Homework 4

By Jarred Parr

1. We can see step by step that this program first checks if token is true and runs the step 2 operation of making it false, if it is already false, it goes into the critical section and operates until the process is done. The problem, and why it violates the bounded waiting requirement, is how the while statement is happened in step 4. Since the bounded wait is improperly implemented, there is a chance for the process to wait infintely, which is not allowed to happen in a bounded wait scenario.
2. A wait operation would need to be placed on steps 2 and 6 to guard the assignment of the token value to true or false. If the wait operation is not adaquately guarded via making it atomic, then there is the strong likelyhood of a race condition occuring. This could cause undefined behavior across threads as they read and write from the values in a free-for-all manner. Because of this, we could see issues like deadlock occur in the program as it waits infinitely.
3.
    1. This program does not suffer from lockstep synchronization because, as the book describes, Dekker's algorithm shows that the processes are not forced to alternate and hand off control to one another. The threads are able to flip the flags to enter the critical section without needing to explicitly want to enter the critical section. We can see this in the following:
        - `flag = [0, 0]`
        - If a thread does wants into the critical section, it sets its respective flag to 1. Any code with access to the critical section with a flag of 0 will simply just pass though and the flag marked 1 will now be able to enter the critical section
        - If the thread marked 0 still does not want to enter the critical section, but the other thread does, then it can simply repeat this process. Dekker's algorithm does not require us to pass off entrance if no other thread needs it.
    2. This program also does not suffer from deadlock due to the fact that, since it hands off after the critical section runs, it gets rid of the final step of deadlock which is circular wait. If only two processes are looking to get into the critical section, then as soon as one finishes, the other one can immediately have access because of how the hand off is done via the turn variable. This only happens in the case of both threads setting their flags to 1, i.e `flag = [1, 1]`.
4.
    1. Mutual Exclusion - This is seen in the dining philosphers problem because they all have access to the same pool of resourcs, the chopsticks, so when they all need two, it leads to some being left without the resources.
    2. Hold and Wait - This holds because each philospher has only one chopstick and when they have one they must wait for resources, but they all have only one, so they all hold and wait.
    3. No Preemption - The philosopher must wait for philosopher to put it down, they cannot just ask for it, but if none of them can ask to relinquish resources, then they cannot allow one another to finish.

4. Circular Wait - Each philosopher has access to only one chopstick at a time, this is how this condition holds because they each need two chopsticks in order to work properly. This leads to them circularly waiting for the last one ti finish.

Practical

The code strucutre for the semaphore is as follows:

```
struct semaphore {
    raw_spinlock_t      lock;
    unsigned int        count;
    struct list_head    wait_list;
};

#define __SEMAPHORE_INITIALIZER(name, n)                \
{                                       \
    .lock       = __RAW_SPIN_LOCK_UNLOCKED((name).lock),    \
    .count      = n,                            \
    .wait_list  = LIST_HEAD_INIT((name).wait_list),     \
}

#define DEFINE_SEMAPHORE(name)  \
    struct semaphore name = __SEMAPHORE_INITIALIZER(name, 1)

static inline void sema_init(struct semaphore *sem, int val)
{
    static struct lock_class_key __key;
    *sem = (struct semaphore) __SEMAPHORE_INITIALIZER(*sem, val);
    lockdep_init_map(&sem->lock.dep_map, "semaphore->lock", &__key, 0);
}
```