# Inter-Process Synchronization and Concurrency

## Summary

In this assignment, you will create three versions of a simple multi-process game. The three versions differ only in the type of synchronization they use: one uses busy-wait, another uses busy-wait with yield, and the third uses blocking semaphores. You will also measure the relative performance of these three versions of your multi-process game.

## Objectives

- Learn the use of shared memory and semaphores for inter-process communication.
- Measure the effectiveness of process scheduling with synchronization.

## Part A: Multi-Process Addition Game Without Locking

In this part, you are asked to write a simple program that takes two command-line arguments **P** and **M**.

The main (parent) process does the following:

- Creates a shared memory region that hold at least three variables -- two numbers and a winner PID -- plus any additional information you want to keep.
- Writes two positive numbers, 1 and 2, into the two numbers in the shared memory region.
- Forks **P** child processes.
- Waits for all the children to complete.
- Prints of the larger of the two numbers and the winner PID from the shared memory.
- Exits.

Each child process repeats the following steps in a loop until the larger of the two numbers in the shared memory is greater than **M** (passed via command-line arguments). Don't introduce any locking yet! Wait to do that in Part B below.

- If any one of the two numbers in the shared memory is greater than **M**, then exit.
- Else replace the smaller number in the shared memory with the sum of the two numbers.
- If the result of addition is greater than **M**, then write this child's process ID into a "winner" variable in the shared memory.

Run this program several times for identical values of P and M. Choose large enough M and a reasonably large P. Observe the parent's output.

## Part B: Addition Game with Locking

Now introduce locking/unlocking around the critical section in each child process. At the beginning of each iteration, acquire the lock. At the end each iteration, release the lock. Something as follows:

```
while(...) {
        lock
        critical section...
        unlock
}
```

You have to write three versions of the locking function.

1. One in which the child **busy-waits** using sem_try_wait(). Something as follows:

```
while ( sem_try_wait(...) < 0 ) {
        if( errno == EAGAIN)
                continue;
        else perror("sem_try_wait failed:");
}
```

2. Another in which you replace the "continue;" statement above with "sched_yield();"
3. And third in which you use a blocking semaphore, using sem_wait(), instead of sem_try_wait(). Change the logic of the abover

code as needed.

## Part C: Comparing the three versions of the Addition game

In this part, you are asked to write user-level profiling code in order to compare the relative performance of the three versions of locking mechanisms you wrote in Part B. Using the gettimeofday() system call, measure and plot the following two graphs.

- Total execution time for the parent process versus number of processes P.
- The average time to acquire the lock in each child process versus the number of processes P

Explain the results you obtain in a PDF report that you must include with your submission.

## Grading Guidelines

```
Part A: Multi-Process Addition without locking - 20
        Creation and execution of processes
        Creation and use of shared memory data region

Part B: Multi-process Addition with Locking - 50
        Proper semaphore initialization.
        Correct use of locking/unlocking so that only one process is in the critical section at a time
        Busy-wait locking with "sem_trywait()" and "continue"
        Busy-wait locking with "sem_trywait()" and "sched_yield()"
        Busy-wait locking with "sem_wait()"

Part C: Comparing the Game of Turns - 25
        Correct implementation and collection of profiling statistics
        Final PDF report with profiling results, graphs and explanations

Error Handling, README, Makefile and Coding style - 5
        Cleanup of shared memory region, checking for return values/error conditions, etc.

Total = 100
```

## Submission Guidelines

Submit all your code and your PDF report as ONE zipped file. Name your directory as "your_username/" and the name the zip file as "your_username.zip". Replace "your_username" with your BU login username (don't use your B number).