Assignment 1 Report - Critical Sections. Locks, Barriers, and Condition Variables

Dean Tsankov, Kacper Lisik

February 2, 2025

Introduction

In this assignment, we learn how to solve the critical section problem, i.e., to access shared variables with mutual exclusion. We implement condition synchronization (waiting for conditions) using condition variables. We build a concurrent (parallel) program of multiple threads with shared variables, namely to create and join concurrent threads and pass input parameters and results. Utilize thread synchronization with locks, barriers, and condition variables and compile and run a concurrent programs using Pthreads.

Problem 1 - Compute the Sum, Min, and Max of Matrix Elements

As an introduction to the concurrent programming workflow with pthreads we are given the matrixSum.c program which we have to manipulate in order to complete the given tasks.

Task description

```
Download matrixSum.c. The program computes a sum of matrix elements in
↪  parallel using Pthreads. Develop the following three modified
↪  versions of the program.
(a) Extend the program so that in addition to the sum, it finds and
↪  prints a value and a position (indexes) of the maximum element of the
↪  matrix and a value and a position of the minimum element of the
↪  matrix. To check your solution, initialize matrix elements to random
↪  values (rather than all ones like in the above example).
(b) Change the program developed in (a) so that the main thread prints
↪  the final results. Do not call the Barrier function, and do not use
↪  arrays for partial results, such as sums in the above example.
```

```
(c) Change the program developed in (b) so that it uses a "bag of tasks"
↪ that is represented as a row counter, which is initialized to 0. A
↪ worker gets a task (i.e., the number of the row to process) out of
↪ the bag by reading a value of the counter and incrementing the
↪ counter as described for the matrix multiplication problem in Slides
↪ 27-29 about the "bag of tasks" in Lecture 5.
```

Key points of solution

Task Part A

Matrix generation now uses `rand()%99` with `srand(time(NULL))` to generate a different seed each time.

Since the thread initialization function receives only a single variable in order to pass multiple arguments we define the following 3 structures, 2 for the global maxima and their coordinates and one for all arguments to the thread:

```
struct Min {
    int value;
    int xCoordinate;
    int yCoordinate;
};
struct Max {
    int value;
    int xCoordinate;
    int yCoordinate;
};
struct Arguments
{
    struct Min* min;
    struct Max* max;
    long id;
};
```

We then initialize the different threads as follows now:

```
    /* do the parallel work: create the workers */
struct Arguments pArg [numWorkers];
start_time = read_timer();
for (l = 0; l < numWorkers; l++)
{
        pArg[l].min = &min;
        pArg[l].max = &max;
        pArg[l].id = l;
```

```
        pthread_create(&workerid[l], &attr, Worker, (void*) &pArg[l]);
    }
    pthread_exit(NULL);
```

In the thread worker function we de-reference the passed arguments like so:

```
    long myid = (long)((struct Arguments*)arg)->id;
    int total, i, j, first, last;
    struct Max* maxPoint = ((struct Arguments*)arg)->max;
    struct Min* minPoint = ((struct Arguments*)arg)->min;
```

We then search and set the global maxima values using locks as such:

```
    total += matrix[i][j];
    if(matrix[i][j]>(*maxPoint).value){
        pthread_mutex_lock(&maxLock);
            if(matrix[i][j]>(*maxPoint).value){
                (*maxPoint).value = matrix[i][j];
                (*maxPoint).xCoordinate = i;
                (*maxPoint).yCoordinate = j;
            }
        pthread_mutex_unlock(&maxLock);
    }
    if(matrix[i][j]<(*minPoint).value){
        pthread_mutex_lock(&minLock);
            if(matrix[i][j]<(*minPoint).value){
                (*minPoint).value = matrix[i][j];
                (*minPoint).xCoordinate = i;
                (*minPoint).yCoordinate = j;
            }
        pthread_mutex_unlock(&minLock);
    }
```

Task Part B

The total is now calculated with a lock in a shared variable in the Worker function:

```
    void *Worker(void *arg)
    {
        (...)
        int* totalPoint = ((struct Arguments *)arg)->matrixTotal;
```

```
    (...)
    pthread_mutex_lock(&sumLock);
        *totalPoint += matrix[i][j];
    pthread_mutex_unlock(&sumLock);
    (...)
    pthread_exit(NULL);
}
```

Potentially the total could have been passed as an argument to the exit function and then the main thread could have dealt with the summation.

