CSC 345 Weekly Report for Project: 02

Full Name: Milian Ingco & Katrina Lucero

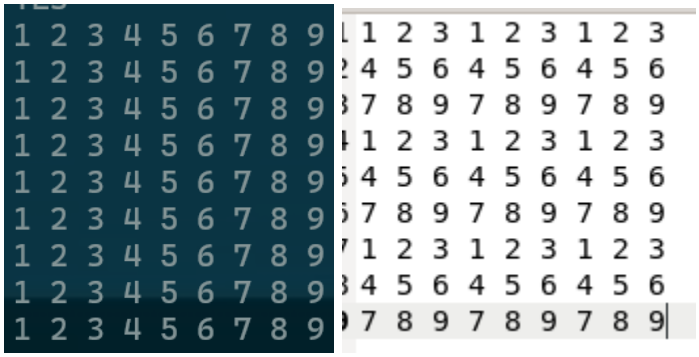
Section: 02

0. Version 1 Code

0.1. Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <stdio.h>
#include <string.h>

/*sets a bit given a position and a bitmap*/
void set(int* bitmap, int pos) {
      *bitmap = *bitmap | (1 << pos);
}

void reset(int* bitmap, int pos) {
      *bitmap = *bitmap & ~(1 << pos);
}

int check(int* bitmap, int pos) {
      return *bitmap & (1 << pos);
}


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

      if(argc < 2) {
            printf("enter an option");
            return 1;
      }

      //open input.txt as read only
      FILE* file = fopen("input.txt", "r");
      if(file == NULL) {
            perror("failed to open file");
      }
      //get to end of file and get file size from position
      fseek(file, 0, SEEK_END);
      long file_size = ftell(file);
      fseek(file, 0, SEEK_SET);
```

```
	//read file
	char word[1];
	int boardrows[81];
	int boardcols[81];
	int boardsquares[81];
	int board_counter = 0;
	while(fscanf(file, "%1s", word) == 1) {
		//store data in arrays preanalyzed
		boardrows[board_counter] = atoi(word);
		boardcols[board_counter * 9 - (80 * (board_counter / 9))] =
atoi(word);
		int mod_counter = board_counter / 3;
		boardsquares[(mod_counter * 3 - (8 * (mod_counter / 3)) + 6 *
(mod_counter / 9)) * 3 + (board_counter % 3)] = atoi(word);
```




```
		board_counter++;
		if(board_counter > 81)
			break;
	}

	int solution = 1;
	int columns[9] = { 0 };
	int rows[9] = { 0 };
	int squares[9] = { 0 };

	/* Single Threaded check */
	if(atoi(argv[1]) == 1) {

	for(int i = 0; (i < 9 && solution); i++) {
		for(int n = 0; n < 9; n++) {
			//check columns
			if(check(&columns[i], boardcols[i * 9 + n]) == 0) {
				set(&columns[i], boardcols[i * 9 + n]);
			} else {
				solution = 0;
				break;
			}
			//check rows
			if(check(&rows[i], boardrows[i * 9 + n]) == 0) {
				set(&rows[i], boardrows[i * 9 + n]);
			} else {
				solution = 0;
```

```
                        break;
                  }
                  //check squares
                  if(check(&squares[i], boardsquares[i * 9 + n]) == 0) {
                        set(&squares[i], boardsquares[i * 9 + n]);
                  } else {
                        solution = 0;
                        break;
                  }
            }
      }

      if(solution) {
            printf("\nYES\n");
      } else {
            printf("\nNO\n");
      }


      /* Multi Threaded check */
      } else if(atoi(argv[1]) == 2) {

      /* Multi Process check */
      } else if(atoi(argv[1]) == 3) {

      }

      /* Check results */

      for(int i = 0; i < 81; i++) {
            printf("%d%s", boardcols[i], (((i + 1) % 9 == 0) ? "\n" : "
"));
      }

      fclose(file);

      return 0;
}
```

0.2. Results/Notes

Started program; completed single-threaded approach

First, we opened the file and parsed through the file into 3 separate one dimensional arrays in such a way that in transpose the board so each array can be checked linearly. There's an array for rows, columns, and squares. Then, in the single threaded approach I run a single for loop that uses a bitmap to check which digits I encountered. If I encounter a digit that I have already seen, I break out and set the solution to 0.

1. Version 2 Code

1.1. Source Code (only new additions; … indicates same as prev code)

Version 2 Code:

