Scheduling Mechanisms Reducing Contention Situations in Multimedia Systems

Jörg Werner, TU Chemnitz-Zwickau
Lars C. Wolf, IBM European Networking Center

Abstract: Multimedia applications have time dependancies and require appropriate resource management and scheduling mechanisms. Additionally, such applications have typically large resource requirements, hence, methods to reduce these requirements are desirable. Contention situations occur when the execution of processes overlaps in time. This leads to additional management efforts like context switches and to the increasing demand for resources like memory space. In this paper we present alternative scheduling methods suitable for real-time processes in multimedia systems. These methods serialize the execution of processes in order to reduce the occurence of overlaps. On the basis of measurements the described scheduling mechanisms are evaluated concerning their effectiveness and the required expenses.

1 Introduction

Multimedia applications are time critical and have typically large resource requirements. To process the data packets of a continuous-media stream, resources such as buffer space are necessary. A simple approach for the assignment of that resource is the fixed, non-shared assignment of buffer space to a stream. This way, the buffer space is always available at the arrival of a data packet. However, the buffer space is unused after the processing of a packet has been finished until the next packet arrives. If a different assignment (which allows sharing) is applied, the total buffer space amount needed for all streams is much smaller, hence, potentially more streams can be served [Williamson95].

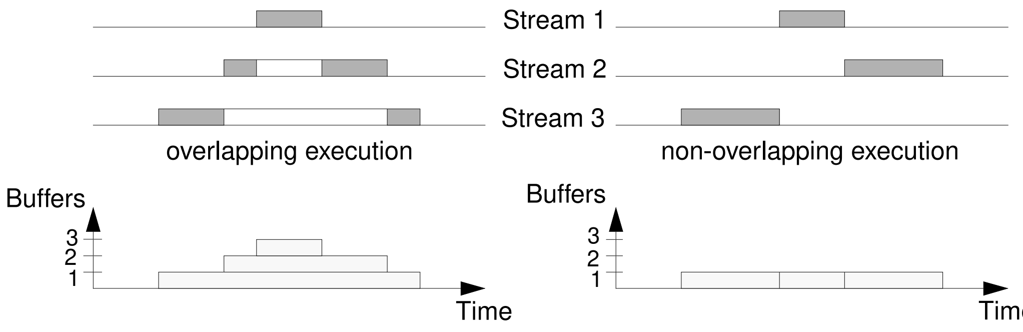

Fig. 1. Resource Utilization.

This buffer space can be shared among streams only if the processing of these streams does not overlap, e.g., due to preemption. This is illustrated in Figure 1, the left side shows the resource requirements if overlapping occurs and the right side presents the case where resources can be shared since no overlapping happens.

In addition to the possibility to apply space sharing strategies, further improvements can be gained if the processing of streams does not overlap. For instance, preemption involves context switch costs, and the execution time for several system mechanisms for synchronization and coordination, e.g., semaphores, is larger due to the management effort if several processes execute them concurrently.

In this paper, scheduling mechanisms are presented and evaluated which reduce the amount of overlapped processing and hence the number of contention situations. A rate-monotonic scheduler is used as the basis for the given mechanisms [Liu73, Wolf96].

The scheduling mechanisms must meet several requirements with respect to correctness and efficiency, i.e., they must:

1. guarantee that no deadlines are missed,
2. avoid overlapped processing whenever possible,
3. use as few resources (CPU time, memory) as possible for scheduling decisions,
4. allow continuous-media processing with as large as possible CPU utilization.

The correctness (1) is the most important item, efficiency aspects are only second. An optimality criterion is given by (2). The scheduling mechanism should not consume all the resource savings itself (3) and it should not reduce the CPU utilization for continuous-media processing (4), compared with the original rate-monotonic scheduler.

2 System Model

Before the examined mechanisms can be described, it is necessary to present the model underlying the design of the mechanisms.

Currently we restrict ourselves to the uniprocessor case. Whether the scheduling methods can be applied successfully to multiprocessors is subject to further investigations.

Each process is periodic and its deadlines are equal to the ends of the periods. When a new period begins, a process is, in principle, ready to run, i.e., its arrival times are known. However, in order to avoid overlappings the scheduler can later change the actual arrival times of a process, so that they do not necessarily coincide with the beginning of the periods (as with the rate-monotonic scheduler). This introduces variability into the start time of the processing of the data packets (jitter), bound by the length of the period. But also for preemptive rate-monotonic scheduling, there exists no guarantee *when* a process executes within a period. It is only guaranteed that processing is finished before the deadline, hence, the worst-case jitter does not deteriorate.

Each process is working with its distinct priority according to the rate-monotonic scheme: the higher the rate of a process, the higher is its priority. The execution time per period given for a process is a worst-case value and includes context-switch overhead so this can be ignored later. All real-time processes are independent and are not influenced by non-real-time processes.

A process never yields the processor voluntarily. This means that a process need not suspend itself to wait for data, e.g., from the filesystem or the network. For the file system, such waiting times, and hence the suspension, can be avoided if a specifically designed continuous-media filesystem such as *Shark* is used [Drake94, Haskin93]. When reading data from a network, such blocking avoidance is not generally possible.

For the time being, only processes that send but not receive, for instance processes inside a video server, can fulfill the requirements given above. They never need to block and have a known behaviour with respect to arrival times. Therefore, the scheduling mechanisms described in the next section are restricted to that class of processes.

