Description of the project

Multi-Process and Thread Manager

I divided this feature into two classes, ProcessManager and ThreadManager. The ProcessManager class reveals all the information about all running processes, including the threads associated, status, and their light-weight-processes. The menu option layout allows users to create, terminate, suspend and resume processes.

The ThreadManager class manages the threads and all their information. Users can monitor active threads, create new ones, and suspend them when needed.

Users can interact efficiently though the CLI.

Inter-Process Communication (IPC) Mechanisms

For this part of the project I use IPC to simulate shared memory and message passing to allow communication between different processes.

- IPC over Processes:

For this feature I simulated IPC over processes that use shared memory and message passing to allow communication between different processes. Shared memory allows processes to have access to the same memory space. In order to allow this I used multiprocessing. Array class. For message passing between processes I used the multiprocessing.queue class.

- IPC over Threads:

IPC over threads has a similar structure approach, just in the context of threads. It also includes features to show shared memory and message passing. For shared memory I used the multiprocessing.Array class as well. The main difference is that I used the standard Python queue.

Parallel Text File Processing

The goal for this feature was to convert characters to uppercase and count the occurrences of each character in a large text file, while also leveraging the computational power of the processes. My implementation of this feature divides the text file into 'chunks' of a defined size (determined in further logic). Then each chunk of data is processed independently in parallel. I utilized the parallel\_file\_processing function to create a pool of processes. There all the processes asynchronously processed a chunk of text using the rest of the logic that converted from uppercase to lowercase and counted each occurrence. To test this I used a 4 KB file.

Structure of the Code with Description Regarding Structure

When I first began this project I came up with a plan of what exactly I needed to accomplish, is as follows:

Need to Implement:

* Multiprocess and thread manager: detailed process and thread management functionalities
  + Displaying information about each process
  + Suspending/resuming/killing processes and threads
  + Managing threads within a process
* Inter-process communication mechanisms
  + Implement shared memory and message passing mechanisms for interprocess communication over both processes and threads.
  + Evaluate and compare the performance of each mechanisms with different sizes
* Parallel Text File Processing
  + System for efficiently processing large text files in parallel
  + Mechanisms to load and distribute segments of the file across multiple threads or processes.
  + Implement parallelized operations for converting characters to uppercase and counting character occurrences
* CLI: Allow users to perform actions such as creating processes, managing threads, initiating IPC, and processing text files.

As the project progressed I began to visualize the structure:

class ProcessManager:

# For managing processes

class ThreadManager:

# For managing threads

def main\_menu():

# Main menu for user interaction

def main():

# Main function to execute the program

Descriptions regarding structure

ProcessManager Class:

* create\_processes(self, target, args): will create a new process with the specified target function and arguments provided. This is achieved using the multiprocess library.
* The psutil library gathers important information like process ID, name, status, and associated threads.

ThreadManager Class:

* create\_thread(self, target, argos=()): creates a new thread with the specified target function and arguments provided. This is achieved using the threading library.

IPC Simulation:

* Creates two processes: a sender and a receiver.
* The sender will process the values and increment them as a shared value and will then send messages to the receiver process using a multiprocessing queue.
* Then also simulates the same process with threads. I create two threads: a sender and a receiver. Follows the same logic of incrementing and sharing values.

Parallel Text File Processing:

* My process\_chunk(chunk) method will take a chunk of text and will count the occurrences of the characters.
* I use the multiprocessing.Pool class to divide the chunks into multiple processes that will function/run in parallel.

Instructions on how to use it for full functionalities provided.

Once the program is run a menu of display options will appear so that the user can interact with all features listed above.

