Jeff Morin

CISC 3320

Fall 2017

HW3

**1)** Author: Jeff Morin. I was the sole author of this assignment. My role was researching the multiple synchronization methods that the Java API provided, to modify the code I wrote in homework 1 to avoid race conditions happening in the ProcessIDManager class.

**2)** Our task was to modify the PID manager we created in HW1 with synchronization mechanisms we learned about in chapter 5, to make our API safe against race conditions.

**3)** It was already assumed that the data structure responsible for holding the PIDs and their statuses back in HW1, had to be thread safe. So I used a ConcurrentHashMap of Integer keys paired with Boolean values to hold the PIDs and their statuses. The ConcurrentHashMap provided by the Java API, is optimized for concurrent operations by means of circumventing the need to explicitly synchronize access; as required by a normal HashMap. It is also more efficient; as explicit synchronization causes considerable overhead. The PID integers remained constant during runtime, unlike their boolean value pairs, so an AtomicBoolean data type was used as the value pair.

Making the boolean value atomic, guaranteed that values would be safe from race conditions, which arise from simultaneous access by multiple threads. This further fortified thread safety of the PIDManager. However, these measures didn’t protect the map from being accessed when there were no PIDs available for allocation! Meaning: if there were more threads than the number of PIDs available by the PIDManager, then excess threads accessing the *allocatePID()* function, would repeatedly be assigned an erroneous value of -1, until a PID becomes available. This bug bit hard, and provided just the itch to scratch.

So, reading up on threads in chapter 4 and synchronization mechanisms in chapter 5, gave me the idea to mutually exclude access from the start of the *allocatePID()* function, to the end of the *releasePID(int)* function blocks, using a counting semaphore. A Semaphore object instance was created with the same number of permits as there were PIDs initially available (the size of the hash-map). The *acquire(int)* method of the semaphore was used, to allowed at most 1 permit to be decremented per thread accessing the *allocatePID()*  function. The number of permits becomes 0 when no PID is available for allocation. A thread releasing a PID would then invoke the *release()* method of the semaphore upon exiting a particular conditional block of the *releasePID(int)* function; in turn, incrementing a permit for the next waiting thread to acquire upon entering *allocatePID()*. I also assumed that the number of threads had to be significantly larger than the initial amount of PIDs, in order to ensure this worked. So, 3000 threads were created instead of 100 as specified in the previous homework, to compete for a limited 1000 process IDs allocated in the PIDManager class.