Multiprocessor Scheduling of Real-Time Tasks with Resource Requirements

Costas Mourlas

Department of Computer Science, University of Cyprus,
75 Kallipoleos str., CY-1678 Nicosia, Cyprus
mourlas@ucy.ac.cy

Abstract. In this paper, we present a new synchronization strategy for real-time tasks executed in a parallel environment. This strategy makes the timing properties of the system predictable since one is able to determine analytically whether the timing requirements of a task set will be met, and if not, which task timing requirements will fail. The proposed synchronization protocol is based on the on-demand paradigm where resources are assigned only when actually required so that the system never wastes unused assignments. Simple formulae for the worst-case determination of task blocking durations as well as for the schedulability analysis of a set of tasks are described. A schedulability analysis example is also presented that illustrates the concepts of scheduling and resource allocation of a set of time critical tasks.

1 Introduction

The term *scheduling of real-time tasks* actually refers to the concept of sequencing the use of any shared resource whose use involves meeting application time constraints [2]. Alternatively, *real-time scheduling* is defined as the ordering of the execution of the tasks such that their timing constraints are met and the consistency of resources is maintained. Thus, the objective of any scheduling algorithm is to find a feasible schedule whenever one exists for a given set of tasks such that each completes its computation before its deadline, and a given set of resource requirements is satisfied. We have to emphasize here that the cpu scheduling is not the main problem of real-time systems but both cpu scheduling and resource allocation.

In this paper, we present a new synchronization strategy for real-time tasks executed in a parallel environment, where each task may have resource requirements, i.e. it can require the use of non-preemptable resources or access shared data. Thus, we focus mainly on the resource allocation part of the scheduling process. This strategy is a continuation of our previous work on synchronization methods for real-time systems [4] where more intelligent techniques and less pessimistic formulae have been investigated and introduced in the current version. The proposed strategy is analyzable and understandable at a high level, meaning that given a set of tasks we know in advance if they will meet their deadlines or not. It is based on the rate monotonic algorithm [6, 1] in the way the priorities

P. Amestoy et al. (Eds.): Euro-Par'99, LNCS 1685, pp. 497–504, 1999.

are assigned to tasks, and which also effectively creates an integrated resource allocation model. Thus, we consider our work as a fixed priority architecture based upon the rate-monotonic scheduling, which is the optimal static priority scheduling algorithm with good performance, low complexity and minimal implementation overhead.

The most well known synchronization protocol used by the rate monotonic algorithm is that of priority ceiling protocol described in [7, 5] where the blocking time is deterministic and bounded. It has many advantages in the uniprocessor environment but its multiprocessor extension, i.e. the distributed priority ceiling protocol [5] is complex enough and requires also the remote procedure call to be supported by the underlying software. In our approach, we try to minimize the implementation requirements and the complexity of the scheduling and synchronization strategy in order to have a more efficient scheme that fits well in the parallel environment. Given a blocking duration B of a task τ with period T, then the ratio B/T is a measure of schedulability loss due to blocking. As we'll see in the following sections, we try to minimize this ratio as much as possible.

2 Assumptions and Notation

In this paper, we consider only hard dealines for critical tasks where also these tasks can require the use of non-preemptable resources or access shared data. To guarantee that critical tasks will never miss their deadlines, they are treated as *periodic* processes, where the period of the process is related to the maximum frequency with which the execution of this process is requested.

We assume that critical tasks are assigned priorities inversely to tasks periods. Hence, task t_i with period T_i receives higher priority than t_j with period T_j if $T_i < T_j$. Ties are broken arbitrarily. Tasks are periodic, are ready at the start of each period and have known, deterministic worst-case execution times. We consider in our analysis that all task deadlines are shorter than (or equal to) the corresponding task periods (i.e. $D_i \leq T_i$). We assume that static binding of tasks to processors is used and for the simplicity of our presentation we also assume that every periodic task is allocated on a different processor/node of the parallel system. Each periodic task can require the use of non-preemptable resources or access shared data. Thus, cpu scheduling is not the main problem at this stage, but since tasks are inter-dependent the main problem is task synchronization and resource allocation. In real-time environments, blocking due to synchronization has to be deterministic in order to have nice analysis properties and a high degree of system predictability.

