Assignment 4.1 Memory Consistency

1. Given an execution path for each thread, what property does the hardware (or the model) have if only a single interleaving is possible?
   - [ ] strict consistency
   - [ ] sequential consistency
   - [ ] weak consistency

2. What consistency guarantee does a system with a MESI cache but without store or invalidate buffers give?
   - [ ] strict consistency
   - [ ] sequential consistency
   - [ ] weak consistency

3. A program reaching a state $S$ on weakly consistent hardware can always reach the same state $S$ on sequentially consistent hardware.
   - [ ] yes
   - [x] no

Assignment 4.2 Semaphores, Locks, and Monitors

Tick one of the answers in each question.

1. A semaphore can be used to implement a mutex.
   - [ ] true
   - [ ] false
1. A semaphore can be used to implement a mutex.
2. A mutex is always re-entrant.
3. A monitor can be used as a mutex.
4. Any deadlock-free program must acquire locks in a fixed order.
5. When acquiring locks in a fixed order to ensure deadlock-freedom, there is no advantage in releasing them in the opposite order.
6. The use of which concurrency construct may lead to starvation, that is, a thread that never manages to execute the critical section to completion, given arbitrary many chances?
   - A wait-free algorithm
   - A lock-free algorithm
   - A lock where blocking threads are put into a queue
   - A signal-and-urgent-wait monitor where all waiting threads are tracked in queues

8. Suppose that a program was shown to be deadlock-free using the lock order argument. This approach to dealing with deadlocks is called
   - Deadlock detection
   - Deadlock prevention
   - Deadlock avoidance
   - Ignoring deadlock

9. Consider the program P whose sole synchronization between its two threads is given by the following two program fragments. According to the definition of deadlocks,
   - $P$ may deadlock. There exists a lock order between the locks.
   - $P$ may deadlock. There exists no lock order between the locks.
   - $P$ cannot deadlock. There exists a lock order between the locks.
   - $P$ cannot deadlock. There exists no lock order between the locks.

10. By recording an interleaving of a program at runtime, we observe the following: A thread that holds a lock is descheduled and another thread is scheduled that then executes holding the same lock.
P may deadlock. There exists a lock order between the locks.

P may deadlock. There exists no lock order between the locks.

P cannot deadlock. There exists a lock order between the locks.

P cannot deadlock. There exists no lock order between the locks.

10. By recording an interleaving of a program at runtime, we observe the following: A thread that holds a lock is descheduled and another thread is scheduled that then executes holding the same lock.

□ This behavior should never happen since it violates the mutual exclusion property, so there must be an error in the program.

□ The lock must be a signal-and-urgent-wait monitor.

□ The lock must be a signal-and-continue monitor.

11. The `enter` operation of a monitor is called `notify` in Java.

1. Additionally, we are given a main function that runs `f` and `g` in parallel:

```java
main() {
    f();
    g();
}
```

Can this possibly cause a deadlock? If not, try to prove it using the freedom of deadlock theorem.

2. Assuming there is no possible deadlock, how can we change the main function in a simple way to render a deadlock possible?

Finally, we change the main function so that it runs `f` and `g` sequentially:

```java
main() {
    f();
    g();
}
```
Can this possibly cause a deadlock? If not, try to prove it using the freedom of deadlock theorem.

Can this possibly cause a deadlock? If not, try to prove it using the freedom of deadlock theorem.
1. Additionally, we are given a main function that runs `f` and `g` in parallel:

```c
36 main() {
37 f();
38 g();
39 }
```

Can this possibly cause a deadlock? If not, try to prove it using the freedom of deadlock theorem.

2. Assuming there is no possible deadlock, how can we change the main function in a simple way to render a deadlock possible?

3. Finally, we change the main function so that it runs `f` and `g` sequentially:

```c
36 main() {
37 ...
38 ...
39 }
```

Obviously, no deadlock can occur (no parallelism and no lock is acquired multiple times without releasing it in between). Again try to prove this using the freedom of deadlock theorem.