Mahdeen Ahmed Khan Sameer

Professor Max Bender

CS333

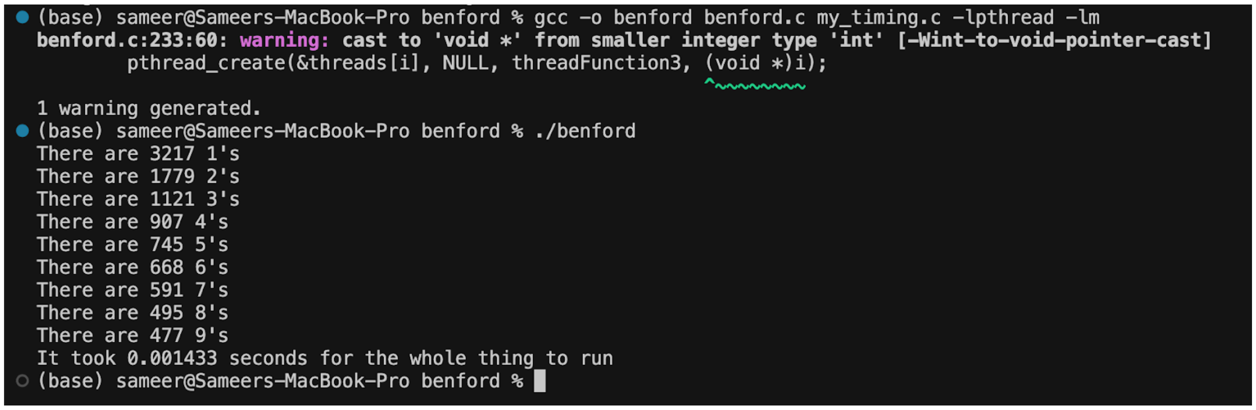
May 2024

Project 08: Parallel Programming in Programming Languages

Google site: https://sites.google.com/colby.edu/sameer-again/home

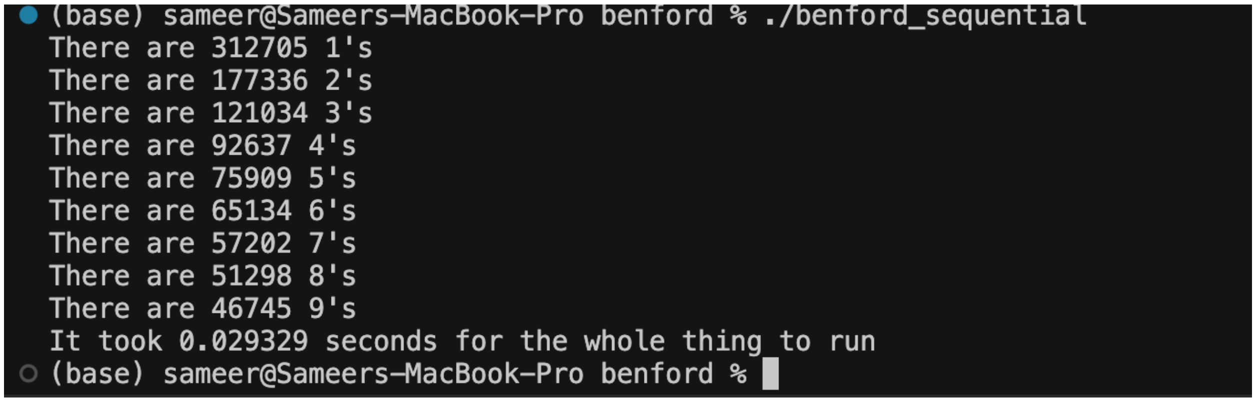
Task 01:

After running all six different parallelized versions of the Benford program with the longer.bin file ten times, dropping the minimum and maximum data points, and calculating the average computation times, we summarized the results in the following table.



```
● (base) sameer@Sameers-MacBook-Pro benford % gcc -o benford benford.c my_timing.c -lpthread -lm
  benford.c:233:60: warning: cast to 'void *' from smaller integer type 'int' [-Wint-to-void-pointer-cast]
          pthread_create(&threads[i], NULL, threadFunction3, (void *)i);
                                                             ^~~~~~~~~

  1 warning generated.
● (base) sameer@Sameers-MacBook-Pro benford % ./benford
  There are 3217 1's
  There are 1779 2's
  There are 1121 3's
  There are 907 4's
  There are 745 5's
  There are 668 6's
  There are 591 7's
  There are 495 8's
  There are 477 9's
  It took 0.001433 seconds for the whole thing to run
○ (base) sameer@Sameers-MacBook-Pro benford %
```



