Abstract:

An attempt to transport an operating system and some associated software from one machine to another is described. Some comments are made on the suitability of the system for such a task, and some recommendations are put forward to future operating system writers to assist in making their systems more portable.

An Exercise in the Transportation of an Operating System

By

C.R. Snow

TECHNICAL REPORT SERIES

Series Editor: Dr. B. Shaw

Number 94
December, 1976

© 1976 University of Newcastle upon Tyne.
Printed and published by the University of Newcastle upon Tyne, Computing Laboratory, Claremont Tower, Claremont Road, Newcastle upon Tyne, NE1 7RU.
SNOW, Charles Richard,

An exercise in the transportation of an operating system. [By] C.R. Snow.

Newcastle upon Tyne: University of Newcastle upon Tyne, Computing Laboratory, 1976.

(University of Newcastle upon Tyne, Computing Laboratory, Technical Report Series, no. 94).

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE,

Suggested classmarks (primary classmark underlined)

Library of Congress:
  Dewey (17th):
    U.D.C.: 001.64404
             681.322.06

Suggested keywords

OPERATING SYSTEMS
PORTABILITY

Abstract

An attempt to transport an operating system and some associated software from one machine to another is described. Some comments are made on the suitability of the system for such a task, and some recommendations are put forward to future operating system writers to assist in making their systems more portable.

About the Author

Dr. Snow is a lecturer in the Computing Laboratory at the University of Newcastle upon Tyne.
1. **Introduction**

After some experience with the Burroughs B1700 machine and its software, it became clear that the Burroughs supplied Master Control Program (MCP) was not entirely suitable for the applications which we had in mind for the computer. This stemmed from two basic factors:

a) The use to which B1700's in general would be put as envisaged by Burroughs differed greatly from our desire to exploit the interesting aspects of this highly individual computer.

b) Manufacturers seem to have a habit of building their software so that any user, whatever his configuration and needs, is able to generate a system which provides him with the facilities he requires. As a result the software becomes larger than is strictly necessary and somewhat cumbersome. This is certainly true for the Burroughs MCP which provides more facilities than we could ever hope to use with a consequent lack of efficiency.

The main uses for the B1700 installed in this Laboratory are, firstly, for research into new machine architectures and operating systems, and secondly to provide an environment for the teaching and practical use of operating system techniques by students.

With these objectives, we may identify a third reason for being unhappy with the Burroughs operating system, namely that it would be difficult to make even small changes to the system: the sections of code affected by a change have to be identified, the appropriate amendments decided upon and lengthy and complex procedures followed in order to incorporate the modifications and recompile the system.

It was decided therefore to go for a smaller operating system which would be easily installed on our machine and would have the advantage that it could easily be altered, both to effect the installation and also to make subsequent modifications if required. In our search we had to bear in mind the features of MCP which we disliked.
and to ensure that the new system would represent an improvement. It also became clear that we would have to accept certain limitations, at least in the first instance.

The nature of the applications envisaged for the machine suggest that any operating system which would be convenient to use would require very little in the way of facility. One very definite requirement however was a filing system. Another requirement was to be able to compile and run new system programs and new interpreters. Beyond these requirements we could see no reason for any more complexity within the operating system.

Further, we could without disadvantage restrict ourselves to a single user at a time and that user having exclusive, hands-on use of the machine.

At the time of the initial decision to replace MCP, two alternatives seemed open to us:

a) Write our own operating system.

b) Borrow an operating system from elsewhere.

Option a. was rejected on the grounds that it would take rather a long time to design and build a new operating system from scratch, given that a large number of decisions (what language, what machine code, as well as the usual design decisions of operating system design) and a large amount of supporting software would be required.

There seemed to be two main candidates for 'borrowing' at the time, Brinch Hansen's SOLO system [1] and Stoy and Strachey's OS6 [2]. The SOLO system was written in Concurrent Pascal, and was on Brinch Hansen's own admission not so much an attempt to build a good operating system as an exercise in writing a reasonable sized Concurrent Pascal program. Stoy and Strachey on the other hand set out to design a 'good' operating system in an already (relatively) well established language BCPL. Richards [3] the designer of BCPL has already made
some remarks about the portability of BCPL programs, whereas at present not much is known about the portability of Concurrent Pascal programs.

