KOÇ UNIVERSITY

COMP 304: Operating Systems

Project - 2

Spring 2023

Ali Gebeşçe, 64294

Yakup Enes Güven, 64045

Table of Contents

Important Note

Please note that all project components were completed by both team members during in-person and online meetings. However, only one team member's computer was used to commit the corresponding work after these meetings. As a result, Git commits do not accurately reflect individual contributions. Nevertheless, we made an effort to ensure that both team members' Git accounts contributed equally in terms of the weight of points for the project.

A. Part I [1, 2, 3, 4, 5, 6]

1. **STL queue in c++:**
   - We implemented our project in C++ to use classes and more importantly STL queues as shown in below.

```
class VoterQueue { // class to represent a queue of voters, with thread-safe operations
private: // private data members
    std::queue<Voter> queue; // the queue of voters
    pthread_mutex_t lock{}; // mutex lock for thread-safe operations

    std::string queue_to_string(std::queue<Voter> q) { // helper function to convert a queue to a string
        std::string str_rep; // the string representation of the queue
        while (!q.empty()) { // while the queue is not empty
            str_rep += std::to_string( val: q.front().id) + " "; // add the front element to the string
            q.pop(); // pop the front element
        }
        return str_rep; // return the string representation
    }

public:
    void push(const Voter &voter) { // push a voter to the queue with thread-safe operations
        pthread_mutex_lock(&lock); // lock the mutex
        queue.push( v: voter); // push the voter to the queue
        pthread_mutex_unlock(&lock); // unlock the mutex
    }

    Voter &front() {
        pthread_mutex_lock(&lock);
        Voter &return_val = queue.front();
        pthread_mutex_unlock(&lock);
        return return_val;
    }

    Voter front_and_pop() { // get the front element of the queue and pop it with thread-safe operations, voter is voted
        pthread_mutex_lock(&lock); // lock the mutex
        Voter return_val = queue.front(); // get the front element
        queue.pop(); // pop the front element
        pthread_mutex_unlock(&lock); // unlock the mutex
        return return_val; // return the front element
    }


    void pop() { // pop the front element of the queue with thread-safe operations
        pthread_mutex_lock(&lock); // lock the mutex
        queue.pop(); // pop the front element
        pthread_mutex_unlock(&lock); // unlock the mutex
    }

    bool empty() { // check if the queue is empty with thread-safe operations
        pthread_mutex_lock(&lock); // lock the mutex
        bool return_val = queue.empty(); // check if the queue is empty
        pthread_mutex_unlock(&lock); // unlock the mutex
        return return_val; // return the result
    }

    int size() { // get the size of the queue with thread-safe operations
        pthread_mutex_lock(&lock); // lock the mutex
        int return_val = queue.size(); // get the size of the queue
        pthread_mutex_unlock(&lock); // unlock the mutex
        return return_val; // return the result
    }

    std::string to_string() { // get the string representation of the queue with thread-safe operations
        pthread_mutex_lock(&lock);  // lock the mutex
        std::string str_rep = queue_to_string( q: queue); // get the string representation of the queue
        pthread_mutex_unlock(&lock); // unlock the mutex
        return str_rep; // return the result
    }

    VoterQueue() {  // constructor
        lock = PTHREAD_MUTEX_INITIALIZER; // initialize the mutex lock
    }
```

