CSC415 – Homework 4 – Word Counter

Aleksandr Kibis

10/16/2014

This assignment was our first dive into multithreaded applications. Having some experience with POSIX, I decided to start with pthread implementation. Out of all the projects we have done so far, this one was on the easier side, possibly due to my past experience. First I created a single threaded word counter, making sure to split up the functions of the program: reading a file into the buffer, creating multiple threads and cutting data into chunks, counting words and updating global counter variable. The biggest challenge that I faced was passing a struct to each individual thread. In the past, I was under the impression that only pointers to primitive variables or objects were able to be passed to threads. Thankfully this isn't the case which really opens up the possibilities. After having implemented the main functions, all that was left was time measurement. C has a great function called clock() that is able to grab current program runtime so my implementation of it used two time stamps, then the difference was taken to calculate the run time for all the calculations (minus variable instantiation). Unfortunately, Windows did not play very nicely with this method so I had to resort to Windows PowerShell in order to take my timings. I believe this is due to differences in how system time is calculated on Windows machines vs Linux/Unix. Both versions of the program were run for 1, 2, 4, and 8 threads. I noticed that the fastest runs were for 4 threads on the Windows machine, the POSIX version seemed to prefer 1 and 8 threads. To really get a better understanding on thread performance, a bigger file would have to be used, preferably something that is multiple megabytes long. That is where multithreading will really shine. Given that the buffer is only 80Kb, the thread creation overhead may be too much to really notice a difference.

Number of Threads	POSIX	Windows
1	2ms	61ms
2	2ms	56ms
4	2ms	54ms
8	2ms	54ms

POSIX:

Code

```
/*
 * File:   wordCounter.c
 * Author: Aleksandr Kibis
 *
 * Multi-threaded word count application. Used clock() instead of time()
 * because it provides more accurate data.
 *
 * Compile: gcc -pthread -o wordCount wordCounter.c
 *
 * Run: ./wordCount <text file>
 *
 * Created on October 3, 2014, 6:53 PM
 */

#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <unistd.h>
#include <time.h>
#include <string.h>

#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1

#define BUFFER_SIZE 1024 * 80
#define THREAD_COUNT 8

int getFile(char* file);
void *counter(void *attributes);

// specifies data range to work on for each thread

struct threadAttr {
    int begin;
    int end;
    int id;
};

char* text;
int* countArray;
int wordCount;

/*
 *
 */
int main(int argc, char** argv) {

    if (argv[1] == NULL) {
        printf("Usage: ./wordCount <text file>\n");
        exit(EXIT_FAILURE);
```

```
}


//clock_t t1, t2;
// get start time
//t1 = clock();

countArray = malloc(THREAD_COUNT * sizeof (int));
text = malloc(BUFFER_SIZE * sizeof (char));

// write input file to global memory
int fileSize;
fileSize = getFile(argv[1]);
//printf("File size: %d\n", fileSize);

// exit if write to buffer is unsuccessful
if (fileSize < 0) {
    free(text);
    return (EXIT_FAILURE);
}
    // count number of words
else {

    // split data into evenly size chunks
    int chunkSize = fileSize / THREAD_COUNT;
    //printf("Split check: %c\n", text[chunkSize]);

    // create threads and run word count on each chunk
    pthread_t threadArray[THREAD_COUNT];
    pthread_attr_t pta;
    int rc = 0;
    int retval;
    struct threadAttr *input;

    rc = pthread_attr_init(&pta);
