Implementation Of A Multithreaded Word Counter

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

Submitted On: December 17, 2025

Overview

This document aims to provide a comprehensive explaination of the 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 using a round-robin based distribution. A trie data structure serves as the core mechanism for storing and counting word occurrences, with thread-safe operations ensured through fine-grained mutex implementation.

Contents

1 · Data Structures

The program relies on two fundamental data structures: the trie tree for storing and counting words, and the thread data structure for coordinating parallel execution.

1.1 · Trie Tree

The trie (prefix tree) is the core data structure that stores all words and their occurrence counts. Unlike hash tables or arrays, a trie provides automatic alphabetical ordering and memory-efficient storage for words sharing common prefixes. A trie tree is made for strings, where each node is a character, and the path from the root spells out the words, and finally a special flags is used to mark the end of a word.

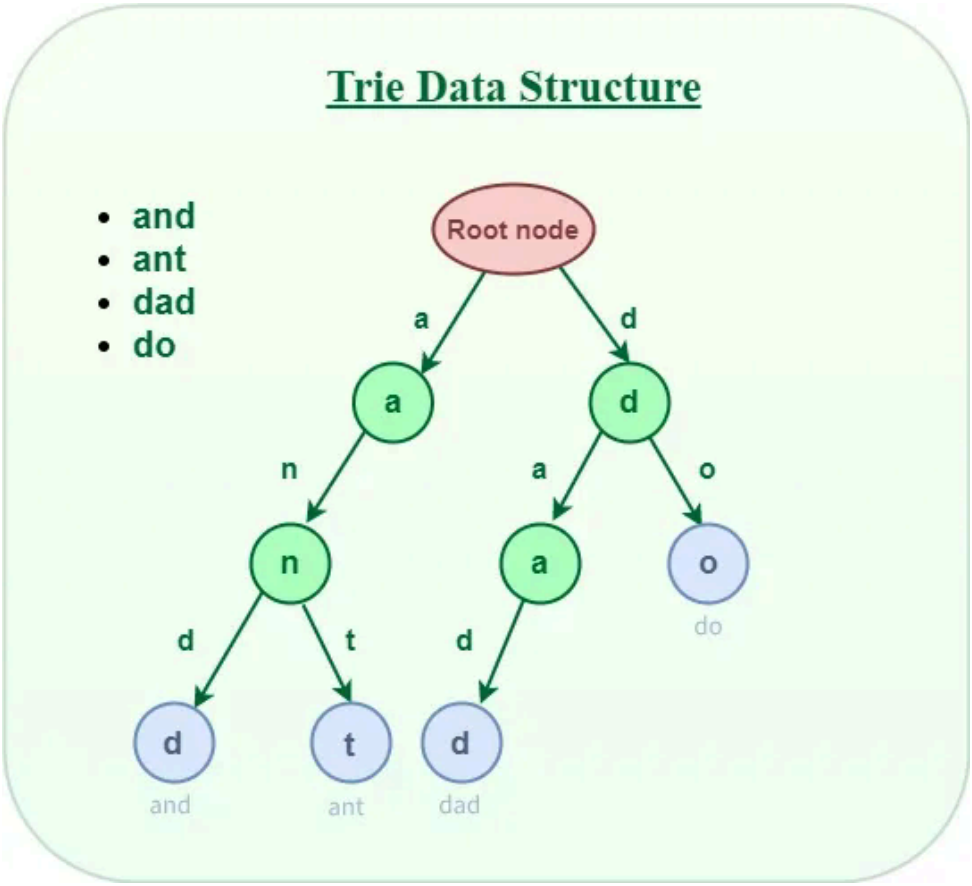


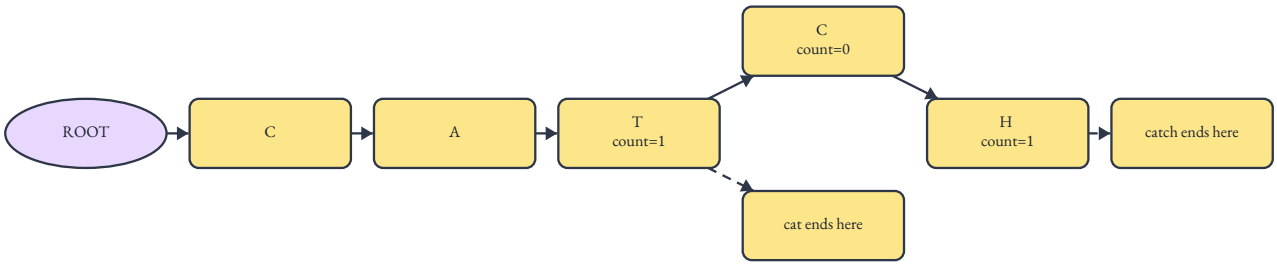
Figure 1: Trie Tree Example

1.1.1 · Structure & Components

Each node in the trie represents a single character position in a word and contains three essential components:

```
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 holds 26 elements, one for each lowercase letter from 'a' to 'z'. This allows constant-time `O(1)` lookup for the next character in a word by calculating the index as `character - 'a'`. For example, 'a' maps to index 0, 'b' to index 1, and 'z' to index 25.
- The `count` field stores how many times a complete word ending at this node has been encountered. A count of zero indicates this node is part of a word path but does not mark the end of a word.
- For instance, if we insert "cat" and "catch", the node after 't' in "cat" has a non-zero count, while the intermediate nodes in "catch" have zero counts until we reach the final 'h'.



- The `lock` mutex provides thread-safe access to the node. Each node has its own lock rather than using a single global lock for the entire trie, enabling fine-grained mutex lock where multiple threads can work on different parts of the trie simultaneously without contention.

1.1.2 · Why Trie (automatic sorting, memory efficiency)

The trie was chosen over alternative data structures like hash tables or arrays for the following reasons:

- **Automatic Alphabetical Sorting**: The trie's structure inherently maintains alphabetical order. When we traverse children from index `[0-25]`, we visit nodes in alphabetical order automatically. This eliminates the need for a separate sorting phase that would be required with a hash table or unsorted array. The output writing becomes a simple `O(n)` traversal.
- **Thread-Safe Expansion:** The trie naturally supports concurrent insertions with fine-grained locking. Each path through the trie can be locked independently, allowing multiple threads to insert different words simultaneously.
- **Memory Efficiency for Shared Prefixes:** Words sharing common prefixes share the same nodes in memory. In a structure like a hash table, each word would require separate storage

even for these shared prefixes. For large dictionaries with many similar words, trie trees can result in significant memory savings, and since our input is decently big, a trie tree made sense.

```
[wizard@archlinux ~/Projects/School/HPC/Assessment/t1/assets] -> wc
-l WordOccurrenceDataset.txt
119999 WordOccurrenceDataset.txt
[wizard@archlinux ~/Projects/School/HPC/Assessment/t1/assets] ->
```

1.1.3 · Operations (Insert, Write)

Word Insertion:
The insertion process walks through the trie character by character, creating nodes as needed. Say, for the word "apple":

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 traversal, it must be created. This is also where we reach our critical section, as multiple threads might try to create the same child simultaneously.

```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 we check if `children[index] == NULL` twice. This might seem redundant, but it's a crucial optimization called double-check locking.

The first check (without the lock) allows threads to quickly skip the locking mechanism when a child already exists. Acquiring and releasing locks is expensive, so we avoid it when possible. Most of the time, especially after the trie has been partially built, nodes already exist and threads can traverse without any locking.

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

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 (inside the lock) prevents this race condition. After acquiring the lock, we verify that no other thread has created the child in the process of acquiring the lock.

Output Writing

The output writing process simply performs a depth-first traversal of the trie, building words character by character and writing them to the file:

