Issues in Multimedia Task Scheduling

André Barroso, Andreas Manoli, Michail Petropoulos

November 21, 1998

Abstract **- Computational support for multimedia applications has been subject of considerable research. Much attention has been paid to communication media protocols in distributed multimedia systems. However, properly defining a share of CPU time for the tasks that will manipulate continuous stream data is fundamental to multimedia applications. This paper focuses in basic concepts of multimedia task scheduling and surveys some approaches to solve this problem.**

I Introduction

Multimedia systems are included in the broader area of real-time systems and they thus inherit the basic characteristics of such systems. They also particularize certain aspects that make them a special subgroup of real-time systems. Real-time systems are defined as those having not only logical constraints, but also timing constraints in the services they provide. In other words, the correctness of service is evaluated in terms of both specified response values and time of response. Despite the temporal dimension added to the specification of these systems, their design will not focus only on speed of processing. For instance, a system responsible for receiving a real-time video stream, processing this stream and showing it in a display is supposed to present the frames in a specified rate, neither faster nor slower. Generally speaking, the major feature that must be present in those systems is predictability. Faster processors can do no worthy job if a deadlock scenario occurs. For this reason, real-time systems have a complex set of issues that have to be solved in order to provide a correct service [1]. Of course, the enforcement of predictability lies on the specific application of the end system. In fact, there exists a wide range of applications that demand services with timing constraints: air traffic control, video conferencing, virtual reality, etc. Real-time systems are commonly classified into two subcategories depending on the intended application: Hard real-time systems and Soft real-time systems. Hard real-time systems are the systems applied in services that must strictly observe temporal constraints. Failures in this observance may lead to catastrophic results, such as loss of human lives. For instance, a system that controls train traffic in a railroad may be responsible for a crash if the proper semaphores are not set in time. On the contrary, soft real-time systems have a much more flexible time constraint. Despite the temporal specifications of the services they provide, violations of deadlines are considered acceptable to a certain extent. Quality of Service degradation is the penalty for out-of-time services in such systems. Multimedia systems are normally soft real-time systems. Nevertheless, as new applications involving multimedia come up, some of then will certainly demand hard real-time guarantees. A remote surgery is an example of such application where delays in the image transferred and processed may be fatal.

It follows from the previous discussion that a successful approach to building systems with timing constraints involves cooperation of many different parts of an operating system. The basic problem to be solved is how to share resources between processes in a way that timing constraints of services are met. In a distributed system this concern may involve for instance the communication media sharing. In the same way processors are important resources of a computer system and the task scheduling problem is related to finding ways of distributing processors cycles between processes in order to achieve their logical and temporal constraints. To solve all these issues, some multimedia systems were designed from scratch because their designers considered that a complete redesign would address the problems in a more efficient way. [2] and [3] are examples of such systems. Other approaches try to extend existing operating systems to adapt them to the multimedia applications demand. Works in this area are observed in [4] and [5].

The remaining of this paper is organized as follows. Section II introduces fundamental concepts involving the task scheduling problem and presents the two most widespread algorithms that try to solve it. In section III the characteristics that make multimedia processes distinct from other classes of real-time tasks are introduced. Section IV presents some solutions proposed to schedule multimedia tasks in recent research projects in the area. Finally, section IV summarizes the work and presents the conclusions.

II The task scheduling problem

Processes are the basic logical unit of computation in a system. They are created to perform some services and assume various states until their termination. When a process is considered ready to perform its job, it is inserted in the ready queue maintained by the operating system, until it is chosen to run by the scheduler. A process has access to the processor until some event occurs that interrupts it. It is then placed again in the ready queue for future processing, terminated, or blocked until some other event takes place. The scheduler is the program responsible for defining the order by which the processes will be serviced by the CPU. To make the choices that define the order of the serviced tasks, the scheduler follows a scheduling policy devised to optimize certain aspects of the service provided by the systems. Scheduling algorithms have been proposed, for instance to maximize the throughput of a system, minimize service latency and guarantee that tasks are executed in the times specified for them. Solving the scheduling problem in an optimal fashion is not a trivial issue. For non-trivial scenarios it is generally a NP-hard problem [6]. The problem can get more complex if more constraints are added, as in the case of real-time systems. In such systems, tasks have a deadline indicating the maximum time allowed for the service to be provided. It may also happen that tasks cannot be executed before a certain instant known as release time. Since the difficulty of defining a scheduling policy is directly influenced by the characteristics of the task set being scheduled, we next introduce different behaviors expected from the tasks in a real-time system.

