Inside the Heidelberg Multimedia Operating System Support: Real-Time Processing of Continuous Media in OS/2

March 17, 1993

Andreas Mauthe*
Werner Schulz°
Ralf Steinmetz°

°IBM European Networking Center
Tiergartenstraße 8
D-6900 Heidelberg 1

University of Mannheim
Lehrstuhl für Praktische Informatik IV
D-6800 Mannheim 1

Contents

1.0 Introduction

The Heidelberg Transport System (HeiTS) is a new-generation end-to-end communication system currently under development at the IBM European Networking Center (ENC) in Heidelberg. HeiTS is aimed at a heterogeneous environment comprising several computers with different operating systems and a variety of underlying local, metropolitan, and wide-area networks. It incorporates both end-system and gateway communication functions.

Conventional communication networks, e.g. Ethernet, were designed for asynchronous data traffic only. In new high-speed networks an integration of different traffic types can be observed. FDDI, for example, supports asynchronous and synchronous traffic. This integration in the network leads to an integrated communication system as a whole. HeiTS coexists with the well-established communication systems like TCP-IP, Netbios or APPC It is designed to support multimedia communication, in particular the transfer of digital audio and video.

Interactive and conversational multimedia applications demand for real-time processing of audio and video data. Such data is said to be correct, if and only if data is delivered errorless within certain time limits. Non-audio and non-video data may also be subject to time critical processing. Consider the example of a telepointer in a shared window. In this case, timing constraints may be useful in order for a particular user to say "pointing at this object we see ...." At this moment the pointer must be placed on the respective object at the screen of all recipients of the data. The guarantee of a maximal end-to-end delay requires real-time processing.

This real-time processing is needed by several multimedia system components, the communication system is just one of them. Audio and video data retrieval from a multimedia database, interactive multimedia document presentation and all types of multimedia applications are examples of such components. Therefore, a common support of real-time processing tailored for audio and video data is required.

The operating systems in the end-systems of HeiTS are OS/2 [1] and AIX. Neither of them is conceived for the extensive support of real-time processing of audio and video data. Enhancements for multimedia on top of the operating system are available as the "Multimedia Presentation Manager/2" for OS/2. This extensions focus on the local audio and video data transfer making use of kernel (ring 0) device driver programming. Alternatively, a real-time scheduler can be used. In this document we elaborate an inside view of our design to incorporate real-time processing into the OS/2 environment. It allows for user level programming making use of real-time scheduling policies adapted to audio and video data processing in a networked environment. The integration of the MMPM/2 with our real-time scheduler is in progress.

The OS/2 system scheduler is priority driven. It schedules each task according to its particular priority. The setting of task priorities can be done by each thread within a process. A general mapping algorithm between the real-time processing and the available priority scheme is required. This algorithm must take into account the environmental constraints (timer resolution, possibility of preemption, etc.). This paper outlines a solution which was implemented in Heidelberg. Additionally we present an exhaustive elaboration of traditional real-time scheduling applied to continuous media which was performed as part of this project.

[1] OS/2,PS/2, PM, AIX, RISC System/6000 are trademarks of IBM Corporation. ActionMedia is a trademark of Intel Corporation.

This document is organized in five chapters. In chapter two we discuss multimedia, real-time and their mutual relationship. This chapter also includes a brief presentation of the related resource management. Several traditional real-time scheduling algorithms, their suitability towards continuous media processing and implemented multimedia prototypes are introduced in chapter three. Chapter four starts with a brief summary of the OS/2 real-time capabilities. Subsequently we introduce two alternative concepts to schedule continuous media tasks under OS/2, present our actual design and the experiences with the running prototype. The final chapter summarizes the work presented in this paper.

2.0 Multimedia and Real-Time

2.1 Multimedia

Multimedia represents a growing area of interest in business, engineering and science. Multimedia in computer science refers to the employment of different media in computers. Computers are used as communication tools. The communication is either between humans and the computer or among humans using the computer [26]. We distinguish between four major types of computer processed data: Text, graphics, audio and video. Text and graphics are the traditional media with a time-independent presentation. Other media, such as audio and video have time-dependent data values. Therefore, their processing requirements on a computer system are different. On computer networks, for example, they are characterized by their sensitivity to delays, high bandwidth requirements and tolerance of high error rates [84].

A multimedia system should be able to handle each type of media independently. Multimedia systems have to be distinguished from other technologies such as television, due to the ability to allow the user more interaction with the system [57].

Specifically, a ***multimedia system*** is defined as *a system that is characterized by the computer-controlled generation, manipulation, presentation, storage and communication of independent discrete and continuous media* [72]. ***Multimedia*** is perceived as the *logical, inevitable convergence of four major technologies: telecommunication, publishing, television and computing* [83]. In the view of these authors a ***multimedia system*** is a desktop or network application which uses at least three of the following media types: video, graphics, text, audio and animation. Hence, a computer system for the processing of multimedia application has to be able to handle many different kinds of media. The innovation that is provided by multimedia systems is the integration of all these media into a single system, obscuring the lines between computing, telecommunications, and even mass media. Since a great deal of experience has been gleaned over the past forty years the major challenge remain is the incorporation of continuous media into computer systems [26].

The main developments in computer science and electrical engineering that support multimedia systems are fast processors, high-speed networks, large-capacity storage devices, new algorithms and data structures, graphics systems, innovative methods for human computer interaction, real-time systems, object oriented programming, etc. [16].

Existing multimedia systems are, for example, employed in education (e.g. as hypermedia systems [65]), Computer Supported Cooperative Work (CSCW) [19], and as information systems (e.g for the presentation of art in museums [76]).

The expression ***continuous media*** (CM) for audio and video is derived from the way in which they are perceived by humans. CM consists of consecutive time-dependent information units. Time attributes semantics to the media. In this sense CM differs from common ***discrete media*** (DM) processed on computer such as text and graphics which consist of time independent information values [26].

The representation of CM in a digital system is discrete. It consists of logical data units (LDU) being, for example, single audio-samples or video-frames. The information content of these LDU is a value of a basic data type and it represents a piece of the original data over a certain period of time. The triple m = (V, T, U) defines the CM-data. V is the value of the basic data type, T is the time value and U is the duration of the digital stream. From these parameters we can derive the life-span of

the CM-data $L = [T, T + U)$.[2] Specifically, CM is characterized by a periodic continuous data stream. Aperiodic continuous streams can be transformed into periodic streams.

The digital representation of CM allows its handling by standard system components such as the CPU, main memory, disk, or network. Other applications can be processed concurrently to the execution of CM operations with no adverse effects due to contention for hardware resources. Further, CM and DM can be handled in the same software framework (operating systems, network protocols, window system, programming languages) [23].

2.2 Real-Time

The German national institute for standardization DIN defines a real-time process in a computer system *as a process which delivers the results of the processing in a given time-span. Programs for the processing of the data have to be available during the whole run-time of the system. The data may require processing to an a priori known time, or occur at a previously not known instant* [14].

A real-time system has the permanent task to receive information from the environment and to deliver it to the environment within time constraints [6]. Speed and efficiency are not the main characteristics of a real-time system. The correctness of a computation in a real-time system depends not only on the results of the computation but also on the time at which they are presented [67]. In a multimedia application a failure occurs when the data of a video or a piece of music is presented too fast or when it is presented with a considerable delay. Therefore, the time behavior of a real-time system has to be both deterministic and predictable [22; 24]. In particular two aspects have to be considered [63]:

1. The processing of tasks in a strongly restricted time interval.
2. Temporal and logical interdependence between two processes that require processing at the same time due to their internal and external restrictions.

Summarized, a real-time system can not only fail because of massive hardware or software failures, but also because the system is unable to execute its critical workload in time [40].

In real-time system we have both ***hard*** and ***soft deadlines*** which represent the latest time for the presentation of a processing result. A ***soft deadline*** is a deadline which cannot be exactly determined and where failing does not produce an unacceptable result, e.g. starting and arrival times of planes or trains can considered to be as soft deadlines. ***Hard deadlines*** are determined by the physical characteristic of real-time processes. They mark the border between normal and failing behavior. Failing such a deadline causes costs which can be measured in monetary (e.g. inefficient use of raw materials in a process control system), aesthetical (e.g. garbled output from audio or video), or human and environmental terms (e.g. accidents due to untimely control in a nuclear power plant or fly-by-wire avionics system) [36].

The deterministic and predictable behavior of a real-time system includes a guarantee requirement for time-critical tasks. Such guarantees cannot be assured for events that occur at random intervals with unknown arrival times, processing requirements or deadlines. Further, all given guarantees are only valid under the premise that no processing machine collapses during run-time of real-time processes. Summarized, task scheduling is a matter of both reliability and performance [40].

[2] ") " indicates that $T + U$ is not part of the valid time span

A real-time system is distinguished by three feature (c.f. [67]):

1. ***Predictably fast response to time-critical events and accurate timing information.*** Constrains such as an upper bound or an average value have to be imposed on these response times. In the control system of a nuclear power plant, e.g., the response to a malfunction must occur within a well-defined period in order to avoid a potential disaster.
2. ***A high degree of schedulability.*** Schedulability refers to the degree of resource utilization at, or below which the deadline of each time-critical task can be ensured.
3. ***Stability under transient overload.*** Under system overload the processing of critical tasks to their deadlines must be ensured. These critical tasks are vital to the basic functionality provided by the system.

Manufacturing process management is a main application area for the use of real-time systems. Such a process control system is responsible for real-time monitoring and control. Real-time systems are also used as command and control system in fly-by-wire aircraft, automobile anti-lock braking systems and the control of nuclear power plants [40]. New areas for real-time systems are computer conferencing and multimedia.

2.3 Real-Time in Multimedia

If concurrent processes handling CM and DM share one machine, the operating system has to provide them with the system resources they need and it must resolve resource conflicts. In traditional multitasking systems such as UNIX, "fairness" is the main criterion for resource administration. This criterion is insufficient for handling CM. Apart from high *throughput* requirements, CM impose *timing* demands on computer systems that result from the periodically changing value of CM data: Each single value in an audio or video stream represents stream information for some fraction of time. Changes in the times at which values are played or recorded result in a modification of the original data semantics and must not happen unintentionally. To ensure correct timing, *delay* and *jitter* for the handling of CM have to be bounded if some I/O equipment (and, obviously, some human user sitting in front of it) is involved in the application [15]. Without I/O (*e.g.*, when copying a video file), the handling of CM is not time-critical.

To fulfill the timing requirements of CM, the operating system must use *real-time scheduling* techniques. These techniques have to be applied to all system resources involved in the CM-data processing. The entire *end-to-end* data path is involved, not just the CPU. Networks and disks often contribute significantly to delay and jitter. With DMA capabilities of controllers, continuous-media data may not even "pass" through the CPU. To support the function of these schedulers, a deterministic behavior of the operating system has to be ensured. Unpredictable effects of caching, process switches or page faults of a virtual memory system, *e.g.*, can ruin any carefully planned schedule.

Unfortunately, existing real-time systems are not well suited to support CM. Real-time scheduling is traditionally used for *command and control systems* in application areas such as factory automation or aircraft piloting. For these applications, a large variety of real-time tasks, a plethora of I/O devices to interface with the technical process to be controlled, and high fault-tolerance requirements (that somewhat counteract to real-time scheduling efforts) are typical. **CM have different (in fact, more favorable) real-time requirements:**

- A sequence of digital continuous-media data results from periodically sampling a sound or image signal. Hence, in processing the items of such a data sequence,

all time-critical operations are periodic. Schedulability considerations for periodic tasks are much easier than for sporadic ones [56].

- For many applications missing a deadline in a multimedia system is - although it should be avoided - not a severe failure. It may even be unnoticed: If an uncompressed video frame (or parts of it) are not available on time it can simply be dropped. The human viewer will hardly notice it, provided this does not happen for a contiguous sequence of frames. For audio, requirements are higher because the human ear is more sensitive to audio gaps than the human eye is to video jitter.
- The fault-tolerance requirements of continuous-media systems are usually less strict than for those real-time systems that have physical impact. The failure of a continuous-media system will not directly lead to the destruction of technical equipment or constitute a threat to human life.
- The bandwidth demand of CM is not always *that* stringent. As some compression algorithms are capable of using different compression ratios - leading to different qualities - the required bandwidth can be negotiated. If not enough capacity for full quality is available the application may also accept a reduced quality (instead of no service at all). The quality may also be adjusted dynamically to the available bandwidth, *e.g.*, by changing encoding parameters.

In a traditional real-time system, timing requirements result from the physical characteristics of the technical process to be controlled, *i.e.*, they are provided externally. Some continuous-media applications have to meet external requirements, too. A distributed music rehearsal is a futuristic example: Music played by one musician on an instrument connected to his workstation has to be made available to all other members of the orchestra within a few milliseconds, otherwise the underlying knowledge of a global unique time is disturbed. If human users are involved in only the input or only the output of CM, delay bounds are flexible. Consider the play-back of a video from a remote disk. The actual delay of a single video frame to be transferred from the disk to the monitor is unimportant. Frames must only arrive in a regular fashion. The user will notice any difference in delay only in the time it takes for the first video frame to be displayed. While the traditional real-time scheduling problem is to find a schedule for a set of processes with given delay bounds, the main problem in multimedia systems is to find reasonable delay bounds so that a set of processes is schedulable.

CM are an addition to - not a substitute for - the DM already available in computing systems. In the future multimedia systems, time-critical continuous-media tasks and non-critical discrete-media processes will run concurrently. Such a mixed operation imposes new demands on scheduling as traditional systems usually have to support only one class of processes. The operating system must fulfill two conflicting goals:

- Time-critical processes must never be subject to *priority inversion* (*i.e.*, be kept from running by non-critical processes for an indefinite time) [62].
- Uncritical processes should not suffer from *starvation* because time-critical processes are executed.

A solution to this conflict is possible if multimedia systems have control over the time-critical workload making use of the resource management [82].

2.4 Resource Management in HeiTS

A distributed multimedia system requires guaranteed processing of CM. The ***quality of service*** (QoS) requirements depend upon the type of data and the nature of the supported applications [74]. We consider three relevant QoS parameters for the processing and transfer of CM-data [82]:

1. The ***throughput*** parameter determines the data rate a connection needs in order to satisfy application requirements. The maximal achievable throughput on the CPU depends on the algorithm that is employed to schedule time-critical tasks.
2. We distinguish between two kinds of ***delays***:
   a. The ***delay at the resource*** is the maximum time span for the completion of a certain task at this resource.
   b. The ***end-to-end delay*** is the the total delay for a data unit to be transmitted from the source to its destination. It is the sum of the delays of all involved resources.
3. The ***reliability*** defines error detection and correction mechanisms used for the transmission and processing of multimedia tasks. We distinguish three classes of error treatment: ignore, indicate and correct. It is important to notice that error correction through re-transmission is rarely appropriate for time-critical data because the re-transmitted message will usually arrive late. On the other hand, single or small errors might not be noticed by the user, and thus, uncompressed data might not even need any error correction. For compressed data, especially encoded video, error detection and the substitution of corrupted or late packets might be useful because a single error may have continuing effects. In terms of reliability the CPU represents little difficulties as no errors occur at the processing of a task.

To guarantee the QoS-parameters the ***resource managers*** allocates for each connection the necessary resources (e.g. CPU, communication network). They ensure that a new connection does not violate performance guarantees already given to existing connections [27]. During the connection establishment the QoS parameters are usually negotiated, mediating the application's needs with the current capabilities of the communication system. There are different ways to negotiate the QoS parameters. The simplest negotiation scheme is the specification of the QoS through the application. The resource manager checks whether this QoS can be provided or not. A more elaborate method is to optimize single parameters. In this case two parameters are determined by the application (e.g. throughput and reliability), the resource manager then calculates the best achievable value for the third parameter (e.g. delay) [82].

A resource manager has four tasks:

1. *Schedulability Test:* The resource manager checks with the given QoS parameter if there is enough remaining resource bandwidth available to handle the new connection.
2. *QoS Calculation:* After the schedulability test the resource manager calculates the best possible performance the resource can provide for the new connection.
3. *Resource Reservation:* The resource manager allocates the required capacity in order to meet the QoS guarantees for each connection is reserved.
4. *Resource Scheduling:* Incoming messages from connections are scheduled according to the given QoS guarantees.

This four tasks can be applied to each resource. For the CPU, real-time scheduling can be considered to be a task of the resource manager. But, in the case of process management real-time scheduling is a duty of the operating system. Therefore, the operating system must use scheduling methods which consider time constraint. The

resource manager has to perform tasks 1,2 and 3 before tasks can be scheduled. However, it must be noted that the schedulability test, QoS calculation and resource reservation depend upon the algorithm used by the scheduler.

Reservation of resources can be made either in a *pessimistic* or in an *optimistic* way:

- The *pessimistic approach* avoids resource conflicts by making reservations for the worst case, i.e. resource bandwidth for the longest processing time and the highest rate needed by a task is reserved. This leads potentially to an underutilization of resources. In a multimedia system the remaining processor time (i.e. the time reserved for traffic but not used) can be used by DM tasks. This method results in a guaranteed QoS.
- The *optimistic approach* reserves according to the average or minimum workload. This results in a *best-effort* QoS. The CPU is reserved for the average or minimum processing time and data rate needed by a task for its processing. This approach overbooks resources with the possibility of a packet loss.

