PREFACE

The aim of this book is to equip students with an integrated view of modern software systems. Concurrency and modularity are the unifying themes. It takes a systems approach rather than a programming approach, since concurrent programming is firmly rooted in system design. The language is an implementation tool for the system designer and programming languages are covered throughout the book from this perspective.

The formal theory of concurrency is not included. Rather, the aim is to provide a systems background to which subsequent formal study can relate. Without prior study of real systems a student can have little intuition about the basis of formal models of concurrent systems. For students about to work on the theory of concurrency the book provides a summary of essential system fundamentals.

The structure of the book is:

- **Introduction**, in which some real-world concurrent systems are described and requirements for building computerized concurrent systems established.
- **Part I**, in which the abstraction of a concurrent system as a set of concurrent processes is established and the implementation of processes in operating systems and language systems is studied.
- **Part II**, which shows how a logically single action (an operation invocation) can be guaranteed to run without interference from other concurrent actions.
- **Part III**, which shows how a number of related actions can be guaranteed to run without interference from other concurrent actions.
- **Part IV**, in which some case studies are considered from perspective developed throughout the book.

Computer systems curriculum

I have taught operating systems, distributed systems and computer architecture for many years. Because distributed operating systems have come into widespread use comparatively recently, most curricula include them at final-year undergraduate or postgraduate level. Distributed operating systems are now commonplace and a student is more likely to be using one than a centralized time-sharing system. It is somewhat artificial to cover the functions of a shared, centralized operating system in great detail in a first course, particularly when the rate of development of technology makes it essential constantly to re-evaluate traditional approaches and algorithms.

In general, there is a tendency for closely related specialisms to diverge, even at undergraduate level. An overview of system components and their relationships is desirable from an early stage:

- Operating systems include communications handling.
- Language runtime systems work closely with (and are constrained by) operating systems.
- Real-time systems need specially tailored operating systems.
- Dedicated communications handling computers need specially tailored operating systems.
- Database management systems run on operating systems and need concurrency and file handling with special guarantees.
- Concurrent programs run on operating systems.
- Many system components employ databases.
- Distributed systems employ distributed databases.
- Distributed databases need communications.
- Distributed operating systems need transactions.

Concurrent Systems achieves this integration by setting up a common framework of modular structure (a simple object model is used throughout) and concurrent execution.

I have used this approach in the Computer Science curriculum at Cambridge since 1988, when a new three-year undergraduate degree programme started. A concurrent systems course, in the second year of a three-year degree course, is a prerequisite for further study in distributed operating systems, communications and networks, theory of concurrency, and various case studies and projects. Figure 1 suggests an order of presentation of systems material. Courses in the general area of real-time, embedded control systems would also follow naturally from this course. At Cambridge, a course which gives an overview of system software precedes "Concurrent Systems" so the students start with some knowledge of file management and memory management. As this may not universally be the case I have included these functions in Part I of the book. I take the detailed aspects of these operating systems functions in the context of special purpose systems' requirements after the concurrent systems material has been covered.

In the ACM Curriculum 91 for Computing (see Denning et al., 1989; Tucker 1991), the general topic 'operating systems' includes distributed operating systems and communications. Curriculum 91 identifies the three major paradigms of the discipline as: **theory**, which is rooted in mathematics; **abstraction** (modeling), which is rooted in experimental scientific method; and**design**, which is rooted

in engineering. Theory deals with the underlying mathematics of each sub-area of computing. Abstraction allows us to model large, complex systems in order to comprehend their structure and behavior and carry our experiments on them. Design deals with the process of implementing a system to meet a specification. The approach taken here embodies abstraction and design and establishes the basis for theory.

Audience

It is assumed that the reader will come to this material with some knowledge and experience of systems and languages. First-year undergraduate courses on programming and systems software are appropriate prerequisites.

This book is intended as a **modern replacement for a first course in operating systems**- modern in the sense that concurrency is a central focus throughout; distributed systems are treated as the norm rather than single-processor systems, and effective links are provided to other systems courses. It is also suitable as a text to be read in parallel with more traditional, specialized courses in operating systems, communications and databases. It al so provides integrating and summarizing study for graduate students and practitioners in systems design including systems programming. Graduate students who are researching the theory of concurrency will find the practical basis for their subject here.