2. main

- Our main method is shown below which summarizes the general workflow of our implementation. We commented on nearly every line of the code so one can grasp the code logic easily.

```
int main(int argc, char *argv[]) { // the main function
    int total_simulation_time; // the total simulation time
    double ordinary_voter_prob; // the probability of an ordinary voter
    int random_seed; // the random seed for the random number generator
    double failure_prob; // the probability of a failure in the polling station
    parse_command_line_arguments(argc, argv, &total_simulation_time, &ordinary_voter_prob, &random_seed, &failure_prob); // parse the command line arguments
    struct timeval tp{}; // struct to get the current time
    gettimeofday(&tp, nullptr); // get the current time
    simulation_start_time = tp.tv_sec; // set the simulation start time
    simulation_end_time = tp.tv_sec + total_simulation_time; // set the simulation end time
    time_t next_failure_time = tp.tv_sec + 10; // set the time of the next failure
    voter_log_file = fopen( filename: "./voters.log",  mode: "w+"); // open the voter log file for writing
    queue_log_file = fopen( filename: "./queues.log",  mode: "w+"); // open the queue log file for writing
    fprintf(voter_log_file, "VoterID\tCategory\tRequest Time\tPolling Station Time\tTurnaround Time\n"); // write the header of the voter log file
    fprintf(voter_log_file, "-------------------------------------------------------------------------------------------\n"); // write the header of the voter log file, continued
    fprintf(voter_log_file, "V\tC\tRT\tPT\tTT\n"); // abbreviations for the header of the voter log file
    srand(random_seed); // set the random seed for the random number generator
    pthread_mutex_lock(&first_voter_mutex); // lock the mutex for the first voter
    // Initial voters one of each type is created
    createVoter( voterType: SPECIAL,  currentTime: tp.tv_sec); // create a special voter as specified in the project description
    createVoter( voterType: ORDINARY,  currentTime: tp.tv_sec); // create an ordinary voter as specified in the project description
    pthread_t queueing_machine_main_thread; // the main thread of the queuing machine
    pthread_create(&queueing_machine_main_thread, nullptr, queuing_machine, nullptr); // create the queuing machine thread

    while (tp.tv_sec < simulation_end_time) { // while the simulation is not over
        if (tp.tv_sec >= simulation_start_time) { // if the simulation has started
            fprintf(queue_log_file, "At %ld sec polling station 1, special: %s\n", // write the queue log file
                    tp.tv_sec - simulation_start_time,
                    special_queue.to_string().c_str());
            fprintf(queue_log_file, "At %ld sec polling station 1, ordinary: %s\n",
                    tp.tv_sec - simulation_start_time,
                    ordinary_queue.to_string().c_str());

            fprintf(queue_log_file, "At %ld sec total votes: Mary: %d, John: %d, Anna: %d\n",
                    tp.tv_sec - simulation_start_time,
                    mary_votes, john_votes, anna_votes);

            printf("At %ld sec polling station 1, special: %s\n", // print the queue log file
                   tp.tv_sec - simulation_start_time,
                   special_queue.to_string().c_str());
            printf("At %ld sec polling station 1, ordinary: %s\n",
                   tp.tv_sec - simulation_start_time,
                   ordinary_queue.to_string().c_str());
            printf("At %ld sec total votes: Mary: %d, John: %d, Anna: %d\n",
                   tp.tv_sec - simulation_start_time,
                   mary_votes, john_votes, anna_votes);
        }
        pthread_sleep( seconds: 1); // sleep for 1 second
        gettimeofday(&tp, nullptr); // get the current time


        if (tp.tv_sec >= next_failure_time && random_double() <= failure_prob) { // if the polling station fails
            createVoter( voterType: MECHANIC,  currentTime: tp.tv_sec); // create a mechanic
            next_failure_time += 10; // set the time of the next failure
        }else{
            double random = random_double(); // get a random double between 0 and 1
            if (random < ordinary_voter_prob) { // if the random double is less than the probability of an ordinary voter
                createVoter( voterType: ORDINARY,  currentTime: tp.tv_sec); // create an ordinary voter
            } else  {
                createVoter( voterType: SPECIAL,  currentTime: tp.tv_sec); // create a special voter
            }
        }
    }

    fclose(voter_log_file); // close the voter log file
    fclose(queue_log_file); // close the queue log file
    pthread_mutex_unlock(&first_voter_mutex); // unlock the mutex for the first voter
```

