CONCURRENT EUCLID, THE UNIX* SYSTEM, AND TUNIS

R.C. Holt

CONCURRENT EUCLID, THE UNIX* SYSTEM, AND TUNIS

R.C. Holt

Computer Systems Research Group
University of Toronto

ADDISON-WESLEY PUBLISHING COMPANY
Reading, Massachusetts | Menlo Park, California
London | Amsterdam | Don Mills, Ontario | Sydney

This book is in the Addison-Wesley Series in Computer Science

Consulting Editor
Michael A. Harrison

Holt, R. C. (Richard C.), 1941-
Concurrent Euclid, UNIX, and TUNIS.

Bibliography: p.
1. Concurrent Euclid (Computer program language)
2. UNIX (Computer system) 3. TUNIS (Computer program)
I. Title
QA76.73.C64H64 1983 001.64'2 82-13742
ISBN 0-201-10694-9

*Unix is a trademark of Bell Laboratories.

SCRABBLE ® is the registered trademark of Selchow & Righter Company for its line of word games and entertainment services. Reprinted with permission.

Diagrams in this book were prepared by the Media Centre, University of Toronto.

Reproduced by Addison-Wesley from camera-ready copy prepared by the authors.

ISBN 0-201-10694-9
ABCDEFGHIJ-AL-898765432

PREFACE

This book introduces the art of concurrent programming. This particular type of programming, with several activities progressing in parallel, is intellectually intriguing and is essential in the design of operating systems. Unix is used as a case study for exploring operating system structures. The Tunis implementation of Unix's nucleus (kernel) is presented as an example of a large concurrent program. Although the emphasis is on operating systems, the design and implementation techniques presented apply as well to other high performance, highly reliable software such as that in computer networks, real time control and embedded microprocessor systems.

The first two chapters overview concurrent programming and operating systems. Chapters 3 and 4 introduce the Concurrent Euclid (CE) language. Chapter 5 presents standard concurrency problems and their solutions. Chapters 6, 7 and 8 concentrate on Unix. Chapter 9 gives the structure of Tunis, a Unix-compatible nucleus written in CE. The last chapter shows how to construct a small kernel to support concurrent processes. An appendix gives the detailed specification of the Concurrent Euclid language.

The required background of the reader is a familiarity with a high-level language such as Pascal or Fortran as well as some familiarity with computer architecture. The programs presented in this book are written in Concurrent Euclid. This is a language that is suited for developing high performance system software as well as for teaching. Student's CE programs that use parallel processes are conveniently executable under systems such as Unix/11 and Unix/VAX. Alternatively, these programs can be down-loaded and executed on microprocessors such as the MC68000 and MC6809. The CE compiler is available from the CE Distribution

Manager, Computer Systems Research Group, University of Toronto, Toronto, M5S 1A4, Canada.

This book can serve as the main or subsidiary text for a course on operating systems or systems programming. Alternately, it may be used as a text book in a specialized course such as one on concurrent programming.

Acknowledgements. This book has evolved from an earlier book "Structured Concurrent Programming with Operating Systems Applications", which I co-authored with G.S. Graham, E.D. Lazowska and M.A. Scott. The present book has been made possible due to their essential contributions to its predecessor. I want to thank the people who have taken the trouble to suggest improvements in the earlier book; in particular the detailed comments by S.S. Toscani have been helpful to me in preparing this new book. J.C. Weber, S.G. Perelgut, D.R. Galloway, M.P. Mendell and D.T. Barnard have helped by reading drafts of the new book.

The Concurrent Euclid language was designed by J.R. Cordy and myself. This language is based on the Euclid language designed by B.W. Lampson, J.J. Horning, R.L. London, J.G. Mitchell and G.J. Popek with assistance from J.V. Guttag. B.A. Spinney, C.R. Lewis, B.W. Thomson, and C.D. McCrosky contributed to the CE language design and/or its compiler. D.B. Wortman, D.R. Crowe and I.H. Griggs helped inspire CE's design by their roles as implementors (with J.R. Cordy and myself) of the Toronto Euclid compiler.

