

**Real Time System**  
**Professor Rajib Mall**  
**Department of Computer Science and Engineering**  
**Indian Institute of Technology, Kharagpur**  
**Lecture: 27**  
**Basic priority inheritance protocol (PIP)**

Welcome to this lecture. In the last lecture, we had discussed about the problems that can occur due to resource sharing in a real time system. There are two problems one is a simple priority inversion and the other is unbounded priority inversion. The simple priority inversion is not a very vexing problem, because we can overcome it through careful programming and the effect of simple priority inversion can be minimized.

But unbounded priority inversion cannot be solved through careful programming, it can still make the tasks to miss their deadline with the most careful programming, we need special mechanisms to be supported by the operating systems to be able to overcome the unbounded priority inversion problem. We had seen what exactly the unbounded priority inversion problem is. Here a low priority task is holding the resource and therefore a high priority task needing the resource is blocking.

But in between the intermediate priority tasks which do not need the resource this start executing and the low priority task cannot make progress with its computation. As a result, the high priority task undergoes blocking not only due to the low priority task which is holding the resource, but due to the other intermediate priority tasks, which do not need the resource.

Now, let us look at the mechanisms that exist to handle the unbounded priority inversion problem. We will understand the mechanisms and also their advantages and disadvantages and the situations in which it is useful.

(Refer Slide Time: 2:15)

**Basic Priority Inheritance Protocol**  
Sha and Rajkumar 1990

- Main idea behind this scheme:
  - Since a task in a critical section cannot be preempted.
  - It should be allowed to complete as early as possible.

The simplest priority the simplest mechanism for unbounded priority inversion is the basic priority inheritance protocol. Very simple mechanism proposed by Sha and Rajkumar in 1990. The main idea behind this scheme is very simple. The idea is that a low priority task executing in its critical section cannot be preempted from resource usage, since it cannot be preempted, it has to use the resource.

So, the only thing that you can do is let it complete as early as possible so that the waiting time or the priority inversion time for the high priority tasks can be at least minimized.

(Refer Slide Time: 3:18)

**Basic Priority Inheritance Protocol**  
Sha and Rajkumar 1990

- How do you make a task complete as early as possible?
  - Raise its priority, so that low priority tasks are not able to preempt it.
- By how much should its priority be raised?
  - Make its priority as much as that of the task it is blocking.

But how do we make the low priority tasks to complete its usage of the resource as early as possible we raise its priority. So, that the intermediate priority tasks which are preempting the low priority task and delaying its execution as a result unbounded priority inversions were occurring they will not occur.

Because we rose the priority of the low priority task holding the resource to very high value equal to the priority of the task that is waiting for it. So basically the low priority task inherits the priority of the high priority tasks that is waiting for it. And therefore, the intermediate priority task cannot preempt it and the low priority task completes its execution without getting interrupted. By how much the priority of the low priority task needs to be raised.

It should not be made the maximum priority because in that case, other tasks, high priority task may miss their deadline. It should be made as much as that of the task that is blocking. Since the low priority task gets the priority value equal to the tasks that are waiting, we say that the low priority task inherits the priority of the task that is waiting for the resource.

(Refer Slide Time: 5:13)

**Basic Priority Inheritance Protocol**  
Sha and Rajkumar 1990 (PIP)

- When a resource is already under use:
  - Requests to lock the resource by different tasks are queued in FIFO order.
  - Inheritance clause applied each time after a higher priority task blocks.

The diagram illustrates the Basic Priority Inheritance Protocol. It shows a resource R (represented by a green circle with 'R') being held by a low-priority task  $T_L$  (represented by a red circle with 'T<sub>L</sub>'). A high-priority task  $T_H$  (represented by a green circle with 'T<sub>H</sub>') is blocked for the resource. Another high-priority task  $T_H'$  (represented by a red circle with 'T<sub>H</sub>'') is also blocked for the resource. The diagram shows  $T_L$  inheriting the priority of  $T_H'$ . Handwritten notes in red ink show  $Pri(T_L) \leftarrow Pri(T_H')$  and  $Pri(T_H)$  with arrows indicating the priority inheritance.

