**Exercise 1**

1. A guarded method suspends an incoming thread when an object is not in a suitable state to process a request, and thread waits for the state to change for a proper state. It’s better to use guarded methods when clients can tolerate indefinite postponement and you have to guarantee that the required states are eventually reached. Also, conditions and actions have to be managed within a single object.
2. It’s important that object have to be in a consistent state before entering calling wait() because calling wait releases the synchronization lock.
3. Put this guarded method call in a try-catch statement in the caller object. Create an errorTryLimit variable(ex. 3) and an error counter in the caller object. If guarded method call gets an exception then should go in to the catch statement and this catch should increase errorCounter by 1 and checks if the errorTryLimit is reached or not. If errorTryLimit is reached then, user should be notified about this situation because collaborater thread may be terminated. If errorTryLimit is not reached than caller can cleanup and retry guarded method call again. By this, if this was a some kind of a momentary error situation then caller can try to fix this and it can be resolved without notifying user. If error is recurring than user should be notified.
4. If there are 4 sufficent conditions can be observed on processes than deadlock happens. These conditions are:
   1. Processes share resources under mutual exclusion.
   2. Processes hold on to acquired resources while waiting to obtain others
   3. Resources cannot be pre-empted once acquired by process, but only released voluntarily.
   4. A cycle of processes exists in which each process holds a resource which its successor in the cycle is waiting to acquire.

You can avoid deadlock by ensuring that these conditions cannot arise. If any of these conditions are removed, then deadlock cannot take place.

1. It’s the safest way to have a design which doesn’t rely on deadlock detection techniques. If you have to rely on these techniques then that means you have a complex design which is needed to be mainted carefully. Also, these designs add time complexity.
2. Progress is a liveness issue because liveness property and progress property are so similar to each other. Also, progress is the opposite of starvation and starvation is a liveness issue.
3. If notify() is used carelessly than it can lead to thread race conditions to happen.
4. In a nested monitor lockout and in a classical deadlock both processes are end up waiting for each other forever. Also, classical deadlock and nested monitor deadlock need at least two participants to occur. In classical deadlock, each process holds some resources waited by another participant. However, in nested monitor lockout, one thread holds the lock to resource and other thread waits to get the lock on the same resource.

**Exercise 2**

Four different solutions to Dining Philosophers problem:

* Deadlock detection: Repeatedly check for waits-for cycles. When detected, choose a victim and force it to release its resources. This solution is not fair because you’re terminating a thread and forcing it to restart.
* Deadlock avoidance: Design the system so that a waits-for cycle cannot possibly arise. Remove one fork from the table. This will break the cycle, but one thread won’t able to progress which will cause starvation.
* Number the forks and let philosophers to grab the lowest numbered fork first. By this one philosopher will grab forks in the reverse order. Also, for this solution, philosopher need to wait to pick up the other fork, but eventually it will happen, so we can say it’s fair to a degree.
* Philosophers queue to sit down. Allow at most four philosophers to sit down simultaneously. So one phisolopher will need to wait, but eventually he will sit down and eat, so we can say it’s fair to a degree.

**Exercise 3**

LOCK = ( acquire -> release -> LOCK ).

PHIL = ( sitdown -> acquire

-> right.get -> left.get -> release

-> eat -> left.put -> right.put

-> arise -> PHIL ).

FORK = ( get -> put -> FORK ).

||DINERS(N=5)= forall [i:0..N-1]

( phil[i]:PHIL || {phil[v:0..N-1]}::LOCK || {phil[i].left,phil[((i-1)+N)%N].right}::FORK).

A waiter (lock) organizes philosophers until the philosopher picked up both of their forks.

**Exercise 4**

START = (start -> N7),

N7 = (east -> N8),

N8 = (west -> N7 | north -> N5),

N5 = (south -> N8 | north -> N2),

N2 = (south -> N5 | west -> N1),

N1 = (east -> N2 | south -> N4 | west -> N0),

N4 = (north -> N1 | west -> N3),

N3 = (east -> N4 | south -> N6),

N6 = (north -> N3),

N0 = (east -> N1 | north -> STOP).

You can change the starting node by modifying the start action. I used N7 as my starting point. For this node I get the correct following result from LTS tool:

DEFAULT = START

State Space:

11 = 2 \*\* 4

Analysing...

Depth 8 -- States: 9 Transitions: 15 Memory used: 7585K

Trace to DEADLOCK:

start

east

north

north

west

west

north

Analysed in: 0ms