OS Project

Carbird fury

(A cool survival game using ncurses and pthreads library in C on linux)

Project by:

Animesh Kumar	CED18I065
T Karthikeyan	CED18I064
Chogale Rudra Jairaj	CED18I012
Akhil.M	CED18I032
Vamsi.B	CED18I011

Overall Setup:

Introduction:

- We have used the ncurses library in C to implement this survival game. The ncurses library is used to place the car and obstacles at a given coordinate precisely.
- The player has to stay up on the game not being hit by any of the obstacles for as long as possible. The demo video illustrates how to play the game.
- We have used detached threads, to implement the game_over logic easily. For threads creation and handling we used our basic POSIX threads setup in linux using C.

Macros:

- SIZE - #obstacles visible on the screen at a moment
- T - microseconds to sleep for fence movement logic

Global Variables:

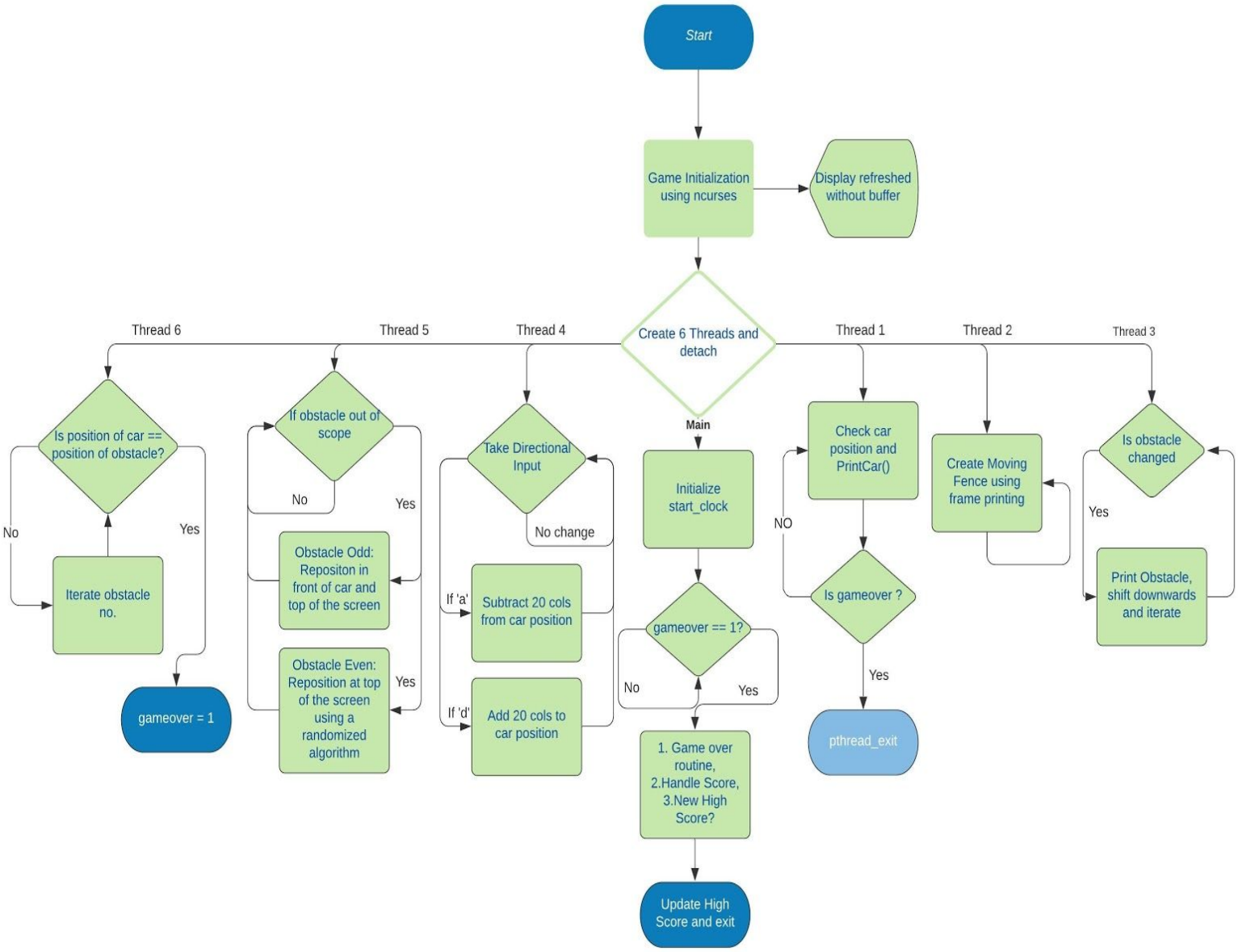
- (row_car, col_car) - coordinates of the car, which can be modified by any of the threads.
- is_car_changed - represents whether the location of the car is changed.
- game_over - represents whether the game is over or not.
- SIZE - the number of obstacles
- row_obs[SIZE] - row coordinates of the obstacle
- col_obs[SIZE] - column coordinates of the obstacle
- is_obs_changed[SIZE] - status of the obstacles.

