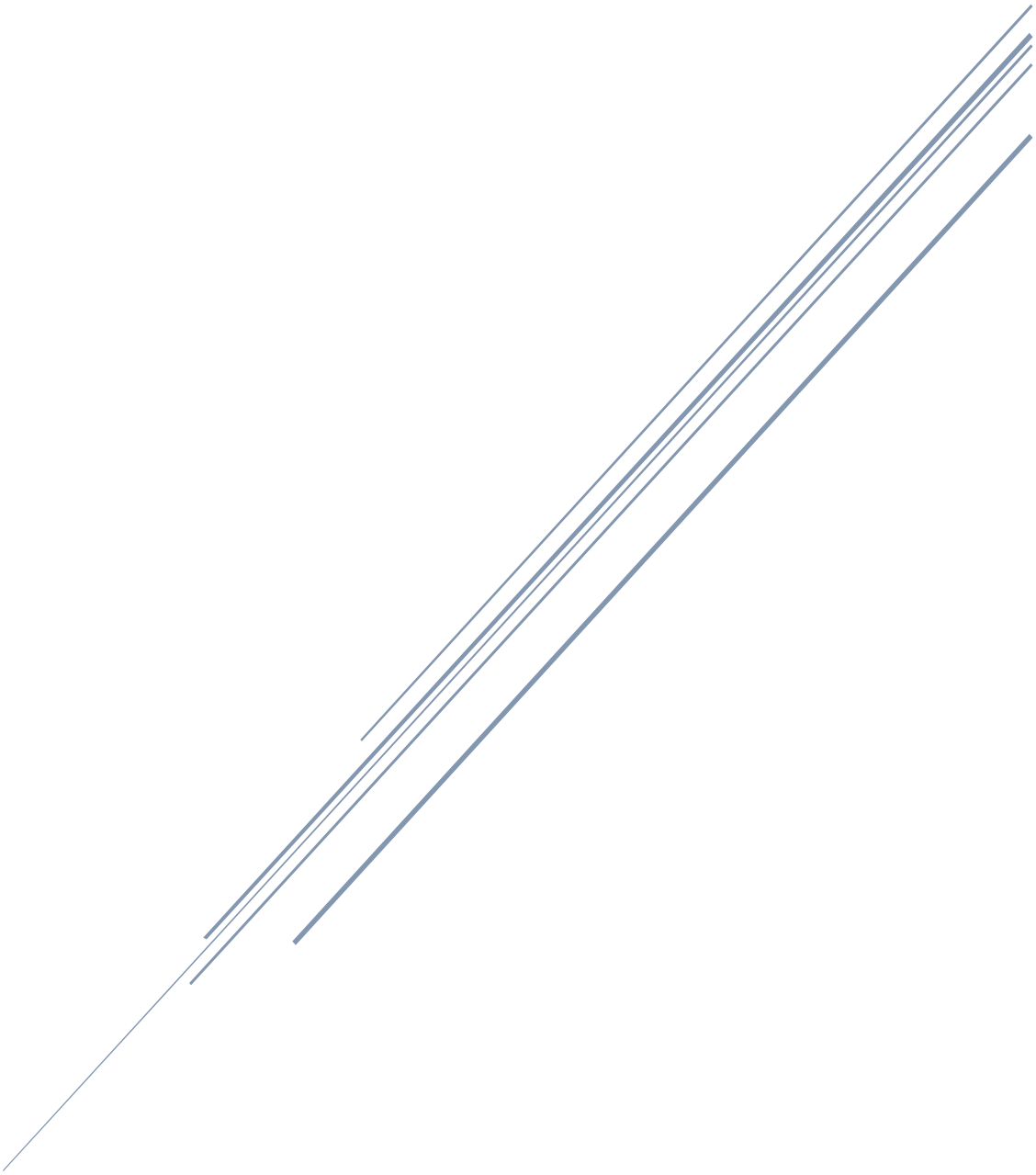
OPERATING SYSTEMS ASSIGNMENT

Jeremy Ciccarelli – 18760376 – 08/05/2017



COMP2006
Curtin University

How is mutual exclusion achieved and what processes/threads access shared resources?

Processes

In the implementation of MSSV through the use of processes, mutual exclusion is achieved through the use of semaphores. Specifically, these semaphores are binary in implementation, with four used in total. When the sem_init() function is called on these semaphores in the createSemaphores() submodule, the second argument is 1, indicating that these semaphores are to be shared between processes.

counterLock

This lock is used to prevent producer processes (e.g. ones performing calculations on subsections of the grid) from incrementing the counter variable in shared memory, and that when the consumer process reads from this piece of shared memory to determine whether the final number of subgrids is 27 (and thus whether the solution is valid or not), no other producer process can edit this value. The counterLock is released by signaling at the end of the block incrementing/accessing the counter.

Because the counterLock is required to update the counter, which takes place just beforeupdating and passing the result struct in shared memory through to the consumer process (via the sem_post() call), the counterLock is also required in order to update the result struct.

writeLock

The writeLock is used to ensure that only one process can write to the Logfile at a time, with a process waiting until the lock is acquired to write to the file, and releasing it upon finishing such writing. Without this, multiple processes may attempt to write to the Logfile at one time, which could result in overlapping strings.

empty and full

These semaphores are used by the consumer and producer processes to indicate to one another in the same manner as the typical producer-consumer scenario requires, using a buffer size of 1. Once a process has completed the calculation component (i.e. determined whether its assigned sub-region is valid), it calls sem_wait() on the empty semaphore. After acquiring the mutex on the counter, it then posts (signals) the full semaphore, indicating that the buffer is ‘full’.

Conversely, the consumer function waits on the full semaphore to be signaled by one of the producer functions, at which point it will acquire the counterLock, perform all required actions and then call sem_post() on the empty semaphore.

Shared memory

The 5 bits of shared memory are for buffer1 (the sudoku grid), buffer2 (a results array), counter (the number of valid subgrids), sems (the semaphores used for this program) and result (which stores the information/results from each process to be passed to the consumer). Buffer1 is accessed firstly by the readFile function which fills the array, although this takes place before the separate processes are created, and then further accessed by all producer processes which are validating the elements within that 2D array. However, as they are only reading from the grid, rather than editing any elements, no mutex locks are required to do so. Similarly with buffer2, no locks are needed to access the shared memory, as the elements are updated into specified locations which would not be impacted if multiple processes were updating it at the same time.

Threads

With respect to the second program, which utilizes multiple threads, mutual exclusion is achieved through the use of a pthread_mutex data type variable, which is used for enforcing mutual exclusion, as well as two pthread_cond_t data types for empty and full. These locks effectively work in similar fashion to the counter/write locks, empty and full (in respective order) utilized by the process implementation.

Producer threads will utilize the pthread_cond_wait() function on empty, which essentially performs waiting until the consumer functions signals (via pthread_cond_signal()) on the empty pthread condition, at which point they will acquire the mutex lock and pass through the relevant information.

