International Journal of Computer Science & Applications
Vol. III, No. I, pp. 40 - 50

Avoidance of Priority Inversion in Real Time Systems Based on Resource Restoration

Tarek Helmy & Syed S. Jafri

College of Computer Science and Engineering,
King Fahd University of Petroleum and Mineral,
Dhahran 31261, Mail Box 413, Kingdom of Saudi Arabia,
Emails :{helmy ,shomaail }@ccse.kfupm.edu.sa

Abstract

Priority inversion is a problem that occurs in concurrent processes when low-priority threads hold shared resources required by some high-priority threads, causing the high priority-threads to block indefinitely. This problem is enlarged when the concurrent processes are in a real time system where high- priority threads must be served on time. A novice approach for avoiding the priority inversion problem is presented for processes in real time systems. This approach is based on backing up and restoring the shared resources. A low priority thread always starts on a shadow version of the shared resource, the original resource remains unchanged. When a high-priority thread needs a resource engaged by a low-priority thread, the low priority thread is preempted, the original resource is restored and the high-priority thread is allowed to use the original resource. The approach has been implemented in Java and the experimental results are fetched which verify that the approach is very suitable for real time systems where high-priority threads must be served on time.

Keywords : CPU Scheduling, Priority Inversion.

1. Introduction

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. 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. In this proposal, the priority inversion in concurrent processing is discussed and a solution for the avoidance of priority inversion based on revocation of threads using resource restoration is presented. The rest of this paper is organized as follows. Section 2 deals with description of the priority inversion problem. Section 3 comprises of a detailed literature survey of the solutions devised to the priority inversion problem. In section 4, we describe some preliminaries for our proposed approach. The details description of the proposed method is presented in Section 5. The implementation details are discussed in section 6.

Section 7 shows the performance of the approach by giving the experimental results and concluded with some comments.

2. The Problem Statement

Priority inversion is a problem which usually arises in priority scheduling of concurrent programs. It occurs in concurrent processes when low-priority threads hold shared resources needed by some higher priority threads. This causes the higher priority threads to block indefinitely. Low level synchronization primitives such as mutual-exclusion locks, semaphores or monitors are usually used for guarding these shared resources. Thus avoiding priority inversion is very essential especially in mission critical or real time applications because in these applications, the higher priority threads demand some level of guaranteed throughput otherwise the system will fail or stop working. Priority inversion is a well known problem and many approaches have been adopted for its avoidance. In the next section, we present a brief survey of the approaches introduced for avoiding priority inversion.

3. 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, 6] and the other is *priority inheritance* [3, 9]. *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 [3].

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 (T_h) now blocks on *m1*, the protocol has to march down the chain (*T1, T2, …*) promoting each element otherwise T_h would be in danger of unbounded inversion as lower tasks in the chain failed to advance because of interme diate 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 [11]. 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 are not mentioned here and can be seen in [11]. A part from the two protocols there are approaches that avoid priority inversion when using fixed priority-based scheduling. The approach in [4] presents several basic program structuring techniques which are effective means of avoiding priority inversion. With the help of these program structures there is no need of any protocols for avoiding priority inversion. Another approach discussed in [7] employs multi-queue based scheduling system in which all tasks pending on a resource are put into different queues with pre-assigned priorities. The technique eliminates the problem of priority inversion as well. Both of the above mentioned techniques involve considerable change in the application structure which may not be viable when dealing with big applications. An approach based on preemption of low-priority thread is discussed in [1]. The approach is based on combining compiler technique with run-time detection. In this approach when priority inversion has been detected, the run-time system selectively revokes the offending thread within a synchronized section. The compiler maintains log updates for the shared state of the threads which are active in the synchronized section. The log updates are used for undoing the work of revoked

threads. This means that when a higher priority thread arrives the low priority threads are revoked by reading the log in the reverse direction. This undoes all the affects the low priority threads have done to the objects in the synchronized section. The run-time system then transfers control for the thread back to the beginning of the section. The approach in [1] is used to preempt only those threads on which no other thread depends. If any thread depends upon the thread to be preempted then the high priority thread is supposed to wait although both of the threads are of lower priority. The reason for this, as mentioned in paper [1], is that the revoking of thread on which other threads are dependent will consequently cause the revoking of the dependent threads. This may lead to a cascade of roll backs. An obvious disadvantage of this cascade of rollback is the need to consider all operations including non-monitored ones for a potential roll back.

