Handling of Priority Inversion Problem in RT-Linux using Priority Ceiling Protocol

Silambarasan D[1], Ramanatha Venkatesan M[2]

[1]Assitant Professor, Department of EEE, AURC- CBE University, Coimbatore, Tamil Nadu
[2]Department of EEE, AURC- CBE University, Coimbatore, Tamil Nadu

Abstract— *Real time system which runs multiple task concurrently or pseudo concurrently shares the resources will face priority inversion phenomenon. This priority inversion phenomenon will reduce the Real Time System predictability which in turn leads to un-predictable error. Continuous Priority Inversion phenomenon will lead the Real Time System to collapse. This paper analyses the cause and effect of the priority inversion phenomenon. Further analyze the various solution and implementation in RT-Linux. This paper improves the already existing priority inheritance protocol and priority ceiling protocol. The proposed algorithm will prevent the Real Time System from deadlock related to priority inversion prevention protocol. Experimental results and analysis in theoretical way indicate that the methods to solve the priority inversion problem in RT-Linux are effective and concise, provides reasonable technical details for the safe running of complex real-time application in RT-Linux.*

Keywords— Priority Inversion, Priority Reversal, Priority Ceiling, Real Time Systems, RT-Linux.

I. INTRODUCTION

Real Time System is a system where a timely response by the computer to external stimuli is vital [1]. The correctness of the response not only depends on logical value of the result but also the time at which response is given. The operating System used in such system should adhere with above definition and should strive to achieve this. Real Time Operating System has one of the important components in Real Time Systems to meet Real Time system's demands. Nowadays, various commercial Real Time Operating System are available off the shelf. But to reduce the cost of the Real Time Operating System, achieve the reliability of the Real Time Operating System and have high configurability of Real Time Operating System, Linux has been considered for having it in Real Time Systems. Embedded Linux has emerged as one such Operating System. Embedded Linux is an Open Source which paves the way to customize and fine tune the Operating System based on the embedded system. Embedded Linux supports wide range of hardware which eases the effort of porting it into new embedded products. So, Embedded Linux could get in Embedded Products easily. Embedded Linux has powerful community available in internet which helps to have good technical support. Embedded Linux has various real time capabilities which make it suitable for the Embedded Systems. Moreover, the Embedded Linux is available in free of cost in most scenarios. Linux has proven its existence in huge servers and in small handheld devices. Above advantages are applicable for RT-Linux. Linux could be made as hard real time Operating System in two ways. One is only to modify the Linux Kernel, the other is to add an abstract hardware layer, that is to say, to add a real-time kernel to have the real-time performance. First approach modifies the Linux and adopting POSIX 1.b standard. In order to reduce the time that Linux masks the interrupts, pre-emption points are inserted in Linux Kernel Code. This method improves the real-time performance of the Linux kernel. But, this method could not meet the hard real-time performance. The second method solute the problem and reaches the hard real-time performance [3].

RT-Linux comes under the category of the Embedded Linux. RT-Linux has relatively good real-time performance compared to its counterparts like RTAI, etc.,. RT-Linux is a hard real-time RTOS microkernel that runs the entire Linux operating system as a fully pre-emptive process. The hard real-time property makes it possible to control robots, data acquisition systems, manufacturing plants, and other time-sensitive instruments and machines from RT-Linux applications.

The key RT-Linux design objective was to add hard real-time capabilities to a commodity operating system to facilitate the development of complex control programs with both capabilities. For example, one might want to develop a real-time motor controller that used a commodity database and exported a web operator interface. Instead of attempting to build a single operating system that could support real-time and non-real-time capabilities, RT-Linux was designed to share a computing device between a real-time and non-real-time operating system so that (1) the real-time operating system could never be blocked from execution by the non-real-time operating system and (2) components running in the two

different environments could easily share data. As the name implies RT-Linux was originally designed to use Linux as the non-real-time system but it eventually evolved so that the RT-Core real-time kernel could run with either Linux or BSD UNIX.

Thus RT-Linux has become an interesting Software component which could be used for Real Time Computing in a Real Time System. But Real time Computing has many more challenges to meet like higher predictability, higher reliability, Should handle the widely varying computational loads [1].

Some of the challenges could be better handled by Real Time Operating System. They could be grouped in below headings Task assignment and scheduling, Communication protocols, Failure Management and recoveries [1]. In this paper one such issue related to Communication Protocols in RT-Linux is discussed.

