Report

CO23BTECH11006

March 2025

0.1 Introduction

This Report is a part of Programming Assignment 2 of CS3523: Operating System - II.

Contents

Chapter 1

Program's Low-Level Design

This chapter contains a detailed explanation of the program and the components used for the assignment.

1.1 Dependencies

The Script we will discuss has dependencies, and we need to understand the dependent header before moving on!!!.

1.1.1 POSIX threads

POSIX threads or pthread is a header file that implements threading in Linux and similar operating systems. The scripts depend entirely on the pthread header to implement multi-threading and synchronization.

1.1.2 atomic

atomic is another header file used to implement synchronization using lock and atomic methods. for further information on `std::atomic` please refer atomic reference link

1.1.3 vector

vector is a template container used for storage and ease of use while scripting. The vector object is entirely used to contain an array of data local to the respective function.

1.1.4 chrono

The `std::chrono` is used for time-related jobs, in the scripts we will be dealing with multiple print statements related to time differences and we need some high precision time value to compute accurate time differences, fortunately, we have high precision clock variable in Chrono which will be used in the scripts.

1.1.5 cmath

Standard math library for C and C++ this contains some essential datatypes like `size_t`.

1.1.6 iostream & fstream

These headers contain C++ file handling, input, and output datatypes, the script won't have that many instances of these headers.

1.2 Datatypes and Objects

Now, that we have seen the dependencies we will see the User-defined datatypes present in the Scripts and their Uses.

1.2.1 `struct thread_params`

The datatype `struct thread_params` contains the necessary information needed for a thread to execute. The structure contains 3 different parts to it.

- **Lock Variables**

```
std::string lock_type;
std::atomic<int> * task_count;
struct cas_lock * cas;
std::atomic_flag * lock;
```

  This contains all necessary lock variables:

  1. `lock_type`: Variable which determines what lock should be used for the respective thread.
  2. `task_count`: Variable used to keep track of the number of tasks or validation completed. implemented using `std::atomic`. This is so that we won't be having any race conditions.
  3. `cas`: struct used to implement compare and swap lock for further info 1.4
  4. `lock`: Variable used to implement test and set lock for further info 1.4

- **Problem Parameters**

```
size_t no_of_threads;
size_t task_increament;
check_params cparmas;
```

  This contains the parameters which will be fetched from the input file.

  1. `no_of_threads`: Variable to contain the number of threads used in the problem.
  2. `task_increament`: Variable to contain the value of task increment used in the problem.
  3. `cparams`: structure used to contain the information about the Sudoku.

- **Miscellaneous**

```
bool * valid;
std::chrono::time_point<std::chrono::high_resolution_clock> anchor_time;
struct file__ file;
```

  This contains the extra parameters that are needed for the output and early termination.

  1. `valid`: Variable need to implement early termination concept.
  2. `anchor_time`: A fixed point in time that is needed for time calculations inside the thread.
  3. `file`: the structure that contains the necessary file pointers which will be used inside the thread.

1.2.2 check_params

```
size_t block_length;
size_t block_height;
size_t label;
int ** sudoku;
char search_type;
```

The datatype `check_params` contains the necessary variables and pointers for the validation of sudoku.

1. `block_length`: Variable which defines the length of the sub-grid in the Sudoku.
2. `block_height`: Variable which defines the height of the sub-grid in the Sudoku.
3. `label`: This variable represents the n^{th} validation type[1] to be validated.
4. `sudoku`: This variable points to the location of the sudoku which is fetched from the input file.
5. `search_type`: This variable represents the validation type to be validated e.g. 'r' for the row.

1.2.3 file__

```
FILE * out;
FILE * log;
bool close() const;
```

This structure contains the output and log file pointer which will be used to print time-related values.

1.2.4 cas_lock

```
std::atomic<int> cas_val;
std::atomic<int> curr_serve; // for bounded implementation.
std::atomic<int> total_tickets; // for bounded implementation.
```

This contains variables needed to implement compare and swap lock.

1. `cas_val`: the value to be compared and swapped inside the lock function.
2. `curr_serve`: the current serving ticket or serving thread's value.
3. `total_tickets`: the total number of threads that accessed the lock.

1.2.5 should_free (present in Src_header.cpp)

```
std::vector<long double> * ret;
```

The main purpose for this structure is to implement pthread_cancel but the current script doesn't implement pthread_cancel.

1.3 Function Definition and Uses

This section will be about the functions used in the script and their uses. This does not include the lock and unlock functions for the implementation of lock and unlock refer 1.4.

[1] row , column or sub-grid

1.3.1 check_part

```
bool check_part(check_params params){
    ...
}
```

This function validates a specific part of the validation type for a given sudoku.

- **Paramters**: `check_params params` information about the sudoku. (refer 1.2.2)
- **Return**: `bool` the specific part is valid or not.

1.3.2 Lock

```
void Lock(thread_params t__){
    ...
}
```

This function locks the global task counter using the pointer present in `thread_params` for a specific lock type which is also present in `thread_params`.

- **Paramters**: `thread_params params` information about lock variables and global variables. (refer 1.2.1)
- **Return**: `void` no return.

1.3.3 Unlock

```
void Unlock(thread_params t__){
    ...
}
```

This function unlocks the global task counter using the pointer present in `thread_params` for a specific lock type.

- **Paramters**: `thread_params params` information about lock variables and global variables. (refer 1.2.1)
- **Return**: `void` no return.

1.3.4 thread_main

```
void * thread_main(void * params){
    ... // initialization.
    while (...){
        ... // critical section.
        ... // remaining section.
    }
    ... // completion.
}
```

This is the main function that will implement the synchronization and contains the critical section of the scripts. We will look in detail at how this function works.

1. **Initialization**

```
should_free f_;
thread_params * param = (thread_params *) params;
thread_params tparams = *param;
pthread_t self_id = pthread_self();
size_t sudoku_size = (tparams.cparmas.block_height*tparams.cparmas.block_length);
long double CSout1 = 0,CSin = 0,CSout2 = 0,count = 0,sumen = 0,sumex = 0;
int local_task_count;
```

most of the code present in the initialization part is initialization but we will focus on 2 local variables. (refer `should_free` 1.2.5)

- `local_task_count`: used to store the global task counter value present in `thread_params`.
- `CS(in/out), sum(en/ex)`: both variables are used to find the time taken to enter and exit the critical section and the average time at entry and exit.

