RIJKSUNIVERSITEIT TE GRONINGEN
MATHEMATISCH INSTITUUT

GUTS

a multi-user operating system
for the pdp 11/40

Wim Bronsvoort

TW-200 deel I

GUTS

Wim Bronsvoort

This project was undertaken during the closing phase of my study under the supervision of Prof. Whitfield.

Inquiries may be addressed to:
Department of Computer Science
Postbus 800
9700 AV Groningen

December 1978

Index.

1) Introduction.

Groningen University Timesharing System (GUTS) is the somewhat ambitious name of the operating system which will be described in this document. Although it is a system with some attractive properties, it was never meant to become generally used at our university, as the name might suggest.

1.1) History.

The first work on GUTS was done in 1975 during a course on operating systems. Prof. Whitfield wrote the first version of the kernel and a number of project groups consisting of students wrote some processes which made use of this kernel. In 1977,two more students worked on the system and in the meantime two student-assistants did some development work on it too. I worked on the system both as a student and a student-assistant.

In the autumn of 1977 we had a system, entirely written in assembly language , which did the synchronization in a tidy way, but was not really usable. There was no loader, no compiler or assembler and the system itself had too many deficiencies to be attractive. At that moment we obtained a compiler for the high-level implementation language IMP for the PDP11. I was looking for a project to complete my study and we decided to install the compiler under the (old) GUTS system and to redesign, rewrite in IMP and, for the first time, to properly document the system.

The system running now is much more attractive than previous versions, but certainly not finished: it can be improved and extended in many ways.

1.2) Type of system.

GUTS is a multi-user, interactive operating system for PDP11 computers with memory management, i.e. several users can work from consoles on the system at the same time. It is not designed for batch processing or real time applications.

We will call "the system" that part of the software which is permanently resident in main memory: the kernel and the supervisor processes. The "subsystem", which is part of the code of every user process and consists of the command interpreter and basic I/O routines, will also be described, but programs which can be run by user processes, such as editors, compilers, etc., are not described in this document.

1.3) Motivation for writing the system.

When the development of GUTS started it was as a teaching project with the following aims:

- learning about synchronization and other operating system principles
- learning to manage a somewhat more complicated program.

Gradually the system became too complicated and in fact unmanageable, mainly because it was written entirely in assembly language. It certainly could not be presented at a new course on operating systems as an example of how an operating system should be written.

The main intention of my project was to learn more about designing and implementing an operating system, and in this way to manage complicated applications in general.

The main objectives chosen for the system were the following:

- -simplicity: it should be shown that a system, which is really usable, need not be complicated. Besides, simplicity increases the adaptability and might improve the performance of the system too.
- -it should be usable as an example at courses on operating systems. This means that the system should be easy to understand from the source-listings and the documentation. I have tried to write the system in a clear style, inserting useful comments.
- -the system should be efficient. Experience with the widely used UNIX system ([4]) showed that the performance of this system is not so good and that it is certainly possible to write a more efficient system.

These objectives have been chosen at the cost of generality. I hope this objection is not too serious, but still more do I hope that the objectives mentioned have been realized.

1.4) Hardware environment.

GUTS is written for the PDP11/40 computer, which consists of a processor connected to one or more storage units and peripheral controllers via a bidirectional parallel communication line called the "Unibus". For details see [1] and [2].

It is a sixteen bit word machine, i.e. instructions and virtual addresses have this length. A word is divided into two bytes, so data can be eight or sixteen bits long. The processor incorporates a processor status word (psw), containing information about the status of the program running, a program counter (pc), two stack pointers (sp) and six general registers. It can operate in two different modes: kernel mode and user mode. The choice of the mode determines which set of memory management registers is used to convert program virtual addresses into physical adresses, which stack pointer is used and whether certain instructions e.g. "halt the computer", are forbidden or not. For more details and the instruction set, see [1].

The minimal hardware requirements for GUTS, besides the processor, are:

- -112 K bytes of main memory, which we will call "core", although it need not really be core memory, to prevent confusion with "virtual memory"
- -a memory management unit; see next sections
- -a decwriter; more consoles can easily be inserted into the system
- -a KW11-L or KW11-P clock
- -an RK05 disk.

The memory management unit converts sixteen bit virtual or program addresses into physical addresses of eighteen bits. It consists of two sets of registers, the active page registers. For each of the two modes there is a set of these registers. Each set is composed of eight pairs, each consisting of a page address register and a page descriptor register. Each pair controls the mapping of one page, i.e. 8K bytes of the virtual address space. Each page consists of a maximum of 128 contiguous blocks of 64 bytes each.

Any virtual address belongs to one page or other. The corresponding physical address is generated by adding the relative address within the page to the contents of the corresponding page address register.The contents of the corresponding page descriptor register determines for which addresses in the page, if any at all, a mapping is valid and whether or not it is allowed to write into the page. Furthermore the register contains a bit indicating whether or not the page has ever been written into. Any attempt to reference an invalid address, i.e. an address for which no mapping is set up, or to write into a page which is protected from writing, is trapped by the hardware.

The memory management unit provides for enlargement of the usable core memory (up to 256 K bytes) and protection against users reading or modifying parts of core outside their own areas. Alas, the design of the unit is far from ideal, as will be shown in the chapter on the file system (chapter 8).

1.5) Implementation language.

As mentioned previously, the first versions of the system were entirely written in assembly language and were assembled by the MACRO assembler running on the RT11 system ([3]). The kernel and the subsystem of the current version are still in assembly language, but all supervisor processes are written in IMP.

IMP was designed as the implementation language for the Edinburgh Multi-Access System ([5]), but is now in use as a general- purpose programming language on many machines. Although it preceded Pascal, it can be described as "Pascal-like". It offers tidy program and data structuring facilities and is easy to understand. For a description of the language, see ([6]).

The advantages of using a higher level language are enormous: programs become more structured, more readable and are easier to maintain. When you use such a language, you are less concerned with details and so make fewer mistakes while programming. The time you win in this way can be used for thinking about the real problems. GUTS is another proof that programs like operating systems can be written in a higher level language, a fact admitted by too few people.

The only problem with the current compiler is that it is not reliable: strings and byte integers cause a lot of trouble, but also in other cases faulty code is sometimes generated or the compiler blows up. Parts of programs for which faulty code was generated could always be rewritten in such a way that the compiler generated correct code. The difficulty was to localize the parts at which the compiler failed. Writing a correct program which does not work is one of the worst things which can happen to a programmer. I call the current version of IMP a higher level language with a lower level compiler and sincerely hope we will shortly get a better implementation.

2) General structure of the system.

To understand the system requires a knowledge of the general features of operating systems like processes, synchronization, resources and scheduling. The book "Operating System Principles" ([7]) can serve as an introduction to these.

2.1) Hierarchal structure.

GUTS can be considered as consisting of a number of levels, each level making use of the facilities offered by the levels below it. This division was made to understand the working of the system better and to give an idea of how the different parts are related.

The levels, in increasing order, are:
- the hardware
- the direct memory accesses by interrupt routines
- the kernel for the basic synchronization
- the supervisor processes
- the subsystem, i.e. the command language interpreter, loader and basic I/O routines
- the user programs.

The hardware has already been described in the previous chapter. The general principles of the other levels will be described in the rest of this chapter, while the description of the internal working of them will follow in later chapters.

2.2) Direct memory access by an interrupt routine.

The term direct memory access is normally used to indicate access to main memory by a device, e.g. the disk, without the processor being involved. Here it is used to indicate access to main memory by an interrupt routine without the rest of the system being involved, i.e. the system continues at the exit of the routine in the same state it was at the time of the interrupt. The idea is to achieve software simulated DMA to reduce kernel activity.

These interrupt routines, running at a non-interruptable priority, are used for actions which are performed very often and are simple and short. Only when there is an exceptional condition will it be necessary to inform a supervisor process of this, e.g. when the counter in a clock interrupt routine gets a value less than or equal to zero. This is done via the kernel by converting the interrupt into a "signal" on a semaphore or by sending a message to the supervisor process. This can also influence the continuation of the system. Generally there will be no exceptional condition and there will be a simple return from the interrupt, continuing the system in the state it was prior to interrupt. Handling interrupts in this way is very efficient.

2.3) Kernel.

The kernel is in fact the basic part of the system, needed to implement the idea of a process. It contains a description of the state of every process and a despatcher, which selects the next process which is going to run. If there is no other process to be run, the idle process is selected.

The primitives for process interaction are implemented by the kernel too. Supervisor processes can request a "signal" and a "wait" on a semaphore, a "send message" and a "receive message". Supervisor and user processes can request a "supervisor call", i.e. a "send message" combined with a "receive message" to receive the reply to the request issued by sending the message. The use of semaphores by user processes would require an allocation strategy for these and the use of "send message" is dangerous because this could cause a message buffer overflow when too many messages are sent without receiving the replies sent by the supervisor. Besides, the use of these primitives by user processes would make the supervisor more complicated, because a second request could be sent to a supervisor process before the first one is serviced. Some interrupts from devices, handled by the kernel, are converted into a "signal" or a "send" too.

The kernel is implemented as a critical region, i.e. the execution of one call cannot be interrupted by another call. This is achieved by running the kernel on a high priority and is needed because the operations performed need to be mutually exclusive.

The code for the idle process is located in the kernel too, although it is not really part of it and certainly does not run on a non-interruptable priority. Furthermore there are a number of useful routines: a trace routine, which can monitor on the system console messages sent and or received by one or more processes (this is very convenient during the development of the system), a dump routine, useable if the system crashes, and a bootstrap routine for the RT11 system, which is used for loading GUTS.

To read more about the idea of a kernel, consult the book by Hansen ([7]), where the term "basic monitor" is used for it.

2.4) Supervisor processes.

The supervisor is the part of the system which enables users to share the resources of the system such as:

-processor time
-core
-disk space
-consoles.

It consist of a number of processes:

-the clock handler, offering time services
-the disk driver, handling the disk
-the core manager, keeping track of used and free space in core
-the file system, allocating files, i.e. collections of data, on the disk and making these files accessible to users. It also contains routines for the starting and stopping of user processes (login and logout) and resetting such a process to a required state when there has been an error trap (recovery)
-the console handlers, each handling one console
-the scheduler, arranging the sharing of core space and processor time amongst the users.

User and supervisor processes can send requests for a service to a supervisor process. All services have a unique number which must be specified when sending a request. This number is converted into a process number by the service exchange mechanism in the kernel, which makes it possible to hold the service number fixed while changing the number of the process handling the request. When a supervisor process receives a request, it depends on the service number which routine of the process is executed. Not all services can be requested successfully by user processes, e.g. the disk can not be read from or written to directly by a user, and all parameters of a call are checked carefully.

The supervisor processes have to set the source of a message themselves when sending one. This enables them to send a request on behalf of another process or to pass on a request.

The supervisor processes are permanently in core. They are protected from being read from or written to by a user process by the memory management unit.

All supervisor processes have the same general structure: they are in a never ending loop, receiving a request, handling the request and mostly sending a reply immediately. Before the loop is entered, some initialization of variables is done. So all supervisor processes look like:

```
initialization
%cycle
   ......
   receive (message)
   ......
   ......
   send (reply)
   ....
%repeat
```

The variable "flag" is always used to indicate the result of the request; only if "flag" has the value zero at the moment the reply is sent, has the request been handled successfully. The supervisor processes extensively use the primitives offered by the kernel to interact with other supervisor processes. (The "<u>control</u> k'100001'" statement in the system sources indicates that multiplication, division etc. should be executed by the hardware, i.e. the Extended Instruction Set option should be used, and that the program is trusted, i.e. certain checks, like switch index checks, are not executed.)

2.5) <u>User processes</u>.

A user process is started for every user who is admitted to the system. It has at its disposal a virtual memory of 64 K bytes: addresses generated by the processor when the process is running are 16 bits long. These virtual addresses are converted into physical addresses by the memory management unit after the appropriate mapping for this conversion has been set up by the system.

It is not necessary to understand what a user process does to be able to understand the system. Every user process could in fact arrange things in its own way. For simplicity and convenience however, a standard subsystem has been chosen as part of the code of each user process. This subsystem consist of a command interpreter, which determines the interface of the system to the user, and basic I/O routines, and is located in segment 7. Segment 6 is partly used as data area for the subsystem and further as data area for programs to be run. Segment 0 and 1 are used as I/O segments (see the description of the file system). These (arbitrary) decisions have had some influence on the design of the system too.

The user processes, which are meant to run programs, can get access to the resources of the system by sending requests to the supervisor. So you could say that every user has at its disposal a 16 bit virtual computer.

3) Direct memory access by an interrupt routine.

There are direct memory access interrupt routines for two purposes:
- to update a counter for the clock handler
- to put a character into a console output buffer.

The second one will be dealt with in the chapter about the console handler (chapter 9), because there is a lot of interaction between the routine and the handler. The first one will be discussed now. The routines are run non-interruptable priority, to ensure mutual exclusion of the interrupt routines and mutual exclusion of an interrupt routine and a kernel call. They are in kernel mode, so that the kernel memory management registers, which are set to map a virtual address onto the location with the same physical address, are used. The assembly language code for the routines is located in the code for the kernel.

3.1) Clock interrupts.

Depending on the available clock, the routine "kwpint" or "kwlint" is used to update the variable "count". This counter is used by the clock handler to indicate the number of milliseconds to pass before it has to be waked up. If an KW11-P clock is used, 1 is subtracted from "count" every interrupt, because this clock gives an interrupt every millisecond. If an KW11-L clock is used, 20 is subtracted from "count", because the frequency of this clock is 50 Hz. As soon as the counter becomes equal to 0 in the routine for the KW11-P clock or less than or equal to 0 in the routine for the KW11-L clock, a message to the clock handler is generated. This is done via the kernel and informs the handler of the fact that the requested interval is over, so that some action has to be performed. See chapter about the clock handler (chapter 5). If no message is sent, there is an immediate return from the interrupt routine.