Now, it may so happen that a task a low priority task is holding a resource and a high priority task became ready, started executing and then waited for the resource because it needed that resource R. But, there can be another. So, in the priority inheritance protocol  $T_L$  will inherit the priority of  $T_H$  or we can say priority of  $T_L$  it inherits the priority of  $T_H$ .

But it may so, happen that another very high priority task let us say  $T_H'$  whose priority is even more than  $T_H$  started executing and it also needed the same resource and it started blocking. So, what will be the priority of  $T_L$ ?  $T_L$  will inherit the priority of all the tasks that are waiting for the resource. So, in this case, since priority of  $T_H'$  is more than  $T_H$  the priority of  $T_L$  will be raised to priority of  $T_H'$ .

There may be another intermediate priority task which may block for the resource, but in this case the  $T_L$  priority will be unaffected it will still be operating at the priority of  $T_H'$ . And all these tasks are maintained in a FIFO queue. And the priority of these FIFO queue is scanned the priority of all the tasks and the highest among these is assigned to the  $T_L$ .

The inheritance clause is applied each time after a higher priority task blocks for the resource. So, that is the basic priority inheritance protocol of Sha and Rajkumar in 1990. We will refer it as

PIP Priority Inheritance Protocol. And the priority of the task  $T_L$  is raised, but what about, does it operate that same priority even after it releases the resource no.

As soon as it releases the resource, it gets back to its own priority. If it is holding no other resource, if it is holding another resource  $R'$  it may hold multiple resources. And if it is holding another resource  $R'$  and it released  $R$  then the priority of  $T_L$  will be the maximum priority of tasks waiting for  $R'$ .

So, that is the mechanism by which the priority of the  $T_L$  is adjusted. As it, other tasks block further resource the priority is raised. And as  $T_L$  completes executing using the critical resource and releases the resource, it gets back its own priority. The inheritance clause is applied each time after a high priority task blocks.

(Refer Slide Time: 09:27)

### Inheritance Clause

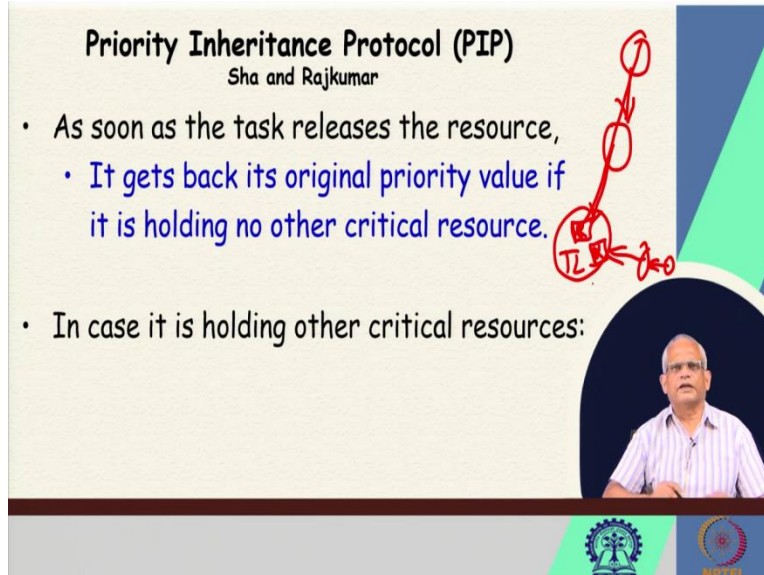
- The priority of the task in the critical section:
  - Raised to equal the highest priority task in the queue.

The diagram shows a task  $T_M$  (represented by a blue circle) holding a resource  $R$  (represented by a yellow box). This resource  $R$  is being used by task  $T_L$  (represented by a green circle). A handwritten red arrow points from  $T_M$  to  $T_L$ , and a handwritten red equation  $Pri(T_L) \leftarrow Pri(T_M)$  indicates that the priority of  $T_L$  is raised to match the priority of  $T_M$ .

