All the users of such a subroutine package would be writing in the same language, so that the unskilled users could build on their elementary knowledge of calling subroutines to become more skilled.

The ease with which standard library subroutines and programs written for a special purpose are combined depends both on the language used and on the computer operating system. A modular language is essential if both identifier clashes, and lengthy and repetitive common-lists, are to be avoided. As the editing and combination of new with previously compiled programs depend on the operating system, efforts should be made to standardize at least on these aspects. It is arguable that these aspects of operating systems should be specified as part of the language design; too often languages are designed as though every program written will be run independently of every other program.

The surest way of getting this sort of language seems to be to band with the non-statistical users and try to influence the language designers to incorporate what is wanted.

References

Falkoff, A. D. and Iverson, K. E. (1968). APL/360 *User,s Manual.* IBM Watson Research Center, Yorktown Heights, N.Y.

Godfrey, M. D. (1971). Operating system considerations for statistical computing. *Appl. Statist.*, **20**, 45–56.

Gower, J. C. (1968). Simulating multidimensional arrays in one dimension. Algorithm AS 1. *Appl. Statist.*, **17**, 180–185.

—— (1969). Autocodes for the statistician. In *Statistical Computing* (J. A. Nelder and R. C. Milton, eds) New York: Academic Press.

Hill, I. D. (1971). Arrays with a variable number of dimensions. Algorithm AS 39. *Appl. Statist.*, **20**, 115–117.

Nelder, J. A. (1969). The description of data structures for statistical computing. In *Statistical Computing* (J. A. Nelder and R. C. Milton, eds), New York: Academic Press. pp. 13–36.

Nelder, J. A. and Cooper, B. E. (1971). Input/output in statistical programming. *Appl. Statist.*, **20**, 56–73.

Peck, J. E. L. (1970). Letter to the editor: Algol 68. *Comp. Bull.*, **14**, 64.

Wirth, N. and Hoare, C. A. R. (1966). A contribution to the development of Algol. *Comm. Ass. Comp. Mach.*, **9**, 413–432.

Operating System Considerations for Statistical Computing

By Michael D. Godfrey

Department of Mathematics, Imperial College, London

Summary

This paper presents a survey of the main components of computer operating systems, indicates some of the requirements of statistical computing and discusses how some needs of statistical computing might be better met.

A main conclusion of the paper is that, since the processing of user programs through the computing system is itself a statistical computing task, if the operating system can effectively perform this function while using only facilities which are also available to the user then the operating system may be useful for other statistical processing tasks.

1. Introduction

There are two principal characteristics of an operating system. The first is that the system must provide an environment in which programs execute in a controlled manner. The second is that the system must provide the functions that are required for the management of permanently stored information. This information will normally be stored on some mass storage devices such as drums or discs.

In order that the operating system may control the execution of programs it must provide an environment for the program which prevents any program from interacting with any other program, including the operating system programs, in any harmful or unintended way. The purpose of this control is to provide for the uninterrupted flow of programs through the system, to provide reasonable sharing of system resources, and to provide security for the information recorded in the file system. The ways in which this control is exercised by the operating system have serious effects on the user's ability to carry out statistical computations.

Partly as an intended convenience to users and partly as an aid to the operating system itself, operating systems provide a file system which permits the recording, and symbolic identification, of data. The essential characteristic of a file system is that the operating system is responsible for the symbolic identification and the security of the recorded data. In addition, operating systems provide some facilities which are intended to facilitate the accessing and manipulation of the contents of the file system. The organization and effectiveness of these facilities vary dramatically from one system to another.

It is useful to note that there is no necessary relationship between these two basic characteristics of control and file facility. Early computing systems which controlled programs had no file system, and early file systems were (usually somewhat unreliably) implemented on systems which had no effective program control mechanism.

It is somewhat surprising that neither users nor computer installations have, for the most part, benefited particularly from the intended advantages of these two basic characteristics. The amount of 'lost time' due to software malfunctions is, in many installations, about as great under current operating systems as it was in uncontrolled 'batch processing' facilities. For many users, the file system which is provided by the operating system is either too obscure, too inconvenient or too unreliable to make it usable. Therefore, one still finds many users who continue to maintain their programs and data on punched cards which are submitted for each run of the program.