4) Kernel.

The kernel consists of a number of routines, handling the entry and exit, the execution of the primitives for process interaction and the selection of the next process to be run. There are also some routines which were useful during the development of the system.

4.1) Implementation language.

The kernel is written in assembly language. Some parts have to be, because one has to address the registers and execute certain instructions which cannot be executed in an IMP program. Other parts could be written in IMP, but this would not make it clearer, because of the interaction required between the different parts. Because the kernel is a critical region, the code for the primitives should be as efficient as possible, which is another reason for writing it in assembly language.

4.2) Priority, mode and mapping of the kernel and the processes.

The kernel, like the direct memory access interrupt routines, runs at a non-interruptable priority to ensure mutual exclusion of different calls and mutual exclusion of a kernel call and an interrupt routine, and in kernel mode, so that the kernel memory management registers are used. These are set to map the virtual addresses in the segments 0-6 onto the locations with the same physical addresses and the virtual addresses in segment 7 onto the area where the device registers are located (the 4K words with the highest possible physical addresses), so that these registers can be addressed via segment 7. The idle process runs at the lowest possible priority in kernel mode. The kernel memory management registers are never changed after initialization.

Both supervisor processes and user processes run at a low priority to enable all devices to interrupt them. These processes run in user mode. Because the user mode has its own stackpointer, the kernel stack is reserved for use by the kernel. The memory management registers used are the user active page registers. For a supervisor process they are set to the same values as the kernel mapping registers. The value of them is never examined, so they never have to be saved. Each user process has its own mapping, which must be copied to the memory management registers every time the process is selected to run. In addition, the contents of the page descriptor registers must be saved every time the process is stopped, to enable the supervisor to examine the written into bit of a segment. When a supervisor process is selected, the user memory management registers only have to be set when the previously running process was a user process, because otherwise the previously running process was another supervisor process, having the same mapping.

4.3) Data area.

The data area of the kernel contains the following variables:

- "process", containing the number of the running process
- "apd", which is a pointer to the process description of the running process
- "runproc", indicating the user process which will be selected to

run when no supervisor process is ready to run
- an array of semaphores,consisting of records with the fields "counter" and "waiting"
- the process table "proctab", which contains an entry for each process with the following information:
    - the state of the registers r0-r5,sp,pc and psw
    - the head of a queue of received messages ("recd")
    - a variable indicating whether or not the process is waiting for a message ("heldup")
    - a variable indicating whether or not the process is waiting for the receipt of a reply to a "supervisor call"; if so, the variable contains the number of the service it is expecting a reply from ("asleep")
    - a variable indicating whether or not messages to or from the process should be monitored, i.e. printed on the console ("rsmoni")
- the queue of supervisor processes which are ready to run ("readyq") with 2 pointers ("begrdy" and endrdy")
- "recproc", containing the process number of the process receiving the message being sent
- "suppar" and "suppdr" containing copies of the supervisor "page address" and "page descriptor registers"
- "userapr", containing copies of the "active page registers" for each user process
- "messtab", which is an array of message buffers
- "free", which is a pointer to the list of free message buffers
- an array of records with the fields "link" and "info" used as elements of lists for several purposes
- "asl", which is the head cell of the list of free records in this array
- "find", which contains a pointer to the process descriptor for each process
- "intflag", indicating whether a send comes from an interrupt or from a process
- "callsch", indicating whether the scheduler should be called when the idle process is selected to run
- "psave", which contains the process number of the process running before the kernel was entered.

4.4) Entry.

Entry to the kernel is possible in several ways:
- by a process executing an emulator trap instruction. This is only permitted when done by a supervisor process. Depending on the number of the trap, a "signal" or a "wait" on a semaphore, a "send message" or a "receive message" is executed.
- by a process executing the iot instruction. This is permitted to both supervisor and user processes and causes a "supervisor call" to be executed.
- by an interrupt from a device.
- by a process trying to execute a non-executable instruction.

In all cases the priority is automatically set to a high value by the loading of a new processor status word from the vector address of the interrupt or trap instruction. This causes the kernel to be non-interruptable.

The routine "save" is used to save the state of the registers r0-r5,sp,pc and psw of the running process in the process table. For a user process, the page descriptor registers are saved too by the routine. For the idle process nothing is saved. There are some simple calls of the kernel, in which only a few registers are used and where the same process continues after the exit from the kernel, for which this saving is unnecessary, and therefore is not done.

4.5) Wait and signal on a semaphore.

Each semaphore consist of two components:

- -"counter", which defines the number of signals sent, but not yet received
- -"waiting", which is a queue of processes waiting to receive signals not yet sent.

For both "wait" and "signal", r0 contains the number of the required semaphore.

"Wait" does the following: if "counter" is greater than zero, it is decreased by one and the calling process continues; otherwise the calling process is entered into "waiting" by the routine "enter" and the despatcher is called to select another process.

When a "signal" is executed the following happens: if one or more processes are in the "waiting" queue, the first one of them is removed from it by the routine "remove" and the process with the highest priority continues; otherwise "counter" is increased by one.

4.6) Service exchange.

The use of the service exchange mechanism has already been explained in the chapter about the general structure of the system (chapter 2). The implementation of it is very straightforward.

For both the "send" and the "supervisor call" operations the right byte of r0 contains the requested service number. The number of the process supplying the service is fetched from the array "serv", which maps a service number onto a process number. If a non-existing service is requested, which is noticed by a negative value of the entry in "serv", this is handled in the same way as when an attempt is made to execute a non-executable instruction. See section on error handling (section 4.11).

For supervisor processes the left byte of r0 contains the source of the request. The two bytes are exchanged when the message is copied to the receiver area or a buffer. For a request from a user process, the left byte of the first word of the message is set to the service number after the copying too, but the right byte is set equal to the service number of the user process. This prevents the user processes from sending messages on behalf of another process.

4.7) Send message.

Messages are contained in the registers r0-r5 when the sending process enters the kernel. Now there are two possibilities for the process to which the message is sent:

- -it is waiting for a message, i.e. "heldup" is set, or it is waiting for a reply from the sending service,i.e. "asleep" is equal to the service number of the sending service

-it is not waiting for a message or it is waiting for a reply from a service different from the one sending the message.

In the first case the message is written immediately by the routine "copy" into the space reserved for copies of the registers r0-r5 in the process-table entry of the receiving process. The receiving process can continue now. The priority of the receiving process and the sending process are compared to select the process with the highest priority as the running process.

In the second case the message is copied to a message buffer by the routine "qmess", which gets a buffer from the list of free buffers ("free") by the routine "remove", copies the message into it and enters the buffer into the list of received messages of the receiving process (routine "enter"). The sending process continues.

4.8) Receive message.

When a process wants to receive a message, there are two possibilities again:

-there are messages in the list of received messages of the process
-there are no messages in this list.

In the first case the first message is removed from the list by the routine "unqmess", which gets it out using the routine "remove", writes the message to the copies of r0-r5 in the process table entry of the process and returns the message buffer to the list of free buffers using the routine "enter". The receiving process continues.

In the second case "heldup" is set to indicate that the process is waiting for a message. The despatcher is called to select the running process.

4.9) Supervisor call.

When a process issues a "supervisor call", it is suspended until a reply to the request is received. The service to which a "supervisor call" was issued, is remembered in the "asleep" field of the process-table entry of the sender. Only a reply from this service can wake up the process.

The sending part of the "supervisor call" is nearly identical to that of the "send message" primitive, except that when a user process does the call, the source of the request is set by the kernel.

The process issueing the supervisor call does not continue until the reply is received. So a new process has to be selected by the despatcher.

4.10) Interrupts.

Interrupts can occur from a number of devices:

-the clock
-the disk
-the consoles

Interrupts from the clock were discussed in chapter 3. The sending of a message to the clock process from the interrupt routine, is done by the routine "intsend", which looks like the normal send routine. Destination and source, which are set equal, and two more words of the message can be set by the interrupt routine. The variable "intflag" is set to indicate a send coming from an interrupt.

Interrupts from the disk ("diskint") are converted into a "signal" by putting the disk semaphore number into r0 and jumping to the "signal" routine.

Interrupts from the consoles are sometimes converted into a "send message", sometimes into a "signal" on a semaphore and sometimes handled immediately. See the chapter on the console handler (chapter 9).

4.11) Error handling.

When there is a power failure ("powint"), the system is stopped.

Trying to execute a non-executable instruction (e.g. an attempt to move a word to an odd address or to address a part of the virtual memory for which no mapping is set up), a request for a non-existing service or an attempt to execute the halt instruction or an emulator trap instruction by a user pocess, are all handled in the same way.

A supervisor process making an error causes the system to stop after printing all the process registers.

If a user process makes an error, a "supervisor call" to the "reset process state" service is generated, containing the trap number. The process goes to sleep until this service sends a reply. Before sending this reply, the "reset process state" service resets the values of r5,sp and pc in the process-table entry of the user process to values previously specified by this process and disconnects all connected files. See chapter about recovery (chapter 10).

The code of the error causing the processor to trap is obtained from the processor status word loaded from the trap vector. The routine "error" converts this code into the trap number via the array "ernumb". This trick, copied from the UNIX system ([4]), is used by the console interrupt routine to identify the console.

4.12) Despatcher.

The despatcher is the part of the kernel which decides which process is going to run. It consists of two routines:
- "resched", used to compare the priority of two processes, both ready to run, select the one with the highest priority and indicate that the other one is ready to run
- "select", used to select, from all processes ready to run, the process with the highest priority.

The following priority rules are used:
- when there are one or more supervisor processes ready to run, the one with the lowest process number is selected
- when there is no supervisor process ready to run, the running user process ("runproc", which is set by the scheduler) is examined. If this is the idle process, it is checked whether

the scheduler should be called to see if any user process has become ready to run (variable "callsch", which is set whenever a message is sent to a user process); if the scheduler is not called, the idle process continues. If the running user process is not the idle process, it is checked whether the process is asleep or not; if it is, the scheduler is called to select a new user process to run, otherwise the process itself continues.

The supervisor processes which are ready to run are in a queue, which is implemented by putting all ready supervisor processes into a contiguous part of the array "readyq". The processes in it are always in the order of increasing process number, so that getting the one out of it with the highest priority is very simple: you take the first. Inserting a process into the queue is more difficult: because the queue must stay in order, this may take some shuffling of part of the array. The routine "intread" is used for this.

To see how the selection of the user process is done, read the chapter about the scheduler (chapter 12).

4.13) Exit.

When the "exit" part is entered, the variable "process" contains the number of the process selected to continue.

The exit for the idle process is the simplest. The program counter and program status word are put onto the stack and a return from interrupt is executed.

For other processes the memory management registers are set first, at least when this process is not the same as the previously running process or both this and the previously running process were supervisor processes. See section about the mapping of the processes (section 4.2). The registers r0-r5 and the stack pointer are set, the program counter and program status word are put onto the stack and the RTI is executed. The values of the registers are taken from the process table entry for the process.

4.14) Idle process.

The idle process is very simple: it executes a "wait" instruction. In this way no use is made of the bus, which is advantageous when a data transport between the memory and the disk is going on.

4.15) Further routines.

The routine "rmoni" is used to monitor, on the system console, all messages received by processes for which the "monitor receive" bit is set in the process-table. The routine "smoni" is used to monitor all messages sent by processes for which the "monitor send" bit is set.

The routine "dump" is used to dump areas of memory. It should be started from the switches, which are used to set begin and end address of the dump area too.

The routine "bootrt" is used to bootstrap the RT11 system and should be started manually too.

5) Clock handler.

The clock handler uses the KW11-P or KW11-L clock to keep track of the time and to send messages to processes after requested intervals. The selection and initialization of the clock is done at the initialization of the kernel. See chapter about initialization (chapter 13).

5.1) Messages from the clock to the handler.

To wake up the handler at the appropriate time there is a variable named "count" in the kernel which is adjusted by the handler every time this is needed. It indicates the number of milliseconds to pass before some action has to be performed by the handler. Every time an interrupt from the clock occurs, "count" is updated. As soon as "count" becomes less than or equal to zero, a message to the handler is generated. See chapter on direct memory access by interrupt (chapter 3).

5.2) Services.

The following services are implemented:

0) reply after a certain interval
1) get current time
2) set current time.

The last service can only be requested by a system process, the other two by all processes. The get current time service immediately gives a reply; the first service sends a reply after the requested interval if this interval was legal, otherwise it immediately sends a refusal.

5.3) Main part of the handler.

The most important information for the handler is held in:

- -the "software clock" named "time", which is an integerarray containing the current time expressed in milliseconds, seconds, minutes, hours, days, months and years
- -the recordarray "waiting", which contains information about processes which are to have a reply from the handler after some time
- -the integer "requested interval", which contains the number of milliseconds which have past at the moment "count" becomes 0 since it was reset by the handler for the last time.

Every time a request for a service is received "time" is updated. If the counter in the kernel has become less than or equal to 0 without the handler having received a message for it (this can have happened while executing the first instructions of the cycle of the handler) the routine "interval over" is used for this update and it is remembered that an extra update took place, so that the message generated because "count" became 0 can be ignored. If the counter is still positive the only things to be done are to update the milliseconds time and the requested interval. Hereafter the requested service routine is called.

If a message generated because "count" became 0 is received and there has been no extra update the routine "interval over" is called.

5.4) Requested interval over.

