Scheduling Algorithms for Real-Time Systems

Fredrik Lindh
Master Program in Computer Engineering
Mälardalens University, Sweden
flh07001@student.mdh.se

Thomas Otnes
Master Program in Computer Engineering
Mälardalens University, Sweden
tos07001@student.mdh.se

Jessica Wennerström
Master Program in Computer Engineering
Mälardalens University, Sweden
jwm07002@student.mdh.se

ABSTRACT

Scheduling algorithms are a governing part of real-time systems and there exists many different scheduling algorithms due to the varying needs and requirements of different real-time systems. The choice of algorithm is important in every real-time system and is greatly influenced by what kind of system the algorithm will serve. Architectures of today cover both uniprocessor and multiprocessors which have their own challenges and complexities. In this paper we present an overview of some scheduling algorithms for real-time systems on both uniprocessor and multiprocessor systems and mention some that are optimal in each category. This overview also includes minor explanations of important concepts in real-time systems and scheduling.

1. INTRODUCTION

The scheduling algorithm is of paramount importance in a real-time system to ensure desired and predictable behavior of the system. Within computer science real-time systems are an important while often less known branch. Real-time systems are used in so many ways today that most of us use them more than PCs, yet we do not know or think about it when we use the devices in which they reside. Cars, planes and entertainment systems are just some devices in which real-time systems reside, governing the workings of that device while we do not consider that such a system exists within the chosen device. A real-time-system is a computer system in which the key aspect of the system is to perform tasks on time, not finishing too early nor too late. A classic example is that of the air-bag in a car; it is of great importance that the bag inflates neither too soon nor too late in order to be of aid and not be potentially harmful.

The choice of algorithm can greatly influence the behavior of a real-time system and for this reason there are many available algorithms. For the different categories of real-time systems there are specialized algorithms developed and in this paper we will attempt to give an overview of many of the available real-time algorithms.

Other overviews of real-time scheduling algorithms have been presented by Burns[1], Burns and Audsley[2] and by Mohammadi and Akl[3]. Those are somewhat more in depth on some topics then this overview.

The organization of this paper is as follows. In section 2 basic concepts of real-time system and scheduling are explained. Section 3 presents the scheduling algorithms being the main section of this paper. It covers static and dynamic uniprocessor and multiprocessor algorithms presenting some algorithms in each area. The final section, 4, covers summary and conclusions.

2. BASIC CONCEPTS

A scheduling algorithm can be seen as a rule set that tells the scheduler how to manage the real-time system, that is, how to queue tasks and give processor-time. The choice of algorithm will in large part depend on whether the system base is uniprocessor, multiprocessor or distributed.

A *uniprocessor* system can only execute one process at a time and must switch between processes, for which reason context switching will add some time to the overall execution time when preemption is used.

A *multiprocessor* system will range from multi-core, essentially several uniprocessors in one processor, to several separate uniprocessors controlling the same system.

A *distributed* system will range from a geographically dispersed system to several processors on the same board. In a distributed system the nodes are autonomous while in a multiprocessor system they collaborate somewhat more, but this line is not as clear cut as it may sound as similar communication delays will occur.

In real-time systems processes are referred to as tasks and these have certain temporal qualities and restrictions.[4] All tasks will have a deadline, an execution time and a release time. In addition there are other temporal attributes that may be assigned to a task. The three mentioned are the basic ones. The *release time*, or *ready time* is when the task is made ready for execution. The *deadline* is when a given task must be done executing and the *execution time* is how long time it takes to run the given task. In addition most tasks are recurring and have a period in which it executes. Such a task is referred to as *periodic*. The period is the time from when a task may start until when the next instance of

the same task may start and the length of the period of a task is static.

An example, shown in Figure 1, of scheduling can be made using three tasks T1, T2 and T3 with execution time and deadline of (1, 3), (4, 9) and (2, 9) respectively and periods equal to their deadlines. These tasks can be scheduled so that all tasks get to execute before the deadlines.

