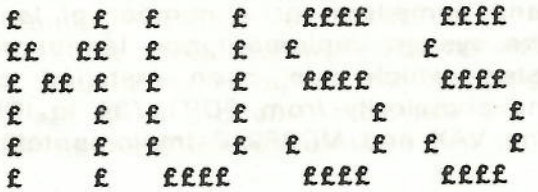
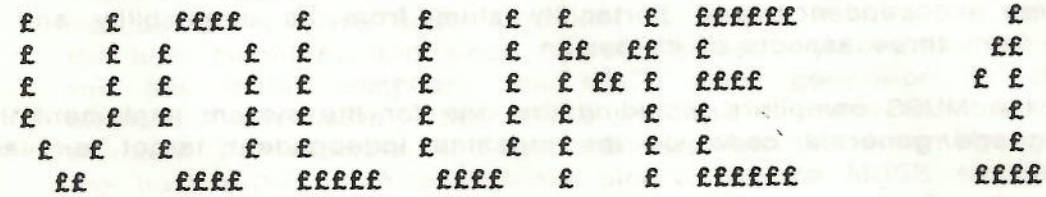
MUSS





BASIC OPERATING SYSTEM

This volume contains some introductory material which is intended to assist those involved with the implementation of MUSS, to new machine architectures, to different peripheral configurations and to give increased functionality. It also contains the implementation of the machine independent operating system kernel.

UNIVERSITY OF MANCHESTER

INTRODUCTION

MUSS comprises an operating system which provides for interactive and background job operation, and compilers for a number of languages including FORTRAN77, PASCAL and the system implementation language MUSL. It is an adaptable and portable system which has been installed on a number of machines ranging in size and complexity from PDP11/34 to ICL2970. The main systems in use at MU are the VAX and MC68010 implementations.

The adaptability of MUSS stems from its well structured design and the strict design/documentation standards on which the implementation is based. Its machine independence and portability stem from its adaptability and in particular from three aspects of its design.

1) All the MUSS compilers including the one for the system implementation language generate code via the machine independent target language MUTL.

2) The Operating System controls the hardware by manipulating a set of variables which represent the peripheral control registers and the special registers of an IDEAL MACHINE. For each actual machine a small amount of machine dependent code is required to map these variables into appropriate actions on the actual machine. This is called the APPENDIX code.

3) The system software is subdivided into modules which relate to its major functions. For example, the Operating System has modules for mainstore management, disc management etc. Each module has a well defined interface with the rest of the system and hence can readily be replaced, if it is unsuitable for a particular machine/user environment. Several alternatives exist for the modules which have needed replacement in the current implementations of MUSS.

In order to implement MUSS for a new computer, the following machine dependent software must be provided.

1) A code generator for the given machine which accepts MUTL and generates executable binary code.

2) The APPENDIX code to map from the IDEAL MACHINE interface to actual control registers in a given machine.

3) Replacement modules, wherever existing ones are unsuitable.

The mechanics of porting the MUSS system to a new machine vary according to the facilities available but the following two methods have been

used.

A) When a machine (e.g. a VAX11/750) which supports MUSS is available for cross compiling a new MUSS implementation, the system can be delivered in source and executable binary form. If the source for the three machine dependent components mentioned above is added, an executable binary version of MUSS for the new machine can be generated. Of the three components mentioned, the code generator is the largest and most complex but existing code generators can be supplied to act as a model. This, like the O/S, is modular and most modules are machine independent.

B) When the initial cross compiling has to be achieved using a non-MUSS machine (called the host machine), an additional code generator is needed for it. This code generator is written in a native language of the host machine, and since its function is only to make it possible to run the MUSL compiler, the MUTL code generator for the target machine, and some other MUSS utilities, it need not be particularly efficient. For example, it might translate into the assembly language of the host rather than executable binary. Before MUSS source can be processed the MUSL compiler must be made to run. For this purpose, an encoded form (called MUBL) of the MUTL translation of the MUSL compiler is provided. Some simple test programs are also provided in the same form. Once the MUBL form of the MUSL compiler can be executed in the host machine, it can generate MUTL (or MUBL) for the rest of the MUSS source. In fact, the MUSS source is normally supplied encoded in the notation of the flowcharting system FLOCODER. Thus the next step after the MUSL compiler is working is to compile FLOCODER. After this the raw MUSS source can be preprocessed by FLOCODER then compiled by MUSL.

