OS6
AN OPERATING SYSTEM FOR A SMALL COMPUTER

by

Joseph Stoy

and

Christopher Strachey

Oxford University Computing Laboratory

Programming Research Group

OS6

An Operating System For a Small Computer

by

Joseph Stoy

and

Christopher Strachey

Oxford University

Technical Monograph PRG-8

May 1972

Oxford University Computing Laboratory,
Programming Research Group,
45, Banbury Road,
Oxford.

Oxford University Computing Laboratory,
Programming Research Group,
45 Banbury Road,
Oxford. OX2 6PE

The contents of this booklet are also to appear as two papers in the *Computer Journal* (Vol.15), and are published as a Technical Monograph by special arrangement with the Editors.

References in the literature should be made to *The Computer Journal*, as the texts are identical and the *Journal* is generally available in libraries.

ABSTRACT

Part I is a general description of a simple operating system, which runs in a virtual machine (implemented on a real machine by an interpreter). OS6 copes with only one user at a time, and is not a multiprogramming system: many major problems associated with large operating systems have therefore been avoided or considerably simplified. It nevertheless has several features of interest, including the fact that it is written almost entirely in the high-level language BCPL. The most important single feature, however, is the hierarchical nature of its control structure, which avoids the need for a special job-control language.

Part II covers the facilities for input/output, and the handling of files on the disc. The input/output system uses a very general form of stream; the filing system is designed to have a clear and logical structure.

CONTENTS

PART I

PART II

OS6 - AN OPERATING SYSTEM FOR A SMALL COMPUTER

PART I - GENERAL PRINCIPLES AND STRUCTURE

0. PRELIMINARIES

0.0. *Introduction*.

Although there is already considerable understanding of the theory of programming languages, it seems that operating systems are still designed *ad hoc*. In both cases, however, the object is the same: to direct the activity of a computing machine. Indeed, many operating systems are driven by "job control languages" (see, for example, Brown[4]), which are nothing more than primitive but specialised programming languages. The fundamental concepts of programming languages are now fairly well understood, but there is a need for theoretical research to bring similar conceptual understanding to the design of operating systems. The need for such a coherent overall design and underlying philosophy is being increasingly emphasized, for example by Cox[6]. Purely theoretical work in this field, if divorced from practical work, easily becomes sterile and unrealistic: we need to put our ideas to the test by using them as the basis of a real operating system. This paper and its companion, Part 2 [15], are a description of such an experimental system.

The object of the experiment is to demonstrate that an operating system designed to be coherent and elegant can nevertheless control a real machine with a group of real users. Needham and Hartley [10] rightly state that in this kind of work one needs both theoretical insight and horse sense. They suggest that with a formalistic approach - which they illustrate with a thumbnail sketch of an "ideally simplified computer system" - "there is a very real danger that a lot of small and awkward corners must be knocked off the problem with a view to achieving simplicity but, unfortunately, at the expense of facility." Since we believe that the system they are talking about is ours, we hope to show that we cope with our real environment, admittedly simplified for the sake of experiment, in a realistic way.

A computer dedicated to the provision of a computing service is an unsuitable vehicle for this kind of experiment. If the experimental system is used for the service work the need for constant reliability and adequate documentation effectively prevents changes to the basic design of the system, but at the same time much effort is devoted to removing small bugs as soon as they are discovered. On the other hand, if the experimental system is not the machine's normal regime, much inconvenience and disruption of service is caused by the frequent changes of system. For these reasons a computer devoted exclusively to software research is most desirable, and, with the aid of a grant from the Science Research Council, the Programming Research Group has recently acquired a Modular One machine for this purpose.

Modular One has a 16-bit word, and a cycle time of 750nS. Our complete configuration includes 32K of core, fast paper tape input/output, a 1M word disc, a multiplexer for several consoles, a line printer and a clock, but the machine was originally delivered without the last four items. Our initial effort was therefore restricted to a single user system with no permanent file store. Such a limitation was, in fact, not so great a disadvantage as it might at first sight appear. The design of a single user system presents several interesting problems which should be solved before the extra complexities of multi-programming and concurrent processing are faced. The function of a multi-user system is to provide each user with all the facilities of a large computer, and it follows that the first stage in the design of such a system should be (but rarely is) the design of a satisfactory single-user system. The problems associated with the organisation of file storage, being logically separate from the rest of the system, were also conveniently postponed for a time; but the filing system has since been added, and is described in Part 2.

0.1. *Choice of Language*

As a deliberate act of policy it was decided to write the entire system in one high-level language, as it seems to us that the current practice of writing software in an assembly language is one of the main sources of the "software problem". There are three main areas in which the use of a high-level language can relieve a programmer of tedious organisational details:

(i) the control of the path of execution,
(ii) storage allocation,
(iii) the representation of information.

For software work, and particularly in the programming of operating systems, elaborate provisions and conventions about storage allocation and data representation can be an embarrassment. It is usually the job of the operating system, for example, to provide a suitable storage allocation algorithm, and it is inappropriate that decisions about it should be prejudiced by the design of the language used. Moreover, the requirement that an operating system should deal with storage allocation implies that the language should allow addresses to be treated as data objects, and for calculations to be performed on them. In assembly code this is a matter of course, but few high level languages are equipped for it. General pointers and addresses, if treated at all, are either so limited that one may not even assign a new value to them (cf. arrays in Algol 60) or are strongly associated with the particular kinds of data structure available in the language (cf. LISP). Operating systems are concerned with the general problem of the allocation of the actual hardware resources, and are one of the few applications where the linear mode of addressing employed in most computer memories must be accessible in the language. Software writers also require great flexibility in their choice of data representation. It is probably the existence of too many constraints in these two areas that turns software writers away from high level languages back to their more permissive machine codes. On the other hand, there is general agreement about what facilities are desirable to control the path of execution – such things as conditional commands and expressions, functions and subroutines, cycle commands and recursion.

Although in most existing languages all three areas are treated with comparable sophistication, it is by no means essential to do so. The resources of the ideal software language should, in our opinion, be concentrated around the control facilities, and matters concerning storage and representation left very much to the programmer.

BCPL, the language used in OS6, is just such a language. It was invented by Martin Richards [11] [12], and is superficially very like CPL [2], from which it gets its name, with the same richness as CPL in the syntax of commands and expressions. There is only one type, the bit pattern: that is to say, the language deals directly with the representation of objects rather than with the abstract objects themselves. This property makes the language unsuitable for the general programmer, as it does not prevent his performing meaningless operations (such as multiplying together two labels); on the other hand, it provides the extra flexibility system programmers require. BCPL has no automatic storage allocation – except for a rudimentary stack for local variables and local vectors – but one of the operators in the language allows a bit pattern to be treated as an address, and the converse operation is also available: this provides the mechanism for control of storage and the manipulation of data structures.

0.2. *The Virtual Machine*

Much of the difficulty in writing software concerns the need to cope with inadequate hardware. Many complaints about the inefficiency of high-level languages arise because most modern machines have an instruction set which is grotesquely unsuitable for implementing them. Nowadays, the majority of computer instructions are generated by compilers, and it might reasonably be supposed that the standard compilers for a new range of machines would be designed before the order code was fixed. The design of the Burroughs B6500 machine [5] is an example of what this approach might produce.

In an experimental environment an early commitment to a particular design of machine code is even more unfortunate. Besides, the instruction set of Modular One is ingenious and elaborate; it is difficult even for an experienced programmer to decide the best way of coding any particular operation. The instruction set is not at all convenient for the implementation of BCPL and a preliminary investigation indicated that the machine code generated by a simple-minded code generator would be very bulky. The solution to these problems was to design an interpreted code, known as IC, to fit the needs of the BCPL system. The IC interpreter was written in Modular One machine code, and takes about 250 instructions.

The interpreter for IC thus behaves as a virtual machine whose instruction set is IC (a description of the structure of this machine and of IC will be deferred to a later paper). The practical effect of using this technique has been to produce very compact programs with a very simple code generator, but to slow down the execution time by a factor of about fifteen. The actual hardware of Modular One is entirely concealed behind this virtual machine; apart from tracking down hardware faults none of our users has required to know anything of the actual hardware structure of Modular One, nor have they been conscious of the extent to which the interpreter slows down the execution.* It is worth pointing out that the virtual machine is of similar complexity to the Modular One, apart from the control of peripheral transfers (see below §2.1) which is at present rather more automatic in the virtual machine. It would therefore be possible to construct a CPU whose hardware worked directly on IC; this, of course, would run about fifteen times faster than the present system.

From now on, when describing OS6, we shall use the terms "machine", "hardware" and "machine code" to refer to the virtual machine and to IC, unless the context specifically implies otherwise.

0.3. *Requirements of an Operating System*

The functions of an operating system may be separated into several classes:

1. Control and protection of users

(1.1) There must be provision for *loading* a program and initiating its execution.

(1.2) The system must have the *ability to recover after failure* of a user's program.

(1.3) The user's program must be given, as far as possible, *protection from outside interference*, including hardware or system failure, and the operation of other programs.

---

* We have strictly enforced the rule that no programs are to be written in Modular One machine code. (In fact all our user programs are written in BCPL.) This has had the dual benefit of insulating our users, who are chiefly research students in computing, from the irrelevancies of the machine design and also of protecting us from a clamour to make full use of the raw speed of the machine.

2. Hardware Management

(2.1) There must be routines for performing those *special operations* on the hardware which cannot be expressed in the user's programming language - for example, operations concerned with the manipulation of the peripherals. We include under this heading the mechanism for loading the operating system itself.

(2.2) The operating system must control the *allocation of resources* - e.g. core store, time, peripherals.

3. Other facilities

Many systems also provide a suite of utility routines, particularly in the realm of input/output, which are in the operating system not from theoretical necessity, but merely to save the user the trouble of programming them himself.

OS6 fulfils all the functions mentioned above, but because of the simple nature of the environment, some of the problems are considerably simplified. We defer to Part 2 all discussion of the input/output scheme and filing system - which comprises part of heading 2 and most of heading 3 - and now proceed to discuss the remaining sections in turn.

1. CONTROL AND PROTECTION

1.0. *General ideas*

The job of getting a program running under an operating system may consist of various operations involving a number of choices. For example, a text may be edited, and compiled by one of a number of compilers; the program may then be loaded, together with previously compiled library routines; the required input/output facilities and post-mortem arrangements may be set up, and the program finally executed. The traditional method of controlling this process is by means of a special-purpose "job-control language". Many of these languages are insufficiently powerful, so that complex activities must be split into several jobs, coupled together by means of complicated verbal instructions to the operators. All of them are inelegant, and their description is usually so complicated that there is a large risk of error when attempting an uncommon sequence of operations.