    //printf("Result: %d\n", rc);

    // thread initialization
    int i;
    int offset = 0;
    for (i = 0; i < THREAD_COUNT; i++) {
        // each thread must have it's own struct so that data won't be overwritten
        input = malloc(sizeof (struct threadAttr));
        (*input).begin = i * chunkSize;
        if (i == THREAD_COUNT - 1)
            (*input).end = fileSize;
        else
            (*input).end = i * chunkSize + chunkSize;
        (*input).id = i;
        retval = pthread_create(&threadArray[i], &pta, counter, (void*) input);
        if (retval > 0) {
            printf("Error in creating thread.\n");
            exit(EXIT_FAILURE);
        }
    }

    // wait for all threads to finish
    for (i = 0; i < THREAD_COUNT; i++) {
        pthread_join(threadArray[i], NULL);
        if (retval > 0) {
            printf("Error in joining thread.\n");
            exit(EXIT_FAILURE);
        }
    }

    // ITERATIVE IMPLEMENTATION

    //        int wordCount = 0;
    //
```

```
        //        for (i = 0; i < fileSize; i++){
        //            // space char is delimiter
        //            if((int)text[i] == 32)
        //                wordCount++;
        //
        //            // first word exception
        //            if((int)text[i] > 64 && (int)text[i] < 91 && (int)text[i-1] != 32)
        //                wordCount++;
        //        }
        //        countArray[0] = wordCount;
        //        printf("Word Count: %d\n", countArray[0]);

        printf("Number of Words: %d\n", wordCount);
        free(input);
        free(text);
        // get end time
        //t2 = clock();
        // calculate total time elapsed
        //printf("Elapsed Time: %.6fs\n", (t2 - t1) / (double) CLOCKS_PER_SEC);
        return (EXIT_SUCCESS);

    }


}

int getFile(char* file) {
    int count = 0;

    //printf("Input file name: %s\n", file);

    FILE *fd;
    int ch;
    fd = fopen("text", "r");
    while (1) {
        ch = fgetc(fd);
        if (ch == EOF)
            break;
        ++count;
    }
    rewind(fd);
    //printf("Number of chars in file: %d\n", count);

    size_t readCount;
    readCount = fread(text, 1, count, fd);
    fclose(fd);

    if (count == (int) readCount) {
        //        printf("Read file into buffer: SUCCESS\n");
        //        printf("************************************************\n");
        //printf("%s\n", text);
        return count;
    } else {
        printf("Buffer read error. Exiting.\n");
        return -1;
    }

    //printf("Chars read: %d\n", (int)readCount);



}

void *counter(void *attributes) {
    int localCount = 0;
    struct threadAttr *newOne = (struct threadAttr*) attributes;

    // assign struct values to new vars for cleaner code
    int begin = (*newOne).begin;
```

```
    int end = (*newOne).end;
    int id = (*newOne).id;

    // if start of chunk is not a space or capital letter, increment starting point
    while ((int) text[begin] != 32) {
        if ((int) text[begin] > 64 && (int) text[begin] < 91)
            break;
        else {
            begin++;
        }
    }
    //printf("Thread Id: %d\tStart: %d\tEnd: %d\n", id, begin, end);
    int i;
    for (i = begin; i <= end; i++) {
        // space char is delimiter
        if ((int) text[i] == 32)
            localCount++;

        // first word exception
        if ((int) text[i] > 64 && (int) text[i] < 91 && (int) text[i - 1] != 32)
            localCount++;
    }

    // use mutex lock before adding local count to global variable
    pthread_mutex_t lock = PTHREAD_MUTEX_INITIALIZER;
    pthread_mutex_lock(&lock);
    wordCount += localCount;
    pthread_mutex_unlock(&lock);
}
```