Processes can have a periodic, aperiodic, or sporadic pattern of activation requests. The first group comprises of those tasks that request execution every t units of time. Their behavior is therefore quite predictable. On the contrary, aperiodic tasks can demand execution at any time, allowing bursts of requests that may overload the processor. The sporadic group of tasks may request activation at any instant (like aperiodic tasks) but requests that an interval of at least t units of time exists between two successive requests. In this way, sporadic processes allow a worst-case scenario analysis. Tasks can also be independent if there are no interactions among them. Dependent tasks may interact in a number of ways. For instance, some tasks may share variables or have precedence constraints that must be considered during the schedule definition. Thus, task dependencies impose additional complexity to the scheduler. The problem of deciding whether it is possible to schedule a set of periodic processes using semaphores to enforce mutual exclusion is a NP-complete problem [7]. In real-time systems the case where interactions between tasks produce uncertainty in the amount of time a process must wait to access the processor is especially undesirable. Deadlocks or priority inversions must be carefully avoided in order to provide services in a timely fashion. Priority inversion occurs when a task of higher priority is preempted by a lower priority task. This situation can be described in a simple example. Assume three tasks T_1, T_2, T_3 where task T_1 has the highest priority and task T_3 has the lowest. Tasks T_1 and T_3 share a common resource whose access is controlled by a semaphore M. At a given instant of time, T_3 is in the critical section controlled by semaphore M . It is then preempted due to the arrival of task T_1. T_3 is then placed in the ready queue and T_1 is allowed to execute. After executing for a while, T_1 tries to acquire the semaphore, but since it belongs to T_3 the task is blocked. If T_2 requests acquisition of the processor, it will then prevent T_3 from gaining access to the CPU and release semaphore M. Therefore, a higher priority task T_1 is being prevented to execute by a lower priority task T_2. A situation may arise in which T_1 can stay blocked for an undefined amount of time. A solution to this problem was proposed by [8] and is known as Priority Ceiling Protocol. PCP is an extension of the Priority Inheritance Protocol [8]. Every semaphore has a ceiling priority equal to the highest priority among the tasks that acquire it. When a task T tries to lock a semaphore, it must be checked before if there is any other semaphore locked with ceiling priority higher than the priority of T. If there is no such a semaphore, T can enter the critical section. If such a semaphore exists, then T is blocked. It may happen that when T is executing this critical section, another task T' with higher priority than T is blocked because of the semaphore acquisition by T. In this situation, T must inherit the priority of T' until it releases the semaphore. This mechanism is proved to avoid deadlocks and limit the amount of time that a higher priority task stays blocked by a lower priority one.

Chen *et al* [9] categorize real-time scheduling algorithms by classifying them into soft and hard real-time scheduling algorithms. For each of these categories, a second level division distinguishes between dynamic and static scheduling algorithms. A dynamic scheduler makes its scheduling decisions at run time on the basis of current service requests. This approach has the advantage of being very adaptive, handling new scenarios as they occur. Static schedulers make all their decisions off-line and generate a dispatching table with the policy to be executed by the dispatcher at run time. Static scheduling is more appropriate for hard real-time systems where it is fundamental to guarantee time constraints *a priori*. Since decisions are made off-line, it is admissible by the scheduler to apply timing consuming heuristics like Simulated Annealing [10] to find a good solution for the task scheduling problem. Dynamic or static schedulers may allow or not allow tasks to be preempted due to the arrival of a higher priority task in the ready queue. If it is possible to remove executing tasks, the scheduling is said to be preemptive. In a non-preemptive scheduling, a task is allowed to execute until it finishes. The following subsections describe two dynamic algorithms that are commonly used to define a feasible scheduling to a set of real-time tasks. A feasible schedule is one that meets all timing and logical constraints of a task set.

Rate Monotonic Algorithm

Proposed by Liu and Layland [11] in a classic paper of the real-time literature, the RM algorithm provides a way of guaranteeing deadlines of a set of tasks under the following assumptions:

1. All tasks must be periodic and independent;
2. The deadline of each task is equal to its period. It means that no task is supposed to request another activation before completion of its previous request;
3. The maximum computation time of each task is known a priori and is constant;
4. The time required for context switching is negligible;

Among tasks in the ready queue with the above characteristics, the algorithm chooses always the one with lowest period to occupy the processor. Reasoning in another way, the task with the greatest request activation rate has priority over all other tasks. Rate monotonic is a dynamic preemptive scheduling algorithm that assigns priorities to the tasks in a static fashion. In fact, the period of a task is not supposed to change and as a consequence its priority remains immutable. Rate monotonic is always able to define a feasible schedule to a set of n independent periodic tasks that obeys the following inequality:

$$\mu = \sum_{i=1}^{n} \frac{c_i}{p_i} \leq n(2^{1/n} - 1) \tag{1}$$

where c_i and p_i are respectively the maximum execution time and period of the i-th task of the set to be scheduled. The variable μ represents the processor utilization and therefore cannot be greater than 1. As the number of tasks in the set to be scheduled grows, the second part of the above inequality tends to $\approxeq 0.69$. This means that any set of periodic tasks demanding processor utilization less than 0.69 is schedulable using RM algorithm. The above inequality provides an easy method of deciding if a given task set is schedulable or not. Any method that decides such a problem is known as a schedulability test. RM algorithm is proved [11] to be an optimal algorithm in the sense that if any other algorithm can find a feasible schedule for a task set based on static priorities, RM can also find it. The limit of 0.69 stated above, is not a necessary condition but a sufficient one. In fact, RM can define a feasible schedule for a set of tasks demanding even hundred percent of the processor, if these tasks present special characteristics.

