Q1:

(a)  
The system is not locked because there is at least one process that can run (it has allocated all the required resources).

In the system shown in the graph two cycles (circular waits) can be identified (involving P1 and P4 and P2 and P3). Nevertheless, the resources involved in the cycles have multiple instances of them. Circular wait is a necessary condition for a deadlock, but not sufficient when multiple resources of the same type exist.

(b)  
To know if the system can go to deadlock we have to study the maximum possible usage of their resources and find a sequence of process execution (checking safeness in Banker's algorithm):

<table>
<thead>
<tr>
<th></th>
<th>Allocated</th>
<th>Max</th>
<th>Need</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>R1</td>
<td>R2</td>
<td>R3</td>
</tr>
<tr>
<td>P1</td>
<td>0</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>P2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P3</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>P4</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>P5</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

Available (0, 0, 0)

A possible sequence of execution is <P5, P4, P1, P2, P3>

Work
Need 5 <= (0,0,0)
Need 4 <= (0,0,1)
Need 1 <= (1,1,1)
Need 2 <= (1,1,3)
Need 3 <= (2,2,3)
(3,2,3)

Q2:

We create a solution based on the producer-consumer pattern whereby the sensors are independent producers and the user of the values is the consumer. Here an Ada like syntax is used to create two processes in a parameterized way. You could just as well create two separate processes for producing in a pseudo-code language that requires each process to be explicitly modeled.
type sensor_system = monitor

var pair : array [0..1] of (data, no_data);
var ready of condition;
var sensor : array [0..1] of integer;

procedure entry produce_value(i:0..1);
//this creates two processes that create one value each by reading
begin
  read(sensor[i]);
  pair[i] = data;
  if ((pair[0]==data) and (pair[1]==data)) then
    begin
      ready.signal();
    end
end

procedure entry consume_pair();
// this creates one process that uses pairs of ready values
begin
  if ((pair[0]!=data) or (pair[1]!=data)) then
    begin
      ready.wait();
      use(sensor[0], sensor[1])
    end
  pair[0] = pair[1] = no_data;
end

begin // initialisation
  pair[0] = pair[1] = no_data;
  sensor[0] = sensor[1] = 0;
end

Q3:
Parallelism can occur at different levels, in a single CPU, in computers with multi-core CPUs or
multiple CPUs, and in distributed (networked) systems. In a single CPU what we have is not
parallelism but pseudo-parallelism (or logical parallelism) which is nevertheless a useful concept, since
it allows running some other process on the CPU when the currently running process has to wait for
some input/output or memory accesses, so that CPU capacity is not wasted. Modeling parallelism is
important because it allows the study of aspects like scheduling, resource sharing, mutual access to
resources in all the above settings. Also, it decouples the application logic (the potential parallelism)
from its implementation on one or more CPUs (actual parallelism), and allows reasoning about what is
allowed/not allowed irrespective of which version is implemented (single CPU, multi processor,
networked).

Q4 (a)
We assume that the semaphore mutex is initialized to 1 as it should be if it is intended for implementing mutual exclusion. Then we can see that there exists a trace of the system in which both P1 and P2 can be in the critical section at the same time, thus claim that this code does not implement the intended function. The trace is the following.

P1: signal (mutex) // i.e. mutex is incremented to value 2
P1: critical_section _1
P2: wait (mutex) // value of mutex is tested for being >= 1 and gives true, process is not suspended
P2: critical_section_2
...
Thus, the system does not implement mutual exclusion of critical sections.

Q4 (b)
1. If P1 crashes in non_critical_section_1, what we are sure about is that the value of mutex is >=1.
   Why? This is due to the fact that signal is implemented with an if statement (either waking up another process or adding 1 to mutex and exiting). Case 1: If mutex is incremented in signal then P2 is able to carry out its function even if P1 crashes in its non_critical_function. Case 2: if signal led to switching to P2, then the control must return to P2 where it was suspended earlier. This can only happen at a wait statement. Hence, when P2 resumed it was in its critical_section_2. Then at some point after that the switch to P_1 takes place (which led to the crash in the non_critical_section_1). There are two possible cases: (a) P2 was still in critical_section_2 when this switch took place. Then, it can continue in that location if the control is transferred back to P2, and eventually running signal will increment mutex. From this point we know that mutex >=1 and P2 will not be blocked if the program only iterates around P2’s loop. (b) P2 was in the signal line or the non_critical_section_2 line when P1 crashed. Then when control is transferred to P2 we are in a similar situation as the last part of case (a).
2. If P2 crashes in its critical_section_2, then we know that mutex is >=0 (otherwise P2 would have been suspended not entering this section. Then what happens to the program is the following. No matter which line P1 is in, the value of mutex can only be increased. This is due to the fact that only the wait operation can decrease mutex and there are no waits in P1. Therefore, P1 can successively enter and exit its critical section (will not be blocked) and the value of mutex keeps rising.

Q5:
   Service A fault descriptions indicate an arbitrary (or Byzantine) fault model. This type of fault leads to any type of error, from loss or duplication of messages, to timing, to modification of message content. The validity property does not hold because the messages can be lost. Notice that the property says that “any message in the outgoing message buffer is eventually delivered to the incoming message buffer”. The integrity property does not hold because the message content can be corrupted and duplicated. Notice the property says that “the message received is identical to one sent, and no messages are delivered twice”.

   Service B fault descriptions indicate the presence of omission and timing faults, because the messages can be lost or delayed, but their content is not modified. The validity property does not hold, since the messages can be lost. The integrity property holds, since the messages are neither corrupted
nor duplicated. Service B is not reliable, since it does not satisfy the validity property.

**Q6:**
We assume that the server and the client run in different hosts, and that they may require several hops to reach each other. In this case, the answer of the server cannot be guaranteed to arrive with a bounded time because networks by default do not offer Quality of Service (QoS) primitives. In order to make sure that the server answers within a given time interval, a network QoS solution is required making latency a main parameter of the service quality. The proposal is only feasible if the clocks of the two hosts are synchronized with an acceptable accuracy (thereby possible to relate the rate of processing at one end to the other) and the message delivery latency is shown to be bounded – i.e. the application is assumed to be classified within the synchronous interaction model of distributed systems.

**Q7:**
Read page 22-23 of the course literature on dependability (Avizienis et al.).

**Q8:**
Given a Mean Time To Failure (MTTF) of 5 days (120h) and 4 hours to fix a failure, it is possible to calculate the availability of a single computer as follows:

\[
Avail_{Computer} = \frac{120 \text{ hours } UpTime}{120 \text{h } UpTime + 4h \text{ DownTime}} = 0.9677 = 96.77 \text{ percent}
\]

Hence the probability of having a failure is: \( P_f = 1 - 0.9677 = 0.0323 \) i.e. roughly 3%.

In case we have three computers offering the same service, the probability of having a failure at the same time in all the computers is:

\[ P_{all} = P_f^3 = (0.0323)^3 = 0.000033698 \text{ (i.e. much lower than 3%!) } \]

Hence, the availability of the whole system is: 1 - 0.000033698 = 0.999966 (i.e. over 99.99%)

**Q9:**
Check pages 605-606 of the course literature on distributed systems (Coularis).