P. Cardozo, M.P. Mendell, I.J. Davis and G.L. Dudek have done MSc projects involving Tunis design and implementation. S.W.K. Tjiang, D.R. Galloway and D.T. Barnard have also contributed to the Tunis work. The continuing interest of P.I.P. Boulton and E.S. Lee in the CE and Tunis work has been important to its progress. The following have been students in my graduate course in which we studied and evolved the Tunis design: P. Cardozo, A. Curley, R.S. Gornitsky, J.S. Hogg, S.A. Ho-Tai, P.M. McKenzie, J.L. More, B.A. Spinney, B.W. Down, G.L. Dudek, D.R. Ings, P. Kates, P.A. Matthews, M.P. Mendell, L.M. Merrill, R. Parker, B.R.J. Walstra, H.E. Briscoe, D. Chan, L. DeMaine, E.L. Fiume, R.D. Hill, L.Z. Zhou, S.G. Perelgut, and Y.C.L. Wong.

The information on Unix in this book is based upon widely available materials, particularly upon excellent articles by the authors of Unix (D.M. Ritchie and K. Thompson). As they have stated, "The success of Unix lies not so much in new inventions but rather in the full exploitation of a carefully selected set of fertile ideas..."

I.S. Weber has prepared the book for publication using a computer text editor and phototypesetter.

The research leading to CE and Tunis would not have been possible without the financial support of the Canadian Natural Sciences and Engineering Research Council and of Bell Northern Research Ltd.

R.C. Holt
May 1982
Toronto

CONTENTS

Chapter 1

CONCURRENT PROGRAMMING AND OPERATING SYSTEMS

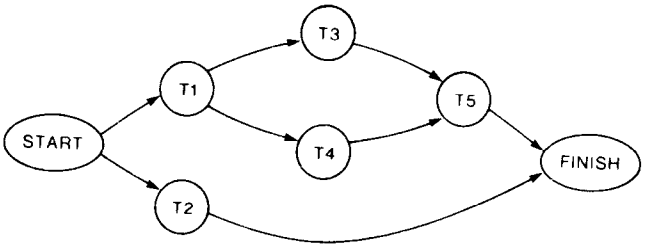
Concurrent programming means writing programs that have several parts in execution at a given time. The concept of concurrent or parallel execution is intellectually intriguing and is essential in the design of computer operating systems. This book covers the fundamentals of concurrent programming using structured techniques. After an introduction to the need for concurrent programming and its basic concepts, a notation called monitors is presented and used for solving problems involving asynchronous program interactions. The concurrent algorithms in the book are presented in the Concurrent Euclid (CE) programming language. It is a language designed to support the development of highly reliable, high performance systems programs.

After giving examples of concurrency, this chapter concentrates on operating systems. Operating systems implement concurrent programs by sharing CPU time among several programs and use concurrent programs to control resources and serve users.

EXAMPLES OF CONCURRENCY

In programming, and in other activities, concurrency problems can arise when an activity involves several people, processes or machines proceeding in parallel. We will give several examples of concurrency, beginning with one that does not involve computers.

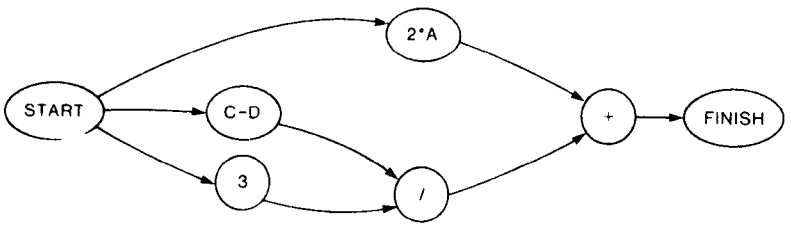
An example: activities in a large project. A large project such as the construction of a building is accomplished by many workers carrying out different tasks. These tasks must be scheduled, and one method of doing this uses *precedence charts,* as shown here.



This example chart shows that in the beginning tasks T1 and T2 can both be started. After T1 is done, T3 and T4 can be started, and when both of them are done, T5 can be started. The whole project is finished when T2 and T5 are done. As the next example will show, precedence charts can be used to specify concurrency in computer programs.

