"Feasibility of effectuating Resource Restitution in QNX real time systems for Priority Inversion"

Mr. Atul Anilkumar Kumbhar, Prof. D G Chougule, Asst. Prof. Amrita Manjrekar

Master of Technology (Computer Science and Technology), Department of Technology,Shivaji University, Kolhapur.
Email: atulkumbhar@aol.in,atul@claysol.com
Department of Technology,Shivaji University, Kolhapur.
Email: dgchougule@yahoo.com
Department of Computer Science and Technology, Department of Technology, Shivaji University, Kolhapur.
Email: amrita5551@gmail.com, aam_tech@unishivaji.ac.in

Abstract :

Priority inversion is where a lower priority process gets a hold of a resource that a higher priority process needs, preventing the higher priority process from proceeding till the resource is freed. This problem is enlarged when the concurrent processes are in a real time system where high- priority threads must be served on time.

We propose a greenhorn approach for the priority inversion avoidance [1] in QNX RTOS which is preemption based technique which restores the resource(s) on the arrival of a high priority thread .

This approach's interpretations verify that the approach is unsuitable i.e. unfeasible for real time systems where high-priority threads must be served on time as against stated in [1].

Keywords : ***CPU Scheduling, Priority Inversion, QNX RTOS,QNX Neutrino.***

1. Introduction:-

A real-time operating system (RTOS) is an operating system (OS) intended to serve real-time application requests. A key characteristic of a RTOS is the level of its consistency concerning the amount of time it takes to accept and complete an application's task; the variability is jitter.

A real-time OS has an advanced algorithm for scheduling. Scheduler flexibility enables a wider, computer-system composition of process priorities, but a real-time OS is more frequently dedicated to a narrow set of applications. Key factors in a real-time OS are minimal interrupt latency and minimal thread switching latency, but a real-time OS is valued more for how quickly or how predictably it can respond than for the amount of work it can perform in a given period of time.

Concurrent executions of processes have become an important feature of systems especially in multi user environment. Concurrent processing gives many advantages like increased response in human interfaces, I/O-bound applications, distributed and parallel systems, etc. However the difficulties in using concurrent processes are also clear. The difficulties include shared data management of different processes, proper switching of processes and their restoration and priority scheduling of processes with different priorities, running at the same time, etc. Typically concurrency is brought up by multithreaded environment in which several threads share the same address space and are executed simultaneously. But processes share resources; events outside the scheduler's control can sometimes prevent the highest-priority ready processes from running when it should. When this happens, a critical deadline could be missed, causing the system to fail.

In priority scheduling, threads are assigned priority by the operating system and the resources are allocated to the threads according to their priorities i.e. if two threads are waiting for a resource then the higher priority thread will precede the lower priority thread on the availability of the resource. A problem that occurs with priority scheduling in multithreaded environment is the priority inversion.

The pair of highest and lowest relative priority must share a resource, say by a mutex, and the third must have a priority between the other two. The scenario is as shown in the figure below. First, the low-priority task acquires the shared resource (time t1). After the high priority task preempts low, it next tries but fails to acquire their shared resource (time t2); control of the CPU returns back to low as high blocks. Finally, the medium priority task—which has no interest at all in the resource shared by low and high—preempts low (time t3). At this point the priorities are inverted: medium is allowed to use the CPU for as long as it wants, while high waits for low. There could even be multiple medium priority tasks.