However, a more basic problem is that the user of an operating system faces a software environment rather than a hardware environment. Prior to operating systems, a user could find out the characteristics of the hardware that he was using in order to aid in designing efficient programs and estimating the time or cost required to carry out a given computation. Roughly reliable information about the hardware was normally available and useful for this task. Under an operating system the hardware characteristics are mainly irrelevent since the user only "sees" the system software. There is very little information available about the operating characteristics of system software. It is sad enough that the user environment is mainly unknown in its operating characteristics, but it is also continually changing. The mean time between significant changes in hardware at most installations is appropriately measured in months. The mean time between significant software changes is more often measured in weeks, if not days. Thus, the user of an operating system faces an unknown and continually changing environment which is usually very much more restricted and inefficient than the hardware on which the operating system operates.

The software environment which the operating system presents is often termed a "virtual" system since it only appears to exist. The concept of presenting a virtual system to the user contains both opportunities for greatly expanded computing power and flexibility and opportunities for the gross errors and uncertainties of design and implementation which are present in many current operating systems.

2. Characteristics and Functions of Operating Systems

In this section we shall discuss the specific functions provided by operating systems and the user environment which is created.

2.1. *Processing Environment*

Operating systems exercise control over user program input/output functions and memory access. In most current computers these controls are exercised through the use of hardware which generates an interrupt if a user program attempts to perform an illegal function. The operating system handles all hardware interrupts and, if the interrupt was caused by what the system considers to be an illegal action by the user program then the program may be inhibited from continuing. This "hardware protection" usually is based on a "processor state word" (PSW) and a memory protection register. The setting of the PSW determines under what conditions interrupts will be generated. The memory protection register determines what portion of memory is available to the currently executing program. One possible setting of the PSW turns off the effect of the memory protection register, while another setting controls whether the hardware treats certain instructions as illegal. Thus, the operating system, by resetting the PSW, can perform operations which a user program cannot. The mechanism is used to prevent user programs from issuing I/O commands and in some cases to prevent references to memory addresses outside the program's allocated limits. It is important to note that all interrupts are handled by the system software, not just those that may imply an illegal action by the user program. Thus, for instance, a divide fault will be handled by the system software. It may or may not be possible for the user to modify the system standard action. It may even be difficult to find out what the system standard action is or if it is the same this week as it was last week. It will almost certainly cost a substantial amount in terms of instructions executed to go through the system interrupt code. Thus, it will normally not be possible to ignore such interrupts as divide fault even though the system may not even inform the user that they occurred. Fig. 1 indicates the steps taken as a result of an interrupt.

A useful list of questions to ask about the behaviour and capabilities of an operating environment is as follows:

1. Is a level of I/O available which has efficiency and power comparable to the hardware?
2. Does a generally available efficient system load routine exist which will load "load modules", segments of load modules, and language processor output?
3. Is the system command language comprehensible and can commands be equally presented to the system between tasks and from within tasks? Does the command language contain conditional logic?
4. Is there anything which a user could input into the system which cannot be permanently stored in the system and used equivalently from there?
5. Would it be convenient and efficient to write, debug, and run an operating system within the operating system?

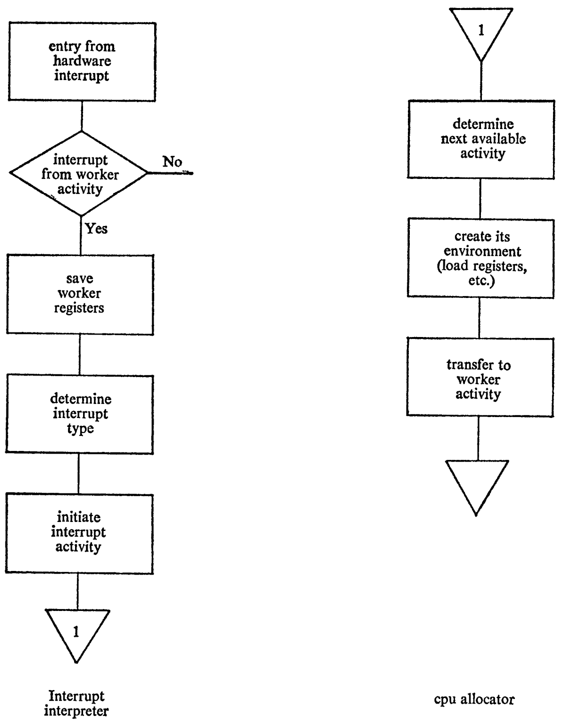
entry from hardware interrupt
interrupt from worker activity
No
Yes
save worker registers
determine interrupt type
initiate interrupt activity
1
Interrupt interpreter
1
determine next available activity
create its environment (load registers, etc.)
transfer to worker activity
cpu allocator

FIG. 1. Typical interrupt handling sequence.

