大学计算机教育丛书（影印版）

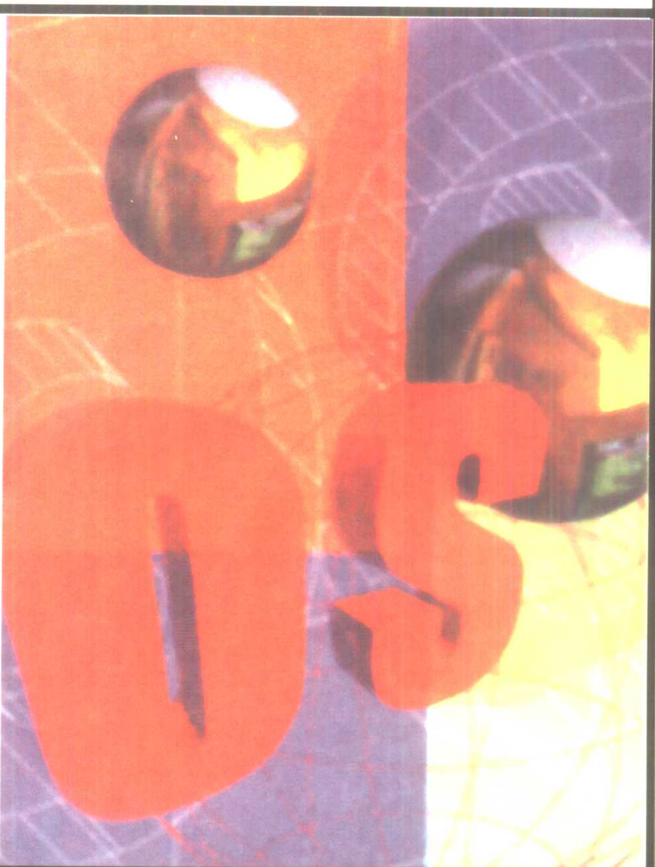
Second Revised Edition

Systems Programming and Operating Systems

D.M.Dhamdhere

系统程序设计和操作系统

第2版修订版



清华大学出版社
http://www.tup.tsinghua.edu.cn

McGraw-Hill
http://www.mhhe.com

Systems Programming and Operating Systems

Second Revised Edition

系统程序设计和操作系统

第 2 版修订版

D M Dhamdhere

Professor and Head

Department of Computer Science & Engineering

Indian Institute of Technology

Mumbai

清 华 大 学 出 版 社

McGraw-Hill Companies, Inc.

(京) 新登字 158 号

Systems Programming and Operating Systems, Second Revised Edition
D. M. Dhamdhere

图书在版编目(CIP)数据

系统程序设计和操作系统: 1: 第 2 版/( )达赫姆汗(Dhamdhere, D. M.)著. -影印版. -北京: 清华大学出版社, 2001.3
(大学计算机教育丛书/周维焜主编)
ISBN 7-302-04314-0

Ⅰ. 系… Ⅱ. 达… Ⅲ. ①系统程序-程序设计-英文②操作系统(软件)-英文 Ⅳ. TP311

中国版本图书馆 CIP 数据核字(2000)第 14574 号

出版者: 清华大学出版社(北京清华大学学研大厦, 邮编 100084)
http://www.tup.tsinghua.edu.cn
印刷者: 清华大学印刷厂
发行者: 新华书店总店北京发行所
开 本: 787×960 1/16 印张: 42.5
版 次: 2001 年 5 月第 1 版 2001 年 5 月第 1 次印刷
书 号: ISBN 7-302-04314-0/TP・2536
印 数: 0001～3000
定 价: 56.00 元

出 版 者 的 话

今天，我们的大学生、研究生和教学、科研工作者，面临的是一个国际化的信息时代。他们将需要随时查阅大量的外文资料；会有更多的机会参加国际性学术交流活动；接待外国学者；走上国际会议的讲坛。作为科技工作者，他们不仅应有与国外同行进行口头和书面交流的能力，更为重要的是，他们必须具备极强的查阅外文资料获取信息的能力。有鉴于此，在原国家教委所颁布的“大学英语教学大纲”中有一条规定：专业阅读应作为必修课程开设。同时，在大纲中还规定了这门课程的学时和教学要求。有些高校除开设“专业阅读”课之外，还在某些专业课拟进行英语授课。但教、学双方都苦于没有一定数量的合适的英文原版教材作为教学参考书。为满足这方面的需要，我们陆续精选了一批国外计算机科学方面最新版本的著名教材，进行影印出版。我社获得国外著名出版公司和原著作者的授权将国际先进水平的教材引入我国高等学校，为师生们提供了教学用书，相信会对高校教材改革产生积极的影响。

我们欢迎高校师生将使用影印版教材的效果、意见反馈给我们，更欢迎国内专家、教授积极向我社推荐国外优秀计算机教育教材，以利我们将《大学计算机教育丛书（影印版）》做得更好，更适合高校师生的需要。

清华大学出版社
《大学计算机教育丛书（影印版）》项目组
1999.6

Preface to the Second Revised Edition

I started work on the second revised edition soon after the second edition was published. The primary motivation was to improve readability and focus, clarity of the fundamental concepts and utility of the examples. This involved a thorough editing of the text, with numerous improvements on each page. Some errors and ambiguities noticed while editing were also corrected.

