5.8

Mutual Exclusion:

This is ensured by the flag and turn implementation. Even if both processes set their flag to true, only one process will enter its critical section only if it is their turn. The other process (even if it's flag is true) cannot enter its critical section until turn is updated after the other process exits its critical section.

Progress:

Ensured by flag and turn implementation as well, turn only gets set to Pj after Pi has exited its critical section. This prevents a problem like we saw in the "hungry philosopher" where each process is politely waiting, here the waiting process (Pj) only enters its critical section after the current process (Pi) finishes its critical section and updates turn.

Bounded Waiting:

Ensured by the turn variable, no process will wait for an infinite amount of time, once a process exits its critical section, it sets the turn to the other process so then the next process can enter its critical section. Without turn, the first process would repeatedly enter it's critical section then end, and the other process would never get to enter its critical section.

5.11

Not sufficient since disabling interrupts only stop processes from being able to execute on the processor that the interrupts are disabled, there is nothing to hold the other processor(s) accountable and there would be no mutual exclusivity. Or if interrupts are disabled for all processors, then everything could come to a screeching halt and nothing would be able to get done.

5.16

Each lock would have a queue associated with it. If a process discovered the lock is unavailable it would be pushed onto the queue, when the lock becomes available it would de-queue (FIFO) a waiting process. The processes on the queue are blocked until the lock becomes available (semaphore = 0) then wakes up the next waiting process. Since queues are FIFO de-queueing a process is equivalent to aging, the process that has been waiting the longest will have a smaller index in the queue.

5.29

Signal in a monitor moves a process from the waiting queue to the ready queue for it to be executed. Signal in reference to semaphore is called when the process is done using a resource. Note also that a monitor ensures mutual exclusivity, only one process can be utilizing resources (changing state of monitor) at a time, if another resource wants access it has to wait. However in the case of a semaphore the process that releases the resource doesn't need to be the same that acquired it.

5.32

```
Monitor fileAccess
begin
        total : integer
        id : integer (id of process attempting to access)
        full : boolean (checks if total + id > n)
        CanAccess : condition

        procedure access(id)
        begin
                if full then CanAccess.wait //total is > n
                total := total + id;
                CanAccess.signal
        end access(id)

        procedure release(id)
        begin
                if full CanAccess.wait //make sure we are full
                total := total - id; //decrement total
                full := false;
                CanAccess.signal
        end release(id)

        begin
                total := 0;
                full := false;
end fileAccess
```