The main thread now looks like so, after introducing the join routine to wait for the completion of all threads, which allows us to not use the barrier function;

```
(...)
start_time = read_timer();
for (l = 0; l < numWorkers; l++)
    {
    pArg[l].min = &min;
    pArg[l].max = &max;
    pArg[l].id = l;
    pArg[l].matrixTotal = &total;
            pthread_create(&workerid[l], &attr, Worker, (void*) &pArg[l]);
}
for (l = 0; l < numWorkers; l++){
    //wait for threads to terminate before continuing
    pthread_join(workerid[l],NULL);
}
/* get end time */
end_time = read_timer();
```

Task Part C

Implementing the bag of tasks approach yields a main thread structured like so, where instead of the for loop we increment a separate counter which relates the tasks. While working with the counter we lock it to avoid conflicts:

```
(...)
 start_time = read_timer();
while (true)
{
    pthread_mutex_lock(&counterLock);
        pArg.id = rowCounter;
```

```
            rowCounter++;
        pthread_mutex_unlock(&counterLock);

        if (rowCounter >= numWorkers)
        {
            break;
        }

        pthread_create(&workerid[pArg.id], &attr, Worker, (void*) &pArg);
    }
    for (l = 0; l < numWorkers; l++)
    {
        //wait for threads to terminate before continuing
        pthread_join(workerid[l],NULL);
    }
    end_time = read_timer();
    (...)
```

Problem 5 - The Linux diff command

Task description

```
Develop a parallel multithreaded program (in C using Pthreads or in Java)
↪  that implements a simplified version of the Linux diff command for
↪  comparing two text files. The command is invoked as follows:
-> diff filename1 filename2
The diff command compares corresponding lines in the given files and
↪  prints both lines to the standard output if the lines differ. If one
↪  of the files is longer than the other, diff prints all extra lines in
↪  the longer file to the standard output.
```

Key points of solution

Since there are already built in functions in C for reading lines from a file and comparing strings I doubt we would be able to parallelize those in order to optimize their execution so We decided to implement to producer consumer paradigm, where there is one thread that will read the following lines in the 2 separate files (i.e. the producer) while a second thread will be the consumer and compare the yielded strings.

Furthermore, considering the fact that the buffer between the two threads will have only one slot we could make use of the semaphore structure.

Then the Producer thread function has the form:

```
void *Producer(void *arg)
{
    while (!filesEnd)
    {
        sem_wait(&empty);
        char *statusf1 = fgets(((struct Arguments *)arg)->linef1, 100,
        ↪  ((struct Arguments *)arg)->file1);
        char *statusf2 = fgets(((struct Arguments *)arg)->linef2, 100,
        ↪  ((struct Arguments *)arg)->file2);
        if(!statusf1 && !statusf2){
            filesEnd = 1;
            memset((((struct Arguments *)arg)->linef1), 0,
            ↪  strlen((((struct Arguments *)arg)->linef1)));
            memset((((struct Arguments *)arg)->linef2), 0,
            ↪  strlen((((struct Arguments *)arg)->linef2)));
        }else{
            if(!statusf1){
                memset((((struct Arguments
                ↪  *)arg)->linef1),0,strlen((((struct Arguments
                ↪  *)arg)->linef1)));
            }else if(!statusf2){
                memset((((struct Arguments
                ↪  *)arg)->linef2),0,strlen((((struct Arguments
                ↪  *)arg)->linef2)));
            }
        }
        sem_post(&full);
    }
}
```

When we have access by the semaphore we retrieve the next lines in the files and set them to the argument pointers. We consider the three edge cases: when both lines are empty the program should finish; when either of the files is over but the other is not we write an empty line to the argument pointer so that we could continue printing them.

Next is the consumer function:

```
void *Consumer(void *arg)
{
    while (!filesEnd)
    {
        sem_wait(&full);
        if ((!(((struct Arguments *)arg)->linef1) &&
```

```
            !((struct Arguments *)arg)->linef2))
        {
            filesEnd = 1;
        }
        else if (strcmp(((struct Arguments *)arg)->linef1,
            ((struct Arguments *)arg)->linef2) != 0)
        {
            char path1[100];
            char path2[100];
            strcpy(path1, ((struct Arguments *)arg)->linef1);
            strcpy(path2, ((struct Arguments *)arg)->linef2);
            path1[strcspn(path1, "\n")] = 0;
            path2[strcspn(path2, "\n")] = 0;
            printf("f1: %s -> f2: %s\n", path1, path2);
        }
        sem_post(&empty);
    }
}
```

Similarly we enter when there is access by the semaphore, we check if by chance the set argument lines are empty in order to terminate execution otherwise we use the string compare method and print out any lines that differ.

The main method has the following lines to initiate the exectution of the threads and the time benchmarking:

```
(...)
    start_time = read_timer();
    pthread_create(&producerid, &attr, Producer, (void *)&dArg);
    pthread_create(&consumerid, &attr, Consumer, (void *)&dArg);
    pthread_join(producerid, NULL);
    pthread_join(consumerid, NULL);
    end_time = read_timer();
    printf("The execution time is %g sec\n", end_time - start_time);
(...)
```

Alternatively we also included a pthread implementation of this problem using a mutex conditional variable and a counter, in the source code segment of this assignment. That solution follows almost the same logic as the presented one except for a slight reordering of the procedures in each function to allow for the mutual exclusion.