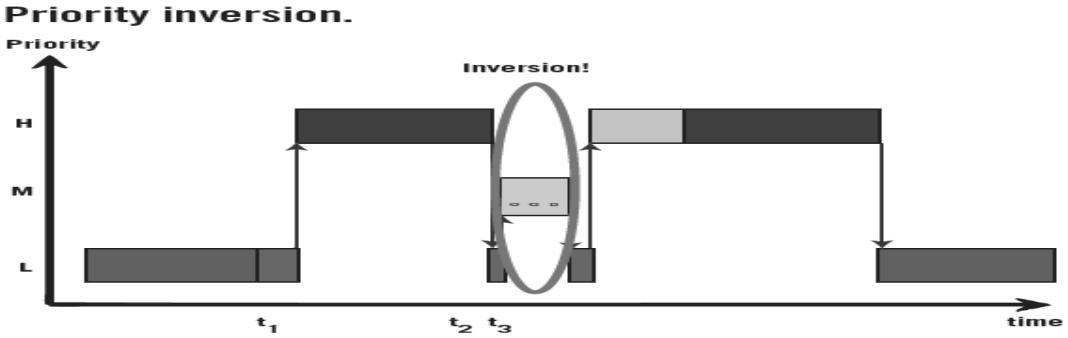

Figure 1: Priority Inversion Problem

The risk with priority inversion is that it can prevent the high-priority task in the set from meeting a real-time deadline. The need to meet deadlines often goes hand-in-hand with the choice of a preemptive RTOS. Depending on the end product, this missed deadline outcome might even be deadly for its user.

One of the major challenges with priority inversion is that it's generally not a reproducible problem. First, the these (as above in figure) steps need to happen—and in that order. And then the high priority task needs to actually miss a deadline. One or both of these may be rare or hard to reproduce events. Unfortunately, no amount of testing can assure they won't ever happen in the field.

2. Literature survey:-

Priority inversion is a well-known problem in concurrent programming especially in real time applications. There are basically two well-known protocols that have been used excessively as attempts to avoid priority inversion. The first is known as priority ceiling [2, 3] and the other is priority inheritance [4, 5]. Priority ceiling protocols require that a priority value, the ceiling, be associated with a resource and the corresponding lock [2]. This ceiling is defined as the maximum priority of tasks contending for the resource. In this way the priority inversion problem is resolved. The basic detriment here is that the protocol requires programmers to supply priority ceiling for each resource. Secondly priority ceiling may result in false blocking of threads [4].

The second type of protocol that is seldom used for priority inversion avoidance is priority inheritance protocol [3]. Priority inheritance protocol involves raising the priority of a thread that is holding a lock causing a higher priority thread to lock. If a thread which is a low priority thread is using a resource due to which a higher priority thread is blocked then the low priority thread inherits the priority of the latter and get executed quickly to give way to the higher priority thread. Suppose T1 owns mutex m1 and is waiting for mutex m2 which is owned by T2 and so on. If a high priority thread (Th) now blocks on m1, the protocol has to march down the chain (T1, T2, …) promoting each element otherwise Th would be in danger of unbounded inversion as lower tasks in the chain failed to advance because of intermediate priority tasks. So priority inheritance needs to be a transitive operation. The priority inheritance solution is transparent to application and removes hazards like "false blocking" present in priority ceiling protocol and its variants.

Besides the advantages the priority inheritance protocol has several other notable disadvantages. Four of the basic detriments of priority inheritance protocol are described in [6]. These detriments are stated below:

- The nested critical regions protected by priority inheritance locks generate long inversion delays.
- Priority inheritance fails if tasks mix inheriting and non-inheriting operations.
- Priority inheritance worst case performance is worse than the easy alternatives in most cases.
- Inheritance algorithms are complicated and easy to get wrong.

The proper proofs of these detriments of priority inheritance can be seen in [6].

3. Problem formulation: - Need of proposed research work

This approach is basically made for serving only real time systems in which high priority thread must be served as soon as it arrives. Therefore, the system in this approach saves the resource as backup and when a higher priority thread arrives it revokes the low-priority thread, restitutes (restores) the resource, let the high priority thread executes and later restarts the revoked threads. Although the low-priority threads may have to be revoked several times but the high priority thread is always served the best which is a basic requirement in real time systems.