The requested interval is over when "count" has become 0. This means that at least one of the following things is going to happen:

- at the update of "time" (routine "adjust time") the milliseconds time gets the value 1000 after adding the requested interval to it, so that other elements of "time" are also changed (when an element in "time" becomes equal to a certain maximum value, e.g. 60 for seconds, the next element in "time" is incremented by 1 and the element itself is assigned a certain minimum value, e.g. 0 for seconds)
- one or more messages to a process must be sent at this moment. The routine "test waiting queue" checks the waiting queue to see which processes (if any) must have a message at this moment, sends them the current time and removes them from the waiting queue.

After the adjustment of "time" and checking of the waiting queue, "count" and "requested interval" must be reset. If there is a process which must have a message before the milliseconds time becomes 1000 again the new values depend on the time this message must be sent, otherwise the values depend on the time till the milliseconds time becomes 1000 again. The new requested interval is added to "count", while interrupts from the clock are disabled by putting the processor at high priority, (routine "adjust count") and the value of "count" is checked to see if it is negative. If so, the interval is already over and the same things are done again.

5.5) Reply after a certain interval.

A request for a reply after a certain interval is first checked to see if the requested interval, which is expressed in milliseconds and seconds, is legal: the milliseconds time must be bigger than or equal to 0 and smaller than or equal to 1000, the seconds time bigger than or equal to 0 and smaller than or equal to 60 and not both times may be 0. If it is a legal request the time at which a message must be sent back is determined. This time is expressed in a milliseconds time bigger than or equal to 0 and smaller 1000 and a seconds time bigger than or equal to 0 and smaller than 60.

Now the request has to be put into the waiting queue. This is a fairly complicated matter because we want to hold the queue in such an order that requests which have to be satisfied first are at the head of the queue. See routine "message after time".

To put the request into the waiting queue the routine "into waiting queue" is used. The relevant information (process identification and time at which a reply must be sent) is put into a new record. If the new record does not become the first one in the queue the rest is simple list processing, otherwise it is necessary, at least if the new request must be satisfied before the present second ends, to adjust "count" and "requested interval". The difference between the new and the requested interval is subtracted from "count" (routine "adjust count"). If "count" is negative now the time is already over and "interval over" is called. It is even possible that "count" was already negative before subtracting the difference between the new interval and the requested interval from it. Then there has been a message generated which has not yet been received and which must be ignored when received ("extra updates" is incremented by 1), because "count" will certainly be negative after the subtraction, so "interval over" will be called and here it will be established that the time at which the message was generated is already over and so things which had to be done at that moment will be done now.

5.6) Get current time.

The get current time service is simple: a copy of the array "time" is sent as the reply.

5.7) Set current time.

The given new time is first checked to see if all values are between a certain minimum and maximum value. Extra attention has to be paid to the day in the month, because the maximum value is variable here. Leap-years are ignored: for the month february the maximum day is always 29. If no error is detected the new time is set into the array "time".

Setting the time disturbs the correctness of the information in the waiting queue, so it should only be done at the startup of the system.

6) Disk driver.

The disk driver is meant for an RK disk. The discipline used for handling the requests is the first come first served discipline.

6.1) RK.

One RK control can service up to 8 drives each handling a disk of 4872 blocks of 512 bytes. Average total access time on each drive is 70 msecs. All data transfers are direct memory access transfers, i.e. the processor is not involved during a transfer.

The RK disk is controlled by 6 registers:
- drive status register
  not used by the driver
- error register
  used to discover which error occurred
- control status register
  used to set interrupt enable and the required function and to start this function
- word count register
  used to indicate total number of words to be affected by a given function (in two's complement form)
- current bus address register
  used to indicate the bus address to or from which data will be transferred
- disk address
  used to indicate the drive and block to or from which data will be transferred.

For a detailed description of the RK disk and its registers see [2].

6.2) Serving a request.

Four different requests to the driver are possible to transfer a specified number of blocks of 512 bytes to or from core:

0) read from the disk to a specified block (512 bytes) in core
1) write to the disk from a specified block in core
2) read from the disk to a specified address in core
3) write to the disk from a specified address in core

The first two services are used to handle file transfers between core and disk, the other two to handle transfers of the directories and small buffers in the system area. Files are located in the user area of core, which is the (higher) part of core left after allocating the system code and data areas, and always start at the beginning of a block. The physical start address cannot be expressed in 16 bits, but takes 18 bits. So it is easier to specify the address in block numbers. Buffers in the system area can start at an arbitrary address which can be expressed in 16 bits, because the system area is located within the first 56 K bytes of memory.

A request is first checked to see where it comes from: when a user has sent it, it is refused and an appropriate reply is sent. The contents of the different registers are determined and the requested function is started. Interrupt enable is set, so that an interrupt will occur when the function has completed its activity. To ensure continuation of other processes while the function is still in progress the driver does a "wait" on the disk semaphore. When the interrupt occurs it is converted into a "signal" on the disk semaphore in the kernel, so that the driver may continue.

When the driver continues, after the function has completed its activity, the control status register is checked to see whether an error occurred. If so, the error register is examined to see whether it was a write-lock-out-violation error, which means that a disk was on write protect while attempting to write on that disk. If such an error occurs an appropriate message is printed on the system console; in case of other errors all disk registers are dumped in octal on the system console. When any error occurs a control reset is executed and the requested function is tried again. This makes it possible to get the disk into the right condition (e.g. by manually putting the disk into the write permit status) and perform the request without the originating process having to reissue it. When the driver continually gives errors other than write-lock-out-violation the only thing that can be done is to halt the computer.

After successful completion of the function a reply is sent to the process which made the request. One of the parameters sent back is an identifier, which is a word received with the request and which is left unchanged. It can be used by the process to identify different requests which are being sent to the disk driver. This facility allows other service disciplines to be used without affecting the rest of the system.

6.3) Service discipline.

The rule implicitly applied for handling the requests is the FCFS (First Come First Served) rule: when a request is received a possible next request is not received and looked at until the present one is completed. Different rules like the SSTF (Shortest Seek Time First) rule, which selects as next request for service the one having the track address closest to the current position of the head, the SCAN rule, which applies the SSTF rule in one direction only and reverses the direction when there are no further requests ahead of the head position in the given direction, and the FSCAN rule, which is like the SCAN rule, but arrivals during a given scan are placed into a queue and not serviced until the next scan, have better response times but also disadvantages such as discrimination against certain blocks of the disk and complexity of the algorithm (see [8]). Nevertheless, because the disk transfers are an important factor in the total system performance, a better discipline than the FCFS rule would probably improve the performance when the system is heavily used.

7) Core manager.

The core manager keeps track of used and free space in core. Core is divided into blocks of 512 bytes, which is more natural with a disk with blocks of that size than to take blocks of 64 bytes as the memory management unit assumes. The first fit algorithm is used to find a free area large enough to satisfy a request for a certain amount of contiguous core.

7.1) Services.

Two services are offered by the core manager:

0) get core
   search for a number of contiguous free blocks of core. If found then mark them as in use and send back the number of the first block (the start block), otherwise send back a refusal
1) release core
   the specified blocks of core are released and are marked as free again.

Both services are reserved for use by supervisor processes. It is assumed that these processes do not make any mistakes with regard to start block and number of blocks when releasing core, so this information is not checked.

7.2) List of free blocks.

Areas of core which are not used at the moment are in a queue of records named "area" ("free core list"). For each area it is remembered where it starts and how many blocks there are. The areas are kept in order of increasing start block. The queue is bidirectional, which means that every record contains a pointer to the preceding and a pointer to the succeeding record. It is also circular, which means that the last record points to the first record again. This is somewhat unnatural, because the first area of free core does not in any sense succeed the last area, but it makes the list processing easier. Records which are not used at the moment are in a queue of free records ("free records list").

7.3) Get core.

If the total number of free blocks is less then the requested number of blocks it is immediately clear that the request cannot be satisfied. Otherwise we go along the list of free areas to find an area large enough. If such an area is found and the total area is used, then the record can be returned to the free records list. If it is found and the total area is not used then the start block and length of that area have to be adapted. In both cases the requested length is subtracted from the total number of free blocks and the start block and length (=the requested length) of the allocated area of core are sent to the process which made the request. If an area large enough is not found, this is reported to the process.

7.4) Release core.

If the released area of core becomes the only free area, a new free core list is made. Otherwise the free core list is searched for the free area behind the released area. Now there are several possibilities:

- -the released area goes with the preceding and the succeeding free area
- -the released area goes with the preceding area
- -the released area goes with the succeeding area
- -the released area cannot be combined with any surrounding area, so that a new record has to be setup for it.

In all these cases the number of released blocks is added to the total number of free blocks.

7.5) First fit.

The first fit algorithm is used to satisfy a request for a certain number of contiguous blocks of core: the first free area of core with size larger than or equal to the requested size is (partially) used. Although one might expect the best fit algorithm, which searches for the smallest free area of core with size larger than or equal to the requested size, to increase the probability of being able to satisfy subsequent requests, the first fit algorithm appears in practice to be better (see [7]). The algorithm is also simpler.

The search for a large enough free area always starts at the end with the lowest block number. This tends to accumulate the smaller free areas at that end and so to increase the search time for larger free areas. A better strategy might be to start the search at different points in the free list every time.

The memory management unit requires that contiguous areas of core are used to map segments onto (see chapter 8 about the file system). This causes some serious problems. The available core is split into used and free areas of different lengths, so it is possible that a free area large enough to satisfy a request cannot be found, although the total number of free blocks is large enough. These problems are solved by the scheduler.

7.6) Initialization.

At the initialization of the core manager (routine "initialize") the total number of free blocks and a record containing the first free block and the total number of free blocks have to be filled in. The relevant information to do this, is got from two external integers in the initialization part of the total system.

8) File system.

The file system has two basic functions:

- -the creation,deletion,renaming,etc. of files
- -making the contents of files available to user processes.

The structure of the file system on the disk is fully RT11 compatible. When a process wants access to a file, the file must be connected to the virtual memory of that process, i.e. a mapping must be set up.

8.1) Files.

A file is a sequence of bytes. The length of the sequence is an arbitrary multiple of 512, the block length on an RK disk. The internal structure and contents of a file are of no interest to the file system, so a text file, a code file or any other file is handled in exactly the same way.

8.2) Structure of the directories and allocation of the files.

RT11 is a single user operating system developed by DEC for the PDP11 series of computers. RT11 was used for the development of GUTS. For more information about this system see [3].

On GUTS the structure of the directories, i.e. the areas on the disk that contain information about which files are on the disk and where they are, is exactly the same as on the RT11 system. Files on GUTS are, like RT11 files, contiguous on the disk, i.e. all blocks of a file follow each other. In this respect the systems are fully compatible.

There were two reasons for choosing this strategy:

- -transfers of several blocks of 512 bytes from or to the disk are considerably faster when these blocks are contiguous in core and on the disk, because several blocks can be transferred by issueing one request to the disk. When, on the contrary, these blocks are scattered across the disk, as many requests have to be issued to the disk as there are blocks and the total seek time on the disk will increase considerably.
- -during the development of GUTS it was very convenient to have an identical structure for both file systems, because the same disk could be used by both systems.

Contiguous allocation of files on the disk has some very serious disadvantages, because the available room on the disk is split into areas of different lengths:

- -it is possible that the total number of free blocks is large enough for the placement of a file of a certain length, but the largest number of contiguous free blocks is too small for it, so that the file cannot be placed
- -once the initial (maximum) length is fixed, the file cannot be extended, because it cannot be guaranteed that there are free blocks behind the already allocated blocks.

There are a number of possibilities of preventing or solving these problems:

- -choose a good allocation strategy for the creation and initial allocation of the files on a disk (see sections about creation of files).

-release parts of files which are not used (see description of closing files).
-choose a good allocation strategy for the creation and allocation of the files on the different disks when using several disks. One could e.g. have one disk on which all created files are initially allocated. When the system is started this disk should be empty and all files which have to be preserved should be copied to another disk during system time or after the system stops.
-stop the system and compact the disk, i.e. copy all used blocks to the beginning of the disk, at the moment this is needed or ahead of this moment. On RT11 this is done by the compress service of the peripheral interchange program. This is a very radical solution, because stopping the system is very unpleasant for the people using the system at that moment.
-compact the disk, without stopping the system, at the moment this is needed or ahead of this moment. On a multi-user system like GUTS this is a very complex operation, because besides changing the directory and copying the files, which is done at the previous solution too, several tables in core will have to be changed and the system will be unable to perform some operations during the time of the compaction. This solution is not yet implemented and should not be chosen before it is proved to be really needed.
-always have available enough disk storage to have no trouble. Compaction could then be done when the system is stopped. This is of course the most ideal but also the most unrealistic solution.

For a time we considered the following solution: divide the disk into blocks of 16 contiguous blocks of 512 bytes, which constitute one segment (see section about memory management), and allocate files in these blocks. If a file is longer than such a block, use several of them and link them one way or the other. This would to a large extent solve the allocation problems. But we could not think of a way to stay fully RT11 compatible when using this strategy, so we rejected it.

At the moment of writing it is not yet clear how serious the described problems are. Experience with the UNIX system ([4]), which uses scattered allocation of one block at a time (and so can use all blocks of the disk before a file cannot be allocated any more) instead of contiguous allocation, shows that the performance of such a system also deteriorates when the disk becomes nearly full, so problems seem to be inevitable. How serious the problems with our allocation strategy are and whether a drastic solution, like compaction without stopping the system, should be implemented, will have to be shown in practice.

8.3) File names.

To be RT11 file system compatible requires that you also have, more or less, the same convention for the names of the files . The names in the directories of RT11 are in the RADIX 50 notation, which permits only the characters space,'a'-'z',dollar,dot and '0'-'9', gives these characters a value, multiplies the value of the first character of three by 1600 (=40*40;40=50 octal), the value of the second by 40 and leaves the value of the third unchanged and adds these 3 values. In this way 3 characters can be stored into one 16-bit word. GUTS uses this convention too and file names sent to the file system are assumed to be already in RADIX 50 notation. In this way less space is needed for the parameters. The conversion can just as easily be done by the subsystem as by the system.

