Daniel Sega
dls148
EECS 338

Assignment 3

1)

```
mutex = Semaphore(1)
oxygen = 0
hydrogen = 0
oxygenWaiting = Semaphore(0)
hydrogenWaiting = Semaphore(0)
```

Oxygen

```
mutex.wait()
oxygen = oxygen + 1
if hydrogen >= 2 {
        hydrogenWaiting.signal()
        hydrogenWaiting.signal()
        hydrogen = hydrogen - 2
        oxygenWaiting.signal()
        oxygen = oxygen - 1
        }
else {
        mutex.signal()
        }
oxygenWaiting.wait()
bond()
mutex.signal()
```

Hydrogen

```
mutex.wait()
hydrogen += 1
if hydrogen >= 2 and oxygen >= 1 {
        hydrogenWaiting.signal()
        hydrogenWaiting.signal()
        hydrogen = hydrogen - 2
        oxygenWaiting.signal()
        oxygen = oxygen - 1
        }
else{
        mutex.signal()
        }
hydrogenWaiting.wait()
bond()
```

If Oxygen or Hydrogen reaches it's mutually exclusive state while there is not enough of the other to bond it will wait as the other piles up. It will then attempt to allow the other to create a bond. Process passes back and forth.

2)

```
mutex = Semaphore(1)
oxygen = 0
hydrogen = 0
barrier = Semaphore(3)
oxygenWaiting = Semaphore(0)
hydrogenWaiting = Semaphore(0)
```

Oxygen

```
mutex.wait()
oxygen = oxygen + 1
if hydrogen >= 2 {
        hydrogenWaiting.signal()
        hydrogenWaiting.signal()
        hydrogen = hydrogen - 2
        oxygenWaiting.signal()
        oxygen = oxygen - 1
        }
else {
        mutex.signal()
        }
oxygenWaiting.wait()
bond()
barrier.wait()
mutex.signal()
```

Hydrogen

```
mutex.wait()
hydrogen += 1
if hydrogen >= 2 and oxygen >= 1 {
        hydrogenWaiting.signal()
        hydrogenWaiting.signal()
        hydrogen = hydrogen - 2
        oxygenWaiting.signal()
        oxygen = oxygen - 1
        }
else{
        mutex.signal()
        }
hydrogenWaiting.wait()
bond()
barrier.wait()
```

Solution now has a barrier that the oxygen and hydrogen atoms pass one at a time after bonding. This should prevent multil bonds from being established, ensuring sequential bonding.

3)

```
current_state     = 'neutral'
mutex     = Semaphore(1)
right_count = ‘number of baboons on right bank’
left_count = ‘number of baboons on left bank’
right_turn = Semaphore(1)
left_turn = Semaphore(1)
```

Right Side:

```
Right_turn.wait()
Right_turn.signal()
mutex.wait()
if state == 'neutral' and right_count >= left_count*2 {
        state = 'right overflow'
        left_turn.wait()
        }
mutex.signal()
cross()
right = right_count - 1
mutex.wait(){
        if state == 'right overflow' and right_count <= left_count {
                state = 'neutral'
                left_turn.signal()
        }
mutex.signal()
```

Left Side:

```
left_turn.wait()
left_turn.signal()
mutex.wait()
if state == 'neutral' and left_count >= right_count*2 {
        state = 'left overflow'
        right_turn.wait()
        }
mutex.signal()
cross()
left_count = left_count - 1
mutex.wait(){
        if state == 'left overflow' and left_count <= right_count {
                state = 'neutral'
                right_turn.signal()
        }
mutex.signal()
```

This algorithm is based on the assumption that one side will have more baboons than the other, triggering that one side to send its baboons until it dips below the other side, causing the process to shift to the other side.