

**process.c**

#ifndef \_SVID\_SOURCE

#define \_SVID\_SOURCE

#endif

#include <unistd.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <errno.h>

#include <stdio.h>

#include <sys/wait.h>

#include <stdlib.h>

#include <fcntl.h>

#include <sys/mman.h>

#include <sys/shm.h>

#include <semaphore.h>

#include <sys/ipc.h>

sem\_t mutexLock;

int running\_processes = 0;

/\* Function used to validate the row \*/

void validRow(int row, int grid[9][9], int\* buffer2, int\* counter, pid\_t parentId)

{

FILE\* f = fopen("log.txt", "a");

int ii, validSubGrids = 0, numTo9;

/\* Used to check each number in the row are valid

\* If along the way, a number is evaluated and at that number's slot in the checkArray

\* is already marked as 1 (meaning there has been an occurrance of that number), then

\* it will print an error and carry on with the next slot.

\* If the number being evaluated against the checkArray marked as 0

\* (meaning the occurence of that number has yet to appear), mark that slot as 1

\* increment the total of validSubGrids by 1 and continue.

\*/

int checkArray[9] = {0,0,0,0,0,0,0,0,0};

/\* Validates that the row and column being evaluated is valid \*/

if(row > 8 || row < 0)

{

printf("Invalid row: %d\n", row);

exit(-1);

}

/\* Checks at row indicated through parameter at the ii'th column to see if the number is valid. \*/

for(ii = 0; ii < 9; ii++)

{

/\* Gets the value at index (row, i) to evaluate \*/

numTo9 = grid[row][ii];

/\* numTo9 signifies all the numbers available for sudoku 1-9. checkArray used numTo9 "-1" because it indexes at 0 \*/

if(numTo9 < 1 || numTo9 > 9 || checkArray[numTo9 - 1] == 1)

{

/\* WRITE TO FILE \*/

if(f == NULL)

{

perror("Error in opening file to write to.\n");

}

else

{

fprintf(f, "%d-%d: row %d is invalid\n", parentId + row, row+1, row+1);

}

}

else

{

/\* mark the slot as valid \*/

checkArray[numTo9-1] = 1;

validSubGrids++;

}

}

/\* Spinlocks until the semaphore is available to be used \*/

while(running\_processes == 1)

{

sem\_wait(&mutexLock);

}

running\_processes++;

/\* Alters shared memory \*/

if(validSubGrids == 9)

{

\*(buffer2+row) = 1;

\*counter = \*counter + 1;

}

/\* Unlocks lock for another process to use \*/

running\_processes--;

sem\_post(&mutexLock);

fclose(f);

}

/\* Function to evaluate each column for validity \*/

void validColumn(int grid[9][9], int\* buffer2, int\* counter, pid\_t parentId)

{

int ii, jj, kk, numTo9, isValid = 0;

int validSubGrids = 0, validColumn = 0;

FILE\* f = fopen("log.txt", "a");

/\* Used to check each number in the column are valid

\* If along the way, a number is evaluated and at that number's slot in the checkArray

\* is already marked as 1 (meaning there has been an occurrance of that number), then

\* it will print an error and carry on with the next slot.

\* If the number being evaluated against the checkArray marked as 0

\* (meaning the occurence of that number has yet to appear), mark that slot as 1

\* increment the total of validSubGrids by 1 and continue.

\*/

int checkArray[9] = {0,0,0,0,0,0,0,0,0};

/\* Increments the column index each time all the rows are gone through at a particular column \*/

for(jj = 0; jj < 9; jj++)

{

/\* Checks at row indicated through parameter at the ii'th column to see if the number is valid. \*/

for(ii = 0; ii < 9; ii++)

{

numTo9 = grid[ii][jj]; /\* Gets the value at index (row, i) to evaluate \*/

/\* numTo9 signifies all the numbers available for sudoku 1-9. checkArray used numTo9 "-1" because it indexes at 0 \*/

if(numTo9 < 1 || numTo9 > 9 || checkArray[numTo9 - 1] == 1)

{

isValid = 1;

}

else

{

/\* mark the slot as valid \*/

checkArray[numTo9-1] = 1;

validColumn++;

}

}

if(validColumn == 9)

{

validSubGrids++;

}

/\* Used to reset the checkArray all to 0 \*/

for(kk = 0; kk < 9; kk++)

{

checkArray[kk] = 0;

}

if(isValid == 1)

{

if(f == NULL)

{

perror("Error in opening file to write to.\n");

}

else

{

fprintf(f, "%d-10: column %d is invalid\n", parentId + 9, jj+1);

}

}

isValid = 0;

validColumn = 0;

}

/\* Spinlocks until lock is available \*/

while(running\_processes == 1)

{

sem\_wait(&mutexLock);

}

running\_processes++;

/\* Alters shared memory \*/

\*(buffer2+9) = validSubGrids;

\*counter = \*counter + validSubGrids;

/\* Unlock for another process to use \*/

running\_processes--;

sem\_post(&mutexLock);

fclose(f);

}

/\* Function used to validate a 3x3 sub-grid \*/

void valid3x3(int grid[9][9], int\* buffer2, int\* counter, pid\_t parentId)

{

int row = 0, col = 0, valid3x3 = 0, numTo9 = 0, validSubGrids = 0, isValid = 0;

int checkArray[9] = {0,0,0,0,0,0,0,0,0};

FILE\* f = fopen("log.txt", "a");

int ii = 0, jj = 0, kk = 0;

do

{

row = 0;

for (ii = row; ii < 9; ii++)

{

for (jj = col; jj < col + 3; jj++)