RT11 file names consist of 2 parts:

- -user defined name of the file (1 letter followed by 0-5 letters or digits)
- -user defined extension, which gives an indication of the contents of the file (3 characters).

So a total of 9 characters is used for a file name. GUTS uses these 9 characters in a slightly different way.

The name is divided into 3 parts:

- -user defined name (1 letter followed by 0-5 letters or digits)
- -user defined extension (1 letter)
- -identification of the owner of the file (2 letters).

The user defined name is written first, followed by a dot and the extension. The last part of the name consists of the owner identification between square brackets. So legal file names are e.g.:

```
list1.i[wb]
copy.o[sy]
```

For every request to the file system the user defined name and extension must be fully specified. For some requests (e.g. create file) the owner identification is filled in by the system, for other requests (e.g. connect file) this identification has to be specified too. Parts of the name can of course be set by the subsystem, so that the user can e.g. ask for the program "copy" and let the subsystem change this name to "copy.o[sy]" before calling the file system. See the description of the subsystem (chapter 14).

8.4) Owner of a file.

The user who creates a file is the owner of that file. Part of the file name consists of the identification of the owner (see previous section). The owner can offer a file to another user and this user can accept the file or not. In this way it is impossible that a file is transferred between users, without the new owner wanting it.

8.5) Access permissions.

Access to a file is possible in 4 different modes:

0) read unshared
   only one user at a time may read from the file
1) read shared
   several users may read from the file simultaneously
2) read-write unshared
   only one user at a time may read from and write to the file
3) read-write shared
   several users may read from and write to the file simultaneously

When several users want to connect a file in shared mode, this mode must be identical for all.

To access a file in a specified mode the user must have permission to connect the file in that mode. The owner of a file can independently set the permissions for himself and for others. These access permissions consist of 4 bits each: 1 bit for each possible mode, indicating whether access in this mode is allowed or not.

8.6) Services.

The following services are offered by the current file system:

-create file

Create a file with the specified name, which must be distinct from all existing file names, and length. The length is specified as follows:

- -either 1/2 the largest unused area or the entire second largest unused area, whichever is largest
- -the largest unused area
- -M blocks

The maximum size is always 250 blocks. The owner gets permission for all modes and others get no permission at all. The three possibilities for the specification of the length are offered to reduce the allocation problems of files on the disk.

-close file

Set the final length of the file to a specified value. This value must be less than or equal to the length allocated when the file was created. A file can be closed only once. When it is not closed, it is deleted at a compaction of the disk. Closing a file can only be done when the file is not connected by any user.

-delete file

Delete the file from the system. This can only be done when the file is not connected by any user.

-rename file

Rename the file from the old name to a new name. The new name must be distinct from all existing names. The file must not be connected by any user.

-set access permissions of file

Set the access permissions of the file for the owner and for others to the specified values. The file may not be on offer to another user.

-get file names and information

Give a list of all files the user owns and information about these files, such as length, access permissions, date of creation, date of last access, number of accesses and offer/accept status.

-offer file

Offer the file to the specified user. The file may not be connected by any user and cannot be used any more until it is accepted. The owner can revoke the offer until the new owner has accepted the file; when this is done the owner gets permission for all modes and others no permission at all.

-accept file

Accept a file from another user: the owner part of the file name is changed to the identification of the new owner, who gets permission for all modes, while others get no permission at all.

-connect file

Make the file accessible by connecting it to the user's virtual memory. I.e. locate the file on the disk, read it (partly) into core and setup a mapping between the virtual memory of the user process and the part(s) of core the file is read into. The position in the virtual memory, the access mode, the start block within the file and the length of the part to be connected have to be specified. Several things can prevent a connect call from being executed successfully:

-the file does not exist or the user does not have

permission to access it in the specified mode
- the specified start block does not exist, i.e. the file is not long enough
- the required mode is in conflict with the mode in which the file is already connected in another virtual memory, i.e. the file is already connected in an unshared mode or in a different shared mode
- there is no room or not enough room at the specified location in the virtual memory.

If a connect is done on a file, which is already connected, and the new connect can be executed successfully, the part of the file already connected is first disconnected. The mode must not be changed however. When a call is executed successfully, the virtual address, the length of the connected part in bytes and the total file length in blocks of 512 bytes are returned to the user.

-disconnect file

Remove the mapping between the virtual memory and (part of) the file. The contents of the file, which is left on the disk after the disconnection, is of course consistent with the changes made by writing into it.

Create file, close file, delete file, rename file, set access permission of file and offer file can only be done by the owner of the file; accept file can only be done successfully by the user to whom the file is offered; connect file can be done successfully by any user who has permission to do it and disconnect file can only be done successfully when the file is actually connected.

8.7) Directory structure.

The directory on a disk starts at physical block 6; the first blocks are used for bootstrapping. It consists of a series of directory segments that contain the names, lengths, etc. of the files on that disk. The directory area is variable in length, from 1 to 31 directory segments; the number of segments can be specified when the disk is initialized. Each directory segment is made up of 2 physical blocks; thus, a single directory segment is 512 words in length.

A directory segment has the following format:
- a header of 5 words
- 46 entries of 11 words for files and unused areas
- 1 unused word.

The header contains the following information (all words):
- number of segments available for entries (between 1 and 31).
- segment number of the next logical directory segment. The directory is a linked list. This word is the link between logically contiguous segments.
- the highest segment currently in use. This word is updated and used in the first segment only.
- unused.
- block number where files in this directory segment begin.

A directory entry has the following format:
- status word. The following (octal) values are possible:

400 :	a tentative entry; indicates a file which has been created, but not yet closed
1000:	an empty entry; indicates an unused area
2000:	a permanent entry; indicates a file which has been

closed
4000: end-of-segment marker, which is used to determine when the end of the directory segment has been reached during a directory search.

-2 words for the name
-1 word for the extension and the owner name
-1 word for the length in blocks
-2 bytes for the access permissions for the owner and for others
-1 word for the date of creation
-1 word for the date of the last access
both these words are in the following format:
bit 15 unused
bit 14-10 month (1-12)
bit 9-5 day (1-31)
bit 4-0 year minus 72
-1 word for the total number of accesses
-1 word for the offer/accept services. When the file is not offered the word is zero; when it is offered and not yet accepted the user identification of the user to whom it is offered is stored in it; as soon as the file is accepted the word becomes zero again.
-1 unused word.

If files are added sequentially without deleting any files roughly one half the total number of entries will be filled before a directory overflow occurs. This results from the way filled directory segments are handled.

When a directory segment becomes full, it is necessary to open a new segment. Approximately one half the entries of the filled segment are moved to the beginning of the newly-opened segment. When files are only created and not deleted, then at the moment the final segment is full, all previous segments have approximately one half of their total entries in use. If this process were not done however and a file were deleted from a full segment, the space from the deleted file could not be reclaimed: every tentative file has to be followed by an empty entry for recovering unused blocks when the file is closed. Though only one file is deleted, two entries (a tentative and an empty one) are needed to reclaim the space.

The entries in the directories are in the same order as the files and unused areas on the disk. So to find the start block of a particular file or unused area one has to find the directory segment containing the entry for that file or unused area, take the start block number given in the header of that segment and add to it the length of each file and unused area in the directory segment before the desired file or unused area.

8.8) Advantages and disadvantages of this directory structure.

The advantages of this directory structure are:
-the names of often used system files can be put into the first directory segment and so, assuming the search for a file starts at the first segment, be located very quickly
-there is no separate free list for unused areas.

The disadvantages are:
-finding a file whose name is located in the last directory segment requires many more disk transfers than one whose name is

located in the first directory segment
- the names of the files of one user are not located together, so making a list of all files of one user requires searching the whole directory
- it is possible, when there are many small files on the disk, that a directory overflow occurs, i.e. the directory is full while the disk is not yet full. This is caused by the fixed number of directory segments.

8.9) Virtual memory of a user process.

For a description of the available memory management unit, see section 1.4.

The memory management unit is far from ideal for implementing nice virtual memory concepts such as segmentation, which divides the virtual address space into a collection of named, linear subspaces of various sizes (segments), and paging, which divides core into units of equal length, into which parts of files are stored (page frames). In a good system the virtual address space is very large, so that one can have a large number of segments which can be divided into a number of pages which are mapped onto page frames in core. Ideally we would like to be able to implement a demand paging system, which offers the possibility of running partially loaded programs. Pages are brought into core at the moment they are needed and not before. For a description of these concepts see [7] and [8].

Demand paging is impossible with the available memory management unit. When a page fault occurs in a demand paging system, it must be possible to determine what caused the fault. This requires that an instruction is first checked to see whether it can be executed successfully or not, before it is actually executed. The PDP 11/40 however just starts executing every instruction. When an address error occurs the state of the program (program status word,registers,etc.) is left as it is at that moment, i.e. it might be the state half way through the execution of the instruction, and some extra information about the reason the address error occurred is left in a special register. For some instructions it is impossible, even with a very complicated routine, to discover why an address error occurred and to reset the state of the process to the value it had before the execution of the instruction started. As an example consider the following instruction:

```
            mov (r0)+,(r0)+
```

When an address error occurs, this can be caused by fetching the source or by setting the destination. With the available information it just cannot be determined which part of the instruction failed and so the r0 register can never be reset to its original value.

The consequence of this is that all addresses which can be refered to during an instruction must be mapped onto core, i.e. all connected (parts of) files must be in core, before the instruction execution is started. This and the fact that the address space is only 64 K bytes long, cause a lot of trouble: the memory management of a user process, i.e. determining when information has to be moved into the virtual memory, determining where in the virtual memory it must be located and which parts of the virtual memory must be removed, must be left to the subsystem and possibly the user himself.

As said before, the virtual memory on a PDP 11/40 is 64 K bytes long, divided into 8 segments of 8 K bytes each. One segment must be mapped onto a contiguous area of core. To have convenient segmentation one must have more segments, which must be larger, because one wants to map a whole file onto one segment. One can of course consider the segments as pages, but this leaves one with one segment (the whole virtual memory) divided into 8, rather large, pages.

Our first idea was to take the 8 segments and use each segment to access a whole file, if the file was smaller than or equal to 8 K bytes, or part of a file, if the file was larger than 8 K bytes. Experience with an earlier version of GUTS showed that with this method the total number of accessible files was too small. As an example consider the second pass of the IMP compiler. One segment is used for the perm for IMP programs and the command interpreter, two segments are needed for data areas and three segments for the code. This leaves one segment for (part of) an input file and one segment for (part of) an output file. But this program needs one input file and two output files, which are used in turn. Every time the output file is switched, the currently connected output file has to be disconnected and written back to the disk and the other output file has to be connected and read from the disk. The overhead becomes too large when using this strategy.

If one wants to make accessible more files at one time, the only solution is to connect more than one file into one segment. The problem arising here is the sharebility of files. Segments have to be mapped onto a contiguous area of core. So when (part of) a file is connected to a segment to which there are also other (parts of) files connected, that file cannot be shared by any other user, because it is possible that this user wants to connect other (parts of) files too at the segment to which he connects that file. This is impossible, because core around the area, in which the file is located, is already in use. However, one wants to take advantage of the sharebility of files too, because sharing a file decreases the amount of core that is needed.

We decided to make the following division of the virtual memory. Segment 7 is used to connect (in shared mode) the IMP perm and the command interpreter (see section about user processes/subsystem), segment 6 to connect (in unshared mode) a data file for perm and command interpreter variables and the highest part of the stack area. Segments 2,3,4 and 5 can each be used to connect (part of) a file in unshared or shared modes. One (part of a) file can occupy several of these segments, but only a contiguous part of a file can be connected to a contiguous part of the virtual memory of the same length. Segments 0 and 1 are used as special I/O segments. The 16 available blocks of 512 bytes of these segments can be used to connect several files and to hold buffers for devices, such as a console. The following restrictions are applied:

- files in segment 0 are in read mode, files in segment 1 in write mode
- files in segment 1 must be connected in unshared mode (see previous section)
- a block in a segment can only be used for one thing at a time: to connect a file or as a buffer for a device
- the number of files connected at one segment cannot exceed three.

As described in the previous section, a contiguous area of core has to be used when one wants to connect several files into one segment. To avoid nasty allocation and copy problems, it was decided to connect a so called base file of length 16 blocks to each I/O segment when a user process is started. Files, which are connected into an I/O segment, are written into the base file. This has the disadvantage of taking core (remember that the file must be in core when the process runs) which perhaps is not used at once, but it makes the handling of the files and buffers, which are going to be located in the segment, easier. See also the chapter about the login service (chapter 11).

The chosen strategy of using segments in different ways is not really nice, but should be prefered to a much less efficient strategy as the one used in the earlier version of GUTS. Of course all the problems are caused by the hardware and the only thing one can do is to restrict the nuisance.

8.10) Write shared problems.

If a file is connected in write shared mode, the users themselves have to take care that meaningful results are achieved. Only when exactly the same part of the file is connected, can that part be shared in core by the users, so that changes made by one user can be observed by another. Even then the order, in which users write into the file, will determine the end result.

Because files connected at segment 1 can never be shared in core, it was decided to allow only write unshared mode for this segment. See previous section.

8.11) General implementation features.

All file services are handled by a single process: the file system process. This process handles requests one by one in first come first served order. Having one process has the advantage of needing only one process entry. Besides, mutual exclusion of operations on large parts of the data area of the file system is needed anyway, so having more processes has almost no advantages.

For operations on the directories there is a buffer named "dblock" into which a directory segment is read from the disk by calling the routine "getdblock". When the required directory segment is already in core, it is not read again. Every time information in a directory segment is changed, it is written back to the disk by calling the routine "writedblock". The predicate "file exists" searches a directory to see whether a file with a specified name exists. The routine "comprdblock" compresses a directory segment when there are two consecutive empty entries (can be combined into one) or when there is an empty entry following a permanent entry (the empty entry can be deleted).

