Burrell, David 300209541
NWEN303 Assignment Two

Task1:

Part a: The Critical Section problem refers to a situation in which a part of your application contains a set of objects/values that represent a self-contained entity. The problem happens when this entity is needed to be accessed by multiple workers at any one time, meaning that each worker relies on the contents of the section, but each worker can also change those contents, which would mess up the work of other workers.

This is relevant in concurrent programming because the application of parallelism into a program would have problems arise when the use of one self-contained entity by multiple workers or separate threads. Without proper management of these sections, unintended/unforeseen behaviour will most likely occur and become a major pain to deal with.

I was made aware of this problem in a separate course, Operating Systems design or some such, and so I can't entirely remember how this influenced me down the line other than being aware of it. In some ways it is an extension of aliasing problems that can occur with mismanagement of objects, and again I have been far more aware and able to notice, and deal with problems like these than I would have without knowing about them. It seems like a logical problem that occurs, and after having it explained it simple makes sense.

Part b:

Part c: Critical Section Attempt 1:

> **Mutual Exclusion:** This property holds because each worker is assigned its own time with the critical section only when it is that workers turn. Because the only way the turn is changed from a worker to another worker is in the post-protocol of the current worker.
>
> **No Stuck:** Yes, this holds, as either worker will eventually have access to the critical section.
>
> **No Individual Holds:** No, if one of the workers loops forever during normal operations, then the waiting worker will have to wait forever before control being passed to it

Critical Section Attempt 2:

> **Mutual Exclusion:** With this a problem arises when both workers arrive at each step at the same time, in which case they both say they want access at the same time and both then enter at the same time. In another case, i.e. worker one is already set to want, then its entirely possible that the worker will complete its work and set to false, before ending up in the first situation again.
>
> **No Stuck:** Considering both workers can be in the critical section at the same time, then yes this holds.
>
> **No Individual Holds:** This solution of implementing desire does manage to fix the problem of attempt one, as such no worker is now restricted by the fact another worker wants access. The positioning of the want=true being after the normal operations means that if a worker loops forever in normal operations then it won't have set want to true yet.

Critical Section Attempt 3:

> **Mutual Exclusion:** Can hold if it doesn't get stuck.

No Stuck: There is a case in which both workers are trying to get access to the critical section at the same time, and in this case, neither would be allowed in.

No Individual Holds: In the case they are stuck, each worker is starving the other.

Critical Section Attempt 4:

Mutual Exclusion: Yes, this makes sure that each worker only enters the critical section in the case that the other worker does not want access.

No Stuck: By toggling the want Boolean within the pre-protocol for each worker, eventually the workers will not be in sync and one will allow the other access

No Individual Holds: If the scheduler keeps both workers in in perfect sync, then it is possible that both workers will starve each other.

Critical Section Attempt 5:

Mutual Exclusion: The only case in which a worker will enter the critical section is in which it is that workers turn and it wants access. If that worker does get access due to wanting it and the other worker not wanting it, then it will check if it is that workers turn, and if it isn't it will surrender desire and wait until its turn arrives.

No Stuck: Due to having 2 separate protocols for allowing access to the critical section and toggling between these occurs in such a way that the other will either surrender the desire to enter or switch the turn to the current worker then at no point will the workers be stuck.

No Individual Holds: Each worker has multiple checks in which it has to pass in order to get access. In the cases these checks fail, it will surrender access to the other, and in doing so neither worker will starve.

Task Two:

Mutual Exclusion: The simple use of checking if the want integers are equal or opposite when allowing either worker into the critical section allows mutual exclusivity. This solution however only really works for 2 workers.

No Stuck: The else statement in both workers prevents the workers getting stuck in the original if check, provided the scheduler syncs them in a way that the if check could be repeated indefinitely.

No Individual Holds: the finally statement switching the integer back to 0 in both cases should prevent either worker from entering a looping process in which it always wants access and could be scheduled in a way that prevents the other worker from ever getting access back.

Task Three:

Mutual Exclusion: As this set up has almost the exact same functionality as Critical Section Attempt 5, the mutual exclusion still holds for the same reasons.

No Stuck:

No Individual Holds: Unfortunately the one change made to Critical Section Attempt 5 does add a factor in which one worker can starve the other provided the other wants it, and the turn is opposite.

Burrell, David 300209541
NWEN303 Assignment Two