2. **Critical Section**

```
local_task_count = tparams.task_count->load();
if (*tparams.task_count >= sudoku_size*3 && *tparams.valid){
    unlock(tparams.lock);
    break;
}
tparams.task_count->fetch_add(tparams.task_increament);
```

The Critical Section of the function is pretty simple as we only modify a single global value. Here `tparams.valid` is used for the implementation of early termination.
The explanation for the Critical section is present in 1.5 section.

3. **Remaining Section**

```
for (...){
    tparams.cparmas.search_type = ...;
    tparams.cparmas.label = (local_task_count + i)%sudoku_size;
    fprintf(...);
    if (check_part(tparams.cparmas)){
        fprintf(...);
    }
    else{
        fprintf(...);
        *tparams.valid = false;
        break;
    }
}
sumen += std::max(CSin - CSout1,0.0L);
sumex += std::max(CSout2 - CSin,0.0L);
count++;
```

The remaining section is used to assign validation type and label through the `tparams` variable and check_part is called to validate for the respective argument. Here the `sum(en/ex)` is incremented with a time difference for each remaining section entry.

4. **Completion**

```
f_.ret->push_back(sumen/(count ? count : 1));
f_.ret->push_back(sumex/(count ? count : 1));
pthread_exit(f_.ret);
```

At the end of the function, we will just return the data we computed.

1.3.5 fetch_params

```
thread_params fetch_params(std::ifstream &file){
    ...
}
```

This function creates a default `thread_params` for the respective input file. Default values

- `label`: 0
- `check_params.search_type`: 'r'
- `lock_type`: "tas"

1.4 locks

This assignment required 3 lock variations to be implemented.

- test and set
- compare and swap
- compare and swap (bounded)

we will look at each of them in detail.

1.4.1 Test and Set

The test and set lock is just a spin-lock implemented with a boolean value. For each Critical section present during the script we need a boolean variable to control it, this variable is given through the structure `thread_params`. The function that implements the test and set lock is not present in the header instead it is local to the `Src_header.cpp` file.

```
void lock(std::atomic_flag * tas_lock){
    while (tas_lock->test_and_set());
}
```

normal c++ boolean value can't be used here because of race conditions instead we will use `std::atomic_flag` datatype from the atomic header file. The test and set function for the boolean values is already defined in the atomic header.
The unlocking part is just assigning the boolean value to `false` this is done through `clear()` method for the datatype `atomic_flag`.

```
void unlock(std::atomic_flag * tas_lock){
    tas_lock->clear();
}
```

1.4.2 Compare and Swap

The compare and swap is also a basic spin-lock implemented with an int value. Similar to test and set we can't use basic C++ int datatype instead we use `std::atomic_int`. This variable will be passed in the `thread_params` structure. The function that implements the compare and set lock is not available in the header file.

```
void lock(cas_lock * cas_lock,bool bound__){
    int cas_expect,self_ticket;
    if (bound__) self_ticket = cas_lock->total_tickets.fetch_add(1);
    do{
        cas_expect = 0;
    }while (!cas_lock->cas_val.compare_exchange_weak(cas_expect,0,std::memory_order_acquire));
}
```

here the atomic header provides a function to implement compare and set for the atomic int datatype `compare_exchange_weak` method. We will define the expected and swapping values local to the function.
The unlocking part is just changing the value of the lock to the expected value.

```
void unlock(cas_lock * cas_lock,bool bound__){
    if (bound__) cas_lock->curr_serve.fetch_add(1);
    cas_lock->cas_val.store(0, std::memory_order_release);
}
```

1.4.3 Bounded Compare and Swap

For the bounded part, we will just reuse the previous compare and swap and add a ticket system to the script i.e. each time the thread calls the lock a ticket is created for the thread so even if the lock is set to the expected value the current ticket should be equal to the thread's ticket for it to acquire the lock.
the serving ticket and the total number of tickets will be present in the structure `cas_lock`(1.2.4).
The implementation of the ticket system is done through boolean algebra.

$$bool\_bounded = \texttt{bound\_\_} \tag{1.1}$$

$$bool\_lock = \texttt{!cas\_lock->cas\_val.compare\_exchange\_weak(...)} \tag{1.2}$$

$$bool\_ticket = \texttt{self\_ticket == cas\_lock->curr\_serv.load()} \tag{1.3}$$

$$final\_bool = (!bool\_bounded \wedge bool\_lock) \vee (bool\_lock \wedge bool\_ticket) \tag{1.4}$$

final code for the lock system.

```
int cas_expect,self_ticket;
if (bound__) self_ticket = cas_lock->total_tickets.fetch_add(1);
do{
    cas_expect = 0;
}while (final_bool);
```

for each function call number of tickets is updated. and assign to `self_ticket`. final code for the unlock system.

```
void unlock(cas_lock * cas_lock,bool bound__){
    if (bound__) cas_lock->curr_serve.fetch_add(1);
    cas_lock->cas_val.store(0, std::memory_order_release);
}
```

for each function call the serving ticket's value is incremented by 1.
That's it for the locks the next section will discuss `thread_main` in detail

1.5 `thread_main` Explaination

for the function's structure refer to 1.3.4. Since we have covered the necessary topics we will keep this short. The `thread_main()` function is just similar to `main()` in C so the thread will be created with `thread_main` in the main script. The function contains 2 important sections:

1. **Critical section**: This section deals with the global values acquired from the main script through `thread_params` struct since this region deals with the global value we will use locks to restrict the region for other threads this will solve the race condition.
   Inside the critical the task counter is incremented and assigned to the local counter variable for further use. The region contains an if statement just in case the tasks are completed after the thread gains access to the region.

2. **Remaining section**: The remaining section contains the rest of the process such as task allotment and validation through `check_part` function. The task allotment can be calculated through the local task counter. if we take sudoku size as N and task counter as C.

$$task = \begin{cases} \text{row} & \lfloor \frac{C}{N} \rfloor = 0 \\ \text{column} & \lfloor \frac{C}{N} \rfloor = 1 \\ \text{sub-grid} & \lfloor \frac{C}{N} \rfloor = 2 \end{cases} \tag{1.5}$$

$$label = CmodN \tag{1.6}$$

   After finding the validation task and label we will call `check_part` function for validation.
   if it is correct no problem we will go to the next task allotted if not we will set the global boolean to false and end the loop.

