Chapter 9

Multimedia Operating Systems

9.1 Introduction

The operating system is the shield of the computer hardware against all software components. It provides a comfortable environment for the execution of programs, and it ensures effective utilization of the computer hardware. The operating system offers various services related to the essential resources of a computer: CPU, main memory, storage and all input and output devices.

For the processing of audio and video, multimedia application demands that humans perceive these media in a natural, error-free way. These continuous media data originate at sources like microphones, cameras and files. From these sources, the data are transferred to destinations like loudspeakers, video windows and files located at the same computer or at a remote station. On the way from source to sink, the digital data are processed by at least some type of move, copy or transmit operation. In this data manipulation process there are always many resources which are under the control of the operating system. The integration of discrete and continuous multimedia data demands additional services from many operating system components.

The major aspect in this context is real-time processing of continuous media data. Process management must take into account the timing requirements imposed by the handling of multimedia data. Appropriate scheduling methods should be applied. In contrast to the traditional real-time operating systems, multimedia operating systems also have to consider tasks without hard timing restrictions under the aspect of fairness.

To obey timing requirements, single components are conceived as resources that are reserved prior to execution. This concept of *resource reservation* has to cover all resources on a data path, i.e., all resources that deal with continuous media. It also may affect parts of the application that process continuous media data. In distributed systems, for example, resource management also comprises network capacity [HVWW94].

The *communication* and *synchronization* between single processes must meet the restrictions of real-time requirements and timing relations among different media. The main memory is available as a shared resource to single processes.

In multimedia systems, *memory management* has to provide access to data with a guaranteed timing delay and efficient data manipulation functions. For instance, physical data copy operations must be avoided due to their negative impact on performance; buffer management operations (such as are known from communication systems) should be used.

Database management is an important component in multimedia systems. However, database management abstracts the details of storing data on secondary media storage. Therefore, database management should rely on file management services provided by the multimedia operating system to access single files and file systems. For example, the incorporation of a CD-ROM XA file system as an integral part of a multimedia *file system* allows transparent and guaranteed continuous retrieval of audio and video data to any application using the file system; the database system is one of those applications. However, database systems often implement their own access to stored data.

Since the operating system shields devices from applications programs, it must provide services for *device management* too. In multimedia systems, the important issue is the integration of audio and video devices in a similar way to any other input/output device. The addressing of a camera can be performed similar to the addressing of a keyboard in the same system, although most current systems do not apply this technique.

Product information dealing with operating system extensions for the integration of multimedia [IBM92c, Win91, DM92, IBM92e, IBM92d] typically provide a detailed description of application interfaces. In this chapter we will concentrate on the basic concepts and internal tasks of a multimedia operating system.

As the essential aspect of any multimedia operating system is the notion of *real-time*, the following section details this idea in its relationship to multimedia. Subsequently, the concept of resource management is discussed. The section on process management contains a brief presentation of traditional real-time scheduling algorithms. Further, their suitability and adaptability toward continuous media processing is examined. The section on file systems outlines disk access algorithms, data placement and structuring. The subsequent sections illustrate interprocess communication and synchronization, memory management and device management. This chapter concludes with a discussion of typical system architectures which comprise real-time and non-real-time environments.

9.2 Real Time

Since the notion of *real-time* developed independently from research in continuous media processing, the next section starts with a general definition of real-time. Later, it shows the relevance of real-time for multimedia data and processes.

9.2.1 The Notion of "Real-Time"

The German National Institute for Standardization, DIN, similar to the American ANSI, defines a real-time process in a computer system as follows:

A real-time process is a process which delivers the results of the processing in a given time-span.

Programs for the processing of data must be available during the entire run-time of the system. The data may require processing at an a priori known point in time, or it may be demanded without any previous knowledge [Ger85]. The system must enforce externally-defined time constraints. Internal dependencies and their related time limits are implicitly considered. External events occur - depending on the application - deterministically (at a predetermined instant) or stochastically (randomly). The real-time system has the permanent task of receiving information from the environment, occurring spontaneously or in periodic time intervals, and/or delivering it to the environment given certain time constraints.

The main characteristic of real-time systems is the correctness of the computation. This correctness does not only apply to errorless computation, but also on the time in which the result is presented [SR89]. Hence, a real-time system can fail not only if massive hardware or software failures occur, but also if the system is unable to execute its critical workload in time [KL91]. Deterministic behavior of the system refers to the adherence of time spans defined in advance for the manipulation of data, i.e., meeting a guaranteed response time. Speed and efficiency are not - as is often mistakenly assumed - the main characteristics of a real-time system. In a petrochemical plant, for example, the result is not only unacceptable when the engine of a vent responds too quickly, but also when it responds with a large delay. Another example is the playback of a video sequence in a multimedia system. The result is only acceptable when the video is presented neither too quickly nor too slowly. Timing and logical dependencies among different related tasks, processed at the same time, also must be considered. These dependencies refer to both internal and external restrictions. In the context of multimedia data streams, this refers to the processing of synchronized audio and video data where the relation between the two media must be considered.

Deadlines

A deadline represents the latest acceptable time for the presentation of a processing result. It marks the border between normal (correct) and anomalous (failing) behavior. A real-time system has both hard and soft deadlines.

The term *soft deadline* is often used for a deadline which cannot be exactly determined and which failing to meet does not produce an unacceptable result. We understand a soft deadline as a deadline which in some cases is missed and may yet be tolerated as long as (1) not too many deadlines are missed and/or (2) the deadlines are not missed by much. Such soft deadlines are only reference points with a certain acceptable tolerance. For example, the start and arrival times of planes or trains, where deadlines can vary by about ten minutes, can be considered as soft deadlines.

Whereas soft deadlines may be violated, ***hard deadlines*** should never be violated. A hard deadline violation is a system failure. Hard deadlines are determined by the physical characteristics of real-time processes. Failing such a deadline results in costs that can be measured in terms of money (e.g., inefficient use of raw materials in a process control system), or human and environmental terms (e.g., accidents due to untimely control in a nuclear power plant or fly-by-wire avionics systems) [Jef90].

Characteristics of Real Time Systems

The necessity of deterministic and predictable behavior of real-time systems requires processing guarantees 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 guarantees are valid only under the premise that no processing machine collapses during the run-time of real-time processes. A real-time system is distinguished by the following features (c.f. [SR89]):

- Predictably fast response to time-critical events and accurate timing information. For example, in the control system of a nuclear power plant, the response to a malfunction must occur within a well-defined period to avoid a potential disaster.

- A high degree of schedulability. Schedulability refers to the degree of resource utilization at which, or below which, the deadline of each time-critical task can be taken into account.

- Stability under transient overload. Under system overload, the processing of critical tasks must be ensured. These critical tasks are vital to the basic functionality provided by the system.

Management of manufacturing processes and the control of military systems are the main application areas for real-time systems. Such process control systems are responsible for real-time monitoring and control. Real-time systems are also used as command and control systems in fly-by-wire aircraft, automobile anti-lock braking systems and the control of nuclear power plants [KL91]. New application areas for real-time systems include computer conferencing and multimedia in general, the topic of our work.

9.2.2 Real Time and Multimedia

Audio and video data streams consist of single, periodically changing values of continuous media data, e.g., audio samples or video frames. Each Logical Data Unit (LDU) must be presented by a well-determined deadline. Jitter is only allowed *before* the final presentation to the user. A piece of music, for example, must be played back at a constant speed. To fulfill the timing requirements of continuous media, the operating system must use real-time scheduling techniques. These techniques must be applied to all system resources involved in the continuous media data processing, i.e., the entire end-to-end data path is involved. The CPU is just one of these resources – all components must be considered including main memory, storage, I/O devices and networks.

The *real-time requirements* of traditional real-time scheduling techniques (used for command and control systems in application areas such as factory automation or aircraft piloting) have a high demand for security and fault-tolerance. The requirements derived from these demands somehow counteract real-time scheduling efforts applied to continuous media. Multimedia systems which are not used in traditional real-time scenarios have different (in fact, more favorable) real-time requirements:

- The *fault-tolerance* requirements of multimedia systems are usually less strict than those of real-time systems that have a direct physical impact. The short time failure of a continuous media system will not directly lead to the destruction of technical equipment or constitute a threat to human life. Please note that this is a general statement which does not always apply. For example, the support of remote surgery by video and audio has stringent delay and correctness requirements.

- For many multimedia system applications, missing a deadline is not a severe failure, although it should be avoided. It may even go unnoticed, e.g., if an uncompressed video frame (or parts of it) is not available on time it can simply be omitted. The viewer will hardly notice this omission, assuming it does not happen for a contiguous sequence of frames. Audio requirements are more stringent because the human ear is more sensitive to audio gaps than the human eye is to video jitter.
- A sequence of digital continuous media data is the result of periodically sampling a sound or image signal. Hence, in processing the data units of such a data sequence, all time-critical operations are periodic. Schedulability considerations for periodic tasks are much easier than for sporadic ones [Mok84].
- The bandwidth demand of continuous media is not always that stringent; it must not be a priori fixed, but it may eventually be lowered. As some compression algorithms are capable of using different compression ratios – leading to different qualities – the required bandwidth can be negotiated. If not enough bandwidth is available for full quality, the application may also accept 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. This is known as scalable video.

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 applications must meet external requirements too. A distributed music rehearsal is a futuristic example: music played by one musician on an instrument connected to his/her workstation must 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 just the input or only the output of continuous media, delay bounds are more flexible. Consider the playback 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 only notice any difference in delay as start delay (i.e., for the first video frame to be displayed).

9.3 Resource Management

Multimedia systems with integrated audio and video processing are at the limit of their capacity, even with data compression and utilization of new technologies. Current computers do not allow *processing* of data according to their deadlines without any resource reservation and real-time process management. *Processing* in this context refers to any kind of manipulation and communication of data. This stage of development is known as *the window of insufficient resources* (see Figure 9.1) [ATW+90]. With CD-DA (Compact Disc Digital Audio) quality, the highest

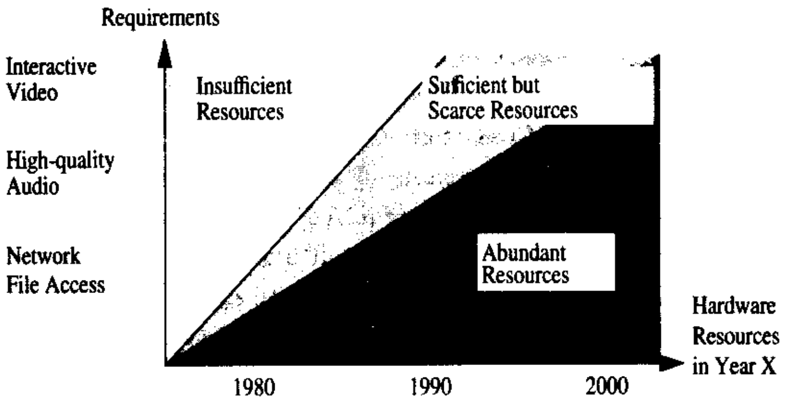

Figure 9.1: *Window of insufficient resources.*

audio requirements are satisfied. In video technology, the required data transfer rate will go up with the development of digital HDTV and larger TV screens. Therefore, no redundancy of resource capacity can be expected in the near future.

In a multimedia system, the given timing guarantees for the processing of continuous media must be adhered to by every hardware and software system component along the data path. The actual requirements depend on the type of media and the nature of the applications supported[SM92a]. For example, a video image should not be presented late because the communication system has been busy with a transaction from a database management system. In any realistic scenario we encounter several multimedia applications which concurrently make use of shared resources. Hence, even high bandwidth networks and huge processing capabilities require the use of real-time mechanisms to provide guaranteed data delivery. Further, the concept of integration does not allow solving this problem just by a slight modification of the system for traditional applications.

Thus, in an integrated distributed multimedia system, several applications compete for system resources. This shortage of resources requires careful allocation. The system management must employ adequate scheduling algorithms to serve the requirements of the applications. Thereby, the resource is first allocated and then managed.

Resource management in distributed multimedia systems covers several computers and the involved communication networks. It allocates all resources involved in the data transfer process between sources and sinks. For instance, a CD-ROM/XA device must be allocated exclusively, each CPU on the data path must provide 20% of its capacity, the network must allocate a certain amount of its bandwidth and the graphic processor must be reserved up to 50% for such a process. The required throughput and a certain delay is guaranteed. At the connection establishment phase, the resource management ensures that the new "connection" does not violate performance guarantees already provided to existing connections. Applied to operating systems, this model covers the CPU (including process management), memory management, the file system and device management. Therefore, we chose to detail this issue for all resources in a generic notion of resources in the following paragraphs. The resource reservation is identical for all resources, whereas the management is different for each.

9.3.1 Resources

A resource is a system entity required by tasks for manipulating data. Each resource has a set of distinguishing characteristics classified using the following scheme:

- A resource can be active or passive. An active resource is the CPU or a network adapter for protocol processing; it provides a service. A passive resource is the main memory, communication bandwidth or a file system (whenever we do not take care of the processing of the adapter); it denotes some system capability required by active resources.

- A resource can be either used exclusively by one process at a time or shared between various processes. Active resources are often exclusive; passive resources can usually be shared among processes.
- A resource that exists only once in the system is known as a single, otherwise it is a multiple resource. In a transputer-based multiprocessor system, the individual CPU is a multiple resource.

Each resource has a capacity which results from the ability of a certain task to perform using the resource in a given time-span. In this context, capacity refers to CPU capacity, frequency range or, for example, the amount of storage. For real-time scheduling, only the temporal division of resource capacity among real-time processes is of interest. Process management belongs to the category of active, shared, and most often single resources. A file system on an optical disk with CD-ROM XA format is a passive, shared, single resource.

9.3.2 Requirements

The requirements of multimedia applications and data streams must be served by the single components of a multimedia system. The resource management maps these requirements onto the respective capacity. The transmission and processing requirements of local and distributed multimedia applications can be specified according to the following characteristics:

1. The throughput is determined by the needed data rate of a connection to satisfy the application requirements. It also depends on the size of the data units.
2. We distinguish between local and global (end-to-end) delay:
   (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 total delay for a data unit to be transmitted from the source to its destination. For example, the source of a video telephone is the camera, the destination is the video window on the screen of the partner.

3. The jitter (or delay jitter) determines the maximum allowed variance in the arrival of data at the destination.

4. The reliability defines error detection and correction mechanisms used for the transmission and processing of multimedia tasks. Errors can be ignored, indicated and/or corrected. It is important to notice that error correction through re-transmission is rarely appropriate for time-critical data because the re-transmitted data will usually arrive late. Forward error correction mechanisms are more useful. In terms of reliability, we also mean the CPU errors due to unwanted delays in processing a task which exceed the demanded deadlines.

In accordance with communication systems, these requirements are also known as *Quality of Service parameters (QoS).*

9.3.3 Components and Phases

One possible realization of resource allocation and management is based on the interaction between clients and their respective resource managers. The client selects the resource and requests a resource allocation by specifying its requirements through a QoS specification. This is equivalent to a workload request. First, the resource manager checks its own resource utilization and decides if the reservation request can be served or not. All existing reservations are stored. This way, their share in terms of the respective resource capacity is guaranteed. Moreover, this component negotiates the reservation request with other resource managers, if necessary.

The following example of a distributed multimedia system illustrates this generic scheme. During the connection establishment phase, the QoS parameters are usually negotiated between the requester (client application) and the addressed resource manager. The negotiation starts in the simplest case with specification of the QoS parameters by the application. The resource manager checks whether these requests can be guaranteed 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), and the resource manager calculates the best achievable value for the third parameter (e.g., delay). To negotiate the parameters for end-to-end connections over one or more computer networks, resource reservation protocols like ST-II are employed [Top90]. Here, resource managers of the single components of the distributed system allocate the necessary resources.

In the case shown in Figure 9.2, two computers are connected over a LAN. The transmission of video data between a camera connected to a computer server and the screen of the computer user involves, for all depicted components, a resource manager.

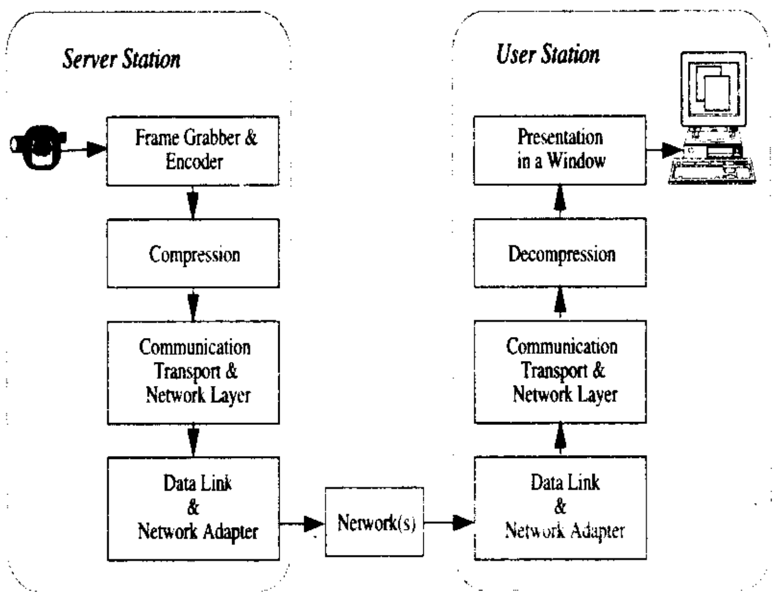
Server Station
Frame Grabber & Encoder
Compression
Communication Transport & Network Layer
Data Link & Network Adapter
Network(s)
User Station
Presentation in a Window
Decompression
Communication Transport & Network Layer
Data Link & Network Adapter

Figure 9.2: *Components grouped for the purpose of video data transmission.*

This example illustrates that, in addition to the individual resource managers, there must exist a protocol for coordination between these services, such as ST-II.