A job control language may be thought of as a simple programming language, and the various editors, compilers and the rest as utility routines available to the programmer in that language. The system then appears to be a simple "load and go" loop: the program in the job description language is read, and possibly compiled and executed, though more probably it is interpreted directly. This being so, it seems reasonable to use the same high-level programming language as is used throughout the system, a language designed with an eye to clarity of expression and ease of correct programming. This is the solution adopted in the present system, except that since the compiler is a multi-pass program with overlays and compilation is consequently a lengthy process, the steering program is at present precompiled, and the load-go loop reads and obeys a binary program.

It is foreign to the spirit of high-level programming to restrict the calling of any routine to the outer level of the program: indeed, one of the results of programming in a high-level language is that one is never quite sure when one is actually writing an "outer level" program. In OS6 we make no such restriction: any program may load, obey, edit or compile any other program, to any depth, subject only to the availability of the core store. It will be noted that there now appears to be little difference between obeying a program and applying a subroutine, except for the trivial point that the former usually has no parameters.

In a perfect world it would be possible for routines to be the largest unit of instruction code. But in practice failures occur, and when they do it is necessary to know how much of the hierarchy of activations to abort. A routine activation is too small a unit for this purpose: firstly because many failures within routines are in fact the result of errors in surrounding program, and secondly because the cost of taking adequate precautions, at every routine call, for recovery after possible error, would be prohibitive. The style and intelligibility of a program is usually improved by making extensive use of subroutines, and it is therefore important to make routine application as cheap as possible.

We may now, at last, give some meaning to the term "program": we use it to denote that part of the nested set of subroutine activations to be aborted if an algorithm fails. The word is often used to denote a portion of code loaded all at one time, which might consist of several concatenated segments, each of which has been separately compiled. Although in many operating systems these are identified, they are two independent concepts, and in OS6 are completely separated.

1.1. *The Loading of Programs*

The store of the OS6 machine is divided by hardware into two areas, of which one is reserved for program code. A compiled segment of BCPL text is also in two main parts, a section of code and one containing pre-initialised variables (e.g. labels and strings); both parts are relocatable. It has proved satisfactory to manage the code area as a stack, so that code is strictly last-in-first-out. The space for the block of variables is claimed from the free store system (described below). After these blocks, there comes a block of code called an *Interlude*. Interludes are loaded into the code area and immediately obeyed; they are subsequently overwritten by the next code loaded. The interlude is a general feature of the system, and is available for any purpose; the particular example occurring in compiled BCPL programs causes the code to be relocated, using information which follows the interlude in the compiled program.

BCPL has a global vector, which corresponds quite closely to Fortran's COMMON storage: it is the only means by which separately compiled segments of program can communicate. When a segment containing global functions, routines or labels is loaded, it is the final job of the loader to initialise the appropriate elements in the global vector.

Any program, of course, may call the loader routine in order to load more code: a program may therefore itself load from the file system the library routines it requires. Code may also be explicitly unloaded. The parameter for the *Unload* routine specifies how much of the code in the code area is to be retained. The variable blocks corresponding to the abandoned code are returned to free storage.

The OS6 machine has only one global vector, which is of limited size. There is therefore some danger that the initialisation of the globals by the loader will overwrite those of some other segment loaded further up the hierarchy. On the other hand, elements of the global vector are also used for storing variables, especially variables used by more than one section of program, and the ability to overwrite these variables is, of course, essential. Ideally we would wish to allow intentional overwriting, but to protect the user from the effects of doing so unintentionally. The compromise adopted in OS6 is that the function, routine or label loader, when initialising a global, should preserve the previous contents, which in turn would be restored hy the *Unload* routine. Each global element is thus effectively a push-down store, "pushed" when a new global routine, function or label is defined for that element, and "popped" when the defining code is unloaded.

Although this compromise is better than nothing, it is not entirely satisfactory. The difficulties are an example of those which often occur when it is attempted to impose hierarchical behaviour on a non-hierarchical structure (in this case a linear vector). Global functions which have been overwritten by subsequent definitions are (if only temporarily) completely inaccessible, and not merely hidden. One would prefer something more like the Algol scope rules, and a more sophisticated method of segmentation: but BCPL is more concerned with ease of implementation.

All this is an example of the general tendency for the power of the hierarchical expression in a language to be reduced by the lack of corresponding power in the mechanism for storage of variables.

The ability to trap the application of an undefined global, whether or not it has been previously defined, has been used in a debugging facility, which is discussed further below (§3.1.).

1.2. *Recovery after failure*

In OS6, the decision as to how much of the activation hierarchy to abort after failure is left to the programmer. There is a routine called *Run*, which takes a program (that is, a parameterless routine) as its parameter:

Run[*Prog*] .

Such a call leads to the application of the parameter in the usual way:

Prog[] .

The execution of *Prog*[], or of any routine, may terminate in one of three ways. The routine itself may decide that it has finished its job, it may decide that it has failed to such an extent that there is no point in continuing, or it may be forcibly interrupted (because it tries to refer outside storage bounds, for example, or because the operator switches off the computer). In each case, however *Prog* terminates, the system resumes execution of the routine containing the call of *Run*, after restoring the machine, in some respects, to the state it had before the call of *Run*.

An example illustrating the use of *Run* is given by the following routine, which is a very simple steering program for a compiler.

```
let SteeringProgram[] be
  §SP §  Failure, Exit := true, false
                                   | these are two global variables
        Run[TextInput]
        unless Failure do Run[Compiler]    $
                                 repeatuntil Exit        $SP
```

This routine invokes two further subroutines, the *TextInput* routine and the *Compiler* itself. The loop continues until one of these routines explicitly sets the global variable *Exit*. If, for example, the machine is switched off while reading text, when switched on again it will continue the loop, omitting the *Run* of *Compiler* (as *Failure* will still be true), and will recommence *TextInput*.

Run is fully recursive: any routine may *Run* another. In the example above, the steering program itself might well be invoked by a call

Run[*SteeringProgram*].

1.2.1. *Implementation*

The routine *Run* creates a vector called the Run-block, for which the space is claimed from the storage allocator. In this vector are stored the values of some of the more important system variables, which include those concerned with storage allocation in both segments, those which conventionally hold the normal input/output streams (see Part 2), and those concerned with post mortem arrangements. Next a routine is called to provide the program being Run with a fresh free storage area (this is discussed further below in §2.2). The program is then applied.

Two methods are provided for terminating the Running of a routine after successful execution. The first, the simpler in conception and the commoner in practice, is invoked by the usual methods in BCPL for returning from a subroutine call. This implies that the routine need not know whether it had been simply applied, or had been executed under control of *Run*; in other words, as far as the writing is concerned, a program and a routine are identical. Sometimes, however, a job becomes completed when the algorithm is deep in subroutine calls. In this case another routine, *Finish*, is available, which (by explicitly manipulating the stack pointer) achieves the same effect. It should be noticed that this second method is invoked by the call of a system routine, not by a special command in the language: that is to say, we write

Finish[]

rather than

finish .

This is quite deliberate: the meaning of the concept of finishing a program depends entirely on the particular operating system, and its *ad hoc* nature is quite out of place in the semantics of a language with an hierarchical structure.

For a program which decides it has catastrophically failed, the proper course is to call the routine *GiveUp*. This routine (which conventionally takes one integer parameter, used for passing diagnostic information) is a variable routine and may be freely altered by the programmer. A default value, which prints some standard diagnostic information, is set by the system. *GiveUp* is one of the variables preserved in the Run-block, so that, as will be seen below, any change made by a program is applicable only until the end of the Run.

When a program is forcibly interrupted, the operator is given a choice. Either he may force a call of *Finish*, which quietly terminates one level of *Run*, or if diagnostic information is required, he may force a call of *GiveUp*.

When the Running of a program has terminated, *Run* replaces the old values of the variables and uses them to restore the machine, in some respects, to the state it had before the call of *Run*. In particular, any code loaded during the Run is unloaded, and any storage claimed during the Run is forcibly returned. Finally the space occupied by the Run-block is also returned.

1.2.2. *Clearing up*

It sometimes happens that a system routine initiates some activity which, to avoid endangering the system's security, requires subsequent completion, even if the program fails. Such activities include output to a file on the disc (which probably requires final housekeeping action), and transfers from peripherals to core store in the free store area (which must be cancelled if incomplete before the storage is reallocated). Whenever such an activity is initiated, the concluding action required is entered in a chain called the *ClearUpChain*. If the activity is completed properly under control of the program, the entry is removed from the chain; otherwise it is obeyed after termination of the program,when the remaining entries in the chain are called in turn by *Run*.

The *ClearUpChain* is also available for activities which require completion, not for the sake of system security, but to avoid inconvenience to the user (for example, to avoid abandoning part of his output in various buffers). However, precautions must be taken to ensure that, even if some clearing-up activity fails, the other entries in the chain are duly obeyed.

When an activity is forcibly completed by means of the *ClearUpChain*, it may itself close down some further activity in the normal way: this would cause embarrassment if the further activity had already been forcibly completed. Care must be taken to avoid this situation. The order of the forcible completions is plainly important, and in practice the difficulties are usually avoided by processing the *ClearUpChain* in a "last in first out" fashion.

1.2.3. *The Load-Go Loop*

We are now in a position to describe the Load-Go Loop which is the heart of the operating system. The following is a slightly simplified version:

```
let LoadGoLoop[] be
    §   Run[LGLoop] repeat §
and LGLoop[] be
    § Load[]
      Run[Prog]  §
```

Note that it is necessary to Run *LGLoop* to allow for the possibility of failure in the loading phase. No specific mention of unloading is required, as it is done automatically by the Run in *LoadGoLoop*. *

1.3. *Protection from interference*

This facet of an operating system's activity is particularly simplified in OS6. In the first place we cater for only one user running only one program at a time, we therefore merely need to protect the integrity of the permanent information in the system. This permanent information is principally in the filing system, which is discussed in Part 2, but it also includes the operating system itself.

* To allow for the possibility of recovery if, say, the machine is switched off between the Runs of *LGLoop* (i.e. while obeying the repeat) it is necessary to Run *LoadGoLoop*; however, to avoid an infinite regression we achieve the same effect by forcibly "tying a loop" at the end of the chain of Run blocks, when the system is initiated.

Most operating systems protect themselves by hardware, but in OS6 there are no privileged routines, and any part of the system may be overwritten. This fact makes it impossible for the system to be wholly immune from corruption by an aberrant program, and we can only hope to cope with the more common forms of error. But since no other users are involved it is always possible to reload the system, if, for example, a program accidentally clears half the store, and there are compensations. The chief advantage is the great flexibility offered by such an open system to programmers who are concerned (as are most of the system's users) with the development of system programs, or other large suites of programs. For example, a programmer developing a new disc housekeeping scheme may replace the basic routines which transfer information to and from the disc by an alternative set which manipulates a simulated disc in the core, instead of the disc itself. All the standard file system routines are still available acting on the simulated disc, and the real disc is not put at risk until the new routines have been debugged.* (It is a simple application of the *Run* apparatus to ensure that the normal primitive routines are replaced at the conclusion of the test).

