Sudoku Validator using Multithreading

Contents

System Specifications

On Machine

OS Name	Microsoft Windows 10 Pro
Version	10.0.18363 Build 18363
System Name	LEGION
System Manufacturer	LENOVO
System Model	81SY
System Type	x64-based PC
System SKU	LENOVO_MT_81SY_BU_idea_FM_Legion Y540-15IRH-PG0
Processor	Intel(R) Core(TM) i7-9750H CPU @ 2.60GHz, 2592 Mhz, 6 Core(s), 12 Logical Processor(s)
BIOS Version/Date	LENOVO BHCN44WW, 1/20/2022
BIOS Mode	UEFI
Installed Physical Memory (RAM)	16.0 GB
Total Physical Memory	15.9 GB
Available Physical Memory	11.7 GB
Total Virtual Memory	18.3 GB
Available Virtual Memory	11.3 GB
Page File Space	2.38 GB

On VirtualBox Ubuntu

OS Name	Ubuntu 22.04.1 LTS
OS Type	x64-based PC
Processor	Intel® Core™ i7-9750H CPU @ 2.60GHz × 6
GNOME Version	42.2
Windowing System	Wayland
Virtualization	Oracle
Hardware Model	innotek GmbH VirtualBox
Memory	8.8 GiB
Disk Capacity	53.7 GB

Pseudocode

Existing structures:

parameters -> structure for holding passing values for thread functions

ret -> structure for holding data a thread returns about cell in the puzzle

invalid_list -> structure for creating linked list of ret objects

1. **In class invalid_list:**

```
insertEntry(ret r)
    Boolean addNum := true

    If head of list is empty then
        head := r
        increment number of invalid entries
        return
    end if

    temp = head

    while temp.next != null
        if temp.index already exists in list then
            addNum := false
        end if

        temp := temp.next
    end while

    temp.next := r

    if addNum == true then
        increment number of invalid entries
    end if
```

2. Thread function for row searching

```
rowCheck(parameters p)
        int rVal
        rVal <- 1
        ret r
        initialize attributes of r

        for i <- p.startRow To 9

                boolean rowFlag : array[9]
                initialize rowFlag with false
                r.currentRow <- i

                for j <- p.startColumn To 9
                        r.currentCol <- j

                        if puzzle[i][j] < 1 OR puzzle[i][j] > 9 then
                                declare value to be outside the range
                                rVal <- 0

                                add mutex lock
                                insertEntry(r)
                                unlock mutex

                                create new r
                                initialize attributes of r

                        else if rowFlag[puzzle[i][j]-1] == false then
                                rowFlag[puzzle[i][j]-1] <- true

                        else
                                declare that value is repeating
                                rVal <- 0

                                add mutex lock
                                insertEntry(r)
                                unlock mutex

                                create new r
                                initialize attributes of r
                        end if

        return rVal
```

3. **Thread function for column searching**

```
colCheck(parameters p)
    int rVal
    rVal <- 1
    ret r
    initialize attributes of r

    for i <- p.startColumn To 9

        boolean colFlag : array[9]
        initialize colFlag with false
        r.currentCol <- i

        for j <- p->startRow To  9
            r.currentRow <- j

            if puzzle[j][i] < 1 OR puzzle[j][i] > 9 then
                declare value to be outside the range
                rVal <- 0

                add mutex lock
                insertEntry(r)
                unlock mutex

                create new r
                initialize attributes of r

            else if rowFlag[puzzle[j][i] -1] == false then
                rowFlag[puzzle[j][i] -1] <- true

            else
                declare that value is repeating
                rVal <- 0

                add mutex lock
                insertEntry(r)
                unlock mutex

                create new r
                initialize attributes of r
            end if

    return rVal
```

4. Thread function for 3x3 grid searching

```
gridCheck(parameters p)
        int rVal
        rVal <- 1
        ret r
        initialize attributes of r

        for i <- p.startRow To p.startRow + 3

            boolean gridFlag : array[9]
            initialize gridFlag with false
            r.currentRow <- i

            for j <- p.startColumn To p.startColumn + 3
                r.currentCol <- j

                if puzzle[i][j] < 1 OR puzzle[i][j] > 9 then
                    declare value to be outside the range
                    rVal <- 0

                    add mutex lock
                    insertEntry(r)
                    unlock mutex

                    create new r
                    initialize attributes of r

                else if rowFlag[puzzle[i][j]-1] == false then
                    rowFlag[puzzle[i][j]-1] <- true

                else
                    declare that value is repeating
                    rVal <- 0

                    add mutex lock
                    insertEntry(r)
                    unlock mutex

                    create new r
                    initialize attributes of r
                end if

        return rVal
```

