CS 610 Semester 2025–2026-I: Assignment 2

23rd August 2025

Due Your assignment is due by Sep 1, 2025, 11:59 PM IST.

General Policies

- You should do this assignment ALONE.
- Do not copy or turn in solutions from other sources. You will be PENALIZED if caught.

Submission

- Submission will be through Canvas.
- Submit a compressed file called "⟨`roll`⟩`-assign2.tar.gz`". The compressed file should have the following structure.

```
-- rollno
-- -- rollno-assign2.pdf
-- -- <problem1-dir>
-- -- -- <rollno-prob1.cpp
-- -- -- <other-source-files>
-- -- <problem2-dir>
-- -- -- <rollno-prob2.cpp
-- -- -- <other-source-files>
-- -- <problem3-dir>
-- -- -- <rollno-prob3.cpp
-- -- -- <other-source-files>
```

  The PDF file should contain descriptions and results for the three problems. Rename the attached source files appropriately.

- We encourage you to use the LaTeX typesetting system for generating the PDF file. You can use tools like Tikz, Inkscape, or Drawio for drawing figures if required. You can alternatively upload a scanned copy of a handwritten solution, but MAKE SURE the submission is legible.
- You will get up to TWO LATE days to submit your assignment, with a 25% penalty for each day.

Evaluation

- Write your programs such that the EXACT output format (if any) is respected.
- We will evaluate your implementations on recent Debian-based distributions, for example, KD first floor lab.
- We will evaluate the implementations with our inputs and test cases, so remember to test thoroughly.

Problem 1 **[40 marks]**

(i) Write a C++ program to implement a naïve 3D convolution operation. The naïve function should implement the following computation.

$$O[i][j][k] = \sum_{u=0}^{K-1} \sum_{v=0}^{K-1} \sum_{w=0}^{K-1} I[i+u][j+v][k+w] \times F[u][v][w]$$

where I is a $N \times N \times N$ input array and F is the $K \times K \times K$ filter (or kernel). O is the 3D output array (or feature map), and each dimension of C is of size $N - K + 1$ (no padding). Ignore flipping the filter.

(ii) Implement a blocked version of the convolution operation. You should focus on blocking the input and output arrays, and size the blocks for private L2 caches. Make the block size configurable because you will evaluate different sizes.

1. Experiment with different block sizes (individual dimensions can be different) and report the block size that gives the best performance for the blocked kernel. Automatically exploring different block sizes and zeroing in on the best setup is called autotuning.
2. Run the naïve and the blocked kernels at least five times and report the average times taken.
3. Use the PAPI library to estimate the number of cache misses for the data cache hierarchy on your target architecture to justify the performance that you observed.

Problem 2 **[50 marks]**

You are given a multithreaded program with `N` threads. The program reads `N` files, and should report the total number of words and lines processed by all the threads.

We provide a driver source code for the problem. Your task is to analyze the source code, and identify and report the performance bugs present in the source code, if any. If a performance bug is identified, provide a manually fixed version. Describe your modifications to the source code and report the performance gain.

Use the following commands to compile the attached driver and run the sample test case.

```
make
./bin/problem2.out 5 ./prob2-test1/input
```

You can use the `perf c2c` tool to identify some forms of performance bugs. You can use the following links to learn more about using `perf`.

- C2C - False Sharing Detection in Linux Perf
- perf c2c man page

You may need to modify the value of `perf_event_paranoid` setting on Linux for collecting useful metrics with `perf`.

Input The input to your program will be a path to a file, say `input`. The file `input` lists the full paths of `N` source files that are to be processed by `N` threads. Read the contents of the file `input` into a shared data structure `X`. Each thread will then pick *one* file from the shared data structure to analyze. A thread reads its file one line at a time and divides the line into tokens. The thread updates a counter to track the words encountered by the threads and updates the shared variable to track the total number of places the line.

Example

```
File 1:
ABC, EFG HIJK.
LMNOP QRST.

File 2:
ABC EF HI LMNOPQ
RST UV

Expected Output:
Thread 0 counter: 6
Thread 1 counter: 5
Total words processed: 11
Total lines processed: 4
```

Problem 3 **[50 marks]**

Implement the following locking strategies in C++: Filter lock, Bakery lock, Spin lock, Ticket lock, and Array-based Queue lock. You will evaluate the performance of your lock implementations using the following code snippet.

```
// The following loop nest will be executed in parallel
for (int i = 0; i < N; i++) {
   acquire(&lock);
   // The following variables are shared
   var1++;
   var2--;
   release(&lock);
}
```

You should use CAS-like APIs available on x86_64 architectures to implement your version of the locks. Do not use lock definitions from existing libraries. Furthermore, you should try to avoid false sharing for implementations that make use of arrays.

Evaluate the above code snippet with 1–64 concurrent threads, in powers of two. Report the total time taken while using each lock type while varying the number of threads. Compare the performance of your lock implementations with `pthread_mutex_t` lock and unlock APIs as the baseline.

We have included a template C++ file that you will extend. Create separate classes for the above lock types, similar to `PthreadMutex`, and fill in the public `acquire()` and `release()` methods. Include all the definitions in the provided "`problem3.cpp`" file.

Fill in the following table with your results. Report any trends that you observe from the results.

	1	2	4	8	16	32	64
Pthread mutex							
Filter lock							
Bakery lock							
Spin lock							
Ticket lock							
Array Q lock							