Problems and Chances of Real-Time Data Processing in UNIX-like Operating Systems

Peter Buchheim / Johannes Schacht
Institut fuer Angewandte Informatik
Technische Universitaet Berlin / Electronic Modular Systems

ABSTRACT

In our age of automation and data communication real-time data processing has increased considerably. Formerly users implemented their application on naked hardware. Today they demand real-time operating systems so that safer and quicker development becomes possible - which of course implies a decrease of expenses. Following this trend a UNIX-like operating system with real-time features seems to stand a fair chance of selling extremely well.

There are various ways of trying to develop such an operating system. This essay describes the difficulties and introduces the different methods of approach. Afterwards one system is described more closely.

1. Introduction

1969 UNIX (TM)* was developed as an operating system in Bell Laboratories. Because of its clear structure which proved to be of great advantage to the developers of programs, UNIX circulated quickly in Bell Laboratories, and soon spread to the universities.

However, it was not before the end of the Seventies that UNIX was really wide-spread - after interested firms and then AT&T itself had distributed it with the necessary support. This development was encouraged by the decision of leading software firms to use UNIX as standard development system.

* UNIX is a trade mark of Bell Laboratories
UNIX was not planned as a real-time operating system. The developers of UNIX probably never thought of using their operating system in such a way. The wish to be able to develop and operate real-time systems under UNIX rather originated with thrilled UNIX-users who did not want to miss the valuable development environment when working on real-time tasks. The wish can be explained by the fact that there is no real-time operating system available on the market that combines any of the advantages of UNIX, as for instance portability, powerfulness and wide circulation.

From a technical standpoint it is difficult to give reasons for equipping UNIX with real-time features. Seen technically, the best solution would be the development of a new operating system with real-time features. It would be wise to evaluate the experiences made with UNIX and other operating systems during the last 15 years, and make use of them when developing such a new operating system. That would solve the problem of having to be compatible on one level (e.g. the system call level). If the demands of real-time data processing are taken into consideration when planning the new operating system, the realization should give more satisfaction than the supplementary remodeling of an existent operating system. However, this is not the subject of the essay, but I do think these aspects should not be ignored.

2. Real-Time Requirements

The expression 'real time' has been mentioned several times. It is used frequently nowadays. People demand the real-time capability of UNIX. Operating systems with real-time kernels are presented. But what does 'real-time capability' mean? I shall examine the question in this chapter without claiming to undertake an all-round investigation.

2.1. Response Time

DIN 44 300, No.161 defines real-time operation as the operating of a computer system, where programs for processing data are ready for service permanently and the results are at disposal after a certain period of time.

This period of time is also called response time. Put differently, the response time is the time the real-time system needs to react to an event in the real world or in the computer itself.

The response time is determined by the specifications of a system. It may be the result of physical laws or of an optional determination. To be able to guarantee the response time, it is necessary that the user's program runs fast enough and is scheduled as soon as the event has occurred. The operating system is responsible for this. Here we have come upon the first approach for thinking about real-time capability.

A general-purpose real-time operating system should, of course, enable response times which are as short as possible. And there is a further demand: To be able to check whether a system keeps to a certain specification it must be possible to calculate the maximum response time exactly.
2.1.1. Context-Switch Time

The Context-Switch Time is that characteristic of a real-time operating system which is spread most widely. The period of time defined by the expression 'context-switching' is the time between the end of the performance of one, and the beginning of the performance of another process. This period of time is generally not added to one or the other process. It is needed to identify the new process, to safeguard the context of the old process and to generate the context of the new process. Therefore users of a real-time operating system want those periods of time to be as short as possible.

2.1.2. Critical Regions

Critical regions are program parts which must not be interrupted whilst working. Critical regions are unavoidable in programs which compete for mutual resources and/or communicate with one another. An operating system contains critical regions. No switch of context is allowed when such a program part is being executed. Therefore it is important to know the length (period of time) of the longest critical region existing.

2.1.3. Memory Residence

The knowledge about the context-switch time and the length of the longest critical region is not sufficient to guarantee a certain response time. If the process image is not resident in the memory it has to be loaded first. The amount of time needed for such an operation is too large for several applications. Nevertheless loading must not exceed a certain period of time even when dealing with applications which have long response times. It is difficult to calculate the time, especially if one has to load from a disk. The period of time varies amazingly. Its length depends on hardware characteristics, on the position of the heads and on the physical distribution of the process image on the disk. Therefore it seems sensible to ask for real-time operating systems with the ability of keeping the processes memory resident.

