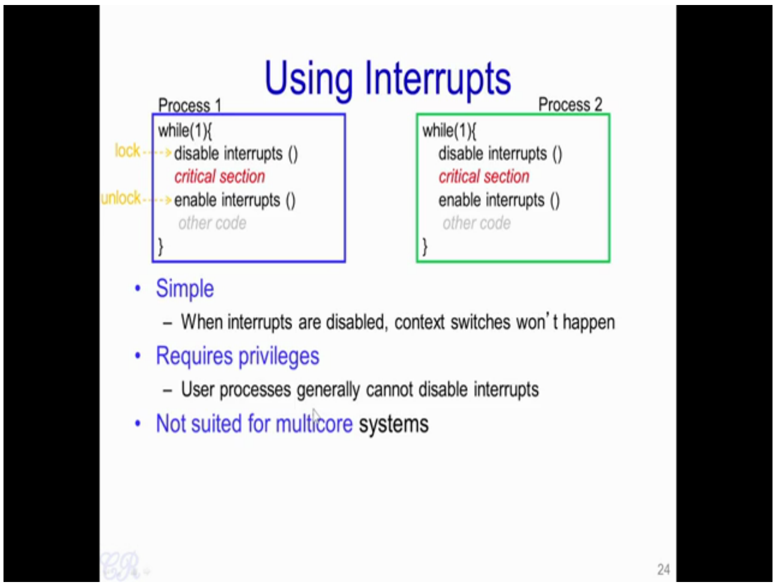
tIntroduction to Operating Systems
Prof. Chester Rebeiro
Department of Computer Science and Engineering
Indian Institute of Technology, Madras

Week - 06
Lecture – 25
How to Implement Locking (Software Solutions)

Hello. In the previous video, we had seen an Introduction to Critical Sections. We had seen that in order to solve the critical section problem, there were 3 requirements. First is mutual exclusion, second progress and third bounded weight. In this video, we will see some software solutions for the critical section problem.

(Refer Slide Time: 00:43)



Using Interrupts

Process 1
```
while(1){
lock   --> disable interrupts ()
           critical section
unlock --> enable interrupts ()
           other code
}
```

Process 2
```
while(1){
    disable interrupts ()
    critical section
    enable interrupts ()
    other code
}
```

- Simple
  - When interrupts are disabled, context switches won't happen
- Requires privileges
  - User processes generally cannot disable interrupts
- Not suited for multicore systems

Let start with the more simple solution that is by disabling interrupts, so this solution is only applicable for single core systems; in with multi core systems, essentially because of the advanced programmable interrupt control has present in the systems interrupts are routed to several processes, therefore this solution will not work on multi core systems. On the other hand, for single core processes or single core systems, disabling interrupts will work. Essentially we have seen that if interrupts are disabled then it put prevent context switching. And preventing context switching would mean that the critical section would be not pre-empted during its execution.

Let us look at how interrupts can be used to solve the critical section problem. So, we have the two processes process 1 and process 2, and they have this critical section which was shown in red. In this critical section, both processes access or modify the same shared data. In order to ensure that the critical section problem is solved and that both processes do not end up in the critical section at the same time, we disable interrupts before entering into the critical section.

While, on the other hand, while leaving the critical section the interrupts are enabled again. Disabling interrupts as we have seen will prevent the process from getting pre empted at the end of it is time slice. Essentially, they would not be a timer interrupt that would occur which would prevent process 1 from pre-empting. Therefore, we see that this is a very simple solution and process 1 will continue to execute without any other interrupts that occurring as a result of the disabling of interrupts.

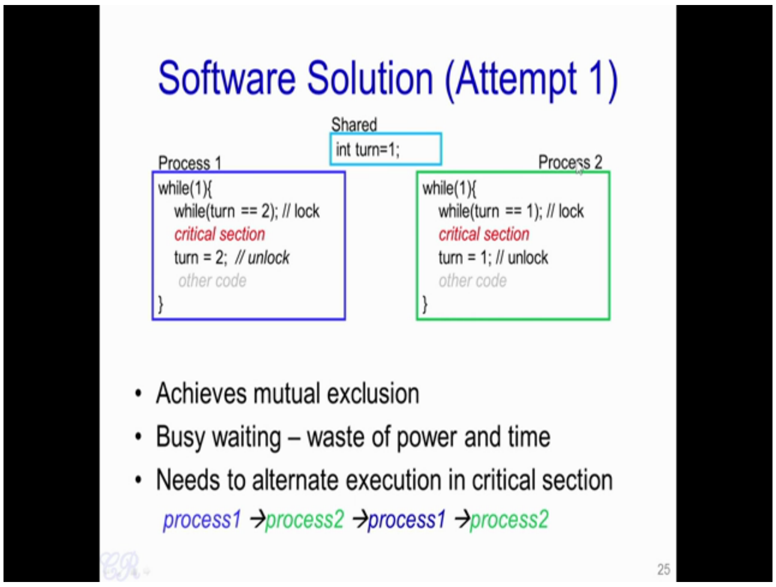
Similarly, when process 2 enters into this critical section, it disables interrupts; and on leaving the critical section, it enables interrupts again. So, this application of disabling and enabling interrupts is similar to the locking mechanism before entering the critical section and the unlocking mechanism at the exit of the critical section. The locking will ensure that only one process enters or executes in the critical section at a given time. While the unlocking mechanism that is when interrupts are enabled would then allow other processes the chance to enter or execute in the critical section.

