Volume 3, Issue 4, April 2013 ISSN: 2277 128X

International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

Reducing the Number of Context Switches in Multithreading

Tarun Sharma
SITE, VIT University
India.

Tapsi Sharma
SITE, VIT University
India.

Sweta Bhattacharya
SITE, VIT University
India.

Abstract— Multithreading is an important technique for computer-based systems. But one of the main issues in the use of multithreading is the cost of context switching. In real time operating systems preemption is one of the main causes of the number of context switches, the overhead caused during the run time and large number of memory requirements. This paper mainly focuses on reducing the number of context switches which directly reduces the pitfalls associated with namely issues like less system responsiveness and low system throughput. The techniques that are highlighted in the paper for reducing the preemptions (which in turn reduces the number of context switches associated with it) are some of the algorithms stated by making use of Round Robin priority based Scheduling and Shortest Job First priority based Scheduling. Apart from the renowned and popular algorithms, Limiting Value Preemption (LVP) algorithm is also used in the scenario of real time operating system. These techniques proved to be an improvement over the existing Round Robin priority based scheduling. The results reveal reduction in context switches by considerable percentage and increases system responsiveness.

Keywords— Context Switches, Preemptions, Limiting value preemption, System responsiveness.

I. INTRODUCTION

In the present fast moving world, all real time applications use multithreading techniques. Multithreading is a technique used for processor allocation in the real time systems. The applications (or the processes) are divided into pieces called threads and the "kernel" helps in their execution. A multithreaded program consists of two or more parts that run concurrently. Multithreading is the proficiency of an operating system to simultaneously run programs that have been divided into subprograms or threads. Thus it can be said that multithreading is a type of multitasking. Multitasking can be divided as process-based and thread-based multitasking. In *the process-based* multitasking the computer runs two or more programs concurrently. Whereas in the *thread -based* multitasking, a thread acts as a single unit of code, means a single program can perform two or more tasks. Multitasking threads have less number of overheads than multitasking processes. Processes are heavyweight tasks that generally require their own address spaces. Moreover the context-switching from one process to another is costly. Inter-process communication is expensive and limited. Threads, on the other hand, are lightweight. They share the same address space. In the thread based system, the inter-thread communication is quite inexpensive, and context switching from one thread to the next is cost efficient. The paper mainly focuses on reducing the number of context switches occurring mainly due to preemptions .In case of priority based round robin scheduling, a specific task with a higher priority preempt a task having lower priority. The context related to the currently running task needs to be saved in the memory which can be retrieved when the task resumes its execution later. Preemptive scheduling has many advantages over non-preemptive scheduling such as flexibility, improved processor utilization etc. But it results in extra CPU cycles and memory requirements due to unnecessary context switches. *Context switches,* if not properly controlled, may lead to reduced responsiveness, unnecessary delays, energy wastage and extra memory requirements. All these can lead to high overhead in real time embedded systems. In this paper, three techniques or algorithms are used to reduce the context switches and the overhead associated with it. *The first algorithm* uses the Round Robin priority based scheduling to allocate the processor to various threads, the average of the burst time of various threads are calculated and the average value is kept as a new *time quantum* which is finally assigned to each thread. *The second algorithm* named as *Divide and Conquer* is based on reducing the average waiting time with the help of Shortest Job First plus the Round Robin Scheduling algorithm which helps eliminating the drawback of less system responsiveness. The reduction in the average waiting time makes the processes response faster i.e. a subtle increase in the system responsiveness. *The third algorithm* known as the *Limiting value preemption* reduces the number of context switches. It uses a Limiting value and the threads between the limiting value and the thread initiator are not preempted i.e. the preemption for those threads are disabled, whereas it is enabled for the others

II. EXISTING SYSTEM

Various thread scheduling is used to implement the task of distributing the CPU time to each thread. The scheduling algorithm for the existing system used is round robin priority scheduling. The Round Robin priority scheduling is a preemptive scheduling. We assume set of ten threads which is represented by T={T1,T2,T3,T4,T5,T6,T7,T8.T9,T10} and to represent the existing process execution the scheduling algorithm taken is Round Robin Priority based scheduling. A round-robin scheduler employs time-sharing, giving each thread a time slot or *time quantum.* Time slices are assigned

© 2013, IJARCSSE All Rights Reserved

to each thread in equal portions, and various priorities are also assigned to each thread. The kernel executes the highest priority thread first. Furthermore, the execution of the thread having the low-priority is pre-empted. This type of multithreading is commonly used and is called as pre-emptive, round robin priority-based scheduling. Assuming the time slice or the time quantum for each thread is 2ms.