Best-effort processes require the ability to detect and solve resource conflicts. Resource conflicts occur when a best-effort process exceeds its reserved processing time and other critical processes require processing. In this case the scheduler has to detect the resource conflict, to preempt the best-effort process, and to schedule another critical task. The OS/2 operating system does not supply the possibility of measuring pure processing time. Therefore, it is difficult to detect and solve resource conflicts. Another solution to this problem is the use of the following preemptive multi-level priority scheme (c.f. [82].)

1. Critical guaranteed processes
2. Critical best-effort processes
3. Processes not executing transport system software (e.g. application processes)
4. Workahead processes (both guaranteed and best-effort)

A request from a guaranteed task will preempt every running best-effort task even if the deadline of the best-effort task is closer. Hence, best-effort tasks can fail to meet their deadlines although they did not exceed their reserved processing time and there would have been a feasible schedule. To use guaranteed processes and best-effort processes concurrently one must to accept this flaw, although it is certainly not ideal.

2.5 CM-Resource Model

The resource model for HeiTS is based on the model of Linear Bounded Arrival Processes (LBAP) as described in [3]. In this model a distributed system is decomposed into a chain of resources traversed by the messages on their end-to-end trips. Examples of such resources are single schedulable devices such as CPU, or combined entities such as networks.

A LBAP is a message arrival process at a resource defined by three fixed parameters.

M = Maximum message size (byte/message)

- R = Maximum message rate (message/second)
- B = Maximum Burstiness (message)

A burst consists of messages which have arrived "ahead of schedule"

In the following this LBAP model is discussed in terms of a specific example:

Two workstations are interconnected by a LAN. A CD-player is attached to one workstation. Mono-audio data is transferred from this CD-player over the network to the other computer. There this audio data is delivered to a speaker.
This mono audio signal is sampled with 44.1 kHZ. Each sample is coded with 16 bit.
Up to 12000 bytes are assembled into one packet and transmitted over the LAN.
This results in a data rate of

$$44100\ 1/s \times \frac{16\ bit}{8\ byte/frame} = 88200\ byte/s$$

The samples on a CD are assembled to frames. This frames are the audio messages to be transmitted.
75 messages per second are transmitted.

$$\frac{88200\ byte/s}{75\ message/s} = 1176\ byte/message$$

In a packet of 12000 byte we can then have not more than

$$\frac{12000\ byte}{1176\ byte/message} \leq 10\ message$$

It obviously follows:

- M = 1176 byte/message
- R = 75 message/s
- B = 10 message

During a time interval of the length t, the ***maximal number of messages*** arriving at a resource must not exceed

$$B + R \times t\ (message)$$

e.g.: Assume t = 1 s

$$10\ message + 75\ message/s \times 1\ s = 85\ message$$

The ***Burstiness*** *B* introduces short time violations of the rate constraint. This allows the modelling of programs and devices that generate burst of messages. Bursts are, e.g., generated when data is transferred from disks in a bulk transfer mode or −as above− when frames are assembled to large packets. The maximum average data rate of a LBAP is:

$$M \times R\ (byte/second)$$

e.g.

$$1176\ byte/message \times 75\ message/s = 88200\ byte/s$$

It is guaranteed that messages are processed according to their rate. Messages which arrive "ahead of schedule" have to be queued. For delay ≤ period the ***buffer size*** is:

$$M \times (B + 1)\ (byte)$$

e.g.

$$1176\ byte/message \times 11\ message = 12936\ byte$$

The function $b(m)$ represents the ***logical backlog*** of messages. This is the number of messages which already have arrived "ahead of schedule" at the arrival of message m. Let a_i be the actual arrival time of message m_i; $0 \leq i \leq n$: then $b(i)$ is defined by:

$$b(m_0) = 0$$

$$b(m_i) = \max(0, b(m_{i-1}) - (a_i - a_{i-1})R + 1)$$

e.g.: $a_{i-1} = 1.00\ s$; $a_i = 1.01\overline{3}\ s$; $b(m_{i-1}) = 4\ s$

$$b(m_i) = \max(0.4\ message - (1.01\overline{3}\ s - 1.00\ s) \times 75\ message/s + 1) = 4\ message$$

The ***logical arrival*** time of a message m_i can then defined as:

$$l(m_i) = a(n_i) + \frac{b(m_i)}{R}$$

e.g.

$$1.01\overline{3}\ s + \frac{4\ message}{75\ message/s} = 1.0\overline{6}\ s$$

Equivalent by it can be computed as:

$$l(m_0) = a_0$$

$$l(m_i) = \max(a_i, l(m_{i-1}) + \frac{1}{R})$$

e.g.: $l(m_{i-1}) = 1.05\overline{3}\ s$

$$\max(1.01\overline{3}\ s,\ 1.05\overline{3}\ s + \frac{1\ message}{75\ message/s}) = 1.0\overline{6}\ s$$

Intuitively $l(m)$ is the earliest time the message m could have arrived if all messages had obeyed their rate.

The ***guaranteed logical delay*** of a message m denotes the maximum time between the logical arrival time of m and its latest completion. It results from the servicing time of the messages and the competition among different sessions for resources, i.e. the waiting time of the messages. If a message arrives "ahead of schedule" the actual delay is the sum of the logical delay and the time by which it arrives to early, it is then greater then the guaranteed logical delay. It can also be less then the logical delay when it is completed "ahead of schedule". The ***deadline*** $d(m)$ is derived from the delay for the processing of a message m at a resource. The deadline is the sum of the logical arrival time and its logical delay.

If a message arrives "ahead of schedule" and the resource is in an idle state, the message can be processed immediately, i.e. it is workahead. The message is then

called a ***workahead message***, the process is a workahead process. A maximum ***workahead time*** A can be specified (e.g. from the application) for each process. This results in a maximum ***workahead limit*** W.

$$W = A \times R$$

e.g.: A = 0.4 s

$$0.4\,s \times 75\,message/s = 3\,message$$

If a message is processed "ahead of schedule" the logical backlog is greater then the actual backlog.

A message is ***critical*** if it has passed its logical arrival time.

Throughout the rest of the paper the LBAP-model is used to describe the arrival processes at each resource. The resource must ensure that the arrival processes at the output interface obeys the LBAP-parameters.

3.0 Application of Traditional Real-Time Scheduling to Multimedia Data Streams

In computer science the problem of real-time processing is widely known [6; 20; 60; 66]. The Rate-monotonic algorithm to schedule periodic real-time tasks, for example, was introduced by Liu and Layland in 1973 [47]. In industrial process management and operation research (OR) scheduling is used in order to find an optimal schedule for the processing of jobs on a single processor or on multiple machines [17]. It differs from real-time scheduling in that it operates in a static environment and must not adapt to any change of workload [85]. Here, task deadlines are not hard. The major task is to get an optimal utilization of the machines. Nevertheless, there are scheduling methods and modifications of the base algorithms applied which are also used in computer science for real-time processing, e.g. shortest processing time scheduling, earliest due data and Moor's Algorithm [42].

There are many proposals to solve real-time scheduling problems with many variations of the basic problem. In order to find the best solution for our problem we analyzed various algorithms and discussed their advantages and disadvantages. In this chapter we focus on the most relevant algorithms. Most of these approaches aim to solve non-multimedia problems but, their basic ideas can be used for our purpose.

The goal of traditional scheduling is optimal throughput, optimal resource utilization, and fair queueing. In real-time scheduling the major task is to provide a schedule according to the constraints of time-critical tasks.

The scheduling algorithm has to map tasks onto resources such that all tasks meet their time requirements. Therefore, it must be possible to show, or to proof, that a scheduling algorithm applied for real-time systems fulfills the timing requirements of the task.

3.1 Real-Time Scheduling: System Model

In this section we describe the system model for the scheduling of real-time tasks. All scheduling algorithms to be introduced are based on this model. The model consists of three components:

Resources: A resource is an entity with a finite capacity that is required by the tasks for their processing. There are ***active*** resources like the CPU, and ***passive*** resources like the main memory. A resource can be used ***exclusively*** by one process or can be ***shared*** with other processes. Active resources are always exclusive. Each resource has a capacity which results from its ability to perform a certain function in a given time-span. For real-time scheduling only the temporal diversion of the resource capacity is of interest. If a resource exists only once in the system, it is called a ***single*** resource, otherwise it is a ***multiple*** resource. In our case we have to deal with an ***active, exclusive, single resource*** – the CPU. In a real-time system the scheduling algorithm has to determine a schedule for exclusive, limited resources that are used concurrently by different processes such that all of them can be processed without violating any deadlines.[3]

Tasks: A task is the schedulable entity of the system. It can be invoked to perform a particular function. In a hard real-time system, a task is characterized by its timing

[3] This notion can be extended to a model with multiple resources (e.g. CPU's) of the same type. It can also be extended to cover different resources such as memory and bandwidth for communication.

constraints as well as resource requirements [12]. In our case we consider only periodic tasks without precedence constraints, an appropriate characteristic of CM-data processing.

The time constraints of the periodic task T are characterized by the following parameters (s, e, d, p) described in [43]:

- s: Starting point
- e: Processing time of T
- d: Deadline of T
- p: Period of T
- r: Rate of T $(r = 1/p)$

$0 < e \leq d \leq p$. The starting point s is the first time where the periodic task requires processing. Afterwards, it requires processing in every period p with e processing time. At $s + (k - 1)p$ the task T is ready for the k-processing. The processing of T in period k has to be finished at $s + (k - 1)p + d$. For CM-tasks we can assume that the deadline of the period $(k - 1)$ is the ready time of period k, this is called ***congestion avoiding deadlines***: The processing time for each data unit is the period of the respective data rate.

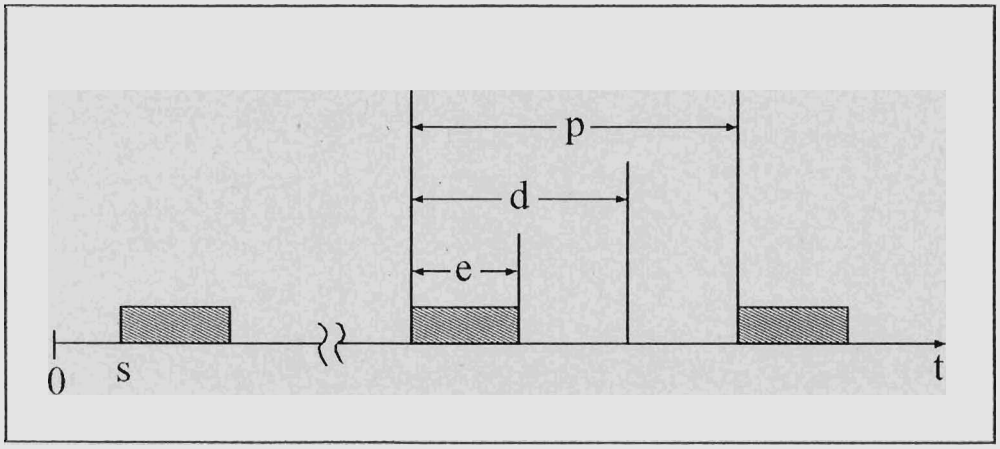

Figure 1. Characterization of Periodic Tasks

Tasks can be preemptive or non-preemptive. A preemptive task can be interrupted by the request of any task with a higher priority. Processing is continued later on. If a task is non-preemptive, the processing can not be interrupted. Any high-priority task has to wait until the low priority task is finished. The high-priority task is then subject to priority inversion. We consider CM-tasks on the CPU as preemptive.

Objectives of Scheduling Algorithms: The function of a scheduling algorithm is to determine for a given task set whether or not a schedule for executing the tasks exists, such that the timing and the resource constraints of the tasks are satisfied. Further, it has to calculate a schedule if one exists. A scheduling algorithm is said to guarantee a newly arrived task if the algorithm can find a schedule where the new task and all previously guaranteed tasks can finish processing in every period over the whole run-time to their deadlines. If a scheduling algorithm guarantees a task, it ensures that the task finishes processing prior to its deadline [12]. To guarantee tasks it must be possible to check the schedulability of the newly arrived tasks.

A major performance metric for a real-time scheduling algorithm is the ***guarantee ratio***. The guarantee ratio is the total number of guaranteed tasks versus the number of tasks which could be processed. Another performance metric is the ***processor utilization***. This is the amount of processing time used by guaranteed task versus the total amount of processing time [47]:

$$U = \sum_{i=1}^{n} \frac{e_i}{p_i}$$

3.2 Earliest Deadline First Algorithm

The Earliest Deadline First (EDF) algorithm is one of the best known algorithms for real-time processing. At every new ready status the processor executes the ready task with the earliest deadline [13; 17]. It gets access to the requested resource. At any arrival of a new task, EDF must be computed immediately heading to a new order – i.e. the running task must be preempted for this scheduling process. The new task is processed immediately if its deadline is earlier then the one of the interrupted task. The processing of the interrupted task is continued according to the EDF algorithm later on. EDF is not only an algorithm for periodic tasks but also for tasks with arbitrary requests and deadlines. Also, the service execution times of the tasks must not be known [13]. In this case no guarantee about the processing of any task can be given.

EDF is an ***optimal, dynamic*** algorithm. It produces a valid schedule whenever one exist. A dynamic algorithm schedules every incoming task according to its specific demands. Tasks of periodic processes have to be scheduled in each period. With n tasks which have arbitrary ready-times and deadlines the complexity is $\theta(n^2)$ [24].

Most of the available schedulers work with priorities. Each task is assigned a priority according to specific policy. The order of the tasks results from this priorities. The task with the highest priority is executed until it is finished or preempted by the request from a higher-priority task. After each time slice the scheduler may rearrange priorities (e.g. in OS/2 in the priority-class "regular"). The determination of the time slice has the goal to keep the number of context switches low (because the check and determination of priorities is also done by the CPU and it requires overhead processing) and to get a fair and valid schedule over the whole run-time of the system.

The EDF algorithm assigns priorities according to the deadlines of tasks if the scheduling is priority driven. The highest priority is assigned to the task with the earliest deadline, the lowest to the one with the furthest. With every arriving task, priorities have to be adjusted.

EDF is used by different models as basic algorithm. The *time-driven scheduler* (TDS) is based on a policy similar to EDF. It extents EDF and handles overload situations. If a overload situation occurs the scheduler aborts tasks which can not meet their deadlines any more and those which have a low value density. The value density corresponds to the importance of a task [80]. In our system we do not expect to have overload situations due to the use of pessimistic resource management schemes prior to scheduling.

In [50] an EDF scheduling algorithm is introduced which is also preemptive and priority-driven. Every task is divided in to a ***mandatory*** and an ***optional*** part. A task is terminated according to the deadline of the mandatory part even if it is not completed at this time. Tasks are scheduled with respect to the deadline of the mandatory parts. A set of task is said to be feasible scheduled if all tasks can meet the

deadlines of their mandatory parts. The optional parts are processed if the resource capacity is not fully utilized. Applying this to CM the method can be used with layered coding. Referring to uncompressed bitmaps, the processing of the MSB's (most significant bit) is mandatory whereas the processing of the LSB's (least significant bit) can be considered as optional. Applied to compressed images based on transformation into frequency domain, the most relevant information is part of the lower frequencies. Their processing is mandatory in contrast to the processing of the higher frequencies where the processing is optional. With this method more processes can be scheduled and in a overload situation no process has to be discarded.

For a dynamic algorithm like EDF the upper bound of the processor utilization is 100% [47]. Compared with any static priority assignment, EDF is optimal in a sense that if a set of tasks can be scheduled by any static priority assignment it can also be scheduled by EDF. In EDF there is no processor idle time prior to overflow.

Applying EDF for the scheduling of CM tasks on a single processor machine with priority scheduling priorities have to be rearranged when the priority required by a new task is currently used for another process. This may cause a considerable overhead. The EDF scheduling algorithm itself makes no use of the previously known occurrence of periodic tasks.

3.3 Rate Monotonic Algorithm

The Rate monotonic scheduling was first introduced by Liu and Layland in 1973 [47]. It is an optimal, static, priority-driven algorithm for preemptive, periodic jobs. Optimal here means that there is no other *static* algorithm that is able to schedule a task set which can not be scheduled by the rate monotonic algorithm. A process is scheduled by a static algorithm at the beginning of the processing. Subsequently, each task is processed with the priority calculated at the beginning. Five assumptions are made about the environment [47]:

1. The requests for all tasks with deadlines are periodic. I.e. with constant intervals between consecutive requests.
2. Deadlines consist of run-ability constraints only. I.e. each task must be completed before the next request occurs.
3. The request of tasks are independent. I.e. the requests for a certain task do not depend on the initiation or completion of requests for other tasks.
4. Run-time for each request of a task is constant. Run-time denotes the time which is required by a processor to execute the task without interruption.
5. Any non-periodic task in the system has no required deadline.

Further work shows that not all of these assumptions are mandatory for CM-data processing.

Static priorities are assigned to tasks once according to their request rates. The priority corresponds to the importance of a task relatively to other tasks. Tasks with higher request rates will have higher priorities [47]. The task with the shortest period gets the highest priority and the task with the longest period the lowest priority.

