Semaphore

In this tutorial, we will be discussing a Semaphore and its types ie. Binary, Mutex, Counting.
Later we will see each semaphore in detail with its pros and cons. Also, we will be looking into priority inversion and priority inheritance.

Contents

- 1 Teminologies
- 2 What is a Semaphore?
- 3 Types of Semaphores
- 4 Priority Inversion
- 5 Priority Inheritance

Teminologies

1. **LPT:** Low Priority Task
2. **MPT:** Medium Priority Task
3. **HPT:** High Priority Task

What is a Semaphore?

Semaphore is a technique for synchronizing two/more task competing for the same resources. When a task wants to use a resource, it requests for the semaphore and will be allocated if the semaphore is available. If the semaphore is not available then the requesting task will go to blocked state till the semaphore becomes free.

Consider a situation where there are two persons who want to share a bike. At one time only one person can use the bike. The one who has the bike key will get the chance to use it. And when this person gives the key to the 2nd person, then the 2nd person can use the bike.

Semaphore is just like this **Key** and the bike is the shared resource. Whenever a task wants access to the shared resource, it must acquire the semaphore first. The task should release the semaphore after it is done with the shared resource. Until this time all other tasks have to wait if they need access to shared resource as semaphore is not available. Even if the task trying to acquire the semaphore is of higher priority than the task acquiring the semaphore, it will be in the wait state until the semaphore is released by the lower priority task.

Types of Semaphores

There are 3-types of semaphores namely Binary, Counting and Mutex semaphore.

- **Binary Semaphore**: Binary semaphore is used when there is only one shared resource.

Binary semaphore exists in two states ie.Acquired(Take), Released(Give). Binary semaphores have no ownership and can be released by any task or ISR regardless of who performed the last take operation. Because of this binary semaphores are often used to synchronize tasks with external events implemented as ISRs, for example waiting for a packet from a network or waiting for a button is pressed.

Because there is no ownership concept a binary semaphore object can be created to be either in the ?taken? or ?not taken? state initially.

Cons:

1. Priority Inversion : HPT needs to wait for the LPT as LPT is holding the resource required for HPT. In between if MPT task comes then LPT will be in blocked state thereby delaying HPT.
2. Does not support recursion : If a task tries to take the semaphore twice then it gets blocked and will not come out of that state till someone releases that semaphore.
3. No ownership : Anyone can release or delete the semaphore because of which the dependent tasks will always be in the blocked state.

- **Counting Semaphore**: To handle more than one shared resource of the same type, counting semaphore is used. Counting semaphore will be initialized with the count(N) and it will allocate the resource as long as count becomes zero after which the requesting task will enter blocked state.

- **Mutex Semaphore**: Mutex is very much similar to binary semaphore and takes care of priority inversion, ownership, and recursion.

