Tutorial 5: POSIX THREADS

Faculty of Engineering and Applied Science

SOFE 3950U : Operating Systems | CRN: 74171

Due: February 26th, 2024

Group 8

Daniel Amasowomwan [100787640]
daniel.amasowomwan@ontariotechu.net

Stanley Watemi [100648403]
stanley.watemi@ontariotechu.net

Fayomi Toyin [100765921]
oluwatoyin.fayomi@ontariotechu.net

Conceptual Questions

1. Read the pthread documentation and explain the following three functions: pthread_create, pthread_join, pthread_exit.

   Pthread_create- It is a function in the POSIX library used to create new threads with attributes specified by attr within a process. It takes several arguments like the thread attributes, a pointer to a thread and so on. It allows concurrent execution of multiple threads in a program.
   Pthread_join- It suspends execution of the calling thread until the target thread terminates. It takes two arguments; the ID of the thread to wait for and a pointer to a location where the exit status of the thread will be stored. It allows the main thread to wait until the specified thread has finished executing before it continues.
   Pthread_exit- It terminates the calling thread and makes the value available to any successful join with the terminating thread. It allows a thread to exit its execution while still allowing other threads in the process to continue running.

2. Explain how the memory of threads works in comparison to processes, do threads share the same memory, can threads access the memory of other threads?

   Threads within the same process share memory space in comparison to processes. Threads share resources within the same process. Threads are not independent of one another like processes are, they share with other threads code sections, data sections and OS resources. Unlike processes, a thread has its own program counter, stack space and register set.

3. Name the differences between multithreading and multiprocessing (multiple processes). What are the advantages and disadvantages of each?

   - In Multiprocessing, many processes are executed simultaneously while in Multithreading, many threads are executed simultaneously.
   - Process creation is time-consuming in multiprocessing whereas in multithreading, process creation is economical
   - Multiprocessing is classified into symmetric and asymmetric whereas multithreading is not classified into any categories.
   - CPUs are added to increase computing power in multiprocessing while in multithreading, to increase computing power more threads are created in a single process.

Advantages of Multiprocessing

- It allows parallel execution of multiple tasks across multiple CPUs which improves performance
- An error in one process does not typically affect other processes
- Utilization of multi-core systems
- Each process has its own memory space which provides strong isolation and protects processes from interfering with each other.

Disadvantages of Multiprocessing

- Creating multiple processes can lead to higher overhead compared to multithreading.
- Communication between processes are slower compared to multithreading
- Data sharing is less efficient

Advantages of Multithreading

- It allows concurrent execution of multiple tasks just like in multiprocessing
- Threads within the same process can share memory space thereby simplifying communication between threads within the same process.
- It simplifies programming model compared to multiprocessing
- It enhances responsiveness in applications by allowing different tasks to run concurrently.

Disadvantage of Multithreading

- Multithreaded programs are prone to concurrency bugs
- It introduces complexity to application design and development.
- Applications that are multithreaded are vulnerable to deadlocks where two or more threads are blocked waiting for each other to release resources.
- Threads competing for resources may experience leading performance degradation or inefficiency.

4. Provide an explanation of mutual exclusion, what is a critical section?

   Mutual exclusion is a property of process synchronization used in concurrent programming. It states that no two processes can exist in the critical section at any given time. It ensures that only one thread at a time can access a shared resource. This is to avoid data corruption or behavior that can occur when multiple threads access shared resources concurrently.

   Critical section refers to a segment of code or part of a program that is executed by multiple concurrent threads or processes which access shared resources. It is critical because improper synchronization can lead to data corruption or other concurrency issues.

5. Research the functions used to perform mutual exclusion with pthreads and explain the purpose of each function

   a. **Pthread_mutex_lock**- it acquires a mutex, blocking until it is available
   b. **Pthread_mutex_init**- It initializes a mutex
   c. **Pthread_mutex_unlock-** It releases a mutex allowing other threads to acquire it
   d. **Pthread_mutex_destroy**- It destroys a mutex
   e. **Pthread_mutex_trylock**- It attempts to acquire a mutex without blocking. It immediately returns a success or failure status.

