**16.35 Assignment 3 Team Report**

by Nicholas Bain, Mohammed Nasir, Codrin Oneci

1. Waiting on a condition variable is better than using a busy-wait loop because synchronization is still assured while consuming less processing time. It is empirically known that when there is a single condition to meet that is relevant for multiple threads that need to continue, it is better to use synchronization variables instead of busy loops. Usually busy-wait loops are more often used when there are some shared resources in the system between threads. However, in our case threads don’t really share resources so it is more efficient to use a synchronization condition variable.
2. The functions *control\_vehicle* and *update\_state* are specific to each vehicle. In our code, each vehicle has its own thread so we know that we are not sharing any vehicle specific resources in those functions between threads. This explains why thread protection is not required for the two aforementioned functions. Moreover, synchronization is handled in the thread itself by using a synchronization condition variable, so no modifications inside *control\_vehicle* and *update\_state* are required.
3. The *vehicle\_updated* condition variable checking associated sequence of instructions is the only code block where only one process executes at a time. Thus that block defines the critical region of our program.
4. Obviously we have to use a mutex on the *vehicle\_updated* variable because of concurrency problems that may appear otherwise. Without protection, multiple threads may try to increment the value of this variable through a process that is not atomic and thus their individual atomic read/write instructions may overlap in time. With a mutex, in the critical section we allow only one thread to execute an increment instruction (which is composed of its read, add, register shift, write etc. atomic instructions), thus not permitting incrementation counting mistakes. Let’s present a very simple concrete example of incrementation counting error that can appear when a mutex is not used. Threads 1 and 2 run in parallel and they simultaneously start the increment instructions. In thread 1, *vehicle\_updated* = 3 is read, afterwise in thread 2 it is read. For some hardware reasons (maybe due to physical processing map distances, or variable volatile memory access times) thread 2 is accessing the variable after thread 1, but because thread 2 is executing faster the +1 incrementation instruction and its write command (resulting in *vehicle\_updated* = 4), thread 1 will actually overwrite the value *vehicle\_updated* = 4 on a state that is already *vehicle\_updated* = 4! But it is evident that the desired program behavior is that of obtaining *vehicle\_updated* = 5 after these two threads increment. This problem wouldn’t appear however if we would use a mutex.
5. For the aforementioned reasons a mutex is appropriate for incrementing the *vehicle\_updated* variable. A semaphore is a signaling mechanism based on an integer variable, while a mutex is a resource access locking mechanism object. In our case we want to lock the resource, not signal some value change, and that is why we use a mutex. Moreover the mutex is an OS object that allows any thread to acquire the condition variable for the incrementation, but not simultaneously, as opposed to semaphores, so it is well suited for our application.
6. This is implementation specific, so we answer it later.