In the second place we have avoided altogether the problem of dealing with malice on the part of the users. Such protection would, indeed, be impossible in a system as open as ours, but in any case we agree with the experience of the Leeds team [16] that it is unnecessary in our sort of environment (at least when the operating system is not required to ration severely limited computer time).

* It must be admitted, however, that our programmers tend to discover the value of such a careful approach by bitter experience in the application of untested routines to the permanent information

The system does, however, embody checks against the more common forms of accidental error. In most cases this activity is quite *ad hoc*: if, for example, a program does a wild jump, the most likely address where it will land is zero, and so we arrange that word zero of the program area is an error trap. Other cases are merely good practice, such as checking the validity or consistency of information supplied to system routines, or coming in from the peripherals. Only a few are a matter of system design, such as the division of the store into two segments. The space reserved for program is accessible, other than for execution, only by two special instructions, which occur only in two special machine code routines. Thus the code in the program segment is most unlikely to be altered accidentally - and in fact such an accident has never, to our knowledge, occurred. We take advantage of the security of this segment by keeping very important variables there.

It is worth pointing out that one of the chief causes of wild errors has been removed by banning assembly code from the system. The exclusive use of a high-level language implies that although "silly mistakes" are still made, most of them are detected by the compiler before they can do harm. The other kind of typical assembly code mistake, caused by conflicting use of storage, is minimised by the "modular" discipline imposed by a block-structured high-level language.

2. HARDWARE MANAGEMENT

2.1. *Special Operations*

OS6 contains three essential routines for providing special operations. They are written in machine code, as each contains a special instruction which is never generated by the BCPL compiler. Two of these special routines serve to read and write into the area of core reserved for code. The other provides a means of communication with EXEC, the Executive program written by Computer Technology Limited, and it is used almost exclusively for initiating peripheral transfers. EXEC then autonomously services the peripheral interrupts, filling or outputting a special buffer. The relegation of this job to the "hardware" is of course evading a difficult problem. It is not, however, an unrealistic simplification: many computers have hardware for this purpose.

There are a few other machine code routines, which embody other special instructions invented to speed up certain input/output operations. These are described in Part 2.

The essential routines occupy seven words, and the others a further ten. This is the only machine code in the operating system itself.

2.1.1. *Bootstraps*

Some mechanism must be provided with an operating system for establishing it in an empty machine. This special mechanism is called a *bootstrap.*

A bootstrap consists of a series of load-go loops*, each of which is used to load and initiate the next one, until the final one which is the load-go loop of the operating system

* The initial and final members at least of the series must be full load-go loops, since both the empty machine and the operating system are, in some sense, permanent. The intermediate members may well be evanescent, in the sense that they are obeyed only once before their code is overwritten: in such cases a closed loop is unnecessary and they may be simply load-go sequences.

itself.

The provision of the initial load-go loop varies from machine to machine. In some it is built completely into the hardware (and called, for example, "initial orders"), in some the "load" and "go" phases have to be separately started by the operator, while in others the loading phase has to be carried out explicitly using the handkeys.

The details of a bootstrap depend greatly on the hardware, as well as on the requirements of the operating system which it loads. They are invariably dirty, and further discussion is therefore inappropriate here.

2.2. *Allocation of Resources*

We turn now to the question of allocation of resources. This again is an area considerably simplified in OS6 by the simple nature of the environment, and in particular by the fact that we are catering for only one user running only one program at a time. This, together with the fact that there is only one example of any particular type of peripheral, allows us to eliminate control of peripheral allocation completely. So far, operating is always "hands on", and we have therefore not yet felt the need to have the machine control the time taken by a program. As the load-go loop becomes more sophisticated, so that it processes jobs previously left in a queue, some form of time control will become necessary: this will be implemented by specifying a time limit as a second parameter to the *Run* routine described above.. At present, however, the control of resources therefore reduces to the problem of storage allocation.

2.2.1. *Storage Allocation.*

We remarked earlier (§0.1) that BCPL has a rudimentary stack for local variables and local vectors. Space for this stack is allocated when the system is initialised (it is given about 1000 words) and its bounds are then permanently fixed. However, BCPL programs frequently require off-stack storage: typical occasions might be when the result of a function is a vector, or when a vector of working space is required to survive until the next activation of a routine. For OS6 it was decided to regard the provision of such semi-permanent storage as a job for the operating system, rather than to require each individual program or compiler system to organise its own housekeeping.

Various algorithms exist for dynamic storage allocation. A good survey is given by Knuth [9]. Ross [13] describes the AED free storage package which, though much more sophisticated than ours, has some similarities. The choice of a system for OS6 was based on some experiments already carried out with the University's KDF9. As usual, the free blocks are chained together. There is a function (called *NewVec*) to find and remove a block of the required size, if necessary by dividing a larger block. A complementary routine (called *ReturnVec*) is used to return blocks of specified size to the chain. Because the word length in the Modular One is only 16 bits the free single words must be kept in a separate chain, as there is no room to record their size.

The chain of free blocks may be ordered in several different ways: the choice is thoroughly discussed by Knuth. After several experiments, the method selected for OS6 is to chain the blocks in order of location. When a block is to be allocated, the chain is scanned until the first block large enough is reached; this block is split if necessary. When a block is returned, it is merged with any block (or word) already free with which it is contiguous. This strategy is identical with the *GARB* strategy of the AED system; it is efficient in its use of available storage, though the overheads of splitting and merging are fairly expensive in execution time. Ross, whose system allows a choice of three different strategies, states that this one is "almost always best... where execution time cost is not important for infrequent wholesale transactions. Also sometimes it is a convenient way to squeeze out a workable version of a program which is tight on storage. Sometimes [this strategy] will use less physical space because it tends to prevent storage from becoming so fragmented that no [block] of suitable size can be found".

The possibility that a program might fail must not be forgotten when designing the storage allocator. In particular, a program might claim some space which it never returns. One solution is an automatic garbage collection system, as used in LISP. This, however, is impossible at present, as BCPL and the virtual machine make no distinction between an address and a number. It would be possible to attach to each word a hidden bit to state whether or not it contained an address, and to add some hardware rules to decide when to mark the result of an operation as an address (e.g. *number* + *number* = *number*, *number* + *address* = *address*, *address* + *address* is forbidden). In this way, a garbage collection scheme could be implemented, but only at the expense of altering the language by introducing some distinctions of type into its previously unstratified universe. The situation is similar in this respect to that described by Ross, where "apparently the propper solution to fully automatic garbage collection must await future language extensions...".

The experiment of constructing a garbage collection system for OS6 has not been made. Instead we have imposed on the free store system the hierarchical structure of Runs. When a program is Run, in the sense described above (§1.2), it is supplied with an area of store for use as off-stack storage. At the end of the Run all the area is forcibly reclaimed. The area employed for a Run is the largest free vector in the storage available to the program invoking the Run. The parameters of a storage area are kept in a seven word block, the FS-block, which is chained to the previous FS-block. Since it is quite possible for a vector claimed from one free store area to be returned while the system is operating in another, smaller, area, the system keeps a "pending chain" of returned blocks which fall outside the current area; whenever the system reverts to a previous free store area, it attempts to return any blocks in the pending chain.

We thus have a hierarchy of free store areas. This is another similarity with the AED system. There, however, a program may split an area into several "offspring" areas, each operating under different strategies, and the motivation seems to be to allow the programmer fine control over the details of allocation. Our concern is rather with protection against failure by allowing the storage allocator to keep in step with the *Run* mechanism.

It should be noticed that this regime prevents a routine under the control of *Run* from returning a vector as its result. This is something of a disadvantage. It is possible to regard the processor and core store as an evaluating mechanism which always leaves its results as files in the backing store. But in practice one wishes to leave results in core, and the desire to do this is counter balanced by the desire that the system should not allow any permanent changes to the core in case they turn out to be mistakes. This is a further example of the mismatch between a hierarchical structure which, because of the way humans think about problems, we find convenient to employ in our operating systems, and the idea of irrevocable change implicit in a finite storage mechanism. Compromise in this case is usually achieved by having the program invoking a Run provide a vector from its own resources to contain the eventual result; however, in Part 2 we describe a situation (the *PutBack* problem, §2.5.3) which requires special treatment.

3. OTHER FACILITIES

Most of the miscellaneous facilities in OS6 concern input/output, and their description is deferred to Part 2. In this section, however, we describe briefly the facilities available for debugging programs.

3.1. *Debugging facilities*

When the operator forcibly interrupts a program, he has the option of forcing a call either of *Finish* or of *GiveUp* (see §1.2). A third possible choice is of calling a "manual postmortem" routine. This provides a range of facilities for examining, and altering, the contents of any word in either the program or the data segments; there are also rather complicated provisions for resuming the execution of the program at any point in the current hierarchy of routine activations within the innermost call of *Run*. Though the ability to resume after a "binary patch" is sometimes very useful, whenever successful a patch should immediately be superseded by the recompilation of the amended source text.

This method of debugging, though convenient, is extremely expensive, and is in any case only possible in an environment where programmers are running their own programs and computer time is freely available. We plan to replace it shortly by a facility for dumping a core image on the disc for subsequent examination (eventually, it is hoped, from an interactive console). The possibility of patching will then no longer be normally available, though because it occasionally allows the talented system programmer to save the filing system from collapse it is too important to be abandoned entirely.

Because of the rather primitive facilities in BCPL for inter-segment communication, one of the more common run-time errors is the use of an undefined global routine (that is, an undefined element in the global vector). As an aid to debugging this sort of error, we arrange that undefined globals contain a special routine, called *Sleuth*. By using the return link planted by the routine call, *Sleuth* examines the program code leading up to the call, and is usually able to determine which undefined element was being accessed. The push-down nature of the global elements (§1.1) ensures that the value reverts to *Sleuth* after use.

4. PRACTICAL RESULTS

Work on the first operating system, OS1, began about three months before the computer arrived. Apart from the design and construction of the operating system and its bootstraps, it was also necessary to design the virtual machine and its machine code and to write the interpreter, to write a simple assembler (principally for the machine code sections of the bootstrap), to write a new code generator for the BCPL compiler already working on the KDF9 and to adapt the compiler to run under OS1. In this work the authors, each of whom had other responsibilities, were greatly helped by Mr. C. Hones, who wrote the code generator and assembler, and thoroughly checked all the other programming. The components of the system were tested as far as possible on the KDF9, and the entire system was compiled on the KDF9, using the code generator for the new machine, to produce a binary tape.

