1. (+25) There is a one-lane east-west bridge in Hawaii such that when a car is on the bridge going eastbound, no westbound cars are allowed until the eastbound car has left the bridge. Similarly when a westbound car is on the bridge, no eastbound cars are allowed until the westbound car has left the bridge. To make matters more complicated, if an eastbound car arrives and sees another eastbound car already on the bridge, then that eastbound car will also proceed onto the bridge. This is true even if there is a westbound car already waiting to enter the bridge. Similarly, a westbound car can tailgate behind another westbound car already on the bridge even if an eastbound car was waiting. Deign a synchronization solution using only locks, semaphores and integer variables that achieves the following: allows all cars bound in a certain direction to continue crossing as long as there is at least one car still on the bridge that is bound in that direction, then toggles to allow all cars in the opposite direction to proceed in a similar manner. The solution need not be starvation-free.

Psuedo-code:

```
Lock bridge_control;
Int ecars=0, wcars=0;
Semaphore east, west;

Eastbound() {
P(east); //to wait on signal to go
Ecars++;
//Now we check if there's cars on the east side, we'll simulate one coming in
if (ecars==1){
P(bridge_control); //lock bridge
;} //lock the west cars from coming in
p(east)
ecars--;
V(east) //release
V(bridge_control) } //release

Westbound() {
P(west); //to wait on signal to go
wcars++;
//Now we check if there's cars on the west side, we'll simulate one coming in
if (wcars==1){
P(bridge_control); //lock bridge
;} //lock the west cars from coming in
p(west)
ecars--;
V(west) //release
V(bridge_control) } //release
```

2. Based on Slides 5.2:

```
lock lock;
conditions rc;
int fcomplete=0;

//T1 with C1
Acquire(lock);
DO(c1)
Fcomplete++;
Release(lock);
Rc.signal(); //release to second process.

//T2 with Cs
Acquire(lock);
While (fcomplete==0){
        Release(lock); //to guarantee c1 can run
        rc.wait();
        If (fcomplete==1):
                DO (c2)
        Fcomplete--;}

Release(lock);
```

3. The 3rd Readers/Writers problem has a constraint that no thread will be allowed to starve. It implements FIFO ordering when blocking and releasing threads. Current reading threads aren't interrupted when a writing thread starts, but future ones have to wait till it's done. This means we can have multiple reader threads while no writing threads are working. This way both writers and readers aren't starved.

4a. Nope! There's no robust scheduling involved here to protect the values so at any point another thread can enter and change the value of *y (pointer) breaking the logic of the swap function since the actual value would be lost and the swap would be false. This could happen with a context switch.

4b. Yes it's reentrant. There's no locks or semaphores or anything that dictates precise scheduling or locking resources so it would just execute normally. There aren't multiple processes sharing the same exact memory.

5. We need to schedule the order of the processes and have shared memory resources so a message-passing IPC works. We could use a semaphore to coordinate the order as it also allows us to synchronizes process with shared resources. P1 access the file from memory first, does its magic on it then releases the file and wakes up p2. P2 locks the file, compresses it and then releases it and sends an interrupt to P1 which would be waiting for it.

6.
We need to coordinate sharing 3 variables amongst 3 processes. We will setup race conditions to handle each specific resource. Each function will wait for the variable to free up, then perform the operation and release it by signaling.

Pseudo code:

```
#include math.h
monitor motor{
private int v1, v2, v3;
conditions  c1, c2, c3;

public function increment(){
c1.wait();
v1++;
c1.signal();}

public function decrement(){
c1.wait();
v1--;
c1.signal();}

public function square(){
c2.wait();
v2*=v2;
c2.signal();}

public function squareroot(){
c2.wait();
v2=sqrt(v2);
c2.signal();}

public function sin(){
c3.wait();
v2=sin(v3);
c3.signal();}

public function cos(){
c3.wait();
v2=cos(v3);
c3.signal();}
}
```