Supermax

System Operation Guide

Dansk Data Elektronik A/S
1 May 1984

1. Introduction

This manual describes the features of the Supermax Operating System. It does not explain how these features are invoked in the various programming languages; this the reader will find in the relevant language manuals. The purpose of this manual is to provide the reader with knowledge of what can be done in the Supermax Operating System, and thus give him or her the necessary background for reading the language manuals.

It should be noted that some programming languages may not give the user access to all the features of the Supermax Operating System described in this manual.

The reader is expected to be familiar with the Supermax Operating System Introductory Guide.

Numbers preceded by '0x' are hexadecimal. All other numbers are decimal.

A few paragraphs are marked with astisks (*) in the margin. These paragraphs describe features that have not yet been implemented, but will be in the near future.

Dansk Data Elektronik A/S reserves the right to change the specifications in this manual without warning. Dansk Data Elektronik A/S is not responsible for the effects of typographical errors or other inaccuracies in this manual and cannot be held liable for the effects of the implementation and use of the structures described herein.

2. Overview.

The Supermax Operating System performs several tasks. They may be divided into three groups:

1) Process Management.
2) Memory Management.
3) I/O management.

The Process Management part of the Supermax Operating System takes care of the execution of programs. An executing program is called a 'process' (other operating systems use the term 'task'). The Process Management loads a program and starts its execution as a process, controls the process while it is running, and removes the process when its job is done.

The Memory Management part of the Supermax Operating System takes care of the allocation and deallocation of memory for processes.

The I/O Management part of the Supermax Operating System services input and output requests from processes.

Of course, these three parts are not independent: Starting the execution of a program, as done by the Process Management, involves both allocation of memory for the process, done by the Memory Management, and the loading of the program from a disk file, done by the I/O Management. Also, there are some features, such as reading the system clock, that can hardly be placed in any of the three categories.

3. Process Management.

A process is an executing program. Processes have various properties, and various operations may be performed on processes.

3.1. Process Properties.

A process has a real user ID and a real group ID. These two 16-bit numbers are almost always identical to the user ID and group ID of the user who gave the command that started the process.

A process has an effective user ID and an effective group ID. These two 16-bit numbers are in most cases identical to the real user and group IDs. When the operating system checks if a process has access right to a given iounit, the effective user and group IDs of the process are used, rather than the real user and group IDs. Further, a process, A, can perform operations on a process, B, only if the effective user ID of process A is the same as the real user ID of process B, or if process A is a privileged process (see below).

A process has a name, which is a string of 8 characters. Together with the real user ID the name uniquely identifies the process within one MCU.

A process has a process ID. This number is in the range from 0 to 30000 and uniquely identifies the process within one MCU.

A process has a priority. This is a number in the range from -20 to 20 where -20 represents the highest priority. If there are several processes that all want access to the CPU, the process with the highest priority (lowest priority number) will run. If there are several processes with the same priority that all want access to the CPU, the CPU is time-shared between them with a time slice of 80 milliseconds.

A process has a process group ID. A process belongs to a so-called 'process group' (more about this later). One process within each group is the 'process group leader'. The process group ID of all processes within a process group is identical to the process ID of the leader of the group.

A process has a tty ID. This is the number of the terminal from which

the process was started. Some processes, various system processes for example, do not belong to any particular terminal; they have a tty ID of -1.

A process may be privileged or unprivileged. Privileged processes have effective user ID 0, unprivileged processes have effect user IDs greater than zero. Privileged processes can do things that are forbidden to unprivileged processes. For example, a privileged process may access any file. We normally say that the superuser is allowed to access any file, but it would be more correct to say that processes started by a command from the superuser will normally be privileged and therefore they may access any file.

A process has a current directory. This is the directory used when searching for iounits whose pathname does not begin with a slash.

A process has a current root directory. This is the directory used when searching for iounits whose pathname begins with a slash. Normally all processes use the same root directory. Only when a different operating system environment is being tested, is the root directory changed.

A process has an iounit creation mask. This is a set of 9 bits that specify the access rights not granted to iounits created by the process. If, for example, bit 4 (corresponding to write access for group) of the iounit creation mask is one, all iounits created by the process will refuse write access for group, even if bit 4 of the access rights specified by the process is one.

A process may be in different states, that are not necessarily mutually exclusive:

1) Running. This means that the CPU is currently actually executing this process's code.
2) Active. This means that the process wants the CPU to execute its code, but another process is currently running.
3) Internally suspended. This means that the process has issued a system call, typically an i/o request, that caused its execution to be suspended until the requested operation has been completed.
4) Externally suspended. This means that another process has issued a 'suspend' system call for this process.
5) Dying. This means that the operating system has been requested to remove the process, but because the process still has some pen-

ding i/o requests whose termination its awaits, it cannot yet be removed.

6) Stopped. This means that the execution of the process is being monitored by another process, and currently the process is not running, but is open to inspection by the monitoring process.

3.2. Operations on Processes.

One of the relationships that exist between two processes is the parent/child relationship. We may use the words 'process A is the parent of process B' or 'process B is a child of process A'. This parent/child relationship has the following implications:

1) if B dies, A will be told why B died,
2) if an attention exception (see below) occurs and B does not catch it, A will get it,
3) if an attention exception occurs and B catches it, A will not get it.
4) A may monitor the execution of B.

The Supermax Operating System thus maintains a hierarchy of processes, such as: A is the parent of B and C, B is the parent of D, D is the parent of E, F, and G, and so on. Here A is called a 'main process', that is, a process with no parent process, the head of the hierarchy. Processes that are not 'main processes' are called 'sub-processes'. Several main processes may be present in the system, each with their own hierarchy of sub-processes.

If process A is the parent process of B or the 'grandparent' process of B or the 'great-grandparent' process of B, etc., A is said to be an 'ancestor' process of B and B is a 'descendant' process of A.

Two processes with the same parent process are called 'sibling' processes.

3.2.1. Process Birth.

A process is always started by another process. A process may be started in five different ways:

1) By 'Spawning'.
2) By 'Production'.
3) By 'Gemmation'.
4) By 'Metamorphosis'.
5) By 'Forking'.

When a process is started it inherits most of the properties of the parent process. The most important exceptions are the effective user and group ID, the rules for which are given below, and the process group ID in the case of production and gemmation.

Spawning.
If process A 'spawns' process B, A will become the parent process of B.

Production.
If process A 'produces' process B, B will become a main process, that is, the head of a new process hierarchy. B will be the leader of a new process group.

Gemmation.
If process A 'gemmates' process B, the parent process of A will become the parent process of B. A and B will thus become sibling processes. If A is a main process, so will B be. B will belong to the same process group as A's parent.

Metamorphosis.
If process A 'metamorphoses', its program code will be replaced by another program code and the execution of A continues with the new program code. Metamorphosis thus really does not involve the starting of a new process, but rather that the process A continues its execution in a new program, the old program having been removed. Metamorphosis is not equivalent to a gemmation followed by the death of process A, for with metamorphosis no death will be reported to the parent process of A.

Forking.

If process A 'forks', an identical copy of the program code for A is made, and execution continues from the fork request both in A and in the copy, being process B. The fork request will return two different values in process A and B, and by examining this value the processes will know the answer to the (almost theological) question: "Who am I? Am I A or am I B?" A will be the parent process of B.

The program code for the execution of processes may come from several different sources:

1) From a file.
2) From an installed program.
3) From a subroutine.
4) From a copy of another program.

The origin of the program code is the factor that controls the value of the effective and real user IDs of the process.

Program Code Taken from a File.

If process A spawns, produces, or gemmates another process, or if process A metamorphoses, the program code to be executed may be taken from a disk file. The file must contain a load module produced by a linker, and the effective user or group ID of A must give the process execute access right to the file.

The real user and group IDs of the new process will be A's real user and group IDs, respectively.

If the file containing the program has the 'set user ID on execution' bit on (see section 5.2.2), the effective user ID of the new process will be the owner ID of the file; otherwise, the effective user ID will be identical to the real user ID. If the file containing the program has the 'set group ID on execution' bit on, the effective group ID of the new process will be the group ID of the file; otherwise, the effective group ID will be identical to the real group ID.

For metamorphosing processes the effective user and group IDs will be set according to the same rules, even though no new process is actually created.

