Systems Programming and Operating Systems	**Assignment 2**
CS414, Summer 1996	**July 17, 1996; Due on July 25, 1996**

Problem 1 (24 points)

1. Mutual Exclusion
   To prove mutual exclusion we have to show that both the processes are not in the critical section at the same time. The necessary condition to prove is that $CS_0 \wedge CS_1 = false$ where
   $CS_0 = ((in[1] == false) \vee (in[1] == true \wedge turn == 0)) \wedge (in[0] == true)$
   and $CS_1 = ((in[0] == false) \vee (in[0] == true \wedge turn == 1)) \wedge (in[1] == true)$

   This simplifies to $(in[0] == true) \wedge (in[1] == true) \wedge (turn == 0) \wedge (turn == 1)$ which cannot be true because turn cannot be both 0 and 1.

2. Finite progress

   Both processes cannot be stuck in the while loop. If they were then

   $(in[1] == true) \wedge (turn! = 0) \wedge (in[0] == true) \wedge (turn! = 1)$

   must be true which again is not possible because turn must be either 1 or 0.

3. No Prevention

   A process p_i in the remainder section cannot prevent a process waiting to enter the critical section because $in[p_i] == false$.

4. No Starvation

   To prove no starvation we must show that a process cannot repeatedly enter the critical section without letting the other process enter (we have already shown no prevention – if p_i is in the remainder section then the other process *can* enter the cirtical section). This follows from the statement

   ```
   turn = other;
   ```

   which gives the chance to the waiting process first.

Problem 2 (24 points)

We assume RISC like instructions ie. we need three instructions LOAD, ADD, STORE to do the required operations.

1. The possible solutions are –
   (a) 31 – P_1 followed by P_2.
   (b) 32 – P_2 followed by P_1.
   (c) 21 – P_1 does a LOAD, followed by P_2 and then P_1 completes.

2. Signals cannot block a process and so all the previous three cases still hold.

3. Since $s_x == 0$ both processes will deadlock.

Problem 3 (12 points)

1. Since there are no interrupts and no premptions, every process will run to completion and hence is inherently atomic. Hence no special code is required to protect critical sections.

2. On a uni-processor, we would use busy-waiting for short critical sections (it is very unlikely that a process in the critical section will be preempted since the critical section is small and so there will be really no need for the entering process to do much busy-waiting) and long critical sections with sleep-wakeup (if the critical section is long, it is likely the process in it will be preempted; if we used busy-waiting the entering process will use it's entire time-slice in the loop when it would be better to just let the preempted process continue).