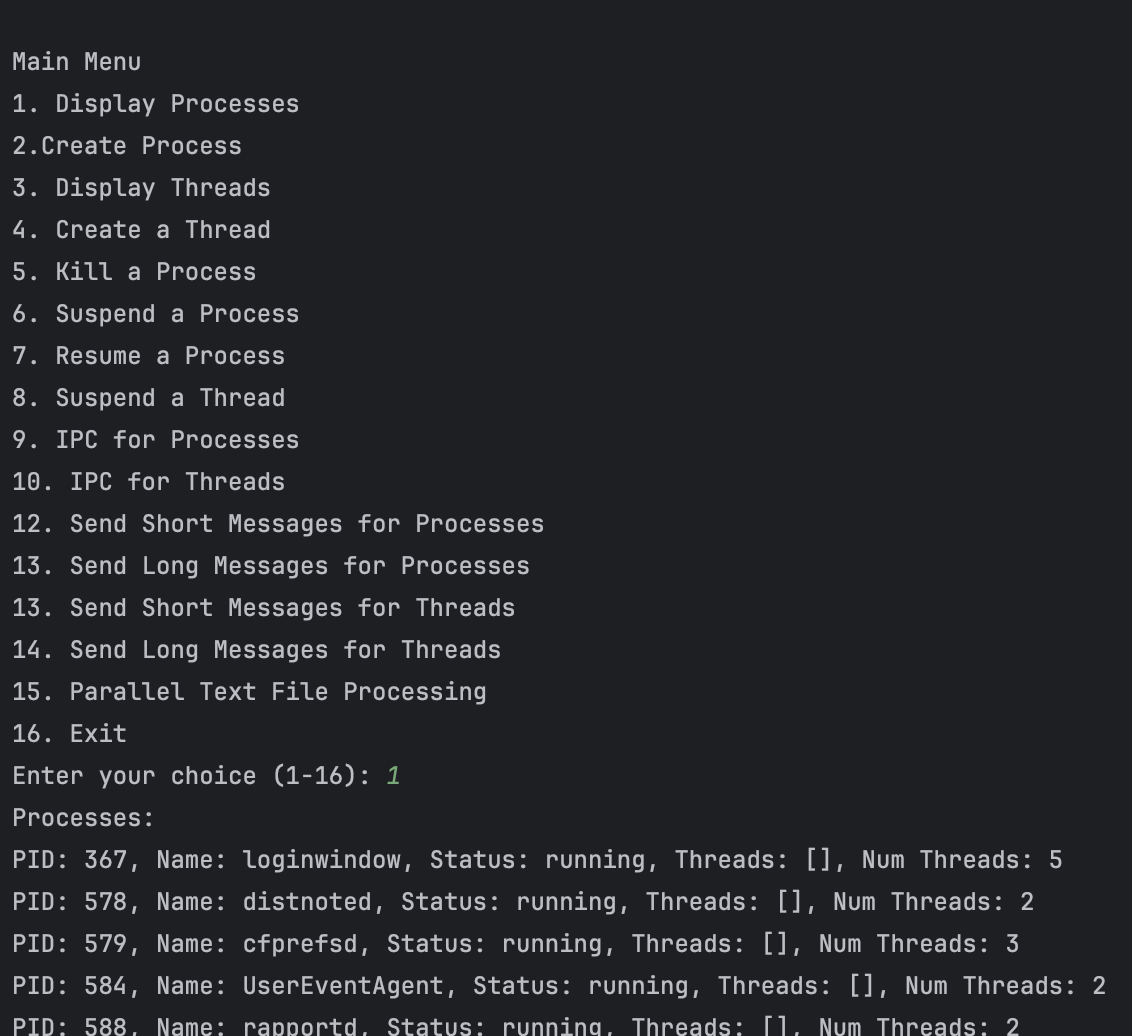
* To display all running processes choose option ‘1’
* To create a process choose option ‘2’
* To display all running threads choose option ‘3’
* To create a thread choose option ‘4’
* To kill a process choose option ‘5’
  + Users will then be prompted to enter the PID of the process you want to kill.
  + If you need to find a PID to specify make sure to choose option ‘1’ so that all running processes are displayed.
* To suspend a process choose option ‘6’
  + Users will then be prompted to enter the PID of the process you want to suspend.
* To resume a process choose option ‘7’
  + Users will then be prompted to enter the PID of the process you want to resume.
* To suspend a thread choose option ‘8’
  + Users will then be prompted to enter the ID of the process you want to suspend.
  + If you need to find an ID to specify make sure to choose option ‘1’ so that all running threads are displayed.
* To simulate IPC for processes choose option ‘9’
* To simulate IPC for threads choose option ‘10’
* To send short messages between processes choose ‘11’
* To send long messages between processes choose ‘12’
* To send short messages between threads choose ‘13’
* To send long messages between threads choose ‘14’
* To perform parallel text file processing choose ‘15’
  + NOTE: if this option is chosen make sure you have a text file available and loaded into your environment. Next you will need to copy that path and attach it to the elif statement with choice == ‘15’’
* To exit choose ‘16’

