Introduction

The purpose of this project is to practice Pthread programming by solving various problems. The objectives of this project is to learn:

1. Get familiar with the Pthread creation and termination.
2. How to use mutexes and conditional variables in Pthread.
3. How to design efficient solutions for mutual exclusion problems.

Project submission

For each project, create a gzipped file containing the following items, and submit it to jrao@uccs.edu via email. Include “CS4500/5500_Spring2016_ProjectX” in the title of your email, where “X” is the project number.

1. A report that briefly describes how did you solve the problems and what you learned.
2. The POSIX thread programming codes and files containing your test cases.

Each team should specify a virtual machine (VM) that the instructor can login to check the project results. Name the VM as “CS4500/5500_LastNameoftheFirstMember_LastNameoftheSecondMember” and create a new user account called instructor in your VM. Place your code in the home directory of the instructor account (i.e., /home/instructor). Make sure the instructor has access to your code. In your submission email, include your password for the instructor account.
Assignments

Assignment 1 (30 pts)

Given two character strings s1 and s2. Write a Pthread program to find out the number of substrings, in string s1, that is exactly the same as s2. For example, suppose number_substring(s1, s2) implements the function, then number_substring(“abcdab”, “ab”) = 2, number_substring(“aaab”, “a””) = 3, number_substring(“abac”, “bc”) = 0. The size of s1 and s2 (n1 and n2) as well as their data are input by users. Assume that n1 mod NUM_THREADS = 0 and n2 < n1/NUM_THREADS.

The following is a sequential solution of the problem. read_f() reads the two strings from a file named “string.txt” and num_substring() calculates the number of substrings.

```c
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define MAX 1024

int total = 0;
int n1,n2;
char *s1,*s2;
FILE *fp;

int read_f(FILE *fp)
{
    if((fp=fopen("strings.txt", "r"))== NULL){
        printf("ERROR: can’t open string.txt!\n");
        return 0;
    }
    s1=(char *)malloc(sizeof(char)*MAX);
    if(s1==NULL){
        printf("ERROR: Out of memory!\n");
        return -1;
    }
    s2=(char *)malloc(sizeof(char)*MAX);
    if(s2==NULL){
        printf("ERROR: Out of memory\n");
        return -1;
    }
    /*read s1 s2 from the file*/
    s1=fgets(s1, MAX, fp);
    s2=fgets(s2, MAX, fp);
    n1=strlen(s1); /*length of s1*/
    n2=strlen(s2)-1; /*length of s2*/
    if(s1==NULL || s2==NULL || n1<n2) /*when error exits*/
        return -1;
}

int num_substring(void)
{
    int i,j,k;
    int count;
    for (i = 0; i <= (n1-n2); i++)
    {
        count=0;
        for(j = i,k = 0; k < n2; j++,k++) /*search for the next string of size of n2*/
            if (*(s1+j)!=(s2+k))
                break;
        else
            count++;
        if(count==n2)
```

2
Write a parallel program using Pthread based on this sequential solution.

HINT: Strings s1 and s2 are stored in a file named “string.txt”. String s1 is evenly partitioned for NUM_THREADS threads to concurrently search for matching with string s2. After a thread finishes its work and obtains the number of local matchings, this local number is added into a global variable showing the total number of matched substrings in string s1. Finally this total number is printed out. You can find an example of the “string.txt” in the attached source code.

Assignment 2 (30 pts)

Using condition variables to implement a producer-consumer algorithm. Here we have two threads: one producer and one consumer. The producer reads characters one by one from a string stored in a file named “message.txt”, then writes sequentially these characters into a circular queue. Meanwhile, the consumer reads sequentially from the queue and prints them in the same order. Assume a buffer (queue) size of 5 characters. Write a Pthread program using conditional variables.