3. **parse_command_line_argument**
    - We implemented **parse_command_line_arguments** to get the parameters of the simulation. Note that the default parameters are
        - int simulation_time = 60; // the default simulation time
        - double ordinary_voter_probability = 0.5; // the default probability of an ordinary voter
        - int seed = 42; // the default random seed
        - double failure_probability = 1; // the default probability of a failure at every 10 sec

```
void parse_command_line_arguments(int argc, char **argv, int *total_simulation_time, double *ordinary_voter_prob,
                                  int *random_seed, double *failure_prob) { // parse the command line arguments
    int simulation_time = 60; // the default simulation time
    double ordinary_voter_probability = 0.5; // the default probability of an ordinary voter
    int seed = 42; // the default random seed
    double failure_probability = 1; // the default probability of a failure

    for (int i = 1; i < argc - 1; i++) { // for each command line argument
        if (strcmp("-t", argv[i]) == 0) { // if the argument is -t
            int time_arg = atoi(argv[i + 1]);
            if (time_arg > 0) {
                simulation_time = time_arg; // set the simulation time to the argument
            } else {
                std::cout << "You entered an illegal value for simulation time, " << simulation_time
                          << " will be used as default" << std::endl;
            }
        } else if (strcmp("-p", argv[i]) == 0) { // if the argument is -p
            double prob_arg = atof(argv[i + 1]);
            // user entered 0 as probability
            if (strcmp(argv[i + 1], "0") == 0) {
                ordinary_voter_probability = 0;
            } else if (prob_arg <= 0.0 || prob_arg > 1.0) {
                std::cout << "You entered an illegal value for the probability of ordinary voter, " << ordinary_voter_probability
                          << " will be used as default" << std::endl;
            } else {
                ordinary_voter_probability = prob_arg;
            }
        } else if (strcmp("-s", argv[i]) == 0) {
            try {
                unsigned long seed_arg_long = std::stoul( str: argv[i + 1],  idx: nullptr,  base: 10);
                auto seed_arg = (int) seed_arg_long;
                if (seed_arg_long != seed_arg) throw std::out_of_range( s: "");
                seed = seed_arg;
            }
            catch (...) {
                std::cout << "You entered an illegal value for seed, " << seed
                          << " will be used as default" << std::endl;
            }
        } else if (strcmp("-f", argv[i]) == 0) { // failure prob
            double failure_prob_arg = atof(argv[i + 1]);
            if (failure_prob_arg < 0.0 || failure_prob_arg > 1.0) {
                std::cout << "You entered an illegal value for the probability of failure, " << failure_prob
                          << " will be used as default" << std::endl;
            } else {
                failure_probability = failure_prob_arg;
            }

        }
    }

    *total_simulation_time = simulation_time; // set the simulation time to the argument
    *ordinary_voter_prob = ordinary_voter_probability; // set the probability of an ordinary voter to the argument
    *random_seed = seed; // set the random seed to the argument
    *failure_prob = failure_probability; // set the probability of a failure to the argument
}
```

B. Part II [7, 8]

- The proposed solution for the Part 2 which suggests the priority of special voters unless
  - No more elderly or pregnant women are waiting,
  - 5 or more non-elderly non-pregnant voters are lined up to vote.
- Creates a starvation for special voters since there is a chance of creation of the ordinary voters so they keep coming to the polling station. As a result ordinary queue size becomes greater than 5 which causes starvation of the special voters.
- To prevent that at each iteration we created a random number between 0 and 1 and if this value is greater than **0.2** then special voters vote. Note that this solution gives a chance to special voters to be executed, a chance of **80%** and **20%** for the ordinary voters. The part that implements this behavior is below:

```
void *queuing_machine(void *) { // the queuing machine thread
    pthread_mutex_lock(&first_voter_mutex); // lock the first voter mutex
    struct timeval tp{}; // the timeval struct
    gettimeofday(&tp, nullptr); // get the current time
    while (tp.tv_sec < simulation_end_time) { // while the simulation is not over
        double random_who_to_serve = random_double(); // generate a random double between 0 and 1
        if (!mechanic_queue.empty()) { // if the mechanic queue is not empty
            Voter voter = mechanic_queue.front_and_pop(); // get the first voter in the mechanic queue
            current_voter_id = voter.id; // set the current voter id to the voter id
            pthread_cond_broadcast(&iteration_cond); // broadcast the iteration condition
            pthread_sleep( seconds: 5); // sleep for 5 seconds
        } else if (random_who_to_serve < probability_to_serve_ord && !ordinary_queue.empty()) { // if the random double
            // is less than the probability to serve an ordinary voter and the ordinary queue is not empty
            double random_vote = random_double(); // generate a random double between 0 and 1
            if (random_vote < 0.4) {  // if the random double is less than 0.4 (40% chance)
                mary_votes++; // increment mary's votes
            } else if (random_vote < 0.55) { // if the random double is less than 0.55 (15% chance)
                john_votes++; // increment john's votes
            } else { // if the random double is greater than 0.55 (45% chance)
                anna_votes++; // increment anna's votes
            }
            Voter voter = ordinary_queue.front_and_pop(); // get the first voter in the ordinary queue
            current_voter_id = voter.id; // set the current voter id to the voter id
            pthread_cond_broadcast(&iteration_cond); // broadcast the iteration condition for the voter threads
            fprintf(voter_log_file, "%d\t%c\t%ld\t%ld\t%ld\n", // print the voter's information to the voter log file
                    voter.id, // the voter id
                    'O', // the voter type
                    voter.request_time - simulation_start_time, // the time the voter requested to vote
                    tp.tv_sec - simulation_start_time, // the time the voter started voting (polling station time)
                    tp.tv_sec - voter.request_time); // the time the voter waited in the queue (turn around time)
            pthread_sleep( seconds: 2); // sleep for 2 seconds

        } else if (!special_queue.empty()) { // if the special queue is not empty and the random double
            // is greater than the probability to serve an ordinary voter
            double random_vote = random_double(); // generate a random double between 0 and 1
            if (random_vote < 0.4) { // if the random double is less than 0.4 (40% chance)
                mary_votes++;
            } else if (random_vote < 0.55) { // if the random double is less than 0.55 (15% chance)
                john_votes++;
            } else { // if the random double is greater than 0.55 (45% chance)
                anna_votes++;
            }
            Voter voter = special_queue.front_and_pop(); // same as above
            current_voter_id = voter.id;
            pthread_cond_broadcast(&iteration_cond);
            fprintf(voter_log_file, "%d\t%c\t%ld\t%ld\t%ld\n",
                    voter.id,
                    'S',
                    voter.request_time - simulation_start_time,
                    tp.tv_sec - simulation_start_time,
                    tp.tv_sec - voter.request_time);
            pthread_sleep( seconds: 2);
        }
        gettimeofday(&tp, nullptr);
    }
    pthread_exit(0);
}
```