If several processes on one MCU execute the same program, they share * the program code in memory. *

Program Code Taken from an Installed Program.

If process A spawns, produces, or gemmates another process, or if process A metamorphoses, the program code to be executed may be taken from an installed program (see below).

The effective user or group ID of A must give the process execute * access right to the installed program. *

The real user and group IDs of the new process will be A's real user and group IDs.

If the installed program has the 'set user ID on execution' bit on * (see below), the effective user ID of the new process will be the * owner ID of the program; otherwise, the effective user ID will be * identical to the real user ID. If the installed program has the 'set * group ID on execution' bit on, the effective group ID of the new pro- * cess will be the group ID of the program; otherwise, the effective * group ID will be identical to the real group ID. *

For metamorphosing processes the effective user and group IDs will be set according to the same rule, even though no new process is actually created.

Program Code Taken from a Subroutine.

Process A may spawn a piece of its own program code, typically a subroutine, as a process. This means that process A has the option of calling this subroutine, in which case the execution of the main program of A stops until the subroutine has completed its job and returns, or starting it as a process, in which case the execution of the main program of A continues concurrently with the execution of the subroutine. A subroutine executing as a process, is called an in-memory process.

In-memory processes may be used, for example, for asynchronous i/o. The execution of a process stops while i/o is being handled. By having an in-memory process do the i/o, process A may continue computing while i/o is being performed.

The effective and real user and group IDs of the new process will be identical to those of the starting process.

Program Code Taken from a Copy of Another Program.

This is what happens when a process forks.

The effective and real user and group IDs of the new process will be identical to those of the starting process.

There are a few restrictions on when the various ways of starting processes may be used:

- In-memory processes can only be spawned. They cannot be produced or gemmated.
- In-memory processes cannot fork.
- In-memory processes cannot metamorphose.
- A process that has in-memory child processes cannot metamorphose.
- A process can only start processes on the same MCU as the one on which it is executing. It may, however, communicate with a process on another MCU and possibly request this other process to start a new process.

When a process forks, its child in-memory processes, if any, are not forked with it.

3.2.2. Installed Programs.

A program may be 'installed' in the memory of an MCU. This means that the program code and the initial values of data areas are permanently present in the MCU memory, regardless of whether a process is currently executing the program or not. Installed programs have both advantages and disadvantages when compared with programs loaded from files.

The advantage is that execution of installed programs can be started very fast because no time is needed for the loading of program code from a file.

The disadvantage is that the program is always present in memory, even when no process is executing it. This gives a less economical use of memory.

Only privileged processes may install and remove programs. When a program is being installed, its code is taken from a disk file to which the installing process must have execute access.

Each installed program has an owner and group ID as well as a set of protection bits with the same value and meaning as the corresponding bits in the file from which the program code originated (see section 5.2.2). *

Installing a program is done only at one MCU at a time.

3.2.3. Iounits at Process Start.

The initial execution environment of a process is largely determined by the iounits with which it communicates. When a process is started it inherits a number of open iounits from the process that started it. The following possiblities exist:

1) When a process is spawned, produced, or gemmated, the starting process specifies which of its own open iounits the new process should inherit.

2) When a process metamorphoses, its open iounits will normally remain open across the metamorphosis. It is, however, possible to specify that a given iounit should be closed on metamorphosis.

3) When a process forks, the new process inherits all the open iounits of the parent process.

3.2.4. Process Death.

A process may die for two reasons:

1) By exitting, that is, committing suicide. This is the normal way in which a process terminates its execution.

2) If an exception is raised for which the process has no exception handler. This is described in detail is section 3.2.6. An exception may, for example, be a division by zero, or an attempt to access memory that the process is not allowed to access.

When a process dies, its death is reported to its parent process, if any. The death information includes the reason for the death (normal termination, division by zero, or whatever) and possibly a completion code, which is a number specified in the exit request. How the parent process will interpret the completion code, depends on the program. A completion code is given only when a process terminates due to an exit request (a suicide).

When a process dies, all its iounits are closed, and it is detached from all memory partitons (see chapter 4).

What happens to the child processes of a dying process? Well, it depends. If the dying process is not the leader of a process group, its child processes are turned into main processes (processes directly subordinate to the operating system). The child processes remain members of the process group. If, on the other hand, the dying process is the leader of a process group, which is normally the case with processes started from log-on, the 'hang-up' exception is raised in all the processes in the process group. This will kill the processes, unless they have taken measures to ensure that this does not happen.

In-memory processes are killed when the process containing them dies.

3.2.5. Suspending a Process.

A process may suspend itself or be suspended by another process. When a suspend request is issued, a time is specified. Execution of the process stops until this time is expired. It is, however, possible to resume the execution of a process before the time is expired.

A process that suspends itself will know whether execution was resumed because of time expiration or because of a resume request from another process.

The system clock has a resolution of 40 milliseconds. Therefore this is the finest resolution of the supend time.

The suspend time may be specified as 'indefinately'.

A process, A, may suspend and resume another process, B, only if A is a privileged process or A has an effective user ID identical to the real user ID of B. A process may always suspend itself.

3.2.6. Exceptions.

An exception is an abnormal event in a process. The most important exceptions in the Supermax Operating System are the following:

- Bus error. That is, access to an inaccessible address or an attempt to write to a read-only address.
- Address error. That is, access to an odd address with an instruction that requires an even address.
- Illegal instruction. That is, an attempt to execute a non-implemented machine code instruction.
- Division by zero.
- Line 1010 trap. That is, an attempt to execute an instruction whose first four bits are 1010. Such instructions are illegal.
- Line 1111 trap. That is, an attempt to execute an instruction whose first four bits are 1111. Such instructions are illegal.
- Illegal trap. That is, exectution of a 68000 machine code TRAP instruction not used by the Supermax Operating System.
- Floating point exception. That is, the occurrence of an error, such as overflow, in the floating point subroutine package.
- Attention. That is, the attention key was pressed on the terminal, if any, from which the process takes input.
- Interrupt. That is, the interrupt key was pressed on the terminal, if any, designated by the process's tty ID.
- Quit. That is, the quit key was pressed on the terminal, if any, designated by the process's tty ID.
- Hang-up. That is, the leader of the process group, to which the process belongs, has died, or the carrier has disappeared from a modem connected to the terminal designated by the process's tty ID.
- Alarm. A process may request the operating system to raise this exception in the process after a specified amount of time.

When an exception occurs, the exception is said to be 'raised' in the process.

The raising of an exception in a process generally causes the process to die, as explained in section 3.2.4. The only exception that does not cause a process to die is the Attention exception, which is described later in this section.

It is, however, possible for a process to 'catch' exceptions by declaring an 'exception handler'. The process informs the operating

system that, if a particular exception is raised, the operating system should call a specified subroutine, rather than kill the process. If and when the exception is raised, the operating system will cause this subroutine to be called; we say that the exception has been 'caught'. Once the exception has been raised, exception handling reverts to default handling until an exception handler is declared again.

A special case of exception handling is to inform the operating system that a particular exception is to be ignored.

To summarize:

When an exception, except the attention exception, is raised, a process may

1) die (default),
2) ignore the exception,
3) catch the exception by execution of an exception handler.

Attention exception handling differs from the handling of other exceptions. There may be several processes running simultaneously, taking input from the same terminal. In which of these processes should the attention exception be raised when the attention key is pressed? Let us assume that the attention key is pressed on the terminal from which processes A and B take their input. What happens is the following:

1) If B is a child process of A and the attention exception is caught in B, the exception will not be raised in A.

2) If B is a child process of A and the attention exception is not caught in B, the exception will be raised in A.

3) If B is not a descendant process of A or vice versa, the attention exception will be raised in both A and B.

The default handling of the attention exception is to ignore it and let an ancestor process catch it. It is, however, possible to specify that the exception should be ignored but not passed on to an ancestor process, or that the process should die when the attention exception is raised.

It is seen that the manner in which a process handles an attention exception is characterized by two things: First, what does the process do when the exception is raised? Second, is the exception considered 'caught', or will it be raised in an ancestor process?

To summarize:

When an attention exception is raised, a process may
1) die, in which case the exception is considered caught;
2) ignore the exception and consider it un-caught (default);
3) ignore the exception and consider it caught;
4) catch the exception by execution of an exception handler.