So, this method of solving the critical section problem is simple. It just requires single instruction in order to disable or enable interrupts. Thus, preventing context which is from happening; however, the limitations of using interrupts is that it requires higher privilege level. So, normal application programs which run in user space will not be able to disable and enable interrupts in such a way. Therefore, this solution is only applicable for code that runs in the kernel.

It is only the kernel or the operating system that is allowed to disable interrupts and enable interrupts again. Therefore, this solution is only applicable for operating system code. And as we have seen before this disabling and enter enabling of interrupts is not suited for multi core systems; essentially because when interrupts are disabled on a multi core system, it would mean that interrupts corresponding to that code alone is disabled. On the other hand, interrupts are still allowed on the other cores which are running. Thus is on a multi core system when disable interrupts is invoked, all interrupts to only that core or that core in the processor is disabled, other processors such as a processor which is running this process 2 will not be affected by the disabling of interrupts.

Therefore, this solution is not suited for multi core systems. So, what we will do next is we will try to actually build our own solution for the critical section problem. So, we will start with a very simple solution, we will see it is drawbacks, and then we will gradually modify that particular solution until we reach a point where we have solved the critical section problem. Let us look at this.

(Refer Slide Time: 05:47).



Software Solution (Attempt 1)

Shared
```
int turn=1;
```

Process 1
```
while(1){
    while(turn == 2); // lock
    critical section
    turn = 2; // unlock
     other code
}
```

Process 2
```
while(1){
    while(turn == 1); // lock
    critical section
    turn = 1; // unlock
    other code
}
```

- Achieves mutual exclusion
- Busy waiting – waste of power and time
- Needs to alternate execution in critical section

  process1 →process2 →process1 →process2

25

Let us start with our first attempt. Let us say we have this critical section in process 1 and this critical section in process 2. And both these critical sections are manipulating the same data. In addition to this shared data, we also define other shared data known as turn, and set this value of turn to 1. So, what happens we look into more detail? So, when process 1 begins to execute it sees that turn equal to 1, therefore this while loop immediately exits and the critical section gets executed.

Now during the time when process 1 is in the critical section, when process 2 arrives at this particular point, it sees that turn equal to 1, and therefore it will continue to loop in this particular while loop. The only time that it is capable of exiting from this while loop is when the first process sets turn to 2, so when the value of turn is set to 2, and remember that turn is a shared variable, therefore, this while loop in process 2 will terminate. And process 2 will then enter into the critical section. At the exit of the critical section, process 2 will set the value of turn to 1. So, setting the value of turn to one would mean that a process 1 can then enter into the critical section.

So, essentially when we look at the locking and unlocking mechanism to solve this critical section problem, we see that this while loop present over here is the locking mechanism, while the unlocking mechanism is the third line that is setting the value of turn to 2. The second observation that we make is that this algorithm or the solution for the critical section problem requires that process 1 executes into the critical sections, and then because turn is set to 2 over here, after process 1 completes its critical section execution, process 2 should execute. So, once process 2 completes its execution, process 1 will execute and so on.

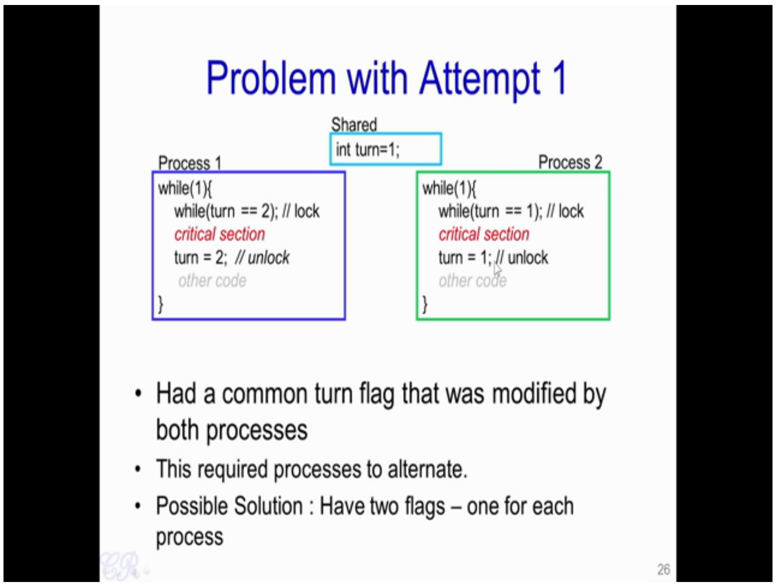
Essentially there is an alternating behaviour for the critical section. First, process 1 will needs to execute the critical section because the value of turn is equal to 1; then when it completes it sets turn to 2, so process 2 will execute. And process 2 will set the value of turn to 1 at the end of its critical section which will result in process 1 executing in the critical section and so on.