Application Question

1. Create a program that does the following, make sure you can complete this before moving to further questions, when compiling add the -lpthread argument, if you are using gcc use the -pthread argument.
    - Creates two threads, the first uses a function hello_world() which prints hello world, the second uses a function goodbye() which prints goodbye.
    - Each function has a random sleep duration before printing the output
    - After running your program a few times you should notice that the order of hello world and goodbye being printed to the screen is not consistent, as each thread is executing independently.

```c
C question1.c > _XOPEN_SOURCE
 6
 7  void *hello_world(void *arg){
 8      //random sleep btw 1-3 sec
 9      sleep(rand() % 3 + 1);
10
11      printf("Hello World\n");
12
13      pthread_exit(NULL);
14  };
15  void *goodbye(void *arg){
16      //random sleep btw 1-3 sec
17      sleep(rand() % 3 + 1);
18
19      printf("Goodbye\n");
20
21      pthread_exit(NULL);
22  };
23
24  int main(void){
25      pthread_t thread1, thread2;
26
27      //create threads
28      if(pthread_create(&thread1, NULL, hello_world, NULL) != 0){
29          perror("Error creating thread 1");
30          return EXIT_FAILURE;
31      }
32
33      if(pthread_create(&thread2, NULL, goodbye, NULL) != 0){
34          perror("Error creating thread 2");
35          return EXIT_FAILURE;
36      }
37
38      pthread_join(thread1, NULL);
39      pthread_join(thread2, NULL);
40
41      return EXIT_SUCCESS;
42  }
```

Ln 1, Col 1 Spaces: 4 UTF-8 LF

```
danielamas@Linux22:~/school/opsystems/tutorials/tut5$ ./question1
Goodbye
Hello World
danielamas@Linux22:~/school/opsystems/tutorials/tut5$ ./question1
Hello World
Goodbye
danielamas@Linux22:~/school/opsystems/tutorials/tut5$ ./question1
Goodbye
Hello World
danielamas@Linux22:~/school/opsystems/tutorials/tut5$ ./question1
Hello World
Goodbye
```

2. Create a program that does the following.
    - Prompts the professor for five student's grades.
    - Creates 5 threads, one for each student.
    - Each thread uses a function called bellcurve(grade) which takes as an argument the grade and bellcurves it by multiplying the grade by 1.50 and then printing the bellcurved grade to the terminal.
    - The program must create the 5 threads and initialize them only after receiving all 5 grades.

C question2.c > ...

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

void *bellcurve (void *arg){
    (void) arg;
    int *grade = (int *) arg;
    double bellcurved = (*grade) * 1.50;
    printf ("Grade after Bellcurve: %.2f\n", bellcurved);
        pthread_exit(NULL);

}

int main() {
    int grades[5];

    printf("Enter 5 student's grades: \n");
    for (int i=0; i<5; i++) {
     scanf("%d", &grades[i]);
    }

    pthread_t thread[5];
    for (int i=0; i<5; i++) {
    pthread_create(&thread[i], NULL, bellcurve, (void *) &grades[i]);
    }

    for (int i=0; i<5; i++) {
    pthread_join(thread[i], NULL);
    }

    return 0;
}
```

```
danielamas@Linux22:~/school/opsystems/tutorials/tut5$ ./question2
Enter 5 student's grades:
45
78
98
69
88
Grade after Bellcurve: 67.50
Grade after Bellcurve: 117.00
Grade after Bellcurve: 132.00
Grade after Bellcurve: 147.00
Grade after Bellcurve: 103.50
```

3. Create a program that does the following.
    - Prompts the professor for five student's names, student_id, and grade.
    - Creates five threads, one for each student.
    - Create a struct named student containing three members, name student_id, and grade.
    - Create a function bellcurve(student) which takes a student (the struct type) as an argument and bellcurves the grades by multiplying it by 1.50 and prints the student name, id, and bellcurved grade to the terminal.
    - The program must create the 5 threads and initialize them only after receiving all 5 grades.

