Implementation Of A Multithreaded Word Counter

Name: Swoyam Pokharel
Student Number: 2431342
Tutor: Bijaya Ghimire

Submitted On: December 17, 2025

Overview

This document provides a comprehensive examination of a multithreaded word counting application implemented in C using the POSIX Threads (Pthread) library. The program efficiently processes large text files by distributing the workload across multiple threads through round-robin scheduling. At its core, a trie data structure handles word storage and frequency counting, with thread safety ensured through fine-grained mutex locking at the node level.

Contents

1 · Data Structures

The program's foundation rests on two primary data structures: the trie tree for managing word storage and counts, and the thread data structure for coordinating parallel execution across multiple workers.

1.1 · Trie Tree

The trie, commonly referred to as a prefix tree, serves as the core mechanism for storing words and tracking their occurrence frequencies. Unlike alternative approaches like hash tables or simple arrays, the trie provides automatic alphabetical ordering and memory-efficient storage for words that share common prefixes. The structure itself is elegant in its simplicity: each node represents a character, paths from the root spell out complete words, and a special flag marks word boundaries.

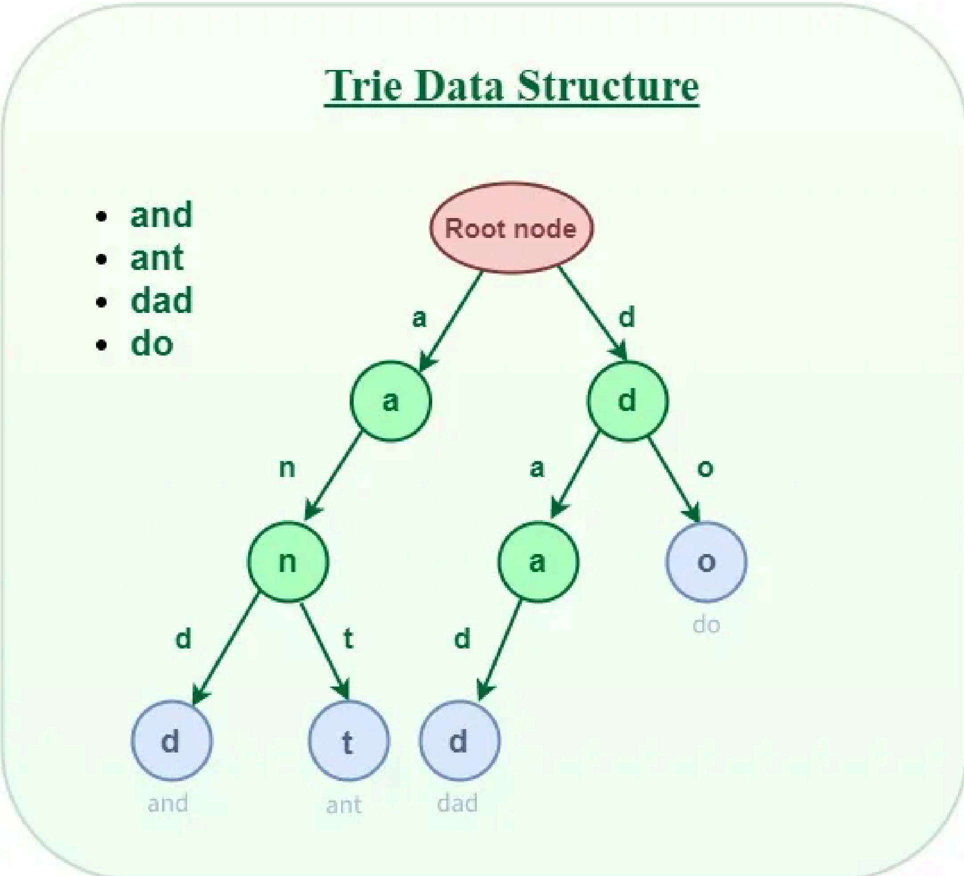


Figure 1: Trie Tree Example

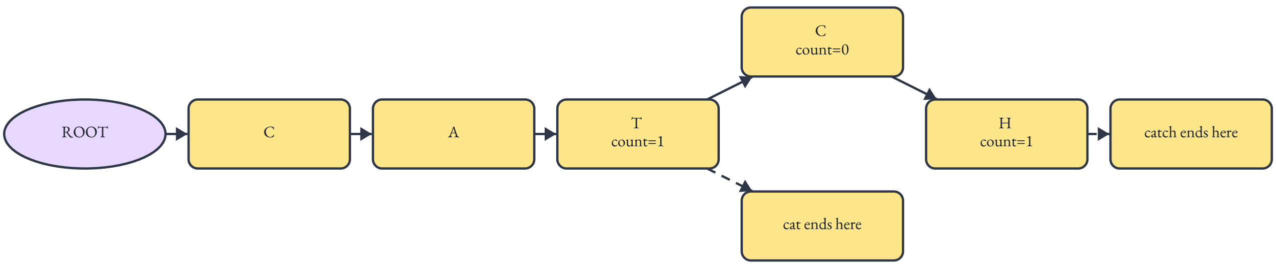
1.1.1 · Structure & Components

Each node in the trie represents a single character position within a word and contains three essential components that work together to enable efficient concurrent access:

```
typedef struct TrieNode {
  struct TrieNode* children[ALPHABET_SIZE];  // 26 pointers (a-z)
  int count;                                  // Word frequency
  pthread_mutex_t lock;                       // Thread safety
} TrieNode;
```

The `children` array maintains 26 elements, one for each lowercase letter from 'a' to 'z'. This design enables constant-time `O(1)` lookup for the next character in a word by calculating the array index as `character - 'a'`. The mapping is straightforward: 'a' corresponds to index 0, 'b' to index 1, and 'z' to index 25. This direct indexing eliminates the need for comparisons or hash computations, making character-by-character traversal extremely fast.

The `count` field tracks how many times a complete word ending at this particular node has been encountered during insertion. A count of zero indicates this node exists as part of a word's path but does not itself mark the end of a valid word. Consider inserting "cat" and "catch" into the trie: the node after 't' in "cat" would have a non-zero count marking it as a complete word, while the intermediate nodes in "catch" would maintain zero counts until reaching the final 'h'.