Section 4 briefly discusses a method which is able to avoid the blocking of processes when receiving data from the network, but, on the other hand, introduces variable arrival times. Approaches to schedule such processes are presented there, too.

3 Scheduling Methods for Processes with Known Arrival Times

3.1 Non-Preemptive Scheduling

The simplest approach to completely avoid contention situations among processes is to use non-preemptive scheduling, where each process runs without interruptions until its execution for that period has finished. To find out whether a specific process set is schedulable, an appropriate schedulability test must be applied, here, the non-preemptive rate-monotonic method [Nagarajan92].

This scheduling mechanism is optimal with respect to the avoidance of contention situations since at no time more than one process is executing. There are no scheduling efforts during run time because the rate-monotonic scheme assigns priorities in a static manner.

The drawback of this approach is that the possible CPU utilization is potentially lower than using the preemptive rate-monotonic scheduling algorithm. This occurs if a process has a long execution time compared with the periods of other processes. Then even for relatively low CPU utilizations, deadline violations might happen. Such a set of processes would be rejected by the schedulability test since it is not schedulable.

The applicability of this approach depends therefore on the usage scenario. For instance, if the scheduler is used for a video server (which has a processing time of a few milliseconds per period and a much longer period), in most cases non-preemptive execution is possible without deadline violations even for CPU utilizations achievable with preemptive rate-monotonic scheduling.

3.2 Non-Preemptive Scheduling of Processes with Equal Rates

The previous section showed that (for specific sets of processes) non-preemptive scheduling can lead to low CPU utilization. However, a modified method can allow for CPU utilizations equal to those achievable with the preemptive rate-monotonic algorithm. The modification is that the execution of a process is non-preemptive with respect to processes with the same rate, however, processes with higher rates may preempt its execution, i.e., processes are grouped into sets of processes with equal rates and preemption can only occur among processes which belong to different sets.

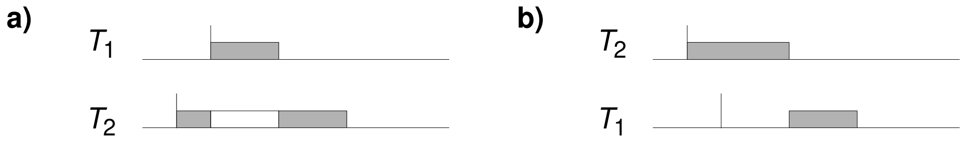

Fig. 2. Possible Execution Sequences of Processes with Equal Rates.

This scheme allows the same maximum CPU utilization as the preemptive rate-monotonic scheduling as explained by the following example. T_1 and T_2 are preemptable processes. In the left side of Figure 2, T_1 has a higher priority than T_2, thus, T_1 preempts process T_2. If the priority order is reverse, i.e., the priority of T_2 is higher than that of T_1, no preemption occurs (right side of Figure 2), and the serialization is achieved without any non-preemptability.

The rate-monotonic scheduling algorithm assigns a unique priority to each process. For processes with equal rates the priority order among them is arbitrary [Liu73]. Therefore, it is permitted to execute T_2 non-preemptively with respect to T_1 independent of the assigned priorities. Since this scheme is still within the conditions of the preemptive rate-monotonic scheduling algorithm, the maximum CPU utilization is the same for both schemes.

Due to the non-preemptive scheduling of processes with equal rates, the execution of such processes never overlaps. Preemptions can now only be caused by processes with a higher rate (and therefore a higher priority). This means that for a process set with n distinct rates at most n–1 preemptions may occur.

The applicability of this scheme depends on the usage scenario, yet, it can be justified as follows. While the processing of continuous-media data is done in principle with different rates, the number of that rates is usually limited. For instance, within video-on-demand applications the data packet transmission is performed with a certain packet rate chosen from a normally small set of rates, hence, the processing of several streams is performed with the same rate. Scaling of streams in regard to their rates will also be possible with a small set of rates only.

A drawback of this method is that it is not optimal with respect to contention avoidance. For a process set with several distinct rates it will generate a schedule that contains overlaps. However, using a different algorithm it might be possible to find a non-overlapping schedule. Further, the algorithm works only well for process sets with a limited number of different rates.

3.3 Modification of Arrival Times

Now a method is presented which offers the same CPU utilization as the preemptive rate-monotonic scheme and avoids overlapped processing whenever possible, i.e., it is optimal with regard to this criterion. The principle approach is to modify all arrival times of the processes individually, hence, processes do not always become ready at the begin of a new period. For instance, process T_1 in Figure 3, which has a higher priority than T_2, does not become ready at t_1 since it would preempt T_2. Instead, the arrival time of T_1 is set to t_2 when T_2 has finished its work, therefore, no preemption can occur.

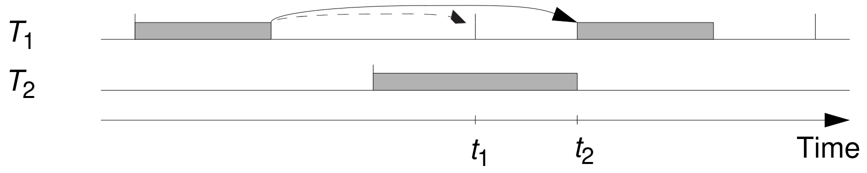

Fig. 3. Modification of Arrival Times.