```c
struct Student {
    char name[50];
    int student_id;
    int grade;
};

void *bellcurve(void *arg){
    struct Student *student = (struct Student *)arg;

    //multiply grade by 1.50 (bell curve)
    double curvedGrade = student->grade * 1.50;

    printf("Student: %s\t, ID: %d\t, Bell curved grade: %.2f\n",
    student->name, student->student_id, curvedGrade);

    pthread_exit(NULL);
}

//Global variables
struct Student students[5];
pthread_t threads[5];
```

```c
//Global variables
struct Student students[5];
pthread_t threads[5];

int main () {
    printf("Enter Student's name, ID, and grade: \n");

    for(int i =0; i<5; i++){
        printf("Student %d name: ", i + 1);
        scanf("%s", students[i].name);
        printf("Student %d ID: ", i + 1);
        scanf("%d", &students[i].student_id);
        printf("Student %d grade: ", i + 1);
        scanf("%d", &students[i].grade);
    }

    //create 5 threads
    for (int i=0; i<5; i++){
        if (pthread_create(&threads[i], NULL, bellcurve, (void *)&students
        [i]) != 0){
            perror("Error creating thread");
            return EXIT_FAILURE;
        }
    }

    for(int i = 0; i<5;i++){
        pthread_join(threads[i], NULL);
    }

    return EXIT_SUCCESS;
}
```

```
Enter Student's name, ID, and grade:
Student 1 name: Daniel
Student 1 ID: 100787640
Student 1 grade: 76
Student 2 name: Stanley
Student 2 ID: 100648403
Student 2 grade: 80
Student 3 name: Fayomi
Student 3 ID: 100765921
Student 3 grade: 78
Student 4 name: John
Student 4 ID: 100898660
Student 4 grade: 87
Student 5 name: Doe
Student 5 ID: 100779840
Student 5 grade: 89
Student: Daniel , ID: 100787640 , Bell curved grade: 114.00
Student: Stanley        , ID: 100648403 , Bell curved grade: 120.00
Student: Fayomi , ID: 100765921 , Bell curved grade: 117.00
Student: John   , ID: 100898660 , Bell curved grade: 130.50
Student: Doe    , ID: 100779840 , Bell curved grade: 133.50
```

4. Create a program that does the following.
    - Prompts the professor for ten student's grades,
    - Creates ten threads, one for each student.
    - Create a function class_total(grade) which adds the grade to a global variable total_grade using the operator += to increment total_grade
    - You MUST use mutual exclusion when incrementing total_grade
    - Print the results of total grade, it should be the correct sum of all ten grades.

```
Enter grades for ten students:
Student 1 grade: 20
Student 2 grade: 27
Student 3 grade: 38
Student 4 grade: 45
Student 5 grade: 52
Student 6 grade: 59
Student 7 grade: 66
Student 8 grade: 70
Student 9 grade: 74
Student 10 grade: 80
Total Grade: 531
```

```c
int grades[10];
int total_grade = 0;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *class_total(void *arg){
    int grade = *(int *)arg;

    //Acquire the lock before updating the total grade
    pthread_mutex_lock(&mutex);

    total_grade += grade;

    pthread_mutex_unlock(&mutex);

    pthread_exit(NULL);
}
```

```c
int main(){
    printf("Enter grades for ten students:\n");

    //Prompt for student grades
    for (int i = 0; i < 10; i++){
        printf("Student %d grade: ", i + 1);
        scanf("%d", &grades[i]);
    }

    //Create Threads
    pthread_t threads[10];
    for (int i = 0; i < 10; i++){
        if(pthread_create(&threads[i], NULL, class_total, (void *)&grades[i]) != 0){
            perror("Error creating Thread");
            return EXIT_FAILURE;
        }
    }

    //Wait for all threads to finish
    for (int i = 0; i<10; i++){
        pthread_join(threads[i], NULL);
    }

    printf("Total Grade: %d\n", total_grade);

    return EXIT_SUCCESS;
}
```

5. Create a program that does the following.
    - Reads in 10 grades from the file grades.txt using one thread with the function called read_grades()
    - You must use a barrier to wait for grades to be read by the program
    - Create 10 threads, each uses the function save_bellcurve(grade) which - Adds the grade to a global variable total_grade using the operator += to increment total_grade
    - Bellcurves the grades by multiplying it by 1.50 and adds the grade to a global variable total_bellcurve
    - Saves (appends) the bellcurved grade to the file bellcurve.txt
    - After saving all the bellcurved grades to the file, the main program then prints to the terminal the total grade and the class average before and after the bellcurve.
    - You will need to use a combination of barriers, mutual exclusion, and thread joining to complete this question.
    - NOTE: For barriers to work you may need to add the following line to the top of your source file and/or compile it with -std=gnu99 #define _XOPEN_SOURCE 600