Phases of the Resource Reservation and Management Process

A resource manager provides components for the different phases of the allocation and management process:

1. *Schedulability Test*

   The resource manager checks with the given QoS parameters (e.g., throughput and reliability) to determine if there is enough remaining resource capacity available to handle this additional request.

2. *Quality of Service Calculation*

   After the schedulability test, the resource manager calculates the best possible performance (e.g., delay) the resource can guarantee for the new request.

3. *Resource Reservation*

   The resource manager allocates the required capacity to meet the QoS guarantees for each request.

4. *Resource Scheduling*

   Incoming messages from connections are scheduled according to the given QoS guarantees. For process management, for instance, the allocation of the resource is done by the scheduler at the moment the data arrive for processing.

With respect to the last phase, for each resource a scheduling algorithm is defined. The schedulability test, QoS calculation and resource reservation depend on this algorithm used by the scheduler.

9.3.4 Allocation Scheme

Reservation of resources can be made either in a pessimistic or 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 which might ever be needed by a task is reserved. Resource conflicts are therefore avoided. 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 discrete media tasks. This method results in a guaranteed QoS.

- With the *optimistic approach,* resources are reserved according to an average workload only. This means that the CPU is only reserved for the average processing time. This approach may overbook resources with the possibility of unpredictable packet delays. QoS parameters are met as far as possible. Resources are highly utilized, though an overload situation may result in failure. To detect an overload situation and to handle it accordingly a monitor can be implemented. The monitor may, for instance, preempt processes according to their importance.

The optimistic approach is considered to be an extension of the pessimistic approach. It requires that additional mechanisms to detect and solve resource conflicts be implemented.

9.3.5 Continuous Media Resource Model

This section specifies a model frequently adopted to define QoS parameters and hence, the characteristics of the data stream. It is based on the model of Linear Bounded Arrival Processes (LBAP), as described in [And93]. In this model a distributed system is decomposed into a chain of resources traversed by the messages on their end-to-end path. Examples of such resources are single schedulable devices (such as CPU) or combined entities (such as networks).

The data stream consists of LDUs. In this context, we call them messages. In a first step, the data stream itself is characterized as strictly periodic, irregular with a definite maximum message size. Various data streams are independent of each other.

A closer inspection shows a possible variance of the message rate, the maximum rate is well-defined. This variance of the data rate results in an accumulation of messages (burst), where the maximal range is defined by the maximum allowed number of messages.

In the LBAP model, a burst of messages consists of messages that arrived ahead of schedule. LBAP is a message arrival process at a resource defined by three parameters:

- M = Maximum message size (byte/message).
- R = Maximum message rate (message/second).
- B = Maximum Burstiness (message).

Example

The 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. Single channel audio data are transferred from the CD player of this workstation over the network to the other computer. At this remote station, the audio data are delivered to a speaker. The audio signal is sampled with the frequency of 44.1 kHz. Each sample is coded with 16 bits. This results in a data rate of:

$$R_{byte} = 44100Hz \times \frac{16bits}{8bits/byte} = 88200bytes/s$$

The samples on a CD are assembled into frames. These frames are the audio messages to be transmitted. Seventy-five of these audio messages are transmitted per second (R) according to the CD-format standard. Therefore, we encounter a maximum message size of:

$$M = \frac{88200bytes/s}{75messages/s} = 1176bytes/message$$

Up to 12000 bytes are assembled into one packet and transmitted over the LAN. In a packet of 12000 bytes transmitted over the LAN, we will never encounter more messages than:

$\frac{12000 bytes}{1176 bytes/message} \geq 10 messages = B$

It obviously follows that:

- M = 1176 bytes/message.
- R = 75 messages/s.
- B = 10 messages.

Burst

In the calculation below it is assumed that, because of lower adjacent data rate, a burst never exceeds a maximum data rate. Hence, bursts do not succeed one another. During a time interval of length t, the maximum number of messages arriving at a resource must not exceed:

$\bar{M} = B + R \times t(message)$

For example, assume $t = 1s$.

$\bar{M} = (10 messages + 75 messages/s \times 1s = 85 messages$

The introduction of Burstiness B allows for short time violations of the rate constraint. This way, programs and devices that generate bursts of messages can be modeled. Bursts are generated, e.g., when data is transferred from disks in a bulk transfer mode or, in our example, when messages are assembled into larger packets.

Maximum Average Data Rate

The maximum average data rate of the LBAP is:

$$\bar{R} = M \times R(bytes/s)$$

For example:

$$\bar{R} = (1176 bytes/message \times 75 messages/s = 88200 bytes/s)$$

Maximum Buffer Size

Messages are processed according to their rate. Messages which arrive "ahead of schedule" must be queued. For delay period, the buffer size is:

$$S = M \times (B+1)(bytes)$$

For example:

$$S = (1176 bytes/message \times 11 message = 12936 bytes$$

Logical Backlog

The function $b(m)$ represents the logical backlog of messages. This is the number of messages which have already 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:

$b(m_0) = 0$ messages

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

For example:

$a_{i-1} = 1.00s; a_i = 1.01\bar{3}s; b(m_{i-1}) = 4$ messages

$b(m_i) = max(0 \text{ messages}, 4 \text{ messages} - (1.01\bar{3}s - 1.00s) \times 75 \text{ messages}/s + 1 \text{ message}) =$ 4 messages

Logical Arrival Time

The logical arrival time defines the earliest time a message m_i can arrive at a resource when all messages arrive according to their rate. The logical arrival time of a message can then be defined as:

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

For example:

$l(m_i) = 1.01\bar{3}s + \frac{4 \text{ messages}}{75 \text{ messages}/s} = 1.0\bar{6}$

Equivalently, it can be computed as:

$l(m_0) = a_0$

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

For example:

$l(m_{i-1}) = 1.05\bar{3}s$

$l(m_i) = max(1.01\bar{3}s, 1.05\bar{3}s + \frac{1\ messsage}{75\ messages/s}) = 1.0\bar{6}$

Guaranteed Logical Delay

The *guaranteed logical delay* of a message m denotes the maximum time between the logical arrival time of m and its latest valid completion time. It results from the processing time of the messages and the competition among different sessions for resources, i.e., the waiting time of the message. If a message arrives "ahead of schedule" the actual delay is the sum of the logical delay and the time by which it arrives too early. It is then larger than the guaranteed logical delay. It can also be smaller than 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.

Workahead Messages

If a message arrives "ahead of schedule" and the resource is in an idle state, the message can be processed immediately. 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*:

$W = A \times R$

For example:

$A = 0.04s$

$0.04s \times 75\ messages/s = 3\ messages$

If a message is processed "ahead of schedule" the logical backlog is greater than the actual backlog. A message is critical if its logical arrival time has passed. Throughout the rest of the chapter the LBAP model is assumed to apply to the arrival processes at each resource. The resource must ensure that the arrival processes at the output interface obeys the LBAP parameters.

9.4 Process Management

Process management deals with the resource main processor. The capacity of this resource is specified as processor capacity. The process manager maps single processes onto resources according to a specified scheduling policy such that all processes meet their requirements. In most systems, a process under control of the process manager can adopt one of the following states:

- In the initial state, no process is assigned to the program. The process is in the idle state.
- If a process is waiting for an event, i.e., the process lacks one of the necessary resources for processing, it is in the blocked state.
- If all necessary resources are assigned to the process, it is ready to run. The process only needs the processor for the execution of the program.
- A process is running as long as the system processor is assigned to it.

The process manager is the *scheduler*. This component transfers a process into the ready-to-run state by assigning it a position in the respective queue of the dispatcher, which is the essential part of the operating system kernel. The dispatcher manages the transition from ready-to-run to run. In most operating systems, the next process to run is chosen according to a priority policy. Between processes with the same priority, the one with the longest ready time is chosen.

Today and in the near future existing operating systems must be considered to be the basis of continuous media processing on workstations and personal computers. In the next four years, there will certainly be no newly developed multimedia operating systems which will be accepted in the market; therefore, existing multitasking systems must cope with multimedia data handling. The next paragraph provides a brief description of real-time support typically available in such systems.

9.4.1 Real Time Process Management in Conventional Operating Systems: An Example

UNIX and its variants, Microsoft's Windows-NT, Apple's System 7 and IBM's OS/2^{TM}, are, and will be, the most widely installed operating systems with multitasking capabilities on personal computers (including the Power PC) and workstations. Although some of them are enhanced with special priority classes for real-time processes, this is not sufficient for multimedia applications. In [NHNW93], for example, the SVR4 UNIX scheduler which provides a real-time static priority scheduler in addition to a standard UNIX timesharing scheduler is analyzed. For this investigation three applications have been chosen to run concurrently; "typing" as an interactive application, "video" as a continuous media application and a batch program. The result was that only through trial and error a particular combination of priorities and scheduling class assignments might be found that works for a specific application set, i.e., additional features must be provided for the scheduling of multimedia data processing. To be more specific, let us have a deeper look into real-time capabilities of one of these systems, namely OS/2. On the basis of this system, the available real-time support is demonstrated.

Threads

OS/2 was designed as a time-sharing operating system without taking serious real-time applications into account. An OS/2 thread can be considered as a light-weight process: it is the dispatchable unit of execution in the operating system. A thread belongs to exactly one address space (called process in OS/2 terminology). All threads share the resources allocated by the respective address space. Each thread has its own execution stack, register values and dispatch state (either executing or ready-to-run). Each thread belongs to one of the following priority classes:

- The time-critical class is reserved for threads that require immediate attention.
- The fixed-high class is intended for applications that require good responsiveness without being time-critical.
- The regular class is used for the executing of normal tasks.
- The idle-time class contains threads with the lowest priorities. Any thread in this class is only dispatched if no thread of any other class is ready to execute.

Priorities

Within each class, 32 different priorities (0, ... , 31) exist. Through time-slicing, threads of equal priority have equal chances to execute. A context switch occurs whenever a thread tries to access an otherwise allocated resource. The thread with the highest priority is dispatched and 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. The goal at the determination of the time slice duration is to keep the number of context switches low and to get a fair and efficient schedule for the whole run-time of the system. Threads of the regular class may be subject to a dynamic rise of priority as a function of the waiting time.

Threads are preemptive, i.e., if a higher-priority thread becomes 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.

Physical Device Driver as Process Manager

In OS/2, applications with real-time requirements can run as *Physical Device Drivers (PDD)* at ring 0 (kernel mode). These PDDs can be made non-preemptable. An interrupt that occurs on a device (e.g., packets arriving at the network adapter) can be serviced from the PDD immediately. As soon as an interrupt happens on a device, the PDD gets control and does all the work to handle the interrupt. This may also include tasks which could be done by application processes running in ring 3 (user mode). The task running at ring 0 should (but must not) leave the kernel mode after 4 msec.

PDD programming is complicated mainly due to difficult testing and debugging. PDD is bound to its device; it only handles requests from its device regardless of any other events happening in the system. Different streams that request real-time scheduling can only be served by their PDDs. They run in competition with each other without the possibility of coordinating or managing them by any higher instance. This is insufficient for a multimedia system where messages can arrive at different adapter cards. Internal time-critical system activities cannot be controlled and managed through PDDs. Therefore, they cannot be considered and accounted for during scheduling decisions. The execution of real-time processes with PDDs is only a reasonable solution for a system where streams arrive at only one device and no other activity in the system has to be considered.

Operating system extensions for continuous media processing can be implemented as PDDs. In this approach, a real-time scheduler and the process management run as a PDD being activated by a high resolution timer. In principle, this is the implementation scheme of the OS/2 Multimedia Presentation Manager^{TM}, which represents the multimedia extension to OS/2.

Enhanced System Scheduler as Process Manager

Time-critical tasks can also be processed together with normal applications running in ring 3, the user level. The critical tasks can be implemented by threads running in the priority class time-critical with one of the 32 priorities within this class. Each real-time task is assigned to one thread. A thread is interrupted if another thread with higher priority requires processing. Non-time-critical applications run as threads in the regular class. They are dispatched by the operating system scheduler according to their priority.

The main advantage of this approach is the control and coordination of all time-critical threads through a higher instance, the system scheduler. This instance, running with a higher priority than all other threads, controls and coordinates threads according to the adapted scheduling algorithm and the respective processing requirements. It can observe the run-time behavior of single threads. Another entity, the resource manager, determines feasible schedules, takes care of QoS calculating and resource reservation. The competition for the CPU is regulated. The employment of an internal scheduling strategy and resource management allows the provision of processing guarantees. Yet it requires that the native scheduler be enhanced.

Meta-scheduler as Process Manager

The normally priority-driven system scheduler is used to schedule all tasks. A meta-scheduler is employed to assign priorities to real-time tasks, i.e., this meta-scheduler considers only tasks with real-time requirements. Non-time-critical tasks are processed when no time-critical task is ready for execution. In an integrated system the process management of continuous data processes will not be realized as a meta-scheduler; it rather will be part of the system process manager itself. This meta-scheduler approach is also applied in many UNIX systems.

9.4.2 Real-time Processing Requirements

Continuous media data processing must occur in exactly predetermined – usually periodic – intervals. Operations on these data recur over and over and must be completed at certain deadlines. The real-time process manager determines a schedule for the resource CPU that allows it to make reservations and to give processing guarantees. The problem is to find a feasible scheduler which schedules all time-critical continuous media tasks in a way that each of them can meet its deadlines. This must be guaranteed for all tasks in every period for the whole run-time of the system. In a multimedia system, continuous and discrete media data are processed concurrently.

For scheduling of multimedia tasks, two conflicting goals must be considered:

- An uncritical process should not suffer from starvation because time-critical processes are executed. Multimedia applications rely 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.
- On the other hand, a time-critical process must never be subject to priority inversion. The scheduler must ensure that any priority inversion (also between time-critical processes with different priorities) is avoided or reduced as much as possible.

Apart from the overhead caused by the schedulability test and the connection establishment, the costs for the scheduling of every message must be minimized. 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 operating system is part of the processing time and therefore must be added to the processing time of the real-time tasks. Thus, it is favorable to keep them low. It is particularly difficult 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 application programs. Therefore, operating systems in real-time systems cannot be viewed as detached from the application programs and vice versa.

9.4.3 Traditional Real-time Scheduling

The problem of real-time processing is widely known in computer science [HS89, Lev89, SG90, TK91]. Some real-time scheduling methods are employed in operations research. They differ from computer science real-time scheduling because they operate in a static environment where no adaptation to changes of the workload is necessary [WC87].

The goal of traditional scheduling on time-sharing computers is optimal throughput, optimal resource utilization and fair queuing. In contrast, the main goal of real-time tasks is to provide a schedule that allows all, respectively, as many time-critical processes as possible, to be processed in time, according to their deadline. The scheduling algorithm must map tasks onto resources such that all tasks meet their time requirements. Therefore, it must be possible to show, or to proove, that a scheduling algorithm applied to real-time systems fulfills the timing requirements of the task.

There are several attempts to solve real-time scheduling problems. Many of them are just variations of basic algorithms. To find the best solutions for multimedia systems, two basic algorithms are analyzed, *Earliest Deadline First Algorithm* and *Rate Monotonic Scheduling*, and their advantages and disadvantages are elaborated. In the next section, a system model is introduced, and the relevant expressions are explained.

9.4.4 Real-time Scheduling: System Model

All scheduling algorithms to be introduced are based on the following system model for the scheduling of real-time tasks. Their essential components are the resources (as discussed previously), tasks and scheduling goals.

A task is a schedulable entity of the system, and it corresponds to the notion of a thread in the previous description. In a hard real-time system, a task is characterized by its timing constraints, as well as by its resource requirements. In the considered case, only periodic tasks without precedence constraints are discussed, i.e., the processing of two tasks is mutually independent. For multimedia systems, this can be assumed without any major restriction. Synchronized data, for example, can be processed by a single process.

The time constraints of the periodic task T are characterized by the following parameters (s, e, d, p) as described in [LM80]:

- s: Starting point
- e: Processing time of T
- d: Deadline of T

- p: Period of T
- r: Rate of $T(r = \frac{1}{p})$

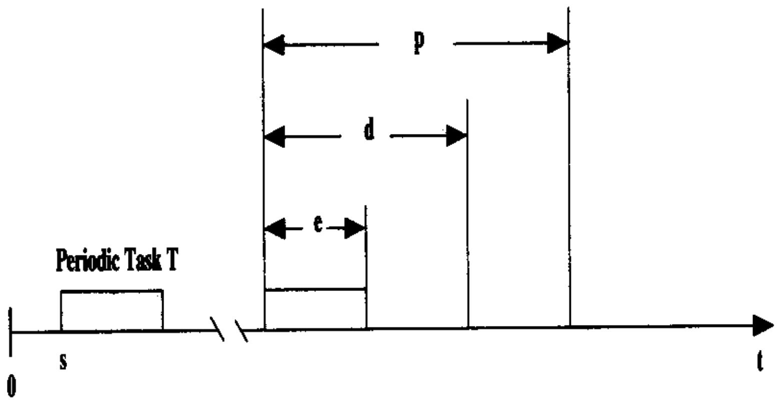

Figure 9.3: *Characterization of periodic tasks.*

whereby $0 \leq e \leq d \leq p$ (see Figure 9.3). The starting point s is the first time when the periodic task requires processing. Afterwards, it requires processing in every period with a processing time of e. At $s + (k - 1)p$, the task T is ready for k-processing. The processing of T in period k must be finished at $s + (k - 1)p + d$. For continuous media tasks, it is assumed that the deadline of the period $(k - 1)$ is the ready time of period k. This is known as *congestion avoiding deadlines*: the deadline for each message (d) coincides with the period of the respective periodic task (p).