Likewise, the consumer thread waits until a producer thread signals full and then retrieves the information stored in the global Result struct.

Shared memory

As the threads all access the same memory within the program, there is no need for shared memory to be used with this implementation of MSSV.

Additional Notes

Global variables were able to be used safely in both programs, due to them either being used as constants or being protected by the semaphores/mutex locks which prevented synchronization issues/race conditions from occurring. It simplified the passing of variables in and out of functions, which makes the design of the program easier to understand.

As per the assignment specification, an array of integers representing the validity of each process/thread is implemented and updated throughout the programs as part of the producer functions. However, it was easier to pass through the information to the consumer process through the Result struct, which contains information such as the process/thread ID, the outcome and number of rows/columns/subgrids valid. As such, shared memory was allocated for this struct and it was subject to mutex lock constraints.

TESTING

I have tested the program with both valid and invalid Sudoku solutions and the program is able to successfully evaluate the validity of the test cases.

Assumptions of the test cases:

The test files read in to the program were assumed to be in the format meeting the specification of "Each solution is in a file, formatted as 9 lines, and for each line each integer is separated by one space."

Thus, the program assumes there are no non-numeric characters contained within the file and that it exists as a 9 x 9 matrix where each element is a digit between 1 and 9.

An example of such is:

6 2 4 5 3 9 1 8 7
5 1 9 7 2 8 6 3 4
8 3 7 6 1 4 2 9 5
1 4 3 8 6 5 7 2 9
9 5 8 2 4 7 3 6 1
7 6 2 3 9 1 4 5 8
3 7 1 9 5 6 8 4 2
4 9 6 1 8 2 5 7 3
2 8 5 4 7 3 9 1 6

Sample inputs and outputs:

Below are the input files and terminal outputs + log files for the two different programs (using processes and threads respectively). The test cases covered are:

- Valid solution
- Matrix of all 0's (all elements invalid due to violating range of 1-9 constraint)
- Matrix of all 1's (all subgrids invalid due to violating unique occurrence constraint)
- Partial validity (approximately half-correct)
- Minimal validity (1 region of each type correct)

Multi-process Sudoku Solution Validator

Valid Solution

Input

```
6 2 4 5 3 9 1 8 7
5 1 9 7 2 8 6 3 4
8 3 7 6 1 4 2 9 5
1 4 3 8 6 5 7 2 9
9 5 8 2 4 7 3 6 1
7 6 2 3 9 1 4 5 8
3 7 1 9 5 6 8 4 2
4 9 6 1 8 2 5 7 3
2 8 5 4 7 3 9 1 6
```

Output

```
Validation result from process ID-18136: row 1 is valid
Validation result from process ID-18864: row 2 is valid
Validation result from process ID-9504: row 3 is valid
Validation result from process ID-10460: row 4 is valid
Validation result from process ID-9036: row 5 is valid
Validation result from process ID-13884: row 6 is valid
Validation result from process ID-15888: row 7 is valid
Validation result from process ID-8272: row 8 is valid
Validation result from process ID-2928: row 9 is valid
Validation result from process ID-16748: 9 of 9 columns are valid
Validation result from process ID-19056: 9 of 9 sub-grids are valid
There are 27 valid subgrids, and thus the solution is valid.
```

Logfile

Matrix of all 0's

Input

```
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
```

Output

```
Validation result from process ID-4128: row 1 is invalid
Validation result from process ID-16936: row 2 is invalid
Validation result from process ID-3084: row 3 is invalid
Validation result from process ID-16244: row 4 is invalid
Validation result from process ID-5992: row 5 is invalid
Validation result from process ID-3628: row 6 is invalid
Validation result from process ID-2924: row 7 is invalid
Validation result from process ID-4356: row 8 is invalid
Validation result from process ID-13076: row 9 is invalid
Validation result from process ID-14180: 0 of 9 columns are valid
Validation result from process ID-6416: 0 of 9 sub-grids are valid
There are 0 valid subgrids, and thus the solution is invalid.
```

Logfile

```
process ID-4128: row 1 is invalid
process ID-16936: row 2 is invalid
process ID-3084: row 3 is invalid
process ID-16244: row 4 is invalid
process ID-5992: row 5 is invalid
process ID-3628: row 6 is invalid
process ID-2924: row 7 is invalid
process ID-4356: row 8 is invalid
process ID-13076: row 9 is invalid
process ID-14180: column 1 is invalid
process ID-14180: column 2 is invalid
process ID-14180: column 3 is invalid
process ID-14180: column 4 is invalid
process ID-14180: column 5 is invalid
process ID-14180: column 6 is invalid
process ID-14180: column 7 is invalid
process ID-14180: column 8 is invalid
process ID-14180: column 9 is invalid
process ID-6416: subgrid [1..3, 1..3] is invalid
process ID-6416: subgrid [1..3, 4..6] is invalid
process ID-6416: subgrid [1..3, 7..9] is invalid
process ID-6416: subgrid [4..6, 1..3] is invalid
process ID-6416: subgrid [4..6, 4..6] is invalid
process ID-6416: subgrid [4..6, 7..9] is invalid
process ID-6416: subgrid [7..9, 1..3] is invalid
process ID-6416: subgrid [7..9, 4..6] is invalid
process ID-6416: subgrid [7..9, 7..9] is invalid
```

Matrix of all 1's

Input

```
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
```

Output

```
Validation result from process ID-12812: row 1 is invalid
Validation result from process ID-3220: row 2 is invalid
Validation result from process ID-5740: row 3 is invalid
Validation result from process ID-17040: row 4 is invalid
Validation result from process ID-10124: row 5 is invalid
Validation result from process ID-7724: row 6 is invalid
Validation result from process ID-17980: row 7 is invalid
Validation result from process ID-15408: row 8 is invalid
Validation result from process ID-11812: row 9 is invalid
Validation result from process ID-10296: 0 of 9 columns are valid
Validation result from process ID-17836: 0 of 9 sub-grids are valid
There are 0 valid subgrids, and thus the solution is invalid.
```

Logfile

```
process ID-12812: row 1 is invalid
process ID-3220: row 2 is invalid
process ID-5740: row 3 is invalid
process ID-17040: row 4 is invalid
process ID-10124: row 5 is invalid
process ID-7724: row 6 is invalid
process ID-17980: row 7 is invalid
process ID-15408: row 8 is invalid
process ID-11812: row 9 is invalid
process ID-10296: column 1 is invalid
process ID-10296: column 2 is invalid
process ID-10296: column 3 is invalid
process ID-10296: column 4 is invalid
process ID-10296: column 5 is invalid
process ID-10296: column 6 is invalid
process ID-10296: column 7 is invalid
process ID-10296: column 8 is invalid
process ID-10296: column 9 is invalid
process ID-17836: subgrid [1..3, 1..3] is invalid
process ID-17836: subgrid [1..3, 4..6] is invalid
process ID-17836: subgrid [1..3, 7..9] is invalid
process ID-17836: subgrid [4..6, 1..3] is invalid
process ID-17836: subgrid [4..6, 4..6] is invalid
process ID-17836: subgrid [4..6, 7..9] is invalid
process ID-17836: subgrid [7..9, 1..3] is invalid
process ID-17836: subgrid [7..9, 4..6] is invalid
process ID-17836: subgrid [7..9, 7..9] is invalid
```