These reasons alone would probably have been sufficient to persuade us to attempt to use OS6, but the relatively close proximity of the source of OS6 (Oxford, England as opposed to Pasadena, California) sway us very definitely in favour of Stoy and Strachey's system.

Thus we were faced with the problem of taking an operating system together with its associated software (in our case a compiler) and moving it from its original environment to a completely new one, and this paper documents the attempt that was made to overcome this specific problem and to demonstrate that in a limited way at least, operating system portability is achievable.

2. The Machine

This discussion of the machine to which the operating system was to be moved falls into two distinct parts - the hard machine and the soft machine. Discussion of the hard machine should clarify the meaning of the soft machine.

The Hard Machine

The hardware is a B1726 with 48K bytes of main memory which is addressable by the bit. It also has a 4K byte writeable control memory. Its peripherals consist of a card reader, line-printer, console teletype and two removable disc cartridges with a total on-line capacity of 4.6 M bytes.

The normal mode of operation for this machine is that a micro-program is written into the control store which fetches data from the main store and interprets this data as the instructions of a hypothetical machine, the so-called soft machine. Thus, by inserting a suitable microprogram into the control store, it is possible to
implement any machine code at all. Part of the problem of implementing a new language on the B1700 therefore is to define the machine code of the soft machine upon which the programs written in the language will run. It is of course possible to write programs directly in the micro-programming language, and since the machine is vertically microprogrammed this task is similar to writing programs in a rather low level assembly code. However, the normal mode of operation for this machine is to write programs in one or more high level languages and translate or compile them into a machine code which is suitable for that particular language, and use a microprogrammed interpreter to emulate that machine. In principle then it would be possible to write a program to emulate any machine at all, e.g. a PDP11, and simply run a standard PDP11 operating system on this emulated machine. In practice however this is a most unsatisfactory way of operating, and a higher level emulated or soft machine is usually more appropriate.

The Soft Machine

In his papers on BCPL, Richards [3,4] stressed the portability of BCPL programs and explains the way in which this is achieved. A hypothetical machine is suggested for which the machine code is OCODE. Portability is then achieved by writing a number of different translators for OCODE into the machine codes of the various machines on which the programs are to run. Richards claims that the construction of such a translator is a relatively trivial task.

Since the B1700 is well suited to interpretive operation it was decided that instead of translating OCODE to some other machine code, an OCDE machine interpreter would be built to interpret the OCDE (or at least a trivial transformation of it) directly. We therefore had to design and build an interpreter for an OCDE machine.

There is no way in BCPL that the code of a program may be modified, and in fact the memory space for program code is quite distinct from the memory space for data. For no good reason other than that we had no criteria by which to make the decision, the store
was divided into two equal parts each consisting of 192K bits (bearing in mind that the main store of the B1700 is bit addressable). For convenience in the microprogram it was decided to regard the data store as a set of 24 bit words, while the code store we would treat as a bit store so that the code could possibly be compacted.

Although BCPL does not distinguish between different types of data, it is relatively easy to discipline oneself when writing programs so that manipulation of code and data addresses does not get confused with operations on other types of data. It could be argued that such self-discipline should be unnecessary but that would not be within the spirit of BCPL.

BCPL provides for three kinds of data storage, the globals, the statics and the dynamic variables. The globals are stored in a vector and are accessible to any program running, simply by forming an association between a program name and an index into the global vector. The statics, which include any string constants used in the program also form a vector but these may only be accessed by name, the association with the position in the vector being defined by the compiler. The dynamic variables are stored in a conventional stack, which is also used for temporary values in expressions, parameter passing and the like. Stoy and Strachey argue that in an operating system it is often inconvenient for storage to be allocated and released according to a strict stack discipline, and so for OS6 purposes a fourth kind of data area is defined – the free store area. This is handed to the operating system at initialization time and (within limits which will be referred to later) is allocated and released entirely by calls to operating system routines. Without sufficient experience to tell us whether our decisions are right or wrong, a configuration of the main store has emerged as shown in Figure 1.
The meaning of the SACRED store and the PROG space will be explained later. BR and LR are hardware registers within the B1700 which allow the microprogrammer to prevent access, except by special operations, to the store area outside these limits. Thus it is possible to prevent accidental or malicious damage to either the code or the SACRED or PROG stores. No other memory protection is provided however and in particular there is nothing to prevent the globals, statics or stack from overflowing, and only the (assumed) correctness of the free store handling routines prevents users' free space from being corrupted.
The instruction set is mainly those OCODE instructions referred to in the paper in which OCODE is described, but it has been found necessary to include a few additional instructions. These are mainly to do with interfacing with the micro-program – e.g. to allow writing and reading main memory outside the base and limit prescribed, and to communicate with the central executive micro-program, usually to perform input/output. Each operation code is of length 6 bits and is followed by zero or more operands. Each operand consists of 1, 2, 3 or 4 six bit syllables and the number of them is indicated by a two bit length field. The length of each operand may be computed at the time the symbolic OCODE is converted (assembled) to its binary form.