After the remaining section, we return whatever data we collected through `pthread_exit(...)`.

Chapter 2

Program's Performance

2.1 Introduction

In this chapter, we will be looking at graphs and tables for 3 different experiments.

1. Time Vs Sudoku Size (Fixed Threads)
2. Time Vs Task Increment (Fixed Size)
3. Time Vs Number of Threads (Fixed Size)

2.2 Experiment - 1

S. No	**X axis**	**TAS Max CS Entry Time (μs)**	**CAS Max CS Entry Time (μs)**	**Bounded CAS Max CS Entry Time (μs)**	**TAS Max CS Exit Time (μs)**	**CAS Max CS Exit Time (μs)**	**Bounded CAS Max CS Exit Time (μs)**
1	10	513	312	366.6	171.4	170.68	145.74
2	20	144	155.89	68.1	90.72	57.18	35.65
3	30	112	49.8	125.23	59.26	34.64	38.18
4	40	67.71	66.9	41.1	34.75	37.88	34.98
5	50	57.31	105.75	61.02	26.96	84.36	63
6	60	92.86	35.32	37.1	70.46	53.02	110
7	70	50.74	33.88	41.25	30	39.2	45.7
8	80	45.25	114.13	128.72	21.28	83.32	83.38
9	90	143.44	157.65	108.55	37.77	158.95	18.86

Table 2.1: Worst time comparison for experiment - 1.

S. No	X axis	TAS Avg CS Entry Time (μs)	CAS Avg CS Entry Time (μs)	Bounded CAS Avg CS Entry Time (μs)	TAS Avg CS Exit Time (μs)	CAS Avg CS Exit Time (μs)	Bounded CAS Avg CS Exit Time (μs)	Total Time Taken (s)
1	10	181.4	126.4	169.1	33.9	38.32	29.14	0.0082 0.0083 0.0095
2	20	64.9	58.7	32.6	22.7	17.3	10	0.033 0.03 0.029
3	30	59.3	25	37.05	11.23	11.5	12.5	0.097 0.12 0.11
4	40	26.75	26.6	18.5	10	11.52	10.7	0.328 0.33 0.30
5	50	22.7	30.5	24.6	9.34	22.34	14.4	0.68 0.78 0.729
6	60	38.2	16.8	15	28.12	13.51	22.01	1.72 1.32 1.458
7	70	20.1	16.2	20.5	14.34	11.8	13.7	2.78 2.38 2.65
8	80	17.4	42.5	65.7	10.23	21.37	24.5	4.43 4.62 5.25
9	90	51.4	58.4	34.4	15	40.85	20.4	8.19 9.925 9.482

Table 2.2: Average time comparison for experiment - 1. (the total time is split into 3 rows tas,cas,bounded cas)

