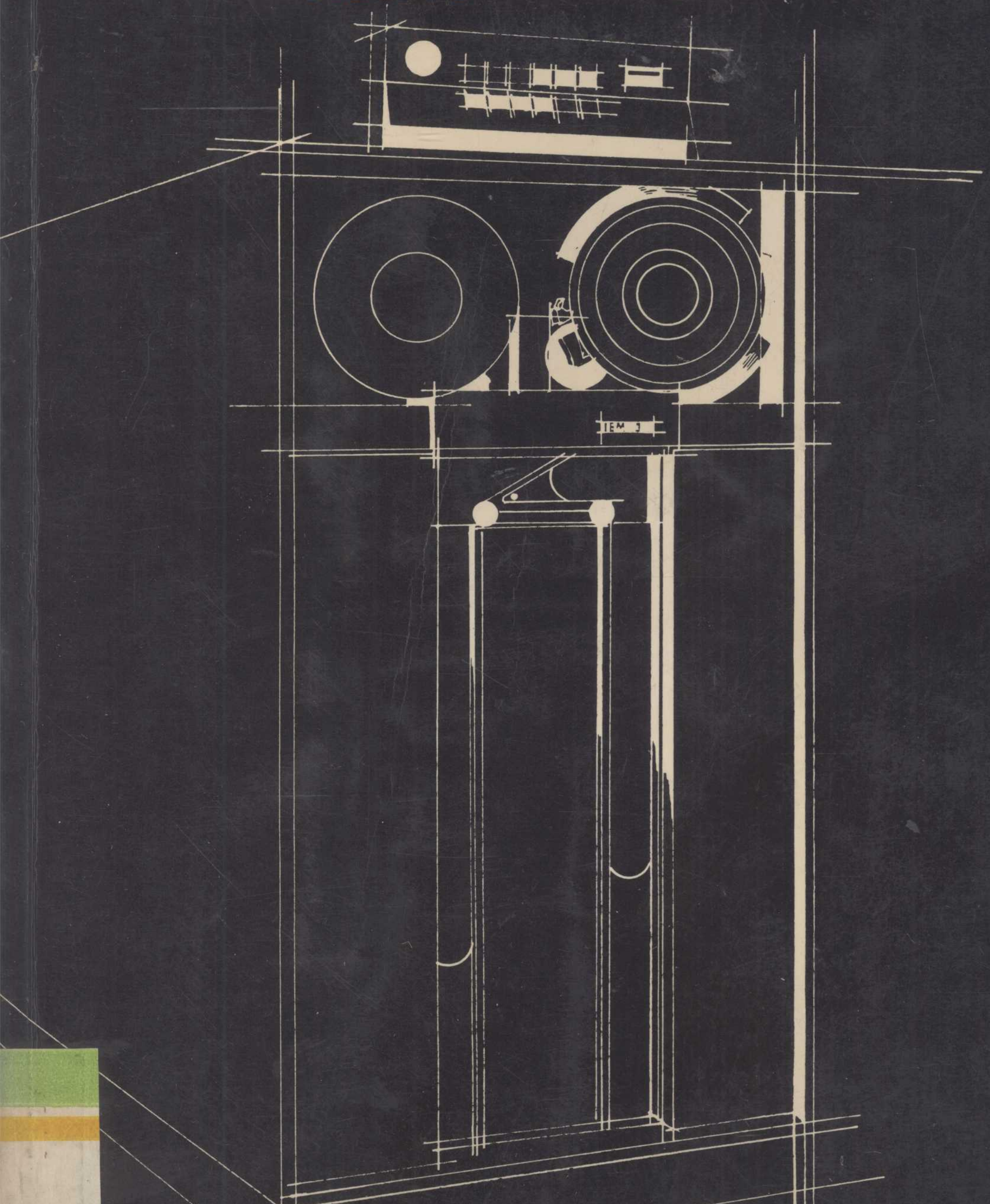
SYSTEM 360/370 JOB CONTROL LANGUAGE AND THE ACCESS METHODS

REINO HANNULA



REINO HANNULA
California Polytechnic State University, San Luis Obispo

SYSTEM 360/370 JOB CONTROL LANGUAGE AND THE ACCESS METHODS

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

Reproduced by Addison-Wesley from camera-ready copy supplied by the author.

Library of Congress Catalog Card No. 76-23986

ISBN 0-201-02755-0
BCDEFGHIJ-AL-798

To the computer science students at Cal Poly, past and present, who have taught me so much.