The hardware control store is completely invisible to the OCODE machine, and contains the OCODE interpreter and the central input/output control program known (for historical reasons) as FROG. These programs are both written in a micro-programming language called BML [5, 6]. The FROG program requires a small amount of main memory for its own private use, and consequently 77 bits at the top of memory are set aside for this purpose as shown in Figure 1. Figure 1 also shows an area of main memory known as the sacred store, which is in the protected area and provides storage for such valuable information as the date, the time-of-day clock, the reason-for-interrupt indication (not used as yet in our implementation) the base address for the filing system and a number of others.

3. The Operating System

Stoy and Strachey describe the general concepts of OS6 in two papers [2], the first describing the overall design of the system and the second concentrating on the input/output and file handling aspects. Two other reports are also available and these contain the full text of the source code and a commentary upon it.
The system itself is designed to be a single user system, providing facilities for loading and running programs with a completely recursive run operation. Recovery from program failure was also designed to be clean and simple. On the input/output side, a unified method of accessing input/output devices through streams is presented, together with a description of a straightforward stream-oriented filing system. The whole system is written in BCPL and is highly structured, in the sense that it has many procedure calls. In addition some of the procedures which are called by the system to carry out its tasks are also available to the user programmer. Thus the interface with the operating system for the running program consists of a set of standard procedures which may be called at any time by the program. There is also a set of routines which are used by the operating system to which it would be undesirable that a user program should have access, for instance the routine which actually writes to a disc, one of whose parameters is the actual disc sector address – clearly the user should be able to access the disc only in the context of the filing system. The names of such procedures are not, in general, available to the user programmer.

Thus in general terms, the operating system allows for a single user at a time, providing him with:

i the ability to load and run programs including mechanism for taking appropriate action on failure.

ii management of the free store area.

iii a unified input/output system from and to a set of standard streams, together with the ability to create new streams out of existing ones.

iv a common filing system.

The system is written in a high level language, BCPL, and runs user programs also written in BCPL. At present, no other programming language may be used with the system, and since the soft machine is
an OCODE machine it is difficult to see how any other language could be run under the system unless a compiler were constructed to compile the language into OCODE. In concept the job control interface with the system is also to be a subset of the BCPL statements with calls to the appropriate system routines, and this may very easily be achieved. However, since the interpreter for the job control statements is not strictly part of the system, we may take the view that the job control language may be anything the user chooses, and that its interpreter is merely a user program running under the control of the operating system. Thus for the purpose of attempting to transport the operating system, this aspect may be disregarded. In fact, quite a large part of the system may be disregarded from the point of view of its portability, and in the next section we describe those parts to which we needed to pay some attention.

4. Machine Dependencies

In this section we isolate those parts of the system which needed to be modified in order to get the system running. These changes are categorised, but needless to say, the way in which the alterations were found to be necessary and made was much more ad hoc than may be suggested here.

