A MODULAR OPERATING SYSTEM

Brian Wichmann
National Physical Laboratory, Teddington, Middlesex. England

Information Processing 68 — North-Holland Publishing Company — Amsterdam 1969

Abstract

The following contains a description of the principles and techniques used in a modular operating system being implemented at the National Physical Laboratory. The computer being used is a KDF9 with 32K words of core store and a disc file. The system provides a time-sharing and file handling capacity as well as the usual services to users.

1 Introduction

The paper contains an outline of the system in a form as machine-independent as possible. This is followed by a detailed description of the main parts of the system to give the reader an accurate account of how the problems of a time-sharing system have been tackled.

2 Outline

Any operating system, other than the most trivial, consists of many programs (compilers, etc.) plus a library system, routines for dealing with backing store, peripherals and external interrupts, etc.

Conventional systems (at least in Britain) deal with this by having a large supervisor permanently resident in core which usually performs the following tasks:

- Job organisation;
- I/O well handling;
- Peripheral handling;
- Interrupt handling;
- Operator interventions;
- Various subsidiary tasks.

Such systems are very difficult to understand, since they are usually one large, tightly coded machine-code program with little overall strategy which hinders possible amendments. A further difficulty is that the supervisor is essentially a real-time program, in which at any one moment, many tasks will be in various stages of execution.

A solution to this problem has been proposed by Dr. J. L. Martin[1] which allows all the tasks of a supervisor to be performed by independent programs, called *modules*, which interact via a minimal program called the *interface*. Modules are similar to user programs, which allows part of the system to be written and tested as user programs before being inserted in the system.

A module is the unit of programming, consisting of four parts:

1. status description;
2. name, type and operating requirements;
3. object code and working space;
4. extra space (if any) allocated to the module.

1. The status description consists of the following:

   - the priority given to running of the module;
   - the computing time spent in the module;
   - inhibition due to one of a number of causes, for instance, failure;
   - waiting for a module to become available;
   - waiting for a peripheral transfer to finish;
   - waiting for operator intervention.

   Failure is, in fact, a form of waiting — waiting for the failure routine to produce a report and delete the module from the core store. A dump for the registers of the machine must also be provided.

2. Modules refer to each other by using the name of the module. So this must also be stored with the module. Modules can also be of different types, allowing them to perform different functions. For instance, a compiler must be allowed to read any program text whereas user programs can only read a limited range of files. This is achieved by having two different types of module— system and user.

3. The object code of the module must clearly be relocatable; to achieve this the hardware base-address register is used.

4. As well as the object code and working space it is possible to allocate more space to the module, which can be used by the system for extra information about the module which the module is not allowed to access.

The purpose of the interface is to deal with interrupts. This entails, for instance, inhibiting modules which cause an interrupt by obeying an invalid instruction. After dealing with the reason for the interrupt, a new module must be chosen to run. This involves searching through the list of modules in core store and picking an uninhibited module of highest priority. To ensure there is always one module to run, a module of permanently low priority is always available. This is called IDLE since the time spent in it represents the wasted time on the machine. The interface also maintains a clock for each module, and alters the priority of modules as circumstances demand.

[1] Now at Department of Physics, Kings College, London.

The interface also passes messages between modules on request. The message is, in fact, the contents of certain registers when a programmed interrupt occurs. In the usual case, a module passes parameters to another module which does some calculation and then returns control to the calling module. This corresponds to the call of a subroutine and the return from it. Some modules have special capacity to violate the rules which apply to the user modules, enabling them to start up new lines of computing or inhibit lines which have exceeded the time allocated to them. The messages available to the user are essentially a subset of those available to the system modules.

The modules in the core store are chained together in an arbitrary order. So the interface needs only the address of the first module in the chain, and the address of the current module to function correctly. Hence the interface is a very small program compared with a conventional supervisor. It is currently about 400 words long.

If for any reason it is impossible to process immediately a programmed interrupt, for instance, if a message is to be passed to a module which is busy, then the module wishing to pass the message is preserved in the state it was at the interrupt, but inhibited. This inhibition is removed whenever any module becomes free, allowing a second attempt to be made at passing the message.

