LANGUAGES FOR WRITING OPERATING SYSTEMS

Like everything else which a computer does, excepting only a rather few primitive actions built into the hardware, the tasks undertaken by an operating system are controlled by a programme which is executed by the computer, and which somebody, or some group, has written. It is self-evident that the quality of the operating system is a direct reflection of the quality of this programme, so any measures which will improve the quality of the programme are of interest.

One such measure is the language used to write the programme. Experience has shown that a programming task is greatly facilitated if it is carried out in a language which provides programme structures and data structures appropriate to the task to be performed. Because of this fact, very many languages have been devised for different sorts (or, too often, the same sort) of programme. Pascal provides facilities appropriate to traditional mathematical algorithms, while Prolog caters for first-order predicate calculus. Each is effective within its own field; each can manage the other's job, but not as easily. At the level of principle, most non-trivial computer languages are equivalent, in that they can all handle the same range of computational problems. It is therefore not usually sensible to argue that language *L* is appropriate for a job simply because *L* can do it. It is more important to ask whether *L* provides the sort of support we need to write the sort of programme we want – accepting that the idea of "support" might include such matters as compatibility with existing software written in a language chosen for different reasons.

REQUIREMENTS FOR OPERATING SYSTEMS PROGRAMMING.

That being so, we ask what sort of support is required to write an operating system. Even better, we ask what *sorts* of support are required, for we want as much help as we can get from our language. We can easily identify at least three distinct ways in which a good language can help us : in programming, in documentation, and in safety.

Programming is obvious. Our language must make it easy to write the programmes we want to write by providing as much as possible of the job already done for us. This is the substance of the discussion at the head of this chapter.

Documentation is also important. A programme is not only a means of communication between a programmer and a computer; it must also communicate the sense of the programme between programmers. If you want to correct or extend a programme which you can't easily understand, you have to spend a lot of time working out what it does before you can begin your real job. Paper documentation can be very helpful, but regrettably often isn't; and in any case the more help you get from the source language, the better.

Safety is yet another requirement, and a very important requirement in something as complicated as an operating system. In this context, we emphasise the importance of getting the programme right. A programming language can contribute to this aim in two ways. First, the language syntax can be designed to make it difficult to make mistakes – for example, by balancing **lock** and **unlock** instructions just as it balances **begin** and **end**, or alternatively by associating a lock with a single statement and inserting the lock and unlock operations automatically, as in this example from Iliad[IMP1] :

```
locking( log_req, done )
begin
      log_req = yes;
      done = yes;
end;
```

The example is self-explanatory. (Iliad is a language designed for process control systems, where there is typically much concurrent activity, so good synchronisation is essential.)

Second, the language semantics can be designed in such a way as to facilitate reasoning about what happens when programmes are executed. The ideal would be to be able to prove that the programme is a correct representation of its specifications. While this ideal has not yet been achieved, languages such as Euclid and Alphard[IMP2] have gone some way towards making it easier.

An interesting example of an operating system in which these considerations have been taken seriously is Oberon[IMP3], designed by Niklaus Wirth. The system is written using a language (also called Oberon) designed for the purpose. Oberon is a complete operating system comparable to those used in sophisticated personal computers (it is designed for a workstation), and was developed from scratch by two people in three years. They attribute their success largely to the Oberon language.

SPECIFIC LANGUAGE REQUIREMENTS.

We can begin by observing that things happen when the computer computes, and that we want the computer to change things in the world; that's called getting results. The point of that rather obvious remark is to emphasise that we really do want our operating system to be a procedural system which has side effects. That doesn't mean that we can't use non-procedural or functional languages, but it does mean that we can't rule out the "old-fashioned" procedural sort. (It is worth remarking that we can use non-procedural or functional languages only because they've been bent in the interests of getting something to happen.) In practice, though many languages have been used for writing operating systems, procedural languages are by far the most common, and we shall restrict our discussion to languages of that sort. We shall therefore assume that we begin with all the conventional facilities of a language something like Pascal or C, and to match the assumption we change the question a little : what *additional* facilities do we need to write an operating system ?

What can we say about the sorts of programme we shall write with the language ? It is clear from the material we've covered so far that we must cater for many essentially independent threads (by which in this context we mean activities *within* the operating system, not the visible processes seen by someone using the system), which can sometimes communicate with each other, share areas of memory, enforce mutual exclusion, and require certain sorts of synchronisation. These are not facilities built into many conventional programming languages.