Earliest Deadline First

Based on the same assumptions stated for the Rate Monotonic algorithm, EDF dynamically assigns priorities to the tasks. Among the tasks ready to execute, the one with earliest deadline has precedence to occupy the processor. EDF is able to always define a feasible schedule to a set of n independent periodic tasks that obey the following inequality:

$$\mu = \sum_{i=1}^{n} \leq 1 \tag{2}$$

where the symbols have the same meaning as those described in the RM algorithm. Thus, EDF always finds a feasible schedule for any set of independent periodic tasks that demand less than hundred percent of processor utilization. In this sense, EDF is proved to be an optimum algorithm as well [11].
The above algorithms rely on the strong assumption that tasks are independent from each other. For most of the cases this is not true. In such circumstances, additional care must be taken to guarantee tasks deadlines. As an example, the Priority Ceiling Protocol can be used to avoid priority inversion and guarantee that a task with certain time constraints accesses the processor in time. Sha, Rajkumar and Lehoczky [8] proposed a sufficient schedulability test for a set of n periodic tasks using PCP that is scheduled by RM algorithm:

$$\mu = \sum_{i=1}^{j} \frac{c_i}{p_i} + \frac{B_i}{p_i} \leq i(2^{1/i} - 1), \forall j, 1 \leq j \leq n \tag{3}$$

where B_i is the worst case blocking time that a task T_i can be blocked for lower priority tasks. It is assumed that $p_i > p_{i+1}$ for $1 \leq i < n$.

III Multimedia Tasks Profile

In this section the characteristics that make multimedia processes distinct from other classes of real-time tasks are presented and basic issues involving QoS are discussed.

Different kinds of data can be regarded as multimedia. Among them continuous media data (audio and video) represent a special challenge for the operating system designer because they demand real-time processing. Continuous media streams have the following important properties:

1. The quality of information they carry depends on the time it is presented to the user. Processes responsible for gathering this type of data or displaying them must provide their services in a timely fashion and they therefore have time constrains;

2. Violation of task timing constraints results most of the times only in degradation of the service quality without further serious consequences. Most of the degradation remains unnoticed by the human perception. This time fault tolerance makes the scheduling problem a soft real-time one.

3. Due to the large amount of communication bandwidth and memory they demand, these data are usually compressed. Variations in the complexity of the information they carry, and their uncertainty, can impose highly variable processing requirements for the tasks responsible for compressing and uncompressing the data. It is therefore difficult to predict and define a worst case execution time for those tasks.

4. Despite the soft real-time requirements of these applications, the user demands a guarantee of Quality of Service (QoS) even during overload situations.

5. Continuous media data is produced and retrieved by periodic processes.

A system can use RM or EDF algorithm to schedule tasks responsible for processing continuous media data. That is exactly what some systems do. For instance, the Artropos scheduler used in the Nemesis operating system [12] applies EDF to schedule entities called scheduling domains (*sdoms*) that have an associated deadline and share the CPU bandwidth. An *sdom* may correspond to a single process or a group of them. In the MMOSS project [13] the scheduling of soft real time tasks is made using a mix of rate-monotonic, weighted round-robin and priority-based scheduling. Other approaches try to further explore the characteristics of continuous media data described above. The claim is that EDF and RM schedulers do not provide any guarantees when CPU bandwidth is overbooked, and defining a worst case execution time for multimedia tasks, are difficult issues. These can pose a serious problem in maintaining the quality of service required. Therefore, several other methods have been devised, some of them being described in the following section. Some of the approaches use dedicated hardware to deal with multimedia tasks, and some of them mix different classes of tasks in the same CPU. These last solutions demand the scheduler to recognize and treat properly each class of tasks by providing the services required by them. Bad behavior caused by priority inversion and deadlocks caused by interactions between multimedia tasks can be solved by the use of PCP.

A general model of providing multimedia services is interposing a QoS Manager between user applications and the scheduler in order to negotiate and enforce quality of service (Figure 1).

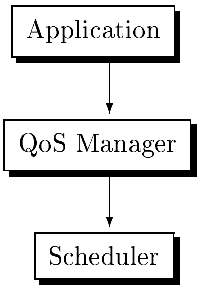

Figure 1 - Quality of Service Management

Managing QoS in an operating system can be done in different ways. For instance, if an activation request of a task will lead to violation of any pervious guarantees offered by the QoS Manager, the QoS Manager can simply refuse to satisfy the new request. Those guarantees may be statistical or not. In the case of overload, the manager may try to adapt to the situation by degrading the service of the lower priority applications.