Implementation

1. **In class invalid_list:**

```
void insertEntry(ret* r)
{
        bool addNum = true; //if true, increment total entries

        //If no node in the list
        if(head == NULL)
        {
                head = r;
                numOfEntries++;
                return;
        }

        //Else
        ret* temp = head;

        while(temp->next != NULL)
        {
                //If index already exists as part of some data, do not increment
                //number of entries

                if(temp->row == r->row && temp->col == r->col)
                {
                        addNum = false;
                }

                //next node
                temp = temp->next;
        }
        if(temp->row == r->row && temp->col == r->col)
        {
                addNum = false;
        }

        temp->next = r; //adding to list

        if(addNum) //If entry is not already mentioned in the list, then increment
//numOfEntries
        {
                numOfEntries++;
        }

        return;
}
```

2. **Thread function for row searching:**

```
void* rowCheck(void* arg)

{

    parameters *p = (parameters*) arg; //Accessing arguments


    //Function return value

    int* rVal = new int;

    *rVal = 1; //Valid by default


    //Invalid entry object

    ret* r = new ret;

    r->id = pthread_self();     //thread id

    r->row_or_col_check = 0; //Signifies row operation



    for(int i = p->row; i < 9; i++)

    {

            //rowFlag[i] = True, if found number i in row, else remains false

            //                      {col1, col2, col3, col4, col5, col6, col7, col8, col9}

            bool rowFlag[9] = {false,false,false,false,false,false,false,false,false};


            r->row = i; //setting current row number


            for(int j = p->column; j < 9; j++)

            {

                    r->col = j; //setting current col number



                    if(puzzle[i][j] < 1 || puzzle[i][j] > 9)

                    {
```

```
                r->out_of_range = true; //Value is out of range
                *rVal = 0; //Invalid

                //Critical section
                pthread_mutex_lock(&m1);

                    InvList->insertEntry(r);   //Adding to invalid entry list

                pthread_mutex_unlock(&m1);
                //Critical section finished

                //After storing invaid entry, a new object is created in case
//another invalid entry is found
                r = new ret;
                r->id = pthread_self();
                r->row_or_col_check = 0; //Signifies row operation
                r->row = i; //setting current row number
            }

            else if(rowFlag[puzzle[i][j]-1] == false)
            {
                rowFlag[puzzle[i][j]-1] = true;      //Valid case
            }
            else        //If number has already been found, the row is invalid since the
//number repeats
            {
                r->repeat = true; //Value is repeating
                *rVal = 0; //Invalid

                //Critical section
```

```
                    pthread_mutex_lock(&m1);

                         InvList->insertEntry(r);   //Adding to invalid entry list

                    pthread_mutex_unlock(&m1);
                    //Critical section finished

                    //After storing invaid entry, a new object is created in case
//another invalid entry is found
                    r = new ret;
                    r->id = pthread_self();
                    r->row_or_col_check = 0; //Signifies row operation
               }
          }

     }

     //If current object is valid, then do not add to list and deallocate space
     if(!r->out_of_range && !r->repeat)
     {
          delete r;
     }

     //exit
     pthread_exit((void*)rVal);
}
```

3. **Thread function for col searching:**

```
void* colCheck(void* arg)
{
    parameters *p = (parameters*) arg; //Accessing arguments

    //Function return value
    int* rVal = new int;
    *rVal = 1; //Valid by default

    //Invalid entry object
    ret* r = new ret;
    r->id = pthread_self();     //thread id
    r->row_or_col_check = 1; //Signifies col operation

    for(int i = p->column; i < 9; i++)
    {
            //colFlag[i] = True, if found number i in col, else remains false
            //                    {row1, row2, row3, row4, row5, row6, row7, row8, row9}
            bool colFlag[9] = {false,false,false,false,false,false,false,false,false};

            r->col = i; //setting current col number

            for(int j = p->row; j < 9; j++)
            {
                    r->row = j; //setting current row number

                    if(puzzle[j][i] < 1 || puzzle[j][i] > 9)
                    {

                            *rVal = 0; //Invalid
                            r->out_of_range = true; //Value is out of range

                            //Critical section
                            pthread_mutex_lock(&m1);

                                    InvList->insertEntry(r);   //Adding to invalid entry list

                            pthread_mutex_unlock(&m1);
                            //Critical section finished

                            //After storing invaid entry, a new object is created in case
//another invalid entry is found
                            r = new ret;
                    r->id = pthread_self();
                    r->row_or_col_check = 1; //Signifies col operation
                    r->col = i; //setting current col number
                }

                else if(colFlag[puzzle[j][i]-1] == false)
                {
                    colFlag[puzzle[j][i]-1] = true;     //Valid case
                }
                else    //If number has already been found, the row is invalid since the
//number repeats
                {
                    *rVal = 0; //Invalid
                    r->repeat = true; //Value is repeating

                    //Critical section
                    pthread_mutex_lock(&m1);

                        InvList->insertEntry(r);   //Adding to invalid entry list

                    pthread_mutex_unlock(&m1);
                    //Critical section finished

                    //After storing invaid entry, a new object is created in case
//another invalid entry is found
                    r = new ret;
                    r->id = pthread_self();
                    r->row_or_col_check = 1; //Signifies col operation
                }
            }

    }

    //If current object is valid, then do not add to list and deallocate space
    if(!r->out_of_range && !r->repeat)
    {
        delete r;
    }

    //exit
    pthread_exit((void*)rVal);
}
```