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 in the same state later on. A non-preemptive task cannot be interrupted until it voluntarily yields the processor. Any high-priority task must wait until the low-priority task is finished. The high-priority task is then subject to priority inversion. In the following, all tasks processed on the CPU are considered as preemptive unless otherwise stated.

In a real-time system, the scheduling algorithm must determine a schedule for an exclusive, limited resource that is used by different processes concurrently such that all of them can be processed without violating any deadlines. This notion can be extended to a model with multiple resources (e.g., CPU) of the same type. It can also be extended to cover different resources such as memory and bandwidth for communication, i.e., the function of a scheduling algorithm is to determine, for a given task set, whether or not a schedule for executing the tasks on an exclusive bounded resource exists, such that the timing and resource constraints of all tasks are satisfied (planning goal). Further, it must 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 to their deadlines in every period over the whole run-time. If a scheduling algorithm guarantees a task, it ensures that the task finishes processing prior to its deadline [CSR88]. To guarantee tasks, it must be possible to check the schedulability of 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 tasks versus the total amount of processing time [LL73]:

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

9.4.5 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 state, the scheduler selects the task with the earliest deadline among the tasks that are ready and not fully processed. The requested resource is assigned to the selected task. At any arrival of a new task (according to the LBAP model), EDF must be computed immediately leading to a new order, i.e., the running task is preempted and the new task is scheduled according to its deadline. The new task is processed immediately if its deadline is earlier than that 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, deadlines and service execution times [Der74]. In this case, no guarantee about the processing of any task can be given.

EDF is an *optimal, dynamic* algorithm, i.e., it produces a valid schedule whenever one exists. A dynamic algorithm schedules every instance of each incoming task according to its specific demands. Tasks of periodic processes must be scheduled in each period again. With n tasks which have arbitrary ready-times and deadlines, the complexity is $\Theta(n^2)$.

For a dynamic algorithm like EDF, the upper bound of the processor utilization is 100%. Compared with any static priority assignment, EDF is optimal in the sense that if a set of tasks can be scheduled by any static priority assignment it also can be scheduled by EDF. With a priority-driven system scheduler, each task is assigned a priority according to its deadline. 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 might have to be adjusted.

Applying EDF to the scheduling of continuous media data on a single processor machine with priority scheduling, process priorities are likely to be rearranged quite often. A priority is assigned to each task ready for processing according to its deadline. Common systems usually provide only a restricted number of priorities. If the computed priority of a new process is not available, the priorities of other processes must be rearranged until the required priority is free. In the worst case, the priorities of all processes must be rearranged. This may cause considerable overhead. The EDF scheduling algorithm itself makes no use of the previously known occurrence of periodic tasks.

EDF is used by different models as a basic algorithm. An extension of EDF is the *Time-Driven Scheduler* (TDS). Tasks are scheduled according to their deadlines. Further, the TDS is able to handle overload situations. If an overload situation occurs the scheduler aborts tasks which cannot meet their deadlines anymore. If there is still an overload situation, the scheduler removes tasks which have a low "value density." The value density corresponds to the importance of a task for the system.

In [LLSY91] another priority-driven EDF scheduling algorithm is introduced. Here, every task is divided into 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 tasks is said to be schedulable 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 continuous media data, the method can be used in combination with the encoding of data according to their importance. Take, for example, a single uncompressed picture in a bitmap format. Each pixel of this monochrome picture is coded with 16 bits. The processing of the eight most significant bits is mandatory, whereas the processing of the least-significant bits can be considered optional. With this method, more processes can be scheduled. In an overload situation, the optional parts are aborted. This implementation leads to decreased quality by media scaling. During QoS requirement specification, the tasks were accepted or informed that scaling may occur. In such a case, scaling QoS parameters can be introduced which reflect the respective implementation. Therefore, this approach avoids errors and improves system performance at the expense of media quality.

9.4.6 Rate Monotonic Algorithm

The *rate monotonic scheduling* principle was introduced by Liu and Layland in 1973 [LL73]. It is an optimal, static, priority-driven algorithm for preemptive, periodic jobs. Optimal in this context means that there is no other static algorithm that is able to schedule a task set which cannot 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. No further scheduling is required. The following five assumptions are necessary prerequisites to apply the rate monotonic algorithm:

1. The requests for all tasks with deadlines are periodic, i.e., have constant intervals between consecutive requests.

2. The processing of a single task must be finished before the next task of the same data stream becomes ready for execution. Deadlines consist of runability constraints only, i.e., each task must be completed before the next request occurs.

3. All tasks are independent. This means that the requests for a certain task do not depend on the initiation or completion of requests for any other task.

4. Run-time for each request of a task is constant. Run-time denotes the maximum 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. Typically, they initiate periodic tasks or are tasks for failure recovery. They usually displace periodic tasks.

Further work has shown that not all of these assumptions are mandatory to employ the rate monotonic algorithm [LSST91, SKG91]. Static priorities are assigned to tasks, once at the connection set-up phase, according to their request rates. The priority corresponds to the importance of a task relative to other tasks. Tasks with higher request rates will have higher priorities. The task with the shortest period gets the highest priority and the task with the longest period gets the lowest priority.

The rate monotonic algorithm is a simple method to schedule time-critical, periodic tasks on the respective resource. A task will always meet its deadline, if this can be proven to be true for the longest response time. The *response time* is the time span between the request and the end of processing the task. This time span is maximal when all processes with a higher priority request to be processed at the same time. This case is known as the *critical instant* (see Figure 9.4). In this figure, the priority of a is, according to the rate monotonic algorithm, higher than b, and b is higher than c. The *critical time zone* is the time interval between the critical instant and the completion of a task.

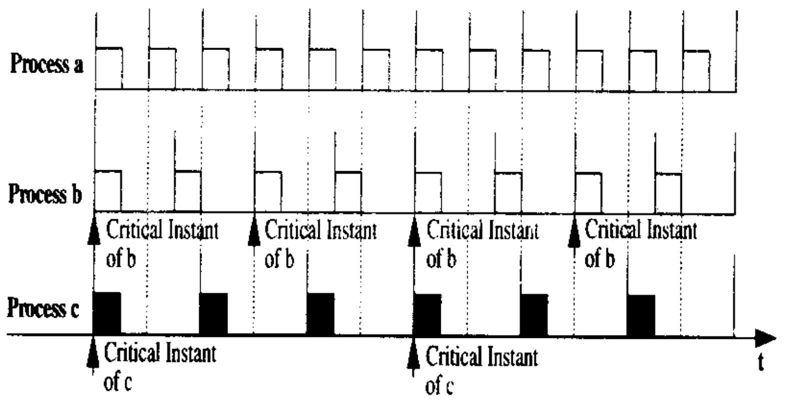

Figure 9.4: *An example of critical instants.*

9.4.7 EDF and Rate Monotonic: Context switches

Consider an audio and a video stream scheduled according to the rate monotonic algorithm. Let the audio stream have a rate of 1/75 s/sample and the video stream a rate of 1/25 s/frame. The priority assigned to the audio stream is then higher than the priority assigned to the video stream. The arrival of 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 that requests processing at the critical instant before its deadline, the processing of all video messages to their deadlines is ensured, thus a feasible schedule exists.

If more than one stream is processed concurrently in a system, it is very likely that there might be more context switches with a scheduler using the rate monotonic algorithm than one using EDF. Figure 9.5 shows an example.

9.4.8 EDF and Rate Monotonic: Processor Utilizations

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. There are two issues to be considered:

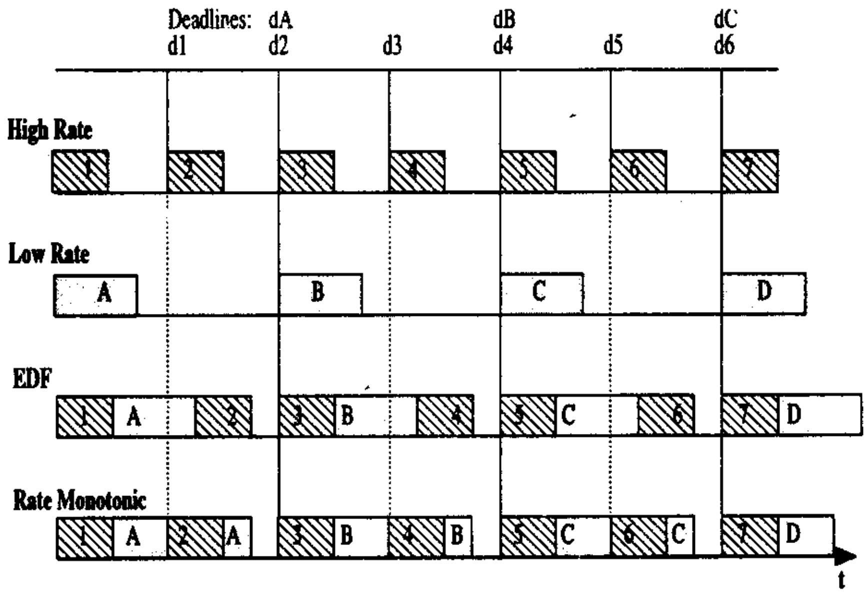

Figure 9.5: *Rate monotonic versus EDF: context switches in preemptive systems.*

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 $t(j)$; $j \in (1, ..., n)$ 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.

Following this assumption, [LL73] gives 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 will always meet its deadline if:

$$U(m) = m \times \left(2^{\frac{1}{m}} - 1\right) \geq \frac{e_1}{p_1} + \ldots + \frac{e_m}{p_m}$$

According to [LS86] and [LL73], for large m, the least upper bound of the processor utilization is $U = ln2$. Hence, it is sufficient to check if the processor utilization is less than or equal to the given upper bound to find out if a task set is schedulable or not. Most existing systems check this by simply comparing the processor utilization with the value of $ln2$.

With EDF, a processor utilization of 100% can be achieved because all tasks are scheduled dynamically according to their deadlines. Figure 9.6 shows an example where the CPU can be utilized to 100% with EDF, but where rate monotonic scheduling fails.

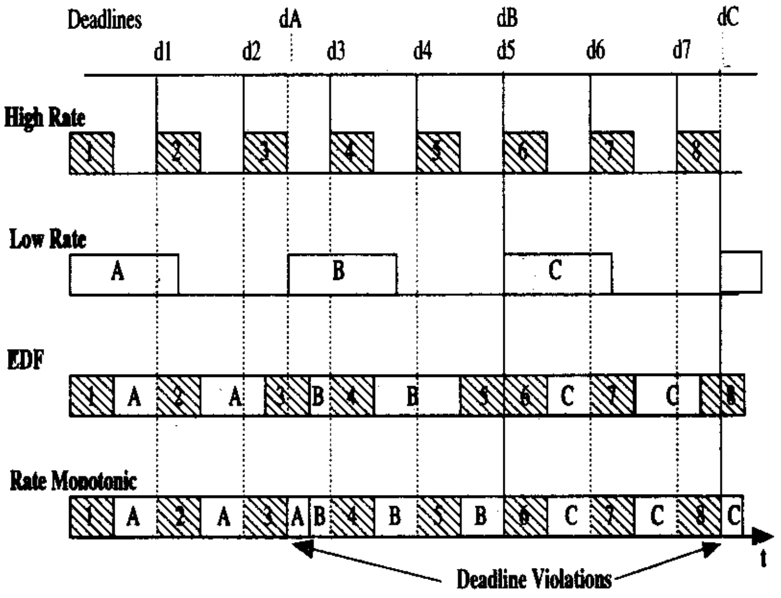

Figure 9.6: *Rate monotonic versus EDF: processor utilization.*

The problem of underutilizing the processor is aggregated by the fact that, 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%. [SSL89] mentions an achieved utilization bound for the Nowy's Inertial

Navigation System of 88%.

As described above, a static algorithm schedules a process once at the beginning of processing. Single tasks are not explicitly scheduled afterwards. A dynamic algorithm schedules every incoming task according to its specific demands. Since the rate monotonic algorithm is an optimal static algorithm, no other static algorithm can achieve a higher processor utilization.

9.4.9 Extensions to Rate Monotonic Scheduling

There are several extensions to this algorithm. One of them divides a task into 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. The mandatory part is scheduled according to the rate monotonic algorithm. For the scheduling of the optional part, other, different policies are suggested [CL88, LLN87, CL89].

In some systems there are aperiodic tasks next to periodic ones. 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 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 [Her90]. If the stream is not continuous, we can apply a sporadic server to respond to aperiodic requests. The server is provided with a computation budget. This budget is refreshed t units of time after it has been exhausted. Earlier refreshing is also possible. The budget represents the computation time reserved for aperiodic tasks. 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 server's assigned priority. The sporadic server is especially suitable for events that occur rarely, but must be handled quickly (e.g., a telepointer in a CSCW application) [SG90, SSL89, Spr90].

The rate monotonic algorithm is, for example, applied in real-time systems and real-time operating systems by NASA and the European Space Agency [SR89]. It is particularly suitable for continuous media data processing because it makes optimal use of their periodicity. Since it is a static algorithm, there is nearly no rearrangement of priorities and hence – in contrast to EDF – no scheduling overhead to determine the next task with the highest priority. Problems emerge with data streams which have no constant processing time per message, as specified in MPEG-2 (e.g., a compressed video stream where one of five pictures is a full picture and all others are updates of a reference picture). The simplest solution is to schedule these 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 non-time-critical tasks. In multimedia systems, for example, this is the processing of discrete media.

9.4.10 Other Approaches for In-Time Scheduling

Apart from the two methods previously discussed, further scheduling algorithms have been evaluated regarding their suitability for the processing of continuous media data. In the following paragraphs, the most significant approaches are briefly described and the reasons for their non-suitability, compared to EDF and rate-monotonic, are enumerated.

Least Laxity First (LLF). The task with the shortest remaining laxity is scheduled first [CW90, LS86]. 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 an *optimal, dynamic* algorithm for *exclusive resources.* Furthermore, it is an optimal algorithm for multiple resources if the ready-times of the real-time tasks are the same. The *laxity* is a function of a deadline, the processing time and the current time. Thereby, the processing time cannot be exactly specified in advance. When calculating the laxity, the worst case is assumed. Therefore, the determination of the laxity is inexact. The laxity of waiting processes dynamically changes over time.

During the run-time of a task, another task may get a lower laxity. This task must then preempt the running task. Consequently, tasks can preempt each other several times without dispatching a new task. This may cause numerous context switches. At each scheduling point (when a process becomes ready-to-run or at the end of a time slice), the laxity of each task must 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. Future multimedia systems might be multiprocessor systems; here, LLF might be of advantage.

Deadline Monotone Algorithm. If the deadlines of tasks are less than their period ($d_i < p_i$), the prerequisites of the rate monotonic algorithm are violated. In this case, a fixed priority assignment according to the deadlines of the tasks is optimal. A task T_i gets a higher priority than a task T_j if $d_i < d_j$. No effective schedulability test for the deadline monotone algorithm exists. To determine the schedulability of a task set, each task must be checked if it meets its deadline in the worst case. In this case, all tasks require execution to their critical instant [LW82, LSST91]. Tasks with a deadline shorter than their period, for example, arise during the measurements of temperature or pressure in control systems. In multimedia systems, deadlines equal to period lengths can be assumed.

Shortest Job First (SJF). The task with the shortest remaining computation time is chosen for execution [CW90, Fre82]. This algorithm guarantees that as many tasks as possible meet their deadlines under an overload situation if all of them have the same deadline. In multimedia systems where the resource management allows overload situations this might be a suitable algorithm.

Apart from the most important real-time scheduling methods discussed above, others might be employed for the processing of continuous media data (an on-line scheduler for tasks with unknown ready-times is introduced by [HL88]; in [HS89], a real-time monitoring system is presented where all necessary data to calculate an optimal schedule are available). In most multimedia systems with preemptive tasks, the rate monotonic algorithm in different variations is employed. So far, no other scheduling technique has been proven to be at least as suitable for multimedia data handling as the EDF and rate monotonic approaches.

9.4.11 Preemptive versus Non-preemptive Task Scheduling

Real-time tasks can be distinguished into *preemptive* and *non-preemptive* tasks. If a task is *non-preemptive*, it is processed and not interrupted until it is finished or requires further resources. The contrary of non-preemptive tasks are *preemptive tasks*. The processing of a preemptive task is interrupted immediately by a request of any higher-priority task. In most cases where tasks are treated as non-preemptive, the arrival times, processing times and deadlines are arbitrary and unknown to the scheduler until the task actually arrives. The best algorithm is the one which maximizes the number of completed tasks. In this case, it is not possible to provide any processing guarantees or to do resource management.

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. Another reason is that for some non-preemptive task sets, no feasible schedule can be found; whereas for preemptive scheduling, it is possible. Figure 9.7 shows an example where the scheduling of preemptive tasks is possible, but non-preemptive tasks cannot be scheduled.

Liu and Layland [LL73] 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, the preemptiveness of tasks is a necessary prerequisite to check their schedulability.