Since we had designed our own OCODE machine and written an assembler to convert symbolic OCODE to our own peculiar form of binary machine code, we also had to design an object module format. In fact, the assembler and the load module format developed in parallel with each other. This naturally implies that the system loader would need to be rewritten to cope with the format of the assembler's output. A corresponding unloading procedure was also necessary. Some parts of the load and unload routines could be kept without modification. For instance, associated with each load is an information block containing certain data about the state of the system before the load operation so that this state could be restored when the corresponding unload was performed. All of this code could be retained and used in our version of the system.
The code space in the machine is organised as a stack, and this facilitates the making of the load and run routines into recursive operations. Since code is loaded in a first-in-last-out manner, the statics associated with the code are also loaded in a first-in-last-out way and consequently it is necessary to preserve the value of the "static stack pointer". This was an addition which was unnecessary in the way in which Stoy and Strachey handled statics. As in the original system, for each global used by a program being loaded, its previous value is saved in a position in free store set aside for the purpose and this value may be (and is) restored on unloading. The actual load module format is not dissimilar to that of Stoy and Strachey in that the module is divided into a number of sections, each one headed by a warning character and a length.

OS6 provides for a dump facility in case of failure of a program. This causes certain information in store to be written away to disc for analysis later by a dump analysis program. Clearly this dump routine will be dependent upon the layout of the main memory of the machine. Thus it was necessary for us to write our own dump procedure to match our own format of main memory. The dump analysis program has not yet been written, and this is one of the outstanding tasks to be completed.

By far the greater part of the effort involved in transferring this operating system to the B1700 was concerned with the modification of the input/output procedures. Even so, OS6 is so constructed that the parts of the input/output system concerned with actually causing peripheral transfers to take place are easily identifiable. Thus the task of replacing calls to the Executive of the Modular One machine (for which OS6 was originally written) by calls to the central I/O microprogram of the B1700 (called FROG) was reasonably straightforward. It turned out to be convenient to write a small BCPL routine which was responsible for ensuring that the control block which the microprogram FROG would receive was correctly set up, for making sure that the input/output operation has been carried out.
correctly, and, in some circumstances, for calling the micro-program a second time. It can be seen from this that in the first version of OS6 on the B1700 there is no asynchronous I/O.

There is a little more to the conversion of the OS6 input/output system, however, than just the substitution of FROG calls for EXEC calls. The Modular One input/output system is largely character oriented, whereas the B1700 is predominantly record oriented. It was therefore necessary to make appropriate adjustments to the code to allow for buffering of input/output, for instance from the card reader and console teletype, and to the line printer. Certain other differences in the hardware gave rise to other necessary changes to the code, such as for example on the console teletype. The Burroughs console is a standard teletype, but typing is only allowed to take place after a specific command to read from the teletype. The teletype is then available for typing a sequence of (less than 73) characters, after which the whole record is passed to the program. The Modular One console on the other hand appears to allow typing to take place at any time, and the program is allowed to ask if anything has been typed. Also a character typed at the console is passed immediately to the program (if the program is ready to receive it) making it possible for the program to decide its course of action on the basis of the character received.

Similar complications arise with regard to the line printer. The output to the Modular One printer is a stream of characters with special characters in the stream for controlling the paper movement, carriage returns, page throws and the like.

The line printer which is attached to our B1700 expects to receive a line of 132 characters and an operation code which contains the carriage control information. The card reader was a little more amenable to being treated as a stream: in most cases a newline character was artificially inserted at a card boundary to convert the input records to a character stream.
Thus it was necessary to modify the stream procedure of OS6 in order to make the B1700 devices behave in a stream-like fashion in addition to replacing the calls to the Executive by calls of FROG.

The FROG program itself was an attempt to simplify the software required to drive the physical I/O devices and to simplify the structure of the control block which FROG needs to carry out its task. It is a specialized program which replaces the Burroughs supplied Central Service Module (GISMO) and is specific to our particular installation. It handles the four I/O devices (console teletype, card reader, line printer and disc) and also provides for updating a clock location in main storage. It also handles interrupts either by a direct jump to a predetermined location on recognition of an interrupt, or by letting an interpreter interrogate the interrupt status itself. At present projects are under way to include in FROG the code required to handle asynchronous communication with a teletype-like device and also some OS6 specific code for passing control to and returning from any micro-program. FROG was programmed by G.M. Tomlinson and is described in [7,8].