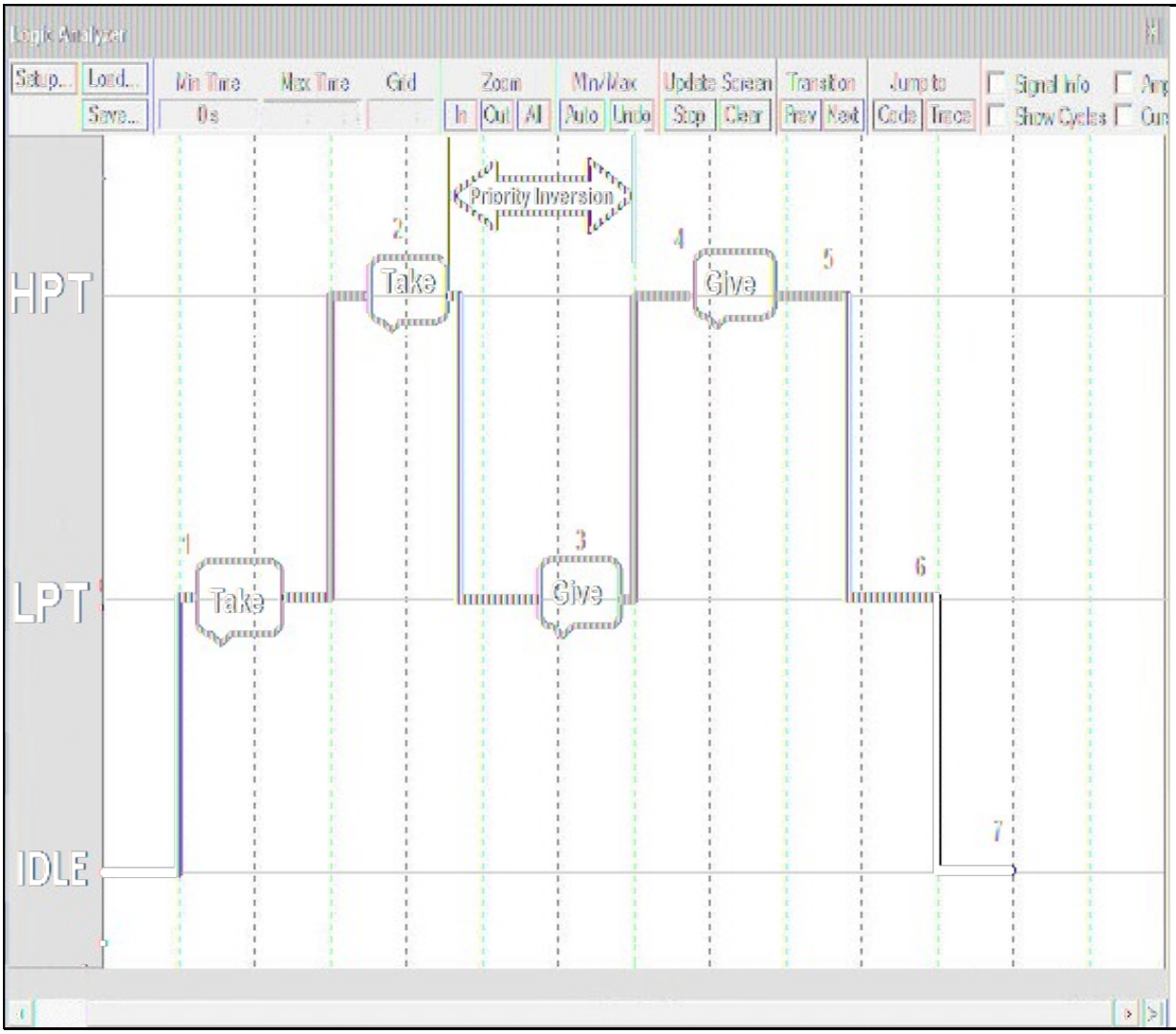
Priority Inversion

This is a scenario where the HPT waits for the LPT as it is using the resource required by HPT. Let's see 3 different scenarios of priority inversion depending on the amount of time HPT waits for the lower priority tasks.

- **Normal Priority Inversion:** Here an HPT tasks waits for the LPT tasks as it is holding the resource(semaphore).

1. LPT starts running and acquires the semaphore.
2. Now HPT is created and it preempts LPT and starts running. It makes the request to acquire the semaphore. Since the semaphore is already with LPT, HPT goes to blocked state.
3. LPT starts executing again and releases the semaphore.
4. Immediately the HPT comes out of the blocked state and starts executing.
5. It releases the semaphore and runs for some time and deletes itself.
6. Now control goes back to LPT which completes its job and deletes itself.
7. Finally the scheduler is left out with the idle task and it keeps running.

