Systems Programming and Operating Systems **Indupraks Kodukula**
CS414, Summer 1996 **Class#4, 7/11/96**

1 Class #4

At this point, a formal definition of the critical solution problem as well as a few solutions are in order to appreciate the complexity of the problem.

1.1 Problem Formulation

For obtaining solutions to the *critical section problem* we'll first consider the case when there are two processes. Each of our processes will be represented as problems with the following template:

```
Process
  while true do
    <enter>
    CS
    <exit>
    NCS
  end
end
```

This code fragment is a *static* representation of a process. For the rest of this discussion, we will assume that there are two dynamic instances of this process and we'll call them P_1 and P_2. (these could arise by two *fork* calls for example). ¡enter¿ and ¡exit¿ represent portions of code that we have to determine. They respectively represent initialization before entering the critical section and cleanup code after executing the critical section.

1.2 Goals for Solutions

- P_1 and P_2 are not executing in the critical section CS in the same time (critical section condition)
- A process not in the critical section cannot prevent the other from entering the critical section
  - P_1 in NCS cannot prevent P_2 from entering CS
  - P_2 in NCS cannot prevent P_1 from entering CS
- If several processes want to enter the critical section at the same time (i.e. they are executing the ¡enter¿ code at the same time), only they can participate in determining which process wins out eventually. Furthermore, the outcome of this "arbitration" process should not be postponed indefinitely.
- Individual processes should not starve.

In this class, we'll study several attempts at a solution to the critical section problem. That will give us some insight into why this is a hard problem.

1.3 First Solution attempt

We'll have a single shared variable called *turn*. Also, assume that each process has a unique id assigned to it. Given these assumptions, we can write the following program to ensure avoidance of critical section overlap.

```
Process
<enter> while turn != id do
CS
<exit>  turn = turn  ⊕  1
```

turn is initialized to 0.

The problem with this approach is that it violates the following condition:

- A process not in the critical section cannot prevent the other from entering the critical section

Think of the case when turn is set to 1, but then process 1 is off doing something and process 0 needs to enter the critical section again.

1.4 Second solution attempt

In this attempt, we'll use a variable called *in[2]* to indicate if any process is inside the critical section. Process 0 sets in[0] to true if it wants to enter the critical section and process 1 sets in[1] to true if it wants to enter the critical section. Each process does the following to enter the critical section: If the entry of in corresponding to the other process is set to true, it simply loops, until it becomes false. Then it enters the critical section and sets its own location of *in* to true.

```
Process
<enter> in[self] := true; while (in[other] == true) do skip;
CS
<exit> in[self] = false;
```

The problem with this approach is that it violates the following condition:

- If several processes want to enter the critical section at the same time (i.e. they are executing the ¡enter¿ code at the same time), only they can participate in determining which process wins out eventually. Furthermore, the outcome of this "arbitration" process should not be postponed indefinitely.

Why is this? Let's see what happens if both the processes desire to enter the critical section at the same time. They respectively set in[0] and in[1]. Then process 1 loops forever because in[0] is set and process 0 loops forever because in[1] is set. So, we do satisfy the critical section property, but we don't satisfy the *finite progress* condition.

1.5 Third Solution

We see that each of the above solutions satisfies only two of the three conditions needed for our solution. So, let's try to combine the two approaches for a solution. We'll use a an array *in[2]* for each process to indicate when it wants to enter the critical section. If at any time, only one process wants to enter the critical section, we are OK. Only time a problem arises is when both the processes want to enter the critical section at the same time. This is where we'll use the variable *turn*. At any time, *turn* will be set to which process should win out if both processes are trying to enter the critical section at the same time. More succintly:

$$\begin{aligned} in[i] &: \text{Process i is trying to enter the critical section} \\ turn &: \text{The process that is permitted to proceed if more than process would like to enter CS} \end{aligned} \tag{1}$$

So, again if we are dealing with two processes, then the following condition holds when process 0 is in the critical section:

$$CS_0 = ((in[1] == false) \vee ((in[1] == true) \wedge (turn == 0))) \wedge (in[0] == true) \tag{2}$$

Similarly, the condition that holds when process 1 is in the critical section is:

$$CS_1 = ((in[0] == false) \vee ((in[0] == true) \wedge (turn == 1))) \wedge (in[1] == true) \tag{3}$$

It is reasonably easy to verify that $CS_0 \wedge CS_1 ==$ false, i.e. the two processes are not in the critical section at the same time. Finally, the code fragment that can be used to implement this approach is the following

```
<enter> in[self] = true; turn = other;
        while (in[other] && turn != self) do skip;
CS
<exit> in[self] = false;
```

It is also quite straightforward to show from this that the other conditions of critical section solutions are not violated.