Partial Validity

Input

```
6 2 4 5 3 9 1 8 7
5 1 9 7 2 1 6 3 4
8 3 7 6 1 4 2 9 5
1 4 5 8 6 5 2 2 9
9 5 8 2 4 7 3 6 1
7 3 2 3 9 1 4 5 8
3 7 1 9 5 5 8 4 2
4 9 6 1 8 2 5 7 3
2 8 5 4 7 3 9 1 6
```

Output

```
Validation result from process ID-7608: row 1 is valid
Validation result from process ID-3176: row 2 is invalid
Validation result from process ID-18160: row 3 is valid
Validation result from process ID-5124: row 4 is invalid
Validation result from process ID-14016: row 5 is valid
Validation result from process ID-11764: row 6 is invalid
Validation result from process ID-14564: row 7 is invalid
Validation result from process ID-1680: row 8 is valid
Validation result from process ID-18708: row 9 is valid
Validation result from process ID-11284: 5 of 9 columns are valid
Validation result from process ID-14840: 5 of 9 sub-grids are valid
There are 15 valid subgrids, and thus the solution is invalid.
```

Logfile

```
process ID-3176: row 2 is invalid
process ID-5124: row 4 is invalid
process ID-11764: row 6 is invalid
process ID-14564: row 7 is invalid
process ID-11284: column 2 is invalid
process ID-11284: column 3 is invalid
process ID-11284: column 6 is invalid
process ID-11284: column 7 is invalid
process ID-14840: subgrid [1..3, 4..6] is invalid
process ID-14840: subgrid [4..6, 1..3] is invalid
process ID-14840: subgrid [4..6, 7..9] is invalid
process ID-14840: subgrid [7..9, 4..6] is invalid
```

Minimal Validity

Input

```
6 2 4 5 3 9 1 8 7
5 1 9 1 1 1 1 1 1
8 3 7 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
9 1 1 1 1 1 1 1 1
7 1 1 1 1 1 1 1 1
3 1 1 1 1 1 1 1 1
4 1 1 1 1 1 1 1 1
2 1 1 1 1 1 1 1 1
```

Output

```
Validation result from process ID-7856: row 1 is valid
Validation result from process ID-18240: row 2 is invalid
Validation result from process ID-19348: row 3 is invalid
Validation result from process ID-7632: row 4 is invalid
Validation result from process ID-15136: row 5 is invalid
Validation result from process ID-8864: row 6 is invalid
Validation result from process ID-1868: row 7 is invalid
Validation result from process ID-14296: row 8 is invalid
Validation result from process ID-5444: row 9 is invalid
Validation result from process ID-17968: 1 of 9 columns are valid
Validation result from process ID-2188: 1 of 9 sub-grids are valid
There are 3 valid subgrids, and thus the solution is invalid.
```

Logfile

```
process ID-18240: row 2 is invalid
process ID-19348: row 3 is invalid
process ID-7632: row 4 is invalid
process ID-15136: row 5 is invalid
process ID-8864: row 6 is invalid
process ID-1868: row 7 is invalid
process ID-14296: row 8 is invalid
process ID-5444: row 9 is invalid
process ID-17968: column 2 is invalid
process ID-17968: column 3 is invalid
process ID-17968: column 4 is invalid
process ID-17968: column 5 is invalid
process ID-17968: column 6 is invalid
process ID-17968: column 7 is invalid
process ID-17968: column 8 is invalid
process ID-17968: column 9 is invalid
process ID-2188: subgrid [1..3, 4..6] is invalid
process ID-2188: subgrid [1..3, 7..9] is invalid
process ID-2188: subgrid [4..6, 1..3] is invalid
process ID-2188: subgrid [4..6, 4..6] is invalid
process ID-2188: subgrid [4..6, 7..9] is invalid
process ID-2188: subgrid [7..9, 1..3] is invalid
process ID-2188: subgrid [7..9, 4..6] is invalid
process ID-2188: subgrid [7..9, 7..9] is invalid
```

Multi-Thread Sudoku Solution Validator

Valid Solution

Input

```
6 2 4 5 3 9 1 8 7
5 1 9 7 2 8 6 3 4
8 3 7 6 1 4 2 9 5
1 4 3 8 6 5 7 2 9
9 5 8 2 4 7 3 6 1
7 6 2 3 9 1 4 5 8
3 7 1 9 5 6 8 4 2
4 9 6 1 8 2 5 7 3
2 8 5 4 7 3 9 1 6
```

Output

```
Validation result from thread ID-25770036704: row 5 is valid
Validation result from thread ID-25770036128: row 3 is valid
Validation result from thread ID-25770036992: row 6 is valid
Validation result from thread ID-25770035552: row 1 is valid
Validation result from thread ID-25770035840: row 2 is valid
Validation result from thread ID-25770036416: row 4 is valid
Validation result from thread ID-25770037280: row 7 is valid
Validation result from thread ID-25770038112: 9 of 9 columns are valid
Validation result from thread ID-25770037568: row 8 is valid
Validation result from thread ID-25770037856: row 9 is valid
Validation result from thread ID-25770038368: 9 of 9 sub-grids are valid
There are 27 valid subgrids, and thus the solution is valid.
```

Logfile

Matrix of all 0's

Input

```
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0
```

Output

```
Validation result from thread ID-25770035552: row 1 is invalid
Validation result from thread ID-25770037280: row 7 is invalid
Validation result from thread ID-25770036704: row 5 is invalid
Validation result from thread ID-25770035840: row 2 is invalid
Validation result from thread ID-25770036128: row 3 is invalid
Validation result from thread ID-25770036992: row 6 is invalid
Validation result from thread ID-25770036416: row 4 is invalid
Validation result from thread ID-25770037856: row 9 is invalid
Validation result from thread ID-25770038368: 0 of 9 sub-grids are valid
Validation result from thread ID-25770037568: row 8 is invalid
Validation result from thread ID-25770038112: 0 of 9 columns are valid
There are 0 valid subgrids, and thus the solution is invalid.
```