The `lock` mutex provides thread-safe access to each individual node. Rather than using a single global lock for the entire trie, which would create a bottleneck forcing threads to wait for each other unnecessarily, each node maintains its own lock. This fine-grained locking approach enables multiple threads to work on different parts of the trie simultaneously, which helps for achieving meaningful parallelization benefits.

1.1.2 · Why Trie (automatic sorting, memory efficiency)

The decision to use a trie over alternative data structures like hash tables or arrays comes from several important advantages that align well with this application's requirements:

- **Automatic Alphabetical Sorting**: Since we are tasked to sort the final result either alphabetically or by count, the trie tree makes sense because it's inherent structure maintains alphabetical order without any additional effort. When traversing children from index `[0-25]`, we naturally visit nodes in alphabetical sequence. This eliminates the need for a separate sorting phase that would be required with a hash table or unsorted array, making the output writing phase a simple `O(n)` traversal. The sorted output comes for free as a consequence of the data structure's design.

- **Thread-Safe Expansion:** The trie naturally accommodates concurrent insertions through fine-grained locking. Each path through the trie can be locked independently, allowing multiple threads to insert different words simultaneously without interfering with each other. This property makes the trie particularly well-suited for multithreaded word counting, as threads rarely contend for the same nodes unless processing very similar words.
- **Memory Efficiency for Shared Prefixes:** Words sharing common prefixes share the same nodes in memory. In a hash table, each word would require separate storage even for these shared prefixes. For large dictionaries containing many similar words, trie trees can result in significant memory savings. Given that our input dataset is substantially sized, this efficiency matters:

```
[wizard@archlinux ~/Projects/School/HPC/Assessment/t1/assets] -> wc -w
WordOccurrenceDataset.txt
120000 WordOccurrenceDataset.txt
[wizard@archlinux ~/Projects/School/HPC/Assessment/t1/assets] ->
```

With 120,000 words, the memory savings from shared prefixes become meaningful, particularly for common prefixes that appear across thousands of words.