Information about the files which are currently connected to a virtual memory of a user process and about the segments, to which they are connected, is held in several tables. In the kernel there is a table "userapr" containing copies of the page address and page descriptor registers of each user process. All files which are currently connected by a user process are in a list of records named "file" (header "files"). The following information is kept for each file:

-the name of the file
-the length of the file
-the disk address of the file

-the number of users of the file
-the access permissions of the file
-the mode the file is connected in
-a pointer to the list of connected blocks of the file, see description of "block"
-pointers to the next and the previous records in the list.

From a file several parts, which we will call blocks and which consist of up to 16 physical blocks of 512 bytes, can be connected at the same time. Because it is not necessary to have several copies of the name, mode, etc., there is, for each file, a list of records named "block" containing information about the connected blocks. For each block the following information is held:

-page address register for a segment the block is connected to
-page descriptor register for a segment the block is connected to
-start block in core where the block is located
-disk address of the block
-start block of this part within the file
-status word, indicating whether the block is in core or not (see chapter 12 about the scheduler)
-number of user processes which have connected this part of the file
-number of user processes currently in core which have connected this part of the file
-length of the block
-pointers to the next and previous records in the list.

The array "segmenttable" contains for each segment from each user process information about that segment. As mentioned in the previous section, segments 0 and 1 are called I/O segments and for these the following information is relevant:

-which blocks are in use and which are free
-number of files connected into that segment
-is the copy on the disk of the base file, i.e. the file which is connected when the process starts, updated or not. When the process is swapped out of memory and later swapped in again, we just want to read in the base file and not all (parts of) files which are connected into that segment. But this implies that the copy of the base file on the disk is updated every time the file is swapped out and something changed in the file. See also chapter 12 about the scheduler.
-for every file connected into the segment a so called subsegment entry, which contains:
    -a pointer to the file information
    -start block within the segment
    -number of blocks.

For all 8 segments the following information is kept:

-a pointer to the relevant file information
-a pointer to the relevant block information (which is an index in "block")
-a status word, used by the scheduler, to indicate whether a segment is assumed to be mapped onto memory or not.

Besides there is a field indicating whether or not segment 0 is requested to be in core by any handler. If this field is unequal 0, it indicates the handler which wants to copy input from its own buffer area to the user area. See section 9.14.

The predicate "connected" checks the list of connected files to see whether or not a file is connected to any virtual memory. The predicate "file in virtual memory" checks the segment table of a user process to see whether or not a file is connected to the virtual memory of that process.

More details about the implementation of the currently offered services are given in the next sections.

8.12) Create file.

There are 3 possibilities for the length of the new file: see section about services. After some checks, e.g. the file must not exist already, the directory is searched. For the first two possibilities the largest and second largest unused areas on the disk are located. When asking for a specific length, an unused area with length larger or equal this length is searched for. If such an area is found, we must enter information into the directory segment.

It is possible that the directory segment is full and a new directory segment has to be opened. For this the routine "extend" is used. If no more directory segments are available, a directory overflow error is reported, otherwise the following is done:

- -one half of the entries from the filled segment are put into the next available segment
- -the directory segment links are set
- -both segments are rewritten to the disk
- -the highest segment currently in use, which is a variable in the header of the first segment, is updated.

The links are needed when there are e.g. 3 segments in use and the second is full. Then a segment must be entered between the second and third segment and the links are used to logically rather than physically link a new segment between the second and third segment. After the extension of the directory, the position where the information about the file has to be inserted is located again.

Now the entry for the file can be inserted. When the previous entry is a tentative one, first an empty entry of length 0 must be inserted, because a tentative entry must always be followed by an empty one to reclaim the unused space when the tentative file is closed. After creating an entry, the relevant information (see section 8.7 for a description of the directory entries) is filled in and the file is made tentative. The entry is followed by the empty entry, containing the still unused part of the area the new file is located in. Finally the directory segment is rewritten to the disk.

When, during the search of the directory, the last directory segment has been done, there are two possibilities:

- -the search was for an unused area of a specified length. This means that the request cannot be satisfied.
- -the search was for the largest and second largest unused areas. In this case, it is determined which unused area is going to be used and the request is converted into a search for an unused area of a specified length, with a maximum size of 250 blocks, which is the maximum file size. The search for the selected area is restarted at the directory segment previously found.

8.13) Close file.

Closing a file is a simple operation. Besides the fact that the file must not be connected by any user at that moment, it must be checked that the file is not closed, i.e. made permanent, previously and that the requested length is less than or equal to the length allocated when the file was created.

The requested length is set for the file and the difference between this length and the allocated length is added to the empty entry following the entry of the file. The file is made permanent now and the directory segment written back to the disk.

8.14) Delete file.

Deleting a file from a directory is done by making the entry of the file an empty one, so that the area can be used again. The file must not be connected by any user at that moment. The altered directory segment is rewritten to the disk.

8.15) Rename file.

Renaming a file is only done when the file is not connected by any user at the moment and there is not yet a file with the new name. The new name is written into the entry of the file and the directory segment is rewritten to the disk.

8.16) Set access permissions of file.

The new access permissions are written into the entry of the file and the directory segment is written back to the disk.

8.17) Get file names and information.

For this service a so called directory file must be specified. Into this file a list with the relevant information (see section 8.7 for a description of the directory structure) about all files, belonging to the user, is written. This directory file must exist and be long enough, otherwise an error is reported.

The whole directory must be searched. When a file belonging to the user is found, information is written into a buffer of 512 bytes named "buffer" by calling the routine "putword". The routine "transfer buffer" writes the buffer to the disk when it is full.

8.18) Offer file.

For an offer of a file, the offer/accept word in the entry of the file is set to the identification of the user to whom the file is offered. Both access permissions are set to zero. For a revocation of an offer, the offer/accept word is set to zero again and the owner gets permission for all modes. The file must not be connected by any user at the moment. The altered directory segment is written back to the disk.

8.19) Accept file.

When a user wants to accept a file from another user, the offer/accept word in the entry of the file must contain his identification. If this is the case, the owner identification is changed to his. The new owner gets all permissions and the offer/accept word is again set to zero.

8.20) Connect file.

The virtual memory of a user process consists of 8 segments, 2 of which are I/O segments. Connecting a file into an I/O segment is different from connecting a file to another segment. When a user process is started, a base file of 16 blocks of 512 bytes is connected in the normal way to each I/O segment. These base files cannot be disconnected by the user and so are always in memory when the user process runs. Files which are connected into an I/O segment are written into the segment file. As explained earlier, files in segment 0 are in read mode and files in segment 1 in read and write unshared mode.

If the file is not yet connected by any user process, the directory has to be searched to see whether the file exists and the user has permission for the specified mode (predicate "permission"). Several things have to be checked when the file is found, such as the startblock within the file. The predicate "room in virtual memory" is used to check whether the specified part of the file fits in the specified part of the virtual memory. The parameter for this predicate indicates whether any room in the virtual memory is going to be released: a value 0 says that there will be no disconnect of a now connected part of the same file, a value 1 says that room will be released by disconnecting the now connected part of the file. For connects into an I/O segment, the number of connected files in that segment and the specified blocks have to be checked; for connects to another segment, the specified length must fit in the segments from the specified segment onwards. If the specified (part of) the file can be allocated in the virtual memory, the information about the file has to be inserted into the list of connected files. This is done by the routine "insert file information", which fills in a record and inserts it into the list.

If the file is already connected, its entry in the directory does not have to be searched for, because all the information about the file is already in the list of connected files. After checking the startblock, the predicate "file in virtual memory" is used to see whether the file is already connected by this user. If not, the permission for this user, the mode the file is connected in and the room in the virtual memory have to be checked. If the file is already connected by the user, the required mode and the current mode are compared and the room in the virtual memory is checked with the extra condition that room will be released by disconnecting the now connected part of the file. The routine "return blocks" is used to perform this disconnect (see next section about disconnect file).

The information about the connected part(s) is put into the list belonging to the file by the routine "insert block information". For a connect into an I/O segment, first a free subsegment entry in the segment table, i.e. one which has no pointer to file information set, is found. The relevant information (see description of the segment table in section 8.11) is put into it and the number of connected files in this segment and the used blocks are updated. For a connect to a normal segment as many segments from the specified segment onwards as needed to allocate the specified part of the file are used. The (part of) the file is divided into a number of blocks of maximum length 16 physical blocks of 512 bytes. Each block is first searched for in the list of already connected blocks. If it is found, the number of users is incremented by one. Otherwise the relevant information about the block is put into the list. One of the things is the contents of the page descriptor register, which is also put into the array "userapr" in the kernel. The status of the block is set to "out of core", because it is not yet in core. The routine "insert block information" also fills in the segment table.

Finally (part of) the file has to be read into core if it was a connect into an I/O segment. The base file is still in core and the connected part of the file is written into it. For a connect to another segment, a message is sent to the scheduler. This process then knows that a connect has been requested by the user process and can read the connected part(s) from the disk into core and set the relevant page address register(s).

8.21) <u>Disconnect file</u>.

A disconnect can only be done for a file which is actually connected. The routine "return blocks" takes different actions again for an I/O segment and another segment. For a file connected into I/O segment 1, it cannot be checked whether anything has actually been written into the file, because there can be several files and I/O buffers in the segment, so the written into bit of the page descriptor register can only be used to see whether the segment as a whole has been written into. Besides, the individual files connected into an I/O segment are not written back to the disk when the process is swapped out. So the file has to written back to the disk now, although in some cases this might be superfluous. For both I/O segments the blocks used and the subsegment entry containing information about the file are released. For a file connected to another segment, we go along all segments used to connect part of the file. Each segment is released for a new connect. The contents of the page descriptor register field of the block information is "ORed" with the copy of the page descriptor register in the kernel, which is cleared afterwards. If this user was the last user in core having this block connected, the core is released and a message is sent to the scheduler to indicate that some core has been released and possibly another file can be read into core. If the written into bit of the page descriptor register field is set by any disconnect (that is why the "OR" is executed) or by the scheduler, the block is written to the disk first. Finally it is checked whether this user process was the last having the block connected; if so, the block is removed from the list of connected blocks of the file concerned. The routine "return file information" removes the information about the file from the list of connected files, at least if this user process was the last having connected (part of) the file.

9) Console handler.

The code of the console handler is shared by a number of processes, each handling both input and output on one console. Each of these processes has his own data area.

9.1) Consoles.

The number of consoles and the device address,the vector address, the maximum line width and the delete character of each of them are set at the initialization of the system.

9.2) Mode.

Two modes are possible for a console:

-ascii mode: while typing a line in, characters can be deleted by typing the delete character, the whole line can be cancelled by typing the cancel character, tabs are converted into the required number of spaces and the parity bit is cleared. An input line is ended by a carriage return, a line feed, an etx character or an eot character at the beginning of a line, in which case 0 is given as length of input to the user to indicate end of input. Typing the escape character causes a message to be sent to the reset process state service. During output a line feed is converted into a carriage return followed by a line feed. If the maximum line width is exceeded, a carriage return followed by a line feed is printed and before the rest of the line the character '>' is printed to indicate an overflow.

-raw mode: no line reconstruction is done during input if the console is in raw mode, all characters are transferred unchanged to the user; the way to get an input block of length 0 is to use the break key. Output is printed exactly as it is sent to the handler: no characters are added anywhere.

It must by specified whether or not typed in characters should be echoed by the handler. Invisible characters, which have no special meaning, are echoed as '^' followed by the character with ascii code 64 plus the ascii code of the invisible character. If the console is in ascii mode with echo and the delete character is backspace, this character is echoed when typed in; if the console is in this mode but the delete character is rubout, a backslash and the deleted character are echoed when the rubout character is typed for the first time, after that all subsequently deleted characters are echoed until a character differing from the rubout character is typed, the echo of this character is preceded by a backslash to indicate the end of the sequence of deleted characters. When the delete character or the cancel character is typed and the length of the line is zero, the bell character is output to indicate the failure of the action.

9.3) Opening a console.

Only the user who has opened a console can make use of it. Opening can be done, when the console is not already in use, in two different ways:

- -pressing any key on the keyboard. The handler prints the string 'name:' and the user should type his name. When the name is ended (by a carriage return or line feed) the string 'passwd:' is printed and the user should type his password. Echoing of the password is suppressed. The name and password are checked by the login service. If the user is admitted to the system, the assigned user service number is set as the user of the console, which is left in ascii mode with echoing. The user process which is started, can now make use of the console. If the user is not admitted to the system, an appropriate message is printed. See also chapter about the login service (chapter 11).
- -sending an open request to the handler. The required mode (ascii or raw) and echo or no echo can be specified. A process can in this way use a console e.g. to print a file.

9.4) Input/output buffers.

Buffers for input and output are located in segment 0 and segment 1 of the user virtual memory respectively. A buffer can occupy one or more blocks of 512 bytes. See the section about the virtual memory of a user process (section 8.9).

9.5) Input.

Characters typed in are first put into a buffer of length 255 bytes in the data area of the handler. If the console is in ascii mode, line reconstruction is done; if echo is required the character is echoed to the console.

The transfer of characters to the user area is performed in blocks. If the console is in ascii mode a block is defined as a sequence of characters up to a carriage return, a line feed, an etx character or eot character at the beginning of a line, or a sequence of 255 characters, which is the buffer size. In raw mode a block can be terminated by a break, which further produces a block of length 0, indicating end of input. When a get input service call is issued by the user, available input is copied to the user area and a reply is sent only when the console is in ascii mode and one or more blocks of input are complete, or when the console is in raw mode and one or more blocks are complete or the number of available characters is equal to or exceeds the maximum number of characters required by the user. Copying is done in the following amounts:

- -if there are a number of complete blocks available, as many of these as fit entirely into the specified area are copied
- -if the first block does not fit into the specified area, as many characters of it as possible are copied
- -if the console is in raw mode or the buffer is full, the specified maximum number of characters is copied.
- -a block of length 0 is not combined with any other blocks, but given as a reply to a separate request to indicate end of input.

