5.8 The first known correct software solution to the critical-section problem for two processes was developed by Dekker. The two processes, P0 and P1, shared the following variables:

```
boolean flag[2]; /* initially false */
int turn;
```

The structure of process P_i $(i \in \{0, 1\})$ is shown in Figure 5.21. The other process is P_j $(j \in \{0, 1\})$. Prove that the algorithm satisfies all three requirements for the critical-section problem.

```
do {
        flag[i] = true;

        while (flag[j]) {
                if (turn == j) {
                        flag[i] = false;
                        // do nothing
                        while (turn == j) { }
                        flag[i] = true;
                }
        }

        /* critical section */

        turn = j;
        flag[i] = false;

        /* remainder section */
} while (true)
```

Proof.

1. Mutual Exclusion
Each process sets their flag to true [2], indicating that they want access to the mutual data, this flag will be set to false once they have completed their critical section [16]. They then both enter a loop which cannot be broken out of while the other process wants access to the critical section [4]. Turn must initially be i or j, if it is the other processes turn, the current process will give up its flag [6], breaking the other out of the loop then the current process will in turn enter a new loop waiting for turn to change (to itself) hl[8]. The other process can now execute it's critical section, set the turn to break the other process out of the inner loop [8](which sets the other flag to true), and then the current process turns off flag breaking the other process out of the outer loop [4]. The other process can now execute critical section code in turn. If the current process after executing its critical section reenters the loop and sets flag back to true before the other process is scheduled [2], the other process will continue to loop [4], However as turn has changed, the current process will give up its flag at [6] which finally allows the other process, and only the other process to enter the critical section. Repeating the logic above.

2. Progress
If no process is currently executing the critical section, as turn is set, one process must be ready to enter the critical section. If the process who's turn does not wish to enter the critical section (i.e. they are in the remainder section) then they must have set turn to the other process [15] thus guaranteeing the other process (who wants access) entrance into the critical section. If both processes do not want access to the critical section, then one must be in their remainder section while the first is still trapped in the loop, again, as turn has changed [15] one process will break out of the loop at [8] switching its state to desire access to the critical section, which it will then be free to execute.

3. Bounded Waiting
Preserved via the turn variable. As mentioned above, the turn variable switches state after each execution of the critical section. This guarantees that the other process will be the next to enter its critical section. □

5.11 Explain why interrupts are not appropriate for implementing synchronization primitives in multiprocessor systems.

If a process does not wish to give up the its processor via interrupts, it can only guarantee mutual exclusion for processes which would run on the same processor. Since we are assuming a multiprocessor system, it is possible that another process could modify the data which we are trying to guarantee mutual exclusion on, by simply running on another of the systems processors. Thus this solution is insufficient.

5.16 The implementation of mutex locks provided in Section 5.5 suffers from busy waiting. Describe what changes would be necessary so that a process waiting to acquire a mutex lock would be blocked and placed in a waiting queue until the lock became available.

We would need to introduce a queue where processes can sleep (not execute) while they are waiting for a locked mutex. They will enter this queue on a call to acquire for a mutex which is unavailable, and will be released from this queue, when the process who currently holds the lock calls release. Note I am using the books terminology from Section 5.5, the lecture equivalent would be wait() and signal() respectively. Below, we can see a rough implementation modifying the books implementation, which incorporates queue's in place of the while loop. Note that like the book, I am not using testAndSet with a lock to guarantee only one process modifies available at a time, thus available is assumed to be thread safe.

```
acquire() {
        if (!available) {
                // add this process to the waiting queue
                block();
        }

        // take control of available
        available = false;
}

release() {
        // open up the mutex for the next process we are going to release
        available = true;
```

```
        // then release the next process P from the waiting queue
        wakeup(P);

}
```

5.29 How does the signal() operation with monitors differ from the corresponding operation defined for semaphores?

If a signal is performed on a monitor with no waiting queue, the signal is simply ignored. If a wait is then performed, the calling thread will simply be blocked. With a semaphore, the signal call will never be ignored as it will always at least increment the semaphore value. A future wait call will, instead of blocking, immediately succeed due to this incremented value.

5.32 A file is to be shared among different processes, each of which has a unique number. The file can be accessed simultaneously by several processes, subject to the following constraint: the sum of all unique numbers associated with all the processes accessing the file must be less than n. Write a monitor to coordinate access to the file.

The monitor is fairly straightforward, assumes n is a constant defined elsewhere, further assumes *pnum* is the corresponding process number to a process which is also available from elsewhere. Note, there is the possibility of a blocking process, if $pnum > n$ then even if no processes are accessing the file, this process will never leave the queue as there is not enough room to access the file. Further, a call to signal with a non-empty queue does not guarantee a process will access the file, even if there is a process in the queue with a small enough pnum to read the file, the queue will continue to wait until the head has enough room (this is why a while loop is used instead of an if statement).

```
// Use of the Monitor will look something like this.
ReadFile.begin_access(pnum); // pass the assigned process number
/* Access file... */
ReadFile.end_access(pnum);

Monitor ReadFile {
        // data section
        int sum; // current sum of readers, sum < n by definition of problem
```

```
        condition available; // is the file available for reading

        // procedure section
        void begin_access(int pnum) {
                if (pnum >= n) {
                        // ERROR - this process can never access the file
                        // and would block all future processes from doing so
                        // here I will just terminate the calling process
                        exit(-1);
                }

                // wait until there is enough room,
                // if sum = 8, n = 9 and pnum = 3
                // we may have to wait for a couple processes
                // to finish if their pnum is only 1
                // before we can continue
                while ((sum + pnum) >= n) {
                        available.wait();
                }

                sum += pnum;
        }

        void end_access(int pnum) {
                // open up some space
                sum -= pnum;

                // check if there is enough room now for the next process
                // if not, they will continue to wait
                available.signal();
        }
}
```