{

numTo9 = grid[ii][jj];

if (numTo9 < 1 || numTo9 > 9 || checkArray[numTo9 - 1] == 1)

{

/\* Notify that there is an invalid subregion \*/

isValid = 1;

}

else

{

checkArray[numTo9 - 1] = 1;

valid3x3++;

}

/\* If an invalid sub region is picked up, write to log file \*/

if(isValid == 1 && (ii == 2 || ii == 5 || ii == 8) && (jj == 2 || jj == 5 || jj == 8))

{

if(f == NULL)

{

perror("Error in opening file to write to.\n");

}

else

{

fprintf(f, "%d-11: sub-grid [%d...%d %d...%d] is valid\n", parentId + 11, ii-1, ii+1, jj-1, jj+1);

}

isValid = 0;

}

}

if(ii == 2 || ii == 5 || ii == 8)

{

/\* Reset checkArray to 0 for next iteration \*/

for(kk = 0; kk < 9; kk++)

{

checkArray[kk] = 0;

}

if(valid3x3 == 9)

{

validSubGrids++;

}

/\* reset counters for valid subregion and valid 3x3 \*/

isValid = 0;

valid3x3 = 0;

}

}

col = col + 3;

}while(col != 9);

/\* Spinlocks until lock is available \*/

while(running\_processes == 1)

{

sem\_wait(&mutexLock);

}

running\_processes++;

/\* Alters shared memory \*/

\*(buffer2+10) = validSubGrids;

\*counter = \*counter + validSubGrids;

/\* Unlocks mutex for another process to use \*/

running\_processes--;

sem\_post(&mutexLock);

fclose(f);

}

/\* Function to print the grid that contains the values of the sudoku puzzle \*/

void printPuzzle(int grid[9][9])

{

int ii, jj;

for(ii = 0; ii < 9; ii++)

{

for(jj = 0; jj < 9; jj++)

{

printf("%d ", grid[ii][jj]);

}

printf("\n");

}

printf("\n");

}

/\* Function to access the buffer1 that contains the grid when the file is opened \*/

void readFile(int grid[9][9], char\* buffer1)

{

int ii, jj, lineCount = 0;

/\* Maps each grid index with the value given via the buffer \*/

for(ii = 0; ii < 9; ii++)

{

for(jj = 0; jj < 9; jj++)

{

sscanf(buffer1 + lineCount, "%d", &grid[ii][jj]);

lineCount = lineCount + 2;

}

}

}

/\* Prints all the validation statements for the specifications with the values in buffer2 and counter with the process IDs \*/

void printValidation(int\* buffer2, int\* counter, pid\_t parentId)

{

int ii;

printf("\n------------------------------------------------------\n");

printf("COUNTER: %d\n", \*counter);

for(ii = 0; ii < 11; ii++)

{

if(ii < 9 && \*(buffer2+ii) == 1)

{

printf("\nValidation result from %d-%d: row %d is valid", parentId + ii, ii+1, ii+1);

}

else if(ii < 9 && \*(buffer2+ii) == 0)

{

printf("\nValidation result from %d-%d: row %d is invalid", parentId + ii, ii+1, ii+1);

}

else if(ii == 9)

{

printf("\nValidation result from %d-%d: %d of 9 columns are valid", parentId + ii, ii+1, \*(buffer2+9));

}

else if(ii == 10)

{

printf("\nValidation result from %d-%d: %d of 9 sub-grids are valid", parentId + ii, ii+1, \*(buffer2+10));

}

else

{

printf("\nERROR!!\n");

}

}

if(\*counter == 27)

{

printf("\n\nThere are 27 valid sub-grids, and thus the solution is valid.\n");

}

else

{

printf("\n\nThere are %d valid sub-grids, and thus the solution is invalid.", \*counter);

}

printf("\n------------------------------------------------------\n");

}

int main(int argc, char\* argv[])

{

pid\_t parentId, pid, childPid = 0;

int grid[9][9];

int sudokuPuzzle = 0, buf2 = 0, countPtr = 0;

char\* buffer1 = NULL;

int\* buf2Ptr;

int\* counterPtr;

char\* filename = argv[1];

int PUZZLE\_SIZE = 9\*9\*sizeof(int);

int maxDelay = atoi(argv[2]);

/\* Resets all content in the log file to blank before executing the validations \*/

FILE\* f = fopen("log.txt", "w");

fclose(f);

/\* Initialises the mutex to value of 1 to be able to be accessed by a process \*/

sem\_init(&mutexLock, 0, 1);

/\* READING IN THE SUDOKU PUZZLE FROM THE FILE AND ALLOCATING IT TO A GRID \*/

sudokuPuzzle = open(filename, O\_RDWR);

if(sudokuPuzzle == -1)

{

printf("Error opening file\n");

}

buffer1 = mmap(0, PUZZLE\_SIZE, PROT\_READ | PROT\_WRITE, MAP\_SHARED, sudokuPuzzle, 0);

readFile(grid, buffer1);

printPuzzle(grid);

/\* Creates the memory space for buffer2 to identify the various subregions that are valid \*/

buf2 = shm\_open("buf2Ptr", O\_CREAT|O\_RDWR, 0666);

buf2Ptr = mmap(0, 11\*sizeof(int), PROT\_READ | PROT\_WRITE, MAP\_SHARED|MAP\_ANONYMOUS, buf2, 0);

/\* Creates the memory space for counter to identify the amount of valid subgrids

\* Using MAP\_ANONYMOUS intiialised the value of the mmap to 0 \*/

countPtr = shm\_open("counterPtr", O\_CREAT|O\_RDWR, 0666);

counterPtr = mmap(NULL, sizeof(int), PROT\_READ | PROT\_WRITE, MAP\_SHARED|MAP\_ANONYMOUS, countPtr, 0);