2.2. *File Systems and Input/Output*

The most general form of a file structure is simply a tree such that at each node (after the first) there may exist either a directory which points to nodes at the next lower level in the graph, or the actual contents of a file. The standard notation for files in such a system is that the directories at successively lower levels are listed sequentially and separated by ".". The file contents are given as the last name in this list. Thus, *DIRECTORY1. DIRECTORY2.DATA* would refer to the contents named *DATA* which are pointed to by the directory *DIRECTORY2*, which, in turn, is pointed to by the directory *DIRECTORY1*. It is usual to provide a security system such that at each level in the hierarchy, a potential user may be required to have some identification in order to gain access to the next level. Usually several forms of access are defined. Read or write access are generally available, with "append" or "execute" being available in some systems. The required identification may also take several possible forms. It is usual to provide some tests of the information provided by the user when he logged on the system, and also tests of specific keys that the user may be required to present when he first attempts to access a given file.

There are three basic file types in terms of the way in which the contents of the file may be accessed.

1. *Sequential.* This file type simply provides a fixed address. Referencing this address creates the next sequential record on output, or obtains the next sequential record on input. Card readers, printers and communication lines are examples of sequential files.

2. *Linked.* A linked file is a file such that there is a fixed initial address which yields or creates the first record. Further records are obtained or created sequentially through an address which is retrieved from or stored in the previous record. There may be link addresses which point forward to the next record or backward to the previous record or both. A very useful property of linked files is that they may be updated by the insertion of new links anywhere in the chain simply by modification of the addresses in the logically adjacent links. Figure 2.2 indicates the structure of a linked file.

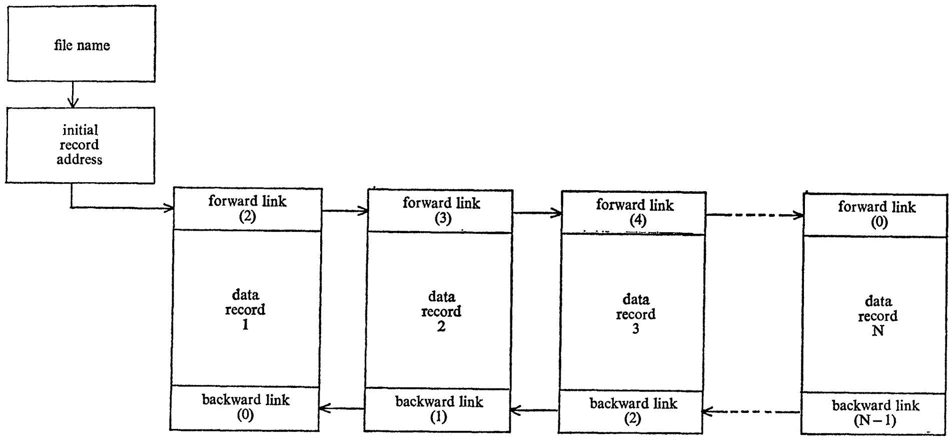

FIG. 2. Typical linked file structure.

3. *Indexed.* The addresses for an indexed file are kept in "file index tables". Thus, the *n*th record in the file may be referenced by retrieving or generating an address at the *n*th position in the file index table. Some variation in the organization of indexed files is possible as the file index tables may themselves be indexed, or they may be linked. The index addresses may reference fixed or variable length records. In addition, the indexing may be done by keys other than the integers. Fig. 3 indicates a possible indexed file structure. It seems clear that only sequential and indexed files are useful at the operating system level since a linked file structure may be based on an indexed file.