Logfile

```
thread ID-25770035552: row 1 is invalid
thread ID-25770037280: row 7 is invalid
thread ID-25770036704: row 5 is invalid
thread ID-25770035840: row 2 is invalid
thread ID-25770036128: row 3 is invalid
thread ID-25770036992: row 6 is invalid
thread ID-25770036416: row 4 is invalid
thread ID-25770037856: row 9 is invalid
thread ID-25770038368: subgrid [1..3, 1..3] is invalid
thread ID-25770038368: subgrid [1..3, 4..6] is invalid
thread ID-25770038368: subgrid [1..3, 7..9] is invalid
thread ID-25770038368: subgrid [4..6, 1..3] is invalid
thread ID-25770038368: subgrid [4..6, 4..6] is invalid
thread ID-25770038368: subgrid [4..6, 7..9] is invalid
thread ID-25770038368: subgrid [7..9, 1..3] is invalid
thread ID-25770038368: subgrid [7..9, 4..6] is invalid
thread ID-25770038368: subgrid [7..9, 7..9] is invalid
thread ID-25770037568: row 8 is invalid
thread ID-25770038112: column 1 is invalid
thread ID-25770038112: column 2 is invalid
thread ID-25770038112: column 3 is invalid
thread ID-25770038112: column 4 is invalid
thread ID-25770038112: column 5 is invalid
thread ID-25770038112: column 6 is invalid
thread ID-25770038112: column 7 is invalid
thread ID-25770038112: column 8 is invalid
thread ID-25770038112: column 9 is invalid
```

Matrix of all 1's

Input

```
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
```

Output

```
Validation result from thread ID-25770036128: row 3 is invalid
Validation result from thread ID-25770035552: row 1 is invalid
Validation result from thread ID-25770035840: row 2 is invalid
Validation result from thread ID-25770036704: row 5 is invalid
Validation result from thread ID-25770036416: row 4 is invalid
Validation result from thread ID-25770037856: row 9 is invalid
Validation result from thread ID-25770037280: row 7 is invalid
Validation result from thread ID-25770037568: row 8 is invalid
Validation result from thread ID-25770036992: row 6 is invalid
Validation result from thread ID-25770038368: 0 of 9 sub-grids are valid
Validation result from thread ID-25770038112: 0 of 9 columns are valid
There are 0 valid subgrids, and thus the solution is invalid.
```

Logfile

```
thread ID-25770036128: row 3 is invalid
thread ID-25770035552: row 1 is invalid
thread ID-25770035840: row 2 is invalid
thread ID-25770036704: row 5 is invalid
thread ID-25770036416: row 4 is invalid
thread ID-25770037856: row 9 is invalid
thread ID-25770037280: row 7 is invalid
thread ID-25770037568: row 8 is invalid
thread ID-25770036992: row 6 is invalid
thread ID-25770038368: subgrid [1..3, 1..3] is invalid
thread ID-25770038368: subgrid [1..3, 4..6] is invalid
thread ID-25770038368: subgrid [1..3, 7..9] is invalid
thread ID-25770038368: subgrid [4..6, 1..3] is invalid
thread ID-25770038368: subgrid [4..6, 4..6] is invalid
thread ID-25770038368: subgrid [4..6, 7..9] is invalid
thread ID-25770038368: subgrid [7..9, 1..3] is invalid
thread ID-25770038368: subgrid [7..9, 4..6] is invalid
thread ID-25770038368: subgrid [7..9, 7..9] is invalid
thread ID-25770038112: column 1 is invalid
thread ID-25770038112: column 2 is invalid
thread ID-25770038112: column 3 is invalid
thread ID-25770038112: column 4 is invalid
thread ID-25770038112: column 5 is invalid
thread ID-25770038112: column 6 is invalid
thread ID-25770038112: column 7 is invalid
thread ID-25770038112: column 8 is invalid
thread ID-25770038112: column 9 is invalid
```

Partial Validity

Input

```
6 2 4 5 3 9 1 8 7
5 1 9 7 2 1 6 3 4
8 3 7 6 1 4 2 9 5
1 4 5 8 6 5 2 2 9
9 5 8 2 4 7 3 6 1
7 3 2 3 9 1 4 5 8
3 7 1 9 5 5 8 4 2
4 9 6 1 8 2 5 7 3
2 8 5 4 7 3 9 1 6
```

Output

```
Validation result from thread ID-25770036416: row 4 is invalid
Validation result from thread ID-25770035840: row 2 is invalid
Validation result from thread ID-25770035552: row 1 is valid
Validation result from thread ID-25770036128: row 3 is valid
Validation result from thread ID-25770038112: 5 of 9 columns are valid
Validation result from thread ID-25770038368: 5 of 9 sub-grids are valid
Validation result from thread ID-25770036704: row 5 is valid
Validation result from thread ID-25770037280: row 7 is invalid
Validation result from thread ID-25770037856: row 9 is valid
Validation result from thread ID-25770037568: row 8 is valid
Validation result from thread ID-25770036992: row 6 is invalid
There are 15 valid subgrids, and thus the solution is invalid.
```

Logfile

```
thread ID-25770036416: row 4 is invalid
thread ID-25770035840: row 2 is invalid
thread ID-25770038112: column 2 is invalid
thread ID-25770038112: column 3 is invalid
thread ID-25770038112: column 6 is invalid
thread ID-25770038112: column 7 is invalid
thread ID-25770038368: subgrid [1..3, 4..6] is invalid
thread ID-25770038368: subgrid [4..6, 1..3] is invalid
thread ID-25770038368: subgrid [4..6, 7..9] is invalid
thread ID-25770038368: subgrid [7..9, 4..6] is invalid
thread ID-25770037280: row 7 is invalid
thread ID-25770036992: row 6 is invalid
```

Minimal Validity

Input

```
6 2 4 5 3 9 1 8 7
5 1 9 1 1 1 1 1 1
8 3 7 1 1 1 1 1 1
1 1 1 1 1 1 1 1 1
9 1 1 1 1 1 1 1 1
7 1 1 1 1 1 1 1 1
3 1 1 1 1 1 1 1 1
4 1 1 1 1 1 1 1 1
2 1 1 1 1 1 1 1 1
```

Output

```
Validation result from thread ID-25770035552: row 1 is valid
Validation result from thread ID-25770036416: row 4 is invalid
Validation result from thread ID-25770035840: row 2 is invalid
Validation result from thread ID-25770036128: row 3 is invalid
Validation result from thread ID-25770036704: row 5 is invalid
Validation result from thread ID-25770037568: row 8 is invalid
Validation result from thread ID-25770036992: row 6 is invalid
Validation result from thread ID-25770037280: row 7 is invalid
Validation result from thread ID-25770037856: row 9 is invalid
Validation result from thread ID-25770038112: 1 of 9 columns are valid
Validation result from thread ID-25770038368: 1 of 9 sub-grids are valid
There are 3 valid subgrids, and thus the solution is invalid.
```