\*counterPtr = 0;

/\* Forks child processes to allocate a region to validate \*/

parentId = getpid();

fork();

pid = fork();

fork();

if(pid == 0)

{

fork();

}

childPid = getpid() - parentId;

if(childPid == 0)

{

wait(NULL);

}

else if(childPid > 0 && childPid < 10)

{

/\* DO ROW STUFF \*/

validRow(childPid-1, grid, buf2Ptr, counterPtr, parentId);

sleep(rand() % maxDelay);

}

else if (childPid == 10)

{

/\* DO COLUMN STUFF \*/

validColumn(grid, buf2Ptr, counterPtr, parentId);

sleep(rand() % maxDelay);

}

else if (childPid == 11)

{

/\* DO 3x3 STUFF \*/

valid3x3(grid, buf2Ptr, counterPtr, parentId);

sleep(rand() % maxDelay);

}

else

{

printf("ERROR IN FORKING PROCESSES -- PID: %d\n", childPid);

}

wait(NULL);

wait(NULL);

wait(NULL);

/\* When all child processes finish, print validation statements \*/

if(childPid == 0)

{

printValidation(buf2Ptr, counterPtr, parentId);

}

shm\_unlink("buf2Ptr");

shm\_unlink("counterPtr");

munmap(buffer1, PUZZLE\_SIZE);

munmap(buf2Ptr, 11\*sizeof(int));

munmap(counterPtr, sizeof(int));

exit(0);

return 0;

}

**process.h**

void validRow(int row, int grid[9][9], int\* buffer2, int\* counter);

void validColumn(int grid[9][9], int\* buffer2, int\* counter);

void valid3x3(int grid[9][9], int\* buffer2, int\* counter);

void printPuzzle(int grid[9][9]);

void readFile(int grid[9][9], char\* buffer1);

void printValidation(int\* buffer2, int\* counter, pid\_t parentId);

**process/README.txt**

This program is to run for my process version of my sudoku validator program.

========== READ ME ==========

1. To compile the program, run:

make clean

make

2. To run the program, run:

./mssv $1 5

where:

$1 - the sudoku puzzle that is to be validated

5 - is the maximum delay that I have specified for the program to run at.

This number can be changed to your liking.

The higher, the slower the program runs.

**threads.c**

#include <pthread.h>

#include <unistd.h>

#include <stdio.h>

#include <stdlib.h>

#include <sched.h>

#include "threads.h"

volatile int running\_threads = 0;

pthread\_t threads[11];

pthread\_mutex\_t mutexLock;

pthread\_cond\_t condition;

/\* This function is to read in the sudoku puzzle and allocate the grid to buffer1 \*/

void readFile(char\* filename, Shared\_Memory\* shared\_memory)

{

FILE\* f = fopen(filename, "r");

int done = 0, readN, sudokuNum, row = 0, jj;

if(f == NULL)

{

perror("Error in opening file for Sudoku Puzzle");

}

else

{

do

{

for(jj = 0; jj < 9; jj++)

{

/\* read the variable in the file \*/

readN = fscanf(f, "%d", &sudokuNum);

/\* If it reads the end of file, stop the loop \*/

if(readN == EOF)

{

done = 1;

}

/\* Otherwise, continue \*/

else

{

/\* At a particular slot in row ii and column jj, place the number being read in into that slot \*/

shared\_memory->buffer1[row][jj] = sudokuNum;

}

/\* Reads in the space that separates the numbers in the sudoku solution \*/

readN = fscanf(f, " ");

}

row++;

}while(!done);

}

fclose(f);

}

/\* This function is to validate the rows to see if all numbers are unique \*/

void\* validRow(void \*sh\_mem)

{

Shared\_Memory\* shared\_memory = (Shared\_Memory\*)sh\_mem;

/\* defines its own address \*/

pthread\_t myTid = pthread\_self();

FILE\* f = fopen("log.txt", "a");

int ii, validSubGrids = 0, foundThread = 0, row = 0, numTo9;

/\* Used to check each number in the row are valid

\* If along the way, a number is evaluated and at that number's slot in the checkArray

\* is already marked as 1 (meaning there has been an occurrance of that number), then

\* it will print an error and carry on with the next slot.

\* If the number being evaluated against the checkArray marked as 0

\* (meaning the occurence of that number has yet to appear), mark that slot as 1

\* increment the total of validSubGrids by 1 and continue.

\*/

int checkArray[9] = {0,0,0,0,0,0,0,0,0};

foundThread = pthread\_equal(myTid, threads[row]);

while((row < 8) && (foundThread == 0))

{

/\* checks current thread against all the other threads incrementing each time it is wrong, to provide the row number \*/

foundThread = pthread\_equal(myTid, threads[row]);

row++;

}

/\* Validates that the row and column being evaluated is valid \*/

if(row > 8 || row < 0)

{

printf("Invalid row: %d", row);

pthread\_exit(NULL);

}

/\* Checks at row indicated through parameter at the ii'th column to see if the number is valid. \*/

for(ii = 0; ii < 9; ii++)

{

/\* Gets the value at index (row, i) to evaluate \*/

numTo9 = shared\_memory->buffer1[row][ii];

/\* numTo9 signifies all the numbers available for sudoku 1-9. checkArray used numTo9 "-1" because it indexes at 0 \*/

if(numTo9 < 1 || numTo9 > 9 || checkArray[numTo9 - 1] == 1)

{

/\* WRITE TO FILE \*/

if(f == NULL)

{

perror("Error in opening file to write to.\n");