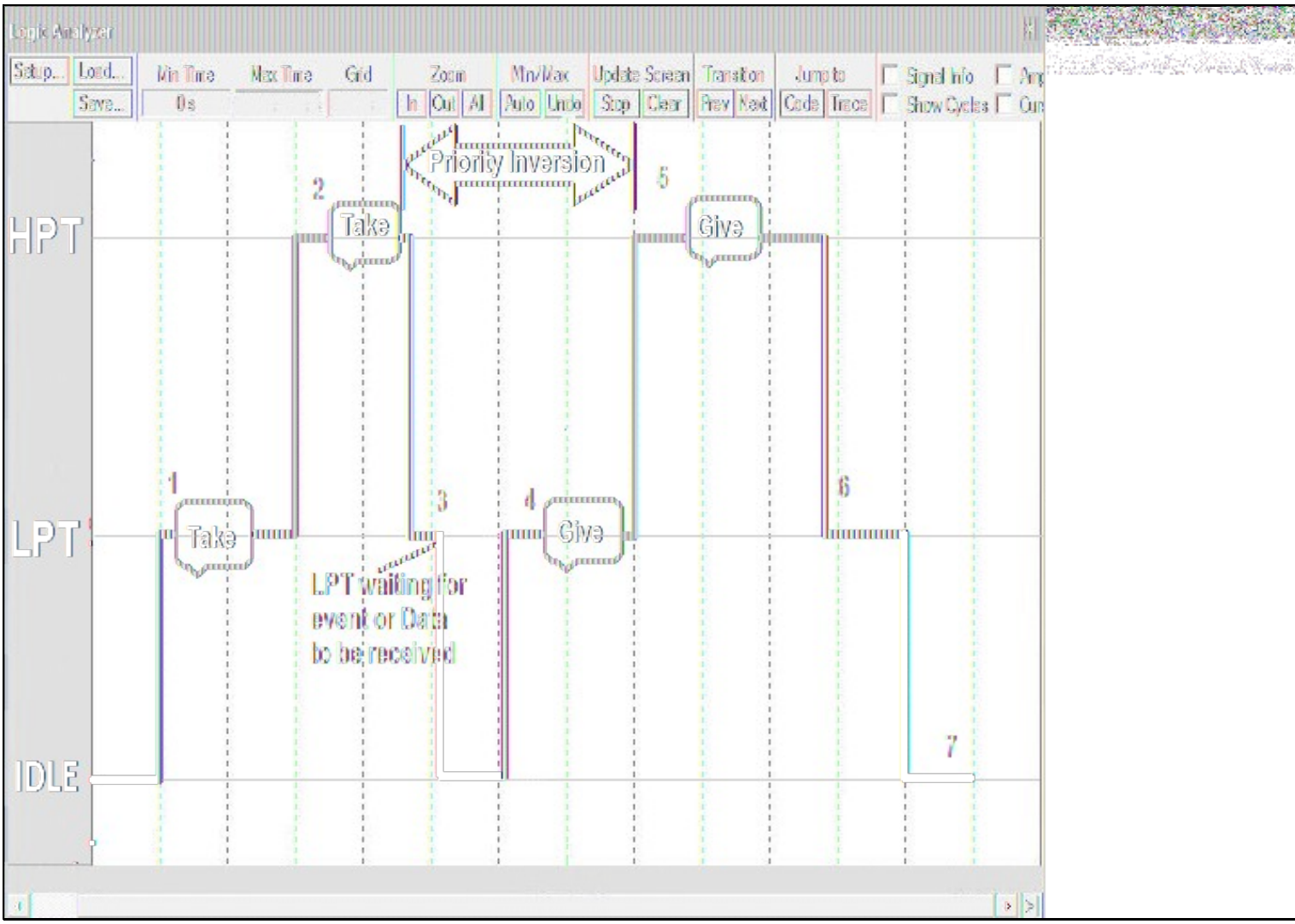
In this scenario, the HPT task waits for LPT from 2-4 which is the priority inversion period.



- **Extended Priority Inversion:**Here an HPT tasks waits for the LPT tasks as it is holding the resource(semaphore). Further, the LPT waits for the event/data to be received and the CPU executes the IDLE tasks. This makes the HPT wait for more time.

