Computer Operating Systems

For micros, minis and mainframes

DAVID BARRON

CHAPMAN AND HALL COMPUTING

Computer Operating Systems

FOR MICROS, MINIS AND MAINFRAMES

David Barron

Professor of Computer Studies
University of Southampton

SECOND EDITION

LONDON NEW YORK
Chapman and Hall

First published 1971 by
Chapman and Hall Ltd
11 New Fetter Lane, London EC4P 4EE
Reprinted 1973, 1975
Second edition 1984
Published in the USA by
Chapman and Hall
733 Third Avenue, New York NY 10017

Printed in Great Britain by
J.W. Arrowsmith Ltd., Bristol

ISBN 0 412 15620 2 (hardback)
ISBN 0 412 15630 X (paperback)

British Library Cataloguing in Publication Data

Barron, D.W.
Computer operating systems.—2nd ed.—
(Chapman and Hall computing)
1. Operating systems (Computers)
I. Title
001.64'25 QA76.6

ISBN 0–412–15620–2
ISBN 0–412–15630–X Pbk

Library of Congress Cataloging in Publication Data

Barron, D. W. (David William), 1935–
Computer operating systems.

Includes index.
1. Operating systems (Computers) I. Title.
QA76.6.B37 1984 001.64'2 84–9537
ISBN 0–412–15620–2
ISBN 0–412–15630–X (pbk.)

Preface

This book is based on my earlier text, *Computer Operating Systems*, written in 1970. In the preface to that book I remarked that '. . .up to the present, development (of operating systems) has been of a largely *ad hoc* and pragmatic nature, and has been aptly characterized as "moderate success at enormous expense" '. Thirteen years later the picture is not much different so far as mainframe systems are concerned. Thousands of man-years have gone into the development of systems like IBM's VMS and ICL's VME, producing systems that occupy prodigious amounts of memory, soak up processor cycles, and require an army of systems programmers to 'maintain' them. Truly it has been said that, 'an elephant is a mouse with an IBM operating system'. While this investment means that systems of this kind will be with us for many years to come, they are in some way the end of the line, and will come to be seen as an evolutionary dead-end.

Looking at these over-large and excessively complicated systems, one cannot help feeling that there must be a better way of doing things, and driven by the very different requirements of small machines a new breed of operating system has developed for mini- and microcomputers, quite unlike the traditional variety. The emphasis in this book is largely on these new operating systems, in keeping with my view that the mainframe has had its day, and that the future lies with the micro. However, the older systems are by no means ignored since there is much to be learned from a study of the past, in computing as in any other subject.

With an increasing understanding of operating systems we see more clearly their underlying structure, and this text is more concerned with the way operating systems can now be constructed than with the sordid details of the internal economy of the older systems. Moreover, it is not a theoretical text: it remains sufficiently close to the realities of modern-day operating systems to give the reader an appreciation of why things are as they are, as well as how they might be.

Whatever kind of computer system is used, an operating system i
likely to loom large in the user's view. Thus, not only students an
computer specialists, but also anyone interested in how compute
systems work, needs to know about operating systems. It is my aim t
convey 'what everyone in computing should know about operatin
systems'. I hope that Computer Operating Systems will be of use i
introductory courses, and will also be of interest to all computer users
including the growing army of amateurs who find in computing a
absorbing hobby.

David Barron

Contents

1
Some background

1.1 WHAT THE OPERATING SYSTEM DOES

Almost without exception, every computer has an operating system. The reasons for having one, and the particular kind of system, are very different on different varieties of machine, but the operating system is always there. Indeed, to most users the operating system *is* the machine. They never see a 'raw' machine: instead they see and use the interface presented by the operating system.

A 'raw' machine is a most inhospitable device. In order to do anything, it requires a program in a specific binary representation of the particular machine code. In particular it needs more-or-less complicated programs to drive its peripherals before it can even communicate with the outside world. It is therefore usual to provide *system software* to make the system usable. In a home computer this system software is an integral part of the system, contained in a read-only memory (ROM) and automatically entered when the system is switched on. Thus the user does not distinguish between the CPU (central processing unit) and the operating system: they combine to provide a programming environment in which programs (usually in BASIC) can be run, saved on cassette tape and later reloaded. In large personal computers (e.g. the IBM PC) and in minicomputers and mainframes we can identify a number of individual pieces of system software, mostly concerned with program preparation – editors, assemblers, compilers, link-editors, etc. Pervading all these is the operating system. It has been described as the 'glue' which holds all the other components together. Alternatively we can view it as providing an infrastructure: a common environment in which the various system programs can operate, including the mechanisms for communicating with the peripheral devices. (From the early days, some of the complexity of controlling input/output devices had been concealed from the user by the provision of a software package called the Input–Output Control System (IOCS), and with the trend in early computers to have I/O functions controlled partly by hardware and partly by software, a

natural development was to incorporate the IOCS into the operating system.)