Our proposed approach share similar goals with these efforts but differ in some important aspects. Our 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 our approach saves the resource as backup and when a higher priority thread arrives it revokes the low-priority thread, 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. This will eliminate the problem of cascading roll back. Secondly our approach is very simple and does not require major change in program structures. Besides preventing priority inversion our approach prevents multiple-blocking and deadlock (the issues present in priority inheritance) as well.

4. Preliminaries

Our approach is applicable to any language that offers the mechanism of multithreading in which concurrent threads execute over object in a shared address space. Since the threads share the same memory space, i.e. they can share resources. There are critical situations where it is desirable that only one thread at a time has access to a shared resource. For instance crediting and debiting a shared bank account concurrently amongst several users without proper discipline will jeopardize the integrity of the data. Problems like lost update, incorrect summary problems may arise. To control access to shared resource, a high-level concept of **synchronized sections** is generally present in languages. Our proposed approach requires language to have proper mechanism for synchronized sections as well.

Synchronized sections are lexically delimited blocks of code guarded by monitors. They may be methods or just code blocks. Threads synchronize on a given monitor, acquiring the synchronized section on entry to the block and releasing it on exit. Only one thread may execute within a synchronized section at any time, ensuring exclusive access to all monitor protected blocks. Monitors are usually implemented using locking, with acquisition of a mutual exclusion lock on entry, and release of the lock on exit. 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.

4.1 The Proposed Approach

After having a detailed survey of the different approaches used for the avoidance of priority inversion problem with the advantages and pitfalls we propose a novice approach for the priority inversion avoidance. Our approach is basically a modification to the approach presented in [1]. Our approach is a preemption based technique which restores the object(s) on the arrival of a high priority thread. In this method no log is maintained as was in [1], instead we use a shadowing technique for resource consistency as mentioned in [4, 8 and 10]. In our approach 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 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.

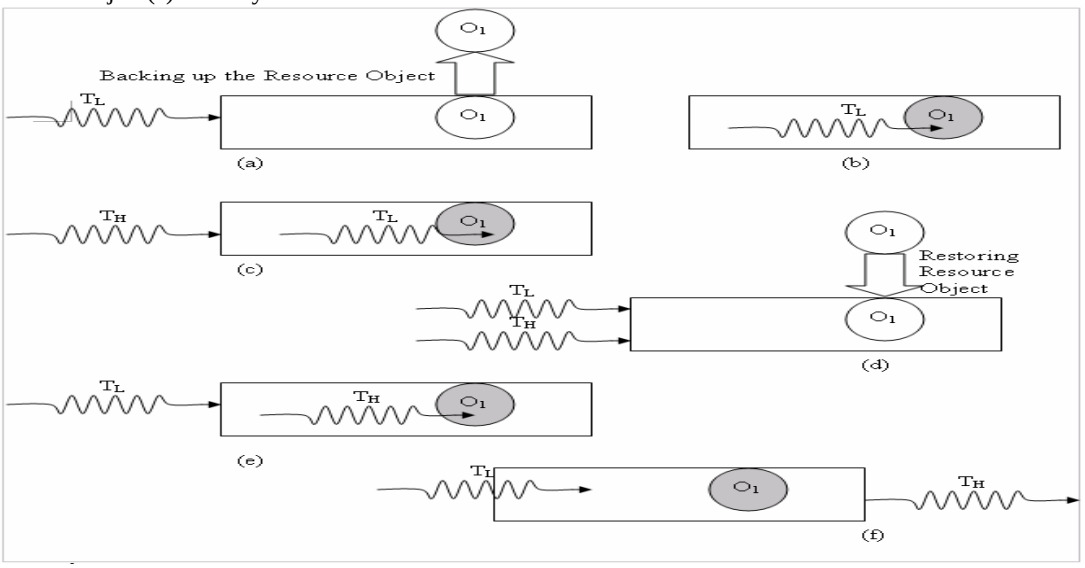

Figure 1: Revocation of threads with resource restoration

Figure 1 depicts a possible scenario of thread revocation. The wavy lines with arrows represent thread. These threads are labeled T_H or T_L meaning High-priority thread and low-priority thread respectively. The rectangular box represents the synchronized sections. The circles inside the rectangular boxes represent the resource objects. The updated resource objects are marked grey in color in order to differentiate from fresh resource objects. In Figure 1(a), a low-priority thread T_L is about to enter the synchronized section. Before a low priority thread in allowed to enter and modify, the resource object is backed up.