So, we see that there is an alternating behaviour between process 1 and process 2. Quite naturally because the value of turn is shared between both the processes and as we have seen only one process could satisfy this while condition, and break out from this while loop. Therefore, exactly one process would enter into the critical section at a given time. Therefore, this solution achieves mutual exclusion. However, there are two limitations of the solution; one is the busy waiting. So while process 1 is executing in this critical section process 2 would endlessly execute in this particular while loop.

Note that process 2 is in the ready state or the running state it is not in the blocked state, and therefore, it would consume power as well as time, because it needs to execute in the CPU periodically in order that this value of turn is checked. Another limitation as you have seen before it is that this solution for the critical section problem requires that process 1 and process 2 alternates, its access into the critical section, so what it means is that a first process 1 should execute then process 2, process 1, process 2 and so on. So, this creates a problem especially where the progress requirement of the critical section solution.

So, the progress requirement condition stated that if no process is in the critical section, and process request for the critical section, then it should be offered the lock, essentially it should be able to execute in the critical section. We see that this solution will not satisfy the progress requirement. For instance let us say that process 2 begins to execute before process 1; so process 2 comes here and it executes this while turn equal to 1, and we see that since process 1 has not get started, so if the progress condition needs to be met, so process 2 should enter into the critical section. But this is not so, therefore this is not an a good solution for the critical section problem.

(Refer Slide Time: 11:16)



Problem with Attempt 1

Shared
```
int turn=1;
```

Process 1
```
while(1){
   while(turn == 2); // lock
   critical section
   turn = 2; // unlock
   other code
}
```

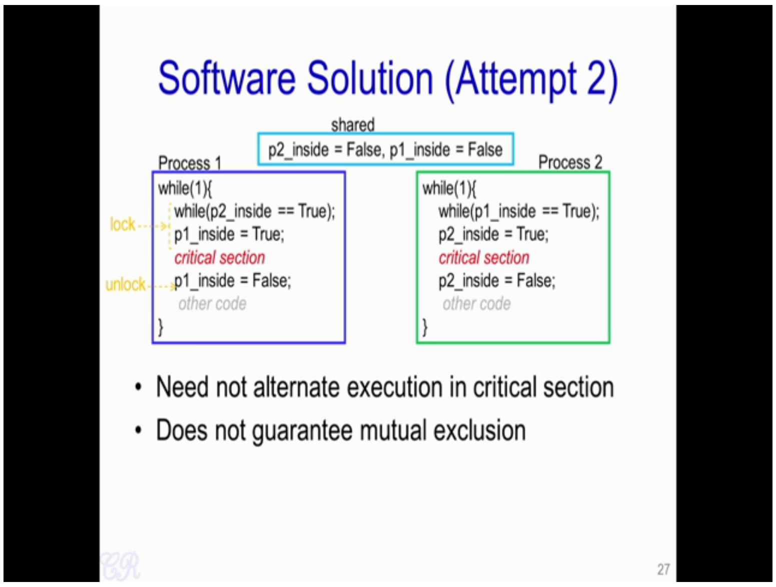
Process 2
```
while(1){
   while(turn == 1); // lock
   critical section
   turn = 1; // unlock
   other code
}
```

- Had a common turn flag that was modified by both processes
- This required processes to alternate.
- Possible Solution : Have two flags – one for each process

26

The main drawback or the main problem with our first attempt to solve the critical section problem was that reduced a common turn flag that was modified by both processes. So, this turn flag was set to 2 in process 1 and set to 1 in process 2; and this force the two processes to alternate execution in the critical section. So, one possible solution for this or one possible thing we could actually make better in our second attempt is to not have a single shared flag, instead we will have two flags one for each process.

(Refer Slide Time: 12:02)



Software Solution (Attempt 2)

shared
```
p2_inside = False, p1_inside = False
```

Process 1
```
while(1){
    while(p2_inside == True);
    p1_inside = True;
    critical section
    p1_inside = False;
    other code
}
```

Process 2
```
while(1){
    while(p1_inside == True);
    p2_inside = True;
    critical section
    p2_inside = False;
    other code
}
```

lock
unlock

- Need not alternate execution in critical section
- Does not guarantee mutual exclusion

