Problem 1

A standard solution to the critical section problem involves semaphors which essentially lock access to a portion of the code. This lock being enabled is effectively a binary state. Similarly, the sign of a number is also a binary state and we can use that binary state to implement our locking mechanism.

```
v = 1, s = NEG;

while( s != POS ) {
    DEC(v, s);
}
S;
v = 1;
```

This implementation will only allow entrance into the system when **DEC** returns **POS** for the sign. Initially, this will happen for a single process, which enters into the critical section. The rest of the processes will sit and decrement v in a spin loop. This will prevent any of the other processes from entering until the process in the critical section exits. Upon exiting the critical section, the process will set v to be 1. This will ensure that one process exits the spin loop and enters the critical section.

Problem 2

Part (a)

Ignoring potential deadlock concerns, the statement itself would be executed atomically because only a single process will be able to proceed beyond the P call each time.

Part (b)

In the case that every process falls into the **while** loop, all of the processes will block on $P(d)$, causing deadlock. If a single process can skip over the **while** loop, then deadlock will be avoided because the first process will wake some process up.

Part (c)

This does not guarentee that B is true when S is executed. This guarentees that B has been **true** before S executes. Consider the possibility that after exiting the **while** loop, the process is interrupted and another process changes B. In this case, when the first process resumes, it is possible that B is **false** when S is executed.

Problem 3

Part (a)

The e semaphor restricts access to the nr, nw, dr, and dw counters. The r and w semaphors represent the restrictions placed on when a reader or writer can enter the database. A reader will be allowed to enter their critical section when there are no writers in the database. Conversely, a writer will be able to enter the database when there are no readers or writers in the database. When a writer is waiting to enter a database, it will increment a counter tracking the number of writers that are waiting. Similarly, when a reader is waiting to enter a database, it will increment a counter to keep track of the number of readers waiting.

Part (b)

This is a writers preference solution. In the Writers process, it will prefer to awake a delayed writer rather than a delayed reader.

Part (c)

One of the most immediate differences is that the solution in figure 4.13 is a readers preference.

In the best case, the reader process in figure 4.21 will make 3 P calls and 2 V calls per iteration of the **while** loop. In the worst case, the reader process will make 3 P calls and 3 V calls per iteration of the **while** loop.

For the writer process in figure 4.21, 3 P calls and 2 V calls will be made per iteration in the best case. In the worst case, 3 P calls and 4 V calls will be made.

For the reader process in figure 4.13, 2 P calls and 2 V calls will be made for each iteration of the loop in the best case. In the worst case, 3 P calls will be made and 3 V calls will be made per loop iteration.

For the writer process in figure 4.13, 2 P calls and 2 V calls will be made per iteration in the best case. In the worst case, 3 P calls and 3 V calls will be made.