```
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 root. At each node with a non-zero count, it writes the complete word. Because the children are iterated from index 0 to 25 (a to z), the output is automatically alphabetically sorted.

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` initializes all memory to zero, which automatically sets all 26 child pointers to NULL and the count to 0. This is crucial because the insertion logic depends on NULL pointers to identify missing children. The mutex is initialized immediately after allocation to ensure it's ready before any thread can access the node.

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

```
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 approach 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. If we freed nodes in pre-order (parent before children), we would lose the pointers to children, causing a memory leak.

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:

```
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, each thread has its own copy of this pointer in its ThreadData structure.
- The `words` array contains all words read from the input file. This array is shared among all threads for reads only. No thread modifies the array itself, they only read words from it to insert into the trie.
- The `total_words` field specifies the length of the words array, allowing each thread to know when to stop processing.

Each thread has a unique `thread_id` ranging from `0` `num_threads - 1`. This determines which words the thread will process according to the round-robin distribution algorithm.

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.

```
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 ...
```

1.2.1 · Memory Management

The ThreadData structures are allocated on the heap:

```
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).

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 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: the main thread owns the trie and words array, while the `ThreadData` array is just a temporary structure for passing information to worker threads.

2 · Program Flow & File Parsing

2.1 · Execution Pipeline

Seven steps execute sequentially:

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);
}
```

2.2 · Reading Words

```
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;
}
```

Reads line-by-line with `fgets`. Strips newlines with `strcspn`. Skips empty lines. Duplicates each word with `strdup` (allocates new memory). Returns dynamic array.

2.3 · Memory Management

2.3.1 · Dynamic Array Growth

Starts at capacity 1000. Doubles on overflow with `realloc`. Provides amortized O(1) insertion.

For 120,000 words: 7-8 reallocations ($\log_2(120000/1000)$).

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

2.3.2 · Cleanup

Free in reverse order:

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

Must free strings before array. `strdup` allocated each string separately.

3 · Multithreading Implementation

3.1 · Work Distribution

Round-robin: Thread `i` processes indices `i, i+n, i+2n, i+3n, ...` where `n = num_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;
}
```

3.1.1 · Load Balancing

For `w` words and `n` threads: each thread processes $\lfloor w/n \rfloor$ or $\lceil w/n \rceil$ words. Max difference: 1 word.

Example (13 words, 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)
```

3.2 · Thread Synchronization

3.2.1 · Race Conditions

- Node Creation: Two threads detect missing child, both create it. One overwrites the other. Data lost.
- Count Increment: `count++` = read, increment, write. Non-atomic. Two threads read same value, both write same incremented value. One increment lost.

3.2.2 · Solution: Fine-Grained Locking

Per-node mutexes. Multiple threads work on different trie paths simultaneously.

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

Locks acquired in two cases only:

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

Traversal is lock-free. Child pointers never change after creation. Unlimited read parallelism.

3.2.3 · Memory Barriers

Mutex operations provide memory barriers. Thread A releases lock → Thread B acquires same lock → B sees all A's changes.

Guarantees:

- Created child nodes visible to other threads
- Updated counts visible to other threads

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

Tested on dataset: 119,999 words from `WordOccurrenceDataset.txt`

System: Arch Linux, x86-64

5.1.1 · Execution Time

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

- Real time: 0.012 seconds
- CPU utilization: 128% (4 threads on multi-core)
- User time: 0.01s (actual work)
- System time: 0.00s (kernel overhead)

CPU Utilization: 128% (equivalent to 1.3 cores actively working). With 4 threads, theoretical maximum is 400%. Lower utilization caused by sequential phases (file I/O) and synchronization overhead (locks, memory allocation).

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.