By courtesy of Computer Technology Limited we were able to use the machine on two occasions while it was being commissioned at the factory; this enabled us to assemble the interpreter and to debug the bootstraps. The machine was delivered and accepted on the 19th March, 1969, and OS1 and the BCPL compiler were running on it within 45 hours (most of this time was spent waiting for our daily access to KDF9 to correct the few faults which were discovered).

Since then the system has gradually evolved to its present form. OS4, for example, the first to contain a disc filing system, came in April, 1970.* In the meantime the system has been employed extensively by several users, and much insight has been gained by analysis of the various ways in which they have used it. This has enabled us not only to correct a few logical errors, but also to adapt the virtual machine by adding new instructions to speed up frequently occurring operations and generally to increase the system's efficiency.

* Since it has taken us two years to evolve from OS1 to OS6, we do not expect an imminent clash of names with the product of any other organisation.

5. CONCLUSIONS

This paper is a progress report: the research which it describes is still under way. It should not be regarded as a definitive statement even on a single-user operating system. It might, however, be useful to list what conclusions we have reached, and to discuss how our work relates to other research on operating systems.

5.1. *The "Single-user" simplification*

Our restriction to a single user situation separates us to a large extent from the mainstream of research, which is principally concerned with the problems of manipulating concurrent autonomous processes and controlling their interaction. The extension of our system to cover concurrent activities is our next step; then we expect to be able to draw considerably on work with other "clean" systems, for example those of Dijkstra [7], Hansen [8], and Spooner [14].

5.2. *Hierarchy and autonomy*

It is interesting that both Dijkstra's system (*op.cit.*) and ours may be described as hierarchical. In fact the hierarchies are quite different. Dijkstra has an hierarchy of resource allocation, because it is easier to administer one resource at a time; we allow hierarchical use of the system, because it is easier to think about a problem at one level at a time. So Dijkstra has a strictly hierarchically structured system to service a set of autonomous user processes, whereas our system is an amorphous set of routines to service a single hierarchically structured process. Moreover, an attempt to impose an hierarchical structure on our system (by forbidding the possible mutual recursion of our routines) would effectively prevent any hierarchy in the structure of the user job.

The conclusion to be drawn from this comparison is that hierarchy and autonomy are both essential features, in some way or other, of any operating system. Certainly our experience has been that most of our difficulties were examples of the clash between these two principles. So far we have simplified matters by having as little autonomy as possible; it remains to be seen what difficulties occur when we attempt to allow several autonomous, hierarchically structured processes. Clashes between hierarchy and autonomy are, of course, by no means confined to computing: history is full of more or less violent attempts to change the balance between them. We should perhaps study examples where fairly stable situations exist, to see if they can help us solve the computing problem.*

5.3. *Avoidance of a job control language*

Our hierarchy was made possible partly by our decision to avoid a "job control language", and to use a high level language instead. Barron [1], for example, is also thinking along similar lines, and rightly points out that the difficulties come when a system includes several languages with disparate conventions. But this problem is not confined to job control languages; it may occur when a user program calls on a system routine written in a different high level language. So far we have avoided this problem, too, by confining ourselves almost exclusively to a single language; we shall have to reckon with it seriously when we come to allow processes to be written in different languages, and even to be run on different virtual machines, controlled and serviced by the same operating system.

5.4 *Machine independence*

The problem of language compatibility within a system is more conspicuous when the operating system itself is written in a high level language. The great advantage of such a system, on the other hand, is its freedom from many of the problems of hard-

* Consider for example, the telephone system (which is biassed towards autonomy and works fairly well), local government (which, when, because humans are involved, it veers towards hierarchy, is less satisfactory) or a collegiate university like Oxford (about which we offer no comment).

ware compatibility. Provided the machines we consider have viable BCPL implementations, and provided their peripheral arrangements are satisfactory, the choice of one particular order code before another is governed purely by questions of compactness of code and speed. So the details of the IC machine are irrelevant to the success of the system we have described. Indeed, during our work with the system we have used several different virtual machines. As the BCPL compiler is written in BCPL, it is not difficult to rewrite the code generator for a new machine; as the operating system is in BCPL, we may then simply recompile it. In the last such exercise, by the expenditure of about two research-student-months, we "tuned" the order code, reducing the size of the code by about 25% in core and (because the amount of relocation information was also reduced) by about 30% on disc, and speeding up execution by about 15%.

5.5. *Importance of the interpreter*

The previous paragraph implies that it would be possible, by using the sophisticated BCPL code generator for Modular One machine code [3], to run the system on the Modular One itself, without an interpreter.* But we are convinced that our decision to use an interpreter was wise. It is the only inexpensive way at present to do practical experiments in processor design. The alternative is to use a microprogrammable machine , and it is sadly true that much of the current research on microprogramming seems to neglect the question of what kind of complex instructions could usefully be implemented: instead, the hardware designers have a new opportunity to avoid considering the needs of the software. In our situation, however, the advantage of microprogramming (a tenfold increase in speed) does not justify the extra expense and complexity. But we feel it is essential for the requirements of our programs to begin to influence the design of our hardware, and the flexibility provided by the interpreter has been of immense value.

We hope to publish the complete text of OS6 as Technical Monograph PRG-9.

* To make the implementation viable it would be necessary to circumvent the hardware restriction limiting the size of a code segment to 8K.

REFERENCES

[1] Barron, D.W. (1971). *Computer Operating Systems.* Chapman and Hall, London.

[2] Barron, D.W., Buxton, J.N., Hartley, D.F., Nixon, E., and Strachey, C. (1963). The main features of CPL, *The Computer Journal*, Vol.6, pp.134-143.

[3] Bath, P. (1970). *M1CG: A BCPL code generator for Modular One:* private communication.

[4] Brown, G.D. (1970). *System/360 Job Control Language.* Wiley, New York.

[5] Burroughs Corporation (1967). *B6500/B7500 information processing systems characteristics manual.* Detroit.

[6] Cox, P.R. (1971). Machine-independent operating systems: a functional approach to design. *The Fourth Generation, International Computer State of the Art Report*, pp.239-258. Infotech, Maidenhead.

[7] Dijkstra, E.W. (1968) The Structure of the "THE" - Multiprogramming System, *Comm. A.C.M.*, Vol.11, pp.341-346.

[8] Hansen, P.B. (1970). The Nucleus of a Multiprogramming System, *Comm. A.C.M.*, Vol.13, pp.238-241, 250.

[9] Knuth, D.E. (1968). *The Art of Computer Programming*, Vol.1, pp.435-451. Addison-Wesley, Reading, Mass.

[10] Needham, R.M., and Hartley, D.F. (1969). Theory and Practice in Operating System Design. *Second Symposium on Operating System Principles*, pp.8-12. A.C.M., Princeton, N.J.

[11] Richards, M. (1969). BCPL: a tool for compiler writing and system programming, *Proc. S.J.C.C.*, p.557. AFIPS.

[12] Richards, M. (1969). BCPL reference manual, *Technical Memorandum 69/1.* University of Cambridge Computing Laboratory.

[13] Ross, D.T. (1967). The AED Free Storage Package, *Comm. A.C.M* Vol. 10, pp.481-492.

[14] Spooner, C.R. (1971). A Software architecture for the 70's: Part I - The General Approach, *Software - Practice and Experience*, Vol.1, pp.5-38.

[15]* Stoy, J.E. and Strachey, C. (1972). OS6 - an experimental operating system for a small computer: Part II - Input/output and filing system, to appear in *The Computer Journal*, Vol.15.

[16] Wells, M., Holdsworth, D., and McCann, A.P. (1971). The Eldon 2 operating system for KDF9, *The Computer Journal*, Vol.14, pp.21-24.

* *Part II of this monograph.*

OS6 - An Operating System

For a Small Computer

Part II - Input/Output and Filing System

0. INTRODUCTION

In Part I [6] we discussed the general design of OS6, an experimental operating system running on a Modular One computer. This paper is devoted to a description of the provisions made for input/output in OS6, and a description of the disc filing system.

The input/output facilities are often the messiest parts of an operating system. The requirements are difficult to satisfy. On the one hand, the system must deal with the flow of information to and from several devices of different kinds. Some of this information may need processing - such things as character code conversion - before a program can conveniently use it; and one device may handle information of several different types. The paper tape reader, for example, handles binary code, which requires packing up into words before it is supplied to the loader, and also text tapes punched in a variety of character sets. On the other hand, a program should be capable of processing information of a given type no matter where it comes from - the program should not require rewriting for each new source. Very flexible provisions are obviously required.

1. CHARACTER SETS

Before we describe the general provisions for input/output, it is convenient to dispose at once of one of the difficult problems, which is that of choosing the character set and character codes, for information processed in the form of text. The problem is particularly acute if the system must cope with peripherals and data preparation devices with a variety of different character sets. A common solution is to deal systematically only with the *intersection* of the character sets in use. This leads to attempts to restrict high-level languages to, say, 48 characters. Since the main purpose of a high-level language is to make programs and programming more intelligible to human beings, such a restriction is unhelpful. The elegance of BCPL programs is due to a considerable extent to the extensive character set employed.

The OS6 solution to the character set problem is almost the reverse of that described above. The character set handled by most of the systems programs is practically the *union* of all the characters available on the various devices (though, for those devices which permit overprinting, only the overprinted characters meaningful in CPL and BCPL are included). This internal character set is represented by an eight bit code, known as *Internal Code*. One bit is used as an underlining indicator, and the remaining seven are based on ASCII, in the sense that the ASCII characters in the set have their ASCII values. There are a few unallocated values to allow for a limited extension, and there are also a few control characters. Although these include TAB, because it is useful when writing routines for controlling devices which use it, it is too device-dependent to be used in the system for any other purpose. To be forced to use only SPACE would be unacceptable, however, as half the characters in the average well laid out program would be spaces, and we therefore include a device-independent character, 4-SPACES.

The result of using Internal Code is that there is a unique representation inside the machine of the contents of any print position. This makes the design of input routines quite straightforward, and it also avoids making any preconceived assumptions about the nature of the information coming in (for example, by treating '&' and '∧' as synonymous, which might be true for logical formulae, but would not do for the names of businesses). It leaves any equivalence of characters to be dealt with, in Internal Code, by the program reading the data. Of course, a program may sometimes attempt to output a character to a device on which it does not appear: in this case the output routines will do the best they can, in an *ad hoc* fashion.

2. STREAMS

2.1. *Basic Properties*

The vehicles provided in OS6 for the transfer of information into and out of the system are called *streams*. Most streams are either *input streams* or *output streams*, though a few streams, such as those connected with a keyboard terminal, are capable of transfer of information in both directions, and are called *bilateral streams*. These are perfectly general objects, and their basic property is that a number of primitive functions and routines may meaningfully be applied to them. The most important primitive applicable to an input stream is the function *Next*, and for output streams the most important is the routine *Out*.