2.2. Programming Environment

Real-time systems have to work on various tasks simultaneously. Accordingly they are always implemented as multi-tasking systems. Thus demands arise which a developer of purely sequential programs does not have.

2.2.1. Interprocess Communication and Synchronization

Interprocess Communication is needed for data transfer (e.g. status or measure data) between co-operating processes. Interprocess Communication has to be reliable and should be capable of eliminating error situations, i.e. in case of an error (e.g. awaiting of a message which will never come) a process must not be delayed endlessly.

When competing for mutual resources processes have to be synchronized. When competing for a printer, for instance, a good operating system should handle synchronization. However a general-purpose real-time
operating system cannot manage peripherals which are different from system to system. This task must be handled by the user's programs. The synchronization of processes is another problem for which an interprocess communication is necessary.

2.2.2. Time Management

The operating system must provide a real-time clock and be capable of executing tasks at specified times.

2.3. High Reliability

Real-time operating systems must be extremely reliable. They should be implemented with a tolerance towards errors. After having detected an error they should eliminate it (if it's a temporary error like a transmission error). If elimination should prove impossible (as with a permanent error) the effects of the error should be reduced to a minimum.

Quite a number of errors -especially those detected in connection with the peripheral- can only be handled satisfactory by users' programs. A real-time operating system should be prepared for this possibility.

2.4. Hardware Access

A typical requirement in the programming of real-time systems is the direct access to the hardware. The hardware of conventional operating systems consists of terminals, disks, tapes and the like. For safety reasons the operating system does not allow any of the user processes to access the hardware directly, and serves the user an abstract model of these devices instead. (In UNIX, for example, all devices appear as (special-) files to a user process.) The abstraction is possible, because the devices are more or less similar.

In real-time applications, however, the hardware components that have to be controlled often vary extremely. For that reason the operating system cannot anticipate the structure of the I/O system, and the application process has to be provided with the means to access the hardware directly. This immediately creates a conflict with the safety aspects of an operating system, and is completely unacceptable to a multi-user operating system like UNIX.

3. UNIX as Real-Time Operating System

The ordinary UNIX like version 7, system III or system V cannot be used as a real-time operating system without modification, because the response time cannot be guaranteed. To achieve real-time capability UNIX either has to be modified or run in a different environment.

3.1. Rewriting UNIX

All UNIX processes are the same: They have a variable priority assigned dynamically by the kernel. The user only has little influence on
the priorities via NICE-value. UNIX would have to be modified in a way enabling the user himself to determine priorities.

The UNIX scheduling concept seems very suitable for real-time activities, since context-switching takes place after every interrupt if necessary. There is, however, one exception: There is no context-switching if a process is running in kernel mode when the interrupt occurs.

Processes in UNIX alternate between user mode and (in case of a system call) kernel mode. Kernel mode processes cannot be interrupted unless they call up 'sleep' themselves. Some UNIX system calls are of a considerable length. For that reason it is intolerable for real-time applications that they cannot be interrupted. Hence there must be modifications to enable interrupts of kernel processes but excluding certain critical regions from this operation.

These are the most important alterations of those necessary. Especially the alteration mentioned last represents a severe interference with the kernel. It is impossible to anticipate all consequences straightaway. Thus the anticipation of the expenditure of work becomes very difficult. There is another disadvantage: The whole procedure has to be repeated every time a new version of UNIX is issued.

3.2. Writing UNIX New

The best solution would be the development of a new operating system with real-time features. The final product could not be called UNIX any longer, but it would possess UNIX features on a level specified before (syscall level or command level). Expenditure would be enormous, and the compatibility with later UNIX releases could only be guaranteed at great expense, too. The market has seen some examples of this approach, but they can either not satisfy in a real-time area or in respect to UNIX, because of great compatibility problems.

3.3. UNIX as Real-Time Process

Another possibility would be the implementation of UNIX as a process of a real-time operating system. The real-time processes would then be managed by the real-time operating system directly, and one of the processes scheduled would be UNIX itself as a whole (including the kernel and the user processes). This scheme is similar to IBM's VM operating system where different operating systems (DOS/CMS) work under one supervisor operating system (CP).