1.1.3 · Operations (Insert, Write)

Word Insertion:
The insertion process walks through the trie character by character, creating nodes as needed along the path. Consider inserting the word "apple" into an empty trie:

1. Start at root, look for child 'a' (index 0)
2. Move to 'a' node, look for child 'p' (index 15)
3. Move to 'p' node, look for child 'p' (index 15)
4. Move to second 'p' node, look for child 'l' (index 11)
5. Move to 'l' node, look for child 'e' (index 4)
6. At final 'e' node, increment count

If any child doesn't exist during this traversal, it must be created immediately. This is where we encounter our critical section, the point where multiple threads might attempt to create the same child node simultaneously, potentially causing a race condition.

```c
void insert_word_into_trie(TrieNode* root, const char* word) {
  TrieNode* current = root;

  for (int i = 0; word[i] != '\0'; i++) {
    char c = tolower(word[i]);

    // Skip non-alphabetic ( if any )
    if (c < 'a' || c > 'z') continue;

    int index = c - 'a';

    if (current->children[index] == `NULL`) {
      pthread_mutex_lock(&current->lock);
```

```
      if (current->children[index] == `NULL`) {
        current->children[index] = create_trie_node();
      }
      pthread_mutex_unlock(&current->lock);
    }

    current = current->children[index];
  }

  pthread_mutex_lock(&current->lock);
  current->count++;
  pthread_mutex_unlock(&current->lock);
}
```

The Double-Check Locking Pattern

Notice the code checks if `children[index] ==` NULL twice. This might appear redundant at first glance, but it's actually a crucial optimization pattern called double-check locking that balances correctness with performance.

The first check, performed without acquiring the lock, allows threads to quickly skip the locking mechanism when a child already exists. Acquiring and releasing locks is computationally expensive, involving system calls and potential context switches, so avoiding locks when possible provides significant performance benefits.

However, if the first check finds the child is NULL, we need to create it. But here's where the problem emerges: between the first check and acquiring the lock, another thread might have already created that same child node. This creates a classic race condition.

Consider this scenario with two threads inserting "apple" and "apply" simultaneously:

- Thread 1 checks: `children[p]` is NULL
- Thread 2 checks: `children[p]` is NULL
- Thread 1 acquires lock, creates the `p` node, releases lock
- Thread 2 acquires lock, but without the second check, it would create another `p` node, overwriting Thread 1's node and losing all the data stored in it

The second check, performed inside the lock after acquisition, prevents this race condition. After acquiring the lock, we verify that no other thread has created the child while we were waiting. If another thread beat us to it, we simply skip the creation and proceed. If the child is still NULL, we're guaranteed to be the only thread that can create it at this moment, ensuring no data loss or corruption.

Output Writing

The output writing process performs a simple depth-first traversal of the trie, building words character by character and writing them to the file when complete words are encountered:

```
void write_trie_to_file(TrieNode* node, char* prefix, int depth, FILE* fp)
{
  if (node->count > 0) {
    prefix[depth] = '\0';
    fprintf(fp, "%s: %d\n", prefix, node->count);
  }
```

```
  for (int i = 0; i < ALPHABET_SIZE; i++) {
    if (node->children[i] != `NULL`) {
      prefix[depth] = 'a' + i;
      write_trie_to_file(node->children[i], prefix, depth + 1, fp);
    }
  }
}
```

The function maintains a prefix string that represents the current path from the root. At each node with a non-zero count, it writes the complete word formed by the prefix along with its frequency. Because the children array is iterated from index 0 to 25 (corresponding to 'a' through 'z'), the output is automatically alphabetically sorted. This eliminates any need for a post-processing sort step, which would be computationally expensive for large dataset.

While recursion can sometimes lead to stack overflow issues with deeply nested structures, the depth of a trie is bounded by the length of the longest word, which is typically under 30 characters even for complex vocabularies. This makes the recursive approach both safe and elegant for this application.

1.1.4 • Memory Management

Node Allocation

Each trie node is allocated using `calloc` rather than `malloc`,

```
TrieNode* create_trie_node() {
    TrieNode* node = calloc(1, sizeof(TrieNode));
    if (!node) {
        fprintf(stderr, "Error: Memory allocation failed\n");
        exit(1);
    }
    pthread_mutex_init(&node->lock, `NULL`);
    return node;
}
```

Using `calloc` instead of `malloc` initializes all memory to zero, which automatically sets all 26 child pointers to `NULL` and the count to 0. This initialization is crucial because the insertion logic depends explicitly on `NULL` pointers to identify missing children. Using `malloc` would leave the memory uninitialized, containing whatever garbage values happened to be there previously, which would break the insertion algorithm completely.