In Figure 1(b) the thread has now entered the synchronized section and has modified the resource object O_1 (which is the shadow of the original resource object). At this point a higher priority thread T_H tries to acquire the same monitor but is blocked by the low-priority thread T_L as shown in Figure 1(c). Now the T_L is revoked and is shifted back to the beginning of the synchronized section. The shadow resource object which was modified by the T_L is replaced by the original resource object previously saved. This is shown in Figure 1(d). After the resource object has been restored the T_H is now allowed to enter the synchronized section as depicted in Figure 1(e) and the T_L stays out of the synchronized section waiting for the monitor to release. Later on when T_H has completed its execution in the synchronized section and the monitor is released; T_L gains reentry and uses the shadow of the modified resource object as shown in Figure 1(f). In the following sections we discuss some peculiarities of the problem and the solution provided for them.

4.2 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.

4.3 Interdependencies of Threads

Another issue that has to be taken care is the interdependencies of threads. The threads may be dependent on each other. Therefore revoking a thread on which a higher priority thread depends will revoke a higher priority thread, which is never desired. In order to tackle this problem the proposed approach scans all the dependencies relation at compile time. Hence if a high priority thread is dependent on any other thread then that thread is also termed as a high priority thread and will never have to wait outside the synchronized section. The dependency relations are found by parsing the class files for object instantiation at compile time. The technical details are covered in the section 6. Due to the restoration of objects in the synchronized section only those threads are affected which are concerned with the objects. This removes the hazard of cascading rollback of threads. Cascading roll back of threads occurs when all the threads, whether high priority or low priority and whether concerned with the resource or not (monitored or not monitored), are roll back because of the preemption of the low priority thread on which it is dependent. This causes the system to halt or have extreme bad performance.

4.4 Deadlock detection

It should be noted that the same technique may also be used to detect and resolve deadlock. The Deadlock results when two or more threads are unable to proceed because each is waiting on a lock held by another. Such a situation is easily constructed for two threads, *T1* and *T2*: *T1* first acquires lock *L1* while *T2* acquires lock *L2*, then *T1* tries to acquire *L2* while *T2* tries to acquire *L1*, resulting in deadlock. Generally, deadlocks may occur among more than two threads, and deadlocking programs are often difficult to diagnose and resolve. As a result, many deployed applications may execute under schedules in which deadlock may occur. Using our techniques, such deadlocks can be detected and resolved automatically, permitting the application to make progress. For mission-critical applications in which running programs cannot be summarily terminated, our approach provides an opportunity for corrective action to be undertaken gracefully.

The proposed method always gives the maximum throughput for the higher priority thread which is a key requirement for the real time systems. Another important positive aspect of this approach is that it is easy to implement and avoid the complex management of read/write logs. The implementation details are provided in the next section.

5. Implementation

The proposed method is implemented on Java platform. We used java thread library for initializing and running the threads only. We do not use any java specific synchronization methods instead threads synchronization in our system is brought up by the use of our own object named *controller*. The controller object in our system is a crucial object that provides system with an interface for using thread. It provides data structures for synchronization of threads and interface for thread execution and termination. This object lies as a middle layer between application programs and java thread library. Our approach is applicable for threads of two priorities: high priority and low priority, because of the nature of the approach as discussed earlier. For multiple priorities levels, the programmer has to classify the different levels of priorities into two levels: high and low. For example, 5 levels of thread priorities can be classified into high and low priorities as shown in Figure 2. This classification will be the work of the programmer at design time of the application. It is obvious that the high priority class must have at most 1 or 2 top most levels in order to have the optimal performance.