### Priority Inheritance Protocol (PIP)

Sha and Rajkumar

- As soon as the task releases the resource,
  - It gets back its original priority value if it is holding no other critical resource.
- In case it is holding other critical resources:



The inheritance clause is that whenever a task waits, the priority of  $T_L$  temporarily is assigned the priority of  $T_H$  or  $T_M$  whatever. And as soon as the task releases the resource it gets back its original priority value, if it is holding no other critical resource. But if it is, if it was holding just one resource  $R$ ,  $T_L$  was holding one resource  $R$  and it release the resource it will get back its own priority.

But it was if it is holding another resource  $R'$ , then there may be some tasks which are waiting for  $R$  and there were some waiting for  $R'$ . And it releases the resource  $R$ . So, the priority of  $T_L$  will be maximum of the tasks waiting for  $R'$ .

(Refer Slide Time: 10:52)

### Inheritance Blocking

- How does PIP prevent unbounded priority inversions?
- Priority of low priority task raised to high value...
- Intermediate priority tasks can no longer preempt it.

(PIP)

Now, does priority inheritance protocol PIP? Does it prevent unbounded priority inversions? Yes, it prevents unbounded priority inversion because the high priority task will undergo inversion due to at most one task for a resource. Because the low priority tasks priorities rose intermediate priorities tasks cannot preempt it.

And therefore, it undergoes inversion due to a single task that is a low priority task and it the unbounded priority inversion problem becomes a simple priority inversion problem. And we know that simple priority inversion problem is not that big a problem, we can overcome using careful programming. So, the intermediate priority tasks they cannot preempt  $T_L$  and therefore  $T_L$  continues to execute and complete its usage of resource  $R$ .

And then  $T_H$  can acquire the resource. So, the unbounded priority problem is effectively solved by the simple priority inheritance mechanism. And this is what we are referring when we said that the inheritance mechanism was done on and off in the Mars Pathfinder, this is the inheritance mechanism priority, inheritance mechanism.

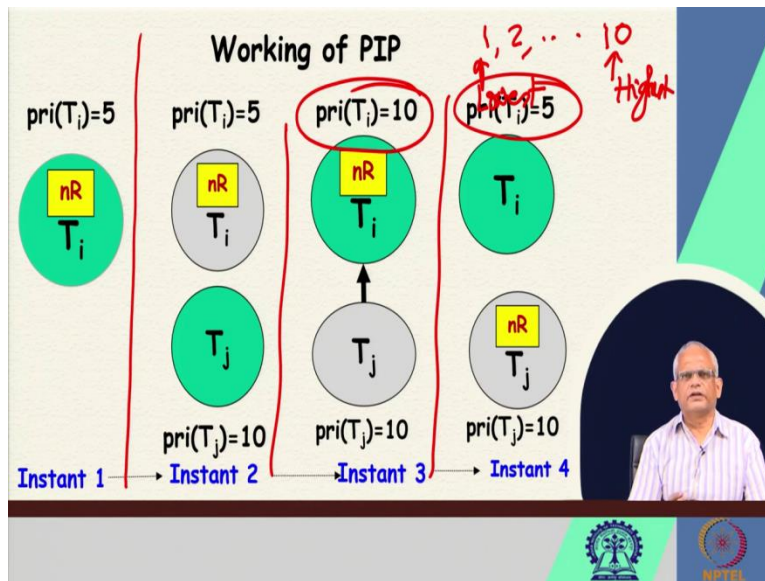
The simplest solution prevents unbounded priority inversion problem, but if we analyze it further, we will find that it has a few problems and only in extremely simple applications. We



can use the priority inheritance principle or PIP the Priority Inheritance Protocol for only very simple embedded real time systems.

Let us understand what the problems are and what are the improvements on the priority inheritance protocol to handle those problems which is suffered by PIP.