The mutex is initialized immediately after allocation to ensure it's ready before any thread can possibly access the node.

Recursive Destruction:

The trie is destroyed using post-order traversal, where children are freed before their parent node:

```
void destroy_trie(TrieNode* node) {
    if (node == `NULL`) return;
```

```
    for (int i = 0; i < ALPHABET_SIZE; i++) {
        if (node->children[i] != `NULL`) {
            destroy_trie(node->children[i]);
        }
    }

    pthread_mutex_destroy(&node->lock);
    free(node);
}
```

This traversal order ensures that no node is freed while still referenced by a parent's children array. The mutex is destroyed before freeing the node to properly release system resources associated with the lock. If we freed nodes in pre-order (parent before children), we would lose the pointers to children, causing a memory leak as those child nodes would become unreachable but remain allocated.

The recursive destruction mirrors the structure of the trie itself, making it conceptually straightforward and less error-prone than an iterative approach would be. Each recursive call handles freeing one subtree, and the base case (node == `NULL`) prevents any attempts to dereference null pointers.

1.2 · Thread Data Structure

Each worker thread receives its own `ThreadData` structure containing all information needed to process its assigned portion of the workload independently:

```
typedef struct {
    TrieNode* root;          // Shared trie root
    char** words;            // All words from file
    int total_words;         // Total word count
    int thread_id;           // This thread's ID
    int num_threads;         // Total number of threads
} ThreadData;
```

The `root` pointer gives each thread access to the shared trie where all words are inserted. While the trie itself is shared among all threads, each thread maintains its own copy of this pointer in its ThreadData structure. This design makes it clear that the trie is shared while the ThreadData structures themselves are thread-local.

The `words` array contains all words read from the input file. This array is shared among all threads in a read-only fashion. No thread modifies the array itself; they only read words from it to insert into the trie. This read-only sharing eliminates the need for synchronization when accessing the words array, as reads are inherently thread-safe.

The `total_words` field specifies the length of the words array, allowing each thread to know when to stop processing without needing to search for a sentinel value or null terminator.

Each thread possesses a unique `thread_id` ranging from `0` to `num_threads - 1`. This identifier determines which words the thread will process according to the round-robin distribution algorithm, creating a natural and balanced work distribution scheme.

The `num_threads` field tells each thread the total number of threads in the pool, which is necessary for calculating the stride in the round-robin algorithm. Together with the thread_id, this enables each thread to independently determine its workload without any coordination.

```
Thread with `thread_id = i` processes words at indices: `i, i+n, i+2n,
i+3n, ...` where `n = num_threads`.

Example with 4 threads (n=4):
- Thread 0: indices 0, 4, 8, 12, 16 ...
- Thread 1: indices 1, 5, 9, 13, 17 ...
- Thread 2: indices 2, 6, 10, 14, 18 ...
- Thread 3: indices 3, 7, 11, 15, 19 ...
```

This distribution scheme ensures that work is spread evenly across threads with minimal overhead. Each thread can calculate its next work item independently without communication, and the load remains balanced even if word processing times vary, since long and short words are distributed fairly uniformly across the threads.

1.2.1 • Memory Management

The ThreadData structures are allocated on the heap in a single contiguous block:

```
ThreadData* thread_data = malloc(num_threads * sizeof(ThreadData));
```

This allocates an array of `ThreadData` structures on the heap using `malloc`. Each structure is then initialized with pointers to shared resources (root and words) and unique values (thread_id and num_threads). The heap allocation is necessary because these structures must outlive the function that creates them, remaining valid for the entire duration of thread execution.

Importantly, the ThreadData structures do not own the memory they point to. The root and words pointers reference memory allocated elsewhere and managed separately. This ownership model is deliberate: the main thread owns the trie and words array, while the `ThreadData` array is just a temporary structure for passing information to worker threads. This means when we free the thread_data array, we only free the array of structures themselves, not the trie or words array:

```
free(thread_data);  // Only frees the ThreadData array
// root and words are freed separately
```

This design prevents double-free errors and keeps ownership clear and unambiguous. The main thread is responsible for allocating the trie and words array before creating worker threads, and for freeing them after all threads have completed. This separation of concerns makes the memory management model straightforward to reason about and less prone to errors.