For mainly historical reasons, it is common practice to provide at least two forms of file access which, with various restrictions, may be used to access the contents of the file types indicated above. The two basic accessing methods are *buffered* and *direct.*

Buffered access should simply imply the use of system routines to manage various possible buffering schemes such that data from files are read into buffer areas before being moved into the user work area and, on output, data are transferred to buffers before being written into the file. Buffering may provide speed advantages, but it also provides flexibility in restructuring data between use in the user program and file residence. In the simplest case logical records in the user program may be grouped into longer physical records for more efficient transmission and file residence.

Direct access should imply the direct transfer of data between a file and the user memory address. Clearly the buffered access routines in the system should be written using only the direct access system commands. Thus, the user could also easily write his own buffered access system.

If the buffered access system has been implemented using the direct access method it is only really necessary to discuss the facilities of the direct access system. In general, the direct access facilities should reflect hardware facilities as required to produce efficient and flexible I/O operations. Thus, for sequential files resident on

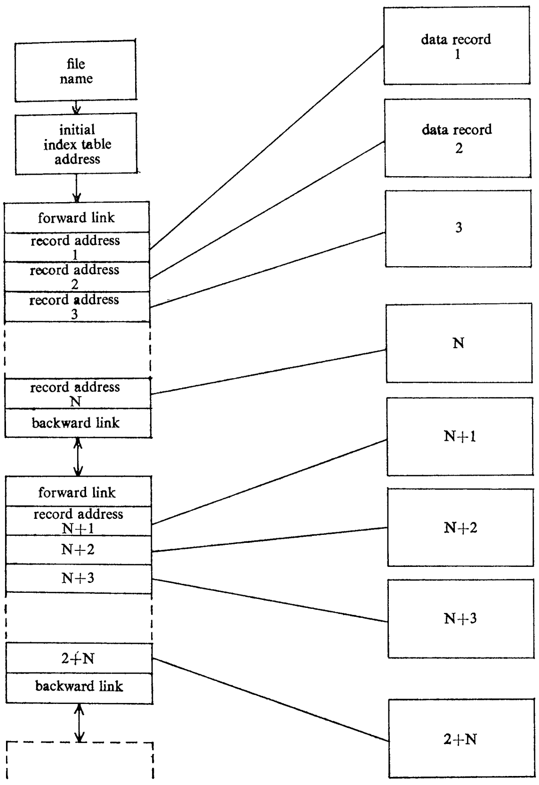

FIG. 3. Typical indexed file structure.

magnetic tape the capability to skip to an "end-of-file" mark on the tape, write an "end-of-file" mark, read backward if defined for the hardware, etc., should be available. For indexed files basic requirements are; reading and writing from a given index using records of variable extent (for instance less than or more than one index space). Exclusive read is also required to control interaction on shared files.

Both access methods require synchronization facilities. The minimal requirements are:

1. The ability to issue the I/O command and regain immediate control.
2. The ability to request an interrupt when the I/O is completed.
3. The ability to wait until I/O completion.

In addition to this basic level of file organization and access, there is a need for system/supported facilities for organizing "library-like" contents within a single file. However, this internal file structure need not, in fact should not, be a part of the basic operating system. Again, the routines for handling such files should be written using the standard system facilities so that a user could easily write his own accessing routines is he so chose.

It may be helpful to note that the OS/360 "queued access method" is basically buffered access while "basic" and "direct" are somewhat like direct. OS/360 mixes internal file structure in its basic types and accessing methods so that "partitioned data sets" are in fact indexed files with a particular internal structure.

While the above general design ideas may prove useful in creating a working file system, they are far from sufficient. More important considerations from the point of view of statistical computing are:

1. What is the efficiency of access to the contents of a file?
2. What is the maximum extent of a file?
3. How, if at all, are files which reside on magnetic tape managed?
4. What provision is there for 'foreign' files? Do foreign files have to be converted as a separate preprocessing step or can the user interject format translation code of his own?

Such questions, curiously, have received almost no consideration in the design or implementation of most file systems. There exist operating systems which require the execution of more than a thousand instructions in system code for each separate file access.