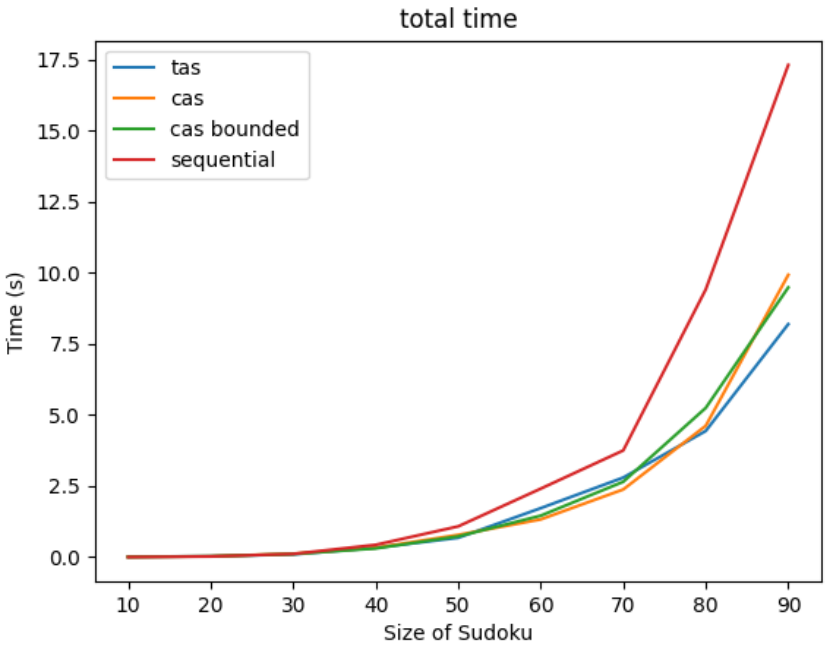

Figure 2.1: Graph between Time and total time taken by each lock type

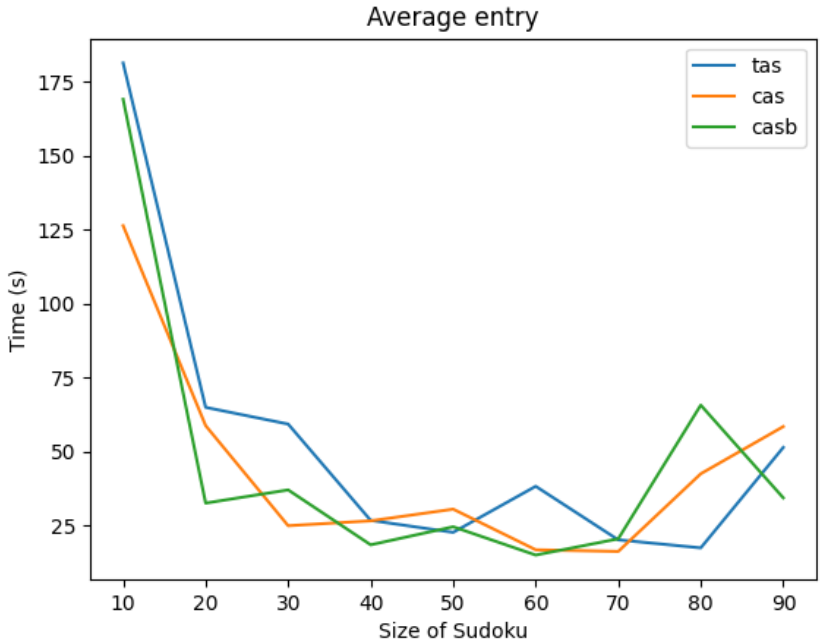

Figure 2.2: Graph between Time and Average entry time taken by each lock type

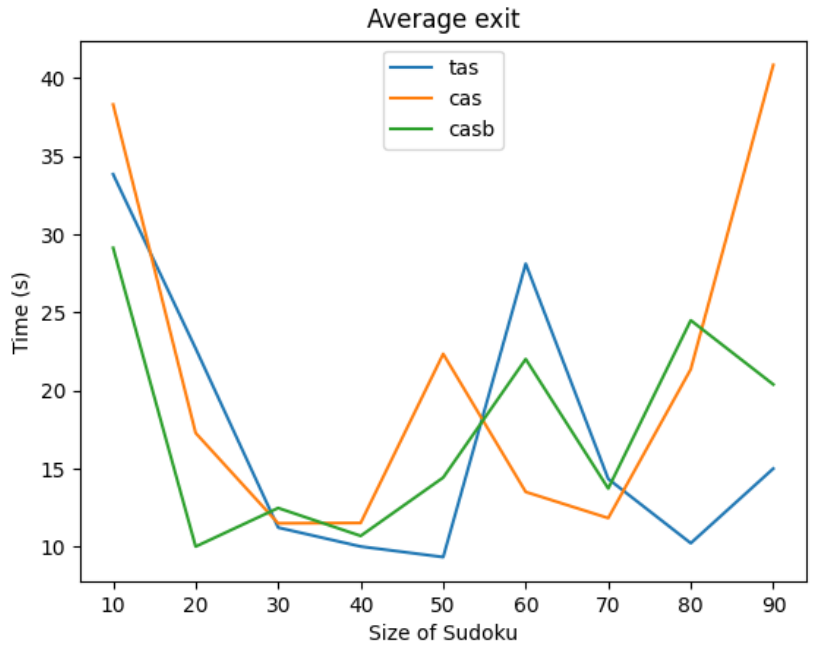

Figure 2.3: Graph between Time and Average exit time taken by each lock type

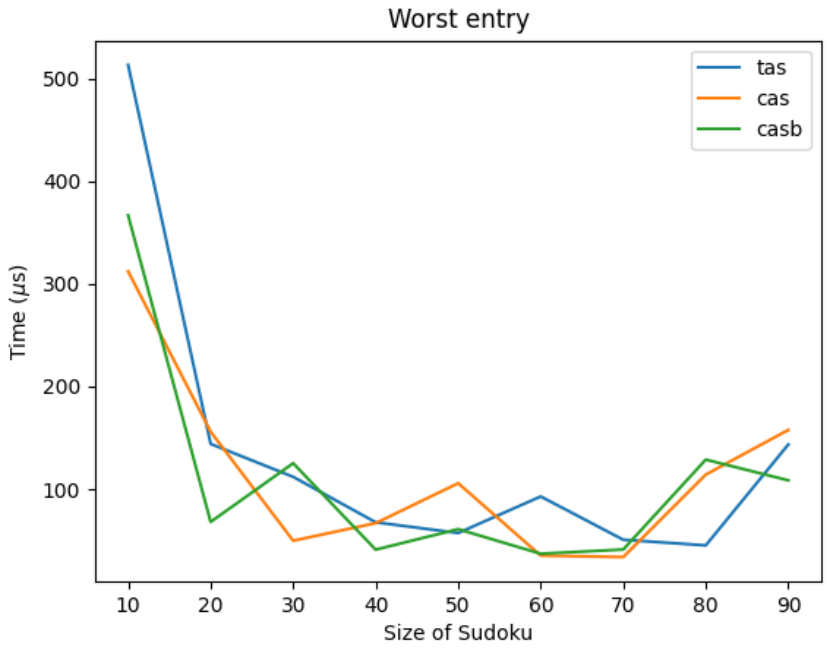

Figure 2.4: Graph between Time and worst entry time taken by each lock type

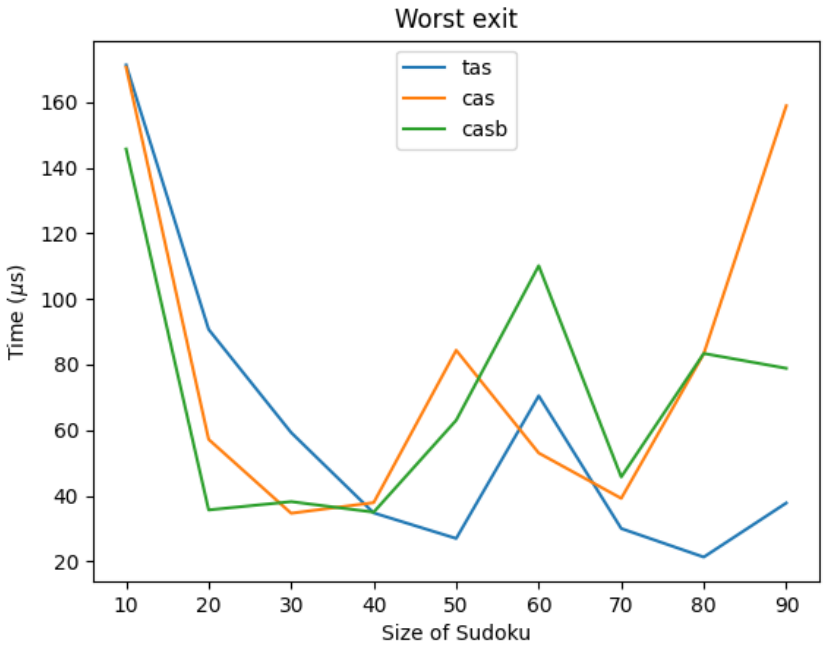

Figure 2.5: Graph between Time and worst exit time taken by each lock type

2.3 Experiment - 2

S. No	X axis	TAS Avg CS Entry Time (μs)	CAS Avg CS Entry Time (μs)	Bounded CAS Avg CS Entry Time (μs)	TAS Avg CS Exit Time (μs)	CAS Avg CS Exit Time (μs)	Bounded CAS Avg CS Exit Time (μs)	Total Time Taken (s)
1	10	231.47	45.43	38.38	67.63	30.94	21.48	3.466 3.469 3.457
2	20	34.7	28.09	66.73	17.44	20.68	40.17	3.321 3.558 3.428
3	30	114.2	49.68	121.83	56.57	44.10	45.04	3.68 3.6 2.83
4	40	47.27	79.92	411.01	18.89	20.36	24.26	3.13 3.25 2.80
5	50	77.36	40.28	20.7	17.21	40	9.11	3.06 3.05 2.95

Table 2.3: Average time comparison for experiment - 2. (the total time is split into 3 rows tas,cas,bounded cas)

S. No	X axis	TAS Max CS Entry Time (μs)	CAS Max CS Entry Time (μs)	Bounded CAS Max CS Entry Time (μs)	TAS Max CS Exit Time (μs)	CAS Max CS Exit Time (μs)	Bounded CAS Max CS Exit Time (μs)
1	10	356.21	91.39	92.64	129.69	105.6	65.38
2	20	133.06	68.18	221.23	69.67	80.26	142.39
3	30	257.19	153.77	215.48	163.7	152.77	219
4	40	177.42	85.32	289.31	80.57	411	92.32
5	50	249.65	115.12	62.53	93.88	177	31.17

Table 2.4: Worst time comparison for experiment - 2.

