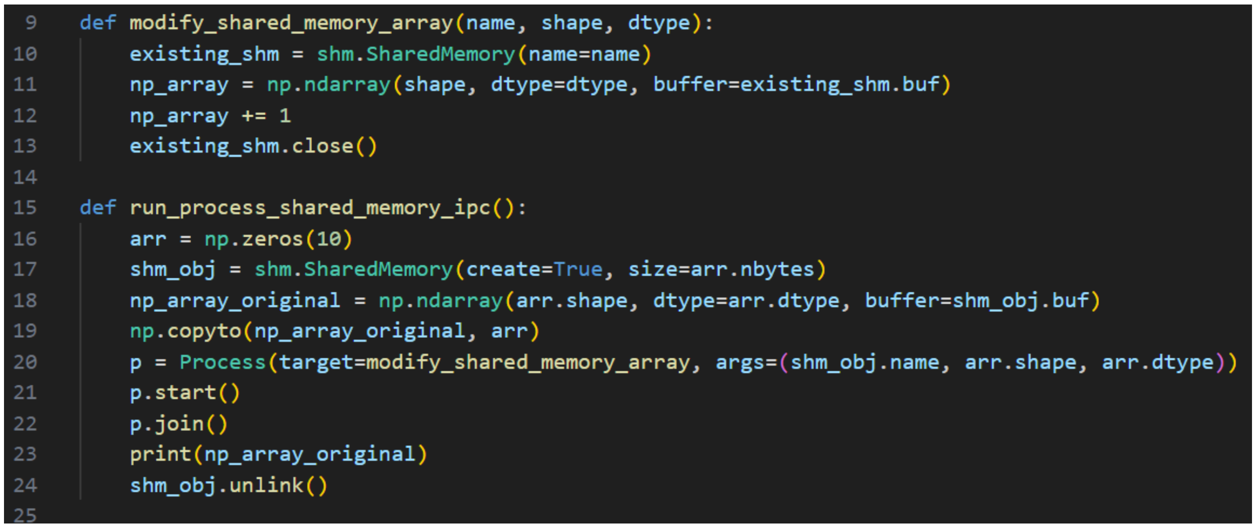
The next aspect of the file would be the manager1.py file which starts with all of the imports and then goes into the functions and those functions are used to create the processes with event handler and do the same thing for the threading. The threads and processes are created manually and then manipulated to list all processes and threads and to terminate them as well.



```python
class ProcessController:
    def __init__(self):
        self.processes = {}
        self.names_to_pids = {}

    def create_process(self, process_name):
        if process_name in self.names_to_pids:
            logging.error(f"A process with the name '{process_name}' already exists.")
            return
        custom_process = CustomProcess(process_name)
        p = Process(target=custom_process.run)
        p.start()
        self.processes[p.pid] = {'process': p, 'pause_event': custom_process.pause_event, 'name': process_name}
        self.names_to_pids[process_name] = p.pid
        logging.info(f"Process '{process_name}' with PID {p.pid} created.")

    def pause_process(self, identifier):
        pid = self.names_to_pids.get(identifier, identifier)
        if pid in self.processes:
            self.processes[pid]['pause_event'].clear()
            logging.info(f"Process '{self.processes[pid]['name']}' with PID {pid} paused.")
        else:
            logging.error(f"Process with identifier '{identifier}' not found.")

    def resume_process(self, identifier):
        pid = self.names_to_pids.get(identifier, identifier)
        if pid in self.processes:
            self.processes[pid]['pause_event'].set()
            logging.info(f"Process '{self.processes[pid]['name']}' with PID {pid} resumed.")
        else:
            logging.error(f"Process with identifier '{identifier}' not found.")

    def terminate_process(self, identifier):
        pid = self.names_to_pids.get(identifier, identifier)
        if pid in self.processes:
            process_name = self.processes[pid]['name']
            self.processes[pid]['process'].terminate()
            self.processes[pid]['process'].join()
            del self.processes[pid]
            del self.names_to_pids[process_name]
            logging.info(f"Process '{process_name}' with PID {pid} terminated.")
        else:
            logging.error(f"Process with identifier '{identifier}' not found.")

    def list_processes(self):
        logging.info("List of running processes:")
        for pid, process_info in self.processes.items():
            status = "Paused" if not process_info['pause_event'].is_set() else "Running"
            logging.info(f"PID: {pid}, Name: {process_info['name']}, Status: {status}")