Apart from these improvements there are plenty of additions to all chapters. The significant ones amongst these are,

- *Evolution of OS Functions (Chapter 9):* A new section on resource allocation and user interface functions.
- *Processes (Chapter 10):* A much enlarged section on threads.
- *Memory Management (Chapter 15):* A new section on the reuse of memory.
- *IO Organization and IO Programming (Chapter 16):* A new section on IO initiation.

I hope the readers will like the new format and compact style. As before, I look forward to comments from the readers.

D M DHAMDHERE

Preface to the Second Edition

This edition presents a more logical arrangement of topics in Systems Programming and Operating Systems than the first edition. This has been achieved by restructuring the following material into smaller chapters with specific focus:

- *Language processors:* Three new chapters on Overview of language processors, Data structures for language processors, and Scanning and parsing techniques have been added. These are followed by chapters on Assemblers, Macro processors, Compilers and interpreters, and Linkers.
- *Process management:* Process management is structured into chapters on Processes, Scheduling, Deadlocks, Process synchronization, and Interprocess communication.
- *Information management:* Information management is now organized in the form of chapters on IO organization and IO programming, File systems, and Protection and security.

Apart from this, some parts of the text have been completely rewritten and new definitions, examples, figures, sections added and exercises and bibliographies updated. New sections on user interfaces, resource instance and resource request models and distributed control algorithms have been added in the chapters on Software tools, Deadlocks and Distributed operating systems, respectively.

I hope instuctors and students will like the new look of the book. Feedback from readers, preferably by email (dmd@cse.iitb.ernet.in), are welcome. I thank my wife and family for their forbearance.

D M DHAMDHERE

Preface to the First Edition

This book has grown out of an earlier book *Introduction to System Software* published in 1986, which addressed the recommended curricula in *Systems Programming* (courses 14 of ACM curriculum 68 and CS-11 of ACM curriculum 77), and *Operating Systems and Computer Architecture* (courses CS-6, 7 of ACM curriculum 77 and SE-6, 7 of IEEE curriculum 77). The present book offers a much expanded coverage of the same subject area and also incorporates the recommendations of the ACM-IEEE joint curriculum task force (Computing Curriculum 1991). The contents have also been updated to keep pace with the developments in the field, and the changing emphasis in the teaching of these courses. An example is the way courses titled *Systems Programming* are taught today. As against the 'mostly theory' approach of the previous decade, today the emphasis is on a familiarity with the necessary theory and available software tools. The instructors of these courses have the hard task of finding instructional material on both these aspects. One of the aims of this book is to cater for this requirement through the incorporation of a large number of examples and case studies of the widely used operating systems and software tools available in the field. Treatment of the standard components of system software, viz. assemblers and loaders, is now aimed at the IBM PC. Due to the wide availability of the IBM PC, this makes it possible for students to appreciate the finer aspects in the design of these software components. Case studies of UNIX and UNIX based tools, viz. LEX and YACC, are similarly motivated.

Organization of the book

This book is organized in two parts—*Systems Programming* and *Operating Systems*. The part on Systems Programming introduces the fundamental models of the processing of an HLL program for execution on a computer system, after which separate chapters deal with different kinds of software processors, viz. assemblers, compilers, interpreters and loaders. Each chapter contains examples and case studies so as to offer a comprehensive coverage of the subject matter.

Part II of the book is devoted to an in-depth study of operating systems. The introductory chapter of this part, Chapter 7, identifies the fundamental functions and techniques common to all operating systems. Chapters 8, 9 and 10 offer a detailed treatment of the processor management, storage management and information management functions of the operating systems. These chapters contain motivating discussions, case studies and a set of exercises which would encourage a student to delve deeper into the subject area. Chapter 11 is devoted to an important area of the study of operating systems, that of *concurrent programming*. Evolution of the primitives and contemporary language features for concurrent programming is presented so as to develop an insight into the essentials of concurrent programming. A case study of a disk manager consolidates the material covered in this chapter. Chapter 12, which is on distributed operating systems, motivates the additional functionalities that de-volve on the operating system due to the distributed environment. This chapter is intended as a primer on distributed operating systems. A detailed treatment has not been possible due to space constraints.