If this concept is followed strictly all real-time tasks are separated from UNIX (they don't appear in a 'ps' command), and run at a higher priority than the UNIX processes. The result would be more or less the same as that presented in the next chapter - except for the fact that the real-time processes run on separate processors there.

3.4. The Shifting of Real-Time Processes to a Slave Processor

This method of approach does not alter the UNIX kernel. As usual the kernel runs on a processor. This approach allows an additional processor which has a real-time kernel and works on real-time tasks.
On the hardware level the processors can communicate via interrupts and shared RAM. On the process level UNIX and real-time processes communicate via special file. A special device driver has to be integrated for that purpose.

This method of approach has several advantages:

(-) UNIX is not altered. The system running is the original UNIX. So there are no problems when a new version of UNIX is released.

(-) Real-time processes do not become a burden to UNIX.

(-) If a good real-time kernel is chosen for the second processor all problems created by the rewriting of UNIX can be avoided. Real-time processes run in a most favourable environment.

(-) Experiments (like the testing of real-time software) can be made during the phase of development without running the risk of the whole system crashing, if hardware means have been established to prevent the real-time processor from accessing the rest of the system.

4. A Multi-Processor Example

This chapter presents a model of a real-time system implemented by moving the real-time task to specialized processors, and leaving the overall system control and maintenance task under the control of the UNIX operating system. The division of a whole application into two parts (where one part can be implemented to run under a real-time operating system, and the other—which does not need real-time response—can run under UNIX) can often be achieved very easily. As an example take a measuring system: The manual operator can start a measuring task (UNIX), then the system has to sample a huge amount of data in a very short time without data loss (real time), and afterwards the data is evaluated (again UNIX).

4.1. Hardware

The system in question is a VMEbus system. Since you can connect several processors to the VMEbus it is ideal for realizing the project. The UNIX system consists of a CPU board with local RAM, a hard-disk controller and a board for serial and parallel I/O.

To be able to tackle real-time tasks one or more additional CPU-boards are put onto the VMEbus. These CPU-boards are also equipped with a local RAM and a real-time clock.

The real-time boards can handle interrupts as well as receive interrupts in a special way. There is an interrupt register on this kind of board. When this register is written into, an interrupt is generated on the board. This interrupt mechanism is used for the communication between the UNIX- and real-time CPU.
Each real-time CPU can generate an interrupt on the VMEbus too. This mechanism is used when a real-time CPU wants to interrupt the UNIX CPU.

The memory of all CPU boards can be addressed by way of the VMEbus. The UNIX CPU can read and write into the memory of each real-time CPU. Real-time CPUs have similar possibilities, but a certain address space can be blocked for them by a PROM. This is important to protect the UNIX memory from being written over by a real-time process.

4.2. Real-Time Kernel

The system described here uses the real-time kernel pSOS-68K (TM)* from Software Components Group Inc., Santa Clara, to run on the 'real-time CPUs'. pSOS is a multi-tasking operating system kernel. It offers the following facilities:

(-) Process Management

(-) Memory Management

(-) Interprocess Communication

* pSOS is a trade mark of Software Components Group
With pSOS the user has an enriched message-passing-model from Hansen at his disposal. Here messages are not bound to processes, but sent per exchanges. Several exchanges may exist. One or more processes can send messages to one exchange or receive them. Generally spoken, this model represents an n-to-m process communication. All typical interprocess communication tasks (sending/receiving data, synchronization and mutual exclusion) can be implemented easily with this scheme.

A process is also associated with seven events. Each process can indicate one or more events by means of signal_v. A process can await one or more events by calling up wait_v.

The event system is not as flexible as the message-passing-system, but easier to handle and more efficient.

(-) Time Management

(-) I/O System

Before talking about the connection between the two (operating) systems a brief description of the pSOS real-time kernel is necessary, since the latter is not so well known as the UNIX operating system.

4.2.1. Process Management

From the pSOS standpoint a process is the smallest program unit which can compete for resources itself. pSOS establishes an environment for processes in which they can use all resources of the system without regard for other processes.