Output

time ./run text

Number of Words: 14369

real 0m0.004s

user 0m0.004s

sys 0m0.002s

Windows:

Code

```
/*
* File:   wordCounter.c
* Author: Aleksandr Kibis
*
* Multi-threaded word count application. Used clock() instead of time()
* because it provides more accurate data.
*
* Compile: cl -o wordCount wordCounter.c
*
* Run: wordCount.exe <text file>
```

```
*
* Created on October 3, 2014, 6:53 PM
*/

#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <time.h>
#include <string.h>

#define EXIT_SUCCESS 0
#define EXIT_FAILURE 1

#define BUFFER_SIZE 1024 * 80
#define THREAD_COUNT 8

int getFile(char* file);
DWORD WINAPI counter(void *attributes);

// specifies data range to work on for each thread

struct threadAttr {
        int begin;
        int end;
        int id;
};

char* text;
int* countArray;
int wordCount;

/*
*
*/
int main(int argc, char** argv) {

        int fileSize;
        int chunkSize;

        HANDLE threadArray[THREAD_COUNT];
        DWORD threadIdArray[THREAD_COUNT];
        struct threadAttr *input;

        int i;
        int offset = 0;

        if (argv[1] == NULL) {
                printf("Usage: ./wordCount <text file>\n");
                exit(EXIT_FAILURE);
        }

        countArray = (int*)malloc(THREAD_COUNT * sizeof (int));
        text = (char*)malloc(BUFFER_SIZE * sizeof (char));

        // write input file to global memory

        fileSize = getFile(argv[1]);
        //printf("File size: %d\n", fileSize);

        // exit if write to buffer is unsuccessful
        if (fileSize < 0) {
                free(text);
                return (EXIT_FAILURE);
        }
        // count number of words
        else {

                // split data into evenly size chunks
                chunkSize = fileSize / THREAD_COUNT;
```

```
                //printf("Split check: %c\n", text[chunkSize]);

                // create threads and run word count on each chunk


                // thread initialization

                for (i = 0; i < THREAD_COUNT; i++) {
                        // each thread must have it's own struct so that data won't be overwritten
                        input = (struct threadAttr*)malloc(sizeof (struct threadAttr));
                        (*input).begin = i * chunkSize;
                        if (i == THREAD_COUNT - 1)
                                (*input).end = fileSize;
                        else
                                (*input).end = i * chunkSize + chunkSize;
                        (*input).id = i;
                        //retval = pthread_create(&threadArray[i], &pta, counter, (void*) input);
                        threadArray[i] = CreateThread(NULL, 0, counter, input, 0, &threadIdArray[i]);
                }

                // wait for all threads to finish
                for (i = 0; i < THREAD_COUNT; i++) {
                        //pthread_join(threadArray[i], NULL);
                        WaitForSingleObject(threadArray[i], INFINITE);

                }

                // ITERATIVE IMPLEMENTATION

                //        int wordCount = 0;
                //
                //        for (i = 0; i < fileSize; i++){
                //            // space char is delimiter
                //            if((int)text[i] == 32)
                //                wordCount++;
                //
                //            // first word exception
                //            if((int)text[i] > 64 && (int)text[i] < 91 && (int)text[i-1] != 32)
                //                wordCount++;
                //        }
                //        countArray[0] = wordCount;
                //        printf("Word Count: %d\n", countArray[0]);

                printf("Number of Words: %d\n", wordCount);
                free(input);
                free(text);
                return (EXIT_SUCCESS);

        }


}

int getFile(char* file) {
        int count = 0;
        size_t readCount;
        FILE *fd;
        int ch;

        //printf("Input file name: %s\n", file);


        fd = fopen("text", "r");
        while (1) {
                ch = fgetc(fd);
                if (ch == EOF)
                        break;
                ++count;
        }
```

```
        rewind(fd);
        //printf("Number of chars in file: %d\n", count);


        readCount = fread(text, 1, count, fd);
        fclose(fd);

        if (count == (int) readCount) {
                        //printf("Read file into buffer: SUCCESS\n");
                        //printf("***********************************************\n");
                //printf("%s\n", text);
                return count;
        } else {
                printf("Buffer read error. Exiting.\n");
                return -1;
        }

        //printf("Chars read: %d\n", (int)readCount);



}

DWORD WINAPI counter(void *attributes) {
        int localCount = 0;
        HANDLE mutex;
        struct threadAttr *newOne;
        int begin;
        int end;
        int id;
        int i;

        newOne = (struct threadAttr*) attributes;

        // assign struct values to new vars for cleaner code
        begin = (*newOne).begin;
        end = (*newOne).end;
        id = (*newOne).id;

        // if start of chunk is not a space or capital letter, increment starting point
        while ((int) text[begin] != 32) {
                if ((int) text[begin] > 64 && (int) text[begin] < 91)
                        break;
                else {
                        begin++;
                }
        }
        //printf("Thread Id: %d\tStart: %d\tEnd: %d\n", id, begin, end);
        for (i = begin; i <= end; i++) {
                // space char is delimiter
                if ((int) text[i] == 32)
                        localCount++;

                // first word exception
                if ((int) text[i] > 64 && (int) text[i] < 91 && (int) text[i - 1] != 32)
                        localCount++;
        }

        // use mutex lock before adding local count to global variable
        mutex = CreateMutex(NULL, FALSE, NULL);
        WaitForSingleObject(mutex, INFINITE);
        wordCount += localCount;
        ReleaseMutex(mutex);
        CloseHandle(mutex);

        return 0;
}
```

Output

PS C:\Users\rusky\Dropbox\Fall 2014\415\415 - Homework 4 - Word Count\415 - Homework 4 - Word Count> Measure-Command {start-process run.exe text}

Days : 0

Hours : 0

Minutes : 0

Seconds : 0

Milliseconds : 55

Ticks : 553017

TotalDays : 6.40065972222222E-07

TotalHours : 1.53615833333333E-05

TotalMinutes : 0.000921695

TotalSeconds : 0.0553017

TotalMilliseconds : 55.3017