We assume that every critical section is guarded by a binary semaphore but the strategy is also applicable when monitors or Ada rendezvous are used. We also assume that the locks on semaphores will be released before or at the end of a task. The term "critical section" will be used to denote any critical section between a `P(S)` and the corresponding `V(S)` statement. `P(S)` and `V(S)` denote the indivisible operations *wait* and *signal* respectively on the binary semaphore `S`. The duration of the critical section is defined to be the time to execute the code

between the P and its corresponding V operations when the task executes alone on the processor. A task τ can have multiple non-overlapping critical sections, e.g.

$$\tau = \{\ldots \texttt{P}(\texttt{S}_1)\ldots\texttt{V}(\texttt{S}_1)\ldots\ldots\texttt{P}(\texttt{S}_2)\ldots\texttt{V}(\texttt{S}_2)\ldots\}$$

but not any nested critical section.

3 The Proposed Synchronization Protocol

In this section, we present a new synchronization protocol suitable for synchronizing real-time tasks executing on parallel systems that requires only the message-passing scheme to be supported by the underlying software. Consider a fixed set of n periodic tasks $\tau_1, \ldots, \tau_n$ each one bound to a different processor. Each task is characterized by four components $(CS^i, C^i_{non-cs}, T_i, D_i)$, $1 \leq i \leq n$, where

CS^i is the set $\{CS^i_{j,k} \mid j, k \geq 1\}$ that includes all the (durations of the) critical sections of the task τ_i. $CS^i_{j,k}$ is the k^{th} critical section in task τ_i guarded by semaphore S_j. This notation is necessary because task τ_i may access the semaphore S_j more than once. $CS^i_{.,k}$ denotes the k^{th} critical section if the corresponding semaphore is not of interest. Similarly, $CS^i_{j,.}$ denotes any critical section of task τ_i guarded by semaphore S_j. We define as C^i_{cs} the total deterministic computation requirement of task τ_i within its critical sections, i.e $C^i_{cs} = \sum_{x \in CS^i} x$,

C^i_{non-cs} is the deterministic computation requirement of task τ_i outside its critical sections,

T_i is the period of task τ_i,

D_i is the deadline of task τ_i.

We also use the notation C_i as the total deterministic computation requirement of task τ_i where $C_i = C^i_{cs} + C^i_{non-cs}$. Moreover, each semaphore S_j can be either *locked* by a task τ_i if τ_i is within its critical section $CS^i_{j,.}$ or $free$ otherwise.

Suppose that a task τ_i requires to enter its critical section $CS^i_{j,k}$ issuing the operation $P(S_j)$. Then the following cases can occur:

1. The semaphore S_j is $free$. Then, the semaphore S_j is allocated to the task τ_i, the task τ_i proceeds to its critical section and the state of S_j becomes *locked*.
2. If case 1 does not hold, i.e. semaphore S_j is currently *locked*, then after its release it is allocated to the highest priority task that is asking for its use. The task τ_i will proceed to its critical section if and only if semaphore S_j has been allocated to τ_i.

Note that by the definition of the protocol, a task τ_i can be blocked by a lower priority task τ_j, only if τ_j is executing within its critical section $CS^j_{l,.}$ when τ_i asked for the use of the semaphore S_l.

Theorem 3.1 The proposed synchronization protocol prevents deadlocks.
Proof: By definition, for every task τ_i there is no any nested critical section. Thus, τ_i will never ask in its critical section for the use of any other semaphore and so a blocking cycle (deadlock) cannot be formed. □

We can easily conclude that a set of n periodic tasks, each one bound to a different processor $\wp$ can be scheduled using the proposed synchronization protocol if the following conditions are satisfied:

$$\forall i, 1 \leq i \leq n, \quad C_i + B_i \leq D_i \tag{1}$$

Once B_is have been computed for all i, the conditions (1) can then be used to determine the schedulability of the set of tasks.

4 Determination of Task Blocking Time

Here, we shall compute the worst-case blocking time that a task has to wait for its resource requirements. Task τ_i requires in every period T_i to enter ν different critical sections $\{CS^i_{.,j} \mid 1 < j \leq \nu\}$. For instance, each time that τ_i attempts to enter its critical section $CS^i_{l,m}$ guarded by S_l, it can find that S_l is currently held by another task. Thus, τ_i blocks and waits at most $B^i_{l,m}$ time units until the semaphore S_l has been allocated to τ_i. In other words, $B^i_{l,m}$ is the worst case blocking time that task τ_i waits to enter its m^{th} critical section guarded by S_l. The total worst-case blocking duration B_i experienced by task τ_i is the sum of all these blocking durations, i.e. $B_i = \sum_{1 \leq j \leq \nu} B^i_{.,j}$. If the blocking time B_i for any task τ_i is non-deterministic then the whole system becomes unpredictable and difficult to analyze. In real-time environments, the blocking has to be deterministic and for this reason our approach imposes a specific structure on blocking to bound the blocking time.

