Roll no: 241110406

Assignment 2

Problem 1:

Code in problem 1 introduces a false sharing problem between multiple threads operating on the same variables / shared variables.
The structure variable *word_tracker,* defined as:

```
struct word_tracker {
        uint64_t word_count[4];
        uint64_t total_lines_processed;
        uint64_t total_words_processed;
        pthread_mutex_t word_count_mutex;
};
```

It has an instance *tracker* whose attributes are shared among all the consumer threads that consume data from a shared buffer *std::queue<std::string> shared_pq.* Multiple threads are able to simultaneously read and update their corresponding element of the word_count array attribute of the tracker variable.
If the user threads are handled by multiple different kernel threads running on different cores of the CPU, which have their own private cache, the same shared variable being loaded in multiple private caches will cause many extra invalidation requests sent back and forth between cores in case of any update in the value of the shared variable.
This phenomenon may be the result of both True sharing, (example: mutex variables), and false sharing (example: shared array variable).
Given: Cache line size is 64 bytes.

Potential variables that can cause false sharing:

- *tracker.word_count[4]* :

  If all elements of the array fall in the same cache line, any update to one element in the private cache of one core will cause invalidation of its copy in another core which is running another thread which accesses another element of the same array.
  Solution:
  Introduce padding/alignment in the array such that only one element falls in each cache line.

  ```
  alignas(64) uint64_t *word_count;
  tracker.word_count = (uint64_t*)aligned_alloc(64, thread_count*sizeof(uint64_t));
  ```

- *struct word_tracker {*
  alignas(64) uint64_t *word_count;
  uint64_t total_lines_processed;
  uint64_t total_words_processed;
  pthread_mutex_t word_count_mutex;
  } tracker;

  If any two or more variables of struct lie on the same cache line in private caches of different cores, updating any one of the variables will cause invalidation of the copy of struct in another core's cache. Example: if core c1 updates the variable tracker.total_lines_processed and core c2 updates tracker.total_words_processed that would still cause invalidation due to false sharing.
  Solution:
  Introduce padding/alignment between variables such that they lie on different cache lines.

```
struct word_tracker {
        alignas(64) uint64_t *word_count;
        uint64_t total_lines_processed;
        uint64_t total_words_processed;
        pthread_mutex_t word_count_mutex;
} tracker __attribute__((aligned(64)));
```

  Although it would not solve the problem of true sharing due to these variables. Example: if both c1 and c2 cores update the same variable tracker.total_words_processed.

- *std::queue<std::string> shared_pq:*

  Multiple cores will load shared_pq in their cache if a user thread running on them reads it. Any update to a copy of this buffer variable in one core's cache will invalidate the copy in another core's cache. BUT this invalidation will not lead to performance loss because each thread will read it only once. No modification needed here.

Problem 2:

The program has got N producer threads each of which read L consecutive lines from the input file and try to fill the buffer which can store M lines simultaneously. A consumer thread constantly reads from the buffer, clears it and writes to an output file. Input file path, output file path, N, L, and M are input parameters of the program.

Implementation is based on the bounded buffer problem related to critical section in operating systems.

One of the requirements of the program is that a producer should produce/read all its L lines together without any other producer interrupting the process. But when buffer size is smaller than the no. of lines to be read by each producer thread, i.e. M < L, the producer would have to inadvertently stop for consumer thread to clear the buffer and consumer should pass the control exclusively to the thread which was already reading lines from thread. This is achieved by storing the half complete thread's thread id in a global variable and using a condition variable to check if the next thread reading from the input when the consumer clears space in the buffer is the original thread or not. If not, the condition variable makes it go to sleep. When the original thread that was already executing completes, it signals all such threads that were blocked and wakes them.

Problem 3:

Consider the following loop nest.

```
for i = 1, N-2
  for j = i+1, N
    A(i, j-i) = A(i, j-i-1) - A(i+1, j-i) + A(i-1, i+j-1)
```

Let,
A(i, j - i - 1) = X
A(i + 1, j - i) = Y
A(i - 1, i + j - 1) = Z and
A(i, j - 1) = X - Y + Z = R

True Dependency between R and X:

Given, i = i + Δi
And j - i = j - i - 1 - Δi + Δj

=> Δi = 0 and Δj = 1
=> direction vector = (O, +)
=> **Flow dependency exists between A(i, j - i) and A(i, j - i - 1)**

True Dependency between R and Y:

Given, i = i + 1 + Δi
And j - i = j + Δj - i - Δi

=> Δi = -1 and Δj = -1
=> direction vector = (-, -), Not a valid true dependency
∴ No true Dependency

Anti Dependency between R and Y:

For anti dependency sink should exist in a previous iteration than source, therefore Δi would have negative sign.

Given, i = i - Δi + 1
And j - i = j - Δj - i + Δi

=> Δi = 1 and Δj = 1
=> direction vector = (+, +)
=> **Anti dependency exists between A(i, j - i) and A(i + 1, j - i)**

True Dependency between R and Z:

Given, i = i - 1 + Δi
And j - i = i + j - 1 + Δi + Δj

=> $\Delta i = +1$ and $\Delta j = -2i$

=> direction vector = (+, -) *(assuming i will take only positive values)*

=> **Flow dependency exists between A(i, j - i) and A(i + 1, j - i)**

No output dependency exists here because it requires two writes (ordering of write operations matter) to the same memory location, but only one write operation exists here.