An Outline of the Contents

Chapter 1 describes a number of types of concurrent system and draws out requirements for supporting concurrent activities. Concurrent systems can exploit a wide range of system topologies and architectures. Although this area us bit addressed in great detail their characteristics are noted for reference throughout the book.

Chapters 2 through 7 form **Part I**. System design and implementation require software to be engineered. Software engineering, which involves the specification, design and implementation, maintenance and evolution of software systems has merited many books in its own right. This book focuses on concurrency issues, but a context of modular software structure is needed: first, to give a context for Part I, where intuition on where intuition on a single logical action is needed; second, in order that the placement of concurrent processes in systems may be understood.

Modular system structure is therefore introduced in **Chapter 2** and the modular structure of operating systems is used as an extended example. The idea that a minimal kernel or 'microkernel' is an appropriate basis for high-performance specialized services is introduced here. The concepts of **process** and protocol to achieve the dynamic execution of software are also introduced.

In **Chapter 3** device handling and communications handling are covered. These topics are treated together to highlight the similarities (between communications and other devices) and differences (communications software is larger and more complex than device handling software). The communications handling subsystem of an operating system is itself a concurrent (sub)system, in that at a given time it may be handling several streams of input coming in from various sources across the network as well as requests for network communication from local clients.

Chapter 4 covers memory management. The address space of a process is an important concept, as also are mechanisms for sharing part of it. Some of the more detailed material in this chapter and the next can be omitted on a first reading or deferred in a teaching context.

Chapter 5 gives the basic concepts of filing systems. File system implementations involve data structures both in main memory and in persistent memory on disk. Both the memory management and file management subsystems of operating systems are concurrent systems in hat they may have in progress both requests from clients and demands for service from the hardware.

Chapter 6 gives the detailed concrete bases for the process abstraction that is provided by operating systems. Once the process abstraction is created as one operating system function we can show how processes are used to achieve the dynamic execution of the rest of the system. Operating system processes may be used within operating system modules, while application-level processes may be located within application modules. There are several design options which are discussed throughout the book.

Chapter 7 is concerned with language systems and a particular concern is the support for concurrency. The relation between operating system and language system processes is discussed in detail. Multi-threaded processes are introduced.

Part I is mostly concerned with implementation. Knowledge of the material presented here is necessary for a thorough understanding of concurrent systems. Care must be taken, when working at the language or theoretical modelling levels, that the assumptions made can be justified for the operating system and hardware that will be used to implement a concurrent system.

We can now work with the abstraction of a concurrent system as a set of concurrent processes. **Part II** proceeds to explain the mechanisms for ensuring that a given concurrent process can execute without interference from any other, bearing in mind that processes maybe cooperating with other processes (and need to synchronize with them) or competing with other processes to acquire some resource.

Chapters 8 to 14 comprise Part II. In Part II we temporarily ignore the issues of composite operations and the need to access multiple resources to carry out some task and confine the discussion to a single operation invocation that takes place within a concurrent system.

The notion of a single abstract operation is informal and closely related to the modular structuring of systems. A process can, in general, read or write a single word of memory without fear of interference from any other process. Such a read or write is indivisible. In practice,

a programming language variable or a useful data abstraction, such as an array, list or record, cannot be read or written atomically. It is the access to such shared abstractions by concurrent processes that is the concern of **Part II**. Chapters 8 to 12 are mostly concerned with "load and go" systems that run in a single or distributed main memory. Chapters 13 and 14 start to consider the effect of failures in system components and process interactions which involve persistent memory.

Chapter 8 discusses the major division between processes which share memory, running in a common address space, and those which do not. Examples are given, showing the need for both types of arrangement.

Chapter 9 is concerned with the lowest level of support for process interactions. The architecture of the computer and the system is relevant here. It is important to know whether any kind of composite read-modify-write instruction is available and whether the system architecture contains shared memory multiprocessors or only uniprocessors.

Semaphores are introduced, with several examples of their use. Implementation issues are covered. A discussion of the difficulty of writing correct semaphore programs leads on to high-level language support for concurrency in the next chapter.

Chapter 10 looks at language primitives that have been introduced into high-level concurrent programming languages where the underlying assumption is that processes execute in a shared address space, for example, conditional critical regions and monitors.