(Refer Slide Time: 13:25)



Before that, let us understand the working of the PIP through a simple example. Let us say in a real time system, the tasks are scheduled using a rate monotonic scheduler and 1 is the highest priority, 2 is the second highest priority and so on and 10 is the lowest priority at some instant of time. A task  $T_i$  was holding a non-preemptable resource in  $R$  and the priority of  $T_i$  is 5.

Now after some time that is instant 2. A task  $T_j$  which started to execute and priority is 10. And, so here 10 is the highest priority and 1 is the lowest priority in this system. So,  $T_j$  is a high priority task, and it started it became ready and started executing. And it needed the critical resource  $nR$  at instant 2.

So, it started executing, and after some time, it needed the resource  $nR$ . Now as it needed the resource  $nR$ , it blocked for the resource. But as soon as it blocked for the resource, the priority of  $T_i$  is has been raised to 10. It has inherited the priority of  $T_j$  and after some time, it released the resource the  $nR$  completed execution and release the  $nR$ .



And as soon as it released nR, it got back its own priority 5 and nR is acquired by  $T_j$ , which has the priority 10. So, this example gives the understanding that when does the priority of a low priority task increase and the instant at which it is, it gets back its original priority value.

(Refer Slide Time: 16:22)

The slide is titled "Shortcomings of the Basic Priority Inheritance Scheme". It lists two main drawbacks of PIP:

- PIP suffers from two important drawbacks:
  - Susceptible to chain blocking. AP
  - Does nothing to prevent deadlocks.

The slide features a speaker in a circular inset at the bottom right and logos for institutions at the bottom. The text "chain blocking" and "AP" are handwritten in red ink, with "chain blocking" underlined and "AP" written to its right.

Now let us understand what the problems with this basic priority inheritance scheme are. There are two major problems one goes by the name chain blocking. Chain blocking can occur in the priority inheritance scheme, the simple PIP chain blocking can occur and chain blocking can also cause a high priority task to miss its deadline.

The second problem is that deadlocks can occur here. This mechanism of priority inheritance does not do anything to prevent the deadlock so deadlocks can occur in a priority simple priority inheritance protocol system. Let us understand these two how they develop in a real time system.

(Refer Slide Time: 17:27)

**Deadlocks**

- Consider two tasks T1 and T2 accessing critical resources CR1 and CR2.
- Assume:
  - T1 has a higher priority than T2
  - T2 starts running first

T1: Lock R1, Lock R2, ...Unlock R2, Unlock R1

T2: Lock R2, ... Lock R1, ...Unlock R1, Unlock R2

First, let us look at deadlock. Let us assume that T1 and T2 are two tasks, which need the resources CR1 and CR2, both need T1 needs CR1 and CR2 and T2 also needs the resources CR1 and CR2. And T1 has higher priority than T2. But then T2 starts running first and executed lock R2 and after some time T1 started running and therefore T2 was preempted.

So, is T1 started running it executed lock R1. So, it acquired R1 and after some time it wanted to lock R2. But R2 is already held by T2 and in this priority inheritance protocol. T2's priority will be raised to T1 here at this instant when it tried to lock R2 and it blocked for R2 because R2 is being held by T2. So, T2's priority is increased and it starts executing, but after some time it wants to use R1.

But R1 is already being held here by T1 and T2 waits for T1 to release R1 and T1 waits for T2 to release R2 and therefore there is a deadlock situation and the two tasks can easily miss their deadline. So, through this example we can see that there are many ways in which deadlock can occur in a simple priority inheritance protocol system.

Even though there is a priority inheritance mechanism supported by the system, still deadlocks can occur and tasks can miss their deadline, if we can somehow develop a resource sharing protocol, which also makes it impossible for deadlock to occur, then that will be really nice.

Because we are sure that we are using the protocol which is safe from deadlocks, you do not have to worry because if deadlocks occur, tasks are going to miss their deadlines.