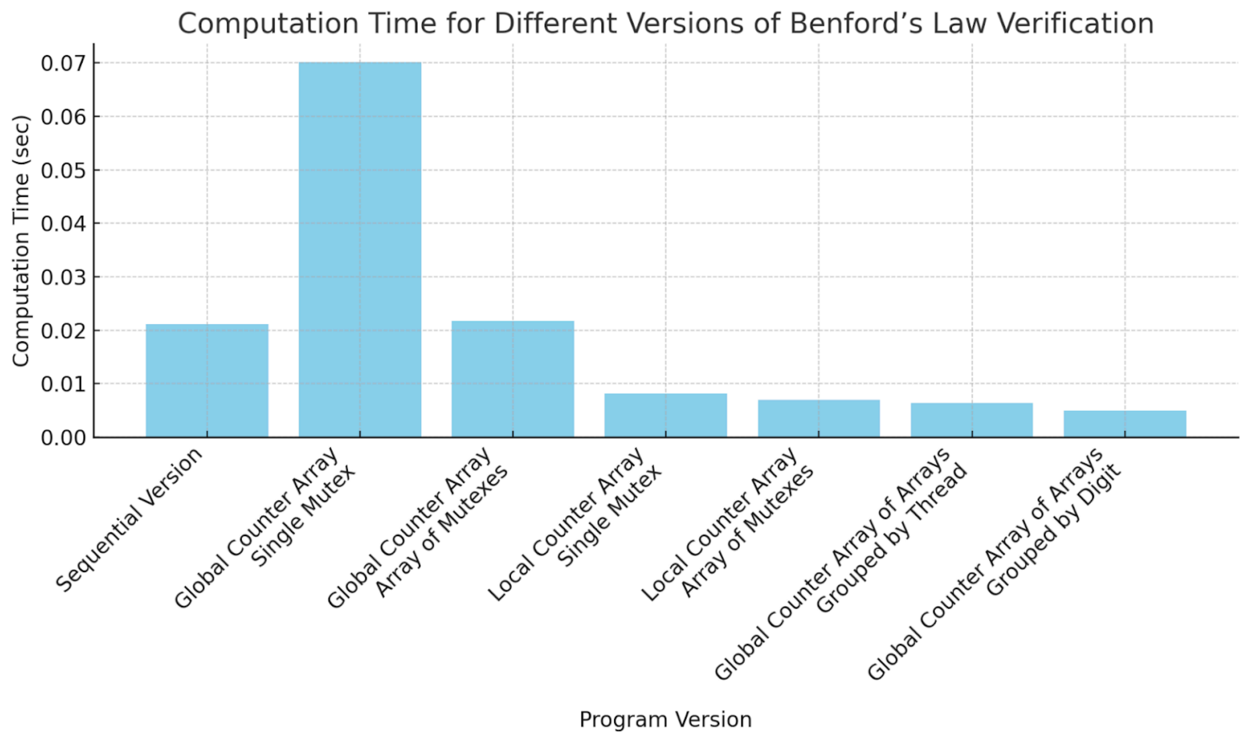
```
● (base) sameer@Sameers-MacBook-Pro benford % ./benford_sequential
  There are 312705 1's
  There are 177336 2's
  There are 121034 3's
  There are 92637 4's
  There are 75909 5's
  There are 65134 6's
  There are 57202 7's
  There are 51298 8's
  There are 46745 9's
  It took 0.029329 seconds for the whole thing to run
○ (base) sameer@Sameers-MacBook-Pro benford %
```

- Sequential Version (benford_sequential): 0.0211 seconds
- Global Counter Array Protected by Single Mutex: 0.0701 seconds
- Global Counter Array Protected by Array of Mutexes: 0.0217 seconds