The key RT-Linux design objective was to add hard real-time capabilities to a commodity operating system to facilitate the development of complex control programs with both capabilities. For example, one might want to develop a real-time motor controller that used a commodity database and exported a web operator interface. Instead of attempting to build a single operating system that could support real-time and non-real-time capabilities, RT-Linux was designed to share a computing device between a real-time and non-real-time operating system so that (1) the real-time operating system could never be blocked from execution by the non-real-time operating system and (2) components running in the two different environments could easily share data. As the name implies RTLinux was originally designed to use Linux as the non-real-time system but it eventually evolved so that the RTCore real-time kernel could run with either Linux or BSD UNIX.

Thus RT-Linux has become an interesting Software component which could be used for Real Time Computing in a Real Time System. But Real time Computing has many more challenges to meet like higher predictability, higher reliability, Should handle the widely varying computational loads[1].

Some of the challenges could be better handled by Real Time Operating System. They could be categorized as Task assignment and scheduling, Communication protocols, Failure Management and recoveries[1]. In this paper one such issue related to Communication Protocols in RT-Linux is discussed.

II. RT-LINUX ARCHITECTURE

RT-Linux provides the capability of running special real-time tasks and interrupt handlers on the same machine as standard Linux. These tasks and handlers are executed when they are needed to be executed no matter what Linux is executing. The worst case time between the moment a hardware interrupt is know by the processor and the moment an interrupt handler starts to execute its first instruction is under 15 microseconds on RT-Linux running on a generic x86. These times are hardware limited, and as hardware improves RT-Linux will also improve. Standard Linux has very good average performance and can even provide millisecond level scheduling precision for tasks using the POSIX soft real-time capabilities. However, Standard Linux is not designed to provide precision in the range of sub-millisecond and reliable timing guarantees. RT-Linux was based on a lightweight virtual machine where the Linux runs as guest Operating System and it was given a virtualized interrupt controller and virtualized timer, and all other hardware access was direct. For the real-time "host", the Linux kernel is a thread. Interrupts needed for deterministic processing are taken care by the real-time core, while other interrupts are sent to Linux kernel, which runs at a lower priority when compared to real-time threads. Linux kernel device drivers handle almost all operations related to I/O. First-In-First-Out pipes (FIFOs) or shared memory can be used to share data between the General purpose Linux operating system and real-time core RT-Linux.

RT-Linux built a virtual software layer. When Linux disables or enables interrupts, one variable of the virtual software layer is set. When one interrupt happens, RT-Linux decides if the interrupt should be handled by Linux or RT-Linux according to the value of the variable. In respect of memory allocation of the real-time task, RT-Linux allocates the memory for all real-time tasks as Linux kernel modules, so that all real-time tasks have same address space as the Linux kernel. It can lessen the difficulty of RT-Linux development to do like this. But Real Time task programmer has to design carefully real-time programs in case of any crash in Real Time task the whole system would be affected. In respect of task scheduling, the scheduler of RT-Linux bases on the priority scheduling. But the priority scheduling is not suitable for all real-time application, So RT-Linux has modularized the scheduler, therefore the user can use the schedulers based on various policies and algorithms. In respect of real-time clock, in order to realize the precise real-time trimming and avoid the task release jitter, RT-Linux adopts internal hardware timer chips as timer interrupt generator. In respect of inter process communication, RT-Linux provides the mechanism of semaphore message queue and especially for the needs of communication between real-time processes and none real-time processes. RT-Linux provides FIFO, and shared memory for inter task communication.

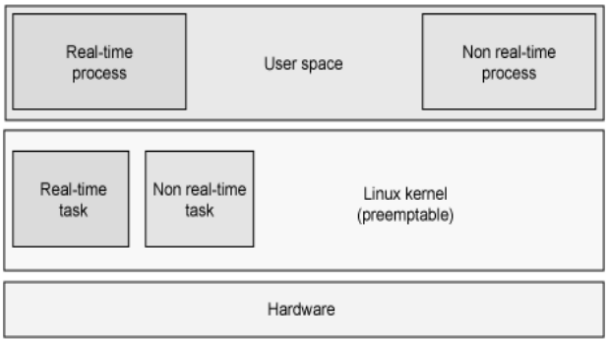

Fig.1: Architecture of RT-Linux

RT Linux kernel is module based kernel, the scheduler is itself a loadable kernel module. So, various scheduling policies like Earliest Deadline First(EDF), RateMonotic (RM) could be used. In RT-Linux the Linux runs as low priority process. Advantages of RT-Linux are small foot size (approx. 150 KB), higher degree of predictability, response in terms milli-second and sophisticated service from GPOS Linux.