1. LPT starts running and acquires the semaphore.
2. Now HPT is created and it preempts LPT and starts running. It makes the request to acquire the semaphore. Since the semaphore is already with LPT, HPT goes to blocked state.
3. LPT starts executing again and waits for an event(packet to be received).
4. Now the Idle task starts running. Now the HPT is starved because of LPT and Idle task.
5. LPT comes out of blocked state as the event(wait time/packet is received) has occurred. Now it releases the semaphore.
6. Immediately the HPT comes out of the blocked state and starts executing. It releases the semaphore and runs for some time and deletes itself.
7. Now control goes back to LPT which completes its job and deletes itself.
8. Finally the scheduler is left out with the idle task and it keeps running.

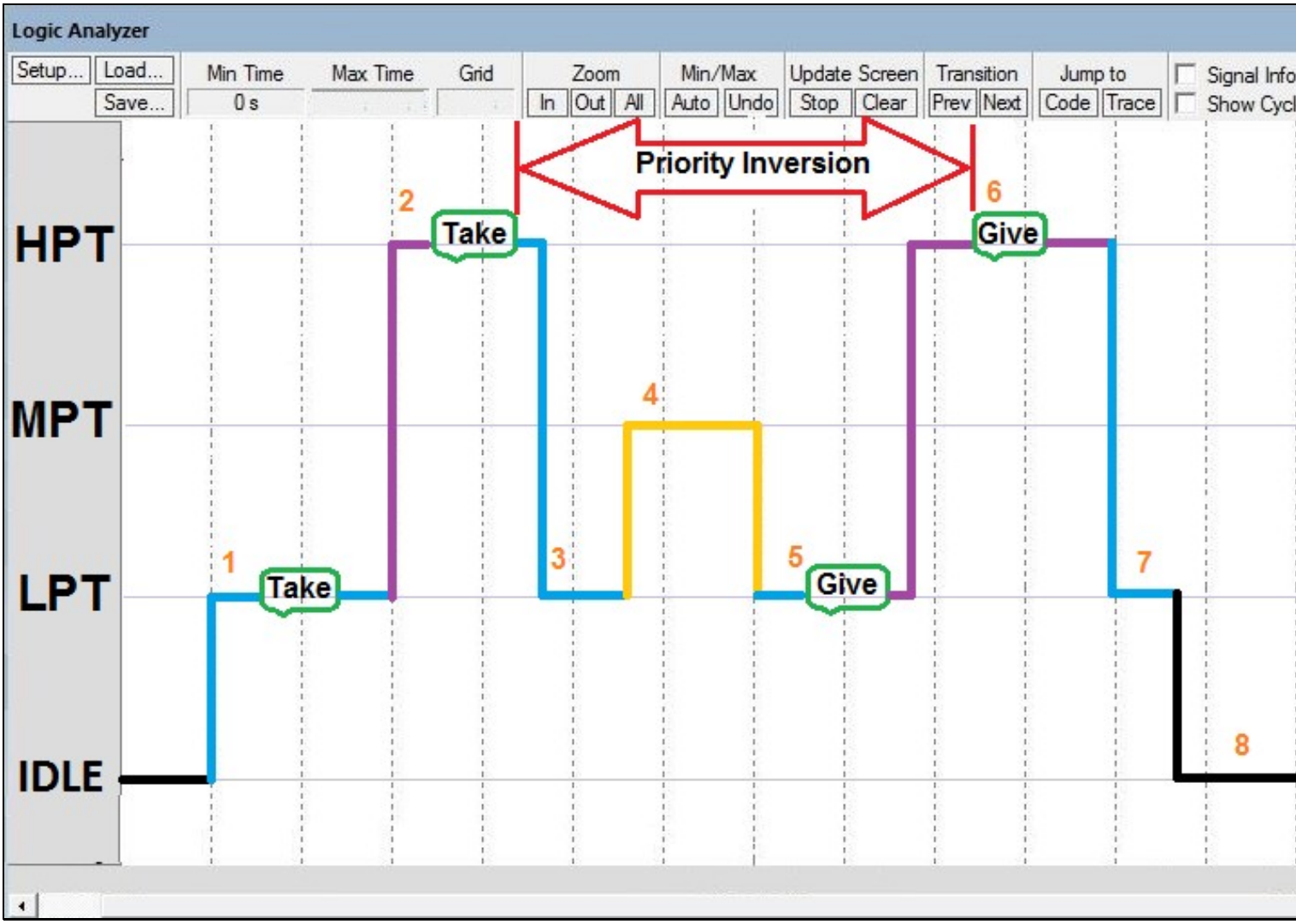
In this scenario, the HPT task waits for LPT and Idle task from 2-5 which is the extended priority inversion period.