- Local Counter Array, with Final Update Protected by Single Mutex: 0.0081 seconds
- Local Counter Array, with Final Update Protected by Array of Mutexes: 0.0070 seconds
- Global Counter Array of Arrays, Grouped by Thread, no Mutex: 0.0064 seconds
- Global Counter Array of Arrays, Grouped by Digit, no Mutex: 0.0050 seconds

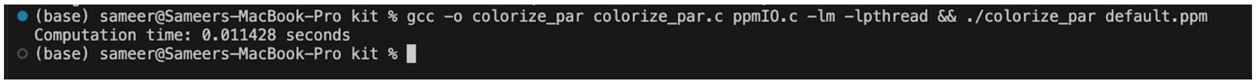
The graph generated from this data, excluding the sequential version, reveals some interesting insights. Nearly all parallel versions outperform the sequential version, except for the first parallel version using a global counter array protected by a single mutex. The explanation is straightforward: with a single mutex, all threads compete for access to update the global counter array, leading to significant synchronization overhead as they frequently lock and unlock the mutex, causing contention and waiting. As we examine other parallel versions, the impact of mutex locks becomes apparent. The version with a global counter array protected by an array of mutexes achieves better concurrency than using a single mutex but still experiences some overhead from managing multiple mutexes. Although this version often performs better than the sequential one, some runs took longer due to the mutex management overhead.

The versions using local counter arrays with final update protection by either a single mutex or an array of mutexes show marked performance improvements. The latter is slightly more efficient because each thread maintains its own local counter array, minimizing contention for shared resources. Unlike the global counter array versions, where threads compete for a single shared resource, the local counter array versions distribute the workload and assign independent data structures to each thread, reducing contention. The last two versions are the fastest. These versions do not use mutexes but instead organize work more intelligently. Each thread maintains its own storage space for counting leading digits, eliminating the need for threads to wait for each other. This independence and efficiency reduce overhead time and contribute to faster performance.



Overall, mutex locks are crucial for protecting critical sections where shared resources are accessed and modified concurrently by multiple threads. In this task, mutex locks ensure safe updates to the global counter array, preventing data corruption and incorrect results from race conditions. However, using a single mutex can cause excessive wait times for threads, potentially slowing the program more than expected. Thus, while mutex locks are essential for concurrent access to shared resources, alternative solutions, like those used in the final two versions, can achieve better performance without mutex overhead.

Task 02:



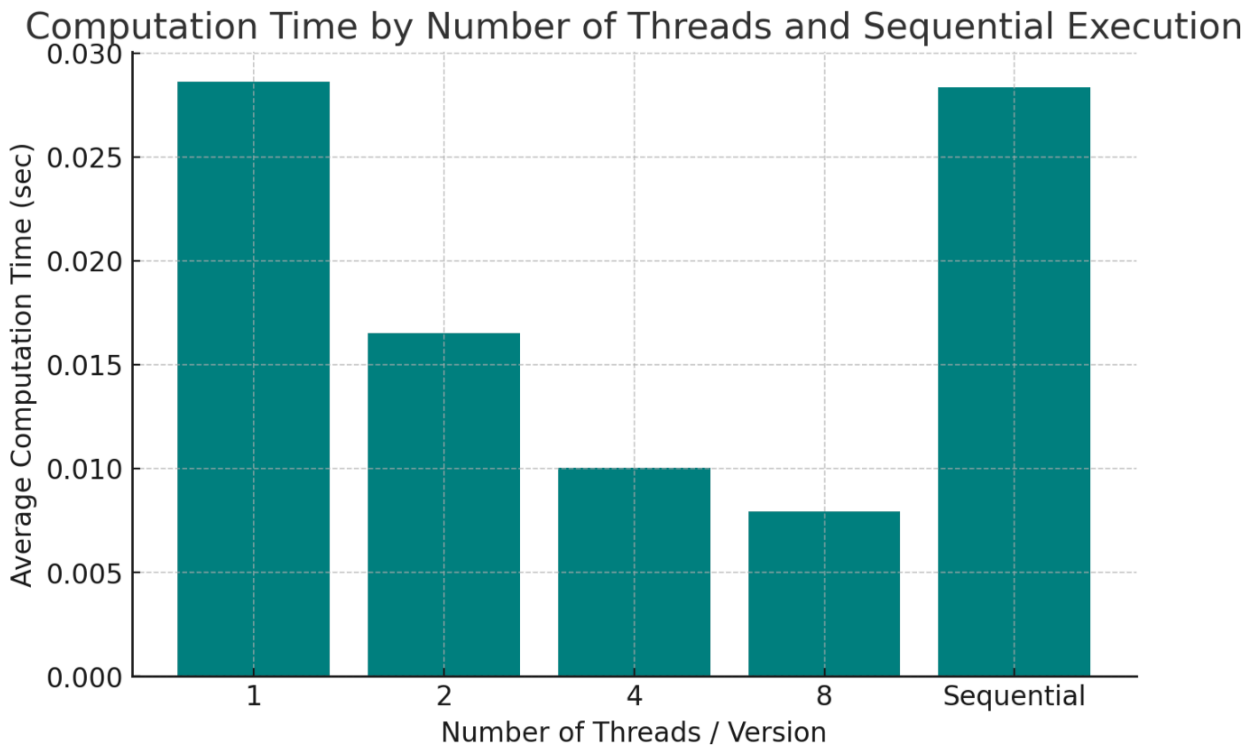
```
● (base) sameer@Sameers-MacBook-Pro kit % gcc -o colorize_par colorize_par.c ppmIO.c -lm -lpthread && ./colorize_par default.ppm
  Computation time: 0.011428 seconds
○ (base) sameer@Sameers-MacBook-Pro kit % █
```

