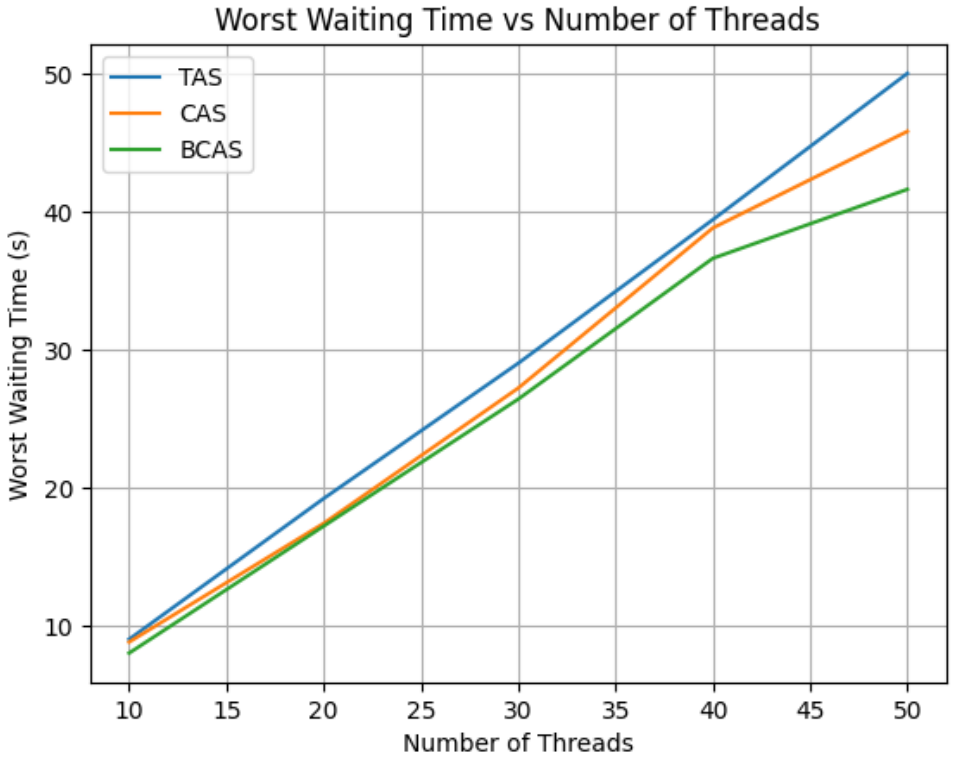
Implementing TAS, CAS and Bounded Waiting CAS Mutual Exclusion Algorithms

Aaryan, CO21BTECH11001

Note: For the sake of saving time and also simulating that threads are performing some time consuming tasks, I have set the values of λ_1 and λ_2 as $\lambda_1 = 0.4$, $\lambda_2 = 0.4$, so that we can get output within a minute or two.

- ## Comparison of Worst Waiting Time:

Here is the plot of $Worst\ Waiting\ Time\ vs\ Number\ of\ Threads$ using different algorithms:



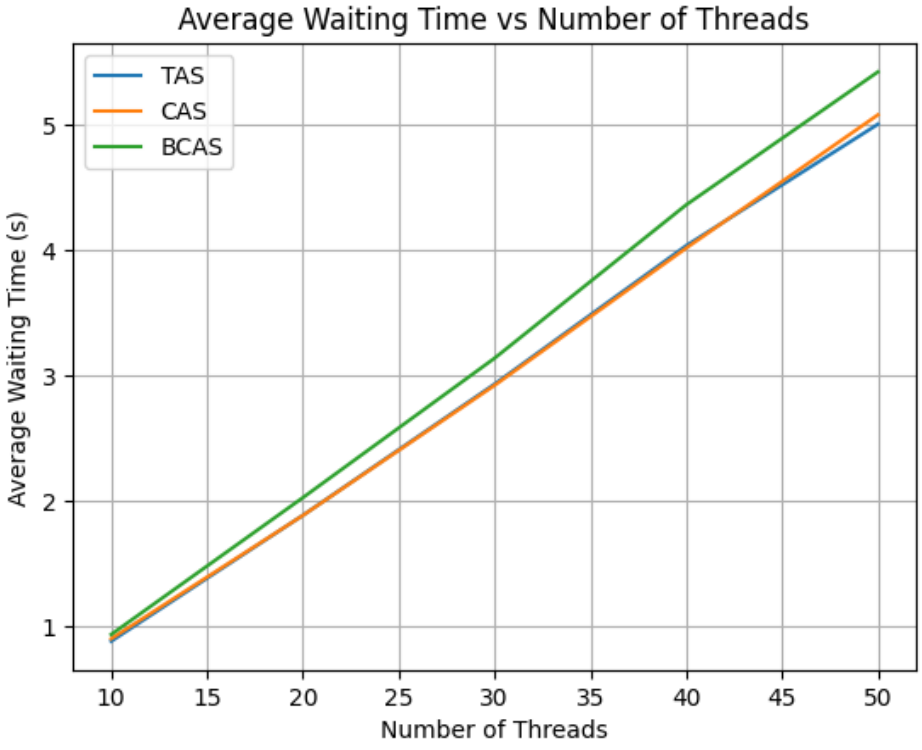
The trivial observation is that the worst waiting time increases by increasing the number of threads in every algorithm. Reason for this is that the worst

waiting time is directly proportional to the number of threads contending to enter the critical section.