III. PRIORITY INVERSION PHENOMENON

Priority inversion is phenomenon that occurs when a higher priority task waits for a lower priority task to release a resource it needs but that is held by lower priority task and meanwhile the intermediate priority tasks pre-empt the lower priority task from CPU. So, high priority task would be blocked. Now the priority of the task effectively gets inversed with respect to the medium priority task. Priority Inversion will occur in Multi-tasking system when resource is shared across the tasks.

Priority inversion would happen when high priority task T(1) and low priority task(T3) share critical resource and T3 first gets the resource, while T1 is ready and ask for access the critical resource, it is blocked for waiting T3 to release the resource, at this time, middle priority task T2 which does not need the resource is ready, T2 seizes the processor to made T1 continue block until T2 ends, and then T3 gets the processor again to complete the use of shared resource, finally T1 pre-empts T3 and running.

In above scenario , the priority of T1 comes down to level of T3's priority, So the high priority task T1 cannot meet its deadline first, if many middle priority has emerged, high priority task would be blocked for undetermined duration which is called as Continuous Priority Inversion Phenomenon. Serious continuous priority inversion phenomenon will end up with collapse of the whole real-time system. One of the most famous victims of Priority Inversion problem is the "mars path finder". Although priority inversion phenomenon was found early in 1970 of the last century, there is no effective and simple solution yet. In some other scenario, there would be another task (T4) which would tries to acquire the common resource which may lead to cyclic change of priority which leads to un-bounded priority inversion problem.

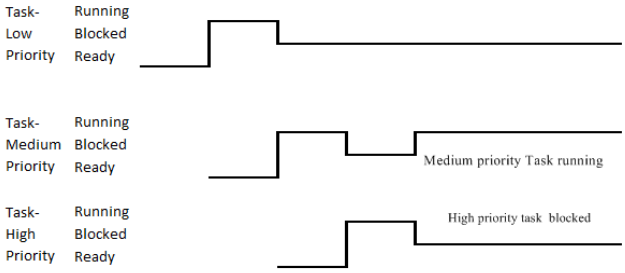

Fig.2: Priority Inversion Phenomenon.

Priority inversion phenomenon is an important reason of unpredictable errors in real-time system. Serious continuous priority inversion phenomenon will lead to real-time system to collapse. The priority inversion is a most common problem that will affect the real-time performance of a real – time kernel. Priority inversion affects heavily on the system predictability. So the real-time system may enter into unpredictable mode.

IV. EXISTING SOLUTION

There are many methods to solve the priority inversion problem and each method has its own advantages as well as disadvantage. A Solution will suite only for the specific application environment.

4.1. Locking the Scheduler

Locking the scheduler will suspend the scheduler. Whenever a task enters into any of the critical section, the scheduler will be locked. Once the task comes out the critical region the scheduler would be released. This method stops temporarily the scheduler till the task is in the critical section.

If task (T1) wants to access critical section, It will lock the scheduler, enters the critical section. Now even a high priority task (T2) is ready in run queue, Task(T1) could not be pre-empted due to fact the scheduler is locked. Once the task(T1) has finished its update on critical section. It will release the scheduler. Now, the scheduler will schedule the high priority task (T2).

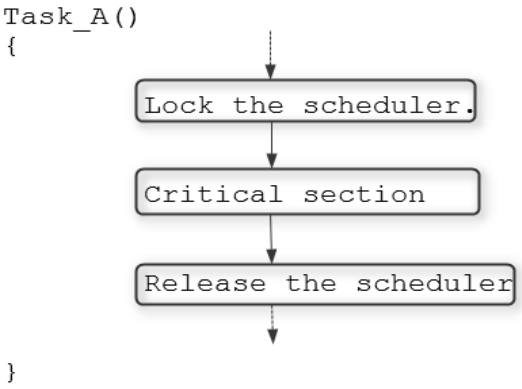

Fig.3: Solution to priority inversion – Locking scheduler.

This approach is simple and easy to implement by programmer. System behaviour Analysis could be done easily. Consider, there are three task in the system namely task(TH) with highest priority, task(TM) with medium priority and task(TL) with lower priority. A resource (RA) is shared between task(TM) and task(TL). Task TL is ready and running enters the critical section. Now the scheduler is locked and the high priority task(TH) is entering into run queue. But, since the scheduler is in locked state, It could not have CPU. So, the task(TH) is blocked due to a resource that is nowhere related to TH.