Just as we utilized pthreads to parallelize the previous task, we applied the same approach to this task to implement a pixel-wise operator for any read image. The average computation times, determined by recording five trials, discarding the highest and lowest values, and averaging the remaining three, are summarized in the table below along with the number of threads used in each experiment.

1 Thread:	0.027310 seconds
2 Threads:	0.011911 seconds
4 Threads:	0.010185 seconds
8 Threads:	0.00779 seconds
Sequential Version:	0.029333 seconds

The average computation time for the sequential version and the parallel version using a single thread is identical, which is expected since using just one thread is equivalent to running the program sequentially. In fact, the single-threaded version may be slightly slower due to the overhead involved in creating the thread.

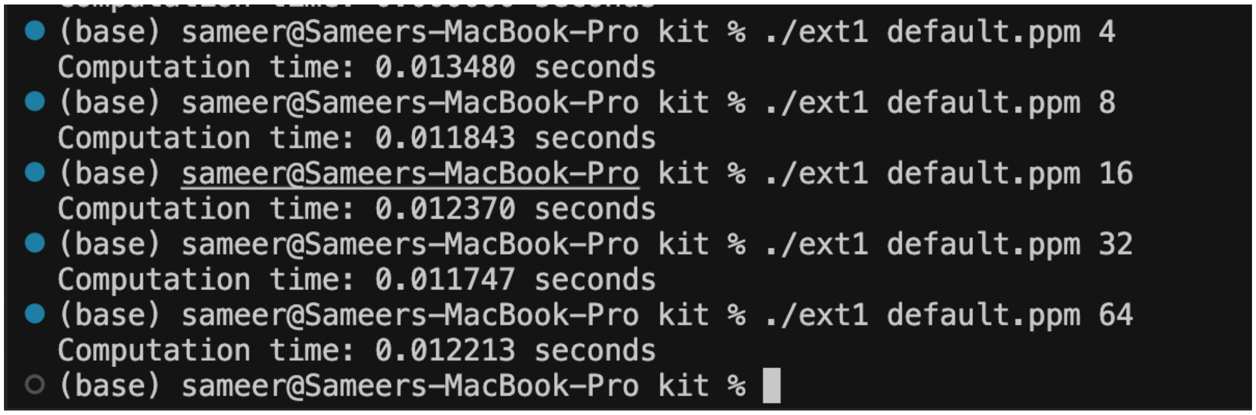
As the number of threads increases, the average computation time decreases, as illustrated more clearly in the graph below:



While the run time doesn't halve each time the number of threads is doubled, the trend is consistent and clear: increasing the number of threads reduces the average computation time. Although the instructions only required data for up to 4 threads, I also experimented with 8 threads. The computation time decreased further, but the improvement was less significant. Presumably, adding more threads beyond this point will have an even smaller impact—or potentially a negative effect—on the average computation time of the program.