It is clear that the basic I/O facilities which the operating system provides, and the operating characteristics of these facilities, are critical to many statistical computing tasks. The view has been taken in most operating systems, though often by accident, that "hardware level" I/O and hardware error indication handling should be carried out entirely within the operating system. These facilities are not made available to the user. Instead the user is typically offered a simplified I/O command facility. If he is lucky and no errors occur he is freed from the need to understand, and provide coding to handle, the various possible error conditions. However, the price that is usually paid is unknown, often unbelievably low, efficiency and greatly restricted flexibility. It came as a mild surprise to learn that the standard magnetic-tape parity-error recovery sequence in the General Electric GECOS operating system involves rewinding the tape and then spacing over all of the previously read records before trying to read the bad record again. (This often appeared to work since the count of previously read records was often in error.) There was no known way to prevent this action.

2.3. *Outline of Operating System Components*

Table 1 presents the basic components of an operating system. The table is arranged into two categories:

1. On the left are the system activities (process controllers) which perform the various system management functions.
2. On the right are the names of the various collections of information which must be present for the operation of the system activities.

2.4. *System Data Representations*

The problems of system acceptable internal and external data representations are becoming increasingly serious for statistical computing. The problems arise with respect to what can be done using a given computing system and with respect to the transmission of information from one system to another.

There are two main problem areas. The first is the internal character codes which are acceptable to a given operating system, while the second concerns the formatting and translation associated with reading and writing external media such as tapes, disk packs and communication lines.

TABLE 1

Operating system Components

System activities	Descriptors
1. Job scheduler	1. Facilities inventory
2. Facilities allocator	1.1 Core description
3. Task allocator	1.2 Core contents description
4. Activity allocator	1.3 Mass storage description
5. I/O handler	1.4 Mass storage use
6. Task communication handler	1.5 Peripheral inventory
	1.6 Peripheral association
7. Interrupt handler	1.7 Communications inventory
8. Command language interpreter	1.8 Communications association
	2. Master file directory
9. System utilities	2.1 File name—directory association
	2.2 Directory—file contents association

A number of detailed points need to be presented in order to isolate some of the problems concerned with data representations. First, the differences between "binary" representation of data and other "external" representations need clarification. Binary representation is normally meant to imply that the bit patterns in locations in memory have been transmitted to the external medium without translation. Other representations, such as BCD, usually imply some translation, by characters, of the bit patterns in memory. However, it is also true that non-binary representations usually imply some restriction on the patterns which may be transmitted to the external medium. Unfortunately, the terms "binary" and "BCD" have also been associated with the entirely separate question of recording parity on magnetic tape. For historical reasons magnetic tapes have been written with a parity bit associated with each record. It was for a time accepted practice to write binary tapes with odd parity and other tapes with even parity. Thus, it became common practice to refer to odd parity tapes as "binary" even though the information on the tape could in fact be BCD. Finally, it is also unfortunately true that there is a restriction on the bit patterns which may appear within a record on an even parity tape which is separate from the restrictions associated with the character codes defined on a given system. Thus, binary information cannot, in general, be written on even parity tapes.

From the above comments it might seem reasonable to conclude that it would be a good idea to write all intersystem interchange tapes in binary, odd parity; but, unfortunately, standards have been developed which prevent this from being an effective means of information interchange. These standards imply, under most language systems, that when binary information is written on to a tape the actual records that go out are reformatted in such a way that they are practically unreadable on any other system. In addition, if floating point numbers are written on the tape as bit patterns, then their conversion to the internal format for floating point numbers on the target system may be a quite laborious task.

Until fairly recently the standard interchange character code was BCD. However, this has meant that one was restricted to the transmission of a subset of the 64 definable characters in the six-bit BCD character code. This standard was unsatisfactory since six bits are in any case too few, and since floating point numbers cannot be exactly represented as a sequence of decimal characters.

The problem of the width of the character code has been modified by the introduction by IBM of EBCDIC (extended BCD) and the establishment of what is (now incorrectly) called the ASCII code. EBCDIC contains 8 data bits while ASCII contains 7. Full ASCII has 96 defined graphics with the other bit patterns used for special control functions. There is also a 6 bit ASCII subset. The number of printable graphics in EBCDIC is variable, but it is still relatively rare that more than 60 are readily available. Both of these codes resolve the major weakness of six bit codes, namely the lack of sufficient width for lower case characters and additional special symbols. It is rather unfortunate that IBM has continued its policy of independence of industry standards in this area since this has delayed the establishment of ASCII as a workable standard. Currently there exist operating systems which can handle only ASCII, only EBCDIC, only some version of BCD, or even one code in one part of the system and another code in another part with no means of translation. Of course, there are systems which effectively can handle none of these.