Logfile

```
thread ID-25770036416: row 4 is invalid
thread ID-25770035840: row 2 is invalid
thread ID-25770036128: row 3 is invalid
thread ID-25770036704: row 5 is invalid
thread ID-25770037568: row 8 is invalid
thread ID-25770036992: row 6 is invalid
thread ID-25770037280: row 7 is invalid
thread ID-25770037856: row 9 is invalid
thread ID-25770038112: column 2 is invalid
thread ID-25770038112: column 3 is invalid
thread ID-25770038112: column 4 is invalid
thread ID-25770038112: column 5 is invalid
thread ID-25770038112: column 6 is invalid
thread ID-25770038112: column 7 is invalid
thread ID-25770038112: column 8 is invalid
thread ID-25770038112: column 9 is invalid
thread ID-25770038368: subgrid [1..3, 4..6] is invalid
thread ID-25770038368: subgrid [1..3, 7..9] is invalid
thread ID-25770038368: subgrid [4..6, 1..3] is invalid
thread ID-25770038368: subgrid [4..6, 4..6] is invalid
thread ID-25770038368: subgrid [4..6, 7..9] is invalid
thread ID-25770038368: subgrid [7..9, 1..3] is invalid
thread ID-25770038368: subgrid [7..9, 4..6] is invalid
thread ID-25770038368: subgrid [7..9, 7..9] is invalid
```

```
/*Name: Jeremy Ciccarelli
  Student ID: 18760376
  Program: Multi-process Sudoku Solution Validator
  Overview: This program reads in a provided sudoku solution in a textfile
            and determines whether or not it is valid, by creating 11 processes
            to validate the 27 separate subgrids of the solution (9 rows,
            9 columns, 9 3x3 subgrids). If all 27 subgrids are valid, then the
            solution will be deemed valid. If any errors are detected within
            the solution, they will be written to the logfile which is created
            at runtime.*/

/*IMPORTS*/
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <sys/shm.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/mman.h>
#include <unistd.h>
#include <sys/wait.h>
#include <semaphore.h>
#include <time.h>
#include "structs.h"


//DECLARATIONS
const int bufferSize1 = (sizeof(int) * 81);
const int bufferSize2 = (sizeof(int) * 11);
const int counterSize = (sizeof(int));
const int semsSize = (sizeof(Semaphores));
const int resultSize = (sizeof(Result));
const char* logfile = "logfile.txt";

int (*ptrBuffer1)[9];
int* ptrBuffer2;
int* ptrCounter;
Semaphores* sems;
Result* result;
FILE* writeFile;
Parameters rows[9];
int maxDelay;

/*FUNCTION: ReadFile
  OVERVIEW: This function performs the reading of the specified .txt file specified
            as a command line parameter, assigning it to the 2D array 'grid',
            ptrBuffer1, which is in shared memory.*/

int readFile(char* filename){
```

```
    /*Initialise the file pointer 'readFile'*/
    int i = 0;
    int j = 0;
    int k = 0;
    int retVal =0;
    FILE* readFile;

    readFile = fopen(filename,"r");
    if(readFile == NULL){
        printf("Error reading file");
    }
    else{
        while(!feof(readFile) && i<9){
            fscanf(readFile, "%1d ",&ptrBuffer1[i][j]);
            k++;
            j++;
            if(j>8){
                i++;
                j=0;
            }
        }
        retVal = 1;
    }
    fclose(readFile);
    if(k != 81){
        retVal = 1;
    }
    return retVal;
}

/*FUNCTION: InitialiseSharedMem
  OVERVIEW: Initialises the values of all elements in ptrBuffer2 to be 0 and sets
            the initial value of the counter (number of valid subgrids) to be 0.*/

void initialiseSharedMem(){
    int i;
    for(i=0;i<11;i++){
        ptrBuffer2[i] = 0;
    }
    *ptrCounter = 0;
}

/*FUNCTION: CheckRow
  OVERVIEW: This function takes in an assigned row of a sudoku grid, checks to make
            sure it's a valid row (contains uniquely numbers 1-9) and passes
            the result (as a theoretical 'producer') to the consumer function
            upon requiring the counter lock. If a row is deemed invalid, it will
            be written to the logfile once the writeLock is acquired.*/

void checkRow(Parameters rc){
```

```
    int i,num,valid=0;
    int row = rc.row;
    int usedArray[9] = {0};
    for(i=0;i<9;i++){
        num = ptrBuffer1[row][i];
        if((num < 1)|| (num >9) || usedArray[num-1] !=0){
            valid =1;
        }
        else{
           usedArray[num-1] = 1;
        }
    }

    if(valid == 1){
        sem_wait(&sems->writeLock);
                writeFile = fopen(logfile,"a");
                fprintf(writeFile,"process ID-%d: row %d is
invalid\n",getpid(),row+1);
                fclose(writeFile);
        sem_post(&sems->writeLock);
    }

    sleep(rand() % maxDelay);

    ptrBuffer2[row] = 1-valid;
    sem_wait(&sems->empty);
        sem_wait(&sems->counterLock);
            result->row = row;
            result->pid = getpid();
            result->group = 1;
            if(valid == 0){
               result->val = "valid";
               *ptrCounter = *ptrCounter+1;
            }
            else{
                result->val = "invalid";
            }
        sem_post(&sems->counterLock);
    sem_post(&sems->full);

    _exit(0);
 }

/*FUNCTION: CheckColumns
  OVERVIEW: This function checks over each column in the sudoku grid, validating
            each one and maintaining a count (totalValid) which is then passed to
            the consumer process via the result struct. Additionally, any errors
            detected are written to the logfile once the writeLock is acquired.*/

void checkColumns(){
```

```
    int i,j,num,curValid,totalValid=0;
    for(i=0;i<9;i++){
        curValid = 0;
        int usedArray[9] = {0};
        for(j=0;j<9;j++){
            num = ptrBuffer1[j][i];
            if((num < 1)|| (num >9) || usedArray[num-1] !=0){
                curValid = 1;
            }
            else{
                usedArray[num-1] = 1;
            }
        }
        if(curValid == 0){
            totalValid++;
        }
        else{
            sem_wait(&sems->writeLock);
                    writeFile = fopen(logfile,"a");
                    fprintf(writeFile,"process ID-%d: column %d is
invalid\n",getpid(),i+1);
                    fclose(writeFile);
            sem_post(&sems->writeLock);
        }
    }

    sleep(rand() % maxDelay);

    ptrBuffer2[9] = totalValid;
    sem_wait(&sems->empty);
        sem_wait(&sems->counterLock);
            *ptrCounter = *ptrCounter+totalValid;
            result->pid = getpid();
            result->group = 2;
            result->totalValid = totalValid;
        sem_post(&sems->counterLock);
    sem_post(&sems->full);
    _exit(0);
}

/*FUNCTION: CheckSubGrids
  OVERVIEW: This function iterates through the 9 3x3 subgrids of the sudoku grid,
            validating each square. Maintains a count of valid subgrids and appends
            error messages detected onto a string which is written to the logfile
            once the writeLock is acquired and results are passed to the consumer