To apply that technique it must be checked that the delay of T_1 is permitted. It is only allowed to execute a low-prioritized process T_i without preemptions if no higher priority process misses its deadline within that period. Processes with a lower priority than T_i need not to be considered since it has no influence on them whether higher priority processes are executed with or without preemptions. The total CPU requirement is equal in both cases.

This scheme does not change the possible CPU utilization compared with preemptive rate-monotonic scheduling. It uses the laxity of the processes, i.e., the time until the deadline is reached, to inhibit overlapped execution.

The arrival times can be determined either by a static precalculation or dynamically during run time. Both approaches will be discussed in the following subsections.

Modification of Arrival Times – Static Precalculation

The length of the time interval for which the precalculation must be done depends on the periods of the processes. To avoid overlapped processing, it is necessary to examine the current phasing of the periods of all processes at each time instance. It is sufficient to consider an interval with a length given by the lowest common multiple of the period lengths of the processes since it contains all possible period phasings. After the first interval of that length has passed, all further intervals are only replicas of the first one. Hence, the precalculation of the schedule must be done for the length of that interval only. The schedule can be applied repeatedly after each such interval has finished (Figure 4).

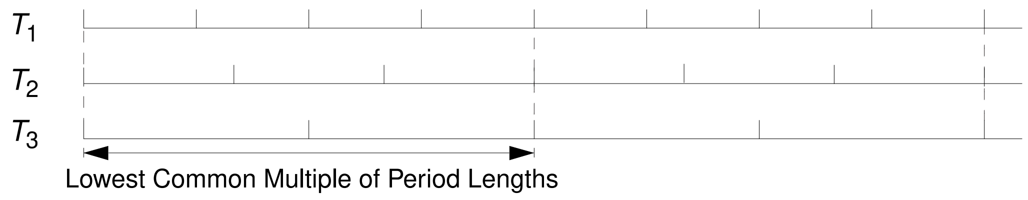

Fig. 4. Schedule Repetition Interval.

To create the schedule, the processes must be arranged for that interval so that each process finishes before its deadline and in summary as few as possible overlaps occur. This way, for each process in each period a time is determined when the process should become ready. These arrival times can be stored in a table. During run time, after the execution of a process has finished for a certain period, the next ready time can be retrieved from the table. If the table is exhausted, it can be applied again, whereas the stored arrival times must be adapted to the current time.

The schedule is determined by simulating the execution of the processes. It must be decided whether the execution of process T_i can continue at time t if the period of a process with a higher priority begins at this moment. To this end, it must be checked whether all processes with a higher priority than T_i meet their deadlines even if T_i is executed non-preemptively. This is done by arranging all processes in a non-preemptive manner, storing the times at which the execution of a process begins in the table, until either the schedule has been completed or a deadline violation occurs. In the latter case, the last part of the schedule must be dropped. The deadline violation must have its reason in the non-preemptive execution of a lower priority process because the process set is schedulable under the preemptive rate-monotonic scheme. The process T_j with the lowest priority among the processes responsible for the deadline violation is determined and it is temporarily marked and handled as preemptable. The simulation continues from the start of the process T_j in the considered period (Figure 5). After all higher priority processes have finished their execution, T_j is marked as non-preemptable again and continues to work. Note that during other periods this process may execute without any preemptions. Moreover, it might be necessary to perform multiple of the described backtracking steps until all higher priority processes can be executed before their deadlines.

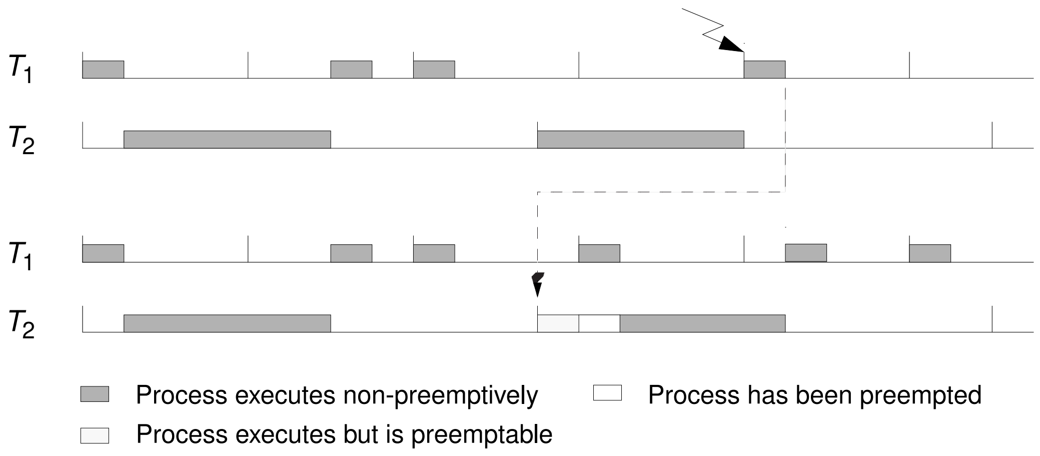
T_1
T_2
T_1
T_2
Process executes non-preemptively
Process has been preempted
Process executes but is preemptable

Fig. 5. Backtracking During Simulation due to Deadline Violation.

The run-time overhead of this method is low since it consists only of the retrieval of the next ready time from the table. The complex part of the scheme is the calculation of the scheduling table which must be performed when the process set changes, i.e., each time a process is created or deleted or when the parameters of a process change, e.g., due to stream scaling. The overhead for the precalculation depends on the process set but can be time and space consuming if the lowest common multiple is large.