}

else

{

fprintf(f, "%lu-%d: row %d is invalid\n", myTid, row+1, row+1);

}

}

else

{

/\* mark the slot as valid \*/

checkArray[numTo9-1] = 1;

validSubGrids++;

}

}

pthread\_mutex\_lock(&mutexLock);

while(running\_threads == 1)

{

pthread\_cond\_wait(&condition, &mutexLock);

}

running\_threads++;

/\* Allocates certain values to the buffer2 and counter \*/

if(validSubGrids == 9)

{

shared\_memory->buffer2[row] = 1;

shared\_memory->counter++;

}

fclose(f);

pthread\_cond\_signal(&condition);

running\_threads--;

pthread\_mutex\_unlock(&mutexLock);

pthread\_exit(NULL);

}

void\* validColumn(void\* sh\_mem)

{

int ii, jj, kk, validSubGrids = 0, validColumn = 0, numTo9, isValid = 0;

FILE\* f = fopen("log.txt", "a");

pthread\_t myTid = pthread\_self();

/\* Used to check each number in the column are valid

\* If along the way, a number is evaluated and at that number's slot in the checkArray

\* is already marked as 1 (meaning there has been an occurrance of that number), then

\* it will print an error and carry on with the next slot.

\* If the number being evaluated against the checkArray marked as 0

\* (meaning the occurence of that number has yet to appear), mark that slot as 1

\* increment the total of validSubGrids by 1 and continue.

\*/

int checkArray[9] = {0,0,0,0,0,0,0,0,0};

Shared\_Memory\* shared\_memory = (Shared\_Memory\*)sh\_mem;

/\* Increments the column index each time all the rows are gone through at a particular column \*/

for(jj = 0; jj < 9; jj++)

{

/\* Checks at row indicated through parameter at the ii'th column to see if the number is valid. \*/

for(ii = 0; ii < 9; ii++)

{

/\* Gets the value at index (row, i) to evaluate \*/

numTo9 = shared\_memory->buffer1[ii][jj];

/\* numTo9 signifies all the numbers available for sudoku 1-9. checkArray used numTo9 "-1" because it indexes at 0 \*/

if(numTo9 < 1 || numTo9 > 9 || checkArray[numTo9 - 1] == 1)

{

/\* WRITE TO FILE \*/

isValid = 1;

}

else

{

/\* mark the slot as valid \*/

checkArray[numTo9-1] = 1;

validColumn++;

}

}

if(validColumn == 9)

{

validSubGrids++;

}

/\* Used to reset the checkArray all to 0 \*/

for(kk = 0; kk < 9; kk++)

{

checkArray[kk] = 0;

}

if(isValid == 1)

{

if(f == NULL)

{

perror("Error in opening file to write to.\n");

}

else

{

fprintf(f, "%lu-10: column %d is invalid\n", myTid, jj+1);

}

}

isValid = 0;

validColumn = 0;

}

fclose(f);

/\* Locks the mutex so this thread can access its critical section \*/

pthread\_mutex\_lock(&mutexLock);

while(running\_threads == 1)

{

pthread\_cond\_wait(&condition, &mutexLock);

}

running\_threads++;

/\* Update the buffer2 at slot [9] with the number of valid columns \*/

shared\_memory->buffer2[9] = validSubGrids;

shared\_memory->counter = shared\_memory->counter + validSubGrids;

/\* Unlocks mutex for another thread to use the lock \*/

pthread\_cond\_signal(&condition);

running\_threads--;

pthread\_mutex\_unlock(&mutexLock);

pthread\_exit(NULL);

}

/\* Function to evaluate a 3x3 sub-grid \*/

void\* valid3x3(void\* sh\_mem)

{

int row = 0, col = 0, valid3x3 = 0, numTo9 = 0, validSubGrids = 0, isValid = 0;

int checkArray[9] = {0,0,0,0,0,0,0,0,0};

FILE\* f = fopen("log.txt", "a");

pthread\_t myTid = pthread\_self();

Shared\_Memory\* shared\_memory = (Shared\_Memory\*)sh\_mem;

int ii = 0, jj = 0, kk = 0;

do

{

row = 0;

for (ii = row; ii < 9; ii++)

{

for (jj = col; jj < col + 3; jj++)

{

numTo9 = shared\_memory->buffer1[ii][jj];

if (numTo9 < 1 || numTo9 > 9 || checkArray[numTo9 - 1] == 1)

{

isValid = 1;

}

else

{

checkArray[numTo9 - 1] = 1;

valid3x3++;

}

/\* If an invalid sub region is picked up, write to log file \*/

if(isValid == 1 && (ii == 2 || ii == 5 || ii == 8) && (jj == 2 || jj == 5 || jj == 8))

{

if(f == NULL)

{

perror("Error in opening file to write to.\n");

}

else

{

fprintf(f, "%lu-11: sub-grid [%d...%d %d...%d] is valid\n", myTid, ii-1, ii+1, jj-1, jj+1);

}

isValid = 0;

}

}

if(ii == 2 || ii == 5 || ii == 8)

{

/\* Reset checkArray to 0 for next iteration \*/

for(kk = 0; kk < 9; kk++)

{

checkArray[kk] = 0;

}

if(valid3x3 == 9)

{

validSubGrids++;

}

valid3x3 = 0;

}

}

col = col + 3;

}while(col != 9);

fclose(f);

/\* Lock the mutex to change shared memory \*/

pthread\_mutex\_lock(&mutexLock);

while(running\_threads == 1)

{

pthread\_cond\_wait(&condition, &mutexLock);

}

running\_threads++;

