103011227 周延儒 CS18

Problem set

1. We need to show that:

    1. Mutual exclusion is preserved,
    2. The progress requirement is satisfied,
    3. The bounded-waiting requirement is met.

    First of all, to prove property, we note that is ensured through the use of the flag and turn variables. If both processes set their flag to true, only one will succeed, namely, the process whose turn it is. The waiting process can only enter its critical section when the other process updates the value of turn. Thus, it satisfies the Mutual exclusion property.

    To prove property 2 & 3, we note that a process Pi can be prevented from entering the critical section only if it is stuck in the while loop with the condition flag[j]=true. If Pj is not ready to enter the critical section, then flag[j]=false, and Pi can enter its critical section. If Pj has set flag[j]=true and is also executing in its while statement, then either turn=i or turn=j. If turn=i, then Pj will set flag[j] to false and wait until turn=j, at this time Pi can enter its critical section. After Pi leave its critical section, it will set turn to j, flag[i] to false, so that Pj can now enter its critical section. If turn=j, then Pi will set flag[i] to false and wait until turn=i, at this time Pj can enter its critical section. After Pj leave its critical section, it will set turn to i, flag[j] to false, so that Pi can now enter its critical section. Thus Pi will enter the critical section (progress) after at most one entry by Pj (bounded-waiting).

2. This is because if a user-level program is given the ability to disable interrupts, it can disable the timer interrupt and prevent context switching from happening, thereby it can monopolize the processor so the other process may never execute so there will be starvation
3. The signal( ) operations associated with monitors is **not persistent.**
    If a signal is performed and if there are no waiting threads, the signal will just simply ignored and the system does not remember that the signal took place. If a

subsequent wait operation is performed, then the corresponding thread simply blocks.

On the contrary in semaphores, every signal results in a corresponding increment of the semaphore value even if there are no waiting threads. A future wait operation would immediately succeed because of the earlier increment.

2.4 [5 points] Show your typescript. Run your code multiple times. Does it show the same or different output?   Why

Well, the outputs are quite different.

In the beginning, we set the Semaphore for the internal counter as the 5(the size of the parking lot). Then, the first 5 cars try to park and call acquire(), where the counter is non-zero and thus there is no block. As a result, the parking sequences are the same for the test cases

However, once the counter is decremented to zero, it blocks, so the rest of the cars must for wait any thread(car) call release(), which releases the semaphore. Once any car calls release(), the rest of the cars will compete to proceed, and the proceeding car is different from each case . Consequently, the output sequence will vary from case to case.