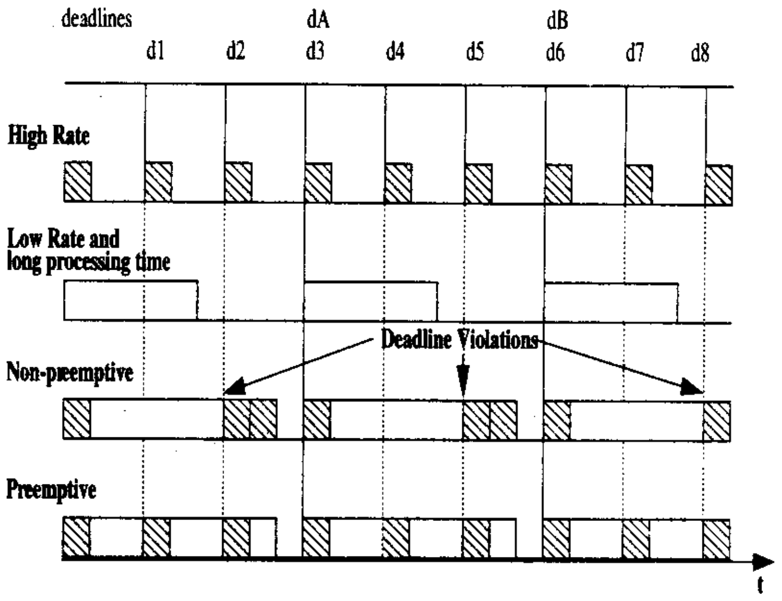

Figure 9.7: *Preemptive vs. non-preemptive scheduling.*

The first schedulability test for the scheduling of *non-preemptive tasks* was introduced by Nagarajan and Vogt in [NV92]. Assume, without loss of generality, that task m has highest priority and task 1 the lowest. They proove that a set of m periodic streams with periods p_i, deadlines d_i, processing times e_i and $d_i \leq p_i \forall i \in (1, ..., m)$ is schedulable with the non-preemptive fixed priority scheme if:

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

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

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 its processing time e_i and the processing time of any higher-priority task that requires execution during that time interval, plus the longest processing time of all lower- and higher-priority tasks $max_{(1 \leq j \leq m)}$ e_j that might be serviced at the arrival of task t_i (Figure 9.8).

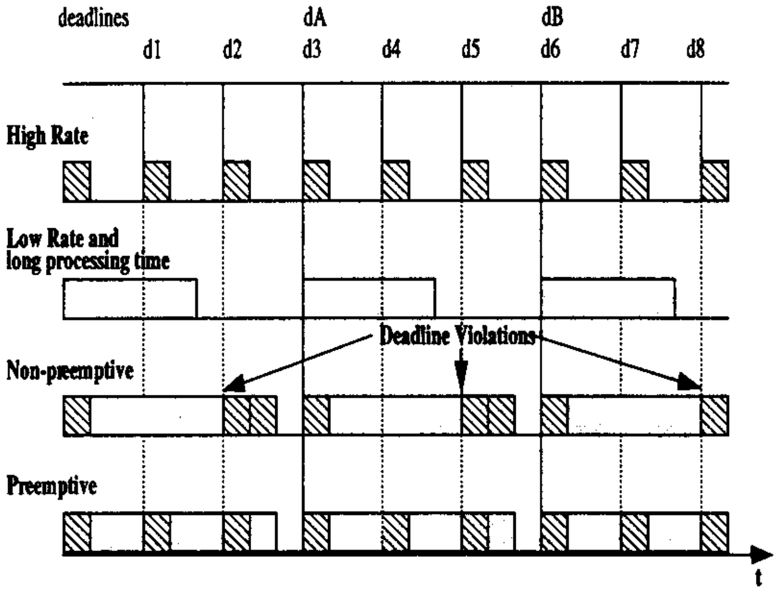

Figure 9.8: *Deadline requirements for non-preemptive scheduling.*

The schedulability test is an extension of Liu's and Layland's. Given m periodic tasks with periods p_i and the same processing time E per message, let $d_i = p_i + E$ be the deadline for task t_i. Then, the streams are schedulable:

- With the non-preemptive rate monotonic scheme with:

$$\sum \frac{1}{p_i} \times E \leq \ln 2$$

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

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

Consequently, non-preemptive continuous media tasks can also be scheduled. However, the scheduling of non-preemptive tasks is less favorable because the number of schedulable task sets is smaller compared to preemptive tasks.

9.4.12 Scheduling of Continuous Media Tasks: Prototype Operating Systems

Most multimedia operating systems apply one of the previously discussed methods. In some systems, the scheduler is replaced by a real-time scheduler. Therefore, these systems can be viewed as new operating systems. They are usually not compatible with existing systems and applications. Other systems apply a meta-scheduler based on an existing process manager. Only these systems will have a commercial impact in the short and medium terms because they allow existing applications to run.

ARTS

The *Advanced Real Time Technology Operating System* is a real-time operating system for a distributed environment with one real-time process manager. It was developed on SUN3 workstations and connected with a real-time network based on the IEEE.802.5 Token Ring and Ethernet by the Computer Science department of Carnegie Mellon University. To solve the scheduling problems, the Time-Driven Scheduler (TDS) with a priority inheritance protocol was adopted. This priority inheritance protocol was used to prevent unbounded priority inversion among communication tasks. Tasks with hard deadlines are scheduled according to the rate monotonic algorithm. The system is also provided with other scheduling methods for experimental reasons [MT90].

YARTOS

Yet Another Real Time Operating System was developed at the University of North Carolina at Chapel Hill as an operating system kernel to support conferencing applications [JSP91]. An optimal, preemptive algorithm to schedule tasks on a single processor was developed. The scheduling algorithm results from the integration of a synchronization scheme to access shared resources with the EDF algorithm. Here, a task has two notions of deadline, one for the initial acquisition of the processor, and one for the execution of operations on resources. To avoid priority inversion, tasks are provided with separate deadlines for performing operations on shared resources. It is guaranteed that no shared resource is accessed simultaneously by more than one task. Further, a shared resource is not occupied by a single task longer than absolutely necessary.

Split-level Scheduling

The *split-level scheduler* was developed within the DASH project at the University of California at Berkeley. Its main goal was to provide a better support for multimedia applications [And93]. The applied scheduling policy is deadline/workahead scheduling. The LBAP-model is used to describe 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 do 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.

Three Class Scheduler

This scheduler was developed as part of a video-on-demand file servicer at DEC, Littleton. The design of the scheduler is based on a combination of weighted round-robin and rate monotonic scheduling [RVG+93]. Three classes of schedulable tasks are supported. The isochronous class with the highest priority applies the rate monotonic algorithm, the real-time and the general-purpose classes use the weighted round-robin scheme. A general-purpose task is preemptive and runs with a low priority. The real-time class is suitable for tasks that require guaranteed throughput and bounded delay. The isochronous class supports real-time periodic tasks that require performance guarantees for throughput, bounded latency and low jitter. Real-time and isochronous tasks can only be preempted in "preemption windows."

The scheduler executes tasks from a ready queue in which all isochronous tasks are arranged according to their priority. At the arrival of a task, the scheduler determines whether the currently running task must be preempted. General-purpose tasks are immediately preempted, real-time tasks are preempted in the next preemption window and isochronous tasks are preempted in the next preemption window if their priority is lower than the one of the new task. Whenever the queue is empty, the scheduler alternates between the real-time and general-purpose classes using a weighted round-robin scheme.

Meta-scheduler

To support real-time processing of continuous media, a meta-scheduler for the operating systems AIX^{TM}[WBV92] and OS/2 [MSS92] was developed at the European Networking Center of IBM in Heidelberg. Both are based on the LBAP model. According to the rate monotonic algorithm, rates are mapped onto system priorities.

Experience with the Meta-scheduler Approach

In this paragraph, the employment of the OS/2 meta-scheduler is discussed [MSS92]. Experience shows the limits of this approach. For example, each process in the system is able to run with a priority initially intended for real-time tasks. These processes are not scheduled by the resource manager and therefore violate the calculated schedule. A malicious process can block the whole system by simply running with the highest priority without giving up control.

The management of scheduling algorithms requires exact time measurement. In OS/2, for example, it is not possible to measure the exact time a thread is using the CPU. Any measurement of the processing time includes interrupts. If a process is interrupted by another process, it also includes the time needed for the context switch. The granularity of the OS/2 system timers is insufficient for the processing of real-time tasks. Hence, the rate control is inaccurate because it is determined by the granularity of the system timer.

To achieve full real-time capabilities, at least the native scheduler of the operating system would must be extended. The operating system should be enhanced by a class of fast, non-preemptive threads and the ability to mask interrupts for a short period of time. Priorities in this thread class should only be assigned to threads that are already registered by the resource manager. This class should be reserved exclusively for selected threads and monitored by a system component with extensive control facilities. Performance enhancement of the scheduler itself, incorporating some mechanisms of real-time scheduling like EDF, would be another solution. The operating system should, in any case, provide a time measurement tool that allows the measurement of pure CPU time and a timer with a finer granularity. This may be achieved through a timer chip.

9.5 File Systems

The *file system* is said to be the most visible part of an operating system. Most programs write or read files. Their program code, as well as user data, are stored in files. The organization of the file system is an important factor for the usability and convenience of the operating system. A file is a sequence of information held as a unit for storage and use in a computer system [Kra88].

Files are stored in secondary storage, so they can be used by different applications. The life-span of files is usually longer than the execution of a program. In traditional file systems, the information types stored in files are sources, objects, libraries and executables of programs, numeric data, text, payroll records, etc. [PS83]. In multimedia systems, the stored information also covers digitized video and audio with their related real-time "read" and "write" demands. Therefore, additional requirements in the design and implementation of file systems must be considered.

The file system provides access and control functions for the storage and retrieval of files. From the user's viewpoint, it is important how the file system allows file organization and structure. The internals, which are more important in our context, i.e., the organization of the file system, deal with the representation of information in files, their structure and organization in secondary storage. Because of its importance for multimedia, disk scheduling is also presented in this context.

The next section starts with a brief characterization of traditional file systems and disk scheduling algorithms. Subsequently, different approaches to organize multimedia files and disk scheduling algorithms for the use in multimedia systems are discussed.

9.5.1 Traditional File Systems

The two main goals of traditional files systems are: (1) to provide a comfortable interface for file access to the user, and (2) to make efficient use of storage media. Whereas the first goal is still an area of interest for research (e.g., indexing for file systems [Sal91] and intelligent file systems for the content-based associative access to file system data [GO91]), the structure, organization and access of data stored on disk have been extensively discussed and investigated over the last decades. To understand the specific multimedia developments in this area, this section gives a brief overview on files, file system organizations and file access mechanisms. Later, disk scheduling algorithms for file retrieval are discussed.

File Structure

We commonly distinguish between two methods of file organization. In sequential storage, each file is organized as a simple sequence of bytes or records. Files are stored consecutively on the secondary storage media as shown in Figure 9.9 . They

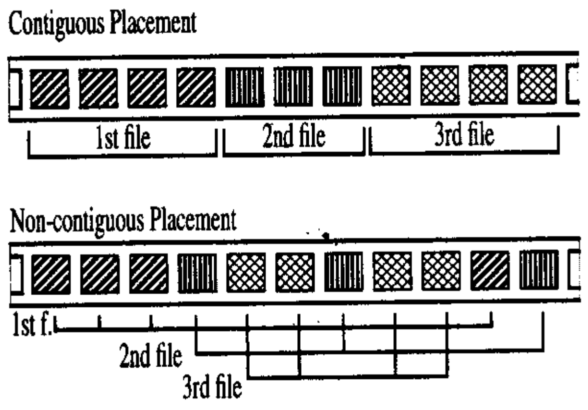

Figure 9.9: Contiguous and non-contiguous storage.

are separated from each other by a well defined "end of file" bit pattern, character or character sequence. A file descriptor is usually placed at the beginning of the file and is, in some systems, repeated at the end of the file. Sequential storage is the only possible way to organize the storage on tape, but it can also be used on disks. The main advantage is its efficiency for sequential access, as well as for direct access [Kra88]. Disk access time for reading and writing is minimized.

Additionally, for further improvement of performance with caching, the file can be read ahead of the user program [Jan85]. In systems where file creation, deletion and size modification ocrur frequently, sequential storage has major disadvantages. Secondary storage is split and fragmented, through creation and deletion operations, and files cannot be extended without copying the whole files into a larger space. The files may be copied such that all files are adjacently located, i.e., without any "holes" between them.

In non-sequential storage, the data items are stored in a non-contiguous order. There exist mainly two approaches:

- One way is to use linked blocks, where physical blocks containing consecutive logical locations are linked using pointers. The file descriptor must contain the number of blocks occupied by the file, the pointer to the first block and it may also have the pointer to the last block. A serious disadvantage of this method is the cost of the implementation for random access because all prior data must be read. In MS-DOS, a similar method is applied. A *File Allocation Table (FAT)* is associated with each disk. One entry in the table represents one disk block. The directory entry of each file holds the block number of the first block. The number in the slot of an entry refers to the next block of a file. The slot of the last block of a file contains an end-of-file mark [Tan87].

- Another approach is to store block information in mapping tables. Each file is associated with a table where, apart from the block numbers, information like owner, file size, creation time, last access time, etc., are stored. Those tables usually have a fixed size, which means that the number of block references is bounded. Files with more blocks are referenced indirectly by additional tables assigned to the files. In UNIX, a small table (on disk) called an i-node is associated with each file (see Figure 9.10). The indexed sequential approach is an example for multi-level mapping; here, logical and physical organization

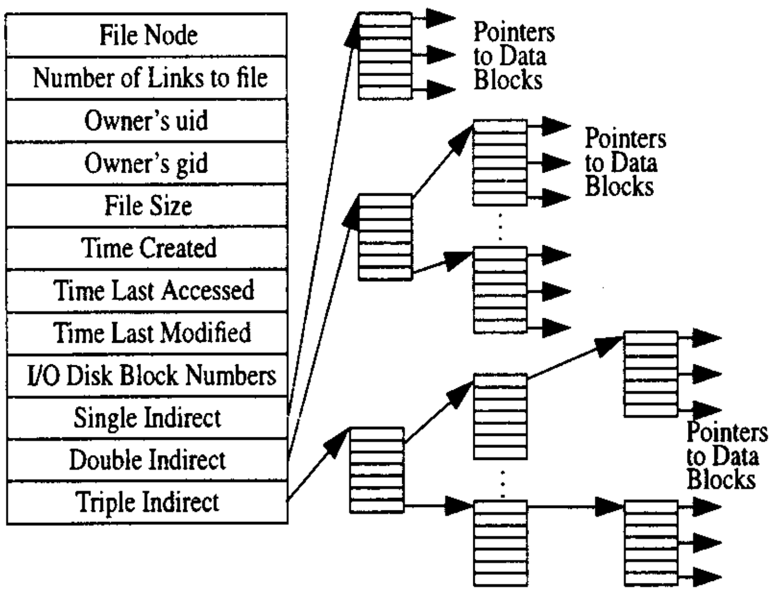

Figure 9.10: *The UNIX i-node [Tane87].*

are not clearly separated [Kra88].

Directory Structure

Files are usually organized in *directories*. Most of today's operating systems provide tree-structured directories where the user can organize the files according to his/her personal needs. In multimedia systems, it is important to organize the files in a way that allows easy, fast, and contiguous data access.

Disk Management

Disk access is a slow and costly transaction. In traditional systems, a common technique to reduce disk access are *block caches*. Using a block cache, blocks are kept in memory because it is expected that future read or write operations access these data again. Thus, performance is enhanced due to shorter access time. Another way to enhance performance is to reduce disk arm motion. Blocks that are likely to be accessed in sequence are *placed together on one cylinder.* To refine this method, rotational positioning can be taken into account. Consecutive blocks are placed on the same cylinder, but in an interleaved way as shown in Figure 9.11. Another

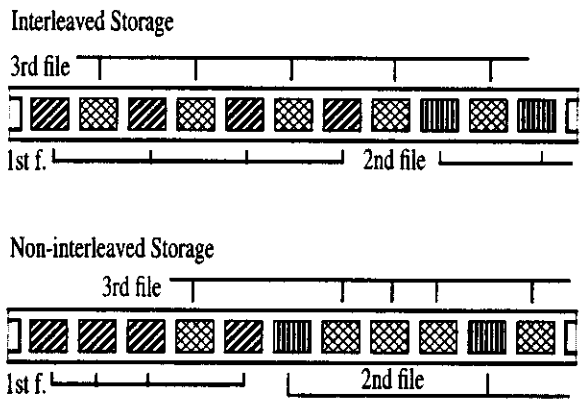

Figure 9.11: *Interleaved and non-interleaved storage.*

important issue is the placement of the mapping tables (e.g., I-nodes in UNIX) on the disk. If they are placed near the beginning of the disk, the distance between them and the blocks will be, on average, half the number of cylinders. To improve this, they can be placed in the middle of the disk. Hence, the average seek time is roughly reduced by a factor of two. In the same way, consecutive blocks should be placed on the same cylinder. The use of the same cylinder for the storage of mapping tables and referred blocks also improves performance.

Disk Scheduling

Whereas strictly sequential storage devices (e.g., tapes) do not have a scheduling problem, for random access storage devices, every file operation may require movements of the read/write head. This operation, known as "to seek," is very time consuming, i.e., a *seek time* in the order of 250 ms for CDs is still state-of-the-art. The actual time to read or write a disk block is determined by:

- The seek time (the time required for the movement of the read/write head).

- The latency time or rotational delay (the time during which the transfer cannot proceed until the right block or sector rotates under the read/write head).
- The actual data transfer time needed for the data to copy from disk into main memory.

