

# Mini-Lab 4: Pthread & Race Condition

COMP3230, The University of Hong Kong

Sept. 2024

## Total 1 point

### Objective

At the end of this mini-lab, you will be able to:

- Gain hands-on experience in multi-thread programming using Pthread.
- Identify a typical adding race condition and address it using condition variables or semaphores.

### Instructions

Pthread refers to **POSIX threads**, a threading standard in Unix-like systems that allows for creating and managing multiple threads within a process. A thread is a smaller unit of a process that shares the same memory space, allowing concurrent execution.

Functions like `pthread_create()`, `pthread_join()`, `pthread_mutex_lock()`, etc., are used for thread creation, synchronization, and management.

Pthread (POSIX threads) and `fork()` both provide parallelism but differ in their behavior. `fork()` creates a new process with its own memory space, providing isolation but at a higher resource cost. In contrast, pthread creates threads within the same process, sharing memory between threads, which is more efficient but requires careful synchronization to avoid race conditions. Threads are useful for shared memory tasks, while `fork()` is ideal for isolated, independent tasks.

In this mini-lab, you will identify and rectify a typical race condition scenario using condition variables or semaphores.

1. Complete multithread addition (TODO1 & TODO2) in `lab4-multithread.c`, compile and run the file.

**Expected behavior:** The counter variable is updated in the `count_up` function. Ideally, the counter value should end up with `4e6`, as each of the 4 threads increases the counter by `1e6` times.

Note: For compilation of a C program that includes the `pthread.h` header, use the option `-pthread` after `gcc`. e.g., `gcc lab4-multithread.c -pthread -o lab4-multithread`.

**Race Condition:** A race condition occurs when multiple threads attempt to add (`counter++`) to the same variable in shared memory. As a result, an unexpected smaller value will be observed. This indicates an undefined order of execution due to the lower-level implementation of the adding operator, which involves at least three steps: 1)register set, 2)register add, and 3)register read.

2. Address this race condition by completing TODO3 using either conditional variable or semaphore, and re-executing the program.
3. If implemented correctly, the final output should match the expected value.

## Submission

(1 pt) Complete all the TODO sections and submit your code as `lab4-pthread_<your_student_id>.c`.

## Appendix

```
// file: lab4-pthread.c
#include <stdio.h>
#include <pthread.h>

#define NUM_THREADS 4
#define NUM_ITERATIONS 1000000

int counter = 0;

// TODO3: define global variables (~1 line)

void *count_up(void *arg) {
    for (int i = 0; i < NUM_ITERATIONS; i++) {
        // TODO3: Protect the counter increment operation to prevent race
        // conditions
        counter++;
    }
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t threads[NUM_THREADS];

    // TODO3: init condition variables/semaphore (~1 line)

    for (int i = 0; i < NUM_THREADS; i++) {
        // TODO1: Create multiple threads to execute count_up (~1 line)
    }
}
```

```
for (int i = 0; i < NUM_THREADS; i++) {
    // TODO2: wait all the threads to finish running (~1 line)
}

// TODO3: free condition variables/semaphore (~1 line)

printf("Final counter value: %d\n", counter);
printf("Expected counter value: %d\n", NUM_ITERATIONS * NUM_THREADS);

return 0;
}
```

---