2023

Project Report
Operating Systems

GROUP MEMBERS:
FATIMA ABDUL WAHID 20I-0711
WALEED BIN OSAMA 20I-2440
SECTION: A

Table of Contents

Pseudocodes for Phase I

(Check if the board is valid or not)

- Print 9*9 Matrix through printBoard function
- Create 9 threads for 3*3 grid check
- Create 2 threads for checking column and rows
- Create structure for parameters and objects

For Grid check:

```
for (int i = 0; i < 9; i++)
    {
        for (int j = 0; j < 9; j++)
        {
            if (i % 3 == 0 && j % 3 == 0)
            {
                grid->row = i;
                grid->col = j;

                pthread_create(&threads[i], NULL, boxValidity, grid);
                pthread_join(threads[i], (void**)&obj);
         }
    }
}
```

- Create an object of struct objects to calculate the duplicates, invalids and remaining in the sudoku matrix.
- Traverse the grid for rows and column
- if(count[x]==2)
  the duplicates are found
- else
  Invalid entries are found
- if the 3*3 sub-grid is valid, map sub-grid to an index in first 9 indexes.

For row and column check:

- Create an object of struct objects to calculate the duplicates, invalids and remaining in the sudoku matrix.
- Apply mutex locks by pthread_mutex_lock (&resources) for row and call its respective functions named as rowValidity.
- Traverse the array to check the validity of row and column.
- if(count[x]==2)
  the duplicates are found
- else

Invalid entries are found
- Unlock resources for row
- Pthread_mutex_unlock(&resources)

For column:
- Create an object of struct objects to calculate the duplicates, invalids and remaining in the sudoku matrix.

- Lock resources for column
- Apply mutex locks by pthread_mutex_lock (&resources) for row and call its respective functions named as colValidity.
- Traverse the array to check the validity of row and column.
- if(count[x]==2)
  the duplicates are found
- else
  Invalid entries are found
- Unlock resources for column pthread_mutex_unlock(&resources).

Pseudocode for Phase II

- Calculate the total invalid entries in a 9*9 matrix
- Create threads which will be equal to invalid entries
- Each thread of either row or column with corresponding invalid entry will replace the invalid value with the remaining values left.
- The remaining values array was calculated above in row and column validity function.
- After correcting the invalid values, the final modified matrix is printed on the screen.

Implementation of Code

Structures

```
struct objects
{
    parameters invalid[30];
    int Ictr;
    parameters duplicate[30];
    int Dctr;
    parameters remaining[30];
    int Rctr;

    objects()
    {
        Ictr = 0;
        Dctr = 0;
        Rctr = 0;
    }
};
```

```
//structure for passing data to threads
struct parameters
{
    int row;
    int col;
    int val;

    parameters()
    {
        row = 0;
        col = 0;
        val = 0;
    }
};
```

Row Validity

```
//checking rows
void *rowValidity(void *args)
{
    objects *R = new objects;

    for(int i = 0; i < 9; i++)
    {
        int count[9] = {0, 0, 0, 0, 0, 0, 0, 0, 0};

        for(int j = 0; j < 9; j++)
        {
            if(sudokuBoard[i][j] >= 1 && sudokuBoard[i][j] <= 9)
            {
                if(sudokuBoard[i][j] == 1)
                    count[0]++;

                else if(sudokuBoard[i][j] == 2)
                    count[1]++;

                else if(sudokuBoard[i][j] == 3)
                    count[2]++;

                else if(sudokuBoard[i][j] == 4)
                    count[3]++;

                else if(sudokuBoard[i][j] == 5)
                    count[4]++;

                else if(sudokuBoard[i][j] == 6)
                    count[5]++;

                else if(sudokuBoard[i][j] == 7)
                    count[6]++;

                else if(sudokuBoard[i][j] == 8)
                    count[7]++;

                else if(sudokuBoard[i][j] == 9)
                    count[8]++;

                for(int x = 0; x < 9; x++)
                {
                    if(count[x] == 2)
```

```
                    {
                        R->duplicate[R->Dctr].row = i;
                        R->duplicate[R->Dctr].col = j;
                        R->duplicate[R->Dctr].val = sudokuBoard[i][j];
                        R->Dctr++;
                        count[x] = 1;
                        row_flag=1;
                        duplicates=true;
                        invalid_entries++;


                    }
                }
            }

            else
            {
                R->invalid[R->Ictr].row = i;
                R->invalid[R->Ictr].col = j;
                R->invalid[R->Ictr].val = sudokuBoard[i][j];
                R->Ictr++;
                row_flag=1;
                invalids=true;

            }
        }

        for(int x = 0; x < 9; x++)
        {
            if(count[x] == 0)
            {
                R->remaining[R->Rctr].row = i;
                R->remaining[R->Rctr].col = 0;
                R->remaining[R->Rctr].val = x + 1;
                R->Rctr++;
                invalid_entries++;




            }
        }
    }

    pthread_exit(R);
}
```