Using this book

This book can be used for the courses on *Systems programming* (or *System software*) at the undergraduate and postgraduate levels, and for an undergraduate course on *Operating systems*. For the former, Part I of the book, together with Chapter 7 from Part II, contains the necessary material. Additionally, parts of Chapter 12 could be used as read-for-yourself material. For a course on *operating systems*, Part II of the book contains the necessary material.

In the courses based on this book, use of concurrently running design-and-implementation projects should be mandatory. Typical project topics would be the development of compilers and interpreters using LEX and YACC, concurrent programming projects, development of OS device drivers, etc.

Apart from use as a text, this book can also be used in the professional computer environment as a reference book or as a text for the enhancement of skills, for new entrants to the field as well as for software managers.

The motivation for writing this book comes from the experience in teaching various courses in the area of system software. I thank all my students for their vital contribution to this book.

DM DHAMDHERE

Contents

CHAPTER 1

Language Processors

1.1 INTRODUCTION

Language processing activities arise due to the differences between the manner in which a software designer describes the ideas concerning the behaviour of a software and the manner in which these ideas are implemented in a computer system. The designer expresses the ideas in terms related to the *application domain* of the software. To implement these ideas, their description has to be interpreted in terms related to the *execution domain* of the computer system. We use the term *semantics* to represent the rules of meaning of a domain, and the term *semantic gap* to represent the difference between the semantics of two domains. Fig. 1.1 depicts the semantic gap between the application and execution domains.

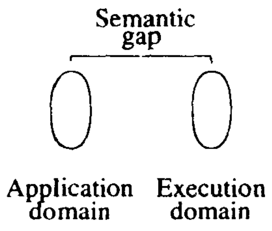


Fig. 1.1 Semantic gap

The semantic gap has many consequences, some of the important ones being large development times, large development efforts, and poor quality of software. These issues are tackled by Software engineering through the use of methodologies and programming languages (PLs). The software engineering steps aimed at the use of a PL can be grouped into

1. Specification, design and coding steps
2. PL implementation steps.

Software implementation using a PL introduces a new domain, the *PL domain*. The semantic gap between the application domain and the execution domain is bridged by the software engineering steps. The first step bridges the gap between the application and PL domains, while the second step bridges the gap between the PL and execution domains. We refer to the gap between the application and PL domains as the *specification-and-design gap* or simply the *specification gap*, and the gap between the PL and execution domains as the *execution gap* (see Fig. 1.2). The specification gap is bridged by the software development team, while the execution gap is bridged by the designer of the programming language processor, viz. a translator or an interpreter.

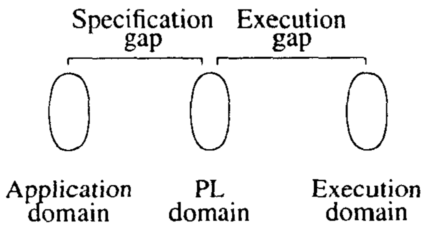


Fig. 1.2 Specification and execution gaps

It is important to note the advantages of introducing the PL domain. The gap to be bridged by the software designer is now between the application domain and the PL domain rather than between the application domain and the execution domain. This reduces the severity of the consequences of semantic gap mentioned earlier. Further, apart from bridging the gap between the PL and execution domains, the language processor provides a diagnostic capability which detects and indicates errors in its input. This helps in improving the quality of the software. (We shall discuss the diagnostic function of language processors in Chapters 3 and 6.)

We define the terms specification gap and execution gap as follows: *Specification gap* is the semantic gap between two specifications of the same task. *Execution gap* is the gap between the semantics of programs (that perform the same task) written in different programming languages. We assume that each domain has a specification language (SL). A specification written in an SL is a *program* in SL. The specification language of the PL domain is the PL itself. The specification language of the execution domain is the machine language of the computer system. We restrict the use of the term execution gap to situations where one of the two specification languages is closer to the machine language of a computer system. In other situations, the term specification gap is more appropriate.

Language processors

Definition 1.1 (Language processor) A *language processor* is a software which bridges a specification or execution gap.

We use the term *language processing* to describe the activity performed by a language processor and assume a diagnostic capability as an implicit part of any form of language processing. We refer to the program form input to a language processor as the *source program* and to its output as the *target program*. The languages in which these programs are written are called *source language* and *target language*, respectively. A language processor typically abandons generation of the target program if it detects errors in the source program.

A spectrum of language processors is defined to meet practical requirements.

1. A *language translator* bridges an execution gap to the machine language (or assembly language) of a computer system. An *assembler* is a language translator whose source language is assembly language. A *compiler* is any language translator which is not an assembler.
2. A *detranslator* bridges the same execution gap as the language translator, but *in the reverse direction.*
3. A *preprocessor* is a language processor which bridges an execution gap but is not a language translator.
4. A *language migrator* bridges the specification gap between two PLs.