The reason for slightly better performance of Bounded CAS as compared to TAS and CAS is that Bounded CAS satisfies the bounded waiting requirement.
Let's suppose that there are n threads. So, when a thread leaves its critical section, it scans the array $waiting$ in the cyclic ordering $(i + 1, i + 2, ..., n - 1, 0, ..., i - 1)$. It designates the first thread in this ordering, that is in the entry section $(waiting[j] == true)$, as the next one to enter the critical section. Any process waiting to enter its critical section will thus do so within $n - 1$ turns.

- **Comparison of Average Waiting Time:**

The plot of $Average\ Waiting\ Time\ vs\ Number\ of\ Threads$ using different algorithms is following:



Again, the worst waiting time increases by increasing the number of threads in every algorithm. Reason for this is that the worst waiting time is directly

proportional to the number of threads contending to enter the critical section.

Implementation of each algorithm has following parts in common:

- $totalWaitTime,\ worstWaitTime$: $totalWaitTime$ is used to calculate the average waiting time of the threads. It is updated every time a thread exits the critical section. The difference between actual entering time and requesting time is added to it.
  $totalWaitTime = totalWaitTime + (actEnterTime - reqEnterTime)$
  Similarly $worstWaitTime$ is updated as follows:
  $worstWaitTime = max(worstWaitTime, (actEnterTime - reqEnterTime))$

- *main( ) function:* Reads values of $n, k, \boldsymbol{\lambda}_1, \boldsymbol{\lambda}_2$ from the input file. Then creates n threads. Each thread will execute the $testCS( )$ function and parameters of each thread are passed by an object of a class which is different for each algorithm. After creating threads, they are joined together.

- $testCS( )$ *function:* First, it accesses all the parameters passed to the thread. Then, the thread will request to enter the critical section k times. Each time, it will print the time when it requests to enter the critical section. When the algorithm allows it to enter the critical section, it stores and prints the current time. Then, it enters the critical section and sleeps for a time of t_1 , where t_1 is generated randomly by an exponential distribution with a mean of $\boldsymbol{\lambda}_1$.
  After exiting critical section, it enters remainder section and sleeps for a time of t_2 , where t_2 is generated randomly by an exponential distribution with a mean of $\boldsymbol{\lambda}_2$.
  After that, $totalWaitTime$ and $worstWaitTime$ are updated.

- Finally, after execution of all threads, $averageWaitTime$ is calculated using $averageWaitTime = (totalWaitTime)/(n * k)$

Specifications of each algorithm are explained below:

- **Test and Set (TAS):**
  This algorithm ensures mutual exclusion between threads. This algorithm can be implemented atomically using $std::atomic\_flag::test\_and\_set()$ in $C++$.
  A mutex lock variable $\_lock$ is declared globally and initialized to $false$.
  A thread can enter the critical section when the value of $\_lock$ is $false$.
  When a thread enters the critical section, it will set the value of $\_lock$ as $true$ so that no other thread can enter the critical section.
  When the thread exits the critical section, it reset the value of $\_lock$ as $false$.

  Parameters of each thread are passed by an object of the class $params\_TAS$ which has following parameters:
  - *id:* Thread id
  - *k:* Number of times to enter critical section
  - *l1:* Mean of sleep time of critical section of thread
  - *l2:* Mean of sleep time of remainder section of thread

- **Compare and Swap (CAS):**
  This algorithm is a more sophisticated mutual exclusion algorithm that uses a multi-step process to ensure mutual exclusion.
  This algorithm can be implemented atomically using $std::atomic\_bool::compare\_exchange\_strong()$ in $C++$.

  A mutex lock variable $\_lock$ is declared globally and initialized to $false$.
  Each thread tries to compare the value of the variable $\_lock$ to a known $expected$ value which is $false$. If comparison succeeds, the thread sets the lock variable to a new_value which is $true$ and enters the critical section. Therefore, no other thread can enter the critical section while any one of the threads is in its critical section.
  When the thread exits the critical section, it reset the value of $\_lock$ as $false$.

Parameters of each thread are passed by an object of the class $params\_CAS$ which has following parameters:

- *id:* Thread id
- *k:* Number of times to enter critical section
- *l1:* Mean of sleep time of critical section of thread
- *l2:* Mean of sleep time of remainder section of thread

- **Bounded Compare and Swap(BCAS):**

This algorithm ensures mutual exclusion, progress requirement and bounded waiting.
This algorithm can be implemented atomically using $std::atomic\_bool::compare\_exchange\_strong()$ along with a few lines of code in $C++$.

A mutex lock variable $\_lock$ is declared globally and initialized to $false$.
An array $waiting$ containing n booleans is declared globally and initialized to $false$. It represents the state of the threads in the sense that if $waiting_i$ is $false$, that means the ith thread is inside the critical section. If it is $true$, that means ith thread is waiting to enter the critical section.
If $waiting_i$ is $false$ or $\_lock$ is $false$, the thread will enter the critical section.
After the thread exits the critical section, it scans the array in cyclic ordering $(i+1, i+2, \ldots, n-1, 0, \ldots, i-1)$. It designates the first thread in this ordering, that is in the entry section $(waiting[j] == true)$, as the next one to enter the critical section.

Parameters of each thread are passed by an object of the class $params\_BCAS$ which has following parameters:

- *id:* Thread id
- *k:* Number of times to enter critical section
- *n:* Total number of threads.
- *l1:* Mean of sleep time of critical section of thread
- *l2:* Mean of sleep time of remainder section of thread