```
...
#include <pthread.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/mman.h>

typedef struct {
   int index;
   int solution;
} Index;

int boardrows[81];
int boardcols[81];
int boardsquares[81];
int board_counter = 0;

...

void* check_cols(void* param) {
   //unpack parameters
   Index* args = (Index*)param;
   args->solution = 1;
   int bitmap = 0;

   for(int i = args->index; (i < args->index + 9 && args->solution); i++) {
      if(check(&bitmap, boardcols[i]) == 0) {
         set(&bitmap, boardcols[i]);
      } else {
         args->solution = 0;
         break;
      }
   }
   pthread_exit(0);
}

void* check_rows(void* param) {
   //unpack parameters
   Index* args = (Index*)param;
```

```
    args->solution = 1;
    int bitmap = 0;

    for(int i = args->index; (i < args->index + 9 && args->solution); i++) {
        if(check(&bitmap, boardrows[i]) == 0) {
            set(&bitmap, boardrows[i]);
        } else {
            args->solution = 0;
            break;
        }
    }
    pthread_exit(0);
}

void* check_squares(void* param) {
    //unpack parameters
    Index* args = (Index*)param;
    args->solution = 1;
    int bitmap = 0;
    for(int i = args->index; (i < args->index + 9 && args->solution); i++) {
        if(check(&bitmap, boardsquares[i]) == 0) {
            set(&bitmap, boardsquares[i]);
        } else {
            args->solution = 0;
            break;
        }
    }
    pthread_exit(0);
}


int main(int argc, char** argv) {
    ...

    //solution flag
    int solution = 1;
    //param structs for threads
    Index col_param[9];
    Index row_param[9];
    Index square_param[9];

    if(atoi(argv[1]) == 1) {
    //bitmaps for singlethreaded and multithreaded checks
```

```
    int columns[9] = { 0 };
    int rows[9] = { 0 };
    int squares[9] = { 0 };

    /* ----------------------------------------------------Single Threaded
check---------------------------------------------------- */
          ...

    } else if(atoi(argv[1]) == 2) {
    /* ----------------------------------------------------Multi Threaded
check---------------------------------------------------- */
    //create parameters for each thread
    //create column threads
    pthread_t tid_col[9];
    pthread_t tid_row[9];
    pthread_t tid_square[9];
    int col_valid = 1;
    int row_valid = 1;
    int square_valid = 1;
    //instantiate index and create threads
    for(int i = 0; i < 9; i++) {
        col_param[i].index = i * 9;
        row_param[i].index = i * 9;
        square_param[i].index = i * 9;
        pthread_create(&tid_col[i], NULL, check_cols, (void*)&col_param[i]);
        pthread_create(&tid_row[i], NULL, check_rows, (void*)&row_param[i]);
        pthread_create(&tid_square[i], NULL, check_squares, (void*)&square_param[i]);
    }
    //join threads
    for(int i = 0; i < 9; i++) {
        pthread_join(tid_col[i], NULL);
        pthread_join(tid_row[i], NULL);
        pthread_join(tid_square[i], NULL);
    }
    //check all returned solutions
    for(int i = 0; i < 9; i++) {
        col_valid = col_valid && col_param[i].solution;
        row_valid = row_valid && row_param[i].solution;
        square_valid = square_valid && square_param[i].solution;
    }
    //join all solutions
    solution = col_valid && row_valid && square_valid;
```

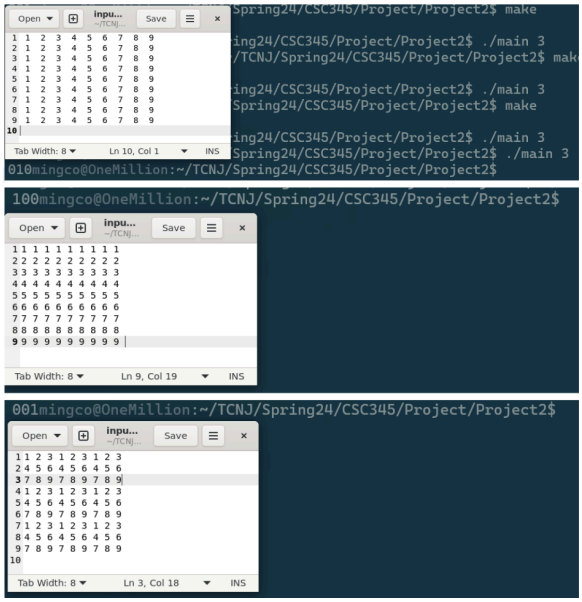
```
    } else if(atoi(argv[1]) == 3) {
    /* ----------------------------------------------------Multi Process
check---------------------------------------------------- */