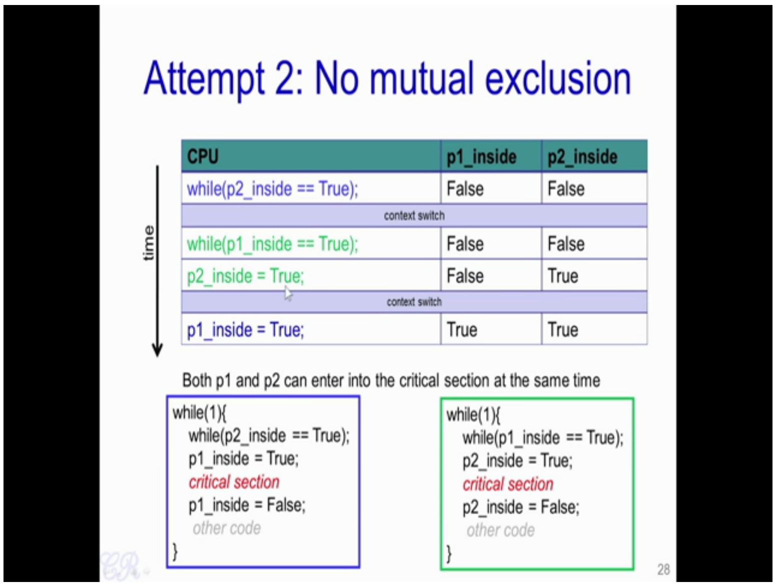
Let us see the second attempt to solve the critical section problem. So, this is the second attempt. So, unlike the previous attempt we now have two flags present; these two flags are also shared among the two processes; that means that this flag p 2 inside can be accessed from both process 1 as well as process 2. However, it is process 2 which actually changes or modifies this flag, so process 2 changes from false to true or true to false, while process 1 only reads the status of this flag; it only determines whether the flag is a true or a false.

Similarly, the second flag is p 1 inside. Now these flags p 2 inside and p 1 inside are used during the locking and unlocking of the critical section. Essentially before process 1 enters into the critical section, it does two things. First, it needs to check that the second process that is p 2 is not inside the critical section; and this happens, when p 2 underscore inside is equal to false. So, if p 2 underscore inside is true, it would mean that the second process is executing in the critical section then it sets it own flag that is p 1 underscore inside to true and executes in the critical section. While at the exit of the critical section is sets p 1 underscore inside to false.

Similarly, process 2 would first execute this while loop until p 1 underscore inside is false, so a value of p 1 underscore inside is false would mean that it process p 1 as exited from the critical section and is no longer in the critical section. And after it obtain such a condition, the while loop is completed and execution comes over here where process 2 sets p 2 underscore inside equal to true indicating that it is inside the critical section. And at the end of the critical section, just before exiting. it sets that p 2 underscore inside equal to false.

Now this second solution does not require that the two processes alternate execution in the critical section. So, for instance, since the initial values of p 1 inside and p 2 inside are false, so suppose process 2 executes first, so it sees p 1 underscore inside equal to false, so it breaks from this loop and enters into the critical section. Similarly, if process 1 does not start executing, it will keep entering into the critical section without requiring process 1 to execute. However, the problem with this particular solution is that it does not guarantee mutual exclusion rather under some circumstances it is possible that process 1 as well as process 2 is present in the critical section at the same instant of time.

(Refer Slide Time: 15:45).



Attempt 2: No mutual exclusion

CPU	p1_inside	p2_inside
while(p2_inside == True);	False	False
context switch		
while(p1_inside == True);	False	False
p2_inside = True;	False	True
context switch		
p1_inside = True;	True	True

Both p1 and p2 can enter into the critical section at the same time

```
while(1){
   while(p2_inside == True);
   p1_inside = True;
   critical section
   p1_inside = False;
    other code
}
```

```
while(1){
   while(p1_inside == True);
   p2_inside = True;
   critical section
   p2_inside = False;
    other code
}
```

28

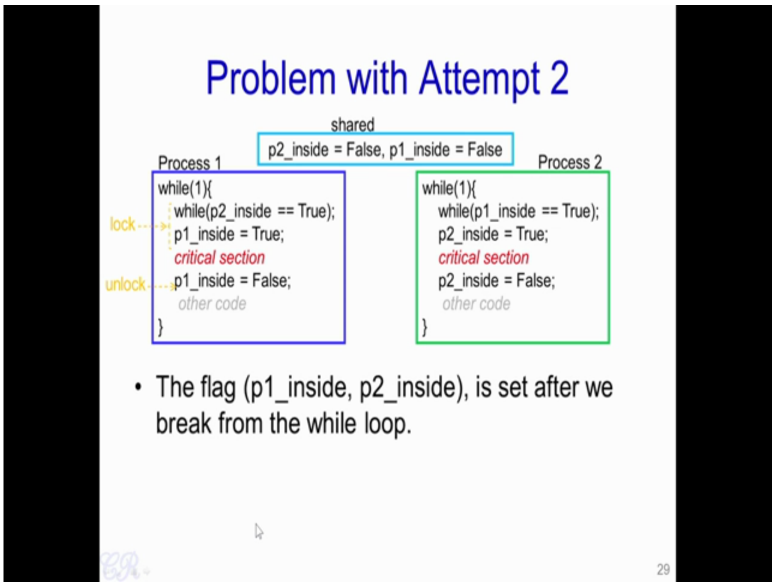
Let us see when this mutual exclusion does not hold for our second attempt. This particular table here shows the execution of various statements in the CPU with respect to time that means, that this particular while loop executes and while since it is in blue colour it means that it corresponds to the process 1. Then while p 1 underscore inside equal to true and p 2 underscore inside equal to true, so this is in the green colour, so it corresponds to process 2. And p 1 underscore inside equal to true again corresponds to process 1.