Preliminary

Our proposed approach requires language to have proper mechanism for synchronized sections (to control access to resource). Synchronized sections are lexically delimited blocks of code guarded by monitors. They may be methods or just code blocks. Only one thread may execute within a synchronized section at any time, ensuring exclusive access to all monitor protected blocks.

Synchronized section may contain any number of objects but typically synchronized sections are made small

by programmers in order to facilitate multiprogramming. Secondly the objects residing in the synchronized sections are shared objects.

Multiple Priority Levels

Here we discuss threads with only two priorities: low and high. However in usual cases systems generally have threads with several levels of priorities. If several levels of priorities are used in our system then the lowest level of priority will suffer a lot in terms of time delay because each time a higher level thread arrives, the lowest priority thread will be preempted and restarted later. This will degrade the overall performance of the system. In order to cope up with this problem we classify the different levels of priorities into two categories. These two categories are high and low. The categories may not contain equal number of levels of priorities. This is up to the programmer to make levels of priority on design time. Since the proposed approach is basically meant for real time system where the high priority threads are given the utmost advantage therefore as a tradeoff the low priority threads have to suffer. Hence making classes of priorities will prevent complete blocking of lowest priority threads.

4. Outline of Proposed Work:-

As proposed in [1] a novice approach for the priority inversion avoidance which is preemption based technique which restores the object(s) on the arrival of a high priority thread. In this method no log is maintained, instead we use a shadowing technique for resource consistency.

In this approach (for QNX RTOS e.g. QNX Neutrino Rtos) when a low priority thread is entering a synchronized section the objects (shared resources) to be used in the synchronized section are backed up and the low priority thread is allowed to use the shadow version of the shared resources. When the low priority thread has finished its execution the backed up resource is replaced by the shadow of the resource updated by the low priority thread.

Now during the execution of the low priority threads, if a higher-priority thread arrives and needs to enter the synchronized section, the lower-priority thread in the synchronized section is preempted and the resource previously saved as backup is restituted (restored). The higher priority thread is now allowed to enter the synchronized section which contains the unaltered resource. When the higher priority thread is finished the low priority thread is restarted and then allowed to enter the synchronized section. The low-priority thread now uses the shadow version of the updated resource (updated by high-priority thread).

This means that whenever a low-priority thread enters the synchronized section it uses the shadow of the original resource object(s) while if a high priority thread enters the synchronized section it uses the original resource object(s) in the synchronized section.

Below are the diagrams showing the proposed approach in QNX Real Time Operating System.