IV Multimedia Scheduler Approaches

In this section we present some solutions proposed to schedule multimedia tasks in recent research projects in the area. The solutions are grouped into two major categories: integrated scheduling and autonomous scheduling. In the former approach, multimedia data are handled by the host CPU together with all other kinds of data. In the latter solution, dedicated processors manipulate real-time tasks. Integrated scheduling has the advantage of flexibility, scalability, and not specialized hardware but introduces more complexity to the scheduler because of its general-purpose characteristic. Autonomous scheduling may require minimal modification to the host operating systems, but more complex hardware is needed.

IV.1 Autonomous Scheduling

The need of dedicated hardware is a typical requirement of hard real-time systems. In some applications, it may be desirable to integrate multimedia tasks with time critical tasks. For instance, attack helicopters are being designed to take advantage of audio and video information [1]. In this situation, the strict deadlines of the tasks responsible for helicopter control must be met, while the system must also provide an acceptable quality of service to the video and audio applications. To deal with multimedia data, Kaneko *et al* [14] proposed the use of a periodic task that is dynamically created and scheduled along with hard real-time tasks. This periodic task, named multimedia server, must schedule multimedia tasks in the best possible way so they can meet their deadlines. The multimedia server encapsulates one or more multimedia tasks creating a unique schedulable entity and delivering it to the hard real-time scheduler. The server is given a fraction of CPU time and is responsible for controlling the execution of multimedia tasks. An algorithm like EDF can now be used to schedule hard real-time tasks and the multimedia server task. Multimedia tasks can be assigned to the server in two different ways: proportionally or individually. Proportional allocation makes each task instance be proportionally split into a multimedia server instance. Individual allocation maps each multimedia task instance individually to a server instance. The latter approach can provide deterministic guarantee for each execution of the multimedia task instance, since the multimedia server is guaranteed too.

In order to achieve a given quality of service (QoS), the computation time and period of the multimedia server must be properly defined. If hard-real time tasks need more CPU time than it is available to meet all tasks, QoS of multimedia server may be degraded to alleviate system overload. This can be accomplished by decreasing the computation time of a multimedia server, increasing its period, or eliminating some of its instances. The problem is how to relate variation of these parameters to the schedulability of hard-real time tasks. This is not an easy issue, since as stated before the problem of scheduling hard real-time tasks is by itself generally NP-hard. The solution presented uses an iterative algorithm to adjust the load of multimedia tasks and hard real-time tasks. When a new task needs execution, the algorithm attempts to meet all task deadlines. If this is not possible, a loop is initiated where CPU time devoted to multimedia tasks is successively adjusted and the overall task set is tested. The times dedicated to the multimedia tasks and the hard real-time tasks are monitored through the concepts of multimedia server ratio (R_s) and hard real-time task ratio (R_r). The hard real-time task ratio R_r is the sum of the execution time of hard real-time tasks divided by the schedule length. The schedule length is computed as the difference between the latest hard real-time task deadline minus the current time. Similarly, the multimedia server ratio R_r is the sum of the computation time of server instances divided by the schedule length. A simple schedulability test in this approach is thus given by:

$$\mu = R_s + R_r \leq 1 \tag{4}$$

since a processor cannot have utilization greater than hundred percent. This mechanism has the disadvantage of degrading multimedia QoS only in terms of server computation time and its period (i.e. multimedia server ratio). Therefore, the impact of this adjustment in the final application is not addressed. Another remark is that hard-real time applications are very special purpose applications. In a system really safety critical it might not be a good idea mixing different kinds of tasks in a processor.

An important argument in favor of dedicated hardware used to deal with multimedia tasks is the proportionally larger time needed to process interrupts that manipulate audio and video. The high data rates of several streams imply that the operating system implementation must be highly efficient if the CPU is to be involved with data transfer and manipulation. In fact, the overheads of I/O processing can impact the ability to guarantee multimedia processing at a high frequency. A description of this issue can be found in [15]. In this sense, a feasible solution is applying a Digital Signal Processor (DSP) subsystem to deal with this kind of data. DSP operating systems are simpler than general purpose operating systems. Some higher level services have been set aside for the favor of highly efficient implementations which can meet the needs of fast interrupt rates. For instance, two different DSP kernels are Mwave/OS from IBM and SPOX from Spectron Microsystems. Those systems provide admission control to ensure that the time requirements of incoming tasks can be met while the previous executing tasks keep their guarantees. Mwave/OS uses EDF scheduling to share CPU time, while SPOX uses fixed priority scheduling methods.

IV.2 Integrated Scheduling

Using no special hardware to deal with real-time tasks imposes the problem of sharing CPU bandwidth between several classes of tasks in a way that conflicting objectives are met. For instance, hard-real time applications require a strict determinism to handle the maximum processing delays experienced by their tasks. Non real-time applications may require high throughput, but this must not hold back the time guarantees of real-time applications. In between, multimedia applications need a mix of throughput and time guarantees in conformance with their QoS requirements. Each class of tasks has its own requirements and therefore demands special purpose scheduling algorithms for better achievement of its goal. Solutions to this problem generally focus on implementing different scheduling policies for the various classes of tasks much in the way previously discussed for integrating hard-real time tasks and multimedia tasks. The difference here is that no special hardware support is present and therefore real-time tasks share CPU bandwidth with non real-time tasks. Some of the approaches do not support hard-real time tasks, and they only deal with soft real-time applications. The basic idea is hence to share CPU bandwidth among the several classes of applications, apply a specific algorithm to each class inside its CPU time share and at the same time coordinate the achievement of conflicting class goals.