process.*/

void checkSubGrids(){
    int i,j,k,l,num,curValid,totalValid=0;
    for(i=0;i<9;i= i+3){
```

```
        for(j=0;j<9;j=j+3){
            //SUB GRID START
            curValid = 0;
            int usedArray[9] = {0};
            for(k=i;k<i+3;k++){
                for(l=j;l<j+3;l++){

                    num = ptrBuffer1[k][l];
                    if((num < 1)|| (num >9) || usedArray[num-1] !=0){
                        curValid = 1;
                    }
                    else{
                        usedArray[num-1] = 1;
                    }
                }
            }
            if(curValid == 0){
                totalValid++;
            }
            else{
                sem_wait(&sems->writeLock);
                            writeFile = fopen(logfile,"a");
                            fprintf(writeFile,"process ID-%d: subgrid [%d..%d, %d..%d]
is invalid\n",getpid(),i+1,i+3,j+1,j+3);
                            fclose(writeFile);
                sem_post(&sems->writeLock);
            }
        }
    }

    sleep(rand() % maxDelay);

    ptrBuffer2[10] = totalValid;
    sem_wait(&sems->empty);
        sem_wait(&sems->counterLock);
            *ptrCounter = *ptrCounter+totalValid;
            result->pid = getpid();
            result->group = 3;
            result->totalValid = totalValid;
        sem_post(&sems->counterLock);
    sem_post(&sems->full);
    _exit(0);
}

/*FUNCTION: Consumer
  OVERVIEW: This function performs 11 iterations (corresponding to the 11 calculation
            threads) of waiting for a result to be produced by one of these such
threads.
            It prints out the results of each validation, taken from the result
struct,
```

```
            and at the end of the 11th thread, it checks to see if the counter for
            the number of valid subgrids. If counter is equal to 27, it will indicate
            that the sudoku grid is valid, otherwise deem it invalid. */

void consumer(){
    int i;
    for(i =0;i<11;i++){
        sem_wait(&sems->full);
            sem_wait(&sems->counterLock);
                if(result->group == 1){
                    printf("Validation result from process ID-%d: row %d is
%s\n",result->pid,result->row+1,result->val);
                }
                else if(result->group==2){
                    printf("Validation result from process ID-%d: %d of 9 columns are
valid\n",result->pid,result->totalValid);
                }
                else if(result->group==3){
                    printf("Validation result from process ID-%d: %d of 9 sub-grids
are valid\n",result->pid,result->totalValid);
                }
            sem_post(&sems->counterLock);
        sem_post(&sems->empty);

        if(i==10){
            sem_wait(&sems->counterLock);
            if(*ptrCounter < 27){
                printf("There are %d valid subgrids, and thus the solution is
invalid.\n",*ptrCounter);
            }
            else{
                printf("There are %d valid subgrids, and thus the solution is
valid.\n",*ptrCounter);
            }
            sem_post(&sems->counterLock);
        }
    }
    _exit(0);
}

/*FUNCTION: CreateProcesses
  OVERVIEW: Creates 11 processes to perform the calculations on the assigned regions
            of the sudoku grid, and then calls the consumer function which reads
            in the produced results from these child processes.*/

int createProcesses(){
    int i,j=0,k;
    pid_t par = getpid();
    pid_t child = 0;
    //Ensures no zombie processes can occur
```

```
    signal(SIGCHLD,SIG_IGN);
    for(i = 0;i<9;i++){
        rows[i].column=0;
        rows[i].row=i;
    }

    for(i = 0; i < 11; i++){
        if(getpid() == par){
            k=i;
            child = fork();
            j++;
        }
    }

    if(child == 0 && j<10){
        checkRow(rows[k]);
    }
    else if(child == 0 && j==10)
    {
        checkColumns();
    }
    else if(child == 0 && j==11){
        checkSubGrids();
    }
    else{
        consumer();
    }
}

/*FUNCTION: CreateMemory
  OVERVIEW: This function takes in the pointer to the integer reference values
            (descriptors) for the shared memory allocation. It uses the shm_open,
            ftruncate and mmap functions to allocate and establish the shared
            memory for the child processes to access.
            The 5 bits of shared memory are for buffer1 (the sudoku grid), buffer2
            (a results array), counter (the number of valid subgrids), sems (the
            semaphores used for this program) and result (which stores the
            information/results from each process to be passed to the consumer).*/

void createMemory(Descriptors* desc){
    desc->buffer1 = shm_open("Buffer1", O_CREAT | O_RDWR, 0666);
    ftruncate(desc->buffer1, bufferSize1);
    ptrBuffer1 = (int (*)[9])mmap(NULL,bufferSize1, PROT_READ|PROT_WRITE,
MAP_SHARED|MAP_ANONYMOUS, desc->buffer1, 0);

    desc->buffer2 = shm_open("Buffer2", O_CREAT | O_RDWR, 0666);
    ftruncate(desc->buffer2, bufferSize2);
    ptrBuffer2 = (int*)mmap(NULL,bufferSize1, PROT_READ|PROT_WRITE,
MAP_SHARED|MAP_ANONYMOUS, desc->buffer2, 0);
```

```
    desc->counter = shm_open("Counter", O_CREAT | O_RDWR, 0666);
    ftruncate(desc->counter, counterSize);
    ptrCounter = (int*)mmap(NULL,bufferSize1, PROT_READ|PROT_WRITE,
MAP_SHARED|MAP_ANONYMOUS, desc->counter, 0);

    desc->sems = shm_open("Semaphores", O_CREAT | O_RDWR, 0666);
    ftruncate(desc->sems, semsSize);
    sems = (Semaphores*)mmap(NULL,semsSize, PROT_READ|PROT_WRITE,
MAP_SHARED|MAP_ANONYMOUS, desc->sems, 0);

    desc->result = shm_open("Result", O_CREAT | O_RDWR, 0666);
    ftruncate(desc->result, resultSize);
    result = (Result*)mmap(NULL,resultSize, PROT_READ|PROT_WRITE,
MAP_SHARED|MAP_ANONYMOUS, desc->result, 0);
}

/*FUNCTION: RemoveMemory
  OVERVIEW: This function removes the shared memory utilised by the program by
            unmapping and unlinking the memory.*/

void removeMemory(){
    munmap(ptrBuffer1, bufferSize1);
    munmap(ptrBuffer2, bufferSize2);
    munmap(ptrCounter, counterSize);
    munmap(sems, semsSize);
    munmap(result,resultSize);

    shm_unlink("Buffer1");
    shm_unlink("Buffer2");
    shm_unlink("Counter");
    shm_unlink("Semaphores");
    shm_unlink("Result");
}

/*FUNCTION: CreateSemaphores
  OVERVIEW: Initialises the 4 semaphores used in the program. writeLock is a mutex
            for access to writing to the logfile, counterLock is a mutex to update
            the counter and empty/full are used for signalling to and from the
            consumer and producer processes (as per the producer-consumer model).*/

void createSemaphores(){
    sem_init(&sems->writeLock,1,1);
    sem_init(&sems->counterLock,1,1);
    sem_init(&sems->empty,1,1);
    sem_init(&sems->full,1,0);
}

/*FUNCTION: RemoveSemaphores
  OVERVIEW: Destroys the semaphores pointed to by the given memory addresses once