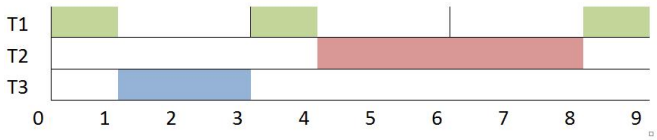

Figure 1: ***Scheduling of T1,T2 and T3.***

A system can consist of many tasks and the example above uses three periodic tasks, but there can also be *aperiodic* tasks which are tasks without a set release-time. These tasks are activated by some event that can occur at more or less any time or maybe even not at all. The scheduling example shows a minor system and the schedule can be made either before the system is activated, which is referred to as off-line scheduling, or during the running of the system and is then referred to as online scheduling. The example is very simple as it does not show priorities or use preemption. There are also other properties of interest when looking at scheduling. Properties a task may use briefly explained:

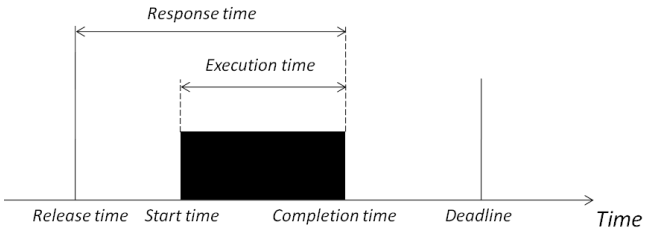

Figure 2: ***The basic temporal attributes of a task.***

- *Release/ready time*: The time a task is ready to run and just waits for the scheduler to activate it.
- *Deadline*: The time when a task must be finished executing.
- *Execution/run time*: The active computation time a tasks need to complete.
- *Worst Case Execution Time (WCET)*: The longest possible execution time for a task on a particular type of system.
- *Response time*: The time it takes a task to finish execution. Measured from release time to execution completes, including preemptions.
- *Priority/weight*: The importance given a task in context of the schedule at hand.

Preemption is when a task that is executing on the processor becomes interrupted, its state is saved and then it is exchanged for another task. This switching of tasks on a processor is referred to as a *context switch* and takes a small amount of time each time it occurs. A preempted task will get to finish executing after the preempting task is done. Not all processors support preemption and algorithms can be divided into preemptive and *non-preemptive* scheduling algorithms. In real-time scheduling preemption is governed by priority.

A real-time system also have requirements based on deadline, a real-time system can either be hard or soft depending on the consequences of missing a deadline.

A *hard* real-time system is never allowed to miss a deadline because that can lead to complete failure of the system. A hard real-time system can be safety-critical and this means that if a deadline is missed it can lead to catastrophically consequences which can harm persons or the environment. It is a crucial requirement that a task starts on time and do not miss the deadline; being not too early nor too late.

In a *soft* real-time system a deadline is allowed to be missed, while there is no complete failure of the system it can lead to decreased performance.

3. SCHEDULING ALGORITHMS

The scheduling algorithms can be divided into off-line scheduling algorithms and online scheduling algorithms. In *offline* scheduling all decisions about scheduling is taken before the system is started and the scheduler has complete knowledge about all the tasks. During runtime the tasks are executed in a predetermined order. Offline scheduling is useful if we have a hard real-time system with complete know-ledge of all the tasks because then a schedule of the tasks can be made which ensures that all tasks will meet their deadlines, if such a schedule exists.

In *online* scheduling the decisions regarding how to schedule tasks are done during the runtime of the system. The scheduling decisions are based on the tasks priorities which are either assigned dynamically or statically. *Static* priority driven algorithms assign fixed priorities to the tasks before the start of the system. *Dynamic* priority driven algorithms assign the priorities to tasks during runtime.

3.1 Uniprocessor

Scheduling algorithms on uniprocessors have to ensure that all tasks in the system are given enough execution time at certain points in time to meet their deadlines if possible. Even if the algorithms presented in the following section only schedules processor time, there exists algorithms taking other resources into account.

3.1.1 Static algorithms

Rate monotonic (RM) scheduling algorithm[5, 6] is a uniprocessor static-priority preemptive scheme. The algorithm is static-priority in the sense that all priorities are determined for all instances of tasks before runtime. What determines the priority of a task is the length of the period of the respective tasks. Tasks with short period times are assigned higher priority. RM is used to schedule periodic tasks.

The following are preconditions for the rate monotonic algorithm formalized by Liu and Layland [5].

1. Periodic tasks have constant known execution times and are ready for execution at the beginning of each period(T).
2. Deadlines(D) for tasks are at the end of each period:
$$(D = T)$$
3. The tasks are independent, that is, there is no precedence between tasks and they do not block each other.
4. Scheduling overhead due to context switches and swapping etc. are assumed to be zero.

