ADDITIONAL POINTS

✔ <bits/stdc++.h> isn't always good

In competitve programming I always use this because it automatically includes most of the standard C++ libraries which is useful there. However, in this case it made my code twice as slow. On including it my code took approx 1.2 seconds on average to execute , whereas removing it made it complete execution in 500-600 miliseconds, for the same test case.

✔ Lock inside a lock

Locks are used in threading for synchronization. They make use of the mutex (mutually exclusive) variable. Its a good idea to use them inside your function but one must be careful while nesting locks. I accidently used the same lock (lock with the same mutex) one inside another during the build. I locked the thread function and called another function from inside it which also used the same lock. This resulted in a deadlock condition and my program froze. Basically it went something like this: A thread took the lock, then the same thread was asking for that same lock again. Ideally, it would wait till the lock is freed again but it will never be freed because the one who's asking for it is the one who has it. A similar situation to searching for your phone using the flashlight of *your* phone.

✔ Pass by value and Pass by Reference

By defaut, the thread constructor copies the arguments passed into it, So say if I'm passing the vector 'MailBag' as an argument, every thread will have its own copy of 'MailBag'. This wastes memory, hence, I've used the `ref()` function to pass by reference only the list of those messages to a thread that it must send, instead of all the messages to be sent by all threads aka. 'MailBag'.

✔ Good Coding/Design practices

- **'using namespace std':**
  Usually it's not a good practice to use this as it imports all the standard library names into the global namespace which can cause name clashes. But, in this case it doesn't cause any name clashes as I'm only using standard library names. I have also used it mainly to make my code cleaner and easier to understand.

- **‘thread::hardware_concurrency()’**
  This function returns how many threads your CPU can safely run concurrently. Actually, it returns how many cores your CPU has. Mine has 8 so thats why I have fixed the number of threads to 8. However my CPU, and probably yours too, can handle well above 8 threads for this program. It’s not a very heavy program that uses too many CPU extensive operations. Its just a good practice to use only as many cores as you have or less, so that it reduces CPU switching overhead and all your threads can run at the same time.

- **‘using std::priority_queue’**
  I have used the inbuilt priority queue provided by the standard queue library in C++ instead of implementing it from scratch because it is the most efficient implementation of priority queue available. It makes use of a heap with a custom comparator provided by me which makes insertion and deletion operations in `O(logn),` and accesses the topmost element in `O(1)` time complexity.

- **‘Including all header files’**
  I didn’t have to include all header files in my “`main.cpp`” file as all the required header files were inlcuded indirectly through the inclusion of my “`whatsapp.hpp`” header file. But it’s not a good practice to rely on indirect inclusion of libraries through some other header file.

- **‘notifications[] array’**
  I made use of a notifications array to keep track of how many messages are left for each thread. This way each thread after getting their quota of messages stops executing. Another way was to make use of the empty_check() function but that would keep all the threads executing untill the message queue was completely empty. This would waste CPU resources.

- **‘Modular Programming’**
  I have applied the software development technique of modular programming which involves dividing a program into separate independent modules. I divided the implementation of my multi-threading messaging system into 3 files, one “`main.cpp`” and two header files, and also made the use of functions to send and log messages. This makes my code short, concise and easy to understand while maintaining its efficiency.