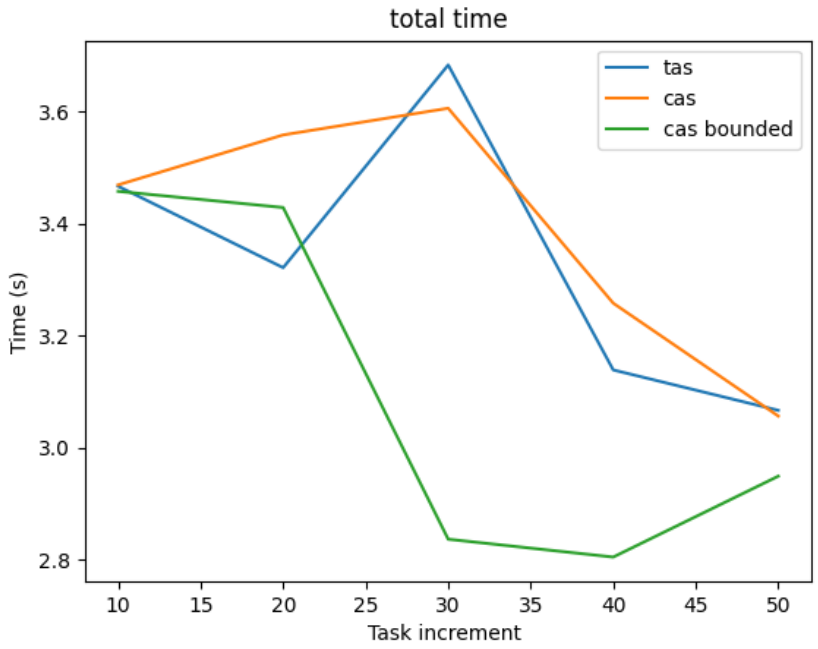

Figure 2.6: Graph between Time and total time taken by each lock type

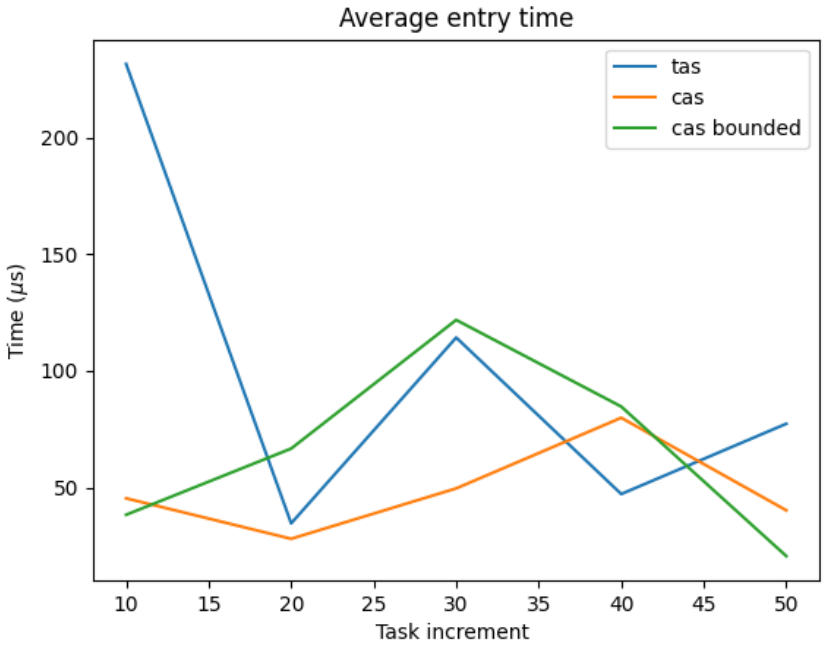

Figure 2.7: Graph between Time and Average entry time taken by each lock type

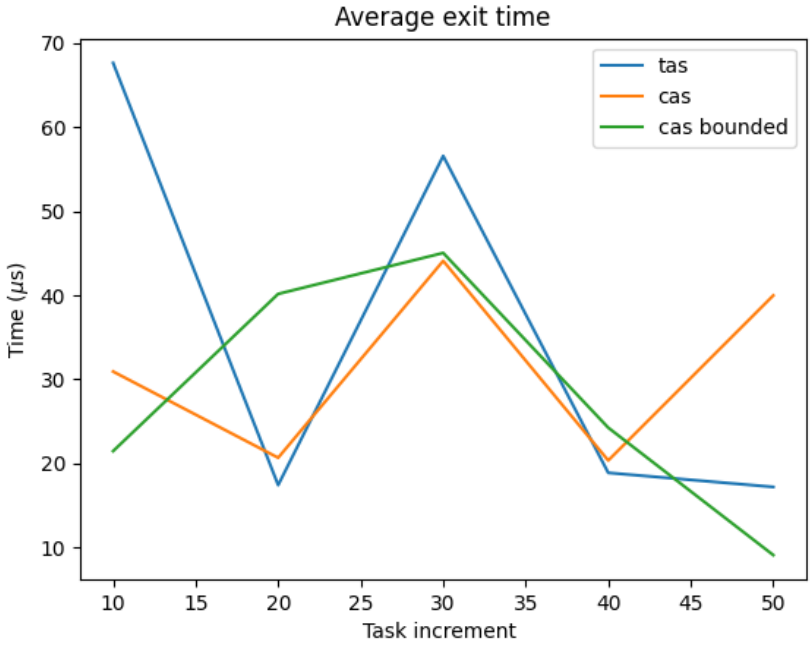

Figure 2.8: Graph between Time and Average exit time taken by each lock type

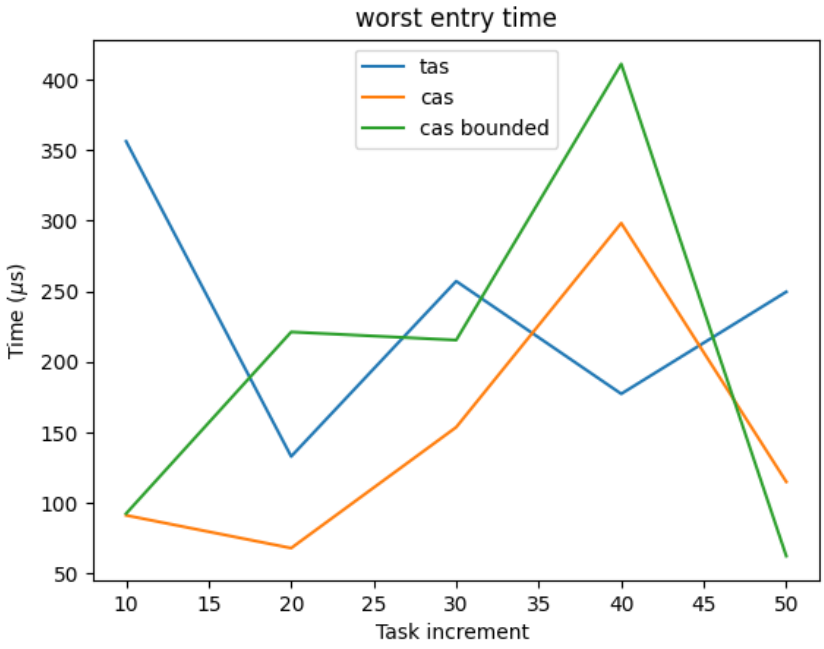

Figure 2.9: Graph between Time and worst entry time taken by each lock type

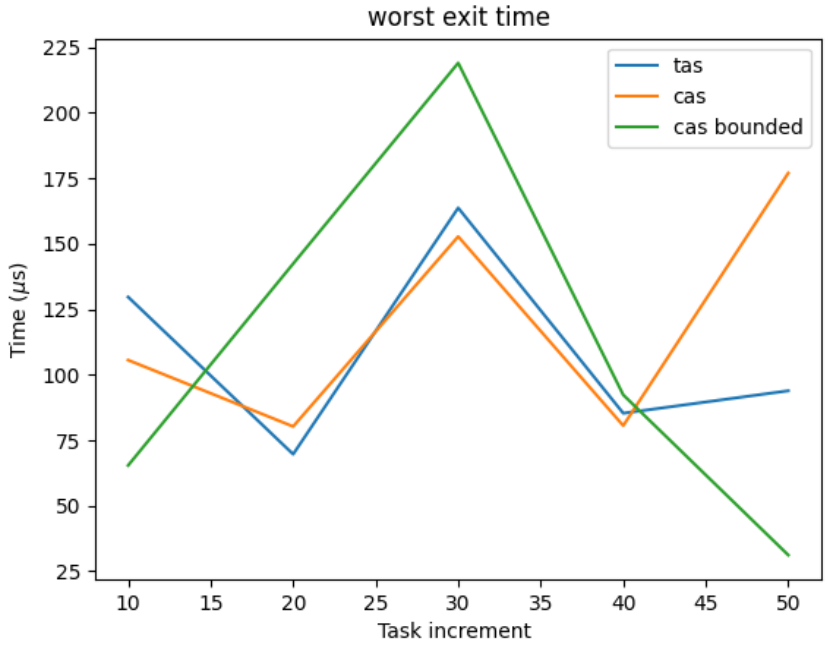

Figure 2.10: Graph between Time and worst exit time taken by each lock type

2.4 Experiment - 3

S. No	X axis	TAS Max CS Entry Time (μs)	CAS Max CS Entry Time (μs)	Bounded CAS Max CS Entry Time (μs)	TAS Max CS Exit Time (μs)	CAS Max CS Exit Time (μs)	Bounded CAS Max CS Exit Time (μs)
1	10	17.16	7.95	8.23	6.41	6.27	6.27