Assignment 3 (40 pts)

Read the following program and modify the program to improve its performance.

```c
/* list-forming.c:
   Each thread generates a data node, attaches it to a global list. This is repeated for K times.
   There are num_threads threads. The value of "num_threads" is input by the student.
*/
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <pthread.h>
#include <sys/time.h>
#include <sys/param.h>
#include <sched.h>
#define K 200  // generate a data node for K times in each thread

struct Node
{
    int data;
    struct Node* next;
};

struct list
{
    struct Node* header;
};
```
struct Node * tail;
};

pthread_mutex_t mutex_lock;

struct list *List;

void bind_thread_to_cpu(int cpuid) {
    cpu_set_t mask;
    CPU_ZERO(&mask);
    CPU_SET(cpuid, &mask);
    if ( sched_set_thread_affinity(0, sizeof(cpu_set_t), &mask) ) {
        fprintf(stderr, "sched_set_thread_affinity");
        exit(EXIT_FAILURE);
    }
}

struct Node* generate_data_node() {
    struct Node *ptr;
    ptr = (struct Node *) malloc(sizeof(struct Node));
    if ( NULL != ptr ){
        ptr->next = NULL;
    } else {
        printf("Node allocation failed!\n");
    }
    return ptr;
}

void *producer_thread(void *arg) {
    bind_thread_to_cpu(*((int*)arg)); // bind this thread to a CPU
    struct Node *ptr, tmp;
    int counter = 0;
    /\* generate and attach K nodes to the global list */
    while( counter < K ) {
        ptr = generate_data_node();
        if ( NULL != ptr ) {
            while(1) {
                /\* access the critical region and add a node to the global list */
                if( !pthread_mutex_trylock(&mutex_lock) ) {
                    ptr->data = 1; // generate data
                    /\* attach the generated node to the global list */
                    if( List->header == NULL ) {
                        List->header = List->tail = ptr;
                        pthread_mutex_unlock(&mutex_lock);
                    } else {
                        List->tail->next = ptr;
                        List->tail = ptr;
                    } 
                    break;
                }
            }
        }
    }
}
int main(int argc, char* argv[])
{
    int i, num_threads;

    int NUM_PROCS; // number of CPU
    int* cpu_array = NULL;

    struct Node *tmp, *next;
    struct timeval starttime, endtime;

    num_threads = atoi(argv[1]); // read num_threads from user
    pthread_t producer[num_threads];
    NUM_PROCS = sysconf(_SC_NPROCESSORS_CONF); // get number of CPU

    if (NUM_PROCS > 0)
    {
        cpu_array = (int*)malloc(NUM_PROCS*sizeof(int));
        if (cpu_array == NULL)
        {
            printf("Allocation failed!\n");
            exit(0);
        }
        else
        {
            for (i = 0; i < NUM_PROCS; i++)
                cpu_array[i] = i;
        }
    }

    pthread_mutex_init(&mutex_lock, NULL);

    List = (struct list*)malloc(sizeof(struct list));
    if (NULL == List)
    {
        printf("End here\n");
        exit(0);
    }
    List->header = List->tail = NULL;

    gettimeofday(&starttime, NULL); // get program start time
    for (i = 0; i < num_threads; i++)
    {
        pthread_create(&(producer[i]), NULL, (void*)producer_thread, &cpu_array[i%NUM_PROCS]);
    }

    for (i = 0; i < num_threads; i++)
    {
        if (producer[i] != 0)
            pthread_join(producer[i], NULL);
    }

    gettimeofday(&endtime, NULL); // get the finish time

    if (List->header != NULL)
    {
        next = tmp = List->header;
    }
while( tmp != NULL )
{
    next = tmp->next;
    free(tmp);
    tmp = next;
    }
    } if( cpu_array!= NULL)
        free(cpu_array);
    /* calculate program runtime */
    printf("Total run time is %ld microseconds.\n", (endtime.tv_sec-starttime.tv_sec) * \
bracht 1000000/(endtime.tv_usec-starttime.tv_usec));
    return 0;
}