The burst time for each threads {T1,T2,T3,T4,T5,T6,T7,T8,T9,T10} are {3,6,5,2,4,1,4,6,2,4} and the priorities are {1,4,3,2,5,6,8,7,10,9}(Assuming lowest number as the highest priority). To continue further waiting time of each threads (i.e. starting time-Arrival time where Arrival time is considered as 0) is calculated followed by an average waiting time and the Gantt chart.

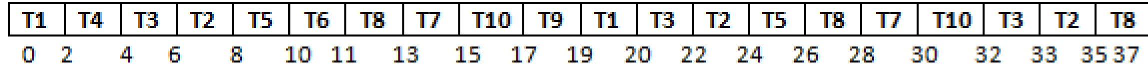

Fig1.The Gantt chart shows the execution of threads using RR priority scheduling.

TABLE I
Waiting time of various threads

Threads	waiting time
T1	17
T2	29
T3	28
T4	2
T5	22
T6	10
T7	26
T8	31
T9	17
T10	28

The waiting time for each thread

Hence the average waiting time=21ms and the context switches are 19. Now, assuming the context switching time of 1ms, the execution of various threads with the Gantt chart is shown in Fig.1

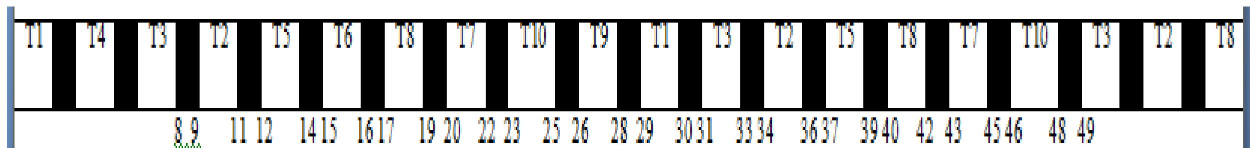

Fig.1 Gantt chart showing the actual execution of the threads with the context switching time of 1ms

TABLE II
Waiting time of various threads

THREADS	WAITING TIME
T1	27
T2	47
T3	45
T4	3
T5	35
T6	15
T7	41
T8	50
T9	26
T10	44

Waiting time using RR priority based scheduling
Considering the context switching time (1ms)

Hence the average *waiting time* =33.3ms and the number of *context switches* are 19.

The problem highlighted in the existing system is the *pre-emption* due to the priority which results in lager number of context switches which further results in high average waiting time. The next section contains the solution to the above methodology.

Implementation of Round-Robin priority based scheduling by considering the average burst time as the time quantum for various threads: Assuming the burst time for each threads as used previously{T1,T2,T3,T4,T5,T6,T7,T8,T9,T10} as {3,6,5,2,4,1,4,6,2,4} and the priorities associated with each thread as

© 2013, IJARCSSE All Rights Reserved

{1,4,3,2,5,6,8,7,10,9}(Assuming lowest number as the highest priority).The time quantum or the time slice is calculated as:

Time Quantum=total burst time/number of threads

i.e Time quantum=(3+6+5+2+4+1+4+6+2+4)/10 which comes out to be 3.7ms taking the approximate value of 4ms. Then the waiting time of each threads (i.e. starting time-Arrival time where Arrival time is considered as 0) is calculated followed by an average waiting time and the Gantt chart is shown in Fig. 2

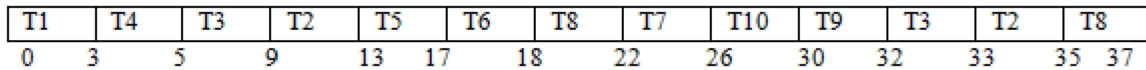

Fig.2 Gantt chart showing the actual execution of threads using the above algorithm

TABLE III

Waiting time of various threads

THREADS	WAITING TIME
T1	0
T2	31
T3	28
T4	3
T5	13
T6	17
T7	22
T8	31
T9	30
T10	26

Waiting time using RR priority based scheduling

The average waiting time is 20.1ms and Hence the number of *context switches =12 as compared to 19 in the existing system.*

Though using the above technique the number of context switches is reduced with the small amount of percentage but preemptive priority based scheduling have **drawbacks** to be considered**.** Preemptive, Round Robin priority-based scheduling is quiet efficient and powerful, but there are pitfalls that can affect system responsiveness and the overall system throughput.