This strategy is chosen because console input is a very slow action and should be handled by a resident part of the system without waking up the user process for every character typed in.

When a get input request is issued by the user process and there is not input yet in the buffer, i.e. the user has not typed ahead, the prompt message is printed. This is a message consisting of up to eight characters, which can be set by the user, indicating what the process expects the user to type next. The prompt mechanism is independent of output performed by the handler and is not used to indicate that a previously requested action is terminated, but indicates a request for more input.

The command interpreter uses this facility to request the next command by setting the prompt message to 'command:'; user programs can request data by setting the prompt to e.g. 'data:'. The prompt message can be set to a null string if no prompt message is required.

If the user has typed 256 characters ahead of get input requests the input buffer is full. The user is informed of this by printing the string 'wait!' and characters now typed in are ignored. At the moment space becomes available again in the buffer by a get input request, the user is informed that he can go on by printing the message 'continue'.

9.6) Output.

Output by a console handler is very straightforward: the characters in the output buffer in segment 1 of the user virtual memory are not copied to a buffer in the data area of the handler, but are directly fetched from the user buffer and put into the output register of the console by a short interrupt routine. When the console is in ascii mode, a line feed is replaced by a carriage return plus a line feed and a line overflow is indicated.

9.7) Simultaneous input and output.

If a put output request is received by the handler and the user is typing ahead, the output is withheld until a block of input is completed.

If the user starts typing whilst output is going on, and the console is in ascii mode, the current output line is finished, after which the already typed in characters are, if necessary, echoed and the user can type a block of input. After that, the rest of the output characters, if any, are printed.

Both strategies are chosen to prevent output characters and echoed input characters from being mixed on the same line.

9.8) Escape.

If a user wants to end the execution of a program, e.g. because it is in a never ending loop, this can only be done by a process independent of the user process. The console handler offers this possibility.

If the console is in ascii mode and he types the escape character, a message is sent to the reset process state service, which puts the user process into a state from which it can recover. As error code the value -1 is used, so that the recovery service can distinguish it from the normal errors. All input which is not yet sent to the user is thrown away and the output which is going on is stopped immediately.

9.9) Further services.

Besides opening the console, getting input, putting output, setting the input request message and the escape possibility, there are the following services:
- change mode, i.e. set ascii or raw mode and echo or no echo
- get mode, i.e. get the mode the console is in at the moment
- close the console, i.e. indicate that the console is no longer needed by the user, so that it can be opened by another user.

9.10) Convenience and efficiency of the handler.

The first aim when designing the handler was to supply a tidy facility to the user: no output of different users is mixed, no echoed input characters and output characters are mixed on the same line and a real prompt mechanism is offered. These things are rarely offered on other systems.

The second aim was to make efficient handlers: input line reconstruction is done by the handler, so that the user does not have to be swapped into core for every character typed in and output is done very efficiently by a short interrupt routine without any copying from the user area to the system area.

The two aims are not really in conflict, as one might expect. The only consequence of making the handler tidy is, that it becomes slightly more complicated, but this is not done at the cost of efficiency.

9.11) Main part of the handler.

When you have understood the principles of the handler described in the previous sections, it is not so difficult to understand the code as well.

For every console handler there is a record in the array "condes", containing information about the physical properties of the console and the current state of the handler.

The code is divided into two parts:
- the part handling interrupts from the console. Interrupts from input are converted into a message to the handler; interrupts from output are handled immediately, i.e. a character is put into the output buffer, are converted into a message to the handler or are converted into a "signal" on the console semaphore. For more details, see following sections. This part is written in assembly language.
- the non-interrupt part, handling user requests in the first instance and input, written in IMP.

The handler can be considered as a finite state machine with the following states:
- "unused": the console is not in use
- "name": a user is typing in his name
- "passwd": a user is typing in his password

-"rest": no input or output is going on
-"input": a get input request is received while no complete block of input was available, so the handler is really waiting for input
-"output": a put output request is being serviced, without any interruption for input
-"input ahead": input is typed ahead, i.e. without having received a get input request for it
-"output while input pending": during output, the user starts typing ahead. If in ascii mode, the current line of output is finished, after which the user can type a block of input
-"input while output pending":the user is typing ahead a block of input, while there is still output to be printed. The output is restarted when the user finishes the input block.

The transitions between the different states are quite straightforward and depend on the requests received from the user process and the things the user sitting at the console is doing.

There are a number of important switches, i.e. vectors of labels, in the non-interrupt part of the handler:
-"userserv", used when receiving a message from a user process
-"consserv", used when receiving a message from the console
-"in", used when a message from the console concerns input
-"out", used when a message from the console concerns output.
To which of the labels of a switch control is passed, depends for the first two switches on the request made and for the last two on the current state of the console.

9.12) <u>More about output</u>.

Characters are put into the output buffer of the console in one of the following ways:
-directly by the output interrupt routine. This is done for the characters from a put output request: the start address, the map register to reach the area the characters are located in and the number of characters are put into "condes" and the interrupt enable bit of the output status register is set. Characters will be put into the output buffer register by the interrupt routine, without intervention of the rest of the handler. The output state field in "condes", called "dmasignal", is set to "dma" in this case.
-by the non-interrupt part of the handler. This is done for input characters which must be echoed, for characters from messages like 'stop input' and for the input request message characters by the routine "outputchar", which does a wait on the console semaphore. The corresponding signal on the semaphore is done by the interrupt routine. When the handler continues, the character is put into the output buffer. The output state field in "condes" is set to "signal" in this case.
These two ways are chosen for simplicity. If the output of input characters which must be echoed, characters from messages to the user and characters from the input request message were done by the output interrupt routine too, the handler would become much more complicated. E.g. a second input character could be received before the first one is echoed. To solve this kind of synchronization problem, the number of states would have to be considerably increased. The present solution is both simpler and tidier. Besides, there is no efficiency problem, because the number of characters output by the non-interrupt part is

very small.

9.13) Interrupts.

The interrupt routines for input ("consin") and for output ("consout") both make use of the new program status word loaded from the interrupt vector to identify the console: the low-order four bits contain the console number. This trick is copied from the UNIX system. The entry in "condes" is determined from the console number.

The input interrupt routine reads the typed in character from the input buffer register and a message containing the character is sent to the handler.

The output interrupt routine is slightly more complicated. If the output state is "signal", the interrupt is converted into a signal on the console semaphore. If the output state is "dma", the mapping register for segment 6 of the kernel is set so that this segment is mapped onto the output to be printed. If the console is in raw mode, the next character is simply printed, but if the console is in ascii mode, a line feed is replaced by a carriage return, later followed by a line feed, and a line overflow is indicated by printing a carriage return, a line feed and the character '>' first. Of course only one character is printed per interrupt. A message is sent to the handler when:

- -all output to be printed has been printed
- -the console is in ascii mode, the console state is "output while input pending" and the current line of output is finished; in this case the handler will enable the user to type a block of input before output is continued.

Mostly there will be a simple exit from the interrupt routine, after enabling further interrupts from the console.

9.14) Handling requests from the user.

Only the implementation of the get input and put output requests is somewhat complicated.

A request for input is handled in the following way. If one or more blocks of input are available, the routine "get ready for copy input" is called, which sends a message to the scheduler to ask for a reply at the moment segment 0 of the user virtual memory, where the input is copied to, is in core. The scheduler checks whether the blocks, where the user input buffer is located in, are used to access a file also and sends a refusal to the user if this is the case; otherwise it sends a reply to the handler at the appropriate moment. When this reply is received, the mapping register for segment 6 of the handler is set so that this segment is mapped onto the core where segment 0 of the user virtual memory is located. Then input is copied, in the amounts previously described, via segment 6. If there is no input available and the user has not typed ahead, the input request message is printed.

A put output request is directed, by the service exchange, via the scheduler, which checks whether the blocks, where the user output buffer is located, are already in use, and sends a refusal to the user if this is the case. Otherwise the scheduler takes care that the blocks are not released before the characters are actually printed and sends the request to the handler. Here the output by means of the output interrupt routine is started, at least if the user is not typing ahead, in which case the state becomes "input while output pending".

9.15) Handling requests from the console.

This section will probably only be understood in conjunction with the source of the console handler.

Input characters received are always handled by the routine "process character", which does the line reconstruction, echoing and puts the character into the input buffer.

If the console is unused at the moment an input character is received, the handler successively asks the user to type in his name and password and sends these to the login service to see whether or not the user is admitted to the system.

Some state transitions for received input characters are interesting. If the state is "input" and a block of input is complete, the scheduler is asked to get segment 0 into core and the state becomes "rest"; if the state is output it becomes "output while input pending"; if the state is "input ahead" and the end of a block is reached, it becomes "rest" again; if the state is "input while output pending" and the end of a block is reached, output is restarted. Characters typed in while the state is "output while input pending" are saved in the array "saveinput".

The output interrupt routine sends a message to the non-interrupt part of the handler when the state is "output" or "output while input pending" at the moment all characters are printed. The blocks of core used can now be released; this is done by sending a message to the scheduler. The state becomes "rest" or "input ahead" now.

If the state is "output while input pending", the output interrupt routine sends a message too when the mode is ascii and the current line is finished. The characters stored in "saveinput" are processed now. When the block of input is already complete, output is restarted, otherwise the state becomes "input while output pending".

10) Recovery.

The recovery services are meant to get a user process into a state from which it can recover after an error has occurred. Possible errors are:

-trying to execute a non-executable instruction (e.g. an attempt to move a word to an odd address or to address a part of the virtual memory for which no mapping is set up)
-a request for a non existing service
-an attempt to execute an emulator trap instruction.

Whatever a user process does when an error has occurred, is left to the process itself. It can for example print the error number and the values of registers like the program counter at the moment the error occurred. The only thing the reset process state service, called by the kernel when an error occurs, does, is to reset the values of the registers r5,sp and pc of the process to values which must be specified by the process itself by a "supervisor call" to the set recovery registers service, and to disconnect all connected files.

10.1) Setting the recovery registers.

The values of the registers r5,sp and pc to be set when an error occurs, are stored in the array "recov info".

10.2) Resetting the state of a process.

When an error occurs, a "supervisor call" to the reset process state service is generated in the kernel, containing the number of the error. See section about error handling (section 4.11).

The reset process state service is called from a console handler if the user of it types an escape character.

The reset process state service sends a message to the user process, which is waiting for it because of the "supervisor call" generated in the kernel. This message contains the error number and the values of the registers r5, sp and pc at the moment the error occurred, which are got from the process table entry of the process. R5 is reset by setting the last parameter of the message to the user process, which is copied to r5 by the kernel, to the specified value. The values of the registers sp and pc in the process table entry of the process are reset to their specified values directly.

All files connected at the I/O segments and segments 2-5 are disconnected. This is not done however for an error with number -1, because this error number is used by a console handler if an escape character has been typed. In this case the files connected by the user process cannot be disconnected, because they might be out of core if the process is not running, which is quite possible because the console handler is independent of the user process. The user process itself must do the disconnects when it starts running again.

The recovery services are part of the file system process, because use is made of a routine to disconnect the files.

11) Login and logout.

When a user wishes to enter the system, he presses any key on the keyboard of the console he wants to make use of. The console handler asks the user to type his name and password, which are stored in a buffer. The addresses of the typed in name and password are sent to the login service, which checks them to see whether the user should be admitted to the system or not.

The names of all users, who have permission to use the system, are in the password file. In this file, which can only be connected by the system manager, there is also an encoded form of the password of every user. The encoding is done by manipulating the values of the characters of the password and throwing away certain bits. The algorithm used for the encoding when a new user gets permission to use the system and his name and password are entered into the password file, is of course the same as the algorithm used to check whether someone who wants to use the system has permission or not. The predicate "identification ok" is used for the last purpose.

If the user is known and the password is correct, the array "id", containing for every possible user process number the identification of the user of the process with that number or a zero indicating that the process number is not in use, is checked. If the user has already logged in at another terminal or no more user process numbers are free to be used, the user is not admitted. The user process with the lowest number is reserved for use by the system manager, so that this process can be given permission to do things the other user processes cannot do. This facility is for example used by the create file service, to offer the possibility to the system manager of creating a file for another user. The entry in "id" with the chosen process number gets the value of the user identification.

The base files seg0.s, seg1.s and seg6.s belonging to the user and seg7.o[sy], containing the standard subsystem, are now connected for the user at the segments 0,1,6 and 7 in read unshared, read-write unshared, read-write unshared and read shared mode respectively. If an error occurs during one of the connects, the already connected base files are disconnected again, the assigned user process number is freed again and the user is not admitted.

If the user is admitted to the system, a message containing the console service number and the user identification is sent to the assigned user process, which is waiting for a message from the login service and is woken up in this way. A message is sent to the scheduler to get the process running. A reply is sent to the console too, indicating that the user is admitted or giving the reason that he is not admitted. In the latter case, the console handler prints an appropriate message.

The logout service is very simple. All files connected to the I/O segments and to segments 2-5 are disconnected, as well as the base files, connected to segments 0,1 and 6, and the subsystem, connected to segment 7. The entry in "id" belonging to the number of the user process is set to zero, so that the process number can be assigned to another user by the login service. The "asleep" field in the process table entry of the user process is set to "login", so that the process can only be woken up again by the login service, and a message is sent to the scheduler.

