Sasha Babicki Comput 379 Assignment 3 1264274

Threads and Program Design

FUNCTIONS

int main();

void lock\_draw();

void unlock\_draw();

void setup\_saucer();

void stats();

int launch\_site();

int welcome();

void saucer\_hit();

void new\_saucer\_position();

void \*saucers();

int rand\_saucers();

void \*replace\_thread();

void \*process\_input();

void \*shots();

void \*find\_end();

int fire\_shot();

void find\_hit();

THREADS

main();

replace\_thread();

process\_input();

saucers();

shots();

find\_end();

GENERAL PROGRAM FLOW AND THREADS DESCRIPTION

The **main** function of my saucer program only does work at the very beginning and very end of the program, most of the work while the program is running is done by other threads. The main function starts with doing some basic housekeeping; including doing some error checking, setting up curses, and allocating memory for the collision detection array, and printing a welcome message by calling the **welcome** function. 2 threads are created in the main function:

1) Thread for replacing terminated threads. Used for dealing with replacing saucer threads that have finished executing. This thread runs primarily in the **replace\_thread** function. Used for replacing old saucers and creating new ones at random. By replacing the threads in the thread array instead of creating new ones we don’t have to allocate memory dynamically and we use less space.

2) Thread for processing user input. Used for dealing with managing all user input and responding to it appropriately. This thread runs primarily in the **process\_input** function.

The two threads mentioned above run constantly until the end of the program is signaled. After the threads are created the main function waits for the *end\_condition* to be signaled. More information about this process and all other mutex and condition variable information is outlined in the “Global Variable Uses and Critical Sections” part of this document.

The replace\_thread thread waits for the *replace\_condition* to be signaled before it does anything. Once the signal is received it creates one new saucer and gives it the index saved in the *replace\_index* global variable. The index is used for the *saucer\_t* array that stores the thread id and the *saucerinfo* array which stores information about that saucer. **Setup\_sacuer** is used to populate the saucerinfo array at that index with information about that saucer. The replace\_thread thread then creates one new saucer thread which runs the **saucer** function and goes back to waiting for a *replace\_condition* signal.

The process\_input thread starts by printing out the score information at the bottom of the screen by calling the **stats** function. Then it prints the original launch site in the middle of the screen by calling the **launch\_site** function. **Setup\_saucer** is called *NUMSAUCER* times to create the *NUMSAUCER* initial saucer threads and then the threads are created to run in the **saucer** function.

3) Thread for printing/updating saucers. Runs in the saucers function. The saucer position is drawn over and the position is updated using **new\_saucer\_position** and drawn on the screen. The position in the *collision\_position* array is updated and hits are detected if the saucerinfo->kill int is set to 1. **Saucer\_hit** deals with covering the hit saucer and removing the saucer information after it had been hit. Saucer\_hit also sets the *replace\_condition* along with the *replace\_index* global variable to replace the saucer using the replace\_thread function. The saucer thread exits once the saucer either leaves the screen (in which case the *escape\_update* global variable is incremented and printed to the screen using stats) or if the saucer is hit.

After the saucer threads are created the process\_input while loop is entered. At the top of the while loop there is code that creates a new saucer at random. Because it is at the top of the while loop it gets executed whenever the user presses any key, meaning that the more key presses there are the more likely it will be that a random saucer will appear on the screen, up until a limit of *MAXSAUCERS* are on the screen. The function then waits for user input and responds to certain key presses. If the ‘,’ or ‘.’ Keys are pressed, the **launch\_site** function is called to update the location of the Launchpad on the screen and the new Launchpad location is saved in the process\_input function. If space pressed the **fire\_shot** function is called, unless there are no shots left in which case the user input is polled again. fire\_shot creates a new shot thread which uses the **shots** function (see 4) The number of available shots (*shot\_update*) is decreased by one and the new score information is printed by calling **stats**. Fire\_shot then updates the *save* global variable with the latest shot index. If the *shot\_update* is 0 after decrementing it a new thread is created that runs the **find\_end** function (see 5). The end of the main function that is returned to when the *end\_condition* is set cancels all the threads, frees memory, prints an exit message, and closes curses.