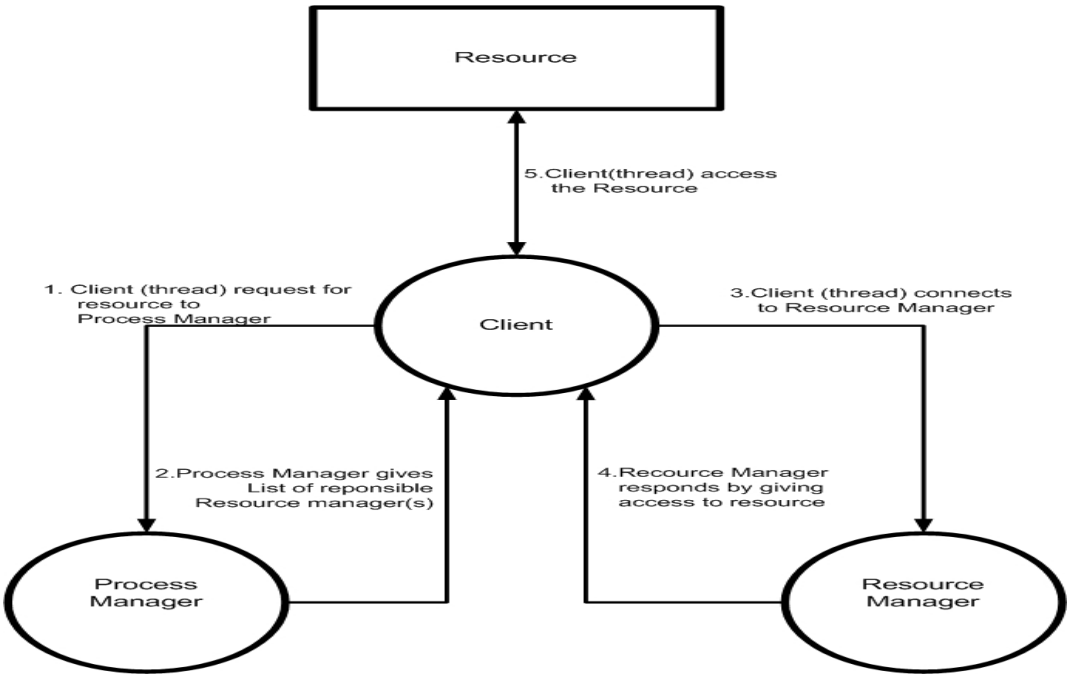

Figure 2: Architecture Design

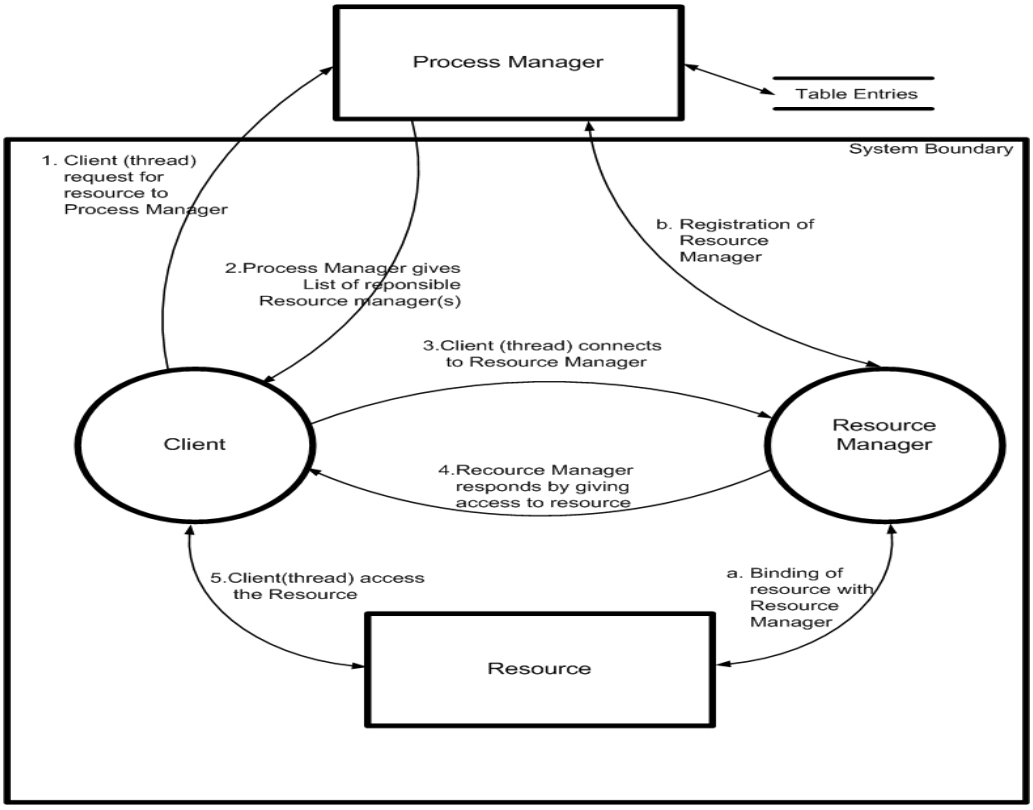

Figure 3:- Data Flow Diagram (DFD)