So, this approach will result in high degree un-predictability in the system if the number of task is more. Higher priority task may often miss the deadline. If the critical section access time is high which in turn increase the time for which scheduler is in locked state. So, this method is not suitable for system with more number of task or system with larger critical section.

4.2. Priority Remapping

This priority remapping method improves the priority inheritance protocol. The method is expanding the priority from 64 to 128 without changing the external interface. For users, there are still only 64 priorities. But in internal task creation function, priority is multiplied by 2 to achieve the priority remapping effect, so internal priority is extended to (0, 2 ..., 126) even priority, the rest odd priority are left for changing when priority inversion phenomenon arises. For example, priority of task that accesses critical resource is 80 while priority of task that asks for the resource is 30, then priority of the task which accesses the critical resource will promote to 31 to prevent priority inversion phenomenon

4.3. Priority Exchange

Priority exchange method is also deriving from the improvement of priority inheritance protocol. The basic idea is that exchange tasks' priorities when high priority task is blocked and priorities will be changed back after critical region is finished. This method makes sure every task has unique priority in the system and solves the priority inversion problem. The drawback is that priority's exchange requires additional costs.

4.4. Priority Inheritance

The idea of the priority inheritance protocol is, When a high priority task is blocked by a low priority task for getting the resource, then the low priority task will inhert temporarily the priority of the higher priority task. When the resource is released then low priority task's priority will be assigned the same value as earlier.

If task (TH) is blocked by low priority task Tl, TL will inherit TH's priority to avoid middle priority task TM seizing the processor. After TL withdraw its critical region, its priority resume to the original low priority. If there are many high priority task being blowe, low priority task would inherit the highest prioirty among all priority tasks. Prioirty inheritance mehod makes developers do not need to know anything about the task's requirements of resurces which reduce the burden of the programmers. This improves the easier predictiability and development of huge real-time embedded system. But this method can't prevent deadlock.

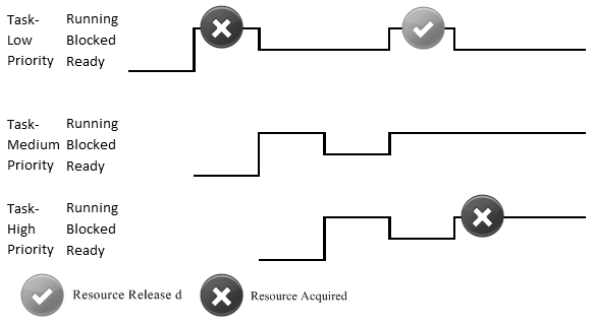

Fig.4: Solution to priority inversion – Priority inheritance

V. PROPOSED SOLUTION

The Proposed solution is another algorithm that is new to RT-Linux. But this algorithm has wide usage in some commercial Real Time Operating Systems. Proposed solution contains more than one algorithm which discuss advantage and disadvantage in each use case level.

5.1. Highest Locker Protocol

The basic idea of priority ceiling protocol shows in Figure 5. Programmer sets a ceiling priority for each shared resource, Highest priority value of the task which is going to use this resource. The ceiling is the highest task priority for requests the resource. High priority task T1 and low priority task T3 need the same resource (the black part in Figure 4). T3 gets ready first and access the resource. At t1 time point, T1 gets ready and tries to access the resource which has already held by T3. So T3's priority promotes to the ceiling at t2 time point. After T3 finishes the use of resource at t3 time point, T3's priority resume to the original low priority; T1 gets the resource and begin to run. After T1 ends, T2 begins to run at t4 time point. After T2 ends, T3 continues to run at t5 time point. This method extends priority inheritance protocol. Priority ceiling protocol prevents deadlock and reduces the block time. The drawback is that developers need to do static analysis in advance and assign a highest priority for each shared resource. This make the programs become more complicated and developers must know all tasks' priorities and all idle resources' priorities. And each resource size a ceiling priority also decreases the number of task created by application system. So priority ceiling protocol is not suitable for complex applications.

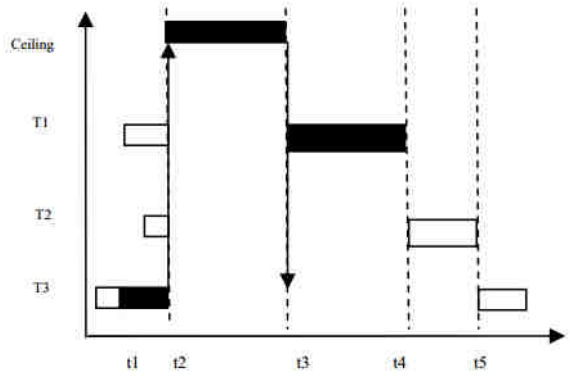

Fig.5: Solution to priority inversion – Highest Locker Protocol.

