CMP3001 Operating Systems
Project (Synchronization)

Due: Monday, December 24th (23:59)

BARRIER IMPLEMENTATION

Barriers are a way by which multiple processes/threads wait a specific location in the code until other threads in the system have reached the same location. This primitive allows a programmer to implement a single synchronization point that can be used to ensure that an application executes in lock step behavior. These operations are extremely popular in parallel distributed systems and a predominant feature in many parallel applications. While they are most often seen in distributed memory systems, they have their use in shared memory environments as well.

As an example of why you would use a barrier consider a scientific application that simulates a physical phenomena over time. The simulation is broken into a series of timesteps that each represent a given quanta of simulated time. The simulation code that executes is exactly the same for each timestep, and modifies a shared buffer that represents the current state of the simulated system at a given point in time. Now imagine what would happen if one core got out of sync with the rest and began simulating a different timestep than the rest of the cores. Because the simulation is using a single shared buffer to store the state data, the simulation would no longer be valid as the buffer no longer represented the correct state at a given point in time. To prevent this from happening, such an application needs to ensure that every core is operating on the same timestep. If one core finishes before the others it needs to wait for the others to finish instead of moving ahead to the next time step before the other cores. Barriers provide a way to ensure this behavior.

In this project you will need to implement a simple barrier primitive. The barrier will be initialized with a certain count which will indicate the number of threads that the barrier will need to wait for.

A thread will signal its arrival at the barrier by calling barrier(), which will be responsible for ensuring the thread does not continue execution until the appropriate number of threads (as specified by the init function) have arrived at the same location. Once all the threads have reached the barrier, they will be allowed to continue.

You are expected to use Semaphore and/or Mutex provided by the library that you have used. (e.g. java.util.concurrent.Semaphore in Java)

Functions to implement:

barrier_init
barrier

Sample pseudo code for barrier usage :

P0	P1	P2

{
A[0] = calc1(0);
barrier();
B[0] = calc2(0, A);
}
	{
	A[1] = calc1(1);
	barrier();
	B[1] = calc2(1, A);
	}
		{
		A[2] = calc1(2);
		barrier();
		B[2] = calc2(2, A);
		}

main(){

barrier_init(3)

create P0;
create P1;
create P2;

.

.

.

print array B

}

Submission:

You must still submit your code over **itslearning** before the deadline. No submission will be accepted after deadline.

Grading:

Your grade will be based on manual inspection of the code and a written answer to the question given during Final exam.

Project Grade = Code %60 + Project Question %40