The implementation documentation of MUSS is presented in the eighteen volumes and a number of appendices (usually 2 per machine):

THE MUSS USER MANUAL

Volume 1 Basic Operating System Implementation

Volume 2 Basic System Library Implementation

Volume 3 Documentation Facilities Implementation

Volume 4 MUTL Implementation

Volume 5 MUSL Compiler Implementation

Volume 6 FORTRAN77 Compiler Implementation

Volume 7 PASCAL Compiler Implementation

Volume 1 CONTENTS

SYS IMPLEMENTATION DESCRIPTION

Section 0

Section 0.0 Structure of the Operating System

1. Introduction

MUSS is a general purpose, machine-independent operating system. Its design is aimed at adaptability. Thus it can easily be modified to suit the requirements and resources of a particular installation.

To achieve the desired level of adaptability, a highly modular structure has been adopted. The basic module is a process, which can communicate with other processes by sending and receiving messages. A typical system might therefore consist of processes for:

controlling input/output devices
managing the file archiving
initiating user jobs.

Controlled and orderly growth of the system is therefore possible, by allowing new components to be developed as independent processes.

This structure provides the flexibility for users to produce their own modules and tailor the system for their own particular requirements. This obviously requires the existing (trusted) parts of the system to be protected from the effects of errors in any new software. Just as the system is protected against errors in user jobs. In MUSS the same solution has been adopted in both cases. The operating system processes are run within protected virtual machines, which are prevented from interfering with one another. The virtual machines are identical with those provided for user jobs, which has the obvious advantage that new system modules can be developed and tested as ordinary jobs during the normal computing service. A small, highly privileged kernel of software is responsible for multiprogramming the virtual machines, managing their virtual stores, and providing mechanisms for inter-process communication.

2. The Kernel Facilities

The primary function of the virtual machine is to provide a protected environment, with a set of facilities suitable for supporting user sub-systems. The kernel therefore provides a set of Organisational Command Procedures.

available in every virtual machine, which allow a process to manipulate its virtual machine, and perform tasks such as create and delete segments, create and control processes, open files and communicate with other processes. A complete specification of these facilities is available in the "MUSS USERS MANUAL".

Since the virtual machines are protected, the form of inter-process communication provided is very important, as it is the only way in which processes may interact. As the system is intended to be suitable for computer networks, the technique adopted is based on a message switching organisation. Each process within the system is identified by a unique name, and the communication system allows messages to be sent to any named process. The message system is network wide, as indicated in figure 1, and so the destination for a message might be in the same machine or any other connected machine.