Discuss your finding through this project, challenges faced during implementation, any limitations or areas for improvement:

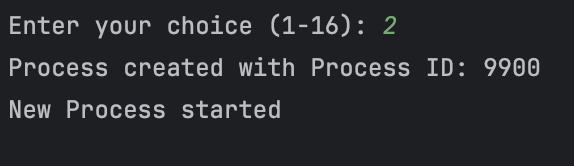
This project allowed me to gain more knowledge with processes and threads. I completed this project in Python in hopes of it being ‘easier’ to understand in contrast to C. Personally completing this project in Python was a lot easier when comparing it to all the homework assignments in C. I was also able to explore into other classes within the multiprocessing module. I need to improve the way my code is organized. I lacked a true structure to it all. I would like to implement some good coding practices/ standards. I was also able to do some error handling which was great practice because those standards are important when working in the industry. There is also a lot of documentation on the internet teaching these concepts making the concepts easier to understand and allow us to play around with different libraries/classes.

Screenshots:

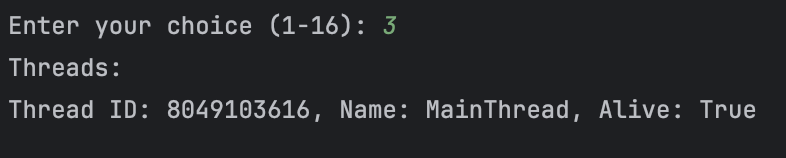
1. Displays Process



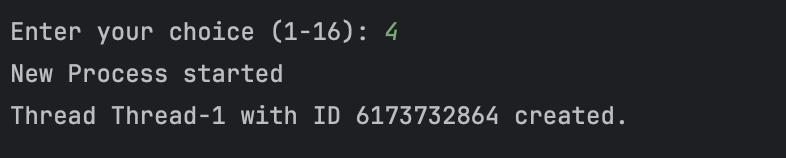
2. Create Process



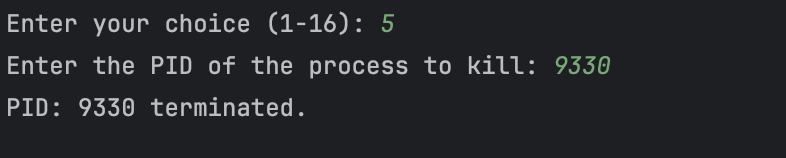
3. Display threads



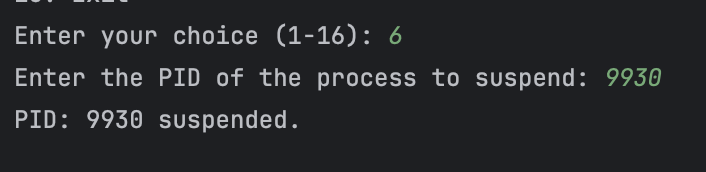
4. Create a thread

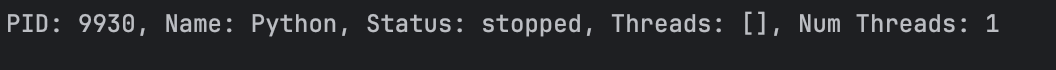


5. Kill a process

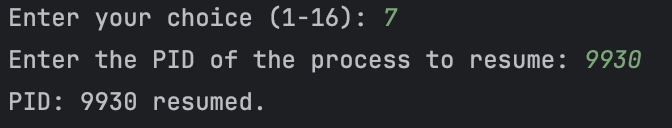


6. Suspend a process

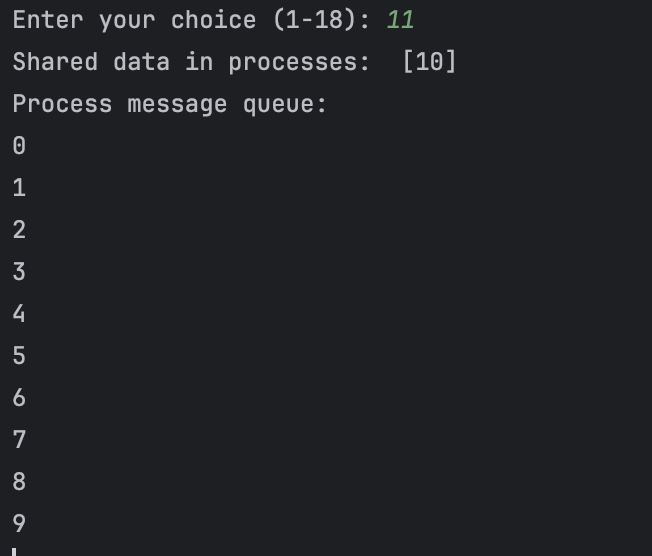