Below structure is used to pass data among threads -

```
struct block {
    WINDOW * win;    // window pointer
    int obs_num;     // obstacle number (0/1/2)
    int i_ptr;
    int j_ptr;
};
```

Flowchart Logic diagram:

There will be six threads in total. Below flowchart shows the control flow of all the threads and their execution summary exhaustively.



Now we'll explain the algorithm(and the code segments) segment by segment in a mixed format of pseudo code and english language.

Main Function:

- Creates a new window for the game to be run in a box format of a given size.
- Record the start time.
- Make the screen blank
- Create 6 detached threads and pass the window pointer to all the threads as a parameter
- wait for game over, when game is over record the end time and calculate the score
- print the GAME OVER notification in a styled format.
- Call the HandleHighScore() utility
- end window
- terminate the main thread using 'return', which eventually terminates all the 6 new threads created if they are still running.

Segment: 1 : Printing the Car logic

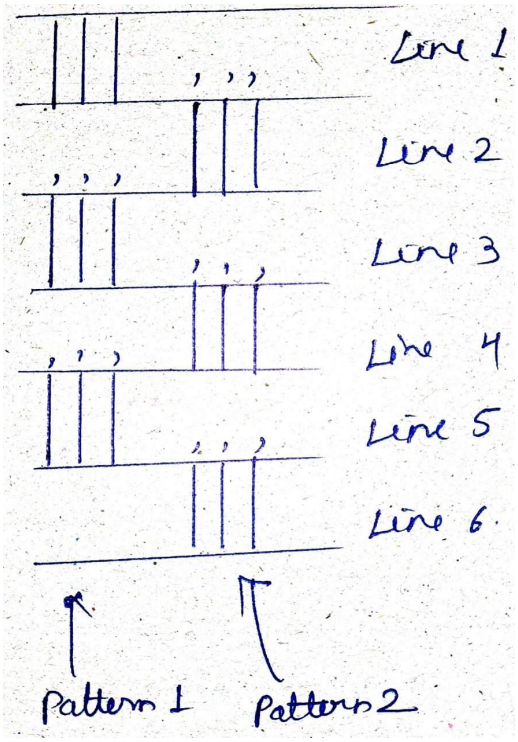
- runner1:
    - while(1)
        - if the location of car is changed then,
            - MakeBlank()
            - PrintCar()
            - is_car_changed = 0
- MakeBlank:
    - Makes the entire game area blank excluding the locations where any of the two obstacles are present.
- PrintCar:
    - if(game over) => terminate the thread in execution
    - else print the car at the current location on the screen.

Segment: 2 : Moving the Car logic

- runner4:
    - while(1)
        - if(game over) terminate the thread in execution
        - else listen to the keyboard entry by the user
            - if the entry is ‘a’: decrement the col_car by 20, and is_car_changed = 1. // this moves the car to the left
            - else if the entry is ‘d’: increment the col_car by 20, and is_car_changed = 1. // this moves the car to the right

Segment: 3 : Fence movement logic

In this segment we are essentially creating an illusion of moving fences using two patterns blinking one after the other in a synchronized manner. The two patterns are:



- runner2:
    - blink_fence(left fence, pattern1)
    - blink_fence(right fence, pattern1)
    - sleep for some time
    - blink_fence(left fence, pattern2)
    - blink_fence(right fence, pattern2)
    - sleep for some time
- blink_fence:
    - based upon the argument, print pattern1 or pattern2.

Segment: 4 : Obstacle handling logic

- runner3:
    - while(1)
        - for every obstacle in the screen
            - if its location is changed
                - print blank spaces at the top row of previous location
                - PrintObstacle(this obstacle)
                - increase the row_obs coordinate of this obstacle to give a feeling of motion in next printing.
- runner5:

  Its a randomised algorithm which creates obstacles based upon a random number generated.
    - Print the first obstacle at the track of the car itself
    - Print the second obstacle at any of the tracks other than the car's track using a random number generator.

- PrintObstacle:
    - if(game over) => terminate the thread in execution
    - else print the obstacle at the coordinates supplied via arguments.

Segment: 5 : Hit obstacle logic

- runner6:
    - for all obstacles on the screen
        - if the obstacle is in the same track/column of the car and its row coordinate is greater than that of the car, then this is the case of collision. So, set the game_over to 1, which causes the game to end.

Segment: 6 : Highscore handler

In this segment, to handle the high score we have created a file ‘highscore_db.txt’ to store the highscore. The score of the player is calculated based upon the time he/she passes uncrashed with the obstacles.

- HandleHighscore:
    - open the file ‘highscore_db.txt’ in ‘r+’ mode.
    - convert the current score in the string format.
    - read the present highscore from the file content and convert it into the integer format.
    - if current score is larger than the existing highscore, then replace the file content with new highscore.
    - Announce the highscore breaking on the window of the game.
    - close the file pointer.
- get_string(char * dest, long long src, int *dest_len):
    - converts src to corresponding ascii string and stores it into dest.

- Store the string length into dest_len.

Thanks