III. DRAWBACKS

i. **System responsiveness** refers to the ability of a system to complete the assigned task within the given time. If the number of preemptions of the threads and the context switches associated are more, then the process is preempted again and again leading to less responsiveness.

ii. Another major issue is the **priority inversion problem.** As per the inversion problem, the higher priority thread is suspended because a lower priority thread currently holds the resource as shown in Fig 3.

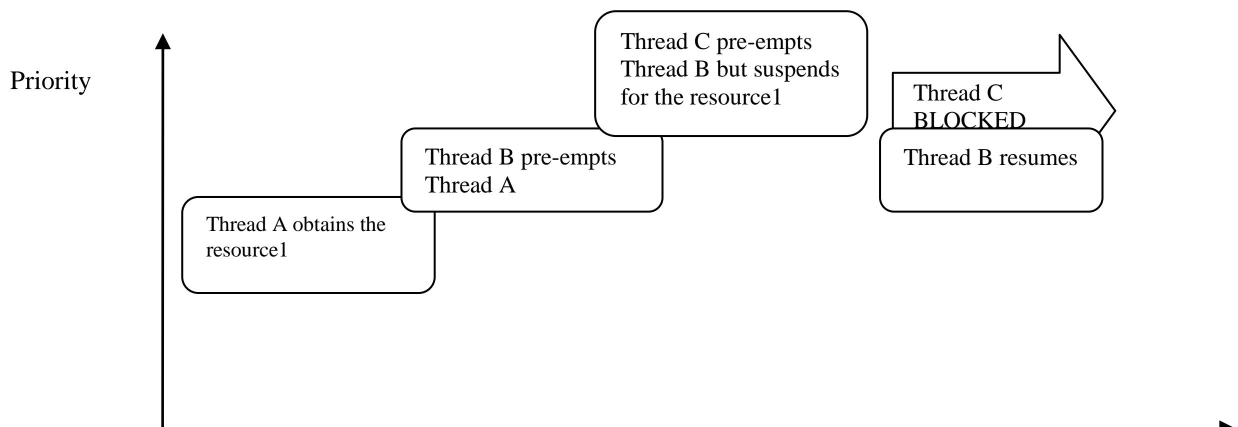

Fig 3. Priority Inversion Problem

The Fig.3 shows that even though thread C has the higher priority it must wait for thread B in order to acquire the resource which is currently being held by the thread B, thus we can say the priorities have become inverted.

iii. Another drawback or pitfall associated with the basic technique, i.e. preemptive priority based scheduling is low system throughput and the solution to this issue is to reduce **the number of context switches.** The kernel performs the context switch between thread1 which is a high priority thread, and the thread2 is a lower priority thread and

© 2013, IJARCSSE All Rights Reserved

context switch takes place from thread2 to thread1. For example considering the three threads A, B, C, where all these threads are waiting for some resource or a message. On receiving the resource or message by A it delivers it to B, which further delivers it to C, which eliminates the message thereafter. The execution depends on their priorities. If all the threads A, B and C had equal priorities then a single context switch occurs. On the contrary if B has a higher priority than A and C has higher priority than B, the context switches becomes twice in number. The solution to this drawback is accomplished by taking into consideration the *"Limiting value"* preemption.

IV. DIVIDE AND CONQUER ALGORITHM

The Divide and Conquer algorithm is implemented to eliminate the drawback related to system responsiveness. Assuming a set of ten threads {T1,T2,T3,T4,T5,T6,T7,T8,T9,T10} with burst times as {3,6,5,2,4,1,4,6,2,4} and also the priorities associated with each thread are{1,4,3,2,5,6,8,7,10,9}(Assuming lowest number as the highest priority).The time quantum or the time slice is considered as 4ms. To start with, the priorities of the threads are sorted in ascending order and after sorting the threads are in the order of {T1,T4,T3,T2,T5,T6,T8,T7,T10,T9}. Then the threads are divided into three checkpoints, named as checkpoint1 (T1, T4, T3, T2), checkpoint2 (T5, T6, T8, T7) and checkpoint3 (T10, T9). (Fig.4.)