It is clearly necessary to have in core a module to load new modules into core from the backing store as requested. This request will come from the interface on meeting a programmed interrupt requesting a module which is not in the core.

This loader is the only module which needs to know the core map, since it only needs to find room for further modules. If necessary, it will dump infrequently used modules onto backing store — a form of segmentation. The characteristics of this module and the use that the system makes of it will clearly depend critically on the particular configuration. In our case, with a rather slow disc, it is hoped that all the frequently used modules will usually be in the core store.

In any multi-job environment protection arrangements must be enforced to ensure that one job does not interfere with another. These security arrangements fall into three classes.

1. Core security. One must have hardware[2] capable of ensuring that no module reads or writes to core not allocated to it. In our case this is done by base address and limit registers.

2. File security. This is a function of the file access module. Only this module must be allowed to access the hardware (hardware protection is required for this). The file access module must validate all requests for access to files before action is taken. In our case, there is no common data base, so each user has his own files which only he can access (or the system on his behalf).

3. In a similar manner to the file access module, all modules offering a general service must validate all requests according to the rules laid down in the design of the system.

The above provides a framework with which an operating system can be written. The object of the system is to provide an effective service to the users and hence a description of the systems as seen by the user should be given here.

The user communicates to the system off-line by a sequence of commands starting with an identification command. The module which reads the paper tape containing the

[2] Since users may write in machine code.

commands merely queues them on a file on the disc. Each user is treated independently so there is a queue for each one. A further module periodically inspects the queues and starts new lines of processing when appropriate resources are available. So there is at each moment at most one process active for each user. Usually the system is active only for a few users, the rest either have commands waiting to be dealt with or output to be printed.

The commands being implemented at the moment cover input and output of files, amending text files, compiling assembly code programs, and running binary programs.

3 Hardware

To give an accurate description of the system it is necessary to explain those features of the hardware which have been exploited. KDF9 is a stack machine. All arithmetic is done in an accumulator stack of 16 cells. The machine code is thus zero address in many cases. There is also a subroutine link stack of the same depth. Unfortunately the machine logic does not permit arbitrary sized stacks to be simulated easily. This is because, when a violation of the stacks occurs, the interrupt that it causes is not immediate. This means that the interrupt handling routine cannot reconstruct the state of the program immediately prior to the stack violation.

KDF9 is a fully protected time-sharing machine. Each peripheral and the core store is protected by special registers against misuse. The core protection is by base address and limit registers. The registers involved can only be set by the interrupt handling routine. Interrupts are also provided for invalid instructions, programmed interrupt, end of peripheral transfer, peripheral hold-up, operator intervention and a one-second interrupt.

The accumulator stack can be represented by an array

stack [1: stack level] stack level < 16.

Each element of the stack is one KDF9 word of 48 bits which ordinarily represent one floating point number, or fixed point integer.

Similarly the subroutine link stack is an array

link [1: link level] link level < 16.

This is used by the hardware for planting return addresses and returning from subroutines. Link [link level] is the address of the current instruction. The elements of the array are 16 bits long or one address length.

Any violation of these stacks (i.e. under- or overflowing them) causes a special interrupt.

The state of the machine is defined, not only by the contents of the stack and link registers but also by a number of control registers. The most important is a boolean *niff* (no interrupt flip-flop). This register is set to *true* by an interrupt and cleared by a special return to program instruction. All the special instructions associated with protection of the machine can only be obeyed if *niff* = *true*.

The reason for interrupt register is logically a boolean array *rfir*[0:47] which is read by one instruction. Each suffix corresponds to different reasons for interrupt as follows:

rfir[0] = *true* when a clock interrupt has occurred;

since the position of the particular interrupt is immaterial one can put *rfir*[clock] = *true* when a clock interrupt has occurred.