The soft real-time framework proposed in the previously cited MMOSS project is an example of a solution to the multimedia scheduler problem. This framework consists of process taxonomy, timing enforcement models and a scheduling subsystem. Processes can be classified into one of three groups. The "best-effort" class comprises those tasks that need no execution time guarantee. Tasks with timing constraints belong to the "sure" or "maybe" classes. None of these tasks has strict guarantees of execution time. This taxonomy imposes an order of service to the tasks in the sense that a class may be prioritized over another. A timing enforcement model specifies how the scheduler will proceed in order to maintain the quality of service initially promised. The project specifies three such models. In the cooperative approach the scheduler assumes that the timing constraints of real-time tasks will be observed strictly by the tasks. Therefore no further monitoring is implemented. In the semi-imperative timing enforcement model the scheduler monitors task execution to observe any deadline violation. If a deviation of the timing specification occurs, an appropriate handler must be invoked. Finally, the imperative model enforces strict observance of task deadlines. This means that a task will be preempted by its deadline time although its service is incomplete. The use of one of these three models depends on the needs of the applications and the system environment. The scheduling subsystem contains an admission controller and a dispatcher implementing basic scheduling methods like: CO-SCHEDULE and SIM-SCHEDULE. These methods reflect the cooperative timing enforcement model and semi-imperative timing enforcement model respectively. Both scheduling methods are designed by integrating some elements from rate-monotonic, priority-based scheduling and weighted round-robin. The CO-SCHEDULE scheduler supports a set of time constrained processes being scheduled according to rate-monotonic algorithm. Any time constraint task has precedence over "best-effort" tasks, thus being able to preempt them. The tasks that belong to the "best-effort" class are scheduled using weighted round-robin. Since the timing enforcement model implemented is cooperative, no monitoring of task execution time is performed. Therefore it is impossible to have good control of timing constraints violations caused by imprecise estimation of processor usage. This solution is clearly poor to meet fairness in the share of the CPU among different task classes. The SIM-SCHEDULE tries to minimize this problem by preempting a task that occupies the processor more than it was supposed to do and invoking a handler that will decide the action that must be performed. A feasible action is to resume the preempted task execution.

A more elegant and effective way of splitting CPU bandwidth was presented in [16]. The solution is based on a hierarchical scheduler that defines different levels of scheduling in a tree structure. Each task in the system belongs to exactly one leaf node, and each node in the tree represents an application class or a group of application classes (Figure 2). In this approach, each internal node allocates a fraction of the CPU bandwidth, which is distributed to its children. Tasks in a leaf node are scheduled using an algorithm appropriate for the application requirements. For instance, a round-robin scheduler can be used for non-real time tasks. Intermediate nodes are scheduled by the Start-time Fair Queuing algorithm (SFQ). This algorithm has the following properties:

1. SFQ achieves fair allocation of CPU regardless of variation in available processing bandwidth. In fact, it is a near-optimum fair scheduling algorithm since no other known algorithm achieves better fairness;
2. SFQ does not require a priori knowledge of computational requirements of tasks;
3. SFQ provides bounds on maximum delay incurred and minimum throughput achieved by the tasks;

A brief description of SFQ is given next. Assume a set of tasks $S = T_1 ... T_i ... T_n$. Each task $T_i \in S$ has an associated weight w_i that indicates the proportion of CPU bandwidth required by it. Each task accesses the CPU for a variable length *quantum* at a time. Now, let us define:

1. $q(j, T_i)$: j^{th} *quantum* of time of task T_i;
2. $l(j, T_i)$: duration of j^{th} *quantum* of time of task T_i;
3. $A[q(j, T_i)]$: time at which the j^{th} *quantum* of time of task T_i is requested;

The following parameters are used by the SFQ algorithm to define the task scheduling order:

1. $S(T_i)$: start tag of task T_i;
2. $F(T_i)$: finish tag of task T_i;
3. $v(t)$: virtual time, where t is a physical clock time;

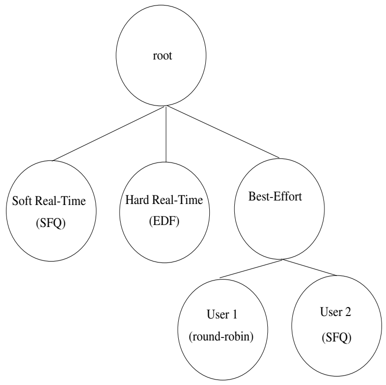

Figure 2 - Quality of Service Management

First, the finish time of every task has initial value equal zero. The virtual time used by the scheduler also is initiated to zero. When the CPU is busy at time t, v(t) is defined to be equal the start tag of the task being executed. When the CPU is idle, $v(t)$ is set to the maximum value of finish tag currently assigned to a task. When a task T_i requests its q^{th} *quantum*, it is stamped with start tag $S(T_i)$ computed as the maximum value between its current finish tag $F(T_i)$ and the virtual time of this request $v(A[q(j,T_i)])$. When the q^{th} quantum finishes, the finish tag of task T_i is incremented as $F(T_i) = S(T_i) + l(j,T_i)/w_i$. Tasks are always served in increasing order of the start tags and ties are broken arbitrarily.

