Word Frequency Counter Using Multithreading

Peera Tienthong

Master of Computer Science, University of the Cumberlands

MSCS-630: Advanced Operating Systems

Dr. Gary Perry

May 31, 2025

Program Design and Structure

This program is intended to count word frequency in a text. The goal is to understand how operating systems manage concurrency and multithreading by implementing a simple program enabling concurrency. Users can provide two parameters to the program: a text file name containing words, and the number of threads used in the program. The main thread is responsible for opening the text file and reading it. Then the next task is to calculate the size of the text segments used in each thread. The main thread also divides the text into several chunks according to the number of segments. The next step is to transfer tasks to each thread to count word frequency and start threads. Finally, the main program combines word counts collected from each thread and prints them out to the console.

Logic Behind Multithreading

The logical explanation of the program is that the main thread is only responsible for text file reading and thread creation. The backbone of the program is how to count words in each thread. Here, I designed the algorithm inside a function to count words. To ensure that each thread's work does not overlap with each other, I need to divide the distinct text segments into each thread. Each thread works independently until it is finished. To combine word counts, each thread assigns the count back to the main thread at the end of the work. Finally, the main thread waits for each thread to finish and prints the total number of counts.

Implementation and Challenges

Text file reading

```
def main():
    file_name = 'sample.txt'
    try:
        with open(file_name, 'r') as file:
            text = file.read()
            print("Contents of text file:\n")
            print(text)
            print('\n')
    except FileNotFoundError:
        print(f"Error: The file {file_name} was not found.")
        return
```

For text file reading, I used the try-except block for handling the exception as a safety reason.

Segment Size Calculations

```
    # Count words in the text
    num_threads = 4
    words = re.findall(r'\b\w+\b', text)
    total_words = len(words)
    chunk_size = (total_words + num_threads - 1) // num_threads
```

The number of threads is assigned by a user. The challenge is how to retrieve words from a text string. Thanks to the regular expressions, the library handles word detection and adds it to a list of words. The \b anchors ensure that only whole words are matched, and \w+ matches sequences of word characters.

Text Segment Generation

```
threads = []
for i in range(num_threads):
    start = i * chunk_size
    end = min(start + chunk_size, total_words)
    text_segment = ' '.join(words[start:end])
    thread = threading.Thread(target=count_words, args=(text_segment,))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join()
```

The segment is created by joining words between “start” and “end” index slicing. Then a text segment is passed to a function “count_words”, which is attached to each thread. The main thread joins all threads and waits until all threads finish.

Count Words

```
# Global counter and lock
word_counter = Counter()
lock = threading.Lock()

def count_words(text_segment):
    local_counter = Counter()
    words = re.findall(r'\b\w+\b', text_segment.lower())
    local_counter.update(words)
    print(f"Intermediate count for segment: {local_counter}")
    with lock:
        word_counter.update(local_counter)
```

For the algorithm, I used a data structure “Counter”, which collects a word as a key and its frequency as a value. If a new word is encountered, the key is appended to the counter; otherwise, the frequency value increments. To keep track of the total counts, I used a global counter and enabled a thread-safe mechanism – locking - for accessing shared data among threads.

Sample Output of the Program

4 Threads

```
Multithreading in Python is fun and powerful.
Python allows multiple threads to run.
Threads can run concurrently and do many tasks.
Fun with threads and Python is amazing.


Intermediate count for segment: Counter({'multithreading': 1, 'in': 1, 'python': 1, 'is': 1, 'fun': 1, 'and': 1, 'powerful': 1})
Intermediate count for segment: Counter({'threads': 2, 'python': 1, 'allows': 1, 'multiple': 1, 'to': 1, 'run': 1})
Intermediate count for segment: Counter({'can': 1, 'run': 1, 'concurrently': 1, 'and': 1, 'do': 1, 'many': 1, 'tasks': 1})
Intermediate count for segment: Counter({'fun': 1, 'with': 1, 'threads': 1, 'and': 1, 'python': 1, 'is': 1, 'amazing': 1})
Combined Word counts:
multithreading: 1
in: 1
python: 3
is: 2
fun: 2
and: 3
powerful: 1
allows: 1
multiple: 1
threads: 3
to: 1
run: 2
can: 1
concurrently: 1
do: 1
many: 1
tasks: 1
with: 1
amazing: 1
```

8 Threads

```
Multithreading in Python is fun and powerful.
Python allows multiple threads to run.
Threads can run concurrently and do many tasks.
Fun with threads and Python is amazing.


Intermediate count for segment: Counter({'multithreading': 1, 'in': 1, 'python': 1, 'is': 1})
Intermediate count for segment: Counter({'fun': 1, 'and': 1, 'powerful': 1, 'python': 1})
Intermediate count for segment: Counter({'allows': 1, 'multiple': 1, 'threads': 1, 'to': 1})
Intermediate count for segment: Counter({'run': 2, 'threads': 1, 'can': 1})
Intermediate count for segment: Counter({'concurrently': 1, 'and': 1, 'do': 1, 'many': 1})
Intermediate count for segment: Counter({'tasks': 1, 'fun': 1, 'with': 1, 'threads': 1})
Intermediate count for segment: Counter({'and': 1, 'python': 1, 'is': 1, 'amazing': 1})
Intermediate count for segment: Counter()
Combined Word counts:
multithreading: 1
in: 1
python: 3
is: 2
fun: 2
and: 3
powerful: 1
allows: 1
multiple: 1
threads: 3
to: 1
run: 2
can: 1
concurrently: 1
do: 1
many: 1
tasks: 1
with: 1
amazing: 1
```