/\* Update shared memory here \*/

shared\_memory->buffer2[10] = validSubGrids;

shared\_memory->counter = shared\_memory->counter + validSubGrids;

/\* Unlock mutex for another thread to use \*/

pthread\_cond\_signal(&condition);

running\_threads--;

pthread\_mutex\_unlock(&mutexLock);

pthread\_exit(NULL);

}

/\* Prints all the validation statements at the end \*/

void printValidation(Shared\_Memory\* shared\_memory)

{

int ii = 0;

printf("COUNTER: %d\n", shared\_memory->counter);

for(ii = 0; ii < 11; ii++)

{

if(ii < 9 && shared\_memory->buffer2[ii] == 1)

{

printf("\nValidation result from %lu-%d: row %d is valid", threads[ii], ii+1, ii+1);

}

else if(ii < 9 && shared\_memory->buffer2[ii] == 0)

{

printf("\nValidation result from %lu-%d: row %d is invalid", threads[ii], ii+1, ii+1);

}

else if(ii == 9)

{

printf("\nValidation result from %lu-%d: %d of 9 columns are valid", threads[9], ii+1, shared\_memory->buffer2[ii]);

}

else if(ii == 10)

{

printf("\nValidation result from %lu-%d: %d of 9 sub-grids are valid", threads[10], ii+1, shared\_memory->buffer2[ii]);

}

else

{

printf("\nERROR!!\n");

}

}

if(shared\_memory->counter == 27)

{

printf("\n\nThere are 27 valid sub-grids, and thus the solution is valid.");

}

else

{

printf("\n\nThere are %d valid sub-grids, and thus the solution is invalid.", shared\_memory->counter);

}

}

/\* Print puzzle that the program is working with \*/

void printPuzzle(Shared\_Memory\* shared\_memory)

{

int ii, jj;

printf("\n");

for(ii = 0; ii < 9; ii++)

{

for(jj = 0; jj < 9; jj ++)

{

printf("%d ", shared\_memory->buffer1[ii][jj]);

}

printf("\n");

}

}

int main(int argc, char\* argv[])

{

FILE\* f;

char\* filename = NULL;

int ii = 0, rc = 1, maxDelay;

Shared\_Memory\* shared\_memory = (Shared\_Memory\*)malloc(sizeof(Shared\_Memory));

filename = argv[1];

maxDelay = atoi(argv[2]);

/\* Used to reset the log file each time the program starts running \*/

f = fopen("log.txt", "w");

fclose(f);

/\* Reads the sudoku puzzle from file \*/

readFile(filename, shared\_memory);

printPuzzle(shared\_memory);

for(ii = 0; ii < 11; ii++)

{

shared\_memory->buffer2[ii] = 0;

}

shared\_memory->counter = 0;

/\* initialises the mutex for use \*/

pthread\_mutex\_init(&mutexLock, NULL);

/\* initialises the condition to wait and signal \*/

pthread\_cond\_init(&condition, NULL);

/\* Allocate each thread its section to validate \*/

for(ii = 0; ii < 11; ii++)

{

if(ii < 9)

{

rc = pthread\_create(&threads[ii], NULL, &validRow, (void\*)shared\_memory);

sleep(rand() % maxDelay);

}

else if(ii == 9)

{

rc = pthread\_create(&threads[9], NULL, &validColumn, (void\*)shared\_memory);

sleep(rand() % maxDelay);

}

else if(ii == 10)

{

rc = pthread\_create(&threads[10], NULL, &valid3x3, (void\*)shared\_memory);

sleep(rand() % maxDelay);

}

else

{

printf("ERROR; return code from pthread\_create() is %d\n", rc);

exit(-1);

}

}

for(ii = 0; ii < 11; ii++)

{

pthread\_join(threads[ii], NULL);

}

/\* Prints all the validation statements \*/

printf("\n------------------------------------------------------\n");

printValidation(shared\_memory);

printf("\n------------------------------------------------------\n");

free(shared\_memory);

/\* destroys the mutex after use \*/

pthread\_mutex\_destroy(&mutexLock);

pthread\_exit(NULL);

}

**threads.h**

typedef struct{

int buffer1[9][9];

int buffer2[11];

int counter;

} Shared\_Memory;

void readFile(char\* filename, Shared\_Memory\* shared\_memory);

void printPuzzle(Shared\_Memory\* shared\_memory);

void\* validRow(void\* sh\_mem);

void\* validColumn(void\* sh\_mem);

void\* valid3x3(void\* shared\_memory);

void printValidation(Shared\_Memory\* shared\_memory);

**threads/README.txt**

This program is to run for my multithreading sudoku validator program.

========== READ ME ==========

1. To compile the program, run:

make clean

make

2. To run the program, run:

./mssv $1 5

where:

$1 - the sudoku puzzle that is to be validated

5 - is the maximum delay that I have specified for the program to run at.

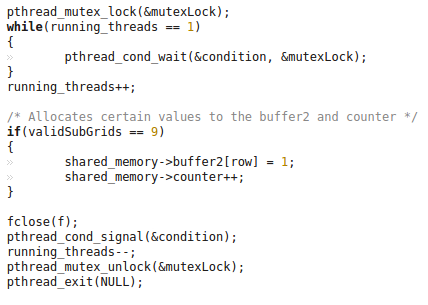
This number can be changed to your liking.

The higher, the slower the program runs.

Discuss how any mutual exclusion is achieved and what processes/threads access the shared resources.