The properties of this algorithm indicate that it is well suited for the integration of tasks of different classes. In particular, it is a highly adaptive approach in the sense that there are no static reservation of bandwidth: if there is only one class of tasks in the system it will receive hundred percent of CPU time. It also avoids that an overload of tasks belonging to a specific class degrades the specified performance of tasks of other classes. SFQ presents another characteristic that is highly desirable in scheduling multimedia tasks: it does not require *a priori* knowledge of computational requirements of tasks. As stated before, a problem of using EDF and RM to schedule multimedia tasks is that by assumption the worst case execution time must be known *a priori* and is constant. If it does not happen, those algorithms can degrade their performance in an unpredictable way. However, the amount of execution time needed, for example to play back a single frame of video, can vary a lot as a result of changes in scene and contents making it difficult to define a tight upper bound. A conservative upper bound can be defined at cost of poor utilization of the CPU. Therefore, SFQ can be used to schedule leaf nodes of the tree structure representing multimedia tasks besides intermediate nodes in the hierarchical structure. Its ability to provide bounds on maximum delay incurred and minimum throughput achieved by the tasks is highly desirable in scheduling multimedia tasks.

The QoS manager of a hierarchical scheduler described in the previous paragraph is implemented by the use of deterministic or statistical admission control algorithm which utilizes the capacity allocated to a given class to determine if a new request can be satisfied. Observe that the arrival of new best-effort task requests does not require *a priori* admission control of the QoS manager, because the SFQ algorithm by itself enforces a bandwidth share that it always observed. Nevertheless, a specific policy of resource sharing to the non real-time tasks may exist that controls admission of tasks belonging to this group. A QoS manager can also dynamically change the hierarchical partition to reflect the relative importance of various applications. The approach described above does not address how one can be sure that real-time tasks will meet their deadlines using this scheme.

All the methods discussed until now do not consider the existence of precedence relations between tasks. Solutions to schedule a set of tasks with this kind of constraints are normally based on a graph model that shows such dependencies and eventually other constraints presented by the task set to be scheduled. In [17] a Multimedia Task Graph is proposed specifically to address the problem of multimedia task scheduling with precedence constraints. An *MTG* is a labeled, directed and acyclic graph. Each node of this graph represents the computation of a task and each link represents a precedence relation between the tasks of the two adjacent nodes. The precedence between two tasks can be a result of messages sent in a communication link. Therefore, this model is appropriate for also defining distributed scheduling. There are three types of tasks in the *MTG* model: starting tasks, processing tasks, and terminal tasks. Starting tasks are source nodes of the graph and represent activities related to input of continuous data stream in the system. Those tasks are periodic for they must collect data at some rate. Processing tasks are represented by all the nodes in the graph that are neither sources nor sinks. They perform some kind of processing in the incoming data stream and pass the data away. Those tasks must also be periodic since they are receiving periodic input. Finally, terminal tasks are represented by sink nodes in the *MTG* and are responsible to generate output. Each terminal task has a specific deadline and/or a utility function. In the graph, two tasks that have a vertical relationship are dependent and must be executed in a given order. Figure 3 is an example of such a graph.

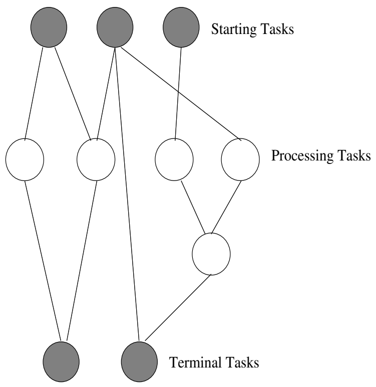

Figure 3 - MTG example

A utility function is a common concept in the optimization area and gives a measure of the quality of a solution to a given problem. In multimedia systems, the strict observance of deadline is not mandatory, but the quality of a scheduling is increased if tasks are executed before their deadlines. Therefore, a utility function provides a way to evaluate a given schedule. In real-time application this function is normally monotonically decreasing with time, which means that a schedule that has more delays has less quality.

The MTG approach may be used to define static schedules. It means that all scheduling decisions are made off-line and a dispatch table is responsible for implementing those decisions. This permits the use of time consuming heuristics to define near optimal schedules. The quality of each solution is evaluated using the utility function. The MTG model may also be implemented giving support to a modified version of EDF that at each time selects for execution the task with earliest deadline that is ready for execution at the moment.

