Operating Systems

A PRACTICAL APPROACH

ROBERT SWITZER

Operating Systems

A practical approach

Robert Switzer

PRENTICE HALL

New York London Toronto Sydney Tokyo Singapore

First published 1993 by
Prentice Hall International (UK) Ltd
Campus 400, Maylands Avenue
Hemel Hempstead
Hertfordshire, HP2 7EZ
A division of
Simon & Schuster International Group

Printed in Great Britain by Redwood Books,
Trowbridge, Wiltshire

Library of Congress Cataloging-in-Publication Data

Switzer, Robert.
Operating systems : a practical approach / Robert Switzer.
p. cm.
Includes index.
ISBN 0-13-640152-X
1. Operating systems (Computers) I. Title.
QA76.76.063S96 1993
005.4'3--dc20 92-27773
CIP

British Library Cataloguing in Publication Data

A catalogue record for this book is available from
the British Library

ISBN 0-13-640152-X

2 3 4 5 96 95 94 93

Operating Systems

Preface

This book is the direct result of a course of lectures I gave at the University of Göttingen in the Winter Semester 1990/91. The course was accompanied by a practical session in which the students had an opportunity to experiment with the operating system TUNIX. TUNIX was developed especially for this purpose and will be presented in this book. One of the chief aims in preparing the course was to have a good balance between the theoretical and the practical aspects of the subject.

The course was designed to be an intermediate to advanced course. Thus the students were assumed to be familiar with topics dealt with in lower level computer science courses such as elementary data structures like linked lists or hash tables or elementary aspects of concurrency such as critical sections and semaphores. The same assumptions are made about a reader of this book. If he or she is not familiar with some of these topics, it might be a good idea to consult another textbook such as *Introduction to Algorithms* by Cormen, Leiserson and Rivest for the data structures or *Principles of Concurrent and Distributed Programming* by Ben-Ari for concurrency.

Another question likely to be raised here is that of programming experience. The sample code given in this book is a sort of pseudocode; but it is a pseudocode that has a strong resemblance to C. Thus it would certainly be helpful although probably not necessary to have a certain familiarity with the language C.

More important perhaps is a modest acquaintance with the programmer's interface to an operating system – the system calls. The interfaces of modern operating systems today have a marked resemblance to one another, but the one that features mostly here is that of UNIX. For more information about this interface the reader is referred to the book *Advanced UNIX Programming* by Marc J. Rochkind.

Very few books are the product solely of the ingenuity and originality of their authors and this one is certainly no exception. I wish to make quite clear here what my chief sources of knowledge and inspiration were for the course and the book.

The most important source was without doubt the excellent book *The De-*

sign of the UNIX Operating System by Maurice J. Bach. I learned more about operating systems from reading that one book than from any other source – even though the book only treats one very concrete operating system. Bach's book confirmed my conviction that operating systems is a subject that should be taught with a strong practical component. That is, students should have a *real* operating system with which they can experiment and see how these complex programs are put together, and what influence the choice of different algorithms can have on the performance of the system.

One of the things I particularly admire in Bach's book is the skilful use of pseudocode. Bach has developed a style of hiding the confusing details and presenting the essential ideas. I hope I have succeeded in imitating his style here.

The second book I wish to mention also strongly supports the philosophy that a course on operating systems should have roughly equal portions of theory and practice: *Operating Systems: Design and Implementation* by Andrew S. Tanenbaum. In his book Tanenbaum develops and explains an operating system called MINIX, which he developed for instructional purposes.

While I am praising books let me recommend to the reader a third one: *Operating System Concepts* by James L. Peterson and Abraham Silberschatz. It is largely theoretical but it has a good survey of the history of operating systems which goes far beyond the sketch I give in Chapter 1.

Why did I think it necessary to write another book on operating systems and implement another experimental operating system? Why didn't I use Tanenbaum's book for my course and MINIX for the practical session? Let me explain.