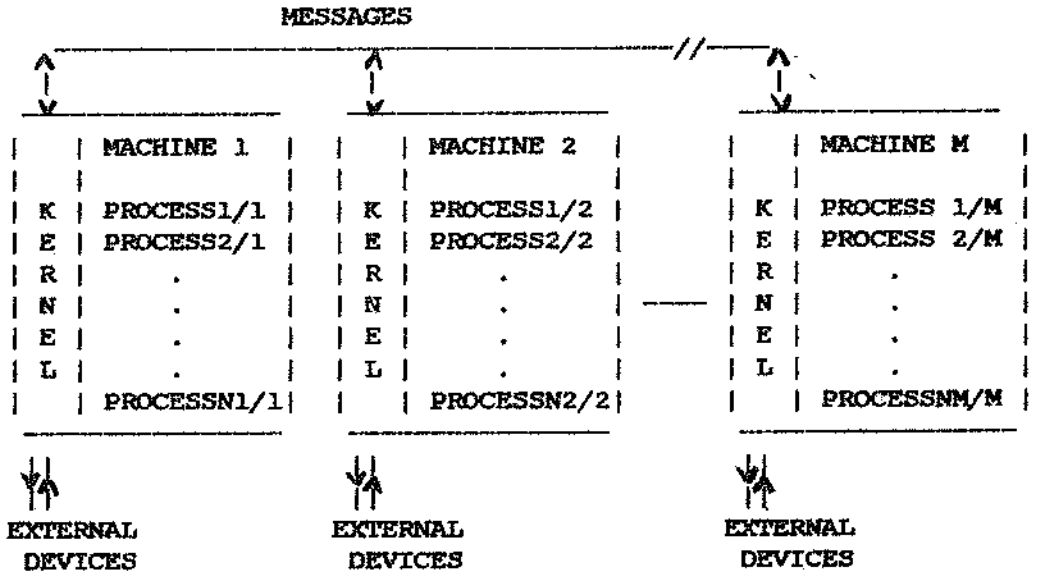


Figure 1 A MUSS NETWORK

To assist in the queuing of messages for a process and to simplify the setting up of conversations between processes, each virtual machine has a number of predefined message channels. A process has to issue an explicit request to read the message on a channel, and messages which are waiting to be read are queued automatically by the kernel. A number of safeguards are built into the system to prevent mischievous users from injecting rogue messages into the communication system. For example, the status of the input channel can be set to refuse all incoming messages, to accept messages only from a specified process or to close the channel after a single message has been received. The recipient process thus has selective control over the acceptance of messages.

The message system handles all communication between processes, and is used for input/output by user jobs as well as for the internal control messages between system modules. The amount of information in a message is clearly variable, and may be large, hence segments of virtual store are passed as part of the message. A short, fixed length header carrying control information always accompanies the message, and where this information is adequate, the segment is omitted. For example, the actual input/output peripherals are controlled by the device controller modules of the system, which provide the interface between the physical devices and the message system. In practice, the interactive and online devices use short messages for passing information, while spooling devices buffer information into a segment and use the long message form.

The message system is also used in connection with file archiving. The main file system is quite closely integrated into the store management part of the system, as open files map onto segments in the virtual store of a process. All file backup and archiving, however, is performed by sending messages to an archiving process. On multicomputer (network) systems, the file system in any one computer can gain access to those of other computers through the message system.

Both the file and inter-process communication systems rely upon having a segmented virtual store, as a segment forms the basic unit of information within the virtual machine. The precise number and size of segments naturally varies between MUSS installations, as it is a function of the hardware. A number of common segments are assumed to exist, which contain shared software such as libraries and compilers which are preloaded into every virtual machine.

3. The User Interface

The normal interface into the system for a user is to establish communication with one of the operating system processes by logging in to it. The basic MUSS system includes a process (or supervisor) specifically for this task (called JOB). The normal role of a job supervisor is to allocate a virtual machine for the user job, and start it using scheduling primitives provided by the kernel. The interface seen by the user is determined by which supervisor process he logs in to. In the case of JOB, the facilities are comparatively simple, being the job control facilities provided by the basic system. New user interfaces providing alternative or more sophisticated facilities can be provided by creating additional job supervisors.

The facilities of the basic system are of two kinds, namely those provided by the privileged kernel, and the procedures within the unprivileged library. The basic system library is primarily designed for scientific computing and general software development. In addition to the mathematical and input/output functions which might be expected in any library, the MUSS system also

regards all compilers, editors and general utility routines as standard library procedures. The entry points are in fixed positions, and directories exist to associate the entry addresses and properties of the procedure with their names. The privileged procedures which form the interface with the system kernel also appear in this directory. A user may thus call a compiler or privileged command as easily as he calls a mathematical function.

4. The Hardware Interface

The MUSS system is designed to be portable across a wide range of machines. To achieve this, it is obviously impractical to base the design of the system on one particular computer, as machine dependencies would distort the system logic, making it unsuitable for different hardware structures. The system is therefore designed for an idealised computer, whose characteristics are sympathetic to the system design and which are easy to map on most real machines. A small amount of machine dependent code (called the appendix code) is therefore required to support the MUSS system on any real machine.

The design of the ideal machine has been guided by two broad principles. The first is that the interface should lend itself to an elegant system structure. This does not imply, however, that the features included in the ideal machine are in any way fanciful or unrealistic, as the second objective is that the design should be a valid proposal for an actual computer. It is therefore reasonable to expect an effective and efficient emulation of the ideal machine characteristics.

Major architectural differences in the target machines present a problem in respect of efficient emulation, as some characteristics, such as the address translation, are so fundamental as to make their emulation totally unrealistic on unsympathetic hardware. No attempt is made therefore to conceal unusual hardware characteristics if this would seriously impair the overall system efficiency. Instead, alternative versions of some modules of the system have been designed to cater for major architectural variations (for example, between paged and multiple base register machines). In effect, the system has been designed for a family of idealised machines, where the most suitable idealised structure can be selected to correspond to the target hardware. The description of each module of the operating system specifies the characteristics of the ideal machine which affect its operation. Selection of an ideal machine structure is therefore implicit in the selection of operating system modules to form a system.