So, this is how the CPU would execute instructions corresponding to the two processes and we have like several contexts which is that occur during the execution and these are the status of the various flags as the execution progresses.

Let start with process 1, we have initially set that p 1 underscore inside and p 2 underscore inside is both false indicating that both processes are not in the critical section. Therefore, this particular while loop will complete, essentially because p 2 inside is false; now suppose there is a context switch that occurs resulting in process p 2 which executes, so it will also execute these statements while p 1 underscore inside is false, so the while loop will break. And then it sets p 2 underscore inside equal to true.

Now immediately after setting this suppose there is a context switch that occurs and process p 1 will continue to execute and it sets p 1 underscore inside equal to true. Essentially due to the context switch, it will continue to execute from where it as stopped that is it has stopped just soon after completion of the while loop and then it continues with the next instruction over here that is setting of p 1 inside equal to true. So, here we see we have a state where p 2 underscore inside equal to true indicating that p 2 is in the critical section as well as p 1 underscore inside equal to true indicating that p 1 is also in the critical section. Therefore, we have not able to achieve mutual exclusion.

(Refer Slide Time: 18:15).



Problem with Attempt 2

shared

```
p2_inside = False, p1_inside = False
```

Process 1

```
while(1){
    while(p2_inside == True);
    p1_inside = True;
    critical section
    p1_inside = False;
    other code
}
```

Process 2

```
while(1){
    while(p1_inside == True);
    p2_inside = True;
    critical section
    p2_inside = False;
    other code
}
```

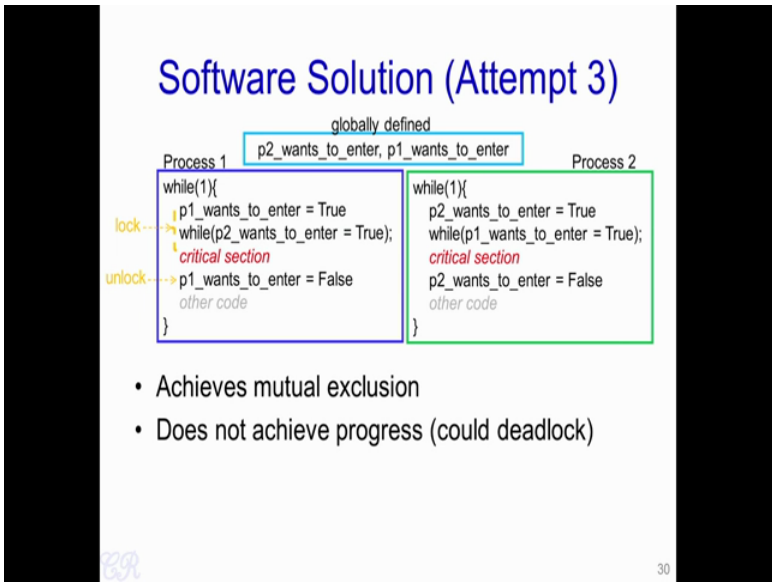
lock
unlock

- The flag (p1_inside, p2_inside), is set after we break from the while loop.

29

So, the main problem with this second attempt to solve the critical section problem was that we had two flags p 1 inside and p 2 inside. However, they were set after we break from the while loop that is only after this particular while loop completes an execution and breaks from this only then are we setting p 1 underscore inside to true and p 2 underscore inside equal to true. Essentially, what this means is that p 1 underscore inside and p 2 underscore inside are set to true only within the critical section, and this is what created the problem. Let us see if we can actually change these ordering a bit.

(Refer Slide Time: 19:03)



Software Solution (Attempt 3)

globally defined

p2_wants_to_enter, p1_wants_to_enter

Process 1

```
while(1){
    p1_wants_to_enter = True
    while(p2_wants_to_enter = True);
    critical section
    p1_wants_to_enter = False
    other code
}
```

lock
unlock

Process 2

```
while(1){
    p2_wants_to_enter = True
    while(p1_wants_to_enter = True);
    critical section
    p2_wants_to_enter = False
    other code
}
```

- Achieves mutual exclusion
- Does not achieve progress (could deadlock)