The result of applying *Next* to an input (or bilateral) stream is an object: the "next" object in the stream. Thus, *BytesfromPT* is an input stream of bytes from the paper tape reader, and the command

$$x := Next[BytesfromPT]$$

will assign to x the value of the next row on the tape (so that $0 \le x \le 255$, for eight-hole tape). It will be seen that two successive applications of *Next* to a stream will not, in general, return the same result. The same function *Next* is applicable to all input streams, and there is no restriction on the type of object produced. If *Next* is applied to a character input stream the result is a character, and if to a word stream the result is a word. Character streams and word streams occur most frequently, but it is also possible to have streams of strings, or of vectors, or of any other data type.

The routine *Out* takes two parameters, an output stream and an object, and its effect is to output the object along the stream. For example, if *BytestoPT* is an output stream of bytes to the paper tape punch, the command

$$Out[BytestoPT,\ x]$$

will cause a tape row corresponding to *x* to be punched (though, because of buffering arrangements, not immediately). The command

```
        Out[BytestoPT, Next[BytesfromPT]] repeat
```

would copy paper tape indefinitely.

2.2. *Stream functions*

Unlike most operating systems, OS6 treats streams as "first-class objects". That is to say, they may be freely assigned to variables, passed as parameters, or returned as the result of a function call. New streams are created by means of *stream functions*, which may be provided by the system or be defined by the user. These stream functions usually take a stream as an argument, and give a new stream as a result.

To illustrate the use of stream functions we may consider the problem of reading, with the paper tape reader, a text tape punched on a machine such as a flexowriter. We have already mentioned the stream *BytesfromPT*, which is an input stream of raw bytes from the reader. These, however, would be in flexowriter code, and would include shift characters, erase characters, runout and so on. What we require is a stream of characters in Internal Code, and to obtain this we use the stream function *IntcodefromFlexowriter* :

```
        let S = IntcodefromFlexowriter[BytesfromPT]   .
```

S is now defined to be an Internal Code stream, so that *Next*[*S*] will produce an internal code character corresponding to one on the tape. (Since the flexowriter allows backspacing, it is in fact necessary to read raw bytes corresponding to a whole line at a time, and to form a line image in some buffer, from which characters in Internal Code are read as required: all this mechanism is specified in the definition of *IntcodefromFlexowriter*.)

If, instead of a flexowriter, the tape had been prepared on an Olivetti terminal, we could have written

```
        let S = IntcodefromOlivetti[BytesfromPT]
```

and the rest of the program using *S* would be unchanged. We are thus able to confine the device-dependent part of the program to where the streams are defined (usually in some kind of steering program).

A stream produced by a stream function can itself be the argument of another stream function, and the functions can perform jobs other than character conversion. As an example, suppose the tape we have been considering is an Algol 60 program, and we are writing an Algol compiler. Then the layout characters on the tape (spaces, newlines etc.) are redundant, and we could write a stream function *RemoveLayoutChs* to remove them altogether. We could write:

let *S2* = *RemoveLayoutChs*[*S*]

or, more directly:

let *S2* = *RemoveLayoutChs*[*IntcodefromFlexowriter*[*BytesfromPT*]].

S2 is also an Internal Code stream, but *Next*[*S2*] will never produce any layout characters. (When we describe the implementation of stream functions we shall give the BCPL text of the definition of *RemoveLayoutChs*: see §2.5.2.)

An important property of stream functions is that streams produced by applications of one of them to two different arguments are quite independent. For example, if *S1* and *S2* are two Internal Code streams from different sources, we might define *S3* and *S4* by:

let *S3* = *RemoveLayoutChs*[*S1*]
and *S4* = *RemoveLayoutChs*[*S2*]

Calls of *Next*[*S3*] and *Next*[*S4*] could then be mixed in any order, and there would be no interaction between the two streams.

2.3. *Errors*

Problems arise when a stream has to cope with error situations: either invalid data coming in, or commands to output data unsuitable for the destination device. The difficulties are due to the wide choice of possible actions. One might abandon the program (that is, call *GiveUp*), one might simply ignore the offending item, or one might subject it to further analysis to determine what it ought to have been. It is impossible to build remedial action into a stream function sufficiently general to satisfy everybody. We therefore arrange that the majority of stream functions are available in a general form taking an extra parameter, which is an error function. E.g.:

```
let S = GeneralIntcodefromFlexowriter[BytesfromPT, ErrorFn].
```

The error function is called when invalid data subsequently occurs, and it decides what to do about it. The majority of programs, which do not need to take peculiar special action, use the non-general form, defined by the system:

```
    let IntcodefromFlexowriter[Str]
          = GeneralIntcodefromFlexowriter[Str,StandardErrorFn] .
```

The standard rule at present seems to be to produce an error report, and otherwise to ignore invalid input data, to replace valid but unprintable output characters by a space (so that they can be subsequently inserted, if required, by hand), and invalid characters if applicable, by a blank tape row.

An exception to the standard rule occurs if a stream functio is acting on the paper tape reader stream *BytesfromPT*. In this case a routine called *TryAgain* is applied. This is defined only on *BytesfromPT*, and only when nothing is "put back" to the stream in the sense described in §2.4.5 below: in all other cases it leads to *GiveUp*. Since the Modular One paper tape reader can read in either direction, it is possible to move the tape back in order to have another attempt to read the offending character, and this is what *TryAgain* does. The job is only slightly complicated by the fact that input is double buffered by *BytesfromPT*. After back-skipping the system pauses to allow the operator to inspect the tape, and to clean it up if necessary. A similar routine deals with sum-checked binary input, where it is necessary to reread a whole block.

The *TryAgain* technique is common practice in magnetic tape usage, but rare with paper tape. Particularly before the disc was delivered, however, this routine proved invaluable, as a great quantity of paper tape was read and it would have been excessively timewasting if every error had been catastrophic.

We have described dealing with errors in some detail, in order to make the point that designing an operating system to be elegant and coherent does not imply that we must pretend that errors do not exist. We can deal with particular errors sensibly without obscuring the basic structure of the system.

2.4. *Other primitives acting on streams*

In addition to *Next* and *Out*, which may be thought of as performing the operations "suck" and "blow", there are a number of other primitives which operate on streams.

2.4.1. *Endof*

Endof is a predicate which is applicable to input streams. It produces the result true if there are no more objects to be input. The interpretation of this criterion depends both on the source of the information and on its nature. When the information comes from a disc file the matter is simple: the housekeeping information on the disc will contain the length of the file. With information of indefinite length, however, like input from the paper tape reader, there is a difficulty. It might be solved by reserving a particular character to signify the end of the information, but this is unsatisfactory for streams like *BytesfromPT* which may have to read a binary tape in which every bit-pattern is significant. It is because of the impossibility of having a separate "end of stream" character that a separate function *Endof* is necessary at all. In the absence of any knowledge about the structure or nature of the information we cannot tell when it ends, and we therefore make *Endof*[*BytesfromPT*], for example, always false. Stream functions concerned with particular kinds of information decide according to their own conventions: text, for example, can be ended by a particular unusual sequence of characters, chosen *ad hoc* (we often use a full stop on a line by itself), while binary information will requir more elaborate rules.

This matter is also discussed by Needham and Hartley[5], who "do not believe at all that this whole problem can be swept under the rug by an appeal to convention". To surmount the difficulty of free-format binary information, they recommend the sensing by the device of the physical end of the medium - in our case, the end of the tape. We think that this is just as much a matter of *ad hoc* convention as any other action, since where the tape happens to run out or tear has no logical connection with the information on the tape. As an approximate test of this view, we wrote one of our stream functions to end when it detected over a foot of runout. This convention was abominated by all, and has been abolished.

Our insistence that determination of the end of the stream depends on the nature of the information it contains would lead to difficulties if we were managing an input well, or "spooling" the input, because we would then be processing information without regard to its content. It would be necessary to impose some sort of convention which could coexist with all the possible types of information - in the last resort, the operator could tell the system when the information had been completely read.

In situations where the terminating character approach is acceptable, it might waste time to be testing *Endof*[*S*] between each call of *Next*[*S*]. So we compromise by arranging that when *Endof*[*S*] has become true a subsequent call of *Next*[*S*] does not lead to failure; instead the result of *Next*[*S*] is a stream-dependent constant, usually known as *EndofStreamCh*.

2.4.2. *Reset*

Reset is a routine applicable both to input streams and to output streams, which restores them, in some sense which varies from stream to stream, to their initial state. In the case of output streams, any information temporarily in buffers associated with the stream is forced to its final destination, while for input streams any information in buffers is discarded, so that the next object requested will be read at that time from the input device. Other action may also be taken, such as setting an input device unready, or moving to a new page on a printer.

2.4.3. *Close*

Close also acts both on input and on output streams. It forces out any information in output buffers, informs the system that the stream is no longer required, and returns its storage areas.

2.4.4. *State and ResetState*

Usually, a stream is either an input or an output stream. Sometimes, however, an input device and an output device, although logically separate, are physically on the same chassis; then, as we have already mentioned, for administrative convenience we combine the two streams into a single bilateral stream. There is another kind of information which can be obtained from a device: rather than obtaining another new object (as *Next* does), we can look at something to see whether it has changed. For example, we could look at the on-line/off-line switch on the reader. In the case of consoles, the question usually is: "Has anyone typed anything yet, and, if so, what?" Again, this kind of information is logically separable from the stream-like kinds; but since both kinds come from the same machine, it is convenient to include it in the stream.

We therefore define two new primitives on streams. The first is a function, *State*[*S*], which produces the current state of the device. For those devices where the state is defined by asking whether some event has yet occurred, we also need a routine, *ResetState*[*S*], to reinitialise the state. (Possibly the *Reset* routine, described above, could also do this, but it seems cleaner to have a separate routine.)

State and *ResetState* were a later addition to the scheme, and for reasons of domestic economy have so far been implemented only for bilateral streams. They are most frequently used when the machine is performing some repetitive operation, in order that the loop may be broken when the operator types a character on the console. Thus the tape-copying loop quoted above might be modified to read:

```
Out[BytestoPT, Next[BytesfromPT]]
    repeatwhile State[Console] = NOTHINGTYPED
```

(in fact our tape-copying program is a little more elaborate). Notice the essential difference between *State* and *Next*: if *Next* is called, the program is held up until a new object comes along, whereas *State* can return the null answer.

2.4.5. *PutBack*

We frequently require to perform operations like reading a multi-digit number from a character input stream. This raises the interesting question of what to do with the terminating character. It must not be simply absorbed as part of the number, for we may later require to consider it independently. For example, we may be trying to parse an expression like

$$27+a$$

and we shall obviously require to know that the character terminating the number was '+'. It would be possible to leave the character in a conventional location; but then it would be necessary to take care to remove it immediately, before the location was used by something else. The cleanest solution would be, if possible, to return the character to the stream, so that it could be produced again the next time there was any input. This is done by the routine *PutBack*, which takes two arguments, an input stream and an object; for example:

PutBack[Stream,TermCh] .