- **Worst Case Priority Inversion:**Here an HPT tasks waits for the LPT tasks as it is holding the resource(semaphore). In between an MPT starts executing thereby blocking the LPT and HPT. This makes the HPT wait for MPT and LPT.

1. LPT starts running and acquires the semaphore.
2. Now HPT is created and it preempts LPT and starts running. It makes the request to acquire the semaphore. Since the semaphore is already with LPT, HPT goes to blocked state.
3. LPT starts executing again and creates an MPT tasks.
4. MPT preempts the LPT and starts running. At this point, the HPT is starved because of LPT and MPT. MPT completes its job and deletes itself.
5. LPT comes out of blocked state and releases the semaphore.
6. Immediately the HPT comes out of the blocked state and starts executing. It releases the semaphore and runs for some time and deletes itself.
7. Now control goes back to LPT which completes its job and deletes itself.
8. Finally the scheduler is left out with the idle task and it keeps running.

In this scenario, the HPT task waits for LPT and MPT from 2-6 which is the extended priority inversion period.



Priority Inheritance

As discussed in the above scenarios, the HPT has to wait for the LPT and also MPT. The scenario of HPT waiting for LPT is justified. But MPT is coming in between and blocking both LPT and HPT. MPT is not at all sharing the same resources and making the HPT wait unnecessarily.