Generally, exceptions are raised because of an abnormal event in the process or the pressing of a key on a terminal. It is, however, possible for a process to raise an exception in another process. Process A may raise an exception in process B if A is privileged or the effective user ID of A is identical to the real user ID of B; a process may, however, always raise an exception in itself. If the exception raised is the attention exception, it will never be passed on to B's ancestor processes, and in this case action 2 and 3 above are identical.

Raising an exception in another process may be quite absurd. Imagine the surprise felt in process B, when the 'division by zero' exception is raised while B was computing 2+2.

If an exception is raised while a process is suspended, the process is placed in the active state. Therefore, when process execution continues, a possible suspend time may or may not have expired, and a pending i/o request may or may not have been serviced.

3.2.7. Setting the Priority.

When a new process is created through spawning, production, or gemmation the starting process specifies the priority of the started process. Only privileged processes can start processes at a higher priority than themselves.

When a process forks, the new process inherits the priority of the parent process.

The most commonly used priority is 10.

A process may change its own priority or that of another process. Only privileged processes can specify increased priorities. Process A may

change the priority of process B only if A is privileged or the effective user ID of A is identical to the real user ID of B.

3.2.8. Changing Various Process Properties.

A process may, under certain circumstances, change its effective and real user and group IDs. A number of different cases exist:

1) Process A wants to set its effective user ID to the value of its real user ID. This is always allowed.

2) Process A wants to set its effective user ID to a value different from its real user ID. This is allowed if A is a privileged process.

3) Process A wants to change its real user ID. This is allowed if A is a privileged process.

What is said here about changing the effective and real user IDs applies equally to changing the effective and real group IDs.

A process may declare itself the leader of a new process group, that is, change its process group ID to the value of its process ID.

A privileged process may change its tty ID to any value.

A process may change its iounit creation mask.

A process may change its current directory.

A privileged process may change its current root directory.

4. Memory Management.

The memory management part of the Supermax Operating System takes care of the allocation and releasing of memory used by processes.

4.1. The Memory Management Unit.

4.1.1. The Need for a Memory Management Unit.

Suppose two processes are running on an MCU, suppose that they both execute the same program but with different data, and suppose that the program at some time during its execution stores a value in the memory cell located at the address 0x300000. Obviously, the two processes should not store their data in the same memory location, one process destroying the other's data. It is clearly required that the computer should ensure that when the two processes cause the CPU to access address 0x300000, two different memory locations should be accessed.

Thus the need arises for a means by which to keep the addressing space of one process seperated from the addressing space of another process.

Each Supermax MCU is equipped with a hardware Memory Management Unit (MMU), designed to meet this need.

4.1.2. Logical and Physical Addresses.

The addresses accessed by a process are called 'logical addresses'. The addresses of the memory locations actually accessed are called 'physical addresses'.

The job of the MMU can now be formulated in the following manner: The MMU should translate logical addresses generated by the CPU into physical addresses before memory is accessed, and the MMU should ensure that logical addresses originating from two different processes should be translated into different physical addresses.

4.1.3. How the MMU Works.

The MC68000 micro processor, which is the CPU used in the Supermax computer, uses 24 addressing bits, thus creating a (logical) address space of 16 megabyte. These 24 address bits are input into the MMU, which then outputs another set of 24 address bits, that are the physical address of the memory location to access. The MMU thus creates a physical address space of 16 megabyte.

The logical address space is divided into 16 so-called 'segments' of one megabyte each. This means that the first 4 bits of the 24 address bits are the segment number, or in other words, of the 6 hexadecimal digits that form the logical address, the first digit is the segment number.

User programs may use segments 2 through 14, the first two and the last one megabyte (segments 0, 1, and 15) being reserved for the operating system. The logical address space of a process therefore has a size of 13 megabyte.

The following example will illustrate how the MMU translates a logical address, such as 0x456789, into a physical address:

When a process is created, the operating system assigns a unique 'Address Space Number' to the process. The MMU uses the Address Space Number together with the segment number of the logical address to find an entry in a table.

If the logical address is 0x456789, the segment number is 4.

In its internal table the MMU finds the following information:

1) Is this logical address segment allocated to this process?
2) Does the process have read/write access to the segment, or does it only have read-only access?
3) How large a part of the one megabyte in the segment may the process actually use?
4) What offset should be added to the logical address to obtain the physical address?

These four items are detailed below:

1) Of course, not all processes use all 13 accessible segments.

Segments are assigned to the process when it is started or during its execution. If, due to some error, the process accesses an address in a segment that has not been assigned to the process, the MMU detects this and the bus error exception is raised in the process.

2) The MMU allows certain segments to be read-only segments, giving the programmer greater protection against errors. If, due to some error, the process tries to write to an address in a read-only segment, the MMU detects this and the bus error exception is raised in the process.

3) Generally, a process does not use the whole megabyte of memory within a segment. When a segment is assigned to a process, a size is specified. If, due to some error, the process accesses an address beyond the specified size, the MMU detects this and the bus error exception is raised in the process. The segment size has a resolution of 256 bytes. If the logical address is 0x456789, the size of segment 4 must be at least 0x568 blocks of 256 bytes each.

4) When a segment is assigned to a process, the required number of 256 byte blocks are allocated contiguously in physical memory. The MMU knows the location of these blocks. The physical address accessed will be the offset within the logical address segment plus the physical starting address of the allocated memory. Suppose that the blocks allocated for segment 4 of a process start at physical address 0x178a00 and the process accesses logical address 0x456789. The physical address accessed will be 0x56789+0x178a00=0x1cf189.

Typically, a process will use a read-only segment for its program code, a read/write segment for its global data, and a read/write segment for its stack. Suppose that the requirements of a certain program are:

0x6780 bytes of read-only memory in segment 2 for its program code.
0x3020 bytes of read/write memory in segment 3 for its global data.
0x1000 bytes of read/write memory in segment 14 for its stack.

A typical memory assignment for a process executing this program could be:

```
Segment  2: Size: 0x6800 bytes. First physical address: 0x075300.
Segment  3: Size: 0x3100 bytes. First physical address: 0x080400.
Segment 14: Size: 0x1000 bytes. First physical address: 0x023a00.
```

It is seen that the physical addresses allocated for the three segments are not necessarily contiguous. Further, its is seen that even though the process accesses addresses ranging from 0x200000 (first address in segment 2) to 0xe00fff (last address in segment 14), only 0x6800+0x3100+0x1000=0xa900 bytes are actually allocated.

4.2. Basic Memory Requirements of a Program.

In a load module, information is stored about the amount of memory required to start the execution of the program. These memory requirements consist of four parts:

- The so-called text part. This is the part of the program that contains the actual program code. When the program runs it will have read-only access to its text part. Normally, the text part will lie in logical address segment 2 of the program.

- The so-called data part. This is the part of the program that contains the initialized global data. When the program runs it will have read/write access to its data part. Normally, the data part will lie in logical address segment 3 of the program.

- The so-called bss part. (Nowhere in the Unix literature has the author been able to find an explanation of what 'bss' stands for. Can somebody enlighten me?) This is the part of the program that contains the uninitialized global data. When the program runs it will have read/write access to its bss part. Normally, the bss part will lie in logical address segment 3 of the program, following the data part.

- The so-called stack part. This is the part of the program that contains local data and subroutine parameters and return addresses. When the program runs it will have read/write access to its stack part. Normally, the stack part will lie in logical address segment 14 of the program.

In the Unix literature the text part, data part, etc. are often re-

ferred to as the 'text segment', 'data segment', etc. This has not been done here, because the word 'segment' is used to designate something else.

4.3. Memory Allocation.

Memory for a process is allocated when it is started. The process may, however, during its execution request more memory to be allocated.

The initial memory requirements for a process are found in the file containing the load module for the program to be executed.

A lump of memory allocated for a logical address segment for a process is termed a 'partition'. In the final example in section 4.1.3 three partitions are allocated for the process. Each partition corresponds to one logical address segment.

A process may request more partitions to be allocated while it is executing. The process informs the operating system of how much memory it wants and which logical address segment it wants to use when accessing this partition. The maximum of 13 address segments puts an upper limit to the number of partitions that can be allocated.