Usually the seek time is the largest factor of the actual transfer time. Most systems try to keep the cost of seeking low by applying special algorithms to the scheduling of disk read/write operations. The access of the storage device is a problem greatly influenced by the file allocation method. For instance, a program reading a contiguously allocated file generates requests which are located close together on a disk. Thus head movement is limited. Linked or indexed files with blocks, which are widely scattered, cause many head movements. In multi-programming systems, where the disk queue may often be non-empty, fairness is also a criterion for scheduling. Most systems apply one of the following scheduling algorithms:

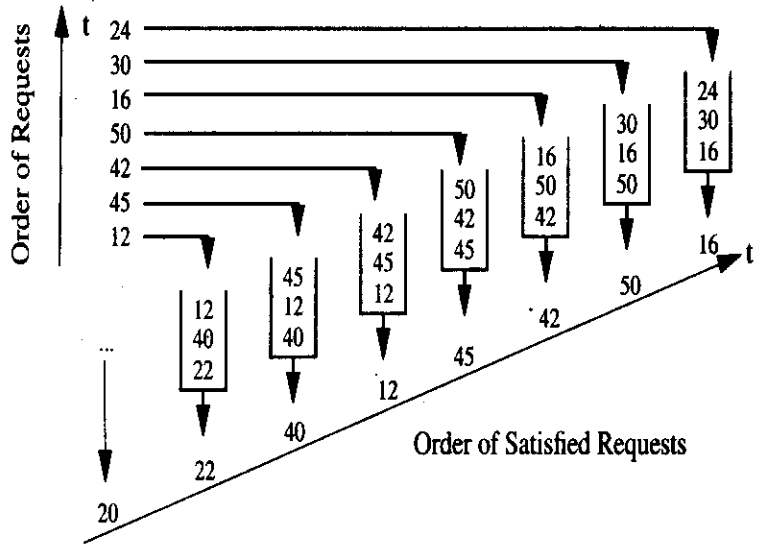

Figure 9.12: *FCFS disk scheduling.*

- *First-Come-First-Served (FCFS)*

With this algorithm, the disk driver accepts requests one-at-a-time and serves them in incoming order. This is easy to program and an intrinsically fair algorithm. However, it is not optimal with respect to head movement because it does not consider the location of the other queued requests. This results in a high average seek time. Figure 9.12 shows an example of the application of FCFS to a request of three queued blocks.

- *Shortest-Seek-Time First (SSTF)*

  At every point in time, when a data transfer is requested, SSTF selects among all requests the one with the minimum seek time from the current head position. Therefore, the head is moved to the closest track in the request queue. This algorithm was developed to minimize seek time and it is in this sense optimal. SSTF is a modification of Shortest Job First (SJF), and like SJF, it may cause starvation of some requests. Request targets in the middle of the disk will get immediate service at the expense of requests in the innermost and outermost disk areas. Figure 9.13 demonstrates the operation of the SSTF algorithm.

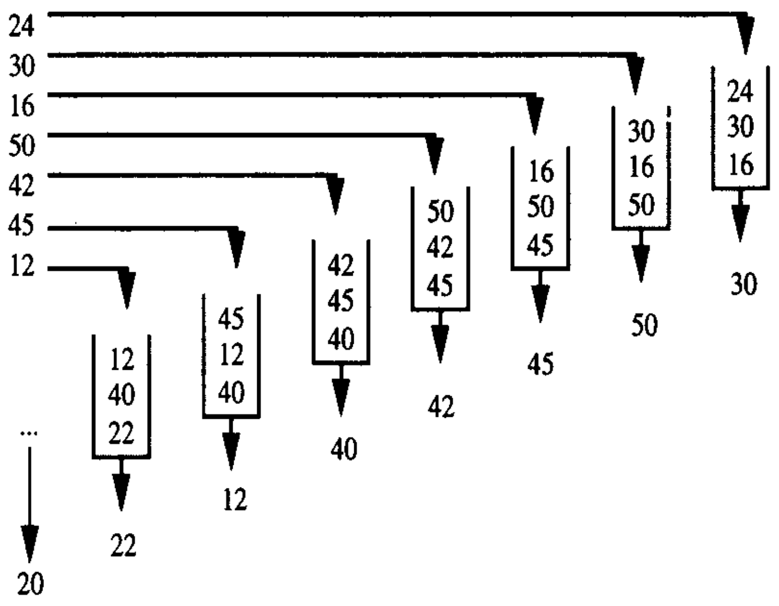

Figure 9.13: *SSTF disk scheduling.*

- *SCAN*

  Like SSTF, SCAN orders requests to minimize seek time. In contrast to SSTF, it takes the direction of the current disk movement into account. It first serves all requests in one direction until it does not have any requests in this direction anymore. The head movement is then reversed and service is continued. SCAN provides a very good seek time because the edge tracks get better service times. Note that middle tracks still get a better service then edge tracks. When the head movement is reversed, it first serves tracks that have recently been serviced, where the heaviest density of requests, assuming a uniform distribution, is at the other end of the disk. Figure 9.14 shows an example of the SCAN algorithm.

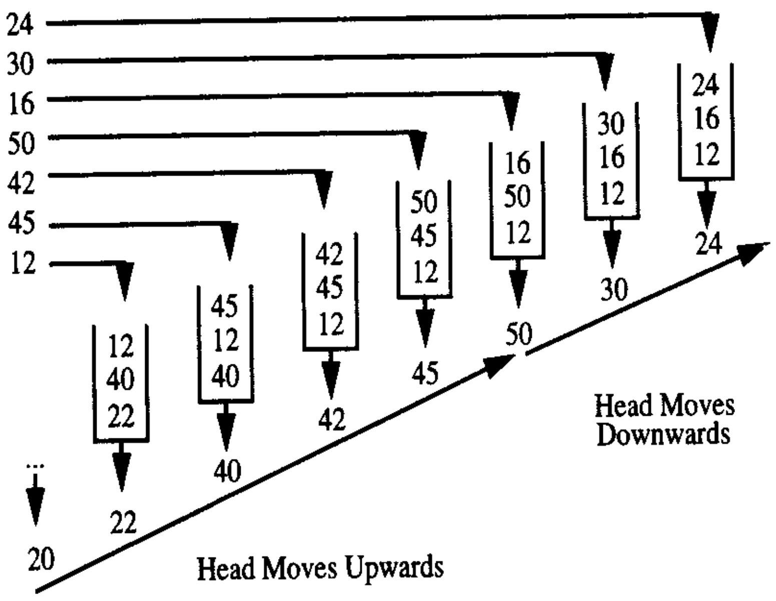

Figure 9.14: *SCAN disk scheduling.*

- *C-SCAN*

  C-SCAN also moves the head in one direction, but it offers fairer service with more uniform waiting times. It does not alter the direction, as in SCAN. Instead, it scans in cycles, always increasing or decreasing, with one idle head movement from one edge to the other between two consecutive scans. The performance of C-SCAN is somewhat less than SCAN. Figure 9.15 shows the operation of the C-SCAN algorithm.

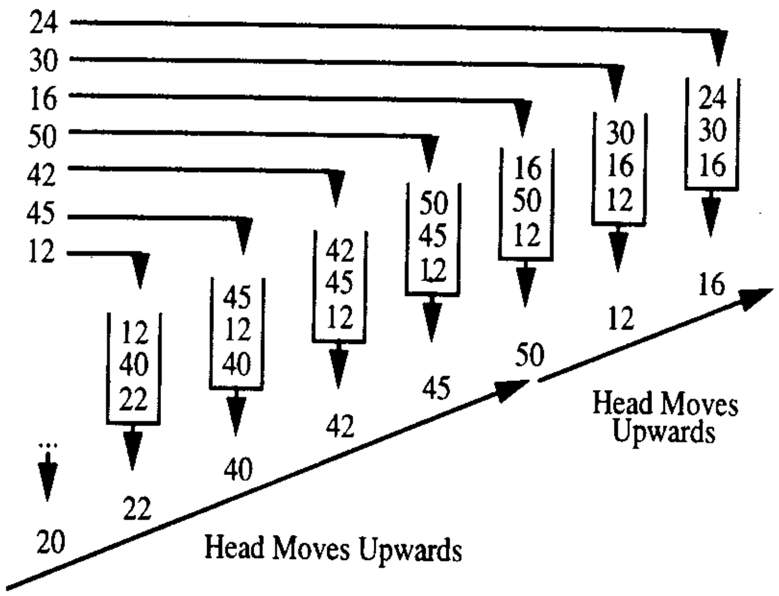

Figure 9.15: *C-SCAN disk scheduling.*

Traditional file systems are not designed for employment in multimedia systems. They do not, for example, consider requirements like real-time which are important to the retrieval of stored audio and video. To serve these requirements, new policies in the structure and organization of files, and in the retrieval of data from the disk, must be applied. The next section outlines the most important developments in this area.

9.5.2 Multimedia File Systems

Compared to the increased performance of processors and networks, storage devices have become only marginally faster [Mul91]. The effect of this increasing speed mismatch is the search for new storage structures, and storage and retrieval mechanisms with respect to the file system. Continuous media data are different from discrete data in:

- *Real Time Characteristics*

  As mentioned previously, the retrieval, computation and presentation of continuous media is time-dependent. The data must be presented (read) before a well-defined deadline with small jitter only. Thus, algorithms for the storage and retrieval of such data must consider time constraints, and additional buffers to smooth the data stream must be provided.

- *File Size*

  Compared to text and graphics, video and audio have very large storage space requirements. Since the file system has to store information ranging from small, unstructured units like text files to large, highly structured data units like video and associated audio, it must organize the data on disk in a way that efficiently uses the limited storage. For example, the storage requirements of uncompressed CD-quality stereo audio are 1.4 Mbits/s; low but acceptable quality compressed video still requires about 1Mbit/s using, e.g., MPEG-1.

- *Multiple Data Streams*

  A multimedia system must support different media at one time. It does not only have to ensure that all of them get a sufficient share of the resources, it also must consider tight relations between different streams arriving from different sources. The retrieval of a movie, for example, requires the processing and synchronization of audio and video.

There are different ways to support continuous media in file systems. Basically there are two approaches. With the first approach, the organization of files on disk remains as is. The necessary real-time support is provided through special disk scheduling algorithms and sufficient buffer to avoid jitter. In the second approach, the organization of audio and video files on disk is optimized for their use in multimedia systems. Scheduling of multiple data streams still remains an issue of research.

In this section, the different approaches are discussed and examples of existing prototypes are introduced. First, a brief introduction of the different storage devices employed in multimedia systems is given. Then, the organization of files on disks is discussed. Subsequently, different disk scheduling algorithms for the retrieval of continuous media are introduced.

Storage Devices

The storage subsystem is a major component of any information system. Due to the immense storage space requirements of continuous media, conventional magnetic storage devices are often not sufficient. Tapes, still in use in some traditional systems, are inadequate for multimedia systems because they cannot provide independent accessible streams, and random access is slow and expensive.

Apart from common disks with large capacity, some multimedia applications, such as kiosk systems, use CD-ROMs to store data. In general, disks can be characterized in two different ways:

- First, how information is stored on them. There are re-writeable (magnetic and optical) disks, write-once (WORM) disks and read-only disks like CD-ROMs.

- The second distinctive feature is the method of recording. It is distinguished between magnetic and optical disks. The main differences between them are the access time and track capacity. The seek time on magnetic disks is typically above 10 ms, whereas on optical disks, 200 ms is a common lower bound. Magnetic disks have a constant rotation speed (Constant Angular Velocity, CAV). Thus, while the density varies, the storage capacity is the same on inner and outer tracks. Optical disks have varying rotation speed (Constant Linear Velocity, CLV) and hence, the storage density is the same on the whole disk.

Therefore, different algorithms for magnetic and optical disks are necessary. File systems on CD-ROMs are defined in ISO 9660. They are considered to be closely related to CD-ROMs and CD-ROM-XA. Very few variations are possible. Hence, we will focus the description on algorithms applicable to magnetic storage devices.

File Structure and Placement on Disk

In conventional file systems, the main goal of the file organization is to make efficient use of the storage capacity (i.e., to reduce internal and external fragmentation) and to allow arbitrary deletion and extension of files. In multimedia systems, the main goal is to *provide a constant and timely retrieval of data.* Internal fragmentation occurs when blocks of data are not entirely filled. On average, the last block of a file is only half utilized. The use of large blocks leads to a larger waste of storage due to this internal fragmentation. External fragmentation mainly occurs when files are stored in a contiguous way. After the deletion of a file, the gap can only be filled by a file with the same or a smaller size. Therefore, there are usually small fractions between two files that are not used, storage space for continuous media is wasted.

As mentioned above, the goals for multimedia file systems can be achieved through providing enough buffer for each data stream and the employment of disk scheduling algorithms, especially optimized for real-time storage and retrieval of data. The advantage of this approach (where data blocks of single files are scattered) is flexibility. External fragmentation is avoided and the same data can be used by several streams (via references). Even using only one stream might be of advantage; for instance, it is possible to access one block twice, e.g., when a phrase in a sonata is repeated. However, due to the large seek operations during playback, even with optimized disk scheduling, large buffers must be provided to smooth jitter at the data retrieval phase. Therefore, there are also long initial delays at the retrieval of continuous media.

Another problem in this context is the *restricted transfer rate.* With upcoming disk arrays, which might have 100 and more parallel heads, the projected seek and latency times of less than 10 ms and a block size of 4 Kbytes at a transfer rate of 0.32 Gigabit/s will be achieved. But this is, for example, not sufficient for the simultaneous retrieval of four or more production-level MPEG-2 videos compressed in HDTV-quality that may require transfer rates of up to 100 Mbit/s. [Ste94a].

Approaches which use specific disk layout take the specialized nature of continuous media data into account to minimize the cost of retrieving and storing streams. The much greater size of continuous media files and the fact that they will usually be retrieved sequentially because of the nature of the operation performed on them (such as play, pause, fast forward, etc.) are reasons for an optimization of the disk layout. Our own application-related experience has shown that continuous media streams predominantly belong to the write-once-read-many nature, and streams that are recorded at the same time are likely to be played back at the same time (e.g., audio and video of a movie) [LS93]. Hence, it seems to be reasonable to store continuous media data in large data blocks contiguously on disk. Files that are likely to be retrieved together are grouped together on the disk. Thus, interference due to concurrent access of these files is minimized. With such a disk layout, the buffer requirements and seek times decrease.

The disadvantage of the contiguous approach is external fragmentation and copying overhead during insertion and deletion. To avoid this without scattering blocks in a random manner over the disk, a multimedia file system can provide constrained block allocation of the continuous media. In [GC92], different placement strategies were compared. The size of the blocks (M) and the size of the gaps (G) between them can be derived from the requirement of continuity. The size is measured in terms of sectors. We assume that the data transfer rate r_{dt} is the same as the disk rotation rate ($sectors/s$). The continuity requirement in this case is met if the time to skip over a gap and to read the next media block does not exceed the duration of its playback $T_{play}(s)$ [RKV92]:

$$T_{play}(s) \geq \frac{M(sectors)+G(sectors)}{r_{dt}(sectors/s)}$$

Since there are two variables in the equation, the storage pattern (M, G) is not unique. There are several combinations possible to satisfy the above equation. Problems occur if the disk is not sufficiently empty, so that single data streams cannot be stored exactly according to their storage pattern. In this case, the continuity requirements for each block are not strictly maintained. To serve the continuity requirements, read-ahead and buffering of a determined number of blocks must be introduced. See, for example, [RV91, RKV92, VR93] for a detailed description of this storage method.

Some systems using scattered storage make use of a special disk space allocation mechanism to allow fast and efficient access. Abbott performed the pioneer work in this field [Abb84]. He was especially concerned about the size of single blocks and their positions on disk. Another topic to be considered is the placement of different streams. With interleaved placement, all n'th blocks of each stream are in close physical proximity on disk. A contiguous interleaved placement is possible, as well as a scattered interleaved placement. With interleaved data streams, synchronization is much easier to handle. On the other hand, the insertion and deletion of single parts of data streams become more complicated.

In [KWY94], a layout algorithm was developed and analyzed which provides a uniform distribution of media blocks on the disk after copying or writing audio and video files. It takes into account that further files will be merged. Therefore, a set of non-filled gaps is left. This uniform distribution is achieved by storing new blocks at the center of existing – so far – non-filled gaps. With this "central merging method" gaps are successively split into two new equal gaps. It was shown that the mean efficiency of the secondary storage usage with this algorithm was about 75% without violation of any real-time constraint [KWY94].

Disk Scheduling Algorithms

The main goals of traditional disk scheduling algorithms are to reduce the cost of seek operations, to achieve a high throughput and to provide fair disk access for every process. The additional real-time requirements introduced by multimedia systems make traditional disk scheduling algorithms, such as described previously, inconvenient for multimedia systems. Systems without any optimized disk layout for the storage of continuous media depend far more on reliable and efficient disk scheduling algorithms than others. In the case of contiguous storage, scheduling is only needed to serve requests from multiple streams concurrently. In [LS93], a round-robin scheduler is employed that is able to serve hard real-time tasks. Here, additional optimization is provided through the close physical placement of streams that are likely to be accessed together.

The overall goal of disk scheduling in multimedia systems is to meet the deadlines of all time-critical tasks. Closely related is the goal of keeping the necessary buffer space requirements low. As many streams as possible should be served concurrently, but aperiodic requests should also be schedulable without delaying their service for an infinite amount of time. The scheduling algorithm must find a balance between time constraints and efficiency.