    //create shared memory
    const char* name = "SOLUTION";
    const int SIZE = 4096;
    int shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    ftruncate(shm_fd, SIZE);
    int* ptr = (int*)mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);


    pid_t child[3];
    //create first process
    child[0] = fork();
    if(child[0] == 0) {
        int col_sol = 1;
        int columns[9] = { 0 };
        for(int i = 0; (i < 9 && solution); i++) {
            for(int n = 0; n < 9; n++) {
                //check columns
                if(check(&columns[i], boardcols[i * 9 + n]) == 0) {
                    set(&columns[i], boardcols[i * 9 + n]);
                } else {
                    col_sol = 0;
                    break;
                }
            }
        }
        ptr[0] = col_sol;
        exit(0);
    } else if(child[0] < 0) {
        perror("fork");
        exit(1);
    }
    //create second process
    child[1] = fork();
    if(child[1] == 0) {
        int row_sol = 1;
        int rows[9] = { 0 };
        for(int i = 0; (i < 9 && solution); i++) {
            for(int n = 0; n < 9; n++) {
```

```
            //check rows
            if(check(&rows[i], boardrows[i * 9 + n]) == 0) {
                set(&rows[i], boardrows[i * 9 + n]);
            } else {
                row_sol = 0;
                break;
            }
        }
    }
    ptr[1] = row_sol;
    exit(0);
} else if(child[1] < 0) {
    perror("fork");
    exit(1);
}
//create third process
child[2] = fork();
if(child[2] == 0) {
    int square_sol = 1;
    int squares[9] = { 0 };
    for(int i = 0; (i < 9 && solution); i++) {
        for(int n = 0; n < 9; n++) {
            //check squares
            if(check(&squares[i], boardsquares[i * 9 + n]) == 0) {
                set(&squares[i], boardsquares[i * 9 + n]);
            } else {
                square_sol = 0;
                break;
            }
        }
    }
    ptr[2] = square_sol;
    exit(0);
} else if(child[2] < 0) {
    perror("fork");
    exit(1);
}

//wait
for(int i = 0; i < 3; i++) {
    waitpid(child[i], NULL, 0);
}
```

```
    solution = ptr[0] && ptr[1] && ptr[2];
    shm_unlink(name);
    close(shm_fd);
    }

    /* Check results */
    printf("BOARD STATE IN input.txt:\n");
    for(int i = 0; i < 81; i++) {
        printf("%d%s", boardcols[i], (((i + 1) % 9 == 0) ? "\n" : " "));
    }

    printf("\nSOLUTION: %s\n", ((solution) ? "YES" : "NO"));

    fclose(file);

    return 0;
}
```

1.2. Results/Notes

Added multi-threaded approach and then multi-process check.



^^^ *checking that multi-process worked*

For the multi-threaded approach and multi process approach, we just split up the for loop. The multi-threaded approach checks a row/column/square each, and each process checks all the rows, columns, or squares. For multi-threading, we used a struct to pass both the parameters and get

back return values. Then we just && the results together to determine if it is true. For multi-processing, we used shared memory to pass the results to the parent process.

2. Version 3 Code

2.1. Source Code

…