Preface

This work covers thoroughly the Job Control Language of System/360-370. It is also the only textbook in the field to consider the System/360-370 access methods -- BDAM, BPAM, BSAM, BISAM, QSAM, and QISAM -- in detail. The organization of the text maintains a logical consistency in that the reader is familiarized with the Job Control Language repertoire and then introduced to the access method concepts.

The book is divided into five parts:

1. an introduction to the System/360-370 operating system (Chapter 1);
2. an introduction to data sets (Chapter 2);
3. a thorough discussion of Job Control Language (Chapters 3-8);
4. a comprehensive presentation of six access methods (Chapters 9-13);
5. an example of a generation data group written in the FORTRAN Language (Chapter 14).

A special section -- the System Notebook -- provides useful information about computer science topics which, in some cases, may be completely familiar to advanced students. Any reader, however, who is not totally conversant with the subject matter in a System Notebook topic will find the information extremely useful in his or her study of Job Control Language and the access methods.

The System Notebook also introduces the student to some very useful IBM utility programs. The study of these utilities will make the reader's research in the IBM *Utility* manual very profitable.

I especially urge the students to study and to run the programs (or ones similar to it) that are given in Topics 1 and 2 of the System Notebook (Chapter 5). This ten-job-step program (which one of my students dubbed as the *Big Job Step Job*) has been extremely helpful in raising the sophistication level of the students in my JCL classes at California Polytechnic State University. These two topics are also a simple but very effective introduction to the partitioned data set organization concept.

I have taught the material presented in this text in the exact order given by the Table of Contents. An instructor will not face, however, any difficulties if he or she modifies the order of the topics to suit his or her own teaching philosophy.

There is sufficient material in this text for a semester's work. I find that in a ten-week quarter, I must omit either the generation data group discussion or the Basic Direct Access Method. My preference is to omit the latter subject since the students are intrigued and highly motivated by the discussion of the generation data group.

The students in my classes have had, generally, at least two high-level language courses (FORTRAN, COBOL, and/or PL/I) and Basic Assembler Language. These students, not required to take the JCL course, are usually highly motivated to master the intricacies of Job Control Language.

This text has been organized so that students with only a high-level language background can master the Job Control Language presented and gain greater insight into the access methods. I would, with such students, avoid a detailed discussion of Chapters 1 and 2.

Note that there are two high-level language appendices -- The *COBOL Appendix* and the *PL/I Appendix* -- which contain the text's Assembler Language programs in COBOL and PL/I, respectively. For example, the COBOL equivalent of the program contained in Figure 9.2 in the text is COBOL 9.2 in the COBOL Appendix. PL/I 2.2 contains the PL/I equivalent of the same program. There are remarks and comments in the routines in the appendices which relate the high-level program with its associated Assembler Language routine in the text. I owe this novel approach to a suggestion made by Professor William Davis of Miami University, Ohio, who reviewed my manuscript for Addison-Wesley.

I note that cards were keypunched for every program in this book from the camera-ready copy. I ran all the programs -- using these decks -- on a system 360 computer. Corrections, when necessary, were made in the camera-ready copy so that I am confident that every program in this book will execute using the code as given in the figures.

My greatest debt in this text is to my JCL students who taught me just about everything I know about Job Control Language and the Access Methods. I cannot list all of their names so I hope that I am forgiven if I inadvertently omit the name of some student who should have been mentioned.

I owe a great debt to Richard Farabaugh, now of IBM, who taught me a great deal about the Basic Direct Access Method and sundry other items. A thank you is also due to Wayne Beavers, Paul Popelka, and Richard Bergquist -- who read IBM manuals as if they were detective stories -- for their considerable assistance. The very interesting example of spanned records (data mode) in Figure 4.4 is based on Richard's suggestion. Paul suggested the inclusion of the short block retrieval program in Chapter 10 and provided other useful ideas. Dana Freiburger, an expert on generation data groups, suggested the inclusion of Chapter 14. My special thanks are also due Marc Lewis, a student in my first Assembler Language class and now the system programmer in our Computer Center, who developed the first INIT and PROLOG macros used at our University.

I must also give my thanks to Sidney Abbott, James Sturch, Kenneth Skewes, Mark Hahn, Gary Nunes, and the other twenty students in the first JCL class at Cal Poly where we began to unravel the intricacies of JCL and the access methods.

Tom Locke wrote the COBOL and FORTRAN programs equivalent to Figure 2.3 and Figures 9.1 through 9.6. Paul Popelka wrote the ISAM and BDAM PL/I routines. The COBOL ISAM and COBOL BDAM routines were written by Conard Sill and Donald Payette, respectively. Carolee Ball did almost all of the keypunching. John Burdett was always helpful with his knowledge. I commend all of these people for their excellent work.