There are two main reasons for using parallel tasks in this example. First, there are many workers available and they must be allowed to work at the same time (in parallel). Second, the project can be completed in less elapsed time if tasks are allowed to overlap. In computer systems, analogous reasons (many asynchronous devices and the need to shorten elapsed time) may result in concurrent programming.

An example: independent program parts. Precedence charts can describe possible concurrency in a computation. The expression (2*A)+((C-D)/3) can be evaluated sequentially (one operation at a time) by finding the product, difference, quotient and sum, in that order. But parallelism is possible, because some parts of the expression are independent, as is shown in this precedence chart.



Groups of statements, as well as expressions, may have independent parts that can be executed in parallel. For example, the following loop written in Pascal determines if 'Jones' is in a list by testing name[1] then name[2] and so on.

```
for i := 1 to size do
    if name[i] = 'Jones' then
        found := true
```

This loop could be executed by checking all of the names at the same time, because the tests are independent. For a large computation, parallelism such as this can minimize the elapsed time for completion.

As computing elements such as microprocessors become cheaper, it becomes more and more attractive to split programs into several parallel tasks. In the future we may find that computers are built as huge collections of tiny processing elements, analogous to building an elephant out of a swarm of mosquitoes or bees, and we will need to know how to program such contraptions.

An example: a simulation. Sometimes programs are written to simulate parallel activities. For example, a program might simulate boats entering a harbor; this program could predict the effects of increased boat traffic. A good way to program this simulation is to have an asynchronous program activity (a *process)* corresponding to each simulated activity (each boat). Each process mimics its boat, and the interaction of these processes models the interaction of boats entering the harbor. Programming the simulation is done by writing the constituent concurrent programs.

An example: control of external activities. Special purpose computer systems are used to control chemical processes such as the manufacture of cement. Sensors transmit signals to the computer to report temperature, pressure, rate of flow, etc. The computer in turn transmits signals that set valves, control speeds, sound alarms, etc. The computer system also keeps a log of its actions and prints reports. A computer system such as this keeps track of many interrelated concurrent activities. One good way to program such a system is to have a concurrent software process in the computer for each external activity. A software process tracks its corresponding activity; it is responsible for sending and receiving signals to and from the activity. Programming this computer system is done by writing the concurrent programs that observe and control the activities.

These examples have given various practical uses of concurrency. One of the most important examples of concurrent programming arises in operating systems. The next sections explain why this concurrency arises and how it is handled.

communication occurs among the devices, the operating system and the

OPERATING SYSTEMS

Modern computer installations have many asynchronous hardware components, such as operator consoles, card readers, printers, disk drives, tape drives and CPUs. The operating system must ensure that these components are used efficiently and that they provide convenient service for the users.

An operating system consists of a collection of software modules. These modules receive requests from users (for example, to execute the users' programs) and must schedule the system's components to satisfy these requests.

The operating system may support *multiprogramming*, that is, it may allow more than one user's program to be in execution at a given time. To support multiprogramming, the operating system must share the system's resources among the executing programs. Some resources, such as tape drives, are exclusively allocated to a program, until the program terminates or no longer needs the resource.

Other resources, such as the CPU, are shared dynamically, in a way that gives the appearance that each program has its own *virtual* resource. For example, the operating system may allocate a "slice" of CPU time to one program, then to another program, and so on. This is called *time slicing* and gives the appearance that each program has a virtual CPU, which is like the physical CPU but somewhat slower. As a second example, the operating system may provide each program with a virtual memory. This is done with the help of special hardware (for "paging" or "segmenting") that allows the operating system to allocate physical memory only to the active parts of programs.