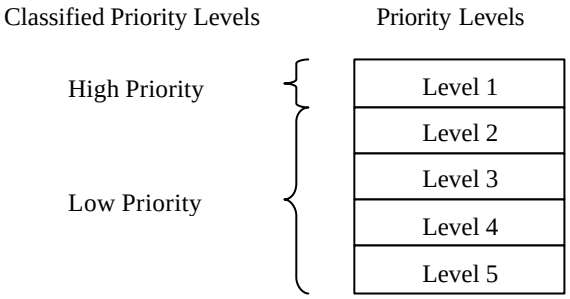

Figure 2: Classification of Priority Levels into High Priority and Low Priority

All the threads whether low-priority or high-priority, are initialized and started by the controller object. Therefore, if an application needs to run a thread of a desired priority, it requests the controller object. Consequently the controller object initializes a thread by using java thread library, set the priority of the thread and finally runs the thread. Similarly the application may also request to run a thread while already one thread is in the synchronized section. Then the controller object will preempt the thread in the synchronized section if the requesting thread is of high priority otherwise put the thread in waiting state. When priority inversion has been detected i.e. a high-priority thread is not allowed to use a resource because of a low-priority thread, the controller object revokes the low priority thread by assigning a true value to a lock on the shared resource. The threads are designed in such a way that each low-priority thread checks the lock variable on resource, periodically. If the lock variable is found false the thread will continue its work in the synchronized section otherwise it will preempt itself. When the low-level thread preempts itself and returns from its run method, the controller object nullifies the low level thread. After the low priority thread has been stopped the controller object starts the high priority thread and the high priority thread now enters the synchronized section. Therefore the higher priority thread has to wait only for a small interval of stopping the low-priority thread. In the mean time the controller object initializes a new low level thread but does not run them. When the high-level thread finishes, the controller object starts the new low level thread and now the low level thread enters the synchronized section. In the next sub-sections some of the special cases that may come up with the proposed approach are described.

5.1 Faulty Revocations

When applying the proposed approach, there may arise some revocations which are not needed. For example, we may have a high-priority thread T_{1H} in the synchronized section. In the mean time if another high priority thread T_{2H} arrives. Then revoking T_{1H} will be a faulty revocation. In the proposed approach T_{1H} will never be revoked. The reason is that the high priority threads unlike low priority thread are not designed to check the lock on the shared resource periodically. Therefore on the arrival of T_{2H}, T_{1H} will remain in the synchronized section and T_{2H} will have to wait outside the synchronized section for T_{1H} to finish.

5.2 Tracing Dependencies

A second case arises when a high-priority thread tries to preempt a thread which is a low-priority thread, but another high priority thread is dependent on the low priority thread. The proposed approach deals with this situation by studying the dependency relationship of threads at compile time. At compile time a parser object parses the class files to find the object instantiations. A thread *T1* is said to be dependent on other thread *T2* if the *T1* has a *T2* object instantiation. For every thread a separate graph is constructed showing the dependencies. Now if a high priority thread T_H is immediately or after some edges is found dependent on a low priority thread T_L then the controller object is informed that T_L must not be revoked.

5.3 Applicability in Deadlock Avoidance

Deadlock is a condition where a thread *T1* is waiting for another thread *T2* to complete while *T2* is also waiting for *T1* to complete or *T2* is waiting for another thread *T3*, which after several levels, is waiting for *T1* to complete. Situations like this cause the system to stop the desired work. On the detection of a deadlock the proposed approach can be used to preempt the thread by restoring the resource so that all the threads involving the resource are restarted and then enter the synchronized section sequentially. This will resolve the deadlock problem. The experiment conducted to prove our approach as a deadlock avoidance technique is discussed in the section 6.1.

6. Experimental Results

We have implemented our approach using a common desktop computer. The experiments were carried over a Pentium IV machine with 256 MB RAM. We illustrate the turn around time of the high-level thread comparing them with the turn around time of low level threads. In Figures 3, 4 and 5 the turn around time of different threads are shown. Turn around time of a thread is calculated as follows:

Turn around Time = Thread Finishing time – Thread Arrival time

The turn around time is scaled to milliseconds by using sleep(100)[1] in all the threads. At first a high priority thread and a low-priority thread are executed in isolation. It is shown as the first instance on the x-axis in Figure 3. Then on the second instance a low priority thread is executed, when the execution is about 50% complete a high priority thread is started. The second instance shown in Figure 3 depicts a small increase in the turn around time of the high-priority thread and a considerable increase in the low-priority thread. The increase in the turn around time of high- priority thread is due to the time to stop and save the low-priority thread for future execution and the time to restore the resource object. We noted that this time for stopping and saving 1 thread is 10 milliseconds. The turn around time for low-priority thread is increased considerably from 120 ms to 180 ms because it is revoked at 50% of its execution, then it remains in waiting state as long as the high–priority thread is not finished. After the execution of the high-priority thread the low priority thread is resumed.