Distributed Multimedia applications introduce new design challenges to the scheduler. The problem of meeting time constraints of tasks must also face the sharing of communication media besides other resources normally shared. The real-time Mach microkernel-based operating system [18] focuses on the subject of distributed real-time OS mechanisms and services for developing and supporting real-time and multimedia applications. The philosophy behind RT-Mach is firmly based on real-time scheduling theory and in particular on priority-driven preemptive scheduling. The principles behind the resource management philosophy have many implications to OS subsystems including scheduling policies.

RT-Mach implements the concept of a resource kernel. This means that the kernel provides timely, guaranteed and protected access to all system resources. Applications need to specify only their resource demands leaving the kernel to satisfy those demands using hidden resource management schemes. The basic mechanism is reserving resources as required by the applications and enforces this reservation by admission control in the same generic model applied by all multimedia systems. Based on the timeliness requirements of reservations, the kernel prioritizes them, and executes a higher priority reservation before a lower priority reservation if both are eligible to execute.

The RT-Mach resource reservation model employs the following parameters: computation time C every t time-units for managing the network utilization of a resource, a deadline D for meeting timeliness requirements, a starting time S of the resource allocation, and L, the life-time of the resource allocation. These parameters C, t, D, S and L are referred as explicit parameters of the reservation model. The semantics are as follows. Each reservation will be allocated C units of usage time every t units of absolute time. These C units of usage time will be guaranteed to be available for consumption before D units of time after the beginning of every periodic interval. The guarantees start at time S and terminate at time $S + L$. The resource kernel also implicitly derives, tracks and enforces the implicit parameter B for each reservation in the system. B represents the maximum (desirably bounded) time that a reservation instance must wait for lower priority reservations while executing. If its B is unbounded, a reservation cannot meet its deadline. Therefore, a priority inheritance protocol is applied when a reservation blocks, waiting for a lower priority reservation to release a lock. Chorus is another example of a microkernel with real-time features. Despite of that, Chorus is not directly suitable for the support of distributed multimedia applications because of its lack of support for QoS control and resource reservation. In [5], it is proposed an extension to the Chorus API with new low level calls and abstractions to support distributed continuous media applications. The scheduling system uses EDF. However, there is no attempt to provide absolute guarantees as in the classical approach. The implementation architecture of the real-time scheduling comprises a single kernel scheduler and cooperating user level thread schedulers in each actor. An actor is an abstraction of the Chorus micro-kernel and corresponds to an address space and a set of resources. Therefore this scheme is bi-level. Each user scheduler is responsible for scheduling user threads in its actor using EDF. The kernel scheduler chooses to execute the kernel thread of the actor containing the user thread with globally earliest deadline. The necessary information exchange between the kernel scheduler and the user level schedulers is done via a combination of shared memory and upcalls. This bi-level mechanism results in considerable time saving as context switching is cheaper at user level. The extension proposed to Chorus attempts to integrate communication and scheduling for more effective scheduling decisions. In conventional operating systems, communications and scheduling are generally not well integrated. Newly arriving messages are received by a system thread running a transport protocol then placed in a buffer. An integrated approach can provide a more effective scheduling by observing the deadlines of the arriving messages and making the proper decisions to meet the timing constraints.

Dynamic mechanisms for globally scheduling soft-real time tasks in distributed systems can be found in [19] and [20]. These mechanisms may be used to improve the number of tasks scheduled in accordance with their time requirements. In a distributed system it may occur that some nodes become overloaded while others are almost idle. Dynamic global schedule schemes try to take advantage of this situation transferring the execution of some tasks from overloaded nodes to idle nodes. The problem here is how to identify the loading state of each node in order to make a good transfer decision. Load states change dynamically and nodes cannot know precisely the actual state of remote nodes due to communication delays in the transfer of state messages. The simplest solution would be to choose a remote node randomly and transfer the execution of a task to it.

More elaborate schemes use state information exchanged between the nodes to define a good destination to a transferring task. Focussed addressing [20] is a method that relies in a very poor inference procedure to guess the load state of remote nodes. Basically, a node that wants to transfer execution of a task "believes" with little restriction in the old state information that it has about remote nodes and make transfer decisions based on them. It is has the advantage of being a fast method but not very precise.

Bidding [20] is a mechanism that makes an overloaded node send an offer for the system when it wants to transfer a task execution. All nodes that consider themselves able to execute the tasks reply with a bid. Then, the overloaded node can choose what is the best bid. This method is more precise than the previous two but is more timing consuming and the deadline of the transferring task can be missed during the process.

A combination of the Bidding and Focussed address is called Flexible method [20]. When a node needs to transfer a task it performs a simple focussed address algorithm, but additionally makes an offer to the system to be replied to the chosen node of the focussed address method. If the chosen node of the first method cannot cope with the transfer task, it will have a set of bid that will help him to choose another node to redirect the transferred task execution.