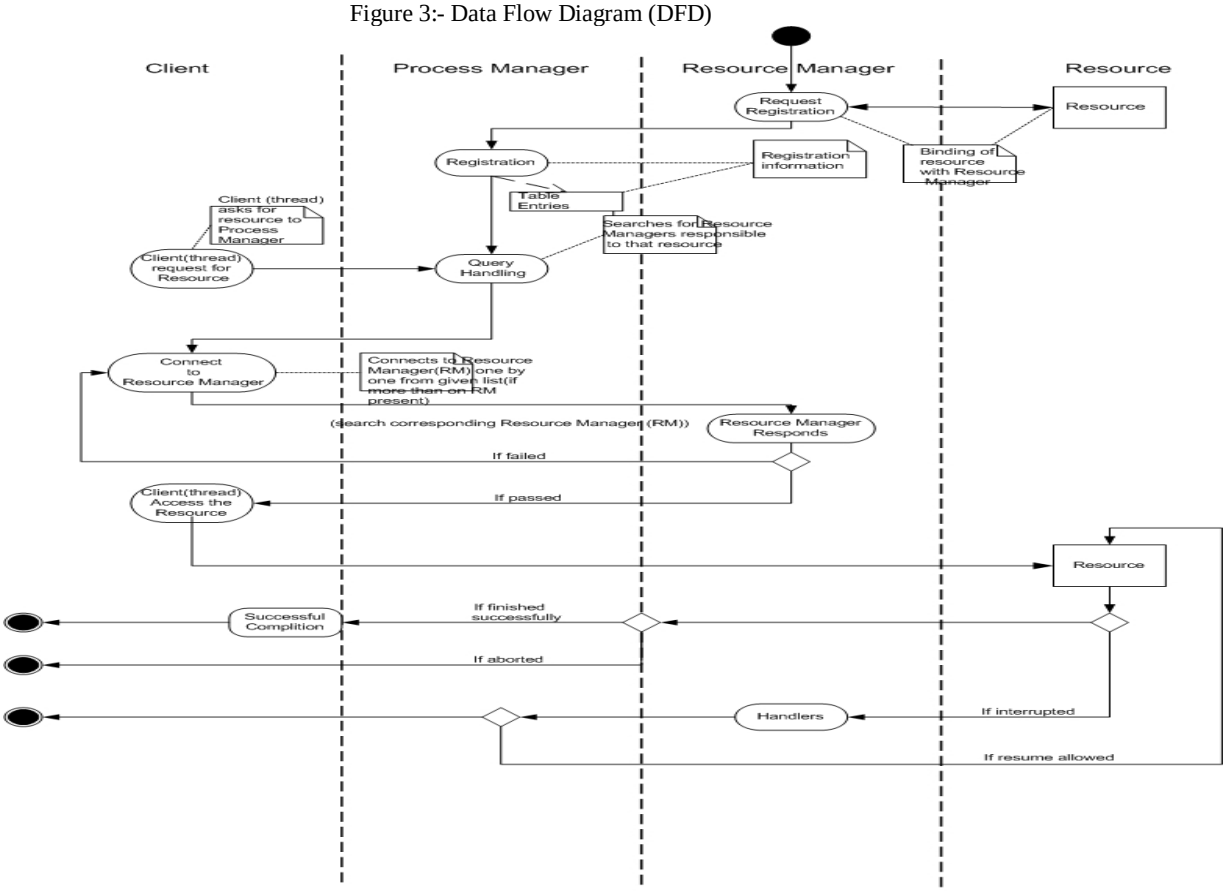

Figure 4: Activity Diagram

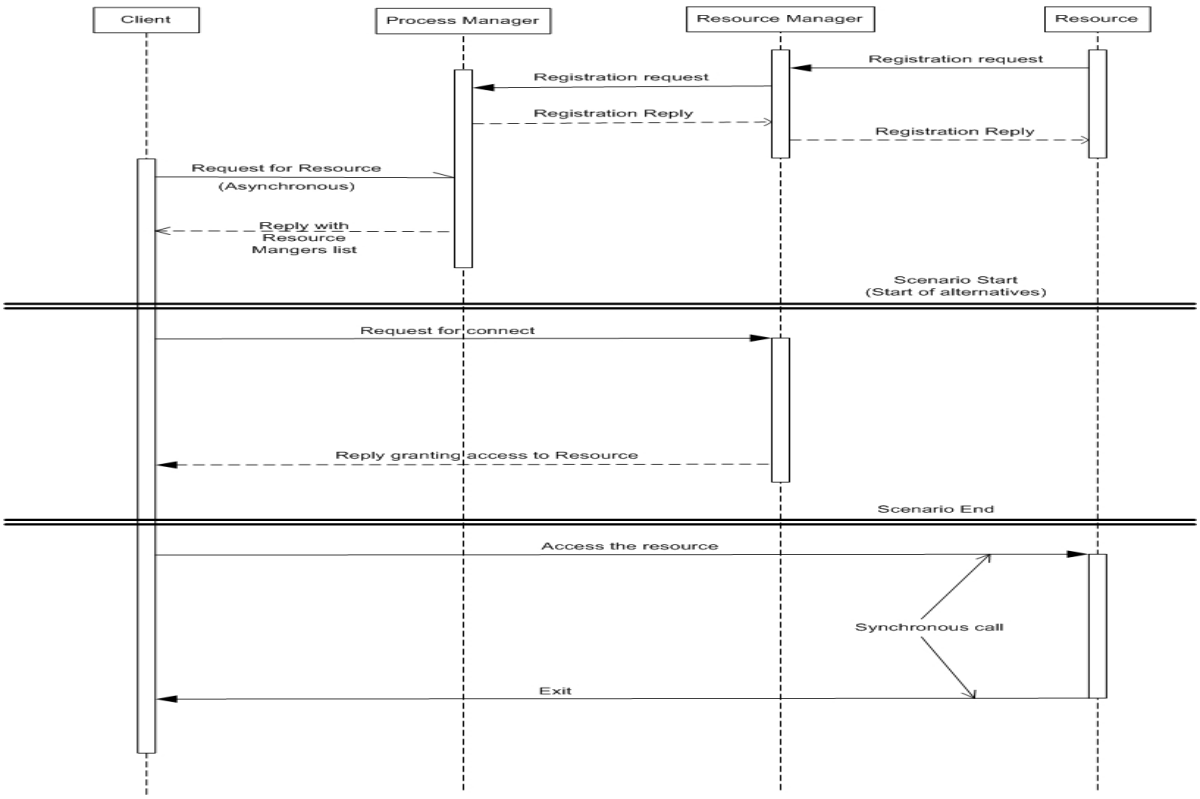

Figure 5:- Sequence Diagram:

***5. Contribution*:**

By considering following points in QNX Neutrino, we conclude that this approach is not feasible. However, this may become a feasible solution for other Real Time Operation System:

- When we perform a fork() or spawn(), we are duplicating the process. Because the new process will have its own independent virtual memory, it will not inherit any resources (resources such as memory allocations, file descriptors, will not be duplicated). For this reason, it is usually recommended that all Resource Manager process creation be performed during system boot-up or initialization.
- We, however, have the Resource Manager main processing thread receive all incoming messages, and then create child threads to process each message as they are received. we can then have two different thread types, one for lower priority Client, and one for higher priority Client.
- The thread processing message from a lower priority Client can be designed to handle a signal. The Resource Manager main thread will send this signal when it receives a new message from a higher priority Client. When the child thread receives this signal, it will then discard all work and terminate.
- As long as the Resource Manager main thread maintains the fact that there is only one child thread, you do not have to worry about data corruption by competing threads.
- All message replies are handled by the Resource Manager main thread. So if a child thread handling a lower priority Client message is terminated, no replies are sent. So it will be up to the Resource Manager main thread to maintain a copy of the OCB(Open Control Block) data structure, such that it does not lose that message.
- We will need to design a mechanism in which the Resource Manager determines the higher/lower priority information based on the Client.