5.2. Prioirity Ceiling Protocol

Priority Ceiling Protocol (PCP) extends the ideas of PIP and HLP to solve the problems of unbounded priority inversion, chain blocking, and deadlocks, while at the same time minimizing inheritance-related inversions.

Resource sharing among tasks under PCP is regulated using two rules for handling resource requests: resource grant and resource release. We elaborate these two rules in the following:

Resource grant rule:

Resource grant rule consists of two clauses. These two clauses are applied when a task requests to lock a resource.

5.2.1. Resource request clause:

(a) If a task Ti is holding a resource whose ceiling priority equals CSC, then the task is granted access to the resource.

(b) Otherwise, Ti will not be granted CRS, unless its priority is greater than CSC (i.e. pri(Ti) > CSC). In both (a) and (b) above, if Ti is granted access to the resource CRS, and if CSC < C eil(C Ri), then CSC is set to C eil(C

2. Inheritance clause: When a task is prevented from locking a resource by failing to meet the resource grant clause, it blocks and the task holding the resource inherits the priority of the blocked task if the priority of the task holding the resource is less than that of the blocked task.

5.2.2 Resource Release Rule:

If a task releases a critical resource it was holding and if the ceiling priority of this resource equals CSC, then CSC is made equal to the maximum of the ceiling value of all other resources in use; else CSC remains unchanged. The task releasing the resource either gets back its original priority or the highest priority of all tasks waiting for any resources which it might still be holding, whichever is higher.

PCP is very similar to HLP except that in PCP a task when granted a resource does not immediately acquire the ceiling priority of the resource. In fact, under PCP the priority of a task does not change upon acquiring a resource merely the value of a system variable CSC changes. The priority of a task changes by the inheritance clause of PCP only when one or more tasks wait for a resource it is holding. Tasks requesting a resource block almost under identical situations under PCP and HLP. The only difference with PCP is that a task Ti can also be blocked from entering a critical section, if there exists any resource currently held by some other task whose priority ceiling is greater or equal to that of T2. A little thought would show that this arrangement prevents the unnecessary inheritance blockings caused due to the priority of a task acquiring a resource being raised to very high values (ceiling priority) at the instant it acquires a resource. In PCP, instead of actually raising the priority of the task acquiring a resource, merely the value of a system variable (CSC) is raised to the ceiling value. By comparing the value of CSC against the priority of a task requesting a resource, the possibility of deadlocks is avoided. If no comparison with CSC would have been made (as in PIP), a higher priority task may later lock some resource required by this task leading to a potential deadlock situation where each task holds a part of the resources required by the other task.

VI. IMPLEMENTATION DETAILS

By analyzing the RT-Linux source code, priority reverse problem may not happen due to the priority inheritance protocol that is enabled in the RT linux kernel patch. But the Priority Inheritance Protocol is not much suitable for the Real Time Application. As it only avoids the Un Bounded Priority inversion problem. The problem arises from the fact that Priority Inheritance Protocol has its own limitation like chain blocking, deadlocks. Moreover chain blocking will result in un-predictable behavior in the system and many task may miss their respective deadline. But Priority Inheritance is used due to its simplicity. To solve this problem of priority reverse or priority inverse problem alogn with chain blocking problem and deadlock problem, priority ceiling or variant of priority ceiling protocols have been adopted. The algorithm and implementation details are discussed below.

- A global variable ‘System_Ceiling_Priority’ is declared and initialized to zero.
- If a task Ti is holding a resource whose ceiling priority equals to the value in System_Ceiling_Priority, then the task is granted access to the resource.
- (b) Otherwise, Ti will not be granted the resource, unless its priority is greater than System_Ceiling_Priority (i.e. pri(Ti) > System_Ceiling_Priority). In both (a) and (b) above, if Ti is granted access to the resource , and if

System_Ceiling_Priority < C eil(C Ri), then System_Ceiling_Priority is set to Ceil(Resource)

- Inheritance clause: When a task is prevented from locking a resource by failing to meet the resource grant clause, it blocks and the task holding the resource inherits the priority of the blocked task if the priority of the task holding the resource is less than that of the blocked task.