Processes can be created dynamically. Process priorities are determined by the user and can even be changed during execution. Process scheduling is tackled by round-robin procedure, priority basement or both. Processes are sorted in respect to their priority, so that no search has to be undertaken to get the next process to a running state, if a context switch occurs.

4.2.2. Memory Management

pSOS-Memory-Management is realized by three calls: alloc_seg, free_seg and assign_seg. Memory can be called up with alloc_seg, and is distributed in accordance to the first-fit algorithm. Free_seg sets a segment of the memory free, and assign_seg hands over the segment to another process.

4.2.3. Interprocess Communication

Two different mechanisms for interprocess communication are available. There is a message-passing system and an event system. Strategies for avoiding deadlocks are not part of the pSOS-kernel. They have to be realized by the user.
4.2.4. Time Management

The time management provides the processes with date and time. Moreover, it realizes the timeout times of the system calls, and places an absolute wait call at the systems disposal.

4.2.5. I/O System

The I/O system in the pSOS kernel is a very thin layer. It only serves to define a standardized interface between user processes and device drivers. The structure of the drivers is similar to the structure of UNIX drivers. A driver is selected by a major device number. A minor device number is passed on to the driver by the call. A driver consists of six routines: device_init, device_open, device_close, device_read, device_write and device_control.

5. The UNIX-pSOS Connection

This section describes in detail the connection between the UNIX operating system and the pSOS real-time kernel over the VMEbus. The description is divided into three parts: Part One discusses the various possibilities of connecting operating systems. Part Two gives an idea of how the connection can be used, and the final part discusses implementation details.

5.1. Basic Design of the UNIX-pSOS Connection

Three possibilities of connecting the operating systems have to be considered: A) a connection by means of the interprocess communication facilities of the operating system, B) a connection over the networking facilities of the operating system, and C) a connection of the I/O systems.

5.1.1. ICP-Connection

It seems natural to connect the interprocess communication facilities, because communication takes place between two processes, and because data can be transferred from one process's address space to the other process directly in a tightly coupled system like the connection via a computer bus.

The interprocess communication facility of pSOS is the message exchange. A pSOS process sends and receives messages (4 longs of net data) to/from a certain exchange. Interprocess communication facilities have been established in UNIX recently - in release V. They include three different means of communication: shared memory, semaphores and message passing.

The connection of the interprocess communication facilities of the operating system would have the following effect: When a pSOS process sends data to a certain exchange, the data appears as a message in a certain message buffer of UNIX. This concept has several disadvantages. 1) pSOS uses its message exchanges to solve three different kinds of interprocess activities (data transmission, synchronization and mutual exclusion), whereas UNIX has at least two different kinds of interprocess com-
munications for that purpose (messages and semaphores). 2) The implementa-
tion includes a modification of the UNIX kernel to enhance the semantic of
the interprocess communication operations, and 3) the interprocess facili-
ties are not used by standard UNIX tools and programs currently. It is
not possible for a UNIX utility to redirect its input from a message queue.

5.1.2. Network Connection

The connection of operating systems using the network facilities
appears to be an attractive solution, however, pSOS does not offer any
network features, and UNIX does not offer a unique network concept. The
socket concept in the Berkeley-UNIX environment could be used to imple-
ment a pSOS network connection. Such a connection would have the fol-
lowing advantage: The same communication scheme could be used, if the two
systems were connected by means other than the VMEbus. However, as the
system we are considering includes the standard AT&T-UNIX, we have decided
to connect the two operating systems by way of their I/O system.

5.1.3. I/O Connection

The connection of the operating systems over their I/O systems means
that each OS takes the other as a peripheral device. In both operating
systems peripheral devices are characterized by a major and a minor device
number specifying a device type and a unit respectively. In UNIX the dev-
ice numbers are obtained from the file system as a 'special file'. For the
connection of the I/O systems the write operation of a process in one
system is interpreted by the OS in a way that a read operation in the
other OS gets the data that has been written.

Advantages of this communication scheme are threefold: The I/O system
of an OS (UNIX and pSOS are no exceptions) can be modified easily. Modifi-
cations of the UNIX I/O system like adding a device driver can be achieved
without modifying the kernel, and additionally, it does not require an
AT&T source licence - a commercially important point. As all devices are
accessed through the UNIX file system, the UNIX file protection mechanism
applies to the UNIX-pSOS connection.