A task will always meet its deadline if it is proven for the longest ***response time***. The response time is the time span between the request and the end of processing of a task. This time span is maximal when all processes with a higher priority request processing at the same time. This case is called ***critical instant***. The ***critical time zone*** is the time interval between the critical instant and the completion of a task. An example is shown in Figure 2.

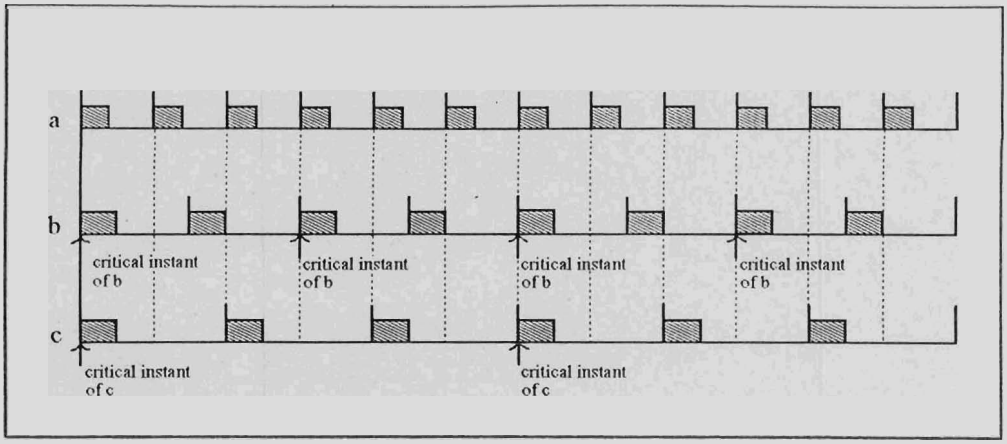

Figure 2. Critical Instant

Consider an audio and a video stream scheduled according to the rate monotonic algorithm. Let the audio stream have a rate of 1/75 and the video stream a rate of 1/25. The priority assigned to the audio stream is then higher then the priority assigned to the video stream. The arrival of a messages from the audio stream will interrupt the processing of the video stream. If it is possible to complete the processing of a video message before its deadline which requests processing at the critical instant, the processing of all video messages to their deadlines is ensured.

The processor utilization of the rate monotonic algorithm is upper bounded. It depends on the number of tasks which are scheduled, their processing times, and their periods. According to [47] there are two issues to consider:

1. The upper bound of the processor utilization which is determined by the critical instant.
2. For each number n of independent tasks $t(j)$ a constellation can be found where the maximum possible processor utilization is minimal. The least upper bound of the processor utilization is the minimum of all processor utilizations over all sets of tasks $t(j)$; $j \in (1, \ldots, n)$ that fully utilize the CPU. A task set fully utilizes the CPU when it is not possible to raise the processing time of one task without violating the schedule.

Under this assumptions [47] give an estimation of the maximal processor utilization where the processing of each task to its deadline is ***guaranteed*** for any constellation. A set of m independent, periodic tasks with fixed priority order will always meet its deadline if:

$$\frac{e_1}{p_1} + \cdots + \frac{e_m}{p_m} \leq m \times (2^{\frac{1}{m}} - 1) = U(m)$$

For large m the least upper bound of the processor utilization is $U = \ln 2$ [46].
Hence it is sufficient to check if the processor utilization is less or equal to the given upper bound to find out if a task set is schedulable or not.

With EDF, a processor utilization of 100% can be achieved because all task are scheduled dynamically according to their deadlines. Figure 3 shows an example where the CPU can be utilized to 100% with EDF but where rate monotonic scheduling fails.

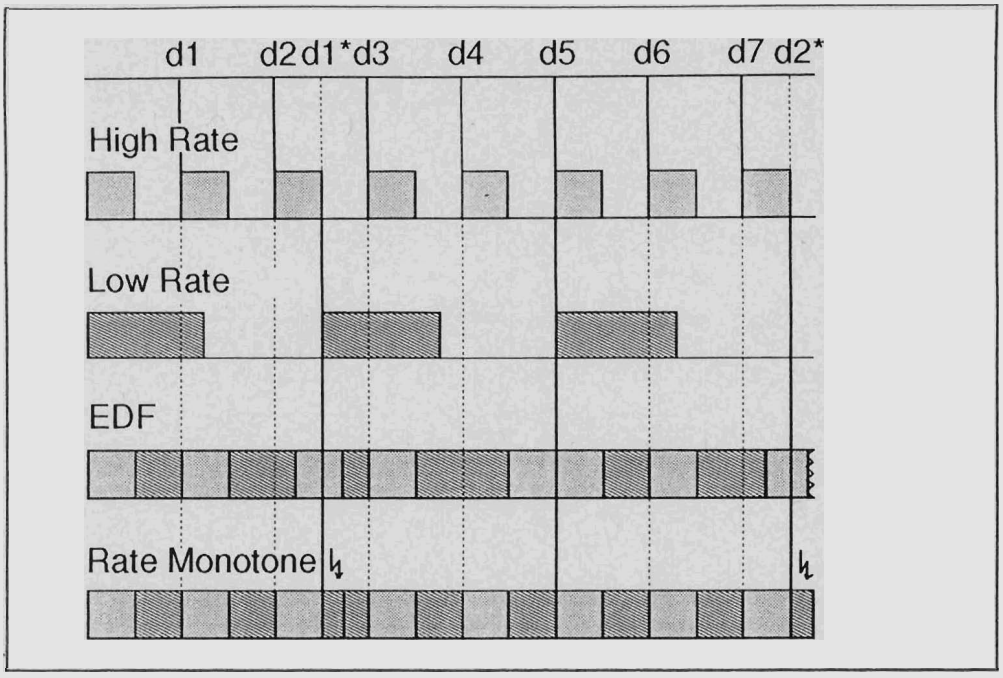

Figure 3. Rate Monotonic versus EDF: Processor Utilization

A related problem is addressed in [69]. In most cases the average task execution time is considerably lower than the worst case execution time. Therefore, scheduling algorithms should be able to handle transient processor overload. The rate monotonic algorithm on average ensures that all deadlines will be met even if the bottleneck utilization is well above 80%. With one deadline postponement, the deadlines on average are met when the utilization is over 90%. [71] mentions an utilization bound achieved for the Nowy's Insertial Navigation System of 88%. In the case of CM and DM-data to be processed, the utilization discussed so far only applies to CM. Even with a CM-utilization of 69%, the remaining 31% can be used for DM processing.

Since the rate monotonic algorithm is an optimal static algorithm no other static algorithm can achieve a higher processor utilization.

As shown in Figure 4 there might be more context switches with a scheduler using the rate monotonic algorithm then one using EDF.

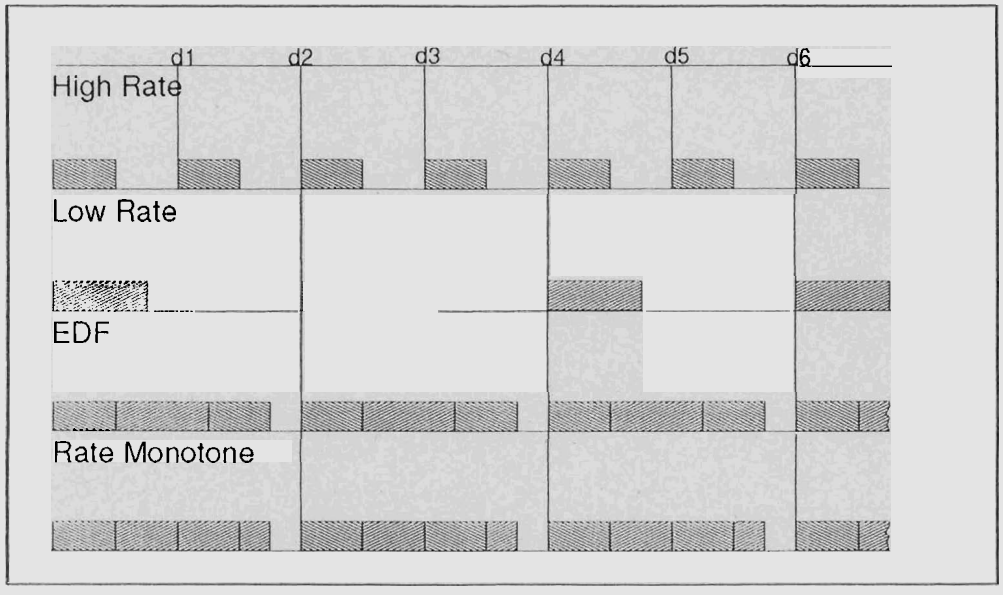

Figure 4. Rate Monotonic versus EDF: Context Switches

There are several extensions to this algorithm. One of them divides a task in to a mandatory and an optional part. The processing of the mandatory part delivers a result which can be accepted by the user. The optional part only refines the result (c.f. 3.2). The mandatory part is scheduled according to the rate monotonic algorithm. For the scheduling of the optional part different policies are suggested [9; 49; 10].

To meet the requirements of periodic tasks and the response time requirements of aperiodic requests, it must be possible to schedule both, aperiodic and periodic tasks. If the aperiodic request is an aperiodic continuous stream (e.g. video images as part of a dia-slide show), we have the possibility to transform it into a periodic stream. Every timed data item can be substituted by n items. The new items have the duration of the minimal life span. The number of streams is increased but since the life span is decreased the semantic remains unchanged. The stream is now periodical because every item has the same life span [23]. If the stream is not continuous we can apply a sporadic server to respond to aperiodic requests. The server gets a computation budget. This budget is refreshed t units of time after it has been exhausted. Earlier refreshing is also possible. The server is only allowed to preempt the execution of periodic tasks as long as the computation budget is not exhausted. Afterwards it can only continue the execution with a background priority. After refreshing the budget, the execution can resume at the servers assigned priority. The sporadic server is especially suitable for events that occur rarely but must be serviced quickly (e.g. a telepointer in a CSCW-application) [66; 71; 70].

The rate monotonic algorithm is applied in real-time systems and real-time operating systems by the NASA and the European Space Agency [67]. It is particularly suitable for CM tasks because it makes optimal use of their periodicity. Since it is a static algorithm there is nearly no rearrangement of priorities and hence no scheduling overhead to determine the next task with the highest priority. There are problems with data streams which have no continuous data rate (e.g. a compressed video stream where one of five pictures is a full picture and all others are up-dates of a reference picture). The solution is to schedule this tasks according to their maximum data rate. In this case the processor utilization is decreasing. The idle time of the CPU can be used to process DM tasks or other non-time-critical programs.

3.4 Other Approaches for Real-Time Scheduling

In these study phase we evaluated further scheduling algorithms toward their suitability for CM processing. In the following we describe briefly the approaches and enumerate the reason for their "non suitability". Compared with EDF and rate monotonic all of them have severe disadvantages for our problem.

Least Laxity First (LLF): The task with the shortest remaining laxity is scheduled first [11; 46]. The laxity is the time between the actual time t and the deadline minus the remaining processing time. The laxity in period k is:

$$l_k = (s + (k-1)p + d) - (t + e)$$

LLF is not only an *optimal, dynamic* algorithm for *exclusive resources* like EDF but also for *multiple resources* if their ready-times are the same [21]. The laxity is a function of deadline, processing-time and the current time. Since the processing-time is not known, worst-case is assumed. Therefore, the determination of the laxity is inexact. The laxity of the waiting processes is dynamically changing over time. During run-time of a task, another task may get a lower laxity. This task has then to preempt the running task. Consequently, tasks can preempt each other several times without dispatching a new task. Hence, there may be more context switches than with EDF. At each scheduling point, the laxity of each task has to be newly determined. This leads to an additional overhead compared with EDF. Since we

have only a single resource to schedule there is no advantage in the employment of LLF compared with EDF. It is not more accurate then EDF but has an higher scheduling overhead. Therefore, we do not consider this a suitable scheduling algorithm for CM-tasks.

Deadline Monotone Algorithm: If the deadline of tasks are shorter then their rate ($d_i < p_i$) the rate monotonic algorithm can not be employed. In this case a fixed priority assignment according to deadlines of tasks is optimal. A task T_i gets a higher priority as a task T_j if $d_i < d_j$. No effective schedulability test for the deadline monotone algorithm exists. In order to determine the schedulability of a task set, it has to be checked for each task if it can meet its deadline when it requires execution to its critical instant [44]. Tasks with a deadline shorter then the rate arise at the measurement of temperature or pressure in a control system. Their data rate is low. But, if there are any difference between nominal and real value the intervention has to be done quickly. In HeiTS we assume that the deadlines of CM-tasks are equal to their rate. In this case, the schedule determined according to the deadline monotone algorithm is the same as the one according to the rate monotonic algorithm.

Shortest Job First (SJF): The task with the shortest remaining computation time is chosen for execution [11; 17]. This algorithm guarantees that as many tasks as possible meet their deadlines under an overload situation, if all of them have the same deadline. Since we encounter in general equal deadlines and do not have overload situation (because of the pessimistic resource management). The SJF is not a suitable algorithm for the scheduling of CM-tasks.

First Come First Serve (FCFS): The task which arrives first is executed first. This method does not consider deadlines, processing times, arrival times or logical arrival times. FCFS is a non-preemptive scheduling strategy which should be applied if there is no other knowledge about the task apart of the fact that it is critical. This strategy does not have any process management overhead. There is a demand for a deterministic and predictable behavior of CM-tasks. With FCFS no guarantee for the processing of any task according to their deadlines can be given. Therefore, it is an insufficient method for the scheduling of CM-tasks.

Real-Time Monitoring: In a monitoring system, data on task activities of the computer system are extracted, processed and presented. Incorrect decisions on the schedulability of the tasks can be avoided because the monitoring process gets all necessary data about their behavior (e.g. performance bottlenecks) [20]. The resource management then has all information to initialize a correct schedule and to handle fault-handling. Unpredictable events can be handled easily. If there is any possibility to schedule a task, it is scheduled. Idle times of resources are minimized. Real-time monitoring is an exact dynamic scheduling method. It requires special software and hardware support. In most of the existing operating systems such an extensive monitoring is not possible. The processing of all the data on the activities of the computer system may lead to a large overhead in the resource management. Our system is designed to support the processing of CM-tasks in a conventional environment with already existing operating systems. Real-time monitoring can not be done in any of the systems employed for the HeiTS project.

MARS: Magnet II Real-Time Scheduling Algorithm: This scheduling algorithm was developed for asynchronous time sharing based switching nodes. There are three classes of data. Class I and II represent CM-data like audio and video, Class III is DM-data. The time is divided into periods called cycles, each consisting of up to II cells. Each cycle is further divided into subcycles. The maximum length of the cycle is defined by the parameters M_i, M_{ii}, M_{iii} with $M_i + M_{ii} + M_{iii} \leq II$. The scheduler first chooses the parameter II which is kept constant. There are two schedules maintained by the scheduler that contain the number of Class I and Class II cells that need to be processed during each future cycle on a finite horizon. At the end of each

cycle the schedules are updated by taking into account the number of new cells that got ready during the previous cycle. The minimum amount of resources that satisfy the Class I and Class II QoS requirements is allocated to each class, the QoS requirements of Class I must always be met. If the remaining resources are not sufficient for Class II tasks, the exceeding Class II cells are clipped. If there is resource capacity left it is allocated to Class III tasks. The decision is always made at the end of cycle times [28]. The Class I cells are guaranteed.

The MARS-algorithm was designed to schedule real-time traffic on a network. The resource is the network, the scheduler is integrated in the packet switch. It runs on a own CPU and does not have to consider the generated scheduling overhead. In our case the scheduler runs on the resource it has to schedule. It has to minimize scheduling overhead. Therefore, the MARS-algorithm is too complex for the CM-scheduling in our envisaged environment.

Search Heuristics for Scheduling: The problem of finding a feasible schedule can be conceived as a search problem. The normal search-algorithms can be employed to solve this problem. In [75] the *guarantee algorithm* is introduced. This algorithm uses a search tree to find a feasible schedule. The root of the search tree is the empty schedule. An intermediate vertex of the search tree is a partial schedule and a leaf is a complete schedule. Not all complete schedules are feasible schedules. The problem is to find a feasible schedule. A heuristic function H was developed. On each level of the search the function H is applied to find the task with the minimum value of H. This task is selected to extend the current, partial schedule. The complexity of this search is not exponential.

An algorithm based on the **network flow technique** is developed in [68]. This algorithm divides a task in a mandatory and in an optional part. With the network flow algorithm a schedule where all mandatory tasks and as many optional tasks as possible can meet their deadlines is determined. The optimal schedule is the one with the maximum flow in the network. The complexity to find an optimal feasible schedule is $\theta(n^3 \log n)$. The disadvantage of the network flow technique and the search heuristic is their complexity. Those algorithms can be applied if the schedule has to be determined only once and must not be altered during run-time. Our system runs in a dynamic environment. At run-time often new connection may be established or released. Every time a new schedule has to be determined. Therefore, no search algorithms or methods based on network flow techniques are not appropriate to schedule CM-tasks.