This is a relatively modern view of operating systems, which were originally developed in response to a quite different need – to maximize the utilization of the central processor and peripheral devices. We must remember that the early computers were phenomenally expensive, and so could not be allowed to be idle. It was therefore necessary to automate the flow of work, and bring decisions on the management of system resources on to the time-scale of the computer rather than that of the human operator. Falling hardware costs removed the preoccupation with 100% processor utilization, but at the same time the typical mode of usage of computers changed, with more emphasis being placed on simultaneous access by a large number of users. Thus the operating system now had to manage the sharing of resources among the users.

As the need to optimize utilization has diminshed, another aspect of the operating system has become of increasing significance: the software interface. Increasing complexity of computer systems led to further functions being incorporated in the operating system to aid the user – for example, organizing disc storage so that it appears as a logical file-structure to the user, unconstrained by the physical organization of tracks, sectors and blocks. The operating system can thus have two distinct facets: on the one hand controlling the allocation and utilization of shared resources, and on the other hand providing an interface to the hardware more convenient and amenable than that presented by the naked machine. This duality of purpose is characteristic of any multi-user operating system, and we will find the same structure in the operating systems both of mainframes and of multi-user minis.

We can thus define the function of an operating system as being to provide a convenient environment for the user(s) of a computer system, whatever their requirements – program development, real-time process control, database management, payroll or whatever. In the case of a personal computer, there is only one user, and the sole function of the operating system is to implement the user-interface. In the case of a multi-user system the operating system must share the resources so as to provide the user-interface to each of the users. Instead of optimizing the utilization of the processor, the aim is to optimize the productivity of the user.

The provision by the operating system of a convenient software environment has had a dramatic effect on the development of personal computers. In the heyday of mainframes and minis, the software was provided by the manufacturer and (to a lesser extent) by the installation's systems programmers, and applications programs were not expected to be portable between machines. In the microcomputer world, how-

ever, the situation is different. Software is sold 'off the shelf', and users shop around for the package that best suits their requirements. The situation was bedevilled by incompatible file formats and programming systems until an operating system called CP/M became the *de facto* standard for 8-bit micros, ensuring that programs written to the CP/M interface could be run on most microcomputers.

1.2 KERNEL AND SUPERSTRUCTURE

In discussing operating systems it is convenient to separate those parts which provide the user environment from those parts more directly concerned with sharing resources and interfacing to the machine hardware, such as interrupt handling, storage management, processor scheduling, etc. This latter area we will describe as the *kernel*, and the remainder we call the *superstructure*. On this definition the superstructure comprises those parts of the operating system that provide the basis of services to the user (filing system, command language, etc.), but does not include system software such as editors (which we regard as applications programs). Many older systems include the editor as part of the operating system, but this is not a good feature, since it makes it impossible for the user to take advantage of new developments – he is stuck with the system editor.

The distinction between kernel and superstructure is a particularly useful one to make. It allows the description of a general-purpose operating system to be divided into two parts with a clean interface – as long as one understands *what* the kernel does, one can follow an explanation of the superstructure without needing to know *how* the kernel does its job. Also, the distinction allows us to categorize neatly the operating systems of smaller machines. Thus many minis have an operating system (typically described as a 'resource-sharing executive') which consists almost entirely of kernel, with a very minimal superstructure. DEC's RSX11-M is a typical example of this category of operating system. Personal computers, on the other hand, have a rudimentary operating system that is almost entirely superstructure – since most resources are dedicated to a single purpose there is little for the kernel to do apart from interrupt handling. (What we have just said about single-user systems applies strictly only to single-user single-process systems, in which the single user can do only one thing at once. More advanced single-user systems (e.g. Concurrent CP/M-86) allow the user to set up a number of tasks to run in parallel. In such a system resource allocation once more becomes a necessary function of the operating system.)