2 • Program Flow & File Parsing

2.1 • Execution Pipeline

The program executes through seven sequential phases, each building upon the previous to transform input text into frequency-counted output:

1. Parse command-line arguments (thread count, filename)
2. Read all words into memory
3. Create shared trie root
4. Allocate and initialize ThreadData structures
5. Launch worker threads
6. Wait for completion (`pthread_join`)
7. Write results and cleanup

```
int main(int argc, char* argv[]) {
    int num_threads = atoi(argv[1]);
    char* input_filename = argv[2];

    int total_words;
    char** words = read_words_from_file(input_filename, &total_words);

    TrieNode* root = create_trie_node();

    pthread_t* threads = malloc(num_threads * sizeof(pthread_t));
    ThreadData* thread_data = malloc(num_threads * sizeof(ThreadData));

    for (int i = 0; i < num_threads; i++) {
        pthread_create(&threads[i], `NULL`, count_words_in_thread,
                      &thread_data[i]);
    }

    for (int i = 0; i < num_threads; i++) {
        pthread_join(threads[i], `NULL`);
    }

    write_trie_to_file(root, prefix, 0, output);

    destroy_trie(root);
    free(words);
    free(threads);
    free(thread_data);
}
```

This structure follows a clear fork-join pattern: the main thread performs setup, spawns worker threads to do the computational work, waits for all workers to complete, then performs cleanup. This pattern is common in parallel programming because it provides clear synchronization points and makes reasoning about the program's behavior straightforward.

The separation between reading words and processing them is deliberate. By reading all words upfront, we avoid file I/O contention during the parallel processing phase. File I/O is inherently sequential and would create a bottleneck if threads tried to read from the file simultaneously. Loading everything into memory first, while requiring more memory, enables true parallelism during the processing phase.

2.2 · Reading Words

The file reading process handles the input file line by line, building a dynamically growing array of strings:

```c
char** read_words_from_file(const char* filename, int* word_count) {
    FILE* fp = fopen(filename, "r");

    int capacity = 1000;
    int count = 0;
    char** words = malloc(capacity * sizeof(char*));
    char buffer[MAX_WORD_LENGTH];

    while (fgets(buffer, sizeof(buffer), fp)) {
        buffer[strcspn(buffer, "\n\r")] = '\0';
        if (strlen(buffer) == 0) continue;

        if (count >= capacity) {
            capacity *= 2;
            words = realloc(words, capacity * sizeof(char*));
        }

        words[count] = strdup(buffer);
        count++;
    }

    *word_count = count;
    return words;
}
```

The function reads line by line using `fgets`, which provides buffered I/O for efficiency. It strips newlines with `strcspn`, which finds the first occurrence of any newline character and replaces it with a null terminator. Empty lines are skipped to avoid inserting empty strings into the trie. Each word is duplicated using `strdup`, which allocates new memory for the string rather than just storing a pointer to the buffer, which would be overwritten on the next iteration.

The function returns a dynamic array containing all words, along with the count through an output parameter. This two-way communication, returning the array directly and the count through a pointer, is a common C idiom for functions that return both a dynamically allocated structure and metadata about that structure.

2.3 · Memory Management

2.3.1 · Dynamic Array Growth

The array starts with a capacity of 1000 elements. When this capacity is reached, the array doubles in size using `realloc`. This doubling strategy provides amortized O(1) insertion time, meaning that over many insertions, the average cost per insertion remains constant despite occasional expensive resize operations.

For a dataset containing 120,000 words, this results in approximately 7-8 reallocations ($\log_2$ (120000/1000)). Each reallocation copies the existing array to a new, larger location, but because these happen exponentially less frequently as the array grows, the total copying overhead remains reasonable.

```
if (count >= capacity) {
    capacity *= 2;
    words = realloc(words, capacity * sizeof(char*));
}
```

The doubling strategy represents a balance between wasted space and reallocation frequency. Smaller growth factors (like 1.5x) waste less space but require more frequent reallocations. Larger growth factors (like 4x) require fewer reallocations but waste more space. Doubling has become the de facto standard because it provides good practical performance while being simple to implement and reason about.

