CIS 520 - Operating Systems I – Homework #2

Due: Wednesday, Oct. 2nd, by 11:59 pm, upload via K-State OnLine

1. When programming scientific code with multiple threads, sometimes it is useful to have all threads rendezvous at a place in the code. This is normally done with an operation called a "barrier". It works so that when threads call the **barrier()** function, none return until all threads have called the barrier function. For example you might find code that looks like:

```
{
    threadsDoSomethingInParallel();
    // Threads may reach the barrier at different times

    barrier(pthread_self());  // call barrier fct with thread id

    // Threads should start here together after the barrier
    threadsDoSomethingElseInParallel();
}
```

   Your job is to write the **`barrier()`** function using only semaphores. You are not permitted to use shared variables other than the semaphores. Be sure to show your initial values for the semaphores. You can assume there is a global constant NUM_THREADS that indicates the number of threads in the system participating in the barrier. Thread ids returned by calling pthread_self are integers starting at 0 and going to NUM_THREADS-1. You may assume that NUM_THREADS is less than some fixed constant bound such as 100. To simplify things you can assume that **`barrier()`** is only called once by each thread participating in the barrier.

   Semaphore declaration and initialization:

```
void barrier(int tid) {




}
```

2. Show that if the **`wait`**`()` and **`signal`**`()` semaphore operations are not executed atomically, then mutual exclusion may be violated.

3. Design an algorithm for a monitor that implements an *alarm clock* that enables a calling process to delay itself for a specified number of time units (*ticks*). You may assume the existence of a real hardware clock that invokes a function **tick()** in your monitor at regular intervals.

4. Write the semaphore pseudo-code to synchronize four processes P1, P2, P3, and P4, so that P1 completes its work before P2 and P3 begin execution of their sections of work code, and P2 completes before P4 begins its work.

   Hint: The following pseudo-code could be used to implement mutually exclusive access to a critical section of code:

   Semaphore mutex = 1; // set semaphore count to 1

Process P1	Process P2	Process P3	Process P4
wait(mutex)	wait(mutex)	wait(mutex)	wait(mutex)
critical section	critical section	critical section	critical section
signal(mutex)	signal(mutex)	signal(mutex)	signal(mutex)