Extension 1:

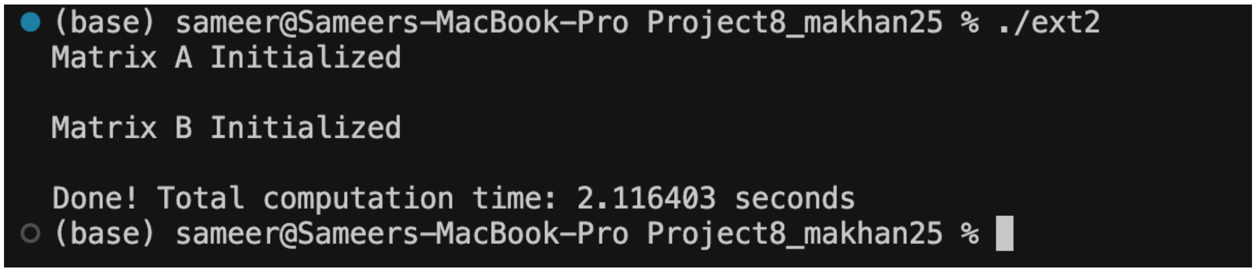
For our first extension, we modified the "colorize_par.c" file to accept the number of threads as a command-line argument, streamlining the process of adding or removing threads without needing to recompile. The "NUM_THREADS" macro was removed, and the number of threads is now read from "argv[2]". We updated the "ThreadArgs" struct to include the "num_threads" variable, which is passed to the "processSegment" function. The function's calculation of start and end indices now uses this value. In the main function, "ThreadArgs" is updated with "num_threads" before creating the threads. Testing showed consistent results for average computation times, validating the modifications. Further experimentation confirmed that adding more than 8 threads yields diminishing returns; while 10 threads provided a slight improvement, any additional threads resulted in negligible or no further reduction in run time. As shown in the provided data, computation times were as follows: 4 threads (0.013480 seconds), 8 threads (0.011843 seconds), 16 threads (0.012370 seconds), 32 threads (0.011747 seconds), and 64 threads (0.012213 seconds).



```
(base) sameer@Sameers-MacBook-Pro kit % ./ext1 default.ppm 4
Computation time: 0.013480 seconds
(base) sameer@Sameers-MacBook-Pro kit % ./ext1 default.ppm 8
Computation time: 0.011843 seconds
(base) sameer@Sameers-MacBook-Pro kit % ./ext1 default.ppm 16
Computation time: 0.012370 seconds
(base) sameer@Sameers-MacBook-Pro kit % ./ext1 default.ppm 32
Computation time: 0.011747 seconds
(base) sameer@Sameers-MacBook-Pro kit % ./ext1 default.ppm 64
Computation time: 0.012213 seconds
(base) sameer@Sameers-MacBook-Pro kit %
```

Extension 2:

For our extension 02, we developed another program utilizing parallel programming to perform matrix multiplication. This parallel matrix multiplication program divides the task into smaller chunks (chunk_size = M / NUM_THREADS) and assigns each chunk to a separate thread for concurrent computation. Initially, the program initializes two matrices, A and B, with random values. It then creates multiple threads, with each thread responsible for processing a segment of the resulting matrix. Each thread computes a subset of the rows in the result matrix by multiplying the corresponding rows of matrix A with columns of matrix B. Finally, the program combines the results from each thread to produce the final result matrix.