1.3 DATA MANAGEMENT AND JOB CONTROL

The major facilities offered by the superstructure are data management and job control. An important part of data managment is the control of the input–output devices. However, there is more to data management than this. The operating system must map logical file structures on to physical devices, and provide *access methods* whereby particular records of a file can be accessed without having to specify at user level the details of file structure and disc layout. If information in files is subject to frequent amendment the system must also take necessary steps to preserve the integrity of the stored information. The operating system can also provide a degree of device-independence, so that the user can see the same logical file structure, independent of the particular devices used for storing his files.

Job control is concerned with executing the user's programs. For a single-user system, or a multi-user on-line terminal system, this requires a command interpreter which accepts commands from the keyboard and performs the operations requested. Job control becomes much more complex in a 'batch' system where users submit jobs to be run without intervention. (For this purpose we define a job as the basic independent unit of work submitted to the machine, i.e. a unit that is neither connected with any other unit of work, nor dependent on any other unit for its completion.) A job will frequently (in fact usually) be composed of job-steps, e.g. compile, link-edit, run. The system must allow the user to link together any number of operations as job-steps, providing the right environment and resources for each step and organizing the passing of files from one job-step to another. It must also be possible to make execution a job-step conditional on the successful running of earlier steps, and even to specify alternative sequences dependent on the outcome of earlier job-steps. For full generality it is desirable for the user to be able to specify steps that can be performed simultaneously, and to be able to synchronize such parallel execution.

1.4 CATEGORIES OF OPERATING SYSTEM

In conclusion we define a few terms that are used to categorize operating systems.

Single user and *multi-user* are self-explanatory.

A *multi-programming* system is one in which the available processor(s) is shared (by the operating system) amongst several programs co-resident in main memory, in order to improve CPU utilization.

A *foreground–background* system is a single-user sytem in which two programs are multi-programmed. One (the foreground) interacts with

the terminal and runs as long as it is able; the background program is assigned to the processor whenever the foreground program is unable to proceed (e.g. because it is waiting for the user to do something). As soon as the foreground job is free to proceed it does so. A foreground–background system is a particular case of a single-user multi-process system.

A *concurrent* or *multi-tasking* system is a generalization of a foreground–background system in which the user of a single-user system can initiate a number of concurrent jobs, assigning the terminal display to any one of the jobs as required. (But note that older texts use the term 'multi-tasking' with the meaning here ascribed to multi-programming. Computer terminology is a minefield.)

A *time-sharing* system shares the processor and memory amongst a number of programs, each associated with a remote interactive terminal, in such a way that each user thinks he has a machine to himself.

A *transaction-processing* system (TP system) resembles a time-sharing system in that it serves a number of remote terminals. However, whereas in a time-sharing system each remote user is associated with a different program, and is totally independent of the other terminals and programs, in a TP system all the terminals are connected to the same program (or suite of programs). This program accepts *transactions* from its terminals, processes them and sends responses. The classic example of a TP system is an airline reservation system.

Finally, a *general-purpose* system is a multi-user system that combines batch processing, time-sharing and possibly transaction processing in a single (usually large and complex) system.

2
Mainly historical

2.1. THE USE OF HISTORY

In this chapter we trace the development of the 'traditional' batch and time-sharing systems. In addition to giving an insight into systems that are widely used at the present time, this historical survey introduces many of the concepts that came together to form the basis for design of modern operating systems. The technology of computing changes rapidly, and today's innovation is tomorrow's museum piece. In many parts of the subject it is best to dismiss the history and launch straight into current technology, but in operating systems we can learn by following the historical development. Seeing this in a modern context provides a good basis for understanding the complexities of modern operating systems – today's Concurrent CP-M/86 has a lot in common with yesterday's multi-programming executive.