All of the described methods and algorithms may be applied for the solution of our problem. Some of them are general algorithms, some are algorithms for special problems. Various other methods and algorithms to schedule real-time tasks are described in literature. E.g., an on-line scheduler for tasks with unknown ready times [29]. In [7] a technique is introduced which is based on the network-flow model for uniform processors. In [86] the Virtual Clock, Fair Queueing, Delay Earliest Due Data, Stop and Go and Hierarchical Round Robin are described. Those are methods for the queueing in a packet switched data network which also could be used with some variations for the scheduling of real-time tasks on the CPU. Most of these approaches are variations of the algorithms described above, some use methods (e.g. round robin) that can not be considered as a real-time scheduling strategy at all [21].

3.5 Preemptive versus Non-Preemptive Task Scheduling

Real-time tasks can be preemptive and non-preemptive. If a task is non-preemptive it is processed and not interrupted until it is finished or requires further resources. If tasks are preemptive, the processing of any task is interrupted immediately by a request for any higher priority task [21].

In most cases where algorithms are treated as non-preemptive, the arrival times, processing times and deadlines are arbitrary and unknown to the scheduler till the task actually arrives. The best algorithm is the one which maximizes the number of completed tasks. It is not possible to give any processing guarantees or do resource management [85]. This methods are used in schedulers for hard real-time tasks with unpredictable occurrence of tasks.

To guarantee the processing of periodic processes and to get a feasible schedule for a periodic task set, tasks are usually treated as preemptive. One reason is, that high preemptability minimizes priority inversion [53]. Another reason is that for some non-preemptive task sets no feasible scheduled can be found, whereas preemptive scheduling is possible. Figure 5 shows an example where the scheduling of preemptive tasks is possible but non-preemptive tasks can not be scheduled.

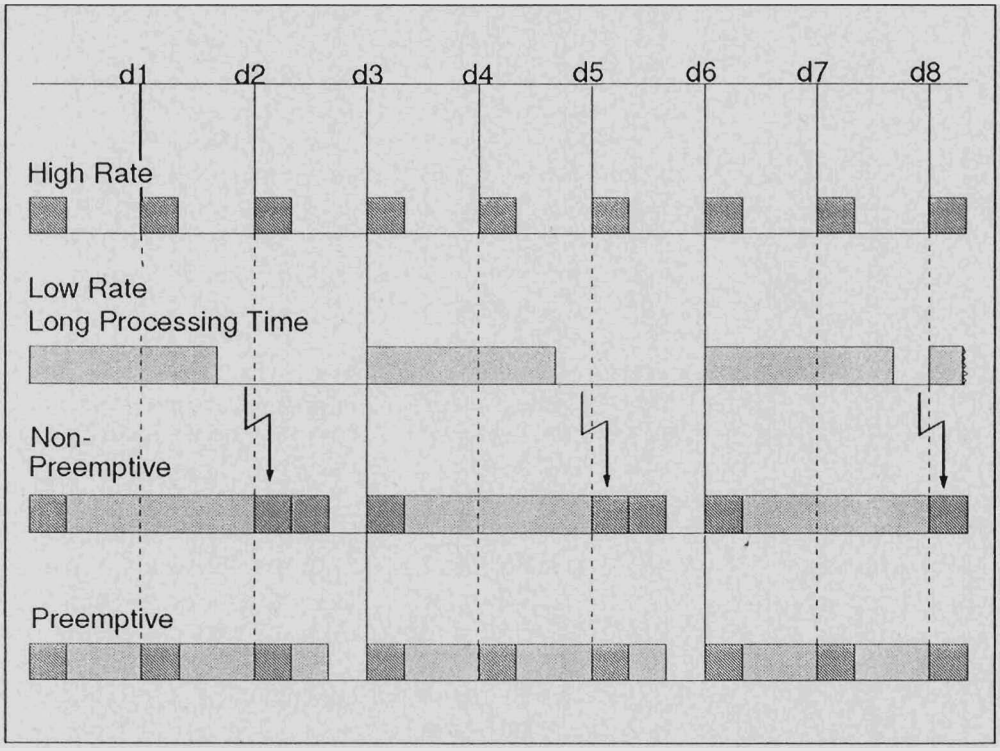

Figure 5. Preemptive versus Non-Preemptive Scheduling

In this case, tasks with high rates and tasks with low rates and long processing times are running concurrently in the same system.

In [47] Liu and Layland show that a task set of m periodic, *preemptive* tasks with processing times e_i and request periods $p_i \forall i \in (1, ..., m)$ is schedulable

- with fixed priority assignment if:

$$\sum \frac{e_i}{p_i} \leq \ln 2$$

- and for deadline driven scheduling if

$$\sum \frac{e_i}{p_i} \leq 1$$

Here, all tasks in the task set have to be preemptive to check their schedulability

The first schedulability test for the scheduling of *non-preemptive* tasks was introduced by Nagarajan and Vogt in [58]. Assume, without loss of generality, that stream M has highest priority and stream 1 lowest. They proof that a set of m peri-

odic streams with periods p_i, deadlines d_i, processing times e_i and $d_i \leq p_i \forall i(1, ..., m)$ is schedulable with the non-preemptive fixed priority scheme if

$$d_m \geq e_m + \max_{(1 \leq i \leq m)} e_i,$$

$$d_i \geq e_i + \max_{(1 \leq j \leq m)} e_j + \sum_{j=i+1}^{m} e_j F(d_i - e_j, T_j)$$

$$d_1 \geq e_1 + \max_{(1 \leq j \leq m)} e_j + \sum_{j=2}^{m} e_j F(d_1 - e_j, T_j)$$

where $F(x,y) = ceil(\frac{x}{y}) + 1$.

This means that the time between the logical arrival time and the deadline of a task t_i has to be larger, or equal to the sum of the own processing time and the processing time of any higher priority task that requires execution during that time interval plus the longest processing time of all lower priority tasks that might be serviced at the arrival of t_i.

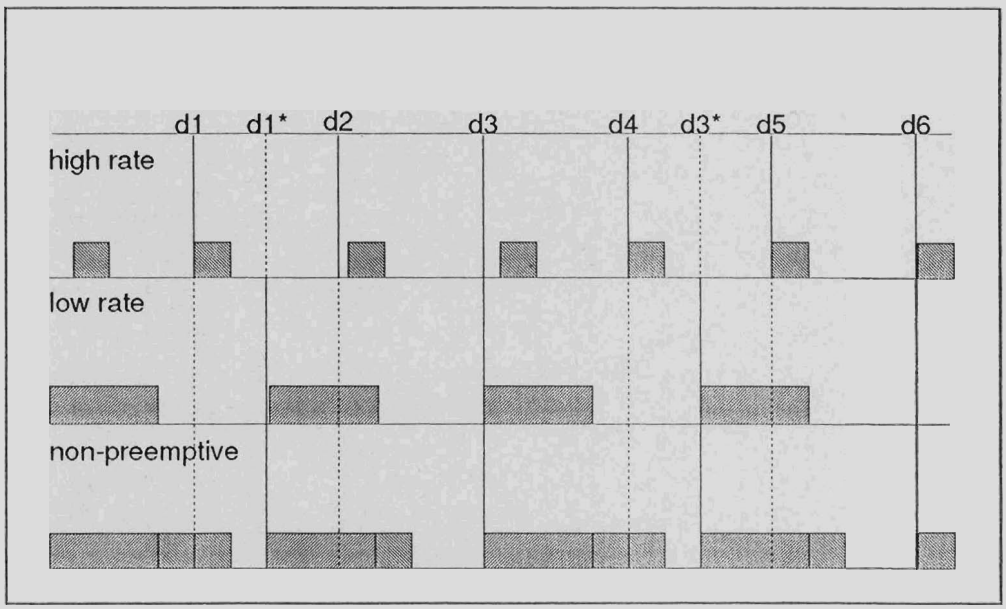

Figure 6. Deadline Requirements for Non-Preemptive Scheduling

The schedulability test is an extension of Liu's and Layland's. Given m periodic streams with periods p_i and unit processing times E per message. Let $d_i = p_i + E$ be the deadline for stream i. Then the streams are schedulable

- with the non-preemptive rate monotonic scheme with:

$$\sum \frac{1}{p_i} \times E \leq ln2$$

- with deadline-based scheduling, the same holds with:

$$\sum \frac{1}{p_i} \times E \leq 1$$

3.6 Scheduling of CM-Tasks: Prototype Works

The above described algorithms to schedule real-time tasks are partly, and with modifications, applied in existing implementations especially designed to schedule real-time multimedia tasks and in implementations which could easily be applied to multimedia. Generally it can be distinguished between two kind of systems:

- **Real-time operating systems:** This are operating systems that are especially designed to handle real-time tasks [60].
- **Meta-scheduler:** Scheduler using existing real-time capabilities of operating systems to handle CM-tasks.

3.6.1 ARTS: A Distributed Real-Time Kernel.

ARTS is a real-time operating system for a distributed environment. It was developed on SUN3 workstations connected by a real-time network based on the IEEE.802.5 Token Ring and Ethernet by the computer science department of the Carnegie Mellon University. In ARTS, a object based programming model was developed. A object in this model is composed of data, one or more threads of execution, and a set of export operations. One or more threads will be assigned to each of the operation exported by the object. Additional threads which are not associated with operations may also be present for doing internal computation [52].

The ARTS kernel consists of the kernel objects which provides the mechanism of the ARTS operating system.

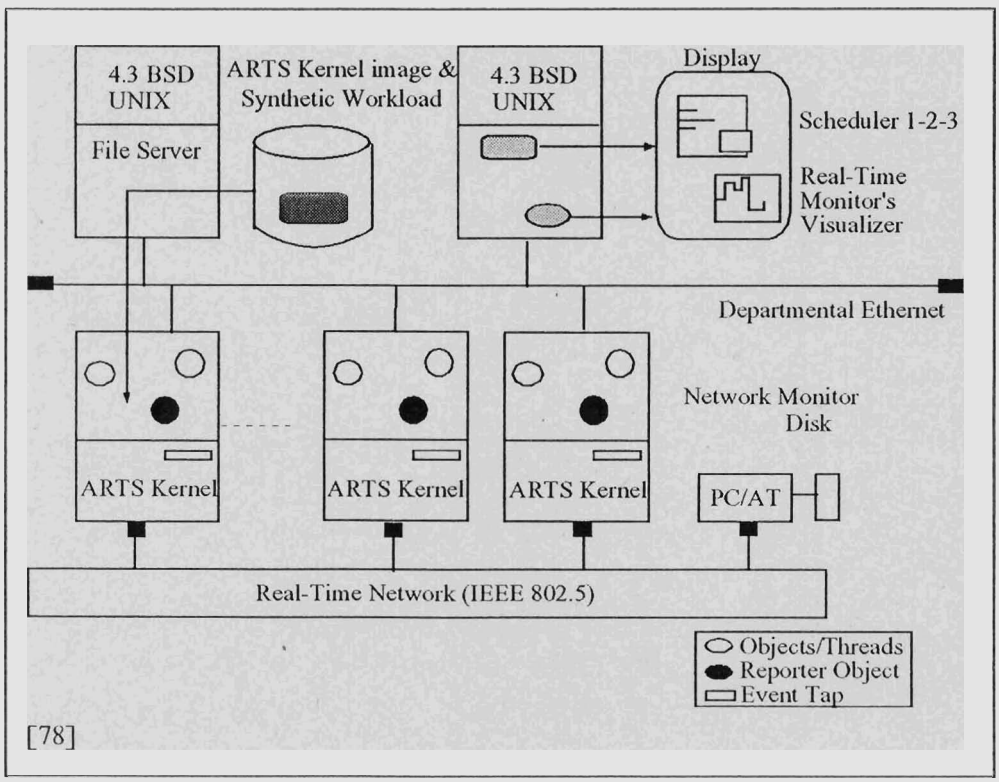

Figure 7. ARTS System Structure

To solve the scheduling problems the *time-driven scheduler* (TDS) with a priority inheritance protocol was adopted. This priority inversion protocol was used to prevent unbounded priority inversion among communication tasks. In particular, almost every waiting queue discipline was replaced by a priority-based discipline with the priority inheritance protocol in the kernel.

The *integrated time-driven scheduling* (ITDS) model is applied which provides predictability, flexibility, and ease of modification for hard and soft real-time activities in various real-time applications. It allows to predict whether the given task of hard real-time activities can meet their deadlines or not. The processor cycles are divided into hard and soft real-time tasks. First, the processor utilization of the hard periodic and sporadic activities are determined and the rate monotonic algorithm is applied, then the remaining processor time is assign to soft aperiodic activities. It allows also to check the schedule for more general task sets which accesses shared resources. As long as there exists a schedulability test, the ITDS can adopt other scheduling policies like EDF.

The integrated time-driven scheduler can schedule the tasks based on their deadlines as well as to the task criticality in the case of transient overload (c.f. 3.2 & [80]). The scheduling policy is separated from the scheduling mechanism layer. The scheduling policy was implemented as a self-contained kernel object, and the mechanism layer performs dispatching and blocking of the threads.

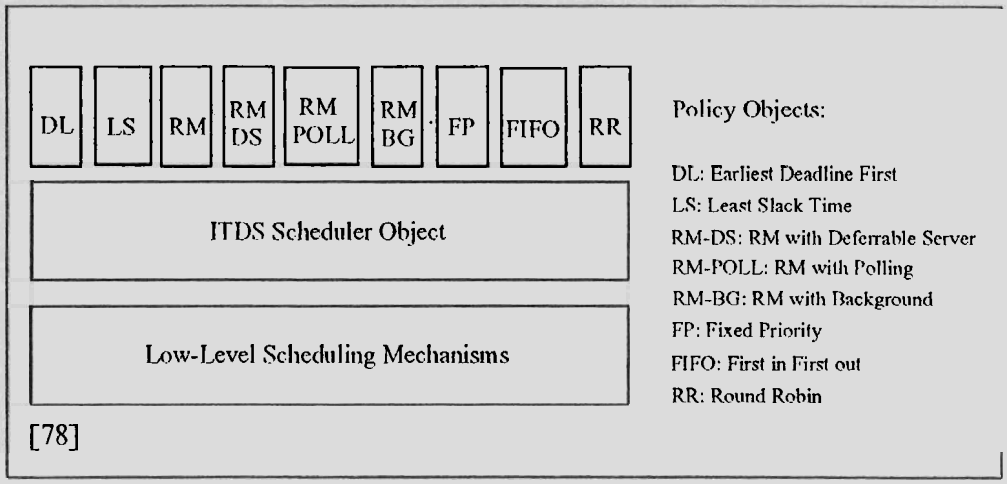

Figure 8. The Structure of the ITDS Scheduler

Static scheduling policies such as rate monotonic are implemented as well as dynamic methods like EDF or least laxity. For comparison with real-time scheduling methods common scheduling algorithms like FIFO, round robin and fixed priority were also implemented. The ITDS scheduler can guarantee schedulability of hard periodic tasks, value function based soft real-time task scheduling, and overload control based on the value functions of the aperiodic tasks.

The ARTS kernel provides a tool set for predicting the behavior of the system and for run-time monitoring. The schedulability analyzer —called Scheduler 1-2-3— is a X11-window based interactive schedulability analyzer for creating, manipulating, and analyzing sets of real-time tasks. It can be used to predict the timing effects due to the software and hardware modification and together with other tools —such as the timing tool and the real-time monitor debugger— as a synthetic workload generator. The objectives are:

- **Schedulability analysis:** Verification of the schedulability of any given hard deadline task set under scheduling algorithms like EDF, rate monotonic etc.
- **Response time analysis for aperiodic tasks:** The performance of soft, aperiodic tasks can be computed.
- **Convenient interface:** Interface through which the user can perform the schedulability analysis.
- **Synthetic workload generator:** Workload table.

The ARM (Advanced Real-Time Monitor) is a tool to analyze and visualize the run-time behavior of the target nodes in real-time. The objectives are:

- **Visualization:** It visualizes the system activity at an arbitrary level of abstraction.
- **Monitorability analysis:** The performance degradation due to the monitoring/debugging features can be minimized and estimated beforehand.
- **Remote debugging**

The scheduling policies of ARTS can easily be changed or other ones can be added by the user because the policies are implemented as kernel objects [78].

3.6.2 YARTOS: Yet Another Real-Time Operating System

YARTOS was developed at the University of North Carolina at Chapel Hill as an operating system kernel to support conferencing applications that uses digital audio and video, and supports a 3-dimensional graphics display system that is used for research in virtual reality. It is a message passing system with a semantic of inter-process communication that specifies the real-time response that an operating system must provide to a message receiver [37]. YARTOS runs currently on IBM PS/2 workstations (Intel 80386 processor) interconnected with a 16Mbit Token Ring network to support digital audio and video. IBM-Intel Action Media 750 adapters are used for the acquisition, compression, decompression, and display of the digital audio and video [38].

Programs which are executed under YARTOS are compiled into a set of sporadic tasks that share a set of serially reusable, single-unit resources. A sporadic task is a sequential program that is invoked in response to the occurrence of an event. An event is a stimulus that may be generated by processes external to the system (e.g. an interrupt from a device), or by processes internal to the system (e.g. the arrival of a message). Software objects like abstract data types that are shared by multiple tasks represent a resource in YARTOS. For a given workload, YARTOS has two goals:

- To guarantee that all requests from all tasks will complete execution before their deadlines.
- To guarantee that no shared resource is accessed simultaneously by more than one task.