The other interrupts are as follows:

- *program ready* (caused by the end of a peripheral transfer)

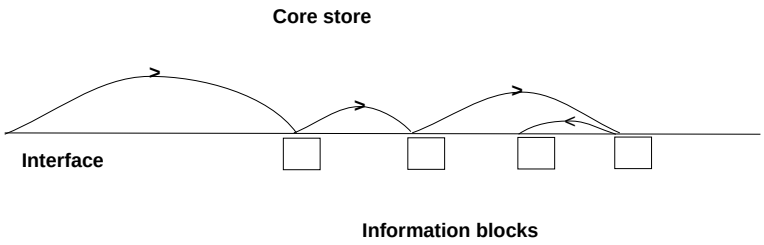

Figure 1:

- *flex* (caused by the operator pressing a button on the monitor flexowriter)
- *liv* (caused by a program violating the protection arrangements)
- *nouv* (caused by a stack vioIation)
- *edt* (caused by the end of a peripheral transfer)
- *out* (caused by the program requesting an interrupt)
- *lov* (caused by a peripheral hold-up)
- *reset* (caused by obeying an invalid instruction).

The only interrupts which can occur when *niff = true* are *out* and *reset*.

There are two special protection registers. The first is the current peripheral device allocation register. This is logically a boolean array *cpdar*[0:15]. A user can access device i if and only if *cpdar[i] = true*. This entire array is set by a single instruction. The other special register contains both the base address and limit registers for core protection. This is set by one instruction. Both the addresses are multiples of 32 words.

Every module has a block of information of fixed size relating to that module. This contains all the information required by the system to run that module. The only word in the block which does not refer to the module itself is the chaining word giving the address of the next information block (see fig. 1).

4 Information block for each module

Specifically, the information block contains the following, apart from the chaining word:

- *identification* i.e. user's number, system or user type and the user's name for the module.
- *mandatory time limit*
- *time spent in module*
- *dump for registers* (including the protection registers).

If a module cannot run for any reason, it is inhibited by setting an element of a boolean array to true. Each element of the array corresponds to different reasons for inhibitions, some of which are permanent (for instance, invalid instruction) and some temporary (for instance, peripheral transfer inhibition). A priority is also kept for each module, the interface choosing the uninhibited module of highest priority to run, after an interrupt. In the case of a module being called as a submodule, a record of the identification of the calling module is kept. The contents of the information block are indicated semi-formally as follows:

- *chaining word* = 0 (meaning last module in chain) or (address of next module in chain)
- *identification* = (module type) and (user's number) and (user's name for module)
- *timing information* = (mandatory time limit) and (time spent in module)
- *submodule call information* = O (if module not submodule) or (number of parameters required) and (identification of calling module)
- *inhibition bits* = (inactive) or (protection violation) or (stack violation) or (invalid instruction) or (invalid programmed interrupt) or (programmed interrupt would cause stack violation) or (copy of module to be removed from core) or (peripheral hold-up) or (module not available) or (submodule call) or (removed by operator intervention) or (removed by next clock tick (every second))
- *priority* = integer between 0 and 127
- *space priority* see description of loading module
- *space allocation* see core store section
- *registers* = (hardware protection registers) (program registers) The hardware protection registers are: base address register, limit address register, current device allocation register, The program registers are: Test and overflow registers, stack[1:16], link[1:16], stack level, link level, modifiers[1:15].
- *core address counter* see description of the loading module.

5 The core store

The information blocks for modules are chained in an arbitrary way. The only module allowed to access this chain, apart from the interface, is WANTED the loading module. The reason for this is that WANTED may find it necessary to move a module in the core store in order to make room for a further module. If this happens during another module's search down a chain, failures would result. Occasionally it is necessary for a system module to know the address of another module. This occurs in the case of the DISC module which needs the addresses in order to perform the peripheral transfers from the disc to the correct position. In this case the interface increments a special count (the core address counter) on giving the required address and decrements after the transfer. WANTED will only move the modules when this count is zero.