```
danielamas@Linux22:~/school/opsystems/tutorials/tut5$ ./q5
Total Grade before bellcurve: 363
Total Grade after bellcurve: 542
Class Average before bellcurve: 36.30
Class Average after bellcurve: 54.20
danielamas@Linux22:~/school/opsystems/tutorials/tut5$
```

```c
#define NUM_GRADES 10

//Global Variables
int grades[NUM_GRADES];
int total_grade = 0;
int total_bellcurve = 0;
pthread_barrier_t barrier;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

void *read_grades(void *arg){
    FILE *file = fopen("grades.txt", "r");
    if (file == NULL) {
        perror("Error opening file");
        exit(EXIT_FAILURE);
    }

    for (int i = 0; i < NUM_GRADES; ++i) {
        fscanf(file, "%d", &grades[i]);
    }

    fclose(file);

    // Signal that grades have been read
    if (pthread_barrier_wait(&barrier) != 0 && errno != 0) {
        perror("Error waiting on barrier");
        exit(EXIT_FAILURE);
    }

    pthread_exit(NULL);
}
```

```c
void *save_bellcurve(void *arg){
    int grade = *(int *)arg;
    int bellcurve = grade * 1.50;

    // Wait for all grades to be read
    if (pthread_barrier_wait(&barrier) != 0 && errno != 0) {
        perror("Error waiting on barrier");
        exit(EXIT_FAILURE);
    }

    // Update total_grade with mutual exclusion
    if (pthread_mutex_lock(&mutex) != 0) {
        perror("Error locking mutex");
        exit(EXIT_FAILURE);
    }

    total_grade += grade;

    if (pthread_mutex_unlock(&mutex) != 0) {
        perror("Error unlocking mutex");
        exit(EXIT_FAILURE);
    }

    // Update total_bellcurve with mutual exclusion
    if (pthread_mutex_lock(&mutex) != 0) {
        perror("Error locking mutex");
        exit(EXIT_FAILURE);
    }

    total_bellcurve += bellcurve;

    if (pthread_mutex_unlock(&mutex) != 0) {
        perror("Error unlocking mutex");
        exit(EXIT_FAILURE);
    }

    // Save bellcurve to file
    FILE *bellcurve_file = fopen("bellcurve.txt", "a");
    if (bellcurve_file == NULL) {
        perror("Error opening bellcurve file");
        exit(EXIT_FAILURE);
    }

    fprintf(bellcurve_file, "%d\n", bellcurve);

    fclose(bellcurve_file);

    pthread_exit(NULL);
}
```

```c
int main(){
    pthread_t read_thread;
    pthread_t bellcurve_threads[NUM_GRADES];

    // Initialize barrier
    if (pthread_barrier_init(&barrier, NULL, NUM_GRADES + 1) != 0) {
        perror("Error initializing barrier");
        return EXIT_FAILURE;
    }

    // Create the thread to read grades
    if (pthread_create(&read_thread, NULL, read_grades, NULL) != 0) {
        perror("Error creating read thread");
        return EXIT_FAILURE;
    }

    // Create threads for bellcurve
    for (int i = 0; i < NUM_GRADES; ++i) {
        if (pthread_create(&bellcurve_threads[i], NULL, save_bellcurve, (void *)&grades[i]) != 0) {
            perror("Error creating bellcurve thread");
            return EXIT_FAILURE;
        }
    }

    // Wait for the reading thread to finish
    pthread_join(read_thread, NULL);

    // Wait for all bellcurve threads to finish
    for (int i = 0; i < NUM_GRADES; ++i) {
        pthread_join(bellcurve_threads[i], NULL);
    }

    // Destroy the barrier
    if (pthread_barrier_destroy(&barrier) != 0) {
        perror("Error destroying barrier");
        return EXIT_FAILURE;
    }

    // Print results
    printf("Total Grade before bellcurve: %d\n", total_grade);
    printf("Total Grade after bellcurve: %d\n", total_bellcurve);
    printf("Class Average before bellcurve: %.2f\n", (float)total_grade / NUM_GRADES);
    printf("Class Average after bellcurve: %.2f\n", (float)total_bellcurve / NUM_GRADES);

    return EXIT_SUCCESS;
}
```