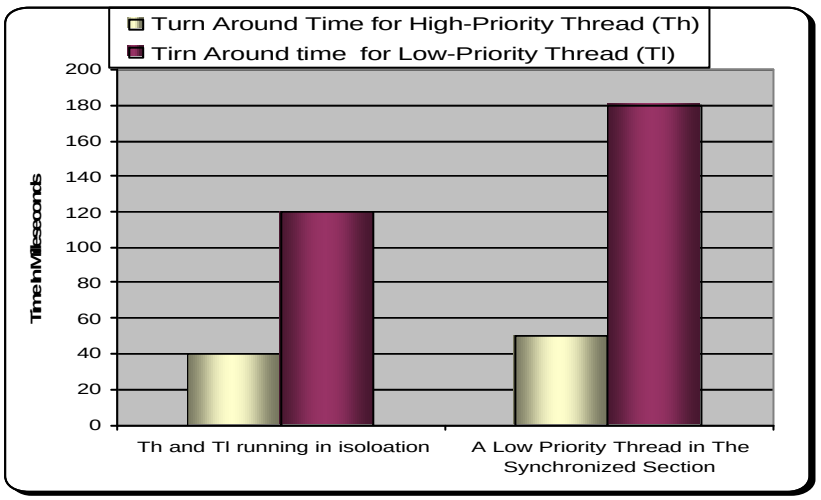

Figure 3: Effect on the turn around time of threads on the arrival of a high priority thread (*Th*) while a low priority thread (*Tl*) is in the synchronized section

[1] sleep(100) is java function that delay execution for 100 milliseconds

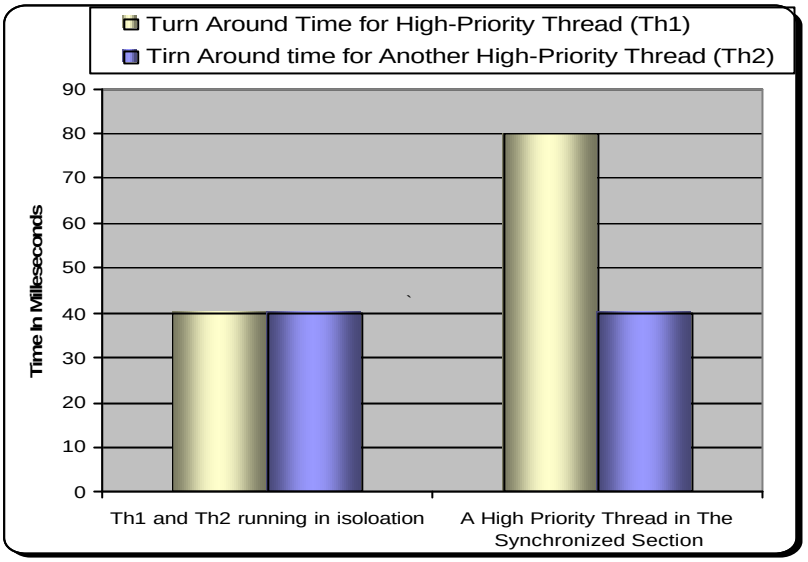

Figure 4: Effect on the turn around time of threads on the arrival of a High priority thread (*Th1*) while another High priority thread (*Th2*) is in the synchronized section

In Figure 4, the turn around time of threads is shown when a high priority thread (*Th2*) is executing in a synchronized section and another high priority thread (*Th1*) arrives. In this situation *Th1* has to wait outside the synchronized section for the *Th2* to finish. Hence the turn around time for the coming thread is increased. This increase is depicted as the second instance in Figure 4. It should be noted that the thread *Th2* is not affected by the arrival of the thread *Th1* and its turn around time remain same as run in isolation.