This book, however, would not have been possible without Ms. Meri Kay Gurnee who typed the original manuscript and the camera-ready material. She also did all but three of the illustrations. Her one watchword was "Let's get everything right." I appreciate her devotion to the task.

I am, of course, solely responsible for any errors that may appear in the text. I thank the reviewers for their help in removing those that they noticed.

Finally, I wish to acknowledge my greatest debt which is to my wife, Alice Beatrice Hannula, who always said "you can." So I did.

Contents

Chapter 1

Introduction— The Operating System

The main purpose of this chapter is to present a general discussion of the components of the System/360-370 Operating System. We will emphasize, in this discussion, those elements of the Operating System which are closely related to the subject matter of this text.

Although this discussion will not be exhaustive, it will suffice to make the Job Control Language (JCL) we present more meaningful to the user. We will assume in all later chapters that the reader is conversant with the material covered here.

Table 1.1 conveniently lists the components of a System/360-370 Operating System. We suggest the reader use this table as a reference. The starred components are covered in some depth.

Our first subject is the Job Management routines.

TABLE 1.1

The Operating System Components

```
I.  Control program routines
   *1.  Job Management
    2.  Task Management
   *3.  Data Management
    4.  Recovery Management
II.  Processing Programs
    1.  Language Translators or Compilers
        a)  Algol
        b)  Assembler
        c)  COBOL
        d)  FORTRAN
        e)  PL/I
        f)  RPG
    2.  Service Programs
       *a)  Utilities
        b)  Linkage Editor
        c)  Sort/Merge
        d)  Emulators
    3.  Application Programs
    4.  User Programs
```

1.1 *THE JOB MANAGEMENT ROUTINES*

The two main components of the job management routines are the *Master Scheduler* and the *Job Scheduler*.

Both the computer operator and the Operating System use the Master Scheduler to communicate with one another; that is, the Master Scheduler analyzes the commands issued by the operator on the console and transmits messages from the Operating System to the computer operator. Commands to the Master Scheduler can also be placed in the input job stream. These commands have a very high priority and are passed to the Master Scheduler for execution as soon as they are encountered.

The Master Scheduler also initiates and terminates the three main modules of the Job Scheduler -- The *Reader/Interpreter*, the *Initiator/Terminator*, and the *Output Writer*. Since the Job Scheduler reads, interprets, and acts on the information provided by the Job Control Language, each of these modules will be covered in depth.

The Reader/Interpreter is discussed in Section 1.1. The discussions of the Initiator/Terminator and the Output Writer are in Sections 1.4 and 1.5, respectively. Table 1.2 lists the Job Scheduler functions. Note that we have divided the Job Scheduler into its three main parts in the table.

TABLE 1.2

Functions of the Job Scheduler

Reader/Interpreter
1. Reads and interprets the job control language
2. Places jobs in the input job queue

Initiator/Terminator
1. Schedules job for execution
2. Allocates I/O devices for the job
3. Initializes the actual processing of a job
4. Handles the termination of a job

Output Writer
1. Writes all program data on system output devices

There are two types of job schedulers. One is called the *Sequential Scheduler*, the other, the *Priority Scheduler*. The type of operating system generated at SYSGEN (SYStem GENeration time) determines which scheduler will be used. If PCP (*Primary Control Program*) is generated, then the Sequential Scheduler will be used. Both of the other OS (Operating System) configurations, MVT (Multiprogramming with a Varied Number of Tasks) and MFT (Multiprogramming with a Fixed Number of Tasks), utilize the Priority Scheduler. (We discuss some differences between MVT and MFT in Section 1.3.)

The main difference between the two schedulers is in how the Reader/Interpreter routine does its work. The R/I (Reader/Interpreter) of the Sequential Scheduler transfers control to the Initiator/Terminator after it (the Reader/Terminator) has finished reading the job control information for each job step. The Reader/Interpreter of the Priority Scheduler reads the input job stream until it reaches an end-of-file condition *or* until it is interrupted by the Operating System for some other reason.

In a sense the Sequential Scheduler starts the Initiator/

Terminator. Under the MVT and MFT configurations, the Reader/Interpreter cannot start (transfer control of the CPU to) the Initiator/Terminator. That routine, the I/T, is initially started by a command issued by the computer operator.

We discuss the Reader/Interpreter of both the Sequential and Priority Schedulers in some depth in the next section. We do, however, limit our discussion in subsequent sections to the routines in the Priority Scheduler.