(Refer Slide Time: 20:42)

**Chain Blocking**

- A task needing to use a set of resources undergoes chain blocking, if :
  - Each time it needs a resource, it undergoes priority inversion.
- Example:
  - Assume a high-priority task T1 needs several resources

Handwritten notes on the slide:

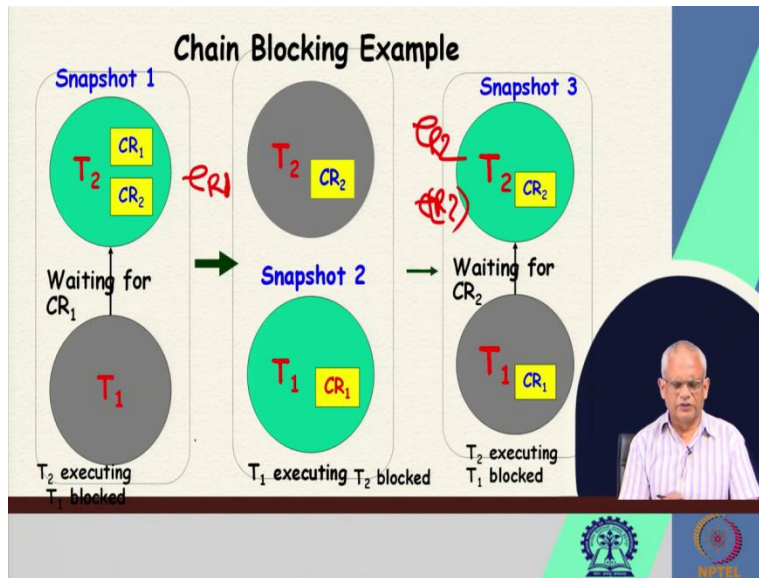
$$T_1 = \{R_1, R_2, R_3, \dots, R_n\}$$
$$e_{R_1} + e_{R_2} + \dots$$

The slide also features a video inset of a man speaking and logos for IIT Bombay and MIT at the bottom.

Now, let us understand the chain blocking problem. A high priority task which needs to use a set of resources can undergo chain blocking, let us say a task  $T_1$  needs a set of resources  $R_1, R_2, R_3, R_n$  it needs some 4, 5 resources. Now it can undergo chain blocking, we say that it undergoes chain blocking, if each time it needs to access a resource, it undergoes priority inversion. A simple priority inversion and we know that the maximum time for which it can undergo a simple priority inversion is  $e_r$ .

$e_r$  is the time of usage of the resource are let us say for  $R_1$  it will be  $e_{r1}$ . So, the low priority task, the maximum time it usage the resource  $R_1$ . And similarly, when it needs  $R_2$ , again it will undergo blocking for  $e_{r2}$  and so on. So, if it needs five resources, it will be five additions here, which can become a large number, large enough duration for  $T_1$  to miss its deadline.

(Refer Slide Time: 22:38)