Another advantage of the I/O system is the fact that the data is
moved by the operating systems. If, for example, the message passing system
of pSOS is used, the amount of data sent or received at once is four long
words. If large amounts of data are to be moved within pSOS, the informa-
tion passed normally consists of pointers which point to data themselves.
Moving these pointers from one system to the other causes a problem, as
local addresses within a system (e.g. a VMEbus board) differ from global
addresses (the base address of the VMEbus board itself must be added as an
offset). The whole problem disappears, if the I/O system is used for com-
unication, since data is moved directly.

The discussion of the pros and cons of the different approaches may
lead to the wrong impression, that the decision for one of them excludes
the other possibilities completely. In fact, only the basic interface
between the operating systems is affected by this decision. If, for example,
it becomes necessary to use the message-system for communication, it can
easily be put on top of the I/O system. A pSOS process would send data to
a specialized message exchange which is served by a process that sends the
message to the other operating system by way of the I/O system. This
implementation is completely transparent to the sending process.

5.2. Using the UNIX-psOS Connection

Assume, a psOS process generates ascii data and sends the data to the
I/O port connected to the UNIX system. In the UNIX system the data can be
received with cat /dev/psOS assuming that /dev/psOS is installed with the
appropriate major/minor device numbers. To bring the data onto the
printer the simple command sequence pr < /dev/psOS/lpr is sufficient.

For large-scale control tasks a UNIX program can simply use the UNIX
'open2Y system call to open the channel to psOS, and then read from and
write data to the psOS system.

From the psOS side the communication works much the same. psOS
processes open the device attached to UNIX and execute read/write supervi-
sor calls. It will often be necessary for psOS processes to have access to
resources of the UNIX system. As those resources most often are an
integral part of the UNIX file system (data files, printer, terminals,
IEEEbus), a simple server process giving access to the UNIX file system is
sufficient for a large group of applications. If a psOS process wants to
open a UNIX file, it calls the (library-) function 'u_open', that sends the
open request (via UNIX-psOS connection) to a UNIX process serving this
request. The server process executes the 'real' UNIX system call and passes
the result back to the psOS process. Read/write requests are executed in a
similar way. More demanding requests from psOS to UNIX, say queries from a
data base, must be served by more specialized server processes.

Another task often required in a multi-processor application is pro-
cess synchronization which can be achieved directly by the I/O link. A
process on one side of the link (either UNIX or psOS) can wait for an
event on the opposite side by simply reading from the I/O link. If there
is no data, the process is deferred until a write from the other side
occurs.

More sophisticated synchronization tasks include semaphores and
mutual exclusion. They can be implemented by a specialized message
exchange on the psOS board. A server process in psOS allows UNIX processes
to send and receive messages to resp. from a psOS exchange. If multiple
psOS systems are connected to each other, each operating system wanting to
access an exchange remotely has to have a remote server process. The
remote exchange scheme provides all synchronization means necessary in a
multi processor environment.

A last feature of the UNIX-psOS connection should be described
separately: the so-called PRobe channel. PRobe is a debugger with special
enhancements to debug psOS applications. If processes are executed under
the control of PRobe, the operator has the possibility to look at process
states, message queues and the like. When a UNIX user connects his terminal
to the PRobe channel logically, he has complete control over the real-time
processes. This feature is especially useful during the test phase of pro-
gram development for the real-time system, as debugging takes place
without having to use a serial channel of the real-time processor.
5.3. Implementation of the UNIX-pSOS Connection

5.3.1. Moving Data

The read/write operations of each operating system are the central action of information transfer. When a process executes a read system call, the events following depend on two circumstances: (a) whether a process on the opposite side already has a write operation (with enough data to satisfy the read request) on the same channel, and (b) whether the channel was opened to operate in blocking or non-blocking mode.

A read request in both operating systems specifies the channel to be used (coded by major/minor device numbers), the number of bytes to be read, and the addresses of data location. If sufficient data is available from a previous write on the other side of the channel, the data is transferred to the specified address, and the read operation is terminated. If no (or not sufficient) data is available, further action depends on the channel mode. In non-blocking mode the read system call returns immediately indicating that only part (or none) of the read request could be satisfied. It is then up to the process that executed the read to proceed appropriately (e.g. retry). If the channel operates in blocking mode, process execution is suspended until enough write operations from the other side have occurred, so that the read can be satisfied. Afterwards the waiting process is prepared for running again.

