ECEC 353 Assignment 2

William Fligor, Avik Bag

July 2016

1 Design

For this assignment, the aim is to implement a search mechanism using the concept of multi-threading. The idea is that the program will be given a few parameters, like number of threads, start directory, type of implementation and the string to be searched. There will be two forms of the implementation.

For the first part, the given directory will be traversed and a queue of all the files and folders will be in it. This queue will then be split up by the number of threads that are available. Each thread will be responsible for it's set of files/folders. If there is a folder, it will recursively go into them and list out all the available files, until all the files are listed in the queue. This implementation was definitely faster than a single thread implementation. But the issue that arises is that the threads are not getting an even distribution of the load. In other words, some threads might need to do more work than others. This is because certain threads might need to deal with folders recursively, causing the size of the queue to increase from it's assigned size. This implementation is called the static load balancing. The drawback of not being able to balance the load will be addressed in the next implementation of load balancing.

In this implementation, called the dynamic load balancing, every thread gets access to the global queue which contains all the files and folders. Every thread will make a blocking call to grab a value from the queue. There will be a mutex ensuring that only one thread will be making any modifications from the queue at any given time. This ensures that every thread will have some task to carry out, either recursively populating the queue from a folder or searching a file for the given string. One issue that came up was that when the queue was empty, a thread would exit, while another thread could have been in the processes of searching through folders and repopulating the queue. This would cause a loss in performance because now there will be lesser number of threads available. The way to tackle the performance issue was by keeping the threads on waiting even if the queue is empty, and also keep a track of all the waiting threads. If all the threads are on wait, this means that there is no more processing left and there was no way the queue was going to get any more new values, thus it meant that the process is over and all the threads can be stopped and joined in the main method.

The implementations make use of a struct that makes it easier to pass multiple arguments to the function pointer when creating a thread. It is as shown below

```
struct ThreadArgument {
    int lock; // Bool to lock or not around queue
    char *search_ptr;
    queue_t *queue;
    int thread_num;
    int result;
};
```

Algorithm 1 Static Load Balancing

```
num ← NUM_THREADS // No. of threads
queue // initialize queue
d ← opendir(filePath) // Open directory and list file/folders
count = 0 // keep a track of the elements
if d == valid then
    while dir = readdir(d) do
        queue_num ← countmodNUM_THREADS
        if dir− > d_name == "."||dir− > d_name == ".." then
            continue
        end if
        Insertfoundfile/directoryintoqueue
        count + +
    end while
    closedir(d)
end if
threadArgs[NUM_THREADS] // Used to initialize thread arguments
pthread[NUM_THREADS] // Initializing threads
occurrences = 0 // Keeps a track of the times the search string was encountered
for i = 0 → NUM_THREADS − 1 do
    threadArgs[i].lock ← 0
    threadArgs[i].search_ptr ← searchString
    threadArgs[i].queue ← queues
    threadArgs[i].thread_num ← i
    pthread_create(pthread[i], NULL, processQueueT, &threadArgs[i])
end for
for i = 0 → NUM_THREADS − 1 do
    pthread_join(pthread[i], NULL)
    occurrences ← occurrences + threadArgs[i].result
end for
return occurrences
```

Algorithm 2 Dynamic Load Balancing

```
num ← NUM_THREADS // No. of threads
queue // initialize queue
d ← opendir(filePath) // Open directory and list file/folders
if d == valid then
    while dir = readdir(d) do
        if dir− > d_name == "."||dir− > d_name == ".." then
            continue
        end if
        element ← create_element(dir− > d_name)
        insertElement(queue, element)
    end while
    closedir(d)
end if
threadArgs[NUM_THREADS] // Used to initialize thread arguments
pthread[NUM_THREADS] // Initializing threads
occurrences = 0 // Keeps a track of the times the search string was encountered
for i = 0 → NUM_THREADS − 1 do
    threadArgs[i].lock ← 1
    threadArgs[i].search_ptr ← searchString
    threadArgs[i].queue ← queue
    threadArgs[i].thread_num ← i
    pthread_create(pthread[i], NULL, processQueueT, &threadArgs[i])
end for
for i = 0 → NUM_THREADS − 1 do
    pthread_join(pthread[i], NULL)
    occurrences ← occurrences + threadArgs[i].result
end for
return occurrences
```

Algorithm 3 processQueueT

```
1: num ← NUM_THREADS // No. of threads
2: thread_status ← 0
3: count ← 0
4: while true do
5:   if threadArgs.lock == 1 then
6:     pthread_mutex_lock(&lock)
7:   end if
8:   if queueis empty then
9:     if threadArgs.lock == 1 then
10:      if !thread_status then
11:        wait_count + +
12:      end if
13:      thread_status ← 1
14:      if wait_count == num then
15:        pthread_mutex_unlock(&lock)
16:        break // Break when all threads are done
17:      else
18:        pthread_mutex_unlock(&lock)
19:        continue
20:      end if
21:    end if
22:  end if
23:  if !thread_status then
24:    wait_count − −
25:    thread_status ← 0
26:  end if
27:  if threadArgs.lock == 1 then
28:    pthread_mutex_lock(&lock)
29:  end if
30:  element ← getElement(queue)
31:  if threadArgs.lock == 1 then
32:    pthread_mutex_unlock(&lock)
33:  end if
34:  if element == file then
35:    Read through file, increment count for every hit
36:  end if
37:  if element == folder then
38:    while dir = readdir(element) do
39:      if dir− > d_name == "."||dir− > d_name == ".." then
40:        continue
41:      end if
42:      element ← create_element(dir− > d_name)
43:      if threadArgs.lock == 1 then
44:        pthread_mutex_lock(&lock)
45:      end if
46:      insertElement(queue, element)
47:      if threadArgs.lock == 1 then
48:        pthread_mutex_unlock(&lock)
49:      end if
50:    end while
51:    closedir(d)
52:  end if
53: end while
54: return count
```

2 Speedup Statistics

Threads	Real Time (s)	User Time (s)
1	0.643	0.269
2	1.076	0.300
4	0.784	0.323
8	1.069	0.335
16	0.639	0.241

Table 1: Run Times for Static implementation

Threads	Real Time (s)	User Time (s)
1	0.669	0.264
2	0.396	0.285
4	0.366	0.372
8	0.335	0.365
16	0.258	0.386

Table 2: Run Times for Dynamic implementation