One of the major design decisions Tanenbaum made before developing MINIX was that it should run on a PC so that students could use it at home. The fact that MINIX is very popular shows that Tanenbaum's decision met a genuine need. However, that decision also crippled MINIX in two serious respects as a system to be used for instruction and experiment:

- It meant that some very important aspects of modern operating systems could not be implemented in MINIX. In particular MINIX could not have virtual memory, one of the most important properties of all modern operating systems. The 8086 processor on which MINIX was supposed to run does not support virtual memory. Also the memory limit of 640 KB imposed by the PC design meant that multiprogramming (or multitasking) could only be implemented in a very limited form. But multiprogramming is one of the most important aspects of all modern operating systems.
- MINIX is a stand-alone operating system capable of running on a 'bare' machine. That is a point in its favor if one wants to use it as a substitute for other operating systems (i.e. 'UNIX for the laptop'). But it is a severe handicap when what one wants to do is alter, test, debug and profile the system. In that case it is much better to have a system running on a virtual machine implemented on top of another operating system with a rich set of tools for debugging, profiling, etc. After all, for the purposes of instruction

and experimentation, it isn't important whether the new system runs on a bare machine.

Tanenbaum rightly criticizes the 'monolithic' structure of traditional operating systems like UNIX and proposes that an operating system should be broken up into a set of independent processes that communicate via messages. But for some reason he does not take this idea very far with MINIX. He makes the file manager an independent process but leaves all the rest as a 'monolith'.

However, the idea of breaking up an operating system into independent communicating processes has innumerable advantages as we shall see. It is the direction in which several modern operating system developments are going: MACH, OSF/1. The idea ought to be consistently followed.

Another very important idea in modern operating system development is that of lightweight processes or threads. This idea should play a central role in the design of an operating system but it does not appear in MINIX.

The reader familiar with the literature on operating systems may wonder why I have not yet mentioned the book *Operating System Design: The XINU Approach* by Douglas Comer. In fact, however, the title of that book is misleading; Comer does not describe the design of operating systems, just that of XINU. And XINU is not really an operating system; it is a set of runtime primitives to support concurrent programming. One wonders why Comer did not take the logical next step and provide a suitable language for concurrent programming. He left it to Gehani and Roome to invent Concurrent C.

All of these points taken together seemed to me to be weighty enough to require the development of a new course and a new operating system. The ultimate goals in developing TUNIX follow directly from the above list.

- TUNIX should have all the properties of a serious operating system including demand paging and genuine multitasking – i.e. supporting a large number of processes running 'simultaneously'.
- TUNIX should have a structure similar to that of MACH: a very small 'microkernel' that manages the message passing. All the rest (including the device drivers) should be allotted to independent processes.
- TUNIX should be implemented at least initially on a virtual machine running on an established operating system – here UNIX System V. Practically this means that the microkernel is an independent UNIX process instead of being a part of each process as it would be in a 'real' implementation. And it means that the drivers emulate the devices they are supposed to control. This does not mean that TUNIX could not be implemented on a real machine. To do that one would 'merely' have to turn the drivers into real drivers and write some code for the specific target CPU (for building page tables and the like). I do not mean to imply that these last steps are trivial; I simply mean they aren't terribly important for the purpose for which TUNIX was developed.
- TUNIX should implement threads and use them to achieve 'concurrency' in a transparent fashion.

For purely practical reasons it was decided to make the programmer's interface (what in modern jargon is called an API = Application Programmer's Interface) to TUNIX completely compatible to that of UNIX System V.3. This meant taking over some things from UNIX that one is very tempted to improve on, but it has at least two practical advantages:

- One does not have to write a manual for the programmer's interface. Everyone familiar with UNIX (and that is quite a crowd) will immediately understand the TUNIX API.
- The great mass of software developed for UNIX System V.3 will run on TUNIX without change.

  If TUNIX should be implemented on a real machine it should be possible with some care with the details to achieve binary compatibility. That is, binary executables that run under UNIX should run without recompilation under TUNIX.

In spite of the remarkable functionality offered by TUNIX (practically that of UNIX System V.3) the code is astonishingly compact. And so the material in this relatively short book could be successfully presented in a one-semester course. It was even possible to discuss things which are not to be found in System V.3.2, such as streams and sockets (Chapter 9) and multiprocessing (Chapter 10).