Chapter 11 compares inter-process communication (IPC) mechanisms within systems where shared memory is available and where it is not. In both cases processes need to access common information and synchronize their activities.

Chapter 12 covers IPC for processes which inhabit separate address spaces. Pipes and message passing are discussed. The material here is relevant to distributed IPC, but the essential characteristics of distributed systems are not yet discussed.

Chapter 13 introduces the possibility that a system might crash at any time and outlines mechanisms that could be used to provide crash resilience. An initial discussion of operations which involve persistent data is also given.

Chapter 14 focuses on distributed systems. Their special characteristics are noted and the client-server and object models for distributed software are outlined. We see how an operation at one node of a distributed system can be invoked from another node using a remote procedure call protocol. Node crashes and restarts and network failures are considered. Although distributed IPC is the main emphasis of the chapter it concludes with a general discussion of naming, location and the binding of names to locations in distributed systems.

Chapters 15 through 20 comprise **Part III** where the discussion is broadened to composite abstract operations and the concurrent execution of their component operations.

Chapter 15 introduces the problems and defines the context for this study. Composite operations may span distributed systems and involve persistent memory.

Chapter 16 discusses the desirability of dynamic resource allocation and the consequent possibility of system deadlock. An introduction to resource allocation and management is given, including algorithms for deadlock detection and avoidance.

Chapter 17 discusses composite operation execution in the presence of concurrency and crashes and builds up a definition of the fundamental properties of transactions. A model based on abstract data objects is used.

Chapter 18 discusses concurrency control for transactions. Two-phase locking, time-stamp ordering and optimistic concurrency control are described and compared.

Chapter 19 is mainly concerned with crash recovery, although the ability to abort transactions for concurrency control purposes is a related problem. A specific implementation is given.

Chapter 20 extends the object model for distributed systems and reconsiders the methods of implementing concurrency control in this context. The problem of atomic commitment is discussed and a two-phase commit protocol is given as an example. A validation protocol for optimistic concurrency control is also given.

Part III has established the concept of transaction which is fundamental to all distributed systems.

Chapters 21 through 24 comprise **Part IV**, in which a number of systems are presented as case studies. Greater depth is possible here than in the examples used earlier. An aim is to show that the approach developed throughout the book helps the reader to comprehend large and complex systems.

Chapter 21 describes the basic UNIX Edition 7 design. The design is evaluated and the process management and interprocess communication facilities, in particular are criticized. The way these systems have been addressed in recent versions of UNIX (BSD 4.3 and System V.4) is then described.

Chapter 22 covers the design of microkernels. Mach and CHORUS are described in some detail because they provide binary compatibility with UNIX BSD 4.3 and UNIX System V.4 respectively. The chapter concludes with an outline of distributed systems research at Cambridge where a microkernel has been designed to support dynamic real-time systems based on ATM networks.

Chapter 23 first discusses how transaction processing monitors are implemented in terms of processes, IPC and communications. Some examples of TP systems in the area of electronic funds transfer are then given, for example, an international automatic teller

machine (ATM) network.

Chapter 24 draws out the main conclusions, integrating the various parts of the book.

Although many **examples** of hardware, systems and language are used, two extended examples have been selected. First, the UNIX 7th Edition design illustrates so many problems of concurrent systems so well that it is frequently used as an illustrative example. The UNIX case study in Chapter 21 shows how these long-understood problems have been addressed in the current UNIX offerings. The second extended example is not such an obvious choice. I needed to show the RISC approach to computer architecture and its implications for concurrent systems. Because the local workstations are based on the MIPS R2000 I know this system well, and have used it as an example of interrupt handling, memory management, processor design and (lack of) support for concurrency.

The **appendix** presents two problems in greater depth than can be justified within a single chapter. N-process mutual exclusion for shared memory and distributed systems is the first. The management of a cache of disk buffers, highlighting the problems arising from concurrent access, is the second. Both are left open-ended and exercises are given.

Order Of Presentation

Figure 2 indicates dependencies in the material and shows how the chapters might be selected for courses emphasizing operating systems, concurrent programming or databases. It is not essential to cover everything in Chapters 4 and 5 before later chapters are taken, as discussed above. Chapter 16 can be treated similarly. Sections from these chapters are referenced when they are needed and the topics could be embedded.