Grid Validity

```
//checking box
void *boxValidity(void * args)
{
    objects *B = new objects;
    parameters *box = (parameters *) args;
    int startRow = box->row;
    int startCol = box->col;
    int count[9] = {0, 0, 0, 0, 0, 0, 0, 0, 0};

    for(int i = startRow; i < startRow + 3; i++)
    {
        for(int j = startCol; j < startCol + 3; j++)
        {
            if(sudokuBoard[i][j] >= 1 && sudokuBoard[i][j] <= 9)
            {
                if(sudokuBoard[i][j] == 1)
                    count[0]++;

                else if(sudokuBoard[i][j] == 2)
                    count[1]++;

                else if(sudokuBoard[i][j] == 3)
                    count[2]++;

                else if(sudokuBoard[i][j] == 4)
                    count[3]++;

                else if(sudokuBoard[i][j] == 5)
                    count[4]++;

                else if(sudokuBoard[i][j] == 6)
                    count[5]++;

                else if(sudokuBoard[i][j] == 7)
                    count[6]++;

                else if(sudokuBoard[i][j] == 8)
                    count[7]++;

                else if(sudokuBoard[i][j] == 9)
                    count[8]++;
```

```
                for(int x = 0; x < 9; x++)
                {
                    if(count[x] == 2)
                    {
                        B->duplicate[B->Dctr].row = i;
                        B->duplicate[B->Dctr].col = j;
                        B->duplicate[B->Dctr].val = sudokuBoard[i][j];
                        B->Dctr++;
                        count[x] = 1;
                        box_flag=1;


                    }
                }
            }

            else
            {
                B->invalid[B->Ictr].row = i;
                B->invalid[B->Ictr].col = j;
                B->invalid[B->Ictr].val = sudokuBoard[i][j];
                B->Ictr++;
                box_flag=1;
            }

        }
    }

    for(int x = 0; x < 9; x++)
    {
        if(count[x] == 0)
        {
            B->remaining[B->Rctr].row = startRow;
            B->remaining[B->Rctr].col = startCol;
            B->remaining[B->Rctr].val = x + 1;
            B->Rctr++;



        }
    }

    pthread_exit(B);
}
```

Threads for Phase I:

```
    pthread_create(&t_row, NULL, rowValidity, NULL);
    pthread_join(t_row, (void**)&obj);

    for(int x = 0; x < obj->Ictr; x++)
    {

        cout << "\n The index " << obj->invalid[x].row << ", " << obj-
>invalid[x].col << " has invalid value " << obj->invalid[x].val << "\n";
        total_invalid++;
        ind[a]=obj->invalid[x].col;
        row[a]=obj->invalid[x].row;
        a++;
    }

    cout << "\n\n";

    for(int x = 0; x < obj->Dctr; x++)
    {
        cout << "\n The index " << obj->duplicate[x].row << ", " << obj-
>duplicate[x].col << " has duplicate value " << obj->duplicate[x].val << "\n";
        total_invalid++;
        ind[a]=obj->duplicate[x].col;
        row[a]=obj->duplicate[x].row;

        a++;


    }
```

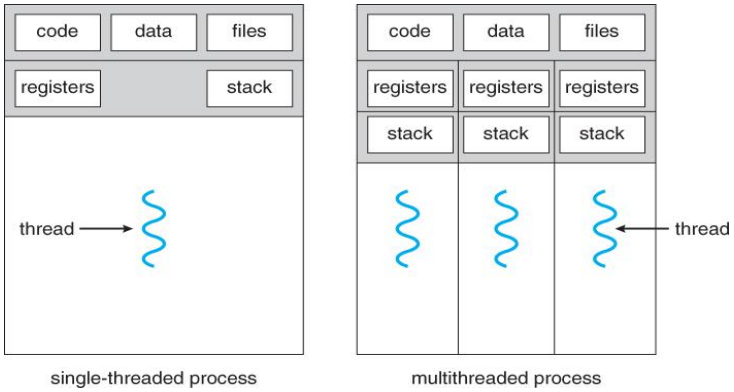
Phase II Sudoku Solution

```
void *solveSudoku(void *args)
{
    if(Robj->invalid[Robj->Ictr].val != 0)
    {
        sudokuBoard[Robj->invalid[Robj->Ictr].row][Robj->invalid[Robj-
>Ictr].col] = Robj->remaining[Robj->Rctr].val;
        Robj->Ictr++;
        Robj->Rctr++;
    }
```