Nevertheless, in order to achieve the goal of presenting material that could be reasonably handled in one semester certain choices had to be made. For this reason some topics that one might regard as belonging to the subject of operating systems are treated only sketchily or not at all.

This book really only treats the central part of the operating system – the part usually called the kernel. All the utility programs that traditionally belong to an operating system and without which it would not be very useful are not treated here. In particular the topic of archiving and making backups of file systems is not treated, since that can best be handled by such utility programs.

Another topic that gets short shrift here is that of system security. That is not to say the topic is unimportant; indeed with the spread of networking the importance of system security has grown almost exponentially. However, this is a thorny subject and deserves a book of its own. A whole new school of experts has grown up to deal with the security problem. Despite such restrictions I believe other teachers of intermediate to advanced courses on operating systems will find this book useful.

Originally I did not intend to include any real C source code in the book. It was my belief that well-formulated pseudocode says more than real code does. Either I am wrong in this conviction or I have not succeeded in giving really informative pseudocode descriptions, because several of the reviewers urged the inclusion of all or at least significant parts of the TUNIX sources. I have acceded and the book now has seven appendices with large parts of the TUNIX code. I have tried to include only those parts that are discussed in the text or are needed to understand other parts. Unfortunately the appendices more than double the length of the book.

Anyone wanting to acquire the complete code for teaching purposes can obtain it from anonymous FTP at the net address gwdu03.gwdg.de in the directory *tunix*. University teachers can obtain the code on a diskette by writing to Prentice Hall at Department 32, Prentice Hall, Campus 400, Maylands Avenue, Hemel Hempstead, HP2 7EZ, UK.

Robert Switzer
Göttingen, May 1992

Contents

Chapter 1

A Short History

It is a very difficult undertaking to give an abstract definition of an operating system. Since the reader of this book is surely not inexperienced with computers, he or she is certain to have at least a rough idea of what an operating system is. So we will not even attempt such an abstract definition.

On the other hand the reader is much less likely to know much about the history of those complex programs called 'operating systems'. Since a knowledge of this history helps to understand the operating systems we use today, we shall briefly recapitulate that development in this chapter.

In the beginning there was the bare machine, and user, programmer and operator were all one person. The user signed up on a list for a particular time – let's say from 3 p.m. to 4 p.m. During that hour he had the machine all to himself. He entered his program – originally word for word or byte for byte using the toggles on the console. Later this was done using punched paper tape or punched cards. Then he used the toggles on the console to enter the starting address of the program and pressed the start button. All the time the program ran the user could stop the machine and read the contents of the registers from the rows of lamps on the console (in binary format!).

One could also 'dump' the contents of the memory to the printer or the card punch. In this way one could debug one's program 'interactively'. As someone said in those days: 'You could feel the bits between your toes.' In many ways these were good times for programmers: running the computer was tedious, but one had full control of everything that happened.

The situation just described repeated itself three times in the history of the computer:

- With the very first computers at the end of the 40s and the beginning of the 50s.
- With the so-called minis like the PDP-1,-2, ..., in the mid 60s.
- With the earliest microcomputers around 1975.

In the case of the latter two categories it was the sheer tediousness that led to the improvement of the operation mode. With the first category, however, it

was chiefly economic considerations that made people look for a better way of doing things. Peterson and Silberschatz illustrate the economic need with the following comparison: the operating costs of an IBM 7094 were well over $50 per hour at a time when the hourly wage of a programmer wasn't likely to be more than about $5. This way of doing things was just too costly.

Here we encounter for the first time two problems, whose solution will be a continuing theme in this book:

- The tediousness of programming and using the system.
- The poor utilization of the expensive machine, especially the CPU.

Let us look at the crux of the first problem. The programmer not only had to know the instructions of the CPU by heart (as bit sequences in the days before there were assemblers); he also had to be intimately acquainted with the properties of peripheral devices like the card reader, printer, etc. He had to know how to program these devices. Each of his programs needed sections of code to control the peripheral devices. If newer, more modern devices were acquired, then he had to change the corresponding parts of all his programs.