Therefore, an optimal, preemptive algorithm for sequencing of such tasks on a single processor was developed [35; 36]. It is optimal in the sense that it can provide the two guarantees whenever there is any possibility to do so. An efficient schedulability test for a given task set was also developed.

The scheduling model is composed of tasks and resources. The workload consist of a set of n sporadic tasks $(T_1,..., T_n)$ and a set of m serially reusable, single unit resources $(M_1,..., M_m)$. A task is described by a pair $T = (E, R)$, where E is the maximum amount of processor time required to execute the program called the computational costs, and R is a response time parameter derived from the rate. When n_i are the number of operations on shared resources performed by an invocation of task T_i with $e_1,..., e_{ni}$ as the maximum execution time required for each operation and e_{i0} the maximum execution time for the remaining code then $E_i = e_{io} + e_1 + ... + e_{ni}$.

The algorithm to schedule the task results from the integration of a synchronization scheme for access to shared resources with the EDF algorithm [47]. A task has two notions of deadline, one for the initial acquisition of the processor, and one for execution of operations on resources. To avoid priority inversion tasks are provided with separate deadlines for performing operations on shared resources. At the invocation of a task at time t it has, as in traditional EDF scheduling, the deadline $d = t + R$. This is the deadline for the task to complete execution. If a task starts an

operation on a shared resource at time t_i then at this time its deadline will be equal to $\min(t + R, t_i + 1 + R_{\min})$; $R_{\min}$ is the smallest response time of all tasks which can access the resources. Therefore, a task which is invoked at t_i and wishes to perform an operation on the same resource will not preempt the other task, because its deadline is necessarily greater than $t_i + R_{\min}$. This method ensures mutual exclusion on resource operations. It is optimal in the sense that it can schedule a set of tasks, without inserted idle time, whenever it is possible.

There is an efficient schedulability test for the algorithm. The first requirement for a feasible schedule is like given in [47]:

$$\sum \frac{E_i}{R_i} \leq 1$$

The second demand is:

$$\forall i, 1 \leq i \leq n; \forall k, 1 \leq k \leq n_i; \forall L, R_{min,k} \leq L \leq R_i:$$

$$L \geq e_{ik} - 1 + \sum_{j=1}^{i-1} f(\frac{L}{R_j}) E_j$$

$$where\ f(x) = largest\ integer \leq x$$

Here, n is the number of tasks, n_i denotes the number of operations on shared resources performed by an invocation of task T_i, and $R_{min,k}$ is the smallest response time requirement of the tasks which are accessing the resource M_k. This condition applies only to task that requires access to resources, and quantifies the processor demand that occurs when tasks simultaneously try to access a shared resource [37].

To ensure that all computational activities are dispatched by the scheduler, traditional non-dispatched activities like interrupt handler are implemented as tasks. They are scheduled in the same manner as user tasks. The reason for this was the demand to ensure that tasks with near deadlines do not fail. Here, a traditional interrupt handler is a task that is created by the user and invoked by a hardware signal. The deadlines of these tasks are based on the expected inter-arrival time of the interrupt. Although this information may not be reliable it turned out that it is not a problem for the YARTOS applications.

According to its designer YARTOS is a useful vehicle for real-time applications that are primarily concerned with processing of long-living, uniform data-streams in particularly CM-applications [37].

3.6.3 Split-Level Scheduling for CM

The split level scheduler was developed within the DASH-project at the University of California at Berkeley. Its main goal is to provide a better support for CM applications. It was developed to prevent CM application from timing errors and lost data due to the overhead of user/kernel interaction such as CPU scheduling and I/O, or any concurrent system activity. A typical application is the ACME (Abstractions for CM) I/O server which supports applications such as audio/video conferencing, editing, and browsing. The supported physical devices are speakers, microphones, video displays, and video cameras [5]. It allows to create logical devices which are associated with physical I/O devices, and do I/O of CM over CM-connections. For each of the CM-connections a network I/O process exists which transfers data between an internal buffer and the network. Each CM I/O device is associated to a device I/O process. For non real-time events such as commands from the window server and request for CM-connection establishment there are event-handling processes. It is implemented on Sun SPARCstations. It is

written in C++ and uses a preemptive lightweight process library. I/O is done using UNIX asynchronous I/O [18].

The applied scheduling policy is deadline/workahead scheduling. The LBAP-model introduced in [3] is used to describe the arrival processes. Critical processes have priority over all other processes and they are scheduled according to the EDF algorithm preemptively. Interactive processes have priority over workahead processes as long as they not become critical. The scheduling policy for workahead processes is unspecified but may be chosen to minimize context switching. For non real-time processes a scheduling strategy like UNIX time-slicing is chosen.

The CM applications are multiple processes that are sharing a virtual address space (VAS). The so called "split-level scheduler" uses lightweight processes (LWPs). They have the advantage, that user/kernel interactions are minimized, so that context switches within a VAS are fast. Figure 9 shows the structure of the split-level scheduler.

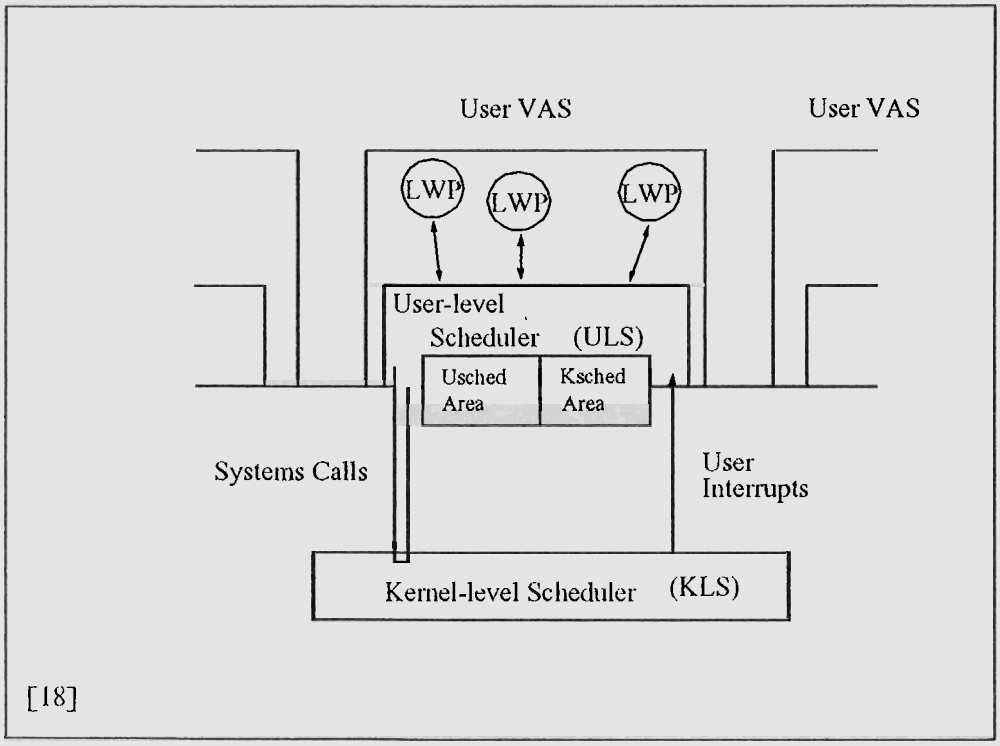

Figure 9. User-Level and Kernel-Level Parts of the Split-Level Scheduler

There is one kernel process and multiple LWPs per VAS. A LWP sleeps or changes the priority by calling its user level scheduler (ULS). The ULS checks whether its VAS still contains the globally highest-priority LWP. This is done by examining an area of memory that is shared with the kernel. If the highest-priority LWP is in the own VAS the LWP context switch is done without kernel intervention, otherwise, a kernel trap is done. The kernel-level scheduler (KLS) decides then according to the information in the shared memory segments which VAS should now be executed.

According to the designer malicious or incorrect programs may keep VAS preemption masked indefinitely, or it may execute indefinitely without changing its deadline. To prevent the other processes from starvation due to this behavior they propose the implementation of a watchdog timer. This watchdog is used to detect such conditions, and to kill or demote the offending process.

Split-level scheduling is a effective scheduling method. Compared with the performance of the normal UNIX scheduling mechanism it is better because it reduces the number of user/kernel interactions [18].

3.6.4 The HeiTS-AIX Approach

The HeiTS multimedia communication system is designed to run on different platforms. Apart from OS/2 running on PS/2, we have IBM RISC System/6000 computer running AIX version 3.1. Both have the task to process CM. The OS/2 and the AIX approach are both based on the same resource model (LBAP-model), have the same QoS-parameters and the same requirements on scheduling [8].

Each connection is associated with an own system process. The communication protocols up to layer 4 are processed in this system process. The communication of the different layers is done by up-calls and down-calls which are implemented as function calls.

AIX, like UNIX, has a user and a kernel space. Interrupts are processed in the kernel. Processes can run in the kernel or in the user space. The scheduling in AIX is priority driven. Time-critical tasks can be processed with 16 different priorities [33]. Processes are preemptive. The kernel can be extended by additional device drivers, kernel processes or system calls. Program components can be programmed as system calls to process them in the kernel. Processes in the kernel can not be interrupted by signals. Kernel processes can only use a restricted set of system calls.

Normal AIX processes are used for the processing of the CM-data. A data stream is associated with one process. This process is used only for one connection, it serves every incoming message from this connection. Since the messages can arrive in bursts, enough buffer has to be provided for each connection. Each message is inserted in a queue that is assigned to the process. After the processing of a message the next message is taken from the queue.

The scheduler is implemented as a set of functions that are called during the interrupt processing or by the application programs. The scheduler determines the priorities according to the rate monotonic algorithm. There are 13 priorities for guaranteed and best-effort processes (best-effort processes run with a lower priority than guaranteed processes), one priority for aperiodic processes (13), guaranteed workahead (14) and best-effort workahead (15) processes.

Every incoming message is indicated by an interrupt. The interrupt handler determines the connection the message belongs to. Subsequently, the interrupt handler stores the message in an allocated buffer and queues the message. Buffer is allocated for the number of messages indicated by the maximum burstiness. All necessary information about a connection is stored in the scheduling-cache. Only messages which obey the LBAP specification are accepted. The process takes a message from the queue and calls the scheduler. Figure 10 illustrates this processes and shows the structure of the AIX meta-scheduler.

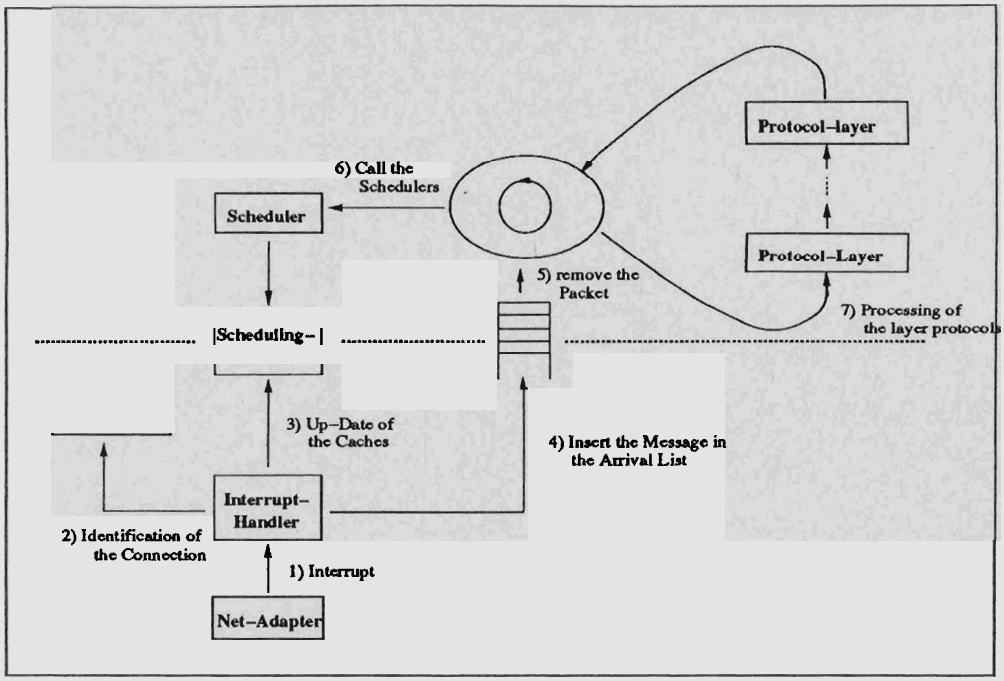

Figure 10. Structure of the AIX Meta-Scheduler

Messages which are ahead of schedule are not processed with a workahead priority. A process checks if a message arrived in time or ahead of schedule. If the message is ahead, the process starts a timer and blocks the processing till the logical arrival time is reached.

There is no mechanism to observe the run-time behavior of the processes without causing a considerable overhead. This is also the reason why there is no sporadic server for the processing of aperiodic tasks and why best-effort processes have to run with a lower priority compared to guaranteed processes. The processing time which is reserved for a best-effort process is the average processing time of this process. There is no reason to give a best-effort process a lower priority than a guaranteed process as long as it does not exceed the reserved processing time. Only when a process exceeds its processing time, it must run with a lower priority. Since it is not possible to measure the processing time, best-effort processes have to run with a lower priority then guaranteed processes.

Measurements of the system performance show that the overhead caused by the scheduling and context switches are not negligible. A context switch takes between $36\mu s$ and $48\mu s$. To start and stop the timer that indicates the logical arrival time of messages takes about $82\mu s$. To decrease the overhead caused by context switches it is proposed to build a non-preemptive scheduler.

4.0 Heidelberg Multimedia Operating System Support

The goal of the Heidelberg Multimedia Operating System Support (HeiMOS) is to provide the necessary real-time support needed by CM-applications. Figure 11 illustrates the position of HeiMOS within the HeiTS project. In this paper we focus on CM-scheduling as the core component of HeiMOS.

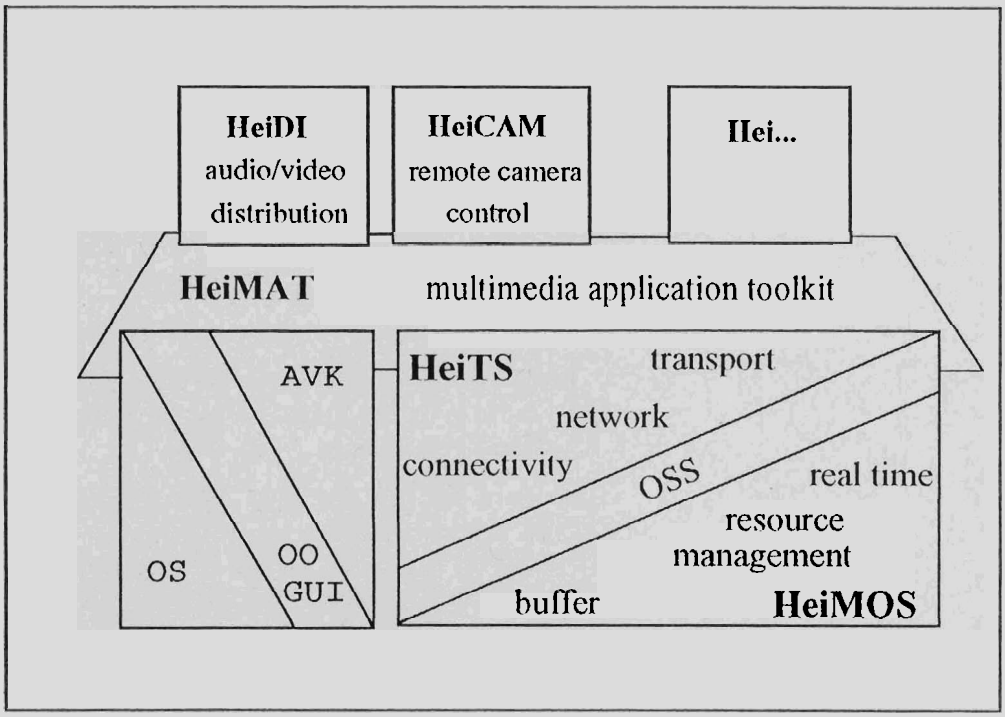

Figure 11. HeiMOS and its Relationship to HeiTS

The component ***HeiDI*** (Heidelberg audio-video distribution application) is a distributed audio-video application developed especially for HeiTS [55; 74]. Another application is ***HeiCAM*** (Heidelberg remote camera control) a remote camera control in a distributed environment [64]. The transport interface of HeiTS makes its services available to all applications. They are implemented as function calls. An object oriented interface to the communication system and other multimedia specific functions are provided for the applications by ***HeiMAT*** (Heidelberg Multimedia Application Toolkit).

The processing of time-critical data requires a careful allocation and manipulation of buffer space. To avoid overhead through copying data the standardized ***buffer management*** provides virtual copying. The ***operating system shield*** (OSS) is a standardized interface to all system extensions and in particular to the buffer management.

OS/2 provides no sufficient support for the processing of CM-tasks in real-time. HeiMOS is intended to provide this support. It is designed to ensure that all time-critical data are processed to meet their deadlines.