2.2. THE EARLY BATCH SYSTEMS

In the early days of computing, the machines were 'hand-operated'. That is to say, the operator (in those days often the programmer) set up a job by loading the card-reader, mounting magnetic tapes, etc., and then started the program by manipulating switches on the console. If the program called for operator intervention the operator took appropriate action and restarted the program. Finally, when the job was finished, the operator dismounted the tapes, unloaded the card reader, removed the listing from the printer, and then started setting up the next job. Given the capital cost of a computer in those days, such a method of working was acceptable only so long as the set-up time was insignificant in comparison with the run-time of the job. The early computers were so slow that this was usually the case, but as computer speeds increased, the ratio of set-up time to run-time grew to unacceptable proportions, and the need arose to automate the job-to-job transition. The increase in processing speed also highlighted the disparity between the speed of

operation of the processor and that of the input–output devices. Efforts to remove this mis-match led to two developments. The first was the introduction of the *I/O channel*, which was a piece of hardware to control I/O devices in an autonomous manner. Once started, the channel ran independently of the central processor, thus allowing I/O to be overlapped with computing. The program could initiate a transfer and at a later time interrogate the channel to determine whether the transfer had been completed. The second development, of greater significance to the development of operating systems, was the introduction of the technique of 'off-lining' I/O. Instead of the computer using the slow peripheral devices directly, input was transcribed from cards to magnetic tape, and the program got its input by reading card-images from the tape. Similarly, output was written as card-images or line-images to tape, and these were later transcribed to card punch or printer as appropriate. The off-line transcription to and from tape was initially done by special-purpose hardware, but it was soon found that it was more economic to use a small computer as a 'satellite'. (The large IBM 7090 scientific computer was rarely to be found without an attendant 1401, a small data-processing machine, as a satellite handling its I/O.) Once the principle of off-lining I/O to tape was established, the way was open for the development of an automated job-sequencing system. All that was required was to record a number of jobs (a *batch*) on tape and arrange that instead of the machine stopping at the end of a program, control should revert to an operating system (or *monitor program*) that immediately started the next job. (This is somewhat of an oversimplification, but will suffice for the present.) Probably the first such system was SOS, the Share Operating System devised by the association of users of IBM machines called SHARE. SOS was the forerunner of the classic Fortran Monitor System, FMS. Since such systems executed a batch of jobs on a magnetic tape in automatic sequence, they became generally known as *batch systems*.

This kind of system had two main attributes: it automated the sequencing of jobs, and it fooled programs into thinking that they had a real card-reader and line-printer when in fact their I/O was being off-lined. This latter did not present much difficulty: I/O via an autonomous channel was so complicated that it was in any case carried out by a package of system routines called the I/O Control System (IOCS). The existence of the off-lining process was easily concealed by modifying the appropriate IOCS routines. Automating the job sequencing had more substantial implications. It was necessary to ensure that the monitor was entered whenever a program terminated, natually or by reason of error. Thus FMS programmers were instructed never to use STOP but to terminate execution by CALL EXIT. (To this day some FORTRAN systems use

CALL EXIT, though few of the users or implementors appreciate the reason.) Important facts to note at this stage are:

(1) The monitor program must reside in memory, and will make use of the processor whenever it runs. This is an example of the *overheads* that arise from the use of an operating system. When assessing operating systems the magnitude of such overheads is a major consideration.

(2) We have to ensure that the control program is not overwritten by the user program. We also have to ensure that a user program never STOPs, and does all its I/O via the routines of the IOCS. In the early days it was necessary to rely on programmer discipline, but later systems had assistance from the hardware. For safety it is necessary to have such hardware assistance: the minimal requirements are some sort of storage protection, to deal with the overwriting problem, and a distinction between normal mode working, in which some operations cause a trap to the monitor system, and privileged mode working in which anything goes, so that attempts by the program to STOP or initiate I/O activities can be intercepted.

(3) The system must be resilient against badly formed input decks and faulty programs. Thus an incomplete program must not result in the compiler consuming the data cards (or, worse, the next program), and a program must not be allowed to read past the end of its own data. The user must therefore provide *control information* to delimit his program and data. This will take the form of *control cards*, and it will be necessary to have some convention to distinguish these cards. A common convention is to put a special character (e.g. a dollar sign) in the first column. This means that no program or data card can start with this character – an example of the *constraints* that the operating system imposes on the user.

(4) There must be a means of communication between the compiler(s) and the operating system in order to signal compiling errors and inhibit the subsequent running of the job. Some systems dodge this issue by making the compiler(s) part of the operating system. This is a reprehensible practice, since it makes it difficult to incorporate new compilers, and so restricts the user to the designer's choice of language(s).