Theorem 4.1 Consider a set of n tasks $\tau_1, \ldots, \tau_n$ arranged in descending order of priority (i.e. $p_i > p_{i+1}$). Each task is bound to a different processor $\wp_i$ and the proposed synchronization protocol is used for the allocation of the semaphores. Let

$H^i_l = \{CS^j_{l,.} \mid 1 \leq j < i\}$, - set of critical sections used by tasks having higher priorities than τ_i and guarded by the same semaphore S_l

$L^i_l = \{CS^j_{l,.} \mid i < j \leq n\}$, - set of critical sections used by tasks having lower priorities than τ_i and guarded by the same semaphore S_l

$\beta^i_l = \max(L^i_l)$. - blocking time due to lower priority tasks

Then, the worst case blocking time $B^i_{l,.}$ each time task τ_i attempts to enter any of its critical sections guarded by semaphore S_l is equal to

$$B^i_{l,\cdot} = \begin{cases} \beta^i_l + \sum_{x \in H^i_l} x + \sum_{y \in \Delta^i_l} y & \text{if } \sum_{x \in H^i_l} x < max(\{T_m \mid CS^m_{l,\cdot} \in H^i_l\}) \\ \infty & \text{otherwise} \end{cases} \quad (2)$$

where Δ^i_l is defined as:

$$\Delta^i_l = \{CS^j_{l,\cdot} \in H^i_l \mid \sum_{x \in H^i_l} x \geq T_j\} \quad (3)$$

Proof: The sum in formula 2 above represents the longest blocking time $B^i_{l,\cdot}$ for a task τ_i trying to enter any of its critical sections $CS^i_{l,\cdot}$ at its worst-case task set phasing. At this worst-case phasing of the tasks, when τ_i wants to enter into critical section $CS^i_{l,\cdot}$ critical section, it finds a lower priority task executing within its critical section guarded by semaphore S_l. Then, just before this lower priority task has finished its critical section and has released semaphore S_l, all the higher priority tasks that use this semaphore S_l come one after the other and ask to enter their critical section.

Thus, $B^i_{l,\cdot}$ has two parts. The first part, namely β^i_l is due to L^i_l and it is equal to the maximum duration of critical section among all the critical sections of the lower priority tasks in L^i_l. The second part comes from tasks in H^i_l. If $\sum_{x \in H^i_l} x$ is less than the minimum period T_{min} of the tasks whose critical sections are in H^i_l then the worst case blocking time $B^i_{l,\cdot}$ equals to the sum $\beta^i_l + \sum_{x \in H^i_l} x$ (note that $\Delta^i_l = \emptyset$). Otherwise, task τ_{min} could block again task τ_i in its next period and due to this fact the element $CS^{min}_{l,\cdot}$ is included in Δ^i_l. The same scenario may happen for all the tasks with critical sections in H^i_l whose periods are less than (or equal to) the value $\sum_{x \in H^i_l} x$ (see definition 3). In all cases, the duration of this sum should be less than the maximum period T_m of the tasks with critical sections in H^i_l, otherwise all these higher priority tasks could block repeatedly the task τ_i and in this case $B^i_{l,\cdot}$ can be prohibitively large or even unbounded (condition of formula 2). Hence the Theorem follows. □

One main implicit assumption has been made throughout our previous analysis. We assumed that the points of time at which the successive instances of a periodic task τ_i request their shared resources, i.e. execute the $P()$ operation, are separated by not less than the task period T_i. This assumption has also been made in the distributed priority ceiling protocol where it is noted that a technique called *Period Enforcer* can be used to ensure that the property holds [5].

The total worst-case blocking duration B_i experienced by task τ_i is the sum of all these blocking durations $\{B^i_{\cdot,j} \mid 1 \leq j \leq \nu\}$. Once these blocking terms B_i, $1 \leq i \leq n$, have been determined, conditions (1) give a fairly complete solution for the real-time task synchronization and scheduling in the parallel environment. It is a fairly complete solution in the sense that the proposed protocol provides sufficient conditions to test whether a given task set is schedulable. The general problem of determining whether a given task set where tasks

Parameters of Task Set					
Task	Period	C_{non-cs}	CS		
τ_1	8	1	$CS^1_{1,1}$	$CS^1_{2,2}$	
			1	1	
τ_2	19	2	$CS^2_{2,1}$	$CS^2_{1,2}$	$CS^2_{3,3}$
			1	4	3
τ_3	24	6	$CS^3_{1,1}$	$CS^3_{3,2}$	
			4	2	
τ_4	27	8	$CS^4_{1,1}$		
			1		

Table 1. Parameters of Task Set in Example

share data can meet its timing requirements is NP-hard even in a uniprocessor environment [5].