Fig. 2 illustrates the core store allocated to the user. There are three distinct cases:

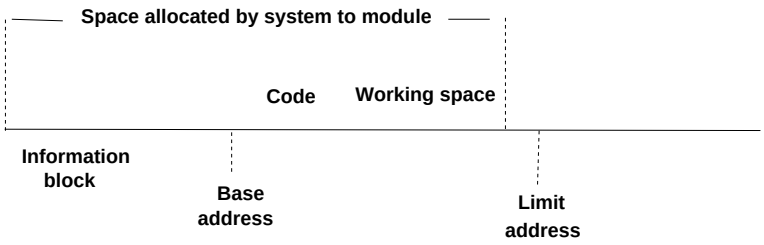

Figure 2:

1. Limit address > end of working area tas illustrated). In this case, the module can access information other than his own. This is reserved for trusted system modules, which require to access the core of another module. There is no need for the system module to occupy a lower core address, since the machine is cyclically addressed.

2. Limit address = end of working area. This is the usual case, as for user modules. Note that the information block is not accessible.

3. Limit address < end of working space. Here the system has reserved space for that module which the module cannot access directly. This facility has not been used in the system but does provide for a method of extending the system information about a module in excess of that provided in the information block which is of fixed length.

6 The interface

As has been stated above, the interface mere- ly deals with interrupts. This takes four distinct stages, namely:

1. Dump the interrupted module;

2. Act upon the reason for interrupt;

3. Choose new module to run;

4. Restore and run the chosen module (i.e. the opposite of stage 1).

Stage 2 is divided into a number of different cases one for each type of interrupt. The interrupts themselves are divided into two classes, those necessarily caused by the current module (i.e. peripheral hold-up, invalid instruction and programmed interrupt) and those which are external (in general) to the current module (i.e. end of peripheral transfer, clock and operator interrupts).

The exact action of the interface is illustrated as follows:

dump: preserve interrupts and time, dump accumulator stack, dump modification registers, dump link stack, dump overflow and test registers.

timing: update computing time for current module, insert inhibit bit if mandatory time limit exceeded.

action from interrupt:

- current module: if peripheral holdup set, insert inhibit bit; if invalid instruction set, insert inhibit bit; if programmed interrupt then process programmed interrupt.
- external to current module: if operator interrupt remove bits from all modules; if end of peripheral transfer interrupt remove bits from all modules; if clock interrupt remove bits from all modules.

choose uninhibited module of highest priority

restore module: restore overflow and test registers; set current peripheral device allocation register; restore accumulator stack; restore modification registers; restore link stack; update time; set base address and limit registers and return.

The most critical part is processing programmed interrupts which is detailed in section 7.

7 The programmed interrupts (on KDF9 these are called OUTS)

Programmed interrupts are the sole method for modules to communicate directly with each other. The interpretation put upon the contents of the accumulator stack which determines the action taken by the interface depends upon the type of the module. This can be either system or user and could be extended to further types. A typical case of a programmed interrupt is the entry of a submodule. The contents of the stack for both system and user modules are as follows:
stack level A
(name if submodule)
(parameter 1 for submodule)
...
(parameter n for submodule)

In this case the action of the interface is as follows in flow diagram form:
At least two cells in stack?

No set invalid out bit of calling module, Exit.

Yes module whose identification is type and user number as for calling module; name as in second element of stack.

Is in core?

No if WANTED busy, then set unavailable module bit, in calling module, Exit
Otherwise: set up call for loading module into core, by making WANTED a submodule of calling module, but returning to OUT instruction.

Yes If required module is busy, then set unavailable module bit in calling module, Exit

Otherwise: transfer cells 3 onwards to submodule. The number of cells transferred being up to the number set in the information block of the submodule.

Delete A (name of submodule) and transferred cells from calling module.

Set submodule inhibit bit in calling module.