C. Part III

Failure of Polling Station and Mechanic Voter Creation

- Note that although **MECHANIC** is not actually a voter we treat as if it was since implementation is much easier. For example as you can see below we can create a mechanic as if he was the problem of the failure. Since for this context causality is not important we can safely do that.

```
    if (tp.tv_sec >= next_failure_time && random_double() <= failure_prob) { // if the polling station fails
        createVoter( voterType: MECHANIC, currentTime: tp.tv_sec); // create a mechanic
        next_failure_time += 10; // set the time of the next failure
    }else{
        double random = random_double(); // get a random double between 0 and 1
        if (random < ordinary_voter_prob) { // if the random double is less than the probability of an ordinary voter
            createVoter( voterType: ORDINARY, currentTime: tp.tv_sec); // create an ordinary voter
        } else  {
            createVoter( voterType: SPECIAL, currentTime: tp.tv_sec); // create a special voter
        }
    }
}
```

```
void createVoter(VoterType voterType, time_t currentTime) { // create a voter
    int voter_id = generate_voter_id(voterType); // generate a voter id
    Voter new_voter = Voter( id: voter_id, request_time: currentTime); // create a new voter
    pthread_t new_voter_thread; // the thread of the new voter
    int *voter_id_pointer = (int *) malloc( size: sizeof(int)); // allocate memory for the voter id pointer and set it to the voter id
    *voter_id_pointer = voter_id; // set the voter id pointer to the voter id
    switch (voterType) { // switch on the voter type
        case ORDINARY: // if the voter is ordinary
            ordinary_queue.push( voter: new_voter); // push the voter to the ordinary queue
            pthread_create(&new_voter_thread, nullptr, ordinary_voter_main, (void *) voter_id_pointer); // create the ordinary voter thread
            break;
        case SPECIAL: // if the voter is special
            special_queue.push( voter: new_voter); // push the voter to the special queue
            pthread_create(&new_voter_thread, nullptr, special_voter_main, (void *) voter_id_pointer); // create the special voter thread
            break;
        case MECHANIC: // if the voter is mechanic
            mechanic_queue.push( voter: new_voter); // push the voter to the mechanic queue
            pthread_create(&new_voter_thread, nullptr, mechanic_main, (void *) voter_id_pointer); // create the mechanic thread
    }
}
```

D. References

1. https://hpc-tutorials.llnl.gov/posix/
2. https://www.geeksforgeeks.org/queue-cpp-stl/
3. https://cplusplus.com/reference/queue/queue/
4. https://learn.microsoft.com/en-us/cpp/c-language/parsing-c-command-line-arguments?view=msvc-170
5. https://www.tutorialspoint.com/cplusplus/cpp_date_time.htm
6. https://www.bogotobogo.com/cplusplus/multithreaded4_cplusplus11B.php
7. https://www.javatpoint.com/what-is-starvation-in-operating-system
8. https://www.geeksforgeeks.org/deadlock-starvation-and-livelock/