Operating Systems Lab (CS 470):

Lab 3: Write a program in C/C++ under Linux to simulate a colonization type game. There are two teams: A team and B team and they play against each other. The goal is to conquer a rectangular map from the opposite team.

The rules of the game are:

1) Originally a certain number (T1 and T2) of team members from both teams should be positioned on the rectangular map (size MxN) using random coordinates (i,j), where 0<=i<M and 0<=j<N. The rest of the map is considered unoccupied territory. Make sure each member is position originally on the map!
2) A team member cannot originally occupy the location (i,j) on the map of some other player from the same team or opponent who already occupied that spot.
3) Each team member is allowed to send a missile to a random location (k,l) on the map. If location (k,l) is not occupied yet or the opposite team conquered that location, the location (k,l) will be occupied by the team member who fired the missile. If the location was occupied by the same team the location is going to be released by the team. After one missile is fired and handled, each player can fire a new missile and repeat the operation as many times possible till the end of the game. An infinite number of missiles are available for each player. Between two consecutive missile firing, introduce a random 1, 2 or 3 seconds wait period simulating the preparation time for each new missile strike.
4) If the majority around the location (k,l) including the (k,l) once it has changed ownership (see rule 3)) is occupied by the team who sent the missile to that location, all locations in the 4 or 8 vicinity (depends on your implementation) will be conquered by the team who sent the missile.
5) In any circumstances, none of the original T1+T2 locations (see rule 1)) can be destroyed or conquered using rule 3) and 4).
6) The game simulation continues until all the locations are conquered by a team or another. The winner is who has occupied all spots or there is a draw, -meaning that each team owes the same number of locations. No spot on the map should stay unoccupied.

Overview

Reading and modifying (increment, decrement, assign) a shared variable/file between different threads needs extra attention due to inconsistency which might occur if the processes run in parallel. In order to avoid this, different mechanism are implemented to solve the critical section problem.

Instructions

The number of participant in each team T1 and T2 should be provided as command line arguments. Same procedure for the size of the map (see M and N). The map should be stored as a binary file, -as a sequence of bytes. One possible way would be to represent the first row,

followed by the second row, etc. Each byte represents one location in the map. After each update (see sending a missile and checking the vicinity) the ownership (belonging to team A, belonging to team B or unoccupied) of the locations should be written in the binary file. No long term storage in the memory is allowed for the map. Each team member should update the map from the file before ending a new missile.
To lock the files consider the *fcntl()* function. To protect the critical section consider *pthread_mutex_lock()* and *pthread_mutex_unlock()* functions, respectively. However, you are not allowed to lock the complete file as you do not need to do such thing. Each time a missile is fired only the location (k,l) and its vicinity should be locked.

Notes

- Each team member should be implemented as a separate thread.
- Each step in the simulation process should be traceable. Printing on the standard output is to be provided.
- Implement a “supervisor” thread who is going to signal all the players that the game is over. This thread is responsible to check if the game reached an end.
- Your executable should work something like this: ***./Lab3 2 3 10 20***, which means team A will have two (T1) players, team B will have 3 (T2) players and the map has 10 rows (M) and 20 columns (N). The Lab3 will be the name of the executable file.

Rubric

Task	**Points**
Error handling	2
Team member thread	6
Supervisor thread	2