There exists a scheduling algorithm similar to RM called *deadline monotonic* (DM)[7], used when D < T which allows us to see RM as a specific case of DM. In the case of the DM algorithm the deadline determines the priority of the task; the shorter the deadline the higher the priority. Both RM[5] and DM[7] are optimal static scheduling algorithms. It has been shown[5] that for an optimal static priority driven algorithm an upper bound for the processor utilization is 70 percent for large task sets.

Example of RM, see Figure 3: Tasks T_1 & T_2 with execution times & periods (1, 4) & (2, 6). T_1 with a shorter period & therefore higher priority runs before T_2. They then run as they are released.

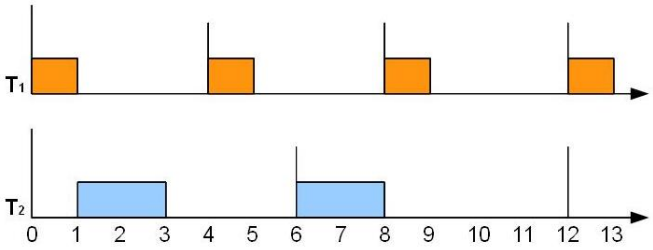

Figure 3: ***Scheduling example of Rate-Monotonic.***

3.1.2 Extensions of RM

There exist some extensions for RM to make it a more useful algorithm. The extensions make it possible to use RM when tasks share resources. To prevent simultaneous use of shared resources a technique called semaphore, which is a sort of lock, is used which a task must take before entering a critical section and then must release after leaving the critical section. A critical section is a piece of code accessing a shared resource. When using semaphores problems with blocking can occur. Blocking is when a task is prevented from executing by tasks with lower priority. This occurs when a task has already gained access to a semaphore and a task with higher priority wants it. Then the task with higher priority needs to wait until the semaphore becomes available. During that time other tasks can preempt the lower priority task and execute as long as they do not require that semaphore and thus the higher priority task is delayed even further. This is called priority inversion, when a high priority task in essence becomes the task with the lowest priority. Different protocols for solving this problem are explained next.

Priority inheritance protocol (PIP)[8] dynamically changes the priority of a task if it is blocking a higher priority task. The priority is then set to the priority of the task it is blocking. This is referred to as inheriting the priority. The amount of times a high priority task can be blocked is the number of semaphores it will use. So if a task will use m semaphores it can in worst case be blocked m times per execution. A problem with this protocol is that it does not prevent deadlock. Deadlock is when some task holds at least one semaphore and still tries to lock another semaphore taken by another task in a similar state and this will cause the system to halt.

Priority ceiling protocol (PCP)[8] avoids deadlock by having each semaphore assigned a priority ceiling which is the priority of the highest priority task that uses it. There are also two restrictions that a task needs to fulfill which is; not hold semaphores between instances and lock semaphores "pyramidically". To be able to preempt a task must have higher priority than all used semaphores. Any holding task will inherit the priority of a task becoming blocked but the priority ceiling will prevent a task from being blocked more than once per execution.

1. Task T wants to lock semaphore S.
2. If the priority of T is higher than the highest priority ceiling of all the currently locked semaphores, or if no semaphores are locked, then T is allowed to lock S; else it becomes blocked.
3. If T however is blocked by another semaphore S' locked by task T', T' then inherits the priority of T.
4. When T' releases semaphore S', T gets to execute and lock S.

Immediate inheritance protocol (IIP) is based on PCP and uses priority ceilings for semaphores but differs in that as soon as a semaphore is taken by a task that task inherits the priority of the semaphore. In both PCP and IIP, in contrast to PIP, deadlock is prevented and a task can only be blocked one time. This means that if a task will use m semaphores it can in worst case be blocked one time per execution.

3.1.3 Dynamic algorithms

Earliest deadline first (EDF)[2, 5] is a dynamic priority driven scheduling algorithm which gives tasks priority based on deadline. The preconditions for RM are also valid for EDF, except the condition that deadline need to be equal to period. The task with the currently earliest deadline during runtime is assigned the highest priority. That is if a task is executing with the highest priority and another task with an earlier deadline becomes ready it receives the highest priority and therefore preempts the currently running task and begins to execute. EDF is an optimal[5, 9] dynamic priority driven scheduling algorithm with preemption for a real-time

system on a uniprocessor. EDF is capable of achieving full processor utilization[5].