30

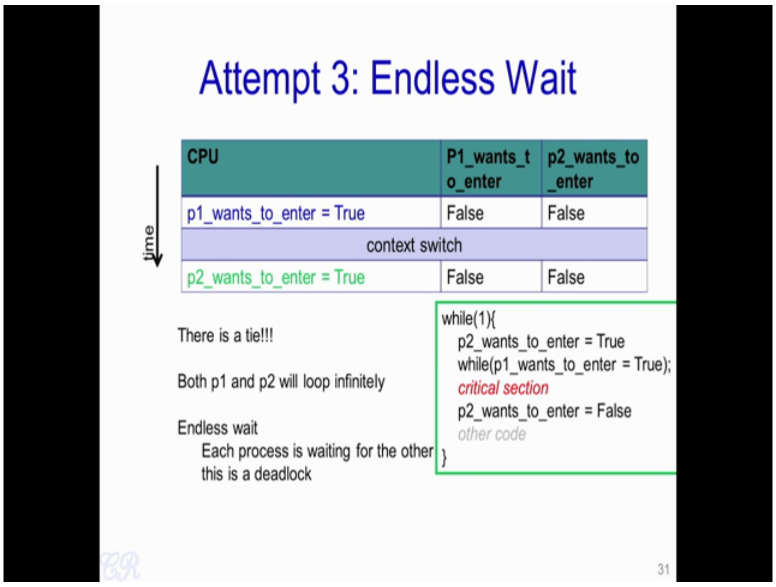
So, we will start with the third attempt that is the third attempt for a solving the critical section problem. And what we will do is just a simple change, so we will have these two flags instead of p 1 underscore inside, we will have now p 1 wants to enter into the critical section, and p 2 wants to enter into the critical section. Essentially, each of these flags is set to true when the corresponding process wants to enter into the critical section. So, the minor change we do over here with respect to the second attempt is that process 1 first shows us intention to enter into the critical section by setting p 1 underscore wants to enter to true.

And only then will it try to determine, if process p 2 is in the critical section or not. So, essentially it sets p 1 underscore wants to enter to true and then it would execute in this while loop until p 2 wants to enter is false. So, when p 2 wants to enter it becomes false indicating that the second process has completed execution in the critical section then process 1 will enter into the critical section.

At the end, after the critical section has completed execution p 1 sets the flag p 1 wants to enter to false so this setting of false will allow process 2 to enter into the critical section. Note that the only difference of this corresponding to our previous attempt was that we move the flag from inside the critical section that is after the while loop to before the while loop. So, what we see is that this particular solution achieves mutual exclusion, so, unlike the previous attempt where we obtained a condition where mutual exclusion was not achieved and we had two processes in the critical section at the same time.

In this solution, the mutual exclusion is achieved; essentially, it is guaranteed that when process 1 is in the critical section process 2 is not in the critical section and vice versa. So, what is not guaranteed is progress; essentially, we could obtain a state where both processes are endlessly waiting or endlessly executing in this while loop, and we will see how this thing occurs, so such a state is known as a deadlocked state.

(Refer Slide Time: 21:54)



Attempt 3: Endless Wait

CPU	P1_wants_to_enter	p2_wants_to_enter
p1_wants_to_enter = True	False	False
context switch		
p2_wants_to_enter = True	False	False

There is a tie!!!

Both p1 and p2 will loop infinitely

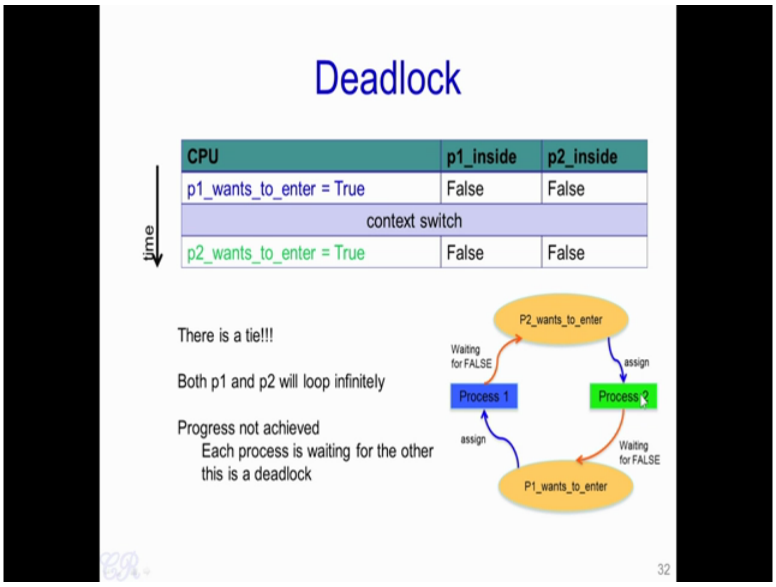
Endless wait
- Each process is waiting for the other
- this is a deadlock

```
while(1){
  p2_wants_to_enter = True
  while(p1_wants_to_enter = True);
  critical section
  p2_wants_to_enter = False
  other code
}
```

Let see the deadlock situation in this particular case. Let us look in this table, which shows how the CPU executes instructions with respect to time, and also this table shows the various values of the flags which are present. Let us start with process 1 executing and the CPU executes this particular statement and it sets the value of p 1 wants to enter to true. Then there is a context switch, which results in process 2 executing which sets the flag p 2 wants to enter to true.

Now you see that both processes have set their respective flags to true. Now when both processes enter into this while loop we see that in both cases process 1 and as well as process 2, this flag is set to true mean indicating that both processes will continue to execute this while loop endlessly. Essentially, process 1 is waiting for process 2 to set its flag to false, while process 2 is waiting or is waiting for process 1 to set its flag to false. So, essentially we have reached a state which where each process is waiting for the other process to do something. So, this is what is known as a deadlocked situation and it could lead to considerable amongst of problem in operating systems.