Set submodule to active, Exit

The general strategy is as follows. The OUT is first validated, if it is not valid the module is inhibited with the appropriate bit, otherwise the processing of the OUT is carried out. If reference is made to a module then two cases arise (i) in core or (ii) not in core. If the module required is in core and available then the OUT can be processed completely. In the case where the module is in core but not available the calling module is inhibited by setting the waiting-for-module bit. This bit is removed from all modules whenever a module becomes free. This in turn will allow the OUT to be reprocessed, since the module will reobey the OUT instruction. This method is somewhat inefficient since an OUT can be reprocessed a number of times, but the alternative of queuing requests for modules would consume storage for the queues and would require more coding in the interface. In practice OUT's are not likely to be frequent enough nor the number of modules in the core store likely to be large enough to make this algorithm consume more than a few percent of the processing time. Since this method depends solely on the interface coding alternative methods could be tried by altering only the interface.

In tables 1 and 2 are given the OUT's available to the user and system modules. Every OUT available to the user is available also to the system modules. This is a summary of documentation provided with the system by Dr. J. L. Martin.

8 The disc module

The disc module is the only module allowed to access the 4 million word disc which is extensively used by the system. The disc is divided into files by a dynamic storage allocation algorithm. Each file, like each module, has its identification in two parts. The first part contains the user's number and an additional serial number if required. The second part is the user name. A user can only access those files with his user number and whose serial number is zero. The serial numbering is used to put files for output (say) into the output queue immediately this is requested, giving the user an opportunity to create a new file of the same name. This is done by putting a serial number into the identification of the file making the file no longer accessible to the user. Thus to the user, the printing is apparently immediate although it may be some time before the file is output and deleted from the system.

The commands to the disc module for the user are as follows: create, recall, save, delete, read, write and insert. Every file is a certain number of sectors (disc storage units of 40 words) long. This is requested when the file is created. Every file is either open or closed, the user only being allowed to operate on open files. A file is closed, i.e. saved for subsequent use by the command (save) and reopened by the command (recall). One can read or write any consecutive set of sectors in a file. The command 'insert' allows one to increase or decrease the length of the file after it has been created.

Additional system functions are provided for various purposes including altering the serial numbering of files as explained above.

Contents of the top of accumulator stack	General effect	Module called
A	Call submodule	Text and user number from calling module, name given in accumulator stack
X	Call next segment	Type and user segment number from calling module, name given in accumulator stack
Y	Return after failure	Module doing failure this OUT is a submodule so return is made to the module whose name is in the information block
Z	Return	As for OUT Y
D	Call DISC module as submodule	Note: special OUTS module as are provided so submodule that users may call ” service” system modules
F	Call FAIL module	

Table 1: User programmed interrupts

Contents of the top of accumulator stack	General effect	Module called
0	Terminate module	
1	Call submodule	Type and user number together with the name are both given in the accumulator stack, allow system modules to call any module
2	Call module as separate process	The parameters are as for OUT 1, but the calling module carries on processing so an extra line of computing is set up
7	Call next segment	Exactly the same as OUT X except that any module can be specified

Table 2: System programmed interrupts

9 The loading module

As already mentioned the purpose of the load- ing module called WANTED is merely to find room to load modules into the core store[3]. This is achieved by deleting erasable modules from the core, dumping modules onto the disc and moving modules where necessary. At any one moment a module may be in one of a number of statuses, i. e.

1. inactive and erasable
2. inactive and capable of being dumped on the disc
3. active and capable of being moved in core
4. active and not capable of being moved.

Modules of status a) can be removed immediately after it is found that there is no gap in the core to load the required module. If this does not free a sufficient area then an attempt is made to see if the total of the gaps between immovable modules is adequate. If it is, then all the modules between the immovable modules are justified to form the required gap. If this does not succeed then a search is made to see if moving modules over the immovable module boundaries will provide the required gap. The final stage is to see if dumping of modules will allow an adequate gap to be formed. A failure here will cause the attempt to be given up until WANTED is called again.

When dumping modules onto the disc, WANTED attempts to dump the modules of lowest space priority first. Hence, by arranging the space priority of system modules appropriately one can ensure some modules to be kept in core if this is required.