4) Thread for printing/updating shots. Runs in the shots function. The old shot position is drawn over and the updated position is drawn on the screen. The position in the *collision\_position* array is updated and hits are detected using **find\_hit**. Find\_hit checks for saucers at the same location as the shot in the *collision\_position* array and updates the score (using the *shot\_update* and *score\_update* global variables and a call to stats) and drawing of the shot on the screen accordingly. Additionally the kill signal (int kill in the saucerinfo array) is updated for all hit saucers. The shots thread exits if a hit is detected or if the shot leaves the screen.

5) Thread for checking if all the shots have been used. Used for waiting until the final shot is fired and checking if the program can end. find\_end checks to see if the final shot hits something (in which case the program can continue) or misses everything (in which case there are no more threads and the program can exit). If the final shot misses find\_end uses the *end\_condition* signal to let the main function know that it can take the steps to exit the program.

Global Variable Uses and Critical Sections

GENERAL OVERVIEW OF MUTEXES

**draw**

Protects any code that updates the screen.

Protects any code that updates the collision\_position array.

Protects the use\_colour global variable.

**score\_mutex**

Protects the score global variables: shot\_update, score\_update, escape\_update.

**shot\_mutex**

Protects updates to shot\_update value and the save global variable.

Protects the save global variable.

**replace\_mutex**

Protects the replace\_index global variable.

Used with the replace\_condition condition variable to signal replacement thread creation.

**end\_mutex**

Used with the end\_condition condition variable to exit program.

GENERAL OVERVIEW OF CONDITION VARIABLES

**replace\_condition**

Used to send signal that a thread should be replaced right before the thread exits.

**end\_condition**

Used to let the main function know to close all threads and exit the program.

SAUCER INFORMATION/THREAD STORAGE

**struct saucerprop saucerinfo[MAXSAUCERS];**

Information about individual saucers are stored in the saucerinfo array and are identified by index. The array contains information about row, speed, index in the thread array (saucer\_t), and kill condition for the saucer. The row and delay are randomly assigned to the saucer when it is created by calling the setup\_saucer function. The kill condition is set by the find\_hit function called by one of the shot threads and signals for the saucer thread to erase itself and exit. Saucerinfo does not need to be protected by a mutex because each thread has its own index in the array where it updates the information.

**pthread\_t saucer\_t[MAXSAUCERS];**

The thread ids for all the active saucer threads are stored in the saucer\_t array. The array can have a fixed size of MAXSAUCERS because the replace\_thread function re-uses space in this array once a thread has exited. Again, each sacuer has one unique index in the array so protection is not required.

SHOT INFORMATION/THREAD STORAGE

**struct shotprop shotinfo[MAXSHOTS];**

Information about individual shots are stored in the shotinfo array and are identified by index. The array contains the row and column corresponding with the shot’s current position. Each shot has a different index so no protection needed.

**pthread\_t shot\_t[MAXSHOTS];**

The thread ids for all active shot arrays are stored in the shot\_t array. Protection is not needed because each thread is stored at a different index. Due to the nature of the shots there can only be a set number of active shots at one time. To take advantage of that property the shot threads get assigned indices ranging from only 0 to MAXSHOTS-1. Then when MAXSHOTS is the next index the function instead assigns the shot an index of 0 and starts over again. This method is simpler than allocating memory each time and does not use much more space if used correctly.

NOTE: the window size and speed of the shots will effect how large your MAXSHOTS number should be. If the window is large and/or the speed is high it will be necessary to increase MAXSHOTS to account for more shot threads being active at one time.

COLLISION DETECTION

**struct screen collision\_position[LINES-1][COLS-1];**

The collision\_position array is multidimentional and is indexed by row and column. This is done so there is a *screen struct* for each spot on the screen. I could have made this array smaller by allocating only the rows that saucer can appear on but since there are no memory restrictions I didn’t bother. Each saucer and shot updates it’s information in the collision\_position array at the row and column it moves to, and removes its old position. Saucers update the *saucer* variable and the *here* array at their index, and shots update the *shots* variable. Saucer can be >1 if there is more than 1 saucer at that location, in which case the here array will have a 1 set at multiple indices corresponding to the present saucers. Because the position of a saucer or shot corresponds with its location on the screen and multiple threads can try to write to the same position, updates to the collision\_position array are always protected by the *draw* mutex.