Earliest Deadline First

Let us first look at the EDF scheduling strategy as described for CPU scheduling, but used for the file system issue as well. Here the block of the stream with the nearest deadline would be read first. The employment of EDF, as shown in Figure 9.16, in the strict sense results in poor throughput and excessive seek time. Further,

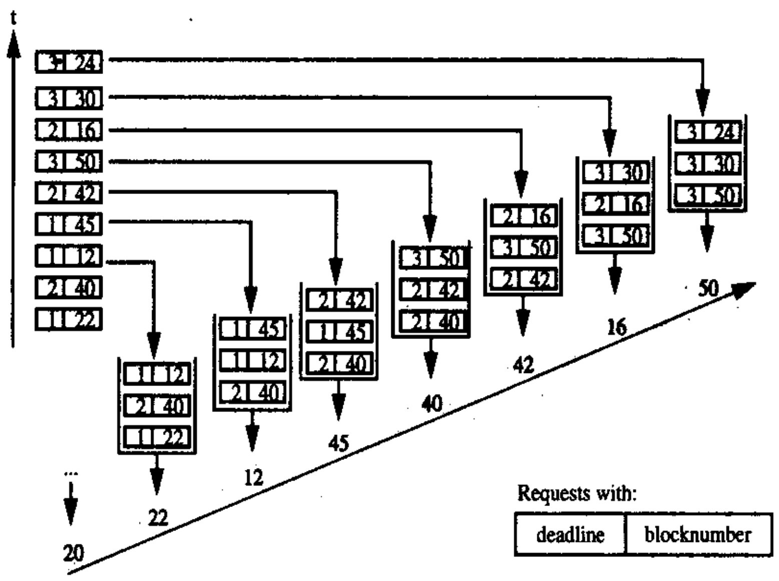

Figure 9.16: *EDF disk scheduling.*

as EDF is most often applied as a preemptive scheduling scheme, the costs for preemption of a task and scheduling of another task are considerably high. The overhead caused by this is in the same order of magnitude as at least one disk seek. Hence, EDF must be adapted or combined with file system strategies.

SCAN-Earliest Deadline First

The SCAN-EDF strategy is a combination of the SCAN and EDF mechanisms [RW93]. The seek optimization of SCAN and the real-time guarantees of EDF are combined in the following way: like in EDF, the request with the earliest deadline is always served first; among requests with the same deadline, the specific one that is first according to the scan direction is served first; among the remaining requests, this principle is repeated until no request with this deadline is left.

Since the optimization only applies for requests with the same deadline, its efficiency depends on how often it can be applied (i.e., how many requests have the same or a similar deadline). To increase this probability, the following tricky technique can be used: all requests have release times that are multiples of the period p. Hence, all requests have deadlines that are multiples of the period p. Therefore, the requests can be grouped together and be served accordingly. For requests with different data rate requirements, in addition to SCAN-EDF, the employment of a periodic fill policy is proposed [YV92] to let all requests have the same deadline. With this policy, all requests are served in cycles. In every cycle, each request gets an amount of service time that is proportional to its required data rate. The cycle length is equal to the sum of the service times of all requests. Thus, in every cycle, all requests can be given a deadline at the end of the cycle.

SCAN-EDF can be easily implemented. Therefore, EDF must be modified slightly. If D_i is the deadline of task i and N_i is the track position, the deadline can be modified to be $D_i + f(N_i)$. Thus the deadline is deferred. The function $f()$ converts the track number of i into a small perturbation of the deadline, as shown in the example of Figure 9.17. It must be small enough so that $D_i + f(N_i) \leq D_j + f(N_j)$ holds for all $D_i \leq D_j$. For $f(N_i)$, it was proposed to choose the following function [RW93]:

$$f(N_i) = \frac{N_i}{N_{max}}$$

where N_{max} is the maximum track number on disk. Other functions might also be appropriate.

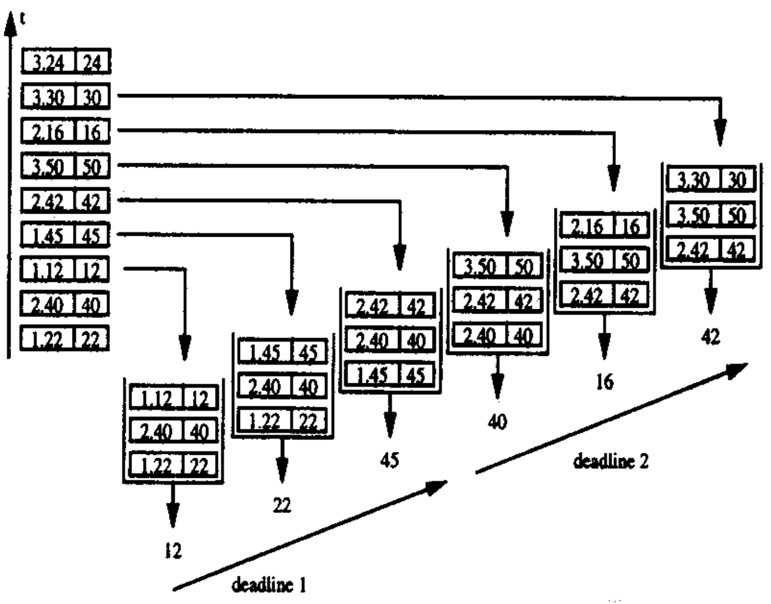

Figure 9.17: *SCAN-EDF disk scheduling with $N_{max} = 100$ and $f(N_i) = N_i/N_{max}$.*

We enhanced this mechanism by proposing a more accurate perturbation of the deadline which takes into account the actual position of the head (N). This position is measured in terms of block numbers and the current direction of the head movement (see also Figures 9.18 and 9.19):

1. If the head moves toward N_{max}, i.e., upward, then

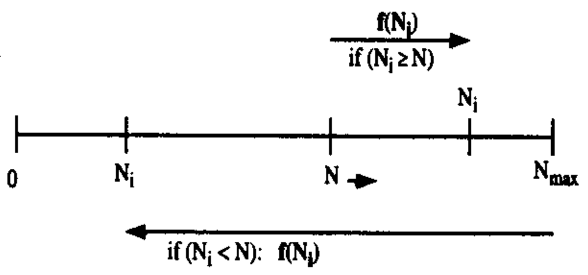

Figure 9.18: *Accurate EDF-SCAN algorithm, head moves upward.*

   (a) for all blocks N_i located between the actual position N and N_{max}, the perturbation of the deadline is:

$f(N_i) = \frac{N_i - N}{N_{max}}$ *for all* $N_i \geq N$

(b) for all blocks N_i located between the actual position and the first block (no. 0):

$f(N_i) = \frac{N_{max} - N_i}{N_{max}}$ *for all* $N_i < N$

2. If the head moves downward towards the first blocks, then

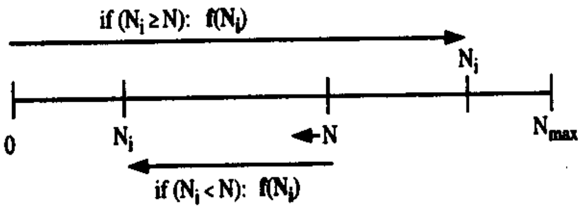

Figure 9.19: *Accurate EDF-SCAN algorithm, head moves downward.*

(a) for all blocks located between the actual position and N_{max}:

$f(N_i) = \frac{N_i}{N_{max}}$ *for all* $N_i > N$

(b) for all blocks located between this first block with the block number 0 and the actual position:

$f(N_i) = \frac{N - N_i}{N_{max}}$ *for all* $N_i \leq N$

Our algorithm is more computing-intensive than those with the simple calculation of [RW93]. In cases with only a few equal deadlines, our algorithm provides improvements and the expenses of the calculations can be tolerated. In situations with many, i.e., typically more than five equal deadlines, the simple calculation provides sufficient optimization and additional calculations should be avoided. SCAN-EDF was compared with pure EDF and different variations of SCAN. It was shown that SCAN-EDF with deferred deadlines performed well in multimedia environments [RW93].

Group Sweeping Scheduling

With *Group Sweeping Scheduling (GSS),* requests are served in cycles, in round-robin manner [CKY93]. To reduce disk arm movements, the set of n streams is divided into g groups. Groups are served in fixed order. Individual streams within a group are served according to SCAN; therefore, it is not fixed at which time or order individual streams within a group are served. In one cycle, a specific stream may be the first to be served; in another cycle, it may be the last in the same group. A smoothing buffer which is sized according to the cycle time and data rate of the stream assures continuity. If the SCAN scheduling strategy is applied to all streams of a cycle without any grouping, the playout of a stream cannot be started until the end of the cycle of its first retrieval (where all requests are served once) because the next service may be in the last slot of the following cycle. As the data must be buffered in GSS, the playout can be started at the end of the group in which the first retrieval takes place. Whereas SCAN requires buffers for all streams, in GSS, the buffer can be reused for each group. Further optimizations of this scheme are proposed in [CKY93]. In this method, it is ensured that each stream is served once in each cycle. GSS is a trade-off between the optimization of buffer space and arm movements. To provide the requested guarantees for continuous media data, we propose here to introduce a "joint deadline" mechanism: we assign to each group of streams one deadline, the "joint deadline." This deadline is specified as being the earliest one out of the deadlines of all streams in the respective group. Streams are grouped in such a way that all of them comprise similar deadlines. Figure 9.20 shows an example of GSS.

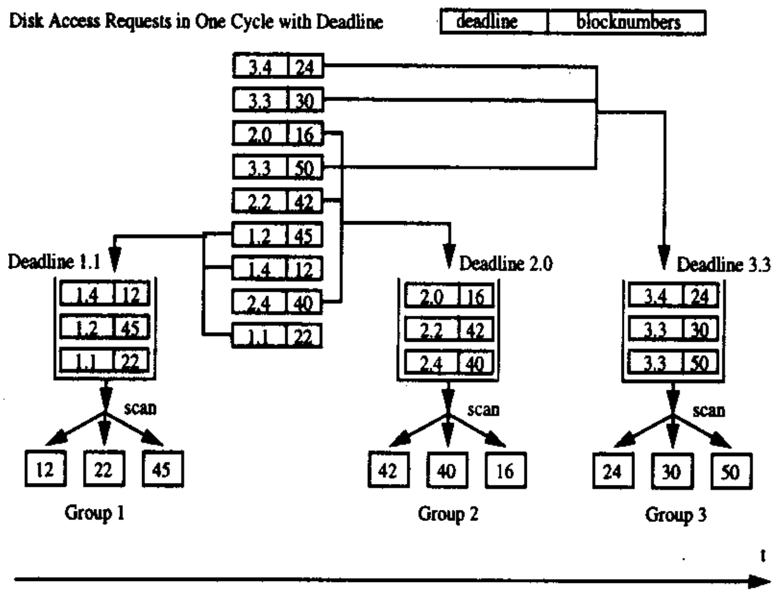

Figure 9.20: *Group sweeping scheduling as a disk access strategy.*

Mixed Strategy

In [Abb84], a *mixed strategy* was introduced based on the *shortest seek* (also called greedy strategy) and the *balanced strategy*. As shown in Figure 9.21, every time data are retrieved from disk they are transferred into buffer memory allocated for the respective data stream. From there, the application process removes them one at a time. The goal of the scheduling algorithm is:

- To maximize transfer efficiency by minimizing seek time and latency.
- To serve process requirements with a limited buffer space.

With shortest seek, the first goal is served, i.e., the process of which data block is closest is served first. The balanced strategy chooses the process which has the least amount of buffered data for service because this process is likely to run out of data. The crucial part of this algorithm is the decision of which of the two strategies must be applied (shortest seek or balanced strategy). For the employment of shortest,

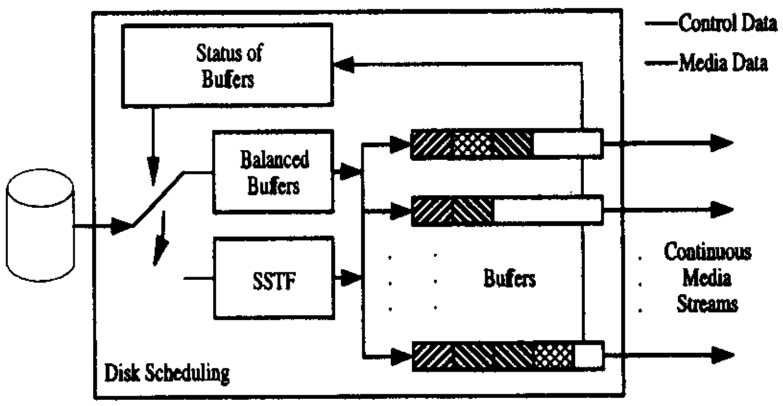

Figure 9.21: em Mixed disk scheduling strategy.

seek two criteria must be fulfilled: the number of buffers for all processes should be balanced (i.e., all processes should nearly have the same number of buffered data) and the overall required bandwidth should be sufficient for the number of active processes, so that none of them will try to immediately read data out of an empty buffer. In [Abb84], the urgency is introduced as an attempt to measure both. The urgency is the sum of the reciprocals of the current "fullness" (amount of buffered data). This number measures both the relative balance of all read processes and the number of read processes. If the urgency is large, the balance strategy will be used; if it is small, it is safe to apply the shortest seek algorithm.

Continuous Media File System

CMFS Disk Scheduling is a non-preemptive disk scheduling scheme designed for the Continuous Media File System (CMFS) at UC-Berkeley [AOG91]. Different policies can be applied in this scheme. Here the notion of the slack time H is introduced. The slack time is the time during which CMFS is free to do non-real-time operations or workahead for real-time processes, because the current workahead of each process is sufficient so that no process would starve, even if it would not be served for H seconds. The considered real-time scheduling policies are:

- The *Static/Minimal policy* is based on the minimal *Workahead Augmenting Set (WAS).* A process p_i reads a file at a determined rate R_i. To each process, a positive integer M_i is assigned which denotes the time overhead required to read a block covering, for example, the seek time. The CMFS performs a set of operations (i.e., disk operations required by all processes) by seeking the next block of a file and reading M_i blocks of this file. Note, we consider only read operations; the same also holds, with minor modifications, for write operations. This seek is done for every process in the system. The data read by a process during this operation "last" $\frac{M_i \times A}{R_i}$, where A is the block size in bytes. The WAS is a set of operations where the data read for each process "last longer" than the worst-case time to perform the operations (i.e., the sum of the read operations of all processes is less than the time read data last for a process). A schedule is derived from the set that is workahead-augmenting and feasible (i.e., the requests are served in the order given by the WAS). The *Minimal Policy,* the minimal WAS, is the schedule where the worst-case elapsed time needed to serve an operation set is the least (i.e., the set is ordered in a way that reduces time needed to perform the operations, for example, by reducing seek times). The *Minimal Policy* does not consider buffer requirements. If there is not enough buffer, this algorithm causes a buffer overflow. The *Static Policy* modifies this schedule such that no block is read if this would cause a buffer overflow for that process. With this approach, starvation is avoided, but its use of short operations causes high seek overhead.

- With the *Greedy Policy,* a process is served as long as possible. Therefore, it computes at each iteration the slack time H. The process with the smallest workahead is served. The maximum number n of blocks for this process is read; n is determined by H (the time needed to read n blocks must be less than or equal to H) and the currently available buffer space.

- The *Cyclical Plan Policy* distributes the slack time among processes to maximize the slack time. It calculates H and increases the minimal WAS with H milliseconds of additional reads; an additional read for each process is done immediately after the regular read determined by the minimal WAS. This policy distributes workahead by identifying the process with the smallest slack time and schedules an extra block for it; this is done until H is exhausted. The number of block reads for the least workahead is determined. This procedure is repeated every time the read has completed.

The *Aggressive* version of the Greedy and the Cyclical Plan Policy calculates H of all processes except the least workahead process that is immediately served by both policies. If the buffer size limit of a process is reached, all policies skip to the next process. Non-real-time operations are served if there is enough slack time. Performance measurements of the above introduced strategy showed that Cyclical Plan increases system slack faster at low values of the slack time (which is likely to be the case at system setup). With a higher system slack time, apart of the Static/Minimal Policy, all policies perform about the same.

All of the disk scheduling strategies described above have been implemented and tested in prototype file systems for continuous media. Their efficiency depends on the design of the entire file system, the disk layout, tightness of deadlines, and last but not least, on the application that is behaving. It is not yet common sense which algorithm is the "best" method for the storage and retrieval of continuous media files. Further research must show which algorithm serves the timing requirements of continuous media best and ensures that aperiodic and non-real-time requests are efficiently served.

Data Structuring

Continuous media data are characterized by consecutive, time-dependent logical data units. The basic data unit of a motion video is a frame. The basic unit of audio is a sample. Frames contain the data associated with a single video image, a sample represents the amplitude of the analog audio signal at a given instance. Further structuring of multimedia data was suggested in the following way [RV91, Ran93, SF92]: a strand is defined as an immutable sequence of continuously recorded video frames, audio samples, or both. It means that it consists of a sequence of blocks which contain either video frames, audio samples or both. Most often it includes headers and further information related to the type of compression used. The file system holds primary indices in a sequence of *Primary Blocks*. They contain mapping from media block numbers to their disk addresses. In *Secondary Blocks* pointers to all primary blocks are stored. The *Header Block* contains pointers to all secondary blocks of a strand. General information about the strand like, recording rate, length, etc., is also included in the header block.

Media strands that together constitute a logical entity of information (e.g., video and associated audio of a movie) are tied together by synchronization to form a multimedia rope. A rope contains the name of its creator, its length and access rights. For each media strand in this rope, the strand ID, rate of recording, granularity of storage and corresponding block-level are stored (information for the synchronization of the playback start for all media at the strand interval boundaries). Editing operations on ropes manipulate pointers to strands only. Strands are regarded as immutable objects because editing operations like insert or delete may require substantial copying which can consume significant amounts of time and space. Intervals of strands can be shared by different ropes. Strands that are not referenced by any rope can be deleted, and storage can be reclaimed [RV91]. The following interfaces are the operations that file systems provide for the manipulation of ropes:

- RECORD [media] [requestID, mmRopeID]

  A multimedia rope, represented by mmRopeID and consisting of media strands, is recorded until a STOP operation is issued.

- PLAY [mmRopeID, interval, media] requestID

  This operation plays a multimedia rope consisting of one or more media strands.