2	20	11.23	9	11.52	7.41	14.87	7.45
3	30	12.6	16.55	16.95	7.86	8.68	7.6
4	40	51.77	60.6	38.63	58.53	38.09	17
4	40	21798	1188	873	4857	689	646
5	50	378753	5911	4493	21852	4533	3465

Table 2.5: Worst time comparison for experiment - 3.

S. No	X axis	TAS Avg CS Entry Time (μs)	CAS Avg CS Entry Time (μs)	Bounded CAS Avg CS Entry Time (μs)	TAS Avg CS Exit Time (μs)	CAS Avg CS Exit Time (μs)	Bounded CAS Avg CS Exit Time (μs)	Total Time Taken (s)
1	10	17.16	7.95	8.23	6.41	6.27	6.26	8.33 7.54 7.77
2	20	9.65	8.05	9.52	6.11	10	6	4.98 4.76 4.78
3	30	9.05	10	11.78	4.5	6.34	4.7	3.30 3.36 3.36
4	40	25.43	17.82	17.04	15.77	13.79	7.4	2.81 2.87 2.49
4	40	16980	249	236	2463	180	167	4.22 2.96 2.76
5	50	241829	1061	899	12225	487	452	9.33 3.16 2.86

Table 2.6: Average time comparison for experiment - 3. (the total time is split into 3 rows tas,cas,bounded cas)