```
#define _POSIX_C_SOURCE 200809L //magic line idk what this does tbh

…

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

        …

        struct timespec ts_start;
        struct timespec ts_end;
        clock_gettime(CLOCK_REALTIME, &ts_start);
        printf("Seconds: %ld, Nanoseconds: %ld\n", ts_start.tv_sec, ts_start.tv_nsec);
        if(atoi(argv[1]) == 1) {

        …

        /* ----------------------------------------------------Single Threaded
check---------------------------------------------------- */
        …

        } else if(atoi(argv[1]) == 2) {
        /* ----------------------------------------------------Multi Threaded
check---------------------------------------------------- */
        …

        } else if(atoi(argv[1]) == 3) {
        /* ----------------------------------------------------Multi Process
check---------------------------------------------------- */
        …
        }

        clock_gettime(CLOCK_REALTIME, &ts_end);
        printf("Seconds: %ld, Nanoseconds: %ld\n", ts_end.tv_sec, ts_end.tv_nsec);
```

```
        printf("Difference: %lds, %ldns\n", (ts_end.tv_sec - ts_start.tv_sec), (ts_end.tv_nsec -
ts_start.tv_nsec));

        …

        return 0;
}
```

2.2. Results/observations

Started displaying time taken to complete program

Here, we used the time.h to determine the start and end time of the programs using the clock_gettime() method. Then we just print the difference.

3. Version 4 Code

3.1. Source Code

```
      ...


void* parse_rows(void* param) {
    for(int i = 0; i < 81; i++) {
        boardrows[i] = board[i];
    }
}


void* parse_cols(void* param) {
    for(int i = 0; i < 81; i++) {
        boardcols[i * 9 - (80 * (i / 9))] = board[i];
    }
}


void* parse_squares(void* param) {
    for(int i = 0; i < 81; i++) {
        int mod_counter = i / 3;
```

```
        boardsquares[(mod_counter * 3 - (8 * (mod_counter / 3)) + 6 *
(mod_counter / 9)) * 3 + (i % 3)] = board[i];
    }
}

      ...

int main(int argc, char** argv) {
      ...

    if(atoi(argv[1]) == 1) {
    /* ----------------------------------------------------Single Threaded
check---------------------------------------------------- */
    //parse array
    for(int i = 0; i < 81; i++) {
        boardrows[i] = board[i];
        boardcols[i * 9 - (80 * (i / 9))] = board[i];
        int mod_counter = i / 3;
        boardsquares[(mod_counter * 3 - (8 * (mod_counter / 3)) + 6 *
(mod_counter / 9)) * 3 + (i % 3)] = board[i];
    }

      ...

    } else if(atoi(argv[1]) == 2) {
    /* ----------------------------------------------------Multi Threaded
check---------------------------------------------------- */
    //create 3 threads to process int array

    // pthread_t tid_parse[3];
    // pthread_create(&tid_parse[0], NULL, parse_rows, NULL);
    // pthread_create(&tid_parse[1], NULL, parse_cols, NULL);
    // pthread_create(&tid_parse[2], NULL, parse_squares, NULL);
    // pthread_join(tid_parse[0], NULL);
    // pthread_join(tid_parse[1], NULL);
    // pthread_join(tid_parse[2], NULL);
    for(int i = 0; i < 81; i++) {
        boardrows[i] = board[i];
        boardcols[i * 9 - (80 * (i / 9))] = board[i];
        int mod_counter = i / 3;
```

```
        boardsquares[(mod_counter * 3 - (8 * (mod_counter / 3)) + 6 *
(mod_counter / 9)) * 3 + (i % 3)] = board[i];
    }

   ...

    } else if(atoi(argv[1]) == 3) {
    /* ----------------------------------------------------Multi Process
check---------------------------------------------------- */
    for(int i = 0; i < 81; i++) {
        boardrows[i] = board[i];
        boardcols[i * 9 - (80 * (i / 9))] = board[i];
        int mod_counter = i / 3;
        boardsquares[(mod_counter * 3 - (8 * (mod_counter / 3)) + 6 *
(mod_counter / 9)) * 3 + (i % 3)] = board[i];
    }

      ...