- STOP [requestID]

  This operation stops the retrieval or storage of the corresponding multimedia rope.

- Additionally, the following operations are supported:

  - INSERT [baseRope, position, media, withRope, withInterval]
  - REPLACE [baseRope, media, baseInterval, withRope, withInterval]
  - SUBSTRING [baseRope, media, interval]
  - CONCATE [mmRopeID1, mmRopeID2]

- DELETE [baseRope, media, interval]

Figure 9.22 provides an example of the INSERT operation, whereas Figure 9.23 shows the REPLACE operation.

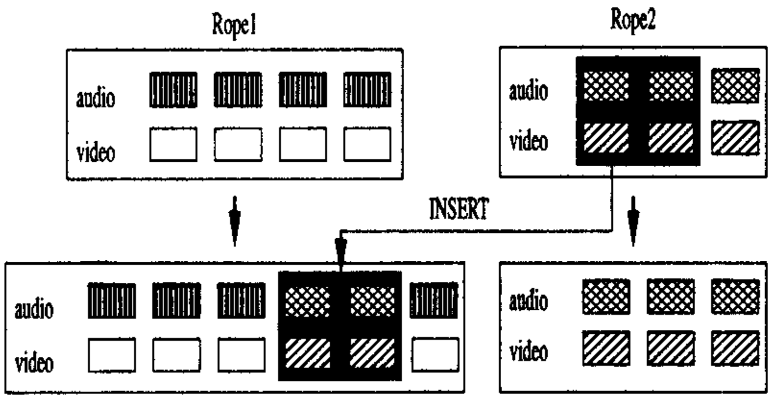

Figure 9.22: *INSERT operation.*

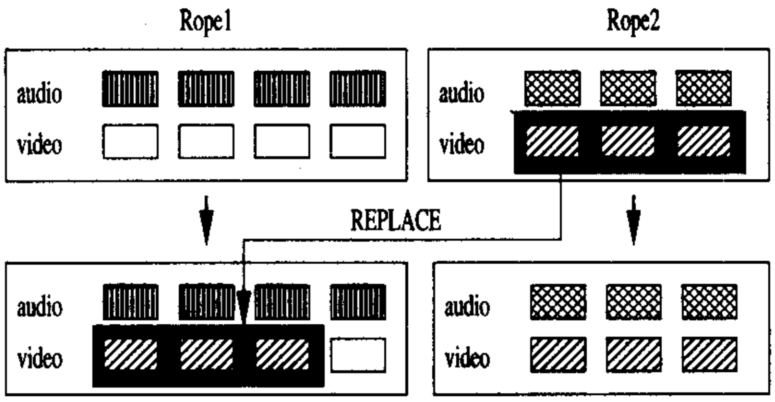

Figure 9.23: *REPLACE operation.*

The storage system is divided into two layers:

- The *rope server* is responsible for the manipulation of multimedia ropes. It communicates with applications, allows the manipulation of ropes and communicates with the underlying *storage manager* to record and play back multimedia strands. It provides the rope abstraction to the application. The rope access methods were designed similarly to UNIX file access routines. Status messages about the state of the play or record operation are passed to the application.

- The *storage manager* is responsible for the manipulation of strands. It places the strands on disk to ensure continuous recording and playback. The interface to the rope server includes four primitives for manipulating strands:

  1. "PlayStrandSequence" takes a sequence of strand intervals and displays the given time interval of each strand in sequence.
  2. "RecordStrand" creates a new strand and records the continuous media data either for a given duration or until StopStrand is called.
  3. "StopStrand" terminates a previous PlayStrandSequence or RecordStrand instance.
  4. "DeleteStrand" removes a strand from storage.

The experimental Video File Server introduced in [Ran93] supports integrated storage and retrieval of video. The "Video Rope Server" presents a device-independent directory interface to users (Video Rope). A Video Rope is characterized as a hierarchical directory structure constructed upon stored video frames. The "Video Disk Manager" manages a frame-oriented motion video storage on disk, including audio and video components.

9.6 Additional Operating System Issues

9.6.1 Interprocess Communication and Synchronization

In multimedia systems, *interprocess communication* refers to the exchange of different data between processes. This data transfer must be very efficient because continuous media require the transfer of a large amount of data in a given time span. For the exchange of discrete media data, the same mechanisms are used as in traditional operating systems. Data interchange of continuous media is closely related to memory management and is discussed in the respective section.

Synchronization guarantees timing requirements between different processes. In the context of multimedia, this is an especially interesting aspect. Different data streams, database entries, document portions, positions, processes, etc., must be synchronized. Thus, synchronization is important for various components of a multimedia system and therefore is not included in this discussion on operating systems.

9.6.2 Memory Management

The memory manager assigns physical resource *memory* to a single process. Virtual memory is mapped onto memory that is actually available. With *paging*, less frequently used data is swapped between main memory and external storage. Pages are transferred back into the main memory when data on them is required by a process. Note, continuous media data must not be swapped out of the main memory. If a page of virtual memory containing code or data required by a real-time process is not in real memory when it is accessed by the process, a page fault occurs, meaning that the page must be read from disk. Page faults affect the real-time performance very seriously, so they must be avoided. A possible approach is to lock code and/or data into real memory. However, care should be taken when locking code and/or data into real memory. Real memory is a very scarce resource to the system. Committing real memory by pinning (locking) will decrease overall system performance. The typical AIX kernel will not allow more than about 70% of real memory to be committed to pinned pages [IBM91].

The transmission and processing of continuous data streams by several components require very efficient data transfer restricted by time constraints. Memory allocation and release functions provide well-defined access to shared memory areas. In most cases, no real processing of data, but only a data transfer, is necessary. For example, the camera with a digitalization process is the source and the presentation process is the sink. The essential task of the other components is the *exchange* of continuous media data with relatively high data rates in real-time. Processing involves computing, adding, interpreting and stripping headers. This is well-known in communications [MR93b]. The actual implementation can either be with external devices and dedicated hardware in the computer, or it can be realized with software components.

Early prototypes of multimedia systems incorporate audio and video based on external data paths only. Memory management, in this case, has a switching function only, i.e., to control an external switch.

A first step toward integration was the incorporation of the external switch function into the computer. Therefore, some dedicated adapter cards that are able to switch data streams with varying data rates were employed.

A complete integration can be achieved with a full digital approach within the computer, i.e., to offer a pure software solution. Data are transmitted between single components in real-time. Copy operations are – as far as possible – reduced to the exchange of pointers and the check of access rights. This requires the access of a shared address space. Data can also be directly transferred between different adapter cards. The transfer of continuous media data takes place in a real-time environment. This exchange is controlled, but not necessarily executed, by the application. The data transfer must be performed by processes running in a real-time environment. The application running in a non-real-time environment generates, manipulates and consumes these data streams at an operating system interface.

9.6.3 Device Management

Device management and the actual access to a device allows the operating system to integrate all hardware components. The physical device is represented by an abstract device driver. The physical characteristics of devices are hidden. In a conventional system, such devices include a graphics adapter card, disk, keyboard and mouse. In multimedia systems, additional devices like cameras, microphones, speakers and dedicated storage devices for audio and video must be considered. In most existing multimedia systems, such devices are not often integrated by device management and the respective device drivers.

Existing operating system extensions for multimedia usually provide one common system-wide interface for the control and management of data streams and devices. In Microsoft Windows and OS/2 this interface is known as the Media Control Interface (MCI). The multimedia extensions of Microsoft Windows, for example, provide the following classes of function calls:

- *System commands* are not forwarded to the single device driver (MCI driver); they are served by a central instance. An example of such a command is the query concerning all devices connected to the system (Sysinfo).
- Each device driver must be able to process *compulsory commands*. For instance, the query for specific characteristics (capability info) and the opening of a device (open) are such commands.
- Basic commands refer to characteristics that all devices have in common. They can be supported by drivers. If a device driver processes such a command, it must consider all variants and parameters of the command. A data transmission is typically started by the basic command "play."
- Extended commands may refer to both device types and special single devices. The "seek" command for the positioning on an audio CD is an example. On the basis of a controllable camera, the required concepts are explained in more detail. A camera has functions to adjust the focal length, focus and position. An abstraction of the functionality provided by the physical camera as an video input device covers the following layers, which relate to different components in a multimedia system:
  - The application has access to a logical camera without knowledge about the specific control functions of the camera. The focal length is adjusted in millimeters. The driver translates a specific "set focal length command" into a sequence of camera hardware control commands and passes them to the control logic. The provision of such an abstract interface and the transformation into hardware-dependent commands is a task of the device management of a multimedia operating system.
  - Different input device classes have similar characteristics. The zoom operation of a camera can be applied in a similar way to the presentation of a still image. The still image could be zoomed. For example, consider an image stored on a Photo-CD with a given resolution. The zoom operation could result in the presentation either of the image with its specified

resolution, or of a particular section of the image. This kind of abstraction is part of the programming environment of a multimedia system and not a task of an operating system, although in some cases it is performed by the operating system. The basic commands define several operations supported by different devices. The basic command used for the start of a data transmission between the camera and the video window of an application – called the play command in this context – can be used in a second realization for file transfer – as a kind of copy command.

To complete the description of the camera control, the positioning of the camera is discussed. To change the position of the camera, the application specifies the target coordinates in a polar coordinate system. Yet, a concrete camera control can only execute commands like "move swivel slope head in a specific direction with a defined speed." The direction can be "left" or "right," respectively "up" or "down." Eight different speed levels are given, but it is only possible to change the speed in steps of the maximum two levels. During acceleration, consecutive commands with speed levels 2, 4, 6, 8 must be executed. It is the task of the camera driver to perform the mapping of coordinates into this positioning controlled by time and speed.

To define the required application interface, the selectable control class can be subdivided into four function categories [RSSS90]:

1. Defined, compulsory and generic: all operations that must be provided for each device driver, regardless of its specific functionality, belong to this category. This corresponds to the above-mentioned commands of the MCI.

2. Defined, compulsory and device specific: all functions and parameters specified in this category must be provided by the device driver. Therefore, there exists a defined interface in the respective operating system. For example, a camera driver must be able to answer an inquiry for an eventual existing auto focus mechanism.

3. Defined but not compulsory: for each device type, a set of functions is defined which covers all known possibilities. The functions cannot be provided by all different devices and drivers. In the case of the camera, such functions are, for example, to position and adjust the focal length, because not every camera

has these facilities. The interface is defined keeping in mind what is possible and meaningful. If such a function is employed, although it is not supported by the implementation, a well-defined error handling mechanism applies. The application can handle these errors, and therefore it is independent of the connected physical devices.

4. Not defined and not compulsory: we must be aware that there will always be unpredictable new devices and special developments. Hence, the operating system provides a fourth category of functions to cover all these calls.

An unambiguous definition of these categories allows easier integration of devices into the programming environment. The multimedia extensions of today's operating systems incorporate device management with a first step of functional distinction toward the above outlined categories.

9.7 System Architecture

The employment of continuous media in multimedia systems also imposes additional, new requirements to the system architecture. A typical multimedia application does not require processing of audio and video to be performed by the application itself. Usually, data are obtained from a source (e.g., microphone, camera, disk, network) and are forwarded to a sink (e.g., speaker, display, network). In such a case, the requirements of continuous media data are satisfied best if they take "the shortest possible path" through the system, i.e., to copy data directly from adapter to adapter. The program then merely sets the correct switches for the data flow by connecting sources to sinks. Hence, the application itself never really touches the data as is the case in traditional processing. A problem with direct copying from adapter to adapter is the control and the change of quality of service parameters. In multimedia systems, such an adapter to adapter connection is defined by the capabilities of the two involved adapters and the bus performance. In today's systems, this connection is static. This architecture of low-level data streaming corresponds with proposals for using additional new busses for audio and video transfer within a computer. It also enables a switch-based rather than a bus-based data transfer architecture [Fin91, HM91]. Note, in practice we encounter headers and trailers surrounding continuous media data coming from devices and being delivered to devices. In the case of compressed video data, e.g., MPEG-2, the program stream contains several layers of headers compared with the actual group of pictures to be displayed.

Most of today's multimedia systems must coexist with conventional data processing. They share hardware and software components. For instance, the traditional way of protocol processing is slow and complicated. In high-speed networks, protocol processing is the bottleneck because it cannot provide the necessary throughput. Protocols like VMTP, NETBLT and XTP try to overcome this drawback, but research in this area has shown that throughput in most communication systems is not bounded by protocol mechanisms, but by the way they are implemented [CJRS89]. A time-intensive operation is, for example, physical buffer copying. Since the memory on the adapter is not very large and it may not store all related compressed images, data must be copied at least once from adapter into main memory. Further copying should be avoided. Appropriate buffer management allows operations on data without performing any physical copying. In operating systems like UNIX, the buffer management must be available in both the user and the kernel space. The data need to be stored in shared memory to avoid copying between user and kernel space. For further performance improvement, protocol processing should be done in threads with upcalls, i.e., the protocol processing for an incoming message is done by a single thread. A development to support such a protocol process management is, for example, the x-Kernel.

The architecture of the protocol processing system is just one issue to be considered in the system architecture of multimedia supporting operating systems. Multimedia data should be delivered from the input device (e.g., CD-ROM) to an output device (e.g., a video decompression board) across the fastest possible path. The paradigm of streaming from source to sink is an appropriate way of doing this. Hence, the multimedia application opens devices, establishes a connection between them, starts the data flow and returns to other duties.

As stated above, the most dominant characteristic of multimedia applications is to preserve the temporal requirement at the presentation time. Therefore, multimedia data is handled in a *Real-Time Environment (RTE)*, i.e., its processing is scheduled according to the inherent timing requirements of multimedia data. On a multimedia computer, the RTE will usually coexist with a *Non-Real-Time Environment (NRTE)*. The NRTE deals with all data that have no timing requirements. Figure 9.24 shows the approached architecture. Multimedia I/O devices are, in gen-

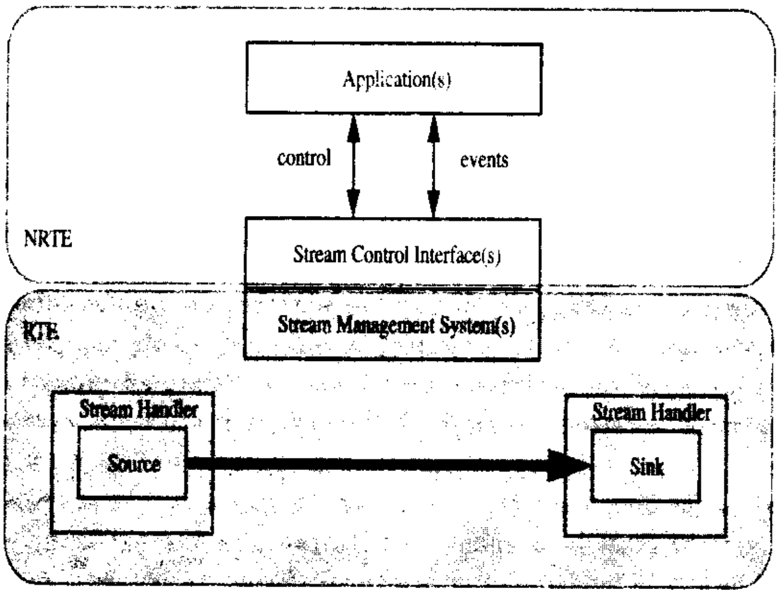

Figure 9.24: *Real-time and non-real-time environments.*

eral, accessed from both environments. Data such as a video frame, for example, is passed from the RTE to the display. The RTE is controlled by related functions in the NRTE. The establishment of communication connections at the start of a stream must not obey timing requirements, but the data processing for established connections is compelled. All control functions are performed in the NRTE. The application usually calls only these control functions and is not involved in active continuous media data handling. Therefore, the multimedia application itself typically runs in the NRTE and is shielded from the RTE. In some scenarios, users may want applications to "process" continuous media data in an application-specific way. In our model, such an application comprises a module running as stream handler in the RTE. The rest of the applications run in the NRTE, using the available stream control interfaces. System programs, such as communication protocol processing and database data transfer programs, make use of this programming in the RTE. Whereas applications like authoring tools and media presentation programs are relieved from the burden of programming in the RTE, they just interface and control the RTE services. Applications determine processing paths which are needed for their data processing, as well as the control devices and paths.

To reduce data copying, buffer management functions are employed in the RTE. This buffer management is located "between" the stream handlers. Stream handlers are entities in the RTE which are in charge of multimedia data. Typical stream handlers are filter and mixing functions, but they are also parts of the communication subsystem described above and can be treated in the same way. Each stream handler has endpoints for input and output through which data units flow. The stream handler consumes data units from one or more input endpoints and generates data units through one or more output endpoints.

Multimedia data usually "enters" the computer through an input device, a source, and "leaves" it through an output device, a sink (where storage can serve as an I/O device in both cases). Sources and sinks are implemented by a device driver. Applications access stream handlers by establishing sessions with them. A session constitutes a virtual stream handler for exclusive use by the application which has created it. Depending on the required QoS of a session, an underlying resource management subsystem multiplexes the capacity of the underlying physical resources among the sessions. To manage the RTE data flow through the stream handlers, control operations are used which belong to the NRTE. These functions make up the stream management system in the multimedia architecture. Operations are provided by all stream handlers (e.g., operations to establish sessions and connect their endpoints) and operations specific to individual stream handlers usually determine the content of a multimedia stream and apply to particular I/O devices.