A process may request its data and bss part to grow or diminish during execution. Normally, this involves increasing or decreasing the size of logical address segment 3.

4.4. Named and Unnamed Partitions.

Partitions created at the request of a running program may be either 'unnamed' or 'named'. Unnamed partitions are the most common, they are deleted when the process dies, and they serve only to fill a temporary need that exists only as long as the process lives.

Occasionally, however, a process wants to leave some data in memory for later use or to be shared by other processes. For this purpose a process may create a 'named' partition. When the process creates the partition it gives the partition a name of up to 16 characters. Other processes may now 'attach' to that partition. This means that they

request the MMU to map one of their logical address segments onto the specified partition.

The data stored in a shared named partition may, for example, be a set of often used subroutines. If these subroutines are reentrant, several processes may use them. There will be no need to have several copies of the routines present in memory.

The static protection scheme used with iounits (see section 5.2.2) is also used with named partitions. When a named partition is created, the operating system stores the effective user and group ID of the creating process as the 'owner ID' and 'group ID' of the partition. The creating process specifies a set of protection bits for the partition. If other processes want read-only access to the partition, their effective user ID must give them read access rights to the partition. A process that wants read/write access to the partition, must have both read and write access rights to the partition.

There is no such thing as write-only access to a partition.

5. I/O Management.

The Supermax Operating System handles input and output in a manner that is, as far as possible, device independent. For example, a line of text is read from a file in the same manner as from a terminal.

5.1. The Unix File System.

The Unix file system is the most important of the two file systems supported in the Supermax Operating System. It is the file system on which access to all other i/o devices depends.

The Unix file system consists logically of three levels:

- the directory level,
- the inode level, and
- the data block level.

Each file on a disk is described by a so-called inode. When a disk is initialized, space is reserved for a number of inodes. The number of inodes sets the upper limit to the number of files that can reside on a disk. Each inode on a disk has a unique positive number. There exists therefore a one-to-one-to-one relationship between a file and an inode and an inode number.

When a file is created and data is written into the file three things happen: On the inode level an unused inode is allocated. On the directory level an entry is created in a directory file containing the name by which the file is to be known and the inode number of the file. On the data block level disk blocks are allocated for storing the contents of the file; the addresses of these data blocks are stored in the inode.

5.1.1. Directorires.

A directory is a file that contains references to other files. Each entry in a directory is 16 bytes long. The first two bytes contain the inode number of the file, the remaining 14 bytes contain a null-terminated pathname component. A deleted entry is indicated by an inode number of zero.

By convention every directory contains at least two entries:

.

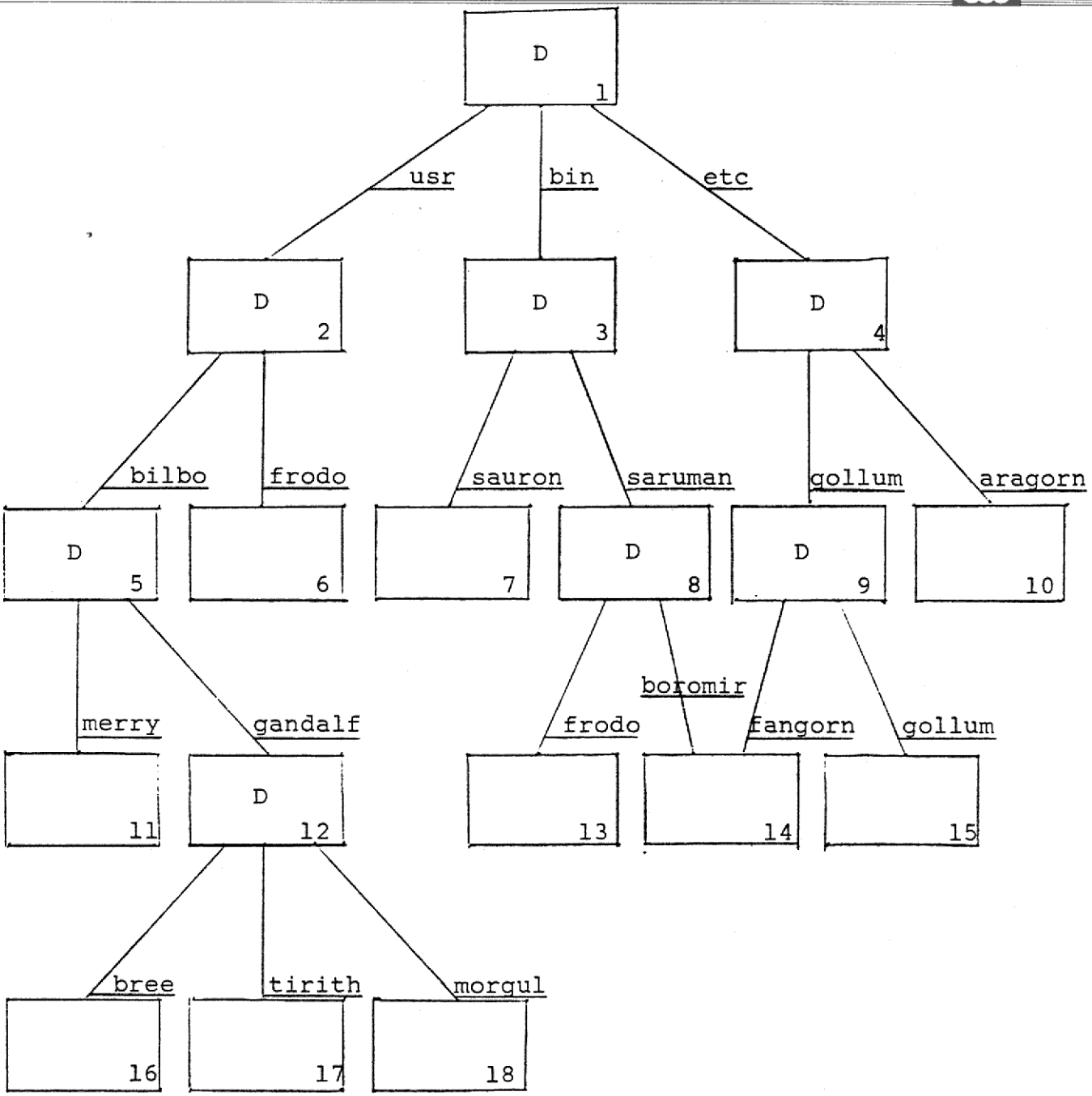
and

..

. always refers to the directory itself.
.. always refers to the parent directory, that is, the directory in which this directory is found.

For the following example, please refer to the figure on the following page. Suppose, for example, that the current directory of a process is /usr/bilbo. A reference in the program to the file called . will be a reference to the /usr/bilbo directory itself. A reference in the program to the file called .. will be a reference to the /usr directory. A reference in the program to the file called ../.. will be a reference to the / directory, the root directory. A reference in the program to the file called ../frodo will be a reference to the /usr/frodo file.

In the root directory the .. entry refers to the root directory itself.


D
1
usr
bin
etc
D
2
D
3
D
4
bilbo
frodo
sauron
saruman
gollum
aragorn
D
5
6
7
D
8
D
9
10
boromir
merry
gandalf
frodo
fangorn
gollum
11
D
12
13
14
15
bree
tirith
morgul
16
17
18

5.1.2. Inodes.

Inodes contain the complete description of a file. An inode is 64 bytes long and contains the following information:

Bytes 0- 1: The file type and protection bits. The file type indicates if this file is a directory, a special file, or an ordinary file.

Bytes 2- 3: The number of links to the file, that is, the number of directory entries referring to this inode.

Bytes 4- 5: The owner ID.

Bytes 6- 7: The group ID.

Bytes 8-11: The number of bytes in the file.

Bytes 12-50: Thirteen 3-byte entries. Each entry is the number of a disk data block reserved for the file (see section 5.1.3).

Byte 51: Not used.

Bytes 52-55: The time of the last access to the file.

Bytes 56-59: The time of the last modification of the data in the file.

Bytes 60-63: The time of the last status change, that is, a change in file size, ownership, protection, or link count.

All times are measured in seconds since January 1, 1970 at 00:00:00 GMT.