Modification of Arrival Times – Dynamic Determination

As an alternative to the static precalculation, the arrival time for each process in each period can be determined dynamically during run time. Again, overlappings can be avoided by considering the execution times of other processes. This dynamic approach promises flexibility. Changes in the process set such as created or deleted processes and changed process parameters are taken into account immediately. Additionally, no memory space to store arrival times is necessary because that information is generated only when it is needed.

However, a major drawback is the run-time overhead which must be paid for each process in each period, hence it must be as small as possible. If, for instance, the execution time of a process per period is in the order of a few milliseconds, an overhead in a similar order is absolutely unacceptable. Therefore, the algorithm must be simple, even if its scheduling decisions are not optimal in the sense that potentially avoidable overlapping situations occur.

The schedule is created in the following way. At the end of its processing for a period each process calls the scheduler to calculate the next ready time for this process. The scheduler needs information about the behavior of the other processes during the next period of the considered process. This information is generated and managed inside the scheduler. It attempts to reserve a time slot in the next period of the process which has a length corresponding to the execution time and puts the process to sleep until that time slot begins. Reservations already made for other processes must be taken into account by the scheduler. This reservation procedure is illustrated in Figure 6. To simplify the presentation, it is assumed that all processes become ready for the first time.

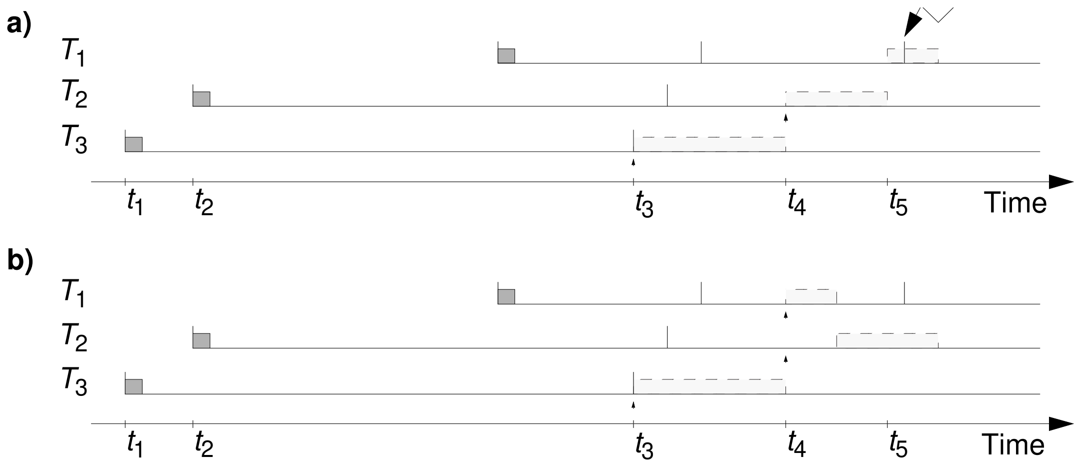

Fig. 6. Time Slot Reservation.

In the upper part, at time t_1 the first process T_3 becomes ready. To reduce conflicts with possibly running processes, no stream processing is performed during the first period, but only the scheduler is called to reserve a time slot for the next period. Since no reservations have been made so far, T_3 is scheduled for the time slot beginning with its next period at t_3. At time t_2 the second process T_2 becomes ready and calls the scheduler. Since a reservation has been set for T_3, the ready time for T_2 is set to t_4 so that the execution of T_3 is not preempted by T_2. If the same procedure is applied to process T_1, it will not be schedulable without deadline violation.

This shows the difficulties which occur at the time slot reservation for a high-priority process. Due to longer periods (compared with high-priority processes) the processes with low priorities such as T_3 perform their reservation early and find enough unreserved time slots. For processes with a high priority this is exactly the opposite. For example, in the above scenario the next time slot available to process T_1 begins at t_5 which is too late to finish before the deadline. Even if T_1 could be processed just in time, this would mean that priorities are not taken into consideration because T_2 would be executed before T_1.

For this reason, a reservation is not fixed in time (lower part of Figure 6). Since T_1 has a higher priority than T_2, the scheduler reserves a time slot for T_1 at t_4 and moves the reservation for T_2 accordingly. In order to reduce the overhead, the arrival time of T_2 is not adapted. This is not necessary because at time t_4 process T_1 is dispatched due to its higher priority. But it is required to move the reservation for T_2 to indicate that T_2 will be processed within that interval.

If the scheduler detects that a process cannot finish its processing in time (either a process which tries to reserve a time slot or a process for which the reservation has just been moved) then the process which is the reason for that violation must be executed preemptably. To this end, its reservation is removed which results in new reservations within this interval for other, higher priority processes. Since the process will wake up at the originally scheduled time, overlaps may occur.

4 Scheduling Methods for Processes with Varying Arrival Times

Section 3 presented mechanisms to reduce the amount of overlapped processing which can be used successfully for certain types of applications such as video servers. In such a scenario a process must never wait for data to transmit and, therefore, never yields the processor voluntarily. However, this is not true for receiver processes.

For example, a receiver process in a video conference must wait until a packet has been received from the network before further processing can be done. The wait time might vary between periods due to jitter introduced by the network (if the used network provides no tight jitter guarantee). If the process would start at the begin of the period and then block to wait for a data packet, overlapping would be introduced if another process would execute meanwhile. However, prohibiting other processes from execution during that wait time leads to unused processor cycles and reduces the CPU utilization contradicting the goals of the algorithms. This problem can be solved if wait times are eliminated.