The earliest solution to this problem consisted of building standardized program segments for controlling peripheral devices. Those were the first software libraries. Every programmer could employ routines out of these collections in his programs – possibly with changes to deal with his special needs.

The first solution to the second problem, the poor utilization of the machine, was the following. Almost every computing center in the world hired professional operators, whose job was to rationalize the sequence of operations on the computer. Now the programmer had to hand her program over to the operator in the form of a deck of punched cards or punched paper tape and a precise description of the running conditions. If her program ran correctly she received the output in the form of printed output listings, punched cards, etc. If it did not run correctly she was given a printed 'core dump' – a long list of octal or hexadecimal numbers representing the contents of the computer memory at the moment her program was aborted.

On the one hand the programmer could now concentrate on her actual task – namely programming – because she no longer needed to worry about operating the computer. On the other hand, however, she had lost the direct contact with the events that occurred during the run. It was usually much harder to find the mistake by inspecting the core dump than it had been when she sat in front of the console.

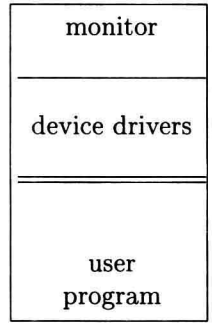
The operators could now group the runs in appropriate ways (batching). They waited until they had several FORTRAN programs to compile; then they loaded the FORTRAN compiler from tape and compiled all the FORTRAN programs at once. After that they could load the COBOL compiler and translate all the COBOL programs. Finally they could run all the successfully translated programs one after the other. FORTRAN and COBOL were the first high level languages to be used on these machines.

This rationalization brought a definite improvement in the utilization of the CPU. Nevertheless the CPU was busy far less than 50% of the time because, for example, magnetic tapes had to be mounted and output had to be printed. Obviously other improvements still had to be found.

The direction these improvements took has already been indicated: one copied several **runs** (user programs) onto a magnetic tape and let the computer run these one after another. The output was initally written to another tape and printed at a later time or, better still, by another (auxiliary) computer.

But in order for this to function smoothly there had to be a sort of 'super program', which controlled the loading, running and terminating of the user programs. This supervisor program was called a **monitor** or **resident monitor**, because it stayed in the memory of the computer at all times from the moment the computer was started (powered up) until it was shut off (powered down). All the other programs came and went in a continual stream under the supervision of this monitor.

Of course the monitor had to be able to drive the peripheral devices – in particular the tape drive. Since the routines to handle the devices were already in the memory in the form of monitor code, it was no longer necessary for each user program to include such routines; that would have been a waste of the scarce computer memory. One could therefore let the user programs call the routines in the monitor.



Thus the monitor offered a solution to both the problems we mentioned earlier. The monitor was the grandfather of our modern operating systems. But even with the monitor CPU utilization was still low.

Example. Let us suppose the user communicates with the computer through a 1200 baud terminal – i.e. at a rate of about 120 characters per second. That means the terminal accepts a character only once every 8 ms, i.e. every 8000 μs. If the CPU produces a character every 8 μs, then it is waiting 99.9% of the time. Now one might suppose the solution would be to install a buffer for the output characters. The CPU puts a character into the buffer whenever it has one ready and the terminal asks the CPU for a character from the buffer whenever it can accept another one. But a moment's thought shows that this is a solution only if the CPU produces characters *on the average* as fast as the

terminal consumes them, because if the CPU is faster than the terminal, then the buffer will run over or else the CPU will very soon have to start waiting again. If, however, this condition is approximately fulfilled, then the buffer idea is a good one. The CPU can get rid of characters as they are produced and then go on to do something useful instead of waiting for the terminal. And the terminal can receive a character whenever it is ready – as long as the buffer is not empty.

For this idea to function properly the terminal has to have a way of 'letting the CPU know' whenever it needs another character from the buffer. This problem was solved with **interrupts**: the CPU has a special line or pin with which its work can be interrupted. The CPU then asks the interrupting device for the reason for the interrupt and proceeds to carry out a suitable routine (interrupt handler) – e.g. to send the terminal the next character from the buffer. After finishing this routine, the CPU resumes its work at the point where it was interrupted.