As stated in the Supermax Operating System Introductory Guide, references to a given file may be found in several directories under different names. Creating a reference in a directory to a file is termed 'linking the file to a directory'. Removing a reference in a directory to a file is termed 'unlinking the file'. When all directory references to a file have been removed, that is, when a file has been unlinked from all directories, it is deleted. To put it another way: When the link count in bytes 2-3 of the inode becomes zero, the file is deleted, that is, the data blocks and the inode are deallocated.

5.1.3. Data Blocks.

The contents of the file are stored in disk data blocks. Each block can hold up to 512 bytes of data.

As data is written into the file, data blocks are allocated, and the addresses of the data blocks are stored in the inode. In the inode there is room for 13 data block addresses. These addresses are used as follows:

The first 10 entries are addresses of data blocks containing file data.

The 11th entry is the address of a data block containing the addresses of up to 128 data blocks containing file data.

The 12th entry is the address of a data block containing the addresses of up to 128 data blocks each containing the addresses of up to 128 data blocks containing file data.

The 13th entry is the address of a data block containing the addresses of up to 128 data blocks each containing the addresses of up to 128 data blocks each containing the addresses of up to 128 data blocks containing file data.

The total number of blocks allocated for a file is therefore

10 + 128 + 128*128 + 128*128*128 = 2,113,674

which gives a maximum file size of

2,113,674*512 bytes = 1,082,201,088 bytes.

It is quite possible that there are 'holes' in the allocated data blocks: Let us number the data blocks that can be allocated for a given file 0, 1, 2, ..., 2113673. If only a byte here and there are written into the file, it is possible to have files for which, for example, only data blocks 0, 5, 1056, and 5223 are allocated. An attempt to read bytes from the unallocated data blocks, will yield zeroes.

5.1.4. Special Files.

Special files are files that refer to i/o devices. No data blocks are reserved for special files. The inode for a special file contains two numbers called the major and minor device number. The major device number is stored in bytes 12-13 of the inode, the minor device number is stored in bytes 14-15.

The major device number indicates the type of the i/o device: Terminal, printer, null device, etc.

The minor device number indicates the terminal number, the printer number, etc. For certain devices, such as the null device, the minor device number has no significance.

In most cases giving the pathname of a special file leads you to an i/o device. This is, in fact, always the case in standard Unix systems. In the Supermax Operating System there are, however, two important exceptions: Boxes and files on disks with a Mikfile structure. With these two types of i/o devices an extra pathname component is required following the pathname of the corresponding special file. This will be described in greater detail in sections 5.5 and 5.6.

5.2. Operations on Iounits.

Different programming languages treat i/o in very different ways. It is therefore next to impossible to give a general description that will not in some way confuse the users of a particular programming language. The description given in the following sections deals with the Supermax Operating System i/o facilities accessible from the C programming language. The operations described are the low level operations where no formatting of input and output takes place. All i/o requests from programs are in some way converted to these low level requests.

5.2.1. Open, Create, Read, Write, and Move Pointer.

Before it can use an iounit, a process must 'open' or 'create' the iounit. The 'open' operation applies to existing iounits, the 'create' operation to non-existant iounits. The open or create operation establishes a link between the iounit and the process, and from this time the iounit is said to be 'open', and in the rest of this section no distinction is made between an opened and a created iounit.

When an iounit has been opened the operating system returns an 'iounit descriptor' to the process. This is an integer by which the process henceforth should identify the iounit when doing i/o on it.

A process can have no more than 32 iounits open at a given time.

Once a process has opened an iounit it may read from or write to the iounit. This read or write operation may involve either a fixed number of unformatted bytes, or a line of text.

Most iounits seperate lines of text by new-line character (ASCII line-feed 0x0a). The only exception is Mikfile files (see section 5.6).

For some iounits (files and disks accessed without a file system) a pointer to the current byte position on the iounit is maintained. This pointer may be moved by the process, allowing random access to the information in the iounit.

When a process has performed the required i/o to the iounit, the iounit must be 'closed'. This may be done explicitly by the process or automatically when the process dies.

5.2.2. Restrictions on Open and Create.

When a process opens an iounit, it specifies what kind of access it wants to the iounit. The desired access may be

read-only with no reservation
read-only with non-exclusive reservation
write-only with no reservation
write-only with exclusive reservation

read-write with no reservation
read-write with exclusive reservation
selective update with non-exclusive reservation

Not all of these access modes may be applied to all iounits.

The operating system enforces two kinds of protection of iounits: Dynamic and static. A request to open or create an iounit may fail if either of these protection mechanisms forbids it.

Dynamic Protection.

Dynamic protection is the reservation of open iounits. Dynamic protection is enforced on printers and files.

For printers the dynamic protection mechanism forbids a process to open a printer if the printer has been opened (and not yet closed) by some other process.

For files the dynamic protection mechanism allows the following simultaneous open operations:

- If the file is open for an access mode with no reservation, other processes may open the same file for an access mode with no reservation.

- If the file is open for an access mode with non-exclusive reservation, other processes may open the same file for the same access mode.

- If the file is open for an access mode with exclusive reservation, other processes may not open the same file.

The access modes with no reservation are not allowed on Mikfile files. These are the only access modes available in standard Unix systems.

Static Protection.

Static protection is the protection of iounits against access from other users.

All iounits are assigned an owner ID, a group ID a set of protection bits that control who may access the iounit.

The protection bits have the following significance:

Bit 11: Set user ID on execution.
Bit 10: Set group ID on execution.
Bit 9: Save text image after execution.
Bit 8: Grant read access to iounit owner.
Bit 7: Grant write access to iounit owner.
Bit 6: Grant execute or search access to iounit owner.
Bit 5: Grant read access to iounit group.
Bit 4: Grant write access to iounit group.
Bit 3: Grant execute or search access to iounit group.
Bit 2: Grant read access to others.
Bit 1: Grant write access to others.
Bit 0: Grant execute access to others.

Chapter 6 in the Supermax Operating System Introductory Guide describes bits 8-0 of this protection mechanism. Here only bits 11-9 will be described.

Bits 11 and 10 are relevant only for files containing programs. If bit 11 is set, the program, when executed, will have its effective user ID set to the owner ID of the program file. If bit 10 is set, the program, when executed, will have its effective group ID set to the group ID of the program file. If these bits are off, the program, when executed, will have its effective user ID and/or group ID set to the same value as the real user ID and/or group ID. As the effective user and group IDs are used when checking access rights, this facility may be used to grant users controlled access to iounits and system operations that they may not otherwise access.

Bit 9 is relevant only for files containing programs. If this bit is * on, the system will not abandon the swap-space image of the text part * of the file when its last user terminates. Thus, when the next user of * the file executes it, the text need not be read from the file system * but can simply be swapped in, saving time. Ability to set this bit is *

restricted to the superuser since swap space is consumed by the images. *

5.2.3. Selective Update.

Several processes may open a file simultaneously for 'selective update' access. For selective update files the dynamic protection mechanism enforces exclusive reservation on the byte level rather than on the file level. This means that before a process may read or write a given number of bytes at a given location, these bytes on the file must have been reserved. This reservation is called 'byte locking'.

Once a process has locked, say, 100 bytes starting at byte number 500, it may read or write these 100 bytes freely. No other process is allowed to lock and access these bytes. When the desired operation is finished, the bytes must be 'unlocked', whereupon other processes may lock them.

When a process closes a file, all bytes locked by that process are automatically unlocked.

5.2.4. Inheriting Open Iounits.

When a process is started, it may inherit open iounits from the starting process. There are three important differences between opening and inheriting an iounit:

1) The process that inherits the iounit does so regardless of the dynamic or static protection mechanisms. Thus, for example, two processes may both have write access with exclusive reservation to the same file if one of the processes has inherited the open file from the other one.

2) If the iounit is a file or a disk used without a file system, the processes share the same iounit pointer, as opposed to when two processes both open the same file, in which case they will have seperate iounit pointers.

When a process inherts an open iounit from another process, the two

processes are said to share not only the open iounit but the same 'opening' of the iounit.

It is customary that all processes are born with four open iounits. These iounits are called the standard input device, the standard output device, the standard error device, and the standard list device. How these devices are used depends on the particular program.