A call to FROG is obviously required for all input and output from and to the disc. OS6 contains a routine called DISC-TRANSFER which was easily modified to make an appropriate call to FROG to cause disc transfers to take place. In the main this was all that was required to convert the filing system to operate on the B1700. There were, however, one or two small places where modifications were necessary mainly due to the fact that the page (or block) size on the Modular One disc is 256 words whereas a B1700 disc sector is only 60 words long. This could be overcome simply by the alteration of one or two constants. It was necessary however to make a substantial change to the stand-alone disc initialisation program which performs the initial setting-up of the filing system.
Stoy and Strachey make the comment in their papers that it would be a good idea from the efficiency point of view if some of the commonly used routines, for instance Next, Out and Endof, were written in what they call machine code. They do provide in their system a BCPL version of these routines, and the first version of OS6 for the B1700 used these BCPL versions. Following Stoy and Strachey's recommendation, these routines have now been written in micro-code and incorporated in the OCDE interpreter.

An internal code has been used by Stoy and Strachey's group to standardise on character coding. The code which is used by any input/output device is always converted to this internal code for manipulation by the system and subsequently converted back for output. For a variety of reasons, the two most important of which being that a certain amount of the early work on the transportation of OS6 was done on an IBM computer, and that the Burroughs devices all work in EBCDIC, it was decided to forget about the internal code of OS6 and to replace it with EBCDIC. This allows us to remove those parts of the system which were responsible for performing the translation of device codes to internal code. Of course, if a device were attached to the B1700 which did not use the EBCDIC code it would be necessary to write a set of stream procedures with which to communicate with this device, and these procedures would have to include the appropriate translation mechanism.

5. Practicalities

Although the OS6 operating system originated at the Programming Research Group at Oxford University, we obtained our copy of the text of OS6 from the University of Cambridge. This was convenient from a number of standpoints. Our discussions with the members of the Oxford group indicated that there would be a number of difficulties with regard to the transportation of an operating system (or any large program) from the paper tape oriented Modular One, whose character set made extensive use of upper and lower case letters,
to the card oriented systems at our disposal in Newcastle. It was clear that some conflicts would arise if we attempted simply to replace lower case letters by upper case letters. In addition there was no provision at Oxford for writing the system to magnetic tape in a form in which it could be easily read at Newcastle.

It was then suggested to us that we might wish to contact a group at Cambridge University who were using OS6 in connection with their Capability Machine Project. This turned out to be more convenient, since they have already resolved the problems of the character sets, and the system could easily be written to a tape in IBM/360 or 370 format. Since the BCPL compiler in use at Newcastle was developed at Cambridge University we had a great deal of confidence that the Cambridge version of OS6 could be compiled using our compiler, and this turned out to be the case. The only disadvantage of this arrangement was that the Cambridge Group had already made a number of changes to the system, and to some extent we had the task of reinstating the Oxford version.

The first version of the system therefore was compiled on the IBM/370 machine in this laboratory. The compilation output was COCODE (symbolic) and it was necessary for an assembler to be written to convert the COCODE to a binary form, which was output in the form of a binary card deck. A micro-programmed loader was then written for the B1700 which would take these cards as input, load up the store of the machine and pass control to the COCODE interpreter. In this way the OS6 system may be started up on the B1700. In fact, if the system is altered in any way, this original bootstrap is still the simplest way to test out the new system. There are provisions now for writing a new system to disc and for starting from cold by loading up the main store from the disc.

It is still the case that new programs to be run on the B1700 under OS6 need to be prepared initially on the 370 machine, since as yet there is no BCPL compiler available on the B1700. This omission we hope to rectify in the very near future.
6. Conclusions and Further Work

This final section tries to sum up the difficulties we have encountered during this exercise, and to highlight some of the good features of OS6 which helped its transportation. It is hoped that some of the points raised here may act as guidelines to future operating system writers in order to ease the portability of such systems.

The most important single feature of OS6 from the portability point of view is the fact that it is written in a high level language. Further the language used, BCPL, is particularly suitable for achieving portability because it produces code for a well-defined virtual machine. We contend that this is an essential feature for producing portable software. The high level language also aided the modification of the system where necessary. Although not an exclusive feature of high level languages, the system was written in a highly structured way. As a result of this it was relatively easy to isolate those parts of the system which needed modification, and to be confident that such modifications would not have any unfortunate side effects.