Our list does not include mutually dependent threads, and that is a curious omission. We have omitted the topic not because it is unimportant, but because it is widely ignored and, in practice, rarely supported. Many operations we would like to do can be elegantly constructed as sets of dependent threads, each maintaining its own state but all part of the same overall process. A good example is the common requirement to read an input stream, to analyse it, and to deliver single units of the stream on request. Here, the reading, the analysis, and the requesting streams are all interdependent, but each is also a self-contained activity which manages its own buffers and associated items. These were once called coroutines, but the usage is not common now, perhaps because it has rarely been supported in programming languages, which gets us back to where we started. To support this structure, we want our language to have a property which is not found in Pascal or C : the ability to describe coupled activities, each with its own state, cooperating in a task. We can get by in conventional languages by dint of contorted programming with state saved in global variables, or - perhaps better - by using threads within a single programme, but in neither case are we helped by many available languages. We shall bear in mind this desirable feature as we proceed with our discussion.

Having now seen the facilities we want in the system, we should be able to answer the question. Here are some examples, selected from a broad field according to no discernible algorithm; you should be able to see where each of the items fits in.

Data structures :

Tables;
Queues;
Messages;
Semaphores;
Files.

Programme structures :

Concurrent execution;
Synchronisation;
Communication;
Critical sections;
Interrupt handling.

Usability :

Readability;
Provability.

That's what we want – perhaps even what we need. What do we get ?

ASSEMBLY LANGUAGE.

(Yes, we've missed out machine language. There are some things we don't mention. But even machine-language programming had its charms, so effective that a poem on the subject mysteriously wrote itself[IMP4].)

In fact, the omission is fair enough; assemblers came along well before anything which could reasonably be thought of as an operating system was developed, so this is a realistic place to start. Indeed, it's the right place to start, because many operating systems were written in assembly language, and it would not surprise us to find that some still are.

We can be more positive than that. Discussing assembly language has a clarifying effect on our thinking, because it eliminates one question entirely : there is no point in asking whether or not the language can perform some function which we think we want, because it is trivially true that with assembly language you can make the computer do anything of which it is capable. That focuses our ideas on the real function of a language, which is to help us efficiently to write satisfactory programmes.

First, then, does assembly language help us to avoid getting code wrong ? This is a question about the syntax of the language : when we make mistakes, does the assembler find them and tell us about it ? Not very well. At the lowest level, the assembler will of course identify errors in the formation of single lines of code; if we write text which isn't assembly language, we'll be told. Unless it's a very very old assembler, it will also notice references to undefined constants. After that, though, there is little that the assembler can do. By the very nature of assembly languages, there is little chance for the assembler to work out what we're trying to do with some piece of code, so it simply hasn't any information which it can use to identify errors at any higher level than the syntactic. An alternative view of the same difficulty is that almost any sequence of machine instructions is a valid programme, so there is hardly any chance of identifying unsatisfactory code.

If the assembler can't identify errors, is there anything in the language which will help us to see errors we've made ? Again, no : not even enthusiasts for assembly languages argue that their programmes are models of clarity. They (the programmes, and sometimes the enthusiasts) are incomprehensible without a very precise mental model of the processor for which they are designed, and the ability to hold the state of the processor in your mind as you read the programme and to make the correct interpretation of the code is a skill acquired only after much practice.

That's why writing programmes in assembly language
is fun, if you like that sort of thing, and there's a

sense of achievement in constructing a working programme which you don't get from high level languages. Assembly language programming is to high level programming as climbing the sheer rock face is to walking up the mountain by the easy track. (Machine language is specialising in overhangs without ropes.)

Second, does assembly language help us to get code right ? This question is more related to the semantics of the language. Does it provide structures (either control structures or data structures) which we can use to build programmes, and also to organise our thoughts ? Again, no, almost by definition. It is often possible to use macros in assemblers, but then it is usually your responsibility to get them right and to use them appropriately, for no additional checking is provided.

Another way in which a language can help to get programmes right is by being constructed in such a manner that it is easy to prove that the function of a piece of code corresponds to a specification of the required function. Yet again, assembly language fails the test; as we pointed out, the programme means nothing without a model of the processor, and this leads to such complexity that there is little prospect of routine proofs. A high level language – even a procedural language – is much easier to handle, because it uses a much tidier virtual processor.