Then, the next time we call *Next[Stream]*, the result will be *TermCh*.

PutBack is of unrestricted application. It may be used on any input stream; it may be used to put back several items *seriatim* to a stream (the last item put back will be the first to reappear); and there is no need for the items put back to have come from the stream in the first place.

2.5. *Implementation*

2.5.1. *Implementation of Streams*

In principle a stream is represented by a data structure, the components of which are functions and routines for performing the primitive operations. In BCPL this is implemented as a vector. Thus if S is an input stream, the element S_0 (which in BCPL is typed as $S\downarrow 0$, the zeroth element of S) is reserved for the function which produces the next object. Let us call the function *NextFn*. *NextFn* will require some working variables to survive from each activation to the next, in order to keep pointers, buffers and so on. In particular, if S was produced from a stream function,

$$S = StreamFn[ArgStream] \quad ,$$

then *NextFn* will require to refer to *ArgStream*.

The mechanism for referring to non-local variables which is built into the BCPL language is inadequate to deal naturally with this situation. (So for that matter are those in Algol 60, Algol 68 and PL/1; two languages which are sufficiently powerful are PAL [3] and POP-2 [1].) This means that we must make special provisions to preserve the information ourselves, which we do by keeping it all in the vector S. The length of S may therefore vary from stream to stream, but the first few elements are always reserved for the basic functions and routines.

To obtain the next object from S, we must supply S to *NextFn* as a parameter,

$$NextFn[S] \quad ,$$

in order to allow access by *NextFn* to elements of S. Since *NextFn* is itself stored in S_0, we may define the general primitive function *Next*, applicable to all input streams, by writing

let *Next*[*S*] = (*S*↓0)[*S*] .

The other primitives are similarly defined; for example

let *Out*[*S*, *x*] be §(*S*↓1)[*S*, *x*] $.

We arrange that in input streams the element corresponding to the *Out* routine contains an error routine, and *vice versa*. Note that the functions and routines which operate on streams store the information they must preserve from one call to the next in the stream vector itself; this means that they can be used on several different streams in the same program without confusion.

2.5.2. *Implementation of Stream Functions*

The result of applying a stream function is a new stream; the function must therefore claim a new vector from the storage allocator, and place in it the functions and routines to perform the standard operations, together with any other initial information that may be necessary, including, for example, the stream supplied as the parameter for the stream function.

To illustrate this, we give the BCPL text of a particularly simple example, the function *RemoveLayoutChs* described in §2.2 above. To simplify still further, we will ignore *State* and *ResetStat* As *PutBack* does not require a vector element (see §2.5.3 below), the vector has to contain five standard elements (for *Next*, *Out*, *Close*, *Endof* and *Reset*). It must also contain the argument stream, so a vector of six elements is required; its layout is shown in the figure.

0	*NextRLC*
1	(*Out*) an error routine
2	*CloseRLC*
3	*Str* (the argument stream)
4	*EndofRLC*
5	*ResetRLC*

The fact that the third element is not reserved for a standard operation, and is therefore available for *Str*, is for historical reasons. In any case, we shall ignore its embarrassing arbitrariness because, to improve readability, we shall refer to the elements by name. We therefore define the following constants (from now on in this section we will be writing BCPL; comments in this language are introduced by two vertical bars (||) and continue to the end of the line):

```
manifest §
     NEXT = 0
     OUT = 1
     CLOSE = 2
     STR = 3
     ENDOF = 4
     RESET = 5
     VECSIZE = 5
          §
```

The definition of *RemoveLayoutChs* is then as follows :

```
let RemoveLayoutChs[Str]
§RLC  let v = NewVec[VECSIZE] | We claim a vector
      v↓NEXT := NextRLC       |
      v↓OUT := StreamError    | and initialise the standard
      v↓CLOSE := CloseRLC     | contents (note that StreamError
      v↓ENDOF := EndofRLC     | is a system error routine).
      v↓RESET := ResetRLC     |
      v↓STR := Str
      resultis v              §RLC
```

We must now define the subsidiary routines. The most important is *NextRLC*.

```
and NextRLC[S] = valof
§N §1 let x = Next[S↓STR]     | We read a character from the
                              | argument stream
      unless x = '*s' v x = '*4' v x = '*n'
                              | Unless it's a layout character,
      then resultis x         | it's the result;
   §1 repeat           §N     | otherwise we repeat the process.
```

Note that '*s' means space, '*4' the 4-space character, and '*n' newline.

|| We next define *CloseRLC*: we close the argument stream,
and return the storage space.

```
  and CloseRLC[S] be
  §C  Close[S↓STR]
      ReturnVec[S, VECSIZE]    $C
```

|| The simplest and fastest definition of *EndofRLC* is merely
to test the end of the argument stream :

```
  and EndofRLC[S] = Endof[S↓STR]
```

|| This, however, would be wrong if *Str* ended with layout
characters, because our function would give the result
false when in fact there were no more characters to come.
If this were important, we would have to get more complicated :

```
  and EndofRLC[S] = valof
  §E  let Str = S↓STR              || Str is the argument stream.
      §1  if Endof[Str] resultis true
        § let Ch = Next[Str]     || Look at the next character.
          unless Ch = '*s' v Ch = '*t' v Ch = '*n'
                                   || If it's not a layout character,
          do § PutBack[Str, Ch]    || put it back on Str,
               resultis false $    || and the answer is false;
      $1  repeat        $E         || otherwise, repeat.
```

|| Note that although *PutBack[Str, Ch]* is more
obvious, *PutBack[S, Ch]* would have been more efficient.

|| *ResetRLC* is, however, simple - we merely reset the argument
stream :

```
  and ResetRLC[S] be Reset[S↓STR]
```

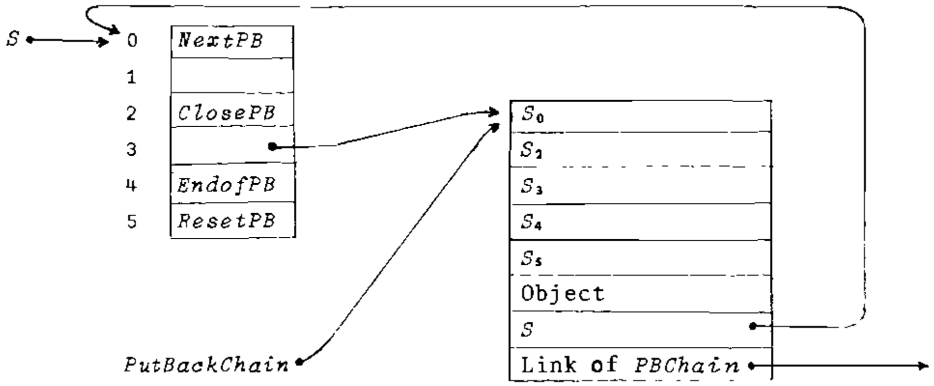
|| That completes the example.

2.5.3. *Implementation of PutBack*

A call of the routine *PutBack* is of the form

PutBack[*S*,*x*]

where *S* is an input stream and *x* is an object. The routine claims a small vector from the free store and stores in it the returned object, and also the values of some of the first few elements of the stream vector, which it then overwrites with other routines. The final situation is as shown in the figure.



Then, when *Next* is applied to *S*, *NextPB* is activated. This restores *S* to the *status quo*, and returns the *PutBack* vector to free storage; its result is then the object put back. *ClosePB* and *ResetPB* also restore the previous state and then apply the appropriate original routine; the result of *EndofPB* is always false. The *PutBack* vectors are chained together for a reason described below, and when a *PutBack* vector is removed, care is taken to heal the breach in the chain. Extra care has to be taken with bilateral streams, to ensure that the output part of the stream still works when an object has been put back to the input part, since some elements in the vector are thereby overwritten.

This implementation gave rise to the following difficulty. If the last action on an input stream before the end of a Run is to perform *PutBack* (e.g. after reading a number), then the *PutBack* vector is returned together with the rest of the free store area, and the stream is thereafter unusable. This is troublesome, of course, only when the stream vector itself is claimed from an earlier free store, and therefore remains in existence longer than the *PutBack* vector.

This particular problem required a special solution. The free storage system was altered, so that when it reverts to an earlier free storage area it copies into the earlier area any *PutBack* vectors still in use. This is why the *PutBackChain* is required.

This is an example of the clash between the hierarchical structure of OS6 and the nature of storage mechanisms, which was discussed in the previous paper. The solution must be *ad hoc*, because in BCPL there can be no systematic way of relocating vectors of information. Fortunately, this is the only place where the system requires its use of free storage to transcend the discipline of the *Run* system, and the special solution is therefore satisfactory. The only general solution would be to alter the language to allow garbage collection, and to make all off-stack storage permanent.

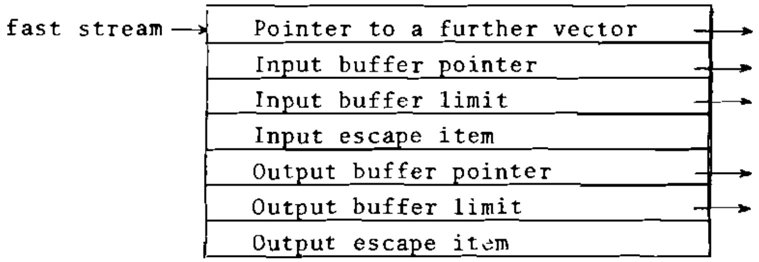
2.6. *Efficiency of Streams*

A possible objection to the use of streams might be that the overheads associated with their structure make them excessively inefficient. Certainly, when a stream is formed from a deep nest of stream functions, processing a single character can involve many function calls. To some extent a greater expenditure of time than usual is unavoidable, simply because the flexible nature of streams and the ease of nesting stream function calls lead to the possibility of specifying much more complex operations to be performed on each character. It is sometimes profitable to examine the program to ensure that some of these operations do not undo the work of others. A stream, for example, might be formed by one stream function which unpacks words into bytes, followed by another one which packs them all up again.

An improvement in speed may be made by streamlining the definition of *Next* and *Out*, by hand-coding them into machine code, and this was done from the start (that is, they were written in virtual machine code - see Part 1 (§0.2) for a definition of "machine code" and "hardware" in this discussion). Further analysis was done by taking measurements of the actual usage of the operating system during the compilation of the null program, when most of the time is spent in loading the compiler. A histogram was produced showing how often each group of 10 words of code had been accessed (there are about 10,000 words of code in the operating system). The result was a startling concentration into a few peaks with virtually nothing measurable in between. Further analysis showed that most of these peaks were at pieces of program concerned with taking single items out of buffers and putting them somewhere else after testing for various conditions such as the end of the buffer.

