L3: Concurrent Ballot Counter

The year is 1865. The United States of America is plunged into chaos following the tragic death of President Abraham Lincoln. The Civil War has come to an end, but tensions between the North and the South persist. The country needs a new leader, someone capable of uniting the fractured nation and restoring peace. To avoid delays and disputes, Congress decides to organize a nationwide vote for the new president as a matter of urgency. However, the challenge before them is enormous: a staggering 256 candidates have entered the election!

Since the voting must be conducted quickly and smoothly to prevent further unrest, it is decided to employ a groundbreaking technology that has emerged alongside the development of industry and telegraphy — modern computing machines operating in multiple parallel data streams. These first-ever multi-core devices boast significant computing power, enabling them to process votes from numerous regions simultaneously. In this way, the results can be gathered and tallied at record speed.

The entire system operates by scanning telegraphic reports from various states, with each report stored in a separate file. In these files, each character represents a vote cast for one of the 256 candidates. Once all the data is collected, the computing machine must quickly and accurately sum up the votes based on these reports.

Your task is to write a program that, using multiple threads, will swiftly and concurrently read files containing votes, count them, and update the results in a central shared memory. Remember, given the country's fragile situation, the program must operate reliably and be resilient to errors. The fate of the United States rests in your hands!

It is prohibited to use global or static variables. For file reading, only low-level APIs (e.g., read, close, readdir, etc.) may be used.

Stages

1. (**8p.**) Complete the implementation of a queue that will be used to distribute file paths among threads. Ensure safe access to the queue by multiple threads simultaneously. To accomplish this, complete the implementation of the methods `queue_*` according to the provided documentation. You may add one new field to the `queue` structure to ensure thread-safe access to the queue.
2. (**6p.**) The program launches between `1 <= t <= 16` threads that count votes. The main thread scans directory `p` for files and places their paths into a queue with a capacity of `1 <= q <= 8`. Worker threads retrieve paths from the queue and print them along with their `id` until no new paths are available, after which the threads terminate.
3. (**4p.**) Threads count votes into a shared array. Each field of this array is protected by a dedicated mutex. Votes are read in batches into a buffer of size 256 bytes, and then counted one by one from the buffer. After counting each vote, threads sleep for `1 ms`. The main thread waits for the worker threads to finish and then prints the final voting result.
4. (**6p.**) The main thread also creates a thread responsible for handling signals. Two signals will be handled in this thread. Handling `SIGUSR1` will involve printing partial voting results. While printing partial results, ensure the voting array is protected against writes during the printout. Receiving `SIGINT` signifies an immediate termination of the voting process. The program must stop counting votes, release all allocated resources, and print the voting results gathered so far.