So far, then, assembly language is not looking too good. Does it match our list of requirements ? In principle, it can do anything, but we've already commented on that line of reasoning. In practical terms, it doesn't fit at all well. Only one item – interrupt handling – is immediately available, and that's only because it happens right down at the hardware level. The language also fails spectacularly to meet our requirements for documentation and safety.

There is a folk tale about IBM's system OS360. (In fact, there are lots of folk tales about OS360, but this is one of them.) OS360 was written in assembly language, and worked most of the time, but it was reputed to have reached an equilibrium state containing a thousand or so errors. The argument was that the system was so difficult to understand that on average each attempt to correct an error introduced another one, and there was of course no mechanism in the assembler which could even notice that an instruction anywhere in the programme didn't make sense.

HIGH-LEVEL LANGUAGES.

Perhaps it becomes clear that assembly languages are less than ideal for writing operating systems. We can summarise the reasons why we need higher level languages :

- To make writing easier;
- To give help in writing correct programmes :
  - by enforcing correctness through syntax;
  - by providing constructs (locks, synchronisation) likely to be useful;
  - by facilitating reasoning about programmes.

We would like to go rather further than that. We suggest that ideally the operating system should be written *completely* in a high-level language; there should be no need to use an assembler anywhere.

The list of characteristics above is far from being a complete specification for a language, and several routes to an effective language have been explored. Here are some of them. We emphasise that this is not intended as a rigorous classification of programming languages, though some languages do seem to fit one or other of the specifications reasonably well. Rather, it is a description of some traits found in

programming languages for operating systems which might be found to some extent in various languages.

Existing language + subroutines :

Most (all ?) high-level languages provide some sort of subroutine. We can use this to disguise the operating system functions as subroutine calls, and use the existing language to describe the required operations which link the system calls together.

This approach provides system facilities, but not much else. There are no special structures, no special syntax, and no special semantics; the result is often a very complicated application programmer interface, because everything must be expressed in terms of the available language constructs. The "subroutines" need lots of information which has to be managed in the programme. The early Primos operating systems for Prime computers were written in Fortran; as Fortran provides no data structures other than arrays, many of the subroutine calls which gave access to the operating system functions required many parameters, and a common error was to omit a parameter or to interchange two. In the Macintosh system, which provides a Pascal interface, the number of parameters is reduced, but at the cost of more or less complicated data structures. (The Macintosh system itself, at least in the early versions, appears to have been written in assembly language; now it's mainly C.) Despite these criticisms, though, the high-level languages are still easier and more comprehensible than an assembly language.

The big advantage of using an existing language is that you begin with lots of ready made support – compilers, libraries, programming techniques, expertise, and so on.

Extended language :

We can hope to get a little, if not the best, of two worlds by taking the background and expertise, and some of the facilities, associated with an existing language and combining them with such special language structures as seem appropriate. To do so, we take the existing language as our model, but add anything convenient. This approach can be very successful, but not transportable. At the least, you have to write your own compiler, though you might be able to make life a little easier by choosing the model language as the compiler's target. (This is often less satisfactory than might appear at first sight. It might be comparatively easy to generate the required code, but it's much harder to link the model language's error reporting and error diagnosis and correcting software with your extensions.)

The Burroughs operating system for their B5500 and B6500 series machines, the MCP, was written in a language of this sort. It was called DCAlgol, for Data Communications Algol, a name which emphasises one of the directions in which the language was extended. The language provided data types for messages, files, and processes, the second and third corresponding closely to the structures we have called the file information block and process control block. New syntax was included to handle these entities, and also to give access to certain specialised hardware operators for string operations such as searching and translating, bit manipulations of various sorts, a test-and-set machine operation called *readlock* as described in the chapter *PROBLEMS OF CONCURRENT PROGRAMMING*, and the linked list operators we mentioned in the chapter *MEMORY MANAGEMENT : THE SYSTEM'S VIEW*. In addition, the semantics of the Algol procedure call was extended to include transfer of control between coroutines, separate execution threads within the same programme, and initiation of new processes. People who became accustomed to using it liked it a lot.

In fairness, it must be added that DCAlgol turned out to be not quite sufficient, and there was also a high-level assembler called Espol (Executive System PrOgramming Language). Later both of these languages were merged into a new high-level language called Newp, of which we have no experience.