2.3 SPOOLING SYSTEMS

For a while, systems of the type just outlined reigned supreme. The next development was triggered by the arrival of a new hardware technique, *the interrupt*. The difficulty of synchronizing an autonomous I/O channel has already been noted. The interrupt obviates this difficulty by having

the channel signal the computer when it has finished its job, or when an error condition arises that requires program intervention. When an interrupt occurs the hardware preserves the current state (register contents) and enters an *interrupt routine* to deal with the channel. When the channel has been serviced the interrupted program can be resumed, the hardware restoring the register contents before returning control. The whole process is transparent to the interrupted program, which has no knowledge of the interrupt. It is thus possible for a program to keep peripheral devices running at full speed without having to keep a constant check on the process of the I/O transfers. (It is of interest to note that the interrupt is probably the first computing concept not to have been anticipated by Babbage.)

Initially, this technique was used within a single program, but it required skilled programming and it was soon appreciated that better utilization of equipment could be obtained by the technique of *multi-programming*. This is the technique of having several programs simultaneously in memory, so that whenever one program is waiting for a peripheral device another will be able to use the processor. Multi-programming is particularly effective if programs that are peripheral-limited (in the sense that their execution time is determined mainly by the time taken doing input and output) are run at the same time as programs that are processor-intensive. The first computer to introduce this concept was the Ferranti Orion; multi-programming was fully exploited in the Ferranti (later ICL) Atlas machine.

The concept of multi-programming had an important, and perhaps unexpected, application in operating system design. In a classic FMS-type system the satellite computer is peripheral-limited, while the large computer is for the most part processor-limited. There is thus a potential advantage in dispensing with the satellite and instead using a single computer to perform both jobs by means of multi-programming. Since there is no longer a satellite computer there is of course no physical transfer of a batch of jobs. The earliest systems of this kind used tape as a backing store, and thus still processed jobs in batches. With the advent of disc storage the system became a continuous flow or *job-stream* process, but the terminology 'batch-system' still persists.

Systems of this kind are called *spooling systems*. The acronym SPOOL, derived from 'Simultaneous Peripheral Operation On-Line' was coined within IBM, though the technique pre-dates the acronym, being known within Ferranti as 'pseudo-off-line input–output'. The technique of spooling was invented at Manchester University as part of the Atlas Supervisor (which incorporated many other innovations.) That spooling system was tape-based – a triumph of ingenuity over technology – but all subsequent spooling systems have been disc-based. A

spectacularly successful early spooling system was IBM's HASP.

A spooling system involves three simultaneous activities: reading card images to disc, running jobs from disc with output to disc, and transcribing line images from disc to printer. The operating system evidently becomes more complex since in addition to all its previous functions it has to simulate concurrent activity with a single processor and memory. The description of the operating system is simplified if we recognize that this simulation of concurrency is a largely self-contained activity, and encapsulate it in a *multi-programming executive*. The benefit of this is that the executive–hardware combination defines an abstract 'machine' that is capable (apparently) of sustaining concurrent activities. We can therefore describe the spooling system in terms of this 'machine' without worrying how concurrency is achieved. This is an example of the 'separation of concerns' that is so important a part of structured design.

2.3.1 **The multi-programming executive**

We start by describing a simple executive that is capable of running a fixed number of programs in a pseudo-concurrent manner. The first thing to note is that the programs and the executive must share the memory, each being allocated a contiguous area of appropriate size. We say that the memory is *partitioned*. We have to ensure that a program in one partition cannot affect a program in any other partition. Although it is sometimes necessary to try to enforce this by software, for complete safety we require hardware assistance in the form of *memory protection* or *memory mapping*. We shall examine possible protection and mapping techniques later.

In order to implement multi-programming the hardware must provide a *real-time clock* or *interval timer*. This is a device that generates an interrupt after a preset interval: since such clocks are usually driven by the mains frequency the interval will be a multiple of 1/50 second (1/60 second in the USA). If the programs being multi-programmed were completely independent of each other and of external events, the executive would be trivial. We would set the clock to a suitable interval, say 1/10 second, and use the interrupt to switch from one program to the next, so that the programs are run in turn for 1/10 second each. The quickness of the hand deceives the eye and the programs appear to run in parallel.

However, a realistic system is not so simple. Although the programs will be largely independent, they will need to communicate with each other from time to time, and if they are using peripheral devices they will need to communicate with those devices. Thus a program must bs able to make requests to the executive. This is usually done by a special machine instruction called a *trap* or *supervisor call* (SVC). This transfers control to a