```

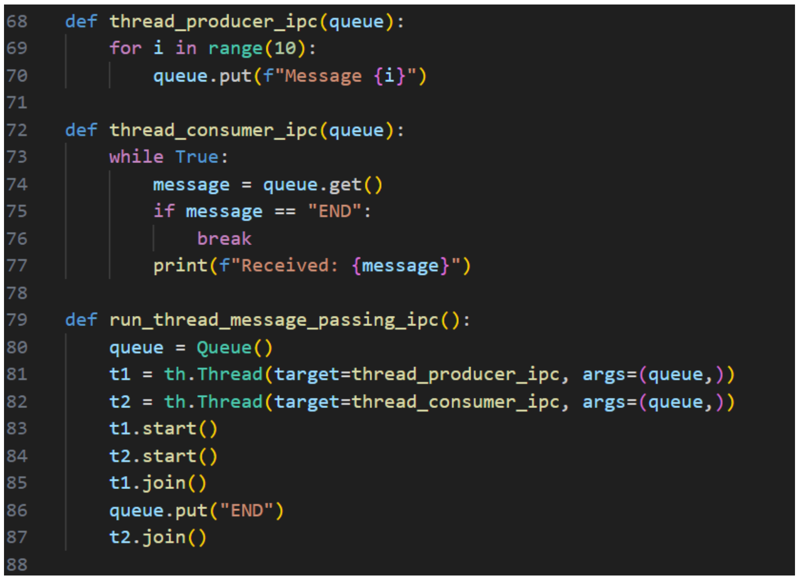
Above is the process controller which contains all of the different functions for the processes and allows for the different functions to be made. the same is basically done for the threads as well.

Another aspect to the project is the IPC1.py file within the project that builds off of the manager and takes the threading and processing and then takes messages and puts them through it. The IPC also as a whole simulates the IPC using shared memory and message passing for both threads and processes. The IPC modifies the shared memory array and then runs process shared memory and thread shared memory. It also has producers and consumers to implement the message passing within the IPC.



```python
def modify_shared_memory_array(name, shape, dtype):
    existing_shm = shm.SharedMemory(name=name)
    np_array = np.ndarray(shape, dtype=dtype, buffer=existing_shm.buf)
    np_array += 1
    existing_shm.close()

def run_process_shared_memory_ipc():
    arr = np.zeros(10)
    shm_obj = shm.SharedMemory(create=True, size=arr.nbytes)
    np_array_original = np.ndarray(arr.shape, dtype=arr.dtype, buffer=shm_obj.buf)
    np.copyto(np_array_original, arr)
    p = Process(target=modify_shared_memory_array, args=(shm_obj.name, arr.shape, arr.dtype))
    p.start()
    p.join()
    print(np_array_original)
    shm_obj.unlink()