High-level assembler :

Languages of this class are designed with only one of our desirable features in mind : to make writing easier. They provide control (and maybe data) structures at a high level – but limited (or no !) checking for abuse. These languages are popular with programmers – "very powerful"; "close to the machine"; "hey, look at this great trick". Because there is no characteristic syntax for significant operations, it is typically possible to produce very obscure code, which is not helpful to others who might have to read it. (We are reminded of a student who participated in one of our courses many years ago, and claimed as one of the good features of the programme which he had written that "the code is not obscured by comments". The difference is that he was joking.)

The high-level assembly languages offer little or no protection against mistakes, and programmes are typically quite unprovable. Most of the time, of course, all is well; programmers who use such languages might be rash, but are not usually stupid, and encode all the required structure correctly. Our objection is that the language offers no help at all in simplifying the programming (by providing appropriate constructs), or in detecting any errors which are made (by identifying them in the syntax analysis). We are human, we are fallible, we make mistakes, and we are very happy if a compiler can tell us that we've made a mistake before the mistake leads to any nasty consequences.

Specially designed language :

Our final category includes the languages which have been constructed from scratch with the intention of implementing our desirable features, or perhaps a slightly different set, depending on the views of the developers. Some examples of such features are :

- Useful data structures (processes, files, messages).
- Control structures appropriate for multiple processes : concurrency, communications, critical regions, etc.
- Policies which facilitate formal treatment : all objects carefully defined, import and export lists, potentially dangerous features (consider **goto** and pointers) abolished or restricted.

We have already mentioned Oberon[IMP3] as one example of a language designed for the job. Another, with stronger emphasis on the formal analysis of systems, is Concurrent Euclid[IMP35]. This is an active research area; languages of this sort are still being developed. We may infer that no one has yet found the ideal solution. Perhaps there isn't one, but it's still worth looking, as every improvement in the languages we use to write our operating systems is likely to result in systems which are more reliable.

WHY NOT HIGH-LEVEL LANGUAGES ?

We have offered arguments in favour of using appropriate high-level languages for writing operating systems; here we present some arguments against. We remark before beginning that we know of no *good* argument against the use of high-level languages. Many of those we have seen can be reduced to something like this assertion : "Because the highish-level language with which I am familiar (or its compiler) doesn't do <something desirable>, it follows that no high-level language (or compiler) could do it". A second sizeable group of objections are equivalent to : "I want to use this demonstrably foolhardy technique, and high-level languages won't let me". We do not believe that either of these arguments has any discernible merit.

Here are a few examples. If you think you have a better one, you are welcome to offer it to us for an opinion. Who knows ? – you might convert us.

Compilers cannot produce efficient code, and efficiency is essential in an operating system.

We concur with the second assertion, but flatly deny the first. A good optimising compiler can produce excellent code, at least up to human standards. It is true that cheap compilers usually generate deplorable code; they cash in on the cheapness of memory and the uncritical attitude of most programmers, nowadays thoroughly trained to ignore considerations of efficiency.

Operating systems might need to refer to specific memory addresses, for interrupt handling and input-output operations.

> So ? It is true that few high-level languages provide such facilities, but that's because the work for which they were designed doesn't need them. The point of a high-level language isn't that it necessarily hides everything to do with the computer; its job is to provide you with the things you need, and then to look after the details of handling them safely and reliably. If specific memory addresses must be used, then we should work out just how they are used, and provide for the requirements in the language. We will accept this argument as suggesting that extended or specially designed languages are potentially better suited to the task than existing languages. (We will also accept it as an excuse for not using a high-level language in a specific project, because you don't usually have time to construct a new language and a compiler in a project, but that isn't an argument against high-level languages in principle.)

Languages designed to produce programmes which are easy (or at least easier) to prove correct usually prohibit such constructs as ***goto****s and pointers. These are essential for writing an operating system.*

> Nonsense. If you know what you're doing, you should be able to specify it clearly enough to be implemented by a compiler in a controlled and checkable way. Of course, this argument doesn't carry much weight if you don't know what you're doing.

C.

In any discussion of operating system programming languages, C[IMP5] must be mentioned. It was developed as a vehicle for the Unix operating system, and is undoubtedly the most influential of our examples. It is rather unfortunate that its strongest characteristic is that of the high-level assembler, which we have roundly condemned. It is undoubtedly powerful, and can be used to good effect; but the C syntax seems peculiarly well adapted to obfuscation and misinterpretation, and the power is just as effective in wrong programmes as in correct ones.