# Threads

Mutual exclusion has been implemented in my program through the use of the mutex locks called by the lines pthread\_mutex\_lock() and pthread\_mutex\_unlock(). These two functions allow for a mutexLock global variable to be altered for a thread to be able to access a function one at a time. This allows for each thread to execute the function completely before another thread can execute its own function that it has been assigned to, to evaluate. Using pthread\_cond\_wait() and pthread\_cond\_signal() allows for threads to suspend their execution while another thread is running (which is defined by a global variable running\_threads) and would break out of the while loop in which a thread is suspended when running\_threads decrements and pthread\_cond\_signal() is called for another thread to start executing its critical section and leaving other threads waiting on the lock to be available again.

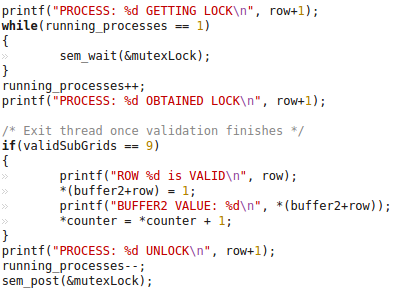


Evident from the figure above, pthread\_mutex\_lock() has been called to lock the mutex so that no other thread is able to use the lock while another thread is using it. A thread will wait (through the calling of pthread\_cond\_wait()) until a running thread signals so that a thread stops waiting. Towards the end of the critical section, pthread\_cond\_signal() is called to identify another thread to stop waiting and execute its function, where this will unlock the mutex with pthread\_mutex\_unlock() for another thread to use and will exit the thread at the end of the function. Buffer2 and counter here are being modified which is to be exclusive for each thread running. Therefore, having mutex locks allows for only one thread to ever run its critical section at a time.

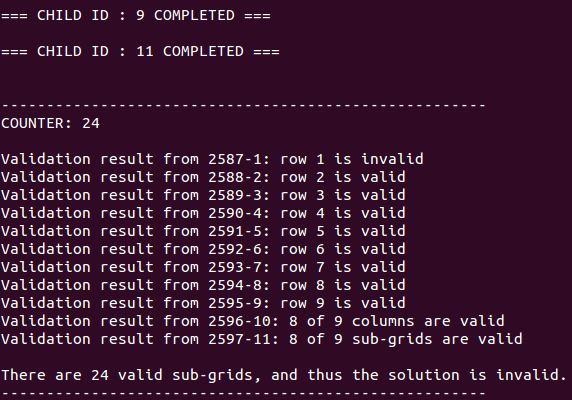
Mutual exclusion has been implemented through the above examples which is also evident in other functions in the program. I think that my program might have allowed for two levels of locking; one with pthread\_mutex\_lock() and pthread\_mutex\_unlock() and one with the while loop running when running\_thread == 1. This allows for a thread to be in a spinlock and will keep waiting until another thread decrements running\_threads for a thread to stop waiting and continue. I feel that mutual exclusion might have been addressed in this way.

# Processes

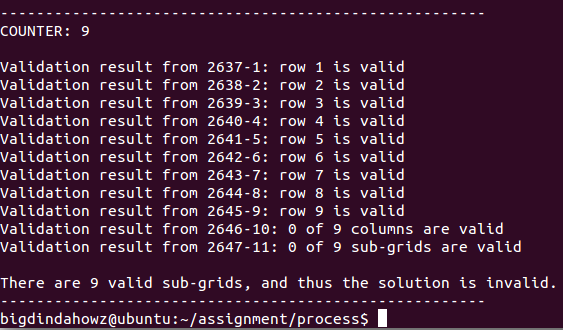
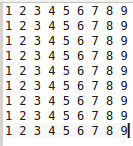
Mutual exclusion has been achieved in my process version of my Sudoku validator through the use of my sem\_wait() and sem\_post() functions.



Each process executes its functions that the parent process has allocated its children to. When all processes execute their functions. Towards the end of the function, the function will call on sem\_wait() while running\_processes is 1. The variable running\_processes identifies the amount of running processes currently running at a particular time. That variable will not increment until the process stops waiting and decides to lock the available lock to enter its critical section. When that happens, only one process will take the lock while other processes are in a spinlock waiting for the lock to free up to be used. Buffer2 and counter are being modified here which is the critical section that each process is to reach. By having a mutex lock before and after the modifications ensures that only one process at a time is able to modify those values at a time.



With the output of the program, with the use of mutex locks, this allows for correct values when validating each sub-regions. The image above is using a grid with one invalid value at the index row 1, column 9. The modification of the valid Sudoku grid into an invalid one by replacing a value will provide three invalid sub-regions which is what the program detects. Thus, the program is utilizing buffer2 and counter as it is supposed to and modifying those values correctly and exclusively one process at a time.



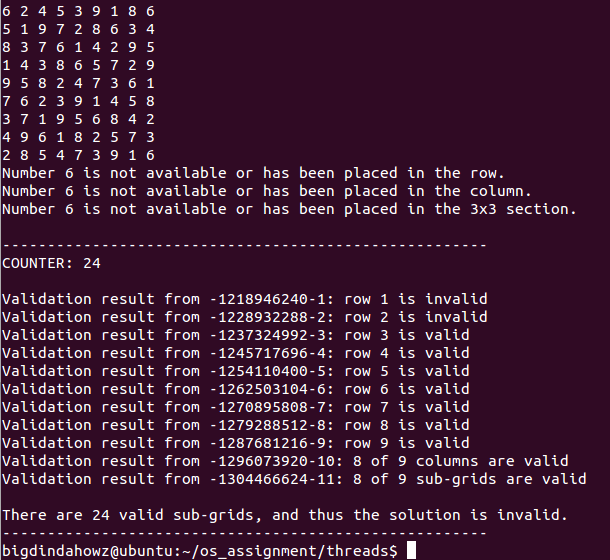
When evaluating the given Sudoku graph, all rows are shown as valid but columns and sub-grids are shown as invalid. Using this test case, this proves to show that accessing buffer2 and counter to store the various values for a region to be valid and the number of sub-regions that are valid are properly implemented. One process can ever only access its critical section to modify buffer2 and counter at a time.