Then, the Round Robin scheduling algorithm is applied among the checkpoints with the time quantum of 4ms and in the threads inside the checkpoints the shortest job first scheduling is also implemented.

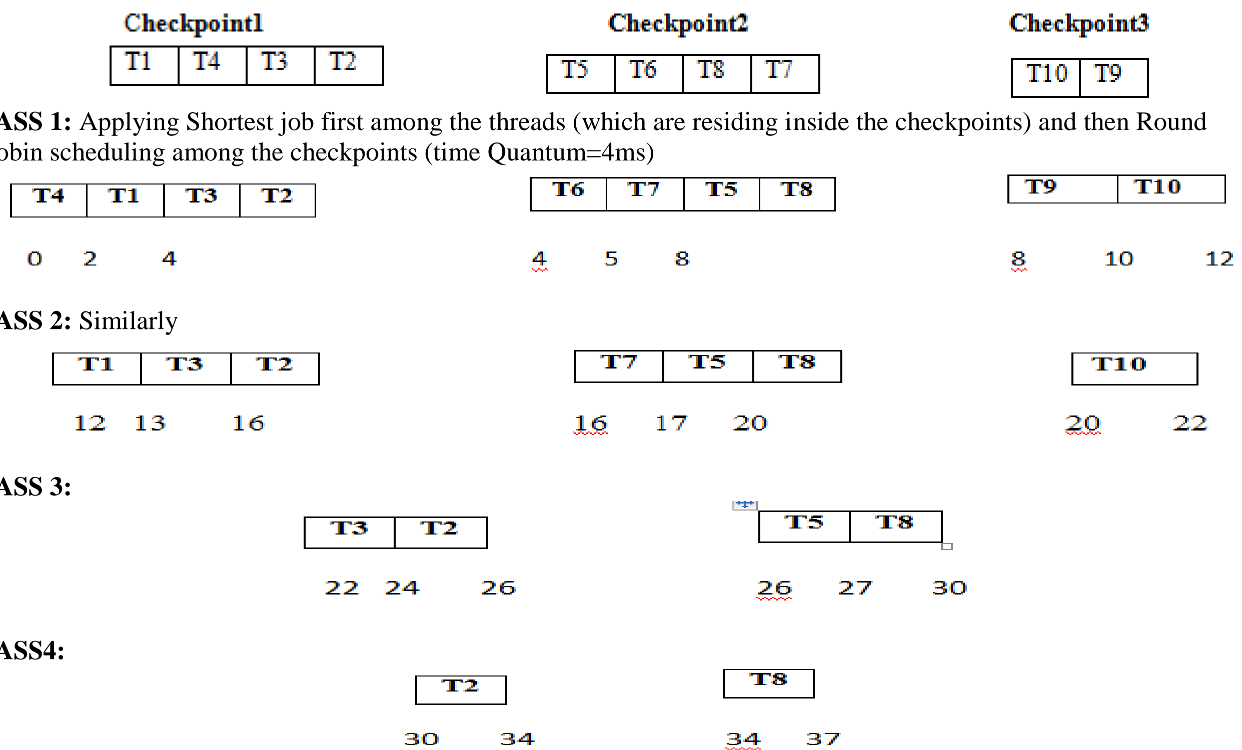

Fig 4. Implementation of Divide and Conquer Algorithm

Now calculating the waiting time of each thread and concluding the average waiting time too.

TABLE IV.
Waiting time of various threads

THREADS	WAITING TIME
T1	10
T2	28
T3	21
T4	0
T5	23
T6	4
T7	13
T8	31
T9	8
T10	18

Waiting time using SJF+RR scheduling used above.

© 2013, IJARCSSE All Rights Reserved

Hence the average waiting time is 15.6ms as compared to the average waiting time in the existing algorithm which was 20.1ms. The conclusion made from the divide and conquer algorithm throws a light on system responsiveness. Considering a thread T6 having burst time of 1ms, in the existing system the thread T6 had to wait for 17ms because it had the priority as 6. But applying this algorithm the waiting time is reduced to 4ms, which is a subtle advantage.