10 The queuing system

The essence of the controlling system is the method used to queue user commands and output[4]. There is a separate queue for each user for his commands to the system and a separate queue for each output device. In order to ensure that the modules concerned with input and output of queue elements cooperate correctly, one module is responsible for queue management which is called QUEUER. For each queue there is one module responsible for input to the queue and another module for output from the queue. Call these two modules INPUT and OUTPUT respectively. The module INPUT does the following: asks QUEUER for the serial number of the next element in the queue, then forms this element, and then informs QUEUER that the element has been put on the queue. INPUT then recycles or returns if it is a submodule as appropriate. QUEUER on being told an extra element has been put on the queue by INPUT, checks to see if there is exactly one element in the queue. lf this is the case, it starts the OUTPUT module as a separate process. The OUTPUT module is given the serial number of the element of the queue to be output by QUEUER. It outputs this element and then calls QUEUER as the next segment. In this case QUEUER either calls OUTPUT again if there are further elements in the queue or terminates itself (and hence OUTPUT) otherwise.

[3] Summarised from "An Algorithm for scheduling storage on a non-paged machine", *Computer Journal*, May 1968, p. 17, by D.C.Knight.

[4] Summarised from system documentation provided by G.G.Alway.

11 The command system

The user requests action from the operating system by a sequence of commands on paper tape[5]. He must precede his commands by an identification message as follows:

⇒⇒ON→ (name allotted to user by management)

All subsequent messages will be assumed to come from this user until an OFF message is met or until a further ON is encountered. The off message is as follows:

⇒⇒OFF→

This also has the effect of producing a log of the commands carried out by the system and details of cost, etc.

The commands currently available enable compiled modules to be read into the system, for such modules to be run, and also files to be read in, printed or punched.

Work is currently in progress to insert a modified version of the assembler for KDF9 together with an editor.

12 Assessment of the system

There are about twelve modules permanently resident in the core occupying about 2500 words. The overhead involved in processing a user command is about 7 seconds, mainly of disc time. Most of this time will be spent in other modules if there are any in the core requiring computing time and little disc time. The amount of time spent in IDLE will depend critically on the job mix. These figures compare favourably with the two conventional operating systems for KDF9, although an exact comparison is not possible as the three systems offer different facilities to the user.

13 Acknowledgements

The work described above has been carried out as part of the research program of the National Physical Laboratory. The team included Dr. J. L. Martin (now at Kings College, London), Mr. G. G. Alway, Mr. M.Woodger and Mr. D. C. Knight (also now at Kings College, London).

14 Discussion

Remark by I. C. Pyle

Along with other speakers you have stressed the importance of simplicity. In this system I think you have gone too far in the direction of simplification and that it would be difficult to implement new features. I have two specific objections to your system: 1) Fragmentation of core arises because of deletion of variable size modules. 2) You have made the FORTRAN compiler privileged. If I invoke the FORTRAN compiler it should operate on my files, not anyone else's.

Answer

As far as simplicity is concerned, we tackled the project with the idea of maximizing simplicity. Whether the system is viable or not depends on how often transfer goes from one module to another and on how long this takes. This depends on the machine. Compilers are only privileged in the way they use files, and are not privileged in the

[5] Summarised from system documentation provided by G.G.Alway.

way they use core. This means that the operating system must validate very carefully the parameters passed by the user, so that it only accesses the files belonging to the user.

Remark by B. Hansen

I should like to support your design. You have created a very general design and I feel that a logical extension would be the ability to create modules dynamically, and to control, stop, start, etc. so that one module could control another.

Answer

There is only one way to start a new process and that is through an interrupt, which is only available to system modules. The user can call sub-modules, but he cannot initiate a process. However, this restriction on the user does not cause any difficulties. Recursion is not allowed in modules. System modules are controlled by careful mapping of the system.

Question by D.M.Rowe

What does your system do for the user that the manufacturer does not?

Answer

The system was started with the intention of radically reorganizing the supervisor. After some investigation we came to the conclusion that we could not do this, so we had to show that it was viable for the user.

A Document details

Converted, October 2001. Original was double-columns.