However, we found a few potential issues with this type of implementation.

- For a “file system”, how do we discard the work? If we are overwriting the contents of a file on the disk, you will need to keep a copy of the data that is being overwritten. And then when the discard and terminate signal is received, halt the current data transfer, and the write the original data back. We can reach a condition where the data write request from a lower priority Client will never be completed, as it is constantly being interrupted by a message from a higher priority Client.
- For the above discard implementation … if we are to store a copy of the original data, you will

need to have sufficient local memory storage for this, which will add to the overall memory usage.

- What about DMA(Direct Memory Access) transactions? If we terminate a DMA transfer (providing the DMA controller permits this), we have no way of telling how much progress has been made, so we will have to overwrite the entire original data to restore it, which will take almost double the amount of DMA time (worst case scenario), as if we had just allowed the original DMA to complete.

As for other implementations other than "file system" or "shared memory" … all we can think of at the moment are communications, audio, and graphics:

- Such an implementation is not feasible for general serial communications, but possible for packetized data such as HDLC or Ethernet. However, most SoC of today uses a communications engine, which handles the packet data using DMA. You could terminate a transmit in mid packet, and the receivers would discard the packet fragment, but how long will this take to complete? Would it not be faster to just let the current packet complete the transmit instead of terminating it?
- Usually, it is generally accepted that the source code required to do so would be so prohibitively large, that it would make the throughput efficiency unacceptable.
- We would not do this with an audio driver, as whatever data received is already sent to the speaker.
- Graphics driver … the only thing we can see would be to put a wrapper layer in front of the rendering engine. But the different Clients would render into their own private video memory anyways (either by different video layers, or different virtual video layers that will be assembled by the Composition Manager), so again, this is not feasible.

6. Conclusion:

In contrast with previous research[1] , our study resoundingly found that the overall QNX RTOS Resource Manager architecture does not require this additional overhead and is also unsuitable i.e. unfeasible.Their reasons are as follows:

- QNX already has built in "Priority Inversion" prevention. The QNX Resource Manager (Resource Manager, hence forth referred to as "Server") will always adjust its execution priority to match that of the Process using this Resource Manager (hence forth referred to as "Client"). Therefore, if a higher priority Client (Client H) sends a message to the Server , but the Server is currently processing a message from a lower priority Client (Client L), it will automatically adjust the priority to match that of the higher priority Client H, complete the message processing for the lower priority Client L at that higher priority, and then immediately begins processing the message from the higher priority Client H.

- The concept of backing out of a message processing for Client L in order to process the message for Client H will create the possibility that it will add more processing load than realistically needed.

- The Resource Manager can be designed to be multi-threaded, where multiple threads can be launched to service messages from the same message queue. Therefore, if both Client H and Client L sends a message to the Server at the same time, then both messages will be processes simultaneously by the two threads, and execution will be controlled by the QNX Neutrino scheduler, as per their Client's priority levels.

7. Future Work:-

- The Resource Manager can also be designed to have multiple incoming message queues, one for higher priority processes, and one for lower priority processes. Each incoming message queue can then be processed by their own thread, or their own set of multiple threads.

- By creating a Resource Manager with multiple threads, they can take advantage of the advanced features of the QNX Neutrino kernel such as Symmetric Multi-Core Processing and Adaptive Partitioning. For example, you can assign the threads that are designated to handle messages from higher priority Clients in a specific partition, and with multiple cores in the system, have the capability to dedicate a percentage of total CPU time to the simultaneous processing of multiple threads in the Resource Manager.

Challenges:-

1] To choose resources which will work with this approach (Presently, File system is Considered).

2] Taking backup of resource & conveying it to process manager.

3] Variant-(How) To know the amount of work done by thread .Then, to allow or disallow its execution.

