RMDP–A Real-time CPU Scheduling Algorithm to Provide QoS Guarantees for Protocol Processing

R. Gopalakrishnan
gopal@dworkin.wustl.edu

Gurudatta M. Parulkar
guru@flora.wustl.edu

Department of Computer Science
Washington University in St.Louis

Abstract

We present the analysis of a real-time CPU scheduling algorithm specifically designed to provide processing guarantees for multimedia applications. It is motivated by the need to reduce the number of context switch operations in protocol processing while being able to provide bandwidth and delay guarantees to each protocol session. We have implemented this scheme in the NetBSD operating system. Our experimental and simulation results show that there is substantial reduction in the number of context switch operations. We present the scheme and derive its schedulability test.

1 Introduction

One of the basic algorithms for real-time scheduling is the rate monotonic(RM) [3] scheme. RM is fully preemptive i.e. a running thread is pre-empted as soon as a higher priority thread becomes runnable. However, pre-emptions can cause context switching which is an expensive operation in most operating systems. We present a modified scheme called rate monotonic with delayed pre-emption(RMDP) in which a running thread can delay yielding the CPU when a higher priority thread becomes runnable. The RMDP scheme reduces context switches and has other advantages [1] as well. After describing some related work we present the RMDP scheme.

1.1 Previous Work

Pre-emption delay causes higher priority threads to experience *blocking.* [2] derives a schedulability test when blocking is caused by system activity. Blocking caused by a lower priority thread executing in a critical section is discussed in [4]. Both results treat blocking time as system overhead. We present a schedulability test for RMDP that provides a better utilization bound by accounting for the work done by the running thread during the blocking interval.

2 Schedulability test

We use the following notations. Threads are represented as $J_1, \ldots, J_n$. A thread J_i has a period T_i, a CPU requirement (or usage requirement) of C_i, and a quantum of $c_i = f_i C_i$ where $f_i \leq 1$. We denote the maximum of all the c_i values as c_{max}. We assume that threads with smaller subscripts have higher priority. In our analysis we will often compare a running thread J_r that is scheduled according to the modified policy with the corresponding thread J'_r that is scheduled in the normal manner. The attributes of J'_i are the same as that of J_i. The utilization u_i of thread J_i is given by C_i/T_i and the total utilization U is the sum of the utilizations for all threads. We assume that the attributes chosen for the threads are such that they satisfy the schedulability test for RM which states that a set of periodic threads whose deadlines are equal to their periods is schedulable if their total utilization U satisfies $U = \sum_{i=1}^{n} u_i \leq n(2^{1/n} - 1)$.

We first look at the effect of delaying the pre-emption of a running thread, on itself and on higher and lower priority threads. We state the following lemma omitting the proof due to lack of space.

Lemma 1 *Suppose that corresponding invocations of J_r and J'_r are running (in their respective schedulers) at a given time, and a thread with higher priority becomes ready. Suppose J_r delays yielding the CPU by a duration $\Delta_r \leq c_r$ thereby gaining a usage of Δ_r compared to J'_r that yields at once. We claim that by the time J_r resumes execution, J'_r would have resumed and regained exactly the amount Δ_r in usage.*

One implication of this lemma is that the relationship between the CPU usage of J_r and J'_r that exists at the time when a higher priority thread arrives is restored once the higher priority arrival completes its execution requirement and both invocations resume execution. Another consequence of the above lemma is that whether a thread delays its pre-emption or not, the time at which the CPU becomes available to lower priority threads is the same in both cases. Hence we only need to worry about higher priority threads that get delayed because of the running thread not yielding the CPU at once.

In the discussion that follows, let J_r be the thread that delays its pre-emption. The *delay interval* Δ_r is the time between the arrival of the higher priority thread and the end of the current quantum of the running thread. We first identify the set of higher priority threads W_r that are affected only by the delay in pre-emption of J_r and not of any other thread. First we include in W_r those higher priority threads that became ready during the delay interval. We then add to W_r threads that became ready after the delay interval but had to wait for some thread in W_r to complete its task. In general, we define the set $W_r = \{J_{r_1}, \ldots J_{r_i}, \ldots J_{r_m}\}$, and the set W'_r as the set that contains for each invocation J_{r_i} in W_r, its counterpart J'_{r_i} that is scheduled according to the RM policy. For each invocation J'_{r_i}, let the amount of usage that it obtains within the delay interval be d_{r_i}. It can be seen that the sum of these usages $\sum d_{r_i} = \Delta_r$.