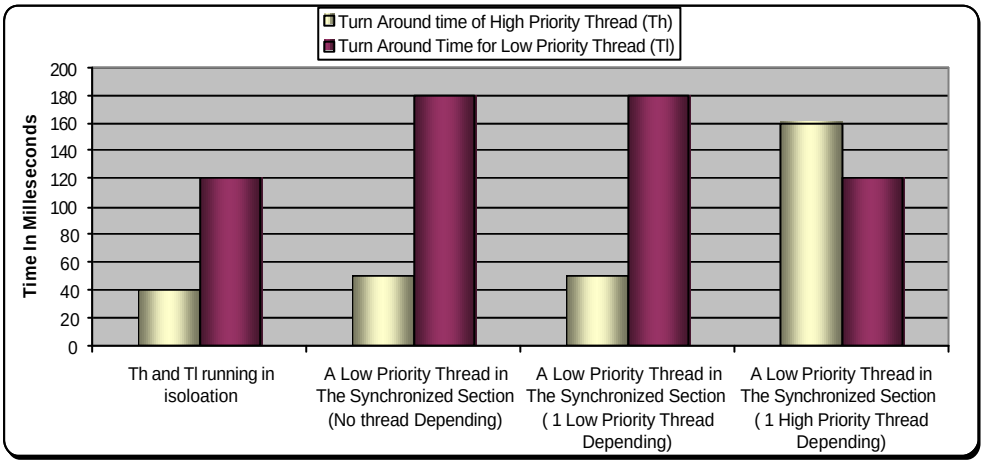

Figure 5: Effect on the turn around time of threads on the arrival of a High priority thread (Th) while another dependent low priority thread (Tl) is in the synchronized section

As discussed earlier there are cases when a thread about to be preempted has some dependencies. In this case the dependency relationship must be studied before preempting a thread. In Figure 5 the experiment is done with a low priority thread *Tl* in the synchronized section having several types of dependency relationships and a high priority thread *Th* wants to use the synchronized section. As the previous experiments the first instance shows the two threads running in isolation. In the second instance no thread is depending on the *Tl* so the case is the same as that in Figure 3. In the third instance another low priority thread is depending upon the *Tl* which is in the synchronized section. Here as *Tl* is preempted by enabling the lock on the shared resource, the low priority thread depending upon the thread will be blocked

but that does not affect the system (refer to the problem of cascading roll back in section 5.2). This transparency of revocation of a dependent thread is evident in third instance of the Figure 5 where it is exactly same as the second instance. In the fourth instance the dependent of *Tl* is a high priority thread. Therefore the thread is not preempted and the requesting *Th* has to wait. This is shown as the increase in the turn around time of *Th*.

6.1 Test for Deadlock Avoidance

We test the result of deadlock avoidance by using a wait-for graph as a deadlock detection technique and our proposed approach as the deadlock avoidance technique. We run three threads of same priority *T1*, *T2* and *T3*. *T1*, *T2* and *T3* acquire the synchronized sections *S1*, *S2* and *S3* for shared resources *R1*, *R2* and *R3* respectively. Now thread T1 needs a shared resource *R2* and therefore stops and starts waiting to enter *S2* i.e. waiting for *T2* to finish. *T2* needs the shared resource *R3* and therefore stops and starts waiting to enter *S3* i.e. waiting *T3* to finish. Later *T3* needs the shared resource *R1* and therefore stops and starts waiting to enter *S1*. Now *T1* is waiting for *T2* and *T2* is waiting for *T3* and *T3* is waiting for *T1*: hence a deadlock is acquired. A wait-for graph of the above thread is depicted in the figure 6. Now applying our algorithm as a deadlock avoidance technique, the algorithm will preempt the thread *T1* by locking the shared resource *R1*. Hence *T1* will be restarted and *T3* will enter the synchronized section *S1* and fulfill its need. Note that *T1* being in waiting state checks the value of lock on its shared resource *R1* periodically. Observing a true value on the lock on *R1*, *T1* makes the lock value false and restart it self by placing it self in the end of the queue of *S1*. Now when *T3* is finished *T2* will enter *S3* and when *T2* finish *T1* will enter *S2*. Hence deadlock is resolved.

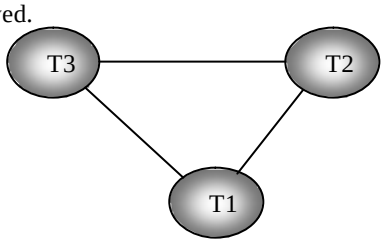

Figure 6: Wait-for Graph for the threads T1, T2 and T3 in deadlock

Figure 7 shows the result of the experiment which depicts the increase in the CPU utilization as the deadlock is resolved. In the first instance it shows the CPU utilization in deadlock. The second instance shows the workload in loading the algorithm and revoking the thread *T1*. In the third instance as the deadlock has been resolved, the CPU utilization does no more increase. The CPU gets busy with running the threads *T3*, *T2* and *T1*.