4.1 Hooks for Real-Time Processing in OS/2

HeiMOS is based on the operating system OS/2. In this section we give a brief overview on OS/2 and discusses its real-time capabilities. OS/2 is a multitasking system. Different tasks can run simultaneously either in the same program, or in different application programs. Each program runs in a virtual address space. The OS/2 dispatcher coordinates the programs so that they do not influence each other. The major change of the new version 2.0 is the step from 16-bit programming environ-

ment to 32-bit programming model that enables applications, sub systems, and the system itself to utilize the 32-bit register set, and the 32-bit instruction and addressing mode, as well as memory objects larger than 64KB [39].

The purpose of the OS/2 scheduler design is to optimize response rather than throughput. The system is not concerned about ensuring that all runable threads get at least some CPU-time, and the system is not primarily concerned about trying to keep the disk busy when the highest-priority thread is compute bounded. This policy and some other provisions with real-time capabilities makes OS/2 suitable for the design of time-critical applications on top of it.

4.1.1 OS/2 Process Management

OS/2 was designed as a time-sharing operating system without taking into account serious real-time applications. Let us start with a short description of the available process management, which was extracted from the available product information [45; 61; 32; 30]

In OS/2 three levels within a multitasking hierarchy exist:

- A ***session*** represents a logically separated unit of screen, keyboard, mouse and their related processes. Sessions can be arranged in parent and child sessions. Each session contain at least one process.
- A ***process*** is the logical unit of resources, including memory, files, and devices that are allocated to run a process. Like sessions, processes can create other processes leading also to a child − parent dependency. A process belongs to one, and only one session. Each process has one or more threads.
- The dispatchable unit of execution is called a ***thread.*** Each thread belongs to exactly one process. A thread shares the resources allocated by the respective process. Threads are not organized hierarchically. Each thread has its own execution stack, register values and dispatch state (either executing or waiting to execute).

Whenever a thread is created it belongs to a **priority class**. Four priority classes exist:

1. The ***time-critical*** class is reserved for threads that require immediate attention. Such threads will be used for communications and real-time applications.
2. The ***fixed-high*** class is intended for applications that require good responsiveness without being critical.
3. The ***regular*** class is used for the executing of normal tasks.
4. The ***idle-time*** class runs threads with a very low priority. Any Thread in this class is only dispatched if no thread of any other class is ready to execute.

Within each class 32 different **priorities** $(0, \ldots, 31)$ exist. Through **time-slicing** threads of equal priority have equal chances to execute. A context switch occurs whenever a thread issues a call to get access to an otherwise allocated resource. The thread with the highest priority is dispatched, the time-slice is started again. At the expiration of the time slice, OS/2 can preempt the dispatched thread if other threads of equal or higher priority are ready to execute. The time slice can be varied between 32 msec. and 65536 msec. (by setting the variable TIMESLICE in the file CONFIG.SYS). The default value is 250 msec.

Threads of the regular class may be subject of a dynamic rise of priority as a function of the waiting time. Whenever the variable PRIORITY is set to AUTOMATIC in CONFIG.SYS this mechanism is enabled. OS/2 boosts the priority of a thread which has waited longer than specified by the MAXWAIT variable.

By definition of the variable PRIORITY = FIXED this mechanism is prohibited, and regular threads behave as those of any other class.

The OS/2 scheduler is priority based and preemptive, i.e. if a higher-priority thread is ready to execute, the scheduler preempts the lower-priority thread and assigns the CPU to the higher-priority thread. The state of the preempted thread is recorded so that execution can be resumed later.

4.1.2 Provision of Real-Time Capabilities by Physical Device Drivers

OS/2 provides the possibility to use physical device drivers (PDD) that run at ring 0 for applications with real-time requirements. These PDDs can be made non interruptible. An interrupt that occurs on a device (e.g. arriving of packets) can be serviced from the PDD immediately. As soon as an interrupt happens on a device, the PDD gets control and can do all the work service that interrupt. This can also include tasks which could be done by application processes running in ring 3. The task running at ring 0 should leave the kernel mode after 4 msec. (called the "4 ms Rule").

In general, ring 0 applications are considered to service a request of time-critical tasks quicker then ring 3 applications because of their lower dispatch times.

The employment of a PDD has several disadvantages. Its implementation is more complicated then the implementation of a ring 3 application. The PDD is bounded to its device. It only services requests from its device regardless to any other events happening in the system. Different streams that request real-time scheduling can only be serviced by their PDDs. They run in competition with each other without the possibility to coordinate or manage them by any higher instant. This is insufficient for a multimedia system where messages can arrive at different adapter cards (e.g. DVI, FPC). It would be a reasonable solution for a system where streams arrive at only one device and no other activity in the system has to be considered.

4.1.3 Provision of Real-Time Capabilities by Time-Critical Threads

Time-critical tasks can also be processed together with normal application running in ring 3. The critical tasks can be serviced by threads running in the priority class time-critical with one of the 32 priorities within this class. The thread with the highest priority gets access to the CPU. All other threads are scheduled according to their priorities. A thread is interrupted if another thread with higher priority requires processing. Normal applications run as regular threads.

The main advantage of this approach is the control and coordination of all time-critical threads. One instance running with a higher priority then all other threads can perform resource management, observe their behavior, and determine a schedule according to specified policy for all time-critical tasks in the system. The task may involve different devices of the system. Their competition for the CPU is regulated and through the resource management and the scheduler, a guarantee for their processing within the required time bounds can be given. Internal time-critical tasks (e.g. stored audio or video from a disk) can also be considered.

The normal system scheduler is used to schedule all tasks. "Normal" applications do not have to be considered by the meta-scheduler. They will run during the time where no time-critical threads are ready for execution. The resource management should therefore not use the whole processor time for time-critical threads. We decided to use time-critical threads with the known limitations.

4.2 Scheduling Continuous Media in the HeiMOS Environment

Real-time scheduling in HeiMOS is done through a system process called meta-scheduler. To employ the introduced algorithms and to build an application on top of the operating system the tasks have to serve certain requirements. In this chapter we describe this requirements

4.2.1 Interaction with Resource Management

Periodic task such as the processing of CM data have regular interarrival times equal to their periods and deadlines that coincides with the end of their current periods [71]. Different CM-streams have different requirements concerning their deadlines. For instance, the processing of bitmaps is more tolerant to deadline failures then the processing of compressed video. To meet the deadline requirements of all CM data types, we consider all deadline to be hard.

In order to build a feasible schedule, we have to know the rate and the processing time. From the rate we derive the logical arrival time and the deadline of a message according to its order number. At connection establishment, the processing time is needed by the resource management to find out if it is possible to build a feasible schedule with the new task.

With every new connection the resource management has to perform a schedulability test. It has to check if it is possible to guarantee the required amount of processing time within the given delay bound in every period. The efficiency of a schedulability test is a major evaluation criterion for a scheduling algorithm. To avoid unacceptable delays during the connection establishment and to keep the CPU-time required by the schedulability test low, it should be simple and easy to perform.

The processing of a CM-task starts with the arrival of the message at the network interface and includes network hardware interrupt handling, session identification, protocol and user level processing. According to [1] there are five processing steps.

1. Packet arrival in the network interface device
2. Hardware interrupt to the CPU
3. Session identification
4. Protocol processing
5. User level processing

The end of the session identification is the first moment where all necessary data for the scheduling of the message like connection, rate, and processing time is known. From this moment on a message can be scheduled according to a specific policy. With a preemptive scheduling scheme a message is processed from its arrival to the session identification with the highest priority. The currently processed message is subject to priority inversion when the newly arrived message is belonging to a low priority task. The resource management has to consider some laxity.

4.2.2 CM Scheduling: Goals

The main goal of our CM real-time scheduling is to schedule the resources (e.g. CPU) that can potentially become bottlenecks in a way that allows reservation (associated with performance guarantees) to be made to individual clients [2]. The problem is to find a feasible schedule which schedules all time-critical CM-tasks in a way that each of them can meet their deadlines. This must be guaranteed for all tasks in every period over the whole run time of the system.

Two conflicting goals have to be considered:

1. In a multimedia system time-critical CM-tasks and non-critical DM-processes will run concurrently. An uncritical process should not suffer from starvation because time-critical processes are executed [73]. A multimedia application relies as much on text and graphics as on audio and video. Therefore, not all resources should be occupied by the time-critical processes and their management processes.
2. On the other hand a time-critical process must never be subject to priority inversion. This means that it should not kept from running by non-critical, or lower priority processes for infinite time. The scheduler has to ensure that any priority inversion is reduced as far as possible [54].

Apart of the overhead caused by the schedulability test and the connection establishment, we have to consider the costs for the scheduling of every message. They are more critical because they occur periodically with every message during the processing of real-time tasks. The overhead generated by the scheduling and the operating system has to be added to the processing time of the real-time tasks. Therefore, it is favorable to keep them low. Particularly difficult is to observe the timing behavior of the operating system and its influence on the scheduling and the processing of time-critical data. It can lead to time garbling of the application programs. Therefore, operating systems in real-time systems can not be viewed detached from the application programs and vice-a-versa [59].

4.2.3 CM Scheduling: Issues to be Considered

At the connection establishment the message rate is indicated. Through the burst parameter a short time violation of the rate is possible. With a static priority algorithm a high priority thread would process a message that is ahead of schedule at the expense of lower priority tasks. To avoid this, a rate control mechanism has to be included that assigns early messages a lower priority then critical messages or delays their processing until their logical arrival time has elapsed.

The second parameter that is indicated by the connection at the connection establishment is the processing time. A task that permanently exceeds its guaranteed processing time violates the calculated schedule. With preemptive tasks only processes with a lower priority then the offending process are affected. All processes are affected if the tasks are non-preemptive. Therefore, the CPU-time needed by single tasks for processing has to be controlled. Neither in AIX nor in OS/2 the pure CPU-time can be measured. The measurement of the processing time always includes interrupts and other delays.

A problem which should not be underestimated is the overhead caused by the scheduling itself, the controlling of processes, the setting and changing of priorities. With a dynamic algorithm a priority driven scheduler might have to change the priorities of all processes at the arrival of a new message. The resource management and the scheduler have to be considered as overhead. This can either be done by adding the processing time needed for the scheduling to the processing time of each task, or by a special process that has to be included in the schedule.

4.3 HeiMOS OS/2 Approach

As a result of our investigations on traditional real-time scheduling algorithms and already implemented prototypes, we developed two methods for the scheduling of CM. The methods are designed for their implementation on top of the OS/2 operating system [51]. We assume that tasks arrive according to the LBAP model.

In the end-systems, messages from different connection are processed. Each connection is associated with a single thread running in the priority class time-critical. Each of this threads are associated with a own message queue. All messages from a connection are processed within this thread up to the transport layer. The communication between the layers is realized as up- and down-calls. The application programs run on top of the transport layer. Apart of the threads for the different connections, there are special threads to perform the connection establishment and to control the application threads. Every incoming message triggers a hardware interrupt. The *second level interrupt handler* (SLIH) generates then a software interrupt. It runs with the highest priority within the priority class time-critical. From this point on the scheduler has control over the message and is able to schedule it according to a specific policy. The number of threads is restricted by the number of different time-critical priorities. The upper bound of connections with different priorities is 27. Priorities 3, ..., 29 are called ***critical priorities***. We do not distinguish different priority classes for guaranteed and best-effort processes. If at the connection establishment a best-effort process is indicated by the resource manager, a critical priority is assigned to that processes. The priority is lower then the priorities of all guaranteed processes. Priority 1 is used by workahead processes. Priority 2 is provided for threads which exceeds their specified processing time. This priority is called ***penalty priority***. If more than 27 connections are necessary a ***constant ratio grid*** could be used for the priority assignment [67]. We consider the number of priorities as sufficient. Therefore, the implementation of a constant ratio grid is not necessary.

In both methods we employ a control mechanism to monitor the behavior of the CM-tasks. To guarantee the processing of tasks the scheduler must have the possibility to monitor their behavior and to ensure that they do not violate the data constraint [4]. This includes a mechanism that is able to observe and react on offending behavior.

4.3.1 Queue Monitoring

This method is based on the EDF-algorithm. We consider one system process with several different threads for different applications. [4] A own message queue is assigned to each thread.

[4] An extension of the model that allows to have different system processes which all perform the function of the meta-scheduler is possible. The different processes are self-coordinating through a table in a shared memory segment. For the sake of simplicity we explain the method only with on system process.

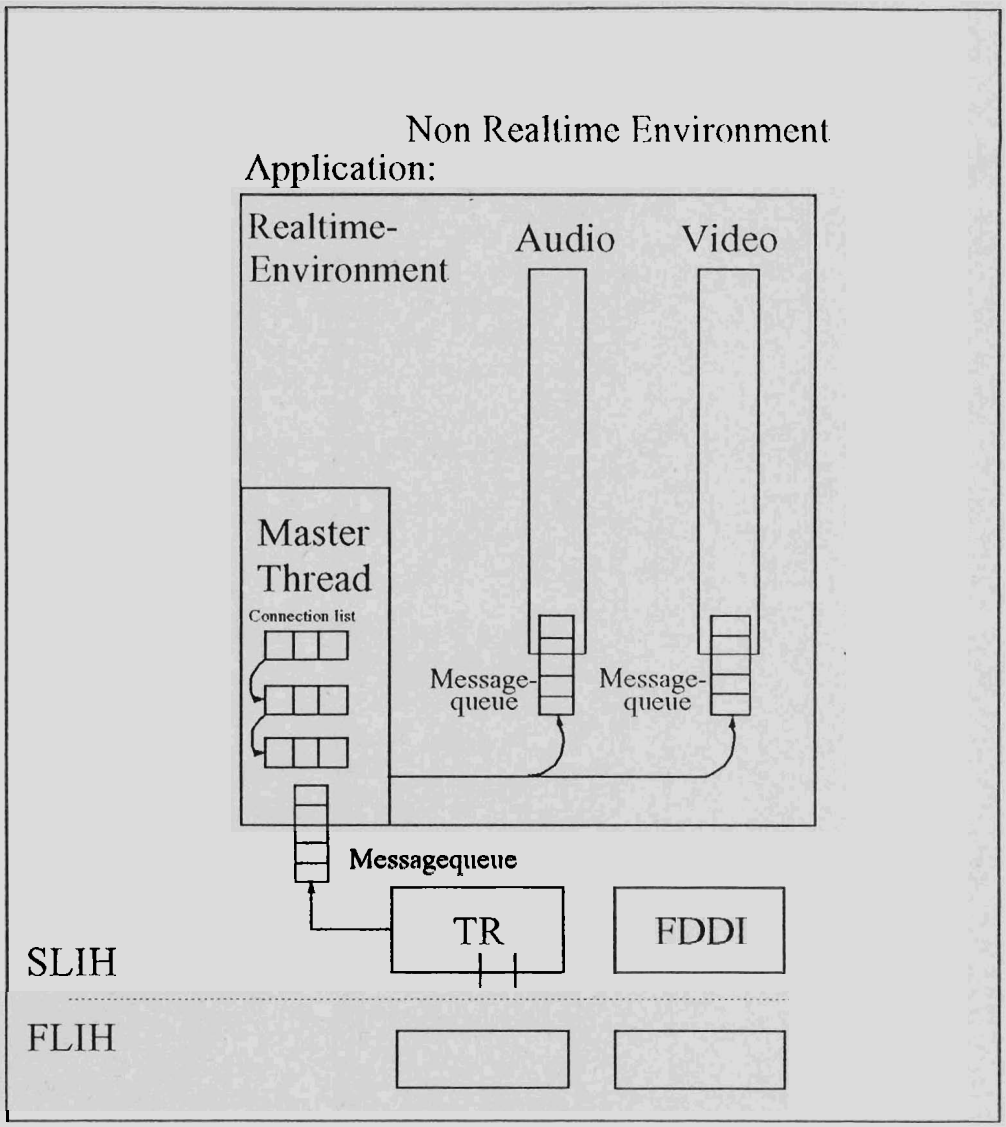

Figure 12. Structure of the Queue Monitoring Scheduler

The major tasks of the meta-scheduler are performed by the master thread that is not assigned to an application function. It runs with priority 30 in the priority class time-critical. Every incoming message is queued in the message queue of the master thread. If a message indicates a connection establishment the resource manager has to check if sufficient CPU-time is left to accept the connection. The SLIH sends all messages to the message queue of the master thread. Messages that are ahead of schedule are queued to their logical arrival time in the message queue of the master thread. At every scheduling-point the master thread dispatches the message with the earliest deadline to the message queue of its application thread. The master thread then sleeps for the duration of the guaranteed processing time plus laxity for possible interrupts. If there is any message in the message queue of the master thread that becomes critical during the run-time of a task and has an earlier deadline than the currently processed one the master thread only sleeps to the logical arrival time of that message. It preempts the former thread calculates its processing time and dispatches the critical message to the message queue of its application thread. The master thread has to ensure that the application thread of the new task has a higher system priority than the application thread of the old task. With every incoming message the master thread determines immediately a new schedule. A running thread is preempted during the re-scheduling.