The material in Part II could be taken in a different order. Although there is a flow argument through the chapters as written, there is no inherent reason why shared-memory IPC has to come before that with no shared memory.

The book could be used for a conventional operating systems course by selecting Part I, Chapters 8 through 12 of Part II, Chapter 16 from Part III, and the UNIX case study. Distributed operating systems could be included in such a course from the start by following the whole of Parts I and II. The cae studies could then include both UNIX and microkernels. Although all of Part III is not essential for such a course, Chapter 16 should be included and parts of Chapters 15 and 17 are highly desirable.

A concurrent programming course could supplement Part I (possibly excluding Chapters 4 and 5) and Part II with full details of a language to be used for project work. Chapters 15 and 16 from Part III should also be included.

A course on concurrency control in database systems would use Part III, but earlier chapters wich cover operating system support for databases provide an excellent background.

Further Study

The book naturally leads on to a course on large-scale distributed system design. Issues of naming, location, placement, protection, authentication and encryption are introduced but not discussed in depth. The replication of data to achieve high availability and performance is not discussed and a follow-on course would cover this aspect of large-scale system design, including protocols for keeping the data consistent. The commitment protocol of Chapter 20 is a starting point. In such a course, operating systems, databases and communications are again integrated.

Optional courses on specialized systems, such as real-time embedded control systems follow from this material. How to achieve high performance in filing systems might be taken as another advanced topic, as might memory management, tailored for specific systems.

It is important to have a basis for correct reasoning about the precise behavior of concurrent systems and various mathematical models are already in use (Hoare, 1985; Milner, 1989). As mentioned above, theory of concurrency might also be further study.

Objective

The main emphasis of the book is system design: how to comprehend existing systems and how to design new systems. One can’t write certain kinds of concurrent systems in certain languages above certain operating systems. This book aims to show the reader why. Computers are marked optimistically. Names such as ‘real-time operating system’ are used with little concern for their meaning. In order to survive an encounter with a salesperson one must know exactly what one wants and must know the pitfalls to look out for in the systems one is offered. I hope the book will help concurrent systems designers to selct the right hardware and software to satisfy their requirements

Instructor's Guide

An Instructor’s Guide is available which contains the following:

- Curriculm design. An outline of parts of the ACM/IEEE-CS Computing Curricula 1991 is given. Uses of Concurrent Systems in the curriculum are discussed.
- Points to emphasise and teaching hints. For each chapter, key points, potential difficulties and suggested approaches to teaching the material are given.
- Solutions to exercises and some additional exercises. The solutions include examples of how the various designs that are discussed have been used in practice.

- A description of some environments for project work and how to get them.
- Overhead transparency masters.

Acknowledgements ...

Jean Bacon
Cambridge, November, 1992

[request_ebook] Concurrent Systems - Operating Systems, Database and Distributed Systems: An Integrated Approach. Author: Jean Bacon. Date: 2003.Â The coverage in this third edition is now wholly relevant to a distributed computing course. It provides students with the most up-to-date knowledge of the theory behind modern distributed systems, enabling them to move seamlessly from a first programming course to being able to program operating systems. The book is also suitable for self-study or distance learning and has been proven on a user base of many thousands of students. Computer Systems: An Integrated Approach to Architecture and Operating Systems. . Available.Â Currently, in the ubiquitous presence project, he is investigating software and hardware mechanisms for ubiquitous distributed computing for an environment comprised of distributed sensors, embedded data concentrators, and backend clusters. He received a Presidential Young Investigator (PYI) Award from the National Science Foundation (NSF) in 1990, the Georgia Tech Doctoral Thesis Advisor award in 1993, the College of Computing Outstanding Senior Research Faculty award in 1996, the College of Computing Dean's Award in 2003, and the College of Computing William "Gus" Baird

A distributed operating system is a software over a collection of independent, networked, communicating, and physically separate computational nodes. They handle jobs which are serviced by multiple CPUs. Each individual node holds a specific software subset of the global aggregate operating system. Each subset is a composite of two distinct service provisioners. The first is a ubiquitous minimal kernel, or microkernel, that directly controls that node's hardware. Second is a higher-level collection of