Some applications which are all in the NRTE have the need to correlate discrete data such as text and graphics with continuous streams, or to post-process multimedia data (e.g., to display the time stamps of a video stream like a VCR). These applications need to obtain segments of multimedia at the stream handler interface. With a grab function, the segments are copied to the application as if stream duplication took place. Due to this operation, the data units lose their temporal properties because they enter the NRTE. Applications that must generate or transform multimedia data keeping real-time characteristics must use a stream handler included in the RTE, which performs the required processing.

The synchronization of streams is a function that is provided by the stream management subsystem. Synchronization is specified on a connection basis and can be expressed using the notions of a clock or logical time systems. It determines points in time at which the processing of data units shall start. For regular streams, the stream rates can be used to relate data units to synchronization points. Sequence numbers can accomplish the same task. Time stamps are a more versatile means for synchronization as they can also be used for non-periodic traffic. Synchronization is often implemented by delaying the execution of a thread or by delaying the receive operation on a buffer exchanged between stream handlers.

Many operating systems already provide extensions to support multimedia applications. In the next paragraphs, three of these multimedia extensions are presented.

9.7.1 UNIX-based Systems

In the UNIX operating system, the applications in the user space generally make use of system calls in the NRTE. Either the whole operating system or a part of it is also located in the NRTE and in the kernel space. Extensions to the operating system providing real-time capabilities make up the RTE part of the kernel space (see Figure 9.25).

The actual implementation of the RTE varies substantially:

- SUN OS does not yet provide an RTE.
- AIX includes real-time priorities. This feature provides the basis for the RTE in the AIX-based Ultimedia^{TM}server.
- The IRIX operating system on Silicon Graphics Workstations has real-time capabilities, i.e., it includes an RTE.

9.7.2 QuickTime

QuickTime is a software extension to the Macintosh System. It provides the capability to capture, store, manage, synchronize and display continuous media data.

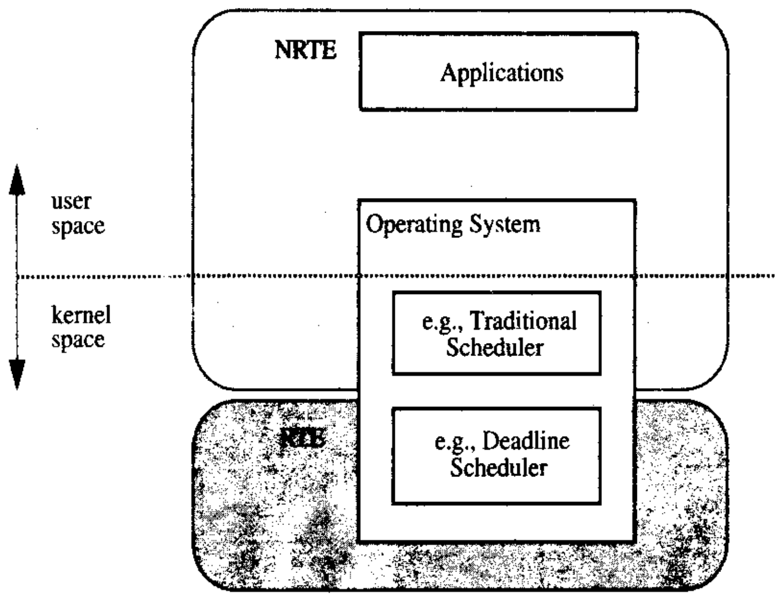

Figure 9.25: *NRTE and RTE in UNIX systems.*

A more detailed description can be found in [DM92]. It introduces digitized video as standard data type into the system, and it allows an easier handling of other continuous media like audio and animation. Standard applications are enhanced by multimedia capabilities. Apple has announced QuickTime to be available for other operating systems like Windows and UNIX as well. An integration of future hardware and software developments is possible.

The standard data type of QuickTime is a movie. All kinds of continuous media data are stored in movie documents. Additionally, time information like the creation and modification date, duration, etc., are also kept in the movie document. With each movie, a poster frame is associated that appears in the dialog box. Other information like current editing selection, spatial characteristics (transformation matrix, clipping region) and a list of one or more tracks are associated with the movie. A track represents a stream of information (audio or video data) that flows in parallel to every other track. With each track, information like creation and modification data, duration, track number, spatial characteristics (transformation matrix, display window, clipping region), a list of related tracks, volume and start time, duration, playback rate and a data reference for each media segment is stored. A media segment is a set of references to audio and video data, including time information (creation, modification, duration), language, display or sound quality, media data type and data pointers. Future releases will have, apart from audio and video tracks, "custom tracks" such as a subtitle track. All tracks can be viewed or heard concurrently. The tracks of a movie are always synchronized. Since movies are documents they cannot only be played (including pausing, stepping through, etc.), but they can also be edited. Operations like cut, copy and paste are possible. Movie documents can be part of other documents. QuickTime is scalable. Hardware components like accelerator or compressor/decompressor cards can be employed.

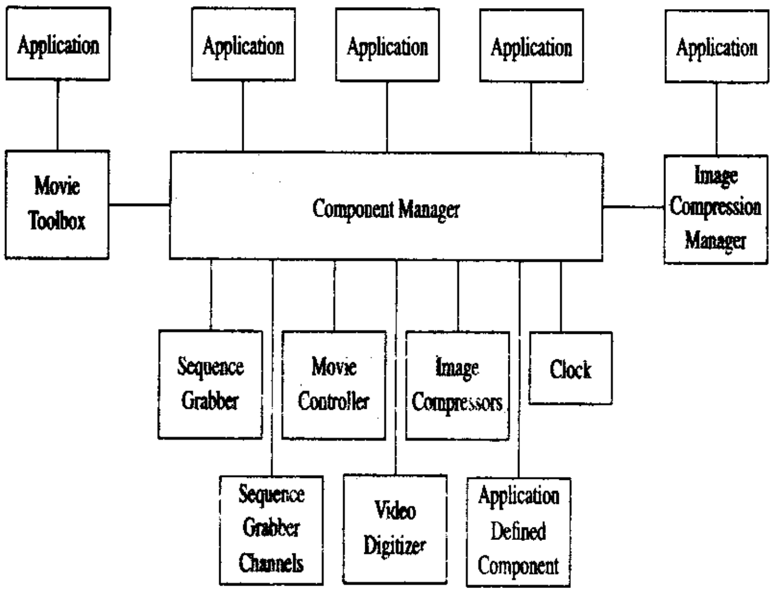

Figure 9.26: *QuickTime architecture.*

The QuickTime architecture comprises three major components (see Figure 9.26): the *Movie Toolbox* offers a set of services to the user that allows him/her to incorporate movies into applications. These applications may directly manipulate characteristics of audio and video data of movies. The movie is integrated in the desktop environment. Movie data can be imported and exported with the system clipboard and a movie can be edited within the Movie Toolbox.

The second component, known as the *Image Compression Manager,* provides a common interface for compression and decompression of data, independent of the implementation, to and from hard disk, CD-ROM and floppy. It offers a directory service to select the correct compression component. Different interface levels for different application requirements are available. The compression techniques are a proprietary image compression scheme, a JPEG implementation and a proprietary video compressor for digitized video data (leading to a compression ratio of 8:1, and if temporal redundancies are also removed, to a ratio of 25:1). An animation compressor can compress digital data in lossy and lossless (error-free) modes. A graphics compressor is also available. The pixel depth conversion in bits per pixel can be used as a filter to be applied in addition to other compressors.

The *Component Manager* provides a directory service related to the components. It is the interface between the application and various system components. It shields developers from having to deal with the details of interfacing with specific hardware. In the Component Manager, object-oriented concepts (e.g., hierarchical structure, extensible class libraries, inheritance of component functionality, instance-based client/server model) are applied. Thus, applications are independent of implementations, can easily integrate new hardware and software and can adapt to the available resources. The components managed by the Component Manager are the Clock, the Image Compressor and Image Decompressor, the Movie Controller, the Sequence Grabber, Sequence Grabber Channel and the Video Digitizer. Furthermore, application-defined components can be added.

There is a simple resource management scheme applied to the local environment only: in the case of scarce resources, audio is prioritized over video, i.e., audio playback is maintained (if possible) whereas single video frames might be skipped. If an application calls the Movie Toolbox during playback, there are the following possibilities to handle these calls:

- The commonly used mode is a preemptive calling sequence, where the application returns to the system after each update. This might cause jerky movie output.
- With a non-preemptive calling sequence, the application does not return to the system while a movie is played. This counteracts the multitasking capability.

- The high-performance controlled preemptive calling sequence is a compromise, where the application gives up control to the Movie Toolbox for a specified time period (e.g., 50 ms).

As an additional resource management scheme for better performance, it is recommended to turn off the virtual memory while playing QuickTime movies. If it is on, it will cause the sound to skip and it will lower the frame rate during the playback of a movie. However, no RTE exists.

The concept of components in QuickTime allows for easy extension without affecting applications. It attempts to form a hierarchical structure of functionality by components. The movie controller component eases user interface programming. A disadvantage of QuickTime is that there is no clear layering of abstractions for programmers and that the functionality of managers and components sometimes overlaps.

9.7.3 Windows Multimedia Extensions

The Microsoft *Windows Multimedia Extensions (WME)* are an enhancement to the Windows programming environment. They provide high-level and low-level services for the development of multimedia applications for application developers, using the extended capabilities of a multimedia personal computer [Win91].

The following services for multimedia applications are provided by the WME:

- A *Media Control Interface (MCI)* for the control of media services. It comprises an extensible string-based and message-based interface for communication with MCI device drivers. The MCI device drivers are designed to support the playing and recording of waveform audio, the playing of MIDI (Musical Instrument Digital Interface) files, the playing of compact disk audio from a CD-ROM disk drive and the control of some video disk players.
- A *Low-level API (Application Programming Interface)* provides access to multimedia-related services like playing and recording audio with waveform and MIDI audio devices. It also supports the handling of input data from joysticks and precise timer services.

- A *multimedia file I/O service* provides buffered and unbuffered file I/O. It also supports the standard IBM/Microsoft Resource Interchange File Format (RIFF) files. These services are extensible with custom I/O procedures that can be shared among applications.
- The most important *device drivers* available for multimedia applications are:
  - An *enhanced high-resolution video display driver* for Video 7 and Paradise VGA cards providing 256 colors, improved performance, and other new features.
  - A *high-resolution VGA video display driver* allowing the use of a custom 16-color palette as well as the standard palette.
  - A *low resolution VGA video display driver* providing 320-by-320 resolution with 256 colors.
  - The *Control Panel Applets* that allow the user to change display drivers, to set up a screen saver, to install multimedia device drivers, to assign waveform sounds to system alerts, to configure the MIDI Mapper and to calibrate joysticks. A MIDI Mapper supports the MIDI patch service that allows MIDI files to be authored independently of end-user MIDI synthesizer setups.

Figure 9.27 shows the rough architecture of MS Windows Multimedia Extensions: MMSYSTEM library provides the Media Control Interface services and low-level multimedia support functions. The communication between the low-level MMSYSTEM functions and multimedia devices, such as waveform, MIDI, joystick and timer, is provided by the multimedia device drivers. The high-level control of media devices is provided by the drivers for the Media Control Interface.

The main concepts of the architecture of the Multimedia Extensions are extensibility and device-independence. They are provided by a translation layer (MMSYSTEM) that isolates applications from device drivers and centralizes device-independent code, run-time linking that allows the MMSYSTEM translation layer to link to the drivers it needs and a well-defined and consistent driver interface that minimizes the development of specialized code and makes the installation and upgrade processes easier.

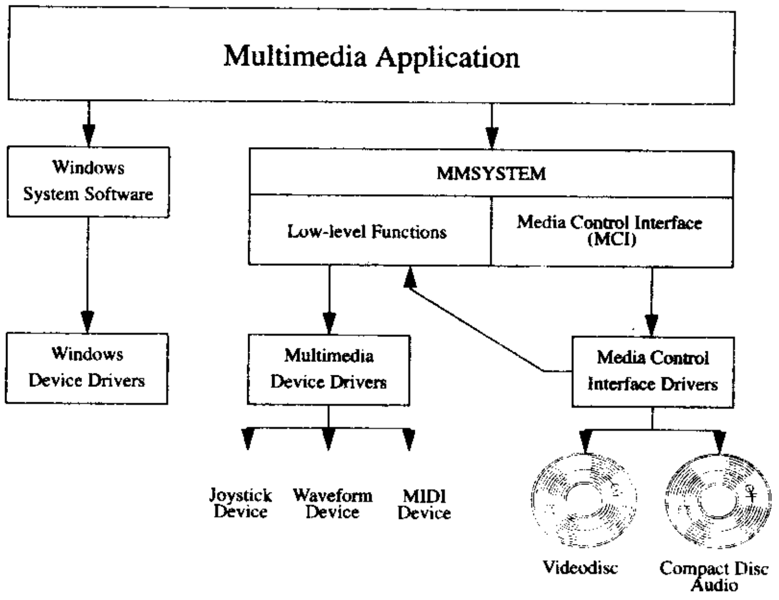

Figure 9.27: *MS Windows Multimedia Extensions architecture.*

9.7.4 OS/2 Multimedia Presentation Manager/2

The Multimedia Presentation Manager/2 (MMPM/2) is part of IBM's Operating System/2 (OS/2). OS/2 is a platform well-suited for multimedia because it supports, e.g., preemptive multitasking, priority scheduling, overlapped I/O and demand-paged virtual memory storage. Figure 9.28 provides an overview of the architecture.

The *Media Control Interface (MCI)* is a device-independent programming interface that offers commands similar to an entertainment system. The following list comprises a selection of typical MCI-commands:

- "Open," "close," and "status of a device" are provided for all devices.
- For playback and recording device-dependent "play," "record," "resume," "stop," "cue" and "seek" commands exist.
- "Set cue point" allows for synchronization.

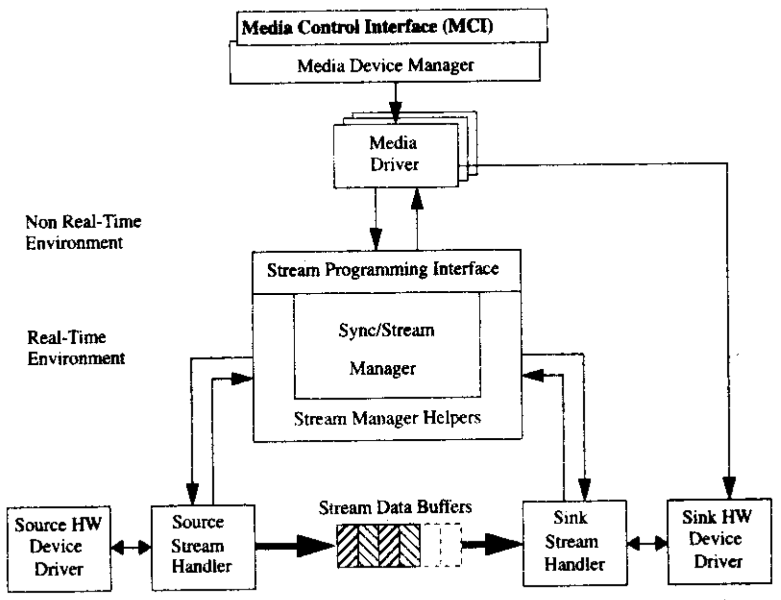

Figure 9.28: *The architecture of the OS/2 Multimedia Presentation Manager/2.*

- "Get table of contents of a CD-ROM" is an example of a device-specific command.

A logical device in MMPM/2 is a logical representation of the functions available from either a hardware device, a hardware device with software emulation or a software emulation only. The actual implementation is not relevant to an application because the MCI provides this device independence.

Examples of logical devices are an "Amplifier-Mixer Device," similar to a home stereo amplifier-mixer, a "Waveform Audio Device" to record and play digital audio, a sequencer device for MIDI-sounds, a "CD Audio Device" that provides access to audio compact disks (CD-DA), a "CD-XA Device" to support CD-ROM/XA disks and a "Videodisk Device" to control video disk players which deliver analog video and audio signals.

The multimedia I/O functions enable media drivers and applications to access and manipulate data objects that are stored in memory or on a file system. Storage system I/O processes handle the access to specific storage devices. File format I/O processes manage the access to data stored in file formats like "RIFF Waveform" and "BitMap." They use the services of the storage system I/O processes.

The implementation of data streaming and synchronization is supported by the Stream Programming Interface (SPI). It provides access to the SyncStream Manager that coordinates and manages the data buffers and synchronization data. Pairs of stream handlers implement the transport of data from source to sink.

Ease of use is supported in MMPM/2 on several levels. The installation of programs and setup of devices is supported by unified graphical user interfaces that centralize these functions for easy access. Also, a style guide for applications ensures that there is a common look and feel of applications that correspond to this guide. There is a high degree of flexibility because application developers and device providers can integrate their own logical devices, I/O processes and stream handlers. So, new media devices, data formats, etc. can be integrated in MMPM/2 and can be used by every application using the MCI.

OS/2 with MMPM/2 is a platform that has some basic operating mechanisms to support the processing and presentation of multimedia information as it is needed in multimedia application scenarios. It incorporates an RTE, implemented as a set of device drivers. MMPM/2 is an advanced platform for the development of these multimedia applications, providing the media and stream abstractions.

Finally it should be pointed out that MMPM/2 and WME look very similar and have many concepts in common.

9.8 Concluding Remarks

In this chapter we addressed the major issues of operating systems related to multimedia data processing, namely, resource management, scheduling and file systems. This discussion includes the most relevant existing architectures of such systems.

The concepts employed by current multimedia operating systems have been initially used in real-time systems and were adapted to the requirements of multimedia data. Today's operating systems incorporate these functions either as device drivers or as extensions based on the existing operating system scheduler and file systems. As a next step, an integration of real-time processing and non-real-time processing in the native system kernel can be expected.