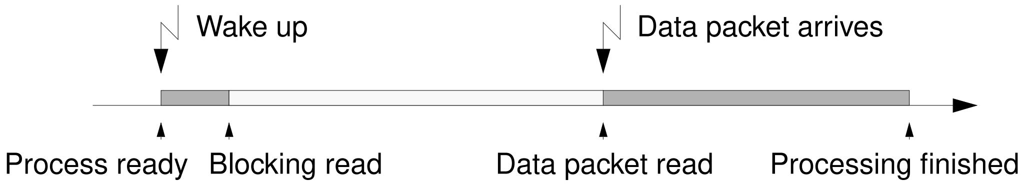

Fig. 7. Wait Times During Processing Period.

Figure 7 illustrates that two events are necessary before a receiver process can run to completion in a period: the process must wake up and the data packet to be processed must have been received. If the process is not awoken before both events occurred as shown in Figure 7, the process can read the packet immediately. This way, it does not have to wait, and the execution can be performed without interruptions.

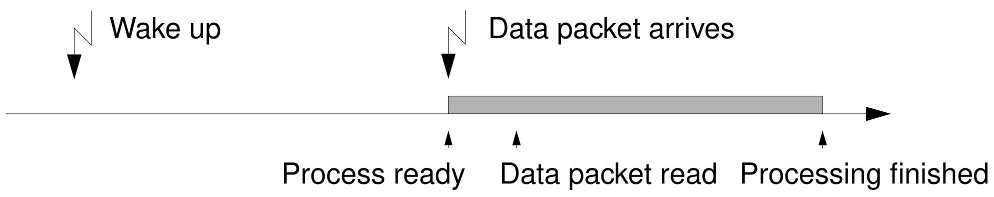

Fig. 8. Elimination of Wait Times.

If that approach to eliminate the wait times is used, no information is available when a receiver process will actually wake up. Receiving processes are now of a more sporadic nature than sending processes which are mostly strictly periodic and have known arrival times. Therefore, the scheduling algorithms must be examined whether they are able to schedule processes with varying arrival times.

The exact arrival time of a receiver process is not known because it depends on the receipt of the data packet. Thus, the algorithms which modify the arrival times (static and dynamic version) can not be used. They are based on the idea that a process indeed starts its execution at the scheduled time. Hence, only the non-preemptive methods can be applied successfully.

5 Evaluation

The purpose of the scheduling mechanisms examined in the previous sections is the reduction of contention situations among real-time processes. Therefore, the following evaluation should clarify to which extend the algorithms are able to fulfill this goal. Since specific mechanisms are needed to avoid overlapped execution, the additional costs, i.e., the amount of resources needed for these mechanisms, must be evaluated as well. The additional costs for the non-preemptive algorithms consists of the functions to inhibit preemptions. For the algorithms which modify the arrival times, the scheduling itself leads to overhead: for the static version the table must be calculated resulting in processing time and memory space requirements, in the case of the dynamic version the next arrival time must be determined. Only the static precalculation of arrival times needs significant memory space to store the scheduling table.

The processing time measurements have been performed on a IBM RISC System/6000 (Model 360) workstation with AIX 3.2.4. The measurement events have been generated using the trace mechanism provided by the operating system.

Tab. 1. Process Sets.

	Number of Processes	Process Parameters			CPU Utilization
		Rate	Period	Processing Time/Period	
Process Set 1	6	5 s^{-1}	200 ms	7 ms	67.5 %
	6	10 s^{-1}	100 ms	4 ms	
	5	15 s^{-1}	66.5 ms*	3 ms	
Process Set 2	20	5 s^{-1}	200 ms	7 ms	70 %
Process Set 3	1	5 s^{-1}	200 ms	60 ms	77.2 %
	1	11.1 s^{-1}	90 ms	20 ms	
	1	25 s^{-1}	40 ms	10 ms	

* rounded to keep the lowest common multiple reasonably small

Scenarios

The developed algorithms are only partially usable for scenarios with varying arrival times, but can be used for applications with known arrival times, for instance, a video server. Therefore, the measurement setup resembles such an application; the test processes execute with timely characteristics found in a video server.

To perform a worst-case evaluation of the implementation of the algorithms, the process sets have been chosen in such a way that the overall CPU utilization is large, i.e., close to the maximum permitted by the preemptive rate-monotonic algorithm. The process sets are given in Table 1.

In *process set 1* for each of the three different rates approximately the same number of processes exists resembling a video server supporting three different retrieval rates, e.g., for heterogeneous clients.

The *process set 2* contains only processes with the same rate. As explained before, such a scenario can be considered as typical for video server which might support only one standard rate. This scenario is directed to the examination of the non-preemptive scheduling of processes with equal rates which should be able to schedule the process set without overlaps.

The purpose of the last scenario, *process set 3*, is to examine the scheduling algorithms if a process set is not schedulable without overlaps. Here, the non-preemptive mechanism cannot be used. This can easily be seen looking at the process parameters which have not been taken from a video server: the long processing time of the first process (60 ms) inhibits the third process from meeting its deadline at the end of its period (40 ms). In this scenario the performance of the remaining algorithms is of major interest.