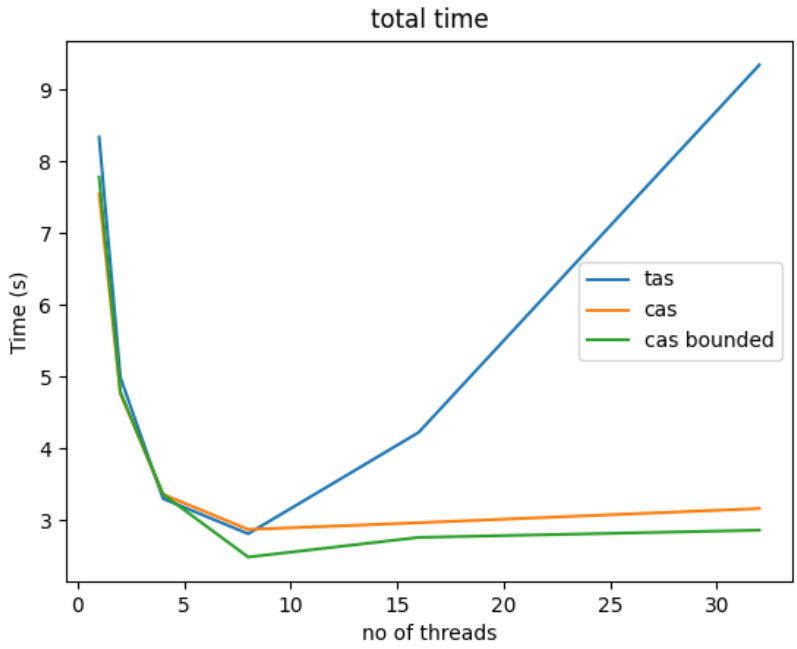

Figure 2.11: Graph between Time and total time taken by each lock type

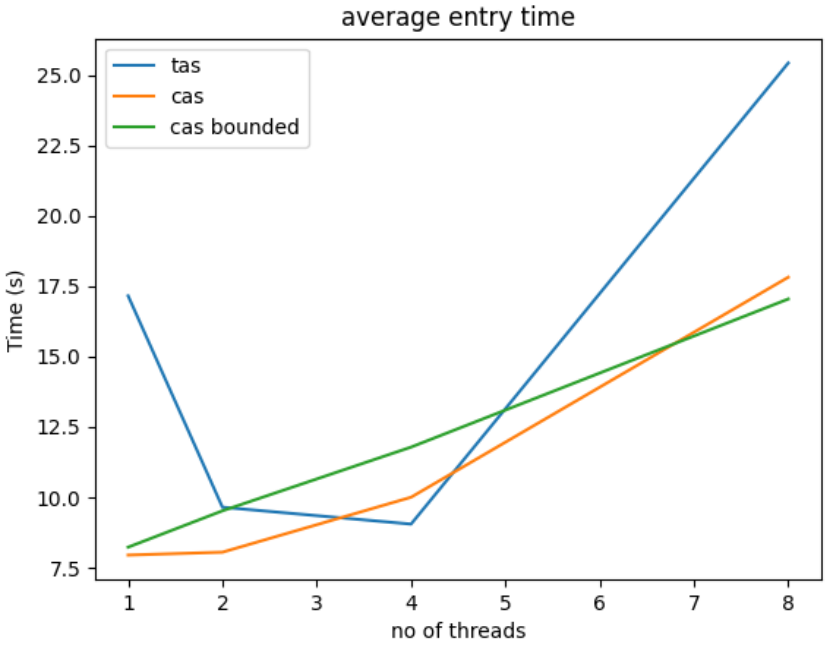

Figure 2.12: Graph between Time and Average entry time taken by each lock type

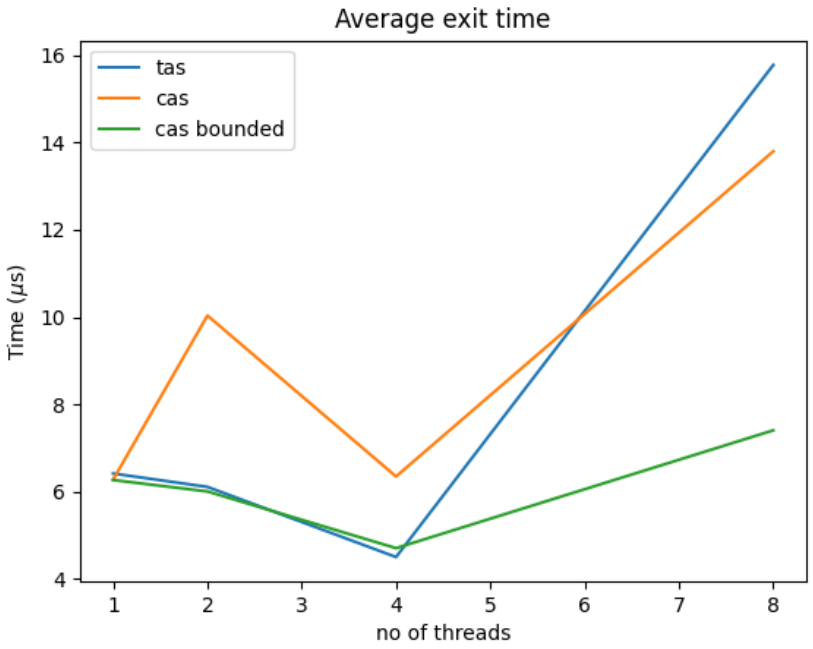

Figure 2.13: Graph between Time and Average exit time taken by each lock type

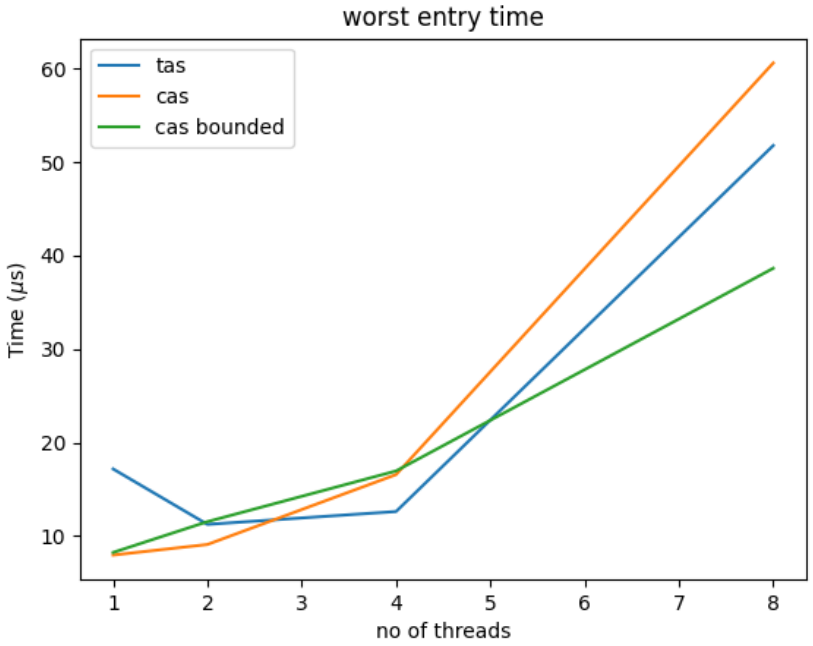

Figure 2.14: Graph between Time and worst entry time taken by each lock type

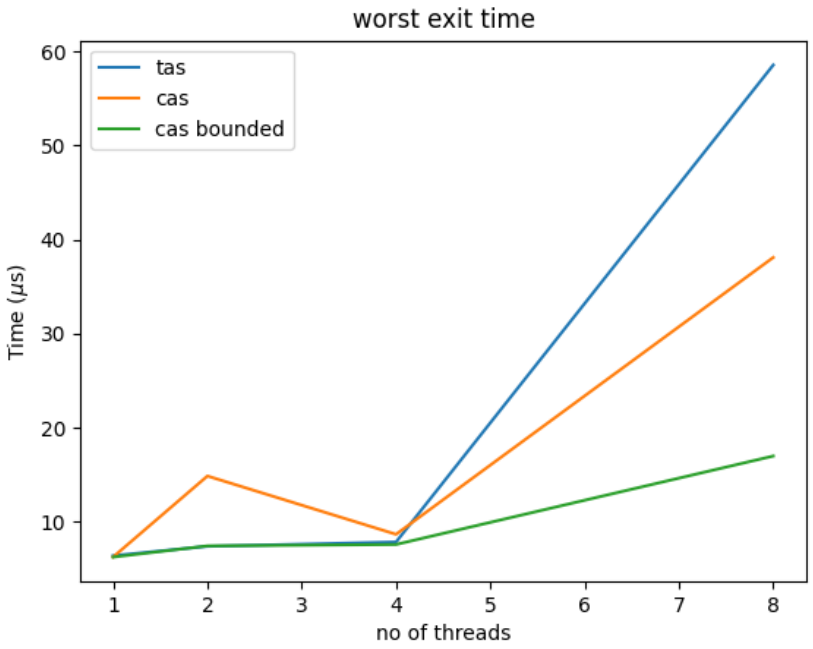

Figure 2.15: Graph between Time and worst exit time taken by each lock type

Chapter 3

Observations

we will take observation for each experiment in the order

1. Total time
2. average entry time
3. average exit time
4. worst entry time
5. worst exit time

3.1 Experiment - 1

For experiment - 1.

3.1.1 Total time

- The first thing we observe is that the graph 2.1 is exponential. and we could get the form for it.

$$\text{time taken } (\tau) \sim \tau_0 \frac{N^4}{K} \tag{3.1}$$

  where N is the size of the sub-grid, and K is the number of threads used.

- next, we can see that the sequential is taking more time to validate than the multi-threading method this also could be explained using the equation 3.1, by increasing the K value we decrease the value attained by the function.
- This means that both sequential and multi-threading methods are of the same order but differ by magnitude of growth.
- Nothing can be said about the differences in the different multi-threading methods since the maximum value of 90×90 is not that large to create sufficient value variations among them.

3.1.2 Average entry time

- The plot 2.2 for the entry and exit time for the threads shouldn't vary for the change in the size of the sudoku and most of the graph is constant except at the start.
- This may be due to extra processes or other external factors that interfere with the data. This interference could also be seen in other graphs.

3.1.3 Average exit time/ Worst entry time/ Worst exit time

There is no relation between the size of the Sudoku and the entry and exit time for the Critical section and all the plots (2.3,2.4,2.5) have some downward to upward trend this may be due to the OS scheduling or other factors.

3.2 Experiment - 2

3.2.1 Total time

- from the graph 2.6 we can observe that the total time taken is decreasing but not for all the methods.
- we can see that the **test and set**,**compare and swap** method's rate of decrease is lower than the **compare and swap bounded method** this is due to less number of threads engaging i.e.
- Even though we have 20 threads for both **test and set** and **compare and swap** not that many threads are getting access to the critical region, this is just *starving*.
- **compare and swap bounded method** does a better job at scheduling the access to the threads. so larger differences in time.

3.2.2 Critical section time differences