1.2 *THE READER/INTERPRETER*

The Reader/Interpreter in the PCP environment[1] reads the input stream (usually the card reader). The job control information is scanned for syntax errors. If the syntax is correct, then the Reader/Interpreter places the job control information into the input work queue called the SYS1.SYSJOBQE[2].

If an error is detected, then the Reader/Interpreter issues a diagnostic for the erroneous statement. The scanning continues until the next SYSIN DD * statement is encountered. At that point, the input stream is flushed until the next EXEC (execute) card is encountered. And then the scanning begins again. (See Section 6.2 for a discussion of the SYSIN DD * statement.)

Even if the control statements are correct, the Reader/Interpreter in the PCP configuration will read and interpret the JCL only until program data or a new job is encountered. In either case, the R/I will transfer control of the computer to the Initiator/Terminator.

The Initiator/Terminator, when it gains control of the CPU, will perform its functions which includes reading the control information in the job queue, allocating the external storage devices requested by the job, and setting up (in main storage) the tables necessary to execute the job.

The job is executed under the supervision of the Task Management routines. When the job is terminated control is returned to the Initiator/Terminator which deactivates the data sets, releases the storage devices used by the program, and returns control to the Reader/Interpreter. This cycle repeats until all jobs are executed.

The job control information needed to execute a job[3] or a job step[4] is called a *procedure*. Since so many jobs and job steps need quite similar JCL (Job Control Language) and because a slight error in some small detail (such as a misplaced comma) is so easy to make many procedures are stored in a system library named SYS1.PROCLIB. The user can invoke a procedure from this procedure library, or

---

[1] Recall that the Sequential Scheduler is generated for the PCP environment.

[2] The SYS1.SYSJOBQE is a data set created at system generation time. It must reside on a direct access device.

[3] A job is a set of related programs. It might be considered the work between two JOB cards.

[4] A job step is a single task. The amount of work defined between two EXEC job control cards is a job step.

PROCLIB (as it is sometimes called), by specifying its name on the EXEC job control card. (See Chapter 5.)

The Reader/Interpreters of both schedulers, Sequential and Priority, merge the job control information provided by the procedure library with the JCL provided by the user in the input job stream. The user may, if he or she wishes, modify the control information in any procedure that he invokes. This is explained in a later chapter.

We pointed out earlier that, under priority scheduling, the Reader/Interpreter reads the control information of all the jobs in the input stream. The jobs are placed into *job queues* in the SYS1.SYSJOBQE (Figure 1.1) based on the job's class and priority. The class and priority are passed to the Reader/Interpreter on the JOB card.

There are fifteen possible job classes which are usually designated by the letters *A* through *O*[1]. The jobs are placed into the designated job queues on a priority basis. These priorities are numeric values which range from the lowest, 0, to the highest, 13. The priority determines the order in which jobs in the same class are loaded into the computer. If two jobs of the same class have the same priority, then the first job placed into the queue is the first job to be placed into the computer.

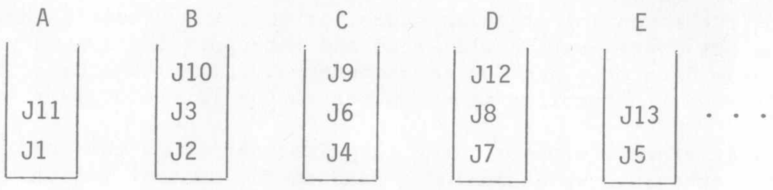


FIGURE 1.1

A SYS1.SYSJOBQE with Five Classes

The discussion in the next section, where we consider the MVT and MFT configurations of OS, will help clarify the class and priority concepts. We suggest the student review the above material after reading Section 1.3.

1.3 *MVT AND MFT*

Our subject matter in this section is not directly connected to Job Control Language. We feel, however, that, if the reader is somewhat conversant with the basic concepts of both MVT and MFT, a discussion of JCL will be a little easier to follow.

We suggest, therefore, that this section be read with care. We do not capsulate our discussion but neither do we provide an exhaustive discussion of these two configurations of OS. The IBM manuals

[1] There is also a HOLD class queue which enables a programmer to defer the execution of his or her program until specified events have occurred. See discussion of the TYPRUN keyword parameter in Chapter 4.

MFT Guide and *MVT Guide*[1] might be consulted if more information is desired.

Let us begin our discussion by defining the concept of *multiprogramming*. Our definition, just as most definitions in computer science, does not have the precision of mathematical definitions; however, it is useful and generally acceptable.