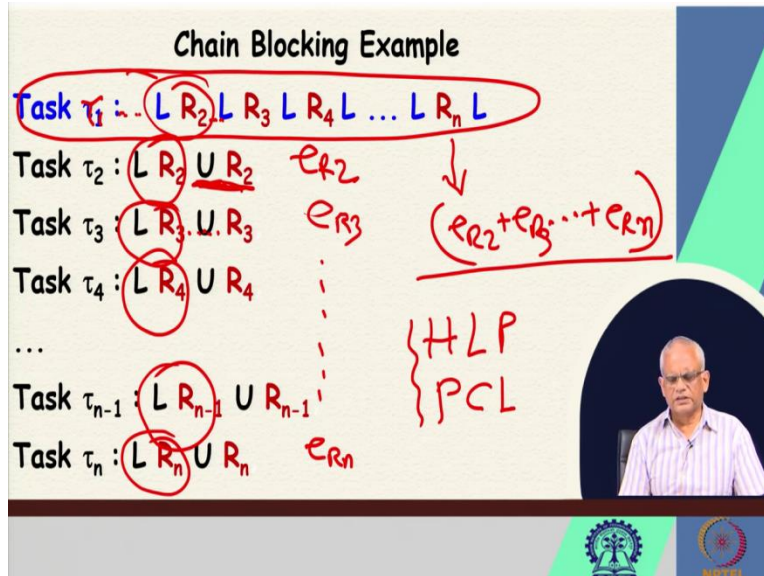


Now, let us try to understand with one example. So, here T<sub>1</sub> is a higher priority task and T<sub>2</sub> is holding two resources CR<sub>1</sub> and CR<sub>2</sub> and T<sub>1</sub> is initially waiting for CR<sub>1</sub>. Now, after some time T<sub>2</sub> started executing and then T<sub>1</sub> was waiting and T<sub>2</sub> executed for  $e_{r1}$ ,  $e_{r1}$  time for each it needed CR<sub>1</sub> it executed for  $e_{r1}$  time and then released CR<sub>1</sub>.

Now CR<sub>1</sub> is acquired by T<sub>1</sub> and T<sub>1</sub> started executing. But after some time T<sub>1</sub> needed CR<sub>2</sub>, started blocking for CR<sub>2</sub> at that time T<sub>2</sub> will again start executing and it will execute for  $e_{r2}$ . So, for the two resources the task T<sub>1</sub> occurs blocking times  $e_{r1}$  and  $e_{r2}$  and if it needs n resources, then we can easily visualize and it may not be held by one task actually.

Here in this example, we have shown that the resource is being held by T<sub>2</sub> all the resources that T<sub>1</sub> requires. It may not be the case it may be that the resource is held by different tasks T<sub>2</sub>, T<sub>3</sub>, T<sub>4</sub>, T<sub>n</sub>, etc. And each time T<sub>1</sub> will block and the total blocking time or the total priority inversion time for T<sub>1</sub> due to chain blocking will become  $e_{r1} + e_{r2} + \dots + e_{rn}$  which can be a large number for T<sub>1</sub> to miss its deadline.

(Refer Slide Time: 25:01)



Now, let us understand the chain blocking example with this, the chain blocking with this example. We have a high priority task  $t_1$  and there are low priority tasks  $t_2, t_3, t_n$ , etc. Now,  $t_2$  initially locked  $R_2$  and  $t_3$  started executing locked  $R_3$ ,  $t_4$  locked  $R_4$  and so on and  $t_{n-1}$  locked resource  $n-1$  and  $t_n$  locked resource  $R_n$ .

But then the high priority task  $t_1$  it got ready after all the resources have been locked. Now, the task  $t_1$  needed  $R_2$  after executing some time needed  $R_2$ . But  $R_2$  is already being held by a task  $t_2$ . So,  $t_1$  will block.  $t_2$  will inherit the priority of  $t_1$  and it will start executing until after some time it unlocks  $R_2$  and then the time for which it executes the task  $t_2$  is  $e_{R_2}$  and then task  $t_1$  starts executing and after some time needs  $R_3$ , lock  $R_3$ .

But  $R_3$  has been already locked by task 3. So, this starts executing, inherits the priority of  $t_1$  starts executing until it does unlock  $R_3$ . So, it executes for let us say  $e_{R_3}$  and so on  $e_n$ . So, the tasks  $t_1$ ; the total priority inversion period or the blocking period is  $e_{R_2} + e_{R_3} + \dots + e_{R_n}$ . If there is a simple priority inversion like  $e_{r_1}$  or something that can be minimized through careful programming, but if it is  $n$  times that a task can miss its deadline.

So, the chain blocking problem where a high priority task each time it needs a resource, needs to wait for a task to release the resource is a very big problem here in the priority inheritance protocol. We will discuss some refined protocols namely the highest locker protocol and the priority ceiling protocol which overcome chain blocking and deadlock in this both these protocols deadlock is not possible and chain blocking is not possible unbounded priority inversion is not possible.

So, but these protocols HLP and priority ceiling protocol are slightly more complicated than the basic priority inheritance protocol. So, unless our system is very primitive, very simple system will go for HLP or PCL. But if our system is very simple, we do not want any complicated mechanisms. We would use the basic priority inheritance protocol for supporting resource sharing among real time tasks. We are at the end of this lecture.

We will stop here and continue from this point discussing about improvements of the priority inheritance protocol in the next lecture. Thank you