```

Above are the functions for the shared memory and the process shared memory

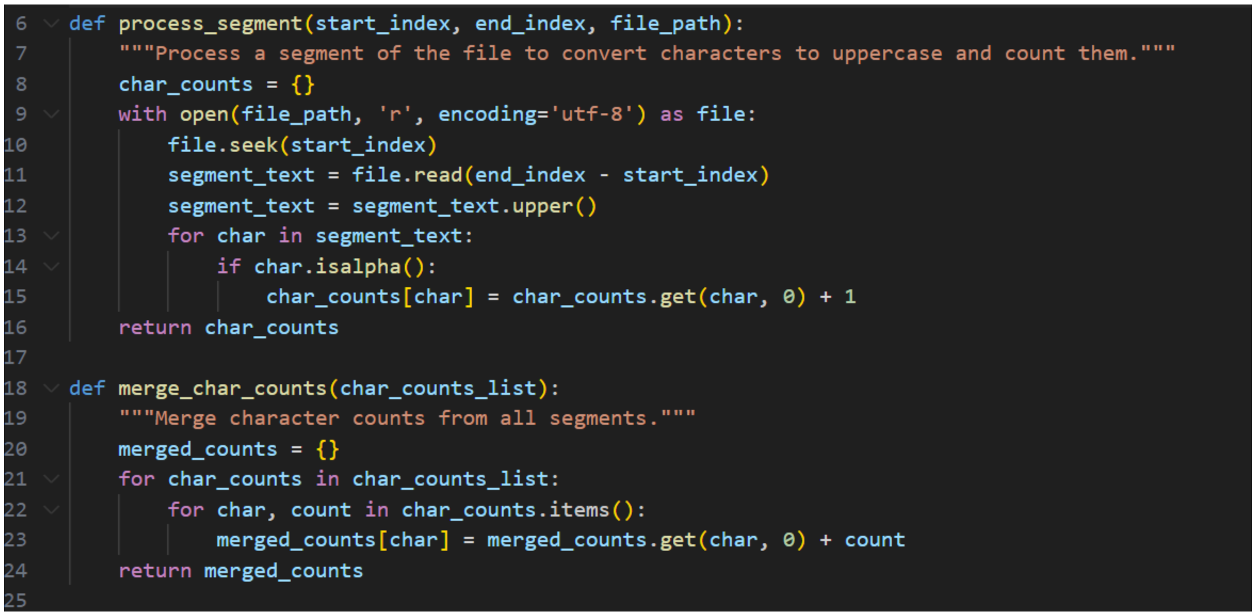


```python
def thread_producer_ipc(queue):
    for i in range(10):
        queue.put(f"Message {i}")

def thread_consumer_ipc(queue):
    while True:
        message = queue.get()
        if message == "END":
            break
        print(f"Received: {message}")

def run_thread_message_passing_ipc():
    queue = Queue()
    t1 = th.Thread(target=thread_producer_ipc, args=(queue,))
    t2 = th.Thread(target=thread_consumer_ipc, args=(queue,))
    t1.start()
    t2.start()
    t1.join()
    queue.put("END")
    t2.join()

```

Above are the functions for the producer and consumer as well as the message passing for the threads

As a whole the implementation for the thread is also done for the process aspect too which allows for this inter process communication mechanism to work.

The final aspect to the project is the parallel_text_processing1.py file which basically will take the .txt file and then lists how many elements within the text file and time it takes to process it in nanoseconds and the CPU usage change and then memory usage increase. Within the parallel text processing contains functions to segment the text file and then converting it to uppercase and count the occurrences and then return a dictionary with the uppercase characters and values of the counts within the segment. Another function for merging the character counts takes all of the segments and their respective dictionary and then merge those counts to the uppercase characters. As a whole this process then takes that and lets the user know the time, CPU usage, and memory usage for the whole process.



```python
def process_segment(start_index, end_index, file_path):
    """Process a segment of the file to convert characters to uppercase and count them."""
    char_counts = {}
    with open(file_path, 'r', encoding='utf-8') as file:
        file.seek(start_index)
        segment_text = file.read(end_index - start_index)
        segment_text = segment_text.upper()
        for char in segment_text:
            if char.isalpha():
                char_counts[char] = char_counts.get(char, 0) + 1
    return char_counts

def merge_char_counts(char_counts_list):
    """Merge character counts from all segments."""
    merged_counts = {}
    for char_counts in char_counts_list:
        for char, count in char_counts.items():
            merged_counts[char] = merged_counts.get(char, 0) + count
    return merged_counts

```

Above is the said functions to segment the text and create the dictionary of uppercase characters and their counts within the text

Screenshots of all of the outputs of the code will be inside of the screenshots folder within the github repository.

As a whole this project showed a better example of all of these different aspects of bigger ideas like multi process and thread manager, inter-process communication mechanisms, parallel text file processing, and using a CLI to bring it all together to make a usable user interface for these different bigger ideas. Throughout the project some of the challenges faced were in the manager1.py file with implementing time with the logging feature because of the list when logging is started and when the logging was ended, but over time it was cleaned up and able to be implemented. Areas to improve for this project would be creating a better CLI to maybe break up these bigger ideas and have them as general projects and then have their respective parts within the CLI for a cleaner use for the user.