SYSTEM ARCHITECTURE.

The earliest systems were implemented as single programmes which did everything. That didn't mean that the whole programme had to be in memory at the same time, of course; by taking advantage of overlay techniques and, later, virtual memory, the memory demands could be kept within reasonable bounds. It did mean that necessary communications could be handled rather easily, as the whole programme shared the same address space, and had access to the same global variables.

Alternatively, it might be better to regard some of the early systems not so much as single programmes but as collections of programme fragments. The operating system was not thought of as a "real" programme, and was therefore constructed in any way which seemed to work – an operation greatly facilitated by the universal use of assembly language. Useful collections of data could therefore be planted at convenient places in memory (usually at the ends), protected – if at all – by appropriate values of memory limit variables, and simply used as required by code implementing the various system functions.

This view had one or two rather curious consequences. By emphasising the difference in kind between the operating system and the programmes executed under its supervision, artificial boundaries were created. The consequence in at least one system (the IBM 1130 Monitor system) was the existence of two procedure libraries, essentially identical, one for the operating system and one for the other programmes. Another byproduct was a continuing debate over what was properly considered part of the operating system, and what was not. Clearly, code intended to load and execute a programme was part of the operating system, while a compiler, which dealt with a high-level language, was not. But what about a compiler which produced a non-standard code file, and therefore had to provide facilities for loading and executing it ? And what about overlay procedures ? – they did a very systemy job, but were clearly built into your programme. It was all very puzzling.

From these debates there developed a rough hierarchy of software. While it has become less significant with the passing years, it is still interesting as illustrating a "scale of systemness".

System software : the operating system programmes themselves.

Utilities : separate programmes, but closely connected with the system – ways of organising files on the disc, or listing files, or checking devices, or other such, typically administrative, actions. These were commonly, though not invariably, provided by the system manufacturer.

Packages : separate programmes which you bought to do things not primarily connected with the system – compilers, payroll packages, simulation and modelling packages, etc. And, much later, databases.

Your programmes : unambiguously nothing to do with the system. (Unless, of course, you were a systems programmer, in which case anything was possible.)

This worked reasonably well so long as systems only handled one programme at a time, so that activities could be arbitrarily divided out between different participants in the computation. Multiprogramming blurred it rather badly by splitting the system in two. Now there were *real* system functions, each of which appeared only once, such as the memory manager and the processor scheduler; but there were others which at least seemed to appear once for each process, as they were required to handle different data from different processes simultaneously, but quite independently. Examples are input and output operations. These operations posed real problems with early hardware, particularly before the development of automatic address translation methods. Another complication was the limited appreciation of the advantages of separating data from code in memory. A procedure would always use its own local memory, so the code could not be simultaneously executed by more than one process, even if the idea had been current. (This wasn't stupidity; keeping code and data close together meant that you could use short instructions, which saved valuable memory space.) Finally, and following from the admixture of code and data, many early computers implemented subroutine entry by planting the return address at some standard point within the called routine's code

(commonly at or immediately before the target address for the jump), so a second entry to the subroutine before the first had been cleared could destroy the return information for the first. These complications led to the idea of the *serially reusable* routine, which could be used by anyone, provided that you made certain no one else was currently using it before attempting to enter.

From such thoughts about the diversity of material included in the operating system there developed the idea of the *kernel.* The kernel is just that part of the operating system without which nothing else can happen. Its precise contents are to some extent a matter of convenience, but it will certainly include primitive memory management and processor management, and possibly some of the disc manager. Once the kernel is defined, the rest of the system can be implemented as something which looks very like a set of utilities – separate programmes, each handling some aspect of the system operation. The best known example of a system of this sort is undoubtedly Unix, which has a rather small kernel which looks after the basic functions, and a set of programmes which execute the various instructions. The shell (a separate programme) reads an instruction (usually) from the terminal, analyses it, and (except for a few simple operations which are built into the shell) runs another programme to execute it. To make this work, there must be provision for adequate communication between the different programmes; so Unix has signals, pipes, and programmes with parameters, and a discipline for executing multiple processes which ensures that the shell will take over again when the other process ends.

In terms of the implementation of the system, the change is not very significant; some procedure calls (not at the level of system calls) have been replaced by programme executions, or – rather more efficiently – by communication between the kernel and other service programmes which are kept ready to carry out the required tasks. For example, instead of entering an operating system subroutine to copy a file, the system will execute a programme which copies a file. In both cases, certain parameters must be passed, and results might be returned, but workable, if not necessarily elegant, solutions to these problems were devised. For the ready-to-run service programmes, this introduces the idea of communications between programmes; while not new, it was perhaps at around this stage that such communications began to be seen as an important part of the operating system in their own right.

For a while, there was much interest in reducing the size of the kernel as far as possible, giving what is sometimes called a *microkernel*. This has great advantages if your aim is to build flexible and adaptable systems, and can even be used as a means of supporting different operating systems at the same time. Its main drawback is its dependence on much communication between programmes[HIS9]. Even with efficient message-passing techniques, the overhead can be significant, and as operating systems are required to deal with faster and faster streams of data, it becomes a severe problem. (The data management problem is severe in any case, but the additional complication of ensuring safe passage of information *between* programmes, as opposed to movement *within* a single programme, is a significant additional load.) At the present, microkernel systems are not favoured for work requiring very high-speed data movements, and are therefore rather unfashionable.

Once the idea of the monolithic operating system gives way to the kernel with specialist programmes for different functions, we can go a step further in the same direction. What's special about operating system functions ? Why not use the same mechanism for any sort of service ? We've been here before – this is the client-server model again. Programmes which require service – the clients – call upon specialist programmes – servers – which do the required work and (all being well) deliver the service.

Within the operating system, the implications of this idea are limited, because, as we remarked, it is close to a description of what was going on anyway. Perhaps the main consequence was to tidy up ideas of what went on in the system, and encourage people to think more in terms of separating the functions of the system where possible. For more general use, a better means of communicating between client and server was required, and eventually settled down to the exchange of messages according to some fairly standardised protocol. This development has been in progress over the last few years, and has coincided with the spread of computer networks. Perhaps not accidentally, the communications method and the networks have turned out to be mutually supportive, and

client-server systems for many purposes now run on local and more widespread networks, so that a programme running on one computer can routinely connect to a server running on a different machine.

The view of an operating system which results from these developments shows a number of largely autonomous specialist programmes, each with a well defined function, and communicating with each other by exchanging messages. This sounds very like object-oriented programming. For this reason, there is much interest in using the object-oriented view as a basis for the design of operating systems, in the (plausibly realistic) hope that the adoption of such a basic design principle will make it easy to extend systems by adding new objects when required. The natural extension of message-based communications to networks is also attractive, as it is clear that networked systems are here to stay.

It is less clear that we know how to control the use of networks as yet. Even though it might be easy to ignore the boundaries between machines in some cases, a number of computers connected by a communications network is still just that. With the possible exceptions of some experimental systems, there is little sense in which the whole can be operated as a single combined system.

REFERENCE.

HIS9 : W.H. Cheung, A.H.S. Loong : "Exploring issues of operating systems structuring : from microkernel to extensible systems", *Operating Systems Review* **29#4**, 4-16 (October, 1995).