Description of any cases for which your program is not working correctly or how you test your program that make you believe it works perfectly.

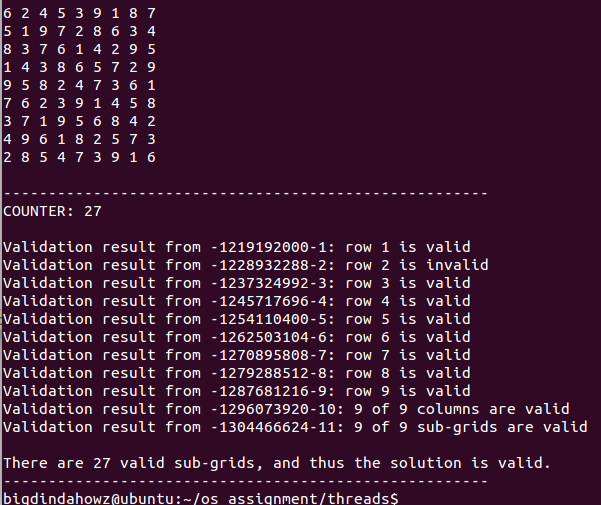
# Threads

There are a few cases in my program which are not working correctly or implemented the way I wanted it to be.

First off, the second thread that maps to the second row to validate is faulty. The thread, although it is created, and it executes its allocated function, when I print out all the validation statements at the end, it seems to have somehow altered the validity of the second row to being invalid. I have tested with a second row to be valid but it seems like the thread somehow changes the validity value to being invalid. All the other threads are fine except for the thread that maps to the second row. I know that the calculations are correct for this row because I have printed out a counter on a bad input Sudoku text file and it has printed out the number of valid sub grids the whole Sudoku puzzle has and it prints as 24/27 which is correct if one slot changes to an invalid number.



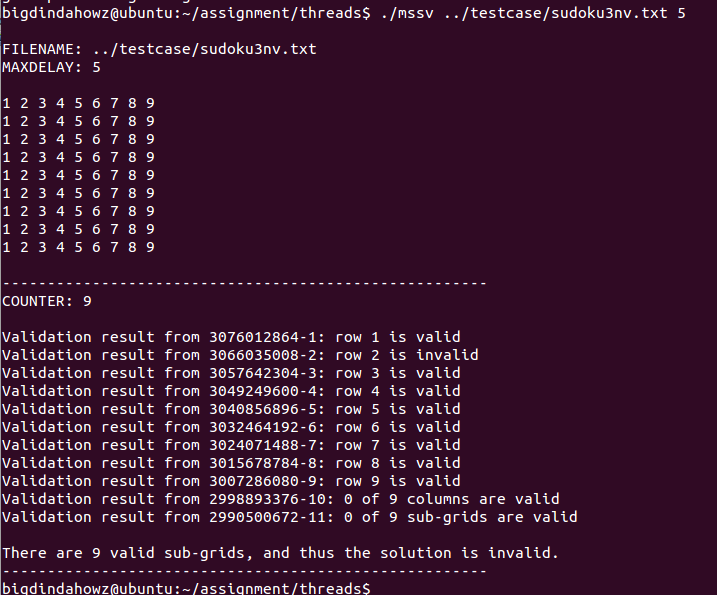
Shown above is the input of an invalid Sudoku file with the index at row 1, column 9 being a 6 which is an invalid input. The total count of sub-grids that are invalid is 24 and notice that row 2 is marked as invalid where the row is not invalid. Row 1 is marked invalid which is correct but a bug is still evident in the program for row 2 to be displaying its invalidity. This is an underlying problem that is still in my program which I find quite strange. Trying to fix this problem led to several changes but in the end, no luck with dealing with this thread. This is a minor issue, but in all cases, all calculations are valid and the output from the calculations have mapped to how I suspect the output to be.



Notice that in a completely valid puzzle, the second row is still outputting “invalid” even though the calculations computed that the puzzle was completely valid. This is the bug in the thread that somehow alters the validity of the second row but is still able to increment the counter on the number of valid sub-grids and can calculate the validity of the second row.

Secondly, writing to my log file isn’t properly working. The format of the log file outputs maximum 9 statements of where a column in invalid and a maximum of 9 invalid 3x3 subgrid validations. My rows output a statement per thread that is evaluating each row. So therefore, the maximum number of outputs written to my log file would be 27 to map the 27 sub regions that are to be evaluated. In the specification, it said to group the columns and output it as one statement which goes the same with the sub-grids. I didn’t have enough time to be able to implement this but by outputting each column and sub-grid as I did in the current log file, you can clearly see what sub-regions are invalid.

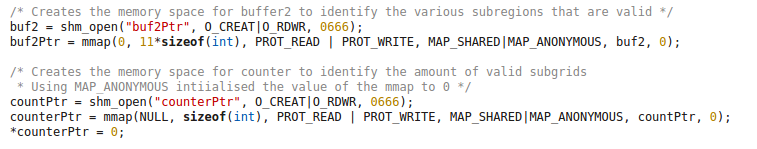
Going back to the previous image of the successful and unsuccessful sub-regions that were evaluated. This test convinced me that the program works how it is supposed to work. By providing an invalid file, the program can distinguish between what Sudoku puzzle is valid and invalid through the use of the counter and the results of the various sub-regions each thread had been assigned to. Reading in the Sudoku puzzle gave a series of numbers which was printed out to the command line before any threads were allocated to their assigned regions which determined the puzzle that the program was dealing with. This assures that the puzzle and the regions the functions deal with correspond to the puzzle output to the terminal. Therefore manually, you are able to distinguish which rows are invalid and valid and how many sub-regions in total should be valid to be output to the screen.