Lemma 2 *Consider corresponding invocations in W_r and W'_r. At the end of the delay interval, each invocation J'_{r_i} has got an extra d_{r_i} amount of usage compared to J_{r_i}. We claim that if the usage requirement of each J'_{r_i} is increased by an amount d_{r_i}, then both J'_{r_i} and J_{r_i} complete execution at the same time. In addition, if a thread J'_{r_i} meets its deadline with this increased requirement, then so does the thread J_{r_i}.*

Proof: We prove this by induction on the index i in r_i. We first consider $i = 1$. After the delay interval, the thread J_{r_1} begins execution and at this point J'_{r_1} would have got a usage of d_{r_1} since its arrival.

We first consider the case when there is no higher priority invocation arrival until J'_{r_1} completes. If the usage requirement of J'_{r_1} were to be increased by an amount d_{r_1}, then both J'_{r_1} and J_{r_1} will have the same CPU requirement at the end of the delay interval and they will therefore complete at the same time. If the thread J'_{r_1} meets its deadline and hence J_{r_1} also meets its deadline. Likewise we can show (using lemma 1), that the result holds even if higher priority threads arrive while J'_{r_1} and J_{r_1} are running.

We now assume that the result holds for all invocations upto $J_{r_{i-1}}$. Since $J_{r_{i-1}}$ and $J'_{r_{i-1}}$ complete at the same time, J_{r_i} and J'_{r_i} get the CPU at the same time. At this time the difference in usage between them is exactly d_{r_i}. Therefore the argument for $i = 1$ again applies, and as before holds good even if we consider pre-emptions.

From the above lemma the threads are schedulable under RMDP provided the requirements of threads under RM are increased by specific amounts. Also, if the usage requirement of the thread J'_r is reduced by the amount Δ_r it will complete at the same time as J_r. The utilization test for RMDP can therefore be stated as–

Theorem 1 *The sufficient condition for schedulability under* RMDP *is* $U = \sum_{i=1}^{n} u_i \leq n(2^{1/n} - 1) - c_{max}(1/T_1 - 1/T_n)$.

Proof: The utilization that is unavailable under RMDP is given by

$$U_u = (d_{r_1}/T_{r_1} + \ldots + d_{r_m}/T_{r_m}) - \Delta_r/T_r$$

$$\leq \frac{1}{T_{r_1}}(d_{r_1} + \ldots + d_{r_m}) - \Delta_r/T_r = \Delta_r(1/T_{r_1} - 1/T_r)$$

Since $\Delta_r \leq c_r \leq c_{max}$, $T_{r_1} \geq T_1$, and $T_r \leq T_n$ we have,

$$U_u \leq \Delta_r(1/T_1 - 1/T_n) \leq c_{max}(1/T_1 - 1/T_n)$$

The schedulability condition follows from the above and [3].

3 Conclusions

The improvement over the result in [4] is captured by the term $-c_{max}/T_n$ which accounts for the work done by the running thread in the delay interval. Our proof technique is general and can be applied to other schemes as well.

References

[1] Gopalakrishnan, R., Parulkar, G.M., "Real-time Signals: A Mechanism to Provide Real-time Processing guarantees," Technical Report WUCS-95-06, Dept. of Computer Science, Washington University, March 1995.

[2] Katcher et. al., "Engineering and Analysis of Fixed Priority Schedulers," *IEEE Transactions on Software Engineering*, October 1993.

[3] Liu, C.L. and Layland, J.W., "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment," *JACM*, Vol. 20,No. 1, January 1973, pp. 46-61.

[4] Sha et. al., "Priority Inheritance Protocols: An Approach to Real-Time Synchronization," *IEEE Transactions on Computers*, Vol.39, No.9, Sep 1990, pp.1175-1185.