CISC2005 Lab 06

1. Prerequisites

1. Successful installation of *nix terminal (e.g., WSL, cygwin) or a *nix operating system (e.g., Linux, Mac).
2. Successful installation of a C compiler, GCC is recommended.
3. Successful installation of a text editor, VIM or Emacs is recommended.

2. Exercises

When multiple threads or processes access shared resources (i.e., critical region or critical section), there is a possibility of race conditions or inconsistent behavior. For example, if two threads simultaneously try to modify a shared variable, the result of the operation may depend on the order in which the threads execute, which may lead to incorrect results.

In this exercise, students are required to complete the following codes to implement several basic operating system synchronization algorithms. For each program, students should capture **screenshots** of the successful execution, and answer the **questions** for the program**.** In the submission file, please attach the **added code** and **execution screenshot** in sequence.

Task:

Suppose that a shared resource can be accessed by several threads, please use a global variable “counter” to count the number of accesses to the resource.

Here is the sample code:

```c
// Global variables
static long counter = 0;
void *thread(void *vargp) {
    // Enter critical region
    for (long i = 0; i < N_ITERATIONS; i++) {
        /* Shared resource */
        counter++;
    }
    // Leave critical region
    return NULL;
}
```

Example 1: Critical region

This example shows what are the race condition and critical region.

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

#define NUM_THREADS 2
#define N_ITERATIONS 10000000 // number of iterations

// Global variables
static volatile long counter = 0;

void *thread(void *vargp) {

    // Enter critical region
    for (long i = 0; i < N_ITERATIONS; i++) {
        /* Shared resource */
        counter++;
    }
    // Leave critical region

    return NULL;
}

int main(int argc, char **argv) {
    pthread_t tid1, tid2;

    pthread_create(&tid1, NULL, thread, NULL);
    pthread_create(&tid2, NULL, thread, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    if (counter != (2 * N_ITERATIONS))
        printf("Wrong! counter=%ld.\n", counter);
    else
        printf("Correct! counter=%ld.\n", counter);

    return 0;
}
```

Code:

1. Download the sample code from UMMoodle (critical_region.c).

2. Compile the above C program and run several times. (Tips: use the following compile command.)
   gcc critical_region.c -o critical_region -l pthread.

Question 1:

1.1 Why the counter value is "Wrong"?

1.2 What happens if the number of iterations is reduced?

Requirements:

1. Capture 2 **screenshots** of different outputs.
2. Answer Question 1.

Example 2: Alternation algorithm

In practice, we will implement mutual exclusion via some algorithms to synchronize access to shared resources, and the Alternation algorithm is the simplest method among them. Complete the code at the **blank underline** to implement the Alternation algorithm and answer questions.

Code:

1. Download the sample code from UMMoodle (alternation.c).
2. Compile the above C program and run several times. (Tips: use the following compile command.)
   gcc alternation.c -o alternation -l pthread.

Incomplete code

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

#define TRUE 1
#define NUM_THREADS 2
#define N_ITERATIONS 100000000

// Global variables
static volatile long counter = 0;
static volatile int turn = 1;


// Thread 1
```

```c
void *thread_1(void *arg) {
    while (TRUE) {
        while (turn != ___???___) {
            // Wait
        }

        // Enter critical region
        for (long i = 0; i < N_ITERATIONS; i++)
            counter++;

        // Leave critical region
        turn = ___???___;

        break;
    }

    pthread_exit(NULL);
}

// Thread 2
void *thread_2(void *arg) {

    while (TRUE) {
        while (turn != ___???___) {
            // Wait
        }

        // Enter critical region
        for (long i = 0; i < N_ITERATIONS; i++)
            counter++;
        turn = ___???___;
        // Leave critical region

        break;
    }

    return NULL;
}

int main() {
    pthread_t tid1, tid2;

    pthread_create(&tid1, NULL, thread_1, NULL);
    pthread_create(&tid2, NULL, thread_2, NULL);
    pthread_join(tid1, NULL);
    pthread_join(tid2, NULL);

    printf("counter=%ld.\n", counter);

    return 0;
}
```

Question 2:

2.1 Please complete the code for the alternation algorithm.

2.2 Why does it satisfy the mutual exclusion?

2.3 What are the drawbacks of the alternation algorithm?

Requirements:

1. Capture 1 **screenshots** of the successful execution.
2. Answer Question 2.

Example 3: Peterson's algorithm

Peterson's algorithm is a more flexible method. Complete the code at the blank underline to implement the Peterson's algorithm and answer questions.

Code:

1. Download the sample code from UMMoodle (peterson.c).
2. Compile the above C program and run several times. (Tips: use the following compile command.)
   gcc peterson.c -o peterson -l pthread.

Incomplete code

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

#define ENTER 1
#define LEAVE 0
#define NUM_THREADS 2
#define N_ITERATIONS 10000000

```

```c
// Global variables
static volatile long counter = 0;
static volatile int turn = 0;
static volatile int flag[NUM_THREADS] = {0, 0};

// Increment function
void *increment(void *arg) {
    int thread_id = *(int*)arg;

    flag[thread_id] = ENTER;
    turn = ___???___;
    while (flag[___???___] && turn == ___???___) {
        // Wait
    }

    // Enter critical region
    for (long i = 0; i < N_ITERATIONS; i++) {
        counter++;
    }

    // Leave critical section
    flag[thread_id] = LEAVE;

    return NULL;
}

int main() {
    pthread_t threads[NUM_THREADS];
    int thread_ids[NUM_THREADS];

    // Create threads
    for (int i = 0; i < NUM_THREADS; i++) {
        thread_ids[i] = i;
        pthread_create(&threads[i], NULL, increment, &thread_ids[i]);
    }

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

    printf("counter=%ld.\n", counter);

    return 0;
}
```

Question 3:

3.1 Please complete the code for the Peterson's algorithm.

3.2 Why does it satisfy the mutual exclusion?

Requirements:

1. Capture 1 **screenshots** of the successful execution.
2. Answer Question 3.

Example 4: Lock

Locks allow threads or processes to acquire exclusive access to shared resources, ensuring that only one thread or process can access the shared resource at a time. This prevents race conditions and ensures the correctness of the program. This example shows how to implement a spinlock.

Code:

1. Download the sample code from UMMoodle (lock.c).
2. Test 2 lock algorithms. Do they both work?

```c
/***********1st algorithm*******/
while (*lock) {
        // Wait
}
*lock = TRUE;
/********************************/

/*********2nd algorithm*********/
while (test_and_set(lock, TRUE) == TRUE) {
       // Wait
}
/********************************/
```

3. Compile the above C program and run several times. (Tips: use the following compile command.)
   gcc lock.c -o lock -l pthread.

Question 4:

4.1 What's the difference between the 2 lock algorithms in the "acquire" function?

4.2 Does the 1st algorithm work? Why?

4.3 Does the 2nd algorithm work? How does the "test and set" operation work?

Requirements:

1. Capture 2 **screenshots** of the 2 lock algorithms.
2. Answer Question 4.