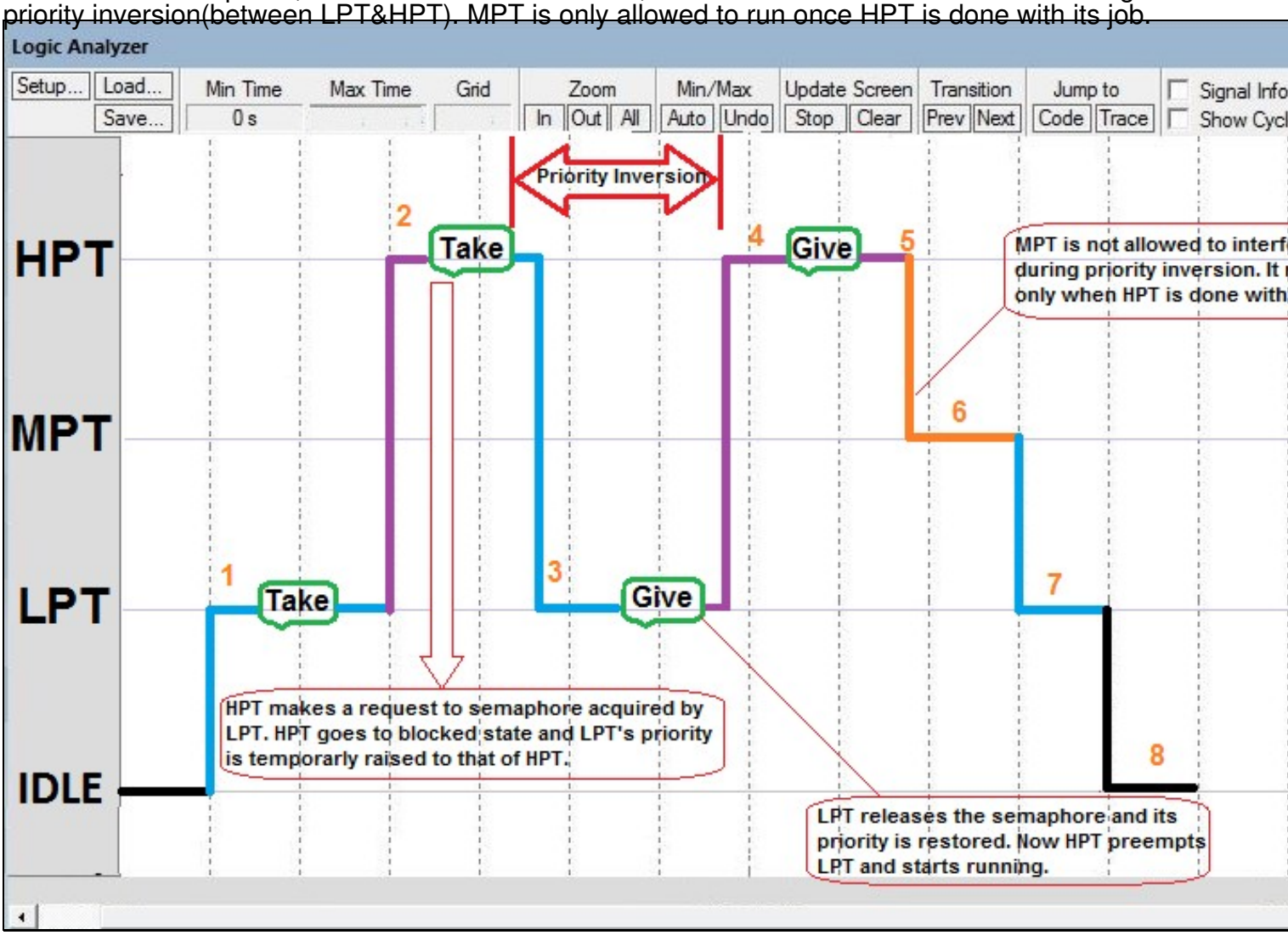
Above problem can be resolved using the mutex semaphore. As soon as the HPT makes a request to semaphore which is allocated to an LPT then, the priority of LPT is temporarily increased to that of HPT. Once the LPT releases the semaphore, its priority will be restored. Because of this, the MPT is never allowed to interfere during the priority inversion(between LPT&HPT). MPT is only allowed to run once HPT is done with its job.

In the below scenario:

1. LPT starts running and acquires the semaphore.
2. Now HPT is created and it preempts LPT and starts running. It makes the request to acquire the semaphore. Since the semaphore is already with LPT, HPT goes to blocked state and LPT's priority is temporarily raised to that of HPT.
3. LPT starts executing again and creates an MPT tasks. It keeps running as its priority is currently higher than MPT. It releases the semaphore and its priority is restored back to lower value.
4. Immediately the HPT comes out of the blocked state and starts executing.
5. HPT releases the semaphore and runs for some time and deletes itself.

6. Now scheduler is left with LPT,MPT, and IDLE task. MPT starts running and deletes itself after completing its job.
7. Now control goes back to LPT which completes its job and deletes itself.
8. Finally the scheduler is left out with the idle task and it keeps running.

In this scenario, the HPT only waits for LPT as long as it has the semaphore. As soon as LPT releases the semaphore, HPT takes the control. Also, MPT is never allowed to interfere during the priority inversion(between LPT&HPT). MPT is only allowed to run once HPT is done with its job.



Have an opinion, suggestion , question or feedback about the article let it out here!

Please enable JavaScript to view the comments powered by Disqus.