Results – Reduction of Overlapped Execution

The following figures show the ability of the algorithms to reduce the number of overlaps occurring during the execution of the process sets. The bars symbolize the normalized measurement interval. Each section of a bar specifies (in average) the portion of the measurement interval where *n* processes were executing concurrently. *n* is given for each section. For comparison purposes, the degree of concurrency when using the preemptive rate-monotonic scheduling algorithm is shown, too. Note that the shaded part corresponds to the CPU utilization of the process set in question.

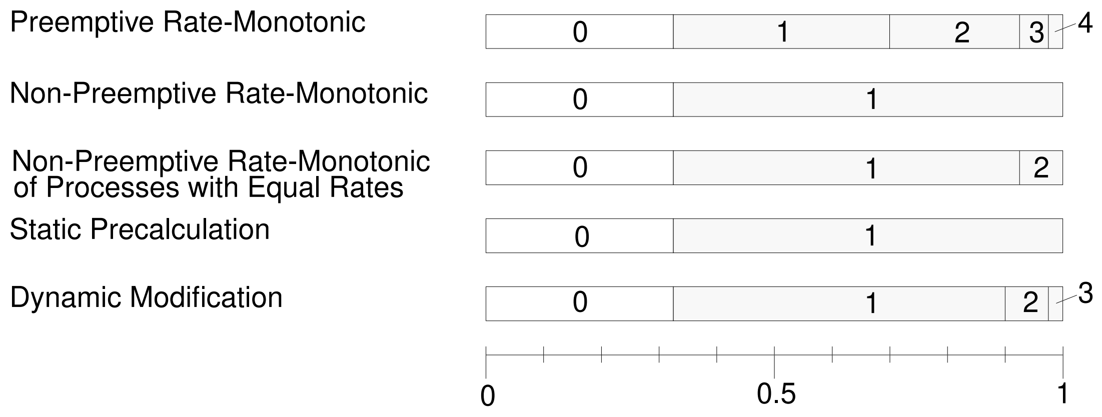

Fig. 9. Reduction of Overlapped Execution for Process Set 1.

Figure 9 illustrates that a large amount of concurrent execution can be observed if the preemptive rate-monotonic scheduling algorithm is used. As expected, the non-preemptive algorithm avoids overlapped execution completely. The algorithm which schedules processes with equal rates without preemptions can reduce the number of overlaps significantly.

The static precalculation is an optimal mechanism. Since the process set is schedulable non-preemptively, this method should also find a schedule without any preemptions, and it is indeed able to do so.

It has been expected that the results for the dynamic modification of the arrival times are comparable to the results gained by the static precalculation (both take a similar approach). However, the number of overlaps using the dynamic scheme is higher; up to three processes are executing concurrently. The reason is that the implementation, in order to reduce the overhead, does not adapt the wake-up times accordingly if time slot reservations are moved. Thus, several processes may wakeup simultaneously. In principle, the execution of the processes should nevertheless be serialized due to their distinct priorities. However, since the number of real-time priorities in AIX 3 is limited, the (logical) process priorities must be mapped to the smaller number of available real-time priorities. Now several processes can have the same priority. Processes with the same real-time priority are scheduled using the round-robin scheme with a maximum but not guaranteed time slice of 10 ms [Britton93], so that a process might be dispatched even if another has not yet finished its execution.

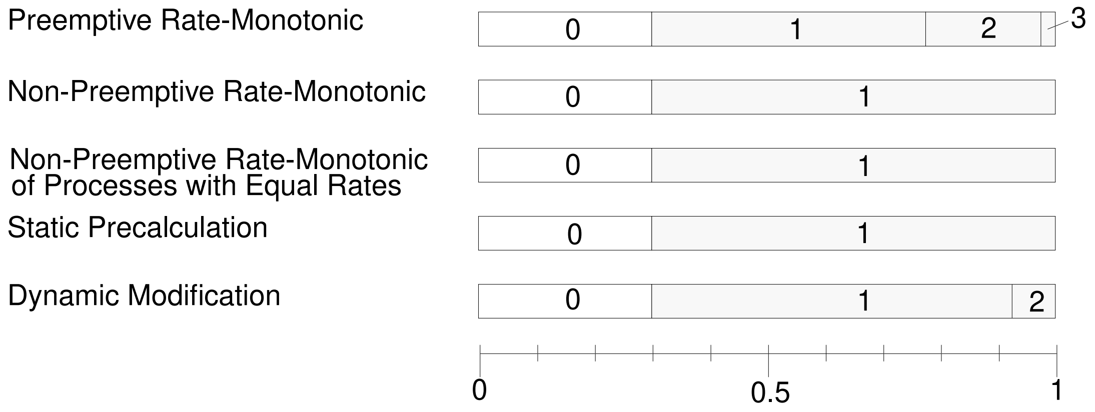

Fig. 10. Reduction of Overlapped Execution for Process Set 2.

The results for process set 2 are better for all scheduling algorithms (Figure 10). The preemptive rate-monotonic scheduling scheme leads also for this process set to a considerable number of overlaps. As expected, both non-preemptive methods are able to schedule the process set without any preemptions and the static precalculation performs equal to them. The dynamic modification scheme achieves a better result than for process set 1, however, its result is still worse than any of the other three methods.

Process set 3 is not schedulable with the non-preemptive rate-monotonic algorithm, hence, Figure 11 presents the results for the other schemes only.

Using the preemptive rate-monotonic scheduler, two processes are executing concurrently for a large portion of the measurement interval. The result for the non-preemptive scheduling of processes with equal rates is similar. The reason is that no two processes have the same rate, so the method is basically identical to the preemptive scheme. That the behavior is slightly better is simply by accident.