The login and logout services, like the recovery services, are for convenience located in the file system process. They make use of routines of the file system to connect and disconnect files. "Id" is used by the file services to get the identification of a user.

12) Scheduler.

The scheduler is meant to share the available processor time in a fair way amongst the user processes which are ready to run. A round-robin scheme has been chosen: each user process, which is ready to run and in core, is given in turn a fixed amount of processor time called a time slice; if processing is not completed at the end of the time slice, the process is interrupted and returned to the end of a queue to wait for another time slice.

The round-robin scheme in itself is very simple. Problems arise from the limited amount of available core. As explained in the chapter about the file system (chapter 8), all parts of files connected by a user process must be loaded into core before the process can start running. To give all user processes, which are ready to run, a fair part of the processor time, swapping, i.e. the exchange of files between core and disk, had to be introduced. After a user process has used a certain number of time slices, the files which are connected by the process and which have been changed, are rewritten to the disk if another user process, which is ready to run, is not yet loaded. The latter can be loaded now and given a number of time slices too.

Some of the decisions made were quite arbitrary and should be evaluated by measuring the performance of the system. As examples consider the number of time slices given to a process when it is ready to run and the way the next process to be loaded is selected.

12.1) Queues.

There are a number of queues used by the scheduler. Each user process is in one and only one of them. The queues are the following:

- -the run queue, containing the processes which are ready to run and entirely loaded
- -the candidate queue, containing the processes which are ready to run, but not yet in the run queue, because they are not entirely loaded or the maximum number of processes in the run queue has already been reached
- -the file system queue, containing processes having issued a "supervisor call" to the connect, disconnect, recover or logout service, which is not yet entirely handled
- -the high priority queue, containing processes which are ready to run but possibly not entirely loaded and which have a high priority to be moved to the run queue
- -the low priority queue, containing processes which are ready to run but possibly not entirely loaded and which have a lower priority to be moved to the run queue
- -the blocked queue, containing the processes which are waiting for a reply to a "supervisor call" to a service other than the connect, disconnect, recover and logout service
- -the login queue, containing the processes at which no user has logged in at the moment

12.2) Round-robin.

As said previously, the round-robin scheme is very simple. Each time the running user process (variable "runproc" in the kernel is used by the despatcher to start running a user process) has to be selected, the first process out of the run queue is taken and a message from the clock is requested at the end of the time slice given to the process. If the run queue is empty, the idle process is chosen. The selection is done by the routine "select running process". See also the section about the despatcher (section 4.12).

At the end of the time slice, assuming that the process has not become heldup previously because of a "supervisor call", there are two possibilities:

- the number of time slices given to the process has not been used completely, so that the process has the right to have another time slice. The process is put to the end of the run queue.
- no more time slices are left for the process. In this case the process is put into the low priority queue with a large allocation of time slices to be used the next time the process is moved into the run queue.

The main advantage of round-robin scheduling is the guarantee that short requests will be handled within a reasonable time. Long requests are prevented from monopolizing the system by interrupting them at the end of a time slice and removing them from the run queue to the low priority queue, from which they may be selected to have files removed from core to disk, after a number of time slices. The main disadvantage of round-robin scheduling is the overhead caused by the swapping.

12.3) The candidate.

Before a process is moved into the running queue, it has to be loaded. The process at the head of the candidate queue, called the candidate, is the first process which will be loaded and moved into the running queue.

Processes enter the candidate queue in one of the following ways. If a process is in the file system queue and the "supervisor call" to the connect, disconnect or recover service is entirely handled, the process is moved to the beginning of the candidate queue. If a candidate is needed and there is none, a candidate is selected from the high priority queue or low priority queue by the routine "select candidate", which selects a process from the high priority queue more often than from the low priority queue.

12.4) Loading the candidate.

The routine "load candidate" is used to bring all parts of files connected by the candidate into core.

The following entries in the segment table of the candidate are relevant for each segment:

- "fileptr", if it is zero no file is connected at the segment, otherwise it contains a pointer to the information about the file (see chapter 8 about the file system)
- "blockptr", a pointer to the information about the connected (part of) the file (idem)
- "status"; if it equals "in" the file connected at the segment is in core, if it equals "out" the file connected at the segment is possibly not in core.

The first thing the routine does, is to check for all segments to which a file is connected and whose status equals "out", whether the (part of) the file connected to the segment is in core. This is possible for shared files. In this case the status of the segment is set to "in", the copy of the page address register in the kernel is set and the number of users in core of the block is updated.

For a segment to which (part of) a file is connected which is out core, the core manager is called to get an area of core of the specified length. If this call is successful, the disk driver is called to bring the file from the disk into core, the file information is reset and the copy of the page address register in the kernel is set. The disk driver is called by a "send", so that the scheduler can continue.

If the call to the core manager was unsuccessful and there are no parts being written from core to the disk, in which case core will soon become free, an attempt is made to get core from the victim, which is the process whose core will be released first. If there is no victim at the moment, an attempt is made to find one by the routine "choose". See next section. If no victim can be found, the loading of the candidate is suspended. If the victim is the same process as the candidate, all core of that process is released by the routine "remove core". This possibility has to be taken into account to prevent a deadlock. If core in use for (part of) a file, which need not be written back to the disk, has been released, the loading continues. If core will only be released after (part of) a file is written back to the disk, the loading is suspended.

The replies to the requests to the disk handler are received in the main cycle of the process, which will be discussed later, as will the routine "remove core".

12.5) The choice of a victim.

The victim is chosen in the following way. First the blocked queue is examined from the beginning. If no process is found with any segments with state "in", the low priority queue is searched from the end. If no process is found in this queue, the high priority queue is searched. If in this queue a victim could not be found either, the candidate queue is searched from the end if there is more than one process in the candidate queue or both the number of processes in the run queue and the file system queue are zero. In the last case, the victim will be the same process as the candidate (see previous section).

Core from processes in the file system queue cannot be released, because files connected by such a process are expected to stay in core to be possibly used by the connect, disconnect, recover or logout service.

12.6) Releasing core in use by the victim.

The routine "remove core" is used to release core in use by the victim.

For the non I/O segments of this process whose status is "in", the copy of the page descriptor register in the block descriptor is ORed with the copy of it in the kernel, whose written into bit is cleared for future use. If the written into bit in the copy in the block descriptor is zero, the status of the segment is set to "out". If the number of users which are in core and are using the block becomes zero, the core used by the block can be released by issueing a "supervisor call" to the core manager and the routine is left.

For the I/O segments whose status is "in", the status is normally set to "out", this is not done however for segment 0 when it is in use by a handler to copy input from its own buffer to the user area, which is indicated by the field "handler" (see chapter 9 about the console handler). Because the files at the I/O segments are connected in unshared mode, the core used for such a file can be released immediately when a handler has not written into it and no (part of) a file is connected into the segment ("copy ok" is set to "false" when one of these things is done) and when the user has also not written into it (the written into bit is checked). When the file has been written into, it has to be written back to the disk before the core can be released. In this case the disk driver is called, the written into bit in the copy of the page descriptor register is cleared and "copy ok" is set to "true". In both cases the routine is left.

For the non I/O segments whose status is still "in", which means that they have been written into, the status is now set to "out". If the number of users, who are in core and are using the (part of the) file connected to the segment, becomes zero, the disk driver is called to write the (part of the) file back to the disk, the written into bit in the copy of the page descriptor register in the file descriptor is cleared and the routine is left.

If the end of the routine is reached, which means that no core from the victim could be released immediately or will be released as soon as a confirmation from the disk driver is received that (part of) a file is written back to the disk, this is indicated by setting "result" to zero. The victim can no longer serve as such; this is indicated by setting "victim" to zero.

12.7) The main cycle of the process.

The scheduler can be called from a number of processes:
- -the clock handler
- -the disk driver
- -the file system
- -the console handlers.

It can be called from the kernel too. A put output request from a user process to a console handler is directed, by the service exchange, via the scheduler. Other requests from user processes are ignored.

When the scheduler is called from the clock handler, this is the reply to a request for a message after a time slice. If there is a process, different from the idle process, running, the time used by the process is adjusted by the routine "adjust time and queue". If the process has gone to sleep on the connect, disconnect, recover or logout service in the meantime, it is moved into the file system queue; if it has gone to sleep on another service, it is moved into the blocked queue; if the current time slice is considered to be over, the process is left in the run queue, at least if there is still a number of time slices to be used; if no time slice is left the process is put into the low priority queue. The fact that a message from the clock is received does not imply that the current time slice is over; it might be the reply to a request issued when another time slice started, which was not entirely needed by the process that got it, because it went to sleep before the end of the slice. The current time slice is considered to be over if less than a certain minimum amount of it is left.

After the adjustment of the time, and possibly the queue, of the process the following is done. The processes in the file system queue are checked to see whether they are still waiting for a reply; if not, they are moved into the run queue. Processes in the blocked queue which are no longer waiting, are moved into the high priority queue. Then an attempt is made to load the candidate, if one can be found, and a running process is selected.

A message from the disk driver might be a confirmation of a (part of a) file being read from the disk to core or of a (part of a) file being written from core to the disk. In the first case the process, for which this transfer was done, is being loaded. If the segment read was segment 0 and a handler wants to copy input from its buffer to this segment, a message is sent to this handler and the "copy ok" field in the segment table is set to "false" to indicate that the (part of the) file has to be written back to the disk before the core used for it is released. If all segments of the candidate are in core now and there is no process in the run queue, the process is put into the run queue and selected to run. In the second case core from the victim can be released now. For segment 1 only those blocks which are not in use by a handler to output characters from, can be released, for other segments all core can be released. If there is still a candidate, the loading of it is continued.

When the scheduler is called from the connect service, the process which requested the service, is moved into the candidate queue after the time used by the running process is adjusted. The candidate is loaded and a running process selected.

At a call from the disconnect service, the time used by the running process is adjusted. The processes in the file system queue which are no longer waiting for a reply are moved into the run queue. An attempt is made to load the candidate, if there is one or one can be selected, and a running process is selected.

A call from a console handler can be a call to get segment 0 of a user process into core, in which case the blocks, to which input should be copied, are checked to see whether they are not used to access a file also; if so an error message is sent to the user. A reply to the handler is sent immediately if the segment is already in core, otherwise it is remembered that the segment is needed by the handler. It can also be a call to release some blocks of output segment 1, which were used to output characters from. If the status of the area of core used to connect the base file to segment 1 is "out core", the blocks released can be given back to the free list by a request to the core manager and the candidate, if any, can be loaded.

At a call from the logout service, the process, which was stopped, is moved into the login queue and the variables belonging to the process are initialized.

The scheduler can be called by the despatcher in the kernel too. See the section about the despatcher (section 4.12). If the running process is not the idle process, it may have gone to sleep on the connect, disconnect, recover or logout service, in which case it is moved into the file system queue. If it has gone to sleep on another service, it is moved into the blocked queue. If it is no longer asleep, the process can continue. The processes in the file system queue and blocked queue are checked, and an attempt is made to load the candidate, if there is one or one can be selected, and a running process is selected.

A put output request from a user process to a console handler is sent to the scheduler first. If the console is not in use by this process or the blocks, where the output buffer is located, are already in use, a refusal is sent to the user. Otherwise care is taken that the blocks are not released until all characters in them are actually printed, and the request is passed on to the console handler.

12.8) Implementation of the queues.

As noted previously, all user processes are in one and only one of the queues. These queues are implemented in the following way.

There is a record containing timing information for every user process (the array "userlist"). These records contain a field to indicate in which queue the process is at the moment and pointers to the previous and next process in that queue too. The queues are cyclic. When there is only one process in a queue, the previous and next pointer both point to the process itself. The routine "change" is used to move a process from one queue to another.

13) Initialization.

At a start or a restart of the system, the data areas have to be initialized. There is a fixed (re) start address (location 40 octal). Part of the initialization is done by code in the kernel and part of it by a seperate initialization program.

13.1) The kernel.

The data area for the kernel is cleared first. The array "find" is filled with pointers to process descriptors. The free list of records to be used for several purposes (head cell "asl") and the list of free message buffers (head cell "free") are set up.

13.2) The clock.

To run the system, either a KW11-P or KW11-L clock is required. The KW11-P clock is prefered, because it can generate an interrupt every millisecond. This makes the clock handler more accurate than what is possible with an KW11-L clock, which can only generate an interrupt every 20 milliseconds.

First an attempt is made to start the KW11-P clock, after having set the trap vector of the location to which a trap will occur if this clock is not there. If a trap occurs, the KW11-L clock is started.

13.3) Idle process.

The process table entry of the idle process, which has the highest process number, is filled in. The idle process is set to become the first running user process. Now the separate initialization program is called.

13.4) Interrupt and trap vectors.

All interrupt and trap vectors are filled in. A vector consists of the address of the interrupt or trap routine and the new processor status word. The processor status words set for error traps contain an error code.

13.5) Processes.

For all supervisor processes and user processes an entry in the process table is filled in by the routine "insert". The stack pointer, the program counter and the processor status word are set. Supervisor processes are entered into the ready queue. User processes all have the same start address, which is in the subsystem, and are put to sleep until the login service wakes them up.

13.6) The service exchange table.

The service exchange table is filled in so that the legal service numbers are converted into the number of the process handling that service.

13.7) Consoles.

The number of consoles is specified by the constant integer "highest console". For each console a record of the array "console" containing the device address, the vector address, the maximum line width and the delete character is filled in.

The interrupt vectors for both input and output are set. The new processor status words contains an identification number to be used by the input interrupt routine ("consin") and the output interrupt routine ("consout") to identify the console.

For each console a record of the array "condes", which contains a description of the console and the state of the handler of the console, is initialized. The array "condes" is used by both the interrupt routines and the handlers.