2.3.2 · Cleanup

Memory must be freed in the reverse order of allocation, which ensures we never attempt to free memory through a pointer that has already been freed:

```
for (int i = 0; i < total_words; i++) {
    free(words[i]);  // Free individual strings first
}
free(words);  // Then free array
```

We must free the individual strings before freeing the array because each string was allocated separately by `strdup`. The words array itself only contains pointers to these strings, not the string data itself. If we freed the array first, we would lose access to the string pointers, making it impossible to free the string data and causing a memory leak.

This two-level deallocation pattern is common in C when working with arrays of dynamically allocated objects. The outer structure (the array) is freed last because it contains the pointers needed to free the inner structures (the strings). This ordering is critical for correct memory management.

3 · Multithreading Implementation

3.1 · Work Distribution

The round-robin distribution scheme assigns work to threads in a cyclic pattern: thread `i` processes indices `i, i+n, i+2n, i+3n, ...` where `n = num_threads`. This creates a natural load balancing mechanism without requiring any coordination between threads.

```
void* count_words_in_thread(void* arg) {
    ThreadData* data = (ThreadData*)arg;

    for (int i = data->thread_id; i < data->total_words; i += data-
>num_threads) {
        insert_word_into_trie(data->root, data->words[i]);
    }
```

```
    return `NULL`;
}
```

The loop structure is elegant in its simplicity. Each thread starts at its own unique offset (thread_id) and increments by the total number of threads (num_threads) on each iteration. This ensures that no two threads ever attempt to process the same word, eliminating the need for any synchronization when accessing the words array.

3.1.1 · Load Balancing

For `w` words and `n` threads, each thread processes either $\lfloor w/n \rfloor$ or $\lceil w/n \rceil$ words. The maximum difference between any two threads is exactly one word, representing nearly perfect load balancing. This property holds regardless of the total word count or number of threads, making the distribution scheme robust and predictable.

Consider a concrete example with 13 words and 4 threads:

```
Thread 0: 0, 4, 8, 12     (4 words)
Thread 1: 1, 5, 9         (3 words)
Thread 2: 2, 6, 10        (3 words)
Thread 3: 3, 7, 11        (3 words)
```

The first thread gets one extra word because 13 doesn't divide evenly by 4, but this imbalance is minimal. In practice, with datasets containing tens of thousands of words, this difference becomes negligible. The round-robin pattern also ensures that if some words take longer to process than others (perhaps due to varying lengths or hash collisions), these variations are distributed evenly across threads rather than concentrating in one thread.

3.2 · Thread Synchronization

3.2.1 · Race Conditions

Without proper synchronization, two primary race conditions could corrupt the trie structure and produce incorrect results:

Node Creation Race: Two threads detect that a child node is missing and both attempt to create it. Without synchronization, both threads would allocate and initialize a new node, and one would overwrite the other's pointer in the children array. This overwrites the first node created, causing a memory leak (the overwritten node is never freed) and data loss (any words already inserted into that subtree are lost).

Count Increment Race: The operation `count++` appears atomic in the source code but actually consists of three separate machine instructions: read the current value, increment it, and write the new value back. If two threads execute this sequence simultaneously, they might both read the same initial value, both increment it to the same new value, and both write that same new value back. The result is that only one increment is recorded instead of two, producing an incorrect word count.

These race conditions would occur unpredictably, making the program's behavior non-deterministic. The same input might produce different outputs on different runs, depending on the exact timing of thread execution. This non-determinism makes debugging extremely difficult, as the program might appear to work correctly most of the time but occasionally produce wrong results.

3.2.2 · Solution: Fine-Grained Locking

The solution employs fine-grained locking, where each node has its own mutex rather than using a single global lock for the entire trie:

```
typedef struct TrieNode {
    struct TrieNode* children[ALPHABET_SIZE];
    int count;
    pthread_mutex_t lock;  // One lock per node
} TrieNode;
```

This design enables multiple threads to work on different paths through the trie simultaneously without interfering with each other. Locks are acquired only in two specific cases:

1. Creating a child node (lock the parent)
2. Incrementing the count (lock the final node)

During normal traversal, when a thread is simply following existing pointers from parent to child, no locking is required. This works because child pointers never change after creation. Once a child node is created and its pointer is stored in the parent's children array, that pointer remains valid for the lifetime of the trie. Multiple threads can safely read this pointer simultaneously because they're not modifying it.