There are two basic reasons why multiprogramming is needed in computer systems. The first is for efficient use of hardware resources and the second is for quick response to users' requests. First we will consider efficiency. The system's hardware components run in parallel at vastly different speeds. For example, the time to process a single character may vary from a tenth of a second for a slow console, to a thousandth of a second for a printer, to a millionth of a second for a CPU. Clearly, the CPU should not be forced to waste time (100,000 of its operations) while a console transmits a character. While a user is typing messages to a running job, another job should be given the CPU. If a job is *I/O bound*, spending most of its time waiting for input/output devices, the spare CPU time can be used by a *compute bound* job, which spends most of its time using the CPU. If the system has a variety of equipment, a job that uses only a few of the devices should not prevent concurrent use of other devices. These examples show how multiprogramming provides more efficient use of

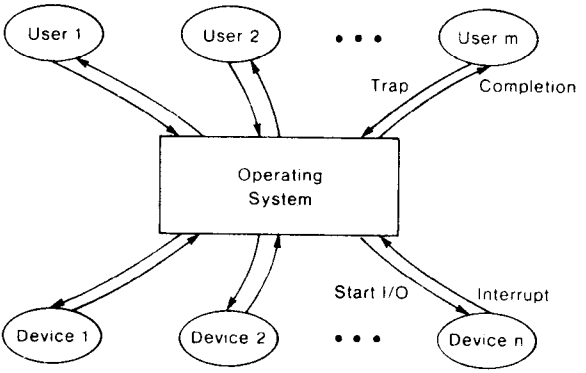
computer equipment.

Apart from efficiency, multiprogramming allows the computer system to respond quickly to users' needs. Suppose a user has a short, urgent job but a long-running job is already in the system. With multiprogramming, the short job can run in parallel with the long one and can finish hours before it. In interactive systems, a form of multiprogramming is necessary, with one program for each user. The system is shared among the interactive users and their programs so that each receives good response; this is called *time sharing.* These examples show how multiprogramming allows prompt attention to users' needs.

COMMUNICATION IN OPERATING SYSTEMS

Operating systems must be organized so as to control hardware devices and run users' programs. This section gives a simplified model of how communication occurs among the devices, the operating system and the users' programs; the next two sections describe how operating systems are organized to handle this communication.

An operating system controls an I/O device by sending it a *start I/O command.* If the device is a tape drive, the command may cause it to read a record, putting the record's characters into main memory. When the device has carried out the command, it can send an *interrupt* signal back to the CPU indicating that it is free to carry out another operation. This signal can switch the CPU from a user's job to the operating system; this allows the operating system to send another command to the device before returning the CPU to a user's job.

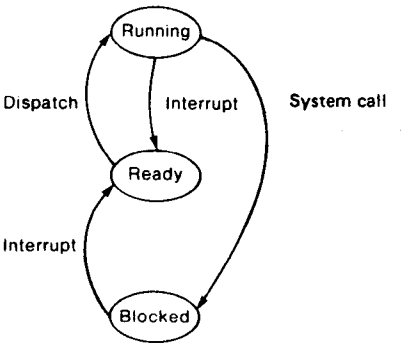


Meanwhile, each user's job occasionally makes requests to the operating system, for example, to read from a terminal or to write a disk track. The job makes a request by a *trap* or a *system call* instruction; this

instruction is like a subroutine call and transfers control from the user to the operating system. Having received such a request, the system blocks the job (gives it no more CPU time) until the requested action has been completed Then the job is unblocked and allowed to continue executing.

There is usually an *interrupting clock;* it sends interrupt signals that transfer control from a user's job to the operating system. This allows the system to implement time slicing by passing the CPU from user to user, and to cancel a user's job that is using excessive CPU time. Without the interrupting clock, an infinite loop in one user's job could prevent other users (and the operating system) from using the CPU.

When a program is actually using the CPU, we say it is *running.* When it is waiting for a request to be serviced, we say it is *blocked.* When a program would be running except that the CPU is allocated to another program, we say it is *ready.* The operating system maintains a queue of the programs that are ready. This transition diagram shows how the states of a program change:



We say the operating system *dispatches* a program when it lets it run, by giving it the CPU.

A trap generally causes a program to be blocked; however in some instances (not shown in the diagram) if the operating system can immediately satisfy the request, the user program is again dispatched and no blocking occurs. Other than by a trap, the only way a running program loses the CPU is by an interrupt. A clock interrupt may signal the end of the program's time slice, or an I/O interrupt may allow another program to run. In a system with only one CPU, at most one program can be in the running