Since nearly all non-IBM communication devices are made to use ASCII code and ASCII transmission standards, it seems clear that eventually ASCII will become the standard code for interchange of character-type information. There is no current hope that the problem of interchange of floating point data will be resolved.

3. Requirements for Statistical Computing

Having briefly discussed some of the characteristics of operating systems in the previous section, we will now discuss some of the characteristics and facilities that would be useful for statistical computing.

3.1. *Organization of Data*

Providing useful facilities for the management of data is made difficult by the diversity of data structures and the extents of data which must be considered. An additional difficulty is introduced by the fact that this problem has implications at many levels of operating system design and implementation. The following list of facilities is intended as indicative of the requirements of statistical computing. All the items in the list can be implemented in an efficient operating system.

1. It is necessary to have a file and record format description system which allows a compact description of expected input and a method of formatting output. For both input and output the facility must exist to permit transfers to executable code which depend on either I/O activation or data record content. The code which may be executed must have sufficient scope to be able to alter dynamically the format information and communicate with the main user code. A special case of such code is the code which the user might provide to handle such conversion errors as illegal or non-corresponding characters. Clearly, such code must be able either to change the data items so that they convert, or change the format so that the original data convert. A reasonable way of organizing this facility is to make it entirely separate from the actual I/O transmission so that these two different functions are not unnecessarily confused

with each other. (The paper by Nelder and Cooper (1971) discusses this problem in more detail.)

2. It must be possible to associate file and record format information with file names in such a way that this information need only be defined once when the file is first created. After that, symbolic reference to file or record names should be sufficient to cause the system to access the data using the correct method and format. This implies that data can be made self-describing.

3. There must be a uniform system interface such that all languages, including the one in which the system is written, have access to the same data-handling facilities. If these facilities effectively serve the needs of the operating system they may prove usable to users as well.

3.2. *Organization of Processes*

It is now being realized that the standard model of a sequential computing machine is not adequate either as a representation or as a working example of a useful computing system. Modern computing systems are made up of a multiplicity of interconnected components which demand multiplexing across the many activities which require their use. In order to control effectively the allocation of both hardware and software resources, the operating system requires the ability to manage the various control activities of which it is composed. These system control activities must operate in parallel, having varying degrees and forms of interconnectedness. One of the basic reasons that the development of modern operating systems has been as painful as it has been is that the tools that have usually been provided for the design and implementation of the system have been based on conventional notions of sequential processing. A more natural set of tools, both for the operating system and for the user, is the set of tools necessary for the creation, manipulation, activation and synchronization of multiple activities. The following is a minimal set of such tools:

1. It must be possible for a large number of activities to co-exist and co-operate under one program unit.
2. Program initiation and activity activation must be functionally separated, as must program termination and activity de-activation.
3. There must be some mechanism for the definition and control of activity interaction. It must be possible to set locks and indicate the unsetting of locks.
4. There must be a uniform process interface such that all languages, including the system language, have access to the same facilities.
5. Finally, and most important, there must be a powerful diagnostic system available to both users and the operating system. This system requires both "snapshot" and "tracing" capabilities. The "snapshot" facility must provide easily controlled dumps of specified information at a point in time and also be able to present the changes in state between two points in time. The "tracing" facility must be able to trace logical flow *in parallel paths* between specified program points.

4. An Outline of a Feasible Operating Environment for Statistical Computing

This section very briefly sketches the features of an operating environment which would make statistical computing substantially more convenient and interesting. Some indications of the implementation requirements of these facilities will also be given. All of the facilities can be implemented relatively easily and effectively.