Least laxity first (LLF)[2, 10], also known as least slack time, is a dynamic priority driven scheduling algorithm that assigns priority based on the laxity. The definition of laxity is the tasks deadline minus the remaining computation time of the task. It can also be described as the maximum time a task can wait before it needs to execute in order to meet its deadline. The task with the currently least laxity is assigned the highest priority and is therefore executed. The executing task will be preempted by any other task with a currently smaller laxity. When the task is executing the laxity remains constant. If two tasks have similar laxity they will continually preempt each other and it will therefore create many context switches. But if we ignore the cost of the context switches LLF is also, as EDF, an optimal[10] dynamic scheduling algorithm.

3.1.4 Serving aperiodic tasks

There are systems where algorithms for scheduling of both periodic and aperiodic tasks are needed. Due to the unpredictable arrival of aperiodic tasks it is harder to guarantee a response time. The server algorithms presented improve the average response time for aperiodic tasks and are based on RM, although similar approaches based on EDF exists. One type of task that is generally not handled well is sporadic tasks. A *sporadic task* is an aperiodic task with a hard deadline, WCET and a known minimum time between permitted arrivals of the task called *interarrival time*. Assigning a server high priority will improve its response time.

The background server algorithm[11] is one simple approach for servicing soft deadline aperiodic tasks. If there are ready periodic tasks they get priority over aperiodic tasks to run; only when the processor is idle, no periodic tasks are executing, aperiodic arrivals are served in the background. This implies that a high load of periodic tasks result in bad response times for aperiodic tasks.

The polling server (PS) algorithm[11] is another aperiodic server algorithm which is a bit more sophisticated. Aperiodic tasks are handled by a periodic task, the PS. Arrivals of aperiodic tasks are served at the beginning of each of the PS's periods, if there are no request pending the server suspends itself allowing periodic tasks to execute. A late arrival, after suspension of the PS, will be queued and served at the next task invocation.

The deferrable server (DS) algorithm[6] is an algorithm with focus on quick response times for aperiodic tasks. The algorithm improves the average response time for aperiodic tasks compared to the background and polling server. The DS is a special kind of periodic task with period and capacity, which serves aperiodic tasks and can do so until available capacity runs out, or the end of the period is reached. Regular periodic tasks are ready at set times whereas the DS may receive requests at anytime during the period and therefore executes at different times. When the DS serves a request capacity is consumed else capacity is preserved. However, any remaining capacity at the end of the period is lost and the DS is replenished at the beginning of the next period.

Assigning the server the highest priority is preferable if we want good responsiveness and ensure that aperiodic tasks meet their deadlines, while assigning the DS medium priority aperiodic tasks might miss their deadlines as other tasks might preempt the server. A drawback is that the algorithm violates the first assumption of RM since it allows the server to defer its execution, thus it may not be ready to execute at the beginning of the period.

An example of DS illustrated by Figure 4. The tasks, T_1 & T_2, have the attributes presented in the RM example. The deferrable server is assigned medium priority. At time two an aperiodic request arrives. Since the server has medium priority, T_2 is preempted and the server serves the request. The server capacity is exhausted by the aperiodic task which executes for two time units. T_1 becomes ready at time four and executes for one time unit delaying the execution of T_2 even further. At time five T_2 is allowed to finish and the server replenishes its capacity. In the second period of T_2 there are no higher priority tasks ready so it is allowed to execute immediately after release. After T_2 has finished executing, an aperiodic request occurs. The server is not allowed to serve this request since T_1 is also ready. When T_1 is finished the server will serve the request.