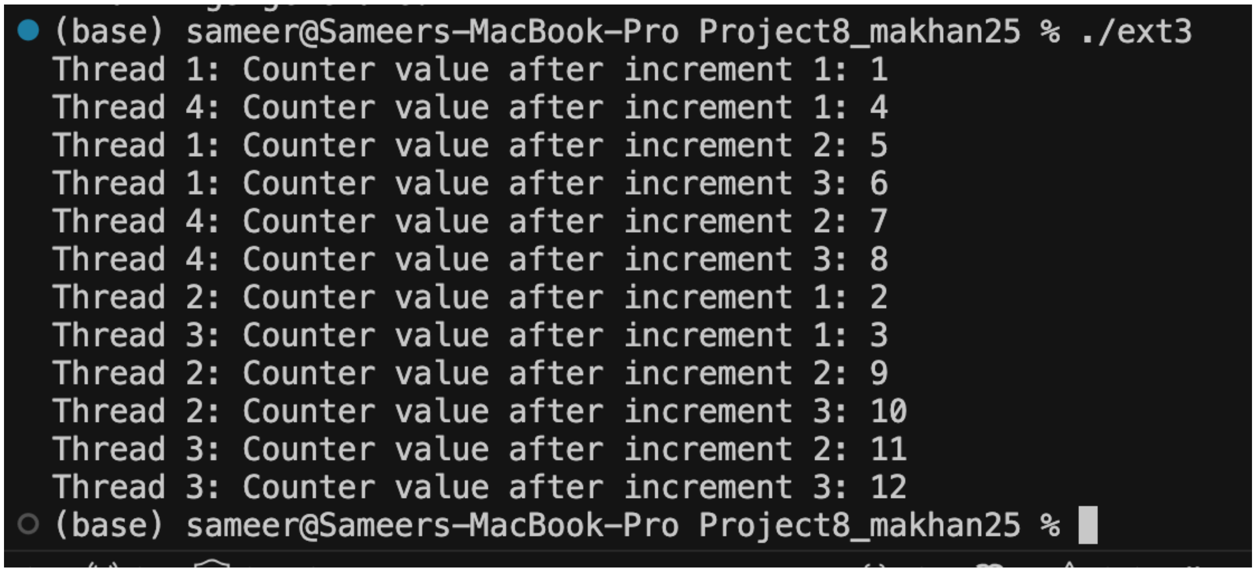
```
(base) sameer@Sameers-MacBook-Pro Project8_makhan25 % ./ext2
Matrix A Initialized

Matrix B Initialized

Done! Total computation time: 2.116403 seconds
(base) sameer@Sameers-MacBook-Pro Project8_makhan25 %
```

Extension 3:

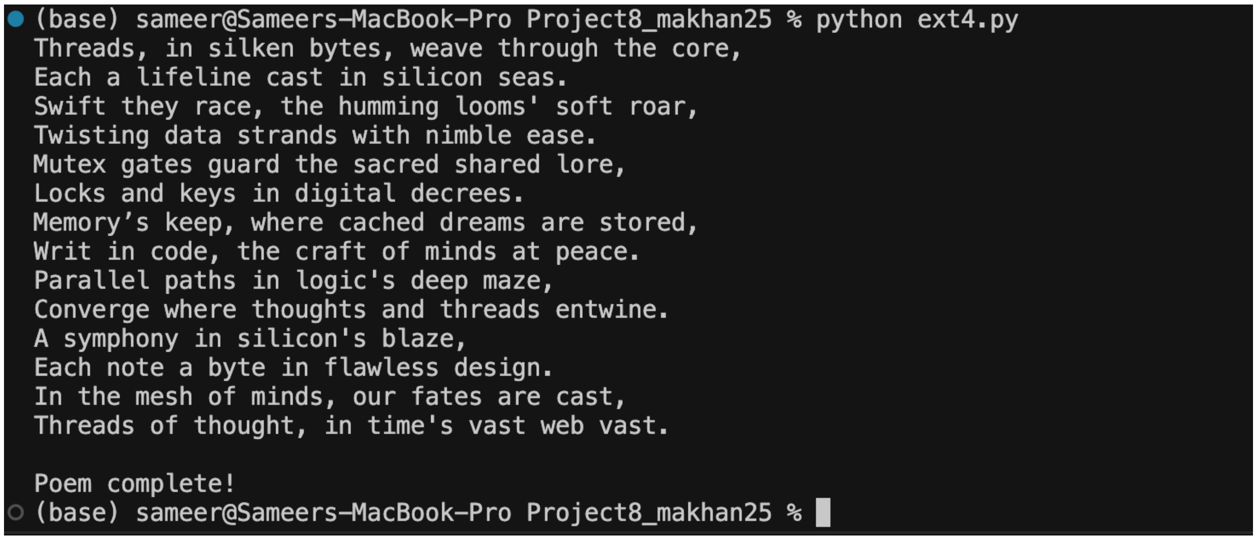
Again, we explored the concept of semaphores and their application in C parallel programming. A semaphore is a synchronization primitive that manages access to critical sections by multiple threads, acting as a counter to allow a specified number of threads to access a resource simultaneously while blocking additional threads beyond this limit. Semaphores in C use two main operations: "sem_wait()" and "sem_post()". The "sem_wait()" function decrements the semaphore value by 1 and blocks the calling thread if the value becomes negative, indicating that the resource is in use. When the resource is available, the thread is unblocked. The "sem_post()" function increments the semaphore value by 1, signaling that the resource is available for other threads. Unlike mutex locks, which only allow one thread to access a resource at a time, semaphores permit multiple threads up to a defined limit. To test this, we wrote a basic parallel program where four threads increment a shared counter. Each thread acquires the semaphore with "sem_wait()", increments the counter, and releases the semaphore with "sem_post()". By limiting access to two threads at a time, the output shows that the threads increment the counter in a specific order, ensuring proper sequence and synchronization as managed by the semaphore.



```
(base) sameer@Sameers-MacBook-Pro Project8_makhan25 % ./ext3
Thread 1: Counter value after increment 1: 1
Thread 4: Counter value after increment 1: 4
Thread 1: Counter value after increment 2: 5
Thread 1: Counter value after increment 3: 6
Thread 4: Counter value after increment 2: 7
Thread 4: Counter value after increment 3: 8
Thread 2: Counter value after increment 1: 2
Thread 3: Counter value after increment 1: 3
Thread 2: Counter value after increment 2: 9
Thread 2: Counter value after increment 3: 10
Thread 3: Counter value after increment 2: 11
Thread 3: Counter value after increment 3: 12
(base) sameer@Sameers-MacBook-Pro Project8_makhan25 %
```