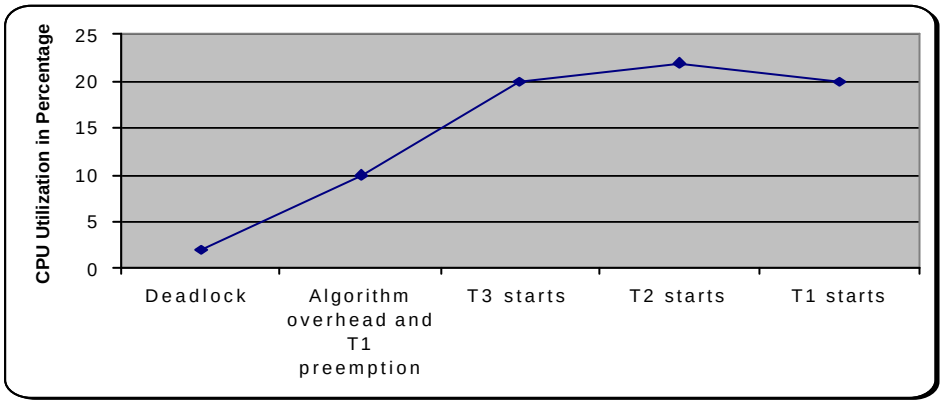

Figure 7: The CPU utilization in percentage as the deadlock is being resolved

7. Conclusion

Priority inversion is a problem that generally arises in priority scheduling in the context of multithreading. There have been several methods used for avoiding priority inversion which have got their advantages and detriments. Revocation based priority inversion by resource restoration is a novice approach that gives the highest throughput of high priority threads in real time systems and maintains the consistency of the system by backing up and restoring the resource objects. The algorithm eliminates the problem of priority inversion and can be adopted to solve other types of problems like deadlocks. The solution has been implemented in Java and the experimental results are fetched which verify that the approach is very suitable for real time systems where high-priority threads must be served on time. Although being implemented in Java with the usage of java thread library, the algorithm does not rely on java or java thread library. The algorithm did not use any of the java thread synchronization APIs for synchronization. It uses its own data structures for starting and stopping threads. Hence the algorithm is a general idea which can be implemented for any multi threaded environment with synchronized section capability. Although our preliminary experiments are encouraging, we believe there are numerous opportunities to improve the performance of our design. One of the basic limitations of our system is that the affects of the low level thread before revocation may never be eliminated properly by restoring resources. For example, the affects on resources such as printers and console output can never be rolled back therefore low level threads involving such resource if revoked and restarted may result in inappropriate results. Hence proper mechanism for revoking such threads may be adopted. Secondly there are some system resources which a low level thread might acquire. These resources also may not be backed up and restored.

Acknowledgment

We would like to thank King Fahd University of Petroleum and Minerals for supporting this research work and providing of the computing facilities. Special thanks to anonymous reviewers for their valuable comments on this paper.

References

[1] Adam Welc, Antony L. Hosking and Suresh Jagannathan, "Preemption-Based Avoidance of Priority Inversion for Java", IEEE Proceedings of the International Conference on Parallel Processing (ICPP'04), 15-18 Aug, 2004 pp.:529 – 538, Vol.1.

[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] 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

[4] Kim, K.H.(Kane), "Basic Program structure for Avoiding Priority Inversion", IEEE Proc. Of 6th International Symposium on Object-oriented Real- time Distributed Computing (ISORC03), Hakodate, Japan, May 2003, pp. 26-34.

[5] Christoph Schuler, Roger Weber, Heiko Schuldt, Hans-J. Schek, "Peer-to-Peer Process Execution with Osiris," in LNCS proceedings of the 1st international conference on service oriented computing (ICSOC03), Italy, pp.483-489, Vol. 2910.

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

[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] Verhofstadt, J. "Recovery technique for Database Systems", ACM Computing Surveys, 10:2. June 1978.
[11] Yodaiken, V. "Against priority inheritance" Finite State Machine Labs (FSMLabs) Technical Report, June 25, 2002: http://www.fsmlabs.com/against-priority-inheritance.html