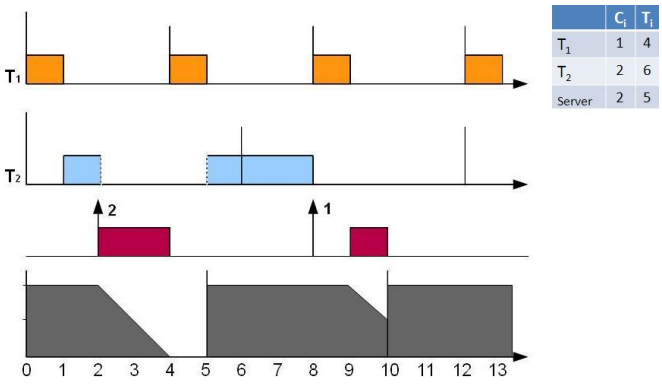

Figure 4: ***Scheduling example of Deferrable Server. C_i is execution time and T_i is the period.***

The priority Exchange server (PE)[11] is another capacity preserving scheduling algorithm that uses a periodic server. The algorithm resembles the DS algorithm but differs in how capacity is preserved. The server replenishes its capacity at the beginning of each period. Aperiodic tasks waiting to be served at the start of a new period will be executed at the priority of the server and consume capacity, assuming the server currently has the highest priority. On the other hand, if there are no aperiodic tasks ready for execution the server allows a ready lower-priority periodic task to execute in exchange for accumulation of capacity at the priority level of that periodic task. Whenever an aperiodic task requests to run capacity available for the server at the task with the highest priority, amongst those with which capacity has been exchanged, will be consumed by the aperiodic task. This type of capacity exchange continues on lower levels with periodic tasks, server capacity will therefore not be lost, just

stored at lower priority levels or consumed by aperiodic requests unless at some point the capacity is exchanged with the idle task.

The DS algorithm is not as complex as the PE algorithm due to the way capacity is preserved at the priority of the server. The price to pay for this simplicity is a lower schedulability bound than PE. In order for both algorithms to function properly particular resource utilization has to be reserved for the server. The server utilization, U_S, which is the ratio of execution time to the period, directly affects the schedulability of the system. The highest utilization bound for periodic tasks at which the periodic tasks can be scheduled, U_P, is determined by RM.

$$DS : U_P = ln\frac{U_S + 2}{2U_S + 1}$$

$$PE : U_P = ln\frac{2}{U_S + 1}$$

From the equations we see that for any given value, Us, where $0 < U_S < 1$, the schedulability bound, U_P, is lower for the DS algorithm than in the case of the PE algorithm. Another implication is that for a given Up the server utilization is lower for the DS algorithm than for the PE algorithm.

From a server perspective scheduling sporadic tasks means that the server is required to have a period less or equal to the interarrival time of the sporadic task and not consume more execution time than the capacity available.

The sporadic server (SS)[11], like the DS, consists of a periodic server for aperiodic tasks but how it replenishes capacity differs. The server determines at which time in the future capacity will be replenished depending on when aperiodic requests occur and the priority of the current executing periodic tasks.

The explanation of the algorithm involves the following terms:

- P_{exe} the priority level of the currently executing task in the system.
- P_S the priority level of the sporadic server.
- *Active* is used to describe the priority level when $P_{exe} > P_S$.
- *Idle* is the opposite of active, $P_{exe} < P_S$.
- RT_S is the time at which the server replenishes consumed capacity.

The sporadic server starts with fully replenished capacity. Whenever the server becomes active RT_S is set to the current time added to the period of the server. How much the server should replenish at RT_S is determined when the server becomes idle or all the capacity has been consumed. The amount to replenish is the capacity consumed from the point the server was activated to the point it becomes idle or runs out of capacity. The sporadic server performs better than the background server and the polling server. The performance is comparable to the DS and PE algorithms although they are in some cases inferior to the SS algorithm.

3.2 Multiprocessor

In multiprocessor scheduling, algorithms developed for uniprocessor scheduling can be applied if we consider each core of the multiprocessor as an isolated core, which is a uniprocessor. However, in multiprocessor scheduling the difficulty of verifying that the execution of different tasks on multiple cores does not interfere with each other and also determining which tasks should be given to a certain core increases the complexity greatly compared to uniprocessor scheduling. Moreover scheduling algorithms for multiprocessors often involves heuristics to simplify the task of finding a feasible schedule. The two main approaches for scheduling algorithms on multiprocessors are global scheduling and partitioning scheduling.

Global scheduling algorithms put ready tasks in a queue sorted based on priority. The task with currently highest priority, which is first in the queue, is selected by the scheduler and will execute on one of the processors and may be preempted or migrated if necessary.