If a thread has not finished processing within the given time it is preempted. A new task is chosen for processing. The preempted task can finish processing on a lower

priority if there is enough processor time left. Every new message of this task is scheduled according to its deadline. Tasks are only processed for the guaranteed amount of processing time with a critical priority. A malicious or incorrect program does not starve other tasks. Since all messages are scheduled according to their deadlines bursts are no processed at the expense of other tasks. There is no workahead of messages if the processor is idle.

Our main intention was to keep the scheduling overhead as low as possible. Priorities only have to be changed in exceptional situations. To reduced this overhead base priorities can be assigned to each thread according to their rate. Nevertheless, the overhead through the scheduling is still dynamic.

4.3.2 Distributed Access Control and Process-Time Monitoring

Tasks are scheduled according to the rate monotonic scheduling policy. There is one system process with multiple threads.[5]

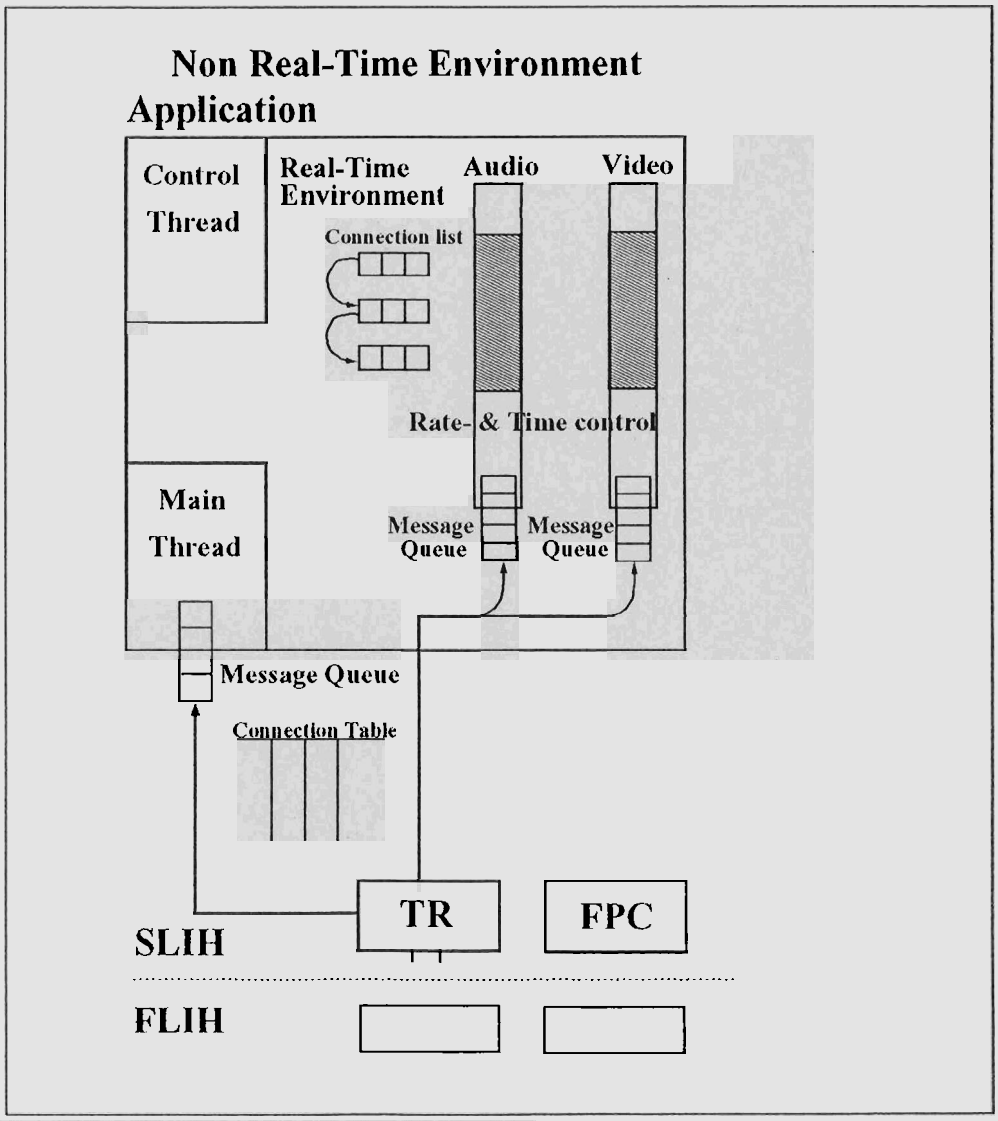

Figure 13. Structure of the DACProM Scheduler

The Distributed Access Control and Process-Time Monitoring (DACProM) meta-scheduler consists of a main thread, a control thread and several application threads. The main thread and the control thread are not assigned to application

[5] This model also can be extended to have more then one system process to perform scheduling functions.

functions.The main thread runs with priority 0 and the control thread with priority 30 in the priority class time-critical. A message that indicates a request for a connection is send to the message queue of the main thread. The main thread assigns an application thread to the connection if the schedulability test was positive. The message queue name of the application thread that is associated with a connection is stored in a table that is shared with the SLIH. After the connection establishment every message from the connection is queued in the message queue of its application thread. A unique priority is assigned to each connection according to the rate monotonic algorithm. This priority corresponds with the system priority of the thread.

In case of bursty traffic a high priority thread would process messages with the high system priority ahead of schedule. Low priority tasks would miss their deadlines while the messages of the high priority thread would be processed ahead of schedule. To prevent from such behavior and to control the rate of the messages each thread controls the arrival times of its messages. After each processing a thread sleeps to the logical arrival time of the next message. Workahead messages are queued till their logical arrival time is reached. The requests for all tasks are now periodic as required by the rate monotonic algorithm.

The control thread observes the processing behavior of the application threads. The processing time $tp(i)$ of a message is measured. During a control period all processing times are summed up $tp = tp(1) + \cdots + tp(n)$. Periodically the control thread checks the average processing time $dt = \frac{tp}{n}$.[6] If dt is larger then the specified processing time the offending thread is set on the penalty priority by the control thread.

The scheduling overhead is kept constant by the assignment of static priorities to each connection. The rate control is performed through each application thread after the processing of every message. The control thread prevents from the permanent violation of the schedule through offending tasks.

4.3.3 Design of the Actual Implementation

4.3.3.1 System Timer Constraints

Two main criterions were considered for the assessment of the two alternatives.

1. The scheduling overhead caused by each method
2. Their adaptability to operating system constrains

The first alternative is based on EDF, the theoretical processor utilization of this algorithm is 100%. The overhead of the scheduling is dynamic. The amount of tasks and messages influences the amount of required scheduling decisions and control to be done by the scheduler. The overhead turned out to be considerable

The second alternative is based on the rate monoton scheduling algorithm. The maximum processor utilization is 69%.[7] The scheduling overhead is nearly constant. Since we have also non-critical tasks running on the computer which are not scheduled by the meta-scheduler we do not consider the bounded processor utilization as a severe drawback.

During the design of the meta-scheduler we discovered that the timers provided by the operating system are not sufficient for the employment in real-time systems. The OS-timer calls are specified in milliseconds. The actual duration of the specified

[6] n = number of processed messages during the period.

[7] This boundary can be widely extended as described in chapter 3.3.

time interval will be affected by the hardware clock tick. A tick interval lasts approximately 31.25 milliseconds. Any time interval that is specified in milliseconds will essentially be rounded up to the next clock tick [34]. Multimedia applications require a timer granularity in the range of milliseconds or even finer [25]. There is no reasonable way to vary the interval of the hardware clock tick. Programs or device drivers that provide a more accurate time measurement are not varying the interval of the hardware clock tick.

A device driver called OS2HRT provides a timer with a granularity in the nanosecond range. The high resolution timer function has two output parameters.

- *timer.tic*
- *timer.count*

The *timer.tic* parameter counts the tick of the time of day clock. This clock is advanced by one tick approximately 18.2 times per second (every 55 milliseconds). To obtain a better accuracy the 8253 Timer/Counter component can be used. Timer 0 runs continuously counting down from 65536 to 0. Each time it reaches 0, it triggers an interrupt which advances the time of day clock by one tick. The 16-bit counter in the 8253 changes every 840 nanoseconds. The *timer.count* contains this counter. With this device driver we achieve a granularity of 840 nanoseconds [48].

The time of day clock is not identical with the hardware timer. The content of *timer.count* is not the number of counts which elapsed since the last hardware clock tick occurred. A hardware clock tick occurs approximately every 31.25 milliseconds whereas the time of day clock tick occurs approximately every 55 milliseconds. Therefore, the two timer have to be synchronized in order to use them simultaneously.

4.3.3.2 Structure of the Implemented Meta-Scheduler

Because of the insufficient timer support it was not possible two realize the introduced alternatives in the proposed way. For the first proposal the timer insufficiency is such a severe drawback that it was not possible to find any reasonable solution for the implementation of it. The second design proposal was modified. The high-resolution timer is employed when the granularity of the hardware timer is insufficient.

The functions of the main thread and the control thread are not affected by the timer problem. The rate control through a simple OS-sleep is replaced through a modified mechanism. A thread does not sleep after the processing of a message. It waits on the message queue for the arrival of a message. Every time a message arrives the thread checks if it is ahead of schedule. In this case it checks with the synchronized timer if it is possible to sleep to the next tick of the hardware clock. A thread is set on a workahead priority if the next hardware tick occurs later then the logical arrival time of the message, or if it resumes processing before the logical arrival time is reached. Every thread has to check at the beginning of processing if there is any higher priority thread running with the workahead priority that becomes critical during the processing of the own task. In this case the priority of the workahead thread is reset on its original priority. If a thread ends processing in a workahead state it resets its priority.

To measure the processing time of each message, the beginning of its processing is recorded. After the end of the processing, the difference between the start time and the end is calculated. If the processing is interrupted by another thread with a higher priority the high priority thread has to calculate the present processing time and after its own processing to reset the start time of the interrupted thread. Problems occur with asynchronous I/O. A thread is inactive as long as an asynchronous event is processed. I.e, the thread gives up control and another task can be processed by another thread during that time frame. It is not possible to measure the processing time of a thread with a lower priority that starts processing in that time

frame because the thread that performs the asynchronous actions resumes processing immediately after the end of the asynchronous event. The present processing time of the low priority thread can not be determinated when the high priority thread resumes processing. Since we only record the beginning and the end of the processing and calculate the processing time out of this the measured processing times are inaccurate. Apart of interrupts it always includes the time where a thread was inactive during an asynchronous event.

The scheduling method we apply prevents from the processing of bursts through higher priority threads at the expense of low priority threads. Rate control was implemented using the timing tools of the operating system. The overhead caused has to be accepted because there is no other way to serve the premises of the rate monotonic algorithm. The measured processing times does not reflect the exact CPU-time needed by a task. These are only a rough estimate of the time a task needs the CPU for processing.

4.3.4 Evaluation of the HeiMOS Solution

4.3.4.1 Performance Measurements of the Implementation

The modified DACProM meta-scheduler was implemented in C under OS/2 on a PS/2 with 25Mhz and a 80486 processor. There are several changes from OS/2 version 1.3 to OS/2 version 2.0. We took these changes into account during the development and kept the programs closely compatible for both versions.

A full description of the implementation can be found in [51]. Since the meta-scheduler is a basic component of HeiTS we have not yet had the opportunity to gather experience with genuine multimedia data generated and transferred through HeiTS. Experiments and measurements have been performed using test programs especially designed for this purpose. These test programs show that the meta-scheduler meets the described requirements.

To estimate the performance of the programs we measured truncated portions of the programs and important system calls individually. We found that the system call to change priorities requires approximately $73\mu s$ whereas a context switch takes approximately $47\mu s$. The processing time control takes for one connection 0.31 *ms*. For two connections 0.4 *ms* is required at average and every additional connection requires another 0.1 *ms*.

To set a thread on a workahead priority takes approximately four times the time then required for simple *DosSleep*. The overhead for the control of the processing time is acceptable but to be a useful tool for controlling it should be much more precise.

4.3.4.2 Known Limitations of the HeiMOS Solution

During the design of the meta-scheduler we had to consider various restrictions mainly through the operating system. This has negative effects on the functionality of the meta-scheduler. To evaluate the solution we have to consider all these restrictions and limitations. In this section we discuss them and show the limits of our solution.

Each single thread in the system is able to run with a priority in the priority class time-critical. A thread running in this priority class without the knowledge of the resource manager violates the calculated schedule, the processing guarantees given by the resource manager are not longer valid. A malicious program can block the whole system simply by running with the highest priority in the priority class time-critical without giving up the control anymore.

In OS/2 it is not possible to measure the exact time a thread is using the CPU. Any measurement of the processing time includes interrupts. Interrupts can not be disa-

bled since they may contain information necessary for the scheduling. If a thread was interrupted by a higher priority thread it also includes the time needed for the context switch. During asynchronous I/O a thread gives up its control. Another thread can use the CPU during that time. It is not possible to interrupt the time measurement during the asynchronous event. Therefore, the measured time is only a hint how long the processing of a task takes and does not reflect the time the CPU is needed by a task.

The system timer provided by OS/2 is insufficient. The hardware timer is enhanced by one clock tick approximately every 31.25 milliseconds. For a real-time system an acceptable granularity would be in the millisecond range. With the High-Resolution-Timer we have an accurate measurement tool. The problem is that it only can be used by an active thread. The granularity of the rate control is therefore determined by the granularity of the system timer. Our main objective was to build the meta-scheduler on top of the operating system without intervening into it. To improve our system we need more support from operating system side.

Real-time capabilities may be achieved in OS/2 by changing the OS itself: Either the scheduler may be enhanced by a class of fast threads, perhaps without time-slicing with the ability to mask interrupts for a short well defined period. Those threads should be reserved for CM-tasks and monitored by a system component with extensive control facilities. Performance enhancement of the scheduler itself incorporating some mechanisms of real-time scheduling like earliest-deadline first or least laxity first would be another solution.

The operating system has to provide sufficient timing and measurement tools. There has to be a possibility to measure the pure CPU-time required by thread for the processing of a task. A kind of watchdog timer would all so be sufficient. A system timer is needed that supplies a granularity in the millisecond range. This may be achieved through a single timer chip with the only task of triggering interrupts in a specified granularity.

The meta-scheduler provides the necessary real-time support for CM-application. It does not serve all requirements of a hard real-time system. Further work has been done by improving the timer capabilities in changing the OS2HRT-device driver.

5.0 Conclusion

Multimedia applications require real-time support either through the operating system or through another system component. The operating systems used in HeiTS are not conceived for the extensive support of real-time processing. HeiMOS is designed to provide this support in the end-systems of HeiTS. Therefore, a meta-scheduler was developed to run on top of OS/2. Time-critical tasks are scheduled to serve their process requirements as well as their time requirements.

To find the best method to schedule time-critical multimedia tasks we evaluated various real-time scheduling algorithms. It turned out that EDF and the rate monotonic algorithm are most suitable for the solution of this problem.

Based on this two algorithms we developed two meta-scheduler to run under OS/2. In the design the occurrence of multimedia data streams according to the linear bounded arrival process model and other restrictions had to be considered. One alternative was implemented. It turned out that the system timer provided by the operating system are not sufficient for real-time applications. To solve this problem we employed a special device driver. With this device driver a timer granularity in the nanosecond range can be achieved. This timer does not replace the system timer since it is a measurement tool that only can be used by active threads. Therefore, the necessary rate control is complicated and expensive. A method to control processing times and to react on offending behavior of tasks was implemented. Since the operating system does not supply the possibility to measure pure CPU-time the measured times include interrupts, context switch times, and asynchronous I/O. Therefore, it is only a rough estimation of the real CPU-time needed by a thread.

An exact and reliable real-time scheduling should be provided, or at least supported by the operating system. Either through the modification of the system scheduler, or through real-time tools that enhance the real-time capabilities of the operating system. It should provide an exclusive priority class especially for real-time processes that is controlled and monitored by a system process. Further, exact time and measurement tools are needed.

Acknowledgements

We gratefully acknowledge the many helpful advices from Wolfgang Burke and Carsten Vogt. We also would like to thank Ingrun Link for the drawings she contributed to this paper.

References

[1] **D. P. Anderson, L. Delgrossi, R. G. Herrtwich:** *Process Structure and Scheduling in Real-Time Protocol Implementations;* Kommunikation in verteilten Systemen, GI/ITG-Fachtagung, Proceedings, Springer Verlag, Mannheim Feb. 1991, pp. 83-95.

[2] **D. P. Anderson, R. G. Herrtwich, C. Schaefer:** *SRP: A Resource Reservation Protocol for Guaranteed-Performance Communication in the Internet;* Technical Report No.90-006, International Computer Science Institute, Feb. 1990.

[3] **D. P. Anderson:** *Meta-Scheduling for Distributed Continuous Media;* Technical Report No. UCB/CSD 90/599, Computer Science Division, University of California, Oct. 1990.

[4] **M. Andrews:** *Guaranteed Performance for Continuous Media in a General Purpose Distributed System;* Master Thesis, University of California, Berkeley, Oct. 1989.

[5] **D. P. Anderson, G. Homsy:** *A Continuous Media I/O Server and Its Synchronization Mechanism;* IEEE-Computer, Vol. 24, No. 10, Oct. 1991, pp. 51-57.