4. **Thread function for 3x3 grid searching:**

```
void* gridCheck(void* arg)
{
    parameters *p = (parameters*) arg; //Accessing arguments

    //Function return value
    int* rVal = new int;
    *rVal = 1; //Valid by default

    //Invalid entry object
    ret* r = new ret;
    r->id = pthread_self();    //thread id
    //r->row_or_col_check = -1 signifies grid check (set in constructor)

    //gridFlag[i] = True, if found number i in row, else remains false
    //                    {r1c1, r1c2, r1c3, r2c1, r2c2, r2c3, r3c1, r3c2, r3c3}
    bool gridFlag[9] = {false,false,false,false,false,false,false,false,false};


    for(int i = p->row; i < p->row + 3; i++)
    {
            r->row = i; //setting current row number

            for(int j = p->column; j < p->column + 3; j++)
            {
                    r->col = j; //setting current col number

                    if(puzzle[i][j] < 1 || puzzle[i][j] > 9)
                    {
                            *rVal = 0; //Invalid
                            r->out_of_range = true; //Value is out of range

                            //Critical section
                            pthread_mutex_lock(&m1);

                                    InvList->insertEntry(r);   //Adding to invalid entry list

                            pthread_mutex_unlock(&m1);
                            //Critical section finished

                            //After storing invaid entry, a new object is created in case
//another invalid entry is found
                            r = new ret;
                    r->id = pthread_self();
                }


                else if(gridFlag[puzzle[i][j]-1] == false)
                {
                    gridFlag[puzzle[i][j]-1] = true;
                }
                else    //If number has already been found, the row is invalid since the
//number repeats
                {
                    *rVal = 0; //Invalid
                    r->repeat = true; //Value is repeating

                    //Critical section
                    pthread_mutex_lock(&m1);

                        InvList->insertEntry(r);   //Adding to invalid entry list

                    pthread_mutex_unlock(&m1);
                    //Critical section finished

                    //After storing invaid entry, a new object is created in case
//another invalid entry is found
                    r = new ret;
                    r->id = pthread_self();
                }
            }

    }

    //If current object is valid, then do not add to list and deallocate space
    if(!r->out_of_range && !r->repeat)
    {
        delete r;
    }

    //exit
    pthread_exit((void*)rVal);
}
```

OS concepts applied

Using OOP concepts, the structures parameters, ret, and invalid_list are made. Using classes, a linked list is used to store each invalid index in the puzzle along with what error it causes (i.e. invalid row, invalid column, or invalid grid).

For this project, the solution concept includes 11 threads to be created. 1 for searching row-wise, 1 for searching column-wise, and 1 for each 3x3 grid in the puzzle (9 grids). For easy reference, let rowT be the row search thread, columnT be the colum search thread, and gridT be a grid search thread.

rowT is created with rowCheck function passed to it. columnT is created with colCheck function passed to it, and gridT is created with gridCheck function passed to it. The arguments are passed to every thread as (void*) since that is what the pthread_create() function expects. The structure to hold the arguments is parameters which stores the row and column from where to start the search.

In turn the threads return values in (void*) and these values are to be received as (void**). Using pointers as seen in the code in the previous section, the values are successfully returned. Additionally, in order to avoid errors, all return values are declared dynamically so that when the thread exits, the return variable is not deleted. A mutex lock is also used to lock the critical section where the threads write onto the shared invalid_list object.

The computations within the functions include simple matrix traversal techniques. At the end of each thread function, pthread_exit((void*)returnVal) is used to exit the thread while in the main function, the mutex is also destroyed.

Throughout this process, any allocated memory which is no longer useful is deallocated.

Other Scenarios

This concept can be used in bingo games. A thread is created for every player in the game and a thread is created for selecting a value from the board (like the announcer in real life bingo). The announcer thread will use a random number generator to traverse a random number of cells on the board to find the value everyone has to try and match. Each player thread traverses the other board of values which is with the player. If all the values presented by the announcer thread are matched by any thread first, then that player wins the round.