(Refer Slide Time: 23:39).



Deadlock

CPU	p1_inside	p2_inside
p1_wants_to_enter = True	False	False
context switch		
p2_wants_to_enter = True	False	False

There is a tie!!!

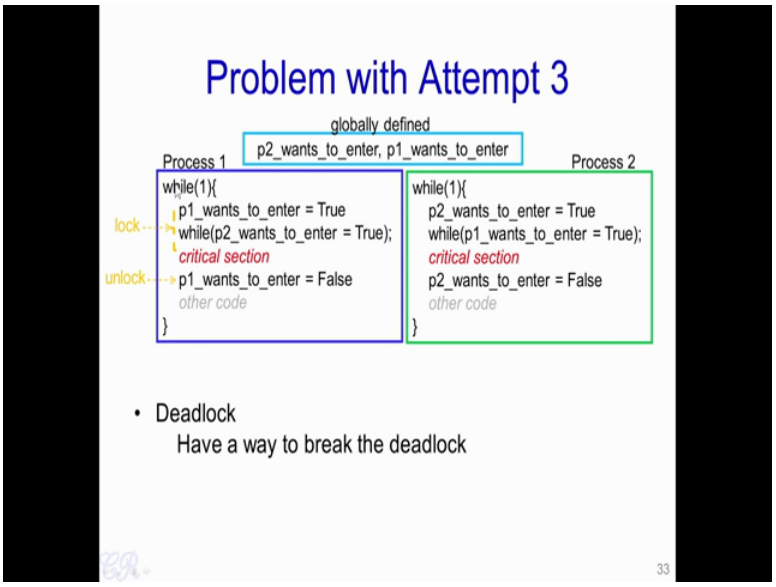
Both p1 and p2 will loop infinitely

Progress not achieved
  Each process is waiting for the other
  this is a deadlock

So, to give a more clear a view of what a deadlock is we have this particular figure here, where we have process 1 and process 2 present here. Now process 1 is in the while loop and it is looping continuously waiting for process 2 to set this flag p 2 wants to enter to false. Similarly, process 2 is waiting in the while loop or looping continuously and waiting for this flag p 1 wants to enter to be set to false. However, the problem here is that process 1 needs to change the value of this flag because this flag can be only changed by process 1; similarly process 2 needs to change the value of this flag since this flag can be only changed by process 2.

And therefore, we see that there is a tie; process 1 is waiting for process 2 to change the value of its flag; however, process 2 cannot change the value of its flag because it is waiting for process 1 to change the value of the flag. So, we get some kind of a cycle present over here. So, this cycle will result in both process 1 and process 2 waiting for each other for an endlessly amount of time, so this is known as a deadlock. So, in a later video, we will look more into details about how deadlocks are caused and how they can be prevented and avoided.

(Refer Slide Time: 25:18).



Problem with Attempt 3

globally defined

```
p2_wants_to_enter, p1_wants_to_enter
```

Process 1

```
while(1){
    p1_wants_to_enter = True
    while(p2_wants_to_enter = True);
    critical section
    p1_wants_to_enter = False
    other code
}
```

lock
unlock

Process 2

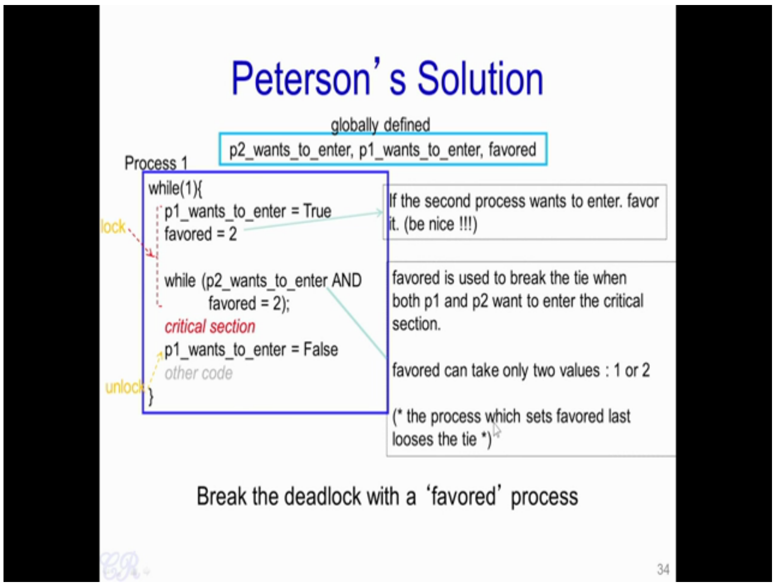
```
while(1){
    p2_wants_to_enter = True
    while(p1_wants_to_enter = True);
    critical section
    p2_wants_to_enter = False
    other code
}
```

- Deadlock
  Have a way to break the deadlock

33

Let us see what the problem was with the third attempt. Essentially we had the process setting it is flag and waiting for the other processes flag to be set to false. And we have seen that because both processors may enter into the while loop and wait for each other, we would have a deadlock situation. Now the next best thing to do is whenever we have such a deadlock situation, we need to find a way to actually come out of the deadlock situation. Essentially we need to ensure that one and exactly one of these two processes will break from the while loop and enter into the critical section.