7. Resume a process

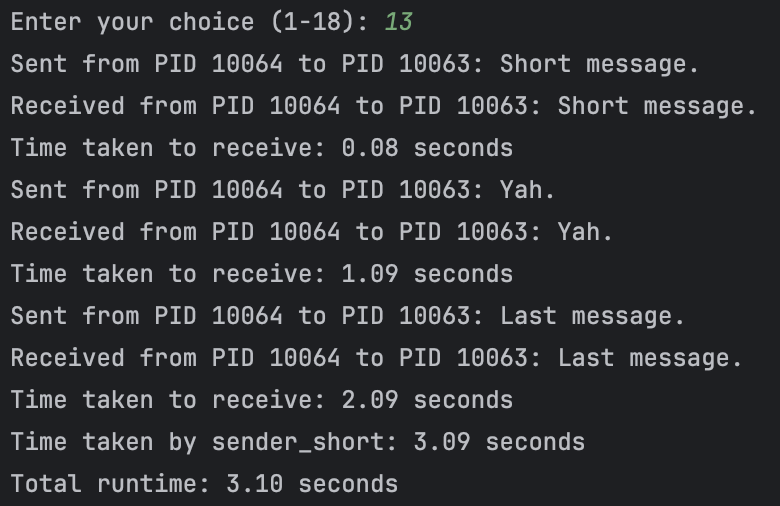


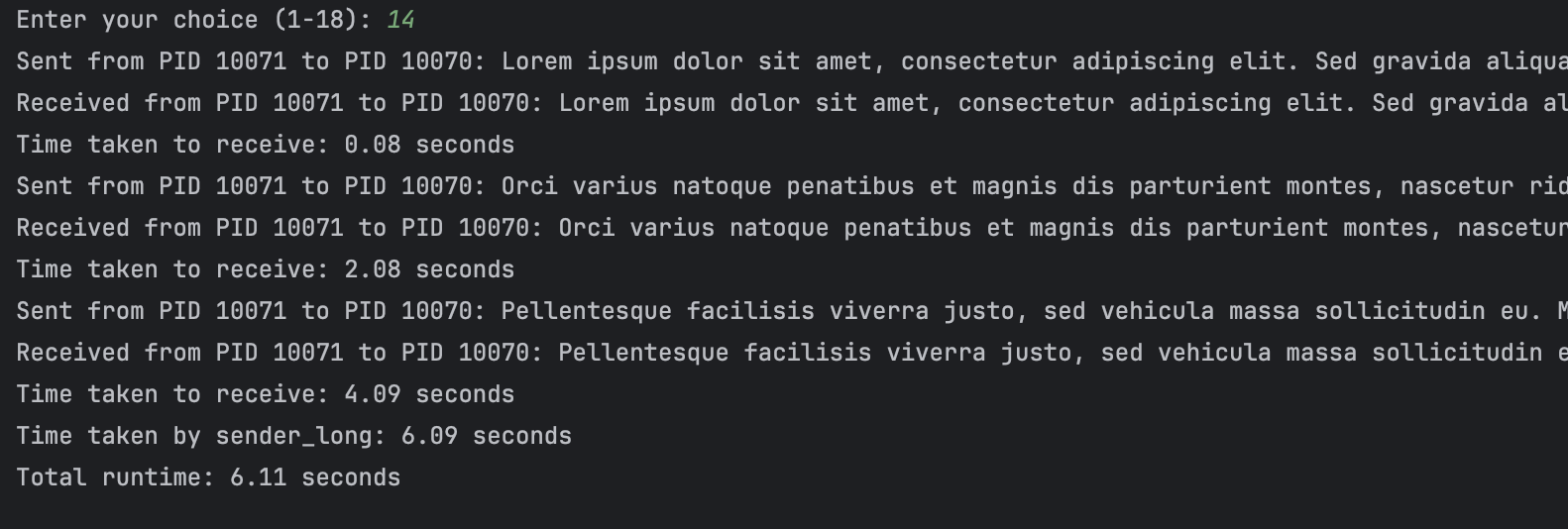
11. IPC for processes



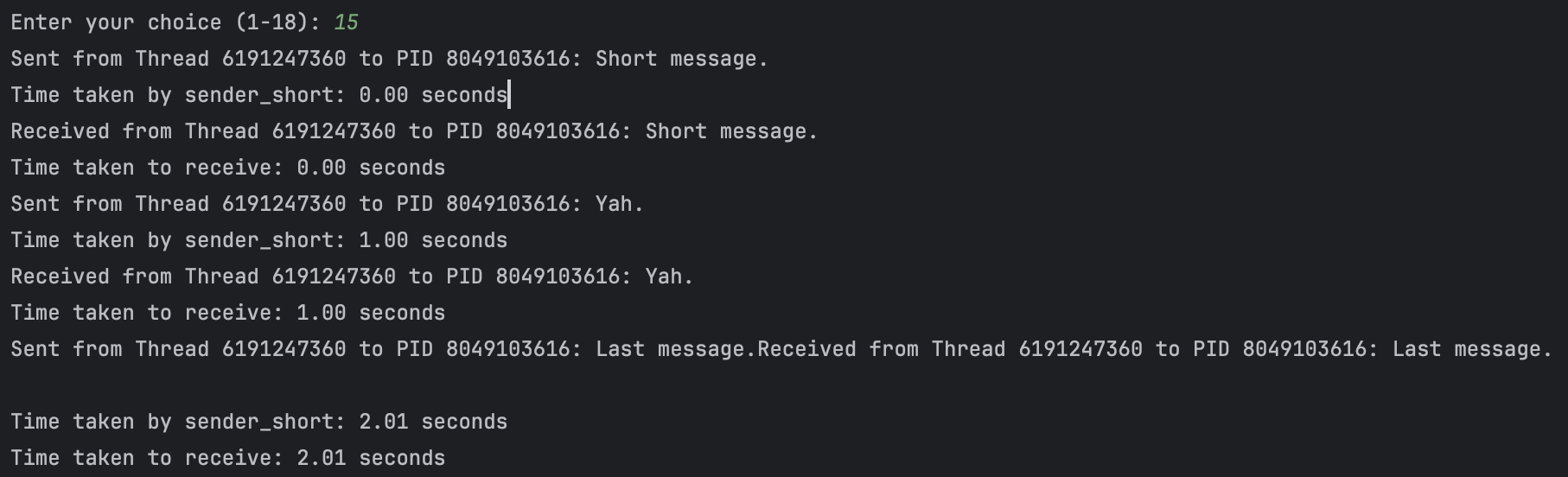
12. IPC for threads



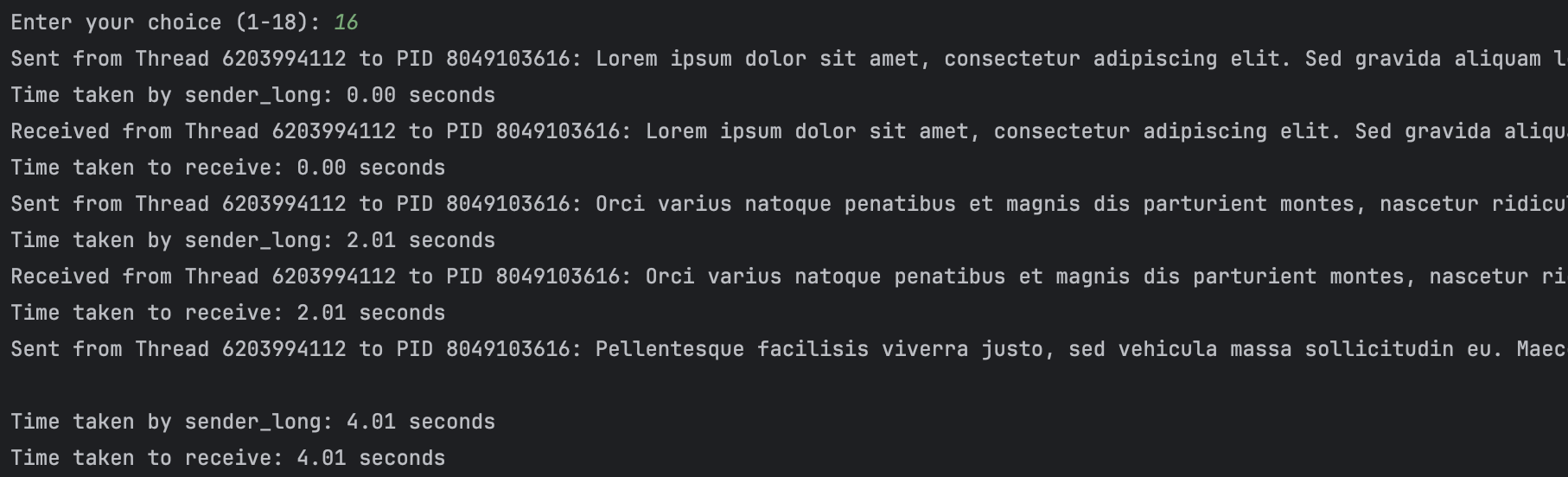
13. Sending short messages for processes

14. Send long messages for processes

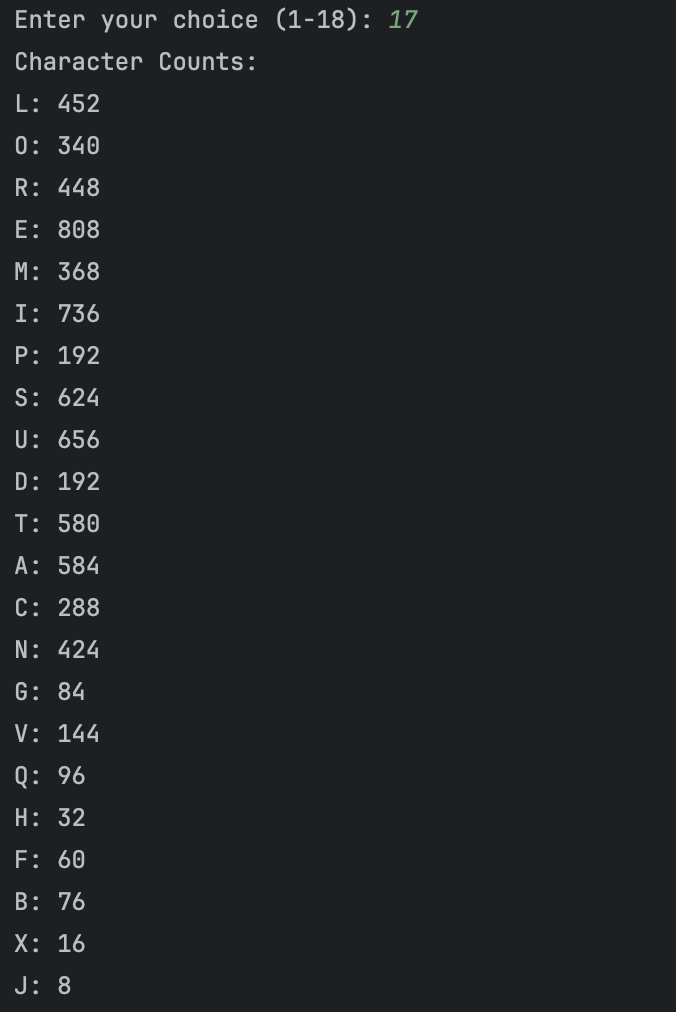
15. Send short messages for threads



16. Send long messages for threads



17. Parallel text file processing



18. Exit