Each handler has his own data area. The start address of this area for a certain console is determined from the size of the system, which is located in the lower part of core, and the number of the console. The initialized data area is copied from the area behind the shared code of the handlers to this area. The rest of the data area is used for uninitialized data.

For each handler an entry in the process table and two entries in the service exchange table are filled in. Specified are the program counter, the stack pointer, which points to the end of the data area and r4, which points to the beginning of the data area. Interrupts from the input of the console are enabled.

13.8) Available core and memory management registers.

The start and size of the area of core to be used for allocating files, which is located in the (higher) part of core left after allocating the system code and data areas, are determined and set into the variables "start user memory" and "available user memory". These variables are used by the core manager to set up its list of free core.

The kernel active page registers, the user active page registers and the copies in the kernel of the supervisor active page registers are initialized to map virtual addresses in segments 0-6 onto the same physical addresses and virtual addresses in segment 7 onto the register addresses. The memory management unit is started.

13.9) Exit.

Process 0, which is the clock handler, is selected to start running and is removed from the ready queue. The routine "exit" in the kernel is called to set the registers for this process.

14) User processes.

A user process has at its disposal a virtual memory of 64 K bytes, divided into 8 segments of 8 K bytes each. It can make use of the resources of the system by issueing requests to the supervisor.

14.1) Standard division of the virtual memory.

For simplicity and convenience, a standard subsystem is part of the code of every user process. This subsystem consists of the command interpreter and the perm, which contains some basic I/O routines. It is connected to segment 7. Part of segment 6 is used as data area for the perm and the rest of the segment as data area for programs. Segment 0 and 1 are used as I/O segments in which buffers for devices are located and to which files are mapped. When the user process starts, base files are connected to segments 0,1 and 6.

14.2) Influence of the IMP compiler on the system.

The way the subsystem is built and the organisation of the I/O segments is very much influenced by the requirements of the IMP compiler and run-time organization. The possibility of running an IMP program, which is one of the aims of a user process, requires that you have a perm, which is a set of standard routines to e.g. read and write a character. The perm in the standard subsystem is suited to this purpose.

Input and output in IMP is done via so called "streams". There are 3 input streams and 3 output streams, numbered 0-2. At any time there is one selected input stream and one selected output stream. All I/O takes place on the selected streams. The stream numbers are linked to devices or files in the following way. Input stream 0 and output stream 0 are always linked to the user's console, i.e. the console the process was started from. To input stream 1 and 2 can be linked devices or files by putting their names behind the name of the program to be executed. Names of devices or files to be linked to output stream 1 and 2 should be specified after a slash in the command string. If a field is left blank, the stream is mapped to "null"- which causes and "end of input" to be read on input and all output to be thrown away. As an example consider the following command string:

```
      test file1,file2/file3
```

It causes the program "test" to be executed. When the program selects input stream 1, it reads from "file1"; when it selects input stream 2, it reads from "file2". Output for stream 1 is written to "file3", while output for stream 2 is thrown away, because no device or file is specified for this stream.

Programs often use the possibility of switching the currently selected streams, so it must be possible to have all files, linked to a stream, connected at the same time. Because the segments 2-5 are used for the code and data area of the program to run, we decided to take segment 0 as an input segment and segment 1 as an output segment. These I/O segments can be used to connect several files and to hold buffers for devices. See section on the virtual memory of a user process (section 8.9).

The I/O segments are divided in the following way. The first 2 blocks of 512 bytes are reserved for stream 0, which is linked to the user's console. The rest of the segment is divided into two parts of 7 blocks, which are used for streams 1 and 2.

14.3) Implementation language.

Although parts of the subsystem could certainly be written in IMP, it is entirely written in assembly language. There are several reasons for this:

-some parts have to be written in assembly language, because one has to address the registers
-routines to read and write a character should be efficient, because they are executed very often. Especially the entry and exit from a routine can be made much shorter when the perm is written in assembly language.
-writing the subsystem partly in assembly language and partly in IMP would introduce some nasty linkage problems.

14.4) Perm routines.

The following routines are provided by the perm:

-%routine read symbol (%integername n); reads next symbol on the current input stream and advances the streampointer so that the symbol is not read again
-%integerfn next symbol; reads next symbol on the current input stream, but does not advance pointer so that the symbol can be read again
-%routine print symbol (%integer n); prints symbol on current output stream
-%routine select input (%integer stream number); further calls of read symbol and next symbol are to operate on the input stream with the specified number
-%routine select output (%integer stream number); further calls are to operate on the output stream with the specified number
-%routine close input; close the current input stream and selects input stream 0
-%routine close output; closes the current output stream and selects output stream 0
-stop; closes all streams and exit to the command interpreter
-signal; this is a language dependent feature.

14.5) Implementation of the perm.

The locations in segment 6 with the addresses 157100-160000 are used as a data area for the perm. Most important in this area are the stream descriptors, containing for every stream:

-pointers to positions in the buffer belonging to the stream
-the name of the device or file linked to the stream
-if a device is linked to it, the service number of the device
-if a file is linked to it, the block number of the first currently connected block of the file.

When the perm is called, there are a number of routines used to check the part of the command string containing the names of devices and files, which have to be linked to the streams, and to set up the stream descriptors.

If there is no error in the command string, the recovery address is set to "stop", so that after an error trap all streams are closed in the normal way, and the input request message, which is used by the console handler to ask for input, is cleared by issueing a supervisor call to the console handler. Register r4 is set to point to the gla, i.e. the initialised data area, register r5, which is used as a pointer to the IMP data area, gets the value of the stack pointer and the program is started by jumping to a fixed start address.

14.6) An example: read symbol.

As an example of how the perm makes use of the supervisor, the routine read symbol will be discussed.

At the entry of the routine ("readsym") it it checked in the descriptor of the current input stream whether all characters in the buffer belonging to the stream have been read. If so, the buffer has to be filled again.

For a device this is done by the routine "getdbuf", which sets the parameters for a get input request and issues a supervisor call to the device linked to the stream. If the call was successful, the stream descriptor is reset and "readsym" returns the first symbol out of the buffer. If the end of the information is reached, the symbol value "100004" is returned. If the device was not opened by the process, a supervisor call is issued for this purpose. If this call is not successful, i.e. the device cannot be opened, the program is stopped; otherwise the get input request is re-issued.

For a stream linked to a file the routine "getfbuf" is used. This routine issues a supervisor call to connect the next part of the file at the buffer area. First a connect in shared mode is tried; if this is not successful a connect in unshared mode is tried. If the call was successful, the stream descriptor is reset and "readsym" returns the first symbol out of the buffer. If the end of the file has been passed, the symbol value "100004" is returned. If the connect cannot be executed successfully for another reason, the program is stopped.

14.7) System routines.

There are a number of system routines, i.e. non-standard routines usable by IMP programs, in the subsystem:

- %systemroutine svc (%record (%integer service, par1, par2, par3, par4, par5) %name mes); issues a supervisor call with the specified parameters
- %systemintegerfn owner; gives identification of the user
- %systemroutine command (%string (80) s); puts the string into the command buffer from which the command interpreter tries to read characters before it starts reading from the user console; this offers the possibility of issueing commands from programs
- %systemstring (12) %fn strname (%integer inout, stream); gives the name of the device or file linked to the specified stream
- %systemroutine setstrm (%integer inout, stream, %string (12) name); links the specified device or file to the specified stream
- %systemroutine setinr (%string (11) s); sets the input request message to s
- %systemroutine specout and %systemroutine normout; used to set

whether or not output of a buffer for a device should be done at a line feed; this offers the possibility of speeding up output, because the process is put asleep only when the buffer for the device is full or the "prompt character" (k'100000') is sent.

To use one of them, one has to include a "%systemroutinespec" for the routine in the program.

14.8) Stopping a program.

When a program stops because it is at the end of its execution or an error occurred, all streams are closed. I.e. files are disconnected and devices, except the user's console, are closed. Streams 0 are selected for further input and output. If an error occurred, an appropriate message or flag is printed on the console. Control is passed back to the command interpreter.

14.9) Command language interpreter.

The command language interpreter's function is to get a command from the user and check it. A command consists of a program name, which should either be the name of an object file to be executed or the name of one of the programs located in the subsystem. In the first case the program name should be followed by the names of the devices and files to be linked to input stream 1 and 2, a slash and the names of the devices and files to be linked to output stream 1 and 2. The program is loaded and the perm is called to execute it. In the second case other parameters are expected. The program does not have to be loaded now, but can be called directly (see section 14.13 about programs in the subsystem).

The reason the perm was discussed prior to the command interpreter is that the last makes use of the perm to get characters from the user's console. If there are no characters in the command buffer (see section 14.7 about system routines), a read symbol on input stream 0 is executed. Output from the command interpreter is printed via the print symbol routine on output stream 0.

14.10) Device and file names.

Device names consists of a dot followed by two letters. The name ".tt" is used to indicate the user's console. Other names in use are:

-".la", to indicate the system console
-".di", to indicate the diablo printer

For the conventions for file names, see section 8.3 .

14.11) Searching for a program.

When searching for a program, the list of programs in the subsystem is checked first. If the name is found, the program is executed. If it is not found, it is checked whether an extension has been set; if not, the extension 'o' is set to indicate an object file. If no owner identification is set, the user's identification is set and a connect of the first block of the file at segment 2 is requested. If the connect is not successful, the user's identification is replaced by the identification of the system manager, i.e. the owner of all files which are intended for general use, and a connect is tried on this file. If an owner identification is set by the user himself, only the fully specified file is tried.

If the program file is found the loader is called to load the program; otherwise the message 'program not found' is printed. The program is started via the perm.

14.12) Loader.

At the moment the loader is called, the first block of the program file is connected at segment 2. The first 3 words of an object file contain the following information:

- -the length of the code
- -the length of the glap, i.e. the initialized data and linkage area
- -the length of the stack area needed by the program

The loader reads these header words from the file.

The next stage is to copy the glap to the gla, i.e.the area in the virtual memory where the initialized data and linkage area will be located during the execution of the program. This copying is necessary, because the program file itself must stay intact, to enable a next execution of it. The glap is located at the end of an object file. The startblock within the file is determined from the header words and the file is connected from this block onwards to segment 2 and possibly higher segments. The gla is located in the area below the stack area, which is the uninitialized data area. The gla bottom, which is the lowest address of the gla, is determined from the header words. If the gla bottom, which is always on the boundary of a segment, is the lowest address of segment 6, copying can be done immediately, because to this segment a base file is always connected. If however the gla bottom is located in one of the lower segments, a scratch file is connected from this segment onwards up to segment 6. The scratch file is used to copy the glap into and as stack area.

After copying the glap to the gla, the code part of the file is connected from segment 2 onwards.

14.13) Programs in the subsystem.

There are a number of programs which are located in the subsystem for two reasons:

- -efficiency. The program does not have to be loaded, so execution can be started immediately. This is an advantage for often used programs like "delete file".
- -no files have to be connected to load the program, so the virtual memory is left in the same state. This is necessary for a program which wants to read a certain area of the virtual memory as it is at that moment. As an example consider a dump program.

These programs must of course be short. The expected parameters are determined by each program separately. As an example, the dump program expects the begin and end address of the area to be dumped.

Stopping the user process ("logout") is also done by a program located in the subsystem.

14.14) Starting the subsystem.

When a user process is started, the system starts the initialization part of the subsystem. The area in segment 6 from 156000-157100 is used as data area by the command interpreter. This area and the stream descriptors for input and output stream 0 are initialized and the streams mentioned are selected as the current input and output streams.

The command interpreter then starts to execute commands, until the logout command is given.

14.15) Available programs.

From the available programs, we mention the following:

- -an IMP compiler
- -a linker
- -some programs to manipulate libraries, i.e. files containing information about routines which can be linked to programs
- -a recode program, giving the code the IMP compiler produces for a program
- -an editor
- -a document layout program, used to prepare this document.

15) Further developments.

The current system certainly is not finished: many extensions and improvements might be implemented. I shall mention some of these.

15.1) Papertape handlers.

A papertape reader and punch handler will be quite easy to implement once you understand the console handler code. The console handler will be much more complicated, because it has to handle both input and output. The papertape reader and punch are independent.

15.2) The use of more than one disk.

The current version uses only one disk. This might be extended to more. It should be considered carefully how the allocation of files on these disks is done. One could e.g. allocate all files of a certain user on (part of) one disk or allocate all files created during a certain run period of the system on one disk and copy them afterwards to another disk.

15.3) Archive system.

It should be possible to archive a file on a disk or tape different from the disk(s) used to run the system. This is required both as a precaution against loss and as a way to save space on the disk(s) used to run the system.

15.4) Performance evaluation.

The performance of the system should be measured. Especially in the design of the scheduler there were made some arbitrary decisions, e.g.the length of a time slice. The consequences of these decisions were not understood and should be investigated.

This evaluation will almost certainly lead to improvements in the scheduler and in the disk driver, which can be rewritten to implement a better service discipline than the first come first served discipline.

References.

[1] PDP11 Processor Handbook, Digital Equipment Corporation

[2] PDP11 Peripherals Handbook, Digital Equipment Corporation

[3] RT11 Software Support Manual, Digital Equipment Corporation
RT11 System Reference Manual, Digital Equipment Corporation

[4] Ritchie,D.M. and Thompson,K., The UNIX Time-Sharing System,
Communications of the ACM, vol.17, no.7 (july 1974), pp.365-375

[5] Whitfield,H. and Wight,A.S., EMAS - The Edinburgh Multi-Access System,
The Computer Journal, vol.16, no.4 (november 1973), pp.331-346

[6] Robertson,P.S., The IMP Language: A Reference Manual,
University of Edinburgh

[7] Hansen,P.B., Operating System Principles,
Prentice-Hall, 1973

[8] Coffman,E.G. and Denning,P.J., Operating System Theory,
Prentice-Hall, 1973