5.3. Terminals and Printers.

Because hardware devices are physically different, the device independence of the i/o operations can only be enforced to a certain degree. There will always be some operations that are peculiar to specific devices. It is, for example, nonsense to change the baud rate of a file.

Terminals and printers are generically called 'sioc devices'. Sioc is an abbreviation of Serial Input/Output Controller. This section describes some of the peculiarities that apply to sioc devices.

If the output buffer contains characters whose most significant bit is on, those characters will be output underlined. Conversely, if the terminal operator enters an underlined character, that character will be stored in the input buffer with the most significant bit on.

When an input request is issued to a terminal, the operator is allowed to enter and edit a line. The operating system returns the number of actually entered characters to the process, discarding trailing blanks. If more characters are entered than requested by the process, the superfluous characters will be used in the next read operation, unless the process requests otherwise.

A special case of the input operation is the edit operation described in section 3.3 of the Supermax Operating System Introductory Guide. When performed on other devices than terminals and printers, edit operations are converted to the reading of a line of text. When issuing an edit command a process may specify that the cursor is to be left at a certain position before control is given to the operator.

Terminals may be equipped with function keys, some of which are used for line editing. The value of the most recently pressed function key is available to the process.

The position of the cursor at the end of the last input operation is available to the process.

The process may request that its i/o operations to sioc devices be interpreted in a special way. This is known as the 'soft sioc mode'. The options are:

- Should initial control sequences in an output buffer be interpreted? (See section 5.3.1.) Default is yes.

- Should line feed characters in an output buffer be translated into a carriage return and a line feed, as is customary in the Unix operating system? Default is yes.

- Should the terminal echo carriage return and line feed or only carriage return when an input operation is terminated? Default is carriage return and line feed. Changing the echo to carriage return only, will prevent scrolling of the terminal screen when the input line is the last on the screen.

A process may desire to get keystrokes from a terminal immediately when they are entered, rather than allowing the operator to perform line editing. This is done by putting the terminal in 'direct input' mode. When a terminal is in direct input mode, keystrokes are not echoed on the device. Only one process at a time may have a given terminal in direct input mode. The direct input mode may be explicitly cleared by the process or automatically cleared when the iounit is closed.

A process may change the baud rate, the number of stop bits, the number of data bits, and the parity of a sioc device.

Three keys are reserved for creating an exception in a process. They are called 'attention', 'interrupt', and 'quit'.

A process may change the values of the xon, xoff, attention, interrupt, and quit keys.

A process may change the characteristics of a sioc device, that is, the specification on how to move the cursor, clear the sceen etc. on the device.

5.3.1. Control Sequences.

Unless interpretation of control sequences is disabled (see section 5.3), they provide a means for controlling the cursor movement and other features of a specific sioc device.

In order to make the differences between various terminals and printers transparent to the user, these media are controlled by certain character sequences in the output strings. These sequences must begin with < and end with >.

For example, to give an 'erase to end of line' command to a terminal screen, simply output a character string where the first three characters are <Z>.

Note that the < of the control sequence must be the first character in the output buffer. Thus the Pascal statement WRITE('<Z>ALPHA') will erase to the end of line and output the string 'ALPHA', whereas the Pascal statement WRITE('ALPHA<Z>') will output the string 'ALPHA<Z>'.

If several controls are needed in the same buffer they must be enclosed within the same < and >. Thus in order to erase the screen and position the cursor to column 10 line 20 and write the text 'BEER' there, the following Pascal stement should be used: WRITE('<XC1020>BEER'). If the statement WRITE('<X><C1020>BEER') were used, the screen would be erased and the text '<C1020>BEER' would be output in the upper left corner.

If a control sequence contains characters that do not apply to a specific terminal or printer, these characters are ignored.

To output the charcter '<' be sure to include an empty control sequence in the buffer. If you want to write '<34' WRITE('<34') will output nothing, whereas WRITE('<><34') will output '<34'.

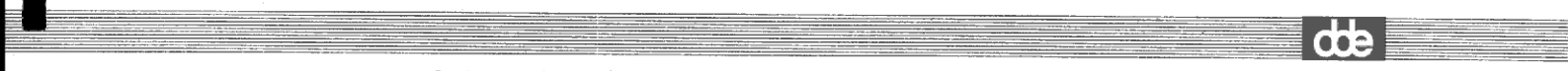
Note that many programming languages provide other means for controlling the sioc devices. For example, Pascal provides the PAGE procedure for ejecting the page on a printer.

If the buffer presented by a program to an edit operation contains a control sequence, the sequence will be interpreted and only the characters following the final > may be changed by the operator. If the edit operation is applied to a non-sioc device, such as a file, a

possible initial control sequence will not be modified by the input operation.

The valid control characters are listed below. The control characters marked with a 't' are supported on terminals only, whereas the control characters marked with a 'p' are supported on printers only.

	<X>	: Clears screen on terminal and moves cursor to upper left corner. Ejects page on printer.
t	<O>	: Clear to end of screen.
t	<Z>	: Clear to end of line.
	<S>	: Omit the final carriage return and line feed after writing a variable length record.
	<N>	: Omit the carriage return after writing a variable length record.
t	<u>	: Cursor up (same func. as <U>).
t	<d>	: Cursor down.
t	<l>	: Cursor left.
t	<r>	: Cursor right.
t	<h>	: Cursor Home.
t	<c>	: Cursor Return
t	<D>	: Cursor to last line column 1.
t	<Cxxyy>	: Cursor to culomn xx on line yy. (Numbering starts at 1.)
t	<Cxxxyy>	: Cursor to column xxx on line yy.
t	<a>	: Delete line.
t	<b>	: Insert line.
t	<e>	: Delete character.
t	<f>	: Insert character.
t	<\>	: Cursor off. The \ is a Ø on Danish terminals.
t	<]>	: Cursor on. The] is a Å on Danish terminals.
	<i>	: Set inverse video on terminal. Start shadow print.
	<j>	: Normal video on terminal. Stop shadow print.
t	<k>	: Set low intensity.
t	<m>	: Set normal intensity.
t	<n>	: Set blinking / bold.
t	<o>	: Reset blinking / reset bold.
t	<p>	: Set invisible.
t	<q>	: Reset invisble.



```
    <+>     : Set underline.
    <->     : Reset underline.
    <G>     : Start expand.
    <I>     : Stop expand.

p   <E>     : Eject page.
p   <Fnn>   : Set page size to nn lines, where nn is an integer.
p   <Hnn>   : Set character width to nn/120".
p   <Lnn>   : Set left margin to column number nn.
p   <Rnn>   : Set right margin to column number nn.
p   <Vnn>   : Set line height to nn/48".
p   <Pnn>   : Set pica size to nn (nn is 10, 12 or 15).
p   <K>     : Start compressed output.
p   <J>     : Stop compressed output.
p   <T>     : Horizontal tab.
p   <Y>     : Vertical tab.

p   <s>     : Half line up.
p   <t>     : Half line down.
p   <A>     : Start proportional spacing.
p   <B>     : Stop proportional spacing.
p   <Q>     : Start automatic justification of right margin (Diablo
              option).
p   <W>     : Terminate all Word Processing (Diablo option).
p   <*>     : Reset printer.

    <'text'>: The text between the apostrophes is output.
```

The last control sequence serves a double purpose:

First, it may be used when it is desired to perform a special operation both before and after a text is output. If, for example, it is desired to move the cursor to culomn 12 on line 10, write the text 'hello', and move the cursor to column 1 on line 3, outputting the buffer <C1210'hello'C0103> will do the job. If the quoted text is to contain apostrophes, two apostrophes should be written.

Second, it may be used to give a prompt to an edit operation. Because text in control sequences cannot be modified by the operator, a text in a control sequence will appear as a non-modifiable prompt on a terminal. An edit operation on the buffer <'Name:'>Jones is different from writing the text Name: and performing an edit operation on the buffer Jones, because in the latter case another process may output a few lines between the 'Name:' prompt and the edit operation.

5.4. Raw Disks.

Raw disks are disks considered without regard to any file system structure on them. When seen in this manner, disks are merely a string of bytes.

Note that the main disk on a Supermax computer may contain a special file /dev/disk0 which refers to the disk itself, seen without a file system. So in a sense, the disk contains itself. (Douglas Hofstadter would leap for joy.)