The static precalculation approach is able to reduce the number of overlaps significantly. This case demonstrates clearly the difference to the non-preemptive scheduling. While the non-preemptive scheduler rejects the process set, the static precalculation can serialize the execution of the processes in an optimal manner. Whenever possible it allows the non-preemptive execution otherwise preemptions are permitted.

Once again, the result gained by the dynamic modification method is not acceptable.

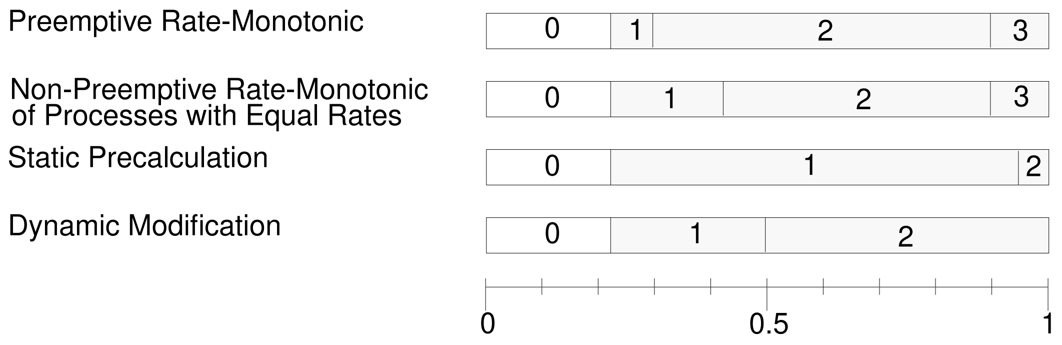

Fig. 11. Reduction of Overlapped Execution for Process Set 3.

Results – Additional Overhead

The scheduling mechanisms are directed to reduce the number of overlapped executions and, hence, to improve the system performance. This implies that the saved resources should not be consumed by the scheduling mechanisms. Here, the overhead introduced by the algorithms is evaluated.

Tab. 2. Overhead due to Coordination Mechanism Inhibiting Preemptions.

	Non-Preemptive Scheduling			Non-Preemptive Scheduling of Processes with Equal Rates		
	Minimum	Maximum	Average	Minimum	Maximum	Average
Process Set 1	2 µs	26 µs	8 µs	2 µs	15 µs	8 µs
Process Set 2	2 µs	22 µs	10 µs	2 µs	15 µs	9 µs
Process Set 3	–*	–*	–*	2 µs	3 µs	2 µs

* not schedulable

Table 2 shows the costs (processing time) generated by the non-preemptive scheduling schemes. For them, coordination mechanisms are required which ensures that the processes execute without preemptions. The overhead for the non-preemptive scheduling mechanism is larger than that for the non-preemptive scheme among processes with equal rates. The reason is that the number of processes to be coordinated is larger for the first scheme since the second must consider smaller subsets only. In both cases the overhead must be paid for each process in each period.

The static precalculation algorithm leads to processing time and storage space overhead, i.e., to calculate and to store the scheduling table. Both depend on the process set what can also be seen in Table 3 showing the costs for the three considered process sets. For a process set $\{T_1, \ldots, T_n\}$ the required space M (in bytes) to store the scheduling table can easily be computed since each process entry consumes 12 bytes and each stored arrival time needs 8 bytes in the current implementation:

$$M = \sum_{i=1}^{n} \left(12 + \frac{\text{lowest common multiple}}{T_i} \times 8 \right)$$

For a short period, while calculating a new scheduling table, more space is needed because the old schedule is valid and must be kept until the new table has been computed.

The required time for the schedule calculation presented in Table 3 has been measured under a significant real-time processing load, i.e., while the system was executing the respective process set. The run-time overhead when retrieving the next ready time from the scheduling table is neglectable.

Table 3 also showes the overhead for the algorithm which modifies the arrival times dynamically. It must be noted that these computation times are necessary for each process in each period.

Tab. 3. Scheduling Overhead.

	Static Precalculation					Dynamic Modification		
	Lowest Common Multiple	Storage Space	Processing Time			Processing Time		
			Min.	Max.	Ave.	Min.	Max.	Ave.
Process Set 1	26.6 s	35356 bytes	148 µs	29840 µs	6734 µs	2 µs	22 µs	5 µs
Process Set 2	0.2 s	400 bytes	147 µs	923 µs	498 µs	2 µs	23 µs	7 µs
Process Set 3	1.8 s	628 bytes	142 µs	724 µs	365 µs	2 µs	22 µs	4 µs

6 Related Work

Real-time scheduling mechanisms which consider the usage of resources have been studied by several research groups.

[Zhao87] presents a method with the two goals that each process meets its deadlines and that each process has exclusive access to resources which must be specified accordingly. No a priori information about process arrival times are required by this method. The scheduling is done each time when a process arrives. The algorithm creates a tree of possible schedules and performs a heuristic search on that tree to determine whether a new arriving process can be scheduled without violating the guarantees given to already existing processes, i.e., that processing can complete before the deadlines and that all exclusive resources required are available exclusively.