> An operating system which controls the execution of two or more concurrent tasks at one time is referred to as a multiprogramming system.

The first question that this definition should raise in our mind is, "What is a *task*"?

IBM, in the manual, *Introduction to the Operating System* (GC 28-6534), defines a *task* simply as work to be accomplished."

We find this definition quite inadequate and suggest therefore the following: A *task* is the smallest unit of work that can be performed under the supervision of the control program. We also suggest that the reader consider a task to be the amount of work defined in a single job step. Or, alternatively, the amount of work specified between two EXEC cards.

The number of tasks which may be in memory under MFT (Multiprogramming with a Fixed Number of Tasks) is said to be fixed. The computer memory is divided into *partitions*.[2] The number of partitions and the size of each partition is determined when the MFT system is generated at SYSGEN time. However, the computer operator may change the number of partitions and/or their size, if necessary, by issuing the DEFINE command.

The *nucleus*, or control program, resides in the low-order addresses of memory. The remainder of memory, which is now called *dynamic storage*, may contain up to fifteen partitions. (Compare this to the fifteen classes in the job queue.)

The minimum size for any partition is 8K, the largest may be as large as the entire area of dynamic storage. In the latter case, of course, it would mean that the MFT system has exactly one partition dedicated to user programs. Figure 1.2 depicts a division of dynamic storage into six partitions. Notice the numbering starts from zero. P0 is the first partition and P5 is the sixth.

We said, in the previous section, that the SYS1.SYSJOBQE had fifteen different job classes designated by the letters *A* to *O* inclusive. The letters themselves do not have any intrinsic meaning but the computer installation may assign attributes to each class. For example, a computer center may decide that all jobs which require more than three hours of CPU time will be in class D. A user, whose job required more than three hours of CPU time, would specify CLASS=D on his JOB card. (See Section 4.3.)

The computer operator may assign from one to three of these job classes to any one partition. Since the partitions have *dispatching*

---

[1] GC 27-6939 and GC 28-6720, respectively.

[2] A partition is a contiguous portion of memory which is dedicated to storing a single task. The task will be executed under the control of the Task Management routines.

priority[1] -- the priority which determines the order in which each job gets control of the CPU -- the assignment of classes to the partitions assigns, in effect, the dispatching priority to the job classes.

A partition's dispatching priority is determined by its relative position in memory. Partition P0 has the highest dispatching priority and Partition P15 has the lowest. Thus, if Class D is assigned to only Partition P4, all Class D jobs would have a lower priority with respect to CPU time than jobs in P0, P1, P2, and P3. (See Figure 1.2.)

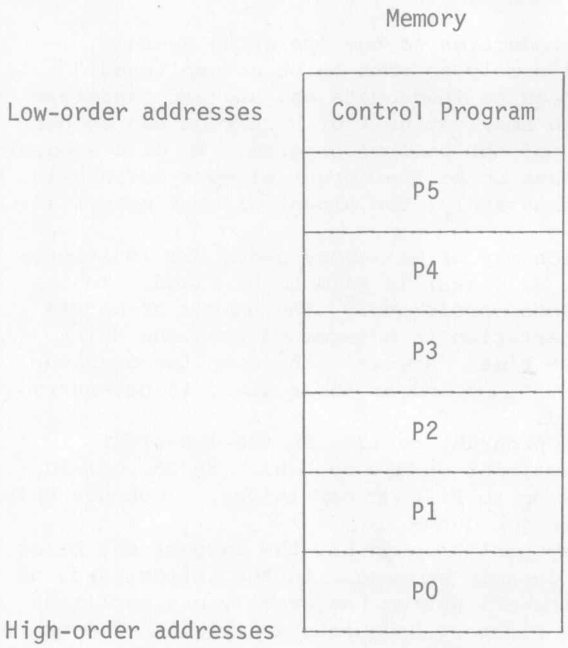


FIGURE 1.2

Partitions under MFT

The MFT configuration of the System/360-370 Operating System can handle three Reader/Interpreters, thirty-six Output Writers, and fifteen different user tasks, or job steps at the same time provided the total does not exceed fifty-two. Very few, if any, computer installations attain this maximum. The limit reached will be determined by the available resources in the computer installation.

We will now consider two differences between MFT and MVT. Figure 1.3 illustrates one of these differences. Note that the

---

[1] Do not confuse dispatching priority with the priority of jobs in the job queue. The priority in the job queue determines the order in which jobs enter the computer. A job's dispatching priority determines, with respect to the other jobs already in memory, the order in which it gains control of the CPU.