V. RECOMMENDATION OF LIMITING VALUE

The recommendation of the limiting value is the solution to reduce the number of context switches by an ample amount of percentage. A set of ten threads {T1,T2,T3,T4,T5,T6,T7,T8,T9,T10} is assumed having the time quantum or the time slice as 2ms.The burst time assigned to each threads are as {3,6,5,2,4,1,4,6,2,4} and the priorities are {1,4,3,2,5,6,8,7,10,9}. The implementing the existing Round Robin Priority scheduling is shown in Figure1. and the Gantt chart is as

T1	T4	T3	T2	T5	T6	T8	T7	T10	T9	T1	T3	T2	T5	T8	T7	T10	T3	T2	T8

0 2 4 6 8 10 11 13 15 17 19 20 22 24 26 28 30 32 33 35 37

The number of context switches is 19.

Now a 'thread' is selected randomly among the various threads and its priority value is checked. The thread is named as *'initiator thread'*. Assuming the thread picked is T6 and this thread is assigned a *limiting value* from the various thread priorities which is less than the T6 priority i.e. and the limiting value as 3(the priority of thread T3) is assigned. Now the threads having priority value less than this limiting value are allowed to pre-empt and the threads having priority value higher than the initiator thread priority are also allowed to pre-empt but the threads having priority value lying between the limiting value and the initiator thread priority is not pre-empted.

TABLE V.

Priorities	Comment
1	Preemption allowed for threads with priorities 1-2
2	
3	Thread is assigned the preemption limiting value as 3 and threads with priority 3-6 will not be preempted
4	
5	
6	
7	preemption allowed from priorities 7-10
8	
9	
10	

The Gantt chart of the algorithm is shown in Fig.5. The chart indicates that the number of context switches reduced to 14.

T1	T4	T3	T2	T5	T6	T8	T7	T10	T9	T1	T8	T7	T10	T8

0 2 4 9 15 19 20 22 24 26 28 29 31 33 35 37

Fig.5 Gantt chart showing execution of threads using the Limiting value pre-emption algorithm

VI. RESULTS AND DISCUSSIONS

In this section a comparative analysis of the above used algorithms are done with the existing algorithm. Fig.6 shows the average waiting time of threads obtained while implementing the existing algorithm and implementing the RR taking average of burst times as the time quantum. Fig.7 shows the no of context switches in the existing as well as the second algorithm used.

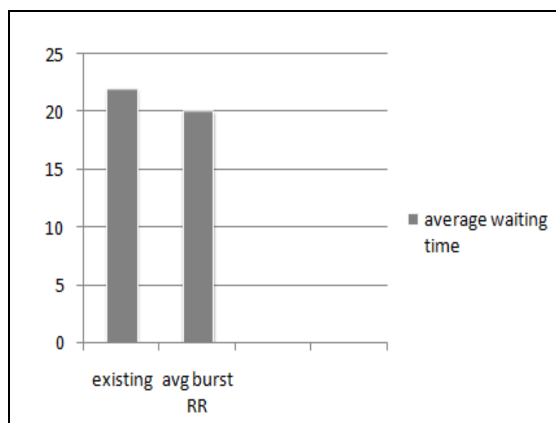

Fig.6 Average waiting time of existing & average burst RR algorithm

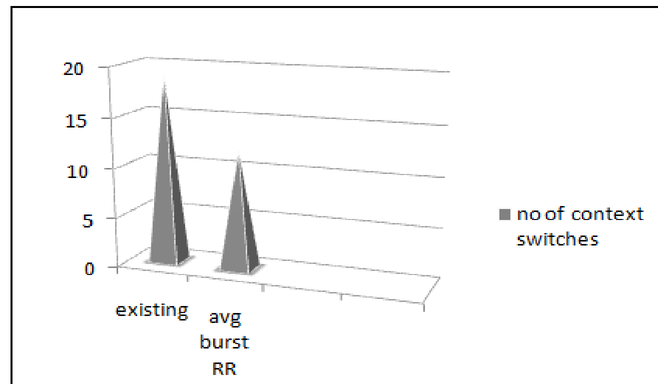

Fig.7 Number of context switches

Similarly Fig.8 shows the comparison of the average waiting time of the existing and the Divide and Conquer algorithm which has been implemented to reduce the pitfall of system responsiveness.