We have to notice here that the worst-case blocking duration B of a task is a function of critical section durations only and the impact of B is minimal by letting a lower priority task rather than a higher priority task to experience blocking. In order to have a better view of the proposed synchronization protocol and the schedulability analysis for a set of real-time tasks we'll give an illustrative example.

5 A Schedulability Analysis Example

We illustrate the schedulability analysis based on the proposed synchronization protocol with the following example. This walk-through example is also used to illustrate the fact that the evaluated blocking times of the tasks serve as an upper bound and thus the schedulability of these tasks can be determined.

Example 5.1 Consider a parallel environment and one application which is comprised of four tasks and eight critical sections guarded by semaphores S_1,S_2 and S_3. Each task τ_i is bound to processor $\wp_i$, the periods and computation times within and outside critical sections for each task are listed in Table 1 and execute the following sequence of steps in each period:
$\tau_1 = \{ \ldots,\mathrm{P}(S_1),\ldots,\mathrm{V}(S_1),\ldots,\mathrm{P}(S_2),\ldots,\mathrm{V}(S_2),\ldots,\}$
$\tau_2 = \{ \ldots,\mathrm{P}(S_2),\ldots,\mathrm{V}(S_2),\ldots,\mathrm{P}(S_1),\ldots,\mathrm{V}(S_1),\ldots,\mathrm{P}(S_3),\ldots,\mathrm{V}(S_3),\ldots,\}$
$\tau_3 = \{ \ldots,\mathrm{P}(S_1),\ldots,\mathrm{V}(S_1),\ldots,\mathrm{P}(S_3),\ldots,\mathrm{V}(S_3),\ldots,\}$
$\tau_3 = \{ \ldots,\ldots,\mathrm{P}(S_1),\ldots,\ldots,\mathrm{V}(S_1),\ldots,\}$

The worst-case blocking durations of each task can be determined using the formula 2 as follows:

`Task` τ_1:

- $H^1_1 = \emptyset$ (The set of critical sections used by tasks having higher priorities than τ_1 and guarded by the semaphore S_1)

- $L_1^1 = \{CS_{1,2}^2, CS_{1,1}^3, CS_{1,1}^4\}$ (The set of critical sections used by tasks having lower priorities than τ_1 and guarded by the semaphore S_1)
- $\beta_1^1 = \max(\{CS_{1,2}^2, CS_{1,1}^3, CS_{1,1}^4\}) = 4$
- $B_{1,1}^1 = \beta_1^1 + \sum_{x \in H_1^1} x \ + \ \sum_{y \in \Delta_1^1} y = \beta_1^1 = 4$ since $H_1^1 = \emptyset$ and $\Delta_1^1 = \emptyset$

- $H_2^1 = \emptyset$
- $L_2^1 = \{CS_{2,1}^2\}$
- $\beta_2^1 = \max(\{CS_{2,1}^2\}) = 1$
- $B_{2,2}^1 = \beta_2^1 + \sum_{x \in H_2^1} x \ + \ \sum_{y \in \Delta_2^1} y = \beta_2^1 = 1$

  Thus, the total worst-case blocking duration for task τ_1 is given by the formula $B_1 = B_{1,1}^1 + B_{2,2}^1 = 4 + 1 = 5$.

`Task` τ_2: Similarly working we take the results $B_{2,1}^2 = 1$, $B_{1,2}^2 = 5$ and $B_{3,3}^2 = 2$. Thus, $B_2 = B_{2,1}^2 + B_{1,2}^2 + B_{3,3}^2 = 1 + 5 + 2 = 8$.

`Task` τ_3: $B_{1,1}^3 = 6$, $B_{3,2}^3 = 3$. Thus, $B_3 = B_{1,1}^3 + B_{3,2}^3 = 6 + 3 = 9$.

