Task 01

Consider the following program written in the "AWAIT" language.

```
int x = 0;
int y = 0;
int z = 0;

sem lock1 = 1;
sem lock2 = 1;

process foo {
   z = z + 2;
   P(lock1);
   x = x + 2;
   P(lock2);
   V(lock1);
   y = y + 2;
   V(lock2);
}

process bar {
   P(lock2);
   y = y + 1;
   P(lock1);
   x = x + 1;
   V(lock1);
   V(lock2);
   z = z + 1;
}
```

1. Can this program deadlock? If yes, then how? If no, then why not?

   Yes! This program <u>might</u> deadlock if the scenario where "foo" waits for lock2 to release from "bar", but at the same time "bar" is waiting for "foo" to release lock1. This can happen if the two processes complete the first P-operation at the same time. This creates a deadlock. But it is not guaranteed to happen every time.

2. What are the possible final values of x, y, and z in the deadlock state?

   x == 2, y == 1, z == 2.

3. What are the possible final values of x, y, and z if the program terminates?

x == 3, y == 3, z == 1/2/3

Task 02

Explain the code at this link: *https://plnkr.co/edit/Q1jyGXvy9INMRG3Y?preview*

1. What does then do in function go and how does then work?
   The ".then" in go() is a result of the promise in the showCircle() function. It is called when the promise "resolves". It executes some functions to the div that the user sees. In this case, it adds the ".message-ball" defined in the start of the file and appends the text "Hello World" to the div. (The div is defined in showCircle() as an element the user sees. We're only appending the text to it when the promise resolves.

2. Why do we have return new Promise(...) in function showCircle?
   Because we need to know when the circle has reached the inputted size (function args) (xcoords, ycoords, radius). So, the function showCircle() will return a promise, this promise will resolve when the promise is fulfilled.

3. What does return new Promise(...) do?
   This promise will set the width and height to the specified size. In this case, it is specified 100 pxs radius, which is the same as 200px width, and 200px height (diameter = radius*2) Once this is done, the promise resolves and go() continues executing.

4. Why do we have setTimeout in line 46?
   If we don't the circle will not gradually increase in size, and the text will never be added. setTimeout() works by wrapping code in between it's brackets then executing it after a given timeout (given on line 54)

5. What is the meaning of lines 50–53?
   This is the way the promise resolves upon completion. We have an event listener for the div that is waiting for the transition to end. (The transition meaning the circle appearing larger and larger in size until it has reached the correct size). Once that is done, it removes the event listener and resolves the promise (yay!)

6. What is the meaning of 0 in line 54, and what consequences does it bring?
   Line 54 has the closing brackets of setTimeout(). It waits 0 milliseconds before executing the code on line 47-53. If we change the "0" on line 54 to for example "5000" it will wait 5 seconds before creating the circle/starting the transition.

Task 03

Mark which of these are true and which are false. Each correct answer gives 1 point. Points are not deducted for wrong answers.

"In monitors, condition synchronization is provided implicitly." **TRUE**
Consider the following code block: int x = 0; int y = 0; co x = y + 1; || y = x + 1; oc. Does the first assignment (x = y + 1) satisfy the At-Most-Once property? **FALSE**
"In Signal-and-Continue signaling discipline for monitors, the process executing signal passes control of the monitor lock to the process awakened by the signal." **TRUE**
"In the Await Language, operation signal(cv) applied to condition variable cv has no effect if the delay queue of cv is empty." **TRUE – it tries to wake up a process, if empty then no effect**
"In Signal-and-Wait signaling discipline for monitors, the signaler continues and the signaled process executes at some later time." **FALSE – it executes immediately**
"In asynchronous message passing, receive is a blocking primitive." **TRUE – it is halted until a message is received**
"A state change caused by an atomic action is indivisible and hence cannot be affected by atomic actions executed at about the same time." **FALSE**
"When emulating monitor-based programs with message-based programs, wait statement corresponds to saving a pending request." **TRUE**
"A parallel execution can be modelled as a linear history, because the effect of executing a set of atomic actions in parallel is equivalent to executing them in some serial order." **FALSE**
"In the Await Language, operation empty(cv) applied to condition variable cv clears the queue of cv." **FALSE – it asks if the queue is empty, it does not empty it**
"Mutual exclusion is an example of a liveness property." **FALSE – it is a safety property**
"Termination is an example of a safety property." **FALSE – it is a liveness property**

"A liveness property is one in which the program always enters a good state, i.e., a state in which variables have desirable values." **TRUE – liveness property has to do with a good eventual outcome**
"A fine-grained atomic action is one that is implemented directly by the hardware on which a concurrent program executes." **TRUE**
"Java uses Signal-and-Continue signaling discipline for monitors." **TRUE – and only Signal & Continue**
"Eventual entry to a critical section is an example of a safety property." **FALSE – it is a liveness property**
"In the AWAIT language, the await statement can only be used to specify fine-grained atomic actions." **TRUE**
"Partial correctness is an example of a liveness property." **FALSE**
"The state of a concurrent program consists of the values of the program variables at a point in time." **TRUE – at a given point in time the values of each variable represents the state of the program**
"A scheduling policy is unconditionally fair if every unconditional atomic action that is eligible is executed eventually." **TRUE – if every eligible action is executed eventually**