A number of features of the ideal machine have a fundamental effect on the structure of the whole operating system. These features, namely

the control mechanism
the machine states and

the interrupt structure

are therefore described within this introduction, rather than within the individual modules.

The ideal machine, or at least the software emulating it, has to comply with certain requirements as far as its control is concerned. These include the ability to interpret commands, return status information back to the operating system, and generate interrupts in the ideal machine as appropriate. To achieve this, there must be a two-way communication mechanism across the ideal machine interface. The mechanism adopted for this is based on the use of dedicated control registers, similar to those found on machines like the PDP11. Thus, writing to the control registers triggers an action in the ideal machine, which responds by setting a status in the control register (and possibly interrupting the machine) when the required action is complete.

To distinguish the pseudo control registers from other forms of operand in the operating system code, their names are prefixed by "V.". (This is an historical mnemonic derived from the Atlas and MU5 name for hardware registers of "V store"). When emulating the ideal machine characteristics, the control registers may either be mapped directly on to the equivalent actual hardware registers, or when this is not appropriate, the required actions may be performed by machine dependent code. The MUSL language used in coding the operating system has a special form of operand for this latter case, whereby any accesses to a variable defined as VSTORE may also involve a call to a machine dependent procedures. Read accesses may invoke a PRE-PROC, called before reading the variable, and write accesses may be followed by an invocation of a POST-PROC.

With the process oriented structure of MUSS, a number of protection mechanisms and modes of execution are required. Most notably, there must be at least two distinct execution states, corresponding to the "user" and "kernel" software. Here, "user" should be interpreted as including both user jobs and any system modules which may run subject to the protection values of the virtual machines.

The distinction between user and kernel execution is primarily one of privilege. To protect the integrity of the system, certain areas of the virtual store are made inaccessible in user mode. In particular the pseudo control registers and data segments of the kernel. The virtual store mechanism automatically allows access to these segments during kernel execution.

The transition from user to kernel mode may be triggered in two ways: either by an interrupt in the ideal machine, or voluntarily as the result of an organisational command procedure call (equivalent to a supervisor call or extracode). The servicing requirements of these two types of entry are quite different. Accordingly, the ideal machine distinguishes two levels within the

kernel software, namely command and interrupt level. Three modes of execution are therefore recognised:

1. USER STATE(U) — Access only required to the user virtual store. MUSS interrupts allowed.

2. COMMAND STATE(C) — Access to the whole virtual store is desirable. MUSS interrupts allowed.

3. INTERRUPT STATE(I) — Access to the protected segments is sufficient. Muss interrupts inhibited.

The transitions between these states are well defined, and they are illustrated in figure 2.

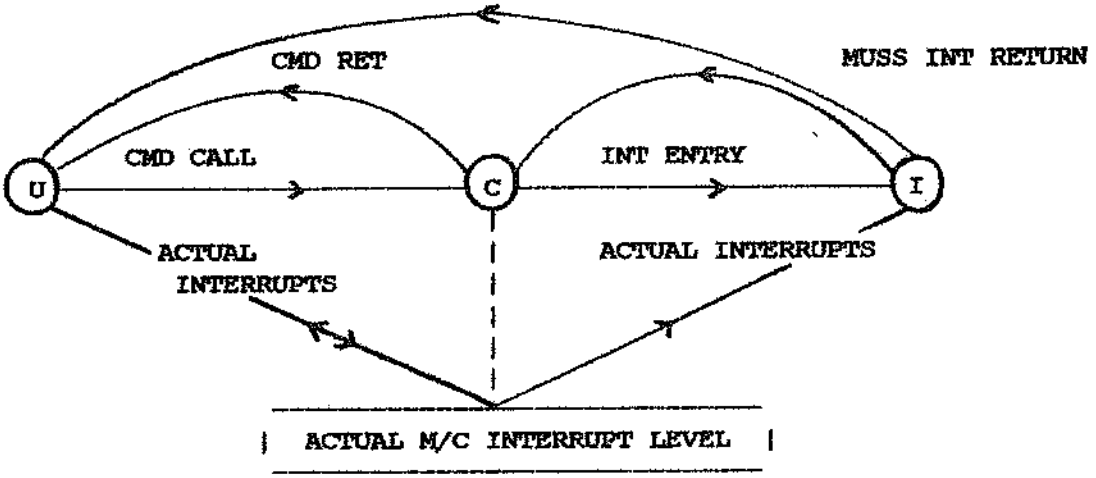


Figure 2. MUSS State Transitions.