C, and its descendant C++[IMP6], will undoubtedly be with us for a long time. The appearance of ANSI standard C has gone some way to taming some of the worst excesses, but the language cannot yet be said to have been domesticated. We therefore think it appropriate to add one more prop to our hate campaign.

Suppose you were a civil engineer, considering a new technique for building bridges. It is claimed that the new technique works very well in experienced hands almost all the time. It does not introduce any new and precise methods, but gives its operator control over a large number of hammers and saws, working very quickly. It is recommended that the operator should work from a plan, but there is no way to tell whether the completed bridge follows the plan.

We hope that you, the civil engineer, would not adopt that technique. It is true that the engineer probably has a rather wider choice of reliable design methods than does the operating system programmer, but it is also true that today's average programmer has less concern for the reliability of the product than does the engineer. C, like other languages which share the same characteristics, is a consequence of this irresponsible attitude. For building well constructed and guaranteed reliable software, C is not the language of choice.

Why is C++ called C++ ? In C, that means "Use C, and increase it afterwards". C++ is C which has already been increased in some sense, so ++C is a better name. We pass no comment on the further implication that it gets better every time you use it.

In practice, most operating systems seem to be written in C – or, nowadays, C++. We get the operating systems we deserve.

A LITERARY CURIOSITY.

It is an odd fact that the name C was associated with a language long before the programming language was invented; unless the inventors of the programming language were familiar with George Orwell's work, it must be a coincidence. Certainly there is no indication in any of the programming language literature we've seen that any connection with Orwell's work is intended. The official derivation of the name C for the programming language is from an earlier language called B, and a development therefrom called BCPL; it's plausible enough. The literary C is noteworthy for being very rigidly defined, limited to a small circle of speakers, and gibberish to anyone else.

> *"**The C vocabulary.** The C vocabulary was supplementary to the others and consisted entirely of scientific and technical terms in use to-day, and were constructed from the same roots, but the usual care was taken to define them rigidly and strip them of undesirable meanings. They followed the same grammatical rules as the words in the other two vocabularies. Very few of the C words had any currency either in everyday speech or in political speech. Any scientific worker or technician could find all the words he needed in the list devoted to his own speciality, but he seldom had more than a smattering of the words occurring in the other lists. Only a very few words were common to all lists, and there was no vocabulary expressing the function of Science as a habit of mind, or a method of thought, irrespective of its particular branches. There was, indeed, no word for 'Science', any meaning that it could possibly bear being already covered by the word **Ingsoc**."*

For the full story, read Nineteen Eighty Four[IMP7]*, which is about thought control in a totalitarian state; the passage above is taken from the appendix on "The principles of Newspeak". A major tool in this control is the language Newspeak, derived from English by eliminating all means of formulating statements which are not politically correct. This is exactly what we want to do in programming languages, but the book shows very clearly that the attempt to impose political correctness by human language engineering is very dangerous.*

COMPARE :

Lane and Mooney[INT3] : Appendix A, sections 8 and 9.

REFERENCES.

IMP1 : H.A. Schutz : "On the design of a language for programming real-time concurrent processes", *IEEE Transactions on Software Engineering* **5**, 248 (1979)

IMP2 : P. Levy, S. Hanson, P. Jackson, R. Jullig, T. Pittman : "Summary of the characteristics of several 'modern' programming languages", *Sigplan Notices* **14#5**, 54 (May, 1979)

IMP3 : N. Wirth : "A plea for lean software", *IEEE Computer* **28#2**, 64 (February, 1995).

IMP4 : E. Nather : *The Story of Mel, a Real Programmer*, http://www.datamation.com/PlugIn/humor/jargon/jargon_48.html (May 21, 1983).

IMP5 : B.W. Kernighan, D.M. Ritchie : *The C programming language* (Prentice-Hall, 1978).

IMP6 : B. Stroustrup : *The C++ programming language* (Addison-Wesley, 2nd ed., 1991).

IMP7 : G. Orwell : *Nineteen Eighty Four* (Penguin Books, 1954).

IMP35 : R.C. Holt : *Concurrent Euclid, the Unix system, and Tunis* (Addison-Wesley, 1983).