The introduction of interrupts as a feature of the hardware illustrates a phenomenon we shall see often. It is self-evident that the architecture of the hardware – in particular of the CPU – influences the form and function of the monitor and later the operating system. But the needs of the monitor or operating system have often influenced the design of the hardware – as in the case of the interrupts.

A second example of this phenomenon was the so-called 'fence register'. User programs are often incorrect and in particular they often produce incorrect addresses – that is, they read or write data at memory addresses they shouldn't be touching. Since the monitor and the user program share the computer memory, this means that an incorrect user program can destroy parts of the monitor. Then the operator has to reload the monitor, which costs valuable time.

To prevent this sort of mischief, the memory was divided into two regions: the monitor region and the user region. A **fence register** contained the address of the boundary between the two regions. The CPU correspondingly had two modes of operation: monitor and user mode. User programs ran in user mode and could only address memory cells in the user region. If the user program had finished its run or if it needed the services of the monitor (input or output through device drivers), then it had to switch to monitor mode. The monitor reinstated user mode before returning control to the user program.

We saw earlier that the idea of buffered input/output and CPU interrupts only improves the CPU utilization if the I/O requirements of the program correspond on the average to the speed of the I/O devices. However, this equilibrium situation seldom occurs in practice. Mostly one has either an

(a) **I/O bound** program: the I/O devices are continually in use, the CPU is continually having to wait;

or a

(b) **CPU bound** program: i.e. intensive computation, little I/O. Now the CPU is fully utilized but the I/O devices have little to do.

To achieve a genuine equilibrium we need **multiprogramming**. This means that several programs are in the computer memory *simultaneously*. If program A is running and needs an I/O operation, then the CPU starts this operation and then switches control to another program B that currently needs a CPU phase.

At a later time the I/O operation for A is done (which is signaled by an interrupt). Then the CPU can switch back to program A at an appropriate time – say when program B needs an I/O operation. For this method to work well one needs a good mixture of programs in the memory; they should not all be I/O bound or all CPU bound.

Multiprogramming throws up a whole new series of interesting problems. It also places new demands on the hardware. For example, we now need several 'fences', so that the various user programs do not disturb or destroy each other. The peripheral devices must also become more independent; they must be able to carry out the tasks the CPU gives them without further support from the CPU. Thus the controller for a magnetic disk should be able to transfer an entire block of data from the memory to the disk (or vice versa), without assistance from the CPU. This is now known as DMA (Direct Memory Access).

The multiprogramming we have just described was intended for so-called batch processing; the programmer still hands the operator his job in the form of punched cards or paper tape and at some later time (often hours later) he gets back his output – for example, a printed listing or dump.

In the early 70s there was a growing demand for **interactive** or **time-sharing** systems. One wanted to enter data or programs at a terminal (video screen with keyboard) and see the results more or less immediately. Multiprogramming makes such interactive processing possible in principle. Each user communicates with his program in the computer memory. The CPU only spends a short time in each program and switches rapidly from one program to another, so that each user has the impression he has the CPU all to himself.

This sort of interactive time-sharing usually functions well, because most users spend most of their time with the entry of characters via the keyboard. If a user can enter at most 5 characters per second and the CPU only needs 20 μs per character, then the CPU only spends about 0.01% of its time with each such user.

Today such time-sharing systems are the rule and batch systems are becoming rarer.

Since about 1980 the scene has been significantly influenced by a further development: **networked** and **distributed** systems. It will not be long before most of the computers in the world communicate with each other via a world-wide network. ('Computer' here is not to be taken to include embedded systems.) This makes it possible for a user at computer A to use resources (files, programs, printers, plotters, etc.) at computer B in the same room or on the other side of the world – almost as if these resources were present on computer A. This also makes it possible (and partly also necessary) to distribute parts of an operating system over several computers. One speaks of **distributed systems**.

In the past decade there has also been a rapid advance of computers with more

than one CPU. This development also puts new demands on operating systems.

A rough time scale of the developments described here is shown below.

	bare machine
1950	monitor
1960	multiprogramming
1970	time-sharing
1980	networks
1990	distributed systems multiprocessor systems
2000	??