Extension 4:

In our fourth extension, we created a simple program to metaphorically illustrate the concept of parallel programming through the generation and display of a poem. Each line of the poem is processed by a separate thread, with a slight delay added to simulate the time taken for each thread to complete its task. By creating and starting a thread for each line, and then joining these threads to ensure the main program waits for all to finish, the script demonstrates how multiple tasks (or threads) can operate concurrently, ultimately contributing to a unified output. This approach highlights the parallel execution of tasks, akin to how threads in computing work together to achieve a common goal.



```
(base) sameer@Sameers-MacBook-Pro Project8_makhan25 % python ext4.py
Threads, in silken bytes, weave through the core,
Each a lifeline cast in silicon seas.
Swift they race, the humming looms' soft roar,
Twisting data strands with nimble ease.
Mutex gates guard the sacred shared lore,
Locks and keys in digital decrees.
Memory’s keep, where cached dreams are stored,
Writ in code, the craft of minds at peace.
Parallel paths in logic's deep maze,
Converge where thoughts and threads entwine.
A symphony in silicon's blaze,
Each note a byte in flawless design.
In the mesh of minds, our fates are cast,
Threads of thought, in time's vast web vast.

Poem complete!
(base) sameer@Sameers-MacBook-Pro Project8_makhan25 %
```

Extension 5:

In our next extension, we used multiple threads to increment a shared counter, with a mutex lock ensuring that only one thread can access the critical section (incrementing the counter) at a time. Each thread runs a function that attempts to increment the counter three times, acquiring the mutex lock before each increment to guarantee exclusive access and releasing it afterward to allow other threads to proceed. The main program creates and starts four threads, waits for all to finish using the join() method, and then prints the final counter value. We hope this demonstrates how a mutex lock effectively prevents race conditions by controlling access to a shared resource in a multithreaded environment.

Extension 6:

In our extension 6, we demonstrated how matrix multiplication is parallelized using the "threading" module to enhance computational efficiency. Matrices "A" and "B" are initialized with random values, and the resulting matrix "C" is computed in parallel by dividing the rows of "A" among multiple threads. Each thread calculates the product of its assigned subset of rows from "A" with the entire matrix "B", storing the results in "C". The script includes a function to manage thread creation and execution, distributing the workload across 1, 2, 4, and 8 threads, and measures the computation time for each configuration. Easy!

Collaborators & Acknowledgements:

Professor Bendor, TA Nafis.