Example 1.1 Figure 1.3 shows two language processors. The language processor of part (a) converts a C++ program into a C program, hence it is a preprocessor. The language processor of part (b) is a language translator for C++ since it produces a machine language program. In both cases the source program is in C++. The target programs are the C program and the machine language program, respectively.

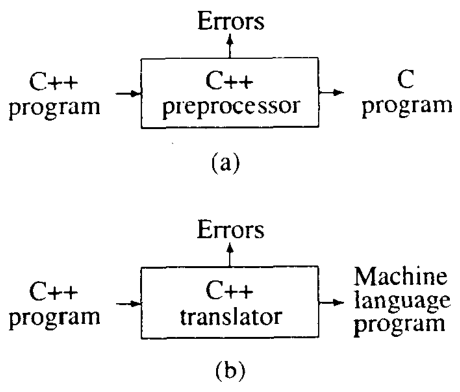


Fig. 1.3 Language processors

Interpreters

An interpreter is a language processor which bridges an execution gap without generating a machine language program. In the classification arising from Definition 1.1, the interpreter is a language translator. This leads to many similarities between translators and interpreters. From a practical viewpoint many differences also exist between translators and interpreters.

The absence of a target program implies the absence of an output interface of the interpreter. Thus the language processing activities of an interpreter cannot be separated from its program execution activities. Hence we say that an interpreter 'executes' a program written in a PL. In essence, the execution gap vanishes totally. Figure 1.4 is a schematic representation of an interpreter, wherein the interpreter domain encompasses the PL domain as well as the execution domain. Thus, the specification language of the PL domain is identical with the specification language of the interpreter domain. Since the interpreter also incorporates the execution domain, it is as if we have a computer system capable of 'understanding' the programming language. We discuss principles of interpretation in Section 1.2.2.

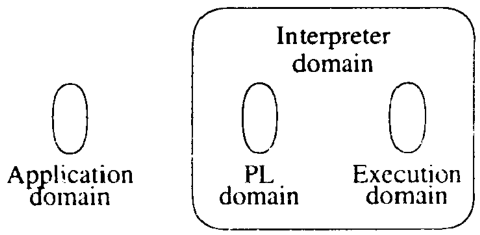


Fig. 1.4 Interpreter

Problem oriented and procedure oriented languages

The three consequences of the semantic gap mentioned at the start of this section are in fact the consequences of a specification gap. Software systems are poor in quality and require large amounts of time and effort to develop due to difficulties in bridging the specification gap. A classical solution is to develop a PL such that the PL domain is very close or identical to the application domain. PL features now directly model aspects of the application domain, which leads to a very small specification gap (see Fig. 1.5). Such PLs can only be used for specific applications, hence they are called *problem oriented languages*. They have large execution gaps, however this is acceptable because the gap is bridged by the translator or interpreter and does not concern the software designer.

A *procedure oriented language* provides general purpose facilities required in most application domains. Such a language is independent of specific application domains and results in a large specification gap which has to be bridged by an application designer.

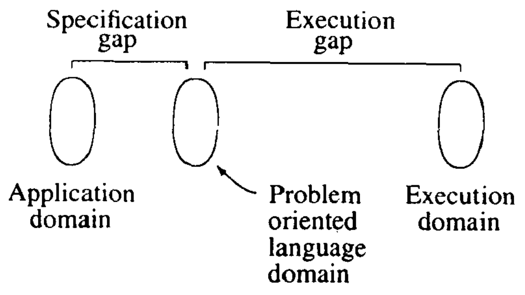


Fig. 1.5 Problem oriented language domain

1.2 LANGUAGE PROCESSING ACTIVITIES

The fundamental language processing activities can be divided into those that bridge the specification gap and those that bridge the execution gap We name these activities as

1. Program generation activities
2. Program execution activities.

A program generation activity aims at automatic generation of a program. The source language is a specification language of an application domain and the target language is typically a procedure oriented PL. A program execution activity organizes the execution of a program written in a PL on a computer system. Its source language could be a procedure oriented language or a problem oriented language.

1.2.1 Program Generation

Figure 1.6 depicts the program generation activity. The program generator is a software system which accepts the specification of a program to be generated, and generates a program in the target PL. In effect, the program generator introduces a new domain between the application and PL domains (see Fig. 1.7). We call this the *program generator domain*. The specification gap is now the gap between the application domain and the program generator domain. This gap is smaller than the gap between the application domain and the target PL domain.

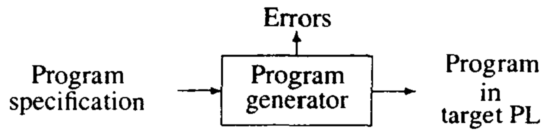


Fig. 1.6 Program generation

Reduction in the specification gap increases the reliability of the generated program. Since the generator domain is close to the application domain, it is easy for the designer or programmer to write the specification of the program to be generated.