The write system call operates in a similar way. If a read request on the other side is already pending for the amount to be written, the system call is executed. If no read is pending, the channel mode flag decides, whether the writing process is to be suspended or should return indicating its disability to transfer data.

It should be obvious from the discussion above that successful data transfer is impossible if both sides try to operate the channel in non-blocking mode, as it is very unlikely that both sides execute the system calls at exactly the same moment. The proper set-up of the channel is ensured during the open system call to the channel. The open system call returns successfully, if a process on the other side is already waiting for the channel to be opened. In case there should be no process waiting on the other end, the open request either will return without having been successful or will block – depending on the mode specified for the channel (blocking/non-blocking).

If a process at one end of a channel is waiting (due to a read, write or open request), it has to be informed that the other side of the channel has acted. This is done by interrupting the operation at the other end. The interrupt service procedure can then wake up the process waiting. More about interrupts in a later paragraph.

5.3.2. Global Considerations

Assuming the co-existence of a UNIX system and several pSOS boards on one VMEbus, communication should be possible between UNIX and each pSOS board, as well as between all pSOS boards. Since addressing requires a major/minor device number only, it is completely transparent to the com-
municating processes whether they are accessing another pSOS board or the UNIX system. Each system is processed by one VMEbus board. Some boards are configured to have access to other VMEbus boards; others are configured for local resources. For the latter there is only one way of attracting the attention of other systems: by executing a VMEbus interrupt.

If two systems (i.e. two VMEbus boards) want to communicate, at least one of the boards needs global VMEbus access in order to move the data from one system to the other.

The distinction of links and channels has been introduced to manage the different aspects of inter-system communication. A link can be set up between two system boards. Whether it is possible to set up a link to another system depends on: the accessibility of the system (physical availability, system software initialized and running), getting the permission from the other side, and as mentioned above the VMEbus access of at least one of the systems. When establishing the link the decision is made as to which of the two systems is responsible for the actual data transferring.

Max. 16 channels can be operated on each link. A channel is the main communication path between two processes. It can be opened, closed and operated independent from all other channels. Thus several independent communication processes can take place between two systems. A link is established automatically by the first open of one of the channels between two systems. At the same time the place of residence is fixed for the channel descriptors. The system not responsible for actual data transfer has to have the channel descriptor inside his local memory. The channel descriptor contains various information e.g. whether a process is waiting for data to arrive.

5.3.3. Initialization

During the powering up of the system the UNIX kernel (or more exactly the initialization procedure of the device driver) reads in the physical characteristics of the VMEbus system from the file system. The information includes the VMEbus addresses of the real-time boards, and information on the capability of a board to access the VMEbus.

With this scheme the addition, removal or reassignment of addresses to boards does not imply the recompilation of either operating system. The only thing necessary is the editing of a UNIX file.

If one of the pSOS operating systems executes its start-up phase (which may happen either before or after UNIX has come up), the communication initialization procedure gives an interrupt to the UNIX system. Some time after its own initialization is completed UNIX is ready for processing the interrupt, and sends the information on the VMEbus configuration to the requesting board. So after all boards have come up, they have enough information to establish links to one another.

5.3.4. Interrupts

The most critical part of UNIX-pSOS connections are the interrupts. It is possible that several processes on a board try to access one or more
VMEbus boards at the same time. However, it may be impossible to execute an interrupt immediately, because the interrupt handler procedure on each board takes quite some time to be completed. For this reason, processes may be deferred until they gain access to their local interrupt resource. Hereby an acknowledge interrupt is automatically introduced with the purpose of waking up sleeping processes.

Another limitation that has to be overcome is the number of interrupt levels on the VMEbus. The VMEbus only has 7 interrupt levels, and some of them could already be being used by the UNIX system for its own resources (terminals, disks etc.). Therefore, each real-time board has a local (mailbox) interrupt, which is triggered when local memory location 1 is written to by another VMEbus board. Unfortunately, this mechanism can only be used by processes that have global VMEbus access.

A special interrupter process must be introduced into the system. The process (which can be located in any of the systems) can be interrupted by a VMEbus interrupt, and will then post that VMEbus interrupt to another board as mailbox interrupt.