The events which cause these transitions, and the actions required in either the hardware, or specially provided mapping software are summarised below

a) CMD CALL - This is an explicit instruction which has most of the characteristics of ENTER except it must set MUSS EXEC STATUS (see below), stack the operand and JUMP to a fixed address in MUSS.

b) CMD ret - This is a PROCEDURE RETURN which also resets MUSS EXEC STATUS.

c) MUSS INT ENTRY - This is implemented by software which is explicit in the MUSS source i.e. there is an INTERRUPT.ENTRY procedure.

d) MUSS INT RETURN - This is explicit code. It will sometimes require to specify the state of MUSS EXEC STATUS.

e) ACTUAL INTERRUPTS – These will usually translate into MUSS INTERRUPTS, although this would not be so in the case of a character interrupt from a device which MUSS assumes to be a block transfer device. Note that the MUSS interrupt level may be interrupted by actual hardware interrupts.

The servicing of interrupts is not logically a part of the normal instruction sequencing of a process. Thus, when the ideal machine enters interrupt mode it performs a complete dumping of the current process state, in addition to setting privilege and inhibiting further interrupts. Each process is allocated a Process Register Block (PRB) for this purpose, and the PRB also holds other process-specific information, such as the segment tables used for address translation.

There are two reasons for interrupts occurring in the ideal machine. The first corresponds directly to a specific event within the currently executing user process, and is therefore known as a process-based interrupt. This includes events such as program faults, virtual store interrupts and instruction counter or virtual machine timer interrupts. Process-based interrupts do not require any elaborate queuing mechanism, as they are defined to be mutually exclusive and always associated with the current process. A process-based interrupt thus has the effect of a forced call to an interrupt mode procedure.

The second class of interrupt, the system-based interrupt, corresponds to an asynchronous event outside the central processor. A real time clock interrupt or the completion of a peripheral transfer would come within this category. As it is unrelated to the execution of the current process, the result of a system-based interrupt is to activate an interrupt level process within the kernel, and in effect perform a complete process change.

The asynchronous nature of system-based interrupts mean that multiple occurrences must be queued by the ideal machine. The queuing mechanism adopted is based on an interrupt decoding tree, where the pseudo control registers for individual peripherals form the outermost branches. An interrupt originating from one of these registers conceptually propagates down the tree until it sets a bit within a single interrupt register. This register may then be decoded to activate an interrupt level process to service the contingency. A detailed description of this structure may be found in the specification for the coordinator (Section 1 - SYS011).