Similar to Experiment - 1 the change in `task increment` shouldn't affect the time taken to acquire the lock and unlock. This means any trend we see at one of the plots should be occurring in all of them and that is what we observe in the plots for the experiment - 2 2.3. All graphs have the the same or similar variations at the start peak at the center and fall back at the end.
This may be due to some external factors or the OS scheduler might have put the kernel threads to sleep.

3.3 Experiment - 3

3.3.1 total time

- for the total time we can see that the graph 2.11 is near hyperbolic curve except for **Test and set**, this can be explained by the equation 3.1 were for a fixed matrix size the equation is hyperbolic.
- The reason why **Test and set** increases as the thread count increases is due to the starving of the other threads. Even though we have 16 to 32 threads only a few of them are validating the sudoku the rest are waiting for access to the Critical section.
- This also explains the reason why **Bounded compare and swap** is doing better than **compare and swap** since bounded does a better job scheduling access of the critical region to the threads reducing the number of starving threads.

3.3.2 Average entry time

- from graph 2.12 we can see that the graph is linear or proportional to the number of threads this should be clear, since if we increase the number of threads the probability for a thread to acquire the lock will decrease so the wait time will increase.
- The reason why the **test and set** is growing more than the **compare and swap and bounded** is due to starvation.

3.3.3 Average exit time

- Even though in theory we shouldn't find any relation between exit time and any parameter, the increasing pattern in the graph could be explained differently.
- we know that both **test and set** and **compare and swap** are affected by starvation this causes the entry time to increase but this also affects exit time indirectly.
- Since a limited number of threads acquire the critical section more often these threads will be put to sleep by the OS this will indirectly give sudden jumps in exit time. Since we are calculating the average of the exit time these values will contribute more to the final result making it look like the graph is increasing.

3.3.4 Worst entry time

The worst exit time entirely depends on the number of threads since as we increase the number of threads the entry time for all threads will increase proportionally; therefore, it is independent of the method of implementation.

3.3.5 Worst exit time

As we said previously, there is no direct connection between the number of thread parameters and exit time. But due to OS scheduling the executing threads might get sudden jumps in value.
The reason for **bounded compare and swap** having less value is due to less starvation; therefore, even if a thread is put to sleep, the probability that the thread is executing is less than other methods.