5.5. Boxes.

Boxes are used for message exchange and synchonization between processes. A box is logically an iounit, but is resident in MCU memory. Data written to a box is stored in a buffer in the box, and from this buffer data is taken when a process reads from the box. Data read from a box is removed and cannot be read again. If a process tries to read from a box whose buffer currently contains no data, the process is supended until something is written to the box. If a process tries to write more data to a box than the box can hold, the process is suspended until so much data has been read that there is room for the data to be written.

There are three kinds of boxes, ordinary boxes, system boxes, and common boxes, associated with special files called /dev/box, /dev/sysbox, and /dev/combox, respectively.

A box has a name of up to 8 characters. The box name is appended to the pathname of the box special file: Reference to an ordinary box called 'saruman' is made by the pathname /dev/box/saruman.

Note here an important difference between box names and file names. For ordinary files the validity of a pathname such as /usr/bilbo/merry implies the existence of an inode called /usr/bilbo/merry on the disk and the existence of a directory file called /usr/bilbo. The pathname /dev/box/saruman, however, does not refer to an inode on the disk and /dev/box is not a directory file - it is a special file.

Ordinary boxes are automatically deleted when they are empty and currently not open by any process. A non-empty box can be deleted by

an explicit delete request. Ordinary boxes are local to an MCU, and their names need only be unique within that MCU.

System boxes are never automatically deleted; a delete request is required to delete a system box. System boxes are local to an MCU, and their names need only be unique within that MCU. Only privileged processes may create system boxes.

Common boxes are never automatically deleted; a delete request is required to delete a common box. Common boxes are common to all MCUs, and their names must be unique within the whole computer. Only privileged processes may create common boxes.

A special kind of ordinary box is the 'pipe'. A pipe is an anonymous ordinary box which a process, A, may create for the purpose of communication with another process, started by A. Pipes are automatically deleted when they are not open by any process.

5.6. Mikfile Files.

The Supermax Operating system supports disks with a Mikfile structure. The Mikfile file system is compatible with the file system used on the SPC/1 micro computer from Dansk Data Elektronik A/S.

A disk with a Mikfile file structure is associated with a special file called /dev/mikfile1, /dev/mikfile2, /dev/mikfloppy, or the like. The minor device number of the special file identifies which of the disks on the computer is used.

Mikfile file names consist of up to 8 characters. Often a so-called file type character is added after the file name, seperated from the file name by a hyphen or a period. Upper and lower case letters are considered identical. The file type may be thought of as a 9th character of the file name; it is used to indicate what kind of information is stored in the file. All characters except n is allowed as file type. If no file type is specified, the file type is assumed to be a space character. The file name is appended to the pathname of the Mikfile special file. Typical Mikfile pathnames may be:

```
/dev/mikfile2/alpha-u
/dev/mikfloppy/beta
/dev/mikfile3/gamma.k
/dev/mikfile3/GAmmA-K
```

The last two pathnames designate the same file.

Note that the validity of the pathname /dev/mikfile2/alpha-u does not imply that /dev/mikfile2 is a directory - it is a special file. And a process cannot make /dev/mikfile2 its current directory.

A Mikfile file consists of a primary file, which is the original file constructed by the create operation, plus 0 to 60 extents. All extents are of the same size as the original file. An attempt to extend a file past the 60th extent is rejected.

In a text file, information is stored in so-called 'variable length records', each record corresponding to one line of text. A variable length record contains the contents of the line, preceded and followed by a byte that indicates the length of the line in bytes. This puts an upper limit of 255 bytes in a variable length record.

For example, in a text file the line 'hello' will be represented by 7 bytes: A byte containing the number 5, the five characters 'hello', and another byte containing the number 5. The contents of the first and last byte must be identical.

After the last line in a text file an end-of-file mark must be found. This is two bytes, both containing zero.

An empty line cannot be stored in a text file. Instead a line containing a single blank is stored.

There is no end-of-file indication in non-text files. The end-of-file condition is returned when an attempt is made to read past the last data block allocated.

5.7. Mounting Disks.

The Supermax computer contains several disks. When the system is bootstrapped the root directory is assumed to be located on disk number 0, the main disk. In order to access files on another disk the disk must be 'mounted'. Mounting a disk implies associating the root directory of the new disk with a directory on an already mounted disk.

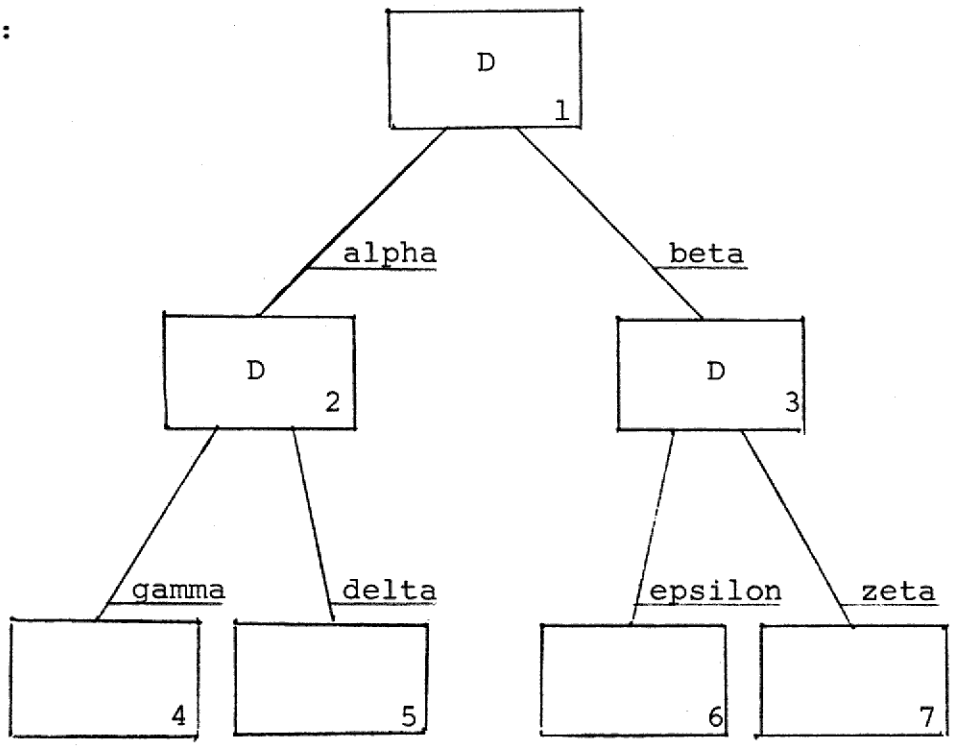
Assume, for example, that disk number 0 on a computer contains the file hierarchy shown on the top half of the following page, whereas disk number 1 contains the file hierarchy shown on the bottom half of the following page.

When the system has been bootstrapped it is only possible to access the files on disk 0. In order to access the files on disk 1 we must mount disk 1 on some directory residing on disk 0. Let us now, for example, mount disk 1 on directory /beta. The mount operation involves the association of the directory file with the special file for disk 1.

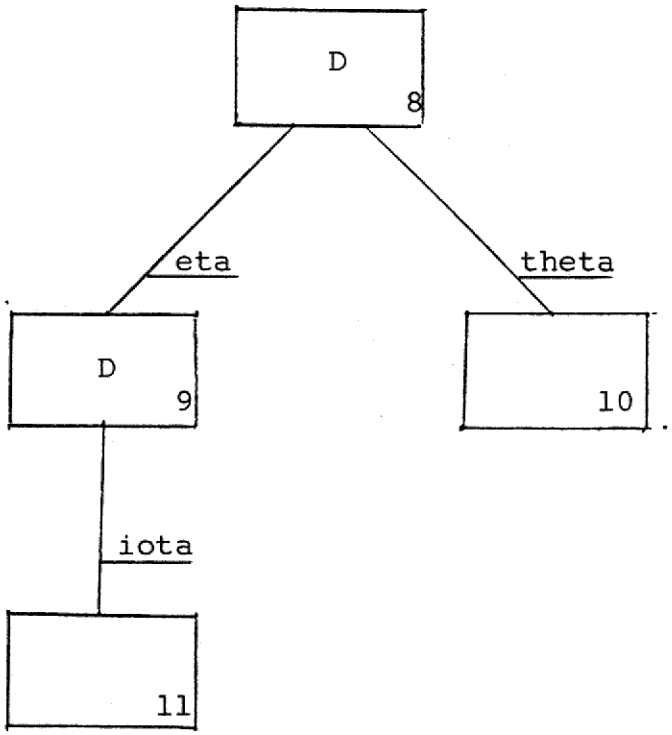
After the mount operation, references to /beta will no longer be directed to that directory on disk 0. Instead /beta will be treated as if it were the root directory of disk 1. So the files labelled 10 and 11 on the figure will have the pathnames /beta/theta and /beta/eta/iota, respectively. The files labelled 6 and 7 are no longer accessible. The .. entry in the root directory of disk 1 will be interpreted as a reference to the parent directory of /beta, that is in this case the root directory of disk 0.