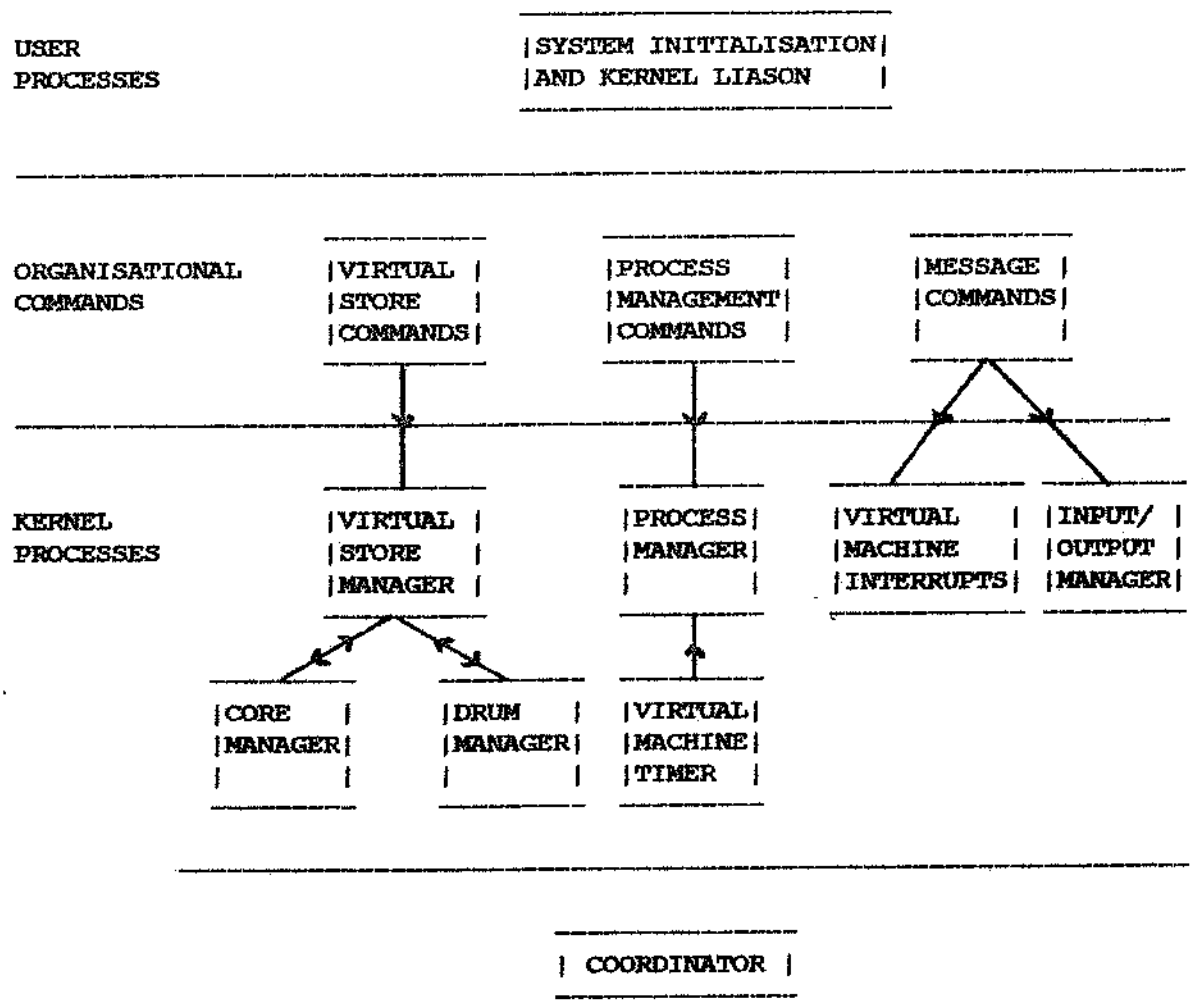
5. The Kernel Structure

The organisation of the kernel is very similar to the structure of the rest of the system, as it is composed of a set of processes operating on their

own private data structures. The set of kernel processes, however, does not change dynamically, and hence static protection measures, built into the compilation system, can replace the hardware protection mechanisms required for running user jobs. They therefore run at interrupt level, sharing a common area of virtual store, rather than each having its own virtual machine.

The external interface presented by the kernel modules is also slightly different, as interactions normally take the form of requests for services, rather than the passing of data. A procedural interface has therefore been adopted in preference to the message switching system. This takes two forms. Each kernel module might include a set of organisational command procedures, which may be called by any user job. These execute at command level: in effect within the current user process but with full kernel privileges. The second form is for interactions between kernel processes running at interrupt level. Here, special procedures called Interface procedures are provided. These are principally used for manipulating data structures and performing similar low level functions. Each section of the operating system therefore has a clearly defined interface, upon which both the users and other kernel modules are dependent. A module of the system can thus be replaced by any alternative version, so long as it conforms to the same external specification. By this means, it is possible to define a number of versions of the operating system modules, which employ alternative algorithms or rely on different hardware structures.

In order to illustrate the structure of the kernel in more detail, it is worth considering the design and function of the modules which might comprise a fairly basic system. One such system is illustrated below:



i) Coordinator.

The coordinator has the function of scheduling the user and kernel processes, and providing the synchronisation mechanisms. The overriding emphasis in this part of the system is on simplicity. This stems partly from an attempt to achieve very rapid process changing, and partly from a desire to integrate the process management functions into the interrupt structure of a potential machine design. Rather than considering all the processes in the system, the low-level process manager controls only a small fixed number of processes (the number being chosen, as an implementation convenience, to be the number of bits in a machine word). At this level, the subset of processes under consideration may thus be represented by bit strings. Some of the processes will correspond to modules in the kernel, while the rest will be assigned dynamically to user processes by a higher level scheduling

module. The advantage of this scheme is that all low-level scheduling and synchronisation operations may be performed by simple logical operations on the bit strings. For example, when a process is halted awaiting a page transfer, it is added to the set awaiting this particular page simply by a logical 'OR' operation. When the page becomes available, another 'OR' operation is used to add these waiting processes to the set requiring the CPU.