Our system runs in a virtual machine, which is implemented by an interpreter. We can therefore easily add new instructions to our virtual hardware, merely by extending the interpreter. We have used this facility several times in order to replace frequently occurring operations by single instructions, thus increasing the efficiency of the system. This activity is quite legitimate, provided that the instructions we add are such as could reasonably be implemented in real hardware if required. The ability to proceed in this way is very liberating, and is in accord with our general philosophy of not allowing ourselves to be bullied by machines. The conclusion to be drawn from our statistical investigation was that the buffering operations were obvious candidates for such optimisation.

A new kind of data structure, called a *fast stream*, was devised. A stream is marked as being either fast or slow (in practice by using the sign bit which is not required to be part of the address). If a stream is marked as being slow, it is a normal stream of the kind we have already described. If it is a fast stream, however, we may derive from it the address of a vector, of the form shown in the figure.



The definitions of the primitive stream operations are extended to deal with fast streams. If *Next* is applied to a fast stream, the normal action is to return as result the object referenced by the input buffer pointer, and to increment the pointer. If the pointer has reached the limit, the appropriate routine is called in the further vector, which is very like a normal stream, to refill the buffer. It is convenient also to break out of the buffering if the item picked up is equal to the input escape item. *Out* is defined similarly. If *S* is a fast stream, *Endof*[*S*] is false if the input buffer pointer is below its limit, and otherwise a function in the further vector is called. The other primitives merely activate routines in the further vector.

Single hardware instructions corresponding to *Next*, *Out* and *Endof* were written into the interpreter; the other primitives are used less frequently, and are therefore defined in BCPL. Hardware instructions were also written to implement the routine

$$TransferIn[S,v,n]$$

where S is an input stream (slow or fast), and v is a vector of length n; its action is to place in the elements of v the results of n calls of *Next*[*S*]. The corresponding output routine

$$TransferOut[S,v,n]$$

was implemented by hardware too.

The result of all this was to reduce the time for compiling the null program by a factor of six. We feel that this is the proper way to treat problems of efficiency. Peter Landin has remarked (in a private communication) that most programs are designed to be as fast as possible - so that one then goes through a lengthy process (debugging) of improving the correctness to a tolerable level while preserving the speed - whereas the sensible course would be to design them to be as correct as possible, and then gradually to increase the speed till it is tolerable while preserving the accuracy. (Dijkstra [2] suggests that this is because many programmers find debugging so much fun that they could not contemplate giving it up, because the element of black magic in it satisfies one of our most undernourished psychological needs.) Streams were designed to be elegant, because in the long run this is the best guarantee that programs using them will be correct; questions of efficiency, including the decision about which operations should be done by hardware, were attended to later.

It may be convenient in the future to modify the behaviour of fast streams. In particular, for doing more complex activities like syntax analysis, there may be advantages in replacing the escape item test by something more elaborate, such as a masked test.

2.7. *System Streams*

OS6 contains one or two permanent streams, closely associated with particular peripheral devices. An example is *BytesfromPT* which is the only route by which paper tape is read by the system. These streams are permanent in the sense that an attempt to close them merely resets them.

In addition to the permanent streams, OS6 has four global variables to hold streams reserved for conventional purposes. These are called variable streams, and may be freely altered by programs. Their values are preserved in the Run-blocks (see Part I §1.2.1), and they are restored to their previous values at the end of each Run. The four variables are *In*, the normal input stream; *Output*, for normal output; *ReportStream*, for error reports; and *Console*, for messages to or from an operator's console.

2.8. *Input/output routines*

Programs frequently require to print numbers, strings etc. on a character device. It would be possible to do this by stream functions: for example, one which when applied to a character output stream would provide a stream for the output of integers.

However, programmers usually need finer control over the layout of their output, and prefer to work directly with characters. For this reason a set of routines is provided to output items of various types along character streams. A typical one is *OutN[S,n]*, which outputs *n* as a decimal integer along the stream *S*. As well as this set which takes the stream as a parameter, two other sets are provided to perform the same operations specifically on *Output* and *ReportStream*. Input functions are also provided for reading a similar range of items from input streams.

3. THE FILING SYSTEM

It now remains for us to describe the outline of the filing system, the regime under which information is kept on the disc. We do not claim any great originality or sophistication in the design of the system. However, this is an area in which there tends to be confusion - it is easy, for example, to muddle the name of an object with the object itself - and we have taken care that our system should be "clean", and that its structure should be clear.

Our previous discussion on input/output in OS6 has been based on the idea of streams. These could be freely manipulated in the programming language and had the same status as any other type of object. We now extend this approach to another kind of object, called a *file*. These, like streams, may freely be assigned to variables, be passed as parameters, or be the result of function calls. This implies that each file has a unique *value*, which may be stored in a single BCPL variable, and is the handle by which to access the two components of the file's structure, the *heading* and the *body*.

Each file has its own unique heading. This contains various items of housekeeping information about the file, including the means by which the system can access the body. The body contains the information stored in the file, and also belongs exclusively to one file: files do not share components. An empty file has no body.

3.1. *Some basic functions*

As in the case of streams, the basic property of a file is that a number of system functions may meaningfully be applied to it. One of these, *FindHeading*, produces the heading of the file as a vector in core:

let H = *FindHeading*[f]

(The contents of this vector are given in full below, in §3.4.). One of the fields of H is called the *title* of the file. It is a BCPL string, of arbitrary length, and its sole purpose is to contain a description, fit for human consumption, of the contents of the file. The properties of the file which might concern a program

- such as the date it was created, its owner, or the type of information stored in it - are all kept in other fields of the heading, and it is not intended that the system should do anything with the title except print it out from time to time. In particular, the title is *not* used when the system is searching for a file.

Most of the entries in the heading of a file (including all those already mentioned) are invariant: they are set when the file is created and may not subsequently be altered. Some of them - such as the date the file was last changed, or its size - are updated automatically by the system. Only one field, which contains an entry stating who is allowed to overwrite the file, may be altered by the programmer (by calling a special routine, *UpdatePermission*).

3.1.1. *Streams from files*

Other basic functions concern the filebody. The commonest way of reading a file into the system is by forming an input stream from it, by the function *InfromFile*:

let *S* = *InfromFile*[*f*] .

S is a word input stream, so that if *f* is a file of packed bytes it is necessary to apply the stream function *BytesfromWords* :

let *S*2 = *BytesfromWords*[*InfromFile*[*f*]] .

S is a fast stream. If at any time it is reset, then any subsequent calls of *Next*[*S*] will recommence from the beginning of the file.

The corresponding function to produce output streams to the file is also available:

let *S*3 = *OuttoFile*[*f*] .

Since this involves overwriting, it is designed to minimise accidents in the case of error. As output *via* *S*3 proceeds, a new body is constructed, but only when *S*3 is closed does this new body replace the old one. Thus if the program fails before *S*3 is explicitly closed, *f* will not be altered; instead the *new* body is abandoned.

3.1.2. *Vectors from files*

Instead of forming a stream from the information in a file body, it may be treated as a vector. There are two possible ways of doing this. If the body is not too large, the simplest way is to transfer it completely to a vector in core. A function is available for this:

let *v* = *VectorfromFile*[*f*] .

By convention v_0 contains *n*, the size of the body in words, and the body itself is in v_1 to v_n. The complementary routine is also available,

VectortoFile[*f*,*v*] ,

in which the same convention is observed.

If the body is too large to exist in core, another mechanism is needed. An object called a *disc vector* is defined, which may be produced by the function *DiscVectorfromFile*:

let *Dv* = *DiscVectorfromFile*[*f*] .

Then a particular element of the body may be accessed by a further function:

let *El* = *DiscVectorElement*[*Dv*,*i*]

or updated by a routine:

UpdateDiscVectorElement[*Dv*,*i*,*x*] .

Dv is a vector in core which contains addressing information allowing reasonably quick random access to any part of the file body. When no longer required it may be relinquished by a call of the routine *ReturnDiscVector*[*Dv*].

3.1.3. *File creation*

Files are created by the function *MakeNewFile*. This is given the title of the file and its type as parameters:

let *f* = *MakeNewFile*['*Line printer stream functions*',*BCPLTEXT*] .

The result is a new empty file.

3.2. *Indexes*

The file *f* may be created and used freely within a single program with no further apparatus. "*f*", however, is the name of an ordinary BCPL variable, and is governed by the usual scope rules: that is, it refers to the file only inside the block of program at the top of which its definition occurs. If the file is to be usable by other programs, we need another mechanism. This mechanism is provided by *indexes*.

An index is a file (of type *INDEX*) on which a number of special routines are defined. When a file is entered in an index, two *names* (BCPL strings) are associated with it. This is done by the routine *Enter*.

```
                    Enter[f,p,q,i]
```

means: "associate the names *p* and *q* with the file *f* in the index file *i*." Particular values for *p* and *q* might be, for example, '*LinePrinter*' and '*Text*'. If another file is already associated with *p* and *q* in *i*, the new mapping supersedes the old.

The complementary operation is provided by the function *LookUp*:

```
            let f = LookUp['LinePrinter','Text',i]
```

defines *f* to be the file previously associated with the names '*LinePrinter*' and '*Text*' in the index *i*. If there is no such file, the result is the constant *NIL*: the request does not lead to failure. On the other hand, an attempt actually to access the components of a non-existent file, for example by setting up a stream, implies that the program has definitely gone wrong, and such an attempt leads to *GiveUp*.

A further function applicable to index files is the stream function *EntriesFrom*

```
                      let S = EntriesFrom[i]      .
```

The result of *Next[S]* is then normally a vector containing the two names and the value of the file (there is also another form of index entry which we shall mention below). This enables programmers to perform operations like: "Compile all the files in the index with the second name '*OS6*'." It should be noted that the system itself attaches no semantic significance to either index name - all the information the system requires about a file is in the heading - but the fact that there are two names allows the programmer to systematise his indexes in any way convenient to himself.

3.2.1. *Index structure and sharing*

There is a special index, called the system index, which the system keeps in a global variable called *SystemIndex*. Each user of the system has his own index, for which there is an entry in the system index. So one may write, for example:

```
    let i = LookUp['JES','Index',SystemIndex]

    let f = LookUp['LinePrinter','Text',i]
```

or, all at once:

```
let f = LookUp['LinePrinter','Text',LookUp['JES','Index',
                                             SystemIndex]]    .
```

In fact, any index may be entered in any other index, not merely in the system index.