            the program has finished using them (i.e. at the end of the consumer
```

```
            process).*/

void removeSemaphores(){
    sem_destroy(&sems->writeLock);
    sem_destroy(&sems->counterLock);
    sem_destroy(&sems->empty);
    sem_destroy(&sems->full);
}

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

    //Validate that the number of command line parameters is correct and that
    //the maxDelay specified is non-negative and non-zero
    if(argc != 3){
        return 1;
    }
    if(atoi(argv[2]) <= 0){
        printf("Delay cannot be negative or zero!\n");
        return 1;
    }

    //Seed the random number generator
    srand(time(NULL));
    maxDelay = atoi(argv[2]);

    //Wipe the logfile for clean writing of errors
    writeFile = fopen(logfile,"w");
    fclose(writeFile);

    //Create and initialise memory
    Descriptors desc;
    createMemory(&desc);
    createSemaphores();
    initialiseSharedMem();

    //Read in the textfile to store the sudoku grid
    readFile(argv[1]);

    //Create the processes, run the calculations and output the result
    createProcesses();

    //Clean up memory and semaphores
    removeMemory();
    removeSemaphores();

    //Close the logfile
    fclose(writeFile);
    return 0;
}
```

```
/*FILE NAME: structs.h
  OVERVIEW: Structs header to support the multi-process main.c file*/
typedef struct{
    int row;
    int column;
}Parameters;

typedef struct{
    int buffer1;
    int buffer2;
    int counter;
    int sems;
    int result;
}Descriptors;

typedef struct{
    sem_t writeLock;
    sem_t counterLock;
    sem_t empty;
    sem_t full;
}Semaphores;

typedef struct{
    int totalValid;
    int row;
    int pid;
    char* val;
    int group;
}Result;
```

```
/*Name: Jeremy Ciccarelli
  Student ID: 18760376
  Program: Multi-threaded Sudoku Solution Validator
  Overview: This program reads in a provided sudoku solution in a textfile
            and determines whether or not it is valid, by creating 11 threads
            to validate the 27 separate subgrids of the solution (9 rows,
            9 columns, 9 3x3 subgrids). If all 27 subgrids are valid, then the
            solution will be deemed valid. If any errors are detected within
            the solution, they will be written to the logfile which is created
            at runtime.*/

/*Imports*/
#include <stdlib.h>
#include <stdio.h>
#include <pthread.h>
#include <string.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <unistd.h>
#include <time.h>
#include <sys/types.h>
#include <sys/syscall.h>
#include "structs.h"


/*Declarations of variables*/
const char* logfile = "logfile.txt";

pthread_mutex_t mutex;
pthread_cond_t empty;
pthread_cond_t full;
FILE* writeFile;
Result* result;

int counter = 0;
int solutionValid[11] = {0};
int grid[9][9];
int buffer = 0;
int maxDelay;
Parameters** param;

/*FUNCTION: CheckRow
  OVERVIEW: This function takes in an assigned row of a sudoku grid, checks to make
            sure it's a valid row (contains uniquely numbers 1-9) and passes
            the result (as a theoretical 'producer') to the consumer function
            upon requiring the mutex lock. If a row is deemed invalid, it will
            be written to the logfile.*/

void* checkRow(void* parameter){
```

```
    Parameters* param = parameter;
    int i,num,valid=0;
    int row = param->row;

    int usedArray[9] = {0};
    for(i=0;i<9;i++){
        num = grid[row][i];
        if((num < 1)|| (num >9) || usedArray[num-1] !=0){
            valid =1;
        }
        else{
           usedArray[num-1] = 1;
        }
    }

    sleep(rand() % maxDelay);

    solutionValid[row] = 1-valid;
    //Acquire mutex
    pthread_mutex_lock(&mutex);
        //Wait until the buffer is empty
        while(buffer != 0){
                pthread_cond_wait(&empty,&mutex);
            }
            //Update the result struct with the row, thread ID etc.
            result->row = row;
            result->tid = pthread_self();
            result->group = 1;
            if(valid == 0){
               result->val = "valid";
               counter = counter+1;
            }
            else{
                //Error detected - write to logfile
                result->val = "invalid";
                writeFile = fopen(logfile,"a");
                fprintf(writeFile,"thread ID-%lu: row %d is
invalid\n",pthread_self(),row+1);
                fclose(writeFile);
            }
            buffer++;
        pthread_cond_signal(&full);
    //Release mutex
    pthread_mutex_unlock(&mutex);

    pthread_exit(0);
 }

/*FUNCTION: CheckColumns
```

```
 OVERVIEW: This function checks over each column in the sudoku grid, validating
           each one and maintaining a count (totalValid) which is then passed to
           the consumer process via the result struct. Additionally, any errors
           detected are written to the logfile once the mutex is acquired.*/

void* checkColumns(){
    int i,j,num,curValid,totalValid=0;
    char* log = (char*)calloc(9*257,sizeof(char));
    char* line = (char*)calloc(257,sizeof(char));
    for(i=0;i<9;i++){
        curValid = 0;
        int usedArray[9] = {0};
        for(j=0;j<9;j++){
            num = grid[j][i];
            if((num < 1)|| (num >9) || usedArray[num-1] !=0){
                curValid = 1;
            }
            else{
                usedArray[num-1] = 1;
            }
        }
        if(curValid == 0){
            totalValid++;
        }
        else{
            sprintf(line,"thread ID-%lu: column %d is invalid\n",pthread_self(),i+1);
            strncat(log,line,257);
        }
    }
    sleep(rand() % maxDelay);

    solutionValid[9] = totalValid;
    //Acquire mutex
    pthread_mutex_lock(&mutex);
        while(buffer != 0){
                pthread_cond_wait(&empty,&mutex);
            }
            counter = counter+totalValid;
            result->tid = pthread_self();
            result->group = 2;
            result->totalValid = totalValid;
            writeFile = fopen(logfile,"a");
            fputs(log,writeFile);
            fclose(writeFile);
            buffer++;
        pthread_cond_signal(&full);
    //Release mutex
    pthread_mutex_unlock(&mutex);

    free(log);
```

```
    free(line);
    pthread_exit(0);
}


/*FUNCTION: CheckSubGrids
  OVERVIEW: This function iterates through the 9 3x3 subgrids of the sudoku grid,
            validating each square. Maintains a count of valid subgrids and appends
            error messages detected onto a string which is written to the logfile
            once the mutex is acquired and results are passed to the consumer
thread.*/

void* checkSubGrids(){
    int i,j,k,l,num,curValid,totalValid=0;

    //To be written to logfile
    char* log = (char*)calloc(9*257,sizeof(char));
    char* line = (char*)calloc(257,sizeof(char));