    /* Check results */
    printf("BOARD STATE IN input.txt:\n");
    for(int i = 0; i < 81; i++) {
        printf("%d%s", boardrows[i], (((i + 1) % 9 == 0) ? "\n" : " "));
    }
    printf("\n");

    for(int i = 0; i < 81; i++) {
        printf("%d%s", boardcols[i], (((i + 1) % 9 == 0) ? "\n" : " "));
    }
printf("\n");
    for(int i = 0; i < 81; i++) {
        printf("%d%s", boardsquares[i], (((i + 1) % 9 == 0) ? "\n" : "
"));
    }
    printf("\n");


    printf("\nSOLUTION: %s\n", ((solution) ? "YES" : "NO"));
```

```
    fclose(file);

    return 0;
}
```

3.2. Results/Notes

Because of the drastic differences between times in the single-threaded approach and the other two, I thought that the reason for it was due to the fact that I had preprocessed the board into 3 separate arrays when reading the file, and so the single-threaded approach only had to read through the arrays. To make it more fair, I had each approach parse the arrays itself but it didn't seem to make any real difference

4. FINAL CODE

4.1. Source Code

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <stdio.h>
#include <string.h>
#include <pthread.h>
#include <sys/wait.h>
#include <fcntl.h>
#include <sys/shm.h>
#include <sys/mman.h>


#define _POSIX_C_SOURCE 200809L //magic line idk what this does tbh


typedef struct {
    int index;
    int solution;
} Index;
```

```c
int board[81];
int boardrows[81];
int boardcols[81];
int boardsquares[81];
int board_counter = 0;



/*sets a bit given a position and a bitmap*/
void set(int* bitmap, int pos) {
    *bitmap = *bitmap | (1 << pos);
}


void reset(int* bitmap, int pos) {
    *bitmap = *bitmap & ~(1 << pos);
}


int check(int* bitmap, int pos) {
    return *bitmap & (1 << pos);
}


void* parse_rows(void* param) {
    for(int i = 0; i < 81; i++) {
        boardrows[i] = board[i];
    }
    pthread_exit(0);
}


void* parse_cols(void* param) {
    for(int i = 0; i < 81; i++) {
        boardcols[i * 9 - (80 * (i / 9))] = board[i];
    }
    pthread_exit(0);
}
```

```
void* parse_squares(void* param) {
    for(int i = 0; i < 81; i++) {
        int mod_counter = i / 3;
        boardsquares[(mod_counter * 3 - (8 * (mod_counter / 3)) + 6 *
(mod_counter / 9)) * 3 + (i % 3)] = board[i];
    }
    pthread_exit(0);
}


void* check_cols(void* param) {
    //unpack parameters
    Index* args = (Index*)param;
    args->solution = 1;
    int bitmap = 0;


    for(int i = args->index; (i < args->index + 9 && args->solution); i++)
{
        if(check(&bitmap, boardcols[i]) == 0) {
            set(&bitmap, boardcols[i]);
        } else {
            args->solution = 0;
            break;
        }
    }


    pthread_exit(0);
}


void* check_rows(void* param) {
    //unpack parameters
    Index* args = (Index*)param;
    args->solution = 1;
    int bitmap = 0;
```

```
    for(int i = args->index; (i < args->index + 9 && args->solution); i++)
{
        if(check(&bitmap, boardrows[i]) == 0) {
            set(&bitmap, boardrows[i]);
        } else {
            args->solution = 0;
            break;
        }
    }

    pthread_exit(0);
}

void* check_squares(void* param) {
    //unpack parameters
    Index* args = (Index*)param;
    args->solution = 1;
    int bitmap = 0;

    for(int i = args->index; (i < args->index + 9 && args->solution); i++)