In *partitioning scheduling algorithms* each task is assigned to one processor and will exclusively execute on that processor. This will result in that instead of being a multiprocessor problem it will be a set of uniprocessor problems. Uniprocessor algorithms can then be used for each processor which is an advantage, but the problem of partitioning the task to the processors is hard and usually solved by using non-optimal heuristic. Also as shown by [13] there exist systems which are schedulable if and only if tasks are not partitioned.

The myopic algorithm[12] is a multiprocessor algorithm that schedules tasks not only for CPU time but also for shared resource requirements and uses heuristics to simplify the search for a feasible schedule at any given time. As a new task arrives it will be given to one of the processors which will see if schedulability of that task can be guaranteed. If no schedule can be found the task will be sent on to another processor, while it will be kept if a schedule is found that can guarantee that task along with all currently existing tasks on that processor. When a new task is made ready this algorithm looks at:

- Arrival time T_A
- Deadline T_D
- Worst case processing time T_P
- Resource requirements T_R

Premises for tasks are that they are independent, nonperiodic, non-preemptive and use resources in shared or exclusive mode. When scheduling a task the algorithm calculates the earliest start time, T_{EST} for the task based upon when the required resources will be available. The following condition is to be true for every task when a new schedule is made:

$$0 < T_A < T_{EST} < (T_D - T_P)$$

All tasks that are at any one point to be scheduled is placed in a list sorted according to deadline and a schedule is searched for by using a tree structure to find feasible schedules in

which the root is an empty schedule and each leaf is a schedule while not every leaf is feasible. The tasks are inserted into the schedule, which then is one node level down in the tree, one at a time until a schedule is found or a deadline miss will occur. In the case of a miss there will be some back tracking in the tree and a new schedule will be looked at and so forth until a plausible schedule is found. Every time another task is to be inserted there is a heuristic function H, which looks at more than deadline, is called to evaluate a portion of the tasks in the list to find the most appropriate task to insert into the schedule at that point.

The original version of this algorithm looked at all tasks that were not in the schedule every time a task was to be added, but this algorithm only looks to the first few in a sorted list and this near sightedness is what makes the algorithm myopic. Compared to the non-myopic version of this algorithm and to other multiprocessor algorithms of its time the myopic algorithm was an efficient improvement. The myopic algorithm is $O(nk)$ where $1 \leq k \leq n$ depending on the number of tasks to be scheduled at the time as n equals all tasks to be scheduled and k is a subset there of. This then makes the myopic algorithm have $O(n)$ while the original non-myopic algorithm was $O(n^2)$.

Pfair scheduling algorithms[14] are based upon the idea of giving each task access to resources in proportion to the demand for resources it has in comparison to the other tasks scheduled. Tasks are given a new attribute called rational weight ($x.w$) which is defined as execution requirement ($x.e$) divided by period ($x.p$), in which $x.e$ and $x.p$ are integer values and $x.p > 1$. To be schedulable $0 < x.w < 1$ must be true for any task and the sum of all tasks $x.w < m$, where m is the sum of all resources. Tasks are then split into subtasks, that is points where a task may be preempted so another will get access to resources for awhile, and a function is used to determine how to schedule these subtasks such that all requirements are met. In addition there are several rules governing how this is done correctly to get the greatest benefit from this approach. There are at least three known optimal pfair algorithms: PF[14], PD[15] and ER-PD[16].

4. SUMMARY AND CONCLUSIONS

In the simple case, scheduling may seem straight forward and is easy enough to understand, but once more tasks are added it becomes more troublesome to complete a schedule. Introducing several resources makes for more complicated schedules and increases complexity in the algorithms used to create these schedules. The requirements that real-time systems must fulfill are several and equally many are the approaches how to schedule such systems. As mentioned in this paper optimal algorithms exist but their optimality is often only theoretical and not practical in actual systems.

The scheduling of aperiodic and sporadic tasks is more difficult than periodic tasks due to their unpredictability. To overcome this, algorithms based on server tasks for handling aperiodic arrivals are one solution. Algorithms of this kind are an extension to the scheduling algorithm governing the system. All servers presented in this paper are based upon systems working on the rate monotonic algorithm, but there are other server algorithms extending systems based on the earliest deadline first algorithm. Several of the presented server algorithms have a drawback in that they violate a requirement of the rate monotonic algorithm stating that a task should be ready to execute at the beginning of every period. Yet, servers offer an advantage when dealing with aperiodic tasks. Another kind of extension algorithms are protocols of which some are mentioned in this paper. They aid in dealing with mutual exclusion and deadlocks.