    for(i=0;i<9;i= i+3){
        for(j=0;j<9;j=j+3){
            //SUB GRID START
            curValid = 0;
            int usedArray[9] = {0};
            for(k=i;k<i+3;k++){
                for(l=j;l<j+3;l++){
                    num = grid[k][l];
                    if((num < 1)|| (num >9) || usedArray[num-1] !=0){
                        curValid = 1;
                    }
                    else{
                        usedArray[num-1] = 1;
                    }
                }
            }
            if(curValid == 0){
                totalValid++;
            }
            else{
                sprintf(line,"thread ID-%lu: subgrid [%d..%d, %d..%d] is
invalid\n",pthread_self(),i+1,i+3,j+1,j+3);
                strncat(log,line,257);
            }
        }
    }
    sleep(rand() % maxDelay);

    solutionValid[10] = totalValid;
    //Acquire mutex
    pthread_mutex_lock(&mutex);
        while(buffer != 0){
```

```
                pthread_cond_wait(&empty,&mutex);
            }
            counter = counter+totalValid;
            result->tid = pthread_self();
            result->group = 3;
            result->totalValid = totalValid;

            //Write to log file
            writeFile = fopen(logfile,"a");
            fputs(log,writeFile);
            fclose(writeFile);

            buffer++;
        pthread_cond_signal(&full);
    //Release mutex
    pthread_mutex_unlock(&mutex);

    //Free memory
    free(log);
    free(line);
    //Terminate thread
    pthread_exit(0);
}
/*FUNCTION: Consumer
  OVERVIEW: This function performs 11 iterations (corresponding to the 11 calculation
            threads) of waiting for a result to be produced by one of these such
threads.
            It prints out the results of each validation, taken from the result
struct,
            and at the end of the 11th thread, it checks to see if the counter for
            the number of valid subgrids. If counter is equal to 27, it will indicate
            that the sudoku grid is valid, otherwise deem it invalid. */

void* consumer(){
    int i;
    for(i =0;i<11;i++){
        pthread_mutex_lock(&mutex);
            while(buffer == 0){
                pthread_cond_wait(&full,&mutex);
            }
                if(result->group == 1){
                    printf("Validation result from thread ID-%lu: row %d is
%s\n",result->tid,result->row+1,result->val);
                }
                else if(result->group==2){
                    printf("Validation result from thread ID-%lu: %d of 9 columns are
valid\n",result->tid,result->totalValid);
                }
                else if(result->group==3){
                    printf("Validation result from thread ID-%lu: %d of 9 sub-grids
```

```
are valid\n",result->tid,result->totalValid);
                }

                buffer--;
            pthread_cond_signal(&empty);
            pthread_mutex_unlock(&mutex);

        /*Once all threads have been read in by the consumer thread, check
          the value of the counter and print out the validity of the solution grid*/
        if(i==10){
            pthread_mutex_lock(&mutex);
            if(counter < 27){
                printf("There are %d valid subgrids, and thus the solution is
invalid.\n",counter);
            }
            else{
                printf("There are %d valid subgrids, and thus the solution is
valid.\n",counter);
            }
            pthread_mutex_unlock(&mutex);
        }
    }

    pthread_exit(0);
}

/*FUNCTION: CreateThreads
  OVERVIEW: Creates 11 threads to perform the calculations on the assigned regions
            of the sudoku grid, as well as a 'consumer' thread which reads
            in the produced results from these threads.
            pthread_join is used in order to suspend the execution of this thread
            (the calling one) whilst the other threads are still in execution. */

int createThreads(){
    pthread_t threads[11];
    pthread_t consumerThread;
    int i;
    //Create the 9 threads to validate the rows
    for(i=0;i<9;i++){
        param[i]->row = i;
        param[i]->column = 0;
        pthread_create(&threads[i],NULL,checkRow,param[i]);
    }
    //Create thread to validate the columns
    pthread_create(&threads[9],NULL,checkColumns,NULL);
    //Create thread to validate the subgrids
    pthread_create(&threads[10],NULL,checkSubGrids,NULL);
    //Create thread to read in all the results
    pthread_create(&consumerThread,NULL,consumer,NULL);
```

```
    //Suspend execution of this thread until all of the other threads have finished
    for(i = 0;i<11;i++){
        pthread_join(threads[i], NULL);
    }
    pthread_join(consumerThread,NULL);
    return 0;
}

/*FUNCTION: ReadFile
  OVERVIEW: This function performs the reading of the specified .txt file specified
            as a command line parameter, assigning it to the 2D array 'grid'.*/

int readFile(char* filename){
    int i = 0;
    int j = 0;
    int retVal =0;
    FILE* readFile;

    readFile = fopen(filename,"r");
    //Check for reading errors
    if(readFile == NULL){
        printf("Error reading file");
    }
    else{
        while(!feof(readFile) && i<9){
            fscanf(readFile, "%1d",&grid[i][j]);
            j++;
            if(j>8){
                i++;
                j=0;
            }
        }
        retVal = 1;
    }
    fclose(readFile);
    return retVal;
}

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

    /*Check number of command line parameters is correct*/
    if(argc != 3){
        return 1;
    }
    srand(time(NULL));
    maxDelay = atoi(argv[2]);

    result = (Result*)malloc(sizeof(Result));
    param = (Parameters**)malloc(9*sizeof(Parameters*));
    int i;
```

```
    for(i = 0;i<9;i++){
        param[i] = (Parameters*)malloc(sizeof(Parameters));
    }
    /*Read in the textfile representation of the sudoku grid*/
    readFile(argv[1]);

    /*Clear the logfile for writing of new errors (if applicable)*/
    writeFile = fopen(logfile,"w");
    fclose(writeFile);

    /*Initialise the mutex and thread condition variables*/
    pthread_mutex_init(&mutex, NULL );
    pthread_cond_init(&full, NULL );
    pthread_cond_init(&empty, NULL );

    /*Create the threads, which perform the necessary calculations for the program*/
    createThreads();

    /*Destroy the locks*/
    pthread_mutex_destroy(&mutex );
    pthread_cond_destroy(&full );
    pthread_cond_destroy(&empty );

    /*Free the memory and return successfully*/
    for(i = 0;i<9;i++){
        free(param[i]);
    }
    free(result);
    free(param);
    return 0;
}
```

```
/*FILE NAME: structs.h
  OVERVIEW: Structs header to support the multi-thread main.c file*/

typedef struct{
    int row;
    int column;
}Parameters;

typedef struct{
    int totalValid;
    int row;
    pthread_t tid;
    char* val;
    int group;
}Result;
```