`Task` τ_4:

- $H_1^4 = \{CS_{1,1}^1, CS_{1,2}^2, CS_{1,1}^3\}$
- $L_1^4 = \emptyset$
- $\beta_1^4 = 0$
- $B_{1,1}^4 = \beta_1^4 + \sum_{x \in H_1^4} x \ + \ \sum_{y \in \Delta_1^4} y = 0 + (CS_{1,1}^1 + CS_{1,2}^2 + CS_{1,1}^3) + (CS_{1,1}^1) = 0 + (1 + 4 + 4) + 1 = 10$ since $\Delta_1^4 = \{CS_1^1\}$ (notice that $\sum_{x \in H_1^4} x \geq T_1$).

  The fact that Δ_1^4 is not empty indicates that task τ_1 can block in its next continuous execution the task τ_4. More precisely, at this worst-case phasing of tasks, τ_1 will ask to enter its critical section $CS_{1,1}^1$ for a second time after exactly eight time units ($T_1 = 8$).

  Thus, $B_4 = B_{1,1}^4 = 10$.

Now that the blocking times for each task have been determined, the next step is to determine whether each task can meet its deadline during execution. Conditions (1) will be used for this test taking into account the $CS_{.,.}^i$ and C_{non-cs}^i values of tasks listed in Table 1 as well as the equation $C_i = C_{cs}^i + C_{non-cs}^i$. Evaluating the conditions we have:

- i=1 $C_1 + B_1 = 3 + 5 = 8 \leq 8 \quad (T_1 = 8)$
- i=2 $C_2 + B_2 = 10 + 8 = 18 \leq 19 \quad (T_2 = 19)$
- i=3 $C_3 + B_3 = 12 + 9 = 21 \leq 24 \quad (T_3 = 24)$
- i=4 $C_4 + B_4 = 9 + 10 = 19 \leq 27 \quad (T_4 = 27)$

We therefore conclude that the above set of tasks $\{\tau_1, \tau_2, \tau_3, \tau_4\}$ is schedulable in a multiprocessor environment where four processors are supported for the execution of the above task set.

6 Conclusions

In this paper, we have presented a fixed priority scheduling mechanism with an integrated resource allocation model that it is analyzable and understandable at a high level, meaning that given a set of tasks we know in advance if they will meet their deadlines or not. This scheduling strategy has been designed for real-time applications that need a parallel computing environment where every real-time task is allocated on a different node or processor and can require the use of global resources. The proposed synchronization protocol places an upper bound on the task blocking duration and once the blocking durations have been computed the conditions (1) can be used to determine the schedulability of a set of tasks.

We have to notice at this point the applicability of the method in other areas except that of real-time systems, like the area of distributed multimedia systems. The proposed scheduling and resource management strategy can be regarded as an important step in meeting the objectives of these systems. A set of distributed multi-media applications running in a network of computers sharing a number of resources such as video and audio units, cameras or file systems can be modelled as a set of periodic tasks that require the use of non-preemptable resources [3]. Hence, with the theory presented in this paper one is able to determine analytically whether the timing and synchronization requirements of a distributed multi-media application will be met, and if not, which requirements will fail.

References

[1] J.P. Lehoczky, L. Sha, J.K. Strosnider, and H. Tokuda. Fixed Priority Scheduling Theory for Hard Real-Time Systems. In A.M. van Tilborg and G.M. Koob, editors, *Foundations of Real-Time Computing: Scheduling and Resource Management*, chapter 1, pages 1–30. Kluwer Academic Publishers, 1991.

[2] C. Douglass Locke. Software Architecture for Hard Real-Time Applications: Cyclic Executives vs. Fixed Priority Executives. *Real-Time Systems*, 4:37–53, 1992.

[3] C. Mourlas, David Duce, and Michael Wilson. On Satisfying Timing and Resource Constraints in Distributed Multimedia Systems. In *Proceedings of the IEEE International Conference on Multimedia Computing and Systems ICMCS'99*. IEEE Computer Society, 1999.

[4] C. Mourlas and C. Halatsis. Task Synchronization for Distributed Real-Time Applications. In *Proceedings of the Ninth Euromicro Workshop on Real-Time Systems*, pages 184–190. IEEE Computer Society, 1997.

[5] Ragunathan Rajkumar, editor. *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[6] L. Sha, M. Klein, and J. Goodenough. Rate Monotonic Analysis for Real-Time Systems. In A.M. van Tilborg and G.M. Koob, editors, *Foundations of Real-Time Computing: Scheduling and Resource Management*, chapter 5, pages 129–155. Kluwer Academic Publishers, 1991.

[7] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. on Computers*, 39(9):1175–1185, September 1990.