The method described in [Xu90] guarantees not only processing before deadlines and exclusive access to resources but also considers precedence relations among parts of different processes. Since the algorithm is aimed to be used for the pre-runtime calculation of the schedule, the process set must be completely known in advance. Each process consists of a set of segments with known arrival times, execution durations and deadlines. Precedence and exclusion rules can be specified between pairs of segments. The precedence rules allow to specify that the processing of one segment must be finished before the execution of the other segment may start. Exclusion rules can be used to get exclusive access to a resource during a segment. The unit considered for scheduling is not a process but a segment. Again, the schedule is generated with the help of a search tree.

Another scheduling algorithm which guarantees execution before deadlines and exclusive access to resources was developed as part of YARTOS [Jeffay90]. Here, resources are shared software objects which can be accessed by only one process at a time. Processes are characterized by an execution time and a deadline, the former is partitioned into phases. By definition, a process accesses at most one resource during each phase. The unit considered for scheduling is a phase. A process can only be preempted by a higher priority process if the higher priority process does not require a resource which is accessed by the currently running process during this processing phase; priority inheritance is used to avoid priority inversions.

In addition to guaranteeing that the processing finishes before the deadline, it is a common characteristic of these three methods that they can ensure that a process can exclusively access specific resources without explicit coordination mechanisms such as semaphores. To be able to provide these guarantees, the last two methods need the information for each process when which resources are required and for how long. The first method assumes that exclusive resources are needed during the whole execution time. Then a given process set, i.e., the set of processes and their resource usage specifications, is only schedulable and hence accepted if all processes can finish their execution before their deadlines and if all specified resources can be accessed exclusively.

Even though these mechanisms ensure that a process can access a resource exclusively, it might nevertheless be preempted by other processes which do not require access to this resource at that time. Hence, conflicts, with respect to the definition used in this paper, can still occur. These methods can avoid overlapping execution only for the special case when the resource *processor* is specified as a resource for which exclusive access is required.

Scheduling mechanisms, such as the methods described in this work, which have been designed to reduce overlapped execution can provide for better efficiency and universality. For instance, it cannot be assumed that resource access characteristics in terms of when and how long resources will be used are usually known. This would require the provision of that information by the operating system which is not the case, e.g., in the used AIX operating system. Further, the methods examined within this work (except the pure non-preemptive mechanism) can accept process sets even if they cannot be executed without overlaps. This increases the processing time spent due to operating system overhead and requires explicit coordination of the processes, however, the advantage is that a larger number of process sets can be scheduled.

7 Summary

A large amount of resources is needed for the processing of continuous-media data. This can be reduced if the execution of processes working on such data does not overlap. To achieve this, several scheduling algorithms have been developed and evaluated by measurements.

Non-preemptive scheduling avoids overlapped processing a priori. The measurements show that only a small additional overhead is required for it. However, it restricts the schedulable process sets to a relatively large extend. Thus, its usability depends strongly on the application scenario. If it is used for process sets where it can be guaranteed that all processes will hold their deadlines even for large CPU utilizations, then this scheme leads to good results.

The same degree of overlap avoidance has been reached by the static precalculation algorithm. Further, this method can always schedule process sets which are schedulable under the preemptive rate-monotonic scheduling scheme, i.e., it leads to equal CPU utilizations. At least for the measurements performed, the overhead to calculate and to store the scheduling table was acceptable. Hence, this scheme offers a good overall performance.

The results of the other two mechanisms are less promising. The number of concurrently active processes is less than using the original preemptive scheduling, however, not satisfactory, especially for the dynamic scheme. The overhead introduced by each method is relatively low. While both schemes can schedule the same process sets as the preemptive scheduler, only the non-preemptive scheduling of processes with equal rates seems to be usable. If the typical process set of an application contains several processes with equal rates, this scheme may be applied successfully.

References

[Britton93] B. Britton: AIX 3.2 Multiuser System Tuning and the New Performance Tuning PTFs, *AIXPRESS*, January 1993, pp. 9-13.

[Drake94] S. Drake, IBM Almaden, private correspondence, May 1994.

[Haskin93] R. L. Haskin: The Shark Continuous-Media File Server, Proceedings of COMPCON '93.

[Jeffay90] K. Jeffay: Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems, TR 90-039, University of North Carolina at Chapel Hill, Department of Computer Science, November 1990.

[Liu73] C. L. Liu, J. W. Layland: Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, *Journal of the ACM*, Vol. 20, No. 1, January 1973, pp. 46-61.

[Nagarajan92] R. Nagarajan, C. Vogt: Guaranteed-Performance Transport of Multimedia Traffic over the Token Ring, TR 43.9201, IBM European Networking Center Heidelberg, Germany, 1992.

[Williamson95] J. Williamson, L. C. Wolf: Reducing Buffer Space Requirements for Multimedia Data Streams: Analysing the effects of staggering streams and preemption in buffer pools, TR 43.9504, IBM European Networking Center, Heidelberg, Germany, 1995.

[Wolf96] L. C. Wolf, W. Burke, C. Vogt: Evaluation of a CPU Scheduling Mechanism for Multimedia Systems, to appear in *Software – Practice and Experience*, 1996.

[Xu90] J. Xu, D. L. Parnas: Scheduling Processes with Release Times, Deadlines, Precedence, and Exclusion Relations, *IEEE Transactions on Software Engineering*, Vol. 16, No. 3, March 1990, pp. 360-369.

[Zhao87] W. Zhao, K. Ramamritham, J. A. Stankovic: Scheduling Tasks Under Time and Resource Constraints, *IEEE Transactions on Computers*, Vol. 36, No. 8, August 1987, pp. 949-960.