Problems with use of semaphores

1. **Dead Lock**
2. **Time dependent errors**
   - a. Timing
   - b. Programmer error

Problems with use of semaphores: Dead Lock

S = Q = 1

P0	P1
Wait(S)	Wait(Q)
Wait(Q)	Wait(S)
.	.
.	.
Signal(S)	Signal(Q)
Signal(Q)	Signal(S)

Consider following order of execution:

Wait(S) in P0
Wait(Q) in P1
Wait(Q) in P0
Wait(S) in P1

Problems with use of semaphores: Timing

Timing Error **refers to those errors that happen only if a particular execution order takes place.**

Solution

Locking

Problems with use of semaphores: Programmer Errors

Suppose we have the semaphore mutex = 1 for a group of cooperating processes as follows:

```
Repeat
    P(mutex)
        Critical section
    V(mutex)
        Remainder
Until false;
```

One process at a time will be in its critical section

Problems with use of semaphores: Programmer Errors

Let us assume that programmer is sleepy and made a mistake in coding as follows:

Case 1	Case 2	Case 3
V(mutex)	P(mutex)	V(mutex)
.	.	.
.	.	.
Critical section	Critical section	Critical section
.	.	.
P(mutex)	P(mutex)	V(mutex)
All processes are in their C. sections	Only one process gets into its C. Section	All processes are in their C. sections

Problems with use of semaphores: Programmer Errors

Solution

Use of Critical Region Construct

We need to use a sharable variable of type T with the C.R.C. which is declared as follows:

Var u: shared T;

Examples

var u : shared integer

var u : shared array[1. .n] of boolean

Critical Region Construct

Var u: shared T;
Region u do A.

(i.e., while task A is being executed, no other processes can use the variable u.)

Critical Region Construct

How is programmer miscoding prevented?

Programmer codes :

Region u do A;

Compiler generates:

```
P(u)
  A;
V(u)
```

Abstract Data Type: skeleton

```
Type class-name = class
     Variable declarations
     Procedure entry Proc1( . . .)
               Begin . . . End;
     Procedure entry Proc2( . . .)
               Begin . . . End;
                    :
     Procedure entry Procn( . . .)
               Begin . . . End;
     Begin
               Initialization code
     End;
```

Integration of C.R.C. and ADT

```
Type frames = class
     var free: shared array[1..n] of boolean
     Procedure entry acquire (var index: integer);
        Begin
              region free
                 do for index = 1 to n
                        do if free[index] then Begin free[index] = false;  exit;
                                               End;
           index = -1;
        End;
     Procedure entry release( index: integer)
        Begin
                 region free
                        do  free[index] = true;
        End;

     Begin
                 region free
                        do for index = 1 to n
                                 do free[index] = true
     End;
```

Nested Critical Region Constructs

Region x do region y do s1;

Problem:

Nested Critical Region Constructs may cause deadlock.

Nested Critical Region Constructs

Example:
Process Q: region x do region y do s1;
Process R: region y do region x do s2;
x = y = 1;

```
P(x)                    P(y)
   p(y)                    p(x)
       S1;                     S2;
   V(y)                    V(x)
V(x)                    V(y)
```

Nested Critical Region Constructs

Solution:

Establishing a priority among shared variables.

Example: x > y

That is, if shared variable x is in use by process Q, then the shared variable y will not be used by any other process until x is released.

Nested Critical Region Constructs

Example:

Process Q: region x do region y do s1;

Process R: region y do region x do s2;

x = y = 1; x > y;

```
P(x)
    p(y)
        S1;
    V(y)
V(x)
```

```
P(y)
    p(x)
        S2;
    V(x)
V(y)
```

Conditional Critical Region Construct

Region u when B do A;

u is a shared variable

B is a boolean expression

If B is true and u is not in use then A is executed.

Conditional Critical Region Construct

Example: ***Producer and Consumer***

```
Var   buffer: shared record
      pool:array[0..n-1] of item;
      count, in, out: integer;
```

Region buffer when count <n
do begin
pool[in] = nextp
in = in +1 mod n;
count = count +1;
end;
	Region buffer when count >0
	do begin
	nextc =pool[out];
	out = out +1 mod n;
	count = count - 1;
	end;

PRODUCER	CONSUMMER

New language constructs

Both

Critical Region Construct and

Conditional Critical Region Construct

Delay multiple processes to enter their critical sections at the Entry section.

∴ **They are not 100% successful in synchronization of processes**

New language constructs

Solution:

There is a need for a construct that can delay the execution of a process in two places:

at the entry section and

inside the critical section.

New language constructs

To do so,

***Conditional Critical Region Construct* is modified.**

```
Old CCRC
Region u when B do A
--------------------
Region u
      when B
         do A
```

```
New CCRC
Region u
    do
        S1;
        await(B)     A
        S2;
    end;
```

New language constructs

For use of the modified *Conditional Critical Region Construct*

We need to define

- ***Condition variable***
- ***Monitor***

New language constructs

1-Condition variable

```
var x: condition;
```

A condition variable is an object of a class defined by an abstract data type with two methods of Wait and Signal.

If process P1 invokes x.Wait then P1 is suspended until another process, like P2 invokes x.Signal.

If process P2 invokes x.Signal then
- **A suspended process, say P1, on x resumes its process.**
- **If there is no suspended process on x, then**
  - **no action takes place. It looks like that x.Signal was never executed.**

New language constructs

2-Monitor follows the syntax of an abstract data type.

We use the integration of condition variable and monitor to solve the problem of dining philosopher using

Monitor

```
Type dining-philosophers = monitor
Var state: array[0..4] of (thinking, hungry, eating);
Var self: array[0..4] of condition;

Procedure entry pickup(i: 0..4)
    begin
        state[i] = hungry;
        test(i);
        if state[i] != Eating { self[i].wait};
    end
Procedure entry putdown(i: 0..4)
    begin
        state[i] =thinking;
        test(i-1 mod 5);
        test(i+1 mod 5);
    end;

Procedure entry test(k: 0..4)
  begin
     If state[k-1 mod 5] != eating;
        and state[k] = hungry
        and state[k +1 mod 5] != eating{
                state[k] = eating;
                self[k].signal;
        }
  end;
Begin
     for i =0 to 4 { state[i] = thinking)
End.
```

Monitor

What is the difference between a monitor and a class?

The difference is in use of:

Critical Region Construct (C.R.C) by class and

Modified Conditional C.R.C by monitor.

C.R.C. provides the mutual exclusion on procedures

M.C.C.R.C provides the mutual exclusion on procedures and resources

In the previous example, a philosopher is a resource.

A resource can self.wait