Finally, a Bayesian [19] approach is a mechanism that uses Bayesian theory to infer the system load based on previous state information collected through messages. This scheme is more precise than all the above providing better results [19].

V Conclusions

A successful approach to build systems that give support to multimedia applications involves cooperation of many different parts of an operating system. The basic problem to are solved is how to share resources between processes in a way that timing constraints of services be met. In a distributed system this concern may involve for instance the communication media sharing. Processors are important resources of a computer system and the task scheduling problem is related to finding ways of distributing processors cycles between processes in order to achieve their logical and temporal constraints. Despite of the special characteristics presented by multimedia tasks many of the solutions applied for their scheduling are based on small modifications of hard-real time algorithms. Solutions are grouped into two major approaches: integrated scheduling and autonomous scheduling. Integrated scheduling has the advantage of less hardware, flexibility and scalability but introduces more complexity to the scheduler for its general-purpose characteristic. Autonomous scheduling requires minimal modification to the host operating systems, but more complex hardware is needed. Distributed Multimedia applications introduce new design challenges to the scheduler. However, the soft-real time characteristic of multimedia applications can be exploited in distributed situations to increase the number of tasks executed in accordance with their timing constraints. Specifically, task execution transfer can be applied to alleviate overloaded nodes.

References

[1] J. Stankovic, "A serious problem for next-generation systems," *IEEE Computer*, vol. 21, no. 12, pp. 10–18, 1988.

[2] D. Hehmann, R. Schultz, T. Schuett, and R. Steinmetz, "Implementing heits: Architecture and implementation strategy of the heidelberg high speed transport system," in *Second International Workshop on Network and Operating System Support for Digital Audio and Video*, (Germany), 1991.

[3] K. Jeffay, D. Stone, and D. Poirer, "Yartos-kernel support for efficient, predictable real-time systems," *Real-Time Systems Newsletter*, vol. 7, no. 4, pp. 8–13, 1991.

[4] J. Neih and M. Lam, "Smart unix svr4 support for multimedia applications," in *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, (Canada), June 1997.

[5] G. Coulson, G. Blair, and P. Robin, "Micro-kernel support for continuous media in distributed systems," *Computer Networks and ISDN systems*, no. 26, pp. 1323–1341, 1994.

[6] R. Garey and D. Johnson, "Complexity bounds for multiprocessor scheduling with resource constraints," *SIAM J. Computing*, vol. 4, no. 3, pp. 187–200, 1975.

[7] A. Mok, *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment.* PhD thesis, Computer Science and Electrical Engineering Department–MIT, 1983.

[8] L. Sha, J. Lehoczky, and R. Rajukmar, *Priority Inheritance Protocols: An Approach to Real-Time Synchronization*, pp. 1124–1142. No. 8, 1990.

[9] S. Chen, Stankovic, and K. J. Ramamritham, *Scheduling Algorithms for Hard Real-Time Systems.* IEEE Computer Society Press, 1988.

[10] M. DiNatale and J. Stankovic, "Applicability of simulated annealing methods to real-time scheduling and jitter control," in *16th IEEE Real-Time Systems Symposium*, (Italy), December 1995.

[11] C. Liu and J. Layland, "Scheduling algorithms for multiprogramming in a hard-real environment," *Journal of the ACM*, vol. 20, no. 1, pp. 46–61, 1973.

[12] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, "The design and implementation of an operating system to support distributed multimedia applications," in *IEEE Journal on Selected Areas in Communications*, vol. 14, September 1996.

[13] C. Fan, "Mmoss: Soft real-time operating system support in a multimedia communication subsystem," in *19th IFAC/IFIP Workshop on Real-Time Programming*, June 1994.

[14] H. Kaneko, J. Stankovic, S. Sen, and K. Ramamritham, "Integrated scheduling of multimedia and hard real-time systems," in *17th IEEE Real-Time Systems Symposium*, pp. 206–17, December 1996.

[15] D. Katcher, K. Kettler, and J. Strosnider, "Real-time operating systems for multimedia processing," in *Proceedings of Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, May 1995.

[16] P. Goyal, X. Guo, and H. Vin, "A hierarchical cpu scheduler for multimedia operating systems," in *Second Symposium on Operating Systems Design and Implementations*, pp. 107–122, October 1996.

[17] M. Hsing, "Scheduling dependent real-time multimedia tasks on distributed systems," in *19th Annual International Computer Software and Applications Conference*, pp. 306–11, August 1995.

[18] H. Tokuda, T. Nakajima, and P. Rao, "Real-time mach: Towards a predictable real-time system," in *Proceedings of USENIX Mach Workshop*, October 1990.

[19] K. Shin and C. Hou, "Design and evaluation of effective load sharing in distributed real-time systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 7, pp. 704–719, 1994.

[20] J. Stankovic, K. Ramamritham, and S. Cheng, "Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems," *IEEE Transactions on Computers*, vol. C34, no. 12, pp. 1130–1143, 1985.