**struct saucerprop saucerinfo[MAXSAUCERS];**

The *saucerprop struct* has a variable called *kill* that always contains either a 1 or a 0. When kill is set to 1 for a specific saucerinfo element it signals that thread to remove the image and position of the saucer from the screen and exit the thread. The kill signal is set in the **find\_hit** function when a shot and saucer are found at the same position in the *collision\_position* array. It is also protected by the *draw* mutex because it updates the collision\_position array.

UPDATING THE SCORE

**int escape\_update ;**

Stores the number of escaped saucers. Incremented only.

**int shot\_update;**

Stores the number of shots used. Can be incremented or decremented.

**int score\_update;**

Stores the score. Incremented only.

**pthread\_mutex\_t score\_mutex;**

This mutex is used to protect any updates to the score (escaped saucers, shots left, score) so that the variables are only updated by one thread at a time.

**pthread\_mutex\_t shot\_mutex;**

This mutex is used to protect the shot\_update and save variable by being updated by more than one thread at a time.

UPDATING THE SCREEN

**pthread\_mutex\_t draw;**

The draw mutex protects the collision\_position array and any functions that update the terminal screen. This is used to prevent multiple threads from trying to write to the screen or update their position. The draw mutex can be locked by calling **lock\_draw** and unlocked by calling **unlock\_draw.** Using those funtions moves the cursor back to the corner of the screen and calls refresh.

REPLACING A THREAD

**int replace\_index;**

Stores the index of a saucer that is ready to be replaced by a new thread.

***pthread\_cond\_t replace\_condition;***

Once a saucer thread is finished it signals the *replace\_condition* which is received by a thread in the **replace\_thread** function that is waiting on that condition. A join is also used to ensure that the old thread has exited before the new thread is created.

**pthread\_mutex\_t replace\_mutex;**

This mutex protects the *replace\_index* from being updated at the same time by multiple threads so that the threads will be replaced one at a time without missing any. It is associated with the *replace\_condition* condition variable.

CHECKING IF SHOTS HAVE RUN OUT

**int save;**

Stores the most up to date shot thread index. Protected by the *shot\_mutex* to ensure that it is updated in the order that the threads are created.

**pthread\_t end\_t;**

Stores the thread that checks if the end conditions of the game are met because no shots are left.

SIGNALLING THE END OF THE GAME

**int save;**

Used for detecting game end by running out of saucers. See CHECKING IF SHOTS HAVE RUN OUT.

**int escape\_update ;**

Used for detecting game end by too many saucers escaping. See UPDATING THE SCORE.

***pthread\_cond\_t end\_condition;***

Once the end condition of a thread (either too many escaped saucers, no shots left, or used quit program) is met the thread signals *end\_condition*. The signal is received by a thread in the **main** function that is waiting on that condition. All other threads are cancelled and the program closes.

**pthread\_mutex\_t end\_mutex;**

The mutex associated with the *end\_condition* condition variable.

Game Specifics

GAME SPECIFICS

-if hit multiple saucers per row get one point for each saucer, and one new saucer for each saucer

-random speed/row for increased user enjoyment

Random variable creation: rand() with seed set by srand(getpid)

Seed of get\_pid() ensures that each game will have different rand() results each time it is played

rand() uses in game:

Saucer speed/row

setup\_saucer: used to randomize the speed and the row of each saucer thread

Random saucer creation

process\_input: the more key presses by the user, the more likely it is for a random saucer to be created

describe the user interface to your game -

including your reasons for making particular decisions about it.

Note - your TA should be able to read your design documentation and

then know how to read your code and play the game.

FIND END THREAD STORAGE

replace for the saucers

looping for the shots array