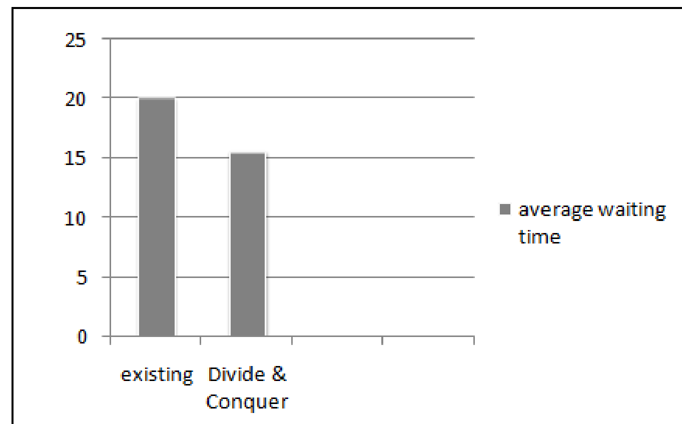

Fig.8 Average waiting times of existing and Divide & conquer

Fig.9 shows the comparison of existing and the limiting value pre-emption algorithm.

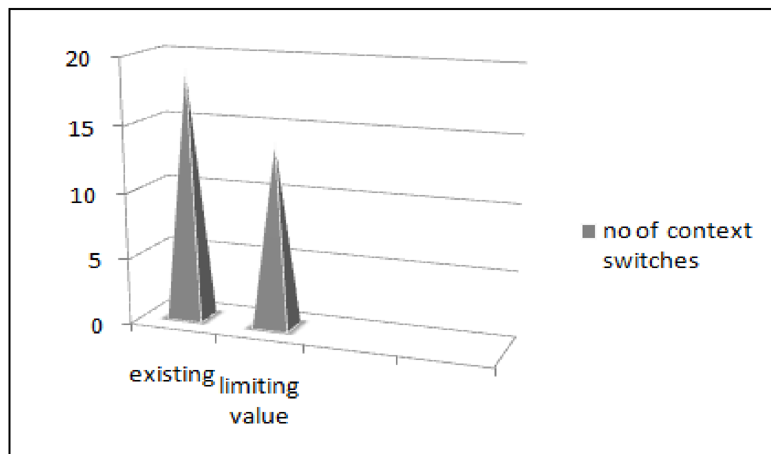

Fig.9 comparison of context switches

VII. CONCLUSION

The paper presents a solution to reduce the number of context switches and the average waiting time by implementing some of the new algorithmic approached and also some of the renowned and popular scheduling algorithms such as Round Robin Priority scheduling and Shortest Job First Scheduling. Also, the various drawbacks associated with multithreading are discussed highlights how these algorithms are implemented to overcome the drawbacks associated with the existing. The research indicates that first algorithm reduces the number of context switches as well as the average waiting time of various threads which results in faster consumption, increased throughput etc. Also, the number of context switches are reduced considerably 36.8%. The paper also proposes the fact that Divide and Conquer Algorithm reduces the average waiting time by 22.38%, thus increasing system responsiveness. The limiting value pre-emption is also proposed such that for half of the threads, pre-emption is disabled which resulted in decrease of number of context switches by 26.316%, thus overcoming the drawbacks associated with the existing system.

© 2013, IJARCSSE All Rights Reserved

REFERENCES

[1]. C.L. Liu, J.W. Layland, "Scheduling algorithms for multiprogramming in a Hard Real Time Environment", Journal of the ACM, vol.20, no.1, 1973.

[2]. H.Chetto and M.Chetto, "Some results of the Earliest Deadline Scheduling Algorithm," IEEE Transactions on Software Engineering, vol. 15, pp.1261-1269, 1989.

[3]. L. Sha, R. Rajkumar and J. P. Lehoczky, "Priority inheritance protocols: an approach to real-time synchronization" IEEE Transactions on Computers, pp. 1175-1185, 1990.

[4]. T.P. BAKER, "Stack-Based Scheduling of Realtime Processes", The Journal of Real Time Systems, vol. 3, pp. 67—99, 1991.

[5]. Too-Seng Tia, Jane W, Jun Sun, Rhan Ha, "A linear time optimal acceptance test for scheduling of Hard Real Time tasks", IEEE Transactions on Software Engineering, 1994.

[6]. William Lamie, "Preemption-threshold", White Paper, Express Logic Inc. 1997, http://www.threadx.com/preemption.html

[7]. Yun Wang and Manas Saksena, "Scheduling Fixed-Priority Tasks with Preemption Threshold", IEEE Sixth International Conference On Real-Time Computing Systems and Applications , 1999.

.

© 2013, IJARCSSE All Rights Reserved