This shows that the complexities on uniprocessor systems can be quite high and yet it is far greater on multiprocessor systems which are becoming the commonplace even in small real-time systems. As complexity increases utilization decreases and on multiprocessor systems with static priority scheduling it can be at most 50%[17].

Multiprocessor systems are the future as we see it now, but finding algorithms that takes full advantage of these systems is an arduous task in which much effort has been and is being made by researchers. Future work could be to focus on these new algorithms being produced as well as dynamic based server algorithms.

5. REFERENCES

[1] Burns A.,"Scheduling hard real-time systems: a review" *Software Engineering Journal*, May 1991.

[2] Burns A. and Audsley N., "REAL-TIME SYSTEM SCHEDULING" *Predicatably Dependable Computer Systems*, Volume 2, Chapter 2, Part II. or Department of Computer Science, University of York, UK.

[3] Mohammadi A. and Akl S. G., "Scheduling Algorithms for Real-Time Systems", *Technical Report No. 2005-499*, School of Computing, Queen's University Kingston, Ontario Canada K7L 3N6, July 15, 2005.

[4] Cottet F., Delacroix J. and Mammeri Z., "Scheduling in Real-Time Systems", published by John Wiley & Sons Ltd, Chichester, West Sussex, England, ISBN 0-470-84766-2, year 2002.

[5] Liu C.L. and Layland J.W., "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment" *Journal of the Association for Computing Machinery*, vol. 20, no. 1, pp. 46-61., year 1973.

[6] Strosnider J. K., Lehoczky J. P. and Sha L., "The Deferrable Server Algorithm for Enhanced Aperiodic Responsiveness in Hard Real-Time Environments", *IEEE Transactions on Computers*, vol. 44, no. 1, January 1995.

[7] Leung J. Y.-T., Whitehead J., "On the complexity of fixed priority scheduling of periodic, real-time tasks", *Performance Evaluation*, vol. 2, issue 4, pages 237-250, December 1982.

[8] Sha L., Rajkumar R. and Lehoczky J. P., "Priority Inheritance Protocols: An Approach to Real-Time Synchronisation", *IEEE Transactions on Computers* 39(9), pp. 1175-1185, September 1990.

[9] Dertouzos M., "Control robotics: The procedural control of physical processes", *Proc. IFIP Cong.*, pp. 807-813, year 1974.

[10] Dertouzos M.L. and Mok A.K.L., "Multiprocessor On-Line Scheduling of Hard Real-Time Tasks" *IEEE Transactions on Software Engineering*, vol. 15, no. 12, December 1989.

[11] Sprunt B., "Aperiodic Task Scheduling for Real-Time Systems" Ph.D. Dissertation, Department of Electrical

and Computer Engineering, Carnegie Mellon University, August 1990.

[12] Ramamritham K., Stankovic J. A. and Shiah P.-F., "Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems", *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 2, April 1990.

[13] Carpenter J., Funk S., Holman P., Srinivasan A., Anderson J. and Baruah S., "A Categorization of Real-time Multiprocessor Scheduling Problems and Algorithms", *Handbook of Scheduling: Algorithms, Models, and Performance Analysis*, Edited by J. Y. Leung, Published by CRC Press, Boca Raton, FL, USA, year 2004.

[14] Baruah S.K., Cohen N., Plaxton C.G., and Varvel D., "Proportionate progress: A notion of fairness in resource allocation.", *Algorithmica*, 15:600-625, year 1996.

[15] Baruah S.K., Gehrke J. and Plaxton C.G., "Fast scheduling of periodic tasks on multiple resources", *Proceedings of the 9th International Parallel Processing Symposium*, p. 280-288, April 1995.

[16] Anderson J. H. and Srinivasan A., "Early-Release Fair Scheduling", Department of Computer Science, University of N. Carolina, Chapel Hill, NC 27599-3175, year 2000.

[17] Andersson B. and Jonsson J., "The utilization bounds of portioned and pfair static-priority scheduling on multiprocessors are 50%", Department of Computer Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, year 2003.