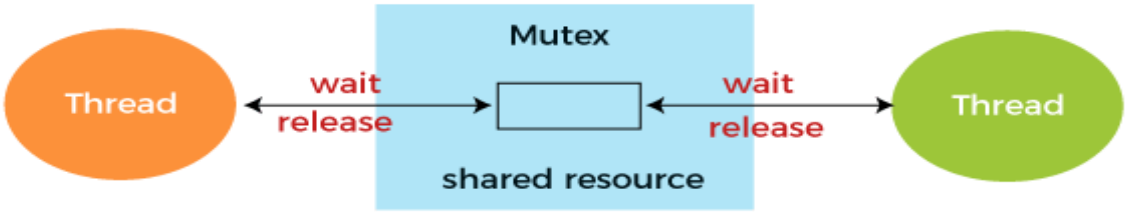
```
    else if(Robj->duplicate[Robj->Dctr].val != 0)
    {
        sudokuBoard[Robj->duplicate[Robj->Dctr].row][Robj->duplicate[Robj-
>Dctr].col] = Robj->remaining[Robj->Rctr].val;
        Robj->Dctr++;
        Robj->Rctr++;
    }

    else if(Cobj->duplicate[Cobj->Dctr].val != 0)
    {
        sudokuBoard[Cobj->duplicate[Cobj->Dctr].row][Cobj->duplicate[Cobj-
>Dctr].col] = Cobj->remaining[Cobj->Rctr].val;
        Cobj->Dctr++;
        Cobj->Rctr++;
    }

    else
    {
        row_flag = 0;
        col_flag = 0;
        box_flag = 0;
    }

    pthread_exit(NULL);
}
```

System Illustrations:

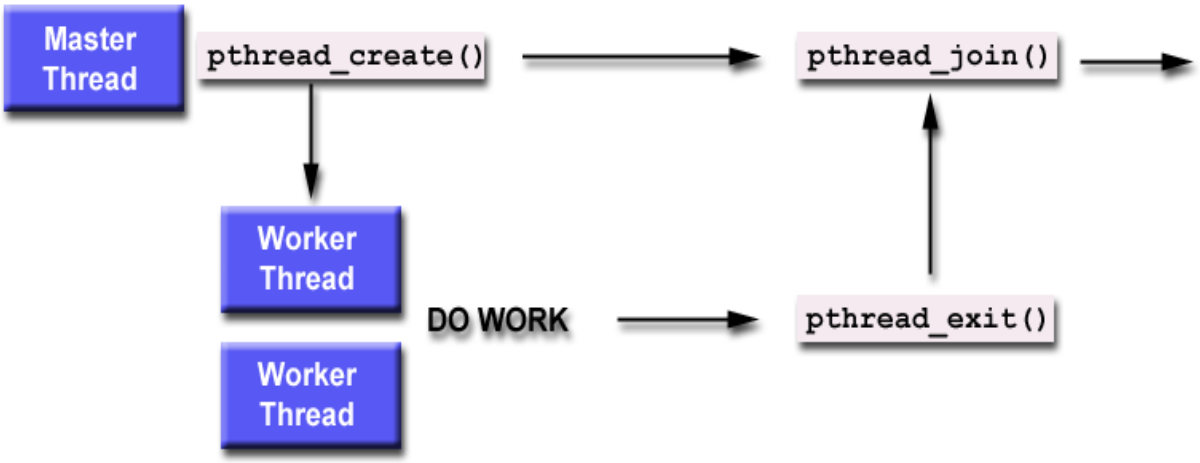
Threads



Mutex



Thread Joining



System Specifications:

Memory: 256 MB (minimum)

Graphics Card: Intel HD3000

CPU: Intel Core i3

File Size: 128 MB

OS: Ubuntu 14.05 LTS

Implementation in another scenario:

Threads are easy to implement in games and are good for CPU bound games. Another scenario could be a railway ticket system where multiple customers are accessing the system. The program creates threads for passengers and trains. Each customer thread will call the function for the station in which the system checks if the train is available. If it is available, the tickets will be generated otherwise the customer will have to wait. Similarly, for each train thread, the function for load seats will be called in which the seats are assigned to customer by checking the availability of the seats in train.