This property enables significant parallelism. Imagine two threads inserting completely different words, like "apple" and "zebra". These threads will follow completely different paths through the trie, touching different nodes. With fine-grained locking, they can proceed completely independently, never waiting for each other. Even threads inserting similar words like "apple" and "apply" only contend for locks when they reach the divergence point in their paths.

The alternative, a single global lock for the entire trie, would serialize all insertions. Threads would have to wait for each other even when working on completely unrelated parts of the trie, effectively eliminating any benefit from parallelization. Fine-grained locking preserves the potential for parallelism while still preventing race conditions.

3.2.3 · Memory Barriers

Mutex operations provide memory barriers that ensure proper synchronization of memory accesses across threads. When thread A releases a lock and thread B subsequently acquires the same lock, B is guaranteed to see all memory modifications made by A before the release. This guarantee is crucial for correctness in multithreaded programs.

For the trie, this means:

- When a thread creates a child node and releases the lock, any other thread that later acquires that lock will see the newly created child
- When a thread increments a count and releases the lock, any other thread that later acquires that lock will see the updated count

Without these guarantees, provided automatically by the mutex implementation, the program could appear to work on some hardware architectures but fail mysteriously on others. Modern CPUs perform various optimizations like instruction reordering and caching that can cause memory writes by one thread to become visible to other threads in unexpected orders. Memory barriers prevent these optimizations from violating the program's intended semantics, ensuring that the program behaves correctly regardless of the underlying hardware architecture.

4 · Compilation Instructions

To Run The Code With A Default Of 4 Threads And Provided Input Dataset:

```
[wizard@archlinux ~/Projects/School/HPC/Assessment/t1/code] -> ls
file_utils.c  file_utils.h  main.c  Makefile  trie.c  trie.h
[wizard@archlinux ~/Projects/School/HPC/Assessment/t1/code] -> make run
```

To Run The Code With A Custom Number Of Threads And Custom Dataset:

```
// compile:
[wizard@archlinux ~/Projects/School/HPC/Assessment/t1/code] -> gcc *.c -
Wall -Wextra -O2

// run
[wizard@archlinux ~/Projects/School/HPC/Assessment/t1/code] -> ./a.out -h
Usage: ./a.out <num_threads> <input_file>
Example: ./a.out 4 input.txt

[wizard@archlinux ~/Projects/School/HPC/Assessment/t1/code] ->
```

5 · Performance Analysis

5.1 · Profiling Results

The program was tested using a dataset containing 119,999 words from `WordOccurrenceDataset.txt`, running on an Arch Linux system with x86-64 architecture. The profiling results provide insight into where the program spends its time and how effectively it utilizes multiple cores.

5.1.1 · Execution Time

Running the program with 4 threads yields the following timing results:

```
$ time ./word_counter 4 WordOccurrenceDataset.txt
0.01s user 0.00s system 128% cpu 0.012 total
```

The execution completes in just 0.012 seconds of real time, with 0.01 seconds spent in user mode and essentially zero time in kernel mode. The CPU utilization of 128% indicates that the program is effectively using more than one core, which is exactly what we expect from a multithreaded application. This value represents the sum of CPU time across all threads divided by the real time, so 128% means the program is utilizing the equivalent of about 1.3 cores actively working.

With 4 threads, the theoretical maximum CPU utilization would be 400% if all four threads were continuously busy. The lower actual utilization of 128% occurs because the program includes sequential phases, particularly file I/O and the final output writing, where only one thread is active. Additionally, synchronization overhead from acquiring and releasing locks, and time spent in memory allocation, contributes to threads occasionally waiting rather than computing.