A single file may be associated with different pairs of names in different indexes, or even in the same index. This is one form of sharing of files: two different entries point to the same file. There is another form of sharing sometimes employed, which is available in OS6. Instead of two entries pointing to the same file, one entry may point to the other entry (this is the 'special' form of index entry referred to above). Such an entry is constructed by the routine *Link*, which takes seven arguments,

```
                    Link[N1,N2,N3,N4,N5,N6,i]     .
```

This means: "Construct a special entry in the index *i*, so that the names *N1*,*N2* refer to the entry with names *N3*,*N4* in a second index. This second index is entered with names *N5*,*N6* in the system index." Then a call

```
                        LookUp[N1,N2,i]
```

will initiate a search in the second index. In principle links may occur to any depth; care is taken when setting up a link to ensure that the chain of links does not lead to a loop of references to each other.

There is a third method of sharing the information in a file, which is simply to copy the contents into a completely new file. Which of these three methods is used in any particular situation is partly a matter of personal taste. The first seems to be most popular in our group. It is possible, however, to imagine a situation in which there is a real choice between all three possibilities.

Suppose a user *A* keeps programs on the disc, and has the convention that when correcting a mistake he overwrites the file body, but if he alters the specification of a program he creates a new file and changes the entry in his index to point to the new program. Then a second user *B* who wishes to access one of *A*'s programs can choose as follows. If he wants the program as it stands, ignoring any of *A*'s later alterations, he copies the file. If he wants to benefit when *A* corrects a mistake, but not to have the specification changes, he constructs an entry pointing to *A*'s file. If he wants to keep up with *A*'s latest ideas on what the program ought to be doing, then he constructs an entry pointing to *A*'s entry.

3.3. *Deletion of Files*

There are three sorts of deleting possible in the system. One may delete the body of a file, preserving the file itself and any index entries pointing to it; one may delete the file, heading and body, but not affect any index entries; or one may delete an index entry, which will not affect the file the entry pointed to. All three kinds are separately available in OS6. Of course, an index entry pointing to a deleted file is not much use, and a file is inaccessible unless at least one entry points to it: these matters are the concern of a special housekeeping program, which performs a garbage-collection operation on the filing system.

3.4. *Outline of implementation*

Files on the disc are accessed through a Master File List

(MFL). The value of a file is the serial number of its entry in the MFL; the MFL entry gives the disc address (page and word) of the file heading. The Burroughs disc has fixed heads, so the access time is half a revolution; moreover, the use of an interpreter slows down the rate at which the system can process information. There is therefore no advantage in optimising the position of the pages in the body of a file; they are allocated in no particular order, and the pages of a body are chained together. Usually, the last body page contains a pointer indicating how much of it is occupied (thus avoiding the *Endof* problem discussed earlier in §2.4.1).

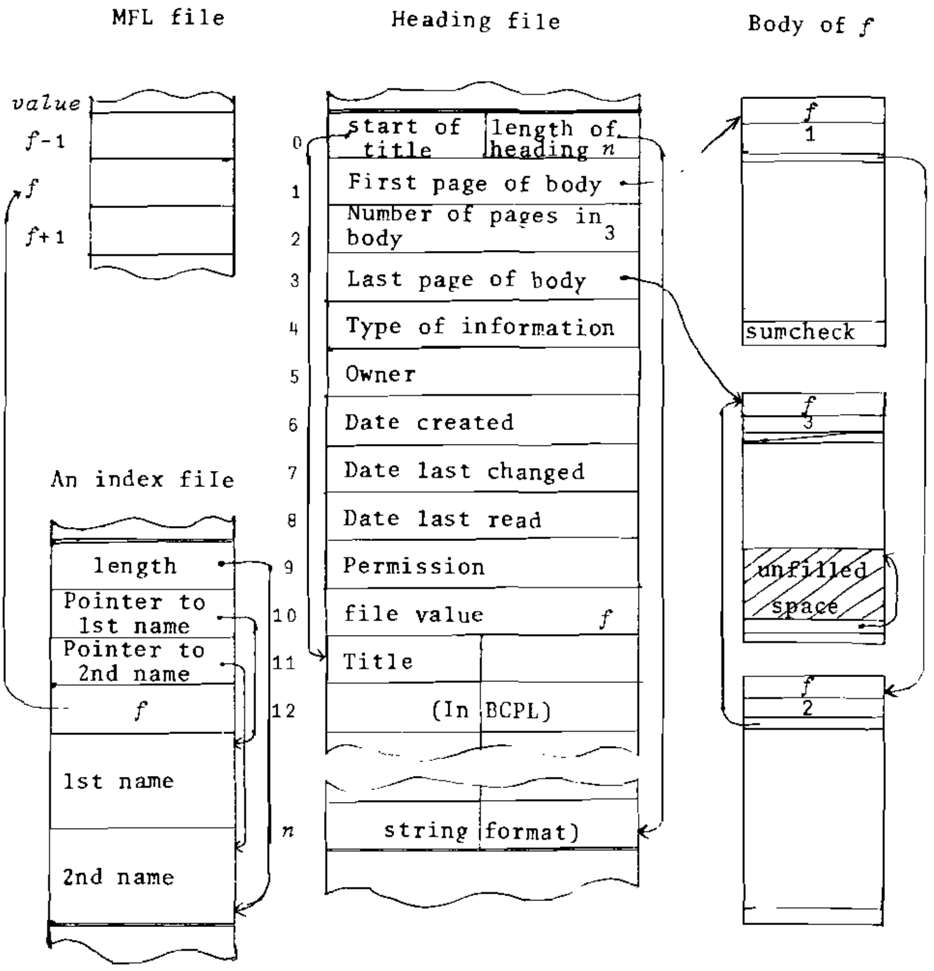
A diagram of the structure is given in the figure (p.59). It should be noted that the headings are kept all together in a heading file, and the MFL is also a file. This implies, of course, that the heading file will have its own heading, somewhere in its own body, and the MFL will contain its own entry. It is necessary to know two quantities in order to access the structure: the address of the first page of the MFL body, and the value of the system index. The second of these is kept in a global variable, set up when the system is initiated, and is available to user programs; the former is private to the system, and is incorporated as a constant declaration at the head of the appropriate segment of the text of the system. This is a mistake: if the particular page developed a fault, it would be impossible to use the filing system until the segment had been recompiled - and the normal compiler uses the disc.

3.4.1. *Disc storage allocation*

The addresses of all the free pages on the disc are kept in a file, the free storage file. For efficiency's sake a page of this file is kept in core, and for safety's sake this core is in the program segment. It is written back to the disc at frequent intervals (at the end of each Run, and whenever the page kept in co
changes to another page).

3.4.2. *User details*

The system has a file of users. A user's entry contains



his name (a string), the unique number by which he is known to the system (which gets entered in the owner field of his file headings), and his main index. A user may "log in" by specifying his name on the console: the system places his personal details in various system variables (*User*, *UserIndex*). The entry also contains the size of the user's allocation of disc storage space. The system warns him when this is nearly used up, and he is prevented from exceeding it.

3.4.3. *Protection from errors*

Protection in the filing system is of two kinds: we try to prevent the occurrence of accidents; and we keep a certain quantity of redundant information, so that if a crash does happen we have some chance of reconstituting the system, rather than having to restart it *ab initio*.

The most important prophylactic is the permission system, governing the access programs may have to files. Since we are solely concerned to prevent accidents, and have no confidential files, we allow anybody to read anything; there are only three values for the permission field in the heading, which have the following meanings:

UNRESTRICTED	:	anyone can write to the file
OWNER	:	only the owner permitted to write
INHIBITED	:	no one may write to the file.

The permission field may be changed only by the owner (or anyone masquerading as the owner).

Our present philosophy is not to be constantly checking the redundant information. Every so often we run a disc validation program, which thoroughly checks everything, and we investigate any discrepancies. If there is a crash, we have an armoury of little programs to aid the system programmers in sorting out the system.

3.4.4. *Garbage collection*

It is probable that the filing system will contain some outdated information in inaccessible places. This may be found and removed by a garbage collection system, which operates in well defined phases. Firstly, the system index and any deeper indexes are scanned, and entries referring to deleted files are removed. Having purged these indexes, we construct a list of the files entered in them, and delete all files which do not occur in the list. At this stage the heading file may be compacted. Finally, we form a list of the disc pages used by the files which remain, and ensure that all other pages are entered in the free storage files.

3.5. *Extensions to the filing system*

We may require to extend the filing system to deal with a more sophisticated system, in particular (a) automatic incremental dumping, if suitable extra equipment becomes available, and (b) the possibility of several users' using files simultaneously through a console system. This will probably require new fields in the heading of a file, to contain new items like its current status, for interlocking purposes. None of this will require changes to the basic principles of the system.

4. RECAPITULATION

In any description of an input/output system it is only too easy to lose sight of the basic outline. Although we wanted to include some merely corroborative detail intended to give artistic verisimilitude to an otherwise bald and unconvincing narrative [4] it may be worth while repeating the principles which we hope underlie our system.

The main objects manipulated in the system are *streams* and *files*. Both may be handled freely by all the facilities in the programming language. In particular, new streams may be created by *stream functions*, which may be written by the user, and which usually take a previously defined stream as argument, returning a new stream as the result. (A new file cannot be created by a user-defined function because the medium in which they are stored is administered solely by the system.)

Streams and files are characterised by the basic functions and routines which act on them. For streams, the most important of these are *Next* and *Out*, which respectively obtain an object from an input stream and consign one to an output stream (the type of the object depends on the stream, and the same basic routines can be applied to all streams).

The basic functions on files allow the information in the file to be accessed in various ways (random, serial or all at once), and allow a file to become the subject of an entry in an *index*. The index entry, however, is not part of the file itself.

5. ACKNOWLEDGEMENTS

The work on streams has been going on for some years, and began long before we had a computer or operating system of our own. Several people have helped in the implementation of the system described in these two papers. Our thanks are due particularly to Julia Bayman, B. N. Biswas, M. K. Harper, C. Hones and P. D. Mosses.

The work described in §2.6 was performed by P. H. F. McGregor.

REFERENCES

[1] Burstall, R.M., Collins, J.S. and Popplestone,R.J. (1968). *POP-2 papers.* Oliver & Boyd, Edinburgh and London.

[2] Dijkstra, E.W. (1971). Concern for correctness as a guiding principle for program composition. *The Fourth Generation, International Computer State of the Art Report,* pp.357-367. Infotech, Maidenhead.

[3] Evans, A.,Jr. (1968). PAL - a language for teaching programming linguistics. *Proc. ACM 23rd National Conf.* Brandon/Systems Press, Princeton, N.J.

[4] Gilbert, W.S. (1885). *The Mikado* or *The Town of Titipu. Act II.* Chappell & Co.Ltd.

[5] Needham, R.M., and Hartley, D.F. (1969). Theory and practice in operating system design. *Second symposium on operating system principles,* pp.8-12., A.C.M, Princeton, N.J.

[6]* Stoy, J.E. and Strachey, C. (1972). OS6 - an experimental operating system for a small computer: Part I - general principles and structure. To appear in *The Computer Journal,* Vol. 15.

* *Part I of this monograph.*