Problem 6 - Find Palindromes and Semordnilaps

Task description

```
There is an online dictionary in /usr/dict/words or /usr/share/dict/words
↪  under Linux. The file is used, in particular, by the Linux spell
↪  command and contains a few thousand words, e.g., 235886 words. You
↪  can download a smaller file called words containing 25143 words.
Recall that palindromes are words or phrases that read the same forward
↪  and backward, such as "noon" or "radar," i.e., if you reverse all the
↪  letters, you get the same word or phrase. Reverse pairs or
↪  semordnilaps are words whose reverse forms are different valid words,
↪  such as "draw" and "ward". Your task is to find all palindromes and
↪  all  semordnilaps in the dictionary. For example, "noon" is a
↪  palindrome; hence, its reverse is trivially in the dictionary. A word
↪  like "draw" is semordnilap and "ward" is also in the dictionary.
Your parallel program should use W worker processes, where W is a
↪  command-line argument. Use the workers just for the compute phase
↪  (i.e., to find the palindromic words); do the input and output phases
↪  sequentially. Each worker should count the number of words
↪  (palindromes and semordnilaps) that it finds. Your program should
↪  write the palindromes and semordnilaps to a results file and output
↪  the total number of palindromes and semordnilaps in the dictionary
↪  and the number of palindromes and semordnilaps found by each worker.
```

Key points of solution

After sequentially reading the words file, we ensure that the words are sorted in alphabetical order. They appear to be sorted, but to be certain, we use the `std::sort()` command, which is computationally inexpensive if the words are already sorted or nearly sorted. We distribute the work and create N threads, with each thread checking whether any of $num\_of\_words/num\_of\_threads$ words is a palindrome or a semordnilap:

```
(...)
    for (int i = 0; i < workers; i++){
        thrd_data[i].start = (words.size()/workers) * i;
        thrd_data[i].end = (i == workers - 1)? words.size()-1 :
        (words.size()/workers) * (i+1);
        thrd_data[i].tid = i;
        if (pthread_create(&tid[i], NULL, thread_fun, &thrd_data[i]) != 0)
            cerr << "error creating thread " << i << "\n";
    }
(...)
```

Thread function:

```
(...)
void* thread_fun (void* arg){
    thread_data* data = (thread_data*) arg;
    data->palindromes = 0;
    data->semordnilaps = 0;
    for (int i = data->start; i < data->end; i++){
        // calculate the reverse
        string reverse = words[i];
        for (int j = 0; j < reverse.size() / 2; j++)
            swap(reverse[j], reverse[reverse.size() - 1 - j]);

        // check if palindrome
        if(compare_strings(reverse, words[i]) == 0){
            data->palindromes_vector.pb(reverse);
            data->palindromes++;
            continue;
        }

        // look for semordnilaps - fast
        if(binsearch(0, words.size()-1, reverse) != -1){
            data->semordnilaps_vector.pb(words[i]);
            data->semordnilaps++;
        }
    }
    return NULL;
}
(...)
```

Each thread goes through its assigned list of words, computes the reverse of every word, and checks if the word is a palindrome. Then, using binary search, it looks through the word list to see if the reverse of the given word is also in the word list.

Binary search function:

```
(...)
// returns index of the string s in words vector or -1 if s is not found
int binsearch(int start, int end, string& s){

    while(start + 1 != end){
        int mid = (start  + end) / 2;
        if(compare_strings(words[mid], s) == 0) return mid;
        (compare_strings(words[mid], s) < 0 )? start = mid : end = mid;
```

```
    }

    if(compare_strings(words[end], s) == 0 )    return end;
    if(compare_strings(words[start], s) == 0 )    return start;
    return -1;
}
(...)
```

The `compare_strings` function returns `-1` if the first argument (string `a`) is alphabetically smaller than the second argument (string `b`), returns `0` when the words are equal, and returns `1` otherwise.

```
(...)
// if string a < b => return -1, if a == b return 0, if a > b return 1
int compare_strings(const string& a, const string& b)
{
    int len_min = min(a.length(), b.length());

    for (int i = 0; i < len_min; i++){
        if (a[i] < b[i])  return -1;
        if (a[i] > b[i])  return 1;
    }
    if(a.length() == b.length())    return 0;
    if(a.length() < b.length())     return -1;
    return 1;
}
(...)
```

The average execution time with 6 threads is around 0.0018sec

Conclusion

In this assignment we learned how to utilize the pthreads library and general parallelism paradigms to implement some simple programs. We also dealt with the critical section problems each of them rose. We gained an understanding of how parallelism improves performance and allows our programs to make the most out of the modern day multi-core multi-thread processors.