4.1. *Processing Facilities*

The basic code unit for the operating environment should be the *activity*. An activity may be defined by an address and, optionally, a name. The address provides the operating system with a starting transfer point when the activity is initially activated. The name provides a way of referring to the activity symbolically. It should be possible to register an activity with the system, request that an activity be activated, terminate an activity, and set locks on specified code or data references. This last requirement provides the essential aspect of the ability to control activity interaction. Various other interaction controls provide convenient ways of doing specific things, but the memory lock is all that is essential. (From the logical design standpoint, it is interesting to note that in a multi-processor environment where several activities may be simultaneously in execution the lock facility requires a special-purpose hardware instruction.). A *program* is a collection of activities. A program is in an executable state if there exists at least one activity in the program which has been activated and not de-activated. Program termination may be defined to occur when there are no more activities in the active state.

Activities should be able to interact with the environment in any way except that they cannot execute a request that would cause termination of their program unit if they expect to regain control after the request. It is particularly important that an activity should be able to request the loading of executable code at either a fixed or a dynamically variable address (in systems where this distinction is meaningful). Thus, in particular, an activity should be able to load several copies of a piece of code into different parts of memory and register them as several new activities. This is the converse of re-entrant code and is surprisingly useful.

4.2. *File and I/O Facilities*

It appears to be necessary for an operating system to provide two levels of file system support. At the first level the operating system should provide only the essential functions of symbolic name association and relative-to-device address translation. At this level a "file" could be a communication line, experimental equipment or any other peripheral device. The user would have complete control over, and responsibility for, file format, acceptable accessing modes and acceptable file commands. At the second level the system should provide a standard file interface for conventional use of the file system for program and data management. This level of support should be designed to facilitate the use of language processors. In particular, it should be as convenient as possible to enter and edit any symbolic text, and direct the various operations associated with text processing. In order to accomplish these functions there must exist a convenient way of creating libraries which will be recognized and effectively handled by the language processors, particularly the link-editor. It should definitely not be necessary to name explicitly the items which a processor, such as the link-editor, is intended to use or look for.

In order to accomplish an adequately flexible I/O facility it is reasonable to require that all hardware instructions can be presented to the operating system for execution. The system must, of course, establish that the operations are legal for the task that presented them. If they are legal they should be executed exactly as they were presented, and the results should be returned to the task as if the commands had been executed directly. This facility provides for the requirement of doing I/O outside of the file system as such. A standard set of I/O functions for convenient access to the file system is also required. This set of functions must be exactly the same set used by the operating system itself for file access. Finally, the user must have available the facilities that the language processors use in accessing files.

5. Conclusions

I have tried to outline a few of the problem areas of operating system interaction with statistical computing. However, many problems went unmentioned and many pitfalls exist for which I have not given adequate warnings. There is currently no substitute for practical tests. However, if the operating system operates efficiently using only those facilities which it provides to the user then there is some chance that user programs may be able to operate effectively. If the system is inefficient or if it has to rely on special facilities which are not available to the user, then there must be considerable doubt about its usefulness.

The simplest question one can ask to establish the possible usefulness of an operating system is: Would you be comfortable about the prospect of writing a powerful and flexible operating system using the operating system? If the answer to this question is "no" then the chances are good that attempts to use the system for serious statistical computing tasks will encounter repeated frustrations and restrictions.

Reference

Nelder, J. A. and Cooper, B. E. (1971). Input/output in statistical programming. *Appl. Statist.*, **20**, 56–73.

Input/output in Statistical Programming

By J. A. Nelder and B. E. Cooper†

Rothamsted Experimental Station *Atlas Computer Laboratory*

Summary

Input/output (I/O) is analysed in terms of the transfer of items of various types having various internal and external representations between various internal and external positions. Ways of describing representation and position are discussed, for both character and binary data. Three basic properties of good I/O facilities for statistical computing are stated, and the facilities offered by four existing programming languages are matched against them. Finally, extensions to Fortran are proposed that would provide the properties sought.

Introduction

We suspect that input/output (I/O) operations cause more trouble than any others to computer users. Making the internal algorithms (not concerned with I/O operations) work is often much easier than reading in the data they require, or printing the results. No doubt the I/O operations are inherently messier than the internal operations, but much of the difficulty arises from deficiencies in existing high-level programming languages. I/O operations may be considered as part of a wider class of operations (called "transput" in the Algol 68 Report) concerned with the transfer of information between parts of a computer and its peripherals. We shall be concerned with the pure I/O operations of reading information from an input medium into core and of outputting information to an output peripheral such as a line printer or card punch. Much

† Now at ICL Dataskil Ltd, Reading.