The choice of this simple, albeit fairly restrictive mechanism for use at the lowest level of the system is typical of the design philosophy adopted throughout MUSS. The aim has been to produce an extremely general system at the top level, but this does not imply that 'generality' must pervade the entire system. In this particular case, this means that process management within the kernel is extremely simple and efficient, and is not at all penalised by the fact that the system may be supporting hundreds of processes. Furthermore, some rather specialised systems, such as single-user and simple job-stack systems, may dispense with the higher levels of scheduling completely, thus resulting in a very compact total system.

II) Memory Management.

The total memory management system divides into three main modules, viz:

core management
drum management
virtual store management.

In the diagram, the virtual store management module is further subdivided into an interrupt process, the virtual store manager, and the virtual store organisational commands.

The core manager is responsible for the allocation and de-allocation of blocks of core, and has interface procedures enabling other modules to request these functions.

The drum manager, in addition to the space allocation and de-allocation functions, also organises and schedules transfers between core and drum at the request of other modules.

The virtual store manager has the task of mapping the virtual store on to the real store, and maintaining the tables used for address translation. It is therefore this module which has to monitor the migration of pages through the real store hierarchy. Its interface with the rest of the system consists of two procedures, to make a specified page

resident in core, and to update the backing store copy of the page. The first of these functions is generally triggered automatically for user processes by the page fault interrupt. Quite obviously, both the core and drum managers may be invoked when carrying out these functions.

Finally, the virtual store commands form the user's interface with the memory management system, providing the organisational commands to enable a process to create and delete segments or alter their attributes (size, access permission, etc.).

iii) Process management

The system is designed to support a potentially large number of user processes (possibly several hundred), and the function of the process management modules is to keep track of these processes and arrange that they obtain processor time when they require it.

The process manager maintains a pool of virtual machines, and provides a set of organisational commands to make the virtual machines available to users. Each virtual machine is characterised by a Process Register Block, which records the current state of the process executing in it, and the resources allocated to it. Any process, on supplying a valid username and password, can request the creation of new processes, and may subsequently issue scheduling directives on them (e.g. start, stop, change priority, kill). Thus any process has the ability to adopt the role of a supervisor, and may subsequently perform high level scheduling operations on the processes under its control.

The middle level of scheduling within the system is provided by the kernel process associated with the process management module. The function of this is to select the subset of user processes to be considered by the coordinator at any instant. The scheduling decisions are based on the priority and state of each process. Thus, when the state of a process is changed by an organisational command, the process commands have to give notice of the change using interface procedures of the kernel process.

iv) Message Management

The most important aspect of the inter-process communication system is the passing of messages between processes in the same machine. This function is carried out by a set of organisational commands, which together constitute the message manager. The commands manipulate queues of messages in an area of common virtual store, linking them to

"channels" within the register block of the recipient process. In doing so, they are often required to interact with other modules, most notably with the virtual store manager (to transfer segments between virtual machines) and with the process manager (to halt processes awaiting messages and free them when a message arrives).

v) System initialisation

This module is not, strictly speaking, a part of the kernel, as it is a full process which is pre-loaded into the system. Being the first process in existence, it is known as Process Zero. Its operation can be divided into two distinct stages. When the system is first started, it has to initialise and check the system's data structures, particularly those of the file and accounting systems. It then has to create and start the remaining system processes.

During the normal running of the system, its role is to support the kernel activities. This involves primarily liaising between the kernel processes and the message system, monitoring operator messages and performing some of the accounting functions.

Process zero is a somewhat unusual module, in that its main function is to call interface procedures of the other system modules. A special type of interface procedure exists for this, known as a task procedure. Thus, a kernel process will invoke process zero by setting a task for it. Process zero (eventually) responds by calling the appropriate task procedure to perform the required actions. A full description of the task mechanism may be found in the specification of process zero (Section 20 - SYS201).

6. Notes on the Implementation of System Modules

A module of the operating system may consist of components which execute at three levels: interrupt level, command level, and task level. In practice there may also be modules, implementing the idealised hardware interface, and the code here is known as appendix level. The following notes are intended to give guidance to the implementor as to the kinds of functions to be placed at each level and the coding conventions which apply at each level.

The reader should first have read and understood the preceeding description of the overall operating system structure. It will also be helpful to read the documentation for the co-ordinator (Section 1 - SYS011), process manager (Section 13 - SYS131) and process zero