Looking at the image above, all nine rows are valid but it shows that row 2 is invalid. This is the bug where it mysteriously changes the value of my buffer2 value for that row to 0 which identifies that the row is invalid which causes the printValidation() function to output the row to be invalid. This is the problem that I am having with one of my threads.

# Processes

A problem that I may have come across when dealing with processes and shared memory is memory mapping.



When I open a shared memory space for buf2 and use the file descriptor that returns from opening with the mmap() function, buf2Ptr is now pointing to a memory space at a particular address. This goes the same for countPtr and counterPtr when mapping a particular memory space for the counter. Accessing the pointer to the shared memory space by the use of the dereference (\*) operand might be altering what the pointer now currently points to. An example being:

SHARED MEMORY

COUNTERPTR

SHARED MEMORY

COUNTERPTR

Int = 0

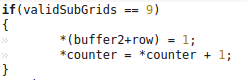
Notice that the counterPtr is now pointing to int = 0 after giving it a shared memory space by dereferencing the counterPtr. This might be leading to the program not using the shared memory that has been allocated through mmap(). Although, I do feel that I am using the memory space because if only one value is being altered when a single process runs through its critical section, then for buffer2, it would hold 11 values to map to the 11 regions that have been validated. The parent process is able to identify the eleven values from all eleven children updating the pointer. Usually when a child accesses and mutates a particular value, it is treated as a local variable that is altered, it won’t be able to be exported to the parent process and will only be exclusive to the child process. So even if I did or did not do the memory mapping correctly, what I intended to do is shown below; creating an integer in the memory space provided.

SHARED MEMORY

COUNTERPTR

Int = 0

Buffer2 however, when I memory mapped it, I did not initialize any values and used the space that was created via the parameter “11\*sizeof(int)” which gave me the space of 11 integer values.



When altering the value at a particular index, I move the pointer at which buffer2 points to which is max at index 10 from the start of where buffer2 points to and alter the values there which complies with the amount of sub-regions which were valid per validation method. (i.e. indexes 0-8 would hold 1 if those rows are valid, 9-10 would hold 9 or otherwise another number between 0 and 8 to dictate how many regions were valid). Here, I may have used the memory mapped to me but in counter, I may not have.

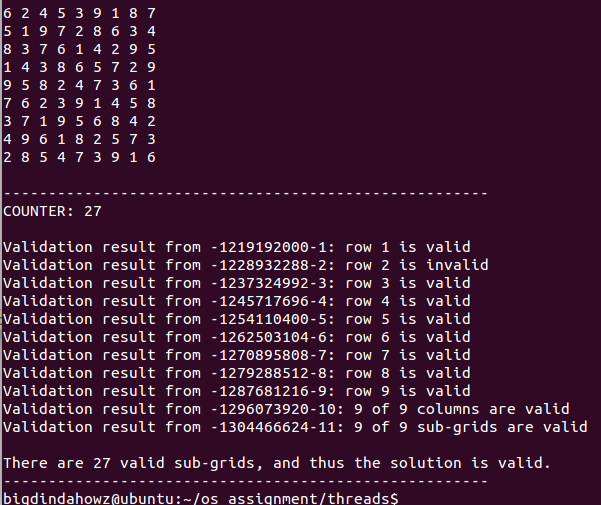
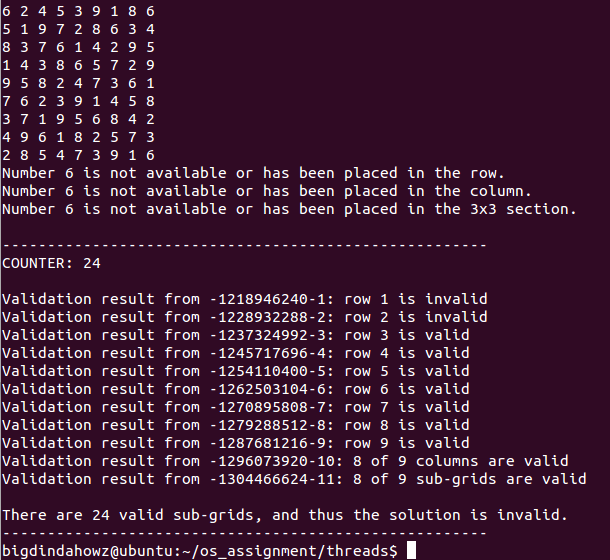
This is one or if not, the biggest problem I may or may not have in my program depending on how the program is interpreted. Explained above of how I visualize the counterPtr, it may seem invalid, but I feel I correctly implemented it for buffer2.

The implementation of the validations worked and defines if each row, column or three by three grid is valid shown through the altering of the shared memory values which later then is output to the command line in statements. I believe that my validation works for validating Sudoku puzzles from the results I get when testing with various valid and invalid inputs of Sudoku grids as shown above and in “Sample inputs and outputs from your running programs” section of this report.

Another problem is regarding my writing to the log file. The format of the statements written to the log file isn’t the format that was given in the specifications completely. I have a maximum of 9 columns and 9 3x3 sub-grid statements if all those sub-regions are invalid. I didn’t have enough time to be able to implement this so I left the formatting as it is. I find that the output of how my program currently writes to the log file is clear and concise and is able to show clearly which regions are invalid.

Sample inputs and outputs from your running programs.

# Threads



# 

# Processes

