DATA STRUCTURES AND ALGORITHMS USED

postoffice.hpp

1. struct Mail

This is a custom data type i have created to help organize and store each message in the "postoffice.hpp" header file. It stores,

- Content of the message in the form of a string
- Index of the sending thread in the form of an integer
- Index of the receiving thread in the form of an integer

By default while creating an element of type 'Mail', if no arguments are passed it sets their value as ("",-1,-1)

2. class PostOffice

This class contains the whole implementation of my priority queue.
For my queue I am using the inbuilt priority queue provided by the <queue> library in C++. It uses heap data-structure in its implementation to store data. I have named my queue 'PObox'.
My 'PObox' stores a pair of integer and 'Mail'. 'Mail' stores the message and the integer signifies its priority. Lower the number, higher is its priority. I have also built a custom comparator named 'MailComparator' which is used for comparing messages in my queue based on their priority, and passed it as an argument while declaring my queue.
In this class I have defined the functions enqueue(), dequeue(), peek() & empty_check()

- enqueue(): inserts an element in the queue
- dequeue(): removes and returns the topmost element from the queue.If the queue is empty it returns a default-valued 'Mail' with priority 0.
- peek(): same as dequeue() except it doesn't remove the element.
- empty_check(): checks whether the queue is empty or not.

All my functions are made thread safe with the help of mutex object 'postman' and a lock_guard wrapper 'gate' around it.

whtasapp.hpp

1. class WhatsApp

This class contains the entire implementation of my thread pool.
It also uses the queue from 'PostOffice' class by including the "postoffice.hpp" header file and creating an object 'Telegram' of that class.The constructor takes in two arguments 'MailBag' and 'notifications'. 'MailBag' stores all the messages in an organized manner. MailBag[i] has all the messages that thread[i] needs to send. notifications[i] stores how many messages thread[i] is set to receive. This helps us stop the execution of the thread once it has received all its designated messages to save CPU resources. Once the constructor is called it spawns 8 threads to start executing concurrently.

- transceivers[]: It is a vector of threads which is useful to determining which thread should get which message with the help of indexing.
- _post(): This function is called when a thread wants to send a message. It pushes the message into the queue shared by all threads.
- _log(): This function is called when a thread receieves a message. It simply prints out the details of the received message.
- routine(): This is the thread function which each thread executes on spawning.

Heres a brief overview of the routine() algorithm,

MailsFromMe[]-array of messages that thread must send,
MyNotifs-numbers number of messages meant for that thread,

Each time a thread pushes or pops an element out of the message-queue, it notifies all the other threads as the topmost element may have been updated which could be meant for one of the previously sleeping threads.
Each thread first starts sending whatever messages it must send by iterating the 'MailsFromMe' vector. On every iteration it also checks if the topmost message is meant for that thread. If not, it continues sending its messages. If yes, it pops it out of the queue, logs it using the _log() function, decreases its 'MyNotifs' by one and again continues sending messages. On completion of this loop it enters another loop based on the

condition if ‘MyNotifs’ is 0 or greater. It it’s 0, it means there aren’t anymore messages left for it, so it stops exection. If it’s greater than 0, that means it must wait for more messages. On every iteration of this second loop it first checks if the topmost message in the queue is meant for it. If yes it follows the same instructions as given in the previous loop. If not it waits till its woken up by another thread.
Mutex object ‘Stamp’ and a lock wrapper ‘chain’ is used to make the functions thread safe. Condtion variable ‘_op_’ is used to make the threads wait or to notify them.
Once the destrucor is called it joins all the threads one by one using the join() function.

Main.cpp

Database[]: I have used this array to store all the messages initially in an unorganized way.

struct Envelope:
This is also a custom built data type that stores message information including it’s priority. It stores,

- Content of the message in string format
- Priority of the message in integer format
- Index of the sending thread in the form of an integer
- Index of the receiveing thread in the form of an integer

readInput():
This function is called if the user wishes to input message data and metadata themselves instead of having it hardcoded. Used must input in exactly the following way, including new line and spaces.

```
tnof_mssgs,
stuff
rank mouth ear
stuff
rank mou..
.
.
```

Note: User must follow this exact method of input for proper parsing of input data.

`main()`:
This is where it all comes together. I create a vector of `'Envelope'` to store all the inital message data in an unorganized way. Then after taking input from the user i iterate over all the `'Envelope'`s and organize it in the `'notifications'` vector and the `'MailBag'` vector. After doing so the program creates an object of class `'WhatsApp'` and passes in `'MailBag'` and `'notifications'` vectors as arguments to the constructor. The threads then execute accodingly and finally we receive the ouput as a detailed log of every message, not necessarily in the same order as the input, nor in the same order each time.