5.1.2 · CPU Profile (`perf report`)

Top functions by CPU time:

```
  16.32%  a.out    a.out                  [.] insert_word_into_trie
  13.08%  a.out    libc.so.6              [.] 0x00000000000a4f15
  11.37%  a.out    libc.so.6              [.] 0x000000000009b9fc
   7.03%  a.out    a.out                  [.] read_words_from_file
   5.82%  a.out    libc.so.6              [.] malloc
   5.27%  a.out    libc.so.6              [.] _IO_fgets
   4.58%  a.out    libc.so.6              [.] pthread_mutex_lock
   4.13%  a.out    libc.so.6              [.] 0x000000000018d60c
   3.74%  a.out    libc.so.6              [.] _IO_getline_info
   3.29%  a.out    libc.so.6              [.] 0x000000000017d4ef
   2.61%  a.out    libc.so.6              [.] 0x000000000018d6d8
   2.47%  a.out    libc.so.6              [.] cfree
   2.20%  a.out    libc.so.6              [.] 0x00000000000a6431
   2.18%  a.out    libc.so.6              [.] 0x000000000017d4c4
   2.17%  a.out    a.out                  [.] main
   2.09%  a.out    [unknown]              [k] 0xffffffff89401280
   2.00%  a.out    libc.so.6              [.] 0x00000000000a686c
   1.91%  a.out    libc.so.6              [.] 0x0000000000180da8
   1.21%  a.out    libc.so.6              [.] 0x0000000000186de8
   1.09%  a.out    libc.so.6              [.] 0x0000000000180da2
   1.00%  a.out    libc.so.6              [.] 0x000000000009b9f0
   0.93%  a.out    libc.so.6              [.] 0x00000000000a643d
   0.83%  a.out    a.out                  [.] count_words_in_thread
   0.83%  a.out    libc.so.6              [.] __strdup
   0.81%  a.out    libc.so.6              [.] 0x0000000000186e37
   0.69%  a.out    libc.so.6              [.] 0x000000000018d68f
   0.20%  a.out    ld-linux-x86-64.so.2   [.] 0x0000000000014dbd
   0.04%  a.out    libc.so.6              [.] 0x00000000000a418d
   0.03%  a.out    libc.so.6              [.] 0x00000000000a6441
   0.03%  a.out    libc.so.6              [.] 0x00000000000a6e82
   0.03%  a.out    ld-linux-x86-64.so.2   [.] 0x0000000000014787
   0.02%  a.out    libc.so.6              [.] 0x00000000000a6aed
   0.00%  a.out    libc.so.6              [.] 0x0000000000065404
   0.00%  a.out    ld-linux-x86-64.so.2   [.] 0x000000000001f70a
   0.00%  a.out    libc.so.6              [.] 0x0000000000186dd3
   0.00%  a.out    libc.so.6              [.] 0x000000000005b56d
   0.00%  a.out    libc.so.6              [.] 0x0000000000186d84
   0.00%  a.out    libc.so.6              [.] 0x00000000001803b0
   0.00%  a.out    libc.so.6              [.] __ctype_init
   0.00%  a.out    libc.so.6              [.] 0x0000000000180380
   0.00%  a.out    libc.so.6              [.] 0x000000000009676c
   0.00%  a.out    libc.so.6              [.] 0x00000000000966e0
   0.00%  a.out    libc.so.6              [.] 0x000000000011aa00
   0.00%  a.out    ld-linux-x86-64.so.2   [.] 0x000000000001eb43
```

```
  0.00%  a.out    ld-linux-x86-64.so.2  [.] 0x000000000001f6ab
  0.00%  a.out    libc.so.6             [.] 0x000000000011aa07
```

- **Hotspot:** `insert_word_into_trie` at 16.32%; this is expected, as this is the core algorithm.
- **Memory allocation:** Combined malloc/free operations 19% overhead. Significant but unavoidable for dynamic trie construction.
- **Lock contention:** `pthread_mutex_lock` at 4.58% - low overhead indicates minimal contention. Fine-grained locking works efficiently.

Bottleneck: Memory operations (allocation + libc internals) consume 30% total.

5.1.3 · Conclusion

Program achieves effective parallelization with minimal synchronization overhead. Primary cost is memory management, not thread contention. Fine-grained locking strategy validated by low lock overhead (4.58%).

6 · Conclusion

Implementation achieves:

- **Automatic sorting:** Trie structure outputs alphabetically without separate sort phase.
- **Balanced distribution:** Round-robin ensures ±1 word difference between threads.
- **Thread safety:** Per-node locks allow concurrent access to different trie regions.
- **Memory safety:** Proper allocation/deallocation order prevents leaks.