If we had mounted disk 1 on /alpha instead of /beta, the files labelled 10 and 11 would have had the pathnames /alpha/theta and /alpha/eta/iota, respectively.

disk 0:


D
1
alpha
beta
D
2
D
3
gamma
delta
epsilon
zeta
4
5
6
7

disk 1:


D
8
eta
theta
D
9
10
iota
11

In this way the file hierarchy is expanded to cover other disks. There is, however, an important difference between a large file hierarchy located on a single disk and several small file hierarchies located on different disks: It is impossible to link a file to a directory located on a disk different from the one that contains the file.

A disk may be mounted read-only. In this case the files located on the disk may only be read, and the operating system will not itself write anything to the disk. This means that the 'time for last access'-field in the inode will not be modified for files on such a disk. Disks that are physically write-protected should be mounted read-only in order to prevent the operating system from modifiying access times in inodes.

A protection mechanism prevents a process from accessing the Mikfile special file associated with a given disk while this disk is mounted on a directory. Conversely, it is impossible to mount a disk if any process currently has open Mikfile files on the disk.

6. Miscellaneous.

This chapter describes a few facilities of the Supermax Operating System that do not fit into the three main operational categories.

6.1. The System Time.

The Supermax Operating System keeps track of the time. It contains a 32-bit counter that is incremented every second. This counter counts the seconds since 00:00:00 GMT on January 1, 1970. The number of bits in the counter makes it valid until the year 2106.

Of course, this system time is highly unpractical for human use, and therefore most programming languages have facilities to convert this number to normal date and time format, taking into consideration the difference between local time and GMT and the possibility of Daylight Saving Time.

Only privileged processes may change the system time.

There is only one clock in the computer. This means that setting the time in one MCU changes the time in all other MCUs as well.

6.2. Hardware Configuration.

A process may request the operating system to inform it of the hardware configuration of the computer.

6.3. Operating System Version.

Each release of the Supermax Operating System has a version time. This time has the same format as the system time. A process may request the operating system to inform it of the version time of the software that runs in any of the CPUs that make up the Supermax computer.

6.4. The MCU Display.

On the chassis of the computer a number of 2-digit displays are found. Each display corresponds to one MCU. The displays are turned off when the computer is running normally, but during bootstrapping and in case of system crash, a value will be displayed. These values are described in the Supermax System Administrator's Guide.

User processes may output any of 16 values to the display, namely the letter 'u' followed by a hexadecimal digit.

Next to the display a red and a green indicator is found. The green indicator is on when the micro processor is executing in 'user mode', that is, executing the user's program code. The red indicator is on when the micro processor is executing in 'supervisor mode', that is, servicing the user program's system requests, handling interrupts, or idling. Both indicators are turned off if the CPU is halted.

Appendix A. Error Codes.

When a system request fails, the operating system returns an error code to the process issuing the request. The following table lists the error codes. The first column gives the symbolic name of the error code as it is known in the C programming language. The second column gives the value of the error code. The third column gives a description of the error.

A more detailed description of the errors is given in the introduction to part 2 of the Supermax Operating System User's Manual.

General errors:

EOK	0	No error detected
EDATFUL	1	Unirex data area is full
EPRIVIO	2	Privilege violation
EBADADDR	3	Bad address in system call
EBADDIR	4	Bad directive number
ENOTIMP	5	Facility not yet implemented
ECOMFUL	6	Common area is full
EBADPARM	7	Bad value of parameter to system call

Memory management errors:

EPARNX	50	Partition does not exist
EPARAX	51	Partition already exists
ESEGUSE	52	Segment in use
EILSEGNO	53	Illegal segment number
EPARNATT	54	Partition not attached
EPARLONG	55	Partition too long
ENOMEM	56	No memory
EASEGUSE	57	All segments in use
EMAXPAR	58	All partition descriptors in use

Process management errors:

EILPRIO	101	Illegal priority
ENOASN	102	No Address Space Number available
EBADLM	103	Bad load module structure
EBADSER	104	Bad serial number
EPROCAX	105	Process already exists
EPROCNX	106	Process does not exist

EPROGAX	107	Program already exists
EPROGNX	108	Program does not exist
EILMETAM	109	Illegal metamorphosis
EPROCABO	110	Process is being aborted
ERESUME	111	Process was resumed by another process
ENOTSUSP	112	Process is not suspended
EMAXPNO	113	The maximum number of processes exist
EDEADPNX	114	There is no dead process
EBADEXNO	116	Bad exception number
ESIGNAL	118	An exception caused the system call to abort
ESTSHORT	119	The stack is too short to hold parameters
ESYSPR	120	The process is a system process
EMAXINS	121	The maximum number of installed programs exist
EILTIME	122	Illegal alarm request time
EILFORK	123	The process may not fork
EIMPROC	124	In-memory processes prevent a memory change
EILPTRAC	125	The process may not be monitored

I/O management errors:

EBADACC	200	Iounit not open for this access mode
EBUFLONG	201	Buffer is too long
EUNAMLNG	202	Iounit name is too long
EILDEVIC	203	Illegal device
EUNITAX	204	Iounit already exists
EUNITNX	205	Iounit does not exist
EILMODE	206	Illegal access mode
EACCVIO	207	Access right violation
ETIMEOUT	208	Time out on i/o operation
EOPEN	209	Iounit is already open
ENOTOPEN	210	Iounit is not open
EILOP	211	Illegal operation on specified iounit
EILPOSM	212	Illegal position mode
EILBUFL	213	Illegal buffer length
EEXCDDSK	214	Transfer exceeds disk
ENMOUNT	215	Disk not mounted
EAMOUNT	216	Disk already mounted
EOPENFIL	217	Files are open on the disk
EEOF	218	End-of-file reached
EBOF	219	Beginning-of-file reached
EISDI	220	The iounit is already in direct input mode
EISNTDI	221	The iounit has not been put in direct input mode by the calling process

ENREADY	222	Disk not ready
EHARD	223	Hard error on disk
EWRPROT	224	Disk write protected
EILSECT	225	Illegal sector number
ECBOXFUL	226	More than 16 waiting processes on a (common) box
ELOCK	227	The byte range must be locked before access
EFULLLOC	228	The lock table is full
EBADPOS	229	Bad position on iounit
ELUSED	230	The byte range is already locked
EILFSYS	231	Illegal file system letter
EILSIZE	232	Illegal file size or file buffer size
EMAXIO	233	The maximum number of iounits (32) are open
ENOTSIOC	234	The specified hardware unit is not a SIOC
EDISK	235	Internal DIOC error
ECATFULL	240	Calalog full
EILEXT	241	Illegal extent
ELASTEXT	242	No more extents posible
EFULLDSK	243	The disk is full
EILVARI	244	Illegal variable length record
EILTYPE	250	Illegal file type
ENULLFIL	251	Illegal operation on nullfile
EFULLRES	254	Full reservation table
EILFORM	255	Illegal disk format
EILFNAM	259	Illegal file name
ENOINO	262	No inode available
EOLFIL	264	Outside legal file size
EMXNLINK	266	More than 1000 links to a file
EMNTMISM	267	Internal copy of superblock (block 1) for disk differs from physical superblock

Utility program errors:

EILLKEY	300	Illegal keyword
EMISPARM	301	Missing parameter
EILLPARM	302	Illegal parameter
EUPARLNG	303	Illegal parameter length