4] Interdependencies of threads to be known earlier.

Acknowledgment:-

We would like to thank Department of Technology, Shivaji University, Kolhapur for supporting and providing facilities to this research works. Special thanks to anonymous reviewers for their valuable comments on this paper. Also to :

Ed Lee - Field Application Engineer ,Asia Pacific Sales, QNX

Software Systems. A subsidiary of Research in Motion Limited.

Jeevan Mathew- Co-Owner, BizDev, Eng. Services at embeteco GmbH & Co.KG , Hannover Area, Germany previously at QNX(now RIM)as Support Specialist From November 2000 – May 2012 (11 years 7 months).

Umesh Thallum- Engineering Director at Claytronics Solutions Pvt. Ltd., Bengaluru Area, India.

Armin Steinhoff - CEO at STEINHOFF Automation & Fieldbus-Systems,Germany.

References:-

1] Tarek Helmy & Syed S. Jafri –"Avoidance of Priority Inversion in Real Time Systems Based on Resource Restoration ", International Journal of Computer Science & Applications , 2006 .Technomathematics Research Foundation,Vol. III, No. I, pp. 40 – 50.

[2] Frank Mueller "Priority Inheritance and Ceilings for Distributed Mutual Exclusion" The 20th IEEE Proceedings on Real-Time Systems Symposium, 1999, 1-3 Dec., pp.:340 – 349.

[3] M. Chen and K. Lin. "Dynamic priority ceilings: A concurrency control protocol for real- time systems " Real-Time Systems, 2(4):325–346, 1990.

[4] Huang, J., Stankovic, J.A., Ramamritham, K. and Towsley, D.,"On using priority inheritance in real-time databases", Twelfth Proceedings on Real-Time Systems Symposium, IEEE 4-6 Dec. 1991 Page(s):210 – 221

[5] Sha L., Rajkumar, R and Lehoczky, J. P., "Priority inheritance protocols: An approach to real-time synchronization.", IEEE Transactions on Computers, Volume 39, Issue 9, Page(s):1175 – 1185 , Sept. 1990.

[6] Yodaiken, V. "Against priority inheritance" Finite State Machine Labs (FSMLabs)Technical Report, June 25, 2002.

[7] Nigolah, Cyprian F., Wang, Yingxu, Tan, Xinming, "Implementing task scheduling and event handling in RTOS", IEEE proceedings of CCECE 2004-CCGEI 2004, May 2004.

[8] Reiter, R. "Towards a logical reconstruction of Relational database theory" in Brodie et al. ch 8, 1984.

[9] Sha L., Rajkumar, R and Lehoczky, J. P., "Priority inheritance protocols: An approach to real-time synchronization.",IEEE Transactions on Computers, Volume 39, Issue 9, Page(s):1175 – 1185 , Sept. 1990.

[10] "The Design and Performance of Real-Time Java Middleware "-Angelo Corsaro, Student Member, IEEE, and Douglas C. Schmidt, Member, IEEE , IEEE Transactions On Parallel And Distributed Systems, Vol. 14, No. 11, November 2003

[11] "A Commercial-Off-the-Shelf(COTS) Dedication of a QNX Real Time Operating System (RTOS)"- Jang Yeol Kim, Young Jun Lee, Se Woo Cheon, Jang Soo Lee, Kee Choon Kwon Instrumentation and Control / Human Factors Division, Korea Atomic Energy Research Institute,1045 Daedeok-daero, Yuseong-gu, Daejeon, Republic of Korea 305-353. 2010 2nd International Conference on Reliability, Safety & Hazard (ICRESH-201O).

[12] " Research and Realization of the Mechanism of Embedded Linux Kernel Semaphore" by WANGYa-Jun - The teaching and research section of computer ,The Chinese People's Armed Police Forces Academy, langfang hebei, China. 2010 3rd International Conference on Advanced Computer Theory and Engineering(ICACTE).