The BCPL language also has the MANIFEST feature. This is the feature whereby constant values may be symbolically named, and the compiler replaces these names by their corresponding values at compile time. This seems also to be an important feature of a language for portable programs, as it allows all the machine dependent constants to be named symbolically and the symbolic names to be used throughout the program. If these MANIFEST definitions are made together in one place, then any changes which may be necessary, for instance change in word size or number of words per disc sector, can be made once and for all and re-compilation will ensure that the change is made at every reference to the constant. A discipline is imposed on the system programmer, that he be sure to use the symbolic name and never use the explicit value in the code.
It has to be said that the Stoy and Strachey operating system was guilty of this fault in very few instances.

OS6 is a very rudimentary operating system in terms of the facilities it provides and as a result it requires very little support in the way of operating system primitives over and above the basic OPCODE instruction set. We have used only five; two instructions to allow reading and writing outside the hardware base and limit register and to revoke this privilege, two instructions to read and to write bit strings of arbitrary length (≤24 bits) and one instruction to call the PMOG program for input and output. We have also provided micro-programmed versions of some of the more commonly used operating system routines, but as was pointed out earlier, this was not necessary, as BCPL versions of these routines could be used. It is likely that operating systems providing more facilities may require more complex support from the underlying machine, but this was not so for OS6.

Looking towards the future development of OS6 as a viable operating system for our requirements on the B1700, we would need to add (at least) the following features. Some of them are already in hand.

a. Provision of a BCPL compiler on the B1700.

In order to make ourselves independent of the IBM/370 and binary decks we need to be able to compile BCPL programs on the machine and write the object code directly into the file system. The BCPL compiler is written in BCPL, but unfortunately it will not fit into the limited main memory space we have available. However work is nearing completion to produce a modified compiler which will work in phases and each phase will fit into memory. The assembler will be incorporated as an extra phase of the compilation.
b. Ability to run other interpreters than OCODE. 
Some work has been done to examine the extent to which OS6 will be affected by handing over control to some other interpreter. Clearly if any micro-program takes control of the machine then in principle there is nothing which cannot be done by that microprogram.

The work proposes a method of saving the state of OS6 before control is passed to the new micro-program, and a method of returning control to OS6 in the simplest possible fashion. Certain modifications to PROG have been made which oversee the loading into control memory of the new micro-program and the restoration of OS6 later. This work is described in [10].

c. Communication with MCP's filing system.
An OS6 stream is being developed which will allow the contents of a file created under the Burroughs MCP to be read. This will be of use during the changeover period in which files may be created by MCP (e.g. BML compilations) and subsequently used under OS6.

d. Compilation of BML programs.
A compiler is to be written for the microprogramming language BML, which will run under OS6. Although new micro-programs may be obtained from the MCP filing system using the stream mentioned in c. above, it would clearly be desirable to be able to compile BML programs while OS6 is running.

e. Communication with other machines.
It is hoped that communications may be established with some of the other machines in this laboratory. It is envisaged that a stream could be devised whereby OS6 could communicate (probably at teletype speeds) with a PDP/11 or IBM/370.

As was stated in the introduction, this experiment was conducted, firstly to establish some viable software on our B1700, and secondly
to see how easy it would be to transport an operating system. It proved to be not too difficult, and some of the problems we encountered were of our own making. It is hoped that if we were to repeat the exercise, some of the difficulties would not occur again. As more and more machines are built and more and more operating systems are written, we can foresee such transportation occurring more and more frequently.

Acknowledgements

The help of Messrs. J.E. Stoy and M.K. Harper of the Programming Research Group at Oxford University and of Mr. D. Watson of the Cambridge University Computer Laboratory in providing me with the OS6 system and other helpful information is gratefully acknowledged.

Thanks are also due to Mr. L. Marshall of this department, who helped in the design of and programmed the OCODE interpreter. Also acknowledged are the contributions of Mr. J.P. Aspden and Mr. D. McGlade in the latter stages of the project and in the continuing work.
References


"The Text of OSPub" and commentary
Technical Monograph PRG - 9(t) & PRG - 9(c)
Oxford University Computing Laboratory. July 1972

[10] Aspden J.P.
"Interpreter Swapping in an Interpreted Operating System"
M.Sc. dissertation
University of Newcastle upon Tyne. September 1976