VII. EXPERIMENTAL RESULTS AND DISCUSSION

Create task T1, T2, T3, T4, T5, T6 with priority value higher for lower task number. And T2, T5 shares the resource R1 and R2. T1 to T2 is grouped as high priority task. T3, T4 is grouped as medium priority task. T5, T6 are grouped as low priority task.

Figure 6 shows the running result that didn't modify the kernel, T2 seizes the processor make T1 blocked long time and give raise to priority inversion phenomenon.

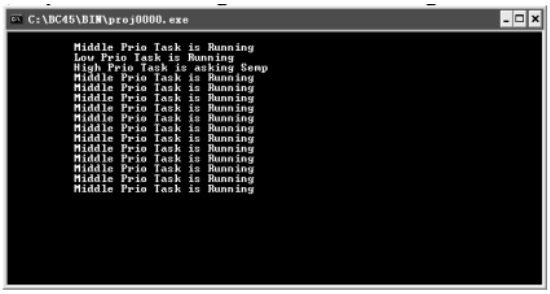

Fig.6: Priority Inversion Problem.

After use modified priority ceiling method, priority inversion phenomenon is confined to one level, and there is not chain blocking and deadlock possibility. The experiment running result is shown in Figure 7.

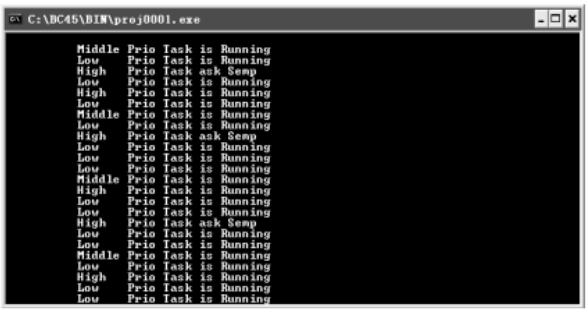

Fig7: Modified Priority Ceiling Protocol.

VIII. CONCLUSION

This paper discuss several methods to solve the priority inversion problem in RT-Linux such as disabling interrupt, priority inheritance protocol, priority remapping method, priority exchange and modified priority ceiling protocol. The simplest method is that to lock the scheduler preventing the system to context switch. It is very simple and effective to limit priority inversion problem when critical region is very short. But if the critical region is relatively long, higher priority tasks would miss their deadline often. Priority exchange method is relative complex in design, need to modify RT-Linux kernel, but it is convenient to develop complex real-time application for application programmer. The drawback is that lack of deadlock prevention, chain blocking and the exchange of priority will pay some unnecessary overhead to Real Time System. In cases of exchange priority frequently, the method will increase burden of the system and effect the predictability of system. Priority Inheritance protocol introduce inheritance related inversion which is not suitable for some Real Time System requirements due to lack of dead lock prevention, chain blocking, Inherited Inversion problem. Modified Priority Ceiling Protocol takes the advantages of priority ceiling protocol (Higher Locker protocol) and the priority Inheritance protocol. By this method, system is free from unbounded priority inversion, Chain Lock, Dead Lock. And, this minimizes the effect of Inheritance related inversion. The discussed method improves effectively the removal of priority inversion problem with relatively less overhead. The Inheritance related inversion could be further reduced. But this algorithm only minimize the impact but not removing the same. By having this algorithm, RT-Linux could be used in many time critical complex systems.

REFERENCES

[1] Ianlin Zhu, Keou Liu, Yu Tu, Yi Yuan, Xialoliang Gao, "RT-Linux priority reversal and priority inheritance mechanisms", 2013 Fifth Conference on Measuring Technology and Mechatrnics Automation.

[2] XU Liang, XU Zhongwei. "Task scheduling optimization in in μC/OSII system"[J] computer project. 2007,33(19):57-59

[3] C.M. Krishna, Kag G. Shin "Real-time Systems" Tenth reprint 2013.

[4] Lee J H, Kim H N. "Implementing Priority Inheritance Semaphore on uC/OS Real-time Kernel[C]" Proc.of IEEE Workshop on Software Technologies for Future Embedded Systems.2003-05.

[5] WANG Jigang, GU Guochang, XIE Shibo, LI Yi. "Priority inheritance Protocol and an improved algorithm" Computer engineering,2007,33(8):41-44.

[6] ZHAO Qi, SUO Xiaoran. "Research of priority inversion in real-time system"[J] Computer Application Research .2008,25(6):1728-1730

[7] ZHOU Xuchuan. "A method to solve priority inversion problem in μC/OS-II system"[J] Application of embedded operating system 2007,23(5-2) :58-64.