(Refer Slide Time: 26:03)



Peterson’s Solution

globally defined

p2_wants_to_enter, p1_wants_to_enter, favored

Process 1

```
while(1){
  p1_wants_to_enter = True
  favored = 2

  while (p2_wants_to_enter AND
         favored = 2);
  critical section
  p1_wants_to_enter = False
  other code
}
```

If the second process wants to enter. favor it. (be nice !!!)

favored is used to break the tie when both p1 and p2 want to enter the critical section.

favored can take only two values : 1 or 2

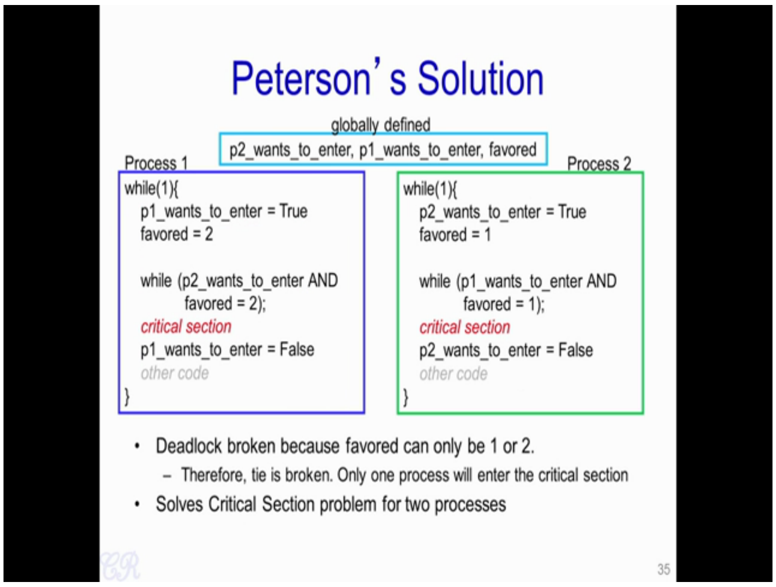
(* the process which sets favored last looses the tie *)

Break the deadlock with a ‘favored’ process

So, this solution was made by Peterson, and it is what is known as a Peterson’s solution, and essentially as other variable known as favored, so this is a globally defined variable which is shared among the two processes. Essentially, what the favored variable does is that whenever deadlock situation occurs, it favors one of the processes, and that process would then enter into the critical section. While the other process will continue to wait until the second process sets the - it is flag to false.

So, essentially Peterson’s solution sets the value of favor to the other process so that it favors the other process to enter into the critical section. Now favored it is used to break the tie when both p 1 and p 2 want to enter into the critical section that is when both p 1 and p 2 are waiting in this while loop or other are executing in this while loop. The reason this particular solution works is that out of the two processes, there is exactly two values that favored can take you can take only one or two, and therefore only one of these while loops will actually break or complete execution.

(Refer Slide Time: 27:31)



Peterson's Solution

globally defined

p2_wants_to_enter, p1_wants_to_enter, favored

Process 1

```
while(1){
   p1_wants_to_enter = True
   favored = 2

   while (p2_wants_to_enter AND
          favored = 2);
   critical section
   p1_wants_to_enter = False
   other code
}
```

Process 2

```
while(1){
   p2_wants_to_enter = True
   favored = 1

   while (p1_wants_to_enter AND
          favored = 1);
   critical section
   p2_wants_to_enter = False
   other code
}
```

- Deadlock broken because favored can only be 1 or 2.
  - Therefore, tie is broken. Only one process will enter the critical section
- Solves Critical Section problem for two processes

35

Let us see this in more detail. So, we have seen the two processors p 1 and p 2, and everything is as the third attempt that we want that p 1 wants to enter set to true and then there is a while loop and then critical section and then p 1 want to enter set to false. Now we also have a favored added here in the Peterson solution and this favored is set to two over here and favored is set to one in the second process. So, in all other cases this particular Peterson solution behaves like our third attempt to create the critical section solution. However the only difference comes when both processors enter into this while loop at the same time. In such a case, the value of favored will determine which of the two processes enter into the critical section.

Let us say process 2 is favored and it enters into the critical section, and at the end of it is critical section it sets p 2 wants to enter to false. So, remember that during this entire period p 1 is still waiting or still executing in the while loop. Now when process 2 sets it is flag to false, it immediately causes this while loop to break, as a result process 1 will enter into the critical section. So, you see that only when process 2 exits from the critical section only then will process 1 enter into the critical section, thus achieving mutual exclusion. Also, we had seen that the bounded weight issue is solved and as well as the progress thing is solved. The Peterson solution works very efficiently for two processes.

In the next video, we will see the bakery algorithm which is used to solve the critical section problem when there are multiple processors in the system. So, the bakery algorithm solves the critical sectional problem when the number of processes is greater than two.

Thank you.