{
        if(check(&bitmap, boardsquares[i]) == 0) {
            set(&bitmap, boardsquares[i]);
        } else {
            args->solution = 0;
            break;
        }
    }

    pthread_exit(0);
}

int main(int argc, char** argv) {
```

```
    if(argc < 2) {
        //printf("enter an option");
        //return 1;
    }


    //open input.txt as read only
    FILE* file = fopen("input.txt", "r");
    if(file == NULL) {
        perror("failed to open file");
    }
    //get to end of file and get file size from position
    fseek(file, 0, SEEK_END);
    long file_size = ftell(file);
    fseek(file, 0, SEEK_SET);


    //read file
    char word[2];
    while(fscanf(file, "%1s", word) == 1) {
        //store data in int array
        board[board_counter] = atoi(word);
        board_counter++;
    }

    //solution flag
    int solution = 1;
    //param structs for threads
    Index col_param[9];
    Index row_param[9];
    Index square_param[9];

    struct timespec ts_start;
    struct timespec ts_end;
    clock_gettime(CLOCK_REALTIME, &ts_start);
    //printf("Seconds: %ld, Nanoseconds: %ld\n", ts_start.tv_sec,
ts_start.tv_nsec);
    if(atoi(argv[1]) == 1) {
```

```
    /* ----------------------------------------------------Single Threaded
check---------------------------------------------------- */
    //parse array
    for(int i = 0; i < 81; i++) {
        boardrows[i] = board[i];
        boardcols[i * 9 - (80 * (i / 9))] = board[i];
        int mod_counter = i / 3;
        boardsquares[(mod_counter * 3 - (8 * (mod_counter / 3)) + 6 *
(mod_counter / 9)) * 3 + (i % 3)] = board[i];
    }

    //bitmaps for singlethreaded checks
    int columns[9] = { 0 };
    int rows[9] = { 0 };
    int squares[9] = { 0 };
    for(int i = 0; (i < 9 && solution); i++) {
        for(int n = 0; n < 9; n++) {
            //check columns
            if(check(&columns[i], boardcols[i * 9 + n]) == 0) {
                set(&columns[i], boardcols[i * 9 + n]);
            } else {
                solution = 0;
                break;
            }
            //check rows
            if(check(&rows[i], boardrows[i * 9 + n]) == 0) {
                set(&rows[i], boardrows[i * 9 + n]);
            } else {
                solution = 0;
                break;
            }
            //check squares
            if(check(&squares[i], boardsquares[i * 9 + n]) == 0) {
                set(&squares[i], boardsquares[i * 9 + n]);
            } else {
                solution = 0;
                break;
            }
        }
    }
```

```
    } else if(atoi(argv[1]) == 2) {
    /* ----------------------------------------------------Multi Threaded
check---------------------------------------------------- */
    //create 3 threads to process int array

    // pthread_t tid_parse[3];
    // pthread_create(&tid_parse[0], NULL, parse_rows, NULL);
    // pthread_create(&tid_parse[1], NULL, parse_cols, NULL);
    // pthread_create(&tid_parse[2], NULL, parse_squares, NULL);
    // pthread_join(tid_parse[0], NULL);
    // pthread_join(tid_parse[1], NULL);
    // pthread_join(tid_parse[2], NULL);
    for(int i = 0; i < 81; i++) {
        boardrows[i] = board[i];
        boardcols[i * 9 - (80 * (i / 9))] = board[i];