[6] **R. Baumann:** *Datenverarbeitung unter Zeitbedingungen;* Informatik Spektrum, Vol. 7, No. 2, Apr. 1984, pp. 62-65.

[7] **J. Blazewicz, G. Finke:** *Minimizing Mean Weighted Execution Time Loss on Identical and Uniform Processors;* Information Processing Letters Vol. 24, March 1987, pp. 258-263.

[8] **W Burke:** *Entwurf und Implementierung eines Pseudo-Echtzeit-Schedulers für AIX;* Diplomarbeit, Institut für Mathematische Maschinen und Datenverarbeitung, FAU Erlangen Nürnberg, Feb. 1992.

[9] **J.-Y. Chung, J. W. S. Liu:** *Algorithms for Scheduling Periodic Jobs to Minimize Average Error;* IEEE-Real-Time Systems Symposium, Huntsville, 1988, pp. 142-151.

[10] **J.-Y. Chung, J. W. S. Liu:** *Performance of Algorithms for Scheduling Periodic Jobs to avoid Timing Faults;* Proceedings of 22'nd Hawaii International Conference on System Sciences, Hawaii 1989, pp. 683-692.

[11] **D. W. Craig, C. M. Woodside:** *The Rejection Rate for Tasks with Random Arrivals, Deadlines and Preemptive Scheduling;* IEEE-Transaction on Software Engineering, Vol. 16, No. 10, Oct. 1990, pp. 1198-1208.

[12] **S.-C. Cheng, J. A. Stankovic, K. Ramamritham:** *Scheduling Algorithms for Hard Real-Time Systems -A Brief Survey;* Hard Real-Time Systems (J. A. Stankovic, K.Ramamritham, Hrsg), Washington, DC: IEEE Computer Society Press, 1988, pp.150-178.

[13] **M. L. Dertouzos:** *Control Robotics; The Procedural Control of Physical Processing;* Information Processing 74 - North Holland Publishing Company, 1974, pp. 807-813.

[14] **Deutsches Institut für Normung:** *Informationsverarbeitung-Begriffe;* DIN 43000, Berlin-Köln: Beuth, 1985.

[15] **D. Ferrari:** *Client Requirements for Real-Time Communication Services;* International Computer Science Institute, Technical Report 90-007, Berkely, March 1990.

[16] **E. A. Fox:** *Advances in Interactive Digital Multimedia Systems;* IEEE-Computer, Vol. 24, No.10, Oct.1991, pp. 9.-22.

[17] **S. French:** *Sequencing and Scheduling; an Introduction to the Mathematics of the Job Shop;* Ellis Horwood Limited, Chichester, 1982.

[18] **R. Govindan, D. P. Anderson:** *Scheduling and IPC Mechanisms for Continuous Media;* University of California, Computer Science Division, Technical Report No. UCB/CSD 91/622, Berkly, March 1991.

[19] **I. Greif, et al.:** *Computer Supported Cooperative Work: A Book of Readings;* Morgan Kaufman Publisher, San Mateo, 1988.

[20] **D. Haban, K. G. Shin:** *Application of Real-Time Monitoring to Scheduling Tasks with Random Execution Times;* IEEE-Real-Time Systems Symposium, Santa Monica, 1989, pp. 172-180.

[21] **R. Henn:** *Deterministische Modelle für die Prozessorzuteilung in einer harten Realzeit-Umgebung;* Dissertation am Fachbereich Mathematik der Technischen Universität München, Sept. 1975.

[22] **R. G. Herrtwich:** *Echtzeit;* Informatik Spektrum, Vol. 12, No. 2, Apr. 1989, pp. 93-96.

[23] **R. G. Herrtwich:** *Time Capsules: An Abstraction for Access to Continuous-Media Data;* IEEE-Real-Time Systems Symposium, Lake Buena Visat, 1990, pp. 11-20.

[24] **R. G. Herrtwich:** *Betriebsmittelvergabe unter Echtzeitgesichtspunkten;* Informatik Spektrum, Band 14, Heft 3, June 1991, pp. 123-136.

[25] **R. G. Herrtwich:** *Zeitkritische Datenströme in verteilten Multimedia-Systemen;* Kommunikation in verteilten Systemen, GI/ITG-Fachtagung, Proceedings, Springer Verlag, Mannheim Feb. 1991, pp. 68-82.

[26] **R. G. Herrtwich, R. Steinmetz:** *Towards Multimedia: Why and How;* Telekommunikation und multimediale Anwendungen der Informatik, GI-21. Jahrestagung, Proceedings, Springer Verlag, Darmstadt Oct.1991, pp. 327-342.

[27] **D. Hehmann, R. G. Herrtwich, R. Steinmetz:** *Creating HeiTS: Objectives of the Heidelberg High-Speed Transport System;* Technical Report No. 439102, IBM-ENC, Heidelberg, 1991.

[28] **J. Hyman, A. A. Lazar, G. Pacifics:** *MARS: The Magnet II Real-Time Scheduling Algorithm;* acm-Press, Computer Communication Review, Vol. 21, No. 4, SIGCOMM '91, Zürich, Sep. 1991, pp. 285-294.

[29] **K. S. Hong, J. Y.-T. Leung:** *On-Line Scheduling of Real-Time of Tasks;* IEEE-Real-Time-System Symposium, Huntsville, 1988, pp. 244-258.

[30] **E. Iacobucci:** *OS/2 Programer's Guid;* McGraw-Hill, Berkeley, 1988.

[31] **IBM Deutschland GmbH, Redaktion: K. Csikai:** *Fachausdrücke der Informationsverarbeitung, Wörterbuch und Glossar;* IBM Deutschland GmbH, 1985.

[32] **IBM Corp.:** *IBM Operating System/2 Programming Tools and Information Version 1.2: Control Program Programming Reference;* IBM Corp., First Edition, Sep. 1989.

[33] **IBM Corp.:** *AIX Version 3.1: Risc System/6000 as a Real-Time System;* IBM-International Technical Support Center, Austin, March 1991.

[34] **IBM Corp.:** *IBM OS/2 Programming Guide, Volume I, Control Program Interface, Preliminary Draft;* IBM Corp., First Edition, March 1992.

[35] **K. Jeffay:** *Analysis of a Synchronization and Scheduling Discipline for Real-Time Tasks with Preemption Constraints;* IEEE-Real-Time Systems Symposium, Santa Monica, 1989, pp. 295-305.

[36] **K. Jeffay:** *Scheduling Sporadic Tasks with Shared Resources in Hard-Real-Time Systems;* University of North Carolina at Chapel Hill, Department of Computer Science, Technical Report TR90-039, Nov. 1990.

[37] **K. Jeffay, D. L. Stone, D. E. Poirier:** *YARTOS: Kernel Support for Efficient, Predictable Real-Time Systems;* Proceedings of IFAC, Workshop on Real-Time Programming, Atlanta, May 1991, Pergamon Press.

[38] **K. Jeffay, D. L. Stone, F. D. Smith:** *Kernel Support for Live Digital Audio and Video;* Second International Workshop on Network and Operating System Support for Digital Audio and Video, Heidelberg, Dec. 1991.

[39] **M. S. Kogan:** *OS/2 2.0 Overview;* OS/2 Notebook, Microsoft Press, Washington 1991.

[40] **C. M. Krishna, Y. H. Lee:** *Real-Time Systems;* IEEE-Computer, May 1991, pp. 10-11.

[41] **B. Lamparter, W. Effelsberg:** *X-Movie: Transmission and Presentation of Digital Movies under X;* Second International Workshop on Network and Operating System Support for Digital Audio and Video, Heidelberg, Nov. 1991.

[42] **E. L. Lawler:** *Optimal Sequencing of a Single Machine Subject to Precedence Constraints;* Management Science, Vol.19, No. 5, Jan. 1973, pp. 544-546.

[43] **J. Y.-T Leung, M. L. Merrill:** *A Note on Preemptive Scheduling of Periodic Real-Time Tasks;* Information Processing Letters, Vol. 11, No. 3, Nov. 1980, pp. 115-118.

[44] **J. Y.-T. Leung, J. Whitehead:** *On the Complexity of Fixed-Priority Scheduling of Periodic Real-Time Tasks;* Performance Evaluation 2, 1982, pp. 237-350.

[45] **G. Letwin:** *Inside OS/2;* Microsoft Press, Washington, 1988.

[46] **J. P. Lehoczky, L. Sha:** *Performance of Real-Time Bus Scheduling Algorithms;* ACM Performance Evaluation Review, Vol. 14, No. 1, May 1986, pp. 44-53.

[47] **C. L. Liu, J. W. Layland:** *Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment;* Journal of the Association for Computing Machinery, Vol.20, No.1, Jan. 1973, pp. 46-61.

[48] **M. H. Linchan:** *Timer Availability, Notice;* Program description of the OS2HRT device driver, Apr. 1984.

[49] **J. W. S. Liu, K.-J. Lin, S. Naturajan:** *Scheduling Real-Time, Periodic Jobs Using Imprecise Results;* IEEE-Real-Time Systems Symposium, San Jose, 1987, pp. 252-260.

[50] **J. W. S. Liu, K.-J. Lin, W.-K. Shin, A. C. Yu:** *Algorithms for Scheduling Imprecise Computations;* IEEE-Computer, May 1991, pp. 58-68.

[51] **A. Mauthe:** *Echtzeit-Verarbeitung von Multimedia-Protokollen;* Diplomarbeit, Lehrstuhl für Praktische Informatik IV, Prof. Dr. W. Effelsberg, Univertsität Mannheim, May 1992.

[52] **C. W. Merce, H. Tokuda:** *The ARTS Real-Time Object Model;* IEEE-Real-Time System Symposium, Lake Buena Vista, 1990, pp. 2-10.

[53] **C. W. Merce, H. Tokuda:** *Priority Consistency in Protocol Architectures;* Second International Workshop on Network and Operating System Support for Digital Audio and Video, Heidelberg, Nov. 1991.

[54] **C. W. Merce, H. Tokuda:** *An Evaluation of Priority Consistency in Protocol Architectures;* Carnegie-Mellon University, Pitsburgh, 1991.

[55] **T. Meyer:** *Anwendungsszenarien für verteilte Multimedia-Systeme basierend auf Hochgeschwindigkeitsnetzen;* Diplomarbeit, Lehrstuhl für Praktische Informatik IV, Prof. Dr. W. Effelsberg, Univertsität Mannheim, Jul. 1991.

[56] **A. K. Mok:** *The Design of Real-Time Programing Systems Based on Process Models;* IEEE-Real-Time Systems Symposium, Austin, 1984.

[57] **A. D. Narasimhalu, S. Christodoulakis:** *Multimedia Information Systems: The Unfolding of a Reality;* IEEE-Transaction on Computer, Vol. 24, No.10, Oct. 91, pp. 6-8.

[58] **R. Nagarajan, C. Vogt:** *Guaranteed-Performance of Multimedia Traffic over the Token Ring;* Technical Report No.439201, IBM-ENC, Heidelberg, 1992.

[59] **J. Nehmer:** *Systemarchitektur von Realzeitsystemen;* Informatik Spektrum, Vol. 7, No. 2, Apr. 1984, pp. 65-72.

[60] **R. Levin, et all.:** *Operating Systems Reveiw;* ACM Press, Operating Systems Review, Vol.23, No.3, Jul. 1989.

[61] **R. Orfali, D. Harkey:** *Client-Server Programming with OS/2, Extended Edition;* Van Nostrand Reinhold, New York, 1991.

[62] **R. Rajkumar, L. Sha, J. P. Lehoczky:** *Real-Time Synchronization Protocols for Multiprocessors;* IEEE-Real-Time Systems Symposium, Huntsville, 1988, pp. 259-269.

[63] **H. Rzehak:** *Echtzeitkommunikationssysteme: Eine Einführung in die Problembereiche und Lösungsansätze;* Telekommunikation und multimediale Anwendungen der Informatik, GI-21. Jahrestagung, Proceedings, Springer Verlag, Darmstadt Oct.1991, pp. 631-642.

[64] **P. Sander:** *Entwurf und Realisierung einer verteilten Videokamerafernsteuerung auf Basis eines Multimedia-Transportsystems (Arbeitstitel);* Diplomarbeit, Lehrstuhl für Praktische Informatik IV, Prof. Dr. W. Effelsberg, Univertsität Mannheim, Jun. 1992.

[65] **E. Schoop:** *HERMES-Ein Hypermediasystem für die betriebswirtschaftliche Ausbildung;* Telekommunikation und multimediale Anwendungen der Informatik, GI-21. Jahrestagung, Proceedings, Springer Verlag, Darmstadt Oct. 1991, pp. 608-617.

[66] **L. Sha, J. B. Goodenough:** *Real-Time Scheduling Theory and ADA;* IEEE-Transaction on Computer, Vol. 23, No. 4, Apr. 1990, pp. 53-64.

[67] **L. Sha, R. Rajkumar:** *Real-Time Systems, A Tutorial of the Rate-Monotonic Scheduling Framework with Bus-Related Issues;* Futurbus+ P896.3, Draft 4.0, 1989.

[68] **W.-K. Shin, J. W. S. Liu, J.-Y. Chung, D. W. Gillies:** *Scheduling Tasks with Ready Times and Deadlines to Minimize Average Error;* ACM Press, Operating Systems Review, Vol. 23, No.3, Jul. 1989, pp. 14-28.

[69] **L. Sha, J. P. Lehoczky, R. Rajkumar:** *Solutions of some Practical Problems in Prioritized Preemptive Scheduling;* IEEE-Real-Time Systems Symposium, New Orleans, 1986, pp. 181-191.

[70] **B. Sprunt, et al:** *Implementing Sporadic Servers in ADA;* Carnegie-Mellon University, Pitsburgh, Software Engineering Institut, May 1990.

[71] **B. Sprunt, L. Sha, J. Lehoczky:** *Aperiodic Task Scheduling for Hard Real-Time Systems;* The Journal of Real-Time Systems, Vol. 1, 1989, pp. 27-60.

[72] **R. Steinmetz, J. Rückert, W. Racke:** *Multimedia-Systeme;* Informatik Spektrum, Springer Verlag, Vol. 13, No. 5, Oct. 1990, pp. 280-282.

[73] **R. Steinmetz, R. G. Herrtwich:** *Integrierte verteilte Multimedia-Systeme;* Informatik Spektrum, Springer Verlag, Vol. 14, No. 5, Oct. 1991, pp. 249-260.

[74] **Ralf Steinmetz, Thomas Meyer:** *Modelling Distributed Multimedia Applications;* IEEE Int. WS on Advanced Communications and Applications for HS Networks, München, March 1992.

[75] **J. A. Stankovic, K. Ramamritham:** *The Spring Kernel: A new Paradigma of Real-Time Operating Systems;* ACM Press, Operating System Review, Vol. 23, No.3, Jul. 1989, pp. 54-71.

[76] **T. Teshima:** *Constructing Image Databases for the Collection of Art Museums;* Telekommunikation und multimediale Anwendungen der Informatik, GI-21. Jahrestagung, Proceedings, Springer Verlag, Darmstadt Oct. 1991, pp. 50-56.

[77] **W. Tawbi, E. Horlait, S. Dupuy:** *High Speed Protocols: State of the Art in Multimedia Applications;* Second International Workshop on Network and Operating System Support for Digital Audio and Video, Heidelberg, Nov. 1991.

[78] **H. Tokuda, C. W. Mercer:** *ARTS: A Distributed Real-Time Kernel;* ACM Press, Operating Systems Review, Vol.23, no.3, Jul.1989, pp. 29-53.

[79] **C. Topolocic (Ed.):** *Experimental Internet Stream Protocol, Version 2 (ST II);* Internet Network Working Group, RFC 1190, Oct. 1990

[80] **H. Tokuda, J. W. Wendorf, H.-Y. Wang:** *Implementation of a Time-Driven Scheduler for Real-Time Operating Systems;* IEEE-Real-Time Systems Symposium, San Jose, 1987, pp. 271-280.

[81] **S. Vesttal:** *The Accuracy of Predicting Rate Monotonic Scheduling Performance;* Proceedings of TRI-ADA '90, Dec. 1990.

[82] **C. Vogt, R.G. Herrtwich, R. Nagarajan:** *HeiRAT: The Heidelberg Resource Administration Technique, Design Philosophy and Goals;* Technical Report No. 43.9213, IBM-ENC, Heidelberg, 1992.

[83] **Video Logic:** *Multimedia Development Tools & Resources;* VideoLogic Limited, Hertfordshire, 1990.

[84] **H. M. Vin, P. T. Zellweger, D. C. Swinehart, P. V. Rangan:** *Multimedia Conferencing in the Etherphone Environment;* IEEE-Computer, Vol. 24, No. 10, Oct. 1991, pp. 69-79.

[85] **C. M. Woodside, D. W. Craig:** *Local Non-Preemptive Scheduling Policies for Hard Real-Time Distributed Systems;* IEEE-Real-Time Systems Symposium, San Jose, 1987, pp. 12-17.

[86] **H. Zhang, S. Keshav:** *Comparison of Rate-Based Service Desciplines;* acm-Press, Computer Communication Review, Vol. 21, No. 4, SIGCOMM'91, Zürich, Sep. 1991, pp. 113-122.