TECHNIQUES FOR AVOIDING RACE CONDITION - I

Techniques for avoiding Race Condition:

1. Disabling Interrupts
2. Lock Variables
3. Strict Alteration
4. Peterson's Solution
5. TSL instruction

1.Disabling Interrupts:

The simplest solution is to have each process disable all interrupts just after entering its critical region and re-enable them just before leaving it. With interrupts disabled, no clock interrupts can occur.

The CPU is only switched from process to process as a result of clock or other interrupts, after all, and with interrupts turned off the CPU will not be switched to another process. Thus, once a process has disabled interrupts, it can examine and update the shared memory without fear that any other process will intervene.

Disadvantages:

1. It is unattractive because it is unwise to give user processes the power to turn off interrupts. Suppose that one of them did, and then never turned them on again?
2.Furthermore, if the system is a multiprocessor, with two or more CPUs, disabling interrupts affects only the CPU that executed the disable instruction. The other ones will continue running and can access the shared memory.

Advantages:

it is frequently convenient for the kernel itself to disable interrupts for a few instructions while it is updating variables or lists. If an interrupt occurred while the list of ready processes, for example, was in an inconsistent state, race conditions could occur.

2.Lock Variables

- a single, shared, (lock) variable, initially 0.
- When a process wants to enter its critical region, it first tests the lock.
- If the lock is 0, the process sets it to 1 and enters the critical region. If the lock is already 1, the process just waits until it becomes 0. Thus, a 0 means that no process is in its critical region, and a 1 means that some process is in its critical region.

Drawbacks:

Unfortunately, this idea contains exactly the same fatal flaw that we saw in the spooler directory. Suppose that one process reads the lock and sees that it is 0. Before it can set the lock to 1, another process is scheduled, runs, and sets the lock to 1. When the first process runs again, it will also set the lock to 1, and two processes will be in their critical regions at the same time.

3.Strict Alteration:

```
while (TRUE){
   while(turn != 0)      /* loop* /;
   critical_region();
   turn  = 1;
   noncritical_region();
}
```

(a)

```
while (TRUE) {
        while(turn != 1)      /* loop* /;
        critical_region();
        turn  = 0;
 noncritical_region();
}
```

(b)

A proposed solution to the critical region problem. (a) Process 0. (b) Process 1. In both cases, be sure to note the semicolons terminating the while statements.

Integer variable turn is initially 0.
It keeps track of whose turn it is to enter the critical region and examine or update the shared memory. Initially, process 0 inspects turn, finds it to be 0, and enters its critical region. Process 1 also finds it to be 0 and therefore sits in a tight loop continually testing turn to see when it becomes 1.

Continuously testing a variable until some value appears is called busy waiting. It should usually be avoided, since it wastes CPU time. Only when there is a reasonable expectation that the wait will be short is busy waiting used. A lock that uses busy waiting is called a spin lock. When process 0 leaves the critical region, it sets turn to 1, to allow process 1 to enter its critical region. This way no two process can enters critical region simultaneously.

Drawbacks:

Taking turn is is not a good idea when one of the process is much slower than other. This situation requires that two processes strictly alternate in entering their critical region.

Example:

- Process 0 finishes the critical region it sets turn to 1 to allow process 1 to enter critical region.
- Suppose that process 1 finishes its critical region quickly so both process are in their non critical region with turn sets to 0.
- Process 0 executes its whole loop quickly, exiting its critical region & setting turn to 1. At this point turn is 1 and both processes are executing in their noncritical regions.
- Suddenly, process 0 finishes its noncritical region and goes back to the top of its loop. Unfortunately, it is not permitted to enter its critical region now since turn is 1 and process 1 is busy with its noncritical region.

This situation violates the condition 3 set above: No process running outside the critical region may block other process. In fact the solution requires that the two processes strictly alternate in entering their critical region.

4.Peterson's Solution:

```
#define FALSE 0
#define TRUE  1
#define N     2                         /* number of processes */
int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE)*/
void enter_region(int process)          /* process is 0 or 1 */
{
    int other;                          /* number of the other process */
    other = 1 - process;                /* the opposite of process */
    interested[process] = TRUE;         /* show that you are interested */
    turn = process;                     /* set flag */
    while (turn == process && interested[other] == TRUE) /* null statement */;
}
void leave_region(int process)          /* process: who is leaving */
{
    interested[process] = FALSE;        /* indicate departure from critical region */
}
```

Peterson's solution for achieving mutual exclusion.

Initially neither process is in critical region. Now process 0 calls enter_region. It indicates its interest by setting its array element and sets turn to 0. Since process 1 is not interested, enter_region returns immediately. If process 1 now calls enter_region, it will hang there until interested[0] goes to FALSE, an event that only happens when process 0 calls leave_region to exit the critical region.

Now consider the case that both processes call enter_region almost simultaneously. Both will store their process number in turn. Whichever store is done last is the one that counts; the first one is lost. Suppose that process 1 stores last, so turn is 1. When both processes come to the while statement, process 0 executes it zero times and enters its critical region. Process 1 loops and does not enter its critical region.

5. The TSL Instruction

TSL RX,LOCK

(Test and Set Lock) that works as follows: it reads the contents of the memory word LOCK into register RX and then stores a nonzero value at the memory address LOCK. The operations of reading the word and storing into it are guaranteed to be indivisibleno other processor can access the memory word until the instruction is finished. The CPU executing the TSL instruction locks the memory bus to prohibit other CPUs from accessing memory until it is done.

```
enter_region:
      TSL REGISTER,LOCK      |copy LOCK to register and set LOCK to 1
      CMP REGISTER,#0        |was LOCK zero?
      JNE enter_region       |if it was non zero, LOCK was set, so loop
      RET                    |return to caller; critical region entered
leave_region:
     MOVE LOCK, #0           |store a 0 in LOCK
     RET                     |return to caller
```

One solution to the critical region problem is now straightforward. Before entering its critical region, a process calls enter_region, which does busy waiting until the lock is free; then it acquires the lock and returns. After the critical region the process calls leave_region, which stores a 0 in LOCK. As with all solutions based on critical regions, the processes must call enter_region and leave_region at the correct times for the method to work. If a process cheats, the mutual exclusion will fail.