        int mod_counter = i / 3;
        boardsquares[(mod_counter * 3 - (8 * (mod_counter / 3)) + 6 *
(mod_counter / 9)) * 3 + (i % 3)] = board[i];
    }

    //create parameters for each thread
    //create column threads
    pthread_t tid_col[9];
    pthread_t tid_row[9];
    pthread_t tid_square[9];
    int col_valid = 1;
    int row_valid = 1;
    int square_valid = 1;
    //instantiate index and create threads
    for(int i = 0; i < 9; i++) {
        col_param[i].index = i * 9;
        row_param[i].index = i * 9;
        square_param[i].index = i * 9;
        pthread_create(&tid_col[i], NULL, check_cols,
(void*)&col_param[i]);
        pthread_create(&tid_row[i], NULL, check_rows,
(void*)&row_param[i]);
```

```
        pthread_create(&tid_square[i], NULL, check_squares,
(void*)&square_param[i]);
    }
    //join threads
    for(int i = 0; i < 9; i++) {
        pthread_join(tid_col[i], NULL);
        pthread_join(tid_row[i], NULL);
        pthread_join(tid_square[i], NULL);
    }
    //check all returned solutions
    for(int i = 0; i < 9; i++) {
        col_valid = col_valid && col_param[i].solution;
        row_valid = row_valid && row_param[i].solution;
        square_valid = square_valid && square_param[i].solution;
    }
    //join all solutions
    solution = col_valid && row_valid && square_valid;


    } else if(atoi(argv[1]) == 3) {
    /* ----------------------------------------------------Multi Process
check---------------------------------------------------- */
    for(int i = 0; i < 81; i++) {
        boardrows[i] = board[i];
        boardcols[i * 9 - (80 * (i / 9))] = board[i];
        int mod_counter = i / 3;
        boardsquares[(mod_counter * 3 - (8 * (mod_counter / 3)) + 6 *
(mod_counter / 9)) * 3 + (i % 3)] = board[i];
    }
    //create shared memory
    const char* name = "SOLUTION";
    const int SIZE = 4096;
    int shm_fd = shm_open(name, O_CREAT | O_RDWR, 0666);
    ftruncate(shm_fd, SIZE);
    int* ptr = (int*)mmap(0, SIZE, PROT_WRITE, MAP_SHARED, shm_fd, 0);


    pid_t child[3];
    //create first process
    child[0] = fork();
```

```
    if(child[0] == 0) {
        int col_sol = 1;
        int columns[9] = { 0 };
        for(int i = 0; (i < 9 && solution); i++) {
            for(int n = 0; n < 9; n++) {
                //check columns
                if(check(&columns[i], boardcols[i * 9 + n]) == 0) {
                    set(&columns[i], boardcols[i * 9 + n]);
                } else {
                    col_sol = 0;
                    break;
                }
            }
        }
        ptr[0] = col_sol;
        exit(0);
    } else if(child[0] < 0) {
        perror("fork");
        exit(1);
    }
    //create second process
    child[1] = fork();
    if(child[1] == 0) {
        int row_sol = 1;
        int rows[9] = { 0 };
        for(int i = 0; (i < 9 && solution); i++) {
            for(int n = 0; n < 9; n++) {
                //check rows
                if(check(&rows[i], boardrows[i * 9 + n]) == 0) {
                    set(&rows[i], boardrows[i * 9 + n]);
                } else {
                    row_sol = 0;
                    break;
                }
            }
        }
        ptr[1] = row_sol;
        exit(0);
    } else if(child[1] < 0) {
        perror("fork");
```

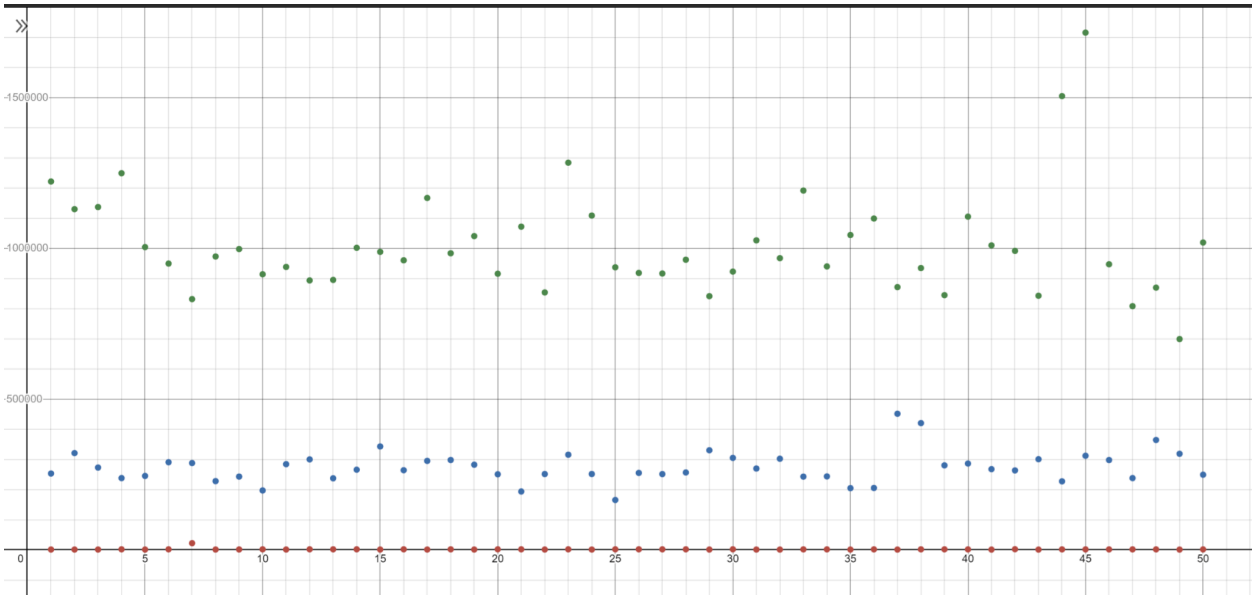
```
        exit(1);
    }
    //create third process
    child[2] = fork();
    if(child[2] == 0) {
        int square_sol = 1;
        int squares[9] = { 0 };
        for(int i = 0; (i < 9 && solution); i++) {
            for(int n = 0; n < 9; n++) {
                //check squares
                if(check(&squares[i], boardsquares[i * 9 + n]) == 0) {
                    set(&squares[i], boardsquares[i * 9 + n]);
                } else {
                    square_sol = 0;
                    break;
                }
            }
        }
        ptr[2] = square_sol;
        exit(0);
    } else if(child[2] < 0) {
        perror("fork");
        exit(1);
    }
    //wait
    for(int i = 0; i < 3; i++) {
        waitpid(child[i], NULL, 0);
    }


    solution = ptr[0] && ptr[1] && ptr[2];
    shm_unlink(name);
    close(shm_fd);


    }
    clock_gettime(CLOCK_REALTIME, &ts_end);
    //printf("Seconds: %ld, Nanoseconds: %ld\n", ts_end.tv_sec,
ts_end.tv_nsec);
    long diff_sec = (ts_end.tv_sec - ts_start.tv_sec);
```

```
    long diff_ns = (ts_end.tv_nsec - ts_start.tv_nsec);

    /* Check results */
    printf("BOARD STATE IN input.txt:\n");
    for(int i = 0; i < 81; i++) {
        printf("%d%s", boardrows[i], (((i + 1) % 9 == 0) ? "\n" : " "));
    }

    printf("\nSOLUTION: %s (%ld nanoseconds)\n", ((solution) ? "YES" :
"NO"),  diff_ns);


    fclose(file);


    return 0;
}
```

4.2. Results/Notes

This is the finalized code, with the only difference from what was asked being it displays the final time in nanoseconds, rather than seconds, as if I were to show seconds for the single threaded approach, it would look like this: 0.0000001393.

5. Statistical Experiment



single threaded; multiprocess; multithreaded

5.1. Comparing Option 1 and 2

Null Hypothesis: There is no statistically significant difference between two methods."

Alternative Hypothesis: Multi-threaded approach is faster than the single-threaded approach

The t-value is -41.17727. The p-value is < .00001. The result is significant at $p < .05$.

As seen through the graph and values, there was indeed a significant difference between the single-threaded approach and the multi-threaded approach. However, the multi-threaded approach was *not* faster than the single-threaded approach. The single-threaded approach was much faster by nearly 10^7 nanoseconds.

5.2. Comparing Option 2 and 3

Null Hypothesis: There is no statistically significant difference between two methods."

Alternative Hypothesis: Multi-threaded approach is faster than the multiprocess approach

The t-value is 28.77184. The p-value is < .00001. The result is significant at $p < .05$.

As the graph shows, there was indeed a significant difference between the multi-threaded approach and the multiprocess approach. Once again, the multi-threaded approach was *not* faster as we had predicted. It was still slower than the multi-processed approach.